叠前Kirchhoff时间偏移（PKTM）算法在Hadoop和Spark上的实现

杨晨, 唐杰

摘要—近年来，随着信息技术的飞速发展，各种网络应用带来了数据规模的高速增长，使得大数据迅速发展成为科技界和企业界甚至世界各国政府关注的热点。为了满足海量数据存储和分析需求，需要使用大量计算机协同工作共同完成空前的复杂任务。相较于传统的数据，人们将大数据的特征总结为5个V，即体量大（Volume）、速度快（Velocity）、模态多（Variety）、难辨识（Veracity）和价值密度低（Value）。人类已经进入了大数据时代。Hadoop 作为一个开源的分布式计算系统，具有高容错性、高扩展性和高可靠性，它允许用户在廉价的机器上部署Hadoop集群。随着Hadoop的发展，部署在Hadoop上的程序与日俱增。然而，Hadoop依然有网络传输和磁盘读写等瓶颈，于是又产生了基于内存模型的分布式计算系统——Spark。在本文中，我们首先介绍Hadoop和Spark，以及PKTM 的一些相关知识；然后，我们会介绍下PKTM算法在Hadoop和Spark 框架下的实现；最后，我们对Hadoop和Spark上的PKTM 算法进行实验分析和总结。

关键字—大数据, Hadoop, Spark, PKTMl, MapReduce, Yarn, RDD.

1 介绍

随着互联网的飞速发展，特别是近年来随着社交网络、物联网、云计算以及多种传感器的广泛应用，以数量庞大，种类众多，时效性强为特征的非结构化数据不断涌现，数据的重要性愈发凸显，传统的数据存储、分析技术难以实时处理大量的非结构化信息，大数据概念应运而生。虽然计算机存储设备的低廉化以及高速的网络到来在一定程度上减缓了大数据带来的挑战，但数据的增长远远的大于硬件设备的增长，因此传统的大数据处理模型受到了很大的挑战。为了对新时代大数据进行有效的处理，应运而生了一系列大数据分布式处理框架，主要有Hadoop，Spark等。本文主要分为以下几个部分介绍：第二章，我们介绍了一些Hadoop和Spark分布式处理框架的相关知识；第三章，我们列出了一系列的相关工作；第四章，我们介绍了PKTM 算法在Hadoop和Spark上的实现；第五章，我们在Hadoop和Spark平台上进行一系列实验；最后一章我们对算法进行总结。

2 背景

在本节中，我们首先在2.1中介绍Hadoop框架；然后在2.2中介绍Spark框架；在2.3节中介绍了Hadoop和Spark对比；最后在2.4中介绍了PKTM算法。

2.1 Hadoop框架

Hadoop是一个分布式系统基础架构，由Apache 基金开发。Apache Hadoop是一款支持数据密集型分布式应用并以Apache 2.0许可协议发布的开源软件框架。它支持在商品硬件上构建的大型集群上运行的应用程序。Hadoop是一个基于Java 语言构建的开源分布式框架，它可以被搭建在廉价的机器上进行各种不同的通用目的海量数据的分析。Hadoop框架透明地为应用提供可靠性和数据移动。它实现了名为MapReduce的编程范式：应用程序被分割成许多小部分，而每个部分都能在集群中的任意节点上执行或重新执行。此外，Hadoop还提供了分布式文件系统，用以存储所有计算节点的数据，这为整个集群带来了非常高的

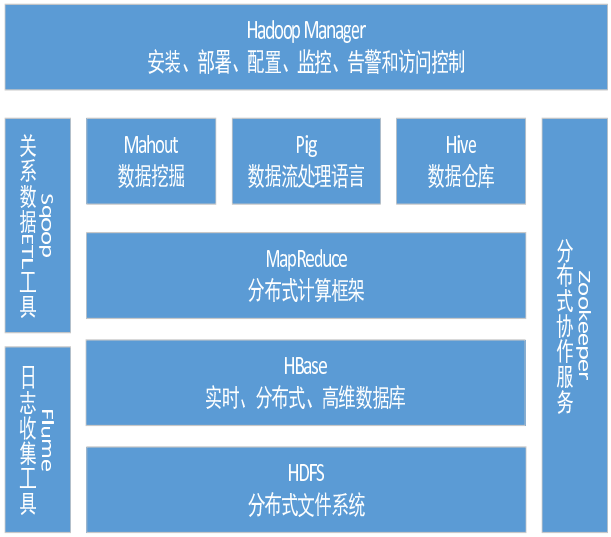

图 1: Hadoop家族架构

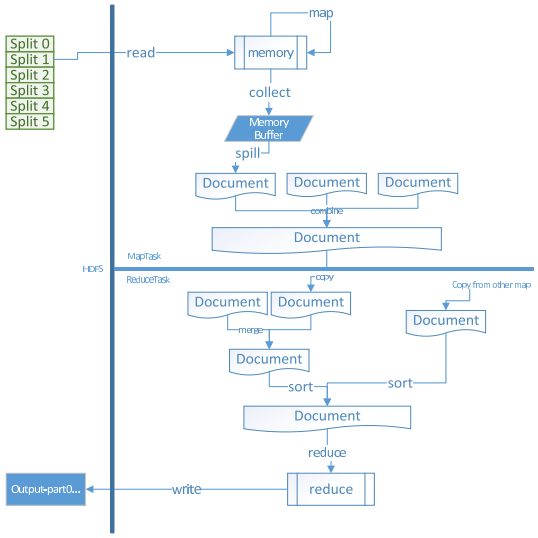

图 2: MapReduce程序流程

带宽。MapReduce和分布式文件系统HDFS的设计，使得整个框架能够自动处理节点故障，它使应用程序运用成千上万的独立计算的电脑来处理PB级的数据。因此，Apache Hadoop 的核心设计主要是：MapReduce和HDFS。Hadoop还包括了很多子项目，包括HBase，Hive，Mahout，Sqoop，Zookeeper，Avro等。Hadoop的主要架构图如图1 所示。

2.1.1 MapReduce

MapReduce 是一个最先由Google 提出的分布式计算软件架构，它可支持大数据量的分布式处理。这个架构最初起源于函数式程式的Map 和Reduce两个函数组成，但它们在MapReduce 框架中的应用和原来的使用上大相径庭。MapReduce 框架中，用户的程序总是被分成Map端和Reduce 端，用户只需要填写Map和Reduce 函数，提交应用程序到Hadoop 系统端，系统会自动切分数据，分布式运行提交的程序。Hadoop上MapReduce程序运行流程如图2所示。

MapReduce是一个高性能的批处理分布式计算框架，用于对海量数据进行并行分析和处理。与传统的数据仓库和分析技术相比，MapReduce 适合处理各种类型的数据，包括结构化、半结构化和非结构化数据。数据量在TB和PB级别，在这个量级上，传统方法通常已经无法处理数据。MapReduce 将分析任务分为大量的并行Map 任务和Reduce任务两类。Map和Reduce任务可以运行在多个服务器上。MapReduce适合处理的任务：

(1) 复杂的数据：业务数据不能适合行列的数据结构。数据可能来源于多种格式：多媒体数据、图像数据、文本数据、实时数据、传感器数据等等。当有新的数据来源时，可能会有新的数据格式的出现。MapReduce可以存放和分析各种原始数据格式。

(2) 超大规模数据：很多公司仅仅因为数据存放成本过高就放弃了很多有价值的数据。新的数据来源使得问题更为严重，新的系统和用户带来比以往更多的数据。Hadoop的创新构架使用低成本的常规服务器存储和处理海量的数据。

