**中图分类号：TP302.1 单位代号：**

**密 级： 学 号：**

**硕 士 学 位 论 文**

**SHANGHAI UNIVERSITY**

**MASTER’S DISSERTATION**

|  |  |
| --- | --- |
| **题**  **目** | **分层并行设计模式及分布式ADMM算法的研究** |

**作 者 方 铃**

**学科专业 计算机应用技术**

**导 师 雷咏梅**

**完成日期 2017年02月**

上海大学

本论文经答辩委员会全体委员审查,确认符合上海大学硕士学位论文质量要求。

答辩委员会签名：

主任：

委员：

导 师：

答辩日期：

**原 创 性 声 明**

本人声明：所呈交的论文是本人在导师指导下进行的研究工作。除了文中特别加以标注和致谢的地方外，论文中不包含其他人已发表或撰写过的研究成果。参与同一工作的其他同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

签 名： 日 期：

**本论文使用授权说明**

本人完全了解上海大学有关保留、使用学位论文的规定，即：学校有权保留论文及送交论文复印件，允许论文被查阅和借阅；学校可以公布论文的全部或部分内容。

（**保密的论文在解密后应遵守此规定**）

签 名： 导师签名： 日期：

上海大学工学硕士学位论文

**分层并行设计模式及分布式ADMM算法的研究**

姓 名： 方 铃

导 师： 雷咏梅

学科专业： 计算机系统结构

上海大学计算机工程与科学学院

二零一七年二月

A Dissertation Submitted to Shanghai University for the Degree of Master in Engineering

**Hierarchical Parallel Model and Research of the Distributed ADMM Algorithm**

MA Candidate：Ling Fang

Supervisor：Yongmei Lei

Major：Computer Architecture

**School of Computer Engineering and Science**

**Shanghai University**

**February, 2017**

摘 要

很多分布式机器学习算法一般情况下可以归结成全局变量一致性优化的问题，如线性回归、Logistic回归、支持向量机等，而这些问题可以有效的通过交替方向乘子法（Alternating Direction Method of Multipliers，ADMM）来解决。ADMM算法是一种解决可分解凸优化问题的简单方法，尤其在解决大规模问题上卓有成效，利用ADMM算法可以将原问题的目标函数等价的分解成若干个可求解的子问题，然后并行求解每一个子问题，最后协调子问题的解得到原问题的全局解。

同步分布式ADMM算法在每次迭代过程中需要进行全局数据的交换，这样会带来较大的同步开销以及通信时间，同时算法的可扩展性也受到一定的限制。本文从通信优化的角度出发，提出两种改进的方法，一是提出一种基于分层并行设计模式的异步分布式ADMM算法；二是设计一种基于SSP（Stale Synchronous Parallel）模型的异步分布式ADMM算法，旨在设计出高效高扩展的分布式ADMM算法。

本文的具体工作集中在以下几点：

（1）提出一种分层并行设计模式，一方面通过多核感知的MPICH中间件去构建层次化通信模型，再利用有效的通信优化策略提高通信效率，另一方面提供数据预处理的方法减少数据分发占用的大量带宽。

（2）通过矩阵乘法和ELM(极限学习机)两个实例来介绍分层设计模式中层次化通信模型的构建、通信优化策略，以及数据划分以及预处理的方法，并且通过实验验证两种基于分层设计模式的并行算法的高效性与高可扩展性。

（3）提出了一种动态的层次化通信模型-层次蝶式通信模型应用到异步分布式ADMM。层次蝶式通信模型首先在网络通信上区分了节点内通信和节点间通信，其次节点间通信采用了蝶形通信模型，层次化的通信模型能大大提高并行算法的可扩展性，节点间的蝶形通信模型能够减少全局通信时间，并且在理论上分析了应用三种通信模型的算法的收敛性。

（4）提出了一种基于SSP模型的异步分布式ADMM算法，利用开源Petuum框架数据并行的SSP计算模型设计一种异步分布式ADMM算法来减少同步ADMM算法每次迭代中的同步开销，并且给予理论收敛性的证明和实验的验证。

**关键词**：机器学习；MPICH3；层次化通信模型；异步；Petuum；SSP模型

ABSTRACT

目 录

[摘 要 I](#_Toc470900913)

[ABSTRACT III](#_Toc470900914)

[目 录 IV](#_Toc470900915)

[**论文图表目录** VII](#_Toc470900916)

[第一章 绪论 1](#_Toc470900917)

[1.1 课题来源 1](#_Toc470900918)

[1.2 课题研究的目的和意义 1](#_Toc470900919)

[1.3 国内外研究现状 2](#_Toc470900920)

[1.3.1 分布式ADMM算法的研究现状 2](#_Toc470900921)

[1.3.2 并行计算模型的研究现状 3](#_Toc470900922)

[1.3.3 MPI的发展和通信优化 4](#_Toc470900923)

[1.3.4 分布式并行框架的发展现状 5](#_Toc470900924)

[1.4 论文主要研究内容 7](#_Toc470900925)

[第二章 分层并行设计模式 8](#_Toc470900926)

[2.1 进程分布对并行算法的影响 8](#_Toc470900927)

[2.2 多核感知的MPICH中间件 10](#_Toc470900928)

[2.2.1 MPICH进程分配策略 10](#_Toc470900929)

[2.2.2 中间件的设计方法 12](#_Toc470900930)

[2.2.3 中间件的API接口 12](#_Toc470900931)

[2.3 层次化通信模型 13](#_Toc470900932)

[2.4 分层并行编程模型 14](#_Toc470900933)

[2.4.1 基于主从模式的分层并行编程模型 14](#_Toc470900934)

[2.4.2 基于对等模式的分层并行编程模型 16](#_Toc470900935)

[2.5 通信优化策略 17](#_Toc470900936)

[2.6 源数据输入优化 17](#_Toc470900937)

[2.6.1 数据输入本地化 17](#_Toc470900938)

[2.6.2 结合HDFS的数据本地化 18](#_Toc470900939)

[2.7 应用实例 18](#_Toc470900940)

[2.7.1 矩阵乘法并行化 18](#_Toc470900941)

[2.7.2 分布式并行ELM算法 21](#_Toc470900942)

[2.8 本章小结 32](#_Toc470900943)

[第三章 基于分层并行设计模式的异步分布式ADMM算法 34](#_Toc470900944)

[3.1 分散一致性分布式ADMM算法 34](#_Toc470900945)

[3.2 异步分布式ADMM算法 36](#_Toc470900946)

[3.3 基于分层并行设计模式的ADMM算法的设计 38](#_Toc470900947)

[3.3.1 三种层次化模型的设计 38](#_Toc470900948)

[3.3.2 算法的描述与分析 40](#_Toc470900949)

[3.4 收敛性分析 42](#_Toc470900950)

[3.4.1 静态通信模型场景下算法收敛性分析 42](#_Toc470900951)

[3.4.2 动态通信模型场景下算法收敛性分析 43](#_Toc470900952)

[3.5 实验结果与分析 44](#_Toc470900953)

[3.5.1 实验环境 44](#_Toc470900954)

[3.5.2 收敛性测试与分析 45](#_Toc470900955)

[3.5.3 性能测试与分析 46](#_Toc470900956)

[3.6 本章小结 47](#_Toc470900957)

[第四章 基于Petuum的异步分布式ADMM算法 48](#_Toc470900958)

[4.1 SSP模型详解 48](#_Toc470900959)

[4.2 全局一致性分布式ADMM算法 49](#_Toc470900960)

[4.3 基于SSP模型的异步分布式ADMM算法 52](#_Toc470900961)

[4.3.1 算法的设计与分析 52](#_Toc470900962)

[4.3.2 一致性变量的存储 54](#_Toc470900963)

[4.4 收敛性分析 54](#_Toc470900964)

[4.5 本章小结 62](#_Toc470900965)

[第五章 实验 63](#_Toc470900966)

[5.1 实验测试环境 63](#_Toc470900967)

[5.2 实验应用与实验数据 64](#_Toc470900968)

[5.3 基于SSP模型的异步分布式ADMM算法收敛性测试 64](#_Toc470900969)

[5.3.1 目标函数为凸和强凸时算法收敛性测试 64](#_Toc470900970)

[5.3.2 目标函数为非凸时算法收敛性测试 65](#_Toc470900971)

[5.3.3 收敛性测试结果与分析 66](#_Toc470900972)

[5.4 基于SSP模型的异步分布式ADMM算法性能测试 66](#_Toc470900973)

[5.4.1 算法通信时间的测试 66](#_Toc470900974)

[5.4.2 算法精度测试 67](#_Toc470900975)

[5.4.3 性能测试结果与分析 68](#_Toc470900976)

[5.5 基于Petuum和基于MPI的ADMM算法对比实验 69](#_Toc470900977)

[5.6 本章小结 69](#_Toc470900978)

[第六章 结论和展望 71](#_Toc470900979)

[6.1 总结 71](#_Toc470900980)

[6.2 进一步研究方向 71](#_Toc470900981)

[参考文献 73](#_Toc470900982)

[作者在攻读硕士学位期间公开发表的论文 79](#_Toc470900983)

[作者在攻读硕士学位期间所作的项目 80](#_Toc470900984)

[致 谢 81](#_Toc470900985)

**论文图表目录**

[图 2.1‑1 二项树算法在不同进程分布下的性能 9](#_Toc470900872)

[图 2.1‑2 二项树算法在不同的进程分布下通信过程 9](#_Toc470900873)

[图 2.3‑1 层次通信模型 14](#_Toc470900874)

[图 2.7‑1 分行算法矩阵数据划分图 19](#_Toc470900875)

[图 2.7‑2 矩阵分行并行算法在两种设计模式下的测试结果 21](#_Toc470900876)

[图 2.7‑3 基于主从模式并行ELM算法流程 24](#_Toc470900877)

[图 2.7‑4 基于分层并行设计模式并行ELM算法流程 26](#_Toc470900878)

[图 2.7‑5 MV和Friedman数据集测试结果 30](#_Toc470900879)

[图 2.7‑6 Covertype数据集测试结果 31](#_Toc470900880)

[图 2.7‑7 通信时间和计算时间所占比例 32](#_Toc470900881)

[图 3.3‑1 Inter-Master环形 39](#_Toc470900882)

[图 3.3‑2 Inter-Master全连接 39](#_Toc470900883)

[图 3.3‑3 Inter-Master蝶形 39](#_Toc470900884)

[图 3.3‑4 一次迭代时序关系图 41](#_Toc470900885)

[图 3.4‑1 本地进程数和计算节点数和的关系 43](#_Toc470900886)

[图 3.5‑1 静态网络下计算节点数与进程数算法收敛性关系 45](#_Toc470900887)

[图 3.5‑2 蝶形网络下计算节点数与进程数算法收敛性和的关系 45](#_Toc470900888)

[图 3.5‑3 FC、IMFC、IMC、IMB四种模型下在本地进程 46](#_Toc470900889)

[图 3.5‑4 FC、IMFC、IMC、IMB四种模型下在本地 46](#_Toc470900890)

[图 4.1‑1 SSP模型 48](#_Toc470900891)

[图 5.3‑1 目标函数为凸和强凸时算法的收敛性测试结果 65](#_Toc470900892)

[图 5.3‑2 目标函数为凸和强凸时算法的收敛性测试结果 65](#_Toc470900893)

[图 5.3‑3 标函数为非凸时算法的收敛性测试结果 66](#_Toc470900894)

[图 5.4‑1 基于SSP模型的异步分布式ADMM算法性能测试结果 67](#_Toc470900895)

[图 5.4‑2 基于SSP模型的异步分布式ADMM算法RMSE测试结果 68](#_Toc470900896)

[表格 2.7‑1 PC机群配置 29](#_Toc470900897)

[表格 2.7‑2 Information of the regression data sets 30](#_Toc470900898)

[表格 2.7‑3 RMSE in ELM and ELM-Parallel-MS and ELM-Parallel-HM 30](#_Toc470900899)

[表格 2.7‑4 Information of the classification data sets 31](#_Toc470900900)

[表格 2.7‑5 Success Rate in ELM and ELM-Parallel-MS and ELM-Parallel-HM 32](#_Toc470900901)

[表格 3.5‑1 FC、IMFC、IMC、IMB四种模型下HD-ADMM算法总时间 47](#_Toc470900902)

[表格 3.5‑2 FC、IMFC、IMC、IMB四种模型下HD-ADMM算法通信时间 47](#_Toc470900903)

[表格 5.5‑1 基于Petuum和基于MPI的ADMM算法对比测试结果 69](#_Toc470900904)

# 绪论

## 课题来源

本课题来源于国家863计划子课题、上海市科研计划重大项目：新概念高效能计算机体系结构及系统研究开发，项目编号：2009AA012201、08dz501600。

## 课题研究的目的和意义

随着高性能计算需求的日益增长，多核处理器在高性能计算中得到了广泛的普及并且多核PC机已经可以完成许多大规模计算任务，为了保证高性能计算机系统的效率，需要保持计算和通信的平衡性，多核的广泛使用对通信系统的效率以及并行算法提出了更高的要求。

Message Passing Interface简称MPI【1】，是一种消息传递接口的标准，用于开发基于消息传递的并行程序，作为MPI实现版本之一的MPICH是目前最重要的基于消息传递模型的并行编程工具，它具有移植性好、功能强大、效率高等优点，并且可以通过算法以及编程的优化带来较高的通信效率。但在大数据的背景下，由于MPI缺乏分布式文件系统的支持和本身消息传递模型的限制【2】，使得其难以应对大规模的数据集的管理和处理，另外，在多核环境下由于节点内和节点间的通信差异性的存在，导致严重的进程非平衡到达的问题【3】。

机器学习和数据挖掘是人们探索和理解海量数据的有力途径，但是随着大数据时代的到来，仅仅依靠单机有限存储和处理速度来处理海量数据是不可能的，因此如何分布式并行化机器学习算法也成为目前研究的热点。绝大数分布式机器学习算法所要解决的最优化问题都可以归结为全局变量一致性优化的问题，形式如下：

|  |  |
| --- | --- |
|  | （1.2-1） |

其中是本地的目标函数，本地模型参数，表示全局一致性参数。而这个问题可以有效的通过ADMM算法解决，ADMM算法可以将大部分机器学习所要求解的凸优化问题分解成可求解的子问题，然后并行求解每一个子问题，最后通过协调子问题而得到原问题的全局解。同步分布式ADMM在一次迭代过程中需要一次全局数据交换来求解一致性参数，对应需要进行一次MPI的AllReduce操作【4】，在复杂的集群环境下，如当某个节点温度过热而导致CPU的计算能力下降，对于同步模型来说，计算速度快的节点需要花大量时间去等待慢节点，这会带来较大的通信开销以及同步所需要的额外开销。

