共享集群基于HDFS的数据块密度调度策略

杜红光 雷 州 陈圣波
(上海大学计算机科学与技术 上海 200444)

摘 要 随着云计算技术和海量数据处理技术的发展,共享集群逐渐采用HDFS作为分布式文件系统并通过虚拟化的方式管理计算资源,为计算框架和应用提供运行资源,造成应用运行过程中计算资源和数据存储的分离。海量数据处理应用的数据本地性是影响其性能的关键因素之一。目前,共享集群管理框架调度器的研究主要集中在通过提升调度的并行度来提高系统的吞吐量和资源利用率,而其在调度的质量方面还存在一些缺陷,如应用的数据本地性问题。提出基于数据块密度的调度策略,来提高应用的数据本地性,根据数据块的密度为应用等比例分配计算资源,减少应用运行过程中的跨主机I/O,从而提升应用的性能。实验表明,基于数据块密度的调度策略能够有效减少数据密集型作业的运行时间,该策略能够使应用达到90%的数据本地性。在测试应用WordCount和TeraSort中,该策略使应用缩短了20%左右的运行时间。

关键词 HDFS,数据块密度,共享集群,调度策略
中图法分类号 TP392 **文献标识码** A

Data Block Density Scheduling Strategy Based on HDFS in Shared Cluster

DU Hong-guang LEI Zhou CHEN Sheng-bo
(Department of Computer Science and Technology, Shanghai University, Shanghai 200444, China)

Abstract With the development of cloud computing technology and mass data processing technology, shared clusters use HDFS as a distributed file system and manage computing resources through virtualization to provide operational resources for computing frameworks and applications. The data localization of mass data processing applications is a key factor which affects its performance. At present, the research of shared cluster management framework's scheduler mainly focuses on improving the throughput and resource utilization of the system by improving the parallelism of dispatching, and there are some defects in the quality of scheduling, such as the data locality. In this paper, a scheduling strategy based on data block density was proposed to improve the data locality of the application. By using this strategy, the performance of the application can be improved by reducing the cross-host I/O during the application operation. Experiments show that the scheduling strategy proposed in this paper can effectively reduce the running time of data-intensive operations. In the test case of WordCount and TeraSort with 2.5G data, the method of this paper achieved 90% data localization and shortened the operation by 20% time.

Keywords HDFS, Data block density, Shared cluster, Scheduling strategy

1 引言

随着大数据技术的发展,MapReduce[1]成为并行处理海量数据的主流计算模型,并催生出众多相关计算框架,如Hadoop[2],Spark[3]等,这些框架方便开发者快速开发分布式应用。另外,分布式文件系统也得到了快速发展,如HDFS[4],GFS等,其中HDFS作为主流的开源分布式文件系统,被设计用来部署在价格低廉的硬件上,并且能够保证数据的容错性,解决了超大规模数据集的存储问题。

同时,伴随着云计算技术的发展,越来越多的应用被部署到共享集群上。由于算法的多样性,相关应用所依赖的计算框架可能不同,运行环境也可能会相互冲突,为每种计算框架或相应的算法构建一个单独的集群不但会浪费硬件资源,而且会增加运维成本。云计算为构建共享集群(即数据中心)提供了相应的解决方案,通过向共享集群提交作业的方式运行应用。共享集群有如下特点:1)硬件共享;2)细粒度的分配集群资源;3)应用之间可以数据共享;4)运维成本低;5)扩展性好。

在数据中心这样的共享集群中,往往通过虚拟化技术对CPU和内存进行分割,使用HDFS作为文件系统,该文件系统直接运行在物理机中,这就会出现应用所分配的计算资源与待处理数据不在同一台物理机的情况,因此产生了大量的跨节点I/O,这不仅影响着整个共享集群的网络负载,也影响着应用的运行性能。

传统的大数据处理系统直接使用物理服务器集群搭建,根据计算任务所运行的节点及待处理的数据存储位置的关系,数据本地性分为4种:"进程本地性",即计算任务及其要

杜红光(1990-),男,硕士,主要研究方向为新型IT架构、大数据,E-mail:duhongguang@foxmail.com;雷 州 副教授,主要研究方向为云计算、海量数据处理;陈圣波 讲师,主要研究方向为软件工程、软件测试。

