CN 43-1258/TP
ISSN 1007-130X

计算机工程与科学
Computer Engineering & Science

第 44 卷第 7 期 2022 年 7 月
Vol. 44, No. 7, Jul. 2022

文章编号:1007-130X(2022)07-1141-11

面向 Flink 的负载均衡任务调度算法的研究与实现*

李文佳[1],史　岚[1],季航旭[1],罗意彭[2]
(1.东北大学计算机科学与工程学院,辽宁 沈阳 110169;2.辽宁工业大学软件学院,辽宁 锦州 121000)

摘　要:Apache Flink 是现在主流的大数据分布式计算引擎之一,其中任务调度问题是分布式计算系统中的关键问题。由于集群的异构性以及不同算子复杂度不同,大数据计算系统 Flink 中不可避免地会出现负载不均的情况,针对这种问题,提出了基于资源反馈的负载均衡任务调度算法 RFTS。通过实时资源监控、区域划分和基于人工萤火虫优化的任务调度算法 3 个模块,把负载过重的机器中处于等待状态的任务分配给负载较轻的机器,来实现集群的负载均衡,提高系统集群利用率和执行效率。最后通过基于 TPC-C 和 TPC-H 数据集的实验结果表明,RFTS 算法从执行时间和吞吐量 2 个方面有效提升了 Apache Flink 计算系统的性能。

关键词:Apache Flink;基于资源反馈的负载均衡任务调度算法;实时资源监控;区域划分;人工萤火虫优化算法

中图分类号:TP391　　**文献标志码**:A

doi:10.3969/j.issn.1007-130X.2022.07.001

Research and implementation of a Flink-oriented load balancing task scheduling algorithm

LI Wen-jia[1], SHI Lan[1], JI Hang-xu[1], LUO Yi-peng[2]
(1. College of Computer Science and Engineering, Northeastern University, Shenyang 110169;
2. School of Software, Liaoning University of Technology, Jinzhou 121000, China)

Abstract: Apache Flink is one of the mainstream big data distributed computing engines, and task scheduling is a key issue in distributed computing systems. Due to the heterogeneity of clusters and the different complexity of operators, uneven load will inevitably appear in the big data computing system Flink. To solve this problem, a load balancing task scheduling algorithm based on resource feedback, named RFTS, is proposed. Through the three modules (real-time resource monitoring, area division, and task scheduling algorithm based on glowworm swarm optimization), the tasks in the waiting queue in the over-loaded machine are allocated to the lighter-loaded machines, so as to reduce the load unevenness of the entire cluster and improve the cluster utilization and execution efficiency of the system. Finally, through the experimental verification based on the TPC-C and TPC-H datasets, the results show that the load balancing task scheduling algorithm based on resource feedback (RFTS) can effectively improve the performance of the Apache Flink computing system in terms of execution time and throughput.

Key words: Apache Flink; load balancing task scheduling algorithm based on resource feedback; real-time resource monitoring; area division; glowworm swarm optimization algorithm

* **收稿日期**:2021-11-10;**修回日期**:2022-01-17
基金项目:科技部重点研发项目(2018YFB1004402)
通信作者:史岚(shilan@cse.neu.edu.cn)
通信地址:110169 辽宁省沈阳市东北大学计算机科学与工程学院
Address: College of Computer Science and Engineering, Northeastern University, Shenyang 110169, Liaoning, P. R. China

1 引言

近年来,随着数据经济在全球的加速推进,以及5G、人工智能、物联网等相关技术的快速发展,全球数据量迎来巨大规模的爆发,越来越多的政府机构和研究人员开始重视这种大量数据的收集、使用与处理。伴随着数据的爆炸式增长与多种多样需求的出现,一些传统的大数据模型和分布式计算引擎已经很难满足当前业务的需求,因此,许多新的分布式计算框架应运而生。大数据计算引擎的发展历程主要分为4个阶段。第一代大数据计算引擎是谷歌于2004年提出的基于MapReduce[1]的Hadoop[2]计算引擎。Hadoop主要依靠把任务拆分成map和reduce 2个阶段去处理,这种模式由于难以支持迭代计算,因此产生了第二代基于有向无环图DAG(Directed Acyclic Graph)[3]的以Tez[4]和Oozie为代表的计算引擎。虽然第二代计算引擎解决了MapReduce中不支持迭代计算的问题,但是由于这种计算引擎只能处理离线任务,在线任务处理需求增加的驱动下,产生了第三代基于弹性分布式数据集RDD(Resilient Distributed Dataset)[5]的Spark[6]计算引擎。Spark既可以处理离线计算也可以处理实时计算,它是在Tez的基础上对Job作了更细粒度的拆分,但是其延迟较大,难以处理实时需求更高的连续流数据请求。因此,产生了现在主流的可以处理高实时性任务的第四代大数据计算引擎Flink[7]。Flink对事件时间的支持、精确一次(Exactly-Once)的状态一致性以及内部检查点机制等特性,决定了其在大数据计算引擎上占据主流地位。现如今越来越多的公司采用基于Flink的大数据计算引擎去实现多种场景,比如阿里巴巴双十一实时大屏的投放、腾讯实时平台的搭建以及美团、饿了么、爱奇艺等公司的数据处理流程都是基于Flink构建的。

在大数据计算引擎Flink中,大量的计算任务需要被调度到资源节点上,如何使整体任务用最少的完成时间,在很大程度上由它的调度算法决定,因此良好的任务调度算法是分布式计算的重要组成部分。有效的任务调度是分布式计算的一个关键问题,其目标是在满足任务依赖关系的前提下,调整任务的执行顺序,将任务分配给对应的资源,使整个系统的任务能在最短的时间内执行完成。

由于集群的异构性以及不同算子复杂度不同,大数据计算系统中不可避免地会出现负载不均的情况,本文提出了基于资源反馈的负载均衡任务调度算法RFTS(load balancing Task Scheduling algorithm based on Resource Feedback)。与传统的负载均衡算法不同的是,RFTS算法综合考虑了集群计算资源的实时负载情况以及处理任务的优先级和顺序,更高效地完成任务与计算资源之间的分配,通过实时资源监控、区域划分和基于人工萤火虫优化GSO(Glowworm Swarm Optimization)的任务调度算法3个模块,把负载过重的机器中处于等待队列中的任务分配给负载较轻的机器,提高系统处理任务的执行效率和集群利用率。

本文的主要贡献包括以下3个方面:

(1) 设计了一个Flink系统的实时监控系统Monitor,实时监控每个从节点(Slave)的CPU核数、CPU利用率、内存利用率和总内存大小等性能指标,从而获取每个资源节点的负载大小。

(2) 提出基于资源反馈的负载均衡任务调度算法。该算法在集群出现负载不均时,重新分配每个资源节点的任务,以提高系统执行效率和整体资源利用率。该算法通过实现的实时监控系统Monitor来监控资源节点的负载情况,并根据区域划分算法把集群划分为过负载、轻负载、近饱和和差饱和4个区域,由于过负载区域的机器负载过重会影响整个集群的执行效率,因此用基于人工萤火虫优化算法的调度策略,把过负载区域中资源节点位于等待队列的任务调度给差饱和区域的资源节点。

(3) 通过编写源码,在大数据计算系统Flink中,实现了RFTS算法。

最后在TPC-C和TPC-H[8]数据集上对基于资源反馈的负载均衡任务调度算法进行了实验,实验结果表明,该算法在执行时间和吞吐量方面均有明显的提升效果.

