Parallel Computation of High Dimensional Robust Correlation and Covariance Matrices

by

James Chilson

B.S., Western Oregon University, 2001

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard

The University of British Columbia

March 2004

Abstract

Currently, data mining applications use classical methods to calculate covariance and correlation matrices. These methods have the drawback that they can be adversely affected by data set outliers. Thus, robust methods for calculating covariance and correlation matrices are useful for these applications. However, robust methods require more time to calculate. To counter this, we propose two parallel robust methods of calculating correlation and covariance matrices. The first algorithm is a parallel version of Quadrant Correlation (QC), and the second is a parallel version of the Maronna method. Parallel QC uses a parallel matrix library and can handle single-dimensional outliers in its data. The parallel Maronna method divides the independent correlation calculations between the processors and is capable of detecting one and two dimensional outliers in data.

We evaluate these algorithms using a dataset from a "real-life" application. It is a genetic data set that comes from cardiovascular research, and it contains 6068 variables. Our evaluation also includes performance results from datasets with varying dimensions, performance of several algorithm components, a communications analysis, and improvements for the Maronna method.

From our results we conclude that our parallel algorithms make the robust calculation of correlation and covariance matrices useful in applications that deal with large dimensional data, such as data mining. Our initial hypothesis was that Maronna would perform better in parallel than QC, to the point that Maronna would be faster. In actuality, we found that Maronna does work better in parallel than a parallel QC in that it scales to more processors. However, our experiments do not show the parallel Maronna takes less time. Our conclusion is QC and Maronna are two viable options for computing robust correlation and covariance matrices. QC is less robust, fast, but does not scale as well to many processors while Maronna takes longer, is more robust, and scales to many processors.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank Dr. Alan Wagner for his supervision and help, Dr. Raymond Ng and Dr. Ruben Zamar for their crucial advice and guidance, and the DSG people for putting up with me.

JAMES CHILSON

The University of British Columbia
March 2004

To my friends and family. Without your support, this journey would not have been possible.

Chapter 1

Introduction

1.1 Motivation

One important analysis of any data is to discover the relationships between the variables. Correlation and covariance are statistical measures of the linear relationship between two variables. When considering several variables, a correlation/covariance matrix can be formed using the correlations/covariances between all the variables. The calculation of correlation and covariance matrices plays an important role in the field of data mining. Data mining is the extraction of useful data or trends from large databases. This is relevant in current times because memory is readily available in large quantities and there are datasets which contain many thousands of variables and records, such as in genetics, oceanography, customer transaction data, and web data. The size of these datasets makes it challenging to effectively compute data mining tasks, including correlation and covariance.

The use of correlation and covariance in data mining fits into the areas of clustering and classification. In classification, covariance matrices are used in the principle components analysis phase to find the most correlated variables in the data. Principle components analysis is used to order the variables in decreasing order of their variability. Then, the number of variables can be reduced if one is interested in just the most wide-ranging variables. For example, covariance matrices are used in clustering to reduce the number of dimensions in the data so that calculations are performed on smaller dimensioned data. For a covariance calculation to be useful in these applications, it needs to be able to handle the large data matrices common in data mining.

Most data mining applications that use correlation and covariance calculations use classical methods to compute them. These methods are not always adequate for applications' needs. In some cases, datasets

can be corrupted with bad data, and we want to calculate the correlation or covariance without having these values spoil the results. This is especially true in biomedical applications, some of which are forced to use small samples that are more easily corrupted by outliers. For example, research geneticists using microarray gene data can be affected by such outliers because in certain circumstances, microarray data can be affected by noise that creates these outliers [8]. It would be valuable if researchers could reduce the outliers' influence on their results in such a case.

In cases where outliers affect the quality of results for correlation and covariance calculation, robust methods are a good alternative to classical methods. Robust methods for correlation and covariance specialize in providing results that are less influenced by outlier data. One robust technique for covariance and correlation matrix computation is the Quadrant Correlation method (QC). QC performs a pairwise calculation of correlation/covariance and is capable of detecting single dimensional outliers. It runs in a reasonable amount of time, but its robustness is limited to only single dimensional outliers. Another robust technique is the Maronna method. Maronna also computes the correlation/covariance in a pairwise fashion, but it can detect one and two dimensional outliers. This added ability comes with a far greater computational cost so that Maronna has generally been viewed as unuseable for all but the smallest data sets.

The performance of robust methods for calculating correlation and covariance must be improved if these methods are going to be used to compute those matrices for large datasets in data mining applications. We hypothesize that using parallel methods to calculate robust covariance/correlation will improve performance to the point where such calculations can be used for data mining applications.

QC and Maronna are both parallelizeable. Their algorithms both consist of two parts, one part performs median calculations for the variables, the other works on the correlation/covariance calculation. The median portion is easily parallelized by dividing the variables between processors and calculating the medians distributively, or if there are few variables, have the processors work together to find the medians. The correlation part is where the two algorithms differ. QC's correlation contains mostly vector operations and a matrix multiply. Matrix multiply is a well studied parallel problem, thus we create parallel QC by using a parallel matrix library, such as PLAPACK [32], which uses efficient parallel matrix multiply and other vector operations. Maronna's correlation is parallelizeable because the algorithm considers each of the correlation entries in the resulting matrix to be independent calculations. For each pair of variables, the algorithm uses an iterative process that converges to the correlation between them. These correlations can easily be divided between the processors so that all of them calculate their portion of the resulting correlation matrix in parallel.

1.2 Problems

This thesis focuses on the creation, implementation, and performance of a parallel robust parallel covariance/correlation algorithm. We consider two algorithms to parallelize, Quadrant Correlation and the Maronna method. The two algorithms break down into similar component parts, the median calculation for the data variables, the correlation calculation, and the positive definiteness repair of the covariance matrix. We look at the individual components to find how to make improvements to them through parallelization. Next, we look for methods of further improving the parallel version of Maronna. Finally, we investigate how the parallel algorithms perform, especially when we run them on a larger number of processors.

1.3 Approaches

The correlation calculation component of QC and Maronna are the main parts of the algorithms, and we used different approaches to parallelize them. As described earlier, we used a parallel matrix library, PLAPACK, to make the matrix and vector operations in QC parallel. For Maronna, we changed the algorithm so that it partitions the correlation matrix between the processors and each processor calculates and fills in the correlations for their partition. Our median component is adaptive and depends on the size of the dataset. It can use a sequential method, a parallel method where each processor performs the sequential median algorithm on a subset of variables, or a parallel method that has groups of processors running a parallel median algorithm for each variable. The last component we parallelized repairs the positive definiteness of the correlation matrix. To do this we find the negative eigenvalues by using a parallel partial eigensolver, then calculate replacement values for the negative eigenvalues to rebuild a positive definite correlation matrix.

Our improvements of the parallel Maronna fall into two areas, load balancing and convergence. We implemented a dynamic load balanced version of Maronna using a processor farm approach to ensure processors have enough work to stay busy throughout the calculation. Maronna uses an iterative process to calculate each correlation, so an improvement in convergence benefits the algorithm. We experimented with improving convergence by manipulating the internal parameters of the Maronna algorithm.

For our investigation of QC and Maronna's performance, we began with a communication analysis of our implementations. This includes a measure of the communication primitives used and a profile of the overall communication during runtime. In addition we experimented with the algorithms on the large cluster environment of WestGrid [35].

1.4 Overview

Parallel Maronna and QC are both good algorithms in different ways. QC is fast, while Maronna takes longer but continues to speedup with a larger number of processors. The two algorithms work well, but for different types of problems. QC should be used when only a few processors are available and you need the solution quickly, but with only a medium amount of robustness. Maronna is a good choice when a lot of robustness is required and many processors are available. In addition, it is desirable to have adaptive components in the algorithms that depend on the dataset dimensions because this can improve runtime as well. Also, effective means of improving Maronna include modifying the internal parameters of the algorithm and the implementation of dynamic load balanced Maronna.

1.5 Outline

The thesis is organized as follows. Chapter 2 covers the background information. Section 2.1 focuses on cluster computing. Section 2.2 is about robust correlation and covariance including an example and components to robust correlation and covariance algorithms. Section 2.3 describes how correlation and covariance are used in data mining.

Chapter 3 covers the algorithms we use to calculate robust correlation and covariance in parallel. Section 3.1 gives a general overview of both our algorithms. Section 3.2 describes our algorithm for calculating median and MAD. Section 3.3 focuses on our method of repairing the positive definiteness of the correlation matrix. Section 3.4 describes our parallel QC algorithm and Section 3.5 describes our parallel Maronna algorithm.

In Chapter 4 we give the results of our experiments. First we describe the experimental setup in Section 4.1. Chapter 4 continues in Section 4.2 with or results analysis for the parallel QC and Maronna algorithms. Section 4.3 covers the I/O performance of our algorithms and Section 4.4 describes our results for repairing the positive definiteness of covariance matrices. We use Section 4.5 to explain our experiments with load balancing the Maronna algorithm. Section 4.6 describes our experiments in varying the size of input data. Finally, Section 4.7 covers our communications analysis of QC and Maronna.

Chapter 5 gives our conclusions and describes some areas for future work.

Chapter 2

Background

The background describes some basic knowledge of the areas that relate to this thesis. Section 2.1 discusses parallel cluster computing, Section 2.2 describes robust correlation/covariance computation, and Section 2.3 focuses on how correlation and covariance are used in data mining.

2.1 Cluster Computing

Parallel computing has two main areas for architectures, shared memory machines and clusters. In this thesis, we are concerned with the cluster architecture. A parallel cluster is made of separate machines, each having its own processor and memory, which are connected together by a network, such as Ethernet. The machines work together by sending messages to one another across the network.

Cluster computing has advantages over single processor computing because cluster computing combines computer power and resources across several computers. If a computation is too large for a single machine, a cluster of machines can provide more computing power to calculate the result faster. Similarly, if there is not enough memory in a single machine to compute a memory intensive job, a cluster can be used to divide the problem across multiple machines to compute the result in a distributed fashion. In an ideal circumstance, if p machines in a cluster are used to perform a computation, it takes $\frac{1}{p}$th the time of the single processor version. This is not usually true in practice because there is an overhead cost to the parallelization and a communication cost associated with passing messages between machines.

Cluster computing is becoming more competitive with other methods of supercomputing. Standards such as MPI [10] let developers create software that is more portable between clusters. Todays commodity processors and equipment are more affordable so that it is more cost effective to assemble a cluster than

to build a supercomputer. These components are also gaining in performance, as can be seen by their presence in the list of 500 top supercomputers [23]. Computing clusters comprise forty percent of the list, and have seven of the top ten positions. The success of computing clusters has increased the interest in grid computing, which involves combining computers or clusters in separate geographic locations to form a system. Users in a grid computing environment only see the system as a large single computer, and the system takes care of all the technical details of combining the resources together for the user. One example of grid computing research is WestGrid in Canada [35].

In the case of calculating robust covariance/correlation, the computation is processor intensive. Thus, in our experiments, we show that by using a cluster with enough processors, parallel robust covariance/correlation provides users with performance that is of interest for data mining applications.

2.2 Robust Correlation and Covariance Calculation

2.2.1 Correlation and Covariance

Covariance measures the degree of linear association between two variables. If the covariance between two variables is high, a change in one variable will bring a similar linear change in the related variable. When the covariance is very negative, the two variables are inversely linear related to one another. Zero covariance implies there is no linear association between the two variables.

Formally, covariance is defined as

$$\text{cov}(X_i, X_j) = E[(X_i - \mu_i)(X_j - \mu_j)]$$

where X_i and X_j are the two column variables, and μ_i and μ_j are their respective means. For a data set with several variables, the covariance matrix is a matrix that contains all the covariance values for every combination of variables. Thus position A_{ij} in the matrix contains the covariance between variables i and j. The correlation between two such variables is the covariance normalized so that it lies between the values -1 and 1. The normalization occurs by dividing the covariance by the product of the standard deviations for the two variables.

$$\text{corr}(X_i, X_j) = \frac{\text{cov}(X_i, X_j)}{\sigma_i \sigma_j}$$

Variables that are directly related will have correlations near one, while those inversely related will have correlations around -1. Again, the correlation is near zero for unrelated variables.

2.2.2 Robust Correlation and Covariance

Robust covariance matrix estimation is a popular area in statistics. The robustness of estimators is determined by their breakdown point, the maximum amount of contamination that an estimator can handle. Of particular interest are the estimators that are positive definite, affine equivariant, and have a high breakdown point near one-half. The reason positive definite estimators are desired is that it does not make sense for a covariance matrix not to be positive definite. Covariance matrices can be represented geometrically as a multidimensional ellipsoid that describes the multivariate scatter of the original data. The eigenvectors of the covariance matrix are the axes of the ellipsoid and the eigenvalues of the covariance matrix are the lengths of these axes. These lengths must be positive, so we only want to consider estimators that are positive definite.

The property of affine equivariance is also related to the covariance ellipsoid. If we apply an affine equivariant transformation to the original data set, then the resulting covariance ellipsoid after applying the transformation is the same as the covariance ellipsoid of the resulting transformed data. This is a good property because if an estimator can detect single dimensional outliers and is affine equivariant, then it is known to be able to detect structural outliers as well. The Maronna method is one example of such an estimator.

Estimators with the properties we desire fall into several classes. One class includes the S estimates, such as the Minimum Volume Ellipsoid (MVE), and another class is the Minimum Covariance Determinant (MCD) estimates [28, 29]. There is also a class for the projection based estimates, such as the Stahel-Donoho estimate (SDE) proposed by [31] and [7], and studied by [22, 17]. Yet another class is the P-estimates [21].

The major problem with these estimators is they are very time consuming to compute, requiring on the order of 2^v operations, where v is the number of variables. The projection estimators further require n^2 operations, where n is the number of cases. This is unacceptable due to the large size of datasets in data mining, which can have thousands of variables and hundreds of thousands to millions of cases. The reason for the high level of complexity is that computing these estimators comes down to solving a highly non-convex optimization problem, which consists of trying to find a good initial estimate that has the global optimum as its nearest local optimum. The initial estimates are usually computed through repeated random sub-sampling of N_s rows of the dataset, where N_s is chosen to result in a high breakdown point with high probability, .99 or .999 (see for example [27]). This results in the exponential complexity.

A quicker alternative is the "Fast MCD" (FMCD) [26]. FMCD is better than using naive subsampling because it can produce "good" solutions using a much smaller N_s. The downside is the running time

for FMCD is still too long for a large number of variables and it does not have a high breakdown point when the number of cases is large.

It is possible to compute faster estimates that have high breakdown points if we do not require the resulting matrix to be affine equivariant. The easiest of such methods are those based on pairwise estimates which have overall high breakdown points because each of the individual pairwise estimates is calculated with a high breakdown point. This reduces the complexity in terms of the number of variables from exponential to quadratic (from 2^p to p^2). One category of these estimates is (i) the classical rank based methods, such as the Spearman's ρ and Kendall's τ (see for example [1]). Another is (ii) classical correlations applied after coordinate-wise outlier insensitive transformations such as the quadrant correlation (QC) and 1-D "Huberized" data (see [15], p. 204). Also, (iii) bivariate outlier resistant methods such as the method proposed by [11] and studied by [6]. The smaller running time for theses pairwise methods make them more appropriate for use in data mining, especially on the datasets with hundreds of variables.

More recent approaches are based on modifications of the pairwise methods. the Gnanadesikan and Kettenring (MGK) approach [19] modifies (iii) and a version of quadrant correlation modified from (ii) [2] improve on the previous approaches in that they have complexity $\mathcal{O}(np^2)$. We chose the version of QC in [2] to parallelize to extend the use of robust correlation in data mining to datasets with thousands of variables, such as the gene dataset of 6068 variables we use in our experiments.

The problem with these pairwise estimators is that they were created with speed in mind instead of robustness. Their lack of affine equivariance means that they will be susceptible to two-dimensional structural outliers. An alternative choice is to use the Maronna M-estimate [20] because it has a higher quality robustness. The Maronna covariance matrix estimate is positive definite, affine equivariant, and can be computed using a re-weighting algorithm. This computation is much too expensive for sequential approaches as we will see in the Experiments chapter, but we choose this method to parallelize because of its better robustness.

2.2.3 Robust Examples

A comparison of the robustness between the classical Pearson correlation, QC, and Maronna is in Figure 2.1. Figure 2.1 shows the three methods used on clean data, data with two dimensional structural outliers, and data with large two dimensional structural outliers. We see that the classical method performs badly and is affected by the structural outliers and even worse by the large structural outliers, going from a .95

correlation to $.40$ and $-.22$. QC was also affected by the outliers in the data, with its correlation going from .98 to .64. The Maronna performs the best and is hardly affected by the outliers. The Maronna correlation changed from .96 to .91.

Another example of the advantage of two dimensional robustness is the Woodmod data set from the S-PLUS robustness library in Figure 2.2. In the rectangles of V1-V2 and V4-V5, both have a small group of two dimensional outliers in one of their corners. Both sets of outliers are not visible when viewing the data along a single dimension. The correlations for V1-V2 and V4-V5 using traditional methods are $-.14$ and $-.24$. However, a robust calculation method shows the correlations are $.85$ and $.65$. This is a vast difference from the non-robust approach. This makes it clear that a robust method is preferred when calculating correlation and covariance matrices.

Even though a robust approach is computationally expensive, there are applications that would benefit from robust correlation and covariance matrix calculations. One such area is bio-medicine, especially genetic related fields. For example, one application of covariance matrix calculations is to examine the genetics of rheumatic and normal heart valves to help identify which genes are responsible for generating rheumatic valves. To do so, it is useful to look at both the genes' correlation, and how the genes fall into clusters based upon their correlations. Figure 2.3 shows an example dendrogram of gene correlation clusters. The dendrogram is formed by first calculating the correlation matrix for the gene expression data. Then group the correlations into clusters using a clustering algorithm. Finally, create the dendrogram based on the clusters. The height in the diagram relates to the gene correlation by $1 - r$, where r is the correlation. The robust methods are especially effective for this application because microarray gene data is affected by noise [8] which can cause outliers in the data that skews the results. In addition, the data usually has few samples, which makes it more important to downplay the outliers to achieve accurate results. Gene data usually has a large number of variables as well, so only a robust technique that could process large data sets would be useful for this application. Thus, it is important to develop the techniques to handle robust approaches to calculating high dimensional covariance and correlation matrices.

