计算机应用
Journal of Computer Applications
ISSN 1001-9081,CN 51-1307/TP

《计算机应用》网络首发论文

题目：大规模短时间任务的低延迟集群调度框架
作者：赵全，汤小春，朱紫钰，毛安琪，李战怀
收稿日期：2020-10-12
网络首发日期：2021-01-14
引用格式：赵全，汤小春，朱紫钰，毛安琪，李战怀．大规模短时间任务的低延迟集群调度框架[J/OL]．计算机应用．
https://kns.cnki.net/kcms/detail/51.1307.TP.20210114.0911.020.html

Journal of Computer Applications
计算机应用
ISSN 1001-9081
CODEN JYIIDU

doi:10.11772/j.issn.1001-9081.2020101566

大规模短时间任务的低延迟集群调度框架

赵全，汤小春*，朱紫钰，毛安琪，李战怀

(西北工业大学 计算机学院,西安 710129)

(*通信作者电子邮箱 tangxc@nwpu.edu.cn)

摘 要: 大规模数据分析环境中，经常存在一些持续时间较短、并行度较大的任务。如何调度这些低延迟要求较高的并发作业是目前研究的一个热点。现有的一些集群资源管理框架中，集中式由于主节点的瓶颈，无法达到低延迟的要求；而一些分布式调度器虽然满足了低延迟的任务调度，但在最优资源分配以及资源分配冲突方面存在一定的不足。从大规模实时作业的需求出发，设计和实现了一个分布式的集群资源调度框架，满足了大规模数据处理的低延迟要求。首先提出了两阶段调度框架以及优化后的两阶段多路调度框架；然后针对两阶段多路调度过程中存在的一些资源冲突问题，提出了基于负载平衡的任务转移机制，解决了各个计算节点的负载不平衡问题。最后使用实际负载以及一个模拟调度器的运行，对大规模集群中任务调度延迟进行了验证。对于实际负载，调度延迟控制在理想调度的12%以内；在模拟环境下，与集中式调度器比较，短时间任务的延迟能够减少40%以上。

关键词: 低延迟；分布式调度；两阶段调度；负载平衡；贪心调度

中图分类号:TP31 文献标志码：**A**

Low latency cluster scheduling framework for large scale short time tasks

ZHAO Quan,TANG Xiaochun*, ZHU Ziyu, MAO Anqi, LI Zhanhuai

(School of Computer Science, Northwestern Polytechnical University, Xi'an Shaanxi, 710129, China)

Abstract: There are always some tasks with short duration and high concurrency in a large-scale data analysis environment. How to schedule these concurrent jobs with low latency requirements is a hot research topic. The centralized type cannot meet the requirements of low latency due to the bottleneck of the master node in the existing cluster resource management frameworks. Although some distributed schedulers meet the requirements of low-latency task scheduling, the optimal resource allocation and resource allocation conflict are not quite efficient. A distributed cluster resource scheduling framework was designed and implemented to meet the low latency requirement of large-scale data processing from the need for large-scale real-time jobs. A two-stage scheduling framework and an optimized two-stage multi-path scheduling framework were proposed. Secondly, a task transfer mechanism based on load balancing was proposed to solve some resource conflicts in two-stage multi-path scheduling, which solved the load imbalance problems of each computing node. At last, the task scheduling delay in large-scale clusters was simulated and verified by using actual load operation and a simulated scheduler. For the actual load, the scheduling delay is controlled within 12% of the ideal scheduling. In the simulated environment, the delay of short-time tasks can be reduced by more than 40% compared with the centralized scheduler.

Keywords: low latency; distributed scheduling; two-stage scheduling; load balancing; greedy scheduling

0 引言

近年来,数据中心上的数据分析作业常常是一些运行时间短的流处理作业,而且要求这些作业共享一套集群资源以便减少基础设施的成本,例如,既存在一些结构化的数据查询作业,也存在一些实时数据监控以及数据分析作业[1-5]。面对这些规模庞大并且延迟要求较低的流处理作业,一些流处理计算框架,如 Flink[1]、Dremel[6]、Spark[7]等开始被大型互联网

收稿日期: 2020-10-12; 修回日期: 2020-12-11; 录用日期: 2020-12-14。

基金项目：国家重点研发计划项目（2018YFB1003400）。

作者简介：赵全（1997—），男，河北衡水人，硕士研究生，主要研究方向：集群资源管理； 汤小春（1969—），男，陕西汉中人，副教授，博士，主要研究方向：图数据管理、分布式计算、集群资源管理； 朱紫钰（1996—），女，河北邯郸人，硕士研究生，主要研究方向：集群资源管理； 毛安琪（1996—），女，河南洛阳人，硕士研究生，主要研究方向：集群资源管理；李战怀（1961—），男，陕西咸阳人，教授，博士，CCF 会员，主要研究方向：海量数据管理、大数据计算。

企业使用。这类作业运行时,任务往往被分散到上千台机器上运行,任务要么从磁盘获得数据进行计算,要么使用存储在内存的数据进行计算,作业的响应时间常常在秒级。例如,使用Spark计算一个数据大小为1 GB的查询作业,运行时间仅仅1 s左右的时间。当这样的作业大量出现在一个数据中心并共享集群计算资源时,作业之间往往会出现相互的干涉或者对计算资源的竞争,导致部分作业被延迟执行或者出现执行失败的情况[7]。本文希望建立一套新的集群资源调度框架,提供一种高效、共享的资源调度环境,提高整个集群的资源使用率,为各种大规模并行作业及时分配计算资源,保证每个作业能够快速得到响应。

由短时间(毫秒级)任务组成的作业,其作业的调度过程具有较大的挑战。在这些作业中,不但包括一些低延迟的流处理任务,也包括一些为了得到资源的公平分配和减少长时间滞留的任务,将长时间运行的批处理作业分解成大量的短时间运行的任务。例如,一些大规模的数据处理中,一个作业的执行流程通常被分解成一个有向无环图(Directed Acyclic Graph, DAG)[8-10](https://tez.apache.org/),DAG中的一个顶点代表一个独立运行的操作,边代表数据流向。在执行中,通过对数据进行分片,将操作和数据结合来形成一个一个可以独立运行的计算任务。因此,大数据处理作业就变成了大量的短时间运行的并行任务。当作业中的每个任务的运行时间在几百毫秒内,为了满足低延迟的要求,集群资源调度的决策能力必须具有较大的吞吐量和较低的延迟。例如,假如一个集群系统通常包含上万台机器,每台机器如果包含16核处理器的话,那么每一秒钟调度100 ms的任务数量就可能达到百万次的级别。另外,调度过程的延迟一定要低,对于一个100 ms的任务,如果调度延迟以及任务的等待时间超过执行时间的十分之一,即数十毫秒,这些都是用户所不能忍受的情况。最后,在客户对于实际系统的使用中,面对大量的短时间交互任务,应用程序对于集群系统的可扩展性以及可靠性也是重点关注的问题。这些挑战导致现有的批处理集群资源管理系统不再能够满足用户的需求,而一些专用的流处理计算框架虽然满足实时性要求,却无法满足集群计算资源在不同作业之间的共享。

针对大规模且高度并行的秒级计算任务,改进现有的集群资源调度框架,实现不同作业对集群资源的共享,满足其低延迟和高吞吐量调度要求,存在一定的挑战。现有的集群调度框架,如Mesos[11]、Yarn[12]、Slurm[13]、Mercury[14]、Omega[15]、Sparrow[16]、低延迟框架[17]、Quincy[18]等,都无法满足大规模秒级任务的调度。对于这种种类繁多且任务并行度较大的计算作业,如果只使用单个调度器实例来分散任务的执行,这将是一件非常困难的事情。另外,为了满足高可扩展性以及高可靠性,大规模任务的复制和恢复也要求在秒级内完成,这也使得单个调度实例成为系统的瓶颈。