2 相关工作

任务调度是指系统将用户提交的任务通过某种方式进行拆分、重组并分配到集群中对应的资源节点进行计算的过程。众所周知,任务调度问题是NP-hard[9]。大数据计算模型是一种新型的分布式计算模型,专门用于处理海量数据的存储、分析和计算,其优点在于能以使用较低的时间和空间成本来实现系统的高可扩性和可伸缩性。这些决定了大数据计算模型不仅可以应对数据日益增长的计算、存储和分析需求,也可以很好地满足并适应网络环境中复杂多变的特点,保障基本的网络性能

请求。大数据计算中资源的服务质量好坏是衡量大数据计算效果的一个重要方面,但是大数据计算中云端存在诸多形态,且系统规模巨大,资源节点之间的结构差异性也较大,因此如何更好地实现任务调度就成为了大数据计算研究中的热点和难点问题。

调度算法的目的是将任务调度到资源节点的同时使得任务的执行时间和吞吐量尽可能好。任务执行所需要的资源、网络 IO、耗费的时间和用户需求等都由任务调度策略决定。因此,在分布式计算中任务调度很大程度上决定了分布式计算系统的系统性能。

分布式计算中常用的调度算法有经典调度算法和启发式调度算法。经典调度算法主要有 Max-Min 算法[10]、Max-Max 算法和 Sufferage 算法[11]、先到先服务、轮询调度算法[12]以及公平调度算法[13]等,这些算法由于参数少、操作简单、容易复现等优点被广泛使用。但是,这类算法也存在着面对复杂场景和数据频繁被调度的场景时任务分配效果较差、容易导致数据倾斜等缺点[14]。因此,研究人员提出了主要针对复杂场景的启发式调度算法。Holland 于 1975 年通过观察生物界进化的规律,通过组合交叉、遗传和变异的方式提出了遗传算法[15];1991 年意大利学者 Dorigo 通过模拟蚁群根据信息素的反馈信息不断改变寻找食物的速度和方向的行为提出了蚁群算法[16];1995 年 Eberhart 和 Kenny 博士通过观察鸟群在迁移过程中根据其他鸟类的飞行轨迹来改变自身速度和位置的行为模式,提出了粒子群算法[17];还有现在被广泛使用的模拟退火算法[18]、差分演化算法[19]等。启发式调度算法虽然适合复杂场景并能够快速展开全局搜索,但也存在随机性过高、容易陷入局部最优解和参数难以控制等缺点。

Flink 系统中默认的调度策略是轮询调度算法,这种算法不考虑每个资源节点的异构性,容易导致集群负载不均,使性能强的资源节点和性能差的资源节点面对同样多的任务,这时性能差的资源节点需要更多的时间,根据水桶效应,系统的执行效率是由性能最差的节点所决定的,因此集群负载不均会降低整个系统的执行效率。

基于上述分析可知,现有的调度算法不足以高效优化实时大数据计算系统 Flink 的执行速度,因此需要研究新的任务调度算法,使其在面对 Flink 系统复杂场景时能保证负载均衡且尽可能地提高系统执行效率。所以,本文提出了基于资源反馈的负载均衡任务调度算法 RFTS。与传统的负载均衡调度算法相比,RFTS 算法根据集群中每台机器当前的负载压力和任务优先级来快速而高效地分配任务,可以在不降低任务处理实时性的前提下,提高系统处理任务的总体执行效率。

3 基于资源反馈的负载均衡任务调度算法 RFTS

由于集群的异构性以及不同算子复杂度不同,分布式计算系统中不可避免地会出现负载不均的情况。针对该问题,本文提出了基于资源反馈的负载均衡任务调度算法。

在本节中,首先介绍 RFTS 算法的基本思想;接着,借助实例分别介绍 RFTS 算法包含的 3 个主要模块。

3.1 定义与说明

首先对 RFTS 中用到的变量和公式进行定义与说明。令 $TM=\{TM_1,TM_2,\cdots,TM_m\}$ 为分布式系统中资源节点的集合,$T=\{T_1,T_2,\cdots,T_n\}$ 为需要执行的全部任务的集合。

定义 1(任务整体完成时间) 任务整体完成时间是指从第一个任务执行开始到所有任务中最后一个任务完成所经历的时间,记为 $TotalTime$。负载均衡的目标是使得 $TotalTime$ 尽可能小。

定义 2(资源节点性能指标) 资源节点的性能指标如式(1)所示:

$$Metric_j=C_1*Core\_Num_j*\omega_{1j}+C_2*(1-\omega_{2j})*\omega_{3j},j=1,2,\cdots,m \quad (1)$$

其中,C_1、C_2 为常数,$Core\_Num_j$ 表示资源节点 TM_j 的 CPU 核数,ω_{1j} 表示资源节点 TM_j 的 CPU 利用率,ω_{2j} 表示资源节点 TM_j 的内存利用率,ω_{3j} 表示资源节点 TM_j 的总内存大小。因此,资源节点性能指标 $Metric_j$ 由 CPU 核数、CPU 利用率和空闲内存大小决定,最后作归一化处理。

定义 3(资源节点负载值) 负载值代表该资源节点上任务负载量相对于该节点当前性能的承担能力。负载值越小,表示该节点承受负载的能力越好,即可以承受更多的任务;反之,负载值越大,该节点承受负载的能力越差,需要减少该节点执行的任务。负载值的定义如式(2)所示:

$$Load_j=\frac{TaskNum_j}{Metric_j},j=1,2,\cdots,m \quad (2)$$

其中,$TaskNum_j$ 表示资源节点 TM_j 的任务队列中的任务数量;$Metric_j$ 代表资源节点 TM_j 的性

能指标，$Metric_j$ 越大，代表该资源节点性能越好，可以在单位时间内处理更多的任务；反之，则代表该资源节点的性能越差，在单位时间内能够处理的任务越少。

集群负载平均值的定义如式(3)所示：

$$\overline{Load}=\frac{1}{m}\sum_{j=1}^{m}Load_j \tag{3}$$

定义 4(集群负载值) 集群负载值表示整个集群的负载均衡程度，如式(4)所示。如果集群负载值 C_Load 小于阈值 β，则说明集群整体负载处于均衡状态，算法结束；否则，集群负载值越大，说明集群负载不均的程度越高，此时需要将每个节点的负载值和集群负载值存入 MongoDB 数据库中，为下一阶段的区域划分和任务重新调度做准备。

$$C\_Load=\sqrt{\frac{1}{m}\sum_{j=1}^{m}(Load_j-\overline{Load})^2} \tag{4}$$

3.2 主要思想

由于 Flink 系统中任务以随机顺序调度，本文提出了一种基于资源反馈的负载均衡任务调度算法，算法构建了一个优化器，通过集群资源实时监控、集群区域划分和任务重新分配去优化 Flink 系统中的调度策略，以避免负载不均的情况。

