2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)

Distributed MCMC Inference in Dirichlet Process Mixture Models Using Julia

Or Dinari*
Department of Computer Science
Ben-Gurion University
Beer-Sheva, Israel
dinari@post.bgu.ac.il

Angel Yu*
CSAIL
MIT
Cambridge MA, USA
angelyu@alum.mit.edu

Oren Freifeld
Department of Computer Science
Ben-Gurion University
Beer-Sheva, Israel
orenfr@cs.bgu.ac.il

John W. Fisher III
CSAIL
MIT
Cambridge MA, USA
fisher@csail.mit.edu

***Abstract*—Due to the increasing availability of large data sets, the need for general-purpose massively-parallel analysis tools become ever greater. In unsupervised learning, Bayesian nonparametric mixture models, exemplified by the Dirichlet-Process Mixture Model (DPMM), provide a principled Bayesian approach to adapt model complexity to the data. Despite their potential, however, DPMMs have yet to become a popular tool. This is partly due to the lack of friendly software tools that can handle large datasets efficiently. Here we show how, using Julia, one can achieve efficient and easily-modifiable implementation of distributed inference in DPMMs. Particularly, we show how a recent parallel MCMC inference algorithm – originally implemented in C++ for a single multi-core machine – can be distributed efficiently across multiple multi-core machines using a distributed-memory model. This leads to speedups, alleviates memory and storage limitations, and lets us learn DPMMs from significantly larger datasets and of higher dimensionality. It also turned out that even on a single machine the proposed Julia implementation handles higher dimensions more gracefully (at least for Gaussians) than the original C++ implementation. Finally, we use the proposed implementation to learn a model of image patches and apply the learned model for image denoising. While we speculate that a highly-optimized distributed implementation in, say, C++ could have been faster than the proposed implementation in Julia, from our perspective as machine-learning researchers (as opposed to HPC researchers), the latter also offers a practical and monetary value due to the ease of development and abstraction level. Our code is publicly available at https://github.com/dinarior/dpmm_subclusters.jl**

I. Introduction

In unsupervised learning, Bayesian nonparametric mixture models, exemplified by the Dirichlet-Process Mixture Model (DPMM), provide a principled approach for Bayesian modeling while adapting the model complexity to the data. This contrasts with finite mixture models whose complexity is determined manually or via model-selection methods. A DPMM example is the Dirichlet-Process Gaussian Mixture Model (DP-GMM), an ∞-dimensional extension of the Bayesian variant of the (finite) Gaussian Mixture Model (GMM). Despite their potential, however, and although many researchers have used them successfully in many applications over the last decade, DPMMs have not enjoyed wide popularity among practitioners, largely due to computational bottlenecks in current implementations. We argue that useful implementations must be able to leverage parallel and distributed computing resources in order for DPMMs to be a practical choice for analysis. One of the missing pieces, we argue, is the availability of friendly software tools that can efficiently handle DPMM inference in large datasets; *i.e.*, we need to be able to leverage parallel- and distributed-computing resources for DPMM inference. This is because not only potential speedups but also memory and storage requirements. This is especially true for distributed mobile robotic sensing applications where robotics agents have limited computational and communication resources. In an analogy, consider how advances in GPU computing contributed to the success of deep learning (at least in supervised learning). That said, here we are more interested in distributing computations across multiple CPU cores and multiple machines. This is not only because not all computation types are appropriate for GPU hardware or that CPU resources are more available but also because it is useful during algorithm development to easily distribute computations as an abstraction.

While DPMMs are theoretically ideal for handling large, unlabeled, datasets, current *implementations* of DPMM inference do not scale well with increases in size of the data sets and/or dimensionality (exceptions are few recent implementations designed for streaming, typically low-dimensional, data). This

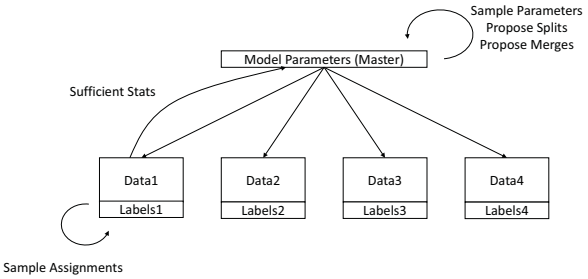

Fig. 1: The architecture of the proposed multi-machine multi-core implementation. In this figure, for concreteness, we show parallelization over 4 machines.

*indicates that both these authors contributed equally. This work was partially supported by NSF award 1622501 and by the Lynn and William Frankel Center for Computer Science at BGU. Or Dinari was partially supported by Trax.

978-1-7281-0912-1/19/$31.00 ©2019 IEEE
DOI 10.1109/CCGRID.2019.00066

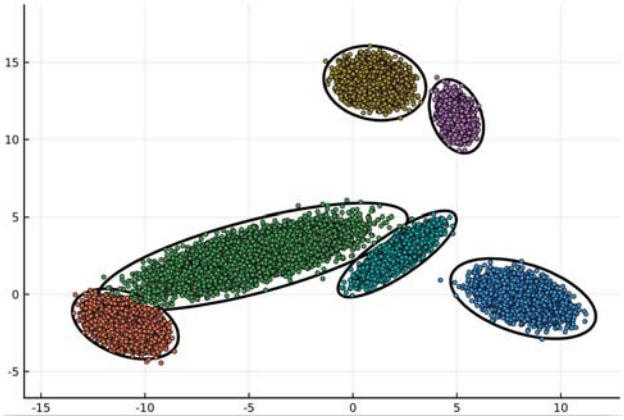

Fig. 2: 2D Synthetic data. 10^6 points drawn from a 6-component GMM. The correctly-inferred clusters are shown, in different colors, as ellipses at 5 standard deviations (the correct value of $K = 6$ was inferred and was not assumed known.

