Parallel Matrix Factorization in Big Data Analytics

by

Zhaochong Liu

(Under the Direction of John A. Miller)

Abstract

From one point of view, Matrix Factorization can be seen as the workhorse for modern predictive big data analytics. It is a very important technique widely used in statistics, including multiple regression. Parallelizing matrix factorization can increase the speed of analytics. In the era of big data, the speedup often can be further magnified by the size of the dataset. This paper discusses several common matrix factorization algorithms that have been developed through the years, as well as their applications in multiple regression. The paper also presents a novel parallel Householder QR Factorization algorithm. After conducting several experiments on two different testing environments, substantial speedups were obtained.

Index words: Big Data; Predictive analytics; Regression; Matrix factorization; Parallel programming.

Parallel Matrix Factorization in Big Data Analytics

by

Zhaochong Liu

B.S., Tianjin University of Technology, 2013

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2016

Parallel Matrix Factorization in Big Data Analytics

by

Zhaochong Liu

Approved:

Major Professor: John A. Miller

Committee: Khaled Rasheed
Laksmish Ramaswamy

Electronic Version Approved:

Suzanne Barbour
Dean of the Graduate School
The University of Georgia
August 2016

Dedication

To my family and all my friends, without whom none of my work would be possible.

Acknowledgments

This thesis owes its existence to the help, support and inspiration of several people. Firstly, I would like to express my sincere appreciation and gratitude to Dr. John A. Miller for his guidence during my research. His support and inspiring suggestions have been precious for the decelopment of this thesis content.

I am also indebted to Dr. Khaled Rasheed and Laksmish Ramaswamy, who serve as my committee members. In addition Hao Peng and Zhe Jin deserve special thanks, as helping me with writing.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the era of big data, when increasing amounts of data are becoming readily available, the ability to analyze the data in a reasonable amount of time becomes increasing important as well. In recent decades, multi-core, hyperthreading computing hardware has become relatively common, and this makes parallelization of many algorithms important for wide distribution and usage. In the field of big data analytics, especially predictive analytics, one important family of algorithms is matrix factorization. It includes LU Factorization, QR Factorization, SVD Factorization, and many more. They are widely used in linear regressions, dimension reductions, recommendation systems, among others. Therefore, parallelization of matrix factorization algorithm could result in significant speedup of analytics.

The JVM ecosystem supports many languages and libraries that can be interoperate as well as universally execute across platforms. JVM languages, including Closure, Groovy, Java, Jython, JRuby, Kotlin and Scala, are currently the most popular language family. For Big Data Analytics, several libraries provide underlying support for numerical linear algebra. As an example, Parallel Colt [1] provides parallel implementations of key algorithms needed for Big Data Analytics.

The ScalaTion Project is a system coded in Scala and supports Analytics, Simulation and Optimization [2]. The `linalgebra` and `analytics` packages as well as their parallel counterparts, `linalgebra.par` and `analytics.par`, provide several techniques, including Inverse, Cholesky Factorization and QR Factorization that can be used for Multiple Linear Regression. Parallel Colt, on the other hand, only has several factorization techniques, but does not apply them to Linear Regression. Householder QR Factorization is available in

both ScalaTion and Parallel Colt. However, the implementation in ScalaTion takes less memory because vector-to-vector operations are used in ScalaTion, while matrix-to-matrix operations are used in Parallel Colt. The goal of the current work is to provide efficient parallel implementations to complement the existing serial ones in ScalaTion.

The paper is organized in the following manner: As background, chapter 2 introduces LU, Cholesky, QR and SVD Factorizations. Chapter 3 discusses how to apply the mentioned matrix factorization techniques to the widely used analytics technique of Multiple Regression. Chapter 4 discusses support for developing and issues concerning parallel factorization algorithms/implementations. Other related work on parallel matrix factorization is given in chapter 5. Explanations of the parallel implementations are given in chapter 6, while the performance of the various implementations is presented in chapter 7. Finally, conclusions and future work are given in chapter 8.

Chapter 2

Matrix Factorization

Matrix Factorization, also known as Matrix Decomposition, is a technique to factor a single matrix into a product of two or more matrices. While it is easy to have highly effective parallel matrix multiplication, inverting or factoring a matrix in parallel can be more challenging [3]. ScalaTion contains several packages, traits, objects and classes supporting numerical linear algebra. Effort is underway in a `par` subpackage to provide parallel versions of these (`www.cs.uga.edu/~jam/scalation_1.2/src/main/scala/scalation/linalgebra/par`. In ScalaTion, switching from serial to parallel implementation is typically as easy as adding `par` to the import statement. Previously, parallel versions of vectors and matrices have been developed. This project has focused in improving serial versions of factorization and developing parallel versions. As part of the ScalaTion big data framework, support for interoperation with a NoSQL database [4] as well as large out-of-core matrices is also provided.

2.1 LU Factorization

"L" and "U" here refer to lower triangular matrix and upper triangular matrix, respectively. Therefore, LU Factorization decomposes a matrix into a product of those two kinds of matrices as follows:

$$A = LU \tag{2.1}$$

where A is the matrix that being factored [3]. This factorization provides an easy (as well as faster and more numerically stable) way to solve a system of equations.

$$A\mathbf{x} = LU\mathbf{x} = \mathbf{b}$$

where the goal is to solve for $\mathbf{x}$ given a square matrix A and a vector $\mathbf{b}$. Simply let $\mathbf{y} = U\mathbf{x}$, then $L\mathbf{y} = b$. First $\mathbf{y}$ can be solved by using forward substitution. Then solve for $\mathbf{x}$ by using backward substitution [5, 6].

2.2 Cholesky Factorization

Cholesky Factorization is a special case of LU Factorization [7] used when A is a positive definite, symmetric matrix, the result of which can be written as

$$A = LL^t \tag{2.2}$$

where L is a lower triangular matrix with positive diagonal elements.

