MapReduce综述

暨南大学信息科技学院07级研究生 周敏

January 3, 2008

摘 要

MapReduce是由Google公司发明，近些年新兴的分布式计算模型。作为Google公司的核心技术，MapReduce在处理T级别以上巨量数据的业务上有着明显的优势。 本文从分布式计算的历史背景开始，论述了MapReduce的灵感源泉及基本原理。

Google公司的MapReduce实现是该公司的保密技术，而来自开源社区Apache的Hadoop项目则是此思想的java克隆版本。 最近几个月里，Stanford大学提出的Phoenix项目将MapReduce应用于共享存储结构的硬件平台上，取得了一定的成果。本文将重点论述，此三者在实现上的关键技术区别。

目 录

1 分布式计算概述

1.1 摩尔定律和免费的午餐

“集成电路芯片上所集成的电路的数目，每隔18个月就翻一番”——这就是著名的摩尔定律。 这条业界普遍承认的定律从1965年问世起已近40多年了， CPU的性能在这几十年内得到了呈指数型地提高， 特别是在2000年至2003年，CPU的频率得到了非常大的发展，顺利地进入了3G时代。 然而从2004年开始,这种发展趋势戛然而止。 主流PC芯版制造商如AMD、Intel、Sparc和PowerPc等公司全部都先后停止了其高频芯片的研发计划， 转而向超线程和多核架构发展。 这标志着尽管摩尔定律预言了过去几十年的发展速度， 但我们应该明白会一天这种发展将会终止， 因为硬件毕竟受物理极限的约束，这一天已经来临了。 c++标准委员会主席Herb Sutter在2005年的时候， 发表了著名的《免费午餐已经结束——软件历史性地向并发靠拢》这一著名文章， 他指出软件在多核时代生存，必须考虑并问题。 接着Herb Sutter做出了更大胆和惊人的预言， 那就是不存在解决多核编程问题的银弹， 不存在可以简单地将并发编程问题化解掉的工具， 开发高性能的并行程序必须要求开发者从根本上改变其编程方

法。 从某种意义上来说，这不仅仅是要改变50年来顺序程序设计的工艺传统， 而且是要改变数百万年来人类顺序化思考问题的习惯。

将近三年后的今天，Herb Sutter的预言初步得到了印证。 单核机器现在已经慢慢过时，而多核机器的价格正在迅速下降。 人们正在紧张地寻找解决多核编程难题的方法。 最终目的是为了提高计算性能，不能让CPU成为性能的颈瓶。 Google公司的MapReduce模型就是在多台普通机器，利用函数式的思想，提高程序执行的性能， 而Stanford的Phoenix则是这一模型在多核时代的解决方案。

1.2 串行与并行编程

在早期的计算里，程序一般是被串行执行的。因为程序是指令的序列，在单处理器的机器里，程序从开始到结束，这些指令是一条接一条执行的。

并行编程的产生是为了提高程序执行的性能和效率。在一个并行的程序里，一道处理可以被划分为几部分,然后它们可以并发地执行。 各部分的指令分别在不同的CPU上同时运行。这些CPU可以存在于单台机器中,也可以存在于多台机器上,它们通过连接起来共同运作。

并行程序不但可以执行得更快，而且可以用来解决非存储在非本地的超大数据集上的问题。 当你拥有一组已经连网的机器时，你也就拥有了一组可计算的CPU，并且你还将拥有读写超大文件集的能力(假设已经配好分布式文件系统)。

1.3 并行基本概念

建立并行程序的第一步是区分哪些任务可以同时做，或者哪些部分的数据可以同时处理。有时候这是不可能的，比如Fibonacci函数:

$$F_k = \begin{cases} 1, & k = 0 \\ 1, & k = 1 \\ F_{k-1} + F_{k-2}, & k \geq 2. \end{cases}$$

此函数的值是靠上一层函数的值得出来的，是不可能被并行化的，因为每个得出来的值都依赖与上一个值。

有一个普遍存在的现象，就是我们拥有大量结构一致的数据要处理。如果数据可以分解成相同大小的部分， 那我们就可以设法使这道处理变成并行的了。

比如一个庞大的数组，我们可以把它分解成一些大小相同的子 数组。如果对每个数组元素的处理都是一样的，不存在依赖计算，而且各个处理

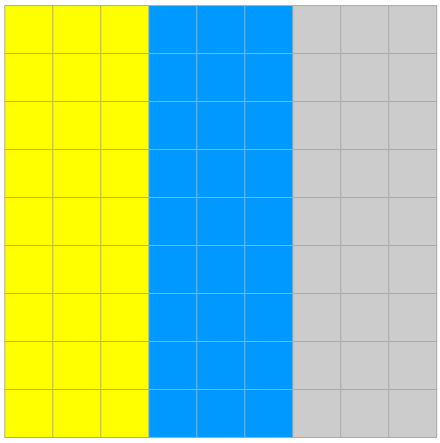

图 1: 数组元素处理的并行化

任务之间不存在通信， 那就表示我们找到一个理想的并行化机会了。下面是我们解决这一问题常用的master/worker模型:
master:

- 初始化数组，同时按照worker的个数将数组分解
- 将分解后的子数组发送给worker
- 从各个worker收到结果

worker:

- 接收来自master的子数组
- 处理子数组
- 将结果发送给master

此模型
我们再举个master/worker技术的例子，计算π的值。第一步是画出一个正方形中内嵌一个圆。 正方形的面积为$A_s = (2r)^2 = 4r^2$; 圆的面积

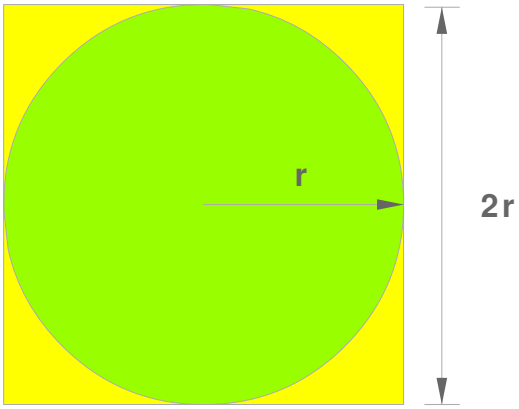

图 2: 计算π的值

为$A_c = \pi r^2$。所以：

$$\pi = \frac{A_c}{r^2},$$
$$\text{又} A_s = 4r^2 \Longrightarrow r^2 = \frac{A_s}{4},$$
$$\Longrightarrow \pi = \frac{A_c}{\frac{A_s}{4}} = \frac{4A_c}{A_s}$$

我们做这个代数推导的原因是我们可以按如下方法让计算变得可并行化：

1. 在正方形内随机地生成一些点
2. 计算这些点在圆形内的个数
3. $p = \frac{\text{圆形内的点的个数}}{\text{正方形内的点的个数}}$
4. $\pi = 4p$

从而我们可以用下面的程序实现并行：

```
NUMPOINTS = 100000; // 数越大，越近似
n = worker的数目;
numPerWorker = NUMPOINTS / n;

worker:
```

```
        countCircle = 0; // 每个worker都有一个计数器
        // 每个worker都像执行以下程序:
        for (i = 0; i < numPerWorker; i++) {
                产生两个在正方形内的随机数;
                xcoord = 第一个随机数;
                ycoord = 第二个随机数;
                如果(xcoord, ycoord)在圆形中
                countCircle++;
        }

master:
        接收各个worker传来的countCircle的值;
        计算π: pi = 4.0 * countCircle / NUMPOINTS;
```

这就是著名的蒙特卡罗概率算法求π值的并行实现。

2 MapReduce基本原理介绍

在过去的数年里，Google的许多员工已经实现了数以百计的为专门目的而写的计算，用来处理大量的原始数据。比如爬虫文档、Web请求日志等等。 为了计算各种类型的派生数据，比如倒排索引，Web文档的图结构的各种表示，每个主机上爬行的页面数量的概要，每天被请求数量最多的集合，等等。 很多这样的计算在概念上很容易理解。 然而，输入的数据量很大，并且只有计算被分布在成百上千的机器上才能在可以接受的时间内完成。 怎样并行计算，分发数据，处理错误，所有这些问题综合在一起，使得原本很简介的计算，因为要大量的复杂代码来处理这些问题，而变得让人难以处理。 作为对这个复杂性的回应，Google公司的Jeffery Dean设计一个新的抽象模型， 使我们只要执行的简单计算，而隐藏并行化、容错、数据分布、负载均衡的那些杂乱细节则放在一个库里，不必关心它们。 此类抽象模型的灵感来自Lisp和许多其他函数语言的map和reduce的原始表示。 事实上许多计算都包含这样的操作： 在我们输入数据的逻辑记录上应用map操作，来计算出一个中间key/value对集； 在所有具有相同key的value上应用reduce操作，来适当地合并派生的数据。 功能模型的使用，再结合用户指定的map和reduce操作，让我们可以非常容易的实现大规模并行化计算， 同时使用重启作为初级机制可以很容易地实现容错。这个工作的主要贡献是通过简单有力的接口来实现自动的并行化和大规模分布式计算， 结合这个接口的实现在大量普通的PC机上实现高性能计

算。

计算利用一个输入key/value对集，来产生一个输出key/value对集。MapReduce库的用户用两个函数表达这个计算：map和reduce。 用户自定义的map函数，接受一个输入对，然后产生一个中间key/value对集。MapReduce库把所有具有相同中间key I的中间value聚合在一起，然后把它们传递给reduce函数。 用户自定义的reduce函数，接受一个中间key I和相关的一个value集。 它合并这些value，形成一个比较小的value集。一般的，每次reduce调用只产生0或1个输出value。 通过一个迭代器把中间value提供给用户自定义的reduce函数。 这样可以使我们根据内存来控制value列表的大小。

