

国防科学技术大学

硕士学位论文

MapReduce集群多用户作业调度方法的研究与实现

姓名：王凯

申请学位级别：硕士

专业：计算机科学与技术

指导教师：吴泉源；杨树强

2010-11

摘要

当前的数据密集型计算需要处理 PB 级数据集和 GB 级数据流，面临着大规模数据管理、复杂计算环境管理、可扩展计算平台等方面的难题。Hadoop 是一种易扩展的分布式计算架构，能将廉价 PC 节点联合起来提供计算服务，其 MapReduce 框架为用户提供了容易使用的并行处理大规模数据的编程模式。

本文在分析了现有的 Hadoop 集群作业调度方法的基础上，对现有的 MapReduce 集群的多用户作业调度方法的数据本地性较差的问题进行了深入研究。并针对 Hadoop 现有调度算法不能很好的保障任务的数据本地性问题，提出了一种基于时间的等待调度方法，该方法优先将任务调度到其所需的数据所在的节点上执行，从而实现了更好的数据本地性，有效减少计算过程中的 IO 开销，实现提高系统吞吐率和减少单个作业平均响应时间的目的。

为验证方法的有效性，我们对提出的作业调度方法给出了设计与实现，并进行了实验验证。结果表明，基于时间的等待调度方法在保证多用户公平共享集群的基础上，节点的数据本地性得到很大的提高，有效增加了集群系统的吞吐量，有效减少了单个作业的平均响应时间。

主题词：分布式计算，MapReduce，Hadoop，作业调度，等待调度，多用户共享

ABSTRACT

The current data intensity computation needs to process the PB level data set and the GB level data stream, facing the large-scale data management, the complex computation environmental management, scalable computing platform problems. Hadoop is a kind of scalable distributed computing architecture which can combine a lot of inexpensive PCs to provide super computing, It's Map-Reduce parallel computing framework prepare an easy programming model for users.

This paper in-depth analysis of the existing Hadoop cluster's job scheduling approach, then we in-depth research the problem of poor data locality that caused by the existing methods of multi-user job scheduling. For the existing scheduling algorithms of Hadoop can not get good data locality, we achieve a waiting time-based scheduling method, which give priority to scheduling task to node where the required data been stored, so can achieve better data locality, effectively reduce the IO overhead in calculation process, to achieve purposes of increasing system throughput and reducing the average response time of a single work.

To verify the validity of the method, we give the design and implementation for our proposed scheduling method and verified by experiments. The results show that the method not only guarantees multi-user's fair share cluster, and the data locality of the node has been greatly improved, increase the throughput of the cluster system effectively, effectively reducing the average response time of a single job.

Key Words: Distributed Computing, MapReduce, Hadoop, Job Scheduling, Waiting Scheduling, Priority, Multi-user Shared

表 目 录

表 2.1 Hadoop 中 MapReduce 作业的组成	17
表 4.1 公平调度方法下作业任务运行图	34
表 4.2 设置等待时间的作业调度顺序实例图	36
表 6.1 小作业负载的数据本地性	55
表 6.2 混合负载中的各个级别的作业大小和类型	58

图 目 录

图 1.1 公用云服务示意图	2
图 1.2 私有云服务示意图	2
图 1.3 混合云提供服务示意图	3
图 1.4 开源软件创建设备示图	5
图 1.5 云计算集群逻辑结构图	5
图 2.1 MapReduce 集群节点结构图	13
图 2.2 MapReduce 模型执行示意图	14
图 2.3 Hadoop 中 MapReduce 作业的组成	18
图 2.4 Hadoop 的 MapReduce 执行流程	19
图 3.1 槽分配实例	28
图 3.2 数据本地性和作业大小关系图	29
图 3.3 不同文件副本和节点槽数下槽粘滞影响图	30
图 4.1 公平调度方法槽分配示例图	35
图 4.2 基于等待时间的槽分配示例图	37
图 5.1 基于时间的等待调度主程序执行流程	42
图 5.2 map 任务执行流程	43
图 5.3 map 任务调度执行图	46
图 5.4 Hadoop 集群结构图	47
图 5.5 任务并行结构图	49
图 5.6 MapReduce 子任务间并行	49
图 5.7 MapReduce 中 map 和 reduce 任务间并行	50
图 5.8 MapReduce 中作业间并行	50
图 6.1 执行时间比较图	55
图 6.2 槽粘滞实验中负载完成时间	56
图 6.3 槽粘滞实验中节点数据本地性	56
图 6.4 不同规模作业的数据本地性图	57
图 6.5 等待调度相对与公平调度的平均加速比，黑线为标准差	57
图 6.6 混合负载中不同规模作业响应时间的累积分布函数图	58

独 创 性 声 明

本人声明所呈交的学位论文是我本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，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经发表和撰写过的研究成果，也不包含为获得国防科学技术大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文题目： MapReduce 集群多用户作业调度方法的研究与实现

学位论文作者签名： 王凯 日期： 2010 年 11 月 10 日

学 位 论 文 版 权 使用 授 权 书

本人完全了解国防科学技术大学有关保留、使用学位论文的规定。本人授权国防科学技术大学可以保留并向国家有关部门或机构送交论文的复印件和电子文档，允许论文被查阅和借阅；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，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

(保密学位论文在解密后适用本授权书。)

学位论文题目： MapReduce 集群多用户作业调度方法的研究与实现

学位论文作者签名： 王凯 日期： 2010 年 11 月 10 日

作者指导教师签名： 吴军源 日期： 2010 年 11 月 10 日

第一章 绪论

1.1 课题研究背景

随着互联网技术的发展和人们对计算能力需求的增加，集群作为一种廉价的可以提供强大计算能力的并行计算技术得到越来越广泛的应用，其具有大型机的超级计算能力和较低成本投入^[1]。从而成为各种高性能计算的主流方向，如科学计算与其他需要大规模并行计算的应用服务等。

近年来，由于在各个领域中大规模、海量数据存储和处理需求的不断增加，传统企业自身 IT 结构的计算能力已远远不能满足计算的需要，必须通过大规模的硬件投入以扩展计算能力。同时，由于传统的并行编程模型的局限性，也对新的并行编程框架提出了更高的要求。正是在这样的节省成本和实现系统可扩展性需求的催化下，云计算^[2]、云存储及 MapReduce 的概念应运而生，且被认为是解决目前大规模数据存储和处理的较好的方案。

1.1.1 课题背景

1.1.1.1 云计算简介

云计算将计算任务分布到由大量计算实体组成的资源池上，各种应用请求可以按需获取相应的计算能力、存储空间和软件服务^[2]，并采取计时付费的支付方式。

云计算有狭义和广义之分。狭义云计算中，提供资源的网络被称为“云”，“云”中的资源在使用者看来是可以无限扩展的，并且可以随时获取和扩展，按需使用，以及按适用付费。广义云计算则是指服务的交付和使用模式，指通过网络以按需、易扩展的方式获得所需的服务。这种服务可以是与软件和互联网相关的，也可以是任意其他的服务^[3]。

采用云计算的计算模型可以带来前所未有的价值。云计算具有更高的安全性，提供了可靠、安全的数据存储中心，数据丢失和病毒入侵等麻烦减少；使用方便，对用户端设备要求较低；实现数据共享，可以轻松实现不同设备间的数据与应用共享；为中小企业节省了大量的成本，通过把数据资源存放在云中，由云计算提供商管理这些数据，节省了企业的硬件、管理成本；云计算具有更大的灵活性和可伸缩性，通过虚拟化的支持可以轻松地扩展基础设施，实现计算资源的可伸缩^{[3][49]}。

云计算是并行计算(Parallel Computing)、分布式计算(Distributed Computing)和网格计算(Grid Computing)的发展，或者说是这些计算机科学概念的商业实现。云

计算是虚拟化(Virtualization)、效用计算(Utility Computing)、IaaS(基础设施即服务)、PaaS(平台即服务)、SaaS(软件即服务)等概念混合演进并跃升的结果^{[1][4]}。

云计算在现阶段有三种基本的服务模式，公用云、专用云（或称私有云）、混合云^{[3][5]}。下面分别对它们进行介绍。

公用云由第三方运行，而不同客户提供的应用程序可能会在云的服务器、存储系统和网络上混合在一起（如图 1.1 所示）。公用云通过提供一种企业基础设施进行的灵活甚至临时的扩展，提供一种降低客户风险和成本的方法。

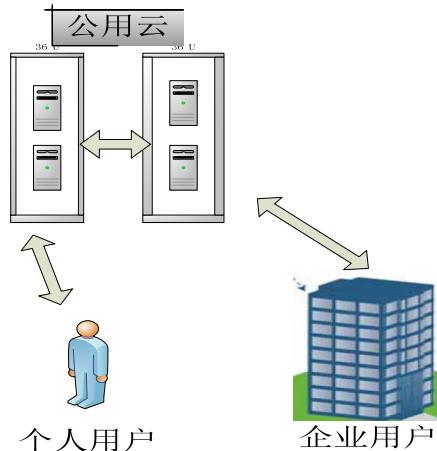


图 1.1 公用云服务示意图

私有云是为一个客户单独使用而构建的，因而提供对数据、安全性和服务质量的最有效控制（如图 1.2 所示）。

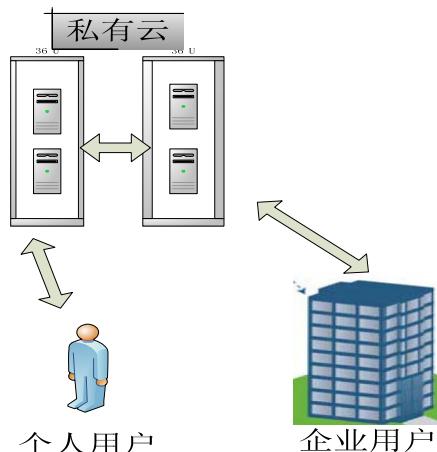


图 1.2 私有云服务示意图

混合云把公用云模式与专用云模式结合在一起（图 1.3）。混合云有助于提供按需的、外部供应的扩展。用公用云的资源扩充专用云的能力可用来在发生工作负载快速波动时维持服务水平。混合云也可用来处理预期的工作负荷高峰。对于数据量小，或应用程序无状态的情况，与必须把大量数据传输到一个公用云中进行

的处理相比，混合云要成功得多。

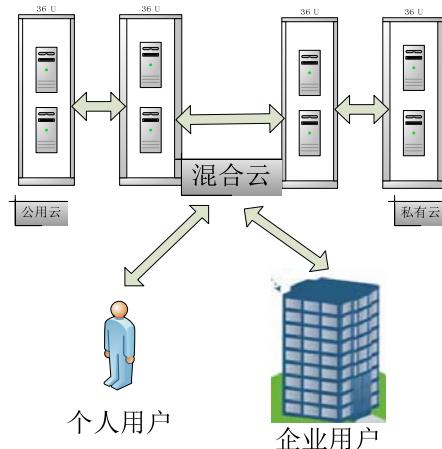


图 1.3 混合云提供服务示意图

上文提到云计算的三种服务设施类型，其所提供的云服务按照现阶段的定义可以分为以下三种：软件即服务（Software as a Service），平台即服务（Platform as a Service），基础设施即服务（Infrastructure as a Service）。云计算作为一种新兴的计算方式^[6]，其性质可归纳如下：

1) 扩大了现有技术的作用

云计算推动降低服务提供成本的已有趋势，同时提高部署服务的速度和敏捷性。它缩短了从设计应用程序架构到实际部署应用程序的时间。云计算把虚拟化、按需部署、网上服务提供和开放源软件融合在一起。

2) 部署对象为虚拟机^[7]

虚拟机已成为一种标准部署对象，虚拟化进一步增强了灵活性，因为它把硬件概括到这样一个高度：在硬件上面，可以在不需要连接具体物理服务器的情况下部署和重新部署软件栈。虚拟机有助于满足快速部署和扩展应用程序的需要。计算云通常由存储云进行补充，存储云通过 API 提供虚拟化存储，而这些 API 为存储虚拟机映像 (Image)、用于诸如 Web 服务器的组件的源文件、应用程序状态数据以及一般业务数据，提供便利。

3) 按需、自助、按使用情况付费

云计算的按需、自助和以使用情况付费的性质也是已有趋势的一种延伸。从企业的观点看，云计算的按需性质有助于支持服务水平目标的性能和容量方面。云计算的自助性质使机构可以创造根据工作负荷和目标性能参数进行扩展和收缩的弹性环境。而且云计算的按使用情况付费的性质可以采取设备租赁的形式，设备租赁保证了云提供商提供一种最低的服务水平。构建计算云时就好像应用程序是临时的，而计费是按照资源消耗情况进行的：使用的 CPU 小时数、移动的数据量或存储的数据的千兆字节 (GB) 数^[8]。

4) 基础设施的可编程

过去，架构设计师确定一个应用程序的各种组件如何在一组服务器上进行布局，即如何连接、固定、管理和扩展这些组件。现在，开发人员可以使用云提供商的 API 不仅在虚拟机上创建应用程序的初始结构，而且还确定该应用程序如何扩展和演进以适应工作负载的变化。

5) 应用程序是可组合的

这种自助式、按使用情况付费的模式的另一个后果是，就像编写应用程序一样，通过汇编和配置设备和开放源软件来组合应用程序。可以重构 (Refactor) 以最大限度地利用标准组件的应用程序和架构，是那些将会在利用云计算效益方面最为成功的应用程序和架构。构建大型完整应用程序已成为过去，因为可直接使用或根据特定用途定制的现有工具库已经变得越来越大。

例如，像 Hadoop (一种开放源 MapReduce 架构实现) 这样的工具可以在多种情况下使用，其中可以对一个问题及其数据进行重构，以便于其多个部分可以同时执行。在后面的章节将做详细介绍。当《纽约时报》想将其档案中 1100 万份文章和映像转换成为 PDF 式时，其内部 IT 机构说这会需要七个星期时间。同时，使用 100 个运行 Hadoop 的 Amazon EC2 简单 Web 服务接口实例的一名开发人员，用 24 小时时间就完成了这项工作，劳动成本只有 300 美元。(这不包括上传数据所需的时间或存储成本。)

6) 通过网络提供服务

不言而喻，云计算扩大了通过网络提供服务的已有趋势。基于互联网的服务提供的美妙之处就在于可以随时随地使用应用程序。如果架构设计的合理，互联网服务提供模式可提供各种规模的企业所需的灵活性和安全性^[9]。

7) 使用开放源软件

开放源软件在云计算中发挥着一种重要的作用，因为开放源软件允许从容易访问的组件创建其基本软件元素：虚拟机映像和设备^[10]。这会产生巨大的影响：

- 例如，开发人员可以通过将 MySQL 软件层叠在一个 OpenSolaris™ 操作系统上并执行自定义来创建一个数据库设备 (图 1.4)。可以通过把开放源软件层叠在一个虚拟机映像之中，并执行简化其部署的自定义，来创建设备。在此示例中，通过把 MySQL 软件层叠在 OpenSolaris 操作系统上来创建一个数据库设备。像这样的设备能够根据需要创建、部署和动态扩展云计算应用程序。例如，看看开放源软件如何使 Animoto 创建的应用程序在几天之内就扩展到 3500 个实例。



图 1.4 开源软件创建设备示图

用开放源组件汇编大型应用程序非常容易，因而生成更多开放源组件。这反过来又使开放源软件的作用更加重要。例如，需要拥有一种可在云计算环境中运行的 MapReduce 算法，这就是刺激开发该算法的因素之一。MapReduce 计算模型也正是借助云计算才更好的实现了其分布式并行计算的功能。大大的提高了集群的计算能力。进一步提高开发人员编写云计算应用程序的水平。

1.1.1.2 云计算集群及云存储

云计算的基本原理是将计算分布到大量的分布式计算机上，而非本地计算机或远程服务器上，其运行方式与互联网相似。不同的厂商提出了不同的云计算的定义和实施结构，但其一般的逻辑结构可用下图 1.5 表示^[11]：

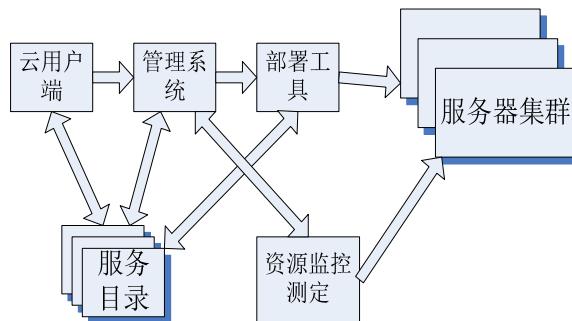


图 1.5 云计算集群逻辑结构图

图 1.5 为云计算集群的逻辑结构图，在具体的实现时各个厂家给出了不同的技术解决方案和实现方法。具体物理实现和部署时有两级结构和三级结构等方式，如作者曾经参与分析研究的 Eucalyptus 系统^{[12][11]}，采用三级体系结构，云控制器 CLC 负责对底层的资源如服务器、存储、网络进行虚拟化。集群控制器 CC 由每个在云中定义的集群形成前端，每个 CC 控制若干台 NC。节点控制器 NCs 是运行虚拟机实例的机器。负责具体任务的完成。Hadoop 系统则从用两级的主从结构 Master/Slave 结构^[10]。相对于 HDFS（Hadoop Distributed File System），Master 为 NameNode 负责维护集群内的元数据，对外提供创建、打开、删除和重命名文件

或目录的功能。Slave 节点为 DataNode 负责数据存储，并负责处理数据的读写请求。DataNode 定期向 NameNode 上报心跳，NameNode 通过响应心跳来控制 DataNode。相对于 MapReduce，Master 叫做 JobTracker，而 Slave 叫做 TaskTracker。将在下一章做详细介绍和分析。

云存储（cloud storage）在云计算概念上延伸和发展出来的一个新的概念，是指通过集群应用、分布式文件系统等，将网络中大量各种不同类型的存储设备通过应用软件集合起来协同工作，共同对外提供数据存储和业务访问功能的一个系统。这种利用“云”系统或互联网来进行的云存储，能够减少个人和企业用户购买硬件的成本，很好的满足了不断增长的存储需求，使之更容易进行在线的备份和灾难恢复。云存储满足了大规模海量数据的存储和计算需求。如亚马逊的 S3 和 Google 的服务都是基础的云存储，以及本文研究的重点 Hadoop 中 HDFS 也实现了云存储的功能，可以通过扩展来增加存储容量。

云存储可以很好的对互联网中各种存储设备进行统一的管理，为针对海量数据的存储、计算和管理提供低成本商业化的服务。极大减少了用户的投入，为并行计算提供数据支持。

1.1.1.3 并行计算与 MapReduce

并行计算是计算机科学中重要的研究内容，已有几十年的发展历程。用并行计算求解问题的大致过程为：对于一个给定的应用问题，首先，计算科学家将这个应用转化为一个数值或非数值的计算问题；然后计算机科学家对此计算问题设计并行算法，并通过某种并行编程语言实现它；最后应用领域的专家在某台具体的并行计算机上运行应用软件求解此问题。由此，可以发现并行计算由以下几个部分构成：并行计算机(并行计算的硬件平台)，并行算法(并行计算的理论基础)，并行程序设计(并行计算的软件支撑)，并行应用(并行计算的发展动力)。并行计算的一个非常重要的思想就是通过大量的计算节点，将计算任务分割成为每个节点可以接受的计算规模，最终将计算的结果合并起来^[20]。这要求，计算问题本身是必须可以分割的。

所谓并行计算可分为时间上的并行和空间上的并行。时间上的并行就是指流水线技术，而空间上的并行则是指用多个处理器并发的执行计算。并行计算科学中主要研究的是空间上的并行问题。从程序和算法设计人员的角度来看，并行计算又可分为数据并行和任务并行。一般来说，因为数据并行主要是将一个大任务化解成相同的各个子任务，比任务并行要容易处理^[21]。

