Parallel Gibbs Sampling: From Colored Fields to Thin Junction Trees

Joseph E. Gonzalez
Carnegie Mellon University
jegonzal@cs.cmu.edu

Yucheng Low
Carnegie Mellon University
ylow@cs.cmu.edu

Arthur Gretton[1]
Gatsby Unit, UCL
arthur.gretton@gmail.com

Carlos Guestrin
Carnegie Mellon University
guestrin@cs.cmu.edu

Abstract

We explore the task of constructing a parallel Gibbs sampler, to both improve mixing and the exploration of high likelihood states. Recent work in parallel Gibbs sampling has focused on update schedules which do not guarantee convergence to the intended stationary distribution. In this work, we propose two methods to construct parallel Gibbs samplers guaranteed to draw from the targeted distribution. The first method, called the Chromatic sampler, uses graph coloring to construct a direct parallelization of the classic sequential scan Gibbs sampler. In the case of 2-colorable models we relate the Chromatic sampler to the Synchronous Gibbs sampler (which draws all variables simultaneously in parallel), and reveal new ergodic properties of Synchronous Gibbs chains. Our second method, the Splash sampler, is a complementary strategy which can be used when the variables are tightly coupled. This constructs and samples multiple blocks in parallel, using a novel locking protocol and an iterative junction tree generation algorithm. We further improve the Splash sampler through adaptive tree construction. We demonstrate the benefits of our two sampling algorithms on large synthetic and real-world models using a 32 processor multi-core system.

1 INTRODUCTION

Gibbs sampling is a popular MCMC inference procedure used widely in statistics and machine learning. On many models, however, the Gibbs sampler can be slow mixing [Kuss and Rasmussen, 2005, Barbu and Zhu, 2005]. Consequently, a number of authors [Doshi-Velez et al., 2009, Newman et al., 2007, Asuncion et al., 2008, Yan et al., 2009] have proposed parallel methods to accelerate Gibbs sampling. Unfortunately, most of the recent methods rely on **Synchronous** Gibbs updates that are not ergodic, and therefore generate chains that do not converge to the targeted stationary distribution.

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

In this work we propose two separate ergodic parallel Gibbs samplers. The first, called the **Chromatic** sampler, applies a classic technique relating graph coloring to parallel job scheduling, to obtain a direct parallelization of the classic sequential scan Gibbs sampler. We show that the Chromatic sampler is provably ergodic and provide strong guarantees on the parallel reduction in mixing time.

For the relatively common case of models with two-colorable Markov random fields, the Chromatic sampler provides substantial insight into the behavior of the non-ergodic Synchronous Gibbs sampler. We show that in the two-colorable case, the Synchronous Gibbs sampler is equivalent to the simultaneous execution of two independent Chromatic samplers and provide a method to recover the corresponding ergodic chains. As a consequence, we are able to derive the invariant distribution of the Synchronous Gibbs sampler and show that is ergodic with respect to functions over single variable marginals.

The Chromatic sampler achieves a linear increase in the rate at which samples are generated and is therefore ideal for models where the variables are weakly coupled. However, for models with strongly coupled variables, the chain can still mix prohibitively slowly. In this case, it is often necessary to jointly sample large subsets of related random variables [Barbu and Zhu, 2005, Jensen and Kong, 1996] in what is known as a blocking Gibbs sampler.

Our second parallel Gibbs sampler, the **Splash** sampler, addresses the challenges of highly correlated variables by incrementally constructing multiple bounded tree-width blocks, called Splashes, and then jointly sampling each Splash using parallel junction-tree inference and backward-sampling. To ensure that multiple simultaneous Splashes are conditionally independent (and hence that the chain is ergodic), we introduce a Markov blanket locking protocol. To accelerate burn-in and ensure high likelihood states

[1]Affiliated with CMU and MPI for Biological Cybernetics

are reached quickly, we introduce a vanishing adaptation heuristic for the initial samples of the chain, which explicitly builds blocks of strongly coupled variables.

We provide a highly tuned open-source implementation of both parallel samplers using the new GraphLab framework [Low et al., 2010] for parallel machine learning, and compare performance on synthetic and real-world sampling problems using a 32 processor multicore system. We find that both algorithms achieve strong speedups in sample generation, and the adaptive Splash sampler can further accelerate mixing on strongly correlated models. Our experiments illustrate that the two sampling strategies complement each other: for weakly coupled variables, the Chromatic sampler performs best, whereas the Splash sampler is needed when strong dependencies are present.

2 THE GIBBS SAMPLER

In this work we focus on large probabilistic models that can be represented as factorized distributions of the form:

$$\pi\left(x_1, \ldots, x_n\right) \propto \prod_{\mathbf{A}\in\mathcal{F}} f_{\mathbf{A}}(\mathbf{x}_{\mathbf{A}}), \tag{2.1}$$

where each clique $\mathbf{A} \in \mathcal{F}$ is a subset, $\mathbf{A} \subseteq \{1, \ldots, n\}$, of indices and the factors $f_{\mathbf{A}}$ are un-normalized positive functions, $f_{\mathbf{A}} : \mathbf{x}_{\mathbf{A}} \to \mathbb{R}^+$ over subsets of random variables. While we will only consider discrete random variables $X_i \in \Omega$, most of the techniques can be directly applied to continuous random variables.

Because the independence structure of Eq. (2.1) is central to the design of efficient parallel Gibbs samplers, we will rely heavily on the Markov Random Field (MRF). The MRF of π is an undirected graph over the variables where X_i is connected to X_j if there is a $\mathbf{A} \in \mathcal{F}$ such that $i, j \in \mathbf{A}$. The set of all variables $X_{\mathcal{N}_i}$ adjacent to variable X_i is called the **Markov Blanket** of X_i. A variable X_i is conditionally independent of all other variables given its Markov Blanket:

$$\pi\left(X_i \mid \mathbf{X}_{\mathcal{N}_i}\right) = \pi\left(X_i \mid \mathbf{X}_{-i}\right) \tag{2.2}$$

where $\mathbf{X}_{-i}$ refers to the set of all variables excluding the variable X_i.

