分类号 密级

UDC

学 位 论 文

异步大规模图处理框架Maiter的负载均衡技术及累积迭代算法的研究

|  |  |  |  |
| --- | --- | --- | --- |
| 作者姓名： | 王春磊 | | |
| 指导教师： | 于戈 教授 | | |
|  | 东北大学计算机软件研究所 | | |
| 申请学位级别： | 硕士 | 学科类别： | 工学 |
| 学科专业名称： | 计算机软件与理论 | | |
| 论文提交日期： | 2015年6月 | 论文答辩日期： | 2015年6月 |
| 学位授予日期： | 2015年7月 | 答辩委员会主席： | 朱靖波 |
| 评阅人： |  | | |

东 北 大 学

2015年6月

**A Thesis in Computer Software and Theory**

**The Research of Load Balance Techniques and Accumulative Iterative Algorithms** **in Large Scale Asynchronous Graph Processing Framework Maiter**

By Wang Chunlei

Supervisor: Professor Yu Ge

**Northeastern University**

**June 2015**

# 独创性声明

本人声明，所呈交的学位论文是在导师的指导下完成的。论文中取得的研究成果除加以标注和致谢的地方外，不包含其他人己经发表或撰写过的研究成果，也不包括本人为获得其他学位而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均己在论文中作了明确的说明并表示谢意。

学位论文作者签名：

日 期：

学位论文版权使用授权书

本学位论文作者和指导教师完全了解东北大学有关保留、使用学位论文的规定：即学校有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人同意东北大学可以将学位论文的全部或部分内容编入有关数据库进行检索、交流。

作者和导师同意网上交流的时间为作者获得学位后：

半年 □ 一年□ 一年半□ 两年□

学位论文作者签名： 导师签名：

签字日期： 签字日期：

异步大规模图处理框架Maiter的负载均衡技术及累积迭代算法的研究

# 摘 要

随着电子商务、社交网络的发展，大规模分布式图处理的作用越来越重要，被广泛的应用在链接分析、产品推荐等应用中。Maiter框架作为完全异步的大规模分布式图处理框架，通过采用DAIC计算模型，避免了同步开销，提升了收敛速度，极大的提高了大规模图处理的效率。为了进一步的提高Maiter框架可用性和通用性，本文的工作在两个方面展开。

在Maiter框架可用性方面，本文采用集中式动态负载均衡机制解决了Maiter框架的负载均衡问题。在负载均衡机制实现的过程，解决了诸多难题。1)提出了基于Hash数据块的数据管理方式和消息分桶标记定位机制，极大的方便了负载均衡处理过程中的数据迁移和迁移数据重定位。2)从理论上证明了Maiter框架在算法计算的过程中进行数据迁移，迁移数据计算被拖延对整个集群的影响并不会影响最终的计算结果，从理论上保证了计算过程中数据迁移的正确性。3)提出了基于缓存的错位消息中转机制，来处理数据迁移的过程中产生的错位消息。并从理论上证明了该机制的正确性。4)在以上工作的基础之上，在Maite框架上实现了基于数据块的集中式动态负载均衡机制。实验结果显示，该机制在集群不存在负载倾斜时，几乎不会增加系统的开销。存在负载倾斜时，可有效的处理负载问题，并提升框架整体的计算效率。

在Maiter框架的通用性方面，本文采用了DAIC计算模型的思想，改进了传统的SimRank算法，1) 本文提出Asyn-SimRank算法，该算法采用迭代-累积的方式完成迭代计算，异步执行SimRank的核心迭代过程，避免了大规模分布式计算中的大量同步开销，同时有效降低计算量并减少通信开销；2)提出关键点优先调度计算，提升了Asyn-SimRank算法的全局收敛速度。3)证明了Asyn-SimRank算法的正确性和收敛性以及关键点优先调度计算的有效性。4)在支持异步迭代的分布式框架Maiter上实现了Asyn-SimRank算法。实验结果显示，相比较于Hadoop，Spark上实现的SimRank算法和Delta-SimRank算法，Asyn-SimRank算法大大提升了算法的计算效率，加速了算法收敛。

经过本文的一系列工作，Maiter框架的通用性和可用性进一步提升，为Maiter框架的实际应用创造了有利的条件。

关键词：负载均衡；异步计算；迭代计算；Asyn-SimRank算法；MapReduce模型；

The Research of Load Balance Techniques and Accumulative Iterative Algorithms in Large Scale Asynchronous Graph Processing Framework Maiter

# Abstract

With the development of electronic commerce and social networks, the role of large-scale graph processing distributed become more and more important. It is widely used on the applications of link analysis, product recommendations, and so on. Maiter as a fully asynchronous large-scale graph processing distributed framework, through the adoption of DAIC calculation model, avoids synchronization overhead, increase the convergence speed, and greatly improve the efficiency of large graph processing. In order to further improve usability and universality of Maiter, we carry out the work in two aspects.

On usability of Maiter, we adopt the centralized dynamic load balancing mechanism to solve the load balancing problem of Maiter. In the implemention of load balance mechanism, we solve many problems. 1) We propose the data management based on the Hash data blocks and message management mode based on Hash assigning to bucket, great convenience to data migration and migration data relocation in the process of load balancing. 2) We prove that the computation delay of migration data as a result of data migration in the process of calculation, desn’t affect the final computation results. 3) We propose message transfer mechanism based on the cached to deal with the messages on the flying in the process of data migration, and prove the correctness of the proposed mechanism in theory. 4) On the basis of above work, we implement the centralized dynamic load balancing mechanism based on data block on Maiter. The experimental results show that the load balancing mechanism almost does not increase the work load when skew does not exist in the cluster and can be effective to deal with the load skew and improve the whole computational efficiency of the Maiter.

On universality of Maiter, 1) we propose Asyn-SimRank, which adopts the iterate-cumulate method, asynchronously executing the core iterative process in order to avoid the high-cost synchronization barriers in large-scale distributed environments, and effectively reduce the amount of computation and communication. 2) We propose the keypoint-prior computation to accelerate global convergence. 3) We prove the accuracy and the convergence of the Asyn-SimRank algorithm and efficiency of the keypoint-prior computation. 4) We then implement Asyn-SimRank on Maiter, which is a distributed framework supporting asynchronous iteration. Our results show that, comparing with the SimRank and Delta-SimRank implementing on Hadoop and Spark, the large-scale Asyn-SimRank significantly promotes the computational efficiency and accelerates the convergence.

Though a series of work this paper, we further improve the usability and universality of Maiter framework, created favorable conditions for the practical application of Maiter framework.

**Key Words:** load balancing; data migration; asynchronous computation; iterative computation; Asyn-SimRank; similarity; big data; MapReduce; Maiter;

目 录

[独创性声明 I](#_Toc422315332)

[摘 要 III](#_Toc422315333)

[Abstract V](#_Toc422315334)

[第1章 绪论 1](#_Toc422315335)

[1.1 研究背景 1](#_Toc422315336)

[1.1.1 分布式计算 1](#_Toc422315337)

[1.1.2 分布式算法 2](#_Toc422315338)

[1.2 目的和意义 3](#_Toc422315339)

[1.3 国内外研究现状 4](#_Toc422315340)

[1.4 主要研究内容 5](#_Toc422315341)

[1.4.1 Maiter框架的负载均衡处理问题的研究 6](#_Toc422315342)

[1.4.2 异步累积迭代算法的研究 7](#_Toc422315343)

[1.5 论文章节安排 7](#_Toc422315344)

[第2章 DAIC模型及Maiter框架 9](#_Toc422315345)

[2.1 传统同步迭代计算 9](#_Toc422315346)

[2.2 传统异步迭代计算 10](#_Toc422315347)

[2.3 DAIC计算模型 10](#_Toc422315348)

[2.3.1 DAIC计算模型的前提条件 10](#_Toc422315349)

[2.3.2 DAIC计算模型的推导 12](#_Toc422315350)

[2.3.3 DAIC计算模型的优先级迭代 13](#_Toc422315351)

[2.4 Maiter框架 14](#_Toc422315352)

[2.4.1 Maiter框架的实现 14](#_Toc422315353)

[2.4.2 Maiter框架的优先级迭代的实现 16](#_Toc422315354)

[2.4.3 Maiter框架的API 17](#_Toc422315355)

[2.5 本章小结 18](#_Toc422315356)

[第3章 Maiter框架的负载均衡技术 19](#_Toc422315357)

[3.1 负载不均衡的原因 19](#_Toc422315358)

[3.1.1 同步框架负载不均衡的原因 19](#_Toc422315359)

[3.1.2 Maiter框架负载不均衡的原因 20](#_Toc422315360)

[3.2 负载均衡方案的选择 21](#_Toc422315361)

[3.3 负载均衡处理决策算法 23](#_Toc422315362)

[3.3.1 决策算法设计思路 23](#_Toc422315363)

[3.3.2 负载均衡决策算法 24](#_Toc422315364)

[3.4 数据迁移 27](#_Toc422315365)

[3.4.1 数据迁移产生的问题 27](#_Toc422315366)

[3.4.2 对Maiter框架数据管理方式的改进 28](#_Toc422315367)

[3.4.3 数据迁移的正确性保证 29](#_Toc422315368)

[3.5 迁移数据重定位 30](#_Toc422315369)

[3.6 错位消息处理 32](#_Toc422315370)

[3.6.1 错位消息的产生及影响 32](#_Toc422315371)

[3.6.2 基于缓存的消息中转解决方案 33](#_Toc422315372)

[3.6.3 基于缓存的消息中转机制正确性证明 34](#_Toc422315373)

[3.7 负载均衡处理详细过程 36](#_Toc422315374)

[3.8 本章总结 38](#_Toc422315375)

[第4章 异步累积迭代算法 39](#_Toc422315376)

[4.1 研究内容介绍 39](#_Toc422315377)

[4.2 SimRank算法原理与Hadoop实现 40](#_Toc422315378)

[4.2.1 SimRank算法原理 40](#_Toc422315379)

[4.2.2 Hadoop框架上的实现 42](#_Toc422315380)

[4.3 Asyn-SimRank算法 44](#_Toc422315381)

[4.3.1 同步迭代-累积形式—Delta-SimRank 44](#_Toc422315382)

[4.3.2 异步迭代-累积形式—Asyn-SimRank 45](#_Toc422315383)

[4.3.3 收敛性与正确性证明 47](#_Toc422315384)

[4.4 关键点优先调度 49](#_Toc422315385)

[4.4.1 基本思想 49](#_Toc422315386)

[4.4.2 理论证明 50](#_Toc422315387)

[4.5 Asyn-SimRank算法的分布式实现 51](#_Toc422315388)

[4..5.1 输入图的预处理 51](#_Toc422315389)

[4.5.2 分布式环境上的实现 51](#_Toc422315390)

[4.6 本章小结 53](#_Toc422315391)

[第5章 性能实验与评价 55](#_Toc422315392)

[5.1 负载均衡性能验证 55](#_Toc422315393)

[5.1.1 实验数据集及分布式环境 55](#_Toc422315394)

[5.1.2 负载均衡机制的有效性 56](#_Toc422315395)

[5.1.3负载均衡机制的开销 58](#_Toc422315396)

[5.1.4 数据分块数量对系统的影响 59](#_Toc422315397)

[5.2 Asyn-SimRank性能验证 60](#_Toc422315398)

[5.2.1 实验数据集及实验环境 60](#_Toc422315399)

[5.2.2 总体运行时间对比 61](#_Toc422315400)

[5.2.3 收敛速度对比 63](#_Toc422315401)

[5.2.4 通信量对比 64](#_Toc422315402)

[5.2.5 算法与分布式环境规模的关系 65](#_Toc422315403)

[5.3 本章小结 65](#_Toc422315404)

[第6章 总结与展望 67](#_Toc422315405)

[6.1总结 67](#_Toc422315406)

[6.2展望 67](#_Toc422315407)

[参考文献 69](#_Toc422315408)

[致 谢 73](#_Toc422315409)

[攻读硕士学位期间的论文和项目情况 75](#_Toc422315410)

# 第1章 绪论

## 1.1 研究背景

网购、网络社交、网络通信、云计算等服务极大地方便并丰富了人们的日常生活，已经成为人们生还中不可或缺的一部分。伴随着这些服务和数以亿计的用户所产生的庞大的数据也已经成为各个互联网公司最宝贵的财富。然而如此庞大的数据如何高效存储、快速访问、及时更新一直是业界和学术界着力解决的难题。为此，大数据处理技术应运而生。

随着大数据处理技术的发展，人们越来越多的采用分布式的机器学习和数据挖掘算法对这些庞大的数据进行处理和分析。由于相关的应用和服务大多要求尽可能快的分析这些庞大的数据，以达到更好的用户体验，那么就对分布式处理框架的开发、分布式的机器学习和数据挖掘算法的设计提出了更高的要求。这个问题也成为大数据领域研究的热点。

### 1.1.1 分布式计算



图1.1 分布式计算原理

Fig. 1.1 The principles of distributed computing

分布式计算[1]是云计算的一个分支，所谓分布式计算是一门计算机科学，它研究如何把一个需要非常巨大的计算能力才能解决的问题分成许多小的部分，然后把这些部分分配给许多计算机进行处理，最后把这些计算结果综合起来得到最终的结果。如图1.1。

分布式计算的思想可以解决这样的问题：将一个大的问题分解成很多子的问题，然后在将这些子问题分配给廉价的计算机，让多个计算机同时来解决这个问题，可以大大的提高计算效率并降低成本。

分布式的一个重要的应用就是实现海量数据集上的机器学习算法和数据挖掘算法。比如PageRank算法[2]，用于计算网络中一个网页的重要性。PageRank算法需要经过多次迭代才能收敛，得到一个有用的结果。然而网络中的网页个数以及他们之间的链接信息是海量的数据，在这样的数据集上完成多次迭代计算在单机上是完全不可能的。

分布式计算的另一个重要的应用是海量文本处理。对数据量以TB计算的海量数据进行分类、查询、数据清洗，采用集中式的方法来解决将花费大量的时间成本，并且很多的应用无法容忍如此长的反应时间。那么采用分而治之的思想，采用分布式计算的方法，将海量数据进行切分，分配到多个计算节点同时进行计算，将节省大量的时间成本，也使得在短时间内进行海量数据处理成为可能。

目前已经有很多的分布式计算框架可以支持很多机器学习算法和数据挖掘算法在海量数据集上的计算，比如比较通用的Hadoop框架[3,4,5,6,7]，还有支持弹性数据集SDD的Spark框架[8]，同时支持同步异步迭代计算的Graphlab框架[9]等等。

随着计算机技术和网络技术的不断发展，支持分布式计算的框架已经多达数十种，但是每种框架都有其相应的不足，因此，分布式计算发展到今天，它在技术上仍然存在着很多的挑战。如分布式框架的容错问题、负载均衡问题、性能问题等等。目前已有很多的相关工作集中到这些问题上来，越来越多的人提出不同的解决策略。

Hadoop是目前最通用并且也是应用最广泛的大数据分布式处理框架，可解决大量的大数据处理问题。但是在迭代计算的图算法支持方面，Hadoop框架由于同步开销、大量中间结果、大量IO等问题，并不高效。因此许多基于内存的异步图处理框架被提出来，如Graphlab，Maiter[10,11]等，来高效地支持迭代的图处理算法。但是在异步框架的可用性方面，由于异步计算过程没有明显的计算阶段，使得这些框架的负载均衡问题变得非常复杂。如何选择恰当的时机进行任务迁移，如何进行数据重定位，如何处理游离状态的消息等问题并不容易解决。提出有效的通用的解决方案虽然面临各种各样的挑战，但是在实际应用中也有迫切的需求。另外，由于异步框架的异步性质，导致一些应用范围较广的图处理算法在异步框架上无法实现，限制的异步框架的应用。

### 1.1.2 分布式算法

分布式算法在进行分布式计算的过程中起到非常重要的作用。设计高效的分布式算法使很多的集中式应用采用分布式计算方法来解决成为可能。

在产品推荐，好友推荐等系统中，随着电子商务、社交网站的发展，数据量变得越发的庞大，那么采用集中式计算的方式来支持推荐，在可容忍的时间内很难完成。SimRank算法[12]，用于计算对象之间的相似度，该算法可以应用到推荐系统中进行产品推荐和好友推荐。由于SimRank算法可以采用消息传递的方式在分布式环境下实现，这样就可以将推荐系统中主要的运算放到分布式环境上进行计算，进而在可容忍的时间内给出推荐信息。由于分布式计算方式具有良好的扩展性，应对更大规模的数据也将不在是难以攻克的问题。

部分的分布式算法会采用迭代的形式进行计算，可以将这一类算法称为分布式迭代算法。分布式迭代算法是分布式算法的重要组成部分，所以提高分布式迭代算法的计算效率也在逐渐的成为越来越多的人的工作目标。

所谓迭代计算就是算法以上一次计算的结果作为初始值进行下一步的计算。然而为了满足这一计算过程的要求，在分布式环境下，就不得不实现同步路障来保证分布式迭代算法的正确性。这样就会带来诸多的问题，比如由于同步路障的存在，就会产生大量的同步开销。另外，每次同步过程何时结束，有最慢的计算节点来决定，当出现负载不均衡的情况时，同步开销将大大增加。这样就不得不通过负载均衡处理来解决。

## 1.2 目的和意义

课题研究目的：本次硕士论文的主要工作旨在通过研究异步图处理框架的特性，寻求高效的，较为通用的异步图处理框架负载均衡问题的解决方案，提出抽象的负载均衡处理模型在理论上论证模型的有效性，并在异步图处理框架上实现该模型，通过实验分析解决方案的优点和不足。并在此基础之上，寻找改进更多的迭代算法，使这些算法能够在Maiter上实现，进一步扩展Maiter框架的应用范围

理论意义和应用价值：异步图处理框架的负载均衡问题相比于同步图处理框架更为复杂。本次硕士论文将在理论上提出抽象的处理模型，尽量使该模型具有一定的通用性，可以用以解决一类问题，避免为单一的问题设计单一的解决方案，尽可能的脱离具体的实现，具有一定的理论意义。

同步类型的大数据处理框架可以支持迭代计算的图算法，但是并不高效，因此较为高效的异步图处理框架有广泛的应用前景。在应用异步图处理框架时，异步框架的高效性固然重要，但是异步框架的可用性也直接决定了框架的应用。负载均衡问题处理的是否合理直接影响框架的高效性，因此提出高效的较为通用的异步框架的负载均衡问题解决方案，就有很高的现实意义。

