注：居中，宋体，小一号，加黑，学术型硕士填写内容为“硕士学位论文”；专业学位硕士填写内容为“专业学位硕士学位论文”。阅后删除此文本框。

注：此页为封一。

阅后删除此文本框。

注：此处是论文中英文题目，中文题目，不超过20个汉字，居中，字体：华文细黑，加黑，字号：二号，行距：多倍行距1.25，间距：前段、后段均为0行，取消网格对齐选项。英文题目，与中文题目对应，居中，字体：Times New Roman，字号：三号，加黑，行距：多倍行距1.25，间距：前段、后段均为0行，取消网格对齐选项。阅后删除此文本框。

硕 士 学 位 论 文

大连理工大学硕士学位论文格式规范

The Format Criterion of Master Degree Thesis of DUT

注：此处是研究生论文的标识，按照实际情况填写即可。打印（宋体，小三）或手写都可以。工程硕士将“学科、 专业”标题改为“工程领域”。阅后删除此文本框。

作 者 姓 名：

学科、 专业：

学 号：

指 导 教 师：

完 成 日 期：

大连理工大学

Dalian University of Technology

大连理工大学学位论文独创性声明

摘 要

目 录

引 言

1 绪论

## 研究背景

### 1.1.1 云计算

云计算是一种基于互联网的计算模型，它将计算处理资源及相关数据按需提供给其他设备。云计算[1]的出现与发展使得用户可以随时随地动态地以最小的管理成本获取相关计算资源，如计算机网络、存储、服务等。数据中心及其相关软件硬件称之为云(cloud)。目前国内很多公司也相继推出了云服务，如七牛云存储、百度云网盘等。按照服务的不同，广义上可以把云计算分为软件即服务(Software as a Service, SaaS)、平台即服务(Platform as a Service, PaaS)、基础设施即服务(Infrastructure as a Service)。图1.1展示了云服务提供商与云服务消费者的关系。

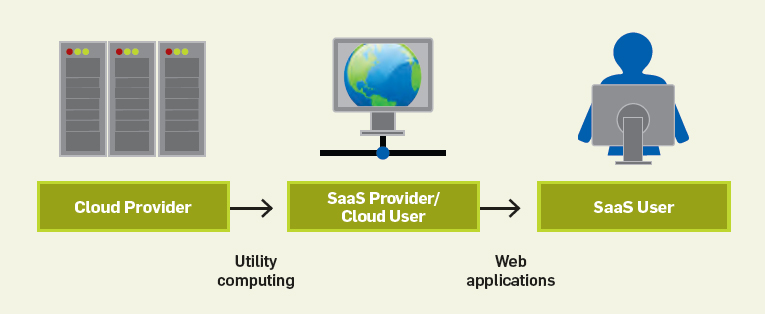


图1.1 云服务提供商与消费者的关系

Fig. 1.1 Relationship between cloud service providers and its consumers

现如今，通过互联网获取信息、知识已变得稀松平常，而且无需了解其底层原理。而云计算更是该模式的一种延伸[2]。云计算已经成为一种流行的资源提供与获取方式。云服务提供商根据其提供的云服务质量(Quality of Service, QoS)向消费者收费；消费者如企业为了寻求降低或在一定程度上避免成本，通过各种各样可能的形式使用云服务。现有的云服务使用方式灵活多样，如可以使用云服务(如百度云)进行数据存储，也可以将自己的任务通过互联网提交到远程服务商进行处理并等待结果的返回。两种方式共同的优点是，不再需要大量的大型存储设备。

云服务商面临的用户群体庞大[3-4]，因而如何为用户提供高效得服务就是云计算中一个重要的问题。而其中一个核心问题就是如何调度云服务用户提交的任务。云服务提供商在大型服务器上部署多种服务[5] ，根据各个服务的服务质量（quality of service, QoS）属性指定收费标准。图1.2展示了云平台的物理架构。用户向服务商提交的工作流由这些服务完成，每个工作流中包含多个任务，每个任务可以由若干个服务完成。