The Gibbs sampler, introduced by Geman and Geman [1984], is a popular Markov Chain Monte Carlo (MCMC) algorithm used to simulate samples from the joint distribution π. The Gibbs sampler is constructed by iteratively sampling each variable,

$$X_i \sim \pi\left(X_i \mid \mathbf{X}_{\mathcal{N}_i} = \mathbf{x}_{\mathcal{N}_i}\right) \propto \prod_{\mathbf{A}: i\in\mathbf{A}, \mathbf{A}\in\mathcal{F}} f_{\mathbf{A}}(X_i, \mathbf{x}_{\mathcal{N}_i}) \tag{2.3}$$

given the assignment to the variables in its Markov blanket. Geman and Geman [1984] showed that if each variable is sampled infinitely often and under reasonable assumptions

Algorithm 1: The Synchronous Gibbs Sampler

```
1 forall Variables $X_j$ do in parallel
2     Execute Gibbs Update: $X_j^{(t+1)} \sim \pi\left(X_j \mid \mathbf{x}_{\mathcal{N}_j}^{(t)}\right)$
3 barrier end
```

on the conditional distributions (e.g., positive support), the Gibbs sampler is ergodic (i.e., it converges to the true distribution). While we have considerable latitude in the update schedule, we shall see in subsequent sections that certain updates must be treated with care: in particular, Geman and Geman were incorrect in their claim that parallel simultaneous sampling of all variables (the *Synchronous* update) yields an ergodic chain.

For large models with complex dependencies, the mixing time and even the time required to obtain a high likelihood sample can be substantial. Therefore, we would like to use parallel resources to increase the speed of the Gibbs sampler. The simplest method to construct a parallel sampler is to run a separate chain on each processor. However, running multiple parallel chains requires large amounts of memory and, more importantly, is not guaranteed to accelerate mixing or the production of high-likelihood samples. As a consequence, we focus on single chain parallel acceleration, where we apply parallel methods to increase the speed at which a single Markov chain is advanced. The single chain setting ensures that any parallel speedup directly contributes to an equivalent reduction in the mixing time, and the time to obtain a high-likelihood sample.

Unfortunately, recent efforts to build parallel single-chain Gibbs samplers have struggled to retain ergodicity. The resulting methods have relied on approximate sampling algorithms [Asuncion et al., 2008] or proposed generally costly extensions to recover ergodicity [Doshi-Velez et al., 2009, Newman et al., 2007, Ferrari et al., 1993]. A central objective in this paper is to design an efficient parallel Gibbs samplers while ensuring ergodicity.

3 THE CHROMATIC SAMPLER

A naive single chain parallel Gibbs sampler is obtained by sampling all variables simultaneously on separate processors. Called the **Synchronous** Gibbs sampler, this highly parallel algorithm (Alg. 1) was originally proposed by Geman and Geman [1984]. Unfortunately the extreme parallelism of the Synchronous Gibbs sampler comes at a cost. As others [e.g., Newman et al., 2007] have observed, one can easily construct cases (see Appendix A) where the Synchronous Gibbs sampler is not ergodic and therefore does not converge to the correct stationary distribution.

Fortunately, the parallel computing community has developed methods to directly transform sequential graph algorithms into equivalent parallel graph algorithms using graph colorings. Here we apply these techniques to obtain the ergodic **Chromatic** parallel Gibbs sampler shown in

Algorithm 2: The Chromatic Sampler

```
Input : $k$-Colored MRF
1 for For each of the $k$ colors $\kappa_i : i \in \{1, \ldots, k\}$ do
2     forall Variables $X_j \in \kappa_i$ in the $i^{th}$ color do in parallel
3         Execute Gibbs Update:
          $X_j^{(t+1)} \sim \pi\left(X_j \mid \mathbf{x}^{(t+1)}_{\mathcal{N}_j \in \kappa_{<i}}, \mathbf{x}^{(t)}_{\mathcal{N}_j \in \kappa_{>i}}\right)$
4     barrier end
5 end
```

Alg. 2. Let there be a k-coloring of the MRF such that each vertex is assigned one of k colors and adjacent vertices have different colors. Let κ_i denote the variables in color i. Then the Chromatic sampler simultaneously draws new values for all variables in κ_i before proceeding to κ_{i+1}. The k-coloring of the MRF ensures that all variables within a color are conditionally independent given the variables in the remaining colors and can therefore be sampled independently and in parallel.

By combining a classic result ([Bertsekas and Tsitsiklis, 1989, Proposition 2.6]) from parallel computing with the original Geman and Geman [1984] proof of ergodicity for the sequential Gibbs sampler one can easily show:

Proposition 3.1 (**Graph Coloring and Parallel Execution**). *Given p processors and a k-coloring of an n-variable MRF, the parallel Chromatic sampler is ergodic and generates a new joint sample in running time:*

$$O\left(\frac{n}{p} + k\right).$$

Proof. From [Bertsekas and Tsitsiklis, 1989, Proposition 2.6] we know that the parallel execution of the Chromatic sampler corresponds exactly to the execution of a sequential scan Gibbs sampler for some permutation over the variables. The running time can be easily derived:

$$O\left(\sum_{i=1}^{k} \left\lceil \frac{|\kappa_i|}{p} \right\rceil\right) = O\left(\sum_{i=1}^{k} \left(\frac{|\kappa_i|}{p} + 1\right)\right) = O\left(\frac{n}{p} + k\right).$$

□

Therefore, given sufficient parallel resource ($p \in O(n)$) and a k-coloring of the MRF, the parallel Chromatic sampler has running-time $O(k)$, which for many MRFs is constant in the number of vertices. It is important to note that this parallel gain directly results in a factor of p reduction in the mixing time.

Unfortunately, constructing the minimal coloring of a general MRF is NP-Complete. However, for many common models the optimal coloring can be quickly derived. For example, given a plate diagram, we can typically compute an optimal coloring of the plates, which can then be applied to the complete model. When an optimal coloring cannot be trivially derived, we find simple graph coloring heuristics (see Kubale [2004]) perform well in practice.