尽管我国在并行计算方面开展的研究和应用较早，目前也拥有很多的并行计算资源，但研究和应用的成效相对美日等发达国家还存在较大的差距，有待进一

步的提高和发展。因此，需要形成一体化并行计算研究体系和方法来解决目前的困境。

并行计算一体化的研究方法：为了解决上面所提到的并行计算研究中存在的问题，亟待建立一个完整的科学的研究体系。并行计算研究历程，既有高潮也有低谷，究其原因是，它没有形成自身的一套研究方法学。文献[20]中已提出了一套并行算法的研究方法学，即“理论-设计-实现-应用”的研究体系，也就是所谓的并行算法研究的生态环境。其中，算法理论包括并行计算模型和并行计算复杂性等；算法设计包括并行算法的设计和并行算法的分析等；算法实现包括软件支撑和硬件平台等；并行应用包括科学工程计算应用和社会计算应用等。

并行计算作为并行算法的一个超集，更加需要建立“结构→算法→编程→应用”的一体化研究方法，这样才能稳定的可持续发展并且变得更加实用。文章[20]认为：研究并行计算的人一定要懂并行计算机，能够设计并行算法，要清楚地知道如何在并行计算机上用合适的编程语言来实现，从而解决实际的应用问题。如图1所示，并行计算的结构部分包含了高端的高性能计算机和低端的普及型计算机；并行计算的算法部分包含了并行算法的设计与分析以及算法库和测试库；并行计算的编程部分包含了并行编程模型以及并行编程的环境工具；并行计算的应用部分包含了科学工程应用和各种新型的应用。

并行计算的目标是求解大规模问题和复杂系统。并行计算可以节省时间和成本，即使用更多的资源使一个任务可以提前完成，而且会节约潜在的成本。且可以使用便宜的、普通的CPU来构建并行计算集群，如现在的云计算系统。对于解决更大规模的问题，因为单个计算机内存和系统资源的限制，很多庞大而复杂的问题是无法完成的。只有借助于并行计算的方式得以完成。另外并行计算模式可以提供很好的并行性，可以使多个资源在同一时刻执行多个任务和计算。并行计算的应用领域非常广泛，从复杂的科学计算到基于真实世界的工程问题建模，同时也有着越来越多的商务应用如数据挖掘、网络搜索引擎等。创建和使用并行计算机的主要原因是因为并行计算机是解决单处理器速度瓶颈的最好方法之一。并行计算的体系结构多发展为以多核为主流。另外就是发展出了以数据为中心的云计算模式。云计算可以看作为分布式计算(distributed computing)、并行计算(parallel computing)和网格计算(grid computing)的最新发展^{[6][20]}。它的产生和成长来自于业界的需要和推动，已经得到IBM、Google、微软、雅虎、SUN等全球主要信息技术公司的支持。

云计算意味着对于服务器端的并行计算要求的增强，因为数以万计用户的应用都是通过互联网在云端来实现的，它在带来用户工作方式和商业模式的根本性改变的同时，也对大规模并行计算的技术提出了新的要求。

本篇论文主要关注 Google 构造的 MapReduce 编程框架, MapReduce 编程框架是用来支持并行计算, 应用程序编写人员只需要专注于应用程序本身, 对于怎样通过分布式集群来支持并行计算, 以及可靠性和可扩展性, 则交由底层平台去处理, 从而保证了后台复杂的并行执行和任务调度向用户和编程人员透明[13]。MapReduce 属于并行计算的编程部分, 是一个简化的并行计算的编程模型, 它让那些没有多少并行计算经验的开发人员也可以开发并行应用。这也就是 MapReduce 的价值所在, 通过简化编程模型, 降低了开发并行应用的入门门槛。相对于现在普通的开发而言, 并行计算需要更多的专业知识, 有了 MapReduce, 并行计算就可以得到更广泛的应用。MapReduce 模型实质上是通过对一个大任务进行分解, 分解出更多的可以并发执行的小任务, 这些小任务则在大规模数据上通过并行的方式完成计算任务。

Google 提出的处理大规模数据的分布式编程框架 MapReduce, 不仅是一个处理和产生大规模数据集的编程模型, 同时也是一种高效的任务调度模型, 它通过“Map”和“Reduce”这样两个简单的概念来构成运算的基本单元^[14]。同时, Google 也发表了 GFS、BigTable 等底层系统以应用 MapReduce 模型。在 2007 年, Google’s MapReduce Programming Model-Revisted 论文发表, 进一步详细介绍了 Google MapReduce 模型以及 Sazwall 并行处理海量数据分析语言。Google 公司以 MapReduce 作为基石, 逐步发展成为全球互联网企业的领头羊。本文使用开源的 Hadoop 进行 MapReduce 的学习和研究。Hadoop 作为 Apache 基金会资助的开源项目, 由 Doug Cutting 带领的团队进行开发, 基于 Lucene 和 Nutch 等开源项目, 实现了 Google 的 GFS 和 MapReduce 思想^[15]。Hadoop 是一种流行的 MapReduce 计算模型的开源实现, 用于大规模数据集的并行化分析处理。Hadoop 由 Hadoop MapReduce 和 HDFS (Hadoop Distributed File System) 组成。Hadoop 广泛应用于日志分析、广告计算、科研实验、Adhoc 处理、数据挖掘等^[16]。

随着 Hadoop 系统的广泛应用, 其处理的问题和领域也愈来愈多。现有的针对 Hadoop 集群的作业调度算法如 FIFO、HOD、计算能力调度算法和公平调度算法^{[16][18][19]}, 都提供了不同的作业调度策略。以上调度算法在集群的发展过程当中针对不同的应用需求实现了一定的功能。但随着新应用的出现和需求的增加, 如何更好的提高 Hadoop 集群的性能, 更好的利用 Hadoop 集群满足更多的应用需求和多用户更好的共享集群, 从而既满足对集群计算能力的需求, 又满足公平性要求成了新的挑战。

1.1.2 课题研究的目的

本文主要从以下几个方面进行相应的工作, 一是研究如何改善

MapReduce 集群作业调度时任务的数据本地性，二是，通过改善改善任务节点的数据本地性来提高集群的吞吐率和减少单个作业的平均响应时间。

满足数据本地性是指 Hadoop 系统在进行 MapReduce 作业调度时一个 Map 任务执行所在的节点也是其所需数据所在的节点。即计算程序和计算所需数据在同一个节点上，从而减少了集群系统内部的移动数据的开销。

集群的吞吐率是单位时间内整个集群所处理的作业数。提高了集群的吞吐率，则单位时间内其所处理的作业数则越多。集群的整体性能则提高。响应时间是作业完成结束的时间减去作业递交到系统的时间，是针对单个作业的性能指标。如果能够尽可能的提高每个作业的响应时间，最终将提高整体的吞吐率和性能是集群作业调度方法必须要考虑的问题。一个好的作业调度方法将会对集群的整体性能的提高起到关键性的作用。同时如何根据自己的集群（Hadoop 集群）的体系结构和技术细节来改善和提高集群的性能是本文所研究的内容。

总之，本文主要是针对开源的 Hadoop 集群系统，研究在多用户共享的条件下，如何更好的进行作业调度和设计从而更加高效的完成用户提交的作业。同时根据对 Hadoop 的研究分析，包括对现有的作业调度方法的研究和分析，来设计和完成高效的调度方法。最后，会给出具体的实验验证分析。为 Hadoop 项目的发展做出一份努力。

1.2 论文研究的主要内容

结合课题的背景，本文主要深入的研究了 MapReduce 集群现有的各种作业调度方法，并针对 Hadoop 现有调度算法数据本地性较差的问题，提出了一种基于时间的等待调度方法，该方法优先将任务调度到其所需的数据所在的节点上执行，从而实现了更好的数据本地性，有效减少计算过程中的 IO 开销，实现提高系统吞吐率和减少单个作业平均响应时间的目的。本文主要是通过对以往的调度方法的学习研究尝试给出一个能够满足多用户共享要求的高效调度方法。使得 Hadoop 集群的效能得到更好的发挥，改善用户的使用体验。具体来讲，主要内容就是改善作业调度时任务的数据本地性。

Hadoop 集群采用分布式文件存储系统 HDFS 来存储数据，数据分布在集群内不同节点上；在计算时，集群会调度 Map 任务或 Reduce 任务到节点上执行，从而存在一个节点的数据本地性问题；一个 Map 任务是一段计算程序，需要输入数据进行计算，如果一个 Map 任务被调度到不是所需要数据所在的节点，那么执行时则需要从其他节点传送数据，增加整个集群系统的 IO 开销，也增加了 Map 任务完成计算所需的时间。目的是尽可能的在作业调度时调度任务到具有数据本地性的节点上执行任务，从而提高了整个集群的计算效能。对系统的吞吐率的提高和作

业响应时间的减少都是有益的。

本文将借助对 Hadoop 集群已有的调度方法的研究和分析来构造一个高效的作业调度方法。以保证多用户公平共享集群的基础上，改善 Hadoop 集群的整体性能。

同时为了验证方法的有效性，我们对提出的作业调度方法给出了设计与实现，并进行了实验验证。结果表明，基于时间的等待调度方法在保证多用户公平共享集群的基础上，节点的数据本地性得到很大的提高，有效增加了集群系统的吞吐量，有效减少了单个作业的平均响应时间。

1.3 论文的组织结构

论文分七章进行叙述，其中第三、第四、第五和第六章是本文的重点。

第一章：绪论。

第二章：MapReduce 相关技术介绍。

第三章：Hadoop 集群的作业调度方法研究。

第四章：基于时间的等待调度方法。

第五章：系统的设计与实现。

第六章：实验验证分析。

第七章：总结与展望。

最后是结束语。结束语总结了论文的主要工作，分析了工作的优点和不足，对未来工作进行了展望。

第二章 MapReduce 相关技术介绍

MapReduce 是 Google 提出的一个软件架构，用于大规模数据集（大于 1TB）的并行运算^[14]。概念“Map（映射）”和“Reduce（化简）”，和他们的主要思想，都是从函数式编程语言借鉴而来的，还有从矢量编程语言借鉴来的特性。MapReduce 是一种简化的分布式编程模式，让程序自动分布到一个由普通机器组成的超大集群上并发执行。就如同 java 程序员可以不考虑内存泄露一样，MapReduce 的 run-time 系统会解决输入数据的分布细节，跨越机器集群的程序执行调度，处理机器的失效，并且管理机器之间的通讯请求。这样的模式允许程序员可以不需要有什么并发处理或者分布式系统的经验，就可以处理超大的分布式系统的资源。

2.1 MapReduce 产生背景

在 Google 每天都有海量的数据需要处理，而且随着时间的积累数据量也在不断增大。其程序员需要编写大量的具有专门目的的计算程序，以用于处理海量的原始数据。如爬虫文档、Web 请求日志、查询请求等等。同时要的计算不同类型的派生数据仍旧是海量的，且成本和时间要求是有限的。此类的计算在概念理解上是容易的，但是实际实现是则因为输入的数据量的巨大，且计算处理需要分布在大量的机器上才有可能在一定的时间内完成。如何实现并行计算，分发数据，容错，管理调度和监控，综合起来，将使原本看似容易的计算，因需要大量的复杂的代码来处理，而变得异常的艰难。即简单的计算在数据规模较大时变得复杂而难以控制。为了更加有效和简洁的处理此类问题，Google 提出了 MapReduce 编程模型，它可以隐藏并行化、容错、数据分布、负载均衡等细节，只需要执行简单的计算。其工作模式是在输入数据的逻辑记录上应用 map 操作，来计算出一个中间 key/value 对集；在所有具有相同 key 的 value 上应用 reduce 操作，来适当地合并派生的数据。功能模型的使用，再结合用户指定的 map 和 reduce 操作，可以非常容易的实现大规模并行化计算，同时使用重启作为初级机制可以很容易地实现容错。在 Google，MapReduce 用在非常广泛的应用程序中，包括“分布 grep，分布排序，web 连接图反转，每台机器的词矢量，web 访问日志分析，反向索引构建，文档聚类，机器学习，基于统计的机器翻译”^[14]。

2.2MapReduce 模型

2.2.1MapReduce 思想

MapReduce 将数据处理过程归结为 Map 和 Reduce 两个步骤。将分布环境中大数据集上的 map 计算和 reduce 计算涉及的公共部分（分解、合并、并行、管理调度等）抽象出来，以提供简单的接口，方便用户实现业务逻辑（map 和 reduce 操作）^{[22][23]}。

MapReduce 模型以一个 key/value 对集作为输入，来产生一个 key/value 对集的输出。MapReduce 库的用户使用两个函数：Map 和 Reduce 来实现这个计算^[24]。

用户自定义的 Map 函数，接受一个输入对，产生中间的 key/value 对的集。MapReduce 库把所有具有相同的 key 值的中间结果的 value 值聚合在一起，并传递给 Reduce 函数进行下一步的计算。

用户自定义的 Reduce 函数，接受一个中间的 key I 和这个 key 的 value 值的集合，并合并这些 value 集产生一个可能更小的 value 集。一般情况下，一次 Reduce 调用只产生 0 个或 1 个输出 value 值。通过一个迭代器把中间结果的 value 值提供给用户自定义的 Reduce 函数。

2.2.2 MapReduce 基本概念

在具体的 MapReduce 实现系统中，如 Google、Amazon 和 Hadoop^{[14][25][26][48]}。系统会产生一个 Job（我们称之为作业）作为计算程序，一个 Job 也即一个 MapReduce 程序，是对具体计算任务的编程实现。由用户负责自己所需数据处理程序的编写，然后将 MapReduce 程序提交到系统。执行过程中一个 Job 会包含很多的 Task（我们称之为任务），主要有 Map Task 和 Reduce Task，负责在具体的物理机器上完成数据处理任务。其中 Map Task 负责部分数据的转换，用户实现的 map 接口负责数据的转换逻辑。Reduce Task 负责对中间结果的规约，用户实现的 reduce 接口负责数据的规约逻辑。

MapReduce 模型的主要贡献是通过简单的接口来实现自动的并行化和大规模的分布式计算，通过使用 MapReduce 模型接口实现在大量普通的 PC 机上高性能计算。在具体的实现系统中，通过将一个作业的 Map 和 Reduce 任务并行的调度到普通的 PC 机上执行数据运算来实现其高性能大规模的分布式计算。

2.2.3MapReduce 结构

一个 MapReduce 集群有成百上千台机器组成，大部分机器节点都是同构的，

一般分为两种类型：主控制节点和执行节点^[22]。在 Google 的集群中主控节点为 Master，执行节点称为 Worker。Hadoop 系统中主控节点称为 JobTracker，执行节点称为 TaskTracker。

主控节点负责接收用户提交的作业（Job），并把一个个任务（Task）分派到执行节点上，监控任务的执行、重新执行失效的任务。执行节点是任务（Task）的执行节点，执行分派的任务，存储运算的中间结果，并在需要时传递中间结果到下一个执行的任务。一个集群的结构图如图 2.1 所示：

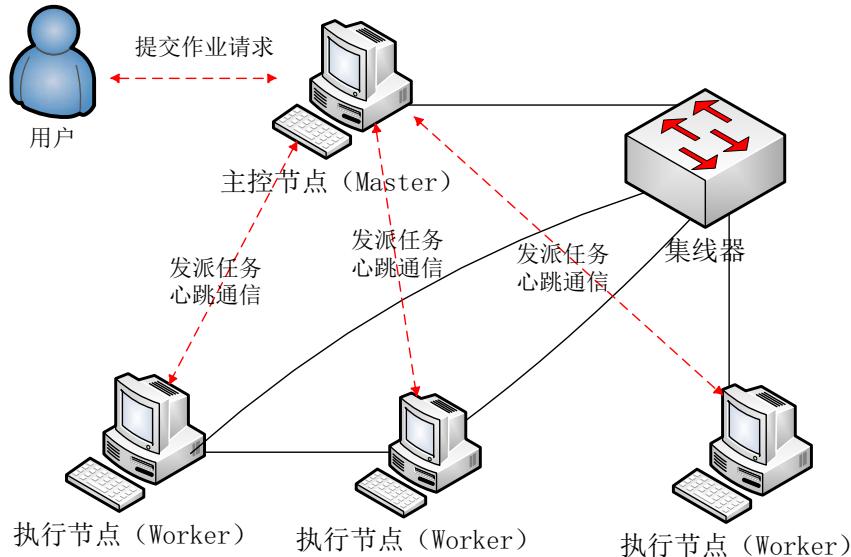


图 2.1 MapReduce 集群节点结构图

2.2.4 MapReduce 执行

MapReduce 模型有不同的实现方式，Hadoop 中使用以太网交换机连接不同的机架，由普通 PC 机组成机架，再由众多的机架组成大型集群。具体执行时通过将用户编写的 MapReduce 函数程序载入系统来完成运算，具体的执行过程如下图 2.2 所示^[16]：

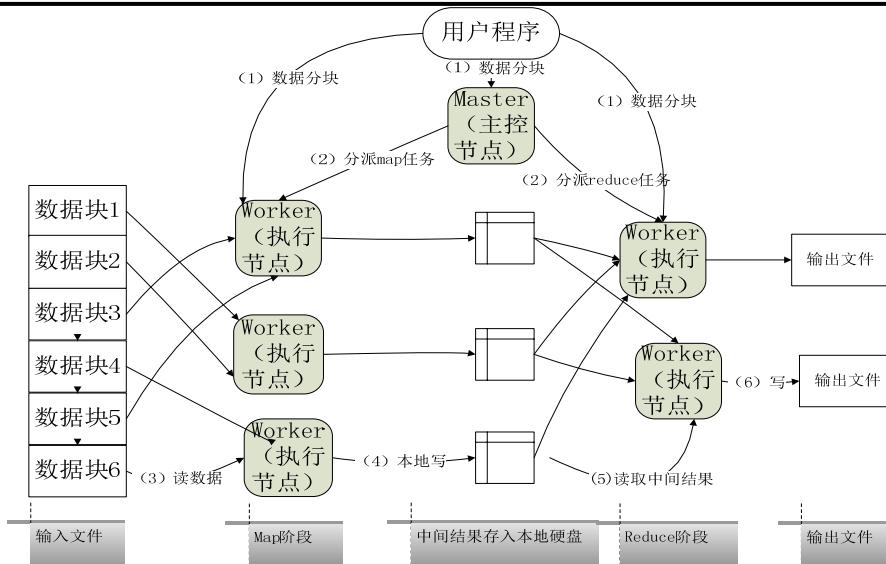


图 2.2 MapReduce 模型执行示意图

通过将 Map 调用的输入数据自动分割为 M 个数据块的集合, Map 调用被分布到多台机器上执行。输入的数据块能够在不同的机器上并行处理。使用分区函数将 Map 调用产生的中间 key 值分成 R 个不同分区(例如, $\text{hash}(\text{key}) \bmod R$), Reduce 调用也被分布到多台机器上执行。分区数量 (R) 和分区函数由用户来指定。

图 2.2 展示了 MapReduce 实现中操作的全部流程。当用户调用 MapReduce 函数时, 将发生下面的一系列动作 (下面的序号和图 2.2 中 的序号一一对应) :

1. 用户程序首先调用的 MapReduce 函数将输入文件分成 M 个数据块, 每个数据块的大小一般从 16MB 到 64MB(可以通过可选的参数来控制每个数据块的大小)。每个数据块会在集群系统内创建副本, 副本数一般为 3。然后用户程序在集群中创建大量的程序副本。

2. 这些程序副本中的有一个特殊的程序—master (主控节点)。副本中其它的程序都是 worker(执行节点)程序, 由 master 分配任务。有 M 个 map 任务和 R 个 reduce 任务将被分配, master 将一个 map 任务或 reduce 任务分配给一个空闲的 worker。

3. 被分配了 map 任务的 worker 程序读取相关的输入数据块, 从输入的数据块中解析出 key/value pair, 然后把 key/value pair 传递给用户自定义的 Map 函数, 由 Map 函数生成并输出的中间 key/value pair, 并缓存在内存中。

4. 缓存中的 key/value pair 通过分区函数分成 R 个区域, 之后周期性的写入到本地磁盘上。缓存的 key/value pair 在本地磁盘上的存储位置将被回传给 master, 由 master 负责把这些存储位置再传送给 Reduce worker。

5. 当 Reduce worker 程序接收到 master 程序发来的数据存储位置信息后, 使用 RPC 从 Map worker 所在主机的磁盘上读取这些缓存数据。当 Reduce worker 读取了所有的中间数据后, 通过对 key 进行排序后使得具有相同 key 值的数据聚合在一

