# 利用Hadoop实现超大矩阵相乘之我见（一）

## 前记

最近，公司一位挺优秀的总务离职，欢送宴上，她对我说“你是一位挺优秀的程序员”，刚说完，立马道歉说“对不起，我说你是程序员是不是侮辱你了？”我挺诧异，程序员现在是很低端，很被人瞧不起的工作吗？或许现在连卖盗版光盘的，修电脑的都称自己为搞IT的，普通人可能已经分不清搞IT的到底是做什么的了。其实我想说，程序员也分很多种的，有些只能写if-then-else，有些只能依葫芦画瓢，但真正的程序员我想肯定是某个领域的专家，或许他是一位数学家，或许他是一位物理学家，再或许他是计算机某个细分领域的专家，他是理论与现实的结合，是凌驾于纯理论的存在！而笔者我正立志成为这样的能让人感到骄傲的程序员。

切入正题吧，谈到云计算，不得不提大数据，处理大数据，肯定逃不离分布式计算。互联网行业，无论是商品推荐还是好友推荐，还是PageRank,所要处理的Items规模、用户规模都是极其庞大的，小则数以百万、千万记，大则数以亿记。在此数据基础上，诞生了很多优秀的推荐算法，推荐算法中大部分会运用到矩阵运算。如此大规模的数据，一台计算机已经没有能力处理，说简单点，一台服务器的内存可能连加载半个矩阵数据都不够，更别谈处理了。“当一头牛拉不动车时，很少有人去找一头更大更强壮的牛，而是找来更多的牛一起拉。”，这就是分布式计算，而Hadoop就是在分布式集群上处理大记录集的强大利器。

笔者最近对推荐算法挺感兴趣的，也研究了一些！部分算法数学公式研究的透彻了，便有自己想实现的冲动，可公式里的矩阵运算可不是那么简单!所以就想从研究超大规模矩阵相乘开始，一方面为以后做大规模矩阵运算、实现推荐算法做技术储备；另一方面也想真正体验一把用Hadoop实现分布式运算的乐趣；最重要的是能够写一些包含独特思想，有研究成分，有技术含量的代码。

## 摘要

本文首先讨论了目前现有的大矩阵运算方法，并指出其不足；接着提出自己的矩阵运算方法来解决目前现有方法所存在的问题，同时通过实验来观测本文方法所存在的问题，并针对这些问题，对本文方法进行再优化。

## 现有方法

- **行列相乘运算**
  - **简介**

传统的矩阵运算是A矩阵中的每一行分别与B矩阵中的每一列相乘。假设矩阵A的规模为（m*r）,矩阵B的规模为（r*n）,则矩阵C的规模为（m*n）。矩阵C中元素C**i,j**是A中第i行与B中第j列元素依次对应相乘并汇总的结果。公式表示如下：

$$A_i \cdot B_j = C_{i,j} = \sum_{k=1}^{r} A_{i,k} \cdot B_{k,j} \qquad (1)$$

每一个C**i,j**的计算都是独立的，所以可以交由不同的计算节点完成。

- **缺点**

1、矩阵规模有一定限制，如果A矩阵或B矩阵有一个超大，则某个运算节点就很有可能由于内存限制，加载不了A矩阵的第i行或B矩阵的第j列。

2、对于稀疏矩阵计算没优势。若A，B中有稀疏矩阵存在，需判断A中i行与B中第j列对应的位置上是否有0元素，换句话说，还是需要加载第i行，第j列的全部内容，若某个位置没有输入，在运算过程中需要将相应位置用0填充，这样会造成上一点所存在的问题：内存放不下。

- **矩阵分块运算**
  - **简介**

当矩阵大到一定程度时，一台服务器由于内存限制已经无法处理，不过由于矩阵具体天然的可分块的特性，许多基于分块的矩阵运算算法诞生了，**《数学之美》**这本书上介绍的大矩阵相乘方法就是基于分块的，现简单介绍如下：

