Chapter 12
Speeding Up Latent Dirichlet Allocation with Parallelization and Pipeline Strategies

Abstract Previous methods of distributed Gibbs sampling for latent Dirichlet allocation (LDA) run into either memory or communication bottleneck. To improve scalability, this chapter[†] presents two strategies: (1) parallelization—carefully assigning documents among processors based on word locality, and (2) pipelining—masking communication behind computation through a pipeline scheme. In addition, we employ a scheduling algorithm to ensure load balancing both spatially (among machines) and temporally. Experiments show that our strategies can significantly reduce the unparallelizable communication bottleneck and achieve good load balancing, and hence improve LDA's scalability.

Keywords Latent Dirichlet allocation · Pipeline processing · Data placement · Distributed systems

12.1 Introduction

Latent Dirichlet allocation (LDA) was first proposed by Blei et al. to model documents [2]. Each document is modeled as a mixture of K latent topics, where each topic, k, is a multinomial distribution $\mathbf{V}\phi_k$ over a W-word vocabulary. For any document d_j, its topic mixture $\mathbf{V}\theta_j$ is a probability distribution drawn from a Dirichlet prior with parameter α. For each ith word x_{ij} in d_j, a topic $z_{ij} = k$ is drawn from $\mathbf{V}\theta_j$, and x_{ij} is drawn from $\mathbf{V}\phi_k$. The generative process for LDA is thus given by

$$\theta_j \sim \text{Dir}(\alpha), \phi_k \sim \text{Dir}(\beta), z_{ij} = k \sim \theta_j, x_{ij} \sim \phi_k, \tag{12.1}$$

[†]

E. Y. Chang, *Foundations of Large-Scale Multimedia Information Management and Retrieval*, DOI: 10.1007/978-3-642-20429-6_12,

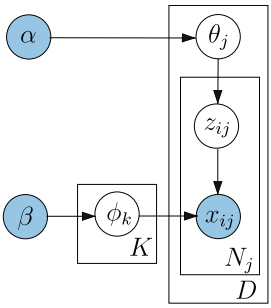

Fig. 12.1 The graphical model for LDA

where Dir(∗) denotes Dirichlet distribution. The graphical model for LDA is illustrated in Fig. 12.1, where the observed variables, i.e., words x_{ij} and hyper parameters α and β, are shaded.

The computation complexity of Gibbs sampling is K multiplied by the total number of word occurrences in the training corpus. Prior work has explored multiple alternatives for speeding up LDA, including both parallelizing LDA across multiple machines and reducing the total amount of work required to build an LDA model. Three representative distributed LDA algorithms are Dirichlet compound multinomial LDA (DCM-LDA) [3], approximate distributed LDA (AD-LDA) [4], and asynchronous distributed LDA (AS-LDA) [5], which all parallelize Gibbs sampling on distributed machines. These algorithms suffer from either high communication cost or long convergence time (an approximate method reduces communication time but increases number of Gibbs sampling iterations). In this chapter, we present PLDA+ [1], which uses distributed data-placement and pipeline strategies to reduce the communication bottleneck. The distributed data placement strategy aims to first separate CPU-bound tasks and communication-bound tasks onto two sets of machines. It then ensures that both computation and communication loads can be balanced among parallel machines. The pipeline strategy aims to mask communication time by computation time; and hence the communication bottleneck can be reduced. Experiments show that the strategies of PLDA+ can significantly improve scalability of LDA over our initial attempt at Google [6].

The rest of the chapter is organized as follows: we first present LDA and related distributed algorithms in Sect. 12.2. In Sect. 12.3 we present AD-LDA and explain how it works via a simple example. In Sect. 12.4 we analyze the bottleneck of AD-LDA. Sections 12.4.3 and 12.4.4 depict PLDA+ in details. Section 12.5 demonstrates that the speedup of PLDA+ on large-scale document collections significantly outperforms AD-LDA. In Sect. 12.6 we introduce two large-scale applications of distributed LDA. Finally, we discuss future research plans in Sect. 12.7. For the convenience of readers, we summarize the notation used in this chapter in Table 12.1.

Table 12.1 Symbols associated with LDA used in this chapter

D	Number of documents
T	Number of topics
W	Vocabulary size
N	Number of words in the corpus
x_{ij}	The ith 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
$\mathbf{V}\theta_j$	Probability of topics given document d_j
$\mathbf{V}\phi_k$	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

12.2 Related Reading

According to the generative process of LDA shown in (12.1), the full joint distribution over all parameters and variables is

$$p(\mathbf{V}x, \mathbf{V}z, \mathbf{V}\theta, \mathbf{V}\phi|\alpha, \beta) = p(\mathbf{V}\phi|\beta) \prod_{j=1}^{D} p(\mathbf{V}\theta_j|\alpha) \prod_{i=1}^{N_j} p(x_{ij}|\mathbf{V}\phi, z_{ij}) p(z_{ij}|\mathbf{V}\theta_j), \tag{12.2}$$

where $\mathbf{V}x = \{x_{ij}\}$ is the observed word occurrences in D documents, $\mathbf{V}z = \{z_{ij}\}$ is the assigned latent topics to words $\mathbf{V}x$ and N_j the number of word occurrences in document d_j. Similar to most previous work, we use symmetric Dirichlet priors in LDA for simplicity. Given the observed words $\mathbf{V}x$, the task of inference for LDA is to compute the posterior distribution of the latent topic assignments $\mathbf{V}z$, the topic mixtures of documents $\mathbf{V}\theta$, and the topics $\mathbf{V}\phi$.

Blei et al. [2] proposed using a variational expectation maximization (VEM) algorithm for obtaining maximum-likelihood estimate of Φ from V. This algorithm iteratively executes an E-step and an M-step, where the E-step infers the topic distribution of each training document, and the M-step updates model parameters using the inference result. Unfortunately, this inference is intractable, so variational Bayes is used in the E-step for approximate inference. Minka and Lafferty proposed a comparable algorithm [7], which uses another approximate inference method, expectation propagation (EP), in the E-step.

Griffiths and Steyvers [8] proposed using Gibbs sampling, a Markov-chain Monte Carlo method, to perform inference for LDA. By assuming a Dirichlet prior, β, $\mathbf{V}\phi$

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 [9], Pachinko allocation [10], and special words with background model [11]. Moreover, since the memory requirement of VEM is not nearly as scalable as that of MCMC [12], most existing distributed methods for LDA use Gibbs sampling for inference, e.g., DCM-LDA [3], AD-LDA [4], and AS-LDA [5]. In this chapter, we thus focus on Gibbs sampling for approximate inference. In Gibbs sampling, it is usual to integrate out the mixtures θ and topics ϕ and just sample the latent variables $\mathbf{z}$. The process is called *collapsing*. When performing Gibbs sampling for LDA, we maintain two matrices: word-topic count matrix C^{word} in which each element C_{wk} is the number of word w assigned to topic k, and document-topic count matrix C^{doc} in which each element C_{kj} is the number of topic k assigned to d_j document. 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

$$p(z_{ij} = k|\mathbf{z}^{\neg ij}, \mathbf{x}^{\neg ij}, x_{ij} = w, \alpha, \beta) \propto \frac{C_{wk}^{\neg ij} + \beta}{C_k^{\neg ij} + W\beta} \left(C_{kj}^{\neg ij} + \alpha\right), \tag{12.3}$$

where $\neg ij$ means that the corresponding word is excluded in the counts. Whenever z_{ij} is assigned to a new topic drawn from (12.3), C^{word}, C^{doc} and C^{topic} are updated. After enough sampling iterations to burn in the Markov chain, $\mathbf{V}\theta$ and $\mathbf{V}\phi$ can be estimated by

$$\theta_{kj} = \frac{C_{kj} + \alpha}{\sum_{k=1}^{T} C_{kj} + T\alpha}, \text{ and} \tag{12.4}$$

$$\phi_{wk} = \frac{C_{wk} + \beta}{\sum_{w=1}^{W} C_{wk} + W\beta}, \tag{12.5}$$

where θ_{kj} indicates the probability of topic k given document j, and ϕ_{wk} indicates the probability of word w given topic k. Griffiths and Steyvers conducted an empirical study of VEM, EP and Gibbs sampling and the comparison shows that Gibbs sampling converges to a known ground-truth model more rapidly than either VEM or EP [8].

