·先进计算与数据处理· 文章编号:1000-3428(2017)03-0001-06 文献标志码:A 中图分类号:TP316.4

基于内存与文件共享机制的 Spark I/O 性能优化

黄廷辉,王玉良,汪 振,崔更申

(桂林电子科技大学 计算机与信息安全学院,广西 桂林 541004)

摘 要:通过对 Spark 采用的弹性分布式数据集及任务调度等关键技术进行分析,发现数据处理 I/O 时间是影响 Spark 计算性能的主要瓶颈。为此,研究 Spark 合并文件运行模式,该模式能够减少缓存文件数量,提高 Spark 的 I/O效率,但存在内存开销较高的缺点。在此基础上,给出改进的 Spark Shuffle 过程,即通过设计一种使每个 Mapper 只生成一个缓存文件的运行模式,并且每个 Mapper 共享同一个内存缓冲区,从而提高 I/O 效率和减少内存开销。仿真结果表明,与 Spark 默认模式相比,该运行模式宽依赖计算过程的 I/O 时间缩短 42.9%,可有效提高内存利用率和 Spark 平台运算效率。

关键词:分布式计算;Spark 平台;Shuffle 过程;磁盘 I/O;任务调度

中文引用格式:黄廷辉,王玉良,汪 振,等. 基于内存与文件共享机制的 Spark I/O 性能优化[J]. 计算机工程,2017,43(3):1-6.

英文引用格式:Huang Tinghui, Wang Yuliang, Wang Zhen, et al. Spark I/O Performance Optimization Based on Memory and File Sharing Mechanism[J]. Computer Engineering,2017,43(3):1-6.

Spark I/O Performance Optimization Based on Memory and File Sharing Mechanism

HUANG Tinghui, WANG Yuliang, WANG Zhen, CUI Gengshen

(School of Computer Science and Information Security,
Guilin University of Electronic Technology, Guilin, Guangxi 541004, China)

【Abstract】 Based on the analysis of the key technologies of Spark, such as flexible distributed data set and Spark task scheduling, it is concluded that time of I/O in data processing has a great effect on the computing performance of Spark. Aiming at this problem, this paper studies the run mode of Spark consolidating files that can reduce the number of cache files and improve the I/O efficiency of Spark to some extent, but it still has the disadvantage of high memory cost. Further more, the paper proposes an improved process of Spark Shuffle which designs a mode that every Mapper only generates one cache file, and every Mapper's bucket shares the same memory buffer, thus these improve I/O efficiency and reduce the memory overhead. Simulation results show that, compared with the default mode of Spark, the I/O time of a wide dependent process is shortened by 42.9%, which improves the memory utilization and the efficiency of the Spark platform.

【Key words】 distributed computing; Spark platform; Shuffle process; disk I/O; task scheduling

DOI:10.3969/j.issn.1000-3428.2017.03.001

0 概述

在目前的大数据时代,机器学习和数据挖掘所需面对的数据量越来越大,甚至达到了 PB 级。传统单机平台的性能无法满足现今大数据处理的要求,闭源的分布式平台价格高昂,难以广泛应用,而以 Hadoop 为代表的开源分布式平台成为了主流选择。Hadoop 平台是 2005 年加入 Apache 软件基金会旗下的一个分布式平台,它集成了 Map/Reduce[1] 分布式计算模式,类似于 Google File System[2] 的分布式存储结构。但在 Hadoop 平台设计中,其计算过程大量依赖磁盘,因为磁盘 I/O 延迟和任务调度方法使其计算性能不高。如今,对 Hadoop 平台的研究主要集中在 Hadoop 的优化与应用方向[3-4],包括 HDFS 存