处理的数据在同一个进程空间中；“节点数据本地性”，即计算任务及其要处理的数据在同一个节点中；“机架数据本地性”，即计算任务的运行位置和数据存储不在同一个节点，但是在同一个机架上；“跨机架数据本地性”，即计算任务所运行的节点及其所要处理的数据存储的节点不在同一个机架上[14]。同样地，在共享集群中也存在着数据本地性问题，但是共享集群以虚拟化的方式为作业分配资源，因此共享集群所关注的数据本地性主要集中在“节点数据本地性”方面。数据中心中的应用程序具有大量的数据读写操作，因此提高应用的数据本地性是提高应用程序性能的一个重要因素，也是目前的一个研究热点。

在对共享集群调度问题的研究工作中，目前主要集中在作业调度的并行度和速度，即吞储量和低延迟的问题[5]。例如，文献[6]根据调度器的并行调度方式将调度器架构分为“单体式调度器”“双层式调度器”和“共享状态调度器”。由于共享集群中存在虚拟化层（容器或虚拟机技术），仅考虑调度的并行度和速度并不能确保调度的质量，如作业的本地性问题。

基于数据本地性的任务能够最大化地利用 CPU 资源，同时减少磁盘和网络的 IO 消耗[7-9]。本文主要从作业的数据本地性角度对共享集群的调度策略进行研究，旨在确保调度器调度速度的同时提高共享集群的资源调度质量。本文研究共享集群中的数据本地性问题，使用 Kubernetes[10] 作为集群的管理系统，并且使用 HDFS 作为分布式文件系统，在应用提交阶段，通过分析应用使用到的数据的存储位置，以及数据在节点存储的数据块的比例，将该节点相应比例的计算资源分配给作业，即基于数据块密度为应用分配资源。通过使用 HiBench[11] 基准测试中的 5 个典型 Spark 作业对调度策略的调度结果进行测试，实验表明该方案在不同程度上缩短了 80％作业的运行时间。

本文的主要思想是对共享集群中待调度应用的处理数据的存储位置进行分析，根据应用提交时配置的输入文件，分析各个节点该文件数据块的数量，在该节点为作业分配等比例的 CPU 和内存资源。

本文第 2 节介绍共享集群资源管理和调度的相关技术，并分析目前资源管理框架在应用调度时存在的问题；第 3 节提出基于数据块密度的调度策略，并给出其设计与实现；第 4 节分析并讨论实验结果；最后总结全文并给出进一步的研究方向。

2　相关工作

目前，很多研究者致力于提高共享集群中调度速度和调度并行度的问题。然而，由于在共享集群中数据和计算完全分离，数据本地性问题成为了影响集群网络负载和应用作业性能的重要因素之一。本文旨在研究共享集群中系统应用性能的优化问题，与本文相关的研究工作可以归纳为以下 4 类：共享集群架构、Hadoop 分布式文件系统、Spark 作业运行过程以及共享集群资源管理。

2.1　共享集群架构

共享集群作为一个为应用作业提供运行环境的平台，提供数据存储、作业调度和资源管理等基本功能。目前，在作为数据中心的共享集群中，集群的基本架构如图 1 所示。

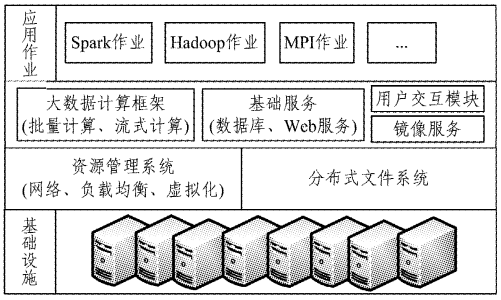

图 1　共享集群架构图

从图 1 可以看出，在一个共享集群中，在硬件上的是相关的软件技术栈，其包括集群服务器的操作系统、集群资源管理软件、分布式文件系统和分布式计算框架等，在这些基础的组件之上构建出应用的运行资源和环境。

2.2　Hadoop 分布式文件系统

Hadoop 包含两个主要组件，分别是分布式文件系统 HDFS（Hadoop 分布式文件系统）和分布式计算模型 Map/Reduce。HDFS 的设计理念来自 Google 分布式文件系统（GFS），与 GFS 设计理念类似，HDFS 被设计成适合运行在通用的硬件设备上的分布式文件系统，在具备一般文件系统的特征的同时，与其他分布式文件系统也有着明显的区别。

HDFS 采用主从式的服务器架构，由 NameNode 和 DataNode 组成，NameNode 是整个分布式文件系统的核心，维护着所有文件系统的元数据，并负责用户端对文件系统的访问，DataNode 是一般的存储节点。在 HDFS 中，文件被分割成若干个数据块，每个 DataNode 负责保存其中的若干个数据块。为保持数据的完整性和系统的健壮性，HDFS 采用了数据冗余备份的策略，即对每一个数据块进行备份，默认备份数量为 3。HDFS在设计之初旨在处理体积较大的文件，通常在 100M 或者以上，以数据分割存储的方式实现对大数据文件的分布式存储，HDFS 默认的 Block 大小为 64M，用户可以根据自己的需求进行调整[12]。本文所构建的共享集群采用大小为 128M 的 Block。

