基于 MapReduce 的增量矩阵乘法设计①

张沛然, 徐 云

(中国科学技术大学 计算机科学与技术学院, 合肥 230027)
(中国科学技术大学 安徽省高性能计算重点实验室, 合肥 230027)

摘 要: 矩阵乘法是许多应用中的核心计算, 在这些应用中只是少量矩阵元素发生改变, 如果全量重新计算则工作量很大, 因此增量计算是解决该问题的有效手段. 本文提出了一种基于 MapReudce 模型的增量矩阵乘法计算方法, 以及计算矩阵中变化元素的高效识别方法, 通过利用矩阵元素的摘要信息快速计算出变化元素, 然后将矩阵乘法计算过程转换为一系列等价的连接问题, 实现了一种有效的矩阵乘法增量计算. 对于矩阵元素变化率较小的情形, 计算实验表明提出的方法计算时间上明显优于全量重新计算方法.

关键词: 矩阵乘法; 增量计算; MapReduce

Design of Incremental Matrix Multiplication Based on MapReduce

ZHANG Pei-Ran, XU Yun

(School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China)
(Key Laboratory of High Performance Computing of Anhui Province, Hefei 230027, China)

Abstract: Matrix Multiplication is a core computation in many applications, it has been observed that only a very small fraction of the matrix elements gradually change in these applications, and the full recomputation costs a lot, thus, incremental processing is a promising approach to handle this problem. This paper proposes a design of incremental matrix multiplication based on MapReduce with a fast method to capture the changes of the matrix, by using the abstract informationof matrix elements to calculate the changes of the matrix effectively and transforms this problem to series of equivalent join problems, which realizes a efficient matrix multiplication in an incremental way. Experimental results show significant performance improvements of our method compared to full recomputation when the change ratio is small.

Key words: matrix multiplication; incremental computing; MapReduce

矩阵乘法广泛应用于机器学习、推荐系统、社交网络分析、图挖掘等领域. 在分析社交网络中用户关系时, 对社交网络图的邻接矩阵进行一次矩阵自乘, 可以计算出“朋友的朋友”[1]. 推荐领域中, 通过用户对物品的打分记录进行矩阵分解来预测还没有评分物品的分数, 而矩阵分解是用迭代矩阵乘法来求得的[2]. 许多图计算问题如最短路径、强连通分量、图匹配等, 矩阵乘法都起到了构建基本单元的作用[3].

随着互联网及云计算应用的迅速发展, 一些求解问题需要处理的矩阵规模也越来越大, 传统的并行方法[4]由于扩展性差而不能满足实际需求, 迫切需要高度可扩展的分布式矩阵乘法. 近年来兴起的 MapReduce 是 Google 公司推出用来处理海量数据的分布式并行计算框架[5]. 该框架具有易于编程、高扩展性等特点, 被广泛应用在各个领域[6]. 因此, 可以利用 MapReduce 框架来实现大规模分布式矩阵乘法.

如今, 应用中处理的数据集不仅仅规模大, 而且具有不断变化的特点, 这使得矩阵乘法的输入也不断改变, 已经计算好的结果因此而失效, 需要进行更新. 例如, 社交网络中用户之间的好友关系会随着用户的喜好而发生变化, 社交网络图中不断有边被删除或添加, 图的邻接矩阵因此而改变, 基于原来的邻接矩阵

① 基金项目:国家自然科学基金(61033009)

收稿时间:2016-01-08;收到修改稿时间:2016-03-01 [doi:10.15888/j.cnki.csa.005321]

算出来的“朋友的朋友”不再准确. 相关研究表明这种变化的数据相对与整个数据集往往比例很小[11], 在一段时间内, 社交网络图中删除或添加的边数量远远小于整个图边的数量, 即社交网络图邻接矩阵中变化的元素个数远远小于总的元素个数, 增量计算成为解决这类问题的有效手段. 鉴于此, 本文设计了一种基于 MapReduce 框架的增量矩阵乘法计算方法, 用来处理下一次计算时, 输入的矩阵相对与上一次计算的输入矩阵元素发生少量变化的情形. 该方法只对那些受变化数据影响的结果矩阵元素值进行重新计算. 以矩阵自乘为例, 实验表明, 相对于全量重新计算, 该方法能够有效缩短程序执行时间.

本文组织结构如下: 第 1 节中对相关工作进行介绍; 第 2 节对基于 MapReduce 框架的增量矩阵乘法计算方法进行了详细的描述与分析; 第 3 节通过实验验证了该方案的有效性; 第 4 节对全文进行总结, 并对下一步研究工作进行展望.