2.2.4 Component Algorithms to Robust Correlation/Covariance

There are a couple of algorithms that are key components in calculating covariance and correlation matrices. One such algorithm calculates the median and MAD (Median Absolute Deviation). The median and MAD act as robust calculations of the mean and standard deviation. The algorithms for both are similar, so a

median algorithm can calculate MAD with minor alterations. Sequential median algorithms range from linear and average linear time, such as [5, 10.3] and [5, 10.2] that use partitioning, to $\mathcal{O}(n \log n)$ algorithms that sort the data using quicksort. Most of the parallel implementations of the median algorithm are based on shared memory architectures, where there is not as great a penalty for communicating data between processors. Existing algorithms for distributed memory architectures, such as [30], attempt to reduce the amount of data communication between processors. One of the best parallel median algorithms uses the PRAM model and has $\mathcal{O}(\log n \cdot \log \log n)$ running time on $\Theta(\frac{n}{\log n})$ processors [16]. [4] has suggested an algorithm that has a worst case time that matches [16] in runtime, but has an expected runtime of $\Theta(\log n)$ with the added restriction that the input values are distributed randomly on an interval $[a, b]$. Finally, [13] provides an optimal version of the parallel selection algorithm for EREW PRAMs that takes $\mathcal{O}(\log n)$ time using $\frac{n}{\log n}$ processors.

The operations performed in QC and Maronna do not preserve the quality of positive definiteness of the correlation matrix. The algorithms must repair the matrix if we desire the property of positive definiteness. Our implementations use an eigensolver to identify the negative eigenvalues so they can be repaired. Various algorithms exist for calculating eigenvectors. The power method is one of the simplest. It solves for extreme eigenvalues of the matrix and a variant can be applied repeatedly to solve for multiple eigenvalues [9, 7.1]. QR-factorization uses a set of similarity transforms to convert the matrix into a form that is easier to solve for eigenvalues [9, 7.3]. Another method used widely in applications such as Matlab is the Arnoldi method, or Lanczos method in the case for real symmetric matrices. The Lanczos method works well for finding extreme eigenvalues, and can find the eigenvalues in the inner portion of the spectrum when functions are used to remap those eigenvalues to the extreme points [34, 5.3].

2.3 Correlation and Covariance in Data Mining

Covariance and correlation matrices have applications in many fields, and one important use is in the area of data mining. Data mining applications deal with very large datasets that can have hundreds of thousands of variables and millions of cases. However these datasets can come in a variety of shapes. For example, some gene datasets will have data on thousands of genes, but only a few samples, such as the rheumatic heart valve gene dataset we use for experiments. Other datasets can have the opposite dimensions, such as atmospheric or oceanographic data, which can have a few thousand variables for the locations they measure and many thousands of samples, one for each measurement.

There are several types of applications that covariance matrices are used for in data mining. These applications include selection of relevant variables to analyze, methods of outlier detection, and as a key component in the processes of clustering and classification.

Clustering is a method of dividing data into groups with similar characteristics. Clustering requires queries and information retrieval on large datasets that can span many dimensions. A fast way of performing these operations is to try to reduce the dimensionality of the dataset. One technique for doing so is to form the covariance matrix for the data and find the principal component eigenvectors and eigenvalues. The principal components are used to form a new, smaller dimensional space with projected data points from the original data.

Covariance matrices are also used in classification data mining. Classification is a means of creating rules that govern which category a data object is filed under. These rules are constructed by examining a small set of already categorized data. Algorithms use these rules to separate new data values into preexisting classes. Classification also uses principle component analysis, and hence covariance matrices, to determine what features of the dataset should be used by the rules to divide the data into classes.

These examples show that covariance and correlation calculations play a part in the field of data mining. If we improve the robust calculation of covariance and correlation matrices through parallelization, all these areas in data mining will benefit.

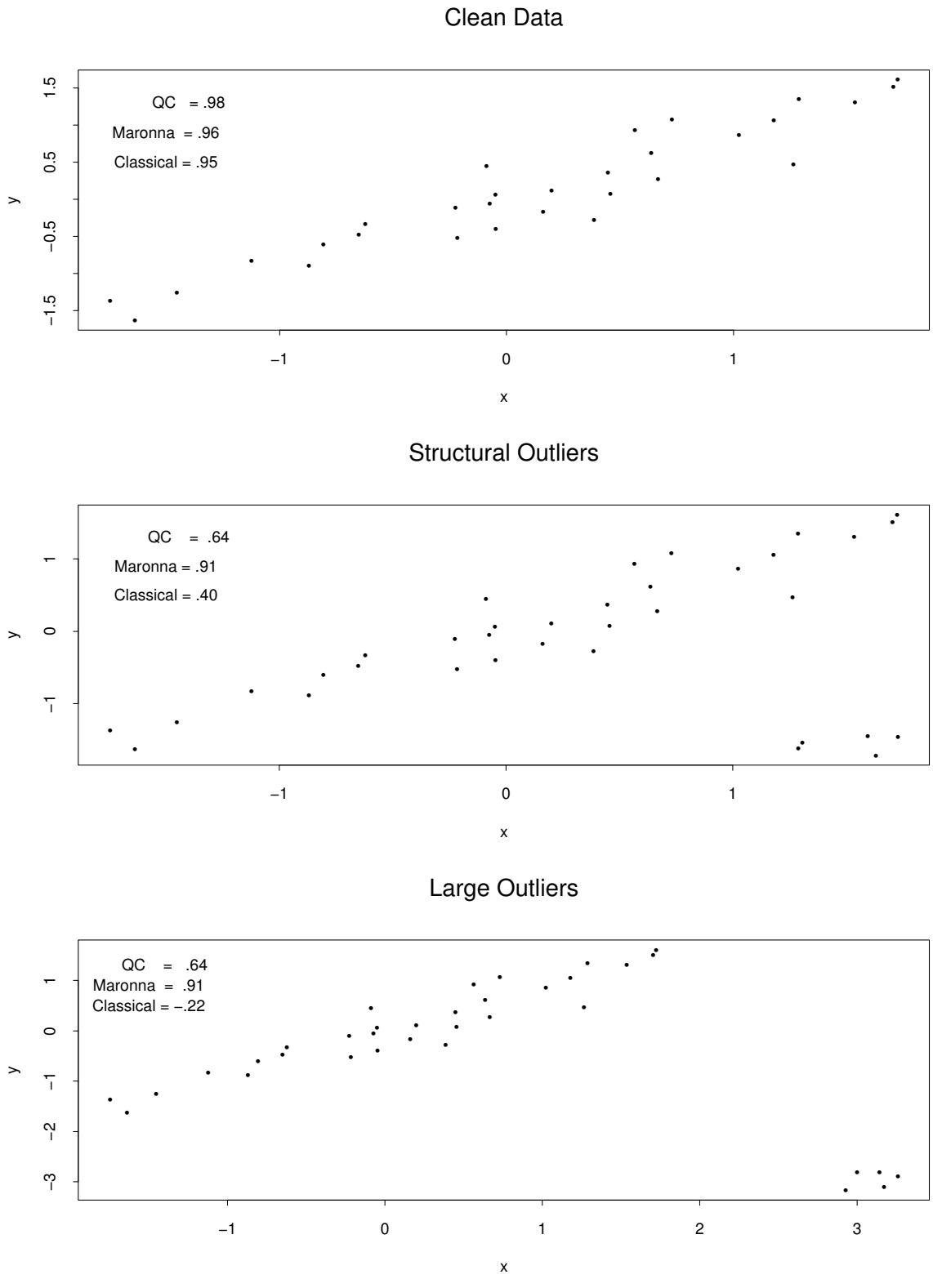

Figure 2.1: Advantages of Maronna Over QC and the Classical Pearson Correlation.

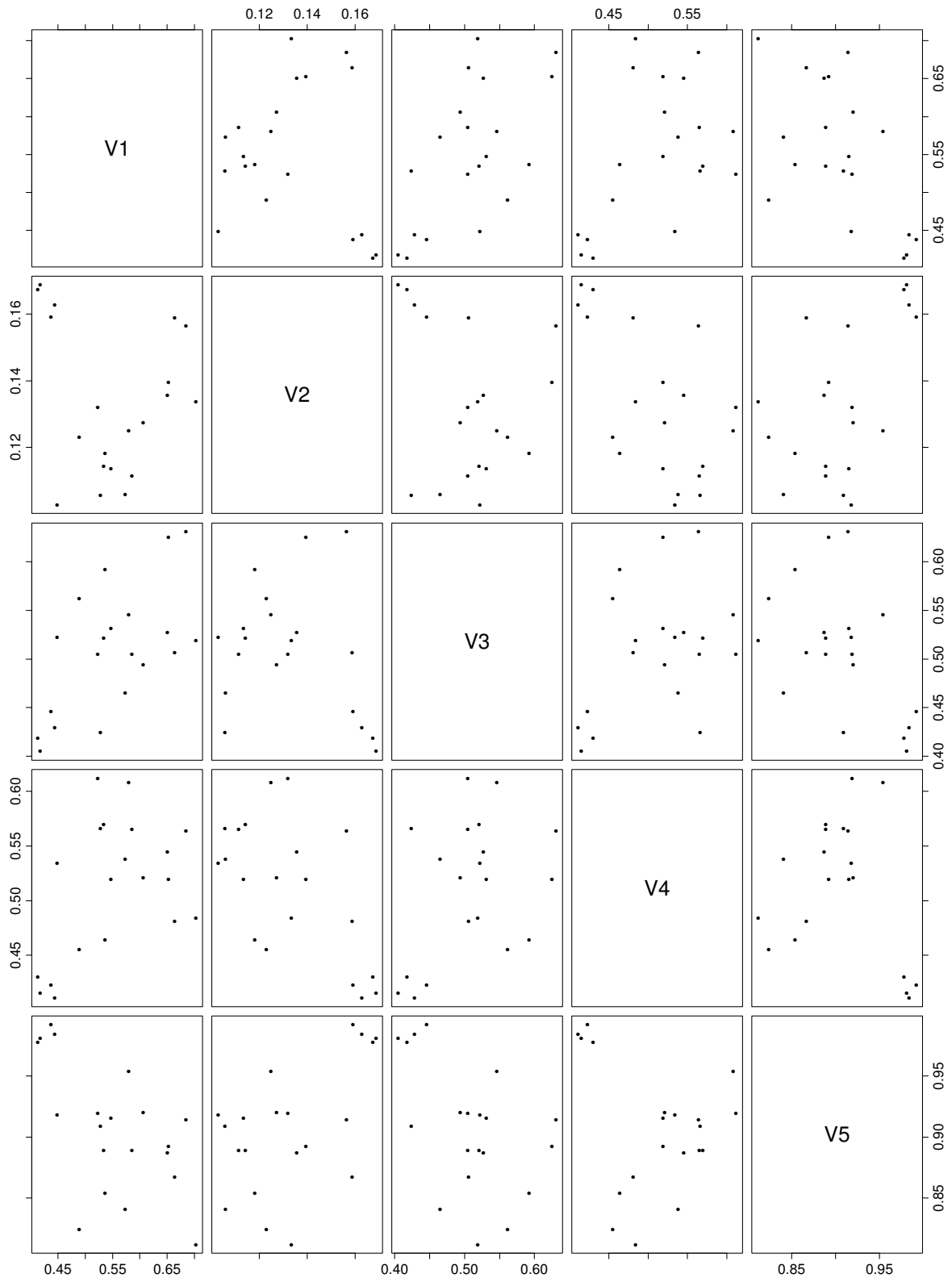

Figure 2.2: Robustness Results for a Small Five-Dimensional Data Set.

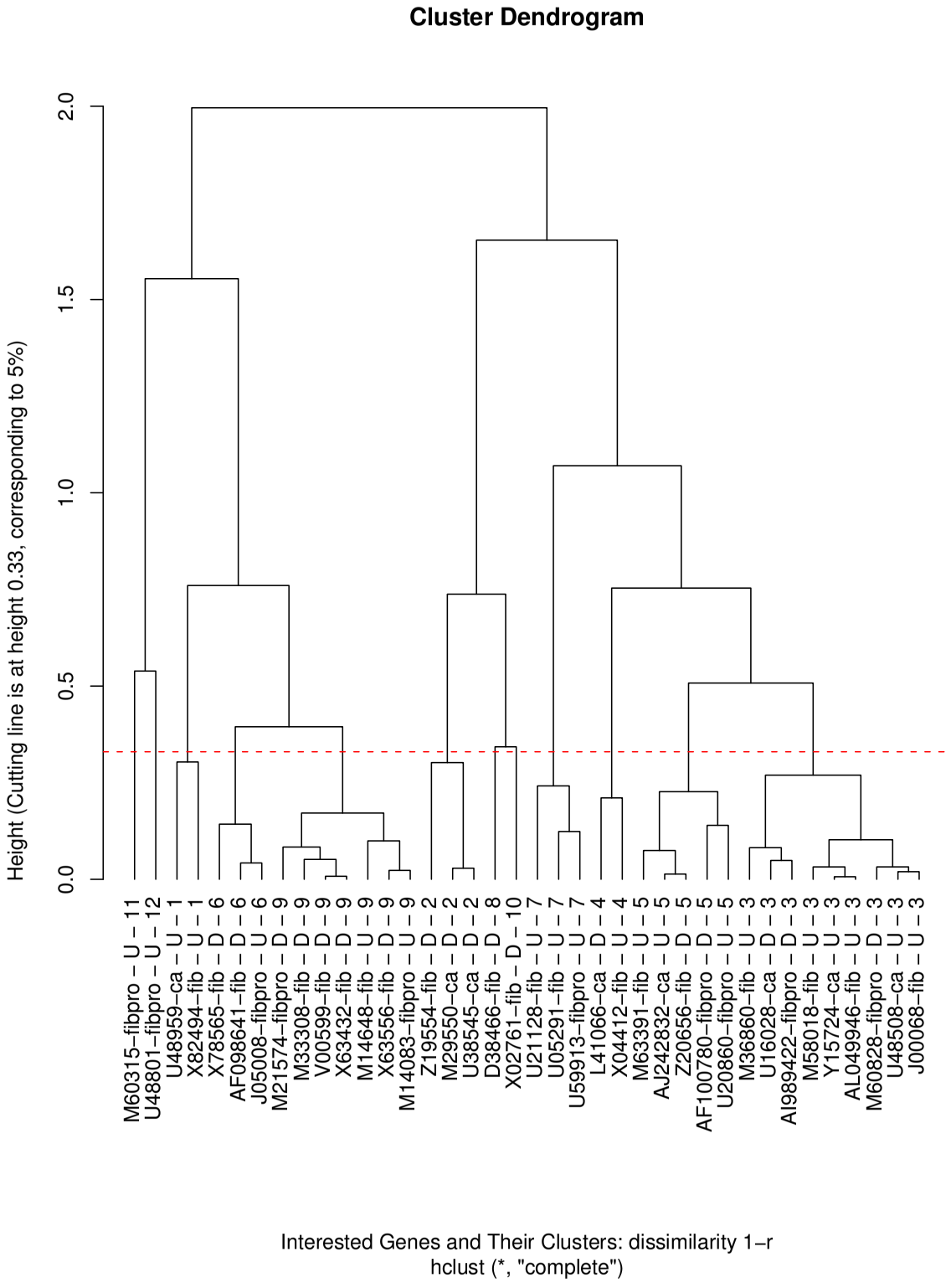

Figure 2.3: Correlation Dendrogram for Gene Data.

Chapter 3

The Algorithms

Two algorithms were implemented for calculating correlation and covariance. This chapter is divided into five sections, an overview describing the main parts of the algorithms, the median/MAD calculation step, using an eigensolver to repair positive definiteness, calculating correlation and covariance using QC, and calculating correlation and covariance using the Maronna method.

Our notation is to represent the number of variables as v, the number of cases as n, and the number of processors as p. In the algorithms, we treat matrices as being in column-major order, so that $MAT[x][y]$ is the item in the xth column and yth row and $MAT[x]$ represents the xth column. We represent a single row, in this case row x, by MAT_x.

3.1 General Algorithm Overview

The general setup for the calculation is similar for both QC and Maronna. The data is read in from disk and distributed to the other processors, then the median and MAD are calculated for each variable. Next, the covariance matrix is calculated. Afterwards, we use the eigensolver to repair positive definiteness, then convert the matrix into a correlation matrix if necessary. Finally, the matrix is gathered to the root processor where it is written to disk.

There are several parts of the parallel algorithms that are similar between QC and Maronna. The input and output procedures are the same; a dataset with n cases, where each case deals with the same v variables. The dataset is stored in column major order. When the data is read by the program, the program stores the transpose of the data matrix. The output for both QC and Maronna is a v-by-v correlation or covariance matrix, depending on which matrix the user wants. In the output matrix, the element at the

intersection of column i and row j is the correlation or covariance between the i^{th} variable and the j^{th} variable. Both algorithms can use the same I/O routines, and they are optimized to read/write binary data in row blocks. Both algorithms need to distribute the data and collect the answer, however, this will vary depending on the algorithm used.

3.2 Median and MAD Calculation

QC and Maronna use the median and MAD values calculated for each variable. MAD stands for the median absolute deviation from the median. MAD represents the scatter of the data, and is a more robust measure than the standard deviation. The definition of MAD is

$$\text{MAD}(X[i]) = \frac{\text{median}(|X[i][j] - \text{median}(X[i])|)}{0.6745}$$

The $.6745$ here represents the inverse of the third quartile of the normal distribution. The division is necessary because the numerator alone tends to underestimate the standard deviation, so dividing by this value makes the MAD more accurate. From the definition, it appears that, once the median values are calculated, the MAD is quite similar to the median calculation. The same median routine used in the algorithm can also be applied to calculate the MAD with the help of a few small changes.