(3) 新的分析手段：海量复杂数据分析需要使用新的方法。新的算法包括自然语言分析、模式识别等。只有Hadoop的架构才能方便高效地使用新的算法来处理和分析海量数据。

MapReduce框架的核心优势：

a) 高度可扩展，可动态增加/削减计算节点，真正实现弹性计算。

b) 高容错能力，支持任务自动迁移、重试和预测执行，不受计算节点故障影响。

c) 公平调度算法，支持优先级和任务抢

占，兼顾长/短任务，有效支持交互任务。

d) 就近调度算法，调度任务到最近的数据节点，有效降低网络带宽。

e) 动态灵活的资源分配和调度，达到资源利用最大化，计算节点出现闲置和过载的情况；同时支持资源配额管理。

目前MapReduce主要有两个版本：MRv1和MRv2（Yarn）。

MRv1是第一代MapReduce计算框架。它由两部分组成：编程模型和运行时环境。它的基本编程模型是将问题抽象成Map和Reduce两个阶段。其中，Map阶段将输入数据解析成< $key, value$ >键值对形式，并行调用map处理后，再以< $key, value$ >的形式输出Reduce节点进行处理；Reduce阶段则将key相同的value进行归约处理，并将最终的结果写到HDFS上。MRv1的核心功能主要是：JobTracker和TaskTracker。MRv1的框架结构如图3所示.

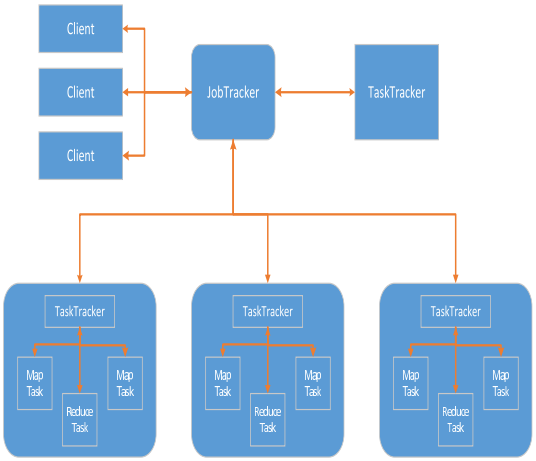

图 3: MRv1框架

1) JobTracker是整个MapReduce计算框架中的主服务，相当于集群的“管理者”，负责整个集群的作业控制和资源管理。在Hadoop内部，每个应用程序被表示成一个作业，每个作业又被进一步分成多个任务，而JobTracker的作业控制模块则负责作业的分解和状态监控。其中，最重要的是状态监控，主要包括TaskTracker的状态监控、作业状态监控和任务状态监控等。其主要作用有两个：容错和为任务调度提供决策依据。一方面，通过状态监控，JobTracker能够及时发现存在异常或者出现故障的TaskTracker、作业或者任务，从而启动相应的容错机制进行处理；另一方面，由于JobTracker保存了作业和任务的近似实时运行信息，这些可用于任务调度时进行任务选择的依据。资源管理模块的作用是通过一定的策略将各个节点上的计算资源分配给集群中的任务。它由可插拔的任务调度器的地方，用户可根据自己的需要编写相应的调度器。

JobTracker是一个后台服务进程，启动之后，会一直监听并接收来自各个TaskTracker发送的心跳信息，这里面包含节点资源使用情况和任务运行情况等信息。JobTracker会将这些信息统一保存起来，并根据需要为TaskTracker分配新任务。JobTracker的主要功能就是作业控制和资源管理。

2) TaskTracker是Hadoop集群中运行于各个节点上的服务，它扮演着“通信枢纽”的角色，是JobTracker与Task之间的“沟通桥梁”：一方面，它从JobTracker端接收并执行各种命令，比如运行任务、提交任务、杀死任务等；另一方面，它将本节点上的各个任务状态通过周期性心跳汇报给JobTracker。TaskTracker与JobTracker和Task之间采用了RPC协议进行通信。对于TaskTracker和JobTracker而言，它们之间采用InterTrackerProtocol协议，其中，JobTracker扮演RPC Server的角色，而TaskTracker扮演RPC Client角色；对于TaskTracker与Task而言，它们之间采用TaskUmbilicalProtocol协议，其中，TaskTracker扮演RPC Server的角色，而Task扮演RPC Client的角色。TaskTracker主要实现了两个功能：汇报心跳和执行命令。

随着数据量的高速增长和新型应用的出现，MRv1在扩展性、可靠性、资源利用率和多框架支持等方面暴露出了明显的不足，由此诞生

了下一代MapReduce 框架（MRv2）。MRv1的主要问题有以下几个方面：

1) 扩展性差：在MRv1中，JobTracker同时具备了资源管理和作业控制两个功能，这成为系统的一个最大的瓶颈，严重制约了Hadoop集群的扩展。

2) 可靠性差：MRv1采用了master/slave结构，其中，master存在单点故障问题，一旦它出现故障，将导致整个集群不可用。

3) 资源利用率低：MRv1采用了基于槽位的资源分配模型。槽位是一种粗粒度的资源划分单位，通常一个任务不会用完槽位对应的资源，且其他任务也无法使用这些空闲资源。此外，Hadoop将槽位分为Map slot和Reduce slot两种，且不允许它们之间共享，这常常会导致一种槽位资源紧张而另外一种闲置的情况出现（比如一个作业刚刚提交时，只会运行Map Task，此时Reduce slot闲置）。

4) 无法支持多种计算框架：随着互联网的高速发展，MapReduce这种基于磁盘的离线计算框架已经不能满足应用要求，从而出现了一些新的计算框架，包括内存计算框架、流式计算框架和迭代式计算框架等，而MRv1不能支持多种计算框架并存。

下一代MapReduce框架的基本设计思想是将JobTracker的两个主要功能，即资源管理和作业控制（包括作业监控、容错等），分拆成两个独立的进程：资源管理进程和作业控制进程。资源管理进程是与具体应用程序无关的模块，它负责整个集群的资源（内存、CPU、磁盘等）管理；而作业控制进程是直接与应用程序相关的模块，且每个作业控制进程只负责管理一个作业。这样，通过将原有JobTracker中与应用程序相关和无关的模块分开，不仅减轻了JobTracker负载，也使得Hadoop支持更多的计算框架。

随着互联网的高速发展，基于数据密集型应用的计算框架不断出现。从支持离线处理的MapReduce，到支持在线处理的Storm，从迭代计算框架Spark 到流式处理框架S4....各种框架诞生于不同的公司或者实验室。它们各有所长，各自解决了某一类应用问题，而在大部分互联网公司中，这几种框架可能同时被采用。公司一般希望将所有这些框架部署到一个公共的资源中，让它们共享集群的资源，这就产生了资源统一管理与调度平台的典型代表Apache Yarn。

Yarn是Apache的下一代MapReduce框架，它的基本设计思想是将JobTracker拆分成两个独立的服务：一个全局的资源管理器ResourceManager负责对各个NodeManager上的资源进行统一管理和调度。当用户提交一个应用程序时，需要提供一个用于跟踪和管理这个程序的ApplicationMaster。它负责向ResourceManager 申请资源，并要求NodeManager启动可以占用一定资源的任务。由于不同的ApplicationMaster分布在不同的节点上，因此它们之间不会相互影响。Yarn主要由三部分组成：ResourceManager、NodeManager、ApplicationMaster。Yarn的结构如图4所示。

