超大集群的简单数据处理

摘要

MapReduce是一个编程模式，它是与处理产生海量数据集的实现相关。用户指定一个map函数，通过这个map函数处理key/value（键/值）对，并且产生一系列的中间key/value对，并且使用reduce函数来合并所有的具有相同key值的中间键值对中的值部分。现实生活中的很多任务的实现都是基于这个模式的，正如本文稍后会讲述的那样。

使用这样的函数形式实现的程序可以自动分布到一个由普通机器组成的超大几群上并发执行。

run-time系统会解决输入数据的分布细节，跨越机器集群的程序执行调度，处理机器的失效，并且管理机器之间的通讯请求。这样的模式允许程序员可以不需要有什么并发处理或者分布式系统的经验，就可以处理超大的分布式系统得资源。

我们的MapReduce系统的实现运行在一个由普通机器组成的大型集群上，并且有着很高的扩展性：一个典型的MapReduce计算处理通常分布到上千台机器上来处理上TB的数据。程序员会发现这样的系统很容易使用：已经开发出来了上百个MapReduce程序，并且每天在Google的集群上有上千个MapReducejob正在执行。

1介绍

在过去的5年内，Google的创造者和其他人实现了上百个用于特别计算目的的程序来出来海量的原始数据，比如蠕虫文档，web请求log，等等，用于计算出不同的数据，比如降序索引，不同的图示展示的web文档，蠕虫采集的每个host的page数量摘要，给定日期内最常用的查询等等。绝大部分计算都是概念上很简洁的。不过，输入的数据通常是非常巨大的，并且为了能在合理时间内执行完毕，其上的计算必须分布到上百个或者上千个计算机上去执行。如何并发计算，如何分布数据，如何处理失败等等相关问题合并在一起就会导致原本简单的计算掩埋在为了解决这些问题而引入的很复杂的代码中。因为这种复杂度，我们设计了一种新的东西来让我们能够方便处理这样的简单计算。这些简单计算原本很简单，但是由于考虑到并发处理细节，容错细节，以及数据分布细节，负载均衡等等细节问题，而导致代码非常复杂。所以我们抽象这些公共的细节到一个lib中。这种抽象是源自Lisp以及其他很多面向功能的语言的map和reduce概念。我们认识到大部分操作都和map操作相关，这些map操作都是运算在输入记录的每个逻辑“record”上，并且map操作为了产生一组中间的key/value键值

并且接着在所有相同key的中间结果上执行reduce操作，这样就可以合并适当的数据。我们得函数模式是使用用户定义的map和reduce操作，这样可以让我们并发执行大规模的运算，并且使用重新执行的方式作为容错的优先机制。

MapReduce的主要贡献在于提供了一个简单强大的接口，通过这个接口，可以把大尺度的计算自动的并发和分布执行。使用这个接口，可以通过普通PC的巨大集群，来达到极高的性能。

第二节讲述了基本的编程模式，并且给出了一些例子。第三节讲述了一个面向我们基于集群的计算环境的MapReduce的实现。第四节讲述了一些我们建议的精巧编程模式。第五节讲述了在不同任务下我们的MapReduce实现的性能比较。第六节讲述了在Google中的MapReduce应用以及尝试重写了我们产品的索引系统。第七节讲述了相关工作和未来的工作。

2编程模式

我们的运算处理一组输入的（input）键值对（key/valuepairs）,并且产生一组输出的（output）键值对。MapReduce函数库德用户用两个函数来表达这样的计算：Map和Reduce。

Map函数，是用户自定义的的函数，处理输入的键值对，并且产生一组中间的（intermediate）键值对。MapReduce函数库稽核所有相同的中间键值键I的值，并且发送给Reduce函数进行处理。

Reduce函数同样也是用户提供的，它处理中间键值I，以及这个中间键值相关的值集合。这个函数合并这些值，最后形成一个相对较小的值集合。通常一个单次Reduce执行会产生0个或者1个输出

值。提供给Reduce函数的中间值是通过一个iterator来提供的。这就让我们可以处理超过内存容量的值列表。

2.1例子

我们考虑这样一个例子在很大的文档集合中通机每一个单词出现的次数。我们写出类似如下的伪代码：

map(Stringkey,Stringvalue):

//key:documentname

//value:documentcontentsforeachwordwinvalue:

EmitIntermediate(w,"1");

reduce(Stringkey,Iteratorvalues):

//key:aword

//values:alistofcountsintresult=0;

foreachvinvalues:

result+=ParseInt(v);Emit(AsString(result));

map函数检查每一个单词，并且对每一个单词增加1到其对应的计数器（在这个例子里就是’1’）.reduce函数把特定单词的所有出现的次数进行合并。

此外，我们还要写代码来对mapreducespecification对象进行赋值，设定输入和输出的文件名，以及设定一些参数。接着我们调用MapReduce函数，把这个对象作为参数调用过去。我们把MapReduce函数库（C++函数库）和我们的程序链接在一起。附件1有完整的这个例子的代码。

2.2类型

即使上边的例子是用字符串作为输入和输入出的，从概念上讲，使用者提供的map和reduce函数有着如下相关类型：

map(k1,v1)list(k2,v2)

reduce(k2,list(v2))list(v2)

也就是，输入的键和值和输出的键值是属于不同的域的。进一步说，中间的键值是和输出的键值属于相同的域的。（比如map的输出，就是作为reduce的输入）。

我们的C++实现上，把字符串作为用户定义函数的输入和输出，由用户代码来自己识别字符串到合适的类型。

2.3其他例子

这里有一些简单有趣的例子，都可以简单的通过MapReduce计算模型来展示：

分布式Grep：如果map函数检查输入行，满足条件的时候，map函数就把本行输出。reduce

函数就是一个直通函数，简单的把中间数据输出就可以了。

URL访问频率统计：map函数处理webpag请求和应答（URL，1）的log。Reduce函数把所有相同的URL的值合并，并且输出一个成对的（URL，总个数）。

逆向Web-Link图：map函数输出所有包含指向targetURL的source网页，用（target,source）这样的结构对输出。Reduce函数局和所有关联相同targetURL的source列表，并且输出一个(target,list(source))这样的结构。

主机关键向量指标（Term-VectorperHosts）:关键词向量指标简而言之就是在一个文档或者一组文档中的重点次出现的频率，用(word,frequency)表达。map函数计算每一个输入文档（主机名字是从文档的URL取出的）的关键词向量，然后输出（hostname,关键词向量(Term-Vector)）。reduce函数处理所有相同host的所有文档关键词向量。去掉不常用的关键词，并且输出最终的(hostname，关键词向量)对。

逆序索引：map函数分析每一个文档，并且产生一个序列（word，documentID）组。reduce函数处理指定word的所有的序列组，并且对相关的documentID进行排序，输出一个(word,list(documentID))组。所有的输出组，组成一个简单的逆序索引。通过这种方法可以很容易保持关键词在文档库中的位置。

分布式排序：map函数从每条记录中抽取关键字，并且产生(key,record)对。reduce函数原样输出所有的关键字对。这个算法是与4.1节描述的分布式处理相关的，并且排序是在4.2节描述的。