Initially, we planned to use a sequential median algorithm since correlation is more computationally complex. However, if the number of variables in the input data is small, the resulting correlation matrix is small and requires less work to calculate than a matrix of larger dimensions. For example, a dataset with twenty variables and 50000 cases requires the calculation of 400 correlations. If the dataset had 1000 variables, there would be 1000000 correlations to calculate. Even though the number of medians to calculate also increases with the number of variables, it is only related linearly, while the number of correlations to calculate increases quadratically. In the case with fewer variables, the median would take a larger percent of the total time, so performance gains from parallelizing the median would have a larger effect.

There are three options for the median calculation. The first is to use a sequential median algorithm. [5] provides median finding algorithms that run in average case $\mathcal{O}(n)$ time. These algorithms are most easily implemented in a recursive fashion, and they are based on a partition scheme similar to quicksort. Even though we are interested in a parallel algorithm, a sequential version of the median algorithm is still useful because there are datasets that are small where the sequential algorithm is faster than a parallel algorithm since the sequential version does not have the overhead from parallelization.

Still, we want a parallel version of the median to deal with large datasets. Two types of parallelization are possible. Given that there are many variables, one method is to divide the variables among the processors and have each processor calculate the medians for its variables using the sequential version of the algorithm. The complexity for this version is $\mathcal{O}(n \cdot \frac{v}{p})$

The second approach handles the case where we have more processors available than variables in the dataset. We can maximize the use of processors by dividing the processors into groups, where each processor group works together to calculate the median for a single variable. The algorithm proceeds by dividing the variable column between the processors in the group. In this case, each processor of the group sorts its piece of the variable column using quick sort, then the processors decide on the lowest and highest medians between the processors. Next, the processors merge the portion of their sorted variable columns that lie between the largest and smallest of their medians, and then select the global median or MAD for the variable from the combined chunk. The time complexity here is $\mathcal{O}(n \log n \cdot \frac{v}{p} + \log \frac{p}{v})$.

3.3 Repair Positive Definiteness via Eigensolving

The operations performed in the QC and Maronna algorithms do not preserve the property of positive definiteness in the covariance matrix. Both algorithms require a step to repair the matrix. This repair consists of two parts, finding the desired eigenvalues and using these eigenvalues to repair the matrix.

The easiest way to find the negative eigenvalues is to do a complete eigensolve of the matrix and examine the eigenvalues. We began by developing a sequential eigensolver that uses the QR method and had planned to parallelize it. There are several existing parallel eigensolvers, and in fact, the PLAPACK package we were already using contained one that performs a complete eigensolve. After adding the full eigensolver to our program, we realized that a complete eigensolve is an expensive operation for large matrices. The PLAPACK implementation first transforms the matrix into tri-diagonal form, then uses a tri-diagonal matrix eigensolver to find the eigenvalues and eigenvectors. Typical tri-diagonal reduction techniques require $\mathcal{O}(n^3)$ time, while the tri-diagonal eigensolve takes $\mathcal{O}(n^2)$ time. Thus, performing a complete eigensolve on a large matrix can be time consuming and take hours of time.

There is another option besides complete eigensolvers. Partial eigensolvers exist that allow users to solve for a few eigenvalues and eigenvectors from different parts of the spectrum, such as the largest algebraic or smallest absolute. We switched our eigensolver for one of these with the idea that only a few eigenvalues would be negative and the eigensolver would be able to find them quickly. However, tests

using the gene dataset show that there were more negative eigenvalues than we expected. It became clear that different datasets could have vast differences in the number of negative eigenvalues in their respective covariance matrices. Thus, we needed a flexible method of isolating the negative eigenvalues.

Our method for finding negative eigenvalues is based on the idea that we want to solve for the least number of eigenvalues necessary. The number of negative eigenvalues is somewhat related to the dimensions of the original data matrix for the problem. If the number of variables is less than or equal to the number of cases, there will be only a few negative eigenvalues. These are the ones that were corrupted by our calculation. When the number of cases is less than the number of variables, there will be as many nonzero eigenvalues as cases in the original data matrix. The rest of the eigenvalues should be zero, but a small amount of error in the calculation will result in these eigenvalues being slightly positive or negative. About half of these will fall on the negative side of zero if we assume the error is random. If the number of cases is small, less than one third of the number of variables, we can simplify things by just solving for the largest eigenvalues instead of the negative ones. Otherwise, it would be less work to solve for the negative eigenvalues, both the ones corrupted through our calculation, and the zero eigenvalues turned negative through error.

Once the eigenvalues are known, whether positive or negative, the covariance matrix is repaired by forming a new matrix by subtracting a matrix formed by the negative eigenvalues and their eigenvectors from the original corrupted covariance matrix. If some of those negative eigenvalues were caused by the corruption of the covariance matrix, we recalculate positive replacement eigenvalues for them and use the negative ones' eigenvectors to form a matrix and add it to the original covariance matrix to replace the negative ones we subtracted out. If we solved for the largest eigenvalues instead, they only need to be combined with their eigenvectors to form the new matrix, unless some of the largest were negative, in which case we fix them with new eigenvalues in the same manner as before.

The following small example illustrates the general process of positive definiteness repair.

Example:

Let $A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$ be the original covariance matrix.

Let $eig = \begin{bmatrix} eg_1 & eg_2 \end{bmatrix}$ be the negative eigenvalues for A.

Let $e = \begin{bmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \\ e_{31} & e_{32} \end{bmatrix}$ be the negative eigenvectors corresponding to eig.

Let $d = \begin{bmatrix} d_{11} & d_{12} & d_{13} \\ d_{21} & d_{22} & d_{23} \\ d_{31} & d_{32} & d_{33} \\ d_{41} & d_{42} & d_{43} \end{bmatrix}$ be the data matrix.

1. Calculate the negative eigenvalue matrix to be subtracted out.

$$n = \begin{bmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \\ e_{31} & e_{32} \end{bmatrix} \cdot \begin{bmatrix} eg_1 & 0 \\ 0 & eg_2 \end{bmatrix} \cdot \begin{bmatrix} e_{11} & e_{21} & e_{31} \\ e_{12} & e_{22} & e_{32} \end{bmatrix} = \begin{bmatrix} n_{11} & n_{12} & n_{13} \\ n_{21} & n_{22} & n_{23} \\ n_{31} & n_{32} & n_{33} \end{bmatrix}$$

2. Find positive replacement eigenvalues, $peig$, for eig using the MAD.

$$peig = MAD\left(\begin{bmatrix} d_{11} & d_{12} & d_{13} \\ d_{21} & d_{22} & d_{23} \\ d_{31} & d_{32} & d_{33} \\ d_{41} & d_{42} & d_{43} \end{bmatrix} \cdot \begin{bmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \\ e_{31} & e_{32} \end{bmatrix}\right) = \begin{bmatrix} peig_1 & peig_2 \end{bmatrix}$$

3. Calculate replacement eigenvalue matrix.

$$p = \begin{bmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \\ e_{31} & e_{32} \end{bmatrix} \cdot \begin{bmatrix} peig_1 & 0 \\ 0 & peig_2 \end{bmatrix} \cdot \begin{bmatrix} e_{11} & e_{21} & e_{31} \\ e_{12} & e_{22} & e_{32} \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ p_{31} & p_{32} & p_{33} \end{bmatrix}$$

4. Subtract n from A and add in p to replace it.

$$A_{new} = A - n + p$$
$$= \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} + \left(\begin{bmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \\ e_{31} & e_{32} \end{bmatrix} \cdot \begin{bmatrix} -eg_1 & 0 \\ 0 & -eg_2 \end{bmatrix} \cdot \begin{bmatrix} e_{11} & e_{21} & e_{31} \\ e_{12} & e_{22} & e_{32} \end{bmatrix}\right)$$

$$+\left(\begin{bmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \\ e_{31} & e_{32} \end{bmatrix} \cdot \begin{bmatrix} peig_1 & 0 \\ 0 & peig_2 \end{bmatrix} \cdot \begin{bmatrix} e_{11} & e_{21} & e_{31} \\ e_{12} & e_{22} & e_{32} \end{bmatrix}\right)$$

$$= \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & 33 \end{bmatrix} + \begin{bmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \\ e_{31} & e_{32} \end{bmatrix} \cdot \begin{bmatrix} peig_1 - eg_1 & 0 \\ 0 & peig_2 - eg_1 \end{bmatrix} \cdot \begin{bmatrix} e_{11} & e_{21} & e_{31} \\ e_{12} & e_{22} & e_{32} \end{bmatrix}$$

The general process for repairing the positive definiteness of a covariance matrix is listed in Figure 3.1. Many times, we do not know the number of negative eigenvalues we are solving for ahead of time. In this case, we solve for a set proportion depending on the dataset size, and repeat a process of fixing and solving for more until no more negative eigenvalues exist. Using this method, we can hopefully avoid solving for all the eigenvalues. Also, Figure 3.1 covers the case of solving for and fixing the negative eigenvalues. As discussed earlier, we sometimes do not want to fix the negative eigenvalues, but to just discard them instead, as in the case where they are created from corrupted zero eigenvalues. Then, we would not create the replacement eigenvalues, but instead rebuild the matrix using the positive ones.

Input: v by v covariance matrix C
n by v original data matrix X
Output: v by v positive definite covariance matrix $C_{adjusted}$
1. **In parallel** Solve for the negative eigenvalues, $\lambda_k..\lambda_q$ of C
and eigenvectors $Q = a_k..a_q$
2. **In parallel** Form $Y = X \cdot Q$
3. **In parallel** Calculate replacements, $m_k..m_q$, for the negative eigenvalues:
4. $m_i =(MAD$ of column i in $Y)^2$
5. Let $A =$ Diag$(\lambda_k..\lambda_q)$ and $B =$ Diag$(m_k..m_q)$, diagonal matrices
6. **In parallel** $C_{adjusted} = Q \cdot (B - A) \cdot Q^T$

Figure 3.1: Parallel Algorithm for Repairing the Positive Definiteness of a Correlation Matrix

At worst, the eigensolver will need to solve for a little over one third of the eigenvalues. Hopefully, we can get away with solving for much fewer eigenvalues if the data matrix has the right dimensions, such as one that is almost square, so that there are fewer negative eigenvalues present.

The rest of the algorithms depend on the method, either QC or Maronna.

3.4 Parallel Algorithm for Quadrant Correlation

Figure 3.2 describes the parallel version of QC. In it, we represent the matrices in column-major order, where the columns are variables. Thus $X[i]$ refers to the ith column (variable). After calculating the median and MAD for all the variables, the algorithm creates a temporary matrix to hold the normalized values.

$$X[i][j] = \frac{X[i][j] - \text{median}(X[i])}{\text{MAD}(X[i])}$$

The X matrix is used to create a matrix Y of all 1's, -1's, and near zero values by applying a function, ψ, that is similar to the sign function, to all the elements in X.

$$\psi(x, c) = \begin{cases} \text{sign}(x) & \text{if } |x| > c \\ \frac{x}{c} & \text{otherwise} \end{cases}$$

Our sign function cuts off the values within c of zero and assigns them to be the value $\frac{x}{c}$. Our choice for c in the code was .00001. In actuality, by our ψ function, we are using a Huberized estimator which, in the limiting case, is Quadrant Correlation [2]. The limiting case here would be to use the sign function in place of ψ.

In the next step, the algorithm calculates the following equation to fill in each entry of the correlation matrix.

$$\text{cor}(i, j) \quad = \quad \frac{\frac{1}{n}\sum_k(\text{Y}[j][k]) \cdot (\text{Y}[i][k])}{\sqrt{(\frac{1}{n}\sum_k(\text{Y}[j][k])^2 \cdot \frac{1}{n}\sum_k(\text{Y}[i][k])^2)}} \tag{3.1}$$

The computationally expensive part of the calculation is the part where the numerator is calculated using a matrix multiplication between Y and its transpose in steps 6 and 7 of Figure 3.2. The operations involved are approximately $\mathcal{O}(v^3)$. The denominator is the geometric mean of the average number of nonzero elements for a pair of columns i and j. This part of the calculation takes $\mathcal{O}(v^2)$ time and is set up in step 10. The equation finishes in step 12 where the denominator divides the numerator. Again, this division occurs for every element in the matrix, thus step 12 requires $\mathcal{O}(v^2)$ time.

Overall, the matrix multiplication step dominates the runtime of the sequential QC algorithm since it requires $\mathcal{O}(v^3)$ operations. This gave us a place to start in creating a parallel algorithm. Parallel matrix multiplication is a well-studied algorithm, and it has been shown to be scalable to thousands of variables on large machines using a variety of different approaches [12]. First, we experimented with a simple matrix multiply routine [33]. It partitions one matrix and gives each processor a block made of rows from the

Input: v by n matrix X for v variables and n cases
Output: v by v matrix cov, the covariance matrix

1. **In parallel** For each column $X[i]$
2. Calculate the median and MAD
3. Let median$[i]$ be the median of column i.
4. Let MAD$[i]$ be the MAD calculation of column i.
5. Construct Y where $Y[i][j] = \psi(X[i][j] - \text{median}[i])/\text{MAD}[i], C)$;
6. **In parallel** compute matrix cor $= Y \cdot Y^T$;
7. **In parallel** $Y = \frac{1}{n} \cdot Y$; (element-wise)
8. For all i, j, set $Y[i][j] = (Y[i][j])^2$;
9. Construct vector D
10. $D[i] = \frac{1}{\sqrt{\text{avg}(Y[i])}}$;
11. **In parallel** For all i, j set
12. cor$[i][j] = \text{cor}[i][j] \times D[i] \times D[j]$
13. cor$[i][j] = \sin(\frac{\pi}{2} \cdot \text{cor}[i][j])$;
14. Construct diagonal matrix V where $V[i][i] = \text{MAD}(X[i])$;
15. **In parallel** cov $= V \cdot \text{cov} \cdot V$;

Figure 3.2: Parallel Algorithm for Quadrant Correlation.

matrix. Then, it broadcasts the second matrix to all the processors and each one performs a multiply to create part of the resulting matrix. This multiply routine scaled well, but it was slow.

Our second approach was to find an optimized parallel matrix library that includes matrix multiplication. We chose the PLAPACK library because it uses low-level libraries such as BLAS and LAPACK in its matrix routines. PLAPACK distributes its matrices in a more complex manner that is based on the distribution of vectors in vector operations. The PLAPACK matrix multiply is faster than our previous approach, but it is difficult to achieve good speedup because this requires careful adjustment to parameters within PLAPACK, such as the distribution block size, the dimensions of the processor mesh, and the algorithm block size.

Overall, the reason PLAPACK led to the best performance, despite the difficulties, is the parallelization of the multiply and other key parts of the algorithm reduce the computation time so that it does not dominate total runtime when compared to other components, such as I/O time and data distribution/collection. This gives us a faster, more practical QC algorithm, however QC still has the downside of a less robust solution when compared to the Maronna method.

3.5 Parallel Algorithm for the Maronna Method

Input: v by n matrix X with v variables and n cases
Output: v by v matrix cor, the correlation matrix

1. **In parallel** For each column $X[i]$
2. Calculate the median and MAD.
3. Let median$[i]$ be the median of column i.
4. Let MAD$[i]$ be the MAD calculation of column i.
5. **In parallel** For each pair of variables i, j
6. Initially
7. $\mu^{(0)} = \Big[\text{median}[i],\ \text{median}[j]\Big]$
8. $\sigma^{(0)} = \begin{bmatrix} (\text{MAD}[i])^2 & 0 \\ 0 & (\text{MAD}[j])^2 \end{bmatrix}$
9. Let $\mathbf{x}_q$ be the vector $[X[i][q],\ X[j][q]]$.
10. ITERATE
11. Given $\mu^{(k)}$ and $\sigma^{(k)}$
12. For $q = 1$ to n
13. $\text{mah}[q] = [\mathbf{x}_q - \mu^{(k)}] \cdot [\sigma^{(k)}]^{-1} \cdot [\mathbf{x}_q - \mu^{(k)}]^T$
14. $W[q] = \text{weight}(\text{mah}[q])$
15. Calculate
16. $\mu^{(k+1)} = \dfrac{\sum_{q=1}^{n} W[q] \cdot \mathbf{x}_q}{\sum_{q=1}^{n} W[q]}$
17. $\sigma^{(k+1)} = \frac{1}{n} \cdot \sum_{q=1}^{n} (W[q])^2 \cdot \Big[\mathbf{x}_q - \mu^{(k+1)}\Big]^T \cdot \Big[\mathbf{x}_q - \mu^{(k+1)}\Big]$
18. UNTIL (determinant $|((\sigma^{(k)})^{-1}(\sigma^{(k+1)}) - 1| < \epsilon$)
19. Let σ^* denote the converged σ for columns $X[i]$ and $X[j]$.
20. $\text{cor}[i][j] = \sigma^*[0][1]/\sqrt{(\sigma^*[0][0] \cdot \sigma^*[1][1])}$

Figure 3.3: Parallel Maronna Method

Figure 3.3 outlines the Maronna algorithm. Maronna begins with the median and MAD calculation described earlier. The algorithm's focus is to divide the p^2 correlation calculations between the processors, then the processors use an iterative algorithm to calculate each correlation assigned to them. The iterative portion consists of steps 5 to 18. First, the values involved in the iteration are initialized in step 7 and 8. μ is a vector of length two, and is initialized with the median of the data variables involved in this correlation

calculation. σ is a 2 x 2 matrix that will hold the estimate values for the correlation upon convergence. It is initialized as a diagonal matrix holding the MAD of the correlation variables in the diagonal. After initialization, the algorithm repeats the following process. The Mahalanobis distance is used to measure the distance between two variables' samples in step 13. The Mahalanobis distance measures the distance a data point is from the centroid of all data points. In our case, this is in the two dimensional space of the two variables, and takes into account the median of the data along with the estimated variance, covariance, and correlation at the current iteration.

