Parallel Implementations of Parsimonious Gaussian Mixture Models and Extensions

PARALLEL IMPLEMENTATIONS OF PARSIMONIOUS GAUSSIAN MIXTURE MODELS AND EXTENSIONS

BY

TYLER ROICK, M.Sc.

A THESIS
SUBMITTED TO THE DEPARTMENT OF MATHEMATICS & STATISTICS
AND THE SCHOOL OF GRADUATE STUDIES
OF MCMASTER UNIVERSITY
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

© Copyright by Tyler Roick, May 2021

All Rights Reserved

Doctor of Philosophy (2021) McMaster University

(Mathematics & Statistics) Hamilton, Ontario, Canada

TITLE:	Parallel Implementations of Parsimonious Gaussian Mixture Models and Extensions
AUTHOR:	Tyler Roick M.Sc., (Statistics) McMaster University, Hamilton, Canada
SUPERVISOR:	Dr. Paul D. McNicholas
NUMBER OF PAGES:	xvii, 107

To Andrea, Anna Maria, Peter, and Perry.

None of this would have been possible without all of you.

Abstract

Cluster analysis is the process of finding underlying group structures in a set of data. Model-based clustering has an array of swiftly growing literature surrounding this topic; however, a Gaussian mixture model has always been a prevalent model in model-based clustering literature. More specially, when dealing with high-dimensional data, the parsimonious Gaussian mixture model has shown great computational efficiency because the number of covariance parameters is linear with the number of variables for each model in the family. Parsimonious Gaussian mixture models generalize the mixture of factor analyzers model. For each group, the number of factors q has traditionally been held constant. An extension to the parsimonious Gaussian mixture model family is developed allowing q to be a vector of equal length to the number of components. Although the parsimonious Gaussian mixture model family has shown great computational potential, this new extension takes away from the aforementioned feat with rapidly growing parameter combinations to fit. Parallel computational techniques are explored throughout this thesis to improve computational runtime and allow the rapidly growing number of parameter combinations to be fit in a realistic time frame, especially in the case of high-dimensional data. The techniques are applied to real data to assess performance and computational efficiency.

Acknowledgments

First and foremost I would like to extend my sincerest gratitude to my advisor Dr. Paul McNicholas for his unwavering support, guidance and patience throughout my graduate studies. Over the course of my degree you have inspired me with your numerous outstanding achievements and have taught me many invaluable lessons. Thank you for believing in me and giving me the opportunity to complete my thesis under your supervision.

I would also like to thank Dr. Roman Viveros-Aguilera and Dr. Ben Bolker for serving as members of my supervisory committee. I have truly appreciated the insight and knowledge that both of you have offered and it has been instrumental in the completion of my research. Thank you to Dr. Xin Gao for examining my thesis and to Dr. Randall Dumont for chairing my thesis defence.

Thank you to all of the present and past members of the McNicholas research group throughout my years as a graduate student. You have all been amazing colleagues throughout this journey and I will cherish the friendships made.

My deepest thanks is owed to my parents and brother for all of their unconditional love and encouragement. I am forever grateful for everything that you all have done for me throughout my studies. A very heartfelt thank you to Andrea, you never let me doubt that this goal was achievable. Thank you for your patience throughout this

journey, I am so incredibly lucky to have had you by my side.

Publications

The following publications are based on the work presented in this thesis and have been published, submitted, or are in preparation for submission for publication:

Roick, T., and McNicholas, P.D. Hybrid parallelization of parsimonious Gaussian mixture models. *In preparation.*

Roick, T., and McNicholas, P.D. Multi-Factor parsimonious Gaussian mixture models. *In preparation.*

The code used within this thesis and the above publications can be found on GitHub at `https://github.com/roickt/Thesis-Code`.

Contents

Abstract **iv**

Acknowledgments **v**

Publications **vii**

1 Introduction **1**

1.1 Cluster Analysis 1

1.2 Statistical and Parallel Computing 3

1.3 Thesis Outline 5

1.3.1 Chapter 2 5

1.3.2 Chapter 3 5

1.3.3 Chapter 4 5

1.3.4 Chapter 5 6

1.3.5 Chapter 6 6

1.4 Contributions to Literature 6

2 Background **8**

2.1 Mixture Models and Model-Based Clustering 8

2.2 EM Algorithm and Extensions 9
2.2.1 EM Algorithm for Model-Based Clustering 9
2.2.2 ECM Algorithm 10
2.2.3 AECM Algorithm 10
2.2.4 Woodbury Identity 11
2.2.5 Stopping Criterion 12
2.3 Model Selection 14
2.4 Performance Assessment 14
2.5 Mixtures of Factor Analyzers and Extensions 15
2.5.1 Factor Analysis 15
2.5.2 Mixture of Factor Analyzers 16
2.6 Parsimonious Gaussian Mixture Models 17
2.6.1 Original PGMM Family 17
2.6.2 Extended PGMM Family 17
2.6.3 Parameter Estimation 18
2.6.4 Mixture of Infinite Factor Analyzers 22

3 Multi-Factor Parsimonious Gaussian Mixture Models 24

3.1 Introduction 24
3.2 Methodology 25
3.3 Implementation 31
3.3.1 Julia 31
3.3.2 HPC Cluster 31
3.3.3 Message Passing Interface 33
3.3.4 Slurm 35

3.3.5 Parallel Design 37
3.4 Applications 39
3.4.1 Overview 39
3.4.2 Coffee Data Analyses 39
3.4.3 Italian Wine Data Analyses 41
3.4.4 Italian Olive Oil Data Analyses 43
3.4.5 Alon Colon Cancer Data Analyses 46
3.4.6 Golub Data Analyses 47
3.4.7 Breast Cancer Data Analyses 49
3.5 Discussion 50
3.5.1 Parallel Run Time Comparison 50
3.5.2 Discussion 51

4 Hybrid Parallelization of PGMMs **55**

4.1 Introduction 55
4.2 Implementation 56
4.2.1 OpenMP 56
4.2.2 Multithreading in Julia 57
4.3 Methodology 60
4.3.1 Hybrid Parallelization 60
4.4 Applications 63
4.4.1 Overview 63
4.4.2 Alon Data Analyses 63
4.4.3 Golub Data Analyses 66
4.5 Performance Analysis 68

4.6 Discussion 76

5 Parallelization with Python and Comparative Approaches 78

5.1 Introduction 78

5.2 Methodology 79

5.2.1 Implementations 79

5.2.2 Parallel Framework 82

5.3 Application 85

5.3.1 Combined Implementation 86

5.3.2 Native Python 88

5.4 Discussion 90

6 Conclusion 92

6.1 Summary 92

6.2 Future Work 93

6.2.1 Improving Parallel Efficiency and Determining Optimal Resources 93

6.2.2 Applications to Big Data and Other Model-Based Clustering Techniques 94

A Functions from Parallel Code 95

B Multi-Factor PGMM Results 98

Bibliography 101

List of Tables

2.1 Nomenclature, component covariance matrix structure, and number of free covariance parameters for eight parsimonious Gaussian mixture models. 18

2.2 Nomenclature, equivalent PGMM member where applicable (Table 2.1), and component covariance matrix structure for each member of the expanded PGMM family. 19

3.1 Twelve chemical properties of the coffee data. 40

3.2 Cross-tabulation of the MAP classifications (A-B) associated with the selected PGMM with multiple factors against true classes for the Coffee data. 41

3.3 Cross-tabulation of the MAP classifications (A-C) associated with the selected PGMM with fixed factors against true classes for the Coffee data. 41

3.4 Twenty-seven constituents of the Italian Wine data. 42

3.5 Cross-tabulation of the MAP classifications (A-C) associated with the selected PGMM against true classes for the Italian Wine data. 42

3.6 Cross-tabulation of the MAP classifications (A-C) associated with the most frequent MIFA model against true classes for the Italian Wine data. 42
3.7 Eight fatty acids from the Italian Olive Oil data. 43
3.8 Cross-tabulation of the MAP classifications (A-C) associated with the selected PGMM against true classes for the Italian Olive Oil data by region. 44
3.9 Cross-tabulation of the MAP classifications (A-B) associated with the most frequent MIFA model against true classes for the Italian Olive Oil data by region. 44
3.10 Cross-tabulation of the MAP classifications (A-I) associated with the selected PGMM against true classes for the Italian Olive Oil data by area. 45
3.11 Cross-tabulation of the MAP classifications (A-F) associated with the most frequent MIFA model against true classes for the Italian Olive Oil data by area. 45
3.12 Cross-tabulation of the MAP classifications (A-B) associated with the selected PGMM against true classes for the Alon Colon Cancer data classified by extraction technique. 47
3.13 Cross-tabulation of the MAP classifications (A-B) associated with the most frequent MIFA model against true classes for the Alon Colon Cancer data classified by extraction technique. 47
3.14 Cross-tabulation of the MAP classifications (A-B) associated with the selected PGMM against true classes for the Golub data. 48

3.15 Cross-tabulation of the MAP classifications (A-B) associated with the most frequent MIFA model against true classes for the Golub data. . 48

3.16 Cross-tabulation of the MAP classifications (A-C) associated with the selected PGMM against true classes for the WDBC data. 49

3.17 Cross-tabulation of the MAP classifications (A-C) associated with the most frequent MIFA model against true classes for the WDBC data. . 49

4.1 Time taken using MPI parallelization to run `pgmm` on the Alon data for different numbers of processes. 65

4.2 Time taken using OpenMP parallelization to run `pgmm` on the Alon data for different numbers of threads. 65

4.3 Time taken in seconds using hybrid parallelization to run `pgmm` on the Alon data for different numbers of slaves/threads. 65

4.4 Time taken using MPI parallelization to run `pgmm` on the Golub data for different numbers of processes. 68

4.5 Time taken using OpenMP parallelization to run `pgmm` on the Golub data for different numbers of threads. 68

4.6 Time taken in seconds using hybrid parallelization to run `pgmm` on the Golub data for different numbers of slaves/threads. 69

4.7 Profiling PGMM for the Alon data in serial. 73

4.8 Hotspot profiling PGMM for the Alon data in serial. 74

4.9 Hotspot profiling PGMM for the Alon data using OpenMP. 74

4.10 Memory access profiling PGMM for the Alon data using OpenMP. . . 75

4.11 Hotspot profiling PGMM for the Alon data using MPI. 75

4.12 Memory access profiling PGMM for the Alon data using MPI. 76

5.1 Time taken using MPI parallelization to run pgmm on the Alon data for different numbers of processes outside of Python. 87
5.2 Time taken using the Multiprocessing module to run pgmm on the Alon data for different numbers of processes outside of Python. 88
5.3 Time taken in seconds using hybrid parallelization to run pgmm on the Alon data for different numbers of slaves/processes outside of Python. 88
5.4 Time taken using MPI parallelization to run pgmm on the Alon data for different numbers of processes in native Python. 89
5.5 Time taken using the Multiprocessing class to run pgmm on the Alon data for different numbers of processes in native Python. 89
5.6 Time taken in seconds using hybrid parallelization to run pgmm on the Alon data for different numbers of slaves/processes in native Python. . 90
B.1 Time taken using MPI parallelization to run mf-pgmm on the coffee data for different numbers of processes. 98
B.2 Time taken using MPI parallelization to run mf-pgmm on the Italian Wine data for different numbers of processes. 99
B.3 Time taken using MPI parallelization to run mf-pgmm on the Italian Olive Oil data for different numbers of processes. 99
B.4 Time taken using MPI parallelization to run mf-pgmm on the Alon for different numbers of processes. 99
B.5 Time taken using MPI parallelization to run mf-pgmm on the Golub data for different numbers of processes. 100

List of Figures

3.1 The generic layout of a high performance computer. 32

3.2 The serial and parallel runtimes for Coffee, Wine, Olive Oil, Alon, and Golub datasets plotted against the dashed-line for ideal scaling using the mean average serial runtime. 52

3.3 The speed-up per number of cores for Coffee, Wine, Olive Oil, Alon, and Golub datasets plotted against the dashed-line representing linear speed-up. 53

3.4 The efficiency per number of cores for Coffee, Wine, Olive Oil, Alon, and Golub datasets plotted against the dashed-line representing perfect efficiency. 54

4.1 Iterations per thread using different base multithreading implementations in Julia. 59

4.2 Hybrid MPI and OpenMP communication occurring within a HPC. . 61

4.3 Graph of speed-up using OpenMP, MPI, and hybrid parallelization techniques for the Alon data. 66

4.4 Graph of speed-up using OpenMP, MPI, and hybrid parallelization techniques for the Alon data in comparison to the pgmm package in R. 67

4.5 Graph of speed-up using OpenMP, MPI, and hybrid parallelization techniques for the Golub data. 69

4.6 Graph of speed-up using OpenMP, MPI, and hybrid parallelization techniques for the Golub data in comparison to the `pgmm` package in R. 70

Chapter 1

Introduction

1.1 Cluster Analysis

A cluster can be defined as a collection of data (or observations). Cluster analysis refers to the process of classifying a set of data into closely related clusters (or groups), where observations placed within groups are as similar to one another as possible and observations across groups are as dissimilar as possible. Multiple different methods for cluster analysis exist in current literature that aim to identify different underlying group structure within a dataset.

Methods for cluster analysis fall into two broad subcategories: hierarchical versus non-hierarchical and parametric versus non-parametric. Hierarchical clustering approaches treat each observation in a set of data as a separate cluster and iteratively merges the two closest clusters, based on specified similarity and distance measures, until there is a single cluster. Non-hierarchical clustering approaches have the added flexibility of splitting and merging clusters. This approach no longer follows a step-by-step tree like structure, but will optimize results by minimizing or maximizing a

distance-based measure overall. An example of a hierarchical versus non-hierarchical clustering technique would be hierarchical clustering (Ward, 1963) versus k-means clustering (MacQueen, 1967; Hartigan and Wong, 1979). Both of these distance-based approaches can further be categorized as non-parametric clustering methods — although k must be selected in the latter. Although non-parametric, distance-based approaches are frequently used due to their ease of implementation, approaches less sensitive to the choice of distance metric are desirable. This is where parametric, model-based approaches using mixture models fit in. Mixture model-based clustering based on parametric finite mixture models offer more flexibility in regards to the dimensionality and complexity of data. Mixture models also fall into the non-hierarchical category for cluster analysis.

We consider the definition of a cluster to be that given by McNicholas (2016a):

> . . . a cluster is a unimodal component within an appropriate finite mixture model.

Mixture model-based approaches offer a great advantage to clustering because they allow the use of approximate Bayes factors to compare models which gives a systematic means of selecting model parametrization and the number of clusters (Fraley and Raftery, 1998). However, McLachlan and Peel (2000) state that applying model-based clustering when a large number of observed variables are present in the data may cause it to be over-parametrized and computationally intensive. This is due to the mixing weights and estimation of the mean vector and covariance matrix for each component fit.

1.2 Statistical and Parallel Computing

Statistical computing is the process of creating computer algorithms to implement statistical methods. It is a necessary technique with modern day big data that may require time intensive calculations such as those mentioned for model-based clustering. Traditionally, these algorithms are written for serial computation. This would suggest that all computations are run on a single computer using a single central processing unit (CPU). Problems are broken down into discrete series of instructions which are executed sequentially, one at a time.

Parallel computing involves the simultaneous use of multiple computing resources to solve a problem. The problems are broken down into discrete parts that can be solved concurrently where each part is executed simultaneously on a different computing resource. Computing resources can include a single computer with multiple processors, a number of computers connected across a network, or any combination thereof. Parallel computing is used for two primary reasons: to reduce computational time and to solve larger scale problems that typically require more resources, e.g., memory.

Parallel computing is done using different levels of granularity, most commonly fine-grain or coarse-grain. The level of granularity refers to the amount of work performed by a specific task. Fine-grain parallelism breaks down a problem into a large number of smaller tasks, but requires more resources, while coarse-grain parallelism would break down the problem into a smaller number of larger tasks using fewer resources. Consider 10 arrays that must each be summed. Given that 10 processors are available, a fine-grain parallelism scenario could consist of sending each array to a different processor to be summed. Coarse-grain parallelism would only require two

processors to be available in which each processor receives five arrays to sum. Note, this is an inefficient example of parallelism due to the overhead required to communicate tasks between processors because each task can be accomplished more efficiently than the communication overhead.

Parallel computing is classified by one of four possible computer architectures using Flynn's classical taxonomy (Flynn, 1966). These classifications distinguish between two independent dimensions, instruction and data, where each dimension can be in a single or multiple state. The focus of parallel computing throughout this thesis falls under the multiple instruction, multiple data category. Each processor has the ability to be working on a different set of instructions during the same clock cycle with a different stream of data.

There are two primary forms of memory architecture in parallel computing, namely shared memory and distributed memory. Shared memory allows for all processors to access all memory space as a global address. Sharing information in this architecture is extremely convenient, but puts a large burden on the programmer to synchronize constructs to ensure access of the global memory is done safely. It can become increasingly difficult to implement this architecture with larger numbers of processors. Distributed memory allows for each processor to have local memory where the memory on the processor does not map to any other space. Memory is scalable with the number of processors but requires the programmer to account for numerous details of data communication between processors and the cost of initial distribution.

1.3 Thesis Outline

1.3.1 Chapter 2

Background information is given, including details on finite mixture models, the expectation-maximization algorithm and variants thereof, and factor analysis in model-based clustering. Parsimonious Gaussian mixture models are introduced alongside details of parameter estimation. Computational techniques for improving performance are reviewed.

1.3.2 Chapter 3

A variational approach for the six of twelve models in the extended parsimonious Gaussian mixture model family with unconstrained factor loadings that will allow for the number of factors fit per component to vary is proposed. Parameter estimation for the six variant models is discussed. The approach is illustrated on real data using parallel computing techniques to improve performance due to the significant computational time required to run such models.

1.3.3 Chapter 4

A computational technique for model-based clustering using a hybrid of existing parallel computing methods is developed in Julia. Details of how such a technique can be implemented on a single computer and on a high-performance computer are discussed and illustrated. The technique is applied to parsimonious Gaussian mixture models using multiple real data sets to asses it's performance over existing methods.

1.3.4 Chapter 5

Details of how the technique developed in Chapter 4 was originally developed in Python are presented. The performance of computational statistics applications using Python, R, C, and combinations of these three languages is given. New approaches for the parallelization of existing code in R using Python and Julia are illustrated.

1.3.5 Chapter 6

The ideas and methods developed in this thesis are summarized and suggestions for future work are discussed.

1.4 Contributions to Literature

The impact on the existing literature of the ideas proposed in this thesis can be summarized as follows:

- The multi-factor parsimonious Gaussian mixture models developed here offer a much more flexible approach compared to the standard extended parsimonious Gaussian mixture model family. By allowing the number of factors fit to vary in models with unconstrained factor loadings a more accurate model may be fit to certain types of data while still allowing the standard models with a fixed number of factors to be chosen as the best model.

- The assessment of a hybrid parallel computing technique for model-based clustering offers insight in to how the future of big data may be handled to optimize computational time. Although heavily dependent on resources available,

an accurate comparison of parallel computing techniques when applied to computational statistical methods is provided.

- Details of how parallel techniques can be applied in both Python and Julia are illustrated. The application of these techniques shows that even with limited resources, large improvements in computational time can be achieved. Code that has been previously written in a popular language such as R can be run in parallel to improve performance. Reasoning as to why the future of computational statistics may be diverging from R become apparent.

Chapter 2

Background

2.1 Mixture Models and Model-Based Clustering

Model-based clustering is a technique for estimating group or cluster memberships, in which no observations are *a priori* labeled, based on finite mixture models. Finite mixture models assume that a population can be modelled as a collection of subpopulations where each subpopulation can be modelled by a statistical distribution. A random vector $\mathbf{X}$ is said to arise from a parametric finite mixture distribution if, for all $\mathbf{x} \subset \mathbf{X}$, its density can be written

$$f(\mathbf{x} \mid \boldsymbol{\vartheta}) = \sum_{g=1}^{G} \pi_g f_g(\mathbf{x} \mid \boldsymbol{\theta}_g),$$

where π_g is the gth mixing proportion satisfying both $\pi_g \in (0, 1]$ and $\sum_{g=1}^{G} \pi_g = 1$. $f_g(\mathbf{x} \mid \boldsymbol{\theta}_g)$ is the gth component density, and $\boldsymbol{\vartheta} = (\pi_1, \ldots, \pi_g, \boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_G)$ denotes the parameters. The component densities $f_1(\mathbf{x} \mid \boldsymbol{\theta}_1), \ldots, f_G(\mathbf{x} \mid \boldsymbol{\theta}_G)$ are commonly taken to be the same type.

A vast amount of work in the field of model-based clustering has been focused on the use of a Gaussian mixture model. In this case, the gth component density above is replaced with the density of a random variable from a multivariate Gaussian distribution. The Gaussian mixture model has $(G-1)+Gp+Gp(p+1)/2$ free parameters, where $G-1$ free parameters come from the estimation of the mixing proportions, Gp from the estimation of the means, and $Gp(p+1)/2$ from the estimation of the covariances. An extensive review of finite mixture models can be found in McNicholas (2016b).

2.2 EM Algorithm and Extensions

2.2.1 EM Algorithm for Model-Based Clustering

The expectation-maximization (EM) algorithm (Dempster *et al.*, 1977) is an iterative algorithm used to find maximum likelihood estimates in the presence of missing or incomplete data. Each iteration of the EM algorithm involves two steps, the expectation step (E-step) and the maximization step (M-step). In the E-step, the conditional expected value of the complete-data log-likelihood, $\mathcal{Q}$, is computed. In the M-step, $\mathcal{Q}$ is maximized with respect to the model parameters. The iterations are repeated until a desired stopping criterion is reached. Possible stopping criteria are discussed in Section 2.2.5. It is worth noting that Titterington *et al.* (1985) cite similar approaches to the EM algorithm that were used by Baum *et al.* (1970), Orchard and Woodbury (1972), and Sundberg (1974).

In a model-based clustering scenario, complete-data refers to the combination of the observed data $\mathbf{x}_1, \ldots, \mathbf{x}_n$ along with the unknown labels $\mathbf{z}_1, \ldots, \mathbf{z}_n$, where $\mathbf{z}_i =$

$(z_{i1}, \ldots, z_{iG})$. Note that $\mathbf{z}_i$ denotes the group memberships of observation i, where z_{ig} is an indicator variable used to represent whether or not observation $\mathbf{x}_i$ belongs to group g. The indicator variable can formally be defined as

$$z_{ig} = \begin{cases} 1 & \text{if } \mathbf{x}_i \text{ belongs to component } g, \\ 0 & \text{otherwise,} \end{cases}$$

for observations $i = 1, \ldots, n$ and components $g = 1, \ldots, G$. The estimation of z_{ig} is considered the primary objective in model-based clustering.

2.2.2 ECM Algorithm

An extension to the EM algorithm, the expectation-conditional maximization (ECM) algorithm, was introduced by Meng and Rubin (1993) for scenarios in which the complete-data likelihood is slightly more complicated. The M-step of the EM algorithm here is replaced by a series of computationally more simple conditional-maximization steps (CM-steps). This is accomplished by conditioning on some of the parameters being estimated. Meng and Rubin (1993) show that the ECM algorithm shares properties of the EM algorithm, e.g., each iteration increases the likelihood. They also show that, although the algorithm may require more iterations to reach convergence, the overall computation time is faster than that of the EM algorithm.