1 相关工作

1.1 MapReduce 框架

一个 MapReduce 程序主要由两个函数构成: Map 函数和 Reduce 函数. Map 函数读取($K1$, $V1$)键值对, 产生零到多个中间($K2$, $V2$)键值对集合. MapReduce 框架根据键 $K2$, 对所有的($K2$, $V2$)键值对分组. Reduce 函数读取键 $K2$ 和 $K2$ 对应的值列表$\{V2\}$, 规约后产生结果($K3$, $V3$)键值对. Apache 的 Hadoop 是 MapReduce 模型的开源实现, 主要包括 MapReduce 计算引擎和用来存储计算输入数据和输出结果的分布式文件系统 HDFS, 本文的研究工作也是基于 Hadoop 平台实现的.

1.2 基于 MapReduce 的矩阵乘法

基于 MapReduce 的大规模矩阵乘法主要有内积法、外积法和分块法. 内积法中, 将左矩阵的行与右矩阵相应列进行点积运算, 因此结果矩阵中每一个元素都可以独立于其他元素来计算, 该方法并行度高, 但会产生大量中间数据, 当矩阵规模达到一定规模时, 性能很低; 文献[7-9]中, 采用外积法来求解矩阵乘法. 通过求左矩阵的列与右矩阵相应行的外积来计算, 从而得到结果矩阵的部分结果, 最后对各个部分结果求和. 外积法产生更少的中间数据, 但损失了并行粒度; 文献[10]中, 采用分块的思想来计算大规模矩阵乘法, 通过计算小块的乘积来计算大矩阵的乘积. 实验表明该方法并行效率与基于 MPI 实现的矩阵乘法接近, 但具有更好的扩展性和容错性. 这类工作的目标是设计基于 MapReduce 框架的高效分布式矩阵乘法, 而没有考虑矩阵数据发生少量变化之下如何快速重新计算的问题.

1.3 增量计算

文献[12-14]中分别设计了新的分布式计算框架来支持增量计算, 这使得原来基于 MapReduce 的应用需要重新实现. 文献[15]中设计了一种基于单机的内存增量 MapReduce 框架. 文献[16,17]分别设计了基于 MapReduce 的通用分布式增量计算框架 Incoop 和 i2MapReduce. Incoop 采用增量文件系统 iHDFS 来识别变化的数据, 能够支持大多数 MapReduce 应用, 但它是基于任务级结果复用, 即使 Map 任务的输入数据只有少量变化, 整个 Map 任务仍需重新计算. 不同于 Incoop, i2MapReduce 能支持(key, value)对级别的细粒度增量处理, 因此能更有效地避免重复数据的计算, 但是它需要用户提供前后两次计算输入数据中变化的部分. Incoop 和 i^{2}MapReduce 需要对 MapReduce 框架进行修改和调整, 通过算法来支持矩阵变化元素的自动识别, 但存在适用性和推广的问题. 文献[18]采用 R 语言实现了一个支持部分线性代数操作增量计算的库, 该文献中以变化的行或列为基本单元进行矩阵乘法的增量分析与处理, 而本文实现的方法以变化元素为粒度进行增量分析, 更能有效的减少不必要的计算.

2 基于MapReduce模型的增量矩阵乘法

为了便于叙述, 本文以矩阵自乘为例进行分析, 但本文提出的增量矩阵乘法适用于任何一般的矩阵相乘. 本节先给出一些符号定义并说明矩阵的存储结构; 其次给出增量矩阵乘法设计思路和计算代价分析; 然后给出在 MapReduce 模型下实现该思路的主要过程; 最后给出如何识别前后两次计算输入矩阵中变化的元素.

2.1 符号定义

设 $A=(a_{ij})$ 是一个 $n\times n$ 矩阵, 那么规定矩阵 A 与矩阵 A 的乘积是一个 $n\times n$ 矩阵 $B=\left(b_{ij}\right)$, 记做 $B=AA$, 其中:

$$b_{ij}=a_{i1}a_{1j}+a_{i2}a_{2j}+\cdots+a_{in}a_{nj}=\sum_{k=1}^{n}a_{ik}a_{kj}(i,j=1,2,\cdots n)$$

设矩阵 A 的行向量表示为 $A_{row}=(a_1^{\mathrm{T}}a_2^{\mathrm{T}}\cdots a_n^{\mathrm{T}})$; 列向量表示为 $A_{col}=(a_1a_2\cdots a_n)$. 其中 a_i^{T} 表示矩阵 A 的

第 i 行，a_i 表示矩阵 A 的第 i 列. 设 $A'=(a'_{ij})$ 是矩阵 A 发生少量元素值变化后的矩阵，同理有 $B'=A'A'$. 增量矩阵乘法问题即研究如何通过减少不必要的计算来快速计算出矩阵 B'. 相对的，本文把直接通过矩阵 A' 乘以 A' 来计算新的结果矩阵 B' 这种方式称为全量重新计算.