图1.2 云平台的物理架构

Fig. 1.2 The physical architecture of cloud platform

云服务提供商为了实现用户的按需请求服务，必然要显著地冗余配置其基础设施并相应地为低利用率的资源支付高昂的运行成本，或者是按比例拒绝一部分用户的任务请求。前者会为云服务提供商带来额外的不合理的成本，而后者无疑会降低用户体验并且某种意义上讲这种服务也就不再是按需请求的了[6]。而从云服务消费者角度来看，并非所有用户都要求完全意义上的按需云服务，甚至许多用户提交的任务在设计之初就设计为允许中断的可分割的任务。一些系统如SETI@HOME[7]被设计为可以利用空闲的资源，同时也可以被强占。这样设计的好处是，可以在服务中断时可以从故障中恢复。因而，云服务商采取怎样的策略或是算法处理用户提交的任务就变得尤为重要。通常，云服务提供商为了高效地调度用户的任务，会根据具体的调度目标采取不同的调度策略，如最低执行开销（花费）、最低执行时间、负载均衡、公平调度等。一个合理而高效的调度策略应该综合考虑任务的完成时间、费用、可靠性。作业调度或者说任务调度一直是云计算中的一个核心问题[8]。

### 1.1.2 异构平台

异构平台由具有不同计算能力的电脑资源组成[9]。由于异构平台计算资源间的异构性，目前提高其计算性能的方法仍旧主要是提高计算资源的数目及其计算能力。在计算资源的数目得到提升时，多核的异构平台还可以通过整合专用处理能力来处理特定任务来获得性能。同时，异构平台还具有综合利用多种处理器（特别是CPU与GPU）的能力。通常意义下，以3D图像渲染能力而闻名的GPU在处理大数据集的计算密集型数学运算方面也表现出众；同时CPU可以继续运行操作系统以及执行传统意义上的。异构平台系统添加而随着异构平台内部互相联通的计算资源数量逐渐增加，对合理、高效的调度策略的需求也日益提高。

### 1.1.2 工作流

工作流（Workflow）最早被商业领域广泛使用[10]。这种意义下的工作流是由一系列可重复的、经过仔细安排的商业活动组成。而这些商业活动会经过一组系统的处理流程，如变换材料、提供服务或者是处理信息。也可以把工作流描述为一个人或者是一个团队的一系列系统化的操作。这种把一个具体的任务划分为一系列具有先后顺序的活动可以明显地提高工作效率，并且可以更好的管理整个工作的流程。上世纪九十年代随着个人计算机、笔记本的普及和网络技术的发展与成熟，工作流及其相关技术与概念已逐渐普及开来，不再局限于商业领域而逐渐进入金融业、银行业、软件工程以及科学实验等相关领域。

工作流管理系统（Workflow Management Systems）[11]最早被引入用来支持一般意义上的需要持久运行的商业过程及其工作流。工作流管理系统是一套完备的管理与支持工作流的软件系统。它将一系列流程及其内部的相关逻辑进行抽象，并根据某些规则使用使用计算机编程语言恰当表示并实现。它不仅可以执行与监管工作流的执行，还负责工作流各个环节的信息交互以保证群体成员之间的信息共享。图1.2展示了一个工作流及其相关角色。创建事项需由任务发起者进行，同时需要监察人员监察到期事项。

通常单个工作流可以使用DAG（directed acyclic graph）图来描述[12-13]。一个DAG图中每一个节点表示对应工作流中的一个工作或一个步骤；连接各个节点的有向边描述某种规则或是约束，并且可以在有向边上标注各种权值或权值对来描述相关属性或参数。DAG图的优势是可以清晰地展示出对应工作流的抽象逻辑，并且抽象直观，可以轻易地用其表达出业务过程。在使用DAG图来描述工作流后可以对工作流进行许多的研究，例如：如何更合理地安排步骤；如何制定相关步骤间的规则以便于节点间的通信或是交接等。其中一个核心问题是，如何合理配置资源使得DAG或者说其对应的工作流得以高效、合理的调度执行，也即是工作流的调度相关问题。



图1.2 工作流工作模式

Fig. 1.2 Workflow work mode

由于一个工作流内可能会有多组相关节点以及多组无关节点（即该组节点间不存在有向边的连接约束），这一特性可以用来将一个DAG图中的节点按照某种规则进行划分，然后在不同的资源上调度执行。而云计算平台具备资源异构性以及资源动态性，因此可以很好地应用于工作流调度。云服务一般由上千的计算资源组成，因此可以分别调度执行多个子任务以缩短整体调度的执行时间从而获得更好的用户体验。在调度过程中，可以对工作流进行划分，然后进行资源筛选，进而调度执行。因此，可以使用DAG图所表示的工作流研究相关调度问题。

