Map/Reduce Uncollapsed Gibbs Sampling for Bayesian Non Parametric Models

Melanie F. Pradier
Department of Signal Theory
and Communications
University Carlos III in Madrid, Spain
melanie@tsc.uc3m.es

Pablo G. Moreno
Department of Signal Theory
and Communications
University Carlos III in Madrid, Spain
pgmoreno@tsc.uc3m.es

Francisco J. R. Ruiz
Department of Signal Theory
and Communications
University Carlos III in Madrid, Spain
franrruiz@tsc.uc3m.es

Isabel Valera
Department of Signal Theory
and Communications
University Carlos III in Madrid, Spain
ivalera@tsc.uc3m.es

Harold Molina-Bulla
Department of Signal Theory
and Communications
University Carlos III in Madrid, Spain
hmolina@tsc.uc3m.es

Fernando Perez-Cruz[*]
Department of Signal Theory
and Communications
University Carlos III in Madrid, Spain
fernando@tsc.uc3m.es

Abstract

We propose a Map/Reduce parallel Gibbs sampling framework for Bayesian nonparametric (BNP) models. Our proposal relies on the idea of sampling all the hidden variables in order to break their dependencies, as it has been proposed in the related literature. We present it in a way that is easily generalizable and readily applicable for any BNP model with a likelihood function in the exponential family. We release our code in Spark/Scala for the Dirichlet and Beta processes, which allows us to run one iteration over 50M observations in less than two minutes.

1 Introduction

Bayesian nonparametric (BNP) models are ideally suited for the challenges in the analysis of *big data* [15]. BNP models promise to find the latent structure in the data without restrictions and, as the amount of data increases, they should be able to incorporate new information seamlessly [16]. When the amount of data is massive, the main challenge is not to reduce a desired error measure, which any sensible algorithm would do, but providing interpretable explanations for the available data in a reasonable amount of time. However, BNP models often present inability to deal with a large number of records in a timely fashion, which is not due to the models themselves, but it is a consequence of the typical inference algorithms.

Although by no means Gibbs sampling is the only inference procedure for BNP (we discuss variational approximations in Section 2), it is commonly used because it is simple and allows integrating out variables to accelerate the convergence of the Markov Chain Monte Carlo (MCMC) [19, 10, 7].

[*]Also member of the Technical Staff at Bell Labs, Alcatel-Lucent, NJ. Email: Perez-Cruz@Alcatel-Lucent.com

Gibbs sampling in this way is suited for estimating a low number of variables in a single core machine. Yet, the number of hidden variables in BNP models grows with the number of instances, which results in a super-exponential increase in the number of possible explanations that fit the data. As a consequence, for large datasets, Gibbs sampling, or any other sampler for that matter, is no longer able to explore (in a finite number of iterations) the whole posterior distribution, and, in the best case, it only samples from its typical sequences [4]. Hence, the ability of Gibbs sampling to provide a solution to the inference problem in BNP lies in the fact that the posterior typical sequences are meaningful for the model we are estimating (e.g., in clustering with Dirichlet processes (DPs), we are interested in the fractions of instances assigned to each cluster, not in the individual assignments). For large databases, we cannot assess or even expect that the MCMC has converged. Hence, we run our sampler for as long as we can, hoping that we have gathered enough of the typical sequences to summarize a meaningful part of the posterior defined by our model.

Nowadays, computational resources are parallel and distributed (e.g., graphical processing units, computer clusters or cloud computing). The development of software frameworks such as Hadoop[1] or Spark [27], combined with programming languages suited for them (e.g., Scala,[2] R or Python), allow us to efficiently exploit these architectures. This has led to a significant amount of literature (reviewed in Section 2) in which sampling algorithms are successfully adapted for parallel architectures, by cleverly breaking the dependency between the latent variables for each datum. However, most of these algorithms are presented in an *ad hoc* manner giving raise to model-dependent software implementations that do not generalize across different BNP models.