12.2.1 LDA Performance Enhancement

The computation complexity of Gibbs sampling is K multiplied by the total number of word occurrences in the document collection. Prior work has explored multiple alternatives for speeding up LDA, including both parallelizing LDA across multiple processors and reducing the total amount of work required to build an LDA model. Relevant distributed methods for LDA include:

- Nallapati et al. [13] and Wolfe et al. [14] both reported distributed computing of the VEM algorithm for LDA [2].
- Mimno and McCallum proposed DCM-LDA [3], where the data sets are distributed to processors, Gibbs sampling is performed in each processor independently without any communication between processors, and finally a global clustering of the topics is performed.
- Newman et al. [4] proposed 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 in each processor is performed with the local word-topic matrix, which is only updated at the end of each iteration, it is named with *approximate* distributed LDA.
- An asynchronous distributed learning algorithm of LDA was proposed in [5], where no global synchronization step like that in [4] is required. Each processor performs a local Gibbs sampling step followed by a step of communicating with other *random* processors. We name this method as AS-LDA.

In addition to these parallelization techniques, the following optimizations can reduce LDA model learning times by reducing the total computational cost:

- Gomes et al. [15] presented an enhancement of the VEM algorithm using a bounded amount of memory.
- Porteous et al. [16] proposed a method to accelerate the computation of (12.3). The acceleration is achieved by no approximations but using the property that the topic probability vectors for document d_j, $\mathbf{V}\theta_j$, are sparse in most cases.

12.3 Approximate Distributed LDA

Before introducing PLDA+, let us review our prior implementation [6] of the AD-LDA algorithm [4]. We present the algorithm's dependency on the collective communication operation, *AllReduce*, and how to express the AD-LDA algorithm in the model of MPI. AD-LDA serves as the performance yardstick of PLDA+.

12.3.1 Parallel Gibbs Sampling and AllReduce

AD-LDA distributes D training documents over P processors, with $D_p = D/P$ documents on each processor. AD-LDA partitions document content $\mathbf{V}x = \{\mathbf{V}x_d\}_{d=1}^{D}$ into $\{\mathbf{V}x_{|1}, \ldots, \mathbf{V}x_{|P}\}$ and the corresponding topic assignments $\mathbf{V}z = \{\mathbf{V}z_d\}_{d=1}^{D}$ into $\{\mathbf{V}z_{|1}, \ldots, \mathbf{V}z_{|P}\}$, where $\mathbf{V}x_{|p}$ and $\mathbf{V}z_{|p}$ exist only on processor p. Document-topic count matrix, C^{doc}, are likewise distributed. We denote the document-topic count matrix on processor p as $C_{|p}^{\text{doc}}$. Each processor maintains its own copy of word-topic count matrix, C^{word}. Moreover, AD-LDA uses $C_{|p}^{\text{word}}$ to temporarily store word-topic counts accumulated from local documents' topic assignments on each processor.

In each Gibbs sampling iteration, each processor p updates $\mathbf{V}z_{|p}$ by sampling every $z_{ij|p} \in \mathbf{V}z_{|p}$ from the approximate posterior distribution:

$$p(z_{ij|p} = k \mid \mathbf{V}z^{\neg ij}, \mathbf{V}x^{\neg ij}, x_{ij|p} = w) \propto \frac{C_{wk}^{\neg ij} + \beta}{C_k^{\neg ij} + W\beta} \left(C_{jk|p}^{\neg ij} + \alpha\right), \quad (12.6)$$

and updates $C_{|p}^{\text{doc}}$ and $C_{|p}^{\text{word}}$ according to the new topic assignments. After each iteration, each processor recomputes word-topic counts of its local documents $C_{|p}^{\text{word}}$ and uses the AllReduce operation to reduce and broadcast the new C^{word} to all processors.

12.3.2 MPI Implementation of AD-LDA

Our AD-LDA implementation [6] uses MPI [17] to parallelize LDA learning. The MPI model supports AllReduce via an API function:

```
int MPI_Allreduce(void *sendbuf, void *recvbuf, int
  count, MPI_Datatype datatype, MPI_Op op);
```

When a *worker*, meaning a thread or a process that executes part of the parallel computing job, finishes sampling, it shares topic assignments and waits for AllReduce by invoking MPI_Allreduce, where sendbuf points to word-topic counts of its local documents: a vector of count elements with type datatype. The worker sleeps until the MPI implementation finishes AllReduce and the results are in each worker's buffer recvbuf. During the reduction process, word-topic counts vectors are aggregated element-wise by the addition operation op explained in Sect. 12.3.1.

Figure 12.2 presents the detail of MPI implementation for AD-LDA. The algorithm first attempts to load checkpoints $\mathbf{V}z_{|p}$ if a machine failure took place and the computation is in the recovery mode. The procedure then performs initialization (lines 5–9), where for each word, its topic is sampled from a uniform distribution. Next, $C_{|p}^{\text{doc}}$ and $C_{|p}^{\text{word}}$ can be computed from the histogram of $\mathbf{V}z_{|p}$ (line 11). To obtain C^{word}, the algorithm invokes MPI_Allreduce (line 12). In the Gibbs sampling iterations, each word's topic is sampled from the approximate posterior distribution (12.6) and $C_{|p}^{\text{word}}$ and $C_{|p}^{\text{doc}}$ is updated accordingly (lines 14 to 18). At the end of each iteration, the algorithm checkpoints $\mathbf{V}z_{|p}$ (line 20) and recomputes $C_{|p}^{\text{word}}$ (line 21). Using $C_{|p}^{\text{word}}$, the algorithm perform global MPI_AllReduce to obtain up-to-date C^{word} for the next iteration (line 22). After a sufficient number of iterations, the "converged" LDA model is outputted by the master (line 24).

Different MPI implementations may use different AllReduce algorithms. The state-of-the-art is the recursive doubling and halving (RDH) algorithm presented in [17], which was used by many MPI implementations including the well known MPICH2. RDH includes two phases: *Reduce-scatter* and *All-gather*. Each phase runs

1: **if** there is a checkpoint **then**
2: $t \leftarrow$ The number of iterations already done
3: Load $\mathbf{V}z_{|p}$ from the checkpoint
4: **end if**
5: $t \leftarrow 0$
6: Load documents on current processor p into $\mathbf{V}x_{|p}$
7: **for** each word $x_{ij|p} \in \mathbf{V}x_{|p}$ **do**
8: Draw a sample k from uniform distribution $U(1,K)$
9: $z_{ij|p} \leftarrow k$
10: **end for**
11: Compute $C^{doc}_{|p}$ and $C^{word}_{|p}$
12: `MPI_AllReduce(`$C^{word}_{|p}$`, `C^{word}`, `$W \times T$`, ''float-number'', ''sum'')`
13: **for** ; $t <$ iteration-num; $t \leftarrow t+1$ **do**
14: **for** each word $x_{ij|p} \in \mathbf{V}x_{|p}$ **do**
15: $C^{doc}_{j,z_{ij}|p} \leftarrow C^{doc}_{j,z_{ij}|p} - 1, C^{word}_{x_{ij},z_{ij}} \leftarrow C^{word}_{x_{ij},z_{ij}} - 1$
16: $z_{ij} \leftarrow$ draw new sample from (12.6), given C^{word} and $C^{doc}_{j|p}$
17: $C^{doc}_{j,z_{ij}|p} \leftarrow C^{doc}_{j,z_{ij}|p} + 1, C^{word}_{x_{ij},z_{ij}} \leftarrow C^{word}_{x_{ij},z_{ij}} + 1$
18: **end for**
19: **end for**
20: Checkpoint $\mathbf{V}z_{|p}$
21: Recompute $C^{word}_{|p}$
22: `MPI_AllReduce(`$C^{word}_{|p}$`, `C^{word}`, `$W \times T$`, ''float-number'', ''sum'')`
23: **if** this is the master worker **then**
24: Output C^{word}
25: **end if**

Fig. 12.2 The MPI implementation of AD-LDA

a recursive algorithm, and in each recursion level, workers are grouped into pairs and exchange data in both directions. This algorithm is particularly efficient when the number of workers is a power of two, because no worker would be idle during communication.

