第57卷 第1期
2021年1月

南京大学学报(自然科学)
JOURNAL OF NANJING UNIVERSITY
(NATURAL SCIENCE)

Vol. 57, No. 1
Jan., 2021

DOI:10.13232/j.cnki.jnju.2021.01.012

异构Flink集群中负载均衡算法研究与实现

汪志峰[1],赵宇海[1*],王国仁[2]

(1.东北大学计算机科学与工程学院,沈阳,110169;2.北京理工大学计算机学院,北京,100081)

摘 要:Flink是目前非常流行的流处理引擎.和先前的Hadoop,Spark,Storm等分布式计算框架相比,Flink能实现低延迟、高吞吐,保证Exactly Once.调度模块是保证集群高性能非常重要的一部分,但目前Flink调度默认把集群中所有节点看作是同等性能的,采用轮询调度策略.但在异构集群里这样的调度就会低效,因为计算资源少的节点运行的Task和计算资源多的节点运行的Task一样多,所以局部负载不均衡,影响Job的运行时间和吞吐量,造成延时.提出平滑加权轮询任务调度算法和基于蚁群算法的任务调度算法,解决运行过程中集群负载不均衡问题.平滑加权轮询任务调度算法在任务调度初始阶段根据集群资源按照权重平滑轮询调度.基于蚁群算法的任务调度算法是在运行过程中当集群已使用资源高于阈值时采用类似蚁群算法去执行任务调度,动态计算全局最优任务分配方案,能重新负载均衡.

关键词:Apache Flink,任务调度,负载均衡,平滑加权轮询任务调度,基于蚁群算法的任务调度

中图分类号:TP391 **文献标识码**:A

Research and implementation of load balancing algorithm in heterogeneous Flink cluster

Wang Zhifeng[1*], Zhao Yuhai[1*], Wang Guoren[2]

(1. School of Computer Science and Engineering, Northeastern University, Shenyang, 110169, China;
2. School of Computer Science and Technology, Beijing Institute of Technology, Beijing, 100081, China)

Abstract: Flink is currently a very popular stream processing engine. Compared to previous distributed computing frameworks such as Hadoop, Spark and Storm, Flink can achieve low latency, high throughput, and exactly-once. The scheduling module is a very important part of ensuring the performance of the cluster. However, at present, Flink scheduling regards all nodes in the cluster as equivalent performance by default, and adopts a polling scheduling policy. In a heterogeneous cluster, such scheduling is inefficient, because a node with fewer computing resources runs as many tasks as nodes with more computing resources, causing local load imbalance, affecting the running time, throughput, and delay of the job. Time. This paper proposes a smooth weighted round-robin task scheduling algorithm and a task scheduling algorithm based on ant colony algorithm to solve the problem of cluster load imbalance during operation. The smooth weighted round-robin task scheduling algorithm smoothes the polling scheduling according to the weight of the cluster resources in the initial stage of the task scheduling. The ant colony algorithm-based task scheduling algorithm uses a similar ant colony algorithm to perform task scheduling when the cluster has used resources above the threshold, and dynamically calculates a global optimal task allocation scheme. This scheme can reload the load.

Key words: Apache Flink, task scheduling, load balancing, smooth weighted round-robin task scheduling, task scheduling based on ant colony algorithm

基金项目:科技部重点研发项目"云计算和大数据"重点专项(2018YFB1004402)

收稿日期:2019-12-29

*通讯联系人,E-mail:zhaoyuhai@mail.neu.edu.cn

近年来,由于数据量呈指数式增长,处理大数据[1]的方式也发生了很大变化,迄今已经历三代引擎的改进:第一代以Hadoop[2]为代表,利用MapReduce[3]进行大数据处理;第二代以Spark[4]为代表,是基于RDD(基于内存计算)的微批流处理框架;第三代是以Flink为代表的面向流,保证Exactly Once的实时流处理框架.Flink[5]的计算平台可以实现毫秒级延迟下每秒处理上亿次的消息或者事件;同时,Flink还提供一种Exactly-once[6]的一致性语义,保证数据的正确性,使Flink大数据引擎可以提供金融级的数据处理能力.高吞吐、低延迟的性能使Flink成为目前流处理的首选.与此同时,各个公司处理大数据的方式也发生了很大变化,例如阿里巴巴、滴滴出行、美团都大规模使用Flink集群,阿里巴巴还开源自己的Blink给Flink社区,给Flink带来了更好的性能优化以及方便的SQL环境.如图1所示的流程中,上游是事务处理、日志、点击事件等,经过Flink的流处理到达下游;下游是处理之后的数据存储到数据库里或直接被应用所利用.Flink在其中可以提供低延迟、高吞吐、Exactly Once的处理.