Next, we apply a weight function to the distance values in step 14. Our weight function uses Huber's score function as the robust M-estimate to score the influence of the sample points to the median and variance. We use this to construct a weight function to decrease the influence of outliers in the data.

$$\text{HSF}(y) = \begin{cases} y & |y| \leq c \\ c \cdot sign(y) & |y| > c \end{cases} \quad \text{HUBER'S SCORE FUNCTION}$$

$$\text{weight}(y) = \begin{cases} HSF(y)/y & y \neq 0 \\ 1 & y = 0 \end{cases} = \begin{cases} 1 & |y| \leq c \\ c/|y| & |y| > c \end{cases} \quad \text{WEIGHT FUNCTION}$$

The weight function gives weights between zero and one that are applied to the data. The weight function will weight normal data variables near one and down-weight the outlier values with weights closer to zero. The outlier values are the ones with large distances from the distance function.

The weighted data is used to calculate new values for μ and σ for the next iteration in steps 16 and 17. Iteration continues until the change in covariance from one step to another is within the desired tolerance. The algorithm is known to converge, but the rate can vary depending on the input. Finally, on step 20, calculate the correlation by dividing the covariance value by the square root of the values on sigma's diagonal.

At this point, the algorithm calculates the correlation matrix for the data. If we want the covariance matrix instead, we convert the correlation matrix to a covariance matrix by scaling the rows and the columns of the matrix by the MAD values corresponding to the variables that the columns/rows represent.

$$\text{cov}(i, j) = \text{cor}(i, j) \cdot \text{MAD}(i) \cdot \text{MAD}(j)$$

Initially, when the iterative process converges in step 19, it will have calculated the covariance for the two variables. We do not want to use this initial covariance value in the covariance matrix because it was calculated in a pairwise fashion and does not represent the covariance relative to the overall global matrix.

Each iteration of the Maronna algorithm takes $\mathcal{O}(n)$ time. If we let k be the number of iterations needed for convergence, then each correlation requires $\mathcal{O}(nk)$ time. There are v^2 correlations to calculate between the processors, so the overall time cost is $\mathcal{O}(\frac{nkv^2}{p})$. Maronna's runtime could be significantly greater than QC depending on the value of k. Maronna does have the advantage that its tasks are independent, so they easily divide between the processors. The value of k can differ between the pairs of variables, so a load-balancing scheme is necessary to evenly distribute the tasks.

Chapter 4

The Experiments

The Experiments chapter begins with a description of the environment we used in our experiments, including the machine setup and software packages. We then go on to directly compare QC and Maronna, then look at the I/O performance and the performance for repairing the positive definiteness of the covariance matrices. Next, we present a dynamic load balanced version of Maronna along with other analysis of Maronna's convergence. Section 4.6 describes QC and Maronna's performance on datasets of different dimensions and looks at how the median and correlation parts of the computation perform based on the data's dimensions. Finally, a communication analysis is provided on both the message passing environment and for overall communication of our algorithms.

4.1 Setup

4.1.1 Parallel Cluster Environment

There are several different environments used for the experiments. The small tests used a cluster of eight machines, all 500 MHz Pentium 3 processors running Red Hat Linux 9. They are interconnected by 100 Mbit Ethernet and use LAM MPI 6.5.9 for message passing between them. The MPI (Message Passing Interface) is public domain software that runs on a variety of platforms. MPI provides a portable standardized interface for message passing applications on distributed memory computers [10].

Even though this setup is limited with only eight processors, it is still valuable because it is a dedicated setup. It is easier to obtain consistent times because our programs do not have to share the machines with other processes or share network services such as file servers.

The experiments measuring the communications analysis used a different machine setup. For these we used a small cluster of four Pentium 2 and Pentium 3 machines connected to a Juniper Networks router via dedicated Ethernet cables. This setup allows us to look at the communications going on between any of the machines using the values that the router records.

We used a new cluster of machines for our experiments involving large numbers of processors. This cluster is a part of the WestGrid computing project [35]. It consists of 504 dual processor 3 GHz Xeon processors running Red Hat 7.3 with 2 GB of RAM. They are connected on a Gigabit Ethernet network. These machines use MPICH [24].

4.1.2 Parallel Matrix Software

We use several software packages in implementing our algorithms. The first is a matrix library written by E. van den Berg [33] that has basic matrix functions, such as creating, destroying, reading, and has parallel matrix multiply. We altered the library to add our own I/O functions, and matrix distribution and collection operations. We did not use the parallel matrix multiplication functionality from this library in our implementations because other packages are available with better performance.

Our implementations use the PLAPACK Parallel Linear Algebra Package for matrix multiplications [32]. PLAPACK is a C based MPI parallel matrix library. It has a fast matrix multiply routine, several solvers, and other matrix and vector operations. The problem with PLAPACK is that there are some difficulties in using it. One difficulty is that it lacks some basic matrix or vector operations. For example, we need to perform an element wise multiply of two vectors, or scale the columns of a matrix by the corresponding elements of a vector, but we must implement them ourselves since they are missing from PLAPACK. Another difficulty with PLAPACK is dealing with the parallel matrix distribution. The matrices in PLAPACK use an unusual block distribution scheme that is not as easy to manipulate as a row based distribution or a simple block distribution. This makes it difficult to implement the missing matrix and vector operations. PLAPACK's eigensolver is a complete eigensolver, which was not needed in our case. It solves for all the eigenvalues and eigenvectors, but does not allow the user to choose only a subset of the values to find.

4.1.3 Parallel Eigensolvers

We had to use a different set of packages for partial eigensolving capabilities. PETSc [3] is the base matrix and vector library we use in the eigensolving component. It is mainly a parallel solver library for matrices

and vectors, written in C, and uses MPI. PETSc focuses more on parallel solvers and there are many solvers created as add-on packages. The only drawback is PETSc does not include a matrix multiplication routine, so we did not abandon PLAPACK.

The SLEPc package is an eigensolving package that is an add-on to PETSc [14]. It allows the user to perform partial eigensolving for eigenvalues within a specified part of the spectrum and their associated eigenvectors. SLEPc contains four built-in eigensolving methods, though they are rather limited in their implementation. SLEPc also acts as an interface to other eigensolving packages so you can use techniques that are more powerful.

Since the eigensolving routines in SLEPc are too limited for our use, we decided to use SLEPc as an interface to ARPACK [18]. ARPACK is a parallel partial eigensolver written in FORTRAN 77 that uses MPI. It uses the Implicitly Restarted Arnoldi Method to solve for eigenvalues. Even though SLEPc can use ARPACK as an add-on, it was still necessary to make some changes for MPI to work between the C in SLEPc and FORTRAN in ARPACK. We added some calls to conversion routines for MPI Communicators where SLEPc uses ARPACK, and the libraries worked fine.

4.1.4 Communication Analysis Software

Our analysis of communication uses two pieces of software. First, we use the MPBench benchmarking tool [25] for MPI that is included with the LLCBench tool in order to analyze the MPI communication primitives. MPBench performs repeated tests using increasing data sizes and provides graphical performance results. It measures bandwidth, latency, turnaround time, and performance of MPI collective operations such as broadcast and reduce.

The second piece of software we used for the communication analysis is a Perl script that makes SNMP calls to the router for statistics related to the number of bytes sent between the individual processors and the router. The script repeatedly queries the router for the number of bytes transmitted, and we use this data to create a profile of the amount of communication our software uses over time. The script is run simultaneously with our program and the queries are sent over a separate Ethernet cable to the router so that they do not interfere with the values we measure.

4.1.5 Experiment Parameters

Our experiments evaluate both the total time for our algorithms, and a component-wise analysis. For the components that are common between all versions of the algorithms, such as the I/O routines and repair of positive definiteness, the times are reported separately. Otherwise, the time for the experiments are broken down into their component parts. The dataset used is the gene dataset with 6068 variables and 20 trials, which produces a 6068 by 6068 correlation matrix. The experiments are run on different sizes of data, from 1000 to 6000, by reading in less variables from the gene data set. We also experiment with more variables and cases by generating a larger dataset from the gene dataset. We generate new variables by choosing a random number of variables, and using a set of random weights that sum to one, combine the variables and weights to form the new variable data. We use a similar strategy for generating more cases for the dataset. The number of processors also varies across the trials to demonstrate how the performance scales. All times are reported in seconds. We ran multiple trials and took the best times to report because the times can vary depending on activity on the processors or network. We report the best time of the runs to give results that are similar to what the results would be under ideal conditions.

4.2 Algorithm Performance Comparison

4.2.1 Small Cluster Performance

Timing results for QC and Maronna are presented in Table 4.1 and Table 4.2. The portion of the algorithm that restores the positive definiteness of the correlation matrix is not included in this comparison because, as we will see in section 4.4, the time for this dominates the total runtime.

Looking at the two tables, we can see that QC performs much faster than Maronna. The parallelization helps QC perform the correlation calculation quicker, but the total time is not affected as much. It appears that QC is reaching the limits of parallelization with just eight processors, but we will need to verify this on a larger cluster to be sure. Also, there appears to be superlinear speedup going from one to two processors. A possible explanation is that we are using the parallel algorithm for the single processor times, so the time for one processor includes some of the parallel overhead and inflates the time value. Another source of inefficiency in the times is related to how the PLAPACK library uses processors. PLAPACK treats the processors as a mesh formation, and in these experiments the library attempts to arrange the processors into a square mesh. Some numbers of processors do not easily arrange into a square mesh, and this could

affect the distribution of objects and the efficiency of the algorithm.

The column labelled "Other" represents the time for initializing variables, allocating memory, and deallocating memory. We can see that this time is very large with QC using a single processor for the 6000 variable experiment. This happened repeatedly, and we speculate that this size is where the problem begins to be too large for the memory of a single machine. The Maronna program also shows the memory constraints when the variable size is 6000, this time in the matrix fill portion, which takes the correlation values gathered from the processors and places them into the proper position of the matrix. Normally, this time would be constant, but here we see it increases as there are fewer processors. Fewer processors mean there is a larger demand for memory on the root processor since it will have to use more memory. Thus, when filling the entries into the matrix, it has to use more swap memory during the memory copy operations, which causes the time to be longer.

There are also unusual values for QC's gather time on the single and two processor experiments on 5000 and 6000 variables. The time seems large for a single processor, then decreases on two processors, and then continues to increase with more processors. The gather portion of QC uses a PLAPACK primitive call to assemble the distributed matrix into a continuous buffer on one processor, and this performance was noted in repeated experiments. This performance may also be due to the memory constraints of such a large problem size on our small cluster because it was not present on the large cluster experiments in Table 4.4.

Maronna takes much longer than QC in these trials, but it is showing good improvement from the parallelization in terms of both the correlation computation time and the total time.

4.2.2 Large Cluster Performance

We were able to run the parallel Maronna and QC on the WestGrid cluster using up to 128 processors on the gene data set. The results are in Table 4.3 for Maronna and Table 4.4 for QC. There was much variation in the experiments we ran because the cluster is a shared resource environment. This accounts for some of the time differences. Because of this we ran each experiment ten times, then chose the best value that was repeated in multiple experiments. We chose the best times to include instead of the average since this would best represent the closest we can achieve to performance in ideal conditions.

For the Maronna algorithm in Table 4.3, the correlation calculation time still appears to be decreasing as we add processors. The time decreases from 360 seconds on a single machine to 6 seconds with 128 processors. The total time is affected as well, but not to the same degree. It appears that using 128 processors

v	p	Input	Output	Scatter	Gather	Median	Cor	Other	Total
1000	1	0.005	1.924	0.001	0.096	0.033	0.911	0.139	3.109
	2	0.005	1.743	0.010	0.469	0.028	0.564	0.103	2.921
	4	0.006	1.818	0.016	0.681	0.022	0.353	0.088	2.984
	8	0.022	2.081	0.036	0.806	0.019	0.244	0.144	3.351
2000	1	0.008	7.029	0.003	0.369	0.066	3.810	0.526	11.809
	2	0.008	6.951	0.022	2.230	0.054	1.870	0.307	11.442
	4	0.009	6.878	0.022	3.123	0.042	1.146	0.215	11.436
	8	0.008	7.168	0.041	3.610	0.035	0.800	0.200	11.862
3000	1	0.011	15.724	0.004	0.815	0.099	8.570	1.146	26.369
	2	0.012	15.784	0.029	4.724	0.082	4.586	0.678	25.894
	4	0.012	15.713	0.032	6.547	0.066	2.830	0.403	25.603
	8	0.023	15.504	0.064	8.535	0.053	1.339	0.279	25.798
4000	1	0.015	29.303	0.006	1.594	0.132	19.098	2.220	52.367
	2	0.016	27.788	0.040	8.897	0.109	8.954	1.169	46.972
	4	0.015	27.763	0.042	12.008	0.089	4.002	0.634	44.553
	8	0.014	27.916	0.073	14.503	0.070	2.411	0.504	45.490
5000	1	0.030	47.974	0.008	15.630	0.164	27.748	4.454	96.007
	2	0.018	50.546	0.048	13.962	0.136	12.789	1.856	79.356
	4	0.019	45.490	0.050	18.732	0.108	7.107	1.005	72.511
	8	0.025	43.179	0.088	22.274	0.096	3.632	0.663	69.957
6000	1	0.088	76.259	0.009	24.625	0.363	44.503	21.005	166.851
	2	0.079	68.537	0.059	21.720	0.163	18.982	2.831	112.371
	4	0.074	64.937	0.058	27.059	0.129	9.503	1.536	103.296
	8	0.079	66.996	0.146	32.477	0.112	4.916	1.188	105.914

Table 4.1: QC Timings on Gene Dataset, where v is the number of variables, p is the number of processors, and cor is the correlation computation time

may be the highest for this dataset because there does not appear to be room for much improvement. The computation time may decrease by a couple of seconds if 256 processors were used, but that would probably be the point where the parallel overhead catches up since the computation time change going from 64 to 128 processors is only three seconds.

On the other hand, it appears that QC already reaches this point and is showing the affects of too much overhead as seen in Table 4.4. The total times are actually getting worse with more processors. The computation time itself is still decreasing up to the point where 64 processors are used, but the increase in gather time and other overhead is eliminating any improvements. The best number of processors seems to be in the four to sixteen processor range. It is difficult to tell because QC seems to be very susceptible to the variation that occurs from sharing the cluster with other jobs. Again, numbers of processors that do not form square meshes could also be a factor in this odd performance. Though the values appear to be anomalies,

v	p	Input	Output	Bcast	Gather	Median	Cor	Fill	Other	Total
1000	1	0.01	1.73	0.00	0.03	0.01	39.33	0.10	0.02	41.23
	2	0.01	1.83	0.01	0.23	0.02	20.92	0.10	0.03	23.14
	4	0.01	2.01	0.04	0.32	0.02	11.02	0.10	0.03	13.55
	8	0.11	2.11	0.04	0.38	0.02	5.71	0.10	0.06	8.52
2000	1	0.01	6.82	0.00	0.11	0.03	160.43	0.43	0.05	167.87
	2	0.01	7.04	0.03	0.90	0.04	85.94	0.43	0.05	94.44
	4	0.01	7.22	0.09	1.24	0.03	43.87	0.43	0.25	53.16
	8	0.01	7.37	0.09	1.43	0.03	22.94	0.43	0.07	32.37
3000	1	0.02	15.59	0.00	0.25	0.04	375.46	1.03	0.08	392.47
	2	0.02	15.92	0.04	2.01	0.05	200.96	1.07	0.08	220.15
	4	0.01	15.81	0.13	2.78	0.05	108.54	1.03	0.08	128.44
	8	0.05	15.97	0.13	3.17	0.05	55.58	1.03	0.11	76.09
4000	1	0.02	27.69	0.00	0.45	0.06	709.11	2.28	0.11	739.72
	2	0.02	27.91	0.06	3.56	0.07	392.05	2.22	0.12	426.01
	4	0.02	28.15	0.18	4.92	0.07	213.08	2.20	0.12	248.74
	8	0.02	28.01	0.19	5.62	0.06	112.39	2.19	0.15	148.63
5000	1	0.03	44.78	0.00	0.70	0.07	1186.50	3.52	0.22	1235.82
	2	0.03	44.81	0.08	5.56	0.09	678.23	3.39	0.21	732.38
	4	0.03	43.80	0.23	7.69	0.08	373.64	3.29	0.16	428.93
	8	0.03	43.64	0.23	8.78	0.09	203.76	3.39	0.19	260.12
6000	1	0.08	63.10	0.00	1.01	0.09	1766.95	40.69	0.42	1872.34
	2	0.09	63.20	0.09	8.02	0.11	1045.99	22.71	0.34	1140.55
	4	0.08	63.49	0.28	11.07	0.10	590.70	8.02	0.22	673.97
	8	0.06	62.97	0.28	12.59	0.10	318.84	7.60	0.28	402.72

Table 4.2: Maronna Timings on Gene Dataset, where v is the number of variables, p is the number of processors, cor is the correlation computation time, and fill is the matrix fill time

especially the high values for the gather time using four and eight processors, this activity appeared in the repeated experiments. Despite the odd numbers from QC, it is still clear that QC does not parallelize to as many processors as Maronna in this case.