本文中，矩阵存储在分布式文件系统 HDFS 中，主要用到了两种矩阵存储结构. 对于一般的非稀疏矩阵，采用行向量的形式存储，矩阵被表示为记录集合 (i,a_i^{T})，i 代表行号；对于稀疏矩阵，采用三元组的形式存储矩阵的每一个非零元素，矩阵被表示为记录集合 (i,j,a_{ij})，i 和 j 分别表示矩阵的第 i 行、第 j 列.

2.2 增量矩阵乘法设计思路

本小节用首先通过例子来说明全量重新计算这种方式在计算矩阵 B' 的过程中存在哪些不必要的计算；然后论述如何去减少这些不必要计算，以此来说明本文提出的增量矩阵乘法设计思路；最后分析总结这种思路的合理性.

图 1 中，A' 是矩阵 A 中一个元素发生变化后的矩阵. 可以发现即使只有一个元素发生变化，但 B' 相对于 B 仍然存在一整行和一整列的不同元素. 根据矩阵乘法的定义，结果矩阵中的任意元素 b'_{ij} 是由左矩阵的 i 行向量 a_i^{T} 与右矩阵 j 列向量 a'_j 点积确定的，那么左矩阵中没有出现变化的一行与右矩阵中没有出现变化的一列进行向量点积得到的元素值也是不会变化的，因此矩阵 B' 中这部分元素无需重新计算. 这正是文献[18]中增量矩阵乘法的思路.

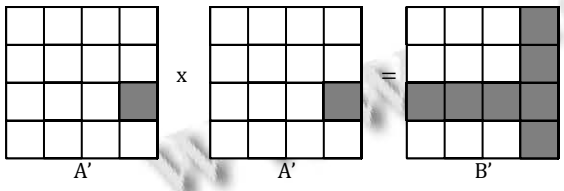

图 1　矩阵相乘示意(灰色部分表示相对上一次计算值发生变化的元素)

显然，全量重新计算矩阵 B' 的时间复杂度为 $O(n^3)$. 假设矩阵 A' 相对于 A 有 k 行和 k 列存在不同元素，那么按照上述方法进行增量计算的时间复杂度为 $O(n^3-(n-k)(n-k)n)$ 即 $O(2kn^2-k^2n)$. 考虑一种情况，当 A' 相对于 A 每一行和每一列有且仅有一个元素不一样，此时 $k=n$，计算复杂度与全量重新计算的复杂度一样，但实际上发生变化的元素个数 $(2n-1)$ 远小于 n^2，因此这种以整行、整列为粒度进行增量计算的方式在一些情况下并不能起到加速计算的作用.

本文从更细的层次上考虑. 以计算 b_{31} 和 b'_{31} 为例，由矩阵乘法定义有 $b_{31}=a_{31}a_{11}+a_{32}a_{21}+a_{33}a_{31}+a_{34}a_{41}$；$b'_{31}=a'_{31}a'_{11}+a'_{32}a'_{21}+a'_{33}a'_{31}+a'_{34}a'_{41}$. 可以看出 b'_{31} 与 b_{31} 不同的原因仅仅是 $a'_{34}a'_{41}$ 与 $a_{34}a_{41}$ 这两项值不同，而这是由于 a'_{34} 与 a_{34} 不同. 事实上 b'_{31} 可以按照另一种方式获得：$b'_{31}=b_{31}+a'_{34}a'_{41}-a_{34}a_{41}$. 由于直接复用上一次计算结果 b_{31}，这种方式减去了不必要的计算 $a'_{31}a'_{11}+a'_{32}a'_{21}+a'_{33}a'_{31}$，因为该计算中涉及到矩阵元素与 $a_{31}a_{11}+a_{32}a_{21}+a_{33}a_{31}$ 中对应位置的元素完全是一样的. 同理，对于 B' 中其他元素也可以按照类似的方式计算出来. 我们发现这种以变化元素为粒度进行分析的增量计算方式更能有效地减少不必要的计算，尤其适用于矩阵规模很大而变化的元素个数很少的情况. 因此，本文基于此来设计基于 MapReduce 模型的增量矩阵乘法.