is partly since most existing implementations are serial; *i.e.*, they do not harness the power of distributed computing. This does not mean that there do not exist distributed *algorithms* for DPMM inference. There is, however, a large practical gap between designing such an algorithm and having it implemented efficiently in a way that fully utilizes all the available computing resources. *Our work is an attempt to close this gap in a way which is useful for typical large-scale applications (such as in,* e.g.*, computer vision).*

More generally than the topic of DPMMs, distributed implementations, at least in traditional programming languages used for such implementations, tend to be hard to debug, read, and modify. This clashes with the usual workflow of algorithm development. Julia, a recent high-level/high-performance programming language for technical computing, substantially addresses this issue. It is "easy to write (and read)", having syntax similar to other technical computing environments (*e.g.*, MATLAB or Python's NumPy). It provides a sophisticated compiler, distributed parallel execution, numerical accuracy, and an extensive mathematical function.

In this work we show how to implement, using Julia, efficient distributed DPMM inference. Particularly, we demonstrate how a recent parallel MCMC inference algorithm [5] – originally implemented in C++ for a single multi-core machine in a highly-specialized fashion using a shared-memory model — can be implemented such that it is distributed efficiently across multiple multi-core machines using a distributed-memory model. This leads to speedups, alleviates memory and storage limitations of a single machine, and lets us learn models from significantly larger datasets. We describe our implementation design and compare our implementation with the one from [5]. Surprisingly, we found that, in the DP-GMM case, even on a single machine our implementation handles higher dimensions better, and not just in terms of speed, than [5]. This is important as computer vision usually deals with higher-dimensional data than standard machine-learning problems. Finally, we use our implementation to learn image-patch models and apply them to image denoising. We emphasize that the goal of this paper is not to claim that Julia is faster/better than language X. For example, we speculate that a highly-optimized distributed implementation in, say, C++ could have been faster than the proposed implementation in Julia. However, from our perspective as machine-learning researchers (as opposed to HPC researchers), distributed implementations in Julia, ours included, also offer a practical and monetary value due to the ease of development and abstraction level. That is, even if other implementations might be faster, and even though the monetary cost of several machines is of course greater than a single machine, the rapid development and ease of modifications to the model in Julia offer significant value in terms of our time as machine-learning algorithm developers while still being able to massively parallelize computations.

II. Related Work

Bayesian non-parameteric mixture models in general, and DPMMs in particular, have been used in computer vision by several authors, *e.g.* [4], [6], [8]–[10], [13], [16]–[19], [21], [22], [24], [25], [27], [28], [30], [32]–[35], [37]. While this list may seem long, note it represents almost an entire decade, yet it is almost exhaustive.

A few DPMM-inference software packages, in several different languages (e.g., R [15], Python [12], and Matlab [20]), are publicly available; however, their implementations are serial. A few works use GPU to distribute DPMM inference in low-dimensional streaming-data applications using, *e.g.*, either small-variance asymptotics [30] or variational methods [3]. Our work differs from those in that we use a multi-machine, multi-core, pure-CPU distributed implementation, that we rely on MCMC inference, and in that we can handle large, high-dimensional data in batch mode.

Julia has been little explored for Bayesian nonparametric mixture models, though there are counterexamples, *e.g.* [36]. Likewise, Lin [23] use Julia for inference in Bayesian nonparametric mixture models. In both [36] and [23] the implementations are serial. While there exist few more such examples, it seems that the strong (and remarkably painless) support Julia offers for parallel and distributed computing has yet to be utilized in this context.

In this work we use Julia to implement a parallel MCMC sampler for DPMM proposed by Chang and Fisher [5] for both Gaussian and Multinomial distributions. Our code can be easily adapted to other component distributions, *e.g.*, Poisson, as long as they belong to the Exponential family [2]. The algorithm from [5], which is based on the split-merge framework [14], was designed as a parallel algorithm and its authors released a multi-core single-machine C++ implementation. The latter assumes a shared-memory model while we extend their work by proposing an implementation that accommodates *a distributed-memory model via an explicit partitioning of*

Cores × Machines	C++ [5]	Julia
1×1	55.87	132.88
2×1	35.48	78.28
4×1	16.45	42.48
8×1	**10.21**	32.95
8×2	–	17.56
8×3	–	16.73
8×4	–	12.93

TABLE I: Time (in [sec]) for running 100 DP-GMM iterations with $d = 2, N = 10^6, K = 6$.

Cores × Machines	C++ [5]	Julia
1×1	1637.52	416.40
2×1	720.29	232.62
4×1	480.50	139.86
8×1	262.41	94.64
8×2	–	53.01
8×3	–	39.30
8×4	–	**35.68**

TABLE II: Time (in [sec]) for running 100 DP-GMM iterations of $d = 30, N = 10^6, K = 6$.

the data. This enables our implementation to extend theirs by allowing parallelization across multiple machines, not just multiple cores. In turn, this enables us to not only leverage more computing power and gain speedups but also provide practical benefits in terms of memory and storage. For example, our implementation can be used within a distributed network of weak agents (*e.g.*, small robots collecting data). It also never transfers data (which is expensive and slow); rather, we transfer only sufficient statistics and parameters. Thus, the proposed implementation is also well suited for a network of low-bandwidth communication.

