软件学报 ISSN 1000-9825, CODEN RUXUEW
*Journal of Software*,2020,31(12):3981−3999 [doi: 10.13328/j.cnki.jos.005892]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

# 一种云环境中的动态细粒度资源调度方法*

周墨颂, 董小社, 陈 衡, 张兴军

(西安交通大学 电子与信息工程学院,陕西 西安 710049)

通讯作者: 陈衡, E-mail: hengchen@mail.xjtu.edu.cn

**摘 要**: 云计算平台中普遍采用固定资源量的粗粒度资源分配方式,由此会引起资源碎片、过度分配、低集群资源利用率等问题.针对此问题,提出一种细粒度资源调度方法,该方法根据相似任务运行时信息推测任务资源需求;将任务划分为若干执行阶段,分阶段匹配资源,从分配时间和分配资源量两方面细化资源分配粒度;资源匹配过程中,基于资源可压缩特性进一步提高资源利用率和性能;采用资源监控、策略调整、约束检查等机制保证资源使用效率和负载性能.在开源云资源管理平台中,基于细粒度资源调度方法实现了调度器.实验结果表明:细粒度资源调度方法可以在不丧失公平性且调度响应时间可接受的前提下,细化资源匹配的粒度,有效提高云计算平台资源利用率和性能.

**关键词**: 细粒度;调度;云计算;资源管理;平台优化

**中图法分类号**: TP316

中文引用格式: 周墨颂,董小社,陈衡,张兴军.一种云环境中的动态细粒度资源调度方法.软件学报,2020,31(12):3981−3999. http://www.jos.org.cn/1000-9825/5892.htm

英文引用格式: Zhou MS, Dong XS, Chen H, Zhang XJ. Dynamically fine-grained scheduling method in cloud environment. Ruan Jian Xue Bao/Journal of Software, 2020,31(12):3981−3999 (in Chinese). http://www.jos.org.cn/1000-9825/5892.htm

## Dynamically Fine-grained Scheduling Method in Cloud Environment

ZHOU Mo-Song, DONG Xiao-She, CHEN Heng, ZHANG Xing-Jun

(Faculty of Electronics and Information Engineering, Xi'an Jiaotong University, Xi'an 710049, China)

**Abstract**: The coarse-grained scheduling used in cloud computing platform allocates fixed quantity resources to tasks. However, this allocation can easily lead to problems such as resource fragmentation, over-commitment and inefficient resource utilization. This study proposes a dynamically fine-grained scheduling method to resolve those problems. This method estimates resource requirement of task according to similar tasks and divides tasks into execution stages according to the task requirement, and it also matches task resource requirement and available server resources by stages to refine two aspects of allocation granularity: allocation duration and allocation quantity. Furthermore, this method may compress resource allocation to further improve resource utilization and performance, and this method uses several mechanisms including runtime resource monitoring, allocation policy adjustments, and scheduling constraint checks to ensure resource utilization and performance of cloud computing platform. Based on this method, a scheduler has been implemented in the open source cloud computing platform Yarn. The test results show that the dynamically fine-grained scheduling method can resolve resource allocation problems by significantly improving resource utilization and performance with acceptable fairness and scheduling response times.

**Key words**: fine-grained; schedule; cloud computing; resource management; platform optimization

资源管理与作业调度是云计算资源管理平台中的关键点之一.随着云计算平台上负载的多样性和动态性日益增加,原有的资源管理与作业调度方式的有效性严重降低[1].

* 基金项目: 国家重点研发计划(2016YFB0200902); 国家自然科学基金(61572394)

Foundation item: National Key Research and Development Program of China (2016YFB0200902); National Natural Science Foundation of China (61572394)

收稿时间: 2017-10-16; 修改时间: 2018-06-20; 采用时间: 2019-09-24

高性能计算平台通常基于 core 分配资源[2],而云计算平台常常基于一种资源(通常为内存)或者将两种固定量的计算资源捆绑定义为 slot,并以 slot 作为资源分配的单位[3−7],这均属于粗粒度的资源分配方式.由于负载对资源的需求具有多样性[1,8−13],基于固定单位的资源分配方式很容易产生资源碎片而造成资源浪费.一些研究[14,15]按资源偏好将负载分为 CPU 密集型和 IO 密集型两种,并在调度中使不同资源偏好的负载一起执行,以减少资源碎片造成的浪费.还有一些研究采用动态调整 slot 数目来避免资源碎片,但是这并未有效解决问题,未经过严密计算而改变 slot 数目或者将负载分为若干类互补执行,极易造成不良的资源共享或资源过度分配.有些资源(比如 CPU)的不良共享会导致任务之间竞争资源,最终造成执行时间大幅增加[16].在数据中心中,超过 53%的落后任务是由不良共享引发的高资源利用率造成的,并且 4%~6%的非正常任务影响着 37%~49%的作业,造成作业完成时间的大幅延长[17].而有些资源(比如内存)的过度分配会直接导致任务失败或者服务器崩溃.

Yarn[18],Fuxi[19],Borg[20]等资源管理平台和 Apollo[11],Omega[12],Tetris[13],DRF[21],Carbyne[22]等调度算法基于资源申请分配固定量资源,以避免资源碎片、过度分配等问题.由于资源申请量通常由人为指定,资源申请量与实际使用量之间存在差异[1].另外,任务资源使用量波动很大,不会始终保持在峰值.使用任务最大资源使用量作为申请量时,资源申请量与实际使用量之间依然存在差异[8].因此,资源管理平台或者调度器按照资源申请量进行分配时,资源碎片依然存在.基于资源申请的分配方式很难取得高资源利用率,该方式一定程度上限制了集群计算资源利用率[8].举例来说,有任务 *A* 和 *B*,其执行阶段及资源需求如图 1 所示.假设服务器配置为 4 核 CPU 和 8GB 内存,任务 *A*1,*A*2,*B*1 依次到达,则基于资源申请的调度结果如图 2 所示,其中,阴影部分为资源碎片.当任务 *A*1,*A*2 分配计算资源之后,CPU 资源剩余 2 核,不满足 *B*1 的资源需求.因此,任务 *B*1 只能等待计算资源,最终负载在 7 单位时间内完成.

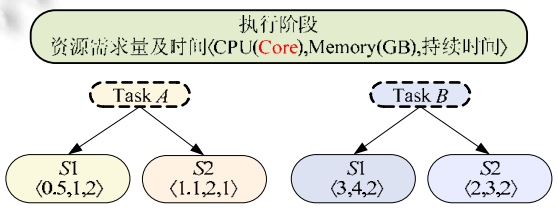


Fig.1 Resource requirements and durations of tasks execution stages

图 1 任务资源阶段的资源需求及持续时间

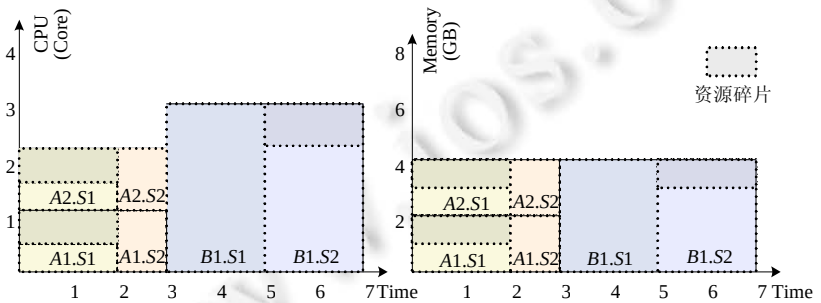


Fig.2 Results of request-based scheduling

图 2 基于资源申请的调度结果

针对上述云计算平台中,资源管理和作业调度中存在的问题,本文提出一种细粒度资源调度方法(fine-grained method,简称 FGM).该方法分阶段匹配资源需求和可用资源,细化资源匹配粒度,并在分配中根据资源特性压缩资源需求,进一步提高资源利用率和负载性能.对于前文例子中的负载,细粒度资源调度的结果如图 3 所示.FGM 推测任务的资源需求,并将任务划分为两个执行阶段;之后,FGM 按照任务各执行阶段的资源需求及持续时间分别匹配资源.FGM 在向 *A*1 和 *A*2 分配其实际所需资源量之后,剩余 CPU 资源不满足 *B*1 的资源需求.

经过严密计算之后,FGM 压缩 *A*1,*A*2 和 *B*1 的 CPU 资源分配,使 3 个任务并行执行.*A*1 和 *A*2 结束之后,*B*1 将解除压缩分配状态,获得足够资源.轻微的资源压缩以完成时间为代价,提高了服务器的资源使用率和任务并行数量,因此,负载任务第 2 阶段的完成时间延长而负载整体完成时间缩短.FGM 调度的负载理论上可在 4.05 时间左右完成,相比图 2 中的调度结果,在资源利用率和负载完成时间上均有很大提升.

