数据本地性感知的 MapReduce 负载均衡策略

李航晨 秦小麟 沈 尧
（南京航空航天大学计算机科学与技术学院 南京 210016）

摘 要 现有针对 MapReduce 的负载均衡调度的研究均未考虑中间数据的分布特点及网络传输的开销，导致额外的网络传输代价与系统效率的下降。为解决上述问题，提出了一种数据本地性感知的负载均衡策略。充分利用 YARN 中资源管理的新特性，在 Map 阶段对内存数据溢写的同时进行统计以获取数据分布，根据数据分布情况及各节点的计算能力进行任务调度，减少网络传输开销的同时尽量保证各节点的负载平衡。此外，通过引入细粒度分区与分区的自适应分裂策略，进一步提高在数据倾斜时调度策略的性能。对比实验结果表明，提出的负载均衡调度策略能有效提升性能，同时较好地降低网络总开销。

关键词 MapReduce，数据本地性，数据倾斜，负载均衡

中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2015.10.012

Load Balancing Strategy on MapReduce with Locality-aware

LI Hang-chen QIN Xiao-lin SHEN Yao
(College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China)

Abstract Intermediate data distribution characteristics and network traffic overhead are not considered in any existing research on load balancing strategy on MapReduce, resulting in additional network traffic overhead and decrease of system efficiency. To solve this problem, this paper presented a locality-aware load balancing strategy. By taking advantage of the new features of resource management brought by YARN, the strategy can obtain the data distribution when the buffered data are written to local disk. The strategy schedules the reduce tasks according to the data distribution along with the processing speed of each node to decrease network overhead while maximizing load balancing of each node. In addition, to further improve the performance of scheduling strategy with data skew, this paper introduced the strategy of fine-grained partitioning and self-adaption fragmentation. The comparative experimental results show that the presented strategy can improve the performance effectively, and reduce the total network traffic overhead.

Keywords MapReduce, Data locality, Data skew, Load balance

1 引言

互联网、物联网等技术的高速发展，在为人们的生活提供了便利的同时，也带来了数据的爆炸式增长。对于海量数据的分析不仅影响着经济、商业等众多领域的决策，也在人们的日常生活中扮演着越来越重要的角色。作为目前最成功的并行处理模型之一的 MapReduce[1]，由于其良好的鲁棒性与可扩展性，自 2004 年谷歌提出后便得到广泛应用。而作为其最著名的开源实现之一的 Hadoop[2]，已经被诸如 Facebook、Amazon、百度、阿里巴巴等公司大规模部署，用于对海量数据的分析与处理。2013 年，Apache 又推出了新架构的 Hadoop——YARN[3]，从原有版本中分离出了资源管理程序，并且为每个应用配置了单独的应用管理程序，解决了上一版 Hadoop 中资源管理的性能瓶颈，进一步提高了 Hadoop 的并行度。

然而，在 MapReduce 现有的调度策略中并未充分考虑数据本地性，在任务的调度过程中并未充分考虑中间数据的分布特点，导致某些盲目的调度增加了网络负载，降低了集群的效率。在 YARN 基于优先级的调度策略中，只有进行优先级最低的一般的 Map 任务时才会考虑数据本地性特点，而对于 Reduce 任务的调度，则只是简单地从队列中取出第一个待分配的任务分配给当前可用节点，因此可能导致大量的中间文件必须通过网络传输到该节点。此外，作为严重影响 MapReduce架构性能因素之一的数据倾斜在 YARN 中依然存在。当数据倾斜发生时，某些节点需要更长的时间来处理输入的数据，成为集群中的“掉队者”，而其他节点必须保持空闲等待这些掉队者完成处理，导致任务执行时间过长。而 MapReduce 模型所提供的用于解决掉队者的备份任务方法，只是在新的节点上重新执行在掉队者上运行的任务。数据倾斜会导致计算量的不均匀分布，进而造成集群效率下降，与节点本身

到稿日期：2014-10-31 返修日期：2015-01-28 本文受国家自然科学基金项目（61373015，61300052），国家教育部高等学校博士学科点专项科研基金（20103218110017），江苏高校优势学科建设工程资助项目（PAPD），中央高校基本科研业务费专项项目（NP2013307，NZ2013306）资助。
李航晨（1990－），男，硕士生，主要研究方向为云计算，E-mail：lihcnuaa@163.com；秦小麟（1953－），男，教授，博士生导师，主要研究方向为分布式环境的数据管理与安全；沈 尧（1986－），男，博士生，主要研究方向为云计算。

资源的情况无关,因此备份任务并不能解决数据倾斜带来集群效率下降的问题。

针对上述问题,本文提出了一种数据本地性感知的MapReduce负载均衡调度策略。策略结合 YARN 架构中资源管理的新特性,在 Map 阶段溢写数据的同时进行统计,使程序调度器获取分区-节点分布情况,在进行 Reduce 任务调度时充分考虑数据本地性特点,减少网络负载,并通过细化分区与分区动态分裂的方法缓解数据倾斜带来的效率下降问题。

本文第 2 节介绍了相关工作;第 3 节提出了数据本地性感知的 Reduce 任务调度策略;第 4 节针对数据倾斜问题为调度策略添加了抗倾斜的调度策略;第 5 节以实验数据验证了提出的调度策略的优越性;最后总结全文,并指出下一步的研究方向。