In this paper, we propose a general Map/Reduce framework for parallel inference in BNP models and we use it to develop a new general inference software that can be used for a wide range of BNP models. We show that using this framework, many BNP model posteriors can be sampled in parallel, such that we should expect an almost linear speed-up with the number of available machines/cores when applied to datasets with millions or tens of millions of instances. Our parallel sampling algorithm relies on a variant of the uncollapsed Gibbs sampling algorithm. This approach does not significantly differ from many of the approaches proposed in our literature review, which are also similar among them. We believe that the novelty of our approach relies on the way we present the algorithm, since we provide a unifying perspective that establish the guidelines to implement parallel or distributed inference algorithms for many BNP models. Based on this approach, we provide an efficient implementation that results in a general inference software that can be used for two of the most popular applications of BNP models: clustering and feature allocation.

Our software implements a scalable sampler for a DP mixture model [19] and for the Indian buffet process (IBP) [10] in Spark/Scala (the extension to the hierarchical DP [23] is straightforward), because we believe it is the most efficient platform for Map/Reduce tasks using Hadoop resources [27]. Also, the generality of the inference framework allows the user to easily extend the software defining other observation models adapted to their needs. We will release with this paper a parallel implementation (single machine with multiple cores) and a distributed implementation (multiple machines with multiple cores), as well as the sequential Gibbs sampler used for the experiments.

2 Related work

With the development of suitable hardware for big data, there has been an increasing amount of works to adapt Bayesian inference to such computer architectures. Two general trends can be found in the literature: variational Bayesian inference techniques and MCMC methods.

Variational methods tackle the inference task as a non-convex optimization problem. They approximate the intractable posterior distribution with a tractable variational parametric distribution that relies on independence assumptions for easier updates of the variational parameters. Those parameters are optimized by minimizing the Kullback-Leibler divergence between the true posterior and the variational distribution. Stochastic variational inference [21] allows scaling variational methods for big data. However, by searching only within a restricted class of distributions we might lose some of the expressiveness of the model, and will typically obtain less accurate results than MCMC methods, which asymptotically sample from the true posterior [25].

[1]http://hadoop.apache.org

[2]http://www.scala-lang.org

Gibbs sampling is one of the most popular MCMC methods and it is widely used in machine learning and statistics, due to its simplicity and attractive asymptotic properties. Gibbs sampling cycles through every sample sequentially and it needs several iterations to forget the initial conditions and return samples from the posterior. As the amount of data grows, the run-time complexity for each iteration grows linearly with it, and a larger number of iterations are needed to forget the initial conditions because each sample makes a small contribution. Gibbs sampling in this way is not scalable, as it becomes prohibitively expensive to run it sequentially when dealing with large databases, and we are always left with the feeling that the MCMC has not properly mixed. Moreover, for massive datasets that cannot be stored in memory, it needs to constantly fetch blocks of data, further slowing down the whole process.

Novel computer architectures are parallel and distributed, and the challenge is to adapt inference algorithms to exploit these architectures in order to get a linear speed-up with the number of nodes. There have been several attempts to parallelize MCMC sampling for diverse probabilistic models using these architectures, such as latent Dirichlet allocation [20, 26], DPs [25, 18], hierarchical DPs (HDP) [25, 2], infinite hidden Markov models [3], Indian buffet processes (IBP) [8], infinite relational models [11], or Markov random fields [9]. Among these works we can find two main approaches for parallelizing Gibbs sampling, either through independence assumptions that maintain ergodicity [9, 25, 18, 8], or approximations that break some of the dependencies [20, 25, 11]. Similarly to our proposal, most of these works resort to auxiliary latent variables such that, conditioned on them, the observations are independent, allowing for parallel updates of the local parameters (e.g., the cluster allocation in the case of the DP mixture model). However, most of these algorithms are, in turn, presented in an *ad hoc* manner, being hard to generalize to other models. For instance, in the case of parallel Gibbs for the DP, the authors of [18] exploit the structure of DP mixture models to develop a two-stage implementation: first they run in parallel K independent DPs over non-overlapping subsets of the data; and then a central node collects all the information of each local DP to combine the local clusters into super-clusters.

Despite the huge amount of works on parallel MCMC techniques, the reality is that most of the machine learning community continues to use sequential inference techniques, limiting the scope of BNP models to small databases. To this end, we provide a simple and general approach based on an uncollapsed Gibbs sampler for parallel inference in many BNP models. The proposed approach, similarly to [12], resorts to sampling the latent measure. Given this latent measure, all the observations are independent, allowing us for a complete parallelization in the updates of the local variables. Moreover, it also allows for efficient implementation in terms of memory management, which is crucial for handling massive databases, because we do not need to store in memory the local variables (e.g., the cluster allocation for each datum in the DP), but only the sufficient statistics necessary to compute the posterior. Hence, the resulting sampler fits perfectly in a Map/Reduce framework that is readily generalizable for many BNP models, allowing one to efficiently deal with millions or tens of millions of instances (with the right architecture).