为了达到上述目的，首先给出矩阵 B' 的另一种计算方式，由矩阵乘法的结合律和分布律有:

$$\Delta A=A'-A$$

$$B'=A'\times A'=(A+\Delta A)(A+\Delta A)$$

$$B'=B+\Delta A\times A+A\times\Delta A+\Delta A\times\Delta A$$

虽然上式将矩阵 B' 的计算分成了几部分矩阵乘法，然后进行矩阵求和，但是我们注意到 ΔA 是一个很稀疏的矩阵，因此在计算 $\Delta A\times A+A\times\Delta A+\Delta A\times\Delta A$ 时有很大的优化空间. 下面以 $\Delta A\times A$ 为例子来探讨如何计算这部分.

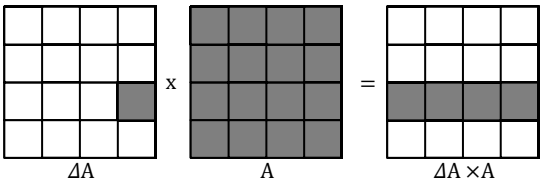

图 2　矩阵相乘示意(灰色部分表示非零元素)

如图 2 中，用 $_{\Delta}a_{34}$ 表示 ΔA 中唯一的非零元素. 根据矩阵乘法的定义可知 $\Delta A\times A$ 中 (i,j) 处元素值为:

$$\sum_{k=1}^{n}{}_{\Delta}a_{ik}\times a_{kj}$$，图 2 中 $n=4$.

观察上式可知在计算 $\Delta A\times A$ 的过程中，左矩阵 ΔA 中任意 (i,j) 处元素都会参与 n 次乘法计算. 然而，ΔA 是一个稀疏矩阵，由于大量零元素的存在，有很多

不需要的计算. 很容易分析出, 假设 ΔA 中只有 k 个非零元素, 那么 $\Delta A\times A$ 的有效计算量实际为 $O(kn)$. 如图 2 中 ΔA 中只有一个非零元素 $_{\Delta}a_{34}$, $\Delta A\times A$ 中只有第三行元素非零, 而且这行 4 个元素值分别是 $_{\Delta}a_{34}\times a_{41}$、$_{\Delta}a_{34}\times a_{42}$、$_{\Delta}a_{34}\times a_{43}$、$_{\Delta}a_{34}\times a_{44}$, 一共只需 4 次乘法计算. 因此, 如何使 $\Delta A\times A$ 的计算复杂度降为 $O(kn)$, 这是本文接下来研究的问题.

前文提到对于稀疏矩阵, 可以采用三元组的形式来存储矩阵中所有非零元素. 我们发现如果将 ΔA 和 A 分别表示为两张关系表 $R1$ 和 $R2$, 这两张表的属性均为 $\{row,col,val\}$ 且矩阵里面的每个非零元素 $(i,j,value)$ 一一对应表中的记录, 那么 $\Delta A\times A$ 等价于连接问题, 其 SQL 表达形式如下:

```
SELECT R1.row, R2.col, sum(R1.val*R2.val)
FROM R1, R2
WHERE R1.col = R2.row
GROUP BY R1.row, R2.col
```

上述 SQL 查询表达式中通过将满足 where 子句条件的 $R1$ 和 $R2$ 中记录一一连接并对 val 属性值之间做乘法, 产生中间记录, 然后以 (row,col) 为键进行分组求和, 这样便得到了 $\Delta A\times A$ 中任意 (row,col) 处的元素值. 由于表中元素都是非零元素, $R1$ 有 k 条记录并且有 where 子句作为连接条件, 所以连接的代价为 $O(kn)$, 这意味着 $\Delta A\times A$ 的计算复杂度可以降为 $O(kn)$. 对于 $A\times\Delta A$ 和 $\Delta A\times\Delta A$ 的计算分析类似, 这里不再叙述. 本文后面用 $A\bowtie\Delta A$、$\Delta A\bowtie A$、$\Delta A\bowtie\Delta A$ 来分别表示上面三部分等价的连接问题.

至此, 我们总结增量矩阵乘法设计思路, 如下:

第一步将 B' 的计算过程进行了转换, 从而达到以变化元素为粒度进行增量分析与计算的目的; 第二步, 针对 ΔA 是稀疏矩阵这一特点, 将 $\Delta A\times A$、$A\times\Delta A$、$\Delta A\times\Delta A$ 的计算分别转换为等价的连接问题, 达到降低计算复杂度的目的.