2.2.3 AECM Algorithm

A further extension to both the EM and ECM algorithm, the alternating expectation-conditional maximization (AECM) algorithm, was introduced by Meng and van Dyk

(1997). This variant replaces a single M-step by a series of CM-steps and allows for a different specification of the complete-data at each stage. Meng and van Dyk (1997) show that the monotone convergence properties of the EM and ECM algorithms hold for the AECM algorithm. As with the ECM algorithm, the AECM algorithm has an increased number of iterations to reach convergence, but can be more computationally efficient than the EM algorithm. An extensive review of the EM algorithm as well as the aforementioned and other extensions can be found in McLachlan and Krishnan (2008).

2.2.4 Woodbury Identity

The AECM algorithm requires the inversion of a $p \times p$ matrix $\mathbf{\Psi} + \mathbf{\Lambda}\mathbf{\Lambda}'$ which can be both computationally expensive when large values of p are present. The Woodbury identity (Woodbury, 1950) is used in this scenario to avoid potentially problematic calculation. McNicholas (2016a) shows that given an $m \times m$ matrix $\mathbf{A}$, an $m \times k$ matrix $\mathbf{U}$, a $k \times k$ matrix $\mathbf{C}$, and a $k \times m$ matrix $\mathbf{V}$, the Woodbury identity states

$$(\mathbf{A} + \mathbf{U}\mathbf{C}\mathbf{V})^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}\left(\mathbf{C}^{-1} + \mathbf{V}\mathbf{A}^{-1}\mathbf{U}\right)^{-1}\mathbf{V}\mathbf{A}^{-1}.$$

When setting $\mathbf{U} = \mathbf{\Lambda}$, $\mathbf{V} = \mathbf{\Lambda}'$, $\mathbf{A} = \mathbf{\Psi}$, and $\mathbf{C} = \mathbf{I}_q$, this yields that

$$(\mathbf{\Psi} + \mathbf{\Lambda}\mathbf{\Lambda}')^{-1} = \mathbf{\Psi}^{-1} - \mathbf{\Psi}^{-1}\mathbf{\Lambda}\left(\mathbf{I}_q + \mathbf{\Lambda}'\mathbf{\Psi}^{-1}\mathbf{\Lambda}\right)^{-1}\mathbf{\Lambda}'\mathbf{\Psi}^{-1},$$

which, in turn, simplifies the problem to the inversion of a $q \times q$ matrix, where q is much less than p, along with some diagonal matrices. A similar identity can also used

for finding the determinant of the covariance matrix in the AECM algorithm, i.e.,

$$|\boldsymbol{\Psi} + \boldsymbol{\Lambda}\boldsymbol{\Lambda}'| = \frac{|\boldsymbol{\Psi}|}{\left|\mathbf{I}_q - \boldsymbol{\Lambda}' \left(\boldsymbol{\Lambda}\boldsymbol{\Lambda}' + \boldsymbol{\Psi}\right)^{-1} \boldsymbol{\Lambda}\right|}.$$

2.2.5 Stopping Criterion

A variety of stopping criteria have been suggested to determine convergence of the EM algorithm and its variants. A common approach is to stop the algorithm when increases in the log-likelihood fall below a certain threshold, ϵ, after a number of iterations. This can be written as,

$$\ell^{(k+1)} - \ell^{(k)} < \epsilon. \tag{2.1}$$

However, (2.1) is more a measure of the "lack of progress" than true convergence because the log-likelihood can "jump" in a stair-like pattern (McNicholas, 2016a). More decisive convergence criteria have since been developed to determine if the EM algorithm has converged. Some well-known approaches revolve around the use of Aitken's acceleration (Aitken, 1926). We can use Aitken's acceleration to estimate the asymptotic maximum log-likelihood at each iteration of the EM algorithm and hence to determine whether the algorithm can be stopped or not. At iteration k, the Aitken acceleration is given by

$$a^{(k)} = \frac{\ell^{(k+1)} - \ell^{(k)}}{\ell^{(k)} - \ell^{(k-1)}},$$

where $\ell^{(k)}$ is the log-likelihood value at iteration k. The asymptotic estimate of the log-likelihood (Böhning *et al.*, 1994) at iteration $k+1$ is given by

$$\ell_{\infty}^{(k+1)} = \ell^{(k)} + \frac{1}{1-a^{(k)}}(\ell^{(k+1)} - \ell^{(k)}),$$

where each value is as previously defined. The stopping criterion developed by Lindsay (1995) suggests that the EM algorithm can be stopped when

$$\ell_{\infty}^{(k)} - \ell^{(k)} < \epsilon, \tag{2.2}$$

where ϵ is a small value. An alternative stop is proposed by McNicholas and Murphy (2010a), which suggests that the algorithm has converged when

$$\ell_{\infty}^{(k+1)} - \ell^{(k)} < \epsilon, \tag{2.3}$$

for a small value of ϵ, conditional on the fact that difference in (2.3) is positive. McNicholas (2016a) points out that the only case in which the difference can achieve a negative value is for $a^{(k)} > 1$, which would not be a reasonable place to stop. It was shown by McNicholas *et al.* (2010) that the criterion in (2.3) is at least as strict as (2.2) since $\ell^{(k+1)} \geq \ell^{(k)}$. It was also shown that the criterion in (2.3) is at least as strict as the lack of progress criterion in (2.1). The stopping criterion proposed by McNicholas and Murphy (2010a) will be adopted herein.

2.3 Model Selection

When using mixture models, an objective criterion is needed to select the 'best' model. While Bayes factors are known to have desirable properties for model selection, they are not evaluated with ease. Instead, the Bayesian information criterion (BIC; Schwarz, 1978) can be used to approximate the Bayes factor and select the best model. When comparing two models, the difference in the BIC gives a rough approximation to the logarithm of the Bayes factor assuming the prior distributions are equal (Kass and Wasserman, 1995a). Given a model of parameters $\boldsymbol{\Theta}$, the BIC can be written

$$\text{BIC} = 2\ell(\hat{\boldsymbol{\Theta}}) - \rho \log n,$$

where $\ell(\hat{\boldsymbol{\Theta}})$ is the maximized log-likelihood, $\hat{\boldsymbol{\Theta}}$ is the maximum likelihood estimate of $\boldsymbol{\Theta}$, ρ is the number of free parameters, and n is the number of observations. The use of the BIC for model selection is a well known approach in model-based clustering. Justifications for its use can be found in Leroux (1992), Kass and Wasserman (1995b), Kass and Wasserman (1995a), and Keribin (2000).

2.4 Performance Assessment

In a true clustering scenario, the group memberships are not known *a priori*. To evaluate the effectiveness of the models, data with known group memberships are treated as unknown. The model is then evaluated using a cross-tabulation of the maximum *a posteriori* (MAP) classification of the predicted group memberships and that of the true group memberships. From the results of the cross-tabulation, the performance can be quantified though the use of the adjusted Rand index (ARI;

Hubert and Arabie, 1985). The Rand index (Rand, 1971) alone does not account for agreement by chance. This is to say that when predicted group memberships are obtained, there is a chance they would be classified correctly by chance. The Rand index is based on pairwise agreement, written as

$$\frac{\text{number of pairwise agreements}}{\text{number of pairs}},$$

where a value on $[0,1]$ is obtained, 1 being perfect class agreement. The ARI is used as it corrects the Rand index for agreement by chance and has an expected value of 0 under random classification while still having a value of 1 for perfect classification.

Steinley (2003) states that the ARI is a better measure of performance for classification problems than simply calculating the misclassification rate. General properties of the ARI can be found in Steinley (2004), who shows that the mean and standard deviation of the ARI are not altered when changes occur in the number of clusters, the number of observations, and the percent of observations within each cluster.

2.5 Mixtures of Factor Analyzers and Extensions

2.5.1 Factor Analysis

Factor Analysis (Spearman, 1904) is a data reduction technique that aims to model p dimensional random variables $\mathbf{X}_1, \ldots, \mathbf{X}_n$ using q dimensional vectors of unobserved latent variables $\mathbf{U}_1, \ldots, \mathbf{U}_n$ where $q < p$ (McNicholas *et al.*, 2010). The factor analysis model assumes $\mathbf{X}_i$ can be modeled as

$$\mathbf{X}_i = \boldsymbol{\mu} + \boldsymbol{\Lambda}\mathbf{U}_i + \boldsymbol{\varepsilon}_i,$$

for $i = 1, \ldots, n$, where $\mathbf{\Lambda}$ represents a $p \times q$ matrix of factor loadings, $\mathbf{U}_i$ are the latent factors which are independent of $\boldsymbol{\varepsilon}_i$ and are independently distributed as $\mathrm{N}(\mathbf{0}, \mathbf{I}_q)$ and $\mathrm{N}(\mathbf{0}, \mathbf{\Psi})$, respectively. Note, $\mathbf{\Psi}$ is a positive $p \times p$ diagonal matrix. From the factor analysis model, the marginal distribution of $\mathbf{X}_i$ can be written as a multivariate Gaussian with mean $\boldsymbol{\mu}$ and covariance $\mathbf{\Lambda}\mathbf{\Lambda}' + \mathbf{\Psi}$. With the decomposed covariance, $\mathbf{\Lambda}\mathbf{\Lambda}' + \mathbf{\Psi}$, the number of free covariance parameters shrinks from $\frac{1}{2}p(p+1)$ to $pq + p - \frac{1}{2}q(q-1)$ (Lawley and Maxwell, 1962). Lawley and Maxwell (1962) show an overall reduction of

$$\frac{1}{2}\left[(p-q)^2 - (p+q)\right]$$

parameters, provided that

$$(p-q)^2 > (p+q).$$

See McNicholas (2016a) for a more in-depth review.

2.5.2 Mixture of Factor Analyzers

Similar to the factor analysis model, the mixture of factor analyzers (MFA) model assumes $\mathbf{X}_i$ can be modeled as

$$\mathbf{X}_i = \boldsymbol{\mu}_g + \mathbf{\Lambda}_g \mathbf{U}_{ig} + \boldsymbol{\varepsilon}_{ig},$$

with probability π_g, for $i = 1, \ldots, n$ and $g = 1, \ldots, G$. The parameters remain unchanged in definition from the aforementioned Factor Analysis model with the exception of being unique to the component g for $g = 1, \ldots, G$. It follows that the

joint distribution of $\mathbf{X}_1, \ldots, \mathbf{X}_n$ can be written as

$$f(\mathbf{x} \mid \boldsymbol{\vartheta}) = \sum_{g=1}^{G} \pi_g \, \phi(\mathbf{x}_i \mid \boldsymbol{\mu}_g, \boldsymbol{\Lambda}_g \boldsymbol{\Lambda}_g' + \boldsymbol{\Psi}_g),$$

where $\boldsymbol{\vartheta}$ represents the model parameters. As with the factor analysis model, the covariance structure is decomposed as $\boldsymbol{\Sigma}_g = \boldsymbol{\Lambda}_g \boldsymbol{\Lambda}_g' + \boldsymbol{\Psi}_g$.

A mixture of factor analyzers was first introduced by Ghahramani and Hinton (1997) with the constraint $\boldsymbol{\Psi}_g = \boldsymbol{\Psi}$, stating that this constraint can be relaxed. Tipping and Bishop (1997, 1999) introduces mixtures of probabilistic principal component analyzers by applying the isotropic constraint $\boldsymbol{\Psi}_g = \psi_g \mathbf{I}_p$. A fully unconstrained model was presented by McLachlan and Peel (2000).

2.6 Parsimonious Gaussian Mixture Models

2.6.1 Original PGMM Family

McNicholas and Murphy (2005, 2008) develop an eight-member family of parsimonious Gaussian mixture models (PGMMs) which use the factor analyzers model. By imposing constraints on $\boldsymbol{\Lambda}_g$ and/or $\boldsymbol{\Psi}_g$ to be equal across components together with the option to impose the isotropic constraint $\boldsymbol{\Psi}_g = \delta_g \mathbf{I}_p$ further reduces the number of parameters and gives way to the PGMM family (Table 2.1).

2.6.2 Extended PGMM Family

This family of eight models was extended by McNicholas and Murphy (2010b) by decomposing $\boldsymbol{\Psi}_g = \omega_g \boldsymbol{\Delta}_g$, where $\omega_g \in \mathbb{R}^+$ and $\boldsymbol{\Delta}_g$ is a diagonal matrix satisfying

Table 2.1: Nomenclature, component covariance matrix structure, and number of free covariance parameters for eight parsimonious Gaussian mixture models.

PGMM Nomenclature				
$\mathbf{\Lambda}_g = \mathbf{\Lambda}$	$\mathbf{\Psi}_g = \mathbf{\Psi}$	$\mathbf{\Psi}_g = \psi_g \mathbf{I}_p$	$\mathbf{\Sigma}_g$	Free Cov. Parameters
C	C	C	$\mathbf{\Lambda}\mathbf{\Lambda}' + \psi\mathbf{I}_p$	$pq - q(q-1)/2 + 1$
C	C	U	$\mathbf{\Lambda}\mathbf{\Lambda}' + \mathbf{\Psi}$	$pq - q(q-1)/2 + p$
C	U	C	$\mathbf{\Lambda}\mathbf{\Lambda}' + \psi_g\mathbf{I}_p$	$pq - q(q-1)/2 + G$
C	U	U	$\mathbf{\Lambda}\mathbf{\Lambda}' + \mathbf{\Psi}_g$	$pq - q(q-1)/2 + Gp$
U	C	C	$\mathbf{\Lambda}_g\mathbf{\Lambda}_g' + \psi\mathbf{I}_p$	$G[pq - q(q-1)/2] + 1$
U	C	U	$\mathbf{\Lambda}_g\mathbf{\Lambda}_g' + \mathbf{\Psi}$	$G[pq - q(q-1)/2] + p$
U	U	C	$\mathbf{\Lambda}_g\mathbf{\Lambda}_g' + \psi_g\mathbf{I}_p$	$G[pq - q(q-1)/2] + G$
U	U	U	$\mathbf{\Lambda}_g\mathbf{\Lambda}_g' + \mathbf{\Psi}_g$	$G[pq - q(q-1)/2] + Gp$

$|\mathbf{\Delta}_g| = 1$. Similar to the previously given component covariance structure, it can now be written as

$$\mathbf{\Sigma}_g = \mathbf{\Lambda}_g\mathbf{\Lambda}_g' + \omega_g\mathbf{\Delta}_g.$$

The option to impose all, some, or none of the four constraints, $\mathbf{\Lambda}_g = \mathbf{\Lambda}$, $\omega_g = \omega$, $\mathbf{\Delta}_g = \mathbf{\Delta}$ and $\mathbf{\Delta}_g = \mathbf{I}_p$, gives way to a family of 12 models referred to as the expanded PGMM (EPGMM) family, see Table 2.2. Imposing any valid combination of constraints allows for a different number of free parameters in the corresponding model as well as different levels of parsimony. Parameter estimation for all members of the PGMM family can be carried out using AECM algorithms. The family of 12 models has been implemented in the `pgmm` package (McNicholas *et al.*, 2019) available in R. For a more detailed review of the PGMM family see McNicholas (2016a).

2.6.3 Parameter Estimation

A brief description of parameter estimation using the AECM algorithm is given from McNicholas (2016a). Allow the complete-data to be the observed data, $\mathbf{x}_1, \ldots, \mathbf{x}_n$,

Table 2.2: Nomenclature, equivalent PGMM member where applicable (Table 2.1), and component covariance matrix structure for each member of the expanded PGMM family.

PGMM Nomenclature					
$\mathbf{\Lambda}_g = \mathbf{\Lambda}$	$\mathbf{\Delta}_g = \mathbf{\Delta}$	$\omega_g = \omega$	$\mathbf{\Delta}_g = \mathbf{I}_p$	PGMM Equivalent	$\mathbf{\Sigma}_g$
C	C	C	C	CCC	$\mathbf{\Lambda}\mathbf{\Lambda}' + \omega\mathbf{I}_p$
C	C	U	C	CUC	$\mathbf{\Lambda}\mathbf{\Lambda}' + \omega_g\mathbf{I}_p$
U	C	C	C	UUC	$\mathbf{\Lambda}_g\mathbf{\Lambda}_g' + \omega\mathbf{I}_p$
U	C	U	C	UUC	$\mathbf{\Lambda}_g\mathbf{\Lambda}_g' + \omega_g\mathbf{I}_p$
C	C	C	U	CCU	$\mathbf{\Lambda}\mathbf{\Lambda}' + \omega\mathbf{\Delta}$
C	C	U	U	–	$\mathbf{\Lambda}\mathbf{\Lambda}' + \omega_g\mathbf{\Delta}$
U	C	C	U	UCU	$\mathbf{\Lambda}_g\mathbf{\Lambda}_g' + \omega\mathbf{\Delta}$
U	C	U	U	–	$\mathbf{\Lambda}_g\mathbf{\Lambda}_g' + \omega\mathbf{\Delta}_g$
C	U	C	U	–	$\mathbf{\Lambda}\mathbf{\Lambda}' + \omega\mathbf{\Delta}_g$
C	U	U	U	CUU	$\mathbf{\Lambda}\mathbf{\Lambda}' + \omega_g\mathbf{\Delta}_g$
U	U	C	U	–	$\mathbf{\Lambda}_g\mathbf{\Lambda}_g' + \omega\mathbf{\Delta}_g$
U	U	U	U	UUU	$\mathbf{\Lambda}_g\mathbf{\Lambda}_g' + \omega_g\mathbf{\Delta}_g$

and $\mathbf{z}_1, \ldots, \mathbf{z}_n$ to be the unknown labels. The mixing proportions, π_g, and component means, $\boldsymbol{\mu}_g$, will be estimated for $g = 1, \ldots, G$. By using the expected values of the component membership labels, $\hat{z}_{ig}$, for $i = 1, \ldots, n$ and $g = 1, \ldots, G$ in the complete data log-likelihood, the expected value of the complete-data log-likelihood is found to be

$$\begin{aligned} Q_1 = & \sum_{i=1}^{n}\sum_{g=1}^{G} \hat{z}_{ig}[\log \pi_g + \log \phi(\mathbf{x}_i \mid \boldsymbol{\mu}_g, \mathbf{\Lambda}_g\mathbf{\Lambda}_g^{'} + \mathbf{\Psi}_g)] \\ = & \sum_{g=1}^{G} n_g \log \pi_g - \frac{np}{2}\log 2\pi - \sum_{g=1}^{G}\frac{n_g}{2}\log|\mathbf{\Lambda}_g\mathbf{\Lambda}_g^{'} + \mathbf{\Psi}_g| \\ & - \sum_{g=1}^{G}\frac{n_g}{2}\operatorname{tr}\{\mathbf{S}(\mathbf{\Lambda}_g\mathbf{\Lambda}_g^{'} + \mathbf{\Psi}_g)^{-1}\}, \end{aligned}$$

where n_g can be found from the sum of the component membership labels for all individuals, and

$$\mathbf{S}_g = \frac{1}{n_g} \sum_{i=1}^{n} \hat{z}_{ig} (\mathbf{x}_i - \boldsymbol{\mu}_g)(\mathbf{x}_i - \boldsymbol{\mu}_g)'. \tag{2.4}$$

Q_1 can then be maximized with respect to the model parameters to obtain

$$\hat{\pi}_g = \frac{n_g}{n} \quad \text{and} \quad \hat{\boldsymbol{\mu}}_g = \frac{\sum_{i=1}^{n} \hat{z}_{ig} \mathbf{x}_i}{\sum_{i=1}^{n} \hat{z}_{ig}}. \tag{2.5}$$

For the next stage of the AECM algorithm, estimates of $\boldsymbol{\Lambda}_g$ and $\boldsymbol{\Psi}_g$ are found. First consider the joint distribution described in McNicholas (2016a),

$$\begin{bmatrix} \mathbf{X}_i \\ \mathbf{U}_i \end{bmatrix} \sim N \left(\begin{bmatrix} \boldsymbol{\mu} \\ \mathbf{0} \end{bmatrix}, \begin{bmatrix} \boldsymbol{\Lambda}\boldsymbol{\Lambda}' + \boldsymbol{\Psi} & \boldsymbol{\Lambda} \\ \boldsymbol{\Lambda}' & \mathbf{I}_q \end{bmatrix} \right).$$

McNicholas (2016a) shows the expected values of the joint distribution to yield

$$\mathbb{E}[\mathbf{U}_i \mid \mathbf{x}_i] = \boldsymbol{\beta}(\mathbf{x}_i - \boldsymbol{\mu}) \quad \text{where} \quad \boldsymbol{\beta} = \boldsymbol{\Lambda}'(\boldsymbol{\Lambda}\boldsymbol{\Lambda}' + \boldsymbol{\Psi})^{-1},$$

and

$$\mathbb{E}[\mathbf{U}_i\mathbf{U}_i' \mid \mathbf{x}_i] = \mathbb{V}\text{ar}[\mathbf{U}_i \mid \mathbf{x}_i] + \mathbb{E}[\mathbf{U}_i \mid \mathbf{x}_i]\mathbb{E}[\mathbf{U}_i \mid \mathbf{x}_i]' = \mathbf{I}_q - \boldsymbol{\beta}\boldsymbol{\Lambda} + \boldsymbol{\beta}(\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})'\boldsymbol{\beta}'.$$

The latent factors, $\mathbf{u}_{ig}$, for $i = 1, \ldots, n$ and $g = 1, \ldots, G$ are added to the complete-data log-likelihood and in a similar fashion to the first stage, the expected value of

the complete-data log-likelihood is obtained to be

$$\begin{aligned}
Q_2 = C + \sum_{g=1}^{G}[-\frac{n_g}{2}\log|\boldsymbol{\Psi}_g| - \frac{n_g}{2}\,\mathrm{tr}\{\boldsymbol{\Psi}_g^{-1}\mathbf{S}_g\} \\
+ \sum_{i=1}^{n}\hat{z}_{ig}(\mathbf{x}_i - \hat{\mathbf{u}}_g)'\boldsymbol{\Psi}_g^{-1}\boldsymbol{\Lambda}_g\mathbb{E}[\mathbf{U}_{ig} \mid \mathbf{x}_i, z_{ig}=1] \\
- \frac{1}{2}\,\mathrm{tr}\{\boldsymbol{\Lambda}_g'\boldsymbol{\Psi}_g^{-1}\boldsymbol{\Lambda}_g\sum_{i=1}^{n}\hat{z}_{ig}\mathbb{E}[\mathbf{U}_{ig}\mathbf{U}_{ig}' \mid \mathbf{x}_i, z_{ig}=1]\}] \\
= C + \frac{1}{2}\sum_{g=1}^{G}n_g[\log|\boldsymbol{\Psi}_g^{-1}| - \,\mathrm{tr}\{\boldsymbol{\Psi}_g^{-1}\mathbf{S}_g\} + 2\,\mathrm{tr}\{\boldsymbol{\Psi}_g^{-1}\boldsymbol{\Lambda}_g\hat{\boldsymbol{\beta}}_g\mathbf{S}_g\} \\
- \,\mathrm{tr}\{\boldsymbol{\Lambda}_g'\boldsymbol{\Psi}_g^{-1}\boldsymbol{\Lambda}_g\boldsymbol{\Theta}_g\}],
\end{aligned}$$

where $\hat{\boldsymbol{\beta}}_g = \hat{\boldsymbol{\Lambda}}_g'(\hat{\boldsymbol{\Lambda}}_g\hat{\boldsymbol{\Lambda}}_g' + \hat{\boldsymbol{\Psi}}_g)^{-1}$, $\boldsymbol{\Theta}_g = \mathbf{I}_q - \hat{\boldsymbol{\beta}}_g\hat{\boldsymbol{\Lambda}}_g + \hat{\boldsymbol{\beta}}_g\mathbf{S}_g\hat{\boldsymbol{\beta}}_g'$, and C is a constant with respect to the $\boldsymbol{\Lambda}_g$ and $\boldsymbol{\Psi}_g$. Depending on the constraints that are imposed on $\boldsymbol{\Lambda}_g$ and/or $\boldsymbol{\Psi}_g$, Q_2 can be differentiated with respect to $\boldsymbol{\Lambda}$ and $\boldsymbol{\Psi}^{-1}$ to yield the score functions. From here, McNicholas and Murphy (2010b) show that solving the score functions when set equal to zero, updates for $\hat{\boldsymbol{\Lambda}}^{\text{new}}$ and $\hat{\boldsymbol{\Psi}}^{\text{new}}$ are obtained.