RFTS 算法的总体框架如图 1 所示，其基本思想是先通过实时监控集群的性能情况，得到每个资源节点的负载值和整个集群的负载值，并把这些信息实时存储到 MongoDB 数据库中，集群的负载越小，则说明整个集群的负载越均衡。因此，当负载均衡值小于阈值时，说明集群处于均衡状态，算法结束；否则，利用区域划分算法划分整个集群，把集群分成过负载、轻负载、近饱和和差饱和 4 个区域，最后采用基于人工萤火虫优化的任务调度算法结合任务的优先级把过负载区域的任务迁移到差饱和区域中的资源节点上，以降低整个集群的负载值。

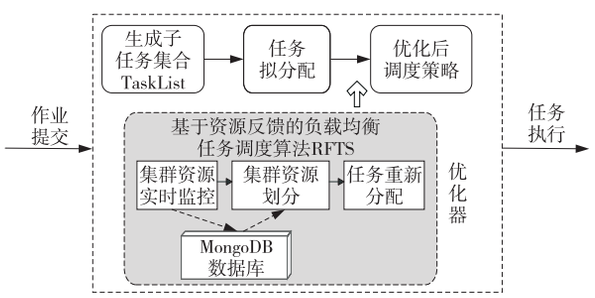

Figure 1 General framework of RFTS

图 1 RFTS 算法总体框架

本节进一步优化基于 Flink 本身的调度算法，针对已经完成初步调度策略的集群进行实时性能监控，当负载不均时根据各节点任务队列中任务的数量和任务的优先级重新产生调度策略。

3.3 资源监控系统

资源监控系统为 RFTS 算法后期的区域划分和基于人工萤火虫优化的任务调度收集重要的系统实时信息。该系统每隔 10 s 对每个资源节点统计一次实时资源性能数据。在计算资源中最具代表性的资源性能数据包括 CPU 核数、CPU 使用率、内存使用率和总内存大小，这 4 个指标反映了节点当前的负载能力。因此，本文用这 4 个指标构成的综合值来代表该资源节点的实时性能，并通过计算得到每个资源节点的负载值，进而得到整个集群的负载值。

资源监控系统的具体流程如算法 1 所示，当集群负载值大于或等于 β 时(第 1 行)，对于集群中的每个资源节点 TM_j，根据监控和管理 Java 虚拟机(JVM)管理接口的 $ManagementFactory$ 管理工厂类中的 $getOperatingSystemMXBean$ 方法去获取资源节点底层的性能指标，计算节点 TM_j 的 CPU 核数 $Core\_Num_j$、CPU 利用率 ω_{1j} 和空闲内存量 ω_{2j}，根据式(1)和式(2)可以得出该资源节点 TM_j 的性能指标 $Metric_j$ 和负载值 $Load_j$(第 2～5 行)；根据式(4)得出集群的负载值，并把每个节点的负载值和集群负载值放入数据库 MongoDB 中(第 6、7 行)，直到集群负载值小于阈值 β，即集群实现整体的负载均衡。

算法 1 资源监控系统实现算法

输入：资源节点集合 TM、每个资源节点上的任务分配情况 $TaskNum$。

输出：数据库 MongoDB。

1. **While** 集群负载值大于或等于阈值 β **do**
2. **For** 集群中的每一个资源节点 **do**
3. 计算资源节点 TM_j 的 CPU 核数、CPU 利用率和当前机器的空闲内存量；
4. 更新资源节点 TM_j 的性能指标和负载值；
5. **End for**
6. 更新集群的整体负载值；
7. 把每个节点的负载值和集群负载值存入数据库 MongoDB 中；
8. 输出数据库 MongoDB；
9. **End while**

下面通过一个实例来描述资源监控系统的执行过程，如图 2 所示。对于整个集群而言，每隔 10 s 统计一次每个资源节点的 CPU 核数、CPU 利用率、内存利用率和总内存大小，根据式(1)计算出

每个节点的性能指标 $Metric_j$,根据每个节点上当前任务队列中的任务数量和实时性能指标计算每个节点的负载值 $Load_j$ 和集群负载值 C_Load,并把这些信息实时存入 MongoDB 数据库中,重复上述过程直到集群负载值 C_Load 小于 β。

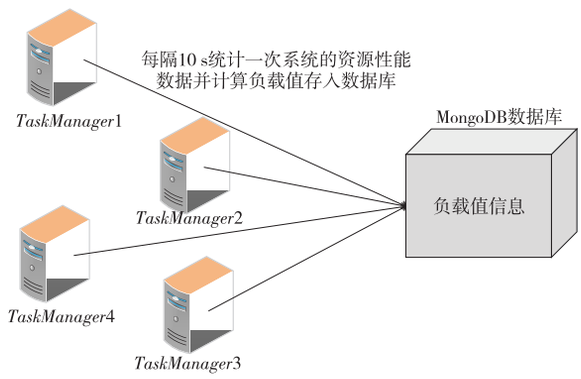

Figure 2 Execution of the resource monitoring system

图 2 资源监控系统执行过程

3.4 区域划分算法

本文将整个集群划分为过负载、轻负载、近饱和和差饱和 4 个区域,依次记为 $UPGroup$、$LPGroup$、$NSGroup$ 和 $DSGroup$。

通过定义如式(5)所示的偏移量 $offset$ 来划分集群区域。$h\_threshold$ 为启发式集群高域值,且 $h\_threshold \geqslant 0$;$l\_threshold$ 为低阈值,且 $l\_threshold \leqslant 0$。如果 $offset_j \geqslant h\_threshold$,则节点 TM_j 属于过负载区域 $UPGroup$;如果 $0 \leqslant offset_j < h\_threshold$,则节点 TM_j 属于轻负载区域 $LPGroup$;如果 $offset_j < l\_threshold$,则节点 TM_j 属于差饱和区域 $DSGroup$;如果 $l\_threshold < offset_j \leqslant 0$,则节点 TM_j 属于近饱和区域 $NSGroup$。

