Scalable Inference in Latent Variable Models

Amr Ahmed, Mohamed Aly, Joseph Gonzalez,[*] Shravan Narayanamurthy, Alexander Smola
Yahoo! Research, Santa Clara, CA, USA
{amahmed, aly, jegonzal, shravanm, smola}@yahoo-inc.com

ABSTRACT

Latent variable techniques are pivotal in tasks ranging from predicting user click patterns and targeting ads to organizing the news and managing user generated content. Latent variable techniques like topic modeling, clustering, and subspace estimation provide substantial insight into the latent structure of complex data with little or no external guidance making them ideal for reasoning about large-scale, rapidly evolving datasets. Unfortunately, due to the data dependencies and global state introduced by latent variables and the iterative nature of latent variable inference, latent-variable techniques are often prohibitively expensive to apply to large-scale, streaming datasets.

In this paper we present a scalable parallel framework for efficient inference in latent variable models over streaming web-scale data. Our framework addresses three key challenges: 1) *synchronizing the global state* which includes global latent variables (e.g., cluster centers and dictionaries); 2) *efficiently storing and retrieving the large local state* which includes the data-points and their corresponding latent variables (e.g., cluster membership); and 3) *sequentially incorporating streaming data* (e.g., the news). We address these challenges by introducing: 1) a novel delta-based aggregation system with a bandwidth-efficient communication protocol; 2) schedule-aware out-of-core storage; and 3) approximate forward sampling to rapidly incorporate new data. We demonstrate state-of-the-art performance of our framework by easily tackling datasets two orders of magnitude larger than those addressed by the current state-of-the-art. Furthermore, we provide an optimized and easily customizable open-source implementation of the framework [1].

Categories and Subject Descriptors

G.3 [**Probability And Statistics**]: Statistical Computing

[*]Visiting on internship from CMU, Department of Machine Learning, Pittsburgh PA; jegonzal@cs.cmu.edu
[1] Available at `https://github.com/shravanmn/Yahoo_LDA`

General Terms

Algorithms, Experimentation, Performance

Keywords

Inference, Graphical Models, Large-scale Systems, Latent Models.

1. INTRODUCTION

In many cases, we are interested in reasoning about the underlying latent causes that give rise to the data we observe. For instance, when dealing with users we may want to elicit the underlying intents and interests that govern their activity and friendship patterns. Alternatively, we might want to discover the underlying topics of discussions on various pages across the web. More generally we may want to assign meaning to linked and interacting objects such as webpages, named entities, users, and their behavior.

Latent variable models have become an indispensable tool for reasoning about the latent causes that give rise to data in tasks ranging from text modeling [18, 3, 6] to bioinformatics [8, 21]. The popularity of latent variable models stems from their ability to easily encode rich structured priors and then infer latent properties of the data without requiring access to costly labels or editorial feedback.

Latent variable models are constructed by introducing unobserved (latent) variables which help explain the observed data and then coupling these latent variables (often by introducing additional latent variables) to capture the underlying problem structure. For example, in the mixture of Gaussians model there are two sets of latent variables. The first denotes the cluster membership of each data point while the second describes the shape and position of the Gaussian clusters and introduces dependencies between the data-points.

Latent variable inference is the process of estimating the most likely assignment (or posterior distribution) of all the latent variables. In the context of the mixture of Gaussians model, latent variable inference is the process of estimating the latent membership of each data-point as well as the center and shape of each cluster. Inference in latent variable models is typically computationally expensive, often requiring the solution to hard combinatorial search problems. As a consequence approximate inference algorithms are typically employed. Unfortunately, even approximate inference algorithms can be costly, requiring iterative transformations of the latent variable assignments (i.e., transforming large amounts of program state).

In most web-scale settings, data is not collected once and then processed offline; instead, data arrives continuously and must be processed online and in real-time. The online setting presents unique challenges for latent variable models. As new data arrives new latent variables are introduced extending the model and requiring both the ability to quickly infer the values (or distributions) of the newly introduced latent variables as well as their effect on the existing model.

As a result of the computational cost of inference, the need to store and manage a massive amount of model state, and the added complexity of online data-processing, latent variable models are often abandoned in web-scale settings. However, [18] demonstrated that by exploiting massive scale parallelism through careful system engineering and the appropriate statistical approximations it is possible to apply latent variable techniques to web-scale problems.

In this paper we generalize the work of [18] by identifying the three principal challenges (Section 2) that must be addressed when applying any latent variable technique to web-scale problems. We then provide a general framework (Section 3) for addressing these challenges in a wide range of latent-variable models. Our general framework substantially extends [18] by introducing:

- A schedule aware disk-based cache for local (per-data) latent variables.
- A significantly improved general communications structure for asynchronously updating global variables.
- An efficient online algorithm for latent variable inference in streaming data.
- A new mechanism for fault-tolerance and fast recovery for large distributed global state.

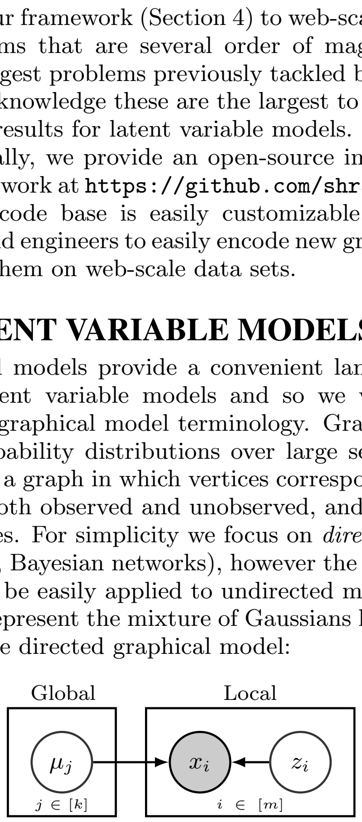
We apply our framework (Section 4) to web-scale latent variable problems that are several order of magnitude larger than the largest problems previously tackled by [18]. To the best of our knowledge these are the largest to date reported estimation results for latent variable models.

Additionally, we provide an open-source implementation of our framework at `https://github.com/shravanmn/Yahoo_LDA`. Our code base is easily customizable, allowing researchers and engineers to easily encode new graphicals model and apply them on web-scale data sets.

2. LATENT VARIABLE MODELS

Graphical models provide a convenient language for describing latent variable models and so we will introduce some basic graphical model terminology. Graphical models encode probability distributions over large sets of random variables as a graph in which vertices correspond to random variables, both observed and unobserved, and edges encode dependencies. For simplicity we focus on *directed* graphical models (i.e., Bayesian networks), however the techniques we present can be easily applied to undirected models.

We can represent the mixture of Gaussians latent variable model as the directed graphical model:



for m data points x_i. For each data point we introduce an unobserved latent variable z_i. In addition we introduce latent variables μ_j corresponding to the k cluster centers. We denote observed variables with shaded circles and unobserved (latent) variables with clear circles.

