****



**本 科 毕 业 设 计**

院 系 计算机科学与技术系

专 业 计算机科学与技术

题 目 基于Spark的大规模矩阵分解与求逆

年 级 2011 学 号 111220026

学生姓名 高兴坤

指导老师 黄宜华 职 称 教授

论文提交日期 2015年06月

**南京大学本科生毕业论文（设计、作品）中文摘要**

题目： 基于Spark的大规模矩阵分解与求逆

计算机科学与技术系 院系 计算机科学与技术 专业 2011 级本科生姓名： 高兴坤

指导教师（姓名、职称）： 黄宜华 教授

摘要：

随着大数据时代的来临，对大规模数据的计算分析成为研究机构和商业用户关注的焦点。相应的，Apache Hadoop, Apache Spark等主流大数据计算引擎的出现为构建大数据分析平台提供了强有力的支撑。作为代数运算的基石，矩阵的分解和求逆在科学计算、数据挖掘、机器学习等诸多领域有着广泛的应用。然而大规模矩阵分解和求逆是一个非常耗时的计算过程，一个重要的原因是矩阵分解与求逆运算结果中的一个元素往往依赖于输入矩阵的若干个元素，这种数据依赖关系使得计算过程难以在现有的大数据平台上很好地并行。因此，需要研究在主流大数据计算引擎上如何基于分布并行化计算方法来提高大规模矩阵分解和求逆的计算性能，提供基于这些平台的大规模分布式矩阵分解与求逆高效运算方法。这对大数据时代的数据分析有着尤为重要的现实意义。

针对这一问题，本文基于Spark平台研究实现了大规模分布式稠密矩阵LU分解、 Cholesky分解以及求逆运算方法。本文分布和并行化设计的思路是分而治之，通过对矩阵进行分块将整个计算任务划分为更小的计算任务，基于对各个分块的分布和并行化处理，最终实现高效的并行化矩阵分解和矩阵求逆运算。本文首先分析上述矩阵操作的分布式算法，再将整个计算过程转化为Spark平台上的一系列 jobs，并探讨实现过程中的一些优化方法。实验结果表明算法的实现具有较高的效率和良好的扩展性。

关键词：矩阵分解与求逆；分布式算法；Spark

**南京大学本科生毕业论文（设计、作品）英文摘要**

THESIS：Large-scale Matrix Decomposition and Inversion Based on Spark

DEPARTMENT：Computer Science and Technology

SPECIALIZATION: Computer Science and Technology

UNDERGRADUATE: Xingkun Gao

MENTOR: Professor. Yihua Huang

ABSTRACT：

With the advent of the era of Big Data, computational analysis of large-scale data has become the focus of attention of research institutes and business users. At the same time, Apache Hadoop, Apache Spark and many other today’s mainstream big data computing engine can provide strong support to build one’s own Big Data Analytics platform. As the cornerstone of algebra, matrix decomposition and inversion have a wide range of applications in scientific computing, data mining, machine learning, and many other fields, but the computing of large scale matrix decomposition and inversion operations are very time-consuming. An important reason is that each element of the result matrix often depends on multiple elements of the input matrix, and these dependencies make the computation difficult to be partitioned. Therefore, study of how to speed up the performance of large-scale matrix decompositions and inversion based on optimized distributed parallel computing and efficient implementations of these operations on these popular platforms is required. It has important practical significance particularly in the era of big data.

To solve this problem, this article presents an implementation of large-scale distributed dense matrix LU decomposition, Cholesky decomposition and Inversion algorithms based on Spark. We use the idea of “divide and conquer” to achieve efficient parallelization of matrix decomposition and inversion, which partitions the entire computing tasks into smaller subtasks by cutting the large matrix into much smaller blocks and calculating each separate block. We first analyzes the distributed algorithm of these matrix operation, and then presents the technique to calculate these tasks using a pipeline of Spark jobs. We also present optimizations of this technique. Experimental results show that the implementation achieves efﬁcient performance and good scalability.

KEY WORDS: matrix decomposition and inversion; distributed algorithm; Spark

目录

[第一章 引言 1](#_Toc421592191)

[1.1 研究背景和意义 1](#_Toc421592192)

[1.2 相关工作 2](#_Toc421592193)

[1.3 本文工作 2](#_Toc421592194)

[1.4 论文结构 3](#_Toc421592195)

[第二章 技术背景 4](#_Toc421592196)

[2.1 Spark背景 4](#_Toc421592197)

[2.2 Spark系统构架 4](#_Toc421592198)

[2.3 Spark数据模型 5](#_Toc421592199)

[2.4 Spark函数调用 6](#_Toc421592200)

[2.4.1转换 6](#_Toc421592201)

[2.4.2动作 7](#_Toc421592202)

[2.5 Spark广播变量 7](#_Toc421592203)

[第三章 算法原理分析 7](#_Toc421592204)

[3.1 矩阵的LU分解 7](#_Toc421592205)

[3.1.1 LU分解的串行算法 8](#_Toc421592206)

[3.1.2 LU分解的分块算法 9](#_Toc421592207)

[3.2 矩阵的Cholesky分解 10](#_Toc421592208)

[3.2.1 Cholesky分解的串行算法 10](#_Toc421592209)

[3.2.2 Cholesky分解的分块算法 11](#_Toc421592210)

[3.3 矩阵的求逆 12](#_Toc421592211)

[3.3.1 下三角矩阵求逆的串行算法 12](#_Toc421592212)

[3.3.2 下三角矩阵求逆的分块算法 13](#_Toc421592213)

[第四章 基于Spark的算法实现 14](#_Toc421592214)

[4.1 迭代与递归 14](#_Toc421592215)

[4.2 矩阵输入格式 15](#_Toc421592216)

[4.2.1 行矩阵 15](#_Toc421592217)

[4.2.2 分块矩阵 15](#_Toc421592218)

[4.3 LU分解具体实现 16](#_Toc421592219)

[4.3.1迭代计算和 16](#_Toc421592220)

[4.3.2计算最终的L 18](#_Toc421592221)

[4.4 Cholesky分解具体实现 19](#_Toc421592222)

[4.4.1迭代计算 19](#_Toc421592223)

[4.5 求逆具体实现 20](#_Toc421592224)

[4.5.1 下三角矩阵Inversion具体实现 20](#_Toc421592225)

[4.6 矩阵乘法 20](#_Toc421592226)

[第五章 算法实现中的优化 21](#_Toc421592227)

[5.1 本地高性能线性代数库加速 21](#_Toc421592228)

[5.2 使用多个RDD 21](#_Toc421592229)

[第六章 实验评估 23](#_Toc421592230)

[6.1 实验环境 23](#_Toc421592231)

[6.2 优化效果评估 23](#_Toc421592232)

