Fast Parallel SAME Gibbs Sampling on General Discrete Bayesian Networks and Factor Graphs

by Haoyu Chen

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor John F. Canny
Research Advisor

(Date)

* * * * * * *

Professor Peter L. Bartlett
Second Reader

(Date)

Abstract

A fundamental task in machine learning and related fields is to perform inference on probabilistic graphical models. Since exact inference takes exponential time in general, a variety of approximate methods are used. Gibbs sampling is one of the most accurate approaches and provides unbiased samples from the posterior but it has historically been too expensive for large models. In this paper, we present an optimized, parallel Gibbs sampler augmented with state replication (SAME or State Augmented Marginal Estimation) to decrease convergence time. We find that SAME can improve the quality of parameter estimates while accelerating convergence. Experiments on both synthetic and real data show that our Gibbs sampler is substantially faster than the state of the art sampler, JAGS, without sacrificing accuracy. Our ultimate objective is to introduce the Gibbs sampler to researchers in many fields to expand their range of feasible inference problems.

1 Introduction

In many machine learning applications, the user has a distribution $P(X \mid Z, \Theta)$ where X is observed data, Z is hidden (latent) state, and Θ represents the model parameters. The goal is generally to find an optimal Θ with respect to X, while marginalizing out Z. To represent these problems, it is common to use graphical models. In this thesis, our focus is on discrete-state graphs, for both the directed and undirected cases. For Bayesian Networks, each vertex i represents one random variable X_i, and node transitions are characterized by a set of conditional probability tables (CPTs). Each CPT represents a local probability distribution $\Pr(X_i \mid X_{\pi_i})$ where X_i is the random variable represented by vertex i, π_i is the set of parent vertices of vertex i and X_{π_i} represents the random variables of parent vertices. We denote the full set of CPTs as Θ for a Bayesian Network. For Factor Graphs, there are two types of vertices: variable vertices, which can be either evidence variables and query variables, and factor vertices, which define the relationships between variables. For variable vertex i, the corresponding random variable is X_i, and for factor vertex i, the corresponding factor table is f_i. We use F_i to represent the set of variable vertices which have an edge to factor vertex f_i. The joint distribution represented by the graph is the product of each factor (i.e. $\Pr(X_1, ..., X_n) = \prod_i f_i(X_{F_i})$). Each factor is represented as a high dimensional discrete table which we call factor table (FT). We denote the full set of FTs as Θ for a Factor Graph.

In general, the network state is partially observed $\mathcal{D} = \{\xi_1, \ldots, \xi_m\}$, where ξ_i is an n-dimensional vector with assignments to the n variables of the graph, or "N/A" to indicate missing data. We assume that the structure of the probabilistic network — its vertices and edges — is known in advance, but not the set of observations. This form allows us to represent latent and well as observed states, but also observations on an arbitrary (sample-dependent) subset of those states.

Well-known strategies for parameter estimation with partially observed data include Expectation-Maximization [5] and variations of gradient ascent [15]. Parameter estimation using these methods marginalizes over missing states, which typically dominates the runtime. Using Monte-Carlo methods for marginalization leads to the MC-EM methods [17].

For parameter estimation with a Gibbs sampler, we sample both latent states and parameters from $P(Z, \Theta \mid \mathcal{D})$. For the case of discrete node states, the Z follow multinomial distributions while the prior distribution of Θ is Dirichlet. Gibbs sampling is widely used in machine learning but has seen limited use on large datasets. It is typically orders of magnitude slower than special-purpose, approximate methods; for Latent Dirichlet Allocation (LDA), these methods include Variational Bayes [2] and Walker's alias method [13]. In a recent result, [19] showed that by combining the State Augmented Monte Carlo (SAME) technique [6] with Gibbs sampling, one can match the speed of approximate methods while obtaining higher quality estimates. But that paper was restricted to LDA and similar models. In this thesis, we build upon that result by presenting a SAME Gibbs sampler for general discrete probabilistic graphical models. The contributions of this thesis are:

- A more general Gibbs sampler using SAME sampling with improved convergence and better-quality MAP or ML parameter estimates over standard samplers.
- We parallelize the sampler which provides acceleration on both CPU and GPU hardware.
- The sampler maintains only model-related state and processes data out-of-memory so it can scale to very large datasets (either from disk or network storage).

We benchmark our sampler versus a state of the art Gibbs sampler, JAGS [14], and find that our Gibbs sampler is at least an order of magnitude faster. The throughput and scalability of the sampler (nodes processed per second) is competitive with special purpose methods while maintaining the accuracy of Gibbs sampling.

2 Related Work

The problem of Bayesian inference for graphical models is well-studied ([11] and [16] cover recent approximate and MCMC methods). Gibbs sampling has proved to be one of the most important practical techniques for large models; collapsed samplers are widely used for LDA and related models [8]. In addition, parallelism has been improved using color groups [7], and approximate, uncoordinated parallelism has been shown to give good results in practice [10]. Gibbs sampling has also been applied to large-scale databases [18].

Recently, Graphics Processing Units (GPUs) have been valuable for deep learning, and are used in most toolkits including Theano [1], CAFFE [9] and Torch [4]. We continue this trend by using GPU acceleration for our sampler.