RDH provides no facilities for fault recovery. In order to provide fault-recovery capability in AD-LDA, the worker state can be check-pointed before AllReduce. This ensures that when one or more processors fail in an iteration, the algorithm can roll back all workers to the end of the most recent succeeded iteration, and restart the failed iteration. The checkpointing code is executed immediately before the invocation of MPI_Allreduce in AD-LDA. In practice, only $\mathbf{V}z_{|p}$ is flushed onto the disk, because $\mathbf{V}x_{|p}$ can be reloaded from data set, $C^{\mathrm{doc}}_{|p}$ and C^{word} can also be recovered from the histogram of $\mathbf{V}z_{|p}$. The recovery code is at the beginning of AD-LDA: if there is a checkpoint on the disk, load it; otherwise perform random initialization.

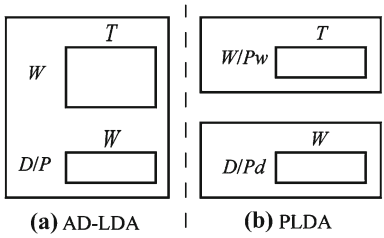

Fig. 12.3 The assignments of documents and word-topic count matrix for AD-LDA and PLDA+

12.4 PLDA+

To further speed up AD-LDA [4], PLDA+ algorithm employs distributed data placement and pipeline processing strategies.

12.4.1 Reduce Bottleneck of AD-LDA

As presented in the previous section, in our AD-LDA implementation [6], D documents are distributed over P processors with approximately D/P documents on each processor. This is shown with a D/P-W matrix in Fig. 12.3a, where W indicates the vocabulary of the collection of documents. The word-topic count matrix is also distributed, with each processor keeping a local copy, which is the W-K matrix in the figure.

In AD-LDA, after each iteration of Gibbs sampling, local word-topic counts on each machine 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 machine has completed its work. To avoid unnecessary wait, AS-LDA does not perform global synchronization like AD-LDA. AS-LDA only synchronizes word-topic counts with its neighbors. However, since word-topic counts can be outdated, the sampling process may take a larger number of iterations than that AD-LDA takes to converge. Figure 12.4 illustrates the spread patterns of the updated topic distribution of a word from one processor to the others. AD-LDA has to synchronize all word updates after one full Gibbs sampling iteration, whereas AS-LDA performs updates only with a small subset of processors. The memory requirement of both AD-LDA and AS-LDA is $O(KW)$, since the whole word-topic matrix is maintained on all machines.

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

- These methods have to maintain all word-topic counts in memory of each processor; and
- These methods have to send and receive the entire word-topic matrix between processors for updates.

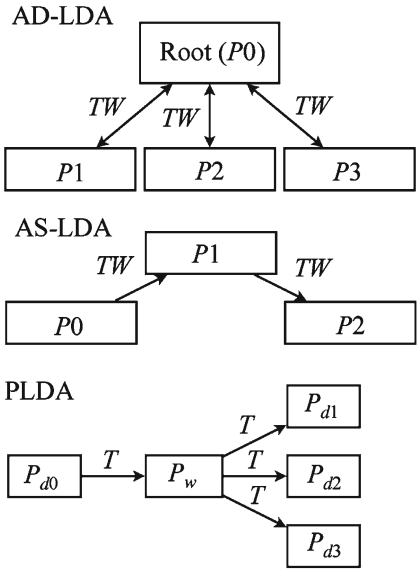

Fig. 12.4 The spread patterns of the updated topic distribution of a word from one processor for AD-LDA, AS-LDA and PLDA+

For the former characteristic, suppose we want to estimate a $\mathbf{V}\phi$ with W words and K topics from a large-scale data set. When either W or K is large to a certain extent, the memory requirement will exceed that available on a typical processor. Due to the bottleneck of latency and transfer-rate of hard disks, it is not practical to maintain the word-topic counts on hard disks. This characteristic makes the existing distributed methods face a significant challenge in terms of memory scalability. For the latter characteristic, the communication bottleneck caps the room for speeding up the algorithm. This communication bottleneck will only acerbate over years as a study of high performance computing [18] shows that floating-point instructions improve speed historically at 59% a year, but inter-processor bandwidth improves 26% a year, and inter-processor latency improves only 15% a year.

12.4.2 Framework of PLDA+

To address the increasing communication bottleneck, PLDA+ uses an enhanced distributed method for LDA. In addition to partitioning documents, PLDA+ also partitions the word-topic count matrix and distributes them to several processors. Thus, processors are divided into two types: one maintains documents and document-topic matrix to perform Gibbs sampling (P_d processors, and the other stores and maintains word-topic count matrix (P_w processors). During each iteration of Gibbs sampling, a P_d processor assigns a new topic to a word in a document in three steps:

1. Fetching the word's topic distribution from a P_w processor,

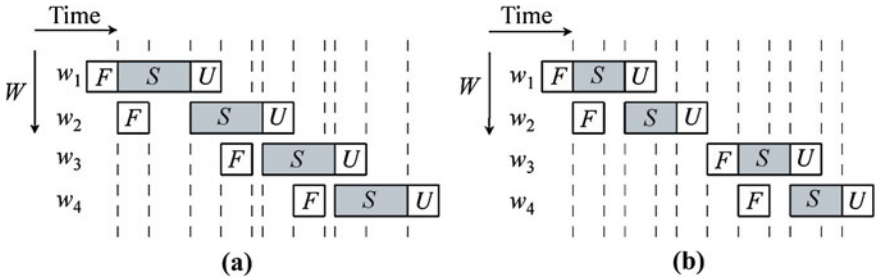

Fig. 12.5 Pipeline-based Gibbs sampling in PLDA+ (*Top*: $t_s > t_f + t_u$. *Bottom*: $t_s < t_f + t_u$)

2. Performing Gibbs sampling at the P_d processor and assigning a new topic to the word, and
3. Updating all P_w processors maintaining that word.

There are two reasons to divide processors into two groups. First, the communication bottleneck can be halved on the CPU-bound processors. This way, not only the communication time on P_w processors is cut into about one half, the reduced IO time can also be masked by the computation time much easily. Second, by separating two tasks onto two sets of machines, load balancing can be more flexibly performed.

Besides improving parallelization, PLDA+ employs pipeline processing. The pipeline technique has been used in many applications to increase throughput, such as the instruction pipeline in modern CPUs [19] and in graphics processors [20]. Although pipeline does not decrease the time for a job to be processed, it can efficiently improve throughput by overlapping IOs with computation. Figure 12.5 illustrates the Pipeline-based Gibbs Sampling for four words, i.e., w_1, w_2, w_3 and w_4, where F indicates the fetching operation, U indicates the updating operation, and S the Gibbs sampling operation. In this figure, the top chart demonstrates the case when $t_s > t_f + t_u$, and the bottom chart the case when $t_s < t_f + t_u$, where t_s, t_f and t_u denote the time of Gibbs sampling, fetching topic distribution, and updating topic distribution, respectively.

On the top chart of Fig. 12.5, PLDA+ begins by fetching the topic distribution of w_1. Then it begins Gibbs sampling on w_1, and at the same time, it fetches the topic distribution of w_2. After it has finished Gibbs sampling for w_1, it updates the topic distribution of w_1 on P_w processors. When $t_s > t_f + t_u$, PLDA+ can begin Gibbs sampling on w_2 immediately after it has completed that for w_1. Total ideal time for PLDA+ to process W words is $Wt_s + t_f + t_u$.

The bottom chart of Fig. 12.5 shows a suboptimal scenario where the IO time cannot be entirely masked. PLDA+ is not able to begin Gibbs Sampling for w_3 until after some communication delay. The example shows that in order to mask communication, the tasks must be scheduled to ensure as much as possible that $t_s > t_f + t_u$. There are two important scheduling considerations:

1. Word bundling. To ensure t_s to be sufficiently long to mask IOs, Gibbs sampling can be performed on a group of words.
2. Low latency IO scheduling. IOs must be scheduled in such a way that a CPU-bound task is minimally delayed by a fetch operation.

Since each round of Gibbs sampling can be performed in any word order, it makes word bundling flexible. First, rather than processing one document after another, PLDA+ performs Gibbs sampling according to a word order. A word that occurs several times on the documents at a node can be process in a loop. Moreover, for words that do not occur frequently, they can be bundled with frequently-occurred words to ensure that t_s is sufficiently long. In fact, if one can estimate $t_f + t_u$, one can decide how many word-occurrences to process in each Gibb Sampling batch. The remaining challenge is that one ought to ensure that $t_f + t_u$ can indeed be shorter than t_s. If a fetch cannot be completed by the time when the last Gibbs sampling task has completed, the wait time adds to the bottleneck, and hence hampers speedup.