Surprisingly, it turns out that even on a single machine, our Julia implementation of DP-GMM handles high dimensions better than that C++ implementation, and not just in terms of speed; *e.g.*, for 250-dimensional data, [5] usually fails (while ours succeeds). Thus, *e.g.*, it is inapplicable for learning, say, a prior of large image patches. In contrast, we easily apply our implementation for learning a DP-GMM over such patches. This is akin to Zoran and Weiss [38], except that they learn a finite (and non-Bayesian) GMM where they manually fix K, the number of Gaussians. Their implementation was also serial and slow (based on the timings they report). In contrast, we automatically infer, in a Bayesian fashion, all the parameters (including K) from the data. Similarly to the experiments presented here, Hughes and Sudderth [11] use a DP-GMM in this problem. While they used a serial implementation of variational inference we used a distributed (and exact) MCMC sampling.

An application suggested in [38] for image-patch models was image denoising. As it was impractical for them to use the entire training dataset (∼44 million patches), they trained their GMM on only a smaller subset (2 millions). Hughes and Sudderth [11] did the same, probably for similar reasons (and report, for their best run, results comparable to [38]). When trained on 2 million patches, our results are comparable to [38] but we obtained them much faster, despite the fact that unlike [38] we also infer K and use a sampling-based inference in a fully Bayesian setting. When trained on a much larger training dataset (∼44 millions), we get slightly better results than them. The fact that more data improves results is unsurprising. Our point, however, is that: 1) the fact we are able to do it at all is due to the distributed nature of our implementation; 2) the fact we use a Bayesian nonparametric model, *i.e.* an infinite-dimensional DP-GMM as opposed to their finite GMM, lets us automatically infer K and adapt it to the data. E.g., for the larger dataset we inferred a larger K than the one we inferred for the smaller one. It means we let the data determine the complexity. This kind of argument is standard in Bayesian nonparametric modeling; our key contribution is to show how to convert this theory to practice, *i.e.*, implement such inference in a distributed way so one can process large, high-dimensional data. Our code, to be made available upon publication of this paper, is also easy to read and adjust, due to the fact that Julia is a high-level language.

III. Background

This section provides an overview of: 1) DPMMs; 2) the inference algorithm from [5] that uses parallel sampling via subcluster splits and merges; 3) Zoran and Weiss' GMMs of image patches and their application to image denoising.

A. DPMMs

A DPMM is an ∞-dimensional Bayesian mixture model with a Dirichlet-Process (DP) prior over the clusters parameters. For the original proof of the existence of Dirichlet Processes, see [7]. For a more detailed discussion on Dirichlet Processes with computer-vision and machine-learning readers in mind, see [31]. The generative model of a DPMM, for data points $(x_i)_{i=1}^N$, is as follows:

$$G \sim DP(\alpha, H), \tag{1}$$

$$\tilde{\theta}_i \sim G(\tilde{\theta}_i), \qquad \forall i \in \{1, 2, \ldots, N\}, \tag{2}$$

$$x_i \sim f_x(x_i; \tilde{\theta}_i), \quad \forall i \in \{1, 2, \ldots, N\}. \tag{3}$$

Here, N is the number of data points, α is the concentration parameter, and H is the base measure for the DP. For each data point, indexed by i, we draw a sample $\tilde{\theta}_i$ from the DP realization G. We then draw x_i from the distribution parameterized by $\tilde{\theta}_i$. Since G is a DP realization, it is a distribution over discrete atoms. Hence, multiple data points can share the same parameter $\tilde{\theta}_i$. Points which share the same parameter form a cluster and the shared $\tilde{\theta}$ are referred to as cluster parameters. *An equivalent representation, but perhaps easier to understand*, is based on a stick-breaking process. This alternative generative model is as follows:

$$\pi \sim \text{GEM}(1, \alpha), \tag{4}$$

$$\theta_k \sim f_\theta(\theta_k; \lambda) = H, \quad \forall k \in \{1, 2, \ldots\}, \tag{5}$$

$$z_i \sim \text{Cat}(\pi), \qquad \forall i \in \{1, 2, \ldots, N\}, \tag{6}$$

$$x_i \sim f_x(x_i; \theta_{z_i}), \qquad \forall i \in \{1, 2, \ldots, N\}. \tag{7}$$

Cores × Machines	C++ [5]	Julia
1× 1	94.53	350.84
2× 1	54.49	155.98
4× 1	34.08	104.95
8× 1	**16.69**	66.46
8× 2	–	34.51
8× 3	–	25.32
8× 4	–	21.28

TABLE III: Time (in [sec]) for running 100 DP-MNMM iterations with $d = 100, N = 10^6, K = 6$.

Here, GEM stands for the Griffiths-Engen-McCloskey stick-breaking process for a DP. By drawing a sample from GEM$(1, \alpha)$, we get an ∞-dimensional vector of weights π. For each of the weights π_k, we draw a sample from the base measure H as our parameter for cluster k. For each data point i, we first draw a sample for the label assignment z_i using the weights π. Finally, we sample x_i from the distribution of cluster z_i which is parameterized by θ_{z_i}.