The result most directly related to the current work, as briefly mentioned in Section 1, is one that shows how the addition of SAME to a GPU-accelerated Gibbs sampler can be very fast for Latent Dirichlet Allocation and the Chinese Restaurant Process [19]. In that paper, they explored the application of SAME to graphical model inference on modern hardware, and showed that combining SAME with factored sample representation (or approximation) gives throughput competitive with the fastest symbolic methods, but with potentially better quality. We extend that result by implementing a general-purpose Gibbs sampler that can be applied to arbitrary discrete graphical models.

3 Fast Parallel SAME Gibbs Sampling

In this section, we introduce the basic strategy we utilize to speed up Gibbs sampling.

3.1 Graph Coloring

We apply graph coloring to the moralized graph of the network so that the vertices in the same color group are conditionally independent given the vertices in the other color groups. Assume there is a $k-$coloring of moralized graph, and denote V_c as the set of variables assigned to color c, where $1 \leq c \leq k$. Our algorithm samples sequentially from V_1 to V_k. Within each color group, we sample all the vertices in parallel. Finding the optimal coloring of a general graph is NP-complete, so we use efficient heuristics [12] for balanced graph coloring, which work well in practice.

3.2 Efficient Parallel Inference

In order to do the sampling efficiently, we formulate the inference problem into matrix operations.

For Bayesian Networks, we denote u the vertex whose value is being sampled, X_u the random variable represented by u, V_c the color group which u belongs to, V the whole vertex set, and g_u the log probability of X_u conditionally on the value of other vertices in the graph i.e. $X_{V\backslash u}$. Then we can write the g_u as the linear combination of the components in the CPT:

$$\begin{aligned} g_u(X_u) &= \log P(X_u|X_{V\backslash u}) = \log P(X_u|X_{V\backslash V_c}) \\ &= \log P(X_u, X_{V\backslash V_c}) + C_1 \\ &= \log \big(P(X_u|\pi_u) \prod_{v\in\{V\backslash V_c\}} P(X_v|\pi_v)\big) + C_1 \\ &= \log P(X_u|\pi_u) + \sum_{v\in\xi_u} \log P(X_v|\pi_v) + C_2 \end{aligned} \tag{1}$$

where π_v represents the parent vertices of v, ξ_u represents the child vertices of u, and C_1, C_2 are normalizing constants corresponding to the value of X_u and get cancelled out later.

With the help of Equation 1, we can write the log probability of one vertex v conditioned on all other vertices as the linear combination of the log probability of v conditioned on its parents and the log probability of the children of v conditioned on their parents.

For Factor Graphs, we can write the g_u as:

$$\begin{aligned} g_u &= \log P(X_u|X_{V\setminus u}) = \log \prod_i f_i(X_{F_i}) + C_1 \\ &= \log \prod_{i\in\{i|u\in F_i\}} f_i(X_{F_i}) + C_2 \\ &= \sum_{i\in\{i|u\in F_i\}} \log f_i(X_u, X_{F_i\setminus u}) + C_2 \end{aligned} \tag{2}$$

where we are only concerned about the factors which contain u, i.e. the set $\{f_i|u \in F_i\}$.

Thus, for both Bayesian Networks and Factor Graphs, we can write the log probability of one vertex conditioned on other vertices as the linear combination of components in CPTs/FTs.

In our system, we store all the components of CPTs/FTs in a one dimensional array. We construct a mapping matrix **M** and an offset array v_{offset}. Given the index of vertices in a color group, we can get the log probability of their value with known other vertices values by matrix operations:

$$\mathbf{I}_g = \mathbf{XM} + v_{offset} \tag{3}$$

$$g_u = \sum \Theta[\mathbf{I}_g] \tag{4}$$

where **M** is the mapping matrix with given the vertices value **X**, v_{offset} is the offset array which shifts the index to be the beginning of each CPT/FT, $\mathbf{I}_g$ is the index array of the components which is related to g_u, and Θ is the CPTs/FTs array. We sum the components of CPTs/FTs which are looked up by $\mathbf{I}_g$ to get the value of g_u, which will be used to sample the unknown vertex u. After computing the g_u for all the possible value of X_u, we sample the value of X_u by multinomial distribution.

Figure 1 shows two simple examples. Figure 1a is a simple Bayesian network with three random variables and their CPTs. Figure 1b is a simple factor graph with three random variables and two factors. The value of CPTs/FTs are store in a one dimensional array (Figure 2). For Bayesian networks the corresponding look-up matrix **M** and offset array v_{offset} are:

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & |X_2| * |X_3| \\ 0 & 1 & |X_3| \\ 0 & 0 & 1 \end{bmatrix}, v_{offset} = [0, |X_1|, |X_1| * |X_2|]$$

where $|X_i|$ represents the cardinality of random variable X_i (i.e. the number of states it can take on). The indices array $\mathbf{I}_g$ is:

$$\mathbf{I}_g = [X_1, X_2, X_3]\begin{bmatrix}1 & 0 & |X_2| * |X_3| \\ 0 & 1 & |X_3| \\ 0 & 0 & 1\end{bmatrix} + [0, |X_1|, |X_1| * |X_2|]$$