起。由于许多不同的 key 值会映射到相同的 Reduce 任务上，因此必须进行排序。如果中间数据太大无法在内存中完成排序，那么就要在外部进行排序。

6. Reduce worker 程序遍历排序后的中间数据，对于每一个唯一的中间 key 值，Reduce worker 程序将这个 key 值和它相关的中间 value 值的集合传递给用户自定义的 Reduce 函数。Reduce 函数的输出被追加到所属分区的输出文件。

7. 当所有的 Map 和 Reduce 任务都完成之后，master 唤醒用户程序。在这个时候，在用户程序里的对 MapReduce 调用才返回。

在成功完成任务之后，MapReduce 的输出存放在 R 个输出文件中（对应每个 Reduce 任务产生一个输出文件，文件名由用户指定）。一般情况下，用户不需要将这 R 个输出文件合并成一个文件，他们经常把这些文件作为另外一个 MapReduce 的输入，或者在另外一个可以处理多个分割文件的分布式应用中使用。

2.2.5 MapReduce 实例

下面通过一个简单的实例来了解 MapReduce 实际计算过程。下面是一个计算每个单词在一个大的文档集合中出现的次数的例子^[26]。伪代码表示如下：

```
map(String key, String value)
//key: 文档名  value: 文档内容
for each word w in value:
    EmitIntermediate(w,"1");
reduce(String key, Iterator values):
//key: 一个单词  value: 一个技术表
int result=0;
for each v in values:
    result+=ParseInt(v);
    Emit(AsString(result));
```

在这个例子里，map 函数计算每个词及其出现的次数(在这个简单的例子里就是 1)。reduce 函数把产生的每一个特定的词的计数合并，计算出每个单词在文档出现的次数。

下面是一些可以容易的用 MapReduce 计算来表示的简单程序^{[16][14]}。

分布式的 Grep(UNIX 工具程序，可做文件内的字符串查找)：如果输入行匹配给定的样式，map 函数就输出这一行。reduce 函数就是把中间数据复制到输出。

计算 URL 访问频率：map 函数处理 web 页面请求的记录，输出(URL, 1)。reduce 函数把相同 URL 的 value 都加起来，产生一个(URL, 记录总数)的对。

倒转网络链接图：map 函数为每个链接输出(目标，源)对，一个 URL 叫做目标，包含这个 URL 的页面叫做源。reduce 函数根据给定的相关目标 URLs 连接所

有的源 URLs 形成一个列表，产生(目标，源列表)对。

每个主机的术语向量：一个术语向量用一个(词，频率)列表来概述出现在一个文档或一个文档集中的最重要的一些词。map 函数为每一个输入文档产生一个(主机名，术语向量)对(主机名来自文档的 URL)。reduce 函数接收给定主机的所有文档的术语向量。它把这些术语向量加在一起，丢弃低频的术语，然后产生一个最终的(主机名，术语向量)对。

倒排索引：map 函数分析每个文档，然后产生一个(词，文档号)对的序列。reduce 函数接受一个给定词的所有对，排序相应的文档 IDs，并且产生一个(词，文档 ID 列表)对。所有的输出对集形成一个简单的倒排索引。它可以简单的增加跟踪词位置的计算。

分布式排序：map 函数从每个记录提取 key，并且产生一个(key，record)对。reduce 函数不改变任何的对。

下节将介绍 MapReduce 模型的开源实现 Hadoop。

2.3 Hadoop 框架介绍

2.3.1 Hadoop 简介

Hadoop 是一种流行的 MapReduce 计算模型的开源实现，是一个开源的分布式计算框架，由 Apache 开源组织开发，用于大规模数据集的并行化分析处理。Hadoop 框架最核心的组成是：MapReduce、HDFS、Hbase。在很多大型网站上都已得到了应用，如亚马逊、Facebook 和 Yahoo！等。其主要应用领域有大规模数据分析处理、计算领域由 TB/PB 量级数据量导致的相关问题等^{[15][16][26]}。

MapReduce 计算模型是 Hadoop 的核心组成部分之一，是 Google 工程师提出的 MapReduce 编程模型的开源实现^[14]。用以进行大规模数据的并行计算。

HDFS 是 Hadoop 分布式文件系统（Hadoop Distributed File System）的缩写，为分布式计算存储提供了底层支持。HDFS 由 java 语言实现，设计目标就是能够方便灵活地跨异构硬件和软件平台部署和运行。

HBase 是 Hadoop 的正式子项目，它是一个面向列的分布式数据库，其思想源于 Google 的 BigTable 论文。建立在 HDFS 之上，HBase 使用和 Bigtable 非常相同的数据模型。用户存储数据行在一个表里。一个数据行拥有一个可选择的键和任意数量的列。表是疏松的存储的，因此用户可以给行定义各种不同的列。

Hadoop 优点：开源，便于深入了解和改进；结构清晰；使用 Java 编程，便于安装实验。本文选择 Hadoop 作为对 MapReduce 集群多用户作业调度方法研究的系统正式因为其具有以上特点。

下面将对 Hadoop 中两个最核心的部分 MapReduce 和 HDFS 进行专门的介绍。

2.3.2 Hadoop 中 MapReduce 作业构成

用户配置和提交一个 MapReduce 作业到 Hadoop 系统，这个作业可以分解为一个 map 任务集、混洗操作（shuffles）、一个排序（sort）操作和一个 reduce 任务集。Hadoop 系统接下来会管理这些任务的分发和执行，并收集输出结果，把作业的状态返回给用户^{[25][15]}。

一个作业的组成部分见下表 2.1 和图 2.3 所示：作业配置（包括输入数据的路径和格式，输出结果的路径和格式），负责进行运算处理的 Map 函数，Reduce 函数等皆由用户来指定。对于输入数据的分片和分发，map 任务的调度执行，中间结果的处理，reduce 任务的调度执行，输出结果的收集存储，向用户报告作业状态等皆由 Hadoop 框架负责完成^[16]。

表 2.1 Hadoop 中 MapReduce 作业的组成

组 件	处理者
作业配置	用户
输入的分片和分发	Hadoop 框架
map 任务在其输入数据上开始执行	Hadoop 框架
Map 函数，每个输入 key/value 队调用一次	用户
Shuffle，对每个 map 的输出进行分区和分类处理	Hadoop 框架
Sort，合并分类对所有 map 结果的 shuffle 输入	Hadoop 框架
Reduce 任务在其输入数据上开始执行	Hadoop 框架
Reduce 函数，针对唯一 key 调用一次	用户
输出结果的收集并存储到作业配置中制定的路径，输出结果有 N 部分，N 是 reduce 任务的个数	Hadoop 框架

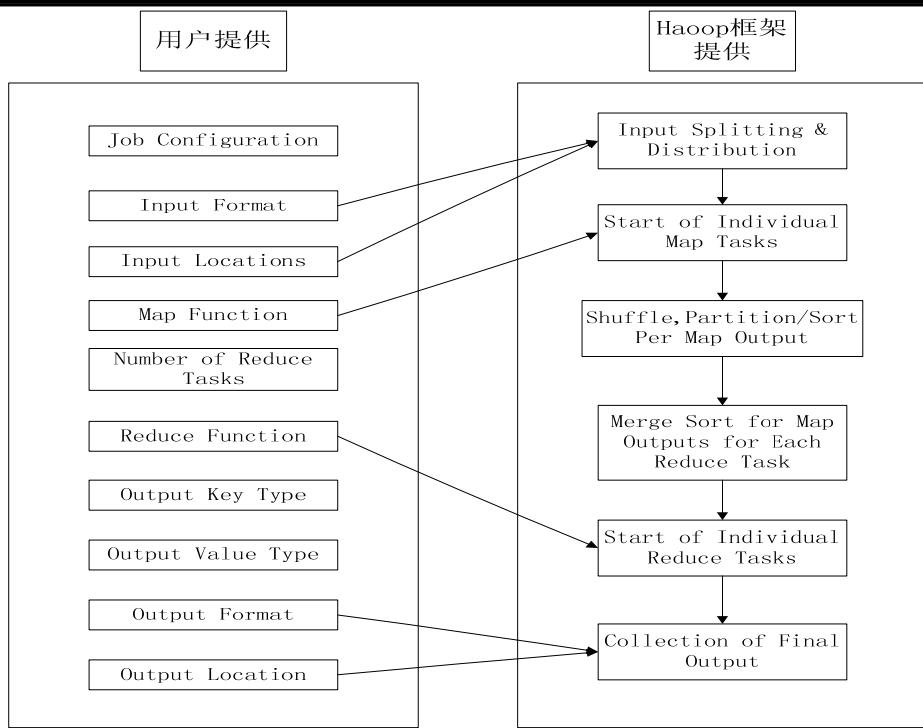
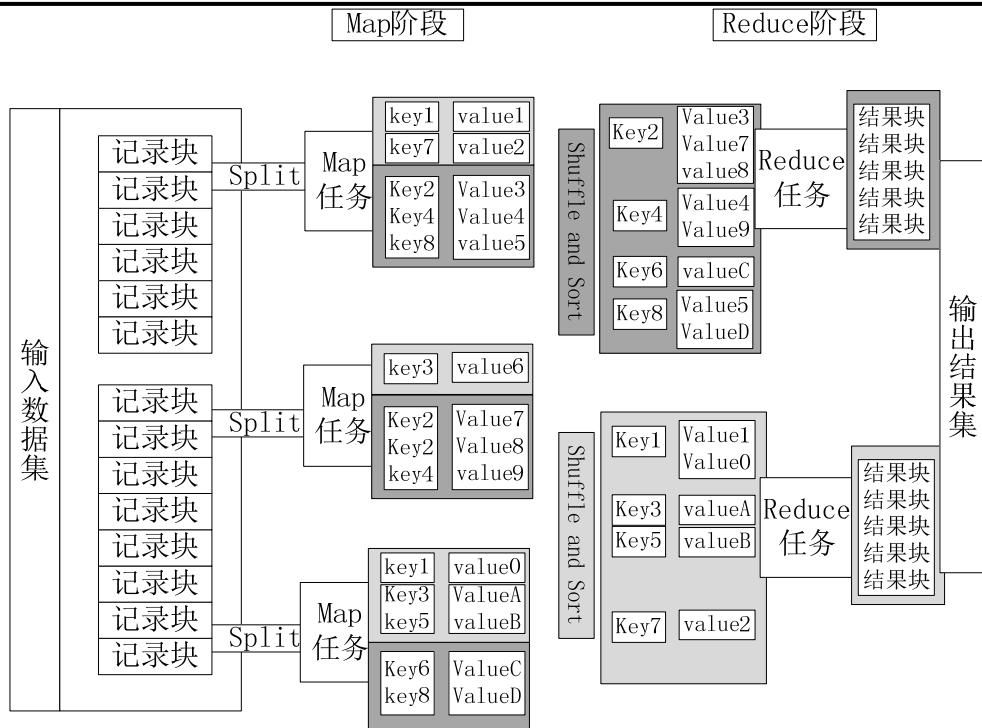


图 2.3 Hadoop 中 MapReduce 作业的组成

2.3.3 Hadoop 中 MapReduce 实现

Hadoop 采用了 JobTracker/TaskTracker 的结构来实现 MapReduce 计算模型。该结构也是一种典型的主从式结构。MapReduce 计算模型处理作业时分为 Map 阶段和 Reduce 阶段，Map 阶段负责最初的提取和转化，使各自独立的块（记录）可以并行的得到处理，Reduce 阶段是聚合操作，把所有相关的记录进行单独的处理 [25][26][26]。

Hadoop 中的 MapReduce 主要思想是原始的输入被分成逻辑块，每个逻辑块最初由一个 map 任务独立的处理。所有 map 任务独自处理的结果被分成不相关的中间结果集，并被分类；每个分类的块由一个 reduce 任务处理。下图 2.4 为 Hadoop 中 MapReduce 执行流程图：

图 2.4 Hadoop 的 MapReduce 执行流程^[26]

一个 map 任务可以在集群内的任一计算节点上运行，多个 map 任务则在集群内的多个节点上并行运行。map 任务负责把输入的记录转化成 key/value 对，所有 map 任务的输出会进行分类，每一个分类经过排序并传递给一个 reduce 任务，经过排序后的 key 值和与这些 key 值相关的 value 值将被 reduce 任务处理。在一个 Hadoop 集群内可以有多个 reduce 任务并行运行。

作为应用开发者需要向 Hadoop 平台提供四个部分：读取输入记录并把它们转化成 key/value 对的类；一个 map 函数；一个 reduce 函数；一个把 reduce 函数输出的 key/value 对结果转化成输出结果的类。有了这四个部分，Hadoop 就可以完成用户的计算，并提供了高的容错性和极大降低计算控制的难度。

如图 2.3 所示，一个 MapReduce 任务首先将输入数据集分割成相互独立的若干数据块，从而多个 map 任务能够完全并发的执行。在分割时并不需要了解文件的内部逻辑结构，具体的分割模式既可以由用户自己指定，也可以使用 Hadoop 已定义的几种简单分隔方式。

当一个 map 任务开始时，InputFormat 类会分析输入的文件，产生<key,value>对。这时，用户自定义的 mapper 类可以对键值对进行任意的操作。完成后，则调用 OutputCollect 类中的方法重新收集自己定义的键值对。此时键与值的类型不必与输入时的相同。产生的输出必需用一个 Key 类和一个 value 类，这是因为 Map 的输出结果要被以 SequenceFile 的形式写入磁盘。map 的输入和输出不必在类型上有联系。对于每个 map 输出的结果，可以通过 combiner 进行初步的合并。合并工

作是在进行 map 操作的同一个节点上进行的。Combiner 的具体操作与 reducer 相同，主要是为了减少中间结果传输时的网络流量。

在中间结果传送给用户自定义 reducer 操作之前，还需要对中间结果进行分区。这个工作由 partitioner 类来完成，默认是以 HashPartitioner 类用 key 类的哈希函数产生的哈希值来区分。然后将每个 map 中间文件中具有相同计算结果的键值对合并到同一个文件，这样就可以保证同样的 key 只会被送给同一个 reducer 来处理。

当一个 reduce 任务开始时，它的输入是分散在各个节点上的 map 的输出文件里。这就需要 reduce 通过 HTTP 将中间结果的相应分区取到本地。与此同时，系统会将 Reducer 的输入根据 key 来排序，使得相同的 key 排列在一起。接着，通过用户自定义的 reducer 函数，对具有相同 key 的键值对进行归并。Reduce 的结果通过 OutputCollector 类输出到 HDFS。

2.3.4HDFS 介绍

人类已经进入海量数据时代，正以前所未有的速度产生数据。为了更好的利用这些数据就需要对这些数据进行有效的存储和管理。MapReduce 编程模型是一种基于普通商业集群的高可扩展数据密集型计算模式。MapReduce 催生了 Hadoop，而 HDFS 是 Hadoop 的关键组成部分。HDFS 是 Hadoop Distributed File System 的缩写，即 Hadoop 分布式文件系统，主要用于存储 MapReduce 应用的输入和输出数据。HDFS 提供了一个集群范围内的全局文件访问机制，HDFS 以 java 的方式实现了一个用户级的文件系统，底层采用每个节点的本地文件系统（如：ext3、NTFS）存储数据。HDFS 每个文件被划分为一组超大文件块，每个文件块典型大小为 64MB，每个块以本地文件系统中的一个独立文件形式存储^[28]。HDFS 的设计目标是能够方便灵活地跨异构硬件和软件平台部署和运行。

HDFS 包括两个服务：NameNode 和 DataNode。NameNode 负责维护 HDFS 的目录树，是 HDFS 中的集中式服务，被部署在集群中某一台节点。客户端通过联系该结点获取文件系统的名字服务，如：打开、关闭、重命名、删除等。NameNode 节点不存储文件的数据信息，只维护文件名与所属块之间的映射关系，数据块存储在一组 DataNode 结点上。

除了单一的 NameNode 节点，其余结点都是 DataNode 节点，提供文件数据的存储服务，每个 DataNode 节点为本地或远程的客户端 HDFS 数据块。每个数据块被存储为数据节点本地文件系统的一个独立的文件。

2.3.5 存储本地化

Hadoop 框架的设计目的是针对大规模数据的存储和计算。Hadoop 使用 HDFS

存储数据，HDFS 中存储数据的节点也是 MapReduce 计算所用的节点。正因为数据规模之大，移动数据的开销太大，Hadoop 通过移动计算程序来代替移动数据去完成计算。这就需要在执行计算任务时，要把 map 程序移动其所需数据所在的节点上执行。即满足数据本地性要求，也是作业调度方法所研究的主要内容。在作业调度时把 map 任务调度到输入数据所在的节点上。输入数据会按块存储在 HDFS 中，每一个块存在一个独立的节点上，调度时只需把 map 任务调度到该节点即可^[29]。

在调度时，如果满足了数据的本地性可大大提高系统的性能，大大减少系统的 IO 开销，增加系统的吞吐量，减少了作业的响应时间。

2.4 MapReduce 模型的其他实现

MapReduce 模型被提出以后，针对它的研究如火如荼，针对它的应用领域也非常广泛，但更多的工作是根据自己的应用或者不同的体系结构来实现自己的 MapReduce 系统，下面将介绍几种已有的 MapReduce 实现。

2.4.1 Google 的 MapReduce 实现

Google 为了解决概念简单而输入数据巨大的问题，同时又要求在合理的时间内完成，需要使用并发的方式完成计算。但是如何更好的分发数据、并发计算和处理失败，Google 设计了 MapReduce 计算模型。用来处理计算简单但数据量巨大的计算，这些简单计算原本很简单，但是并发处理细节，容错细节，以及数据分布细节，负载均衡等等细节问题导致代码非常复杂。所以可以把这些公共的细节抽象到一个库中，由一个运行时系统来负责。而将对数据的操作抽象为 map 和 reduce 两个概念，这种抽象是源自 Lisp 以及其它很多函数式语言的 map 和 reduce 概念。大部分对数据的操作都和 map 相关，这些 map 负责处理输入记录中的每个逻辑“record”，并且会产生一组中间的 key/value 键值对，接着在所有具有相同 key 的中间结果上执行 reduce 操作，这样就能将适当的数据进行合并^[14]。Google 的 MapReduce 的实现，详细内容在第二章前两节已有论述。

2.4.2 Stanford 的 Phoenix 系统

Phoenix(C. Ranger et al. 2007)是共享内存系统上的一个 MapReduce 实现，它包括一组编程接口 API 和一个运行时系统。Phoenix 的运行时系统会自动的管理线程创建，动态任务调度，数据划分和容错处理^[30]。

Phoenix 采用线程来实现 Map 和 Reduce 任务，同时使用共享内存缓冲区的方

法来实现通信，这样就避免了数据拷贝产生的开销。运行时系统动态地在各个可用核上调度任务从而达到负载平衡，通过并行粒度调整和并行任务分配来保证数据的本地性。运行时系统还通过任务重做或者重新分配来处理传输或者参数错误，然后把输出与其它输出进行合并。

Phoenix 的 API 由 C 和 C++ 编写，而这些 API 也可以用 Java 或 C 撰这样的语言来定义。它一共可以分为两类：第一类可以被程序用来使用完成初始化系统和产生输出；第二类包括一系列用户定义的函数。Phoenix 运行时的任务调度器可由用户配置，它将创建并控制线程完成所有的 Map 和 Reduce 任务，同时管理用于通信的缓冲区。在初始化完成以后，调度器决定有多少个核要参与计算，并为每个核产生一个线程用来动态的执行 Map 和 Reduce 任务。Map 阶段分割器(Splitter)先把输入数据分成大小相同的块，然后交由 map 任务线程去操作，Partition 函数会把中间结果按 key 进行统计。调度器必须等所有的 map 工作完成以后才初始化 Reduce 阶段，动态分配 reduce 任务并确保该阶段负载平衡。最后一步就是将所有的最终输出结果合并并放入连续的内存缓冲区。

2.4.3 GPU 上的 MapReduce 实现

Mars(B. He et al. 2008)是一个设计并实现在图形处理器(GPU)上的 MapReduce 系统。与通用的 CPU 相比，GPU 拥有更高的计算能力和更宽的带宽，但是由于它的体系结构是按照协处理器(co-processor)来设计的，并且它的程序设计接口是为图形处理而专门设计的，因此对 GPU 编程还是比较复杂的。作为第一种尝试使用 GPU 为硬件而实现 MapReduce 系统，Mars 所使用的硬件是 NVIDIA G80 GPU，这个 GPU 包含一百个图形处理单元^[7]。