## 1.2 研究内容及贡献

一律用阿拉伯数字连续编页码。页码应由引言首页开始，作为第1页。封一、封二

## 论文组织结构

2 DAG工作流的分类及相关调度算法

## 2.1 DAG工作流的分类

### 2.1.1 单DAG与多DAG

可以将工作流管理系统按照用户提交的任务是否需要多用户间协作或交互分为两种类型，MC类型以及MI类型[14]。MC类型的工作流管理系统中，多个用户提交的任务需要以协作的方式进行监控、调用执行、数据传递等；MI类型的工作流管理系统中多个用户提交的任务相互独立，只需要工作流管理系统进行宏观调度即可。按照需要调度的DAG数量可以将DAG划分为单DAG与多DAG两类[15-16]。

单DAG调度是在一组资源上单独调度某个用户提交的任务。但是多数情况下，一个任务需要多个用户进行协作，此时可以将多个用户的工作合理地合并为一个工作流并用DAG图进行描述，然后进入工作流管理系统进行调度。在调度到该单DAG时，整个工作流可以按3需占有所有资源，调度时按照各个任务间的依赖关系采取某种策略进行资源分配即可。常见的单DAG调度策略有表调度策略，基于分层的调度策略和基于遗传算法的调度策略等。

多DAG工作流[17]不仅要考虑单DAG调度中的问题，还要处理多个DAG之间的资源竞争、公平性调度等多方面的问题。针对不同类型的调度问题，对多DAG工作流的限制条件也不同。部分工作流管理系统需要综合考虑用户提交的多DAG工作流的截止期，以尽可能多地调度用户提交的任务为目标。换言之，调度的目标是针对各个用户提交的工作流，尽可能多的提高系统吞吐率。而从用户的角度考虑，通常情况下工作流执行时间越少，花费越少。同时，也需要相应的策略来确定多个工作流之间的调度优先级以保证调度的公平性。因此相较于单DAG工作流，多DAG工作流的调度策略需要对多个问题进行平衡。Henan Zhao等人[18]深入研究了多DAG工作流调度中的公平性问题，并提出了一种侧重公平性的调度策略以保证多个用户提交的工作流能得到相对公平的调度。

### 2.1.2 静态DAG与动态DAG

按照DAG的生成方式，可以将DAG工作流分为静态DAG和动态DAG[19-20]。通常意义下，静态DAG指的是在调度时，DAG的结构、每个DAG内所代表任务数量、任务间的约束规则和需要调度的DAG数量等信息都可以获得。这意味着，调度静态DAG的调度策略在分配资源、决定多DAG中DAG的调度优先级时可用条件更多，因此能给出局部较优的解。对于静态DAG，一般采用离线调度的方式，只需要根据DAG的具体信息，根据某种离线调度策略确定优先级后合理分配资源即可。

动态DAG则包含两层意思。其一是指对于单个的DAG，它的相关信息如内部任务数量、任务间的约束规则等信息可能是无法获得的，或者有可能做出改变；其二是指对于一组动态DAG，有可能在某些时间点会有新的DAG到达，也有可能用户在某一时刻撤销所提交的任务。对于后者，如何合理地安排多个DAG间的调度优先级时一个调度策略的核心。针对动态DAG的调度策略称为在线调度，动态DAG中的各种不确定性使得在线调度策略相对更加复杂。

## 2.2 相关DAG调度算法研究

### 2.2.1 表调度算法与分层算法

表调度算法是最经典的DAG调度算法[21-22]。表调度算法可以作用于静态DAG，通常只有两个步骤：先根据给定DAG中的所有任务的关系，按照前后依赖关系并结合某种策略决定任务间的优先级并构造出一个调度列表；然后为优先级最高的任务分配能够使它最快完成的资源并调度执行，同时将该任务从列表中移除。通过循环上述两个步骤即可完成对给定DAG的调度。上述表调度算法是在调度之初就根据相应的条件确定了DAG内部所有任务的优先级，并且在调度过程中不做出改变，因此该算法也成为静态表调度算法。图2.1展示了表调度算法的一般流程。相对于静态表调度算法，还有动态表调度算法。与静态表调度算法不同的是，动态表调度算法会在执行完每一个任务后根据综合考虑当前计算资源可用情况以及后续任务的依赖关系重新决定部分任务的优先级，并根据新计算出的优先级将调度列表重新排列。因此，动态表调度算法比静态表调度算法多一个步骤。