1、**当A矩阵纵向很大，横向不大时**，我们将A矩阵分块，将A矩阵中的分块分别与B矩阵相乘，通过Hadoop，这些计算可以并行进行，如图1所示：

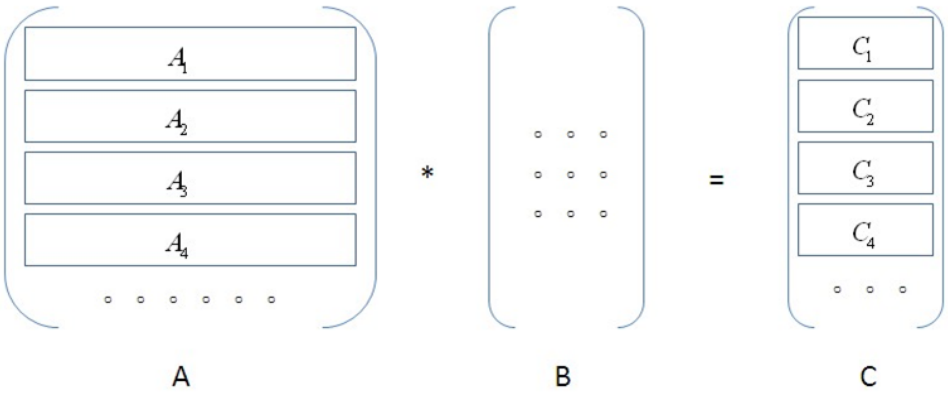


图1

图中$A_1*B=C_1$,$A_2*B=C_2$,...,每部分计算分别可在不同的计算节点完成，最后将结果组合在一起。

2、**当A矩阵为一个真正的超大矩阵（横向纵向都很大），与之相乘的B矩阵也必是一个超大矩阵（至少纵向很大）**，此时A，B矩阵都需要按行按列进行分块，并将不同的分块计算交由不同的计算节点完成，如图2所示。

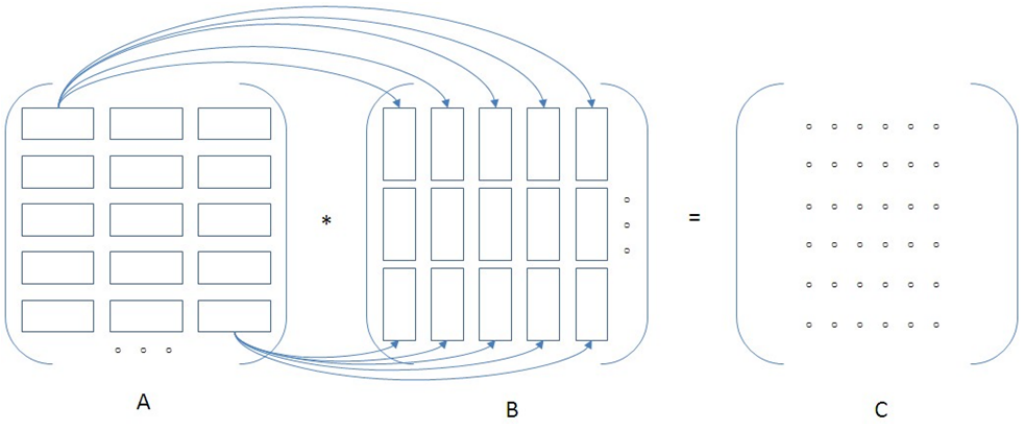


图2

图中，矩阵A中的每一块都需要和矩阵B中对应位置的块依次相乘，这些块与块之间的相乘运算可以由不同的计算节点完成，最后将不同块与块的运算结果，经过严密精确的控制，对相关结果进行合并（主要是相加），得到最终的运算结果C。

- **缺点**

1、对于不同的矩阵规模，如何分块是难点，同时块的大小受限于内存大小。

2、块与块之间的运算以及组织较繁琐。

3、不太利于稀疏矩阵的运算（0值占用较多的存储空间，以及会做很多无效运算）

- **基于最小粒度相乘的算法**

为了文档的命名结构，笔者自己根据算法原理，起了这个名字。

- **简介**

“行列相乘运算”和“分块运算”都受限于计算节点的内存限制。那么有没有一种运算，跟计算节点的内存大小无关呢？答案是：肯定有！总所周知，矩阵相乘的最小粒度计算是两个矩阵中的两个数相乘，比如，且计算结果是的一个组成部分。