[6.2.1 测试调用BLAS优化 23](#_Toc421592233)

[6.2.2 测试使用多个RDD保存中间结果优化 24](#_Toc421592234)

[6.3 可扩展性评估 25](#_Toc421592235)

[6.3.1 数据可扩展性 25](#_Toc421592236)

[6.3.2 系统可扩展性 25](#_Toc421592237)

[第七章 总结与展望 26](#_Toc421592238)

[7.1 课题总结 26](#_Toc421592239)

[7.2 进一步的研究工作 26](#_Toc421592240)

[参考文献 28](#_Toc421592241)

[致谢 30](#_Toc421592242)

**基于Spark的大规模矩阵分解与求逆**

第一章 引言

## 1.1 研究背景和意义

随着新技术的发展特别是互联网的普及，数据不断爆炸式增长，同时计算能力也在不断提高，很多大规模数据密集型科学应用（例如数值优化，数据挖掘，人工智能等）开始越来越频繁地进入人们的视野。矩阵分解和求逆运算作为线性代数中的重要算子，在这些领域有着极为广泛的应用。例如，在线性优化中，可以基于矩阵的LU分解加速单纯形法[1]；在数值计算中，矩阵分解可以有效地降低计算复杂度[2]；而在当下逐渐流行的数据分析应用中，矩阵运算同样扮演着重要的角色。例如，在推荐算法中，矩阵分解就是重要的中间步骤[3]。

而随着大数据时代的来临，很多计算密集型软件应用的数据规模不断增长，传统意义上的单节点运算由于受到有限的内存、处理器等计算资源的制约，已经无法满足人们对数据处理的需求。这种情况下，使用分布式计算平台几乎成了唯一的选择。相对于单节点计算，分布式计算有以下优点：

1. 成本低廉。分布式集群可以通过一定数量的普通计算机搭建,成本相对较低。
2. 扩展性强。传统单节点计算往往采用提升硬件性能的方法在纵向上提高计算机处理数据的能力，而分布式系统可以通过增加计算节点的个数在横向上扩展系统的性能。
3. 系统吞吐容量大。分布式系统能通过对整个计算集群的管理获得非常大的计算容量或数据吞吐量。

Apache Spark就是一个满足上述特点的、优秀的大数据分布式处理平台，基于Spark编程框架，用户可以忽略分布式计算环境中的复杂细节，只需要根据自己的特定需求，调用相应的接口就可以高效地开发自己的软件应用。正是由于其高效、通用、易用、生态系统完善等优点，越来越多的研究机构和商业用户选择基于Spark搭建自己的数据处理与分析平台。

基于此，本文选择Spark平台来实现分布式矩阵分解和求逆运算方法，这些操作可以作为相关大规模数据计算分析过程的中间步骤，其高效的实现会大大提高整个计算任务的速度。

## 1.2 相关工作

当前，在对矩阵运算操作的研究方面，常见的加速手段基本上分为下面几类:

1. 针对线性代数运算的特点，利用计算机体系结构对计算本身做大量的底层优化。这类高性能运算包（库）往往涵盖了常见的线性代数操作或者拓展到常见的数值操作，适用于单个计算节点小规模数据量的计算，并且已经发展得相当成熟，性能非常优异。例如，BLAS[7]是一个基本线性代数函数集合，它提供对三种不同层次的线性代数操作（向量与向量，矩阵与向量，矩阵与矩阵）的支持，拥有良好的可移植性和通用性，成为很多高质量线性代数软件包的基石。LAPACK[8]是一个拓展了BLAS第三层次操作的高质量线性代数库，它包含矩阵的LU、 Cholesky、QR等分解和求逆运算。它充分利用现代计算机的缓存机制，拥有良好的性能。
2. 基于多核、GPU等特定架构的算法实现。文献[10]针对多CPU/GPU混合结构实现了高效的LU分解算法，但它对计算机硬件的定制要求很高。
3. 传统分布式高性能线性代数运算包。ScaLAPAC[9]是一个拓展了LAPACK,可以运行在分布式集群上的高效运算库，但是它缺少统一的容错机制，在节点非常多的集群中往往表现出较差的可扩展性。
4. 基于Hadoop/Spark的矩阵并行计算。这类算法基于Hadoop、Spark或其他当下主流的分布式处理平台，实现算法的并行。文献[11]基于Hadoop平台利用LU分解给出了矩阵求逆的实现，拥有较好的效率和扩展性，但Hadoop具有不利于迭代计算的缺点。文献[12]针对稀疏、低秩矩阵采用了分布式随机梯度下降的方法提出了矩阵分解的算法，并在Spark上予以实现，但并没有涉及到普通的稠密矩阵。

## 1.3 本文工作

本文的主要工作是分析稠密矩阵的LU分解, Cholesky分解与求逆操作的算法思想，对算法做出改进使其能够在分布式环境下并行化，并基于当下主流的大规模数据处理引擎Spark给出算法的具体实现。本文还将讨论实现过程中的一些优化，最后通过实验展示算法良好的效率和可扩展性。

## 1.4 论文结构

本文一共分为七个部分。

第一部分是引言，介绍了本文的研究背景与研究意义，国内外的研究现状，本文的主要研究内容以及论文的整体结构。

第二部分介绍Spark平台相关的技术背景。

第三部分分别回顾稠密矩阵的LU分解、Cholesky分解以及求逆的单节点算法，并介绍适用于Spark平台的分块算法。

第四部分阐述如何结合Spark平台的编程框架给出这些算法的并行实现。

第五部分讨论实现过程中的一些性能优化思路。

第六部分通过实验展示算法的性能和可扩展性。

第七部分对本文工作做出总结，指出当前工作的不足并对未来工作进行展望。

# 第二章 技术背景

## 2.1 Spark背景

Spark分布式计算框架最早由UC Berkeley大学的AMPLab实验室开发，目前它已经成为Apache Software Foundation下以及同类大数据开源项目中最受关注的项目之一。在执行效率上，Spark[13]改进了Hadoop[14]批处理框架在迭代计算与交互式处理方面的不足，引入了RDD（弹性分布式数据集）的概念，允许将计算的中间结果保存在内存之中而无需写入磁盘，大大改善了迭代计算的效率，应用程序的性能得到数十倍甚至百倍的提升；在用户易用性上，Spark支持Java, Scala, Python等主流编程语言，提供了更加丰富的编程API；在通用性上，Spark针对不同的开发需求提供了更高阶的库，包括SQL查询，流式计算，机器学习算法以及图计算等等，开发者可以根据自己的需要，单独或组合使用这些库来处理复杂的数据分析任务。