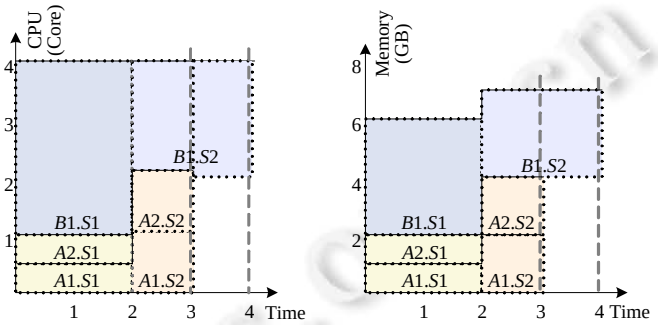


Fig.3 Results of fine-grained scheduling

图 3 细粒度资源调度结果

本文方法的创新及特色之处在于:

- 分执行阶段按需求匹配资源,细化资源量和时间的匹配粒度,以避免资源碎片和过度分配.任务资源需求根据相似任务运行时信息推测得到,执行阶段依据任务资源需求划分得到;
- 根据计算资源特性,将资源分为可压缩资源和不可压缩资源两类.必要时对任务所需的可压缩资源进行一定程度的压缩分配,以提高计算资源使用率和任务并行数目;
- 通过运行时资源监控、分配策略调整、调度约束检查等机制调节资源压缩和细粒度调度,保证资源使用率和性能.

本文所提的细粒度资源调度方法可以有效解决负载资源需求具有多样性和动态性环境中因现有固定、粗粒度的资源分配方式引发的资源碎片和过度分配问题.本文在细粒度资源调度方法的基础上实现了细粒度资源调度器.多种环境下的测试结果显示:细粒度调度器可以在不影响公平性且调度响应时间可接受的前提下,解决资源碎片、过度分配等问题,提高资源利用率和性能.

## 1 相关工作

近年来,云计算领域发展迅速,关于云计算资源管理平台及调度算法的研究越来越多.

Hadoop[7]是采用集中式调度架构的开源云计算平台,它将计算节点上的资源均分到固定数目的 slot 中,并将 slot 分配给 MapReduce[23]任务.云计算负载的资源需求具有多样性[1,8−13],而 Hadoop 忽略资源需求的多样性,因此,任务执行过程中很容易产生资源碎片造成资源浪费.

Yarn[18]是从 Hadoop 发展而来的云计算资源管理平台,它采用双层调度架构解耦资源管理和应用逻辑,支持多种应用负载.Fuxi[19]是阿里巴巴公司提出的资源管理、作业调度系统,它通过增量式资源管理、错误恢复等机制,解决了大规模集群调度可扩展性和错误容忍等问题.Fuxi 在资源管理中定义了调度单元,资源申请以调度单元为单位进行.Borg[20]是谷歌公司用于管理由上万节点集群的管理系统,Borg 基于资源申请进行调度,用户分别以千分之一核、字节为单位申请 CPU、内存和磁盘资源.但是,Borg 在分配资源时以 0.5 核和 1GB 内存为粒度,将用户请求的资源量向上取整决定实际分配量.本文和 Borg 均将资源分为了可压缩和不可压缩两种,但是 Borg 只将该特性应用在了资源回收方式的决策中,而并未应用在资源分配中.

Yarn,Fuxi 和 Borg 均采用基于资源请求的机制决定资源分配量,但是分配量在任务执行期间不再变化.由于资源申请量通常由人为指定或采用资源最大使用量[1],而任务资源使用不会时刻保持在峰值,因此资源申请量

与实际使用量之间存在差异,这一定程度上限制了集群资源利用率[8].

Mesos[24]是由加州大学伯克利分校提出的支持多种应用的资源管理平台,它采用基于资源邀约的机制,允许应用框架决定自身的资源使用量.然而通常情况下,应用编程框架很难清楚地知道每个任务的资源需求情况.因此,资源邀约机制与资源请求的机制有着相似的问题.本文提出的方法分阶段精细匹配资源需求与分配量,该方法可以在上述任意平台中实现,用于改变资源调度粒度,减少资源碎片,避免过度分配等问题.

云计算资源管理和作业调度算法相关的研究有很多,但是各研究的侧重点有很大差异.Capacity[3]由 Yahoo 公司开发的调度器,它以队列为单位划分资源,以实现多用户高效共享集群计算资源.Tetris[13]针对负载多维需求引发的资源碎片和过度分配问题,分析历史执行信息获取作业资源需求,将调度问题抽象为多维背包问题,以提高集群的资源利用率.Tetris 假定任务的资源需求是恒定的,而细粒度匹配方法分阶段匹配,且在匹配过程中考虑了资源特性,因此可以进一步提高集群资源利用率.

Delay[5]提出将任务调度到数据所在服务器进行计算,以避免数据网络传输消耗的时间并提高性能.Quincy[25]是微软公司提出的高效灵活的分布式作业调度框架,它将调度问题映射为有向无环图,将数据本地性、公平性等信息编码成边的权值,并在调度中使用最小流选择代价最小的任务进行调度.当集群规模巨大时,Quincy 的调度延迟会大幅增加.Firmament[26]采用近似、增量流等多种方式对 Quincy 进行了最小流优化,解决了 Quincy 调度的扩展性问题.Delay,Quincy 和 Firmament 非常关注数据本地性,本文细粒度调度中使用了与 Delay 相同数据本地性机制.但是,上述调度器在资源分配时并没有改变资源分配的粒度,而本文关注资源分配的粒度,细粒度匹配方法也可以应用在上述调度器的资源匹配中,以提高资源分配的粒度.

LATE[6]与 Mantri[27]通过重新启动任务等方式解决个别落后任务阻碍作业完成的问题.本文调度器中并没有集中式的推测执行机制,而将推测执行策略交由各应用负责.

Apollo[11]是微软公司提出的高度可扩展的协同调度框架,它根据相似任务推测任务的完成时间,并以此为依据构建资源可用性表,最终使用资源可用性表分布式的做出调度决策.Omega[12]是谷歌公司提出的基于乐观锁[28]的共享状态调度器,它可以大幅提高调度决策的并发度.Apollo 和 Omega 可以提高决策质量和调度的可扩展性,但是分布式调度决策对公平性保证是一个挑战.

Fair[4]公平调度算法解决了多用户集群中计算资源共享的公平性问题,它通过限定用户或作业队列的最小份额和公平份额,避免作业饥饿.DRF[21]提出了多种类型资源下的公平策略,进一步提高了用户间的公平性.Choosy[29]扩展了最大最小公平算法,在具有调度约束条件的情况下实现公平性.Carbyne[22]放弃了瞬时公平性而只保持长时间总体公平,它将资源碎片优先分配给快完成的作业,提高资源使用效率及性能.细粒度调度器与 Carbyne 的目的相似,均是为了提高集群资源利用率和整体性能.但是本文中没有提出新的公平策略,仅使用 DRF 作为默认公平策略.它仅通过提高资源分配粒度,并在分配中考虑资源特征压缩资源需求等方式提高资源利用率.此外,细粒度调度器为各种资源公平行策略的实现和应用提供了扩展支持,放弃瞬时公平性的策略可以在 FGM 中实现,达到共同优化.

本文研究侧重于以更细的粒度匹配资源的供应与需求,并在必要时,根据资源特性压缩资源需求,以提高资源利用率和负载性能.因此,本文提出的方法与上述很多方法无互斥关系,多种方式可以协同工作,做出更优异的资源管理和作业调度决策.例如,文献[30]提出一种具有启动时间感知的虚拟机分配策略及一种粗粒度资源调度算法,在任务时效性和节能之间进行权衡.如果该文献采用本文的细粒度资源匹配方式,则可以提高资源供需匹配程度并增加虚拟机并行数量,进一步提高资源使用效率和任务时效性.

任务资源需求及其他信息的量化主要有两种思路:使用历史信息推测[13,31−36]和利用相似任务推测[11,37−39].两种方式均可以达到推测资源需求的目的,本文采用相似任务推测相关信息.相比已有的研究工作,本文使用的推测方法有以下不同:首先,本文推测时使用迭代更新的方式,减轻数据抖动的影响,使推测曲线平滑;第二,细粒度匹配中,通过压缩资源需求进一步提升资源利用率,这造成了运行时信息的准确性发生改变,本文推测中,根据资源需求压缩情况动态调整迭代更新的权重,保证推测准确性.

## 2 细粒度资源调度方法

细粒度资源调度方法(fine-grained method,简称 FGM)根据相似任务运行时信息推测任务的资源需求及持续时间,并依据资源需求将任务划分为若干执行阶段;分阶段按需求匹配资源和服务器可用计算资源,避免资源碎片和过度分配等问题,提高资源利用率;根据计算资源的特性,将资源抽象为可压缩资源和不可压缩资源两类,必要时,在不影响负载完成的前提下,一定程度压缩资源分配,进一步提高资源利用率和负载性能;在运行时感知服务器上计算资源的使用情况,并动态调整调度策略,保证资源平台高效运行.

FGM 的细粒度体现在两个方面:分配时间粒度方面,资源匹配以任务执行阶段为粒度进行,而不是以任务的整个执行过程或者某个固定长度的时间为粒度;分配资源量粒度方面,任务执行阶段的资源分配以该阶段资源需求量为依据,而不是以任务最大资源使用量、人为指定申请量、计算核心及 slot 等进行分配.

### 2.1 计算资源特征抽象

任务在执行过程中需要多种不同的计算资源,每种计算资源的特性有所差异.对于有些计算资源,如果资源供给量少于任务需求量,任务的完成时间会被延长,但不影响任务的顺利完成.例如,任务需要 CPU 的所有时间片进行计算,但是分配到了 CPU 时间片的一半.此时任务仍然可以正常完成,只是任务完成时间会增加一倍.对于另外一些计算资源,如果资源供给量少于资源需求量,任务则无法正常完成.例如,任务需要 1GB 内存空间,如果分配量少于 1GB,则任务失败.FGM 依据计算资源特性将计算资源抽象为可压缩资源和不可压缩资源两种类型,并定义资源压缩率来衡量计算资源被压缩的程度.