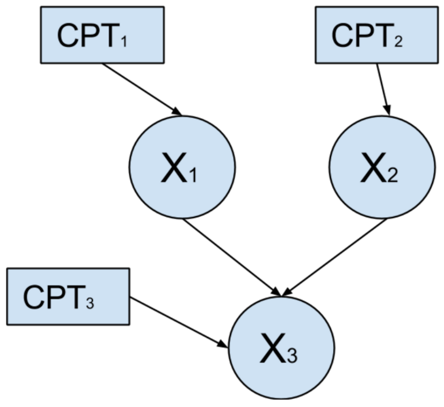
For factor graphs, the corresponding look-up matrix $\mathbf{M}$ and offset array v_{offset} are:

$$\mathbf{M} = \begin{bmatrix}|X_2| * |X_3| & 0 \\ |X_3| & 1 \\ 1 & 0\end{bmatrix}, v_{offset} = [0, |X_1| * |X_2| * |X_3|]$$

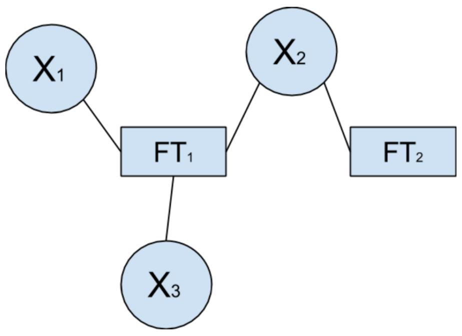
The indices array $\mathbf{I}_g$ is:

$$\mathbf{I}_g = [X_1, X_2, X_3]\begin{bmatrix}|X_2| * |X_3| & 0 \\ |X_3| & 1 \\ 1 & 0\end{bmatrix} + [0, |X_1| * |X_2| * |X_3|]$$

After we sample the unknown value vertices, the CPTs/FTs are sampled with the Dirichlet prior as one update iteration.



(a) A simple Bayesian network.



(b) A simple factor graph.

Figure 1: Two examples of graphical models along with their CPTs or FTs

3.3 SAME Sampling

SAME is a variant of MCMC where one artificially replicates latent states to create distributions that concentrate themselves on the global modes [6]. It is an efficient way of performing MAP estimation in high-dimensional spaces when needing to integrate out a large number

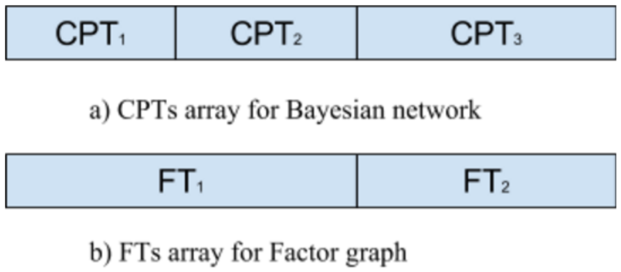

CPT₁
CPT₂
CPT₃
a) CPTs array for Bayesian network
FT₁
FT₂
b) FTs array for Factor graph

Figure 2: The CPTs/FTs array.

of variables. Given a distribution $P(X, Z \mid \Theta)$, to estimate the most likely Θ based on the data (X, Z) using SAME, one would define a new joint Q:

$$Q(X, \Theta, Z^{(1)}, \ldots, Z^{(m)}) = \prod_{j=1}^{m} P(X, \Theta, Z^{(j)}) \tag{5}$$

which models m copies of the distribution tied to the same set of parameters Θ, which in our case forms the set of Bayesian network CPTs. This new distribution Q is proportional to a *power* of the original distribution, so $Q(\Theta \mid X) \propto (P(\Theta \mid X))^m$. Thus, it has the same optima, including the global optimum, but its peaks are sharpened [6]. Note that as m increases, SAME approaches Expectation-Maximization [5] since the distribution would peak at the value corresponding to the maximum likelihood estimate.

We argue that SAME is beneficial for Gibbs sampling because it helps to reduce excess variance derived from discrete sample quantization. It is important, however, not to set the SAME replication factor m too high, which reduces sampling to EM and may lead to poor local optima. Instead, since the replication factor is formally equivalent to a temperature control, it can be used to anneal the sampler to a low-variance, high-quality parameter estimate.

4 Implementation of SAME Gibbs Sampling

Our Gibbs sampler is implemented as part of the open-source BIDMach library [3] for machine learning. Figure 3 shows a visualization of how it works on a large dataset. Our sampler expects a data matrix (typically sparse), with rows representing variables and columns representing cases. BIDMach divides data into same-sized *minibatches* and iterates through them to update parameters. If there are M minibatches in the entire dataset, we maintain a moving sum of the state counts for the M most-recently processed minibatches. Updating