For the EPGMM family, parameter estimation is carried out in an analogous manor for models with PGMM analogues as shown by McNicholas and Murphy (2010b). Using both

$$\begin{aligned}
\hat{\omega}_g = |\hat{\boldsymbol{\Psi}}_g|^{1/p}, \text{ and} \\
\hat{\boldsymbol{\Delta}}_g = \hat{\boldsymbol{\Psi}}_g/|\hat{\boldsymbol{\Psi}}_g|^{1/p},
\end{aligned}$$

updates for each of the eight models can be found. The primary difference in parameter estimation occurs in the second stage of the AECM for the four new models without PGMM analogues. Due to the constraint on $\boldsymbol{\Delta}_g$, where $|\boldsymbol{\Delta}_g| = 1$ a slight change in the calculation of the estimates is required. McNicholas and Murphy (2010b) describe this approach in detail through the use of Lagrange multipliers (Lagrange, 1788; Fraleigh, 1990).

By subtracting the Lagrange multiplier with respect to $|\boldsymbol{\Delta}_g| - 1$ from the complete-data log-likelihood with respect to $\boldsymbol{\Lambda}_g, \omega_g$, and $\boldsymbol{\Delta}_g$ under the appropriate constraints, the Lagrangian can be found. McNicholas and Murphy (2010b) show the Lagrangian can then be differentiated with respect to $\boldsymbol{\Lambda}_g$, ω_g^{-1}, $\boldsymbol{\Delta}^{-1}$, and ι, where ι is used to denote the Lagrangian multiplier, to yield four respective score functions. Setting these score functions to zero and solving will provide the respective parameter updates. See McNicholas and Murphy (2008, 2010b) for more extensive details and examples on parameter estimation of the PGMM and EPGMM families.

2.6.4 Mixture of Infinite Factor Analyzers

A more recent extension of the MFA model is the mixture of infinite factor analyzers (MIFA) model. The MIFA model applies infinite factor analysis (IFA) models (Bhattacharya and Dunson, 2011) to work around the fact that the value of q must be pre-specified and held constant across components in the MFA models. Murphy *et al.* (2020) states that by applying a multiplicative gamma process (MGP) shrinkage prior on the loadings matrix in the IFA model, the degree of shrinking loadings towards zero increases as the column index tends toward infinity allowing infinitely many factors. The MGP shrinkage prior when applied to the parameter expanded

loadings matrix also allows the variable ordering to remain unaffected in the component covariance matrix. Murphy *et al.* (2020) use the joint conjugacy property of the MGP prior to allow for block updates of the loadings matrix where an adaptive Gibbs sampler is used for truncating the infinite factor loadings matrix. Details on the application of the MGP prior, the MIFA model and its properties can be found in Murphy *et al.* (2020).

Chapter 3

Multi-Factor Parsimonious Gaussian Mixture Models

3.1 Introduction

In this chapter, a variational approach for the six of twelve models in the (extended) PGMM family with unconstrained factor loadings that will allow for the number of factors fit per component to vary is developed.

Model-based clustering *via* parsimonious Gaussian mixture models is typically applied to a data set for a given range of components, factors, and models. There is potentially a time consuming number of tasks depending on the range of parameters applied and the size of the data. The total number of tasks in any scenario equals the product of the length of the number of components, factors, and models that are to be fit, i.e., for $G = 2, \ldots, 5$, $q = 1, \ldots, 6$, and $\mathcal{M} \in \{\mathcal{CCC}, \ldots, \mathcal{UUU}\}$ there are $4 \times 6 \times 12 = 288$ triples, where a triple refers to any combination of G, q, and $\mathcal{M}$. Next, consider the 6 of 12 models in which the factor loadings are not constrained across

components. If the number of factors fit for each component is allowed to vary instead of being held constant, this would vastly increase the number of triples. Applying this to the previous scenario would suggest that for a model with five components, the number of factors would now be a vector of length five which can be any combination of $q = 1, \ldots, 6$, repetition included. For each of the six models with unconstrained factor loadings and $G = 5$, there are now $6^5 = 7776$ combinations of $\mathbf{q}$ to fit. Overall, this creates a total number of tasks equal to $56,088$ instead of the original 288 triples. This approach is infeasible if it is to be done in serial as the computational time taken to fit all triples would exponentially increase for larger values of G and ranges of q. To improve computational time, parallel computing techniques are used.

The chapter is outlined as follows. In Section 3.2 six multi-factor PGMMs are proposed. In Section 3.3, the parallel serial and parallel implementation using Julia with various HPC resources are discussed. In Section 3.4 the proposed method is applied in both serial and parallel to multiple real data sets. The chapter is concluded with a comparison of parallel performance and a discussion of the proposed methodology (Section 3.5.2) .

3.2 Methodology

A simple, but effective extension of the PGMM family is to allow a different number of factors to be fit for each component. This extension is only possible in the six of twelve models of the PGMM family for which the factor loadings are not constrained across components. The six models for which the factor loadings are unconstrained are the UCC, UCU, UUC, UUU, UCUU, and UUCU models. The six models with constrained factor loadings and the extension to the six models with unconstrained

factor loadings will herein be referred to as the multi-factor PGMM (MFPGMM) family. Instead of fitting q factors for each these models, where the number of factors would remain the same for each component fitted, $\mathbf{q}$ is now a vector of length G, the number of components to be fit. This gives way to q^G possible factor combinations to be fit for each component size. Estimation of the model parameters for these six models, *via* the AECM algorithm, is nearly analogous to that of the PGMM parameter estimation procedure described by McNicholas and Murphy (2008, 2010b). The primary difference occurs with the factors loadings, $\boldsymbol{\Lambda}_g$, which are no longer a $p \times q$ matrix, but rather a $p \times q_g$ matrix of factor loadings, herein denoted as $\boldsymbol{\Lambda}_{g_q}$. In an elegant fashion, the addition of the new dimension in the factor loading matrix will become void with the calculation of the component covariance structure. The component covariance structure can now be written as

$$\boldsymbol{\Sigma}_g = \boldsymbol{\Lambda}_{g_q}\boldsymbol{\Lambda}_{g_q}^{'} + \omega_g\boldsymbol{\Delta}_g.$$

The following provides a description of the changes in calculations required to compute the estimates for the six aforementioned models. Given that the addition of the new dimension in the factor loading becomes void in the calculation of the covariance structure, there are no changes to the first stage of the AECM algorithm. Jumping ahead to the second stage of the AECM, consider the UCUU model, the complete-data log-likelihood can be written as

$$Q_2 = C + \frac{1}{2}\sum_{g=1}^{G} n_g[p\log\omega_g^{-1} + \log|\boldsymbol{\Delta}^{-1}| - \omega_g^{-1}\operatorname{tr}\{\boldsymbol{\Delta}^{-1}\mathbf{S}_g\}$$
$$+ 2\omega_g^{-1}\operatorname{tr}\{\boldsymbol{\Delta}^{-1}\boldsymbol{\Lambda}_{g_q}\hat{\boldsymbol{\beta}}_{g_q}\mathbf{S}_g\} - \omega_g^{-1}\operatorname{tr}\{\boldsymbol{\Lambda}_{g_q}^{'}\boldsymbol{\Delta}^{-1}\boldsymbol{\Lambda}_{g_q}\boldsymbol{\Theta}_{g_q}\}],$$

where $\hat{\boldsymbol{\beta}}_{g_q} = \hat{\boldsymbol{\Lambda}}'_{g_q}(\hat{\boldsymbol{\Lambda}}_{g_q}\hat{\boldsymbol{\Lambda}}'_{g_q} + \hat{\omega}_g\boldsymbol{\Delta})^{-1}$ and $\boldsymbol{\Theta}_{g_q} = \mathbf{I}_q - \hat{\boldsymbol{\beta}}_{g_q}\hat{\boldsymbol{\Lambda}}_{g_q} + \hat{\boldsymbol{\beta}}_{g_q}\mathbf{S}_g\hat{\boldsymbol{\beta}}'_{g_q}$. The Lagrangian can be written as

$$L(\boldsymbol{\Lambda}_{g_q}, \omega_g, \boldsymbol{\Delta}, \lambda) = Q_2(\boldsymbol{\Lambda}_{g_q}, \omega_g, \boldsymbol{\Delta}) - \iota(|\boldsymbol{\Delta}| - 1),$$

where ι is the Lagrangian multiplier. The Lagrangian can then be differentiated with respect to $\boldsymbol{\Lambda}_{g_q}$, ω_g^{-1}, $\boldsymbol{\Delta}^{-1}$, and ι to yield four respective score functions

$$\begin{aligned}
S_1(\boldsymbol{\Lambda}_{g_q}, \omega_g, \boldsymbol{\Delta}, \iota) &= \frac{\partial L}{\partial \boldsymbol{\Lambda}_{g_q}} = \sum_{g=1}^{G} \frac{n_g}{\omega_g}[\boldsymbol{\Delta}^{-1}\mathbf{S}_g\hat{\boldsymbol{\beta}}'_{g_q} - \boldsymbol{\Delta}^{-1}\boldsymbol{\Lambda}_{g_q}\tilde{\boldsymbol{\Theta}}_{g_q}], \\
S_2(\boldsymbol{\Lambda}_{g_q}, \omega_g, \boldsymbol{\Delta}, \iota) &= \frac{\partial L}{\partial \omega_g^{-1}} \\
&= \frac{n_g}{2}[p\omega_g - \operatorname{tr}\{\boldsymbol{\Delta}^{-1}\mathbf{S}_g\} + 2\operatorname{tr}\{\boldsymbol{\Delta}^{-1}\boldsymbol{\Lambda}_{g_q}\hat{\boldsymbol{\beta}}_{g_q}\mathbf{S}_g\} - \operatorname{tr}\{\boldsymbol{\Lambda}'_{g_q}\boldsymbol{\Delta}^{-1}\boldsymbol{\Lambda}_{g_q}\boldsymbol{\Theta}_{g_q}\}], \\
S_3(\boldsymbol{\Lambda}_{g_q}, \omega_g, \boldsymbol{\Delta}, \iota) &= \frac{\partial L}{\partial \boldsymbol{\Delta}^{-1}} \\
&= \frac{1}{2}\sum_{g=1}^{G} n_g[\boldsymbol{\Delta} - \omega_g^{-1}\mathbf{S}'_g + 2\omega_g^{-1}\boldsymbol{\Lambda}_{g_q}\hat{\boldsymbol{\beta}}_{g_q}\mathbf{S}_g - \omega_g^{-1}\boldsymbol{\Lambda}_{g_q}\boldsymbol{\Theta}'_{g_q}\boldsymbol{\Lambda}'_{g_q}] + \lambda|\boldsymbol{\Delta}|\boldsymbol{\Delta}, \\
S_4(\boldsymbol{\Lambda}_{g_q}, \omega_g, \boldsymbol{\Delta}, \iota) &= \frac{\partial L}{\partial \iota} = |\boldsymbol{\Delta}| - 1.
\end{aligned}$$

Setting these score functions to zero and solving will provide the respective parameter

updates. The parameter updates are as follows:

$$
\begin{aligned}
S_1(\hat{\mathbf{\Lambda}}_{g_q}^{\text{new}}, \hat{\omega}_g, \hat{\mathbf{\Delta}}, \iota) = \mathbf{0} \implies \quad & \hat{\mathbf{\Lambda}}_{g_q}^{\text{new}} &&= \mathbf{S}_g \hat{\boldsymbol{\beta}}_g^{'} \mathbf{\Theta}_{g_q}^{-1}, \\
S_2(\hat{\mathbf{\Lambda}}_{g_q}^{\text{new}}, \hat{\omega}_g^{\text{new}}, \hat{\mathbf{\Delta}}, \iota) = \mathbf{0} \implies \quad & \hat{\omega}_g^{\text{new}} &&= \frac{1}{p} \operatorname{tr}\{\hat{\mathbf{\Delta}}^{-1} \mathbf{S}_g - \hat{\mathbf{\Delta}}^{-1} \hat{\mathbf{\Lambda}}_{g_q}^{\text{new}} \hat{\boldsymbol{\beta}}_{g_q} \mathbf{S}_g\}, \\
\operatorname{diag}\{S_3(\hat{\mathbf{\Lambda}}_{g_q}^{\text{new}}, \hat{\omega}_g^{\text{new}}, \hat{\mathbf{\Delta}}^{\text{new}}, \iota)\} = \mathbf{0} \implies & \hat{\mathbf{\Delta}}^{\text{new}} &&= \frac{1}{n + 2\iota |\hat{\mathbf{\Delta}}^{\text{new}}|} \sum_{g=1}^{G} \frac{n_g}{\hat{\omega}_g^{\text{new}}} \operatorname{diag}\{\mathbf{S}_g - \hat{\mathbf{\Lambda}}_{g_q}^{\text{new}} \hat{\boldsymbol{\beta}}_{g_q} \mathbf{S}_g\}, \\
& &&= \frac{1}{n + 2\iota} \operatorname{diag}\{\sum_{g=1}^{G} \frac{n_g}{\hat{\omega}_g^{\text{new}}} [\mathbf{S}_g - \hat{\mathbf{\Lambda}}_{g_q}^{\text{new}} \hat{\boldsymbol{\beta}}_{g_q} \mathbf{S}_g]\}, \\
S_4(\hat{\mathbf{\Lambda}}_{g_q}^{\text{new}}, \hat{\omega}_g^{\text{new}}, \hat{\mathbf{\Delta}}^{\text{new}}, \iota^{\text{new}}) = \mathbf{0} \implies \quad & |\hat{\mathbf{\Delta}}^{\text{new}}| = 1.
\end{aligned}
$$

Given that $\hat{\mathbf{\Delta}}^{\text{new}}$ is a diagonal matrix with the constraint found by setting S_4 equal to zero, it is found that

$$
\iota = \frac{1}{2} \left[\left(\prod_{j=1}^{p} \xi_j \right)^{\frac{1}{p}} - n \right],
$$

where ξ_j is the jth element along the diagonal of the matrix

$$
\sum_{g=1}^{G} \frac{n_g}{\hat{\omega}_g^{\text{new}}} [\mathbf{S}_g - \hat{\mathbf{\Lambda}}_{g_q}^{\text{new}} \hat{\boldsymbol{\beta}}_{g_q} \mathbf{S}_g].
$$

See McNicholas and Murphy (2008, 2010b) for further details and examples on how parameter estimation is carried out for the PGMM family.

Derivations for the remaining five models with unconstrained factor loadings are all done in a similar fashion. The following gives an idea of what these estimates look like for the remaining models. For Model UUCU, the parameter estimates are given

by

$$\begin{aligned}
\hat{\mathbf{\Lambda}}_{g_q}^{\text{new}} &= \mathbf{S}_g \hat{\boldsymbol{\beta}}_{g_q} \mathbf{\Theta}_{g_q}^{-1}, \\
(\hat{\omega})^{\text{new}} &= \frac{1}{p} \sum_{g=1}^{G} \hat{\pi}_g \operatorname{tr}\{\hat{\mathbf{\Delta}}_g^{-1} (\mathbf{S}_g - \hat{\mathbf{\Lambda}}_{g_q}^{\text{new}} \hat{\boldsymbol{\beta}}_{g_q} \mathbf{S}_g)\}, \\
\hat{\mathbf{\Delta}}_g^{\text{new}} &= \frac{1}{(\hat{\omega})^{\text{new}} (1 + 2\iota_g / n_g)} \operatorname{diag}\{\mathbf{S}_g - \hat{\mathbf{\Lambda}}^{\text{new}} \hat{\boldsymbol{\beta}}_{g_q} \mathbf{S}_g\},
\end{aligned}$$

where

$$\iota_g = \frac{n_g}{2} \left[\frac{1}{(\hat{\omega})^{\text{new}}} \left(\prod_{j=1}^{p} \xi_{gj} \right)^{\frac{1}{p}} - 1 \right],$$

where $\xi_g j$ is the jth element along the diagonal of the matrix $\mathbf{S}_g - \hat{\mathbf{\Lambda}}^{\text{new}} \hat{\boldsymbol{\beta}}_{g_q} \mathbf{S}_g$.

For Model UCC with $\mathbf{\Psi}_g = \mathbf{\Psi} = \psi \mathbf{I}_p$, the parameter estimates are given by

$$\begin{aligned}
\hat{\mathbf{\Lambda}}_{g_q}^{\text{new}} &= \mathbf{S}_g \hat{\boldsymbol{\beta}}_{g_q}^{'} \mathbf{\Theta}_{g_q}^{-1}, \\
(\hat{\omega})^{\text{new}} &= \left| \frac{1}{p} \mathbf{I}_p \sum_{g=1}^{G} \hat{\pi}_g \operatorname{tr}\{\mathbf{S}_g - \hat{\mathbf{\Lambda}}_{g_q}^{\text{new}} \hat{\boldsymbol{\beta}}_{g_q} \mathbf{S}_g\} \right|^{1/p}, \\
\hat{\mathbf{\Delta}}^{\text{new}} &= \frac{1}{|\frac{1}{p} \mathbf{I}_p \sum_{g=1}^{G} \hat{\pi}_g \operatorname{tr}\{\mathbf{S}_g - \hat{\mathbf{\Lambda}}_{g_q}^{\text{new}} \hat{\boldsymbol{\beta}}_{g_q} \mathbf{S}_g\}|^{1/p}} \frac{1}{p} \mathbf{I}_p \sum_{g=1}^{G} \hat{\pi}_g \operatorname{tr}\{\mathbf{S}_g - \hat{\mathbf{\Lambda}}_{g_q}^{\text{new}} \hat{\boldsymbol{\beta}}_{g_q} \mathbf{S}_g\} = \mathbf{I}_p.
\end{aligned}$$

For Model UCU with $\mathbf{\Psi}_g = \mathbf{\Psi}$, the parameter estimates are given by

$$\hat{\mathbf{\Lambda}}_{g_q}^{\text{new}} = \mathbf{S}_g \hat{\boldsymbol{\beta}}_{g_q}^{'} \mathbf{\Theta}_{g_q}^{-1},$$
$$(\hat{\omega})^{\text{new}} = \left| \sum_{g=1}^{G} \hat{\pi}_g \operatorname{diag}\{\mathbf{S}_g - \hat{\mathbf{\Lambda}}_{g_q}^{\text{new}} \hat{\boldsymbol{\beta}}_{g_q} \mathbf{S}_g\} \right|^{1/p},$$
$$\hat{\mathbf{\Delta}}^{\text{new}} = \frac{1}{|\sum_{g=1}^{G} \hat{\pi}_g \operatorname{diag}\{\mathbf{S}_g - \hat{\mathbf{\Lambda}}_{g_q}^{\text{new}} \hat{\boldsymbol{\beta}}_{g_q} \mathbf{S}_g\}|^{1/p}} \sum_{g=1}^{G} \hat{\pi}_g \operatorname{diag}\{\mathbf{S}_g - \hat{\mathbf{\Lambda}}_{g_q}^{\text{new}} \hat{\boldsymbol{\beta}}_{g_q} \mathbf{S}_g\}.$$

For Model UUC with $\mathbf{\Psi}_g = \psi_g \mathbf{I}_p$, the parameter estimates are given by

$$\hat{\mathbf{\Lambda}}_{g_q}^{\text{new}} = \mathbf{S}_g \hat{\boldsymbol{\beta}}_{g_q}^{'} \mathbf{\Theta}_{g_q}^{-1},$$
$$(\hat{\omega})_g^{\text{new}} = \left| \frac{1}{p} \mathbf{I}_p \operatorname{tr}\{\mathbf{S}_g - \hat{\mathbf{\Lambda}}_{g_q}^{\text{new}} \hat{\boldsymbol{\beta}}_{g_q} \mathbf{S}_g\} \right|^{1/p},$$
$$\hat{\mathbf{\Delta}}^{\text{new}} = \frac{1}{|\frac{1}{p} \mathbf{I}_p \operatorname{tr}\{\mathbf{S}_g - \hat{\mathbf{\Lambda}}_{g_q}^{\text{new}} \hat{\boldsymbol{\beta}}_{g_q} \mathbf{S}_g\}|^{1/p}} \frac{1}{p} \mathbf{I}_p \operatorname{tr}\{\mathbf{S}_g - \hat{\mathbf{\Lambda}}_{g_q}^{\text{new}} \hat{\boldsymbol{\beta}}_{g_q} \mathbf{S}_g\} = \mathbf{I}_p.$$

For Model UUU with no constraints imposed, the parameter estimates are given by

$$\hat{\mathbf{\Lambda}}_{g_q}^{\text{new}} = \mathbf{S}_g \hat{\boldsymbol{\beta}}_{g_q}^{'} \mathbf{\Theta}_{g_q}^{-1},$$
$$(\hat{\omega})_g^{\text{new}} = \left| \operatorname{diag}\{\mathbf{S}_g - \hat{\mathbf{\Lambda}}_{g_q}^{\text{new}} \hat{\boldsymbol{\beta}}_{g_q} \mathbf{S}_g\} \right|^{1/p},$$
$$\hat{\mathbf{\Delta}}_g^{\text{new}} = \frac{1}{| \operatorname{diag}\{\mathbf{S}_g - \hat{\mathbf{\Lambda}}_{g_q}^{\text{new}} \hat{\boldsymbol{\beta}}_{g_q} \mathbf{S}_g\}|^{1/p}} \operatorname{diag}\{\mathbf{S}_g - \hat{\mathbf{\Lambda}}_{g_q}^{\text{new}} \hat{\boldsymbol{\beta}}_{g_q} \mathbf{S}_g\}.$$

The MFPGMM models will be applied to real data in Section 3.4 with the results analyzed and compared to the performance of the original PGMM models.

3.3 Implementation

3.3.1 Julia

Before diving into the parallelization technique used throughout this chapter, a broad description of how the technique is being implemented will be given. Work in this thesis was conducted using version 1.3 of Julia (Bezanson *et al.*, 2017; McNicholas and Tait, 2019). Julia is a high-performance, dynamic programming language that is built for numerical analysis. It offers many of the features that a user would expect to see in a statistical computing language such as R (R Core Team, 2019), but also offers the flexibility that one would experience with languages such as C and Python. A key feature of Julia is that it supports distributed computing with or without the use of a message passing interface. A message passing interface has been used to conduct this work for reasons that will become more apparent in Chapter 4. Note, it is possible to accomplish a similar style of parallelization using the `Distributed` package.