本文考虑了现有集群资源管理系统的特点,在调度框架方面,取消了集中调度器或者逻辑全局资源状态,让每个计算节点具有自主资源管理和控制能力,实现调度器的多个实例,将调度器实例部署在不同的计算节点上。这种调度器实例分散的方式以及计算节点的自治特性,使得可扩展性以及可靠性得到大大的提高。当系统中在某个时间段出现大量的作业时,可以通过增加调度器实例来满足高峰时段任务的调度需求;当某个调度器实例故障,可以快速启动新的调度器实例来替代故障调度器实例,满足用户作业的调度要求。这种分散的多调度实例带来了低延迟的好处的同时,也带来了一些矛盾。与集中式的单个调度器比较,多个分散的调度器实例可能导致资源分配的矛盾[15-16,19-21],即不同的调度器实例将各自的任务分配到同一个CPU核上,造成多个任务对同一个资源进行竞争,导致任务的阻塞,从而延迟了任务的完成。

本文设计和实现了一个分布式集群资源调度框架DHRM(Distributed Heterogeneous Resource Management),它是一种无全局资源状态、支持两阶段任务调度策略的集群资源调度框架。所谓两阶段任务调度,是指一个任务在调度过程中,首先由调度器向各个计算节点发送一个保持状态的请求,一旦请求到达计算节点上的资源管理器,计算节点就发送自身的负载状态信息到调度器;然后调度器进行决策后,向被选中的计算节点发送释放消息,启动任务在计算节点的执行,对于没有被选中的计算节点,发送取消消息,从这些计算节点删除保留状态的任务。DHRM提供了以下三种主要的策略来实现不同类型作业对于集群计算资源的共享:

1)两阶段多路调度策略。对于并行的作业,直接使用两阶段任务调度可能会导致作业完成时间的延长。作业的完成时间与作业中最后一个任务结束时间直接相关,如果一个作业的最后一个任务长时间得不到执行,那么作业的完成时间就延长,必须等待最后一个任务完成,整个作业才能认为结束。而最后一个任务的长时间等待使得两阶段调度策略的效果不能满足预期要求。两阶段多路调度可以同时为每个任务提供多个随机选择选项,使得作业中的每个任务可以同时选择较好的资源来解决此问题。不像两阶段调度每次仅仅为作业中的一个任务提供候选资源,两阶段多路调度为一个作业中的n个任务同时至少提供$d \bullet n$个计算节点($d>1$)。经过论证分析,不像两阶段调度策略,当作业的并行度增加时,两阶段多路调度策略的调度延迟并不会增加。

2)执行队列策略。对于集群节点上的CPU资源,可以划分为多个执行队列,每个队列包含一定的CPU资源、内存资源、带宽资源等。当集群计算节点上只存在CPU资源,如果采用传统的调度算法,可能导致队列头的一个长时间运行的任务阻塞其它短时间运行的任务,造成短时间运行的任务延迟增大。一些调度方案采用强力搜索策略,遍历每个可用资源和每个任务来匹配资源,缺点是调度的复杂度上升,调度开销增大。通过设置多个执行队列,可以保证CPU资源有空闲时,短时间运行的任务的快速调度。通过论证,这种方式并不会增加调度的开销,不同种类任务混合时,调度延迟也并没有明显的增加。

3)资源协调策略。两阶段多路调度方法存在两个性能瓶颈:第一,计算节点上的可以同时运行的任务数量不能很好地表示新到达的任务需要等待的时间; 第二,由于资源状态的变化以及网络通信的延迟,多个并行的调度器实例可能会遇到资源竞争的情况。通过资源协调策略,某些等待时间过长的任务可能会被分散到其它等待时间较短的计算节点的队列上。通过资源协调来解决资源分配冲突,实现计算资源的负载平衡。相比较单纯的两阶段多路调度来说,可以大幅度减少作业的平均完成时间。

共享集群资源时,需要对每用户资源使用量进行限制。DHRM 通过计算节点上的多个队列,来保证全局计算资源的分配目标,为不同的用户设置不同的资源使用量限制。通过资源使用限制,来确保各个作业能够共享集群计算资源,不会导致某些用户由于缺乏资源而放弃对资源的共享使用。

本文将 DHRM 部署在包含 16 台服务器的集群上。通过对事务处理性能委员会(Transaction Processing Performance Council, TPC)给出的基准测试程序以及大规模的模拟作业进行调度，与理想状态对比,DHRM 的响应时间在 10%以内,任务在队列中的等待延迟平均在 12 ms 以内。对于大规模的集群,DHRM 可为任务较短的任务提供较短的响应时间。通过设计调度模拟器,仿真结果表明,随着集群规模增加到成千上万 CPU 内核,DHRM 将继续表现良好。实验结果表明,DHRM 分布式调度是集中调度的一种可行的替代方案,可以实现大规模并行任务的低延迟调度。

1 相关工作

对于集群资源管理系统,存在大量的文献研究集中式调度框架,也存在一些系统使用分布式调度框架和混合式调度框架。例如 Mesos 以及 Yarn 等系统,它们是集中式调度框架,无法满足大规模并发短时间任务的低延迟调度。DHRM 是一个分布式集群资源调度框架,能够满足大规模并发的短时间任务的低延迟调度要求。

Sparrow 的研究目的是减少任务的尾部延迟,它建议使用对冲请求,其中客户端将每个请求发送给两个服务器,并在收到第一个结果时取消剩余的未完成请求。它还描述了绑定请求,客户端将每个请求发送到两个不同的服务器,但是服务器直接就请求状态进行通信。当一台服务器开始执行请求时,它将取消另一台服务器。但是它无法估计不同的计算节点的负载状态,也无法满足集群计算节点上的负载平衡,造成资源饥饿,即一部分节点负载较重而另外一部分计算节点没有计算任务的运行。

针对分布式系统中的负载共享机制,Sparrow 使用随机技术,DHRM 采用仲裁。在负载共享系统中,对于每个任务的产生和任务的处理,缺省情况下,计算任务在产生的地方处理。当某个处理器上的负载超过其阈值限制,负载需要分散到其它处理器。分散过程要么采用接受者方式,即轻量级的计算节点随机选择一些处理任务,请求任务的转送;要么采用发送者方式,重量级的计算节点随机选择一些计算节点,向这些计算节点发送请求,其完成的过程通过随机采样的方式实现。DHRM 采用协调仲裁方式,轻量级的计算节点向协调器发送资源邀约,重量级的计算节点发送任务转送请求,协调器匹配后通知双方仲裁结果。

Quincy 的目标是计算机集群上的任务调度,类似于 Sparrow。Quincy 将调度问题变成图的最大流最小代价优化,目标是在集群资源在作业之间的公平共享、数据本地化。Quincy 的图调度支持更高等级的调度,但是,对于一个 2500 台机器的集群,它至少需要一秒钟的时间来计算任务的调度分配结果,因此,大规模集群环境下,Quincy 无法满足低延迟的调度要求。

其它许多调度框架旨在以粗粒度分配资源[22-23],这是因为任务往往需要长时间的运行,或者因为集群支持许多应用程序,每个应用程序都获取一定数量的资源并执行自己的任务级调度,例如 Mesos、Yarn、Omega 等。这些调度为了实现复杂资源公平调度策略而牺牲了请求的粒度。作为结果,它们存在反馈延迟[24],在调度秒级任务时无法提供低延迟和高吞吐量。对于高性能计算环境下的调度,它们针对具有复杂约束的大型作业进行了优化,其调度目标是最大吞吐量,它以每秒能够实现数十到数百个作业为调度目标,例如,Slurm。类似的系统还包括 Condor[25],它支持一些复杂特征的作业流调度,使用丰富的约束语言、作业的检查点以及群调度等。其匹配算法能够达到的最大调度量是每秒数十到数百个作业。

