PLDA: Parallel Latent Dirichlet Allocation for Large-scale Applications

Yi Wang[1], Hongjie Bai[1], Matt Stanton[2],
Wen-Yen Chen[1], and Edward Y. Chang[1]

[1] Google Beijing Research, Beijing, 100084, China
{wyi,hjbai,wenyen,edchang}@google.com
[2] Computer Science, CMU, USA
mlstanto@cs.cmu.edu

Abstract. This paper presents PLDA, our parallel implementation of Latent Dirichlet Allocation on MPI and MapReduce. PLDA smooths out storage and computation bottlenecks and provides fault recovery for lengthy distributed computations. We show that PLDA can be applied to large, real-world applications and achieves good scalability. We have released MPI-PLDA to open source at http://code.google.com/p/plda under the Apache License.

1 Introduction

Latent Dirichlet Allocation (LDA) was first proposed by Blei, Ng and Jordan to model documents [1]. Each document is modeled as a mixture of K latent topics, where each topic, k, is a multinomial distribution $\boldsymbol{\phi}_k$ over a V-word vocabulary. For any document d, its topic mixture $\boldsymbol{\theta}_d$ is a probability distribution drawn from a Dirichlet prior with parameter α. For each ith word $w_{d,i}$ in d, a topic $z_{d,i}$ is drawn from $\boldsymbol{\theta}_d$, and $w_{d,i}$ is drawn from $\boldsymbol{\phi}_{z_{d,i}}$.

Given an input corpus W, the LDA learning process consists of calculating Φ, a maximum-likelihood estimate of model parameters. Given this model, we can infer topic distributions for arbitrary documents. The idea of describing documents in terms of their topic compositions has seen broad application in information-management applications. For example, in a query 'apple pie', LDA can infer from the presence of 'pie' that the meaning of 'apple' is closer to 'fruit' than 'computer'. Using this meaning information obtained by learning an LDA model, documents with the meaning 'fruit' can be effectively identified and returned to answer the 'apple pie' query.

In this paper, we first present LDA and related work in Section 2. In Section 3 we present parallel LDA (PLDA) and explain how it works via a simple example. We then present our two fault-tolerant PLDA implementations (the current core algorithm of PLDA is the AD-LDA algorithm [2]), one on MPI [3] and the other on MapReduce [4]. Section 4 uses two large-scale applications to demonstrate the scalability of PLDA. Finally, we discuss future research plans in Section 5.

2 Learning Algorithms for LDA

Blei, Ng and Jordan [1] proposed using a Variational Expectation Maximization (VEM) algorithm for obtaining maximum-likelihood estimate of Φ from W. 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 [5], which uses another approximate inference method, Expectation Propagation (EP), in the E-step.

Griffiths and Steyvers [6] proposed using Gibbs sampling, a Markov-chain Monte Carlo method, to perform inference. By assuming a Dirichlet prior, β, on model parameters $\Phi = \{\boldsymbol{\phi}_k\}$ (a set of topics), Φ can be integrated (hence removed from the equation) using the Dirichlet-multinomial conjugacy. The posterior distribution $P(Z|W)$ can then be estimated using a collapsed Gibbs sampling algorithm, which, in each iteration, updates each topic assignment $z_{d,i} \in Z$ by sampling the full conditional posterior distribution:

$$p(z_{d,i} = k \mid Z_{-(d,i)}, w_{d,i} = v, W_{-(d,i)}) \propto \left(C^{\text{doc}}_{d,k} + \alpha\right) \frac{C^{\text{word}}_{v,k} + \beta}{\sum_{v'} C^{\text{word}}_{v',k} + V\beta}, \quad (1)$$

where $k \in [1, K]$ is a topic, $v \in [1, V]$ is a word in the vocabulary, $w_{d,i}$ denotes the ith word in document d and $z_{d,i}$ the topic assigned to $w_{d,i}$, $W_{-(d,i)}$ denotes the words in the training corpus with $w_{d,i}$ excluded, and $Z_{-(d,i)}$ the corresponding topic assignments of $W_{-(d,i)}$. In addition, $C^{\text{word}}_{v,k}$ denotes the number of times that word v is assigned to topic k not including the current instance $w_{d,i}$ and $z_{d,i}$, and $C^{\text{doc}}_{d,k}$ the number of times that topic k has occurred in document d not including $w_{d,i}$ and $z_{d,i}$. Whenever $z_{d,i}$ is assigned to a sample drawn from (1), matrices C^{word} and C^{doc} are updated. After enough sampling iterations to burn in the Markov chain, $\Theta = \{\boldsymbol{\theta}_d\}_{d=1}^{D}$ and $\Phi = \{\boldsymbol{\phi}_k\}_{k=1}^{K}$ can be estimated by

$$\theta_{d,k} = \frac{C^{\text{doc}}_{d,k} + \alpha}{\sum_{k'=1}^{K} C^{\text{doc}}_{d,k'} + K\alpha} \qquad \phi_{v,k} = \frac{C^{\text{word}}_{v,k} + \beta}{\sum_{v'=1}^{V} C^{\text{word}}_{v',k} + V\beta}. \quad (2)$$

Griffiths and Steyvers [6] conducted an empirical study of VEM, EP and Gibbs sampling. The comparison shows that Gibbs sampling converges to a known ground-truth model more rapidly than either VEM or EP.

