Parallel and Distributed MCMC Inference using Julia

by

Angel Yu

S.B. Computer Science and Engineering, Mathematics
Massachusetts Institute of Technology, 2016

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering
in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

September 2016

© 2016 Massachusetts Institute of Technology
All Rights Reserved.

Signature of Author: ________________________________

Department of Electrical Engineering and Computer Science
August 16, 2016

Certified by: _______________________________________

John W. Fisher III, Senior Research Scientist
Thesis Supervisor

Certified by: _______________________________________

Oren Freifeld, Postdoctoral Associate
Thesis Co-Supervisor

Accepted by: _______________________________________

Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee
Parallel and Distributed MCMC Inference using Julia

by

Angel Yu

Submitted to the Department of Electrical Engineering and Computer Science on August 16, 2016 in Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

Machine learning algorithms are often computationally intensive and operate on large datasets. Being able to efficiently learn models on large datasets holds the future of machine learning. As the speed of serial computation stalls, it is necessary to utilize the power of parallel computing in order to better scale with the growing complexity of algorithms and the growing size of datasets. In this thesis, we explore the use of Julia, a fairly new high level programming language that lends itself to easy parallelization over multiple CPU cores as well as multiple machines, on Markov chain Monte Carlo (MCMC) inference algorithms.

First, we take existing algorithms and implement them in Julia. We focus on MCMC inference using Continuous Piecewise-Affine Based (CPAB) transformations and a parallel MCMC sampler for Dirichlet Process Mixture Models (DPMM). Instead of parallelizing over multiple cores on a single machine, our Julia implementations extend existing implementations by parallelizing over multiple machines. We compare our implementation with these existing implementations written in more traditional programming languages. Next, we develop a model Projections Dirichlet Process Gaussian Mixture Model (PDP-GMM) which relaxes the assumption that the draws from a Dirichlet Process Gaussian Mixture Model (DP-GMM) are directly observed. We extend our DPMM Julia implementation and present a few applications of this model.

Thesis Supervisor: John W. Fisher III
Title: Senior Research Scientist
First of all, I would like to thank my advisor John Fisher for his guidance throughout my time in the group. He has always provided valuable insight and ideas during group meetings as well as grouplets.

I would like to thank my co-advisor Oren Freifeld for his mentorship and help with all my questions. Oren introduced me to the topic of Bayesian Inference and was always available to help me work out any technical difficulties. This work would not have been possible without him.

I would also like to thank my collaborator Sue Zheng for providing the derivations for our PDP-GMM model in Chapter 4. Her contributions were key to the work presented in Chapter 4.

Finally, I would like to thank my family and friends. In particular, my parents have always been supportive of me throughout my MIT journey.
List of Figures

2.1 (a) An example of a 1-dimensional velocity field. Here the tessellation contains 5 cells equally spaced between 0 and 10. (b) An example of a 2-dimensional tessellation we will use. Here the tessellation contains 64 cells. ................................. 18

2.2 An inference problem: Inferring the underlying parameters of a CPAB transformations that maps the red points to the blue points. .................. 21

2.3 (a) Original Image. (b) Image after applying a CPAB transformation. .... 26

2.4 Applying Gradient Descent to infer the underlying CPAB transformation. The points in blue are the target points and the points in red are the points after applying the inferred transformation. .................. 28

2.5 (a) Applying Metropolis’ Algorithm to infer the underlying CPAB transformation. The points in blue are the target points and the points in red are the points after applying the inferred transformation. (b) Plot showing the log likelihood and log prior over 50,000 iterations. ............... 29

2.6 (a) Applying the Sequential Importance Resampling algorithm to infer the underlying CPAB transformation. The points in blue are the target points and the points in red are the points after applying the inferred transformation. (b) Plot showing the log likelihood and log prior over 200 iterations. ............................... 30

2.7 A plot showing the parallelization speedup obtained when running the Sequential Importance Resampling inference algorithm with 1,000 particles on 512 x 512 correspondences. .............................. 31

3.1 The proposed architecture of our proposed multi-machine implementation. We show a parallelization over 4 machines here. ................. 40

3.2 (a) The 10^6 points in our synthetic data, in the 2-dimensional case, sampled from an underlying GMM with 6 clusters are shown in blue. (b) The inferred clusters from our implementation are shown as ellipses at 5 standard deviations in different colors. ......................... 48

3.3 Eigenvalues of 6 randomly-chosen clusters learned by our DP-GMM on 2 million 8 x 8 image patches. ............................... 50
3.4 An example of denoising using our model trained on all $8 \times 8$ patches. The corrupted image (middle) has a PSNR of 20.17 and the restored image (right) has a PSNR of 29.14. ........................................ 52

4.1 (Left) We show the KDE estimates of the 1-dimensional marginals computed from the observations. (Right) We show both the unobserved draws as well as the KDE estimates. ........................................ 58

4.2 A simple example with 10,000 points and 3 clusters. We show the inferred cluster means and (isotropic) covariances as circles at 4 standard deviations together with the estimated 1-dimensional marginals and the unobserved 2-dimensional draws. ........................................ 59

4.3 A more complicated example with 10,000 points and 10 clusters. (Left) We show the inferred cluster means and (isotropic) covariances as circles at 3 standard deviations together with the estimated 1-dimensional marginals and the unobserved 2-dimensional draws. (Right) We show the weights of each inferred cluster. ........................................ 60

4.4 An example of using our model to fit a mixture of linear regressions. (Left) We show the observations. (Right) We show the resulting clusters after inference. The black lines are the ground truth lines. The colored lines are cluster centers inferred using our model. Importantly, note that we did not assume that we know that the number of the regression functions, $K$, is three. Rather, this quantity was estimated as part of our inference procedure. ........................................ 61

4.5 An example of using our model to fit a mixture of quadratic regressions. (Left) We show the observations. (Right) We show the resulting clusters after inference. The black curves are the ground truth quadratics. The colored curves are cluster centers inferred using our model. Importantly, note that we did not assume that we know that the number of the regression functions, $K$, is three. Rather, this quantity was estimated as part of our inference procedure. ........................................ 62
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Time to compute a CPAB transformation on a 512x512 image on an Intel Core i7-6700K. The graphics card used in [8] was an Nvidia GTX 780.</td>
<td>27</td>
</tr>
<tr>
<td>2.2</td>
<td>Running time of the Sequential Importance Resampling inference algorithm with 1,000 particles for 200 iterations on 256 correspondences on an Intel Core i7-6700K.</td>
<td>30</td>
</tr>
<tr>
<td>2.3</td>
<td>Running time of each iteration of the Sequential Importance Resampling inference algorithm with 1,000 particles on 512 $\times$ 512 correspondences using distributed computing.</td>
<td>31</td>
</tr>
<tr>
<td>3.1</td>
<td>Time(sec) to run 100 DP-GMM iterations of $d = 2, N = 10^6, K = 6$.</td>
<td>48</td>
</tr>
<tr>
<td>3.2</td>
<td>Time(sec) to run 100 DP-GMM iterations of $d = 30, N = 10^6, K = 6$.</td>
<td>49</td>
</tr>
<tr>
<td>3.3</td>
<td>Time(sec) to run 100 DP-MNMM iterations of $d = 100, N = 10^6, K = 6$.</td>
<td>50</td>
</tr>
<tr>
<td>3.4</td>
<td>Time(sec) to run 100 DP-MNMM iterations of $d = 100, N = 10^6, K = 60$.</td>
<td>50</td>
</tr>
<tr>
<td>3.5</td>
<td>Average PSNR obtained from denoising 50 test images with added Gaussian noise using different priors.</td>
<td>51</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The clock speed of modern processors is beginning to hit its limit. This is because power consumption and heat dissipation increases much faster than the clock speed of the chip. Hence, we now see a trend towards processors with multiple cores instead of higher clock speeds. In order to fully utilize the power of these processors, we would have to design and implement algorithms that make effective use of parallel computing. Designing and implementing efficient parallel algorithms are often harder and more complicated than their serial counterparts. This is because additional care must be taken to design meaningful subtasks so as to minimize communication and synchronization overhead.

Julia is a fairly new high level programming language that has parallel computing built in. Its parallel computing model allows for easy parallelization across multiple cores as well as multiple machines, something that is usually hard to achieve in most other programming languages. It is also syntactically similar to MATLAB but with performance comparable to C. This makes Julia an attractive language for developing parallel implementations of machine-learning algorithms.

1.1 Thesis Outline and Contributions

In this thesis, we explore the utility of using Julia for machine-learning algorithms. We use Julia for parallel and distributed implementations of two different Bayesian MCMC inference algorithms and compare their performance with existing implementations. We
discuss various challenges and our experience with parallel implementations in Julia.

**Chapter 2: CPAB Transformations**

In Chapter 2, we implement inference on latent Continuous Piecewise-Affine Based (CPAB) transformations proposed by Freifeld et al. [8]. There is currently a GPU implementation of CPAB transformations and we compare that with our Julia implementation. We apply CPAB transformations to the correspondence-based image warping problem by using Bayesian inference on the transformations. We experimented with 3 different methods: Gradient Descent, Metropolis' Algorithm and Sequential Importance Resampling. We discuss our implementation using Julia as well as the suitability of these methods to the problem including how they lend themselves to parallelization.

**Chapter 3: Parallel Sampling of DPMM using Subcluster splits**

In Chapter 3, we implement a Dirichlet Process Mixture Model (DPMM) with subcluster splits that was proposed by Chang and Fisher III [4]. That algorithm is based on the split-merge framework by Jain and Neal [14]. It was designed as a parallel algorithm and there is a (single-machine) multi-core C++ implementation available. We extend this algorithm to accommodate a distributed memory model which allows our implementation to be able to parallelize over multiple machines. We describe our implementation design and compare our distributed implementation with the existing C++ implementation. In addition, we applied our DP-GMM implementation to learn image patch priors for image denoising. We show that with our distributed implementation, we are able to handle much larger datasets.
Chapter 4: Projection Dirichlet Process Gaussian Mixture Model

In Chapter 4, we propose a new model which relaxes the assumption that the draws from the Dirichlet Process Gaussian Mixture Model (DP-GMM) are directly observed. Our model instead assumes that the observations are merely affine projections of the DP-GMM draws. This allows us to tackle interesting problems in a new light. We discuss our experience with modifying our DPMM code to handle this new model and present a few applications of this model.

Chapter 5: Conclusion

Finally, we summarize our work and contributions in this thesis. We offer some general comments on our experience with Julia.