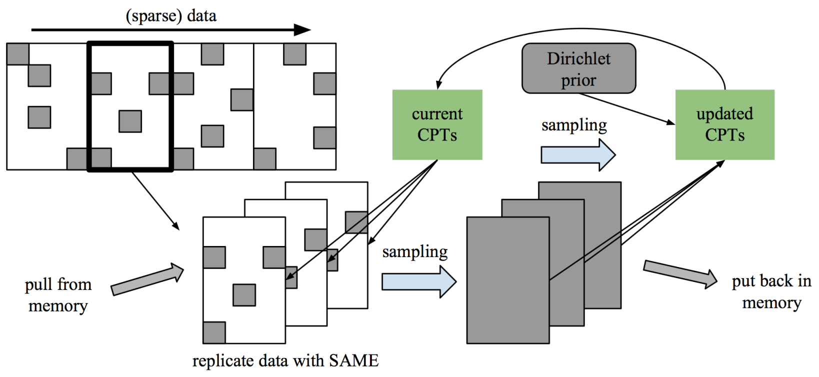


Figure 3: This visualizes our Gibbs sampler at work. The original data is split into four minibatches of equal sizes (for caching purposes), with each having some known data (shaded gray) and unknown data (white). For each minibatch, the sampler replicates the data three times ($m = 3$), samples the unknown values, and then uses those with the prior to update its estimate of the CPTs.

this average requires us to maintain the state counts for each minibatch from the last iteration. We subtract these from the total state counts before adding in the updated counts for this minibatch from the current iteration[1].

Our Gibbs sampler is augmented with SAME. Consequently, if m is the SAME parameter, for each minibatch our sampler forms m copies of the known data. Then, it performs Gibbs sampling to fill in the missing data in each copy of the minibatch using the current CPTs. These sampled results are combined with an adjustable Dirichlet prior and the current CPTs to form a set of discrete counts, which are the Dirichlet parameters to update (via sampling) the CPTs.

There are several optimizations used to improve performance. First, since storage allocation is very expensive on GPUs and their memory is limited (3-12 GB is typical), BIDMach uses a matrix caching strategy to reuse memory for matrices of the same dimensions. (This is why minibatches in BIDMach need to be the same size.) Second, graphical structure and SAME-replicated states for a particular color group are fused into large matrices to maximize parallelism and hide GPU-kernel overhead. The sampling process itself is done via a series of matrix multiplication operations as described in Section 3.2.

As mentioned in Section 3.1, we further parallelize Gibbs sampling in an exact manner via

[1]For larger datasets, this state is not saved and instead the counts are maintained using an exponential moving sum. Thus the memory require scales only with model size, not dataset size.

Table 1: BIDMach (CPU) vs. JAGS Runtime on Koller Data

	$m = 1$	$m = 2$	$m = 5$	$m = 10$	$m = 20$
BIDMach Total Time (sec)	11.6	18.6	33.7	62.9	116.0
JAGS Total Time (sec)	42.0	98.2	281.2	535.0	1037.6

chromatic partitioning. In Bayesian networks, nodes u and v are independent conditioned on a set of variables $\mathcal{C}$ if $\mathcal{C}$ includes at least one variable on every path connecting u and v in the *moralized graph* of the network, which is the graph formed by connecting parents and dropping edge orientations.

Suppose there is a k-coloring of the moralized graph. Let $\mathcal{V}_c$ be the set of variables assigned color c where $1 \leq c \leq k$. One can sample sequentially from $\mathcal{V}_1$ to $\mathcal{V}_k$, and within each color group, sample all its variables in parallel. This parallel sampler corresponds exactly to the execution of a sequential scan Gibbs sampler for some permutation over the variables and will converge to the desired distribution because variables within one color group are independent given all the other variables.

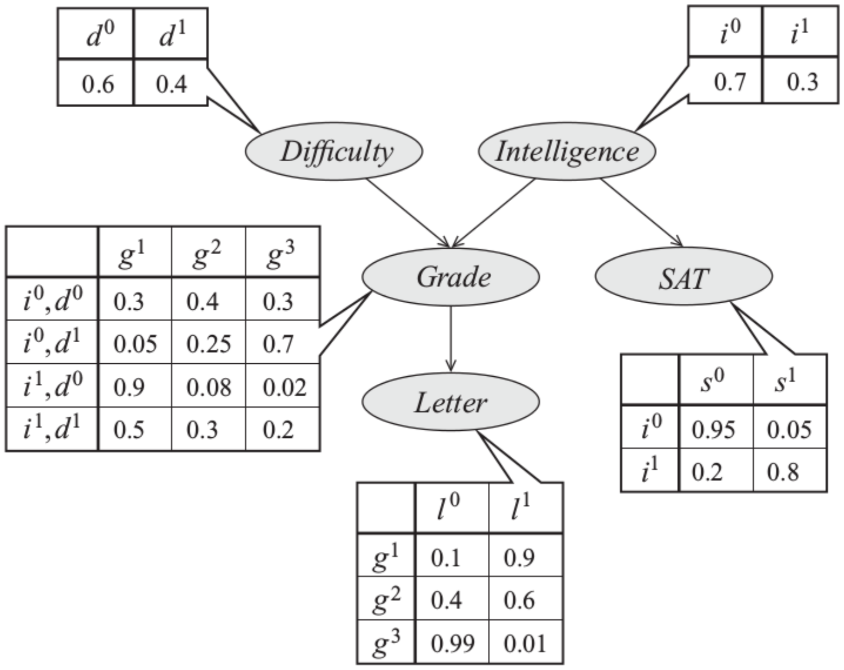
5 Evaluating our Gibbs Sampler

We benchmark our Gibbs sampler based on one synthetic and one real dataset. We compare it with JAGS [14], which is one of the most popular and efficient tool for Bayesian inference, and also uses Gibbs sampling as the primary inference algorithm.