3.3.2 HPC Cluster

In order to visualize the forms of communication discussed in the following sections, the generic layout of a HPC is provided. Referring to Figure 3.1, an illustration of a HPC containing a head node and five individual nodes is given. All HPCs have a head node, typically labelled as the rank 0 node, which is not commonly used in computations, but used to distribute the tasks for computation. Although the head node may be of a similar size (available resources, e.g., number of processors) to the worker nodes, only one processor may be needed to distribute such tasks. The individual nodes, which make up the HPC as a whole, each contain a certain number

of processors (and threads) that can be used for computation. Referring to Figure 3.1, each of the five nodes are considered to be comprised of five individual processors. Excluding the head node, all individual nodes have the same system architecture.

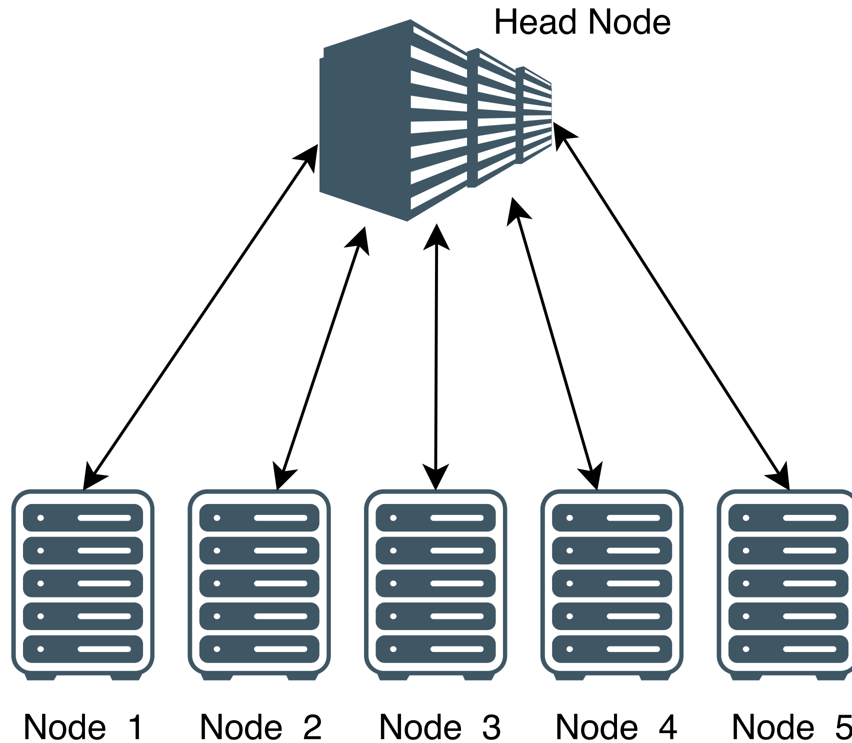

Figure 3.1: The generic layout of a high performance computer.

For the purposes of this thesis, all code has been run on a HPC that is comprised of a head node, which is substantially less powerful than the compute nodes, and 16 compute nodes. The head node is mainly a storage server (Dell PowerEdge R740xd), with two 10-core Intel Xeon Silver 4114 processors and 12×8G DDR4/2400 DIMMs. The 16 compute nodes are dual-socket, Intel® Xeon® Gold 6148 CPUs with a clock speed of 2.40GHz. Each socket contains 20 cores without hyper-threading enabled.

They also contain 12×16G DDR4/2666 memory, a total of 192G, and 100Gb Infiniband. This is part of the PowerEdge C6400 family. Hyper-threading is a feature of some Intel CPUs that allows one physical CPU to appear as if it were two logical CPUs. Each compute node appears to have 80 cores with hyper-threading enabled even though there are only 40 logical cores. Hyper-threading has the potential to negatively impact results because it splits resources by turning physical cores into two virtual cores and as such, will not be considered throughout this chapter as it may only be used in selective cases.

3.3.3 Message Passing Interface

Message passing interface (MPI) is a communication protocol that allows for point-to-point message passing and operations that are sent to a user-specified group of processes. Processes are named by their rank in the relevant group where communication is occurring. A communicator which houses groups and communication content (scoping) information provides important safety measures that are necessary and useful for building up library-orientated parallel code (Gropp *et al.*, 1999). MPI is a dominant form of parallel programming used today for high performance computing (HPC) due to the flexibility it offers in terms of inter-node communication. A simple and very common parallel programming paradigm used in conjunction with MPI is referred to as "master-slave". Using a HPC, where multiple nodes are available, assigns the rank 0 node to be the "master" node and all other nodes to be the "slave" nodes. Communication is costly in terms of time and therefore communication between the master and slave is limited to when the slave is requesting a new job and when the slave is sending results back to the master. There is no communication occurring

between slaves at any point during this process. Refer to Gropp *et al.* (1999) for an in-depth look at MPI.

The description of an HPC cluster was given in Section 3.3.2. Using this description, a simplistic illustration of the MPI parallelism technique can be given. Consider the scenario in which a HPC is tasked to perform a computation on a range of 25 values. The tasks can be completed independently of one another, meaning there are 25 tasks to be completed. A single processor from the head node can evenly distribute the 25 tasks across the HPC where each processor receives its own task, this would also be considered as fine-grain parallelism. The processors will then work simultaneously to complete their task and send the results back to the single processor on the head node. Note that if these jobs were small, say taking the square root of a value, then using this form of parallelization would be extremely costly in terms of communication. Not only would the processors individually requesting a job take a lengthy amount of time, but the processors would also have to individually communicate with the single processor on the head node to return the results. These jobs would be done more efficiently in serial due to the overhead created from the communication required in parallel.

There are two implementations of MPI software, Open-MPI and MPICH. The MPI parallelization technique has been implemented using version 3.2.1 of MPICH which does not support InfiniBand communication. This is primarily due to issues with Unified Communication X (UCX), a widely used communication framework for HPC protocols. Julia developer documents state that the profiler uses SIGUSR2 for sampling, while the garbage collector uses SIGSEGV for thread synchronization. Some up-to-date versions of UCX and Open-MPI, which are not available on the HPC

cluster used, can potentially resolve this issue by disabling UCX from intercepting the SIGSEGV error signal prior to launching the Julia environment. Without the use of InfiniBand, there is the potential for lower throughput and increased latency between MPI communications.

3.3.4 Slurm

The Simple Linux Utility for Resource Management (SLURM), now referred to as the Slurm workload manager or just Slurm, is an open-source job scheduler for Linux. It is used by many HPCs for cluster management and job scheduling. All jobs throughout this chapter are run using Slurm to allocate the number of resources accordingly which is done prior to executing the code. Due to Julia being in the earlier stages of its life, users are unable to add processes (unless using Distributed) and threads (Julia pre version 1.5) at run time. Both the number of processes and threads can be set using Slurm prior to run time in order to have these resources available to Julia at launch.

Slurm is highly advanced and offers the capability to run code in parallel by distributing tasks using its scheduler. This is a feature that is not considered throughout this chapter due to how the tasks must be implemented. Slurm offers multiple ways to run a parallel job that involve the use of an array or job packing. Since parallel code must be executed using the srun command, which executes multiple tasks simultaneously, instead of distributing them individually, the same user input arguments for the MFPGMM code will be read by each process and run multiple times. To prevent this, while still sending each triple to a different process, a significant amount of work would be required prior to the execution of each parallel run.

A Slurm array allows the user to specify a range of job ID values where only a single input value is changed and will be taken as input. In order to make use of this type of parallelization, the job ID would be required to point to a specific triple that the process will take as input arguments. A packed job would involve the user writing a predetermined for loop, containing the input arguments, using the `--exclusive` argument of `srun` to schedule independent processes inside of a Slurm job allocation. The key problem here is that the $\mathbf{q}$ vector is changing in combination and length frequently. The depth of a nested for loop would not be able to change as the vector length increased and if the full range of q is sent, there is no option to keep track of which combination of factors is run, i.e., they would all be run multiple times.

Although the job array approach could be done by writing all input arguments to a large file where each job ID will correspond to a line in the file containing the input arguments, this creates potential issues. There would be a significant amount of idle threads as component sizes increase and new starting values are calculated prior to job execution. Both approaches require use of the `srun` command which executes multiple tasks simultaneously; therefore, each result would be placed in a separate output file. Further work would be required to sort through these output files to obtain the best starting value or result which could be unrealistic depending on the number of triples to run. It is also well known that creating short jobs, approximately ten minutes or less, using an array will incur a large overhead. The implementation described in Section 3.3.3 is able to prevent all of the aforementioned issues from occurring while also allowing the parallel code to be run on a computer that does not have a scheduler installed.

3.3.5 Parallel Design

There are multiple ways to initiate an MPI environment for Julia from the command line. For the purposes of this section, the MPI environment is created using a batch script containing the following commands:

```
#!/bin/bash
#SBATCH --time=7-00:00:00
#SBATCH --nodes="numproc"
#SBATCH --ntasks-per-node="numtask"
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=0
mpiexec julia  driver_file.jl input_variables
```

The batch script example will initialize a job that may take up to seven days with `numproc` nodes and `numtask` tasks per node where each task will use a single CPU. This is equivalent to saying that `numproc` × `numtask` processes will be started, where one CPU acts as the master and the remaining as the slaves. This batch script will also request all available memory on the allocated nodes for each CPU to use.

When using any form of parallelization in Julia, the environment variable for the number of threads to use is set before run time using the Linux command:

```
export JULIA_NUM_THREADS="numthread"
```

where `numthread` represents the value for the desired number of threads to initialize. It is vital to note that some packages in Julia may already be written in parallel. In order to avoid oversubscribing threads which will drastically deteriorate performance, the Linux command:

```
BLAS.set_num_threads(1)
```

must be set at run time to limit the BLAS package to a single thread and allow Julia to handle all parallelization. Unless otherwise stated, we assume the number of threads for the environment outside of Julia, even in serial, to be set to 1 herein as this may otherwise alter the performance.

To find the best member of the PGMM family, the goal is to maximize the BIC over the triple $(\mathcal{M}, G, \mathbf{q})$, where each triple will be sent to a different process (McNicholas *et al.*, 2010). $\mathcal{M}$ represents the model in question, G the number of components, and $\mathbf{q}$, a vector of length G, containing the factors. The proceeding pseudocodes show the design of the parallel implementation. First, consider the function `main()` (Algorithm 5) in Appendix A.

This function will initialize the MPI environment within Julia while obtaining the size of the resources available and ranks of each processor. It then proceeds to define the master (rank 0 processor) which will farm out jobs to slaves, i.e., all processors excluding the rank 0 processor. Once the jobs are complete, the MPI environment waits for all processors to join together before finalizing the environment (required to successfully exit the code).

The `master()` (Algorithm 6) and `slave()` (Algorithm 7) functions found in Appendix A give a rough idea how the MPI process is applied to the methodology presented in Section 3.2. The master process is responsible for communication to the slaves which includes creating an array of all triples to run, distribution of tasks, as well as collecting results and storing them. The slave processes are responsible for the execution of each job and communicating results back to the master.

3.4 Applications

3.4.1 Overview

The methodology for the MFPGMM models developed in Section 3.2, in combination with the MPI implementation discussed in Section 3.3, will now be applied to multiple real data sets. Considering that each triple will be sent to a different process, fine-grain parallelism is being applied for the MPI technique. The performance of the new techniques will be analyzed and compared. In addition, the serial versus parallel performance of each run will then be evaluated by its speed-up. The speed-up can be calculated by taking the serial runtime and dividing by the parallel runtime. In an ideal world, the expected scaling would be one to one, i.e., for each processor added the speed-up would increased by one. Each data set will be run using all 12 MFPGMM models for a range of components and factors. The runs will contain an even number of random starts, half of which will be random using the CUU model and the other half done using k-means, and will be repeated for a specified number of total loops. The results of the MFPGMM models will be compared to that of the MIFA models (Section 2.6.4), using 10 random k-means starts and run for a total of 10 loops, to illustrate the difference in solutions. The MIFA model results will be restricted for only non-empty components as it has the potential to fit multiple empty components which will skew results.

3.4.2 Coffee Data Analyses

The Coffee data (Streuli, 1973, as cited in McNicholas (2016a)) shows 12 chemical properties of 43 coffee beans collected across 29 countries. There are two types of

beans measured, 36 of which are Arabica beans and 7 Robusta beans. The twelve chemical properties collected are shown in Table 3.1.

Table 3.1: Twelve chemical properties of the coffee data.

Water	Bean Weight	Caffeine
Fat	Trigonelline	Mineral Content
pH Value	Extract Yield	Free Acid
Chlorogenic Acid	Neochlorogenic Acid	Isochlorogenic Acid

The MFPGMM models are fit to the Coffee data for $G = 2, \ldots, 4$ components and $q = 1, \ldots, 4$ factors. This is done for 10 random starts, five of which are k-means and the remaining five are random starts using the CUU model. This is repeated for a total of five loops, resulting in 11040 runs. The BIC of -996.64 chose the UCU model as the best model for $G = 2$ components and $\mathbf{q} = [1, 2]$ factors. This suggests that the Arabica bean is best described using a single factor, while the Robusta bean requires two factors. A cross-tabulation of the corresponding MAP classifications versus the true classes can be seen in Table 3.2. An ARI of 1 is found, meaning it achieved perfect classification.

For comparison purposes, under the same run conditions with fixed factors, a BIC of -1007.68 (ΔBIC $= 11.04$) chose a CCUU model with $G = 3$ components and $q = 1$ factors. This resulted in an ARI of 0.43, where all Robusta beans were correctly classified, but Arabica beans were taken to be two separate groups as seen in Table 3.3.

Applying the MIFA models for the same parameters results in a perfect classification (identical solution to Table 3.2 with ARI $= 1$) for each of the 10 runs, selecting a two component model with $\mathbf{q} = [4, 4]$. The notable difference here is that the MIFA approach has fixed factors across components and requires additional factors.

Table 3.2: Cross-tabulation of the MAP classifications (A-B) associated with the selected PGMM with multiple factors against true classes for the Coffee data.

	A	B
Arabica	36	0
Robusta	0	7

Table 3.3: Cross-tabulation of the MAP classifications (A-C) associated with the selected PGMM with fixed factors against true classes for the Coffee data.

	A	B	C
Arabica	25	0	11
Robusta	0	7	0

The initial parallel implementation, using MPI, for this analysis was done using 210 cores, 209 of these cores were considered to be slaves. A speed-up of 24.6 was found by dividing the serial run time of 1920 seconds by the parallel run time of 78 seconds. This result is nowhere near linear, but this is expected due to the increased overhead that would be accumulated from MPI communications considering the data has limited observations and variables meaning calculations will finish more swiftly.

3.4.3 Italian Wine Data Analyses

The Italian Wine dataset is a chemical analysis of 178 wines grown from the same region in Italy, but derived from three different cultivars. The size and cultivars are 59 Barolo, 71 Grignolino, and 48 Barbera. The dataset contains the results of 27 constituents found within each of the three wines, see Table 3.4 for a list of these constituents.

The Italian Wine data was fit for all MFPGMM models with $G = 2, \dots, 5$ components and $q = 1, \dots, 5$ factors over 10 random starts, and repeated for 5 loops. This resulted in a total of 118600 triples run. The BIC of -23280.40 chose a CUU model with $G = 3$ components and $q = 4$ factors as the best model. The MAP classifications, seen in Table 3.5, achieves an ARI of 0.95 which is a near perfect classification

Table 3.4: Twenty-seven constituents of the Italian Wine data.

Alcohol	Sugar-free extract	Fixed acidity
pH	Malic Acid	Uronic acids
Tartaric Acid	Hue	Calcium
Potassium	Alkalinity of ash	Chloride
Phosphate	Magnesium	Total phenols
Nonflavonoid phenols	Flavonoids	Proanthocyanins
OD_{280}/OD_{315} of diluted wines	OD_{280}/OD_{315} of flavonoids	Color intensity
Ash	Glycerol	2-3-butanediol
Total nitrogen	Proline	Methanol

with only three misclassifications.

Table 3.5: Cross-tabulation of the MAP classifications (A-C) associated with the selected PGMM against true classes for the Italian Wine data.

	A	B	C
Barolo	59	0	0
Grignolino	2	68	1
Barbera	0	0	48

Table 3.6: Cross-tabulation of the MAP classifications (A-C) associated with the most frequent MIFA model against true classes for the Italian Wine data.

	A	B	C
Barolo	58	1	0
Grignolino	11	60	0
Barbera	0	0	48

Applying the MIFA models under the same run conditions produces a mean ARI of 0.62. In the 10 runs, an incorrect number of components is chosen in 7 scenarios. The component sizes are frequently taken to be either 2 or 5, where the 5 component scenario typically has 1 or 2 empty components that are ignored for the calculation of the results. The number of factors per component is frequently held at 5 with select scenarios containing one component with only 4 factors. The most frequent result of the 10 runs can be seen in Table 3.6 where an approximate ARI of 0.80 is found.

The MPI implementation was again done using 209 slaves and one master which had a run time of 2672 seconds. The serial implementation had a run time of 245437

seconds resulting in a speed-up of 91.86.

3.4.4 Italian Olive Oil Data Analyses

The Italian Olive Oil dataset contains the percentage composition of fatty acids reported by Forina and Tiscornia (1982) and Forina *et al.* (1983). The eight constituents of the lipid fraction can be seen in Table 3.7.

Table 3.7: Eight fatty acids from the Italian Olive Oil data.

Oleic acid	Arachidic acid	Stearic acid
Palmitic acid	Linoleic acid	Eicosenoic acid
Palmitoleic acid	Linolenic acid	

The data contains 572 samples of Italian olive oils that have been collected from three regions, 323 from Southern Italy, 98 from Sardinia, and 151 from Northern Italy. Each of these regions comprises a total of nine different areas. The 323 samples from Southern Italy are composed of 25 from North Apulia, 56 from Calabria, 206 from South Apulia, and 36 from Sicily. The Sardinia samples are split into 65 from Inland Sardinia and 33 from Coastal Sardinia. Finally, Northern Italy is made up of 51 samples from Umbria, 50 from East Liguria, and 50 from West Liguria.

The Italian Olive Oil data will be explored through a cluster analysis on both the aforementioned regions and areas. For the olive oil sorted by region, all 12 models were fitted for $G = 2, 3$ components, $q = 1, \ldots, 8$ factors, eight random starts, and five loops. This created a total of 18400 triples. The BIC of -5557.56 chose a UCU model with $G = 3$ components, and $\mathbf{q} = [4, 7, 7]$ factors. This implies that the Southern Italy olive oil can be described with three fewer factors than the Sardinia and Northern Italy olive oils. Referring to the classification table, see Table 3.8, an

ARI of 1 is achieved. Note, even though models with fixed factors also achieves an ARI of 1, the BIC is slightly worse at -5586.25 (ΔBIC $= 28.69$).

When applying the MIFA models under the same conditions a mean ARI of 0.87 is found. The MIFA models commonly chose the incorrect component size of $G = 2$ and hold the number of factors constant at $q = 8$. Two cases did produce an ARI of 1 when a three-component model was chosen using $\mathbf{q} = [8, 8, 3]$ factors. Using 210 cores, a speed-up of 61.03 is achieved from a serial run time of 42842 seconds and a parallel run time of 702 seconds.

Table 3.8: Cross-tabulation of the MAP classifications (A-C) associated with the selected PGMM against true classes for the Italian Olive Oil data by region.

	A	B	C
Southern Italy	323	0	0
Sardinia	0	98	0
Nothern Italy	0	0	151

Table 3.9: Cross-tabulation of the MAP classifications (A-B) associated with the most frequent MIFA model against true classes for the Italian Olive Oil data by region.

	A	B
Southern Italy	323	0
Sardinia	0	98
Nothern Italy	0	151

In a similar fashion, the MFPGMM models are fitted for $G = 9$ components, $q = 1, \ldots, 3$ factors, six random starts, and two loops resulting in 236268 runs. The BIC of -4573.23 chose a UUU model with $\mathbf{q} = [2, 1, 2, 2, 2, 3, 1, 3, 3]$ factors. The classification table (Table 3.10) shows the results of classifying the olive oils by area and achieves an ARI of 0.77. When clustering the olive oils by area with fixed factors, a more significant difference is seen. A BIC of -5550.35 (ΔBIC $= 977.12$) chose a UUU model with $q = 3$ factors, resulting in an ARI of 0.62.

Applying the MIFA models to the Italian Olive Oil when classified by area had a mean ARI of 0.76. In eight runs $\mathbf{q} = [3, 3, 3, 3, 3, 3, 2, 3, 3]$ factors were chosen, while the remaining two runs selected 2 factors for the fourth component. Although the

number of components was fixed at $G = 9$, the MIFA model produced two to three empty components in all 10 runs. The most frequent result can be seen in Table 3.11 with three empty components achieving an approximate ARI of 0.78.

Table 3.10: Cross-tabulation of the MAP classifications (A-I) associated with the selected PGMM against true classes for the Italian Olive Oil data by area.

	A	B	C	D	E	F	G	H	I
North Apulia	23	2	0	0	0	0	0	0	0
Calabria	0	55	1	0	0	0	0	0	0
South Apulia	0	1	186	3	0	16	0	0	0
Sicily	9	13	3	11	0	0	0	0	0
Inland Sardinia	0	0	0	0	65	0	0	0	0
Coastal Sardinia	0	0	0	0	33	0	0	0	0
East Liguria	0	0	0	0	0	0	47	3	0
West Liguria	0	0	0	0	0	0	23	27	0
Umbria	0	0	0	0	0	0	10	0	41

Table 3.11: Cross-tabulation of the MAP classifications (A-F) associated with the most frequent MIFA model against true classes for the Italian Olive Oil data by area.

	A	B	C	D	E	F
North Apulia	25	0	0	0	0	0
Calabria	55	1	0	0	0	0
South Apulia	11	195	0	0	0	0
Sicily	35	1	0	0	0	0
Inland Sardinia	0	0	0	65	0	0
Coastal Sardinia	0	0	0	33	0	0
East Liguria	0	0	0	0	39	11
West Liguria	0	0	0	0	0	50
Umbria	0	0	48	0	3	0

When clustering the olive oil by region, due to the fact that there are nine regions, the large number of triples run prevents this task from being run in serial within a reasonable amount of time. As it will be discussed in Section 3.5, the size of the data and parallel run time under similar conditions gives a general idea of how long the

serial run time will be. The parallel run time for this analysis was 16730 seconds or 4.65 hours. If this was to achieve a reasonably low estimated speed-up of around 100, the resulting serial run time would be 19.36 days. A perfect speed-up of 209 would suggest the run time in serial would take 40.47 days.