目前，Spark已经成长为包含Spark SQL，Spark Streaming, MLlib, GraphX等多个子项目的完善的生态系统。使用Spark，用户只需调用编程接口来实现自己的数据计算任务，而无需关心数据的具体分布、并行调度策略、集群节点之间的数据传送与错误恢复等复杂的底层细节。凭借其高效、用户友好、通用的优点，Spark在学术界和工业界得到了广泛的实际应用。

## 2.2 Spark系统构架

1. Driver Program, 每一个Spark应用程序都包含一个driver program，它执行用户的main函数，并负责在集群调度所有的并行化操作。当应用程序刚从客户端提交上来的时候，Driver Program需要连接到负责分配整个集群上各种资源的Cluster Manager，并向其请求集群上能够使用的Executor。 然后，Driver Program会把应用程序的Jar包发送给这些Executor，计算过程中会将计算Task分配给这些Executor。
2. Cluster Manager, 它是向集群申请各种资源的一个额外的服务。Spark支持Mesos, YARN等等。
3. Executor, 它是运行在集群中每一个Worker Node上的一个进程，负责本节点上数据的读写以及各种计算。

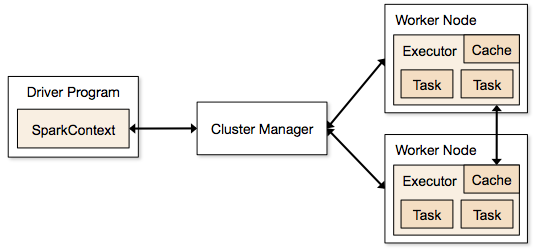


图1 Spark系统架构

## 2.3 Spark数据模型

Spark采用了一种被称为弹性分布式数据集（RDDs, Resilient Distributed Datasets）的分布式数据抽象，它支持工作集的重用，同时拥有非循环数据流模型（acyclic data flow model）的优点，即数据的负载均衡、位置感知和容错机制。同时RDD是一种受到限制的抽象，这体现在它是只读的，并且只能由已有的分布式数据集或其他RDD执行转换操作来创建。通过递归式地保存自己与其ParentRDD之间的转换记录(lineage)，当发生数据分区丢失时RDD可以利用这些记录重新进行计算，快速地恢复数据。

RDD的缺点是它只能提供粒度较粗的转换(coarse-grained transformation)并且它是只读的，这对于一些需要频繁更新数据集的一部分的应用，可能会带来性能上的不足。矩阵计算就是一个需要频繁更新矩阵本身的操作，因此实现算法时要格外注意如何解决这个问题。

RDD可以调用的操作包括转换(transformation)和动作(action)两类。每一个转换操作都会基于调用它的一个或多个parentRdd产生一个新的RDD，所有的转换操作都是懒惰(lazy evaluation)的，这意味着用户在一个RDD上调用一个转换操作并不会立即执行，Spark只是把它当作matadata记住了这些操作。只有当该RDD执行动作操作时，Spark才会利用这些metatada自后往前地追踪到当前RDD最古老的parentRDD，然后从前往后顺序计算这些RDD之间的转换。Spark提供了多种多样的转换，例如map, mapPartitions, union, join, filter等等。

动作(action)指的是需要向driver program返回计算结果或需要将数据导出至存储系统的一类操作。每一个动作都会生成一个Job，对数据进行计算并将计算结果返回给driver。Spark同样提供了多种多样的动作,例如reduce，collect, count， saveAsTextFile等等。

## 2.4 Spark函数调用

Spark提供了丰富的函数调用。下面简单介绍一下本文使用的一些函数。

### 2.4.1转换

1. map。 一个RDD可以通过map将RDD中的每一个元素通过用户指定的函数转化为另一种类型的元素，也就是将原来的RDD转变为指定类型的RDD。map函数接受的参数是一个函数算子。
2. mapValues。 与map类似，mapValues用于类型为(key, value)的PairRDD。不同的是mapValues不改变元素的key而只改变元素的value，因此新的RDD会保留原RDD的partitioner。
3. mapPartitions。 与map类似，它的第一个参数是一个函数，但mapPartitions是对原RDD的每一个partition而不是每一个element实施这个函数；它的第二个参数是一个Boolen类型，当原RDD的元素类型为(key, value)键值对，并且mapPartitions操作不会改变元素的key时，将第二个参数设为true告诉Spark不要改变RDD的partitioner。
4. filter。一个RDD可以通过filter对RDD中的所有元素按照一定条件进行过滤,过滤后的元素组成一个新的RDD。因此，filter函数接受的参数是一个返回类型为Boolean的函数，这个函数指明一个元素能不能通过筛选。
5. join。两个元素类型为(key, value)的PairRDD可以通过join操作将双方key相同的元素合并，返回一个新的RDD。
6. union。两个RDD可以通过union操作合并为一个RDD。
7. reduceByKey。一个PairRDD调用这个操作可以将原RDD中key相同的若干元素规约为一个元素，返回一个新的RDD。它的第一个参数是一个函数，这个函数相当于元素之间的运算符；第二个可选的参数可以指定新RDD的partition个数。类似于Hadoop中的combiner，spark在shuffle之前会首先对每个计算节点本地的元素先进行合并，从而减小shuffle的开销。

### 2.4.2动作

1. reduce。它的参数是一个满足交换律、结合律的二元运算符，原RDD的所有元素经过这个运算符计算，返回一个最终值。
2. collect。它将RDD中所有元素从集群上以一个数组的形式返回到Driver Program。collect是一个开销非常大的操作，只适合在小数据集上调用。

## 2.5 Spark广播变量

通常来说，当集群中的某个节点计算过程中发现需要使用用户自定义的变量时，会向driver program请求将该变量传送到该节点，每一个节点单独地对本地的变量副本进行计算，计算过程中对该副本做出的改动并不会反映到driver program。Spark提供了一种广播变量（broadcast variable）， 它允许开发者主动地将可能在在节点用到的变量广播到集群中的每个节点上，而不用在程序执行过程中当特定节点发现要使用这个变量再从driver program发送到这个节点上。Spark在传送广播变量时使用了一些高效的算法，减少了广播过程中的通信开销。文献[15]表明，在进行迭代计算时，广播变量能极大地提高整体性能。

# 第三章 算法原理分析

本章将分析矩阵的LU分解、Cholesky分解以及求逆的算法，着重分析算法设计的思想，为后面在Spark平台上的实现提供参考。

## 3.1 矩阵的LU分解

对于n阶方阵A, LU分解是指将A分解成一个下三角矩阵L和一个上三角矩阵U的乘积,即。若令，，，则LU分解可以表示为如下公式：

上面的公式其实可以展开为一个方程组，其中下三角矩阵L中的未知变量的个数为，同理上三角矩阵U中的未知变量的个数也为,即未知变量总数为，而实际的方程总数为，这是一个不定方程组。为了能够求出确定解，通常约定把下三角矩阵L的对角线的元素全部设为1.0。

实际运算过程中，为了提高运算结果的数值精度在每一轮迭代的时候需要选取主元。常用的方法有部分选主元方法和全选主元方法，其中部分选主元法能够取得算法复杂度和数值精度的较好平衡。我们这里考虑部分选主元方法，使用了部分选主元意味着在分解的迭代计算过程中原矩阵A会有行互换操作，实际上相当于左乘了一个置换矩阵P，即。

### 3.1.1 LU分解的串行算法

直接将公式(1)展开，不难得出，L和U可以由下面的公式推导出：

(2)

结合上面的公式，并考虑由部分选主元带来的行互换操作，则有如下的单节点LU分解算法：

算法**1**. 单节点矩阵LU分解算法.