The boxes, referred to as *plates*, describe repeated (sets of) variables. For example, while there is only one circle μ_i the plate indicates that there are in fact k variables $\{\mu_1, \ldots, \mu_k\}$ all of which are connected to each of the $\{x_1, \ldots, x_m\}$. Note that each z_i is connected only to the corresponding x_i.

We have labeled the plates **Global** and **Local** to denote two general classes of latent variables. The global latent variables are those latent variables that are not directly associated with the data. The number of global latent variables is fixed or slowly increasing with the size of the dataset. Alternatively, the local latent variables are directly associated with each new record and grow directly with the dataset. There are often billions of local latent variables in web-scale problems.

The joint probability distribution for the mixture of Gaussian latent variable model is then given by:

$$\mathbf{P}(\{x_i\}_{i=1}^m, \{z_i\}_{i=1}^m, \{\mu_j\}_{j=1}^k) = \left(\prod_{j=1}^k \mathbf{P}(\mu_j)\right) \prod_{i=1}^m \mathbf{P}(z_i)\mathbf{P}(x_i \mid z_i, \{\mu_j\}_{j=1}^k)$$

where $\mathbf{P}(\mu_j)$ is the prior for each cluster center, $\mathbf{P}(z_i)$ is the latent cluster membership prior for each data-point, and $\mathbf{P}(x_i \mid z_i, \{\mu_j\}_{j=1}^k)$ is the probability of each data point given its cluster membership and the cluster centers.

Inference in the mixture of Gaussian model estimates the posterior conditional $\mathbf{P}(\{z_i\}_{i=1}^m, \{\mu_j\}_{j=1}^k \mid \{x_i\}_{i=1}^m)$ (or in many cases just the maximizing assignments). One of the most widely used interference technique is Gibbs sampling introduced by[9]. The Gibbs sampler iterates over the sets of latent variables, drawing one variable, or a small tractable subset of variables, at a time while keeping the remainder fixed. In the case of a Gaussian mixture model we would alternate between sampling:

$$\{\mu_j\}_{j=1}^k \sim \mathbf{P}(\{\mu_j\}_{j=1}^k \mid \{z_i\}_{i=1}^m, \{x_i\}_{i=1}^m) \tag{1}$$

$$\{z_i\}_{i=1}^m \sim \mathbf{P}(\{z_i\}_{i=1}^m \mid \{\mu_j\}_{j=1}^k \{x_i\}_{i=1}^m) \tag{2}$$

This process is *iterated a significant number of times* (depending on the complexity of the statistical model) until a sample that is approximately independent of the initial conditions can be drawn. Due to its popularity, performance, and general applicability we will focus on Gibbs sampling in this paper. However, many other approximate inference algorithm share similar computational patterns and would likely also benefit from the techniques discussed here.

In many cases the convergence of the Gibbs sampler can be greatly accelerated by analytically eliminating (collapsing) a subset of the latent variables. For example, if the normal inverse Wishart prior is placed on the Gaussian clusters, then we can analytically eliminate $\{\mu_j\}_{j=1}^k$ to construct the conditional $\mathbf{P}(z_i \mid z_{-i}, \{x_i\}_{i=1}^m)$ which couples each latent variable z_i with all the remaining latent variables z_{-i}. It is important to note that in most cases each variable z_i only depends on the remaining variable through sufficient statistics (e.g., the mean and variances). In this paper we will focus on the so called collapsed Gibbs sampler.

We now describe the three key challenges: 1) synchronizing the global state (e.g., $\{\mu_j\}_{j=1}^k$ or the sufficient statistics in the collapsed model); 2) efficiently storing and retrieving the large local state (e.g., $\{z_i\}_{i=1}^m$), and 3) sequentially incorporating streaming data (e.g., increasing (x_{m+1}, z_{m+1})).

2.1 Managing A Massive Local State

As discussed earlier the local state consists of the actual data and any latent variables directly associated with each data record (e.g., the data-points x_i and cluster memberships z_i). In web-scale applications the local state can be very large. For example, if we apply the popular LDA latent variable model to a web-scale text corpus with billions of documents we can end up facing in the orders of 10^{11} to 10^{12} local variables z_i (one for each unique word on each web-page) and corresponding textual data x_i. It is clearly infeasible to retain this amount of data in memory even on a very large cluster.

To make matters worse we will need to *repeatedly* read and *modify* the local latent variables z_i as we run our collapsed Gibbs sampler. At the same time our system must be robust to node failure which could lead to a partial loss of the local state z_i.

2.2 Synchronizing Global State

While the local state can be naturally partitioned over a large collection of distributed machines, the global state (e.g., the latent cluster center $\{\mu_j\}_{j=1}^k$ or the sufficient statistics in the collapsed modle) both depends on the assignments to the local variables. For example, in the mixture of Gaussians model, in order to estimate the value of each μ_j we need to know the location of all the datapoints x_i for which $z_i = j$ (i.e., the points assigned to cluster j). More generally, we will need to be able to estimate sufficient statistics over the entire cluster even as the values of the latent variables are changing and potentially new data is being introduced.

Even worse, as the Gibbs sampler redraws each local latent variable it will need access to the latest global variables (or the latests global sufficient statistics in the case of the collapsed Gibbs sampler). After each local variable is updated the global sufficient statistics and all other nodes on the cluster will need to be notified of the resulting change in the global statistics.

An effective solution to address these problem is to make local copies of the global variable and to keep these copies synchronized. Such a strategy is pursued e.g. by [18, 13, 17] using both synchronous and asynchronous communication patterns for reconciling local copies.

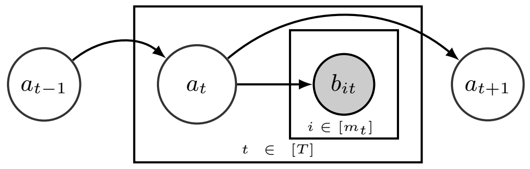
Matters are rather more complex whenever the global state is large. In this case aggressive synchronization may not be easily possible. In some cases, the global state may even be too large to be locally cached on a single machine. We will discuss both issues in the next section and show how a 'star of stars' communication architecture, paired with consistent hashing [12], and effective message ordering can be used to ensure good synchronization.

A final problem arises from the fact that modern microprocessors have many cores but the memory footprint of the models makes it impossible to store copies for each core separately. This requires that several cores access a shared state simultaneously. While read access is not a problem, simultaneous write access can lead to race conditions and correspondingly to unpredictable (and possibly wrong) state of the estimator. One means of dealing with this situation is to allow for diverging states and to decouple sampling from the application of the changes, both locally within a computer and between computers.