2 设计目标

本文的设计目标是为短时间任务提供低延迟调度框架,支持大规模并行任务的运行,满足不同作业对集群计算资源的共享。

资源调度的低延迟。相对于批处理工作负载,低延迟的工作负载通常具有更复杂的技术要求。因为批处理负载会长时间使用资源,所以调度的频率不高。而低延迟的负载,其任务的执行时间非常短,而且数量大,为了支持毫秒级别的任务调度,那么调度器就必须具有更快的调度频率,支持每秒钟百万级别的任务调度规模。此外,调度框架向应用程序提供低延迟服务后,调度程序就必须承受任务的失效恢复。

资源的细粒度共享。支持多个不同的作业对集群计算资源的共享,但是相对于粗粒度的资源共享方式,细粒度资源共享允许不同作业的多个任务并发共享资源。DHRM 提供细粒度的资源共享,允许多个任务对同一资源的并发访问,它是对现有的粗粒度集群资源管理的一种补充。

降低调度反馈延迟。现有的批处理资源调度框架中,任务完成后,释放资源,下一个任务才有可能获得该资源,这样会出现调度的反馈延迟,降低了资源的使用率。DHRM 通过计算节点资源的细粒度共享,减少反馈延迟。

DHRM 支持可扩展性和高可用性。通过可扩展性[26],提供更多的计算资源,降低任务的执行延迟;当存在大量的应用程序的低延迟流处理请求并超过可使用的资源总量时,DHRM 提供严格的优先级别和带权重的资源共享。DHRM 还提供每作业的资源使用约束,来保证各个不同的作业都能及时获得资源。当某个执行器发生故障时,可以通过快速启动新的执行器的方式来保证任务的失效恢复功能。

3 并行作业的两阶段调度

3.1 一些基本的概念

1)计算节点。集群中的一个机器,其主要任务是运行各种计算任务。一个集群中包含大量的计算节点。如果一个计算节点上安排了大量的任务,超过其可以并行执行的最大值,那么一些任务就会在计算节点的队列上排队。

2)作业。一个作业中包含 n 个任务,每个任务都会被安排到计算节点上。

3)调度器实例。一个进程,运行在某个计算节点上,负责将作业的任务映射到某个计算节点上。在该分布式资源调度框架中,调度器实例是有多个的,其数量最多不超过集群中机器的数量,作业可以被任何一个调度实例处理。在某一时刻下,可能会有多个包含不同数量任务的作业向该分布式资源调度框架请求注册。若是集中式调度器,则只有一个调度实例来处理大量作业的注册请求并进行资源的调度,这样会导致集中式调度器的调度压力过大。但在该分布式调度框架中,有多个分布式调度器实例供作业选择,作业可平均分配到各个调度器上进行注册,或者寻找较为“闲”的调度器实例进行注册。通过多个分布式调度实例可以并行地对大量作业进行调度,这大大降低了作业的调度延迟。但是多个分布式调度器实例对资源进行调度管理,有可能会出现多个调度器实例将同一块资源分给不同作业的任务的情况,造成较多的任务争抢相同的资源的情况,从而延长了作业的整体执行时间。该分布式调度框架采用两阶段的多路调度策略对不同作业的任务进行调度,关于两阶段的多路调度的详细内容将在后续的章节详细介绍。

4)作业响应时间。当一个作业的任务提交到调度器后,在其得到资源之前,这个时间被称为调度时间;任务获得资源后,在计算节点的队列上排队,当计算节点的并发执行数低于最大值时,任务实际开始执行,这段时间称为等待时间;当任务实际开始执行到任务结束期间,被称为执行时间。作业响应时间就是作业被提交到调度器,一直到最后一个任务完成,这个期间称为作业响应时间。当一个作业的延迟时间包括调度时间和等待时间的累计。如果采用零等待时间(相当于该作业中所有任务的最长执行时间)来调度作业的所有任务,可以得到一个理想作业响应时间。使用给定调度技术来调度作业的所有任务,可以得到一个具体的响应时间。理想作业响应时间和指定调度下的作业响应时间的差值叫延迟。

3.2 计算节点上的队列

对于分布式调度框架来说,各个计算节点上需要有队列。分布式调度器将任务提交到计算节点的队列后,任务在队列中处于等待状态。当计算节点上正在运行的任务数量低于设置的运行数量限制或者说存在可用计算资源的话,队列中等待状态的任务就被调度执行。最简单的调度方式是先进先出(First Input First Output, FIFO),也可以实施优先级别调度、短任务优先等策略。

对于 CPU 有关的资源分配,操作系统本身就支持按照时间片共享调度、或者按照优先级别抢占式调度等;因此,在各个计算节点上,DHRM 允许用户建立自己的队列,队列设置不同的优先级别。每个队列上用户可以设置自己的资源限制属性。

执行队列。执行队列为 CPU 任务的专用队列。执行队列具有三类属性:1)资源量。CPU 资源数量包括 CPU 核数、内存数量、IO 带宽以及网络带宽值。该限制值用来与要提交到队列的任务的资源使用量限制作比较。要提交到队列的任务中所设置的资源限制值若超过该值,任务的提交将被拒绝。2)队列优先级。表示队列间的优先级。其决定调度最先运行哪个队列中的任务。该值较大的队列中的任务将被优先运行。若队列间的优先级相同,则按照请求被提交到队列的时间顺序运行请求。3)同时可运行的任务数。队列内同时可运行的任务数。当前正在运行的任务数若达到该数值,下一个应该运行的任务将一直处于等待状态,直到当前正在运行的某个任务运行结束后该任务才会被启动。

3.3 任务的两阶段调度

DHRM 的候选资源选择策略是借鉴文献 sparrow[16]方法,即两种随机选择策略来实现获得候选的计算节点,该策略使用无状态随机方法提供了较低的预期任务等待时间。但是,DHRM 对两种随机选择策略进行了改进:不同于 sparrow 的依据队列长度来决策计算节点方式,DHRM 会将任务放置在两个随机选择的工作计算机中负载最小的一个上。与使用随机选择相比,以这种方式调度任务可显著缩短预期的等待时间。

3.3.1 调度过程

本文将两种随机选择策略直接应用于并行作业调度,其整个过程分为两个阶段:第一阶段,调度程序为作业中的每个任务随机选择两个计算节点的队列,并向每个计算节点的队列发送一个任务请求,任务请求中仅包含用户的任务资源描述。每个计算节点上的队列都会用当前队列状态回复调度程

序。第二阶段,调度程序收到全部的返回的消息后,根据队列的状态信息,选择一个负载最轻的计算节点的队列,即等待时间最短的队列,将任务放在该队列上。调度程序重复此过程,直到作业中的所有任务全部获得计算资源。

其实现过程如图 1 所示。调度器为任务 1 选择两个计算节点的队列,然后发送 hold(保留)消息(图 1 中 1a:hold 及 1b:hold)。根据返回消息,选择其中一个最优队列(图 1 中的第二个队列),然后调度器向选中的计算节点发送 run(运行)消息(图 1 中 1b:run),使得任务在计算节点上变成等待状态,对于另外一个计算节点的队列,则发送 del(删除)消息(图 1 中 1a:del),取消任务的保留。一旦任务 1 调度完成,调度器开始任务 2 的调度,任务 2 的调度过程与任务 1 相同。该过程被称为为两阶段调度。

