**谭文江、朱宝、张开**

**映射化简：大型集群上简化的数据处理**

Jeffrey Dean和Sanjay Ghemawat

jeff@google.com, sanjay@google.com

*Google*

摘要

映射化简是用于处理和生成大型数据集的编程模型和相关实现。用户指定一个映射函数，该函数处理键/值对以生成一组中间键/值对，以及一个化简函数，该函数合并与同一中间键相关联的所有中间值。许多现实世界中的任务都可以在这个模型中表达，如本文所示。

用这种函数式编写的程序会自动并行化，并在大型商用计算机集群上执行。运行时系统负责对输入数据进行分区、在一组机器上调度程序的执行、处理机器故障以及管理所需的机器间通信等细节。这使得没有并行和分布式系统经验的程序员能够轻松地利用大型分布式系统的资源。

我们的映射化简实现运行在一个大型的商品机器集群上，并且具有很高的可伸缩性：典型的映射化简计算在数千台机器上处理许多兆字节的数据。程序员们发现这个系统很容易使用：数百个映射化简程序已经实现，每天在Google的集群上执行的映射化简作业超过1000个。

1.介绍

在过去的五年里，作者和谷歌的许多人已经实现了数百种特殊用途的计算，这些计算处理大量原始数据，如爬网文档、web请求日志等，以计算各种派生数据，如反向索引、web文档图结构的各种表示形式，每个主机爬网的页面数的摘要，一天中最频繁的查询集等等。大多数这样的计算在概念上是直接进行的。然而，输入的数据通常很大，为了在合理的时间内完成计算，必须将计算分布在成百上千台机器上。如何并行化计算、分发数据和处理失败等问题，使原本简单的计算变得模糊，而处理这些问题的大量复杂代码的作用就能体现出来。

作为对这种复杂性的反应，我们设计了一种新的抽象，它允许我们表达我们试图执行的简单计算，但隐藏了库中并行化、容错、数据分布和负载平衡的混乱细节。我们的抽象源于Lisp和许多其他函数语言中的map（映射）和reduce（缩小，化简）原语。我们意识到，我们的大多数计算都涉及对输入中的每个逻辑“记录”应用映射操作，以便计算一组中间键/值对，然后对共享同一密钥的所有值应用化简操作，以便适当地组合派生的数据。我们使用带有用户指定的map（映射）和reduce（缩小，化简）操作的函数模型，可以轻松地并行化大型计算，并将重新执行作为容错的主要机制。

这项工作的主要贡献是一个简单而强大的接口，它可以实现大规模计算的自动并行和分布，再加上这个接口的实现，可以在大型商用PC机群上实现高性能。

第2节描述了基本编程模型并给出了几个示例。第3节描述了为我们基于集群的计算环境定制的映射化简接口的实现。第4节描述了我们发现有用的编程模型的一些改进。第5节介绍了我们对各种任务的实现的性能度量。第6节探讨了映射化简在Google中的使用，包括我们使用它作为重写产品索引系统的基础的经验。第7节讨论了相关和未来的工作。

2.程序设计模型

计算采用一组输入键/值对，并生成一组输出键/值对。映射化简库中的用户将计算表示为两个函数：映射和化简。

map（映射），由用户编写，接受一个输入对并生成一组中间键/值对。映射化简库将与同一中间键I关联的所有中间值组合在一起，并将它们传递给化简函数。

reduce（缩小，化简）函数同样也是由用户编写的，它接受中间键I和该键的一组值。它将这些值合并在一起形成一个可能更小的值集。通常每次化简调用只产生零或一个输出值。中间值通过迭代器提供给用户的化简函数。这使我们能够处理太大而无法放入内存的值列表。

**2.1 例子**

考虑计算一个大型文档集合中每个单词出现的次数的问题。用户将编写类似以下伪代码的代码：

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));

映射函数发出每个单词加上一个相关的出现次数（在这个简单的例子中是“1”）。化简函数将为特定单词发出的所有计数相加。

此外，用户编写代码，用输入和输出文件的名称以及可选的调优参数填充映射化简规范对象。然后，用户调用映射化简函数，将规范对象传递给它。用户的代码与映射化简库（C++实现）连接在一起。附录A包含此示例的完整程序文本。

**2.2 类型**

尽管前面的伪代码是根据字符串输入和输出编写的，但从概念上讲，用户提供的映射和化简函数具有关联类型：

      map（k1，v1）→list（k2，v2）

reduce（k2，list（v2））→list（v2）

即输入键和值是从与输出键和值不同的域中提取的。此外，中间键和值与输出键和值来自同一个域。

我们的C++实现将字符串传递到用户定义的函数，并将其留给用户代码，以便在字符串和适当类型之间转换。

**2.3更多示例**

下面是一些有趣程序的简单示例，它们可以很容易地表示为映射化简计算。

**分布式Grep：**如果映射函数与提供的模式匹配，那么它将发出一条线。化简函数是一个标识函数，它只将提供的中间数据复制到输出。

**URL访问频率计数：**映射函数处理网页请求的日志并输出（URL，1）。化简函数将同一URL的所有值相加，并发出一个（URL，total count）对。

**反向Web链接图：**在每个source-arget函数的源页面中找到一个名为target的URL链接。化简函数连接与给定目标URL关联的所有源URL的列表，并发出对：< target，list（source）>

**每个主机的术语向量：**术语向量将出现在文档或一组文档中的最重要的单词概括为（word、frequency）对的列表。映射函数为每个输入文档（主机名从文档的URL中提取）发出（hostname，term vector）对。化简函数传递给定主机的所有每个文档项向量。它将这些项向量相加，丢弃不常出现的项，然后发出最后一个（hostname，term vectori）对。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 输入文件 | 地图  阶段 | 中间文件  （在本地磁盘上）  图1：执行概述 | 减少相位 | 输出文件 |