2.3 Online Data with Temporal Structure

A third problem arises when there is a significant sequential order inherent in the data which couples a lengthy chain of random variables. This problem is exacerbated whenever we wish to operate the system in a realtime setting where data is received continuously and needs to be annotated.

Example 1 (Latent Markov Chain) *Assume that a latent state (e.g. the distribution of words over topics, the interests of a user) exists that allows us to annotate a user's actions efficiently. Moreover assume that this latent change is allowed to change (smoothly) over time. Such a model can be represented as follows:*



Here we observe b whereas a is latent and can only be inferred indirectly by inverting the 'emissions' model.

Techniques for dealing with such problems can be found in the use Sequential Monte Carlo estimation and in the approximation of using only a single particle whenever the state space is too large to perform proper particle filtering.

3. GENERAL FRAMEWORK

In the following we discuss a set of general techniques for obtaining efficient parallel algorithms for inference in web-scale latent variable models. These techniques address the core challenges presented in the previous section. Most of the discussion will be devoted to the synchronization of global variables and the associated systems aspects.

3.1 Global variables on a single machine

We begin by discussing the role of approximation in exposing parallelsim in collapsed Gibbs sampling. Recall that the uncollapsed Gibbs sampler typically alternates between sampling global (Eq. (1)) and local (Eq. (2)) latent variables. Because all the local variables are typically conditionally independent given the global variables (and vice versa), we can construct the following parallelization:

```
1: Fix all $\{z_i\}_{i=1}^m$
2: for all $j \in [k]$ parallel do
3:     Sample $\mu_j \sim \mathbf{P}(\mu_j \mid \{z_i\}_{i=1}^m, \{x_i\}_{i=1}^m)$
4: end for
5: Fix $\{\mu_j\}_{j=1}^k$.
6: for all $i \in [m]$ parallel do
7:     Sample $z_i \sim \mathbf{P}(z_i \mid \{\mu_j\}_{j=1}^k \{x_i\}_{i=1}^m)$.
8: end for
```

While this approach is both statistically sound and exposes considerable parallelism, especially when m and k are large, it does not leverage the greatly accelerated mixing [11] of the collapsed Gibbs sampler. The problem is that in this case the random variables $\{z_i\}_{i=1}^m$ are no longer independent and we obtain the following *sequential* sampling algorithm:

```
1: for all $i \in [m]$ do
2:     Sample $z_i \sim \mathbf{P}(z_i \mid z_{-i}, \{x_i\}_{i=1}^m)$
3: end forv
```

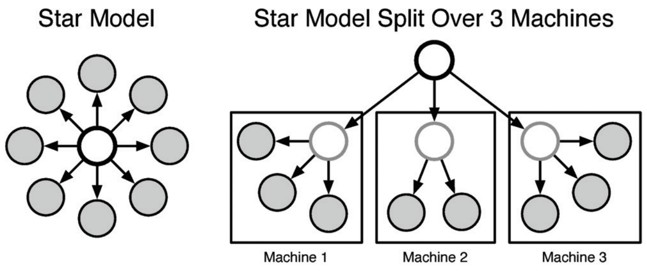
The above algorithm unfortunately is not amenable to multicore parallelization since only one variable may be changed at a time. However, if we are willing to admit a slight approximation we may parallelize this step in analogy to [15, 18, 5], simply by executing it for different z_i on several cores simultaneously:

```
1: Allocate a task pool $T$ of $m$ tasks
2: while $T$ not empty do
3:   if core available then
4:     Remove $i$ from $T$ and draw:
         $z_i \sim \mathbf{P}(z_i \mid z_{-i}, \{x_i\}_{i=1}^m)$
5:     Update sufficient statistics for $\{z_i\}_{i=1}^m$.
6:   end if
7: end while
```

Clearly the above is an approximation since z_i is drawn using partially stale sufficient statistics. However, whenever we have millions of z_i, the error in using a slightly stale distribution is negligible: the number of cores is small relative to the number of random variables. The only locking that occurs in this context is that writes to the sufficient statistics (as executed by an update thread) must not occur simultaneously as reads (as executed by the sampling threads). This, however, is much more benign since all but one thread only require read locks.

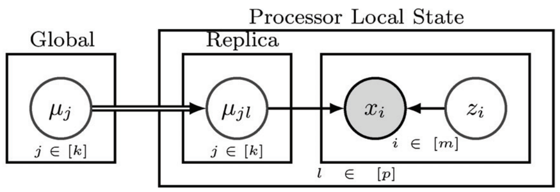
3.2 Global variables between machines

In principle, the same design decisions that guided the single core parallelism are also applicable to systems of many machines. However, what is a cheap in-memory operation for a multicore system becomes an excessively expensive operation when carried out across the network. Instead, we borrow an idea from dual decomposition methods [4] — we make local copies of the global variables and constrain the local copies to be consistent. The graph below explains the basic idea:



On the left we have a dependency structure with a shared global variable. On the right we inserted local copies (the clear vertices inside each box) per machine (the box) which are then synchronized with the original global variable. Note that this approach applies hierarchically whenever we face additional communications boundaries, e.g. when keeping sets of machines on different switches (or racks) synchronized.

We can actually express this transformed model in the context of a modified version of the original graphical model:



Here μ_{jl} denotes the *local* copy of the global variable μ_j. Note that our approach is quite different from [17] despite the similar structure. The key difference is that [17] assume that there exists a hierarchical generative process which led to divergent copies of the same random variable between different machines. In other words, they assume that the partitioning between machines is meaningful. Quite contrary to that, we simply assume that the variables may go out of sync due to insufficiently frequent communication between the machines. In other words, after complete synchronization we expect *equality* rather than just similarity between the global variable and its per-machine copies. This is made explicit by the double arrow connecting μ_j and μ_{jl}. To make the above concept practical we need to specify a) detailed update equations, b) a effective means of distributing global variables across many machines and c) a schedule to avoid communications overhead.

3.3 Asynchronous Delta Aggregation

The message passing system described by [18] suffers from high latency as a result of one sided communication through *memcached* (a distributed key-value store). In particular rather than using server-side transactional consistency, each node follows a remote lock-request-update-send-unlock protocol for each transaction. This introduces considerable latency since each of the lock, request, send, and unlock operations requires round-trip communication between the updating node and the hosting node. Furthermore, since many distributed key-value stores lack server-side processing, these problems cannot be addressed within the framework of [18] making it especially unsuitable for efficient synchronization of global state.

In addition the protocol described by [18] updates the global value on the client machine and then pushes the new global value back to the hosting machine. This leads to the need to potentially push more data than was changed in the update. Ideally, we would like to push only revisions to the global value (i.e., changes in components of the mean vector rather than the entire mean vector). With appropriate server-side processing we would like to be able to aggregate client side revisions before modifying the server-state.