Appendix A: Derivations Pertaining to PDP-GMM

A few of the expressions presented in Chapter 4 require a considerable amount of algebra to derive. We present these derivations in Appendix A for the interested reader.
Diffeomorphisms are bijective differentiable transformations such that the inverse is differentiable as well. They have many applications in a wide variety of domains. In this thesis, we focus on the computer vision applications of image warping and image registration. However, current representations of this class of transformations are complicated and hard to compute on large datasets. CPAB transformations proposed by Freifeld et al. [8] are a representation of a sub-class of these transformations that can be computed in high accuracy efficiently while maintaining a high level of expressiveness. That class of transformations is based on the integration of Continuous Piecewise-Affine (CPA) velocity fields. Hence, they are referred to as CPA-based (CPAB) transformations. CPA velocity fields are continuous velocity fields that are piecewise affine with respect to a tessellation of the domain. Because these are continuous velocity fields, we are able to define a trajectory \( \phi(x, t) \) at every point \( x \) in the space. Here a trajectory refers to a path in space as a function of time and the origin point. This trajectory \( \phi(x, t) \) is given by the following integral equation:

\[
\phi^\theta(x, t) = x + \int_0^t v^\theta(\phi^\theta(x, \tau)) d\tau, \tag{2.1}
\]

where \( \theta \) is the parameter of the CPA velocity field and \( v^\theta \) is the velocity function. By fixing \( t \) and mapping all points to the point at time \( t \) on its trajectory, we obtain a CPAB
transformation. It can be shown that this transformation is indeed a diffeomorphism as the inverse is defined by the negative of the velocity field.

To calculate this trajectory, we could have used standard numerical methods to approximate the integral of (2.1). However, Freifeld et al. [8] has shown that one can combine a tessellation of the domain along with a specific parameterization of the velocity field to achieve a highly accurate and efficient method for computing CPAB transformations. See Algorithm 1 and its discussion in [8] for more details.

For a 1-dimensional domain, we tessellate the domain into segments on the $x$-axis. As the velocity field is CPA, the velocity in each cell is affine and has to be continuous across different cells. An example of such a velocity field is given in Figure 2.1a. In the 1-dimensional case, the number of parameters (i.e., $\dim(\theta)$) is given by the number of vertices which in this example is 6.

![Figure 2.1](image_url)

Figure 2.1: (a) An example of a 1-dimensional velocity field. Here the tessellation contains 5 cells equally spaced between 0 and 10. (b) An example of a 2-dimensional tessellation we will use. Here the tessellation contains 64 cells.

For a 2-dimensional domain, we choose a tessellation which allows for fast computation of which cell a point is in, although, other tessellation choices can certainly be used. We first partition the 2-dimensional domain into a rectangular grid and then
for each rectangle, we draw the diagonals to create 4 triangles. An example of such a tessellation is given in Figure 2.1b. There are 64 cells in this example tessellation with the added constraint that the velocity field has to be continuous. Without this continuity constraint, the dimension would have been $6 \times 64$ as there are 6 degrees of freedom for each cell. The continuity constraints reduce the dimensionality to 58. See lemma 5 in [8] for more details.

2.1 Bayesian Inference using CPAB transformations

We apply CPAB transformations to the problem of correspondence-based image warping. The problem is, given 2 images and a set of correspondences, to infer the underlying transformation that mapped the source image to the destination image. This is illustrated in Figure 2.2 where we would like to find a CPAB transformation that maps the red points to the blue points. Note that the figure does not show the explicit correspondences, but we treat them as known in the inference. In most cases for a given tessellation, there may not exist any $\theta$ that maps the source points to the destination points with no error. Hence, we formulate this as a probabilistic inference problem where we want to infer the parameters $\theta$ of a CPAB transformation that maximizes the posterior. This allows for solutions with transformations that do not match the correspondences perfectly. We model the destination points via a spherical Gaussian noise centered about their corresponding transformed source points. We have the following likelihood distribution:

$$P(X|\theta) = C_1 e^{-\frac{\sum_i ||T^{\theta}(x_{i,t}) - y_i||^2}{2\sigma^2}}$$ (2.2)

where $C_1$ is a constant (w.r.t. $\theta$), $\theta$ is the parameter of the CPAB transformation, $t$ is the time to evaluate the trajectory, $\sigma$ is the standard deviation of the Gaussian, $X = x_1, x_2, \ldots, x_n$ are the source points and $Y = y_1, y_2, \ldots, y_n$ are the destination points.
points. This distribution obtains its maximum when the sum of squared errors obtains its minimum. In addition, we also have a Gaussian prior over $\theta$ which encourages smoothness of the velocity field; see [8] for more details. Thus, the posterior distribution is:

$$P(\theta|X) \propto P(X|\theta)P(\theta)$$

$$= C_1 e^{-\frac{1}{2\sigma^2} \sum T(\theta(x_i,t) - y_i)^2} C_2 e^{-\frac{1}{2}(\theta - \theta_\mu)^T \Sigma^{-1}(\theta - \theta_\mu)}$$

where $C_1, C_2$ are constants (w.r.t. $\theta$), $\theta_\mu$ is the mean of the prior and $\Sigma_\theta$ is the covariance of the prior. We now have an optimization problem where we want to maximize the posterior $P(\theta|X)$. To avoid overflow and underflow, we define our objective function $f(\theta)$ to be the posterior in the log domain:

$$f(\theta) = \frac{1}{2}(\theta - \theta_\mu)^T \Sigma^{-1}(\theta - \theta_\mu) - \frac{1}{2\sigma^2} \sum_i ||T(\theta(x_i,t) - y_i)||^2,$$

We consider three different optimization techniques to minimize this objective function. These are Gradient Descent, Metropolis’ algorithm and Sequential Importance Resampling. We then discuss the utility of implementing these methods in Julia. In particular, we focus on the ease of coding the implementation and the performance of the implementation.

### 2.1.1 Gradient Descent

Gradient Descent [2] is one of the most popular methods in optimization as it is easy to implement and understand. Gradient Descent finds a local minimum of an objective function by taking steps in the direction of the steepest descent. We start with an initial value of $\theta_0$. At each iteration, we find the gradient $f'(\theta_i)$ numerically and update the parameter as follows:

$$\theta_{i+1} = \theta_i - \gamma f'(\theta_i)$$
Figure 2.2: An inference problem: Inferring the underlying parameters of a CPAB transformations that maps the red points to the blue points.

where $\gamma$ is a chosen step size which determines the rate of convergence.

A significant drawback of using Gradient Descent is that it can only find local minima and in most cases we would like to find the global minimum. A common workaround is to try multiple initial $\theta_0$ and running Gradient Descent on each of them. However, even with this, it will perform poorly on a function with many local minima, especially when the dimension of $\theta$ is high. In our implementation for CPAB, we observe that gradient-based optimization methods do not perform well in comparison to the other methods.
2.1.2 Metropolis’ Algorithm

Metropolis’ Algorithm [15] is an MCMC sampling algorithm that allows us to sample from a difficult-to-sample probability distribution \( P(x) \). Again, we start off with an initial value of \( x_0 \). At each iteration, we have a symmetric proposal distribution \( Q(x|x_i) \).

A common choice for continuous parameters is to use a Gaussian distribution centered at \( x_i \). We generate a sample \( x' \) from this proposal distribution. We then calculate the acceptance ratio \( \alpha = P(x')/P(x_i) \). We automatically accept this sample if the acceptance ratio \( \alpha \geq 1 \) and set \( x_{i+1} = x' \). Otherwise, we accept \( x' \) with probability \( \alpha \) by setting \( x_{i+1} = x' \) and reject with probability \( 1 - \alpha \) by setting \( x_{i+1} = x_i \). Note that we do not need to calculate \( P(x) \) explicitly, we only need to be able to calculate a \( g(x) \propto P(x) \) since normalization terms cancel in the ratio. Using Metropolis’ Algorithm to sample from a distribution \( P(x) \), we can also approximate the mode of the distribution by storing the greatest value of \( P(x) \) seen so far at each iteration. This optimization method is likely to produce better results than Gradient Descent for objectives with multiple local optima. This is a consequence of occasionally accepting samples that might not be better than the previous sample and thereby escaping local optima. We note that this algorithm is serial and thus, the only parallelization has to come from evaluating \( P(x) \).

2.1.3 Sequential Importance Resampling

Another popular MCMC algorithm is Sequential Importance Resampling [11]. Instead of sampling one sample each iteration in Metropolis, we now sample multiple samples (particles) each iteration that approximates the distribution. For a target distribution \( P(x) \) we first initialize a set of particles randomly sampled from the prior. At each iteration, we calculate a weight for each particle proportional to their probability such that the weights sum up to 1. We then resample another set of particles from these
particles based on the weights distribution. So, higher weighted samples will have a higher chance of being picked. After getting the resampled particles, for each particle, we perturb it with a perturbation distribution centered at that particle. We have now obtained our set of particles for the next iterations. After some number of iterations, we should obtain a good approximation of the target distribution.

We now apply it to our problem. In our case, our weights \( w_j \) are defined to be:

\[
\tilde{w}_j = P(\theta^i_j | X) \tag{2.8}
\]
\[
w_j = \frac{\tilde{w}_j}{\sum_k \tilde{w}_k} \tag{2.9}
\]

where \( \theta^i_j \) is the \( j \)th particle in the \( i \)th iteration and \( P(\theta | X) \) is given by Equation (2.5). Since we are doing operations on multiple particles in each iteration, we see that we are able to parallelize the computation over the particles. In addition, this algorithm is able to escape local minima as long as the perturbation distribution is chosen so that the model explores particles sufficiently far from the original particles.

### 2.2 Implementation and Results

We implemented CPAB transformations as well as Metropolis’ Algorithm and Sequential Importance Resampling in Julia. In addition, we used a Julia package `Optim` which includes Gradient Descent. We now look at some implementation details along with the results we obtained from our experiments. Our code is available online at https://github.com/angel8yu/cpab-diffeo-julia.

#### 2.2.1 Parallel Computing in Julia

Parallel Computing in Julia is built on managing different processes through a single master process. Typically, a number of worker processes are added at the start of a Julia session and the master process sends work to each process by calling the `@spawnat` macro, `remotecall_fetch` or `remotecall_wait` functions. Julia currently does not have
An important part of designing a parallel application is working out how the data is stored. There are two ways to store data so it can be processed in parallel in Julia. The first way is to use an `SharedArray` object which uses shared memory so it can be accessed across different processes. The other way is to use `DistributedArray` objects where each participating process has a chunk of data it is in charge of. An advantage of `SharedArray` is that each process has read-and-write access to the entire array, while `DistributedArray` has read-and-write access only to its own chunk of data. It is also possible for a process to read data that it is not in charge of. However, the following is what actually happens when a process reads an entry not part of its own chunk. The process first sends a request to the master for the entry which then looks up the corresponding owner process that owns the entry. The master then requests the entry from the owner process and once it gets it, the master will forward the entry to the original process. The problem, however, is that the communication overhead of this can quickly dominate processing time if processes constantly require reading data remotely. On the other hand, `SharedArray` allows for both fast reads and writes on the entire data from every process. One major disadvantage of `SharedArray` is that the data can only be distributed and accessed from processes on a single machine, while `DistributedArray` allows for data to be distributed across processes on different machines. This means that for multi-machine parallelization, we are mostly limited to using `DistributedArray` to store the data in memory.

