Multicore Gibbs Sampling in Dense, Unstructured Graphs

Tianbing Xu
Department of Computer Science
University of California, Irvine

Alexander Ihler
Department of Computer Science
University of California, Irvine

Abstract

Multicore computing is on the rise, but algorithms such as Gibbs sampling are fundamentally sequential and may require close consideration to be made parallel. Existing techniques either exploit sparse problem structure or make approximations to the algorithm; in this work, we explore an alternative to these ideas. We develop a parallel Gibbs sampling algorithm for shared-memory systems that does not require any independence structure among the variables yet does not approximate the sampling distributions. Our method uses a look-ahead sampler, which uses bounds to attempt to sample variables before the results of other threads are made available. We demonstrate our algorithm on Gibbs sampling in Boltzmann machines and latent Dirichlet allocation (LDA). We show in experiments that our algorithm achieves near linear speed-up in the number of cores, is faster than existing exact samplers, and is nearly as fast as approximate samplers while maintaining the correct stationary distribution.

1 Introduction

As new computer architectures move increasingly toward parallel computing structures such as multi- and many-core architectures, machine learning algorithms must adapt to understand and take advantage of the theoretical implications of parallel computing. Parallel computing has garnered considerable attention in learning (Chu et al., 2006) and in inference using message passing algorithms (Gonzalez et al., 2009a,b) and Markov chain Monte Carlo techniques (Newman et al., 2009; Ren and Orkoulas, 2007). Although in many cases the required tasks are trivially ("embarassingly") parallel, in others the obvious methods of parallelization cannot compete with a well-designed sequential algorithm, and more carefully designed parallel algorithms must be constructed (Martens and Sutskever, 2010; Whiley and Wilson, 2004).

The class of Markov chain Monte Carlo (MCMC) algorithms provide many such examples. MCMC is a fundamentally sequential set of operations; for example, in Gibbs sampling each variable is sampled from its conditional distribution given the previously drawn samples. One obvious method to parallelize MCMC is to construct multiple Markov chains, each of which runs in parallel. After each processor has progressed past the burn-in stage, each will provide its own sequence of independent samples. More samples typically provide better estimates; for P processors, we can obtain the same quality estimates with only $1/P$ samples per process, or reduce variance by a factor of $\sqrt{P}$. Unfortunately this means every processor must pay the same fixed cost of burn-in, which in many problems is the most costly aspect: only a few samples are needed, but burn-in takes a long time.

Another straightforward method is to use independence structure among variables, often encoded using a graphical model, to decouple samples. In a Markov random field, each variable is independent given its neighbors in the graph. If each variable has only a few neighbors, its sampling process will depend on only those variables, and other variables can be sampled concurrently by other processes so long as they avoid direct interaction. This is particularly effective in models with regular, repeated structure which enable large groups of variables to be scheduled concurrently without creating dependencies (Ren and Orkoulas, 2007).

However, the most difficult class of models are those which are densely connected. Examples from the literature include Gaussian processes (Whiley and Wilson, 2004), Boltzmann machines (Martens and Sutskever,

Appearing in Proceedings of the 14^{th} International Conference on Artificial Intelligence and Statistics (AISTATS) 2011, Fort Lauderdale, FL, USA. Volume 15 of JMLR: W&CP 15.

2010), and the latent Dirichlet allocation (LDA) model for text mining (Blei et al., 2003). In these cases, each sample depends on many or all of the other values, making it difficult to find large collections of variables which can be sampled independently. One possible solution (Newman et al., 2009) is to define an *approximate* sampler, which draws its samples from a probability distribution which is close to the correct distribution, but approximated in a way that enables parallel sampling. The results of such algorithms are often excellent in practice, and achieve nearly linear speed-up in the number of processes. One significant problem, however, is that we have no real idea how close our sampling distributions are to the desired model. Is this the best we can hope for, or can we adapt our algorithms such that we manage to fulfill both roles: drawing samples from the desired model, while still taking advantage of parallel computing?

In this paper, we describe a new mechanism for performing Gibbs sampling in densely connected models, using two dense graphical models (Boltzmann machines and LDA) as case studies. Our method is specialized to shared memory parallel architectures such as multicore desktop machines, in which coordination among processes and data sharing are relatively efficient. It works by constructing a *look-ahead* sampler for each process. In the first part, the sampler attempts to draw its value without knowing the samples of preceeding processes; if this succeeds, it can continue without waiting. If it fails, however, it waits until the samples scheduled before it are complete, then finishes its own sampling operations.

2 Motivating examples

We first describe the models and Gibbs sampling updates for two motivating example problems with densely structured graphs: Boltzmann machines, used widely in machine learning, and the latent Dirichlet allocation (LDA) model for text.