We now present a substantially improved protocol which is over an order of magnitude faster in our experiments. Our new protocol applies whenever the global data μ_j is an element of an algebraic ring. Specifically we require that μ_j is closed under 'addition' and that an 'inverse' element exists. These conditions are satisfied for common operations like addition/subtraction in $\mathbb{R}$ $(+, \mathbb{R})$ or for multiplication/division in $\mathbb{R}^+$ $(\cdot, \mathbb{R}^+)$ allowing us to easily express the standard multiplicative or additive updates to s global sufficient statistic or latent variable estimates.

The algorithm below summarizes how the algebraic ring can be used to collect a change (δ) from a client, incorporate the change into the master version on the server node, and then update the clients to the change in the master value. We denote by μ_j the master value of the global variable. Moreover, μ_{jl} denotes the *local* copy of μ_j on machine l.

For the moment simply assume that there exists some easily identifiable server containing μ_j. Then the $(+, \mathbb{R})$ ring yields:

```
 1: Initialization
 2: for all $j \in [k], l \in [p]$ do
 3:   $\mu_{jl}^{\text{old}} \leftarrow \mu_{jl} \leftarrow \mu_j$
 4:   $s_{jl} \leftarrow$ FALSE (sent bit)
 5: end for
 6: ClientSend($l$)
 7: for all $j$ do
 8:   Acquire lock for $s_{jl}$
 9:   if $s_{jl} =$ FALSE then
10:     Acquire lock for $\mu_{jl}$
11:     Compute local change $\delta \leftarrow \mu_{jl} - \mu_{jl}^{\text{old}}$
12:     Release lock for $\mu_{jl}$
13:     Server($j, l, \delta$)
14:     $s_{jl} \leftarrow$ TRUE (we sent a message)
15:     $\mu_{jl}^{\text{old}} \leftarrow \mu_{jl}^{\text{old}} + \delta$ (locally commit changes)
16:   end if
17:   Release lock for $s_{jl}$
18: end for
19: ClientReceive($j, l, \mu^{\text{new}}$)
20: Acquire lock for $s_{jl}$
21: $s_{jl} \leftarrow$ FALSE (we received data)
22: Set write lock for $b_{jl}$
23: $\mu_{jl} \leftarrow \mu_{jl} + \mu^{\text{new}} - \mu_{jl}^{\text{old}}$ (get global changes)
24: $\mu_{jl}^{\text{old}} \leftarrow \mu^{\text{new}}$ (we are up to date)
25: Release write lock for $\mu_{jl}$ and $s_{jl}$
26: Server($j, l, \delta$) (buffered input)
27: $\mu_j \leftarrow \mu_j + \delta$
28: ClientReceive($j, l, \mu_j$)
```

Initially, the local copies and global state are set to the same values. Each client l cycles through its set of local copies and determines how much has changed locally since the previous communication round. It then transmits this information to a server and stores a backup of the local estimate. A sent bit s_{jl} ensures that no more than one message is 'in flight' from client to server. Note that the call to the server is *non blocking*, that is, we need not wait for the server to process the message. This fact is vital in designing synchronization algorithms which are bandwidth limited rather than latency limited (for comparison, [18] requires 4 TCP/IP roundtrips per variable).

The server incorporates the changes into its global state. This can be done efficiently since it only receives δ containing the change in state. Finally, it triggers a return message to the client. To see why the updates make sense assume for a moment that nothing changed after the client sent its changes: in this case we have $\mu_{jl}^{\text{old}} = \mu_{jl}$. Moreover, μ_j will contain all changes that locally occurred on μ_{jl} *and* all changes from other copies. Hence the update $\mu_{jl}^{\text{old}} \leftarrow \mu^{\text{new}}$ is valid. The update of μ_{jl} ensures that any local changes that might have occurred in the meantime are recorded and will be transmitted at the next step.

This approach dramatically reduces latency since the post to the server is non-blocking and the client is able to proceed processing using the freshly updated local copy. When the server completes the transaction it captures any additional changes made by other machines and then asynchronously notifies the other nodes in the cluster again with a non-blocking call. Because transactions are additive, they can be merged at bottleneckes in the network (e.g., switches) to reduce the necessary bandwidth between racks. Finally, for very large μ_j for which only small parts are changing, we need only send 'diffs' further reducing the bandwidth requirements.

3.4 Distribution and Scheduling

Now that we established *how* local copies of a global variable are kept synchronized we need to establish how to distribute the global copies over several servers: given p clients and k global variables the above algorithms creates $O(kp)$ traffic. If we select p servers (conveniently the same machines as the clients) we arrive at $O(k)$ traffic per server, provided that we are able to distribute the global variables uniformly over all servers. The latter is achieved by consistent hashing [12], i.e. by selecting server

$$s(j) = \operatorname*{argmin}_{s \in \{1,\ldots,p\}} \text{hash(s, j)}. \tag{3}$$

The advantage of (3) is that it does not require any additional storage to record the mapping of variables to servers. In expectation each server stores $\frac{k}{p}$ global variables. Since the assignment is randomized we can apply Chernoff bounds to show that with probability at least $1 - \gamma$ no machine receives more than $k/p + \sqrt{k(\log p - \log \gamma)}$. This follows from bounding the number of variables per server and by taking the union bound over p servers.

This strategy works well for up to $p = 100$ machines. At this point a secondary inefficiency becomes significant: Since each client is communicating with each server, it means that the amount of data exchanged between any pair of machines is $O(k/p)$. This means that the data *rate* between two machines *decreases* with $O(p^{-1})$ and that moreover the number of open network connections per machine increases with $O(p)$. Neither is good if we want to keep the communications overhead at a minimum (e.g. minimum packet size, collision avoidance, and sockets all contribute to a decreased efficiency as the rate decreases). In fact, for 1000 machines this overhead is quite substantial, hence the naive consistent hashing protocol requires an extension.

The key insight for improvement is that the synchronization occurs repeatedly and that we know for each machine which messages require sending beforehand. We therefore arrange the transmission of the messages in the synchronization algorithm of Section 3.3 such that we only communicate with one machine at a time. Consequently each client needs to keep only one connection to a server open at a time and we can transmit all data to be sent to this particular server jointly.