|  |
| --- |
|  |
|  |  |

**反向索引：**映射函数解析每个文档，并发出一系列（word, document ID）对。化简函数接受给定单词的所有对，对相应的文档ID进行排序，并发出（word，list（document ID））对。所有输出对的集合形成一个简单的倒排索引。很容易增加这种计算来跟踪单词的位置。

**分布式排序：**映射函数从每个记录中提取密钥，并发出一个（key，record）对。化简函数发出所有对不变。该计算取决于第4.1节中描述的分区设施和第4.2节中描述的排序属性。

3.实施

映射化简接口有许多不同的实现。正确的选择取决于环境。例如，一种实现可能适合于小型共享内存机器，另一种适用于大型NUMA多处理器，以及另一种更大的联网机器集合。

本文描述了在大范围内使用以太网连接到大型计算机集群的实现。在我们的环境中：

（1） 机器通常是运行Linux的双处理器x86处理器，每台机器有2-4 GB的内存。

（2） 使用的是商品网络硬件——通常在机器级别为100兆位/秒或1千兆位/秒，但平均总平分带宽要少得多。

（3） 集群由成百上千台机器组成，因此机器故障很常见。

（4） 存储由直接连接到单个机器的廉价IDE磁盘提供。内部开发的分布式文件系统用于管理存储在这些磁盘上的数据。文件系统使用复制在不可靠的硬件上提供可用性和可靠性。

（5） 用户向调度系统提交作业。每个作业由一组任务组成，并由调度程序映射到集群中的一组可用计算机。

**3.1执行概述**

通过自动将输入数据划分为一组M个分割，映射调用分布在多台机器上。输入拆分可以由不同的机器并行处理。化简调用是通过使用一个分区函数*(e.g., hash(key) mod R)*将中间密钥空间划分为R个片段来分布的。分区数（R）和分区函数由用户指定。

图1显示了我们实现中映射化简操作的总体流程。当用户程序调用映射化简函数时，将发生以下操作序列（图1中编号的标签对应于下面列表中的数字）：

1.    用户程序中的映射化简库首先将输入文件分成M个部分，通常每段16兆字节到64兆字节（MB）（由用户通过可选参数控制）。然后它会在一组机器上启动程序的许多副本。

2.    这个程序的其中一个副本是特别的 -- 车间主。其余的都是车间主指派的工人。有M个映射任务和R个化简任务要分配。主节点选择空闲的工人并为每个工人分配一个映射任务或化简任务。

3.    分配了映射任务的工作人员读取相应输入拆分的内容。它从输入数据中解析出键/值对，并将每个对传递给用户定义的映射函数。由映射函数产生的中间键/值对被缓冲在内存中。

4.    缓冲对被周期性地写入本地磁盘，由分区函数划分为R个区域。本地磁盘上这些缓冲对的位置被传回主机，主机负责将这些位置转发给化简工人。

5.    当主机通知化简 工人这些位置时，它使用远程过程调用从映射工人的本地磁盘读取缓冲数据。当化简 工人读取了所有中间数据时，它将按中间键对其进行排序，以便将同一键的所有出现组合在一起。之所以需要排序，是因为通常有许多不同的键映射到同一个化简任务。如果中间数据量太大而无法放入内存，则使用外部排序。

6.    化简工人迭代已排序的中间数据，对于遇到的每个唯一中间键，它将键和相应的中间值集传递给用户的化简函数。化简函数的输出被附加到这个化简分区的最终输出文件中。

7.    当所有映射任务和化简任务都已完成时，主程序将唤醒用户程序。此时，用户程序中的映射化简调用返回到用户代码。

成功完成后，映射化简执行的输出将在R输出文件中可用（每个化简任务一个，文件名由用户指定）。通常，用户不需要将这些R输出文件合并到一个文件中 -- 他们通常将这些文件作为输入传递给另一个映射化简调用，或者从另一个能够处理划分为多个文件的输入的分布式应用程序中使用这些文件。

**3.2 主数据结构**

车间主保留几个数据结构。对于每个映射任务和化简任务，它存储状态（空闲、正在进行或已完成）以及工作机的标识（对于非空闲任务）。

主节点是从映射任务传播中间文件区域位置以减少任务的管道。因此，对于每个完成的映射任务，车间主存储由映射任务生成的R中间文件区域的位置和大小。当地图任务完成时，会收到对此位置和大小信息的更新。信息将以增量方式推送到具有正在进行的化简任务的工作人员。

**3.3 容错**

由于映射化简库的设计目的是帮助使用成百上千台机器处理大量数据，因此该库必须优雅地容忍机器故障。

**工人故障**

车间主定期对每个工人进行ping（乒网）检查。如果在一定时间内没有从工作线程接收到响应，则主进程会将该工作进程标记为失败。由工作线程完成的任何映射任务都会重置回其初始空闲状态，因此有资格在其他工作进程上进行调度。类似地，在失败的辅助进程上正在进行的任何映射任务或化简任务也将重置为空闲，并有资格重新安排。