另外，本次硕士论文的另一部分工作是拓展异步框架Maiter的应用，利用Maiter的DAIC计算模型[10]和优先级迭代方法优化迭代算法。如果某个算法能够应用DAIC计算模型和优先级迭代方法进行计算，就可以提高迭代算法的计算效率达几十倍。并且目前有很多应用广泛的迭代算法如Belief Propagation算法[13,14]、Residual belief propagation[15,16]、Sum-product算法[17]等等，都满足实现DAIC计算模型的基本要求，均有可能实现DAIC计算模型。大幅度的提高算法的计算效率，在实际应用中具有很高的现实意义。

## 1.3 国内外研究现状

大数据处理框架研究现状：Hadoop框架仍然是目前研究的最为深入，应用的最为广泛的分布式框架。各方面的研究深入到任务调度、负载均衡、容错处理、配置参数调优、HDFS[3,7]的高效存储和容错、高效地支持迭代计算、基于内存计算等等，各个研究方向均得到了深入的研究。

但是由于Hadoop框架所支持的MapReduce编程模型[18]仍然是同步计算模型，存在大量的同步开销，并且不适合大规模图处理。因此适合大规模图处理的模型Pregel[19]被提出来，Pregel采用以图顶点为中心计算方式，采用超级步来保证同步图算法的正确性。但是仍然存在同步开销。

更为高效的异步处理框架应运而生。异步处理框架的典型代表是Graphlab框架。Graphlab框架主要用于大规模的图处，可以高效的实现分布式图算法。Graphlab在计算过程中以子图为中心，创新的使用不同的封锁机制来达到不同粒度的异步计算。在保证算法正确性的同时实现了异步计算。

另外同样是大规模异步模图处理框架Maiter创新的采用DAIC (Delta Based Incremental Computation)计算模型，实现了完全的异步计算，即：各个计算节点完全自治，无需任何同步过程。同时采用优先级迭代加快了算法的收敛速度，有效的提高了大规模图处理的效率。

分布式系统负载均衡处理问题研究现状：目前负载均衡处理的方法根据文献[6]主要有两种：一种是静态的负载均衡处理，一种是动态的负载均衡处理。

在静态的负载均衡处理方式下，重要的工作都集中在分布式处理过程的任务部署阶段。如Spark框架和Graphlab框架均采用这种负载均衡方案。主要采用的方法是在数据划分的过程中，尽可能平均地将数据分配给每一个计算节点，常用的划分方法有哈希划分、范围划分、min-cut[20]等等。在计算的过程中不进行任何的负载均衡处理。静态的负载均衡处理方法实现简单，但是仅仅均匀的划分数据并不能保证每个计算节点的计算量均匀，主要的原因在于计算节点的计算量不仅与输入数据量有关，还与运行的算法或执行计算的行为有关，因此仅仅使用静态的负载均衡处理方法并不能满足完全负载均衡的要求。

动态的负载均衡处理方法可以和静态的负载均衡处理方式配合使用。动态负载均衡方法的主要工作集中在计算的过程中，普遍采用的方法是首先建立计算量监测机制，该监测机制通过统计各个计算节点上的各种相关的信息，并根据这些统计信息做出决策，即：将某一个负载过重计算节点上特定数量的数据迁移到某一台负载较轻的计算节点上。

根据做出决策的节点的分布，动态负载均衡处理又分为集中式动态负载均衡处理和分散式负载均衡处理。所谓集中式的负载均衡处理是指由单一的计算节点来做出决策（如Master节点）。集中式负载均衡处理的优点是通信量较小，但是容易造成决策节点任务过重。分散式的负载均衡是指每个计算节点均维护一份统计数据，如文献[9]提出的Pregel负载均衡解决方案，主要根据统计数据自发的进行负载均衡处理。不存在集中的决策节点，可以有效的避免单一节点任务过重，但是由于各个计算节点的统计信息需要同步，通信量较大。

大规模图处理算法的研究现状：目前对于迭代图处理算法的研究主要分为两个方面：

(1)改进分布式框架以高效的支持迭代。Hadoop在支持迭代计算的过程中，将每次迭代作为一个Job来执行，这样进行迭代计算，使得计算过程中需要大量的任务调度开销，产生大量的中间结果，进行大量的IO操作，导致迭代的效率不是很高，因此很多的改进框架由此产生。Spark通过反复使用的数据长驻内存，来挺高迭代计算的运行效率。Hadloop[21]则通过将多次迭代过程使用一个Job来完成的方式，减少了任务初始化、任务调度的开销。还有像IMapReduce[22]，Priter框架[23]通过减少任务调度和采用优先级迭代方法来提高迭代算法的计算效率。Graphlab则通过以子图为中心的计算方式来提高运行的并行度，通过封锁机制实现异步计算来减少同步开销的方法提高迭代算法运行效率。

(2)改进算法提高计算效率：在迭代算法的改进方面，主要的工作都集中在减少通信，减少计算量方面，如文献[24,25]，采用的方法多种多样，Maiter采用的DAIC计算模型和优先级迭代方法，就是通过有效的避免收敛点的计算来减少通信量和计算量。

## 1.4 主要研究内容

异步图处理框架Maiter实现了完全的异步图处理，并通过累积累积迭代计算，优先级迭代计算极大的增加了分布式迭代算法的收敛速度，在计算性能上有了极大的提升。本次硕士论文的研究内容将围绕着Maiter框架展开，主要包括两个部分：Maiter框架的负载均衡处理问题的研究和DAIC计算模型在分布式迭代算法中的应用。

### 1.4.1 Maiter框架的负载均衡处理问题的研究

负载不均衡的问题在分布式框架中普遍存在。产生的原因主要是数据分配不均匀，各个计算节点的计算能力不同以及由于应用的性质导致各计算节点在分到相同的数据量时，仍然产生不同的负载。

在同步框架中，由于同步路障的存在，同步过程何时结束，由最慢的一台计算节点决定，所以当出现负载不均衡问题时，会大大的增加同步开销，进而增加整体的时间开销。例如Hadoop框架，在Map任务和Reduce任务之间存在一个同步过程，当所有的Map任务结束，Reduce任务才能开始。在Map任务开始之初，数据是以Hash划分或者范围划分的形式分配到每台计算节点上的，每台计算节点可能负责多个Map任务和Reduce任务。假设数据划分均匀，但是有一台计算节点任务中途出错重启或计算能力较差，那么该计算节点上Map任务完成之前，所有计算节点上的Reduce任务全部都不能开始，显然一台计算缓慢的计算节点会严重的拖慢整个集群的计算速度。

在异步框架中，虽然不存在同步过程，但是负载均衡问题仍然存在。例如在异步图处理框架中，所有的图顶点在没有得到消息的时候，不会进行计算，但是会反复的试探是否有与之相关的消息到达，这也是对计算资源的消耗。假设在集群中，有一台计算节点比其他的计算节点慢很多。那么在这台计算节点上的图顶点更新的就会比较慢，进而导致这些点产生的消息也不能及时的传播出去，由于图顶点会采用Hash划分或者范围划分的方式分布都不同的计算节点上，使得那些其他节点上使用这些信息的顶点一直处于试探状态，计算资源被消耗，但是却没有为整体的计算做出任何的贡献。如果可以将该节点上的部分任务迁移到计算较快的节点上，那么就可以将消息尽快的传播出去，进而加快整体的运算速度。

Maiter框架是异步类型的图处理框架，也就是在计算的过程中，不存在同步过程。从任务提交到结束，计算一直在进行。Maiter框架在计算的过程中也存在负载不均衡问题。如同上文所述，当Maiter框架出现负载不均衡时，也会阻碍消息的传播，进而拖延整体计算。此时，就需要将较慢的计算节点上的数据迁移一部分到其他运算较快的计算节点上去，但此时计算正在进行，与迁移数据相关的消息有的可能仍在传输的路上，有的可能刚刚产生，并且这些消息的数量难以确定。由于数据迁移到了新的节点，那么这些与之相关的消息如何正确的定位是非常困难的。另外，在计算的过程中进行任务迁移，数据迁移会导致迁移数据的计算被拖延，是否会导致最终的结果错误，这需要给出理论证明。

本文提出的负载均衡解决方案是基于数据块的集中式动态负载均衡处理机制。数据迁移以数据块为单位，这种迁移方式使得数据定位更加方便。在计算进行的过程中，监测各个计算节点的一些相关信息，并汇总到Master节点，Master节点则通过统计这些信息来进行实时的负载均衡处理。详细的解决方案将在后续章节详细介绍。

### 1.4.2 异步累积迭代算法的研究

Maiter框架所实现的计算形式就是DAIC计算模型，也可以说DAIC计算模型是Maiter框架的精髓所在。DAIC计算模型实现了部分分布式迭代算法的异步计算，使得这些算法的计算不在需要同步过程，大大的提高了计算效率。DAIC计算模型在实现了异步计算的同时，创新的采用了累积迭代的计算方式。累积迭代的好处在于，每次只计算增量，但是那些已经收敛的图顶点的增量为零，所以可以让这些点不在参加计算，节省了计开销。另外，DAIC计算模型是实现了分布式迭代算法的优先级迭代，可以动态的决定每个顶点的计算优先级，让那些对全局收敛更重要的顶点优先的计算，极大的提高了分布式算法的收敛速度，进而提高整体的计算性能。DAIC计算模型将在后续章节详述。

因为DAIC计算模型拥有各种各样的优点，所以如果能够在某个分布式算法上应用DAIC计算模型，那么对算法本身就是一种极大的改进。但是在分布式算法上应用DAIC计算模型却并不是一件容易的事。首先分布式算法本身要满足DAIC计算模型所要求的一些条件。其次在应用DAIC计算模型时，要对分布式算法的计算形式进行相应的变换，而这种变换根据分布式算法性质的不同而不同，并且有的算法可能无法进行变换。最后在某个分布式算法上应用了DAIC计算模型之后，其原有的计算形式将会被改变，那么应用了DAIC计算模型的分布式算法是否收敛，最终的结果是否正确，需给出严格的理论证明。

本论文采用DAIC计算模型的思想，改进了SimRank算法，提出Asyn-SimRank算法。实现了相似度计算的异步计算和关键点优先计算，并给出了Asyn-SimRank算法的收敛性和正确性证明，和关键点优先策略的有效性证明。实验表明，Asyn-SimRank算法比SimRank算法拥有更快的收敛速度和计算效率。

## 1.5 论文章节安排

论文分为六章，各章的内容安排如下：

第1章，绪论。本章详细的介绍本文的研究背景，主要工作的目的和意义以及相关工作的国内外研究现状。

第2章，Maiter框架介绍。由于本文的主要工作围绕着Maiter框架进行展开，涉及到Maiter框架的实现细节，Maiter框架的计算原理，因此详细的介绍Maiter框架可以便于读者的理解。

第3章，Maiter框架负载均衡技术。本章主要介绍Maiter的负载均衡处理过程中所遇到的数据迁移问题、迁移数据重定位问题、错位消息处理问题，以及数据迁移的正确性的相关证明。

第4章，异步累积迭代算法。本章将应用DAIC计算模型改进了SimRank算法，并提出了Asyn-SimRank算法。本章将详细的介绍Asyn-SimRank算法的推导过程，收敛性和正确性证明，以及优化技术的有效性证明。

第5章，性能实验与评价。本章将首先验证本文采用的基于数据块的集中式负载均衡处理方案的有效性，分析负载均衡处理过程的开销，对实验结果进行分析。然后验证Asyn-SimRank算法的计算速度、收敛速度、扩展性等等，并对实验结果进行分析

第6章，总结和展望。对本文工作进行了总结，并给出本文采用方法的不足之处和相应的改进思路。

# 第2章 DAIC模型及Maiter框架

Maiter框架是基于消息传递的大规模异步图处理框架，它实现了DAIC计算模型，支持异步累计迭代计算，支持优先级迭代计算。Maiter框架实现的DAIC计算模型解决了分布式迭代算法计算过程中的中间结果问题，网路高负荷以及运算效率低等问题问题。Maiter框架采用消息通信机制有效的减小了机群中各节点的的通信量，减小网络负荷。实现异步式累积迭代计算，避免了同步开销，取得了很好的计算效率。同时实现了优先级迭代计算，可以动态的决定图顶点的计算优先级，极大的提升了分布式算法的收敛速度。

Maiter框架基于内存实现，它将迭代计算中反复使用的数据放在内存中，并将这些信息存放在每个节点维护的状态表中，避免了计算节点在处理过程中的大量IO操作，使得计算速度不再受到IO速度的限制的同时，提高了数据定位、寻址的速率。下面我们来介绍一下Maiter框架的原理。

## 2.1 传统同步迭代计算

所谓迭代计算就是在一个数据集上执行若干次相同的计算，将上一次计算的结果作为下一次迭代的初始值，迭代计算的关键是迭代更新函数，记作F：

*=*()。 (2.1)

其中代表一个由图顶点上的值组成向量，，其中*n*为图中顶点的个数。代表第*k*-1次迭代计算的结果。由于也是有所有图顶点上的值构成的向量。因此我们可以进一步将同步迭代计算更新函数记为()来代替()，那么对于图中第*j*个顶点上执行的迭代更新函数为：

。 (2.2)

在分布式计算中，多个处理器并行的执行迭代更新函数。为了陈述简便，假设：有*n*个处理器，在迭代计算的第*k*步，那么在处理器*i*上的第*j*个图顶点运行迭代更新函数时，处理器*i*首先接收所有的从其他处理器发来的，其中i=1，2，3，…，*n*。也就是说，在同步的分布式环境下，进行同步的迭代更新函数，有一个主要的缺点：在第*k*-1迭代没有完全完成之前，第*k*次迭代就不能开始，即第*k*次迭代依赖于全部*k*-1次迭代的结果。在同步迭代过程中，为了满足这一要求，完成同步迭代更新函数，就必须在每一次迭代结束时实现同步路障，然而这些同步过程大大的降低了分布式迭代算法的计算效率。

## 2.2 传统异步迭代计算

为了解决同步迭代计算过程中需多次同步过程而引起的效率低的问题，异步迭代计算被提出来。实现异步迭代计算的思路尽可能的避免同步路障。

一种思路是在迭代计算过程中，图顶点*j*会主动的去执行更新函数，它会主动的从其他的处理器上去“拉”所有必须的数据元素，然后用这些数据对进行更新操作，而非等待所有的数据到达，这种方式无需实现同步路障。然而，主动的进行更新，采用“拉”数据的方式，与实现同步路障相比，虽然避免了同步开销，然而却需要更大的通信量，以为图顶点j要不断的试探自己计算所必须的值是否已经产生，进行了很多无用的通信。一个处理器需要从所有其他的处理器上“拉”数据，但是并非所有的都需要更新，甚至在最坏的情况所有的都不需要更新，在这种情况下，异步迭代计算需要巨大的通信负载并进行了很多无意义的操作，因此，基于“拉”数据的异步迭代计算只适用于通信负载可忽略不计的情况下。

采用同步路障可以保证同步分布式算法的正确性，但是保证同步分布式算法的正确性同步路障并不是必须的。在图处理过程中，同步路障可以保证所有的图顶点所必须的值全部到达，才开始下一步迭代计算。然而，实际上当一个图顶点所需的值已经到达，该点就可以开始计算，而不用等其他的顶点。另一种思路是采用子图划分的思想，在计算的过程中，以子图为中心进行计算，这样相较于以顶点为中心，极大地提高了各个计算单位的独立性，为异步计算来创造了更多的机会。当某个子图所必须的值已经到达时。该子图就可以并调度计算。

使用这种思路实现异步计算的典型框架是Graphlab。Graphlab框架在计算开始之初，采用划分图顶点的方式，将要处理的原始图划分成若干个比较小的子图，然后均匀的分配到各个计算节点上去，每个计算节点负责多个子图。在计算的过程中，当一个计算节点中负责的某一个子图已经收到了全都所必须的值，就会将自己的状态标记为可计算，然后调度器就会适时的调度该子图进行计算，计算完成后，子图状态改为等待信息。从上述过程中可以看出，Graphlab框架在计算的过程中，不存在同步路障，所以也不存在同步开销，实现了异步图处理。

## 2.3 DAIC计算模型

### 2.3.1 DAIC计算模型的前提条件

Maiter框架所实现的DAIC计算模型的提出，有效地避免了同步路障带来的开销，同时也解决了异步框架中高通信量、高内存消耗问题，也避免了大量无意义的操作。DAIC计算模型最重要的特点是支持异步累积迭代计算和优先级迭代计算，极大的提高了分布式迭代算法的计算效率。DAIC计算模型可以简单的翻译为累积迭代计算。

首先，同步迭代计算要求上一步的计算结果作为下一步迭代计算的初始值。即第*k*次迭代中的值，依赖于第*k*-1次迭代的计算结果，同步迭代更新函数可以表示成如公式(2.2)所示。

在给出累积迭代计算的定义之前，我们先讨论一下累积迭代计算执行时需满足的条件。如果分布式迭代算法可以采用DAIC计算模型，那么该算法的同步计算形似应该能够表示成以下形式：

(2.3)

其中：*k*=1，2，3，……n 。⊕是一个抽象操作。*(x)*代表图顶点*i*对图顶点*j*的影响。换句话说，就是图顶点*i*将值()(而不是)“推”给图顶点*j*，然后图顶点*j*就可以将其他处理器传来的()值和使用⊕操作将其累积到上，这样一步累计更新就完成了。实际上这一前提条件就是要求，能够使用DAIC计算模型的分布式算法的计算公式能够表示成一系列值累计的结果，这一前提，是后续推导DAIC计算模型的充分条件。

以大家熟知的PageRank算法为例，PageRank算法主要用于网页排名，主要应用于搜索引擎中，其核心思想是与重要的网页链接的网页也很重要。PageRank算法迭代的更新有PR值组成的长度为*n*的数组R，迭代过程如下：

*=d\* +(1-d)* (2.4)

*d*是阻尼系数，|*N(i)*|是网页*i*的链出网页数，(*ij*)是网页*i*指向网页*j*的一个连接。如果我们把上式改写成累积迭代计算的形式的话，PageRank的，如果网页*i*到网页*j*有一个连接，()=d，否则()=0，操作“⊕”时 “+”。