1) ResourceManager是一个全局的资源管理器，负责整个系统的资源管理和分配。它主要由两个组件构成：调度器（Scheduler）和应用管理器（ASM）。调度器根据容量、队列等限制条件，将系统中的资源分配给各个正在运行的应用程序。该调度器只是个单纯的调度器，不再从事任何与应用程序相关的工作。调度器仅根据各个应用的资源需求进行资源分配，而资源分配单位用一个抽象概念“资源容器”（Container）表示。Container是一个动态资源分配单位，它将内存、CPU、磁盘、网络等资源封装在一起，从而限定每个任务使用的资源量。调度器是一个可插拔的组件，用户可以根据自己的需要设计新的调度器；应用程序管理器（ASM）负责整个系统中所有应用程序，包括应用程序提交、与调度器协商资源以启动ApplicationMaster、监控ApplicationMaster运行状态并在失败时重新启动它等。

2) NodeManager是每个节点上的资源和任务管理器。一方面，它会定时地向RM汇报

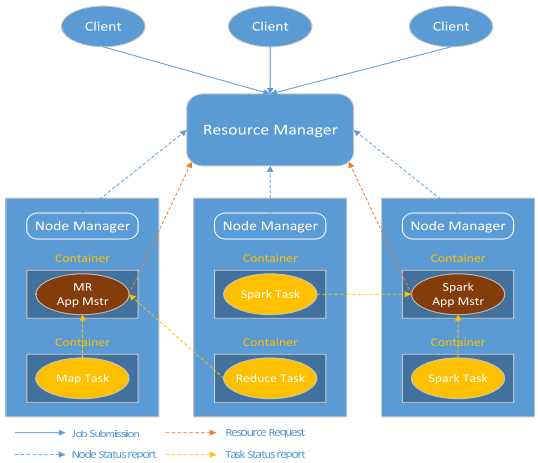

图 4: Yarn架构

本节点上的资源使用情况和各个Container的运行状态；另一方面，它会接收并处理来自AM的任务启动/停止等各种请求。

3) 用户提交的每个应用程序均包含一个AM。它实际上是一个简化版的JobTracker，主要功能包括：与RM 调度器协商以获取资源、与NM通信以启动/停止任务、监控所有任务的运行状态，并在任务运行失败时重新为任务申请资源以重启任务。

2.1.2 HDFS

HDFS （Hadoop Distributed File System）是Apache Hadoop 项目的一个子项目，是一个高度容错的分布式文件系统，设计用于在低成本硬件上运行。HDFS 提供高吞吐量应用程序数据访问功能，适合带有大型数据集的应用程序。HDFS作为一个分布式文件系统，具有高容错的特点，它可以部署在廉价的通用硬件上，提供高吞吐率的数据访问，适合那些需要处理海量数据集的应用程序。HDFS 没有遵循可移植操作系统接口（POSIX）要求，不支持“ls”或“cp”这样的标准UNIX命令，也不支持如fopen和fread这样的文件读写方法，二是提供了一套特有的、基于Hadoop抽象文件系统的API，支持以流的形式访问文件系统的数据。HDFS的主要特性包括：

(1) 支持超大文件。超大文件在这里指的是几百MB，几百GB甚至几百TB大小的文件，一般来说，一个Hadoop文件系统会存储T、P级别的数据。Hadoop需要能够支持这种级别的大文件。

(2) 检测和快速应对硬件故障。在大量通用硬件平台上构建集群时，故障，特别是硬件故障是常见的问题。一般的HDFS系统是由数百台甚至上千台存储着数据文件的服务器组成，这么多的服务器意味着高故障率。因此故障检测和自动恢复是HDFS的一个设计目标。

(3) 流式数据访问。HDFS处理的数据规模都比较大，应用一次需要访问大量的数据。同时，这些应用一般是批量处理，而不是用户交互式处理。HDFS 使应用程序能够以流的形式访问数据集，注重的是数据的吞吐量，而不是数据访问的速度。

(4) 简化的一致模型。大部分的HDFS程序操作文件时需要一次写入，多次读取。在HDFS中，一个文件一旦经过创建、写入、关闭后，一般就不需要修改。这样简单的一致性模型，有利于提供高吞吐量的数据访问模型。

为了支持流式数据访问和存储超大文件，HDFS引入了一些比较特殊的设计，在一个全配置的集群上，“运行HDFS”意味着在网络分布的不同服务器上运行一些守护进程，这些进程有各自的特殊角色，并相互配合，一起形成一个分布式文件系统。HDFS采用了主从体系结构，名字节点NameNode、数据节点DataNode和客户端Client 是HDFS中3个重要的角色。NameNode是HDFS主从结构中主节点上运行的主要进程，它指导主从结构中的从节点，DataNode执行底层的I/O任务。NameNode 管理着文件系统的Namespace。它维护着文件系统树以及文件树中所有文件和文件夹的元数据。管理这些信息的文件有两个，分别是Namespace镜像文件和操作日志文件，这些信息被Cache 在RAM中，当然，这两个文件也会被持久化存储在本地硬盘。NameNode记录着每个文件中各个块所在的数据节点的位置信息，但是他并不持久化

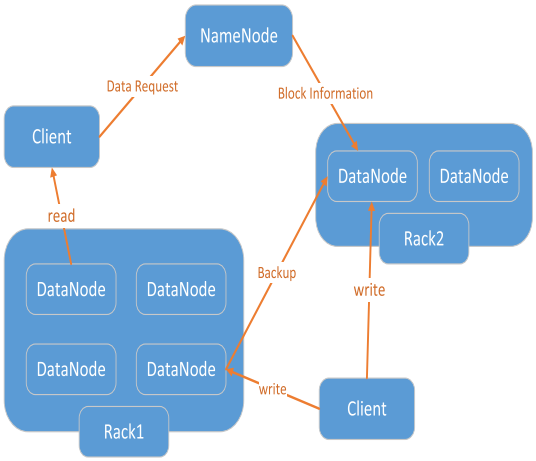

图 5: HDFS结构

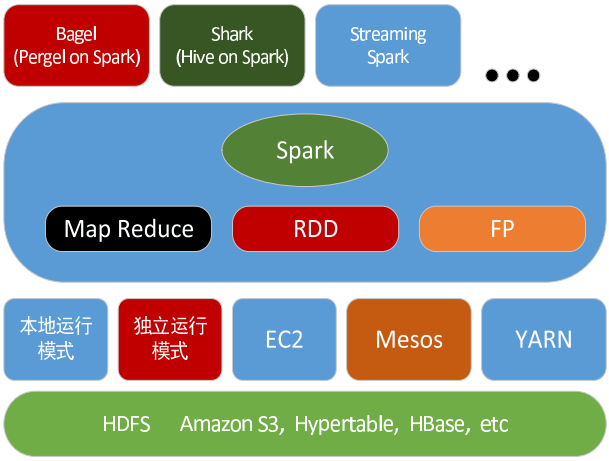

图 6: Spark架构

存储这些信息，因为这些信息会在系统启动时从数据节点重建；DataNode是文件系统的工作节点，他们根据客户端或者NameNode 的调度存储和检索数据，并且定期向NameNode发送他们所存储的块的列表；Client 代表用户与NameNode和DataNode交互来访问整个文件系统。Client提供了一些列的文件系统接口，因此我们在编程时，几乎无须知道DataNode和NameNode，即可完成我们所需要的功能。HDFS的结构图如图5所示。

2.2 Spark框架

Spark是UC Berkeley AMP lab所开源的类Hadoop MapReduce的通用的并行计算框架，Spark基于Map Reduce 算法实现的分布式计算，拥有Hadoop MapReduce所具有的优点；但不同于MapReduce的是Job中间输出的结果可以保存在内存中，从而不再需要读写HDFS，因此Spark 能更好地适用于数据挖掘与机器学习等需要迭代的MapReduce 的算法。Spark的核心组件是RDD（Resilient Distributed Datasets）。Spark的架构图如图6所示。