失败时会重新执行已完成的映射任务，因为它们的输出存储在发生故障的计算机的本地磁盘上，因此无法访问。完成的化简任务不需要重新执行，因为它们的输出存储在全局文件系统中。

当一个映射任务首先由工人 a执行，然后由工人 B执行（因为a失败），所有执行化简任务的工人都会收到重新执行的通知。任何尚未从工人A读取数据的化简任务都将从工人B读取数据。

映射化简对大规模的工人故障具有弹性。例如，在一次映射化简操作期间，运行集群上的网络维护导致一次80台机器的组在几分钟内无法访问。映射化简主机只需重新执行无法访问的工作机所做的工作，并继续向前推进，最终完成映射化简操作。

**主机故障**

很容易使车间主写入上述主数据结构的周期性检查点。如果主任务终止，则可以从上一个检查点状态启动新副本。但是，考虑到只有一个主节点，它不太可能失败；因此，如果主节点失败，我们当前的实现将中止映射化简计算。客户端可以检查这种情况，如果需要，可以重试映射化简操作。

**出现故障时的语义**

当用户提供的映射和化简运算符是其输入值的确定函数时，我们的分布式实现产生的输出与整个程序的无故障顺序执行所产生的输出相同。

我们依靠映射的原子提交和减少任务输出来实现这个属性。每个正在进行的任务将其输出写入私有临时文件。化简任务生成一个这样的文件，映射任务生成R这样的文件（每个化简任务一个）。映射任务完成后，worker向主服务器发送一条消息，并在消息中包含R临时文件的名称。如果主机接收到已完成的映射任务的完成消息，它将忽略该消息。否则，它会在主数据结构中记录R文件的名称。

当化简任务完成时，化简 worker将其临时输出文件自动重命名为最终输出文件。如果在多台计算机上执行相同的化简任务，则将对同一最终输出文件执行多个重命名调用。我们依赖底层文件系统提供的原子重命名操作来保证最终文件系统状态只包含化简任务一次执行所产生的数据。

我们的映射和化简运算符绝大多数都是确定性的，而且在这种情况下，我们的语义相当于一个顺序执行，这使得程序员很容易对他们的程序的行为进行推理。当映射和/或化简运算符不确定时，我们提供了较弱但仍然合理的语义。在非确定性算子存在的情况下，一个特定的化简 task Ris的输出相当于由非确定性程序的顺序执行所产生的r的输出。然而，不同化简 task rma的输出可能对应于由非确定性程序的不同顺序执行产生的r的输出。

考虑映射任务M和化简任务 *R1*和*R2*，让e（Ri）被承诺的Ri执行（只有一个这样的执行）。语义较弱的原因是e（R1）可能已经读取了M的一次执行所产生的输出，而e（R2）可能已经读取了M的另一次执行所产生的输出。

**3.4 地点**

在我们的计算环境中，网络带宽是一种相对稀缺的资源。我们利用输入数据（由GFS管理）存储在组成集群的机器的本地磁盘上，从而节省了网络带宽。GFS将每个文件分成64mb块，并在不同的计算机上存储每个块的多个副本（通常为3个副本）。映射化简主机考虑输入文件的位置信息，并尝试在包含相应输入数据副本的计算机上调度映射任务。否则，它会尝试在任务输入数据的副本附近安排映射任务（例如，在与包含数据的计算机位于同一网络交换机上的工作机上）。当在集群中的大部分工作线程上运行大型映射化简操作时，大多数输入数据都是在本地读取的，并且不消耗网络带宽。

**3.5任务粒度**

我们将映射阶段细分为M个片段，并将化简阶段细分为R个片段，如上所述。理想情况下，M和R应该远大于工作机器的数量。让每个工人执行许多不同的任务可以提高动态负载平衡，并在工人失败时加速恢复：它完成的许多映射任务可以分散到所有其他工人计算机上。

在我们的实现中，M和R的大小有实际的限制，因为主机必须做出O（M+R）调度决策，并如上所述在内存中保持O（M\*R）状态。（但是，内存使用的常量因子很小：状态的O（M\*R）部分由每个映射 任务/化简任务对大约一个字节的数据组成。）

此外，R常常受到用户的限制，因为每个化简任务的输出都会在一个单独的输出文件中结束。在实践中，我们倾向于选择M，以便每个单独的任务大约有16mb到64mb的输入数据（因此上面描述的局部性优化是最有效的），并且我们将R设为期望使用的工作机数量的一个小倍数。我们通常使用2000台工人机器，以M=200000和R=5000执行映射化简计算。

**3.6备份任务**

延长映射化简操作所花费的总时间的一个常见原因是“掉队者”：一台需要非常长时间来完成计算中最后几个映射或化简任务之一的机器。掉队者的出现有很多原因。例如，一台有坏磁盘的机器可能会遇到频繁的可纠正错误，导致其读取性能从30 MB/s降低到1 MB/s。群集调度系统可能已在计算机上计划了其他任务，导致其由于CPU、内存、本地磁盘或网络带宽的竞争而更慢地执行映射化简代码。最近我们遇到的一个问题是机器初始化代码中的一个错误，导致处理器缓存被禁用：受影响机器上的计算速度减慢了100倍以上。