3.4.5 Alon Colon Cancer Data Analyses

The Alon dataset (Alon *et al.*, 1999) contains the gene expression data for 62 samples from colon-cancer patients that were analyzed with an Affymetrix oligonucleotide Hum6000 array. A subset of 461 genes, predetermined by McNicholas and Murphy (2010b), are used from the 6500 genes in the data for the original study. This dataset contains five different clusterings as stated by McNicholas and Murphy (2010b) in correspondence with McLachlan *et al.* (2002). Two classifications are considered in these analyses, the first clustering being where tissues samples are classified by the type of tissue, i.e., tumor or normal. There are 40 tumor samples and 22 normal samples. The second clustering comes from a change of protocol during the experiment(Getz *et al.*, 2000; McLachlan *et al.*, 2002). In this clustering, tissue samples $1-11$ and $41-51$ were collected from the first 11 patients using a poly detector. The remaining 40 samples were collected from patients using total extraction of RNA.

The Alon Colon Cancer data was fitted with all MFPGMM models for $G = 2, \dots, 3$ components, $q = 1, \dots, 8$ factors, six random starts, and two loops resulting in 7296 triples. The BIC of -70181.93 chose a CUU model with $G = 2$ components and $q = 8$ factors as the best model. Upon applying the MIFA models for the same parameters, a two-component model MIFA model with $q = 8$ factors held constant across components was chosen. Both the MFPGMM and MIFA models show poor

performance when clustering tissue samples by type with ARI values near 0.

When classified by extraction technique, an improvement in clustering can see in the results. Table 3.12 shows an ARI of 0.29 is achieved where all samples extracted by poly detector are classified perfectly and approximately two-thirds of the samples taken using total extraction of RNA are correctly classified. Comparing this to the classification estimates from the chosen MIFA model, an average ARI of 0.13 is found from the results in Table 3.13.

Table 3.12: Cross-tabulation of the MAP classifications (A-B) associated with the selected PGMM against true classes for the Alon Colon Cancer data classified by extraction technique.

	A	B
Total extraction of RNA	26	14
Poly detector	0	22

Table 3.13: Cross-tabulation of the MAP classifications (A-B) associated with the most frequent MIFA model against true classes for the Alon Colon Cancer data classified by extraction technique.

	A	B
Total extraction of RNA	21	19
Poly detector	0	22

Again, using 210 processors, a speed-up of 115.63 was achieved where the parallel run time was 1837 seconds and the serial run time was 212407 seconds.

3.4.6 Golub Data Analyses

The Golub dataset (Golub *et al.*, 1999) contains the gene expression data for 72 tissue samples from acute leukemia patients that were analyzed using an Affymetrix array. From these samples, 47 are acute lymphoblastic leukaemia tissues and 25 are acute myeloid leukaemia tissues. A subset of 2030 genes, predetermined by McNicholas and Murphy (2010b), are used from the 7129 genes in the original data.

The Golub data was fitted with all MFPGMM models for $G = 2$ components,

$q = 1, \ldots, 10$ factors, 10 random starts, and one loop. This created a total of 760 triples run. Note that the number of components here is fixed because McNicholas and Murphy (2010b) state that the BIC is not effective for estimating G in high-dimensional applications because the penalty term dominates. The BIC of -408539.61 chose a CUC model with $G = 2$ components and $q = 4$ factors as the best model. Referring to Table 3.14, an ARI of 0.84 was found. Although a worse BIC value, $\Delta\text{BIC} = 5869.77$, the exact same performance can be seen in the two-component UUC model with $\mathbf{q} = [1, 1]$. This is however not the best choice as the second best model with a BIC of -408996.64 ($\Delta\text{BIC} = 457.03$) chose a CCC model with $G = 2$ components and $q = 4$ factors, obtaining an ARI of 0.89.

A significant decrease in performance can be seen from the chosen MIFA model (Table 3.15) for this particular scenario. A mean ARI of 0.21 is found where a two-component model is chosen with $q = 10$ factors held constant across components in all cases.

Table 3.14: Cross-tabulation of the MAP classifications (A-B) associated with the selected PGMM against true classes for the Golub data.

	A	B
ALL	45	2
AML	1	24

Table 3.15: Cross-tabulation of the MAP classifications (A-B) associated with the most frequent MIFA model against true classes for the Golub data.

	A	B
ALL	30	17
AML	2	23

With 209 slaves and one master process, a parallel run time of 4971 seconds was achieved. A serial run time of 271336 seconds was found, resulting in a speed-up of 54.58.

3.4.7 Breast Cancer Data Analyses

The Wisconsin Diagnostic Breast Cancer (WDBC) dataset (Street *et al.*, 1993) contains information for 569 tumors samples that are classified as either malignant or benign. There exists 357 benign and 212 malignant instances in the data. The data are clustered using a subset of the variables from the original study. The variables used are five real-valued features of the cell nucleus being radius (mean of distances from center to points on the perimeter), texture (standard deviation of gray-scale values), perimeter, area, and smoothness (local variation in radius lengths).

This data is used as a comparison between the techniques used and is not considered for performance. The WDBC data was again fitted with all MFPGMM models for $G = 2, 3$ components and $q = 1, \ldots, 4$ factors over two random starts and two loops. Note that fitting $q = 4$ factors requires that the assumption $(p-q)^2 > p+q$ be relaxed as the MIFA models typically require a wider range of factors in implementation. The BIC of -9227.03 chose a three-component UCU model with $\mathbf{q} = [2, 3, 4]$ factors, resulting in an approximate ARI of 37.26 seen in Table 3.16. When compared to the PGMM approach with fixed factors, results are nearly identical where a three-component UCU model is chosen with $q = 3$ factors. There is a total of 4 different classifications resulting in an approximate ARI of 36.87

Table 3.16: Cross-tabulation of the MAP classifications (A-C) associated with the selected PGMM against true classes for the WDBC data.

	A	B	C
Malignant	163	47	2
Benign	8	196	153

Table 3.17: Cross-tabulation of the MAP classifications (A-C) associated with the most frequent MIFA model against true classes for the WDBC data.

	A	B
Malignant	167	45
Benign	11	346

Comparing this to the MIFA approach, the results can be seen in Table 3.17. A mean ARI of 63.95 is achieved across 10 runs with $G = 2$ components and $q = 4$ factors held constant. The improvement in ARI is due to the selection of a two-component model over a three-component model. Although unavailable, the third component found in the MFPGMM model may be due to an underlying type of benign tumor because the groups are fairly distinct overall.

3.5 Discussion

3.5.1 Parallel Run Time Comparison

Throughout Section 3.4, the serial run time and parallel run time using MPI with 209 slaves was given. The speed-up when applied to real data alongside the efficiency with the addition of new cores was also explored. Tables B.1, B.2, B.3, B.4, & B.5 in Appendix B show the run time in seconds, the speed-up from serial, and the efficiency using 90, 150, 210, 270, 330 cores alongside some additional trials. A comparison between all datasets can be seen in Figures 3.2, 3.3, & 3.4. All figures show the same generic pattern; more cores will result in a higher speed-up, but a lower efficiency. The biggest difference can be seen in Table B.2, where going from 90 to 330 cores saves just over an hour of run time. This brings up the question, what is more important, the speed-up or the efficiency? If resources are not of concern, using a substantial amount of cores will almost always provide a greater speed-up, but this does have a limit.

Consider the Coffee data, a very small data set, where 11040 triples are run. Minor speed-ups are seen from 90 all the way to 330 cores, but at 420 cores the speed-up

has decreased. Using only 5 cores for the same amount of work results in a speed-up of 2.5, but has a greater efficiency considering it has completed in less than half the time of the serial run. In all scenarios, even the Golub data with 72 tissue samples and 2030 genes, it requires 90 or less cores to achieve an efficiency of 37.55% or more meaning that even with minor resources a significant amount of time can be saved.

Next, consider the Italian Wine data with 178 wines, 27 constituents, and 118600 triples. Speed-ups can be seen from 90 to 980 (using hyper-threading) cores, but with little gain. From 420 to 980 cores, only 224 seconds are saved. This is primarily due to the increased communication cost by adding more cores. The master process is unable to send and receive information significantly faster with 980 cores than what occurred with 420 cores.

Overall, there are many variables to consider when trying to choose the optimal number of MPI processes to start. Variables such as the size of the data which may caused increased communication time, the time per iteration, the number of iterations which is large dependent on the starting values, the resources available, and the number of triples run can all play large factors in this decision. Efficient speed-ups can be seen with little to no resources, but significantly reductions in run time can be brought about by larger resources. The examples discussed within this chapter show little correlation in regards to determining an optimal strategy as many factors can not yet be predetermined.

3.5.2 Discussion

A new technique in which the number of factors per component may vary for PGMMs where the factor loadings are unconstrained has been developed in Section 3.2. The

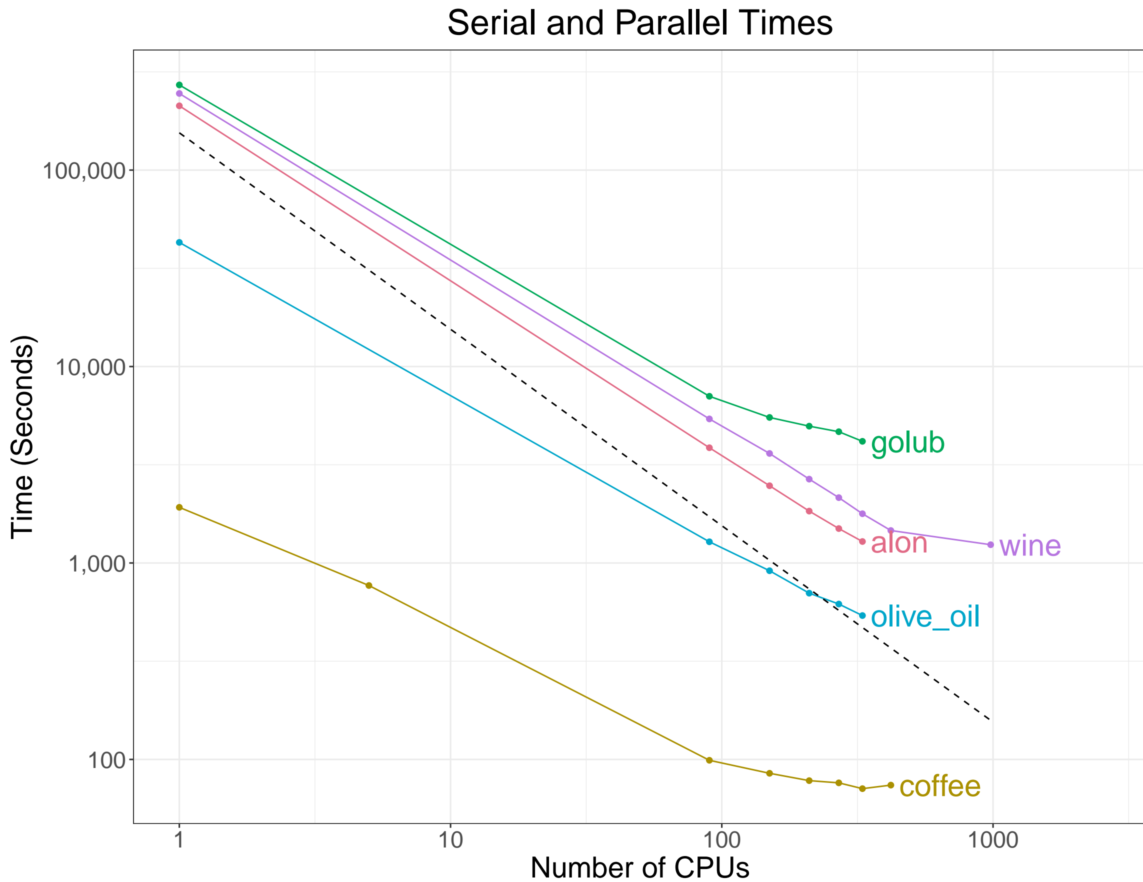

Figure 3.2: The serial and parallel runtimes for Coffee, Wine, Olive Oil, Alon, and Golub datasets plotted against the dashed-line for ideal scaling using the mean average serial runtime.

technique has been applied to six real data sets in Section 3.4 where it performed as well as, if not better than, the original models with fixed factors for all components under the same random starts and run conditions. This is to say that although the new models were chosen in four of seven run scenarios, a model with constrained factor loadings can still be chosen if it better represents the data. The technique was implemented in both serial and parallel scenarios to show how it may be unrealistic to

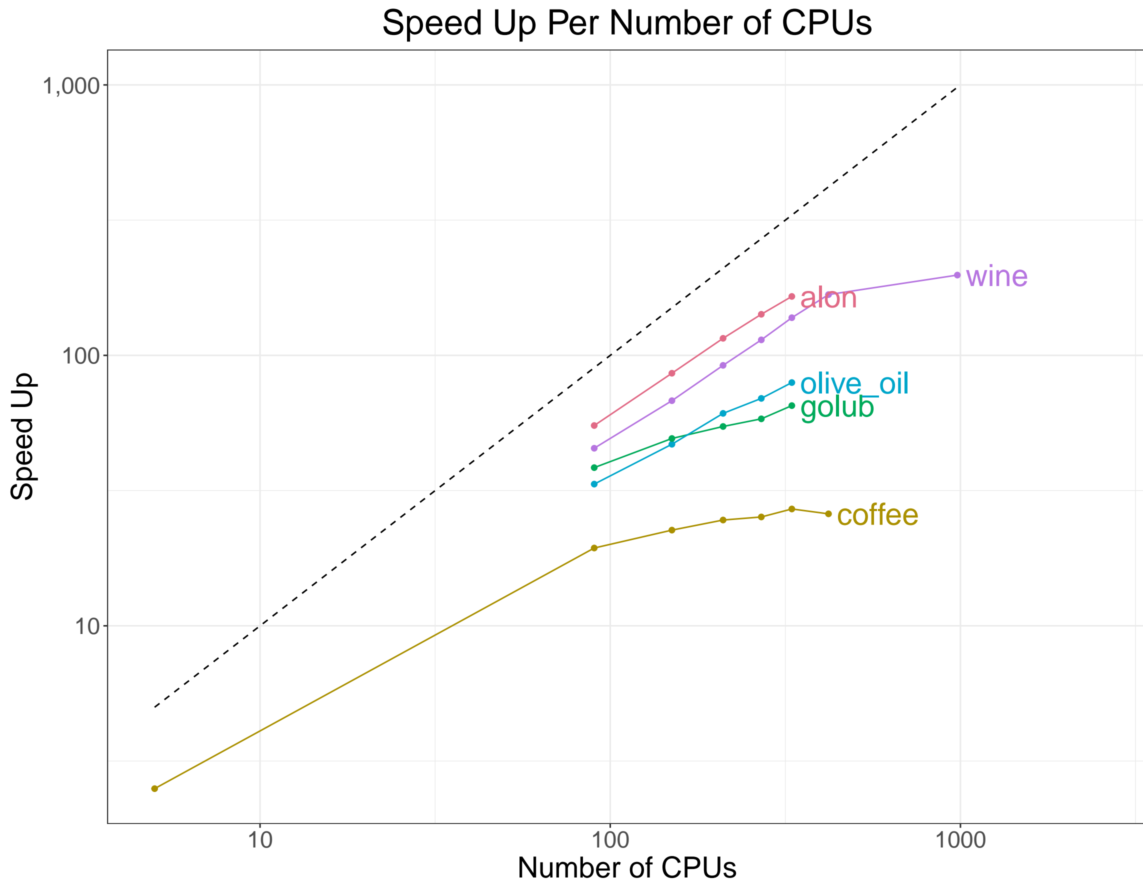

Figure 3.3: The speed-up per number of cores for Coffee, Wine, Olive Oil, Alon, and Golub datasets plotted against the dashed-line representing linear speed-up.

run a large number of triples on a large data set without having more resources available, but even a minor amount of resources can reduce the run time by a significant amount.

The technique was compared to the MIFA approach and outperformed it in all but one scenario. The drawback to the MIFA approach, although more time efficient, is that it commonly selects the incorrect number of components and will frequently fit empty components, even though this can be restricted. In addition, the number of factors required by each component is almost always overestimated.

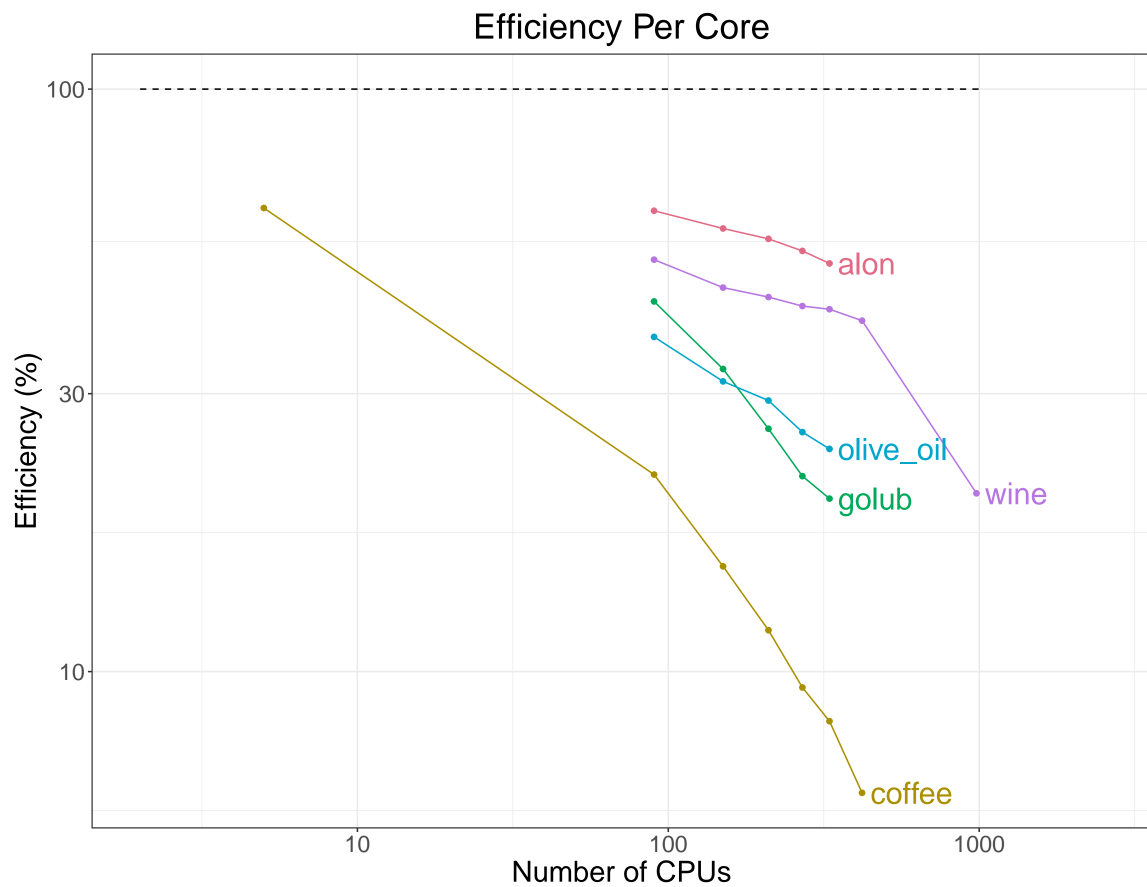

Figure 3.4: The efficiency per number of cores for Coffee, Wine, Olive Oil, Alon, and Golub datasets plotted against the dashed-line representing perfect efficiency.

Chapter 4

Hybrid Parallelization of PGMMs

4.1 Introduction

New types of big data along with the rapidly growing literature on model-based clustering techniques bring computational performance and efficiency into question. Parallel computing allows for the execution of multiple calculations to occur simultaneously. This is commonly done in programming languages through the use of a MPI, described in Chapter 3 and Open Multi-Processing (OpenMP), but rarely done using a hybrid of both MPI and OpenMP. Model-based clustering techniques where models can be run independently is a great example of an approach that can take full advantage of hybrid parallelization. Motivation for this idea comes from model-based clustering *via* parsimonious Gaussian mixture models (McNicholas and Murphy, 2008) where each triple can be run independently of each other.

4.2 Implementation

4.2.1 OpenMP

A well known method of parallel programming is Open Multi-Processing (OpenMP), an application programming interface which supports multi-platform shared memory multiprocessing programming. OpenMP allows for users to parallelize their code without having to rewrite their code entirely. OpenMP is an implementation of multi-threading, meaning a parent thread forks a number of predetermined children threads and divides jobs between them. A fork refers to the operation of a process creating a copy of itself which becomes a child process of the calling process. Although OpenMP is much easier to implement in terms of code as it does not require carefully constructed communication such as MPI, it has potential disadvantages in large scale scenarios due to shared memory. For a detailed look at OpenMP refer to Chapman *et al.* (2007).

Referring to Figure 4.2, OpenMP communication can be thought of as what would occur within each node excluding the head node. OpenMP must take place on individual nodes as MPI communication would be required to send additional tasks outside of the node. Consider the previously described scenario where 25 tasks must be completed. OpenMP would consist of a processor forking itself to all other processors, this can be thought of as the processor creating duplicates of itself. Processors are typically considered to be threads in a HPC scenario, although, this is not always the case. Each processor that is part of the fork would take a portion of the 25 values and complete the task before saving the result, commonly to a shared array. In the case where there are more tasks than processors, once a processor has completed a

specific task, it will take a new task from the shared memory space and continue this process until all tasks are completed. Theoretically, this form of parallelization should occur much faster than MPI parallelization because tasks can be plucked from shared memory. The disadvantage to this technique is that it is limited by the number of processors and memory contained on a single node as well as the parts of the global memory space that can be accessed concurrently.

4.2.2 Multithreading in Julia

Julia offers a variety of multi-threading implementations as part of its base package and user-written packages. In the example given in Section 4.2.1, it was mentioned that each processor would take a portion of the 25 values. The way in which the portion is taken refers to the type of scheduling used by the multi-threading implementation. There are two types of scheduling, namely static and dynamic. Static scheduling would mean that all jobs are allocated at compilation time, without any knowledge regarding the length of the job, and are evenly distributed across the given resources. If there are 25 jobs and 5 processors, each processes will be given five jobs regardless of size or duration. Dynamic scheduling is a scheduling algorithm where jobs are allocated at execution time in order to reduce wait times and optimize resources. This can be thought of as being on a first-come first-serve basis for new work when current jobs are completed and returned.

In version 1.3 of Julia there are two multi-threading macros that are part of the base packages. These macros are `@threads` and `@spawn` which are both considered experimental in this version (`@threads` is no longer considered experimental in more recent versions). The `@threads` macro is a simplistic implementation of a parallel

for loop which currently only offers static scheduling. In order to take advantage of a scheduling system similar to dynamic scheduling, the `@spawn` macro can be used. This macro will spawn a given task on any available thread, but is primarily built for operations that require nested parallelism where asynchronous events must be spawned and have an impending call to `wait` for the result of this task. In order to turn this into a dynamic scheduling where each of the tasks are independent, the `@spawn` macro can be wrapped inside a synchronous for loop so that one task per thread will be spawned for the given number of threads and will complete all tasks before joining back to the main thread. The following pseudocode gives a simple example of this. A similar implementation of this pseudocode is used in the multithreading implementation as it prevents any race conditions from occurring between threads because variables created within threads are private and results are saved to different memory locations. A race condition would occur if multiple threads attempted to alter a piece of memory at the same time.

Algorithm 1: Dynamic scheduling with `@spawn`

```
Function Main():
    Create array of tasks
    @sync begin
        for i in 1:number of tasks
            Threads.@spawn begin
                results = do work here
                save_results[i] = results
            end
        end
    end
```

Using the Alon dataset described in Section 3.4 and fitting 240 triples as described in Section 4.4.2, Figure 4.1 shows the workload distribution per thread by comparing

the iterations completed using `@spawn` and `@threads`.