To perform Gibbs sampling word by word, PLDA+ builds word indexes to documents on each P_d processor. Words are organized in a *circular queue* as shown on the top of Fig. 12.6. Gibbs sampling is performed by going around the circular queue. To avoid concurrent access to the same words, different processes are scheduled to begin at a different position of the queue. For example, Fig. 12.6 shows four P_d processors, P_{d1}, P_{d2}, P_{d3} and P_{d4} start their first word from w_1, w_3, w_5 and w_7, respectively. To ensure that this scheduling algorithm works, PLDA+ must distribute the word-topic matrix also in a circular fashion on P_w machines. 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. This makes PLDA+ more advantageous over an asynchronous scheme such as AS-LDA [5], which may miss updates. The detailed description of word placement is presented in Sect. 12.4.3.1.

12.4.3 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 machines according to words. After allocation, each P_w processor keeps approximately W/P_w words with their topic distributions. Figure 12.7 depicts the word-topic matrix distribution process to P_w machines.

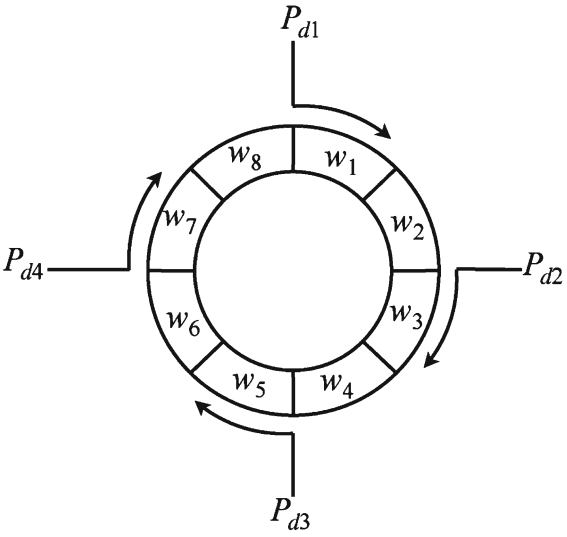

Fig. 12.6 Vocabulary circular queue in PLDA+

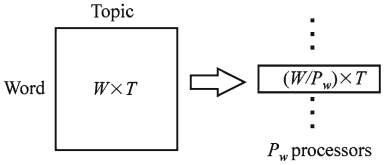

Fig. 12.7 The construction of word-topic matrix in P_w processors

12.4.3.1 Word Placement Over P_w Processors

The goal of word allocation is to ensure *spatial* load balancing. To balance load, one would like to make sure that all nodes receive about the same number of work requests in a round of Gibbs sampling.

For bookkeeping, PLDA+ maintains two data structures. First, for each word, it records how many P_d processors on which that word resides. Form W words, PLDA+ maintains a P_d vector $\mathbf{m} = (m_1, m_2, \ldots, m_W)$. The second data structure keeps track of each P_w processor's workload, or the number of word occurrences on that processor. The workload vector is denoted as $\mathbf{l} = (l_1, l_2, \ldots, l_{P_w})$.

A simple placement method is to place words independently and uniformly at random onto P_w processors. This method is referred to as *random word allocation*. Unfortunately, this random placement method may cause load imbalance among P_w processors in high probability. To balance workload, PLDA+ uses the *weighted round-robin* method for word placement. It first sorts words in *decreasing* order by their weights, and then picks the word with the largest weight from the vocabulary and assigns to a processor in a round-robin fashion. 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 [21].

12.4.3.2 Processing Requests from P_d Processors

Each P_w processor handles all requests related to the words it is responsible for maintaining. After allocating words with their topic distributions over P_w processors, P_w processors begin to receive and respond the requests from P_d processors. A P_w processor pw first builds its responsible word-topic count matrix $C^{\text{word}}_{|pw}$ by receiving initial word-topic counts from all P_d processors. Then, that P_w processor pw begins to process requests from P_d processors. PLDA+ defines three types of requests (communications):

- *fetch*(w, pw, pd). Node pw requests for fetching topic distribution of word w from a P_d processor pd. For the request, the P_w processor pw retrieves the topic distribution $\phi_w^{(pw)}$, which will be used by the pd node as $n_{wk}^{\neg ij}$ in (12.3) for performing Gibbs sampling.
- *update*(w, $\mathbf{u}$, pw, pd). Node pw updates topic distribution for word w using $\mathbf{u}$ after receiving the information from node pd.
- *fetch*(pw, pd). Node pw requests for all topic counts from node pd. The P_w Processor pw requires the data from pd to sum up the topic distributions of all words on pw in vector $\mathbf{n}^{(pw)} = (n_k^{(pw)}, k = 1, \ldots, T)$, which will be used as $n_k^{\neg ij}$ in (12.3) for performing Gibbs Sampling.

12.4.4 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 inverted index for documents on each P_d processor.
2. It groups the words in vocabulary into *bundles* for Gibbs Sampling and IO requests.
3. It schedules word bundles to minimize communication bottleneck.
4. Finally, it performs pipelined Gibbs sampling iteratively until the terminate condition is met.

In the following, we present these four steps in details.

12.4.4.1 Document Allocation and Building Inverted Index

Before performing Gibbs sampling, D documents must be distributed onto P_d processors. The goal of document allocation is to achieve good CPU load balance among P_d processors. AD-LDA may suffer from imbalanced load problem 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 without the synchronization requirement. In other

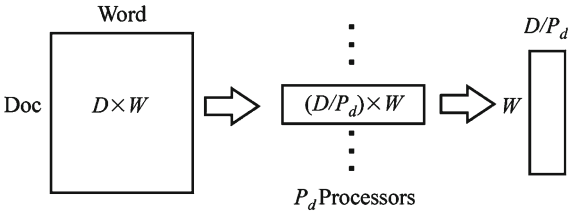

Fig. 12.8 The construction of data structure in P_d processors

words, a processor that completes its work early can start its next round of sampling without having to wait for stragglers. Dealing with stragglers is a critical issue in distributed computing. PLDA+ tackles this problem through both static allocation and dynamic migration. PLDA+ first allocates words to nodes in a balanced fashion. Each P_d processor hosts approximate D/P_d documents. The time complexity of this allocation step is $O(D)$. After documents have been distributed, we build inverted index for documents on each P_d processor. The construction process is demonstrated in Fig. 12.8. If a node is always a straggler due to run-time load imbalance or hardware configuration, the data on that node can be split and migrated onto additional nodes to eliminate stragglers.

Using inverted index, each time after a P_d processor has fetched the topic distribution of a word w, that processor performs Gibbs sampling for all instances of w on that node. After that, the processor (or node) sends back the updated information to the corresponding P_w processor. The clear benefit is that for multiple occurrences of a word on a node, PLDA+ requires to perform only two communications: one fetch and one update, and substantially reducing communication cost. The index structure for each word w is:

$$w \rightarrow \{(d_1, z_1), (d_1, z_2), (d_2, z_1), \ldots\}, \tag{12.7}$$

in which, w occurs in document d_1 twice and there are two entries. In implementation, to save memory, all occurrences of w in d_1 can be recorded in one entry, $(d_1, \{z_1, z_2\})$.

12.4.4.2 Word Bundle

Bundling words is to prevent the situation that too short the duration of Gibbs samplings cannot mask a communication IO. Use an extreme example: a word appears only once in one document on a node. Performing Gibbs sampling on that word takes a much shorter time than the time required to fetch and update the word-topic matrix. The remedy is intuitive: combining a few words into a bundle so that the IO time can be masked by the longer duration of Gibbs sampling time.

To bundle words, each P_d processor groups words in sets, each matches words on a P_w processor. For each word set, words are sorted into a list according to their occurrence times in descending order. Then, words are picked from both ends of the list to form bundles. Each time a P_d node sends a request to a P_w node to fetch topic

distributions for words in a bundle. The size of a bundle should be large enough so that the time to perform Gibbs sampling on a bundle is longer than the time to fetch the bundle from a P_w node.

12.4.4.3 Pipelined Gibbs Sampling

The core step of PLDA+ is the pipelined Gibbs sampling. As shown in (12.3), 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 j is maintained by P_d processors. While the up-to-date topic distribution C_w^{word} is maintained by a P_w processor, global topic count C^{topic} should be collected over all P_w processors. Therefore, before assigning a new topic for 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 w. Then the P_d processor returns the updated topic distribution of word w to the responsible P_w processor.