3.1 Properties of 2-Colorable Models

Many popular models in machine learning have natural two-colorings. For example, Latent Dirichlet Allocation, the Indian Buffet process, the Boltzmann machine, hidden Markov models, and the grid models commonly used in computer vision all have two-colorings. For these models, the Chromatic sampler provides substantial insight into properties of the Synchronous sampler. The following theorem relates the Synchronous Gibbs sampler to the Chromatic sampler in the two-colorable setting and provides a method to recover two ergodic chains from a single Synchronous Gibbs chain:

Theorem 3.2 (**2-Color Ergodic Synchronous Samples**). *Let $(X^{(t)})_{t=0}^m$ be the non-ergodic Markov chain constructed by the Synchronous Gibbs sampler (Alg. 1) then using only $(X^{(t)})_{t=0}^m$ we can construct two ergodic chains $(Y^{(t)})_{t=0}^m$ and $(Z^{(t)})_{t=0}^m$ which are conditionally independent given $X^{(0)}$ and correspond to the simultaneous execution of two Chromatic samplers (Alg. 2).*

Proof. We split the chain $(X^{(t)})_{t=0}^m = (X^{(t)}_{\kappa_1}, X^{(t)}_{\kappa_2})_{t=0}^m$ over the two colors and then construct the chains $(Y^{(t)})_{t=0}^m$ and $(Z^{(t)})_{t=0}^m$ by simulating the two Chromatic Gibbs samplers, which each advance only one color at a time conditioned on the other color (as illustrated in Fig. 1):

$$\left(Y^{(t)}\right)_{t=0}^{m} = \left[\begin{pmatrix} X^{(0)}_{\kappa_1} \\ X^{(1)}_{\kappa_2} \end{pmatrix}, \begin{pmatrix} X^{(2)}_{\kappa_1} \\ X^{(1)}_{\kappa_2} \end{pmatrix}, \begin{pmatrix} X^{(2)}_{\kappa_1} \\ X^{(3)}_{\kappa_2} \end{pmatrix}, \ldots\right]$$

$$\left(Z^{(t)}\right)_{t=0}^{m} = \left[\begin{pmatrix} X^{(1)}_{\kappa_1} \\ X^{(0)}_{\kappa_2} \end{pmatrix}, \begin{pmatrix} X^{(1)}_{\kappa_1} \\ X^{(2)}_{\kappa_2} \end{pmatrix}, \begin{pmatrix} X^{(3)}_{\kappa_1} \\ X^{(2)}_{\kappa_2} \end{pmatrix}, \ldots\right]$$

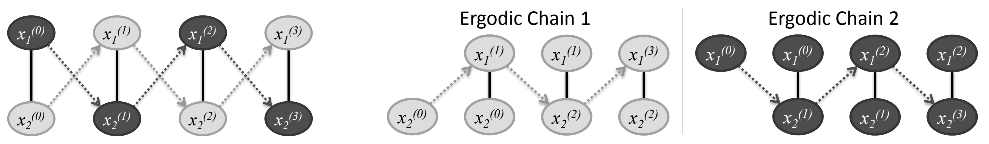
Observe that no samples are shared between chains, and given $X^{(0)}$ both chains are independent. Finally, because both derived chains are simulated from the Chromatic sampler they are provably ergodic. □

Using the partitioning induced by the 2-coloring of the MRF we can analytically construct the invariant distribution of the Synchronous Gibbs sampler:

Theorem 3.3 (**Invariant Distribution of Sync. Gibbs**). *Let $(X_{\kappa_1}, X_{\kappa_2}) = X$ be the partitioning of the variables over the two colors, then the invariant distribution of the Synchronous Gibbs sampler is the product of the marginals $\pi(X_{\kappa_1})\pi(X_{\kappa_2})$.*

Proof. Let the current state of the sampler be $X = (X_{\kappa_1}, X_{\kappa_2})$ and the result of a single Synchronous Gibbs update be $X' = (X'_{\kappa_1}, X'_{\kappa_2})$. By definition the Synchronous Gibbs update simulates $X'_{\kappa_1} \sim \pi(x'_{\kappa_1} \mid x_{\kappa_2})$ and $X'_{\kappa_2} \sim \pi(x'_{\kappa_2} \mid x_{\kappa_1})$. Therefore the transition kernel for the Synchronous Gibbs sampler is:

$$K(x' \mid x) = \pi(x'_{\kappa_1} \mid x_{\kappa_2})\pi(x'_{\kappa_2} \mid x_{\kappa_1})$$



(a) Synchronous Chain (b) Two Ergodic Chains

Figure 1: **(a)** Execution of a two colored model using the synchronous Gibbs sampler. The dotted lines represent dependencies between samples. **(b)** Two ergodic chains obtained by executing the Synchronous Gibbs sampler. Note that ergodic sums with respect to marginals are equivalent to those obtained using the Synchronous sampler.

We can easily show that $\pi(X_{\kappa_1})\pi(X_{\kappa_2})$ is the invariant distribution of the Synchronous Gibbs sampler:

$$\begin{aligned} \mathbf{P}(x') &= \sum_x K(x' \mid x)\pi(x) \\ &= \sum_{x_{\kappa_1}} \sum_{x_{\kappa_2}} \pi(x'_{\kappa_1} \mid x_{\kappa_2})\pi(x'_{\kappa_2} | x_{\kappa_1})\pi(x_{\kappa_1})\pi(x_{\kappa_2}) \\ &= \sum_{x_{\kappa_1}} \sum_{x_{\kappa_2}} \pi(x_{\kappa_1}, x'_{\kappa_2})\pi(x'_{\kappa_1}, x_{\kappa_2}) \\ &= \pi(x'_{\kappa_1})\pi(x'_{\kappa_2}) \end{aligned}$$

□

A useful consequence of Theorem 3.3 is that when computing ergodic averages over sets of variables with the same color, we can directly use the non-ergodic Synchronous samples and still obtain convergent estimators:

Corollary 3.4 (**Monochromatic Marginal Ergodicity**). *Given a sequence of samples $(x^{(t)})_{t=0}^m$ drawn from the Synchronous Gibbs sampler on two-colorable model, empirical expectations computed with respect to single color marginals are ergodic:*

$$\forall f, i \in 1, 2 : \lim_{m \to \infty} \frac{1}{m} \sum_{t=0}^{m} f(x_{\kappa_i}^{(t)}) \xrightarrow{a.s.} E_\pi \left[f(X_{\kappa_i}) \right]$$

Corollary 3.4 therefore justifies many applications where the Synchronous Gibbs sampler is used to estimate single variables marginals and explains why the Synchronous Gibbs sampler performs well in these settings. However, Corollary 3.4 also highlights the danger of computing empirical expectations over variables that span both colors without splitting the chains as shown in Theorem 3.2.

