Google MapReduce 中文版[1]

摘要

MapReduce 是一个编程模型，也是一个处理和生成超大数据集的算法模型的相关实现。用户首先创建一个 Map 函数处理一个基于 key/value pair 的数据集合，输出中间的基于 key/value pair 的数据集合；然后再创建一个 Reduce 函数用来合并所有的具有相同中间 key 值的中间 value 值。现实世界中有很多满足上述处理模型的例子，本论文将详细描述这个模型。

MapReduce 架构的程序能够在大量的普通配置的计算机上实现并行化处理。这个系统在运行时只关心：如何分割输入数据，在大量计算机组成的集群上的调度，集群中计算机的错误处理，管理集群中计算机之间必要的通信。采用 MapReduce 架构可以使那些没有并行计算和分布式处理系统开发经验的程序员有效利用分布式系统的丰富资源。

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集群上：一个典型的 MapReduce 计算往往由几千台机器组成、处理以 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，在 Google 的集群上，每天都有 1000 多个 MapReduce 程序在执行。

1 介绍

在过去的 5 年里，包括本文作者在内的 Google 的很多程序员，为了处理海量的原始数据，已经实现了数以百计的、专用的计算方法。这些计算方法用来处理大量的原始数据，比如，文档抓取（类似网络爬虫的程序）、Web 请求日志等等；也为了计算处理各种类型的衍生数据，比如倒排索引、Web 文档的图结构的各种表示形势、每台主机上网络爬虫抓取的页面数量的汇总、每天被请求的最多的查询的集合等等。大多数这样的数据处理运算在概念上很容易理解。然而由于输入的数据量巨大，因此要想在可接受的时间内完成运算，只有将这些计算分布在成百上千的主机上。如何处理并行计算、如何分发数据、如何处理错误？所有这些问题综合在一起，需要大量的代码处理，因此也使得原本简单的运算变得难以处理。

为了解决上述复杂的问题，我们设计一个新的抽象模型，使用这个抽象模型，我们只要表述我们想要执行的简单运算即可，而不必关心并行计算、容错、数据分布、负载均衡等复杂的细节，这些问题都被封装在了一个库里面。设计这个抽象模型的灵感来自 Lisp 和许多其他函数式语言的 Map 和 Reduce 的原语。我们意识到我们大多数的运算都包含这样的操作：在输入数据的“逻辑”记录上应用 Map 操作得出一个中间 key/value pair 集合，然后在所有具有相同 key 值的 value 值上应用 Reduce 操作，从而达到合并中间的数据，得到一个

[1] 译者 alex，原文地址 http://blademaster.ixiezi.com/

想要的结果的目的。使用 MapReduce 模型，再结合用户实现的 Map 和 Reduce 函数，我们就可以非常容易的实现大规模并行化计算；通过 MapReduce 模型自带的“再次执行”（re-execution）功能，也提供了初级的容灾实现方案。

这个工作(实现一个 MapReduce 框架模型)的主要贡献是通过简单的接口来实现自动的并行化和大规模的分布式计算，通过使用 MapReduce 模型接口实现在大量普通的 PC 机上高性能计算。

第二部分描述基本的编程模型和一些使用案例。

第三部分描述了一个经过裁剪的、适合我们的基于集群的计算环境的 MapReduce 实现。

第四部分描述我们认为在 MapReduce 编程模型中一些实用的技巧。

第五部分对于各种不同的任务，测量我们 MapReduce 实现的性能。

第六部分揭示了在 Google 内部如何使用 MapReduce 作为基础重写我们的索引系统产品，包括其它一些使用 MapReduce 的经验。

第七部分讨论相关的和未来的工作。

2 编程模型

MapReduce 编程模型的原理是：利用一个输入 key/value pair 集合来产生一个输出的 key/value pair 集合。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 函数接受一个输入的 key/value pair 值，然后产生一个中间 key/value pair 值的集合。MapReduce 库把所有具有相同中间 key 值 I 的中间 value 值集合在一起后传递给 reduce 函数。

用户自定义的 Reduce 函数接受一个中间 key 的值 I 和相关的一个 value 值的集合。Reduce 函数合并这些 value 值，形成一个较小的 value 值的集合。一般的，每次 Reduce 函数调用只产生 0 或 1 个输出 value 值。通常我们通过一个迭代器把中间 value 值提供给 Reduce 函数，这样我们就可以处理无法全部放入内存中的大量的 value 值的集合。

2.1 例子

例如，计算一个大的文档集合中每个单词出现的次数，下面是伪代码段：

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, “1″);
```

```
reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

Map 函数输出文档中的每个词、以及这个词的出现次数(在这个简单的例子里就是 1)。Reduce 函数把 Map 函数产生的每一个特定的词的计数累加起来。

另外，用户编写代码，使用输入和输出文件的名字、可选的调节参数来完成一个符合 MapReduce 模型规范的对象，然后调用 MapReduce 函数，并把这个规范对象传递给它。用户的代码和 MapReduce 库链接在一起(用 C++实现)。附录 A 包含了这个实例的全部程序代码。

2.2 类型

尽管在前面例子的伪代码中使用了以字符串表示的输入输出值，但是在概念上，用户定义的 Map 和 Reduce 函数都有相关联的类型：

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

比如，输入的 key 和 value 值与输出的 key 和 value 值在类型上推导的域不同。此外，中间 key 和 value 值与输出 key 和 value 值在类型上推导的域相同。[2]

我们的 C++中使用字符串类型作为用户自定义函数的输入输出，用户在自己的代码中对字符串进行适当的类型转换。

2.3 更多的例子

这里还有一些有趣的简单例子，可以很容易的使用 MapReduce 模型来表示：

分布式的 Grep：Map 函数输出匹配某个模式的一行，Reduce 函数是一个恒等函数，即把中间数据复制到输出。

计算 URL 访问频率：Map 函数处理日志中 web 页面请求的记录，然后输出(URL,1)。Reduce 函数把相同

[2] 原文中这个 domain 的含义不是很清楚，我参考 Hadoop、KFS 等实现，map 和 reduce 都使用了泛型，因此，我把 domain 翻译成类型推导的域。

URL 的 value 值都累加起来，产生(URL,记录总数)结果。

倒转网络链接图：Map 函数在源页面（source）中搜索所有的链接目标（target）并输出为(target,source)。Reduce 函数把给定链接目标（target）的链接组合成一个列表，输出(target,list(source))。

每个主机的检索词向量：检索词向量用一个(词,频率)列表来概述出现在文档或文档集中的最重要的一些词。Map 函数为每一个输入文档输出(主机名,检索词向量)，其中主机名来自文档的 URL。Reduce 函数接收给定主机的所有文档的检索词向量，并把这些检索词向量加在一起，丢弃掉低频的检索词，输出一个最终的(主机名,检索词向量)。