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

1. Fetch overall topic counts for Gibbs sampling.
2. Select F word bundles and put them in thread pool tp to fetch words' topic distributions. Once a request is responded from P_w processors, the returned topic distributions are put in a wait queue Q_{pd}.
3. Pick words' topic distributions from Q_{pd} to perform Gibbs Sampling.
4. After Gibbs sampling, put the updated topic distributions in 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.
7. If the termination condition has not met, go to Step 3 to start Gibbs sampling for other words.

In Step 1, processor pd fetches overall topic distributions C^{topic}. In this step, pd sends requests $fetch(pw, pd)$ to each P_w processor $pw = 1, 2, \ldots, P_w$. The requests are returned with $(C_{|pw}^{\text{topic}}, pw = 1, 2, \ldots, P_w)$, and pd thus gets C^{topic} by sum overall topic counts from each P_w processors:

$$C^{\text{topic}} \leftarrow \sum_{pw} C_{|pw}^{\text{topic}}. \tag{12.8}$$

Since 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 responded with latency. Thus, once a response is returned, it can start Gibbs sampling immediately. Here, we mention the pre-fetch number of requests as F. In PLDA+, F should be properly set to make sure that the wait queue Q_{pd} always has returned topic distributions of words waiting for Gibbs Sampling.

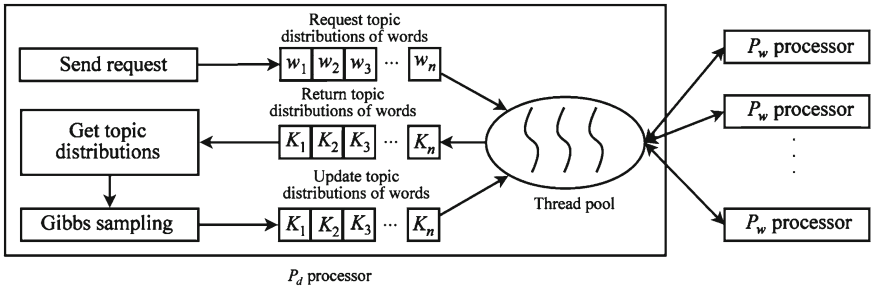

Fig. 12.9 PLDA+ Gibbs sampling

If not, Gibbs sampling is stalled by communication, which is considered a part of communication time 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 topic distributions of each word on each P_w processor. Fortunately, as indicated by the results of AD-LDA, topic assignments in Gibbs Sampling is not sensitive to the values of overall topic counts. Thus PLDA+ reduces the frequency of fetching overall topic counts to improve the efficiency of P_w processors. Therefore, in Step 6, PLDA+ does not fetch overall topic counts frequently. Experimental results show that fetching new overall topic counts only after performing one pass of Gibbs sampling can obtain the same learning quality compared to LDA and AD-LDA.

Figure 12.9 summarizes a P_d node's interprocess communication with multiple P_w nodes. The figure shows a key reason for PLDA+ to reduce communication bottleneck: that a P_d node of PLDA+ commuicates with multiple P_w nodes, rather than that multiple P_d nodes of AD-LDA communicate with one master P_w node. Furthermore, the thread pool on P_d nodes enables pre-fetching, and thereby allows communication to be masked by computation working on completed requests.

12.4.5 Straggler Handling

So far, both presented data placement and scheduling schemes of PLDA+ are *static*. Static placement and scheduling cannot guarantee run-time load balancing. Run-time imbalanced workload can be caused by at least three reasons:

1. *Uneven hardware configuration*. Not all nodes are equally configured. In a realistic distributed environment, not all computer nodes are equipped with exactly the same class of processors, memory, and disks. Also, not all nodes are equally distanced. Computation on and communication with different nodes can thus take different amount of time to complete.

2. *Resource contention*. Distributed data centers must deal with a large number of simultaneous computation tasks. It is impossible to ask all nodes to be in a quiesce mode when PLDA+ is being executed. Therefore, PLDA+ can be slowed down by tass competing for resources.
3. *Failures*. When a large number of nodes are involved, the probability of failure becomes non-negligible. When a processor or a router fails, no static scheme can continue ensuring balanced workload among all nodes.

PLDA+ deals with run-time dynamics by employing two simple approaches. First, PLDA+ uses a reset and timeout scheme. When a P_w node notices that the number of requests in its work queue has reached a threshold, it informs all P_d nodes to reset their pointers into the circular queue depicted in Fig. 12.6. In each request, the P_d node also registers a deadline. When the deadline has expired, the P_w node discards that request and proceeds to processing the next request. Occasionally missing a round of Gibbs Sampling does not affect overall performance due to the stochastic nature of Gibbs sampling.

If a P_w node has missed too many request deadlines, then PLDA+ replicates that node to balance workload. For the details of a data replication scheme that can guarantee balanced workload in probability, please consult our previous work in [22].

12.4.6 Parameters and Complexity

In this section, we discuss the parameters that may influence the performance of PLDA+. We also analyze the complexity of PLDA+ compared to other distributed methods represented by AD-LDA.

12.4.6.1 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, $\gamma = \frac{P_w}{P_d}$. The larger the value of γ, the more processors serve as P_w, and hence the average time of communication at P_d processors decreases. At the same time, the average time of Gibbs sampling will increase due to less processors are used to perform that CPU-bound task. A good system design must balance the number of P_w and P_d processors to (1) make both computation and communication time low, and (2) ensure that communication is short enough to be masked by computation. This parameter can be derived once the average time for Gibbs sampling and communication of the word-topic matrix is known. Suppose the total time of Gibbs sampling for the whole data set is T_s, the communication time of 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 topic distributions of words can be simultaneously transferred to P_w processors, and thus transfer time will be T_t/P_w. To make sure the sampling time

can overlap the fetching and updating process, PLDA+ thus must make sure that

$$\frac{T_s}{P_d} > \frac{2T_t}{P_w}. \tag{12.9}$$

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 $\bar{t}_u$ are the average fetching and update time for a word, we can get

$$\gamma = \frac{P_w}{P_d} > \frac{\bar{t}_f + \bar{t}_u}{\bar{t}_s}, \tag{12.10}$$

where $\bar{t}_f, \bar{t}_u$ and $\bar{t}_s$ can be obtained by performing PLDA+ on a small data set and then empirically set a appropriate γ value. Under the computing environment of our experiments, we empirically set $\gamma = 3/5$.

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

The third parameter is the number of requests F for pre-fetching topic distributions before performing Gibbs sampling on P_d processors. This parameter is dependent on R.

The last parameter is the maximum interval $inter_{max}$ for fetching overall topic counts from all P_w processors during Gibbs Sampling of P_d processors. This parameter influences the quality of PLDA+. Experiments show that in order to learn LDA models with similar quality to AD-LDA and LDA, $inter_{max}$ should be set to 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.

12.4.6.2 Complexity

Table 12.2 summarizes the complexity of P_d processors and P_w processors in both time and space. For comparison, the table also lists the complexity of LDA and AD-LDA. We assume $P = P_w + P_d$ when comparing PLDA+ with AD-LDA.

Finally, let us analyze the speedup efficiency of PLDA+. Suppose $\delta \to 0$ and $\gamma = \frac{P_w}{P_d}$ for PLDA+, the ideal parallel efficiency will be always:

$$\text{speedup efficiency} = \frac{S/P}{S/P_d} = \frac{P_d}{P} = \frac{1}{1+\gamma}, \tag{12.11}$$

where S denotes the running time of LDA on a single machine, 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.

Table 12.2 Algorithm complexity

Method	Time complexity	Space complexity
LDA	NT	$T(D+W)+N$
AD-LDA	$\frac{NT}{P} + TW \log P$	$\frac{(N+TD)}{P} + TW$
PLDA+ $-P_d$	$\frac{NT}{P_d} + \delta$	$\frac{(N+TD)}{P_d}$
PLDA+ $-P_w$	–	$\frac{TW}{P_w}$

Table 12.3 Detailed information of data sets

	NIPS	Dianping	Wiki-20T	Wiki-200T
D_{train}	1,540	113,754	2,122,618	2,122,618
W	11,909	27,752	20,000	200,000
N	1,260,732	3,625,275	447,004,756	486,904,674
D_{test}	200	1,000	–	–

12.5 Experimental Results

This section compares the performance of PLDA+ and AD-LDA. The comparisons help understand benefits of data placement and pipeline processing strategies.