由于 GPU 的计算能力和可编程性越来越高，很多众核处理器将 GPU 作为一个协作处理器来完成计算任务(A. Ailamaki et al. 2006)，由 GPU 组成众核处理器。GPU 由很多 SIMD 的多核处理单元组成，并且支持上千个线程的并发执行。与 CPU 的线程相比，GPU 中线程的上下文切换开销要小的多。每个多核处理单元上的所有线程被划分成一个组进行管理，该组中的所有线程共享这个多核处理单元上的诸如寄存器等资源。

Mars 上的 MapReduce 流程如下：在 map 阶段，划分器把输入的 key/value 对划分成与线程数目一样多的数据块，每个 GPU 线程处理一个数据块。这样使得所有线程在 map 阶段可以达到负载平衡。map 阶段完成以后，产生的中间结果 key/value 会被重新组织，具有相同 key 的 value 会被有序存储。在 reduce 阶段，划分器会把存储的中间结果 key/value 对划分成更小的数据块，具有相同 key 的 key/value 对会被分在同一个块里面，同样数据块的数目与线程的数目是相同的。

每个进程有一个很长的 ID，它与每个数据块拥有的一个很长的 key 所对应。reduce 阶段通常也是负载平衡的，除非 value 列表的长度不同。而由于当前的 GPU 并不支持动态线程调度，因此 Mars 也不支持动态的负载平衡。在最后一步，所有进程产生的输出都会被存入同一个缓冲区。

2.5 本章小结

本章首先介绍提出了本文将要研究的主要对象 MapReduce 计算模型和 Hadoop 计算框架，然后分别介绍了 MapReduce 计算模型的基本含义和应用领域，以及在 Hadoop 框架中 MapReduce 计算模型的实现。最后介绍了 Hadoop 的主要组成部件以及 MapReduce 模型的其他实现方式。

第三章 Hadoop 集群的作业调度方法研究

本章主要针对已知的基于 Hadoop 集群的作业调度方法, 进行深入的分析研究, 并着重针对公平调度方法给出详细的分析, 并指出其在面对多用户情况下应用的不足。

3.1 Hadoop 结构

从数据存储管理角度, Hadoop 中节点分为 NameNode 和 DataNode。NameNode 和 DataNode 已在上文 2.3.3 做了介绍, 在此不再赘述。从作业调度和管理的角度, Hadoop 中节分为 JobTracker 和 TaskTracker。其中, JobTracker 负责对集群中的 MapReduce 作业进行调度以及执行的监督和管理, 其他节点被称为 TaskTracker, 负责 MapReduce 作业中 Map 任务和 Reduce 任务具体实现。当一个用户向 Hadoop 集群提交一个 MapReduce 作业时, 首先将相关的输入数据划分为一定数量的片段, 然后 JobTracker 负责将作业中的 Map 任务调度到空闲的 TaskTracker 节点上执行计算, 接着对得到的中间结果进行重新划分排序后调度给空闲的 TaskTracker 节点执行 Reduce 任务, 最后得出每个 key 值相对应的数据集的计算结果。循环重复此计算过程, 直至用户提交的作业被计算完毕^{[15][28]}。

Hadoop 中的作业调度是 JobTracker 指派任务 (tasks) 到相应 TaskTracker 上执行的过程。下面对 Hadoop 中的调度算法进行介绍和分析。

3.2 Hadoop 调度方法研究

作业调度即调度资源进行作业运行的过程。在 Hadoop 集群中则指主要强调对作业执行顺序和计算资源的分配进行控制。作业调度方法直接关系到 Hadoop 集群的整体性能和系统资源的利用情况。在多用户情况下如何实现效率和公平的均衡是其主要目的。针对 MapReduce 集群先后提出了很多的调度方法, 有 FIFO 调度, HOD 调度, 计算能力调度, 公平调度等^{[18][31][19]}。

3.2.1 FIFO 方法

Hadoop 默认的调度策略是带有优先级的 FIFO (First In First Out)。所有的用户作业被提交到唯一的一个队列中。按照优先级高低和提交时间先后的顺序扫描整个作业队列选择一个满足要求的作业执行。

FIFO 实现较简单, 整个集群的调度开销较少。FIFO 调度算法最大的缺点是在存在大作业的情况下小作业响应时间较差。忽略了不同作业的需求差异。如生产

性作业长期占据集群资源将造成其他用户的批处理作业难以忍受如此长的响应时间，也使其他用户的交互型作业变得长时间得不到处理。且资源利用率低。在多用户情况下，可能造成严重的不公平性，且系统吞吐率收到影响。

FIFO 方法主要用为 Hadoop 的默认作业级调度策略，由于其实现简单、高效，主要适用于单用户同类型作业的调度。在作业调度中二级调度即任务调度中也多有使用。

在 Hadoop 中这个问题的第一个解决方案是 HOD (Hadoop On Demand)，将在下一节做介绍。

3.2.2 HOD 方法

HOD 是一个能在共享物理集群上使用 Torque (资源管理器) 提供私有的 Hadoop MapReduce 集群和 Hadoop 分布式文件系统实例的系统。HOD 依赖资源管理器(RM)来分配节点，这些节点被用来在之上运行 hadoop 实例^[17]。

用户在提交节点上用 HOD 客户端分配所需数目节点的集群，在私有集群上供应 Hadoop。

HOD 在改善小作业的响应时间方面相较 FIFO 有了很大的进步。但是也存在着不可忽视的问题。其一，较差的数据本地性，HDFS 文件是存在所有的节点上的，而每一个私有的 Reduce 集群却运行在固定的子集节点上。这就导致一部分 map 任务的输入数据不在本私有集群的节点上，必须通过网络读取数据，从而降低了系统的吞吐率和作业的响应时间。其二，较差的资源利用率，正因为私有集群大小的固定，会出现的有的私有集群出现作业等待而有的私有集群存在空闲节点。

3.2.3 计算能力调度方法

计算能力调度方法 (Capacity Scheduler)，支持多个队列，每个队列采用的是 FIFO 调度方法。可支持优先级，但不支持优先级抢占，一旦一个作业开始执行，不会被高优先级作业中断执行。为队列定义了一个指标—队列中正在运行的任务数与其应该分得的计算资源(配置文件中为此队列分配了相应数量的资源，而实际中该队列可能没有分配到)之间的比值。当系统中出现空闲的 task tracker，算法会首先选择一个该比值最低的队列。队列被选中后，将按照作业优先级(如果支持的话)和提交时间顺序选择执行的作业。在选择作业的时候，还需要考虑作业所属的用户是否已经超出了他所能使用的资源限制。此外，还会考虑 task tracker 内存资源是否满足作业的要求^[18]。

3.2.4 公平调度方法

为了解决 HOD 调度方法产生的问题，Facebook 提出了公平调度算法（Fair Scheduling）^{[19][32]}。Fair scheduler 是一种给 job 分配资源的调度方法，通过这种调度方式，可以使得所有 job 在一定时间内能够平均共享所有资源。公平调度算法尽可能的保证每个用户都获得相等的资源份额。当单独一个作业在运行时，它将使用整个集群。当有新用户提交作业后，系统会将任务槽（task slot）赋给这些新的用户，从而使得每个用户都能获取大致等量的 CPU 资源。除了提供公平共享方法外，公平调度还提供了最小共享额度方法。每个池 i 设置一个最小共享额度 n_i （槽个数），调度器在进行调度时会确保每个池在有需要时可以获得它的最小共享额度。当一个池的最小共享额度没有完全使用时，空闲的槽可以被分配给其他的池。

Fair Scheduler 将作业（jobs）用池（pools）来管理，并将整个集群的资源按照池来进行划分。默认情况下，每一个用户会对应一个单独的池，所以每个用户可以分配到等量的集群资源份额而不论它提交了多少作业。同时还可以根据用户所属的组或者其他 jobconf 中的属性来决定如何来对池之间的资源进行划分。在每一个池内部，每一个作业既可以公平共享（fair share）的方式平分作业池的资源，也可以使用 FIFO（first-in-first-out）的方式来调度，或者可以提供一定的优先权进行调度。具体实施用户可根据作业的重要程度和实际的计算需求合理的分配资源^[33]。

上节提出除了提供根据 fair share 进行调度的方式以外，Fair Scheduler 还对每个作业池提供了一个“可以确保该池一定能满足的资源数”的一个值，这个值能够确保：即使在集群非常繁忙，资源非常紧张的情况下，至少能够确保给这个池一定份额的集群资源，也就是计算槽位；槽位多少，就是这个值所定义的量。当一个池中运行了一些作业的时候，这个池中的所有作业加起来“至少能够确保能够获得这个值定义数值的集群槽位”。但是当一个池中的所有作业加起来也用不了这么多槽位的时候，多余的槽位仍然是可以被其他的池使用的。这样可以达到整个集群资源的合理利用，减少集群计算资源的浪费。

公平调度为两级调度。第一级，在池间分配作业槽，其中，多用户情况下把每个用户组织成一个池，第二级，每个池（用户）在自己的作业间分配槽。在第一级采用有最小共享额度的公平调度方法，在池（用户）级别分配计算槽。在每个池（用户）内部每个用户可以使用不同的调度策略如 FIFO、HOD、公平调度方法等来合理的分配自己所拥有的槽。

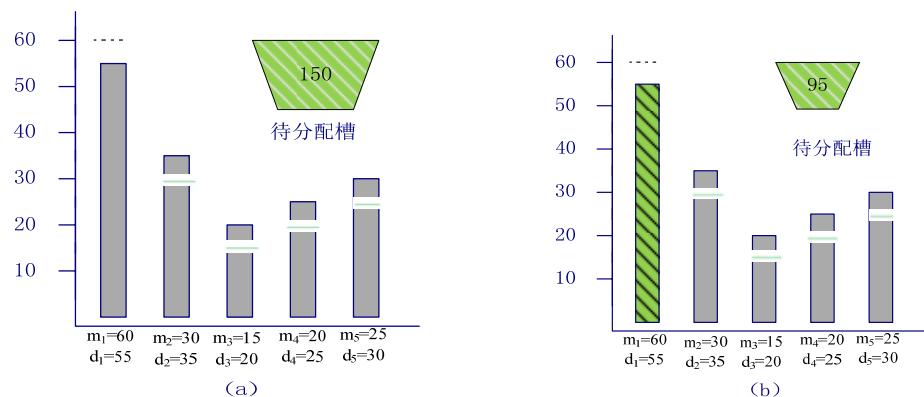
3.3 公平调度方法分析

公平调度方法是现阶段 Hadoop 中使用的最广泛的调度方法，下面将针对公平调度方法从以下几个方面进行分析研究，公平调度的槽分配及再分配算法、调度过程中的数据本地性问题^{[34][35]}。

3.3.1 槽分配算法

用户级的槽分配算法采用最小共享额度的公平调度。具体实施用下面例子进行说明。我们在 5 个池之间分配 150 个槽。其最小共享额度 (m_i) 和要求额度(d_i) 分别为：($m_1=60, d_1=55$)，($m_2=30, d_2=35$)，($m_3=15, d_3=20$)，($m_4=20, d_4=25$)，($m_5=25, d_5=30$)。槽分配分为三个阶段；第一阶段分配槽给那些最小共享额度大于要求额度的池。第二阶段，分配给每个池所需要的最小共享额度。第三阶段，分配剩余的槽给拥有最少槽的池。上面各池最后的槽分配结果如下：池 1 为 55 个槽，池 2 为 30 个槽，池 3 为 18 个槽，池 4 为 22 个槽，池 5 为 25 个槽。

槽分配实例如下图 3.1 所示，其中 (a) 给出最小共享额度和要求额度值，(b) 分配槽给最小共享额度大于要求额度值的用户池，(c) 给余下的池分配其最小额度的槽，(d) 把剩余的槽分配给拥有最少槽的池。



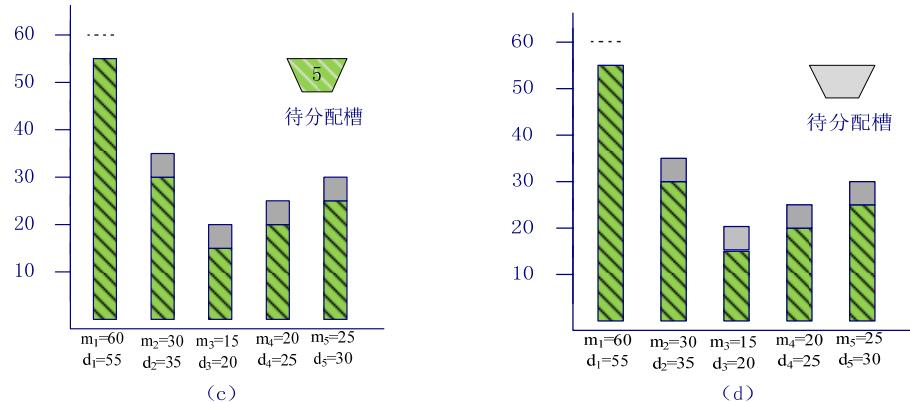


图 3.1 槽分配实例

3.3.2 槽的再分配

公平调度主要目标是保证每个提交作业的用户都能获得一定份额的槽运行作业。所以当有的新的用户到来时，采取何种方式分配槽给新的用户非常关键。公平调度方法采用任务杀除和任务等待相结合的方式来为新用户分配任务槽。

一般情况下，MapReduce 作业都是很短的（map 任务为 15-30s，reduce 任务为几分钟之内），一个作业可以很快的得到它的共享份额，等待时间较短。但是当出现生产性作业或者出现非并行化的计算时，可能会导致作业很长时间内得不到响应。为此，公平调度采用了两个时钟来定义槽的分配过程， T_{min} 用以保证最小共享额度， T_{fair} 用以保证公平共享额度，其中 $T_{min} < T_{fair}$ 。新用户将首先在 T_{min} 时间内等待其他用户任务完成后释放槽，如果一个新用户的运行作业在时间 T_{min} 期限内没有得到其最小共享额度，公平调度器将杀死其他池内的任务，并把槽分配给新作业。再次新用户在 T_{fair} 时间内继续等待其他用户任务完成后释放槽，如果新用户的运行作业在时间 T_{fair} 期限内未得到其公平共享额度，公平调度器将杀死更多的任务来满足新作业的需求。在选择杀死的任务时，默认选择最近开始运行的任务，以最小化被浪费的计算。抢占不会导致被抢占的作业失败，因为 Hadoop 作业容忍任务丢失。只是会延长作业的运行时间。降低整个系统的吞吐率。但却可以达到更好的公平性^[26]。

3.3.3 数据本地性

MapReduce 计算模型设计初衷即使针对大规模数据的分布式并行处理，Hadoop 集群在实现 MapReduce 模型时的挑战也是如何把计算程序尽量的靠近数

据，最佳效果是保证每个 map 任务程序都能被调度到其所要处理的数据所在的节点上。这也是在 2.3.4 中所讨论的存储本地化的问题。因为在一个大的集群中网络带宽是远远小于整个集群的节点磁盘总 IO 带宽。在作业调度时，首先，最好的情况是调度计算程序到数据所在节点运行，即具有节点本地性；其次，调度计算程序到本机架内节点运行，即具有机架内本地性；最次，调度计算程序到外机架节点上运行。因为要处理数据量的巨大，能否保证数据的本地性对集群的计算性能有着很大的影响^{[29][22]}。

公平调度方法对于多用户共享集群的公平性具有很好的效果。可以保证任何一个提交作业的用户在一定时间内得到响应。在集群实验中发现，当出现较多的小作业时，满足数据本地性在本节点只占 6%，在本机架内为 61%。而从 Facebook 等使用 Hadoop 集群的企业的报告得知其 58% 的作业都为此种类型，小作业多为即席式查询和报表服务。而 MapReduce 最重要的一点就是把计算放在距离所需数据较近的节点。增强数据本地性可以增加集群的吞吐率，提高集群的整体性能。图 3.2 为不同作业大小的本地性视图。在实验中总结出几个数据本地性相关的问题。分别因小作业和大作业的频繁提交引起。小作业引起的我们称之为头队列问题，大作业引起的我们称之为槽粘滞问题。下面会进行专门论述。

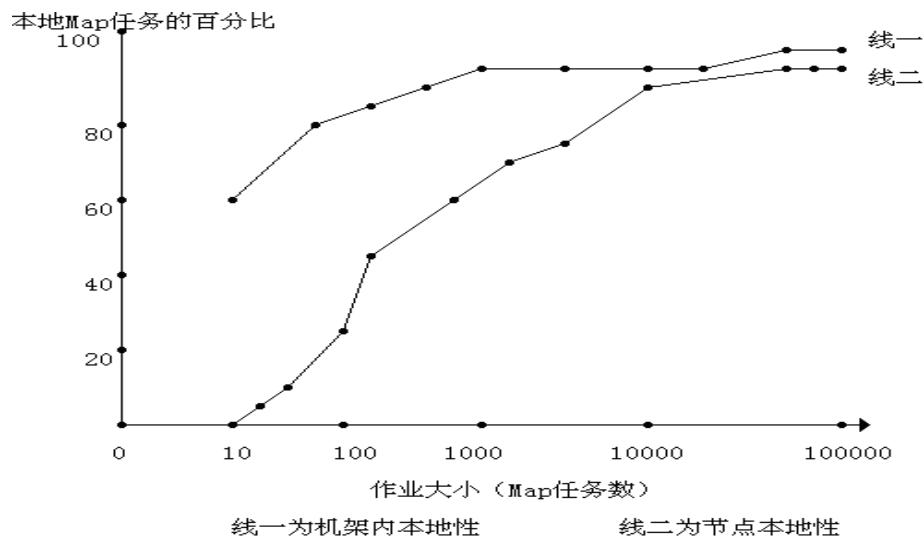
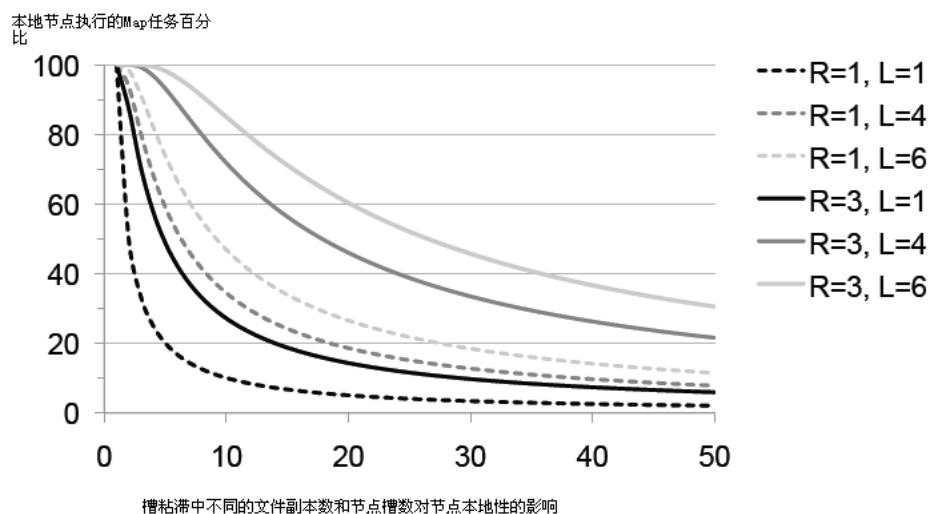


图 3.2 数据本地性和作业大小关系图

头队列问题发生在小作业调度时，小作业输入数据规模小，在大规模集群上其数据分布的节点占整个集群节点的比例较小。当一个小作业按照最小共享额度公平调度算法被调度到调度队列的头部时，无论哪个节点有一个空闲槽都要求小作业提交一个任务去运行。从而造成小任务的数据本地性较差。如果集群 5% 的节点存有小作业的数据其数据本地性的概率就为 5%。较差的数据本地性则要求每次执行 map 任务时都需要从其他节点传输数据，消耗了系统网络带宽，增加了任务

