计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.191000103

面向 MapReduce 的中间数据传输流水线优化机制

张元鸣　虞家睿　蒋建波　陆佳炜　肖　刚
浙江工业大学计算机科学与技术学院　杭州 310023
(zym@zjut.edu.cn)

摘　要　MapReduce 是一种适用于大数据处理的重要并行计算框架,通过在大量集群节点上并行执行多个任务,极大地提高了数据的处理性能。然而,由于中间数据需要等到 Mapper 任务完成之后才能被发送给 Reducer 任务,由此导致的大量传输延迟成为 MapReduce 框架性能的重要瓶颈。为此,文中提出了一种面向 MapReduce 的中间数据传输流水线优化机制,将有效计算与中间数据传输解耦,以流水线的方式重叠执行各个阶段,有效隐藏数据传输开销。文中还给出了中间数据传输流水线执行机制和实现策略,包括流水线划分、数据细分、数据归并和数据传输粒度等。在公开数据集上对所提中间数据传输流水线优化机制进行了评价,当 Shuffle 数据量较大时,该优化机制比默认框架的整体性能提高了 60.2%。

关键词:MapReduce 框架;中间数据传输;传输延迟;流水线;溢写文件归并
中图法分类号　TP391

Intermediate Data Transmission Pipeline Optimization Mechanism for MapReduce Framework

ZHANG Yuan-ming, YU Jia-rui, JIANG Jian-bo, LU Jia-wei and XIAO Gang
College of Computer Science and Technology, Zhejiang University of Technology, Hangzhou 310023, China

Abstract　MapReduce is an important parallel computing framework for large data processing, which greatly improves the performance of data processing by performing multiple tasks in parallel on a large number of cluster nodes. However, since the intermediate data needs to wait until the Mapper task is completed, it can be sent to the Reducer task. The massive transmission delay becomes an important bottleneck of the MapReduce framework performance. To this end, an intermediate data transmission pipeline mechanism for MapReduce is proposed. It decouples the effective computation from intermediate data transmission, overlaps each stage in pipeline mode, and effectively hides data transmission delay. The execution mechanism and implementation strategy of the approach are given, including pipeline partition, data subdivision, data merging and data transmission granularity. The proposed mechanism is evaluated on public data sets. When the Shuffle data volume is large, the overall performance improves by 60.2% compared with the default framework.

Keywords　MapReduce framework, Intermediate data, Transmission delay, Pipeline, Overflow file merging

1　引言

随着信息技术的高速发展,网络中的数据呈现爆炸性增长,为了应对海量数据的处理需求,需要大量的计算机节点协同完成。MapReduce 是 Google 公司在 2004 年提出的一个适用于海量数据处理的并行计算框架[1],通过在大量廉价的机群节点同时运行多个任务并行地处理数据,提高了数据的处理能力,被广泛应用于数据挖掘、风险分析、客户行为分析等领域[2]。Hadoop 和 Spark 都是当前基于 MapReduce 框架的主流开源实现[3]。

MapReduce 框架的数据处理过程主要包括 Map 阶段和 Reduce 阶段。Map 阶段读取划分数据块,划分后的数据块按照一定的规则解析成键值对;Reduce 阶段接收来自 Map 阶段的数据集,汇总得到最后结果。此外,Shuffle 阶段是连接 Map 和 Reduce 之间的桥梁,该阶段将 Map 阶段处理之后的数据集经过分区之后传输给 Reduce 阶段。中间数据的操作涉及两个关键的操作:一个是 Write 操作,将键值对逐个写入内存或磁盘文件;另一个是 Fetch 操作,将内存或磁盘文件传输到 Reduce 阶段。由于 Shuffle 阶段涉及大量的数据读写操作和数据传输操作,因此中间数据处理所产生的数据延迟开销成为影响 MapReduce 框架性能的重要瓶颈[4]。实际测算表明,Shuffle 阶段会产生大量的数据,以 Facebook 的数据中心为例,中间数据的流量占数据中心总流量的 46%[5]。

针对 MapReduce 框架的中间数据处理优化问题,国内外研究者已从流程优化设计[6-13]、读写优化与参数优化[14-19]等方面展开了研究,通过特殊的机制或参数来减少和优化数据

