# An Architecture for Parallel Topic Models

Alexander Smola
Yahoo! Research, Santa Clara, CA, USA
Australian National University, Canberra
alex@smola.org

Shravan Narayanamurthy
Yahoo! Labs, Bangalore
Torrey Pines Road, Bangalore, India
shravanm@yahoo-inc.com

## ABSTRACT

This paper describes a high performance sampling architecture for inference of latent topic models on a cluster of workstations. Our system is faster than previous work by over an order of magnitude and it is capable of dealing with hundreds of millions of documents and thousands of topics.

The algorithm relies on a novel communication structure, namely the use of a distributed (key, value) storage for synchronizing the sampler state between computers. Our architecture entirely obviates the need for separate computation and synchronization phases. Instead, disk, CPU, and network are used simultaneously to achieve high performance. We show that this architecture is entirely general and that it can be extended easily to more sophisticated latent variable models such as n-grams and hierarchies.

## 1. INTRODUCTION

Latent variable models are a popular tool for encoding long-range dependencies between collections of observations. For instance, when dealing with documents it is highly desirable to go beyond a simple weighted bag-of-words representation and to take co-occurrence information between words in a document into account. Similarly for the purpose of inferring similarity in social networks and recommender systems it is desirable to obtain compact representations.

Clustering and topic models are particularly useful since they allow one to infer structure from large collections of objects without the need of (much) human intervention. Latent Dirichlet Allocation (LDA) [3] and related topic models are particularly useful when it comes to infer overall groups of information (e.g. the information that a particular text contains information about an 'athlete' and a 'scandal', whereas clustering is better suited inferring that a given set of documents refers to the 'Tiger Woods scandal'. In other words, clustering attempts to model objects as one out of $n$ possible classes, whereas topic models represent objects as a mixture of $k$ out of $n$ possible classes (with $k$ being variable but small). It is easy to see from a coding theory point of view that the latter leads to a much more parsimonious representation of an object generation model.

While such models have found widespread use in academia their deployment in industry is largely hampered by limits on scalability. More specifically, the largest published work on LDA by the UC Irvine team [7] is to use 1000 computers for 10 hours to process 8 Million documents taken from PubMed abstracts, which is equivalent to a processing speed of 6,400 documents per computer and hour. By comparison, our implementation is able to generate a model at a rate in excess of 75,000 documents per hour on a single 8-core computer and of 42,000 documents per hour when used in a multi-machine configuration. Google's LDA implementation [9, Table 6b] has a throughput of less than 150 documents per hour and machine (assuming 1000 collapsed sampler iterations) in the most favorable case.

**Outline:** We begin with an overview over the mathematical model underlying Latent Dirichlet Allocation and a discussion of efficient sampling algorithms in Section 2. We proceed with a description of the multicore/cluster pipeline architecture in Section 3 and its implementation in Section 4. We demonstrate the efficiency in multi-machine and multicore experiments in Section 5. Related work and further implementation details are given in the appendix.

## 2. LATENT DIRICHLET ALLOCATION

We give a brief overview of topic models in the context of text modeling (note, though, that the algorithm and our implementation are in no way limited to documents). For a much more detailed discussion see [3, 6].

The basic idea is that each document contains a mix of topics from which words are drawn. The generative process works as follows: for a document $d$ a multinomial distribution $\Theta_i$ is drawn from a Dirichlet prior with parameters $\alpha$. Subsequently, for each word in the document a topic $z_{ij}$ is drawn from the multinomial distribution $\Theta_i$. Finally, the word $w_{ij}$ is drawn from the multinomial distribution $\Psi_{z_{ij}}$. To complete the mode specification we assume that $\Psi$ itself is drawn from a Dirichlet with coefficients $\beta$.[1]

Given the model of Figure 1 we may express the full joint

[1] In our implementation we omit using a Pitman-Yor or Dirichlet Process. The rationale is that memory allocation becomes a crucial issue and we prefer being able to have direct control over it rather than relying on a suitably chosen set of parameters of the DP to address memory allocation. That said, there is no mathematical reason for this limitation.

*Proceedings of the VLDB Endowment,* Vol. 3, No. 1
Copyright 2010 VLDB Endowment 2150-8097/10/09... $ 10.00.

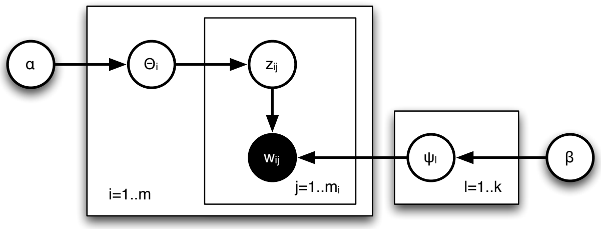


**Figure 1: Latent Dirichlet Allocation: words $w_{ij}$ in a document $i$ are drawn according to the topic-specific distributions $\psi_{z_{ij}}$. The topic distribution per document $\Theta_i$ is drawn from a conjugate Dirichlet. The same applies to $\psi$.**

probability of the data under the LDA model as follows:

$$p(w, z, \Theta, \psi|\alpha, \beta) = \tag{1}$$
$$\left[\prod_{i=1}^{m}\prod_{j=1}^{m_i} p(w_{ij}|z_{ij}, \psi)p(z_{ij}|\Theta_i)\right]\left[\prod_{i=1}^{m} p(\Theta_i|\alpha)\right]\left[\prod_{j=1}^{k} p(\psi_j|\beta)\right]$$

Here $p(w_{ij}|z_{ij}, \psi)$ and $p(z_{ij}|\Theta_i)$ are multinomial distributions and the remaining two distributions are Dirichlet. We could impose a hyperprior on $\alpha$ and $\beta$ as needed. That said, empirical evidence shows that simply performing a maximum likelihood fit is sufficient.