A simple example of parallelizing the computation of the function `f` which acts over a `DistributedArray` is shown as follows:

```julia
@sync for p in workers()
    @async remotecall_wait(p, f, A)
end
```
Here, $f$ can be a simple function which doubles all the entries of $A$ that the current process is in charge of like below:

```julia
@everywhere function f(A::DArray{Float64,1})
    local_A = localpart(A)
    for i=1:length(local_A)
        local_A[i] = 2*local_A[i]
    end
end
```

The previous 2 code snippets include most of the commonly used inbuilt Julia functions for parallel computing. @sync blocks will wait until all asynchronous tasks to complete before moving on. @async blocks creates an asynchronous task that will be executed asynchronously on the same process. remotecall_wait will spawn a function call at a process indicated by the first argument and wait until it has finished. In the example, it will spawn the function call $f(A)$ on process number $p$. Together, the 3 lines of code will parallelize the computation of $f$ over the DistributedArray $A$. In the implementation of $f$, we obtain the locally stored portion of the DistributedArray $A$ by using the inbuilt function localpart. As we can see with this simple example, it is straightforward to parallelize code in Julia while still maintaining a granular level of control over how the data is stored and computed. For an in-depth discussion of parallel computing in Julia, see [http://docs.julialang.org/en/release-0.4/manual/parallel-computing/](http://docs.julialang.org/en/release-0.4/manual/parallel-computing/).

### 2.2.2 CPAB transformations

We implemented the calculation of both 1-dimensional and 2-dimensional CPAB transformations in Julia. Note that this is the implementation of the CPAB transformation, not the inference which will be discussed later. Figure 2.3 shows an example of a 2-dimensional CPAB transformation on an image. Since, we are computing each pixel independently, this problem is embarrassingly parallel. Using Julia’s parallel comput-
ing framework, it was simple and straightforward to parallelize this computation. We store the source points in a **SharedArray** so that all processes have access to the entire array. Each worker then calculates the destination points for an equal number of source points. The computation of the destination point is implemented based on Algorithm 1 in [8]. Implementing this function is straightforward as Julia is syntactically similar to MATLAB although with a few noteworthy differences which are summarized in [http://docs.julialang.org/en/release-0.4/manual/noteworthy-differences/](http://docs.julialang.org/en/release-0.4/manual/noteworthy-differences/).

The time needed to compute a 2-dimensional CPAB transformation on a 512x512 image is shown in Table 2.1. Even though this problem is embarrassingly parallel, we can see from the timings that there is much communication overhead in the parallelization. We found that the majority of the overhead comes from using the **@spawnat** macro which spawns work at a specific process. We suspect this is because the calculation of the transformation is quite fast and **@spawnat** is more suitable for spawning larger chunks of work. Freifeld et al. [8] reported a time of 0.15s to do this computation using a GPU while our Julia implementation takes 0.37s on 8 CPU processes.

![Figure 2.3: (a) Original Image. (b) Image after applying a CPAB transformation.](image)
Section 2.2. Implementation and Results

<table>
<thead>
<tr>
<th>Number of processes</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>GPU [8]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time [sec]</td>
<td>0.78</td>
<td>0.49</td>
<td>0.38</td>
<td>0.37</td>
<td>0.15</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.00x</td>
<td>1.59x</td>
<td>2.05x</td>
<td>2.11x</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 2.1: Time to compute a CPAB transformation on a 512x512 image on an Intel Core i7-6700K. The graphics card used in [8] was an Nvidia GTX 780.

### 2.2.3 Gradient-based Optimization

We used Julia’s Optim package and applied Gradient Descent to the problem described in Figure 2.2, namely inferring the underlying CPAB transformation given 16x16=256 correspondences. The algorithm converged after 359 iterations and the result is shown in Figure 2.4. As this figure shows, the transformation inferred only somewhat matches the underlying transformation. This is most likely because the transformation found is a local minimum in our objective function. We also tried using Conjugate Gradient Descent [16] and we obtained almost the same result, but it did converge faster than regular Gradient Descent. We tried parallelizing over the 16x16 points when computing the objective function. However, because the number of points is very small, we did not observe any speedup from parallelizing.

### 2.2.4 Metropolis’ Algorithm

We implemented Metropolis’ Algorithm described in Section 2.1.2 and applied it to infer the underlying CPAB transformation in Figure 2.2 given a prior distribution. The prior is a multivariate Gaussian distribution with a mean of $\vec{0}$ and some covariance $\Sigma_{\text{prior}}$. We ran the inference algorithm for 50,000 iterations and set our proposal distribution to be a Gaussian centered at the current sample with a covariance of $0.00001\Sigma_{\text{prior}}$. The result of this algorithm is shown in Figure 2.5a. We can see that the points obtained using the inferred transformation match pretty well with the target points. This is much better than the results we obtained from using Gradient Descent. We note that there is an
CHAPTER 2. CPAB TRANSFORMATIONS

Figure 2.4: Applying Gradient Descent to infer the underlying CPAB transformation. The points in blue are the target points and the points in red are the points after applying the inferred transformation.

underlying CPAB transformation that maps the source points to the destination points exactly, although, we did not find it using this algorithm. In Figure 2.5b, we can see that the algorithm converges at around 20,000 iterations. We tried parallelizing this by parallelizing the calculation of CPAB transformations described earlier. However, since there are only 256 points, it runs very fast and parallelizing it actually slows it down because of the overhead in communication. In particular, recall that we did not see great speedup when parallelizing the computation over 512x512 points in Section 2.2.2, so it is not surprising that parallelizing hurts us with only 256 points. This is due to the fact that the overhead when spawning work is mostly constant with respect to the number of points. We also note that we found a small Julia bug while running these experiments. After talking to a Julia developer, it seems that there is a race condition when spawning processes. In particular, a segmentation fault will occur when spawning a large number of processes in a short time span. We have submitted an issue ticket on
Sec. 2.2. Implementation and Results

github at: https://github.com/JuliaLang/julia/issues/13999.

Figure 2.5: (a) Applying Metropolis’ Algorithm to infer the underlying CPAB transformation. The points in blue are the target points and the points in red are the points after applying the inferred transformation. (b) Plot showing the log likelihood and log prior over 50,000 iterations.

2.2.5 Sequential Importance Resampling

We implemented the Sequential Importance Resampling algorithm described in Section 2.1.3 and applied it to infer the underlying CPAB transformation in Figure 2.2 given a prior distribution of $\theta$. We again have the prior as a multivariate Gaussian distribution with a mean of $\vec{0}$ and covariance of $\Sigma_{\text{prior}}$. We took the number of particles to be 1,000 and the number of iterations to be 200. We have considerably fewer iterations than in Metropolis’ Algorithm as now we explore 1,000 samples in each iteration instead of just 1. In addition, we let the perturbation distribution be a Gaussian centered at the current particle with covariance $0.001\Sigma_{\text{prior}}$. The result of this algorithm is shown in Figure 2.6a. We see that the inferred points almost match the target points and it is comparable with the results from using Metropolis’ algorithm. However, this algorithm allows for parallelization over the particles instead of points. This allows for a bigger amount of computation that is sent to the workers each time which improves the communication overhead. The time needed is summarized in Table 2.2. We see
that we obtain almost linear speedup when using 2 processes, but we obtain much less than linear speedup when using 4 and 8 processes. This is because there is a big communication overhead when re-sampling from the current particles as processes are pulling data from other processes and this communication overhead dominates when using many processes.

Figure 2.6: (a) Applying the Sequential Importance Resampling algorithm to infer the underlying CPAB transformation. The points in blue are the target points and the points in red are the points after applying the inferred transformation. (b) Plot showing the log likelihood and log prior over 200 iterations.

<table>
<thead>
<tr>
<th>Number of processes</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time [sec]</td>
<td>199</td>
<td>105</td>
<td>63</td>
<td>54</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.00x</td>
<td>1.90x</td>
<td>3.06x</td>
<td>3.67x</td>
</tr>
</tbody>
</table>

Table 2.2: Running time of the Sequential Importance Resampling inference algorithm with 1,000 particles for 200 iterations on 256 correspondences on an Intel Core i7-6700K.

In addition to using 16x16=256 correspondences, we experimented with using full 512x512 correspondences. We ran the experiments on a cluster of 4 machines each with a Intel Xeon E5-2670v3 CPU. The timings for them are summarized in Table 2.3. The speedup obtained from parallelization is shown in Figure 2.7. We observe that we obtain almost linear speedup when parallelizing over 96 processes and it did not show signs on slowing down.
Table 2.3: Running time of each iteration of the Sequential Importance Resampling inference algorithm with 1,000 particles on $512 \times 512$ correspondences using distributed computing.

<table>
<thead>
<tr>
<th>Number of processes</th>
<th>1</th>
<th>8</th>
<th>24</th>
<th>48</th>
<th>72</th>
<th>96</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time [sec]</td>
<td>2240</td>
<td>308.5</td>
<td>98.54</td>
<td>54.71</td>
<td>40.54</td>
<td>30.09</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.00x</td>
<td>7.26x</td>
<td>22.7x</td>
<td>40.9x</td>
<td>55.3x</td>
<td>74.4x</td>
</tr>
</tbody>
</table>

Figure 2.7: A plot showing the parallelization speedup obtained when running the Sequential Importance Resampling inference algorithm with 1,000 particles on $512 \times 512$ correspondences.

**2.3 Conclusion**

In this chapter, we implemented the calculation of CPAB transformations in Julia in parallel. We saw that it did not benefit too much from the parallelization as the time to compute the transformation was already quite fast and the overhead in spawning work on the worker processes dominated. The GPU implementation in [8] ran around twice as fast as our parallel implementation. However, although this problem is suited towards
CHAPTER 2. CPAB TRANSFORMATIONS

GPU programming, we note that a Julia parallel implementation is quite straightforward and works on all machines.