2 相关工作

2.1 YARN 中的 MapReduce 调度

在 YARN 中,MapReduce 的任务调度分为 3 个优先级,分别是失败的 Map 任务、达到启动要求的 Reduce 任务和一般的 Map 任务。而在这 3 个优先级的调度策略中,仅有在进行优先级最低的一般 Map 任务的调度时才会考虑数据本地性。对等待调度的 Map 任务,YARN 按照输入数据与 Container 在同一节点、输入数据与 Container 在同一机架,以及输入数据与 Container 不在同一机架 3 个优先级进行调度。然而对于 Reduce 任务,YARN 仅从队列中取出第一个任务,而不考虑 Map 任务之后中间数据的分布,因此可能导致数据混洗阶段因为拉取数据而产生大量网络通信开销,加重系统负载的同时也降低了系统的效率。

YARN 中的 Reduce 任务调度情况如图 1 所示,集群中有 3 个节点分别属于 2 个机架,每个节点上 Map 任务生成了若干已分区的中间文件片。在进行 Reduce 任务调度过程中,在默认的 YARN 调度策略中,如果调度器获取的 Container 的优先级为用于执行 Reduce 任务,则仅从 Reduce 任务队列中取出第一个等待调度的任务,从而可能导致当该 Container 位于 $node_3$ 上时,Reduce 任务队列中的当前任务为分区 P_0,调度器按默认调度策略在 $node_3$ 上执行了分区 P_0 的 Reduce 任务,因此导致 $node_3$ 不得不通过网络从其他节点拉取较大的分区文件。显然不考虑数据本地性的调度策略增加了网络负载,降低了集群效率。

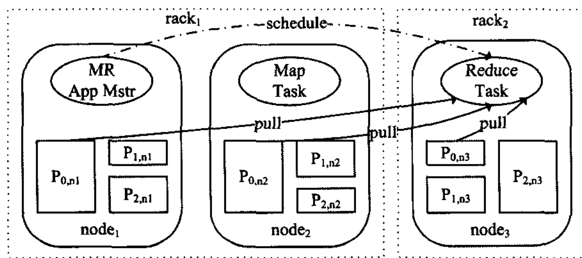

图 1 YARN 中的 Reduce 任务调度

现有的研究中,Ibrahim 等人提出了 LEEN 算法[4],通过分离 Map 和 Reduce 2 个阶段使其异步化,在 Map 阶段之后对中间数据进行扫描以获取数据分布,以此指导 reducer 的任务调度,在考虑数据分布的同时尽量平衡 reducer 的负载;Guo L 等人提出基于已知节点数据分布的 MapReduce 任务调度策略[5],利用已知节点的数据分布和任务优先级进行调度;Polo J 等人提出了根据任务剩余时间动态预测 MapReduce 任务的性能和为该任务调整资源分配的方法[6];唐一韬等人使用 DAG 的方法[7]为 MapReduce 任务进行优先级排序,在选择节点时考虑数据本地性等因素,提高集群利用率。

2.2 数据倾斜

导致 MapReduce 模型产生数据倾斜的原因有 2 种:不平衡的数据输入和不同节点对中间结果的处理代价不同[8]。Reduce 阶段的数据倾斜模型如图 2 所示。系统中 mapper 和 reducer 的数量同为 2,则 Map 端的中间结果被划分为 2 个分区,分别为 $P(1)=\{I(key0),I(key2)\}$,$P(2)=\{I(key1)\}$。由于数据的非均匀分布,使得 Map 端输出的中间数据产生倾斜,$I(key0)$明显大于其他中间结果集。而通过 Hadoop 默认的哈希分区则更加重了这一倾斜情况。在 Reduce 端,每个 reducer 获取一个分区作为输入。$reducer_2$ 在完成了计算任务后,必须等待 $reducer_1$ 的完成。如果 reducer 对数据进行处理的时间复杂度是非线性的,将会进一步放大数据倾斜带来的影响。

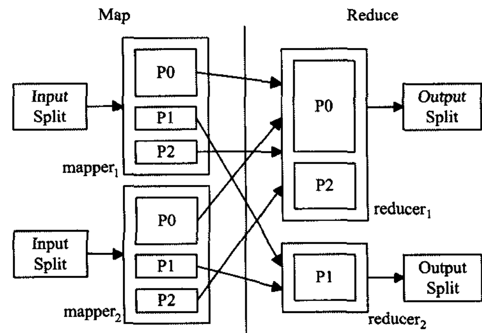

图 2 MapReduce 中的数据倾斜

对 MapReduce 中数据倾斜的研究可以追溯到早期在并行系统中对并行连接[9,10]与并行聚集[11]操作的研究。雅虎的 Pig 系统提供了一种 SkewedJoin[12]的方法,通过对数据采样以估算某个 key 的所有记录数以及所占内存,用户必须在创建 MapReduce 任务时根据文件内容给出针对该应用的分区方法;Kwon 等人设计了 SkewReduce 方法[13],通过采样的方式,由用户提供基于采样值的代价函数,以判断数据分布情况,避免盲目分区造成数据倾斜;Morton 等人设计了 ParaTimer 系统[14]将 MapReduce 的任务转换成有向无环图,以基于关键路径选择的方式对任务的剩余时间进行估计,并给出不同情景下由数据倾斜引发的执行时间的变化的不同估计方法,用于缓解数据倾斜带来的影响;Shi 等人在ParaTimer系统的基础上设计了用于异构环境的 PEQC 估计方法[15],用随机 PERT(project estimate and review technique)图来抽象查询生成的 MapReduce 任务和失败概率,并在此基础上计算出关键路径来进行任务的估计。Hassan M 提出了一种利用分布直方图和随机重分配 key 值的方法[16]缓解在进行连接操作时的数据倾斜问题;Zacheilas N 等人在提出一种缓解倾斜的分区方法的同时还考虑到了节点的计算能力[17],进而提高了 MapReduce 的性能。