基金项目:国家自然科学基金(61363029);赛尔网络下一代互联网技术创新计划项目(NGII20160306)。

作者简介:黄廷辉(1970—),男,副教授、硕士,主研方向为分布式计算、物联网;王玉良、汪 振、崔更申,硕士。

收稿日期:2016-01-19 修回日期:2016-04-19 E-mail:1530841698@qq.com

储平台的性能研究、MapReduce 计算框架的参数调整及调度算法优化。

除 Hadoop 以外，国内外很多研究机构和公司实现了多种分布式并行计算平台，主要包括 Google Dremel，Strom，Apache Drill，Spark 等。其中，Google 在 MapReduce 的基础上进一步提出了 Dremel 工具[5]，通过新的数据模型和任务分割机制，使得在 PB 级的数据上进行交互式查询的时间降低到秒级，但无法运行数据处理或数据计算任务。Apache Drill 是一个低延迟的分布式海量数据交互式查询引擎，是基于 Google Dremel 而实现的，使用 ANSI SQL 兼容语法，支持 HDFS，HBase，MongoDB 等后端存储，其可以方便地使用 SQL 语言对 PB 级的数据进行快速的交叉查询，但不能进行复杂的数据分析，无法支持包括数据挖掘、图计算等应用。Strom 是一个分布式实时计算系统，它可以简单、可靠地处理大量的数据流。其通过全内存计算技术，使得处理流数据任务时延迟较低，支持复杂的流处理应用，并且可以动态扩展集群数量，实现对硬件设施的高效利用，但 Strom 不能对历史数据进行处理，仅能针对流数据处理应用。

美国 Berkeley 大学的 AMP Lab 在 2009 提出了一种基于分布式内存抽象的通用计算平台——Spark[6]。Spark 采用 master-slave 结构，基于内存计算方式，有效提高了在大数据环境下的计算效率，同时保证了高容错性和高可伸缩性。并且，Spark 提供了丰富的 API 组件，可以高效地开发多种大数据应用系统。

Spark 通过弹性分布式数据集、划分 Stage 和 DAG 图来进行任务调度，明显提高了分布式计算的效率和容错性，并且实现了高效的 SQL 查询、机器学习、图计算、流数据处理等功能。文献[6]提出的 RDD 模型，已经成为 Spark 的基础组件。文献[7]实现了 Spark GraphX，通过 Spark 平台对图计算和图挖掘提供了丰富、易用的接口。文献[8]依托 Spark 实现了对 SQL 语言的支持开发出了 Spark SQL；通过 DataFrame 与 RDD 的相互转换，实现了对 JSON，Parquet，JDBC 等数据源的支持；并且 DataFrame 带有 schema 元信息，使得 Spark SQL 得以洞察数据集的结构信息，从而可以对数据源以及 DataFrame 变换进行针对性的优化，最终达到大幅提升运行效率的目标。

目前国内关于 Spark 的研究主要集中在应用方面[9-10]，对于 Spark 性能优化的技术研究不多[11]。为此，本文结合现有大数据快速实时处理的需要，对 Spark 内核执行引擎中弹性分布式数据集和任务调度两方面进行研究分析，提出一种新的共享内存缓冲机制与文件结构，通过减少中间缓存文件的数量，最终减少 Shuffle 过程的 I/O 时间。

1 Spark 内核关键技术

Spark 内核执行引擎主要包含弹性分布式数据集（Resilient Distributed Datasets，RDD）与任务调度模块两部分[12]。其中 Spark 将所有的数据集转换为统一的数据结构——RDD，所有对数据集的操作都通过 RDD 对外提供的接口来实现。Spark 通过划分 RDD 之间的依赖关系，从而生成 DAG 图。通过 DAG 图进行任务调度，不仅使任务可以以管道的方式进行并行化处理，而且使中间结果保存在内存中，减少了 I/O 时间。Spark 系统层次结构如图 1 所示。

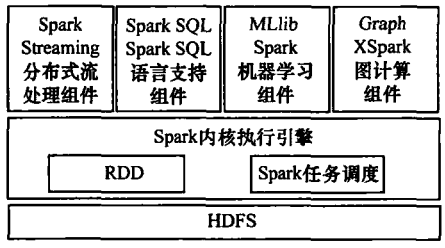

图 1 Spark 系统层次结构

1.1 RDD 技术

RDD 是一个容错、分区、并行的数据结构，可以让用户显式地将数据存储到磁盘和内存中，并提供了丰富的 API 来控制数据。RDD 具有以下特点：

1）RDD 作为数据结构，本质上是一个只读的分区记录集合。

2）RDD 只能基于在稳定物理存储中的数据集和其他已有的 RDD 上执行确定性操作来创建。

3）所有 RDD 提供了大量 API 函数，如 map 函数、filter 函数、groupByKey 函数、count 函数等。这些函数都被分为 2 种类型：action 操作和 transformation 操作。

1.2 Spark 任务调度

Spark 任务调度的核心技术主要有 2 点：1）将 RDD 之间的关系分为窄依赖（narrow dependency）与宽依赖（wide dependency）；2）依靠 DAG 图进行任务调度[13]。