2.1 Boltzmann Machines

A Boltzmann machine is a binary-valued, pairwise Markov random field, defined by a quadratic energy function and its associated Boltzmann distribution,

$$E(z) = -z'Az \qquad p(z) \propto \exp(-E(z)) \qquad (1)$$

where $z = [z_1, \ldots, z_n]'$ and $z_i \in \{0, 1\}$. We assume that A is symmetric, so that $a_{ij} = a_{ji}$. (The formulation often includes a bias term $b'z$ in the energy, but since z_i is binary, $b_i z_i = z_i b_i z_i$ and b can be absorbed into the diagonal elements of A.) The standard Gibbs sampler proceeds by sampling from the conditional distributions

$$p(z_i = 1 \,|z_{\neg i}) = \sigma(a_{ii} + \sum_{j \neq i} 2\, a_{ij} z_j)$$

where $\sigma(x) = 1/(1+e^{-x})$ is the logistic function. If the matrix A is dense, z_i will depend on a large number of z_j, making parallel computation difficult.

An alternative proposed by Martens and Sutskever (2010) is to construct a partially continuous, *restricted* Boltzmann machine over variables y, z whose marginal distribution over z is the desired target:

$$E(y, z) = \frac{1}{2} y'y - y'Wz - \eta' z \quad , \quad W = (A - \mathrm{diag}(\eta))^{\frac{1}{2}}$$

where y is a continuous (Gaussian) random vector and η is chosen to make $(A - \mathrm{diag}\,\eta)$ positive definite. In this system, given y all variables z_i can be sampled in parallel, and given z all variables y_i can be sampled in parallel. For consistency with the terminology of LDA (Section 2.2), we refer to the augmented system over y, z as the "complete" model, and the original model over only z as a "collapsed" model.

Deviating from the original, collapsed system may have unintended consequences. There is some additional overhead in computing W; when the system is large or when A is changing (for example during learning) this can actually dominate the total execution time of the sampler. Each iteration also requires twice as many samples, and may be slightly more expensive to sample than uniform random variables. Finally, the augmented system may mix at a different rate than the original system. This effect was also noted by Martens and Sutskever (2010), who showed that the mixing rate depends on the value of η.

2.2 Latent Dirichlet Allocation (LDA)

LDA (Blei et al., 2003) is a generative probabilistic model for representing collections of discrete-valued data; while developed originally to model corpora of text documents, it has since been applied to a broad range of other problems. A graphical model representing LDA is shown in Fig.1a.

The observed data consist of D documents, where document d is of length N_d and consists of a series of tokens x_{di} which are words chosen from a vocabulary of size W. (We refer to the x_{di}, the i^{th} word appearing in document d as tokens to distingish them from the values they take on, which are words in the vocabulary.) The LDA model describes these data as arising from a generative process in which there are T possible topics, each described by a relatively sparse multinomial distribution ϕ_t over the W words; these distributions are not observed, but are given a Dirichlet prior with

parameter β.[1] Each document d is associated with a hidden multinomial distribution over topics, θ_d, with Dirichlet prior α. Then, for word $i \in \{1 \dots N_d\}$, a topic z_{di} is chosen according to θ_d, and a word x_{di} is chosen according to $\phi_{z_{di}}$.

The goal of inference is to compute the posterior distribution of topics in each document, θ_d, and words in each topic, ϕ_t, given the observed document corpus. There are several Gibbs sampling possibilities. Given samples of the $\{\theta_d\}$ and $\{\phi_t\}$, the topic assignments $\{z_{di}\}$ are all independent, and vice versa, providing a trivially parallel Gibbs sampler. In many ways, this is similar to the parallel RBM approach. Unfortunately, this Gibbs sampler mixes very slowly. In contrast, a collapsed Gibbs sampler integrates out the θ_d and ϕ_t, and samples only the topic assignments z_{di}. Define the sufficient statistics $N_{dwt} = |\{i : x_{di} = w, z_{di} = t\}|$; sampling requires the document-topic counts $N_{dt} = \sum_w N_{dwt}$, word-topic counts $N_{wt} = \sum_d N_{dwt}$ and topic counts $N_t = \sum_w N_{wt}$. Additionally we use $\neg di$ to indicate that token i in document d is excluded (subtracted) from the counts. The collapsed Gibbs sampler simply proceeds through each token, sampling its topic assignment z_{di} given all the other assignments

$$p(z_{di} = t|\mathbf{z}^{\neg di}, \mathbf{x}, \alpha, \beta) = \frac{1}{Z} \frac{a_{dt} b_{wt}}{c_t} \tag{2}$$

where

$$a_{dt} = N_{dt}^{\neg di} + \alpha \quad b_{wt} = N_{wt}^{\neg di} + \beta \quad c_t = N_t^{\neg di} + W\beta,$$

$w = x_{di}$, and Z is a constant that serves to normalize the distribution.

This algorithm is linear in the size of the corpus, and is not easily made parallel since the distribution of each z_{di} depends on every other assignment. However, for a large corpus, we may hope that the sums on which the distribution depends are relatively stable, and allow samples to be taken using "old" values of other variables. Approximate distributed LDA, or adLDA (Newman et al., 2007), uses this idea to parallelize LDA; it splits the collection of documents into a number of blocks, then resamples each block in parallel using the previous iteration's values for non-local blocks, and exchanges the results before continuing to the next iteration. This idea appears to work well in practice and has been extended to hierarchical models and asynchronous updates (Asuncion et al., 2009), and is particularly useful in applying LDA to massive data sets which do not fit in a single computer's memory (Wang et al., 2009).