Petuum是一个分布式[机器学习](http://www.yidianzixun.com/home?page=channel&keyword=%E6%9C%BA%E5%99%A8%E5%AD%A6%E4%B9%A0)专用计算框架【5】，其中针对数据并行的SSP模型是利用机器学习算法的容错性构建的一种并行计算模型【6】，机器学习的容错性是算法迭代计算过程中可以存在一定的误差。例如随机梯度下降算法，可以在每一次迭代过程中容许存在一定的噪音梯度，而前提是不影响算法的收敛性。在SSP模型中，每一个线程在有限时间窗口里做局部的运算，用参数值的局部版本做运算，由于每个线程的计算速度不一致，当线程超出了这一时间窗口，那么该线程就需要去等待较慢的线程。SSP模型中采用半同步的方式可以有效的减少同步模型中的同步开销。在Petuum上实现基于SSP模型的分布式ADMM算法是可以减少同步分布式ADMM算法的同步开销。

本文的研究重点主要是如何改善基于MPI的并行算法通信问题以及数据存储的问题、减少同步分布式ADMM算法的同步开销和通信开销。本文的研究意义是提出一种分层并行设计模式，主要是通过多核感知的MPICH中间件为并行算法构建高效的层次化通信模型和通信优化策略来提高通信效率，以及数据预处理方法可以利用机群中的大量数据存储节点同时访问数据，以此利用分布集群中大量节点上的磁盘集合提供高带宽的数据访问和传输，另外将这种分层并行设计模式与异步分布式ADMM算法相结合以及设计了一种基于SSP模型的异步分布式ADMM算法来减少同步分布式ADMM算法的同步开销和通信开销。

## 国内外研究现状

### 分布式ADMM算法的研究现状

分布式ADMM算法从并行角度来说可以分成同步分布式ADMM算法和异步的ADMM算法， 从算法形式上可以分成全局一致性ADMM【4】和分散一致性ADMM【7】【8】。全局一致性ADMM和分散一致性ADMM的不同点是，前者是主从模式结构，而后者是对等模式结构。Boyd给出了在BSP【9】、MPI【1】、MapReduce【10】上ADMM的实现方式【4】。Lubell-Doughtie在MapReduce模型上通过ADMM解决带L2正则的Logistic Regression【11】。但由于网络延迟和同步所需要的额外开销，MapReduce并不太适合ADMM这种需要在数据上迭代多次的算法。

无论全局一致性ADMM还是分散一致性ADMM在同步模型上一次迭代过程中需要一次全局数据交换，这会带来较大的通信开销以及同步所需要的额外开销。R. Zhang 以及T.-H. Chang等人在全局一致性ADMM基础上设计出异步分布式ADMM【12】【13】，其思想和旧同步模型【6】相似，Wei E等人在分散一致性ADMM基础给出异步分布式ADMM算法【14】，这两种异步分布式ADMM算法是子线性收敛率，其中是迭代次数。

### 并行计算模型的研究现状

模型是对现实存在的事物或时间的一种抽象概述。并行计算模型是指以预测应用程序运行结果为目的而对并行计算机的硬件和软件资源、计算过程进行的一种抽象。并行计算模型是并行算法的设计与分析的基础，它屏蔽掉各个并行计算机的具体差异，从具体的并行计算机中抽取若干个能反映计算特性的、为数不多的可计算或可测量的一组机器参数【15】。在具体的并行机中，算法设计者可以按照模型所定义的计算行为构造出某算法在该并行机上执行的成本函数，函数中的机器及运行时参数可以通过实验或测试工具测量得到，然后根据该成本函数对算法的效率进行分析，从而对算法设计或编程实现起到指导作用。

从存储系统的角度进行分类，传统的并行计算模型可分为共享存储的并行计算模型、分布存储的并行计算模型、层次存储的并行计算模型【16】。常见的并行计算模型有PRAM模型【17】、APRAM模型【18】、SIMD模型【19】、BSP模型【9】、LogP模型【20】、Memory-LogP模型【21】、DRAM（h）模型【22】、HPM模型【23】、NHBL模型【24】等。

并行计算拥有众多的计算模型，但是却不存在一个统一的、高度抽象、可概括所有计算的模型，例如串行计算中的RAM模型。

RAM模型【25】以冯·诺依曼结构为原型进行简化，将一台串行计算机抽象为一个指令执行部件和一个容量无上限的存储设备，其它如输入输出等设备被忽略，存储设备用于存储程序运行所需要的指令和数据，指令执行部件可在任何单位时间按照顺序原则（若遇到转移指令则进行转移读取）访问存储设备进行指令和相应所需数据的读取，并执行该指令。同时RAM模型以时间复杂度和空间复杂度作为度量算法性能的准则。

并行计算机未形成统一的计算模型的原因主要有两点【26】：第一并行计算机的性能指标众多，除了串行计算机所用的时间复杂度和空间复杂度外，还包括加速比、效率、可扩展性等；第二并行计算机体系结构复杂且多样，单从结构模型的角度就可分为单指令多数据流计算机、并行向量计算机、对称多处理机、大规模并行处理机、工作站机群、分布式共享存储六种；从访存模型角度又可分为均匀存储访问、非均匀存储访问、全高速缓存存储访问、高速缓存一致性非均匀存储访问和非远程存储访问五种。若用统一的计算模型来表示并行计算，则该计算模型必然是极其复杂的，包含了诸多参数。但是，参数过多又与模型的简洁性相矛盾，必将提高模型使用的难度。

分层并行模型【27】是近几年提出的，它将并行计算模型分为三个层次：并行算法设计模型、并行程序设计模型和并行程序执行模型。并行算法设计模型从算法的角度，将不同的并行机抽象为一种通用虚拟并行机，算法设计者在其上设计和分析并行算法；并行程序设计模型，从编程的角度指导程序员，按照程序的执行流程，选用某种并行语言，正确编程实现某并行算法；并行程序执行模型，从性能效率的角度指导程序运行者，将不同的并行语言实现的程序，在具体的并行机上编译和优化运行。

SSP模型【6】是Petuum框架【5】针对数据并行的分布式机器学习算法提出了一种并行计算模型，该模型结合机器学习算法的迭代性和容错性，通过时间窗口参数staleness控制分布式机器学习算法异步执行，减少了同步模型中大量的同步开销，同时有理论收敛性的保证。

### MPI的发展和通信优化

MPI是一种消息传递标准，多个国际组织或厂商遵循这个标准推出了各自的实现版本，各个软件版本各有特点。MPI软件包括运行库、编译环境以及任务管理工具等，其中运行库是该软件的核心。

MPI的实现版本主要分为两类，MPICH及其衍生版本与Open MPI【28】【29】【30】。其中，MPICH拥有最多的用户，MPICH针对不同的通信库衍生出多种实现版本。如Cray MPI【31】、Intel MPI【32】、MVAPICH2【33】、IBM MPI【34】等。Open MPI则是糅合了LA-MPI、FT-MPI、LAM/MPI和PACX-MPI等4个MPI版本【35】，它提供免费高效的开源MPI通信库实现，OpenMPI比MPICH具有更好的可扩展性。

在MPI系统层的研究中，通信函数的实现必须考虑底层硬件特点和物理网络结构才能发挥最佳的性能。优化通信函数一直是MPI研究的重点内容，具体包括点对点通信的优化和集合通信的优化。

在MPI中，点对点的通信是指两个进程之间发送和接收消息，主要分为节点间的点对点通信和节点内的点对点通信。点对点通信优化的目标是提高进程间传输数据的效率，在MPICH3中的Nemesis【36】通信模块已经对节点内和节点间的通信进行了区分和优化【37】【38】。提高节点间的通信效率主要方法是兼容高速的网络设备，如MVAPICH系统采用InfiniBand互联技术具有高速稳定的网络性能【39】；西北工业大学设计针对先锋光纤通道交换网的MPI通信库【40】【41】。节点内点对点的通信优化主要是通过共享内存的通信机制代替了网络接口卡环路的机制【42】。同一个节点内的两个进程共享一块内存区域，发送进程与接收进程通过共享区域传递数据相比基于网络接口卡的通信机制有更快的传输速度。

集合通信的优化一直是国内外高性能计算研究的热点内容，一些研究根据特定的网络拓扑优化集合通信，进程间通信性能与节点分布的物理位置密切相关，任何一种调度算法都不能满足所有异构的通信环境【43】【44】。因此获知准确的网络拓扑结构并做出针对性的优化十分重要。随着多核处理器的普及，多核环境下集合通信的优化得到广泛的关注。张攀勇等人研究多核处理器共享缓存和内存竞争的特点提出了层次化优化算法【45】。闫浩在工作站集群环境下对集合通信进行了分析和优化【46】。

### 分布式并行框架的发展现状

Hadoop【47】是一个专门用于处理大数据的分布式处理框架，并且其生态环境内包括一个用于数据管理的分布式文件系统HDFS及一个编程模型MapReduce【10】。Hadoop使用易于掌握，用户在不了解底层分布式细节的情况下也可以很容易的进行分布式应用开发。Hadoop使用传统的网络通信协议进行数据通信，比如TCP/IP、RPC和HTTP【48】。与MPI相比，Hadoop模型通信和计算的并行度低，而且Hadoop通信性能低于MPI。

Phoenix【49】【50】是共享内存的体系结构上的MapReduce模型【10】的实现，主要用于数据密集型任务的处理，它的目标是适合于共享内存的多处理器计算机以及多核计算机，使程序执行得更高效，而且使程序员不必关心并发的管理。Phoenix使用线程创建并行化的Map和Reduce任务，通过任务调度器给可用的处理器动态调度任务，实现负载均衡和最大化吞吐量，并通过调整任务粒度和并行任务的分配来进行局部管理。

Spark【51】【52】是由UC Berkeley AMPLab开发的一种基于MapReduce模型【10】的分布式计算框架，它不仅支持内存中数据集的保存和恢复，而且Spark启用了内存分布数据集，除了能够提供交互式查询外，还可以优化迭代工作的负载，实现MapReduce的迭代计算。另外，Spark作为一种批处理的交互式实时处理类型，对外提供丰富的Java、Python等API扩展了其应用开发范围。

Storm【53】【54】是一个分布式实时计算系统，对外提供了一系列用于批处理的组件，可有效的应用于信息流处理、连续计算和分布式远程过程调用。Storm结构简单，可以兼容多种开发语言，并且其集群类似于Hadoop集群。不同之处在于Hadoop集群运行MapReduce Job，而Storm运行Topologies。Job和Topologies存在的差异主要是，MapReduce Job最终会被完成，而Topologies进程将一直运行。

Petuum 是一个分布式机器学习平台【5】，用以处理两类大规模机器学习问题： （1）大数据（数据样本数目庞大）；（2）大模型（模型参数众多）。不同于一般用途的分布式编程平台（如数据流系统），Petuum专门为机器学习而设计。充分利用机器学习算法的特征，如迭代性，容错性，参数收敛的不均匀性等等，以最大化系统性能。Petuum主要包括两个组件：（1）基于受限异步一致性协议的分布式键值存储系统Bösen；（2）动态参数更新调度器Strads。

## 论文主要研究内容

本文的研究内容主要分为三个部分进行讲述，第一部分是提出分层并行设计模式，并且通过矩阵乘法和ELM算法两个实例进行验证测试以及性能分析；第二部分是通过分层并行设计模式构建的特殊的层次化通信模型应用到异步分布式ADMM算法上，并且给予收敛性的分析；第三部分是在Petuum分布式框架上实现基于SSP模型的异步分布式的ADMM算法，同时进行理论收敛性的证明。

本文的结构如下所述：

第一章，简要介绍了论文的研究的目的与意义、相关技术的研究现状以及论文的主要研究内容。

第二章，介绍了分层并行设计模式的相关技术细节，主要包括多核感知的MPI中间件的设计、层次化通信模型、分层编程模型、通信优化的策略、源数据输入优化，同时通过矩阵乘法和ELM算法两个实验进行验证测试以及详细的性能分析。

第三章，提出了一种基于分层设计模式的异步分布式ADMM算法，利用分层设计模式的层次化蝶形通信模型应用到异步分布式ADMM算法中，并且还另外比较了层次全连接和环形通信模型，在理论上分析了应用三种通信模型的算法的收敛性，并通过实验验证了层次蝶式通信模型的异步ADMM算法的高效性和高可扩展性。

第四章，介绍Petuum分布式框架和SSP并行计算模型，并介绍了基于SSP模型的异步分布式ADMM算法设计和实现方法，以及理论收敛性的分析。

第五章，实验测试部分包括基于SSP模型的异步分布式ADMM算法收敛性测试和性能测试，收敛性测试针对凸和非凸的目标函数进行测试来验证理论收敛性分析的结论，性能测试主是对算法的通信时间和算法的精度进行测试来验证算法的高效性，最后对基于MPI的分布式ADMM算法和基于Petuum的分布式ADMM算法进行实验对比分析。

第六章，总结全文的工作，概括论文的理论内容和实验结论，指出本文的不足之处并给出下一步研究的方向和期望。

# 分层并行设计模式

本章首先介绍简单的二项树算法在不同进程分布场景下性能的差异性，从而引出基于MPI并行算法在大数据背景下的通信与数据存储问题，进而提出一种新的并行算法设计模式-分层并行设计模式。本章详细介绍分层并行设计模式的技术细节，并通过矩阵乘法和ELM算法(极限学习机)进行验证测试和详细的性能分析。

## 进程分布对并行算法的影响

MPI缺乏分布式文件系统的支持和本身消息传递模型的限制【2】，使得其难以应对大规模的数据集的管理和处理，基于MPI并行算法，第一，在面对大规模数据集，数据分发加大了通信时间，从而增加算法的整体运行时间；第二，由于单机内存的限制，无法开辟大量的空间来保存大规模的训练样本。

除了上面两个问题外，在传统的设计模式下，并行程序开发者只关注如何设计算法的通信拓扑结构，不会关心进程在节点分布情况，但在多核环境下，由于存在节点内和节点间的通信差异性，使得同一并行程序在不同进程分布情况下存在性能的差异。本文通过一个实例来进行说明，由5台四核CPU的PC组成的多核机群环境下对二项树求和算法进行测试，specify1和specify4分别表示每个节点预分配进程数1个和4个，总分配总程数20个。从图 2.1‑2中可以看出在两种不同的进程分布情况下二项树算法的执行时间存在差异性。如图4是二项树算法在三个节点两种不同进程分布情况下的通信过程。