We also applied these transformations to solve the correspondence-based image warping inference problem and experimented with 3 different inference algorithms in Julia. We used Optim’s Gradient Descent as well as Conjugate Gradient Descent implementation and compared it with our implementations of Metropolis’ Algorithm and Sequential Importance Resampling. We found that Gradient Descent and Conjugate Gradient Descent tend to get stuck at local minima and both Metropolis’ Algorithm and Sequential Importance Resampling produced better results. While the results of these two algorithms are similar, it should be noted that Sequential Importance Resampling allowed for easy parallelization over the particles instead of over the points and will therefore run faster for bigger problems and machines with more cores. We also saw that we obtained excellent speedup using Sequential Importance Resampling.
In this chapter, we explore the utility of using Julia for Bayesian nonparametric inference algorithms. There is currently a Julia Bayesian nonparametric package BNP.jl [19]. However, this package only includes implementations of serial algorithms which does not scale well with large datasets. We use Julia to implement a Markov chain Monte Carlo (MCMC) sampler for Dirichlet Process Mixture Model (DPMM) proposed by Chang and Fisher III [4] for both Gaussian and Multinomial distributions. That algorithm is based on the split-merge framework by Jain and Neal [14]. It was designed as a parallel algorithm and there is a multi-core C++ implementation available. We refer the reader to some of the negative points about parallel inference algorithms in [9]. This algorithm assumes a shared memory model and we extend the algorithm to accommodate a distributed memory model via an explicit partitioning of the data. This enables our Julia implementation to extend the existing C++ implementation by allowing parallelization across multiple machines. In addition, we also apply this algorithm to the image denoising problem by using our Dirichlet Process Gaussian Mixture Model (DP-GMM) implementation to learn image priors similar to the work by Zoran and Weiss [21]. We note that Hughes and Sudderth [13] also applied DP-GMM to this problem. However, their algorithm is based on variational inference while we use a sampling based
algorithm.

3.1 Background

In this section, we give a brief overview of the DPMM model, the parallel MCMC sampling algorithm that we implemented and our image denoising application. We refer the reader to Section 2.2.1 for some background about parallel computing in Julia in general.

3.1.1 DPMM Model

A DPMM is an infinite-dimensional mixture model which uses a Dirichlet Process as the prior over the clusters parameters. The original proof of the existence of Dirichlet Processes can be found in Ferguson [7]. We refer the readers to [18] for a detailed discussion on Dirichlet Processes. The generative model of a DPMM is as follows:

\[ G \sim DP(\alpha, H), \]  
\[ \tilde{\theta}_i \sim G(\tilde{\theta}_i), \quad \forall i \in \{1, 2, \ldots, N\}, \]  
\[ x_i \sim f_x(x_i; \tilde{\theta}_i), \quad \forall i \in \{1, 2, \ldots, N\}. \]

Here, \( \alpha \) is the concentration parameter and \( H \) is the base measure for the DP. For each point \( i \), we draw a sample \( \tilde{\theta}_i \) from the Dirichlet process realization \( G \). We then draw our \( x_i \) from the distribution parameterized by \( \tilde{\theta}_i \). Because, \( G \) is a realization of a DP, it is a distribution over discrete atoms. Hence, multiple data points will 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 the
stick-breaking process. This alternative generative model is as follows:

\[ \pi \sim \text{GEM}(1, \alpha), \quad (3.4) \]
\[ \theta_k \sim f_\theta(\theta_k; \lambda) = H, \quad \forall k \in \{1, 2, \ldots\}, \quad (3.5) \]
\[ z_i \sim \text{Cat}(\pi), \quad \forall i \in \{1, 2, \ldots, N\}, \quad (3.6) \]
\[ x_i \sim f_x(x_i; \theta_{z_i}), \quad \forall i \in \{1, 2, \ldots, N\}. \quad (3.7) \]

Here, GEM refers to the (Griffiths-Engen-McCloskey) stick-breaking process for a DP. By drawing a sample from \( \text{GEM}(1, \alpha) \), we get an infinite-dimensional vector of weights \( \pi \). For each of the weights \( \pi_k \), 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 \( \pi \). Finally, we sample \( x_i \) from the distribution of cluster \( z_i \) which is parameterized by \( \theta_{z_i} \).

### 3.1.2 Parallel Sampler for DPMMs

Chang and Fisher III [4] proposed a parallel sampling algorithm for DPMMs. The main idea of that algorithm is to have a restricted Gibbs sampler 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, the algorithm uses auxiliary variables such that each cluster is the composed of 2 sub-clusters. These variables are \( \bar{z}_i \in \{l, r\} \) which denotes 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 denotes the weights and parameters of the sub-clusters of cluster \( k \). By sampling the sub-clusters, we are 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 3.1. We use the
same notation as in [4] 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 \( \theta \), \( f_\theta(\theta; \lambda) \) is the likelihood of the parameter \( \theta \) under the prior \( \lambda \), \( \propto \) denotes sampling proportional to the right-hand side of the equation and \( \mathcal{I}_k \) denotes the set of indices of \( i \) where \( z_i = k \).

**Algorithm 3.1** An iteration of DPMM with sub-cluster splits

Run an iteration of restricted Gibbs sampling:

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

\[
(\pi_1, \ldots, \pi_K, \bar{\pi}_{K+1}) \sim \text{Dir}(N_1, \ldots, N_K, \alpha). \tag{3.8}
\]

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

\[
(\bar{\pi}_{kl}, \bar{\pi}_{kr}) \sim \text{Dir}(N_{kl} + \alpha/2, N_{kr} + \alpha/2). \tag{3.9}
\]

(c) Sample cluster parameters \( \theta_k \) for each cluster \( k \):

\[
\theta_k \propto f_x(x_{\mathcal{I}_k}; \theta_k) f_\theta(\theta_k; \lambda). \tag{3.10}
\]

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

\[
\bar{\theta}_{kh} \propto f_x(x_{\mathcal{I}_{kh}}; \theta_{kh}) f_\theta(\bar{\theta}_{kh}; \lambda). \tag{3.11}
\]

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

\[
z_i \propto \sum_{k=1}^{K} \pi_k f_x(x_i; \theta_k) \mathbb{1}(z_i = k). \tag{3.12}
\]

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

\[
\bar{z}_i \propto \sum_{h \in \{l, r\}} \pi_{z_i h} f_x(x_i; \bar{\theta}_{z_i h}) \mathbb{1}(\bar{z}_i = h). \tag{3.13}
\]
Algorithm 3.1 An iteration of DPMM with sub-cluster splits (continued)

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 $\min(1, H)$:

$$H_{\text{split}} = \frac{\alpha}{\Gamma(N_k)f_x(x_{I_k}; \lambda)} \cdot \Gamma(N_{kl})f_x(x_{I_{kl}}; \lambda) \cdot \Gamma(N_{kr})f_x(x_{I_{kr}}; \lambda)$$  \hfill (3.14)

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 $\min(1, H)$.

$$H_{\text{merge}} = 0.01 \frac{\Gamma(N_{k_1} + N_{k_2})}{\alpha \Gamma(N_{k_1}) \Gamma(N_{k_2})} \frac{p(x|\hat{z})}{p(x|z)} \frac{\Gamma(\alpha)}{\Gamma(\alpha + N_{k_1} + N_{k_2})} \frac{\Gamma(\alpha + N_{k_1} + N_{k_2})}{\Gamma(\alpha + N_{k_1}) \Gamma(\alpha + N_{k_2})}$$  \hfill (3.15)

As the algorithm is designed with parallelization in mind, almost all of the operations can be parallelized. In particular, sampling the cluster parameters can be parallelized over the clusters, sampling the assignments can be computed independently for each point and splits can also be proposed in parallel. Hence, a multi-core implementation of this algorithm would be quite straightforward. However, we have to be careful with merges as we do not want to have more than 2 clusters merging into a single cluster. For example, we do not want to merge clusters 1 and 2 along with clusters 2 and 3.

3.1.3 Image Denoising using Expected Patch Log Likelihood

Zoran and Weiss [21] developed a method to perform image denoising by maximizing the Expected Patch Log Likelihood (EPLL) under a learned prior while staying close to the original noisy image. For an image $x$ and prior $p$, the EPLL is defined to be:

$$EPLL_p(x) = \sum_i \log p(P_i x)$$  \hfill (3.16)
where \( \mathbf{P}_i \mathbf{x} \) extracts the \( i \)th patch in image \( \mathbf{x} \) and \( \log p(\mathbf{P}_i \mathbf{x}) \) gives the log likelihood of the patch under the prior \( p \). Note that the sum is over all overlapping patches of a given size across the image. Since we also would not like to deviate too much from the original noisy image \( \mathbf{y} \), the proposed cost function is:

\[
f_p(\mathbf{x}|\mathbf{y}) = \frac{\lambda}{2} ||\mathbf{x} - \mathbf{y}||^2 - EPLL_p(\mathbf{x}).
\] (3.17)

Here, \( \lambda \) is a parameter related to the variance of the noise (\( \approx \frac{1}{\sigma^2} \)).

To optimize the cost function given in Equation (3.17), an optimization method called “Half Quadratic Splitting” [10] is used. In this method, we have an auxiliary variable \( \mathbf{z}^i \) for every patch \( \mathbf{P}_i \mathbf{x} \) and instead optimize the following cost function with respect to both \( \mathbf{x} \) and \( \mathbf{z} \):

\[
c_{p,\beta}(\mathbf{x}, \mathbf{z}|\mathbf{y}) = \frac{\lambda}{2} ||\mathbf{x} - \mathbf{y}||^2 + \sum_i \left( \frac{\beta}{2} ||\mathbf{P}_i \mathbf{x} - \mathbf{z}^i||^2 - \log p(\mathbf{z}^i) \right).
\] (3.18)

We see that for \( \beta \to \infty \), maximizing Equation (3.18) is equivalent to maximizing Equation (3.17). For a given \( \beta \), we are able to optimize Equation (3.18) iteratively by alternating between optimizing with respect to \( \mathbf{x} \) and with respect to \( \mathbf{z} \). We do this for a large \( \beta \) in order to optimize the cost function given in 3.17. For more details, we refer the reader to [21].

### 3.2 Implementation

We implemented a sampler with multi-machine parallelization for the case when each cluster is a Gaussian i.e. a Dirichlet Process Gaussian Mixture Model (DP-GMM) in Julia as well as when each cluster is Multinomial i.e. a Dirichlet Process Multinomial Mixture Model (DP-MNMM). We discuss key implementation details and challenges below. Assume we have \( N \) data points, \( K \) clusters and \( M \) machines each with \( P \) cores.