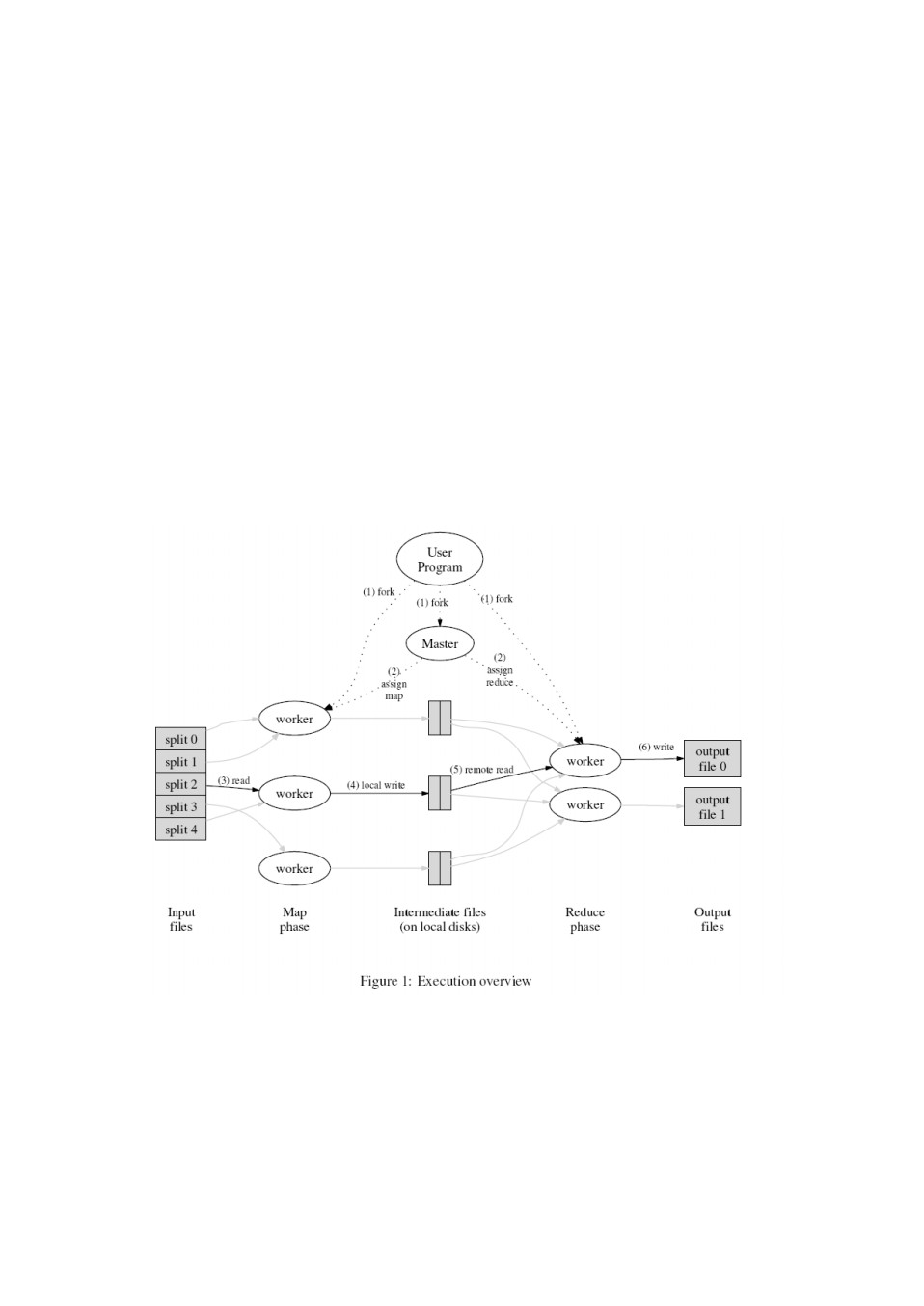
3实现

MapReduce接口可以有很多种不同的实现。应当根据不同的环境选择不同的实现。比如，一个实现可以适用于小型的共享内存的机器，另一个实现可能是基于大型NUMA多处理器系统，还可能有为大规模计算机集群的实现。

本届描述了Google广泛使用的计算环境：用交换机网络[4]连接的，由普通PC构成的超大集群。在我们的环境里：

（1）每个节点通常是双x86处理器，运行Linux,每台机器2-4GB内存。

（2）使用的网络设备都是常用的。一般在节点上使用的是100M/或者千M网络，一般情况下都用不到一半的网络带宽。



（3）一个cluster中常常有成百上千台机器，所以，机器故障是家常便饭。

（4）存储时使用的便宜的IDE硬盘，直接放在每一个机器上。并且有一个分布式的文件系统来管理这些分布在各个机器上的硬盘。文件系统通过复制的方法来在不可靠的硬件上保证可用性和可靠性。

（5）用户向调度系统提交请求。每一个请求都包含一组任务，映射到这个计算机cluster里的一组机器上执行。

3.1执行概览

Map操作通过把输入数据进行分区（partition）（比如分为M块），就可以分布到不同的机器上执行了。输入块的拆成多块，可以并行在不同机器上执行。Reduce操作是通过对中间产生的key的分布来进行分布的，中间产生的key可以根据某种分区函数进行分布（比如hash(key)modR）,分布成为R块。分区（R）的数量和分区函数都是由用户指定的。

图1是我们实现的MapReduce操作的整体数据流。当用户程序调用MapReduce函数，就会引起如下的操作（图一中的数字标示和下表的数字标示相同）。

1．用户程序中的MapReduce函数库首先把输入文件分成M块，每块大概16M到64M（可以通过参数决定）。接着在cluster的机器上执行处理程序。

第4页

2．这些分排的执行程序中有一个程序比较特别，它是主控程序master。剩下的执行程序都是作为master分排工作的worker。总共有M个map任务和R个reduce任务需要分排。master选择空闲的worker并且分配这些map任务或者reduce任务

3．一个分配了map任务的worker读取并处理相关的输入小块。他处理输入的数据，并且将分析出

的key/value对传递给用户定义的map函数。map函数产生的中间结果key/value对暂时缓冲到内存。

4．这些缓冲到内存的中间结果将被定时刷写到本地硬盘，这些数据通过分区函数分成R个区。这些中间结果在本地硬盘的位置信息将被发送回master，然后这个master负责把这些位置信息传送给reduce的worker。

5．当master通知reduce的worker关于中间key/value对的位置时，他调用remoteprocedure来从mapworker的本地硬盘上读取缓冲的中间数据。当reduce的worker读到了所有的中间数据，他就使用中间key进行排序，这样可以使得相同key的值都在一起。因为有许多不同key的map都对应相同的reduce任务，所以，排序是必须的。如果中间结果集太大了，那么就需要使用外排序。

6．reduceworker根据每一个唯一中间key来遍历所有的排序后的中间数据，并且把key和相关的中间结果值集合传递给用户定义的reduce函数。reduce函数的对于本reduce区块的输出到一个最终的输出文件。

7．当所有的map任务和reduce任务都已经完成了的时候，master激活用户程序。在这时候

MapReduce返回用户程序的调用点。

当这些成功结束以后，mapreduce的执行数据存放在总计R个输出文件中（每个都是由reduce任务产生的，这些文件名是用户指定的）。通常，用户不需要合并这R个输出文件到一个文件，他们通常把这些文件作为输入传递到另一个MapReduce调用，或者用另一个分布式应用来处理这些文件，并且这些分布式应用把这些文件看成为输入文件由于分区（partition）成为的多个块文件。

3.2Master的数据结构

master需要保存一定的数据结构。对于每一个map和reduce任务来说，都需要保存它的状态（idle，in-progress或者completed），并且识别不同的worker机器（对于非idel的任务状态）。

master是一个由map任务产生的中间区域文件位置信息到reduce任务的一个管道。因此，对于每一个完成得map任务，master保存下来这个map任务产生的R中间区域文件信息的位置和大小。对于这个位置和大小信息是当接收到map任务完成得时候做的。这些信息是增量推送到处于

in-progress状态的reduce任务的worker上的。

3.3容错考虑

由于MapReduce函数库是设计用于在成百上千台机器上处理海量数据的，所以这个函数库必须考虑到机器故障的容错处理。

Worker失效的考虑

master会定期ping每一个worker机器。如果在一定时间内没有worker机器的返回，master就认为这个worker失效了。所有这台worker完成的map任务都被设置成为他们的初始idel状态，并且因此可以被其他worker所调度执行。类似的，所有这个机器上正在处理的map任务或者reduce任务都被设置成为idle状态，可以被其他worker所重新执行。

