Journal of Computer Applications
计算机应用,2017,37(2):305 - 310
ISSN 1001-9081
CODEN JYIIDU
2017-02-10
http://www.joca.cn

文章编号:1001-9081(2017)02-0305-06
DOI:10.11772/j.issn.1001-9081.2017.02.0305

轻量级大数据运算系统 Helius

丁梦苏,陈世敏*
(计算机体系结构国家重点实验室(中国科学院计算技术研究所),北京 100190)
(*通信作者电子邮箱 chensm@ict.ac.cn)

摘　要:针对 Spark 数据集不可变,以及 Java 虚拟机(JVM)依赖环境引起的代码执行、内存管理、数据序列化/反序列化等开销过多的不足,采用 C/C++语言,设计并实现了一种轻量级的大数据运算系统——Helius。Helius 支持 Spark 的基本操作,同时允许数据集整体修改;同时,Helius 利用 C/C++优化内存管理和网络传输,并采用 stateless worker 机制简化分布式计算平台的容错恢复过程。实验结果显示:5 次迭代中,Helius 运行 PageRank 算法的时间仅为 Spark 的 25.12% ~53.14%,运行 TPCH Q6 的时间仅为 Spark 的 57.37%;在 PageRank 迭代 1 次的基础上,运行在 Helius 系统下时,master 节点 IP 接收和发送数据量约为运行于 Spark 系统的 40%和 15%,而且 200 s 的运行过程中,Helius 占用的总内存约为 Spark 的 25%。实验结果与分析表明,与 Spark 相比,Helius 具有节约内存、不需要序列化和反序列化、减少网络交互以及容错简单等优点。

关键词:内存计算;大数据运算;分布式计算;有向无环图调度;容错恢复

中图分类号:TP311.133.1　　**文献标志码**:A

Helius: a lightweight big data processing system

DING Mengsu, CHEN Shimin*
(*Key Laboratory of Computer System and Architecture* (*Institute of Computing Technology, Chinese Academy of Sciences*), *Beijing* 100190, *China*)

www.joca.cn

Abstract: Concerning the limitations of Spark, including immutable datasets and significant costs of code execution, memory management and data serialization/deserialization caused by running environment of Java Virtual Machine (JVM), a light-weight big data processing system, named Helius, was implemented in C/C++. Helius supports the basic operations of Spark, while allowing the data set to be modified as a whole. In Helius, the C/C++ is utilized to optimize the memory management and network communication, and a stateless worker mechanism is utilized to simplify the fault tolerance and recovery process of the distributed computing platform. The experimental results showed that in 5 iterations, the running time in Helius was only 25.12% to 53.14% of that in Spark when running PageRank iterative jobs, and the running time in Helius was only 57.37% of that in Spark when processing TPCH Q6. On the basis of one iteration of PageRank, the IP incoming and outcoming data sizes of master node in Helius were about 40% and 15% of those in Sparks, and the total memory consumed in the worker node in Helius was only 25% of that in Spark. Compared with Spark, Helius has the advantages of saving memory, eliminating the need for serialization and deserialization, reducing network interaction and simplifying fault tolerance.

Key words: in-memory computation; big data processing; distributed computation; Directed Acyclic Graph (DAG) scheduling; fault tolerance and recovery

0　引言

在科学研究和产业实践中,MapReduce[1] 集群编程模型已经广泛应用于大规模数据处理。MapReduce 系统把用户编制的串行 Map 和 Reduce 程序自动地分布并行执行,在每次运算前,系统需要从分布式文件系统中读取输入数据,运算完成后,系统要将计算结果写入分布式文件系统中。如此一来,多个 MapReduce 运算之间只能通过分布式文件系统才能共享数据,这不仅产生了大量的中间文件,而且反复读写磁盘大幅降低了运算性能。随着内存容量指数级增长和单位内存价格不断下降,大容量内存正成为服务器的标准配置,于是内存计算逐渐被主流商用系统和开源工具所接受。以内存计算为核心思想的 Spark[2-4] 在性能上远超基于 MapReduce 的 Hadoop[5]:迭代计算性能和数据分析性能分别可以提高 20 倍和 40 倍。Spark 在保持 MapReduce 自动容错、位置感知调度、可扩展性等优点的同时,高效地支持多个运算通过内存重用中间结果,从而避免了外存访问的开销。Spark 的基本数据模型是弹性分布式数据集(Resilient Distributed Dataset, RDD)[3]。一个 RDD 是一个只读的数据集合,生成之后不能修改。RDD 支持粗粒度的运算,即集合中的每个数据元素都进行统一的运算。RDD 可以划分为分区分布在多个机器节点上,所以一个运算可以在多个节点上分布式地执行。Spark 通过记录计算间的沿袭(Lineage)以支持容错,当出现故障导致 RDD 分区丢失时,Spark 根据记录的计算沿袭,重新计算并