$$offset_j = \delta_j - \beta, j = 1,2,\cdots,m \tag{5}$$

其中资源节点 TM_j 负载偏差值 δ_j 如式(6)所示:

$$\delta_j = |Load_j - \overline{Load}|, j = 1,2,\cdots,m \tag{6}$$

区域划分算法的伪代码如算法 2 所示。当满足过负载区域 $UPGroup$ 不为空且差饱和区域 $DSGroup$ 不为空,或者集群负载值 $C\_Load < \beta$ 这 2 个条件中的一条时(第 1 行)开始区域划分算法,每隔 10 s 从 MongoDB 数据库中提取当前资源中每个节点的负载值 $load_j$,计算得出当前整个集群的平均负载,根据式(5)计算得到每个资源节点的当前偏移量 $offset_j$(第 3、4 行),并结合启发式集群高低阈值 $h\_threshold$ 和 $l\_threshold$ 计算出该节点的负载承受能力,根据该节点负载承受能力的高低把该节点分配到对应所属区域(第 5~16 行)。如果区域划分算法结束时 C_Load 小于 β,则算法结束,否则,进入到基于人工萤火虫优化的任务调度算法重新分配任务调度集合。

算法 2 区域划分算法

输入:数据库 MongoDB。

输出:过负载区域 $UPGroup$、轻负载区域 $LPGroup$、近饱和区域 $NSGroup$ 和差饱和区域 $DSGroup$。

1. **While** ($UPGroup \neq \varnothing \wedge DSGroup \neq \varnothing$) $\vee$ ($C\_Load < \beta$) **do**
2. **For** MongoDB 数据库中的每一个资源节点
3. 计算每个资源节点 TM_j 的负载偏差值 δ_j;
4. 计算每个资源节点 TM_j 的偏移量 $offset_j$;
5. **Case when** $offset_j \geqslant h\_threshold$ **then**
6. 把资源节点 TM_j 放入过负载区域 $UPGroup$;
7. **When** $0 \leqslant offset_j < h\_threshold$ **then**
8. 把资源节点 TM_j 放入轻负载区域 $LPGroup$;
9. **When** $offset_j < l\_threshold$ **then**
10. 把资源节点 TM_j 放入差饱和区域 $DSGroup$;
11. **When** $l\_threshold \leqslant offset_j < 0$ **then**
12. 把资源节点 TM_j 放入近饱和区域 $NSGroup$;
13. **End While**
14. 输出划分好的过负载区域 $UPGroup$、轻负载区域 $LPGroup$、近饱和区域 $NSGroup$ 和差饱和区域 $DSGroup$.

下面通过一个实例来描述区域划分算法的具

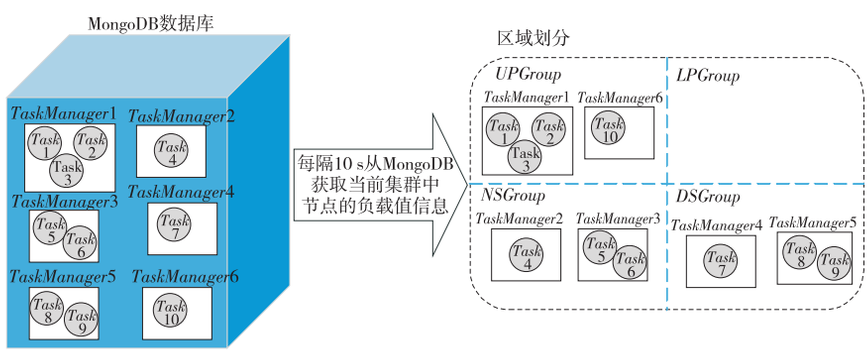

Figure 3 Execution of regional division

图 3 区域划分执行过程

体执行过程,如图3所示。假定当前集群由6个TaskManager组成,根据3.3节提出的资源监控系统,每个TaskManager的实时负载值都存储在MongoDB数据库中,每隔10 s从该数据库中重新提取当前资源中每个节点的负载值$Load_j$和集群的负载值C_Load,并计算得到每个节点相对于集群负载平均值的偏移量。该实例中,$TaskManager1 \sim TaskManager6$的偏移量分别是0.91,-0.09,-0.12,-0.58,-0.31和0.87,集群的高阈值为0.83,低阈值为-0.27,根据区域划分算法,分配结果如图3的右图所示,$TaskManager1$和$TaskManager6$被分配到$UPGroup$,$TaskManager2$和$TaskManager3$被分配到$NSGroup$,$TaskManager4$和$TaskManager5$被分配到$DSGroup$。

3.5 基于人工萤火虫优化的任务调度算法

人工萤火虫优化GSO算法是2005年由印度学者Krishnanand和Ghose提出的一种新型的全局智能优化算法。该算法定义了萤火虫的解空间,每个萤火虫都有决策域,即自己的视线范围。每只萤火虫的亮度与其所在位置的目标函数值有关,萤火虫位置的目标函数值越高,其亮度越大;相反,则亮度越小。根据萤火虫的自然生活规律,它将在决策域中找到下一次运动方向。在决策域中,区域越亮,对萤火虫的吸引力越强。萤火虫的飞行方向会根据邻域改变。另外,决策域的大小受邻域中个体数目的影响,当邻域密度减小时,萤火虫决策域半径增大,为了发现更多的邻居,邻域密度越大,其决策域半径越小。最终,大部分萤火虫会在一个区域内凝结,即达到极值点。

本节提出的基于人工萤火虫优化的任务调度算法的原理是,基于3.4节的区域划分算法把过负载区域$UPGroup$中的节点TM_j上的任务T_i按照基于人工萤火虫优化的任务调度策略调度到差饱和区域$DSGroup$中的节点TM_p上,$UPGroup$中的节点按照负载值的大小降序排序,依次调度到$DSGroup$中,直到集群负载值C_Load小于阈值β。在该算法中任务T_i被定义为萤火虫,$DSGroup$区域中的节点TM_p被定义为萤火虫的目标区域,任务的目标函数由$UPGroup$中节点TM_j的任务数量、节点TM_j中的任务T_i的优先级、目标节点TM_p的负载值和目标节点TM_p上高优先级任务的数量共同决定。下面先对本文提出的基于人工萤火虫优化的任务调度算法进行基本定义与概念说明:

令$N=\{n_1,n_2,\cdots,n_n\}$为萤火虫集合,初始化每个萤火虫的荧光素为l_0,决策域为r_0。萤火虫n_i在时刻t的荧光素值如式(7)所示:

$$l_i(t)=(1-\rho)l_i(t-1)+\gamma f(\boldsymbol{x}_i(t)),\quad i=1,2,\cdots,n \tag{7}$$