2.3　Spark 作业运行过程

Spark 应用程序作为集群上独立的进程集运行，由主程序中的 SparkContext 对象协调。通过 spark-submit 脚本向集群管理器提交作业，首先创建一个 Driver，在 Driver 中启动 SparkContext，SparkContext 连接到集群管理器并申请资源，集群管理器选择节点为 Spark 作业分配 executors 资源；然后在 Driver 中发送应用程序的代码到这些 executors；最后 SparkContext 将计算任务发送到 executors 运行。Spark 作业运行过程图如图 2 所示。

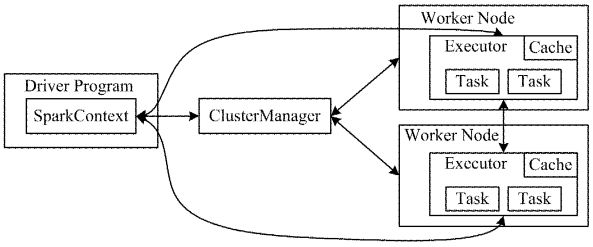

图 2　Spark 作业运行过程图

从 Spark 的作业运行过程来看，共享集群调度器可以控制对应用 executors 的调度，本文所研究的数据本地性主要集

中在将应用的各个进程根据数据块密度分配到相应的节点，主要考虑应用的“节点数据本地性”。

2.4 共享集群资源管理

在共享集群资源的管理方面，针对资源的分割和隔离的方式，目前有两种方案：1)基于虚拟机的虚拟化技术的解决方案(如 Xen，VMware 和 KVM 等)；2)基于容器的虚拟化实现(如 Linux-VServer，Linux 容器(LXC)和 Docker[15])。由于容器技术性能开销小，资源利用率高，并且通常可以在 1s 内启动，目前主流的数据中心采用容器作为资源管理和调度的基本单位[16]。

随着越来越多的应用开始部署在公有数据中心或私有数据中心之上，集群管理框架也得到了快速发展，如 Mesos，Yarn 和 Kubernetes[13] 等资源管理框架，这些框架支持以容器作为资源分割和调度的基本单位，为作业提供运行资源和运行环境隔离。这里以 Kubernetes 作为本文所使用的集群资源管理框架。

Kubernetes 是 Google 团队发起并维护的开源容器集群管理系统，其在设计过程中借鉴了 Google 的 Borg 集群管理系统[13]，提供应用部署、维护、扩展等功能，Kubernetes 支持用户自定义调度器，这些调度器以乐观锁的方式进行并发调度，用户能方便地利用 Kubernetes 管理共享集群管理跨机器运行 Docker 容器化的应用。

资源管理框架 Kubernetes 中有一些核心的概念，在设计本文提出的基于数据块密度的调度策略时会使用到，如 pod，replication controller 和 service 等，其中 pod 是 Kubernetes 创建、调度和管理的最小部署单元，包括一个或多个容器，replication controller 主要是控制 pod 副本的数量，service 为 pod 提供访问服务，同时也为 pod 副本之间提供负载均衡。

3 设计与实现

本文以共享集群管理框架 Kubernetes 的调度策略为研究对象，以 Spark 批处理作业为例，提出一种基于数据块密度的调度策略，目的是通过提高数据本地性来优化共享集群中应用程序的性能，进而提高集群调度器的调度质量。在研究基于数据块密度调度策略之前，本文首先对目前主流的调度架构和调度策略进行分析，并分析这些架构与策略存在的不足，最后以此提出基于数据块密度调度策略的设计和实现。

3.1 目前主流的资源管理器调度架构

从共享集群的资源调度器的架构来看，文献[6]将调度器分为 3 种：中央式调度架构、双层式调度架构和基于共享状态调度架构，如图 3 所示。调度器架构区分依据主要是调度的并行度和调度器对集群状态的感知。