在失效机器上的已经完成的map任务还需要再次重新执行，这是因为中间结果存放在这个失效的机器上，所以导致中间结果无法访问。已经完成的recude任务无需再次执行，因为他们的结果已经保存在全局的文件系统中了。

当map任务首先由Aworker执行，随后被Bworker执行的时候（因为A失效了），所有执行reduce

任务的worker都会被通知。所有还没有来得及从A上读取数据的worker都会从B上读取数据。

MapReduce可以有效地支持到很大尺度的worker失效的情况。比如，在一个MapReduce操作中，在一个网络例行维护中，可能会导致每次大约有80台机器在几分钟之内不能访问。MapReduce的master制式简单的把这些不能访问的worker上的工作再执行一次，并且继续调度进程，最后完成MapReduce的操作。

Master失效

在master中，定期会设定checkpoint，写出master的数据结构。如果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。不过，由于只有一个master在运行，所以他如果失效就比较麻烦，因此我们当前的实现上，是如果master失效了，就终止MapReduce执行。客户端可以检测这种失效并且如果需要就重新尝试MapReduce操作。

失效的处理设计

当用户提供的map和reduce函数对于他们的输入来说是确定性的函数，我们的分布式的输出就应当和在一个整个程序没有失败的连续执行相同。

我们依靠对map和reduce任务的输出进行原子提交来完成这样的可靠性。每一个in-progress任务把输出写道一个私有的临时文件中。reduce任务产生一个这样的文件，map任务产生R个这样的任务（每一个对应一个reduce任务）。当一个map任务完成的时候，worker发送一个消息给master，并且这个消息中包含了这个R临时文件的名字。如果master又收到一个已经完成的map任务的完成消息，他就忽略这个消息。否则，他就在master数据结构中记录这个R文件。

当一个reduce任务完成的时候，reduceworker自动把临时输出的文件名改为正式的输出文件。如果再多台机器上有相同的reduce任务执行，那么就会有多个针对最终输出文件的更名动作。我们依靠文件系统提供的原子操作’改名字’，来保证最终的文件系统状态中记录的是其中一个reduce任务的输出。

我们的绝大部分map和reduce操作都是确定性的，实际上在语义角度，这个map和reduce并发执行和顺序执行市一样的，这就使得程序员很容易推测程序行为。当map和reduce操作是非确定性的时候，我们有稍弱的但是依旧是有道理的错误处理机制。对于非确定性操作来说，特定reduce任务R1的输出，与，非确定性的顺序执行的程序对R1的输出是等价的。另外，另一个reduce任务R2

的输出，是和另一个顺序执行的非确定性程序对应的R2输出相关的。

考虑map任务M和reduce任务R1，R2。我们设定e(Ri)为已经提交的Ri执行（有且仅有一个这样的执行）。当e(R1)处理得是M的一次执行，而e(R2)是处理M的另一次执行的时候，那么就会导致稍弱的失效处理了。

3.4存储位置

在我们的环境下，网络带宽资源是相对缺乏的。我们用尽量让输入数据保存在构成集群机器的本地硬盘上（通过GFS管理[8]）的方式来减少网络带宽的开销。GFS把文件分成64M一块，并且每一块都有几个拷贝（通常是3个拷贝），分布到不同的机器上。MapReduce的master有输入文件组的位置信息，并且尝试分派map任务在对应包含了相关输入数据块的设备上执行。如果不能分配map

任务到对应其输入数据的机器上执行，他就尝试分配map任务到尽量靠近这个任务的输入数据库的机器上执行（比如，分配到一个和包含输入数据块在一个switch网段的worker机器上执行）。当在一个足够大的cluster集群上运行大型MapReduce操作的时候，大部分输入数据都是在本地机器读取的，他们消耗比较少的网络带宽。

3.5任务颗粒度

如果上边我们讲的，我们把map阶段拆分到M小块，并且reduce阶段拆分到R小块执行。在理想状态下，M和R应当比worker机器数量要多得多。每一个worker机器都通过执行大量的任务来提高动态的负载均衡能力，并且能够加快故障恢复的速度：这个失效机器上执行的大量map任务都可以分布到所有其他worker机器上执行。

但是我们的实现中，实际上对于M和R的取值有一定的限制，因为master必须执行O(M+R)次调度，并且在内存中保存O(M\*R)个状态。（对影响内存使用的因素还是比较小的：O(M\*R)块状态，大概每对map任务/reduce任务1个字节就可以了）

进一步来说，用户通常会指定R的值，因为每一个reduce任务最终都是一个独立的输出文件。在实际中，我们倾向于调整M的值，使得每一个独立任务都是处理大约16M到64M的输入数据（这样，上面描写的本地优化策略会最有效），另外，我们使R比较小，这样使得R占用不多的worker机器。我们通常会用这样的比例来执行MapReduce:M=200，000，R=5，000，使用2，000台worker机器。

3.6备用任务

通常情况下，一个MapReduce的总执行时间会受到最后的几个”拖后腿”的任务影响：在计算过程中，会有一个机器过了比正常执行时间长得多的时间还没有执行完map或者reduce任务，导致

MapReduce总任务不能按时完成。出现拖后腿的情况有很多原因。比如：一个机器的硬盘有点问题，经常需要反复读取纠错，然后把读取输入数据的性能从30M/s降低到1M/s。cluster调度系统已经在某台机器上调度了其他的任务，所以因为CPU/内存/本地硬盘/网络带宽等竞争的关系，导致执行MapReduce的代码性能比较慢。我们最近出现的一个问题是机器的启动代码有问题，导致关闭了cpu的cache：在这些机器上的任务性能有上百倍的影响。

我们有一个通用的机制来减少拖后腿的情况。当MapReduce操作接近完成的时候，master调度备用进程来执行那些剩下的in-progress状态的任务。无论当最初的任务还是backup任务执行完成的时候，都把这个任务标记成为已经完成。我们调优了这个机制，通常只会占用多几个百分点的机器资源。但是我们发现这样做以后对于减少超大MapReduce操作的总处理时间来说非常有效。例如，在5.3

节描述的排序任务，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

虽然简单写map和reduce函数实现基本功能就已经对大部分需要都足够了，我们还是开发了一些有用的扩展，这些在本节详细描述。

4.1分区函数

MapReduce的使用者通过指定（R）来给出reduce任务/输出文件的数量。他们处理的数据在这些任务上通过对中间结果key得分区函数来进行分区。缺省的分区函数时使用hash函数（例如hash(key)modR）。这一般就可以得到分散均匀的分区。不过，在某些情况下，对key用其他的函数进行分区可能更有用。比如，某些情况下key是URL，那么我们希望所有对单个host的入口URL都保存在相同的输出文件。为了支持类似的情况，MapReduce函数库可以让用户提供一个特定的分

区函数。比如使用hash(hostname(urlkey))modR作为分区函数，这样可以让指向同一个hostname

的URL分配到相同的输出文件中。

4.2顺序保证

我们确保在给定的分区中，中间键值对key/value的处理顺序是根据key增量处理的。这样的顺序保证可以很容易生成每一个分区有序的输出文件，这对于输出文件格式需要支持客户端的对key的随机存取的时候就很有用，或者对输出数据集再作排序就很容易。