We have shown that both the Chromatic sampler and the Synchronous sampler can provide ergodic samples. However the Chromatic sampler is clearly superior when the number of processors is less than the half the number of vertices ($p < n/2$) since it will advance a single chain twice as fast as the Synchronous sampler.

4 THE PARALLEL SPLASH SAMPLER

The Chromatic sampler provides a linear speedup for single-chain sampling, advancing the Markov chain for a k-colorable model in time $O\left(\frac{n}{p} + k\right)$ rather than $O(n)$.

Algorithm 3: Parallel Splash Sampler

```
  Input: Maximum treewidth $w_{\max}$
  Input: Maximum Splash size $h_{\max}$
1 while $t \leq \infty$ do
      // Make $p$ bounded treewidth Splashes
3     $\{\mathbf{J}_{\mathcal{S}_i}\}_{i=1}^p \leftarrow$ ParSplash($w_{max}, h_{max}, x^{(t)}$);
      // Calibrate each junction trees
5     $\{\mathbf{J}_{\mathcal{S}_i}\}_{i=1}^p \leftarrow$ ParCalibrate($x^{(t)}, \{\mathbf{J}_{\mathcal{S}_i}\}_{i=1}^p$);
      // Sample each Splash
7     $\{x_{\mathcal{S}_i}\}_{i=1}^p \leftarrow$ ParSample($\{\mathbf{J}_{\mathcal{S}_i}\}_{i=1}^p$);
      // Advance the chain
8     $x^{(t+1)} \leftarrow \left\{ x_{\mathcal{S}_1}, ..., x_{\mathcal{S}_1}, x^{(t)}_{\neg \bigcup_{i=1}^p \mathcal{S}_i} \right\}$
```

Unfortunately, some models possess strongly correlated variables and complex dependencies, which can cause the Chromatic sampler to mix prohibitively slowly.

In the single processor setting, a common method to accelerate a slowly mixing Gibbs sampler is to introduce blocking updates [Barbu and Zhu, 2005, Jensen and Kong, 1996, Hamze and de Freitas, 2004]. In a blocked Gibbs sampler, blocks of strongly coupled random variables are sampled jointly conditioned on their combined Markov blanket. The blocked Gibbs sampler improves mixing by enabling strongly coupled variables to update jointly when individual conditional updates would cause the chain to mix too slowly.

To improve mixing in the parallel setting we introduce the **Splash** sampler (Alg. 3), a general purpose blocking sampler. For each joint sample, the Splash sampler exploits parallelism both to construct multiple random blocks, called Splashes, and to accelerate the joint sampling of each Splash. To ensure each Splash can be safely and efficiently sampled in parallel, we developed a novel Splash generation algorithm which incrementally builds multiple conditionally independent bounded treewidth junction trees for every new sample. In the initial rounds of sampling, the Splash algorithm uses a novel adaptation heuristic which groups strongly dependent variables together based on the state of the chain. Adaptation is then disabled after a finite number of rounds to ensure ergodicity.

We present the Splash sampler in three parts. First, we present the parallel algorithm used to construct multiple conditionally independent Splashes. Next, we describe the parallel junction tree sampling procedure used to jointly sample all variables in a Splash. Finally, we present our

Algorithm 4: `ParSplash`: Parallel Splash Generation

```
Input: Maximum treewidth $w_{\max}$
Input: Maximum Splash size $h_{\max}$
Output: Disjoint Splashes $\{\mathcal{S}_1, \ldots, \mathcal{S}_p\}$
1  do in parallel on processor $i \in \{1, \ldots, p\}$
3      $r \leftarrow$ NextRoot($i$) // Unique roots
4      $\mathcal{S}_i \leftarrow \{r\}$ // Add $r$ to splash
5      $\mathcal{B} \leftarrow \mathcal{N}_r$ // Add neighbors to boundary
6      $\mathcal{V} \leftarrow \{r\} \cup \mathcal{N}_r$ // Visited vertices
7      $\mathbf{J}_{\mathcal{S}_i} \leftarrow$ JunctionTree($\{r\}$)
8      while $(|\mathcal{S}_i| < h_{max}) \bigwedge (|\mathcal{B}| > 0)$ do
10         $v \leftarrow$ NextVertexToExplore($\mathcal{B}$)
11         MarkovBlanketLock($X_v$)
           // Check that $v$ and its neighbors $\mathcal{N}_v$
              are not in other Splashes.
12         safe $\leftarrow \left|(\{v\} \cup \mathcal{N}_v) \cap \left(\bigcup_{j \neq i} \mathcal{S}_j\right)\right| = 0$
13         $\mathbf{J}_{\mathcal{S}_i+v} \leftarrow$ ExtendJunctionTree($\mathbf{J}_{\mathcal{S}_i}$, $v$)
14         if safe $\bigwedge$ TreeWidth($\mathbf{J}_{\mathcal{S}_i+v}$) $< w_{max}$ then
15             $\mathbf{J}_{\mathcal{S}_i} \leftarrow \mathbf{J}_{\mathcal{S}_i+v}$ // Accept new tree
16             $\mathcal{S}_i \leftarrow \mathcal{S}_i \cup \{v\}$
18             $\mathcal{B} \leftarrow \mathcal{B} \cup (\mathcal{N}_v \backslash \mathcal{V})$ // Extend boundary
19             $\mathcal{V} \leftarrow \mathcal{V} \cup \mathcal{N}_v$ // Mark visited
20         MarkovBlanketFree($X_v$)
```