[1] Although many applications of LDA learn asymmetric Dirichlet parameters (Wallach et al., 2009; Minka, 2003), we will use a symmetric prior with scalar parameter β for simplicity; all results are easily generalized to the more general case.

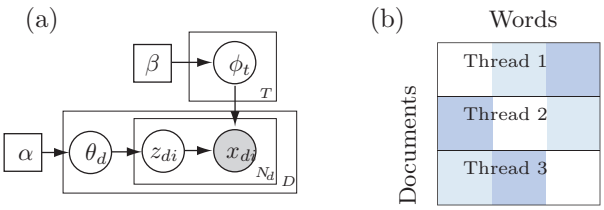


Figure 1: (a) Graphical model for LDA. (b) adLDA operates by allocating B blocks of documents across multiple compute elements (here, threads); "blocked" adLDA-B also partitions across the vocabulary (colors) and samples only same-colored blocks in parallel before resynchronizing.

The adLDA algorithm has also been applied in shared memory systems, including desktops (Newman et al., 2007) and graphics processing units (Yan et al., 2009). In shared memory systems, more rapid synchronization is possible between processes and several improvements are possible. One significant improvement is to subdivide the data more finely, blocking both groups of documents and words in the vocabulary, and coordinate the parallel samplers such that no two samplers are working on the same document or the same word. This organization enables the values of a_{dt} and b_{wt} to reside in shared memory without risk of access conflicts, reducing the amount of memory required and increasing the accuracy of the sampler (Yan et al., 2009; Ihler and Newman, 2009).

3 Exact Parallel Gibbs Sampling

In this section we describe our technique for drawing samples in (near) parallel. First, we describe the general form of the bounds we require, then discuss how to organize the operations across threads to easily compute the required quantities.

3.1 Look-Ahead Sampling

The difficulty with parallelizing the operations of collapsed Gibbs sampling in, for example, the LDA model is that every variable z_{di} depends at least slighlty on every other variable. However, the observation underlying adLDA and its variants is that although all variables are dependent, many of the dependencies are very weak. The adLDA algorithm exploits this fact to create an approximate sampler, while in the sequel we explore using it to create a "nearly parallel" sampler.

Consider a system in which two threads are attempting to sample a single variable each. One thread (or variable) can be considered to preceed the other, and can simply sample. The issue is that we would like the second thread to not wait until the first is finished,

but attempt to draw its sample early, before the value produced by the first thread is known. The picture as seen by a single thread is shown in Figure 2 – we would like to sample a variable, say z_6, given neighboring variable's values, only some of which are known. (We mark unknown values by "?".)

We can construct such a *look-ahead* sampler by finding lower bounds on the probability of each possible outcome for z_6 given any configuration δ of the unknown values. In other words, we find values $\hat{p}_t$ such that

$$\hat{p}_t \leq \min_{\delta_1,\delta_2,\ldots} \Pr[z_6 = t \,|\, z_1 = 2, z_2 = 1, z_3 = \delta_1, z_4 = \delta_2, \ldots]$$

Given such a set of lower bounds, we can sample the value of z_6 without knowing δ with some probability, $\sum_t \hat{p}_t$. With probability $1 - \sum_t \hat{p}_t$, our sampler fails to complete its action, and waits until the values of δ_1, δ_2 are known (i.e., they have been sampled by another thread). It is then easy to compute the remainder,

$$r_t = \Pr[z_6 = t \,|\, z_1 = 2, z_2 = 1, z_3 = \delta_1, z_4 = \delta_2, \ldots] - \hat{p}_t$$

and complete the sample.[2]

Our look-ahead sampler is closely related to techniques for drawing exact samples from Markov chain Monte Carlo simulations, in which similar "missing" values are included explicitly and used to check whether the Markov chain has become independent of its initial conditions. In application to LDA, discussed further in Section 4.2, it also has similarities to bound-based approaches such as shortcut sampling (Porteous et al., 2008) and sparse LDA (Yao et al., 2009).

3.2 Data Structures

A natural question is then how to coordinate among multiple threads, each trying to access and change the variables on which the others depend? In particular, we need access to the implicit order in which the samples are scheduled to take place: when a thread computes $\hat{p}$, it needs to know which variables z_i have been scheduled for update earlier to bound their influence and potentially wait for them to be updated. The thread must also maintain access to the old values of variables scheduled for later updates, despite the fact that they may be updated by another thread before our sample has been finished.

We resolve these difficulties with a shared two-row queue structure, with one row ("`old`") indicating variables scheduled for resampling and their values, and the other row ("`new`") indicating the new values which have been drawn for them. The ordering of the queue represents the sequential ordering of variables which the parallel sampler will preserve. An example queue structure is shown in Figure 3.