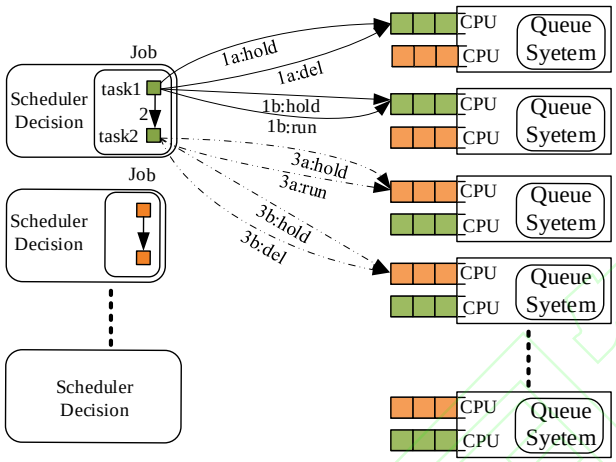

图 1 任务的两阶段调度过程

Fig. 1 Process of two-stage scheduling for tasks

3.3.2 队列状态的表示

在图1所示的第一个阶段中,计算节点的队列向调度器返回队列的状态。最简单的方式是返回队列中执行任务的数量和处于等待中的任务的数量,调度器根据队列的长度选择一个最短的队列作为任务提交的目标队列。但是,在高负载集群环境下,这种方式的调度性能是比较差的。主要原因是因为队列长度只能提供一种粗粒度的等待时间标准。例如,假如在两个不同的计算节点上存在两个队列;第一个计算节点的队列上存在两个任务,执行时间之和为 200 ms;而第二个计算节点的队列上只有一个任务,执行时间为 300 ms,如果任务按照队列长度来决策调度节点,那么任务就会提交到后者,这将导致任务不必要的延迟。与选择前者相比,延迟时间增加 100 ms。因此,依据队列长度来调度,效果较差,通过估计每个队列中任务的执行时间来分配资源,即确定队列的状态,然后决策任务的分散节点,这将是一个较好的方案。

对于 DHRM,将任务提交到某个节点的队列后,需要经过等待和执行过程。首先给出各个队列状况的表示方式、任务的参数形式、等待时间的计算方法和决定一个任务投入到哪个节点的调度依据。

对于执行队列,考虑 CPU 资源以及内存资源,暂时不考虑其它 IO、带宽等计算资源。调度器中作业 j_k 的任务可表示为:

$$t_k(m,c,et,P) \tag{1}$$

其中: k 表示来自第 k 个作业, m 表示内存需求量, c 表示 CPU 使用时间, et 表示作业估计执行时间, P 表示优先级(由上面给出的算法得出)。

任务的等待时间, 是指一个任务从提交到队列上进入排队状态到其开始正式运行所经历的时间。

现设任务投入第 l 个节点的第 i 个队列, 并设该任务需要的等待时间为 w_i, 有如下三种情况:

当队列中无排队的任务时且队列尚可以继续提交任务, 等待时间 w_i 为

$$w_i = 0 \tag{2}$$

当队列中无排队的任务时且不能继续提交任务时, w_i 等于队列中剩余时间最小的任务的剩余时间 pt_i。

$$w_i = pt_i \tag{3}$$

当队列中运行着任务, 并且存在 n 个排队的任务, 则等待时间等于当前队列中剩余时间最小的任务的剩余时间 pt_i 与该队列中来自 n 个作业中的所有排队的任务的估计执行时间之和。即

$$w_i = pt_i + \sum_{k=1}^{n} t_k.et \tag{4}$$

假如队列 i 中的可用CPU核数为 q_i,内存大小为 m_i,新的任务提交到该队列后,内存以及 CPU 核数满足队列限制条件时,作业需要等待的时间为 w_i / q_i。对于每个任务中的 CPU 使用时间、内存需求量等参数,使用修正系数来进行调整,修正系数记为 ε_i (该系数是通过对不同类型的作业进行测试并进行归一化的结果)。

调度器收到各个队列发出的 hold 消息后,选择等待时间最短的队列作为目标队列。则最终任务的最小等待时间可以通过如下公式计算:

$$\min\{\varepsilon_i \bullet w_i / q_i\} \tag{5}$$

通过上述方式,可以获取候选计算节点中最小的等待的时间。每次分布式调度器对各个作业的任务进行调度时,通过对候选计算节点计算任务的最小等待时间,以此为依据进行调度。

3.3.3 调度性能分析

与随机调度相比,两阶段调度通过选择等待时间最短的计算节点,提高了性能,但是相对于理想调度(即根据整个集群资源最新状态,选择一个空闲的CPU核,将任务提交到该资源,

任务就可以立即执行,不需要等待计算资源)相比,其执行性能却差的较多。直观地讲,对每个任务进行一次两阶段调度的问题在于,作业的响应时间取决于作业中任何一个任务的最长等待时间,这使得平均作业响应时间比平均任务响应时间长得多,特别是某些任务出现长时间等待的情况,即尾任务出现滞后的场合。本文模拟了由 10 000 个 4 核计算节点组成的集群中,每个任务使用两阶段调度和随机分配,网络通信代价为 1 ms。如果作业按照均匀分布到达,并且每个作业包含 100 个任务。作业中任务的运行时间在指定范围(平均 100 ms)内随机产生。在整个作业中,响应时间随着负载的增加而增加,这是因为调度器串行为作业中的每个任务找到空闲计算资源的概率降低的结果。

3.4 两阶段多路调度

两阶段调度过程中,为作业中的每个任务串行提供调度对性能有影响。例如,对于任务 1,任意选择两个队列,可能会出现两个队列都忙的情况,但是任务 1 必须从两者之间选择一个,所以无论怎么选择,任务都必须等待计算资源。对于任务 2,任意选择两个队列后,可能两个队列都处于空闲状态,但是任务 2 只能选择其中一个,这就造成另外一个计算资源的饥饿。如果两个作业同时选择队列,调度器可以根据返回的结构,将任务 1 和任务 2 都分散到任务 2 选择的空闲队列上,那么两个任务都能够获得空闲资源,减少了等待,性能就会得到提高。为了解决该问题,DHRM 使用两阶段多路调度。使用两阶段多路调度,调度器采用批量方式,一次对调度器上可以并发调度的所有任务进行一次资源分配。

假如分布式调度器存在 n 个作业,即 $J=\{J_1,J_2,....,J_n\}$,每个作业中时刻 s 可以调度的任务数量为 $k_1,k_2,.....,k_n$ 个,即 $J_{i,t}=\{t_1^i,t_2^i,.....t_{k_i}^i\}$,对于不同的作业,$k_i$ 的大小也不同。调度器计算出全部可调度任务

$$k=\sum_{i=1}^{n}k_i \tag{6}$$

假如每个任务可以选择两个队列,即 $d=2$,如果系统中所有队列的个数 $q\le 2\times k$,那么调度器向全部的队列发送任务的 hold 请求。反之,调度器选择 $2\times k$ 个队列,向这些队列发送 hold 请求。当所有 hold 请求的返回消息全部到达调度器后,调度器按照作业的截止时间为基准,采用贪心算法为作业的每个任务分配执行队列。假设每个作业的提交时间为 dt,作业中每个任务在队列中等待时间为 w,依据 dt 按照递增顺序排序,对于 w 也按照递增顺序排序,贪心调度(Greedy Schedulering , GS)算法如下所示。

算法 **1** 贪心调度(GS)。

输入:$Q=\{q_1,q_2,...,q_n\}$, $n=2\times k$,

$J=\{J_1,J_2,...,J_n\}$

输出:任务 J 到 Q 上的一个映射

```
1) sort(Q) , sort(J) //递增排序
2) for( J_j in J )
3)  for( t_i in J_i )
4)     for( q_k in Q )
5)      if( q_k.m ≥ t_i.m ){
6)        Q = Q \ q_k
7)       J_i = J_i \ t_i
8)     }
9)   }
10)}
```

GS 算法中的第 1）行,分别按照作业提交时间为作业排序,按照执行队列上的等待时间为执行队列排序。对于作业中的每个任务,从队列中选择满足条件的执行队列,将任务分配给执行队列,如 GS 算法中的第 3）~9）行。GS 算法为每个作业分配完成后,如果还存在有未分配的任务,则继续调用两阶段多路分配算法,为剩余作业中的任务继续分配资源。