## 2.1 Collapsed Representation

A direct Gibbs sampler using (1) does not mix sufficiently quickly and an improved strategy is to integrate out $\Theta$ and $\psi$. Moreover, collapsed sampling [10, 9, 7] tends to lead to somewhat better models than a variational approach. Most importantly, it allows for a much more compact representation of the model whenever the number of parameters is large — it is only necessary to store the *sparse vector* of statistics for topics actually assigned to words — in the variational or non-collapsed case case dense vectors are required.

To introduce the collapsed representation we need to define a number of statistics of the topic assignments $z_{ij}$:

$$n(t, w) := \sum_{i,j}\{z_{ij} = t \text{ and } w_{ij} = w\}; n(t, d) := \sum_{j}\{z_{dj} = t\}$$

and $n(t) := \sum_w n(t, w)$. Here $n(t, w)$ keeps a list of the topic assignments on a per-word basis. $n(t)$ stores the total of number of times a word is assigned topic $t$. $n(t, d)$ stores topic assignments in a given document. $n(t, w)$ and $n(t, d)$ are sparse. We have

$$p(w, z|\alpha, \beta) \tag{2}$$
$$= \underbrace{\left[\prod_{i=1}^{m}\frac{\prod_{j=1}^{k}\Gamma(\alpha_j + n(t = j, d = i))}{\Gamma\left(\bar{\alpha} + n(d = i)\right)}\frac{\Gamma(\bar{\alpha})}{\prod_{j=1}^{k}\Gamma(\alpha_j)}\right]}_{\text{topic likelihood}} \times$$
$$\underbrace{\left[\prod_{i=1}^{k}\frac{\prod_{j=1}^{W}\Gamma(\beta_j + n(t = i, w = j))}{\Gamma\left(\bar{\beta} + n(t = i)\right)}\frac{\Gamma(\bar{\beta})}{\prod_{j=1}^{W}\Gamma(\beta_j)}\right]}_{\text{word likelihood}}$$

Here $\bar{\alpha} := \sum_i \alpha_i$ and $\bar{\beta} := \sum_w \beta_w$ are aggregates of the Dirichlet smoothing coefficients. Quite often one sets $\bar{\beta} = \beta_0 \cdot N$ where $N$ denotes the number of words. This corresponds to a flat probability model for words. While this design choice is quite unrealistic it turns out not to matter significantly in practice [8]. Nonetheless it is easy to adjust the sampler we discuss to a more adaptive prior. Note that the factors in the products $\prod_{j=1}^{k}$ and $\prod_{j=1}^{W}$ only need to be evaluated whenever $n(t, d) > 0$ and $n(t, w) > 0$ respectively.

## 2.2 Inference for $z$ via Collapsed Sampling

Following [6] we can rewrite (2) to obtain the following unnormalized probabilities to resample the topic $z_{dj}$ for the word $w_{dj}$ in document $d$:

$$p(t|w_{dj}, \text{rest}) \propto \frac{[n(t, w_{dj}) + \beta_w]\,[n(t, d) + \alpha_t]}{n(t) + \bar{\beta}} \tag{3}$$

Eq. (3) can be used in a Gibbs sampler which traverses the set of observations and resamples the values of $z_{dj}$. In terms of locality, the following observations are useful, since they allow us to design parallel samplers by prioritising updates:

**Topic assignments $z_{dj}$:** The values of the variables $z_{ij}$ are entirely local to each document and need not be shared. They can be written to disk after resampling.

**Topic counts for document $n(t, d)$:** These variables are local to document $d$. Again, they need not be shared.

**Topic-word count table $n(t, w)$:** These variables change slowly — for 1 million documents it is unlikely that changing the topic assignments in a single document will have a significant effect on $n(t, w)$ for almost all words. Hence, a modest delay in incorporating changes occurring in a document into $n(t, w)$ is acceptable.

**Topic counts $n(t)$:** This variable is even more slowly varying. A delay in obtaining an up-to-date representation of $n(t)$ will not affect the sampler significantly.

The key idea in designing our sampler is that when resampling on a *per document* basis, we may defer updates to $n(t, w)$ and $n(t)$ until after a document has been resampled. This means that only the topic-document counts $n(t, d)$ change. Such a strategy has been discussed by [10] in the context of generating samples for a *test* distribution. Instead, we use it here to design a sampler for inference of the full model. We decompose (3) into

$$p(t|w_{dj}) \propto \beta_w\frac{\alpha_t}{n(t) + \bar{\beta}} + \beta_w\frac{n(t, d)}{n(t) + \bar{\beta}} + \frac{n(t, w_{dj})\,[n(t, d) + \alpha_t]}{n(t) + \bar{\beta}}$$

The first term in the sum only depends on $w_{dj}$ in a *multiplicative* fashion via $\beta_{w_{dj}}$ and it is constant throughout the document otherwise. The second term is typically sparse as it counts the distribution of topics in a document. Moreover, only two terms need updating whenever we reassign a word to a new topic. Finally, the third term is as sparse as the topic distribution per word.

This shows that in order to compute a proper normalization of $p(t|w_{dj})$ one only needs to compute a normalization for each of the three terms separately. This is cheap since, besides an initial cost for computing $A := \sum_t \frac{\alpha_t}{n(t)+\bar{\beta}}$ and $B := \sum_t \frac{n(t,d)}{n(t)+\bar{\beta}}$, the incremental cost per word is given by the nonzero terms in $n(t, w)$ via $C := \sum_t \frac{n(t,w_{dj})[n(t,d)+\alpha_t]}{n(t)+\bar{\beta}}$. Here the following comes to our aid: only the frequently occurring words (which are likely to occur several times per document, hence we only need to compute the normalization once) are likely dense.