上述方法在避免或缓解数据倾斜的同时并未考虑数据本地性特点,可能导致额外的网络传输开销。在考虑数据本地性的方法中,Seo 等人设计了 HPMR 系统[18],通过预采样的方

式对 Map 阶段中间数据的分布进行估计，以此进行 reducer 的分配以减少网络传输；LEEN 算法[4]通过扫描中间文件的方式获取数据分布，以此指导 reducer 的任务调度，利用数据本地性的同时尽量保证 reducer 的负载均衡。然而上述方法在考虑网络传输代价的同时并未考虑资源竞争或异构环境下节点计算能力不同的问题。

3 数据本地性感知的 Reduce 任务调度

在 YARN 中，对于 Reduce 阶段的任务，现有的调度策略只从队列中取出第一个任务进行调度，而没有考虑网络负载与数据本地性特点。针对这一问题，本节提出数据本地性感知的 Reduce 任务调度策略，在调度过程中考虑 Map 阶段中间结果的分布及节点的计算能力，以减少网络传输开销并提高系统的计算效率。

3.1 中间文件分布获取

在 Map 阶段，每个 mapper 将经过 map 函数处理后的中间结果存于内存中，当内存中的数据量达到设定的阈值后则开始溢写过程。当溢写线程启动后，需要对内存中的数据进行基于 key 值的排序、分区。每次溢写过程都会在磁盘上生成一个溢写文件，在 map 函数执行完成后，这些溢写文件会再次进行合并与排序，最终形成一个文件，保存在节点的本地磁盘中。这一过程如图 3 所示。

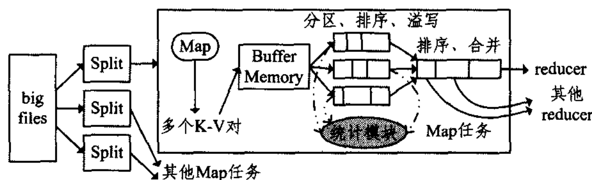

图 3 中间文件分布获取

保存于内存中的 map 函数的输出采用多级索引的存储格式，如图 4 所示，kvoffsets 记录了分区的排序信息，kvindices 则是详细记录了分区号与键、值在 kvbuffer 中的偏移位置，而 kvbuffer 中则实际存储着键值信息。kvoffsets 与 kvindices 均为 int 格式，而 kvbuffer 为 byte 格式。

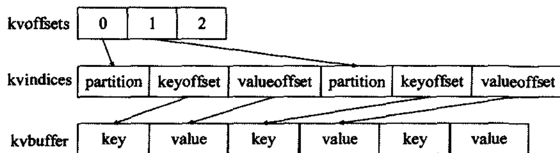

图 4 中间文件存储格式

在 Map 任务的执行过程中，使用上述参数可以计算出当前内存的分区信息与物理大小。利用这一特点，本策略设计了一种利用溢写阶段获取中间文件分布的方法。统计模块在每次 mapper 从内存向磁盘进行溢写的过程中，利用上述参数统计本次溢写过程中每个分区的键值对实际大小，得到一个表示分区号与分区大小对应关系的一维矩阵，并保存在内存中。由于 Map 任务需定期通过 RPC 协议向调度节点汇报任务状态与进度，利用该连接可将该一维矩阵传输至调度节点，调度节点使用一个二维矩阵 P 存储整个集群分区-节点对应关系，其中 $P_{i,j}$ 表示节点 i 上第 j 个分区的大小。设每次通过 RPC 连接收到来自 i 节点的一维矩阵 D，则调度节点将二维矩阵 P 的第 i 行的每个值与矩阵 D 相加，得到更新的 i 节点当前全部中间数据的分布情况。

此外，由于内存缓冲区的大小在提交作业时已设定，各节点在执行计算任务时的内存缓冲区大小一致。根据各节点内存溢写过程的间隔，可以计算出该节点中间数据的生成速度，即对输入文件的处理速度。尽管由于 Map 阶段与 Reduce 阶段各自处理过程不同，使得节点处理 Map 阶段的速度与处理 Reduce 阶段的速度不具有纵向可比性，但仍可用于节点间的横向比较，来估计不同节点的处理速度。各 mapper 同样利用 RPC 协议建立的连接将处理速度 v 报告给调度节点，调度节点根据来自同一节点多次 RPC 连接传输的速度值计算该节点数据处理的平均速度，并将最终的结果保存于一个一维矩阵 V 中，V_i 表示节点 i 的数据处理速度。

3.2 数据本地性感知的任务调度策略

YARN 中取消了 MRv1 中 slot 的概念，而由 ResourceManager负责为应用程序分配资源。对于单个应用而言，应用的调度程序 ApplicationMaster 每次从系统获得在指定节点上一组指定大小的资源，而由调度程序根据资源带有的优先级信息具体决定在此节点上运行何种任务。

因此在不考虑其他限制条件的前提下，在进行 Reduce 任务的调度过程中，执行在该节点上拥有最大中间文件的某个分区的 Reduce 任务将最大限度地减少通过网络传输的数据量，有效利用数据本地性的特点，降低系统负载。然而在实际情况中，考虑到节点间计算能力的不同，以及当前节点上的已有负载，仅以减少网络传输为优化目标考虑数据本地性 Reduce 任务调度可能产生盲目的调度，使得任务总执行时间不会得到明显改善。