另外，DAIC计算模型要求运算“⊕”和函数*(x)*满足一些常见的运算定律，这些定律都将成为推导DAIC计算模型的前提条件。

1. *(x*)在操作“⊕”上满足分配率,即：
2. 操作“⊕”满足交换律，即：

*x*⊕*y=y*⊕*x*

1. 操作“⊕”满足结合律，即：

*x*⊕(*y*⊕*z*)=(*x*⊕*y*)⊕*z*

1. 操作“⊕”满足等同律，即：存在抽象值*o*满足：

*x*⊕*o*=*x*

这些条件将在下一节中的DAIC计算模型的推导过程中用到，并且在推导过程中可以很容易的看出，这些条件均为充分条件。

### 2.3.2 DAIC计算模型的推导

上文已经提到，如果某个分布式迭代算法的同步迭代更新函数：

可以表示成以下形式：

那么用*Δ*来表示从到相对于运算⊕的变化，即：

*=* (2.5)

那么将(2.5)式带入到(2.3)是中，可得：

(2.6)

由于*(x*)在操作“⊕”上满足分配率，即：

那么(2.6)式就可以写成：

因为运算⊕满足交换律和结合律，因此整理后可得：

由于

且：*=*⊕*Δ，*可以推出：

*Δ*

即：

由以上的一系列推导，推出了的计算公式，那么应用上文的一系列推导，满足条件的分布式算法可以分以两个步骤来计算：

其中*k*=1，2，3，…，*n*，可以根据算法被初始化为任何值，但是为了简便，根据我们前面提到的Maiter需满足的形式，我可以将赋值为0*，*赋值为，这样就有：+,=。这同时也是迭代开始的充要条件。

公式(2.8)只是推导出了DAIC计算模型的同步计算形式，因为公式(2.8)中仍然存在*k，*也就是说必须是第*k*次和第*k*-1次迭代的差值，然而由于DAIC计算模型所具有的性值，实际上，在真实计算的时候，并不需要累积的粒度必须是两次迭代的差值，而是可以是任意的粒度。

通过以上的推导，我们发现一些传统的同步类型的分布式迭代运算如果可以改进成DAIC计算模型，那么下一次迭代什么时候开始，并不取决于上一次迭代什么时候结束，只要我们得到一些最近的值就可以进行接下来的运算，无需同步，这就为需迭代运算的算法的并行异步执行创造了更多的条件。同时使运算的速度大大的提升。

### 2.3.3 DAIC计算模型的优先级迭代

所谓优先级迭代，就是在分布式迭代算法计算的过程中，给每一个图顶点符一个优先级，并在后续的计算过程中，优先地调度优先级高的图顶点进行计算，以达到快速收敛的目的。

对于同步类型的分布式迭代算法，经过若干次迭代计算后，都会收敛到一个固定的值。但是在同步类型的分布式分布式算法中无法采用优先级迭代，异步同步的分布式算法要求所有的图顶点同时参加运算。但是在DAIC计算模型中则不同，由于DAIC计算模型实现的异步迭代计算，这就为优先的计算一些顶点创造了条件。

但是到底优先的计算哪些图顶点才会加快分布式算法的收敛呢？实际上DAIC计算模型在计算的过程中有一个非常重要的特性，就是多有的值在计算的过程中是单调递增的，也就是说，在计算的过程中，所有的值都会从初始值不断的递增，直到收敛值。既然DAIC计算模型采用的累积差值的方式进行计算的，那么在异步计算过程中，优先的去计算那些当前差值大的顶点，就可以加快分布式算法的速度。

从另一个角度来思考，优先的计算差值较大的图顶点也是合理的。在几乎所有的图数据中，都存在80/20原则，即少部分的图顶点具有很大的入度和出度，而大部分的图顶点的入度出度都很小，而这少部分具有较大入度和出度的点却对分布式算法的全局收敛影响更大。而这部分顶点在进行DAIC计算时，差值往往都比其他的顶点大。因此，优先的计算对于增快算法的收敛是有益的。

在理论上，DAIC计算模型的提出者也给出了优先级迭代的有效性证明。详见文献[11]，此处不再赘述。

## 2.4 Maiter框架

以上我们介绍了Maiter框架的理论依据，下面我们介绍一下Maiter框架式如何实现的。Maiter框架是picoolo框架的基础上改变而来，但是改变很大。

### 2.4.1 Maiter框架的实现

它和Hadoop模型一样，包括一个Master节点和若干个Worker节点，Master节点用于控制和监控Worker节点的工作状态，并进行任务分配。若干个Worker节点是并行工作的。但是和Hadoop文件传输不同的是，Maiter框架使用MPI消息通信进行计算节点间的通信，因此Maiter采用的是消息通信形式，减小了通信量，增加了通信的效率。

Maiter框架的主要任务就是如何使每个Worker节点并行的完成异步累计迭代运算，其实也就是我们上一节讨论的两步运算。所以我们下面重点讨论一下Worker节点的设计。

分布式运算的第一步，就是将待处理的原始图数据平均的分配到每个Worker节点上，Maiter也同样有这样的过程。每个Worker节点会将原始数据的一个子集载入到内存中，等待处理，Worker任务是处理自己所负责的图顶点，在这些顶点上执行异步累积迭代计算，每个图顶点由一个全局唯一的*key*来标识。*key*为*j*的数据元素会被分配给第*h*(*j*)个Worker节点，*h*(*j*)是一个哈希函数，这样采用Hash函数划分数据，会使得数据定位更加容易。为了分配均匀，我们可以让*h*(*j*)为取余运算。除此之外，对数据分片进行预处理也可以使得数据分配的更均匀，避免出现负载不均衡问题，比如我们可以使用轻量聚集算法对图进行预处理，把一些关联的比较紧密的图节点分配到一个Worker上，可以大大的减少通信。

每个Worker节点分配到的图数据会放在内存中的状态表中，状态表中的每一行保存一个图顶点的相关数据。为了获得更快的计算效率，状态表采用哈希表实现，并全部存放在内存中。因为在计算的过程中，会不断的在状态表中对特定的*key*进行定位，这种情况下，哈希表的复杂度是O(1)。状态表的结构如图2.1所示。

每个状态表都包含五个域，*k*，*v*，*v*，*pri*，*data*。其中，*k*域用来保持上文所提到的全局唯一的*key*值，用来唯一标识这个状态表。第二个域用来存储，第三个域用来存放，*pri*域用来存放状态表的优先级，系统会根据该值的不同，按不同的顺序进行更新状态表中的一些域。*data*域则是用来存放分配给它的信息。

接下来的问题就是，如何在每张状态表上完成异步累积迭代计算呢？Maiter框架为每张状态表设计了两个线程，分别是Receive线程和Update线程。它们也相应的完成Receive操作和Update操作。每个Worker节点通过Receive进程不断的接收从其他计算节点上传给它的消息，然后把这些信息累积起来，Update进程则使用该累积值来更新状态表的*v*域，并产生新的消息传送出去。其整个过程如图2.1所示。



图2.1 Maiter执行过程

Fig. 2.1 Executing Processes of Maiter

首先Receive操作将其他Worker节点发来的消息用运算⊕将其累积到*v*域上，当累积的一定程度的时候， Receive进程停止对*v*域进行修改，并将以后发来的值先缓存起来。这时Update进程开始工作，Update操作将*v*中的累积值仍用⊕运算累积到*v*域上，然后在用*v*中的值完成函数*(v)*，然后将该值发给*i*节点之后，将*v*置**o，**其详细过程如下中所示：

至此，Maiter的理论、设计、实现我们就讨论完了。从上文的Receive操作和Update操作可以明显的发现，Maiter的计算只依赖于当前所接收到的消息，就可以开始运算。并不像同步迭代计算那样，必须等到所有的值到达，下一步的运算才能开始进行。这样就实现了完全的异步迭代计算。

### 2.4.2 Maiter框架的优先级迭代的实现

Maiter框架实现了DAIC计算模型的异步累积形式，下面介绍Maiter框架如何实现的优先级迭代。

在上一节我们介绍的Maiter框架的实现中，已经介绍，Maiter框架为每个图顶点在内存中的状态表中创建一项来保存该顶点相关的信息，其中有一个域*pri，*就是用来保存每个图顶点的优先级。该域中的值一般被设为当前该图顶点收到的差值的总和。但是由于Maiter框架处理的是海量的数据，处理的图也非常的庞大，在如此海量的图顶点中，找到优先级最高的点，并进行调度计算，这样的复杂度很高。

Maiter框架解决这一问题的方案是每次调度优先级高的一部分进行计算，而不是每次调度优先级最高的进行计算，这样就避免了多次的寻找优先级较高的顶点，也节省了调度的开销。

但是精确的找到优先级最高的一部分，计算复杂度仍然很高，可能大大的增加计算开销。如果采用采样的方法，找到近似的优先级较高的一部分进行调度，就可以大大的减少选取图顶点的开销。其采样的过程是：首先在计算节点的状态表中采取一定量的采样点，将这些采样点进行排序，然后根据用户配置的参数，选取多少优先级高的进行调，例如10%，那么就将采样点中的排在10%位置的图顶点的*pri*值作为临界值，Maiter框架在调度计算时，将所有*pri*域大于该临界值的图顶点调度进行计算。这样Maiter框架就实现了对优先级迭代的支持。

### 2.4.3 Maiter框架的API

如果说Hadoop框架支持的是MapReduce编程模型，那么Maiter框架支持的就是ReceiveUpate编程模型，但是不同的是Maiter框架并没有Receive和Update这两个API，而是由一些其他的API来完成Receive和Update操作，并且这些API在Maiter所实现的DAIC计算模型中都有其固定的意义。并且部分API只是需要设定某些值就可以完成，不需要复杂的操作。因此Maiter编程模型编程更加便利。下面介绍一下Maiter框架提供的API。

Maiter框架的API分为两部分，一部分API用于完成Receive和Update操作，另一部分用于控制计算何时结束。第一部分如下所示：

* + *void read\_data(string& line, K\* k, D\* data) ;*
  + *void init\_c(const K& k, V\* delta)*
  + *const V& default\_v() const*
  + *void accumulate(V\* a, const V& b)*
  + *void priority(V\* pri, const V& value, const V& delta) ;*
  + *void g\_func(const V& delta, const D& data, vector<pair<K, V> >\* output)*

其中*read\_data*()用于每个Worker节点读取图数据分片的信息，并用这些信息来初始化自己状态表中的*data*域和*key*域；inti\_c()用于初始化状态表中的*v*域；*default\_v*()用于初始化状态表的*v*域，同时用来设置Maiter运算模型中的满足等同率*o*。*accumulate*()用于设置Maiter运算模型中的⊕运算；*priority*()用来设置Worker状态表中每一项的计算优先级，在Maiter框架运行时，会选择优先级比较高的状态表中的项优先进行计算，这有助于快速收敛；*g\_func*()则是用*v*域中的信息来运算的到，并将算出的其他节点的信息发送给相应的节点。

控制程序终止的API如下所示：

* *double estimate\_prog(LoaclTableTterator<K,V> \*table\_itr)*
* *bool terminate(list<double> local\_report)*

由于Maiter特殊的异步迭代形式，实际上Maiter框架是没有迭代步的概念的，在一次运算中用到的数据可能是*k*次迭代的结果，也可能是*k*+1此迭代的结果，因此它不能像Hadoop框架一样用迭代的次数来控制程序的终止。

Maiter框架采用的方式是：在固定的时间内所有节点的状态表中*v*域的总和的变化情况来决定程序什么时候结束。其中*estimate\_prog*()是用来设置时间间隔和变化的大小的，*terminate*()用来决定是否停下来。这两个API并不是必须的，在未指定的情况下，Maiter框架会调用默认的API。

## 2.5 本章小结

本章详细的介绍了DAIC计算模型的推导和大规模异步图处理框架Maiter的实现。本章将作为后续章节的铺垫，在后续章节的研究内容中，Maiter框架的负载均衡处理问题离不开Maiter的实现。DAIC模型的应用研究离不开DAIC计算模型的推导过程。因此，在后续章节中会反复的提到本章中所介绍的内容。

# 第3章 Maiter框架的负载均衡技术

Maiter框架作为完全异步的分布式框架，与同步的分布式框架一样，也存在负载不均衡问题，两者负载不均衡的原因类似但是也有所不同。在本章，将详细介绍Maiter框架负载均衡问题产生的原因，主要的解决思路，系统的实现细节，解决问题的过程中面临的新问题等内容。

## 3.1 负载不均衡的原因

### 3.1.1 同步框架负载不均衡的原因

负载不均衡问题在分布式计算过程中普遍存在，例如在同步类型框架Hadoop框架，基于BSP模型的Pregel框架，都存在负载均衡问题。但是产生的原因确实多种多样的，如数据划分不均匀，集群中计算节点的计算能力不同，有时还跟框架上运行的应用的特性有关。

在Hadoop框架中，数据一般按着固定大小的数据块均匀的分配到每个计算节点上去。但是数据块的划分一般采用Hash换份或者范围划分的方式，这种静态划分的方式并不能保证每个数据块的大小绝对相同。有时这种划分导致的不均匀也会导致Hadoop框架的各个计算节点的Map任务完成时间非常不同，增加同步开销。

另外即使数据划分大致均匀，每台计算节点的计算能力也有可能是不同的，系统并不能在划分数据之前有效的预见计算节点的计算能力。在有，应用本身的特性也会造成负载不均衡问题。例如在Map任务结束之后，相同key值对应的value的个数可能差距很大，但是每个key值所对应的Reduce任务是固定的。当某个计算节点上的Reduce任务多处理的key值对应的value值的个数相比其他Reduce任务都大的时候，就会产生负载不均衡问题。

Pregel框架数同步类型的图处理框架，采用消息传递的方式运行图处理算法。Pregel框架产生负载不均衡问题的主要原因同样是数据划分不均匀以及计算节点的计算能力不同。由于在运行分布式迭代算法的时候，两次迭代计算之间存在同步过程，因此负载倾斜也会极大的增加听不开销。

对于同步类型的框架负载均衡的处理，主要的思路是采用数据迁移的方式解决。将负载较重的计算节点上的一部分未处理的数据迁移到负载较轻的计算节点上，最终使得各个计算节点的运行时间大致一致，尽量同时完成，这样就可以极大地减少同步开销，从而提高整体计算效率。

对于同步类型的负载均衡处理的最佳时机是在同步路障完成以后，因为此时，所有计算节点的状态的已知的，并且保持不变，在集群中没有消息正在传递，这样就为迁移数据提供了很多的方便，使得数据迁移之后，不用再考虑为正确定位的消息的处理问题，是负载均衡处理变得容易。

### 3.1.2 Maiter框架负载不均衡的原因

Maiter框架是完全异步的图处理框架，那么在计算的过程中，就不存在同步过程，进而也不会出现像同步框架那样，所有计算较快的计算节点等待某个较慢的计算节点完成，才能进行后续的计算。进而也就不会因为数据分配不均匀，节点计算能力不同等原因造成最慢的计算节点拖延整体的计算。那么为什么像Maiter框架这样完全异步的图处理框架也会存在负载不均衡问题呢？

实际上，图处理框架与其他的文本处理框架有所不同。在计算的过程中，文本处理分布式框架可能更多在意的是能否更快的处理更多的量，而在图处理框架上，往往运行的都是图处理算法，所以更多的在意的是算法是否能够快速的收敛，结束计算。那么都有哪些因素能够影响到算法的收敛呢？

首先就是消息传递的快慢，可以假设一种场景：如图3.1所示。假设有两台计算节点Worker1和Worker2，Worker1计算能力较低，而另一台计算节点Worker2的计算能力较高，假设它们所处理的图顶点的个数相同。从图3.1中可以看出，Worker2上的图



图3.1 Maiter负载倾斜原因示例

Fig. 3.1 The case of load skew

顶点依赖于Worker1上的值。但是由于计算节点Worker1的计算能力较差，假设将其负责的所有顶点都计算一次所用的时间为10s，而Worker2将所有负责的节点计算一遍要1s。但是在这种情况下，由于图顶点7要至少10秒才能获得图顶点6传来的值。尽管Worker2节点的计算能力较高，但是大多时候可能都是在试探是否有值传来，而不是在进行有用的计算。

又因为Maiter框架进行的是异步计算，在计算的过程中，有些点对于全局的收敛是至关重要的，如果这些点在图顶点6的上游，那么如3.1图所示的情况出现的话，就会拖延整体计算的收敛速度。

虽然上文所举的例子只是一种极端的情况。由于是异步计算，Worker2接收不到图顶点6传来的值，期间可能会受到其他的计算节点传来的值并进行计算，不会完全的空闲。但是在图处理的过程中，消息会在整张图中不断的传播。由于Worker1节点计算较慢，使得经过Worker1负责的节点上的消息全部被拖延，进而导致整体运算被拖延。

上文所说的是所举的例子是负载倾斜问题发生的时候，是如何影响异步处理框架Maiter的整体计算的。但是多种多样的原因可能会导致这种状况出现，如某个计算节点的计算能力较差，数据分割不均匀，应用本身特性等等，和同步类型的框架负载不均衡的原因是一样的，但是作用机理不同，因此负载均衡处理也就不同，在后续内容中会详细的介绍Maiter的负载均衡处理过程。

## 3.2 负载均衡方案的选择

在1.3节中已经介绍，主要的负载均衡处理主要分为两类：静态负载均衡处理和动态负载均衡处理。

所谓静态负载均衡处理实际上是在计算之前，通过静态的数据划分如Hash划分、范围划分或者任务分配等方式让数据或者任务均匀的分配到各个计算节点上去。而在计算的过程中不在进行负载均衡处理。静态的负载均衡处理简单并且有效，但是并不足够。前文已经讲过，导致负载不均衡原因除了数据或任务分配不均匀之外，计算节点的计算能力，应用本身的特性，也一样会导致负载不均衡。显然静态的负载均衡处理只是解决了数据划分不均匀儿女导致的负载不均衡，其他因素导致的负载不均衡并没有解决。

虽然如此，但是静态的负载均衡处理方式在大部分的分布式框架中都被应用，因为这种负载均衡处理方式简单可行，与其他的负载均衡处理方法配合使用，可以得到更好的负载均衡处理效果。Maiter框架在运行之前，要求将原始图数据采用Hash划分的方式均匀的分配到各个计算节点上，因此静态负载均衡处理在Maiter框架中已经应用，不作为本文的主要可选方案。