## 2.3 Inference for $\alpha$ and $\beta$

Computing $p(w, z|\alpha, \beta)$ requires one pass through the data (or at least access to $n(t, d)$ for all documents) and one pass through the word-topic table for computation of the likelihood. In particular, the data-dependent term decomposes into a sum over terms which depend on one document at a time. The word dependent contribution decomposes into a product over terms depend on a single topic each.

**Optimizing $\alpha$:** While in principle we could use a sampler to obtain $\alpha$ and $\beta$, it is much easier to employ simple (stochastic) convex optimization techniques for hyperparameter adjustment. In order for this to work we need to assume that data is provided in random order.[2] For convenience we denote by

$$\gamma(x) := \partial_x \log \Gamma(x) = \Gamma^{-1}(x)\partial_x \Gamma(x) \tag{4}$$

the derivative of the log-gamma function, sometimes also referred to as the *Digamma* function. Using (2) we obtain

$$\begin{aligned} &\partial_{\alpha_j} - \log p(w, z|\alpha, \beta) \quad (5)\\ =&\sum_{i=1}^{m} \gamma(\alpha_j) - \gamma(\alpha_j + n(t{=}j, d{=}i)) + \gamma(\bar\alpha + n(d{=}i)) - \gamma(\bar\alpha)\end{aligned}$$

Here the difference between the first two terms is nonvanishing only if $n(t, d) \neq 0$ — otherwise they are identical. This suggests a stochastic gradient descent procedure of the form

$$\alpha_i \leftarrow \alpha_i - \eta\Big[\underbrace{\gamma(\alpha_j) - \gamma(\alpha_i + n(t{=}i, d))}_{\text{evaluate only if } n(t\,=\,i,\,d)\,\neq\,0} + \underbrace{\gamma(\bar\alpha + n(d)) - \gamma(\bar\alpha)}_{\text{same value for all } i}\Big]$$

This is obtained simply by canceling out terms in denominator and numerator where $n(t, d) = 0$ and $n(t, w) = 0$ respectively. It allows us to evaluate the normalization for sparse count tables with cost linear in the number of nonzero coefficients. Moreover, it ensures that for sufficiently large collections of documents even a single pass suffices to obtain a good value of $\alpha$ ([10] use gradient descent which may be slow for large collections of data). Here $\eta_i = \frac{1}{\sqrt{\text{const.} + i}}$ is a decreasing step length.

Carrying out updates after each document is inefficient since the only meaningful signal in (5) occurs whenever a topic actually occurs in a document. To address this we aggregate gradients over a range $\tau$ of topics before carrying out updates (with a suitably rescaled step length).

**Optimizing $\beta$:** Unfortunately *stochastic* gradient descent is not applicable for optimizing $\beta$. In the simplest case we may assume a multiplicative form

$$\beta_w = \beta_0 \cdot \tilde\beta_w \text{ and hence } \bar\beta = \beta_0 \cdot B \text{ where } B = \sum_w \tilde\beta_w \tag{6}$$

where we can optimize over the overall smoothing weight $\beta_0$ by gradient descent or a suitable second-order method. Since the objective is convex this is guaranteed to converge.

The computation of the gradient, though, is rather costly — we need to sum over all topics and words with nonzero $n(w, t)$ in the topic-word table. This is best achieved at the same time as when performing likelihood computations since they, too, require a pass over $\beta$. We have

$$\begin{aligned}\partial_{\beta_0}\left[-\log p(w, z|\alpha, \beta)\right] = &\sum_{n(t,w)\neq 0} \tilde\beta_w \left[\gamma(\beta_w) - \gamma(\beta_w + n(t, w))\right]\\ &+ B \sum_{n(t)\neq 0} \left[\gamma(\bar\beta + n(t)) - \gamma(\bar\beta)\right]\end{aligned}$$

Updates of $\beta_0$ occur via $\beta_0 \leftarrow \beta_0 - \eta\partial_{\beta_0}\left[-\log p(w, z|\alpha, \beta)\right]$ for a decreasing update rate $\eta$.

## 2.4 Variational Optimization

At test time (once the model has been obtained) it is often desirable to have a fast mechanism for estimating the topic probabilities for a given document. We can express the likelihood of a given document via

$$p(w, \theta|\bar\alpha, \bar\Psi) \propto \prod_{j=1}^{n}\left[\sum_{i=1}^{k} \theta_i \bar\Psi_{iw_j}\right] \cdot \prod_{i=1}^{k} \theta_i^{\alpha_i} \tag{7}$$

where $\bar\Psi$ encodes smoothed probability estimates via $\bar\Psi_{tw} = \frac{\beta_w + n(t,w)}{\bar\beta + n(t)}$. Using an exponential families $\theta_l = \exp(\gamma_l - g(\gamma))$ where $g(\gamma) = \log \sum_j \exp \gamma_j$ and $\partial_{\gamma_j} \log \theta_l = \delta_{l,j} - \theta_j$ yields the following gradients of $p(w, \theta|\bar\alpha, \bar\Psi)$ with respect to $\gamma$:

$$\partial_{\gamma_l}\left[\ldots\right] = \sum_{j=1}^{n} \frac{\theta_l \bar\Psi_{lw_j}}{\sum_{i=1}^{k} \theta_i \Psi_{iw_j}} + \alpha_l - \left[n + \bar\alpha\right]\theta_l \tag{8}$$

which leads to the following update equations

$$\theta_l \leftarrow \left[n + \bar\alpha\right]^{-1}\Big[\sum_w n(d, w) \frac{\theta_l \bar\Psi_{lw}}{\sum_{i=1}^{k} \theta_i \Psi_{iw}} + \alpha_l\Big].$$

Here we rearranged the summation such as to perform only one update per distinctly occurring word $w_j$, thereby accelerating summation by a factor of 2-3. Note that the same trick can be applied to the collapsed Gibbs sampler, that is, sampling all topics for a given word in a row.