1.2.1 Spark 窄依赖过程

窄依赖是指父 RDD 的每个分区都只被子 RDD

的一个分区所使用,既一对一或多对一关系,如图 2 所示。

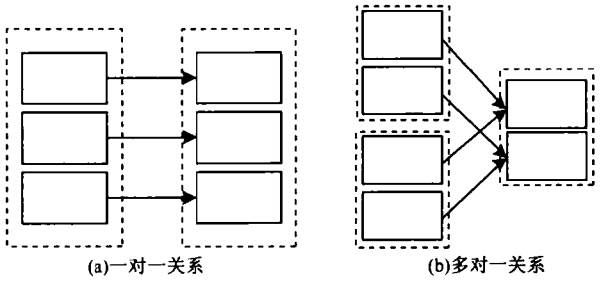

图 2　窄依赖过程

窄依赖的 RDD 关系在进行分布式计算时拥有众多优势。首先,当一个 RDD 分区有多个窄依赖操作相邻时,这些窄依赖操作可以以流水线的方式计算。这样计算完成的结果可以继续保存在内存中,直接提供给下一个操作进行调用,从而节省了 I/O 时间。其次,窄依赖能够更有效地进行失效节点的恢复,即只需重新计算丢失 RDD 分区的父分区,而不需要依赖其他 RDD 分区。最后,不同节点之间的 RDD 区块可以并行地计算窄依赖操作。

1.2.2　Spark 宽依赖过程

宽依赖是指父 RDD 的每个分区都被所有的子 RDD 分区所使用,既多对多关系,如图 3 所示。

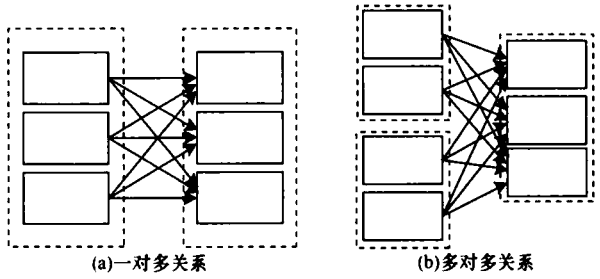

图 3　宽依赖过程

Spark 的宽依赖过程与 MapReduce 相似,被分为 Shuffle write 和 Shuffle fetch 2 个部分。与窄依赖不同,宽依赖的单个节点失效可能导致这个 RDD 的所有祖先丢失部分分区,因而需要整体重新计算。首先,Shuffle write 过程使得每一个 Mapper 会根据 Reducer 的数量创建出相应的 bucket,bucket 的数量为 $M \times R$,其中,M 是运算 map 任务 Mapper 的个数;R 是运算 Reduce 任务 Reducer 的个数。其次,Mapper 产生的结果会根据设置的 partition 算法填充到每个 bucket 中。这里的 partition 算法是可以自定义的,默认的算法是通过 Hash 函数分配到不同的 bucket 中去。当 Reducer 启动时,它会根据自己 task ID 和所依赖的 Mapper ID,从远端或是本地的 block manager 中取得相应的 bucket 作为 Reducer 的输入进行处理,其过程如图 4 所示。

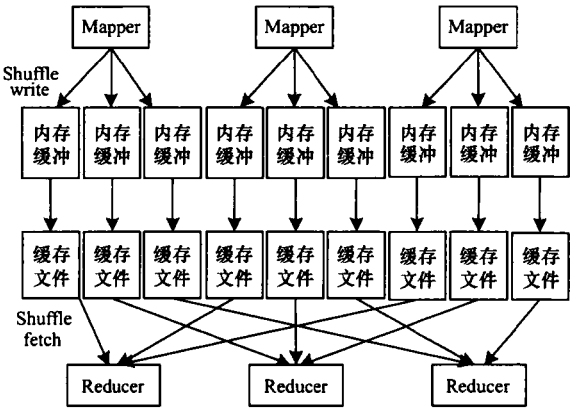

图 4　Spark Shuffle 过程

1.2.3　Spark 任务调度器

Spark 任务调度器的主要任务是根据 RDD 的 Lineage 图创建的无回路有向图(DAG)进行任务调度,使用 DAG 图调度的,可以减少磁盘 I/O 以及提高任务的并行化程度。Spark 任务调度的主要步骤如下:

1)根据 Lineage 图划分 stage 并生成 DAG 图。Stage 划分方法为每个 stage 内部尽可能多地包含一组具有窄依赖关系的 transformation 操作,并将它们流水线化。其中 stage 的边界有 2 种情况:(1)宽依赖上的 Shuffle 操作;(2)缓存分区,它可以减少宽依赖过程的时间。如图 5 所示。

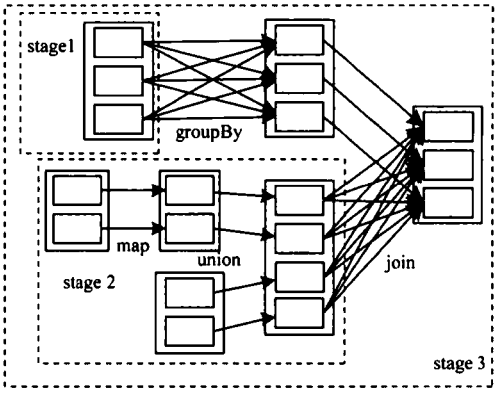

图 5　DAG 图

2)Spark 将给 stage 中的每一个 RDD 分区创建一个对应的 task,该 task 将对应的 RDD 分区从 stage 开始计算到 stage 结尾,其中间结果将保存到内存中,计算完成后才会写入磁盘或继续缓存到内存中。每一个节点根据其硬件配置可同时运行多个 task 任务,在窄依赖中每一个 task 都是独立运行的,这样就实现了高并发处理。

3)当一个 stage 内所有的 task 执行完成后,

Spark 将执行 stage 间的宽依赖操作。

根据以上的步骤可知,DAG 图进行调度是因为窄依赖可以支持在同一个节点上以管道形式执行多条命令。而宽依赖需要所有的父分区都是可用的,可能还需要进行跨节点传递。传统的 MapReduce 不区分宽依赖与窄依赖,每一个操作都与宽依赖相似,中间不仅需要通过磁盘来存储缓存文件,而且只有当上一个 MapReduce 操作运行完才能运行下一个 MapReduce 操作。Spark 通过区分窄依赖与宽依赖操作和生成 DAG 图的方式来进行任务调度,其具有以下优点:stage 内的窄依赖操作不需要多次读写磁盘,降低了 I/O 开销;窄依赖可以并发运行,每个 task 不需要等待别的 task 的计算结果。以上两点使得 Spark 相对于 MapReduce 明显减少了计算时间,提高了计算效率。

2 Spark 宽依赖技术优化

上文描述了 Spark 原有的宽依赖计算过程,但这些 shuffle 过程会造成 I/O 时间增加,由于每一个 Mapper 都会产生 Reducer 个 shuffle 文件,如图 5 所示。当 Mapper 与 Reducer 数量较多时,产生的缓存文件会对文件系统造成非常大的负担,并且体积较小、数量较多的缓存文件会显著地增加磁盘 I/O 的时间,导致 Spark 性能下降。为此,Spark 可以通过设置 spark.shuffle.consolidateFiles(以下简称为 consolidateFiles)参数为 true 来缓解这个问题,该参数是将每个 bucket 对应文件中的一个段(segment)。此时每个节点只需要创建 $N \times R$ 个文件,N 代表一个节点同时可以运行的 Mapper 个数,R 是 reduce 的分区个数,当一个 task 结束后,下一个 task 继续向该文件的下一个 segment 写入。当硬件平台不变,需要执行的 Mapper 与 Reducer 数量越多时,节点中在同一时间可以执行的任务占总任务的比例越小。所以,当 Mapper 与 Reducer 数量较多时,consolidateFiles 参数可以有效减少文件的数量,从而减少文件系统的压力,提高 I/O 性能。

虽然采用 consolidateFiles 参数可以有效减少 shuffle 操作缓存文件的数量,但缓存文件的数量依然是 Mapper 与 Reducer 的乘积关系。当节点与数据量较大时依然会遇到缓存文件数量快速增长的问题。除此以外,上面提到每个 bucket 拥有独立的 100 KB缓存,这样当 Mapper 与 Reducer 数量较多时,也会出现内存占用较高的问题。而且 Map 操作生成的中间结果不是均匀分布的,此时就会出现有的缓存已经存满但有的缓存还有剩余,这样就会造成一定的内存浪费。