3 Inference

Many BNP models of interest can be simplified to the graphical model in Figure 1, where x_n represents the exchangeable observations and z_n represents a latent variable that only influences x_n. Additionally, β represents a general stochastic process, which allows for an open-ended number of degrees of freedom in the model [14], and α is the vector of hyperparameters. This general model includes, but is not limited to, the DP, HDP and IBP mixture models. For instance, in the DP Gaussian mixture model, z_n represents the cluster assignment for each observation, and β is the latent measure containing the mixing proportions (atom weights) and the cluster means (atom locations).

The joint distribution of the observations and latent variables is given by

$$p(\beta, \mathbf{x}, \mathbf{z}|\alpha) = p(\beta|\alpha) \prod_{n=1}^{N} p(x_n, z_n|\beta), \tag{1}$$

where $\mathbf{x} = \{x_1, \ldots, x_N\}$ and $\mathbf{z} = \{z_1, \ldots, z_N\}$. We assume that the likelihood function lies in the exponential family, and therefore it can be written as

$$p(x_n, z_n|\beta) = h(x_n, z_n) \exp\left\{\eta^\top(\beta) t(x_n, z_n) - a(\eta(\beta))\right\}, \tag{2}$$

where $h(x_n, z_n)$, $\eta(\cdot)$, $t(x_n, z_n)$ and $a(\cdot)$ stand for the base measure, natural parameters, sufficient statistics, and log-normalizer, respectively. Every exponential family likelihood has a conjugate prior $p(\beta|\alpha)$ [6] and, therefore, we can integrate out β and run a collapsed Gibbs sampling procedure in which we sequentially sample each local variable z_n from $p(z_n|\mathbf{z}_{-n}, \mathbf{x}, \alpha)$, where $\mathbf{z}_{-n} = \{z_1, \ldots, z_{n-1}, z_{n+1}, \ldots, z_N\}$.

According to de Finetti's theorem, the exchangeable observations $x_1, \ldots, x_N$ are conditionally independent given some latent variables to which an epistemic probability distribution would then be assigned [5]. The latent variables z_n are also exchangeable and, by de Finetti's theorem, there is a latent variable that makes them conditionally independent, so we should be able to sample the variables z_n in parallel given that latent variable. For the general BNP model in Figure 1, it is obvious that the latent variable that makes the local variables z_n conditionally independent is β. Therefore, instead of integrating out β, we can sample from it. Hence, we can readily derive a Gibbs sampling procedure that samples from the posterior $p(\beta, \mathbf{z}|\mathbf{x}, \alpha)$:

1. Sample β from $p(\beta|\mathbf{z}, \mathbf{x}, \alpha)$.
2. Sample z_n from $p(z_n|x_n, \beta)$ for $n = 1, \ldots, N$.

In this procedure, the computationally demanding step for big data is the second step, which can be run in up to N parallel machines/cores.

The first step needs to be executed on a central node. After sampling, the central node distributes β to the other nodes/cores, and each of them samples its local variable z_n. After the second step, each node responds with the sufficient statistics $t(x_n, z_n)$ and the central node updates the summation $\sum_n t(x_n, z_n)$, which is needed to sample β afterwards. Note that our algorithm does not need to store the N local latent variables z_n, since this information is summarized in the sufficient statistics. This provides us with significant memory and communication savings, which is crucial when dealing with massive databases. Also note that we do not need conjugacy as we do not integrate out β.

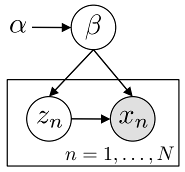


Figure 1: Graphical model for a general BNP model.

3.1 Dirichlet process

The DP is the de Finetti measure for the Dirichlet process mixture model [14], which extends the classical mixture model with a finite number of components. A draw from a DP is almost surely a discrete probability distribution with infinite support[3] and, therefore, it can be characterized by its atom locations and its atom weights, also referred to as "stick proportions" [13].