动态负载均衡处理方案是指在分布式框架计算的过程中，通过监测一些能够表征各个计算节点运行负载的量，如预期的的处理所有数据的时间、输入信息量、输出信息量等等，通过实时的信息统计，有决策节点做出负载均衡处理决策。根据负载均衡处理决策节点的不同，动态负载均衡处理可以分为集中式负载均衡处理和分散式负载均衡处理两种方式。集中式负载均衡处理指的是有单一的计算节点进行负载均衡决策。而分散式的动态负载均衡处理则是所有的计算节点自发的进行负载均衡处理。对于集中式的动态负载均衡处理，由于单一节点做出负载均衡决策，因此统计信息只是由计算节点传到决策节点，这样节省通信并且不增加计算节点的开销，但是由于决策节点要维护监测信息并运行决策算法，大大的决策节点的负载。分散式动态负载均衡处理虽然避免了单一决策节点负载过大，但是由于各个计算节点自发独立的进行负载均衡处理，状态信息却需要进行同步，比如哪个节点做出了什么决策，需要每个计算节点都知道，同步这些信息会增加通信量。另外每个计算节点都需要知道其他计算节点的运行状态，这也需要很大的通信开销。



图3.2集中式动态负载均衡处理功能模块图

Fig. 3.2 The centralized dynamic load balancing function block diagram

动态的负载均衡处理能够实时的解决随机出现的负载不均衡问题，可以解决几乎所有因素导致的负载不均衡问题。实时的解决负载倾斜问题，也正是Maiter框架负载均衡处理的需求。另外，由于Maiter框架的Master节点只是在任务初始化时进行任务部署，在计算的过程中只是监测少数的值用于决定何时任务结束，因此Maiter框架的负载是比较轻的。综上，Maiter框架适合集中式的动态负载均衡处理方式。集中式动态负载均衡机制的功能模块图如图3.2所示，在接下来的小节，将详细的探讨各个功能的实现细节。

## 3.3 负载均衡处理决策算法

负载均衡决策算法是负载均衡处理的灵魂，负载均衡决策算法将直接决定负载均衡处理是否有效，是否高效。所以负载均衡决策算法的设计是Maiter框架负载均衡处理的重中之重。

### 3.3.1 决策算法设计思路

决策算法的主要功能是根据计算节点上报来的监测信息，通过对这些信息的统计，分析每个计算节点的运行状态，当出现负载不均衡的时候做出恰当的负载均衡处理决策。虽然，每个计算节点快速的执行负载均衡决策，是负载均衡处理机制的高效性不可缺失的。但是能否做出正确的负载均衡处理决策，将直接决定负载均衡处理是否有效，因此负载均衡处理决策算法的设计必须从导致不均衡的原因根源出发，才能保证负载均衡决策算法的有效性。

在前文已经介绍过Maiter框架负载不均衡的深层原因。当某台计算节点出现负载不均衡的时候，会将消息阻塞在其所负责的图顶点的上游，而无法快速的传播下去。因此对于图处理框架的负载均衡处理，绝对不能简单的处理，仅仅是为了每个计算节点能够同时完成计算任务。而是应该将主要的关注点放在消息的疏导上，使在图中传播的消息能够快速的传播出去。

由于Maiter框架有优先级迭代的概念，那么最简单的负载均衡思路就是，能否将优先级高的图顶点迁移到计算能力较高的计算节点上，这样可以获得更快的收敛速度。为了实现这种思路，可以将计算机的计算能力排序后，将每个数据块的优先级总和进行排序，然后将优先级最高的数据块迁移到计算能力最强的计算节点上，将优先级排序第二的数据块迁移到计算能力第二的计算节点，依次类推。

这种负载均衡处理方案，可以有效的应用Maiter框架的优先级迭代，进一步加快分布式算法的收敛速度。负载均衡处理的出发点就是以少量的开销，换取更大的收益。但是采用这种负载均衡处理算法很容易得不偿失。因为数据块以随机的方式分配到各个计算节点，并没有考虑到数据块优先级与计算节点计算能力之间的关系，并且这些信息在计算之前是无法确定的。在这种前提条件下，如果采用上述的思路进行负载均衡，几乎要将所有的数据块以任务迁移的方式进行重新分配。另外，由于随着计算的进行，每个数据块的优先级会变化，但是计算节点的计算能力不会变，那么维持数据块优先级与计算节点计算能力的对应关系，也需要大量的任务迁移操作。这种情况下，进行大量的任务迁移，任务迁移的开销很可能会大于负载均衡处理所带来的收益，因此，这种思路是不可取的。

在上文的分析中，有一点十分明确，负载均衡处理必须尽可能的利用各个计算节点原有的计算状态，有针对性的做微量的调整，尽可能的减少任务迁移的量，以减少负载均衡处理开销。这是设计负载均衡处理决策算法的基本要求。

为了能够有效的疏导整个图中的消息，避免出现消息阻塞的情况出现，复杂均衡处理必须保证，所有的消息能够以等同的速度在图中流动。在Maiter框架中，如果一个图顶点被调度计算一次，那么该图顶点会将它所持有的值经过计算后，以消息的形式传递给它的邻接图顶点。所以每个图顶点被调度一次，该图顶点上的消息就向前传递一次。因此，如果负载均衡处理可以根据每个计算节点的计算能力，调整每个计算节点上的任务量，使得集群中每个图顶点在相同的时间内被调度计算相同的次数，那么就能保证所有的消息都能够以相同的速度向前传播。

保证所有的消息在图中以相同的速度传播的同时，也可以考虑到各个数据块的优先级。实际上，在负载均衡处理之后，每台计算节点上消息的传递速度是一样的，因此，优先级高的数据块无论在哪个计算节点上被调度计算的速度是一样的，优先级高的数据块在哪个计算节点上实际上是不会影响收敛速度的。但是在数据块大小划分合理的情况下，大多数情况下，负载较重的往往要迁移多个数据块才能够解决负载较重的问题，因此负载较重的节点可能要持续一段时间才能够被解决。那么在这段时间内，负载较重的计算节点上负责的数据可能要处于计算速度较慢的状态。如果在负载均衡处理的过程中，总是优先的将优先级最大的数据块迁移出来，那么优先级较大的数据块就能够很快的转移到负载较轻的计算节点上得到计算。这样就可以一定程度上的加快算法的收敛，进而提高整体的计算效率。

### 3.3.2 负载均衡决策算法

在上一小节的思路分析中，负载均衡决策算法的目的其实已经十分明确，就是要通过负载均衡处理，使得集群中所有的消息以相同的速度传播，避免因节点的计算能力不足或负载过重而导致的消息阻塞的现象出现。

决策节点需要通过监测一些信息来实时的掌握每个计算节点上消息传播的速度。由上文分析可知，消息的传播速度与图顶点被调度计算的速度是相等的。那么假设，第*i*个计算节点将其所负责的所有的图顶点调度计算一遍所用的时间为，就可以表征消息的传递速度。即经过时间，计算节点*i*上的所有消息均向前传递一次。因此每个Worker节点只需要将自己计算所有图顶点所用的时间报告给决策节点即可。这样固然简单，但是有一个问题。因为每做出一个决策，完成决策需要一段时间。那么在这段时间内，在统计信息里体现不出决策对于集群负载状态的影响。这样做决策之前的不均衡状态会一直持续，直到决策完成。因此在这段时间内，决策算法不能在做决策，否则会他会做出相同决策。简单的解决方法就是当做出一个决策后，就不在做决策，直到该决策被完成。但是由于在通常情况下，往往需要多个决策才能解决负载倾斜问题。采用这种方式不仅会影响到决策的并行，也需要大量的同步开销。

另外的一个解决方案是，如果决策节点做出一个决策，该决策的影响能够立即反映到决策算法里。通过调整一些数据，决策算法可以立即计算出决策作出之后，集群的负载应该处于什么样的状态。

令*i*代表计算节点的编号，，*m*为集群中计算节点的个数。*j*代表数据块的编号，，*n*为集群中数据块的个数。代表分配到计算节点*i*上的数据块的标号的集合，由于，其中为计算节点*i*在单位时间内处理的图顶点的个数，为第j个数据块所包含的图顶点的个数。由于之和计算节点*i*的计算能力有关，不受负载均衡处理的影响，是个定值。如果做出负载均衡处理决策，决策算法可以通过调整所属的计算节点，可以马上通过计算，求得做出决策后的变化，掌握集群在负载均衡处理后处于的负载状态，即使此时决策还没有被执行。就能够保证，在上一个决策没有完成之前，决策决策节点做出的决策是正确的。

由于在数据迁移之前不会发生变化，因此只需要上报一次。计算进行的程度有关，因此应该适时的进行报告。另外因为要适时的掌握每个数据块的优先级，因此每个Worker节点也要实时的报告自己所负责的数据块的优先级。

由于计算节点不停的报告自己当前的监测信息，决策算法必须能够及时的应用最新的信息，来做出决策。只有使用最新的信息做出的决策，才是最正确的。因此决策算法必须能够快速的做出决策的同时更新相应的监测信息，然后在根据最新的信息做出最新的决策。决策节点做出的决策主要是指导计算节点将特定的数据块迁移到特定的节点上去。由于迁移以数据块为单位，但是将所有的计算节点调整到负载均衡可能需要做出过个决策。为了能够尽可能的应用最新的监测信息，没做出一个决策之后，会跟新监测信息之后再做出新的决策。

下面本文给出决策算法的伪代码，详细的说明决策算法做出决策的过程。代表第*j*个数据块中所有图顶点的优先级的总和。表示的是第*i*个计算节点的偏差量，代表的意义是第*i*个计算节点上的负载偏离平均值的程度。

|  |
| --- |
| **算法3.1：负载均衡决策算法** |
| 1. Begin |
| 1. 接收(*j=0,1,…n*) |
| 1. While(*Running*) |
| 1. Begin |
| 1. 接收： |
| 1. 接收： |
| 1. For *i*=0 to *m* |
| 1. Begin |
|  |
| 1. End |
| 1. 计算节点*，* |
| 1. IF or |
| 1. Begin |
|  |
| 1. 做出决策：将数据块*shard*从节点*a*迁移到节点*b*上 |
|  |
| 2. End 3. End 4. End |

算法3.1的主题部分是一个循环，循环很是结束，由标志位*running*控制。当*running*的值为*true*的时候，代表计算正在进行。在循环开始之前，决策阶段会先接受各个数据块所包含的图顶点的个数。由于该值在计算的过程中是不变化的，因此只需要报告一次。循环开始后，决策节点要接收所有计算节点报告来的监测信息。然后会根据这些监测信息通过公式(3.1)来计算每个计算节点的偏差值。如果该值大于1.96或者小于-1.96就认为该计算节点处于负载不均衡状态，则做出调整。一个决策完成之后，会进入下一次循环，重新跟新监测信息，做下一轮的决策。

## 3.4 数据迁移

将负载较重的计算节点上的一部分任务或者数据迁移到任务较轻的计算节点上是处理负载不均衡的主要方式。上节中本文提到，Maiter框架采用集中式动态负载均衡处理。那么在Maiter框架运行的时候，决策节点也就是Master节点会根据各个计算节点的运行状态做出负载均衡处理决策，而这个决策就是将那些数据从哪个计算节点上迁移到哪个计算节点上。

### 3.4.1 数据迁移产生的问题

进行数据迁移并不是直接将数据放到其他的计算节点上那么简单，在数据迁移的过程中会产生很多复杂的问题。由于Maiter框架是异步类型的图处理框架，不存在同步过程，因此数据迁移只能在计算的过程中进行迁移，不能像同步类型的框架那样，在同步路障结束之后进行迁移。这样就造成了很多的问题，如在数据状态不确定的情况下如何进行数据迁移、迁移哪些数据等问题。

在第二章介绍的Maiter框架实现细节中，可以知道，Maiter框架中的每个计算节点将自己负责的数据分片全部载入到内存，在内存中建立一张状态表，统一的将所有的数据和相关的信息管理在状态表内。换句话说，这些数据被Maiter框架的计算节点看做一个整体统一管理。在负载均衡处理的过程中，迁移一部分数据是不方便。因为首先要在状态表中筛选出一部分数据，并且需要在恰当的时机将这些数据在状态表中删除，在删除的时候，数据的状态应该是确定的，而不应该是变换的，否则很难保证数据迁移后，计算结果的正确性。

另外Maiter框架的数据定位是通过Hash函数完成的，一台计算节点上的数据都具有相同的Hash值，以方便数据定位，但是将其一部分数据迁移到其他的计算节点上，数据的位置变了，但是数据的Hash值不会变，这将给这些迁移数据的重定位带来很大的麻烦。

最后，由于Maiter框架是异步类型的图处理框架，从任务的提交到任务结束，计算一直在进行，因此，Maiter框架的负载均衡处理必须在计算的过程中进行。并不能像同步类型的框架那样，在同步路障结束之后进行。在计算的过程中进行数据迁移，就会导致两个问题。一个是由于数据迁移需要一定的时间，因此迁移数据的计算会被拖延。另一个是由于迁移数据计算的拖延，那么也会影响到和迁移数据相关的数据的计算，这种影响会不断向外蔓延，进而影响到整个集群的计算。在这中影响下，整体计算的正确性能否得到保证，需要严格的数学证明。

### 3.4.2 对Maiter框架数据管理方式的改进

Maiter框架管理数据的方式很不利于数据迁移不利于数据从定位，因此，本文在不显著的增加Maiter框架的开销的基础之上，适当的更改了Maiter框架管理数据的方式。如图3.3左图所示，Maiter框架原来每个Worker节点上只有一个Hash分片，且在内存中值简历了一个状态表，这样很不便于数据迁移。那么本文将Maiter框架的数据管理方式进行更改，如图3.3右图所示。数据集在进行Hash划分的时候，会被换份成更多的数据分片，可能是集群中计算节点的数倍。那么每个Worker节点将负责多个数据分片，并且在在初始化的过程中，为每个数据分片建立相应的状态表，单独的进行维护。在计算的过程中计算节点会选取适当的调度策略，调度不同的数据块进行计算。

在这种数据管理的方式下，每个数据块的数据、状态表、定位信息及其他的一些相关的信息都是单独进行管理的，如果在负载均衡处理的过程中，数据迁移以数据块为单位，也就是说每次都定量的迁移一个数据块到其他的节点上，将是十分便捷的，后续的处理也会十分容易。



图3.3Maiter框架数据管理的改变

Fig. 3.3 The change of data mangemnet of Maiter

以数据块为迁移单位的这种数据迁移方案，首先方便了迁移数据的选取，每次迁移一个数据块，就避免了扫描所有的数据，选取合适的数据进行迁移。另外由于与迁移数据相关的信息都是单独维护的，在迁移的时候，避免了对这些信息的维护和更改，进一步节省了开销。第二方便了决策节点进行决策，决策节点只需要给出迁移的数据块、迁移数据块的所有节点和迁移目标节点就可以完成负载均衡的处理。第三，减少了监测信息量。由于选用的是集中式动态负载均衡处理方式，Maiter框架中的每个计算节点都会不断地将相应的监测信息不断的报告给决策节点。那么以数据块为单位进行负载均衡处理，Worker节点只需要报告本机负责的数据块的监测信息即可，而不再是报告每个图顶点的监测信息。但是由于每个计算节点负责的数据块要远远小于图顶点的量，因此极大地减少了监测信息量，减少了通信开销。第四，方便了迁移数据的重定位，这将在下一小节详细介绍。

### 3.4.3 数据迁移的正确性保证

在前文中已经提到，在计算的过程中进行数据迁移，由于迁移数据的计算被拖延，首先会影响到迁移数据本身的计算，然后影响到与迁移收据相关的数据的计算，进而影响到整个集群的计算。在这一节将详细的介绍，在任意时刻进行数据迁移究竟会对全局运算产生怎样的影响，并探讨这种影响与Maiter框架最终的计算结果直接的关系。

迁移数据块中的任意一个图顶点，假设如果不进行迁移，在时刻发出消息*m*。但是由于数据迁移，使得该图顶点的计算被延误，延误至时刻才发出消息*m*，由于*m*的延误，导致多有的邻接节点依赖于*m*产生的消息也被延误。这种延误会沿着图中的通路蔓延开来，进而影响到整个Maiter集群。这种结果最终会导致的结果只有一个，图顶点接收消息的序列的次序发生了变化。既然数据迁移会造成图顶点接收消息的序列发生变化，如果本文能够证明，对于任意图顶点，接收消息的次序发生变化，不会影响最终计算结果，那么就可以证明，在计算过程中的任意一时刻，进行数据迁移都不会影响计算结果的正确性。

图中的任意一个顶点*a*，从计算开始到结束，它所接收到的消息会构成一个消息队列，将该序列标记为{*，，，，，*}，其中*k*为计算开始到结束a收到的消息的个数，并且每个顶点的*k*可能不同，定义*P*{}为序列{*，，，，，*}中的元素的任意一个排列所构成的不同顺序的序列构成的集合。{}是集合*P*{}中的一个元素，代表由元素*，，，，，*以一个顺序排列所构成的序列。

定理3.1：在图*G*(*V，E*)在进行异步累积迭代计算过程中，对于任意图顶点以任意一个消息顺序{}接收消息进行计算，计算结果均相等。

证明：为了证明最终的计算结果与消息到达的顺序无关，必须推导出任意图顶点上的值的计算公式。在异步迭代计算开始时，任意图顶点*a*的初始值为，在异步累积迭代计算结束时，图顶点*a*上的计算值为，由DAIC计算模型的计算过程我们可以推导出，顶点向顶点*a*发送的消息为，其中*I*(*a*)为指向顶点*a*的顶点所构成的集合。距离*a*点两跳的顶点发来的值为，那么距离顶点*a*为*h*跳的发来的消息为，如果是理论上的收敛值，那么。但是计算结果会是收敛值的近似值，取得理论的上收敛值是不现实的，所以一定存在一个定值，使得在计算结束时任意一个顶点*aV*最远只能收到距离其*h*跳的顶点发来的消息*。*那么我们能推测的计算公式如公式3.2所示。