其中,ρ代表萤火虫中荧光素的消失率,γ代表荧光素的更新率,$f(\boldsymbol{x}_i(t))$表示萤火虫n_i在时刻t时位置$\boldsymbol{x}_i(t)$的目标函数值。

萤火虫n_i在t时刻的邻居集合$G_i(t)$如式(8)所示:

$$G_i(t)=\{n_j:\|\boldsymbol{x}_j(t)-\boldsymbol{x}_i(t)\|<r_{\mathrm{d}}^i(t);\ l_i(t)<l_j(t)\} \tag{8}$$

其中,$\boldsymbol{x}_i(t)$表示萤火虫n_i在t时刻所在的位置,$r_{\mathrm{d}}^i(t)$表示萤火虫n_i在t时刻时的决策域范围。

萤火虫n_i的速度方向$\boldsymbol{v}$的定义如式(9)所示。

$$\boldsymbol{v}=\max(\boldsymbol{p}_i),\boldsymbol{p}_i=\{p_{i1},p_{i2},\cdots,p_{iG_i(t)}\} \tag{9}$$

其中,

$$p_{ij}(t)=(l_j(t)-l_i(t))/\sum_{n_k\in G_i(t)}(l_k(t)-l_i(t)) \tag{10}$$

为萤火虫n_i向萤火虫n_j方向的转移概率。

萤火虫n_i在时刻t时的位置定义如式(11)所示:

$$\boldsymbol{x}_i(t)=\boldsymbol{x}_i(t-1)+s\left(\frac{\boldsymbol{x}_j(t-1)-\boldsymbol{x}_i(t-1)}{\|\boldsymbol{x}_j(t-1)-\boldsymbol{x}_i(t-1)\|}\right) \tag{11}$$

其中,s为萤火虫的步长。

萤火虫n_i在时刻t时的决策域$r_{\mathrm{d}}^i(t)$更新公式如式(12)所示:

$$r_{\mathrm{d}}^i(t)=\min\{r_{\mathrm{s}},\max\{0,r_{\mathrm{d}}^i(t-1),\varphi(g_{t-1}-|G_i(t-1)|)\}\} \tag{12}$$

其中,r_{s}表示萤火虫的感知域范围,φ表示决策域的更新率,g_{t-1}表示萤火虫在$t-1$时刻的邻域阈值,$G_i(t-1)$表示萤火虫n_i在$t-1$时刻的邻域。

基于人工萤火虫算法的任务调度算法的流程如算法3所示,首先把$UPGroup$中的任务按照负载值高低进行排序,把$UPGroup$中的任务设定为萤火虫,$DSGroup$区域设定为发光区域,在发光区域中寻找最优解并把任务调度到最优的资源节点上。

算法3 基于人工萤火虫优化的任务调度算法

输入:萤火虫集合$N=\{n_1,n_2,\cdots,n_n\}$、迭代次数M、过负载区域$UPGroup$、轻负载区域$LPGroup$、近饱和区域$NSGroup$、差饱和区域$DSGroup$。

输出:把$UPGroup$区域的任务调度给$DSGroup$中的

节点 DataNode。

1. 初始化算法中需要用到的参数;
2. **For** n_i in N
3. 初始化荧光素、初始位置和初始决策域;
4. **While** 当 $UPGroup$ 区域和 $DSGroup$ 区域都不空时,才有任务可以调度的空间 **do**
5. **For** n_i in N
6. Update $l_i(t)$、$G_i(t)$、$r_d^i(t)$/* 更新萤火虫 n_i 的荧光素值、邻域集合和决策域半径 */
7. **For** 邻域集合中的萤火虫 n_j
8. 根据式(10)更新萤火虫 n_i 向萤火虫 n_j 方向的转移概率;
9. **If** 萤火虫 n_i 向萤火虫 n_j 方向的转移概率大于此时最大值 max **then**
10. 更新最大值为此时的转移概率;
11. **End if**
12. **End for**
13. 选出最大值作为萤火虫 n_i 的移动速度方向 $\boldsymbol{v}$;
14. 使萤火虫 n_i 向下一次迭代速度方向 $\boldsymbol{v}$ 移动并更新萤火虫移动后的位置点;
15. **End for**
16. 更新萤火虫在 $t+1$ 时刻的决策域范围并求出全局最优点 $pbest$;
17. **For** 过负载区域 $UPGroup$ 中的任务 T_j
18. 把任务 T_j 调度到差饱和区域 $DSGroup$ 中选出的最优节点 $pbest$;
19. **End for**
20. **End while**
21. **End for**
22. 输出更新过后的过负载区域 $UPGroup$、轻负载区域 $LPGroup$、近饱和区域 $NSGroup$ 和差饱和区域 $DSGroup$ 集合.

算法过程中首先需要初始化萤火虫的步长 s、荧光素的消失率 ρ、荧光素的更新率 γ 和决策域的更新率 β(第 1 行);对于萤火虫集合中的每个萤火虫,初始化它们的荧光素为 l_0、初始决策域为 r_0 和初始位置为 $x_i(t)$(第 2、3 行);更新萤火虫的荧光素值、邻域集合和决策域半径(第 6 行);对于该萤火虫的邻域集合 $G_i(t)$,计算得到每个萤火虫 n_i 移向邻域集合 $G_i(t)$ 内每个个体 n_j 的概率 $p_{ij}(t)$,并选出概率最大的 n_j 方向作为萤火虫 n_i 的移动方向(第 8~14 行);更新位置和萤火虫 $t+1$ 时刻的决策域范围 $r_d^i(t+1)$,直到完成迭代次数 M,选出全局最优解,把萤火虫 n_i($UPGroup$ 中的任务 T_i)移到全局最优解的位置($DSGroup$ 中的资源节点 TM_j)(第 19 行);输出更新过后的过负载区域 $UPGroup$、轻负载区域 $LPGroup$、近饱和区域 $NSGroup$ 和差饱和区域 $DSGroup$ 集合(第 22 行)。

下面通过一个实例来描述基于人工萤火虫优化的任务调度算法的具体执行过程,如图 4 所示。

图 4a 为原本的区域分配情况以及任务和资源节点的对应调度情况,当 $UPGroup$ 和 $DSGroup$ 都不为空时,把过负载区域 $UPGroup$ 中的资源节点根据负载值由高到低进行排序,并把该资源节点中的任务按照优先级由高到低进行排序,对于

