一种优化 MapReduce 系统能耗的任务分发算法

宋 杰[1)] 徐 澍[1)] 郭朝鹏[1)] 鲍玉斌[2)] 于 戈[2)]

[1)](东北大学软件学院 沈阳 110819)

[2)](东北大学信息科学与工程学院 沈阳 110819)

摘 要 MapReduce 是一种典型的分布式计算模型,一经提出就被迅速应用到大数据处理系统中.文中认为 MapReduce 系统在能耗方面存在优化空间.对于一个分布式并行计算系统,任务的并行性对任务执行性能影响显著,并行性保证方法在优化性能的前提下还应该考虑系统能耗.在 MapReduce 系统中,传统的 Map 任务分发算法采用"小任务多次分发的策略",这种策略虽然保证了并行性,但会浪费节点的处理能力,消耗额外的能量;而 Reduce 任务分发算法尚不能保证 Reduce 任务间的并行性.文中提出通过动态地调整 Map 任务和 Reduce 任务大小,也即任务处理数据量的规模来保证任务并行性,降低 MapReduce 系统的整体能耗.文中通过实验证明该方法能够有效地降低典型 MapReduce 作业的能耗.

关键词 MapReduce;能耗;能耗优化;任务分发;并行性;云计算;大数据

中图法分类号 TP311 **DOI 号** 10.11897/SP.J.1016.2016.00323

A Task Distribution Algorithm for Energy Consumption Optimization of MapReduce System

SONG Jie[1)] XU Shu[1)] GUO Chao-Peng[1)] BAO Yu-Bin[2)] YU Ge[2)]

[1)](*Software College*, *Northeastern University*, *Shenyang* 110819)

[2)](*School of Information Science and Engineering*, *Northeastern University*, *Shenyang* 110819)

Abstract While MapReduce is proposed as a typical distributed computing model, it caused a huge repercussion and applied rapidly to big data processing. However, its energy consumption still can be optimized, for the distributed parallel computing system, the parallelism of tasks is the key to performance, the parallelism ensuring approach should consider not only time consumption but also energy consumption. In order to improve the parallelism, the traditional Map task distribution algorithm use the "fine granular task distribution strategy" to improve the parallelism, but it wastes the energy; and Reduce task distribution algorithm cannot guarantee the parallelism among the Reduce tasks. In this paper, we optimize MapReduce by adjusting the size of Map tasks and Reduce tasks dynamically, which can save energy consumed by MapReduce system. The algorithm proposed to this paper has been proved effective in reducing the energy consumption through a series of experiments.

Keywords MapReduce; energy consumption; energy consumption optimization; task distribution; parallelism; cloud computing; big data

收稿日期:2014-08-25;在线出版日期:2015-07-23.本课题得到国家自然科学基金(61433008,61202088,61272179,61173028)、教育部博士点基金(20120042110028)、教育部-英特尔信息技术专项科研基金(MOE-INTEL-2012-06)、中央高校基本科研业务费专项资金(N130417001)、中国博士后科学基金面上项目(2013M540232)、辽宁省博士启动基金(201403314)资助.**宋 杰**,男,1980 年生,博士,副教授,中国计算机学会(CCF)高级会员,主要研究方向为高能效计算、大数据存储与管理、云计算. E-mail: songjie@mail.neu.edu.cn. **徐 澍**,男,1991 年生,硕士研究生,主要研究方向为高能效计算. **郭朝鹏**,男,1990 年生,硕士研究生,主要研究方向为大规模数据计算. **鲍玉斌**,男,1968 年生,博士,教授,中国计算机学会(CCF)高级会员,主要研究领域为数据仓库. **于 戈**,男,1962 年生,博士,教授,博士生导师,中国计算机学会(CCF)高级会员,主要研究领域为数据库理论.

1 引 言

MapReduce 是一种分布式地完成大规模数据集处理和分析运算的有效技术，基于 MapReduce 的应用程序能够在大量普通配置的计算机上并行地执行，而 MapReduce 框架则需完成数据分割、任务分发、资源分配、节点容错、节点通信以及结果保存等复杂操作. 通常，性能是衡量一个软件系统优劣的重要指标，现在的 MapReduce 研究热点大致为以下三种：对 MapReduce 框架进行性能优化[1]，或提高基于 MapReduce 的数据分析算法的性能[2]，或在 MapReduce 的基础上寻找更为高效的编程模型[3]. 然而，随着 IT 设施功率的逐渐增加和集群规模的不断扩大，系统能耗过大已经成为 IT 界面临的一个难题，因此近些年能耗逐渐也成为衡量软硬件系统的一个新指标. 美国斯坦福大学的一份研究报告指出 2010 年全球数据中心的耗电量为 1988 亿千瓦时，占全球总耗电量的 1.3%，前者在过去十年增加了 10%①，未来五年还将增长 56%[4]. 此外，绿色和平组织预测，到 2020 年，全球主要 IT 运营商的能耗将达到 2 万亿千瓦时，超过德、法、加和巴西等 4 国的能耗总和②. 从环境角度，一台功率为 300 W 的计算机一年间接排放出多达 1300 kg 的二氧化碳，亿万台这样的计算机将给环境带来沉重压力[5]. 在此背景下，如何利用 MapReduce 系统，以一种低能耗的方式完成计算，成为当今研究的热点问题.

本文重点从 MapReduce 系统能耗优化角度展开研究. 文中提及的 MapReduce 模型指由 Google 为了解决海量数据的处理而提出的分布式编程模型；MapReduce 框架则是 MapReduce 模型的实现，以 Hadoop MapReduce 为例；MapReduce 系统是视 MapReduce 框架和其运行的集群环境为一个整体的系统. 本文将改进 MapReduce 框架中 Map 和 Reduce 任务的任务分发算法，在保证性能的同时降低 MapReduce 系统能耗.

本文第 2 节介绍问题的定义和本文提出的能耗优化算法的基本思路；第 3 节描述相关工作；第 4 节给出改进的 Map/Reduce 任务模型；第 5 节和第 6 节分别描述改进的 Map 任务分发算法和 Reduce 任务分发算法；第 7 节分析算法的时间和空间复杂度；第 8 节通过实验验证本文提出的能耗优化方法；第 9 节总结全文并提出进一步工作方向.

2 能耗优化思路

总览现有高能效计算的研究成果，经常出现的关键词为数据中心、云计算、虚拟化、大规模分布式系统和传感器网络等等，由此可见，能耗优化均集中在硬件层和平台软件层，同时充分考虑网络特性. 目前能耗优化主要有两种思路：一是开发更加节能的硬件设备，二是考虑软件系统特性，采用基于软件的能耗优化方法，如改变 CPU 的状态、关闭或休眠节点、动态迁移虚拟机、作业调度等. 如今随着 MapReduce 编程模型的推广与应用，大多数的软件都与之相关，而本文的优化目标就是降低 MapReduce 系统能耗. 我们将 MapReduce 框架和其运行的硬件环境视为一个整体，称为 MapReduce 系统，其能耗属于软件系统的能耗，文献[6-7]都对软件系统的能耗度量和优化问题加以研究. 而我们则结合 MapReduce 的特点，提出能耗优化的新思路：减少“等待能耗”. 在集群环境中，计算机节点(或计算机某组件)因等待其他资源而处于“被动空闲”的时间段内消耗的电能称为等待能耗[8]. 例如，当集群中某个节点在等待其他节点的运算结果时，该节点消耗的能量就是等待能耗. 同理，对于一个计算机内部的各个组成部分之间也相互等待，如 CPU 运算会因等待 I/O 操作而阻塞，这部分等待能耗同样可以优化. 现有基于动态开关机的算法并不适用于这种短暂的节点被动空闲，我们应该尽量减少节点对资源、中间结果的等待，以减少等待能耗. 根据等待能耗的定义，我们认为 MapReduce 系统在能耗方面存在很大的优化空间.

MapReduce 模型把作业分解成顺序执行的 Map 阶段和 Reduce 阶段，Map/Reduce 阶段包含若干部署到 Map/Reduce 节点并行执行的 Map/Reduce 任务，当所有 Map/Reduce 任务执行结束后 Map/Reduce 阶段才结束. 首先，我们讨论 Map 阶段的节点等待问题. Map 阶段的执行时间取决于最后一个 Map 任务的结束时间，若 Map 任务的并行度不高，会导致节点等待，产生等待能耗. MapReduce 模型采用细化 Map 任务大小的方法来保证 Map 任务的

① http://www.intel.com/content/dam/doc/technology-brief/efficient-datacenter-high-ambient-temperature-operation-brief.pdf

② http://www.greenpeace.org/international/en/

并行性，因此，在最坏的情况下，其他 Map 节点将会等待“性能最差的节点远程处理 64 MB 数据”所消耗的时间. Map 任务越小，其并行性越好，但细分 Map 任务带来的任务调度开销不容忽视. 设 Map 阶段处理 1 TB 的数据，每个 Map 任务的大小为默认的 64 MB，则会有 16 384 个 Map 任务，若每次任务分发的时间平均为 1.5 s①，则 Map 节点将会有 1.5 s 被动空闲时间来等待新任务，那么 Map 阶段因任务分发而产生的节点空闲时间为 24 576 s，若节点的空转功率为 40 W，则会消耗 983 kJ 的能量，因此在 MapReduce 系统中由任务分发带来的能耗难以忽视.

其次，我们讨论 Reduce 阶段的任务等待问题，Reduce 阶段的结束时间取决于最慢的 Reduce 任务的结束时间. Map 任务的大小是固定的，而 Reduce 任务的大小则是动态的，是由每个 Reduce 任务所包含的键（Key）的种类决定，且 Reduce 任务与 Reduce 节点存在一一对应的关系. Reduce 任务分发算法根据 Map 阶段的输出结果决定每个 Reduce 任务的大小以及与节点的对应关系. 因此，当某 Reduce 节点处理完当前 Reduce 任务后，其既不能从其他节点获取部分 Reduce 任务继续处理，也不存在“仍未开始执行的 Reduce 任务再分发给此 Reduce 节点”的情况，因此 Reduce 阶段的处理时间取决初始分发后最慢的 Reduce 任务执行时间. Reduce 任务规模不均衡将直接影响运算的并行性，延长作业完成时间，产生等待能耗.

