mapreduce 中文版

mapreduce 中文版 1

MapReduce:超大机群上的简单数据处理

摘要

MapReduce 是一个编程模型,和处理,产生大数据集的相关实现.用户指定一个 map 函数处理一个 key/value 对,从而产生中间的 key/value 对集.然后再指定一个 reduce 函数合并所有的具有相同中间 key 的中间 value.下面将列举许多可以用这个模型来表示的现实世界的工作.

以这种方式写的程序能自动的在大规模的普通机器上实现并行化.这个运行时系统关心这些细节:分割输入数据,在机群上的调度,机器的错误处理,管理机器之间必要的通信.这样就可以让那些没有并行分布式处理系统经验的程序员利用大量分布式系统的资源.

我们的 MapReduce 实现运行在规模可以灵活调整的由普通机器组成的机群上,一个典型的 MapReduce 计算处理几千台机器上的以 TB 计算的数据.程序员发现这个系统非常好用:已经实现了数以百计的 MapReduce 程序,每天在 Google 的机群上都有 1000 多个 MapReduce 程序在执行.

1.介绍

在过去的5年里,作者和Google的许多人已经实现了数以百计的为专门目的而写的计算来处理大量的原始数据,比如,爬行的文档,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,我们设计一个新的抽象模型,它让我们表示我们将要执行的简单计算,而隐藏并行化,容错,数据分布,负载均衡的那些杂乱的细节,在一个库里.我们的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.我们认识到我们的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,来适当的合并派生的数据.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,和使用再次执行作为初级机制来实现容错.

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来在大量普通的 PC 机上实现高性能计算.

第二部分描述基本的编程模型,并且给一些例子.第三部分描述符合我们的基于集群的计算环境

的 MapReduce 的接口的实现.第四部分描述我们觉得编程模型中一些有用的技巧.第五部分对于各种不同的任务,测量我们实现的性能.第六部分探究在 Google 内部使用 MapReduce 作为基础来重写我们的索引系统产品.第七部分讨论相关的,和未来的工作.

2.编程模型

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 实例

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(用 C++实现).附录 A 包含这个实例的全部文本.

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 更多实例

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.这个计算依赖分割工具(在 4.1 描述)和排序属性(在 4.2 描述).

mapreduce 中文版 2

3 实现

MapReduce 接口可能有许多不同的实现.根据环境进行正确的选择.例如,一个实现对一个共享内存较小的机器是合适的,另外的适合一个大 NUMA 的多处理器的机器,而有的适合一个更大的网络机器的集合.

这部分描述一个在 Google 广泛使用的计算环境的实现:用交换机连接的普通 PC 机的大机群.我们的环境是:

1.Linux 操作系统,双处理器,2-4GB 内存的机器.

2.普通的网络硬件,每个机器的带宽或者是百兆或者千兆,但是平均小于全部带宽的一半.

3.因为一个机群包含成百上千的机器,所有机器会经常出现问题.

4.存储用直接连到每个机器上的廉价 IDE 硬盘.一个从内部文件系统发展起来的分布式文件系统被用来管理存储在这些磁盘上的数据.文件系统用复制的方式在不可靠的硬件上来保证可靠性和有效性.

5.用户提交工作给调度系统.每个工作包含一个任务集,每个工作被调度者映射到机群中一个可用的机器集上.

3.1 执行预览

通过自动分割输入数据成一个有 M 个 split 的集,map 调用被分布到多台机器上.输入的 split 能够在不同的机器上被并行处理.通过用分割函数分割中间 key,来形成 R 个片(例如,hash(key) mod R),reduce 调用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 1 显示了我们实现的 MapReduce 操作的全部流程.当用户的程序调用 MapReduce 的函数的时候,将发生下面的一系列动作(下面的数字和图 1 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的拷贝程序.

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.管理者分配一个 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,管理者唤醒用户程序.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.

3.2master 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.

3.3 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 失败

可以很容易的让管理者周期的写入上面描述的数据结构的 checkpoints.如果这个 master 任务失效了,可以从上次最后一个 checkpoint 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/和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.