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。比如2.1节的一个统计单词出现次数的例子。由于word的频率趋势符合Zipf分布（齐夫分布），每一个map任务都回产生成百上千的<the,1>这样格式的记录。所有这些记录都通过网络发

送给一个单个的reduce任务，通过reduce函数进行相加，最后产生单个数字。我们允许用户指定一

个可选的组合函数Combiner函数，先在本地进行合并以下，然后再通过网络发送。

Combiner函数在每一个map任务的机器上执行。通常这个combiner函数的代码和reduce的代码实现上都是一样的。reduce函数和combiner函数唯一的不同就是MapReduce对于这两个函数的输出处理上不同。对于reduce函数的输出是直接写到最终的输出文件。对于combiner函数来说，输出是写到中间文件，并且会被发送到reduce任务中去。

部分使用combiner函数可以显著提高某些类型的MapReduce操作。附录A有这样的使用combiner

的例子。

4.4输入和输出类型

MapReduce函数库提供了读取几种不同格式的输入的支持。例如，”text”模式下，每行输入都被看成一个key/value对：key是在文件的偏移量，value是行的内容。另一个宠用格式保存了根据key进行排序key/value对的顺序。每一个输入类型的实现都知道如何把输入为了分别得map任务而进行有效分隔（比如，text模式下的分隔就是要确保分隔的边界只能按照行来进行分隔）。用户可以通过简单

的提供reader接口来进行新的输入类型的支持。不过大部分用户都只用一小部分预先定义的输入类型。

reader函数不需要提供从文件读取数据。例如，我们很容易定义一个reader函数从数据库读取数据，或者从保存在内存中的数据结构中读取数据。

类似的，我们提供了一组用于输出的类型，可以产生不同格式的数据，并且用户也可以很简单的增加新的输出类型。

4.5边界效应

在某些情况下，MapReduce的使用上，如果再map操作或者reduce操作时，增加辅助的输出文件，会比较有用。我们依靠程序来提供这样的边界原子操作。通常应用程序写一个临时文件并且用系统的原子操作：改名字操作，来再这个文件写完的时候，一次把这个文件改名改掉。

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，对于产生多个输出文件的，对于跨文件有一致性要求的任务，都必须是确定性的任务。这个限制到现在为止还没有真正在实际中遇到过。

4.6跳过损坏的记录

某些情况下，用户程序的代码会让map或者reduce函数在处理某些记录的时候crash掉。这种情况下MapReduce操作就不能完成。一般的做法是改掉bug然后再执行，但是有时候这种先改掉bug

的方式不太可行；也许是因为bug是在第三方的lib里边，它的原代码不存在等等。并且，很多时候，忽略一些记录不处理也是可以接受的，比如，在一个大数据集上进行统计分析的时候，就可以忽略有问题的少量记录。我们提供了一种执行模式，在这种执行模式下，MapReduce会检测到哪些记录会导致确定的crash，并且跳过这些记录不处理，使得整个处理能继续进行。

每一个worker处理进程都有一个signalhandler，可以捕获内存段异常和总线错误。在执行用户map或者reduce操作之前，MapReduce函数库通过全局变量保存记录序号。如果用户代码产生了这个信号，signalhandler于是用”最后一口气”通过UDP包向master发送上次处理的最后一条记录的序号。当master看到在这个特定记录上，有不止一个失效的时候，他就标志着条记录需要被跳过，，并且在下次重新执行相关的Map或者Reduce任务的时候跳过这条记录。

4.7本地执行

因为实际执行操作时分布在系统中执行的，通常是在好几千台计算机上执行得，并且是由master机器进行动态调度的任务，所以对map和reduce函数的调试就比较麻烦。为了能够让调试方便，profiling和小规模测试，我们开发了一套MapReduce的本地实现，也就是说，MapReduce函数库在本地机器上顺序执行所有的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。用户可以通过设定特别的标志来执行他们的程序，同时也可以很容易的使用调试和测试工具（比

如gdb）等等。4.8状态信息

master内部有一个HTTP服务器，并且可以输出状态报告。状态页提供了计算的进度报告，比如有多少任务已经完成，有多少任务正在处理，输入的字节数，中间数据的字节数，输出的字节数，处理

百分比，等等。这些页面也包括了指向每个任务输出的标准错误和输出的标准文件的连接。用户可以根据这些数据来预测计算需要大约执行多长时间，是否需要为这个计算增加额外的计算资源。这些页面也可以用来分析为何计算执行的会比预期的慢。

|  |  |  |
| --- | --- | --- |
| 此外，最上层的状态页面也显示了哪些任务。这些信息对于调试用户代码中的 | worker失效了，以及他们失效的时候上面运行的  bug很有帮助。 | map和reduce |
| 4.9计数器 |  |  |

MapReduce函数库提供了用于统计不同事件发生次数的计数器。比如，用户可能想统计所有已经索引的German文档数量或者已经处理了多少单词的数量，等等。

为了使用这样的特性，用户代码创建一个叫做counter的对象，并且在map和reduce函数中在适当的时候增加counter的值。例如：

Counter\*uppercase;

uppercase=GetCounter("uppercase");

map(Stringname,Stringcontents):foreachwordwincontents:

if(IsCapitalized(w)):uppercase->Increment();

EmitIntermediate(w,"1");

这些counter的值，会定时从各个单独的worker机器上传递给master（通过ping的应答包传递）。master把执行成功的map或者reduce任务的counter值进行累计，并且当MapReduce操作完成之后，返回给用户代码。当前counter值也会显示在master的状态页面，这样人可以看到计算现场的进度。当累计counter的值的时候，master会检查是否有对同一个map或者reduce任务的相同累计，避免累计重复。（backup任务或者机器失效导致的重新执行map任务或者reduce任务或导致这个

counter重复执行，所以需要检查，避免master进行重复统计）。

部分计数器的值是由MapReduce函数库进行自动维持的，比如已经处理的输入的key/value对的数量，或者输出的key/value键值对等等。

counter特性对于MapReduce操作的完整性检查非常有用。比如，在某些MapReduce操作中，用户程序需要确保输出的键值对精确的等于处理的输入键值对，或者处理得German文档数量是在处理的整个文档数量中属于合理范围内。

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。

这两个程序代表了大量的用MapReduce实现的真实的程序的主要类型-一类是对数据进行洗牌，另一类是从海量数据集中抽取少部分的关心的数据。

5.1集群配置

所有这些程序都是运行在一个大约有1800台机器的集群上。每台机器配置2个2GIntelXeon支持超线程的处理器，4GB内存，两个160GBIDE硬盘，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，在最上层大概有100-200G的聚合贷款。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内存里，大概有1-1.5G用于运行在集群上的其他任务。这个程序是在周末下午执行的，这时候的CPU，磁盘和网络基本上属于空闲状态。

* 1. GREP

grep程序需要扫描大概10的10次方个由100个字节组成的记录，查找比较少见的3个字符的查找串（这个查找串在92，337个记录中存在）。输入的记录被拆分成大约64M一个的块（M=15000），整个输出方在一个文件中（R=1）。