本文提出一种改进的 Map 任务和 Reduce 任务分发算法，其核心思想是尽量减少任务分发次数并确保高并行性，减少因任务同步而造成的等待. 其大体思路如下：(1) 采用 Map 任务单次分发算法，一次性将所有 Map 任务全部分发给 Map 节点，减少 Map 节点因等待 Map 任务而产生的被动空闲，降低等待能耗；(2) 采用 Map 任务 Self-resizing 算法，使每个 Map 任务在执行过程中对自身的数据量进行调整，优先处理本地数据，并保证所有任务可以同时完成，减少 Map 节点因 Map 并行性不高而产生的被动空闲，也减少的网络 I/O 量，降低等待能耗；(3) 采用 Reduce 任务均衡分发算法，在分发 Reduce 任务时确保每个任务的执行代价近似相等，使 Reduce 节点间的负载更加均衡，提高 Reduce 任务的并行性，降低等待能耗. 实现上述算法存在一些挑战：(1) 首先我们需要扩展传统的 Map/Reduce 任务模型，对 Map/Reduce 任务处理的数据块、数据特征、任务执行时间、任务分发状态等进行抽象、建模；(2) 在 Map 任务自调整算法和 Reduce 任务均衡分发算法中，数据块与任务间存在海量的组合方式，我们需要快速地找到最优或较优的组合方式；(3) 算法应考虑分布式环境的特点，对于 Map/Reduce 任务的分发算法可采用主从模式，由 JobTracker 完成任务分发，Map 任务自调整算法则应该是每个任务自发的，且采用去中心化的体系结构. 一般情况下，大部分的优化问题都可以用智能算法，如遗传算法（GA）、粒子群优化算法（PSO）、蚁群优化算法（ACO）等解决，但这些算法都过于复杂，大多适用于允许用较长时间确定一个最优解的应用，并不适合于本研究提出的实时任务分发算法.

最后，由前文分析可知，Map/Reduce 任务的并行性越好，等待能耗越小，而我们讨论 Map/Reduce 任务的执行时间以及并行度时都仅考虑任务处理的数据量和数据位置，而忽略了算法复杂度和数据特征的影响. 首先，我们讨论算法复杂度. 由于每个 Map/Reduce 任务的算法相同，其复杂度也相同，因此算法复杂度本身不会影响任务的并行程度，但当 Map/Reduce 任务大小不同时，越是复杂的算法越会导致其处理时间差距越大，并行度越差，因此算法复杂度应为并行性的一个常数系数，在同一作业的任务间并行度度量时可以被约去，所以本文并不做考虑. 其次，我们讨论 Map 任务的数据特征. 若 Map 任务中包含“对数据进行先判断后处理”的筛选逻辑，则输入数据的数据特征会影响任务的执行时间，但本研究未考虑 Map 任务的数据特征，因为如果数据是随机分布在节点上的，每个 Map 任务筛选数据的比例是近似相同的，因此总体上数据特征不会直接影响任务并行度；更重要的一个原因是，本文采用的 Map 任务 Self-resizing 算法会根据实际的执行情况动态调整任务大小，即使某任务数据倾斜导致 Map 任务执行时间与数据量之间比例关系失效，任务也能通过 Self-resizing 算法进行自我调整. 最后，我们讨论 Reduce 任务的数据特征. Reduce 任务通常完成数据聚集逻辑，一般会对所有输入数据逐一处理，鲜有数据筛选逻辑，部分 Reduce 任务的运算

① Hadoop MapReduce 的心跳间隔为 3 s，任务会随心跳检测发送到 JobTracker，即使不考虑通讯和运算时间，因此任务分发时间应该在 3 s 内的一个随机值，过于频繁的心跳检测会浪费大量的网络带宽.

逻辑与输入数据特征有关，如 Reduce-Join[9] 中一个 Reduce 任务输入的数据量是来自两个表的数据量 $R+S$，而运算量 $R\times S$ 的笛卡尔连接次数则与数据特征相关，即相同的数据量也会导致不同次数的笛卡尔连接运算，但在这种情况下我们可以通过改变数据量的评价指标，使数据特征与数据量间形成某种关系，从而兼顾了数据特征对并行度的影响，因此本研究同样未直接考虑 Reduce 任务的数据特征.

3 相关工作

MapReduce 的优化大致存在两个方向，一个是性能的优化，另一个是能耗的优化，前者又可以分为两类，一类是基于任务调度算法的优化，而另一类则侧重于单个作业的运行效率优化：(1) 任务调度算法研究更多关注“对多个同时运行的作业调整其任务运行顺序”以及“对计算资源的分配”，可以说是从宏观上优化 MapReduce，比如 Hadoop 默认的 FIFO 算法；针对 FIFO 作业调度算法在实际应用中存在的问题，Facebook 和 Yahoo 的工程师分别提出新的作业调度算法：公平份额调度(Fair Scheduling)[10] 和计算能力调度(Capacity Scheduling)[11]，这两种算法已经获得广泛认可并付诸实践；(2) 而另一种优化方式从程序设计的角度出发，尽量减少影响程序运行效率因素，合理利用计算资源，提高程序的并行性. 文献[12]提出将一个作业划分成几个小作业，每一个小作业执行完就删除数据，从而降低程序运行过程中本地磁盘上保存的中间数据量，使其维持在合理的水平；文献[13] 将首次任务分配从“拉”模式转变为“推”模式，以提高短作业的执行效率；文献[14]提出通过自动调整 MapReduce 框架的性能参数来实现对应用程序的自动优化；文献[15]提出一种名为 LATE 的调度器，通过提高 Hadoop 集群的响应时间优化性能；文献[16]提出一种静态的数据布局的方法，通过保证数据的本地性，从而提升异构集群的性能. 但是现有工作中对于任务 Map 阶段间性能浪费的关注却很少，任务分发算法不同于任务调度算法，前者关注分派何种任务给哪个节点，而非在任务队列中选择下一个执行的任务，在任务分发研究中，我们则更多地考虑任务特征，如任务的大小，而非考虑任务的分发目的地.

与传统的性能优化相比，近些年能耗问题由于大规模集群系统能源消耗不断增长而倍受关注. 同时还有很多以软件系统的能耗为优化对象的研究，例如 MapReduce 系统. MapReduce 系统的能耗优化方法大多以集群资源为出发点，根据任务或任务队列的特性，通过提高资源利用率，减少能源损失，进而优化能耗. Leverich 等人[17] 提出通过改变一系列 Hadoop 集群的配置参数，可以节省 9%～50% 的能源消耗. Stillwell 等人[18] 按照任务的最佳资源请求、平均收益、失败率等排序，然后通过启发算法解决资源分配问题. Song 等人[19] 关注 MapReduce 系统的数据传输、数据压缩时的能耗优化，为了实现资源最佳分配提出了 3 种调度器：应用层调度器、本地调度器和全局调度器. Chen 等人[20] 将 Map/Reduce 任务继续划分为 CPU 密集型和 I/O 密集型阶段，采取称为“资源使用管道”的技术让细分后的不同任务阶段同时运行，从而提高 CPU 和 I/O 的使用率. Wirtz 等人[21] 提出通过合理的资源分配和动态电压和频率的调整可以有效地节约 MapReduce 系统对能源的消耗，同时他在另一篇文献中认为“通常情况下数据迁移是性能和能耗的瓶颈”，并基于此问题提出一种可预测的针对 MapReduce 系统的数据迁移方法和分析框架[22]. Chen 等人[23] 针对基于 MapReduce 的交互型应用提出，采用一部分节点作为池保存短时间内经常访问的数据，可以无需重复在海量数据中查找数据，以节约能源；Lang 等人[24] 则提出一个框架，系统地考虑 MapReduce 系统中各类节点的断电策略，将使用率低的节点断电，以降低集群的整体能耗. 但是这里所有的研究都是针对于资源分配而非任务分发，而本文所提出的优化方法正是弥补这一类研究的空白.

4 改进任务模型

本节定义一个改进的 Map/Reduce 任务模型. 经过前文分析，节点等待是产生等待能耗的主要原因，而造成节点等待的主要原因是“等待任务分发”以及“等待执行较慢任务”，前者通过 Map 任务的单次算法解决，后者则通过 Map 任务自调整以及 Reduce 任务均衡分发实现. 改进的任务分发算法核心思想是通过尽可能少的任务分发次数保证任务并行性，前文分析影响任务并行性的主要因素包括 Map/Reduce 任务的大小和数据特征. 本节将定义

改进的任务模型的相关属性，表 1 首先列出了本文涉及所有符号及其含义.

表 1　相关符号说明

符号	含义描述
p	机架数量
q	机架中节点数量，为了描述功能更加简单，假设每个机架的节点数量相等
δ	并行度
n_{ij}	第 i 个机架的第 j 个节点
n_m	Map 任务的数量
ε_{ij}	节点 n_{ij} 的性能以及执行算法复杂度常量($\varepsilon_{ij}>0$)
m_{ij}	第 i 个机架的第 j 个节点处理的 Map 任务
r_{ij}	第 i 个机架的第 j 个节点处理的 Reduce 任务
$\tau(r_{ij}),\tau(m_{ij})$	m_{ij} 或 r_{ij} 的处理时间
e_s	Map 输出的 w 种 Key 的一种，其中 $s\in[1,w]$
$\boldsymbol{S}^k$	第 k 次 $p\times q$ Map 任务矩阵，映射任务块在集群中的分布方式
L_{ij}^k	m_{ij} 在第 k 次 Map 任务矩阵中的本地任务数量
G_{ij}^k	m_{ij} 在第 k 次 Map 任务矩阵中的机架任务数量
R_{ij}^k	m_{ij} 在第 k 次 Map 任务矩阵中的远程任务数量
$\alpha:\beta:\gamma$	读取本地、机架以及远程数据的 I/O 速度之比

定义 1.　Map 任务规模. 一个 Map 任务的规模定义为该 Map 任务处理的数据块的数量. 其中数据块是文件系统中数据最小组织单位，MapReduce 作业的输入数据被划分为若干数据块. 设 m_{ij} 是运行在第 i 个机架上第 j 个节点 n_{ij} 的 Map 任务，L_{ij}，G_{ij} 和 R_{ij} 分别表示该任务处理的本地数据块，机架内(In-rack)数据块和远程数据块的个数. 并将该 Map 任务的规模 $|m_{ij}|$ 定义为 $|m_{ij}|=L_{ij}+G_{ij}+R_{ij}$.

Reduce 任务与 Map 任务不同，Reduce 任务处理的数据来自于 Map 任务的输出数据，数据量是动态的，且数据都位于 Reduce 节点本地，因此无需考虑数据位置对任务执行时间的影响. Map 任务和 Reduce 任务的输入和输出都是由用户定义的键值对形式，其中 Map 任务输入 $\langle Key_M^{\text{In}},Value_M^{\text{In}}\rangle$，输出 $\langle Key_M^{\text{Out}},Value_M^{\text{Out}}\rangle$. Reduce 任务输入为 $\langle Key_R^{\text{In}},Value_R^{\text{In}}\rangle$，输出为 $\langle Key_R^{\text{Out}},Value_R^{\text{Out}}\rangle$，其中 Key_M^{Out} 能隐式的转换为 Key_R^{In}，$Value_R^{\text{In}}$ 是保存 $Value_M^{\text{Out}}$ 的集合. Reduce 任务和 Reduce 节点一一对应，且 Reduce 任务个数不大于 Key_R^{In} 的种类，并保证相同 Key_R^{In} 对应的 $Value_R^{\text{In}}$ 在同一个 Reduce 任务中处理. 因此，只需要考虑何种不同 Key_R^{In} 之间的组合方式可以确保 Reduce 任务并行度最大.