倒排索引：Map 函数分析每个文档输出一个(词,文档号)的列表，Reduce 函数的输入是一个给定词的所有（词，文档号），排序所有的文档号，输出(词,list（文档号）)。所有的输出集合形成一个简单的倒排索引，它以一种简单的算法跟踪词在文档中的位置。

分布式排序：Map 函数从每个记录提取 key，输出(key,record)。Reduce 函数不改变任何的值。这个运算依赖分区机制(在 4.1 描述)和排序属性(在 4.2 描述)。

3 实现

MapReduce 模型可以有多种不同的实现方式。如何正确选择取决于具体的环境。例如，一种实现方式适用于小型的共享内存方式的机器，另外一种实现方式则适用于大型 NUMA 架构的多处理器的主机，而有的实现方式更适合大型的网络连接集群。

本章节描述一个适用于 Google 内部广泛使用的运算环境的实现：用以太网交换机连接、由普通 PC 机组成的大型集群。在我们的环境里包括：

1. x86 架构、运行 Linux 操作系统、双处理器、2-4GB 内存的机器。
2. 普通的网络硬件设备，每个机器的带宽为百兆或者千兆，但是远小于网络的平均带宽的一半。
3. 集群中包含成百上千的机器，因此，机器故障是常态。
4. 存储为廉价的内置 IDE 硬盘。一个内部分布式文件系统用来管理存储在这些磁盘上的数据。文件系统通过数据复制来在不可靠的硬件上保证数据的可靠性和有效性。
5. 用户提交工作（job）给调度系统。每个工作（job）都包含一系列的任务（task），调度系统将这些任务调度到集群中多台可用的机器上。

3.1 执行概括

通过将 Map 调用的输入数据自动分割为 M 个数据片段的集合，Map 调用被分布到多台机器上执行。输入的数据片段能够在不同的机器上并行处理。使用分区函数将 Map 调用产生的中间 key 值分成 R 个不同分区

（例如，hash(key) mod R），Reduce 调用也被分布到多台机器上执行。分区数量（R）和分区函数由用户来指定。

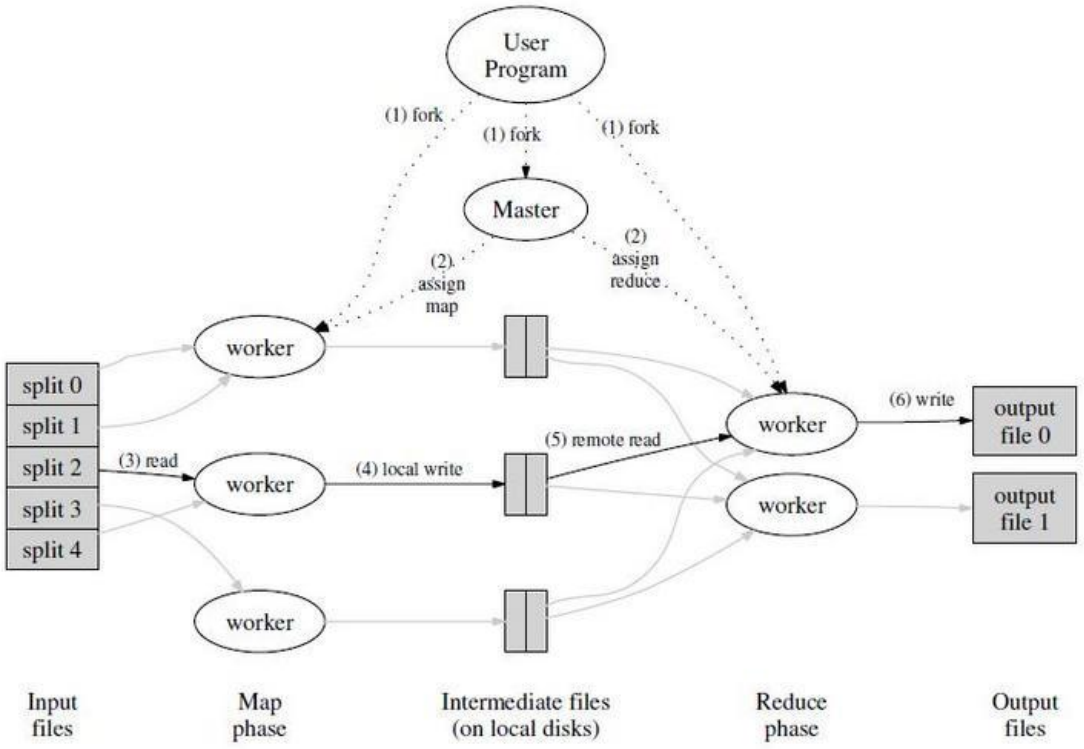

Figure 1: Execution overview

图 1 展示了我们的 MapReduce 实现中操作的全部流程。当用户调用 MapReduce 函数时，将发生下面的一系列动作（下面的序号和图 1 中的序号一一对应）：

1. 用户程序首先调用的 MapReduce 库将输入文件分成 M 个数据片度，每个数据片段的大小一般从 16MB 到 64MB(可以通过可选的参数来控制每个数据片段的大小)。然后用户程序在机群中创建大量的程序副本。
2. 这些程序副本中的有一个特殊的程序－master。副本中其它的程序都是 worker 程序，由 master 分配任务。有 M 个 Map 任务和 R 个 Reduce 任务将被分配，master 将一个 Map 任务或 Reduce 任务分配给一个空闲的 worker。
3. 被分配了 map 任务的 worker 程序读取相关的输入数据片段，从输入的数据片段中解析出 key/value pair，然后把 key/value pair 传递给用户自定义的 Map 函数，由 Map 函数生成并输出的中间 key/value pair，并缓存在内存中。
4. 缓存中的 key/value pair 通过分区函数分成 R 个区域，之后周期性的写入到本地磁盘上。缓存的 key/value pair 在本地磁盘上的存储位置将被回传给 master，由 master 负责把这些存储位置再传送给 Reduce worker。

5. 当 Reduce worker 程序接收到 master 程序发来的数据存储位置信息后，使用 RPC 从 Map worker 所在主机的磁盘上读取这些缓存数据。当 Reduce worker 读取了所有的中间数据后，通过对 key 进行排序后使得具有相同 key 值的数据聚合在一起。由于许多不同的 key 值会映射到相同的 Reduce 任务上，因此必须进行排序。如果中间数据太大无法在内存中完成排序，那么就要在外部进行排序。
6. Reduce worker 程序遍历排序后的中间数据，对于每一个唯一的中间 key 值，Reduce worker 程序将这个 key 值和它相关的中间 value 值的集合传递给用户自定义的 Reduce 函数。Reduce 函数的输出被追加到所属分区的输出文件。
7. 当所有的 Map 和 Reduce 任务都完成之后，master 唤醒用户程序。在这个时候，在用户程序里的对 MapReduce 调用才返回。