假设有两个超大矩阵A和B，A的规模是（m*r）,B的规模是（r*n），将矩阵相乘中的最小粒度乘法运算进行统计，我们不难发现：A中每个元素Ai,k需要与B中第k行的元素Bk,j(j=1,2,...,n)依次相乘，计算结果分别为Ci,j的一个组成部分；而B中每个元素Bk,j需要与A中第j列的元素Ai,k(i=1,2,...,m)依次相乘，计算结果分别为$C_{i,j}$的一个组成部分。具体如图3所示。

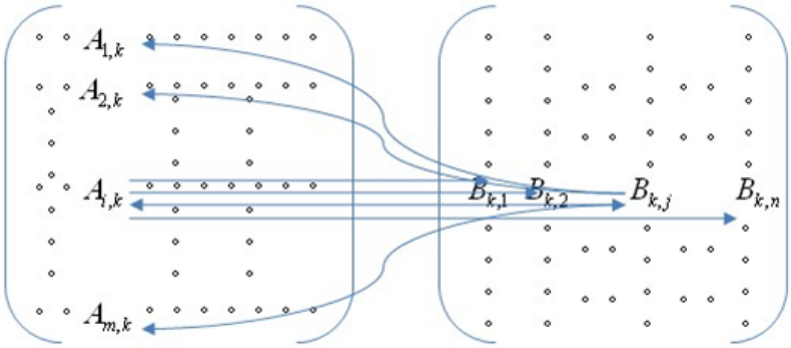


图3

由于Ai,k*Bk,j是独立的，因此可以由不同的计算节点进行运算，最后根据key （i,j）将运算结果进行汇总相加，得到结果$C_{i,j}$。同时，每个计算节点每次计算时都是只加载两个数进行相乘，并不需要加载矩阵的某个块或者某行某列，因此没有内存的限制问题，理论上只要hadoop的HDFS文件系统够大，就可以计算任意大规模的矩阵相乘。

在Map-Reduce过程中，由于Map的每条输入记录只被处理一次便不再使用，因此根据图3理论，对于矩阵A中的每个元素，在实质进行乘法运算之前，我们需要生成n个副本，对于矩阵B中每个元素，我们需要生成m个副本，并将相应位置上的副本进行对应好。比如对于Ai,k需生成n个副本，并与B中相应元素对应好，并以A中元素的行号，B中元素的列号作为key：

$$i-1 \quad A_{i,k}-B_{k,1}$$
$$i-2 \quad A_{i,k}-B_{k,2}$$
$$\cdots$$
$$i-j \quad A_{i,k}-B_{k,j}$$
$$\cdots$$
$$i-n \quad A_{i,k}-B_{k,n}$$

以以上文件作为Map输入，在Map中进行乘法运算，在Reduce阶段按key进行加法运算，就得到矩阵相乘计算结果了。

- **缺点及难点**

1、矩阵元素副本准备。

如果想以以上格式作为初始Map输入，那么我们就需要事先将数据整理成以上格式。对于两个超大矩阵相乘来说，这是一个艰巨的任务。矩阵元素一般来源于数据库（暂且如此假设，比如说做商品推荐，用户数据，商品数据都是存在数据库中的），那么整理成以上格式的文档作为Map输入文件，我们需要查询数据库的次数为：

$$m*r*n + r*n*m$$

由于m,r,n都是极其庞大的，这个查询次数是我们万万不能忍受的。理想的数据库查询次数是：

$$m*r + r*n$$

即矩阵元素只取一次。

还有一种方法是矩阵元素只取一次，每个元素的副本生成交给Map-Reduce去做，但是这样存在另一个问题：如果在Map-Reduce过程中将A矩阵和B矩阵中的元素进行副本拷贝，单个Map的运算时间有点让人接受不了，打个比方，一个Map块为64M，大约存了500万条A矩阵的元素，同时B矩阵的n为10亿，那么计算这个Map的节点需生成500万*10亿条副本，这个时间是难以忍受的。