In the DP mixture model, variables z_n represent the cluster assignment of each observation, and sampling the global variable β requires sampling the atom weights and the atom locations (or per-cluster parameters). For the atom weights, the local sufficient statistic can be written as $t(x_n, z_n) = \mathbb{1}[z_n = k]$ for $k = 1, \ldots, K_+$ (being K_+ the current number of "active" clusters), and the complete conditional distribution over the weight vector π is given by Dirichlet$(N_1, \ldots, N_{K_+}, \gamma)$, where $N_k = \sum_n \mathbb{1}[z_n = k]$. Here, γ is the concentration parameter, and a draw for this Dirichlet distribution contains the atom weights for the K_+ active clusters, $\pi_1, \ldots, \pi_{K_+}$, and the probability mass of the infinitely many clusters that have not been assigned to any datum, represented by $\pi_{K_+ + 1}$. Additionally, if we consider the DP mixture of Gaussians, the local sufficient statistics for the atom locations (cluster means) for each of the K_+ active clusters is given by $\mathbb{1}[z_n = k]x_n$.

In order to allow the sampler to create new clusters, and following the idea in Algorithm 8 in [19], at each iteration we propose T new clusters and sample the atom weights from

$$\pi \sim \text{Dirichlet}\Big(N_1, \ldots, N_{K_+}, \underbrace{\frac{\gamma}{T}, \ldots, \frac{\gamma}{T}}_{T \text{ times}}\Big), \tag{3}$$

[3]Assuming either a continuous base measure or a discrete base measure with infinite support.

for some fixed value of T. Note that we do not truncate the number of clusters, but instead propose T new atom locations (dishes) at each iteration of the sampler, with their cluster-specific parameters drawn from the prior. Our sampling procedure converges to the true posterior distribution for the same reasons discussed in [19].

Therefore, our algorithm proceeds as follows:

1. Centralized step: Remove empty clusters and sample global variables. Sample the atom weights π (it involves assigning non-zero probability to T new potential clusters) and the per-cluster parameters.
2. Parallel step: Sample the cluster assignment z_n for each datum.

If we apply this inference procedure for an infinite mixture of Gaussians, in the first step we sample the proportions of the clusters and the proportions of new potential clusters. For the existing clusters we sample their posterior means, while for the new clusters we sample the cluster means from the prior. If β contains any other global variables (such as γ or the cluster covariance matrices), we also sample them from their posterior.

3.2 Beta process

The Beta process (BP) is the de Finetti measure for the IBP [24]. Again, if the base measure is continuous, a draw from a BP yields an almost surely discrete random measure, and can be represented by an infinite collection of both atom locations and probabilities in the unit interval. These probabilities are converted into binary-valued features by drawing from a Bernoulli process afterwards, yielding the IBP. The BP and the Bernouilli process are conjugate, and therefore the posterior distribution of the random measure is also distributed following a BP.

Therefore, in the IBP model, the local variables z_n correspond to the binary latent feature vectors for each data point, and the global variables β are the probabilities of each feature (atom weights) and their atom locations. In this case, the sufficient statistics for the feature probabilities are the number of instances that have each feature active, i.e., $N_k = \sum_n z_{nk}$. The problem of sampling from the posterior of a BP has been widely studied in the literature [24, 17]. We make use of the stick-breaking construction of the IBP [22], where the posterior distribution of the atom weights π_k is a Beta$(N_k, N - N_k + 1)$ distribution for $k = 1, \dots, K_+$, being K_+ the current number of active latent features. In addition, under the standard linear-Gaussian IBP [10], the sufficient statistics for the atom locations are given by $\sum_n z_{nk} x_n$ and $\sum_n z_{nk} x_n^\top x_n$ for $k = 1, \dots, K_+$.

When dealing with *Big data*, we cannot rely on the slice sampling scheme in [22] to propose new new latent features, because for large N we run into numerical instabilities. Instead, we use a two-step heuristic that performs well in our experiments. In particular, we sample the total probability mass corresponding to the sum of the stick proportions (which can be done exactly), and then propose new latent features with random stick proportions until the remaining probability mass is negligible. Further details about this procedure are provided in the Supplementary Material. As before, the global variables corresponding to the new atom locations are sampled from the prior.