Unfortunately, this approach also has a subtle flaw: while we reduced the number of connections from p^2 to p, we obtain a rather nonuniform distribution in terms of the number of connections that each server has. For instance, if each client opens connections in a (different) random order, it follows that the open connections at any time are given by p draws with replacement from a set of p alternatives. Hence the probability of not receiving any data at all is given by $(1 - p^{-1})^p \approx e^{-1}$ per server. This is clearly inefficient — the problem is that we opened connections to *too few* servers. The solution is to communicate with $r \ll p$ servers simultaneously while selecting them in a random order.

```
1: RandomOrder($l, p$)
2: for $i = 1$ to $p$ do
```

```
3:     $f[i] = \text{crypt}(i, \text{range} = p, \text{key} = l)$
4: end for
5: return $f$
```

The above algorithm achieves precisely that by means of cryptography [14], e.g. via Feistel ciphers. The communication schedule for synchronization then works as follows:

```
1: ClientSend$(l, p)$
2: Generate $s(j)$ for all $j$ (compute machine keys)
3: Sort $j$ into sets $S_m$ with $s(j) = m$ if $j \in S_m$.
4: $f$ = RandomOrder$(l, p)$
5: for $i = 1$ to $p$ step $r$ do
6:     Variable pool $S = \cup_{v=i}^{i+r-1} S_{f[v]}$
7:     for all $w \in S$ do
8:         Perform communication with server as before.
9:     end for
10: end for
```

The main modification to the plain communication algorithm is that we now synchronize the global variables in a specific prescribed order which ensures that at any given time we only open r random connections per machine. This ensures that we have fewer idle servers.

Lemma 1 *The efficiency (as measured by the expected proportion of bandwidth used) of communicating with r machines at a time is bounded in the limit of large p by:*

$$1 - e^{-r} \sum_{i=0}^{r} \left[1 - \frac{i}{r}\right] \frac{r^i}{i!} \leq \text{Eff} \leq 1 - e^{-r}$$

Proof. The data flow is represented by a bipartite graph between p clients and p servers, where each client has r random connections to the set of servers. Randomness is assured by the random permutation over the order in which a given client connects to the servers, hence at any fixed instant in time the connectivity structure of each vertex is drawn iid from a uniform distribution over all servers. For any given server v the probability of not receiving any incoming edge is $\left(1 - \frac{1}{p}\right)^{pr} \to e^{-r}$. If each of the connected vertices receives data at full rate, this amounts to an efficiency of $1 - e^{-r}$. The maximally efficient communication order can be obtained by solving the optimal flow problem.

Now consider the model where the efficiency of the server is bounded by $\min(1, i/r)$. That is, the flow is linear in the number of incoming connections but never larger than the maximum flow. Using a Poisson approximation for the probability of receiving i incoming connections we see that $\mathbf{P}(i) = e^{-r} r^i / i!$ with expected number of incoming connections r. We can then compute the expected efficiency:

$$\begin{aligned}
\text{Eff} &\geq \sum_{i=0}^{\infty} \min\left(1, \frac{i}{r}\right) \mathbf{P}(i) = \sum_{i=0}^{\infty} \min\left(1, \frac{i}{r}\right) e^{-r} \frac{r^i}{i!} \\
&= e^{-r} \sum_{i=0}^{r} \frac{i}{r} \frac{r^i}{i!} + e^{-r} \sum_{i=r+1}^{\infty} \frac{r^i}{i!} \\
&= e^{-r} \sum_{i=0}^{r} \frac{i}{r} \frac{r^i}{i!} + 1 - e^{-r} \sum_{i=0}^{r} \frac{r^i}{i!} \\
&= 1 - e^{-r} \sum_{i=0}^{r} \left(1 - \frac{i}{r}\right) \frac{r^i}{i!} \qquad (4)
\end{aligned}$$

□

Plugging some numbers into the bounds obtained above shows that a small number of connections suffices to yield satisfactory efficiency:

r	1	2	3	4	5
lower bound	0.63	0.73	0.78	0.80	0.82
upper bound	0.63	0.86	0.95	0.98	0.99

3.5 Fault tolerance

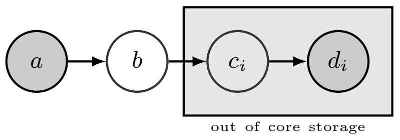
An important issue in large distributed systems is fault tolerance. For instance, a job executed on 1000 machines lasting one day assumes that the mean time between failures should significantly exceed 3 years to make such experiments feasible without fault tolerance.

[18] were only able to provide a rather weak guarantee due to the fact that in case of a machine failure the entire global state needed to be regenerated from scratch. This is very costly. Instead, we take advantage of the fact that, unlike when using *memcached*, we are able to perform operations directly on the data store of the server. One fact that complicates matters is that whenever the global synchronization is executed it requires one full pass through the entire set of variables before all local copies are in sync. This can be costly since synchronization may take several minutes, thus making writing a checkpoint an exceedingly slow matter.

Instead we use the following strategy: at any given time when we want to compute a checkpoint we *halt* all communication and sampling and wait until the incoming message queues have been processed (this is easily satisfied by reaching a barrier). At this point each client and server simply write their state to the cloud file system (HadoopFS in our case). This means that we are able to checkpoint the distributed synchronized state. Recovery is simply possible by loading the state from file system.

3.6 Local variables and out of core storage

Quite often, evidence is associated with local latent variables. For instance, in topic models we may want to obtain a topic estimate for each word. This leads to a state space which typically exceeds the amount of RAM available. An option to deal with this issue is to store part of the latent state out of core and to stream evidence and latent variables from disk whenever they are required. This approach is effective since the typical schedule for the Gibbs sampler iterates over the variables in a fixed order. Consider the following variant of the model of Section 2.1.



Here it suffices to traverse the local data sequentially.

```
1: for all $i$ do
2:     buffered read $c_i, d_i$ from storage
3:     sample $c_i | d_i, b$
4:     buffered write $c_i$
5: end for
```

Note that we only write the variable part of the state back to disc (the constant part is only read to avoid having to store the entire set of evidence nodes in memory). This strategy is very effective for document collections but less so, e.g. for highly interdependent entities such as social networks.

3.7 Sequential estimation

Often data comes with an additional sequential order as described in Section 2.3. Inference in this setting is possible, e.g. by the use of Sequential Monte Carlo estimation [7]. Given a model of the form

$$\mathbf{P}(a,b) = \mathbf{P}(a_0)\prod_t \mathbf{P}(a_t \,|\, a_{t-1})\mathbf{P}(b_t \,|\, a_t) \tag{5}$$

it is our goal to draw from $\mathbf{P}(a \,|\, b) \propto \mathbf{P}(a,b)$. In the following denote by $a^t := (a_0,\ldots,a_t)$ and $b^t := (b_1,\ldots,b_t)$ the sets of variables up to time t respectively. Assume that we draw from some proposal distribution

$$a_t \sim \pi_t(a_t \,|\, a^{t-1}, b^t). \tag{6}$$

In this case we need to re-weight the draw by

$$\beta := \frac{\mathbf{P}(a \,|\, b)}{\prod_t \pi_t(a_t \,|\, a^{t-1}, b^t)} = \frac{\mathbf{P}(a_0)}{\pi_0(a_0)}\prod_t \frac{\mathbf{P}(b_t \,|\, a_t)\mathbf{P}(a_t \,|\, a_{t-1})}{\pi(a_t \,|\, a^{t-1}, b^{-t})}$$

One may show [7] that the minimum variance draw is achieved by using as proposal distribution π_t the posterior $\mathbf{P}(a_t \,|\, a^{t-1}, b^t)$. Moreover, we set $\pi_0(a_0) = \mathbf{P}(a_0)$. By Bayes rule we have

$$\mathbf{P}(a_t \,|\, a^{t-1}, b^t) = \frac{\mathbf{P}(b_t \,|\, a_t)\mathbf{P}(a_t \,|\, a_{t-1})}{\mathbf{P}(b_t \,|\, a^{t-1}b^{t-1})} \tag{7}$$

Plugging this expansion back into β yields

$$\beta = \prod_t \mathbf{P}(b_t \,|\, a^{t-1}b^{t-1}). \tag{8}$$

In other words, we re-weight particles according to how well they explain future data while adjusting the parameter estimate at each step once we observe the evidence. [1] use this model to estimate topics and clusters on streaming news.