12.5.1 Datasets and Experiment Environment

We used four datasets shown in Table 12.3 to conduct experiments. The NIPS dataset consists of scientific articles appeared at NIPS conferences. Dianping dataset consists of restaurant reviews from dianping.com. NIPS and Dianping datasets are both relatively small, and we used them to carry out training-quality assessment. Two Wikipedia datasets were collected from English Wikipedia articles of the March 2008 snapshot from en.wikipedia.org. By setting the size of vocabulary to 20,000 and 200,000, respectively, the two Wikipedia datasets are named Wiki-20T and Wiki-200T. These two large datasets were used for testing scalability of PLDA+. The experiment environment was run on distributed machines with 2,048 processors, each with a 2 HZ CPU, 3 GB memory, and disk allocation of 100 GB.

12.5.2 Perplexity

We used *test set perplexity* to measure the quality of LDA models learned by various distributed methods for LDA. Perplexity is a common way of evaluating language models in natural language processing, computed as:

$$\text{Perp}(\mathbf{x}^{\text{test}}) = \exp\left(-\frac{1}{N^{\text{test}}} \log p(\mathbf{x}^{\text{test}})\right), \tag{12.12}$$

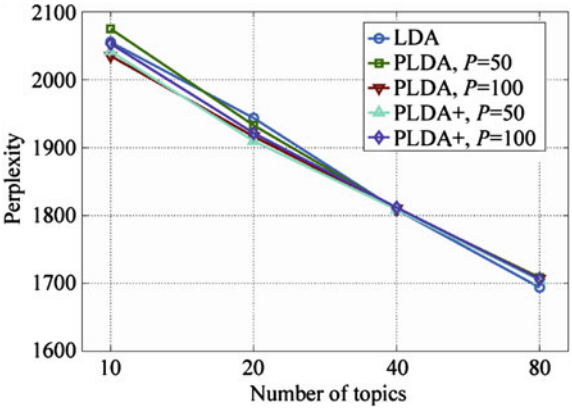

Fig. 12.10 Test perplexity on NIPS versus # topics T when the number of iterations is 400 (See color insert)

where $\mathbf{x}^{\text{test}}$ denotes test set. A lower test 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 θ_j was learned using the fold-in part, and the log probability of the test words was computed using this mixture. This ensures the test words were not used in estimating model parameters. The perplexity computation follows the standard way of averaging over multiple chains when making predictions with LDA models trained via Gibbs sampling as shown in [8]. For PLDA+ and LDA, the test perplexity was computed using $S=40$ samples from the posteriors of 40 independent chains using:

$$\log p(\mathbf{x}^{\text{test}}) = \sum_{j,w} n_{jw}^{\text{test}} \log \frac{1}{S} \sum_{k} \theta_{kj}^{S} \phi_{wk}^{S}, \tag{12.13}$$

where

$$\theta_{kj} = \frac{C_{kj}^{S} + \alpha}{\sum_{k=1}^{T} C_{kj}^{S} + T\alpha}, \quad \phi_{wk} = \frac{C_{wk}^{S} + \beta}{\sum_{w=1}^{W} C_{wk}^{S} + W\beta}. \tag{12.14}$$

To compare the quality of PLDA+ to single-machine LDA and distributed AD-LDA, we computed the test perplexity for all methods after each iteration of Gibbs sampling going through a round of whole vocabulary. The test perplexities on NIPS with the number of topics $K=10$, 20, 40, 80, and Dianping with $K=8$, 16, 32, 64 are shown in Figs. 12.10 and 12.11, respectively. (Since we concerned only about training quality, the number of machines used in this experiment may not be relevant.)

From both figures we can see that the quality of PLDA+ is similar to single-machine LDA and distributed AD-LDA. Thus, we can conclude that PLDA+ can train as good a model as traditional LDA methods.

Figures 12.12 and 12.13 show the convergence of test perplexity versus # of iteration for LDA, AD-LDA and PLDA+ on NIPS and Dianping with different number of

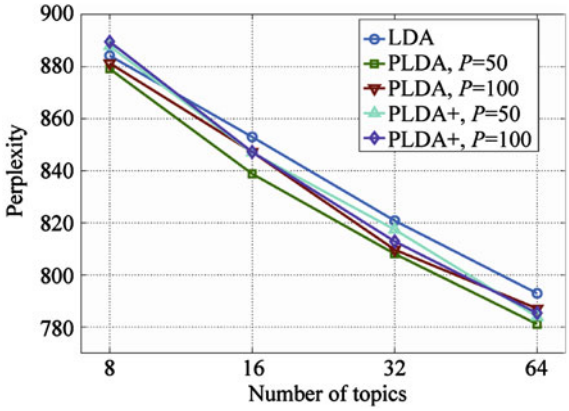

Fig. 12.11 Test perplexity on Dianping versus # topics T when the number of iterations is 400 (See color insert)

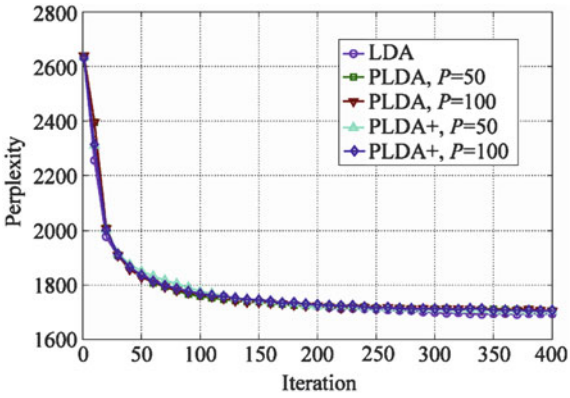

Fig. 12.12 Convergence of test perplexity versus iteration on NIPS with $T=80$ (See color insert)

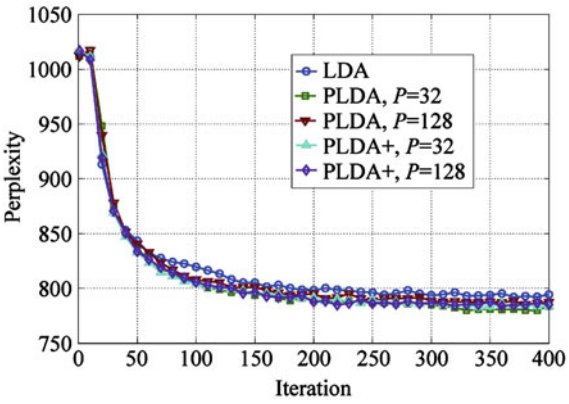

Fig. 12.13 Convergence of test perplexity versus iteration on Dianping with $T=64$ (See color insert)

processors. (The parameters were set as depicted in Sect. 12.5.2.) The figures show the convergence rate of PLDA+ is virtually identical to LDA and AD-LDA.

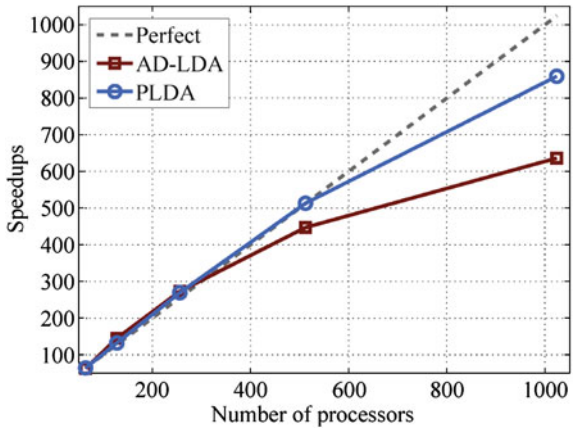

Fig. 12.14 Parallel speedup results for 64 to 1,024 processors on Wiki-20T (see color insert)

12.5.3 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+ comparing to AD-LDA. We used Wiki-20T and Wiki-200T for speedup experiments. By setting the number of topics $T = 1{,}000$, we ran PLDA+ and AD-LDA 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. For PLDA+, the ratio of $P_w P_d$ was empirically set to $\gamma = 0.6$ according to the unit sampling time and transferring time. The number of threads in a thread pool is 50, which is sufficient to handle the peak load. As analyzed in Sect. 12.4.6.2, the ideal speedup efficiency of PLDA+ is $\frac{1}{1+\gamma} = 0.625$.