2、两个矩阵中需相乘的元素如何对应。

由于利用数据库查询进行矩阵元素对对应时间复杂度太高，一般不太可行。所以可以考虑利用Map-Reduce对相应元素进行对应。不过Map只对输入记录进行一次处理，处理完毕便结束，不存在内存的概念，所以对两个矩阵中元素进行对应是一个难点。

3、文件大小规模。

对于超大规模矩阵，由于A中m和B中n太大，除去稀疏元素（值为0）不纳入计算，需拷贝的元素依旧很多，拷贝完的文件大小是极其庞大的。笔者做了个实验，将A（1000*1000） B（1000*1000）两个稠密矩阵中的元素按规定（A中每个元素拷贝1000份，B中每个元素拷贝1000份）进行副本拷贝，拷贝完的记录数为$2*10^9$条，文件大小达到24G。那么对于亿万规模的矩阵，文件大小将成指数级增长。

**本文方法**

“行列相乘运算”对于稀疏矩阵可以，但是对于大型的稠密矩阵显得有点力不从心；而对于“分块矩阵运算”很多学者做了很多研究，但是笔者不太喜欢该算法，第一是逻辑控制麻烦，第二是对块的大小优化来优化去，没有解决本质上的问题。笔者我喜欢简单的东西，所以更倾向于“基于最小粒度相乘的算法”。不过，就像我们之前所说的，“基于最小粒度相乘的运算”存在三个问题，接下来，笔者将针对其中的两个问题阐述笔者自己的想法。

- **新颖的矩阵相乘元素映射方法**
  - **简介**

矩阵A*B=C中，$C_{i,j}$是A中第i行与B中第j列相乘的结果,如公式（1）所示。通俗点可以写成如下格式：

$$\mathbf{C_{i,j} = A_{i,1} \cdot B_{1,j} + A_{i,2} \cdot B_{2,j} + \cdots + A_{i,k} \cdot B_{k,j} + \cdots + A_{i,r} \cdot B_{r,j} \quad (2)}$$

传统的方法在Map输入段通常将输入记录组织成如下格式：

$$i - j \;\; A_{i,1} - B_{1,j}$$

$$i - j \;\; A_{i,2} - B_{2,j}$$

$$\cdots$$

$$i - j \;\; A_{i,k} - B_{k,j}$$

$$\cdots$$

$$i - j \;\; A_{i,r} - B_{r,j}$$

然后在Map端进行各条记录的相乘运算，最后在Reduce阶段进行汇总，得出最终矩阵相乘的结果。不过正像“基于最小粒度相乘的算法”中所说的，由于key i–j 不具备明显的区分度，且Map过程中，内存不保留矩阵元素，将数据组织成以上格式是极其困难的。如果在Map输入前，将数据组织成以上格式，查询数据库的时间复杂度也是难以接受的。

通过思考我们不难发现，最终结果$C_{i,j}$是由r个值相加而成的，第k个组成成分为：Ai,k–Bk,j，为了使key更有区分度，我们将key修改为：

$$i - j - k \;\; A_{i,k}$$

$$i - j - k \;\; B_{k,j}$$

这样的key所代表的两个值相乘，得到了$C_{i,j}$中第k个组成元素。所以对于A矩阵和B矩阵在Map阶段完成数据副本拷贝完后，所有的Map数据记录中，i–j–k的key有且至多只有两个（由于稀疏元素不纳入计算与拷贝，所以若为一个，则说明与之相乘的另一个元素为0，若一个也没有，则说明Ai,k与Bk,j都为0，没有纳入计算与拷贝）。

由于A中每个元素理论上都需要被计算n遍，所以可以将A中元素按如下规则进行n遍拷贝，对于Ai,k，拷贝方式如下：

$$\mathbf{i - j - k \;\; A_{i,k} \;\; j \in (1, 2, \ldots, k, \ldots, n)} \qquad \mathbf{(3)}$$