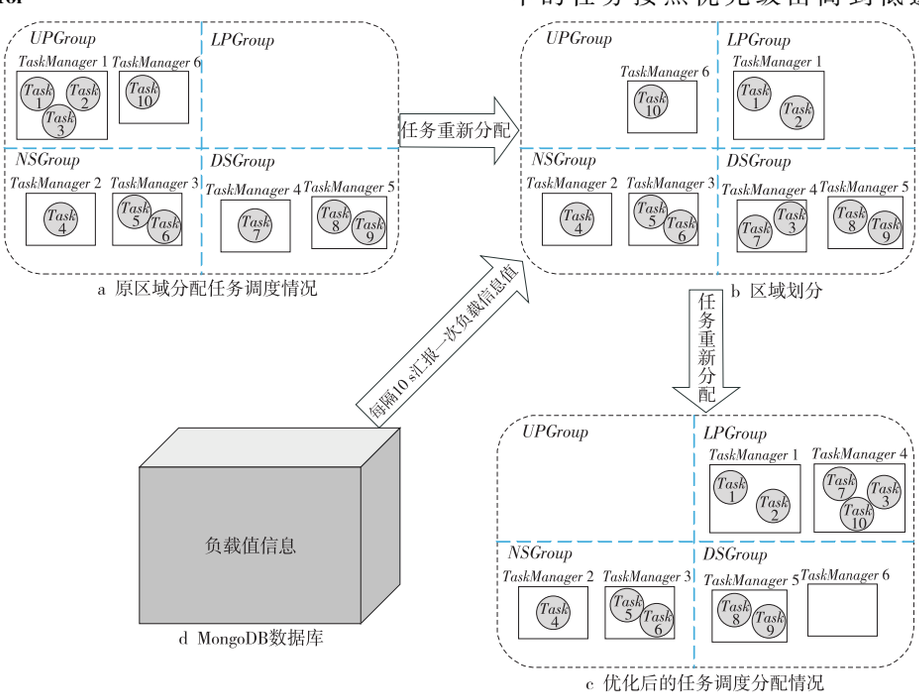

Figure 4 Execution process of task scheduling algorithm based on GSO algorithm

图 4 基于人工萤火虫算法的任务调度算法执行过程

UPGroup 中的节点对应的分配任务模拟为萤火虫，*DSGroup* 定义为萤火虫的区域移动范围。

在本次实例中 *UPGroup* 中资源节点为 *TaskManager*1 和 *TaskManager*6，它们的偏移量分别是 0.91 和 0.87，*TaskManager*1 中的任务 *Task*1、*Task*2、*Task*3 的优先级分别是低、低和高，*TaskManager*1 中的任务 *Task*3 的优先级分高；对于 *DSGroup* 中的 *TaskManager*4 和 *TaskManager*5 的偏移量分别是－0.58 和－0.31；*NSGroup* 中资源节点为 *TaskManager*2 和 *TaskManager*3，*LPGroup* 区域为空。因为 *UPGroup* 中任务重新调度的顺序为先按照资源节点中偏移量由高到低排序，对于同一资源节点中的任务再按照任务的优先级由高到低排序，因此 *UPGroup* 中第一个被调度的任务为 *TaskManager*1 中的 *Task*3，由于 *DSGroup* 中 *TaskManager*4 的偏移量低于 *TaskManager*5 的偏移量，因此 *TaskManager*1 中的 *Task*3 任务被调度到 *TaskManager*4 中。

此时系统实时性能发生改变，从 MongoDB 数据库中获取实时的资源节点负载值信息，并根据节点当前的任务队列数量进行重新分区。此时集群任务分配情况如图 4b 所示，*TaskManager*1 偏移量更新为 0.64，处于 *LPGroup* 区域，下一个被调度的任务为 *UPGroup* 中的 *TaskManager*6 的 *Task*10，此时 *DSGroup* 中的 *TaskManager*4 和 *TaskManager*5 的偏移量分别是－0.37 和－0.28，因此 *Task*10 被调度到 *TaskManager*4 中，并且资源节点的性能发生变化；此时集群重新划分后如图 4c 所示，*TaskManager*4 被划分到 *LPGroup*，*TaskManager*6 被划分到 *DSGroup* 区域，此时 *UPGroup* 为空，即最终的任务调度分配方案。

4 实验与分析

本文通过修改 Flink 1.8.0 的源码，实现了基于资源反馈的负载均衡任务调度算法 RFTS。本节设计并实施了一系列实验，基于 TPC-C 和 TPC-H 数据集，从执行时间和吞吐量 2 个方面对 Flink 默认的调度算法、公平调度算法、遗传算法和本文提出的 RFTS 算法进行对比实验，测试了本文提出的面向 Flink 系统的任务调度优化算法的实用性。

4.1 实验环境配置

实验所用环境为 4 个节点组成的分布式集群，包括 1 个主节点(Master)和 3 个从节点(Slave)，每台服务器的配置信息如表 1 所示。

Table 1 Hardware configuration

表 1 硬件配置

硬件	配置
处理器	Intel Xeon E5-2603 V4 *2 6 核心 1.7 GHz
内存	DDR4 内存(8 GB)*8
固态硬盘	固态硬盘*1，容量 400 GB
机械硬盘	6 TB 日立机械硬盘*4

搭建成由 4 台服务器组成的 Flink 集群，其中的 1 台 Master 为 Flink 集群中的 JobManager 节点，3 台 Slave 节点为 Flink 中的 TaskManager 节点，节点间通过千兆以太网连接，节点间的运行方式为 Standalone 模式。Flink 集群的软件及其版本如表 2 所示。

Table 2 Software configuration

表 2 软件配置

软件	配置
操作系统	CentOS 7
编程环境	Intel liJ IDEA 2018、Maven 3.5.3、Git
集成环境	Flink 版本 V1.0，Hadoop 版本 2.7.5
开发语言	Java
数据库	MongoDB 4.4.4

4.2 数据集

本文实验使用 TPC-C 和 TPC-H 数据集分别生成 6 个不同大小的数据集，测试程序将在这 12 个数据集上进行测试。数据集的来源和规模如表 3 所示。

Table 3 Dataset size

表 3 数据集规模

数据集	来源	大小/GB
*Dataset*1	TPC-C	2
*Dataset*2	TPC-C	5
*Dataset*3	TPC-C	8
*Dataset*4	TPC-C	11
*Dataset*5	TPC-C	15
*Dataset*6	TPC-C	18
*Dataset*7	TPC-H	2
*Dataset*8	TPC-H	5
*Dataset*9	TPC-H	8
*Dataset*10	TPC-H	11
*Dataset*11	TPC-H	15
*Dataset*12	TPC-H	18

下面分别介绍这 2 个数据集的数据特征：

TPC-C 是联机交易处理系统 OLTP(On-Line Transaction Processing)的规范,TPC-C 测试中使用的模型是一家大型商品批发销售公司,在不同地区中设有多个仓库,随着业务的增长,公司需要添加新的仓库,每个仓库有 10 个销售点,每个销售点为 3 000 个客户提供服务,每个销售订单对应 10 种产品,大约有 1%的产品显示缺货时,需要从其他区域的仓库中调运。整个 TPC-C 数据集由 9 张表组成,包括客户表、区域、订单表等等,产生的交易事务主要有 5 种,分别是新订单、支付操作、发货、订单状态查询和库存状态查询,该数据集可以通过命令指定生成数据集的大小。