到稿日期:2019-10-16　返修日期:2019-12-06　本文已加入开放科学计划(OSID),请扫描上方二维码获取补充信息。
基金项目:计算机体系结构国家重点实验室开放课题(CARCH201804)
This work was supported by the Open Projects Funding of State Key Lab of Computer Architecture(CARCH201804), ICT, CAS.
通信作者:肖刚(xg@zjut.edu.cn)

延迟，对提高整体性能具有一定的作用，但是这些方法在适用性和有效性上存在一定的局限性。为此，本文提出一种面向MapReduce的中间数据传输流水线优化机制，将Map阶段、Reduce阶段的有效计算和中间数据处理以流水线的方式重叠执行，隐藏了中间数据的延迟开销，从而提高MapReduce框架的整体性能。本文的主要贡献包括：

(1)提出了面向MapReduce的有效计算和数据传输重叠执行的流水线执行机制，基于流水线机制将数据传输延迟隐藏于有效计算中；

(2)设计了面向MapReduce的中间数据传输流水线优化实现策略，给出了数据块划分方法、数据归并方式和数据发送时机；

(3)在公开数据集上评价了面向MapReduce的数据传输流水线优化机制，证明该机制能够隐藏数据延迟开销，提高MapReduce的整体性能。

2 相关工作

以MapReduce框架为代表的数据并行系统成为了当前大数据技术体系中最重要的基础性框架，提高其整体性能具有重要的实际应用价值。由于Shuffle阶段的中间数据读写与传输延迟开销已经成为MapReduce框架的主要性能瓶颈[20-22]，国内外学者围绕该问题进行了深入的研究。

在数据传输优化方面，Ahmad等[5]提出了一种传输重叠的MapReduce模型——MaRCO，以减少传输延迟；Yu等[6]提出了一种Hadoop-A框架，采用一种新的网络悬浮合并算法，实现了无重复、无磁盘访问的数据合并，并将Shuffle，Merge和Reduce阶段重叠执行，提高了中间数据吞吐量；Chowdhury等[7]提出了全局优化算法以缩短常见通信模式的传输时间，且允许在传输时改变调度策略；Li等[8]提出一种局部增强负载均衡算法，将MapReduce执行流程扩展到Map，Local Reduce，Shuffle以及Final Reduce，使得Reduce计算和Shuffle可以共享，充分利用CPU和I/O资源。以上4种方法主要通过提高传输重叠度来加快中间数据传输。

Rahman等[9]提出了一种面向高性能集群的MapReduce框架，通过本地磁盘和Lustre部署的中间数据存储架构，在运行时在线分析选择最佳中间数据存储位置。Arslan等[10]提出了一种LoNARS算法，通过考虑数据局部性和网络流量来减少Reducer任务的调度；并通过数据位置感知将Reducer任务安排在更靠近Mapper任务的位置以减少数据访问延迟；还通过网络流量感知最小化热点来减少网络拥塞。Cao等[11]提出了一种优化Map密集型作业的中间数据通信优化方法，提取MapReduce计算作业的运行前调度信息的特征并量化数据通信活跃度，采用朴素贝叶斯分类模型实现分类预测，然后根据预测结果把通信活跃的作业集中映射到同一机架中。Liu等[12]设计了一个主动聚合框架以优化Shuffle阶段网络流量，该框架将Map任务的输出数据主动聚合到数据中心的子集。Seo等[13]提出一种预取与预先混洗的方案，预取方案利用数据局部性，预先混洗方案可以减少Shuffle键值对的网络开销，能够在共享的MapReduce计算环境中有效地提高整体性能。这些方法主要通过减少Shuffle阶段数据的传输量来加快中间数据的传输。

在读写优化方面，Zhang等[14]通过分布式缓存提高MapReduce性能，使Reducer任务尽可能早地获取Mapper任务发送到分布式内存缓存中的数据；Wang等[15]提出了一种基于固态硬盘的数据缓存技术mpCache，用于缓存输入数据和本地数据，从而提高读写性能，该方法需要配置额外的存储资源以获取较好的读写性能；Li等[16]提出了一种独特的轻量级Shuffle服务组件，包括提取Reducer任务中的Shuffle任务，将Shuffle任务重建为服务以及改进Map端的I/O的调度策略。这3种方法主要通过特殊的硬件加快中间数据传输。