## 3. PARALLELIZATION

### 3.1 Design Considerations

In its uncollapsed form LDA is quite trivial to parallelize — simply sample from the topic assignments for all documents and subsequently (in a central pass) resample the topic priors and the word model. The problem is that this representation is slow mixing, hence the collapsed sampler.

Implementations such as Mallet [10] and the UCI LDA code [7] make a key approximation: given $k$ processors it is acceptable to partition a collection of $n$ documents into $k$ blocks which are processed independently. After each pass the document statistics are synchronized in a separate step. This approach has some disadvantages:

- The network remains unused while sampling proceeds. Subsequently peak demands on bandwidth are exerted.
- Due to a number of reasons (system, disk access, general job load, sampler burn-in) the time to process $k$ documents may differ widely. Waiting for the *last* processor to finish before synchronization can occur, introduces potentially long idle times.
- On multiprocessor systems this automatically leads to an $O(k)$ increase in allocated memory and thereby out-of-memory situations when many cores are involved.
- Partitioning creates delay between synchronizations.

[2]For instance, if we were to see all documents related to politics in sequence our instantaneous parameter choice for the topic prior $\alpha$ would become significantly biased towards related topics, thus slowing down convergence.

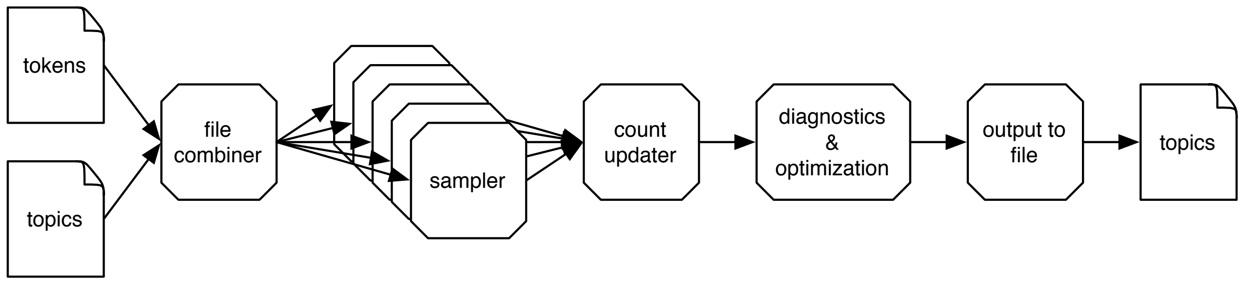


**Figure 2: LDA Pipeline. Each module in the pipeline is implemented as a filter and executes in parallel.**

We address this problem in two steps: firstly we introduce an approximation which allows us to decouple instant updates between different processor cores by a joint deferred update mechanism. Secondly, we introduce a blackboard-style architecture to facilitate simultaneous communication and sampling between different computers in a cluster environment. Both approximations allow us to perform sampling, updating, disk access, and network access *simultaneously* without the need for synchronization delay. In particular we will see that the memory requirement for $k$ processor cores is $O(1)$ and that, moreover, the communications load in the cluster setting is $O(1)$ for each workstation involved with an $O(1/k)$ overhead in memory allocation per machine.

## 3.2 Pipeline Architecture for Multicore

The key idea for parallelizing the sampler in the multicore setting is that the global topic distribution and the topic-word table (which we will refer to as *state* of the system) change only little given the changes in a *single* document (we may have millions of documents). Hence, we can assume that $n(t)$ and $n(t, w)$ are essentially constant while sampling topics for a document. This means that there is no need to update $n(t)$ and $n(t, w)$ during the sampling process and we can defer this action to a separate synchronization thread which takes action once a document has been entirely re-sampled. Consequently we can execute a large number of sampling threads *simultaneously.*

Figure 2 describes the data flow in the sampler. Words and topic assignments are stored in two separate files which are merged by the first filter. The combined documents are processed by a number of sampling threads executed in parallel. Each of these threads accesses the joint state variables $n(t)$ and $n(t, w)$ by acquiring a read lock before requesting their values. After processing an entire document, the list of changes in $n(t, w)$ and $n(t)$ is sent to the count updater filter. Since updates are considerably cheaper we found it sufficient to implement the latter in a single thread (there is no in-principle reason not to parallelize the updater thread as well, if required). While documents are being processed we can perform further diagnostics (e.g. we may compute the perplexity), and finally, a separate filter writes the new topic assignments to file. This has several advantages:

- We only need a single set of state variables $n(t, w)$ and $n(t)$ per computer rather than per core. This dramatically reduces the memory requirements per machine (relative to Mallet which keeps a copy per core — in our experiments Mallet reached its scalability limit at 300,000 documents and 1000 topics).
- The state is by definition always synchronized besides a minimal delay given by the documents that are being processed and whose new topic assignments are not yet integrated into the state table.
- It entirely avoids a second synchronization stage.
- The samplers never need to acquire write lock — they only *read* $n(t)$ and $n(t, w)$. Since our counters are 32 bit integers updates are atomic and consequently the updater usually can avoid acquiring write locks, thus dramatically reducing the number of samplers stalled due to lock contention. More on this in Section 4.

## 3.3 Blackboard Architecture for Clusters

When deploying LDA on multiple machines in a cluster we face the problem that it is impossible to keep the state table entirely synchronized between different computers. However, the strategy to synchronize only after each pass through data has a number of drawbacks, most importantly that all samplers need to wait for the slowest.