We can summarize our inference algorithm for the IBP as follows:

1. Centralized step: Remove empty latent variables and sample global variables. Sample the atom weights π_k for existing latent variables, as well as for potential new features, and the parameters for each latent feature.
2. Parallel step: Sample the binary latent feature vector z_n for each datum.

Let us remark that, since the complete conditional distributions only depend on the sufficient statistics detailed above, we do not need to store nor distribute the $N \times K_+$ latent feature matrix containing the local feature allocations z_n, which leads to significant savings of memory and communications, allowing us to deal with big data in distributed architectures.

4 Software implementation

Map/Reduce Justification. In order to speed-up algorithms using parallelization techniques and the Map/Reduce framework, two conditions have to be satisfied: each operation must be independent, and the results from these operations must be endowed with an associative and commutative operation. Our sampler described in Section 3 fulfills both conditions, as each sufficient statistic is calculated independently, and the sampling outcomes (sufficient statistics $t(x_n, z_n)$ and per-datum log-likelihood) are additive. Hence, we can use the Map/Reduce framework to distribute our sampling procedure benefiting from multiple cores and/or multiple machines running in parallel.

Implementation Description. We consider two approaches. For moderate amount of data we use a shared memory multicore machine (parallel implementation) and for large amounts of data we use several parallel multicore machines[4] (distributed implementation). The first implementation is limited by the memory of the multicore machine and the number of cores. The second implementation is limited by the communication burden. At some stage, adding more machines and reducing the number of samples processed at each machine does not result in a speed-up, as communication overhead becomes more important.

In the parallel implementation, we use the parallel iterator implementation to calculate simultaneously the sufficient statistics for each datum and, at the end, apply a reduce operator to merge all the partial results. Both data and results are stored in memory. The use of Scala programming language and immutable structures allows developing a thread safe parallel application. We show a scheme of the parallel implementation in Figure 2. Note that the log-likelihood is not strictly needed to apply the algorithm, but it is useful for assessing convergence. This scheme is memory limited, since all the information (including the dataset itself) is stored in the computer's memory. In the distributed

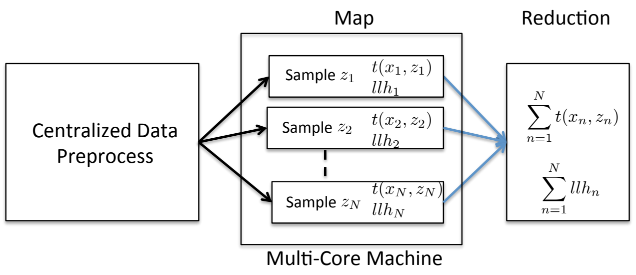


Figure 2: Process distribution in a multicore machine (*llh* stands for log-lokelihood).

implementation, we use several machines with multiple cores and we also distribute the data among the different machines (so that it is not memory limited). Algorithm 1 summarizes the process.

Algorithm 1 Distributed parallel sampling.

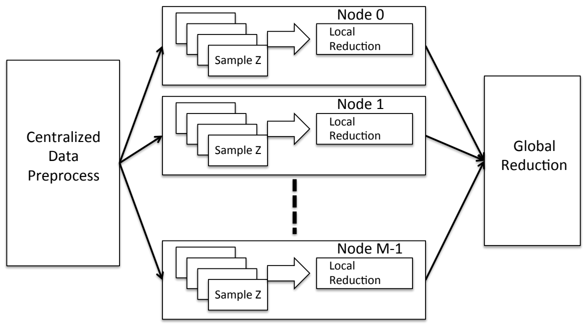
```
1:  Split the data into chunks.
2:  Store the pieces in HDFS.
3:  Initialize the task scheduler.
4:  while it<maxIter do
5:      Sample the shared variable $\beta$.
6:      Distribute to each node: chunk reference and $\beta$.
7:      Sample the per-datum variables in each node and return sum of local sufficient statistics.
8:      Apply reduce operator to join all the sub-results.
9:      Clean empty clusters/features.
10: end while
```

Following Amdahl's law [1], the expected speed-up is less than the number of parallel distributed subprocesses, and it depends on the percentage of sequential code and the overhead introduced by the communications between nodes. We want the computational time for each node to be long compared to the time spent in communications. There is a trade-off between the size of the data in each machine (memory and time to process each chunk of data) and the communication burden. Using this design, the computer's memory only limits the chunk's size, and the global database can grow as much as the distributed file system resources allow. In order to reduce the time spent in communications, we distribute a chunk reference to the secondary nodes instead of the data itself.