针对上述问题,本文通过新的文件结构和内存共享机制来解决。首先设计一种使每个 Mapper 只生成一个缓存文件的模式,这样可以显著减少缓存文件的数量,提高磁盘 I/O 的性能。其次通过将每个 Mapper 的 bucket 共享同一个内存缓冲区,可以提高内存的利用率,更加灵活地设置内存缓冲区的大小。其具体运行过程如下:

1)新的文件结构包含 3 个部分:文件头,bucket 区,溢出存储区。文件头包含每一个 bucket 块对应的地址、大小以及 bucket 区的剩余空间;bucket 区包含多个 bucket 块,这些块的数量等于 Mapper 生成 bucket 的数量,并且可以设定 bucket 块的大小,块的大小设定与 RDD 分区的大小和该函数操作的类型有关;溢出存储区是当 bucket 区的块存满之后,将向溢出存储区申请新的空块来存储。bucket 区的块之间是顺序存储关系,而 bucket 区的块与溢出存储区的块之间构成了类似于链表的结构,当在 shuffle fetch 过程中要将 bucket 发送到对应的 Reducer 时,按链式关系读取即可,如图 6 所示。

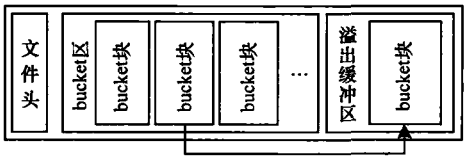

图 6 改进的缓存文件结构

2)共享内存缓冲区的结构为一块可调整大小的内存区域(在初始化时设定其大小),每一个 bucket 对应一个链表,用来取代每个 bucket 文件对应的内存缓冲,这样就能有效提高内存的利用效率。

3)采用新的文件结构和内存共享机制的运行流程如下:Mapper 将生成的中间结果写入共享内存缓冲中,当共享内存缓冲的占用率达到阈值(如 80%)时,Manager 会将共享内存缓冲中的内容写入磁盘文件。当 map 操作完成后,进入 shuffle fetch 过程,Manager 并行地读入缓存文件,以链表的方式读取各个 bucket,将其发送到对应的 Reducer 中。Reducer 将接收到的数据放入一个 hashmap 中,如果是 reduceByKey 等操作,则需要更新 hashmap 中 value 的值;如果是 groupByKey 等操作,则需要将所有的 value 全部存放在 hashmap 中,然后将每个 key 所对

应的 value 合并成一个数组。最后将 hashmap 提交给 reduce 进行处理，其过程如图 7 所示。

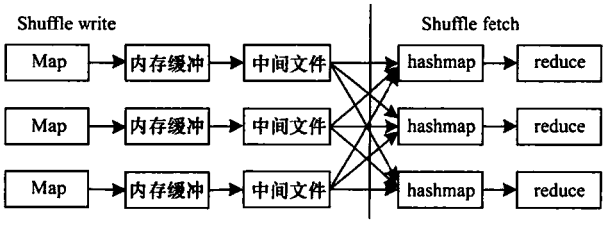

图 7　改进的 Shuffle 过程

3　仿真与性能分析

3.1　I/O 性能模拟

3.1.1　仿真环境设置

为估算 Spark 平台在多种情况下的 I/O 性能，本文通过 Java 实现了一个 Spark I/O 模拟系统[14]。通过线程来模拟节点和 task，每个父线程模拟一台分布式节点，共计 10 个父线程（节点）；每个子线程模拟 Spark 的 task，每个父线程拥有 10 个子线程（task），每个子线程通过式（1）、式（2）来计算磁盘 I/O时间和网络 I/O 时间。最后统计各个子线程运行的模拟结果作为评估指标。

基本的仿真参数设置为：磁盘延迟为 7 ms（磁盘寻道延迟 + 磁盘旋转延迟），磁盘带宽为 100 MB/s；网络延迟为 5 ms，网络带宽为 1 000 Mb/s[15]；RDD 分区大小为 128 MB。

在基本参数的基础上，设置每台 slave 节点最大同时运行任务数为 10，整个计算网络中有 10 台 slave 节点。对于分布式系统，其 I/O 开销主要来自磁盘 I/O 与网络 I/O[16]。其中，磁盘 I/O 时间的计算为：

$$T_1 = D + P_1/B_1 \tag{1}$$

其中，T_1 为磁盘 I/O 时间；D 为磁盘延迟；P_1 为读（写）文件大小；B_1 为磁盘带宽。

网络 I/O 时间的计算为：

$$T_2 = L \times 3 + P_2/B_2 \tag{2}$$

其中，T_2 为网络 I/O 时间；L 为网络延迟；P_2 为传输文件大小；B_2 为网络带宽。因为 Spark 是通过 SSH 协议来通信的，建立连接需要经过“三次握手”，所以需要 3 倍的网络延迟。