对于B中每个元素，理论上每个元素都需要被计算m遍，所以可以将B中元素按如下规则进行m遍拷贝，对于Bk,j，拷贝方式如下：

$$i-j-k\ B_{k,j}\ \ i \in (1,2,\dots,k,\dots,m) \qquad (4)$$

由于每个元素的副本拷贝都是独立的，所以可以由不同的Map进行，大大加快了拷贝速度。

- **实验结果**

笔者用以上方法做了实验，A(m,r)*B(r,n)=C，其中m=r=n=1000,所以两个矩阵中共有$2*10^6$个元素，A与B都为稠密矩阵，以“A–i–k value”和“B–k–j value”的形式存储原始的A矩阵和B矩阵的元素，文件大小为24M。由于文件太小，所以只交给一个Map进行副本的拷贝工作，每个元素都被拷贝一千遍，拷贝完总记录数为$2*10^9$条。消耗时间如下：

| Kind | Total Tasks(successful+failed+killed) | Successful tasks | Failed tasks | Killed tasks | Start Time | Finish Time |
|---|---|---|---|---|---|---|
| Setup | 1 | 1 | 0 | 0 | 10-六月-2013 01:54:34 | 10-六月-2013 01:54:39 (4sec) |
| Map | 1 | 1 | 0 | 0 | 10-六月-2013 01:54:40 | 10-六月-2013 06:54:56 (5hrs, 16sec) |

| Map input records | 2,000,000 | 0 | 2,000,000 |
|---|---|---|---|

| Map output records | 2,000,000,000 | 0 | 2,000,000,000 |
|---|---|---|---|

图4

由图4可以看出，一个Map执行的时间非常长，这是因为Map中的每条记录都需要拷贝1000遍。如果在现实应用中，两个矩阵超大，那么许多Map的块大小都将被填满，一个块大概放500万条记录，同时由于每条记录都被拷贝m或n遍（m,n很可能就是数以亿计），那么一个Map的执行时间就是无底洞了。

为了减少每个Map的执行时间，笔者苦苦冥想，终于想出一种方法，将在接下来的小节进行介绍。

- **创新的细胞分裂拷贝算法**

上一小节中有讲到Map的执行时间过长，有同事建议我说将Map的块变小点，这样里面的记录数也少点，不同的块由不同的节点执行。但是笔者认为这种想法不合理，一个块是变小了，里面的记录也小了，但是若每条记录需拷贝的数量是庞大的，那根本于事无补。而且对于不同大小的矩阵相乘，矩阵元素需拷贝的数量也都是不一样的，因此块的大小很难控制。再者，对于Hadoop运算，正常情况下都是加大Map块的大小，这样有利于计算的集中。

而在本方法中，Map拷贝过程之所以时间太长，笔者认为是由于每条记录拷贝的数量太多造成的，如果一条记录的拷贝能分段在不同的节点完成就好了，出于这样的想法，笔者设计了一种利用Map迭代进行拷贝的方法，由于迭代过程中Map数量的扩张有点像细胞分裂，笔者称之为“细胞分裂拷贝算法”。

- **简介**

由于每个矩阵元素需要拷贝多少遍是确定的，因此我们可以设计一种分段拷贝方法来让不同的节点进行拷贝工作。这里有两个变量需要介绍，一是num_split,代表一条记录在一次迭代过程中最多被分的段数，另一个是num_copy,代表每个最终分段最多需拷贝的记录数。在迭代过程中，如果某条记录的某个分段范围大于num_copy,则继续进行分段，否则就进行拷贝工作。现举例说明迭代过程。

对于A中元素Ai,k，其需要被拷贝1000遍，为了将其拷贝成公式（3）所示，我们利用细胞分裂拷贝算法将拷贝工作分配到不同的计算节点进行，该例中num_split和num_copy的数值都为10，那么迭代过程如下：