Our code is available at [https://github.com/angel8yu/Subcluster-DPMM.jl](https://github.com/angel8yu/Subcluster-DPMM.jl).
In order to achieve multi-machine parallelization in Julia, we use the DistributedArray object to store our data. In our implementation, we make use of sufficient statistics [17] in our sampler. The sufficient statistics for a Gaussian cluster with points $x_1, x_2, \ldots, x_m \in \mathbb{R}^d$ are:

$$T^G_1 = \sum_{i=1}^{m} x_i,$$

$$T^G_2 = \sum_{i=1}^{m} x_i^T x_i. \quad (3.19)$$

While the sufficient statistics for a Multinomial cluster with points $x_1, x_2, \ldots, x_m$ are:

$$T^{MN}_1 = \sum_{i=1}^{m} x_i. \quad (3.21)$$

Note that in the Gaussian case, the points $x_i \in \mathbb{R}^d$ while in the Multinomial case, the points $x_i \in \mathbb{N}_0^d$ is a vector of counts. Here, $d$ is the dimensionality of the points.

By aggregating sufficient statistics for each cluster, we are able to sample the cluster parameters. Because the sufficient statistic is a sum over the 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 the concept of sufficient statistics in general, and in the exponential family in particular, please see [17] and [3].

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 $K$ is generally small compared to the number of data points. Sampling the cluster and sub-cluster assignments takes $O(NK)$ time as can be seen from Equations (3.12) and (3.13). 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 then maintain the cluster and sub-cluster parameters in the
Figure 3.1: The proposed architecture of our proposed multi-machine implementation. We show a parallelization over 4 machines here.

master machine along with their sufficient statistics. They are then broadcasted to all processes so we are able to sample the assignments. Proposing splits is 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 3.1.

### 3.2.2 Detailed Implementation Design

We maintain the following **SharedArray** objects on the master machine:

- Cluster and sub-cluster weights
  
  $\pi_k, \bar{\pi}_{kl}, \bar{\pi}_{kr}$

- Cluster parameters
  
  $\theta_k$ (e.g. $\mu_k$ (mean) and $\Sigma_k$ (covariance) for Gaussian)

- Sub-cluster parameters
  
  $\theta_{kl}, \theta_{kr}$
• Cluster sufficient statistics

\[ N_k \text{ (number of points), } T_k \text{ (sufficient statistics)} \]

• Sub-cluster sufficient statistics

\[ N_{kl}, N_{kr} \text{ (number of points), } T_{kl}, T_{kr} \]

In addition, we also maintain an array `cluster_index` on the master process mapping the current clusters to indices in each of the `SharedArray` objects. This is because the size of `SharedArray` objects cannot be changed after creation so we would need to pre-allocate a larger array and then fill it up. Once it is filled, we will allocate a new larger array and migrate from the old array to new array. This is similar to table doubling in hash table implementations [5].

We maintain the following objects on every worker process:

• Cluster and sub-cluster weights (broadcasted from master)

• Cluster and sub-cluster parameters (broadcasted from master)

• Chunk of data (as part of a `DistributedArray`)

\[ x_i \]

• Chunk of cluster and sub-cluster assignments (as part of a `DistributedArray`)

\[ z_i, \bar{z}_i \]

We now describe our implementation which follows the steps in Algorithm 3.1.

1. Run an iteration of restricted Gibbs sampling:

   (a) Sample the cluster weights using Equation (3.8) on the master process.

   (b) Sample the sub-cluster weights using Equation (3.9) on the master process.
(c) Sample the cluster parameters using Equation (3.10) on worker processes on the master machine in parallel.

(d) Sample the sub-cluster parameters using Equation (3.11) on worker processes on the master machine in parallel.

(e) Broadcast cluster and sub-cluster weights and parameters to all worker processes.

(f) Sample the cluster assignments using Equation (3.12) in parallel across all worker processes. Each worker is responsible for sampling the assignments of the data it is in charge of updating the assignments in the DistributedArray.

(g) Sample the sub-cluster assignments using Equation (3.13) in parallel across all worker processes. Each worker is responsible for sampling the assignments of the data it is in charge of updating the assignments in the DistributedArray.

(h) Update the cluster and sub-cluster sufficient statistics in parallel across all worker processes. Each worker is responsible for calculating the sufficient statistics of all the clusters for the data it is in charge of. We then aggregate the sufficient statistics across all workers by simply summing them. For example, if we have 4 workers and we wanted to calculate the sufficient statistics $T$ for cluster 1, we would first calculate the sufficient statistics for each of 4 workers $T^i$ on the data points it is in charge of that is assigned to cluster 1 and sum them as follows:

$$T = T^1 + T^2 + T^3 + T^4$$

(3.22)

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 the sufficient statistics received as this would allow us to undo the previous contribution to the aggregated statistics.
2. Splits:

(a) Propose to split cluster each cluster in parallel on the master machine. The calculation of the Hastings ratio using Equation (3.14) only requires the sufficient statistics which are available via a SharedArray on the master.

(b) Process all the accepted splits by creating an assignment mapping which maps old assignments to new assignments. This is then broadcasted to all worker processes which updates the assignments of all the points it is responsible for. The sufficient statistics for the newly-split clusters are also calculated and updated in the same way as shown in Equation (3.22).

3. Merges:

(a) Propose to merge all pairs of clusters in serial on the master process by using Equation (3.15). As in [4], when a merge between 2 clusters has been accepted, we do not consider them for future merges in this iteration.

(b) Process all the accepted merges by creating an assignment mapping which maps old assignments to new assignments. This is then broadcasted to all worker processes and each of which updates the assignments of all the points it is responsible for. The sufficient statistics for the newly merged clusters are also calculated and updated in the same way as shown in Equation (3.22).

3.2.3 Optimizing Processes Communication

In each iteration, we have to 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 to send the objects from the master process to each of the workers 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 (be-
tween processes on different machines). To reduce this communication overhead in
our implementation, we first send the objects to be broadcasted from the master to 1
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. This reduced our communication overhead tenfold.

In theory, we could have further reduced communication overhead by sending the
objects to be broadcasted from the master to 1 process in each machine and then have
that process update a SharedArray object which will be accessible to all processes.
However, as we were trying to implement this, we found that slices of SharedArray
objects did not support matrix multiplication in the newest official release of Julia at the
time of writing (v0.4.6). This prevented us from adopting this method of broadcasting
as we needed matrix multiplication when calculating the probabilities used for sampling.
We filed an issue ticket at https://github.com/JuliaLang/julia/issues/17100 and
found that it would be supported in the new v0.5 version which is under development
at the time of writing.

3.2.4 Runtime Complexity

In Section 3.2.1, we touched on the runtime complexity of our implementation. We now
analyze the total runtime complexity of our parallel implementation.

1. Run an iteration of restricted Gibbs sampling:

   (a) Sampling the cluster weights takes constant time.

   (b) Sampling sub-cluster weights takes constant time for each cluster and is not
       parallelized. This takes $O(K)$ time.

   (c) Sampling the cluster and sub-cluster parameters also take constant time for
each cluster but are parallelized over $P$ processes on the master. This takes $O(K/P)$ time.

(d) 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.

(e) Sampling the cluster assignments takes $O(NK)$ time in serial, but this is parallelized over $MP$ processes so this takes $O(NK/(MP))$.

(f) Sampling the sub-cluster assignments takes $O(N)$ time in serial, but this is parallelized over $MP$ processes so this takes $O(N/(MP))$.

(g) 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.

2. Splits:

(a) Proposing splits by looking at each cluster takes $O(K)$ time.

(b) Processing all the accepted splits requires updating the sufficient statistics which could take worst case $O(N/(MP)) + O(M + P)$ if all clusters are split.

3. Merges:

(a) Proposing merges by looking at each pair of clusters takes $O(K^2)$ time.

(b) Processing all the accepted merges also requires updating the sufficient statistics which could take worst case $O(N/(MP)) + O(M + P)$ if all clusters are merged.
Hence, we see a total runtime complexity of:

\[ O(K) + O(M + P) + O(NK/(MP)) \]  

(3.23)

Since \( N \gg K, p, M \), this implementation achieves linear parallelization theoretically.

### 3.2.5 Memory Complexity

The benefit of a multi-machine implementation goes beyond the gains from a faster runtime as it also allows computation over larger datasets that cannot fit into the memory of a single machine. With our multi-machine architecture, the bulk of the data is split evenly across all machines.

We now look at the amount of memory our implementation needs on each machine. The data is stored as a distributed array across all processes, so we have \( O(N/M) \) 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 compared to the data itself.

### 3.3 Results

We compared our implementation with the implementation from [4] by running them on synthetic data. We also applied our implementation to the real-data problem of image denoising, where we first learned a DP-GMM prior over images patches and then used that prior within a standard method of patch-based image denoising. Our setting is akin to the one described in [21]; note, however, that their prior is a finite (and non-Bayesian) Gaussian Mixture Model (GMM) where \( K \), the number of Gaussians, is fixed manually. Also, as we will show, our distributed implementation easily handles larger training datasets than theirs.
3.3.1 Synthetic Data: DP-GMM

We first show the results of applying our implementation on large synthetic datasets. We generated synthetic data for our DP-GMM implementation by sampling $10^6$ points from an underlying GMM with 6 clusters. For illustration, we start with 2-dimensional points but note that our implementation works for arbitrary dimensions. Figure 3.2a shows the location of the points in the 2-dimensional case. After running our implementation on this synthetic data, we obtain the results shown in Figure 3.2b. We obtained similar results when running the implementation from [4].

We compare the runtime of our implementation with the implementation from [4] in Table 3.1. They were run on machines with Intel Xeon E5-2670 v3 processors. Recall that the implementation from [4] does not support multiple machines. We limit the number of cores in each machine to 8 in order to better show the speedups we get by parallelizing over multiple machines for this size of data. As we can see, our Julia implementation is consistently around 14 times slower on a single machine. As we increase the dimensionality of our data set, our implementation performs relatively better. For example, if we increase the dimensionality to 30, we see from Table 3.2 that our implementation is almost twice as fast on a single machine and we obtain additional good speedup on multiple machines. This leads us to believe that their implementation is optimized for low dimensions. However, we have also noted (empirically) that the implementation from [4] fails when we go past a certain dimension (while our implementation still succeeds). For example, when we set the dimensionality to be 250, their implementation converged to 1 cluster while ours converges to the correct 6 clusters. Attempts to debug their implementation have led us to suspect the problem is caused by underflow/overflow errors.
Figure 3.2: (a) The $10^6$ points in our synthetic data, in the 2-dimensional case, sampled from an underlying GMM with 6 clusters are shown in blue. (b) The inferred clusters from our implementation are shown as ellipses at 5 standard deviations in different colors.