我们有一个总的机制来缓解落伍者的问题。当映射化简操作即将完成时，主服务器将调度剩余正在进行的任务的备份执行。每当主或备份执行完成时，任务都会标记为已完成。我们已经对该机制进行了优化，使其通常只将操作使用的计算资源增加不超过百分之几。我们发现这大大缩短了完成大型映射化简操作的时间。例如，当备份任务机制被禁用时，第5.3节中描述的排序程序需要花费44%的时间才能完成。

4.改进

尽管简单地编写映射和化简函数所提供的基本功能已经足够满足大多数需求，但是我们发现一些扩展非常有用。这些将在本节中描述。

**4.1配分函数**

映射化简的用户指定他们想要的化简任务/输出文件的数量（R）。使用中间键上的分区函数在这些任务之间对数据进行分区。提供了一个使用哈希的默认分区函数（例如“*hash*（*key*）mod R”）。这会导致相当平衡的分区。但是，在某些情况下，通过键的其他函数对数据进行分区是很有用的。例如，有时输出键是网址，我们希望单个主机的所有条目都在同一个输出文件中结束。为了支持这种情况，映射化简库的用户可以提供一个特殊的分区函数。例如，使用“hash（Hostname（urlkey））mod R”作为分区函数，会导致来自同一主机的所有网址最终位于同一个输出文件中。

**4.2订货保证书**

我们保证在给定的分区内，中间的键/值对按递增的键顺序进行处理。这种排序保证使得为每个分区生成一个已排序的输出文件变得很容易，当输出文件格式需要支持按键进行有效的随机访问查找，或者输出的用户发现对数据进行排序很方便时，这一点非常有用。

**4.3合路器功能**

在某些情况下，每个映射任务产生的中间密钥中存在显著的重复，并且用户指定的化简函数是可交换和关联的。这方面的一个很好的例子是第2.1节中的单词计数示例。由于词频倾向于遵循Zipf分布，每个映射任务将产生数百个

或是成千上万的记录。所有这些计数将通过网络发送给一个化简任务，然后由化简函数加在一起生成一个数字。我们允许用户指定一个可选的组合器函数，在数据通过网络发送之前对其进行部分合并。

合并器功能在执行映射任务的每台机器上执行。通常，相同的代码用于实现组合器和化简函数。化简函数和结合函数之间的唯一区别是映射化简库如何处理函数的输出。化简函数的输出将写入最终输出文件。一个任务将被发送到化简函数的中间输出。

部分合并可以显著加快某些类的映射化简操作。附录A包含一个使用组合器的示例。

**4.4输入和输出类型**

映射化简库支持读取几种不同格式的输入数据。例如，“文本”模式输入将每一行视为键/值对：键是文件中的偏移量，值是行的内容。另一种常见的支持格式存储按键排序的键/值对序列。每个输入类型实现都知道如何将自身分割成有意义的范围，以便作为单独的映射任务进行处理（例如，文本模式的范围分割确保范围分割仅在行边界处发生）。用户可以通过提供一个简单的读取器接口的实现来添加对新输入类型的支持，尽管大多数用户只使用少数预定义输入类型中的一种。

读取器不一定需要提供从文件中读取的数据。例如，很容易定义一个读卡器，从数据库或内存中映射的数据结构读取记录。

以类似的方式，我们支持一组输出类型来生成不同格式的数据，而且用户代码很容易添加对新输出类型的支持。

**4.5副作用**

在某些情况下，映射化简的用户发现从映射和/或化简操作符生成辅助文件作为附加输出非常方便。我们依靠应用程序编写器使这种副作用原子化和幂等。通常，应用程序写入临时文件，并在文件完全生成后自动重命名该文件。

我们不支持由单个任务生成的多个输出文件的原子两阶段提交。因此，生成具有跨文件一致性要求的多个输出文件的任务应该是确定性的。这一限制在实践中从来不是一个问题。

**4.6 跳过不良记录**

有时用户代码中存在错误，导致映射或化简函数在某些记录上发生决定性崩溃。这些错误会阻止映射化简操作完成。通常的做法是修复错误，但有时这是不可行的；可能错误位于第三方库中，而该库的源代码不可用。此外，有时忽略一些记录是可以接受的，例如在对一个大数据集进行统计分析时。我们提供了一种可选的执行模式，映射化简库会检测哪些记录会导致确定性的崩溃，并跳过这些记录，以便向前推进。

每个工作进程都安装一个信号处理程序，用于捕获分段冲突和总线错误。在调用用户映射或化简操作之前，映射化简库将参数的序列号存储在全局变量中。如果用户代码生成信号，信号处理程序将向映射化简主机发送一个包含序列号的“最后一口气”UDP包。当主机在某个特定记录上发现多个失败时，它指示在它发出下一次重新执行对应的映射或化简任务时应跳过该记录。

**4.7本地执行**

在映射或化简函数中调试问题可能很棘手，因为实际的计算发生在一个分布式系统中，通常在几千台机器上，由主机动态地做出工作分配决策。为了帮助简化调试、分析和小规模测试，我们开发了一个映射化简库的替代实现，它顺序地在本地计算机上执行映射化简操作的所有工作。控件提供给用户，以便计算可以限制到特定的地图任务。用户使用一个特殊的标志来调用他们的程序，然后可以轻松地使用他们认为有用的任何调试或测试工具（例如：gdb）。