3.4 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.

3.5 任务粒度

象上面描述的那样,我们细分map阶段成M个片,reduce阶段成R个片.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,这个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.(这个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.我们经常执行MapReduce计算,在M=200000,R=5000,使用2000台工作者机器的情况下.

mapreduce 中文版 3

3.6 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.我们最近遇到的一个问题是,一个在机器初始化时的Bug引起处理器缓存的

失效:在一个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的减少完成大规模MapReduce操作的时间.作为一个例子,将要在5.3描述的排序程序,在关闭掉备用任务的情况下,要比有备用任务的情况下多花 44%的时间.

4 技巧

尽管简单的map和reduce函数的功能对于大多数需求是足够的了,但是我们开发了一些有用的扩充.这些将在这个部分描述.

4.1 分割函数

MapReduce 用户指定 reduce 任务和 reduce 任务需要的输出文件的数量.在中间 key 上使用分割函数,使数据分割后通过这些任务.一个缺省的分割函数使用 hash 方法(例如,hash(key) mod R).这个导致非常平衡的分割.然后,有的时候,使用其他的 key 分割函数来分割数据有非常有用的.例如,有时候,输出的 key 是 URLs,并且我们希望每个主机的所有条目保持在同一个输出文件中.为了支持像这样的情况,MapReduce 库的用户可以提供专门的分割函数.例如,使用"hash(Hostname(urlkey)) mod R"作为分割函数,使所有来自同一个主机的 URLs 保存在同一个输出文件中.

4.2 顺序保证

我们保证在一个给定的分割里面,中间key/value对以key递增的顺序处理.这个顺序保证可以使每个分割产出一个有序的输出文件,当输出文件的格式需要支持有效率的随机访问 key 的时候,或者对输出数据集再作排序的时候,就很容易.

4.3combiner 函数

在某些情况下,允许中间结果 key 重复会占据相当的比重,并且用户定义的 reduce 函数满足结合律和交换律.一个很好的例子就是在 2.1 部分的词统计程序.因为词频率倾向于一个 zipf 分布(齐夫分布),每个 map 任务将产生成百上千个这样的记录<the,1>.所有的这些计数将通过网络被传输到一个单独的reduce任务,然后由reduce函数加在一起产生一个数字.我们允许用户指定一个可选的 combiner 函数,先在本地进行合并一下,然后再通过网络发送.

在每一个执行 map 任务的机器上 combiner 函数被执行.一般的,相同的代码被用在 combiner 和 reduce 函数.在 combiner 和 reduce 函数之间唯一的区别是 MapReduce 库怎样控制函数的输出.reduce 函数的输出被保存最终输出文件里.combiner 函数的输出被写到中间文件里,然后被发送给 reduce 任务.

部分使用 combiner 可以显著的提高一些 MapReduce 操作的速度.附录 A 包含一个使用 combiner 函数的例子.

4.4 输入输出类型

MapReduce 库支持以几种不同的格式读取输入数据.例如,文本模式输入把每一行看作是一个

key/value 对.key 是文件的偏移量,value 是那一行的内容.其他普通的支持格式以 key 的顺序存储 key/value 对序列.每一个输入类型的实现知道怎样把输入分割成对每个单独的 map 任务来说是有意义的(例如,文本模式的范围分割确保仅仅在每行的边界进行范围分割).虽然许多用户仅仅使用很少的预定意输入类型的一个,但是用户可以通过提供一个简单的 reader 接口来支持一个新的输入类型.

一个 reader 不必要从文件里读数据.例如,我们可以很容易的定义它从数据库里读记录,或从内存中的数据结构读取.

4.5 副作用

有的时候,MapReduce 的用户发现在 map 操作或/和 reduce 操作时产生辅助文件作为一个附加的输出是很方便的.我们依靠应用程序写来使这个副作用成为原子的.一般的,应用程序写一个临时文件,然后一旦这个文件全部产生完,就自动的被重命名.