有诸多学者对表调度算法进行了研究。[H. Topcuoglu](http://ieeexplore.ieee.org/search/searchresult.jsp?searchWithin=%22Authors%22:.QT.H.%20Topcuoglu.QT.&newsearch=true)等人[23]提出的HEFT算法是十分著名的调度算法，其后又有诸多研究者根据不同的应用场景对其进行了性能优化。HEFT算法综合考虑一个任务的平均计算开销和平均通信开销然后决定总体的调度顺序，即优先级序列。由于HEFT算法的调度结果相对较好加之其算法复杂度仅在DAG内部任务数量的平方数量级，HEFT算法被广泛应用在实际调度中。Guan W等人[24]在2016年提出的HSIP（Heterogeneous Scheduling with Improved Task Priority，HSIP）算法是HEFT算法的一种优化。不用于HEFT的是，HSIP根据目前异构计算的近况，创造性地提出了一种针对异构计算平台异构性的改进，并给出了相应的数学公式描述及改进算法。HSIP算法在此仅做简单介绍，由于后续章节与该算法有关，故该算法的详细介绍放在随后的章节。



图2.1 表调度算法流程图

Fig. 2.1 Flow chart of list scheduling algorithm

分层算法基于表调度算法，是对其的一种优化。分层算法的基本思想是，将DAG内的任务根据某种策略进行分层，然后在同层内进行局部时间分配以求得局部最优解。常见的分层策略有正向分层和逆向分层[25]。 分层算法用于具有截止的工作流优化问题，将全局的时间分配问题转化为局部优化，并规定同层的任务具有相同的开始时间与截止时间。由于目前云服务都基于时间收费，并且计算速度越快的资源或物理机单位时间的价格越高。将DAG进行分层的目的是，在同一层内，在保证该层截止期的条件下尽量将任务分配给计算速度相对慢的物理机以降低费用。逐层调度下去不仅可以保证DAG能够再其截止期前完成调度，可以充分利用时间窗口优化费用。苑迎春[25]等人提出截止期约束的逆向分层算法（Deadline Bottom Level, DBL）。该算法将DAG的截止期转化为可变的时间窗口以进一步优化费用，并规定同层任务的开始时间由其上一层任务决定。刘灿灿等人[26-27]考察工作流的时序特征，并在分层调度过程中收集“时间碎片”以进一步优化费用，并在此基础上提出基于时序一致的截止期约束逆向分层算法，即TCDBL算法。

### 2.2.2 多DAG调度算法

多DAG调度算法可根据所调度的DAG特性分为静态调度算法和动态调度算法。异构系统中多DAG调度管理系统的模型如图2.2所示。云服务提供商会同时为多个租户提供软件服务，以利用规模经济效应，通过在租户间共享人力、设备等资源来降低成本、提高收益[28]。而这种多租户运营模式必然会导致多个租户之间的资源竞争或抢占。考虑到多DAG调度的复杂性，诸多研究者对此进行了研究并针对不同的场景提出了诸多调度策略。