**4.8状态信息**

主服务器运行一个内部HTTP服务器并导出一组状态页供用户使用。状态页显示计算的进度，如已完成的任务数、正在进行的任务数、输入字节数、中间数据字节数、输出字节数、处理速率等。这些页还包含指向每个任务生成的标准错误和标准输出文件的链接。用户可以使用这些数据来预测计算需要多长时间，以及是否应该向计算中添加更多的资源。这些页面还可以用于计算何时比预期慢得多。

此外，顶层状态页显示哪些工作线程失败，以及在失败时映射和减少正在处理的任务。当试图诊断用户代码中的错误时，此信息非常有用。

**4.9计数器**

映射化简库提供了一个计数器工具来统计各种事件的发生次数。例如，用户代码可能需要统计处理的总字数或索引的德语文档数等。

为了使用这个工具，用户代码创建一个命名的counter对象，然后在映射和/或化简函数中适当地增加计数器。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");

来自单个工作计算机的计数器值定期传播到主计算机（附带在ping响应上）。车间主从成功的映射和化简任务聚合计数器值，并在映射化简操作完成时将它们返回给用户代码。当前的计数器值也显示在主状态页上，以便人类可以观察实时计算的进度。当聚集计数器值时，主机会消除重复执行同一映射的影响，或减少任务以避免重复计算。（使用备份任务和由于失败而重新执行任务可能会导致重复执行。）

有些计数器值由映射化简库自动维护，例如处理的输入键/值对的数量和生成的输出键/值对的数量。

用户发现counter工具对于检查映射化简操作的行为非常有用。例如，在某些映射化简操作中，用户代码可能希望确保生成的输出对的数量正好等于处理的输入对的数量，或者确保处理的德语文档的部分在处理的文档总数的某个可容忍的分数范围内。

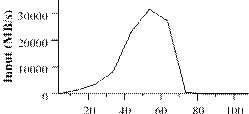
5.性能

在本节中，我们将在大型计算机集群上运行两个计算来度量映射化简的性能。一个计算在大约1TB的数据中搜索特定的模式。另一种计算方法对大约1TB的数据进行排序。

这两个程序代表了映射化简用户编写的大量实际程序的子集——一类程序将数据从一种表示方式转换为另一种表示形式，另一类程序从大数据集中提取少量有趣的数据。

**5.1群集配置**

所有的程序都是在一个由大约1800台机器组成的集群上执行的。每台机器有两个2GHz Intel Xeon处理器，支持超线程，4GB内存，两个160GB IDE



秒

图2：随时间变化的数据传输速率



磁盘和千兆位以太网链路。这些机器被安排在一个两级树形交换网络中，其根部可用的总带宽约为100-200 Gbps。所有的机器都在同一个主机上，因此任何一对机器之间的往返时间不到一毫秒。

在4GB内存中，大约有1-1.5GB被集群上运行的其他任务保留。这些程序是在一个周末的下午执行的，那时CPU、磁盘和网络大多处于空闲状态。

**5.2Grep**

grep程序扫描1010个100字节的记录，搜索一个相对罕见的三字符模式（该模式出现在92337个记录中）。输入被分成大约64MB的部分（M=15000），整个输出被放在一个文件中（R=1）。

图2显示了一段时间内计算的进度。Y轴显示扫描输入数据的速率。随着越来越多的机器被分配到这个映射化简计算中，这个速率逐渐加快，当分配了1764个工人时，这个速率达到峰值，超过30gb/s。当映射任务完成时，速率开始下降，并在计算大约80秒时达到0。整个计算从开始到结束大约需要150秒。这包括大约一分钟的启动开销。开销是由于程序传播到所有工作机，以及延迟与GFS交互以打开1000个输入文件集并获取局部优化所需的信息。

**5.3分类**

分类程序对1010个100字节的记录（大约1tbyte的数据）进行排序。这个程序是以TeraSort基准测试[10]为模型的。

|  |
| --- |
| （a） 正常执行 （b） 没有备份任务 （c）200个任务被终止  图3:分类程序不同执行时间的数据传输率 |

分拣程序由不到50行用户代码组成。三行映射函数从文本行中提取一个10字节的排序键，并将该键和原始文本行作为中间键/值对。我们使用一个内置的恒等式函数作为化简运算符。此函数将中间键/值对不变地传递为输出键/值对。最终排序的输出被写入一组双向复制的GFS文件（即，2tb字节被写入程序的输出）。

如前所述，输入数据被分成64MB（M=15000）。我们将排序后的输出分成4000个文件（R=4000）。分区函数使用密钥的初始字节将其分隔为R块中的一个。

对于这个基准测试，我们的分区函数有关于密钥分布的内置知识。在一般的排序程序中，我们将添加一个pre-pass 映射化简操作，该操作将收集密钥的样本，并使用采样密钥的分布来计算最终排序过程的拆分点。

图3（a）显示了排序程序正常执行的进度。左上角的图表显示了读取输入的速率。该速率的峰值约为13gb/s，并很快消失，因为所有映射任务都在200秒之前完成。注意，输入速率小于grep。这是因为排序映射任务花费大约一半的时间和I/O带宽将中间输出写入本地磁盘。相应的grep中间输出的大小可以忽略不计。