对于单个任务产生的多个输出文件来说,我们没有提供其上的两阶段提交的原子操作支持.因此,一个产生需要交叉文件连接的多个输出文件的任务,应该使确定性的任务.不过这个限制在实际的工作中并不是一个问题.

4.6 跳过错误记录

有的时候因为用户的代码里有 bug,导致在某一个记录上 map 或 reduce 函数突然 crash 掉.这样的 bug 使得 MapReduce 操作不能完成.虽然一般是修复这个 bug,但是有时候这是不现实的;也许这个 bug 是在源代码不可得到的第三方库里.有的时候也可以忽略一些记录,例如,当在一个大的数据集上进行统计分析.我们提供一个可选的执行模式,在这个模式下,MapReduce 库检测那些记录引起的 crash,然后跳过那些记录,来继续执行程序.

每个 worker 程序安装一个信号处理器来获取内存段异常和总线错误.在调用一个用户自定义的 map 或 reduce 操作之前,MapReduce 库把记录的序列号存储在一个全局变量里.如果用户代码产生一个信号,那个信号处理器就会发送一个包含序号的"last gasp"UDP 包给 MapReduce 的 master.当 master 不止一次看到同一个记录的时候,它就会指出,当相关的 map 或 reduce 任务再次执行的时候,这个记录应当被跳过.

4.7 本地执行

调试在 map 或 reduce 函数中问题是很困难的,因为实际的计算发生在一个分布式的系统中,经常是有一个 master 动态的分配工作给几千台机器.为了简化调试和测试,我们开发了一个可替换的实现,这个实现在本地执行所有的 MapReduce 操作.用户可以控制执行,这样计算可以限制到特定的 map 任务上.用户以一个标志调用他们的程序,然后可以容易的使用他们认为好用的任何调试和测试工具(例如,gdb).

4.8 状态信息

master 运行一个 HTTP 服务器,并且可以输出一组状况页来供人们使用.状态页显示计算进度,象多少个任务已经完成,多少个还在运行,输入的字节数,中间数据字节数,输出字节数,处理百分比,等等.这个页也包含到标准错误的链接,和由每个任务产生的标准输出的链接.用户可以根据这些数据预测计算需要花费的时间,和是否需要更多的资源.当计算比预期的要慢很多的时候,这些页

面也可以被用来判断是不是这样.

此外,最上面的状态页显示已经有多少个工作者失败了,和当它们失败的时候,那个map和reduce任务正在运行.当试图诊断在用户代码里的bug时,这个信息也是有用的.

4.9 计数器

MapReduce库提供一个计数器工具,来计算各种事件的发生次数.例如,用户代码想要计算所有处理的词的个数,或者被索引的德文文档的数量.

为了使用这个工具,用户代码创建一个命名的计数器对象,然后在map或/和reduce函数里适当的增加计数器.例如:

```
Counter * uppercase;
uppercase=GetCounter("uppercase");
map(String name,String contents):
for each word w in contents:
    if(IsCapitalized(w)):
     uppercase->Increment();
    EmitIntermediate(w,"1");
```

来自不同worker机器上的计数器值被周期性的传送给master(在ping回应里).master把来自成功的map和reduce任务的计数器值加起来,在MapReduce操作完成的时候,把它返回给用户代码.当前计数器的值也被显示在master状态页里,以便人们可以查看实际的计算进度.当计算计数器值的时候消除重复执行的影响,避免数据的累加.(在备用任务的使用,和由于出错的重新执行,可以产生重复执行)

有些计数器值被MapReduce库自动的维护,比如,被处理的输入key/value对的数量,和被产生的输出key/value对的数量.

用户发现计数器工具对于检查MapReduce操作的完整性很有用.例如,在一些MapReduce操作中,用户代码也许想要确保输出对的数量完全等于输入对的数量,或者处理过的德文文档的数量是在全部被处理的文档数量中属于合理的范围.

mapreduce 中文版 4

5 性能

在本节,我们用在一个大型集群上运行的两个计算来衡量MapReduce的性能.一个计算用来在一个大概1TB的数据中查找特定的匹配串.另一个计算排序大概1TB的数据.