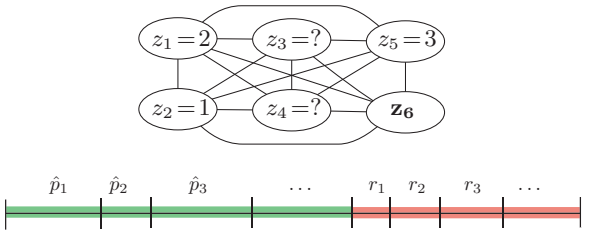


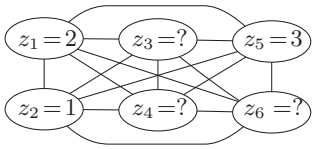
Figure 2: Look-ahead sampling of z_6. Although the values of z_3, z_4 are unknown ("?"), we can lower bound the probabilities of each outcome for z_6 as $\hat{p}_1 \ldots$ and draw a sample. With some probability (shown in green) this sampling succeeds; otherwise we wait until observing the values of z_3, z_4 to compute $r_1 \ldots$ (red), at which point we can determine the new value of z_6.

Each thread maintains a local copy of whatever sufficient statistics are required for sampling, and uses the queue to update and bound these statistics in order. Threads each use two local pointers to manage the sequence of variable updates. Local pointer `current` indicates the variable currently being sampled by this thread, while pointer `back` indicates those variables which have already been sampled before our attempt begins. Although it may be the case that many of the variables between `back` and `current` have been sampled already, it is difficult to maintain an instantaneous snapshot of drawn versus missing values; therefore, we treat *all* values between `back` and `current` as missing.

To begin sampling a block, our thread locks the queue and allocates a new set of K variables to sample using a shared `head` pointer, filling in the old values and "?" for new values. Then, we simply advance `current`, removing each value in `old` from our sufficient statistics and keeping track of bound information. We also advance `back`, adding each value in `new` to our sufficient statistics and removing its bound influence, until we reach the first "?". We attempt to sample each of our variables; if any fail, we wait until the entire region between our `back` and `current` pointers has been filled, at which point we can calculate the remainder probabilities r_i and finish our sample. Between drawing samples, we can advance `back` if desired. The size K is a parameter of the algorithm; we shall see in experiments how changing K can affect performance.

Our sampler is predicated on the idea that many of the variables we encounter in the queue have little or no influence on our outcome probabilities, so that the

[2]We consider only discrete-valued probability distributions, in which these bounds are relatively easy to obtain; whether look-ahead sampling can be easily applied to continuous-valued variables is an open question for further research.



Variables	z_1	z_2	z_3	z_4	z_5	z_6	$\cdots$
`old` z	1	3	2	1	1	2	$\cdots$
`new` z	2	1	?	?	3	?	$\cdots$
			thread 2		*thread 1*		

Figure 3: Queue used for coordination among sampling threads. The first row indicates the variable referred to for that column, the second its value prior to this sampling pass, and the third its newly updated value if drawn, or "?" if not. Each thread allocates a small block (here, 2 variables) to be sampled before re-acquiring a lock on the queue to allocate the next block. Each thread also maintains two local pointers: `current`, indicating which sample it is currently drawing, and `back`, indicating through which sample it has incorporated all new values.

probability of a successful sample, $\sum \hat{p}_t$, will be close to 1. Our data structure can be used regardless of the independence structure of the model; in the case of independent or weakly dependent samples it can reach near linear speed-up, while if the variables currently being sampled are tightly coupled it will slow down but maintain a correct sampling distribution.

4 Developing Bounds

The main component of our algorithm is to define what statistics each thread will need to maintain locally, and how bounds $\hat{p}_t$ on each probability can be computed easily. In this section, we describe the creation of such bounds for both Boltzmann machines and LDA.

4.1 Bounds for Boltzmann Machines

The conditional distribution required for Gibbs sampling in the Boltzmann model is given by

$$p(z_i = 1|z_{\neg i}) = \sigma(a_{ii} + \sum_{j \neq i} 2\, a_{ij} z_j).$$

We develop a simple upper and lower bound on the sum in terms of the pointers `back` and `current`.

We can split the sum over j into three components: $j <$`back`, $j \geq$`current`, and $j \in$ [`back`,`current`-1] $= \mathcal{J}$. The first of these involves only values in `new` which are not equal to "?", while the second involves only values in `old`; for each of these we compute directly:

$$h^{\texttt{new}} = \sum_{j<\texttt{back}} 2\, a_{ij} z_j^{\texttt{new}} \qquad h^{\texttt{old}} = a_{ii} + \sum_{j>\texttt{current}} 2\, a_{ij} z_j^{\texttt{old}}$$

The last term requires `new` values which may not yet be available, and we create upper and lower bounds:

$$h^+ = \sum_{j \in \mathcal{J}} \max(2\, a_{ij}, 0) \qquad h^- = \sum_{j \in \mathcal{J}} \min(2\, a_{ij}, 0)$$