**定义 1(可压缩计算资源)**. 在任务执行过程中,如果任务分配到的某种资源少于任务对该资源的需求量,任务可以通过延长执行时间正常完成,则该资源为可压缩资源;否则为不可压缩资源.

**定义 2(计算资源压缩率)**. 假设在一段时间内资源的需求量为 $R_r$,资源分配量为 $R_u$.如果资源为可压缩资源,则该段时间内压缩率 $r_c$ 按公式(1)计算;如果资源为不可压缩资源,则其压缩率 $r_c$ 始终为 0:

$$r_c=\begin{cases}(R_r-R_u)/R_r, & R_r>R_u\\ 0, & R_r\leqslant R_u\end{cases} \tag{1}$$

可压缩计算资源性能变化示意图如图 4 所示,图中的计算资源性能变化可分为两个阶段.

- 第 1 阶段中,计算资源需求小于 100%,资源利用率等于资源需求.此阶段内,计算资源的性能随资源需求增加而上升;
- 第 2 阶段时,资源需求大于资源总量,资源利用率等于 100%.此时,各任务的资源需求被压缩,服务器上任务间资源竞争加剧,性能开始大幅度下降.

根据测试所得的可压缩计算资源性能变化数据,FGM 中可压缩计算资源的最大压缩率限制为 10%.当方法所处的场景发生改变时,应当重新就行测试,并根据测试结果调整最大压缩率限制值.

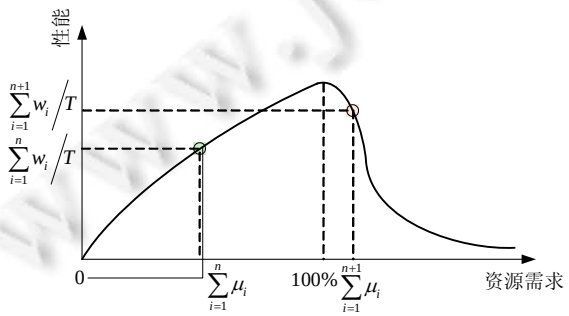


Fig.4 Performance change of compressible resource

图 4 可压缩计算资源性能变化图

资源压缩的目的是:在可接受范围内延长任务的完成时间,从而换取更高的资源利用率和任务并行数量,最

终提高整体性能.因此,在运行时根据多种因素动态调整最大压缩率.资源压缩应满足以下约束:

$$\begin{cases}\sum_{i=1}^{n}\mu_i<1\\ \sum_{i=1}^{n+1}\mu_i>1\\ \frac{\sum_{i=1}^{n}w_i}{T}<\frac{\sum_{i=1}^{n+1}w_i}{T'}\end{cases} \tag{2}$$

其中,服务器上某种资源总量为 1,第 $i$ 个任务占用该资源 $\mu_i$;在 $T$ 时间内完成的工作量为 $w_i$;$T'$为资源压缩后实际完成时间.

对任务 $m$ 而言,完成工作量 $w_m$ 所使用的资源总量一定,因此有等式关系(3):

$$T\times\mu_m=T_t'\times\mu_m\times(1-r_c) \tag{3}$$

其中,$T_t'$ 为资源压缩时所需时间,$r_c$ 为资源压缩率.

由等式(3)可以推导出 $T_t'$ 计算公式如等式(4)所示:

$$T_t'=\frac{T}{1-r_c} \tag{4}$$

实际运行中,由于资源竞争的原因,资源压缩后实际完成时间 $T'>T_t'$.$T'$计算公式如等式(5)所示:

$$T'=(1+p)\times T_t'=(1+p)\times\frac{T}{1-r_c} \tag{5}$$

其中,$p$ 代表资源压缩引起的性能变化.

将等式(5)代入不等式 $\frac{\sum_{i=1}^{n}w_i}{T}<\frac{\sum_{i=1}^{n+1}w_i}{T'}$ 中,可得到 $T$ 时间内最大压缩率约束,如不等式(6)所示:

$$r_c<\frac{w_{n+1}-\sum_{i=1}^{n}w_i\times p}{\sum_{i=1}^{n+1}w_i} \tag{6}$$

由于最大压缩率为非负数,且应小于最大压缩率限制值,FGM 中,通过不等式(7)分析一段时间内某种资源的最大压缩率,以确保资源压缩之后的资源利用率和性能相较无压缩情况有所提高:

$$r_c=\min\left(\max\left(limitation,\frac{w_{n+1}-\sum_{i=1}^{n}w_i\times p}{\sum_{i=1}^{n+1}w_i}\right),10\%\right) \tag{7}$$

公式(7)中,*limitation* 为可压缩资源的最大压缩率限制.

### 2.2 资源需求运行时推测

FGM 通过细化资源分配的粒度,向任务分配时刻符合资源需求的资源量,避免产生资源碎片,提高资源利用率.因此,准确量化任务各时刻的各种资源需求量及持续时间是 FGM 的关键点之一.云计算中的应用大多为大规模并行而设计,因此,负载中普遍存在执行逻辑相同的任务,这些任务具有相同的资源使用行为.FGM 在执行逻辑相同任务的基础上定义相似任务,并根据相似任务的运行时信息推测任务的各种资源需求及持续时间.由于各种推测方法均存在局限性,当应用场景发生改变时,推测方法也应当随场景及负载而调整,其他可选推测方法包括:沙盒执行、数学建模、静态剖析、历史信息等.

**定义 3(相似任务)**. 假设任务执行逻辑为 $L$,处理数据量为 $D$,如果任务 $i$ 和 $j$ 存在关系 $L_i=L_j\wedge D_i=D_j$,则任务 $i$ 和 $j$ 互为相似任务.

定义 3 只约束了相似任务的执行逻辑和输入数据量,而假设需要处理的信息在输入数据中均匀分布.其原因有 3 个方面:首先,不同执行逻辑任务的资源使用行为很可能具有明显差异,因此不具有比较性,无法应用于任务资源需求推测中;其次,使用执行逻辑相同而输入数据量不同的任务相互推测资源需求时,得出的资源需求结果的准确性会受到影响,最终影响 FGM 方法的效率;最后,需要处理的信息在输入数据中通常均匀分布,而在处理完成之前分布无法确定.

FGM 推测任务在每个进度上对各种资源的需求量以及持续时间,某个进度任务 CPU、内存资源需求量及持续时间的迭代推测公式如(8)所示:

$$\alpha_n=\begin{cases} average(\beta_n,\beta_{n+1},\beta_{n+2}), & n=1 \\ e^{\min(Th_r\times r_c-5/60,0)}\times\alpha_{n-1}+(1-e^{\min(Th_r\times r_c-5/60,0)})\times\beta_{n+2}, & n>1 \end{cases} \tag{8}$$

公式(8)中,$\alpha_n$ 为第 $n$ 次推测结果,$\beta_n$ 为第 $n$ 次的任务资源信息,$r_c$ 为信息采集时的资源压缩率,$Th_r$ 为最大压缩率限制系数.公式(8)参考了 Linux 系统资源利用率的计算公式,这样设计可以避免数据抖动影响推测结果.FGM 在分配资源中可能对资源需求进行压缩,这会导致相似任务运行时信息的准确性下降.公式(8)在推测时,使用资源压缩率动态调整迭代计算中新信息的权重来保证推测结果的准确性.当资源压缩率大于 $5/(60\times Th_r)$时,推测中认为该信息不具有参考性,在迭代中将该信息的权重降为 0.

对于推测得出的任务在各进度的资源需求量及持续时间,FGM 依据 CPU 和内存空间资源需求将任务划分若干执行阶段,执行阶段的资源需求为阶段内各进度资源需求的最大值.FGM 以一定的周期更新阶段划分,划分算法流程图如图 5 所示,图中的 $P$,$P_s$,$P_e$ 分别代表任务进度、阶段开始进度和结束进度,$C_{\max}$,$C_{\min}$,$M_{\max}$ 和 $M_{\min}$ 分别表示阶段内 CPU、内存资源需求的最大值和最小值.当阶段内 CPU 或者内存资源需求量的波动达到一定幅度且阶段长度达到要求时,将[$P_s$,$P_e$]划分为一个执行阶段,直到任务完成.