2.3 QR Factorization

QR Factorization is one of the most widely used matrix decomposition methods. Given an m-by-n matrix A, it can be factored into

$$A = QR \tag{2.3}$$

where Q is orthogonal (i.e, has orthogonal columns) and R is upper triangular. Assume $m \geq n$ in this paper unless noted otherwise. Most QR algorithms work on columns of $A_{m\times n}$,

$$A_{m\times n} = [\mathbf{a_1}, \mathbf{a_2}, \ldots, \mathbf{a_n}] \tag{2.4}$$

where $\mathbf{a_j}$ is the jth column vector in A.

2.3.1 MGS QR Factorization

One type of factorization algorithm is the Gram-Schmidt orthogonalization, either Classical Gram-Schmidt (CGS) or the more numerically stable Modified Gram-Schmidt (MGS). CGS computes the length of the orthogonal projections of a vector onto certain directions and subtracts them from the original vector. In this way, the new vector will be orthogonal to all the previously calculated vectors. However, [8] argues that the vectors may not have truly orthogonal due to rounding errors. To reduce this problem [8], MGS stores the 2-norm of $\mathbf{a_1}$ in R and normalizes $\mathbf{a_1}$, denoted as $\mathbf{q_1}$. For the rest of the column vectors, subtract their own projections onto $\mathbf{q_1}$.

$$\begin{cases} R_{11} = \left\| \mathbf{a_1} \right\|, \ \mathbf{q_1} = \mathbf{a_1}/R_{11} \\ R_{1j} = \mathbf{q_1^t a_j}, \ \mathbf{a_j}^{(1)} = \mathbf{a_j} - \mathbf{q_1} R_{1j} \end{cases}$$

where j goes from 2 to n. By doing the subtraction, $\mathbf{q_1}$ will be orthogonal to the rest columns. Then compute the 2-norm of and normalize vector $\mathbf{a_2}^{(1)}$, denoted as $\mathbf{q_2}$ As with the first step, for column vectors $\mathbf{a_3}^{(1)}, \mathbf{a_4}^{(1)} \dots \mathbf{a_n}^{(1)}$ subtract their projections onto $\mathbf{q_2}$.

Therefore, when $1 \leq j < k-1$ and $2 \leq k \leq n$,

$$\begin{cases} R_{kk} = \left\| \mathbf{a_k}^{(k-1)} \right\|, \ \mathbf{q_k} = \mathbf{a_k}/R_{kk} \\ R_{kj} = \mathbf{q_k}^{\mathbf{t}} \mathbf{a_j}^{(k-1)}, \ \mathbf{a_j}^{(k)} = \mathbf{a_j}^{(k-1)} - \mathbf{q_k} R_{kj} \end{cases}$$

The orthogonal Q matrix and the updated parts of the origianl A matrix can be written into one single matrix when doing the calculations.

2.3.2 Householder QR Factorization

A Householder QR Factorization avoids numerical instability by reflecting a vector onto a direction (represented by a unit vector, e.g., $\mathbf{e_0} = [1, 0, \dots, 0]^t$, rather than projecting (which loses accuracy near orthogonality). The reflected vector will have the same length as the original.

Householder QR Factorization can factor a full n rank matrix $A_{m \times n}$ into Q and R. The following formulas and corresponding code (synced to start indices at 0) represent a simple and straightforward way to implement Householder QR Factorization and are based on the following classnotes [9]. Before factoring matrix A, a method is needed to compute a Householder vector $\mathbf{v}$ from a vector $\mathbf{x}$.

$$\alpha = \text{sign}(x_0) \left\| \mathbf{x} \right\| \text{ and } \mathbf{v} = \mathbf{x} + \alpha \mathbf{e_0}$$

In ScalaTion these become

```
def houseV (x: VectorD): VectorD = {
    val α = signum (x(0)) * x.norm
    x + (0, α)
} // houseV
```

Column by column the A matrix can be turned into upper triangular (i.e., R) by performing Householder reflections to zero A below the main diagonal. In the simple implementation, this is done by computing a Householder matrix H from the Householder vector $\mathbf{v}$. See chapter 6 for a more efficient way to do this.

$$d = \frac{2}{\mathbf{v^t v}} \text{ and } H = I - d \mathbf{v v^t}$$

In ScalaTion these become

```
def hReflect (v: VectorD) = {
    val d = 2.0 / v.normSq
    eye(v.dim) - outer (v, v) * d
} // hReflect
```

For each column j in A, compute

$$\mathbf{v} = \text{houseV}(A(j:m, j))$$

$$H = \text{hReflect}(\mathbf{v})$$

$$A = HA(j:m, j:n)$$

In ScalaTion these become

```
def factor () {
    for (j ← 0 until n)
        val v = houseV (a.col (j, j))
        val h = hReflect (v)
        a.times_ip_pre (h, j)
    // for
} // factor
```

If the full matrix Q is needed, it is simply the product of the Householder matrices. Several algorithms, such as multiple regression, do not need to fully materialize the Q matrix.

2.3.3 Rank Revealing QR Factorization

Even though Householder QR Factorization is the most popular method employed in Multiple Linear Regression, it can be only applied to the matrices with full column rank. Rank Revealing QR (RRQR) can handle the situation when the matrix is rank deficient and determine the rank of it [10].

This is actually a modified algorithm of the Householder QR Factorization. Before each step of Householder vector calculation, compute the norm of each column. Then, interchange the position of the column with the maximum norm and the first column in that step (kth column in kth step). If there are more than one column having the largest norm, choose the one with the smallest index. The rest of the procedure remains the same until the value of the maximum norm becomes zero, at which point the algorithms terminates yielding the rank and a factorization. In the end, RRQR gives one more matrix as the result of the Factorization,

$$Q^t A\Pi = \begin{bmatrix} R_{11}, & R_{12} \\ 0, & 0 \end{bmatrix}$$

where for $r = \text{rank}(A)$, R_{11} is an r-by-r upper triangular, nonsingular matrix, R_{12} is an r-by-(n-r) matrix, and Π is a permutation matrix.