在成功完成任务之后，MapReduce 的输出存放在 R 个输出文件中（对应每个 Reduce 任务产生一个输出文件，文件名由用户指定）。一般情况下，用户不需要将这 R 个输出文件合并成一个文件－他们经常把这些文件作为另外一个 MapReduce 的输入，或者在另外一个可以处理多个分割文件的分布式应用中使用。

3.2 Master 数据结构

Master 持有一些数据结构，它存储每一个 Map 和 Reduce 任务的状态（空闲、工作中或完成），以及 Worker 机器(非空闲任务的机器)的标识。

Master 就像一个数据管道，中间文件存储区域的位置信息通过这个管道从 Map 传递到 Reduce。因此，对于每个已经完成的 Map 任务，master 存储了 Map 任务产生的 R 个中间文件存储区域的大小和位置。当 Map 任务完成时，Master 接收到位置和大小的更新信息，这些信息被逐步递增的推送给那些正在工作的 Reduce 任务。

3.3 容错

因为 MapReduce 库的设计初衷是使用由成百上千的机器组成的集群来处理超大规模的数据，所以，这个库必须要能很好的处理机器故障。

3.3.1 worker 故障

master 周期性的 ping 每个 worker。如果在一个约定的时间范围内没有收到 worker 返回的信息，master 将把这个 worker 标记为失效。所有由这个失效的 worker 完成的 Map 任务被重设为初始的空闲状态，之后这些任务就可以被安排给其他的 worker。同样的，worker 失效时正在运行的 Map 或 Reduce 任务也将被重新置为空闲状态，等待重新调度。

当 worker 故障时，由于已经完成的 Map 任务的输出存储在这台机器上，Map 任务的输出已不可访问了，

因此必须重新执行。而已经完成的 Reduce 任务的输出存储在全局文件系统上，因此不需要再次执行。

当一个 Map 任务首先被 worker A 执行，之后由于 worker A 失效了又被调度到 worker B 执行，这个“重新执行”的动作会被通知给所有执行 Reduce 任务的 worker。任何还没有从 worker A 读取数据的 Reduce 任务将从 worker B 读取数据。

MapReduce 可以处理大规模 worker 失效的情况。比如，在一个 MapReduce 操作执行期间，在正在运行的集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 只需要简单的再次执行那些不可访问的 worker 完成的工作，之后继续执行未完成的任务，直到最终完成这个 MapReduce 操作。

3.3.2 master 失败

一个简单的解决办法是让 master 周期性的将上面描述的数据结构（alex 注：指 3.2 节）的写入磁盘，即检查点（checkpoint）。如果这个 master 任务失效了，可以从最后一个检查点（checkpoint）开始启动另一个 master 进程。然而，由于只有一个 master 进程，master 失效后再恢复是比较麻烦的，因此我们现在的实现是如果 master 失效，就中止 MapReduce 运算。客户可以检查到这个状态，并且可以根据需要重新执行 MapReduce 操作。

3.3.3 在失效方面的处理机制

（alex 注：原文为”semantics in the presence of failures”）

当用户提供的 Map 和 Reduce 操作是输入确定性函数（即相同的输入产生相同的输出）时，我们的分布式实现在任何情况下的输出都和所有程序没有出现任何错误、顺序的执行产生的输出是一样的。

我们依赖对 Map 和 Reduce 任务的输出是原子提交的来完成这个特性。每个工作中的任务把它的输出写到私有的临时文件中。每个 Reduce 任务生成一个这样的文件，而每个 Map 任务则生成 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，worker 发送一个包含 R 个临时文件名的完成消息给 master。如果 master 从一个已经完成的 Map 任务再次接收到到一个完成消息，master 将忽略这个消息；否则，master 将这 R 个文件的名字记录在数据结构里。

当 Reduce 任务完成时，Reduce worker 进程以原子的方式把临时文件重命名为最终的输出文件。如果同一个 Reduce 任务在多台机器上执行，针对同一个最终的输出文件将有多个重命名操作执行。我们依赖底层文件系统提供的重命名操作的原子性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。

使用 MapReduce 模型的程序员可以很容易的理解他们程序的行为，因为我们绝大多数的 Map 和 Reduce 操作是确定性的，而且存在这样的一个事实：我们的失效处理机制等价于一个顺序的执行的操作。当 Map 或/和 Reduce 操作是不确定性的时候，我们提供虽然较弱但是依然合理的处理机制。当使用非确定操作的时候，一个 Reduce 任务 R1 的输出等价于一个非确定性程序顺序执行产生时的输出。但是，另一个 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 R2 的输出。

考虑 Map 任务 M 和 Reduce 任务 R1、R2 的情况。我们设定 e(Ri)是 Ri 已经提交的执行过程（有且仅有一个这样的执行过程）。当 e(R1)读取了由 M 一次执行产生的输出，而 e(R2)读取了由 M 的另一次执行产生的输出，导致了较弱的失效处理。

3.4 存储位置

在我们的计算运行环境中，网络带宽是一个相当匮乏的资源。我们通过尽量把输入数据(由 GFS 管理)存储在集群中机器的本地磁盘上来节省网络带宽。GFS 把每个文件按 64MB 一个 Block 分隔，每个 Block 保存在多台机器上，环境中就存放了多份拷贝(一般是 3 个拷贝)。MapReduce 的 master 在调度 Map 任务时会考虑输入文件的位置信息，尽量将一个 Map 任务调度在包含相关输入数据拷贝的机器上执行；如果上述努力失败了，master 将尝试在保存有输入数据拷贝的机器附近的机器上执行 Map 任务(例如，分配到一个和包含输入数据的机器在一个 switch 里的 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，大部分的输入数据都能从本地机器读取，因此消耗非常少的网络带宽。

3.5 任务粒度

如前所述，我们把 Map 拆分成了 M 个片段、把 Reduce 拆分成 R 个片段执行。理想情况下，M 和 R 应当比集群中 worker 的机器数量要多得多。在每台 worker 机器都执行大量的不同任务能够提高集群的动态的负载均衡能力，并且能够加快故障恢复的速度：失效机器上执行的大量 Map 任务都可以分布到所有其他的 worker 机器上去执行。

但是实际上，在我们的具体实现中对 M 和 R 的取值都有一定的客观限制，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）。

更进一步，R 值通常是由用户指定的，因为每个 Reduce 任务最终都会生成一个独立的输出文件。实际使用时我们也倾向于选择合适的 M 值，以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，上面描写的输入数据本地存储优化策略才最有效），另外，我们把 R 值设置为我们想使用的 worker 机器数量的小的倍数。我们通常会用这样的比例来执行 MapReduce：M=200000，R=5000，使用 2000 台 worker 机器。