收稿日期:2016-08-12;**修回日期**:2016-10-22。
基金项目:中国科学院"百人计划"项目;国家自然科学基金面上项目(61572468);国家自然科学基金创新群体项目(61521092)。
作者简介:丁梦苏(1993—),女,江西吉安人,硕士研究生,主要研究方向:大数据处理、并行分布式计算;　陈世敏(1973—),男,北京人,研究员,博士,主要研究方向:数据管理系统、大数据处理、计算机体系结构。

重建丢失的分区。

然而,Spark 的设计和实现存在着一定的局限性。首先,RDD 被设计成只读的数据集,既不支持重写,也不支持数据追加。于是,Spark 需要为每个新创建的 RDD 分配内存空间,尤其在迭代计算时,每个循环都产生一组新的 RDD,这加大了内存开销。其次,Spark 采用 Scala 程序设计语言实现,在 Java 虚拟机(Java Virtual Machine, JVM)[6]上运行,继承了 Java 的一系列问题。程序编译后生成字节码,执行时再由 JVM 解释执行或进行即时(Just-In-Time, JIT)编译成为机器码。内存管理无法主动释放内存,必须由 JVM 的垃圾回收机制才能释放内存。数据传输时,需要经历数据的序列化和反序列化,不仅增加了转换的计算代价,而且序列化的数据通常增加了类型等信息,引起网络传输数据量的增加。这些问题在一定程度上限制了系统的性能。

为此,用 C/C++ 语言设计并实现了一种轻量级的大数据运算系统——Helius。Helius 采用了一种类似于 RDD 的数据模型,称为 BPD(Bulk Parallel Dataset)。BPD 与 RDD 的区别在于 BPD 可写,RDD 只可读。用户可以选择重写 BPD,系统无须重新分配内存,而是直接覆盖原来的区域。这样不仅节省了内存开销,而且提高了运算性能。BPD 在多个计算间提供了一种高效的共享方式,计算结果存入内存,其他计算通过直接访问内存快速地获取输入。

与 Spark 相同,Helius 也采用 master-worker 分布式架构,通过记录各个 BPD 操作之间的计算沿袭构建依赖关系,动态生成计算的有向无环图(Directed Acyclic Graph, DAG)[7],划分计算阶段,在每个阶段中多个计算任务并行执行。Helius 支持 Spark 的各项计算、自动容错、感知调度和可扩展性。相对 Spark 而言,Helius 的优势具体如下:

1)降低内存开销。Helius 采用 C/C++ 实现,程序运行时能够实时回收内存;此外,BPD 的可变性支持系统在计算过程中充分利用已有的内存空间,减少了不必要的内存开销。

2)不需要序列化和反序列化。数据在系统中以二进制字节的方式存储,当集群节点都是 x86 机器时,网络传输时可以直接发送二进制数据,不需要进行 Endian 转换和序列化/反序列化。

3)减少网络交互。Helius 使用一种类似于 push 的方式传递数据,master 直接操控数据的传输,worker 之间不需要互相发送请求,从而减少了网络请求的交互。

4)简化容错恢复。Helius 应用了一种 stateless worker 的思想,worker 遵循网络请求进行工作,请求包含计算所需的状态信息,而 worker 除 BPD 数据分区外,不保存计算状态。这样,系统将多点故障集中到了对单点 master 的故障处理。

1 基于 BPD 的编程模型

1.1 BPD 数据模型

类似于 Spark 系统中的 RDD,BPD 是一种分布式的数据集合,可以划分为多个数据分区,存放在多个 worker 节点上。BPD 支持粗粒度的运算,集合中的每个数据元素都进行统一的操作。这样,不同 worker 节点上的 BPD 分区可以并行执行相同的运算。

与 RDD 不同,BPD 是可变数据集。考虑一个简单的例子,对所有数据元素自增,因 RDD 只读性的限制,Spark 需要分配内存空间,为这个操作创建一个新的 RDD;而 Helius 避免了额外的空间开销,新的结果可以直接填充覆盖原始的数据集。