Figure 12.14 compares speedup performance. The speedup was computed relative to the time per iteration when using $P = 64$ processors, because it was not possible to run the algorithms on a smaller number of processors due to memory limitations. We assumed that the speedup on $P = 64$ to be 64, and then extrapolated on that basis. From the figure, we can observe that when P increases, PLDA+ simply achieves much better speedup than AD-LDA, thanks to the much reduced communication bottleneck of PLDA+.

Figure 12.15 compares the ratio of communication time over computation time when $P = 1{,}024$. The communication time of AD-LDA is 13.38 s, much longer than that of PLDA+'s 3.68 s. The communication time of AD-LDA is about the same as its computation time at $P = 512$.

From the results, we can conclude that: (1) when word-topic matrix is not large, PLDA+ performs similarly to AD-LDA, and when the number of processors increases to large enough (e.g., $P = 512$), PLDA+ begins to achieve better speedup than AD-LDA; (2) In fact, if we take the waiting time for synchronization in AD-LDA into consideration, the speedup of AD-LDA could have been even worse. For example, in a busy distributed computing environment, when $P = 128$, AD-LDA may take about

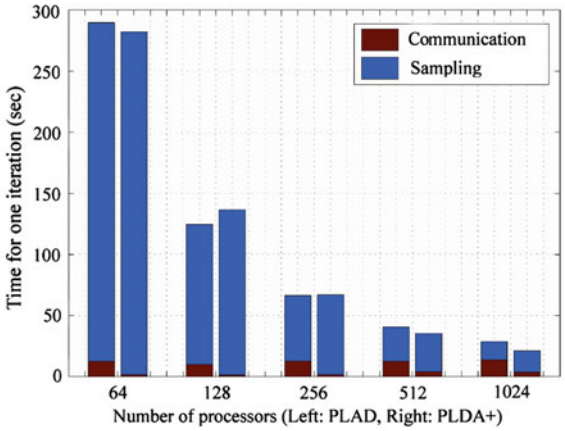

Fig. 12.15 Communication and sampling time 64 to 1,024 processors on Wiki-20T (see color insert)

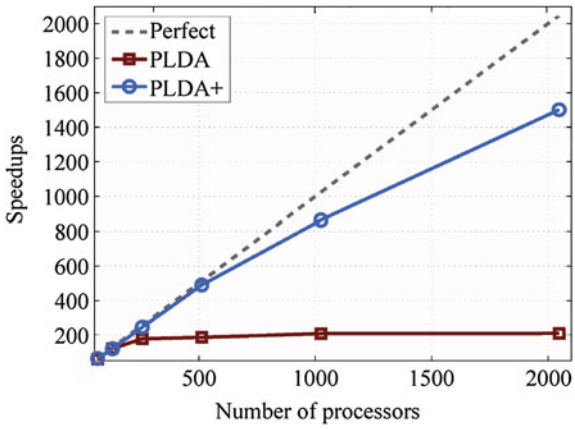

Fig. 12.16 Parallel speedup results for 64 to 2,048 processors on Wiki-200T (see color insert)

70 s for communication in which only about 10 s are used for transmitting word-topic matrix and most of time is used to wait for each other (Fig. 12.16).

On the larger Wiki-200T dataset, the speedup of AD-LDA starts to flat out at $P=512$, whereas PLDA continues to gain in speed.[1] For this dataset, we also list the sampling and communication time ratio of AD-LDA and PLDA+ in Fig. 12.17. As shown in this figure, PLDA+ keeps communication time to quite low values from $P=64$ to $P=2{,}048$. While for AD-LDA, the communication time finally became a bottleneck to prevent it from speedup as the number of processors grows. Though eventually the 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 larger datasets.

[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 $W=1{,}000$, the matrix is 1.6 GB, which is large for communication.

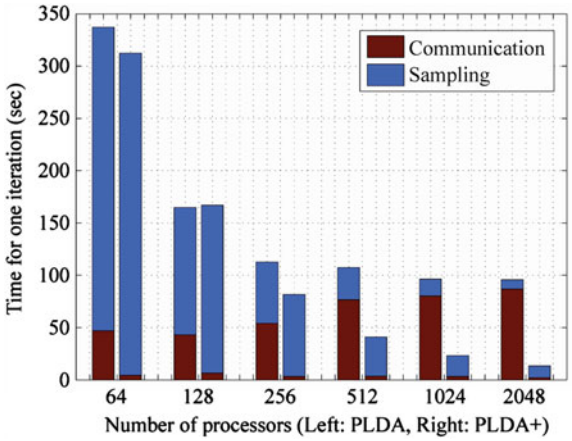

Fig. 12.17 Communication and sampling time 64 to 2,048 processors on Wiki-200T (see color insert)

12.6 Large-Scale Applications

LDA has been shown effective in many tasks (e.g., [23–25]). In this section, we use two large-scale applications, *community recommendation* of Google Orkut and *label suggestion* of Google Confucius [26], to demonstrate the usefulness of PLDA+.

12.6.1 Mining Social-Network User Latent Behavior

Users of social networking services (e.g., Orkut, Facebook, and MySpace) can connect to each other explicitly by adding friends, or implicitly by joining communities. When the number of communities grows over time, finding an interesting community to join can be time consuming. We use LDA to model users' community membership [27]. On a matrix formed by users as rows and communities as columns, all values in user-community cells are initially unknown. When a user joins a community, the corresponding user-community cell is set to one. We apply LDA on the matrix to assign a probability value between zero and one to the unknown cells. When LDA assigns a high probability to a cell, this can be interpreted as a prediction that that cell's user would be very interested in joining that cell's community.

The work of [27] conducted experiments on a large community data set of 492,104 users and 118,002 communities in a privacy-preserved way. The experimental results show that PLDA V1.0 (AD-LDA based implementation) achieves effective performance for personalized community recommendation.

12.6.2 Question Labeling

Confucius is a Q&A system developed by my team at Google Beijing, and has been launched in more than 60 countries [26]. The goal of Question Labeling is to help organize and route questions with automatically recommended labels. The Question Labeling subroutine takes a question as the input and outputs an ordered list of labels that best describe the question. Labels consist of a set of words or phrases that best describe the topic or type of the question. Confucius allows at most five labels per question, but puts no limit on the size of the global label vocabulary. Confucius organizes the most important category labels into a two-layer hierarchy, in order to provide a better browsing experience. Question Labeling is used by two other subroutines: User Rank and Question Labeling . When ranking users, User Rank uses popular labels to compute the topic-dependent rank scores. Question Routing assigns questions to users via either subscription or expert identification, during which labels generated byQuestion Labeling are used for matching. The precision and recall of suggested labels are two important metrics for measuring Question Labeling performance. Precision measures the correctness of suggested labels, while recall measures the completeness.

Figure 12.18 shows the two parts of Question Labeling offline training and online suggestion. In the offline training part, we employ LDA to model the relationship between words and topic labels. The training data is the existing set of questions with user-submitted labels. First, we merge all questions with the same label l into a meta-document d_l, and form a set of meta-documents $\{d_l\}$ (Fig. 12.18, Steps 1 and 2). Second, we remove all stop words and rare words to reduce the size of each meta-document (Step 3). Third, we use $\{d_l\}$ as the corpus to train LDA models (Steps 5–6). The label corresponds to the document in LDA definition, while the words in the meta-documents correspond to the words. The resulted LDA model decomposes the probability Question Labeling—this is similar to the factor model in recommendation algorithms, expressed in terms of probabilities. Instead of a single model, Question Labeling trains several LDA models with different number of latent topics. Using multiple LDA models with different k-s is known as *bagging*, which typically outperforms a single model and avoids the difficult task of setting an optimal k, as discussed by Hofmann [28]. In the current Question Labeling Question Labeling system, the following numbers of topics are used: $k=32$, 64, 128 and 256. We collect all LDA models into a set M (Step 7) and save it to disk. The training part works offline. To handle large training data, we use PLDA+ on thousands of machines in order to maintain training time within the range of a few hours.