定义 2.　Key 规模. 某个 Key 的规模是指 Map 任务产生的中间结果中，该 Key_M^{Out} 所对应的全部 $Value_M^{\text{Out}}$ 数据量，或 Reduce 任务的输入中该 Key_R^{In} 对应 $Value_R^{\text{In}}$ 包含的数据量($Value_R^{\text{In}}$ 是保存 $Value_M^{\text{Out}}$ 的集合). 为简化描述，下文中出现的 Key 和 Value 分别为 Key_R^{In} 和 $Value_R^{\text{In}}$. 设 Map 输出了 w 种 Key，分别记为 $e_1,e_2,e_s,\cdots,e_w$，并记 e_s 的规模为 $|e_s|$.

定义 3.　Reduce 任务规模. Reduce 任务规模为该 Reduce 任务输入所有 Key 的规模之和. 为了与 Map 表述一致，我们设集群所有节点均参与 Reduce 任务及 Key 的种类远大于节点数目，$w\gg p\times q$. 设 r_{ij} 是运行在第 i 个机架上第 j 个节点 n_{ij} 的 Reduce 任务，$|r_{ij}|$ 表示其任务规模. 若 r_{ij} 的输入仅有两种 Key，e_s 和 e_t($s,t\in[1,w]$)，则 $|r_{ij}|=|e_s|+|e_t|$.

定义 4.　τ 函数. τ 函数返回该任务的执行时间，设 n_{ij} 为第 i 个机架上第 j 个节点，m_{ij} 是运行在 n_{ij} 的 Map 任务，r_{ij} 是运行在 n_{ij} 的 Reduce 任务. 同时，可知一个算法复杂度已知的任务其执行时间取决于两个因素，其一是任务量，其二则是节点性能. 节点性能越好，任务量越小，任务执行时间也越短，因此可以得到式(1)和式(2)：

$$\tau(m_{ij})=\varepsilon_{ij}\times(\alpha L_{ij}+\beta G_{ij}+\gamma R_{ij}) \tag{1}$$

$$\tau(r_{ij})=\varepsilon_{ij}\times|r_{ij}| \tag{2}$$

其中 $\alpha:\beta:\gamma$ 代表本地、机架以及远程数据的 I/O 速度比. ε_{ij} 是性能常数，包括节点 n_{ij} 的性能常量，值越小则表示节点性能越好. 式(1)给出了第 i 个机架上以及整个 Map 阶段的执行时间，其中 p 为机架数量，q 为每机架中节点数量，为了简化描述，假设每个机架的节点数量相等. 另外，依据木桶效应[25]，易知在同一机架中 Map 阶段的完成时间取决于机架中最后完成 Map 任务的节点；同理，整个系统的 Map 阶段完成时间取决于最后完成 Map 阶段的机架. 由此可以得到式(3)，$\tau(m_i)$ 表示同一机架中 Map 阶段的完成时间，$\tau(m)$ 是求解整个系统中 Map 阶段的完成时间的表达式.

$$\begin{aligned}&\tau(m_i)=\max\{\tau(m_{ij})\,|\,1\le j\le q\},\\&\tau(m)=\max\{\tau(m_i)\,|\,1\le i\le p\}\end{aligned} \tag{3}$$

定义 5.　任务并行度. Map 任务并行度 δ_m 表示各 Map 任务的并行执行的程度，其取值区间为 $(0,1]$，当取值为 1 时，任务并行度达到最大. 同理，定义 Reduce 任务并行度 δ_r.

以 Map 任务并行度为例，设 Δ 表示所有 Map 任务(节点)的等待时间，其数值为每个节点上任务执行时间与整个 Map 阶段完成时间的绝对偏差之和. 且 Map 任务的等待时间与 Map 任务的并行度成反比，等待时间越长则任务并行度越小，因此定义 $\delta_m=(1+\Delta)^{-1}$ 使 δ_m 与 Δ 负相关，且避免 Δ 值为零的

情况发生,其中 Δ 表达式如式(4)所示.

$$\Delta=\sum_{i=1,j=1}^{p,q}(\tau(m)-\tau(m_{ij})) \tag{4}$$

又由式(3)可知,$\tau(m_{ij})$间差值越小,则 Δ 的值就会最小,$\delta_m=(1+\Delta)^{-1}$的值就越大,并行度越高.又因为总任务量是一致的,随着每个 Map 任务完成时间的差值变小,Map 阶段的总体的完成时间 $\tau(m)$的值也会变小.因此 Map 任务分发算法的目标是最小化 $\tau(m_{ij})$间的差值.同理 Reduce 任务分发算法要最小化 $\tau(r_{ij})$之间的差值,又因为 Reduce 任务与 Reduce 节点的关系不会影响 Reduce 任务的完成时间,因此此时 $\tau(r_{ij})$的总和为常量,仅 $\tau(r)$为变量,所以 Reduce 任务分发算法只需要最小化 $\tau(r)$的值.

5 Map 任务分发算法

Map 任务单次分发算法将 Map 任务以合适的规模映射给每个 Map 节点,以此减少反复分发任务的开销;且 Map 任务 Self-resizing 算法动态地调整任务规模,最大化 Map 任务的并行性.本节将描述这两个算法的细节.本文提出的 Map 任务分发算法能够保证 Map 任务和 Map 节点之间的一一映射,且集群所有节点都执行 Map 任务.下面我们对算法过程中所用的抽象 Map 任务矩阵进行定义.

定义 6. Map 任务矩阵. Map 任务矩阵 $\boldsymbol{S}=[\boldsymbol{s}_{ij}]_{p\times q}$表示 Map 任务规模以及数据块在 Map 任务中的分发状态.矩阵中元素 $\boldsymbol{s}_{ij}$表示 Map 任务 m_{ij},并且 $\boldsymbol{s}_{ij}$为一个向量$\langle L_{ij},G_{ij},R_{ij}\rangle$,向量内的元素值分别代表本地、机架以及远程的数据量.并且定义若机架中并不存在此节点,则置 $\boldsymbol{s}_{ij}$值为 null.

$\boldsymbol{S}^0$是 Map 任务矩阵的初始状态,每个节点上不存在机架以及远程数据,只有本地数据,即 $G_{ij}=R_{ij}=0$.以 Map 任务矩阵(5)为例,矩阵中行表示同一机架.集群中有 4 个机架以及 13 个节点,每一个节点上只有本地任务.矩阵元素 s_{11}表示第一个机架内的第一个节点有 5 个本地数据块,机架和远程数据块个数都为 0 个.

$$\boldsymbol{S}^0=\begin{bmatrix}\langle5,0,0\rangle & \langle6,0,0\rangle & \langle7,0,0\rangle & \langle8,0,0\rangle\\ \langle3,0,0\rangle & \langle4,0,0\rangle & \langle5,0,0\rangle & \text{null}\\ \langle5,0,0\rangle & \langle5,0,0\rangle & \langle5,0,0\rangle & \langle5,0,0\rangle\\ \langle2,0,0\rangle & \langle3,0,0\rangle & \text{null} & \text{null}\end{bmatrix} \tag{5}$$

因为每个机架所包含的节点个数不同,因此 Map 任务矩阵应为一个锯齿状数组(Jagged Array),即实际中的 Map 任务矩阵应该是参差不齐的.但为了简化算法描述,我们设集群中有 q 个机架,并且每个机架内都有 p 个节点.

定义 7. Map 任务单次分发. Map 任务单次分发是将 MapReduce 作业一次性分配给 Map 节点的过程,每个节点仅获得一个 Map 任务,同时最大化任务并行度.

首先我们按 $\boldsymbol{S}^0$ 初始化任务,但若按 $\boldsymbol{S}^0$ 执行 MapReduce 任务,那么 Map 任务的并行性则仅取决于数据在分布式文件系统的布局(Data Placement),负载均衡的数据布局将有利于提高并行性.在大数据环境,我们很难通过数据的预定义和分类来获得负载均衡的数据布局.因此 Map 任务单次分发算法将 $\boldsymbol{S}^0$转化成 $\boldsymbol{S}^1$,重新调整任务包含的本地、机架和远程数据块个数,保证每个节点的负载均衡.值得注意的是,Map 任务单次分发算法并不会真正地分发任务,而只是通知节点其待处理的数据块所在位置,任务按 $\boldsymbol{S}^0$初始化以后就不会再次分发.同时,易知若 $\forall\boldsymbol{s}_{ij},\boldsymbol{s}_{xy}\in S,\alpha L_{ij}+\beta G_{ij}+\gamma R_{ij}=\alpha L_{xy}+\beta G_{xy}+\gamma R_{xy}$,集群内 Map 阶段的并行性将会达到最佳状态.但由于集群中每个节点的性能不同,而且由于数据特征等其他因素的影响,在 Map 任务的执行过程中,Self-resizing 算法会将 $\boldsymbol{S}^1$转换成 $\boldsymbol{S}^2$,$\boldsymbol{S}^3$直至最终状态 $\boldsymbol{S}^k$.

定义 8. Map 任务 Self-resizing. Map 任务 Self-resizing 是指 Map 任务根据当前的任务矩阵自发地、动态地调整自身规模(待处理的数据块)的过程.节点自调整是一个有效地保证节点间并行度的算法.在每次节点自调整过程中,矩阵 $\boldsymbol{S}^k$会被转换成 $\boldsymbol{S}^{k+1}(k=0,1,2,3\cdots)$,并且在每次转换中,$L_{ij}^k$,$G_{ij}^k$和 R_{ij}^k都会被调整,基于 $\boldsymbol{S}^k$,$\boldsymbol{S}^{k+1}$需要满足 $\delta^k<\delta^{k+1}$,且 $\sum_{i=1,j=1}^{p,q}(L_{ij}^{k+1}+G_{ij}^{k+1}+R_{ij}^{k+1})=\sum_{i=1,j=1}^{p,q}(L_{ij}^k+G_{ij}^k+R_{ij}^k)$.