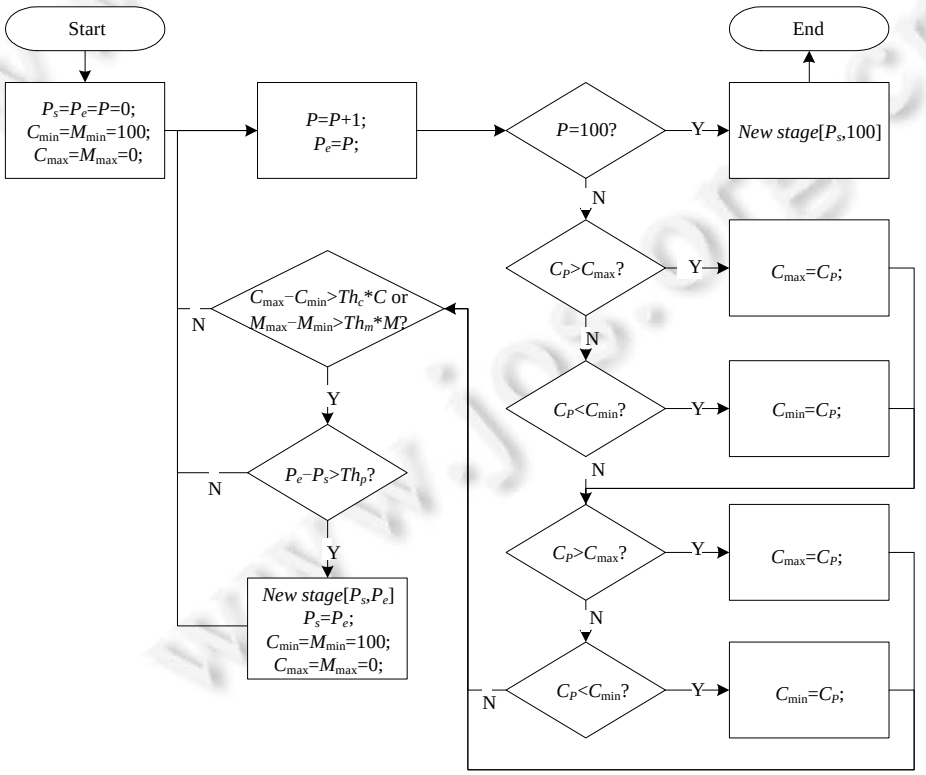


Fig.5 Flow chart of task execution stage division

图 5 任务执行阶段划分算法流程图

### 2.3 细粒度资源调度算法

本文将一些资源相关的变量表示为元组形式,以精简算法的表达.等式(9)所示的元组 $\vec{R}$ 代表一组有序数值,其中,$R^1$ 为第 1 个数值,依此类推.

$$\vec{R}=\langle R^1,...,R^n\rangle \tag{9}$$

假设有元组 $\vec{R}_a=\langle R_a^1,...,R_a^n\rangle$ 和 $\vec{R}_b=\langle R_b^1,...,R_b^n\rangle$,则两元组相加、相乘分别按公式(10)和公式(11)计算,元组与常数 $c$ 相乘按公式(12)计算:

$$\vec{R}_a+\vec{R}_b=\langle R_a^1+R_b^1,...,R_a^n+R_b^n\rangle \tag{10}$$

$$\vec{R}_a\times\vec{R}_b=\langle R_a^1\times R_b^1,...,R_a^n\times R_b^n\rangle \tag{11}$$

$$c\times\vec{R}_a=\langle c\times R_a^1,...,c\times R_a^n\rangle \tag{12}$$

两元组间的小于关系依据公式(13)判定,即 $\vec{R}_a$ 中有任意元素小于 $\vec{R}_b$ 中相应位置上的元素,则 $\vec{R}_a<\vec{R}_b$ 关系为真:

$$(\exists i)(0<i<n\wedge i\in Z\wedge R_a^i<R_b^i\rightarrow\vec{R}_a<\vec{R}_b) \tag{13}$$

服务器上出现空闲计算资源时,细粒度资源调度算法开始筛选资源申请进行调度,见算法 1.

**算法 1**. Fine-grain Scheduling.

Input:$T_s$:待调度任务集合;

$P_a$:服务器调度策略;

$\vec{I}$ :服务器可用资源元组;

$\vec{N}$ :服务器资源总量元组;

```
1.  if  I < Th × N  or isUnschedulable(P_a) then
2.      return;                                  //不可分配,直接返回
3.  end if
4.  sort(T_s,P_f);                               //按照公平性策略排序待调度集合
5.  for t in T_s do
6.      check data locality of task t;
7.      match←true;
8.      if unpredictable(t) then                 //任务 t 为服务或暂无足够推测信息
9.          match←allocRequestQuantity(·);
10.     else if P_a==FineGrained then
11.         match←fineGrainedMatching(·);        //使用算法 2 所示的细粒度匹配算法
12.     else
13.         If  A_x < M_x  then
14.             match←false;
15.         end if
16.     end if
17.     if match and checkQoS(·) then
18.         allocate(t);                         //分配计算资源给任务 t
19.         if the resource is used up then
20.             break;
21.         end if
22.     end if
23. end for
```

调度中,算法首先检查服务器空闲资源和该服务器调度策略,如果不可调度,则结束调度(第 1 行~第 3 行).服务器空闲资源元组内各项资源值 $r$ 按照公式(14)计算:

$$r=\sum_{i=1}^{n}(r_i\times t_i)\Big/T \tag{14}$$

其中,$r_i$ 为第 $i$ 次采样的资源量,$t_i$ 为第 $i$ 次采样的持续时间,$T$ 为采样计算的总时间.

如果满足调度条件,则算法按照公平性策略 $P_f$ 排序待调度集合 $T_s$,以保证资源分配公平性(第 4 行).之后,算法遍历待调度集合 $T_s$,检查数据本地性,根据服务器资源调度策略和资源需求推测情况,分别按照不同匹配策略进行调度.如果推测信息不足或需求不可推测,则算法按照资源申请量进行资源匹配(第 8 行、第 9 行);如果服务器调度策略为细粒度,则使用细粒度资源匹配算法匹配资源需求和可用资源(第 11 行、第 12 行),具体匹配算法如算法 2 所示;如果服务器不支持细粒度匹配,则按照粗粒度策略进行资源匹配(第 12 行~第 16 行).在粗粒度匹配中,资源需求元组 $\vec{M}_x$ 中包含所需 CPU、内存资源的最大值,而可用资源元组 $\vec{A}_x$ 包含服务器 CPU、内存资源在[0,$T$]区间(以当前时刻为 0 点,$T$ 为任务持续时间)内的最小可用量.如果 $\vec{A}_x<\vec{M}_x$ 比较关系成立,即有资源需求未得到满足,则匹配失败.如果资源匹配成功,算法会调用 *checkQoS*(·)检查该分配决策是否影响服务器上运行中任务的服务质量(第 19 行).如果分配决策通过所有检查,则算法分配资源,并检查是否剩余足够资源进入下一轮调度(第 20 行~第 23 行).

目前,细粒度资源调度算法中,服务质量检查默认检查任务截止时间约束,该检查只在服务器存在资源压缩时进行.截止时间检查时,假设本轮资源分配已经生效,并在此基础上计算分析服务器上所有运行中任务是否能在截止时间内完成,以保证服务质量.在资源压缩情况下,任务的理论持续时间 $D_t$ 可以按公式(15)计算:

$$D_t=\sum_{i=1}^{n}(D_i/(1-r_{ci})) \tag{15}$$

其中,$D_i$ 为不压缩资源需求时任务第 $i$ 段的持续时间,$r_{ci}$ 为任务第 $i$ 段在运行中各种可压缩资源压缩率的最大值.

细粒度资源匹配算法如算法 2 所示,算法以执行阶段为单位匹配资源需求与可用资源,并在匹配中应用资源可压缩特性.算法遍历任务的所有执行阶段进行匹配,如果任务所有阶段的各种资源需求均得到满足,则匹配成功(第 4 行~第 20 行).细粒度匹配算法在匹配中区分计算可用资源中的可压缩资源 $\vec{n}_c$ 和不可压缩资源 $\vec{n}_n$:对于不可压缩资源,算法直接比较可用资源元组与需求元组;而对于可压缩资源,算法将可用资源元组加上总资源量元组 $\vec{N}$ 与该阶段允许的最大压缩率元组 $\vec{r}_c$ 的乘积之后,再与可压缩资源需求元组 $\vec{s}_c$ 进行比较.服务器某个阶段允许的某种资源最大压缩率,依据该服务器该阶段已分配任务和当前匹配任务相关信息,通过等式(7)分析得到,其值还受到服务器调度策略的影响.

**算法 2**. Fine-grain Matching.

Input:$n_s$:服务器可用资源阶段集合;

$t_s$:推测得出的任务 $t$ 执行阶段集合;

$\vec{N}$:服务器资源总量元组;

Output:匹配结果.

1. $n$←the first stage in $n_s$;
2. $\vec{n}_c$ ←the tuple of compressible resource of the stage $n$;
3. $\vec{n}_n$ ←the tuple of non-compressible resource of the stage $n$;
4. **for** $s$ in $t_s$ **do**
5. $\vec{s}_c$ ←the tuple of compressible resource of the stage $s$;
6. $\vec{s}_n$ ←the tuple of non-compressible resource of the stage $s$;
7. **while** *n.startTime*<*s.endTime* **do**
8. **if** $\vec{n}_c+\vec{r}_c\times\vec{N}<\vec{s}_c$ **or** $\vec{n}_n<\vec{s}_n$ **then**
9. **return** false; //匹配失败

10. **end if**
11. **if** *n.endTime*≤*s.endTime* **then**
12. *n*←the next stage in $n_s$;
13. $\vec{n}_c$ ←the tuple of compressible resource of the stage *n*;
14. $\vec{n}_n$ ←the tuple of non-compressible resource of the stage *n*;
15. **else**
16. break;
17. **end if**
18. **end while**
19. **end for**
20. **return** true;

### 2.4 自适应策略调节

任务执行中的资源使用行为受多方面因素影响,因此,资源推测使用值与实际使用值之间存在不可避免的偏差.FGM 定义资源推测符合度来衡量服务器上某种资源推测使用量与实际使用量之间的偏差程度.