Algorithm 5: `ExtendJunctionTree` Algorithm

```
Input: The original junction tree $(\mathcal{C}, E) = \mathbf{J}_{\mathcal{S}}$.
Input: The variable $X_i$ to add to $\mathbf{J}_{\mathcal{S}}$
Output: $\mathbf{J}_{\mathcal{S}+i}$
Define : $\mathbf{C}_u$ as the clique created by eliminating $u \in \mathcal{S}$
Define : V[$\mathbf{C}$] $\in \mathcal{S}$ as the variable eliminated when creating $\mathbf{C}$
Define : t[$v$] as the time $v \in \mathcal{S}$ was added to $\mathcal{S}$
Define : P[$v$] $\in \mathcal{N}_v \cap \mathcal{S}$ as the next neighbor of $v$ to be
         eliminated.
1  $\mathbf{C}_i \leftarrow (\mathcal{N}_i \cap \mathcal{S}) \cup \{i\}$
2  P[$i$] $\leftarrow \arg\max_{v \in \mathbf{C}_i \backslash \{i\}}$ t[$v$]
   // ----------- Repair RIP -------------
3  $\mathcal{R} \leftarrow \mathbf{C}_i \backslash \{i\}$ // RIP Set
4  $v \leftarrow$ P[$i$]
5  while $|\mathcal{R}| > 0$ do
6      $\mathbf{C}_v \leftarrow \mathbf{C}_v \cup \mathcal{R}$ // Add variables to parent
7      $w \leftarrow \arg\max_{w \in \mathbf{C}_v \backslash \{v\}}$ t[$w$] // Find new parent
8      if $w =$ P[$v$] then
9          $\mathcal{R} \leftarrow (\mathcal{R} \backslash \mathbf{C}_i) \backslash \{i\}$
10     else
11         $\mathcal{R} \leftarrow (\mathcal{R} \cup \mathbf{C}_i) \backslash \{i\}$
12         P[$v$] $\leftarrow w$ // New parent
13     $v \leftarrow$ P[$v$] // Move upwards
```

Splash adaptation heuristic which sets the priorities used during Splash generation.

4.1 Parallel Splash Generation

The Splash generation algorithm (Alg. 4) uses p processors to incrementally build p disjoint Splashes in parallel. Each processor grows a Splash rooted at a unique vertex in the MRF (Line 3). To preserve ergodicity we require that no two roots share a common edge in the MRF, and that every variable is a root infinitely often.

Each Splash is grown incrementally using a best first search (BeFS) of the MRF. The exact order in which variables are explored is determined by the call to `NextVertexToExplore(`$\mathcal{B}$`)` on Line 10 of Alg. 4 which selects (and removes) the next vertex from the boundary $\mathcal{B}$. In Fig. 2 we plot several simultaneous Splashes constructed using a first-in first-out (FIFO) ordering (Fig. 2(b)) and a prioritized ordering (Fig. 2(d)).

The Splash boundary is extended until there are no remaining variables that can be safely added or the Splash is sufficiently large. A variable cannot be safely added to a Splash if sampling the resulting Splash is excessively costly (violates a treewidth bound) or if the variable or any of its neighbors are members of other Splashes (violates conditional independence of Splashes).

To bound the computational complexity of sampling, and later to jointly sample the Splash, we rely on junction trees. A junction tree, or clique graph, is an undirected acyclic graphical representation of the joint distribution over a collection of random variables. For a Splash containing the variables $X_{\mathcal{S}}$, we construct a junction tree $(\mathcal{C}, E) = \mathbf{J}_{\mathcal{S}}$ representing the conditional distribution $\pi(X_{\mathcal{S}} \mid x_{-\mathcal{S}})$. The vertices $\mathbf{C} \in \mathcal{C}$ are often called cliques and represent a subset of the indices (i.e., $\mathbf{C} \subseteq \mathcal{S}$) in the Splash $\mathcal{S}$. The cliques satisfy the constraint that for every factor domain $\mathbf{A} \in \mathcal{F}$ there exists a clique $\mathbf{C} \in \mathcal{C}$ such that $\mathbf{A} \cap \mathcal{S} \subseteq \mathbf{C}$. The edges E of the junction tree satisfy the running intersection property (RIP) which ensures that all cliques sharing a common variable form a connected tree.

The computational complexity of a inference, and consequently sampling in a junction tree, is exponential in the treewidth; one less than number of variables in the largest clique. Therefore, to evaluate the computational cost of adding a new variable X_v to the Splash, we need an efficient method to extend the junction tree $\mathbf{J}_{\mathcal{S}}$ over $X_{\mathcal{S}}$ to a junction tree $\mathbf{J}_{\mathcal{S}+v}$ over $X_{\mathcal{S} \cup \{v\}}$ and evaluate the resulting treewidth.

To efficiently build incremental junction trees, we developed a novel junction tree extension algorithm (Alg. 5) which emulates standard variable elimination, with variables being eliminated in the reverse of the order they are added to the Splash (e.g., if X_i is added to $\mathbf{J}_{\mathcal{S}}$ then X_i is eliminated before all $X_{\mathcal{S}}$). Because each Splash grows outwards from the root, the resulting elimination ordering is optimal on tree MRFs and typically performs well on cyclic MRFs.

The incremental junction tree extension algorithm (Alg. 5) begins by eliminating X_i and forming the new clique $\mathbf{C}_i = (\mathcal{N}_i \cap \mathcal{S}) \cup \{i\}$ which is added $\mathbf{J}_{\mathcal{S}+i}$. We then attach $\mathbf{C}_i$ to the *most recently added* clique $\mathbf{C}_{\mathrm{P}[i]}$ which contains a variable in $\mathbf{C}_i$ ($\mathbf{C}_{\mathrm{P}[i]}$ denotes the parent of $\mathbf{C}_i$). We then restore the RIP by propagating the newly added variables back up the tree. Letting $\mathcal{R} = \mathbf{C}_i \backslash \{i\}$, we insert $\mathcal{R}$ into its

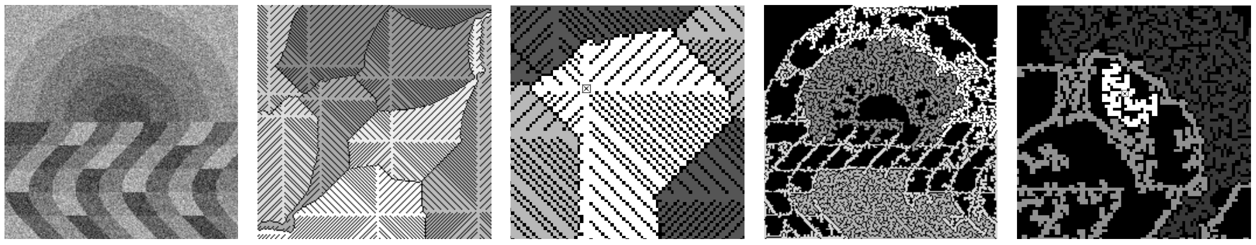


Figure 2: Different Splashes constructed on a 200 × 200 image denoising grid MRF. **(a)** A noisy sunset image. Eight Splashes of treewidth 5 were constructed using the FIFO **(b)** and priority **(d)** ordering. Each splash is shown in a different shade of gray and the black pixels are not assigned to any Splash. The priorities were obtained using the adaptive heuristic. In **(c)** and (e) we zoom in on the Splashes to illustrate their structure and the black pixels along the boundary needed to maintain conditional independence.