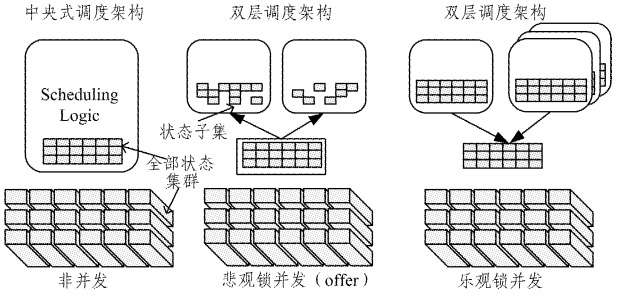

图 3 调度架构

其中，中央式调度器能够感知到整个集群的状态信息，但是其调度过程是非并行的，调度效率低且扩展性差；双层式调度器有一个资源管理器，该资源管理器能够感知集群状态，将资源通过 offer 的方式提供给多个并行且相互独立的调度框架，由于各个调度器无法感知到集群的整体使用状况，资源管理器通过轮询的方式进行资源分配，并且使用“悲观锁”的机制进行并发，会造成调度响应较慢。调度效率低的问题；基于共享状态的调度架构中，每个调度器都能够感知到集群状态，同时采用“乐观锁”的方式进行并发控制，能够实现调度器的高并发和高可扩展性。

本文所使用的资源管理平台 Kubernetes 采用了基于共享状态的调度架构，每个调度框架都可以感知到整个集群的状态信息，各个调度框架之间通过“乐观锁”维持并发，这样的设计提升了调度器的灵活性和调度速度，同时也支持自定义调度模块，并且该调度模块可以感知整个集群的状态，为实现本文所提出的调度策略带来了诸多方便。本文基于此为作业调度增加调度模块，对大数据处理应用提供基于数据块密度的调度策略，提升应用的数据本地性，进而提升应用的性能。

3.2 目前主流的资源调度器调度策略

目前主流的调度策略包括 3 种：Spread，Binpack 和 Random。Spread 和 Binpack 这两种调度策略都是根据节点可用的 CPU、内存和已经运行的资源量来计算节点的排名，而 Random 策略则不考虑这些因素，随机地从集群中选择一个合适的节点进行调度。Spread 策略将作业调度到资源使用量最小的节点，可以实现整个集群的负载均衡，而 Binpack 策略正好相反。

以 Spark 应用为例，目前主要有 Standalone，Spark on Yarn 和 Spark on Mesos 3 种部署模式。在 Standalone 模式中可以选择 Spread 和 Binpack 两种调度策略，采用 Spread 策略会将 Spark 作业的 executors 分配到尽可能多的 Node 上，可以提供良好的数据本地性，而使用 Binpack 调度策略，会将 Spark 应用的 executors 优先分配到一台机器中，数据本地性效果比较差。Mesos 的 offer 机制相当于 Binpack 策略，本地性较差。而在 Yarn 集群中，只有采用动态调度策略，并且设置初始 executors 数量为 0，且出现作业计算任务 pending 时才会根据任务的本地性进行调度。

Kubernetes 框架默认的调度器的调度策略分为两个阶段：Predicates 阶段和 Priorities 阶段。Predicates 阶段相当于过滤器，判断节点能否作为 pod 的候选节点；Priorities 阶段是对候选节点进行评分，即为候选节点设置优先级，Kubernetes 的调度器将该 pod 调度到评分最高的节点。节点评分公式如式(1)所示：

$$nodeFinalScore=\sum_{i=1}^{n} priorityFun_i * weight_i \quad (1)$$

其中，$priorityFun_i$ 代表调度算法中的第 i 个优先级指标，$weight_i$ 代表该指标的权重，节点最后的得分是所有指标得分的和。Kubernetes 的默认调度器可以通过加大集群的负载均衡指标的权重来实现 Spread 策略，但是在并不能保证有效性，尤其是在待处理数据分布不均匀的情况下，通过 Spread 策略来提升数据本地性不能够根据待处理数据所在的节点的数据量分配等比例的计算资源。

3.3 基于数据块密度调度策略的设计与实现

在共享集群中，一个作业往往使用部分计算节点中的资源，待处理数据也是存储在部分的节点中，本文提出基于数据块密度的调度策略旨在在各个节点上为作业分配数据块等比例的计算资源，以此来提高调度器的调度质量，提升应用程序的性能，尤其对于分布式作业。

本文将 Kubernetes 作为集群的资源管理框架，并在此基础上实现对 Spark 作业的调度，其运行模型如图 4 所示。本文提出基于数据块密度的调度策略来解决共享集群中节点数据本地性的问题，即尽可能地将作业的 pod 调度到待处理的数据所在的宿主机。第 i 个机架上的节点的集合如式(2)所示：

$$r_i=\{node_{i,1},node_{i,2},\cdots,node_{i,n-1}\},n\geqslant 0 \tag{2}$$

共享集群由式(3)所示：

$$C=\{r_0,r_1,r_2,\cdots,r_{n-1}\},n\geqslant 0 \tag{3}$$

该集群应用所需要的资源 $J_i=\{pod_{i,0},pod_{i,1},\cdots,pod_{i,n-1}\}(n\geqslant 0)$，$pod_{i,n}$ 代表分配给作业的第 $n+1$ 个 pod。