2.2.1 RDD弹性分布式数据集

RDD是一个容错的、并行的数据结构，可以让用户显式地将数据存储到磁盘和内存中，并能控制数据的分区。同时，RDD还提供了一组丰富的操作来操作这些数据。RDD作为数据结构，本质上是一个只读的分区记录集合。一个RDD可以包含多个分区，每个分区就是一个dataset片段。RDD可以相互依赖。如果RDD的每个分区最多只能被一个Child RDD的一个分区使用，则称之为narrow dependency；若多个Child RDD分区都可以依赖，则称之为wide dependency。不同的操作依据其特性，可能会产生不同的依赖。RDD的依赖模型如图7所示。Spark之所以将依赖分为narrow与wide，基于两点原因。首先，narrow dependencies可以支持在同一个cluster node上以管道形式执行多条命令；其次，则是从失败恢复的角度考虑，narrow dependencies 的失败恢复更有效，因为它只需要重新计算丢失的parent partition即可，而且可以并行地在不同节点进行重计算，而wide dependencies牵涉到RDD各级的多个parent partitions。

RDD本质上是一个内存数据集，在访问RDD时，指针只会指向与操作有关的部分。RDD将操作分为两类：transformation和action。无论执行多少次transformation 操作，RDD都不会真正执行运算，只有当action 操作被执行时，运算才会触发。而在RDD内部实现机制中，底层接口则是基于迭代器的，从而使得数据访问变得更高效，也避免了大量中间结果对内存的消耗。

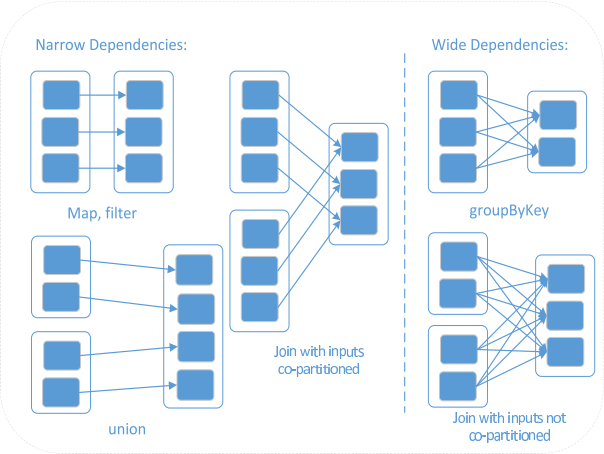

图 7: RDD dependencies

RDD是Spark最核心的东西，它表示已被分区、不可变的并能够被并行操作的数据集合，不同的数据集格式对应不同的RDD实现。RDD必须是可序列化的。RDD 可以cache到内存中，每次对RDD数据集的操作之后的结构，都可以存放到内存中，下一个操作可以直接从内存中输入，省去了MapReduce大量的磁盘I/O操作。这对迭代运算比较常见的机器学习算法，交互式数据挖掘来说，效率大大提升。总结来说，RDD有一下几个特点：

(1) 它是在集群节点上的不可变的，已分区的集合对象。

(2) 通过并行转换的方式来创建如（map，filter，join，etc）。

(3) 失败自动重建。

(4) 可以控制存储级别（内存、磁盘等）来进行重用。

(5) 必须是可序列化的。

(6) 是静态类型的。

RDD的生成有两种创建方式：

(1) 从Hadoop文件系统（或与Hadoop兼容的其他存储系统）输入（例如HDFS）创建。

(2) 从父RDD转换得到新RDD。

2.3 Hadoop vs Spark

Spark与Hadoop的对比：尽管Hadoop适合大多数批处理工作负载，而且在大数据时代成为企业的首选技术，但由于以下几个限制，它对一些工作负载并不是最优选择：缺少对迭代的支持；需要将中间数据存在硬盘上以保持一致性，因此会有比较高的延迟。在Spark 集群中，有两个重要的元素，即driver和worker。driver程序是应用逻辑执行的起点，而多个worker用来对数据进行并行处理。尽管不是强制的，但数据通常是与worker搭配，并在集群内的同一套机器中进行分区。在执行阶段，driver程序会将code/closure传递给worker 机器，同时相应分区的数据将进行处理。数据会经历转换的各个阶段，同时尽可能地保持在同一分区之内。执行结束后，worker会将结果返回到driver程序。Spark 上driver和worker运行模式如图8所示。总的来说Hadoop 和Spark对比有一下几个方面：

(1) Spark的中间数据放到内存中，对于迭代运算效率更高。Spark更适合于迭代运算比较多的ML 和DM运算，因为在Spark中，有RDD概念。

(2) Spark比Hadoop更通用。Spark提供的数据集操作类型有很多种，不像Hadoop只提供Map和Reduce 两种操作。然而由于RDD特性，Spark不适用那种异步细粒度更新状态的应用。

(3) 容错性。在分布式数据集计算时通过checkpoint 来实现容错，而checkpoint 有两种方式，一个是checkpoint data，一个是logging the updates。用户可以控制采用哪种方式来实现容错。

(4) 可用性。Spark通过提供丰富的Scala，Java，Python及交互式Shell来提高可用性。

2.4 Kirchhoff

Kirchhoff叠前时间偏移是地震数据处理中最耗时的常用模块之一。Kirchhoff 叠前时间偏移假定已知地震道数据t_0，对于所有的反射点，从激发点到接收点的传输时间$T_{SR} = t_0$，已知地震道数据的激发点和接收点的坐标，在传输时间T_{SR}内，反射点的轨迹如图9所示。

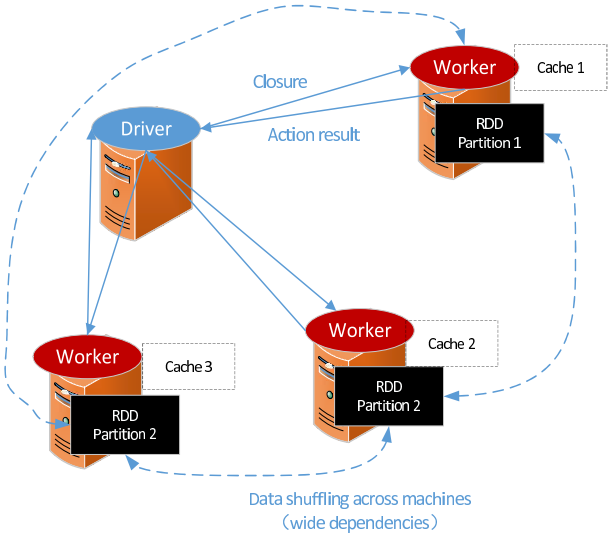

图 8: Spark driver和worker架构

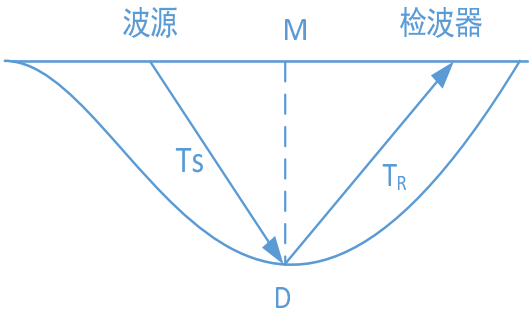

图 9: 地震成像示意图

利用叠前时间偏移可以有效地解决交叉地层速度的矛盾性问题，通过比较图中常规速度谱和叠前时间偏移速度谱，可以看到经过处理后，速度谱能量更加集中，速度拾取矛盾也可以得到有效解决。Kirchhoff CPU算法如Algorithm 1所示。 Kirchhoff算法流程和文件处理流程如图10所示。