2.3.4 Block Householder QR Factorization

The Block Householder QR Factorization technique basically uses the same algorithm as the Householder QR Factorization [11]. Rather than operating column by column, adjacent columns are grouped in blocks. Householder reflections are used to transform the first block A_{b1} into an upper triangular R_{b1}. These are then used to update the rest of blocks across the matrix. The same procedure is then applied to transform the second block A_{b2} into an upper triangular R_{b2}. Iterating through all the blocks achieves the desired result. The WY Representation for products of Householder Matrices [12] provides a means for reducing space requirements.

2.4 SVD Factorization

Singular Value Decomposition (SVD) Factorization is an important matrix decomposition algorithm in linear algebra. It is widely used in the area of signal processing and statistics and may be used for regression as well.

Given an m-by-n matrix A, which can be a real or a complex matrix, A can be factored into three matrices,

$$A = U\Sigma V^t$$

where U is an m-by-m orthogonal (unitary for complex) matrix, Σ is an m-by-n diagonal matrix, which carries the singular values of matrix A on the diagonal, and V^t is the transpose of matrix V, which is an n-by-n orthogonal/unitary matrix.

ScalaTion provides three algorithms for SVD Factorization: (1) find the eigenvalues and eigenvectors of A^tA [13], (2) the Golub-Kahan-Reinsch Algorithm [14], and (3) translation of the LAPACK Fortran DBDSQR subroutine [15] that uses an implicit zero-shift

QR algorithm. Each subsequent implementation is more robust, but also more complex. Currently, some of the SVD implementations are still undergoing testing.

Chapter 3

Application of Factorization in Regression

The Regression class in ScalaTion supports multiple linear regression. This is a technique which can develop correspondence and relationships among all the features or attributes [16]. Often, there is only one response variable defined as y. The rest of the variables are called explanatory x-variables upon which variable y linearly depends.

$$y = \mathbf{b} \cdot \mathbf{x} + \varepsilon = b_1 x_1 + b_2 x_2 + \cdots + b_n x_n + \varepsilon$$

where $x_1 = 1$ and ε represents residuals or noise. A dataset or training set will provide m occurrences for both y and $\mathbf{x}$. Collecting all the response values yields an m-dimensional response vector $\mathbf{y}$, while collecting all the x values yields an m-by-n matrix X.

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1n} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & x_{m3} & \dots & x_{mn} \end{bmatrix} \times \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} + \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_m \end{bmatrix}$$

Defining the vector ϵ as the collection of $\varepsilon's$, multiple linear regression models may be written as follows [16].

$$\mathbf{y} = X\mathbf{b} + \epsilon$$

ScalaTion, provides generalizations of regression including General Linear Models as well as Generalized Linear Models [4]. In order to solve for the coefficient/parameter vector $\mathbf{b}$, one minimizes a norm of the residuals $\left\| \epsilon \right\|_k$. In general this requires optimization, but when

$k = 2$, setting the gradient equal to zero, allows $\mathbf{b}$ to be found by solving a system of linear equations. This involves either inversion of factorization.

3.1 Using Matrix Inversion

Matrix Inversion is the easiest way to understand among all the other solution methods for multiple linear regression. To perform multiple linear regression, the most important step is to minimizes the residuals. Assuming a 2-norm (or sum of squares), one may write.

$$\left\|\epsilon\right\|^2 = \left\|\mathbf{y} - X\mathbf{b}\right\|^2$$

$$\left\|\epsilon\right\|^2 = \mathbf{y^t y} - 2\mathbf{b^t} X^t \mathbf{y} + \mathbf{b^t} X^t X \mathbf{b}$$

Taking the gradient with respect to $\mathbf{b}$ and setting it equal to $\mathbf{0}$ yields

$$X^t X\mathbf{b} - X^t\mathbf{y} = \mathbf{0}$$

Adding $X^t\mathbf{y}$ to both sides yields the Normal Equations.

$$X^t X\mathbf{b} = X^t\mathbf{y} \tag{3.1}$$

Solving for $\mathbf{b}$ gives the classical textbook solution [10].

$$\mathbf{b} = \left(X^t X\right)^{-1} X^t\mathbf{y}$$

In ScalaTion, the code looks very similar to the math.

```
val b = (x.t * x).inverse * x.t * y
```

3.2 Using Cholesky Factorization

It is well-known that solving for the inverse of a large matrix can lead to rounding errors and irregularities [17]. In order to deal with this problem, Cholesky Factorization can be used when solving for $\mathbf{b}$. Starting with the Normal Equations (equation 5),

$$X^t X\mathbf{b} = X^t\mathbf{y}$$

it can be shown that X^tX is a positive definite, symmetric matrix. Letting $A = X^tX$ and $\mathbf{c} = X^t\mathbf{y}$, one obtains $A\mathbf{b} = \mathbf{c}$. Now, matrix A can be factored using Cholesky Factorization.

$$LL^t\mathbf{b} = \mathbf{c}$$

As introduced earlier, L is a lower triangular matrix, so letting $\mathbf{z} = L^t\mathbf{b}$ yields the following.

$$L\mathbf{z} = \mathbf{c}$$

First use forward substitution to solve for $\mathbf{z}$ and then use backward substitution to solve for $\mathbf{b}$.

As the Cholesky Factorization is applied on X^tX rather than X, the rounding errors will be squared [18]. The QR Factorization avoids this drawback.

3.3 Using QR Factorization

QR Factorization is usually considered more robust than Matrix Inversion and Cholesky Factorization [10]. Similar to the Cholesky Factorization, QR begins with the Normal Equations, $X^tX\mathbf{b} = X^t\mathbf{y}$. Now, matrix X may be factored into using QR Factorization.

$$(QR)^t(QR)\mathbf{b} = (QR)^t\mathbf{y}$$

By using the property of the transpose of the product of two matrices, one may obtain the following.

$$R^t(Q^tQ)R\mathbf{b} = R^tQ^t\mathbf{y}$$

Since Q is an orthogonal matrix, its transpose is the same as its inverse, so Q^tQ simplifies to the identity matrix.

$$R^tR\mathbf{b} = R^tQ^t\mathbf{y}$$

Then, multipling both sides of the equation by $(R^t)^{-1}$ yields

$$R\mathbf{b} = Q^t\mathbf{y}$$

Since R is an upper triangular matrix, $\mathbf{b}$ can be easily solved for by using backward substitution. Note, actual implementations may save time and space by avoiding complete computation of Q or Q^t, see chapter 6.