利用 3.1 节中所述的方法可以使 MapReduce 应用的调度程序 MRAppMaster 获取已生成中间文件的分区-节点分布情况以及各个节点的平均处理速度，同时各节点的物理分布情况及网络带宽可以事先获取。调度程序在获取一个在指定节点上的 Container 资源后，可以根据上述参数计算得到每个分区文件传输到该节点的传输时间以及该分区在此节点上的执行时间，并从中选择传输时间与执行时间和最小的分区，将其分配给当前节点执行。从集群角度来看，数据本地性感知的 Reduce 任务调度实质是确定一种分区到节点的多对一映射关系，以最小化数据传输与任务执行时间之和。

定义 1 设 p 表示中间结果的分区数，n 表示集群中节点数目，$p_{i,m}$ 表示分区 i 在节点 m 上的中间文件，v_m 表示节点 m 的处理速度，系统带宽为 b，则节点 m 上分区 i 的传输时间与执行时间之和可以由式(1)表示：

$$T_m = T_{transmit} + T_{execute} = \max_{1 \leqslant j \leqslant n, j \neq m} p_{i,j}/b + \sum_{k=1}^{n} p_{i,k}/v_m \quad (1)$$

根据式(1)可将上述问题转化成如何将 p 个分区分配给 n 个节点，使得每个节点上的 T 最小，即寻找空间为 p^n 个可能解的最优解问题。为解决上述问题，本策略引入贪心算法，每次为分区选取传输时间与计算时间之和最小的节点作为执行节点，以求得局部最优解。通过这样的方法，使最终的调度策略较好地平衡数据传输量与任务计算时间，在考虑数据本地性的同时，尽可能提升计算效率。

数据本地性感知的调度策略执行过程如下：

(1)调度节点收集每次 RPC 连接传输来的分区分布信息，与之前已获得的数据进行汇总，得到节点-分区号-分区大

小的对应关系的二维矩阵 D,同时监视当前作业的完成情况。

(2)当达到 Reduce 阶段启动条件时,调度节点暂不进行 Reduce 任务的调度,而首先利用二维矩阵 D 生成一张分区-节点匹配表。分区-节点匹配表的实质是考虑了分区选取传输时间与实际计算时间后的分区与实际执行节点的映射关系。

(3)利用分区-节点匹配表,调度节点在每次获得一个位于指定节点上的 Container 后,通过查看匹配信息,选取下一个映射在该节点上的 Reduce 任务进行调度。当 Reduce 任务执行完成,调度节点从匹配表中将对应分区信息删除。

数据本地性感知的调度策略的核心在于调度节点利用收集的分区分布信息生成分区-节点匹配表以指导后续任务的调度,生成匹配表的贪心策略如下:

(1)对于当前分区 i,查看二维矩阵的第 i 列获取该分区的中间文件在每一节点上的分布情况,结合系统带宽 b 与表示节点计算速度的一维矩阵 V 计算每个可用节点上该分区文件的传输时间与执行时间,选出两者之和最小的节点,进入步骤(2)。

(2)将对应分区的 Reduce 任务标记分配给该节点,查看该节点当前任务,如果当前任务数不大于 $\lceil p/n \rceil$ 则直接转向步骤(3)。否则将该节点加入黑名单,不再为该节点分配任何新任务,并转向步骤(3)。

(3)从待调度列表中除去当前分区,如果列表不为空,则取出下一分区并转向步骤(1),否则生成结束。

定理 1 在 p、n 取整的情况下,上述调度策略可以达到终止条件。

使用反证法证明上述定理。假设策略无法达到终止条件,则存在至少一个分区,使调度策略无法为其分配节点。当 $p<n$ 时,显然此时系统中尚有空闲节点可以进行调度,假设不成立;当 $p\geqslant n$ 时,若满足假设条件,则此时每个节点上的分区数均大于 $\lceil p/n \rceil$。此时系统中的分区总数为 $\lceil p/n \rceil * n+1>p$,显然与已知条件分区数量为 p 矛盾。综上,假设不成立,不可能存在一个分区,使调度策略无法为其分配节点,即上述调度策略可以达到终止条件。

分区-节点匹配表的生成算法如算法 1 所述。

算法 1 生成分区-节点匹配表策略的贪心算法

输入:分区-节点分布情况 $p_{i,n}$, $i=1,\cdots,p$,节点数目 n,处理速度 v_n,系统带宽 b

输出:分区-节点匹配信息 M

1. $P=\{1,\cdots,p\}$, $N=\{1,\cdots,n\}$
2. while $P\neq\emptyset$ do
3. q←get a task from P//从分区列表中取出一分区
4. $s\leftarrow\min\limits_{l\in n}(\max\limits_{1\leqslant j\leqslant n,j\neq l} p_{q,j}/B+\sum\limits_{1\leqslant k\leqslant n}^{n} p_{q,k}/v_l)$//为其分配一节点
5. M(q)←s//更新分区-节点匹配
6. if task amount on $s>\lceil p/n \rceil$
7. $N\leftarrow N\backslash\{s\}$
8. end if
9. $P\leftarrow P\backslash\{q\}$
10. end while
11. return M

其中,$p_{i,n}$ 与 v_n 即为 3.1 节中得到的分别表示分区-节点关系的二维矩阵 P 和节点处理速度的一维矩阵 V,系统的带宽为 b。

3.3 Reduce 阶段的启动时间

在 YARN 中,当 Map 任务的完成数目达到一定的比例,系统将启动 Reduce 阶段的执行,并为准备好的 Reduce 任务分配资源。在本调度策略中,Reduce 阶段开始时间的设计将对系统的效率产生影响:

(1)如果 Reduce 阶段开始时间过早,由于 Reduce 任务必须等待所有的 Map 任务完成,才能执行用户创建的 reduce 函数,因此得到资源并启动的 reducer 只能停留在数据准备阶段,无法执行计算任务,浪费了系统的资源,同时可能导致其他 Map 任务因无法得到足够的资源而被挂起,延长了系统的计算时间,降低了系统的效率。此外,在引入了 Map 阶段的 key 值分布统计策略后,如果 Reduce 阶段开始时间过早,调度模块无法从足够多的 Map 任务中获取 key 值分布信息,可能导致出现盲目的 Reduce 任务调度,进而影响调度策略的准确性。

(2)如果 Reduce 阶段开始时间过晚,虽然可以保证先开始的 Map 任务有足够的资源,并且可以从足够的 Map 任务中获取 key 值分布信息以便做出准确的调度,然而过晚地开始 Reduce 任务的调度将导致大量等候调度的 Reduce 任务在短时间内同时开始执行,每个 Reduce 任务都将与所有 Map 任务建立连接以拉取数据,将导致短时间内网络负载的剧烈增长,增加了系统的负载压力。此外,如果在所有 Map 任务完成后,Reduce 的调度依然没有完成,将使得系统处于等待状态,降低了系统的效率。

因此,Reduce 阶段的开始时间的设计将影响系统效率,有关 Reduce 阶段的启动时间对系统效率的影响见第 5 节实验部分。

4 抗倾斜的 Reduce 任务调度策略

在数据均匀分布的情况下,使用 3.2 节中提出的数据本地性感知的调度策略可以有效降低网络负载,提高系统效率。然而实际情况下,调度策略使用的贪心算法往往无法取得全局最优解。另外,在任务的实际执行过程中,可能受节点资源竞争、reduce 函数计算时间复杂度等影响,导致 reduce 任务的实际执行时间与估计时间出现较大差异,不同节点上任务的完成时间不同,因此使得系统效率仍将受倾斜的影响。针对上述问题,本节首先在 3.2 节所提调度策略的基础上添加细粒度的分区划分,进一步提升数据本地性感知调度策略的性能,然后针对 Reduce 任务的实际计算倾斜问题,添加结合自适应分裂抗倾斜的 Reduce 任务再调度策略,并对该策略进行说明和分析。

4.1 细粒度的分区划分

在 MapReduce 模型中,Map 阶段的分区结果直接影响 Reduce 阶段任务的负载均衡。在数据分布未知的情况下,如果 Map 阶段的分区过少,导致分区数目 p 少于系统设定的 reducer 数目 r,将使得某些 reducer 启动后无法获取输入,降低了系统效率,因此在 MapReduce 模型中分区数目的下限应不低于系统设定的 reducer 数目。随着分区数增多,当 p 大于 r 时,系统将获得一定的对任务进行调度的空间,有利于系统的负载均衡,然而 p 的上限应不大于中间结果 key 值的数目 K,否则将产生无意义的空白分区。

根据式(1),节点 m 上任务执行时间可由下列公式估算:

$$T_{execute_m}=\sum_{1\leqslant k\leqslant n}^{n}p_{i,k}/v_m, i\in\{j|M(j)=m\} \quad (2)$$

设 $r_m=\sum_{1\leqslant k\leqslant n}^{n}p_{i,k}, i\in\{j|M(j)=m\}$ 表示节点 m 上的获取的分区总大小,则系统的执行时间和倾斜度可以分别由下列公式估算:

$$T_{cluster}=\max_{1\leqslant i\leqslant n}(\frac{r_i}{v_i}) \quad (3)$$

$$T_{skew}=\max_{1\leqslant i\leqslant n}(\frac{r_i}{v_i})-\min_{1\leqslant j\leqslant n}(\frac{r_j}{v_j}) \quad (4)$$

随着 p 数目的增多,系统用于调度的自由度增加,数据本地性感知调度策略所表现出的性能越来越好,使得 $\lim_{p\to K}|\max_{1\leqslant i\leqslant n}(\frac{r_i}{v_i})-\min_{1\leqslant j\leqslant n}(\frac{r_j}{v_j})|\to 0$。根据式(4)可得,此时系统倾斜度降低,系统的负载平衡程度提升。

有关分区划分对 Reduce 阶段任务的负载均衡的影响如图 5 所示。同图 2 对比,显然由于 Map 阶段生成的分区变多,系统可用于调度的自由度增大,使得系统倾斜度降低。

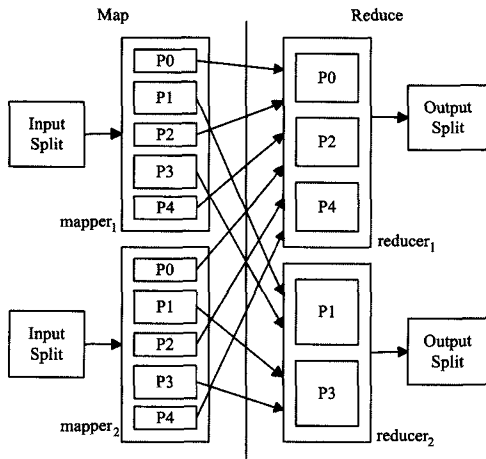

图 5 分区划分对 Reduce 阶段负载均衡的影响

然而随着分区的划分增多,需要从 mapper 传输到redu-cer 的中间文件数量将变得多而琐碎,而就网络传输而言,显然传输少量大文件的代价要远小于传输大量小文件。因此在划分分区的过程中需要考虑分区数目增加带来的额外网络传输开销的问题。