This procedure is carried out for a number of 'particles' which provide a discrete approximation of the posterior distribution over the latent variables. The computational cost scales linearly in the number of particles. However, we face two challenges: firstly every time some particles get too heavy we are forced to re-sample. This requires an update of the data structures. The latter is highly nontrivial since the state is distributed over many computers. Second, computing $\mathbf{P}(b_t \,|\, a^{t-1}, b^{t-1})$ is costly. Third, due to concentration of mass effects the behavior of the particles is quite similar ([1] show that even a small number of particles performs well).

Consequently we resort to a more drastic approximation: we only choose a single particle which avoids the need to track and recompute β. Such an approximation is justified for a number of reasons: firstly, we are primarily interested in the present estimate of a nonstationary distribution. For a contractive mapping the contribution of past errors decreases, hence the error arising from using a single particle diminishes. Finally, the single-particle approach is used in situations where the state of a particle is significant (e.g. all user actions of a given day). In this case concentration of measure effects allow us to show, e.g. via martingales, that all particles would perform extremely similarly, simply since the aggregate state is sufficiently large. Consequently, the single particle approach renders the problem amenable to practical implementation. This yields the following algorithm.

```
1: for $t = 1$ to $T$ do
2:     Sample $a_t \sim \mathbf{P}(a_t \,|\, a^{t-1}, b^t)$
3: end for
```

It proceeds in stages, using past latent and observed variables to sample the current set of parameters. We should stress here that the primitive Sample $a_t \sim \mathbf{P}(a_t \,|\, a^{t-1}, b^t)$ means perform a few Gibbs iterations over the Markov chain constructed over (a_t) where a_t is usually a set of hidden variables as we will show in Section 4.Running a Gibbs sweep over a_t using the state of the sampled values form earlier time steps (a^{t-1}) has been reported to work well in the literature [19, 20, 2, 1] especially if a_t depends on (a^{t-1}) via aggregates – which is true in many web applications. In other words, we design our sequential estimation primitives to perform well under the characteristics of our target applications.

4. APPLICATIONS

We now show based on a number of use-cases that the approximations above are effective at obtaining scalable estimates. More specifically, we give details on the improved variable replication strategy described in the previous section. Subsequently we show how the three primitives for dealing with local state, global state, and sequential order can be used in the context of dynamic user profiling.

4.1 Message Passing

To demonstrate the efficacy of the message passing scheme of Section 3.2 we briefly compare the algorithm described in [18] to the algorithm of Section 3.2. That is, we compare the performance of Yahoo's LDA implementation using both synchronization schemes.

We used 20 million news articles from Yahoo's news article collection, 100 computers, and 1000 topics. We performed 1000 iterations of collapsed Gibbs sampling for the documents and synchronization was carried by one of the following schedules:

Memcached: We use the schedule of [18]. That is, sufficient statistics are stored in *memcached*, a distributed (key,value) storage. Since *memcached* does not provide locking or versioning we emulate this as follows:

```
1: Set lock on memcached (fails if lock exists)
2: Get data from memcached
3: Update local values (no TCP/IP needed here)
4: Send data to memcached
5: Release lock record
```

As is evident this requires 4 TCP/IP roundtrips to update a single record. This means that the algorithm is considerably latency bound. We mitigate this effect by performing parallel synchronization with several threads simultaneously.

Ice: We implemented our own distributed (key,value) storage as part of the LDA codebase. To ensure that client and server remain synchronized we limit the number of tokens in flight at any given time, i.e. we limit the number of outstanding sent requests s_{jl} = TRUE that a client can have, e.g. $\sum_j s_{jl} \leq 100$. This is desirable since we want to ensure that messages are recent rather than being queued up for a long time. For communication we use the *ice* inter process communication library of zeroc.com.

Below we show how the time to complete a single synchronization pass change as we increases the number of machines while fixing the dataset characteristic. As we can see, the new architecture presented in this paper is actually able to leverage the increased number of machines more efficiently.

Protocol	*old*[18]			*new* (this paper)		
Machines	100	200	400	100	200	400
Sync time (sec)	155.61	131.47	149.14	33.2	28.22	24.12

The improved synchronization rate has additional desirable benefits — it significantly accelerates convergence of the sampler, as can be seen in Figure 1. More to the point, the sampler converges up to 5 times faster, due to an improved synchronization strategy: the negative log-likelihood using *ice* at 100 iterations is higher than what *memcached* reaches after 500 iterations. In other words, not only is the new scheme faster *per iteration*, it also converges faster, hence we may terminate the sampler earlier. Moreover, as evident from the Figure, as we increase the number of machines, the difference in performance gap between the old and new architecture increases significantly which suggests that the new architecture is more suitable for large-scale web application using large number of machines.

4.2 Message Scheduling

Next we tested the efficiency of the communication scheduling algorithm described in Section 3.4. For this purpose we used 150 machines and a dataset of 20 million documents with a vocabulary size of 3.5 million words. In particular, we compared the average time required for a synchronization pass when using no schedule and when using aggregate schedules for various aggregation factors p.

Schedule	none	$p = 2$	$p = 5$	$p = 10$
Sync time	231s	227s	**209s**	233s

As can be seen, using 5 machines at a time yields a 10% speedup in synchronization. This is desirable in its own right, however it also ensures somewhat faster convergence due to faster mixing. The reason that $p = 2$ and $p = 10$ show essentially no effect can be explained by a lack of enough connections (for $p = 2$) and overhead due to thread management and TCP/IP packet inefficiencies. We believe that the throughput can be improved further by adjusting the operating parameters of *ice* to force serialization of several messages into a single wire packet.

4.3 Temporal user profiling

The second application described in this paper is that of temporal user profiling. Here the goal is to obtain an estimate of the interest distribution of a user where the latter keeps on changing over time. The associated graphical model is given in Figure 2. Essentially each user i has an interest distribution θ_{it} which is a function of time t and which changes slowly. At any given moment the user draws a particular interest z_{itj} from this interest distribution and then he draws an action w_{itj} from the associated activity distribution ϕ_{tk}. Likewise the interest prior Ω_t depends on prior activity through the count variables m_t. The model is considerably more complicated than a conventional topic model due to the fact that now user interest, activity distribution and general topic distribution all become time-dependent random variables. Figure 2 provides a simplified version. For details of the full model see [2].