Figure 4.1: Iterations per thread using different base multithreading implementations in Julia.

There are other packages available in Julia that may complete such tasks, but have the potential to be deprecated with upcoming versions of Julia. The `ThreadPools` package is a great example of this. This package exposes three similar macros that mimic the actions of Julia's base threading module. These macros include the `@bthreads` macro which is a mirror of `@threads` where the main thread is free, the `@qthreads` macro which is a dynamic scheduling version of `@threads` for non-uniform jobs, and the `@qbthreads` macro which is again dynamic scheduling with the main thread free.

There has been limited work done impacting the overall performance of multi-threading between version 1.3.0 and 1.5.2, and many features remain experimental. Reasons to maintain the use of version 1.3.0 will become apparent in the following sections. Notable benefits from the more recent versions include the ability to add threads at run time, which would prevent idle threads on the master node, and the ability to set scheduling in the `@threads` macro. An important detail to note is that `@threads` scheduling still only offers static scheduling, but plans for dynamic scheduling have been proposed by Julia developers.

4.3 Methodology

4.3.1 Hybrid Parallelization

Perhaps it is clear at this point why a hybrid of the aforementioned parallel methods is ideal. A hybrid of MPI and OpenMP offers a "best of both worlds" scenario. The hybrid approach is best suited for HPC clusters where MPI is used for parallelism across nodes and OpenMP is exploited within the node. Hybrid parallelization allows for the combination of distributed memory parallelization on the node inter-connect with shared memory parallelization inside of each node, i.e., taking advantage of the ideal attributes from both parallelization techniques.

Referring to Figure 4.2, hybrid parallelization is illustrated as it would occur on a HPC. Again, consider the previously mentioned scenario where there are 25 tasks to be completed. In the hybrid parallelization approach, these tasks can quickly be divided up into groups of five by the head node which would then be sent to individual nodes using MPI parallelization. Once the compute node receives one of the five tasks

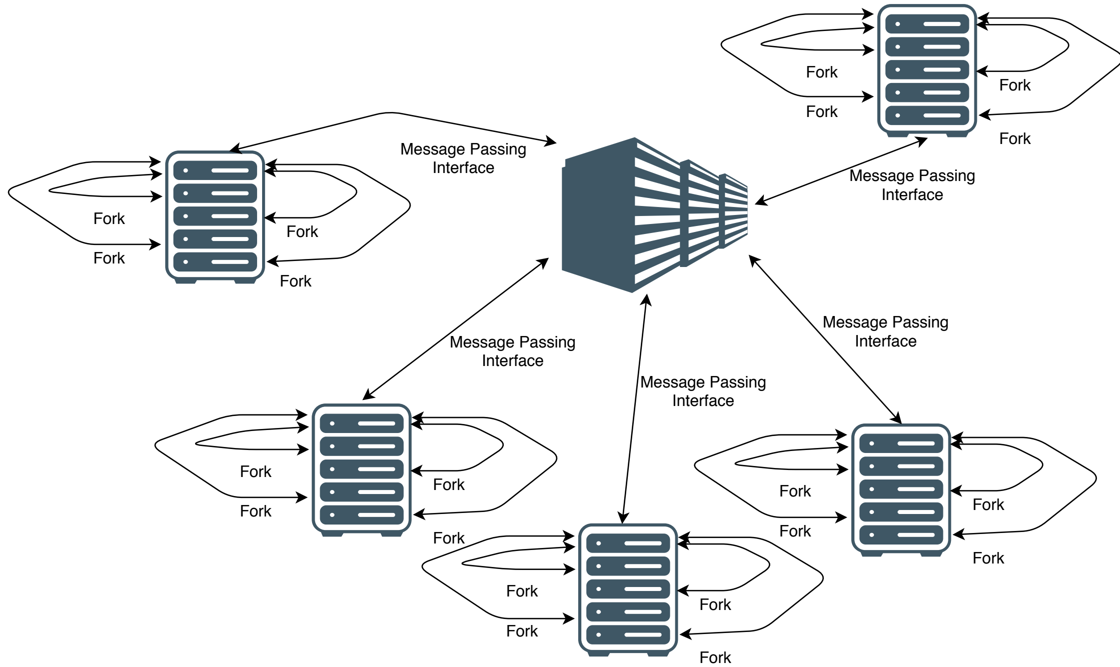

Figure 4.2: Hybrid MPI and OpenMP communication occurring within a HPC.

it would use OpenMP parallelization to complete all five tasks simultaneously and only have to send one message back, e.g., a shared array containing the results, to the head node. Next, this approach is described as it would occur in a model-based clustering scenario *via* PGMMs.

It was mentioned in Section 3.3.4 that the user is unable to add processes using the `MPI` package and threads using the `Base.Threads` package at run time. Due to this, the number of processes and threads are set using Slurm prior to launching Julia which does create a minor problem. The master process, even though it only requires one process and no threads, will still be allocated the same number of threads as each of the slaves. These threads, which are actually CPUs, will remain idle throughout the computation and potentially hinder the hybrid performance slightly. This is to say that in a hybrid scenario given N processes and T threads, $N-1$ processes will

use all available threads, while one process (the master) will have $T-1$ idle threads. Until such environment variables can be set after run time, there is no alternative to this option.

To reiterate from Chapter 3, McNicholas *et al.* (2010) states that to find the best member of the PGMM family, the goal is to maximize the BIC over the triple $(\mathcal{M}, G, q)$, where each triple will be sent to a different process. The pseudocode in Appendix A can be thought of as containing three important steps of the hybrid parallelization process. The first step being the function of the rank zero node on the HPC cluster. This node takes the job of the master and will provide jobs to slaves who are free in addition to collecting results from these slaves. The second step being the nonzero rank nodes who take the job of a slave. The slaves send tags to the master indicating that they are either free and requesting a job or have completed a job and are returning results. Combining steps one and two are what makes up MPI parallelization. The third step is what occurs within the slave nodes. Here, once the slave has received a job, OpenMP parallelism would occur within the compute node using Algorithm 1 where each individual task from the for loop will be spawned on a processor once it becomes available. The master process that receives the information sent *via* MPI will then fork itself to all available threads. Note, each processor on a compute node is treated as a thread. The MPI parallelization sends a range of $\mathcal{M}$ and G to the slave, but only one specific q. The master process on the slave node then duplicates itself with a total of $\mathcal{M} \times G$ jobs to available processes. The master process on the slave node will then gather all results upon completion and return the best result to the master process on the head node. Note, the pseudocode in Appendix A can be completed by using a probe command on slave nodes to see when they have

results to return or are free to start a new task, but is instead accomplished using an open call to receive.

4.4 Applications

4.4.1 Overview

The hybrid parallelization technique described in Section 3.3.3 will now be applied to two real data sets and have its performance analyzed. For each analysis, OpenMP, MPI, and the hybrid approach will be run. MPI will be run for 40, 80, 120, 160, 200, and 240 processes. OpenMP will be run for 5, 10, 15, and 20 threads (processors). The hybrid approach will then be run for 3, 5, 7, 9, and 11 processes each with 5, 10, and 15 threads. For both the MPI and hybrid approach with N processes, there is one master process and $N-1$ slaves.

The performance of each run will then be evaluated using speed-up. In the case of hybrid parallelization where there exists idle threads, speed-up will be calculated excluding these threads. The speed-up of each technique will be graphed and superimposed with linear speed-up. Linear speed-up is an ideal scenario where the addition of each slave process/thread would add an additional speed-up of one.

4.4.2 Alon Data Analyses

The Alon dataset described in Section 3.4 is used again for analyses. PGMMs are fit for $G = 2, 3$, $q = 1, \dots, 10$, and $\mathcal{M} \in \mathcal{CCC}, \dots, \mathcal{UUU}$. This implies that there are $2 \times 10 \times 12$ or 240 tasks to be completed in total. OpenMP will complete tasks for the number of threads given while storing results in a shared array and plucking new

tasks from the shared memory. MPI will complete tasks for the number of processes given before sending results back to the master and requesting a new job. While for the hybrid approach with N processes, $N-1$ tasks, for a total of q, will be requested through MPI communication and 2×12 or 24 tasks will be completed using OpenMP by each slave node.

The serial time taken to run `pgmm` on the Alon data is approximately 2084 seconds or 35 minutes. Referring to Table 4.1, the results of applying MPI parallelization to run `pgmm` on the Alon data can be seen. The best result found under MPI parallelization uses 240 processes, i.e., a master and 239 slaves. The time taken is 193 seconds which results in an approximate speed-up of 10.8. Referring to Table 4.2, the results can be seen when run using OpenMP parallelization. An approximate speed-up of 3.6 is found using 20 threads. For comparative purposes, using these 20 processors under MPI parallelization results in a time of 408 seconds and a speed-up of 5.1. Note, under OpenMP parallelization, using all of the logical cores on a given node does not produce superior results and is not always possible. Using 25 threads is not a possibility for the given scenario due to memory consumption.

The performance from the application of the hybrid approach can be seen in Table 4.3. With similar overall resources sizes, a comparable, yet slightly better, speed-up to that of MPI can be achieved in some hybrid scenarios. The best result, being 183 seconds, is found using 11 processes and 15 threads. The speed-up of this hybrid scenario is 11.39. A comparison of all results can be seen in Figure 4.3.

Note that the serial time taken using R is approximately 2.29 hours. For comparative purposes, a plot is also shown of the superimposed speed-up values as they would be viewed in comparison to the `pgmm` package in R, see Figure 4.4. It can be

Table 4.1: Time taken using MPI parallelization to run pgmm on the Alon data for different numbers of processes.

MPI Time Results	
Number of Processes	Time in Seconds
40	284
80	228
120	233
160	204
200	222
240	193

Table 4.2: Time taken using OpenMP parallelization to run pgmm on the Alon data for different numbers of threads.

OpenMP Time Results	
Number of Threads	Time in Seconds
5	796
10	637
15	587
20	579

Table 4.3: Time taken in seconds using hybrid parallelization to run pgmm on the Alon data for different numbers of slaves/threads.

Hybrid Parallelization Time Results				
Number of Slaves \ Number of Threads	5	10	15	20
2	506	410	373	401
4	298	257	236	241
6	265	211	219	201
8	236	193	187	188
10	193	195	183	183

seen that the speed-up is actually hyper-linear for OpenMP and the hybrid model with two slaves when a smaller number of processes/threads are present.

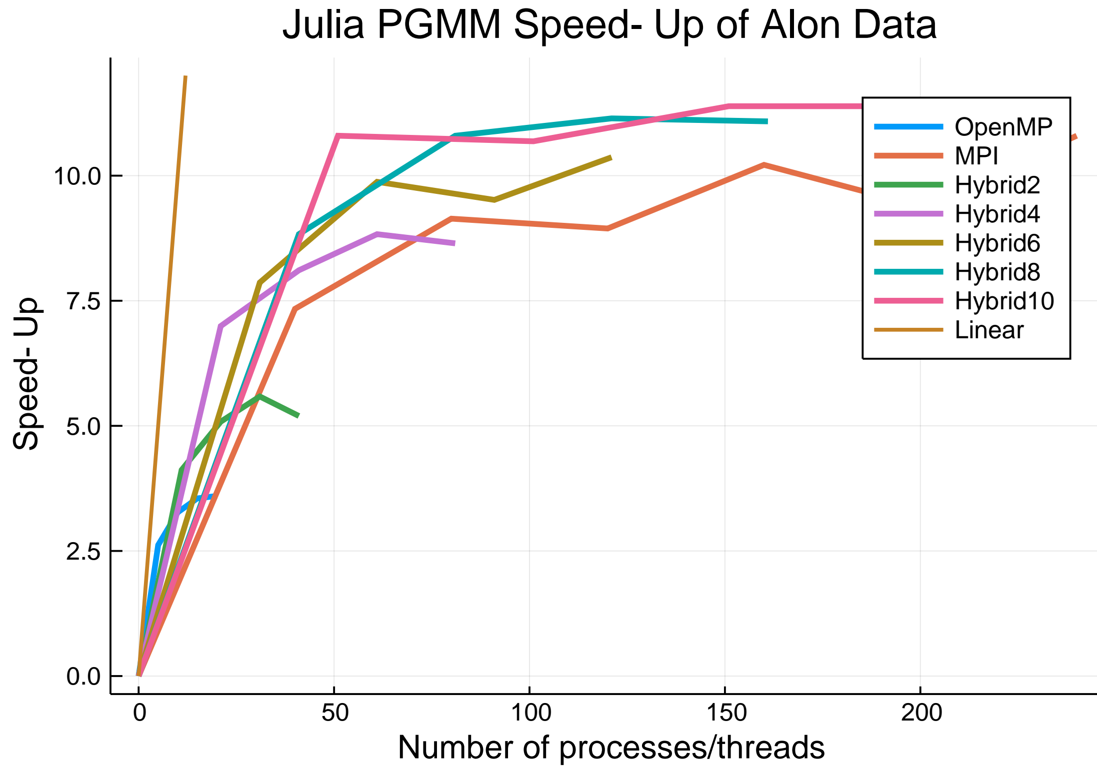

Figure 4.3: Graph of speed-up using OpenMP, MPI, and hybrid parallelization techniques for the Alon data.

4.4.3 Golub Data Analyses

The Golub dataset described in Section 3.4 is again used for analysis. PGMMs are fit for $G = 2, 3$, $q = 1, \ldots, 10$, and $\mathcal{M} \in \mathcal{CCC}, \ldots, \mathcal{UUU}$. This implies that there are $2 \times 10 \times 12$ or 240 tasks to be completed in total. The serial time taken to run `pgmm` on the Golub data is approximately 54523 seconds or 15.15 hours. Note that R fails to produce a result after a serial time of approximately 54.65 hours, over three and a half times the length taken by Julia.

Referring to Table 4.4, the results of applying MPI parallelization to run `pgmm` on the Golub data can be seen. The best result is achieved for 200 processes, i.e., a master

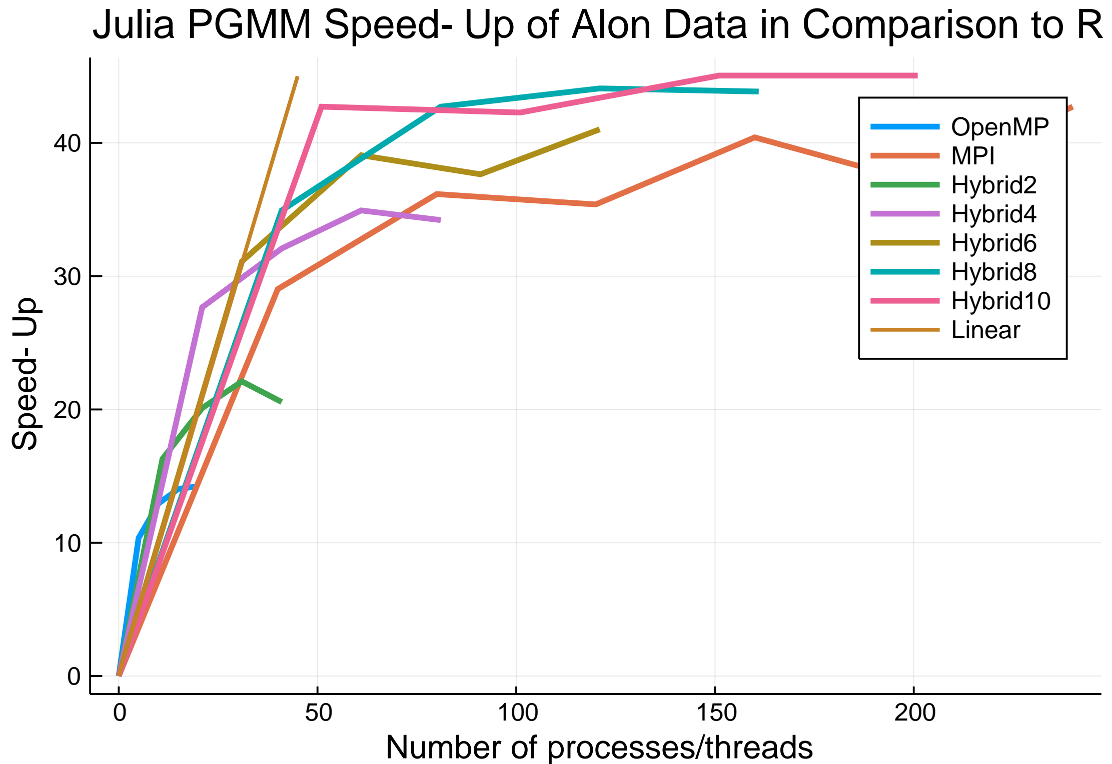

Figure 4.4: Graph of speed-up using OpenMP, MPI, and hybrid parallelization techniques for the Alon data in comparison to the `pgmm` package in R.

and 199 slaves. The time taken is 5167 seconds which results in an approximate speed-up of 10.6. Referring to Table 4.5, the results can be seen when run using OpenMP parallelization. A speed-up of 2.4 is achieved using 10 threads. Note, using 20 threads produces a memory consumption error for the size of this data.

A significant difference in results can be seen in Table 4.6 from the application of the hybrid approach. With fewer overall processes and threads, a greater speed-up can be achieved in almost all hybrid scenarios. The best result being 2970 seconds which is found using 11 processes and 15 threads, achieving a speed-up of 18.4. Comparing this to MPI parallelization using 160 processes, the difference in speed-up is an increase of 8.4 using 9 fewer CPUs (one process is only acting as the master and contains 14

idle CPUs). A comparison of all results can be seen in Figure 4.5.

Table 4.4: Time taken using MPI parallelization to run `pgmm` on the Golub data for different numbers of processes.

MPI Time Results	
Number of Processes	Time in Seconds
40	8587
80	6022
120	5528
160	5474
200	5167
240	6990

Table 4.5: Time taken using OpenMP parallelization to run `pgmm` on the Golub data for different numbers of threads.

OpenMP Time Results	
Number of Threads	Time in Seconds
5	27873
10	22808
15	25805
20	NA

4.5 Performance Analysis

In Section 3.3.3 and Section 4.2.1, two parallel techniques, OpenMP and MPI, were explored. Both techniques serve a key purpose in parallel programming, but give way to the hybrid technique used in Section 4.3.1.

The hybrid technique showed a comparable speed-up to that of the larger scale MPI scenarios using the smaller Alon dataset, but when using a larger dataset, i.e.,

Table 4.6: Time taken in seconds using hybrid parallelization to run pgmm on the Golub data for different numbers of slaves/threads.

Hybrid Parallelization Time Results			
Number of Slaves \ Number of Threads	5	10	15
2	17136	11360	9548
4	9362	6953	5430
6	7769	5493	4557
8	5706	5049	3559
10	4584	3366	2970

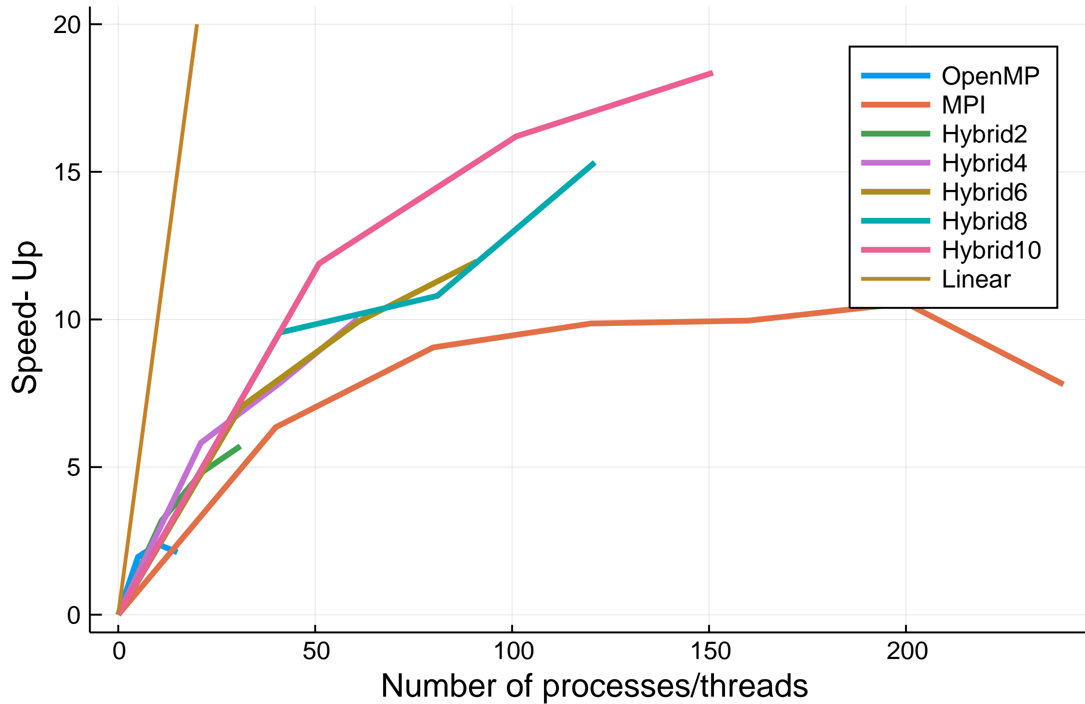

Figure 4.5: Graph of speed-up using OpenMP, MPI, and hybrid parallelization techniques for the Golub data.

Golub, the hybrid technique was shown to outperform these scenarios by larger margins. The disadvantage to the hybrid technique at this stage is that the number

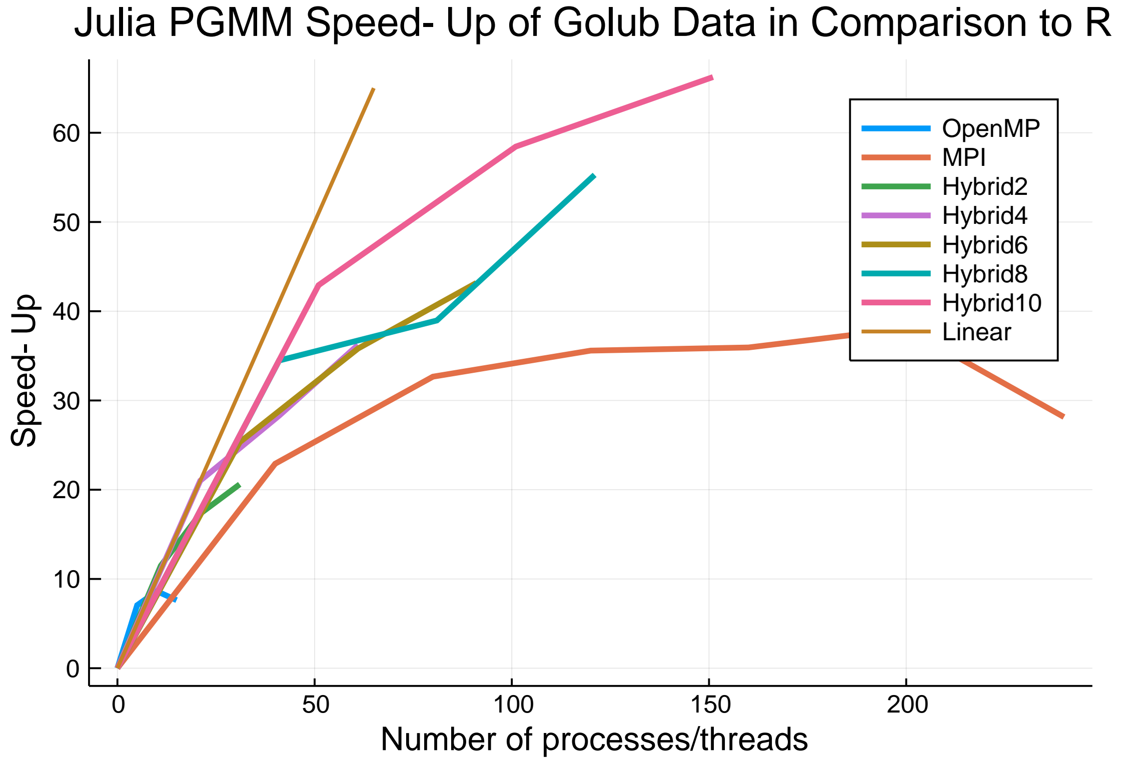

Figure 4.6: Graph of speed-up using OpenMP, MPI, and hybrid parallelization techniques for the Golub data in comparison to the `pgmm` package in R.

of MPI processes is limited to the number of available nodes due to processor and memory constraints for threading within nodes.