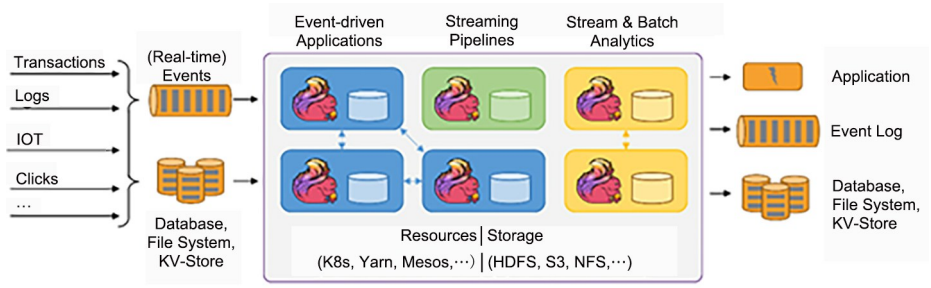

图1　Flink应用场景

Fig. 1　Flink application scenario

任务调度是Flink很重要的功能.Flink通过JobManger任务调度器管理Slot,把任务分配到合适的Slot等待TaskManager执行.Flink的任务调度图如图2所示,Flink集群启动后首先会启动一个JobManger和一个或多个TaskManager,Client提交任务给JobManager,JobManager再调度任务到各个TaskManager去执行.在这个过程中TaskManager把心跳和任务处理信息汇报给JobManager,TaskManager之间以流的形式进行数据传输.在集群运行的过程中,如果流任务失败则利用快照加检查点的形式恢复,如果批任务失败则重新开始.在把任务分配到具体的Slot的过程中,优先选择符合Local属性的节点.如果有SlotSharingGroup限制,则在SlotSharingGroup里再创建一个SimpleSlot;如果有CoLocationGroup限制,则必须在同一个CoLocationGroup里创建一个SimpleSlot;如果没有上述限制,则从Slot集合里挑一个.

所以,Flink的默认调度策略是轮询,每个任务需要去可用的Slot集合顺序选择一个Slot分配

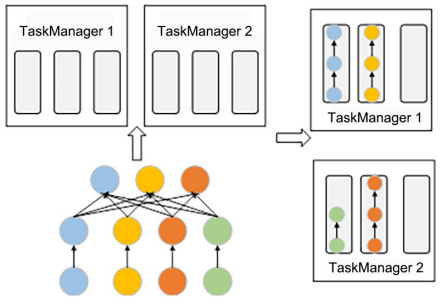

图2　Flink任务调度图

Fig. 2　Flink task scheduling diagram

给这个Task.在调度过程中,属于同一个Task的所有SubTask不能分配在Slot里面,因为任务需要分布式运行,所以不同的子任务必须分配在不同的Slot里.

Flink默认的调度策略把所有节点认作同等性能(这里的性能指CPU的处理能力、内存等),但在实际的集群搭建部署过程中,集群中的节点可能性能相差较大.若在这种异构的集群中采用轮询的调度策略就可能因为没有考虑每个节点的不同负载能力,使性能较差的节点和性能较好的

节点被分配同样的任务. 性能较差的节点负载过多的 Task 会影响整体 Job 的运行效率,使集群的吞吐降低、延时增加,延长整个 Job 的运行时间,因此在异构集群中根据集群各个节点的不同性能调整任务的分配成为提升异构集群整体性能的关键. 基于此,本文提出自适应负载均衡算法,提升系统的吞吐量、降低延时、减少 Job 的运行时间,使集群的整体性能有较大的提升.

本文的主要贡献:

(1)通过对 Hadoop 和 Spark 的研究发现,在异构集群中应用默认的轮询调度策略没有考虑节点的处理性能,导致集群执行效率的下降. 本文通过实验证明 Flink 系统中也会出现这种现象,即异构 Flink 集群中可能存在由于任务分配不均衡导致 Job 完成时间被拖长、吞吐量降低、延时增加.

(2)提出异构集群中平滑加权轮询任务调度算法(Smooth Weighted Polling Task Scheduling, SWPTS). 根据集群中每个节点的性能权重 Ew,按从大到小的次序,依次给每个节点的 Slot 分配 SubTask. 与此同时,记录节点当前的权重 Cw,降低 Cw 最大的节点来避免有效权重大的节点被连续多次选中,从而使集群在开始调度时就能保持负载均衡.

(3)提出基于蚁群算法的任务调度算法(Task Scheduling Algorithm Based on Ant Colony Algorithm, ACTS). 在集群运行过程中,当集群资源的使用高于预设值 ε 时则使用 ACTS 寻找全局最优任务分配方案,使每个 SubTask 被分配到合适的 Slot 上,让整个集群的效率达到最高.

(4)在真实数据集和人工数据集上进行了实验分析和验证,结果表明,在 Flink 集群里应用 SWPTS 和 ACTS 确实对缩短 Job 运行时间、提高吞吐量、降低延时有很好的效果.

1 相关工作

在流计算框架中任务调度是很重要的一个模块,负责把 Task 调度给 Slave 的 Slot 执行,不同的流计算引擎尽管实现方式不同,但实现的功能都是把任务根据某种算法分配给指定的 Slave 执行.

Hadoop 提供可插拔的任务调度器,它根据用户的需求选择合适的调度方案,并可以随时切换. Hadoop 的任务调度算法有三种. (1)FIFO 调度器[7]. FIFO 调度器是最开始被集成到 Hadoop 里的,Task 按 FIFO 的顺序进入大工作队列,JobTracker 从工作队列里取最先到达的 Task. 这种调度策略没有考虑作业的优先级和作业的大小,但这种策略最容易实现,也是有效率的. (2)公平调度器(Fair Scheduler)[8]. 公平调度是一种分配作业资源的策略,目的是让所有的作业随着时间的推移都能平均地获取等同的共享资源,所有 Job 享有相同的资源. (3)计算能力调度器(Capacity Scheduler)[9]支持多个队列,每个队列可配置一定的资源量并采用 FIFO 调度策略. 为防止同一个用户的作业独占队列中的资源,该调度器会对同一用户提交的作业所占的资源量进行限定,优先执行占用最小、优先级高的作业.