BPD 遵循一套严格且灵活的可变机制。严格性是系统层考虑的问题,体现在只有用户计算产生的 BPD 可变,并且要求计算过程中新产生的数据元素占用的内存空间维持不变。Helius 针对少部分遵循可变机制的用户计算函数(UDFListCombine 函数)实现更新接口,其他函数不提供 BPD 更新支持。灵活性针对用户层而言,用户调用支持 BPD 更新的函数(如 UDFListCombine)时,可以通过设置函数参数(真或假)指示该 BPD 是否在该运算中更新。若不更新,系统将新创建一个 BPD;若更新,新结果将覆盖待计算的 BPD,无需重新分配空间。对于一系列不改变数据结构的操作而言,系统只需覆盖相应数值,在处理大量的数据时节省了时空开销。

RDD 的只读性简化了数据一致性的实现。在 Helius 中则需要考虑如何保持 BPD 的一致性。与 Spark 相似,在 Helius 中,用户提供一个主驱动程序,BPD 体现为程序中的特殊变量,变量之间的运算对应于 BPD 分布式运算。Helius 的 master 节点加载主驱动程序,按照执行步骤,执行相应的分布式运算。从概念上看,虽然 BPD 运算是分布并行的,但是这个主驱动程序实际上是一个串行程序(当然可以包含循环、分支等控制流语句),它描述了 BPD 运算步骤之间的串行执行顺序和依赖关系。所以,单一的主驱动程序可以保证 BPD 数据的一致性。对于多个并发执行的主驱动程序,Helius 禁止发生修改的 BPD 在多个并发程序之间共享,只有当进行修改操作的程序执行完毕后,被修改的 BPD 才可以被其他程序所使用。

具体实现时,master 记录 BPD 的元数据,主要包含依赖关系、分区数据、分区划分信息和存储方式。依赖关系记录了父子 BPD 之间的转换关系,分区数据记录了子分区与一个或多个父分区之间的生成规则。一个 BPD 的多个分区大小可以不等,存储在 worker 节点上。worker 把内存划分为等长的数据块,一个 BPD 分区由一个或多个数据块组成,这些数据块分布在内存或文件中,具体的存储位置由 BPD 的存储方式决定。用户可调用系统接口选择数据的存储方式。BPD 可以按照用户指定的划分方式重新哈希散列成指定个数的分区。

1.2 BPD 记录数据类型

Helius 系统将 BPD 按二进制数据进行存储和处理。一个 BPD 数据集中的所有记录都具有相同的结构,可以是键值对 Key-Value 元组,也可以是无 key 或是无 value 的单个元素。对于 key 或是 value,它的结构可以是定长或变长的数据,可以表达 C/C++ 中的原子类型(数值类型、字符串类型等)、struct 和 class 数据(内部不允许指针、没有虚函数)。key 或 value 也可以进一步有内部嵌套结构,可以嵌套包含两个值或是多个值。嵌套主要发生在 Join 等操作的结果 BPD 上。系统提供方法获取 BPD 记录的 key 或 value 的二进制数据,二进制数据与正确的数值类型的转换依赖于用户的代码。通常在 C/C++ 程序中,只需要对相应类型的指针变量赋值即可,

www.joca.cn

不需要额外的转换和拷贝。

1.3　编程模型

用户将 C/C ++ 的主驱动程序编译成动态库后提交给 master,与此同时指定 master 的运行入口函数。master 解析库文件,依次执行函数体内的语句。在主驱动程序中,一个 BPD 表现为一个可操作的 C ++ 对象。各种计算通过调用该对象相应的方法而实现,计算可以生成新的 BPD 对象,或者修改已有的 BPD 对象。而这些 BPD 对象上的操作,就被 Helius 对应为对 BPD 多个分区上的分布式运算。

Helius 提供两大类计算,用以处理 BPD 数据集:系统计算和用户计算。

1)系统计算:完全由系统实现的计算,包括 union、cartesianProduct、partitionBy、join、groupBy 等,用户可以直接调用系统计算函数处理 BPD 数据集,这些计算都不改变输入的 BPD 数据集。

$$
\begin{aligned}
&\text{union}\ (A,B) && \rightarrow\ A \cup B \\
&\text{cartesianProduct}\ (\{<ak,\ av>\},\{<bk,\ bv>\}) && \\
& && \rightarrow\ \{<ak \mid bk,\ av \mid bv>\} \\
&\text{partitionBy}\ (n,\ \{<k,\ v>\}) && \rightarrow \\
&\text{join}\ (\{<k,\ av>\},\ \{<k,\ bv>\}) && \rightarrow\ \{<k,\ av \mid bv>\} \\
&\text{groupBy}\ (\{<k,v>\}) && \rightarrow\ \{<k,\ list(v)>\} \\
&\text{sortedGroupBy}\ (\{<k,v>\}) && \rightarrow\ \{<k,\ list(v)>\} \\
&\text{lookUp}\ (k,\ \{<k,\ v>\}) && \rightarrow\ list(v) \\
&\text{collect}\ (\{<k,\ v>\}) && \rightarrow\ \{<k,\ v>\}
\end{aligned}
$$