The online suggestion part assigns labels to a question as the user types it. The bottom half of Fig. 12.18 depicts the suggestion algorithm. First, we use each LDA model in M to infer the topic distributions $\{\theta_{q,k}\}$ of the question q (Step 1). Then, we compute the cosine similarity $CosSim(\theta_{q,k}, \theta_{d_l,k})$ between $\theta_{q,k}$ and $\theta_{d_l,k}$ (Step 2). Third, we use the mean similarity over different values of k as the final similarity $S(q,l)$ between a question and a label (Step 3). Finally, we sort all $l \in$

Question Labeling Subroutine (Offline Training)
Input: Questions $Q = \{q_1, \ldots, q_n\}$, in which $q_i = \{w_{i,1}, \ldots, w_{i,n}\}$, Labels $L = \{l_1, \ldots, l_m\}$ and their relationship with labels $R \in Q \times L$
Output: LDA models $M = \{(\theta, \phi, k)\}$
1: **for** $l \in L$
2: $d_l = \{w | w \in d, \forall (d,l) \in R\}$
3: Remove stop words and rare words from all $d_l, l \in L$
4: $M = \{\}$
5: **for** $k \in \{32, 64, 128, 512\}$
6: Train LDA model (θ, ϕ, k), with $\{d_l\}$ and k topics
7: $M \leftarrow (\theta, \phi, k)$
Question Labeling Subroutine (Online)
Given: LDA Models M = $\{(\theta, \phi, k)\}$
Input: Question $q = \{w_1, \ldots, w_{|q|}\}$
Output: Suggested Labels $L_q = \{l_1, \ldots, l_n\}$
1: Infer $\theta_{q,k}$ with M
2: $S_k(\theta_q, \theta_{d_l}) = CosSim(\theta_{q,k}, \theta_{d_l}), \forall l \in L$
3: $S(q,l) = \{S_k(\theta_{q,k}, \theta_{d_l,k})\}$ /* *Mean similarity* */
4: $L_q = \{l || \{l' | S(q,l') > S(q,l)\}| < N\}$ /**Top N labels*/

Fig. 12.18 Question Labeling subroutine

L by $S(q, l)$ in descending order, and take the first N (say ten) labels as recommended ones
(Step 4).

Using PLDA+ for Question Labeling has two benefits: semantic matching and scalability. PLDA+ decomposes each question and answer into a distribution over a set of latent topics. When encountering ambiguous words, PLDA+ can use the context to decide the correct semantics. For example, Question Labeling suggests only labels such as 'mobile' and 'iPhone' to the question *How to crack an apple?*, although the word *apple* also means the fruit "apple." In addition, PLDA+ is designed to scale gracefully to more input data by employing more machines.

12.7 Concluding Remarks

In this chapter, we first presented the implementation of AD-LDA based on MPI. We then analyzed the communication bottleneck of AD-LDA. In order to reduce this communication bottleneck, PLDA+ divides processors into two types, namely P_d processors and P_w processors, and also employs pipeline-based Gibbs sampling (PGS). Though any distributed scheme may subject to pathological workload, PLDA+ appears to be resilient to substantial deadline misses caused by imbalanced workload. Extensive experiments on large-scale document collections demonstrated that PLDA+ can achieve much higher speedup than AD-LDA, thanks to both its improved load balancing and reduced communication overhead. From the experience with implementing PLDA+ we learned that on top of MapReduce or MPI, advanced

strategies such as data placement and pipeline processing should be considered to further smooth out bottlenecks.

References

1. Z. Liu, Y. Zhang, E.Y. Chang, M. Sun, Plda+: parallel latent dirichlet allocation with data placement and pipeline processing. ACM Trans. Intell. Syst. Technol. (ACM TIST) **2**(3) (2003)
2. D.M. Blei , A.Y. Ng, M.I. Jordan, Latent dirichlet allocation. J. Mach. Learn. Res. **3**, 993–1022 (2003)
3. D.M. Mimno, A. McCallum, Organizing the oca: learning faceted subjects from a library of digital books, in *Proceedings of ACM/IEEE Joint Conference on Digital Libraries*, pp. 376–385, 2007
4. D. Newman, A. Asuncion, P. Smyth, M. Welling, Distributed inference for latent dirichlet allocation, in *Proceedings of NIPS*, 2007
5. A. Asuncion, P. Smyth, M. Welling Asynchronous distributed learning of topic models, in *Proceedings of NIPS*, pp. 81–88, 2008
6. Y. Wang, H. Bai, M. Stanton, W.Y. Chen, E.Y. Chang, Plda: Parallel latent dirichlet allocation for large-scale applications, in *Proceedings of AAIM*, pp. 301–314, 2009
7. T. Minka, J. Lafferty, Expectation-propagation for the generative aspect model, in *Proceedings of UAI*, 2002, pp. 352–359
8. T. Griffiths, M. Steyvers, Finding scientific topics. Proc. Natl. Acad. Sci. U S A **101**(90001), 5228–5235 (2004)
9. M. Rosen-Zvi, C. Chemudugunta, T. Griffiths, P. Smyth, M. Steyvers, Learning author-topic models from text corpora. ACM Trans. Inf. Syst. **28**(1), 1–38 (2010)
10. W. Li, A. MaCallum, Pachinko allocation: DAG-structured mixture models of topic correlations, in *Proceedings of ICML*, 2006, pp. 577–584
11. C. Chemudugunta, P. Smyth, M. Steyvers, Modeling general and specific aspects of documents with a probabilistic topic model, in *Proceedings of NIPS*, 2007
12. D. Newman, A. Asuncion, P. Smyth, M. Welling, Distributed algorithms for topic models. J. Mach. Learn. Res. **10**, 1801–1828 (2009)
13. R. Nallapati, W. Cohen, J. Lafferty, Parallelized variational em for latent dirichlet allocation: an experimental evaluation of speed and scalability, in *ICDM Workshop*, pp. 349–354, 2007
14. J. Wolfe, A. Haghighi, D. Klein, Fully distributed em for very large datasets, in *Proceedings of ICML*, pp. 1184–1191, 2008
15. R. Gomes, M. Welling, P. Perona, Memory bounded inference in topic models, in *Proceedings of ICML*, pp. 344–351, 2008
16. I. Porteous, D. Newman, A. Ihler, A. Asuncion, P. Smyth, M. Welling, Fast collapsed Gibbs sampling for latent dirichlet allocation, in *Proceedings of ACM KDD*, pp. 569–577, 2008
17. R. Thakur, R. Rabenseifnery, W. Gropp, Optimization of collective communication operations in mpich. Int. J. High. Perform. Comput. Appl. **19**(1), 49–66 (2005)
18. S. Graham, M. Snir, C. Patterson, *Getting up to speed: The future of supercomputing* (National Academies Press, 2005)
19. J.P. Shen, M.H. Lipasti *Modern Processor Design: Fundamentals of Superscalar Processors* (McGraw-Hill Higher Education, 2005)
20. J. Blinn, A trip down the graphics pipeline: line clipping. IEEE Comput. Graph. Appl. **11**(1), 98–105 (1991)
21. P. Berenbrink, T. Friedetzky, Z. H.u., R. Martin, On weighted balls-into-bins games. Theor. Comput. Sci. **409**(3), 511–520 (2008)
22. E.Y. Chang, H. Garcia-Molina, C. Li 2d, bubbleup: Managing parallel disks for media servers, in *Proceedings of FODO*, pp. 221–230, 1998

23. D. Cohn, H. Chang, Learning to probabilistically identify authoritative documents, in *Proceedings of ICML*, pp. 167–174, 2000
24. A. Popescul, L. Ungar, D. Pennock, S. Lawrence, Probabilistic models for unified collaborative and content-based recommendation in sparse-data environments, in *UAI*, pp. 437–444, 2001
25. L.J. Li, G. Wang, L. Fei-Fei, OPTIMOL: automatic online picture collection via incremental model learning, in *Proceedings of CVPR*, 2007
26. X. Si, E.Y. Chang, Z. Gyöngyi, M. Sun, Confucius and its intelligent disciples: Integrating social with search, in *Proceedings of the VLDB*, pp. 1505–1516, 2010
27. W.Y. Chen, J.C. Chu, J. Luan, H. Bai, Y. Wang, E.Y. Chang, Collaborative filtering for orkut communities: Discovery of user latent behavior, in *Proceedings of the 18th International WWW Conference*, pp. 681–690, 2009
28. T . Hofmann, Probabilistic latent semantic analysis, in *Proceedings of UAI*, pp. 289–296, 1999