For all JAGS experiments and for CPU benchmarks for BIDMach, we use a single computer with an Intel Core Xeon processor with eight cores and 2.20 GHz clock speed (E5-2650 Sandy Bridge). The computer has 64 GB of CPU RAM. We used a different machine with an NVIDIA Titan X GPU for the GPU experiments since the CPU characteristics of this second machine do not affect performance while using the GPU.

5.1 Synthetic "Koller" Data

We first use synthetic data generated from a small Bayesian network to check correctness and the use of SAME. The network has five variables: $X_0 = \text{Intelligence}$, $X_1 = \text{Difficulty}$, $X_2 = \text{SAT}$, $X_3 = \text{Grade}$, and $X_4 = \text{Letter}$. The directed edges are $\mathcal{E} = \{(X_0, X_2), (X_0, X_3), (X_1, X_3), (X_3, X_4)\}$, where (X_i, X_j) means an arrow points from X_i



d^0	d^1
0.6	0.4

i^0	i^1
0.7	0.3

	g^1	g^2	g^3
i^0,d^0	0.3	0.4	0.3
i^0,d^1	0.05	0.25	0.7
i^1,d^0	0.9	0.08	0.02
i^1,d^1	0.5	0.3	0.2

	s^0	s^1
i^0	0.95	0.05
i^1	0.2	0.8

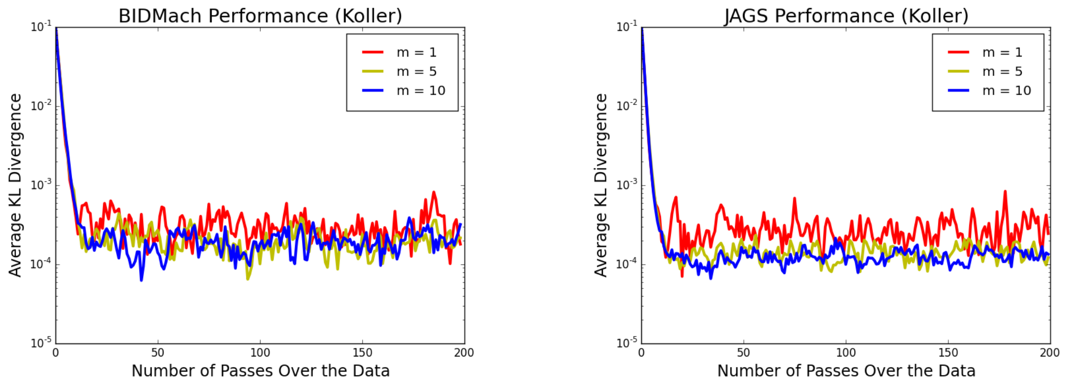
	l^0	l^1
g^1	0.1	0.9
g^2	0.4	0.6
g^3	0.99	0.01

Figure 4: The Bayesian network of Koller data [11]

to X_j. Variable X_3 is ternary, and all others are binary. This network models a student taking a class, and considers ability metrics (Intelligence and SAT score), the class difficulty, and the student's resulting grade, which subsequently affects the quality of a letter of recommendation. This network, along with the true set of CPTs, is from Chapter 3 of [11], see Fig 4. Due to the name of the author, we call this the "Koller" data to distinguish it from the MOOC data we use in (5.2).

To generate the data, we use the standard technique of forward sampling, where X_i gets sampled based on the true distributions from [11], which depends on X_i's parents (if any). We repeat this to get 50,000 samples, then randomly hide 50% of the data points. The goal is to apply Gibbs sampling to estimate the CPTs that generated the data. Note that BIDMach can handle millions of samples, but due to limitations of JAGS, we use only 50,000 for benchmarking.

To evaluate our Gibbs sampler, we compute the average KL Divergences of all the distributions in the set of CPTs, denoted as KL_{avg}. For two distributions $p(x)$ and $q(x)$, the KL divergence is $\sum_x p(x)\log(p(x)/q(x))$, summing over x such that $q(x) > 0$. In the Koller data, there are eleven probability distributions that form the set of CPTs. For ex-



(a) The KL_{avg} from BIDMach.

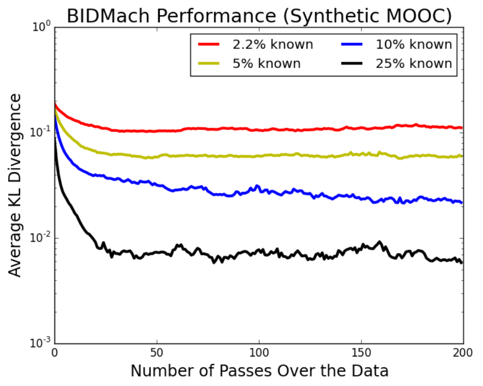
(b) The KL_{avg} from JAGS.

Figure 5: Plots of our KL_{avg} curves as a function of the iteration, on Koller data (these must be viewed in color). Both BIDMach and JAGS converge to the true CPTs quickly, but JAGS appears to benefit more from the extra data replication with SAME parameters $m = 5$ and $m = 10$.