这两个程序代表了MapReduce的用户实现的真实的程序的一个大子集.一类是,把数据从一种表示转化到另一种表示.另一类是,从一个大的数据集中提取少量的关心的数据.

5.1 机群配置

所有的程序在包含大概1800台机器的机群上执行.机器的配置是:2个2G的Intel Xeon超线程处理器,4GB内存,两个160GB IDE磁盘,一个千兆网卡.这些机器部署在一个由两层的,树形交换网络中,在根节点上大概有100到2000G的带宽.所有这些机器都有相同的部署(对等部署),因此

任意两点之间的来回时间小于 1 毫秒.

在 4GB 的内存里,大概有 1-1.5GB 被用来运行在机群中其他的任务.这个程序是在周末的下午开始执行的,这个时候 CPU,磁盘,网络基本上是空闲的.

5.2Grep

这个 Grep 程序扫描大概 10^10 个,每个 100 字节的记录,查找比较少的 3 字符的查找串(这个查找串出现在 92337 个记录中).输入数据被分割成大概 64MB 的片(M=15000),全部 的输出存放在一个文件中(R=1).

图 2 显示计算过程随时间变化的情况.Y 轴表示输入数据被扫描的速度.随着更多的机群被分配给这个 MapReduce 计算,速度在逐步的提高,当有 1764 个 worker 的时候这个速度达到最高的 30GB/s.当 map 任务完成的时候,速度开始下降,在计算开始后 80 秒,输入的速度降到 0.这个计算持续的时间大概是 150 秒.这包括了前面大概一分钟的启动时间.启动时间用来把程序传播到所有的机器上,等待 GFS 打开 1000 个输入文件,得到必要的位置优化信息.

5.3 排序

这个 sort 程序排序 10^10 个记录,每个记录 100 个字节(大概 1TB 的数据).这个程序是模仿 TeraSort 的.

这个排序程序只包含不到 50 行的用户代码.其中有 3 行 map 函数用来从文本行提取 10 字节的排序 key,并且产生一个由这个 key 和原始文本行组成的中间 key/value 对.我们使用一个内置的 Identity 函数作为 reduce 操作.这个函数直接把中间 key/value 对作为输出的 key/value 对.最终的排序输出写到一个 2 路复制的 GFS 文件中(也就是,程序的输出会写 2TB 的数据).

象以前一样,输入数据被分割成 64MB 的片(M=15000).我们把排序后的输出写到 4000 个文件中(R=4000).分区函数使用 key 的原始字节来把数据分区到 R 个小片中.

我们以这个基准的分割函数,知道 key 的分布情况.在一般的排序程序中,我们会增加一个预处理的 MapReduce 操作,这个操作用于采样 key 的情况,并且用这个采样的 key 的分布情况来计算对最终排序处理的分割点。

图 3(a)显示这个排序程序的正常执行情况.左上图显示输入数据的读取速度.这个速度最高到达 13GB/s,并且在不到 200 秒所有 map 任务完成之后迅速滑落到 0.注意到这个输入速度小于 Grep.这是因为这个排序 map 任务花费大概一半的时间和带宽,来把中间数据写到本地硬盘中.而 Grep 相关的中间数据可以忽略不计.

左中图显示数据通过网络从 map 任务传输给 reduce 任务的速度.当第一个 map 任务完成后,这个排序过程就开始了.图示上的第一个高峰是启动了第一批大概 1700 个 reduce 任务(整个 MapReduce 任务被分配到 1700 台机器上,每个机器一次只执行一个 reduce 任务).大概开始计算后的 300 秒,第一批 reduce 任务中的一些完成了,我们开始执行剩下的 reduce 任务.全部的排序过程持续了大概 600 秒的时间.

左下图显示排序后的数据被 reduce 任务写入最终文件的速度.因为机器忙于排序中间数据,所以在第一个排序阶段的结束和写阶段的开始有一个延迟.写的速度大概是 2-4GB/s.大概开始计算

后的 850 秒写过程结束. 包括前面的启动过程,全部的计算任务持续的 891 秒. 这个和 TeraSort benchmark 的最高纪录 1057 秒差不多.