We then have

$$\hat{p}_0 = \sigma(-(h^{\texttt{old}} + h^{\texttt{new}} + h^+)) \quad \leq p(z_i = 0|z_{\neg i})$$
$$\hat{p}_1 = \sigma(h^{\texttt{old}} + h^{\texttt{new}} + h^-) \quad \leq p(z_i = 1|z_{\neg i})$$

4.2 Bounds for LDA

We next show how to construct similar bounds that lead to an efficient parallel sampler for LDA. As in adLDA-B, we break the set of documents and words (vocabulary) into smaller sub-blocks, assigned to different threads. Each thread samples the topic assignments within a single sub-block, ensuring that any given region of the document-topic (a_{dt}) and word-topic (b_{wt}) count matrices are accessed by only one thread and can reside in shared memory. Thus, from Eq. 2 the only sufficient statistic that must be maintained locally is the overall topic-count vector c_t. Within this vector, each variable in the model has equal "influence", so the identity of the variables will not be required when updating.

Now consider constructing a bound on the probability given a collection of unknown assignments. Define the variables $\delta = \{\delta_1, \delta_2, ...\delta_T\}$, and $\Delta = \sum_t \delta_t$ to be the number of variables currently marked as missing which will eventually be assigned to each of the T topics. When advancing `current`, we subtract each old topic assignment from our local copy of c_t, and eventually will re-add the entries of δ. The true probability of sampling a topic t if the entries of δ were known is

$$p_t(\delta) = \frac{q_t(\delta)}{Z(\delta)} = \frac{1}{Z(\delta)} \frac{a_{dt} b_{wt}}{c_t + \delta_t} \tag{3}$$

with $q_t(\delta) = \frac{a_{dt} b_{wt}}{c_t + \delta_t}$ and the normalizing constant $Z(\delta) = \sum_t q_t(\delta)$. Unfortunately, we cannot know the assignments δ_t, but it is easy for us to compute the total *number* of variables which are treated as missing, $\Delta = \sum_t \delta_t$, since this is simply the distance between pointers `back` and `current`.

We derive a lower bound $\hat{p}$ of the true probability mass (Eq.3) which depends only on Δ. It is easy to verify that at any point in our algorithm,

$$\hat{q}_t = \frac{a_{dt} b_{wt}}{c_t + \Delta} \leq q_t(\delta) \qquad Z^+ = \sum_t \frac{a_{dt} b_{wt}}{c_t} \geq Z(\delta)$$

ensuring that we can define a valid lower bound on p_t

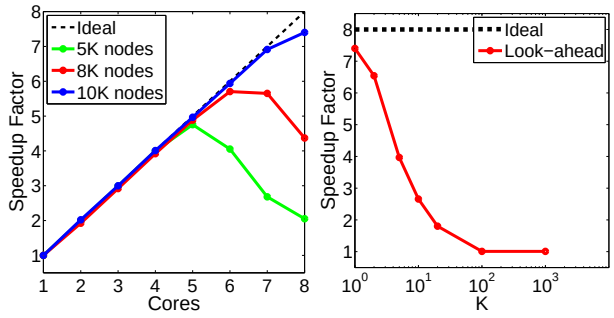


Figure 4: Speedup analysis for Boltzmann machines. (a) Speedup factor for several model sizes N; as N increases our sampler becomes more efficient. (b) Speedup as a function of K, the number of variables per lock; the decline suggests that delays are due to the conflict rate, rather than the lock time.

as

$$\hat{p}_t = \frac{\hat{q}_t}{Z^+} \qquad \leq \qquad \min_{\delta \,:\, \sum_t \delta_t = \Delta} p_t(\delta) \qquad (4)$$

As discussed previously, our sampler will succeed with probability $\sum \hat{p}_t$, and fail (forcing the thread to wait) with probability $\sum r_t$. We refer to the latter quantity as the *conflict rate*, denoted R_c. In our experiments, the conflict rate is often very small (between 4×10^{-5} and 2×10^{-2}). Thus, more than 98% of the time our thread does not need to wait on the previous threads' unavailable samples, so that the samples are taken very nearly in parallel without compromising the accuracy of their distribution.

5 Experiments

We assessed our look-ahead sampler experimentally on both Boltzmann machines and LDA. All experiments were performed on an 8-core Intel Xeon 2.33GHz machine with 16GB memory, and used wall clock time to assess speed-up performance.

5.1 Boltzmann machines

We first compare our sampler on Boltzmann machines using synthetically generated binary Markov random fields. We constructed random, 50% dense graphs of size $N = 5000$, 8000, and 10000 whose parameters are i.i.d. Gaussian samples, $\theta_{ij} \sim N(0, .1)$. We then compared the performance of our look-ahead sampler against the parallel RBM sampler of Martens and Sutskever (2010). For the RBM-based sampler, to ensure $A - \eta$ is positive definite we set η as the minimum of zero or 1.01 times the minimum eigenvalue of A.