Lee and Chung[10]提出一个针对 Hadoop 的截止时间约束调度算法,使用统计学方法来测量数据节点的性能,并基于该信息创建检查点来监视作业的进度;根据每个检查点的作业进度,算法将任务分配给不同的作业队列. Kc and Anyanwu[11]提出基于工作执行代价模型来满足用户规定的数据处理截止日期的调度算法,这些工作执行代价包括 map 和 reduce 两个阶段的运行时间、map 和 reduce 输入数据的规模和分布.

Spark 是当下十分流行的流计算框架. 默认情况下,Spark 调度器按照 FIFO(先进先出)[12]的形式来调度任务,每个工作被分为多个“阶段”(如 map 和 reduce 语句). 对于所有可用的资源,第一个工作的优先级最高,其任务即被启动;之后是第二个,依次类推. 如果集群不需要队列头的工作,后面的工作将被立刻启动;如果队列头的工作很大,则后面的工作可能被大大推迟. Spark 后来的版本模仿 Hadoop 的公平调度器[13]也添加了公平的调度策池,不同的工作可以被分到不同的组,每组对应一个任务池,不同的任务池设置不同的调度选项(权重).

Mao et al[14]提出用机器学习领域的增强学习和神经网络,在无须手动设置最小工作完成时间的情况下得到一种在指定工作负载情况下的调度算法,在这个过程中设计可扩展的 RL 模型,并发

明RL训练方法处理连续到来的随机Job. Ren et al[15]设计了第一个分散感知调度器Hopper,为了提供可拓展性和可预测性,Hopper被设计成一个分散的Job调度器,它把资源分配给Job的同时也能预测拖慢工作进度的子任务,从而采取合理算法降低系统的延时.

Storm里也有很多对Storm的调度器进行改进的工作. Peng et al[16]在2015年提出R-Storm资源感知调度器,通过增加最大化资源利用率来提升总吞吐量,同时最小化网络延迟,在任务调度时R-Storm可以满足软资源和硬资源的限制以及最小化组件之间的网络通信代价,在标准的Yahoo基准测试下吞吐量提高30%~47%,CPU利用率提高69%~350%. Chen and Lee[17]发现Storm默认使用轮询的算法来分配任务,这对异构计算环境不是最佳,于是提出G-Storm调度算法. G-Storm利用集群节点GPU来加速整体性能,实验结果表明,与原始Storm调度算法相比,G-Storm在轻量级和高负载拓扑上可以实现1.65倍至2.04倍的性能提升.

目前针对异构集群的负载均衡算法没有通用的高效算法,大部分还是通过感应集群中某种资源来作出动态调度,或利用集群的其他计算能力(如GPU)来负载均衡,通常存在三个问题:(1)这些算法只从单一的性能指标去实现负载均衡,没有综合多个性能指标来思考;(2)虽然其中有加权轮询的算法,但这种加权在选择节点时没有考虑平滑,导致有些权重大的节点被连续选中,造成短时间内局部负载过重,影响节点效率;(3)这些算法只考虑负载均衡,没有考虑集群负载均衡算法后已使用的资源高于设定的阈值时的处理.

针对以上问题,本文改进和实现如下:(1)考虑CPU利用率、内存利用率和总内存这三个关键性能指标,而非单一指标,并且动态地监控三个性能指标来作出调度决策;(2)不仅考虑多个性能指标的加权,而且在加权过程中进行平滑处理,即避免让权重大的节点连续被选中造成局部短时间内负载过重;(3)在集群已使用的资源高于设定阈值后采用ACTS,能够在一定的迭代次数内得到全局最优任务分配方案,按照这个方案调度任务来重新负载均衡,也能使集群处于最佳状态.

本研究考虑Flink集群CPU利用率和内存利用率等性能指标,在初始调度阶段采用SWPTS,在集群已使用资源高于设定阈值的情况下采用ACTS,经过实验验证,能够在Job的运行时间、延时、吞吐量等性能指标上有明显提升.

2　基本概念

定义1　内存使用率(*Mu*)　这是描述计算机的重要性能指标之一,可描述集群中已使用的内存占用总内存的比重. 定义如下:

$$Mu=1-\frac{memfree}{memtotal} \tag{1}$$

其中,*memfree*表示剩余的内存,*memtotal*表示总内存大小.

定义2　CPU使用率(*Cu*)　它描述CPU当前被占用的情况,使用率越大表示CPU越忙,在这种情况下CPU很难空出时钟周期运行即将提交该节点的任务. 定义如下:

$$Cu=1-\frac{idle}{idle+other} \tag{2}$$

定义3　有效权重(*Ew*)　这是在初始调度时根据CPU使用率、内存使用率等性能指标得到的一个综合权重,可衡量节点的总体资源使用情况. 计算方式如下:

$$Ew=0.9Cu+0.1Mu \tag{3}$$

定义4　当前权重(*Cw*)　它可以用来衡量运行过程中每个节点被选中的权重,每经过一次调度当前权重都会发生变化,可以用来挑选最合适的节点. 定义如下:

$$Cw=Cw_{\text{last}}+Ew \tag{4}$$

其中,Cw_{last}表示上次调度时的有效权重.