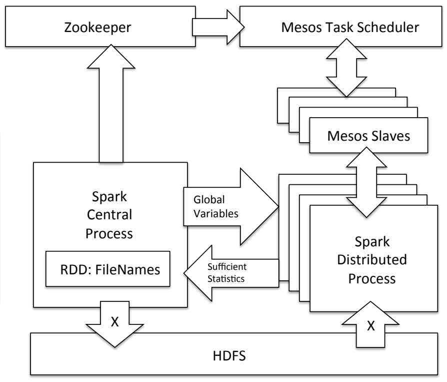
[4]For our architecture and experiments, $N = 10^6$ was a good trade-off value, but this value is obviously dependent on the specific configuration of the computer cluster and the given problem.

In order to deploy this infrastructure, we use the Apache Spark framework[5] to handle the communications and distribution of processes, over an infrastructure based on Apache Mesos[6] as task scheduler, Apache Zookeeper[7] for fault tolerance, and Apache Hadoop Distributed File System (HDFS) to store the data information. Spark is a fast and powerful engine for processing Hadoop data,[8] which allows us to distribute and process data stored in a Hadoop resource with high performance, due to its Resilent Distributed Datasets (RDD) [27]. Such approach stores in memory the results fetched from the distributed clients, and improves the speed in the final operations, as opposed to Hadoop, which stores all the intermediate data in the distributed file system.

One of the weaknesses of Scala is that it runs over the Java Virtual Machine (JVM), which implies an overhead when compared with a native program written in C or any medium level programming language. To alleviate this problem, we use specialized libraries for complex processes that use callbacks to high performance native libraries. For example, to handle matrices, we use the Breeze library,[9] which calls the native BLAS implementation, therefore speeding up the code execution.



(a) Process distribution.



(b) Platform architecture.

Figure 3: Implementation of the distributed algorithm.

5 Experiments

We study the performance of the parallel and distributed implementations for the DP and the BP (results for the BP are shown in the Supplementary Material). For comparisons, we also implement, for each model, a sequential uncollapsed Gibbs sampler, which presents linear complexity with the number of data. In the following, the parallel simulations are run in a machine with 16 cores, while for the distributed implementation we use at most 80 cores (distributed in 5 identical machines).

Synthetic data. We generate a synthetic dataset from a 10-dimensional mixture of Gaussians. We use 50 equally probable components, each with mean distributed according to a (10-dimensional) Uniform$(0, 10)$, and fixed noise variance of 0.01. The synthetic dataset allows us to verify that the algorithm recovers the ground truth. We run the three algorithms (sequential, parallel and distributed with chunks of 100K) for different dataset sizes, initialized with $K_+ = 100$ clusters. Our results are shown in Figure 4. In particular, Figure 4a shows the evolution with time of the mean log-likelihood for 1M observations. Both the parallel and the distributed implementations present similar convergence performance, while the sequential approach is not able to achieve comparable log-likelihood values in 10 hours. This result can be understood from Figure 4b, which presents the average time (over 24 hours) per iteration of the different algorithms for different amounts of data. Note that, for 1M observations, the time per iteration of the sequential implementation is about one order of magnitude above the parallel and distributed algorithms, which explains its slower convergence observed in Figure 4a. The dashes in the table indicate that this set-up could not be run due to either computational complexity or memory issues. For instance, the parallel implementation is only able

[5]http://spark.apache.org

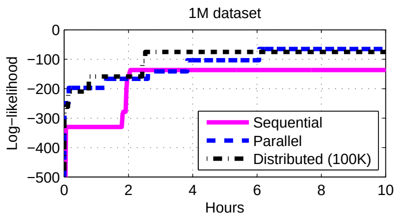
[6]http://mesos.apache.org

[7]http://zookeeper.apache.org

[8]Spark: Lightning-fast cluster computing. See http://spark.apache.org

[9]Scientific computing machine learning and natural language processing: http://www.scalanlp.org

to handle 5M observations due to memory limitations. We can also observe that: i) the distributed implementation behaves similarly to the sequential implementation for 100K because it considers chunks of 100K observations and therefore uses a unique core; ii) for 1M observations the parallel (16 cores) and the distributed (10 cores) are comparable, being the latter slightly slower; and iii) for the 5M dataset the distributed implementation (50 cores) presents a speed-up approximately linear with the number of cores (the deviations from linear speed-up are due to the current model complexity, i.e., number of clusters, at each iteration). Finally, we remark that the distributed implementation allows one to deal with 50M observations, with 1.3 minutes per iteration.