2)用户计算:系统提供应用程序编程接口(Application Programming Interface, API),由用户实现具体操作功能。用户在处理数据集之前需要根据 API 实现函数接口。在用户计算中,用户可以选择是否改变待计算的 BPD 数据集。

A = udfCompute (B, udf)

A = udfComputeMulti (B, C, $\cdots$, udf)

A = udfListCombine (B, udf)

系统计算函数的语义很清晰。例如:join 操作把两组输入 BPD 的 Key-Value 记录,按照 key 进行等值连接,输出记录的 value 部分是嵌套结构,由两个匹配记录的 value 部分组合形成;groupBy 操作按照 key 进行分组,把同一组的所有 value 表示成一个 list,即一个包含多值的嵌套结构。而用户计算接口主要包括三类,都要求用户提供一个根据相应接口实现的函数(在表中以 udf 表示)。首先,udfCompute 方法针对单个 BPD 数据集的每条 Key-Value 记录进行处理,例如可以实现 WordCount 中单词的拆分。udfComputeMulti 方法对多个 BPD 数据集的 Key-Value 记录进行某种运算。实际上,多个输入的 BPD 数据集进行了一次 join 操作,系统对每个 join 的结果调用一次用户实现的 udfComputeMulti 函数。这两类操作对数据集的结构没有要求,可为 1.2 节提及的任意一种存在形式。udfListCombine 与 udfCompute 的处理对象类似,不同的是数据集 key、value 必须同时存在,并且 value 为包含多值的嵌套结构。它实现对每个 key 的多个 value 值进行聚合的操作,类似 MapReduce 系统中的 Reduce 操作。

1.4　实例介绍

以 WordCount 为例,统计文本中所有单词出现的次数,用户的主驱动程序如下:

```
BPD * lines = sc.loadFile(file);
BPD * words = udfCompute(lines, new mySplit());
BPD * wordgroup = words -> groupBy();
BPD * wordcount = udfListCombine(wordgroup, new myCombine());
```

loadFile 函数用于从文本生成一个 BPD 对象——lines,它的每个记录是一行文本。udfCompute 函数调用用户自定义的 mySplit 函数对每行文本记录进行处理,在该示例中表现为将字符串拆分成多个单词,产生的 words 结果记录中 key 为单词,value 为数值 1。groupBy 函数将 Key-Value 数据集按 key 进行分组。在这里,每个不同的单词为一组。最后,udfListCombine 函数调用用户自定义的 myCombine 函数对同一 key 的所有 value 进行某种运算(在该示例中为求和)。

下面以 udfListCombine 函数为例,介绍 udf 函数的实现。用户自定义实现的函数如下:

```
class myCombine: public UDFListCombine {
  void call(ValueIterator * it, Value * out) {
    int sum = 0;
    while (it -> hasNext()) {
      int * val = (int * )(it -> next());
      sum + = * val;
    }
    out -> put(&sum, sizeof(int));
  }};
```

用户实现了一个 myCombine 类,它继承了 UDFListCombine 类,实现 UDFListCombine 中的虚函数call()。call()的第一个输入参数是一个定义在输入 BPD 记录 value 列表上的 Iterator 迭代器。上述实现在 while 循环中通过这个 Iterator 依次访问列表中的每个 value,把 value 的地址赋值给相应类型的指针,就可以直接操作。call()的第二个参数用于输出结果的 BPD 的 value 部分。在这里,把求和的结果写入 out。

从上面的示例可见,用户可以使用 C/C ++ 程序简洁地表达大数据的运算。

2　分布式运行

Helius 分布式运行的基础是表达 BPD 运算关系的 DAG。用户的主驱动程序提交给 master 执行时,系统通过 BPD 变量获取具体运算及依赖关系,形成运算 DAG。然后,Helius 把一个 DAG 划分成多个阶段,每个阶段内部的运算可以在一起执行,从而减少中间结果的生成。一个阶段的输出结果为另一个阶段的输入。其中,最后一个阶段的输入来自原始数据源(例如文件),第一个阶段的计算结果是程序最终的输出结果。按照这种层次依赖关系,系统自上而下检查各个阶段(首先检查第一个阶段),当前阶段运行时将自动检测其依赖的其他阶段,若其他阶段准备就绪,则提交该阶段的任务;否则,迭代检查依赖的所有阶段,直至所有依赖阶段准备就绪后提交。每个阶段包含了一系列任务,系统将这些任务分配到最佳节点位置,并确保所有数据就绪。