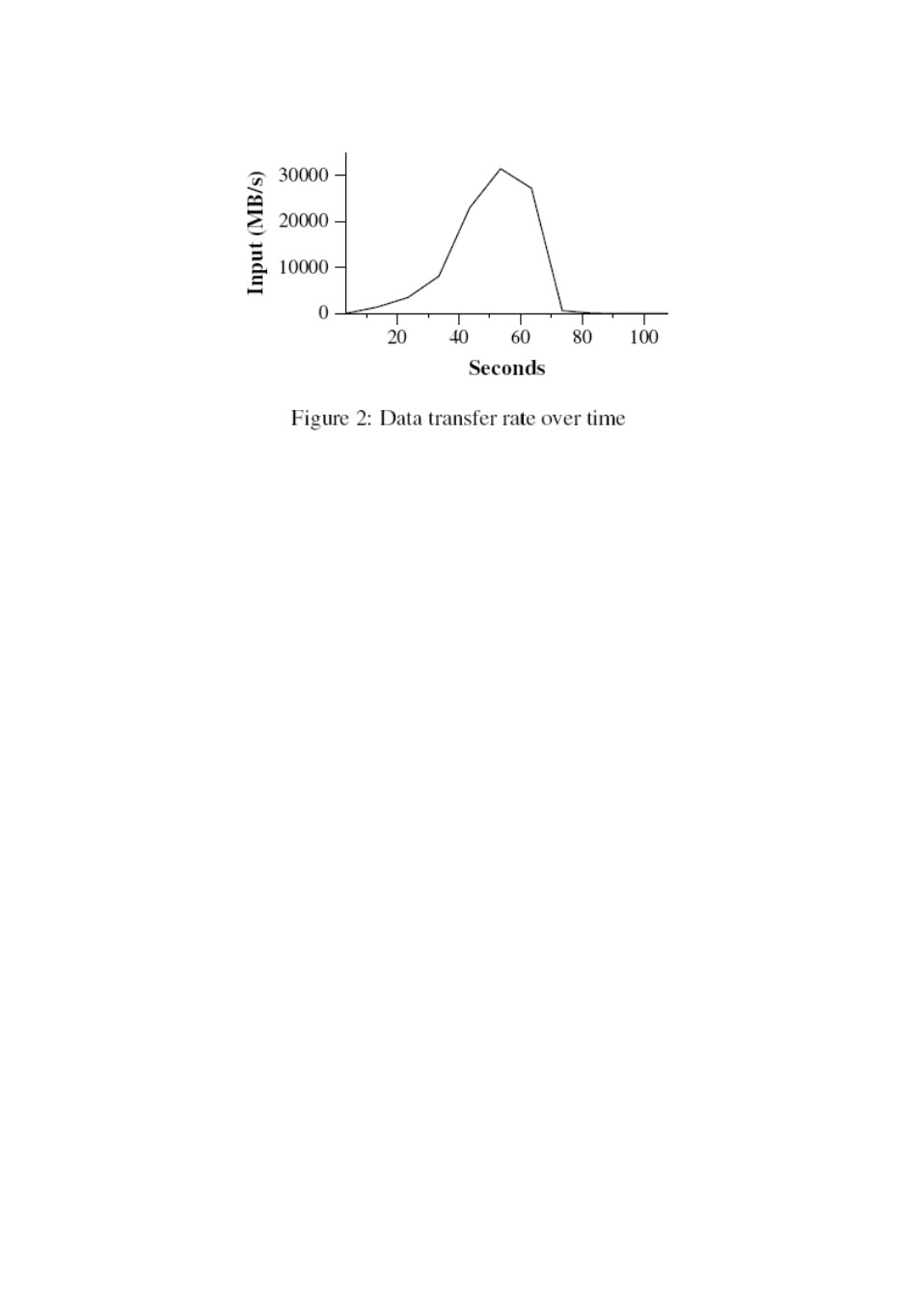


图2表示了这个程序随时间的处理过程。Y轴是输入数据的处理速度。处理速度逐渐随着参与MapReduce计算的机器增加而增加，当1764台worker开始工作的时候，达到了30G/s的速度。当map任务结束的时候，在计算开始后80秒，输入的速度降到0。整个计算过程从开始到结束一共花了大概150秒。这包括了大约一分钟的开头启动部分。开头的部分是用来把这个程序传播到各个

worker机器上的时间，并且等待GFS系统打开100个输入文件集合并且获得相关的文件位置优化信息。

* 1. SORT排序

SORT程序排序10的10次方个100个字节组成的记录（大概1TB的数据）。这个程序是仿制TeraSortbenchmark[10]的。

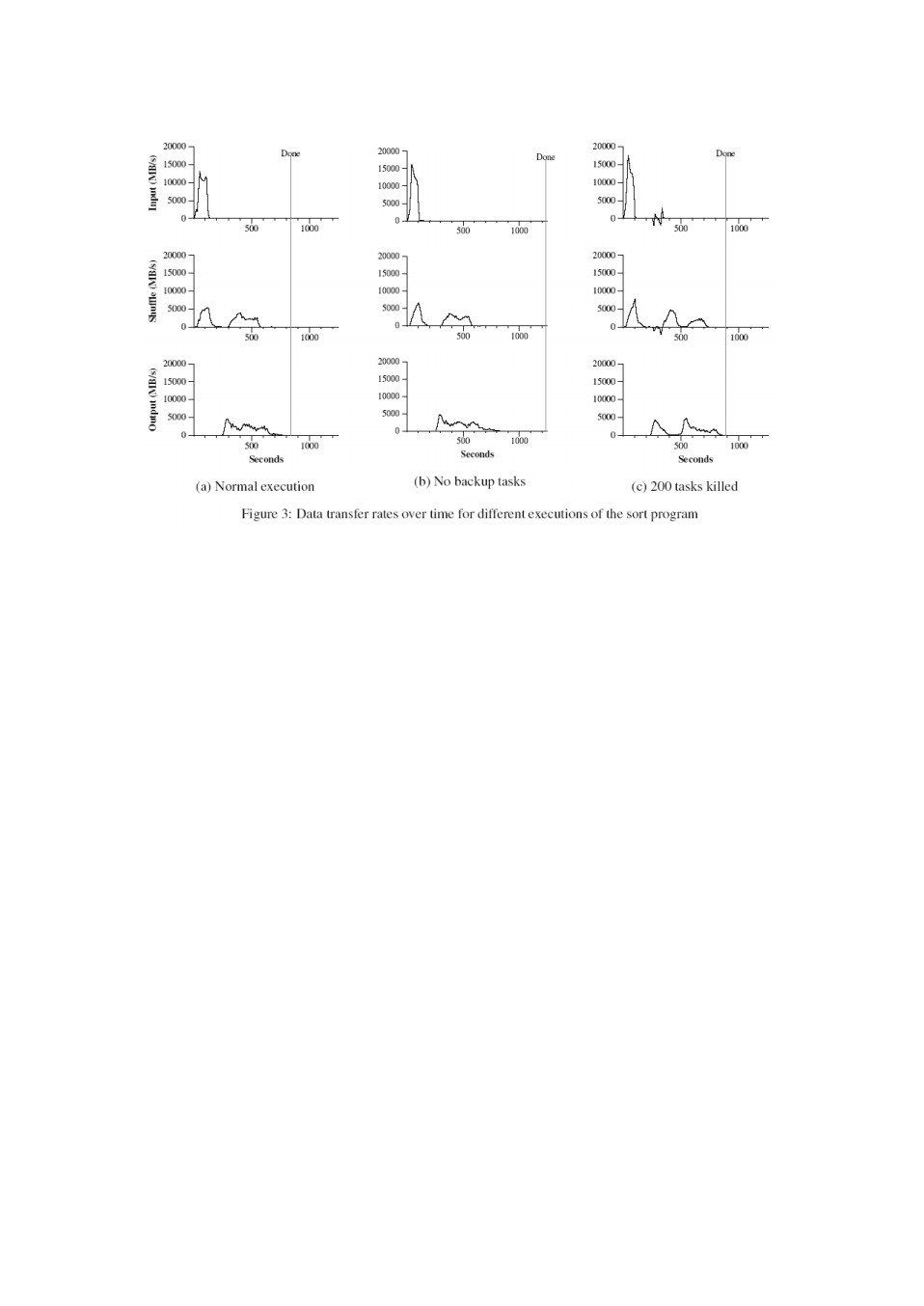
sort程序是由不到50行用户代码组成。三行的map函数从文本行中解出10个字节的排序key，并且把这个key和原始行作为中间结果key/value键值对输出。我们使用了一个内嵌的identitiy函数作为reduce的操作。这个函数把中间结果key/value键值对不变的作为输出的key/value键值对。最终排序输出写到一个两路复制的GFS文件中（就是说，程序的输出会写2TB的数据）。