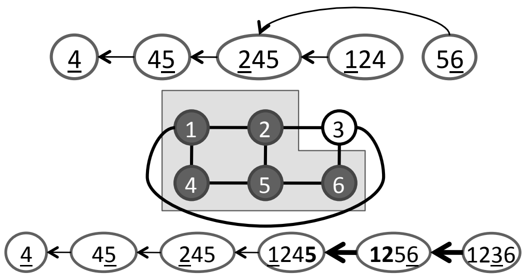


Figure 3: **Incremental Junction Tree Example**: The junction tree on the top comprises the subset of variables {1,2,4,5,6} of the MRF (center). The tree is formed by the variable elimination ordering {6,1,2,5,4} (reading the underlined variables of the tree in reverse). To perform an incremental insertion of variable 3, we first create the clique formed by the elimination of 3 ({1,2,3,6}) and insert it into the end of the tree. Its parent is set to the latest occurrence of any of the variables in the new clique. Next the set {1,2,6} is inserted into its parent (boldface variables), and its parent is recomputed in the same way.

parent clique $\mathbf{C}_{\mathrm{P}[i]}$. The RIP condition is now satisfied for variables in $\mathcal{R}$ which were already in $\mathbf{C}_{\mathrm{P}[i]}$. The parent for $\mathbf{C}_{\mathrm{P}[i]}$ is then recomputed, and any unsatisfied variables are propagated up the tree in the same way. We demonstrate this algorithm with a simple example in Fig. 3.

To ensure that simultaneously constructed Splashes are conditionally independent, we develop the Markov blanket locking (MBL) protocol which associates a lock with each variable in the model. The Markov blanket lock for variable X_v is obtained by acquiring the read-locks on all neighboring variables $X_{\mathcal{N}_v}$ and the write lock on X_v. Locks are acquired and released using a canonical ordering of the variables to prevent deadlocks.

Once the `MarkovBlanketLock(`X_v`)` has been acquired, no other processor can assign X_v or any of it neighbors $X_{\mathcal{N}_v}$ to a Splash. Therefore, we can safely test if X_v or any of its neighbors $X_{\mathcal{N}_v}$ are currently assigned to other Splashes. Since we only add X_v to the Splash if both X_v and all its neighbors are currently unassigned to other Splashes, there will never be an edge in the MRF that connects two Splashes. Consequently, simultaneously constructed Splashes are conditionally independent given all remaining unassigned variables.

4.2 Parallel Splash Sampling

Once we have constructed p conditionally independent Splashes $\{\mathcal{S}_1\}_{i=1}^{p}$, we jointly sample each Splash by drawing from $\pi(X_{\mathcal{S}_i} \mid x_{-\mathcal{S}_i})$ in parallel. This is accomplished by calibrating the junction trees $\{\mathbf{J}_{\mathcal{S}_i}\}_{i=1}^{p}$, and then running backward-sampling starting at the root to jointly sample all the variables in each Splash. We also use the calibrated junction trees to construct Rao-Blackwellized marginal estimators. If the treewidth or the size of each Splash is large, it may be beneficial to construct fewer Splashes, and instead assign multiple processors to accelerate the calibration and sampling of individual junction trees.

To calibrate the junction tree we use the `ParCalibrate` function. The `ParCalibrate` function constructs all clique potentials in parallel by computing the products of the assigned factors conditioned on the variables not in the Splash. Finally, parallel belief propagation is used to calibrate the tree by propagating messages in parallel following the optimal forward-backward schedule.

Parallel backward-sampling is accomplished by the function `ParSample` which takes the calibrated junction tree and draws a new joint assignment in parallel. The `ParSample` function begins by drawing a new joint assignment for the root clique using the calibrated marginal. Then in parallel each child is sampled conditioned on the parent assignment and the messages from the children.

4.3 Adaptive Splash Generation

As discussed earlier, the order in which variables are explored when constructing a Splash is determined on Line 10 in the `ParSplash` algorithm (Alg. 4). We propose a simple adaptive prioritization heuristic, based on the current assignment to $x^{(t)}$, that prioritizes variables at the boundary of the current tree which are strongly coupled with variables already in the Splash. We assign each variable $X_v \in \mathcal{B}$ a score using the likelihood ratio:

$$\mathbf{s}[X_v] = \left|\left| \log \frac{\sum_x \pi\left(X_{\mathcal{S}}, X_v = x \mid X_{-\mathcal{S}} = x_{-\mathcal{S}}^{(t)}\right)}{\pi\left(X_{\mathcal{S}}, X_v = x_v^{(t)} \mid X_{-\mathcal{S}} = x_{-\mathcal{S}}^{(t)}\right)} \right|\right|_1, \tag{4.1}$$

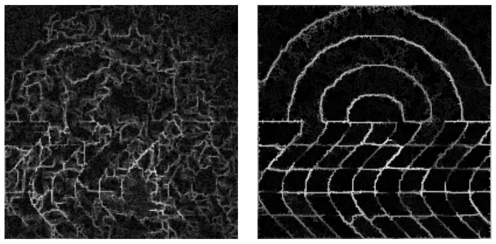


Figure 4: The update frequencies of each variable in the 200×200 image denoising grid MRF for the synthetic noisy image shown in Fig. 2(a). The brighter pixes have been prioritized higher and are therefore updated more frequently. **(a)** The early update counts are relatively uniform as the adaptive heuristic has not converged on the priorities. **(b)** The final update counts are focused on the boundaries of the regions in the model corresponding to pixels that can be most readily changed by blocked steps.

and includes the variable with the highest score. Effectively, Eq. (4.1) favors variables with greater average log likelihood than conditional log likelihood. We illustrate the consequence of applying this metric to an image denoising task in which we denoise the synthetic image shown in Fig. 2(a). In Fig. 4 we show a collection of Splashes constructed using the score function Eq. (4.1). To see how priorities evolve over time, we plot the update frequencies early (Fig. 4(a)) and later (Fig. 4(b)) in the execution of the Splash scheduler.