Figure 4 shows the performance of the look-ahead sampler as a function of model size N, and as a function of the block size K for $N = 10^5$. As can be seen, speed-up is approximately linear with the number of cores up to a point, after which coordination issues among the processors (due to locking or an inability to sample without access to other processors' results) begins to dominate. Performance decreases with K, suggesting that it is the conflict rate (failure to draw a sample using the bounds) that is responsible.

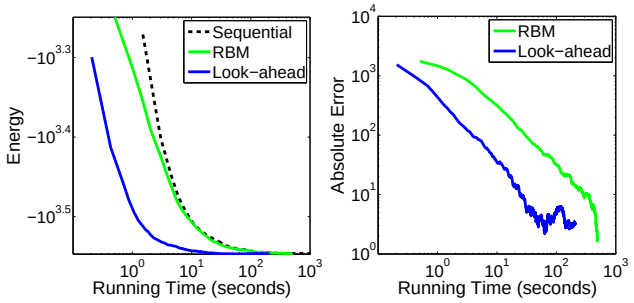


Figure 5: Convergence over time for Boltzmann machines. (a) Estimated energy vs. running time; (b) absolute error vs. running time. On 8 cores, our look-ahead sampler is faster than either the sequential or parallel RBM-based samplers.

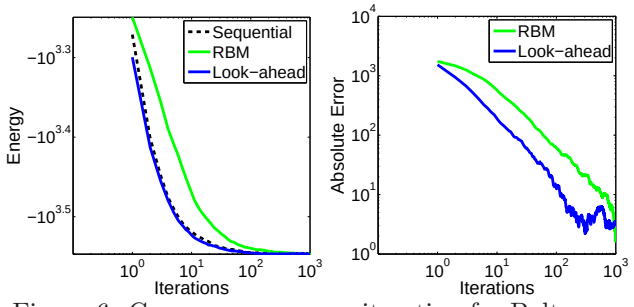


Figure 6: Convergence versus iteration for Boltzmann machines. (a) Estimated energy; (b) absolute error vs. iterations. Our look-ahead sampler converges at the same rate as the sequential sampler, while the RBM-based method is slightly slower.

Figure 5 illustrates the convergence of the look-ahead sampler compared to the sequential and the RBM-based parallel samplers as a function of time. To gauge convergence, we monitor the sample-based estimate of the expected energy of the system, $\mathbb{E}_{p(z)}[-z'Az]$, a weighted average of second-order statistics. We ran a sequential sampler to 10,000 iterations to obtain an estimate of the true value, and show both the convergence toward this value (Figure 5a) and the error (difference in estimated energies) as a function of time. In our implementation, the RBM approach with 8 cores was approximately the same speed as the sequential sampler, while the look-ahead sampler was about 7.5× faster. This appears to be due to a combination of the slower sampling and slower mixing. For comparison, Figure 6 shows the same energy estimates for each of

the three algorithms, but plotted against the number of iterations; here we can see that both the original and parallel look-ahead collapsed samplers have identical behavior, while the RBM converges slightly slower due to its different mixing rate. In fairness of course, as the number of cores increase the look-ahead sampler will reach some maximum speed, while the RBM-based sampler may continue to improve, making it possibly better suited to GPUs and other massively parallel platforms.

5.2 Latent Dirichlet allocation

In the following experiments for LDA, we used three data sets from the UCI machine learning repository (Asuncion and Newman, 2007), to which LDA is commonly applied. The relative sizes of the data sets are listed in Table 1. In all cases, we fixed the hyperparameters to $\alpha = .1$, $\beta = .01$.

Parameter Choices. A critical component of the algorithm in practice is the balance between conflicts caused by being unable to sample a variable and the time spent interacting with the shared queue head pointer, to allocate space for subsequent variables. This balance is controlled in part by the parameter K, the size of each allocated region of the queue. It also depends on, for example, the time required to perform a single sample and the relative influence of a small number of missing values, and thus depends on parameters of the model and data such as the number of topics T and data set size, as well as the number of threads competing for the shared resources. In this section, we experimentally analyze the behavior of our algorithm under various conditions.

We first show the overall speed-up of the algorithm for various settings of K and T, and additionally the total number of blocks B into which the data are initially partitioned (see Figure 1). These results are shown in Figure 7 for the New York Times data set. There is a wide range of values of K in which the algorithm performs well; too small K means insufficient work per lock operation, while too large K means a high probability of a conflict and high wait ratio. The algorithm also performs better for sufficiently large numbers of topics, since this too affects the relative time spent on computation between lock operations. We found the number of blocks B also has relatively little effect over a broad range of values, except at the extrema.

Speedup for different data sizes. In order to illustrate our multicore LDA is able to achieve good speedup, we test it on three different data sets with parameters set to $T = 500$, $B = 100$ and $K = 10$ for the NYTimes and $K = 3$ for ENRON and NIPS data sets. Fig. 8a shows the comparison of speedup for all three data sets as we vary the number of cores used, P, from 1 to 8 in an 8-node shared memory machine. Smaller data sets such as NIPS do not increase beyond a certain point, when a sufficient number of sampling conflicts are occurring to offset the advantages of more cores. As the size of the data set grows, the coupling between assignments becomes weaker and the algorithm can take more advantage of the available parallelism.