B. Parallel Sampler for DPMMs

Chang and Fisher [5] proposed a parallel sampling algorithm for DPMMs. The main idea there is to have a restricted Gibbs sampler [26] which fixes the number of clusters and then proposes splits and merges. This allows a Gibbs sampling step to be done in parallel without having to worry about the creation of new clusters. In order to propose meaningful splits that are likely to be accepted, their algorithm uses auxiliary variables such that each cluster consists of 2 sub-clusters. These variables are $\bar{z}_i \in \{l, r\}$ (conceptually thought of as "left" and "right") which indicate which of the sub-cluster the i^{th} point is assigned to and $\bar{\pi}_k = \{\bar{\pi}_{kl}, \bar{\pi}_{kr}\}, \bar{\theta}_k = \{\bar{\theta}_{kl}, \bar{\theta}_{kr}\}$ which denote the weights and parameters of the sub-clusters of cluster k. By sampling the sub-clusters, one is able to propose meaningful splits that split a cluster into its 2 sub-clusters. Merges are proposed by merging 2 sub-clusters into one with each of the original clusters becoming a sub-cluster of the new merged cluster. The algorithm is summarized in Algorithm 1. We use the same notation as in [5] where N is the number of data points, K is the number of clusters, $f_x(X; \theta)$ is the likelihood of a set of data under the parameter θ, $f_\theta(\theta; \lambda)$ is the likelihood of the parameter θ under the prior λ, $\overset{\propto}{\sim}$ denotes sampling proportional to the right-hand side of the equation, and $\mathcal{I}_k = \{i : z_i = k\}$. As that algorithm was designed with parallelization in mind, most of its operations are parallelizable. Particularly, sampling cluster parameters is parallelizable over the clusters, sampling assignments can be computed independently for each point, and cluster splits can be proposed in parallel. Thus, a multi-core implementation is quite straightforward, with the caveat that one needs to be careful with merges; *i.e.*, to prevent more than 2 clusters merge into the same single cluster. *E.g.*, if clusters 1 and 2 are merged at the same time that clusters 2 and 3 are merged this would imply the three clusters (1,2, and 3) would be merged into a single one. See [5] for more details.

Algorithm 1 An inference iteration with sub-cluster splits

Run an iteration of restricted Gibbs sampling:

(a) Sample cluster weights $\pi_1, \pi_2, \ldots, \pi_K$,:

$$(\pi_1, \ldots, \pi_K, \tilde{\pi}_{K+1}) \sim \text{Dir}(N_1, \ldots, N_K, \alpha). \quad (8)$$

(b) Sample sub-cluster weights $\bar{\pi}_{kl}, \bar{\pi}_{kr}$ for each cluster $k \in \{1, \ldots, K\}$:

$$(\bar{\pi}_{kl}, \bar{\pi}_{kr}) \sim \text{Dir}(N_{kl} + \alpha/2, N_{kr} + \alpha/2). \quad (9)$$

(c) Sample cluster parameters θ_k for each cluster k:

$$\theta_k \overset{\propto}{\sim} f_x(x_{\mathcal{I}_k}; \theta_k) f_\theta(\theta_k; \lambda) \quad (10)$$

(d) Sample sub-cluster parameters $\bar{\theta}_{kh}$ for each cluster $k \in \{1, \ldots, K\}$ and $h \in \{l, r\}$:

$$\bar{\theta}_{kh} \overset{\propto}{\sim} f_x(x_{\mathcal{I}_{kh}}; \bar{\theta}_{kh}) f_\theta(\bar{\theta}_{kh}; \lambda). \quad (11)$$

(e) Sample cluster assignments z_i for each point $i \in \{1, \ldots, N\}$:

$$z_i \overset{\propto}{\sim} \sum_{k=1}^{K} \pi_k f_x(x_i; \theta_k) \mathbb{1}(z_i = k). \quad (12)$$

(f) Sample sub-cluster assignments $\bar{z}_i$ for each point $i \in \{1, \ldots, N\}$:

$$\bar{z}_i \overset{\propto}{\sim} \sum\nolimits_{h \in \{l,r\}} \pi_{z_i h} f_x(x_i; \bar{\theta}_{z_i h}) \mathbb{1}(\bar{z}_i = h). \quad (13)$$

Propose and Accept Splits:

(a) Propose to split cluster k into its 2 sub-clusters for all $k \in \{1, 2, \ldots, K\}$.

(b) Calculate the Hastings ratio H and accept the split with probability $\min(1, H)$:

$$H_{\text{split}} = \frac{\alpha \Gamma(N_{kl}) f_x(x_{\mathcal{I}_{kl}}; \lambda) \cdot \Gamma(N_{kr}) f_x(x_{\mathcal{I}_{kr}}; \lambda)}{\Gamma(N_k) f_x(x_{\mathcal{I}_k}; \lambda)} \quad (14)$$

Propose and Accept Merges:

(a) Propose to merge clusters k_1, k_2 for all pairs $k_1, k_2 \in \{1, 2, \ldots, K\}$.

(b) Calculate the Hastings ratio H and accept the merge with probability $\min(1, H)$.

$$H_{\text{merge}} = \frac{\Gamma(N_{k_1} + N_{k_2})}{\alpha \Gamma(N_{k_1}) \Gamma(N_{k_2})} \frac{p(x|\hat{z})}{p(x|z)} \times \frac{\Gamma(\alpha)}{\Gamma(\alpha + N_{k_1} + N_{k_2})} \quad (15)$$

$$\times \frac{\Gamma(\frac{\alpha}{2} + N_{k_1}) \Gamma(\frac{\alpha}{2} + N_{k_2})}{\Gamma(\frac{\alpha}{2}) \Gamma(\frac{\alpha}{2})} \quad (16)$$

C. GMMs and Image Denoising

Zoran and Weiss [38] proposed an image-denoising method that is based on maximizing the Expected Patch Log Likelihood (EPLL) under a (learned) model while staying close to the observed noisy image. See [38] for more details. For our purposes, it is sufficient to summarize as follows: one first needs to somehow learn a statistical model of images patches (in [38], it was a finite GMM of $K = 200$ components, learned via Expectation Maximization); next, any such learned model can be used within their denoising method whose MATLAB

Cores × Machines	8×1	8×2	8×3	8×4
C++ [5]	79.57	-	-	-
Julia	234.09	125.51	86.87	**67.10**

TABLE IV: Time (in [sec]) for running 100 DP-MNMM iterations with $d = 100, N = 10^6, K = 60$.

code, together with their learned GMM, are publicly available. Thus, their framework provides a way to test and showcase the utility of our method.