2.1 LDA Performance Enhancement

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. Relevant parallelization efforts include:

- Nallapati and et al. [7] reported distributed computing of the VEM algorithm for LDA [1].
- Newman and et al. [2] presented two synchronous methods, AD-LDA and HD-LDA, to perform distributed Gibbs sampling. AD-LDA is similar to distributed EM [8] from a data-flow perspective; HD-LDA is theoretically equivalent to learning a mixture of LDA models but suffers from high computation cost.
- Asuncion, Smyth and Welling [9] presented an asynchronous distributed Gibbs sampling algorithm.

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

- Gomes, Welling and Perona [10] presented an enhancement of the VEM algorithm using a bounded amount of memory.
- Porteous and et al. [11] proposed a method to accelerate the computation of (Eq.1). The acceleration is achieved by no approximations but using the property that the probability vectors, $\boldsymbol{\theta}_d$, are sparse in most cases.

3 PLDA

We consider two well-known distributed programming models, MPI [3] and MapReduce [4], to parallelize LDA learning. Before introducing PLDA, we briefly review the AD-LDA algorithm [2], and its dependency on the collective communication operation, *AllReduce*. We show how to express the AD-LDA algorithm [6] in both models of MPI and MapReduce.

3.1 Parallel Gibbs Sampling and AllReduce

The AD-LDA algorithm [2] distributes D training documents over P processors, with $D_p = D/P$ documents on each processor. AD-LDA partitions document content $W = \{\boldsymbol{w}_d\}_{d=1}^{D}$ into $\{W_{|1}, \ldots, W_{|P}\}$ and corresponding topic assignments $Z = \{\boldsymbol{z}_d\}_{d=1}^{D}$ into $\{Z_{|1}, \ldots, Z_{|P}\}$, where $W_{|p}$ and $Z_{|p}$ exist only on processor p. Document-specific counts, C^{doc}, are likewise distributed; however, each processor maintains its own copy of word-topic counts, C^{word}. We represent processor-specific counts as $C^{\text{doc}}_{|p}$. $C^{\text{word}}_{|p}$ is used to temporarily store word-topic counts accumulated from local documents' topic assignments on each processor.

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

$$p(z_{d,i|p} = k \mid Z_{-(d,i)}, w_{d,i|p} = v, W_{-(d,i)}) \propto \left(C^{\text{doc}}_{d,k|p} + \alpha\right) \frac{C^{\text{word}}_{v,k} + \beta}{\sum_{v'} C^{\text{word}}_{v',k} + V\beta}, \quad (3)$$

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

Table 1: Nine-Document Example.

d	*p*	*Document Title*
h1	*p1*	*Human* machine *interface* for ABC *computer* applications
h2	*p2*	A *survey* of user opinion of *computer system response time*
h3	*p1*	The *EPS* user *interface* management *system*
h4	*p2*	*System* and *human system* engineering testing of *EPS*
h5	*p1*	Relation of *user* perceived *response time* to error measurement
m1	*p2*	The generation of random, binary, ordered *trees*
m2	*p1*	The intersection *graph* of paths in *trees*
m3	*p2*	*Graph minors* IV: Widths of *trees* and well-quasi-ordering
m4	*p1*	*Graph minors*: A *survey*

Table 2: C^{doc} Matrices on machines $p1$ and $p2$.

p	*d*	$C^{doc}_{d,t1}$	$C^{doc}_{d,t2}$	*Topic Assignment*
p1	*h1*	2	1	human=$t1$, interface=$t1$, computer=$t2$
	h3	2	2	interface=$t1$, user=$t2$, system=$t1$, EPS=$t2$
	h5	2	1	user=$t1$, response=$t2$, time=$t1$
	m2	2	0	trees=$t1$, graph=$t1$
	m4	1	2	survey=$t2$, graph=$t1$, minors=$t2$
p2	*h2*	4	2	computer=$t1$, user=$t1$, system=$t2$, response=$t1$, time=$t2$, survey=$t1$
	h4	2	2	human=$t2$, system=$t1$, system=$t2$, EPS=$t1$
	m1	1	0	trees=$t1$
	m3	2	1	trees=$t1$, graph=$t2$, minors=$t1$

3.2 Illustrative Example

We use a two-category, nine-document example, originally presented in [12] for explaining LSA, to illustrate how PLDA works. Table 1 shows nine documents separated into two categories, where symbol *h* stands for human computer interaction, and *m* for mathematical graph theory. There are five document titles (with extracted terms italicized) in the *h* category, labeled from *h1* to *h5*, and four documents in the *m* category, from *m1* to *m4*.