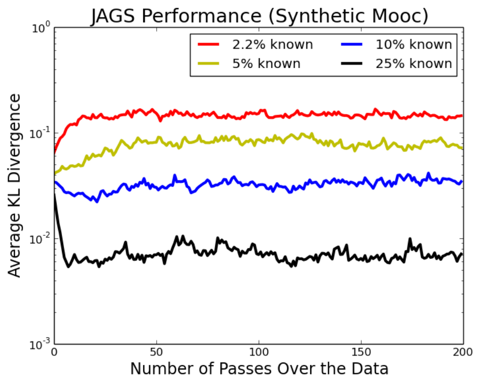
ample, X_4 "contributes" three distributions: $\Pr(X_4 \mid X_3 = 0), \Pr(X_4 \mid X_3 = 1)$, and $\Pr(X_4 \mid X_3 = 2)$, where X_3 (the parent) is fixed. For this network, we have $KL_{\text{avg}} = \frac{1}{11}\sum_{i=1}^{11} p_i(x)\log(p_i(x)/q_i(x))$, where q_i is the distribution our sampler estimates and i is some arbitrary indexing notation. We do not use the KL Divergence of the full joint distribution $P(X_1, X_2, X_3, X_4, X_5)$ since with high-dimensional data (e.g., the MOOC data in (5.2)), computing the full joint is intractable and we wish to facilitate comparisons across different datasets.

Figures 5a and 5b plot the KL_{avg} metric for the student data using BIDMach and JAGS, respectively with three SAME factors (note the log scale). The plots indicate that the KL_{avg} for BIDMach and JAGS reach roughly the same values, with a slight advantage to JAGS, though this is amplified because of the log scale. In practice, the difference would be indistinguishable to humans. For instance, with $m = 1$, the average difference between a random number in the sampled CPT and its corresponding number in the true CPT for BIDMach and JAGS, respectively, is 0.0039 and 0.0043. Thus, both BIDMach and JAGS will sample CPTs that are accurate to two/three decimal places.

For BIDMach, we tuned the minibatch size to be 12,500. Using a smaller size means that, for a fixed number of passes, we tend to get faster convergence, but this often comes with slower runtime per iteration. In addition, we observed that increasing m results in CPT estimates that more closely match the true CPTs. The red curves (i.e., $m = 1$) for Figures 5a and 5b generally correspond to worse KL_{avg} than the respective yellow and blue curves, with the



(a) The KL_{avg} from BIDMach.



(b) The KL_{avg} from JAGS.

Figure 6: Plots of our KL_{avg} curves as a function of the iteration, on Synthetic MOOC Data with different sparsity levels (these must be viewed in color).

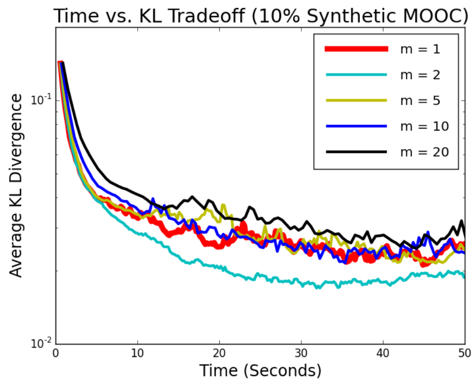
difference more noticeable for JAGS. Also, going from $m = 5$ to $m = 10$ seems to have a negligible benefit.

We further benchmark the speed of our CPU Gibbs sampler with JAGS on this data. For a fair comparison, we keep our minibatch size to be 12,500 to match the runs from Figure 5a. Table 1 shows the total runtime on the data with different m for 200 passes. We used total runtime because the JAGS API does not enable us to separate the initialization time from the updating/sampling time. (JAGS spends a substantial amount of time initializing since it needs to form a graph, but BIDMach uses matrices and spends less than one second in initialization.) The results demonstrate that BIDMach is almost four times faster than JAGS on the original data, and when the SAME parameter increases, the gap widens (up to a factor of nine with $m = 20$).

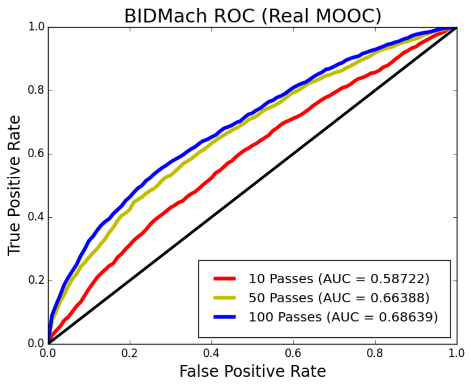
5.2 Dynamic Learning Maps "MOOC" Data

We now benchmark our code on a Bayesian network with a nation-wide examination dataset from the Dynamic Learning Maps (DLM) project[2]. The data contains the assessment (correct or not) of student responses to questions from the DLM Alternate Assessment System. After preprocessing, there are 4367 students and 319 questions. Each of the questions is derived from a set of 15 basic concepts. We encode questions and concepts as variables and describe their relationships with a Bayesian network from the DLM data. (The 15 concepts are latent variables across all students.) Each question is a variable with a very high missing

[2]URL: `http://dynamiclearningmaps.org/`. There is no official paper reference for this data.