在此基础上，仿真环境分别为 Spark 默认模式（default）、文件合并模式（consolidateFiles 为 true）以及本文提出的改进模式，共计 3 种运行模式。

3.1.2　仿真结果分析

为了验证 Spark3 种运行模式 I/O 效率的差异，分别选取了数据挖掘、图计算和机器学习 3 个热门领域的经典算法 Kmeans[17]，PageRank[18] 和 Logistic[19] 来检测 3 种运行模式的 I/O 效率。所有实验结果均为 10 次仿真结果的平均值。实验结果的横坐标轴为 Spark 运行模式，纵坐标轴代表所有任务 I/O 时间的累加和，并不是实际运行时间，实际系统中的 task 为并行运行，根据硬件平台不同其实际 I/O 时间会有数十倍到上百倍的提高，仿真结果如图 8 所示。在 Kmeans 算法中，ConsoliDateFiles 为 true 的运行模式，比默认 Spark 运行模式的 I/O 效率提高了26.3%，改进的 Spark 运行模式比默认 Spark 运行模式的 I/O 效率提高了 57.6%。在 Logistic 算法中，ConsoliDateFiles 为 true 的运行模式，比默认 Spark 运行模式的 I/O 效率提高了 19.6%，改进的 Spark 运行模式比默认 Spark 运行模式的 I/O 效率提高了 42.9%。在 PageRank 算法中，ConsoliDateFiles 为 true 的运行模式，比默认 Spark 运行模式的 I/O 效率提高了25.3%，改进的 Spark 运行模式比默认 Spark 运行模式的 I/O 效率提高了 55.4%。可以发现采用改进的 Spark 运行模式明显提高了 Spark 宽依赖过程的 I/O 效率，而 ConsoliDateFiles 参数也可以有效地提高宽依赖过程的 I/O 效率。

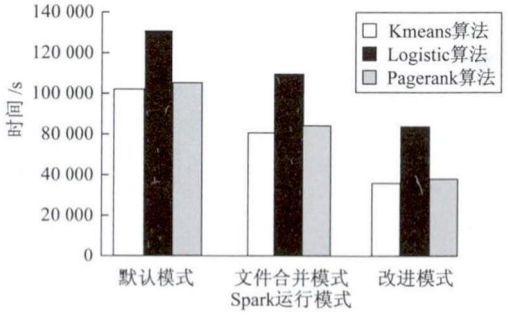

图 8　3 个算法仿真结果

3.2　实际运行结果

为了验证 Spark 3 种运行模式实际运行的差异，本文分别选取了数据挖掘、图计算领域的经典算法：Kmeans，PageRank 来检测 3 种运行模式的运行结果。其中，Kmeans 实验的数据集为美国加州大学欧文分校 UCI 数据库提供的数据集（Heterogeneity Activity Recognition Data Set，http://archive.ics.uci.edu/ml/datasets/Heterogeneity + Activity + Recognition），大小为 741 MB；PageRank 实验采用 Twitter 用户数据集（Twitter Memes Dataset，https://github.com/mongodb-labs/big-data-exploration/wiki/

PageRank-on-Twitter-Memes-Dataset),大小为11 GB。硬件环境为 3 台 PC 电脑,CPU 为酷睿 e8500,内存 4 GB,操作系统采用 Centos6.5,Spark 版本为1.4.1。实验结果的横坐标轴为 Spark 运行模式;因为 2 个实验数据集与算法不同,运行时间差异较大,所以纵坐标为结果相对于 Spark 默认模式的百分比,其结果如图 9 所示。