Suppose we use two machines *p1* and *p2* and target for finding two latent topics *t1* and *t2*. Nine documents are assigned to *p1* or *p2* as depicted in the second column of the table. PLDA first initializes each word's topic from a uniform distribution $U(1, K = 2)$. Table 2 depicts the document-topic matrices on machines $p1$ and $p2$, or $C^{\text{doc}}_{|p1}$ $C^{\text{doc}}_{|p2}$, respectively. The first row shows that document $h1$ on machine $p1$ receives topic assignment $t1$ on words *human* and *interface*, and topic assignment $t2$ on word *computer*. Therefore, the $h1$ row of topic counts are 2 for topic $t1$ and 1 for $t2$. PLDA performs this counting process on all documents on machines $p1$ and $p2$, respectively. Notice that $C^{\text{doc}}_{|p1}$ and $C^{\text{doc}}_{|p2}$ reside on local machines, and no inter-machine communication is involved.

The other important data structure is the word-topic matrices depicted in Table 3. For instance, the first column under machine $p1$, $C^{\text{word}}_{w,t1|p1}$, records how many times topic $t1$ is assigned to each word on machine $p1$. The second column under machine $p2$, $C^{\text{word}}_{w,t2|p2}$, records topic $t2$ assignment on machine $p2$. Each machine also replicates a global topic assignment matrix C^{word}, which is updated at the end of each iteration

Table 3: C^{word} Matrices.

w	Machine $p1$				Machine $p2$			
	$C^{word}_{w,t1\|p1}$	$C^{word}_{w,t2\|p1}$	$C^{word}_{w,t1}$	$C^{word}_{w,t2}$	$C^{word}_{w,t1\|p2}$	$C^{word}_{w,t2\|p2}$	$C^{word}_{w,t1}$	$C^{word}_{w,t2}$
EPS	0	1	1	1	1	0	1	1
computer	0	1	1	1	1	0	1	1
graph	2	0	2	1	0	1	2	1
human	1	0	1	1	0	1	1	1
interface	2	0	2	0	0	0	2	0
minors	0	1	1	1	1	0	1	1
response	0	1	1	1	1	0	1	1
survey	0	1	1	1	1	0	1	1
system	1	0	2	2	1	2	2	2
time	1	0	1	1	0	1	1	1
trees	1	0	3	0	2	0	3	0
user	1	1	2	1	1	0	2	1

through the AllReduce operation. This is where inter-machine communication takes place.

Next, PLDA performs a number of Gibbs sampling iterations. Rather than performing topic assignment randomly in the initialization step, Gibbs sampling performs topic assignment according to Equation (3). After each iteration, both Tables 2 and 3 are updated. At the end, the master machine outputs C^{word}, on which one can look up for the topic distribution of a word.

3.3 Parallel LDA Using MPI

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 Section 3.1.

Figure 1 presents the details of Procedure `MPI-PLDA`. The algorithm first attempts to load checkpoints $Z_{|p}$ if a machine failure took place and the computation is in the recovery mode. The procedure then performs initialization (lines 5 to 10), where for each word, its topic is sampled from a uniform distribution. Next, $C^{\text{doc}}_{|p}$ and $C^{\text{word}}_{|p}$ can be computed from the histogram of $Z_{|p}$ (line 12). To obtain C^{word}, Procedure `MPI-PLDA` invokes `MPI_Allreduce` (line 13). In the Gibbs sampling iterations, each word's topic is sampled from the approximate posterior distribution (Eq.3) and $C^{\text{doc}}_{|p}$ is updated accordingly (lines 15 to 19). At the end of each iteration, the procedure checkpoints $Z_{|p}$ (line 20) and recomputes $C^{\text{word}}_{|p}$ and C^{word} (lines 21 to 22). After a sufficient number of iterations, the converged LDA model is outputted by the master(line 25).

Procedure `MPI-PLDA`(*iteration-num*)

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

Fig. 1: The MPI Procedure of PLDA.

Performance and Fault Recovery. Various MPI implementation systems use different AllReduce algorithms; the state-of-the-art is the recursive doubling and halving (RDH) algorithm presented in [3], which was used by many MPI implementations including the well known MPICH2. RDH includes two phases: *Reduce-scatter* and *All-gather*. Each phase runs 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 2, because no worker would be idle during communication.