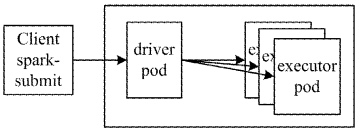

图 4　Spark 作业在 Kubernetes 集群中的运行模型

因此，共享集群的调度问题可以描述为：对于一个作业 i，经过 Predicates 和 Priorities 两个阶段，选取一组 Node，将作业的 pod 集合绑定到这组 Node 上，本节要解决的本地性问题的思路是尽量将 pod 绑定到待处理数据所在的 Node 上。

作业所处理的数据存储在 HDFS 文件系统中，数据存储块的大小为 128M，副本系数为 3，HDFS 文件系统会将一块数据在一个 Node 上存储一份，在同一个机架中其他 Node 存储一份，在另一个机架存储一份。通过式(4)对一个文件进行表示，用式(5)的关系矩阵表示文件数据块和存储节点的对应关系。

$$F=\{block_0,block_1,\cdots,block_{k-1}\},k\geqslant 0 \tag{4}$$

其中，$block_k$ 表示文件的第 k 块数据。

$$M=\begin{bmatrix} m_{0,0} & \cdots & m_{0,k-1} \\ \vdots & \ddots & \vdots \\ m_{n-1,0} & \cdots & m_{n-1,k-1} \end{bmatrix},k\geqslant 0,n\geqslant 0 \tag{5}$$

其中，$m_{i,j}=0$ 或 1，满足 $\sum_{i=0}^{n-1} m_{i,j}=3$，表示每块数据有 3 个副本。

每个 Node 的剩余资源为 $(cpu,memory)$，分别代表该节点剩余可用 CPU 核数和内存容量，一个作业中的每个 pod 的资源需求量也用该二元组表示。假设作业 A 包含 t 个 pod，则基于数据块密度来决定第 i 个 Node 可以创建 pod 的数量，由式(6)计算得到：

$$num_i=\min\left(\frac{\sum_{j=0}^{k-1} m_{i,j}}{k}\cdot t,\frac{node_i.cpu}{pod.cpu},\frac{node_i.memory}{pod.memory}\right) \tag{6}$$

其中，num_i 表示在第 i 个节点启动 pod 的数量。

假设 $node_0$ 的数据块向量为(1,1,1,0,0)，$node_1$ 的数据块向量为(0,1,1,1,0)，若直接用式(6)进行计算，则会造成为 $node_1$ 分配多余的计算资源，因为已经为 $node_0$ 分配了第二、三个数据块的处理 pod，因此需要剔除已经分配资源的数据块。同时，由于在计算 pod 数量的过程中使用的是向下取余的方式，会造成在 Node 上分配的资源无法满足作业请求的资源总量，本文使用 num 的小数部分进行从大到小排序，然后依次取相应的节点作为一个 pod 的选择节点，以此来确保作业的资源分配。

若只考虑每个节点都符合 job 相应的资源，则该 job 的调度策略的伪代码如算法 1 所示。算法变量表如表 1 所列。

算法 1　基于数据块密度的调度策略

```
for i in 0:nodes.length do
  nodeVector=nodes[i].vector-(nodes[i].vector&tmp);
  tmp |=nodes[i].vector;
  num=min(sum(nodeVector)/nodeVector.length*job.nums),
  nodes[i].cpu/job.pod.cpu,
        nodes[i].memory/jobs.pod.memory();
 if(num!=0) then
        nums+=floor(num);//向下取整
        nodes[i].num=num;
            if(nodes[i]>1) then
              results.put(nodes[i],floor(num));
            fi
   fi
   if(nums>job.nums) then
        break;
   fi
end for
// 按照 nodes[i].num一对 nodes 进行排序
// floor(nodes[i].num)降序排序
for i in 0:(job.nums-nums) do
results.put(nodes[i].node,results.getOrDefault(nodes[i],0)+1);
end for
```