2.1　DAG 的生成及阶段的创建

在 Helius 系统中,DAG 的生成过程以及阶段的创建过程与 Spark 系统类似,都是根据用户的主驱动程序进行的。用户主驱动程序执行时,系统先记录 BPD 的运算和依赖关系,并不立即执行所对应的分布式运算,只有当遇到 lookup、

www.joca.cn

collect 和程序结束时，才执行之前记录的所有 BPD 运算。

与 Spark 不同的是，Helius 将数据的 shuffle 操作单独抽取出来，显示地表达在 DAG 中，而非表示在其他的操作里。这样，DAG 可以记录 shuffle 的状态信息，而不需要每个 worker 在实现 BPD 运算（例如 groupBy）时，记录 shuffle 的状态信息。

记录的 BPD 形成了一个运算有向无环图（DAG），如图 1 所示。图的每个顶点是一个 BPD 或者 BPD 的版本（若被修改），顶点之间的有向边代表 BPD 运算的生成关系。有向边从输入 BPD 指向结果 BPD。

图 1 中每个顶点代表一个 BPD，其中 BPD1 和 BPD2 的 union 操作生成 BPD3，BPD3 的 groupBy 操作产生 BPD4，BPD4 是最终的计算目标。系统在执行用户主驱动程序时，记录 BPD 的运算和依赖关系。在这个例子中，当程序结束时，才生成 DAG 开始分布式计算。需要注意，图 1 中 BPD3 和 BPD4 之间的边是虚线，实际上 DAG 中删除了这条边。这也正体现了 Helius 与 Spark 的不同点。因为 groupBy 操作隐含地需要 shuffle 数据，系统自动生成了 BPD5（图中深色填充表示），并修改了图，使 BPD3 的输出指向 BPD5，BPD5 的输出指向 BPD4。

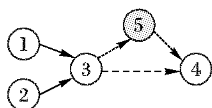

图 1 DAG 生成过程

Fig. 1 Generation process of DAG

阶段的创建由目标顶点和 shuffle 操作确定。在图 1 所示的 DAG 基础上，master 开始自下而上创建阶段，如图 2 所示。master 首先为目标顶点 BPD4 创建一个阶段（记为阶段 0），并从该位置开始迭代遍历其父 BPD。检测发现 BPD4 依赖于 shuffle 的结果，而 shuffle 必然需要网络传输，所以 master 以 shuffle 对应的 BPD5 为目的创建一个新的阶段（记为阶段 1）。依此类推，master 将 DAG 以 shuffle 为边界分为多个阶段，每个阶段内的 BPD 运算可以整合在一起执行，以提高运算的性能。

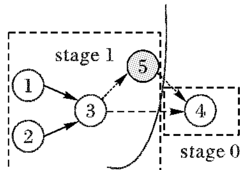

图 2 阶段创建过程

Fig. 2 Generation process of stage

所有阶段创建完毕后，master 从上向下依次递归提交阶段：在尝试提交阶段 0，master 检测到该阶段依赖于阶段 1，于是 master 挂起阶段 0 重新提交阶段 1；由于阶段 1 无依赖阶段，因此阶段 1 顺利被提交；master 开始提交阶段 1 对应的所有任务，阶段 1 完成后递归提交阶段 0；目标阶段完成后，结束调度。

2.2 任务提交

当一个阶段成功提交后，master 将为该阶段的目标 BPD 创建并提交任务。BPD 的每个分区作为一个任务，各个分区独立地执行相同的计算，这使得多个任务可以在多个 worker 节点上并行执行。

在分布式运算环境下，基于位置感知分配任务到存储数据的节点会大幅提高运算的性能，减小网络传输的带宽。Helius 提供位置感知调度。在 DAG 的基础上，master 进一步确定父子 BPD 每个分区之间的映射关系（shuffle 过程除外）。在分配任务时，递归计算该任务所在的分区依赖的父分区的位置，直到找到已经缓存的父分区后，将该任务发送到该父分区所在的 worker 节点，完成位置感知调度。