定义5　任务执行时间(*timeMatrix*[*i*][*j*])　它表示任务*i*分配给节点*j*所需的时间,在算法执行的初始时刻初始化. 定义如下:

$$timeMatrix[i][j]=\frac{tasks[i]}{nodes[j]} \tag{5}$$

定义6　最大信息素概率(*maxphe*)　它主要用来确定蚂蚁的临界下标(indexbound),即确定一个临界蚂蚁编号,在这个编号之前的蚂蚁选择信息素最大的节点从而把任务分配给该节点,而这个节点之后的蚂蚁选择一个随机的节点,把任

务分配给该节点. 定义如下:

$$maxphe = \frac{maxpheMatrix[i]}{\sum_{i=0}^{maxantCount} pheMatrix[i]} \tag{6}$$

其中,$pheMatrix[i]$是节点 i 的信息素. 临界下标的定义如下:

$$indexbound = maxphe \times antCount \tag{7}$$

3 算法描述

本节详细介绍集群资源监控、SWPTS 和 ACTS. 集群资源监控主要负责监控集群中每个节点的 CPU 使用率、内存使用率,并把这些数据通过 Http 协议[18]发送给 Flink 的 JobManager 作为性能数据来确定权重. 与传统的负载均衡算法不同,SWPTS 和 ACTS 是基于集群资源的动态负载均衡算法.

3.1 集群资源监控 集群资源监控主要负责为调度器提供性能数据,Flink 集群的 JobManager 根据性能数据来确定每个 Slave 的权重,调度器利用权重来实现整体的调度负载均衡. 集群资源监控整体的架构如图 3 所示,每个 slave 节点利用正则表达式解析/proc 目录下的 cpuinfo 和 meminfo 这两个文件得到每个时刻 CPU 的使用率和内存使用率,然后通过 Socket 通信把数据发送给 Master. Master 中的 Redis[19]完成性能数据的持久化处理,同时通过 Web 技术对外提供 Http 接口,Flink 集群可以通过 RPC 调用获取集群中各个节点的实时性能数据.

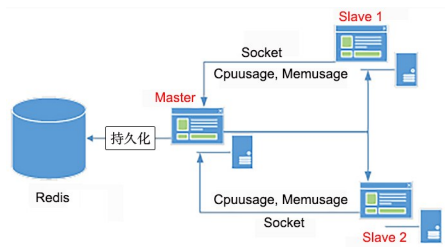

图 3 资源调度架构图

Fig. 3 The diagram of resource scheduling architecture

3.2 SWPTS 总览 初始化 *bestInstance* 和 *totalWeight*. 其中 *bestInstance* 用来保存最后选中的最好的 *DataMetric*,它在每次循环过程中都记录当前节点为止 *Cw* 最大的 *DataMetric*. *totalWeight* 则在每次循环过程中累加每个 *DataMetric* 的 *Ew*,循环结束时,*totalWeight* 保存所有节点 *Ew* 的和.

在每次循环过程中,每个 *DataMetric* 的 *Cw* 都会加上它自身的 *Ew*,并用 *total* 累加当前 *Ew*. 如果当前 *DataMetric* 的 *Cw* 大于 *bestInstance* 的 *Cw*,则记录 *bestInstance* 到目前为止具有最大 *Cw* 的 *DataMetric*.

在循环完成之后,找到当前的最优的 *DataMetric*,它有最大的 *Cw*. 为使该 *DataMetric* 不被重复选中,需要降低它的 *Cw*. 即用 *bestInstance* 的 *Cw* 减去 *total*,并返回这个最优的 *bestInstance*. 具体见算法 1.

算法 1 SWPTS

输入:*DataMetric* 节点集合 *DataMetrics*

输出:对本次调度过程中被选中的节点 *DataMetric*(*i*)

```
1. Init bestInstance=null, totalWeight= 0
2. for every DataMetric in DataMetrics
3.    add DataMetric's Ew to Cw
4.   add Ew to total
5.   if Cw>bestInstance. Cw
6.   bestInstance=DataMetric
7. bestInstance. Cw=bestInstance. Cw-total
8. return bestInstance
```

SWPTS 主要是在 Job 运行的初始阶段把任务分配给 TaskManager 的 Slot. 为减少频繁 Rpc 带来的时间影响,在初始的调度过程中 JobManager 没有一直请求集群资源监控的性能数据,而是在第一次请求后缓存数据,在后续任务到来时 SWPTS 利用初始缓存的性能数据,再用 ACTS 选择一个节点的 Slot 分配给 Task.

以一个有三个 Slave 节点的集群为例,假设 weight(Slave1):weight(Slave2):weight(Slave3)=3:1:2,集群的前六次调度如表 1 所示.

可以看到,Slave1 的权重最大,被调度的次数为 3;Slave3 的权重次之,被调度的次数为 2;Slave2 的权重最小,被调度的次数为 1. 上述调度体现了加权,即权重大的节点被调度选中的次数也多;同时也体现了平滑的特点,权重大的节点没有被连续选中,而是被间断地选中. 综上所述,SWPTS 能使集群调度负载均衡且避免权重较大的节点在局部短时间内负载过重.

表1　Flink任务调度过程

Table 1　Flink task scheduling process

	Slave1(Cw)	Slave2(Cw)	Slave3(Cw)	选中节点(Cw)
1	6	2	4	Slave1
2	3	3	6	Slave3
3	6	4	2	Slave1
4	3	5	4	Slave2
5	6	−1	6	Slave1
6	3	0	8	Slave3

3.3　ACTS总览　SWPTS主要用在初始调度,但当集群中已使用资源高于设定阈值ε时,如果不考虑全局的最优调度,则资源有可能进一步降低,负载持续不均衡.此时采用ACTS,可以得到一种全局的任务分配方案,按照这个方案把Task分配给指定的Slot运行能使集群处于最佳运行状态,此时集群资源的利用率最好.

蚁群算法(Ant Colony Algorithm)是一种模拟蚂蚁觅食行为的模拟优化算法,由意大利学者Dorigo and Gambardella[20]于1991年首先提出,并首先使用在解决TSP(旅行商问题)[21]上.蚁群算法的基本原理如下:

(1)蚂蚁在随意行走的路径上释放信息素[22],这些信息素有利于后面的蚂蚁继续寻找路径.