的执行时间。

槽粘滞问题由大作业长期驻守在集群中引起。在采用公平调度算法时，会出现同一个槽循环分配给同一个作业的趋势。如在一个有 1000 个节点的集群中运行着 100 个作业，每个作业运行 10 个任务。整个集群的 1000 个节点全部分配给这 100 个作业。作业 1 在节点 A 上完成了一个任务，节点 A 发一个心跳信号要求一个新的任务。而此时，其他作业都在运行 10 个任务。于是，节点 A 上的空闲槽再次分配给作业 1。这就造成作业总在相同的槽上运行任务，无法保证较好的数据本地性。影响系统的吞吐率。在数据副本数为 R，每个节点槽数为 L，作业数据占集群百分比为 f 的环境下，该作业能够达到的最大的节点数据本地性为 $1-(1-f)RL$ 。针对不同的 R 和 L 绘出的节点数据本地性大小见图 3.3。



槽粘滞中不同的文件副本数和节点槽数对节点本地性的影响

图 3.3 不同文件副本数和节点槽数下槽粘滞影响图

为了实现多用户共享的 MapReduce 集群，有两个方面需要考虑：公平性和效率。公平性可以借鉴公平调度算法的部分思想来实现；高效率则要求集群有高的吞吐率、作业有很好的响应时间、集群硬件得到充分的利用。为此，本文在公平调度方法的基础上提出了基于时间的等待调度方法。

3.4 本章小结

本章针对已有的基于 Hadoop 集群的作业调度方法进行了详细的介绍，并对其各自适应的应用场景和不足做了分析。调度方法不断改进的过程也是用户对于 Hadoop 集群提出新的要求的过程。面对更加复杂的多用户多作业的调度和应用请求，对集群的调度也提出了新的挑战。

详细分析了公平调度方法的槽分配策略，通过针对多用户多任务的槽分配过程研究，发现了槽分配策略存在的数据本地性问题。并给出多用户共享 MapReduce 集群所需要考虑的两个原则：公平性和效率。在现有的调度方法的基础上，结合多用户调度需求，将在下一章给出自己的调度策略。

第四章 基于时间的等待调度方法

上一章中分析了公平调度方法在作业调度过程中所存在的一些问题，主要是由于遵守一个严格的队列顺序造成一些任务被调度到不满足数据本地性的节点上，从而影响了系统效率。同时严格的队列顺序也造成了一些紧急作业的无法得到优先的执行。在这里提出了一种基于时间的等待调度来尝试解决上述问题。通过打破严格的队列顺序，使任务的执行有更多的选择性，从而改善数据本地性问题。

4.1 基于时间的等待调度方法

4.1.1 调度方法描述

本文所讲的作业调度是 Hadoop 集群的作业调度，而不是通常意义上讲的操作系统内部的作业调度。Hadoop 集群的作业调度是指对作业的执行顺序和集群的计算资源的分配进行控制。系统根据特定的调度方法以及用户对作业的要求选择计算资源来完成用户的作业。调度方法的好坏决定了集群的计算能力能否得到充分的利用，以及用户是否能够迅速的得到计算结果。作业调度过程一般如下：1) 把用户提交的作业放入系统的一个作业调度队列。2) 系统根据调度方法从调度队列中选择一个作业调度执行。3) 把选出要执行的作业分发到集群的计算节点上。4) 被分发的作业的各个任务在分配的节点上执行。5) 系统可以根据调度方法对第 3) 步中给出的决定做出修正，使改作业获得一些新的计算资源或者放弃一些已得的计算资源。作业调度的过程就是按照一定的调度方法从集群系统的用户提交的作业队列中选出一些作业，并分配给它们需要的计算资源，使它们可以在集群系统上执行。

基于时间的等待调度通过设置一个等待时间使队列头部的 map 任务可以暂缓执行以等待满足其数据本地性的节点空闲，对于公平调度方法中出现的头队列和槽粘滞问题是一个好的解决办法。以最大化减少任务在不满足数据本地性的节点上执行造成的网络带宽损耗和作业完成时间的延长。主要有两个方面，一是如何在多用户间进行槽分配，二是在增加新用户时的槽分配^{[34][36][37]}。

在分配槽时，仍然使用两级策略：顶级，对每个提交作业的用户设置权值，其权值代表其可分得的资源的比例大小。把所占槽低于最小共享份额的用户按照低于最小共享份额的距离由小至大顺序放在调度列表的后面，对已经获得最小共享份额的用户池按照（现有份额/权值）由小至大放在调度列表的后面。在有节点释放空闲槽时，在调度队列头部的用户池的任务（task）不满足节点本地性时，允

许任务在 T_1 时间内等待满足节点本地性的节点释放槽；允许任务在 T_2 时间内等待满足机架本地性的节点释放槽；其中， $T_2 > T_1$ 。等待时间可以根据集群资源的大小和作业大小来设置。算法如下：

```

初始化用户作业 j 变量：
j.level=0,j.wait=0,j.skip=false.
if 节点 n 发送一个心跳 then
    对每一个 j.skip=true 的作业，增加 j.wait 的值；其他作业 j.skip=false
    if 节点 n 有一个空闲 map 槽 then
        对作业进行队列排队，排队策略为两级策略
        for 队列头部作业 j do
            if j 有一个在节点 n 上的节点本地性任务 t then
                j.wait=0,j.level=0
                调度 t 给 n
            else if j 有一个在节点 n 上的机架内本地性任务 t and (j.level >=1 or
j.wait>=T1) then
                j.wait=0, j.level =1
                调度 t 给 n
            else if j.level=2 or (j.level=1 and j.wait>=T2) or (j.level=0 and
j.wait>=T1+T2) then
                set j.wait=0 and j.level=2
                调度 t 给 n
            else j.skip=true
            end if
        end for
        end if
    end if

```

每个作业在 $level=0$ 时，只允许提交节点本地性任务。在等待了时间 T_1 后其 $level$ 变为 1 可以在本机架内提交任务。在等待了时间 T_2 后其 $level$ 变为 2 可以提交跨机架任务。

带时间的等待调度方法对提高集群的吞吐率和资源利用性能有很好的作用。在具体应用设置 T_1 、 T_2 为 10s 时，对数据本地性的提高也是显著的。Hadoop 集群中运行的 map 任务长度基本为 8-16s。每个节点有 4 个 map 槽，数据的副本数为 3。即使 map 长度为 60s，其在 10s 内至少有一个节点空闲的可能性为 $1-(1-1/6)^{12}=89.8\%$ 。

4.1.2 槽分配算法

基于时间的等待调度所采用的槽分配算法来自公平调度，同样采用基于最小

共享额度的分配策略。具体实例见 3.3.1。

4.1.3 槽的再分配

基于时间的等待调度是在保证每个提交作业的用户都能获得一定份额的槽运行作业的基础上，保证每个用户的作业执行时满足尽可能高的数据本地性。当有的新的用户或者新的作业到来时，在分配槽给新的用户或作业时，基于时间的等待调度方法通过设置两个时间来提高作业执行时的数据本地性。如 4.1.1 中所描述两个时间值为 T_1 和 T_2 。当一个作业已经到达了作业调度队列的头部，在时间 T_1 内，该作业只允许提交具有节点本地性的任务。当作业等待时间位于 T_1 和 T_2 之间时，允许提交具有机架内本地性的任务，即允许该作业的任务在其需要计算的数据所在节点的机架内的其他节点上执行。当作业等待时间大于 T_2 时，允许该作业的任务在集群的任一节点上执行。

4.1.4 调度方法比较

公平调度方法已在第三章给出了详细的介绍和分析。其主要思想就是在提交了作业的用户间平均分配计算槽，且当一个作业到达作业调度队列头部时即调度器执行。基于时间的等待调度方法，允许一个作业在到达了作业调度队列的头部时，如果集群释放的空闲槽所在的节点没有该作业执行所需的数据时可以等待一定的时间。作业等待的目的是为了提高任务执行的数据本地性，提高集群资源的利用率，减少单个作业的平均响应时间。

对比公平调度方法和基于时间的等待调度，会发现在公平调度时关注的是用户/作业的是否得到了其公平共享份额，不去考虑槽分配时的任务调度的数据本地性。假设有两个作业，其分到的共享额度都为 2，集群由 4 个任务节点组成，且 Job1 和 Job2 的运算数据分布在集群的不同节点上，每个作业的数据分布在两个节点上。与表中作业名称相同的颜色的节点代表该作业数据存储在该节点上。

表 4.1 公平调度方法下作业任务运行图

Job	共享额度	运行任务数
Job1	2	2
Job2	2	2

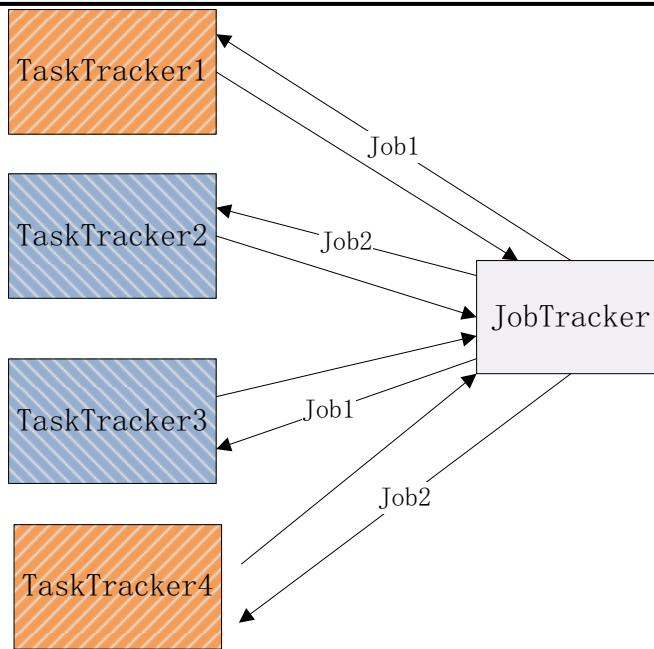


图 4.1 公平调度方法槽分配示例图

表 4.1 和图 4.1 为公平调度方法下作业槽分配实例，当有槽空闲时 JobTracker 会把空闲槽分配给集群作业调度队列头部的作业，而不会检查空闲槽是否满足该作业待执行任务的数据本地性。作业队列在初始分配时顺序为 Job1->Job2，TaskTracker1 被分配给 Job1 后，顺序为 Job2->Job1，TaskTracker2 被分配 Job2 后又变为最初的顺序。如此循环至两个作业都得到所得的共享额度后分配完毕。公平共享调度时，作业分配是严格按照队列顺序的。任务执行时数据本地性无法得到保证。从而增加了集群的数据 IO 开销，也增加了用户的平均作业响应时间。

表 4.2 中 (1) - (5) 为基于时间的等待调度槽分配过程图。(1) 为分配未开始时的作业运行任务状态；首先 TaskTracker1 空闲，因为 Job1 初始时位于集群的作业调度队列首部，且 TaskTracker1 上存有 Job1 要执行任务的数据即满足数据本地性，故在表 (2) 中 Job1 首先得到 TaskTracker1 的空闲槽。接下来 Job2 来到调度队列的首部，TaskTracker2 空闲且满足 Job2 要执行任务的数据本地性要求，故在 (3) 中 Job2 得到 TaskTracker2 的空闲槽。Job1 再次来到调度队列的首部，此时 TaskTracker3 出现空闲槽，但是 TaskTracker3 不满足 Job1 要执行任务的数据本地性，故 Job1 等待一定时间，把 TaskTracker3 的空闲槽让给队列中下一个位置的 Job2，而此时 TaskTracker3 满足 Job2 要执行任务的数据本地性，Job2 占有 TaskTracker3 运行任务。最后，TaskTracker4 空闲，而此时 Job1 因为等待仍旧位于调度队列的首部，TaskTracker4 满足 Job1 要执行任务的数据本地性，Job1 占有 TaskTracker4 运行任务。槽分配最终结果图 4.2 所示，基于时间的等待调度实现很好的数据本地性。

表 4.2 设置等待时间的作业调度顺序实例图

Job	共享额度	运行任务数
Job1	2	0
Job2	2	0

(1)

Job	共享额度	运行任务数
Job1	2	1
Job2	2	0

(2)

Job	共享额度	运行任务数
Job1	2	1
Job2	2	1

(3)

Job	共享额度	运行任务数
Job1	2	1
Job2	2	2

(4)

Job	共享额度	运行任务数
Job1	2	2
Job2	2	2

(5)

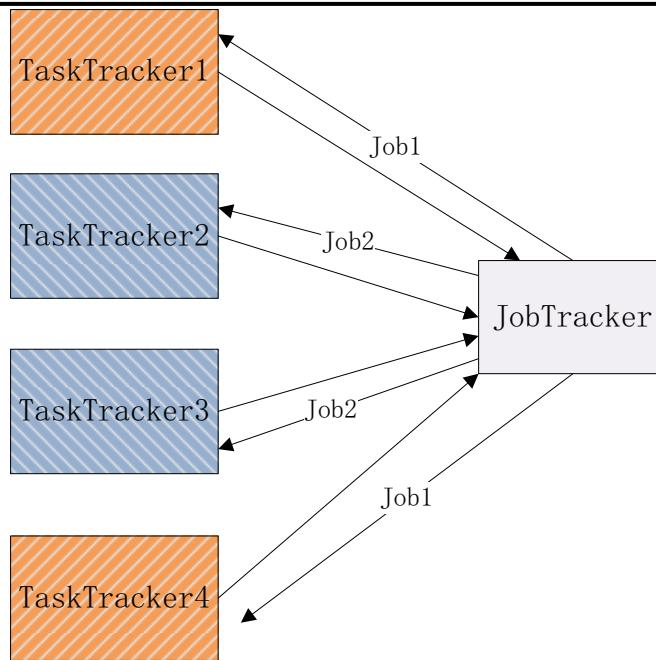


图 4.2 基于时间的等待调度槽分配示例图

4.2 调度方法分析

一个好的集群作业调度算法应该能够尽量提高系统资源的利用率、吞吐率、减少作业响应时间和等待时间。下面首先通过对等待调度算法的最小响应时间和最大吞吐率来分析其性能。对用户来讲就是为了获得最小的响应时间。对集群系统则是希望获得最大的吞吐量，提高集群的单位时间的计算能力。

4.2.1 响应时间分析

响应时间就是作业完成结束的时间减去作业递交到系统的时间。本文使用一个简单的数学模型在讨论在只为了改善作业响应时间时基于时间的等待调度是否值得。在此，假设一个 map 任务在非本地节点运行比本地节点多耗费 T s。把 Job tracker 上作业请求的到达近似为一个泊松分布，每次出现请求节点满足任务数据本地性的几率为 t s 一个。假设提交节点数据本地性任务的等待时间为 w 。那么，基于时间的等待调度的对响应时间影响的预期增益表示如下：

$$R \text{ (增益)} = (1 - e^{-w/t}) (T - t)$$

因为 $(1 - e^{-w/t})$ 永远为正值，故要想获得正的增益只有满足 $T > t$ 。如果 $T < t$ ，要么是 T 较小（运行的任务主要工作为计算而不是数据传送），要么 t 较大（运行的作业只有很少的 map 任务待调度或者集群正在运行一批长任务）。在调度时，对于偏向计算的作业等待调度会应影响其计算性能，而对于数据传送量较大的作

业等待调度性能会有很大的提高。在实际集群中。整体性能实验见第六部分。

4.2.2 吞吐率分析

吞吐率就是集群系统在单位时间内所处理的作业数。吞吐率越大说明单位时间内处理的作业越多。在有很多作业运行且各节点之间负载均衡时，当一个无法作业提交本地任务时调度队列后面将会有符合数据本地性的作业等待调度，故等待调度对系统的吞吐率的提高是有利的。而 HDFS 在进行数据的存储时是考虑了数据在整个集群分布的均衡性的。但不排除在一些情况下会出现多个调度队列中的作业都等待在同一个节点上执行任务。如果这多个待执行任务在性能特性上相似（如都为计算型任务），那么等待调度对吞吐率几乎没有影响。此时，可以通过设置较小的等待时间来调度其他任务到其他节点上执行。从而减少任务间对热点节点的竞争。为了进一步提高系统的吞吐率，可以根据任务的性能特性如计算性任务或 IO 型任务进行不同的调度策略，面对计算型任务可以调度它到本机架内的节点上执行，IO 型任务可以调度到具有节点本地性的节点上执行。同样数据大小的输入数据 IO 型任务需要花费更多的时间去读取数据，需要较高的数据带宽。而计算型任务的主要时间花费在计算上从而可以在边计算边进行数据的读取，对数据带宽要求相对较低。算法设计时可以对计算型任务设置比计算型任务相对小的 T1 和 T2 时间。实验结果显示通过此种方法可以提高 12% 的吞吐率。

4.2.3 公平性分析

公平性是指多个用户在提交的作业在集群上能否得到公平的处理，也就是能否公平的得到集群的计算资源。因为使用集群的用户是多个，且每个用户提交的作业量不同，公平调度方法的公平共享是基于用户而不是作业。保证的是用户级别的公平。作业多的用户可以分享更多的集群资源。本文提出的基于时间的等待调度也借鉴了这种思想，在分配资源时通过给用户设置权值来实现用户级的公平共享集群资源。在调度前会根据每个用户/作业的共享额度和时间进行排队，决定集群调度的队列顺序。

4.2.4 用户体验分析

作业调度是在集群系统内部进行资源的分配和任务调度，从方法上来讲对用户是完全透明的，但具体的实施则由系统负责。用户要处理的作业类型，由纯粹的批量作业到多用户共享集群情况下的批量作业、即时查询等短交互作业共存。增加了集群数据的利用率的同时，也增加了作业调度的难度。如何使得作业响应

时间能为用户接受变得愈发的重要。使用本文的方法进行作业调度，集群整体的吞吐量和作业响应时间都有显著的改善，但对于某一用户来讲有可能会出现响应时间有一定的延长。本文的方法消除了作业长时间得不到响应的情况，总体上提高了系统的公平性。

4.3 本章小结

本章主要介绍了基于时间的等待调度方法，并从响应时间、吞吐率、公平性和用户体验等角度对算法进行了简单分析。分析结果表明，与原系统相比，本文提出的方法在提高系统吞吐率与公平性上有较为明显的优势。下一章将给出详细的设计与实现。

第五章 系统的设计与实现

本章对第四章中提出的调度方法进行了较为详细的设计与实现。方法追求的设计指标就是要显著的提高作业调度时任务执行的数据本地性，缩小作业的平均响应时间，提高集群的吞吐率。

5.1 提出背景

MapReduce 集群得到越来越广泛的应用，除了著名的搜索引擎 Google 外，Yahoo!、Facebook 和 Amazon 等大的互联网企业也已经开始大规模的应用开源的 Hadoop 系统来使用 MapReduce 计算模型^{[38][32][5]}。在国内，淘宝、百度、腾讯等互联网企业亦开始涉足云计算并采用 MapReduce 计算模型来完成计算任务，并已经开始了对 Hadoop 的应用和研究^[26]。随着 MapReduce 模型得到更多的应用，基于 MapReduce 的集群的作业调度的实现变得愈发的重要。MapReduce 只是一种编程的思想，而在实现时如何更好的体现其优越性，作业的调度方法起着重要的作用。一个好的作业调度方法既可以很好的体现并行计算的优点，亦可以更加充分的利用集群的硬件资源，从而达到并发高效的计算。

在 Hadoop 集群中，上文已经分析了其多个作业调度方法。但随着更多的用户共享集群，提交集群处理的作业类型也多种多样，有批处理作业、短交互式作业、生产性作业等。多用户的共享和作业类型的多样化给 Hadoop 集群的调度策略提出了新的要求和考验。给出的新的调度策略既要解决现有的多用户共享条件下的用户公平性和集群系统的利用率方面的问题，还要避免因新的调度方法而造成系统管理的开销过大，过大的管理开销同样将损害集群的性能。本文针对现有的公平调度方法在数据本地性方面存在不足，从而影响整个集群吞吐率和性能发挥的问题。结合现有的应用要求和简单高效的原则提出了一种基于时间的等待调度方法。

5.2 设计原则和要求

在设计基于时间的等待调度方法时，其主要出发点：一是易用性，使其在 Hadoop 易于实现和使用。二是高效，对系统的性能有所改善，针对存在的问题给出简单有效的解决方法，其调度性能要优于以往的调度方法。三是简单，系统开销较小，如果一个调度方法可以实现较好的效果而过于复杂和庞大的话，其实现起来将会耗费较多的系统开销，影响系统的性能的提高。四是减少用户作业的响应时间，改善用户的使用体验，降低用户的等待时间。五是提高集群系统的资源利用率和作业吞吐率，这也是本文方法设计的一个很重要的原则^{[36][37]}。