RDH provides no facilities for fault recovery. In order to provide fault-recovery capability in `MPI-PLDA`, we checkpoint the worker state before AllReduce. This ensures that when one or more processors fail in an iteration, we 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

MPI-PLDA. In practice, we checkpoint only $Z_{|p}$, because $W_{|p}$ can be reloaded from training data, $C^{\text{doc}}_{|p}$ and C^{word} can also be recovered from the histogram of $Z_{|p}$. The recovery code is at the beginning of MPI-PLDA: if there is a checkpoint on the disk, load it; otherwise perform the random initialization.

3.4 Parallel LDA using MapReduce

MapReduce processes input and output in the form of key-value pairs known as *tuples*. A set of tuples is normally distributed across multiple processors, so each processor can efficiently load and process the local tuple subset, known as a *shard*. The operation of dividing and distributing tuples is known as *sharding*.

A MapReduce job consists of three successive phases: *mapping*, *shuffling* and *reducing*. The mapping and reducing phases are *programmable*. To program the mapping phase for LDA, we define three procedures: MapperStart, Map and MapperFlush. To program the reducing phase, we define ReducerStart, Reduce and ReducerFlush.

For every input shard, the MapReduce implementation system creates a thread, known as a *map worker*, on the processor where the shard resides. Each map worker invokes Map to process each tuple in the shard. A map worker invokes MapperStart before the first invocation of Map, and invokes MapperFlush after the last invocation. These user-defined functions invoke an API function MapperOutput to output tuples known as *map-outputs*. Map-output tuples are collected and processed by the shuffling phase; values of map outputs that share the same key are aggregated into a new tuple known as *reduce-input*. The value of a reduce-input is a set of values of map-outputs. Reduce-inputs are grouped into reduce-input shards. For each reduce-input shard, the MapReduce implementation system creates a thread, *reduce worker*, which invokes ReducerStart, Reduce and ReducerFlush in turn to process each reduce-input in the local reduce-input shard. All map workers run in parallel, as do all reduce workers. Workers communicate only in the shuffling phase.

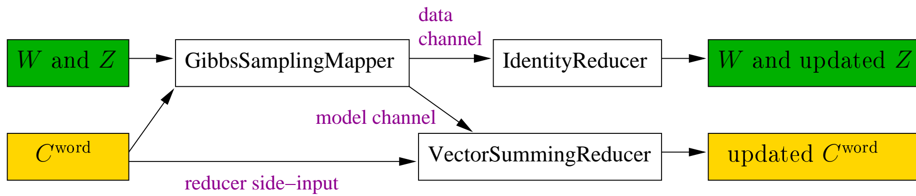


Fig. 2: The MapReduce job corresponding to one Gibbs sampling iteration of PLDA.

We model each Gibbs sampling iteration of PLDA as a MapReduce job, as illustrated in Figure 2, where the map phase does Gibbs sampling and the reduce phase updates the model and topic assignments. Related procedures are depicted in Figure 3. We organize each fraction of $D_p = D/P$ documents, denoted by $W_{|p}$ in Section 3.1, in an input shard, which is then assigned to a map worker by the MapReduce implementation system. Each map worker loads a local copy of the model, C^{word}, when executing

Procedure `PLDA-MapperStart`

1 Load C^{word} updated by previous iteration from GFS;
2 Initialize ΔC^{word} as a zero matrix with the same size as C^{word};
3 Seed the random number generator in a shard-dependent way;

Procedure `PLDA-Map`(*key, value*)

1 $d \leftarrow$ parse `key`;
2 $\{\boldsymbol{w}_d, \boldsymbol{z}_d\} \leftarrow$ parse `value`;
3 $C_d^{\text{doc}} \leftarrow$ histogram unique topics in $\boldsymbol{z}_d$;
4 **for** *each* $w_{d,i} \in \boldsymbol{w}_d$ **do**
5 $\quad C_{d,z_{d,i}}^{\text{doc}} \leftarrow C_{d,z_{d,i}}^{\text{doc}} - 1$, $\Delta C_{d,z_{d,i}}^{\text{word}} \leftarrow \Delta C_{d,z_{d,i}}^{\text{word}} - 1$, $C_{w_{d,i},z_{d,i}}^{\text{word}} \leftarrow C_{w_{d,i},z_{d,i}}^{\text{word}} - 1$;
6 $\quad z_{d,i} \leftarrow$ draw new sample from (1), given C^{word} and C_d^{doc};
7 $\quad C_{d,z_{d,i}}^{\text{doc}} \leftarrow C_{d,z_{d,i}}^{\text{doc}} + 1$, $\Delta C_{d,z_{d,i}}^{\text{word}} \leftarrow \Delta C_{d,z_{d,i}}^{\text{word}} + 1$, $C_{w_{d,i},z_{d,i}}^{\text{word}} \leftarrow C_{w_{d,i},z_{d,i}}^{\text{word}} + 1$;
8 **end**
9 `Output(channel=data, key=`d`, value=`$\{\boldsymbol{w}_d, \boldsymbol{z}_d\}$`);`

Procedure `PLDA-MapperFlush`

1 **for** *each unique word* v *in the vocabulary* **do**
2 $\quad$ `Output(channel=model, key=`v`, value=`ΔC_v^{word}`);`
3 **end**

Fig. 3: Three MapReduce Procedures for PLDA.