To ensure the chain remains ergodic, we disable the prioritized tree growth after a finite number of iterations, and replace it with a random choice of variables to add (we call this *vanishing adaptation*). Indeed, a Gibbs chain cannot be ergodic if the distribution over variables to sample is a function of the current state of the chain. This result may appear surprising, as it contradicts Algorithm 7.1 of Levine and Casella [2006], of which our adaptive algorithm is an example: Levine and Casella claim this to be a valid adaptive Gibbs sampler. We are able to construct a simple counter example, using two uniform independent binary random variables (see Appendix C), thus disproving the claim. By contrast, we show in Appendix B that our Splash sampler with vanishing adaptation is ergodic:

Theorem 4.1 (Splash Sampler Ergodicity). *The adaptive Splash sampler with vanishing adaptation is ergodic and converges to the true distribution π.*

5 EXPERIMENTS

We implemented an optimized C++ version of both the Chromatic and Splash samplers for arbitrary discrete factorized models and all our code has been released[1] along with a user-friendly Matlab interface. Our implementation was built using the open-source GraphLab API [Low et al., 2010]. The GraphLab API provides the graph based locking routines needed to implement the Markov blanket locking protocol. The GraphLab API also substantially simplifies the design and implementation of the Chromatic sampler, which uses the highly-tuned lock-free GraphLab scheduler and built-in graph coloring tools.

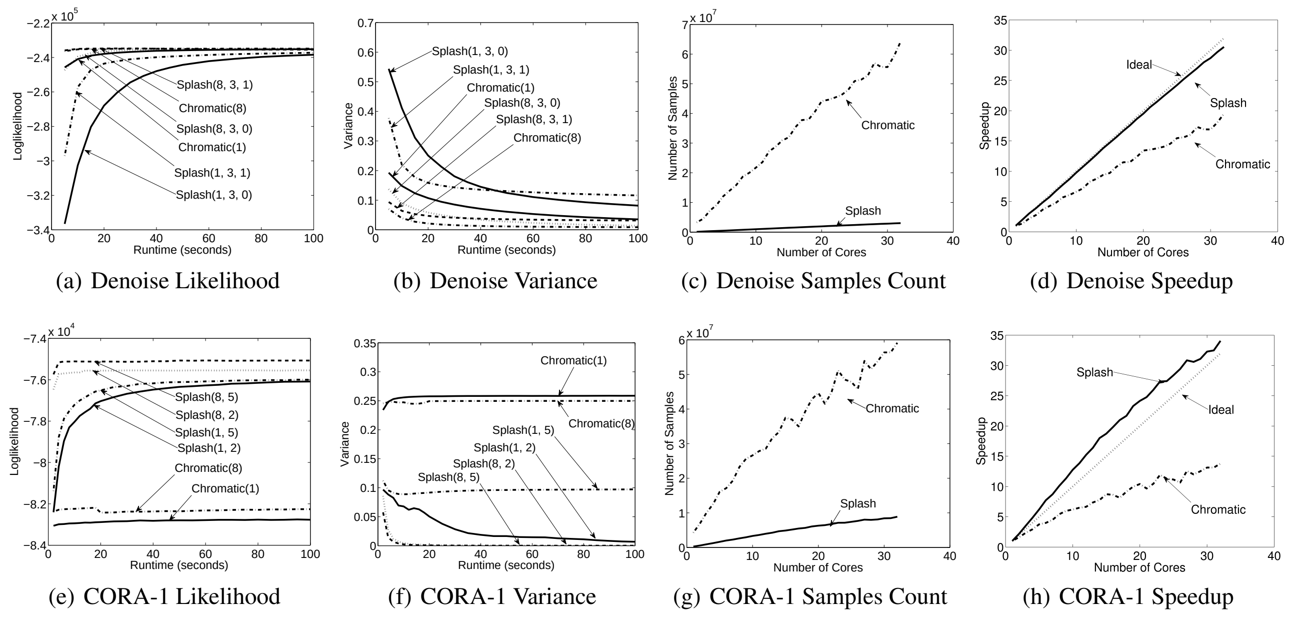
[1]`http://www.select.cs.cmu.edu/code`

Although Alg. 3 is presented as a sequence of synchronous parallel steps, our implementation splits these steps over separate processors to maximize performance and eliminate the need for threads to join between phases. We also implemented the parallel junction tree calibration and sampling algorithms using the GraphLab API. However, we found that for the typically small maximum treewidth used in our experiments, the overhead associated the additional parallelism overrode any gains. Nonetheless, when we made the treewidth sufficiently large (e.g., 10^6 sized factors) we were able to obtain $13\times$-speedup on 32 cores.

To evaluate the proposed algorithms in both the weakly and strongly correlated setting we selected two representative large-scale models. In the weakly correlated setting we used a $40,000$ variable 200×200 grid MRF similar to those used in image processing. The latent pixel values were discretized into 5 states. Gibbs sampling was used to compute the expected pixel assignments for the synthetic noisy image shown in Fig. 2(a). We used Gaussian node potentials centered around the pixel observations with $\sigma^2 = 1$ and Ising-Potts edge potentials of the form $\exp(-3\delta(x_i \neq x_j))$. To test the algorithms in the strongly correlated setting, we used the CORA-1 Markov Logic Network (MLN) obtained from Domingos [2009]. This large real-world factorized model consists of over $10,000$ variables and $28,000$ factors, and has a much higher connectivity and higher order factors than the pairwise MRF.