3.6 备用任务

影响一个 MapReduce 的总执行时间最通常的因素是“落伍者”：在运算过程中，如果有一台机器花了很长的时间才完成最后几个 Map 或 Reduce 任务，导致 MapReduce 操作总的执行时间超过预期。出现“落伍者”的原因非常多。比如：如果一个机器的硬盘出了问题，在读取的时候要经常的进行读取纠错操作，导致读取

数据的速度从 30M/s 降低到 1M/s。如果 cluster 的调度系统在这台机器上又调度了其他的任务，由于 CPU、内存、本地硬盘和网络带宽等竞争因素的存在，导致执行 MapReduce 代码的执行效率更加缓慢。我们最近遇到的一个问题是由于机器的初始化代码有 bug，导致关闭了的处理器的缓存：在这些机器上执行任务的性能和正常情况相差上百倍。

我们有一个通用的机制来减少“落伍者”出现的情况。当一个 MapReduce 操作接近完成的时候，master 调度备用（backup）任务进程来执行剩下的、处于处理中状态（in-progress）的任务。无论是最初的执行进程、还是备用（backup）任务进程完成了任务，我们都把这个任务标记成为已经完成。我们调优了这个机制，通常只会占用比正常操作多几个百分点的计算资源。我们发现采用这样的机制对于减少超大 MapReduce 操作的总处理时间效果显著。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多花 44%的时间完成排序任务。

4 技巧

虽然简单的 Map 和 Reduce 函数提供的基本功能已经能够满足大部分的计算需要，我们还是发掘出了一些有价值的扩展功能。本节将描述这些扩展功能。

4.1 分区函数

MapReduce 的使用者通常会指定 Reduce 任务和 Reduce 任务输出文件的数量（R）。我们在中间 key 上使用分区函数来对数据进行分区，之后再输入到后续任务执行进程。一个缺省的分区函数是使用 hash 方法(比如，hash(key) mod R)进行分区。hash 方法能产生非常平衡的分区。然而，有的时候，其它的一些分区函数对 key 值进行的分区将非常有用。比如，输出的 key 值是 URLs，我们希望每个主机的所有条目保持在同一个输出文件中。为了支持类似的情况，MapReduce 库的用户需要提供专门的分区函数。例如，使用“hash(Hostname(urlkey)) mod R”作为分区函数就可以把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

我们确保在给定的分区中，中间 key/value pair 数据的处理顺序是按照 key 值增量顺序处理的。这样的顺序保证对每个分成生成一个有序的输出文件，这对于需要对输出文件按 key 值随机存取的应用非常有意义，对在排序输出的数据集也很有帮助。

4.3 Combiner 函数

在某些情况下，Map 函数产生的中间 key 值的重复数据会占很大的比重，并且，用户自定义的 Reduce 函数满足结合律和交换律。在 2.1 节的词数统计程序是个很好的例子。由于词频率倾向于一个 zipf 分布(齐夫分

布)，每个 Map 任务将产生成千上万个这样的记录<the,1>。所有的这些记录将通过网络被发送到一个单独的 Reduce 任务，然后由这个 Reduce 任务把所有这些记录累加起来产生一个数字。我们允许用户指定一个可选的 combiner 函数，combiner 函数首先在本地将这些记录进行一次合并，然后将合并的结果再通过网络发送出去。

Combiner 函数在每台执行 Map 任务的机器上都会被执行一次。一般情况下，Combiner 和 Reduce 函数是一样的。Combiner 函数和 Reduce 函数之间唯一的区别是 MapReduce 库怎样控制函数的输出。Reduce 函数的输出被保存在最终的输出文件里，而 Combiner 函数的输出被写到中间文件里，然后被发送给 Reduce 任务。

部分的合并中间结果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 输入和输出的类型

MapReduce 库支持几种不同的格式的输入数据。比如，文本模式的输入数据的每一行被视为是一个 key/value pair。key 是文件的偏移量，value 是那一行的内容。另外一种常见的格式是以 key 进行排序来存储的 key/value pair 的序列。每种输入类型的实现都必须能够把输入数据分割成数据片段，该数据片段能够由单独的 Map 任务来进行后续处理(例如，文本模式的范围分割必须确保仅仅在每行的边界进行范围分割)。虽然大多数 MapReduce 的使用者仅仅使用很少的预定义输入类型就满足要求了，但是使用者依然可以通过提供一个简单的 Reader 接口实现就能够支持一个新的输入类型。

Reader 并非一定要从文件中读取数据，比如，我们可以很容易的实现一个从数据库里读记录的 Reader，或者从内存中的数据结构读取数据的 Reader。

类似的，我们提供了一些预定义的输出数据的类型，通过这些预定义类型能够产生不同格式的数据。用户采用类似添加新的输入数据类型的方式增加新的输出类型。

4.5 副作用

在某些情况下，MapReduce 的使用者发现，如果在 Map 和/或 Reduce 操作过程中增加辅助的输出文件会比较省事。我们依靠程序 writer 把这种“副作用”变成原子的和幂等的[3]。通常应用程序首先把输出结果写到一个临时文件中，在输出全部数据之后，在使用系统级的原子操作 rename 重新命名这个临时文件。

如果一个任务产生了多个输出文件，我们没有提供类似两阶段提交的原子操作支持这种情况。因此，对于会产生多个输出文件、并且对于跨文件有一致性要求的任务，都必须是确定性的任务。但是在实际应用过程中，这个限制还没有给我们带来过麻烦。

[3] 幂等的指一个总是产生相同结果的数学运算

4.6 跳过损坏的记录

有时候，用户程序中的 bug 导致 Map 或者 Reduce 函数在处理某些记录的时候 crash 掉，MapReduce 操作无法顺利完成。惯常的做法是修复 bug 后再次执行 MapReduce 操作，但是，有时候找出这些 bug 并修复它们不是一件容易的事情；这些 bug 也许是在第三方库里边，而我们手头没有这些库的源代码。而且在很多时候，忽略一些有问题的记录也是可以接受的，比如在一个巨大的数据集上进行统计分析的时候。我们提供了一种执行模式，在这种模式下，为了保证保证整个处理能继续进行，MapReduce 会检测哪些记录导致确定性的 crash，并且跳过这些记录不处理。

每个 worker 进程都设置了信号处理函数捕获内存段异常（segmentation violation）和总线错误（bus error）。在执行 Map 或者 Reduce 操作之前，MapReduce 库通过全局变量保存记录序号。如果用户程序触发了一个系统信号，消息处理函数将用“最后一口气”通过 UDP 包向 master 发送处理的最后一条记录的序号。当 master 看到在处理某条特定记录不止失败一次时，master 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