(2)碰到没走过的路口就随机挑选一条路走,同时释放与路径长度有关的信息素.

(3)信息素浓度与路径长度成反比.后来的蚂蚁再次碰到该路口时,选择信息素浓度较高的路径.

(4)随着蚂蚁觅食过程中信息素的不断累积,最优路径上的信息素浓度越来越大,让后续的蚂蚁更多地选择信息素浓度高的这条路径.

(5)一段时间后,蚁群找到最优觅食路径,是一条全局的最短路径,即距离食物源最近.

如图4所示,路径A-B-C比A-D-C短,同样时间内,经过路径A-B-C的蚂蚁数量比A-D-C多,因此路径A-B-C的信息素越来越多,所以更多的蚂蚁也选择这条路径,A-B-C即是要寻找的最短路径.

ACTS就是根据蚁群算法改造的.经典的蚁群算法是寻找一条最短路径,而ACTS的目标是寻找一种全局最优分配方案,使Task的完成时间最短、资源使用率最好、集群重新负载均衡.ACTS以Task的运行时间为衡量信息素增减的标准,由每只蚂蚁把每个任务分配给对应的节点.由于分布式并行的特点,每只蚂蚁取运行时间最长的Task所需要的时间作为Job的完成时间.

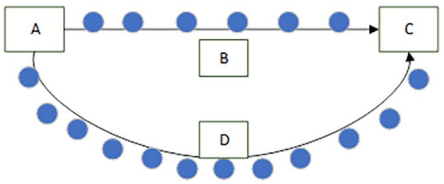

图4　蚁群算法示意图

Fig. 4　Ant colony algorithm

算法的整体框架如算法2所示,主要包括三部分:第一部分是初始化任务集合,第二部分是初始化信息素矩阵,第三部分是迭代搜索.

算法2　ACTS

输入:迭代次数*iteratorNum*,蚂蚁数量*antNum*,任务集合*tasks*,节点集合*nodes*,信息素矩阵*pheromoneMatrix*

输出:任务分配集合paths

1. *InitTaskMatrix* (*tasks*,*nodes*)
2. *initPheromoneMatrix* (*pheromoneMatrix*)
3. *ActsSearch* (*iteratorNum*,*antNum*);

步骤1,初始化任务矩阵.$timeMatrix[i][j]$表示任务i分配给节点j所需的时间.根据任务集合tasks和节点集合nodes,应用式(5)计算每个任务在每个节点完成需要的时间.表2是一个示例,$Task[1]$,$Task[2]$,$Task[3]$对应的数据规模大小分别是10,8,6;$Node[1]$,$Node[2]$,$Node[3]$对应的处理能力分别为2,2,1;每个任务在每个节点完成需要的时间如表2所示.

表2　初始化任务矩阵

Table 2　Initialization task matrix

	任务(数据集规模)		
节点名称(处理能力)	$Task[1](10)$	$Task[2](10)$	$Task[3](6)$
$Node[1](2)$	5	4	3
$Node[2](2)$	5	4	3
$Node[3](1)$	10	8	6

步骤2,初始化信息素矩阵.将负载均衡调度过程中的一次任务分配看作一条路径,因此 $pheromoneMatrix[i][j]$ 表示将任务 i 分配给节点 j 这条路径的信息素浓度.初始化信息素矩阵,将所有值置为1,因为初始时所有的路径都没有蚂蚁选择,默认为1.

步骤3,迭代搜索.这是ACTS最关键的一步,由三部分组成:迭代分配任务、计算每只蚂蚁的任务处理时间、更新信息素.整个迭代搜索的算法框架如算法3所示.

算法3 迭代搜索算法

输入:迭代次数 *iteratorNum*,蚂蚁数量 *antNum*

输出:最短时间的任务分配矩阵

```
1. for iteratorCount in iteratorNum
2.     init Path_AllAnt
3.     for antCount in antNum
4.         init Path_OneAnt
5.         for taskCount in tasks
6.             nodeCount=asignTask(antCount, taskCount)
7.         Path_OneAnt[taskCount][nodeCount]=1
8.     Path_AllAnt[i] equal to Path_OneAnt
9. put the task runtime of everyAnt to timeArray
10. put the shortest time and pathOneAnt of every ant in resultData
11. updatePheromoneMatrix(Path_AllAnt, Path_OneAnt, timeArray)
12. return resultData
```

算法3中,第2行表示初始化 *Path_AllAnt*,它是一个三维数组,用来存保第 *iteratorCount* 次迭代过程中第 *antCount* 只蚂蚁将 i 个任务分配给 j 个节点处理.第4行是初始化 *Path_OneAnt*,表示第 *antCount* 只蚂蚁将 i 任务分配给 j 节点处理.第5~8行表示循环每个任务,为每个任务通过任务分配函数 asignTask 分配到一个节点 node,给相应的 *Path_OneAnt*,*Path_AllAnt* 赋值.第9行和第10行是计算本次迭代中所有蚂蚁的任务处理时间,并将所有蚂蚁的任务处理时间加入总结果集 *resultData*.第11行是更新信息素.最后返回 *resultData*.

在整个蚁群算法中,共进行 *iteratorNum* 次迭代.每次迭代都会产生当前的最优分配策略,即"局部最优解",迭代的次数越多,局部最优解就越接近全局最优解.但迭代次数过多会造成调度器大量的时间和性能开销,无法满足海量任务的调度;而迭代次数太少,则得到的可能不是全局最优解.本文采用固定迭代次数为100次.