左中图显示了通过网络将数据从映射任务发送到化简任务的速率。当第一个映射任务完成时，这种洗牌就开始了。图中的第一个驼峰是第一批大约1700个化简任务（整个映射化简被分配了大约1700台机器，每台机器一次最多执行一个化简任务）。在计算大约300秒后，第一批化简任务中的一些完成，我们开始为剩余的化简任务重新排列数据。所有的洗牌都是在600秒左右完成的。左下角的图形显示化简任务将排序数据写入最终输出文件的速率。由于机器正忙于对中间数据进行排序，所以在第一个洗牌周期结束和写入周期开始之间有一个延迟。写操作会以大约2-4 GB/s的速度持续一段时间。所有的写操作都在850秒后完成。包括启动开销，整个计算需要891秒。这与TeraSort基准测试的当前最佳报告结果1057秒相似。

需要注意的是：由于我们的局部优化，输入速率高于洗牌速率和输出速率–大多数数据都是从本地磁盘读取的，并绕过带宽相对受限的网络。洗牌率高于输出速率，因为输出阶段写入两个已排序数据的副本（出于可靠性和可用性的原因，我们制作两个输出副本）。我们编写两个副本，因为这是底层文件系统提供的可靠性和可用性机制。如果底层文件系统使用擦除编码而不是复制，则写入数据的网络带宽需求将减少。

**5.4备份任务的效果**

在图3（b）中，我们显示了禁用备份任务的分类程序的执行。执行流程与图3（a）中所示的类似，只是有一个非常长的尾部，几乎没有任何写入活动发生。960秒后，除了5个化简任务外，其他任务都完成了。然而，最后的几个掉队者直到300秒后才结束。整个计算时间为1283秒，所用时间增加了44%。

**5.5机器故障**

在图3（c）中，我们展示了sort程序的一个执行过程，在这个过程中，我们故意在1746个工作进程中的200个进程在计算过程中被杀死。底层集群调度程序立即在这些计算机上重新启动新的工作进程（因为只有进程被终止，所以这些计算机仍然正常工作）。

工人死亡显示为负输入率，因为一些先前完成的地图工作消失了（因为相应的地图工作人员被杀），需要重做。重新执行此地图工作相对较快。整个计算在933秒内完成，包括启动开销（仅比正常执行时间增加5%）。

6 .经验

我们在年编写了映射化简库的第一个版本，2003年2月，并在2003年8月对其进行了重大改进，包括局部性优化、跨工作机任务执行的动态负载平衡等。从那时起，我们对映射化简库在我们处理的各种问题上的广泛应用感到惊喜。它已经被广泛应用于谷歌的各个领域，包括：

•大规模机器学习问题

•谷歌新闻和Froogle产品

•提取用于生成热门查询报告的数据（如Google Zeitgeist）

•为新实验和新产品提取网页属性（例如，从大量网页语料库中提取地理位置以进行本地化搜索），以及

•大规模图形计算

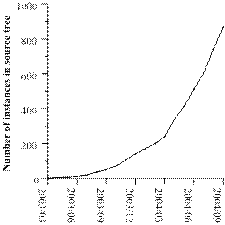


图4:映射化简实例随时间变化

|  |
| --- |
| 工作岗位数29423  平均作业完成时间634秒  机器使用天数79186天 |
| 输入数据读取3288 TB  中间数据生成758 TB输出数据写入193 TB |
| 每个工作的平均工人机器157每个工作的平均工人死亡1.2每个工作的平均映射任务3351每个工作平均减少任务55 |
| 唯一映射实现395唯一化简实现269唯一映射/化简组合426 |

表1:2004年8月运行的映射化简作业



图4显示了随着时间的推移，检入我们的主要源代码管理系统的独立映射化简程序数量的显著增长，从2003年初的0到2004年9月底的几乎900个独立实例。映射化简之所以如此成功，是因为它可以在半小时内编写一个简单的程序，并在上千台机器上高效运行，大大加快了开发和原型设计的周期。此外，它还允许没有分布式和/或并行系统经验的程序员轻松地利用大量资源。

在每个作业结束时，映射化简库会记录作业使用的计算资源的统计信息。在表1中，我们展示了2004年8月Google运行的映射化简作业子集的一些统计数据。

**6.1大规模索引**

到目前为止，我们对映射化简最重要的用途之一是对生产索引系统进行了彻底重写，该系统生成了用于Google web搜索服务的数据结构。索引系统将我们的爬行系统检索到的大量文档作为输入，存储为一组GFS文件。这些文档的原始内容是超过20兆字节的数据。索引过程以5到10个映射化简操作的顺序运行。使用映射化简（而不是以前版本的索引系统中的即席分布式过程）提供了几个好处：

•索引代码更简单、更小、更易于理解，因为处理容错、分布和并行化的代码隐藏在映射化简库中。例如，当使用映射化简表示时，计算的一个阶段的大小从大约3800行的C++代码下降到大约700行。

•映射化简库的性能足够好，我们可以将概念上不相关的计算分开，而不是将它们混合在一起以避免额外的数据传递。这使得更改索引过程变得很容易。例如，在我们的旧索引系统中，一个花费了几个月时间的更改在新系统中只花了几天的时间就实现了。