`PLDA-MapperStart`. Then it invokes `PLDA-Map` for each document $\boldsymbol{w}_d \in W_{|p}$ to updates the corresponding topic assignments, $\boldsymbol{z}_d$, and outputs $\boldsymbol{z}_d$ to the channel *data*. After Gibbs sampling on all documents in a shard are finished, the map worker invokes `PLDA-MapperFlush` to output the model update opinion matrix, ΔC^{word}, to the channel model with each row of ΔC^{word} as a map-output tuple. The concept *channel* comes from an extension to the standard MapReduce model that allows us to use two reducers to output both the updated topic assignments, Z, and the model, C^{word}. This extension adds an additional parameter to `MapperOutput`, indicating a mapper output channel, where each channel connects to a reducer. Not all MapReduce implementations support this extension. However, we can implement the extension using the standard MapReduce model by appending the channel indicator to each map-output key, then defining `Reduce` to decompose the indicator by parsing from the reduce-input key and invoking different reduce algorithms according the indicator.

In the reduce phase, we use two standard reducers in Figure 2. For each $\boldsymbol{z}_d$, output by GibbsSamplingMapper, IdentityReducer copies it to GFS; for each word v in the vocabulary, VectorSummingReducer aggregates and outputs $C_v^{\text{word}} \leftarrow C_v^{\text{word}} + \sum_{p=1}^{P} \Delta C_{v|p}^{\text{word}}$. Here we use another extension for VectorSummingReducer, the side-input of reducers, which can also be implemented using the standard MapReduce model by appending tuples in side-input, C^{word}, after the standard map-input, $\{W, Z\}$, and defining `Map` identically to output tuples in the side input.

Table 4: Comparing MPI and MapReduce in supporting PLDA.

	Communication Efficiency	Inter-iteration fault-recovery	Intra-iteration fault-recovery
MPI	AllReduce through memory/network	By customized checkpointing	Not yet supported
MapReduce	*Shuffling* via GFS/Disk IO	Not necessary	Built in

From Figure 2 we see that the input and output of the PLDA MapReduce job are identical—both consist of document accompanied by topics assignments, $W_{|p}$ and $Z_{|p}$, as well the model, C^{word}. This allows us to chain up a series of PLDA MapReduce jobs to model the Gibbs sampling iterations.

Performance and Fault Recovery. MapReduce performs AllReduce in the shuffling and reducing phases after the mapping phase. In these phases map outputs are buffered on the local disk, creating a temporary checkpoint, and then aggregated and re-distributed by the shuffling phase. This implementation helps fault recovery. In order to guarantee correct fault recovery, the map workers must execute a *deterministic* map algorithm, which ensures that repeating a map input shard gives the same result. This is necessary because when a map worker fails, the corresponding map input shard is repeated, and the previous map-output may already have been consumed by some reducers. We do not want these map-outputs be generated and reduced again when we repeat processing this input shard. For PLDA, over-reduction will over-accumulate some elements in C^{word}. MapReduce performs consistency checking by comparing the output checksums from a map shard. The checksum itself is commutative: if you generate the same set of outputs in a different ordering, the checksum remains the same. This checksum duplication detection avoids over-reduction. But it also requires that the duplication is detectable—the MapReduce program must generate the same map-outputs for an input shard in different runs. However, the Gibbs sampling algorithm of PLDA is *stochastic* instead of deterministic: the outputs of recovered map workers are different from and will be reduced together with those old outputs. To avoid over-reduction in `PLDA-MapperStart`, we seed the random number generator in a shard-dependent way to ensure that whenever a failed map worker is recovered, it generates the same map-output as in its previous run.

Table 4 compares the MPI and MapReduce implementations of PLDA. In the absence of machine failures, `MPI-PLDA` is more efficient because no disk IO is required between computational iterations. When the number of machine is large, and the mean-time to machine failures becomes a legitimate concern, the target application should either use `MapReduce-PLDA` or force checkpoints with `MPI-PLDA`.

4 Large-scale Applications

LDA has been shown effective in many tasks (e.g.,[13–15]). In this section, we use two large-scale applications, *community recommendation* and *document summarization*, to demonstrate the scalability of PLDA.