图 2.1‑1 二项树算法在不同进程分布下的性能



图 2.1‑2 二项树算法在不同的进程分布下通信过程

图 2.1‑3(a)是sepcify1情况下的通信过程， 图 2.1‑3(b)是sepcify4下的通信过程（颜色相同的进程是在同一个节点上），假设两种通信过程的算法运行时间分别为、，节点间通信时间为,节点内通信时间为，那么根据图中的通信过程可以求得：

由于，则。当节点数为偶数时，两种进程分布的通信时间是相同的，当节点数为奇数时，两种进程分布的通信时间不相同。从上面的实例中可以看出，一般性的集合通信算法二项树算法在不同进程分布中通信时间可能会存在差异，所以很多的研究人员对MPI集合通信算法进行优化【55】【56】【57】【58】，能有效的减小通信差异性导致性能的问题，但不能针对所有的应用，例如分布式并行支持向量回归机算法【59】，在级联通信拓扑结构下不能通过集合通信来解决合并问题，因为两两合并会产生新的训练集，新训练集再通过训练产生新的支持向量。基于此，本文提出一种分层并行设计模式，在处理大规模数据时在数据预处理阶段提供数据划分的方法来优化通信问题和存储问题，另外，减小并行算法由于进程分布不同所带来的影响。本章会详细介绍分层并行设计模式依赖的MPICH中间件、层次通信模型的结构、分层并行设计模式的编程模型，数据预处理方法和通信优化策略。

## 多核感知的MPICH中间件

### MPICH进程分配策略

MPICH3.0的Hydra模块构建了进程管理框架，MPI进程需要经常与该框架进行交互，其交互的语言就是PMI标准，即进程管理接口（Process Management Interface）。Hydra由一个运行在服务器上的作业加载程序JLE（Job Launching Executable）和多个运行在不同计算节点上的进程管理代理PMP（Process Manager Proxy）组成。之所以称为PMP是因为它基本上具备了PM（Proxy Manager）的所有功能，代理PM并运行在计算节点上负责具体维护本节点上所有进程的状态。换言之，包括MPI进程在内系统进程拓扑是一个倒树状结构，JLE处于树根，负责在各计算节点上创建进程PMP；PMP在本节点上创建一个或多个MPI进程。JLE对应的执行程序就是我们经常使用的作业加载程序mpiexec.hydra；PMP为hydra\_pmi\_proxy。

MPICH3.0进程分配工作是由PMP和PM负责的，采用的是round robin算法，当hydra启动时，首先读取用户启动的参数，然后解析参数，创建node链表和exec链表，node链表保存了节点及预分配进程数等信息，exec链表保存了客户端进程信息，根据这两个链表产生为每一个node生成一个proxy启动代理链表，主要保存每一个节点分配的进程名、进程数、启动参数等信息。

具体算法是：

第一步：读取要启动进程的总数目np，并保存在exec链表中；

第二步：读取节点列表文件，创建node链表，并把每个节点预分配进程数保存在node链表中；

第三步：为每一个节点创建proxy与之关联；

第四步：遍历node链表，根据node中的预分配进程数n和总的进程数np进行分配，规则如下：如果n > = np 则把所有np分配该node,并结束分配过程；如果 n < np,则给该node分配n个进程，并继续遍历，无论哪种情况都需要创建一个exec保存分配数np,并把该exec挂载到proxy的exec链表后；

第五步：遍历过程中，如果在未遍历完node已经分配完，就结束整个分配过程，如果遍历完整链表后还没有分配完，则跳到第四步继续执行，直到分配完成。

启动进程的方式是由hydra的作业加载程序在每一个节点产生一个PMP，再将对应node上的proxy信息发送给PMP，PMP再去根据该节点进程分配情况去启动MPI客户端进程，但是对于启动后的每一个应用进程需要根据一个三维的向量组信息在自己的进程空间重构进程分布，具体的向量结构如下：

其中， 表示node id的起始值， 表示节点个数， 节点分配的进程数。

具体的重构算法如下：

第一步：从三维向量组的第一个向量开始进行遍历

第二步：获得三维向量中的节点个数（第二项，一个向量可能包含多个节点信息，因为一旦节点信息一样时就会合并两向量），遍历每个节点。

第三步：将向量中的node id起始值与节点个数相加（节点分配的进程数）次，分别得到的值就是进程所在的节点，过程中进程计数器每次加1，

第四步：直到进程计数器与总进程数相等时，算法结束。

从上面的分析可以看出进程是按类似于循环分配的，而对于很多并行算法来说，并没有考虑进程在各个节点分布情况，举例来说，对于拉普拉斯方程雅克比迭代算法求解时中，数据按连接进程号对等划分，每一个进程需要同邻近的进程进行数据交换，但如果进程以这种循环分配的情况，邻近的进程可能分布在不同节点，这样就需要进行大量的节点间的通信，相比而言节点内的通信所需时间应远小于节点间的通信。

### 中间件的设计方法

MPI中间件的设计目标在于，一方面为用户提供一个构建层次化通信模型的接口，可以根据算法本身的特性来优化通信结构，另一方面根据MPI进程分配算法重构出客户端进程分布并为用户提供数据预处理。因此中间件包括两个重要的模块，第一、重构进程分布，为用户提供客户端进程分布，构建局部通信域，提供高效的集合通信；第二、利用进程分布提供数据预处理功能。

中间件中重构进程分布的方法有两种，一种在每个进程初始化完成以后，获取到应用程序进程数，再通过读hostfile文件去获取每一个节点的预分配进程数和节点信息，最后利用round robin算法，建立一张进程和节点的映射表；另一种每个进程初始化时获取主机信息，然后通过MPI的Allgather方法汇集所有进程和节点关联信息，最后同样能够构建进程和节点的映射关系。

### 中间件的API接口

多核感知MPI中间件提供以下主要API有：

MPIN\_get\_next\_rank(int curRank)*：*根据当前进程可以查找本节点下一个进程的rank值，如果是最后一个进程，则返回-1；

MPIN\_get\_prev\_rank(int curRank)*：*根据当前进程可以查找本节点上一个进程的rank值，如果是第一个进程，则返回-1；

MPIN\_get\_first\_rank(int curRank)*：*得到当前进程序列中第一个进程的进程号，error则返回-1；

MPIN\_get\_last\_rank(int curRank)*：*得到当前进程序列中最后一个进程的进程号error则返回-1；

MPIN\_get\_number\_of\_node()*：*得到总节点个数；

MPIN\_get\_node\_number\_of\_process(int node\_id)：根据node id查找当前节点进程数；

MPIN\_get\_node\_master\_rank(int node\_id)：得到当前节点上的master进程，在应对存储转发时，要每一个节点上设置一个主进程，以第一个位置进程主进程；

MPIN\_get\_master\_rank(int node\_id)：得到节点间主进程，一般情况情是0号进程；

MPIN\_get\_node\_process\_rank(int nodeid,int loc)：在给定节点上查找给定编号的进程rank值。在每个节点上除了有进程的rank值，还有对每个进程从0开始的编号；

MPIN\_get\_node\_by\_rank(int rank)：查找当前进程所在节点的id。

MPIN\_get\_local\_comm(comm,rank)：创建节点内的通信域。

MPIN\_get\_master\_comm(comm,rank)：创建节点间的通信域。

具体实现代码请参考：<https://github.com/linuxfl/paperTest/blob/master/butterfly/mpiNode.py>

## 层次化通信模型

从2.1中介绍和测试的结果来看，不同的进程分布对并行算法有一定的影响，因此，本文对于引入一种层次通信模型的结构来消除多核环境所带来的性能的差异。层次通信模型如图 2.3‑1所示，主要有两层通信，第一层是每一个节点内的通信，每一个节点内设定一个节点内主进程，第二层是节点间，节点间也设定一个节点间主进程。总体的通信关系是每一个节点只有节点主进程和其他节点主进程进行通信，而每个节点内部组成一个通信域。



图 2.3‑1 层次通信模型

在这种层次通信模型下，不建议进行节点间非主进程的通信，设定节点内主进程的目的在于减少节点之间的通信，提高并行算法可扩展性，同时由于节点内共享内存的IPC方式比节点间的Socket要高效的多，因此最大化节点内的进程之间的通信有利于提高算法的性能，并且上一节所提到的进程分布的导致的性能的差异性在层次通信模型中也不会存在。

## 分层并行编程模型

本节会基于常用的两种并行算法设计模式主从模式和对等模式提出相应的分层编程模型，以指导并行算法的设计和实现。

### 基于主从模式的分层并行编程模型

在映射表中设定了两层主从结构，节点内以进程rank值最小进程为主进程，其他的进程为从进程，而在节点间会以节点内主进程中rank值最小的(也就是0号进程)为主进程，其他进程为从进程，因此基于主从模式的分层编程模型如下：

*……*

*MPI\_Init(&argc,&argv);*

*MPI\_Comm\_rank(MPI\_COMM\_WORLD,&rank);*

*MPI\_Comm\_size(MPI\_COMM\_WORLD,&size);*

*MPIN\_init(…);*

*node\_id = MPIN\_get\_node\_by\_rank(rank);*

*master\_rank = MPIN\_get\_master\_rank(node\_id);*

*node\_master\_rank = MPIN\_get\_node\_master\_rank(node\_id);*

*if(rank == master\_rank){*

*numberofnode = MPIN\_get\_number\_of\_node();*

*for(i = 1;i < numberofnode;i++)*

*{*

*MPI\_Send/MPI\_Recv with master rank of every node;*

*}*

*}else if(rank == node\_master\_rank){*

*MPI\_Send/MPI\_Recv with master rank;*

*numberofprocess =MPIN\_get\_node\_number\_of\_process(node\_id);*

*for(i = 1;i < numberofprocess;i++){*

*slave\_rank = MPIN\_get\_node\_process\_rank(node\_id,i);*

*MPI\_Send/MPI\_Recv with slave rank of local node;*

*}*

*}else{*

*MPI\_Send/MPI\_Recv with master rank of local node;*

*}*

*MPI\_Finalize();*

*……*

以上是分层设计模式最基本的使用点对点的编程模型，对于节点间的集合通信，支撑库提供已组建好的进程通信域，使用MPICH的所有集合通信算法来提高性能。

### 基于对等模式的分层并行编程模型

MPIN\_get\_next\_rank(int curRank)、MPIN\_get\_prev\_rank(int curRank)两个API提供查找下一个进程号，但并不是按照进程号大小进行查找，而是根据进程在映射表中位置进行搜索，尽可能返回相同节点内的进程作为上一个或下一个进程，这样的方式最小化基于对等模式的并行算法节点间的通信，如Jacobi迭代，具体的基于对等模式的分层编程模型如下：

*……*

*MPI\_Init(&argc,&argv);*

*MPI\_Comm\_rank(MPI\_COMM\_WORLD,&rank);*

*MPI\_Comm\_size(MPI\_COMM\_WORLD,&size);*

*MPIN\_init(…);*

*if((prev = MPIN\_get\_prev\_rank(rank)) != -1){*

*MPI\_Send/MPI\_Recv with prev;*

*}else{*

*if((last = MPIN\_get\_last\_rank(rank) != -1 )){*

*MPI\_Send/MPI\_Recv with last;*

*}*

*}*

*if((next = MPIN\_get\_next\_rank (rank)) != -1){*

*MPI\_Send/MPI\_Recv with next;*

*}else{*

*if((first = MPIN\_get\_first\_rank(rank)) != -1){*

*MPI\_Send/MPI\_Recv with first;*

*}*

*}*

*MPI\_Finalize();*

*……*

## 通信优化策略

有了多核感知MPICH的中间件以及分层并行编程模型支持，可以对分层设计模式下通信提出两点的优化策略：第一、消息整合发送，即把多个单独的消息合并成一个大消息一次传递, 这样可以有效减少消息传递的启动时间【60】；第二、计算和全局通信尽量先在节点内进行，然后在节点间进行的方式，这样能够有效减少节点间通信。另外对等模式的算法尽量选择中间件中提供的API来减小节点间通信次数。

## 源数据输入优化

### 数据输入本地化

MPICH缺少分布式文件系统的支持，在分散式文件系统中，对大规模数据集输入需要通过加载到内存，然后在内存中进行划分，再通过进程间通信的方式，将数据块发送给其他的进程，这样方式有两个弊端：第一，大块的数据发送需要占用大量的带宽；第二，单机内存限制，无法加载超大规模的数据到内存。基于此，本文设计了一种数据输入本地化的方式，这个输入本地化可以看成是一种数据预处理的过程。

对于数据并行的算法，数据预处理实际上在并行算法之前将大块数据划分成若干小块数据，并分发到相应的节点，具体的是通过上节中动态构建节点和进程映射的方法，具体是让每个进程读取解析节点信息，同时获取分配总的进程数，再利用MPICH3.x中默认使用round robin的进程分配算法构建一个节点和进程映射表，映射表中保存节点和进程的对应关系。通过节点进程映射表查询每一个进程的分布情况，两种划分策略：第一是节点级划分方法，将原训练集按节点数和每一个节点划分到每个节点，第二是进程级的划分方法，将原训练集按总进程数和每一个节点进程分布来划分。节点级划分方法需要通过节点主进程将本地数据集读入内存，然后向每一个节点内子进程分发。

### 结合HDFS的数据本地化

MapReduce的核心是“数据本地化”，也就是hadoop会在块所存储的节点上启动一个Map任务，这样大大的减少数据在网络传输所带来的带宽，相比MPI虽然有提供了一些分布式文件系统并行IO的支持，如Lurster、GPFS，但是并没有“数据本地化”的特点，所以，本节结合分层设计模式所依赖的进程分布和HDFS的块分布规则，提供MPI数据本地化的读取方法。

HDFS块大小默认设为64M，而且建议块大小不要设置太小，其目的在于最小化寻址开销，但是通过MPI进程分布式读取的过程中，我们会按照MPI进程数和文件大小来划分块大小，这样可以避免进程的跨块数据读取，另外尽可能使用Hadoop提供的数据负载均衡的功能start-balancer.sh，具体过程如下：

1. 获得进程和节点分布；
2. 获得HDFS上文件的块分布；
3. 遍历块所在所有节点，通过查询进程和节点分布，找到当前节点的进程信息；
4. 遍历每个节点上分配的所有进程，找到没有分配块的进程
5. 如果块所有所在节点上进程都被分配完，就去其他节点上查找未被分配的进程
6. 如果找到没有分配的进程就标记该进程。

从上述步骤来说，最好的情况是所有块都可以通过第4步中的b步，此时是最好的方式，但是当块在集群节点上放置不均匀时，就可能出现a的情况，此时，就需要占用网络带宽来处理数据，因此我们要求尽量使用Hadoop提供的数据负载均衡的功能达到负载均衡的目的。