IV. The Proposed Implementation

Chang and Fisher [5] released a multi-core, single-machine, C++ implementation of their algorithm for the case when each cluster is a Gaussian, *i.e.*, a DP-GMM, and the case when each cluster is Multinomial, *i.e.*, a Dirichlet-Process Multinomial Mixture Model (DP-MNMM). Their implementation relies on a shared-memory model and is also highly-optimized and not easily adjustable. In contrast, we extend their work by proposing a flexible multi-core muli-machine implementation that relies on a distributed-memory model and is written in Julia. Similar to their implementation [5], and for the interest of comparison, we focused on the Gaussian and Multinomial cases. As an aside, we remark that one aspect of the flexibility of our implementation is the ease in which the Gaussian-to-Multinomial modification was made, requiring far less code changes than was the case in Chang and Fisher's implementation. We now discuss key implementation details and challenges. Assume we have N data points, K clusters and M machines where each machine has P cores.

A. High-Level Design/Architecture

To achieve efficient multi-machine parallelization in Julia, we use `DistributedArray` objects to store the data, the labels and the sub-cluster labels. We make an extensive use of *sufficient statistics* [29], including for inter-core and inter-machine communication. The sufficient statistics for a Gaussian cluster with points $x_1, x_2, \ldots, x_m \in \mathbb{R}^d$ are:

$$T^{G_1} = \sum_{i=1}^{m} x_i \in \mathbb{R}^d\,; \quad T^{G_2} = \sum_{i=1}^{m} x_i x_i^T \in \mathbb{R}^{d\times d}\,. \tag{17}$$

The sufficient statistic for a Multinomial cluster with points $x_1, x_2, \ldots, x_m \in \mathbb{N}_0^d$ is:

$$T^{M} = \sum_{i=1}^{m} x_i \in \mathbb{N}_0^d\,. \tag{18}$$

In both cases d is the dimensionality of the points. Aggregating sufficient statistics for each cluster enables sampling the cluster parameters. Since each sufficient statistic is a sum over a set of data points, we can collect the sums for each process and aggregate them. In fact, this property holds true for all distributions in the exponential family. For an introduction to sufficient statistics in general, and in the exponential family in particular, see [29] and [2].

Sampling the cluster and sub-cluster parameters takes $O(K)$ time given the sufficient statistics, so we parallelize them only on the master machine as usually $K \ll N$. Sampling the cluster and sub-cluster assignments takes $O(NK)$ time as can be seen from Eqs. (12) and (13). These are the only steps that scale with both N and K, so parallelizing these steps is key. Since the assignment of each data point is sampled independently from the rest, we distribute the data points evenly across every process in every machine along with their cluster assignments. We maintain the cluster and sub-cluster parameters in the master machine along with their sufficient statistics. They are then broadcasted to all processes so we can sample the assignments. Proposing splits takes $O(K)$ time and proposing merges takes $O(K^2)$ time which does not scale with N so they are relatively fast and are thus done on the master machine. This architecture is illustrated in Figure 1.

Model	K	Avg. PSNR
GMM [38]	200 (fixed)	28.5061
DP-GMM ($\alpha = 1$)	72	28.4173
DP-GMM ($\alpha = 10^3$)	80	28.4276
DP-GMM ($\alpha = 10^6$)	99	28.4303
DP-GMM ($\alpha = 10^6$, all patches)	330	**28.5402**

TABLE V: Ave. PSNR (the higher the better) on 50 test images (with added Gaussian noise) for different models.

B. Detailed Implementation Design

We maintain the following objects on the master machine: 1) an `Array` of the local clusters where each item in that vector holds the cluster and sub-cluster parameters (*i.e.*, θ_k, μ_k, Σ_k,θ_{kl}, θ_{kr}, $\bar{\pi}_{kl}$ and $\bar{\pi}_{kr}$ for the k^{th} Gaussian), the point counts (*i.e.*, N_k, N_{kl}, N_{kr}) and the sufficient statistic (*i.e.*, T_k, T_{kl}, T_{kr}) as well as the cluster weights (*i.e.*, $\pi_1, \pi_2, ..\pi_k$). We have the following objects on each worker process: cluster and sub-cluster weights (broadcasted from the master); a thin version of the cluster and sub-cluster parameters which includes only the distributions (broadcasted from master); chunks of the data, x_i (as part of the `DistributedArray`); chunks of the cluster and sub-cluster assignments, $z_i, \bar{z}_i$ (as part of a `DistributedArray`). We now describe our implementation of Algorithm 1:

1) Run an iteration of a restricted Gibbs sampling:

a. Sample cluster parameters (Eq. (10)) and sub-cluster parameters (Eq. (11)) on the worker processes on the master machine in parallel.
b. Sample cluster weights (Eq. (8)) and sub-cluster weights (Eq. (9)) on the master process.
c. Broadcast cluster and sub-cluster weights and parameters to all worker processes.
d. Sample cluster assignments (Eq. (12)) in parallel across all worker processes. Each worker samples the assignments of the data it is in charge of, updating the assignments in the `DistributedArray`.
e. Sample sub-cluster assignments (Eq. (13)) in parallel across all worker processes. Each worker samples the assignments of the data it is in charge of, updating the assignments in the `DistributedArray`.

f. Update the cluster and sub-cluster sufficient statistics in parallel across all worker processes. Each worker calculates the sufficient statistics from the data it is in charge of. We then aggregate the sufficient statistics across all workers by simply summing them. *E.g.*, if we have 4 workers and we want to calculate the sufficient statistics T_1 (*i.e.*, the sufficient statistics of for cluster 1), we would first calculate $\{T_1^i\}_{i=1}^4$, the sufficient statistics for the 4 workers, using the data points each worker is in charge of that are assigned to cluster 1 and then sum :

$$T_1 = T_1^1 + T_1^2 + T_1^3 + T_1^4 \tag{19}$$

Note that we do not maintain the sufficient statistics received from each worker as this is a synchronous implementation. If we were doing this asynchronously, we would maintain these as this would have allowed us to undo the previous contribution to the aggregated statistics.