表 1　算法变量表

变量名	变量描述
nodes	*nodes* 代表节点数据块关系矩阵，假设已经按照节点数据块数量降序排序；*nodes*[*i*].*vector* 表示第 *i* 个节点的数据块向量；*nodes*[*i*].*node* 表示第 *i* 个节点的节点元数据；*nodes*[*i*].*num* 表示在第 *i* 个节点分配 *num* 个 pod。
nums	*nums* 代表当前已确定资源的 pod 的数量。
job	*job* 代表 *job* 的元信息；*job*.*nums* 代表 *job* 所需要的 *pod* 的数量；*job*.*pod*.*cpu* 和 *job*.*pod*.*memory* 代表每个 pod 所需要的 cpu 和内存的大小。
results	*results* 代表选择的节点，结构是 Map〈*key*，*value*〉，*key* 为节点的元信息，*value* 是 pod 数量。
tmp	*tmp* 代表值为 0 对数据块关系向量，记录已经分配 pod 的数据块。

在每个 Node 的资源足够用的情况下，算法 1 能够快速完成资源的申请并启动作业，但是，在一个基于共享状态的调度模型中，为了提高作业调度的并行度，采用乐观锁来实现资源的分配，这样在多个作业并发提交的情况下，会出现资源竞争的情况。对于资源竞争的情况，需要进行重新调度，重新调度可以分为两种：局部重新调度和全部重新调度，局部重新调度只调度那些资源分配出现冲突的 pod，全局重新调度是对该作业的资源请求进行重新调度。

本文采用局部重新调度的策略，这样可以降低重新调度资源竞争的概率。在进行局部重新调度的策略中，为了让作

业快速获取运行资源，此轮调度不再为调度指定具体的Node，而是放宽 Kubernetes 的 Predicates 阶段对 Node 的指定，为 Priorities 阶段加入数据的本地性级别的指标。

$$score=10-LocalLevelValue \tag{7}$$

其中，*LocalLevelValue* 代表数据本地性指标的值，包括 5 个级别，其值分别为{*ProcessLocal*，*NodeLocal*，*RackLocal*，*NoPref*，*Any*}={0，2，4，6，8}。

本节主要分析影响共享集群中应用性能的一个关键性因素，即数据本地性问题。目前，数据中心管理系统大多是两层调度模型，未在调度时考虑作业待处理数据的分布问题，大部分通过 *Spread* 策略将作业的计算资源均匀地分散到集群中来增加数据的本地性。基于此，本文提出了基于作业待处理数据块密度的作业调度策略，将计算资源分配到待处理数据所在的节点，并且根据数据块的密度分配等比例的计算资源；另外，由于 *Kubernetes* 使用共享持久化状态的方式支持调度并发，并使用乐观锁来维护调度器的并发，在调度过程中出现资源竞争的情况时，本文提出了局部调度重试的策略，并根据数据本地性级别来对候选主机节点进行评级，确保调度的速度和质量。

4 实验

4.1 实验环境

本文实验所使用到的硬件设备是由 11 台 ThinkServer TS430 服务器构成，配置是 CPU Xeon E3-1220 3.1GHz 16 核处理器，内存为 16GB 的 DDR3 内存条，硬盘大小为 512GB，2 个千兆以太网卡，其中 10 台服务器作为工作节点，1 台服务器作为主节点。其他硬件设备是 1 台思科 24 口千兆交换机，以及 2 台高清 IP 摄像头。

本文实验所涉及的软件环境包括两个方面：1）是集群的软件环境，2）具体应用的运行环境。集群的软件环境包括：操作系统 CentOS 7，集群管理软件 Kubernetes 1.51，分布式文件系统 Hadoop 2.7.3，Docker1.12.6，以及 JDK1.8。应用软件运行环境为 spark-2.1.0，JDK1.7。详细配置如表 2 所列。

表 2 主节点配置

硬件配置		操作系统	基础软件配置
CPU 类型	Intel(R) Xeon E3-1220 3.1GHz	CentOS 7.3.1611	JDK1.8，Hadoop-2.7.3，Docker 1.12.6，Kubernetes 1.51
CPU 核数	16		
内存	16GB		
IP	192.168.3.110		

表 3 从节点配置

硬件配置		操作系统	基础软件配置
CPU 类型	Intel(R) Xeon E3-1220 3.1GHz	CentOS 7.3.1611	JDK1.8，Hadoop-2.7.3，Docker 1.12.6，Kubernetes 1.51
CPU 核数	16		
内存	16GB		
IP	192.168.3.100～192.168.3.109		

4.2 实验结果

本实验对 3 种调度策略进行测试，即 Spread 策略、Binpack 策略和基于数据块密度的调度策略。

在对基于数据块密度调度策略（DBD）的评估中，采用同 Spread 和 Binpack 调度策略对比的方式，使用 HiBench 的 SparkBench 基准测试中的 5 个典型 Spark 作业进行测试，包括微基准测试 WordCount，Sort，TeraSort，机器学习基准测试 Kmeans 算法和 Web 搜索基准测试 PageRank，其中微基准测试的数据是 2.5GB。每个作业启动 8 个 executor 进行测试，这 8 个 executor 分别按照 Spread，Binpack 和 DBD 在平台中进行调度，其中 TeraSort 测试的数据分布和调度分布如图 5 所示。