<table>
<thead>
<tr>
<th>Cores × Machines</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>8×2</th>
<th>8×3</th>
<th>8×4</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++ [4]</td>
<td>76.94</td>
<td>40.57</td>
<td>22.23</td>
<td>13.01</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Julia</td>
<td>1101.97</td>
<td>572.50</td>
<td>345.58</td>
<td>172.30</td>
<td>107.58</td>
<td>80.10</td>
<td>63.55</td>
</tr>
</tbody>
</table>

Table 3.1: Time(sec) to run 100 DP-GMM iterations of $d = 2, N = 10^6, K = 6$. 
Table 3.2: Time (sec) to run 100 DP-GMM iterations of $d = 30$, $N = 10^6$, $K = 6$.

<table>
<thead>
<tr>
<th>Cores × Machines</th>
<th>8</th>
<th>8×2</th>
<th>8×3</th>
<th>8×4</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++ [4]</td>
<td>798.94</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Julia</td>
<td>398.67</td>
<td>218.42</td>
<td>146.71</td>
<td>124.55</td>
</tr>
</tbody>
</table>

### 3.3.2 Synthetic Data: DP-MNMM

We generated synthetic data for our DP-MNMM implementation by sampling $10^6$ points with dimension 100 from an underlying MNMM with 6 clusters. The results we obtained from running our DP-MNMM are similar to the implementation from [4]. We compare the runtime of our implementation with the implementation from [4] in Table 3.3. Again, they were run on machines with Intel Xeon E5-2670 v3 processors. As we can see, our Julia implementation is around 2 times slower on a single machine. By looking at the C++ implementation from [4], we see that they used sparse arrays for the DP-MNMM model which most likely saves computation time as well as requires lower memory allocations. We also tried changing the dimensionality but did not find that it made any difference to the relative performance of our implementation unlike in the DP-GMM case. The relative performance of our DP-MNMM implementation seems to be consistent with the relative performance of Julia to C/C++ given in http://julialang.org/benchmarks/. We also note that since running our implementation on our generated synthetic data is fast, we do not obtain any significant speedup after 2 machines using 8 cores each. If we increase the number of underlying clusters to 60 which increases the runtime, we are able to see a better speedup as shown in Table 3.4. This is due to the parallelization overhead dominating the gains from having extra cores.
3.3.3 Image Denoising

We apply this implementation to image denoising using image patches. Zoran and Weiss [21] used a GMM with 200 clusters on this problem by training on a sample of 2 million $8 \times 8$ image patches from BSDS500 [1]. They set the number 200 manually, and their formulation of the GMM learning was not Bayesian. We use the same denoising method but instead use our DP-GMM implementation to learn the prior over the patches. The eigenvalues of 6 randomly-chosen clusters are shown in Figure 3.3. By inspection, they are similar to the eigenvalues learned by the GMM in [21].

![Figure 3.3: Eigenvalues of 6 randomly-chosen clusters learned by our DP-GMM on 2 million $8 \times 8$ image patches.](image-url)
Our DP-GMM implementation took less than an hour to train when parallelized over 4 machines each with an Intel Xeon E5-2670v3 processor. We experimented with different $\alpha$ parameters in our model and applied our trained prior to the image denoising using the MATLAB code provided in [21]. Because we were able to run 2 million patches with ease, we also ran all of 44,650,800 the overlapping $8 \times 8$ patches in the BSDS500 dataset. This took our implementation around 2 days. We believe that learning a model from such a large training set was not possible to do using the serial, non-distributed, implementation from [21]; however, this is just a speculation since Zoran and Weiss’s released code for image denoising did not include the code for the GMM learning. We compared the priors learned by our DP-GMM implementation to the prior provided by [21] which was trained using a GMM with 200 clusters by performing image denoising on the 50 test images in the dataset. The results are summarized in Table 3.5. We show an example of denoising using our model in Figure 3.4.

<table>
<thead>
<tr>
<th>Prior</th>
<th>Num. Clusters</th>
<th>Avg. PSNR</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMM [21]</td>
<td>200 (fixed manually)</td>
<td>28.5061</td>
</tr>
<tr>
<td>DP-GMM ($\alpha = 1$)</td>
<td>72 (inferred)</td>
<td>28.4173</td>
</tr>
<tr>
<td>DP-GMM ($\alpha = 10^3$)</td>
<td>80 (inferred)</td>
<td>28.4276</td>
</tr>
<tr>
<td>DP-GMM ($\alpha = 10^5$)</td>
<td>99 (inferred)</td>
<td>28.4303</td>
</tr>
<tr>
<td>DP-GMM ($\alpha = 10^6$, all patches)</td>
<td>330 (inferred)</td>
<td><strong>28.5402</strong></td>
</tr>
</tbody>
</table>

Table 3.5: Average PSNR obtained from denoising 50 test images with added Gaussian noise using different priors.

### 3.4 Conclusion

In this chapter, we implemented a parallel MCMC sampling algorithm for DPMMs proposed by Chang and Fisher III [4] in Julia. We extended their algorithm to support a distributed memory model to allow for parallelization over multiple machines. Like their existing C++ implementation, our Julia implementation supported both DP-GMM and
Figure 3.4: An example of denoising using our model trained on all $8 \times 8$ patches. The corrupted image (middle) has a PSNR of 20.17 and the restored image (right) has a PSNR of 29.14.

DP-MNMM models. We compared our distributed Julia implementation with their implementation and found that on a single machine our implementation was generally slower with some exceptions. However, their DP-GMM implementation seem to be optimized for lower dimensions and we do better on higher dimensions. In any case, the point of re-implementing their C++ code in Julia was not, of course, in order to a speedup on a single machine. Rather, we wanted to have an implementation that 1) lets us easily distribute the inference over multiple multi-core machines (the implementation from [4] supports only a single multi-core machine); 2) is written in a higher-level language - so development is easy - and that is still relatively fast. Judging by these two criteria, we can report that our experience with Julia was positive. We also applied our DP-GMM implementation on image denoising using learned image patch priors [21]. We reported slightly worse results than [21] when using just a sample of 2 million patches as data for our model. However, we were able to handle all 44 million $8 \times 8$ patches with our distributed implementation and obtained slightly better results.
Chapter 4

Projection Dirichlet Process
Gaussian Mixture Model

A Dirichlet Process Gaussian Mixture models (DP-GMM) is a widely-used Bayesian nonparametric extension of the classic Gaussian Mixture Model (GMM). In the previous chapter, we implemented an effective parallel sampling method for DP-GMM inference proposed by Chang and Fisher III [4]. A standard assumption in DP-GMM inference, [4] included, is that the draws from the DP-GMM, denoted by \( \{x_i\}_{i=1}^{N} \subset \mathbb{R}^{D} \), are directly observed. There are cases, however, where this assumption does not hold. Consequently, Chang and Fisher III’s method (similar to other DP-GMM inference methods) is not directly applicable.

In this chapter, we present a model which relaxes the assumption above. Our model, called the Projection Dirichlet Process Gaussian Mixture Model (PDP-GMM) instead assumes that the observations, denoted by \( \{y_i\}_{i=1}^{N} \) are merely affine projections of the DP-GMM draws. We also show how Chang and Fisher III’s method as well as our implementation in the previous chapter can be adapted to handle this model. We then apply this model to solve a few seemingly-unrelated problems.
4.1 Model and Inference

In our model, we relax the assumption that we directly observe the draws from the DP-GMM and instead assume that only affine projections of the draws are observed. In particular, we have for $i = 1, 2, \ldots, N$:

$$y_i = h_i x_i + c_i, \quad y_i \in \mathbb{R}^d, \quad x_i \in \mathbb{R}^D, \quad h_i \in \mathbb{R}^{d \times D}, \quad c_i \in \mathbb{R}^d, \quad d < D. \tag{4.1}$$

Here, $\{x_i\}_{i=1}^N \subset \mathbb{R}^D$ is the set of draws from the DP-GMM, $\{y_i\}_{i=1}^N \subset \mathbb{R}^d$ is the set of observations and $h_i$'s and $c_i$'s are the parameters of the affine projection which are assumed to be known. Since the $c_i$'s are known, they can be absorbed – by subtraction – into the $y_i$'s. Hence, we may assume, without loss of generality, that the projections are linear:

$$y_i = h_i x_i, \quad y_i \in \mathbb{R}^d, \quad x_i \in \mathbb{R}^D, \quad h_i \in \mathbb{R}^{d \times D} \quad d < D. \tag{4.2}$$

Note that if $d \geq D$ and all $h_i$'s are invertible, this is equivalent to the standard DP-GMM model. Hence, we focus on the case when $d < D$. Let $K$ denote the number of Gaussian clusters. As our Bayesian model is nonparametric, $K$ will be inferred together with the other parameters of the mixture model. We consider first the case where $K = 1$ and we have a single Gaussian with a $D$-dimensional mean, $\mu$, and a $D$-by-$D$ covariance, $\Sigma$:

$$x_i | \mu, \Sigma \sim \mathcal{N}(\mu, \Sigma) \implies y_i | \mu, \Sigma \sim \mathcal{N}(h_i \mu, h_i \Sigma h_i^T). \tag{4.3}$$

Let $y = \{y_i\}_{i=1}^N$. The likelihood is given by:

$$p(y | \mu, \Sigma) = (2\pi)^{-\frac{dN}{2}} \left( \prod_{i=1}^N \left| h_i \Sigma h_i^T \right|^{-\frac{1}{2}} \right) \exp \left( -\frac{1}{2} \left[ \sum_{i=1}^N (y_i - h_i \mu)^T (h_i \Sigma h_i^T)^{-1} (y_i - h_i \mu) \right] \right). \tag{4.4}$$
4.1.1 Conjugacy for Isotropic Covariances

Zheng [20] worked out the following details for the isotropic covariance case. If $\Sigma$ is isotropic, we can write $\Sigma = c^{-1}I$ where $c \in \mathbb{R}^+$. Substituting this into Equation (4.4),

$$p(y|\mu, c) = (2\pi)^{-dN/2} c^{DN/2} \left( \prod_{i=1}^{N} |h_i h_i^T|^{-\frac{1}{2}} \right) \exp \left( -\frac{c}{2} \sum_{i=1}^{N} (y_i - h_i \mu)^T (h_i h_i^T)^{-1} (y_i - h_i \mu) \right).$$

(4.5)

We show in Appendix A that the following distribution is a conjugate distribution on $(\mu, c)$:

$$p(\mu, c) = p(\mu|c)p(c) = \mathcal{N}(\mu; \eta, (cM)^{-1}) \times \text{Gamma}(c; \alpha, \beta).$$

(4.6)

The hyperparameters, suppressed in the notation $p(\mu|c)p(c)$, are:

$$\alpha > 0; \quad \beta > 0; \quad \eta \in \mathbb{R}^D; \quad M \in \text{SPD}(D).$$

(4.7)