其中，因为异步累积迭代计算的过程中，未必会接收到所有从一定距离发来的消息。那么从公式3.2可以看出，无论是从相同的距离发来的消息，还是从不同距离发来的消息，最终都是以运算累积在一起，由于在推导DAIC计算模型时，要求运算满足交换律，因此，任意一个顶点*aV*无论以何种顺序收到消息，最终的计算结果都相等。定理3.1得证。

## 3.5 迁移数据重定位

集群中的每个数据都有其相应的定位信息。当一个图顶点想其他的图顶点发送消息的时候，该图顶点会给出消息的目的顶点的顶点标号，但是不会给出目的节点在那个计算节点上。为了在计算的过程中，发出的消息能够正确的到达目标图顶点，进群就必须维护每个顶点的定位信息，集群通过目的图顶点的标号，快速的判断出目标图顶点所在的位置，并正确的进行传输。

比较常用的分布式定位方法有几种。如Hash定位，集群全局采用统一的Hash函数对数据进行划分，将具有相同Hash值的数据放到相同的计算节点上。例如在图划分时，对图顶点的标号进行Hash，将所有标号Hash值相同的图顶点放在同一台计算节点上，这样在消息传递的时候，通过对标号进行Hash，就能够快速的判断出目的图顶点所在的计算节点。Hash定位方法简单有效，多被采用。还有其他的定位方法如范围定位以及复杂一点的分布式散列表定位方法等。

Maiter框架采用的是Hash定位的方式。Hash定位的方式简单有效，但是却十分不利于负载均衡处理中的数据迁移。因为在数据迁移的过程中数据所属的计算节点发生了变化，但是数据的Hash值却不会发生变化。如果仍然采用Hash划分的方式进行定位，就必须对迁移数据的的定位信息单独的进行维护。但是单独维护的前提是必须有效的判断哪些是迁移数据，哪些不是，这无疑会增加数据定位延迟，影响整个集群的计算性能。所以对迁移数据的定位信息进行单独维护，是不可取的。

在上一小节，为了方便数据迁移，本文已经对Maiter框架的计算节点管理数据的方式进行了更改。那么能不能根据目前的数据管理方式，设计出简单有效同时便于迁移数据重定位的定位方法呢。

定位方式的设计目标是：首先应该能够像Hash定位方法一样简单有效。其次适应更改后的Maiter框架的数据管理方式。再次要能够快速的对迁移数据进行重定位，并且在重定位之后，不会增加数据的定位开销。最后，不能增加太多的定位数据维护开销。针对这样的设计需求，本文提出了消息分桶缓存分布式数据定位方式。



图 3.4消息分桶标记定位机制

Fig. 3.4 Message marking points bucket positioning mechanism

如图3.4所示的就是消息分桶标记定位机制的处理过程。在上一小节，本文已经介绍过，数据集会被采用Hash划分的方式划分更多的数据分片，并且每个Worker节点对负责多个数据分片，并且每个数据分片的Hash值是不同的。在这种情况下，当一个图顶点发送消息的时候，通过一次哈希，可以确定消息所在数据块。如果每个Worker节点都存有数据块与计算节点的对应关系，那么就可以确定数据块所在的计算节点，这样就完成了数据定位过程。

如图3.4所示的就是消息分桶标记定位机制的处理过程，虽然在实际实现的时候，为了能够更快的数据定位，实现细节稍有差别，但是基本思路是一样的。例如在3.4图中，每个Worker上的消息缓存被分成与集群中数据块个数相同的分桶，并且每个分桶都会用一个对应的标记，用于标记分桶中的消息所属的计算节点。那么Worker0上产生的消息，会通过一次Hash分配到对应的分桶中，当向外发送时只需要根据分桶的标记发送到对应的计算节点上就行了。

那么消息分桶标记定位机制能不能满足设计要求呢。首先与简单的Hash定位方式相比，消息分桶标记定位机制使用一次哈希也能完成数据定位。虽然简单的Hash定位多了一些标记，但是空间开销几乎可以忽略不计。

消息分桶标记定位机制也极大的方便了数据的迁移。由于数据迁移以数据块为单位，因此数据迁移只会改变数据块与计算节点的对应位置，并且通过Hash仍然可以确定消息所属的数据块。因此在进行数据迁移的时候，只需要改变数据块与计算节点的对应关系就可以。由于数据迁移的决策是由决策决策节点发出的，那么在发出数据迁移决策的时候，决策节点只需采用广播的形式，通知所有的计算节点修改数据块与计算节点的对应关系。并且在修改之后。不会增加数据定位的开销。

在存储开销方面，储存数据块与计算节点间的对应关系，需要一定的空间开销，但是数据划分时，划分的数据块个数并不会过多，至多是计算节点的十几倍，因此空间开销是常数级的。消息在分桶时所用的存储空间只是将原有的缓存改变了一下结构，并没有增加存储开销。

## 3.6 错位消息处理

### 3.6.1 错位消息的产生及影响

本文中所说的错位消息是指由数据迁移造成的没能够正确定位的消息。在复杂的分布式环境中，并在计算的过程中对数据的定位信息进行更改，会带来严重的问题。当决策节点以广播的形式通知说有计算节点修改定位信息之后，每个计算节点何时能够修改完成时不确定的。这种不确定性就会导致错位消息的产生。

当决策节点做出负载均衡处理决策的同时也会通知所有计算节点修改定位信息。由于分布式环境下，每个计算节点没有协同过程，对定位信息的修改的步调是不一致的，那么就会出现这种情况：数据块已经从Worker1节点迁移到Worker2节点。但是由于Worker3节点还没来的及修改定位信息，因此该节点上所有属于迁移数据上的信息仍然会发送到Worker1节点上，其他的计算节点也有可能出现在这种情况。假设某一时刻，所有的计算节点的定位信息全部修改完成。那么从数据块从Worker1上迁移走到到所有计算节点定位信息修改完成这段时间内，会产生大量的错位消息，并且错位信息的数量无法得知。

这些错位消息最大的特点就是无法获知其数量。有可能很多，极端的情况下有可能没有。但是这些错位消息必须要全部投递到正确的计算节点上去，任意一个消息丢失，都会造成最终计算结果错误。

由于每个计算节点都有可能收到错位消息，并且错位信息的数量数不确定的。为了保证最终计算结果的，必须在每个计算节点上都简历处理错位消息的机制，并且从计算开始值到结束，错位消息处理机制必须持续处理，才能够保证负载均衡处理过程中产生的错位消息全部被处理。实际上，这些错位消息的数据并不是很多，但是却要在每个计算节点建立持久的处理机制，如果处理机制设计的不恰当，会极大的增加Maiter框架的整体计算开销。

### 3.6.2 基于缓存的消息中转解决方案

面对这些挑战。本文巧妙的运用了数据缓存中所用到的桶。解决了错位消息处理的同时，也方便了数据迁移。



图3.5错位消息处理示例

Fig. 3.5 Dislocation message processing examples

为了清晰的表述错位消息的处理过程，假设数据被换分成成6个数据块，集群中只有3个计算节点。如图3.5所示，每个计算节点负责两个数据块，对应于3.5图中每个计算节点上深色数据块。每个计算节点上的白色块代表对应与缓存相应数据块的桶，在计算的过程中，计算机节点上发送的消息会经过一次哈希先分到对应的桶中进行缓存。

在进行代码实现的时候，真实的数据块和缓存消息的桶会继承于相同的类，也就是它们具有相同的父类。更改消息的接收方式，采用多态的形式，当接收到消息的时候，会调用相同的函数，但是如果消息属于该计算节点所负责的数据块，机会参加计算，如果不是，则缓存到相应的桶中，当发送信息的时候，会被转发到正确的计算节点上去。

举例说明，假设决策节点做出的负载均衡决策为将数据块0从Worker0节点迁移到Worker1节点。在做出决策的同时，决策节点也会通知多有的计算节点修改对应的数据块定位信息。假设数据块0已经从Worker0上迁走，但是Worker2仍没有来的及修改定位信息，因此Worker2上发送的属于数据块0的消息仍然会发送到Worker0节点。由于实现了多态，Worker0节点会接收这些错位的消息。接收到错位消息之后，Worker0节点会将这些错位信息缓存到相应的缓存中去。因为Worker0已经将定位信息修改完，所以当Worker0发送消息的时候，这些错位信息会正确的发送到Worker1节点上去。

这样，本文就巧妙的运用缓存解决了错位信息的处理问题。这种处理机制，几乎不会增加计算节点的开销，并且这种机制与错位消息的数量是没有关系的，一定能够将所有的错位消息发送到正确的位置。

本文将这些存储消息的桶称为虚拟数据块，因为它们在某种意义上在就算节点上代替了不在该节点上的数据块。虚拟数据块和真实的数据块具有相同的父类，极大地方便了数据迁移。如果一个计算节点需要将一个数据块迁走，实际上就是使用虚拟数据块代替真实的数据块之后，在将真实的数据块中的信息迁走。如果一个计算节点需要接收一个数据块，那么实际上，该计算节点只需要接收该数据块的相关信息，建立一个真实的数据块，然后替换相应的虚拟数据块。由于虚拟数据块和真实的数据块具有相同的父类，父类指针可以指向虚拟数据块和真实数据块。那么复杂的数据块之间的替换，通过简单的指针修改就可以完成。

### 3.6.3 基于缓存的消息中转机制正确性证明

在上一小节只是给出了简单的例子，在该情况下，基于缓存的中转机制能够将消息正确的发送到正确的计算节点。但是实际情况要复杂的多。由于任意一条消息丢失，都会导致整个计算错误，因此，在理论上给出基于缓存的消息中转机制的正确性证明是必要的。

在数据迁移的时候，集群中的节点可以分为三中。一种是迁移数据的所有这，只有一个节点，标记为OW。第二种是迁移数据的目的节点，也只有一个，标记为DS。第三种是节点既不是说有节点也不是目的节点，可能有多个，标记为OT。OW节点在迁移数据迁走之后修改定位信息。DS、OT节点在任意时刻修改。修改下面本文给出基于缓存的消息中转机制的正确性证明。如定理3.2所示。

定义3.1。在集群的全局时钟*t*时刻，OW节点修改了定位信息，那么所有在*t*时刻及t时刻以后，认为修改定位信息的DS节点和OT节点所发送的与迁移数据相关的全部为错位消息。

定理3.2：采用基于缓存的消息中转机制处理错位消息，OW节点、DS节点、OT节点以任意的次序修改定位信息，所有的错位消息均能正确的传送到DS节点。

证明：三类节点修改的次序是三者的全排列，一共有6种情况。

首先，假设集群中的消息传递延迟为0，即在任意的全局时刻*t*，集群中不错在t时刻以前发送单t时刻仍未到达目的节点的消息。本文用节点类型标号的下角标来标记定位信息的修改的次序。下面分别讨论6中情况。

1. ，,由错位消息的定义可知，只要OW节点最后修改，不会出现错位消息。DS节点在OT节点修改之后修改，在DS节点修改之前，OT发送的与迁移数据块相关的消息会先到达DS节点，然后由其转发给OW，由于OW为修改定位信息，仍具有处理能力。
2. ，OW节点最后修改，不会出现错位消息。
3. ，在OW修改之前不存在错位消息。在OW修改以后，DS点发的的相关消息为错位消息，OW收到之后，判断不是自己的消息，会将其缓存在对应的缓存中，由于此时OW的定位信息已修改，因此这消息会在OW节点发送正确的发送到DS点，直到DS点定位信息修改。此时会在OW和DS之间出现消息环路。
4. ，在OW修改之前不存在错位消息。在OW修改以后，OT节点发送的相关消息为错位消息。OW收到之后，判断不是自己的消息，会将其缓存在对应的缓存中，由于此时OW的定位信息已修改，因此这消息会在OW节点发送正确的发送到DS点，直到OT点定位信息修改。
5. ，OW节点已经修改后，DS节点和OT节点发送的相关消息均为错位消息。由于OW已经修改，会将错位消息全部转发给DS节点，直达OT节点修改完成。此时在OW和DS节点之间会存在消息环路。
6. ，OW节点已经修改后，DS节点和OT节点发送的相关消息均为错位消息。由于OW已经修改，会将错位消息全部转发给DS节点，直达DS节点修改完成。此时在OW和DS节点之间会存在消息环路。

在所有的6中情况下，错位消息均能被正确的传送到正确的位置。在实际情况下，消息的传送存在延时，而消息传送延时产生的唯一影响是会导致节点的真实定位信息修改的时间延后，进而导致三类节点的定位信息修改次序发生变化，但是无论如何变化，一定是上述6中情况中的一个或多个。因此，在消息传送存在延时的情况下，错位消息仍然能够正确传递。

由于OT节点可能是多个，那么所有的OT节点的修改时间不可能完全一致，但是无论各个OT节点在何时修改，都是6中情况中的一种或多种，因此错位消息人能够真确处理。至此，基于缓存的消息中转机制的正确性得证。

## 3.7 负载均衡处理详细过程



图3.6Maiter框架负载均衡处理构架图

Fig. 3.6 Load balancing process architecture of Maiter framework

在前面几节中已将详细的介绍了Maiter框架负载均衡处理过程中所遇到的问题的解决方案，这一节将详细的介绍Maiter框架的负载均衡处理是如何一步步进行的。

Maiter框架在任务提交到任务结束分为三个阶段：初始化阶段、计算阶段、结果输出阶段。负载均衡处理机制主要在计算阶段其作用。就是说当Maiter框架进入计算阶段之后，负载均衡处理机制启动，当计算阶段结束之后，负载均衡处理机制关闭。图3.6给出了Maiter框架的负载均衡处理的构架图。

在图3.6中，由1个Master节点和3个Worker节点组成。Master节点作为负载均衡的决策节点。负载均衡决策过程以及检测信息的维护由Master节点上的DecisionThread进程完成。每个Worker节点上有两个进程，分别是BalanceThread和MainThread，分别负责负载均衡处理和图处理过程。MainThread中有若干个数据块，四列的代表该节点负责的数据块，两列的代表虚拟数据块。BalanceThread进程中会保留一个指针，该指针是数据块和虚拟数据块的父类，该指针会视情况指向迁移数据块。接下来，本文将详细的描述一下Maiter框架的负载均衡过程。

如图3.7，给出了Maiter框架负载均衡处理过程的流程图，给出了负载均衡处理过程中所涉及到的各个步骤，各个步骤的详细处理细节如下：

1. Maiter框架初始化完成进入计算阶段
2. Master节点的DecisionThread和Worker节点上的BalanceThread同时启动，此时负载均衡机制启动。
3. BalanceThread向DecisionThread报告*t、num、pri*等监测信息。
4. DecisionThread接收监测信息，初始化监测信息，对集群中的所有计算节点有了整体的了解之后，启动决策算法，如果没有出现负载倾斜，不作出任何决策。如果出现负载不均衡问题，则做出负载均衡处理决策。
5. Woker节点收到负载均衡处理决策后，做相应的判断。如果该计算节点上迁移数据块的说有者，进程进入第6步，进行balance shard owner处理。如果是迁移数据块的目的节点，进程进入第7步，进行balance shard destination处理。如果都不是，则将迁移数据块的位置与计算节点的对应关系中，数据块的位置改为目的节点。
6. 在Balance shard owner阶段，BalanceThread会初始化一个虚拟数据块，并有Pointer指向该虚拟数据块。初始化结束之后，用初始化好的虚拟数据块代替迁移数据块，并将Pionter指向被替换的迁移数据块。随后修改数据块定位信息，并将迁移数据块中的信息以消息的形式发送给目的节点。
7. 在Balance shard destination阶段，BalanceThread会首先从HDFS系统中读取迁移数据块所包含的图数据，利用这些信息初始化一个迁移数据块，初始化完成之后，替换目标节点上对应的虚拟数据块，修改迁移数据块定位信息，并将Pioneer指向代替后的虚拟数据块，虚拟数据块中的消息，会在稍后的计算中传给迁移数据块。
8. 至此，一轮负载均衡处理完成，回到第3步，进行下一轮负载均衡处理。



图 3.7 负载均衡处理流程图

Fig. 3.7 Load balancing process flow chart

## 3.8 本章总结

在这一章，本文详细的介绍了Maiter框架的负载均衡处理解决方案，巧妙的采用缓存解决了错位消息的问题，利用多态，有效的方便了数据迁移，提出了基于Hash的两级分布式定位方式，在解决迁移数据重定位的同时，并不增加消息的定位开销。

整个Maiter框架负载均衡处理的实现，对原有的Maiter框架的结构改动较大，编程代码量较大，需要很大的工作量。实验显示，本文使用的基于数据块的集中式动态负载均衡处理方案在出现负载不均衡问题时是有效的，并且在没有负载倾斜问题是，并不增加太多的计算开销。

# 第4章 异步累积迭代算法

Maiter框架的计算效率优于hadoop、Spark、Graphlab等框架，因此本文希望Maiter框架能够具有更高的通用性，能够解决更多的实际问题。那么异步累积迭代算法的研究的主要工作就是通过改进分布式算法的计算方式，进而使该算法能能够在Maiter框架上实现，以提高该算法的计算效率。Maiter框架目前已经支持很多的分布式算法，如PageRank算法、但愿最短路径求解、Sum-Product算法、连用子图划分算法等等。本章通过改进SimRank算法，使之能够在Maiter框架上实现的同时，进一步扩展了Maiter框架的API，有效的提高了Maiter框架的通用性。

## 4.1 研究内容介绍

随着数据采集和数据存储等技术的发展，各类应用所处理的数据量不断增大，人们采用机器学习和数据挖掘算法对这些庞大的数据进行处理和分析。由于所涉及的数据量巨大，人们通常利用分布式环境进行大数据的处理。如何有效利用分布式环境设计高效的大规模机器学习和数据挖掘算法，成为当今大数据领域的热点研究问题。

SimRank算法利用图中各顶点之间的相互关系评估任意两顶点的相似度。它被广泛应用在社交网络、引用关系网络、链接预测等诸多领域中。例如，SimRank算法被用于评估购物网络中产品之间的相似度或社交网络中朋友之间的相似度，再根据相似度来进行产品推荐或好友推荐。近年来，随着推荐系统的流行，SimRank算法的重要性也更加突显出来。