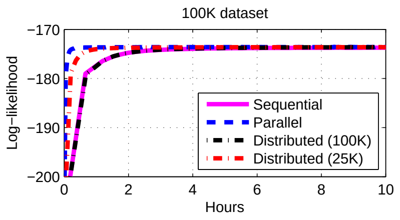
(a) Log-likelihood per observation.

Algorithm \ N	100K	1M	5M	50M
Sequential	0.1349	1.3963	-	-
Parallel	0.0123	0.1397	0.8736	-
Distributed (100K)	0.1795	0.1512	0.2143	1.3429

(b) Time (in minutes) per iteration

Figure 4: Experiments over synthetic data.

Real data. We use a subset of the Tiny Images database.[10] In particular, we consider a grayscale version of the images and apply a randomized approximation to principal components analysis (PCA), based on $100,000$ images and thresholding the top 256 principal components. The dimension of our data is thus 256, and consider two subsets: 100K and 1M of images, assuming fixed noise variance of 0.5. We initialize the three algorithms with 1000 clusters. Figure 5 shows the results. Specifically, Figure 5a shows the log-likelihood evolution with time of the sequential, parallel and distributed (with chunk sizes of 25K and 100K) implementations for a subset of $100,000$ images. As expected, the sequential and distributed implementation with chunks of 100K present similar convergence rates, while decreasing the chunk size to 25K (4 cores) makes the distributed approach achieve a rate closer to the parallel one. Figure 5b shows the average time per iteration for the four implementations and for both datasets. We remark that all the algorithms present linear complexity with the dimensionality of the observations and with the number of components K_+, which explains the differences between the real and synthetic datasets. As before, the parallel implementation is the optimal when dealing with 100K images, while for the 1M dataset, the distributed algorithm with chunk size of 25K (40 cores) is able to run an iteration every 10 minutes. The one-to-three performance degradation between 100K and 1M is due to the fact that the inferred number of clusters for the latter is three times the number of clusters for the former, which is the expected behavior for BNP models, which adapt the model complexity to the number of observations.



(a) Log-likelihood per observation.

Algorithm \ N	100K	1M
Sequential	9.9169	-
Parallel	0.6791	26.4195
Distributed (100K)	10.7375	32.8588
Distributed (25K)	3.1215	9.6388

(b) Time (in minutes) per iteration

Figure 5: Experiments over Tiny Images dataset.

6 Conclusions

This paper presents a general approach to perform parallel inference based on an uncollapsed Gibbs sampling variant. This approach is scalable and general for any probabilistic model with an exponential family likelihood function and exchangeable observations. We propose a general Map/Reduce framework for immediate deployment to the desired model, as well as a detailed description of the implementation. Simulations with both synthetic and real data —with up to 50 and 1 million observations, respectively— support the scalability and efficiency of our algorithms.

[10]This data is publicly available at http://horatio.cs.nyu.edu/mit/tiny/data

References

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[2] A. Asuncion, P. Smyth, and M. Welling. Asynchronous Distributed Learning of Topic Models. In *Advances in Neural Information Processing Systems 21 (NIPS'08)*, pages 81–88, 2008.

[3] S. Bratières, J. V. Gael, A. Vlachos, and Z. Ghahramani. Scaling the iHMM: Parallelization versus Hadoop. In *CIT*, pages 1235–1240. IEEE Computer Society, 2010.

[4] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley, New York, NY, USA, 1991.

[5] B. de Finetti. La prévision : ses lois logiques, ses sources subjectives. *Annales de l'institut Henri Poincaré*, 7(1):1–68, 1937.