综合上述两个因素,本节为数据本地性感知调度策略添加细粒度的分区划分策略。鉴于算法 1 的实质是确定了从分区结果与 Reduce 任务实际执行节点的映射关系,首先 Map 阶段直接以 key 值数目作为分区数进行划分,以最大程度地提升调度节点在确定分区-节点映射关系时的自由度;之后 reduce 任务在进行数据拉取时可利用算法 1 生成分区-节点匹配表,利用 mapper 建立的连接将所有已生成的、根据分区-节点映射关系属于 reduce 任务所在节点的中间数据全部拉取至本地磁盘上,减少网络通信次数。添加细粒度分区划分的数据本地性调度策略如下:

(1)在 Map 阶段,使用自定义的哈希分区方法:$Hash(key)=key$ MOD $keyNum$,使 map 任务的每个中间 key 值自成一个分区。

(2)当达到设定的 Reduce 阶段启动条件时,利用算法 1 生成 key 值-节点匹配表,调度节点利用之后的 RPC 连接将 key 值-节点映射关系回传至各个节点,并继续根据数据本地性调度策略进行 Reduce 任务调度。

(3)当某个 Reduce 任务与刚完成的 mapper 节点建立连接后,根据 key 值-节点映射关系将该 mapper 上所有属于 Reduce 任务所在节点的数据一次拉取至本地磁盘,并在节点的控制程序中进行标记。当该节点的其他 Reduce 任务需要该 mapper 生成的中间数据时,可直接从节点的本地磁盘读取并处理。

融合了细粒度分区划分的 Reduce 任务调度策略通过细化分区,极大提高了系统对 Reduce 任务进行调度的自由度,有利于系统的负载均衡。同时利用 key 值-节点映射关系,通过与 mapper 的一次连接将所有需要在当前节点处理的数据一次拉取,解决了分区过细导致的传输次数增多、网络开销增加的问题。

4.2 分区的自适应分裂

由于生成分区-节点匹配表的贪心算法只能生成局部最优解,因此在不同节点上的负载仍可能存在差异,并且在任务实际计算过程中可能由于计算复杂度、节点资源竞争、节点故障等原因使得任务的实际计算时间与估计时间出现较大差异,由此导致倾斜。先执行完成的节点必须保持空闲等待其他节点的完成,降低了集群的利用率。

针对上述情况,本节利用算法 1 得到的分区-节点映射信息,提出延迟的分区自适应分裂调度方法,在 Reduce 任务的执行过程中,对负载较大的节点进行分裂后再调度,以平均集群负载,减少空闲节点,提高系统整体资源利用率。

由于在任务计算过程中,每个节点都将各自的全部可用资源用于计算任务,则从整个集群角度来看,只要每个节点上都运行着计算任务,整个集群就处于一个较高的资源利用率。因此自适应分裂策略使用延迟的分裂方法,当集群中有节点完成计算任务,变为空闲时才进行自适应分裂调度。通过这样的调度策略,可以在不影响集群正常任务的同时,充分利用每个节点的资源,提高集群的效率。

利用算法 1 中的本地性感知调度策略生成的匹配信息,本节提出的自适应分裂的调度策略分为如下 4 步:

(1)利用算法 1 生成 key 值-节点匹配表,调度节点根据匹配表计算得到每一节点上全部任务总量。

(2)当调度程序获取一个新的 Container 后,首先在匹配表中查找当前 Container 所在节点的剩余任务信息。如果任务信息不为空,则取出一个 Reduce 任务进行计算。当 Reduce 任务计算完成后,调度程序在匹配表中删除对应任务;如果当前节点的剩余任务信息为空,则启动自适应分裂的调度。

(3)调度程序根据当前匹配表上记录的剩余分区与节点的映射信息,选取剩余分区大小最大的节点作为目标节点,将目标节点上剩余的分区文件以分区为单位分为 2 份。由当前节点向目标节点发起一次 RPC 连接,拉取其中一份文件作为当前节点新的输入。

(4)修改匹配表,将拉取走的分区信息从目标节点删除,并添加至当前节点。从当前节点的匹配信息中取出一个任务,交由 Container 执行。

分区自适应分裂的调度策略实现过程如算法 2 所述。

算法 2 自适应分裂的调度算法

输入：根据分区-节点映射关系得到的节点 n 的任务队列 W_n

输出：更新后的节点任务队列 W_n

1. i←getNodeID(Container)//获取当前 Container 所在节点
2. while $W_i \neq \emptyset$ do//如果当前节点有任务则继续执行
3. task←get a input from W_i
4. execute the reduce function on task
5. $W_i = W_i \setminus \{task\}$
6. end while
7. m←getID($\max_{1 \leq i \leq n} W_i$)//查找当前负载最大节点
8. get the remain input on m and divide it by partition
9. s←get a input from m to i//从选定节点拉取数据
10. $W_m \leftarrow W_m \setminus \{s\}$
11. $W_i \leftarrow W_i \cup \{s\}$//更新两个节点的任务队列信息
12. return W_i

其中，W_n 为根据算法 1 的分区-节点匹配表获得的当前节点 n 的全部剩余 Reduce 任务列表。

分区自适应分裂的调度策略作为 3.2 节中数据本地性感知调度策略的补充，提升了集群计算资源的利用率，避免了空闲节点的出现，进而提高了数据本地性感知调度策略在由外部原因导致 Reduce 阶段出现实际倾斜时的性能。