Here, SPD($D$) stands for $D$-by-$D$ symmetric positive-definite matrices. Note that $M$ need not be isotropic. We define the following quantities:

$$\tilde{T}_1 = \sum_{i=1}^{N} h_i^T (h_i h_i^T)^{-1} y_i, \quad \tilde{T}_2 = \sum_{i=1}^{N} h_i^T (h_i h_i^T)^{-1} h_i,$$

$$\tilde{T}_3 = \sum_{i=1}^{N} y_i^T (h_i h_i^T)^{-1} y_i, \quad \tilde{T}_4 = \prod_{i=1}^{N} |h_i h_i^T|^{-\frac{1}{2}}.$$

(4.8)

After some algebra (see Appendix A), the conjugacy yields the following closed-form expressions:

$$p(y|\mu, c) \propto c^{\frac{DN}{2}} \tilde{T}_4 \exp \left( -\frac{c}{2} \left[ \tilde{T}_3 + \mu^T \tilde{T}_2 \mu - 2 \mu^T \tilde{T}_1 \right] \right),$$

(4.9)

$$p(\mu, c|y) = p(\mu|y, c)p(c|y) = \mathcal{N}(\mu; \eta', (cM')^{-1}) \times \text{Gamma}(c; \alpha', \beta'),$$

(4.10)

$$p(y) = (2\pi)^{-dN/2} \tilde{T}_4 |M|^{-\frac{1}{2}} |M'|^{-\frac{1}{2}} \frac{\beta' \Gamma(\alpha')}{\beta \Gamma(\alpha)}.$$
\[ \alpha' = \alpha + \frac{DN}{2}, \quad M' = M + \tilde{T}_2, \quad \eta' = M'^{-1}(M\eta + \tilde{T}_1), \]
\[ \beta' = \beta + \frac{1}{2} \left( \tilde{T}_3 + \eta^T M \eta - \eta'^T M' \eta' \right). \]  
(4.12)

\section*{4.1.2 Anisotropic Covariance}

In the anisotropic case, it can be shown that there is no conjugate prior for \( \Sigma \) when \( d < D \). Particularly, the natural choice of a Normal Inverse-Wishart (NIW) prior for \((\mu, \Sigma)\) does not lead to an NIW posterior over these parameters. Although it is possible to modify the algorithm to handle the difficulty of a non-conjugate prior, we focus on the isotropic case for simplicity.

\section*{4.2 Implementation}

We extended our implementation from Chapter 3 to support the PDP-GMM model. We made use of Julia’s native multiple dispatch feature which uses the all of a function’s arguments to decide which method to invoke. In particular, our core DPMM code takes different models and each model has its own types. This allows for the core code to be shared across different models which include the DP-GMM, the DP-MNMM as well as the PDP-GMM. Adding an additional model is straightforward as only a minimal amount of code needs to be written which include defining new types and providing methods to calculate various model specific quantities. In particular, the quantities we need are given in Equations (4.8)–(4.12). Our code is available at https://github.com/angel8yu/Subcluster-DPMM.jl.

\section*{4.3 Results}

We now show our results when we apply this model to a few seemingly-unrelated problems.
4.3.1 Tomography

A natural application for this model would be tomography. In tomography, the observations lie in a low dimensional space and we would like to reconstruct the original data in the higher dimensional space. Here we show an example of our model applied in a tomography setting. Assume we have 2-dimensional draws from a DP-GMM which are not observed. Instead, we observe the projections of these draws on lines through the origin. Every draw is projected onto each of these lines. The setting is shown in Figure 4.1. Without loss of generality, and to simplify the explanation, assume that projections are taken at evenly-spaced angles; e.g., every, say, 1 or 10 degrees. For visualization, at each of these angles we take its corresponding 1-dimensional measurements and display (in Figure 4.1) their associated 1D probability density function, estimated using a standard Kernel Density Estimator (KDE). Note these estimated 1D marginals are used only for visualization. Particularly, our inference algorithm does not use these marginals (and thus errors in the KDE do not propagate into our results). In fact, our algorithm can also handle the case where each angle is associated with even just a single observation (in which case, the use of KDE would not make sense anyway).

Using our inference algorithm, we are able to infer the latent parameters of the model. We show select results in Figure 4.2. Note that although there are (latent) correspondences, we do not need to estimate them during our inference. In other words, our algorithm utilizes implicit observation-to-cluster correspondences as opposed to the more error-prone and computationally intensive observation-to-observation correspondences. As we can see, the inferred parameters explain the unobserved draws well. Figure 4.3 shows that this remains the case even when the number of (true) clusters increases.
We now use our model to fit a mixture of polynomial regressions. This particular application has been studied before by Faria and Soromenho [6] as well as in Huang and Li [12]. Both of their models are finite mixture models while our model is an infinite-dimensional mixture model. In terms of the type of regression, Faria and Soromenho focused their work on linear regressions while Huang and Li learn nonparametric regressions. Our model is able to handle finite linear combinations of basis functions, but we focus on mixtures of polynomial regressions here. To clarify the terminology, the word 'finite' in the previous sentence refers to the dimensionality of the regression functions (e.g., $D = 2$ for $\mathbb{R} \rightarrow \mathbb{R}$ linear regression functions as there is one degree of freedom for the slope and one for the intercept). However, our model is infinite-dimensional in the sense that we entertain the notion that $K$, the (latent) number of these functions (e.g., $K$ may stand for the number of linear regression functions) can be infinite.

We first start off with fitting a mixture of linear regressions. Note that we take a
Figure 4.2: A simple example with 10,000 points and 3 clusters. We show the inferred cluster means and (isotropic) covariances as circles at 4 standard deviations together with the estimated 1-dimensional marginals and the unobserved 2-dimensional draws.

slightly different but more general view than in the classical setting where we have a few lines that generate points with noise being in the $y$-dimension. Instead, each line generates a single point and we cluster the lines. This allows for noise in both the slope and the intercept.

Assume we have a mixture of linear regressions with $(p_i^0, p_i^1)$ being the coefficients of the $i$th linear regression:

$$y = p_i^1 x + p_i^0.$$  

(4.13)
Figure 4.3: A more complicated example with 10,000 points and 10 clusters. (Left) We show the inferred cluster means and (isotropic) covariances as circles at 3 standard deviations together with the estimated 1-dimensional marginals and the unobserved 2-dimensional draws. (Right) We show the weights of each inferred cluster.

We then observe a point \((a_i, b_i) \in \mathbb{R}^2\) on this line and we have:

\[
b_i = p_i^1 a_i + p_i^0
\]

\[
= \begin{bmatrix} 1 & a_i \end{bmatrix} \begin{bmatrix} p_i^0 \\ p_i^1 \end{bmatrix}
\]  \hspace{1cm} (4.14)

Using the notation in Section 4.1, we set:

\[
y_i = b_i, \quad x_i = \begin{bmatrix} p_i^0 \\ p_i^1 \end{bmatrix}, \quad h_i = \begin{bmatrix} 1 & a_i \end{bmatrix}.
\]  \hspace{1cm} (4.16)

This allows us to use our model to cluster the parameters of the linear regressions \(x_i\).

For example, in Figure 4.4, we have points generated from the lines \(y = 3x + 1\), \(y = 3x + 3\) and \(y = 0\). We use our model to fit a mixture of linear regressions and the results are shown in Figure 4.4. As we can see, the inferred cluster centers fit the data and are close to the ground truth lines.

Similarly, we are able to use our model to fit polynomial regressions in general. As an example, we show some results of fitting a mixture of quadratic regressions. Assume
we have a mixture of quadratic regressions with \((p_i^0, p_i^1, p_i^2)\) being the coefficients of the \(i\)th quadratic regression:

\[
y = p_i^2 x^2 + p_i^1 x + p_i^0.
\]

(4.17)

We then observe a point \((a_i, b_i) \in \mathbb{R}^2\) on this quadratic and we can set:

\[
y_i = b_i, \quad x_i = \begin{bmatrix} \begin{bmatrix} p_i^0 \\ p_i^1 \\ p_i^2 \end{bmatrix} \\ a_i \end{bmatrix}, \quad h_i = \begin{bmatrix} 1 & a_i & a_i^2 \end{bmatrix}.
\]

(4.18)

For example, in Figure 4.5, we have points generated from the quadratics \(y = x^2 + 3x + 30\), \(y = \frac{1}{2}x^2 + 6x\) and \(y = -x^2 + 60\). We use our model to fit a mixture of quadratic regressions and the results are shown in Figure 4.5. As we can see, the inferred cluster centers fit the data and are close to the ground truth quadratics.
Figure 4.5: An example of using our model to fit a mixture of quadratic regressions. (Left) We show the observations. (Right) We show the resulting clusters after inference. The black curves are the ground truth quadratics. The colored curves are cluster centers inferred using our model. Importantly, note that we did not assume that we know that the number of the regression functions, $K$, is three. Rather, this quantity was estimated as part of our inference procedure.

4.4 Conclusion

In this chapter, we presented a model PDP-GMM which relaxes the assumption that the draws from a DP-GMM are directly observed. Instead our model assumes that only known affine projections of the draws are observed. This allows us to handle a wide range of problems. As discussed in Section 4.2, we implemented this as an extension of our DP-GMM code from Chapter 3 making extensive use of Julia’s native multiple dispatch feature which allowed us to reuse most of the core DP-GMM code. We then presented results of this model applied to a number of synthetic examples and saw that it performed well. In particular, the results suggest that this model is suitable for tomography applications as well as fitting a possibly-infinite mixture of polynomial regressions.
Chapter 5

Conclusion

In this thesis, we have focused on exploring the utility of using Julia for parallel and distributed MCMC inference algorithms. In addition, we have also proposed a new model which relaxes the standard DP-GMM assumption that the draws are directly observed.

In Chapter 2, we implemented the calculation of CPAB transformations [8] in Julia in parallel. We compared it with an existing GPU implementation given in [8] and found it was around twice as fast as our CPU implementation. This is a good result considering that the problem is well suited towards GPU computation and that a CPU implementation works on virtually all machines. We then applied CPAB transformations to the correspondence-based image warping problem. After trying Gradient Descent, Metropolis’ Algorithm and Sequential Importance Resampling, we found that Gradient Descent tends to get stuck in local minima and both Metropolis’ Algorithm and Sequential Importance Resampling produced better results. However, the Sequential Importance Resampling approach led to better parallelization speedups.

In Chapter 3, we implemented a parallel MCMC sampler for DPMMs proposed by Chang and Fisher III [4] in a distributed fashion in Julia. We extended their algorithm by distributing the data and computing on sufficient statistics. This allowed us to parallelize over multiple machines. We compared our implementation with their C++ implementation and found that although our Julia implementation is generally slower
on a single machine, we also showed that in several settings our implementation is the faster one and that it also handles higher dimensions more gracefully in terms of the quality of the results. In addition, our distributed architecture also allows for the ability to use multiple machines to speed up runtime. Using our distributed Julia implementation, we applied the DP-GMM model to the real-data problem of image denoising using learned image patches priors. Previous works were trained on a sample of 2 million patches while we were able to train our DP-GMM model on all 44 million patches using this implementation. We reported better results than using a sample of 2 million patches and slightly better results than the original work by Zoran and Weiss [21].