2.3 MapReduce 框架下增量矩阵乘法的实现

本小节给出基于 MapReduce 模型的增量矩阵乘法具体实现. 我们采用的平台是 Hadoop.

首先需要说明的是: 对于矩阵 A, 本文是按行来进行划分并依次分配给每个 Mapper 任务. 假设有 m 个 Mapper 任务, 则有 $A=\{A_1,A_2,\cdots,A_m\}$; 而对于 ΔA, 我们假设其可以存放在内存中, 在计算开始时, 每个 Mapper 任务都会加载一份 ΔA 到内存里. 这主要基于以下两点考虑: 第一, ΔA 中非零元素的个数远远小于 A 中元素个数, 占用的内存空间很小. 第二, 由于每个 Mapper 任务都有一份 ΔA, 由连接操作的性质可知, 各任务便可独立进行 $A_i\bowtie\Delta A$ 和 $\Delta A\bowtie A_i$ 操作并且能保证结果的准确性. 另外, 我们注意到一种分布式存储系统 HBase[19], 它建立在 HDFS 之上, 能够高效支持数据的存储与实时查询, 并且 MapReduce 程序中可以直接读写 HBase 中的数据, 因此, 也可以将 ΔA 中所有非零元素以表的形式存储到 HBase 中, 这样各个 Mapper 执行任务时可以随时读取 ΔA 中任意元素而无需先将 ΔA 加载到内存里.

下面来说明内存中如何存储 ΔA 中所有非零元素, 使得 Mapper 任务 M_i 能够高效的进行 $A_i\bowtie\Delta A$ 和 $\Delta A\bowtie A_i$ 连接操作. 在计算 $A_i\bowtie\Delta A$ 时, A_i 中任意元素 (i,j,val_1) 与 ΔA 中任意元素 (row,col,val_2) 能够连接的条件是 $j=row$, 从而产生 $A\times\Delta A$ 中位于 (i,col) 处元素值的部分结果 val_1*val_2, 这要求任务 M_i 处理 A_i 中的每个元素时都需要遍历一遍 ΔA, 找出 ΔA 中满足条件的元素与其连接. 为了加速查询, 本文采用哈希表来存储 ΔA 中所有元素, 该哈希表中键为元素的行下标, 哈希表中每一项表示为键值对 $\langle row,List\{(col,val)\}\rangle$, 即每个键对应一个列表, 该列表含有 ΔA 中 row 行所有的元素及元素的列号. 同理, $\Delta A\bowtie A_i$ 与 $A_i\bowtie\Delta A$ 分析类似, 但由于该连接顺序与 $A_i\bowtie\Delta A$ 相反, ΔA 中任意元素 (row,col,val_2) 与 A_i 中任意元素 (i,j,val_1) 连接的条件是 $col=i$, 因此, 在计算 $\Delta A\bowtie A_i$ 时, 以 ΔA 中元素的列号作为哈希表的键来构建哈希表, 哈希表中每一项表示为 $\langle col,List\{(row,val)\}\rangle$. 这样, 在计算 $A_i\bowtie\Delta A$ 和 $\Delta A\bowtie A_i$ 时, 分别以不同的方式构建哈希表来存储 ΔA 中的元素, 来加速连接中的查询操作, 后面分别用 H_r 和 H_l 来表示这两个哈希表.

整个增量矩阵乘法实现需要两轮 MapReduce 作业. 第一轮用来计算 $\Delta A\times A+A\times\Delta A+\Delta A\times\Delta A$. 第二轮用第一轮结果来更新 B ,从而得到新的结果矩阵 B'.

作业一的输入数据为矩阵 A, 该矩阵按照行向量的形式存储在 HDFS 中. Mapper 任务 M_i 首先调用一次 setup 函数, 按照前面提到的方式构建哈希表 H_r 和 H_l. 其次, 依次处理 A_i 中的每一行行向量, 对每行行向量调用 map 函数, 如图 3 所示. map 函数输入为(行号, 行向量)键值对, 按照前文提到的连接条件依次处理行向