Procs	Input	Output	Broadcast	Gather	Median	Cor Comp	Matrix Fill	Other	Total
1	0.017	12.590	0.000	0.280	0.018	359.642	2.034	0.104	374.684
2	0.009	13.136	0.007	0.557	0.012	220.361	2.025	0.104	236.212
4	0.016	9.327	0.024	1.056	0.012	118.945	2.038	0.104	131.521
8	0.009	9.508	0.034	1.394	0.012	58.208	2.037	0.104	71.306
16	0.010	3.787	0.065	1.463	0.010	29.677	2.069	0.104	37.185
32	0.010	5.057	0.057	1.839	0.011	18.972	2.464	0.280	28.689
64	0.011	4.863	0.303	1.941	0.252	9.087	2.444	0.283	19.185
128	0.009	4.241	0.301	1.920	0.020	6.182	2.474	0.386	15.533

Table 4.3: Maronna Timings on 6068 Variable Gene Dataset Using WestGrid

Procs	Input	Output	Scatter	Gather	Median	Cor Comp	Other	Total
1	0.009	6.260	0.002	0.920	0.067	18.524	1.197	26.977
2	0.011	5.382	0.004	1.648	0.037	9.430	0.969	17.480
4	0.009	5.366	0.005	10.058	0.024	4.157	0.558	20.176
8	0.010	5.405	0.012	14.383	0.020	1.513	0.528	21.870
16	0.009	4.171	0.019	7.038	0.018	1.455	1.501	14.212
32	0.008	4.510	0.044	7.530	0.057	1.082	2.945	16.177
64	0.018	5.536	0.145	15.489	0.291	0.331	9.414	31.225
128	0.008	3.903	0.485	47.815	0.266	0.497	27.226	80.201

Table 4.4: QC Timings on 6068 Variable Gene Dataset Using WestGrid

4.3 I/O Performance

Variables	Read Time	Write Time
1000	.0045	1.72
2000	.0075	7.0
3000	.01	15.5
4000	.015	27.7
5000	.018	43.6
6000	.074	63.2

Table 4.5: I/O Timings on Gene Dataset

All the implementations use the same I/O routines to read and write the data to disk. The I/O routines operate on data a row at a time and perform the read and write operations on binary data. The I/O operations are sequential, as the root processor performs the reading and broadcasts the data or gathers the result and performs a write. The performance of the I/O routines is listed in Table 4.5. The table shows that input time increases about linearly except for the experiments using 6000 variables. This was a repeatable occurrence, and is likely due to the input matrix being large enough to cause one more page miss. This did not occur in the experiments with the larger cluster, so we assume the input time is linear with the variable size. Though these experiments only deal with a small number of cases, the input time is linear in those as well. The output time scales quadratically with the problem size, as can be expected since the size of the correlation matrix is the square of the number of variables.

Variables	Processors	Time
1000	1	10.635
	2	7.625
	4	5.018
	8	7.485
2000	1	38.071
	2	24.422
	4	14.620
	8	15.194
3000	1	83.804
	2	50.105
	4	30.564
	8	23.565
4000	1	160.265
	2	89.006
	4	52.768
	8	43.595
5000	1	356.491
	2	148.677
	4	84.777
	8	63.362
6000	1	829.985
	2	276.929
	4	124.555
	8	83.900

Table 4.6: Time for Repairing Positive Definiteness on the Gene Dataset

4.4 Performance for Eigenvector Calculation

When the Maronna method or QC are used to calculate a covariance matrix, the resulting matrix may not be positive definite. We created a routine that uses a parallel eigensolver to solve for the negative eigenvalues, then creates a positive replacement for them and shifts the covariance matrix to repair the positive definiteness. Also, the routine can choose to solve for the positive eigenvalues instead, if there are fewer, and form the new covariance matrix from these. Again, this routine is identical for each algorithm. Table 4.6 shows the timings for repairing the positive definiteness.

We see that repairing the positive definiteness is no small matter because the times are nearly equivalent to QC's running time and are a significant portion of Maronna's running time. However, there does seem to be a benefit to parallelization here. For 3000 variables or more, the times show reasonable improvement with an increase in processors. With 1000 and 2000 variables, the algorithm seems to hit the point

where there is no more benefit to parallelization with eight processors. In these cases, the problem size is probably too small to where the overhead cost draws even with the benefits of adding processors. Also, the times for 5000 and 6000 variables seem to be too good. This could be due to several reasons. First, the problem size may be large enough to tax memory capacity, similar to what was seen with the Maronna and QC times for similar sizes. The sequential versions are just single processors running the parallel code, so this could penalize the single processor times and make them look extra slow. Finally, we used the PLAPACK matrix library in this code for the matrix operations, such as multiplication. As discussed earlier, PLAPACK looks to form the processors into a mesh formation, so certain numbers of processors, such as four, could experience better performance since the processors form a more square shaped mesh.

4.5 Load Balanced Maronna

4.5.1 Correlation Convergence

The motivation for implementing the load balanced Maronna came about from investigating the iteration counts for the algorithm. Initially we used random data sets that were not correlated, and Maronna converged very quickly. Each correlation calculation converged in roughly five iterations. However, Maronna reacts differently on real datasets, such as the gene data whose correlation iterations are shown in Table 4.8. While over ninety-nine percent of the calculations converged rapidly, others required more iterations. This behavior is bad for a static load-balancing scheme because some processors could receive a large number of the slow converging correlations to compute, and thus require more time to do their work. In the worst case, one processor could get all the bad correlations and end up being the bottleneck for the program while all the other processors wait for the unlucky processor to finish. In addition, a processor can receive one of the slow converging correlations as the last to compute of its correlations, which causes another bottleneck while all the processors wait for the unbalanced processor to finish calculating its last correlation.

One strategy to deal with the slow converging correlations is to identify them and spend less time by truncating the calculation. The absence of these correlations in the random data and their presence in the gene data led us to believe that calculating correlations for strongly positive or strongly negative correlations were the calculations that require the most time. We theorized that once we have performed enough iterations to identify a correlation to be within this group, its correlation could be truncated and then assigned the proper value. Unfortunately, closer analysis of the Maronna program showed that the slow

Iterations	Correlations	Percent of Total Corrs
0-2	11814349	64.18303%
3-5	5698891	30.95999%
6-8	741905	4.03050%
9-11	133822	0.72701%
12-14	14989	0.08143%
15-17	2983	0.01621%
18-20	317	0.00172%
21-23	21	0.00011%
24-26	1	0.00001%

Table 4.7: Correlation Convergence for Maronna on 6068 by 20 Gene Dataset with $\epsilon = .1$

Iterations	Correlations	Percent of Total Corrs
0-200	18396529	99.9416%
201-400	8665	0.04707%
401-600	1362	0.00740%
601-800	379	0.00206%
801-1000	127	0.00069%
1001-1200	78	0.00042%
1201-1400	53	0.00029%
1401-1600	28	0.00015%
1601-1800	22	0.00012%
1801-2000	9	0.00005%
2001-2200	7	0.00004%
2201-2400	4	0.00002%
$>$2400	19	0.00010%

Table 4.8: Correlation Convergence for Maronna on 6068 by 20 Gene Dataset with $\epsilon = 10^{-7}$

converging correlations might not be limited to only the strongly related correlations. Table 4.10 shows the slow converging correlations are from all parts of the spectrum, large, small, positive, and negative. Also, this distribution matched the distribution of all the correlation values, so we conclude that the correlation value itself does not determine whether convergence is fast or slow.

We discovered that the slow converging correlations are actually caused by outlier data by carefully examining how the Maronna algorithm calculates these correlations. Outlier values result in large distance values from the distance function. The Maronna algorithm then iterates and changes its parameters to decrease the distance values until convergence. Under normal circumstances, the distances are less than nine. The slow converging correlations had very large distance values, as seen in Table 4.11, which means

Iterations	Correlations	Percent of Total Corrs
0-200	18353376	99.70717%
201-400	42114	0.22879%
401-600	6981	0.03793%
601-800	2278	0.01238%
801-1000	993	0.00539%
1001-1200	532	0.00289%
1201-1400	316	0.00172%
1401-1600	180	0.00098%
1601-1800	123	0.00067%
1801-2000	92	0.00050%
2001-2200	61	0.00033%
2201-2400	46	0.00025%
>2400	185	0.00101%

Table 4.9: Correlation Convergence for Maronna on 6068 by 20 Gene Dataset with $\epsilon = 10^{-13}$

the Maronna algorithm had to iterate more times to decrease these distances.

Range	Number of Correlations
-1 to -.8	1
-.8 to -.6	8
-.6 to -.4	21
-.4 to -.2	30
-.2 to 0	45
0 to .2	36
.2 to .4	34
.4 to .6	26
.6 to .8	12
.8 to 1	0

Table 4.10: Range of Slow Converging Correlations on 6068 by 20 Gene Dataset with $\epsilon = 10^{-7}$

Iterations	Distances
2961	45.6, 33.3
5860	20.9, 21.5
3079	22.7, 19.5
3333	14.0
10332	11.9

Table 4.11: Slow Converging Correlations and Their Large Distance Values

4.5.2 Changing ϵ

Since a selective approach for dealing with the slow converging correlations is difficult, we thought to try a more global approach. The Maronna algorithm contains an epsilon argument as a stopping condition that determines how close the iterated correlation estimate is to the real correlation value. By increasing the value of epsilon, the correlation values, including the slow converging ones, will converge faster at the cost of accuracy. We can see how changing epsilon affects the convergence in Table 4.7 and Table 4.9. Table 4.9 shows the convergence using $\epsilon = 10^{-13}$. In comparison to Table 4.8, the iteration categories contain about ten times more correlations. Thus, when epsilon decreases, more correlations will take longer to converge. Table 4.7 shows the convergence using a very big epsilon, $\epsilon = .1$. As expected, the correlations all converge quickly in under 30 iterations and the majority converge within five iterations. In this case, decreasing epsilon results in faster convergence, but it is not yet clear what the effect is on accuracy.

We defined the accuracy to be the absolute difference between a correlation estimate and the correlations real value. We define the accuracy of a correlation matrix estimate to be the largest of the accuracy values for the matrix's individual correlation entries compared to the corresponding entries in the real correlation matrix. To calculate the accuracy in practice, we used the correlation matrix estimate with the smallest epsilon you can calculate, 10^{-13} for the gene data. The accuracies for various epsilon appear in Table 4.12. The largest epsilon that seems reasonable is 10^{-7}. The differences in time for the various epsilon show that we gain some improvement with a careful choice of epsilon. This improvement helps in both cases of slow converging correlations, whether a single slow converging correlation at the end of a processor's batch, or an unbalanced load to a processor since all the correlation calculations benefit with faster convergence for a smaller epsilon.

4.5.3 Dynamic Load Balanced Maronna

Another approach that avoids unbalanced processor loads is dynamic load balancing. For Maronna, we choose a block size and send each processor a starting block. Then, when a processor finishes with the first block, we send them another block of the same size. If one processor is stuck with several of the long converging correlations, then the load disperses to other processors that are not working as hard. This has an added benefit in that the user can run this implementation on a heterogeneous group of processors with greater success because the balancing will even out the differences of running on different speed processors.

Table 4.14 shows the block division that occurs during load balanced Maronna. The top half of the

ϵ	Largest Cor Quality	10th Largest Cor Quality	Time 2 Processors	Time 4 Processors	Time 10 Processors
10^{-1}	1.726	1.624	423.34	139.28	64.24
10^{-2}	1.77	1.433	524.88	182.88	86.82
10^{-3}	1.438	8.95E-01	577.23	202.65	95.72
10^{-4}	1.337	6.94E-01	618.59	221.01	105.22
10^{-5}	8.14E-01	4.39E-01	660.18	239.60	114.15
10^{-6}	4.22E-01	1.59E-01	714.74	255.23	122.87
10^{-7}	1.45E-01	1.71E-05	745.55	272.03	131.68
10^{-8}	2.68E-06	1.27E-06	782.27	288.77	140.21
10^{-9}	3.36E-07	1.51E-07	840.36	305.56	148.85
10^{-10}	3.36E-07	1.36E-08	864.89	322.26	157.53
10^{-11}	3.36E-07	1.23E-09	911.88	339.01	166.19
10^{-12}	2.41E-10	1.10E-10	947.70	357.77	174.85
10^{-13}	0	0	991.89	372.48	182.58

Table 4.12: Correlation Accuracy Compared to Iteration Time for Various Processor Sizes

table shows the division when identical processors are used with three worker processors and seven worker processors. The number of blocks each processor computes is nearly the same size for all, but it shows that some processors work through fewer blocks, which means the blocks they received contained correlations that took longer to converge. The bottom half of the table shows what happens when a heterogeneous group of processors is used. Processor seven is half as fast as the others and processor six is one and a half times faster. The table shows the slower processor does not process as many work blocks, but the quicker processor makes up for it and works through more blocks. If the regular Maronna version were used instead, the program would only proceed as fast as the slowest processor. The faster machines would simply wait until the slow one finished before continuing, which wastes computational resources and time. Thus, load balancing is beneficial to spread the work load evenly between the processors, and to improve performance in a heterogeneous environment.

We have created two versions of Maronna based on dynamic load balancing. The two load balanced versions are very similar and only differ in the way the result is gathered to the root processor. One version has the worker processors send their results to the root whenever they finish calculating the results for a block. The second version has the root gather all the results from the worker processors at the end of the entire calculation. The timings for the two load balanced Maronna algorithms are in Table 4.15 and Table 4.16. We can see that dynamic load balancing is a feasible approach to the problem of long converging correlations by comparing these to tables to the static Maronna in Table 4.2.

We see that the load balanced version with the ending gather in Table 4.15 is similar to the normal Maronna, but is a bit faster in the category of correlation calculation. Overall, the charts show this version is better than the original. The time on 5000 variables with 8 processors is 40 seconds faster and the time with 6000 variables and 8 processors is 100 seconds faster. This benefit is likely due to the positive results of load balancing.

Comparing both the load balanced versions to the regular Maronna in general, we see that the load balanced versions are not as constrained for memory as the regular Maronna. For example, there are no irregular values in the gather column for the ending gather version, and the matrix fill times seem more consistent. There is less of a memory constraint because the load balanced versions use a processor farm approach. The root processor's role is to keep feeding work to the other processors. Thus, it does not need to use memory for computation purposes as in the original version.

Even though the ending gather version of Maronna is better than the original, the block gather version is a bigger improvement. Table 4.16 shows that, compared to the original Maronna in Table 4.2, the block gather version takes only $60 - 70\%$ of the running time. This is better than the few seconds improvement of the ending gather version. Thus, the block gather version provides superior load balancing than the ending gather Maronna.

4.5.4 Load Balance Block Size

The block size plays a key role in dynamic load balancing. If it is too small, the communication overhead increases, while large sizes do not successfully balance the load. Performance analysis of load balancing with the gene data shows that a block size of around 50,000 works best. A detailed graph of load balanced Maronna times for different block sizes is show for 6000 variables on eight processors as an example in Figure 4.1. The figure shows that the best choice in block size is at the bottom of the curve somewhere between 6000 and 75000. Nothing smaller is a good choice because the running time increases rapidly for smaller block sizes since the root node becomes saturated with work requests. Sizes that are bigger than this range are bad because they are too close to the static division that the regular Maronna uses, and thus see little of the dynamic load balancing benefits.

The optimal block sizes for different variable and processor combinations are listed in Table 4.13. All of the combinations have optimal block sizes in the 6000 to 75000 range. It is interesting to note that in the experiments to discover the best block sizes for the different combinations, all of them had curves

that were rounded similarly to Figure 4.1. None had a sharp point where one single value had the best range. Instead, they all had a range near the optimal with similar performance. This range seems best to take advantage of the benefits of dynamic load balancing for Maronna.

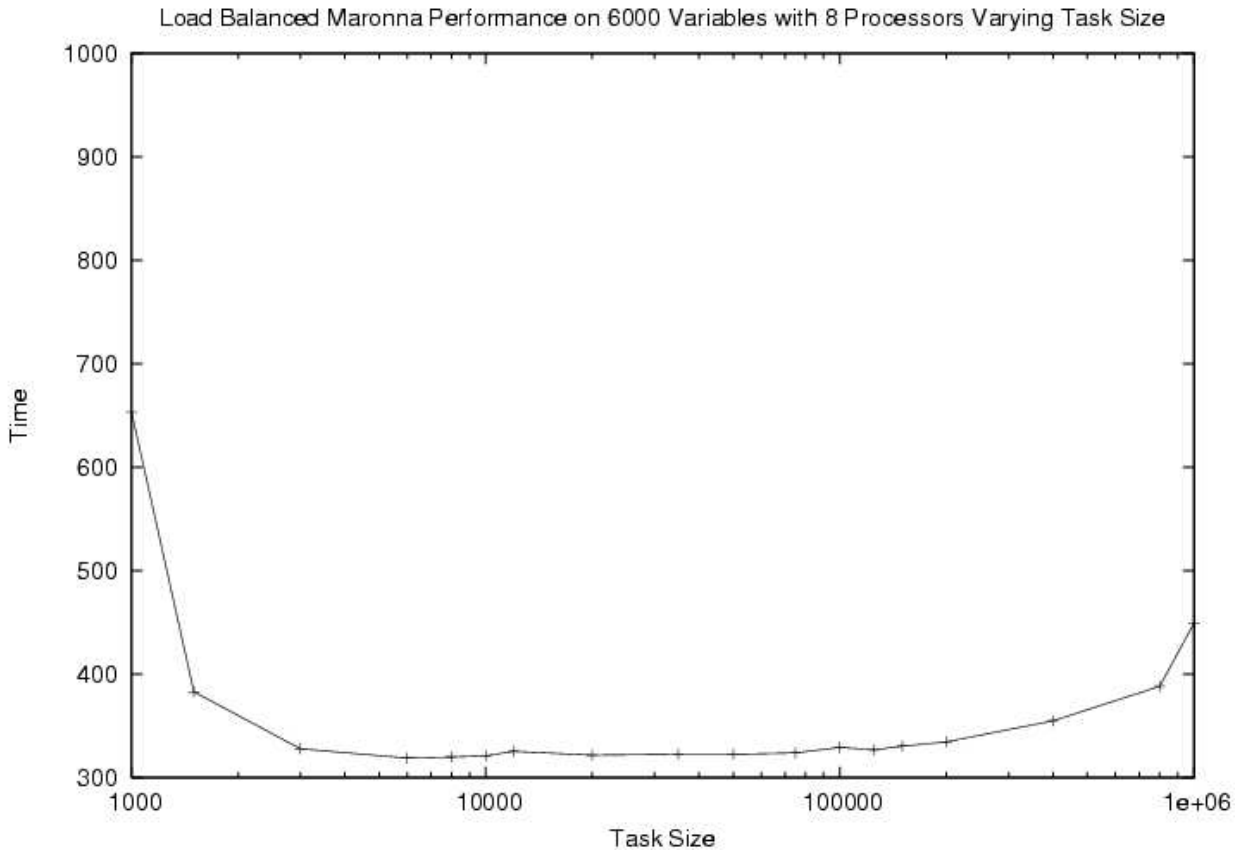

Figure 4.1: Load Balanced Maronna Various Task Sizes on 6000 Variables with 8 Processors