**定义 4(资源推测符合度)**. 假设 *T* 时间分为 *n* 段,第 *i* 段某种资源实际使用量为$\mu_i$,资源推测使用量为$\alpha_i$,阶段的持续时间为 $t_i$,则该资源推测符合度 *F* 以公式(16)计算:

$$F=\begin{cases}\left(\sum_{i=1}^{n}((\mu_i-\alpha_i)^2\times t_i)\right)\Big/T, & \mu>\alpha \text{ and } I<Th\times N\\ 0, & \mu\leqslant\alpha \text{ or } I\geqslant Th\times N\end{cases} \tag{16}$$

其中,$\mu$为实际资源使用量,$\alpha$为推测资源使用量,*I* 为资源的空闲量,*Th* 为资源的空闲阈值,*N* 为该资源总量.$\mu$,$\alpha$和 *I* 均按照公式(14)计算.

根据该定义,仅当资源的空闲量小于一定阈值,且资源的实际使用量大于推测使用量时,资源推测符合度才可能不为 0.这样设计的原因在于:推测值趋于准确需要过程,在推测值大于等于实际值或者空闲资源量充足的情况下,并不会产生严重的运行时后果,此时应该允许推测机制自行迭代调整推测值.

FGM 保存服务器各种资源最近几次资源推测符合度值,并在运行时根据服务器各资源的推测符合度以及服务器上运行中任务完成时间非正常增加程度等因素调整服务器调度策略,以减轻由资源需求推测偏差、资源压缩错估等引起的不当调度决策的影响.具体调度策略调整算法见算法 3.

**算法 3**. Policy Adjustment.

Input:$S_s$:服务器集合;

1. **for** *s* in $S_s$ **do**
2. **if** *averageConformity*(*s*)>$Th_{\text{sa}}$ **then**
3. *s.policy=stopAllocation*;
4. **if** *isDegradation*(*s*) **then**
5. recycle the resource of *s*;
6. **end if**
7. **else if** *averageConformity*(*s*)>$Th_{cg}$ **then**
8. *s.policy=CoarseGrained*;
9. **else if** *averageConformity*(*s*)>$Th_{\text{fg}}$ **then**
10. *s.policy=FineGrained*;
11. *s.compressible*=false;
12. **else**
13. *s.policy=FineGrained*;

14. *s.compressible*=true;
15. **end if**
16. **end for**

算法遍历服务器集合中所有服务器,根据服务器上问题的严重程度,依次调整服务器的调度策略:如果服务器近期平均资源符合度大于阈值 $Th_{sa}$,则停止分配该服务器资源,并根据该服务器上任务完成情况采取进一步措施(第 2 行~第 6 行);如果服务器近期平均资源符合度大于阈值 $Th_{cg}$ 而小于 $Th_{sa}$,则将该服务器调度策略调整为粗粒度匹配(第 7 行、第 8 行);如果服务器近期平均资源符合度大于阈值 $Th_{fg}$ 而小于 $Th_{cg}$,则将该服务器调度策略调整为不可压缩的细粒度调度(第 9 行~第 11 行);如果服务器近期平均资源符合度小于阈值 $Th_{fg}$,则将该服务器调度策略调整为细粒度调度(第 12 行~第 15 行).

## 3 细粒度调度器架构与实现

本文基于细粒度资源调度方法,在 Yarn 平台中设计并实现了细粒度资源调度器,其架构图如图 6 所示.图中的调度模块为细粒度调度器的核心,负责与 Resource Manager 通信,并应用调度算法协调各个调度模块,完成整个调度过程.为了保证调度的高效,本文将信息处理、需求推测、约束检查、策略管理和压缩管理等功能从调度模块中抽离,设计为独立的模块.信息推测模块负责处理、存储任务及服务器节点汇报的运行时信息,推测任务资源需求和服务器资源的可用性,并周期性地划分任务执行阶段.信息推测模块与调度模块异步运行,它不保证向调度模块提供包含最新运行时信息的推测结果,极大地减少了运行时信息处理、资源信息推测、执行阶段划分等工作对调度的影响.策略管理模块负责管理资源分配、公平性等策略,并根据配置与服务器当前的资源调度策略等向调度模块提供各种策略支持.策略管理模块支持在运行时添加资源匹配、公平性策略及更改策略配置,默认的公平性策略为主资源公平.约束条件检查模块用于保证调度决策生效后,服务器上所有运行中任务的服务质量约束条件都能得到满足.约束条件检查模块并不等待完成资源匹配等步骤之后才开始检查,它与资源匹配同时工作,如果资源匹配等前置步骤未通过,则检查工作直接停止.这样的预检查机制很大程度上提高了调度请求的处理效率.策略调整模块异步运行于调度模块,它存储各服务器各资源近期资源推测符合度,并根据服务器的资源推测符合度及任务完成时间信息自动调整服务器的资源调度策略.压缩管理模块从信息推测模块和策略调整模块获取服务器运行中任务信息、待匹配任务信息以及服务器当前资源调度策略等信息,并根据相关信息计算服务器各阶段各资源的最大资源压缩率,供调度模块在匹配中使用.

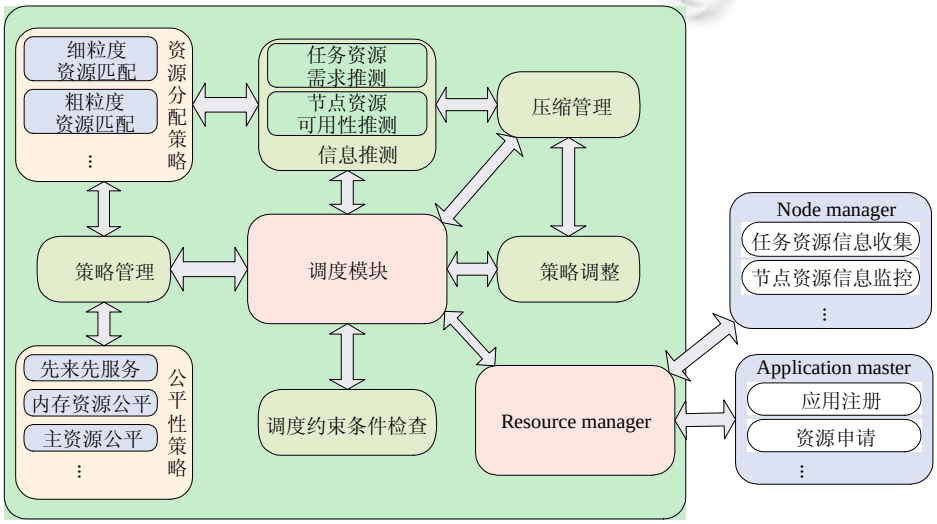


Fig.6 Architecture of the fine-grained scheduler

图 6 细粒度调度器架构图

Yarn 平台 Node Manager 增加了周期性解析 Proc 文件夹信息的处理逻辑,以采集运行时服务器以及负载的运行时信息.Node Manager 会根据采集信息计算服务器资源推测符合度和空闲资源量等信息,计算得到的信息会连同原始运行时信息一起汇报给 Resource Manager.Application Master 在向 Resource Manager 注册时,需要提供应用程序代码及参数的 MD5 值和输入数据量等信息,以帮助信息推测服务标识各 Application Master; Application Master 在向 Resource Manager 申请资源时,需要使用自身标识和任务类型等信息来标识任务,用以帮助调度器和信息推测服务确定推测信息.

虽然本文只在 Yarn 平台中实现了细粒度资源调度器,但是细粒度资源调度方法并不局限于某一个特定的云计算资源管理平台中.该方法也可以借鉴或实现到其他平台中,用以提高平台的资源利用率及性能.

## 4 实验与结果

### 4.1 测试环境及负载

本文使用中国国家高性能计算中心(西安)的集群进行测实验证,集群共包含 24 台服务器,服务器具体匹配见表 1.测试中,使用 1 台服务器配置 Yarn 平台相关管理服务,其余服务器配置为平台计算节点.平台上配置 MapReduce 和 Spark 等计算框架,测试负载由多种计算框架应用组成,且各应用输入数据大小服从文献[4]描述的 Facebook 公司负载的数据大小分布.测试输入数据包括 Wikipedia 数据和随机生成数据.

**Table 1** Hardware information of server

**表 1** 服务器硬件资源配置

| 硬件资源 | 配置 |
|---|---|
| CPU | 2×Intel Xeon E5-2670(2.6GHz,8 Cores,16 Threads) |
| Memory | 8×4GB REG ECC DDR3 1600MHz |
| Storage | 2×300GB 10kr/m SAS |
| Network | 2×1000 Mb/s,InfiniBand QDR HCA 40Gb/s |

测试中,对比算法包括 FIFO,Capacity[3],Fair[4]和 DRF[21],测试环境分为独享环境和云环境,测试使用的负载分为离线负载和在线负载.独享环境中,集群只运行 Yarn 平台及其应用负载;云环境中,集群除了运行 Yarn 平台及其应用负载之外,还存在一些真实的科学计算应用共享计算资源.离线负载指测试负载中所有作业在开始时集中提交,总数据量为 400GB;在线负载指负载提交时间服从泊松分布,总数据量从 50GB 逐步递增到 400GB,每次递增 50GB.

测试中,采样资源分配信息计算 Jain's fairness index[40]衡量资源分配的公平性.完成时间指从负载提交到负载完成所用时间.响应时间指从调度器收到调度请求到第一次匹配资源成功所用时间,其包括排队等待时间和资源匹配时间.

### 4.2 离线负载独享环境测试

离线负载独享环境下各算法的公平性采样结果如图 7 所示.采样结果显示:DRF 的平均公平性最优,而 FIFO 的平均公平性最差,FGM 的平均公平性略低于 DRF 而高于其他调度算法.FGM 在调度过程中,按照公平性策略排序待调度集合,因此调度分配的公平性得以保证.相比测试中其他调度方法,细粒度资源匹配算法的粒度更细,匹配的成功率有所下将,因此公平性也略受影响,平均公平性相比 DRF 下降 0.31%左右.图 7 中的公平性测试结果说明,FGM 可以在离线负载独享环境中提供良好的资源公平性.