如果一个任务同时依赖两个父分区，并且两个父分区均已缓存时，那么默认将该任务分配到第一个依赖的父分区上。当两个父分区的数据在不同节点上，并且第二个父分区的数据量远大于第一个父分区时，将任务分发给第一个父分区所处的工作节点会增加网络开销，降低系统性能。一种优化方法是根据多个依赖的父分区的数据量确定最佳分配节点。

2.3 数据传输

当一个任务依赖多个数据源（多个父 BPD 分区），并且多个数据源不在同一工作节点时，worker 节点需要获得所有的输入数据，才能开始计算任务。

Spark 提供一种类似 pull 的获取方式，如图 3 所示。worker A 向 master 请求数据，master 定位数据所在的 worker B，由 worker B 将数据发送给 worker A。worker A 完成任务后回答 master。

www.joca.cn

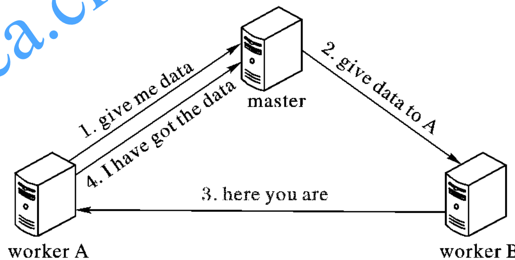

图 3 Spark 数据传输机制

Fig. 3 Data transmition mechanism in Spark

值得注意的是，在 Spark 系统中，对依赖数据的获取是计算任务的一部分，worker 在运行提交的任务时，可能需要远程获取数据。在发送消息 1 和消息 4 之间，worker A 需要保持相应的状态信息，使 worker 的工作和故障处理变得相对复杂。

Helius 提出一种 stateless worker 的机制，对数据的获取过程类似于 push。在该机制下，worker 不负责获取数据，而是由 master 指示其进行操作。worker 对于每个网络请求，只完成相应的操作，而在网络请求之间，不记录额外的状态。如果一个任务所需数据在本节点不存在时，向 master 报错。

将提交任务分成两个步骤：传输数据和提交作业。数据传输的过程如图 4 所示：master 告诉 worker B 传输数据给 worker A，worker B 传输指定的数据，worker A 接收完数据后回复 master 传输完成；master 接收到传输完毕信号后，紧接着提交作业。这样一来，系统保证了在分配工作之前，工作节点有需要的数据支持工作的进行，同时 worker 不需要保持额外的状态。这种 stateless worker 的机制简化了系统的容错处理，由于 worker 严格地按照 master 的指示工作，worker 的工作机制相对来说简单了许多，在该点的故障及故障处理随之简化。系统将故障处理主要集中在 master 执行。

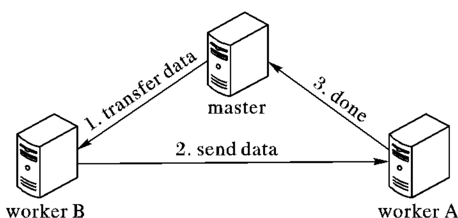

图 4　Helius 数据传输机制

Fig. 4　Data transmition mechanism in Helius

2.4　数据重组

不同于数据传输(transfer)操作,数据重组(shuffle)操作需要将数组重组分发到所有的工作节点,可能会占用大量的内存空间和网络带宽。

为了减少 shuffle 对系统性能的影响,采用一种基于双缓冲的边计算边发送的策略。worker 为每个 shuffle 目标 worker 节点都维持着一个缓冲区,包含 2 个数据块空间(分区数据由多个等长数据块组成)。在处理 shuffle 时,将数据写入相应 worker 的缓冲区的数据块中。当缓冲区中一个数据块已满,可以发送这个数据块,同时将数据写入另一个数据块。

图 5 呈现的是针对 worker A 单方面 shuffle 产生的数据发送的过程。图中的连线表示 worker 之间的连接状态,填充灰色部分代表该部分内存已满。

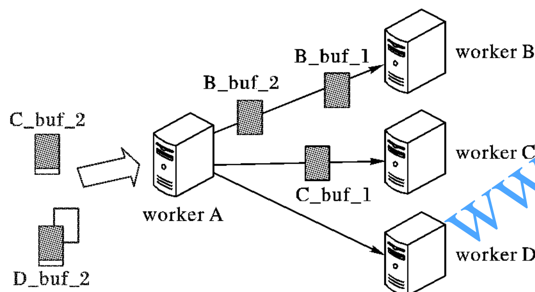

图 5　数据 shuffle 过程

Fig. 5　Data shuffle in Helius

3　实验评估与分析

3.1　实验环境

集群环境由 5 台服务器组成,其中 1 台 master,4 台 worker,服务器的处理器为 Intel Xeon CPU ES-2650 v2 @2.60 GHz×8,内存 128 GB,硬盘 1 TB,操作系统为 Ubuntu 14.04 64 位。集群中的工作节点均单线程运行。Helius 和 Spark 的实验版本分别为 0.0.1 和 1.6.1。Helius 编译器为 G++ 4.8.1,-o2 选项优化,Spark 编译器为 Sbt 0.13.12。