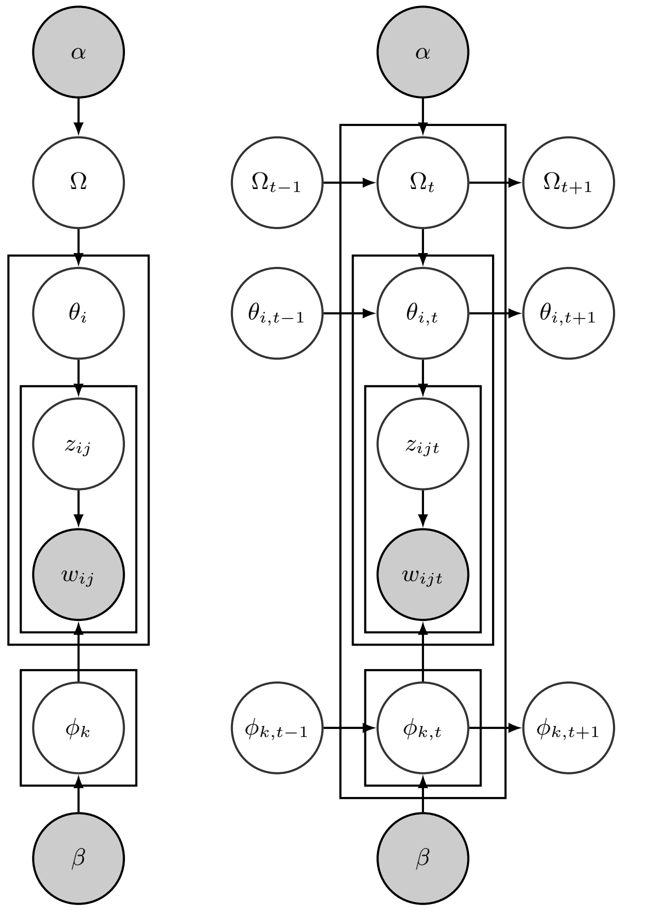


Figure 2: Left: Latent Dirichlet Allocation. Right: a simplified variant of Temporal LDA [2]. The key difference between both models is that in Temporal LDA, the prior over topics, the actual topic distribution and the activity distribution per topic depend on the past assignments which are summarized as m_t, n_t and β_t.

Scalability

The key motivation for discussing the model is that it demonstrates the interaction between all previously described approximations. In particular, we use the distributed LDA sampler described above as the main building block to perform inference on a daily basis. The state of the sampler comprises the topic-word counts matrix, and the document-topic counts matrix. The former is shared across users and is maintained in a distributed hash table as described in Section 3.2. The latter is document-specific and can be maintained locally in each client. We distribute the users at time t across N computers. Each client executes a foreground thread to resample a user for a given day and a background thread synchronizing its local copy of the topic-word counts matrix with the global copy.

Figure 3 confirms one of the most basic requirements of scalable inference — the amount of time required to process data scales *linearly* with the amount of data when keeping the computational resources fixed. Furthermore it also shows that while our algorithm does not scale perfectly, we lose only a small amount of performance when increasing

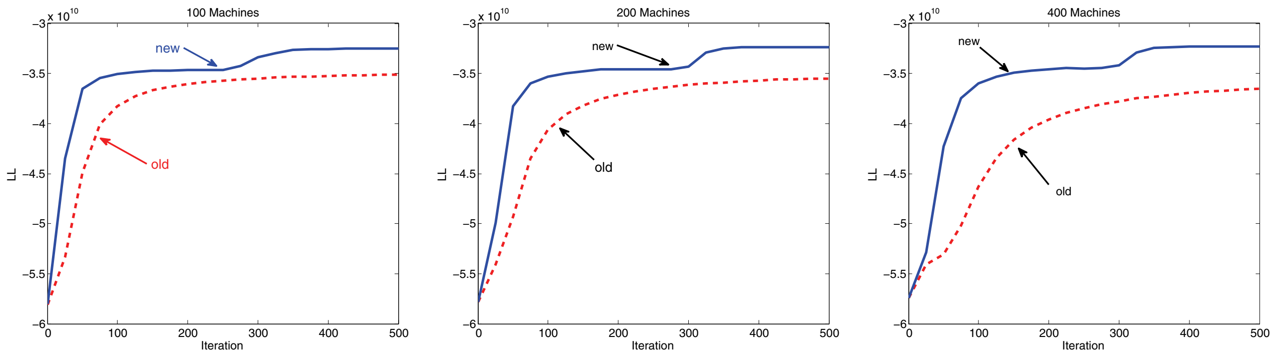


Figure 1: Log-likelihood of topic assignments during the sampling iterations. We compare the *old* architecture [18] and the *new* architecture described in this paper using LDA on 20 million news articles and1000 topics. We report results over 100, 200 and 400 machines. Note that the improved sampler converges approximately 5 times faster than the old *memcached* based variant. Moreover note that this gap increases as the number of machines increases, which suggests that the new architecture is more scalable than the old one

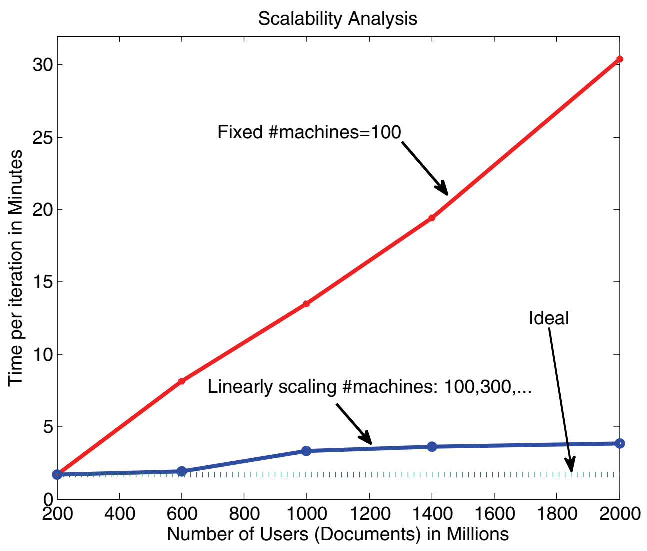


Figure 3: Time per iteration vs. millions of user days. When fixing the number of machines we observe a linear dependence between iteration time and the amount of data. The blue line depicts the time per iteration when keeping the data to machine ratio constant. That is, we use $\{100, 300, 500, 700, 1000\}$ **machines such that each machine receives** 2 **million user days.**

the data tenfold. We believe that some of the associated inefficiencies are due to the fact that the experiments were carried out on a production cluster where it would have been more difficult to allocate 1000 machines in close proximity on Hadoop. Hence the communications overhead arising from inter-rack communication would have had a significant impact on the runtime. Overall, the results show that we are able to process the equivalent of 2 billion documents (i.e. user days) on 1000 machines in a satisfactory time.