由于单个计算机的计算能力和磁盘存储空间有限，所以单机上实现的SimRank算法，在计算效率上以及处理数据的规模上，都难以满足大数据的要求。分布式计算通过多台计算机协同计算、共享磁盘存储空间等方式，有效地解决了单机计算的限制，因此在实际应用SimRank算法时，往往需要设计高效的分布式SimRank算法来支持大数据处理。并且随着SimRank算法应用的愈加广泛，设计高性能的分布式SimRank算法也愈加重要。

目前已经有很多SimRank算法的分布式实现。MapReduce模型和Hadoop框架是目前最流行的大规模数据处理分布式编程模型和框架。在Hadoop框架上实现分布式SimRank算法时，虽然实现比较简单，但是通信量大，空间复杂度高，并且产生的大量中间结果，影响SimRank算法的计算效率。另外，由于SimRank算法是同步迭代算法。同步算法的计算效率受到超级步的限制，每次计算何时开始，依赖于上一次迭代何时结束。多次迭代的多个同步过程在大规模异构分布式环境下将造成大量的同步开销，导致SimRank算法在分布式环境下不能高效地实现。虽然目前有很多新型分布式处理框架支持高效的迭代计算，但是SimRank算法只具备同步计算形式，框架的改进对SimRank算法的效率提升有限。

Liangliang Cao等提出的Delta-SimRank算法[24]采用迭代-累积（迭代地计算两次迭代的差值，然后将差值累积起来，得到最终相似度值）计算形式计算相似度值。迭代-累积计算形式避免了计算过程中收敛顶点参加运算，从而减少了计算量和通信量。但是Delta-SimRnak算法是SimRank基于MapReduce编程模型的改进算法，它仍然同步地执行迭代，没有避免同步开销。另外，Delta-SimRank算法需要额外的MapReduce作业完成差值累积，增加了一部分计算开销。

针对于SimRank算法和Delta-SimRnak算法所存在的同步开销问题，本文提出了可异步执行的Asyn-SimRank算法，在Asyn-SimRank算法的计算过程中，各个计算节点相似度值的更新，不依赖上一次迭代的计算结果，更新操作可以随时进行，无需等待，节省了同步开销。本文在异步迭代的基础之上提出了关键点优先调度计算，进一步提升了Asyn-SimRank算法的收敛速度。本文的主要贡献总结如下：

* 在Delta-SimRank算法的基础之上，提出可以异步计算的Asyn-SimRank算法。Asyn-SimRank算法在保持Delta-SimRank算法优点的同时，进一步实现异步计算，节省了同步开销。本文同时证明了Asyn-SimRank算法的正确性和收敛性。
* 在异步迭代的基础之上，提出关键点优先调度计算。优先地计算影响全局收敛的关键点，有效地提升了Asyn-SimRnak算法的收敛速度。本文同时证明了关键点优先调度计算对于提升算法整体收敛速度的有效性。
* 改进了分布式异步计算框架Maiter[5]并扩展了Maiter的API，在Maiter框架上实现了Asyn-SimRank算法。
* 在真实的数据集上对Asyn-SimRnak算法进行了性能评估，实验结果显示了Asyn-SimRank算法的优良性能。

## 4.2 SimRank算法原理与Hadoop实现

### 4.2.1 SimRank算法原理

SimRank算法的基本思想是：如果两个对象与相同的对象或相似的对象有关系，那么他们是相似的。如果将对象作为一个顶点，对象间的关系当作一条边，就可以得到一张对象图*G*=(*V*，*E*)，*V*为对象的集合，*E*为关系的集合。SimRank算法基于图*G*进行相似度计算。

如果将对象*a*与对象*b*之间的相似度记为*S(a,b)*，SimRank算法的迭代计算公式为：

其中和分别为顶点和顶点*b*的指入顶点集合；的取值范围为[0,1]；系数是一个阻尼系数，其取值范围为(0,1)。当时，；当时，。由公式(1)经多次迭代，就可以得到最终的顶点间的相似度。

为了形象地描述SimRank算法的计算过程，可以构造一张顶点对图[12]来直观地观察SimRank算法在计算过程中消息的传递过程。G2图的构造方法为：如果且，则顶点对。如果边且，则边。

图4.1中的(*a*)图为原始图*G*，图(*b*)为*G*转化而来的顶点对图。以顶点2，3，4，5为例，顶点2，3，4，5出现在*G*图中，那么由图的构造规则，顶点对(2,4)，(3,5)出现在图中。又由于*G*图中有边23，4，那么在图中就有一条由顶点对(2,4)指向(3,5)的边。其他顶点的转化过程类似。



图4.1*G*图到图的对应实例

Fig. 4.1 An example of and

在构造图过程中，有一些边和顶点对在转化过程中被删除了，主要有两种情况：1)由于*S*(*a,a*)始终为1，(*a,a*)接收到的值并不在(*a,a*)上参加运算，因此向(*a,a*)传值没有必要，可以删去指向顶点对(*a,a*)的边；2)入度为零的顶点对没有初值也没有非零值传入，始终为零，没有必要参加运算，可以删除入度为零的顶点对(*a,b*)及其所有出边，其中。在删除顶点对(*a,b*)及其所有出边后，会导致一些顶点对(*c,d*)入度为零，其中，级联地删除这些顶点对及其所有出边。删除这些点可以节省计算开销。

以图(b)中的部分顶点对为例，在SimRank算法计算开始时的第一次迭代过程中，由上文的初值设定规则，只有顶点对(1,1)的相似度不为零。顶点对(1,1)将自己的初值1传给顶点对(2,4)，(2,4)接收到该值后，利用公式(4.1)计算出自己的相似度值，在第二次迭代时，(2,4)再将自己的相似度值传给顶点对(3,5)，第三次迭代时，(3,5)再将自己的相似度值传给(1,4)，依次类推，直至算法收敛。

由上例可以看出，SimRank算法在计算的过程中消息是从一个顶点对传给另一个顶点对，即两顶点的相似度依赖于各自相邻顶点之间的相似度，其传递过程确切地反应了SimRank算法的计算原理。

图可以将计算顶点间关系的SimRank算法转变成顶点对值的计算，以便于SimRank算法在以顶点为中心的分布式框架上实现，这在4.5.1小节将详细介绍。

### 4.2.2 Hadoop框架上的实现

SimRank算法能够应用到大规模数据集，一个重要的特性是它可以在分布式编程模型上，以消息传递的方式实现，这样就可以有效地利用分布式资源，进行大规模数据集上的相似度计算。SimRank算法在MapReduce编程模型上的实现如算法4.1所示。

MapReduce编程模型将每次迭代作为一个Job来完成，每个Job中*Map*阶段负责将顶点对的相似度值发给邻接顶点对，*Reduce*阶段则负责将其他顶点对发来的相似度值累加起来计算出自己的相似度值并将结果输出，下一个Job用这些输出结果作为输入信息进行下一次迭代。

基于MapReduce模型实现的SimRank算法存在如下若干问题：

(1)存储复杂度高。在*Map*阶段，由于分配的顶点对是任意的，*Map*要获得每个顶点对中顶点的邻接表，并且保证下一次迭代时仍然可以用到这些信息，每个*Map*任务只能存储整张原始图G。

(2)计算复杂度高。由于SimRank算法计算的是图中任意两个顶点的相似度。所以该算法的实际计算的组合顶点对(*a,b*)的数量级会达到，其中*n*为原始图中顶点的个数。假设每个顶点的平均邻接顶点数为*p*，每个顶点对会给邻接顶点对发送信息，平均发送次数为*p2*，那么就可得出，SimRank算法一次迭代的时间复杂度为*O(n2p2)*。由于处理的数据是海量数据，*n*通常很大，所以实际的计算量非常大。

(3)通信复杂度高。SimRank是高通信的算法，每次迭代计算任意两点的相似度，在Map阶段将发送*p2*个消息，经过Shuffle阶段传递给对应的Reduce，所以每次迭代的通信量是*O(n2p2)*。

(4)同步代价高。每次迭代对应的每个MapReduce作业存在两次全局同步，即Shuffle过程的同步和Job之间的同步。全局同步要求所有任务结束后才可以进行下一步的操作，这样就使速度较快的计算节点等待计算较慢的计算节点，造成不必要的全局等待开销。

|  |
| --- |
| **算法4.1** MapReduce上实现的SimRank算法 |
| 输入：图*G*，初始值集合 |
| 输出：相似度值结果集合 |
| 1. **For** *t* = 0 to *T*-1 |
| 1. ***Mapphase*** 输入: < *key*=(*a,b*) ,*value*= > |
| 1. **Foreach** (*c,d*) **|** *cO(a)，dO(b)* |
| 1. 发送< *key*=(*c,d*)，*value*=> |
| 1. **End Foreach0** |
| 1. ***Reducephase*** 输入：< *key* = (*c,d*)，*value* = *vs*[ ] > |
| // *vs*[]为具有相同*key*值的所有*value*值构成的数组 |
| 1. **IF** *c = d* |
| 1. *=* 1 |
| 1. **ELSE** |
| 1. = *sum*(vs) 2. **End IF** |
| 1. 输出< key=(*c,d*), *value*= > |
| 2. **End For** |

总之，SimRank算法在MapReduce编程模型上的实现，虽然实现了相似度的计算，却并不能保证计算的高效性，消耗大量的计算资源和时间成本。接下来，本文将提出一种高效的SimRank改进算法，它可以避免MapReduce编程模型上实现的SimRank算法的若干问题。

## 4.3 Asyn-SimRank算法

本章首先简要地介绍Delta-SimRnak算法，并在Delta-SimRnak算法的基础之上提出Asyn-SimRank算法，并证明Asyn-SimRank算法的收敛性和正确性。

### 4.3.1 同步迭代-累积形式—Delta-SimRank

由4.2.2节SimRank算法的分布式实现可以看出，传统的SimRank算法在迭代计算过程中，所有的顶点对都会参加后续的运算。但是实际上，有很多的顶点对的相似度值已经收敛，在后续的计算过程中不会再变化，继续参加运算会无谓的消耗计算资源。

为了避免收敛的顶点对继续参加运算，Dlta-SimRank算法对公式(4.1)进行了如下变换：

对于任意的顶点对(*a,b*)，当时：

令：

=

**-**

=

=

当时，则有：

从前文推导过程可以看出，迭代的差值也是可以迭代地进行计算的。这样就可以先迭代计算差值，然后再将这些差值累积起来：

当时：

当时：，。

由公式(4.2)可以看出，Delta-SimRank的计算过程被分为两步，（1）迭代：迭代的计算两次迭代的差值；（2）累积：将迭代差值累积起来。本文将这种迭代过程总结为迭代-累积计算方式。由于收敛顶点对的相似度值在计算过程中不再变化，因此这些顶点对迭代差值一定为零，在后续计算中迭代差值为零的定点对不再参加运算，进而减少了计算量。

### 4.3.2 异步迭代-累积形式—Asyn-SimRank

上文推导出的公式(4.2)是同步执行的迭代-累积计算方式。所谓同步形式，就是每一次累积的迭代差值恰好是第*k+1*次和第*k*次迭代结果之差。但是为了获得更高的运行效率，一个可行的计算方式是让每个顶点对的迭代-累积操作异步地进行。任意顶点对的迭代-累积操作可以随时执行，不必等待两次迭代的差值全部算出来，才开始执行后续操作。为了达到异步执行的目的，就必须打破同步迭代对每次累积差值大小的限制。若用表示顶点对(*a,b*)在第*k*+1次迭代中接收到的差值之和，则有：

其中，，下标*p*为顶点对(*a,b*)的指入顶点对的个数，即顶点对(*a,b*)入度。在同步方式下可以保证{}全部到达之后才开始后续的操作。但是在异步执行的过程中，它们未必能够同时到达，那么假设一种简单的情况，在某一时刻{}首先到达，而{,}随后到达，其中，此时令：

那么就有：

在到达而未到达时就开始迭代-累积操作，那么就可以得到如下计算过程：

当到达的时候：

=

=

==

由以上过程可以发现，在假设的情况下，即使是顶点对(*a,b*)在计算时所需的差值并没有全部到达就开始接下来的操作，最终仍然可得到正确的结果。当然这只是在异步计算的实际情况中最简单的一种情况。

为了能够涵盖异步计算过程中所有的情况，下面不再像上例中那样将顶点对(*a,b*)接收到的差值*∆*的个数限定在一次迭代的*p*个，而是将从计算开始到计算结束顶点对(*a,b*)接收到的所有值均算在内，假设为*q*，*q*在理论上趋于无穷。那么顶点对(*a,b*)的相似度值异步计算的过程可表示为公式(4.3)。

其中，{}表示当前时刻(*a,b*)已经接收到的差值集合。公式(4.3)就是本文提出的Asyn-SimRank算法的计算公式。该公式所表示的计算方式就是异步计算方式：在任意时刻，每个顶点对都可以利用当前时刻所接收到的差值进行接下来的运算，而完全不用等待某一个差值是否接收到。这就是说整个计算过程不再像Delta-SimRank那样，等待一次迭代的*p*个差值全部接收到之后才开始后续操作，而是只要接收到了差值就可以开始计算。

但是前文的例子中，本文简单的验证了在一次同步迭代中，(*a,b*)接收的差值分两次到达时，仍然可以得到正确的结果。但是在公式(4.3)中，(*a,b*)当前时刻接收到的差值集合为{}，由于在异步计算中完全不限制任何顶点对的计算，各个顶点对的计算异步执行，可能有快有慢，那么就可能出现以下情况：某个顶点对计算的快，已经达到同步的情况下迭代很多次的程度，此时它向(*a,b*)发来。而另一个顶点对可能只达到在同步情况下迭代较少次的程度，此时它向(*a,b*)发来了。除此之外，发来}的顶点对都处于不同的同步迭代程度，而在(*a,b*)上这些差值却同时参加了运算。这种情况下，异步计算最终结果的收敛性和正确性必须有严格的理论保证。这正是下一节所解决的问题。

### 4.3.3 收敛性与正确性证明

在证明之前，先简要描述证明的主要思路。实际上无论是Delta-SimRank还是Asyn-SimRank算法，其主要的计算过程都是将初始差值沿所有可走的路径一跳一跳的传播出去，每经过一个点，就在该点进行计算并被累积。然后再将计算出的新的差值继续向前传。下面通过一个类比实例来形象地描述证明思路。

类比实例：该计算过程十分的类似于加工物品的流水线。Delta-SimRank是所有顶点对同步的将接收到的差值计算并累积后，将计算后的差值传给邻接点，就好像加工流水线上的每个人每一次都加工特定数量的物品之后，等待最慢的人将自己特定数量的物品加工完成，大家再同时将处理好的物品交给下一批人进行后续处理。而Asyn-SimRank是接收到差值后就马上计算并累积，并将新差值向前传。就好像流水线上的每个人每拿到一个物品就马上加工，完成后马上交给下一批人进行后续处理。但是比较这两种工作方式，虽然可能有快慢之分，但是两种工作方式却是等效的，不管哪种工作方式，每个物品都会一道工序不差的完成处理。

对应到两个算法，无论是哪种计算方式，每个顶点对的初始差值都会经过路径上的每一个点，被计算并累积，并最终都会到达目的顶点对。因此只要Delta-SimRank算法收敛，则Asyn-SimRank算法收敛，并收敛到相同的值。

接下来，本文将通过定理4.1证明Asyn-SimRank算法的收敛性。通过引理4.1，引理4.2以及定理4.2来证明Asyn-SimRank算法的正确性。

定理4.1：如果在进行无穷次迭代之后Delta-SimRank算法收敛，那么异步地进行无穷次迭代-累积操作之后，Asyn-SimRank算法也收敛。

证明：文献中给出的Delta-SimRank算法是SimRank算法的等价推导，等价推导可保证如果SimRank算法收敛则Delta-SimRank算法一定收敛，并且收敛到相同的值。其等价推导过程本文已给出。SimRank算法的收敛性在文献[12]中已经给出，在此不做赘述，设其收敛值为，那么Delta-SimRank算法也收敛，其收敛值也为。

首先由Delta-SimRank的计算公式(4.2)可以推导出，经过*k*次迭代之后：

其中

由公式(4.4)可以看出，Delta-SimRank算法进行*k*次迭代计算后，由于所有的顶点对都会同步完成迭代-累积操作，所以可以收到所有的以(*a,b*)为终点，距离它跳数小于或等于*k*的顶点对传来的差值。公式(4.4)中的第*m*项就对应以(*a,b*)为终点，距离它为m-1跳的顶点对传来的差值。

对于Asyn-SimRank算法，为了表示点(*a,b*)接收到的差值，定义时间序列*，，，……，*。并定义*S=*{}，其中表示一个顶点对集，*⊂，*并且中的顶点对发出去的所有差值在到时间段内全部被其邻接顶点对接收，显然Delta-SimRank迭代计算的过程中产生*S=*{}，到为完成第*k*次迭代所用的时间段。

定义时刻任意顶点对(*a,b*)的相似度值为，由式(3)可以得到时刻：

其中，时刻，(*a,b*)最远能够收到距离它k跳的顶点对传来的差值。如果(*a,b*)收到()传来的差值，那么对于路径上的所有k个点 (*a,b*)，()，()，，()一定有(*a,b*)，()，()，…，()。

将公式(4.5)与公式(4.4)相对比，由于，当取Delta-SimRank的迭代次数为k时，公式(4.5)与公式(4.4)具有相同的项数，并且公式(4.5)中的每一项都小于或等于公式(4.4)中对应的项，对于任意顶点对(*a,b*)，显然有，又由于当*k*时，收敛，因此有：当*k*时，收敛。设其收敛值为。Asyn-SimRank算法收敛性得证。

引理4.1：对于任意的*k*，总有成立。

证明：在定理4.1的证明中已经得出该结论。证毕。

引理4.2：当*k*，一定能够找到一个值，使得成立。

证明：对于任意的顶点对(*a,b*)，当*k*，总能够找到这样一个，，使得*S=*{}包含所有可走的以(*a,b*)为终点跳数小于或等于*k*的路径，即：(*a,b*)可以接收到所有距离其跳数小于或等于*k*的顶点对发来的差值。又由于，此时(*a,b*)可能接收到距离它跳数大于*k*的顶点对发来的差值，所以。证毕。

定理4.2：如果在进行无穷次迭代之后Delta-SimRank收敛，其收敛值为，那么异步的进行无穷次迭代-累积操作之后，Asyn-SimRank也收敛，其收敛值为，并且有。