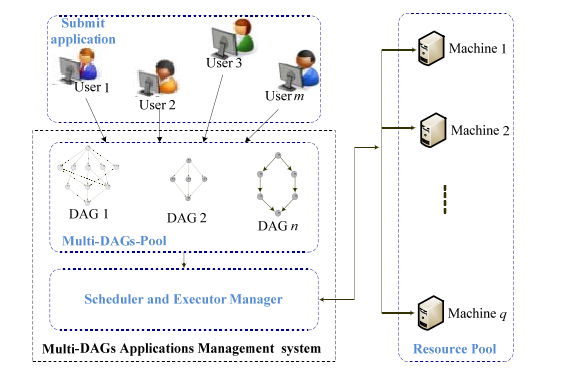


图2.2 异构系统中多DAG调度管理系统模型

Fig. 2.2 The model of scheduling multi-DAGs in heterogeneous systems

针对多DAG的调度，考虑到多个租户或者说多个DAG之间的调度公平性，由于调度目标和调度平台的复杂性，导致多DAG调度问题比单DAG调度问题更加复杂，尤其是多个DAG之间的调度优先级问题。图2.3展示了具有多优先级的多DAG调度模型。Henan Zhao等人[18]提出一种调度将一组静态DAG在一个异构系统中进行公平调度的算法：Fairness算法。该策略给出了一种量化一组DAG中每个DAG滞后程度的方法，在调度过程不断更新所有DAG的滞后程度（*slowdown*），并在调度完成一个任务后为滞后程度最高的任务分配计算资源以进入调度。该算法能够在一定程度上解决异构系统中多个DAG调度中的公平性问题。在此之后有诸多研究者对其进行了优化。由于Fairness算法只能调度静态多DAG，无法适用于现今的诸多调度系统，田国忠等人[29]在Fairness算法的基础上，提出E-Fairness算法。E-Fairness算法的主要改进在于，针对动态DAG，通过修改了计算每个DAG的滞后程度来修改优先级计算方式，使得新到达的DAG能够进入系统并加入调度。E-Fairness算法的改进主要以下几个方面：其一是当有DAG新到达时，根据其到达时间计算优先级，然后撤销系统中所有优先级低于该DAG中的所有任务以便与新到达的DAG一同调度；其二是新到达的DAG中优先级最高的任务可以得到一次调度机会，然后计算其相对滞后程度并决定该DAG的调度优先级，与其他DAG一同参与调度。

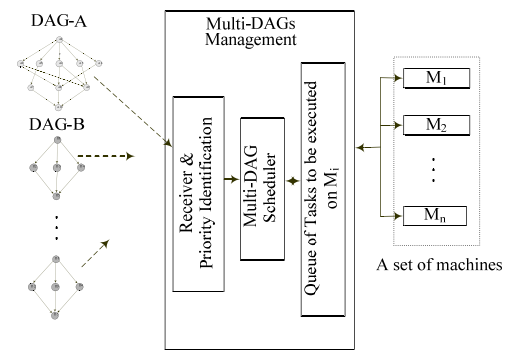


图 2.3 多优先级的多 DAG 调度模型

Fig.2.3 Scheduling model for Multi-DAGs with multiple priorities

Chih-Chiang Hsu[30]提出一种新的调度混合并行工作流的在线工作流管理系统OWM（Online Workflow Management）。该管理系统在调度工作流时主要有4个步骤：首先根据所有DAG的关键路径长度进行排序，然后不断地从中选取任务进入等待序列直到有未准备好的任务；然后，如果需要调度的任务均来自同一DAG，则对其进行降序排列，否则进行升序排列；最后按照RANK\_HYBD[31]算法执行调度。田国忠等人[32]提出了一种具有多优先级的多DAG混合调度策略MMHS（(hybrid scheduling strategy for multiple DAGs workflow with multiple priorities, MMHS）。MMHS调度策略的主体基于两部分，上述的E-Fairness算法以及文献[32]中提出的多DAG的Backfill算法。其中，多DAG的Backfill算法的主要思想是，由于高优先级的DAG可以中断相对低优先级的DAG的执行，因此公平性的比较不应该比较不同优先级之间的DAG；当新到达的DAG优先级较高时，采用抢占式的方式中断比它优先级低的DAG，然后按照HEFT算法执行其内部的任务。而MMHS调度策略根据不同的场景选用两种多DAG调度策略，而E-Fairness和Backfill都使用经典的HEFT算法进行单个DAG内部所有任务的调度。当有新到达的DAG时，先根据HEFT算法计算出该DAG内部所有任务的向上权值Rank，然后计算该DAG的优先级。当其优先级与目前系统中正在调度的DAG相同时，根据E-Fairness算法调度这两个优先级相同的DAG；若大于，则撤销其他任务，根据HEFT算法分配资源并调度执行新到达的DAG；若小于则根据Backfill算法处理该组DAG的调度顺序问题。

## 2.3 DAG工作流优化模型

基于DAG的工作流调度模型一般由两种优化目标：调度时间和调度费用。两者是互相矛盾的，调度时间短意味着需要使用相对计算速度快的物理机。而基于费用优化时间的调度策略，通常是用户给定调度成本，云服务提供商基于成本给出最快捷的调度方案，即成本约束下的工作流的时间优化问题。

另一种是用户在提交任务时，给定一个截止期，以费用最小为调度目标。这种类型的调度一般要求用户给定的截止期是合理的。调度策略通过合理地选择相对廉价的计算资源以降低费用，即截止期约束下的费用优化问题。针对这一问题诸多研究者进行了研究。本文之前章节所述的算法中，有很多都是处理截止期约束下的费用优化问题，如DBL、TCDBL等。然而，这些分层算法都忽视了一个重点因素：通信开销。由于云环境的特性，各个物理机之间的通信开销无法忽视。因此上述算法都无法处理真实云环境中的调度问题。