[6] P. Diaconis and D. Ylvisaker. Conjugate priors for exponential families. *The Annals of Statistics*, 7(2):269–281, March 1979.

[7] F. Doshi-Velez and Z. Ghahramani. Accelerated sampling for the Indian buffet process. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 273–280, New York, NY, USA, 2009. ACM.

[8] F. Doshi-Velez, S. Mohamed, Z. Ghahramani, and D. A. Knowles. Large scale nonparametric bayesian inference: Data parallelisation in the Indian buffet process. In *Advances in Neural Information Processing Systems 22 (NIPS)*, pages 1294–1302, December 2009.

[9] J. Gonzalez, Y. Low, A. Gretton, and C. Guestrin. Parallel Gibbs sampling: From colored fields to thin junction trees. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, May 2011.

[10] T. L. Griffiths and Z. Ghahramani. The Indian buffet process: an introduction and review. *Journal of Machine Learning Research*, 12:1185–1224, 2011.

[11] T. J. Hansen, M. Mørup, and L. K. Hansen. Large scale GPU based inference for the infinite relational model.

[12] H. Ishwaran and L. F. James. Gibbs sampling methods for stick-breaking priors. *Journal of the American Statistical Association*, 96(453):161–173, 2001.

[13] H. Ishwaran and M. Zarepour. Exact and approximate sum representations for the Dirichlet process. *The Canadian Journal of Statistics / La Revue Canadienne de Statistique*, 30(2):269–283, 2002.

[14] M. I. Jordan. Hierarchical models, nested models, and completely random measures. In M.-H. Chen, D. Dey, P. Muller, D. Sun, and K. Ye, editors, *Frontiers of Statistical Decision Making and Bayesian Analysis: In Honor of James O. Berger*, pages 207–217. Springer, New York, NY, 2010.

[15] M. I. Jordan. The era of big data. *ISBA Bulletin*, 18(2):1–3, 2011.

[16] M. I. Jordan. What are the open problems in Bayesian statistics? *ISBA Bulletin*, 18(1):1–4, 2011.

[17] J. Lee and Y. Kim. A new algorithm to generate beta processes. *Computational Statistics & Data Analysis*, 47(3):441–453, 2004.

[18] D. Lovell, J. Malmaud, R. P. Adams, and V. K. Mansinghka. Clustercluster: Parallel Markov chain Monte Carlo for Dirichlet process mixtures. *arXiv preprint arXiv:1304.2302*, 2013.

[19] R. M. Neal. Markov chain sampling methods for Dirichlet process mixture models. *Journal of Computational and Graphical Statistics*, 9(2):249–265, 2000.

[20] D. Newman, P. Smyth, M. Welling, and A. U. Asuncion. Distributed inference for latent dirichlet allocation. In J.C. Platt, D. Koller, Y. Singer, and S.T. Roweis, editors, *Advances in Neural Information Processing Systems 20 (NIPS'07)*, volume 20, pages 1081–1088, 2007.

[21] J. W. Paisley, D. M. Blei, and M. I. Jordan. Variational bayesian inference with stochastic search. In *International Conference on Machine Learning 29. (ICML'12)*. icml.cc / Omnipress, 2012.

[22] Y. W. Teh, D. Görür, and Z. Ghahramani. Stick-breaking construction for the Indian buffet process. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, volume 11, 2007.

[23] Y. W. Teh, M. J. Jordan, M. J. Beal, and D. M. Blei. Hierarchical Dirichlet processes. *Journal of the American Statistical Association*, 101(476):1566–1581, 2006.

[24] R. Thibaux and M. I. Jordan. Hierarchical beta processes and the Indian buffet process. In Marina Meila and Xiaotong Shen, editors, *AISTATS*, volume 2 of *JMLR Proceedings*, pages 564–571, 2007.

[25] S. Williamson, A. Dubey, and E. P. Xing. Parallel Markov chain Monte Carlo for nonparametric mixture models. In *International Conference on Machine Learning (ICML)*, volume 28 of *JMLR Proceedings*, pages 98–106, 2013.

[26] F. Yan, N. Xu, and Y. Qi. Parallel inference for latent dirichlet allocation on graphics processing units. In *Advances in Neural Information Processing Systems 22 (NIPS'09)*, pages 2134–2142, December 2009.

[27] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.