证明：由引理4.1和引理4.2可得，并且当*k*时，，又由定理4.1，收敛于，也收敛于，由夹逼准则可得与收敛到相同的值，即：。Asyn-SimRank算法正确性得证。

## 4.4 关键点优先调度

本节在Asyn-SimRank算法异步计算的基础之上提出关键点优先调度计算，来提升Asyn-SimRank算法的全局收敛速度，并证明关键点优先调度计算在提升Asyn-SimRank算法全局收敛速度方面的有效性。

### 4.4.1 基本思想

在分布式环境下，每个工作节点负责一部分顶点对相似度值的计算，在任意单核工作节点上，每次只能有一个顶点对参加计算，那么就必须按一定的顺序调度不同的顶点对参加计算，最简单的方法就是轮询式的调度。假设在同步计算过程中，顶点对(*a,b*)在第*k*次迭代时接收到的差值集合为}，那么同步计算时，这些差值会在(*a,b*)上迭代一次得到，然后将累积后再发送出去，显然一次迭代-累积操作就可以完成这些运算。

但是在异步计算的过程中，如果集合中}中的元素分若干次到达，而(*a,b*)恰好被轮询调度了若干次，因为异步计算接到差值就进行计算，这样就有可能在同步时，一次迭代-累积操作可以完成的计算，在异步计算过程中却要多次，这样无疑增加了计算量，这些增加的计算量会抵消掉异步计算所带来的收益。

解决这个问题很直观的方法就是设置缓存，将接收到的差值缓存到一定数目再参加运算，避免差值到达就直接参加运算，导致迭代-累积操作过于频繁，但是单一的使用缓存有一定的难度，比如如何设定缓存的大小。如果设置的过大，会造成大量的点没有差值计算而等待，增加计算的时间开销。如果设置的过小，显然又不能有效的避免迭代-累积操作过于频繁。

本文提出的方法是关键点优先调度。主要的思想是：异步计算过程中顶点对何时参加运算都是任意的，那么就将计算的机会尽量地分给那些对全局收敛影响较大的点，本文将这样的点称为关键点。这样就可以合理的分配计算资源，使得关键点及时的被调度计算，减少无关紧要的点的计算机会，从而减小计算频率，减少计算量。那么主要的问题是哪些点优先地计算会得到更快的收敛速度，即：如何准确地找到关键点。如果在计算中能够动态地确定关键点，并给这些关键点赋较高的优先级，在异步计算中优先地被调度计算，那么就可以实现关键点优先调度，节省计算量。

本文发现优先计算接收到的差值之和**|**较大的点可以得到较快的收敛速度，接下来，本文将证明基于的关键点优先调度计算的有效性。

### 4.4.2 理论证明

令向量()为所有的顶点对的相似度值构成的相似度值向量，其中为向量的1-范数，为全局收敛时所有顶点对的相似度值构成的收敛相似度向量。由公式(4.5)可知，采用迭代-累积计算形式，由于每次累计的差值大于零，因此每个顶点对的相似度值是递增的，进而也是递增的，由于收敛，会趋于定值。

定理4.3：在异步计算过程中，进行相同次数的迭代-累积操作，优先地调度顶点对(*a,b*) =**|**∆Ṗ参加计算得到的相似度向量为，轮询调度计算得到的相似度值向量为，更接近收敛点即：。

证明：在异步计算的过程中，将|作为该顶点对的优先级，即：|∆Ṗ越大的点，越优先地进行迭代-累积计算。假设关键点优先调度和轮询调度，从计算开始到当前时刻都进行了*k*次迭代-累积操作。关键点优先调度产生的新差值向量为，如果顶点对*i*没有进行迭代-累积操作，则，如果进行了多次，则为多次累积的差值之和。轮询调度计算产生的新差值向量为。由于在关键点优先调度计算过程中，总是选择顶点对(*a,b*)=|∆Ṗ优先调度，显然有，因为向量中的元素总是选择该时刻最大的差值，中的元素则是轮询式选择的。那么进行累积之后，，，显然有，进而得到。证毕。

## 4.5 Asyn-SimRank算法的分布式实现

本章将结合分布式环境的特点以及图例详细地阐述Asyn-SimRank算法如何在分布式环境下实现。

### 4..5.1 输入图的预处理

SimRank算法计算的是两顶点的相似程度，但很多以顶点为中心的分布式框架，由于只支持在图顶点上运行更新操作，所以不能直接进行顶点关系的计算。如果利用4.2.1节提到的将原始图G构造成图的方式，将顶点间关系的计算，转化为顶点对上相似度值的计算，就可以在以顶点为中心的分布式框架上实现SimRank算法。

这种方式有诸多的优点。一方面消息恰好是按图的结构传递的，在进行数据发送的时候只需用到图中顶点对的邻接表，那么集群中的每个计算节点只需存储所处理的顶点对的邻接表，不再需要存储整张原始图。另一方面，可以利用4.2.1节所提到的删减方式，对中一些无用的边和点进行删减，可以有效的减少计算量。

### 4.5.2 分布式环境上的实现

为了能够发挥出Asyn-SimRank算法异步计算和关键点优先调度计算的优势，必须选取合适的异步分布式框架来实现Asyn-SimRank算法。Maiter框架[5]支持异步计算，并且可以在计算的过程中动态的设定顶点的优先级，并进行优先 级调度，这为实现Asyn-SimRank算法的异步计算和关键点优先调度计算提供了条件。



图4.2Maiter上Asyn-SimRank的实现

Fig. 4.2 The implementation of Asyn-SimRank on Maiter

Maiter是Master-Slave结构的分布式框架。包含一个Master控制节点和多个Slave计算节点。Master节点负责任务的提交、任务的分配和部署以及负载均衡。Slave节点则负责完成相应的任务分片。

在Maiter框架上实现Asyn-SimRank算法时，每个Slave节点需要完成的工作就是迭代-累积操作。在任务开始时，Master节点会将相应的数据分片partition分配给Slave节点，其中包含顶点对、顶点对的邻接表、相似度初值、迭代差值初值等信息。Slave节点为partition中的每个顶点对保存3个变量：唯一标识*K*、相似度累积值*V*、迭代差值*ΔV*。如果该Slave节点的partition中有*m*个顶点对，那么就会创建一个长度为*m*的三元组数组来存储所有顶点对的*K，V*和*ΔV*。在构建的同时，Slave节点会用partition中的信息，初始化这些变量。

|  |
| --- |
| **算法4.2** 分布式Asyn-SimRank算法 |
| 输入：图 |
| 输出：顶点对相似度值 |
| 1. **While** not converge |
| 1. **Foreach** (*a,b*) in partition //partition为数据分片 |
| 1. **IF**(|| !*checkpri()*) // *checkpri()*检测记录的优 |
| //先级是否满足要求，满足要求返回1 |
| 1. Continue 2. **End IF** |
| 1. 接受其他节点法来的累积到 |
| 1. **Foreach** |
| 1. 发送 给(*c,d*) 2. **End Foreach** |
| 1. *V=V+* //累积操作 |
|  |
| 1. **End Foreach** |
| 1. **End While** |
| 1. 将所有*V*值输出 |

图4.2所示的是Maiter上实现的Asyn-SimRank算法的运算过程。Slave1节点和Slave2节点用各自*ΔV*中的值（）计算出要发送的差值并发送。Slave3节点则接收Slave1和Slave2发来的，并将这些数据累积到变量*ΔV*，并适时的用*ΔV*计算出要发送的，发送到其他的Slave节点，然后将*ΔV*变量清零。

给出算法4.2，详细地阐述Asyn-SimRank算法的分布式实现方法。必须指出的是，每个Slave节点都会执行算法4.2，但是每个Slave节点完全是独立的，有数据的时候可以马上进行计算，无需同步，这就达到了异步计算的目的。

## 4.6 本章小结

本文提出Asyn-SimRank算法通过改变SimRank算法的迭代计算方法，解决了传统SimRank算法计算量高、通信量大、计算效率低等的诸多问题。Asyn-SimRank算法为计算图顶点间相似度提供了高效的方法。

Asyn-SimRank算法的提出还进一步说明，迭代-累积方式是一种高效率的迭代方式，它不仅能够减小计算量、通信量，还为算法支持异步计算提供了有利的条件。很多其他算法可以变换成这种形式。所以迭代-累积方式为解决大规模数据处理上的迭代计算问题提供了有效途径。

# 

# 第5章 性能实验与评价

在本章，通过真实的数据集和真实的分布式环境，对本文提出的基于数据块的集中式动态负载均衡机制以及Asyn-SimRank算法的有效性进行了一系列的实验验证。验证了本文提出的负载均衡机制，在集群不存在负不均问题时对集群的影响以及存在负载不均问题时，负载均衡机制的有效性。也验证了本文提出的Asyn-SimRank算法在计算速度、通信量、收敛速度，扩展性方面的优越性，极大的提高了相似度的计算效率。

## 5.1 负载均衡性能验证

为了验证本文提出的基于数据块的集中式动态负载均衡的处理机制的有效性，本文采用了真实的数据集在真实的分布式环境下，进行了一系列的实验。通过实验，分析了再集群不存在负载不均问题时，启用负载均衡机制相比于不启用负载均衡机在通信量和计算效率上会有那些不同。验证了在集群存在负载不均问题时，启用负载均衡机制比不启用负载均衡机制，计算效率更高。也通过实验，对负载均衡机制中的一些关键性的参数对计算效率的影响做了验证。

### 5.1.1 实验数据集及分布式环境

**实验数据集：**采用真实的数据集进行实验验证，能够尽可能接近真实的应用场景，这样进行实验才更有说服力，因此，对于Maiter的负载均衡实验，使用的也是真实的数据集，数据集的出处，可查看参考文献[26]，表5.1为本实验所使用的数据集的相关信息。

表5.1实验数据集相关信息

Table 5.1 Summary of the experimental datasets

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Vertices | Eages | Diameter |
| Journal | 4847571 | 68993773 | 16 |
| Pokec | 1632803 | 30622564 | 11 |
| Google | 875713 | 5105039 | 21 |
| Wiki | 2394385 | 5021410 | 9 |
| Astro | 2473 | 9472 | 10 |
| Gnutella | 6301 | 20777 | 9 |

在表5.1中，数据集的名字为了简便，采用的是名字首字母的缩写，其中Journal代表LiveJournal social network。Pokec代表Pokec social network。Google代表Google web graph。Wiki代表Wikipedia Talk network。Astro代表Astro Physics collaboration network。Gnutella代表Gnutella peer-to-peer network。

**分布式环境：**本实验使用的分布式集群包括4台计算节点，其中一台计算节点既是Master节点也是Worker节点，其余三台只是Worker节点，四台计算节点的软硬件配置信息如表5.2所示。

表5.2 节点配置

Table 5.2 Configuration of computation node

|  |  |
| --- | --- |
| CPU | Intel I5-4690 3.3GHZ 4Core |
| Memory | apacer 4G-DDR3 |
| Disk | hitachi 1TB/7200RPM |
| Netwoker | 1000M Ethernet |
| Opertor | Ubuntu14.04 64Bit |
| Maiter | Release 0.1、0.2 |

### 5.1.2 负载均衡机制的有效性

负载均衡机制的有效性是指在集群出现负载不均衡问题时，启用负载均衡机制与不启用负载均衡机制相比，Maiter集群的计算速度更快。这样就能够说明，由于有负载均衡机制的存在，及时的对负载不均衡问题进行了调整，协调了各个计算节点之间的计算速度，有效的避免了负载不均衡问题对整体计算速度的影响。本实验在采用Maiter集群在两个不同的数据集上运行了PageRank算法，在另外的两个数据集上运行了本文提出的Asyn-SimRank算法。在集群存在负载不均衡问题时，对比了启用负载均衡机制和不启用负载均衡机制时两个算法的运行时间和通信量，实验结果如图5.1和图5.2所示所示。

需要特别说明的是，由于造成Maiter负载不均的问题的原因分为量方面，一方面是数据分配不均匀，一方面是计算节点的计算能力不同。由于本实验所选用的数据集通过哈希划分后，并没有出现分配不均的情况。另外由于本实验使用的计算机集群中的每个计算节点的计算能力也是相同的，因此，在这种实验环境下，很少出现负载不均衡问题，因此，为了验证负载均衡机制的有效性，只能人为的制造出负载均衡问题。由于数据哈希分配是否均匀，只与数据本身有关，很难认为分干预。因此本实验选择人为的干预计算节点的计算能力。主要的方法是在集群运行Maiter框架的同时，在不同的计算节点上运行不同的程序，占用不同量的CPU，这样每个计算节点由于CPU被其他程序占用了一部分，进行Maiter框架的计算的能力就会有所不同，这和各个计算节点的计算能力不同的原理是一样的。这样，就可以在实验中，人为的制造出负载不均衡问题。

图5.1启用与不启用负载均衡处理运行时间对比图

Fig. 5.1 Runing time of algorithms on loading balance start and stop

从5.1图中可以看出，在出现负载不均衡问题时，启用本文提出的负载均衡机制，可以有效的提升Maiter框架的计算效率。结果中也显示，当任务的运行时间越长，负载均衡处理的效果也越明显。在运行时间较短时，也有一定的效果，但是有限。

图5.2启用与不启用负载均衡处理通信量对比图

Fig. 5.2 Total communication of algorithms on loading balance start and stop

从图5.2中可以看出，在启用负载均衡处理之后，集群的整体通信量会减少。从通常意义上讲，在启用负载均衡机制后，要进行数据迁移，并需要上报监测信息，集群的通信量应该增加才对。实际上并不是这样，这与Maiter框架的性质有关。在Maiter集群中，消息一次性到达，计算一次，不管到达的消息有多少，只会产生一次通信。但是由于集群中存在计算较慢的计算节点，就会导致本该一次性到达的消息，可能分成几次才能到达，当这些消息全部到达时，可能已经进行了几次计算，此时通信量已经翻了几倍。因此Maiter集群负载不均不仅会降低收敛速度，还会增加集群的通信量。

### 5.1.3负载均衡机制的开销

由于负载不均衡问题是否出现是不确定的，在运行之前，没有办法预判是否会出现负载不均衡问题，因此在多数情况下，负载均衡机制都应该启用。但是可能在很多情况下，负载不均衡问题并不会出现。因此，在启用负载均衡机制时，如果没有出现负载不均衡问题时，负载均衡机制不应该有太多的计算开销，不应该影响正常的计算。

为了验证启用负载均衡机制时，对Maiter框架的影响，本实验在不出现负载不均衡问题时，分别对比了启用负载均衡机制和不启用负载均衡机制时的计算效率和系统的通信量，实验结果如图5.3和5.4所示。实验结果显示，在不存在负载不均衡问题时，启用负载均衡机制并没有显著的增加Maiter框架的通信量，也没有影响Maiter框架的计算性能。由于当集群不存在负载不均衡问题时，各个计算节点的balance进程不会创建，不会增加Worker节点的计算开销。同时监测信息量与集群的通信量不在一个数量级，对集群的通信总量的影响微乎其微。

图5.3启用与不启用负载均衡处理运行时间对比图

Fig. 5.3 Runing time of algorithms on loading balance start and stop

图5.4启用与不启用负载均衡处理通信量对比图

Fig. 5.4 Total communication of algorithms on loading balance start and stop

### 5.1.4 数据分块数量对系统的影响

在实现负载均衡处理机制的过程中，本文改变的Maiter框架的数据管理方式，将每个计算节点管理一个数据块改为管理多个数据块。因此，必须通过实验来验证这种更改对集群本身的影响。在本实验中，本文将一个相同的数据集切分成不同的数据块个数，并在Maiter计算上运行。实验结果如图5.5所示。

图5.5数据分块个数对运行时间的影响

Fig. 5.5 The impact on running time of data blocks number

从图5.5可以看出，随着数据块数量的增大，系统的运行时间会首先呈现出下降的趋势，随后平稳。由于集群中只有4台计算节点，因此当数据块个数为4的时候，与系统没更改之前的数据管理方式是一样的。因此，合理的将数据分块，可以一定程度上加快Maiter框架的计算效率。

## 5.2 Asyn-SimRank性能验证

为了验证本文提出的Asyn-SimRank算法的在通信量、收敛速度、整体计算速度等方面的优越性，本实验选取不同的实际应用系统产生的数据集作为输入数据，分别在Hadoop、Spark框架上实现SimRank算法、delta-SimRank算法，在异步分布式框架Maiter上实现Asyn-SimRank算法，比较三者运行过程中的通信量、收敛速度、以及总体的运行时间。

在本文的对比图中，SR，D-SR分别代表SimRank算法和Delta-SimRank算法， A-SR-UP代表轮询调度计算的Asyn-SimRank算法，A-SR-P代表关键点优先调度计算的Asyn-SimRank算法。Asyn-SimRank算法均在Maiter框架上实现。

### 5.2.1 实验数据集及实验环境

**实验数据集：**本实验所选用的所有数据均出自斯坦福大学的大规模网络数据集[[1]](#footnote-0)（Stanford Large Network Dataset Collection）。该数据集中的所有数据均是实际应用中产生的数据。选取实际的数据集运行算法可以有力的说明算法在实际应用中运行的真实情况。实验中选择的五个数据集包括：天文物理论文合作网络ca-AstroPh、相对论及量子宇宙论文合作网络ca-GrQc、高能物理论文合作网络ca-HepTh、凝态物理论文合作网路ca-CondMat、文件共享网络P2P。数据集的相关信息如表5.3所示。

表5.3 实验数据集相关信息

Table 5.3 Summary of the experimental datasets

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Vertices | Eages | Diameter | The node-pairs |
| ca-AstroPh | 2403 | 9852 | 10 | 2888406 |
| ca-GrQc | 5242 | 28980 | 17 | 13741903 |
| ca-HepTh | 9877 | 51971 | 17 | 48782503 |
| ca-CondMat | 15014 | 61557 | 14 | 112717605 |
| P2P | 22687 | 54705 | 10 | 257361328 |

**实验环境**：本实验中所用到的分布式环境包括33台计算机。大部分实验如运行时间、收敛速度、通信量实验采用17台计算机，一个Master节点， 16个Slave节点。在分布式环境规模与运行时间的关系实验中，最多用到了33台计算机。分布式环境中计算机的软件硬件配置如表5.4所示。