## 应用实例

### 矩阵乘法并行化

本文以矩阵乘法为例，利用上文介绍的通信优化策略和多核感知的MPI中间件，对主从模式下，简单按行划分矩阵乘法进行优化。

矩阵分行算法是指对于进行乘法运算的两个原始矩阵A与B，将矩阵A按行分行块，每个进程得到A的一个具有连续行号的行块，即进程得到的矩阵A的数据只包含完整的行，不包含完整的列。然后各进程将负责的矩阵A的行块与整个矩阵B进行乘法计算，得到的解为结果矩阵C的一个与参与该计算的矩阵A子块位置对应的行块。由于矩阵A的规模为，因此，分行算法的并行度最高为n。

分行算法具体的算法步骤如下：

1. 将矩阵A进行按行分块，0号进程为主进程，其它为从进程；
2. 分块的数量为进程个数p；
3. 前p-1个块中每一块的行数为，最后一个块中的行数为（考虑n对p除不尽的情况）；
4. p个A的行块数据依次分发给1号进程、2号进程……p-1号进程、0号进程；
5. 0号进程将矩阵B的数据广播给各进程；
6. 各进程将相应的A行块和B矩阵进行乘法运算，得到结果矩阵C的一个行块；
7. 各进程将计算结果归约至0号进程。

分行算法矩阵数据划分示意图如图2.7‑1所示。

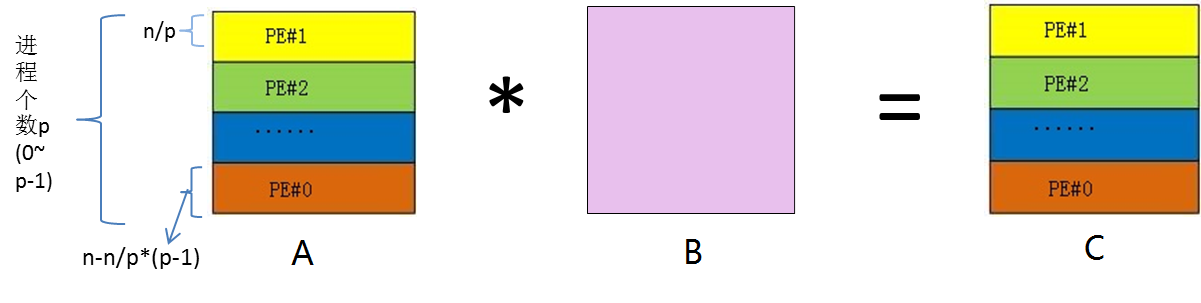


图 2.7‑1 分行算法矩阵数据划分图

各从进程得到规模为的矩阵A的子块及整个矩阵B，并进行子矩阵的相乘，主进程（在这里为0号进程）得到规模为的矩阵A的子块，因为原始矩阵规模n对进程数p不一定可以整除，所以由主进程完成剩余数据的乘法工作。各进程得到的结果矩阵的子块在结果矩阵中的位置与该进程所负责的A的子块在矩阵A中的位置是相对应的。

MPICH实现分行算法时，由主进程完成矩阵A的行块划分工作，并分发给各从进程，从进程完成自身负责的部分乘法运算后，将部分解传递至主进程，归约为最终的计算结果。

利用本章的通信优化方法进行改进，为了尽量减少节点间通信，数据以大数块一次向节点发送，而不是一小块一小块向每一个进程发送，然后在节点内分发，归约时也是需要通过节点主进程收集节点内所有进程的结果再一次向节点间主进程发送。

基于分层并行设计模式的分行矩阵乘法算法步骤如下：

1. 将矩阵A进行按行分块，0号进程为主进程，其它为从进程；
2. 获取节点个数m，分块的数量为节点个数m；
3. 前m-1个块中每一块的行数为，最后一个块中的行数为（考虑n对m除不尽的情况，这里考虑每一个节点的进程数一样）；
4. 获取每一个节点的节点内主进程；
5. m个A的行块数据依次分发给号进程、号进程……号进程、0号进程；
6. 每一个节点内主进程遍历所有节点内的子进程，并把数据均等划分，再发送给所有子进程；
7. 0号进程将矩阵B的数据结合本地通信域和节点间主进程通信域广播给各进程；
8. 各进程将相应的A行块和B矩阵进行乘法运算，得到结果矩阵C的一个行块；
9. 节点内子进程将结果先归约到每个节点内主进程，再由节点内主进程将结果归约到节点间主进程，也就是0号进程；

本文在3台PC组成的多核机群上对矩阵乘法在主从模式分行算法下(Master-Slave)和在分层设计模式下(Hierarchical Model)的分行算法进行测试。测试结果如图2.7‑2。



图 2.7‑2 矩阵分行并行算法在两种设计模式下的测试结果

主从设计模式下数据分发和广播分别使用MPICH的Scatter方法和Broadcast，在分层并行设计模式下通过消息合并的方法，将发送到同一节点的所有进程的数据进行合并，发送到节点主进程，然后由节点主进程在节点内进行分发；计算结果收集的过程相反，先由节点内收集，再节点间收集，这样减少大量节点间通信，同时减少了消息传递的启动时间。从测试结果可以看出分层并行设计模式下分行矩阵乘法性能比主从设计模式高出1.4~1.7倍。

### 分布式并行ELM算法

SLFN通过非梯度下降型快速学习算法取代了训练效率较低的梯度下降算法，并且使用Sigmoid激活函数的SLFN，最多需要N个隐层神经元，就能精确学习N个不同的样本，并且隐层神经元的输入权值和偏置可以进行随机赋值。更进一步，扩展至具有任意无限可微激活函数的SLFN，当隐层神经元数量小于训练样本数量时，具有随机隐层输入权值和偏置的SLFN能以任意小的误差逼近N个样本。因此ELM算法【61】通过对SLFN隐层神经元和偏置进行随机赋值，然后通过Moore-Penrose广义逆直接求解SLFN网络的最小二乘问题。

ELM算法无需迭代，要显著快于传统的BP神经网络和SVM。对于 个隐层神经元的SLFN网络的输出可以表示为：

|  |  |
| --- | --- |
|  | (2.7-1) |

其中，表示激活函数，表示第个神经元的输出权重。

对于个不同的样本，其中**,**，标准的带有 个隐层神经元和激活函数为的SLFN可以表示为：

|  |  |
| --- | --- |
|  | (2.7-2) |

上式可以简化为：

|  |  |
| --- | --- |
|  | (2.7-3) |

其中：

，

上式的最小二乘解为：

|  |  |
| --- | --- |
|  | (2.7-4) |

其中，是H的Moore-Penrose广义逆矩阵。当非奇异时， ，但是实际在大规模数据集训练过程中有可能奇异或接收奇异，Huang G B 等人提出了基于Equality Constrained-Optimization ELM【62】，使得ELM算法获得更好的性能和更具有鲁棒性。其中输出权重计算如下：

|  |  |
| --- | --- |
|  | (2.7-5) |

或者：

|  |  |
| --- | --- |
|  | (2.7-6) |

根据以上的讨论，可以给出ELM串行算法：

给定一组训练集，激活函数和隐层神经元个数

步骤1：随机分配任意的输入权重和偏置

步骤2：根据输入和输入权重和偏置来计算隐层输出矩阵H

步骤3：根据输出矩阵H计算输出权重

面对大规模数据集时，传统串行ELM算法难以应对规模庞大的隐层输出矩阵进行计算和存储。所以将ELM算法分布式并行是一个有效的解决方法。从ELM串行算法的步骤来看，计算过程大体分为三步，第一步是随机分配输入权重和偏置，此步耗时较少，第二步，计算输出矩阵H主要是矩阵乘法，较易并行化，第三步，求解广义逆矩阵时，和求解过程花费时间较多，由于是，在一般情况下，训练数据集，所以求解耗时较少，因此可以采用的并行方案是，将计算隐层输出矩阵H和输出矩阵T在主进程中按总进程数进行按行划分到每一个进程，然后将每一个进程计算得到各自的和，最后汇集得到最终结果，计算过程如图1。



图 2.7‑3 基于主从模式并行ELM算法流程

如图2.7‑3所示，整个主从模式并行ELM算法流程包括主进程处理流程和从进程处理流程：

主进程处理流程如下：

步骤1：从本地读取所有训练集。

步骤2：向子进程分发训练集。

步骤3：随机生成隐层神经元权重和偏置并广播给所有子进程。

步骤4：计算本地隐层输出矩阵，并计算和。

步骤5：收集从各个子进程发送来的和。

步骤6：计算并结束。

子进程处理流程如下：

步骤1：接收部分训练集。

步骤2：接收隐层神经元权重和偏置。

步骤3：计算本地隐层输出矩阵，并计算和。

步骤4：将和发给主进程。

步骤5：结束并退出。

ELM并行算法的通信过程中主要是数据归约部分，在分层设计模式下分为两步进行，第一步，每一个节点内归约到节点内主进程(node master rank)，第二步，节点间归约到每一个节点间主进程(mater rank)，同时数据的划分方法是选择进程级的划分方法，具体流程



图 2.7‑4 基于分层并行设计模式并行ELM算法流程

如图2.7‑4所示，整个分层并行设计模式下ELM并行算法的主要有两个阶段，一个数据预处理阶段，一个是并行算法执行阶段，并行算法执行阶段包括三节点间主进程处理流程、节点内主进程处理流程、节点内从进程处理流程。

数据预处理阶段：

步骤1：读取主机信息和总进程数

步骤2：构建节点进程映射表。

步骤3：将原样本训练按照进程分布进行分发。

步骤4：结束。

并行算法执行阶段：

节点间主进程处理流程：

步骤1：从本地读取训练集。

步骤2：随机生成隐层神经元权重和偏置，并广播给所有节点内主进程。

步骤3：计算本地隐层输出矩阵，并计算和。

步骤4：收集从各个节点内主进程发送来的和。

步骤5：计算并结束。

节点内主进程处理流程：

步骤1：从本地读取训练集。

步骤2：接收节点间主进程广播的隐层神经元权重和偏置。

步骤3：计算本地隐层输出矩阵，并计算和。

步骤4：收集从各个节点内从进程发送来的和。

步骤5：将计算总的和，并将结果发送给节点间主进程。

步骤6：结束并退出

节点内从进程处理流程：

步骤1：从本地读取训练集。

步骤2：接收节点内主进程发送的隐层神经元权重和偏置。

步骤3：计算本地隐层输出矩阵，并计算和。

步骤4：将和发送给节点内主进程。

步骤5：结束并退出。

假设现在有个训练样本，SLFN网络的隐层神经元个数为，为了区别分类和回归应用，令输出神经元个数为(在回归问题中，在分类问题中等于训练样本的类别数)【62】，得到隐层输出矩阵，在不采用任何硬件和软件优化方法的情况下，的时间复杂度为，矩阵求逆的时间复杂度是，因此计算出串行ELM算法的时间复杂度为。

基于主从设计模式ELM并行算法的运算时间主要包括从进程的计算时间和数据传输时间，假设当前启动的进程数为个，因此原训练样本划分成块，整个算法的计算时间复杂度为，数据传输时间包括训练集分发时间和每个从进程把计算结果发给主进程的时间，为了准确分析数据传输时间，算法通信使用基本的点对点方式，同时引用一种简化的通信模型【63】：

|  |  |
| --- | --- |
|  | (2.7-7) |

其中的是发送和和接收的启动开销时间，是单位数据的传输时间，是传输的消息量。但是为了区别节点间通信和节点内通信，节点内的通信模型【64】为：

|  |  |
| --- | --- |
|  | (2.7-8) |

其中是节点内进程发送和接收和启动开销时间，是节点内单位数据的传输时间。假设与主进程同一节点的从进程数为，原始样本分发通信时间为，从进程把计算结果发给主进程的时间为，由于权重偏置广播的通信时间复杂度是，所以相对上面两个通信时间可以忽略。

基于分层设计模式ELM并行算法在进程数为时，运算时间复杂度和主从模式下一样为，但通信时间只包含从进程把计算结果发给主进程的时间。假设所有节点内从进程数有个，则节点内主进程有个，那么所有从进程把计算结果发主进程的时间为。

将串行ELM算法和基于分层设计模式ELM并行算法进行比较，一般情况下，面向大规模数据集时，，则串行ELM算法的计算时间复杂度约为，分层设计模式ELM并行算法计算复杂度约为，在多核机群环境下，其中为每个节点平均分配的进程数，分层设计模式通信时间复杂度为，而，为了方便比较使用一个统一的系数，令，，而，由于，则，由于在多核环境下节点平均分配进程数一般多于1个，即，则，一般情况下，训练样本通常是上百万甚至更多，而大规模集群环境下，通常在两三位数,则一般会在一两位数，所以，即。根据以上时间复杂度的分析，一般在大规模数据集下，分层设计模式的ELM并行算法性能要明显优于串行ELM算法。

基于主从模式的ELM并行算法和基于分层设计模式的ELM并行算法主要对比通信时间，主从模式ELM并行算法通信时间由上文介绍分为两个部分，所以总的通信时间为，分层设计模式的ELM并行算法通信时间。由于，而，所以，即，因此分层模式ELM并行算法性能优于主从模式下ELM并行算法。

实验在由5台PC组成的多核机群上进行测试，具体硬件配置参考表格2.7‑1，实验代码是在Linux下用C、[OpenBLAS](http://sourceforge.net/projects/openblas/)库【65】与MPI并行库实现。

表格 2.7‑1 PC机群配置

| Hardware spec. | Model1 | Model2 |
| --- | --- | --- |
| CPU | Inter(R) Core(TM) i5-3230 CPU @ 2.60GHz | Inter(R) Core(TM) i3-2130 CPU @ 3.40GHz |
| MEM | DDRIII 1066 4G | DDRIII 1066 4G |
| Hard Disk | 500G | 500G |
| Soft System | Linux 2.6.32-279. El6. I686 | Linux 2.6.32-279. El6. I686 |

测试的数据集有Friedman人工数据集和mv 人工数据集两个回归数据集【66】，和UCI Covertype 分类数据集【67】如表格2.7‑3和表格2.7‑5，对ELM串行算法(ELM)、基于主从模式ELM并行算法(ELM-Parallel-MS)以及基于分层设计模式ELM并行算法(ELM-Parallel-HM)进行测试，测试内容包括训练时间，RMSE以及分类正确率及通信时间。