输入：A为稠密矩阵

输出：A存放下三角矩阵L和上三角矩阵U；P为部分选主元生成的置换矩阵

LUDecomposition（A）

FOR i = 1 TO n

j = {j|a­ji =*max*(a­ii­, a­i+1i­,…, a­ni­)}

add j to P

swap row i and row j

FOR j = i + 1 TO n

a­ji = a­jj / a­ii

ENDFOR

FOR j = i + 1 TO n

FOR k = i + 1 TO n

a­jk ­-= a­ji xa­ik

ENDFOR

ENDFOR

ENDFOR

RETURN (A, P)

代码1 单节点矩阵LU分解算法

### 3.1.2 LU分解的分块算法

分块矩阵的思想在矩阵计算方法中很常见，也尤为有效。对一些各个元素之间依赖关系不是很强的矩阵操作而言，分块往往意味着可以实现计算的并行。传统意义上的单节点算法其实可以看作是一种“退化的”分块分解算法，输入矩阵的每一个分块是仅仅包含一个元素的子矩阵。

设对方阵A进行LU分解：

其中P为置换矩阵。考虑其2x2的分块形式：

化简一下可得:

其中，A1和A4都为方阵。整理一下有如下等式：

根据上面的等式，可以自然而然地想出下面的递归算法：先对子矩阵A1进行LU分解，得到P1、L1、U1；接着计算L2与U2；然后更新A4，再继续递归对A4进行LU分解。

进一步观察上面的等式可以发现，要想计算L2，首先需要计算出P2；P2的计算需要对进行LU分解，然而要想得到，则必须先计算出L2。可以发现计算依赖关系是循环错杂在一起的，因此需要对公式进行变形：

上述公式使用来代替原来的,消除了原来计算过程中的循环依赖关系。这样，原矩阵A的LU分解转化为两个更小的矩阵A1和A4的LU分解。具体实现时，可以设置A1的维度适当小，使其可以在单节点完成计算；接着计算与U2；再对A4更新，继续对A4进行LU分解。上面的过程是一个典型的递归。

## 3.2 矩阵的Cholesky分解

对n阶对称正定矩阵A，它的Cholesky分解指的将A分解成一个下三角矩阵和它的转置矩阵的乘积, 即。若令，，则矩阵的Cholesky分解可以表示为如下公式：

当矩阵A为n阶对称正定矩阵时，这样的L是一定存在的。可以看出，Cholesky分解其实可以看作是一种特殊的LU分解。并且由于输入矩阵是对称矩阵，其实计算过程中只需要使用矩阵的下三角部分；相应地，输出结果也只需要输出下三角矩阵L。

### 3.2.1 Cholesky分解的串行算法

直接将公式(5)展开，不难得出，矩阵L可以由下面的公式推导而出：

(6)

结合上面的公式，则有如下的单节点Cholesky分解算法：

算法**2**. 单节点矩阵Cholesky分解算法.

输入：A为对称正定矩阵

输出：A下三角矩阵L

CholeskyDecomposition（A）

FOR i = 1 TO n

FOR j = 1 TO i - 1

s = 0

FOR k = 1 TO j -1

s += a­jkxa­ik

ENDFOR

a­ij=(a­ii-s) / a­jj

ENDFOR

s = 0

FOR j = 1 TO i - 1

s += a­ijxa­ij

ENDFOR

a­ii­-=sqrt(a­ji-s)

ENDFOR

RETURN (A)

代码2 单节点矩阵Cholesky分解算法

### 3.2.2 Cholesky分解的分块算法

同理，对Cholesky分解也有相应的分块分解算法。

设对对称正定矩阵A进行Cholesky分解：

考虑其2x2的分块形式：

其中，A1和A4都为方阵。因此，整理一下有如下等式：

根据上面的等式，可以自然而然地想出下面的递归算法：先对子矩阵A1进行Cholesky分解，求出；再根据求出；然后更新，继续递归对进行Cholesky分解。这是一个典型的递归过程。

## 3.3 矩阵的求逆

对一个行列式值不为零的方阵A来说，它的求逆是指求出矩阵B，使得BA=AB=I，其中I为n阶单位矩阵。称B为方阵A的逆矩阵，记作。

与分解相比，矩阵的求逆操作更加复杂，一个主要原因是逆矩阵的元素与输入矩阵的很多元素都存在依赖关系，这使得将整个大的计算任务划分为若干个并行的子任务比较困难。传统的矩阵求逆方法有高斯-约当消元法，QR分解法、LU分解法等等。应当注意，在基于Spark的实现中，求逆运算最后转化为Spark中一系列jobs，LU分解的好处在于通过将块状矩阵的某个分块在集群的某个节点上直接计算，可以有效地减少整个算法迭代的次数；而高斯-约当消元法和QR分解法所需要的迭代次数几乎等于矩阵的阶数，每次的迭代只会更新矩阵的极小一部分，而其他部分在迭代中并没有改变。受Spark框架下RDD模型只能进行粗粒度的转换(coarse-grained transformation)这一限制， RDD上的转换大多数是不必要的，这直接导致算法的性能较差。例如，考虑初始输入矩阵的阶数为n,集群上单个节点可以直接进行本地LU分解的矩阵阶数限值为b，则高斯-约当消元法和QR分解法需要n次迭代，而LU分解的方法只需要n/b次迭代就可以将得到下三角矩阵L和上三角矩阵U，然后可以通过分别对这两个三角矩阵求逆，得到初始输入矩阵的逆矩阵。

为了对可逆矩阵A求逆，首先对A进行LU分解，即：

然后分别求出和，则有

根据(8)式，首先要对下三角矩阵L和上三角矩阵U求逆，这样问题变为如何对三角矩阵求逆。下面的例子以下三角矩阵为例讲述三角矩阵的求逆方法。直接展开公式，可以得到下面的公式

### 3.3.1 下三角矩阵求逆的串行算法

根据(9)式，则有如下的单节点下三角矩阵求逆算法

算法**3**. 单节点下三角矩阵求逆算法.