2.1 计算单词数WordCount

考虑这个问题：计算在一个大的文档集合中每个词出现的次数。用户将写和下面类似的伪代码：

```
map(String key, String value):
 //key: 文档的名字
 //value: 文档的内容
 for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
//key: 一个词
//values: 一个计数列表
 int result=0;
 for each v in values:
   result+=ParseInt(v);
 Emit(AsString(resut));
```

map函数产生每个词和这个词的出现次数(在这个简单的例子里就是1)。 reduce函数把产生的每一个特定的词的计数加在一起。 另外，用户用输入输出文件的名字和可选的调节参数来填充一个mapreduce规范对象。 用户然后调用MapReduce函数，并把规范对象传递给它。 用户的代码和MapReduce库链接在一起。

2.2 类型

即使前面的伪代码写成了字符串输入和输出的term格式，但是概念上用户写的map和reduce函数有关联的类型：

```
map(k1, v1) ->list(k2, v2)
reduce(k2, list(v2)) ->list(v2)
```

例如输入的key，value和输出的key，value的域不同。此外，中间key，value和输出key，values的域相同。我们的C++实现传递字符串来和用户自定义的函数交互，并把它留给用户的代码，来在字符串和适当的类型间进行转换。

2.3 其它实例

这里有一些让人感兴趣的简单程序，可以容易的用MapReduce计算来表示。

分布式的Grep(UNIX工具程序，可做文件内的字符串查找)：如果输入行匹配给定的样式，map函数就输出这一行。reduce函数就是把中间数据复制到输出。

计算URL访问频率：map函数处理web页面请求的记录，输出(URL，1)。reduce函数把相同URL的value都加起来，产生一个(URL，记录总数)的对。

倒转网络链接图：map函数为每个链接输出(目标，源)对，一个URL叫做目标，包含这个URL的页面叫做源。reduce函数根据给定的相关目标URLs连接所有的源URLs形成一个列表，产生(目标，源列表)对。

每个主机的术语向量：一个术语向量用一个(词，频率)列表来概述出现在一个文档或一个文档集中的最重要的一些词。map函数为每一个输入文档产生一个(主机名，术语向量)对(主机名来自文档的URL)。reduce函数接收给定主机的所有文档的术语向量。它把这些术语向量加在一起，丢弃低频的术语，然后产生一个最终的(主机名，术语向量)对。

倒排索引：map函数分析每个文档，然后产生一个(词，文档号)对的序列。reduce函数接受一个给定词的所有对，排序相应的文档IDs，并且产生一个(词，文档ID列表)对。所有的输出对集形成一个简单的倒排索引。它可以简单的增加跟踪词位置的计算。

分布式排序：map函数从每个记录提取key，并且产生一个(key，record)对。reduce函数不改变任何的对。

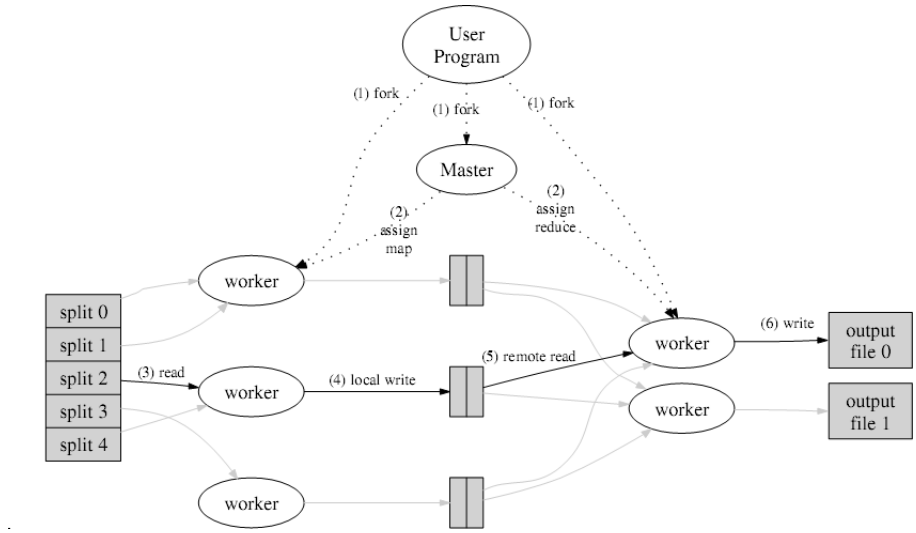

图 3: Google的MapReduce执行方式

3 MapReduce实现

3.1 Google的MapReduce实现

3.1.1 执行方式

Map调用通过把输入数据自动分割成M片被分布到多台机器上，输入的片能够在不同的机器上被并行处理。 Reduce调用则通过分割函数分割中间key，从而形成R片(例如，hash(key) mod R)，它们也会被分布到多台机器上。 分割数量R和分割函数由用户来指定。

图3显示了Google实现的MapReduce操作的全部流程。 当用户的程序调用MapReduce函数的时候，将发生下面一系列动作(下面的数字和图3中的数字标签相对应)：

1. 在用户程序里的MapReduce库首先分割输入文件成M个片，每个片的大小一般从16到64MB(用户可以通过可选的参数来控制)。 然后在机群中开始大量地拷贝程序。

2. 这些程序拷贝中的一个是master，其他的都是由master分配任务的worker。 有M个map任务和R个reduce任务将被分配。master分配一个map任务或reduce任务给一个空闲的worker。