Dataset	NYTimes	ENRON	NIPS
N	100M	6.4M	1.9M
W	102K	28K	12K
D	300K	40K	1.5K

Table 1: Data sets used in the experiments and their relative sizes, from (Asuncion and Newman, 2007).

Speedup for different models. We also compare the speedup of our multicore LDA and two different implementations of adLDA, adLDA (Newman et al., 2007) and adLDA-B (Ihler and Newman, 2009), on the same multicore machine. We use the NYTimes data set and the same parameter values as before. All three algorithms are nearly identical in speed-up, with the original adLDA slightly slower than adLDA-B due to its lack of shared resources for the word-topic matrix. Our multicore LDA is only slightly slower than either adLDA, caused by the overhead of frequent communication and lock exchange. However, unlike either version of adLDA, our algorithm makes no approximations to the sampling distributions and thus adapts automatically to data sets or parameter values in which the sampling steps must be more tightly coupled.

Perplexity Comparisons. Finally, we evaluate the performance of LDA, adLDA, and our multicore LDA using perplexity, evaluated on the Enron dataset. We held out 10% of documents as test data and used the remaining 90% for training. For each model, we run 100 Markov chains starting from different random seeds, and plot error bars of perplexity for each model (indicating ± 2 standard deviations) across runs. Parameters are set for all these models to $T = 500, B = 100$ and $K = 10$.

Figure 9a shows the perplexity from iterations 100 to 200 (after most burn-in has occurred). The resulting perplexities are comparable, and in fact the original adLDA has slightly lower average perplexity, which is consistent with (Newman et al., 2007, 2009). Our algorithm is closest in mean to sequential LDA, to which it should be equivalent and blocked adLDA (adLDA-B) is also very close, well within two standard deviations.

To assess the significance of this difference in accuracy, we perform a Kolmogorov-Smirnov (KS) two-sample

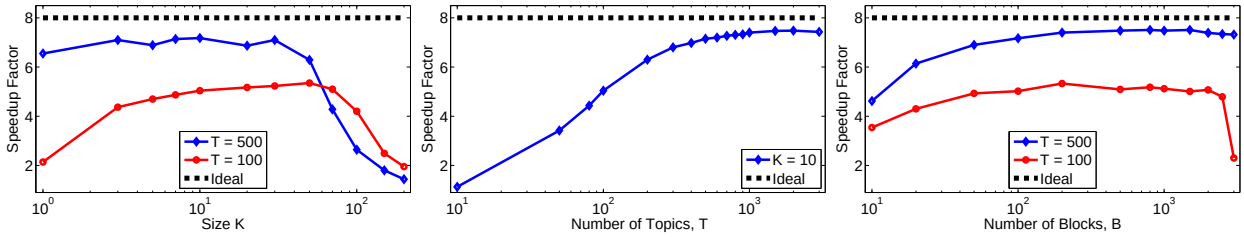


Figure 7: Speed-up of multicore LDA on the NYTimes dataset with (a) varying K and $T = 500$, 100; (b) varying topics T and $K = 10$; (c) number of block partitions B, with $T = 500$, 100.

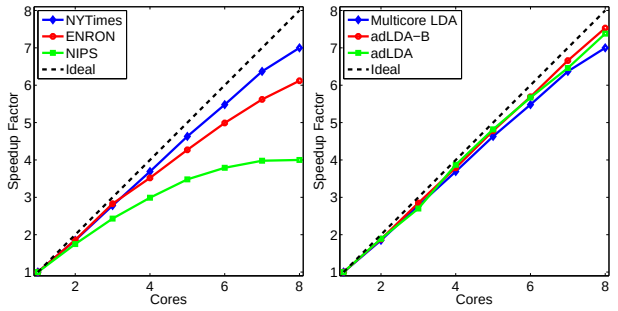


Figure 8: Speed-up analysis. (a) Speedup for three datasets; (b) comparing speed-up for multicore vs. adLDA methods.

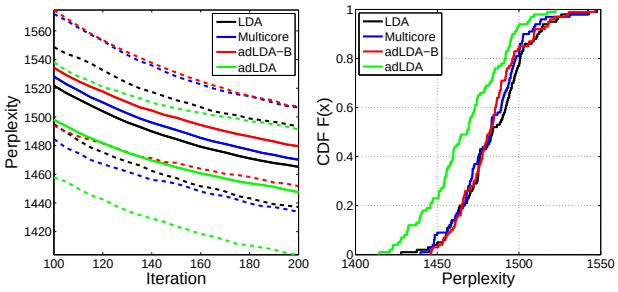


Figure 9: Perplexity comparisions: (a) perplexity convergence for different models; (b) perplexity CDFs of 4 models, used for a KS test of equal distributions.

test between the 100 perplexities found by each initialization of sequential LDA, against those for each of adLDA, adLDA-B, and our multicore sampler. Figure 9b shows the cumulative distribution functions of perplexity for each set of 100 samples, which form the KS test statistic. Both multicore and adLDA-B are not statistically significantly different, with p-values of 0.794 and 0.140, respectively; adLDA is significantly different, with p-value 10^{-5}. It is difficult to assess the differences in stationary distributions for sampling methods, but this provides empirical evidence that our sampler is correct, and is consistent with adLDA-B providing a more accurate approximation.