我们首先描述 Map 任务单次分发算法,即 $\boldsymbol{S}^0$转换成 $\boldsymbol{S}^1$的算法.在 $\boldsymbol{S}^0$中,每个任务仅处理本地数据,若进一步提高并行性,必然需要重新分配数据块,则会存在部分任务处理机架内数据块或远程数据块.为了在提高并行性的同时确保性能,应该尽可能地减少任务处理远程数据块的数据量,让数据块优先在机架内再次分配.因此算法的基本步骤是先

逐一保证每个机架内每个任务的规模相同，即 $\boldsymbol{S}^0$ 的同一行内任务的并行性，再保证机架间的任务并行性，及 $\boldsymbol{S}^0$ 行间的并行性. 由于 $\boldsymbol{S}^0$ 中任务不含机架内和远程数据块，$G_{ij}^0=R_{ij}^0=0$，$|m_{ij}^0|=L_{ij}^0$.

Map 任务矩阵的同一行内任务代表运行在同一机架内的节点. 对于 $\forall i\in[1,p]$，我们首先研究数据块如何在 m_{i1}^0 到 m_{iq}^0 间分布以保证它们规模一致. 不失一般性，我们设 Map 任务 m_{i1}^0 到 m_{iq}^0 按任务规模升序排序，即 $L_{i1}^0=\min(L_{i1}^0,L_{i2}^0,\cdots,L_{iq}^0)$，$L_{iq}^0=\max(L_{i1}^0,L_{i2}^0,\cdots,L_{iq}^0)$；设 $\overline{L_i^0}=\frac{1}{q}\sum_{j=1}^{q}L_{ij}^0$，若 m_{i1}^0 到 m_{iq}^0 中有 t 个规模小于平均规模 $\overline{L_i^0}$ 的任务，则将其统一视为一个虚拟任务 m_{iu}^0，并将剩余任务视为一个虚拟任务 m_{iv}^0，二者的任务规模分别是 $|m_{iu}^0|$ 和 $|m_{iv}^0|$，$|m_{iu}^0|=L_{iu}^0=\sum_{j=1}^{t}L_{ij}^0$，$|m_{iv}^0|=L_{iv}^0=\sum_{j=t+1}^{q}L_{ij}^0$. 这样原问题就转换为“在第 i 个机架中，从节点 m_{iv}^0 移动多少数据块到 m_{iu}^0 才能保证二者规模一致”. 设需要移动的数据块数量为 x，并且需要注意的是，当数据块从别的节点转移，还需要考虑到 I/O 的代价. 因此，令需要移动的数据块 x 为未知数，α、β、L_{ij}^0 以及 G_{ij}^0 为已知数，依据：(1) 任务的数据量守恒，即 m_{iv}^0 本地数据的减少(移出)量等于 m_{iu}^0 数据远程数据的增加(移入)量；(2) 按式(4)还需最大化 m_{iv}^0 和 m_{iu}^0 数据转移的并行度；可以建立一元一次方程组(6). 求解方程得从节点 m_{iv}^0 移动到 m_{iu}^0 的数据块数量 x 的表达式，$x=\dfrac{\alpha(\sum_{j=t+1}^{q}L_{ij}^0-\sum_{j=1}^{t}L_{ij}^0)}{\alpha+\beta}$.

$$\begin{cases}L_{iu}^1=L_{iu}^0,\ G_{iu}^1=x\\ L_{iv}^1=L_{iv}^0-x,\ G_{iv}^1=0\\ \alpha L_{iu}^1+\beta G_{iu}^1=\alpha L_{iv}^1+\beta G_{iv}^1\end{cases}\Rightarrow$$

$$x=\frac{\alpha(L_{iv}^0-L_{iu}^0)}{\alpha+\beta}=\frac{\alpha(\sum_{j=t+1}^{q}L_{ij}^0-\sum_{j=1}^{t}L_{ij}^0)}{\alpha+\beta}\tag{6}$$

因为 $\begin{cases}L_{iu}^1=L_{iu}^0, & G_{iu}^1=\dfrac{\alpha(L_{iv}^0-L_{iu}^0)}{\alpha+\beta}\\ L_{iv}^1=L_{iv}^0-\dfrac{\alpha(L_{iv}^0-L_{iu}^0)}{\alpha+\beta}, & G_{iv}^1=0\end{cases}$.

此时为了保证虚拟节点 m_{iu}^0 和 m_{iv}^0 内部的并行度，对于 $\forall g\in[1,t]$，m_{ig}^0 会从虚拟节点 m_{iv}^0 获得 $\dfrac{\overline{L_i^0}-L_{ig}^0}{\sum_{j=1}^{t}(\overline{L_i^0}-L_{ij}^0)}\times x$ 个数据块，其中 $\dfrac{\overline{L_i^0}-L_{ig}^0}{\sum_{j=1}^{t}(\overline{L_i^0}-L_{ij}^0)}$ 表示节点 g 在虚拟节点 m_{iv}^0 中所占的权重；同理，$\forall g\in[t+1,q]$，m_{ig}^0 会提供 $\dfrac{L_{ig}^0-\overline{L_i^0}}{\sum_{j=t+1}^{q}(L_{ij}^0-\overline{L_i^0})}\times x$ 个数据块给虚拟节点 m_{iu}^0. 同时为了保证迁移的代价最小，即移动的数据块的次数最少，我们规定，获得节点的计算顺序是从 m_{i1}^0 到 m_{it}^0，而提供数据块节点的计算顺序是从 m_{iq}^0 到 $m_{i(t+1)}^0$，这样就能保证 m_{i1}^0 得到较多的数据块，m_{iq}^0 提供较多的数据块，这与实际情况相符.

综上所述，式(7)给出了在保证同机架节点的最大并行度的前提下，同一机架内的节点，在经过调整后，所需要处理的本地、机架和远程数据量.

$$L_{ig}^1=Lr(i,g)=\begin{cases}L_{ig}^0, & g\le t\\ L_{ig}^0-\dfrac{L_{ig}^0-\overline{L_i^0}}{\sum_{j=t+1}^{q}(L_{ij}^0-\overline{L_i^0})}\times\dfrac{\alpha(\sum_{j=t+1}^{q}L_{ij}^0-\sum_{j=1}^{t}L_{ij}^0)}{\alpha+\beta}, & g>t\end{cases},$$

$$G_{ig}^1=Gr(i,g)=\begin{cases}\dfrac{\overline{L_i^0}-L_{ig}^0}{\sum_{j=1}^{t}(\overline{L_i^0}-L_{ij}^0)}\times\dfrac{\alpha(\sum_{j=t+1}^{q}L_{ij}^0-\sum_{j=1}^{t}L_{ij}^0)}{\alpha+\beta}, & g\le t\\ 0, & g>t\end{cases},$$

$$R_{ig}^1=Rr(i,g)=0\tag{7}$$

借助式(7)，同一机架内的任务规模一致，并行度得以保证，但尚未考虑机架间任务规模的均衡性. 我们将一个机架的所有任务假设为一个虚拟任务. 则那么“机架间任务规模一致化问题”就转换成“在同一机架内的任务规模一致性问题”，我们依然可以借助上述方法解决问题. 在 $\boldsymbol{S}^0$ 拥有少于本地数据块的机架会从其他机架任务获取数据块作为自己的远程任务. 不失一般性地设有 c 个少于平均本地数据块的机架为虚拟任务 m_u，拥有多于平均数本地任务的机架为虚拟任务 m_v. 以需要移动的数据块 x 为未知数，α、β、γ 和 L_{ij}^1、G_{ij}^1 以及 R_{ij}^0 为已知数，依据数据量守恒和最大化并行度，可以建立类似于方程组(6)的一元一次方程组，从而计算出 m_v 应提供给 m_u 多少数据块，以及按比例为每个机架的每个任务提供或获得多少数据块. 鉴于描述繁冗，本文仅给出最终结果. 对于 $\forall h,k\in[1,p]$，$\forall g,j\in[1,q]$，$\boldsymbol{S}^1$ 可以根据式(8)计算得出，其中 y 代表集群中节点的平均

数据块数量.

$$L_u=\sum_{k=1,j=1}^{c,q} Lr(k,j),\quad L_v=\sum_{k=c+1,j=1}^{p,q} Lr(k,j),$$

$$G_u=\sum_{k=1,j=1}^{c,q} Gr(k,j),\quad G_v=\sum_{k=c+1,j=1}^{p,q} Gr(k,j),$$

$$x=\frac{\alpha\times(L_v-L_u)+\beta\times(G_v-G_u)}{\alpha+\gamma},$$

$$y=\frac{1}{p}\times\sum_{k=1,j=1}^{p,q} Lr(k,j),$$

$$L_{hg}^1=\begin{cases}Lr(h,g), & h\le c\\ Lr(h,g)-\dfrac{Lr(h,g)-y/q}{\sum_{k=c+1}^{p}\left(\sum_{j=1}^{q}Lr(k,j)-y\right)}\times x, & h>c,\end{cases}$$

$$G_{hg}^1=Gr(h,g),$$

$$R_{hg}^1=\begin{cases}\dfrac{y/q-Lr(h,g)}{\sum_{k=1}^{c}\left(y-\sum_{j=1}^{q}Lr(k,j)\right)}\times x, & h\le c\\ 0, & h>c\end{cases}\tag{8}$$

Map 任务单次分发算法将 Map 任务矩阵由 $\boldsymbol{S}^0$ 转换成 $\boldsymbol{S}^1$,并且 $\boldsymbol{S}^1$会保存到集群中的每个节点,作为 Self-resizing 所依据的初始任务分发表. Map 阶段的任务分发算法伪代码如算法 1 所示.

算法 1. Map 阶段的任务分发算法.

输入:Map 任务矩阵初始状态 $\boldsymbol{S}^0$

输出:Map 任务单次分发结果 $\boldsymbol{S}^1$

```
1.  Enum SOURCE {LOCAL,RACK,REMOTE}
2.  MapDistribution(S^0)
3.  S^1=cloneScheduleMatrix(S^0)
4.  For i=1,2,…,q
5.    Sort(S^1[i]);
6.    t=FindTheMeanOfTheRank(S^1[i]);
7.    For j=1,2,…,t
8.      taskList=FindMoveableTasks(S^1[i]);
9.      For task in taskList
10.       S^1[i][j][RACK]+=1;
11.       S^1[i][task.host][LOCAL]-=1;
12.     End for
13.   End for
14. End for
15. For i=1,2,…,q
16.   Sort(S^1);
17.   t=FindTheMeanOfTheCenter(S^1);
18.   For j=1,2,…,t
19.     taskList=FindMoveableTasks(S^1);
20.     MoveTaskFromRank(S^1[0],taskList)
21.   End for
22. End for
23. return S^1;
```

为了进一步提高并行性,需要执行 Map 任务 Self-resizing 算法. 与 Map 任务单次分发算法不同,Self-resizing 是节点抢占并处理数据块的过程. Self-resizing 算法由任务自发调用,当一个任务处理完所有的数据块且 Map 阶段尚未结束时,这个任务执行节点将不会因等待其他任务而空闲,而是执行 Self-resizing 算法,去争抢其他节点未处理的数据块,对应的,任务矩阵由当前的 $\boldsymbol{S}^k$ 变换为 $\boldsymbol{S}^{k+1}$,直至所有 Map 任务执行结束. Self-resizing 的基本思想是:(1)每个节点都优先处理本地任务;(2)性能高的节点尽可能处理远程任务;(3)性能低的节点尽可能处理本地任务. 借助 Self-resizing,我们可以最小化"所有节点等待一个性能差的节点处理一个远程数据块"这一最差情况发生的概率. 对于某一任务 i,参照当前 Map 任务矩阵,争抢规则如下:

(1)如果任务 i 存在被分配给其他任务的本地数据块,则优先争抢处理这样的数据块.