GS 算法的目的在于尽量降低每个作业的完成时间,所以采用贪心算法保证每个作业的滞后任务尽量早一点分配到内存,能够尽快完成作业的执行。但是,算法执行过程中,可能每次都有任务不能够获得资源,算法将这些任务与新的任务一起,在下一次两阶段多路调度过程中为这些任务重新分配执行队列。

3.5 负载平衡的处理

使用两阶段多路调度策略后,虽然在一定程度上提高了性能,但是还存在一个问题。例如,假如随机选择的执行队列都是负载较重的队列,而一部分没有选择的执行队列却处于空闲状态,造成负载的不平衡。可以观察到,从一个任务被分散到某个队列开始,在任务实际开始执行的这个期间,整个集群节点上会出现一个更好的执行队列。原因可能是预计开始时间的估计错误、资源的竞争或者计算节点失效等情况时,造成任务选择的执行队列不是最优的。因此需要考虑一些策略来减少这种情况的发生,例如动态负载平衡技术。在 DHRM 中,通过采用负载平衡技术来实现任务在不同队列上的计算资源的平衡。

为实现负载平衡,需要指定一个计算节点为协调器,负载较轻的计算节点向协调器发送空闲队列的资源状态,负载较重的计算节点的队列向协调器发送任务的资源请求。协调器收到资源状态和资源请求后,通过匹配方式,为负载较重的计算节点上的任务匹配到需要的资源,实现负载在计算节点之间的平衡。其实现过程图 2 所示。图 2 中包含资源协调器(Negotiator)、计算节点 1(machine1)和计算节点 2 (machine2)三个组成部分。计算节点 1 用来检测需要转送的任务,计算节

点 2 用来确定负载较轻的队列,协调器用来将计算节点 1 上的任务匹配到计算节点 2 上。

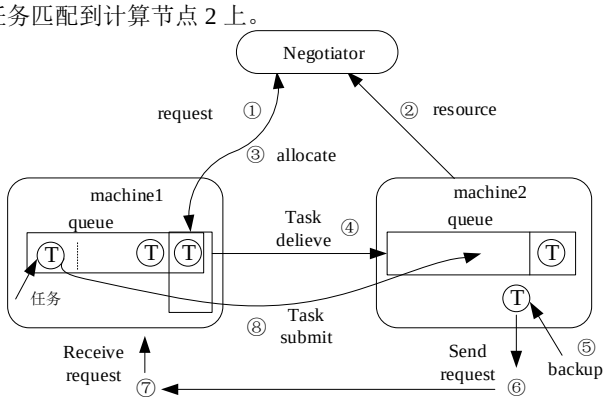

图 2 负载平衡过程

Fig. 2 Process of Load balancing

在正常情况下,任务在各自的队列中等待计算资源,一旦获得计算资源就开始运行任务。当某个节点的队列中任务的等待时间超过某个阈值 hw,计算节点就为队列中的任务启动负载平衡分散过程。另外,当某个计算节点的队列等待时间低于阈值 lw 时,计算节点向协调器发送资源状态信息,协调器根据资源状态和资源请求,从负载高的节点上转移部分任务到负载低的节点上,实现计算节点的负载平衡。负载平衡主要包含三个阶段:第一阶段,任务向资源协调器提交分散请求;第二阶段,协调器经过调度而得到最终的目标节点,然后协调器将目标节点信息返回给原节点;第三阶段,原节点和目标节点进行确认后,将任务分散到目标节点。如图 2 所示,计算节点 1 的一个任务发出分散请求(图 2 中的过程①)。协调器接收到任务的请求后,将其放入队列中,一旦计算节点 2 上的负载较轻,即等待执行的任务低于阈值,计算节点 2 向协调器发送资源状态(图 2 中的过程②),协调器根据最新的队列等待时间矩阵,选择一个最佳的目标节点,例如本例子中的计算节点 2,然后将分配结果返回给计算节点 1(图 2 中的过程③)。计算节点 1 收到消息后,开始向计算节点 2 发送确认信息(图 2 中的过程④),计算节点 2 收到信息后,根据本节点的状态对任务请求进行检查,如资源检查、用户权限检查、队列负载状态检查等,检查结果通常会有如下两种情况:

1)如果请求不被目标节点许可的原因不是由于节点忙,那么该请求就会被终止。

2)如果请求不被目标节点许可的原因是由于该节点忙,比如当前状态下队列上已经被运行着的任务占满了,并且队列上存在一些排队的任务。那么该请求的标识符将被保存到计算节点 2 上(图 2 中的过程⑤),请求在计算节点 1 上处于等待状态。

当计算节点 2 可以执行该任务,而要求转送保存在该节点上的备份任务时,就会发送转送请求给计算节点 1(图 2 中的过程⑥)。此时,计算节点 1 就会将该任务由等待状态变成排队状态,并设置较高的优先级别,同时运送任务请求的内容到目标节点(图 2 中的过程⑥和⑦)。

任务请求转送完成,计算节点 1 正式向计算节点 2 提交任务(图 2 中的过程⑧)。此时新的任务的加入,如果导致计算节点的等待时间超过阈值,计算节点向协调器发送取消资源状态消息,不再接受负载平衡任务。

4 调度策略以及约束

在调度策略方面,DHRM 只支持两种调度策略。本节介绍对两种流行的调度程序策略的支持:第一,任务执行时的数据本地化访问,DHRM 在进行任务调度的时候,为了提高任务的响应速度,通常需要尽量保证执行任务的机器就是数据所在机器。调度器可以对两阶段多路调度的贪心调度算法添加本地化约束,从而最大程度地保证数据本地化;第二,不同用户资源共享时的隔离问题。DHRM 旨在实现不同类型的作业共享同一套集群资源,因此,在集群中大部分的时间内,会有不同类型的作业以及任务运行在集群中,DHRM 的调度器需要保证作业之间运行的时候不能互相干扰,这点调度器通过为每个作业启动若干个容器,作业的任务都会在容器中运行,从逻辑上隔离开了资源的使用,确保作业之间不会互相影响。

DHRM 支持作业级别以及任务级别的约束。这些约束在大数据处理中经常出现,当任务运行时,数据的访问位置与数据计算位置最好在同一个计算节点上。输入数据与任务计算在同一个计算节点时,因为不需要通过网络传输输入数据,通常会减少任务的响应时间。对于每个任务的本地化访问约束,每个任务都可能具有一组计算节点,任务按照本地化要求进行计算,DHRM 无法使用两阶段多路调度来汇总每个作业中所有任务的本地化信息。相反,DHRM 使用两阶段调度时,可以从该任务的本地化约束中找到待选的目标计算节点,这样通过限制每个任务发送 hold 消息的目标计算节点数量,使得每个任务在执行时才会为其选择执行的计算节点,从而实现任务与数据的后期绑定。虽然 DHRM 无法对作业中的每个任务实现数据本地化约束,但是在贪心算法中添加本地化约束策略,可以最大限度地支持任务的数据本地化约束。为了在贪心算法中添加本地化约束,可以将 GS 算法第 5）行的条件判断略作丰富,即除了判断队列资源是否满足任务需求的同时,需要判断该队列所在计算节点是否存在于任务本地化访问约束集合中。若存在,则可以将其添加到任务到队列的映射中;若不存在,可以使用原有的方法为其找到一个合适的队列。

当总的资源需求超过集群计算节点的容量时,集群调度程序将根据特定策略分配资源。DHRM 支持严格优先级资源分配。许多集群共享策略简化为使用严格的优先级,DHRM 通过在工作节点上维护多个队列来支持所有此类策略,依据任务的优先级别和队列的优先级别,实现作业的优先调度,减少作业中尾任务的滞留。从常用的 FIFO、最早截止日期优先和最短作业优先转变到为每个作业分配优先级,然后优先运行优先级最高的作业。例如,以最早的截止时间为最高优先级别