Qi等[17]提出了一种基于概率的B树构造算法和多路查询算法，利用读写开销估算和缓冲区信息改造文件读写策略和内外存替换算法，优化中间数据的高并发读写性能；Wang等[18]构建了一个数学模型以判断Map阶段不同操作顺序的计算复杂度，给出了一种在溢写阶段可重新配置排序和分组的策略，将Map阶段的溢写视为可调节架构并提供灵活的操作顺序；Tan等[19]通过动态交错执行Reducer任务和Mapper任务，使多个任务能够在同一个JVM正确执行，提高了数据处理的性能。这些方法主要从算法层面优化了溢写文件的读写操作。

现有的数据传输优化方法主要涉及传输重叠、数据本地性和数据读写性能等方面，并没有考虑将中间数据提前传输。本文提出的中间数据传输流水线优化机制能够将Map阶段产生的中间数据尽可能早地开启传输，将Map和Reduce阶段的有效计算与中间数据传输解耦，以流水线的方式重叠执行，隐藏中间数据传输开销，从而提高MapReduce框架的整体性能。

3 MapReduce框架原理

MapReduce框架对数据块的处理过程从执行逻辑上可以划分为Map阶段、Shuffle阶段和Reduce阶段。在Map阶段启动多个Mapper任务并行地处理多个数据块，在Reduce阶段启动多个Reducer任务并行处理从Map阶段传输过来的中间数据，而Shuffle则是中间数据处理的重要阶段。图1给出了MapReduce框架的数据处理过程，其涉及的关键数据处理步骤如下。

(1)数据块读取：从分布式文件系统中读取被分配的数据块内容；

(2)键值对生成：将数据块中的数据解析为键值对〈*key*，*value*〉的形式；

(3)键值对映射：根据业务逻辑将原键值对变换得到新的键值对；

(4)数据分区：根据分区算法(如Hash分区)，将键值对映射到相应的Reducer任务；

(5)数据溢写：当分区数据超过内存限制时，将数据溢写到磁盘文件；

(6)数据传输：Mapper阶段完成后，将分区结果传输到Reducer任务；

(7)数据归约：根据业务逻辑对数据进行汇总。

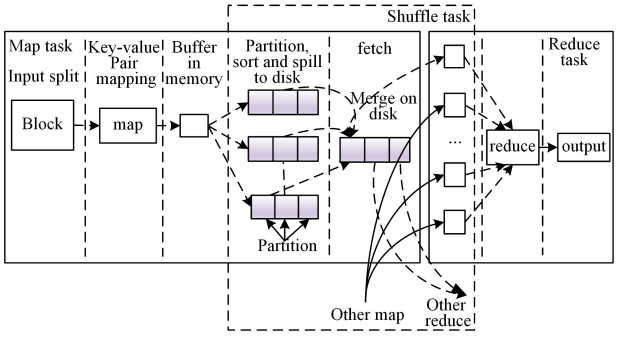

图 1 MapReduce 框架数据处理基本流程

Fig. 1 Data processing flow of MapReduce framework

上述数据处理步骤中的数据分区、数据溢写和数据传输统称为中间数据处理，是 Shuffle 阶段的主要任务。在现有的默认数据处理流程中，Map 阶段、Shuffle 阶段与 Reduce 阶段是串行执行的，主要表现在：Shuffle 阶段的数据传输是在各个 Map 任务完成之后才开始的；相应地，Reduce 阶段是在接收到所有分区数据后才开始启动。这种串行模式尽管简化了框架的实现，但是给数据传输带来了不必要的延迟开销，并且显著降低了 MapReduce 框架的整体性能，而且也造成过多的数据积聚在 Mapper 端，而 Reducer 端却处于空闲状态，不利于充分利用集群硬件资源。

图 2 给出了一个实例以说明 Mapper 任务、数据传输和 Reducer 任务的执行顺序。假设有 2 个节点，每个节点上依次运行 2 个 Mapper 任务和 1 个 Reducer 任务。节点 1 读取数据块，执行 Mapper1 任务，将数据划分为 P1 和 P2 两个分区，并将分区数据溢写到磁盘中；Mapper1 任务结束后，将 P1 分区和 P2 分区传输到各个 Reducer 任务。假设 P1 分区在本地节点处理，P2 分区在节点 2 处理，此时，只需将 P2 分区传输给节点 2，然后节点 1 启动下一个 Mapper 任务，直到所有的 Mapper 任务计算完成。当所有的分区数据都传输到 Reducer 端时，则开启 Reducer 任务，执行数据归约操作。