An alternative is to use a blackboard architecture similar in spirit to decomposition methods from optimization [4]. The key idea is to have a global consensus of the state variables and to reconcile their values one word at a time asynchronously for all samplers. The advantage is that no synchronization (short of locking the very word whose statistics are being updated) is required between samplers. Moreover, we can parallelize communication and storage by means of a distributed (key,value) storage using `memcached`. For $n$ servers and $n$ clients the network load is $O(1)$ per server and the memory requirements for storing a given amount of information over $n$ servers is $O(n^{-1})$.

We now specify the communications protocol in more detail: first, there is no need to synchronize $n(t)$ and $n(t, w)$ separately or even to store $n(t)$ globally at all. After all $n(t) = \sum_w n(t, w)$ and therefore any update on $n(t, w)$ can immediately be used to update $n(t)$. For the purpose of the algorithm we assume that at some point all samplers have the same identical state as the global state keeper.

Denote by $n(t, w)$ the current *global* state as stored in `memcached`, by $n^i(t, w)$ the current *local* state, and by $n^i_{\text{old}}(t, w)$ a copy of the *old local* state at the time of synchronization with the global state keeper. Then the following algorithm keeps the topic word counts synchronized.

Algorithm 1 incorporates any local changes in $n^i(t, w)$ that occurred since the last update into its global counterpart. Subsequently it sets $n^i$ to match the new consensus state and it updates $n^i_{\text{old}} = n^i$ to take a snapshot of the state variables at the time of synchronization. Since this process

**Algorithm 1** State Synchronization

```
Initialize $n(t,w) = n^i(t,w) = n^i_{\text{old}}(t,w)$ for all $i$.
while sampling do
    Lock $n(t,w)$ globally for some $w$.
    Lock $n_i(t,w)$ locally.
    Update $n(t,w) = n(t,w) + \left[n^i(t,w) - n^i_{\text{old}}(t,w)\right]$
    Update $n^i(t,w) = n^i_{\text{old}}(t,w) = n(t,w)$
    Update local $n^i(t)$.
    Release $n_i(t,w)$ locally.
    Release $n(t,w)$ globally.
end while
```

is happening one word at a time the algorithm does not induce deadlocks in the sampler. Moreover, the probability of lock contention between different computers is minimal (we have $> 10^6$ distinct words and typically $10^2$ computers with less than 10 synchronization threads per computer). Note that the high number of synchronization threads (up to 10) in practice is due to the high latency of `memcached`.

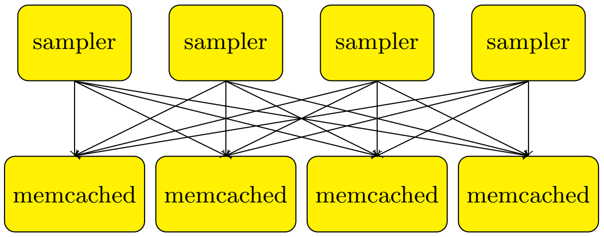


**Figure 3: Each sampler keeps on processing the subset of data associated with it. Simultaneously a synchronization thread keeps on reconciling the local and global state tables.**

Note that this communications template could be used in a considerably more general context: the blackboard architecture supports any system where a common state is shared between a large number of systems whose changes affect the global value of the state. For instance, we may use it to synchronize parameters in a stochastic gradient descent scenario by asynchronously averaging local and global parameter values as is needed in dual decomposition methods. Likewise, the same architecture could be used to perform message passing [1] whenever the junction tree of a graphical model has star topology. By keeping copies of the old messages local (represented by $n^i_{\text{old}}$) on the nodes it is possible to scale such methods to large numbers of clients without exhausting memory on `memcached`.

## 4. IMPLEMENTATION

### 4.1 Basic Tools

We use Google's `protobuf`[3] with optimization set to favor speed, since it provides disk-speed data serialization with little overhead. Since `protobuf` cannot deal well with arbitrary length messages (it tries loading them into memory entirely before parsing) we treat each document separately as a message to be parsed. To minimize write requirements we store documents and their topic assignments separately.

Data flow in terms of documents is *entirely local.* On each machine it is handled by Intel's Threading-Building-Blocks[4] library since it provides a convenient pipeline structure which automatically handles parallelization and scheduling for multicore processors. Locking between samplers, updaters, and synchronizers is handled by a read-write lock (spinlock) — the samplers impose a non-exclusive read lock while the update thread imposes an exclusive write lock.

The *asynchronous* communication between a cluster of computers is handled by `memcached`[5] servers which run standalone on each of the computers and the `libmemcached` client access library which is integrated into the LDA codebase. The advantage of this design is that no dedicated server code needs to be written. A downside is the high *latency* of `memcached`, in particular, when client and server are located on different racks in the server center. Given the modularity of our design it would be easy to replace it by a service with lower latency, such as `RAMCloud` once the latter becomes available. In particular, a *versioned* write would be highly preferable to the current pessimistic locking mechanism that is implemented in Algorithm 1 — collisions are far less likely than successful independent updates. Failed writes due to versioned data, as they will be provided in `RAMCloud` would address this problem.

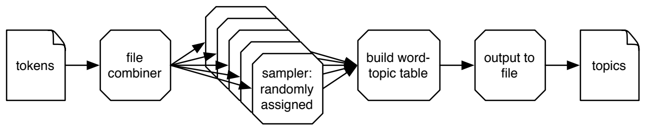
### 4.2 Data Layout

To store the $n(t,w)$ we use the same memory layout as Mallet. That is, we maintain a list of (topic, count) pairs for each word $w$ sorting in order of decreasing counts. This allows us to implement a sampler efficiently (with high probability we do not reach the end of the list) since the most likely topics occur first. Random access (which occurs rarely), however, is $O(k)$ where $k$ is the number of topics with nonzero count. Our code requires twice the memory footprint as that of Mallet (64bit rather than 32bit per (topic, count) pair) since for millions of documents the counters would overflow.