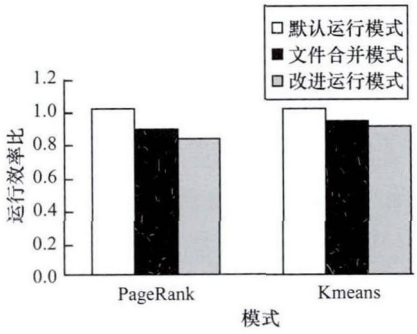

图 9 实际运行结果

在 Kmeans 算法中,文件合并模式比默认模式性能提高了 7%,而改进模式比默认模式性能提高了 11%。在 PageRank 算法中,文件合并模式比默认模式性能提高了 12%,而改进模式比默认模式性能提高了 18%。可以发现采用改进的 Spark 运行模式有效提高了 Spark 的运行效率。但无论是改进的 Spark 运行模式还是文件合并模式都只有在数据量较大与 Map 及 Reduce 任务数量较多时才能有效提高宽依赖的效率,而在 Map 与 Reduce 任务数量较少时,采用改进的 Spark 运行模式和 ConsoliDateFiles 参数有可能会造成较大的系统开销。

4 结束语

本文通过分析 Spark 运行模式,认为 I/O 效率低是 Spark 性能的主要瓶颈之一,进而提出了改进的 Spark 运行模式,提高了内存的利用率和 Spark 平台的运算效率。通过仿真系统模拟了 3 种运行模式 I/O的时间的差别,得出该改进运行模式可以有效提高 Spark 的 I/O 效率。下一步将结合 Linux 系统,实现本文提出的文件结构和共享内存缓冲。

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] Ghemawat S, Gobioff H, Leung S T. The Google File System[C]//Proceedings of ACM SIGOPS Ooperating Systems Review. New York, USA: ACM Press, 2003: 29-43.

[3] 陶永才,石 磊. 异构资源环境下的 MapReduce 性能优化[J]. 小型微型计算机系统,2013,34(2):287-292.

[4] 顾 荣,严金双,杨晓亮,等. Hadoop MapReduce 短作业执行性能优化[J]. 计算机研究与发展,2014,51(6):1270-1280.

[5] Melnik S, Gubarev A, Long J J, et al. Dremel: Interactive Analysis of Web-scale Datasets[J]. Communications of the ACM,2010,3(12):114-123.

[6] Zaharia M, Chowdhury M, Das T, et al. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing[C]//Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation. San Jose, USA: USENIX Association, 2012:1-12.

[7] Xin R S, Crankshaw D, Dave A, et al. GraphX: Unifying Data - parallel and Graph - parallel Analytics[EB/OL]. (2014-10-20). http://www. researchgate. net/publication/260147249.

[8] Armbrust M, Xin R S, Lian C, et al. Spark SQL: Relational Data Processing in Spark[C]//Proceedings of 2015 ACM SIGMOD International Conference on Management of Data. Melbourne, Australia: ACM Press,2015:1383-1394.

[9] 吴雯祺. Spark 性能数据收集分析系统的设计与实现[D]. 哈尔滨:哈尔滨工业大学,2015.

[10] 王诏远,王宏杰,邢焕来,等. 基于 Spark 的蚁群优化算法[J]. 计算机应用,2015,35(10):2777-2780.

[11] 冯 琳. 集群计算引擎 Spark 中的内存优化研究与实现[D]. 北京:清华大学,2013.

[12] 梁 彦. 基于分布式平台 Spark 和 YARN 的数据挖掘算法的并行化研究[D]. 广州:中山大学,2014.

[13] 杨志伟,郑 烇,王 嵩,等. 异构 Spark 集群下自适应任务调度策略[J]. 计算机工程,2016,42(1):31-35,40.

[14] 薛志云,何 军,张丹阳,等. Hadoop 和 Spark 在实验室中部署与性能评估[J]. 实验室研究与探索,2015,34(11):77-81.

[15] 高燕飞,陈俊杰,强 彦. Hadoop 平台下的动态调度算法[J]. 计算机科学,2015,42(9):45-49,69.

[16] 龙赛琴. 云存储系统中的数据布局策略研究[D]. 广州:华南理工大学,2014.

[17] Wagstaff K, Cardie C, Rogers S, et al. Constrained k-means Clustering with Background Knowledge[C]//Proceedings of International Conference on Machine Learning. Williamstown, USA: International Machine Learning Society,2001:577-584.

[18] Page L, Brin S, Motwani R, et al. The Page Rank Citation Ranking: Bringing Order to the Web[EB/OL]. (1999-10-21). http://ilpubs. stanford. edu:8090/422/.

[19] Hosmer J D W, Lemeshow S. Applied Logistic Regression[M]. [S.1.]: John Wiley & Sons,2004.

编辑 索书志