除了上述两种针对性的调度算法之外，还有一些多维优化算法。这种调度策略往往是在时间与费用之间找一个合理的平衡点。著名的有基于分层算法的BLSuff、BLMax等。这两种算法都是对DAG进行分层，并且基于优先级因子，在调度一层任务前综合衡量这一层任务的特点以及时间窗口，以满足用户调度目标为目的。然而，这些算法与DBL、DTL等类似，都忽视了通信开销而无法应用于真实云环境中的调度问题。

3 DAG带通信开销的工作流优化模型简介

## 3.1 CA-DAG模型介绍及相关基本定义

本课题组提出了一种通信感知的工作流费用优化模型，CA-DAG[32]。该模型同样使用DAG来描述工作流，并在此基础上加入了通信开销，来描述云环境中多个物理机之间进行数据传递的费用。在此之前，多个物理机之间的通信开销通常使用静态方式进行描述，即对于*q*个物理机，使用一个*q×q*的一个矩阵*B*来表述每个物理机间的通信开销，使用一个向量*L*表述所有物理机的数据传输启动时间。可以看出，这种方式需要在调度前确定所有物理机之间的通信开销。一个具有*v*个任务的DAG在一组具有*q*个物理机的集合*Q*上进行调度时，假设所有任务在所有物理机上执行的运行时间已知，并使用一个*v×q*的矩阵*W*表述。任务在所有物理机上的平均执行时间可以用公式3.1描述：

（3.1）

DAG内部两个任务、调度时之间的平均数据传输时间为，其中为所有物理机的平均数据传输启动时间，为不同物理机之间的数据传递时间。由该定义可以看出，传统意义上的模型只考虑了不同物理机间数据传递的时间开销，而忽略了该过程中的费用开销，包括传输数据所需的费用以及控制这一流程所需的费用。基于这一点，CA-DAG将不同物理机之间的通信开销加入模型，该模型可以用DAG图描述，图3.1和表3.1给出了一个CA-DAG的例子。一般情况下，一个DAG都分别只有一个入口节点和一个出口节点（如果没有则可以相应地选择加入虚拟头结点或虚拟尾结点）。*V*代表所有任务的集合。图中有向边的集合描述了所有任务之间的关系。*E*中的边表示一条由任务指向任务的有向边。此时称为的直接前驱结点，为的直接后继结点。所有的前驱结点构成的前驱集合，用表述。所有的后继结点构成的后继集合，用表述。有向边的指向描述了两个任务和之间的调度顺序，即任务的调度要在完成之后。通常这是因为两个任务之间存在控制关系或者数据依赖关系。用描述的大小，指的是两个任务间的数据传输量。由于任务在多个物理机之间调度执行，网络带宽等诸多外界因素都会造成影响。本模型为简化问题将其假设为固定值，因此这个可以用这个值描述两个任务间的数据通信量。

表3.1 图3.1中各任务的服务集合

Tab. 3.1 The Services Set of the Activities in Fig.3.1

|  |  |  |
| --- | --- | --- |
| Task |  |  |
| 1 | {<3,9,2>,<4,6,3>,<8,2,1>} | 3 |
| 2 | {<4,8,3>,<7,5,1>} | 2 |
| 3 | {<2,9,1>,<4,7,2>,<6,5,4>,<9,2,3>} | 4 |
| 4 | {<2,7,1>,<3,5,4>,<5,3,2>} | 3 |
| 5 | {<3,8,4>,<4,7,1>,<9,2,2>} | 3 |
| 6 | {<4,7,3>,<6,5,1>,<7,3,4>} | 3 |



图3.1 一个工作流例子

Fig.3.1 The instance of workflow(CA-DAG)

CA-DAG模型使用来描述一个需要调度的工作流。不同于其他模型的是，其中*S*表示的是服务，指的是云环境中的各种资源，即Services。*S(i)*表示任务的可选服务集合。相应的，*S(i)*表示任务的可选服务数量。为了简化调度问题，本模型假设一个任务的可选集合中的任何一个任务都可以完成该任务，不存在服务质量的差别。任务的一个服务使用一个三元组表示，。其中，为使用该服务执行任务所需的时间；为使用该服务执行任务的费用，一般意义上服务执行任务所需时间越短，花费也越高；引入用来标识该服务所在的物理机。当两个具有依赖关系的任务选择的服务分别在不同的物理机上时，此时通信开销无法忽视。为了考虑不同服务之间可能存在的通信开销，在调度时根据当前任务所选的服务所在的物理机标识，在调度中对任务所选的服务的物理机标识进行判断，如果该标识与某个和该任务有数据依赖关系的任务所选的服务的物理机标识一致，则两者不产生通信开销；如果需要说明的是，是否产生通信开销以及通信开销的多少都是在调度过程中动态决定的。两个具有依赖关系的任务、分别选择服务,，那么这两个任务间的通信开销可以定义为：