量中每个元素与 ΔA 的连接, 实现 $A \bowtie \Delta A$ 和 $\Delta A \bowtie A$, map 的输出为键值对 $\langle (row,col), val \rangle$, 表示结果矩阵 (row,col) 处元素的部分结果. 对于 $\Delta A \bowtie \Delta A$, 由于 ΔA 中非零元素很少, 我们单独启动一个 Mapper 任务, 由该任务中的 setup 函数来执行, 该函数首先根据 ΔA 构建表 H_r 和 H_l, 然后直接计算 $\Delta A \bowtie \Delta A$, 如图4所示. 等所有的 Mapper 任务结束之后, 框架会根据键 (row,col) 将所有任务产生的中间记录发送给相应的 Reduce 任务, 如图 5 所示, reduce 函数的输入为 $\langle (row,col), list(val) \rangle$, 对列表中值累加求和, 最后输出 $\langle (row,col), val \rangle$ 到 HDFS 中, 这样便得到了 $\Delta A \times A + A \times \Delta A + \Delta A \times \Delta A$ 的计算结果.

Algorithm 1 map() in Mapper for $A \bowtie \Delta A$ and $\Delta A \bowtie A$

Input: KV Pair $\langle i, a_i^{\mathrm{T}} \rangle$
Output: KV Pair $\langle (row,col), value \rangle$

```
1: flag = false
2: if H_l.containsKey(i) then
3:    flag = true
4: end if
5: for j ← 1  to  a^T_rowId.length
6:    if H_r.containsKey(j) then
7:       for all (col, val_2) in H_r(j).List
8:          Output ⟨(i, col), a^T_rowId(j) * val_2⟩
9:       end for
10:   end if
11:   if flag == true then
12:      for all (row, val_2) in H_l(rowId).List
13:         Output ⟨(row, j), a_i^T(j) * val_2⟩
14:      end for
15:   end if
16: end for
```

图 3　job1 Mapper 中的 map 函数伪代码

Algorithm 2 setup() in Mapper for $\Delta A \bowtie \Delta A$

Input: ΔA
Output: KV Pair $\langle (row,col), value \rangle$

```
1: init H_r , H_l
2: for all (row,col,val) in ΔA
3:    insert (col,val) in H_r(row).List
4:    insert (row,val) in H_l(col).List
5: end for
6: for all (joinKey, H_l(joinKey).List) in H_l
7:    if H_r.containsKey(joinKey)
8:       for all (row, val_1) in H_l(joinKey).List
9:          for all (col, val_2) in H_r(joinKey).List
10:            Output ⟨(row,col), val_1 * val_2⟩
11:         end for
12:      end for
13:   end if
14: end for
```

图 4　Mapper 中的 setup 函数伪代码

Algorithm 3 reduce() in Reducer for job 1,2

Input: KV Pair $\langle (row,col), List(val) \rangle$
Output: KV Pair $\langle (row,col), val \rangle$

```
1: sum = 0
2: for all v in List
3:    sum += v
4: end for
5: Output ⟨(row,col), sum⟩
```

图 5　job1,2 Reducer 中的 reduce 函数伪代码

作业二的输入为作业一的输出以及上一次计算结果矩阵 B, 该作业的 Mapper 任务输入为 $\langle (row,col), val \rangle$, 不做处理, 直接输出. 该作业的 Reducer 与第一轮作业的 Reducer 一样, 对元素值进行累加求和, 如图 5 所示. 这样, 通过两轮 MapReduce 过程, 我们得到最终的结果矩阵 B'.

2.4 矩阵变化元素识别

本文通过一个 MapReduce 作业来获得 ΔA. Mapper 任务的输入为行向量 $\langle row, a_i^{\mathrm{T}} \rangle$ 或 $\langle row, a_i^{'\mathrm{T}} \rangle$, 分别来自矩阵 A 和矩阵 A', 不做处理直接输出. 以行向量而不是以矩阵元素作为输入输出, 可以减少 Mapper 任务产生的中间记录数量, 进而减少框架内部的排序代价. 然后框架根据键 row, 将这些记录发送给不同的 Reduce 任务, Reduce 任务对来自两个矩阵中同一行的行向量分别计算一个哈希值, 如果两个哈希值相同, 则认为该行没有不同的元素, 否则, 再通过一次向量减法得到 ΔA 中位于该行的所有非零元素. 如图 6 所示, 通过先计算向量的哈希值来减少不必要的向量减法操作, 达到加速计算的目的. 对于一些特殊的矩阵, 如图的邻接矩阵, 可以进一步优化, 因为矩阵元素值为 0 或 1, Mapper 任务中可以用位向量来代替行向量输出, 起到数据压缩的作用, 同时减少框架

Shuffle 过程中读写 IO 的开销, Reduce 任务中可以对位向量进行异或计算来得到 ΔA 中非零元素.