The updater thread receives a list of messages of the form (word, old topic id, new topic id) from the sampler for every document (see Figure 1). Whenever the changes in counts do not result in a reordering of the list of (topic, count) pairs and update is carried out *without* locking. This is possible since on modern x86 architectures updates of 32bit integers are atomic provided that the data is aligned with the bus boundaries. Whenever changes necessitate a reordering we acquire a write lock (any sampler using this word at the very moment stalls at this point) before effecting changes. Since counts change only by 1 it is unlikely that (topic, count) pairs move far within the list. This reduces lock time.

### 4.3 Initialization and Recovery for Multicore

At initialization time no useful topic assignment exists and we want to assign topics at random to words of the documents. This can be accommodated by a random assignment sampler as described in the diagram below:



In particular, the file combiner and the output routine are identical. Obviously this could be replaced with a more

[3]`http://code.google.com/p/protobuf/`

[4]`http://www.threadingbuildingblocks.org/`

[5]`http://www.danga.com/memcached/`

sophisticated initialization, e.g. by a system trained on a smaller dataset. When recovering from failure the multicore system simply loads the topic assignments from file and it rebuilds $n(t, w)$ and $n(t)$ with code identical to that used for initialization. The key difference is that obviously for this purpose no sampler is required.

## 4.4 Initialization for Cluster Parallelism

Our discussion in Section 3.3 assumed that at some point the state tables were synchronized. This requires synchronization between all clients. We use the following protocol:

**Local Initialization (stage 0):** We assume that initially all clients have a list of the IP numbers of all other clients involved.[6] At startup the clients set (IP, 'stage 0') as a (key, value) pair in `memcached`. Subsequently they independently build a local topic assignment table as described in Section 4.3.

**State Aggregation (stage 1):** Once the local statistics have been aggregated each machine proceeds by aggregating its local counts $n(t, w)$ with `memcached` on a per-word basis.

**Algorithm 2** Global State Aggregation

```
Set (IP, 'stage 1') on memcached
for all words $w$ on computer do
    Lock word $w$ on memcached globally
    Retrieve $n(t, w)$ from memcached
    Add local counts via $n(t, w) = n(t, w) + n_{\mathrm{local}}(t, w)$
    Write $n(t, w)$ to memcached
    Release lock on $w$
end for
```

Note that while each machine locally generates a dictionary to store a tokenized version of its documents for the purpose of a compressed representation, synchronization between machines occurs by using the words directly. That is, rather than synchronizing the topic counts for token 42 we synchronize the topic counts for the word 'hitchhiker'. The reason is that the dictionaries of local machines may differ widely and we want to avoid the need to synchronize them. Moreover, this way we can control the size of each local (token, word) dictionary simply by not allocating too many documents to each computer (the size of a unified dictionary would grow with the number of documents). This is particularly useful if different computers process different corpora: the local dictionaries can be much smaller than their union.

**Local State Synchronization (stage 2):** After stage 1 each computer sets (IP, 'stage 2') in `memcached` and starts polling `memcached` until all other computers on the cluster also have reached stage 2. This is important since only then we have the guarantee that all counts of all computers have been merged. After that we retrieve all $n(t, w)$ pairs that occur in the local collection and we set

$$n^i_{\mathrm{old}}(t, w) = n^i(t, w) = n(t, w)$$

**Sampling (stage 3):** After stage 2 each computer sets (IP, 'stage 3') in `memcached` and starts sampling. Note that there is no need to wait for all other computers to finish their local state synchronization. After all, if any global updates occurred they did not affect any state assignments of the client and therefore the state variables remain consistent. The code concludes by setting (IP, 'stage 4') to indicate completion of the algorithm.

[6]This is easily achieved by a suitable startup script or alternatively by registering its IP number with `memcached` with a known server.

## 5. EXPERIMENTS

In our experiments we investigate a number of aspects of our pipelined and `memcached`-based algorithm. There are three main questions that require answering: a) how well does the algorithm perform compared to existing implementations, b) does the model degrade with an increase in the number of computers, c) how scalable is the code. We begin with a competitive overview. Note that it is impossible for us to evaluate performance directly on many of the datasets used by competing algorithms since they are proprietary (e.g. Google's Orkut network). However, we compared our algorithm on Pubmed.

## 5.1 Performance Overview

In order to obtain a fair performance comparison we need to normalize throughput between different implementations. When in doubt, we upconverted the approximation in favour of the competing algorithms (e.g. document size, number of documents, number of topics). We normalize data to 1000 collapsed Gibbs sampling iterations on a documents per machine hour basis.

**PLDA:** The results reported in [9] were carried out on two datasets — a Wikipedia subset of 2.12 million documents, using 500 topics and 20 iterations, and the Orkut dataset of 2.45 million 'documents' using 500 topics and 10 iterations. The most favourable results were the throughput rates of [9, Table 6a] in the case of 16 machines — 11940s for 20 iterations on 2.12 million documents. This is equivalent to a per-machine throughput of 800 documents per hour and machine (at an average document size of 210 tokens, hence smaller than the news dataset we used in our experiments). The least favourable results are 65 documents per hour and machine (for 10 iterations on the forum dataset on 1024 machines). We (reasonably) assumed that an increase in the number of topics would only slow down the code.

**UC Irvine:** The results reported in [7] cover a number of datasets. Unfortunately, the authors focus mainly on speedup via parallelization rather than raw speed. The fixed number available was that for 2000 topics and 1024 processors it took 10 hours on 8.2 million documents. Note that the documents were quite short (less than 100 words per document and with a very limited vocabulary). Assuming comparable speed (IBM Power4+ 8 core) this amounts to a throughput of 6,400 documents per computer hour. [7, Sec. 5] also argue that their code would require 300 days on a single computer (incorporating the parallelization penalty).