就像前边讲的，输入数据分成64MB每块（M=15000）。我们把排序后的输出分区成为4000个文件

（R=4000）。分区函数使用key的原始字节来吧数据分区到R个小块中。

我们这个benchmark中的分区函数自身知道key的分区情况。通常对于排序程序来说，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得分区点。

第11页



图三是这个排序程序的正常执行过程。左上的图表示了输入数据读取的速度。数据读取速度会达到

13G/s，并且在不到200秒所有map任务完成之后迅速滑落到0。我们注意到数据读取速度小于grep粒子。这是因为排序map任务划了大概一半时间和I/O带宽写入中间输出到本地硬盘。相对应的grep中间结果输出几乎可以忽略不计。

左边中间的图是map任务把中间数据发送到reduce任务的网络速度。这个排序过程自从第一个任务完成之后就开始了。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分布到大概1700台机器上，每台机器一次大概执行1个reduce任务）。大概计算开始300秒以后，这些第一批reduce任务完成了，并且我们开始执行剩下的reduce任务。所有这些排序任务会在计算开始后大概600秒结束。

左下的图表示reduce任务把排序后的数据写到最终的输出文件的速度。在第一个排序期结束后到写盘开始之前有一个小延时，这是因为机器正在忙于内部排序中间数据。写盘速度持续大概2-4G/s。在计算开始后大概850秒左右写盘完成。包括启动部分，整个计算用了891秒。这个和TeraSortbenchmark[18]的最高纪录1057秒差不多。

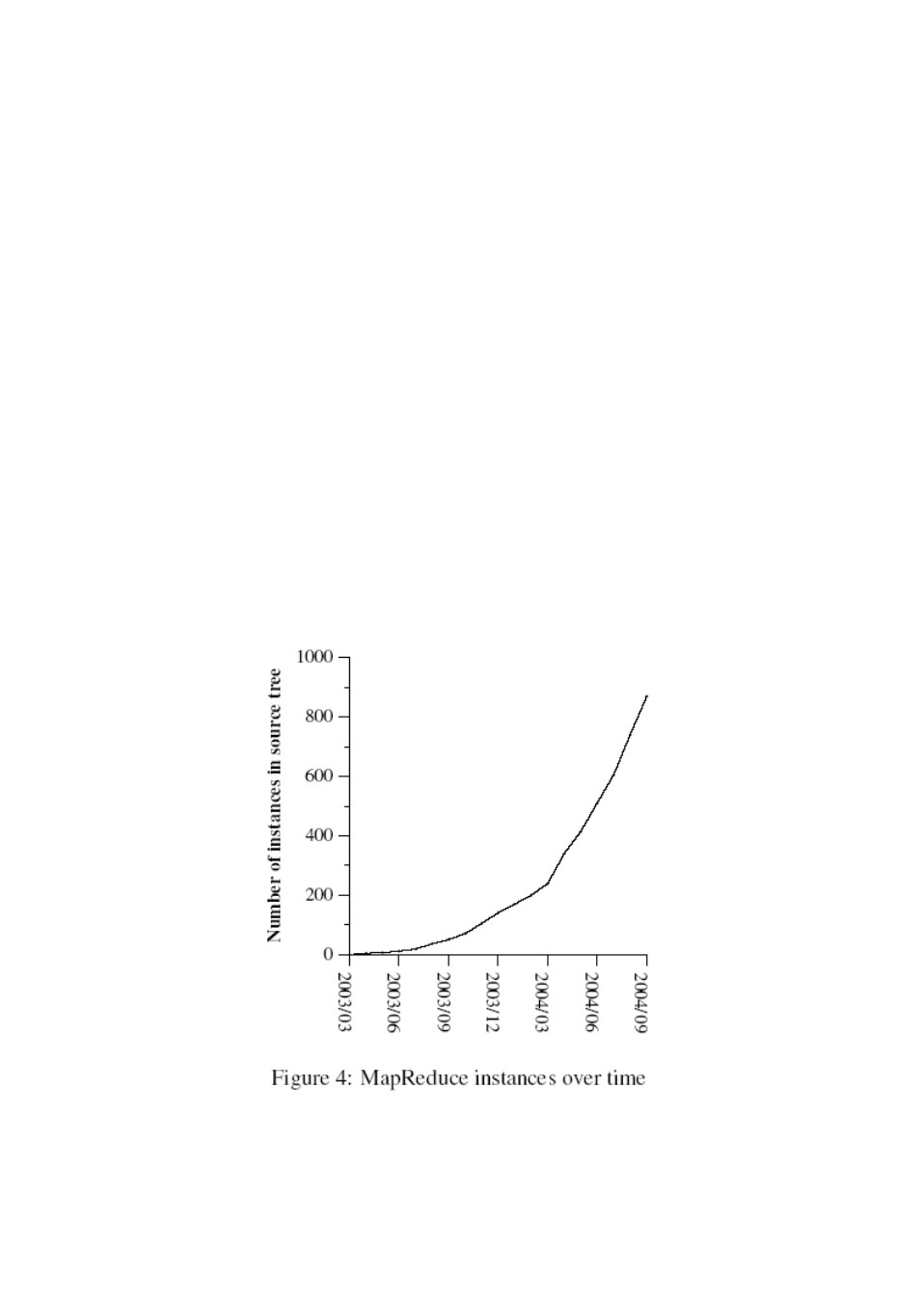
需要注意的事情是：输入速度要比排序速度和输出速度快，这是因为我们本地化的优化策略，绝大部分数据都是从本地硬盘读取而上去了我们相关的网络消耗。排序速度比输出速度快，这是因为输出阶段写了两份排序后的速度（我们写两份的原因是为了可靠性可可用性的原因）。我们写两份的原因是因为底层文件系统的可靠性和可用性的要求。如果底层文件系统用类似容错编码[14](erasurecoding)的方式，而不采用复制写的方式，在写盘阶段可以降低网络带宽的要求。

* 1. 高效的backup任务

在图三（b），是我们在关闭掉backup任务的时候，sort程序的执行情况。执行流和上边讲述的图3

（a）很类似，但是这个关闭掉backup任务的时候，执行的尾巴很长，并且执行的尾巴没有什么有效的写盘动作。在960秒以后，除了5个reduce以外，其他reduce任务都已经完成。不过这些拖后腿的任务又执行了300秒才完成。整个计算化了1283秒，多了44%的执行时间。

第12页



5.5失效的机器

在图三（c）中，我们演示了在sort程序执行过程中故意暂时杀掉1746个worker中的200个worker进程的执行情况。底层的集群调度立刻在这些机器上重新创建了新的worker处理（因为我们只是把这些机器上的处理进程杀掉，而机器依旧是可以操作的）。

因为已经完成的mapwork丢失了（由于相关的mapworker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1月写了第一个版本的MapReduce函数库，并且在2003年8月作了显著的增强，包括了本地优化，worker机器之间的动态负载均衡等等。自那以后，MapReduce函数库就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：