3. 一个被分配了map任务的worker读取相关输入split的内容。 它从输入数据中分析出key/value对，然后把key/value对传递给用户自定义的map函数。 由map函数产生的中间key/value对被缓存在内存中。

4. 缓存在内存中的key/value对被周期性的写入到本地磁盘上，通过分割函数把它们写入R个区域。 在本地磁盘上的缓存对的位置被传送给master，master负责把这些位置传送给reduce worker。

5. 当一个reduce worker得到master的位置通知的时候， 它使用远程过程调用来从map worker的磁盘上读取缓存的数据。 当reduce worker读取了所有的中间数据后，它通过排序使具有相同key的内容聚合在一起。 因为许多不同的key映射到相同的reduce任务，所以排序是必须的。 如果中间数据比内存还大，那么还需要一个外部排序。

6. reduce worker迭代排过序的中间数据，对于遇到的每一个唯一的中间key， 它把key和相关的中间value集传递给用户自定义的reduce函数。 reduce函数的输出被添加到这个reduce分割的最终的输出文件中。

7. 当所有的map和reduce任务都完成了，master唤醒用户程序。 在这个时候，在用户程序里的MapReduce调用返回到用户代码。

在成功完成之后，mapreduce执行的输出存放在R个输出文件中(每一个reduce任务产生一个由用户指定名字的文件)。 一般，用户不需要合并这R个输出文件成一个文件–他们经常把这些文件当作一个输入传递给其他的MapReduce调用， 或者在可以处理多个分割文件的分布式应用中使用他们。

3.1.2 Master的数据结构

master保持一些数据结构。 它为每一个map和reduce任务存储它们的状态(空闲，工作中，完成)，和worker机器(非空闲任务的机器)的标识。 master 就像一个管道，通过它，中间文件区域的位置从map任务传递到reduce任务。 因此，对于每个完成的map任务，master存储由map任务产生的R 个中间文件区域的大小和位置。 当map任务完成的时候，位置和大小的更新信息被接受。这些信息被逐步增加的传递给那些正在工作的reduce任务。

3.1.3 容错

因为MapReduce库被设计用来使用成百上千的机器来帮助处理非常大规模的数据，所以这个库必须要能很好的处理机器故障。

worker故障

master 周期性的ping每个worker。 如果master在一个确定的时间段内没有收到worker返回的信息，那么它将把这个worker标记成失效。 因为每一个由这个失效的worker完成的map任务被重新设置成它初始的空闲状态，所以它可以被安排给其他的worker。 同样的，每一个在失败的 worker上正在运行的map或reduce任务，也被重新设置成空闲状态，并且将被重新调度。 在一个失败机器上已经完成的map任务将被再次执行，因为它的输出存储在它的磁盘上，所以不可访问。 已经完成的reduce任务将不会再次执行，因为它的输出存储在全局文件系统中。 当一个map任务首先被worker A执行之后，又被B执行了(因为A失效了)， 重新执行这个情况被通知给所有执行reduce任务的worker。 任何还没有从A读数据的reduce任务将从worker B读取数据。 MapReduce可以处理大规模worker失败的情况。 例如，在一个MapReduce操作期间， 在正在运行的机群上进行网络维护引起80台机器在几分钟内不可访问了， MapReduce master只是简单的再次执行已经被不可访问的worker完成的工作， 继续执行，最终完成这个MapReduce操作。

master故障

可以很容易地让master周期地写入上面描述的数据结构的检查点。 如果这个master任务失效了，可以从上次最后一个检查点开始启动另一个master进程。 然而，因为只有一个master，所以它的失败是比较麻烦的。 因此我们现在的实现是，如果master失败，就中止MapReduce计算。客户可以检查这个状态，并且可以根据需要重新执行MapReduce操作。

在错误面前的处理机制

当用户提供的map和reduce操作对它的输出值是确定的函数时， 我们的分布式实现产生，和全部程序没有错误的顺序执行一样，相同的输出。我们依赖对map和reduce任务的输出进行原子提交来完成这个性质。 每个工作中的任务把它的输出写到私有临时文件中。 一个reduce任务产生一个这样的文件，而一个map任务产生R个这样的文件(一个reduce任务对应一个文件)。 当一个map任务完成的时候，worker发送一个消息给 master，在这个消息中包含这R个临时文件的名字。 如果master从一个已经完成的map任务再次收到一个完成的消息，它将忽略这个消息。 否则，它在master的数据结构里记录这R个文件的名字。 当一个reduce任务完成的时候，这个reduce worker原子的把临时文件重命名成最终的输出文件。 如果相同的reduce任务在多个机器上执行，多个重命名调用将被执