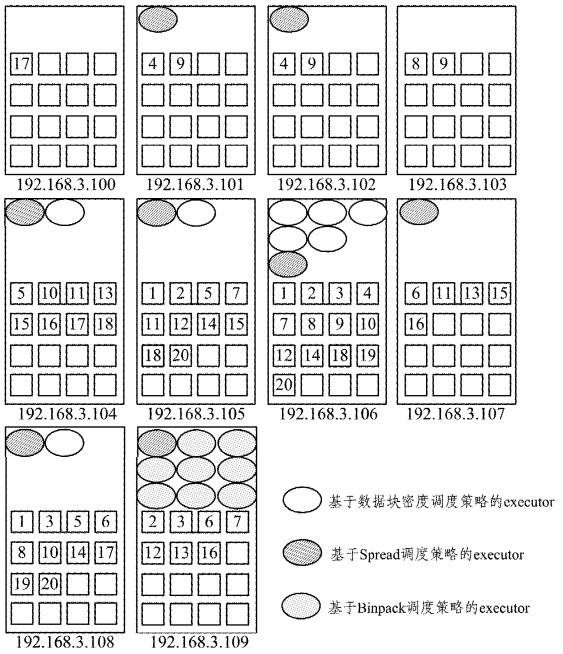

图 5 TeraSort 测试的数据块分布和调度分布图

如图 5 所示，实验中使用到的文件在 HDFS 文件中的分布用数字标号表示，一共包括 20 块数据，基于数据块密度的调度策略的数据本地性达到 90%，而 Spread 和 Binpack 两种策略应用的数据本地性只有 60%和 35%。3 种调度算法对应用性能的影响通过运行时间进行比较，基准测试的运行时间对比如图 6 所示，图 6 中的数据是作业在 3 种不同的调度策略下独立运行 10 次的平均值。

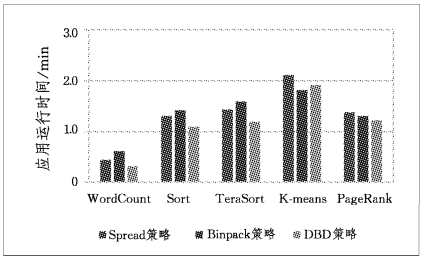

图 6 不同调度策略下应用完成时间的对比

从图 6 中可以看出，基于数据块密度的调度策略能够不同程度地提高大部分 Spark 典型应用的性能，在本文所使用的 5 种测试样例中，有 4 个应用在基于数据块密度的调度策略的作业完成时间有所减少。在微基准测试中，这些测试应用属于数据密集型任务，针对密度较大的数据块，大调度策略对作业的运行时间的影响也相对较大，其中 WordCount 和 TeraSort 两个算法比 Spread 调度策略少了 27%和 16.7%的运行时间；另外对于 Web 搜索算法 PageRank，本文提出的调度策略能够减少 7.6%的运行时间；对于机器学习算法 Kmeans，其运行时间介于 Spread 和 Binpack 调度策略之间。

总体看来，本文提出的调度策略对于计算密集型的应用效果不佳，如 Kmeans 算法和 PageRank 算法，这些算法需要大量的迭代运算，属于计算密集型作业。下一步，我们计划深入研究如何提高共享集群中计算密集型作业的性能。

结束语　在数据中心这样的共享集群中，往往运行着大量的数据处理作业，调度器的目的是提升集群的资源利用率和吞储量，集群的资源利用率和吞储量不但受调度速度的影响，也受调度质量的影响。调度的质量影响着作业的性能，进一步影响着集群的资源利用率和吞吐量。本文提出了基于数据块密度的调度策略，目的是提高共享集群中作业的数据本地性，从而优化其性能。该策略通过计算资源根据数据块的密度为作业进行分配，提升应用运行过程中的数据本地性，从而提升资源调度的调度质量。在实验测试过程中应用作业达到 90%的数据本地性，加快了应用的运行速度，并且可以有效降低集群的网络 I/O，对于数据密集型任务，比 Spread 策略缩短了 20%左右的作业运行时间。

基于数据块密度的调度策略能够在一定程度上提升作业的本地性。本文只对 Spark 作业进行了微基准测试，在下一步的工作中，我们将对更多的算法和计算框架进行测试，研究基于数据块密度的调度策略对不同类型算法性能的影响，并且深入研究如何提高共享集群中计算密集型作业的性能。

参 考 文 献

[1] DEAN J, GHEMAWAT S. MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.