(3.2)

使用及 分别表示任务在物理机上的最早开始时间和最早完成时间。根据定义，入口节点的最早开始时间为0。对于其他任务而言，可以用以下两个公式描述：

(3.3)

(3.4)

其中，表示物理机可以开始的时间，表示任务的完成时间。由公式（3.3）可以看出，一个任务的最早开始时间由它所选的服务和其前驱任务有关，而内侧的max表达式表示任务的所有前驱都已完成并且需要的数据已传送到物理机的时间。任务的最早完成时间取决于所选的物理机和该任务在该物理机上的最早开始时间。用表示DAG的出口结点，那么该DAG的调度长度可以如下表示:

（3.5）

由公式（3.5）可以看出，DAG的调度长度为其出口任务的调度完成时间。对于有多个出口任务的DAG，为了便于统一调度，上文提到添加一个虚拟出口节点，这种DAG的调度长度实际上是所有出口任务的调度完成时间的最大值。

文献[23]中的提出了一种计算DAG内所有任务优先级的方法，向上权值ranku。该文献中提出的HEFT算法就是根据这一优先级决定所有任务的调度顺序。一个任务的向上权值可以由出口节点开始，递归地定义为：

（3.6）

出口任务的向上权值定义为该任务在所有物理机上的执行时间的平均值。从公式（3.6）可以看出，向上权值越大的任务其调度优先级越高。由于该定义是由出口任务开始进行递归地定义，因此入口任务具有最高的向上权值，这和通常意义上入口任务具有最高调度优先级符合。

## 3.2 异构系统中多DAG公平调度算法简介

### 3.2.1 公平性算法Fairness

在此之前，常见的多DAG调度算法是DAG合成算法。DAG合成算法首先构造一个虚拟的头结点以及尾结点，然后将需要调度所有的DAG连接到虚拟的头结点，使得所有DAG都是该虚拟头结点的直接后继结点。然后针对不同的调度目标使用相应的调度策略，将多DAG调度问题转化为单DAG调度问题。然后由于各个DAG之间的结构存在很大差异，使用单DAG调度算法难免会引起公平性问题，导致某个或某些DAG等待太久或者产生“饿死”现象。Henan Zhao等人[18]提出的Fairness算法在一定程度上解决了异构系统中多DAG调度的公平性，并提出了一系列度量调度算法公平与否的标准。该算法的主要思想是：在每个任务调度完成后更新当前所有DAG的滞后程度并根据HEFT算法调度滞后程度最高的DAG的一个任务。本文在此不做详述，仅给出相关定义：

（1）给定一组DAG中的一个DAG，a，的滞后程序可以如下定义：

（3.7）

其中，指的是*a*单独在这组资源上进行调度时的调度长度，即*Makespan*;而指的是*a*在和这组资源进行混合调度时的完成时间。是个介于0-1之间的数字，越大表明其滞后程度越小，混合调度时的完成时间与单独调度时相差不大；反之则表明该DAG进行混合调度时滞后程度大。

（2）针对一个多DAG调度策略，提供了一种衡量该调度策略公平性的方法。一个调度策略*S*的不公平程度可以定义为：

（3.8）

（3.9）

其中，表示当前所有DAG的平均滞后程度；|A|表示该组DAG的数量。

### 3.2.2 带通信开销的工作流调度优化算法

本课题组基于CA-DAG模型，提出一种带通信开销的DAG工作流费用优化算法CACO。由于本文后续章节与该算法有关，在此做简单介绍并给出相关概念的定义。

CACO算法基于CA-DAG模型。CACO第一步利用提出的FC原则在考虑通信开销的基础上确定最小完工时间。FC原则是指：采用递归的方法确定每个任务的最快服务（使任务最早完成的服务）及最早完成时间，仅当此任务的所有前驱任务的最快服务都确定下来才能确定此任务的最快服务。CACO调度分为两个阶段：分层阶段和调度阶段。分层阶段按照DBL的逆向分层方法，求解每一层的时间窗口，进一步求得最小完工时间，也即最后一层的层截止时间。的计算方式由公式（3.10）给出。  
  （3.10）