任务分配函数负责将一个指定的任务按照某种策略分配给某一节点处理.分配策略有两种:(1)按信息素浓度分配,即是将任务分配给本行中信息素浓度最高的节点处理.例如:当前的任务编号是 *taskCount*,当前的信息素浓度矩阵是 *pheromoneMatrix*,则任务会分配给 $pheromoneMatrix[taskCount]$ 这一行中信息素浓度最高的节点.(2)随机分配,即将任务随意分配给某个节点处理,一般根据蚂蚁的编号 *antNum* 来选择.$boundPointMatrix[i]=5$ 表示编号为0~5的蚂蚁在分配任务 i 的时候采用"按信息素浓度"的方式分配,即将任务 i 分配给信息素浓度最高的节点处理;而编号为6~9的蚂蚁在分配任务 i 时,采用随机分配策略.

蚁群算法有三个问题需要注意:

(1)计算任务处理问题.由于Flink集群里任务都是并行运行的,因此在计算任务处理时通常以最晚完成的任务的时间为整个Job的完成时间.

(2)更新信息素问题.将所有路径的信息素降低 $m\%$,表示信息素的挥发;找出所有蚂蚁的最短路径,则该路径的信息素提升 $n\%$,表示该路径是最短路径,信息素不断提升.

(3)更新 *boundPointMatrix* 问题.*boundPointMatrix* 表示临界蚂蚁下标集合,在该下标之前的蚂蚁选择信息素浓度最高的节点分配,在该下标之后的蚂蚁选择随机一个节点分配,计算方式如式(6)和式(7)所示.

4 实验分析

本文的调度器及算法均已在Flink中实现.通过修改Flink runtime包下面的调度模块,可添加数据HTTP Api数据访问和数据解析器.整个系统依赖Redis和maskmonitor性能监视应用.使用SWPTS和使用默认调度算法的Flink系统在延时、吞吐量、运行时间方面做了对比和分析,并在不同的数据集和不同并行度上进行了实验验

证. 实验使用Flink自带的实例WordCount. 由于临界资源阈值是个经验值,不同阈值效率对比如图5. 根据图5可以看出阈值为80%效率最高,因此本文实验默认阈值设定为80%.

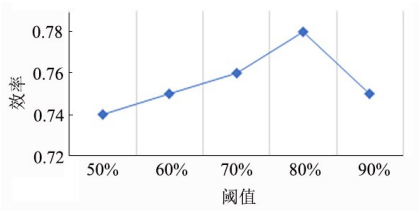

图5　Flink临界阈值效率对比

Fig. 5　Comparison of Flink critical threshold efficiency

4.1　数据集　针对WordCount使用真实数据集和模拟数据集. 真实数据集是TPC-C[23],用九个表生成模拟五种事务处理,产生三个大数据集,模拟真实场景下的批计算. 此外,自己写程序生成三个只包括字符串的模拟数据集,用以扩充实验测试. 数据集的来源和规模如表3所示.

表3　数据集规模

Table 3　The size of datasets

数据集	来源	大小
TPC-C Data1	TPC-C	1 G
TPC-C Data2	TPC-C	2 G
TPC-C Data3	TPC-C	3 G
MyData1	程序生成	1 G
MyData2	程序生成	2 G
MyData3	程序生成	3 G

4.2　实验环境及配置　SWPTS和ACTS均基于Flink1.4.2,编程语言为Java. 实验所用的集群硬件配置和参数如下:

实验环境:分布式集群由一台服务器和两台虚拟主机构成,两台虚拟主机模拟异构集群的效果,服务器主机为Master节点,两台虚拟机为Slave节点.

Master节点的配置如下:CPU为Intel(R) Core(TM) i7-6700,4 Core,主频3.40 GHz;内存为64 GB 2133 MHz;机械硬盘为WDC WD10 EZEX-08WN4A0 1 TB;编程环境为IntelliJ IDEA 2018, Maven, Git;操作系统为Ubuntu 16.04.5;Flink版本1.4.2,JDK版本1.8.0_151,Hadoop版本2.7.5.

Slave1的配置如下:CPU为Intel(R) Core(TM) i5-4690,4 Core,主频3.50 GHz;内存为4 GB 2133 MHz;机械硬盘60 GB;编程环境为IntelliJIDEA 2018,Maven,Git;操作系统为Ubuntu 16.04.5;Flink版本1.4.2,JDK版本1.8.0_151,Hadoop版本2.7.5.

Slave2的配置如下:CPU为Intel(R) Core(TM) i5-4690,2 Core,主频3.50 GHz;内存为2 GB 2133 MHz;机械硬盘60 GB;编程环境为IntelliJ IDEA 2018,Maven,Git;操作系统为Ubuntu 16.04.5;Flink版本1.4.2,JDK版本1.8.0_151,Hadoop版本2.7.5.

4.3　实验结果及分析　通过修改Flink1.4.2的默认任务调度器为自适应任务调度器,新增SWPTS和ACTS两种负载均衡调度算法. 在异构集群中不同的数据集下分别与Flink1.4.2在运行时间、延时、吞吐量[24]方面做性能对比,可以看出改进之后的算法在三个方面都有所提升.

4.3.1　运行时间对比分析　通常运行时间是一个可以让用户对比算法性能的最直观的指标. 运行快说明算法节省了Job的完成时间,加快了用户响应时间,对提升性能非常重要. SWPTS和ACTS能将任务尽可能多地分配给资源丰富的节点,应用到自适应任务调度器中能使资源丰富的节点更多地接收Task,比Flink默认调度器的负载更均衡,运行时间也相应缩短.