Variables	Processors	Block Size
2000	4	15000
4000		12000
6000		4000
2000	8	5000
4000		5000
6000		50000

Table 4.13: Load Balanced Maronna Optimal Block Sizes

Proc Group	Variables	Block Size	P1	P2	P3	P4	P5	P6	P7
Identical	2000	15000	45	44	45				
	4000	12000	222	225	220				
	6000	8000	745	751	754				
	2000	10000	28	30	29	29	27	28	29
	4000	10000	112	116	120	113	111	113	115
	6000	50000	52	51	52	51	53	50	51
Fast and Slow	2000	15000	27					92	15
	4000	12000	130					478	59
	6000	8000	423					1618	209
	2000	10000	23	22	22	21	23	78	11
	4000	10000	90	84	93	86	84	321	42
	6000	50000	39	38	39	40	39	145	20

Table 4.14: Block Division Between Processors

v	p	Input	Output	Bcast	Gather	Median	Cor	Fill	Other	Total
1000	1	0.010	1.828	0.012	0.355	0.020	38.844	0.096	0.021	41.186
	2	0.009	1.809	0.020	0.351	0.019	19.658	0.098	0.020	21.984
	4	0.008	1.711	0.031	0.327	0.018	9.885	0.102	0.020	12.103
	8	0.091	2.207	0.044	0.335	0.017	4.989	0.103	0.020	7.807
2000	1	0.013	6.973	0.026	1.382	0.035	158.452	0.433	0.040	167.354
	2	0.013	7.035	0.059	1.295	0.036	79.472	0.427	0.040	88.377
	4	0.013	6.795	0.086	1.178	0.033	40.035	0.437	0.041	48.617
	8	0.012	6.919	0.117	1.010	0.031	19.928	0.440	0.041	28.498
3000	1	0.016	15.590	0.045	3.706	0.052	375.065	1.044	0.060	395.577
	2	0.016	15.564	0.093	3.104	0.055	186.543	1.036	0.060	206.471
	4	0.017	15.481	0.134	3.100	0.048	93.532	1.069	0.061	113.442
	8	0.052	15.751	0.191	3.103	0.046	47.261	1.033	0.061	67.498
4000	1	0.020	27.811	0.061	6.441	0.070	706.150	2.327	0.081	742.960
	2	0.018	27.591	0.124	5.888	0.071	353.164	2.201	0.081	389.138
	4	0.018	27.555	0.182	5.516	0.066	180.484	2.225	0.081	216.127
	8	0.022	28.041	0.252	5.505	0.061	88.534	3.015	0.081	125.509
5000	1	0.031	44.472	0.076	9.988	0.088	1189.482	3.376	0.139	1247.651
	2	0.030	44.285	0.155	9.907	0.085	599.196	3.394	0.143	657.195
	4	0.036	44.011	0.233	8.630	0.084	296.905	3.374	0.164	353.437
	8	0.035	43.412	0.321	8.575	0.077	147.789	4.294	0.101	204.604
6000	1	0.040	62.324	0.092	14.394	0.105	1828.638	7.978	0.122	1913.693
	2	0.040	63.153	0.191	13.013	0.100	891.910	7.696	0.122	976.224
	4	0.040	63.028	0.286	12.524	0.103	450.486	7.859	0.122	534.449
	8	0.035	63.714	0.386	12.374	0.108	222.788	9.245	0.221	308.871

Table 4.15: Load Balanced Maronna with End Gather, where v is the number of variables, p is the number of processors, cor is the correlation computation time, and fill is the matrix fill time

v	p	Input	Output	Bcast	Median	Cor	Fill	Other	Total
1000	1	0.007	1.795	0.012	0.021	19.001	0.097	0.020	20.954
	2	0.009	1.925	0.024	0.019	9.693	0.096	0.020	11.786
	4	0.007	1.994	0.040	0.018	4.901	0.096	0.020	7.076
	8	0.061	2.017	0.056	0.017	2.498	0.097	0.021	4.766
2000	1	0.012	7.160	0.026	0.036	83.175	0.434	0.040	90.883
	2	0.011	6.841	0.059	0.039	41.753	0.437	0.040	49.181
	4	0.011	6.910	0.086	0.033	21.260	0.439	0.041	28.780
	8	0.013	7.033	0.125	0.031	10.671	0.455	0.041	18.368
3000	1	0.016	15.820	0.042	0.057	211.414	1.044	0.060	228.453
	2	0.015	15.680	0.092	0.055	106.303	1.044	0.060	123.249
	4	0.019	15.481	0.135	0.048	54.855	1.053	0.061	71.652
	8	0.049	15.503	0.190	0.046	26.882	1.066	0.061	43.796
4000	1	0.018	28.031	0.061	0.070	416.773	2.193	0.081	447.227
	2	0.018	28.190	0.126	0.070	210.005	2.187	0.081	240.677
	4	0.018	27.935	0.182	0.065	108.365	2.211	0.081	138.857
	8	0.020	27.783	0.254	0.061	52.997	3.192	0.081	84.390
5000	1	0.031	44.119	0.076	0.088	724.090	3.360	0.157	771.921
	2	0.031	44.914	0.158	0.088	363.619	3.347	0.151	412.309
	4	0.035	44.306	0.233	0.083	183.444	3.340	0.140	231.582
	8	0.035	42.825	0.321	0.076	92.819	4.291	0.101	140.469
6000	1	0.040	63.159	0.092	0.105	1156.037	7.836	0.121	1227.390
	2	0.043	62.946	0.188	0.103	580.643	7.990	0.122	652.034
	4	0.040	62.803	0.287	0.103	295.477	7.587	0.121	366.418
	8	0.034	63.934	0.385	0.105	147.254	10.035	0.122	221.868

Table 4.16: Load Balanced Maronna with Block Gather, where v is the number of variables, p is the number of processors, cor is the correlation computation time, and fill is the matrix fill time

4.6 Varying Data Shape

The QC and Maronna algorithms performance is partially dependent on the shape of the dataset, i.e. the number of variables and cases. It is clear that the algorithms depend on the number of variables in the dataset because this determines the size of the output correlation/covariance matrix. The number of cases plays a smaller role. The runtime of the median and MAD portion of the algorithm depends on the number of cases in the input dataset. If the number of cases or the number of variables is small enough, it may be more appropriate to perform a sequential operation in place of the parallel because the parallel version may have too high of an overhead. Also, it may be better to run the sequential median algorithm when the number of processors is large and creates enough overhead so that the parallel is slower than the sequential. We separated the two algorithms each into two parts, the median-MAD calculation and the rest of the

correlation/covariance computation.

With these two components, we have four separate combinations if we consider the parallel and sequential versions of each. One would combine the sequential median and the sequential correlation algorithms for data sets that have few variables and few cases. We see in the trends in Table 4.18 that the parallel median does provide improvement when there are a few cases and many variables. However, with more processors the overhead may be too great so that there is little gain in using the parallel median. Thus, if there are many variables and few cases, it is good to use the parallel median with a small number of processors, but when there are many processors, it is faster to use the sequential version. If the dataset has few variables but many cases, we would still use the parallel correlation algorithm because even with a few variables, the number of correlations is a quadratic function of the variables. Plus, the number of cases affects the correlation runtime, as we see in Table 4.17. Therefore, there is always some time savings possible using a parallel correlation algorithm unless the data set has few variables and cases. The most obvious choice is when the data set has many variables and cases, where parallel algorithms should be used for both the median and correlation calculations.

For our experiments varying the number of variables and cases, we use the gene dataset, however, we generate more variables or cases at random for the dataset to evaluate the algorithms on different data sizes. We construct a new variable or row in the dataset from a random group of existing variables or cases. To create each element in the new variable/case, we assign random weights to the corresponding elements in the set of existing variables/cases, where the weights sum to one, then sum up the products of the weights and elements.

The results of our experiments involving the variation of cases and variables on our correlation algorithms are listed in Table 4.17 and 4.18. For Table 4.17, we hold the number of variables constant at 6000 and vary the number of cases in the data set. QC seems to handle increasingly larger numbers of cases well, and has a moderate time increase as the number of cases goes from twenty to one hundred to one thousand. However, our runs with ten thousand cases show a significant increase in time. It is difficult to tell how accurate these times are because this is well into the point where memory constraints kick in. In fact, the runs for one and two processors were unable to complete due to lack of memory.

The Maronna algorithm does not fair as well as QC as the number of cases increase. In fact, the increase in running time seems to increase linearly with the increase in cases. In the bottom half of Table 4.17, as the cases increase from twenty to one hundred, the running times increase by about a factor of five, and

when the cases increase from one hundred to a thousand, the running times increase by about a factor of ten. this makes sense because every iteration of the Maronna algorithm requires a sum across the current values of the cases for the variables involved in the current correlation. Also, the runs with one thousand cases are the point where Maronna runs into memory problems as the single processor experiment ran out of memory before completion.

We also experimented with increasing the number of variables in Table 4.18. In these experiments, we held the number of cases constant at twenty and increased the number of variables. Unfortunately, memory constraints kept us from trying anything beyond ten thousand variables. The running times clearly do not increase linearly with the number of variables. From 2000 variables to 4000, the total time increases by a factor of four, from 4000 to 6000 by a factor of two, from 6000 to 8000 by a factor of almost ten, and from 8000 to 10000 by a factor of two. The correlation computation time also increases at a nonlinear rate, going from a factor of three to two to one and a half over the changes in variables. The changing increase in times could be caused by some parts of the algorithm being related linearly to the number of variables, while others are reacting quadratically to the number of variables, such as the output time. Again, we can see where the algorithm is running into memory problems on the single processor 8000 variable run and also in the fact that the single and two processor runs with 10000 variables did not finish and are absent from the table.

The bottom half of Table 4.18 shows the effects on Maronna when the number of variables increases. The total time increases by a factor of five from 2000 to 4000 variables, but from that point, it is increasing by about a factor of 2.7 for the larger numbers of variables. The correlation computation time is not that steady, and increases by a factor of 5 from 2000 to 4000 variables, by 2.88 from 4000 to 6000, by 2.38 from 6000 to 8000, and by 1.87 from 8000 to 10000. The times seem to be increasing at a decreasing rate, and less so than QC. Maronna also ran into memory problems with 10000 variables, as the single processor experiment ran out of memory.

In summary, both algorithms are affected by the number of variables and cases in the input data set. Both react quadratically to the number of variables, QC seemingly more so than Maronna. Maronna reacts linearly to changes in the number of cases, and QC does not react as much, though it may for very large numbers of cases, such as 10000.

Algorithm	Cases	Procs	Median	Cor Comp	Total
QC	20	1	0.380	45.551	167.486
		2	0.162	19.936	113.477
		4	0.130	9.988	108.931
		8	0.110	5.159	102.084
	100	1	2.056	64.587	184.321
		2	0.949	29.130	123.999
		4	0.706	15.591	109.909
		8	0.579	9.024	108.964
	1000	1	26.197	293.737	667.494
		2	12.504	147.608	247.294
		4	8.558	89.336	193.375
		8	6.641	57.945	169.754
	10000	4	357.696	8411.704	9295.692
		8	176.208	6301.003	6857.489
Maronna	20	1	0.087	1768.519	1872.328
		2	0.105	1045.453	1129.290
		4	0.103	589.543	673.169
		8	0.101	318.835	400.893
	100	1	0.299	9730.242	9819.595
		2	0.428	5006.885	5105.313
		4	0.452	2595.059	2680.493
		8	0.459	1315.559	1401.823
	1000	2	4.025	50057.706	50186.467
		4	4.364	25596.003	25709.269
		8	4.551	13191.708	13315.304

Table 4.17: Algorithm Performance Varying Cases Using 6000 Variables

4.7 Communication Analysis

4.7.1 MPI Performance

The MPBench tool measures the performance of MPI primitives. We use the tool to examine some of the costs of using MPI. Figures 4.2-4.6 show the graphical results for the MPI primitives relevant to our program. Figures 4.2 and 4.3 show the unidirectional and bidirectional bandwidth for MPI. The unidirectional bandwidth is implemented using the normal send and receives while the bidirectional uses nonblocking sends and receives. Figures 4.4 and 4.5 give the roundtrip time and latency for the send function. Finally, Figure 4.6 shows the performance of the broadcast primitive.

One of the things evident in the figures is that sending small messages requires only a constant amount of time. If a message is small, then MPI just transfers the message into the buffer of the receiving

Algorithm	Variables	Procs	Median	Cor Comp	Total
QC	2000	1	0.066	3.915	11.906
		2	0.055	1.967	11.435
		4	0.042	1.232	11.632
		8	0.037	0.885	11.475
	4000	1	0.132	19.419	51.747
		2	0.108	9.286	47.286
		4	0.089	4.170	44.638
		8	0.069	2.571	45.768
	6000	1	0.435	45.293	164.516
		2	0.163	19.224	113.248
		4	0.130	10.037	103.713
		8	0.114	5.201	105.826
	8000	1	0.436	436.336	2192.327
		2	0.331	40.027	1065.459
		4	0.180	20.564	998.024
		8	0.165	9.697	857.481
	10000	4	0.224	30.731	1913.898
		8	0.197	15.111	1812.029
Maronna	2000	1	0.029	161.155	168.681
		2	0.038	86.056	94.665
		4	0.034	43.849	52.665
		8	0.032	22.889	32.095
	4000	1	0.058	708.679	739.346
		2	0.071	392.950	427.056
		4	0.069	213.169	249.073
		8	0.064	112.564	149.675
	6000	1	0.086	1772.464	1874.283
		2	0.105	1050.567	1149.855
		4	0.104	593.846	676.139
		8	0.104	323.927	409.366
	8000	1	0.115	3761.701	4091.564
		2	0.140	2345.051	2692.082
		4	0.135	1400.404	1733.449
		8	0.142	769.024	1102.883
	10000	2	0.173	4350.110	6905.358
		4	0.166	2599.129	4636.957
		8	0.173	1440.370	3009.094

Table 4.18: Algorithm Performance Varying Variables Using 20 Cases

process. When messages are large, the sender and receiver have to agree on the memory location where the message will be stored on the receiver's end, and then transfer the message. This introduces a per-byte cost, but this is more efficient than buffering large messages because the extra buffering would require more time

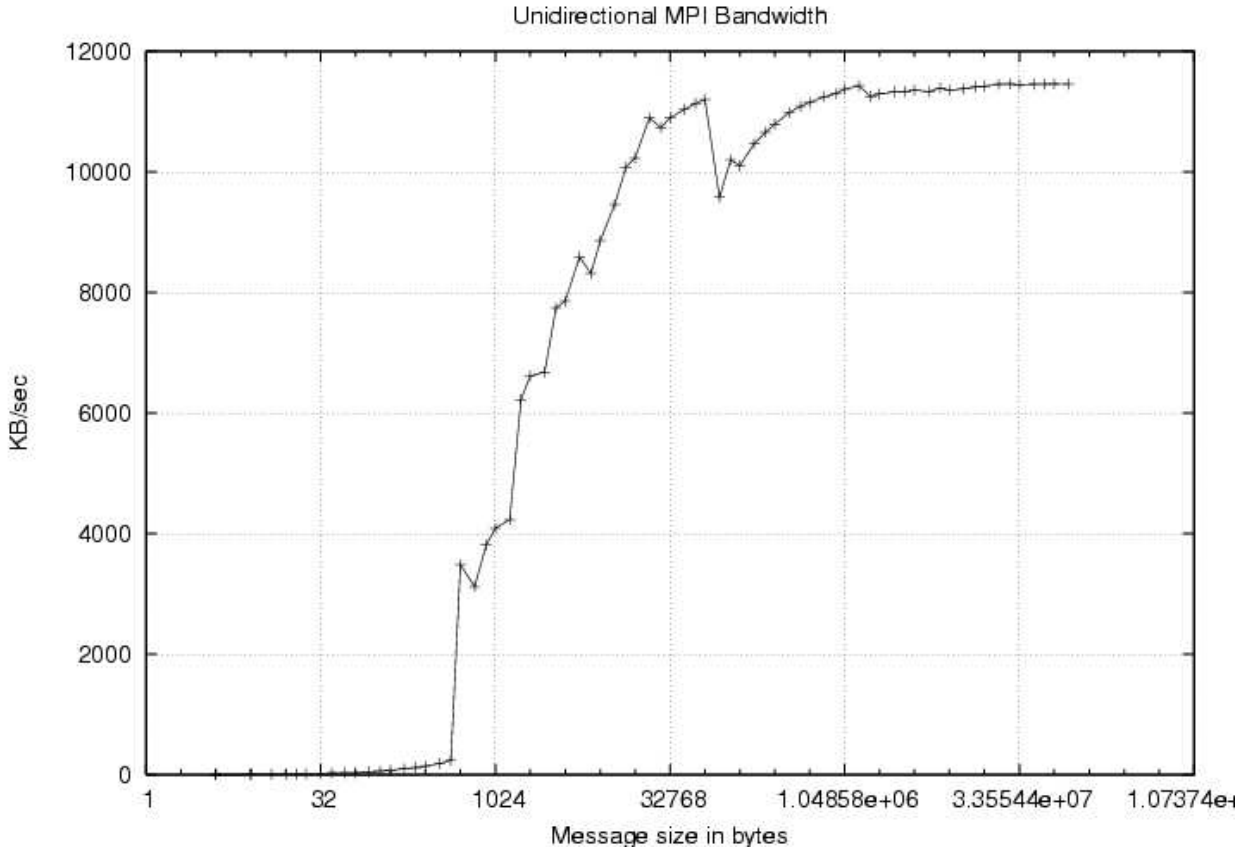

Figure 4.2: Unidirectional MPI Bandwidth

for buffer copying. We can estimate the costs using the performance graphs. The constant startup time for a send is roughly 16 microseconds, while the per byte cost for larger messages is about .0856 microseconds per byte.