需要注意的事情是:因此位置优化的原因,很多数据都是从本地磁盘读取的而没有通过我们有限带宽的网络,所以输入速度比排序速度和输出速度都要快. 排序速度比输出速度快的原因是输出阶段写两个排序后数据的拷贝(我们写两个副本的原因是为了可靠性和可用性). 我们写两份的原因是因为底层文件系统的可靠性和可用性的要求. 如果底层文件系统用类似容错编码(erasure coding)的方式,而不采用复制写的方式,在写盘阶段可以降低网络带宽的要求。

5.4 备用任务的影响

在图 3(b)中,显示我们不用备用任务的排序程序的执行情况. 除了它有一个很长的几乎没有写动作发生的尾巴外,执行流程和图 3(a)相似. 在 960 秒后,只有 5 个 reduce 任务没有完成. 然而,就是这最后几个落后者知道 300 秒后才完成. 全部的计算任务执行了 1283 秒,多花了 44%的时间.

5.5 机器失效

在图 3(c)中,显示我们有意的在排序程序计算过程中停止 1746 台 worker 中的 200 台机器上的程序的情况. 底层机群调度者在这些机器上马上重新开始新的 worker 程序(因为仅仅程序被停止,而机器仍然在正常运行).

因为已经完成的 map 工作丢失了(由于相关的 map worker 被杀掉了),需要重新再作,所以 worker 死掉会导致一个负数的输入速率. 相关 map 任务的重新执行很快就重新执行了. 整个计算过程在 933 秒内完成,包括了前边的启动时间(只比正常执行时间多了 5%的时间).

6 经验

我们在 2003 年的 2 月写了 MapReduce 库的第一个版本,并且在 2003 年的 8 月做了显著的增强,包括位置优化,worker 机器间任务执行的动态负载均衡,等等. 从那个时候起,我们惊奇的发现 MapReduce 函数库广泛用于我们日常处理的问题. 它现在在 Google 内部各个领域内广泛应用,包括:

大规模机器学习问题

Google News 和 Froogle 产品的机器问题.

提取数据产生一个流行查询的报告(例如,Google Zeitgeist).

为新的试验和产品提取网页的属性(例如,从一个 web 页的大集合中提取位置信息　用在位置查询).

大规模的图计算.

图 4 显示了我们主要的源代码管理系统中,随着时间推移,MapReduce 程序的显著增加,从 2003 年早先时候的 0 个增长到 2004 年 9 月份的差不多 900 个不同的程序. MapReduce 之所以这样的成功,是因为他能够在不到半小时时间内写出一个简单的能够应用于上千台机器的大规模并发程序,并且极大的提高了开发和原形设计的周期效率. 并且,他可以让一个完全没有分布式和/或并行系统经验的程序员,能够很容易的利用大量的资源.

在每一个任务结束的时候,MapReduce 函数库记录使用的计算资源的统计信息. 在图 1 里,我们列出了 2004 年 8 月份在 Google 运行的一些 MapReduce 的工作的统计信息.

mapreduce 中文版 5

6.1 大规模索引

到目前为止,最成功的 MapReduce 的应用就是重写了 Google web 搜索服务所使用到的 index 系统.索引系统处理爬虫系统抓回来的超大量的文档集,这些文档集保存在 GFS 文件里.这些文档的原始内容的大小,超过了 20TB.索引程序是通过一系列的,大概 5 到 10 次 MapReduce 操作来建立索引.通过利用 MapReduce(替换掉上一个版本的特别设计的分布处理的索引程序版本)有这样一些好处:

索引的代码简单,量少,容易理解,因为容错,分布式,并行处理都隐藏在 MapReduce 库中了.例如,当使用 MapReduce 函数库的时候,计算的代码行数从原来的 3800 行 C++代码一下减少到大概 700 行代码.

MapReduce 的函数库的性能已经非常好,所以我们可以把概念上不相关的计算步骤分开处理,而不是混在一起以期减少在数据上的处理.这使得改变索引过程很容易.例如,我们对老索引系统的一个小更改可能要好几个月的时间,但是在新系统内,只需要花几天时间就可以了.