在并行度为8和16时,在自己程序生成的数据集(图6)和TPC-C数据集(图7)上进行对比实验,可以看到,应用自适应任务调度器(LoadbalanceFlink)之后的运行时间比默认任务调度器(NativeFlink)平均减少8%左右,这是因为前者将任务均匀分配,资源多的节点可以完成尽可能多的Task,缩短了整体的完成时间,而默认任务调度器让资源少的节点完成和其他节点同等的任务,拖慢了整体工作的完成进度.

4.3.2　吞吐量对比分析　吞吐量(Throughput)指单位时间内由计算引擎成功处理的数据量,反映系统的负载能力. 吞吐量常用于资源规划,也能协助分析系统性能瓶颈,从而进行相应的资源

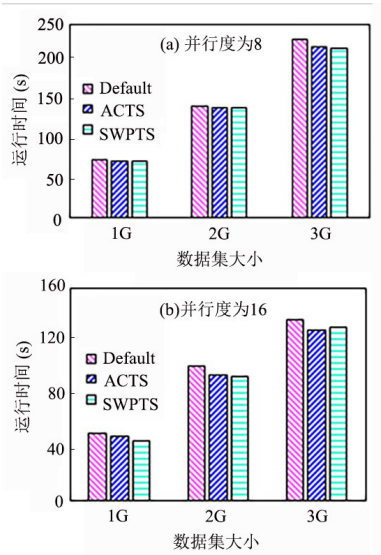

图 6 Flink 运行时间对比(自定义数据集)

Fig.6 Flink runtime on custom dataset

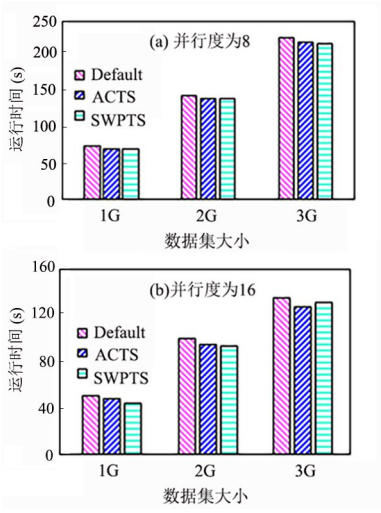

图 7 Flink 运行时间对比(TPC-C 数据集)

Fig.7 Flink runtime on TPC-C dataset

调整以保证系统达到用户要求的处理能力.

实验使用阿里巴巴提供的 advertising 工具,它是标准流测试工具 yahoo stream benchmark 的简化版.测试原理是随机产生两个广告流,把 ad_id 相同的 join 起来存放到 Redis 里,通过单位时间内在 Redis 读到多少条数据来计算吞吐量.计算如式(8)所示:

$$throughput = \frac{currentNum - lastNum}{currentTime - lastTime} \tag{8}$$

其中,*currentNum* 代表当前读到的数据编号,即已经读到多少条数据;*lastNum* 表示前一次读到的数据编号;*currentTime* 表示当前时间,*lastTime* 表示上一次的时间.

从实验结果(图 8)可以看出,改进之后的 Flink 自适应负载均衡算法比默认任务调度算法的吞吐量更高,原因同前,本文算法能使负载均衡,即资源多的节点负载较多的任务,则整个集群在单位时间内能完成更多的任务,吞吐量增大.

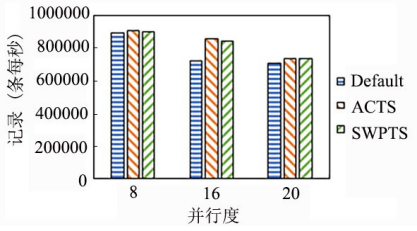

图 8 Flink 不同并行度下的吞吐量对比

Fig. 8 Throughput of Flink with different parallelism

4.3.3 延时对比分析 延时(latency)指数据从进入数据窗口的时间到真正被处理的时间间隔,单位为毫秒(ms),反映系统处理的实时性.金融交易分析等大量实时计算业务对延迟要求较高,因为延时越小,数据实时性越强.

实验使用阿里巴巴提供的 advertising 工具,原理是随机产生两个广告流,把 ad_id 相同的 join 起来存放到 Redis 里.计算如式(9)所示:

$$latency = handleTime - windowTime \tag{9}$$

其中,*handleTime* 表示某条记录实际被处理的时间,*windowTime* 表示流里面该记录属于的时间窗口开始时间.

从实验结果(图 9)可以看出,本文改进之后的 Flink 自适应负载均衡算法比默认的调度算法延时更小,原因亦同前,本文算法能够实现负载均衡,使资源多的节点能负载较多的任务,则每个任务在被处理之前需要等待的时间也相应变短,即延时变小.

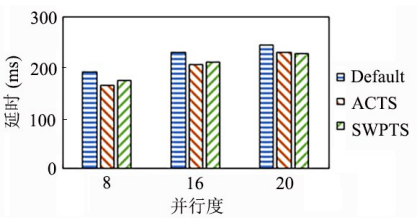

图 9 Flink 不同并行度的延时对比

Fig. 9 Delay of Flink with different parallelism

上述在运行时间、吞吐量、延时三方面的实验表明:SWPTS和ACTS改变了任务的默认分配策略,可以尽量按动态资源的大小将任务优先分配给资源较多的节点,解决异构集群负载不均衡的问题.