5 实验结果与分析

5.1 实验准备

为了验证本文所提调度方法的有效性，我们对其进行了实验分析。实验环境为 Hadoop 2.2.0，硬件设备为 8 台节点，具体配置为 2.4GHz 的四核 CPU，8GB 的内存；同时为验证调度策略在节点计算能力不均等时的效率，我们还配置了异构环境，硬件设备为 3 台节点，其中一台使用 3.3GHz 的四核 CPU，4GB 内存，另外两台使用 2.4GHz 的八核 CPU，8GB 内存，实验平台同样为 Hadoop 2.2.0。实验所采用的测试方法为分别利用 Hadoop 进行 WordCount 和 Grep 计算，用于反映在不同应用场景下调度算法的性能，并分别与默认 Hadoop 进行了对比。

实验所用数据摘自英文维基百科。为验证调度算法的抗倾斜性，对实验数据进行了处理，增加了其倾斜度。由于在同构环境与异构环境下的实验数据量不同，为使实验结果具有可比性，以 Hadoop 的运行时间为标准，对实验结果进行了标准化处理。此外，为验证调度算法对网络负载的影响，在默认 Hadoop 中同样添加了对中间数据分布的统计，区分 Reduce 任务调度过程中的本地数据与通过网络拉取的数据，用于估算网络负载。

为了更加全面地反映提出的调度策略性能，将其与文献[4]所提出的 LEEN 方法进行了对比。虽然 LEEN 方法基于 Hadoop 的早期版本，即 MRv1，但由于 YARN 平台上 MapReduce的实现细节与 MRv1 相同，且完全兼容 MRv1 的 API，因此我们在 YARN 平台上重新运行了 LEEN 方法，并从计算时间与网络负载 2 个方面与其进行了对比。

5.2 Reduce 阶段启动时间对性能的影响

Reduce 阶段开始时间对调度算法性能的影响结果如图 6 所示。当 Map 阶段任务数完成百分比大于设定的参数时，系统开始 Reduce 阶段的执行。从图上可以看出，随着 Reduce 阶段启动时间的推迟，调度程序获取的 key 值分布信息逐渐充分，据此作出的调度策略的准确性也逐渐提高，调度算法的性能也逐渐提升。但当 Reduce 阶段开始过晚时，调度算法的性能急剧下降，原因是部分节点在完成 Map 任务后等待调度程序的任务调度而未执行任何任务，因此降低了系统的效率，并导致调度算法性能下降。此外，通过图 6 还可得出，WordCount 的最佳 Reduce 阶段开始时间要晚于 Grep，对其工作流程分析可以得出，WordCount 作业的 Reduce 阶段只接收 key 值和其对应的计数值，对应的 reduce 函数只需遍历计数值求和。一旦 Map 任务完成，Reduce 阶段便很快结束，因此在完成 Map 任务后节点空闲的可能性降低。

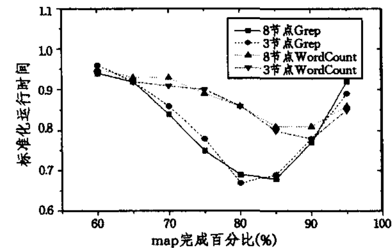

图 6 Reduce 阶段开始时间对调度性能的影响

5.3 调度策略对网络负载的影响

图 7 展示了数据本地性感知的调度策略对网络负载的影响以及与 Hadoop 的对比，图 8 展示了调度策略与 LEEN 方法对网络负载影响的对比。通过实验结果可以得出，考虑数据本地性的调度策略能够有效降低通过网络传输的数据总量，调度策略的平均网络传输量仅略高于异步化 Map 和 Reduce 两阶段从而获得中间文件精确分布信息的 LEEN 方法。此外，调度策略对于 Grep 的优化效果明显优于 WordCount，原因同样是 WordCount 作业的 Reduce 阶段需传输的总数据量较少。另外从实验结果还可得出，在实验所用的贪心策略下，同构和异构环境对网络负载的影响不大。

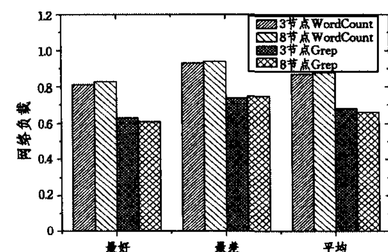

图 7 调度策略对网络负载的影响

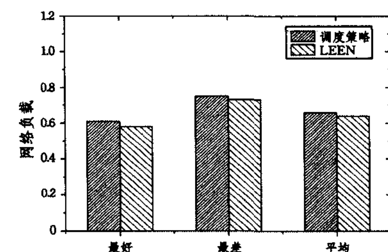

图 8 调度策略与 LEEN 的网络负载对比

5.4 对数据本地性感知调度策略的性能评估

图 9 展示了引入调度算法的 WordCount 和 Grep 计算时间，并通过标准化处理与 Hadoop 进行了对比。通过实验可以看出，引入调度算法后，应用的执行时间明显减少，并且在

采用相同的调度策略的情况下，对 Grep 作业进行调度后的性能要优于对 WordCount 作业进行调度后的性能。除了前述 WordCount 作业的 Reduce 阶段需传输的总数据量较少导致调度算法的优化效果不显著外，另一原因是 WordCount 的 reduce 函数较简单，因此 Reduce 阶段的总时间较短，抗倾斜调度策略未能起到很好的作用。反观 Grep 作业，由于对字符串的匹配计算需要 2 轮 MapReduce 任务，因此在一轮任务完成后需要将大量匹配的字符串和其对应频率写入本地文件，因此 Reduce 阶段的时间较长，调度策略可以起到较好的作用。此外，从实验结果还可看出，在异构环境下调度策略的性能要优于在同构环境下的性能。