6 Discussion and Conclusion

Previous work on parallelizing Gibbs sampling has focused on using extended variable sets that decouple (Martens and Sutskever, 2010), or making approximations to the true transition probabilities (Newman et al., 2007). The former may introduce changes in the mixing rate, while the latter introduces additional unknown approximations.

This work asks whether it is possible to take advantage of shared-memory parallel architectures in Gibbs sampling without compromising the theoretical properties of the method. To this end, we developed a look-ahead sampler which is capable of drawing a sample in parallel with some probability, and defaulting back to sequential sampling in the case of failure. We showed how operations can be organized among threads to ensure that each thread is able to keep track of the values required for its own probability distributions, and compute the bounds required by the sampler. We showed the form of these operations and bounds on two models studied for parallel sampling: Boltzmann machines and LDA. In both cases, we found that our algorithm provides competitive speed on a multicore machine, without fundamentally changing the sampling process.

The advantage of our method is primarily theoretical, in that it preserves all the properties of the original Gibbs sampler at the potential risk of not gaining full advantage of the parallel architecture. As multicore architectures become increasingly common, it is important that we understand the variety of trade-offs at our disposal. Open questions include how these techniques can fit into alternate computational models and hardware, such as GPUs and data storage hierarchies. We also plan to explore extending the approach to non-parametric Bayesian models (Asuncion et al., 2009) by developing bounds that can be used in these cases.

Acknowledgements

The authors thank David Newman and Ian Porteous for many useful discussions.

References

A. Asuncion and D. Newman. *UCI machine learning repository*, 2007. `http://archive.ics.uci.edu/ml/`.

A. Asuncion, P. Smyth, and M. Welling. Asynchronous distributed learning of topic models. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems 21*, pages 81–88. 2009.

D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet allocation. *Journal of Machine Learning Research*, 3: 993–1022, 2003.

C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *NIPS 19*, pages 281–288, 2006.

J. Gonzalez, Y. Low, and C. Guestrin. Residual splash for optimally parallelizing belief propagation. In *In Artificial Intelligence and Statistics (AISTATS)*, 2009a.

J. Gonzalez, Y. Low, C. Guestrin, and D. O'Hallaron. Distributed parallel inference on large factor graphs. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Montreal, Canada, July 2009b.

A. Ihler and D. Newman. Bounding sample errors in approximate distributed latent dirichlet allocation. In *Large Scale Machine Learning Workshop, NIPS,UCI ICS Technical Report 09-06*, 2009.

J. Martens and I. Sutskever. Parallelizable sampling of markov random fields. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, 2010*, pages 517–524, 2010.

T. Minka. Estimating a Dirichlet distribution, 2003. URL `http://research.microsoft.com/en-us/um/people/minka/papers/dirichlet/`.

D. Newman, A. Asuncion, P. Smyth, and M. Welling. Distributed inference for latent dirichlet allocation. In *NIPS 20*, 2007.

D. Newman, A. Asuncion, P. Smyth, and M. Welling. Distributed algorithm for topic models. *Journal of Machine Learning Research*, pages 1801–1828, 2009.

I. Porteous, D. Newman, A. Ihler, A. Asuncion, P. Smyth, and M. Welling. Fast collapsed Gibbs sampling for latent Dirichlet allocation. In *ACM SIGKDD*, pages 569–577, 2008.

R. Ren and G. Orkoulas. Parallel Markov chain Monte Carlo simulations. *Journal of Chemical Physics*, 126: 211102–211102–4, 2007.

H. Wallach, D. Mimno, and A. McCallum. Rethinking lda: Why priors matter. In *NIPS 22*, 2009.

Y. Wang, H. Bai, M. Stanton, W.-Y. Chen, and E. Y. Chang. PLDA: Parallel latent Dirichlet allocation for large-scale applications. In *Int'l Conf. Algorithmic Aspects in Inform. and Management (AAIM)*, June 2009.

M. Whiley and S. Wilson. Parallel algorithms for markov chain monte carlo methods in latent spatial gaussian models. *Statistics and Computing*, pages 171–179, 2004.

F. Yan, N. Xu, and Y. Qi. Parallel inference for latent Dirichlet allocation on graphics processing units. In To appear*, Advances in Neural Information Processing Systems*, Dec. 2009.

L. Yao, D. Mimno, and A. McCallum. Efficient methods for topic model inference on streaming document collections. In *ACM SIGKDD*, pages 937–946. ACM, 2009.