3.4 Using SVD Factorization

Applying SVD Factorization to regression is a little different from the two methods introduced earlier when using them for regression. This time, start from the linear regression model instead of the Normal Equations. Decomposing the matrix X using SVD Factorization and introducing the result into the linear regression model yields,

$$\mathbf{y} = U\Sigma V^t\mathbf{b} + \epsilon$$

Letting $\mathbf{a} = \Sigma V^t\mathbf{b}$ gives

$$\mathbf{y} = U\mathbf{a} + \epsilon$$

where $\mathbf{a}$ is a vector of m elements, and $\mathbf{a}$ has the same dimension as vector $\mathbf{b}$. As with the other solution techniques, the 2-norm or sum of squares of residuals still needs to be minimized. By analogy,

$$\mathbf{a} = \left(U^t U\right)^{-1} U^t \mathbf{y}$$

As mentioned earlier, the matrix U is an orthogonal/unitary matrix, so $U^t U = I$.

$$\mathbf{a} = U^t \mathbf{y}$$

from which $\mathbf{a}$ can be solved. Once $\mathbf{a}$ is known, multiplying $\mathbf{a} = \Sigma V^t \mathbf{b}$ by the inverse of ΣV^t yields,

$$\mathbf{b} = V \Sigma^{-1} \mathbf{a}$$

where $\mathbf{a}$ is a vector and Σ^{-1} is a diagonal matrix where the diagonal elements are the reciprocal of the those in Σ [19].

Chapter 4

Support for Parallelism in Java and Scala

The central idea of parallel computing is to divide one problem into many sub-problems, where each can be solved simultaneously. In order to do that with maximum efficiency, the sub-problems should be independent of each other. Based on the analysis of all the aforementioned matrix factorization methods, it can be shown that no matter which method is applied, every step always requires the results from the previous step as inputs. Parallel processing requires operating system, language and library support as highlighted in Table I [20, 21]

Table 4.1: Support for Parallel Processing

Source	Feature	Description
Java	Thread	unit of concurrent/parallel execution
Java	ForkJoinTask	lighter weight thread-like entity
Java	Thread-safe Collections	allow safe access by multiple threads
Java	Parallel Streams	parallel execution on streams
Java	Synchronization	Semaphores, Locks, etc.
Scala	Parallel Collections	Collection class with methods that execute in parallel
Scala	.par method	returns a parallel version of a collection class
Scala	Akka Actors	parallelism via local or remote actors
ScalaTion	Coroutine	lightweight non-parallel concurrent execution
ScalaTion	Master-Worker	master distributes work to parallel workers

As mentioned above, due to the dependencies between each step, it requires synchronization when applying Householder QR Factorization on a matrix. In Java, Semaphore is a class that controls the number of threads accessing a target object. A single column should be treated as a target object, because either the Householder transformation or the calculation of a Householder vector should be done independently. Furthermore, the calculation of a

Householder vector should always be the first step when processing a column. A semaphore can be used to limit access to a column to one thread at a time. However, when using semaphores to achieve parallelism in this way, even though the Householder transformation of the columns can execute simultaneously, the algorithm still needs to block the threads to make sure that the Householder vector can be acquired correctly. This can lead to limited speedup, due to threads waiting for others to complete.

However, even if no good results seem to come from parallelizing the Householder matrix construction process, this does not mean that parallel algorithm is not suitable for the Householder Factorization algorithm. It can be shown that in Householder Factorization, the process of Householder transformation can be done column by column and there is no dependency between each other, which means it can be efficiently parallelized. The main technique will be well explained in chapter 6. In other words, even though parallelization of the global process did not produce good results, parallelizing the process of constructing the Householder matrix and processing Householder transformations is still reasonable after obtaining the Householder vectors. By applying the serial algorithm on the process of obtaining Householder vectors and parallelizing the rest, there is still considerable speedup.

Parallel collections in Scala provide a simple way to introduce parallelization into serial code. Many of the standard Collection classes have parallel versions (e.g., Array, ParArray). The key means for providing parallelism is via parallel versions of the `foreach` methods [21]. Since `for` comprehensions, call `foreach`, loops can be easily made to run in parallel. Consider the following two lines of code (the second appears in the `colHouse` shown in chapter 6).

for (j ← k + 1 until n) ...
for (j ← (k + 1 until n).par) ...

The first line uses a `Range` (`k + 1 until n`) to set up a loop that is executed sequentially. The `Range` collection class only stores the upper bound and the lower bound of the numeric range, as well as the traversal step. The call to the `.par` in the second line of code converts

the `Range` into a `ParRange`, so the loop can be executed in parallel. When the `foreach` method is invoked on a collection, subsets of elements will be assigned to different threads. To improve efficiency and reduce overhead, various strategies such as thread pools, fork/join framework and work stealing are utilized, see [21] for details.

Chapter 5

Related Work

In this chapter, work on parallelization of matrix factorization is reviewed. More specifically, it focuses on techniques to parallelize the Householder QR Factorization. As mentioned above, simply parallelizing the global process of Householder QR Factorization is unlikely to offer high speedups due to dependencies between each step.

The procedure of the Householder QR Factorization can be divided in to several tasks. If these tasks have no dependencies, they can be run in parallel without any interference, otherwise, they should be sheduled in order to minimize the wait times due to dependencies. Typically, a task would correspond to a transformation of an entire column of the original matrix, a transformation of a block of columns of the original matrix or making part of a transformation parallel (see chapter 6). The first two parallelization techniques have dependencies, while the third does not.