输入：L为下三角矩阵

输出：L为逆矩阵

LowerTriangularInversion（L）

FOR i = 1 TO n

FOR j = 1 TO i - 1

s = 0

FOR k = j TO i -1

ENDFOR

/

ENDFOR

= 1 /

ENDFOR

RETURN (L)

代码3 单节点下三角矩阵求逆算法

上三角矩阵的求逆算法与下三角矩阵极为类似，这里不再赘述。

### 3.3.2 下三角矩阵求逆的分块算法

设有下三角可逆矩阵L, 易知其逆矩阵也必为下三角矩阵,设其为X。考虑其2x2的分块形式：

即:

化简可得：

根据上面的等式，可以自然而然地想出下面的递归算法：首先递归对和求逆，对子矩阵求逆得到；再对子矩阵求逆得到；接着利用这两个子问题的解求出。这里的递归也是一个典型的递归。

# 第四章 基于Spark的算法实现

本章对上一章节介绍的矩阵LU分解、Cholesky分解以及求逆的分块算法做进一步分析，然后讨论在实际分布式平台实现时算法需要做出的一些改动以及算法层面的一些优化，接着详细阐明如何高效地将这些矩阵运算操作转化为Spark平台的一系列jobs。

## 4.1 迭代与递归

公式(4)、(7)、(10)都是递归表达式，它们都使用了分而治之（divide and conquer）的策略，即将整体的计算任务划分为规模更小的子任务分别求解，最后根据这些子任务的解合并为原问题的解。我们基于Spark的实现方法也来源于由这几个公式衍生出来的分块算法。在这里，对子问题划分的方法非常重要，尤其是当数据规模非常庞大的时候。

以LU分解为例，由公式(4)中的

可知

可以看到，为了求出和，必须首先算出、以及。而对子矩阵的分解只能得到(，，)，如果，的规模过于巨大的话（比如它们都是分布式的），对他们的求逆过程会非常困难。因此，需要将，的阶数(即的阶数)设置为较小的值，这样集群中的一个节点就可以存放并在本地计算，。实际基于Spark实现时，需要将这个子矩阵collect到driver program，由driver program计算(，，)，以及，然后利用Spark的广播机制，将它们广播至集群上所有的executor，然后拥有和矩阵partition的节点可以利用本地的广播副本进行各自的计算。这样实际上的分解过程完全在单节点上完成，算法只是再对继续递归，即算法退化成了尾递归算法。尾递归算法总可以实现为迭代算法。

而对于矩阵求逆过程中的下（上）三角矩阵求逆，由公式(10)可得

对和递归求逆可以得到和。此时和的阶数不需要太小，因为分布式矩阵的乘法天然就可以并行，也就无需设置它们的阶数使其可以在集群的单节点计算，它的实现仍然是一个递归实现。

## 4.2 矩阵输入格式

输入矩阵的元素类型使用双精度浮点数（Double）来保证数值精度，每个元素二进制表示下占8个字节。本文实现中矩阵可以有行矩阵和分块矩阵两种形式。

### 4.2.1 行矩阵

行矩阵格式用于从HDFS中读入原始的输入矩阵数据，采用这种存储方式更加符合开发者在Spark平台下读取输入数据的习惯，也有利于与其他的应用整合。每一行对应于原稠密矩阵的一行，每行的开始加上该行在矩阵中对应的行号，后面每个矩阵元素用逗号隔开，行号与元素之间用冒号隔开。例如，

0L: 1.0, 0.0, 0.0

1L：0.0，1.0，0.0

2L：0.0，0.0，1.0

上面这个文本共有三行，对应于一个三阶的单位矩阵。

4.2.2 分块矩阵

实际计算过程中主要使用分块矩阵。行矩阵可以其划分为若干个子矩阵，均匀分布在集群中的节点上。设原方阵的阶数为n，子矩阵的阶数为b，则共划分为个子块，如图2所示。其中，b的大小由driver program的计算能力决定，也可由开发者根据自己的情况指定。我们在实现中发现b等于1000能够取得较好的效果。

HDFS中输入矩阵由textFile方法读入变为按行存储的行矩阵，集群上每个节点存取矩阵的若干行，每行按照设定的分块数用flatMap方法将自己分成若干段，并为每一段附加一个该段所属于的分块标志blockID，再用groupByKey方法将blockID相同所有的分段聚集起来，并将它们组织为键值为blockID的分块。这样就成功将原行矩阵变为特定结构的分块矩阵，每一个分块表示为(blockID, matrix)。

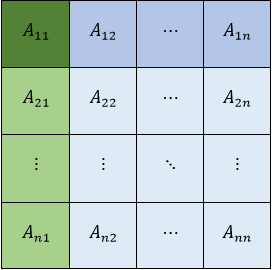


图2 LU分解：分块矩阵

## 4.3 LU分解具体实现

在Spark平台实现分布式矩阵LU分解的思路主要来自于公式(4)。整个算法是一个迭代算法。下面是具体迭代流程。

4.3.1迭代计算和

1. 每次迭代时，把矩阵尚未处理的部分（始终是一个方阵）看作新的待处理矩阵。通过filter方法将这个矩阵分为、、、四个子矩阵，每个子矩阵都是一个RDD。
2. 因为的阶数足够小,全部存放在集群中的某个节点上，可以首先把它collect到driver program利用单节点分解算法进行LU分解，并调用broadcast方法把计算结果（，，）和本身分别从driver program广播到集群上所有的节点。其中为稀疏的置换矩阵，实际实现中被压缩为一个数组。
3. 计算。对于集群中存放分布式矩阵的部分partition的节点而言，它们都接收到了driver广播的和的副本，根据公式可以推出，把属于这个RDD的所有分块左乘，这样就得到了。
4. 计算。对于集群中存放分布式矩阵的部分partition的节点而言，它们都接收到了driver广播的的副本，根据公式可以推出，把属于这个RDD的所有分块右乘，这样就得到了 。
5. 更新下一轮迭代的输入矩阵A。考察公式(4)，注意到有

故有

原来的公式可以重写为

这是一个非常重要的优化。式子表明分布式矩阵的更新计算实际上并不依赖于，也就是说不需要等到步骤3)和4)全部完成才能计算，它们可以同时进行。计算完成后，将分布式矩阵标志为下一轮迭代的输入矩阵。

1. 重复步骤1)继续迭代。

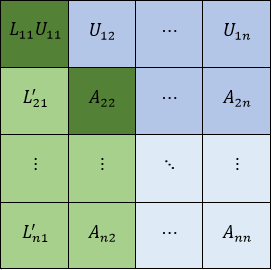


图3 LU分解：第一次迭代结束

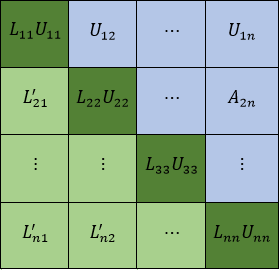


图4 LU分解：所有迭代结束

4.3.2计算最终的L

由公式可知，迭代完成后得到的所有需要用置换矩阵计算出。由于整个LU分解的置换矩阵P被压缩为一个一维的置换数组，数据量较小，可以将整个P广播到集群节点上。分布式矩阵中的blockID为(i,j)的分块左乘子置换矩阵计算出，如图5示。