3 相关工作

目前已经有一些Kirchhoff算法并行计算的相关工作：首先我们介绍PKTM在MapReduce上的实现；然后介绍PKTM在GPU上的实现。

(1) MapReduce： 这篇文章[1]介绍了PKTM 在Hadoop上的实现，它按照输入道划分mapper，每个mapper包含一系列输入道数据，然后计算输出道，shuffle数据，并把输出道数据发送到reduce 端，

Algorithm 1 Kirchhoff算法

```
1: procedure KIRCHHOFF(inputtraces)
2:     for all input traces do
3:         read input trace
4:         filter input trace
5:         for all output traces within aperture do
6:             for all output trace contributed samples do
7:                 compute travel time
8:                 compute amplitude correction
9:                 select input sample and filter
10:                accumulate input sample contribution into output sample
11:            end for
12:        end for
13:    end for
14:    dump output volume
15: end procedure
```

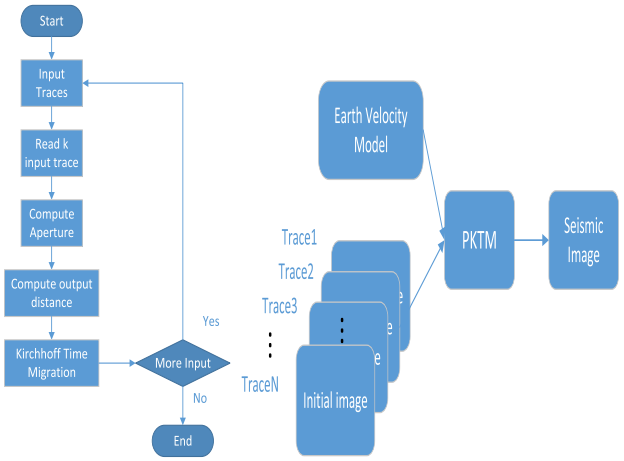

图 10: PKTM

在reduce端合并输出道，并把结果输出到HDFS。该程序没有考虑集群的环境，当集群环境变化时，程序需要做很大的改动。还有篇文章[2]同样介绍了PKTM运行在MapReduce框架下，该程序使用了同样的划分方法划分输入道，然后在mapper中计算，并把结果发送到reduce端，但是该程序

没有考虑网络带宽的问题，发送的中间数据特别巨大，大大的影响了程序的性能。总的来说，MapReduce上的PKTM实现，主要的问题有三个方面：

a) 应对数据量的不同，性能变化很大。一些程序在小量的数据上运行性能很好，但是在大数据集上运行性能会变的很差；还有就是随着集群机器的扩展，性能没有稳步的提升。

b) 网络传输量的问题。当数据量变大时，不可避免的一个问题是中间数据量的增大，这就给网络传输带来了一定的负担。不同节点之间的传输性能会严重影响程序的性能。

c) HDFS读写问题。程序的数据保存在HDFS 上，这样就保证了所有节点可以共享所有数据，HDFS还提供了强大的容错能力，可以帮助恢复失败的任务，但是HDFS的读写性能有一定的瓶颈，不支持并行读写，因此解决了HDFS的读写问题可以大大的提高程序的性能。

(2) GPU：这篇文章[3]介绍了PKTM在GPGPU上的实现，它用multi-GPUs来计算数据，该程序大大加速了PKTM算法相对于CPU算法。主要是利用GPU 的成百上千的线程来计算数据。大大的提高了计算的并行性。但当数据量特别大时，GPU的显存容量很小，GPU和CPU就需要不断的来交换数据，以及CPU不断的读取数据。这是GPU的一个瓶颈。另一篇文章[4]也是介绍了GPU上的PKTM的实现，它提供了大概20倍的加速相对于CPU计算，但是它的CPU I/O读写和循环控制过载。总的来说，GPU上的PKTM实现主要有一下几方面问题：

a) 显存容量问题。虽然GPU有成百上千的线程用于计算数据，但是GPU的显存容量太小，是它的一个瓶颈，计算时需要等待数据从内存交换到显存，在大数据的情况下这就大大的影响了程序的性能。

b) GPU同步问题。在多个GPU的情况下，GPU 的同步显然就是个严重的问题，如果单纯的只使用一个GPU来加速，由于显存容量以及系统的资源问题，加速的效果大大的弱化，因此使用多GPU加速，GPU之间的同步成为了程序的一个瓶颈，怎样很好的去同步计算的数据依然是现在的一个问题。

c) 任务失败重启。目前已有的分布式集群计算框架中对GPU任务的管理不够完善，几乎没有GPU任务失败重启的功能，也就是说当数据量特别大时，总会有GPU任务运行失败，就会导致整个程序的失败，这是个很严重的瓶颈问题。

4 算法设计与实现

4.1 Hadoop上的Kirchhoff算法实现

Hadoop算法的设计主要是Map和Reduce的过程，算法的主要流程如下所示：

1) 分析系统环境：根据Hadoop的官方介绍，一个节点上的mapper数在10~100之间的效果是最好的，我们可以获得集群的环境配置，如：集群节点个数n，每个节点上的内存总量$M_1, M_2 ... M_n$，CPU的总数*cpus*，每个CPU的核数为*cores*，每个核的线程数为*threads*。

2) 自定义输入阶段：在Hadoop中，每个输入文件块对应一个mapper。Hadoop提供了一个默认的分块机制，64M大小对应一块，不足64M的文件也默认是一块。我们通过中心点文件对输入道进行划分，由于中心点文件大小比较小，所以我们要自定义划分mapper的数量。Hadoop提供了重写*FileInputFormat*类来进行逻辑划分输入文件为split，每个split都对应一个mapper。因此我们只要设置了每个split的大小就可以确定mapper 的数量。首先在*FileInputFormat*中获取所有split的总大小：$S_1, S_2, S_3 ... S_n - \sum_{i=1}^{n} S_i$。在Hadoop 中，Reduce任务的个数是由用户设置的r_n，在mapper任务没有结束时，reduce任务就会被启动，因

此，reduce占用的资源会影响mapper的数量，基于这些考虑设置split 大小：

$$f_{split} = \frac{(\sum_{i=1}^{n} S_i)}{k * min\left((cpus * cores + r_n), \left(\frac{\sum_{i=1}^{n} M_i}{M}\right)\right)}$$

$k \in [1, \infty)$是一个可控参数，保证mapper的数量是集群资源容纳mapper数量的倍数，这样每次mappers都可以并行执行，不会剩余很少的mappers最后运行。最后我们通过设置*mapreduce.input.fileinputformat.split.maxsize*为f_{split}来划分输入文件。另一个限制是，保证split的大小为一道数据大小的倍数，防止切分一道数据。通过输入split，我们读取输入道数据，形成$< key, value >$键值对，并发送到每个mapper 中进行处理。*key* 代表输入道在文件中的道号，*value*代表每个输入道对应的中心点坐标。