调试 Map 和 Reduce 函数的 bug 是非常困难的，因为实际执行操作时不但是分布在系统中执行的，而且通常是在好几千台计算机上执行，具体的执行位置是由 master 进行动态调度的，这又大大增加了调试的难度。为了简化调试、profile 和小规模测试，我们开发了一套 MapReduce 库的本地实现版本，通过使用本地版本的 MapReduce 库，MapReduce 操作在本地计算机上顺序的执行。用户可以控制 MapReduce 操作的执行，可以把操作限制到特定的 Map 任务上。用户通过设定特别的标志来在本地执行他们的程序，之后就可以很容易的使用本地调试和测试工具（比如 gdb）。

4.8 状态信息

master 使用嵌入式的 HTTP 服务器（如 Jetty）显示一组状态信息页面，用户可以监控各种执行状态。状态信息页面显示了包括计算执行的进度，比如已经完成了多少任务、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。页面还包含了指向每个任务的 stderr 和 stdout 文件的链接。用户根据这些数据预测计算需要执行大约多长时间、是否需要增加额外的计算资源。这些页面也可以用来分析什么时候计算执行的比预期的要慢。

另外，处于最顶层的状态页面显示了哪些 worker 失效了，以及他们失效的时候正在运行的 Map 和 Reduce 任务。这些信息对于调试用户代码中的 bug 很有帮助。