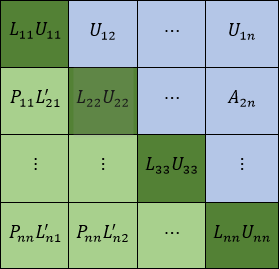


图5 LU分解：计算最终的L

## 4.4 Cholesky分解具体实现

Cholesky分解实际上和LU分解是类似的，但略有简化。因为输入的矩阵式对称的，所以只需要使用矩阵的上三角部分或下三角部分进行计算，计算结果也只需要返回下三角矩阵，上三角矩阵可以直接由其转置而得。

### 4.4.1迭代计算

1. 每次迭代时，把矩阵尚未处理的部分（始终是一个方阵）看作新的待处理矩阵A。通过filter方法将这个矩阵分为、、三个子矩阵，每个子矩阵都是一个RDD。
2. 因为的阶数足够小而全部存放在集群中的某个节点上，首先把它collect到driver program，利用单节点分解算法进行Cholesky分解，并调用broadcast方法把计算结果和本身分别从driver program广播到集群上所有的节点。
3. 计算。对于集群中存放分布式矩阵的部分partition的节点而言，它们都接收到了driver广播的的副本，根据公式(7)中可以推出，把属于这个RDD的所有分块右乘，这样就得到了。
4. 更新下一轮迭代的输入矩阵A。考察公式(7)，注意到有

故有

原来的公式可以重写为

这是一个非常重要的优化。这表明分布式矩阵的更新计算实际上并不依赖于，也就是说不需要等到步骤3)全部完成才能计算，它们可以同时进行。计算完成后，将分布式矩阵标志为下一轮迭代的输入矩阵。

1. 重复步骤1)继续迭代, 直到所有的迭代全部完成。

## 4.5 求逆具体实现

由(8)式可知，矩阵的求逆可以转化为对矩阵LU分解的结果，即下三角矩阵L和上三角矩阵U的求逆。LU分解过程已经在上面的小节详细地阐述，因此这一节主要以下三角矩阵为例，讨论三角矩阵的求逆实现。

### 4.5.1 下三角矩阵Inversion具体实现

由(8)式可知，三角矩阵的求逆过程是一个典型的递归过程。它可以归纳为以下几个步骤。

1. 如果输入三角矩阵的规模比较小，可以在单节点运行串行算法直接计算并返回；否则，通过filter方法把输入的三角矩阵划分为、和，其中和都是三角矩阵，每个子矩阵都用一个RDD表示。
2. 对子下三角矩阵和分别递归调用求逆过程，即转向步骤1)，计算得到结果 和。
3. 计算。这是典型的分布式矩阵乘法运算。
4. 使用union函数将 、和这三个RDD合并为一个RDD，并返回。

## 4.6 矩阵乘法

在上面的矩阵分解和求逆实现中，分布式矩阵乘法是运算过程中一个重要的中间步骤。实际上，矩阵乘法运算各元素之间的依赖关系比较低，可以很好地并行。考察参与乘法的分布式矩阵A(m\*k个分块)和分布式矩阵B(k\*n个分块)，它们的最终结果矩阵C（m\*n个分块）中的每一个分块依赖于A中的一行(k个分块)以及B中的一列(k个分块)。单独来看，A中的每一个分块在C中使用了n次，B中的每一个分块在C中使用了m次。这样可以得到下面的乘法运算步骤：

1. 调用flatMap方法，将A中每一个分块复制n份生成一个新的RDD，B中每一个分块复制m份生成一个新的RDD。
2. 对这两个新的RDD以blockID为key，调用join方法。这样结果矩阵C中blockID为(i,j)的分块就获得了它所需要的所有的分块，即A中的第i行分块和B中的第j列分块。
3. 调用mapPartitions方法，将2)中生成的RDD中属于同一个blockID的所有分块做相应乘法，即A中的k个分块与B中的k个分块依次做本地单节点乘法。
4. 调用reduceByKey，把3) 中生成的RDD中属于同一个blockID的所有分块（每个分块是两个分块的乘积）累加，这样就得到了最终结果C。

# 第五章 算法实现中的优化

本章将结合算法的特点和Spark平台的编程框架编程特点，阐述在实现过程中一些可行的优化。

## 5.1 本地高性能线性代数库加速

在本文讨论的三个算法中，每个算法都对原输入矩阵进行了分块，这些分块散落在集群所有节点上。在每一轮迭代中，每个分块往往只需在本地进行代数计算，例如，在矩阵LU分解的第i轮迭代中，标号为(i，i)的分块需要在本地进行单节点的LU分解，其他的节点需要利用广播得到的(i，i)分块计算结果对本地的分块进行更新。在这些步骤中，计算只发生在本节点上，因此可以使用高性能的线性代数库加速本地的计算。

BLAS、LAPACK、MKL等都是非常成熟的高效线性代数运算库，它们针对机器的底层架构做了大量的优化。在本文的实现中，这些线性代数计算直接由JNILoader调用本地的线性代数运算库BLAS实现，而不用JVM直接计算。这一优化可以提高计算的性能，并且本地的矩阵分块阶数越大，性能提升越明显。

## 5.2 使用多个RDD

Spark编程框架下RDD受到其只允许粗粒度转换的限制，在进行频繁迭代更新工作集的一小部分这样的操作时，会带来性能上的不足。如果始终用一个RDD（RDD的一个元素表示矩阵的一个分块）来表示整个分布式矩阵，在计算过程中即使只需要对RDD中的某一个元素更新，却需要在整个RDD上做相应的转换操作。例如，在进行LU分解时，迭代时有一个步骤实际上只需要在未处理的这部分上做转换，而、、此时已经是最终结果的一部分，不需要变动。整个RDD在map时候会根据分块的blockID判断该分块属于、、还是。如果属于前者，则传给map的函数算子什么也不做；如果是后者则做相应的转换。由于每个RDD的转换(transformation)总是生成一个的新的RDD，因此对于、、，Spark仍需要重新生成它们，从而降低了效率。

因此，本文实现时对中间结果分别用不同的RDD表示，迭代完全结束后再将所有的中间结果union成一个RDD，而不是每次迭代后将这些中间结果重新union成为一个表示目前整个分布式矩阵的RDD。实验显示，这样会带来性能上的提高。