3) Map阶段：对split传过来的输入道数据进行处理，读取HDFS上的速度数据，炮点坐标数据和检波器坐标数据，每个mapper都是并行的进行处理，这加快了运算的速度。由于每个输入道都会产生一定量的输出道，每个不同的输入道产生的输出道有很多重合，因此，我们再mapper中用一个HashMap保存输出道的数据，相同的输出道被叠加到同一个value上，节省了数据传输时间。另一点是，每个输入道产生输出道会进行多次循环，大大的浪费了时间，因此在每个mapper 中我们用多线程来处理输入道数据，我们设置每个mapper数据的线程数为$(threads - 2)$，如果系统支持超线程，则设置每个mapper中的线程数为$2 * (threads - 2)$。最后一个策略是将输出道的数据写入本地文件，减少数据的发送量，减轻网络带宽。将一个mapper中的输出道写入二进制文件，保存输出道的道号和文件名、偏移量，格式如：$< key, filename\#offset >$，只发送这些键值对到Reduce 任务。

4) Combine阶段：在Combine阶段，对同一节点上的mappers产生的数据进行聚合，同一个key的value 值进行聚合，这将会大大减少一个节点上的网络传输量，从而提高程序性能。

5) Partition阶段：由于mapper产生的输出道的key 值为输出道道号，Hadoop会自动按文本来排序，而不是按照数字大小排序，所以，在mapper发送到reduce端时，进行一个映射，把道号按照reduce 任务的大小进行划分，reduce id小的对应道号小的输出道。映射公式为：

$$f_{hash} = [\frac{key}{\left(\frac{key_{max}}{reduce_{id}}\right)}]$$

。这个策略会减少最后整个输出道的排序时间，只需要在每个reduce中进行排序，这同时也进行了并行的排序。

6) Reduce阶段：对mapper中传来的输出道$< key, filename\#offset >$键值对，我们根据键值对读取mapper生成的文件，进行聚合相同的key值，对key 值相同的value 进行叠加。

7) Output阶段：在Hadoop中，每个reduce 会对应一个输出文件，Hadoop中默认的是把reduce的输出键值对按文本的形式输出到文件，这里我们对reduce 重写输出文件类：*FileOutputFormat*，在每个输出类中，我们获取键值对并排序，按照排好的序列将value值写入二进制文件，并把文件名用最小的key值命名。

8) 成像步骤：根据reduce阶段生成输出文件，我们根据文件名中的最小key 值进行排序，然后依次把所以的二进制文件聚合到一个image二进制文件中。最后删除所有的中间文件以节省空间。

Hadoop上Kirchhoff算法结构图如图11所示。

4.2 Spark框架上的Kirchhoff算法实现

Spark提供了弹性分布式数据集（RDD）屏蔽了与HDFS的交互。Spark还开发了ApplicationMaster来适用于Yarn，如图3所

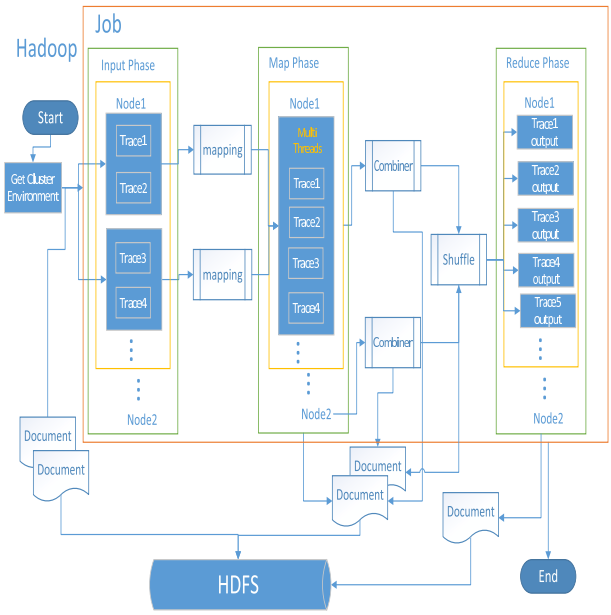

图 11: Hadoop上的Kirchhoff算法流程

示。Spark在Yarn上的部署是最有前景的部署方法。因此我们将在Yarn上运行Spark程序。Spark上的Kirchhoff算法设计如下：

1) 获取集群环境阶段：Spark提供了*newAPIHadoopFile*方法来获取HFDS上的输入文件，该方法读取HDFS文件产生的$< key, value >$键值对生成一个RDD数据集，*key*代表的是输入道道号，*value*代表的是输入道对应的中心点坐标。一个RDD记录就代表一系列输入道，该RDD的分区会默认是HDFS上输入文件的分块数。RDD的一个分区就对应了一个executor（相当于mapper），多个executor分布在不同节点上并行执行。此外，Spark提供了一个*spark-submit*脚本来提交用户程序。该脚本可以通过命令行设置executor的个数N，每个executor使用的内存量以及每个executor使用的CPU核数，因此我们只需要读取这些参数就可以知道集群的环境。这种设置非常方便，从这些参数中我们就可以知道RDD的分区可以设置为：

$$f_{pars} = k * min\left(N, \frac{\sum_{i=1}^{n} M_i}{M_{min}}\right)$$

$k \in [1, \infty)$是一个可变参数，为一个整数，表示分区数为executor数的倍数，这样就保证了每次都并行处理分区，不会遗留剩余的分区被executor单独处理。N代表这个集群上的executor个数，一般1 ∼ 2CPU核一个executor，如果内存特别小，则会影响executor数量，因此选择CPU核心数和内存划分数最小的一个作为executor的数量。

2) 输入阶段：Spark提供的*newAPIHadoopFile*方法可以读取HDFS文件，并且能通过自定义的输入格式来读取文件，我们定义一个输入文件类，把输入道文件读取成$< key, value >$键值对，并且保证每道数据都是一个整体，不能切分一道数据，方便计算。*key*代表输入道道号，*value*代表输入道道号对应的中心点坐标。通过读取HDFS，Spark返回一个RDD数据集，RDD的每条记录包含一系列输入道，因为RDD支持手动分区，根据上面的公式，我们把RDD分区以在executor上并行执行。另外，RDD还支持*persist*操作，可以把中间数据缓存在内存中，*persist*提供了多级缓存机制，可以把数据完全缓存在内存中，也可以缓存在硬盘上，或者两种都使用。不仅如此，*persist*操作还支持缓存双份在内存或硬盘中，加速了并行读写速度。*persist*在第一次计算后就缓存了数据，等下次计算时直接读取缓存，加快了执行速度。因此，我们把从HDFS中读取的输入道数据缓存在内存和硬盘中（防止过大，内存不足），当再次读取数据时加快速度，并且能在后续RDD失败时，能快速重建该RDD。

3) FlatMap阶段：在Spark中，RDD的每个分区都会进行*FlatMap*操作，一次一个executor计算一个分区，当分区计算完成后继续计算其他分区，多个executor并行执行。在*FlatMap*阶段，获取$< key, value >$键值对中的输入道道号和中心点坐标，对每个输入道读取HDFS上的速度文件，炮点坐标文件和检波器坐标文件来计算出输出道的数据。每个输入道对应一个孔径，每个孔径中有一系列输出道，相邻的输入道产生的输出道有很多交集，因

此我们在*FlatMap*中使用一个*HashMap* 来保存输出道，相同的输出道叠加到一起。这有利于减少节点之间和节点内的通信量。因为每个输入道会产生大量的输出道，一个*FlatMap*包含了大量的输入道，因此我们用个多线程来并行加速一个RDD分区，如果机器系统开启了*Hyper-Threading*，则设置线程数为$2*(threads-2)$，如果*Hyper-Threading*关闭，则设置线程数为$(threads-2)$。在Hadoop中，当mapper任务运行到一个合适的百分比时，系统会启动reduce任务来收集mapper任务产生的中间结果，reduce 任务会一直占用资源，这会影响后期mapper的数量。使得程序性能下降。而在Spark 中，当mapper任务结束后会返回一个RDD，然后这个RDD被应用到reduce操作，因此，系统不会在mapper没执行完时进行reduce操作，这使得mapper 可以完全并行执行，提高了程序的性能。另外，在Spark中，一旦RDD分区后，分区信息会一直保留着，也就是说，在执行reduce操作时，RDD 的分区数依然是mapper规定的分区数。如果mapper产生的分区数不适合reduce任务，则可以使用*repartition*函数对RDD进行重新分区来适应reduce任务。在这阶段，对输出道进行聚合生成$< key, value >$键值对，*key*代表输出道道号，*value*代表输出道对应的成像数据。返回这些键值对生成的RDD。