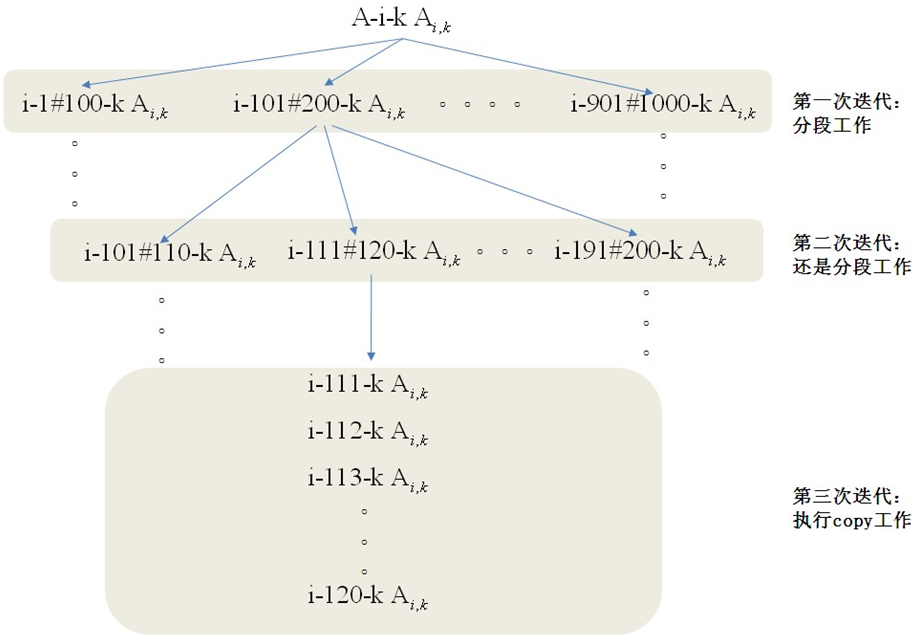


图5

而对于B中元素，我们同理利用迭代拷贝的方法将其拷贝成公式（4）所示格式，即主要的范围辨别集中在i上。

由图5可以可以看到，每一次迭代，数据记录都是成num_split的倍数增长，这样，随着记录集文件大小的增长，文件被分成越来越多的Map，自然也就被分配到越来越多的计算节点进行执行。查看图5中的第三次迭代工作，由于记录范围符合记录生成条件，即记录范围<=num_copy，第三次迭代过程中，每个Map上的每条记录只被copy了num_copy遍，相较于之前每条记录被copy1000遍，时间大大减少，这种方法对于规模大的矩阵尤其适用。

此外值得一提的是，由于现实中矩阵A和矩阵B的规模往往不一样，在实现“细胞分裂拷贝算法”时，需要设置两个标志变量来判断不同矩阵的记录分段迭代过程是否结束，若两个矩阵的分段迭代过程都结束了，则进入最后一次迭代过程：记录拷贝的生成。

- **实验结果**

笔者对，A(m,r)*B(r,n)=C，其中m=r=n=1000做实验，共经历三次迭代完成矩阵元素的拷贝工作，如图6所示，第一次迭代，输入只有24M，所以只有一个Map，输出了3个Map，第二次迭代，由于输入的3个Map，输出了30个Map，符合num_split的扩张倍数，第三次迭代的工作是执行拷贝工作。

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| job_201306140105_0006 | NORMAL | hadoop | Matrix Multiplication--Step3 | SUCCEEDED | Fri Jun 14 18:18:42 CST 2013 | Fri Jun 14 18:58:39 CST 2013 | 100.00% | 100.00% | NA | NA |
| job_201306140105_0005 | NORMAL | hadoop | Matrix Multiplication--Step2 | SUCCEEDED | Fri Jun 14 18:12:54 CST 2013 | Fri Jun 14 18:18:42 CST 2013 | 100.00% | 100.00% | NA | NA |
| job_201306140105_0004 | NORMAL | hadoop | Matrix Multiplication--Step1 | SUCCEEDED | Fri Jun 14 18:11:42 CST 2013 | Fri Jun 14 18:12:53 CST 2013 | 100.00% | 100.00% | NA | NA |

| | | | | | | |
|---|---|---|---|---|---|---|
| Map | 1 | 1 | 0 | 0 | 14-六月-2013 18:11:53 | 14-六月-2013 18:12:46 (53sec) |
| Map | 3 | 3 | 0 | 0 | 14-六月-2013 18:13:05 | 14-六月-2013 18:18:33 (5mins, 28sec) |
| Map | 30 | 30 | 0 | 0 | 14-六月-2013 18:18:50 | 14-六月-2013 18:58:28 (39mins, 37sec) |