TPC-H 是商品零售业决策支持系统的测试基准,测试系统中复杂查询的执行时间。它包含 8 个基本表,数据量可以设置为 1 GB～3 TB 不等,其基准测试包含 22 个查询,查询语句严格遵守 SQL-92 语法,并且不允许修改,主要指标为每个请求的响应时间,即提交任务后返回结果所花费的总时间。TPC-H 中数据量的大小对查询速度的影响很大,使用 SF 描述数据量,1SF 对应 1 GB 单位,并且人工设定的数据量只是 8 个表中的总数据量并不包含索引和临时表等空间占用情况,因此设定数据时需要预留更多的空间。对于同样规模大小的数据集,TPC-H 产生的数据类型比 TPC-C 产生的数据类型更多样,关系更复杂。

4.3 算法本身处理时间

本文分别在 TPC-C 和 TPC-H 不同规模的数据集上测试 RFTS 算法的处理时间,结果如表 4 所示,TPC-C 代表简单场景下的测试,TPC-H 代表复杂场景下的测试。随着数据集规模的增大,算法处理时间占任务整体执行时间的比例也增大了,这是因为处理任务的数据规模越大,出现负载不均的情况越多,需要迁移的任务急剧增多,因此调度算法所占整体执行时间的本身的比例也就越大。从表 4 可以看出,无论对于简单数据还是复杂数据而言,调度算法所占整体执行时间的比例都小于 1%。

其中,*Dataset*7～*Dataset*12 对应的数据是基于 TPC-H 数据集的测试结果,*Dataset*1～*Dataset*6 对应的数据是基于 TPC-C 数据集的测试结果。通过对比 *Dataset*1 & *Dataset*7,*Dataset*2 & *Dataset*8, *Dataset*3 & *Dataset*9, *Dataset*4 & *Dataset*10, *Dataset*5 & *Dataset*11, *Dataset*6 & *Dataset*12,可以发现基于 TPC-H 数据集的实验中任务整体执行时间和调度算法本身占用的时间都比基于 TPC-C 的要大,但是算法处理时间占任务整体执行时间的比例反而更小了。这是因为对于复杂数据场景,负载不均的情况更常见,RFTS 算法的调度算法本身增加的时间对于优化的系统执行效率来说影响更小,即优化的效果更好。实验结果表明,无论是 TPC-C 对应的简单环境还是 TPC-H 对应的复杂场景,算法本身处理时间占任务整体执行时间的比例都很小。

Table 4 Proportion of processing time of the RFTS algorithm

表 4 RFTS 算法的处理时间占比

数据集	数据规模/GB	RFTS 和 MMPSO 算法处理时间/s	任务整体执行时间/s	算法处理时间占任务整体执行时间的比例/%
*Dataset*1	2	0.318	41	0.778
*Dataset*2	5	1.128	133	0.848
*Dataset*3	8	1.533	176	0.871
*Dataset*4	11	1.975	218	0.906
*Dataset*5	15	3.319	349	0.951
*Dataset*6	18	4.117	428	0.962
*Dataset*7	2	0.438	56	0.782
*Dataset*8	5	1.391	176	0.790
*Dataset*9	8	1.925	240	0.802
*Dataset*10	11	2.421	291	0.831
*Dataset*11	15	4.064	453	0.897
*Dataset*12	18	5.274	554	0.952

4.4 执行时间对比分析

执行时间是指从任务提交到完成所需要的总时间。执行时间越少,代表系统处理任务的计算能力越强。本节对 RFTS 算法、公平调度算法(Fair)、遗传算法(Genetic)和 Flink 默认的轮询调度算法(Default)在任务整体执行时间上进行对比实验,分别基于 TPC-C 和 TPC-H 的 6 个数据集进行测试,作业并行度都设置为 12,测试用例为 WordCount。图 5 和图 6 分别为基于 TPC-C 和 TPC-H 数据集的执行时间对比图。

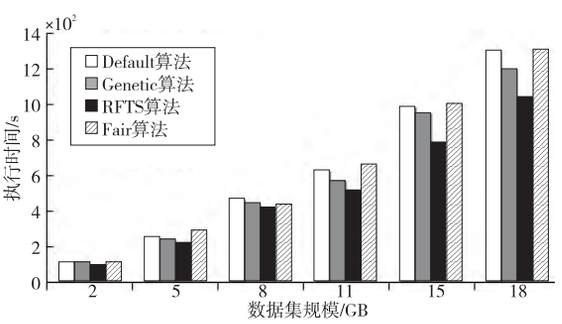

Figure 5 Comparison of execution time based on TPC-C dataset

图 5 基于 TPC-C 数据集的执行时间对比

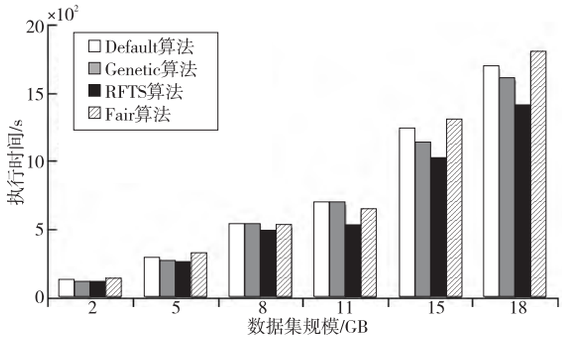

Figure 6 Comparison of execution time based on TPC-H dataset

图 6 基于 TPC-H 数据集的执行时间对比

从图 5 和图 6 可以看出,采用 Fair 调度算法与默认的轮询调度算法的执行效率相差不大;采用遗传算法 Genetic 的执行效率要优于采用 Fair 调度和默认的轮询调度算法的,优化效果约为 3.1%,在数据集为 11 GB 时优化效果最好;而采用本文提出的 RFTS 算法比采用 Flink 默认的轮询调度算法、Fair 调度算法和 Genetic 算法的执行时间都要短,且随着数据集规模的增大,执行时间的优化效果越好,且在数据集规模大小相同的情况下,图 6 对应的基于 TPC-H 数据集的执行时间优化效果优于图 5 对应的基于 TPC-C 数据集的执行时间优化效果。原因同上,这说明 RFTS 算法对于复杂的、大规模场景执行时间优化效果更好。但是,无论是在 TPC-C 对应的简单场景还是在 TPC-H 对应的复杂场景下,采用 RFTS 算法后的整体任务执行时间都要少于采用 Flink 默认的轮询任务调度算法的。在多种数据集的测试下最终求出采用 RFTS 算法时整体任务执行时间的平均优化效率为 6.3%。