(a) The KL_{avg} from BIDMach.



(b) MOOC data ROC curve.

Figure 7: Some more results (KL_{avg} and ROCs) of our sampler on Synthetic/Real MOOC data.

value rate. Our corresponding data matrix, which has dimensions (334×4367), has a 2.2% density level. The inference task is to learn the CPTs of the Bayesian network from this extremely sparse data. All variables are binary.

We evaluate our sampler using two methods. The first involves running it on the original data until convergence. Then we use the resulting estimated set of CPTs and sample from that via forward sampling to generate "Synthetic MOOC" data, also of dimension (334×4367). We create several versions of this data, each with a different fraction of missing data. Then we re-run BIDMach on these and evaluate the CPTs using the same KL_{avg} metric from (5.1). Computing KL_{avg} involves averaging over 682 distributions based on the Bayesian network, which has relatively few edges (the most amount of parents any node has is three).

Figures 6a and 6b plot the KL_{avg} for BIDMach and JAGS, respectively on four different levels of sparsity (note again the log scale). As expected, BIDMach and JAGS converge to roughly the same KL_{avg}. Furthermore, denser data allows the Gibbs sampler to get closer to the true CPTs. Note that BIDMach and JAGS start at different KL_{avg} values due to different initializations (BIDMach initializes randomly, while JAGS tends to start with "even" $(0.5, 0.5)$ distributions), and the JAGS curve increases for extremely sparse data because the initialization point happens to be better than to what it converges.

We also analyze the tradeoffs involved in increasing the SAME factor m. While increasing m intuitively (and theoretically) should lead to faster convergence to higher quality parameter estimates, it is possible that increasing m too much will result in getting stuck in a local

Table 2: BIDMach (CPU) vs. JAGS Runtime on (Replicated) Real MOOC Data

	1x	2x	5x	10x	20x	40x
GPU BIDMach Total Time (sec)	11.2	21.2	48.1	92.9	183.8	365.9
CPU BIDMach Total Time (sec)	39.5	76.1	187.3	359.6	701.4	1437.4
JAGS Total Time (sec)	975.2	2749.0	5830.0	18815.0	34309.0	OOM

maxima since this is a highly noncovex problem. Figure 7a plots[3] KL_{avg} as a function of total time elapsed. The $m = 2$ curve is clearly better as for a fixed time t, it has lower KL_{avg} than $m = 1$, despite how the $m = 1$ curve will have had more full passes over the data. Interestingly enough, $m = 5$ results in similar results, and the $m = 20$ curve indicates that higher m will not be beneficial. We ran the experiment for Figure 7a for 300 seconds and the longer-term trend was that the KL_{avg} values tended to remain at their 50-second values.

In addition to measuring distance between predicted and actual parameters, we also evaluate the sampler using its accuracy at predicting missing labels. We randomly split the original data into a training and testing batch so that 80% of the known data is in training. The training set is used to update the CPTs. During the testing phase, we sample n times using the current CPTs without updating them. For each (correct) test state, this yields an unbiased estimate of the correct state probability (note that this is a binary classification problem).

The training and testing data have 61.9% and 61.3% of known data points in the positive class, respectively. Due to this imbalance, we do not use raw accuracy, but instead use Receiver Operating Characteristic (ROC) curves to estimate the predictor's effectiveness. Figure 7b shows the ROC curves obtained after 10, 50, and 100 passes through the data, with increasing AUCs of 0.58722, 0.66388, and 0.68639, respectively, indicating that our sampler is effective on this extremely sparse data.

6 Speed and Scalability of Our Sampler on Large Data

We now discuss the extent to which BIDMach (and JAGS) can scale up to larger datasets in order to identify the limits of current software. To obtain arbitrarily large datasets, we replicated the MOOC data (from (5.2)) horizontally: we took the data matrix and made n copies to get a $(334 \times 4367n)$-dimensional data matrix. In Table 2, we report the total runtime for 200 iterations for BIDMach and JAGS on a variety of replicated data, from the original data (1x) to data replicated 40 times (40x). For our sampler, we used the same tunable settings

[3]The baseline $m = 1$ curve is thicker for readability.

Table 3: BIDMach (GPU) Runtime Per Iteration and GigaFlops on Large Data

	$m = 1$	$m = 2$	$m = 5$	$m = 10$
GigaFlops (Koller, 1M)	2.38	4.07	6.98	9.08
Time (sec) / Iteration (Koller, 1M)	0.307	0.359	0.523	0.804
GigaFlops (30x MOOC)	2.69	4.46	7.83	10.37
Time (sec) / Iteration (30x MOOC)	2.440	2.940	4.181	6.313

(e.g, minibatch size) as we did in Figure 6, to keep our benchmarks consistent.

The results indicate that BIDMach holds a clear speed advantage over JAGS (for both the CPU and GPU), though one must interpret the results carefully. From Figure 6, we see that, while BIDMach and JAGS both eventually converge to similar KL_{avg} values, our sampler takes roughly three times as many iterations to reach convergence (this is most easily observed on the 25% data).