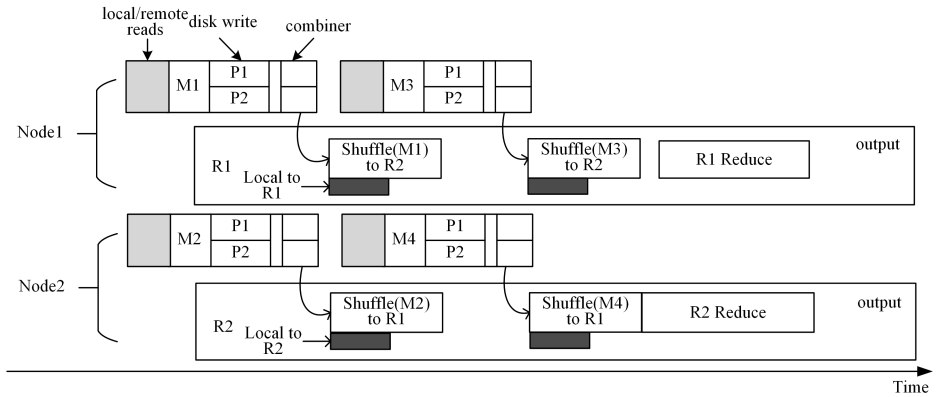

图 2 MapReduce 默认执行过程

Fig. 2 Default execution process of MapReduce framework

4 中间数据传输流水线机制

4.1 中间数据传输流水线划分

本节详细给出了中间数据传输流水线优化机制，图 3 给出了基本原理，整个计算过程划分为 Map 阶段的有效计算、数据传输和 Reduce 阶段的有效计算，这 3 个阶段能够以流水线的方式重叠执行，数据传输的延迟被隐藏在前后两个阶段的有效计算中。

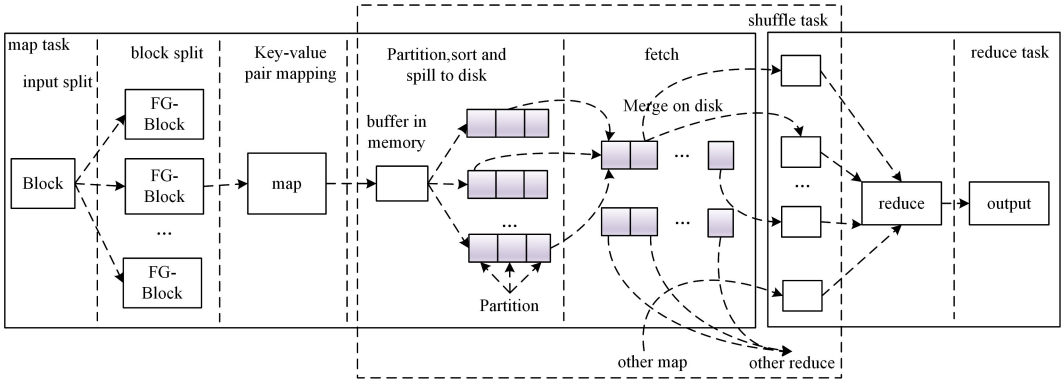

图 3 MapReduce 的中间数据传输流水线机制

Fig. 3 Pipeline mechanism of intermediate data transmission in MapReduce

数据发送时机指中间数据何时发送到 Reducer 端以及发送时满足的条件。默认 MapReduce 框架中间数据是等到 Mapper 端计算完成，最终的溢写文件生成，才开始进行数据传输。本文算法在数据传输流水线优化机制中将 Shuffle 阶段的数据传输提前，及早地将中间数据发送到 Reducer 端，Reducer 端接收到数据就开始进行有效计算。

为了构建中间数据传输流水线机制，需要源源不断地提供处理任务，然而 MapReduce 框架中默认的数据处理单位是粗粒度数据块 Block，其大小一般为 64 MB 或 128 MB。为此，需要将 Block 进行细分，得到细粒度的数据块（Fine-grain

Blocks, FGB), 并将其作为 Mapper 任务处理的基本单位。

以 FGB 作为 MapReduce 框架的处理对象, 依次将 Map 计算、数据分区以及溢写、数据归并、数据传输、数据合并和 Reduce 计算 6 种操作通过流水线方式重叠执行, 其中, Map 计算、数据分区以及溢写文件与默认的 MapReduce 框架是相同的, 不同之处在于 Shuffle 阶段中间数据的传输不再等到 Mapper 任务完成。由于分区时已经确定溢写文件将被分配到哪个 Reducer 任务, 因此可以直接将其传输到 Reducer 任务。为了避免网络中的数据传输量过大, 首先对溢写文件进行部分归并然后再发送; 同时, Reducer 任务提前接收到数据并进行有效计算, 如果有效计算所需要的数据来自不同的节点, 则存在数据依赖, 此时需等待数据全部传输完成之后才能进行有效计算。图 4 为中间数据传输流水线执行时空图。