。大尺度的计算机学习问题。

。GoogleNews和Froogle产品的集群问题。

。从公众查询产品（比如Google的Zeitgeist）的报告中抽取数据。

。从web网页作新试验和抽取新的产品（例如，从大量的webpage中的本地查找抽取物理位置信息）。

。大尺度的图型计算。

第13页

任务数平均任务完成时间使用的机器时间读取的输入数据产生的中间数据写出的输出数据

每个job平均worker机器数每个job平均死掉work数每个job平均map任务每个job平均reduce任务map唯一实现

reduce的唯一实现

map/reduce的combiner实现

表1：MapReduce2004年8月的执行情况

29423

634秒

79,186天

3,288TB

758TB

193TB

157

1.2

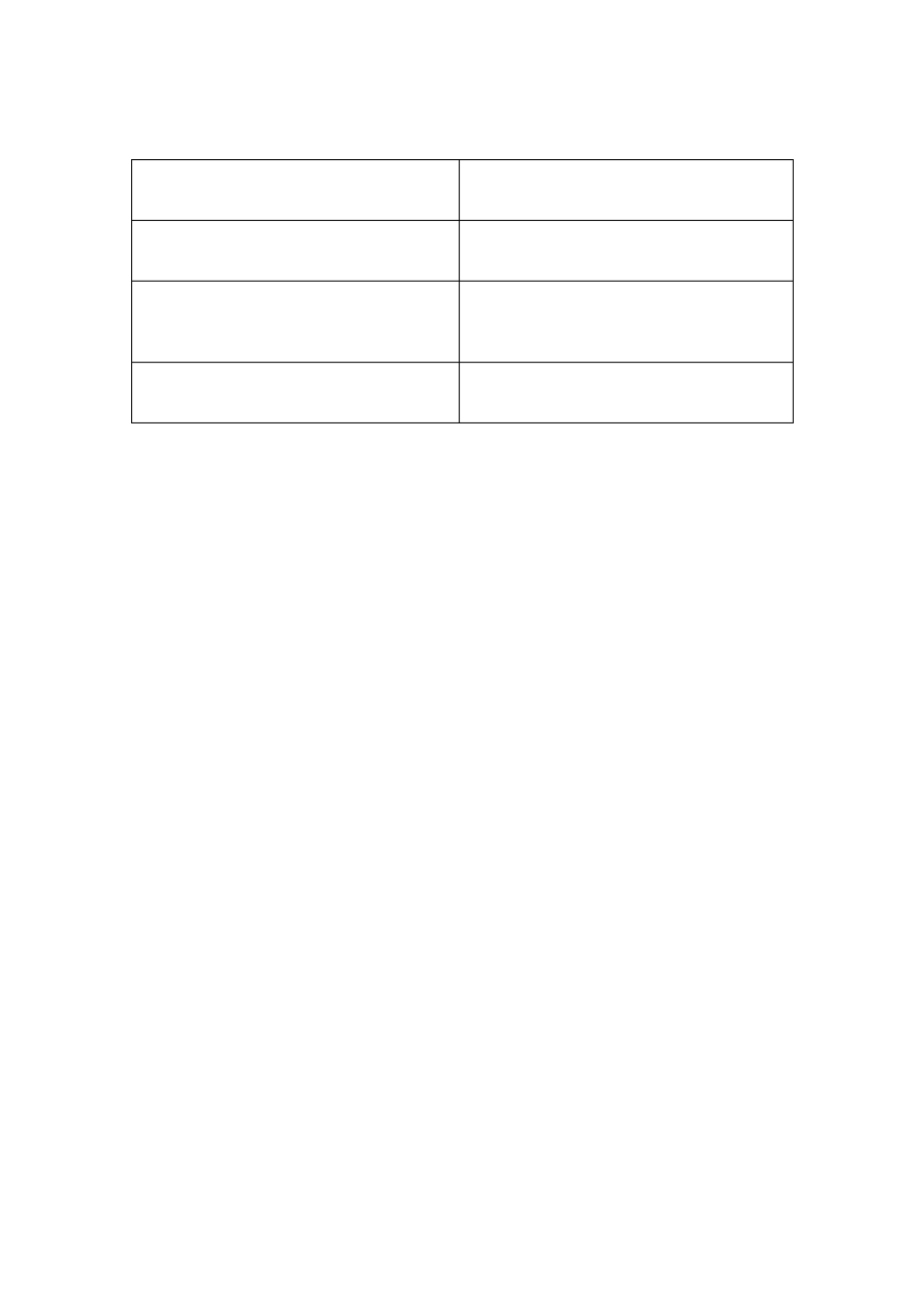
3,351

55

395

296

426



图四显示了我们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，能够很容易的开发处理海量数据的程序。

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状态。在表1，我们列出了2004

年8月份MapReduce运行的任务所占用的相关资源。

6.1大尺度的索引

到目前为止，最成功的MapReduce的应用就是重写了Googleweb搜索服务所使用到的index系统。索引系统处理蠕虫系统抓回来的超大量的数据，这些数据保存在GFS文件里。普通这些文档的大小是超过了20TB的数据。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：

索引代码很简单，很小，很容易理解。因为对于容错的处理代码，分布以及并行处理代码都通过MapReduce函数库封装了，所以索引代码很简单，很小，很容易理解。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处理次数。这使得我们容易改变索引处理方式。比如，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，以及网络风暴都已经由MapReduce自己解决了，而不需要操作人员的交互。此外，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编程模式，并且通过对编程的严格限制来实现自动的并行计算。例如，一个结合函数可以在一个N个元素的所有前缀上进行计算，并且使用并发前缀计算，会在在N个并发节

第14页

点上会耗费logN的时间[6，9，13]。MapReduce是这些模式下的，一个我们基于超大系统的现实经验的一个简化和精炼。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操心的。

BulkSynchronousProgramming[17]以及一些MPIprimitives[11]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系统的不同之处在于，MapReduce是通过限制性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activedisks[12,15]等技术的影响的，在activedisks中，计算任务是尽量推送到数据在本地磁盘的节点处理，这样就减少了网络系统的I/O吞吐。我们是在直接附带几个硬盘的通机器上执行我们的计算工作，不是在磁盘处理器上执行我们的工作，但是总的效果是一样的。

我们的backuptask机制和早先CharlotteSystem[3]的机制比较类似。早先的简单调度的一个缺点是如果一个任务导致反复失效，那么整个计算就不能完成。我们通过在故障情况下跳过故障记录的方式，在某种程度上解决了这个问题。

MapReduce的实现依赖于一个内部的集群管理系统，这个集群管理系统负责在一个超大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，集群管理系统在理念上和Condor[16]等其他系统一样。

MapReduce函数库的排序部分和NOW-Sort[1]的操作上很类似。源机器（mapworkers）把待排序的数据进行分区，并且发送到R个reduceworker中的一个进行处理。每一个reduceworker作本地排序（尽可能在内存排序）。当然NOW-Sort没有刻意用户定义的Map和Reduce函数，而我们的函数库有，所以我们的函数库可以有很高的适应性。

River[2]提供了一个编程模式，在这样的编程模式下，处理进程可以通过分布式查询来互相传送数据的方式进行通讯。和MapReduce类似，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能。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的框架是通过限制性编程模式，来把问题分解成为大量的任务。每一个任务都是动态调度到可用的worker上执行，这样快速的worker可以执行更多的任务。限制性编程模式同样允许我们在接近计算完成的时候调度backup任务，在出现处理不均匀的情况下，大量的缩小整个完成的时间（比如在有慢机或者阻塞的worker的时候）。

BAD-FS[5]和MapReduce的编程模式完全不同，它不像MapReduce是基于很大的网络计算的。不过，这两个系统有两个基本原理很类似。（1）两个系统都使用重复执行来防止由于失效导致的数据丢失。（2）两个都使用数据本地化调度策略，使得处理尽可能在本地数据上进行，减少通过网络通讯的数据量。