实验中选用Sigmoid函数作为SLFN的激活函数，同时通过两种模式下ELM并行算法经过反复测试，确定在每个数据规模下，当总进程数为20时，训练时间最少，因此图 2.7‑5和图 2.7‑6是总进程数为20时的测试结果。

表格 2.7‑2 Information of the regression data sets

|  |  |  |  |
| --- | --- | --- | --- |
| Data sets | Attributes | Samples | |
| Training | Testing |
| Friedman | 6 | 1 000 000 | 200 000 |
| MV | 11 | 1 000 000 | 200 000 |



图 2.7‑5 MV和Friedman数据集测试结果

从测试结果可以看出串行ELM算法在训练集从100000增加到200000过程中，训练时间急剧增长，而两种模式下的ELM算法在训练集增长的过程中训练时间增长平稳。对比两种并行ELM算法，分层设计模式下的ELM并行算法性能明显高于主从模式下的ELM算法。

表格 2.7‑3 RMSE in ELM and ELM-Parallel-MS and ELM-Parallel-HM

|  |  |  |  |
| --- | --- | --- | --- |
| Methods | Datasets | RMSE | |
| Training | Testing |
| ELM |  | 0.019326 | 0.018386 |
| ELM-Parallel-MS | Friedman | 0.020343 | 0.019221 |
| ELM-Parallel-HM |  | 0.019345 | 0.020696 |
| ELM |  | 0.023223 | 0.024624 |
| ELM-Parallel-MS | MV | 0.024256 | 0.029639 |
| ELM-Parallel-HM |  | 0.024312 | 0.028656 |

表格 2.7‑4是三种算法的对应Friedman和MV数据集数据规模在200000时RMSE的测试结果，从数据可以看出三种算法两种数据集下的测试出的对应RMSE没有太大差别，因为三种算法计算输出权重矩阵的过程相同。

表格 2.7‑4 Information of the classification data sets

|  |  |  |  |
| --- | --- | --- | --- |
| Data sets | Attributes/Classes | Samples | |
| Training | Testing |
| Covertype | 54/7 | 400 000 | 18 000 |



图 2.7‑6 Covertype数据集测试结果

图 2.7‑6是Covertype分类数据集测试结果（源数据集有降维处理），同Friedman和MV数据集测试结果相似，串行算法在训练集在100000到200000之间训练时间呈指数级增长，两种并行算法性能对比，分层设计模式下的ELM并行算法性能更好。

表格 2.7‑5 Success Rate in ELM and ELM-Parallel-MS and ELM-Parallel-HM

|  |  |  |  |
| --- | --- | --- | --- |
| Methods | Datasets | Success Rate | |
| Training | Testing |
| ELM | Covertype | 0.7447 | 0.7512 |
| ELM-Parallel-MS | Covertype | 0.7767 | 0.7580 |
| ELM-Parallel-HM | Covertype | 0.7784 | 0.7701 |

分类数据集是带有类别标签，串行ELM算法测试时，由于单机内存限制，只能在原训练集中选出部分数据集进行训练，表格 2.7‑6中的分类正确率是200000个样本的测试结果，并且选出的数据是包含所有类别标签。而两种并行算法在性能测试时只是选用部分训练集测试，表格 2.7‑6中分类正确率是所有训练集(400000)的测试结果。从表格 2.7‑6中可以看出三种算法测试出的分类正确率并没有太大的差异。



图 2.7‑7 通信时间和计算时间所占比例

图 2.7‑7给出了三个数据集在不同的设计模式下并行算法的通信和计算时间比例，由于分层设计模式提供的数据预处理的方法减少了主从模式下原数据集经过网络分发的通信时间，另一方面层次化通信模型减少了大量的节点间通信，使得ELM-Parallel-HM算法相对ELM-Parallel-MS算法通信时间所占总运行时间比例要少，从而提高了算法的整体性能。

## 本章小结

本章首先分析了进程分布对并行算法的影响，以及MPI本身缺少数据管理等问题，提出了一种分层并行设计模式，并且详细介绍多核感知中间件，层次化通信模型，分层并行编程模型和通信优化策略，同时通过矩阵乘法和ELM算法来验证分层并行设计模式的并行优化效果，从测试结果显示，通过分层并行设计模式进行的设计矩阵乘法和ELM并行化算法性能比传统的主从模式的更具高效性和高可扩展性。

# 基于分层并行设计模式的异步分布式ADMM算法

本章首先介绍一种分布式ADMM算法，然后提出一种异步分布式ADMM算法模型，并利用分层并行设计模式构建三种层次化通信模型应用到异步分布式ADMM算法，同时对静态通信模型和动态通信模型两种场景下异步分布式ADMM算法进行理论收敛性的分析，最后通过实验进行收敛性和性能进行测试。

## 分散一致性分布式ADMM算法

考虑分散一致性优化问题，假设有个结点，条边的网络结构，可以通过一个有向图或者一个无向图，其中表示顶点集合，且，表示弧集合，，表示边集合，，具体问题描述如下：

|  |  |
| --- | --- |
|  | （3.1-1） |

其中是本地目标函数，是本地模型参数，而表示连接node i和node j的辅助一致性变量。

为了能应用ADMM算法，定义是所有的向量集合，；是所有向量集合，，因此将上式转换成矩阵形式如下：

|  |  |
| --- | --- |
|  | （3.1-2） |

其中表示 单位矩阵，是块的单位矩阵，即行列个。当且是的第块，对应位置的是单位阵，位置的是单位阵，否则为。

现在通过ADMM算法来解决问题(2)，增广拉格朗日函数如下

|  |  |
| --- | --- |
|  | （3.1-3） |

其中，是拉格朗日乘子，。

由最优性条件可知：

|  |  |
| --- | --- |
|  | （3.1-4） |

其中如果可导，则是处的导数，如果不可导，则是处的次导数，将第三式两边乘以加到第一式上，并且将第三式两边乘以加到第二式上得

|  |  |
| --- | --- |
|  | （3.1-5） |

令，其中，由于和二式，则，公式（3.1-5）第一式可以转化为

|  |  |
| --- | --- |
|  | （3.1-6） |

其中，，将第三式划分成以下两式

|  |  |
| --- | --- |
|  | （3.1-7） |

给定初始化，这样在恒成立，然后将上次两式相加和相减得

|  |  |
| --- | --- |
|  | （3.1-8） |

如果初始化，那么在恒成立，综上所述

|  |  |
| --- | --- |
|  | （3.1-9） |

将第三式代入一式得

|  |  |
| --- | --- |
|  | （3.1-10） |

将第二式两边乘以加到一式上得

|  |  |
| --- | --- |
|  | （3.1-11） |

令，，其中，，是扩展有符号和无符号的拉普拉斯矩阵，是扩展的度矩阵，同时重新定一个乘子，则可以得

|  |  |
| --- | --- |
|  | （3.1-12） |

由于，，其中，可以由上式得到ADMM的迭代形式

|  |  |
| --- | --- |
|  | （3.1-13） |

其中表示结点i的邻居结点集合。从迭代形式来看，每个结点更新依赖于这个结点邻居结点的本地模型参数，全连接网络中，每一个结点需要将本地模型参数广播给其他结点，在MPI中每次迭次时可以通过一次AllReduce函数的调用来实现以上的需求。

## 异步分布式ADMM算法

我们可以把异步看成每次迭代有着不同的通信拓扑，因此从迭代开始到结束通过一组有向图来描述算法的异步执行的过程，其中是一个有向图，表示所有顶点的集合，且，表示边的集合，，。根据上面的描述优化问题如下(2)：

|  |  |
| --- | --- |
|  | （3.2-1） |

定义是所有的向量集合，；是所有向量集合，包括所有，因此将上式转换成矩阵形式如下（3）：

|  |  |
| --- | --- |
|  | （3.2-2） |

其中,和表示网络选择的指示矩阵(Indicator matrix)，表示不同的网络个数，所以在第次迭代限制条件可以转化为两个式子，，而和是块的单位矩阵，即行列个，是块的矩阵，除了对角线，其他的位置全是，当且是的第块,的位置是，对应位置的是单位阵，位置的是单位阵。

应用拉格朗日来解决问题（3），其增广拉格朗日如下：

|  |  |
| --- | --- |
|  | （3.2-3） |

ADMM的更新迭代形式为：

|  |  |
| --- | --- |
|  | （3.2-4） |

更新迭代形式如下：

|  |  |
| --- | --- |
|  | （3.2-5） |

其中表示的导数当其可导时，或者是次导数当其不可导时，表示第次迭代时的邻居节点集合，，，。

## 基于分层并行设计模式的ADMM算法的设计

### 三种层次化模型的设计

从异步分布式ADMM算法迭代更新的形式来看，问题(2)可以通过分布式的来解决，因为每一个进程可以通过和邻居进程进行消息交换来优化求解目标函数得到模型参数，另外，问题(2)本质上是一个基于网络的一致性问题，给定的等式限制，当网络是连通时，我们考虑任意的两个节点，必然存在一条路径，使得，因此对于所有节点，存在，，所以解决了问题(2)等价于解决了问题(1)。

上一节讨论的异步分布式ADMM算法是面向所有通信模型，但是我们目标是，第一必须要满足全局一致性问题的要求，即通信拓扑结构是连通的，第二是需要寻求一种高效通信模型，并且这种通信模型不会影响到算法的收敛性，因为需要得到高效通信的方式无非是在原有的全连接通信基础上添加一定的通信限制，例如减少通信的次数，但是这样造成的后果是减慢了算法的收敛性，所以需要权衡算法的收敛性和高效的通信，为此，本章结合上一章提出一种基于分层并行设计模式，即将层次通信模型应用到异步分布式ADMM算法。

在层次通信模型中，节点内通信大多使用share memory的方法而且每个计算节点由于核数的限制，能达到最大负载所启动的进程数有限，所以一般情况下在本地使用全局一致，即所有进程互为相邻进程。各个节点主进程因为分布在不同节点，所以不同的通信结构会对算法的通信时间造成不同的影响。图 3图 4是三种不同的节点内主进程（Intra-Master）的通信模型。



图 3.3‑1 Inter-Master环形



图 3.3‑2 Inter-Master全连接



图 3.3‑3 Inter-Master蝶形

一般的并行环境下，所有进程是全连接，简称为FC，应用这种通信模型的分布式ADMM算法，一次迭代需要一次全局的数据交换。本文提出三种层次模型，第一种是节点内主进程全连接(Intra-Master Fully Connected)的情况，简称为IMFC，即所有节点内主进程都互为邻近进程，第二种是节点内主进程环形结构(Intra-Master Cycle)，简称为IMC。第三种是节点内主进程碟式通信模型(Intra-Master Butterfly)，简称为IMB，该模型属于动态模型。本文选取了三种通信模型都属于对称性的模型，而且每个节点内主进程的邻近进程数都相等，这种有利于通信平衡，以免带来较大的非平衡到达的问题。

### 算法的描述与分析

应用层次通信模型的异步分布式ADMM算法有两层通信关系，第一层是本地局部通信，第二层是节点间的按不同通信结构的通信，具体HD-ADMM算法如下：

HD-ADMM算法流程

算法：基于层次通信模型的分布式ADMM算法(HD-ADMM)

输入：（进程数），，，，，

第一步：更新本地模型参数：

第二步：节点从进程归约到节点内主进程：

第三步：如果是节点内主进程，根据节点间通信模型归约节点内主进程

模型参数：，如果不是则跳过此步。

第四步：节点内主进程 在节点内广播

第五步：更新：

第六步：判断是否满足停机准则，如果满足，则停止迭代，否则迭代次数加1并跳到第一步

该算法中在次迭代中首先更新，然后每一个进程（processor）将本地归约到Intra-Master，如果当前进程为Intra-Master，再根据节点间通信模型进行数据交换。其中表示第进程第次迭代本地邻近进程数，表示第进程第次迭代节点间（inter-node）邻近进程数。如果进程是Intra-Master则，如果进程不是Intra Master，则，最后由Intra-Master在本地广播，时序关系如图 3.3‑4。



图 3.3‑4 一次迭代时序关系图

当网络拓扑为蝶式时，应用异步分布式ADMM算法，其特点在于假设节点数是，且是2的幂次方，通信模型以为变化周期，且算法经过次迭代相当一次Allreduce过程，此时蝶式通信拓扑可以看一个连通图，因此有：

其中，表示迭代次数，是符号函数，即：

例如当时，一次循环需要3步,即，所以当时，时，时。

蝶式通信模型数据交换算法如算法2：

|  |
| --- |
| **Algorithm 2** Butterfly data exchange Algorithm |
|  |
|  |
| **function** |
| /\*size is power of 2\*/ |
|  |
|  |
|  |
|  |
|  |
| exchange data with |
|  |

算法2中假设进程从0开始编号，当第次迭代时，比较每一个进程号二进制数的位的值，若为1和其他位相等且位为0的进程号的进程进行数据交换，同理，若为0则和其他位相等且位为1的进程号的进程进行数据交换。

## 收敛性分析

本文讨论的4种通信模型中，FC属于非层次通信模型，其他三种是层次通信模型，因此假设层次通信模型节点内通信时间为，节点间通信时间为，计算时间为，迭代次数为，则层次通信模型的总时间为；假设非层次通信模型FC每次AllReduce通信时间为，计算时间为，迭代次数为，则非层次通信模型的总时间。

由于而计算时间相等，所以，因此算法的总运行时间受迭代次数以及通信时间的影响。在大规模计算环境下，当计算节点增多时，的通信时间也就是一次AllReduce的时间会不断的增多，不会受到影响，此时主要迭代次数会影响算法的性能，所以下面分析增加计算节点对算法收敛性也就是迭代数的影响。

### 静态通信模型场景下算法收敛性分析

下面来分析算法的收敛性，Shi W给出网络拓扑结构对算法收敛性影响的指标和连通率【7】，其中表示最大的奇异值，表示的最小奇异值，在静态网络中，。我们通过模拟实验测试了IMC和IMFC网络结构下和计算节点数以及本地进程数的关系。



图 3.4‑1 本地进程数和计算节点数和的关系

从图 3.4‑1可以看出，IMFC下随着计算节点数的增长几乎不会增长，对收敛性影响较小，在IMC（cycle）下随着节点数的增长，值会呈折线式的变化，对收敛性造成一定的影响。