索引系统的操作更容易了,这是因为机器的失效,速度慢的机器,以及网络失效都已经由 MapReduce 自己解决了,而不需要操作人员的交互.另外,我们可以简单的通过对索引系统增加机器的方式提高处理性能.

7 相关工作

很多系统都提供了严格的设计模式,并且通过对编程的严格限制来实现自动的并行计算.例如,一个结合函数可以通过 N 个元素的数组的前缀在 N 个处理器上使用并行前缀计算在 log N 的时间内计算完.MapReduce 是基于我们的大型现实计算的经验,对这些模型的一个简化和精炼.并且,我们还提供了基于上千台处理器的容错实现.而大部分并发处理系统都只在小规模的尺度上实现,并且机器的容错还是程序员来控制的.

Bulk Synchronous Programming 以及一些 MPI primitives 提供了更高级别的抽象,可以更容易写出并行处理的程序.这些系统和 MapReduce 系统的不同之处在,MapReduce 利用严格的编程模式自动实现用户程序的并发处理,并且提供了透明的容错处理.

我们本地的优化策略是受 active disks 等技术的启发,在 active disks 中,计算任务是尽量推送到靠近本地磁盘的处理单元上,这样就减少了通过 I/O 子系统或网络的数据量.我们在少量磁盘直接连接到普通处理机运行,来代替直接连接到磁盘控制器的处理机上,但是一般的步骤是相似的.

我们的备用任务的机制和在 Charlotte 系统上的积极调度机制相似.这个简单的积极调度的一个缺陷是,如果一个任务引起了一个重复性的失败,那个整个计算将无法完成.我们通过在故障情况下跳过故障记录的机制,在某种程度上解决了这个问题.

MapReduce 实现依赖一个内置的机群管理系统来在一个大规模共享机器组上分布和运行用户任务.虽然这个不是本论文的重点,但是集群管理系统在理念上和 Condor 等其他系统是一样的.

在 MapReduce 库中的排序工具在操作上和 NOW-Sort 相似.源机器(map worker)分割将要被排序的

数据,然后把它发送到R个reduce worker中的一个上.每个reduce worker来本地排序它的数据(如果可能,就在内存中).当然,NOW-Sort没有用户自定义的map和reduce函数,使得我们的库可以广泛的应用.

River提供一个编程模型,在这个模型下,处理进程可以靠在分布式的队列上发送数据进行彼此通讯.和MapReduce一样,River系统尝试提供对不同应用有近似平均的性能,即使在不对等的硬件环境下或者在系统颠簸的情况下也能提供近似平均的性.River是通过精心调度硬盘和网络的通讯,来平衡任务的完成时间.MapReduce不和它不同.利用严格编程模型,MapReduce构架来把问题分割成大量的任务.这些任务被自动的在可用的worker上调度,以便速度快的worker可以处理更多的任务.这个严格编程模型也让我们可以在工作快要结束的时候安排冗余的执行,来在非一致处理的情况减少完成时间(比如,在有慢机或者阻塞的worker的时候).

BAD-FS是一个很MapReduce完全不同的编程模型,它的目标是在一个广阔的网络上执行工作.然而,它们有两个基本原理是相同的.(1)这两个系统使用冗余的执行来从由失效引起的数据丢失中恢复.(2)这两个系统使用本地化调度策略,来减少通过拥挤的网络连接发送的数据数量.

TACC是一个被设计用来简化高有效性网络服务结构的系统.和MapReduce一样,它通过再次执行来实现容错.

8 结束语

MapReduce编程模型已经在Google成功的用在不同的目的.我们把这个成功归于以下几个原因:第一,这个模型使用简单,甚至对没有并行和分布式经验的程序员也是如此,因为它隐藏了并行化,容错,位置优化和负载均衡的细节.第二,大量不同的问题可以用MapReduce计算来表达.例如,MapReduce被用来,为Google的产品web搜索服务,排序,数据挖掘,机器学习,和其他许多系统,产生数据.第三,我们已经在一个好几千台计算机的大型集群上开发实现了这个MapReduce.这个实现使得对于这些机器资源的利用非常简单,因此也适用于解决Google遇到的其他很多需要大量计算的问题.