时,将截止时间较早的工作分配给更高的优先级。集群资源管理系统也可能希望直接分配优先级,例如,将生产作业设置高优先级别,批处理作业设置较低的优先级别。为支持这些策略,DHRM 为计算节点的每个队列维护一个优先级别。当资源变为空闲时,DHRM 从优先级最高的非空队列中取出任务并执行。当高优先级任务到达那些正在运行低优先级任务的计算机时,DHRM 也支持抢占。 DHRM 的设计思想是提高每个作业的完成时间,不出现某个作业中一个或者几个任务的严重滞后的任务,因此 DHRM 暂时不支持计算资源在各个作业之间的公平访问,这也是将来探索的问题。

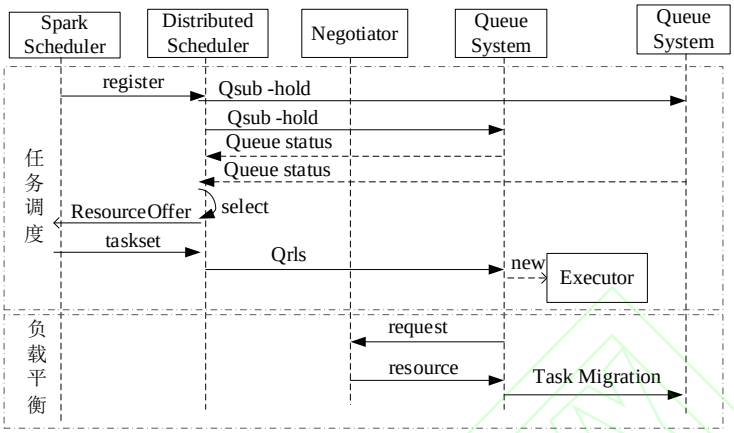

图 3 DHRM 顺序图

Fig. 3 Sequence diagram of DHRM

5 系统实现

本文中的 DHRM 采用了网络队列系统(Network Queue System,NQS)[27-28]和 Mesos 资源调度框架。NQS 在各个计算节点安装,本文为 NQS 添加了一个 ResourceTracker 服务,用来获得各个执行队列中任务的最新状态。“Qsub-hold”命令作为第一阶段的消息,将原来 NQS 中返回结果进行了变更,即由单纯的请求编号改变为编号和任务等待时间两个部分;分布式调度器由 mesos 改造而来,借鉴了 mesos 中“推送”的资源管理模式,作业注册到 mesos 调度框架后,mesos 根据作业中的任务描述,向各个集群节点请求资源;获得资源后,将资源打包成 ResourceOffer,然后推送给 Spark 计算框架,由 Spark 的调度器经过决策后,选择需要的资源,启动执行任务需要的 Executor。最后由 Executor 完成任务的执行。

主要的改进有:1)对于作业框架,重新封装了 Spark 的调度器,以便适应于细粒度的任务调度模式。2)资源管理方面,改进了 Mesos 集群资源管理框架,计算资源的获得不再通过主导资源分配公平(Dominant Resource Fairness,DRF)[29]方式分配。3)在计算节点改进 NQS 系统后,DHRM 支持两阶段多路调度中执行队列的等待时间的计算以及负载平衡时的空闲队列的资源状态的收集。

图 3 给出了 DHRM 任务调度以及负载平衡的过程。任务调度是一个循环过程;负载平衡是动态触发的循环的过程,通常是在集群负载较高的时候才会触发。任务调度中,一旦计算框架的调度器(Spark Scheduler)向集群资源管理系统的一个调度器实例(Distributed Scheduler)注册,Distributed Scheduler 就选择 2 倍的任务数量的候选队列,启动 qsub-hold 命令,等待全部返回执行队列状态(queue status)后,调度器从返回的结果中,选择最优的执行队列,然后向 Spark Scheduler 发送资源邀约(Resource Offer)命令。当 Distributed Scheduler 收到 Spark Scheduler 的返回的任务集合(taskset)后,使用 qrls 命令启动不同计算节点上的执行器(Executor),最后 Executor 从计算框架获得任务并运行。负载平衡中,当一个队列系统中的任务等待时间超过阈值,就向协调器(Negotiator)发送资源请求(request),协调器选择负载较轻的目标队列返回给资源请求者(resource),然后资源请求者向目标队列转送任务(Task Migration)。

6 性能评价

系统在一个集群上进行实验,集群中包含 16 台服务器,其中 15 台服务器(浪潮 NF5468M5 服务器)作为计算节点,1 台中科曙光服务器 620/420,作为协调器。每个服务器节点包含 2 颗 Xeon2.1 处理器,每个处理器包含 8 个核,32 GB DDR4 内存。集群上包含 1 台 AS2150G2 磁盘阵列。服务器操作系统为 Ubuntu 7.5.0,采用 C++11 作为编程语言,Mesos 的基础版本为 1.8,Spark 的基础版本为 2.4.3。这些计算机被组织在 4 个机架内,每个机架包含 4 台计算机。机架内的计算机通过机架连接,各个机架交换机通过级联方式与汇聚交换机连接。

测试时使用了 5 个分布式调度器,首先,本文使用 DHRM 为 TPC-H 工作负载调度任务,其负载具有分析查询功能。本文首先在理想的调度程序的基础上比较作业的响应时间。本文提供了 DHRM 细粒度资源共享时的开销并量化了其性能。

同时,为了比较本文中所述任务调度算法与已有的调度算法的优劣,于是在同等条件下测试了 sparrow 的任务调度算法包括批采样以及每任务采样等技术所造成的延迟,并与 DHRM 进行了比较,分析两者之间的差距。其次,本文展示了 DHRM 在调度延迟以及等待时间方面的对比。最后,为了体现大规模集群环境下,大量任务的调度延迟,本文设计了一个基于 Spark 调度器的模拟程序和一个模拟的大规模集群环境,通过模拟程序在模拟集群环境的调度,分析了调度延迟的特点。

6.1 TPC-H 负载的性能对比

本文使用运行在 Spark 的 TPC-H(www.tpc.org/tpch/) 查询基准来测试 DHRM 的调度性能。TPC-H 基准代表对事务数据的查询,这是低延迟数据并行框架的常见用例。每个 TPC-H 查询在 Spark 下运行,Spark 将查询转换成 DAG,按照不同的阶段来调度执行。查询的响应时间是各个阶段响应时间之和。5 个不同的用户启动多个 TPC-H 查询负载,查询负载随机排列,并且在大约 15 分钟的时间内维持集群负载在 80%左右。本文针对实验中 200 s 的数据进行了分析,在 200 s 中,DHRM 调度了超过 4000 个作业,这些作业构成了 1200 个 TPC-H 查询。每个用户都在 TPC-H 数据集的副本上运行查询,数据集的大小大约为 2 GB,并分为 30 个分区,每个分区的副本数设置为 3。

为了比较 DHRM 的性能,本文假设任务在计算节点上没有等待时间,就像粗粒度资源分配方式一样,这个响应时间称为理想时间。在计算某个查询的理想响应时间时,本文分析了一次查询的 DAG 过程中的每个阶段,去掉队列等待时间后,将其它阶段的时间相加,最后得到查询的理想响应时间。由于 DHRM 在计算理想响应时间时,采用了任务的执行时间,所以理想响应时间也包括每个任务的本地化约束。理想的响应时间不包括将任务发送到计算节点所需的时间,也不包括在利用率突发期间不可避免的排队等待时间,所以,理想响应时间是调度器可以实现的响应时间的下限。

图 4 给出的数据可以看出,与随机调度、两阶段调度相比,DHRM 的两阶段多路调度最优,基本上不超过理想调度时间的 13%。与随机调度相比,两阶段多路调度只是随机调度平均响应时间的 25%-33%,95%分位数的响应时间最多只是随机调度时间的 20%。与两阶段调度相比,两阶段多路调度的响应时间是两阶段调度响应时间的 80%,95%分位数的响应时间也只达到两阶段调度的 60%。与两阶段多路调度相比,负载平衡策略的使用,平均响应时间减少了 14%。DHRM 还提供了良好的绝对性能:其中值响应时间仅比理想调度程序所提供的响应时间高 12%。