5 总　结

本文提出的自适应负载均衡算法由平滑加权轮询任务调度算法(SWPTS)和基于蚁群算法的任务调度算法(ACTS)组成.经过实验验证,在异构Flink集群的环境下,自适应负载均衡算法的运行时间、吞吐量和延时与默认调度算法相比都有所提升.在运行初期,利用SWPTS负载均衡,使得任务在初始分配的时候负载均衡.在运行过程中,当集群已使用资源高于设定阈值时,采用ACTS寻找一种全局最优分配方案,也能重新均衡负载.等已使用资源低于设定阈值时,则继续采用之前的算法进行调度.

参考文献

[1] Wang Y C, Kung L A, Byrd T A. Big data analytics: understanding its capabilities and potential benefits for healthcare organizations. Technological Forecasting and Social Change, 2018, 126: 3－13.

[2] Gummaraju J, Mcdougall R, Nelson M, et al. Container virtual machines for hadoop. U.S. Patent 20150120928, 2015－04－30.

[3] Afrati F N, Stasinopoulos N, Ullman J D, et al. SharesSkew: an algorithm to handle skew for joins in mapReduce. Information Systems, 2018, 77: 129－150.

[4] Verma S, Das L M. Spark advance modeling of hydrogen - fueled spark ignition engines using combustion descriptors. Journal of Engineering for Gas Turbines and Power, 2018, 140(8): 081501.

[5] Kamburugamuve S, Wickramasinghe P, Ekanayake S, et al. Anatomy of machine learning algorithm implementations in MPI, Spark and Flink. The International Journal of High Performance Computing Applications, 2018, 32(1): 61－73.

[6] Katayama Y. Memory access for exactly - once messaging. U. S. Patent 20180095878, 2018－04－05.

[7] Kc K, Anyanwu K. Scheduling hadoop jobs to meet deadlines// 2010 IEEE 2nd International Conference on Cloud Computing Technology and Science. Indianapolis, IN, USA: IEEE, 2010: 388－392.

[8] Zaharia M. Job scheduling with the fair and capacity schedulers. Hadoop Summit, 2009, 9.

[9] Zaharia M. Job scheduling with the fair and capacity schedulers. Hadoop Summit, 2009, 9.

[10] Lee J, Kim B, Chung J M. Time estimation and resource minimization scheme for apache spark and hadoop big data systems with failures. IEEE Access, 2019, 7: 9658－9666.

[11] Kc K, Anyanwu K. Scheduling hadoop jobs to meet deadlines// 2010 IEEE 2nd International Conference on Cloud Computing Technology and Science. Indianapolis, IN, USA: IEEE, 2010: 8－16.

[12] Cheng D Z, Chen Y, Zhou X B, et al. Adaptive scheduling of parallel jobs in spark streaming// IEEE Conference on Computer Communications. Atlanta, GA, USA: IEEE, 2017: 1－9.

[13] Chen H K, Wang F Z. Spark on entropy: a reliable & efficient scheduler for low - latency parallel jobs in heterogeneous cloud// 2015 IEEE 40th Local Computer Networks Conference Workshops. Clearwater Beach, FL, USA: IEEE, 2015: 708－713.

[14] Mao H Z, Schwarzkopf M, Venkatakrishnan S B, et al. Learning scheduling algorithms for data processing clusters// Proceedings of the ACM Special Interest Group on Data Communication. New York, NY, USA: ACM, 2018: 8－12.

[15] Ren X Q, Ananthanarayanan G, Wierman A, et al. Hopper: decentralized speculation - aware cluster scheduling at scale// Proceedings of 2015 ACM Conference on Special Interest Group on Data Communication. New York, NY, USA: ACM, 2015: 8－12.

[16] Peng B Y, Hosseini M, Hong Z H, et al. R-storm: resource-aware scheduling in storm// Proceedings of the 16th Annual Middleware Conference. New York, NY, USA: ACM, 2015: 1－6.

[17] Chen Y R, Lee C R. G-storm: A GPU-aware storm scheduler// 2016 IEEE 14th International Conference on Dependable, Autonomic and Secure Computing, the 14th International Conference on Pervasive Intelligence and Computing, the 2nd International

Conference on Big Data Intelligence and Computing and Cyber Science and Technology Congress. Auckland, New Zealand: IEEE, 2016: 9－16.

[18] Anderson B, Chi A, Dunlop S, et al. Limitless HTTP in an HTTPS world: inferring the semantics of the HTTPS protocol without decryption// Proceedings of the 9th ACM Conference on Data and Application Security and Privacy. New York, NY, USA: ACM, 2019: 267－278.

[19] Vasavi S, Priyanka G V N, Gokhale A A. framework for visualization of geospatial query processing by integrating redis with spark. International Journal of Natural Computing Research, 2019, 8(3): 1－25.

[20] Dorigo M, Gambardella L M. Ant colony system: a cooperative learning approach to the traveling salesman problem. IEEE Transactions on Evolutionary Computation, 1997, 1(1): 53－66.

[21] Haddadan A, Newman A. Towards improving christofides algorithm on fundamental classes by gluing convex combinations of tours. 2019, arXiv: 1907.02120.

[22] Mirjalili S. Ant colony optimisation//Evolutionary Algorithms and Neural Networks. Springer Berlin Heidelberg, 2019: 33－42.

[23] Kamsky A. Adapting TPC-C benchmark to measure performance of multi - document transactions in MongoDB. Proceedings of the VLDB Endowment, 2019, 12(12): 2254－2262.

[24] Doupe M B, Chateau D, Chochinov A, et al. Comparing the effect of throughput and output factors on emergency department crowding: a retrospective observational cohort study. Annals of Emergency Medicine, 2018, 72(4): 410－419.

(责任编辑 杨可盛)