4.9 计数器

MapReduce 库使用计数器统计不同事件发生次数。比如，用户可能想统计已经处理了多少个单词、已经索引的多少篇 German 文档等等。

为了使用这个特性，用户在程序中创建一个命名的计数器对象，在 Map 和 Reduce 函数中相应的增加计数器的值。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");
```

这些计数器的值周期性的从各个单独的 worker 机器上传递给 master（附加在 ping 的应答包中传递）。master 把执行成功的 Map 和 Reduce 任务的计数器值进行累计，当 MapReduce 操作完成之后，返回给用户代码。

计数器当前的值也会显示在 master 的状态页面上，这样用户就可以看到当前计算的进度。当累加计数器的值的时候，master 要检查重复运行的 Map 或者 Reduce 任务，避免重复累加（之前提到的备用任务和失效后重新执行任务这两种情况会导致相同的任务被多次执行）。

有些计数器的值是由 MapReduce 库自动维持的，比如已经处理的输入的 key/value pair 的数量、输出的 key/value pair 的数量等等。

计数器机制对于 MapReduce 操作的完整性检查非常有用。比如，在某些 MapReduce 操作中，用户需要确保输出的 key value pair 精确的等于输入的 key value pair，或者处理的 German 文档数量在处理的整个文档数量中属于合理范围。

5 性能

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。一个计算在大约 1TB 的数据中进行特定的模式匹配，另一个计算对大约 1TB 的数据进行排序。

这两个程序在大量的使用 MapReduce 的实际应用中是非常典型的 — 一类是对数据格式进行转换，从一种表现形式转换为另外一种表现形式；另一类是从海量数据中抽取少部分的用户感兴趣的数据。

5.1 集群配置

所有这些程序都运行在一个大约由 1800 台机器构成的集群上。每台机器配置 2 个 2G 主频、支持超线程的 Intel Xeon 处理器，4GB 的物理内存，两个 160GB 的 IDE 硬盘和一个千兆以太网卡。这些机器部署在一个两层的树形交换网络中，在 root 节点大概有 100-200GBPS 的传输带宽。所有这些机器都采用相同的部署（对等部署），因此任意两点之间的网络来回时间小于 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。测试程序在周末下午开始执行，这时主机的 CPU、磁盘和网络基本上处于空闲状态。

5.2 GREP

这个分布式的 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，查找出现概率较小的 3 个字符的模式（这个模式在 92337 个记录中出现）。输入数据被拆分成大约 64M 的 Block（M=15000），整个输出数据存放在一个文件中（R=1）。

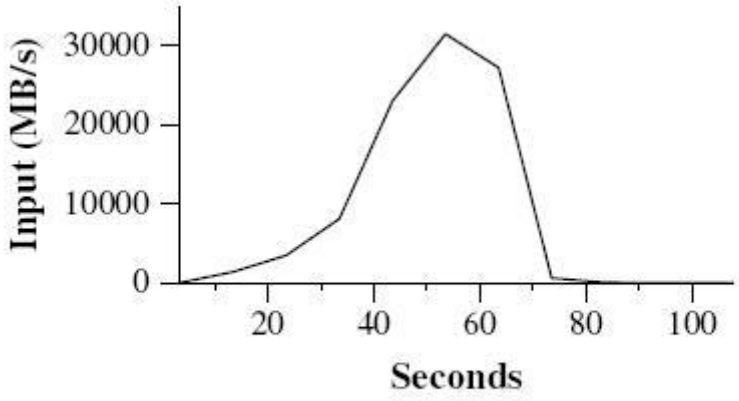

Figure 2: Data transfer rate over time

图 2 显示了这个运算随时间的处理过程。其中 Y 轴表示输入数据的处理速度。处理速度随着参与 MapReduce 计算的机器数量的增加而增加，当 1764 台 worker 参与计算的时，处理速度达到了 30GB/s。当 Map 任务结束的时候，即在计算开始后 80 秒，输入的处理速度降到 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初始启动阶段。初始启动阶段消耗的时间包括了是把这个程序传送到各个 worker 机器上的时间、等待 GFS 文件系统打开 1000 个输入文件集合的时间、获取相关的文件本地位置优化信息的时间。

5.3 排序

排序程序处理 10 的 10 次方个 100 个字节组成的记录（大概 1TB 的数据）。这个程序模仿 TeraSort benchmark[10]。

排序程序由不到 50 行代码组成。只有三行的 Map 函数从文本行中解析出 10 个字节的 key 值作为排序的 key，并且把这个 key 和原始文本行作为中间的 key/value pair 值输出。我们使用了一个内置的恒等函数作为 Reduce 操作函数。这个函数把中间的 key/value pair 值不作任何改变输出。最终排序结果输出到两路复制的 GFS 文件系统（也就是说，程序输出 2TB 的数据）。

如前所述，输入数据被分成 64MB 的 Block（M=15000）。我们把排序后的输出结果分区后存储到 4000 个文件（R=4000）。分区函数使用 key 的原始字节来把数据分区到 R 个片段中。

在这个 benchmark 测试中，我们使用的分区函数知道 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 操作用于采样 key 值的分布情况，通过采样的数据来计算对最终排序处理的分区点。

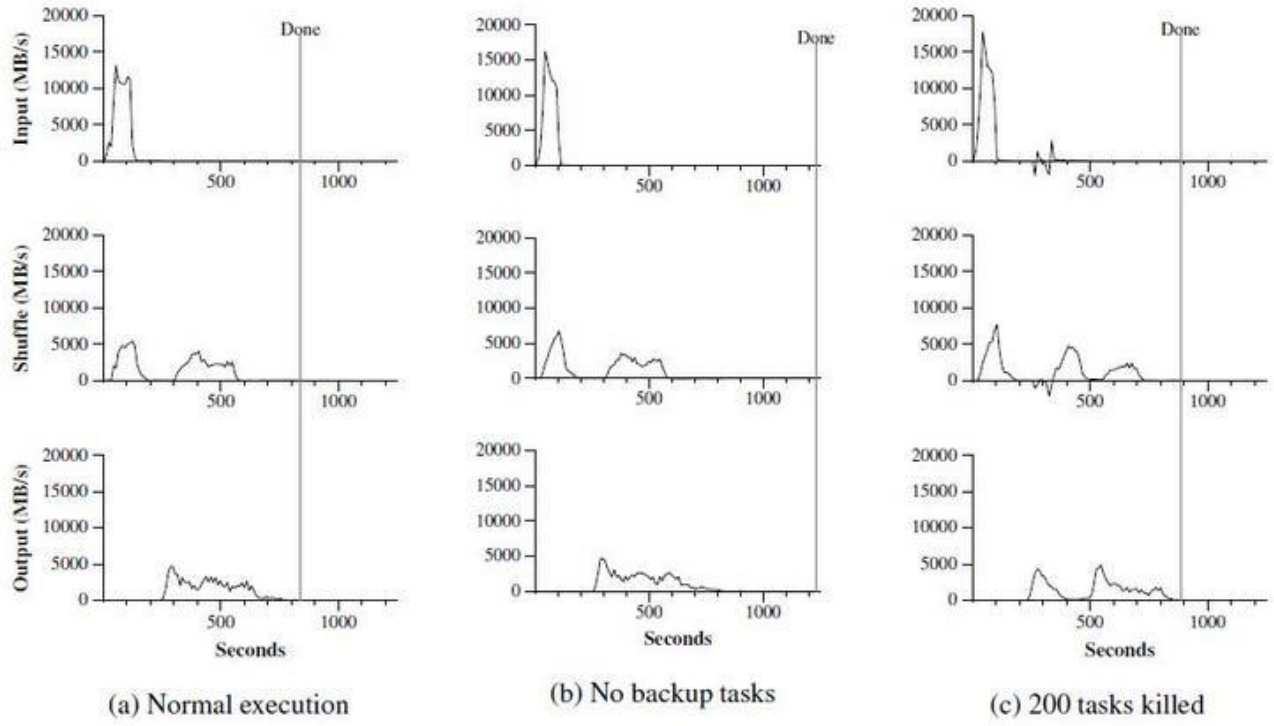

Figure 3: Data transfer rates over time for different executions of the sort program

图三（a）显示了这个排序程序的正常执行过程。左上的图显示了输入数据读取的速度。数据读取速度峰值会达到 13GB/s，并且所有 Map 任务完成之后，即大约 200 秒之后迅速滑落到 0。值得注意的是，排序程序输入数据读取速度小于分布式 grep 程序。这是因为排序程序的 Map 任务花了大约一半的处理时间和 I/O 带宽把中间输出结果写到本地硬盘。相应的分布式 grep 程序的中间结果输出几乎可以忽略不计。

左边中间的图显示了中间数据从 Map 任务发送到 Reduce 任务的网络速度。这个过程从第一个 Map 任务完成之后就开始缓慢启动了。图示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 1 次最多执行 1 个 Reduce 任务）。排序程序运行大约 300 秒后，第一批启动的 Reduce 任务有些完成了，我们开始执行剩下的 Reduce 任务。所有的处理在大约 600 秒后结束。

左下图表示 Reduce 任务把排序后的数据写到最终的输出文件的速度。在第一个排序阶段结束和数据开始写入磁盘之间有一个小的延时，这是因为 worker 机器正在忙于排序中间数据。磁盘写入速度在 2-4GB/s 持续一段时间。输出数据写入磁盘大约持续 850 秒。计入初始启动部分的时间，整个运算消耗了 891 秒。这个速度和 TeraSort benchmark[18]的最高纪录 1057 秒相差不多。

还有一些值得注意的现象：输入数据的读取速度比排序速度和输出数据写入磁盘速度要高不少，这是因为我们的输入数据本地化优化策略起了作用 — 绝大部分数据都是从本地硬盘读取的，从而节省了网络带宽。排序速度比输出数据写入到磁盘的速度快，这是因为输出数据写了两份（我们使用了 2 路的 GFS 文件系统，写入复制节点的原因是为了保证数据可靠性和可用性）。我们把输出数据写入到两个复制节点的原因是因为这是底层文件系统的保证数据可靠性和可用性的实现机制。如果底层文件系统使用类似容错编码[14](erasure coding)的方式而不是复制的方式保证数据的可靠性和可用性，那么在输出数据写入磁盘的时候，就可以降低网络带宽的使用。

5.4 高效的 backup 任务

图三（b）显示了关闭了备用任务后排序程序执行情况。执行的过程和图 3（a）很相似，除了输出数据写磁盘的动作在时间上拖了一个很长的尾巴，而且在这段时间里，几乎没有什么写入动作。在 960 秒后，只有 5 个 Reduce 任务没有完成。这些拖后腿的任务又执行了 300 秒才完成。整个计算消耗了 1283 秒，多了 44%的执行时间。

5.5 失效的机器

在图三（c）中演示的排序程序执行的过程中，我们在程序开始后几分钟有意的 kill 了 1746 个 worker 中的 200 个。集群底层的调度立刻在这些机器上重新开始新的 worker 处理进程（因为只是 worker 机器上的处理进程被 kill 了，机器本身还在工作）。

图三（c）显示出了一个“负”的输入数据读取速度，这是因为一些已经完成的 Map 任务丢失了（由于相应的执行 Map 任务的 worker 进程被 kill 了），需要重新执行这些任务。相关 Map 任务很快就被重新执行了。整个运算在 933 秒内完成，包括了初始启动时间（只比正常执行多消耗了 5%的时间）。

6 经验

我们在 2003 年 1 月完成了第一个版本的 MapReduce 库，在 2003 年 8 月的版本有了显著的增强，这包括了输入数据本地优化、worker 机器之间的动态负载均衡等等。从那以后，我们惊喜的发现，MapReduce 库能广泛应用于我们日常工作中遇到的各类问题。它现在在 Google 内部各个领域得到广泛应用，包括：

1. 大规模机器学习问题

2. Google News 和 Froogle 产品的集群问题
3. 从公众查询产品（比如 Google 的 Zeitgeist）的报告中抽取数据。
4. 从大量的新应用和新产品的网页中提取有用信息（比如，从大量的位置搜索网页中抽取地理位置信息）。
5. 大规模的图形计算。

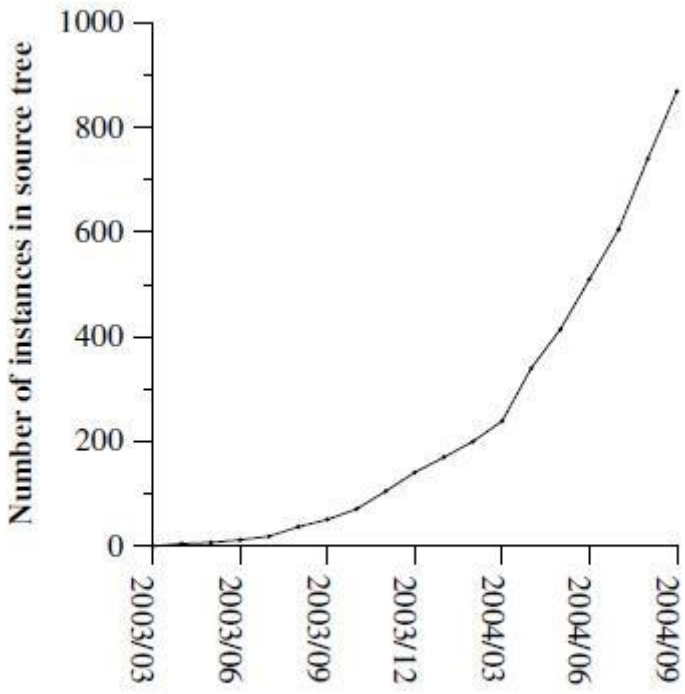

Figure 4: MapReduce instances over time

图四显示了在我们的源代码管理系统中，随着时间推移，独立的 MapReduce 程序数量的显著增加。从 2003 年早些时候的 0 个增长到 2004 年 9 月份的差不多 900 个不同的程序。MapReduce 的成功取决于采用 MapReduce 库能够在不到半个小时时间内写出一个简单的程序，这个简单的程序能够在上千台机器的组成的集群上做大规模并发处理，这极大的加快了开发和原形设计的周期。另外，采用 MapReduce 库，可以让完全没有分布式和/或并行系统开发经验的程序员很容易的利用大量的资源，开发出分布式和/或并行处理的应用。