In the case of the Golub data, the increase from 2 slaves and 5 threads to 10 slaves and 15 threads saved 14166 seconds or 3.9 hours of computation time. Although this may not seem like much, in terms of overall performance these hybrid approaches saved 37387 seconds or 10.4 hours and 51553 seconds or 14.3 hours, respectively. Applying these methods can potentially save hours or days of computation time depending on the size of the data and the number of triples.

At the time of writing, Julia is currently on version 1.5.2, but the work done in this thesis can not be accomplished using this version. The up-to-date version

of Julia is not currently compatible with MPICH and would require the Open-MPI implementation of MPI software, but due to the aforementioned segmentation faults caused within the UCX environment this is not a possibility.

In order to analyze performance, namely during the application of multithreading parallelization, third party software was needed. The default profiler in Julia does not work with parallel code and requires additional packages to interpret results, even when in serial. Julia has limited compatibility with third party profiling software, i.e., Intel VTune, OProfile, and perf are the only available options. Intel Vtune was used to analyze the results throughout this chapter. The significant downside to the use of a third party profiling software on Julia is that it is a just-in-time (JIT) compiled language meaning that a significant amount of the time spent measuring performance is measuring the compilation of different packages and functions at run time. Options such as using the `PackageCompiler` module in version 1.4.1 were considered to reduce run time, but this proved inefficient as it does not pre-compile in parallel and will recompile functions when new data is used or dimensions change during computation. e.g., for an increased number of factors. Performance was measured for the weaker results using the Alon data. When running the VTune concurrency analysis on the multithreading version using 20 threads for the `@threads` and `@spawn` macro in base, approximate concurrency values of 8 and 18 were obtained, respectively. The concurrency value is the average number of CPUs active during the elapsed execution time. Applying each macro to the Golub data for a master with 10 slaves and 15 threads resulted in runtimes of 3250, and 2970 seconds, respectively. The difference between the macros is an approximate speed-up of 1.6. The `@spawn` macro shows an improvement over the predetermined static scheduling used by the

`@threads` macro in almost all scenarios.

Referring to Table 4.7 a "manual" profiling of the serial code is done in order to see what functions are costly to the overall performance. The $\mathcal{UUCU}$ model is the most costly for overall time, a breakdown of the cost per function in the model can be seen in Table 4.7. Although the $\mathcal{UUCU}$ is a more expensive function due to the number of iterations, it is not costly in comparison when based on the average time per number of calls (ncalls). Unfortunately, this form of manual profiling is unable to be done in parallel, but Table 4.7 provides a good comparison to the time spent on certain function calls in parallel that accumulate similar times to the $\mathcal{UUCU}$ model. To give a better comparison to parallel profiling techniques used in VTune, Table 4.8 shows the most cost expensive functions "under the hood" of Julia during the serial run time. The most expensive function, `dgemm_`, is used by the linear algebra package (LAPACK) for matrix multiplication while the remaining functions are Julia functions to free and allocate memory, as well as use the garbage collector. Next, Table 4.9 & 4.10 show the performance results of running VTune on the multithreaded with 20 threads for the Alon data while Table 4.11 & 4.12 show the MPI results.

Table 4.9 shows the collective call time for all threads to the top six most expensive functions/call stacks. Some calls like `dgemm_kernel` are referenced twice because they are called from different stacks, while the second most expensive stack, although unknown by VTune, has calls to Julia's `jl_apply` for command execution and the garbage collector. For a more in-depth look at multithreading the memory access is analyzed, it can be seen from Table 4.10 that majority of the time is spent in or waiting for the garbage collector. Even though Julia has made garbage collection safe for threading, it is only thread safe because it requires a mutex (or lock) which is

Table 4.7: Profiling PGMM for the Alon data in serial.

		Time			Allocations		
Function	ncalls	Total	% Total	Average	Total	% Total	Average
update $\mathcal{UUCU}$	20	456s	21.2%	22.8s	931GiB	22.7%	46.66GiB
↪ update z12	26.6k	142s	6.62%	5.35ms	482GiB	11.8%	18.6MiB
↪ woodbury2	4.30M	105s	4.91%	24.5µs	309GiB	7.54%	75.4KiB
↪ update delta3	34.7k	109s	5.09%	3.15ms	167GiB	4.06%	4.92MiB
↪ update omega2	34.7k	97.6s	4.55%	2.81ms	110GiB	2.69%	3.25MiB
↪ update sg	13.3k	86.0s	4.01%	6.47ms	132GiB	3.22%	10.2MiB
↪ update mu	13.3k	6.41s	0.30%	482µs	26.2GiB	0.64%	2.02MiB
↪ update theta	34.7k	5.87s	0.27%	169µs	0.97GiB	0.02%	29.2KiB
↪ update lambda	34.7k	4.47s	0.21%	129µs	1.81GiB	0.04%	54.8KiB
↪ update det2	34.7k	1.12s	0.05%	32.4µs	2.88GiB	0.07%	87.0KiB
↪ update beta2	34.7k	964ms	0.04%	27.8µs	2.78GiB	0.07%	84.0KiB
↪ update beta2	34.7k	937ms	0.04%	27.0µs	2.78GiB	0.07%	84.0KiB
↪ update $\mathcal{CUCU}$	20	329s	15.3%	16.5s	526GiB	12.8%	26.3GiB
update lambda $\mathcal{CUU}$	7.36k	97.7s	4.55%	13.3ms	62.6GiB	1.53%	8.71MiB
↪ update z11	14.7k	66.5s	3.10%	4.52ms	194GiB	4.74%	13.5MiB
↪ woodbury2	2.32M	56.2s	2.62%	24.2µs	160GiB	3.91%	72.5KiB
↪ update delta3	18.7k	61.2s	2.85%	3.26ms	90.0GiB	2.20%	4.92MiB
↪ update omega	18.7k	60.9s	2.84%	3.25ms	89.9GiB	2.19%	4.91MiB
↪ update sg	7.36k	34.2s	1.60%	4.65ms	71.5GiB	1.74%	9.94MiB
↪ update mu	7.36k	3.63s	0.17%	493µs	14.2GiB	0.35%	1.97MiB
↪ update theta	18.7k	3.12s	0.15%	167µs	507MiB	0.01%	27.7KiB
↪ update det2	18.7k	647ms	0.03%	34.5µs	1.47GiB	0.04%	82.5KiB
↪ update beta2	18.7k	542ms	0.03%	28.9µs	1.42GiB	0.03%	79.7KiB
↪ update beta2	18.7k	555ms	0.03%	29.6µs	1.42GiB	0.03%	79.7KiB

Table 4.8: Hotspot profiling PGMM for the Alon data in serial.

Function	CPU Time
dgemm_	1165s
jl_apply	495s
_int_free	56s
_int_malloc	29s
malloc_consolidate	22s
jl_gc_pool_alloc	22s
sweep_page	20s

a synchronization primitive that limits access to a certain resource when multiple threads may try to use it concurrently. In the case of Julia's garbage collector, only one thread may access it at a time while other threads requiring access will be put in a queue to gain control over the resource. Julia documentation further states that "compute-bound, non-memory-allocating tasks can prevent garbage collection from running in other threads that are allocating memory which may further slow the garbage collection process".

Table 4.9: Hotspot profiling PGMM for the Alon data using OpenMP.

Function	CPU Time
dgemm_kernel_0	1178s
Outside of known module	1124s
dgemm_ker0	660s
jl_mutex_wait	301s
_madvise	282s
maybe_collect	144s

In the case of the MPI implementation, Table 4.11 shows that, again, much of the time is spent doing matrix multiplication, and waiting for the MPI finalize to execute which means that all processes have completed their tasks (inclusive of the master) and the program is ready to complete. Note that the `_poll` function is directly related to `MPID_Finalize` as it is polling for communication. Consider the memory access

Table 4.10: Memory access profiling PGMM for the Alon data using OpenMP.

Function	CPU Time	Instructions Retired	CPI Rate	CPU Frequency
jl_safepoint_wait_gc	416s	43173600000	7.111	0.308
jl_mutex_wait ↪ jl_mutex_lock	333s	41709600000	15.232	0.796
memfd:julia-codegen	82s	187473600000	0.228	0.217
dgemm_kernel	60s	147408000000	0.192	0.196
jl_mutex_wait ↪ jl_mutex_lock	53s	6597600000	16.135	0.842
jl_safepoint_wait_gc	39s	487200000	54.640	0.305

shown in Table 4.12, the CPI Rate of all functions, which is the cycles per instruction (CPI) retired, is close to zero with an overall average of 0.016. Intel states that a CPI of 1 is typically considered acceptable for HPC performance. There is minimal contention here for communication with the main process as sending and receiving information is accomplished very quickly with the added benefit of multiple processes having the ability to run their own garbage collector concurrently to one another.

Table 4.11: Hotspot profiling PGMM for the Alon data using MPI.

Function	CPU Time
dgemm_	2549s
MPID_Finalize	2345s
jl_apply	989s
_poll	227s
MPIDI_CH3I_Progress	210s
MPI_Recv	154s

Table 4.12: Memory access profiling PGMM for the Alon data using MPI.

Function	CPU Time	Instructions Retired	CPI Rate	CPU Frequency
MPIDI_CH3I_Progress	127s	419642400000	0.000	0.000
jl_unwrap_unionall	42s	66470400000	0.048	0.031
obviously_disjoint	36s	70740000000	0.042	0.034
MPID_nem_tcp_connpoll	34s	171247200000	0.000	0.000
jl_obvious_subtype	26s	98440800000	0.018	0.028

4.6 Discussion

Overall, the performance of serial execution in Julia is far too efficient to fully realize the speed-ups obtained from parallel programming techniques. The MPI efficiency in Julia may look weak, but performance will continue to improve with the application to larger data sets. A large gap was seen in the CPI rates with overall averages going from 0.016 using MPI to 1.471 using OpenMP, suggesting this is where most of the downfall in the hybrid technique has come from. Considering that Julia is very early in its life cycle, there are many additional potential problems that could be occurring within the multithreading implementation.

From the VTune analysis, 31.2% of pipeline slots, hardware resources needed to process one micro-op such as a missing piece of information, were memory bound stalls. DRAM bandwidth bound, a metric for the percentage of elapsed time communicating with the main memory, attributed 11.6% of overall execution time. A total of 716893011 last-level cache (LLC) misses were found during execution time, more than three times the number of misses in serial. The LLC is the last and longest latency memory level where each miss must be filled by the local or remote DRAM with a substantial latency attributed to it. Phenomena in parallel programming, namely false sharing and cache coherency, are plausible issues contributing to these problems.

False sharing refers to a performance hindering pattern that occurs in HPCs with distributed, coherent caches. While a thread attempts to access data that is unique to itself, but this data shares a cache block with other data that are read and/or altered by other threads, the system may force the thread to wait for access despite the lack of conflict. This causes memory stalls and will significantly reduce run time without proper synchronization.The processors on the HPC used are part of the Skylake family where there are three levels of cache in the subsystem memory for these processors. Each core has an L1 and L2 cache with an L3 cache shared among the cores. In OpenMP applications, copies of the same cache can be present in the L1 or L2 cache for multiple cores.

Cache coherency refers to the shared memory of resources that are stored across multiple caches. For example, although all results are stored in a different part of an array in the code and one thread will not interfere with another, when one thread updates the array, this may force a similar thread updating another portion to be invalidated. This is a typical disadvantage when it comes to scaling hybrid parallelization that uses threading within nodes in comparison to MPI parallelization alone. When a core in the Skylake processor family updates information stored in a shared cache it typically invalidates all other cores that possess this memory, even when the core resides in a different socket. This would force a core that later needs this data on its local L1 and/or L2 cache to refer to the main memory to create a new copy of the information. These two phenomenons will cause significant performance delays because they are both hardware related, and can not always be remedied in shared memory programs.

Chapter 5

Parallelization with Python and Comparative Approaches

5.1 Introduction

The methodology presented throughout this thesis, namely Chapter 4, was not originally intended to have been accomplished using Julia. The author had originally set out to complete the goals in this thesis using Python 3. Many attempts had been made to improve performance before the decision to move to another language, i.e., Julia, was made. Although a much more established language with many great qualities, Python posed many challenges for model-based clustering in both serial and parallel implementations.

This chapter is outlined as follows. In Section 5.2 two approaches to parallelization with Python are discussed, one a naive approach that led to implementing model-based clustering *via* PGMMs in parallel using a combination of programming languages, followed by a native Python approach to compare and improve the results using the

previous combination of languages. In Section 5.3 both proposed approaches will be illustrated in serial and parallel in order to compare and contrast the benefits of each in Section 5.4.

5.2 Methodology

5.2.1 Implementations

With the idea in mind to create the MFPGMM approach as developed in Chapter 3 of this thesis, the first problem was to make the runtime realistic for the multitude of triples that would have to be run. The `pgmm` package developed in R by (McNicholas *et al.*, 2019) could be altered to suit the approach, but at the cost of days of computational time. The work in McNicholas *et al.* (2010) had shown that improvements in performance were obtainable using MPI parallelization and the goal was to expand on this approach and implement hybrid parallelization to further improve performance using the `pgmm` package in R, but with a more established language for its parallel performance, i.e., Python.

Python has no inherently "pleasant" way of communicating with other languages such as R and/or C, especially when a transfer of data is required between the languages. Python does however have several modules that allow communication with the operating system which can be used to run external processes. The more up-to-date method used for this implementation is the `subprocess` module (older modules such as `os` may produce similar results). Within the `subprocess` module is the 'Popen' class that allows the execution of external programs as a child process. A child process refers to any process that is started *via* a parent process which is part of the main

program. The advantage of the Popen class is that it takes 'args' as a parameter, allowing the arguments for the child process to be specified before the program is executed on the command line. From the Python `subprocess` documentation, the args parameter must always be specified and contain a string of program arguments.

Consider Algorithm 2, this function shows how a number of arguments can be effectively sent to R *via* the command line when using the Popen class. Arguments that may vary in size from run-to-run or for different PGMM models, e.g., the data (represented as `x`), are written to a text file as a string under specific naming conventions that correspond to the component size and number of factors. This is important for other arguments to ensure they are reading the correct files without causing a race condition to read or write, and allows for universal use of the code with different data sets. Command line arguments specified in Popen are read into R followed by the appropriate text files while being converted to matrices and numeric values. From R, the `pgmm` executable file, written in C, attached to the `pgmm` library is run for the given parameters and results are again written to name specific text files which Python will read once it receives a completion signal from the communicate action appended to the `subprocess.Popen` function. Note that all tasks run from `subprocess` are asynchronous and some form of interaction or wait is required to have the tasks finish before importing results.

Analogous to that of the code used in Chapter 4, a native version of PGMMs was developed in Python that negated the use of the `subprocess` module that will become clear in Section 5.3. The decision to use a message passing inference over distributed computing in Julia was inspired by the original implementation developed in Python and will not be described in detail because they are nearly identical. The primary

Algorithm 2: The subprocess.Popen function used within Python to communicate with external programs.

```
Function run_pgmm(x, z, bic, kls, q8, p, g8, n, model, cluster, llambda, psi,
  tol):
    np.savetxt("x" + str(g8) + str(q8) + str(model) + ".txt", x)
    np.savetxt("z" + str(g8) + str(q8) + str(model) + ".txt", z)
    np.savetxt("kls" + str(g8) + str(q8) + str(model) + ".txt", kls)
    np.savetxt("lam_temp" + str(g8) + str(q8) + str(model) + ".txt",
      llambda)
    if (isinstance(psi, float) == True):
        np.savetxt("psi_temp" + str(g8) + str(q8) + str(model) + ".txt",
          psi[None])
    else:
        np.savetxt("psi_temp" + str(g8) + str(q8) + str(model) + ".txt", psi)
    p4 = subprocess.Popen(["Rscript –vanilla run_pgmm_prc2.R %s %s %s
      %s %s %s %s %s %s %s %s %s %s" % ("x" + str(g8) + str(q8) +
      str(model), "z" + str(g8) + str(q8) + str(model), str(bic), "kls" +
      str(g8) + str(q8) + str(model), str(q8), str(p), str(g8), str(n),
      str(model), str(cluster), "lam_temp" + str(g8) + str(q8) + str(model),
      "psi_temp" + str(g8) + str(q8) + str(model), str(tol))],
      stdout=subprocess.PIPE, stdin=subprocess.PIPE,
      shell=True).communicate()[0]
    zbest = pd.read_csv("zbest" + str(g8) + str(q8) + str(model) + ".txt",
      sep=" ", header=None)
    zbest = np.asarray(zbest)
    bic_best = pd.read_csv("bicbest" + str(g8) + str(q8) + str(model) +
      ".txt", sep=" ", header=None)
    bic_best = np.asarray(bic_best)
    lmbda_best = pd.read_csv("lambdabest" + str(g8) + str(q8) +
      str(model) + ".txt", sep=" ", header=None)
    lmbda_best = np.asarray(lmbda_best)
    psi_best = pd.read_csv("psibest" + str(g8) + str(q8) + str(model) +
      ".txt", sep=" ", header=None)
    psi_best = np.asarray(psi_best)
    return None, zbest, bic_best, lmbda_best, psi_best
```

difference between the languages, and a deciding factor when it came to move away from Python, can be seen in the shared memory parallelism for which Python was not designed with in mind. The following section details the parallel framework applied to the two aforementioned implementations.

5.2.2 Parallel Framework

Shared memory parallelism in Python does exist within the 'Threading' module where it will share mutable objects, i.e., dictionaries, created using the same memory address when attempting to access it. The downfall when it comes to using Python for shared memory parallelism is that in order to make threads attempting to access this memory address safe, and free of race conditions, a global interpreter lock (GIL) is used. A GIL is a mechanism used by Python to ensure that only one thread is executing bytecode (Python source code after compilation) at a time. In other words, although threading gives the appearance of being run in parallel, the source code is being run in serial fashion. Threading in Python does have potential advantages if the targets were executables outside of the native environment, but for the native implementation of an iterative model the runtime will suffer.

Perhaps a step away, but in the right direction, from shared memory parallelism is the use of the `Multiprocessing` module. Pointers are objects that store a memory address and are the standard way to accomplish shared memory parallelism, but they are forbidden between processes to allow for safe memory access. In addition, by preventing the use of a shared pointer between processes it also allows for variables created within processes to remain local. To accomplish shared memory parallelism between processes, inter-thread communication can be used where multiple pointers

to one memory address would exist. The `Multiprocessing` module is very similar to the `@spawn` macro described in Chapter 4 in the sense that it will spawn processes for tasks. This module allows for both local and remote concurrency which negates the GIL from threading. Note, there are multiple available methods for starting a process, e.g., fork and spawn, but this chapter will focus on the use of forking as it is the recommended and default method of Linux based operating systems.

There are two primary classes within the `Multiprocessing` module, i.e., Process and Pool. The Pool class uses a group of specified workers to execute a process, while the Process class itself will only spawn a Process object. The Pool class will start a process by calling apply or map on the process, while the Process class must have each process called by a `.start()` for Python to allocate the process. Both methods require a `.join()` to wait for all processes to complete before resuming the code run on the parent process. When using a Pool, the results must be retrieved using `.get()` while the Process class has a main process, also referred to as the parent, that owns a dictionary which can be shared to each process to store results in. Algorithms 3 & 4 show the ease of implementation for multiprocessing in Python. As in the Julia implementation,

```
export OMP_NUM_THREADS=1
```

must be called prior to execution which is identical to

```
BLAS.set_num_threads(1)
```

called within Julia to prevent oversubscribing. Although this is distributed memory, it requires no call to a process laucher, e.g., mpiexec or mpirun, in order to accomplish process based parallelism as it would in a MPI framework.

Algorithm 3: Application of the Process class in the `Multiprocessing` module for an array of tasks in Python.

```
Function parallel_pgmm(data, rg=range(g_min, g_max), rq=range(q_min,
  q_max), rt=range(t_min, t_max), n, p, seed):
    triples = [rg, rq, rt]
    set_tuple = list(itertools.product(*triples))
    set_tuple = np.asarray(set_tuple)
    index = range(0, len(set_tuple))
    set_tuple = np.c_[set_tuple, index]
    mgr = multiprocessing.Manager()
    results_save = mgr.dict()
    jobs = [multiprocessing.Process
    (target=work, args=(data, j[0], j[1], j[3], n, p, seed, j[4], results_save))
    for j in set_tuple
    ]
    for j in jobs:
    j.start()
    for j in jobs:
    j.join()
```

Algorithm 4: Application of the Pool class in the `Multiprocessing` module for an array of tasks in Python.

```
Function parallel_pgmm(data, rg=range(g_min, g_max), rq=range(q_min,
  q_max), rt=range(t_min, t_max), n, p, seed, ntasks_node):
    triple = [rg, rq, rt]
    set_tuple = list(itertools.product(*triple))
    set_tuple = np.asarray(set_tuple)
    index = range(0, len(set_tuple))
    set_tuple = np.c_[set_tuple, index]
    with multiprocessing.Pool(processes=ntasks_node) as pool:
    jobs = [pool.apply_async
    (work, args=(data, j[0], j[1], j[2], n, p, seeder, j[3]))
    for j in set_tuple
    ]
    pool.close()
    pool.join()
    results_save = [p.get() for p in jobs]
```

When starting tasks contained within jobs using the Process class, there is no limitation to the number of available CPUs. Although this can be beneficial at times, ideally there will only be one job running per available CPU. The Poll class limits applications to a group of workers which asynchronously apply tasks to free workers within the group, i.e., the number of available CPUs. The Pool class also has benefits when applied for model-based clustering because only the processes under execution are kept in the memory, while if the task contained less repetition and smaller amounts of data the Process class would be more beneficial.

Referring to the hybrid pseudocode for the master-slave paradigm in Appendix A, the slave in Python would simply contain one of the algorithms described throughout this section in place of the shared memory parallelism algorithm used by Julia in Chapter 4. The applications in Section 5.3 will be completed using the Pool class shown in Algorithm 4.