LAPACK [22]/PLASMA [23] provide parallel QR Factorization algorithms. Parallel Colt [1] mentioned in chapter 1, is a parallel Java implementation of the Colt package [24]. It consists a package called JPlasma, an Java interface of PLASMA, which provides parallel algorithms for Matrix Factorization including Block Householder QR Factorization, LU Factorizaion and Cholesky Factorization. The way they make Block Householder QR Factorization parallel is through the use of the parallel matrix operations, primarily matrix multiplication and subtraction, e.g,

$$A_k = H_k A_{k-1} = A_{k-1} - (WY^t)A_{k-1}$$

A good way of parallelizing Householder QR Factorization is introduced in [3]. The algorithm improves upon the QR Factorization package in LAPACK. This algorithm divides the

matrix into square blocks and operates on them as small tasks. Based on the dependencies among those tasks, they can be dynamically scheduled according to the computational resources. It can be shown that there are dependencies between each step of the algorithm. The researchers of [3] found a DAG, in which the nodes are elementary tasks and the edges represent the dependencies among them. The way of finding the DAG is using the technique of "look ahead" which is presented in [25]. The DAG shows all the dependencies among tasks and can be used to schedule the tasks "asynchronously and independently" [3].

According to the authors of [3], this algorithm can scale almost perfectly in multi-core architectures by using this efficient dynamic scheduling technique, but it requires 25% more floating-point operations than the serial algorithm. Compared to [3], the current paper uses an unblocked means to achieve parallelism in Householder QR Factorization. Our experimental results in Householder Factorization show that a Householder transformation can be done column by column and there is no dependency among the columns, which means the algorithm can be easily parallelized.

OptiML [26] is a domain-specific language (DSL) build on top of Scala for developing machine learning (ML) algorithms and applications. Algorithms can be written in the OptiML language in a straight forward, serial manner. OptiML then converts the code into appropriate parallel and distributed C++, Scala, and CUDA code. Users are then responsible for running the generated code on the appropriate hardware. For different algorithms, datasets, and characteristics of the target hardware, it is hard to predit the optimal parallel or distributed hardware to use. However, since OptiML is able to generate code for a variety of hardware, users can test their algorithms on all available parallel and distributed hardware platforms and obtain the optimal results from the most appropriate hardware. The technique named "lightweight modular staging" is used in OpitML to build an intermediate representation (IR) between a program written in OptiML code and the corresponding generated code. OptiML can also analyze and optimize this IR to generate a portable, productive programming model with the best possible performance. Because this

paper only focuses on Multiple Linear Regression, only the experimental results of Linear Regression are presented. The authors of OptiML compare the OptiML application with the corresponding code of MATLAB and C++. The results show that as the number of cores increase, OptiML can get better results when comparing with MATLAB. Also, with the optimization of OptiML, the OptiML version code can be nearly as fast as a "hand-optimized, manually-parallelized" C++ code. Another big advantage of OptiML is that the OptiML code is easy to write and understand. However, when compiling OptiML code, the compiler errors reported are very technical, therefore hard to understand for average users. The OptiML code is also very difficult to debug because users must dig through layers of generated code in order to find the source of the run-time error.

Chapter 6

Parallel Implementations

Except for SVD, which is still under development, the most complex and hardest to make parallel was Householder QR Factorization. From chapter 2 where Householder Factorization is discussed in detail, it can be shown that in order to obtain the correct Householder matrix in each step, the procedure would always need to wait for the required matrix transformations to be completed in the previous step in order to extract the corresponding Householder vectors. The Householder vectors must be extracted from left to right column wise. Therefore, the primary dependency when constructing the Householder matrix is that the current Householder matrix Q_n would require a vector that has been transformed $n-1$ times.

Unlike the other QR Factorization techniques, Householder QR Factorization results in one single matrix at the end of the calculation. The R matrix can be easily obtained, because it is stored in the upper-right part of the final matrix. If a Q is actually needed, one may use the Householder vectors to create it. The Householder vectors are stored in the lower-left part of the final matrix.

6.1 Improved Serial Implementation

Chapter 4 presented a simple Householder QR Factoriztion which utilizes high level matrix operations. Although it is concise, it is not as efficient as the serial algorithm discussed in this section that also serves as the basis for the parallel implementation. Greater efficiency is achieved by replacing matrix operations such as the computation of the H matrix and its multiplication times the A matrix.

```
val v = houseV (a.col (j, j))
val h = hReflect (v)
a.times_ip_pre (h, j)
```

First, in the `houseV` method, the Householder vector $\mathbf{u}$ for $\mathbf{x} = A(j:m, j)$ is computed based upon the formula given in the section two of paper [27]. Then, the method to compute the Householder Reflection matrix and its multiplication by A are replaced by a simpler vector operations.

6.2 Making the Serial Algorithm Parallel

As mentioned in chapter 4, the Householder transformation can be efficiently parallelized after obtaining the Householder vector. The process of forming the Householder Matrix and subsequently using it for Householder transformation can be done by vector-to-vector operations rather than matrix-to-matrix operations. This procedure can be treated as updating the original matrix column by column, which can be done in parallel. The function below is used to obtain the Householder vector for column k, and is needed for the Householder transformation of the whole matrix.

```
def colHouse (k: Int) {
    var xnorm = aa.col(k, k).norm
    if (xnorm != 0.0) {
        if (aa(k, k) < 0.0) xnorm = -xnorm
        for (i ← k until m) aa(i, k) /= xnorm
        aa(k, k) += 1.0
    } // if
    r(k, k) = -xnorm
```

```
    for (j ← (k + 1 until n).par) transformAA (j, k)
  } // colHouse
```

The kth step of the Householder QR Factorization starts by computing the Householder vector for kth column, which is used in the Householder transformation, $A_k = Q_k A_{k-1}$. The first part of the `colHouse` method sets $r_{kk} = -\text{sign}(a_{kk}) \left\| \mathbf{a_{k:m,k}} \right\|$ and calculates the kth column of A_{k-1} by dividing the portion of the column below a_{kk} by $-r_{kk}$ and then incrementing a_{kk}. The second part of the `colHouse` method contains a loop that repeatedly calls the `transformAA` method, which transforms the jth column of A_{k-1} based on the kth Householder vector. Each of columns can be computed in parallel because there are no dependencies between them.