(2)任务 i 随机争抢同机架任务 j 的一个远程数据块,并且,这个数据块对于任务 i 同样也是远程的数据块.

(3)如果同机架任务不存在远程数据块,任务 i 则随机争抢一个原属于该一机架却被分配给其他机架的数据块.

(4)对于任务 i,如果不存在一个原属于该机架却被分配给其他机架的数据块,那么则从其他机架随机争抢一个远程数据块.

(5)如果对于所有任务都不存在远程数据块,任务 i 则从同机架任务随机争抢一个机架内数据块.

(6)如果对于任务 i 的同机架任务都不存在机架内数据块,任务 i 则随机从远程任务争抢一个机架内数据块.

(7)如果没有远程任务机架内数据块,则随机从同机架的任务内争抢它们的本地数据块.

(8)如果所有同机架的任务都处理完成或正在处理最后一个数据块,任务 i 则随机争抢一个远程任务的本地数据块.

任务 Self-resizing 确保了处理速度快的节点最大概率地处理远程数据块,这提高了 Map 阶段的整体性能. 图 1 是一个任务 Self-resizing 的示意图. 为简

化作图，不妨设集群中存在两个机架，每个机架中有两个节点，节点读取本地、机架内以及远程数据的 I/O 速度之比简化为 1∶1∶2，单次分发和 Self-resizing 的具体步骤如图 1 所示.

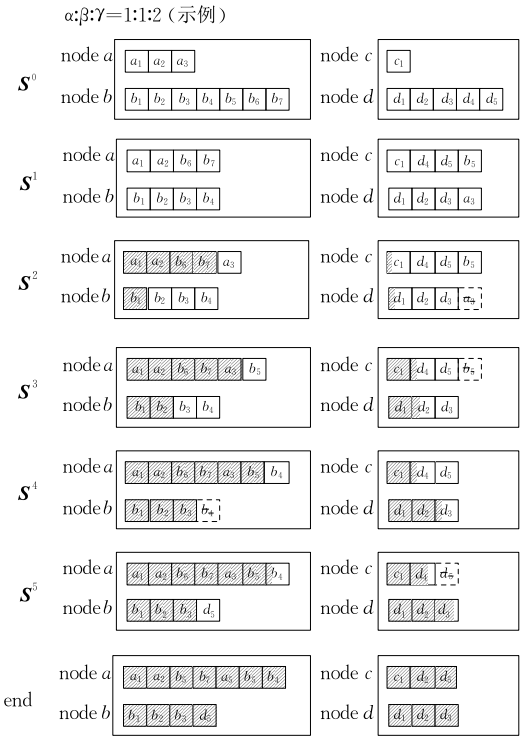


图 1　Map 任务单次分发和 Self-resizing 算法示例

其中，图 1 中包含 1 步单次分发操作，5 步 Self-resizing 操作. 图中的小方块代表数据块，数据块名称标识出它是本地、机架内还是远程数据块. 阴影的方块代表已处理（完全填充）或正在处理（部分填充）的数据块，虚线边框的方块表示被其他节点争抢的数据块. 由于节点的异构性，节点 a 和 b 的性能要优于节点 c 和 d，且节点 a 和 b 包含的本地数据量要大于节点 c 和 d.

6　Reduce 任务分发算法

在 MapReduce 模型中，Map 阶段的数据输入来自于底层分布式文件系统，分配到各个 Map 任务的数据大小是静态的已知数据. 但 Reduce 阶段不同于 Map 阶段，它处理的是 Map 阶段完成后产生的中间结果，这里称为中间数据，不同的 Map 任务产生的数据是动态的，只有在 Map 阶段完成后才能确定数据的分布和每个 Reduce 任务的大小. 传统 MapReduce 框架的 Reduce 任务分发算法首先确定 Reduce 节点的个数并将 Reduce 任务和节点一一对应，随后采用 Hash 算法将 Map 任务产生的中间数据按不同的 Key 值映射到某个 Reduce 任务，Reduce 任务的个数和 Key 与 Reduce 任务之间的映射关系都无法动态修改，因此容易导致 Reduce 任务大小不均衡. 例如，有些 Reduce 任务分配的 Key 其对应大量的数据，而有些 Reduce 任务只需要处理少量数据，如此以来，各 Reduce 任务的完成时间差别较大，并行度很低，产生大量等待能耗. 在极端情况下，可能出现某个 Reduce 分配的数据超出本地磁盘空间，导致任务失败的情况. 基于上述问题，我们提出“Reduce 任务均衡分发算法”.

图 2 是一个 Reduce 端的任务规模不均衡的例子. 图中矩形中每个小图标代表一个键/值对（Key-value Pair），其形状代表 Key. 在 Map 阶段和 Shuffle 阶段结束后，相同形状的图标放入同一个 Reduce 任务中. 由于任务分发的不均衡，Reducer1 的规模最大，是 Reducer2 和 Reducer3 的两倍，任务规模不均衡会导致 Reduce 任务并行度很低.

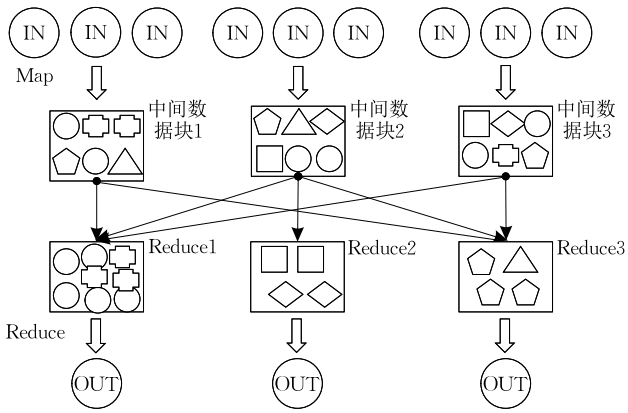


图 2　Reduce 任务规模不均衡的示例

Reduce 任务均衡分发算法的基础是对所有 Key 及其规模的统计，我们采用精确和估计两种方法统计 Key 规模. 其一是数据采样，大多分布式文件系统都提供数据采样接口，因此在 Map 阶段增加一个采样 Map 任务，其处理逻辑与其他 Map 任务相同，处理的数据为输入数据的随机采样，当其他 Map 任务结束时该采样 Map 任务也主动结束. 由于采样的随机性，采样任务获得的“Key 和 Key 规模之间的比值”等同于“所有数据的输出的 Key 和 Key 规模之间的比值”. 采样的方法不会对 Map 阶段性能产生影响，但需要额外的节点参与运算，消耗少量电能. 其二是精确统计，在 Shuffle 阶段之间由

每个 Map 任务向主节点汇报自身的 Key 种类和规模.每个 Map 任务的输出均在本地,按 MapReduce 执行流程,在 Shuffle 阶段开始之前,每个 Map 任务均会按 Key 进行本地排序操作,因此 Map 任务很容易获得 Key 数量和规模,并返回给调度节点,调度节点进行汇总后,即可以把这些 Key 按规模进行分配,该方法仅仅需要较小的代价.后文的实验可以证明对 Key 规模的统计带来的额外代价很小,不会影响能耗优化效果.

Reduce 任务均衡分发算法与装箱问题(Bin Packing)很类似,在装箱问题中,拟将大小各异的物品装入统一规格的箱子中,且寻找一种方法将物品全部装入箱内且使用的箱子数目最少.而 Reduce 任务均衡分发算法是中 Key 类比物品,Reduce 任务类比箱子,箱子的数量为 $p\times q$,箱子的容量未知但相对于物品来说是足够大的,且寻找一种方式使得物品全部装入箱内,并且最小化"箱子占用容量的最大值"(参见式(4)),等价于最小化"最大的 Reduce 规模".可类比为算法"将 w 堆沙子合并为 $p\times q$ 堆且使最大的沙堆最小".

定义 9. Reduce 任务均衡分发.设 Map 输出了 w 种 Key,分别记为 $e_1,e_2,\cdots,e_w$,每个 Key 都有自己的规模,记为 $|e|$. Reduce 任务均衡分发是指将这些 Key 及其对应的 Value 按 Key 的规模分发到 $p\times q$ 个 Reduce 任务中($w\gg p\times q$),使 $\forall i,x\in[1,p]$,$\forall j,y\in[1,q]$,设 $|r_{\max}|=\max(|r_{ij}|)$,且 $|r_{\max}|$ 在所有分发方案中最小.

可证明该问题是一个 NP 难的问题.易知,若存在足够多规模为 1 的 Key,则 Reduce 任务均衡分发的最优解一定为 $|r_{\max}|=\frac{1}{p\times q}\sum_{i=1}^{w}|e_i|$,但该解并不一定存在.本文提出一种类似离线装箱算法的扩展版本,称为最小适合递减算法(Most Minimum Fit Decreasing),简记为 Mid 算法.Mid 算法将 Key 按其规模降序排序,随后从前至后取出一个 Key,并放入当前规模最小的 Reduce 任务中,如此往复直至所有 Key 被分发完毕.不失一般性,设 $e_1,e_2,\cdots,e_w$ 已经按其规模排序.$\forall s\in[1,w]$,我们称第 s 次分配将 e_s 分配给 r_{ij},且满足 $\forall x\in[1,p]$,$\forall y\in[1,q]|r_{ij}|\leq|r_{xy}|$.可知第 1 次分配 e_1,第 w 次分配 e_w.Mid 是一种近似算法,设该问题的最优解为 T^*,通过 Mid 算法获得的解为 T,我们可以证明 $T\leq1.5T^*$.证明如下:

因为易知 $T^*\geq\frac{1}{p\times q}\sum_{i=1}^{w}|e_i|$,又因为 $w\gg p\times q$,令 $m=p\times q+1$;

所以 $T^*\geq2|e_m|$,也即第 1 轮所有任务规模为零时,每个节点都依次放入 e_1 至 e_{m-1},随后再将 e_m 放入某规模最小的任务后,该任务不再放入任何 Key 的临界情况.