Algorithm 4 reduce() in Reducer for job:Get ΔA

Input: KV Pair $\langle rowId, List(a^{T}_{rowId}) \rangle$

Output: KV Pair $\langle (row, col), val \rangle$

1: a^{T}_{row} = $List$.getFirst(), $a^{'T}_{row}$ = $List$.getSecond()

2: $h1 = Hash(a^{T}_{row})$, $h2 = Hash(a^{'T}_{row})$

3: if $h1 \neq h2$

4: $\Delta a^{T}_{row} = a^{'T}_{row} - a^{T}_{row}$

5: Output all nonzero (row, col, val) in Δa^{T}_{row}

6: end if

图 6 Reducer 中的 reduce 函数伪代码

3 实验分析

本节按照外积法实现了全量矩阵乘法, 与本文提出的增量矩阵乘法进行对比实验. 以矩阵自乘为例进行了分析, 实验编写了一个图邻接矩阵生成器, 通过生成不同规模的矩阵以及固定矩阵规模而设置矩阵变化元素的比例来验证增量矩阵乘法性能, 实验中以程序执行时间为参考进行性能分析, 其中增量计算的时间为前文提到的两轮 MapReduce 作业时间及计算 ΔA 的时间之和. 另外, 在计算 ΔA 时, 采用了前面提到的针对邻接矩阵的优化方法. 实验环境为中国科大云平台[20], 在上面搭建了一个集群, 集群采用 Hadoop 1.2.1 版本, 由 6 个云主机构成, 其中 5 个为计算节点, 1 个为主节点, 每个节点为 8 核 16G 内存, 操作系统采用 Ubuntu 12.04.

首先分析矩阵规模对增量矩阵乘法性能的影响. 实验中分别生成了规模为 $2^{10} \times 2^{10}$ 至 $2^{14} \times 2^{14}$ 的五组矩阵, 然后进行矩阵自乘计算, 将所得的结果 B 存到 HDFS 中. 接着固定变化比例为 0.00001, 重新生成新的五组矩阵, 然后计算 B'. 实验记录了不同矩阵规模下增量计算 B' 的执行时间以及全量重新计算 B' 的时间, 结果如图 7 所示.

从图 7 中可以看出, 当矩阵规模不大时, 增量矩阵乘法计算效率与全量重新计算效率接近, 这主要是各个节点的启动、任务的初始化需要一定的时间. 当矩阵规模增大时, 增量计算的效率明显优于全量重新计算, 且随着矩阵规模的继续增大, 增量计算的时间增长比全量计算的时间增长缓慢, 这与我们前文计算复杂度分析情况一致.

然后我们考虑输入矩阵变化比例对增量矩阵乘法性能的影响. 实验中, 固定矩阵的规模为 $2^{14} \times 2^{14}$, 然后用矩阵生成器分别产生变化比例为 0.0001、0.0003、0.0005、0.0008、0.001 的新矩阵, 分别记录下增量矩阵乘法计算的时间, 如图 8 所示.

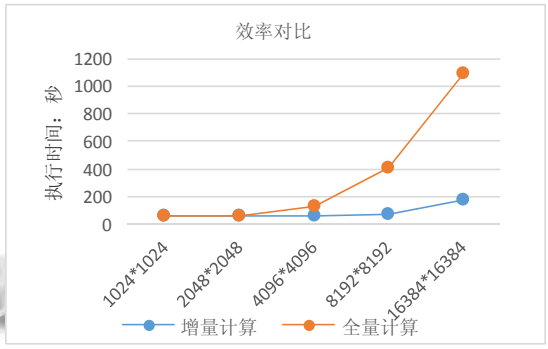

图 7 增量计算与全量重新计算效率对比

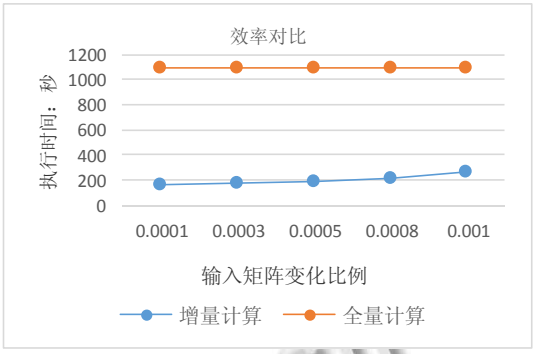

图 8 输入矩阵变化比例对增量计算的影响

从图 8 中可以看出, 随着变化比例的增加, 增量矩阵乘法计算的时间也缓慢增加. 实验中, 变化比例一直小于千分之一, 可以预测当变化比例达到一定程度时, 增量计算的性能将比全量重复计算低. 因此, 增量矩阵乘法适合输入数据变化比例较小的情况.