JAMA [28] uses a more memory efficient approach by transforming the matrix column by column. Instead of computing the Householder matrix H, this procedure directly calculates the transformed vector $H\mathbf{c}$. The `colHouse` method implemented in ScalaTion uses the same idea. Method `colHouse` first computes the non-unit Householder vector, and method `transformAA` replaces every $\mathbf{c}$ with corresponding $H\mathbf{c}$ based on the code above in parallel. As discussed in chapter 4, parallelizing the Householder transformation is the main parallel portion of this algorithm. Another important feature of the implementation is that it requires substantially less memory.

Besides parallelism, there are also several ways to further improve the performance. The `MatrixD` class in ScalaTion manipulates numeric matrices with elements of type `Double`. Accessing an element, e.g., `mat(i, j)` is a two step process of using the row index `i` to get the ith row, which is a one dimensional array, and then using the column index `j` to get the jth element in this array. If code iterates over the column index with i fixed, then it is up to twice as fast to maintain a reference to the row and only use single level indexing. This is done as much as possible in ScalaTion while still maintaining the high-level abstraction of a matrix (e.g., some packages simply reduce everything to a single dimensional array with mn elements). Although less convenient, the same can be done on a column basis.

Another means is to take full advantage of the caches. Typically the caches are roughly an order of magnitude faster than main memory, so if algorithms maximize their locality of reference (e.g., access row-wise if matrices are stored row-by-row), the code may run substantially faster. If an algorithm processes columns and the matrix is stored row-by-row, it may be beneficial to transpose the matrix first.

Chapter 7

Performance Results

The tests are done by using different numbers of instances in order to see the trend of the speedup with the growth of the matrix. The test datasets are all simulated because much of the real data that can be found on the Internet either are too small or not of full rank. The sizes of datasets are 10000*300, 20000*600, 30000*900, 40000*1200 and 50000*1500. Since in realistic situations, the number of instances is usually much larger than the number of attributes, the simulated matrices are all tall and thin.

The Zcluster is provided by Georgia Advanced Computing Resource Center. The operating system on the Zcluster is 64-bit Red Hat Enterprise Linux 5. Any particular test was run a single compute node on an Intel X5650 2.67Hz limited to 12 cores and 48 GB of memory. The test result are shown in figure 7.1. The x-axis indicates the size of the matrix and the y-axis indicates the running time in millisecond (ms).

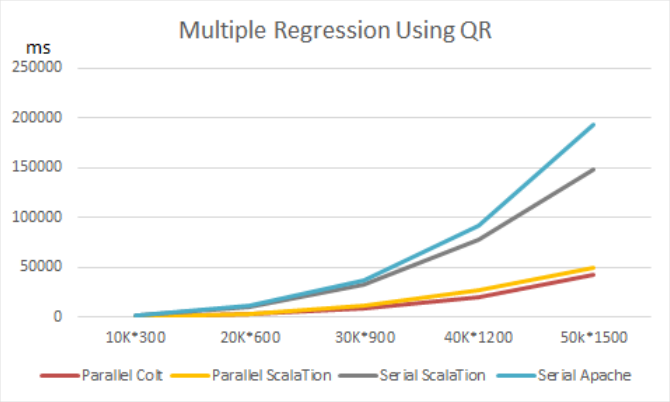

Figure 7.1: Regression using QR Factorization on Zcluster

Figure 7.1 indicates the running time of Linear Regression in Apache Commons, Parallel Colt and ScalaTion on matrices of increasing sizes. It shows Linear Regression in Apache

Commons and the serial Regression in ScalaTion seem to have similar performance. While for relatively small matrices, Apache Regression is almost the same with that of ScalaTion, but as the size of the matrix increases, the ScalaTion serial Regression performs better. The parallel implementation of Regression in ScalaTion and Parallel Colt is much faster than the other techniques. As the size of the matrix grows, the Parallel Colt performs slightly better than that of ScalaTion.

Table 7.1: Memory Usage

	10K*300	**20K*600**	**30K*900**	**40K*1200**	**50K*1500**
Par. Colt	1.2%	2.7%	5.4%	9.8%	15.7%
ScalaTion	1.0%	2.1%	4.4%	8.4%	13.1%

Even though the Regression in Parallel Colt is little faster than the parallel Regression in ScalaTion, the memory usage of Parallel Colt is higher. Table 7.1 shows that the memory usage of Parallel Colt is always 20% higher than that of ScalaTion when doing Multiple Linear Regression using QR Factorization. This test was conducted on a interactive node of Zcluster with 48 cores and 128G memory on an AMD Opteron(tm) processor 6174.

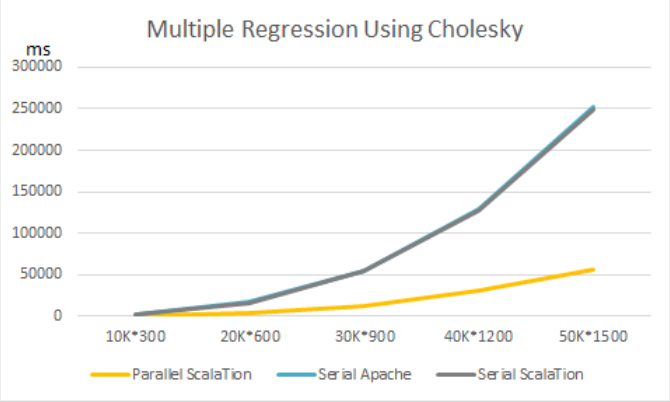

Figure 7.2: Regression using Cholesky Factorization on Zcluster

Figures 7.2 and 7.3 compare the performance between Apache Commons and ScalaTion when doing Multiple Regression using Cholesky Factorization and the Inverse Technique on different sizes of matrices. Since Regression in Apache Commons uses QR Factorization, we use the package to do the key steps on Regression for the Cholesky Factorization and Inverse