4.7.2 Algorithm Communication

We were able to create rough communications profiles for the algorithms using a setup with several machines connected to a router via dedicated Ethernet cables. Then, we repeatedly queried the router for the traffic through the connections to monitor the activity as time passed. The communications traffic is measured in bytes over the queries we made. We could query the router at approximately 37 times per second. We grouped the results of 1000 of these queries together and report the sum of the traffic that the router reported for this period. The x-axis in the graphs represents a rough estimation of time, where we report total router traffic about every twenty-seven second interval. The reported traffic is measured in kilobytes.

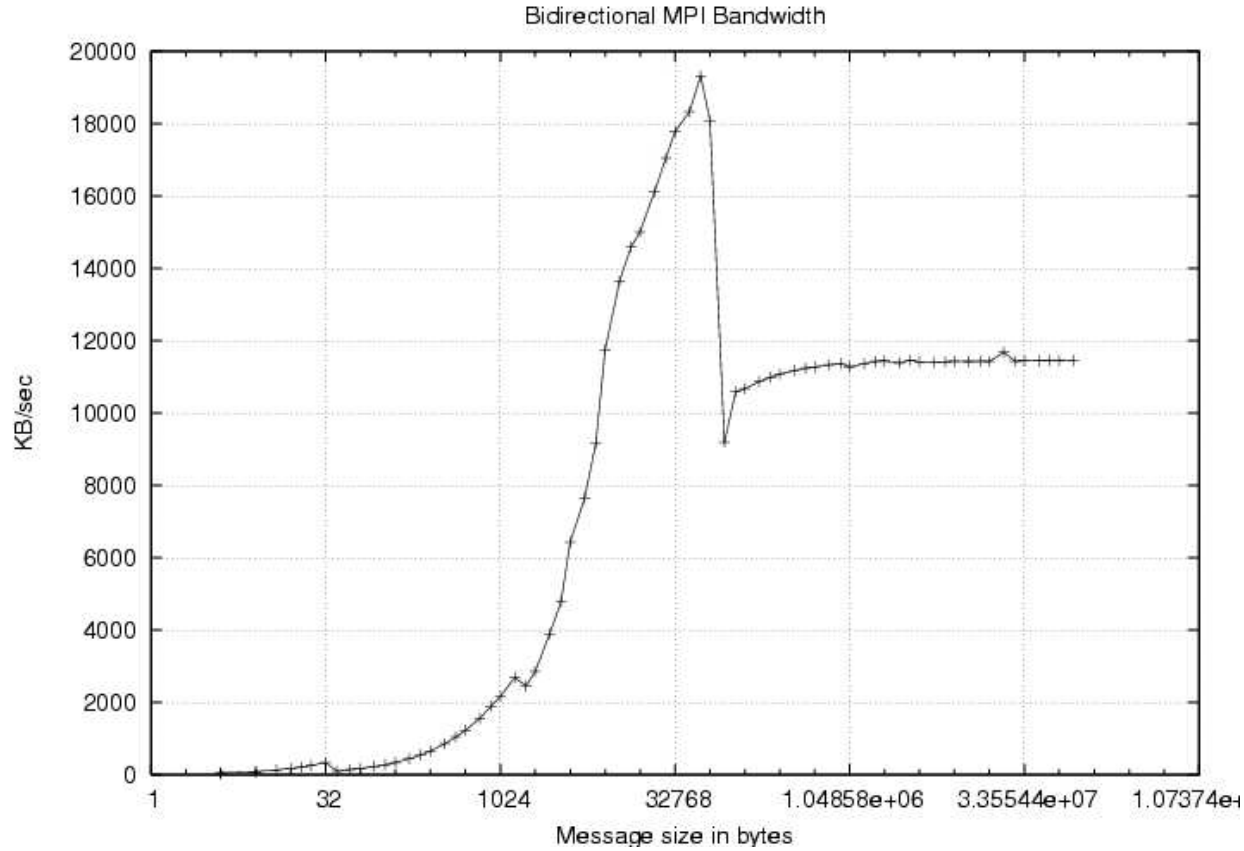

Figure 4.3: Bidirectional MPI Bandwidth

Looking at QC's graph, Figure 4.7, we see that the largest amount of traffic occurs during the result gathering stage. Figure 4.8 shows the Maronna algorithm's profile. Maronna also has the heaviest traffic during the gather stage. There are two large communication points on the graph because one of our machines was slower than the rest. When the faster machines completed earlier, they had to wait for the slowest one to finish and meanwhile sat idly. This is one reason why load balancing is an improvement.

We have profiled two versions of the load balanced Maronna. The first has the processors returning their results after they calculate a block of correlations, and is shown in Figure 4.9. The height of all the communications here and their thickness in the graph are all related to the block size of the algorithm. Small blocks make the messages smaller and more frequent, while larger blocks make for larger messages that are not as frequent. Thus, the total traffic the network can handle is something to consider when choosing the block size.

The profile for the second load balanced Maronna is in Figure 4.10. It is similar to the original Maronna in that all the correlations are saved up until the end for one massive gather. The processors still

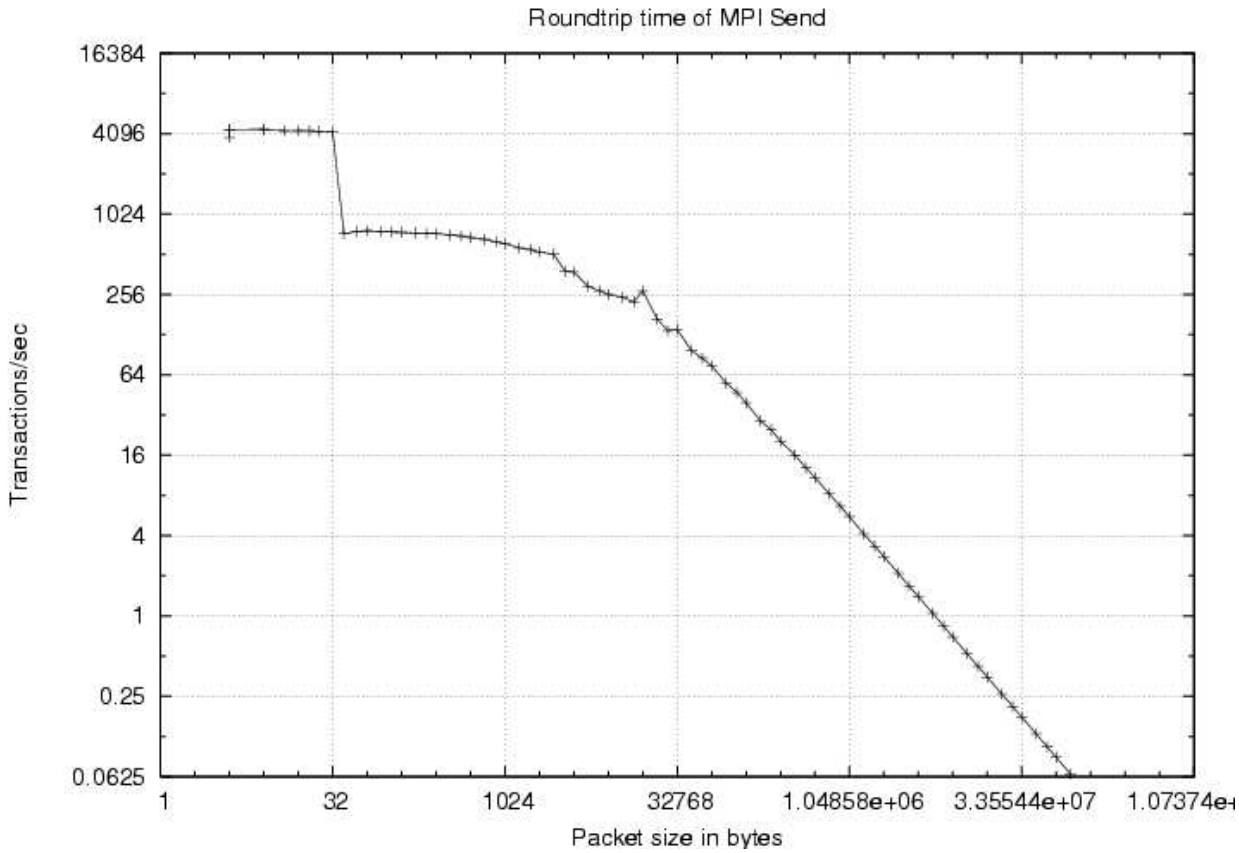

Figure 4.4: Roundtrip Time of MPI Send

send messages throughout the algorithm, but they are small and only serve as requests for more work.