我们可以根据连通率来分析，根据定义个进程（processor）形成网络时最多的边长为，因此网络的连通率就是当前网络的边数除以。IMFC的通信模型的连接通和FC一样为1，而环形的连通率如下

|  |  |
| --- | --- |
|  | （3.4-1） |

当时，而越小，算法的收敛越慢。

### 动态通信模型场景下算法收敛性分析

蝶式通信模型是动态模型，无法使用静态网络中的值和连通率来评价算法收敛率，但是我们可以通过下面的方法来分析。

**假设 1：目标函数是强凸性，且为利普希茨连续。**

假设现在有计算节点且，由于蝶形通信模型每次迭代节点间两两进行数据交换，并且每两个节点会组成FC（fully connected）通信模型，而应用FC通信模型的ADMM算法的收敛性在【68】等文献中有证明，并且算法在假设1的条件下可达到线性收敛率【7】，则有：

合并上面各式得到：

|  |  |
| --- | --- |
|  | （3.4-2） |

其中，因此，我们把n次迭代当作一次大迭代，显然此时大迭代次数的会随着计算节点的增多而不断的减少，因此

|  |  |
| --- | --- |
|  | （3.4-3） |

其中，为算法的总迭代次数，是基于FC（fully connected）通信模型的ADMM算法迭代次数，是关于的一个有下界的递减函数，而应用FC通信模型的ADMM算法的迭代次数是不会随着节点增多而增多的，所以算法的总迭代次数随着计算节点的增长不会过分增长，并且应用层次蝶形通信模型仍然能使异步分布式ADMM算法保证线性收敛率。

## 实验结果与分析

### 实验环境

收敛性测试通过单机模拟实验，而性能测试，我们在32个节点的集群系统，千兆以太网，每一个节点4核CPU和4GB内存，操作系统为Ubuntu 14.04进行测试，实验代码通过python实现，需要的并行库有MPICH3.1、mpi4py。

我们使用线性回归应用进行测试，线性回归所要求解的最优化问题如下：

|  |  |
| --- | --- |
|  | （3.5-1） |

其中是系数矩阵，是输出矩阵。

应用异步分布式ADMM的迭代更新步骤如下：

|  |  |
| --- | --- |
|  | （3.5-2） |

实验的收敛条件为当任意的满足，其中为原问题的解向量。

### 收敛性测试与分析

我们通过模拟实验测试三种通信模型在计算节点数增加时的算法迭代次数变化来验证收敛性分析出的结论，测试结果如下：



图 3.5‑1 静态网络下计算节点数与进程数算法收敛性关系



图 3.5‑2 蝶形网络下计算节点数与进程数算法收敛性和的关系

从图 3.5‑1中数据显示，IMFC和FC通信模型下，随着节点数的增加，迭代次数没有太大的变化，而环形结构下，迭代次数会随着节点的增加有大幅增加，主要由于节点增多对造成一定的影响。

图 3.5‑2右图中可以看出应用IMB通信模型的算法大迭代次数是随着计算节点数的增加而递减，并且算法总迭代次数随着计算节点数加而略微有所增加，这个结果和理论分析结果相一致。

### 性能测试与分析

考虑使用线性回归作为测试用例，特征矩阵，其中总样本数，特征数。每个进程分配样本，测试随着节点数增加时，在4种通信模型下的通信时间以及算法总时间。

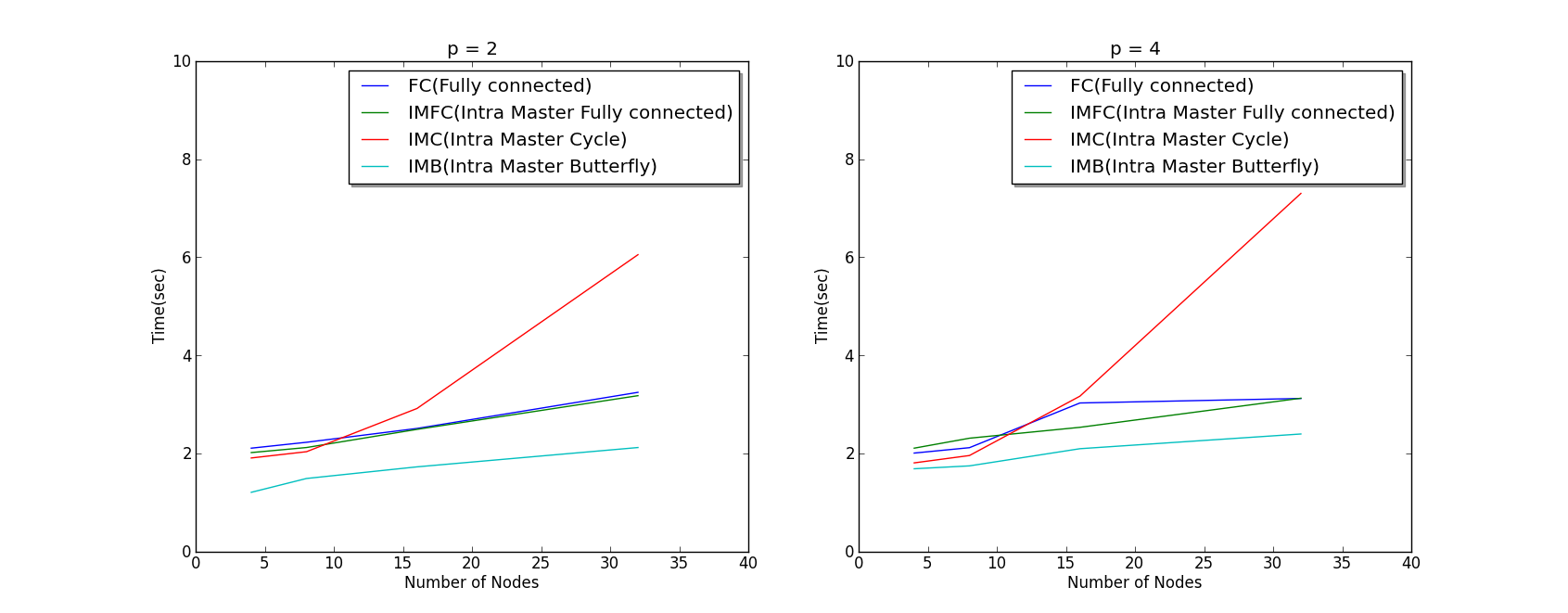


图 3.5‑3 FC、IMFC、IMC、IMB四种模型下在本地进程

数为2、4时的HD-ADMM算法性能测试



图 3.5‑4 FC、IMFC、IMC、IMB四种模型下在本地

进程数为2、4时的HD-ADMM算法通信时间

从图 3.5‑3、图 3.5‑4和表格 3.5‑1、表格 3.5‑2所示，在IMC通信模型下由于算法迭代次数随着节点会快速增长，计算时间也随着大量增加，所以算法的运行时间会增长较快，而IMFC和FC在每次迭代时间上相差无几，因此算法运行时间几乎相等，而在IMB通信模型下，节点越多，算法迭代次数并没有大量增加，而通信时间也不会随着节点的增多而增长，因此算法运行时间增长较为平稳。

表格 3.5‑1 FC、IMFC、IMC、IMB四种模型下HD-ADMM算法总时间

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 通信模型 | 本地进程数为2 | | | | 本地进程数为4 | | | |
| N=4 | N=8 | N=16 | N=32 | N=4 | N=8 | N=16 | N=32 |
| FC | 2.11 | 2.23 | 2.512 | 3.248 | 2.01 | 2.12 | 3.031 | 3.123 |
| IMFC | 2.02 | 2.123 | 2.493 | 3.18 | 2.11 | 2.313 | 2.535 | 3.128 |
| IMC | 1.91 | 2.038 | 2.919 | 6.054 | 1.81 | 1.96 | 3.17 | 7.303 |
| IMB | 1.21 | 1.492 | 1.729 | 2.123 | 1.69 | 1.749 | 2.099 | 2.399 |

表格 3.5‑2 FC、IMFC、IMC、IMB四种模型下HD-ADMM算法通信时间

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 通信模型 | 本地进程数为2 | | | | 本地进程数为4 | | | |
| N=4 | N=8 | N=16 | N=32 | N=4 | N=8 | N=16 | N=32 |
| FC | 0.712 | 1.00 | 1.12 | 1.65 | 0.881 | 0.90 | 1.51 | 1.554 |
| IMFC | 0.741 | 0.8874 | 1.135 | 1.566 | 0.782 | 0.812 | 1.14 | 1.555 |
| IMC | 0.4371 | 0.5895 | 0.7314 | 1.39 | 0.483 | 0.58 | 0.7575 | 1.59 |
| IMB | 0.213 | 0.237 | 0.3227 | 0.3527 | 0.2917 | 0.4217 | 0.323 | 0.421 |

单位：秒

## 本章小结

本章详细介绍了一种分布式ADMM算法，并根据此算法，提出了异步分布式ADMM算法模型，并且结合上一章的分层并行设计模式构建三种层次化通信模型应用到异步分布式ADMM算法中，然后通过理论分析和实验验证得出应用层次化蝶形通信模型的异步分布式ADMM算法更加高效及高扩展性。

# 基于Petuum的异步分布式ADMM算法

本章首先详细介绍Petuum中数据并行模块的SSP模型，并且分析了同步分布式ADMM算法，然后利用SSP模型的旧值访问的机制，提出一种基于SSP模型的异步分布式ADMM算法，并在收敛性分析中，针对目标函数为非凸和凸两种场景进行分析。

## SSP模型详解

SSP模型是利用机器学习算法的容错性构建的一种并行计算模型【6】，机器学习的容错性是指在误差容许的范围内，算法迭代计算过程中可以存在一定的误差。例如随机梯度下降算法，可以在每一次迭代过程中容许存在一定的噪音梯度，而前提是不影响算法的收敛性。在SSP模型中，给定了一个时间窗口，并保证了两个线程维持在这个时间窗口内，这样使得误差保持在一定的范围，并且有理论收敛性的保证【6】，具体模型如图 4.1‑1：



图 4.1‑1 SSP模型

假设给定一个用户输入的staleness阈值，SSP模型会保证以下条件成立：

1. 慢的thread和快的thread时间间隔必须小于等于staleness，否则快的thread必须要停下来等待，这里快的thread并不是停下来不做任何事，而是去访问一次parameter server[\*]，使本地更新到最新值，这样做既满足时间窗口的条件，又可以更新最新值，减少噪音。
2. 每一个thread读取parameter server时，读到可能是旧值，即本地cache里面值，而旧值时间一定是大于等于c-s-1，c是thread的当前时间.
3. 每一个thread每一次读parameter server时，读到的数据一定包含自己的上一次提交的那一部分数据，这就是所谓的“read-my-write”.
4. 每一个thread本地时钟是s整数倍时，去读取一次parameter server，而不是访问本地cache.
5. 每一个thread提交数据时会更新本地时钟，同时更新本地cache的值.

SSP模型仅仅提供了一种异步并行机制，它还需要依赖于SSPTable，一种高效的SSP系统，它满足分布式客户端-服务器框架，包括远程服务器端的table和客户端的thread cache、process cache等，并且SSPTable提供了以下的API去读写共享变量，

1. read\_row(table,row,s)：在staleness为s时，读取table的row行的数据。
2. inc(table,row,el,val)：向table的row行的el提交一个增量val，不会立即向服务端提交，只有调用clock方法时才提交。
3. clock()：通知server当前调用的线程和进程已经完成了一次clock。

## 全局一致性分布式ADMM算法

假设有如下优化问题：

|  |  |
| --- | --- |
| s.t. | （4.2-1） |

其中

其增广拉格朗日函数如下：

|  |  |
| --- | --- |
|  | （4.2-2） |

ADMM迭代更新：

|  |  |
| --- | --- |
|  | （4.2-3） |

其中，以上就是最原始的ADMM算法迭代形式。

如果要将ADMM应用到机器学习算法分布式计算中，需要转换一个比较广义的优化问题如下：

|  |  |
| --- | --- |
| *s.t.* ， | （4.2-3） |

以上问题形式就是全局一致性问题，是局部变量，是全局一致变量，从限制约束条件来看，每一个局部变量需要保持一致，其增广拉格朗日函数为：

|  |  |
| --- | --- |
|  | （4.2-4） |

ADMM迭代更新形式：

|  |  |
| --- | --- |
|  | （4.2-5） |

更进一步将上述问题推广更广义的优化问题，带正则化的全局一致性优化问题，具体问题形式如下：

|  |  |
| --- | --- |
| *s.t.* ， | （4.2-5） |

ADMM迭代更新形式：

|  |  |
| --- | --- |
|  | （4.2-6） |

在MPICH上要实现ADMM算法，每个进程先计算本地的模型参数，然后通过reduce的方法计算所有进程的，和然后广播给所有进程，这里为了方便并行化，可以将更新进行变换如下

从上式知，可以用一次MPI的AllReduce来实现，具体算法如下：

sync-ADMM算法流程

算法：基于MPI的同步分布式ADMM算法(sync-ADMM)

输入：（进程数），

第一步：更新本地模型参数：

第二步：每个进程使用MPICH AllReduce得到

第三步：每个进程更新本地

第四步：每个进程更新

第五步：判断是否满足停机准则，如果满足，则停止迭代，否则迭代次数加1并跳到第一步

## 基于SSP模型的异步分布式ADMM算法

### 算法的设计与分析

基于MPI的同步分布式ADMM算法中每一个线程的时钟也就是迭代次数始终是一致的，但在异步分布式ADMM算法中每一个线程是独立的进行迭代，因此，每一个线程设置一个自己的时钟为，另外参数服务器不参与运算，但是为了方便，设置一个参数服务器的时钟为，与是独立计时的，并且、表示线程*i*在次迭代的原变量和对偶变量。

Petuum是基于参数服务器模型【69】【70】的分布式框架，而ADMM算法需要保存一致性参数，但是由于是要通过计算得到，而Petuum不存在一个master线程来计算值和其他线程通信，但由上一节可知，值的更新是和相关的，令，因此我们把保存在参数服务器，让每一个线程读取，因此每一个线程中的更新

|  |  |
| --- | --- |
|  | （4.3-1） |

其中是为了方便后文的收敛性的讨论[13]。由SSP模型条件可知，每一个线程读取到的可能会有两种情况，一是从参数服务器里读到，一种是从cache中读到，如果是cache中的旧值，则从使用线程上一次从参数服务器中的来更新，直到其读到参数服务器中的数据。

、更新是通过来更新，这里的由上文讨论可知有可能是旧值，则

然后提交计算得到的和的差值提交参数服务器。

SSP-ADMM算法流程

算法：基于SSP模型的异步分布式ADMM算法(SSP-ADMM)