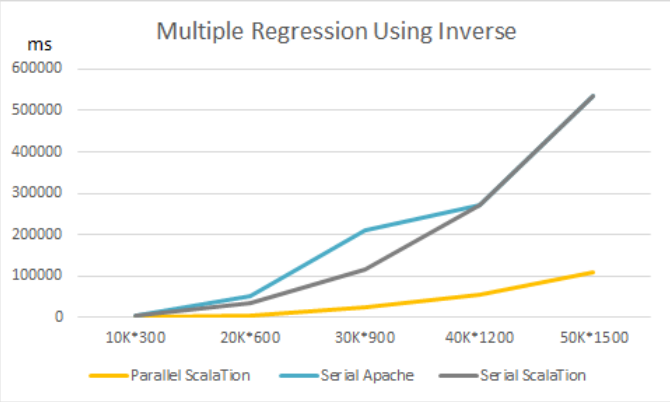

Figure 7.3: Regression using Inverse Technique on Zcluster

techniques that start with X^tX. This produces a positive definite, symmetric n-by-n square matrix. Then we implement the rest of Multiple Regression including solving for the coefficients. The test matrices are all the same as the matrices used in Multiple Regression using Householder QR Factorization. These two figures show that the performance of Regression using parallel Cholesky and parallel Inverse [29] are much better than ScalaTion's serial or Apache Commons implementations.

To further test the performance of the parallel algorithm, several scalability tests are also performed. We are using the matrix of size 10000*300 to test the running time on different numbers of threads, such as, 2 threads, 4 threads, 6 threads, 8 threads, 10 threads and 12 threads. We also calculated the speed ups of each number of threads compared with the serial algorithm.

Figure 7.4 shows the scalability testing result of Householder QR Factorization. The x-axis indicates different numbers of threads. The left y-axis indicates the running time in millisecond(ms) and the right y-axis indicates the speed up compared with the serial algorithm. As Zcluster is often saturated, the nodes typically have a load around 8 with 12 cores. However, we can still observe the trend that as the number of threads increases, the speed up goes up and the the running time goes down.

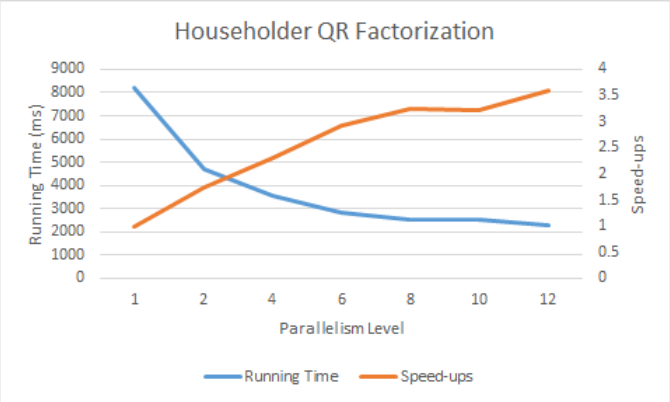

Figure 7.4: Scalability of QR Factorization on Zcluster

Figure 7.5, Figure 7.6 and Figure 7.7 show the scalability testing results of Regression Using Householder QR Factorization, Cholesky Factorization and Inverse Technique respectively. They all achieved a inreasing trend of speed up as the number of threads increases.

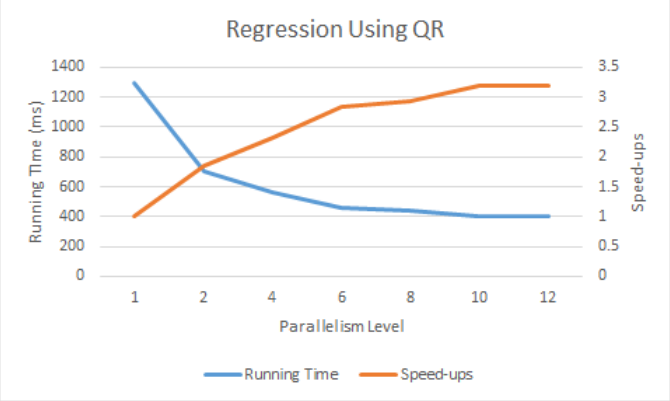

Figure 7.5: Scalability of Regression using QR on Zcluster

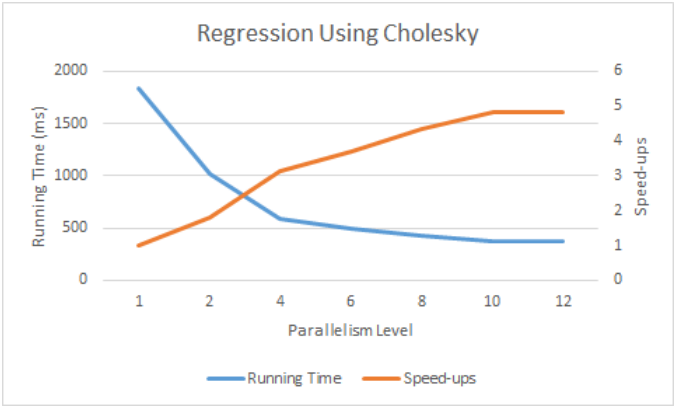

Figure 7.6: Scalability of Regression using Cholesky on Zcluster

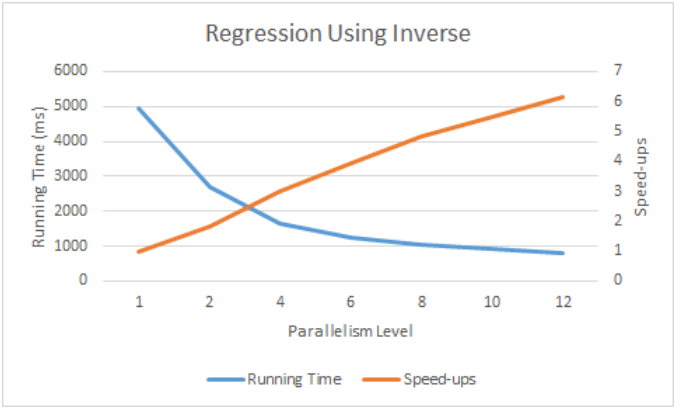

Figure 7.7: Scalability of Regression using Inverse on Zcluster

Chapter 8

Conclusions and Future Work

There are many forms of Regression used in Big Data Analytics with solution techniques ranging from Matrix Factorization for Least Squares to sophisticated optimization algorithms used to minimize "minus log-likelihoods". ScalaTion provides both and now is in the process of parallelizing its serial algorithms. The first step is parallelizing the Matrix Factorization algorithms and applying them to the problem of Multiple Linear Regression. A novel algorithm for parallel Matrix Factorization was developed and applied to the problem of Multiple Linear Regression. Based on the testing environments, speedup factors up to five were obtained by this algorithm.