4.5 吞吐量对比分析

吞吐量(throughput)是指系统单位时间内能够处理的数据量大小,代表了系统的负载能力。图 7 为 RFTS 算法、公平调度算法(Fair)、遗传算法(Genetic)和 Flink 默认的调度算法(Default)在不同并行度下的吞吐量对比分析图。分析图 7 可知,采用 Fair 算法时的吞吐量比采用 Flink 默认的轮询算法的低;采用 Genetic 算法时的吞吐量优于采用 Flink 默认的轮询算法和 Fair 算法的,在多种数据集测试下最终求出的采用 Genetic 算法的吞吐量平均优化效率为 3.9%;而采用 RFTS 算法相比采用 Flink 默认的轮询算法、Fair 算法和 Genetic 算法的吞吐量更大,并且随着并行度的增大,增加了同一时间内处理任务的机器数量,系统承受负载的能力增大,即系统单位时间内可以处理更多的数据。相比而言,使用 RFTS 算法吞吐量优化效果更好,这是因为使用 RFTS 算法会实时监控系统性能,并根据每个资源节点的负载承受能力,去调整任务队列中的任务数量,时刻保证集群负载均衡,大大提升了资源利用率以及单位时间内可以处理的数据量。在多种数据集测试下最终求出的采用 RFTS 算法的吞吐量平均优化效率为 11.7%。

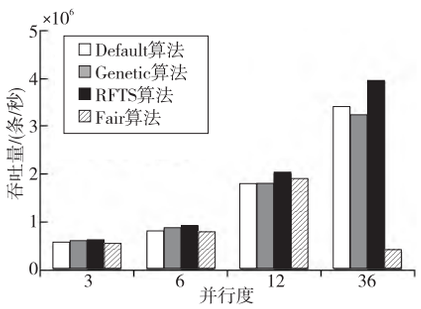

Figure 7 Comparison of throughput at different degrees of parallelism

图 7 不同并行度下吞吐量对比

5 结束语

Flink 作为现在主流的大数据计算引擎,在实时数据处理和离线数据处理上都表现出了良好的效果,然而 Flink 计算引擎中的任务调度还有许多待优化的空间,因此本文提出了基于资源反馈的负载均衡任务调度算法 RFTS。实验结果表明,本文提出的 RFTS 算法能够有效减少 Flink 计算引擎中任务的整体执行时间,增加吞吐量。

未来希望本文提出的 RFTS 算法可以应用于其它的大数据计算引擎中,并取得性能提升。

参考文献:

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

[2] Bhandarkar M. MapReduce programming with Apache Hadoop[C]//Proc of 2010 IEEE International Symposium on Parallel & Distributed Processing,2010:1.

[3] Yao Y,Wang Y F,Liu Z J. An algorithm used to improve task parallelization for directed acyclic graphs[C]//Proc of the 26th International Conference on Data Engineering,2010:238-240.

[4] Singh R,Kaur P J. Analyzing performance of Apache Tez and MapReduce with Hadoop multinode cluster on Amazon cloud[J]. Journal of Big Data,2016,3(1):1-10.

[5] Wang D W,Zhou F F,Li J M. Cloud-based parallel power flow calculation using resilient distributed datasets and direct-

ed acyclic graph[J]. Journal of Modern Power Systems and Clean Energy,2019,7(1):65-77.

[6] Zaharia M,Chowdhury M,Franklin M J,et al. Spark:Cluster computing with working sets[C]//Proc of the 2nd USENIX Conference on Hot Topics in Cloud Computing,2010:1-7.

[7] 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):28-38.

[8] TPC. TPC-H benchmark standard specification [EB/OL]. [2021-09-18]. http://www.tpc.org/tpch/.

[9] Moerkotte G,Scheufele W. Constructing optimal bushy processing trees for join queries is NP-hard:1996 Technical reports[R]. Mannheim:University of Mannheim,1996.

[10] Devipriya S,Ramesh C. Improved Max-Min heuristic model for task scheduling in cloud[C]//Proc of 2013 International Conference on Green Computing,Communication and Conservation of Energy,2013:883-888.

[11] Reda N M. An improved sufferage meta-task scheduling algorithm in grid computing systems[J]. International Journal of Advanced Research,2015,3(10):123-129.

[12] Teixeira M A,Guardieiro P R. Uplink scheduling algorithm with dynamic polling management in IEEE 802.16 broadband wirelss networks[C]//Proc of the 2010 ACM Symposium on Applied Computing,2010:601-602.

[13] Doulamis N D,Doulamis A D,Varvarigos E A,et al. Fair scheduling alogrithms in grids[J]. IEEE Transactions on Parallel and Distributed Systems,2007,18(1):1630-1648.

[14] Singh H,Bhasin A,Kaveri P R. QRAS:Efficient resource allocation for task scheduling in cloud computing[J]. SN Applied Sciences,2021,3:Article number:474.

[15] Todd D,Sen P. Distributed task scheduling and allocation using genetic algorithms[J]. Computers & Industrial Engineering,1999,37(1):47-50.

[16] Chen Q,Hou M. Research on gird task scheduling based on ant colony algorithm[J]. Advanced Materials Research,2010,129-131:1438-1443.

[17] Sivanandam S N, Visalakshi P. A new approach for task scheduling using elite particle swarm optimization[J]. Artificial Intelligent Systems and Machine Learning, 2009, 1(2):45-51.

[18] Shu W N,Zheng S J. A parallel genetic simulated annealing hybrid algorithm for task scheduling[J]. Wuhan University Journal of Natural Sciences,2006,11(5):1378-1382.

[19] Tsai J T,Fang J C,Chou J H. Optimized task scheduling and resource allocation on cloud computing environment using improved differential evolution algorithm[J]. Computers & Operations Research,2013,40(12):3045-3055.

作者简介:

李文佳(1996-),女,辽宁大连人,硕士,研究方向为大数据挖掘和并行计算。**E-mail**:2952781366@qq.com

LI Wen-jia, born in 1996, MS, her research interests include big data mining, and parallel computing.

史岚(1964-),女,辽宁沈阳人,硕士,副教授,研究方向为计算机体系结构和网络信息安全。**E-mail**: shilan@cse.neu.edu.cn

SHI Lan, born in 1964, MS, associate professor, her research interests include computer architecture, and network information security.

季航旭(1990-),男,辽宁沈阳人,博士生,研究方向为图嵌入和分布式计算。**E-mail**:406338743@qq.com

JI Hang-xu, born in 1990, PhD candidate, his research interests include graph embedding, and distributed computing.

罗意彭(1998-),男,辽宁鞍山人,研究方向为计算机网络工程与管理。**E-mail**:1073316546@qq.com

LUO Yi-peng, born in 1998, his research interest includes engineering & management of computer network.