输入：（进程数），,,

第一步：每个线程从参数服务器中得到：

第二步：如果是旧值，则，否则每个进程更新本地：

第三步：每个进程更新：

第四步：每个进程更新：

第五步：每个进程更新：

第六步：计算差值：

第七步：将推送到参数服务器端：

第八步：判断是否满足停机准则，如果满足，则停止迭代，否则迭代次数加1(调用clock)并跳到第一步

首先初始化所有参数值包括参数服务器的参数值为0，第一步每一个线程会从参数服务器端读取到也是就第k次迭代所有线程保存在参数服务器中和的和，如果是旧值，则令，否则通过更新，然后每一个线程更新本地的和，并通过提交差值，将服务器每一个进程的和更新到当前迭代计算的最新值，然后进入下次迭代，直到满足结束条件，算法结束。

这里有两个参数服务器的操作，PS.read\_row操作是从参数服务器或者本地cache读取数据，实际从哪里读取Petuum会根据SSP模型和当前运行情况来判断，不需要用户干预，另一个PS.inc操作向参数服务器提交一个增量，这里本来的需求向参数服务器提交本地最新的参数，因此PS.inc最新值和上一次迭代的差值，这样一旦提交成功，参数服务器和本cache都会更新成最新值。

### 一致性变量的存储

在SSP模型下，每一个线程读到的参数服务器table的数据可能不一致，在某一时刻，有的线程可能只读到本地cache中的旧值，而有些线程可能读到远程服务器的新值，因此对于任意一个线程 获取到的包括了以下两个的部分的累加：

1、本地上一次更新的，这是由“read-my-write”特性决定的。

2、其他线程 第次更新的，其中。

每一个线程计算出的可能不相等，主要取决于线程是读取了本地的cache还是读取到了远程参数服务器的table，而这又关系到系统中每一个线程的执行进度，因为由SSP模型的条件1和条件4可知，当线程将要突破设定的时间窗口或者是线程时钟是时间窗口整数部时，都会使得该线程主动去读取参数服务器table中的数据，此时是相等的，并且这里“相等”的意思是线程可能读到这个，但不是一定是在同一时钟上，另外如果staleness值为0时，SSP-ADMM算法将会退化成syn-ADMM算法。

## 收敛性分析

在分析收敛性时，我们作一个标准的假设和对参数服务器迭代次数作一个标准的定义，方便收敛性的讨论。

**假设1：**

**每一个存在二阶导，并且是利普希茨连续，且利普希茨常量为，是凸函数，且的值域是紧集。**

**定义1：**

**从第0次迭代开始，下一次迭代次数的计数从当参数服务器最早接收到worker的参数更新开始。每次迭代中表示在第k次读到参数服务器的worker编号，而表示没有读到参数服务器worker的编号，另外，在分成两个子集，其中表示读到服务器数据的worker，而表示没有读到服务器的worker，当worker 读到本地的cache时，使用上次读到的服务器得到的旧值来作为下一次迭代计算.**

因此

又因为当足够大时，可使成为关于的强凸函数，我们设此时的强凸系数为【71】，则：

由最优性条件可得：

由于当时

当时

当时

综上所述

又因为成为关于的强凸函数，我们设此时的强凸系数为，则：

由于是一个凸函数则：

所以

则

综上所述

**引理1：在给定的假设1成立时，**

证明：

由最优性条件可知：

同时当对偶变量更新时，由最优性条件可知：

代入上式可得

因此：

又由于是利普希茨连续，且利普希茨常量为即

则

引理1得证。

下面我们来化简

由杨式不等式

其中，可得

则

由于

其中由于SSP模型的staleness限制，

从至少出现次，则

所以

将从0累加到第次化简可得：

**引理2：在给定的假设1成立时**， **假设是异步ADMM算法产生的点，则存在**

证明：

是利普希茨连续则

所以

即存在下界

则

当满足

上式左边是恒大于0随着的增加，而上式左边是有限的，所以当时，。

这里由于是成为关于的强凸系数，由于是利普希茨连续，则的Hessian矩阵的最小奇异值大于【71】，因此至少，才能保证是一个关于的强凸函数，且；同理。

则

得

综上所述，当满足上两式时，，，则由得

又由最优性条件可知：

则

所以以上三式可以证明异步ADMM迭代所产生的点满足KKT条件，另外假设中的值域是紧集且，表示的值，则是有界序列和有限点，也同样满足。

当为凸函数时，只需要即能保证关于的强凸函数，则，其他条件不变，则

即

## 本章小结

本章详细的介绍了SSP模型，并且分析了基于MPI的同步分布式ADMM算法的实现，然后通过SSP模型设计和实现了异步的ADMM算法，最后从理论上分析了目标函数为非凸和凸两种场景下算法的收敛性。

# 实验

本章在上海大学自强4000集群环境下针对本文第四章所提出基于SSP模型的异步分布式ADMM算法进行测试和分析，验证理论收敛分析的结果以及高效性，并对基于SSP模型与基于MPICH的ADMM算法的性能进行了对比分析。

## 实验测试环境

本论文的实验环境为基于上海大学自强4000上16个节点的集群环境，节点使用双核四线程的Intel Core i3-2130 64位CPU，主频3.40GHz、内存4GB、每个节点拥有一个500GB的本地磁盘。节点装载Ubuntu 14.04 64位操作系统，所安装的软件有Petuum 、MPICH 3.1.0、mpi4py、HDFS 1.2.1 、JDK 1.7.0 、Gcc 、make 、ssh、mpi4py、python2.7等。各节点之间配置了SSH无密码登录，安装了HDFS，HDFS中的数据备份数为3，实验代码通过C++和python实现。

具体的硬件及软件配置如表5.1-1及表5.1-2所示。

表5.1-1集群节点硬件配置列表

|  |  |
| --- | --- |
| **项目** | **参数** |
| CPU | Intel Core i3-2130/64位/3.40GHz |
| 核数 | 双核四线程 |
| 内存 | 4GB |
| 硬盘 | 500G/7200转 |

表5.1-2 实验集群节点软件环境列表

|  |  |
| --- | --- |
| **项目** | **参数** |
| 操作系统 | Ubuntu 14.04 64位 |
| 所需软件 | Petuum  MPICH 3.1.0  HDFS 1.2.1  JDK 1.7.0  Gcc , make , ssh |

## 实验应用与实验数据

我们选择Lasso问题也就是带L1正则的线性回归作为实验的应用，L1正则是机器学习中使用较为广泛的一种正则化方法，一方面，可以防止模型过拟合，另一方面，由于L1可以得到稀疏解，所以还可以用作模型选择，其具体的问题形式如下：

其中，,,.

在实验数据方面，特征矩阵按均值为0，标准差为1的高斯分布进行随机产生，,其中,是一个大约有0.1n个非0条目的随机向量，是一个满足均值为0，标准差为0.01的高斯分布的噪音向量。

## 基于SSP模型的异步分布式ADMM算法收敛性测试

本节就第四章基于SSP模型的异步分布式ADMM算法理论收敛性分析结果进行实验验证，实验分成目标函数为凸和非凸两种情况，并且根据这两种情况的测试结果进行分析。

### 目标函数为凸和强凸时算法收敛性测试

将特征矩阵的特征数n降到200和100也就是特征矩阵为和分别进行测试，测试方式同非凸目标函数，测试结果如下：



图 5.3‑1 目标函数为凸和强凸时算法的收敛性测试结果



图 5.3‑2 目标函数为凸和强凸时算法的收敛性测试结果

### 目标函数为非凸时算法收敛性测试

我们通过上一节的数据生成的方法去生成64个的特征矩阵分配到16个节点上，每一个节点4个线程去处理。测试分成两种场景进行，第一个场景是固定时间窗口值来变化值进行收敛性测试，第二个场景是固定值来变化时间窗口值进行测试，测试结果如下：



图 5.3‑3 标函数为非凸时算法的收敛性测试结果

### 收敛性测试结果与分析

当目标函数为非凸时，场景一中选取，值在0.001、0.01、1时SSP-ADMM算法都难以收敛，只有当值在500时，算法能正常收敛，这个结果和理论收敛性分析的结果是一致的，在目标函数非凸时，需要有足够大的值保证算法的收敛性；场景二中，我们设置值为500，测试了staleness为1、5、10、50，图显示算法都能正常收敛，但是从收敛曲线来看，时间窗口值越大，算法收敛速度越慢。

当目标函数为凸时，测试了200x200和200x100两种规模的特征矩阵，场景一中同样设置时间窗口值，值在0.001、0.01、1时SSP-ADMM算法都难以收敛，只有当值在500时，算法能正常收敛，这个结果和非凸是一致的，也和理论分析结果一致；场景二中算法正常收敛，但是纵向比较显示，值越大算法收敛越慢，横向比较同样参数，特征数为100时收敛速度比特征数为200时稍快。

## 基于SSP模型的异步分布式ADMM算法性能测试

本节就第四章基于SSP模型的异步分布式ADMM算法的性能进行实验验证，具体的实验通过两个项目进行测试，第一是算法计算和通信比的测试；第二是算法的精度测试。

### 算法通信时间的测试

异步相比同步的优势在于少量的同步开销，本节主要是测试算法计算时间和通信时间，测试方式是通过调整不同staleness值来测试。测试样本是随机生成1000x1000的特征矩阵，也就是总样本数量为64000x1000，将近1.6G，，，并且测试使用的参数能够保证算法正常收敛，同时算法是以迭代次数为停机标准，固定迭代50次，测试结果如下图。

图 5.4‑1 基于SSP模型的异步分布式ADMM算法性能测试结果

### 算法精度测试

由于测试应用是一个回归应用，所以我们采用RMSE(均方根误差)来测试其精度，测试方式是通过调整不同staleness值来测试，实验样本和参数与性能测试样本与参数一样，并且算法是以迭代次数为停机标准，对固定迭代次数为100次和500次分别进行测试，测试结果如下图。



图 5.4‑2 基于SSP模型的异步分布式ADMM算法RMSE测试结果

### 性能测试结果与分析

从图 5.4‑1当时，SSP-ADMM会退化成sync-ADMM，因此，可以通过对比较时和其他值来对比同步和异步分布式ADMM算法，从性能测试结果可以看出，当staleness值越大，通信时间占总算法时间比越少，由于算法执行相同的迭代次数，计算时间不变，算法总运行时间越来越少，所以从实验结果来看异步比同步分布式ADMM算法更高效。

精度测试结果如图 5.4‑2中，固定迭代次数为100时，RMSE会随着staleness值越来越大，固定迭代次数为500时，staleness的增加不会对RMSE有所影响。这是由于足够的迭代次数可以使得算法最终能收敛到最优解，因此不同的staleness值时算法均能够在500次迭代后收敛，而当迭代次数为100时，不同的staleness值就会有较大的差别，这个结果和收敛性测试结果是一致的。实际应用中，staleness越大，通信时间越少，但是低质量的迭代的数量越会多，所以需要注意平衡算法迭代的质量和数量。

## 基于Petuum和基于MPI的ADMM算法对比实验

Petuum和MPI有很多异同点， Petuum中模型并行模块是通过MPI来实现的，数据并行是基于参数服务器框架，并且是基于线程，而MPI是消息传递接口，基于进程的。Petuum是面向机器学习算法的分布式框架，而MPI可以实现除了机器学习以外大多数算法。

前几节的测试全部是在Petuum平台上进行的，本节将对比基于Petuum和基于MPI的ADMM算法，来比较两个平台的差异。实验数据使用上一节性能测试的数据集，基于MPI的同步分布式ADMM算法通过mpi4py和numpy实现，测试方式，固定迭代次数50次，，，测试结果如表格 5.5‑1：

表格 5.5‑1 基于Petuum和基于MPI的ADMM算法对比测试结果

|  |  |  |  |
| --- | --- | --- | --- |
|  | staleness | 计算时间 | 通信时间 |
| MPICH |  | 311031 | 48507 |
| Petuum | 0 | 60050 | 62872 |
| 5 | 59843 | 50733 |
| 10 | 59797 | 35004 |
| 15 | 59339 | 19795 |
| 20 | 60893 | 2085 |

单位：毫秒

从表格数据看出，由于基于MPI的ADMM算法是python实现的，在计算时间上比C++实现的基于Petuum的分布式ADMM算法要多；另外使用MPICH的AllReduce函数在通信效率上比staleness值在小于10时要高效，当staleness大于10时，Petuum上的本地cache访问次数增多，此时基于Petuum的分布式ADMM算法的通信效率较高。

## 本章小结

从本章的实验可以得出，收敛性测试的结果和上一章理论收敛性分析结果是一致的，性能测试的结果中，时间窗口值越小，算法运行的通信时间越少，但从精度测试来看，固定迭代次数足够大时，时间窗口值的改变对RMSE没有影响，否则会有影响；最后在基于Petuum和MPI分布式ADMM算法的对比实验显示，当时间窗口值超过一定值时，基于Petuum分布式ADMM算法比MPI的通信效率要高。

# 结论和展望

## 总结

本文首先通过一个二项树算法的实例来介绍MPICH并行环境下进程分布对于并行算法的影响，同时研究的同步分布式ADMM算法面临较大的同步开销和通信开销，因此，本文从通信优化角度出发，提出了分层并行设计模式和异步分布式ADMM算法。本文研究的具体的核心内容归纳以下几个方面。

（1）提出一种分层并行设计模式，该设计模式包括了多核感知的MPI中间件、层次化通信模型的构建、通信优化策略、数据预处理的方法。通过多核感知的MPICH中间件为并行算法构建高效的层次化通信模型和通信优化策略来提高通信效率，以及数据预处理方法可以利用机群中的大量数据存储节点同时访问数据，以此利用分布集群中大量节点上的磁盘集合提供高带宽的数据访问和传输。

（2）提出了一种动态的层次化通信模型-层次蝶式通信模型应用到异步分布式ADMM。层次蝶式通信模型首先在网络通信上区分了节点内通信和节点间通信，其次节点间通信采用了蝶形通信模型，层次化的通信模型能大大提高并行算法的可扩展性，节点间的蝶形通信模型能够减少全局通信时间，并且在理论上分析了应用三种通信模型的算法的收敛性。

（3）提出了一种基于SSP模型的异步分布式ADMM算法，利用开源Petuum框架数据并行的SSP计算模型设计一种异步分布式ADMM算法来减少在集群环境下同步ADMM算法每次迭代中的同步开销，并且给予理论收敛性的证明，从实验部分对算法的收敛性和性能测试结果和分析可以看出，算法的收敛测试结果与理论分析一致同时SSP模型中时间窗口值越小时算法通信开销越小，但是在固定迭代次数较小时，需要去注意算法的精度问题。

## 进一步研究方向

本文在并行算法的设计模式和异步分布式ADMM算法的研究方面做了一些工作，一定程度上提高了算法性能。但是从应用角度来说，本文的研究工作仍需进一步的深入与完善，在阅读相关文献的基础上，本人对今后的研究工作整理如下：