对于我们的方法，需要满足两个要求，首先，保证用户共享集群的公平性，保证所有向集群提交了任务的用户都可以得到一定的计算资源且在一定的时间内得到响应。其次，利于整个集群性能的提高，因为是多用户共享，整个集群系能得到提高对所有用户来讲就是减少了其作业的响应时间，提高了集群的吞吐率。

针对公平性本文借鉴了公平调度方法的一些原则，给每个用户设置了最小共享份额，保证每个用户可以得到一定的集群资源。对于数据本地性问题，通过设置等待时间来解决调度过程中产生的头队列问题和槽粘滞问题，提高执行任务的节点的数据本地性。同时，为了应对紧急用户的作业请求，提出通过设置时间的方式来实现对紧急用户作业的及时处理。

5.3 基于时间的等待调度方法的设计与实现

第四章中已经给出了调度方法的伪代码和槽分配策略，下面给出方法的具体设计。

5.3.1 数据定义

定义 1: T1

T1 为 map 任务调度到具有数据本地性的节点的等待时间。在该时间内位于整个系统的调度队列头部的 map 任务在不满足数据本地性时等待，超过该时间可调度到本机架内的节点上执行。

定义 2: T2

$T2 > T1$ ，在 $T1$ 到 $T2$ 时间内，map 任务可以在本机架内执行，超过 $T2$ 则可在跨机架节点上执行。

定义 3: priority

priority 为优先级设置，当一个用户获得一定的时间时其所属的任务将被设置为该优先级，将优先获得资源的分配权。

定义 4: level

level 的值为 map 任务范围限定值， $level=0$ 时，只允许 map 任务在具有数据本地性的节点上执行。 $level=1$ 时，允许 map 任务在具有数据本地性的本机架内的节点上执行。 $level=2$ 时，允许 map 任务跨机架执行，即在集群的任何一个节点上都可执行以防止因等待而造成任务长期得不到执行。

定义 5: wait

wait 为一个系统调度队列头部的 map 任务等待时间进行计时，并在资源调度时通过与 $T1$ 和 $T2$ 的比较来限定任务的调度权限。

定义 6: skip

`skip` 的初始值都为 `false`, 当头队列的一个 `map` 任务没有得到执行时其 `skip` 值设为 `true`。对于 `skip` 值为 `true` 的任务的增加其 `wait` 值。作为下次调度的依据。

5.3.2 方法流程描述

1) 主程序

输入: 用户的提交的作业队列;
 运行中的作业队列;
 系统节点队列;
 数据存储结构信息;
 当前正在调度的作业队列;
 空闲节点队列;

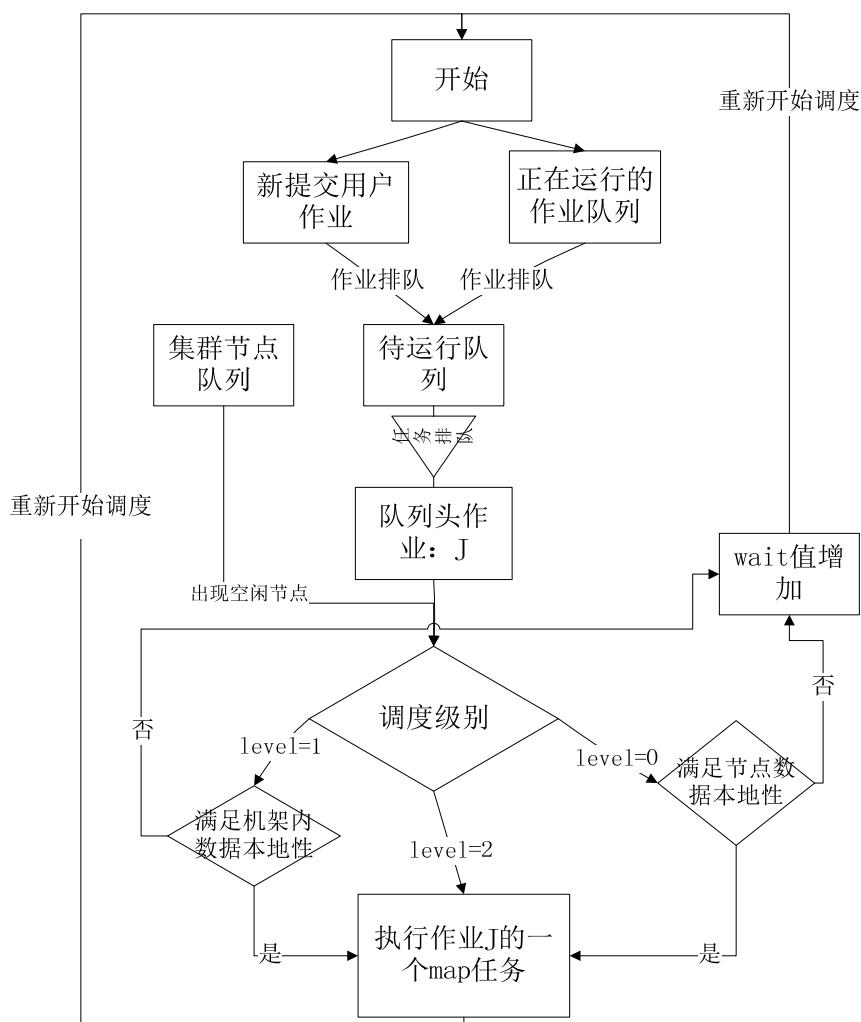


图 5.1 基于时间的等待调度主程序执行流程

基于时间的等待调度方法主程序执行流程, 流程图如 5.1 所示:

根据低于最小共享额度的值进行队列排序。

将队列头部的 map 任务作为下一个被调度执行的任务。

集群出现空闲的计算节点时，进行任务调度。

level=0 时，进行节点的数据本地性判断，满足调度执行。否则增加 wait 变量值，开始下一次调度。

level=1 时，进行机架内数据本地性判断，满足调度执行。否则增加 wait 变量值，开始下一次调度。

level=2 时，直接调度任务执行。

任务执行完后，同样开始下一次调度执行。

2) map 任务生命周期

Hadoop 系统在用户提交作业后，会根据用户提交的 MapReduce 程序对作业进行 map 任务和 reduce 任务的分解。产生一系列任务来完成用户运算任务。当一个 map 任务调度到集群的队列的头部时，其整个执行过程才正式开始。如下图 5.2 所示：

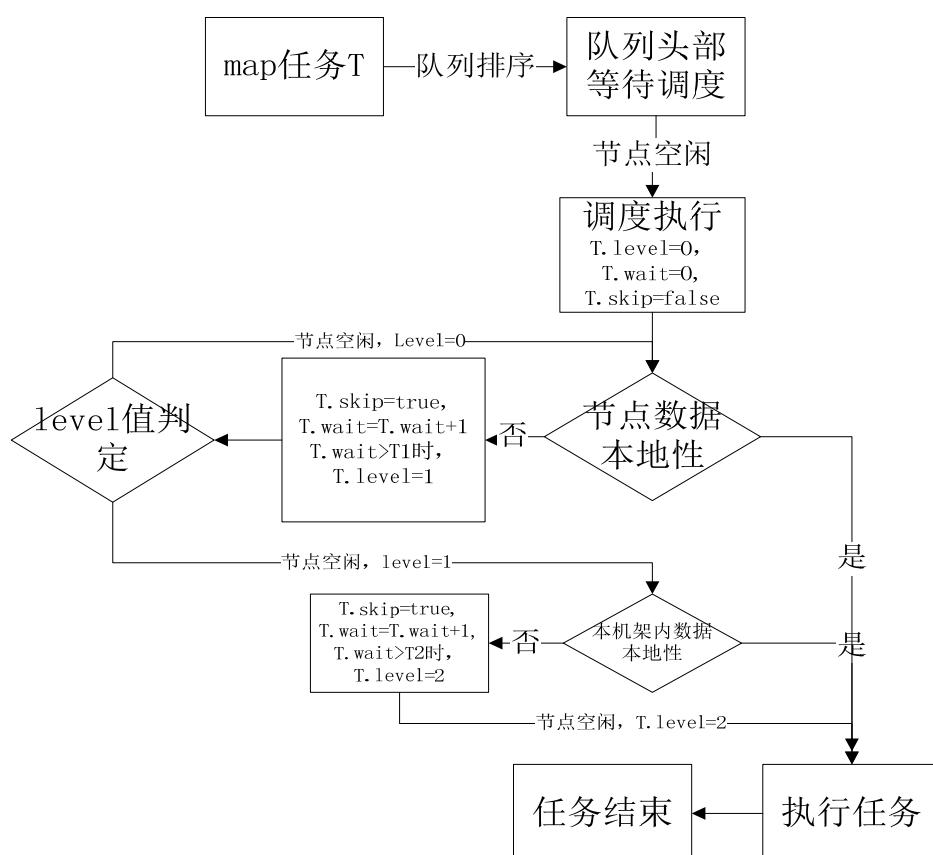


图 5.2 map 任务执行流程

如上图所示，在进行作业调度时资源分配对象针对的是用户或者作业，每一个作业包含有很多的 map 和 reduce 任务，当资源分配对象是用户时，该用户作业的一个 map 任务满足调度方法中的条件时会得到该计算资源开始执行运算任务。系统会根据该 map 任务的执行情况修改 map 任务所属的作业和用户的用于作业调

度的变量。该 map 任务执行完后会产生中间的运算结果，并保存在本地的磁盘中，为下一步的 reduce 任务提供数据输入。

5.3.3 作业调度流程

Hadoop 中的分布式并行计算实现方式为，Hadoop 中有一个作为主控的 JobTracker，用于调度和管理其它的 TaskTracker。JobTracker 可以运行于集群中任一台计算机上。TaskTracker 负责执行任务，必须运行于 DataNode 上，即 DataNode 既是数据存储结点，也是计算结点。JobTracker 将 Map 任务和 Reduce 任务分发给空闲的 TaskTracker，让这些任务并行运行，并负责监控任务的运行情况。如果某一个 TaskTracker 出现故障，JobTracker 会将其负责的任务转交给另一个空闲的 TaskTracker 重新运行。

Hadoop 中 MapReduce 模型的调度由 JobTracker 负责。整个集群中由一个 JobTracker 控制若干个 TaskTracker，JobTracker 主要负责任务的分配和节点的管理控制。TaskTracker 负责具体任务的执行，并向 JobTracker 报告自己所处的状态，接受其管理调度。JobTracker 和 TaskTracker 之间通过心跳机制通信。

（1）JobTracker 分配任务

当有用户提交作业后，JobTracker 会根据作业队列的顺序和调度方法分配计算资源给作业。TaskTracker 会通过心跳机制定期向 JobTracker 发送状态信息报告，报告自身目前是否空闲等；如若空闲 JobTracker 则会向该 TaskTracker 节点分配计算任务，并接受该节点的完成情况报告。如果运算失败或节点失效，系统会重启该作业，进行重新运算。如果节点正常完成该任务的运算，则在本机磁盘保存运算结果。

（2）TaskTracker 执行任务

TaskTracker 节点又称 slave 节点，顾名思义是被控制节点。接受 JobTracker 的指令进行任务的运算。并通过心跳机制定期向 JobTracker 节点更新自己的状态信息。另外还负责原始输入数据和中间运算结果的存储和传递。

若 TaskTracker 节点失效，则在此节点上正在执行和已经完成的任务都需要重新执行。即设为空闲重新分配给其他 TaskTracker 节点执行。

（3）Map 阶段任务

当一个 map 任务被分配到执行节点执行时，系统会移动 map 计算程序到该节点。该 map 任务会对自己处理的输入数据块进行解析，解析出 key/value 对，并传给用户自定义的 map 函数产生中间结果 key/value 对，然后将中间结果 key/value 对写入本地磁盘并将其散布在有分割函数指定的 R 个区域中，最后将这些缓存对在局部磁盘的具体位置传回给 JobTracker。

如果一个接受到 map 任务的 TaskTracker 在一定时间后, JobTracker 还没有接收到中间 key/value 对在本地磁盘上的位置信息, 则将此 map 任务的状态重新设置为空闲, 并分配给其他的空闲的 TaskTracker 节点执行。

(4) Reduce 阶段任务

在一个 reduce 任务被分配到一个空闲的 TaskTracker 节点上执行时, JobTracker 会先将中间结果的 key/value 对在执行 map 任务的 TaskTracker 节点上局部磁盘上位置信息发送给 reduce 任务, reduce 任务采用远程过程调用机制从 map 任务节点的磁盘中读取数据。读取完所有的数据后, reduce 任务会按照中间 key 进行排序, reduce 任务遍历排好序的中间数据, 把特定的 key 和其所对应的中间 value 传给用户自定义的 reduce 函数, 最后将 reduce 函数的输出写到与之对应的最终输出文件中。最终文件结果存入 HDFS 文件系统中, 一般情况下对最终结果不会进行合并。分离的结果为作为 MapReduce 执行的输入数据时提供了方便。

同 map 任务一样, 在一定时间后如果 JobTracker 没有接收到执行 reduce 任务的节点传回的已完成的报告, 则将此 reduce 任务的状态重新设置为空闲, 并分配其他的空闲节点执行。

(5) 最终结果返回

当所有的 map 和 reduce 任务都完成后, 则说明一个 MapReduce 作业的完成。系统返回给用户最终结果。

5.3.4 任务执行状态图

本文的调度方法主要是针对作业的 map 任务在集群中执行时的公平和效率问题进行分析和研究。集群的性能在 MapReduce 编程模式下主要影响因素是执行 map 任务的节点的数据本地性满足情况, 如果能保证每个 map 任务都能有节点的数据本地性, 那么集群的整体性能将会有很大的提高和改进。下图 5.3 为一个 map 任务在 Hadoop 集群系统中的被调度和执行状态图。

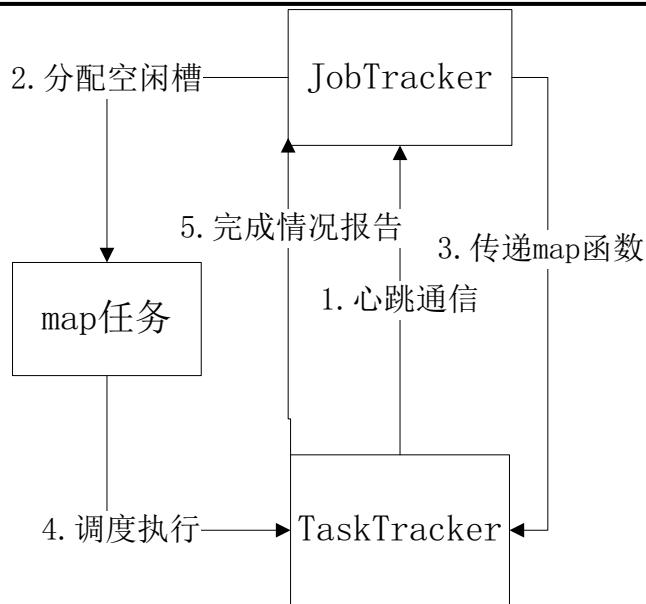


图 5.3 map 任务调度执行图

如上图所示，1 为当一个 TaskTracker 节点空闲时通过心跳机制向 JobTracker 报告空闲状态表示自己可用。2 为 JobTracker 根据集群作业队列情况把该空闲节点分配给一个作业的 map 任务，当然具体分配时根据基于时间的等待调度方法进行空闲 map 任务的选择。3 是 JobTracker 向空闲节点移动计算程序，该程序负责一块数据的处理。4 是 TaskTracker 节点得到计算程序和输入数据后开始进行运算，执行 map 任务。5 为 TaskTracker 节点向 JobTracker 节点汇报任务完成情况，正常执行完毕则标记该 map 任务完成。出现异常或长时间无完成情况报告则重新启动该 map 任务。

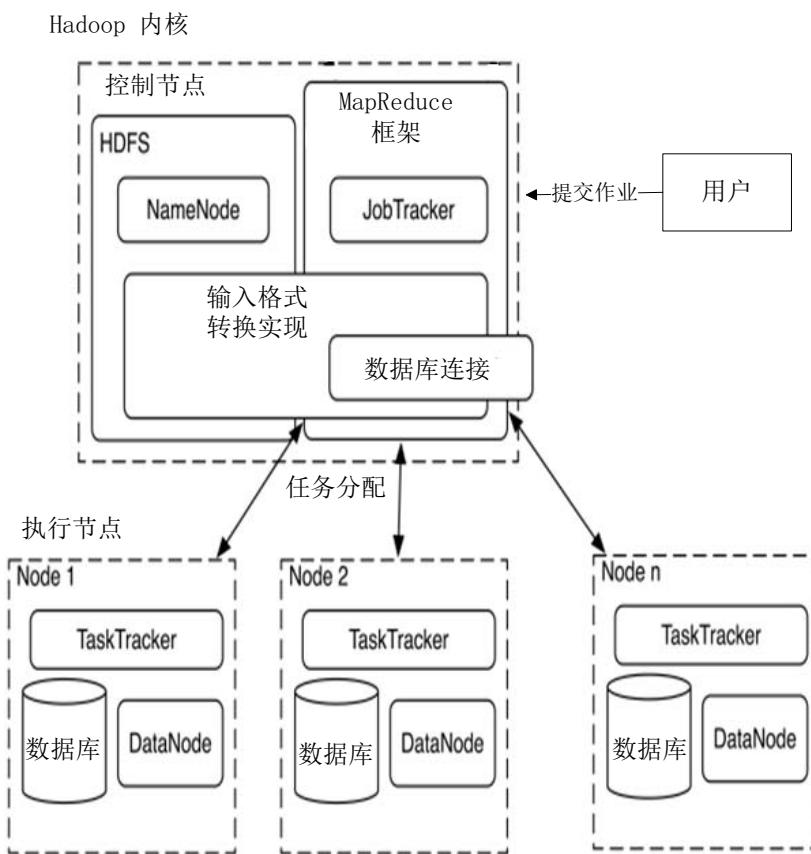
5.4 调度方法在 Hadoop 中的实现

5.4.1 Hadoop 集群

集群(Cluster)由多个成为节点的计算机组成，一般情况下这些节点通过网络连接起来作为一个系统来工作，这个系统为用户提供计算能力、数据存储和通信资源等。同时，集群中每个节点都以一个完整的系统，可以提供所有的单机服务，在集群中各节点相互协作，作为一个整体它提供了强大的性能和极高的可用性。在高可用性集群中，单个节点的故障并不会影响整个集群的工作，通过备份等技术高可用集群可以提供连续不间断的各种服务。最近几年集群系统发展的异常迅猛，随着多核处理器的出现，多核集群也在大量增长，已经成了当前集群系统的主要形式^{[39][40]}。

上文已有介绍 Hadoop 主要有两个部分组成 MapReduce 编程模型和 HDFS

(Hadoop 分布式文件系统)。在 Hadoop 集群中一个节点要担负两个方面的任务，一是数据的存储，二是集群的管理和数据的运算。从存储的观点出发，集群有两类节点：NameNode (名字节点) 和 DataNode (数据节点)。其详细介绍见第二章。从作业调度的角度出发，集群亦分为两类节点：JobTracker (控制节点) 和 TaskTracker (任务节点/执行节点)。详情见第二章。在实际的系统实现和安装时，会把运算节点和存储节点合二为一，一般情况下 NameNode 和 JobTracker 会安装在同一台物理机器上，DataNode 和 TaskTracker 作为集群的主要部分也会被安装在相同节点上而大量的散布于集群中。Hadoop 集群目前是保持一个主控节点，既一个 NameNode 和 JobTracker 节点。具体结构图见图 5.4.1。

图 5.4 Hadoop 集群结构图^[16]

如上图 5.4 所示，集群由一个主控节点负责 HDFS 和 MapReduce 执行的管理。其余节点负责数据存储和运算。Hadoop 集群是一个针对大规模数据的存储和计算平台。如何处理好数据存储和计算的关系决定着集群的性能的优劣。也就是本文所研究的数据本地性问题。数据存储在哪一个 DataNode 节点上，就由这个节点计算机执行这部分数据的计算，从而可以减少数据在网络上的传输，降低对网络带宽的需求。在 Hadoop 这样的基于集群的分布式并行系统中，计算结点可以很方便地扩充，因而它所能够提供的计算能力近乎是无限的，但由于数据需要在不同的