因为考虑完成分配后的规模最大的任务 $T=r_{\max}$,对于该任务即将放入最后一个 Keye$_s$ 时,它一定是所有任务中规模最小的;

所以 $T-|e_s|\leq\frac{1}{p\times q}\sum_{i=1}^{s}|e_i|\leq\frac{1}{p\times q}\sum_{i=1}^{w}|e_i|\leq T^*$

又因为 $T^*\geq2|e_{m+1}|$,且 $|e_s|\leq|e_{m+1}|$

所以 $(T-|e_s|)+|e_s|\leq T^*+|e_s|\leq T^*+|e_{m+1}|=1.5T^*$

因此得证:$T\leq1.5T^*$

$1.5T^*$ 是可以证明的 Mid 算法的上界,事实上,参考文献[26],通过更复杂的证明可以得到一个更紧确的上界 $T\leq\frac{4}{3}T^*$.本文略去了证明细节.Reduce 任务均衡分发算法伪代码如下所示.

算法 2. Reduce 任务均衡分发算法.

输入:Reduce 任务列表和集群中节点列表

输出:Reduce 任务均衡分发结果

```
1. MidReduceDistribution(reducerList,nodelist)
2. reducerKeyScaleList=
                getReduceTaskKeyScale(reducerList)
3. Sort(reducerKeyScaleList)
4. For keyScale in reducerKeyScaleList
5.    node=findSmallestScaleNode(nodelist);
6.    node.addReduceTask(keyScale.tasks())
7. End for
```

图 3 给出了一个 Reduce 分发算法的例子.整个过程共分 9 步,左边矩形表示 Map 输出的中间结果,其中的方块代表 Key,且方块的大小表示 Key 的规模,阴影的方块表示已经分发的 Key;右边矩形代表所有 Reduce 任务,本例有 a,b,c 这 3 个 Reduce 任务.可以看出每次都将 Key 放入规模最小的 Reduce 任务中,最终所有任务的规模非常相近.

7 算法复杂度分析

本小节将对本文提出的 Map 任务分发算法和

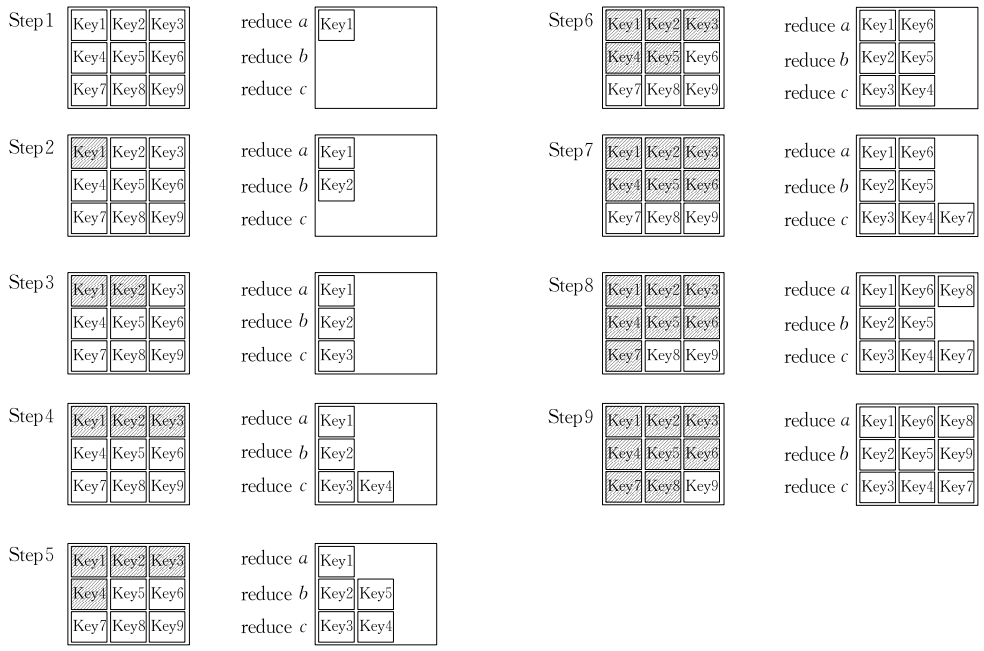


图 3　Reduce 任务均衡分发算法示例

Reduce 任务衡分发算法的时间复杂度和空间复杂度进行分析. Map 任务分发算法包括单次分发和 Self-resizing 两个部分，其前者仅执行一次而后者需要反复执行.

设机架数量为 p，每个机架节点数量为 q，也即机架内 Map 任务的数量. 任务单次分发过程包含机架内任务规模均衡和机架间任务规模均衡两个部分，两者算法是一致的，仅规模不同. 对于机架内算法，需要对所有的任务按照其规模从大到小排序，这个过程的时间复杂度至多为 $O(q^2)$；随后求解式(6)多元一次方程的时间复杂度为常数 $O(1)$；最后是保证两个虚拟节点内部的数据块均衡的过程，由于每个任务均需要提供或接收数据块，因此算法的时间复杂度为 $O(q)$. 机架内的任务单次分发复杂度为 $O(q^2+q+1)$，对于 p 个机架复杂度为 $O(pq^2+pq+p)$.

机架间算法和机架内算法相同，排序的复杂度为 $O(p^2)$，且在均衡过程，无论某机架提供还是获得数据块，机架内的所有任务均需要调整自己的数据块，因此复杂度为 $O(pq)$. 而 Self-resizing 与任务数量无关，每次 Self-resizing 仅涉及一个任务按规则选择另外一个任务的数据块执行，时间复杂度为常数 $O(1)$.

综上所述，Map 任务分发算法的时间复杂度为 $O(p^2+p+2pq+pq^2+1)$.

Reduce 任务均衡分发算法的主要运算代价集中在对 Key 按其规模排序的过程. 设 w 为 Key 的数量，其排序时间复杂度至多为 $O(w^2)$，而随后最小适合递减算法(Mid)将遍历所有 Key，对于每个 Key 查找规模最小的 Reduce 任务，在不做任何遍历优化的前提下算法复杂度为 $O(wpq)$. 综上所述，Reduce 任务分发算法的时间复杂度为 $O(w^2+wpq)$.

由时间复杂度分析可知，本文提出的算法时间复杂度在 $O(n^2)$ 量级，属于较为常见的复杂度，且算法规模与节点数量相同，一个 MapReduce 系统的节点数量会是一个合理大小的数字.

Map 任务分发算法和 Reduce 任务分发算法在执行过程中仅需占用少量的内存空间. Map 任务分发算法的输入输出数据所占用的存储空间为一个 Map 任务矩阵，其空间复杂度为 $O(pq)$. Reduce 任务均衡分发算法的输入输出数据为所有任务的序列，其空间复杂度为 $O(w)$.

8　实验评估

本节将从实验目的、实验环境、实验用例与数据选择、实验过程与结果分析 4 个角度分析，以评价本文提出的任务分发算法.

(1) 实验目的

本实验的主要目的是在 MapReduce 系统中执行不同任务量的 CPU 密集型计算测试用例和 I/O

密集型计算测试用例.通过对比改进任务分发算法和 Hadoop 原生任务分发算法下的两种作业的性能和能耗,从而验证本文提出的任务分发算法的正确性和有效性.

(2)实验环境

我们采用小规模的 PC 机集群环境和 Hadoop MapReduce 框架构建 MapReduce 系统,并且比较了 MR、M^+R、MR^+、M^+R^+ 这 4 种任务分发算法,具体参数描述如表 2 所示.

表 2 MapReduce 系统以及能耗测试方法

项	描述
节点	1 个管理节点(JobTracker/NameNode,0 号机), 12 个数据节点(TaskTracker/DataNode,1~12 号机), 其中 10 个为同构的清华同方超翔 Z900 计算机, CPU 为 Inter Core-i5-2300 2.80GHz, 8GB 内存,1TB 硬盘,千兆网卡; 另 2 个为清华同方超翔 C3001 计算机,Inter Core Pentium(R) 4 3.00GHz,1GB 内存,80GB 硬盘,百兆网卡
操作系统	CentOS 6, Linux 2.6.32 内核
MapReduce 框架版本	Hadoop 1.0.4
任务分发方法	MR:原始 Map 任务和 Reduce 任务分发算法; M^+R:仅优化 Map 任务分发算法; MR^+:仅优化 Reduce 任务分发算法; M^+R^+:优化 Map 任务和 Reduce 任务分发算法;
能耗测量方法	采用文献[27]的方法,每隔 1 秒对集群中的每个节点进行 CPU、磁盘以及 I/O 信息采集
系统信息采集工具	SAR

此外,12 个数据节点放置在两个机架中,每个机架中各有 5 台 Z900 计算机和 1 台 C3001 计算机;机架内节点使用千兆交换机连接,机架间使用百兆交换机连接,以实现节点读取本地、机架以及远程数据的速度差别.

(3)实验用例与数据选择

为保证基准测试集能够代表典型的 MapReduce 作业,本实验选择 Hadoop MapReduce 提供的用例 WordCount① 和 Sort②,两者分别代表 CPU 密集型计算和 I/O 密集型计算[28].此外为了保证实验数据集的可再现,我们随机生成大量字符串作为 WordCount 和 Sort 作业处理的数据集其中串长度为[1,20]的随机数,串的每一位在 26 个英文字母中随机选取.在该数据上执行 WordCount 和 Sort,可以确保 Reduce 输入数据的 Key 种类远远大于 Reduce 节点的数量.我们设置 3 组实验的数据量分别是 3GB 每节点、5GB 每节点和 10GB 每节点,共 12 个数据节点.若按平均串长度为 10 个字符,占 10 字节,那么,数据量最大的作业将会处理 12 亿条随机生成字符串,相对于实验环境属于海量数据.