行，并产生相同的输出文件。 我们依赖由底层文件系统提供的原子重命名操作来保证，最终的文件系统状态仅仅包含一个reduce任务产生的数据。 我们的map和reduce操作大部分都是确定的，并且我们的处理机制等价于一个顺序的执行的这个事实， 使得程序员可以很容易的理解程序的行为。 当map或reduce操作是不确定的时候，我们提供虽然比较弱但是合理的处理机制。 当在一个非确定操作的前面，一个reduce任务R1的输出等价于一个非确定顺序程序执行产生的输出。 然而，一个不同的reduce任务R2的输出也许符合一个不同的非确定顺序程序执行产生的输出。 考虑map任务M和reduce任务R1，R2的情况。 我们设定e(Ri)为已经提交的Ri的执行(有且仅有一个这样的执行)。 这个比较弱的语义出现，因为e(R1)也许已经读取了由M的执行产生的输出，而e(R2)也许已经读取了由M的不同执行产生的输出。

3.1.4 存储位置

在我们的计算机环境里，网络带宽是一个相当缺乏的资源。我们利用把输入数据(由GFS管理)存储在机器的本地磁盘上来保存网络带宽。 GFS把每个文件分成64MB的一些块，然后每个块的几个拷贝存储在不同的机器上(一般是3个拷贝)。 MapReduce的master考虑输入文件的位置信息，并且努力在一个包含相关输入数据的机器上安排一个map任务。 如果这样做失败了，它尝试在那个任务的输入数据的附近安排一个map任务(例如，分配到一个和包含输入数据块在一个switch里的worker机器上执行)。 当运行巨大的MapReduce操作在一个机群中的一部分机器上的时候，大部分输入数据在本地被读取，从而不消耗网络带宽。

3.1.5 任务粒度

象上面描述的那样，我们细分map阶段成M片，reduce阶段成R片。 M和R应当比worker机器的数量大许多。每个worker执行许多不同的工作来提高动态负载均衡， 也可以加速从一个worker失效中的恢复，这台机器上的许多已经完成的map任务可以被分配到所有其他的worker机器上。 在我们的实现里，M和R的范围是有大小限制的，因为master必须做O(M+R)次调度，并且保存O(M*R)个状态在内存中。 (此因素使用的内存是很少的，在O(M*R)个状态片里，大约每个map任务/reduce任务对使用一个字节的数据)。 此外，R经常被用户限制，因为每一个reduce任务最终都是一个独立的输出文件。 实际上，我们倾向于选择M，以便每一个单独的任务大概都是16到64MB 的输入数据(以便上面描述的位置优化是最有效的)， 我们把R设置成我们希望使用的worker机器数量的小倍

数。 我们经常在M=200000，R=5000，使用2000台worker机器的情况下，执行MapReduce计算。

3.1.6 备用任务

一个落后者是延长MapReduce操作时间的原因之一： 一个机器花费一个异乎寻常地的长时间来完成最后的一些map或reduce任务中的一个。 有很多原因可能产生落后者。 例如，一个有坏磁盘的机器经常发生可以纠正的错误，这样就使读性能从30MB/s降低到3MB/s。 机群调度系统也许已经安排其他的任务在这个机器上， 由于计算要使用CPU、内存、本地磁盘、网络带宽的原因，引起它执行MapReduce代码很慢。 我们最近遇到的问题是，在机器初始化时的Bug引起处理器缓存的失效：在一台被影响的机器上的计算性能有上百倍的影响。 我们有一个一般的机制来减轻这个落后者的问题。 当一个MapReduce操作将要完成的时候，master调度备用进程来执行那些剩下的还在执行的任务。 无论是原来的还是备用的执行完成了，工作都被标记成完成。 我们已经调整了这个机制，通常只会占用多几个百分点的机器资源。 我们发现这可以显著地减少完成大规模MapReduce操作的时间。

3.2 Apache的Hadoop项目

3.3 Stanford的Phoenix系统

Phoenix[12]是在共享内存的体系结构上的MapReduce实现。 它的目标是在多核平台上，使程序执行得更高效，而且使程序员不必关心并发的管理。 事实上并发管理，尽管是经验丰富的程序员，也难免在这上面出错。Phoenix由一组对程序应用开发者开放的简单API和一个高效的运行时组成。 运行时系统处理程序的并发、资源管理和错误修复， 它的实现是建立在P-thread之上的，当然也可以很方便地移植到其它的共享内存线程库上。

3.3.1 Phoenix API

目前的Phoenix为应用程序员实现提供了向C和C++的接口(API)。 类似的API也提供给Java或C#，这项工程Stanford大学正在实现中。 API包括了两部分函数集。基础部分提供了指向输入/输出数据缓冲和指向用户自定义函数的指针。 在设置这些指针之关，用户得先调用phoenix_scheduler()函数。 剩下的部分是用户可选的，它可以让用户在运行时控制进程调度的决