计算节点之间传输，所以网络带宽往往容易变成瓶颈，计算节点的数据本地性成为了最有效的一种节约网络带宽的手段，业界把这形容为“移动计算比移动数据更经济”。调度方法就是作为 Hadoop 系统的核心程序负责移动计算到相应的数据上来完成计算任务。

5.4.2 Hadoop MapReduce 并行化处理

一个作业进行任务分解，分解成 map 和 reduce 任务。数据进行分块处理，分解成一个个数据块，存放于 HDFS 中。任务在输入数据的数据块上通过并行执行完成计算。在实体世界中，并行化技术通常是通过在多个服务器之间分配入站请求的负载平衡器或内容交换机实现的。而在云计算世界里，并行化可以通过在多个虚拟机之间分配入站请求的负载平衡设备或内容交换机来实现。无论是上述哪种情况，应用程序都可设计为吸收补充资源来适应工作负荷尖峰^[41]。云计算架构的可扩展性则为计算性能的扩展提供了无限可能。

MapReduce 模型是基于任务的并行，把一个问题分解成可以并发执行的任务集合，然后通过底层的并行计算结构来完成计算。假设一种应用，它的输入是 n 组数据，分别对每组数据进行的操作并不一样，比如第一组要求先做乘法后做加法，第二组要求先做加法后做乘法，第三组……，这样每一组数据上的任务都不相同，让它们并行来做可以看作是典型的按任务划分。如图 5.5 示。用 MapReduce 模式来对这个问题进行描述，可以看到在这个问题中计算步只有一个，且不存在数据依赖。首先从数据划分方面来看，每一个数据集合对应着不同的操作，所以每一个数据集合作为一个单独的 map 输入 key/value 对中的 value。其次，从计算任务的分配角度来看，由于只有一个数据无关的计算步，所以在设计 MapReduce 的两个主要步骤 map 和 reduce 时，可以将计算任务放入其中任何一个步骤。最后，从数据结构的设计方面来看，一方面在 map 的输入数据中要包含数据集合中的数据，另一方面每个数据集合所要做的计算是不一样的，所以要有一个标识能够表示次数据集合要做什么计算，比如用 1 作为第一组数据的标识，当发现标识为 1 则对每个元素做先加后乘的运算，而且还要有加法参数 a 和乘法参数 b 的信息。由此看来在设计数据结构的时候由两部分信息要考虑，(1)数据集合的参数和标识信息，即数据的特征信息；(2)数据集合本身的数据，即原始数据。

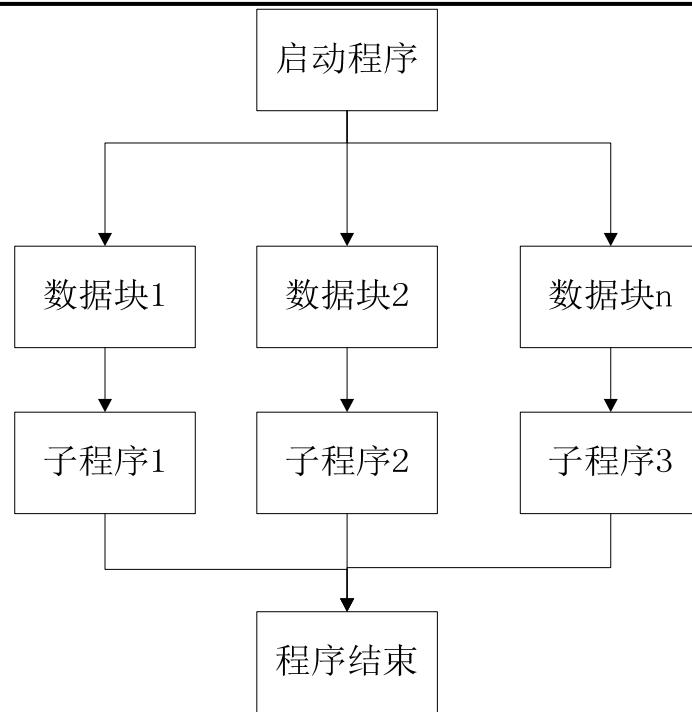


图 5.5 任务并行结构图

MapReduce 的并行能力主要有三个部分，一是子任务间并行，一个作业的各个 map 子任务并行执行，见图 5.6 所示；二是 map 和 reduce 任务间并行，一个作业的 map 和 reduce 任务同时执行，见图 5.7 所示。三是作业间并行当然有时也是用户间的并行，见图 5.8 所示。

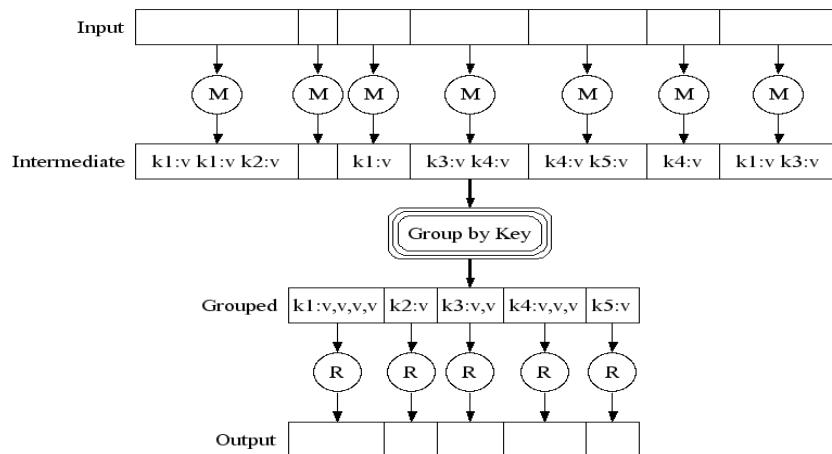


图 5.6 MapReduce 子任务间并行

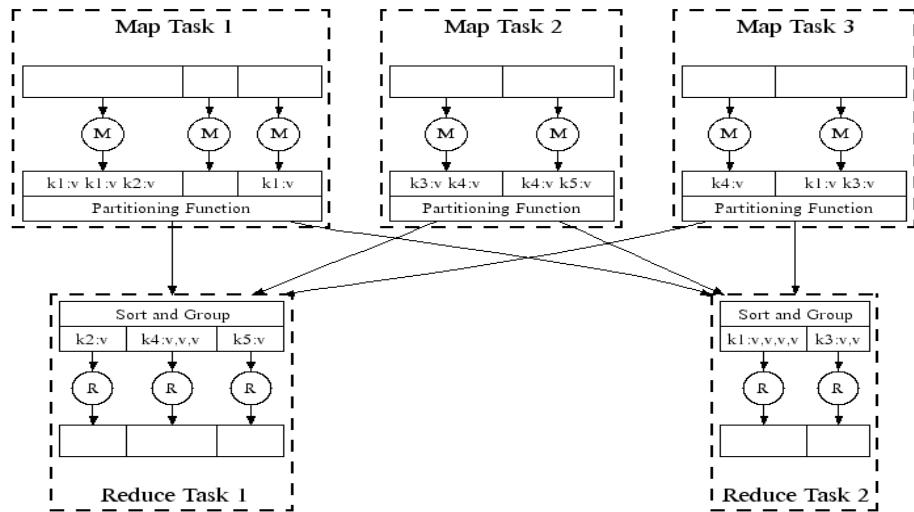


图 5.7 MapReduce 中 map 和 reduce 任务间并行

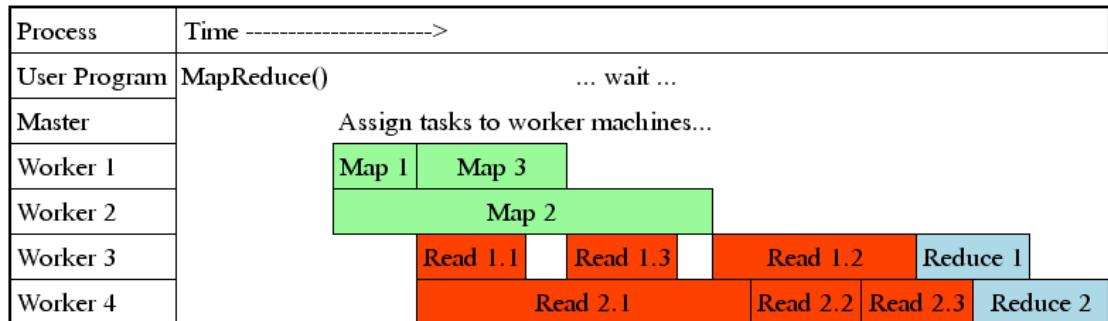


图 5.8 MapReduce 中作业间并行

MapReduce 模型只是给出了问题可以用并行化的方式解决的思想，具体的并行处理实现则需要用户通过自定义的函数去完成。而当给出了具体的用户自定义函数后，下面的工作就需要系统去完成各个并行任务的调度和运行，调度方法就是用来分配和调度用户自定义函数所分解出的各个子任务（map 和 reduce 任务）。

5.4.3 基于时间的等待调度在 Hadoop 系统中的使用

在 Hadoop 系统中调度器是作为插件的形式集成在 Hadoop 中的，本文给出的基于时间的等待调度方法经过编译打包可以产生一个基于时间的等待调度器，基于时间的等待调度器同样是一个用于 Hadoop 的插件式的 MapReduce 调度器，下面给出基于时间的等待调度器的生成过程。

首先，在 Eclipse 下把我们写的基于时间的等待调度方法的程序打包成 jar 文件，我们命名为 hadoop*myscheduler.jar；然后把 hadoop*myshceduler.jar 拷贝到 HADOOP_HOME/lib 目录下。也可以修改 HADOOP_CONF_DIR/hadoopenv.sh 中的 HADOOP_CLASSPATH，加入基于时间的等待调度器的 jar 包。

在加入了 jar 包以后，还需要在 Hadoop 的配置文件 HADOOP_CONF_DIR/mapredsite.xml 中设置下列属性让 Hadoop 使用基于时间的等待调度器：

```
<property>
<name>mapred.jobtracker.taskScheduler</name>
<value>org.apache.hadoop.mapred.MyScheduler</value>
</property>
```

在属性设置完成后，重启 Hadop 集群后可以通过 JobTracker 的 web 用户界面中的 <http://<jobtrackerURL>/scheduler> 检查基于时间的等待调度器是否正在运行。应该能看到一个"job scheduleradministration"页面。出现这个页面则说明调度器成功运行。

在 Hadoop 系统中，基于时间的等待调度器有两处配置文件，方法参数在 mapred-site.xml 中设置，以及一个单独的称为配额文件的 XML 文件，可以用来配置资源池、最小共享资源、运行作业限制和抢占超时时间、等待时间。配额文件在运行时会定期被重新加载。

在具体的调度器实现时，首先计算用户或作业的公平共享资源；再次，当有任务槽空闲时根据基于时间的等待调度方法进行分配资源。

5.5Hadoop 系统运行模式

Hadoop 在安装和配置时，有三种模式可供选择。分别是单机模式、伪分布式模式和分布式模式^{[42][43]}。基于时间的等待调度器在这三种模式中都可使用。

5.5.1 单机和伪分布式模式

默认情况下，Hadoop 被配置成以非分布式模式运行的一个独立 Java 进程。这对调试非常有帮助。也即单机运行模式^[47]。

下面的实例将已解压的 conf 目录拷贝作为输入，查找并显示匹配给定正则表达式的条目。输出写入到指定的 output 目录。

```
$ mkdir input
$ cp conf/*.xml input
$ bin/hadoop jar hadoop-*examples.jar grep input output 'dfs[a-z.]+' 
$ cat output/*
```

Hadoop 可以在单节点上以所谓的伪分布式模式运行，此时每一个 Hadoop 守护进程都作为一个独立的 Java 进程运行。需要配置文件 conf/hadoop-site.xml 如下所示：

```

<configuration>
  <property>
    <name>fs.default.name</name>
    <value>localhost:9000</value>
  </property>
  <property>
    <name>mapred.job.tracker</name>
    <value>localhost:9001</value>
  </property>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
</configuration>

```

接着进行免密码 ssh 登陆设置，格式化一个新的分布式文件系统，命令为：
\$ bin/hadoop namenode -format；最后启动 Hadoop 守护进程运行 Hadoop 系统进行数据的计算和存储。在伪分布式模式下，可以模拟完全分布式集群完成一些简单的实验，不具有实际的意义。

5.5.2 完全分布式模式

完全分布式模式是具有实际意义的 Hadoop 集群，其规模可以从几个节点的小集群扩展到几千个节点的超大集群。安装 Hadoop 集群通常要将安装软件解压到集群内的所有机器上。安装的运行环境为 Ubuntu Server、JDK1.6、Hadoop-0.20.1。这些软件需要安装集群的所有节点上。通常，集群里的一台机器被指定为 NameNode，另一台不同的机器被指定为 JobTracker。当然也可以指定一台机器同时作为 NameNode 和 JobTracker，这些机器是主控节点（masters）。余下的机器即作为 DataNode 也作为 TaskTracker。这些机器是执行节点（slaves）。我们的实验就是基于完全分布式模式进行的。

要配置 Hadoop 集群，需要设置 Hadoop 守护进程的运行环境和 Hadoop 守护进程的运行参数。Hadoop 守护进程指 NameNode/DataNode 和 JobTracker/TaskTracker。配置 Hadoop 守护进程的运行环境，管理员可在 conf/hadoop-env.sh 脚本内对 Hadoop 守护进程的运行环境做特别指定。至少，你得设定 JAVA_HOME 使之在每一远端节点上都被正确设置。管理员可以通过配置选项 HADOOP_*_OPTS 来分别配置各个守护进程。还需要配置集群系统的 Java 环境、HDFS 设置、配置 slaves 文件、配置 Hadoop core 文件和 DataNode 文件。详

情可参见 Apache Hadoop 文档。

在完全分布式模式搭建完后，就成了一一个分布式的存储和计算集群。使用 HDFS 对数据进行存储和管理，使用 MapReduce 模型进行数据的处理和运算。Map/Reduce 框架和 HDFS 是运行在一组相同的节点上的，也就是说，计算节点和存储节点通常在一起。这种配置允许 Hadoop 框架在那些已经存好数据的节点上高效地调度任务，从而使整个集群的网络带宽被非常高效地利用。

上文已有介绍 MapReduce 框架由一个单独的 master 节点 JobTracker 和每个集群节点一个 slave 节点 TaskTracker 共同组成。master 负责调度构成一个作业的所有任务，这些任务分布在不同的 slave 上，master 监控它们的执行，重新执行已经失败的任务。而 slave 仅负责执行由 master 指派的任务。

在 Hadoop 框架完成后，用户向 Hadoop 提交作业时，其提供的应用程序至少应该指明输入/输出的位置（路径），并通过实现合适的接口或抽象类提供 map 和 reduce 函数。再加上其他作业的参数，就构成了作业配置（job configuration）。然后，Hadoop 的 job client 提交作业(jar 包/可执行程序等)和配置信息给 JobTracker，后者负责分发这些软件和配置信息给 slave、调度任务并监控它们的执行，同时提供状态和诊断信息给 job client。

如果一个作业不需要 reduce 函数时，用户就不需要实现 Reducer 类，作业的 reduce 阶段当然也不会执行。Hadoop 系统将会对输入数据进行分片，在整个集群上调度 map 任务执行。如果需要 reduce 函数，Hadoop 系统将会对 map 任务产生的中间结果进行合并分类处理，reduce 任务将使用这些中间结果作为输入来完成计算。最后的输出结果将被存储到输出路径，作业的完成状态也将传递给用户。

5.6 本章小结

本章主要给出了基于时间的等待调度方法的设计原则和要求，以及其具体的实现。并结合 Hadoop 系统给出了方法在实际应用时的操作流程和 Hadoop 系统的运行方式。

第六章 实验验证分析

本章主要针对前文提出的方法进行实验验证。针对基于时间的等待调度方法对小作业与槽粘滞问题的影响，以及对 IO 型负载、计算型负载与混合型负载问题的影响都设计了相应的实验，制定了实验步骤与流程，并对各个实验结果进行了详细的分析。

6.1 实验环境

实验是在一个有 90 个节点的集群上进行的，其硬件配置信息为：节点为宝德“双子星” PR1760T 服务器，拥有四个处理器。机器上安装的操作系统为 Ubuntu Server^{[42][46]}。这 90 个节点被分布在 3 个机架上。在其上安装了 Hadoop 系统，Hadoop 版本为 Hadoop-0.20.1，使用 HDFS 进行数据的存储。其中一个节点设置为 Jobtracker 和 NameNode，其他的 89 个节点为 Tasktracker 和 DataNode。查询客户端使用 Hive^[44]，Hive 可以放在任一台安装有 Hadoop 的节点上，把 Hive 放在设置为 Jobtracker 的节点上^[45]。每个节点包含 4 个处理器，执行 MapReduce 任务时组织为 2 个 map 槽和 2 个 reduce 槽。

使用三种负载类型的作业去验证方法的性能，三种负载分别为计算型，IO 型，混合型。测试程序取自 Hive 官方给出的基准测试程序^{[44][43]}。这些测试程序包括：文本搜索，一个简单的过滤查询，一个聚合操作，一个可以转换为多阶段 MapReduce 的连接操作。将在三种调度方法上应用这些基准测试程序。分别为 FIFO，公平调度算法，基于时间的等待调度方法。等待时间 T1 为 4s，T2 为 6s。每种类型的作业作为一个用户提交到集群^[45]。

本文还针对于公平调度过程所产生的两个问题，槽粘滞问题和头队列问题分别作了实验分析。

6.2 基于时间的等待调度对小作业的影响实验

为了测试调度方法对于小作业负载的数据本地性和吞吐量的影响，产生了一个大的一些简单数据的集合，针对这些数据的作业都是不带 reduce 操作的扫描作业，每个作业都是读取输入的数据，输出约为其 1% 左右的结果。执行的作业分别为 3, 10 和 100 个 map 任务，每个作业块的大小为 64M，输入文件大小分别为 192M, 640M, 6400M。比较了公平调度和基于时间的等待调度的执行性能，其执行时间如图 6.1 所示，表 6.1 给出数据本地性效果。基于时间的等待调度方法对于有 3 个 map 任务的作业其吞吐量增加了 1.2 倍，10 个 map 任务的作业增加了 1.7 倍，100

个 map 任务的作业增加了 1.3 倍；达到的节点数据本地性至少为 76%，机架内数据本地性为 95%。

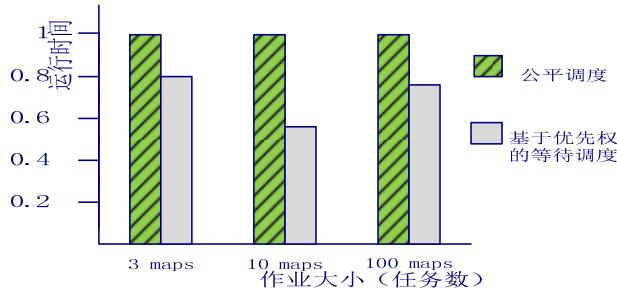


图 6.1 执行时间比较图

表 6.1 小作业负载的数据本地性

作业大小	节点/机架本地性 公平调度	节点/机架本地性 基于等待的调度
3 maps	3% / 52%	76% / 97%
10 maps	39% / 98%	99% / 99.8%
100 maps	85% / 99%	95% / 99%

对于执行时间的减少和数据本地性的提高，主要是因为提高了节点的数据本地性。把原来跨机架执行的任务提高到在机架内执行，机架内执行的任务提高到在满足数据本地性的节点上执行。

6.3 基于时间的等待调度对槽粘滞的影响实验

在上文已经介绍了在公平调度过程中，因为大作业长期执行造成的某个作业长期占有固定的槽，从而造成槽粘滞的现象。在这里构造了一个大的数据集，针对该数据集提交扫描作业，并记录下作业的完成时间和数据本地性的性能。如图 6.2 和图 6.3 所示。