**Our Codebase:** Since some of the datasets from [9] were unavailable and others (such as the NIPS collection) were too small for our purpose we used the following data for comparison purposes: a collection of 20 million news documents, each of them containing on average over 300 words and secondly the Pubmed collection, containing 8.2 million documents with an average length of 90 words each. Minimal processing was applied to the documents (we removed all non-ASCII characters and all words of two characters or less). For our experiments we used both workstations with server grade 8 core Intel CPUs of approximately 2GHz speed and a Hadoop cluster with similar configuration which was being used for production work during our exper-

| dataset | | 10k | 20k | 50k | 100k | 200k | 500k | 1m |
|---|---|---|---|---|---|---|---|---|
| pubmed | runtime (hours) | .28 | .43 | 1.03 | 1.33 | 4.42 | 7.48 | 15.12 |
| | initial # topics/word | 12.2 | 17.0 | 26.8 | 38.9 | 58.2 | 99.1 | 154.6 |
| | throughput (documents/hour) | 35.3K | 46.2K | 48.4K | 75.0K | 45.3K | 66.8K | 66.3K |
| news | runtime (hours) | .40 | .72 | 1.75 | 2.87 | 4.70 | 11.45 | 24.40 |
| | initial # topics/word | 10.4 | 12.9 | 16.5 | 19.4 | 22.1 | 24.8 | 25.7 |
| | throughput (documents/hour) | 25.0K | 27.9K | 28.6K | 34.9K | 42.5K | 43.7K | 41.0K |

**Table 1: Runtime for single machine execution (1000 topics for news, 2000 topics for pubmed, 1000 Gibbs sampler iterations each). The experiments were carried out on a dedicated 8-core workstation.**

| | pubmed | | | | news |
|---|---|---|---|---|---|
| computers | 10 | 20 | 50 | 100 | 100 |
| runtime (hours) | 17.2 | 9.0 | 4.1 | 2.8 | 12.5 |
| throughput (documents/hour) | 47.6K | 45.9K | 40.3K | 28.9K | 16.3K |

**Table 2: Runtime for multi machine execution (1000 topics for news, 2000 topics for pubmed, 1000 Gibbs sampler iterations each). The experiments were carried out on a production Hadoop cluster which was executing other jobs at the same time. The timing results for news are only reported on 100 nodes since the amount of memory required to store the (topic,word) table for a larger number of documents would have exceeded the amount of memory available per machine.**

| computers | | 1 | 2 | 5 | 10 | 20 | 41 | |
|---|---|---|---|---|---|---|---|---|
| pubmed | runtime (hours) | 3.2 | 4.2 | 4.1 | 4.2 | 4.4 | 4.8 | |
| | initial # topics/word | 58.2 | 107.1 | 209.1 | 318.6 | 472.6 | 679.6 | |
| | throughput (documents/hour) | 62.8K | 47.4K | 49.3K | 47.4K | 45.2K | 41.7K | |
| computers | | 1 | 2 | 5 | 10 | 20 | 50 | 100 |
| news | runtime (hours) | 4.6 | 7.5 | 7.9 | 8.1 | 9.0 | 10.9 | 12.5 |
| | initial # topics/word | 22.1 | 39.3 | 74.7 | 108.9 | 159.4 | 244.2 | 322.4 |
| | throughput (documents/hour) | 43.8K | 26.8K | 25.2K | 24.6K | 22.3K | 18.3K | 16.3K |

**Table 3: Runtime for multi machine execution (1000 topics for news, 2000 topics for pubmed, 1000 Gibbs sampler iterations each) when keeping the number of documents per processor fixed at 200,000.**

iments (hence we had no guarantee of exclusive ownership of the system). This is reflected in the slight fluctuations in throughput as seen in Table 3. Overall, on PubMed we achieved a throughput between 29k and 70k documents per hour and machine. In particular, for a comparable runtime of 9 hours our codebase is approximately 8x faster than the UCI implementation. This despite the fact that the system was being used for production work simultaneously without the guarantee of being able to use any of the nodes exclusively. On longer documents the performance results are similar. Note that document length is not as significant as expected. This is due to the decomposition of sampling effort into a document and word independent part and additional sparse parts which are cheap to compute.

## 5.2 Scalability

To test scalability we performed three types of experiments: a) we need to establish scalability in terms of the number of documents. b) we need to establish scalability in terms of a speedup in runtime as we increase the number of computers available. c) we need to show that as we have more computers we are able to process more data in a given time frame. The latter is the most relevant aspect in practice — as the amount of data in the server center grows we want to be able to increase our processing ability.

For the first experiment we ran LDA for 1000 Gibbs sampler iterations on Pubmed and the news dataset on a single 8 core workstation. A slight increase in per-document runtime is to be expected: as we obtain more documents the number of topics with nonzero $n(t, w)$ per word increases and with it the time spent in sampling. In fact, we see initial gains in scalability in Table 1 as we move to larger datasets.

A second experiment tested scalability by carrying out runtime experiments on a production Hadoop cluster. Since there was other regular activity ongoing while we ran our experiments (i.e. disk access, some background processing from other threads) we usually were not able to make full use of all 8 cores on the computers. Moreover, network connectivity between racks is less than 1Gb/s (our code was sharing the network with production jobs) and latency is increased due to the need to pass more than one switch. The latter adversely affects the synchronization time via `memcached`.