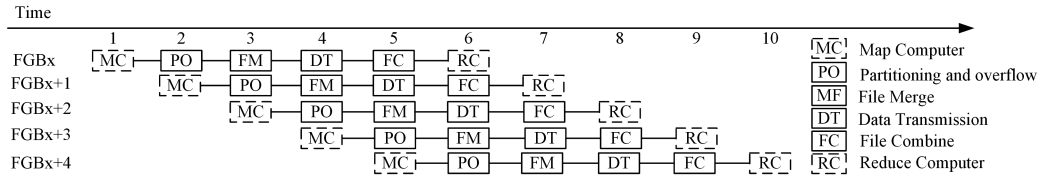

图 4 中间数据传输流水线执行时空图

Fig. 4 Time-space diagram of pipeline for intermediate data transmission

4.2 中间数据传输的归并策略

Map 阶段产生的溢写数据量是非常大的, 若直接进行数据传输会使得传输过于频繁, 并造成网络拥塞, 进而降低中间数据传输性能。为此, 需对溢写数据做并归操作。考虑到溢写数据内部都是有序的, 本文采用多路归并的方法, 假设归并路数为 k, 总溢写文件数为 m, 总记录数为 n。常见的多路归并算法的比较次数表达式为:

$$\lceil \log_2 m / \log_2 k \rceil (k-1)(n-1) \tag{1}$$

从式(1)可以看出, 在常见的多路归并算法中, k 值越大, 比较次数就越多, 归并所需要的时间也就更多。为避免该缺点, 本文采用败者树方式[23], 使用树的叶子节点存放文件信息, 中间节点记录败者信息, 能够快速找到最大值或最小值。k 路归并的树每次选取最大或最小值的比较次数就是树的深度 $\log_2 k$。n 个记录的总比较次数是 $\lceil \log_2 k \rceil (n-1)$, 归并路数是 $\lceil \log_2 m / \log_2 k \rceil$, 败者树的比较次数表达式为:

$$\lceil \log_2 m / \log_2 k \rceil \lceil \log_2 k \rceil (n-1) = \lceil \log_2 m \rceil (n-1) \tag{2}$$

从式(2)可以看出, 败者树的比较次数与归并路数 k 无关。由于溢写文件并不是一次性全部同时产生的, 归并路数 k 也不宜设置过大, 否则一方面等到 k 个溢写文件产生会浪费一些时间, 另一方面 k 过大也许会超出内存的阈值。

4.3 中间数据传输的粒度

为了避免频繁地将 Mapper 任务的小粒度溢写文件传输给 Reducer 任务, 需要将小粒度溢写文件合并为一个大粒度溢写文件, 待溢写文件数据量达到一定的阈值时再传输给 Reducer 任务, 这样将有效地提高数据传输的效率。因此, 中间数据传输的粒度成为提高流水线性能的另外一个重要参数。

设中间数据传输的粒度为 T, 则当溢写文件的数据量超过这个阈值时, 数据传输操作才启动, 将溢写文件从 Mapper 任务传输给 Reducer 任务, 否则继续等待, 当剩下的溢写文件数只剩 1 个时, 可以直接进行数据传输。实验结果表明, 当 T 较小时, 会导致频繁的数据传输; 而当 T 较大时, 则会造成数据传输的延迟。

图 5 为溢写文件归并示意图。将 Mapper 任务产生的溢写文件不断加入到溢写队列, 从队列中取前 k 个溢写文件作为归并段进行归并排序, 得到一个新的溢写文件。若溢写文件的粒度超过所设定的阈值, 则进行数据传输操作, 否则将其送回溢写队列。