Table 5: Speedup Performance of MPI-PLDA.

# Machines	*Computation*	*Communication*	*Synch*	*Total Time*	*Speedup*
1	28911s	0s	0s	28911s	1
2	14543s	417s	1s	14961s	1.93
4	7755s	686s	1s	8442s	3.42
8	4560s	949s	2s	5511s	5.25
16	2840s	1040s	1s	3881s	7.45
32	1553s	1158s	2s	2713s	10.66
64	1558s	1209s	2s	2769s	10.44

4.1 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 PLDA to model users' community membership [16]. 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 PLDA on the matrix to assign a probability value between zero and one to the unknown cells. When PLDA 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 [16] 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 MPI-PLDA achieves effective performance for personalized community recommendation. Table 5 shows the speedup performance and overhead analysis. When we increased the number of machines, we could always reduce computation time in a near-linear fashion. Unfortunately, the communication time increased as number of machines increased. When 32 machines were used, the communication time approached the computation on a single machine. When 64 machines were used, the speedup was worse than using 32 machines. The result was expected due to Amdahl's law: the speedup of a parallel algorithm is limited by the time needed for the overhead or sequential fraction of the algorithm. When accounting for communication and synchronization overheads (see the total time column), the speedup deteriorates as the number of machines increases. Between the two overheads, the synchronization overhead has very little impact on the speedup compared to the communication overhead (which increases with the number of machines). The good news is that when the data size increases (the results of two larger datasets are reported in the next section), we can add more machines to achieve better speedup, because the deterioration point is deferred.

4.2 Category-sensitive Document Summarization

In recent years there is a surge of studies on keyword extraction and document summarization that use graph-based ranking algorithms like PageRank and HITS to rank text entities such as words and sentences. However, in many cases, documents have

category labels, a factor ignored in most previous work. Consider e-business websites like `amazon.com`, which categorize products and support reviews by users. It is useful to summarize reviews for each product by extracting that product's most relevant properties. For example, properties such as size, weight, and stand-by time are relevant for mobile phones, whereas properties such as safety, exterior/interior design, and equipment packages are for automobiles.

We can realize this category-sensitive summarization using PLDA. By creating a training document from all sentences in product reviews. By taking each sentence from product reviews as a training document, we run the PLDA learning algorithm to cluster words into topics. This step estimates the conditional probability distribution of words given topics, $P(w|z)$. By normalizing each column of C^{word}, we can also obtain $P(z|w)$. Note that during the Gibbs sampling iterations, every word in the training corpus is assigned a most likely topic. Given the category labels of each sentence, we can estimate the conditional probability distributions of topics given categories ($P(z|c)$) and vice versa ($P(c|z)$) by counting the co-occurrences of topics and categories.

Using the learning result, we can rank all review sentences of a product, given the category of that product. Denote the input reviews by $\mathcal{I} = \{\boldsymbol{w}_1, \ldots, \boldsymbol{w}_D\}$, where $\boldsymbol{w}_d$ represents a sentence. By running the Gibbs sampling inference algorithm, we estimate the topic assignment of every word in $\mathcal{I}$. By counting the co-occurrence of topic assignments and sentences, we can estimate $P(z|\boldsymbol{w}_d)$ and $P(\boldsymbol{w}_d|z)$. The inference result is useful to compute a category-sensitive characteristic measure $\text{char}(\boldsymbol{w}_d; c) = P(\boldsymbol{w}_d|c)P(c|\boldsymbol{w}_d)$, where c denotes the product category of reviews $\mathcal{I}$. $\text{char}(\boldsymbol{w}_d; c)$ is a natural extension of the topic-sensitive characteristic measure, $\text{char}(\boldsymbol{w}_d; z) = P(\boldsymbol{w}_d|z)P(z|\boldsymbol{w}_d)$, proposed by [13]. Expanding $\text{char}(\boldsymbol{w}_d; c)$, we obtain:

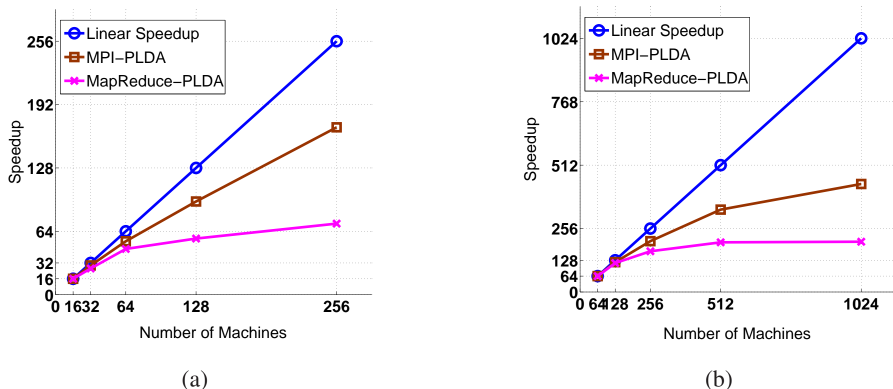
$$\begin{aligned}\text{char}(\boldsymbol{w}_d; c) &= P(\boldsymbol{w}_d|c)P(c|\boldsymbol{w}_d) \\ &= \left[\sum_z P(\boldsymbol{w}_d|z)P(z|c)\right]\left[\sum_z P(z|\boldsymbol{w}_d)P(c|z)\right]\end{aligned} \tag{4}$$