5.3 Application

The Alon dataset described in Section 3.4 is used again for analyses in order to accurately compare performance. PGMMs are fit for $G = 2, 3$, $q = 1, \dots, 10$, and $\mathcal{M} \in \mathcal{CCC}, \dots, \mathcal{UUU}$ for a total of 240 tasks. The `Multiprocessing` module will be applied using an identical numbers of processes(threads) as in Section 4.4 while the serial and MPI implementations remain unchanged.

5.3.1 Combined Implementation

The serial time taken to calculate the speed-up for these results will use the same time found in the R application, i.e., 2.29 hours, because the combined implementation primarily uses Python for parallelization while tasks are run through R. The combined approach would require at least two CPUs to be available in order to spawn a task. This would not truly be considered a serial implementation because the performance would suffer due to increased communication required for each triple preventing an accurate comparison from being given.

In Section 3.3.3 Open-MPI and MPICH implementations for MPI software were discussed. The latter was used for Julia, but the former must be used for Python which brings about many issues due to the interaction between this software and UCX memory hooks. Some issues can be negated, but at the potential cost of performance. When running parallel code, almost all circumstances are identical to Chapter 4 in terms of modules and resources excluding the MPI software. When using a MPI, tags can be passed through to each node using the `-mca` switch which allows parameters to be passed to Modular Component Architecture (MCA) modules which directly impact runtime. To avoid the use of UCX, the MPI point-to-point management layer (pml) must be set to ob1. ob1 is a a component in the PML framework that is used to select a communication protocol that is based on the message to send and the MPI point-to-point Byte Transfer Layer (BTL) components. The BTL communication device used to transport messages must be set to openib and self. The openib BTL component is used to select InfiniBand communication and self is set for loopback communication (this must always be set). These communication protocols will prevent almost all segmentation faults, but it can be seen from results listed as NA that it will not

prevent all errors from occurring.

Referring to Tables 5.1, 5.2, & 5.3 the performance of MPI, multiprocessing, and hybrid parallelization can be seen, respectively. It is clear from the results that MPI is dominant if resources are readily available, but `Multiprocessing` is more efficient. It is also important to consider that if MPI is the sole form of parallelization, resources must double the number of processes or else a segmentation fault will occur when trying to fork tasks outside of Python, i.e., for every process there are two CPUs assigned. While when using the `Multiprocessing` module, this is not required because tasks are forked and not transported using a MPI. The best speed-up achieved from `Multiprocessing` is 12.4 using 20 processes which is equivalent to that of MPI using 20 processes, but 40 CPUs. The disadvantage of `Multiprocessing` is the lack of scaling when compared to MPI. Starting half of the total tasks at the same time using MPI achieves a speed-up of approximately 27.5 which provides a negligible increase in performance from starting between 60 and 100 processes.

Table 5.1: Time taken using MPI parallelization to run `pgmm` on the Alon data for different numbers of processes outside of Python.

MPI Time Results	
Number of Processes	Time in Seconds
20	667
40	769
60	354
80	330
100	301
120	300

The hybrid scenario does not outperform `Multiprocessing` or MPI in Python for any scenarios because the forced usage of Open-MPI software and the distributed

Table 5.2: Time taken using the `Multiprocessing` module to run `pgmm` on the Alon data for different numbers of processes outside of Python.

`Multiprocessing` Time Results	
Number of Processes	Time in Seconds
5	2560
10	1571
15	1245
20	667

messaging within MPI processes acquire too much overhead in comparison to using only one form of parallelization.

Table 5.3: Time taken in seconds using hybrid parallelization to run `pgmm` on the Alon data for different numbers of slaves/processes outside of Python.

Hybrid Parallelization Time Results				
Number of Processes / Number of Slaves	5	10	15	20
2	2667	1956	1267	1193
4	2685	2018	1485	1237
6	1367	1875	1371	1171
8	1431	1686	1179	1035
10	NA	1438	1157	998

5.3.2 Native Python

The serial time taken to run `pgmm` on the Alon data is approximately 4189 seconds or 1.2 hours, just less than half of the time taken using R. Referring to Tables 5.4, 5.5, & 5.6 the results of each parallel application for the given number of processes can be seen. A visible improvement in performance can be seen from the results in

Section 5.3.1. Although the overhead acquired by using MPI parallelization far exceeds that of `Multiprocessing`, it is still vastly improved and required for the hybrid approach. The `Multiprocessing` implementation achieved a speed-up of 6.6 using 20 processes where the decrease in performance can be attributed to the increased serial performance. MPI parallelization achieved an approximate speed-up of 19, almost triple that of `Multiprocessing`, but with a significant loss in efficiency.

Table 5.4: Time taken using MPI parallelization to run `pgmm` on the Alon data for different numbers of processes in native Python.

MPI Time Results	
Number of Processes	Time in Seconds
40	419
80	285
120	246
160	235
200	234
240	220

Table 5.5: Time taken using the Multiprocessing class to run `pgmm` on the Alon data for different numbers of processes in native Python.

`Multiprocessing` Time Results	
Number of Processes	Time in Seconds
5	1795
10	1057
15	729
20	636

It can still be seen that hybrid parallelization acquired far too much overhead for similar reasons to that of Section 5.3.1. It is prevalent that the more `Multiprocessing` processes started, the greater the performance gain.

Table 5.6: Time taken in seconds using hybrid parallelization to run `pgmm` on the Alon data for different numbers of slaves/processes in native Python.

Hybrid Parallelization Time Results				
Number of Slaves \ Number of Processes	5	10	15	20
2	1648	1253	879	711
4	1656	1030	749	655
6	826	1015	717	692
8	850	1068	822	679
10	NA	872	737	661

5.4 Discussion

In this chapter, two approaches were considered for the implementation of PGMMs in Python using multiple parallelization techniques. Between the two approaches with multiple forms of parallelization, the native Python implementation using `Multiprocessing` for parallelization demonstrated superior performance over all others, but lacked the opportunity for scaling. MPI parallelization showed improved results when applied to the native implementation of PGMMs and with the addition of more processes, but showed decreased efficiency with scaling.

Python has long since been known as a standard programming language that offers immense flexibility, but has never been known for its speed. While other programming languages such as C and Julia are compiled languages, Python is an interpreted language. Interpreted code is known to be slower than machine code used by compiled languages because it takes a plethora of instructions to execute an interpreted instruction versus a simple machine instruction. An interpreted language will acquire a greater amount of overhead upon execution as it runs through the code line by line

performing tasks such as removing blank space, removing comments, and allocating memory prior to execution. While compiled language can be directly converted into machine code and executed.

To add further difference between the languages, C is a purely compiled language that requires an additional step for manual compilation prior to execution, while Julia is JIT compiled as mentioned in Section 4.5. JIT compilation minimizes the gap between compiled and interpreted languages as it attempts to give the appearance of an interpreted language while still offering the performance of a compiled language, but it is not without its downfalls. JIT compiled languages have overhead related to compilation during runtime and visible delays in startup time. The work completed throughout this chapter brought into question whether or not the hybrid parallelization could be further improved, which led to the use of Julia throughout this thesis.

Chapter 6

Conclusion

6.1 Summary

The work developed in this thesis represents a significant contribution to the growing body of literature on computational statistics for model-based clustering with independent models.

In Chapter 3, a new technique in which the number of factors per component may vary for PGMMs where the factor loadings are unconstrained is introduced. This is a novel approach which is applicable to any scenario in which the PGMM family may be fit, but allows for more flexible solutions. This work also demonstrated the computational benefit of applying such models to high-dimensional data in parallel.

In Chapter 4, hybrid parallelization that allows for the combination of distributed memory parallelization on the node inter-connect with shared memory parallelization inside of each node is developed in Julia. The approach demonstrated how improved computational runtime can be obtained using less resources in comparison to OpenMP

and MPI parallelization alone. The serial and parallel performance of Julia was analyzed using third party software, i.e., VTune, to find where computational efficiency becomes a concern.

In Chapter 5, the parallelization techniques detailed in Chapter 4 are shown as they were originally implemented in Python using multiple different approaches. This chapter sheds light on the downfall of hybrid parallelization in Python, or lack thereof. A comparison of the different programming languages used throughout this thesis and the benefits of Julia for model-based clustering is brought to fruition. The combined approach discussed throughout Chapter 5 can also be made applicable to Julia with the language's unique ability to execute R code directly within the Julia environment. Making a strong case for the departure from standalone R as a mainstream statistical programming language.

6.2 Future Work

6.2.1 Improving Parallel Efficiency and Determining Optimal Resources

Throughout this thesis, it has been shown that computational runtime can be vastly improved by applying parallel computing techniques, but there is a visible decrease in efficiency with scaling. Consider the hybrid parallelization results in Chapter 4 for the Alon data, the difference between 50 CPUs and 200 CPUs is 10 seconds and offers no practical gain in efficiency. Ideally, the runtime per model for the size of the data can be averaged, given that communication time is relatively consistent, and resources can be adjusted (if available) to determine the optimal number of resources

to use when the gain of those resources falls below a specified percentage.

Julia has many upcoming advancements as well as the removal of certain limitations, some of which were discussed in Chapter 4, which will provide significant improvement in threading and allow for further efficiency. Although the hybrid scenario has proven to be efficient, OpenMP may be a far more practical option when it comes to parallel model-based clustering. Implementing large-scale OpenMP parallelization on substantially larger HPC nodes such as those available on IBM BlueGene system would provide more insight into this topic.

6.2.2 Applications to Big Data and Other Model-Based Clustering Techniques

Parallel model-based clustering can be extremely useful with new sets of increasingly larger data being gathered around the world. The speed-up from applications to larger datasets showed increasingly better performance. It would be interesting to study the application of these techniques to scenarios which may take weeks or months to complete due to the size of the data. This may also call into question the effectiveness of the BIC for estimation of component sizes with higher dimensional data which may provide the opportunity to apply other techniques in parallel. Parallelization can not only be applied to larger data, but to different model-based clustering techniques with multiple parameters that involve any form of tuning or prolonged runtime where each instance can be run independently.

Appendix A

Functions from Parallel Code

Algorithm 5: The main() Function

```
Function Main():
    MPI.Init()
    comm = MPI.COMM_WORLD
    size = MPI.Comm_size(comm)
    rank = MPI.Comm_rank(comm)
    if rank == 0 then
        master(comm, size, rank)
    else
        slave(comm, size, rank)
    MPI.Barrier(comm)
    MPI.Finalize()
```

Algorithm 6: The master() Function

```
Function Master():
    read input arguments
    while closed workers < total workers do
        probe all slaves
        read the status of the slave
        if slave is waiting for master then
            get the source of the slave
            receive message from the respective slave
            if slave is looking for work then
                if there are random starts required then
                    send random or k−means start parameters to slave
                    send data to slave
                    increase random start index
                else if starts are complete and there are jobs to do then
                    send a triple parameters for the slave to work on
                    if q is a vector then
                        send vector for q to slave
                    send data to slave
                    increase job index
                else
                    let the slave know it is time to close
                    increase closed workers index
            else if slave is returning a job then
                receive (BIC, G, q, M) from slave
                if q == −1 then
                    receive q vector of length G
                receive n × G matrix of z
                if BIC > Saved_BIC then
                    update results
            else if slave is returning a start then
                receive (BIC, G, q, M) from slave
                receive z_start matrix from slave
                if BIC_Start > Saved_BIC_Start then
                    update z_start matrix
    Print final results
```

Algorithm 7: The slave() Function

```
Function Slave():
    while true do
        send request for work to master
        receive job from master
        read status of job
        if task is a job and factor loadings are constrained then
            receive parameters for job
            receive z_start matrix
            receive data
            run AECM for respective triple
            send BIC and parameters to master
            send z to master
        else if task is a job and factor loadings are unconstrained then
            receive parameters for job
            receive q vector matrix
            receive z_start matrix
            receive data
            run AECM for respective triple
            send BIC and parameters to master
            send z to master
        else if task is a k−means start then
            receive parameters for job
            receive data
            run AECM for respective triple
            send BIC and parameters to master
            send z to master
        else if task is a random start then
            receive parameters for job
            receive data
            run AECM for respective triple
            send BIC and parameters to master
            send z to master
        else if task is empty then
            close slave
```

Appendix B

Multi-Factor PGMM Results

Table B.1: Time taken using MPI parallelization to run mf-pgmm on the coffee data for different numbers of processes.

Number of Cores	Time in Seconds	Speed-Up (% usage)
1 (Serial)	1920	NA
5	768	2.5(62.50%)
90	99	19.40(21.80%)
150	85	22.59(15.16%)
210	78	24.6(11.77%)
270	76	25.26(9.39%)
330	71	27.04(8.22%)
420	74	25.95(6.19%)

Table B.2: Time taken using MPI parallelization to run mf-pgmm on the Italian Wine data for different numbers of processes.

Number of Cores	Time in Seconds	Speed-Up (% usage)
1 (Serial)	245437	NA
90	5412	45.35(50.96%)
150	3610	67.99(45.63%)
210	2672	91.86(43.74%)
270	2151	114.1(42.42%)
330	1781	137.81(41.89%)
420	1463	167.76(39.94%)
980	1239	198.09(20.23%)

Table B.3: Time taken using MPI parallelization to run mf-pgmm on the Italian Olive Oil data for different numbers of processes.

Number of Cores	Time in Seconds	Speed-Up (% usage)
1 (Serial)	42842	NA
90	1282	33.42(37.55%)
150	913	46.92(31.49%)
210	702	61.03(29.2%)
270	618	69.32(25.77%)
330	540	79.34(24.11%)

Table B.4: Time taken using MPI parallelization to run mf-pgmm on the Alon for different numbers of processes.

Number of Cores	Time in Seconds	Speed-Up (% usage)
1 (Serial)	212407	NA
90	3860	55.03(61.83%)
150	2473	85.89(57.64%)
210	1837	115.63(55.32%)
270	1497	141.89(52.75%)
330	1286	165.17(50.20%)

Table B.5: Time taken using MPI parallelization to run mf-pgmm on the Golub data for different numbers of processes.

Number of Cores	Time in Seconds	Speed-Up (% usage)
1 (Serial)	271336	NA
90	7057	38.45(43.20%)
150	5506	49.28(33.07%)
210	4971	54.58(26.12%)
270	4658	58.25(21.65%)
330	4162	65.19(19.82%)

Bibliography

Aitken, A. C. (1926). A series formula for the roots of algebraic and transcendental equations. *Proceedings of the Royal Society of Edinburgh*, **45**, 14–22.

Alon, U., Barkai, N., Notterman, D. A., Gish, K., Ybarra, S., Mack, D., and Levine, A. J. (1999). Broad patterns of gene expression revealed by clustering analysis of tumor and normal colon tissues probed by oligonucleotide arrays. *Proceedings of the National Academy of Sciences of the United States of America*, **96**(12), 6745–6750.

Baum, L. E., Petrie, T., Soules, G., and Weiss, N. (1970). A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. *Annals of Mathematical Statistics*, **41**, 164–171.

Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, **59**(1), 65–98.

Bhattacharya, A. and Dunson, D. B. (2011). Sparse Bayesian infinite factor models. *Biometrika*, **98**(2), 291–306.

Böhning, D., Dietz, E., Schaub, R., Schlattmann, P., and Lindsay, B. (1994). The distribution of the likelihood ratio for mixtures of densities from the one-parameter exponential family. *Annals of the Institute of Statistical Mathematics*, **46**, 373–388.

Chapman, B., Jost, G., and van der Pas, R. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press.

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

Flynn, M. J. (1966). Very high-speed computing systems. *Proceedings of the IEEE*, **54**(12), 1901–1909.

Forina, M. and Tiscornia, E. (1982). Pattern recognition methods in the prediction of Italian olive oil origin by their fatty acid content. *Annali di Chimica*, **72**, 143–155.

Forina, M., Armanino, C., Lanteri, S., and Tiscornia, E. (1983). Classification of olive oils from their fatty acid composition. In H. Martens and H. Russwurm Jr, editors, *Food Research and Data Analysis*, pages 189–214. Applied Science Publishers, London.

Fraleigh, J. B. (1990). *Calculus with Analytic Geometry*. Addison-Wesley, Reading, MA, 3rd edition.

Fraley, C. and Raftery, A. E. (1998). How many clusters? Which clustering methods? Answers via model-based cluster analysis. *The Computer Journal*, **41**(8), 578–588.

Getz, G., Levine, E., and Domany, E. (2000). Coupled two-way clustering analysis of gene microarray data. *Proceedings of the National Academy of Sciences of the United States of America*, **97**(22), 12079–12084.

Ghahramani, Z. and Hinton, G. E. (1997). The EM algorithm for factor analyzers. Technical Report CRG-TR-96-1, University of Toronto, Toronto, Canada.

Golub, T. R., Slonim, D. K., Tamayo, P., Huard, C., Gaasenbeek, M., Mesirov, J. P., Coller, H., Loh, M. L., Downing, J. R., Caligiuri, M. A., Bloomfield, C. D., and Lander, E. S. (1999). Molecular classification of cancer: Class discovery and class prediction by gene expression monitoring. *Science*, **286**, 531–537.

Gropp, W., Lusk, E., and Skjellum, A. (1999). *Using MPI (2nd Ed.): Portable Parallel Programming with the Message-passing Interface*. MIT Press, Cambridge, MA, USA.

Hartigan, J. A. and Wong, M. A. (1979). A k-means clustering algorithm. *Applied Statistics*, **28**(1), 100–108.

Hubert, L. and Arabie, P. (1985). Comparing partitions. *Journal of Classification*, **2**(1), 193–218.

Kass, R. E. and Wasserman, L. (1995a). A reference Bayesian test for nested hypotheses and its relationship to the Schwarz criterion. *Journal of the American Statistical Association*, **90**(431), 928–934.

Kass, R. E. and Wasserman, L. (1995b). A reference Bayesian test for nested hypotheses and its relationship to the Schwarz criterion. *Journal of the American Statistical Association*, **90**(431), 928–934.

Keribin, C. (2000). Consistent estimation of the order of mixture models. *Sankhyā. The Indian Journal of Statistics. Series A*, **62**(1), 49–66.

Lagrange, J. L. (1788). *Méchanique Analitique*. Chez le Veuve Desaint, Paris.

Lawley, D. N. and Maxwell, A. E. (1962). Factor analysis as a statistical method. *Journal of the Royal Statistical Society: Series D*, **12**(3), 209–229.

Leroux, B. G. (1992). Consistent estimation of a mixing distribution. *The Annals of Statistics*, **20**(3), 1350–1360.

Lindsay, B. G. (1995). Mixture models: Theory, geometry and applications. In *NSF-CBMS Regional Conference Series in Probability and Statistics*, volume 5. Hayward, California: Institute of Mathematical Statistics.

MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297, Berkeley. University of California Press.

McLachlan, G. J. and Krishnan, T. (2008). *The EM Algorithm and Extensions*. Wiley, New York, 2 edition.

McLachlan, G. J. and Peel, D. (2000). Mixtures of factor analyzers. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 599–606, San Francisco. Morgan Kaufmann.

McLachlan, G. J., Bean, R. W., and Peel, D. (2002). A mixture model-based approach to the clustering of microarray expression data. *Bioinformatics*, **18**(3), 412–422.

McNicholas, P. D. (2016a). *Mixture Model-Based Classification*. Chapman & Hall/CRC Press, Boca Raton.

McNicholas, P. D. (2016b). Model-based clustering. *Journal of Classification*, **33**.

McNicholas, P. D. and Murphy, T. B. (2005). Parsimonious Gaussian mixture models. Technical Report 05/11, Department of Statistics, Trinity College Dublin, Dublin, Ireland.

McNicholas, P. D. and Murphy, T. B. (2008). Parsimonious Gaussian mixture models. *Statistics and Computing*, **18**(3), 285–296.

McNicholas, P. D. and Murphy, T. B. (2010a). Model-based clustering of longitudinal data. *The Canadian Journal of Statistics*, **38**(1), 153–168.

McNicholas, P. D. and Murphy, T. B. (2010b). Model-based clustering of microarray expression data via latent Gaussian mixture models. *Bioinformatics*, **26**(21), 2705–2712.

McNicholas, P. D. and Tait, P. A. (2019). *Data Science with Julia*. Chapman & Hall/CRC Press, Boca Raton.

McNicholas, P. D., Murphy, T. B., McDaid, A. F., and Frost, D. (2010). Serial and parallel implementations of model-based clustering via parsimonious Gaussian mixture models. *Computational Statistics and Data Analysis*, **54**(3), 711–723.

McNicholas, P. D., ElSherbiny, A., McDaid, A. F., and Murphy, T. B. (2019). *pgmm: Parsimonious Gaussian Mixture Models*. R package version 1.2.4.

Meng, X.-L. and Rubin, D. B. (1993). Maximum likelihood estimation via the ECM algorithm: a general framework. *Biometrika*, **80**, 267–278.

Meng, X.-L. and van Dyk, D. (1997). The EM algorithm — an old folk song sung to a fast new tune (with discussion). *Journal of the Royal Statistical Society: Series B*, **59**(3), 511–567.

Murphy, K., Viroli, C., and Gormley, I. C. (2020). Infinite mixtures of infinite factor analysers. *Bayesian Analysis*, **15**(3), 937–963.

Orchard, T. and Woodbury, M. A. (1972). A missing information principle: Theory and applications. In L. M. Le Cam, J. Neyman, and E. L. Scott, editors, *Proceedings of the Sixth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Theory of Statistics*, pages 697–715. University of California Press, Berkeley.

R Core Team (2019). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Rand, W. M. (1971). Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association*, **66**(336), 846–850.

Schwarz, G. (1978). Estimating the dimension of a model. *The Annals of Statistics*, **6**(2), 461–464.

Spearman, C. (1904). The proof and measurement of association between two things. *American Journal of Psychology*, **15**, 72–101.

Steinley, D. (2003). Local optima in k-means clustering: What you don't know may hurt you. *Psychological methods*, **8**(3), 294–304.

Steinley, D. (2004). Properties of the Hubert-Arabie adjusted Rand index. *Psychological Methods*, **9**, 386–396.

Street, W., Wolberg, W. H., and Mangasarian, O. L. (1993). Nuclear feature extraction for breast tumor diagnosis. volume 1905, pages 861–870. International Society for Optics and Photonics, SPIE.

Streuli, H. (1973). Der heutige stand der kaffeechemie. In *Association Scientifique International du Cafe, 6th International Colloquium on Coffee Chemisrty*, pages 61–72, Bogatá, Columbia.

Sundberg, R. (1974). Maximum likelihood theory for incomplete data from an exponential family. *Scandinavian Journal of Statistics*, **1**(2), 49–58.

Tipping, M. E. and Bishop, C. M. (1997). Mixtures of probabilistic principal component analysers. Technical Report NCRG/97/003, Aston University (Neural Computing Research Group), Birmingham, UK.

Tipping, M. E. and Bishop, C. M. (1999). Mixtures of probabilistic principal component analysers. *Neural Computation*, **11**(2), 443–482.

Titterington, D. M., Smith, A. F. M., and Makov, U. E. (1985). *Statistical Analysis of Finite Mixture Distributions*. John Wiley & Sons, Chichester.

Ward, J. H. (1963). Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, **58**(301), 236–244.

Woodbury, M. A. (1950). *Inverting modified matrices*. Statistical Research Group, Memorandum Report 42. Princeton University, Princeton, New Jersey.