[2] WHITE T. Hadoop: The definitive guide[M]. O'Reilly Media, Inc., 2012.

[3] ZAHARIA M, CHOWDHURY M, FRANKLIN M J, et al. Spark: cluster computing with working sets[J]. HotCloud, 2010, 15(1): 10.

[4] SHVACHKO K, KUANG H, RADIA S, et al. The hadoop distributed file system[C]// 2010 IEEE 26th symposium on mass storage systems and technologies (MSST). IEEE, 2010: 1-10.

[5] DELIMTROU C, SANCHEZ D, KOZYRAKIS C. Tarcil: reconciling scheduling speed and quality in large shared clusters[C]// Proceedings of the Sixth ACM Symposium on Cloud Computing. ACM, 2015: 97-110.

[6] SCHWARZKOPF M, KONWINSKI A, ABD-EL-MALEK M, et al. Omega: flexible, scalable schedulers for large compute clusters[C]// Proceedings of the 8th ACM European Conference on Computer Systems. ACM, 2013: 351-364.

[7] ZAHARIA M, BORTHAKUR D, SEN SARMA J, et al. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling[C]// Proceedings of the 5th European conference on Computer systems. ACM, 2010: 265-278.

[8] DEAN J, GHEMAWAT S. MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.

[9] BEZERRA A, HERNÁNDEZ P, Espinosa A, et al. Job scheduling for optimizing data locality in Hadoop clusters[C]// Proceedings of the 20th European MPI Users' Group Meeting. ACM, 2013: 271-276.

[10] Kubernetes[OL]. http://kubernetes.io/

[11] HUANG S, HUANG J, DAI J, et al. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis[C]// 2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW). IEEE, 2010: 41-51.

[12] Apache Hadoop 官方网站[OL]. https://hadoop.apache.org/

[13] 浙江大学 SEL 实验室. Docker——容器与容器云[M]. 北京：人民邮电出版社，2016.

[14] 孙瑞琦，杨杰，高瞻，等. 一种提高虚拟化 Hadoop 系统数据本地性的资源调度方法[J]. 计算机研究与发展，2014(S2)：189-198.

[15] Docker[OL]. https://www.docker.com.

[16] REY J, Cogorno M, Nesmachnow S, et al. Efficient Prototyping of Fault Tolerant Map-Reduce Applications with Docker-Hadoop[C]// IEEE International Conference on Cloud Engineering. 2015: 369-376.

(上接第 489 页)

[5] HUI Z W, HUANG S. A Formal Model For Metamorphic Relation[C]// Proc. of Decomposition Software Engineering (WCSE 2013). IEEE, 2013: 64-68.

[6] ASRAFI M, LIU H, KUO F C. On Testing Effectiveness of Metamorphic Relations: A Case Study[C]// 2011 Fifth International Conference on Secure Software Integration and Reliability Improvement. IEEE Computer Society Washington, DC, USA, 2011: 147-156.

[7] KANEWALA U, BIEMAN J M. Using Machine Leaning Techniques to Detect Metamorphic Relations for Programs without Test Oracles[C]// Proceeding of the Software Reliability Engineering (ISSRE 2013). IEEE, 2013: 1-10.

[8] HUI Z W, HUANG S. A Formal Model For Metamorphic Relation[C]// Decomposition Software Engineering (WCSE 2013). IEEE, 2013: 64-68.

[9] CHEN T Y, HUANG D H, TSE T H, et al. Case studies on the selection of useful relations in metamorphic testing[C]// Proc. of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004). IEEE, 2004: 569-583.

[10] 董国伟，聂长海，徐宝文，等. 基于程序路径分析的有效蜕变测试[J]. 计算机学报，2009，32(3)：1002-1013.

[11] MAYER J, GUDERLEI R. An empirical study on the selection of good metamorphic relations[C]// Proc. of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06). Chicago, USA, 2006: 475-484.

[12] CAO Y, ZHOU Z Q, CHEN T Y. On the Correlation between the Effectiveness of Metamorphic Relations and Dissimilarities of Test Case Executions[C]// International Conference on Quality Software. 2013: 153-162.

[13] 侯雪梅，于磊，张兴隆，等. 面向对象软件测试的蜕变关系构造方法[J]. 计算机应用，2015，35(10)：2990-2994.

[14] OFFUTT A J, LEE A, ROHERMEL G, et al. An experimental determination of sufficient mutant operators[J]. ACM Transactions on Software Engineering & Methodology, 2000, 5(2): 99-118.