Therefore, to determine BIDMach's speed advantages, one would need to divide the JAGS time by three (roughly). Despite this caveat, our sampler is still substantially faster. With the original data, BIDMach's advantage is a factor of $(975.2/3)/11.2 \approx 29.0$ (with the GPU), and the gap widens with more data (for 20x, we have an advantage of $(34309.0/3)/183.8 \approx 66.2$). In terms of memory usage, BIDMach also outperforms JAGS. BIDMach structures data in matrices, whereas JAGS internally forms a graph. For the 40x data, JAGS ran out of memory on our 64 GB CPU RAM machine, whereas BIDMach used up only 11.4% of the CPU RAM.

It is also worthy to investigate the impact of the SAME parameter m, in terms of runtime *and throughput*, which is measured in GigaFlops (gflops), a billion floating point operations per second. As the SAME parameter increases, it increases both the throughput and runtime by increasing the number of data points in our computations. The results from [19] suggest that increasing m for small values will increase throughput while not costing too much in runtime. As one increases m beyond a certain data-dependent value, then SAME "saturates" the algorithm and results in stagnant throughput while significantly increasing runtime. Figure 7a also raises the question as to whether one really needs a large m in the first place.

Table 3 reports the performance of BIDMach's GPU version (only) on two expanded datasets with varying m. One is the Koller data from (5.1), with 50% of the data known versus missing, but with *one million* cases. The other is 30x replicated MOOC data. Table 3 reports that BIDMach gets a healthy amount of throughput from the expanded data (especially the 30x

MOOC data), and that increasing m further is a logical possibility.

Finally, we point out that the runtimes in all tables in this paper are listed are in seconds, and even with large m and a large replication factor, we have barely scratched the limit of our sampler. To test the limit of our GPU sampler, we ran it on 300x replicated MOOC data with $m = 1$. Our sampler completed 200 iterations in 1726.1 seconds with a gflops of 7.59. In fact, the primary limiting factor of our sampler is not the size of the data itself, but the memory of the GPU; the 300x data used up almost all of the 12GB of GPU RAM. As we get better and faster GPUs, the performance of our sampler will improve and we will be able to run it on larger and larger datasets.

7 Conclusions

We conclude that our Gibbs sampler is much faster than the state of the art (JAGS) and can be applied to data with hundreds of discrete variables and millions of instances. We also suggest that SAME can be beneficial for Gibbs sampling, and that SAME Gibbs sampling should be the go-to method for researchers who wish to perform inference on (discrete) probabilistic graphical models. Future work will explore the application of our sampler to a wider class of real-world datasets.

References

[1] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[2] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, March 2003.

[3] John Canny and Huasha Zhao. Bidmach: Large-scale learning with zero memory allocation. In *BigLearn Workship, Neural Information Processing Systems*, 2013.

[4] Ronan Collobert, Koray Kavukcuoglu, and Clement Farabet. Torch7: A matlab-like environment for machine learning, 2011.

[5] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society, Series B*, 39(1):1–38, 1977.

[6] Arnaud Doucet, Simon J. Godsill, and Christian P. Robert. Marginal maximum a posteriori estimation using markov chain monte carlo. Technical report, Statistics and Computing, 2002.

[7] Joseph Gonzalez, Yucheng Low, Arthur Gretton, and Carlos Guestrin. Parallel gibbs sampling: From colored fields to thin junction trees. In *In Artificial Intelligence and Statistics (AISTATS)*, Ft. Lauderdale, FL, May 2011.

[8] T. L. Griffiths and M. Steyvers. Finding scientific topics. *Proceedings of the National Academy of Sciences*, 101(Suppl. 1):5228–5235, April 2004.

[9] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[10] Matthew Johnson, James Saunderson, and Alan Willsky. Analyzing hogwild parallel gaussian gibbs sampling. In C.J.C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 2715–2723. 2013.

[11] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press, 2009.

[12] M. Kubale. *Graph Colorings*. Contemporary mathematics (American Mathematical Society) v. 352. American Mathematical Society, 2004.

[13] Aaron Q. Li, Amr Ahmed, Sujith Ravi, and Alexander J. Smola. Reducing the sampling complexity of topic models. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 891–900, New York, NY, USA, 2014. ACM.

[14] Martyn Plummer. Jags: A program for analysis of bayesian graphical models using gibbs sampling, 2003.

[15] Bo Thiesson. Accelerated quantification of bayesian networks with incomplete data. In Usama M. Fayyad and Ramasamy Uthurusamy, editors, *KDD*, pages 306–311. AAAI Press, 1995.

[16] Martin J. Wainwright and Michael I. Jordan. Graphical models, exponential families, and variational inference. *Found. Trends Mach. Learn.*, 1(1-2):1–305, January 2008.

[17] Greg C. G. Wei and Martin A. Tanner. A Monte Carlo Implementation of the EM Algorithm and the Poor Man's Data Augmentation Algorithms. *Journal of the American Statistical Association*, 85(411):699–704, 1990.

[18] Ce Zhang and Christopher Ré. Towards high-throughput gibbs sampling at scale: A study across storage managers. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 397–408, New York, NY, USA, 2013. ACM.

[19] Huasha Zhao, Biye Jiang, John F. Canny, and Bobby Jaros. Same but different: Fast and high quality gibbs parameter estimation. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 1495–1502, New York, NY, USA, 2015. ACM.