独享环境下离线负载集中提交,平均调度响应时间相比其他环境更高.因此,我们在独享环境下使用离线负载测试 FGM 的调度请求响应时间.图 8 所示为独享环境中各调度算法对离线负载的调度请求平均响应时间.从图中测试结果可以发现,FIFO 的调度响应时间远远高于其他算法,这与其算法调度策略造成的后提交作业等待有关.Capacity 算法的调度响应时间大于除 FIFO 之外的其他方法,这与其配置的排队策略有关.FGM 的平均调度响应时间相比 FIFO 和 Capacity 有很大降低,而相比 Fair 和 DRF 增加 75ms 左右.FGM 中,资源信息处理和推测等机制均独立于资源调度异步执行,因此增加的调度响应时间主要是因为细化了 CPU、内存资源量和时间

两方面的匹配粒度.略微增加的平均响应时间使得 FGM 相较 DRF 提升了资源利用率和负载性能,相较 Fair 还拥有更优异的资源公平性.因此,本文认为,FGM 在调度响应中增加的时间代价是可以接受的.

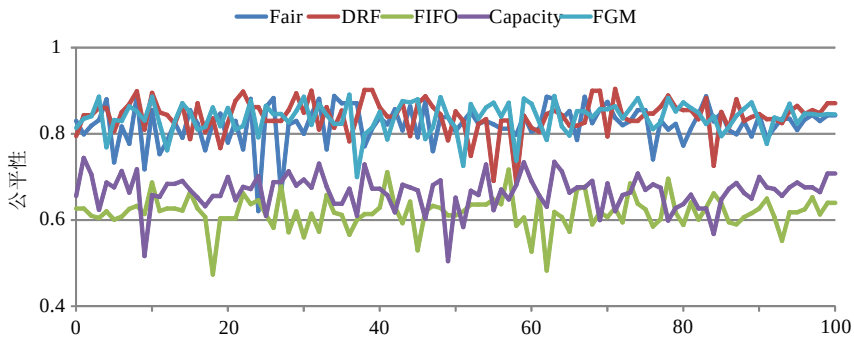


Fig.7 Resource fairness results of offline workload in the dedicated environment

图 7 离线负载独享环境公平性

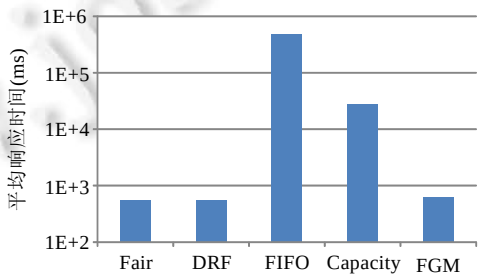


Fig.8 Average response time of scheduling in the dedicated environment using offline workload

图 8 离线负载独享环境平均调度响应时间

离线负载独享环境下各算法的作业平均完成时间及其累积分布分别如图 9 和图 10 所示.图 9 中测试结果显示,FGM 的作业平均完成时间相对于其他 4 种对比方法分别提升 30.92%,34.49%,52.09%,19.97%.从图 10 的完成时间累积分布图可以看出:FGM 相较 Fair,DRF 算法对于离线负载中完成时间小于 100s 的作业方面并无优势,FGM 中压缩资源的优化方式对作业的执行时间有所影响.而对于所需完成时间大于 100s 的作业,FGM 的优势逐渐体现,并在 150s 左右超过其他算法的作业完成率.这说明 FGM 中资源压缩和细粒度匹配等优化方式对于执行时间较长的作业更加友好.最终,FGM 最长作业完成时间只需 600s 左右,大幅领先各种对比算法.由于负载完成具有长尾现象,负载完成时间与执行时间长的作业也很大关系,因此缩短长作业的完成时间更有利于优化负载完成时间.

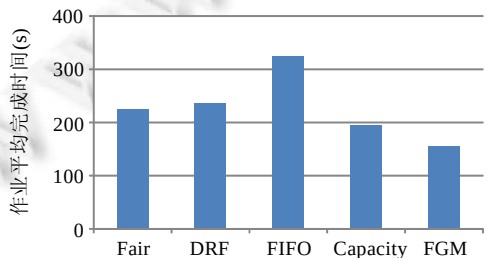


Fig.9 Average completion time of jobs in the dedicated environment using offline workload

图 9 离线负载独享环境作业平均完成时间

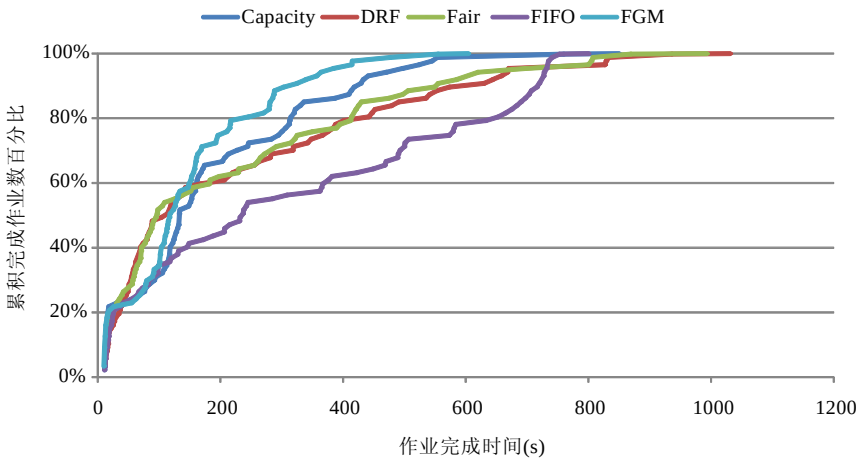


Fig.10 CDF of job completion times in the dedicated environment using offline workload

图 10 离线负载独享环境作业完成时间累积分布图

各算法在离线负载独享环境中的负载完成时间如图 11 所示.图中测试结果显示,FGM 相对其他 4 种对比算法取得了 19.47%~27.94%不等的性能提升.这证实了该方法可以有效提高独享环境下,离线负载的资源利用率和负载性能.

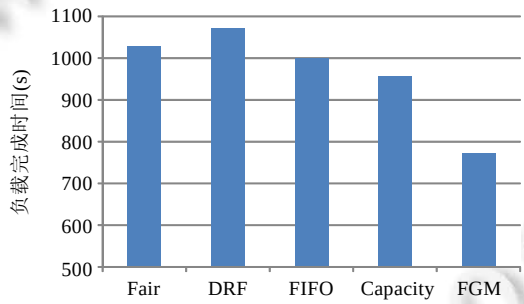


Fig.11 Completion times of offline workloads in the dedicated environment

图 11 离线负载独享环境负载完成时间

### 4.3 在线负载独享环境测试

图 12 所示为在线负载独享环境下各算法的任务平均完成时间,从测试结果可以看出:FGM 的小任务平均完成时间相较其他算法略有增加,增加的幅度在 1.83s~5.45s 之间.这是由于两方面原因共同造成的:一方面,细粒度资源匹配过程中对任务资源需求进行一定程度的压缩;另一方面,小任务执行时间短、对资源压缩较为敏感.FGM 的大任务平均完成时间相较其他算法减少 6.84s~39.02s 不等.原因包括以下 3 个方面:首先,任务执行过程中资源压缩并不是持续存在的,大任务执行时间长,对资源压缩不敏感;其次,细粒度资源匹配算法细化了资源匹配的粒度,提高了资源的匹配程度;最后,FGM 将压缩控制在一定程度内,并采取监控资源使用、调度约束检查、调度策略调整等机制,避免了严重资源竞争.小任务和大任务的平均完成时间再次证实了资源压缩对执行时间较短的任务或作业影响更大的结论.FGM 的任务完成时间相对其他算法略有增加,幅度在 1.2s~3.53s 之间.任务平均完成时间的增加并不一定意味着性能的下降,这是因为细粒度资源匹配、资源压缩分配等机制一定程度上提高了服务器上可并行执行的任务数量,可以优化整体性能.

独享环境中,各算法对不同数据量在线负载取得的完成时间如图 13 所示.图中测试结果曲线表明:FGM 对负载的性能提升效果随负载输入数据量增大而显著,最终达到 31%左右.这是由于负载输入数据量一定程度上

决定了负载中完成时间长的作业的数量,数据量越大,长作业数量也会越多.而 FGM 对所需完成时间长的作业的优化效果明显,因此其对数据量越大的负载的提升效果越明显.FGM 在独享环境中对在线负载的测试结果说明,该方法可以提高独享环境中资源利用率和负载整体性能.