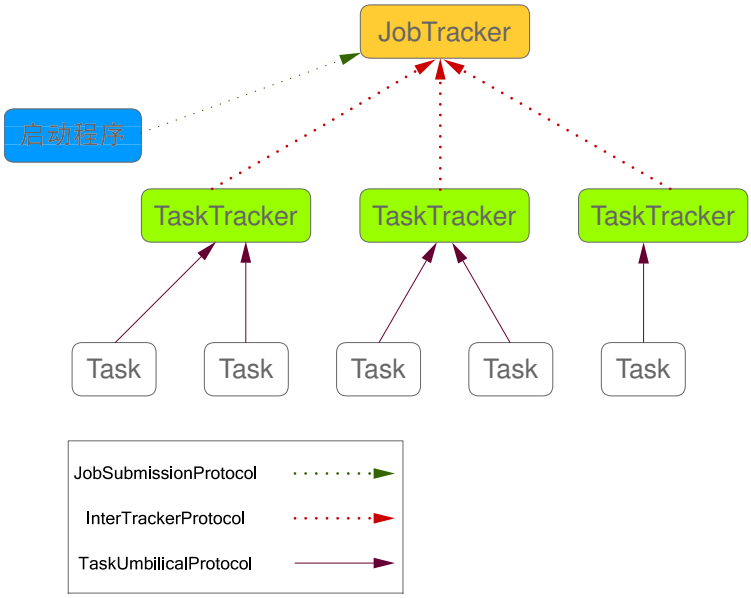

图 4: hadoop进程运行图

策。 我们将在3.3.5节讲解这部分。 在Splitter、Map、Partion和Reduce函数之间存在一些额外的数据结构，用来协助这四者之间的通信。

3.3.2 基本操作和控制流程

图5表示了Phoenix运行时的基本流程。 运行时是由调度器控制的，而调度器是由用户代码初始化的。 调度器的职责是建立和管理所有运行Map和Reduce任务的线程。 它也管理用来任务之间通信的缓冲区。程序员通过scheduler_args结构体来初始化调度器需要的所有数据和函数指针。 初始化完毕后，调度器检测可以用来计算的处理器的个数。 它会给每个处理器建立一条worker线程，Map和Reduce任务会动态地分配给此线程来执行。

为了启动Map环节，调度器会用Splitter把用户输入对划分为一些大小相等的单元。 Map任务负责处理这些单元。 每项Map任务只调用Splitter一次，之后，Splitter返回Map任务将要处理的数据。 Map任务是动态地分配到worker线程上的，每个任务会产生中间$\langle key, value\rangle$对。 Partition函数会把中间对分解成Reduce任务所需的单元。 此函数可以确保相同key的所有value都会转到同一单元里。 在各缓冲区中，value是按key来辅助最终排序的。 此时，Map环节结束。 调度器必需等待所有Map任务执行完毕，才

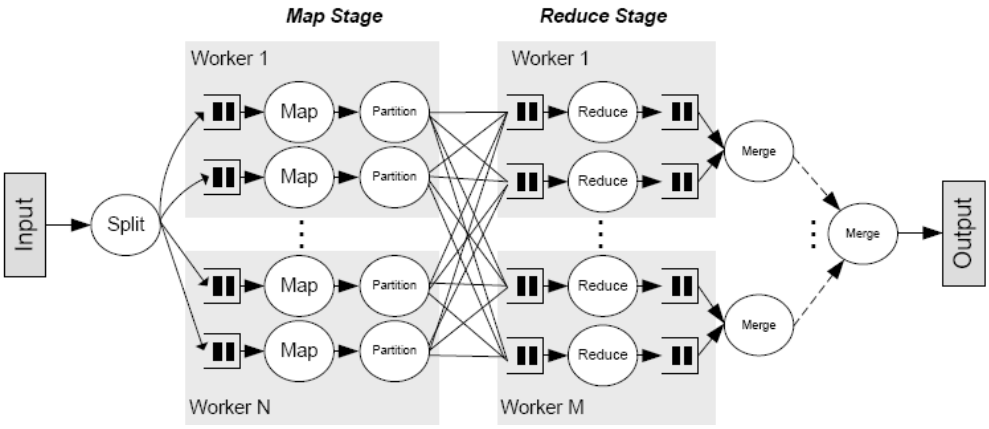

图 5: Phoenix运行时的基本流程

开始Reduce环节。

Reduce任务和Map任务一样，也是动态地分配到worker的。 唯一的不同点是， Map任务在处理对时，各任务是完全独立的； 而Reduce任务必需在一个任务中，处理相同key的所有value。 因此，Reduce环节的各个worker可能会出现负载不均衡，动态分配至关重要。 每项Reduce任务的输出由key排序。 最后，所有任务的最终输出结果会被合并到一个单缓冲区中，由key排序。 合并过程有 $\log_2 \frac{p}{2}$ 步，其中p是实际运行worker的数目。 读者可能会发现，输出结果对可以不用排序，目前Phoneix输出结果的顺序和Google实现的顺序是一样的。

3.3.3 缓冲管理

两个不同环节之间，需要两种临时缓冲区。 所有缓冲区是分配在共享内存里，但它们只能够由一组特定的函数去访问。 当我们必须重不是操作真实的数据，因为数据有可能很大。 另外，中间缓冲区对用户程序来说是不可见的。

Map-Reduce缓冲区用来存储Map任务所产生的中间输出对。 每个worker都有属于自己的一组缓冲区。 初始化时，缓冲区的大小为默认值；在执行任务时，其大小根据需求动态地变化。 在此环节中，同一key可能对应多个对。 为了加快Partition函数的速度，Emit_intermediate函数会把相同key的所有值都保存在同一缓冲区中。 在Map任务结束时，我们把所有的缓冲区按key排好序。 Reduce-Merge缓冲区用来存储Reduce任务还未排序前的输出。 在此环节里，每个key只关联一个值。 排序后，最终输出可以通过用