4) Partition阶段：首先，我们获取输出道的总大小*onx*，然后根据输出道的*key*值进行划分，保证小的*key*值对应同一个reduce 任务，这样就避免了最后成像数据的排序时间，只需要在每个reduce任务中并行排序就行。我们根据reduce任务的个数R_n来确定每个*key*的映射情况，公式如下：

$$f_{partition} = [\frac{key}{(\frac{onx}{R_n})}]$$

5) ReduceByKey阶段：该阶段接收来自映射的FlatMap的$< key, value >$键值对，对每个相同的*key*值的*value*进行叠加，生成图像数据。*reduceByKey*方法的另一个优化就是：它会先把同一个节点的相同输出道道号*key*进行聚合，然后再传递到reduce任务中，这减少了节点间的通信量。*reduceByKey*方法将返回一个RDD，保存了输出道道号和成像数据组成的键值对。每个reduce任务会并行的运行在不同的executor上，一个reduce任务完成后就会执行下一个reduce任务，只要保证reduce分区是reduce任务的倍数，这样才能保证所有的reduce任务并行执行，不会浪费资源。

6) SortByKey阶段：在Reduce阶段生成的RDD进行排序操作，因为生成的成像文件是按照输出道道号的大小进行排序的，运用Spark的*sortByKey*函数可以对每个RDD分区中的输出道进行聚合。所有的分区都是并行排序，加快了排序时间。最后返回排序好的RDD。

7) 成像阶段：根据排序好的RDD进行写文件到HDFS，Spark提供了*saveAsNewAPIHadoopFile* 函数对RDD分区分别写入文件，产生的文件名由该分区中最小的输出道道号的值来命名，最后根据文件名来合并所有的成像文件，生成一个二进制成像文件，并删除中间文件。

这个程序的流程图如图12所示。

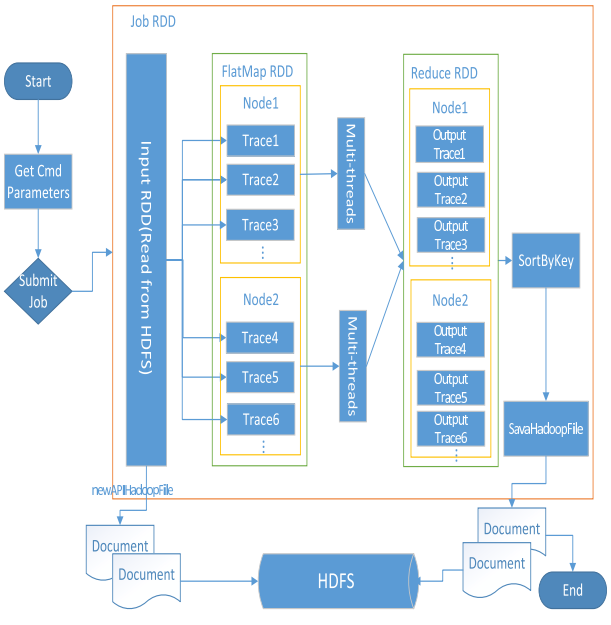

图 12: Spark框架上的Kirchhoff实现

5 实验评价

5.1 实验环境

集群有4个节点，分别是*Master,Slave1,Slave3,Slave4*。集群每个节点的资源如表1所示。

Name	CPUs	Cores Per CPU	Thread Per Core	Memory(G)
Master	2	8	4	32
Slave1	2	6	4	32
Slave3	2	6	4	32
Slave4	2	6	4	32

TABLE 1: Cluster Configuration

在集群中每个节点只配置20G内存给Yarn做资源，保证节点上其他程序和系统运行流畅。

5.2 实验配置

“Master”节点作为Hadoop的Master节点，用于调度集群中的任务，不进行计算（但当我们需要测试四台机器性能时，也会把该节点扩展成计算节点），该节点主要用于运行HDFS的管理进程NameNode以及Yarn的管理进程ResourceManager；其他“Slave1”、“Slave3”、“Slave4”三个节点作为作为Hadoop的Slave节点，这些节点主要用于运行HDFS的存储进程DataNode和Yarn的节点任务管理进程NodeManager。

5.3 数据准备

PKTM主要包括三个方面：数据预处理，数据迁移，输出成像。PKTM使用两种输入数据文件格式，包括“meta”文件和“data”文件。在本程序中输入文件主要包括：输入道meta文件（shot.meta），输入道震源坐标文件（fsxy.meta），输入道检波器坐标文件（fgxy.meta），输入道中心点坐标文件（fcxy.meta），速度信息文件（rmsv.meta）。每个meta信息文件都对应有一个data文件，用于保存数据文件：shot.data，fsxy.data，fgxy.data，fcxy.data，rmsv.data。

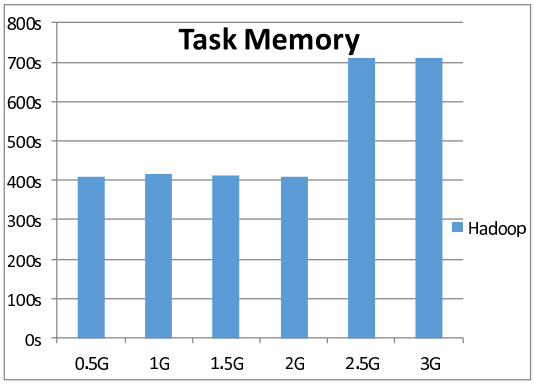

图 13: Mapper内存变化

5.4 实验结果

我们主要从以下几个方面来测试我们的程序：Hadoop上的单独测试、Spark的单独测试、Hadoop和Spark对比测试。为了节省时间，我们使用了小数据集进行测试，对于同一个测试使用同一个数据集，但是对于不同的测试可能使用的不是同样大小的数据集。

1. 在Hadoop上的测试，

   a. 我们首先测试mapper任务的container内存量变化给程序性能带来影响。测试结果如图13所示。我们可以看出当内存扩展到一个阈值后，mapper的数量就由于内存的增大而变少，因此执行时间就会变长。

   b. 其次，我们测试mapper的数量如何影响程序的性能。程序测试结果如图14所示。在这幅图中，当我们的mapper数量适应了集群的资源后，就获得了最佳的执行时间，而其他的数量就会获得相对较差的性能。

   c. 最后，在Hadoop中，当mapper完成一定的百分比时，集群就会启动reduce任务来接收mapper的输出文件，但是当reduce任务过早启动的话，会一直占用集群的内存和CPU资源，使得后来执行的mapper任务数量大大的减少，如果reduce 任务数量较多，最后mapper会