2) Splits:

a. Propose to split each cluster in parallel on the master machine. The calculation of the Hastings ratio (Eq. (14)) requires only the sufficient statistics which are available via the local clusters' `Array` on the master.

b. Process all accepted splits by creating an assignment map which maps old assignments to new ones. This is then broadcasted to each worker process, which in turn updates the assignments of the points it is in charge of. Sufficient statistics for the newly-split clusters are calculated and updated (similarly to Eq. (19)).

3) Merges:

a. Propose to merge all pairs of clusters in serial on the master process (Eq. (16)). As in [5], when a merge of 2 clusters is accepted, we do not consider them for future merges in this iteration.

b. Process all accepted merges by creating an assignment mapping which maps old assignments to new ones. This is then broadcasted to all worker processes and each of which updates the assignments of the points it is in charge of. Sufficient statistics for the newly merged clusters are calculated and updated (similarly to Eq. (19)).

C. Optimizing Process Communication

In each iteration, we broadcast a number of objects from the master process to all processes as well as the sufficient statistics from the workers so they can be aggregated. A naive way to do this would be sending the objects from the master process to each worker and the sufficient statistics from the workers to the master directly. Unsurprisingly, however, we found that inter-machine communication (between processes on the same machine) is much faster than intra-machine communication (between processes on different machines). To reduce this communication overhead, we first send the objects to be broadcasted from the master to one process in each machine and then using that process to send it to all processes in its machine. For sufficient statistics that need to be aggregated, we first aggregate them within each machine and then send those aggregated sufficient statistics to the master for final aggregation. Empirically, *this reduced communication overhead tenfold.*

D. Runtime Complexity

In § IV-A, we touched upon the runtime complexity. We now analyze the total runtime complexity.

Run an iteration of a restricted Gibbs sampling. Sampling the cluster and sub-cluster parameters (including weights) takes constant time for each cluster but is parallelized over P processes on the master. This takes $O(K/P)$ time. Sampling the cluster weights is also done in constant time. Broadcasting cluster and sub-cluster weights and parameters to all worker processes take $O(M+P)$ as we first broadcast to each machine and then, from each machine, we broadcast to every process. Sampling the cluster assignments takes $O(NK)$ time in serial, but this is parallelized over MP processes so this takes $O(NK/(MP))$. Sampling the sub-cluster assignments takes $O(N)$ time in serial, but this is parallelized over MP processes so this takes $O(N/(MP))$. Updating the cluster and sub-cluster sufficient statistics can be split up into 2 steps. The first step is to for all workers to calculate the sufficient statistics for the data it is in charge of. This step takes $O(N/(MP))$ time. The second step is to aggregate across all workers. This step takes $O(M+P)$ time. Overall, this take $O(N/(MP)) + O(M+P)$ time.

Splits: Proposing splits by looking at each cluster is $O(K)$. Processing all the accepted splits requires updating the sufficient statistics which could take at the worst case $O(N/(MP)) + O(M+P)$ if all clusters are split.

Merges: Proposing merges by inspecting each cluster pair is $O(K^2)$. Processing all the accepted merges also requires updating sufficient statistics. The worst case (*i.e.*, if all clusters are merged) is thus $O(N/(MP)) + O(M+P)$.

To summarize, the total runtime complexity is $O(K) + O(M+P) + O(NK/(MP))$. Since $N \gg K, P, M$, this implementation achieves linear parallelization theoretically.

E. Memory Complexity

We now look at the amount of memory used on each machine. The data is stored as a distributed array across all processes, so we have $O(D * N/M)$ on each machine. Additionally, we have labels and sub-labels distributed arrays across all processes, adding $O(2N/M)$ on each machine. Each process also has a copy of the cluster and sub-cluster parameters which takes $O(KP)$ space for each machine. We also have to aggregate sufficient statistics for each cluster after sampling the assignments. This also takes $O(KP)$ for each machine. Hence, we have a total memory usage of $O(N/M + KP)$. Since usually $N \gg K, P, M$, the memory overhead is insignificant in comparison to the data itself.

V. Results

The experiments below were done on Dell machines with Intel I7-6800K CPU @ 3.40GHz processors, treating each hyperthread as a core.

Synthetic Data: DP-GMM. We started by testing the proposed implementation on large synthetic datasets. We generated synthetic data by drawing 10^6 points from an underlying GMM with 6 clusters; see Fig. 2. For illustration, we use 2D points here but note that the proposed implementation works in arbitrary dimensions. Having run the proposed implementation on this data set, we obtain the results shown in Fig. 2. We obtained similar results when running the implementation from [5]. Table I compares the runtime of both implementations. Recall that the implementation from [5] does not support multiple machines. On a single machine, and for this low dimensionality (*i.e.*, $d = 2$), the proposed Julia implementation is between 2 to 4 times slower than their optimized C++ implementation. As we increase the dimensionality of the data set, however, the proposed implementation performs relatively better; *e.g.*, when we increase the dimensionality to 30, Table II shows that the proposed implementation is almost 4 times as fast on a single machine than the C++ implementation and that we also obtain additional good speedup on multiple machines. This is probably since their implementation was optimized for low dimensions. However, we have also noted (empirically) that the implementation from [5] fails when we go past a certain dimension (while the proposed implementation still succeeds); *e.g.*, when we set the dimensionality to 250, the implementation from [5] converged, wrongly, to 1 cluster while the proposed implementation converged to the correct 6 clusters (attempts to debug their implementation suggest that the problem is due to underflow/overflow). This observation is consistent with the claim that it is easier to develop, read, and debug algorithms in a high(er)-level language like Julia than in a language like C++. The timing results we obtained suggest that this does not have to come at the expense of performance.