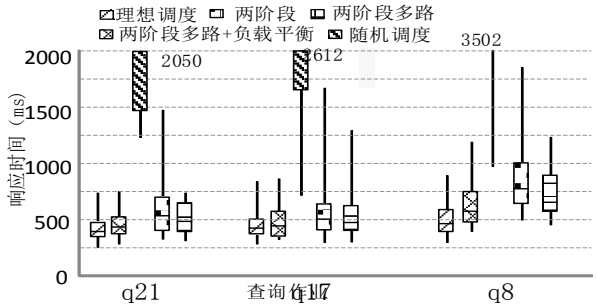

图 4 DHRM 不同调度算法的响应时间对比

Fig. 4 Comparison of response time of different scheduling algorithms in DHRM

除了在 DHRM 内测试不同调度算法的性能,本文还比较了 DHRM 与 sparrow 的批采样调度算法以及每任务采样调度算法。结果如图 5 所示。从图 5 中的结果可以看出,DHRM 两阶段多路调度算法加负载均衡技术和 sparrow 的批采样算法最优,其中 sparrow 的批采样调度算法中值响应时间仅比理想的影响时间高出 13%左右,和 DHRM 的两阶段多路调度算法性能大致相当,这表明本文的两阶段多路调度算法具有较为良好的性能。但是两阶段多路调度若是不使用负载平衡技术,则性能要低于 sparrow 的批采样技术。虽然 sparrow 的批采样技术性能较佳,但是 sparrow 的每任务采样技术性能比其他调度技术要差,其比理想调度延迟高出大约 40%。两阶段多路技术即使不使用负载平衡技术在性能上也略高于每任务采样技术。从图 5 的结果可以看出,DHRM 的调度技术在性能上不输于 sparrow 调度框架,而 sparrow 也是应用于大规模低延迟任务的调度框架。

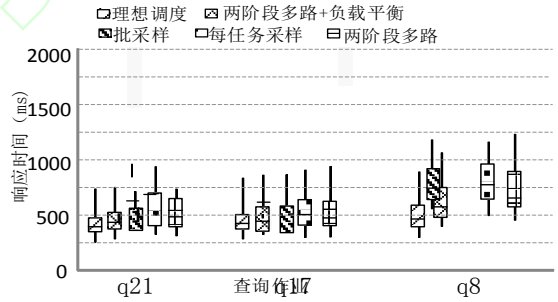

图 5 DHRM 和 sparrow 不同调度算法的对比

Fig. 5 Comparison of different scheduling algorithms between DHRM and sparrow

6.2 作业响应时间对比

为了理解 DHRM 相对于理想调度程序增加的延迟的组成部分,本文将作业的响应时间分解为调度时间、队列中等待时间以及任务执行时间 3 个单独的时间。图 6 中给出了不同

时间中任务的累计概率函数图。图中的每一根曲线代表一个任务数量与时间的变化曲线。

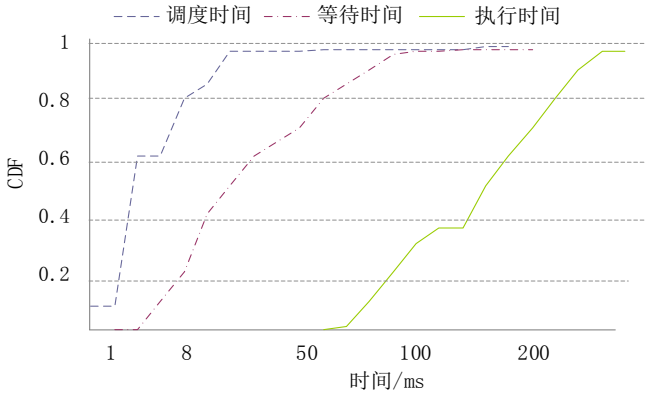

图 6 任务延迟统计

Fig. 6 Chart of task delay statistics

从图 6 的曲线看出,60%的任务其调度延迟时间为 5 ms 左右,40%的作业其调度延迟为 12 ms;而任务在队列中的等待时间较长,75%的任务在队列中的等待时间为 50 ms,剩余 25%的任务的等待时间大约为 100 ms。任务的执行时间在 350 ms 以内,其中 40%的任务执行时间在 150 ms 内,超过 250 ms 的任务只占有 15%左右。从图 6 可以看出,分布式集群调度框架确实能够减少调度延迟。

6.3 大规模集群的性能模拟

为了验证 DHRM 的资源调度框架在大规模集群环境下的调度延迟,本文设计了一套资源调度框架性能模拟器工具,DHRM_Simulator,用于对 DHRM 的资源调度功能进行测试。它可以在多台机器上模拟大规模集群,并根据历史日志分析提取出应用程序负载信息,模拟完整的资源分配、任务调度和资源回收过程。通过使用 8 台服务器来模拟大规模的集群环境。实验过程中每台机器均为 NF5468M5 服务器,包含 2 颗 Xeon2.1 处理器,每个处理器包含 8 个核,32GB DDR4 内存。

6.3.1 模拟机器数量的设置

为了测试大规模集群计算资源的调度性能,使用一台机器模拟协调器,其余 7 台机器模拟计算节点。每台工作节点机器设置 50 个队列,每个队列模拟一台物理机器,称为逻辑机器。队列的同时运行的槽数(slot)设置为 8,代表一个逻辑机器上包含 8 个模拟的 CPU 核。这样的话,一台物理机器就可以模拟 50 台逻辑机器,7 台物理机器模拟 350 台逻辑机器,每台模拟的逻辑机器上模拟 8 个 CPU 核,那么整个模拟系统可以使用的 CPU 计算资源为 2800 个模拟的 CPU 核。

6.3.2 作业的定义

采用类似 Spark 调度器,本文实现了一个模拟作业框架。每个作业框架上包含数量不等的任务。任务的执行时间由 sleep 代替。作业提交后,按照 task 任务数量,随机选择 2 倍的 task 任务数量的队列(一个队列代表一个模拟机器),获得队列中的剩余的可以使用的 slot 数,选择剩余 slot 最大的队列作为选择的资源,使用 qsub 启动一个 Spark Executor 类似的进程,进程内的作业执行类似 sleep 执行,不设置具体的数据执行任务。任务的执行时间由 sleep 长短来决定,即等待时间长短,如 200,代表任务执行时间是 200 ms。

向模拟器连续启动 7×20 个作业(即 7 个分布式调度器实例),即每个分布式调度器实例启动 20 个作业,每个作业按照 160 个任务设置。作业按照一定的时间间隔向各自的分布式调度器注册并请求资源执行。集中式调度器的测试则是将 7×20 个作业同时向该集中式调度器注册启动。

6.3.3 响应时间

为了测试分布式调度框架的低延迟,使用集中方式与分布方式进行对比。集中式调度器采用类似 mesos 的方式实现。测试中,分别设置任务的 sleep 时间为 100 ms,200 ms,1000 ms 三种不同的任务执行时间,用来验证作业响应时间与调度器的调度延迟之间的关系。图 7 给出了三种情况下的模拟测试结果。

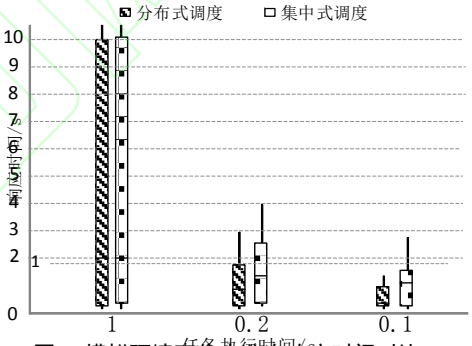

图 7 模拟环境下集中分布响应时间对比

Fig. 7 Comparison of the response time of centraliazed and distributed in simulated environment