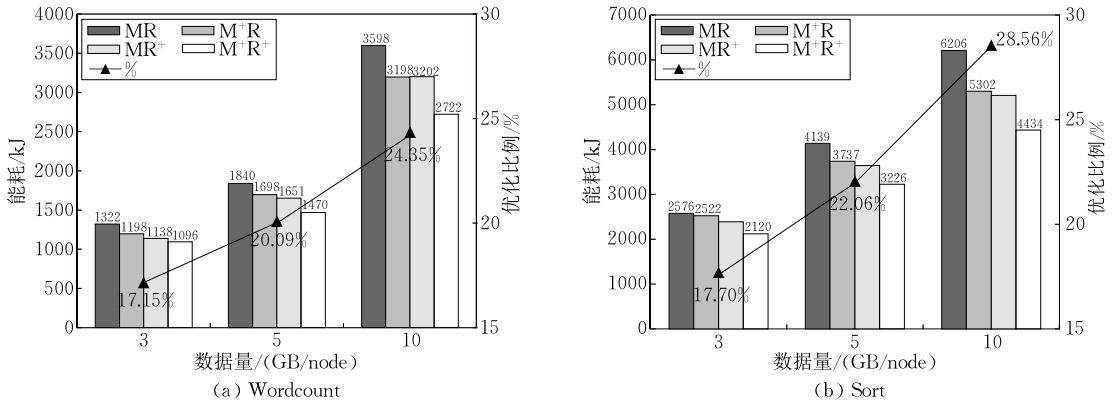
(4)能耗实验与结果分析

首先,我们对比了 MR,M^+R,MR^+ 和 M^+R^+ 这 4 种任务分发策略下的测试用例的能耗,证明本文提出的改进分发算法比 Hadoop MapReduce 任务分发算法较有优势.

从图 4 中可以看出,改进的 Map 任务分发算法和 Reduce 任务分发算法都能够不同程度地优化能耗.由图 4 可以得出如下结论:① 对于 3 种数据集,在 WordCount 用例下,M^+R^+ 较 MR 能耗分别优化了 17.15%、20.09% 和 24.35%;在 Sort 用例下,M^+R^+ 较 MR 能耗优化了 17.0%、22.06% 和 28.56%,优化效果显著;② 不同用例的能耗优化效果相似,但当数据量增加的时候,能耗优化显著,说明任务越多,本文提出的改进任务分发算法的优势越明显;③ MR^+ 对能耗的优化程度比 M^+R 更好,说明对 Reduce 任务分发的改进较 Map 任务分发的改进能耗优化效果明显,因为 MR^+ 仅优化任务分发时间,也即 Map 节点等待任务的时间,在 MR 策略中,Map 任务的并行性通过"反复分发小规模的 Map 任务"这一策略而得以保障,最坏情况下为"节点等待一个较小规模的 Map 任务的执行时间";而 Reduce 端则很可能因任务分发不均衡,导致单个任务执行时间较长,其他 Reduce 任务执行节点因等待该任务而浪费电能.为了更好地证明能耗优化的原因,我们搭建了个仅有单个主节点和单个任务节点的集群环境,在该环境中执行 MapReduce 作业产生的等待能耗,将仅因为"任务节点因等待主节点分发任务而被动空闲",而不会因为"任务不同步而导致节点被动空闲".在此实验中仅有一个任务节点,所以处理作业所消耗的能耗仅与 Map 任务的分发算法有关,与 Reduce 任务的分发算法的无关,因此在图 5 中仅比较了 MR 和 M^+R 的能耗对比.图 5 给出了该用例的能耗测试结果,从图中可以看出,M^+R 能耗仍小于 MR 的能耗,虽然优化比例较图 4 明显下降.图 5 很好地证明节点等待 Map 任务而空闲的存在,进而证明了改进 Map 任务分发算法降低了系统能耗.

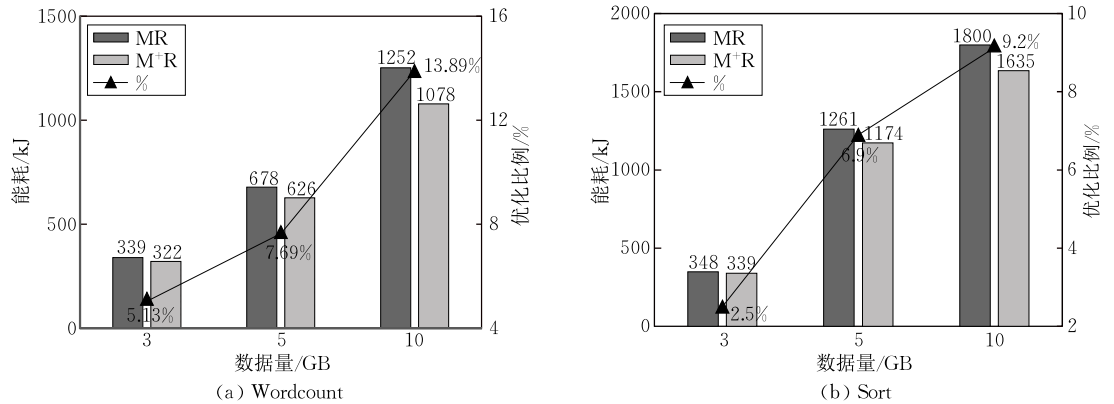
① WordCount program. Available in Hadoop source distribution: src/examples/org/apache/hadoop/examples/WordCount

② Sort program. Available in Hadoop source distribution: src/examples/org/apache/hadoop/examples/sort



(a) Wordcount　　(b) Sort

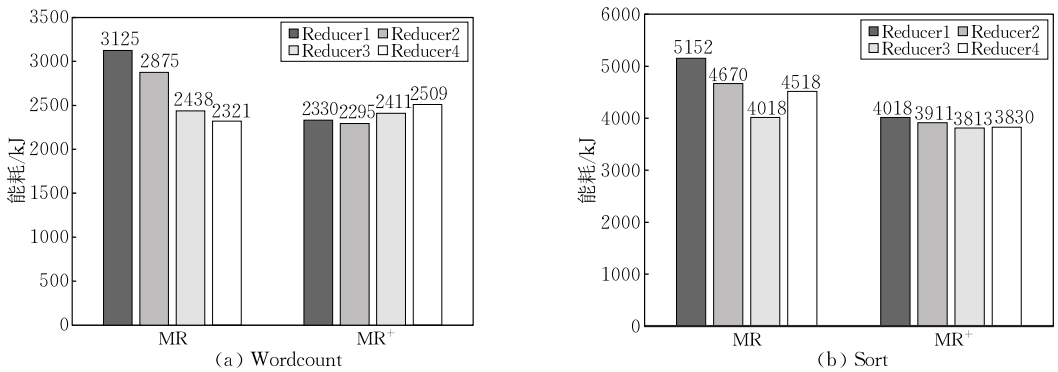
图 4　不同任务分发算法下用例能耗对比



(a) Wordcount　　(b) Sort

图 5　简化 MapReduce 系统下作业的能耗对比

进一步地，为证明 Reduce 任务分发算法的能耗优化效果，我们还考察了 MR 和 MR+ 两者执行过程中每个节点在 Reduce 任务执行阶段的能耗的对比. 我们仅记录了集群中 4 个同构节点的能耗，图 6 比较了 MR 和 MR+ 算法下两个用例的节点能耗，以单独考察 Reduce 任务分发算法的优化效果. 数据集大小为 10 GB 每节点. 从图 6 中可以明显地看出，采用 MR+ 方法分发 Reduce 任务后，每个节点在 Reduce 阶段的能耗近似相等，由于节点是同构的，因此可以推断任务的执行状态，包括 CPU 和 I/O 的使用率、执行时间等等，也近似相同，任务并行性好；与之相反的是，采用 MR 方法分发 Reduce 任务，节点能耗差距较大. 这证明本文提出的 Reduce 任务分发算法能够有效地提高任务的并行性，减少节点间等待，降低能耗.



(a) Wordcount　　(b) Sort

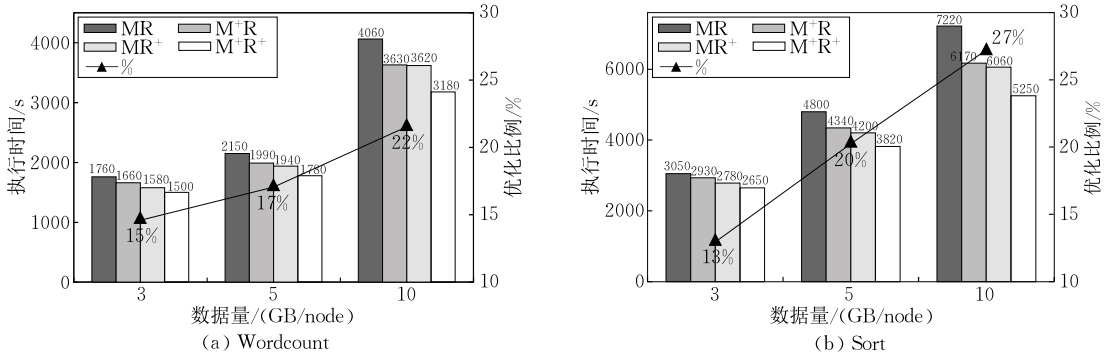
图 6　Reduce 节点能耗对比

(5) 能耗-性能分析

对于一个软硬件整体考虑的系统，电能既可以认为是被硬件直接消耗的，也可以认为是软件间接消耗的. 由于能耗和性能之间的关系，能耗优化研究均伴随着性能的变化. 现有研究对能耗优化和性能优化之间的关系持两种截然相反的观点. 在一些研

究中,能耗优化和性能优化负相关,通常需要损失性能来换取低能耗,这符合硬件的设计思路,硬件工作在高性能状态时功率也高,反之若降低性能则会获得较低的能耗,所以它们之间存在着某种折中关系[29];在另一些文章中则声称能耗优化和性能优化是一致的,通过资源分配和任务调度,使资源充分利用,减少因资源浪费而产生的额外能耗,这种优化方法使更多的有效资源投入到计算,能够同时提高计算性能[30].本文主要采用后一种优化思路.我们提出的能耗优化方法的核心思想是"减少各个节点之间不必要的等待",本文采用能耗作为优化对象,因此将上述"等待"定义为"等待能耗",若采用性能作为优化对象,则上述"等待"可以定义为"等待时间".因为硬件设备在空等时的空载功率是稳定的,而能耗等于功率乘以时间,因此降低"等待能耗"也会同时降低"等待时间",能耗优化伴随着性能优化.但实际上能耗优化效果较性能优化效果略为明显.

图 7 给出图 4 能耗实验对应的作业执行时间对比.从图 7 中可以看出,改进的任务分发算法对性能的优化规律同对能耗的优化基本规律一致.然而,我们在优化等待能耗的同时减少了网络数据读写(Map 阶段),因此减小了网络 I/O 设备(如网卡)的工作时间,降低其工作负载(功率),网络 I/O 设备的能耗降低,进而整体能耗优化率会大于性能优化率.由图 4 和图 7 可知,对于 3 种数据集,在 WordCount 用例下,M^+R^+ 较 MR 能耗和性能分别优化了 17.15% 和 15%、20.09% 和 17%、24.35% 和 22%;在 Sort 用例下,M^+R^+ 较 MR 能耗和性能分别优化了 17.0% 和 13%、22.06% 和 20%、28.56% 和 27%,能耗优化效果略优于性能优化效果.