TACC[7]是一个用于简单构造高可用性网络服务的系统。就像MapReduce，它依靠重新执行机制来实现的容错处理。

8结束语

MapReduce的编程模式在Google成功应用于许多方面。我们把这种成功应用归结为几个方面：首先，这个编程模式易于使用，即使程序员没有并行或者分布式系统经验，由于MapReduce封装了并行的细节和容错处理，本地化计算，负载均衡等等，所以，使得编程非常容易。其次，大量不同的问

题都可以简单通过MapReduce来解决。例如，MapReduce用于产生Google的web搜索服务所需要的数据，用来排序，用来数据挖掘，用于机器智能学习，以及很多其他系统。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，并且因此也适用于解决Google遇到的其他很多需要大量计算的问题。

我们也从MapReduce上学到了不少内容。首先，先执行编程模式使得并行和分布式计算非常容易，并且也易于构造这样的容错计算环境。其次，网络带宽是系统的资源的瓶颈。我们系统的一系列优化都使因此针对减少网络传输量为目的的：本地优化使得我们读取数据时，是从本地磁盘读取的，并且写出单个中间数据文件到本地磁盘也节约了网络带宽。第三，冗余执行可以减少慢机器带来的影响，并且解决由于机器失效导致的数据丢失问题。

9感谢

JoshLevenberg校定和扩展了用户级别的MapReduceAPI，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使用Google文件系统GFS[8]来作为数据和输出。我们还感谢PercyLiangOlcanSercinoglu在开发用于MapReduce的集群管理系统得工作。MikeBurrows,WilsonHsieh,JoshLevenberg,SharonPerl,RobPike,DebbyWallach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EricBrewer，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，以及建议，以及错误报告等等。

10参考资料

[1]AndreaC.Arpaci-Dusseau,RemziH.Arpaci-Dusseau,DavidE.Culler,JosephM.Hellerstein,andDavidA.Patterson.High-performancesortingonnetworksofworkstations.InProceedingsofthe1997ACMSIGMODInternationalConferenceonManagementofData,Tucson,Arizona,May1997.

[2]RemziH.Arpaci-Dusseau,EricAnderson,NoahTreuhaft,DavidE.Culler,JosephM.Hellerstein,DavidPatterson,andKathyYelick.ClusterI/OwithRiver:Makingthefastcasecommon.InProceedingsoftheSixthWorkshoponInput/OutputinParallelandDistributedSystems(IOPADS'99),pages10.22,Atlanta,Georgia,May1999.

[3]ArashBaratloo,MehmetKaraul,ZviKedem,andPeterWyckoff.Charlotte:Metacomputingontheweb.InProceedingsofthe9thInternationalConferenceonParallelandDistributedComputingSystems,1996.[4]LuizA.Barroso,JeffreyDean,and

UrsH¨olzle.Websearchforaplanet:TheGoogleclusterarchitecture.IEEEMicro,

23(2):22.28,April2003.

[5]JohnBent,DouglasThain,AndreaC.Arpaci-Dusseau,RemziH.Arpaci-Dusseau,and

MironLivny.Explicitcontrolinabatch-awaredistributedfilesystem.InProceedingsofthe1stUSENIXSymposiumonNetworkedSystemsDesignandImplementationNSDI,

March2004.

[6]GuyE.Blelloch.Scansasprimitiveparalleloperations.IEEETransactionsonComputers,C-38(11),November1989.

[7]ArmandoFox,StevenD.Gribble,YatinChawathe,EricA.Brewer,andPaulGauthier.

Cluster-basedscalablenetworkservices.InProceedingsofthe16thACMSymposiumonOperatingSystemPrinciples,pages78.91,Saint-Malo,France,1997.

[8]SanjayGhemawat,HowardGobioff,andShun-TakLeung.TheGooglefilesystem.In19thSymposiumonOperatingSystemsPrinciples,pages29.43,LakeGeorge,NewYork,2003.ToappearinOSDI200412

[9]S.Gorlatch.Systematicefficientparallelizationofscanandotherlisthomomorphisms.In

L.Bouge,P.Fraigniaud,A.Mignotte,andY.Robert,editors,Euro-Par'96.ParallelProcessing,LectureNotesinComputerScience1124,pages401.408.Springer-Verlag,1996.

[10]JimGray.Sortbenchmarkhomepage.[http://research.microsoft.com/barc/SortBenchmark/.](http://research.microsoft.com/barc/SortBenchmark/)

[11]WilliamGropp,EwingLusk,andAnthonySkjellum.UsingMPI:PortableParallelProgrammingwiththeMessage-PassingInterface.MITPress,Cambridge,MA,1999.

[12]L.Huston,R.Sukthankar,R.Wickremesinghe,M.Satyanarayanan,G.R.Ganger,E.Riedel,andA.Ailamaki.Diamond:Astoragearchitectureforearlydiscardininteractive

search.InProceedingsofthe2004USENIXFileandStorageTechnologiesFASTConference,April2004.

[13]RichardE.LadnerandMichaelJ.Fischer.Parallelprefixcomputation.Journalofthe

ACM,27(4):831.838,1980.[14]MichaelO.Rabin.Efficientdispersalofinformationforsecurity,loadbalancingandfaulttolerance.JournaloftheACM,36(2):335.348,1989.

[15]ErikRiedel,ChristosFaloutsos,GarthA.Gibson,andDavidNagle.Activedisksforlarge-scaledataprocessing.IEEEComputer,pages68.74,June2001.

[16]DouglasThain,ToddTannenbaum,andMironLivny.Distributedcomputinginpractice:TheCondorexperience.ConcurrencyandComputation:PracticeandExperience,2004.

[17]L.G.Valiant.Abridgingmodelforparallelcomputation.CommunicationsoftheACM,33(8):103.111,1997.

[18]JimWyllie.Spsort:Howtosortaterabytequickly.[http://alme1.almaden.ibm.com/cs/spsort.pdf.](http://alme1.almaden.ibm.com/cs/spsort.pdf)

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"

//User'smapfunction

classWordCounter:publicMapper{public:

virtualvoidMap(constMapInput&input){conststring&text=input.value();constintn=text.size();

for(inti=0;i<n;){

//Skippastleadingwhitespacewhile((i<n)&&isspace(text[i]))

i++;

//Findwordendintstart=i;

while((i<n)&&!isspace(text[i]))i++;

if(start<i)

Emit(text.substr(start,i-start),"1");

}

}

};

REGISTER\_MAPPER(WordCounter);

//User'sreducefunctionclassAdder:publicReducer{

virtualvoidReduce(ReduceInput\*input){

//Iterateoverallentrieswiththe

//samekeyandaddthevaluesint64value=0;

while(!input->done()){

value+=StringToInt(input->value());input->NextValue();

}

//Emitsumforinput->key()Emit(IntToString(value));

}

};

REGISTER\_REDUCER(Adder);

intmain(intargc,char\*\*argv){ParseCommandLineFlags(argc,argv);

MapReduceSpecificationspec;

//Storelistofinputfilesinto"spec"for(inti=1;i<argc;i++){

MapReduceInput\*input=spec.add\_input();

input->set\_format("text");input->set\_filepattern(argv[i]);

input->set\_mapper\_class("WordCounter");

}

//Specifytheoutputfiles:

///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");

//Optional:dopartialsumswithinmap

//taskstosavenetworkbandwidthout->set\_combiner\_class("Adder");

//Tuningparameters:useatmost2000

//machinesand100MBofmemorypertaskspec.set\_machines(2000);spec.set\_map\_megabytes(100);spec.set\_reduce\_megabytes(100);

//NowrunitMapReduceResultresult;

if(!MapReduce(spec,&result))abort();

//Done:'result'structurecontainsinfo

//aboutcounters,timetaken,numberof

//machinesused,etc.return0;

}