where $P(z|c)$ and $P(c|z)$ come from the learning result and $P(\boldsymbol{w}_d|z)$ and $P(z|\boldsymbol{w}_d)$ come from the inference result.

We performed experiments on two datasets: a Wikipedia dataset and a forum dataset. The Wikipedia set consists of $2,122,618$ articles after removing those with less than 100 words. The forum set consists of $2,450,379$ entries extracted from http://www.tianya.cn. While the effectiveness of PLDA on document summarization on the Wikipedia dataset is reported in [17], we report here our experimental results on scalability conducted upon both datasets.

We measured and compared the speedup of `MPI-PLDA` and `MapReduce-PLDA` using these two datasets. The dataset size and training parameters are shown in Figure 4. The experiments were conducted on up to $1,024$ machines at Google's distributed data centers. Not all machines are identically configured; however, each machine is configured with a CPU faster than 2GHz and memory larger than 4GBytes. We ran Wikipedia dataset on $16/32/64/128/256$ distributed machines. Because the data set is too large to be fit into a single machine's memory, we used 16 machines as the baseline to measure the speedup of using more than 16 machines. To quantify speedup, we made an assumption that the speedup of using 16 machines is 16 compared to using one machine. This

assumption is reasonable for our experiments, since PLDA does enjoy approximately linear speedup when the number of machines is up to 32. Similarly we ran the forum dataset on $64/128/256/512/1,024$ distributed machines and used 64 machines as the baseline. Since our aim was to measure speedup, not convergence, we ran 20 iterations on Wikipedia and 10 on the forum dataset[3].



(a) (b)

Fig. 4: The speedup of (a) Wikipedia: $K = 500$, $V = 20000$, $D = 2,122,618$, TotalWordOccurrences $= 447,004,756$, iterations $= 20$, $\alpha = 0.1$, $\beta = 0.1$. (b) Forum Dataset: $K = 500$, $V = 50000$, $D = 2,450,379$, TotalWordOccurrences $= 3,223,704,976$, iterations $= 10$, $\alpha = 0.1$, $\beta = 0.1$.

Figure 4 shows that PLDA can achieve linear speedup when the number of machines is below about 100. It can no longer achieve linear speedup when the number of machines continues to increase beyond a data-size dependent threshold. This is expected due to both the increase in the absolute time spending in communication between machines, and the increase in the fraction of the communication time in the entire execution time. When the fraction of the computation part dwindles, adding more machines (CPUs) cannot improve much speedup. Worse yet, when the communication time continues to increase, the computation time reduced by parallelization cannot compensate for the increase in the communication time, and speedup actually decreases. On the one hand, as we previously stated and also observed in [18], when the dataset size increases, and hence the computation time increases, we can add more machines to productively improve speedup. On the other hand, a job will eventually be dominated by the communication overhead, and adding more machines may be counter-productive. Therefore, the next logical step in performance enhancement is to consider communication time reduction [19] (discussed further in concluding remarks).

[3] Since the time of running N Gibbs sampling iterations is the same as N times the time of running one iteration, we do not need to run PLDA to convergence in order to measure and compare speedup.

Table 6: Speedup Performance of MPI-PLDA and MapReduce-PLDA

# Machines	MPI-PLDA		MapReduce-PLDA	
	Running Time	*Speedup*	*Running Time*	*Speedup*
16	11940s	16	12022s	16
32	6468s	30	7288s	26
64	3546s	54	4165s	46
128	2030s	94	3395s	57
256	1130s	169	2680s	72

(a) Widipedia dataset (Runtime of 20 iterations)

# Machines	MPI-PLDA		MapReduce-PLDA	
	Running Time	*Speedup*	*Running Time*	*Speedup*
64	9012s	64	10612s	64
128	4792s	120	5817s	117
256	2811s	205	4132s	164
512	1735s	332	3390s	200
1024	1323s	436	3349s	203

(b) Forum dataset (Runtime of 10 iterations)

Comparing `MPI-PLDA` with `MapReduce-PLDA`, `MPI-PLDA` enjoys better speedup than `MapReduce-PLDA`. This is because `MPI-PLDA` uses highly efficient in-memory communication, whereas `MapReduce-PLDA` involves machine scheduling and disk IO between iterations. Indeed, Table 6 shows that when more machines are added, `MPI-PLDA` enjoys better scalability. For instance, the running time that `MPI-PLDA` takes on the Wikipedia dataset, using 256 machines, is $1,130$ seconds, which is less than a half of the running time that `MapReduce-PLDA` takes. While training the Wikipedia set on one machine for 20 iterations can take two days (if we configure that machine with sufficient memory), it takes just 20 minutes to complete on 256 machines.