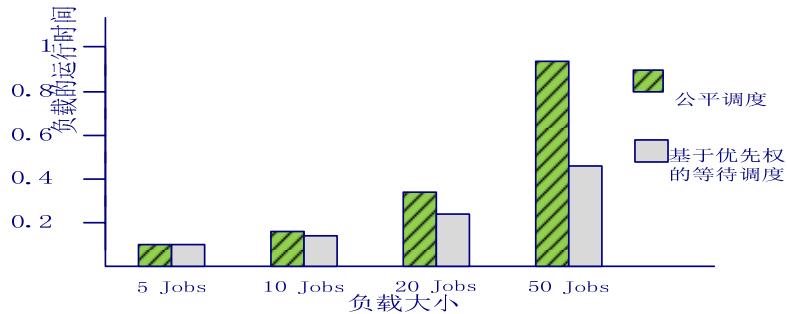


图 6.2 槽粘滞实验中负载完成时间

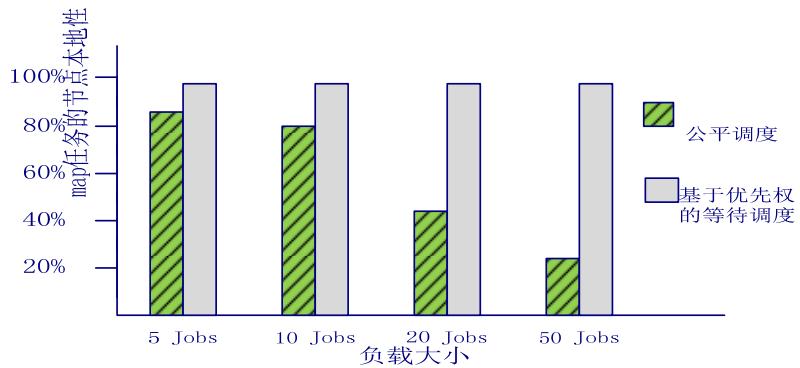


图 6.3 槽粘滞实验中节点数据本地性

基于时间的等待调度对系统的吞吐量有很大的改善，对于 10 作业的负载提高了 1.1 倍，20 个作业的负载提高了 1.7 倍，50 个作业的负载提高了近 2 倍。对于数据本地性也有大大的提高。

6.4 IO 型负载实验

选取 Hive 基准测试程序中的文本搜索进行 IO 型实验，因为文本搜索主要是扫描整个文本，搜索到匹配值输出。输出结果是非常小的，且集群绝大部分时间是在进行数据读取和扫描操作，主要受限于硬盘的 IO 速度^[43]。对不同的作业规模试验结果如下图示：

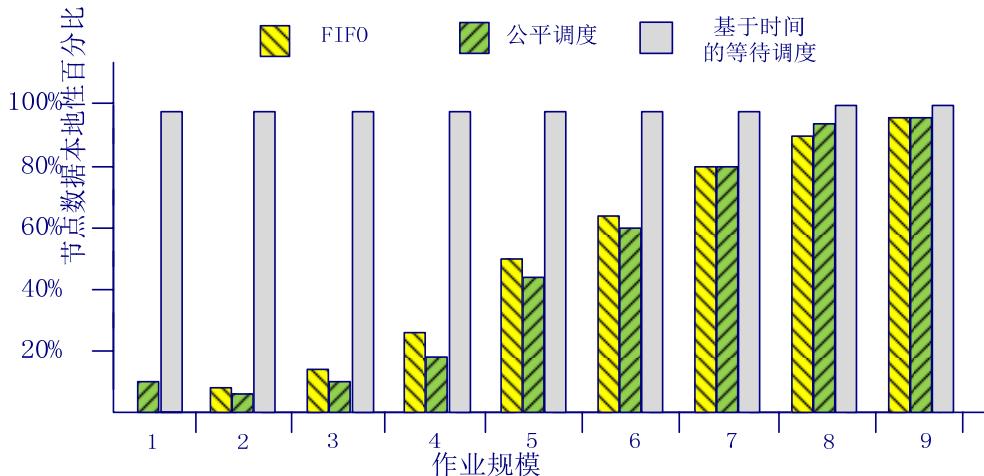


图 6.4 不同规模作业的数据本地性图

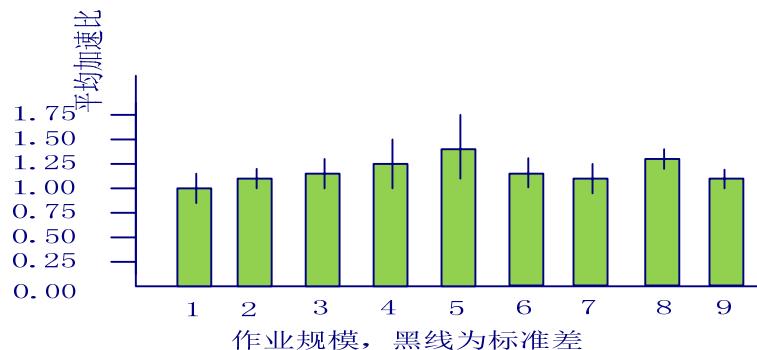


图 6.5 等待调度相对与公平调度的平均加速比，黑线为标准差

图 6.4 为三种调度算法在 9 中不同规模的 IO 型作业上节点本地性百分比图。等待调度在所有的规模作业的几乎都可达到 100% 的节点本地性。FIFO 和公平调度在小作业上节点本地性都很差。图 6.5 为等待调度相对于公平调度的平均加速比，等待调度对小作业的运行时间几乎没有影响，因为小作业 map 任务运行时间较短，且输入的数据较小，不具有节点本地性对整体时间没有大的影响。对中等规模的作业（100map 任务左右），等待调度的加速比还是很明显的。但当作业规模很大时，加速的效果又有回落。

6.5 计算型负载实验

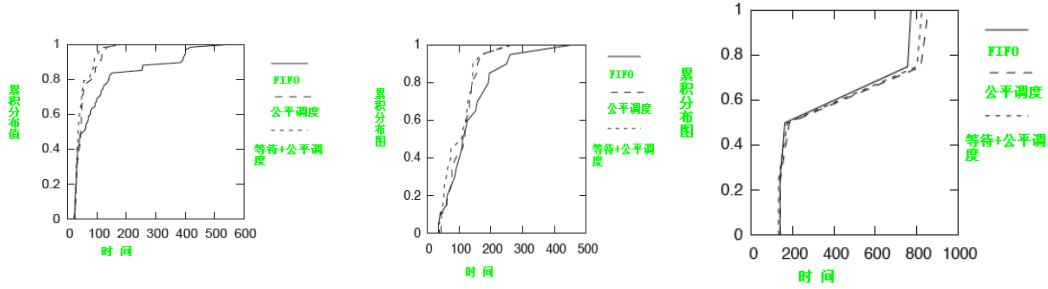
通过一个用户自定义的函数 UDF (user defined function)，该函数主要偏重于计算。该函数作用于文本搜索中每一个输入数据上，主要时间为计算时间。结果显示不同规模数据本地性于 6.2 中 IO 型作业相似，而等待调度相对于公平调度的平均加速比有所下降，但对中等规模的作业，加速比仍旧比较明显，只是略有下降。

6.6 混合负载实验

混合负载即计算型作业和 IO 型作业等多种作业同时存在。利用 Hive 给出的基准测试程序产生四种作业，IO 密集型的简单选择查询、计算密集型的聚合操作作业、复杂的 join 查询、文本搜索。根据作业的大小把实验负载分为 10 个级别，级别越高代表这作业规模越大，其 map 任务数越多。表 6.2 给出了作业的规模和类型。图 6.6 处各个级别负载作业的运行时间的累积分布函数 (CDF)。

表 6.2 混合负载中的各个级别的作业大小和类型

级别	作业类型	Map 任务数	Reduce 任务数	作业运行数
1	select	1	无	38
2	text	search	2	无
3	aggregation	10	3	14
4	select	50	无	8
5	text	search	100	无
6	aggregation	200	50	6
7	select	400	无	4
8	aggregation	800	180	4
9	join	2400	360	2
10	text	search	4800	无



(1) 级别 1-3

(2) 级别 4-8

(3) 级别 9-10

图 6.6 混合负载中不同规模作业响应时间的累积分布函数图

图 6.6 给出了不同规模作业响应时间的累积分布函数图。图中可以看出公平调度和带等待时间的公平调度都显著的减少了小作业的响应时间，对大作业的响应时间的改善很小。

6.7 本章小结

本章主要通过实验来验证提出的基于时间的等待调度方法的有效性。通过针对不同的负载类型和作业类型在 FIFO、公平调度方法和基于时间的等待调度方法下的性能比较验证方法的可用性和高效性。实验结果与预期效果基本吻合，结果表明，基于时间的等待调度方法在提高系统吞吐率与公平性方面都具有较为显著的效果。

第七章 总结和展望

7.1 总结

随着对大规模数据密集型运算集群的使用更加广泛和普遍，对于集群的共享的研究将日益引起业界和学术界的关注。大型计算集群的搭建为人们在大数据集上进行查询和计算提供了硬件基础，MapReduce 计算模型的提出则为人们更好的使用集群的计算能力更加有效和快速的完成运算提供了知道方法。如何更好的向更多的用户提供服务，使得每个用户都能公平的享用技术革新带来的成果，使得整个集群的计算和存储能力得到更大的发挥作用，则是集群系统中作业调度要解决的问题。试图通过对实现了 MapReduce 计算模型的开源系统 Hadoop 中作业调度的研究来解决这一问题。基于时间的等待调度方法，对于现有的作业调度方法存在的数据本地性方面的问题给出了基于时间的等待调度的方案，从而使在集群进行计算时更少的移动数据，减少对整个集群的 IO 带宽的压力，改善集群的吞吐率和单个用户的平均响应时间。

本文的主要贡献：

- 1) 设计了一种基于时间的等待调度方法，能够更好的满足多用户在使用 MapReduce 集群时的公平性，同时提高整个集群系统的使用性能，增加集群的吞吐率，减少单个用户的平均作业响应时间。
- 2) 对于 Hadoop 集群使用的作业调度方法进行了深入的研究。根据 MapReduce 模型的特点分析了现有调度方法在多用户共享方面的不足。
- 3) 通过实验对各种调度方法的实际性能进行了验证分析。

7.2 下一步工作

下一步将从以下几方面来扩展和深入我的研究工作：

- 1) 对于 Hadoop 集群的用户使用情况和作业的提交类型进行更深一步的研究。实践是检验真理的唯一标准，只有用户的具体需求和实际需要才能有的放矢，更好的改进集群的性能。
- 2) 对于集群的作业调度方法，将本文提出的方法进行新的研究与改进，并且也针对 MapReduce 相关技术进行相应的研究。
- 3) 将新的研究成果应用于系统中，使系统成为一个能够有效进行大规模数据存储和计算分析的系统。

致 谢

在本文完成之际，谨向所有给予我指导、关心、支持和帮助的老师、领导、同学和亲人致以衷心的感谢！

首先衷心地感谢我的导师吴泉源老师在我的攻读硕士阶段对我的课题、学习的指导、关心和帮助。吴老师严谨踏实的治学作风，杰出的教学科研能力，谦虚热情的为人，渊博的知识和严密的思维，令我受益匪浅。

诚挚地感谢杨树强老师。在做课题的日子里，杨老师对项目任务的悉心规划和安排，是我课题得以顺利进展的重要因素。杨老师对问题的宏观把握能力，敏锐思维，迅速定位和解决问题的能力都让我受益良多。杨老师的出色的科研能力，高深的学术造诣，平易近人的作风，精益求精和严谨创新的治学态度，对事业的热爱都是我学习的榜样。

在整个毕业课题设计过程中，还得到了周斌老师的悉心指导，周老师在学术上的高深造诣以及精益求精和严谨创新的治学态度，使我受益非浅，在此谨向周老师表示衷心的感谢！

诚挚地感谢贾焰、韩伟红、李爱平老师，他们的悉心指导和帮助也是我的毕业课题得以完成的基础。

感谢樊华博士、赵辉博士、陈志坤博士，他们对我的课题工作进行了全面指导，跟他们在一起学习研究，我受益良多。

诚挚的感谢王忠儒、金松昌等同学，在我做课题期间，和他们的讨论给了我很大的帮助。

感谢实验室的同学，与他们一起渡过了一个愉快的研究生学习生活。

感谢舍友、同班同学，与他们一起生活，带给我很多快乐，与他们一起讨论，是我受益匪浅，感谢和我一起学习、生活的所有同学，他们的关心和帮助是我感受到集体的温暖。

感谢研究生院、计算机学院、硕士生队领导以及全体同学对我的关心和帮助。

要深深地感谢我的父母、妹妹。在我在外求学的时时刻刻，我都能感受到你们对我的温暖关怀，在我成长的每一个足印里，都倾注了他们的心血和汗水，再次感谢我远方的亲人们。

衷心感谢所有给予我支持和帮助的人！

参考文献

- [1] Porter, M. E. (1998) Clusters and the new economics of competition. Harvard Business Review, November/December: 77~90.
- [2] Cloud Computing. http://en.wikipedia.org/wiki/Cloud_computing. 2010-06-20/2010-11-7.
- [3] Vouk, M. A. 2008. Cloud computing Issues, research and implementations. In 30th International Conference on Information Technology Interfaces (ITI 2008). Cavtat/Dubrovnik, Croatia, June 2008, 25~40.
- [4] Fellows, W. 2008. Partly Cloudy, Blue-Sky Thinking About Cloud Computing. Whitepaper. 451 Group. 2008.
- [5] Amazon Elastic Compute. <http://aws.amazon.com/ec2/>. 2010-06-20/2010-10-07.
- [6] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A.Konwinski, G. Lee,D.Patterson,A.Rabkin,I.Stoica,M.Zaharia.Above the Clouds: A Berkeley View of Cloud Computing. Technical Report No. UCB/EECS-2009-28, University of California at Berkley, USA, Feb. 10, 2009.
- [7] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A mapreduce framework on graphics processors. In PACT, 2008.
- [8] 黎春兰, 邓仲华等. 论云计算的价值. 图书与情报. 2009(4):34-36.
- [9] Danielson, Krissi (2008-03-26). Distinguishing Cloud Computing from Utility Computing. Ebizq.net. Retrieved 2010-08-22.
- [10] Sun. 云计算架构介绍. 2009-06:15~35.
- [11] Eucalyptus home page . <http://www.eucalyptus.com/>. 2010-06-20/2010-11-07.
- [12] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youse_, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In Proceedings of Cloud Computing and Its Applications [Online], Chicago, Illinois, 2008-10.
- [13] Luiz A. Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The Google cluster architecture. IEEE Micro, 23(2):22–28, April 2003.
- [14] R. Lämmel. Google's MapReduce Programming Model - Revisited. Draft; Online since 2 January, 2006; 26 pages, 22 Jan. 2006.
- [15] Apache Hadoop framework. <http://hadoop.apache.org>. 2010-06-20/2010-11-07.
- [16] Jason Venner. Pro HADOOP. Apress L. P., Berkeley, California, first edition, 2009:10~68.
- [17] Hadoop On Demand Documentation. <http://hadoop.apache.org/core/docs/r0.17.2/hod.html>. 2010-06-20/2010-11-07.

- [18] Hadoop's Capacity Scheduler.
http://hadoop.apache.org/common/docs/r0.19.2/capacity_scheduler.html.
2010-07-10/2010-11-02.
- [19] Hadoop's Fair Scheduler Guide.
http://hadoop.apache.org/common/docs/r0.20.0/fair_scheduler.html.
2010-06-22/2010-11-05.
- [20] 陈国良, 孙广中, 徐云, 等. 并行算法研究方法学[J]. 计算机学报, 2008, 31(9): 1493~1502.
- [21] 陈国良. 并行计算一结构-算法-编程[M]. 北京: 高等教育出版社, 1999:35~83.
- [22] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. Commun. ACM, 2008.
- [23] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce Online," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-136, 2009.
- [24] Ekanayake, J., S. Pallickara, et al. (2008). MapReduce for Data Intensive Scientific Analyses. eScience, 2008. IEEE Fourth International Conference on eScience '08.2008.
- [25] Hadoop Map/Reduce tutorial.
http://hadoop.apache.org/common/docs/current/mapred_tutorial.html.
2010-06-28/2010-11-02.
- [26] T. White. 2009. Hadoop: The Definitive Guide. O'Reilly Media, Inc.June 2009:25~89.
- [27] MapReduce and Hadoop 在淘宝.
<http://www.tbdata.org/>. 2010-08-09/2010-11-01.
- [28] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System.In Proceedings of the 26th IEEE Symposium on Massive Storage Systems and Technologies (MSST '10), Incline Village, Nevada, May 2010.
- [29] J. Leverich and C. Kozyrakis. On the energy (in)efficiency of hadoop clusters. In HotPower, 2009.
- [30] The Phoenix System for MapReduce Programming.
<http://mapreduce.stanford.edu/>. 2010-06-24/2010-11-05.
- [31] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In SOSP 2009, 2009.
- [32] MapReduce on Facebook.
<http://www.facebook.com/pages/MapReduce/239177839884>.2010-08-05/2010

-11-03.

- [33] M.Zaharia. Hadoop Fair Scheduler.
<http://developer.yahoo.net/blogs/hadoop/FairSharePres.ppt>. 2009.
- [34] Mark E. Crovella , Mor Harchol-Balter , Cristina D. Murta, Task assignment in a distributed system (extended abstract): improving performance by unbalancing load, Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems, p.268-269, June 22-26, 1998, Madison, Wisconsin, United States .1998.
- [35] Santa Clara Marriott.Job Scheduling with the Fair and Capacity Schedulers.cloudera.2009.
- [36] Bora Ucar , Cevdet Aykanat , Kamer Kaya , Murat Ikinci, Task assignment in heterogeneous computing systems, Journal of Parallel and Distributed Computing, v.66 n.1, p.32-46, January 2006 .
- [37] B. Schroeder and M. Harchol-Balter. Evaluation of task assignment policies for supercomputing servers: The case for load unbalancing and fairness. In Proceedings of High Performance Distributed Computing (HPDC ' 00), pages211–219, 2000.
- [38] MapReduce on Yahoo!.
<http://developer.yahoo.com/hadoop/tutorial/module4.html>.
2010-07-02/2010-11-05.
- [39] 孟令芬. pc 集群作业调度算法研究[D].上海. 中国石油大学（华东）.2009.
- [40] J. Varia. Cloud architectures. Technical report, Amazon Webservices, 2008.
- [41] L. G. Valiant. A bridging model for parallel computation.Communications of the ACM, 33(8):103–111, 1997.
- [42] Cluster Setup of Hadoop.
http://hadoop.apache.org/common/docs/current/cluster_setup.html.
2010-03-05/2010-11-03.
- [43] Hive performance benchmarks.
<https://issues.apache.org/jira/browse/HIVE-396>. 2010.
- [44] Apache Hive. <http://hadoop.apache.org/hive/>.2010-03-02/2010-11-04.
- [45] Ashish Thusoo , Joydeep Sen Sarma , Namit Jain , Zheng Shao , Prasad Chakka , Suresh Anthony , Hao Liu , Pete Wyckoff , Raghotham Murthy, Hive: a warehousing solution over a map-reduce framework, Proceedings of the VLDB Endowment, v.2 n.2, August 2009.
- [46] J. Basney and M. Livny, Deploying a High Throughput Computing Cluster, High Performance Cluster Computing, R. Buyya (editor). Vol. 2, Chapter 5, Prentice Hall PTR, May 1999.
- [47] Single Node Setup of Hadoop.

[http://hadoop.apache.org/common/docs/current/single_node_setup.html.](http://hadoop.apache.org/common/docs/current/single_node_setup.html)

2010-03-02/2010-11-05.

[48] Amazon Elastic MapReduce. [http://aws.amazon.com/elasticmapreduce/.](http://aws.amazon.com/elasticmapreduce/)
2010-08-05/2010-11-03.

[49] 王凯,杨树强,吴泉源.虚拟化技术在云计算系统中的应用.第三届湖南省研究生创新论坛高性能计算与信号处理分论坛.2010.10:194~199.

作者在学期间取得的学术成果

- 1.王凯, 吴泉源, 杨树强.一种多用户 MapReduce 集群的作业调度算法的设计与实现.计算机与现代化,2010(10). (已录用)。
- 2.王凯, 杨树强, 吴泉源.虚拟化技术在云计算系统的应用.湖南省第三届研究生创新论坛高性能计算与信号处理分论坛,2010.10. (已录用)。