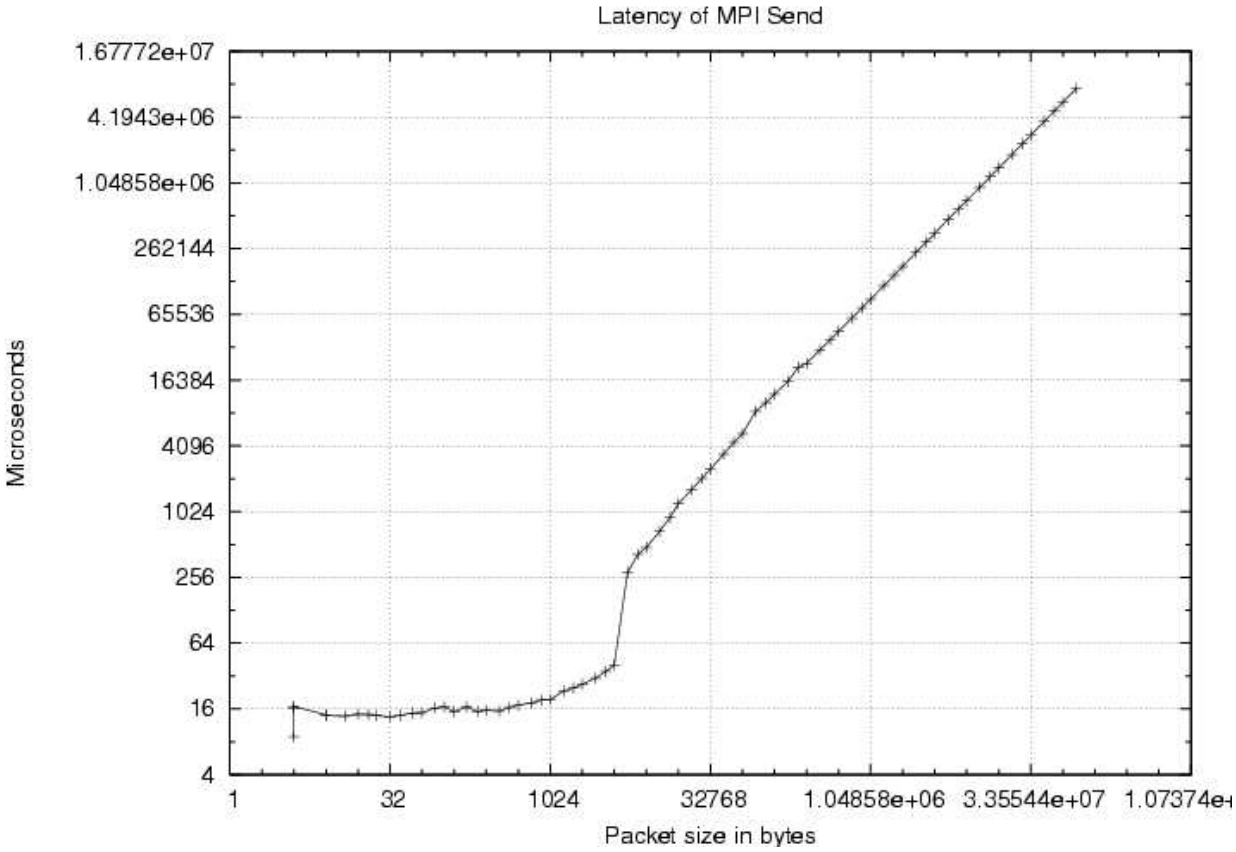

Figure 4.5: Latency of MPI Send

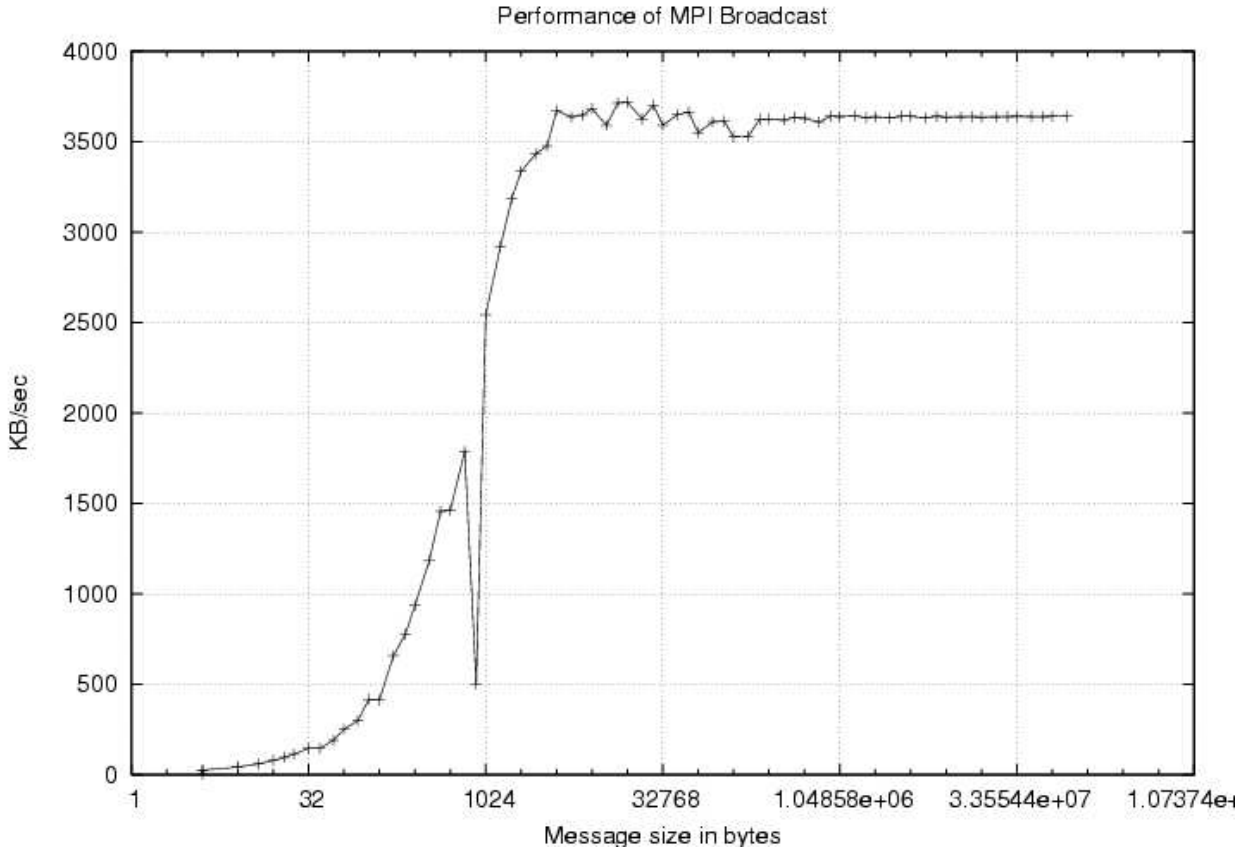

Figure 4.6: Performance of MPI Broadcast

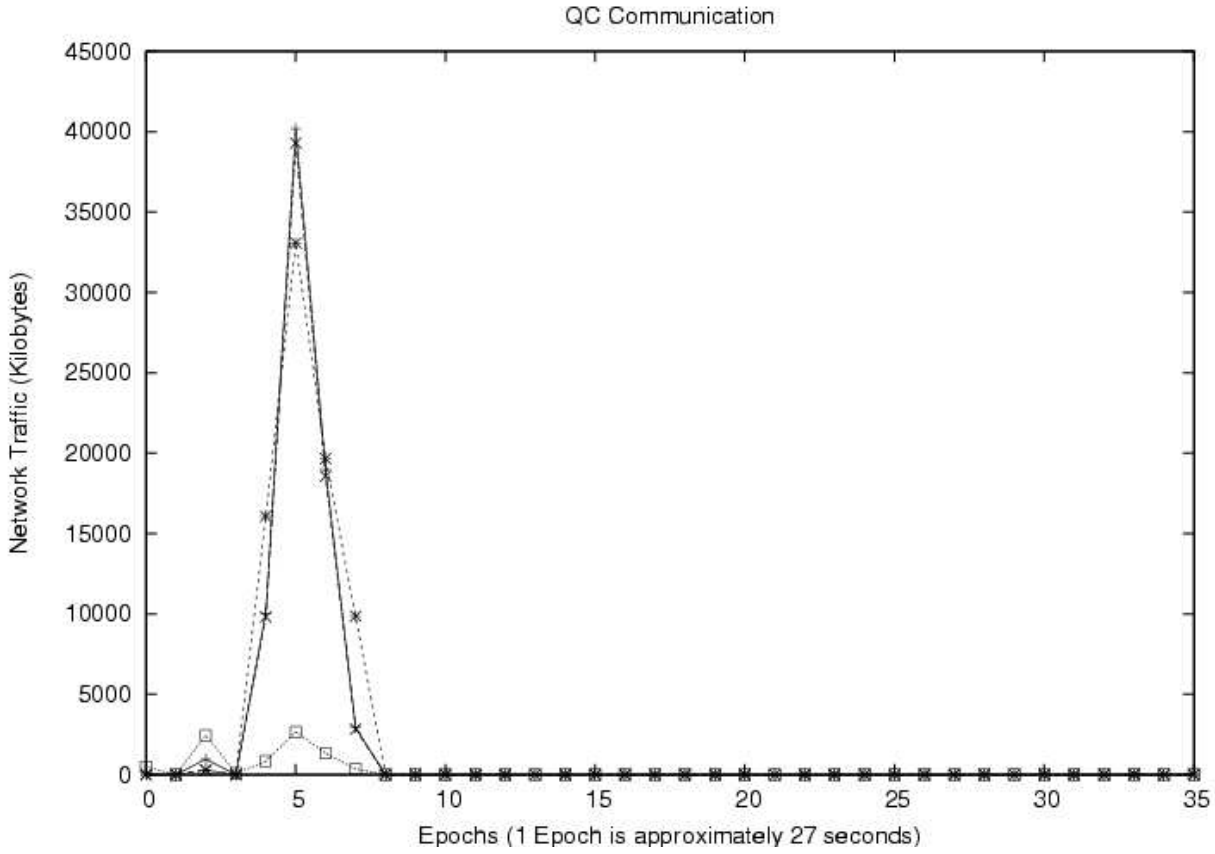

Figure 4.7: QC Communications Profile

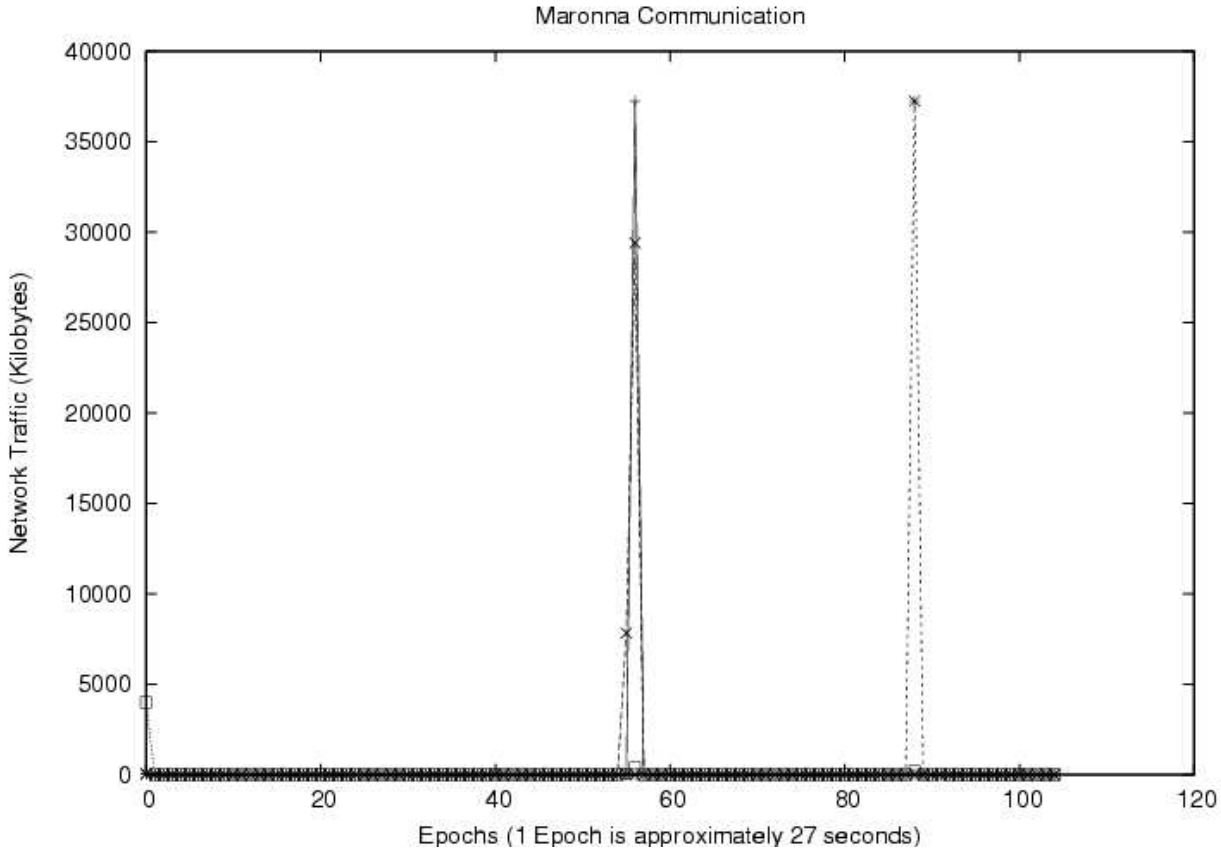

Figure 4.8: Maronna Communications Profile

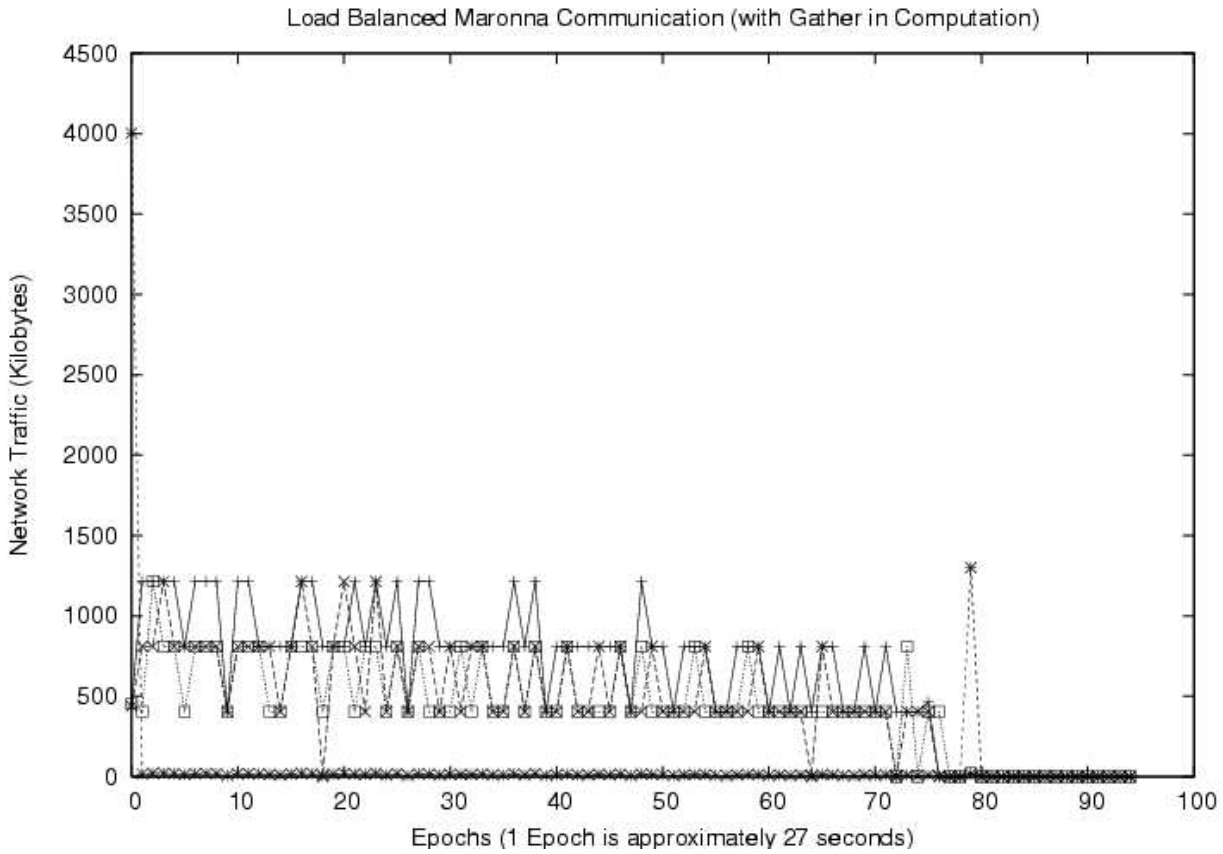

Figure 4.9: Load Balanced Maronna with Block Gather Communications Profile

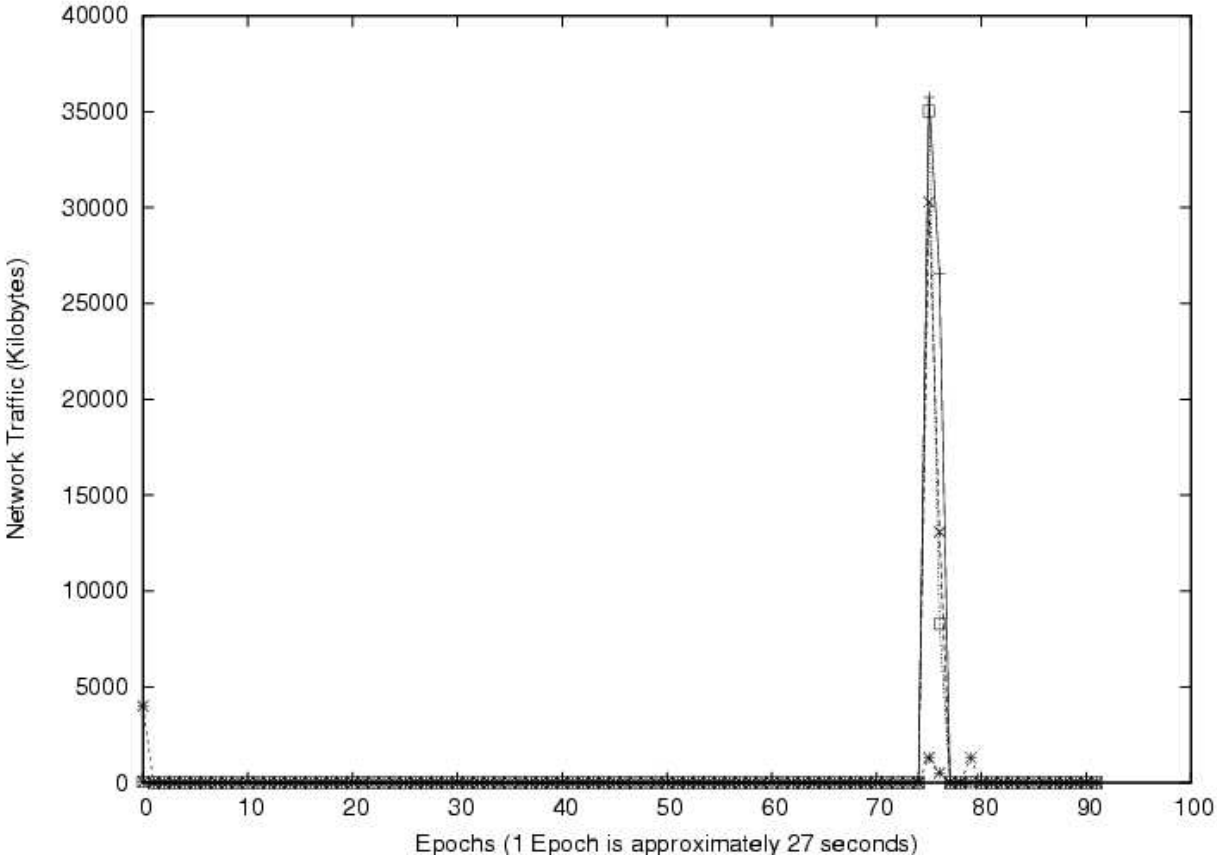

Figure 4.10: Load Balanced Maronna with End Gather Communications Profile

Chapter 5

Conclusions and Future Work

5.1 Conclusion

This thesis has shown that robust methods for calculating correlation and covariance matrices are feasible when implemented in parallel. These methods now make it possible to not only solve for large correlation and covariance matrices in a timely fashion, but also compute them with a more robust approach.

Our experiments were performed on a relevant dataset with 6068 variables representing different genes in 20 cases. The results show that both QC and Maronna scale well for up to 8 processors. Maronna seems to scale exceptionally well since the computation portion requires no communication between processors. This helps Maronna to achieve speedup on more than 8 processors, up to 128 as can be seen from the WestGrid results. QC is still faster, but Maronna is more robust and scalable to more processors. The two algorithms are good for solving different types of problems. If an application requires speed, has few processors available, and is willing to sacrifice some robustness in its results, then QC is the algorithm of choice. On the other hand, if many processors are available and greater robustness is required, then Maronna is a good choice if one is willing to put up with a longer wait. Both algorithms have their advantages and disadvantages, and the best method depends on the circumstances of the problem being solved.

We examined Maronna closely and found that some correlations, namely the ones involving outlier data values, converge at a slower rate. In response, we developed a load balanced version of Maronna and also experimented with several values for accuracy to improve the run time.

When we varied the size of the dataset in both cases and variables, we found that both algorithms scale linearly when the number of cases increase, and scales quadratically when the number of variables increase. We explain that there is a dividing line where it is more advantageous to run sequential versions of

the median and correlation components when the overhead of the parallel algorithms was too much.

Finally, our communication analysis on QC and Maronna give us an idea of which parts of the algorithms have the greatest communication cost and how the algorithms compare in terms of communication.

5.2 Future Work

With the success from parallelizing these techniques, the next step is to wonder whether other methods of robust calculation for correlation and covariance matrices would see similar results. Other algorithms include a version of Maronna that considers three variables at a time instead of just two and also the Stahel-Donoho method. Another area to improve these algorithms is the I/O time or repairing the positive-definiteness since parallelization has decreased the computation time of the main algorithm to a small portion of total time. Also of interest would be a hybrid method that initially calculates the correlation pairwise, but could detect when better outlier detection is needed for certain correlations and run a triplet correlation method for those values. Much benefit would come from parallel I/O routines or a more efficient eigensolving routine that is specialized for our purpose of fixing the positive definiteness.

Bibliography

[1] M. B. Abdullah. On a robust correlation coefficient. *The Statistician*, 39:455–460, 1990.

[2] F. A. Alqallaf, K. P. Konis, and R. D. Martin. Scalable robust covariance and correlation estimates for data mining. In *Proceedings of the Seventh ACM SIGKDD*, pages 455–460, 1990.

[3] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc home page. http://www.mcs.anl.gov/petsc, 2001.

[4] L. Boxer. Expected optimal selection on the PRAM. Technical Report 2002-17, Department of Computer Science and Engineering, University at Buffalo, 2002.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1990.

[6] S. J. Devlin, R. Gnanadesikan, and J. R. Kettenring. Robust estimation of dispersion matrices and principal components. *Journal of the American Statistical Association*, 76:354–362, 1981.

[7] D. Donoho. *Breakdown properties of multivariate location estimators*. PhD thesis, Harvard University, 1982.

[8] R. O. Dror. Noise models in gene array analysis, June 2001. Area exam report, MIT Department of Engineering and Computer Science.

[9] L. V. Fausett. *Applied Numerical Analysis Using Matlab*. Prentice Hall, 1999.

[10] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, Department of Computer Science, University of Tennessee, 1994.

[11] R. Gnanadesikan and J. R. Kettenring. Robust estimates, residuals, and outlier detection with multiresponse data. *Biometrics*, 28:81–124, 1972.

[12] A. Gupta and V. Kumar. Scalability of parallel algorithms for matrix multiplication. In *Proceedings of the 1993 International Conference on Parallel Processing*, volume III - Algorithms & Applications, pages III–115–III–123, Boca Raton, FL, 1993. CRC Press.

[13] Y. Han. Optimal parallel selection. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1–9. Society for Industrial and Applied Mathematics, 2003.

[14] V. Hernández, J. E. Román, and V. Vidal. SLEPc: Scalable Library for Eigenvalue Problem Computations. *Lecture Notes in Computer Science*, 2565:377–391, 2003.

[15] P. J. Huber. *Robust Statistics*. John Wiley & Sons, 1981.

[16] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[17] E. M. Knorr, R. T. Ng, and R. H. Zamar. Robust space transformations for distance-based operations. In *Knowledge Discovery and Data Mining*, pages 126–135, 2001.

[18] R. Lehoucq, D. Sorensen, and C. Yang. Arpack users' guide: Solution of large scale eigenvalue problems with implicitly restarted Arnoldi methods, 1997.

[19] R. Maronna and R. Zamar. Robust estimates of location and dispersion for high dimensional data sets. *Technometrics*, 2002. to appear.

[20] R. A. Maronna. Robust M-estimators of multivariate location and scatter. *The Annals of Statistics*, 4(1):51–67, 1976.

[21] R. A. Maronna, W. A. Stahel, and V. Yohai. Bias-robust estimation of multivariate scatter based on projections. *Journal of Multivariate Analysis*, 42:141–161, 1992.

[22] R. A. Maronna and V. Yohai. The behaviour of the Stahel-Donoho robust multivariate estimator. *Journal of the American Statistical Association*, 90(429):330–341, 1995.

[23] H. Meuer, E. Strohmajer, J. Dongarra, and H. Simon. Top 500 supercomputers, November 2003.

[24] MPICH - a portable implementation of MPI. www-unix.mcs.anl.gov/mpi/mpich/.

[25] P. Mucci and K. London. The MPBench report, March 1998. Available at www.cs.utk.edu/mucci/DOD/mpbench.ps.

[26] P. Rousseeuw and V. Driessen. A fast algorithm for the minimum covariance determinant estimator. *Technometrics*, 41:212–223, 1999.

[27] P. Rousseeuw and A. Leroy. *Robust Regression and Outlier Detection*. John Wiley & Sons, 1987.

[28] P. J. Rousseeuw. Least median of squares regression. *Journal of the American Statistical Association*, pages 871–880, Dec 1984.

[29] P. J. Rousseeuw. Multivariate estimation with high breakdown point. In *Mathematical Statistics and Applications*, pages 283–297. Reidel Publishing, 1985.

[30] E. L. G. Saukas and S. W. Song. Efficient selection algorithms on distributed memory computers. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–26. IEEE Computer Society, 1998.

[31] W. Stahel. Breakdown of covariance estimators, 1981. Research Report 31, Fachgruppe fur Statistik, ETH, Zurich.

[32] R. A. van de Geijn. *Using PLAPACK*. Scientific and Engineering Computation Series. MIT Press, 1997.

[33] E. van den Berg. Seminar on high-performance computing: Parallel. Seminar for ACES: Centre for Advanced Computations in Engineering Science, available at http://www.nus.edu.sg/ACES/seminars/2001/ewout/index.htm, 2001.

[34] D. S. Watkins. *Fundamentals of Matrix Computations*. John Wiley & Sons, 1991.

[35] Westgrid: Western canada research grid. www.westgrid.ca.