图6

同时，我们可以看到，在最后生成拷贝的过程中，每个Map的执行时间比较稳定，如图7所示，这样，当我们的集群够大时，这30个Map在一轮过程中便可以执行完毕。

| Task Id | Start Time | Finish Time | Error |
| --- | --- | --- | --- |
| task_201306140105_0006_m_000000 | 14/06 18:18:51 | 14/06 18:21:31 (2mins, 39sec) | |
| task_201306140105_0006_m_000001 | 14/06 18:21:31 | 14/06 18:24:10 (2mins, 38sec) | |
| task_201306140105_0006_m_000002 | 14/06 18:24:10 | 14/06 18:26:44 (2mins, 34sec) | |
| task_201306140105_0006_m_000003 | 14/06 18:26:44 | 14/06 18:29:14 (2mins, 29sec) | |
| task_201306140105_0006_m_000004 | 14/06 18:29:14 | 14/06 18:31:46 (2mins, 31sec) | |
| task_201306140105_0006_m_000005 | 14/06 18:31:46 | 14/06 18:34:18 (2mins, 31sec) | |

图7

最后，当矩阵元素拷贝工作与对应工作完成后，接下来就比较简单了，再经历两轮Map-Reduce过程，就可以得到运算结果了。

**总结**

本文方法针对“基于最小粒度相乘的算法”中所固有的缺点及难点，利用巧妙的设计，有效的利用Map-Reduce工具进行相乘元素的对应，同时为了减少单个元素在一个节点上拷贝太多记录所造成的时间损耗，设计了“细胞分裂拷贝算法”，有效的将同一条记录的拷贝工作分发到不同的节点进行，大大缩短了一个节点的执行时间，同时充分利用和发挥了集群运算的优势。

但是本文由于算法的固有特性，并没有解决“基于最小粒度相乘的算法”中最后一个缺点:文件占用空间太大。理论上说，这个缺点对于HDFS系统是不算一个缺点的，本身HDFS系统就有足够大的空间容纳足够的数据。但是，通过实验发现，这个文件实在是庞大的，对于（1000,1000）与（1000,1000）两个稠密矩阵的相乘，所有元素的拷贝工作完成后，记录数目达到$2*10^9$条，占用空间大小为二三十G，如图8所示。那么对于更大规模的矩阵运算，文件空间要占用多大？答案是：难以估量。

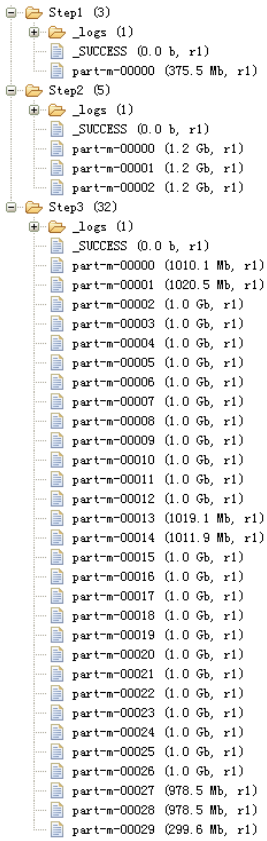


图8

**后记**

大部分算法都有其优势与局限性，针对本文方法本质所固有的文件存储空间占用大这个问题，笔者一直是耿耿于怀，至少这样的算法是不完美的，虽然它解决了一些问题。连续几天笔者是冥思又苦想，连做梦时脑海里都是两个矩阵元素在打架了！黄天不负有心人，灵光一现，一种新的方法在笔者脑海里浮现！尽请期待下期《利用Hadoop实现超大矩阵相乘之我见（二）》，在下期中，笔者会分析本文方法本质上造成文件占用空间大的原因，同时介绍笔者新想到的自认为还比较完美的方法。新方法非常适用于大规模稠密矩阵与稀疏矩阵的相乘计算，尤其是对于稀疏矩阵，基本上没有无效计算，也不会造成多余空间的浪费。