# 第六章 实验评估

本章通过一系列实验来阐明矩阵LU分解、Cholesky分解以及求逆运算的性能，注重分析算法的效率以及可拓展性。

## 6.1 实验环境

本文使用的计算集群总共包含13个节点，其中一个作为主控节点，其余作为计算节点。集群节点具体配置见下表。

表1 计算节点配置信息

|  |  |
| --- | --- |
| Item | Information or Setting |
| CPU | 4 Core Intel Xeon 2.4GHz2 |
| Memory | 64 GB |
| Disk | 2 TB SAS2 |
| Network Bandwidth | 1Gbps |
| OS | Red Hat Enterprise Linux Server 6.0 |
| JVM Version | Java 1.6.0 |
| Spark Version | Spark 1.2.0 |
| BLAS Version | BLAS 3.2.1 |

## 6.2 优化效果评估

### 6.2.1 测试调用BLAS优化

首先评估在计算节点调用本地BLAS线性代数运算库的优化效果。实验时共使用了12个计算节点，运行分布式矩阵LU分解，设置输入矩阵的分块大小(blockSize)为2000，这样计算节点本地的矩阵运算量会加大，能更好地显示出加速效果。

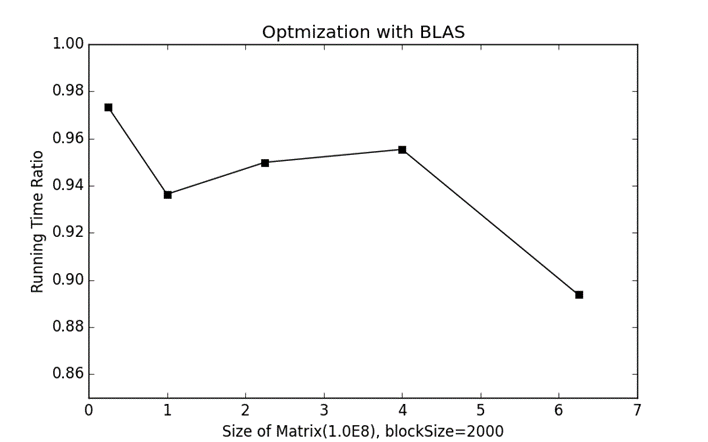


图6 LU分解：调用与不调用BLAS运行时间比

由于BLAS只是对本地矩阵计算加速而并没有带来其他任何额外的开销，因此实验显示出明显的加速效果。从图示中可以看出，在计算节点本地调用了BLAS线性代数运算库后，最多有到达10% 的速度提升。并且可以预见，当输入矩阵的规模更大时，提升的效果更明显。

### 6.2.2 测试使用多个RDD保存中间结果优化

本文实现的第二个优化是使用多个RDD来表示中间计算结果。实验时共使用了12个计算节点，运行分布式矩阵LU分解，设置输入矩阵的分块大小(blockSize)为1000。

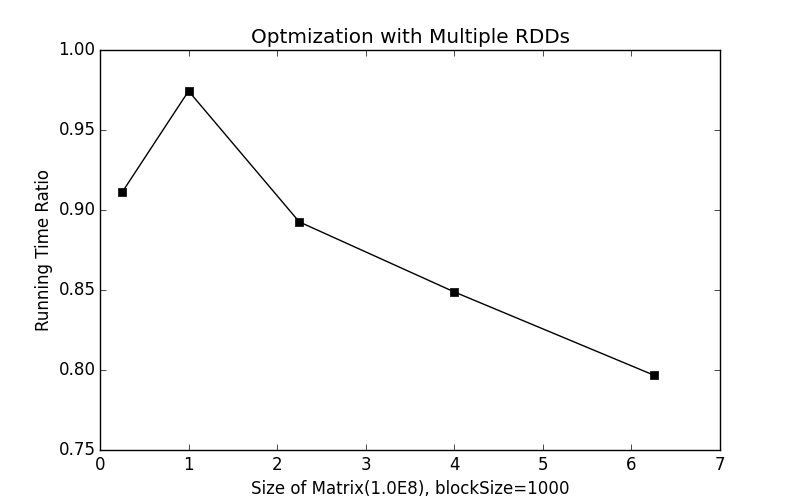


图7 LU分解：合并中间结果和使用多个RDD运行时间比

由图示可以看出，使用多个RDD保存迭代计算过程中的中间结果带来了很好的计算速度提升。当数据规模较大时，缩短了接近20%的计算时间。

## 6.3 可扩展性评估

### 6.3.1 数据可扩展性

为了评估输入矩阵规模增长时计算时间的变化情况，本文在不同的数据规模下测试了三个矩阵操作的实际计算时间。测试时使用了12个计算节点。

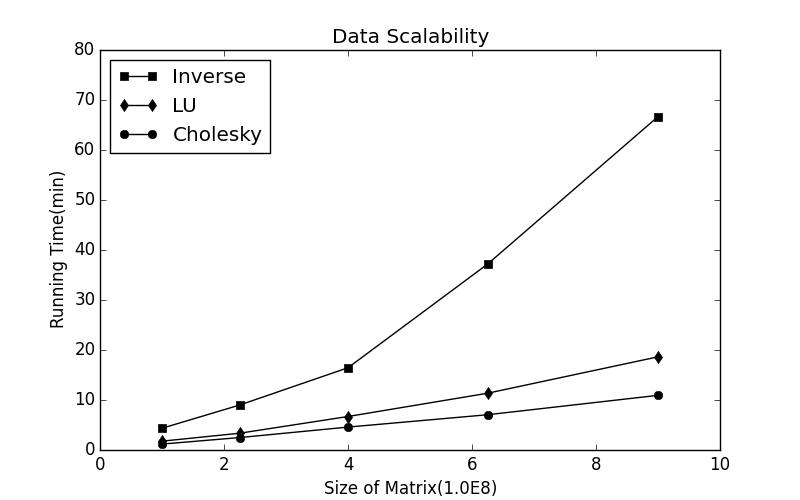


图8：运算时间与数据规模的关系

从图示中可以看到，三个算法都显示出了较好的数据可扩展性，其中LU分解和Cholesky分解的运行时间与输入矩阵的规模成明显的线性关系，求逆的运行时间也几乎与数据规模成线性关系。这表示这三个算法的实现具有较好的数据扩展性。