(a) Wordcount (b) Sort

图 7 不同任务分发算法下用例执行性能对比

9 结论和进一步工作

本文改进任务分发算法来优化 MapReduce 系统能耗,提出 Map 任务单次分发算法、Map 任务 Self-resizing 算法和 Reduce 任务均衡分发算法,不同于其他任务分发算法,我们着重于最小化节点等待 Map 任务分发而造成的被动空闲,以及因任务不同步而造成的节点被动空闲.Map 任务单次分发算法节省了重复任务分发消耗的能源,Map 任务 Self-resizing 算法确保了 Map 任务的最大并行度,而 Reduce 任务均衡分发算法则使 Reduce 任务规模更加均衡.实验结果充分证明上述算法能够有效降低 MapReduce 系统的能耗.

进一步的研究工作包括充分考虑节点处理性能差异对任务分发算法的影响.本文提出的 Map 任务自调整算法可以很好地适应异构集群环境,但节点性能不同会影响 Reduce 任务分发算法的优化效果.一个任务的执行时间与任务大小成正比,且与节点性能成反比.参照定义 9,我们提出对异构集群环境下 Reduce 任务分发算法改进的初步思路:Reduce 规模不仅取决于其所有 Key 的规模,还与节点性能有关,当节点性能高,Reduce 规模大于 Key 规模之和,反之则反.若类比装箱问题,则物品在装入箱子时会根据箱子的特点压缩或膨胀体积.具体细节将在后续研究中陆续展开.

参 考 文 献

[1] Huang Shan, Wang Bo-Tao, Wang Guo-Ren, et al. A survey on MapReduce optimization technologies. Journal of Frontiers of Computer Science and Technology, 2013, 7(10): 865-885 (in Chinese)
(黄山,王波涛,王国仁等. MapReduce 优化技术综述. 计算机科学与探索, 2013, 7(10): 865-885)

[2] Liu Yi, Jing Ning, Chen Luo, et al. Algorithm for processing k-nearest join based on R-tree in MapReduce. Journal of Software, 2013, 24(8): 1836-1851(in Chinese)

(刘义，景宁，陈荦等. MapReduce 框架下基于 R-树的 k-近邻连接算法. 软件学报，2013，24(8)：1836-1851)

[3] Bu Y，Howe B，Balazinska M，et al. HaLoop：Efficient iterative data processing on large clusters. Proceedings of the VLDB Endowment，2010，3(1-2)：285-296

[4] Koomey J. Growth in Data Center Electricity Use 2005 to 2010. Oakland，CA：Analytics Press，2011

[5] Ricardo B，Ramakrishnan R. Power and energy management for server systems. IEEE Computer，2004，37(11)：68-76

[6] Zhang Teng-Teng，Wu Xiao，Li Chang-De，Dong Yun-Wei. On energy-consumption analysis and evaluation for component-based embedded system with CSP. Chinese Journal of Computers，2009，32(9)：1876-1883(in Chinese)
(张滕滕，吴晓，李长德，董云卫. 基于CSP的构件化嵌入式软件能耗分析与评估方法研究. 计算机学报，2009，32(9)：1876-1883)

[7] Chen Li-Qiong，Shao Zhi-Qing，Fan Gui-Sheng. Energy consumption modeling and analysis for distributed real-time and embedded systems. Journal of East China University of Science and Technology (Natural Science Edition)，2009，35(2)：250-255(in Chinese)
(陈丽琼，邵志清，范贵生. 分布式实时嵌入式系统的能耗建模与分析. 华东理工大学学报(自然科学版)，2009，35(2)：250-255)

[8] Song Jie，Li Tian-Tian，Zhu Zhi-Liang，et al. Benchmarking and analyzing the energy consumption of cloud data management system. Chinese Journal of Computers，2013，36(7)：1485-1499(in Chinese)
(宋杰，李甜甜，朱志良等. 云数据管理系统能耗基准测试与分析. 计算机学报，2013，36(7)：1485-1499)

[9] Blanas S，Patel J M，et al. A comparison of join algorithms for log processing in MapReduce//Proceedings of the ACM SIGMOD International Conference on Management of Data. Indianapolis，USA，2010：975-986

[10] Isard M，Prabhakaran V，Currey J，et al. Quiney：Fair scheduling for distributed computing clusters//Proceedings of the ACM SIGOPS 22nd Symposium on Operating System Principles. Big Sky，MT，USA，2009：261-176

[11] Zaharia M，Borthakur D，et al. Job scheduling for multi-user MapReduce clusters. EECS Department，University of California，Berkeley：Technical Report UCB/EECS-2009-55，2009

[12] He Rong-Bo. The Performance Optimization and Improvement of MapReduce in Hadoop [M. S. dissertation]. Beijing University of Chemical Technology，Beijing，2011(in Chinese)
(何荣波. MapReduce 模型在 Hadoop 中的性能优化及改进[硕士学位论文]. 北京化工大学，北京，2011)

[13] Gu Rong，Yan Jin-Shuang，et al. Performance optimization for short job execution in Hadoop MapReduce. Journal of Computer Research and Development，2014，51(6)：1270-1280(in Chinese)
(顾荣，严金双等. Hadoop MapReduce 短作业执行性能优化. 计算机研究与发展，2014，51(6)：1270-1280)

[14] Babu S. Towards automatic optimization of MapReduce programs//Proceedings of the 1st ACM Symposium on Cloud Computing. Indianapolis，USA，2010：137-142

[15] Zaharia M，Konwinski A，Joseph A D，et al. Improving MapReduce performance in heterogeneous environments//Proceedings of the Operating Systems Design and Implementation. San Diego，USA，2008：7

[16] Xie J，Yin S，Ruan X，et al. Improving mapreduce performance through data placement in heterogeneous hadoop clusters//Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing，Workshops and PhD Forum. Atlanta，USA，2010：1-9

[17] Leverich J，Kozyrakis C. On the energy (in) efficiency of Hadoop clusters. ACM SIGOPS Operating Systems Review，2010，44(1)：61-65

[18] Stillwell M，Schanzenbach D，et al. Resource allocation using virtual clusters//Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid. Shanghai，China，2009：260-267

[19] Song Y，Wang H，Li Y，et al. Multi-tiered on-demand resource scheduling for VM-based data center//Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid. Shanghai，China，2009：148-155

[20] Chen Y P，Archana G. To compress or not to compress-Compute vs. IO tradeoffs for mapreduce energy efficiency//Proceedings of the 1st ACM SIGCOMM Workshop on Green Networking. New Delhi，India，2010：23-28

[21] Wirtz T，Ge R. Improving MapReduce energy efficiency for computation intensive workloads//Proceedings of the 2011 International Green Computing Conference and Workshops. Orlando，USA，2011：1-8

[22] Wirtz T，Ge R，et al. Power and energy characteristics of MapReduce data movements//Proceedings of the 2013 International Green Computing Conference. Arlington，USA，2013：1-7

[23] Chen Y，Alspaugh S，Borthakur D，et al. Energy efficiency for large-scale mapreduce workloads with significant interactive analysis//Proceedings of the 7th ACM European Conference on Computer Systems. Bern，Switzerland，2012：43-56

[24] Lang W，Patel J M. Energy management for mapreduce clusters. Proceedings of the VLDB Endowment，2010，3(1-2)：129-139

[25] Wang Shan，Wang Hui-Ju，Qin Xiong-Pai，et al. Architecting Big Data：Challenges，Studies and Forecasts. Chinese Journal of Computers，2011，34(10)：1741-1752(in Chinese)
(王珊，王会举，覃雄派. 架构大数据：挑战，现状与展望. 计算机学报，2011，34(10)：1741-1752)

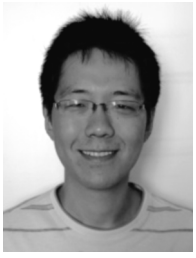
[26] Graham，R L. Bounds on multiprocessing timing anomalies. SIAM Journal on Applied Mathematics，1969，17(2)：416-429

[27] Song Jie, Li Tian-Tian, et al. Study on energy-consumption regularities of cloud computing systems by a novel evaluation model. Computing, 2013, 95(4): 269-287

[28] Huang Sheng-Sheng, Huang Jie, et al. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis//Proceedings of the 2010 IEEE 6th International Conference on Data Engineering Workshops. Long Beach, USA, 2010: 41-51

[29] Xu Zi-Chen, Tu Yi-Cheng, Wang Xiao-Rui. Exploring power-performance tradeoffs in database systems//Proceedings of the 2010 IEEE 6th International Conference on Data Engineering Workshops. Long Beach, USA, 2010: 485-496

[30] Tsirogiannis D, Harizopoulos S, Shah M A. Analyzing the energy efficiency of a database server//Proceedings of the 2010 ACM International Conference on Management of Data. Indianapolis, USA, 2010: 231-242



SONG Jie, born in 1980, Ph. D., associate professor. His research interests include energy-efficient computing, big data storage and management, cloud computing.

XU Shu, born in 1990, M. S. candidate. His current research focuses on energy-efficient computing.

GUO Chao-Peng, born in 1990, M. S. candidate. His current research focuses on massive data computing.

BAO Yu-Bin, born in 1968, Ph. D., professor. His research interest is data warehouse.

YU Ge, born in 1962, Ph. D., professor, Ph. D. supervisor. His research interest is database theory.

Background

The coming of Cloud computing (CC) era has also brought a serious problem of computer power consumption. CC is considered to be a green computing mode, but it does not provide mature solutions to reduce energy consumption. Therefore, we still need an energy-efficient way to achieve real "green computing". At present, there have been some studies on the optimization of energy consumption by treating both hardware and software as a whole system, but these studies are not completely effective for MapReduce system because of its distinctive features (such as Shared-Nothing, migration of operations rather than migration of data, distributed, parallel, high availability). Therefore, it is necessary to introduce an energy consumption optimization solution special for MapReduce System. This paper is a work of such topic; it focuses on the MapReduce system and applications in CC.

This work is mainly supported by the National Natural Science Foundation of China (No. 61202088), named as "Research on Energy-Consumption Optimization Approaches for Cloud Database Systems". The project aims to study an optimization approach of "reducing idle power consumption by reducing node's waiting" in cloud database system; this paper focuses on the "Energy-Consumption Optimized Computing Framework (System)" part of the project. This work is also supported by the Fundamental Research Funds for the Central Universities of China (No. N100704001) and the Research Funds of China Post Doctor (No. 2013M540232). Cooperated with a research team in France, our group has been working in the area of massive data computing and energy-efficient computing for four years and has published many papers.