户分配的Output_data缓冲来访问。

3.3.4 容错

运行时为在执行Map或Reduce任务时发生的意外或持久的错误提供容错处理。 它只主要针对有限的服务提供修复。

Phoenix通过超时来检测错误。 以其它worker上做类似任务所需的时间作为超时间隔时间的标准， 如果一个worker没有在规定的时间内完成任务，我们就怀疑其间有错误发生。 当然错误通常可能导致不正确的或未完成的数据，而不是完完全全地失败。 Phoenix自身没有办法检测这种情况，并且不能阻止有错误的任务污染共享内存区。 我们可以把Phoenix的运行时与已有的错误检测技术[13, 14, 15]来补全此缺陷。 通过MapReduce模型的函数式本能，Phoenix可以提供简化错误检测的相关信息。 比如输入和输出的地址范围既然是已知的， 那Phoenix可以通知硬件哪些共享结构的加载或存储地址对于worker来说是安全的， 哪些可能是不安全的。如果可能是对worker不安全的，则Phoenix会发送信号给硬件。

当有错误发生或者被挂起时，运行时系统会尝试重新执行已失败的任务。 由于原始的失败任务可能还在执行，新启动的代替任务会获得新的输出缓冲区， 否则新任务的地址空间会与原任务的起冲突而污染数据。 当两任务的其中一个执行完毕时，运行时会采纳执行完毕的任务， 并把它的结果合并到该环节的输出数据中。 调度器开始假设此错误是只是个意外，它会把新的代替任务分配给原来那个worker。 如果此任务在很短的时候又发生错误，或者此worker上的错误频繁出现， 调度器则认为此错误不是意外，而是一个持久的错误。 此时它再也不会把任务分配给此worker了。

目前Phoenix的调度器自身不提供错误修复。

3.3.5 并发和本地管理

运行时调度策略的效率直接引响到所有并行任务的性能。 一般情况下，有三种可以选择的调度方法： 1) 在开发中已经为其特性作过考虑的系统中使用默认的策略； 2) 监视有效资源和运行时的行为，动态决定最优的调度策略； 3) 允许用户自定义策略。 三种方法,Phoenix在作调度决策时都使用了。

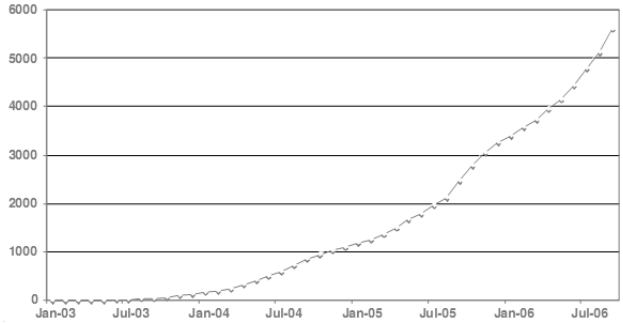

图 6: MapReduce程序在Google源代码树的个数(图片来自Google)

图 7: Yahoo!的Hadoop集群(图片来自Yahoo!)

4 案例研究与未来展望

4.1 Google

了简化大规模并行程序的开发，Google和IBM提供了以下资源： 一个处理器集群，运行的是Google的计算设施的一个开源实现（来自Apache Hadoop项目的MapReduce和GFS）。 由Google和University of Washington共同开发了一个课程， 讲授大规模并行计算技术，并且采用的是Creative Commons许可。

4.2 Lucene和Nutch

4.3 Yahoo!的M45与PIG

今年11月12日，Yahoo!公司公布了一个研究高级分布式计算的开源项目。

该声明的关键是Yahoo!将建立一个使用Hadoop的超级计算数据中心，名为M45。 该集群“约有4,000个处理器，3T的内存， 1.5P字节的磁盘，性能峰值可超过每秒27万亿次计算（27 teraflops）， 跻身全球排名前50的超级计算机之列”。 12月Yahoo Developer Network还首发了一个以Hadoop和分布式计算为主题的博客。 Carnegie Mellon大学将是此项研发工作的第一个高校合作者。

Pig是Yahoo公司研发部的产品，专门用来分析大数据集的平台。它由表达数据分析流程的高级语言和执行流程的基础结构两者组成。 Pig程序最显著的特点是，它们的基础结构使能建立非常稳固的并行化操作，足以应付大数据集的分析与处理。

目前的Pig基础结构层由一个产生MapReduce执行序列的编译器组成。MapReduce的大规模并行实现已经存在，比如Hadoop子项目。Pig语言层目前由一种叫Pig Latin文本语言组成。 这种语言有以下几个特征:

- 简单易懂。它使“让人头痛的并行”数据分析任务变得很简单。 程序代码中可以通过数据流序列显式地声明 由多种交叉数据转换合成的复合任务，让这项工作变得容易编写、理解，同时也易于推护。
- 优化空间大。 系统会自动地优化任务的执行，这种方法让用户更关注语义层面的优化，而不是效率。
- 良好的扩展性。用户可以为特定的数据处理建立自己定义的函数。