实验以 PageRank[8-9] 算法和 TPCH[10] 基准为例,从时间、网络、内存三方面开销比较 Helius 和 Spark 的性能,并在最后对 BPD 的更新性能以及 Helius 的可扩展性进行评估。

3.2　PageRank

实验输入文本为 1.1 GB,包含网页 4 847 570 个,链接记录 68 993 773 条。Spark 集群运行 PageRank 算法的配置选项为:spark. driver. memory = 16g, spark. executor. memory = 16g。

3.2.1　时间开销

运行 PageRank 算法时,分别记录迭代 1、2、3、4、5 次的时间开销。表 1 呈现的实现结果表明,在迭代 5 次的过程中,Helius 运行 PageRank 算法的时间仅为 Spark 的 25.12% ~ 53.14%。因为 Helius 在实现 PageRank 算法时,采用的是一种建立在数据块内有序、块间无序的基础上优化 join 操作的策略,在每次更新 rank 值时直接重写旧值,而非重新创建新的 BPD。

表 1　Helius 和 Spark 迭代时间对比

Tab. 1　Iteration time comparison of Helius and Spark

迭代次数	时间开销/s		Helius 与 Spark 运行时间之比/%
	Helius	Spark	
1	53.701	213.810	25.12
2	83.341	249.189	33.44
3	121.279	286.940	42.27
4	156.613	307.368	50.95
5	174.471	328.340	53.14

3.2.2　网络开销

在 PageRank 迭代 1 次的基础上,记录 master 节点在程序运行过程中接收到的字节数和发送的字节数,结果如表 2 所示。在分布式环境中,运行在 Helius 系统下时,master 节点 IP 接收和发送数据量约为运行于 Spark 系统的 40% 和 15%。

表 2　Helius 和 Spark 网络开销对比

Tab. 2　Network performance comparison of Helius and Spark

系统	接收字节数	发送字节数
Helius	54 723 054	132 259
Spark	136 659 386	873 799
Helius 与 Spark 的网络开销之比/%	40	15

3.2.3　内存开销

在 PageRank 迭代 1 次的基础上, 每隔 5 s 记录 worker 节点内存剩余情况。表 3 呈现的是以 20 s 为间隔记录的 worker 节点使用的内存量(单位:MB)。Helius 在 50 s 左右运行结束,逐渐回收内存;此时,Spark 仍处于工作状态,直到 210 s 左右结束。在 worker 运行的过程中,Helius 占用内存6 758 MB,Spark 占用内存26 648 MB,Helius 约为 Spark 的 25%。

表 3　Helius 和 Spark 内存开销对比

Tab. 3　Memory performance comparison of Helius and Spark

运行时间/s	占用内存/MB	
	Helius	Spark
0	5 413	4 794
20	8 129	18 232
40	12 171	25 017
60	5 417	25 900
80	—	26 316
100	—	26 648
120	—	29 315
140	—	29 533
160	—	31 264
180	—	31 442
200	—	5 124
总计	6 758	26 648

3.3　TPCH Q6 性能

以 TPCH 的 Forecasting Revenue Change Query(Q6)为例,取 Scale Factor 为 100(文本 79.8 GB),测试 Helius 和 Spark 的

www.joca.cn

运行时间。Spark 在该例中为默认配置。实验结果为 Helius 花费 271.595 s,Spark 花费 473.382 s,Helius 消耗时间仅为 Spark 的 57.37%。

Helius 从文本获取输入数据是一种筛选-丢弃的过程,根据用户提供的查询字段的列值,在读取文本记录时选取相应的字段值构成数据集,后续所有操作都建立在已筛选字段的数据集的基础上;而 Spark 程序在加载文件时没有对字段进行筛选,运行过程中,所有的数据集中的每条记录都保持了输入文本的所有字段。

3.4 BPD 更新性能

在 PageRank 迭代 1 次的基础上,测试在 BPD 更新与不更新的情况下,worker 运行 UDFListCombine 函数的开销时间,以及 master 运行用户提交的驱动程序所用的总时间。PageRank 在迭代 1 次的基础上会运行 1 次 UDFListCombine 函数。