•索引过程变得更容易操作，因为大多数由机器故障、机器速度慢和网络中断引起的问题都由映射化简库自动处理，无需操作员干预。此外，通过向索引集群中添加新的机器，可以很容易地提高索引过程的性能。

7 .相关工作

许多系统提供了受限的编程模型，并利用这些限制自动并行化计算。例如，在N个处理器上，可以使用并行前缀计算对N个元素数组的所有前缀进行logN时间计算[6，9，13]。映射化简可以被认为是基于我们在大量实际计算中的经验对其中一些模型的简化和升华。更重要的是，我们提供了可扩展到数千个处理器的容错实现。相比之下，大多数并行处理系统只在较小的规模上实现，并将处理机器故障的细节留给程序员。

批量同步编程和一些MPI原语提供了更高层次的抽象，使程序员更容易编写并行程序。这些系统与映射化简的一个关键区别是映射化简利用一个受限的编程模型自动地并行化用户程序，并提供透明的容错能力。

我们的局部性优化从诸如活动磁盘[12,15]之类的技术中获得灵感，在这些技术中，计算被推到靠近本地磁盘的处理元素中，以减少通过I/O子系统或网络发送的数据量。我们运行在有少量磁盘直接连接的商品处理器上，而不是直接在磁盘控制器处理器上运行，但一般的方法是相似的。

我们的备份任务机制类似于夏洛特系统中使用的急切调度机制。简单急切调度的缺点之一是，如果一个给定的任务导致重复失败，整个计算都无法完成。我们用跳过坏记录的机制修复了这个问题的一些实例。

映射化简实现依赖于一个内部集群管理系统，该系统负责在大量共享机器上分发和运行用户任务。虽然不是本文的重点，但集群管理系统在精神上与其他系统（如Condor）相似。

作为映射化简库一部分的排序工具在操作上与现在的Sort类似。源机器（映射工人）对要排序的数据进行分区，并将其发送给R 化简 工人中的一个。每个化简 工人对其数据进行本地排序（如果可能，在内存中）。当然，现在Sort没有用户定义的映射和化简函数，这使得我们的库可以广泛应用。

River提供了一个编程模型，其中进程通过在分布式队列上发送数据来相互通信。与映射化简一样，River系统试图提供良好的平均情况下的性能，即使存在由异构硬件或系统扰动引起的不一致性。River通过精心安排磁盘和网络传输来实现这一点，以实现均衡的完成时间。映射化简有不同的方法。通过限制编程模型，映射化简框架能够将问题划分为大量细粒度的任务。这些任务在可用的工作线程上动态调度，以便更快的工人处理更多的任务。受限编程模型还允许我们在接近作业结束时调度任务的冗余执行，这大大减少了在存在非一致性（例如速度慢或卡住的工人）的情况下的完成时间。

BAD-FS与映射化简有着非常不同的编程模型，而且与映射化简不同，它的目标是在广域网中执行作业。然而，有两个基本的相似之处。（1） 两个系统都使用冗余执行来从故障导致的数据丢失中恢复。（2） 两者都使用位置感知调度来减少通过拥塞的网络链路发送的数据量。

TACC[7]是一个旨在简化高可用性网络服务构建的系统。与映射化简一样，它依赖于重新执行作为实现容错的机制。

8 .结论

映射化简编程模型已经在Google成功地用于许多不同的目的。我们把这次成功归因于几个原因。分布式系统的并行化、局部性、并行性、局部性和并行性是程序员的首要考虑。第二，各种各样的问题都很容易用映射化简计算来表达。例如，映射化简用于为Google的生产性we搜索服务生成数据，用于排序、数据挖掘、机器学习和许多其他系统。第三，我们开发了映射化简的实现，它可以扩展到由数千台机器组成的大型机器集群。该实现有效地利用了这些机器资源，因此适用于在Google遇到的许多大型计算问题。

我们从这项工作中学到了一些东西。首先，通过对编程模型的限制，可以方便地并行和分布计算，并使这些计算具有容错性。其次，网络带宽是稀缺资源。因此，我们系统中的许多优化都是为了减少通过网络发送的数据量：局部性优化允许我们从本地磁盘读取数据，将中间数据的一个副本写入本地磁盘可以节省网络带宽。第三，冗余执行可以用来减少慢速机器的影响，并处理机器故障和数据丢失。

致谢:

Josh Levenberg根据他使用映射化简的经验和其他人对增强的建议，在修改和扩展用户级映射化简 API方面发挥了重要作用。映射化简从Google文件系统读取输入并将输出写入Google文件系统[8]。我们要感谢Mohit Aron、Howard Gobioff、Markus Gutschke、David Kramer、Shun Tak Leung和Josh Redstone在开发GFS方面所做的工作。我们还要感谢Percy Liang和Olcan Sercinoglu在开发映射化简使用的集群管理系统方面所做的工作。mikeburrows、wilsonhsieh、Josh Levenberg、sharonperl、robpike和debbywallach对本文的早期草稿提供了有益的评论。匿名的OSDI评论员和我们的shepherd，Eric Brewer，提供了许多有用的建议，指出了论文可以改进的地方。最后，我们感谢Google工程组织中映射化简的所有用户提供了有用的反馈、建议和bug报告。