任务数	29423
平均任务完成时间	634 秒
使用的机器时间	79,186 天
读取的输入数据	3,288TB
产生的中间数据	758TB
写出的输出数据	193TB
每个 job 平均 worker 机器数	157
每个 job 平均死掉 work 数	1.2
每个 job 平均 map 任务	3,351
每个 job 平均 reduce 任务	55
map 唯一实现	395
reduce 的唯一实现	296
map/reduce 的 combiner 实现	426

表 1：MapReduce2004 年 8 月的执行情况

在每个任务结束的时候，MapReduce 库统计计算资源的使用状况。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 大规模索引

到目前为止，MapReduce 最成功的应用就是重写了 Google 网络搜索服务所使用到的 index 系统。索引系统的输入数据是网络爬虫抓取回来的海量的文档，这些文档数据都保存在 GFS 文件系统里。这些文档原始内容[4]的大小超过了 20TB。索引程序是通过一系列的 MapReduce 操作（大约 5 到 10 次）来建立索引。使用 MapReduce（替换上一个特别设计的、分布式处理的索引程序）带来这些好处：

实现索引部分的代码简单、小巧、容易理解，因为对于容错、分布式以及并行计算的处理都是 MapReduce 库提供的。比如，使用 MapReduce 库，计算的代码行数从原来的 3800 行 C++代码减少到大概 700 行代码。

MapReduce 库的性能已经足够好了，因此我们可以把在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数据传递的额外消耗。概念上不相关的计算步骤的隔离也使得我们可以很容易改变索引处理方式。比如，对之前的索引系统的一个小更改可能要耗费好几个月的时间，但是在使用 MapReduce 的新系统上，这样的更改只需要花几天时间就可以了。

索引系统的操作管理更容易了。因为由机器失效、机器处理速度缓慢、以及网络的瞬间阻塞等引起的绝大部分问题都已经由 MapReduce 库解决了，不再需要操作人员的介入了。另外，我们可以通过在索引系统集群中增加机器的简单方法提高整体处理性能。

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并行计算。例如，一个结合函数

[4] raw contents，我认为就是网页中的剔除 html 标记后的内容、pdf 和 word 等有格式文档中提取的文本内容等

可以通过把 N 个元素的数组的前缀在 N 个处理器上使用并行前缀算法，在 log N 的时间内计算完[6，9，13][5]。MapReduce 可以看作是我们结合在真实环境下处理海量数据的经验，对这些经典模型进行简化和萃取的成果。更加值得骄傲的是，我们还实现了基于上千台处理器的集群的容错处理。相比而言，大部分并发处理系统都只在小规模的集群上实现，并且把容错处理交给了程序员。

Bulk Synchronous Programming[17]和一些 MPI 原语[11]提供了更高级别的并行处理抽象，可以更容易写出并行处理的程序。MapReduce 和这些系统的关键不同之处在于，MapReduce 利用限制性编程模式实现了用户程序的自动并发处理，并且提供了透明的容错处理。

我们数据本地优化策略的灵感来源于 active disks[12,15]等技术，在 active disks 中，计算任务是尽量推送到数据存储的节点处理[6]，这样就减少了网络和 IO 子系统的吞吐量。我们在挂载几个硬盘的普通机器上执行我们的运算，而不是在磁盘处理器上执行我们的工作，但是达到的目的一样的。

我们的备用任务机制和 Charlotte System[3]提出的 eager 调度机制比较类似。Eager 调度机制的一个缺点是如果一个任务反复失效，那么整个计算就不能完成。我们通过忽略引起故障的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的、共享机器的集群上分布和运行用户任务。虽然这个不是本论文的重点，但是有必要提一下，这个集群管理系统在理念上和其它系统，如 Condor[16]是一样。

MapReduce 库的排序机制和 NOW-Sort[1]的操作上很类似。读取输入源的机器（map workers）把待排序的数据进行分区后，发送到 R 个 Reduce worker 中的一个进行处理。每个 Reduce worker 在本地对数据进行排序（尽可能在内存中排序）。当然，NOW-Sort 没有给用户自定义的 Map 和 Reduce 函数的机会，因此不具备 MapReduce 库广泛的实用性。

River[2]提供了一个编程模型：处理进程通过分布式队列传送数据的方式进行互相通讯。和 MapReduce 类似，River 系统尝试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 库采用了其它的方法。通过对编程模型进行限制，MapReduce 框架把问题分解成为大量的“小”任务。这些任务在可用的 worker 集群上动态的调度，这样快速的 worker 就可以执行更多的任务。通过对编程模型进行限制，我们可用在工作接近完成的时候调度备用任务，缩短在硬件配置不均衡的情况下缩小整个操作完成的时间（比如有的机器性能差、或者机器被某些操作阻塞了）。

BAD-FS[5]采用了和 MapReduce 完全不同的编程模式，它是面向广域网（alex 注：wide-area network）的。

---

[5] 注：完全没有明白作者在说啥，具体参考相关 6、9、13 文档

[6] 即靠近数据源处理

不过，这两个系统有两个基础功能很类似。（1）两个系统采用重新执行的方式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，减少网络通讯的数据量。

TACC[7]是一个用于简化构造高可用性网络服务的系统。和 MapReduce 一样，它也依靠重新执行机制来实现的容错处理。

8 结束语

MapReduce 编程模型在 Google 内部成功应用于多个领域。我们把这种成功归结为几个方面：首先，由于 MapReduce 封装了并行处理、容错处理、数据本地化优化、负载均衡等等技术难点的细节，这使得 MapReduce 库易于使用。即便对于完全没有并行或者分布式系统开发经验的程序员而言；其次，大量不同类型的问题都可以通过 MapReduce 简单的解决。比如，MapReduce 用于生成 Google 的网络搜索服务所需要的数据、用来排序、用来数据挖掘、用于机器学习，以及很多其它的系统；第三，我们实现了一个在数千台计算机组成的大型集群上灵活部署运行的 MapReduce。这个实现使得有效利用这些丰富的计算资源变得非常简单，因此也适合用来解决 Google 遇到的其他很多需要大量计算的问题。

我们也从 MapReduce 开发过程中学到了不少东西。首先，约束编程模式使得并行和分布式计算非常容易，也易于构造容错的计算环境；其次，网络带宽是稀有资源。大量的系统优化是针对减少网络传输量为目的的：本地优化策略使大量的数据从本地磁盘读取，中间文件写入本地磁盘、并且只写一份中间文件也节约了网络带宽；第三，多次执行相同的任务可以减少性能缓慢的机器带来的负面影响（alex 注：即硬件配置的不平衡），同时解决了由于机器失效导致的数据丢失问题。

9 感谢

Josh Levenberg has been instrumental in revising and extending the user-level MapReduce API with a number of new features based on his experience with using MapReduce and other people’s suggestions for enhancements. MapReduce reads its input from and writes its output to the Google File System [8]. We would like to thank Mohit Aron, Howard Gobioff, Markus Gutschke, David Kramer, Shun-Tak Leung, and Josh Redstone for their work in developing GFS. We would also like to thank Percy Liang and Olcan Sercinoglu for their work in developing the cluster management system used by MapReduce. Mike Burrows, Wilson Hsieh, Josh Levenberg, Sharon Perl, Rob Pike, and Debby Wallach provided helpful comments on earlier drafts of this paper.The anonymous OSDI reviewers, and our shepherd, Eric Brewer, provided many useful suggestions of areas where the paper could be improved. Finally, we thank all the users of MapReduce within Google’s engineering organization for providing helpful feedback, suggestions, and bug reports.