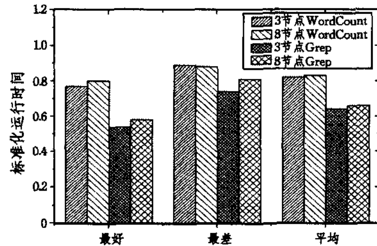

图 9　引入调度策略后 WordCount 和 Grep 执行时间

图 10 展示了本文所提出的调度策略与文献[4]中提出的 LEEN 方法执行时间的对比，采用的测试方法为在同构环境下进行 Grep 计算。从实验结果可以看出，本文所提出的调度策略平均性能要优于 LEEN 方法，原因在于在 MapReduce 模型中，Reduce 阶段可以与 Map 阶段并行进行，虽然 reduce 函数的执行必须等待所有 map 任务结束，但提前开始的 reduce 任务可以提前从已完成的 map 任务处拉取数据进行准备，减少 Map 阶段结束后中间数据传输所用的时间，而 LEEN 方法使用了异步的 Map 与 Reduce 阶段，虽然提高了本地性调度的精确度而减少了网络传输，却因为要在 Reduce 阶段开始时集中传输全部的中间数据而影响了整体的性能。

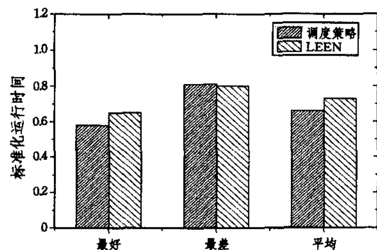

图 10　调度策略与 LEEN 的执行时间对比

结束语　Hadoop 中默认的调度策略并未充分考虑数据本地性特点。本文针对上述问题，提出了一种考虑中间文件数据本地性的负载均衡调度方法。该方法充分利用 YARN 的新特性，减少了网络传输开销。通过细化分区和自适应的分区分裂，提高了算法在数据倾斜情况下的性能。实验证明，本文提出的算法提高了 Hadoop 的处理性能。但是，本文调度算法的应用场景有限，下一步将考虑大规模部署下的容错等问题，研究更具普适性的 Hadoop 负载均衡调度策略。

参 考 文 献

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

[2] Apache Hadoop [EB/OL]. http://hadoop.apache.org, 2014

[3] Vavilapalli V K, Murthy A C, Douglas C, et al. Apache hadoop yarn: Yet another resource negotiator[C]//Proceedings of the 4th annual Symposium on Cloud Computing. ACM, 2013

[4] Ibrahim S, Jin H, Lu L, et al. Handling partitioning skew in MapReduce using LEEN[J]. Peer-to-Peer Networking and Applications, 2013, 6(4): 409-424

[5] Guo L, Sun H, Luo Z. A data distribution aware task scheduling strategy for mapreduce system[M]//Cloud Computing. Springer Berlin Heidelberg, 2009: 694-699

[6] Polo J, Carrera D, Becerra Y, et al. Performance-driven task co-scheduling for mapreduce environments[C]//Network Operations and Management Sympo-sium (NOMS), 2010 IEEE. IEEE, 2010: 373-380

[7] 唐一韬，黄晶，肖球. 一种基于 DAG 的 MapReduce 任务调度算法[J]. 计算机科学，2014，41(6A)：42-46，51
Tang Yi-tao, Huang Jing, Xiao Qiu. Task Scheduling Algorithm for MapReduce Based on DAG[J]. Computer Science, 2014, 41(6A): 42-46, 51

[8] Dhawalia P, Kailasam S, Janakiram D. Chisel: A Resource Savvy Approach for Handling Skew in MapReduce Applications[C]//2013 IEEE Sixth International Conference on Cloud Computing (CLOUD). IEEE, 2013: 652-660

[9] Dewitt D J, Naughton J F, Schneider D A, et al. Practical skew handling in parallel joins[C]//Proceedings of the 18th International Conference on Very Large Data Bases. 1992: 27-40

[10] Poosala V, Ioannidis Y E. Estimation of query-result distribution and its application in parallel-join load balancing[C]//VLDB. 1996: 448-459

[11] Shatdal A, Naughton J F. Adaptive parallel aggregation algorithms[J]. ACM SIGMOD Record. ACM, 1995, 24(2): 104-114

[12] Gates A F, Natkovich O, Chopra S, et al. Building a high-level dataflow system on top of Map-Reduce: the Pig experience[J]. Proceedings of the VLDB Endowment, 2009, 2(2): 1414-1425

[13] Kwon Y C, Balazinska M, Howe B, et al. Skew-resistant parallel processing of feature-extracting scientific user-defined functions [C]//Proceedings of the 1st ACM Symposium on Cloud Computing. ACM, 2010: 75-86

[14] Morton K, Balazinska M, Grossman D. ParaTimer: a progress indicator for MapReduce DAGs[C]//Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. ACM, 2010: 507-518

[15] Shi Y, Meng X, Liu B. Halt or continue: estimating progress of queries in the cloud[M]//Database Systems for Advanced Applications. Springer Berlin Heidelberg, 2012: 169-184

[16] Hassan M, Bamha M, Loulergue F. Handling Data-skew Effects in Join Operations Using MapReduce[J]. Procedia Computer Science, 2014, 29: 145-158

[17] Zacheilas N, Kalogeraki V. Real-Time Scheduling of Skewed MapReduce Jobs in Heterogeneous Environments[C]//International Conference on Autonomic Computing. 2014: 145-158

[18] Seo S, Jang I, Woo K, et al. HPMR: Prefetching and pre-shuffling in shared MapReduce computation environment[C]//IEEE International Conference on Cluster Computing and Workshops, 2009(CLUSTER'09). IEEE, 2009: 2736-2743