### 6.3.2 系统可扩展性

为了评估计算节点个数变化时计算时间的变化情况，本文使用不同的计算节点个数测试了三个矩阵操作的实际计算时间。测试时使用了20000x20000的输入矩阵。

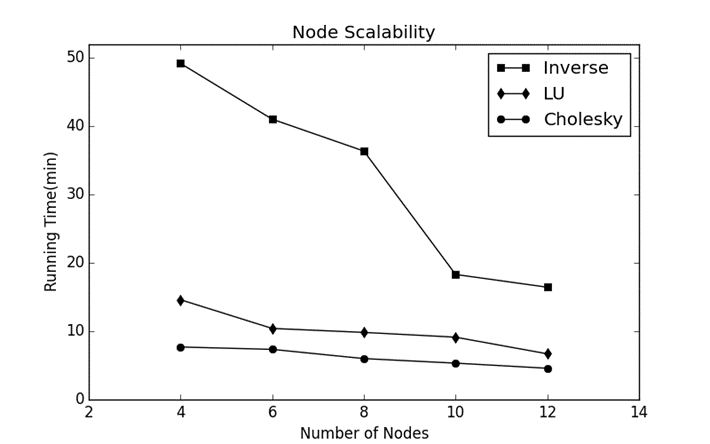


图9：运算时间与节点数量的关系

从图示中可以看到，三个算法都显示出了较好的系统可扩展性，其中LU分解和Cholesky分解的运行时间与计算节点的个数成较明显的线性关系，求逆的运行时间随计算节点个数的不同变化得特别剧烈，这是因为求逆运算中需要进行大量的矩阵乘法，矩阵乘法操作对计算节点的个数更加敏感。

# 第七章 总结与展望

本章首先将对本文工作进行简要的总结，指明本文工作的创新点以及意义所在。同时本章也对现有的工作进行了反思和展望，并给出下一步可能的后续工作。

## 7.1 课题总结

本文结合Spark平台针对大规模分布式矩阵的LU分解、Cholesky分解和求逆给出了具体的算法实现，实验测试结果表明了其具有良好的数据可拓展性和系统可扩展性。本文的实现可以作为很多基于Spark平台的数据计算和分析任务的中间步骤，降低开发者的开发难度，提高计算任务的效率。相对其他类似工作而言，本文选用更加流行的Spark平台，具有更加广泛的适用性，这一点在大数据时代意义更加明显。

## 7.2 进一步的研究工作

矩阵分解和求逆操作需要频繁地迭代“更新”原来的矩阵，但是受到Spark编程框架下RDD只读不可写并且只允许粗粒度的转换(coarse-grained transformation)这一制约，本文基于Spark的实现相比于传统的线性代数并行技术(如ScaLAPACK)有很多效率不高的地方。在下一步工作中我们会更加深入地了解Spark的原理与机制，改善这种情况。

另外，由于分布式计算的特性，相比单节点算法本文实现会在一定程度上损失精度甚至是正确性。例如，本文矩阵LU分解实现中每次迭代都会在一个单节点上对节点本地的矩阵分块进行串行的LU分解，这时候算法就只能针对本地分块进行部分选主元，而无法在整个分布式矩阵全局范围内进行部分选主元(否则会极大地降低计算速度)，这样就造成了精度的下降。同理，分布式操作会打破矩阵操作条件的充分必要性。例如，分布式矩阵可能整体是可逆的，但可能局部节点分配到的分块却是不可逆的，反之亦然。如何在计算速度与精度之间取得较好的平衡，在局部与整体之间保持矩阵性质的一致性，可以在后续工作中继续探讨。

# 参考文献

[1] Bartels R H, Golub G H. The simplex method of linear programming using LU decomposition[J]. Communications of the ACM, 1969, 12(5): 266-268.

[2] Abbasbandy S, Ezzati R, Jafarian A. LU decomposition method for solving fuzzy system of linear equations[J]. Applied Mathematics and Computation, 2006, 172(1): 633-643.

[3] Koren Y, Bell R, Volinsky C. Matrix factorization techniques for recommender systems[J]. Computer, 2009 (8): 30-37.

[4] Shvachko K, Kuang H, Radia S, et al. The hadoop distributed file system[C]//Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on. IEEE, 2010: 1-10.

[5] Chang F, Dean J, Ghemawat S, et al. Bigtable: A distributed storage system for structured data[J]. ACM Transactions on Computer Systems (TOCS), 2008, 26(2): 4.

[6] Shvachko K, Kuang H, Radia S, et al. The hadoop distributed file system[C]//Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on. IEEE, 2010: 1-10.

[7] Lawson C L, Hanson R J, Kincaid D R, et al. Basic linear algebra subprograms for Fortran usage[J]. ACM Transactions on Mathematical Software (TOMS), 1979, 5(3): 308-323.

[8] Anderson E, Bai Z, Dongarra J, et al. LAPACK: A portable linear algebra library for high-performance computers[C]//Proceedings of the 1990 ACM/IEEE conference on Supercomputing. IEEE Computer Society Press, 1990: 2-11.

[9] Blackford L S, Choi J, Cleary A, et al. ScaLAPACK users' guide[M]. siam, 1997.

[10] Agullo E, Augonnet C, Dongarra J, et al. LU factorization for accelerator-based systems[C]//Computer Systems and Applications (AICCSA), 2011 9th IEEE/ACS International Conference on. IEEE, 2011: 217-224.

[11] Xiang J, Meng H, Aboulnaga A. Scalable matrix inversion using MapReduce[C]//Proceedings of the 23rd international symposium on High-performance parallel and distributed computing. ACM, 2014: 177-190.

[12] Li B, Tata S, Sismanis Y. Sparkler: Supporting large-scale matrix factorization[C]//Proceedings of the 16th International Conference on Extending Database Technology. ACM, 2013: 625-636.

[13] Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing[C]//Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012: 2-2.

[14] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.

[15] Chowdhury M, Zaharia M, Stoica I. Performance and Scalability of Broadcast in Spark[J].

[16] 黄宜华，苗凯翔. 深入理解大数据:大数据处理与编程实践[J]. 2014.

# 致谢

在毕业设计过程中，我遇到了许多的挑战，但是在导师的指导下，在学长和同学的带领和帮助下，我解决了一个又一个问题，最终成功完成了这篇代表我四年学习成果的本科毕业设计。

在此我首先要感谢我的毕业设计指导导师黄宜华教授，他在毕业设计的选题、进度管理、答辩演示以及最后的毕业论文撰写都一直以严谨认真的态度引导我和要求我，没有这些指导我无法完成我的毕业设计。其次我要感谢顾荣学长和唐云学长。顾荣学长在大的框架设计思想方面给了我很多具体的指导和帮助，让我受益良多。唐云学长在Spark平台的架构以及应用编程等问题上给了我很大的帮助，耐心地回答了我很多具体的细节问题，使我对Spark平台的整体认识有了极大地提高。在我撰写毕业论文的过程中，两位学长同样给了我很多有价值建议。同时我也要感谢南京大学计算机科学与技术系PASALab实验组提供了计算集群，使我能够验证算法的实际性能。

最后我要感谢完成本篇论文所引用参考资料的所有专家和学者，没有他们已有的这些研究成果这篇论文将难以完成。