10 参考资料

[1] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau,David E. Culler, Joseph M. Hellerstein, and David A. Patterson.High-performance sorting on networks of workstations.In Proceedings of the 1997 ACM SIGMOD InternationalConference on Management of Data, Tucson,Arizona, May 1997.

[2] Remzi H. Arpaci-Dusseau, Eric Anderson, NoahTreuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River:Making the fast case common. In Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS ’99), pages 10.22, Atlanta, Georgia, May 1999.

[3] Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the web. In Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems, 1996. [4] Luiz A. Barroso, Jeffrey Dean, and Urs H ̈olzle. Web search for a planet: The Google cluster architecture. IEEE Micro, 23(2):22.28, April 2003.

[5] John Bent, Douglas Thain, Andrea C.Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control in a batch-aware distributed file system. In Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation NSDI, March 2004.

[6] Guy E. Blelloch. Scans as primitive parallel operations.IEEE Transactions on Computers, C-38(11), November 1989.

[7] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In Proceedings of the 16th ACM Symposium on Operating System Principles, pages 78. 91, Saint-Malo, France, 1997.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In 19th Symposium on Operating Systems Principles, pages 29.43, Lake George, New York, 2003. To appear in OSDI 2004 12

[9] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, Euro-Par’96. Parallel Processing, Lecture Notes in Computer Science 1124, pages 401.408. Springer-Verlag, 1996.

[10] Jim Gray. Sort benchmark home page. http://research.microsoft.com/barc/SortBenchmark/.

[11] William Gropp, Ewing Lusk, and Anthony Skjellum. Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press, Cambridge, MA, 1999.

[12] L. Huston, R. Sukthankar, R.Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In Proceedings of the 2004

USENIX File and Storage Technologies FAST Conference, April 2004.

[13] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. Journal of the ACM, 27(4):831.838, 1980.

[14] Michael O. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. Journal of the ACM, 36(2):335.348, 1989.

[15] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. IEEE Computer, pages 68.74, June 2001.

[16] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The Condor experience. Concurrency and Computation: Practice and Experience, 2004.

[17] L. G. Valiant. A bridging model for parallel computation. Communications of the ACM, 33(8):103.111, 1997.

[18] Jim Wyllie. Spsort: How to sort a terabyte quickly. http://alme1.almaden.ibm.com/cs/spsort.pdf.

11 附录 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’s map function
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; ) {
      // Skip past leading whitespace
      while ((i < n) && isspace(text[i]))
        i++;

      // Find word end
      int start = i;
```

```
      while ((i < n) && !isspace(text[i]))
        i++;
      if (start < i)
        Emit(text.substr(start,i-start),” 1″ );
    }
  }
};

REGISTER_MAPPER(WordCounter);

// User’s reduce function
class Adder : public Reducer {
  virtual void Reduce(ReduceInput* input) {
    // Iterate over all entries with the
    // same key and add the values
    int64 value = 0;
    while (!input->done()) {
      value += StringToInt(input->value());
      input->NextValue();
    }

    // Emit sum for input->key()
    Emit(IntToString(value));
  }
};

REGISTER_REDUCER(Adder);

int main(int argc, char** argv) {
  ParseCommandLineFlags(argc, argv);
```

```
MapReduceSpecification spec;

// Store list of input files into "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");
}

// Specify the output files:
// /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: do partial sums within map
// tasks to save network bandwidth
out->set_combiner_class("Adder");

// Tuning parameters: use at most 2000
// machines and 100 MB of memory per task
spec.set_machines(2000);
spec.set_map_megabytes(100);
spec.set_reduce_megabytes(100);
```

```
  // Now run it
  MapReduceResult result;
  if (!MapReduce(spec, &result)) abort();

  // Done: 'result' structure contains info
  // about counters, time taken, number of
  // machines used, etc.
  return 0;
}
```