从图7的结果看,任务运行时间在1000 ms时,分布式调度和集中调度方式下,作业的响应时间基本相同。但是当任务的执行时间是 200 ms 时,分布式调度的平均响应时间是集中式调度方式下的一半左右。而任务执行更短,即 100 ms 时,分布式调度器和集中式调度器的差别就更大,几乎是 10 倍左右的差别。因为,分布式调度器能够减少调度的延迟时间,为大规模并行任务的执行提高了效率。因此,相较于 mesos 等集中式资源调度器,DHRM 的分布式调度器能够对大批量的短任务有更好的调度效率,使大批量的短任务具有较低的响应时间。

7 结语

大规模运行时间较短的任务越来越多,呈现出并行度越来越高的趋势,这种作业的调度普遍受到重视。数据中心上运行的这类任务对延迟非常敏感。另外,随着集群计算节点规模的扩大,调度延迟是影响作业吞吐量和性能的主要瓶颈。但是,传统的集群资源调度框架在低延迟方面存在一定的缺陷,本

文通过两阶段多路调度以及负载平衡技术,解决了现有调度框架中延迟较高和负载不均衡的问题,通过 TPC-H 基准测试以及大规模集群下的模拟测试,调度框架能够保证短时间任务的低延迟要求。但是,本文算法在数据本地化访问以及资源分配的公平性方面有待进一步的提高。

参考文献 (References)

[1] CARBONE P, KATSIFODIMOS A, EWEN S, et al. Apache flink: stream and batch processing in a single engine[J]. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2015, 36(4).

[2] AKIDAU T, BRADSHAW R, CHAMBERS C, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing[J]. Proceedings of the Vldb Endowment, 2015, 8(12): 1792-1803.

[3] CHANDRASEKARAN S, COOPER O, DESHPANDE A, et al. TelegraphCQ: continuous dataflow processing[C]//Proceedings of the 2003 ACM SIGMOD international conference on Management of data. New York: ACM, 2003: 668-668.

[4] XIN R S, GONZALEZ J E, FRANKLIN M J, et al. Graphx: A resilient distributed graph system on spark[C]// Proceedings of the First international workshop on graph data management experiences and systems. New York: ACM, 2013: 1-6.

[5] 郝春亮, 沈捷, 张珩,等. 大数据背景下集群调度结构与研究进展[J]. 计算机研究与发展, 2018, 55(1):53-70.(HAO CHUNLIANG, SHEN JIE, ZHANG HENG, et al. Cluster scheduling structure and research progress under the background of big data [J]. Computer research and development, 2018, 55 (1): 53-70).

[6] MELNIK S, GUBAREV A, LONG J J, et al. Dremel: Interactive Analysis of Web-Scale Datasets[J]. Communications of the Acm, 2010, 3(6):114-123.

[7] EAGER D L, LAZOWSKA E D, ZAHORJAN J. Adaptive load sharing in homogeneous distributed systems[J]. IEEE transactions on software engineering, 1986 (5): 662-675.

[8] ZAHARIA M, CHOWDHURY M, DAS T, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing[C]//Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. Berkeley, CA :USENIX Association, 2012:2.

[9] CHAMBERS C, RANIWALA A, PERRY F, et al. FlumeJava: easy, efficient data-parallel pipelines[J]. ACM Sigplan Notices, 2010, 45(6): 363-375.

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

[11] HINDMAN B, KONWINSKI A, ZAHARIA M, et al. Mesos: A platform for fine-grained resource sharing in the data center[C] // Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation. Berkeley, CA: USENIX Association, 2011: 295-308.

[12] VAVILAPALLI V K, MURTHY A C, DOUGLAS C, et al. Apache hadoop yarn: Yet another resource negotiator[C]//Proceedings of the 4th annual Symposium on Cloud Computing. New York: ACM ,2013: 1-16.

[13] YOO A B, JETTE M A, GRONDONA M. Slurm: Simple linux utility for resource management[C]// Proceedings of the 2003 Workshop on Job Scheduling Strategies for Parallel Processing. Cham: Springer, 2003: 44-60.

[14] KARANASOS K, RAO S, CURINO C, et al. Mercury: Hybrid centralized and distributed scheduling in large shared clusters[C]// Proceedings of 2015 USENIX Annual Technical Conference. Berkeley, CA: USENIX Association, 2015: 485-497.

[15] 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. New York: ACM, 2013: 351-364.

[16] OUSTERHOUT K, WENDELL P, ZAHARIA M, et al. Sparrow: distributed, low latency scheduling[C]//Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. New York: ACM, 2013: 69-84.

[17] 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. New York: ACM,2010: 265-278.

[18] ISARD M, PRABHAKARAN V, CURREY J, et al. Quincy: fair scheduling for distributed computing clusters[C]//Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. New York: ACM, 2009: 261-276.

[19] GODER A, SPIRIDONOV A, WANG Y. Bistro: Scheduling data-parallel jobs against live production systems[C]// Proceedings of the 2015 USENIX Conference on USENIX Annual Technical Conference. Berkeley, CA: USENIX Association, 2015: 459-471.

[20] BOUTIN E, EKANAYAKE J, LIN W, et al. Apollo: Scalable and coordinated scheduling for cloud-scale computing[C]// Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2014: 285-300.

[21] DELIMITROU 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. New York: ACM, 2015: 97-110.

[22] DELGADO P, DINU F, KERMARREC A M, et al. Hawk: Hybrid datacenter scheduling[C]// Proceedings of the 2015 USENIX Annual Technical Conference. Berkeley, CA: USENIX Association, 2015: 499-510.

[23] DELGADO P, DIDONA D, DINU F, et al. Job-aware scheduling in eagle: Divide and stick to your probes[C]//Proceedings of the 7th ACM Symposium on Cloud Computing. New York: ACM, 2016: 497-509.

[24] RASLEY J, KARANASOS K, KANDULA S, et al. Efficient queue management for cluster scheduling[C]//Proceedings of the Eleventh European Conference on Computer Systems. New York: ACM, 2016: 1-15.

[25] LITZKOW M J, LIVNY M, MUTKA M W. Condor-a hunter of idle workstations[C]// Proceedings of the 8th International Conference on Distributed Computing Systems. San Jose, CA: IEEE, 2012:104-111.

[26] WANG K, LIU N, SADOOGHI I, et al. Overcoming hadoop scaling limitations through distributed task execution[C]// Proceedings of the 2015 IEEE International Conference on Cluster Computing. Piscataway: IEEE, 2015: 236-245.

[27] 汤小春. 基于集群技术的作业管理系统的研究与实现[D]. 西安:西北工业大学, 2001: 127. (TANG XIAOCHUN. The research and implementation of job management system based on cluster computing technology[D]. Xi'an: Northwestern Polytechnical University,2001:127).

[28] BRENT A. KINGSBURY. KINGSBURY B K. The network queueing system[EB/OL].[2019-10.02]. http://gnqs.sourceforge.net/docs/papers/mnqs_papers/original_cosmic_nqs_paper.htm.

[29] GHODSI A, ZAHARIA M, HINDMAN B, et al. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types [C]// Proceedings of the 8th USENIX conference on Networked systems design and implementation. Berkeley, CA: USENIX Association,2011: 323–336.

This work is partially supported by National Key Research and Development Plan (2018YFB1003400)

ZHAO Quan, born in 1997, master candidate. His research interests include big data compute and cluster resource management.

TANG Xiaochun, born in 1969, Ph.D., associate professor, His research interests include graph base management, distributed computing, and cluster resource management.

ZHU Ziyu, born in 1996, master candidate. Her research interests include big data compute and cluster resource management.

MAO Anqi, born in 1996, master candidate. Her research interests include big data compute and cluster resource management.

LI Zhanhuai, born in 1961, Ph.D., professor, a member of Chinese Computer Federation. His research interests include ocean base management and big data computing.