Synthetic Data: DP-MNMM. In this experiment we generated synthetic data by drawing 10^6 points, of dimension 100, from an underlying MNMM with 6 clusters. The results we obtained using both the proposed implementation and the one from [5] are similar. Table III compares runtimes, showing that the proposed Julia implementation is around between 2 times to 4 times slower on a single machine; this is consistent with the typical relative performance of Julia to C/C++ given in http://julialang.org/benchmarks/. Note, however, that the C++ code form [5] used sparse arrays for the DP-MNMM which saved some computation time; presumably, the proposed implementation could have benefited from sparse arrays in a similar way, but we have not explored that direction. Table III shows that, for such a small K, adding machines still did not help the proposed Julia implementation beat the single-machine C++ implementation. When increasing the true K to 60, however, a speedup is obtained; see Table IV.

Finally, we also tried changing the dimensionality to see if has a similar effect as it had with the DP-GMM; in this case, however (and unlike the DP-GMM case), we did not find that the dimensionality impacted relative performance between the implementations.

Image-patch Models and Image Denoising. Zoran and Weiss [38] used a finite GMM to model image patches, having trained their GMM (in a non-Bayesian fashion) on a subset of 2 million (out of 44 million patches that exist in the entire training set available) 8×8 image patches from BSDS500 [1]. Having tried different values of K manually, they ended up setting $K = 200$. Next, they used the learned model within an image-denosing method they proposed. They released their image-denoising code and learned model but not their GMM-learning code (which was based on EM). We applied the proposed DP-GMM inference implementation to learn an image-patch model from a similar subset of 2 million patches from BSDS500. Using 4 machines, this took us less than 1 hour (Zoran and Weiss reported 30 hours for their serial MATALB implementation of GMM, which explains why they avoided using the entire training data). Next, we used the learned DP-GMM, with their code, for image denoising. We also experimented with different values of α. Because we were able to run 2 million patches with such an ease, we also ran all of the $\sim$44 million 8×8 patches in BSDS500 (the test set excluded). On 4 machines, this took us less than 2 days. We compared the models we learned with the one provided by [38] by performing image denoising on 50 test images from the dataset. The results are summarized in Table V. For visual examples, see Fig. 3.

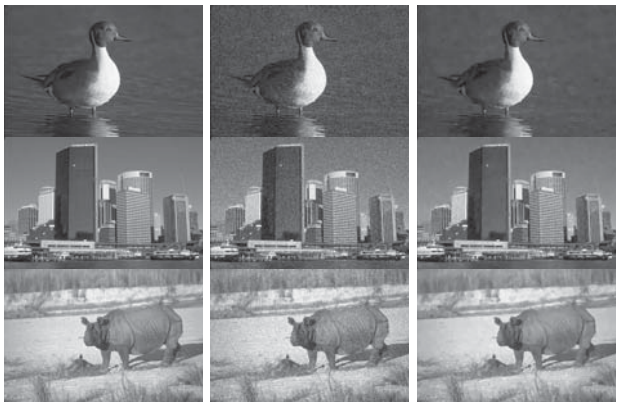

Fig. 3: Denoising examples using a DP-GMM we trained on 44 millions 8×8 patches. From left to right: originals images, corrupted images (with an ave. PSNR of 20.17), and denoised images (ave. PSNR: 29.14.).

VI. Conclusion

We extended a recent DPMM inference algorithm, proposed by Chang and Fisher [5], by showing how it can be distributed, using Julia and a distributed-memory model, across multiple multi-core machines. Like the (single-machine, multi-core) C++ implementation from [5], the proposed implementation supports both DP-GMM and DP-MNMM models. Empirically, we found that on a single machine the proposed implementation was usually slower than implementation from [5]. However, for DP-GMM in high dimensions (and still on a single machine), the proposed implementation was faster and also produced better results – probably since the implementation

from [5] was optimized for low dimensions. In any case, the point of re-implementing in Julia was not, of course, to try to gain speedups on a single machine. Rather, we wanted an implementation that: 1) lets us easily distribute inference over multiple multi-core machines; 2) is written in a higher-level language and that is still fast. Judging by these two criteria, we deem our experience with Julia extremely positive. We also applied the proposed implementation for learning image-patch models and used the latter for image denoising within an framework proposed in [38]. We obtained comparable results to [38] when using a training set of 2 million patches (like they did), but note that the proposed learning was much faster and we inferred K automatically from the data while they fixed it manually. We were also able to handle the entire training set of 44 million 8×8 patches with the proposed distributed implementation – something which was probably infeasible with their GMM implementation (which is not publicly available) and is infeasible using the DP-GMM implementation from [5]. This led to slightly better results, highlighting the utility of DPMMs and the importance of being able to handle large datasets.