In Fig. 5 we present the results of running both algorithms on both models using a state-of-the-art 32 core Intel Nehalem (X7560) server with hyper-threading disabled. We plot un-normalized log-likelihood and across chain variance in-terms of wall-clock time. In Fig. 5(a) and Fig. 5(e) we plot the un-normalized log-likelihood of the last sample as a function of time. While in both cases the Splash algorithm out-performs the chromatic sampler, the difference is more visible in the CORA-1 MLN. We found that the adaptation heuristic had little effect in likelihood maximization on the CORA-1 MLN, but did improve performance on the denoising model by focusing the Splashes on the higher variance regions. In Fig. 5(b) and Fig. 5(f) we plot the variance in the expected variable assignments across 10 independent chains with random starting points. Here we observe that for the faster mixing denoising model, the increased sampling rate of the Chromatic sampler leads to a greater reduction in variance while in the slowly mixing CORA-1 MLN only the Splash sampler is able to reduce the variance.

To illustrate the parallel scaling we plot the number of samples generated in a 20 seconds (Fig. 5(c) and Fig. 5(d)) as



(a) Denoise Likelihood (b) Denoise Variance (c) Denoise Samples Count (d) Denoise Speedup

(e) CORA-1 Likelihood (f) CORA-1 Variance (g) CORA-1 Samples Count (h) CORA-1 Speedup

Figure 5: Comparison of Chromatic sampler and the Splash sampler at different settings (i.e., Chromatic(p), Splash(p, $w_{\max}$, adaptation) for p processors and treewidth $w_{\max}$) on the synthetic image denoising grid model and the Cora Markov logic network. Adaptation was not used in the CORA-1 MLN. **(a,e)** The un-normalized log-likelihood plotted as a function of running-time. **(b,f)** The variance in the estimator of the expected assignment computed across 10 independent chains with random starting points. **(c,g)** The total number of variables sampled in a 20 second window plotted as a function of the number of cores. **(d,h)** The speedup in number of samples drawn as a function of the number of processors.

well as the speedup in sample generation (Fig. 5(c) and Fig. 5(d)). The speedup is computed by measuring the multiple of the number of samples generated in 20 seconds using a single processor. The ideal speedup is linear with 32× speedup on 32 cores.

We find that the Chromatic sampler typically generates an order of magnitude more samples per second than the more costly Splash sampler. However, if we examine speedup curves we see that the larger cost associated with the Splash construction and inference contributes to more exploitable coarse grain parallelism. Interestingly, in Fig. 5(h) we see that the Splash sampler exceeds the ideal scaling. This is actually a consequence of the high connectivity forcing each of the parallel Splashes to be smaller as the number of processors increases. As a consequence the cost of computing each Splash is reduced and the sampling rate increases. However, this also reduces some of the benefit from the Splash procedure as the size of each Splash is smaller resulting a potential increase in mixing time.

6 CONCLUSION

We have proposed two ergodic parallel single chain Gibbs samplers for high-dimensional models: the Chromatic sampler, and the Splash sampler, both implemented using the GraphLab framework. The Chromatic parallel Gibbs sampler can be applied where single variable updates still mix well, and uses graph coloring techniques to schedule conditionally independent updates in parallel. We related Chromatic sampler to the commonly used (but non-ergodic) Synchronous Gibbs sampler, and showed that we can recover two ergodic chains from a single non-ergodic Synchronous Gibbs chain.

In settings with tightly couples variables, the parallelism afforded by the Chromatic Gibbs sampler may be insufficient to achieve rapid mixing. We therefore proposed the Splash Gibbs sampler which incrementally constructs multiple conditionally independent bounded treewidth blocks (Splashes) in parallel. To construct the Splash sampler we developed a novel incremental junction tree construction algorithm which quickly and efficiently updates the junction tree as new variables are added. We further proposed a procedure to accelerate burn-in by explicitly grouping strongly dependent variables, which is disabled after the initial samples are drawn to ensure ergodicity. An interesting topic for future work is whether one can design an adaptive parallel sampler (i.e., one that *continually* modifies its behavior depending on the current state of the chain) that still retains ergodicity.

Acknowledgments

We thank Yee Whye Teh and Maneesh Sahani for helpful discussions. This work is supported by the ONR Young Investigator Program grant N00014-08-1-0752, the ARO under MURI W911NF0810242, and the ONR PECASE-N00014-10-1-0672. Joseph Gonzalez is supported by a Graduate Research Fellowship from the National Science Foundation.

References

M. Kuss and C. E. Rasmussen. Assessing approximate inference for binary gaussian process classification. *J. Mach. Learn. Res.*, 6, 2005.

A. Barbu and S. Zhu. Generalizing swendsen-wang to sampling arbitrary posterior probabilities. *IEEE Trans. Pattern Anal. Mach. Intell.*, 27(8), 2005.

F. Doshi-Velez, D. Knowles, S. Mohamed, and Z. Ghahramani. Large scale nonparametric bayesian inference: Data parallelisation in the indian buffet process. In *NIPS 22*, 2009.

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

A. Asuncion, P. Smyth, and M. Welling. Asynchronous distributed learning of topic models. In *NIPS*, 2008.

F. Yan, N. Xu, and Y. Qi. Parallel inference for latent dirichlet allocation on graphics processing units. In *NIPS*, 2009.

C. S. Jensen and A. Kong. Blocking gibbs sampling for linkage analysis in large pedigrees with many loops. In *American Journal of Human Genetics*, 1996.

Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. Graphlab: A new framework for parallel machine learning. In *UAI*, 2010.

S. Geman and D. Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. In *PAMI*, 1984.

P. A. Ferrari, A. Frigessi, and R. H. Schonmann. Convergence of some partially parallel gibbs samplers with annealing. *The Annals of Applied Probability*, 3(1), 1993.

D. Bertsekas and J. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific, 1989.

M. Kubale. *Graph Colorings*. American Mathematical Society, 2004.

F. Hamze and N. de Freitas. From fields to trees. In *UAI*, 2004.

R. A. Levine and G. Casella. Optimizing random scan gibbs samplers. *J. Multivar. Anal.*, 97(10), 2006.

P. Domingos. Uw-cse mlns, 2009. URL `alchemy.cs.washington.edu/mlns/cora`.