表5.4 节点配置

Table 5.4 Configuration of computation node

|  |  |
| --- | --- |
| CPU | INTEL CORE i3-2100 LGA-1155 4 slots |
| Memory | apacer 4G-DDR3 \*2 |
| Disk | hitachi 500G/7200RPM |
| Natwoker | 1000M Ethernet |
| Opertor | Redhat 6.1 64Bit |
| Hadoop | Release 1.0.4 |
| Maiter | Release 0.1 |

### 5.2.2 总体运行时间对比

本实验在不同的同步框架上实现了同步算法SimRank和Delta-DimRank，与异步框架Maiter上实现的Asyn-SimRank算法在运行时间上进行了比较。

**Hadoop：**图5.6所示的是三个算法在不同的数据集上运行时间的对比。可以看出Asyn-SimRank算法总体计算时间明显缩短。Asyn-SimRank算法最快能比Hadoop上实现的SimRank算法快75.3倍，比Hadoop上实现的Delta-SimRank算法最快可快64.7倍，运行速度提升明显，说明了Asyn-SimRank算法的在计算速度方面的优势。

图5.6Maiter和Hadoop实现的算法的运行时间对比

Fig. 5.6 Run time of algorithms implemented on Maiter and Hadoop

**Spark:**由于目前很多Hadoop的改进框架如Spark等可以高效的支持迭代运算，在这些改进框架上实现SimRank和Delta-SimRank算法，也可以有效提升运算效率，为了有力的说明Asyn-SimRank算法的在运行速度上的优势，本实验将Maiter上实现的Asyn-SimRank算法与Spark上实现的SimRank算法和Delta-SimRank算法进行了性能对比，实验中使用的数据集为ca-AstroPh，实验结果如图5.7所示。可见，Asyn-SimRank算法在运行速度上仍具有明显的优势。

图5.7Maiter和Spark实现的算法的运行时间对比

Fig. 5.7 Run time of algorithms implemented on Maiter and Spark

以上实验说明Maiter框架上实现的Asyn-SimRank算法比主流同步类型的框架Hadoop、Spark上实现的SimRank算法和Delta-SimRank算法的计算效率都要高。但是不同算法在不同的框架上实现不能屏蔽框架的差异对实验结果的影响。然而Maiter框架是异步类型的框架，同步类型的算法SimRank算法、Delta-SimRank算法不能在Maiter上实现。

图5.8Maiter和Maiter-syn实现的算法的运行时间对比

Fig. 5.8 Run time of algorithms implemented on Maiter and Maiter-syn

**Maiter-syn:**为了尽可能的屏蔽框架差异对实验结果的影响，本文在Maiter框架的基础之上，人为的增加了同步路障，以保证同步算法的正确性，实现了Maiter-syn**[[2]](#footnote-1)**同步框架，可以实现Delta-SimRank算法。本实验在三个不同的数据集上对Maiter-syn上实现的Delta-SimRank算法和Maiter上实现的Asyn-SimRank算法进行了性能对比，实验结果如图5.8所示。Asyn-SimRank算法运算速度较快。

### 5.2.3 收敛速度对比

算法收敛的速度是评价算法好坏的重要指标。为了能够公平的衡量各个算法的收敛速度并剔除系统的影响，本实验通过监测相似值的总量增长的速度随顶点更新总次数的变化来衡量算法的收敛速度。所谓顶点更新总次数是指从算法开始到某一时刻，所有顶点的相似度值被更新的次数之和。相似度值总量是指所有顶点的相似度之和。因为在SimRank算法的计算过程中，可以证明相似度之和是递增的。那么随着更新次数的增加，相似度值的总量也会增加，并且增加的越快说明算法收敛越快。

图5.9 ca-AstroPh数据集上收敛速度对比

Fig. 5.9 Comparison of convergence speed on ca-AstroPh

图5.9与图5.10所示的曲线为三个算法的收敛速的对比。从图中可以看出，没有采用关键点优先调度计算的Asyn-SimRank算法的收敛速度比SimRank好，但是并不比Delta-SimRank好，主要原因是因

为在没有采用关键点优先调度计算的情况下，Asyn-SimRank算法会和Delta-SimRank算法一样，会对所有迭代差值非零的顶点更新，但是Asyn-SimRank是异步更新，累积差值的大小可以是任意的，所以累积的差值要比Delta-SimRank算法小，达到相同的相似度总量时更新次数比较大。虽然如此，在运行速度上没有使用关键点优先调度计算的Asyn-SimRank算法还是远远快于Delta-SimRank，主要得益于异步计算。

图5.10 P2P数据集上收敛速度对比

Fig. 5.10 Comparison of convergence speed on P2P

关键点优先调度计算的Asyn-SimRank算法的收敛速度远远比SimRank和Delta-SimRank算法快。在关键点优先调度计算过程中，Asyn-SimRank将更新全部作用于那些对算法全局收敛影响明显的顶点对上，让影响尽快地传播出去。并且让重要的或收敛慢的顶点先进行运算，有效的提升了算法收敛的速度。

### 5.2.4 通信量对比

图 5.11 通信量对比

Fig. 5.11 Comparison of communication cost

图5.11所示的是各算法在不同的数据集上通信量的对比。从图中可以看出，Asyn-SimRank算法的通信量相与其他的算法相比明显降低。Asyn-SimRank算法和SimRank算法的通信量最大差距可达到51倍，和Delta-SimRank算法的通信量差距最大可达15倍。Asyn-SimRank算法的通信量小的原因主要是因为，通过收敛顶点和相似度值暂时不变顶点的迭代差值为零的原理，节省了这些顶点的计算。由于SimRank算法本身的性质，总是大部分顶点对提前收敛，少部分的值后收敛，因此计算量的节省和通信量的节省是十分可观的。

### 5.2.5 算法与分布式环境规模的关系

为了充分的验证Asyn-SimRank算法能否适应大规模的分布式环境，处理大规模的数据，本实验对Asyn-SimRank算法的运行时间与分布式环境规模的关系进行了实验分析。在本次实验中，分别在16台、24台、32台计算节点上使用相同的数据集(ca-CondMat，1.1亿实际计算顶点)运行了Asyn-SimRank算法，并监测算法运行的时间，实验结果如图5.12所示。

图5.12 Asyn-SimRank算法的扩展性

Fig. 5.12 Scaling performance of Asyn-SimRank algorithm

## 5.3 本章小结

本章在真实的分布式环境中，使用真实的数据集运行了不同的应用，详细的探究了基于数据块的集中式动态负载均衡机制的有效性、系统开销以及各种配置参数对系统的影响。详细的探究了Asyn-SimRank在计算效率、通信量、扩展性方面的表现。实验有力的说明了本文提出的负载均衡处理机制是有效的。有力的说明了Asyn-SimRank速算在计算效率上的巨大提升。

# 

# 第6章 总结与展望

## 6.1总结

本文首先介绍了本次课题的研究背景与研究意义，接着详细的介绍了Maiter框架的原理与实现。Maiter框架是本文的研究基础，本文的主要研究内容也是围绕着Maiter框架展开。然后本文有详细的介绍了本文提出的局域数据块的集中式动态负载均衡集中的原理和实现以及Asyn-SimRank算法的推导和正确性和收敛性的相关证明。

本文提出的基于数据块的集中式动态负载均衡处理集中，能够在计算的过程中完成负载不均衡问题的处理，并保证算法的正确性，文中对于一些方法的正确性也给出了相关的理论证明。实验结果显示，在不存在负载不均衡问题时，负载均衡集中几乎不会影响Maiter框架的计算性能和通信量。在存在负载不均衡问题是，负载均衡机制能够有效的进行相应的处理，有效的提升Maiter框架的计算效率。

本文提出的Asyn-SimRank算法，采用了累积迭代的的思想，实现了异步计算相似度，节省了大量同步开销的同时，通过避免收敛顶点的计算，有效的降低了计算量和通信量。同时本文还提出了关键点优先计算策略，有效的提升了算法的收敛速度。本文也在理论上证明了Asyn-SimRank算法的正确性和收敛性，以及关键点优先策略的有效性。实验结果显示，Asyn-SimRank算法在计算效率、收敛速度、通信量上都较传统SimRank算法有了很大的提升。

## 6.2展望

本文所做的一系列工作，虽然在一定程度上取得了所期望的效果，但是有些问题在诸多方面有待完善，有些问题仍可以继续探究。

本文在Maiter框架上实现了基于数据块的集中式动态负载均衡处理机制。在实现的过程中，将代码从集中式转移到分布式，从运行小的实验数据到运行大规模数据，都出现了很多问题，虽然这些问题最终都得以解决，但是系统的健壮性仍然有待提高。主要表现在当配置参数配置不合理时，就会导致系统处于不稳定的状态，容易出错。对不稳定状态没有太好的容忍性。

在DAIC计算模型的应用研究方面，目前还有很多的其他算法都有可能使用

DAIC计算模型的思想进行改进。但是究竟能够实现，还需要更深入的探究。由于本文精力有限，未能完成。

# 参考文献

1. Bertsekas. Parallel and Distributed Computation[M], USA:Prentice Hall, 1997, 30-56.
2. [EB/OL]. http://zh.wikipedia.org/wiki/PageRank, 2014-05-01.
3. Shvachko K, Kuang H, Sanjay R, et al. The hadoop distributed file system[C]//Proc of the 2010 IEEE 26th Symp on Mass Storage Systems and Technologies. New York: ACM, 2010:1-10.
4. [EB/OL]. Hadoop, http://hadoop.apache.org, 2015-03-02.
5. 蔡斌, 陈湘萍. Hadoop技术内幕[M], 北京:机械工业出版社, 2013, 32-50.
6. 周敏琪, 王晓玲, 金澈清, 等. Hadoop权威指南[M], 北京:清华大学出版社, 2012, 115-130.
7. 文艾, 王磊. HDFS: Hadoop分布式文件系统深度实践[M], 北京:清华大学出版社, 2012, 58-70.
8. Zaharia M, Chowdhury M, Franklin M. Spark: Cluster computing with working sets[C]//Proc of the 2nd USENIX Conf on Hot Topics in Cloud Computing. Berlin: Springer, 2010:10-10.
9. [EB/OL]. Graphlab, http://docs.graphlab.org/index.html, 2014-05-07.
10. [EB/OL]. Maiter Project, http://code.google.com/p/maiter, 2013-07-15.
11. Zhang. Y, Gao. Q. Accelerate large-scale iterative computation through asynchronous accumulative updates[C]//Proc of the 3rd Workshop on Scientific Cloud Computing. New York: ACM, 2012:13-22.
12. Jeh. G, Widom. J. Simrank: A measure of structural-sontext similarity[C]//Proc of the 8th ACM SIGKDD Int Conf on Knowledge Discovery and Data Mining. New York: ACM, 2002:538-543.
13. Crick. C, Pfeer. A. Loopy belief propagation as a basis for communication in sensor networks[C]//Proc of the 19th Conf on Uncertainty in Artificial Intelligence. New York: ACM, 2003:159-166.
14. Ihler. J. Willsky A. Loopy belief propagation: Convergence and effects of message errors[J], Machine Learning Research, 2005, 5(6):905-936.
15. Elidan. G, McGraw. I, Koller. D, et al. Residual belief propagation: Informed scheduling for asynchronous message passing[C]//Proc of The AI&M Procedure for Learning from Incomplete Data. NJ: IEEE, 2006:165-173.
16. Gonzalez. J. E, Low. Y, Guestrin. C, et al. Residual splash for optimally parallelizing belief propagation[C]//Proc of the Twelfth International Conference on Artificial Intelligence and Statistics. New York: ACM, 2009:177-184.
17. Gonzalez. J. E, Low. Y, Guestrin. C, et al. Distributed parallel inference on large factor graphs[C]//Proc of The 25th Conference on Uncertainty in Artificial Intelligence. NJ: IEEE, 2009:203-212.
18. Dean. J, Ghemawat. S. MapReduce: simplified data processing on large clusters[J], Communications of the ACM, 2004, 5(1):107-113.
19. Malewicz. G, Austern. M. H, Bik. A. J, et al. Pregel: a system for large-scale graph processing[C]//Proc of the 2010 ACM SIGMOD Inter Conf on Management of data. New York: ACM, 2010:135-146.
20. Pavol. Q. G, Boting, H. Finding Robust Minimum Cuts[C]//Proc of 10th Int Conf Algorithmic Aspects in Information and Management. Berlin: Springer, 2014:122-136.
21. Bu Y, Howe B, Balazinska M, et al. Haloop: Efficient iterative data processing on large clusters[J], VLDB Endowment, 2010, 3(1/2):285-296.
22. Zhang. Y, Gao. Q, Gao. L, et al. iMapReduce: a distributed computing framework for iterative computation[C]//Proc of the 2nd ACM Symposium on Cloud Computing. New York: ACM, 2011:1112-1121.
23. Zhang. Y, Gao. Q, Gao. L et al. PrIter: a distributed framework for prioritized iterative computations[C]//Proc of the 2nd ACM Symposium on Cloud Computing. New York: ACM, 2012:13-25.
24. Cao L, Kim. H. D, Tsai. M, et al. Delta-SimRank computing on mapreduce[C]//Proc of the 1st International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms Systems Programming Models and Applications. New York: ACM, 2012:28-35.
25. Zhang. Y, Li. C. Fast simrank computation over disk-resident graphs[C]//Proc Of 18th International Conference on Database Systems for Advanced Applications. Berlin: Springer, 2013:16-30.
26. [EB/OL]. http://snap.stanford.edu/data/, 2015-01-12.
27. H. Balakrishnan, M. F. Kaashoek, D. Karger. Looking Up Data in P2P Systems[J], Communications of the ACM, 2003, 46(2):43-48.
28. Chen. R, Yang. M, Weng. X, et al. Improving Large Graph Processing on Partitioned Graphs in the Cloud[C]//Proc of the 3rd ACM Symposium on Cloud Computing. NJ: IEEE, 2012:3:1-3:13.
29. Gonzalez. J. E, Low. Y, Gu. H, et al. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs[C]//Proc of the 10th USENIX Symposium on Operating Systems Design and Implementation. NJ: IEEE, 2012:17-30.
30. Ho. L, Wu. J, Liu. P. Distributed Graph Database for Large-Scale Social Computing[C]//Proc of the IEEE 5th International Conference in Cloud Computing. NJ: IEEE, 2012:455-462.
31. Kang. U, Tsourakakis. C, Appel. A. P, et al. HADI: Fast Diameter Estimation and Mining in Massive Graphs with Hadoop[J], ACM Trasactions on Knowledge Discovery from Data, 2011, 5(2):8:1-8:24.
32. Karypis .G, Kumar. V. Multilevel k-way Partitioning Scheme for Irregular Graphs[J], Parallel and Distributed Computing, 1998, 48(1):96-129.
33. Low Y, Bickson D, Gonzalez J, et al. Distributed GraphLab: a framework for machine learning and data mining in the cloud[J], VLDB Endowment, 2012, 5(8): 716-727.
34. Kolb L, Thor A, Rahm E. Load balancing for mapreduce-based entity resolution[C]//Proc of 28th Inter Conf on Data Engineering. NJ: IEEE, 2012: 618-629.
35. Lee R, Jeng B. Load-balancing tactics in cloud[C]//Proc of International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery. NJ: IEEE, 2012:447-454.
36. Antonellis. I, Molina. H, Chang. C. Simrank++: query rewriting through link analysis of the click graph[J], Machine Learning Research, 2008, 1(1):408–421.
37. F. Fouss, A. Pirotte, J. Renders, M. Saerens. Random walk computation of similarities between nodes of a graph with application to collaborative recommendation[J], IEEE TKDE, 2007, 19(3):355–369.
38. Jeh. G Widom. J. Simrank: A measure of structural-context similarity[C]//Proc of the Eighth ACM SIGKDD Inter Conf on Knowledge Discovery and Data Mining. New York: ACM, 2002:538-543.
39. P. Li, Y. Cai, H. Liu, J. He, X. Du. Exploiting the block structure of link graph for efficient similarity computation[C]//Proc of Pacific-Asia Conf on Knowledge Discovery and Data Mining. Berlin: Springer, 2009:389-400.
40. Maguitman. A. G. Algorithmic computation and approxi-mation of semantic similarity[C]//Proc of World Wide Web. New York: ACM, 2006:431-456.
41. Blondel. V. D. A Measure of Similarity between Graph Vertices: Applications to Synonym Extraction and Web Searching[J], SIAM REVIEW, 2004, 46(4):647-666.

# 致 谢

本论文工作历时一年左右的时间，在此期间遇到了很多的问题和困难，经历诸多波折，但是总是能够一一解决，并圆满的完成。然而光靠一己之力解决诸多的困难是不可能的，离不开老师的谆谆教导和同学们的帮助。

感谢于戈老师。能够做于戈老师的研究生，给了我极大的鼓励，总是希望能够倍加努力，不辜负于老师的期望。非常感谢于老师给予了优质的学习环境和研究环境，并在百忙之中，对论文的选题、方向、写作、答辩方面的给予指导。

感谢张岩峰老师。非常高兴能够参加到张老师的大数据项目组，在这里。我不仅学会了很多知识，同时本文的主要工作也是围绕着张老师的项目展开。张老师的项目研究内容新颖，有深度，解决实际问题，具有很高的实用价值。能够围绕着张老师的项目展开工作，也使得本文的工作有了更深的实际意义。非常感谢张老师在我研究生期间的教导、鼓励、支持。

感谢项目组的所有老师和同学，感谢焦明海老师给予的宝贵指导意见，也感谢项目组的同学们，没有大家的帮助支持，本文无法顺利完成。

# 攻读硕士学位期间的论文和项目情况

论文

(1)王春磊, 张岩峰, 鲍玉斌, 赵长宽, 于戈, 高立新. Asyn-SimRank: 一种可异步执行的大规模SimRank算法[C]第一届CCF大数据会议. 北京, 2013

(2)王春磊, 张岩峰, 鲍玉斌, 赵长宽, 于戈, 高立新. Asyn-SimRank: 一种可异步执行的大规模SimRank算法.[J], 计算机研究与发展. 2015，52(7)

参与项目

2013.03-2015.07国家自然科学基金项目：云环境下基于BSP模型的大规模不动点迭代计算研究（61300023）

1. http://snap.stanford.edu/data/index.html [↑](#footnote-ref-0)
2. http://code.google.com/Maiter/ [↑](#footnote-ref-1)