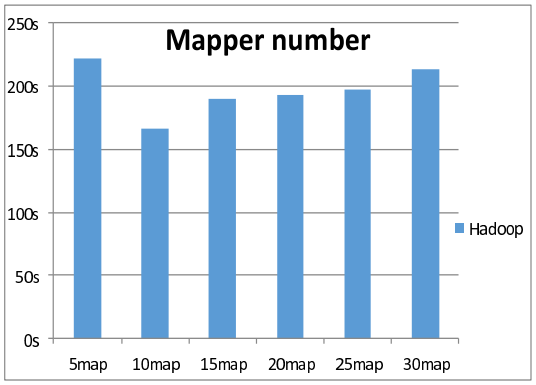

图 14: Mappers数量变化

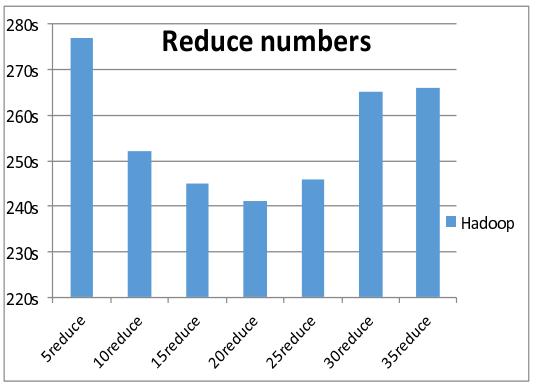

图 15: Reduces数量变化

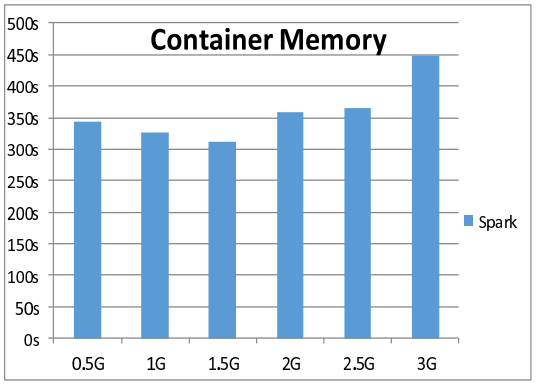

图 16: Container内存变化

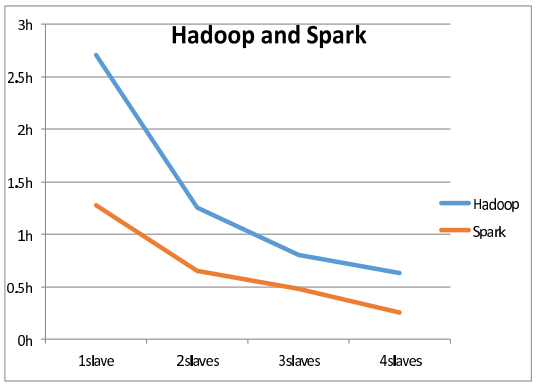

图 18: Hadoop和Spark性能对比

剩下很多单独执行的任务，失去了并行执行的优势。因此，我们测试reduce任务启动的时间对程序性能的影响。测试结果如图15所示。在图中，我们看到了当启动任务的时间达到一个合适的值时，程序会达到一个好的性能。

2. 在Spark上的实验：在Spark中，我们只测试两个方面，每个executor的内存量和RDD的分区数。

a) 我们首先改变container内存的配置来测试程序的性能，测试结果如图16所示。像Hadoop一样，如果container的内存量超过一个阈值后，程序的执行时间会变得更长。

b) 在本部分，我们测试输入道数据的RDD分区，测试结果如图17所示。图中显示了当RDD分区不足集群所能容纳的最大资源数时，程序的性能是上升的，但当RDD的分区继续增加时，程序的性能会下降，但当分区超过集群资源容纳的executor数量时，性能的变化不大，因为所有分区都是并行执行。

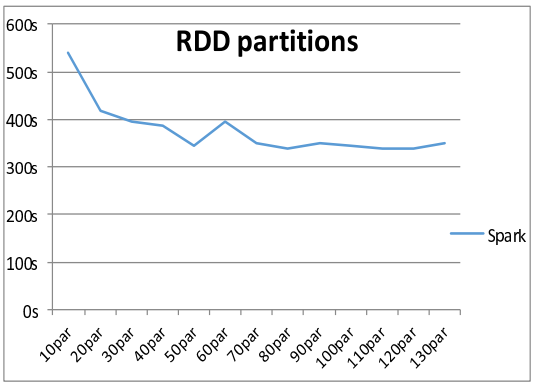

图 17: RDD partitions

3. Hadoop和Spark上的程序测试对比：在本次测试中，我们用了相同的数据集，以及大致相同的配置，如container的内存量，mapper中的线程数等。测试结如图18所示。

6 总结

在本文中，Kirchhoff算法的设计目的主要是避免HDFS读写和输入道计算分开，因为CPU的计算执行时间要比HDFS读写速度快很多。当CPU计算时如果需要数据来计算，则会等待HDFS读写完毕后继续执行计算，这会导致CPU的一个读写等待时间，浪费了CPU的资源。因此我们会尽量让数据事先读到内存中，这样当CPU需要数据进行计算时，会从内存中读取数据，速度远远小于直接读取HDFS文件。尽管我们做了这样的改善，但当内存量不够的情况下，程序依旧需要从HDFS读取文件，然后缓存到内存中，等CPU计算时获取。在Spark中，提供了RDD内存计算，我们可以直接把数据读取到RDD中，然后缓存到内存或硬盘上，当CPU计算需要数据时，直接从缓存的内存或硬盘中读取，加快了读写的速度，这个程序依然适用于大数据，只需要修改些配置就行了。

本程序的性能和集群的性能有关，如果集群有网络高通量和高速的硬盘读写能力，程序会有更好的性能。这意味着，网络通量和硬盘的读写能力是程序的一个性能瓶颈。另一个程序的瓶颈就是程序中依然存在着CPU计算和HDFS读写的交叉执行。还有一个瓶颈就是内存的容量，当内存的容量特别大时，程序每次可以缓存更多的数据到内存中，这样就减少了读取HDFS的次数，提高了程序的性能。

本段中我们介绍一些Kirchhoff改进的想法，在Hadoop中，我们通过Infiniband在HDFS上应用了RDMA（Remote Direct Memory Access）技术，这将在很大程度上提高程序的性能。这是解决网络传输慢的瓶颈的一个方法。同样的，在Spark中，我们也可以应用Infiniband 高速网卡来加速网络传输。Spark还可以使用Tachyon来加速访问HDFS文件，Tachyon完全兼容HDFS，搭建在HDFS 上层。Tachyon是一个高容错的分布式文件系统，允许文件以内存的速度在集群框架中进行可靠的共享，就像Spark和MapReduce那样。通过利用信息继承，内存侵入，Tachyon获得了高性能。Tachyon工作集文件缓存在内存中，并且让不同的Jobs/Queries以及框架都能以内存的速度来访问缓存文件。

REFERENCES

[1] Rizvandi N B, Boloori A J, Kamyabpour N, et al. MapReduce implementation of prestack Kirchhoff time migration (PKTM) on seismic data[C]//Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2011 12th International Conference on. IEEE, 2011: 86-91.

[2] Emami M, Setayesh A, Jaberi N. Distributed computing of Seismic Imaging Algorithms[J]. arXiv preprint arXiv:1204.1225, 2012.

[3] Shi X, Wang X, Zhao C, et al. Practical Pre-stack Kirchhoff Time Migration of Seismic Processing on General Purpose GPU[C]//CSIE (2). 2009: 461-465.

[4] Panetta J, Teixeira T, de Souza Filho P R P, et al. Accelerating Kirchhoff migration by CPU and GPU cooperation[C]//Computer Architecture and High Performance Computing, 2009. SBAC-PAD'09. 21st International Symposium on. IEEE, 2009: 26-32.