REFERENCES

[1] P. Arbelaez, M. Maire, C. Fowlkes, and J. Malik. Contour detection and hierarchical image segmentation. *IEEE TPAMI*, 2011. 7

[2] L. D. Brown. *Fundamentals of statistical exponential families with applications in statistical decision theory*. JSTOR, 1986. 2, 5

[3] T. Campbell, J. Straub, J. W. Fisher, and J. How. Streaming, distributed variational inference for Bayesian nonparametrics. In *NIPS*, 2015. 2

[4] J. Chang, R. Cabezas, and J. W. Fisher. Bayesian nonparametric intrinsic image decomposition. In *ECCV*, 2014. 2

[5] J. Chang and J. W. Fisher. Parallel sampling of DP mixture models using sub-cluster splits. In *NIPS*, 2013. 2, 3, 4, 5, 6, 7, 8

[6] R. Emonet, J. Varadarajan, and J.-M. Odobez. Extracting and locating temporal motifs in video scenes using a hierarchical non parametric Bayesian model. In *CVPR*, 2011. 2

[7] T. S. Ferguson. A Bayesian analysis of some nonparametric problems. *The Annals of Statistics*, 1973. 3

[8] R. Gomes, M. Welling, and P. Perona. Incremental learning of nonparametric Bayesian mixture models. In *CVPR*, 2008. 2

[9] T. S. Haines and T. Xiang. Background subtraction with Dirichlet. In *ECCV*, 2012. 2

[10] T. M. Hospedales, S. Gong, and T. Xiang. A unifying theory of active discovery and learning. In *ECCV*, 2012. 2

[11] M. C. Hughes and E. Sudderth. Memoized online variational inference for Dirichlet process mixture models. In *NIPS*, 2013. 3

[12] M. C. Hughes and E. B. Sudderth. bnpy: Reliable and scalable variational inference for Bayesian nonparametric models. In *NIPSW on Probabilistic Programming*, 2014. 2

[13] K. Ishiguro, T. Yamada, and N. Ueda. Simultaneous clustering and tracking unknown number of objects. In *CVPR*, 2008. 2

[14] S. Jain and R. M. Neal. A split-merge Markov chain Monte Carlo procedure for the Dirichlet process mixture model. 2012. 2

[15] A. Jara. R package for Bayesian Non- and Semi-parametric Analysis, www.mat.uc.cl//~ajara/Softwares.html. 2

[16] Y.-D. Jian and C.-S. Chen. Two-view motion segmentation by mixtures of Dirichlet process with model selection and outlier removal. In *ICCV*, 2007. 2

[17] K. M. Kitani, T. Okabe, Y. Sato, and A. Sugimoto. Fast unsupervised ego-action learning for first-person sports videos. In *CVPR*, 2011. 2

[18] J. J. Kivinen, E. B. Sudderth, and M. I. Jordan. Learning multiscale representations of natural scenes using Dirichlet processes. In *ICCV*, 2007. 2

[19] D. Kuettel, M. D. Breitenstein, L. Van Gool, and V. Ferrari. What's going on? discovering spatio-temporal dependencies in dynamic scenes. In *CVPR*, 2010. 2

[20] K. Kurihara. MATLAB package for Variational DP-GMM, sites.google.com/site/kenichikurihara/academic-software. 2

[21] L.-J. Li and L. Fei-Fei. Optimol: automatic online picture collection via incremental model learning. *IJCV*, (2), 2010. 2

[22] R. Li and R. Chellappa. Group motion segmentation using a spatio-temporal driving force model. In *CVPR*, 2010. 2

[23] D. Lin. Online learning of nonparametric mixture models via sequential variational approximation. In *NIPS*, 2013. 2

[24] C. C. Loy, T. M. Hospedales, T. Xiang, and S. Gong. Stream-based joint exploration-exploitation active learning. In *CVPR*, 2012. 2

[25] P. Orbanz and J. M. Buhmann. Smooth image segmentation by nonparametric Bayesian inference. In *ECCV*, 2006. 2

[26] C. Robert and G. Casella. *Monte Carlo statistical methods*. Springer Science & Business Media, 2013. 4

[27] M. Sanzari, V. Ntouskos, and F. Pirri. Bayesian image based 3d pose estimation. In *ECCV*, 2016. 2

[28] A. Shyr, T. Darrell, M. Jordan, and R. Urtasun. Supervised hierarchical Pitman-Yor process for natural scene segmentation. In *CVPR*, 2011. 2

[29] S. D. Silvey. *Statistical inference*. CRC Press, 1975. 5

[30] J. Straub, T. Campbell, J. P. How, and J. W. Fisher. Small-variance nonparametric clustering on the hypersphere. In *CVPR*, 2015. 2

[31] E. B. Sudderth. *Graphical models for visual object recognition and tracking*. PhD thesis, MIT, 2006. 3

[32] E. B. Sudderth and M. I. Jordan. Shared segmentation of natural scenes using dependent Pitman-Yor processes. In *NIPS*, 2009. 2

[33] E. B. Sudderth, A. Torralba, W. T. Freeman, and A. S. Willsky. Depth from familiar objects: A hierarchical model for 3d scenes. In *CVPR*, 2006. 2

[34] E. B. Sudderth, A. Torralba, W. T. Freeman, and A. S. Willsky. Describing visual scenes using transformed objects and parts. *IJCV*, 2008. 2

[35] A. Torralba, A. S. Willsky, E. B. Sudderth, and W. T. Freeman. Describing visual scenes using transformed Dirichlet processes. In *NIPS*, 2005. 2

[36] M. Trapp. BNP. jl: Bayesian nonparametrics in Julia, github.com/trappmartin/BNP.jl. 2

[37] X. Wang, X. Ma, and W. E. L. Grimson. Unsupervised activity perception in crowded and complicated scenes using hierarchical Bayesian models. *IEEE TPAMI*, 2009. 2

[38] D. Zoran and Y. Weiss. From learning models of natural image patches to whole image restoration. In *ICCV*, 2011. 3, 4, 5, 7, 8