When the data size becomes much larger and hence more machines (say, tens of thousands) are used, the chance that some machines may fail during a computation iteration becomes non-negligible. In such situation, we can either employ `MapReduce-PLDA` because of its support of intra-iteration fault recovery, or we can support intra-interaction recovery in `MPI-PLDA`[4].

5 Conclusion

In this paper, we presented two parallel implementations of PLDA, one based on MPI and the other on MapReduce. We have released the MPI version to open source at http://code.google.com/p/plda under the Apache License.

We plan to further our work in several directions. First, we plan to experiment with different probabilistic distributions or processes such as Pitman Yor and Chinese Restaurant Process. Second, we are investigating algorithms for further speeding up

[4] When machine can fail frequently during one iteration, hardware redundancy may be necessary to ensure reliability.

Gibbs sampling. Third, communication time increases as the number of machines increases, and this further reduces the computation fraction of an algorithm. As pointed by J. Demmel [19], since the improvement of CPU performance outpaces the improvement of IO/communication performance, communication cost increasingly dominates a parallel algorithm. We will look into strategies to reduce communication time.

References

1. Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent Dirichlet allocation. Journal of Machine Learning Research (2003)
2. Newman, D., Asuncion, A., Smyth, P., Welling, M.: Distributed inference for latent Dirichlet allocation. In: NIPS. (2007)
3. Thakur, R., Rabenseifnery, R., Gropp, W.: Optimization of collective communication operations in mpich. Int'l Journal of High Performance Computing Applications **19**(1) (2005) 49–66
4. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: OSDI. (2004) 137–150
5. Minka, T., Lafferty, J.: Expectation-propagation for the generative aspect model. In: UAI. (2002)
6. Griffiths, T.L., Steyvers, M.: Finding scientific topics. Proceedings of the National Academy of Sciences of U.S. **101** (2004) 5228–5235
7. Nallapati, R., Cohen, W., Lafferty, J.: Parallelized variational em for latent dirichlet allocation: An experimental evaluation of speed and scalability. In: ICDM Workshops. (2007)
8. Chu, C.T., Kim, S.K., Lin, Y.A., Yu, Y., Bradski, G., Ng, A.Y., Olukotun, K.: Mapreduce for machine learning on multicore. In: NIPS. (2006)
9. Asuncion, A., Smyth, P., Welling, M.: Asynchronous distributed learning of topic models. In: NIPS. (2008)
10. Gomes, R., Welling, M., Perona, P.: Memory bounded inference in topic models. In: ICML. 2008
11. Porteous, I., Newman, D., Ihler, A., Asuncion, A., Smyth, P., Welling, M.: Fast collapsed gibbs sampling for latent dirichlet allocation. In: KDD. (2008)
12. Landauer, T.K., Foltz, P.W., Laham, D.: An introduction to latent semantic analysis. Discourse Processes **25** (1998) 259–284
13. Cohn, D., Chang, H.: Learning to probabilistically identify authoritative documents. In: ICML. (2000)
14. Popescul, A., Ungar, L., Pennock, D., Lawrence, S.: Probabilistic models for unified collaborative and content-based recommendation in sparse-data environments. In: UAI. (2001)
15. Li, L.J., Wang, G., Fei-Fei, L.: OPTIMOL: automatic online picture collection via incremental model learning. In: CVPR. (2007)
16. Chen, W.Y., Chu, J.C., Luan, J., Bai, H., Wang, Y., Chang, E.Y.: Collaborative filtering for orkut communities: Discovery of user latent behavior. In: Proc. of the 18th International WWW Conference. (2009)
17. Liu, Z., Wang, Y., Zhu, K., Sun, M.: Category-focused ranking for keyword extraction and document summarization. Google Technical Report, submitted for publication (2009)
18. Chang, E.Y., Zhu, K., Wang, H., Bai, H., Li, J., Qiu, Z., Cui, H.: Psvm: Parallelizing support vector machines on distributed computers. In: NIPS. (2007)
19. Demmel, J.: Avoiding communication in dense and sparse linear algebra (invited talk). In: MMDS. (2008)