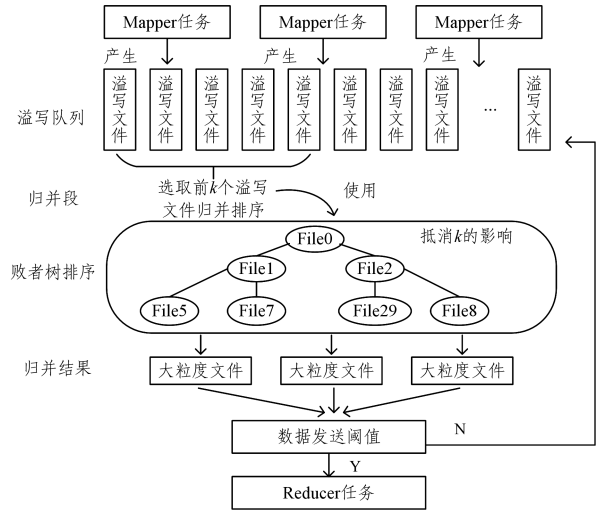

图 5 溢写文件归并流程

Fig. 5 Merging process of overflow file

5 实验与分析

5.1 实验环境

本节对中间数据传输流水线机制进行评价。实验环境是一个包含 16 个计算节点的集群, 每个节点是 4 核处理器, 操作系统为 64 位 Ubuntu16.04, 每个节点的内存为 16GB, 通过 Actor 来模拟 Mapper 任务与 Reducer 任务, Mapper 和 Reducer 各 8 个。实验采用公开的 BTS 和 Konect 数据集, 如表 1 所列。BTS 数据集是真实的航天航空数据, 一共包含约 500 个文件, 平均每个文件包含记录数为 20 万条; Konect 数据集包含各种类型的网络大数据, 实验选取了其中的维基百科链接数据集。

表 1 实验数据集统计信息

Table 1 Statistics of experimental data sets

Data Set	Running Examples	Size/GB	Describe
BTS	WordCount	32	US 航空数据
Konect	PageRank	8	网络数据集
Konect	InvertedIndex	8	网络数据集

实验采用了 WordCount 算法、PageRank 算法和 Inverted Index 算法作为对比算法。

5.2 参数评价

本节通过实验来评价 FGB 大小、归并路数以及数据传输粒度 3 个参数对数据传输流水线机制性能的影响，评价指标是作业整体运行时间。

图 6 给出了在 BST 数据集上执行 WordCount 算法时 FGB 大小对流水线机制性能的影响，设置归并路数为 16 路，溢写文件大小为 40 kB，以获得较优的结果。实验结果表明，作业的整体运行时间随着 FGB 大小的增加而缩短，主要原因是较小细粒度数据块在处理之后，数据的大小仍没有达到溢写的阈值，一直被存放在内存中等待下一个细粒度数据块，因此，大数据块的作业运行时间要优于小数据块。

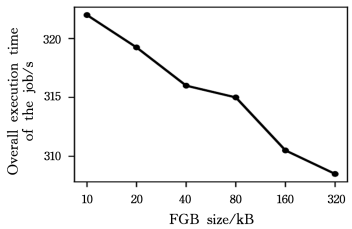

图 6 FGB 大小对性能的影响

Fig. 6 Impact of FGB size on performance

图 7 给出了在 BST 数据集上执行 WordCount 算法时归并路数对流水线机制性能的影响，设置 FGB 的大小为 160 kB，溢写文件大小为 40 kB，以获得较优的结果。实验结果表明，随着归并路数的增大，作业的整体运行时间先缩短后增加。这是因为归并路数过大，需要等待的溢写文件数也就越多，归并需要的内存开销也越大，同时会带来数据传输延迟；如果归并路数过小，则多次归并带来的额外开销同样也会造成过多的数据传输延迟。

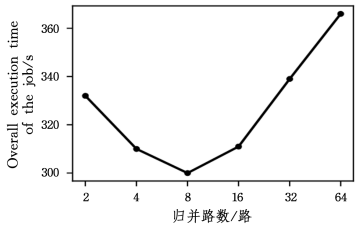

图 7 归并路数对性能的影响

Fig. 7 Impact of merging routes on performance

图 8 给出了在 BST 数据集上执行 WordCount 算法时数据传输粒度对流水线机制性能的影响，设置 FGB 的大小为 160 kB，归并路数为 16 路，以获得较优的结果。实验结果表明，随着数据传输粒度的增大，作业整体的运行时间先缩短后增加。其主要原因是如果数据传输粒度过大，将带来数据传输延迟，并由此导致 Reducer 任务计算延迟；如果数据传输粒度过小，将增加数据传输开销以及 Reducer 任务的读写开销。