从这个工作中我们也学习到了一些东西.首先,严格的编程模型使得并行化和分布式计算简单,并且也易于构造这样的容错计算环境.第二,网络带宽是系统的瓶颈.因此在我们的系统中大量的优化目标是减少通过网络发送的数据量,本地优化使用我们从本地磁盘读取数据,并且把中间数据写到本地磁盘,以保留网络带宽.第三,冗余的执行可以用来减少速度慢的机器的影响,和控制机器失效和数据丢失.

感谢

Josh Levenberg校定和扩展了用户级别的MapReduce API,并且结合他的适用经验和其他人的改进建议,增加了很多新的功能.MapReduce从GFS中读取和写入数据.我们要感谢Mohit Aron,Howard Gobioff,Markus Gutschke,David Krame,Shun-Tak Leung,和Josh Redstone,他们在开发GFS中的工作.我们还感谢Percy Liang Olcan Sercinoglu在开发用于MapReduce的集群管理系统得工作.Mike Burrows,Wilson Hsieh,Josh Levenberg,Sharon Perl,RobPike,Debby Wallach为本论文提出了宝贵的意见.OSDI的无名审阅者,以及我们的审核者Eric Brewer,在论文应当如何改进方面给出了有益的意见.最后,我们感谢Google的工程部的所有MapReduce的用

户,感谢他们提供了有用的反馈,建议,以及错误报告等等.

A 单词频率统计

本节包含了一个完整的程序,用于统计在一组命令行指定的输入文件中,每一个不同的单词出现频率.

```
#include "mapreduce/mapreduce.h"
//用户 map 函数
class WordCounter : public Mapper {
public:
virtual void Map(const MapInput& input) {
const string& text = input.value();
const int n = text.size();
for (int i = 0; i < n; ) {
//跳过前导空格
while ((i < n) && isspace(text[i]))
i++;
// 查找单词的结束位置
int start = i;
while ((i < n) && !isspace(text[i]))
i++;
if (start < i)
Emit(text.substr(start,i-start),"1");
}

}

};

REGISTER_MAPPER(WordCounter);
//用户的 reduce 函数
class Adder : public Reducer {
virtual void Reduce(ReduceInput* input) {
//迭代具有相同 key 的所有条目,并且累加它们的 value
int64 value = 0;
while (!input->done()) {
value += StringToInt(input->value());
input->NextValue();
```

```
                    }
                //提交这个输入 key 的综合
                Emit(IntToString(value));
                    }

                };
        REGISTER_REDUCER(Adder);
    int main(int argc, char** argv) {
        ParseCommandLineFlags(argc, argv);
            MapReduceSpecification spec;
            // 把输入文件列表存入"spec"
        for (int i = 1; i < argc; i++) {
            MapReduceInput* input = spec.add_input();
                input->set_format("text");
            input->set_filepattern(argv[i]);
            input->set_mapper_class("WordCounter");
                    }
                //指定输出文件:
            // /gfs/test/freq-00000-of-00100
            // /gfs/test/freq-00001-of-00100
                    // ...
        MapReduceOutput* out = spec.output();
        out->set_filebase("/gfs/test/freq");
            out->set_num_tasks(100);
            out->set_format("text");
        out->set_reducer_class("Adder");
    // 可选操作:在 map 任务中做部分累加工作,以便节省带宽
        out->set_combiner_class("Adder");
    // 调整参数: 使用 2000 台机器,每个任务 100MB 内存
            spec.set_machines(2000);
        spec.set_map_megabytes(100);
        spec.set_reduce_megabytes(100);
                // 运行它
            MapReduceResult result;
        if (!MapReduce(spec, &result)) abort();
    // 完成: 'result'结构包含计数,花费时间,和使用机器的信息
```

```
    return 0;
}
```