Resampling global parameters

As shown in Section 3.7, after each day (for each value of t), the random variables Ω_t should be sampled. However, Ω_t requires a *reduction* step across clients since it depends on the joint counts from all users across all clients. This is a classical Reduce operation. Using the distributed (key, value) storage it is implemented as follows: First each client writes its contribution for m_t based on its assinged users to the master, then the clients reach a *barrier* where they wait for each other to proceed to the next stage of the cycle. We implemented a sense-reversing barrier algorithm which arranges the nodes in a tree and thus has a latency that scales logarithmically with N [16]. After this barrier, all counts are in global storage and as such one client sums these counts to obtain m_t, use it to sample Ω_t, and finally writes it back. All other clients wait, then they read the new value of Ω_t and this finalizes a single cycle.

Performance

While desirable in terms of speed, we also need to resolve whether the aforementioned modifications affect the quality of the estimates. For this purpose we compared the results from [2] (old) to the current sampler (new) when applied to the problem of temporal user profiling. In other words, we did not change the model significantly but rather only changed the sampler to allow for faster synchronization.

For evaluation purposes we compute the ROC score when using simple bag of words features (BOW) as baseline, features from temporal LDA (TLDA), and a combination of both features (BOW+TLDA).

Feature set	BOW	TLDA		TLDA+BOW	
Variant		old [2]	new	old [2]	new
ROC score	57.03	58.70	60.2	60.38	**63.8**

The data we used for this experiment were 435GB of behavioral data spanning 44 days with 33.5 million users. It clearly shows that improving speed *also* improves performance in our case, thus giving us the best of both worlds.

Fault Tolerance

Finally, we need to address the issue of fault tolerance and recovery from failure. We described an improved mechanism of writing backup copies per server locally. As can be seen, this leads to a dramatic improvement in terms of recovery time by at least one order of magnitude. Since the time is primarily dominated by the vocabulary size and the number of topics we compared the times for a number of parameter combinations.

words	topics	machines	old [2]	new
1.5M	100	100	25.1 min	3 min
3M	100	100	64.2 min	3.5 min
5M	200	300	80.2 min	6.5 min

This improvement in terms of fault recovery (and correspondingly reduced time for taking snapshots of the current state) allows us to perform backups much more eagerly, thus allowing for a more stable production system.

5. DISCUSSION

In this paper we presented a number of powerful strategies for performing approximate inference on web-scale data collections. While quite effective in isolation, their joint use opens the door to the use of very sophisticated latent variable models in actual production settings in industry. To the best of our knowledge, the results presented here are on significantly much larger problems than the previously largest experiments of [18, 2].

To ensure widespread dissemination of these ideas we released an open-source implementation of our framework at `https://github.com/shravanmn/Yahoo_LDA`, together with the full documentation of the framework architecture, its different components, the steps to run it, and the mechanism of exapnding it with new graphical models. We believe this framework has the great potential of allowing, both researchers and practitioners, of easily tackling web-scale inference problems with little coding effort.

Note that the approximations are quite orthogonal to the approach of [10, 13] which focus heavily on graph partitioning for general graphs which are small enough to be retained in main memory and on efficient message passing schedules. Future work will be to combine these techniques with our approximations. We have reason to believe that this will yield an enterprise ready graphical model inference toolkit which is capable of analyzing data from hundreds of millions of users in social networks and billions of webpages.

Acknowledgments

We thank Yucheng Low and Tamas Sarlos for fruitful discussions. This work was supported by the ARC.

6. REFERENCES

[1] A. Ahmed, Q. Ho, C. H. Teo, J. Eisenstein, A. J. Smola and E. P. Xing. Online Inference for The Infinite Topic-cluster model: Storylines from Streaming Text. In *Artificial Intelligence and Statistics AISTATS*, 2011.

[2] A. Ahmed, Y. Low, M. Aly, V. Josifovski, and A. Smola. Scalable inference of dynamic user interests for behavioural targeting. In *Knowledge Discovery and Data Mining*, 2011. submitted.

[3] A. Asuncion, P. Smyth, and M. Welling. Asynchronous distributed learning of topic models. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *NIPS*, pages 81–88. MIT Press, 2008.

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

[5] J. K. Bradley, A. Kyrola, D. Bickson, and C. Guestrin. Parallel coordinate descent for l1-regularized loss minimization. In *International Conference on Machine Learning ICML*, Bellevue, WA, 2011.

[6] W. Chen, D. Zhang, and E. Chang. Combinational collaborative filtering for personalized community recommendation. In Y. Li, B. Liu, and S. Sarawagi, editors, *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 115–123. ACM, 2008.

[7] A. Doucet, N. de Freitas, and N. Gordon. *Sequential Monte Carlo Methods in Practice*. Springer-Verlag, 2001.

[8] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic models of proteins and nucleic acids*. Cambridge University Press, 1998.

[9] S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6:721–741, 1984.

[10] J. Gonzalez, Y. Low, A. Gretton, and C. Guestrin. Parallel Gibbs Sampling: From Colored Fields to Thin Junction Trees. In *Artificial Intelligence and Statistics AISTATS*, 2011.

[11] T. Griffiths and M. Steyvers. Finding scientific topics. *Proceedings of the National Academy of Sciences*, 101:5228–5235, 2004.

[12] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Symposium on the Theory of Computing STOC*, pages 654–663, New York, May 1997. Association for Computing Machinery.

[13] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence*, 2010.

[14] M. Luby and C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing*, 17(2):373–386, 1988.

[15] W. Macready, A. Siapas, and S. Kauffman. Criticality and parallelization in combinatorial optimization. *Science*, 271:56–59, 1996.

[16] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, Feb. 1991.

[17] D. Newman, A. Asuncion, P. Smyth, and M. Welling. Distributed algorithms for topic models. *Journal of Machine Learning Research*, 10:1801–1828, 2009.

[18] A. Smola and S. Narayanamurthy. An architecture for parallel topic models. In *Very Large Databases (VLDB)*, 2010.

[19] T. Iwata, T. Yamada, Y. Sakurai, and N. UedaOnline multiscale dynamic topic models. In *KDD*, 2010.

[20] N. Bartlett, D. Pfau, and F. Wood . Forgetting counts : Constant Memory inference for a dependent hierarchical Pitman-Yor Process In *ICML*, 2010.

[21] E. Xing, M. Jordan, and R. Sharan. Bayesian haplotype inference via the dirichlet process. *Journal of Computational Biology*, 14(3):267–284, 2007.