4.4 Amazon的EC2与S3

Amazon公司的EC2，全称Elastic Compute Cloud，是一种提供主机的计算服务，目前仍在beta阶段，但已在 硅谷火热蔓延中。用户通过按小时计费的方法，租用一组主机，使用者可以做好Amazon映像(AMI)文件 [1] 放在他们的平台上运行，运行完毕后主机使用权交还给Amazon公司。 因此EC2可以让用户只需付年租费就可以部署Hadoop应用，而不必购买和管理自己的集群系统。 而Hadoop分布式计算服务在Amazon的此项商业战略当中占着重要地位。

Amazon的S3可以让用户在EC2运行Hadoop。S3 (Simple Storage Service)是一种数据存储服务，数据的存 储和转发按月计费，而与EC2之间的数据转发是免费的。这是吸引EC2上面运行Hadoop的用户，同时可以享用S3的商业策略。当集群系统开始运行时，初始的输入可以从S3上读入，

[1] Amazon机器映像(AMI)，即Amazon Machine Image，是一种自启动的Linux映像文件。 在集群中使用Hadoop，可以使用一些开放的Hadoop AMI，它们提供了所有运行Hadoop应用需要的程序。

最终的输出可以在集群系统退出 之前写回到S3上。而在MapReduce过程中产生的中间、临时数据被高效地存储在Hadoop的分布式文件系统 HDFS上面。

在Hadoop的MapReduce中当下面碰到两种情况时，可以考虑使用S3:

- 用户要求支持巨大数据集，同时又保持安全性的分布式文件系统(用S3代替HDFS)
- 用户需要方便输出输出的存储仓库

运行Hadoop实例的集群系统是建立在安全的组上的。 组内的机器可以自由地相互访问，而在组外的机器（例如你的工作站）只能通过端口22的SSH， 端口50030(JobTracker的web接口，允许用户查看任务的状态)， 和端口50060(TaskTracker的web接口，用来进行详细的调试)对组内机器进行访问。

4.5 未来展望

在高能物理研究方面，很多实验每秒钟都能几个TB的数据量。 但因为处理能力和存储能力的不足，科学家不得不把绝大部分未经处理的数据丢弃掉。 可大家要知道，新元素的信息很有可能就藏在我们来不及处理的数据里面。 同样的，在其他领域里，高性能计算可以改变人类的生活。 例如人类基因的研究，就可能因为算法而发明新的医疗方式。 在国家安全领域，有效的算法可能避免下一个911的发生。 在气象方面，算法可以更好地预测未来天灾的发生，以拯救生命。 MapReduce模型适用于这些领域，而我们相信这些应用的实现只是时间问题。

参 考 文 献

[1] J. Dean and S. Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, In OSDI' 04, 6th Symposiumon Operating Systems Design and Implementation, Sponsored by USENIX, incooperation with ACM SIGOPS, pages 137-150, 2004.

[2] Ralf Lammel, *Google' s MapReduce programming model — Revisited*, Data Programmability Team, Microsoft Corp., Redmond, WA, USA, 18 July 2007.

[3] Jeffrey Dean, *Experiences with MapReduce, an abstraction for large-scale computation. Proc*, 15th International Conference on Parallel Architectures and Compilation Techniques, 2006, pp. 1.

[4] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung, *The Google File System*, Proceedings of the 19th ACM Symposium on Operating Systems Principles, 2003, pp. 20-43.

[5] Alex Rasmussen and Maximus Daehan Choi, *Improving Performance in MapReduce Through Pipelining*,December 18, 2006.

[6] Chris Marth, *Google MapReduce*, CS580 - Computational Science, Montana State University - Bozeman, April 4, 2005.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach,Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber, *Bigtable: A Distributed Storage System for Structured Data*, 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2006, pp. 205-218.

[8] Doug Cutting, *Scalable Computing with Hadoop*, Yahoo Inc!, May 4th, 2006.

[9] Owen O' Malley, Eric Baldeschwieler, *Petabyte Scale on Commodity Infrastructure*, Yahoo Inc!.

[10] Owen O' Malley, *Introduction to Hadoop*, Yahoo Inc!.

[11] Dhruba Borthakur, *The Hadoop Distributed File System: Architecture and Design*, The Apache Software Foundation, 2007.

[12] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, Christos Kozyrakis, *Evaluating MapReduce for Multi-core and Multiprocessor Systems*, Proceedings of the 13th Intl, Symposium on High-Performance Computer Architecture (HPCA), Phoenix, AZ, February 2007.

[13] S.Mitra et al, *Robust System Design with Built-In Soft-Error Resilience*, IEEE Computer, 38(2), Feb 2005.

[14] S.S.Mukherjee et al, *Detailed Design and Evaluation of Redundant Multithreading Alternatives*, In the proceedings of the 29th Intl. Symp. on Computer architechture, May 2002.

[15] J.C.Smolens et al, *Fingerprinting: Bounding Soft-error Detection Latency and Bandwidth*, In Proceedings of the 11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, Oct. 2004.