工具书类

[1] Andrea C.Arpaci Dusseau，Remzi H.Arpaci Dusseau，David E.Culler，Joseph M.Hellerstein和David A.Patterson。工作站网络上的高性能分拣。1997年ACM SIGMOD国际数据管理会议记录，图森，亚利桑那州，1997年5月。

[2] Remzi H.Arpaci Dusseau，Eric Anderson，Noah Treuhaft，David E.Culler，Joseph M.Hellerstein，David Patterson和Kathy Yelick。使用River的集群I/O：使fast案例变得常见。第六届并行和分布式系统输入/输出研讨会（IOPADS’99），第10-22页，乔治亚州亚特兰大，1999年5月。

[3] 阿拉什·巴拉特罗、迈赫迈特·卡拉尔、兹维·凯德姆和彼得·怀科夫。夏洛特：网上的元计算。1996年第九届并行与分布式计算系统国际会议论文集。

[4] 路易斯巴罗佐，杰弗里迪安，乌尔斯霍尔兹尔。一个星球的网络搜索：谷歌集群架构。IEEE Micro，23（2）：22–28，2003年4月。

[5] 西娅·拉斯本特，约翰·里玛齐，约翰·拉普·本特。批处理感知分布式文件系统中的显式控制。2004年3月，网络系统设计与实现第一届USENIX研讨会论文集。

[6] 盖伊·布莱洛奇。作为原始并行操作进行扫描。IEEE计算机事务，C-38（11），1989年11月。

[7] 阿曼多·福克斯、史蒂芬·D·格里布尔、亚丁·查瓦特、埃里克·A·布鲁尔和保罗·高蒂尔。基于集群的可扩展网络服务。第16届ACM操作系统原理研讨会论文集，第78-91页，法国圣马洛，1997年。

[8] 桑杰·格马沃特、霍华德·戈比夫和梁顺德。谷歌文件系统。第19届操作系统原理研讨会，第29-43页，乔治湖，纽约，2003年。

[9] 戈拉奇。系统有效地并行扫描和其他列表同态。在L.Bouge，P.Fraignaud，A.Mignotte和Y.Robert编辑，96年欧洲标准普尔。并行处理，计算机科学1124课堂讲稿，第401-408页。斯普林格·韦拉格，1996年。

[10] 吉姆·格雷。排序基准主页。http://research.microsoft.com/barc/SortBenchmark/。

[11] 威廉·格罗普、尤因·罗斯克和安东尼·斯凯勒姆。使用MPI：带有消息传递接口的可移植并行编程。麻省理工学院出版社，剑桥，马萨诸塞州，1999年。

[12] L.Huston、R.Sukhankar、R.Wickremesinghe、M.Satyanarayanan、G.R.Ganger、E.Riedel和A.Ailamaki。Diamond：交互式搜索中早期丢弃的存储架构。2004年USENIX文件和存储技术快速会议记录，2004年4月。

[13] 理查德·E·拉德纳和迈克尔·J·费舍尔。并行前缀计算。ACM杂志，27（4）：831-8381980。

[14] 迈克尔O拉宾。为安全、负载平衡和容错而有效地分散信息。ACM杂志，36（2）：335-3481989。

[15] 埃里克·里德尔、克里斯托斯·法卢索斯、加思·A·吉布森和大卫·纳格尔。用于大规模数据处理的活动磁盘。IEEE计算机，第68-74页，2001年6月。

[16] 道格拉斯·塞恩、托德·坦南鲍姆和米隆·利夫尼。分布式计算在实践中的应用：秃鹰的经验。并发与计算：实践与经验，2004。

[17] L.G.勇敢者。并行计算的桥接模型。

ACM的通信，33（8）：103–1111997年。

[18] 吉姆·怀利。sport：如何快速地对一个tb字节进行排序。http://alme1.almaden.ibm.com/cs/spsort.pdf。

词频

本节包含一个程序，该程序统计命令行中指定的一组输入文件中每个唯一单词的出现次数。

#include "映射化简/映射化简.h"

// User&apos;s 映射 function class WordCounter : public 映射per {

public:

virtual void 映射(const 映射Input& 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\_映射PER(WordCounter);

// User&apos;s 化简 function class Adder : public 化简r {

virtual void 化简(化简Input\* 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\_化简R(Adder);

int main(int argc, char\*\* argv) {

ParseCommandLineFlags(argc, argv);

映射化简Specification spec;

// Store list of input files into "spec" for (int i = 1; i < argc; i++) {

映射化简Input\* input = spec.add\_input(); input->set\_format("text"); input->set\_filepattern(argv[i]); input->set\_映射per\_class("WordCounter");

}

// Specify the output files:

    // /gfs/test/freq-00000-of-00100

    // /gfs/test/freq-00001-of-00100

    //    ...

映射化简Output\* out = spec.output(); out->set\_filebase("/gfs/test/freq"); out->set\_num\_tasks(100); out->set\_format("text"); out->set\_化简r\_class("Adder");

// Optional: do partial sums within 映射 // 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\_映射\_megabytes(100); spec.set\_化简\_megabytes(100);

// Now run it

映射化简Result result; if (!映射化简(spec, &result)) abort();

// Done: &apos;result&apos; structure contains info

// about counters, time taken, number of // machines used, etc.

return 0;

}