从表 4 可以看出,在 BPD 更新的情况下,worker 运行 UDFListCombine 的速度比不更新稍快;master 运行整个程序也稍快。就表 4 的结果而言,BPD 更新在运行时间方面的性能提升不大,这种结果很大程度上受到 Helius 实现的限制,我们将在后续的工作中进一步研究 BPD 的更新。

表 4 有否 BPD 更新时运行时间对比

Tab. 4 Running time comparison with or without BPD update

工作节点	运行时间/s	
	BPD 更新	BPD 不更新
worker1	19.918	20.055
worker2	20.624	20.715
worker3	23.394	25.889
worker4	20.412	20.731
master(driven program)	53.839	54.895

3.5 可扩展性

以 3.3 节中的 TPCH Q6 为例,测试 Helius 集群分别搭建在 2、4、6、8 台 worker 的运行时间,结果如表 5 所示。在当前实验条件考虑的扩展情况下,当 worker 节点数增加 1 倍时,Helius 运行任务所需的时间减少 50% 左右。

表 5 Helius 可扩展性性能

Tab. 5 Helius scalable performance

worker 节点数	Helius 运行时间/s
2	552.982
4	286.620
6	186.794
8	141.545

4 结语

本文介绍了一种轻量级的基于内存计算的大数据运算系统 Helius。Helius 由 C/C++语言实现,避免了 Spark 因 JVM 运行环境引起的开销,利用数据集整体修改这一特性实现高效计算,采用一种 stateless worker 的机制简化容错处理,并通过维持一套严格的修改机制确保了数据一致性。Helius 在时间、网络、内存三方面性能相对 Spark 均有所提升。就数据集更新性能而言,Helius 存在很大的提升空间。此外,目前 Helius 还未实现节点故障恢复,故障处理以及深层次的一致性管理问题有待后续深入研究。

参考文献 (References)

[1] DEAN J, GHEMAWAT S. MapReduce: simplified data processing on large cluster [J]. Communication of the ACM — 50th Anniversary Issue: 1958 – 2008, 2008, 51(1): 107 – 113.

[2] ZAHARIA M. An architecture for fast and general data processing on large clusters, UCB/EECS-2014-12 [R]. Berkeley: University of California at Berkeley, 2014.

[3] ZAHARIA M, CHOWDHURY M, DAS T, et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing [C]// NSDI '12: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation. Berkeley, CA: USENIX Association, 2012: 15 – 28.

[4] The Apache Software Foundation. Apache Spark [EB/OL]. [2016-05-30]. http://spark.apache.org/.

[5] The Apache Software Foundation . Apache Hadoop [EB/OL]. [2016-05-30]. http://hadoop.apache.org/.

[6] SARIMBEKOV A, STADLER L, BULEJ L, et al. Workload characterization of JVM languages [J]. Software: Practice and Experience, 2016, 46(8): 1053 – 1089.

[7] ISARD M, BUDIU M, YU Y, et al. Dryad: distributed data-parallel programs for sequential building blocks [C]// EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007. New York: ACM, 2007: 59 – 72.

[8] BERKHIUT J. Google's PageRank algorithm for ranking nodes in general networks [C]// Proceedings of the 2016 13th International Workshop on Discrete Event Systems. Piscataway, NJ: IEEE, 2016: 163 – 172.

[9] PAGE L, BRIN S, MOTWANI R, et al. The PageRank citation ranking: bringing order to the Web, Technical Report 1999- 66 [R/OL]. California: Stanford University, 1999 [2016- 04- 11]. http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf.

[10] Transaction Processing Performance Council. TPC Benchmark™ H Standard Specification Revision 2.17.1 [S/OL]. [2016-05-30]. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.1.pdf.

[11] MALEWIEZ G, AUSTEM M H, BIK A J C, et al. Pregel: a system for large-scale graph processing [C]// SIGMOD '10: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. NewYork: ACM, 2010: 135 – 146.

[12] CARSTOIU D, LEPADATU E, GASPAR M. Hbase – non-SQL database, performances evaluation [J]. International Journal of Advancements in Computing Technology, 2010: 2(5): 42 – 52.

www.joca.cn

This work is partially supported by the CAS Hundred Talents Program, the General Project of the National Natural Science Foundation of China (61572468), the Innovative Community Project of the National Natural Science Foundation of China (61521092).

DING Mengsu, born in 1993, M. S. candidate. Her research interests include big data processing, parallel distributed computing.

CHEN Shimin, born in 1973, Ph. D., professor. His research interests include data management system, big data processing, computer architecture.