4 结语

本文提出了一种基于 MapReduce 模型的增量矩阵乘法计算方法, 不需对 MapReduce 框架进行改动, 通过算法本身来识别矩阵中变化的元素, 以变化元素为粒度进行增量分析与处理. 实验表明了该方法在矩阵变化率较低的情形下, 具有良好的性能和实用意义. 如何根据矩阵的变化比例选择增量计算还是全量重新计算以及若增量计算过程中 ΔA 矩阵非零元素太多而放不下内存时, 采用第 2 章提到的利用 HBase 来存储

的方案对程序性能有哪些影响, 是下一步要研究的问题.

参考文献

1 Fischer MJ, Meyer AR. Boolean matrix multiplication and transitive closure. Proc. of the 12th Annual Symposium on Switching and Automata Theory (SWAT'71). IEEE Computer Society. Washington, DC, USA. 1971. 129–131.

2 Koren Y, Bell R, Volinsky C. Matrix factorization techniques for recommender systems. Computer, 2009, 42(8): 30–37.

3 Kepner J, Gilbert J. Graph algorithms in the language of linear algebra. Fundamentals of Algorithm. SIAM, 2011.

4 陈国良.并行算法的设计与分析.修订版.北京:高等教育出版社,2002.

5 Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. Communications of the ACM, 2008, 51(1): 107–113.

6 Lee KH, Lee YJ, Choi H, Chung YD, Moon B. Parallel data processing with MapReduce:a survey. SIGMOD Rec, 2012, 40(4): 11–20.

7 Sun ZG, Li T, Rishe N. Large-scale matrix factorization using MapReduce. Proc. of the 2010 IEEE International Conference on Data Mining Workshops. IEEE Computer Society. Washington, DC, USA. 2010. 1242–1248.

8 Liu C, Yang H, Fan JL, He LW, Wang YM. Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. Proc. of the 19th international conference on World wide web(WWW'10). ACM. New York, NY, USA. 2010. 681–690.

9 Gemulla R, Nijkamp E, Haas PJ, et al. Large-scale matrix factorization with distributed stochastic gradient descent. Proc. of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining(KDD'11). ACM. New York, NY, USA. 2011. 69–77.

10 Seo S, Kim J, Jin S, et al. HAMA: An efficient matrix computation with the MapReduce framework. Proc. of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science(CLOUDCOM'10). IEEE Computer Society. Washington, DC, USA. 2010. 721–726.

11 Kim S, Han H, Jung H, et al. Harnessing input redundancy in a MapReduce framework. Proc. of the 2010 ACM Symposium on Applied Computing(SAC'10). ACM. New York, NY, USA. 2010. 362–366.

12 Peng D, Dabek F. Large-scale incremental processing using distributed transactions and notifications. Proc. of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10). USENIX Association. Berkeley, CA, USA. 2010. 1–15.

13 Logothetis D, Olston C, Reed B, et al. Stateful bulk processing for incremental analytics. Proc. of the 1st ACM symposium on Cloud computing.(SoCC'10). ACM. New York, NY, USA. 2010. 51–62.

14 Murray DG, McSherry F, Isaacs R, et al. Naiad: A timely dataflow system. Proc. of the Twenty-Fourth ACM Symposium on Operating Systems Principles(SOSP '13). ACM. New York, NY, USA. 2013. 439–455.

15 Tiwari D, Solihin Y. MapReuse: Reusing computation in an in-memory MapReduce system. Proc. of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium(IPDPS'14). IEEE Computer Society. Washington, DC, USA. 2014. 61–71.

16 Bhatotia P, Wieder A, Rodrigues R, et al. Incoop: MapReduce for incremental computations. Proc. of the 2nd ACM Symposium on Cloud Computing.(SOCC'11). ACM. New York, NY, USA. 2011. 7–20.

17 Zhang YF, Chen SM, Wang Q, Yu G. i2MapReduce: Incremental MapReduce for mining evolving big data. IEEE Trans. on Knowledge & Data Engineering, 2015, 27(7): 1906–1919.

18 Venkataraman S, Roy I, AuYoung A, et al. Using R for iterative and incremental processing. Proc. of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing (HotCloud'12). USENIX Association. Berkeley, CA, USA. 2012. 14–18

19 https://hbase.apache.org.

20 http://cloud.ustc.edu.cn.