For future work, the plan is to test the algorithms on systems with more cores and lighter loads, which should be more indicative of the true potential of the algorithms. QR Factorization algorithms based on Given rotations will be considered given their potential for parallelism. The plan also calls for testing the newly developed parallel algorithm against the already developed, but not widely used parallel block-oriented Householder QR Factorization. Beyond this, for cases were these algorithms may fail, serial and parallel implementation for RRQR, SVD and even stochastic gradient will be tested for performance and robustness.

Bibliography

[1] P. Wendykier and J. G. Nagy, "Parallel colt: A high-performance Java library for scientific computing and image processing," *ACM transactions on mathematical software (TOMS)*, vol. 37, no. 3, p. 31, 2010.

[2] J. A. Miller, J. Han, and M. Hybinette, "Using domain specific language for modeling and simulation: Scalation as a case study," in *Simulation Conference (WSC), Proceedings of the 2010 Winter*. IEEE, 2010, pp. 741–752.

[3] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "Parallel tiled QR factorization for multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 13, pp. 1573–1590, 2008.

[4] J. A. Miller. (2009) Scalation. [Online]. Available: http://www.cs.uga.edu/~jam/scalation.html

[5] X. Zhang, "Matrix analysis and applications," *Tsinghua and Springer Publishing house, Beijing*, 2004.

[6] MathWorld. LU decomposition. [Online]. Available: http://mathworld.wolfram.com/LUDecomposition.html

[7] J. Wang and G. Wu, "Recurrent neural networks for lu decomposition and cholesky factorization," *Mathematical and computer modelling*, vol. 18, no. 6, pp. 1–8, 1993.

[8] Å. Björck, "Numerics of Gram-Schmidt orthogonalization," *Linear Algebra and Its Applications*, vol. 197, pp. 297–316, 1994.

[9] (2009) The QR factorization. [Online]. Available: http://www.math.usm.edu/lambers/mat610/sum10/lecture9.pdf

[10] G. H. Golub and C. F. Van Loan, *Matrix Computations*. JHU Press, 2012, vol. 3.

[11] F. Rotella and I. Zambettakis, "Block householder transformation for parallel QR factorization," *Applied mathematics letters*, vol. 12, no. 4, pp. 29–34, 1999.

[12] R. Schreiber and C. Van Loan, "A storage-efficient WY representation for products of Householder transformations," *SIAM Journal on Scientific and Statistical Computing*, vol. 10, no. 1, pp. 53–57, 1989.

[13] K. Baker, "Singular value decomposition tutorial," *The Ohio State University*, vol. 24, 2005.

[14] G. H. Golub and C. Reinsch, "Singular value decomposition and least squares solutions," *Numerische mathematik*, vol. 14, no. 5, pp. 403–420, 1970.

[15] J. Demmel and W. Kahan, "Accurate singular values of bidiagonal matrices," *SIAM Journal on Scientific and Statistical Computing*, vol. 11, no. 5, pp. 873–912, 1990.

[16] A. S. Kapadia, W. Chan, and L. A. Moyé, *Mathematical statistics with applications*. CRC Press, 2005.

[17] A. M. Turing, "Rounding-off errors in matrix processes," *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 1, no. 1, pp. 287–308, 1948.

[18] L. DO Q, "Numerically efficient methods for solving least squares problems," 2012.

[19] J. Mandel, "Use of the singular value decomposition in regression analysis," *The American Statistician*, vol. 36, no. 1, pp. 15–24, 1982.

[20] D. Grossman. (2011) A sophomoric introduction to shared-memory parallelism and concurrency. [Online]. Available: https://homes.cs.washington.edu/~djg/teachingMaterials/spac/sophomoricParallelismAndConcurrency.pdf

[21] A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky, "A generic parallel collection framework," in *Euro-Par 2011 Parallel Processing*. Springer, 2011, pp. 136–147.

[22] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' guide*. Siam, 1999, vol. 9.

[23] J. Kurzak, H. Ltaief, J. Dongarra, and R. M. Badia, "Scheduling dense linear algebra operations on multicore processors," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 1, pp. 15–44, 2010.

[24] Colt. [Online]. Available: http://dst.lbl.gov/ACSSoftware/colt/

[25] J. Kurzak and J. Dongarra, "Implementing linear algebra routines on multi-core processors with pipelining and a look ahead," in *Applied Parallel Computing. State of the Art in Scientific Computing*. Springer, 2006, pp. 147–156.

[26] A. Sujeeth, H. Lee, K. Brown, T. Rompf, H. Chafi, M. Wu, A. Atreya, M. Odersky, and K. Olukotun, "OptiML: an implicitly parallel domain-specific language for machine learning," in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 2011, pp. 609–616.

[27] A. Dubrulle, "Householder transformations revisited," *SIAM Journal on Matrix Analysis and Applications*, vol. 22, no. 1, pp. 33–40, 2000.

[28] J. Hicklin, C. Moler, P. Webb, R. F. Boisvert, B. Miller, R. Pozo, and K. Remington. Jama: A Java matrix package. [Online]. Available: http://math.nist.gov/javanumerics/jama

[29] Y. L. LI, "Evaluation of parallel implementation of dense and sparse matrix for sclation library."