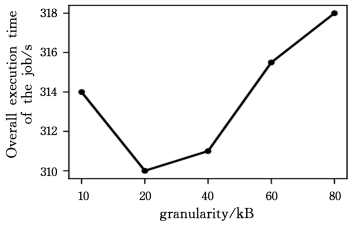

图 8 数据传输粒度对性能的影响

Fig. 8 Impact of data transmission granularity on performance

5.3 性能对比

本节将数据传输流水线优化机制与默认框架进行对比，采用的数据集是 BTS 数据集和 Konect 数据集，在这两种数据集上执行 3 种算法并评价整体性能。

图 9 给出了在 BST 数据集上执行 WordCount 算法的对比结果，数据传输流水线优化机制的归并路数为 16 路，数据传输粒度为 40 kB。实验结果表明，数据传输流水线优化机制在作业整体时间上要优于默认框架，随着数据规模的增大，数据传输流水线优化机制与默认框架的时间差逐渐增大，流水线机制比默认框架的整体性能平均提高了 1.6%。

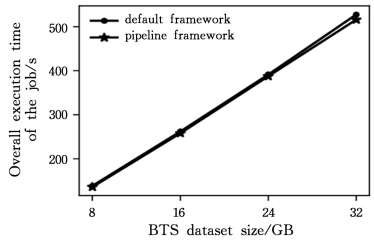

图 9 执行 WordCount 算法的比较结果

Fig. 9 Results comparison with WordCount algorithm

图 10 给出了在 Konect 数据集上执行 PageRank 算法的对比结果，归并路数为 32 路，数据传输粒度为 100 MB。实验结果表明，随着数据集规模的增大，数据传输流水线优化机制与默认框架的作业整体运行时间差逐渐增大，流水线优化机制比默认框架的整体性能平均提高了 54.1%。

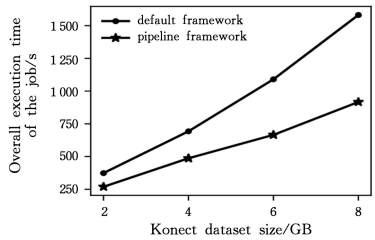

图 10 执行 PageRank 算法的比较结果

Fig. 10 Results comparison with PageRank algorithm

图 11 给出了在 Konect 数据集上执行 Inverted Index 算法的对比结果，归并路数为 32 路，数据传输粒度为 50 MB。实验结果表明，随着数据规模的增大，数据传输流水线优化机制与默认框架的作业整体运行时间差也在逐渐增大，流水线优化机制比默认框架的整体性能平均提高了 60.2%。

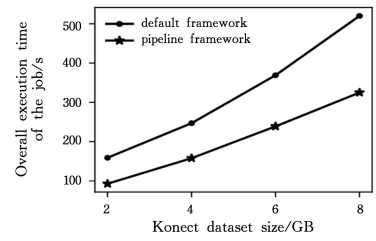

图 11 执行 Inverted Index 算法的比较结果

Fig. 11 Results comparison with Invert Index algorithm

综合分析以上 3 种不同算法的对比结果可知，当 Shuffle 中间数据量较大时，数据传输流水线优化机制明显优于默认框架（见图 10 和图 11）；当 Shuffle 中间数据量较小时，所提机

制也不会导致整体性能下降(见图 9)。这是因为,较大的 Shuffle 中间数据量所导致的数据延迟也较大,对 MapReduce 整体性能带来显著影响;较小的 Shuffle 中间数据量所导致的数据延迟较小,对 MapReduce 整体性的影响也较小。

结束语 中间数据传输延迟对 MapReduce 框架的整体性能具有重要影响。针对该问题,本文提出了一种面向 MapReduce 的数据传输流水线优化机制,通过将 Mapper 端的有效计算、中间数据传输和 Reducer 端的有效计算解耦,以流水线的方式重叠执行,隐藏中间数据传输导致的延迟开销,详细描述了中间数据传输流水线优化的实现机制和策略。实验结果表明,在 Shuffle 中间数据量较大时流水线优化机制能够显著提高数据传输的效率。

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] WANG S, WANG H J, QIN X P. Architecting Big Data: Challenges, Studies and Forecasts[J]. Chinese Journal of Computers, 2011, 34(10): 1741-1752.

[3] ZAHARIA M, CHOWDHURY M, FRANKLIN M, et al. Spark: cluster computing with working sets[C]// Usenix Conference on Hot Topics in Cloud Computing. Berkeley: USENIX Association, 2010: 1-12.