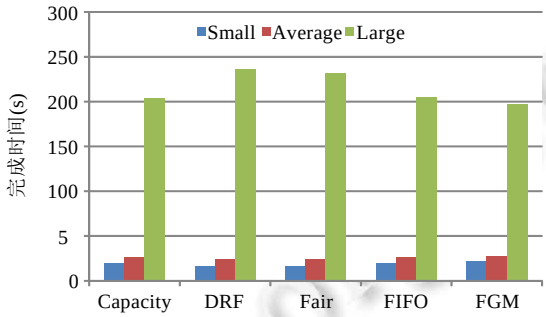


Fig.12 Average completion time of tasks in the dedicated environment using the online workload

图 12 在线负载独享环境任务平均完成时间

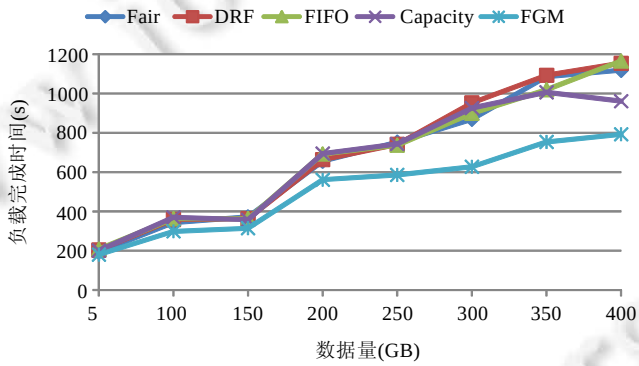


Fig.13 Completion times of online workloads in the dedicated environment

图 13 在线负载独享环境负载完成时间

### 4.4 在线负载云环境测试

在线负载云环境中,公平性采样结果如图 14 所示.在线负载下,各算法平均公平性相比离线负载均略有降低,这可能由于在线负载提交时间引起的.FGM 的平均公平性结果非常接近 DRF,仅略低 0.09%.因此,该方法在云计算环境中仍然保持了资源分配的公平性.

在线负载云环境中,任务平均完成时间如图 15 所示,图中各颜色柱形意义与图 12 中相同.图中测试结果显示:FGM 的小任务平均完成时间较 Fair 和 DRF 高 2.3s 左右,而低于其他对比算法 5s 左右;大任务平均完成时间相对其他算法低 34.68s~135.47s 不等,降低幅度大幅增加;所有任务平均完成时间相对其他算法低 4.24s~16.89s 不等.此环境下测试结果与之前有显著不同,这由两方面原因导致:一方面,测试环境中存在的科学计算等其他应用影响了 Yarn 平台负载的执行,导致任务执行时间增加;另一方面,FGM 适应了环境中其他应用的资源使用行为,并采取相应措施有效避免了激烈的资源争用,因此任务执行时间增加幅度明显小于其他算法.

在线负载云环境下作业平均完成时间及其累积分布如图 16 和图 17 所示.

图 16 中,FGM 相对于对比算法最少提高 35.74%,最多提高 61.87%.这很大程度上是因为云环境中的非 Yarn 应用大幅增加了作业完成所需的时间.FGM 能够感知服务器可用资源的变化,因此受到影响较小,最终导致 FGM 作业平均完成时间远远优于对比算法.

从图 17 的累积分布图中也可以观察到相同结论:FGM 对于执行时间短的作业并无明显优势,而 FGM 作业

最长完成时间为 1 000s 左右,其他算法的作业最长完成时间均远大于此.

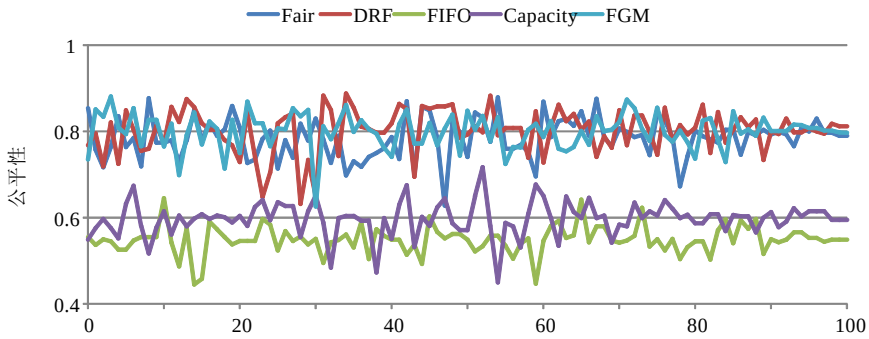


Fig.14 Resource fairness results of online workload in the cloud environment

图 14 在线负载云环境公平性

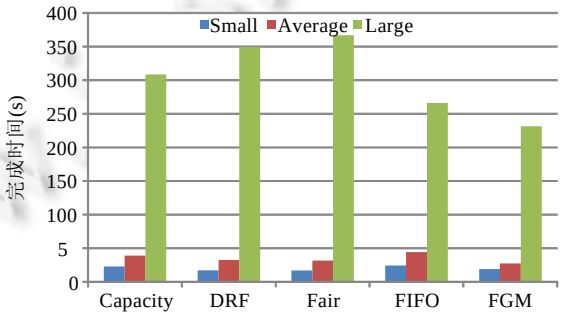


Fig.15 Average completion time of tasks in the cloud environment using the online workload

图 15 在线负载云环境任务平均完成时间

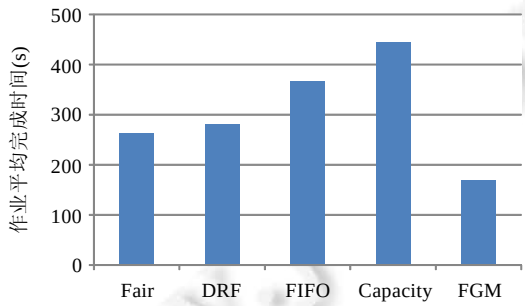


Fig.16 Average completion time of jobs in the cloud environment using online workload

图 16 在线负载云环境作业平均完成时间

在线负载云环境不同数据量负载的完成时间如图 18 所示.对比图 13 观察,可以发现:云环境的负载完成时间相比独享环境中相同数据量负载完成时间普遍大幅增加,且负载数据量越大完成时间增加越明显.FGM 取得的优化效果与之前测试中的趋势相同:优化效果随负载数据量增大而增加.不同的是,FGM 最终取得 57.48%~62.4%的提升,相比之前环境中的优化效果更明显.这是由多方面原因共同导致的:首先,FGM 通过细粒度资源匹配、资源需求压缩等方式提高了集群资源的利用率;其次,FGM 通过感知服务器计算资源可用性、调整服务器调度策略等方式避免了激烈的资源争用,从而降低长作业完成时间受影响程度;最后,其他对比算法受到环境中其他应用的影响,长作业完成时间大幅增加导致负载完成时间大幅增加,凸显了 FGM 的优化效果.根据图 18 中

的测试结果,我们可以得出结论:FGM 可以在云环境下提高资源利用,优化负载完成时间.

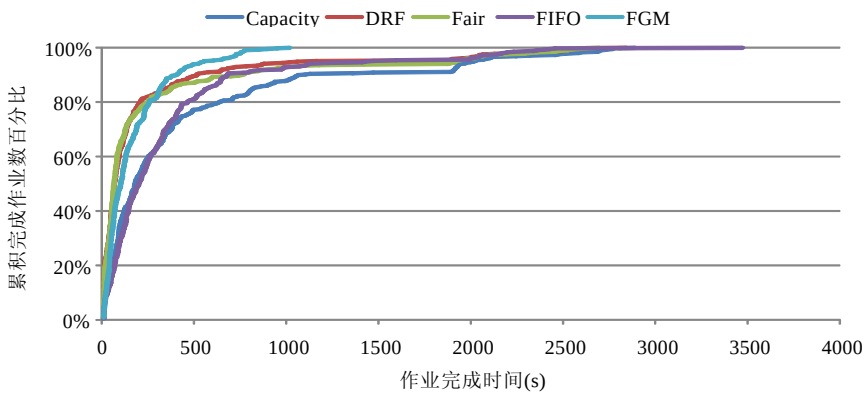


Fig.17 CDF of job completion times in the cloud environment using online workload

图 17 在线负载云环境作业完成时间累积分布图

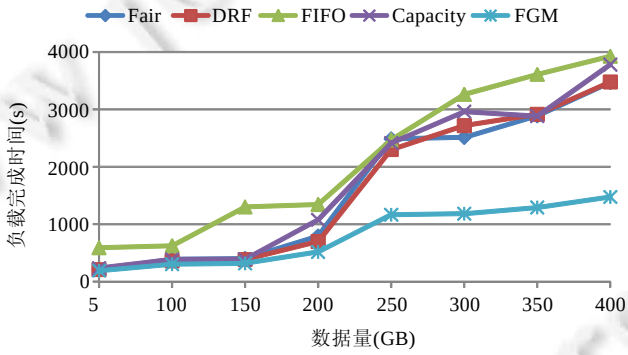


Fig.18 Completion times of online workloads in the cloud environment

图 18 在线负载云环境负载完成时间

## 5 结 论

针对云计算资源管理平台中使用固定资源量的粗粒度分配方式引起的资源碎片、不良共享、过度分配及限制集群资源利用率的问题,本文提出了一种细粒度资源调度方法.该方法推测任务资源需求量,并将任务划分为若干执行阶段;在资源分配中兼顾公平性和数据本地性,分阶段考虑资源需求、资源特性等因素,细化分配粒度,提高资源利用率,解决上述问题;采用分配前检查调度约束条件、运行时监控资源使用状态、动态调整调度策略等机制,调节资源匹配及资源压缩,保证资源利用效率和负载性能.在多种测试环境下,采用不同负载对本文工作进行验证,测试结果显示:本文提出的细粒度资源调度方法最高使独享环境下离线及在线负载的负载完成时间分别减少 27%和 31%左右;使云计算共享环境下在线负载完成时间减少 60%左右.因此,该方法可以在不丧失公平性、调度响应时间可接受的前提下有效地细化资源分配粒度,提高云计算平台资源利用效率.