（1）分层并行设计模式中的数据预处理方法比较粗糙，对超大块的数据预处理比较耗时，而与HDFS相结合的数据本地化方法限制太多，实用性不高，因此后面的工作可以更多关注如何去开发一套适应于MPI的分布式文件系统，同时分层并行编程模型比传统的设计模式更加复杂，所以如何简化编程模型也是今后比较重要的工作内容。

（2）本文对数据并行的分布式ADMM算法的重点研究，但对于模型并行的分布式ADMM算法以及数据并行和模型并行相结合的分布式ADMM算法没有进行讨论，而大数据背景下，模型的参数普遍较高，如何通过模型并行的分布式ADMM算法解决高维模型的分布式计算问题。

（3）Petuum的模型并行模块可以和ADMM算法相结合可以高效的解决高维模型的问题，同时目前较流行的Parameter Server也可以用来设计数据并行和模型并行相结合的分布式ADMM算法。

参考文献

1. Argonne National Laboratory. <http://www-unix.mcs.anl.gov/mpi/mpich>
2. 黄宜华，苗凯翔. 深入理解大数据. 北京:机械工业出版社，2014. 456
3. Faraj A, Patarasuk P, Yuan X. A Study of Process Arrival Patterns for MPI Collective Operations[J]. International Journal of Parallel Programming, 2008, 36(6):543-570.
4. Boyd S, Parikh N, Chu E, et al. Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers[J]. Foundations & Trends in Machine Learning, 2010, 3(1):1-122.
5. Dai W, Wei J, Zheng X, et al. Petuum: A Framework for Iterative-Convergent Distributed ML[J]. 2013.
6. Ho Q, Cipar J, Cui H, et al. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server[J]. Advances in Neural Information Processing Systems, 2013, 2013(2013):1223-1231.
7. Shi W, Ling Q, Yuan K, et al. On the Linear Convergence of the ADMM in Decentralized Consensus Optimization[J]. IEEE Transactions on Signal Processing, 2014, 62(7):1750-1761.
8. Mota J F C, Xavier J M F, Aguiar P M Q, et al. D-ADMM: A Communication-Efficient Distributed Algorithm For Separable Optimization[J]. IEEE Transactions on Signal Processing, 2012, 61(10):2718-2723.
9. Valiant L G. Bulk-synchrony: a bridging model for parallel computation[J]. Communications of the Acm, 1990, 33(8):103-111.
10. Dean J, Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters.[J]. In Proceedings of Operating Systems Design and Implementation (OSDI, 2004, 51(1):107-113.
11. Lubell-Doughtie P, Sondag J. Practical distributed classification using the Alternating Direction Method of Multipliers algorithm[C]// IEEE International Conference on Big Data. 2013:773-776.
12. R. Zhang and J. T. Kwok, “Asynchronous distributed ADMM for consensus optimization,” in Proc. 31th ICML, , 2014., Beijing, China, June 21-26, 2014, pp. 1–9.
13. T.-H. Chang, M. Hong, W.-C. Liao, and X. Wang, “Asynchronous distributed ADMM for large-scale optimization- Part I: Algorithm and convergence analysis,” submitted for publication
14. Wei E, Ozdaglar A. On the O(1/k) Convergence of Asynchronous Distributed Alternating Direction Method of Multipliers[C]// IEEE Global Conference on Signal & Information Processing. Institute of Electrical and Electronics Engineers (IEEE), 2013:551-554.
15. 陈国良, 并行算法的设计与分析[A]. 高等教育出版社, 2009
16. 霍旭光, 基于MPI高性能计算方法的研究[D], 北京：中国地质大学, 2006
17. Fortune S, Wyllie J. Parallelism in random access machines. Proceeding of the tenth annual ACM symposium on Theory of architectures. ACM,1978. 114-118
18. R.Cole and O.Zajicek. The APRAM: Incorporating Asynchrony into the PRAM model. In Proceedings of the first annual ACM symposium on Parallel algorithms and architectures. ACM, 1989. 169-178.
19. Flynn M J.Very high-speed computing systems[J]. IEEE, 1966,54(4):1901一1909
20. Culler D, Karp R, Patterson D. LogP: Towards a realistic model of parallel computation, vol. 28. ACM,1993
21. Cameron, K.W, Xian-He Sun. Quantifying locality effect in data access delay: memory logP[C], Parallel and Distributed Processing Symposium, 2003. Proceedings. International, 2003, DOI:10.1109/IPDPS.2003.1213137:1530~2075
22. 张云泉, 面向高性能数值计算的并行计算模型DRAM(h)[J], 计算机学报, Vol .26 No .12, 2003
23. X. Qiao, S. Chen, and L.T. Yang. HPM: a Hierarchical Model for Parallel Com-putations[J]. International Journal of High Performance Computing and Network-ing, 1(1):117-127, 2004.
24. 计永爬, 丁卫群, 陈国良, 安虹. 一种实用的并行计算模型[J]. 计算机学报, Vo1.24 No.4,2001(4):437-441
25. S.A. Cook and R.A. Reckhow. Time Bounded Random Access Machines[J]. Jour-nal of Computer and System Sciences, 7(4):354-375, 1973.
26. 方维, 面向数据通信优化的并行程序执行模型[D], 合肥: 中国科学技术大学, 2011
27. 陈国良, 苗乾坤. 分层并行计算模型[J].中国科学技术大学学报, 38(7):841-847,2008.
28. Gabriel E, Fagg G E, Bosilca G, et al. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation.[J]. Proceedings European Pvm/mpi Users’ Group Meeting, 2004, 3241:97--104.
29. Graham R L, Woodall T S, Squyres J M. Open MPI: a flexible high performance MPI[C]// In International Conference on Parallel Processing & Applied Mathematics. Springer-Verlag, 2005:228-239.
30. Graham R L, Shipman G M, Barrett B W, et al. Open MPI: A High-Performance, Heterogeneous MPI[C]// Cluster Computing, 2006 IEEE International Conference on. IEEE, 2006:1-9.
31. Graham R L. A Comparison of Application Performance Using Open MPI and Cray MPI[J]. Grbovic, 2007.
32. Bukhamsin A, Sindi M, Al-Jallal J. Using the Intel MPI benchmarks (IMB) to evaluate MPI implementations on an Infiniband Nehalem Linux cluster.[C]// Spring Simulation Multiconference. 2010:1-4.
33. Huang W, Santhanaraman G, Jin H W, et al. Design of High Performance MVAPICH2: MPI2 over InfiniBand[C]// Cluster Computing and the Grid, IEEE International Symposium on. IEEE Computer Society, 2006:43-48.
34. Banikazemi M, Govihdaraju R K, Blackmore R, et al. MPI-LAPI: An Efficient Implementation of MPI for IBM RS/6000 SP Systems[J]. Parallel & Distributed Systems IEEE Transactions on, 2001, 12(10):1081-1093.
35. Knoth A. Open MPI[M]// Grid-Computing. Springer Berlin Heidelberg, 2008:117-126.
36. Buntinas D, Mercier G, Gropp W. Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem[C]//Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on. IEEE, 2006, 1: 10 pp.-530.
37. Buntinas D, Mercier G, Gropp W. Implementation and evaluation of shared-memory communication and synchronization operations in MPICH2 using the Nemesis communication subsystem[J]. Parallel Computing, 2007, 33(9): 634-644.
38. Buntinas D, Goglin B, Goodell D, et al. Cache-efficient, intranode, large-message MPI communication with MPICH2-Nemesis[C]//Parallel Processing, 2009. ICPP'09. International Conference on. IEEE, 2009: 462-469.
39. Koop M J, Jones T, Panda D K. MVAPICH-Aptus: Scalable high-performance multi-transport MPI over InfiniBand[C]//Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on. IEEE, 2008: 1-12.
40. 郭旭东, 冯萍, 康继昌, 等. 先锋光纤通道交换网通信性能优化的研究与实现[J]. 计算机测量与控制, 2009 (1): 135-137.
41. 郑魁. 先锋光纤通道交换网 MPI 通信库的设计[D]. 西北工业大学, 2007.
42. Chai L, Lai P, Jin H W, et al. Designing an efficient kernel-level and user-level hybrid approach for mpi intra-node communication on multi-core systems[C]//Parallel Processing, 2008. ICPP'08. 37th International Conference on. IEEE, 2008: 222-229.
43. Thakur R, Gropp W D. Improving the performance of collective operations in MPICH[M]//Recent Advances in Parallel Virtual Machine and Message Passing Interface. Springer Berlin Heidelberg, 2003: 257-267.
44. Faraj A, Yuan X. Automatic generation and tuning of MPI collective communication routines[C]//Proceedings of the 19th annual international conference on Supercomputing. ACM, 2005: 393-402.
45. 张攀勇, 孟丹, 霍志刚. 多核环境下高效集合通信关键技术研究[J]. 计算机学报, 2010, 33(2): 317-325.
46. 闫浩. MPICH 在工作站机群环境下通信机制的分析与优化[D]. 解放军信息工程大学, 2010.
47. <http://hadoop.apache.org/>
48. Xiaoyi Lu, Bing Wang, Li Zha, and Zhiwei Xu. Can MPI benefit Hadoop and MapReduce applications [C] 2011 International Conference on Parellel Processing Workshops, Taipei City, DOI: 10.1109/ICPPW.2011.56:371~379
49. Ranger C, Raghuraman R, Penmetsa A, et al. Evaluating MapReduce for Multi-core and Multiprocessor Systems[J]. 2007:13-24.
50. Kaashoek F, Morris R, Mao Y. Optimizing MapReduce for Multicore Architectures[J]. 2010.
51. Zaharia M, Chowdhury M, Franklin M J, et al. Spark: Cluster Computing with Working Sets[J]. 2010:10-10.
52. Spark[EB/OL]. <http://spark.incubator.apache.org/>.
53. Olsen B, Mckenney M. Storm System Database: A Big Data Approach to Moving Object Databases[C]// Fourth International Conference on Computing for Geospatial Research and Application. 2013:142-143.
54. GitHub[EB/OL]. <https://github.com/nathanmarz/storm/wiki/Tutorial>.
55. Tipparaju V, Nieplocha J, Panda D. Fast collective operations using shared and remote memory access protocols on clusters[J]. International Parallel & Distributed Processing Symposium, 2003:84a.
56. Traff J L. Hierarchical Gather/Scatter Algorithms with Graceful Degradation[C]// Parallel and Distributed Processing Symposium, International. IEEE Computer Society, 2004:80-80.
57. Karonis N T, De Supinski B R, Foster I, et al. Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance[C]// Parallel and Distributed Processing Symposium, International. IEEE Computer Society, 2000:377-377.
58. Li S, Hoefler T, Hu C, et al. Improved MPI collectives for MPI processes in shared address spaces[J]. Cluster Computing, 2014, 17(4):1-17.
59. Vapnik V, Golowich S E, Smola A. Support Vector Method for Function Approximation, Regression Estimation, and Signal Processing[J]. Advances in Neural Information Processing Systems, 1970, 9:281--287.
60. Christara C, Ding X, Jackson K. An Efficient Transposition Algorithm For Distributed Memory Computers[J]. International, 1999, 541:349-370.
61. Huang G B, Zhu Q Y, Siew C K. Extreme learning machine: a new learning scheme of feedforward neural networks[C]// Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on. IEEE, 2004:985-990 vol.2.
62. Huang G B, Zhou H, Ding X, et al. Extreme learning machine for regression and multiclass classification.[J]. IEEE Transactions on Systems Man & Cybernetics Part B Cybernetics A Publication of the IEEE Systems Man & Cybernetics Society, 2012, 42(2):513 - 529.
63. Tam A T C, Wang C L. Efficient Scheduling of Complete Exchange on Clusters[C]// The 13th International Conference on Parallel and Distributed Computing Systems (PDCS 2000). ISCA., 2000:111-116.
64. Kale L V, Kumar S, Varadarajan K. A framework for collective personalized communication, communicated to ipdps 2003[C]// In Proceedings of IPDPS’03. 2002:69a.
65. <http://sourceforge.net/projects/openblas/files/>
66. <http://www.dcc.fc.up.pt/~ltorgo/Regression/DataSets.html>
67. Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.
68. Deng, Wei, and Wotao Yin. "On the global and linear convergence of the generalized alternating direction method of multipliers." Journal of Scientific Computing (2012): 1-28.
69. Li M, Andersen D G, Park J W, et al. Scaling distributed machine learning with the parameter server[C]// International Conference. 2014:1-1.
70. <https://github.com/dmlc/ps-lite>
71. Nesterov Y. Introductory Lectures on Convex Optimization[J]. 2007, 87(5):xviii,236.

作者在攻读硕士学位期间公开发表的论文

【1】. 方铃，雷咏梅. 一种高效MPI设备层扩展库的设计与实现[J]. 计算机技术与发展.

【2】. Ling Fang, Yongmei Lei. An Asynchronous Distributed ADMM Algorithm and Efficient Communication Model [C], The 2016 Smart World Congress, Beijing, DOI 10.1109/UIC-ATC-ScalCom-CBDCom-IoP.2015.327:1803-1808

作者在攻读硕士学位期间所作的项目

致 谢

本文是在导师雷咏梅老师的悉心指导下完成的。承蒙雷老师的亲切关怀和精心指导，虽然有繁忙的工作，但仍抽出时间给予我学术上的指导和帮助，特别是给我提供了良好的学习环境，使我从中获益不浅。雷老师对学生认真负责的态度、严谨的科学研究方法、敏锐的学术洞察力、勤勉的工作作风以及勇于新、勇于开拓的精神是我永远学习的榜样。在此，衷心感谢雷卫民老师两年半的栽培和指导，谨向雷老师致以深深的敬意和由衷的感谢，衷心祝愿她心想事成，万事如意！

同时还要感谢课题组每位导师和实验室的每位同学，是你们让我感觉到融合到一个大家庭中，温馨融洽。感谢吉成恒、刘建军、周鹏飞、张佳峰、王舒睛。感谢1010的成员们，感谢你们陪我走过了一段珍贵的研究生生活。感谢各位师兄师姐，是你们的文章让我更好更快的进入研究状态，你们为我树立了一个好的榜样，值得我一直追寻。感谢我的室友，谢谢你们的帮助和爱护。

还要感谢我的父母我的兄弟姐妹他们在生活上给予我很大的支持和鼓励，是他们给予我努力学习的信心和力量。

最后，感谢所有关心我、支持我和帮助过我的同学、朋友、老师和亲人。

在这里，我仅用一句话来表明我无法言语的心情：感谢一路上有你们！