In Chapter 4, we presented a model PDP-GMM that extends the DP-GMM model by relaxing the assumption that the draws are directly observed. Instead we assume that only known affine projections of the draws are observed. In addition, we showed that there is a conjugate prior when we restrict covariances to be isotropic. This was implemented on top of the DPMM code in Julia and we noted that with Julia’s native multiple dispatch feature, we were able to add this extension with a minimal amount of additional code. Furthermore, we saw that the model performed well when applied to synthetic examples of tomography and fitting a mixture of polynomial regressions.

5.1 Remarks on Julia

Overall, our experience with Julia has been positive. The high level nature of the language makes development easy and straightforward. In addition to having a syntax similar to other technical computing languages, it does not take long to pick up the language. Furthermore, the parallel abstractions in Julia allow for a straightforward implementation of parallel algorithms while maintaining a granular level of control for data storage and communication. Since it is also designed for high performance com-
puting with performance similar to low level languages such as C/C++, Julia surfaces as one of the top programming languages for technical computing.

Being a relatively new language, there are still a few nice-to-have features that are currently under development. This includes support for native multithreading as well as work stealing. As we saw in Chapters 2 and 3, our Julia implementations were able to be competitive with a GPU implementation as well as a C++ implementation. Although slower than low level languages, we think that the ease of development in Julia outweighs the performance gains achieved in low level languages such as C/C++.
In this appendix, we derive expressions related to Chapter 4. We note that the derivations in this appendix were derived by Zheng [20]. In particular, we compute the likelihood, posterior parameters as well as the data likelihood of our PDP-GMM model.

A.1 Likelihood

We show that Equation (4.10) follows from Equation (4.5) using the property that $h_i h_i^T$ is symmetric:

$$p(y|\mu, c) \propto c \frac{D_N}{2} \left(\prod_{i=1}^{N} |h_i h_i^T|^{-\frac{1}{2}}\right) \exp \left(-\frac{c}{2} \left[\sum_{i=1}^{N} (y_i - h_i \mu)^T (h_i h_i^T)^{-1} (y_i - h_i \mu)\right]\right)$$

(A.1)

$$= c \frac{D_N}{2} \tilde{T}_4 \exp \left(-\frac{c}{2} \left[\sum_{i=1}^{N} y_i^T (h_i h_i^T)^{-1} y_i - \mu^T h_i^T (h_i h_i^T)^{-1} h_i \mu\right]\right) - y_i^T (h_i h_i^T)^{-1} h_i \mu + \mu^T h_i^T (h_i h_i^T)^{-1} h_i \mu\right)$$

(A.2)

$$= c \frac{D_N}{2} \tilde{T}_4 \exp \left(-\frac{c}{2} \left[\tilde{T}_3 - 2\mu^T \tilde{T}_1 + \mu^T \tilde{T}_2 \mu\right]\right).$$

(A.3)

A.2 Posterior Parameters

We show that the distribution on $(\mu, c)$ in Equation (4.6) is a conjugate prior of the distribution given in Equation (4.10) by calculating the posterior parameters. We calculate the posterior of $\mu$:

$$p(\mu|y, c) = \frac{p(y|\mu, c)p(\mu|c)}{p(y|c)}$$

(A.4)

$$\propto p(y|\mu, c)p(\mu|c)$$

(A.5)
\[ \propto c^{\frac{\beta}{2}} \tilde{T}_4 \exp \left( -\frac{c}{2} \left[ \tilde{T}_3 - 2\mu^T \tilde{T}_1 + \mu^T \tilde{T}_2 \mu \right] \right) \]
\[ \cdot |cM|^{\frac{1}{2}} \exp \left( -\frac{1}{2} (\mu - \eta)^T cM (\mu - \eta) \right) \]  

(A.6)

\[ \propto \exp \left( -\frac{c}{2} \left[ \mu^T (\tilde{T}_2 + M) \mu - 2\mu^T (\tilde{T}_1 + M\eta) \right] \right) \]

(A.7)

\[ \propto \exp \left( -\frac{c}{2} \left( \mu - \frac{M\eta + \tilde{T}_1}{M + \tilde{T}_2} \right)^T (M + \tilde{T}_2) \left( \mu - \frac{M\eta + \tilde{T}_1}{M + \tilde{T}_2} \right) \right) \]

(A.8)

\[ \propto \mathcal{N} \left( \mu; \frac{M\eta + \tilde{T}_1}{M + \tilde{T}_2}, c(M + \tilde{T}_2) \right). \]

(A.9)

Hence, we have a conjugate prior on \( \mu \) with the posterior parameters being:

\[ M' = M + \tilde{T}_2, \]  

(A.10)

\[ \eta' = M'^{-1} (M\eta + \tilde{T}_1), \]  

(A.11)

where,

\[ p(\mu|y,c) = \mathcal{N}(\mu; M', \eta'). \]  

(A.12)

Now, we calculate the posterior of \( c \):

\[ p(c|y) = \frac{p(y,\mu,c)}{p(y)p(\mu|y,c)} \]

(A.13)

\[ \propto \frac{p(y,\mu,c)}{p(\mu|y,c)} \]

(A.14)

\[ = \frac{p(y|\mu,c)p(\mu|c)p(c)}{p(\mu|y,c)} \]

(A.15)

\[ \propto c^{\frac{\beta}{2}} \tilde{T}_4 \exp \left( -\frac{c}{2} \left[ \tilde{T}_3 - 2\mu^T \tilde{T}_1 + \mu^T \tilde{T}_2 \mu \right] \right) \]
\[ \cdot |cM|^{\frac{1}{2}} \exp \left( -\frac{1}{2} (\mu - \eta)^T cM (\mu - \eta) \right) \]
\[ \cdot \frac{\beta^\alpha}{\Gamma(\alpha)} c^{\alpha-1} \exp (-\beta c) \]
\[ \cdot |cM'|^{\frac{1}{2}} \exp \left( \frac{1}{2} (\mu - \eta')^T cM' (\mu - \eta') \right) \]

(A.16)

\[ \propto c^{\frac{\beta}{2} + \alpha - 1} \exp \left( -\frac{c}{2} \left[ \tilde{T}_3 - 2\mu^T \tilde{T}_1 + \mu^T \tilde{T}_2 \mu \right. \right. \]
\[ \left. \left. + \mu^T M\mu - 2\mu^T M\eta + \eta^T M\eta + 2\beta \right. \right. \]
\[ \left. \left. - \mu^T (M + \tilde{T}_2) \mu + 2\mu^T M'M'^{-1} (M\eta + \tilde{T}_1) - \eta'^T M'\eta' \right] \right) \]

(A.17)
\[ p(c | \eta) = \text{Gamma}(c; \alpha + DN^2, \beta + \frac{1}{2} \left[ \tilde{T}_3 + \eta^T M \eta - \eta'^T M' \eta' \right]) \] (A.19)

Hence, we have a conjugate prior on \( c \) with the posterior parameters being:

\[ \alpha' = \alpha + \frac{DN^2}{2}, \] (A.20)
\[ \beta' = \beta + \frac{1}{2} \left[ \tilde{T}_3 + \eta^T M \eta - \eta'^T M' \eta' \right], \] (A.21)

where,

\[ p(c | y) = \text{Gamma}(c; \alpha', \beta'). \] (A.22)

\section*{A.3 Data Likelihood}

We now derive the data likelihood expression given in Equation (4.11):

\[ p(y) = \frac{p(y, \mu, c)}{p(\mu, c | y)} \] (A.23)
\[ = \frac{p(y | \mu, c)p(\mu | c)p(c)}{p(\mu | c, y)p(c | y)} \] (A.24)
\[ = (2\pi)^{-dN/2} e^{DN/2} \tilde{T}_4 \exp \left( -\frac{c}{2} \left[ \tilde{T}_3 - 2\mu^T \tilde{T}_1 + \mu^T \tilde{T}_2 \mu \right] \right) \]
\[ \cdot (2\pi)^{-\frac{d}{2}} |M|^{-\frac{1}{2}} \exp \left( -\frac{1}{2} (\mu - \eta)^T cM(\mu - \eta) \right) \]
\[ \cdot \frac{\beta\alpha}{\Gamma(\alpha)} c^{\alpha-1} \exp(-\beta c) \]
\[ \cdot (2\pi)^{\frac{d}{2}} |M'|^{-\frac{1}{2}} \exp \left( \frac{1}{2} (\mu - \eta')^T cM'(\mu - \eta') \right) \]
\[ \cdot \frac{\Gamma(\alpha')}{\beta'\alpha'} c^{1-\alpha'} \exp(\beta' c) \] (A.25)
\[ = (2\pi)^{-dN} \tilde{T}_4 |M|^{-\frac{1}{2}} |M'|^{-\frac{1}{2}} \frac{\beta\alpha}{\beta'\alpha'} \frac{\Gamma(\alpha')}{\Gamma(\alpha)} \]
\[ \cdot \exp \left( -\frac{c}{2} \left[ \tilde{T}_3 - 2\mu^T \tilde{T}_1 + \mu^T \tilde{T}_2 \mu \right] \right. \]
\[ \left. + \mu^T M \mu - 2\mu^T M \eta + \eta^T M \eta + 2\beta \right. \]
\[ \left. - \mu^T (M + \tilde{T}_2) \mu + 2\mu^T M' M'^{-1} (M \eta + \tilde{T}_1) - \eta'^T M' \eta \right) \]
\[
-2\beta - \left( \tilde{T}_3 + \eta^T M \eta - \eta'^T M' \eta' \right)
\]  
(A.26)

\[
= (2\pi)^{-\frac{dN}{2}} \tilde{T}_4 |M|^{\frac{1}{2}} |M'|^{-\frac{1}{2}} \frac{\beta^\alpha}{\beta'^{\alpha'}} \frac{\Gamma(\alpha')}{\Gamma(\alpha)}.
\]  
(A.27)

Therefore, we have the following expression for the data likelihood:

\[
p(y) = (2\pi)^{-\frac{dN}{2}} \tilde{T}_4 |M|^{\frac{1}{2}} |M'|^{-\frac{1}{2}} \frac{\beta^\alpha}{\beta'^{\alpha'}} \frac{\Gamma(\alpha')}{\Gamma(\alpha)}.
\]  
(A.28)
Bibliography