在未来的研究工作中,我们计划在细粒度资源调度方法中增加匹配资源的种类,进一步提高匹配程度.

**References**:

[1] Reiss C, Tumanov A, Ganger GR, Katz RH, Kozuch MA. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In: Proc. of the 3rd ACM Symp. on Cloud Computing. 2012.

[2] Staples G. TORQUE resource manager. In: Proc. of the 2006 ACM/IEEE Conf. on Supercomputing. 2006.

[3] Apache hadoop 2.6.0—Hadoop map reduce next generation-2.6.0—Capacity scheduler. https://hadoop.apache.org/docs/r2.6.0/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html

[4] Zaharia M, Borthakur D, Sarma JS, Elmeleegy K, Shenker S, Stoica I. Job scheduling for multi-user mapreduce clusters. Technical Report, UCB/EECS-2009-55, Berkeley: EECS Department, University of California, 2009.

[5] Zaharia M, Borthakur D, Sen Sarma J, Elmeleegy K, Shenker S, Stoica I. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In: Proc. of the 5th European Conf. on Computer Systems. 2010. 265−278.

[6] Zaharia M, Konwinski A, Joseph AD, Katz RH, Stoica I. Improving MapReduce performance in heterogeneous environments. In: Proc. of the OSDI, Vol.8. 2008. 7.

[7] Welcome to ApacheTM hadoop®! http://hadoop.apache.org/

[8] Reiss C, Tumanov A, Ganger GR, Katz RH, Kozuch MA. Towards understanding heterogeneous clouds at scale: Google trace analysis. Technical Report, Intel Science and Technology Center for Cloud Computing, 2012. 84.

[9] Abdul-Rahman OA, Aida K. Towards understanding the usage behavior of google cloud users: The mice and elephants phenomenon. In: Proc. of the 2014 IEEE 6th Int'l Conf. on Cloud Computing Technology and Science (CloudCom). 2014. 272−277.

[10] Di S, Kondo D, Cappello F. Characterizing cloud applications on a google data center. In: Proc. of the 2013 42nd Int'l Conf. on Parallel Processing (ICPP). 2013. 468−473.

[11] Boutin E, *et al*. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In: Proc. of the 11th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2014). 2014. 285−300.

[12] Schwarzkopf M, Konwinski A, Abd-El-Malek M, Wilkes J. Omega: Flexible, scalable schedulers for large compute clusters. In: Proc. of the 8th ACM European Conf. on Computer Systems. 2013. 351−364.

[13] Grandl R, Ananthanarayanan G, Kandula S, Rao S, Akella A. Multi-Resource packing for cluster schedulers. In: Proc. of the 2014 ACM Conf. on SIGCOMM. 2014. 455−466.

[14] Lu P, Lee YC, Wang C, Zhou BB, Chen J, Zomaya AY. Workload characteristic oriented scheduler for mapreduce. In: Proc. of the 2012 IEEE 18th Int'l Conf. on Parallel and Distributed Systems (ICPADS). 2012. 156−163.

[15] Tian C, Zhou H, He Y, Zha L. A dynamic mapreduce scheduler for heterogeneous workloads. In: Proc. of the 2009 8th Int'l Conf. on Grid and Cooperative Computing. 2009. 218−224.

[16] Dean J, Barroso LA. The tail at scale. Communications of the ACM, 2013,56(2):74−80.

[17] Garraghan P, Ouyang X, Yang R, McKee D, Xu J. Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters. IEEE Trans. on Services Computing, 2016,12(1).

[18] Vavilapalli VK, *et al*. Apache hadoop yarn: Yet another resource negotiator. In: Proc. of the 4th Annual Symp. on Cloud Computing. 2013. 5.

[19] Zhang Z, Li C, Tao Y, Yang R, Tang H, Xu J. Fuxi: A fault-tolerant resource management and job scheduling system at Internet scale. Proc. of the VLDB Endowment, 2014,7(13):1393−1404.

[20] Verma A, Pedrosa L, Korupolu M, Oppenheimer D, Tune E, Wilkes J. Large-Scale cluster management at google with Borg. In: Proc. of the 10th European Conf. on Computer Systems. 2015. 18.

[21] Ghodsi A, Zaharia M, Hindman B, Konwinski A, Shenker S, Stoica I. Dominant resource fairness: Fair allocation of multiple resource types. In: Proc. of the NSDI, Vol.11. 2011..

[22] Grandl R, Chowdhury M, Akella A, Ananthanarayanan G. Altruistic scheduling in multi-resource clusters. In: Proc. of the 12th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2016). 2016.

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

[24] Hindman B, *et al*. Mesos: A platform for fine-grained resource sharing in the data center. In: Proc. of the NSDI, Vol.11. 2011.

[25] Isard M, Prabhakaran V, Currey J, Wieder U, Talwar K, Goldberg A. Quincy: Fair scheduling for distributed computing clusters. In: Proc. of the ACM SIGOPS 22nd Symp. on Operating Systems Principles. 2009. 261−276.

[26] Gog I, Schwarzkopf M, Gleave A, Watson RN, Hand S. Firmament: Fast, centralized cluster scheduling at scale. In: Proc. of the 12th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2016). 2016. 99.

[27] Ananthanarayanan G, *et al*. Reining in the outliers in map-reduce clusters using mantri. In: Proc. of the OSDI, Vol.10. 2010. 24.

[28] Kung HT, Robinson JT. On optimistic methods for concurrency control. ACM Trans. on Database Systems (TODS), 1981,6(2): 213−226.

[29] Ghodsi A, Zaharia M, Shenker S, Stoica I. Choosy: Max-min fair sharing for datacenter jobs with constraints. In: Proc. of the 8th ACM European Conf. on Computer Systems. 2013. 365−378.

[30] Chen HK, Zhu JH, Zhu XM, Ma MH, Zhang ZS. Resource-Delay-Aware scheduling for real-time tasks in clouds. Journal of Computer Research and Development, 2017,54(2):446−456 (in Chinese with English abstract).

[31] Agarwal S, Kandula S, Bruno N, Wu MC, Stoica I, Zhou J. Re-Optimizing data-parallel computing. In: Proc. of the 9th USENIX Conf. on Networked Systems Design and Implementation. 2012. 21.

[32] Ferguson AD, Bodik P, Kandula S, Boutin E, Fonseca R. Jockey: Guaranteed job latency in data parallel clusters. In: Proc. of the 7th ACM European Conf. on Computer Systems. 2012. 99−112.

[33] Khan M, Jin Y, Li M, Xiang Y, Jiang C. Hadoop performance modeling for job estimation and resource provisioning. IEEE Trans. on Parallel and Distributed Systems, 2016,27(2):441−454.

[34] Liu KN, Tong W, Liu JN, Zhang J. Flexible and efficient VCPU scheduling algorithm. Ruan Jian Xue Bao/Journal of Software, 2017,28(2):398−410 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5059.htm [doi: 10.13328/j.cnki.jos.005059]

[35] Xu SY, Lin WW, Wang ZJ. Virtual machine placement algorithm based on peak workload characteristics. Ruan Jian Xue Bao/Journal of Software, 2016,27(7):1876−1887 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4918.htm [doi: 10.13328/j.cnki.jos.004918]

[36] Ren LF, Wang WJ, Xu X. Uncertainty-Aware adaptive service composition in cloud computing. Journal of Computer Research and Development, 2016,53(12):2867−2881 (in Chinese with English abstract).

[37] Ananthanarayanan G, *et al*. Pacman: Coordinated memory caching for parallel jobs. In: Proc. of the 9th USENIX Conf. on Networked Systems Design and Implementation. 2012.

[38] Morton K, Balazinska M, Grossman D ParaTimer: A progress indicator for MapReduce DAGs. In: Proc. of the 2010 ACM SIGMOD Int'l Conf. on Management of Data. 2010. 507−518.

[39] Zhang X, Tune E, Hagmann R, Jnagal R, Gokhale V, Wilkes J. CPI 2: CPU performance isolation for shared compute clusters. In: Proc. of the 8th ACM European Conf. on Computer Systems. 2013. 379−391.

[40] Jain R, Chiu DM, Hawe WR. A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer System. Vol.38, 1984.

**附中文参考文献:**

[30] 陈黄科,祝江汉,朱晓敏,马满好,张振仕.云计算中资源延迟感知的实时任务调度方法.计算机研究与发展,2017,54(2):446−456.

[34] 刘珂男,童薇,冯丹,刘景宁,张炬.一种灵活高效的虚拟 CPU 调度算法.软件学报,2017,28(2):398−410. http://www.jos.org.cn/1000-9825/5059.htm [doi: 10.13328/j.cnki.jos.005059]

[35] 徐思尧,林伟伟,王子骏.基于负载高峰特征的虚拟机放置算法.软件学报,2016,27(7):1876−1887. http://www.jos.org.cn/1000-9825/4918.htm [doi: 10.13328/j.cnki.jos.004918]

[36] 任丽芳,王文剑,许行.不确定感知的自适应云计算服务组合.计算机研究与发展,2016,53(12):2867−2881.

**周墨颂**(1988－),男,博士生,主要研究领域为云计算资源管理.

**陈衡**(1979－),男,博士,讲师,CCF 专业会员,主要研究领域为高性能计算机体系结构及其核心软件,并行算法与编程模型.

**董小社**(1963－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为高性能计算机体系结构及其核心软件,并行算法与编程模型,网格/云计算,存储系统.

**张兴军**(1969－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为高性能计算机体系结构,网络存储,网络计算.