Finally, for the most realistic test (see Table 3) we fixed the number of documents per machine and measured throughput as a function of increasing sample size. The processing time per document increases considerably (by a factor of 2.5) as we increase the amount of data hundredfold and accordingly as we move from 1 computer to 100 computers. This is due to a number of reasons — the model becomes more complex (as can be seen by the increase in the initial number of topics assigned to each word). Secondly, we encounter more off-rack network traffic. To ensure sufficiently fast synchronization more threads need to be dedicated to communication with `memcached` rather than sampling. These additional threads increase the amount of cache misses for the samplers thus slowing them down. Thirdly, we switched from a single-machine scheme which did *not* require any network I/O to one which required network I/O.

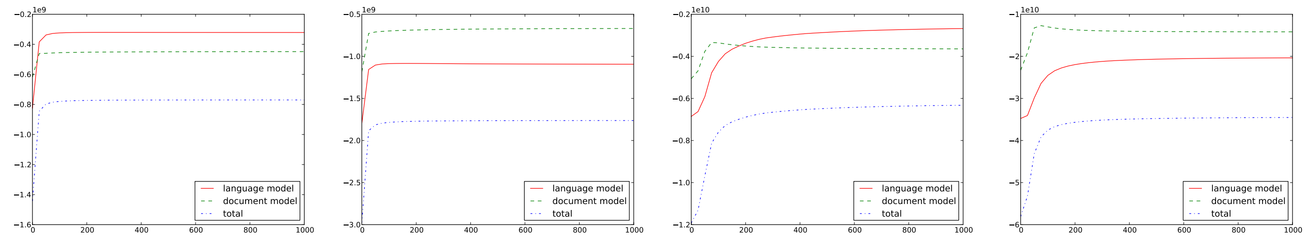


**Figure 4: Convergence properties for single and multi-machine LDA. The single machine results were carried out on 1 million documents whereas the multi-machine results were obtained on 100 machines on the full datasets. From left to right: (single machine, pubmed), (single machine, news), (multi machine, pubmed), (multi machine, news).**

## 5.3 Model Quality

Obviously there is no point in parallelizing inference if the model quality should suffer. Hence we computed the log-likelihood scores for increasing sample size. Using 2M documents (see Table 4) we see that the log-likelihood scores remain constant or possibly increase ever so slightly. This increase is likely due to the fact that (for reasons of convenience) we optimize over $\alpha$ separately for each computer, hence small changes in the distribution of topics between different chunks of data are likely exploited by slightly different optimal values of $\alpha$.

| computers | model | documents | total |
|---|---|---|---|
| 1 | -2.1136e+09 | -1.1946e+09 | -3.3082e+09 |
| 5 | -2.0812e+09 | -1.2143e+09 | -3.2954e+09 |
| 10 | -2.0681e+09 | -1.2110e+09 | -3.2880e+09 |
| 20 | -2.0631e+09 | -1.2238e+09 | -3.2869e+09 |

**Table 4: Log-likelihood for 2m news documents after 1000 sampling iterations.**

We see the latter as a feature of our system (rather than a defect): in practice it is not uncommon to receive data obtained from different sources (e.g. Wikipedia vs. high quality webpages vs. general web). While we may wish to analyze all data based on the same language model, it is quite likely that the distribution of topics differs between these sources. In this case, a different prior over topic distributions per group is a natural statistical modelling choice. Figure 4 shows convergence in log-likelihood for single machine and multi-machine runs. Note that initial convergence of the overall model is slightly slower since it takes some time to synchronize the language model between the computers — the document likelihood peaks around 25-50 documents. This is partly also due to the fact that we optimize the document model (i.e. the $\alpha$ parameters) only every 25 iterations.

## 6. SUMMARY AND DISCUSSION

In this work we proposed two novel parallelization paradigms for Latent Dirichlet Allocation: a decoupling between sampling and state updates for multicore and a blackboard architecture to deal with state synchronization for large clusters of workstations. We believe that of those two innovations the blackboard architecture is the more significant one as it is entirely general and can be used to address general large scale systems which share a common state. This work is complementary to recent progress on efficient inference in graphical models [5]. The latter focus on message passing algorithms where the entire model is small enough to fit into (distributed) main memory whereas our approach is specifically geared towards models where only an intersection of shared state variables needs to be exchanged and where the data considerably exceeds the amount of memory available for estimation.

In this sense a combination of [5] and the blackboard style approach presented in this paper are a good fit, allowing one to solve inference problems efficiently in memory whenever they are small enough to fit into main memory and to decompose the remainder via a set of tightly coupled (via asynchronous communication) cluster nodes.

### *Acknowledgments*

The authors thank Wray Buntine and James Petterson for valuable suggestions. This work is supported by a grant of the Australian Research Council. We plan on making our codebase available for public use.

## 7. REFERENCES

[1] S. Aji and R. McEliece. The generalized distributive law. *IEEE IT*, 46:325–343, 2000.

[2] A. Asuncion, P. Smyth, and M. Welling. Asynchronous distributed learning of topic models. In *NIPS*, pages 81–88. MIT Press, 2008.

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

[4] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, UK, 2004.

[5] J. Gonzalez, Y. Low, and C. Guestrin. Residual splash for optimally parallelizing belief propagation. In *AISTATS*, Clearwater Beach, FL, 2009.

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

[7] D. Newman, A. Asuncion, P. Smyth, and M. Welling. Distributed algorithms for topic models, NIPS 2009.

[8] H. Wallach, D. Mimno, and A. McCallum. Rethinking LDA: Why priors matter. *NIPS*, p. 1973–1981. 2009.

[9] Y. Wang, H. Bai, M. Stanton, W. Chen, and E. Chang. PLDA: Parallel latent dirichlet allocation for large-scale applications. In *Proc. of 5th International Conference on Algorithmic Aspects in Information and Management*, 2009.

[10] L. Yao, D. Mimno, and A. McCallum. Efficient methods for topic model inference on streaming document collections. In *KDD'09*, 2009.