[4] XUN Y L, ZHANG J F, QIN X. Data Placement Strategy for MapReduce Cluster Environment[J]. Jounary of Software, 2015 (8): 2056-2073.

[5] AHMAD F, LEE S, THOTTETHODI M, et al. MapReduce with communication overlap (MaRCO)[J]. Journal of Parallel & Distributed Computing, 2013, 73(5): 608-620.

[6] YU W, WANG Y, QUE X. Design and Evaluation of Network-Levitated Merge for Hadoop Acceleration[J]. IEEE Transactions on Parallel & Distributed Systems, 2014, 25(3): 602-611.

[7] CHOWDHURY M, ZAHARIA M, MA J, et al. Managing data transfers in computer clusters with orchestra[J]. ACM SIGCOMM Computer Communication Review, 2011, 41(4): 98-109.

[8] LI J J, WU J, YANG X L, et al. Optimizing MapReduce Based on Locality of K-V Pairs and Overlap between Shuffle and Local Reduce[C]// IEEE International Conference on Parallel Processing. 2015: 939-948.

[9] RAHMAN M W, ISLAM N S, LU X, et al. A Comprehensive Study of MapReduce Over Lustre for Intermediate Data Placement and Shuffle Strategies on HPC Clusters[J]. IEEE Transactions on Parallel & Distributed Systems, 2017, PP(99): 1-1.

[10] ARSLAN E, SHEKHAR M, KOSAR T. Locality and Network-Aware Reduce Task Scheduling for Data-Intensive Applications [C]// International Workshop on Data-intensive Computing in the Clouds. IEEE, 2014.

[11] CAO Y, WANG H. Communication optimization for intermediate data of MapReduce computing model[J]. Jounary of Computer Applications, 2018, 38(4): 1078-1083.

[12] LIU S, WANG H, LI B. Optimizing Shuffle in Wide-Area Data Analytics[C]// 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS). IEEE Computer Society, 2017.

[13] SEO S, JANG I, WOO K, et al. HPMR: Prefetching and Pre-Shuffling in Shared MapReduce Computation Environment [C]// Proc of the IEEE International Conference on Cluster Computing and Workshops. 2009: 1-8.

[14] ZHANG S, HAN J, LIU Z, et al. Accelerating MapReduce with Distributed Memory Cache [C]// International Conference on Parallel & Distributed Systems. 2009: 472-478.

[15] WANG B, JIANG J, YANG G. mpCache: Accelerating MapReduce with Hybrid Storage System on Many-Core Clusters [C]// NPC 2014. 2014: 220-233.

[16] LI J, LIN X, CUI X, et al. Improving the Shuffle of Hadoop MapReduce[C]// IEEE International Conference on Cloud Computing Technology & Science. IEEE, 2014.

[17] QI K Y, HAN Y B, ZHAO Z F. MapReduce Intermediate Result Cache for Concurrent Data Stream Processing[J]. Journal of Computer Research and Development, 2013, 50(1): 111-121.

[18] WANG J, QIU M, GUO B, et al. Phase-Reconfigurable Shuffle Optimization for Hadoop MapReduce[J]. IEEE Transactions on Cloud Computing, 2015, 8(2): 418-431.

[19] TAN J, CHIN A, HU Z Z, et al. DynMR: dynamic MapReduce with ReduceTask interleaving and MapTask backfilling[C]// Proc of the Ninth European Conference on Computer Systems. ACM, 2014: 1-14.

[20] QIN X P, WANG H J, DU X Y. Big Data Analysis-Computition and Symbiosis of RDBMS and MapReduce[J]. Journal of Software, 2012, 23(1): 32-45.

[21] KAMBATLA K, KOLLIAS G, KUMAR V, et al. Trends in Big Data Analytics [J]. Journal of Parallel & Distributed Computing, 2014, 74(7): 2561-2573.

[22] HO L Y, WU J J, LIU P. Optimal Algorithms for Cross-Rack Communication Optimization in MapReduce Framework [C]// IEEE International Conference on Cloud Computing. 2011.

[23] YAN W M, WANG W M. Data Structure[M]. Beijing: Tsinghua University Press, 2013: 297-304.

ZHANG Yuan-ming, born in 1977, Ph.D, associate professor. His main research interests include parallel computing and cloud computing.

XIAO Gang, born in 1965, Ph.D, professor. His main research interests include cloud computing and intelligent information system.