其中，是在时计算出来的；是第m层的层截止时间；是第*i*层的最早开始时间。下面给出CACO算法的分层阶段和调度阶段的伪码描述。

**算法1.**CACO分层阶段.

将带入公式（3.10），求得各任务的时间窗口；

；

；



利用求得的重新计算所有任务的时间窗口；



**算法2.** CACO调度阶段.

输入：工作流及各任务的候选服务集合以及用户给定的截止期；

按照FC计算；

根据算法1对待调度DAG进行分层，并计算；



无法找到满足时间约束的调度方案；



选择计算速度最快服务，计算整体调度费用；



计算冗余时间，按照式（8）确定各任务的时间窗口；

对各任务进行调度，调度过程中按照式（9）迭代确定各任务的开始时间及结束时间，从各任务的服务吃池中选择最优服务，计算出调度费用；



返回调度结果.

当给定截止期时，可以通过选择相对“便宜”的服务来降低费用。为了充分利用各层任务时间窗口内的“时间碎片”，CACO调度阶段采用动态规划的方法调整所有任务的最终最早开始时间和最晚结束时间，任务*i*的最优服务使用表示，若任务*i*选择该任务能得到当前局部最优解。CACO的时间复杂度是*O(v2)*，*v*为任务数，而先前基于分层算法的时间复杂度也为*O(v2)*，可见CACO算法做了优化单并没有改变时间复杂度。

### 3.2.3 改进优先级策略的异构系统上的调度策略

本课题组提出一种针对异构系统的HEFT优化算法HSIP（Heterogeneous Sche-duling with Improved Task Priority，HSIP）[24]。由于本文后续章节与该算法有关，在此做简单介绍并给出相关概念的定义。

HEFT算法被广泛应用于调度系统，然后它仍有优化空间。其一，HEFT算法以及今年来的诸多基于HEFT的改进算法都忽视了计算资源上的异构度以及各个资源间的通信开销；其二，现有的算法大都采用入口任务复制策略，而这会导致过度增加CPU的负载。基于此，本课题组提出一种改进算法HSIP。HSIP调度策略分为两个步骤，任务优先级划分以及处理器分配。

在任务优先级划分阶段，HSIP算法引入OCCW（Out-degree communication costweigh）到优先级确定公式中，基于公式（3.6），提出一种新的优先级计算方式。任务的优先级可以递归地定义如下：

(3.11)

(3.12)

其中，是任务在服务池中可选服务的计算花费的标准差。不同与别的算法，以标准差而不是平均值来定义能获得更好的优化结果。这是因为，当一个任务的多个服务间的计算开销相差很大时，标准差会是一个相对较大的值，进而使得该任务获得较大的优先级。然而，从数量级上看，是比小得多的量。因此，将与相乘，这样使得需要较大数据传输时间的任务也会获得相对较大的优先级。值得一提的是，针对公式（3.6）以及出口任务，进行 补充定义。

在HSIP的处理器分配阶段，文献[24]中还提处了一种对插入机制（Insertion-Based strategy）的数学描述。插入机制最先由文献[23]提出，针对处理器上调度完成一个任务与调度下一个任务之间的空闲时间间隙（idle time slots, ITS）进行插入。此后诸多研究者均采用了这一策略。HEFT算法在调度过程中，当发现有空闲时间间隙时，不做过多考虑而直接选择第一个ITS，这显然会影响调度结果。HSIP算法对这一策略进行了改进，当发现有空闲时间间隙时，选择最快完成的。改进后的插入机制基本思想是：当完成一个任务时，更新所有处理器的空闲时间列表；当需要对任务进行处理器分配时，遍历该列表寻找能够在空闲时间内执行任务的处理器；当有多个空闲时间间隙满足条件时选择能够使得任务最早完成的处理器。同时，文献[24]还对这一过程进行了精确的数学描述，在此不做赘述。

HSIP的调度过程伪码下所示。

输入：DAG，一组任务*V*,一组处理器*P*

输出：调度结果，调度长度(Makespan)

⑴从出口任务开始，递归地计算根据改进任务优先级策略计算计算所有任务的向上权值

⑵将所有待调度列表中的任务根据其进行降序排序

⑶While 仍有未调度的任务

⑷ 从待调度列表中选择第一个任务

⑸ If任务是入口任务

⑹ 使用入口任务选择副本策略

⑺ Else(任务不是入口任务)

⑻ if满足插入机制优化策略所需的条件

⑼ 使用空闲时间间隙处理器分配策略

⑽ else

⑾ for do

⑿ 根据公式（3.4）及（3.13）计算任务的最早开始时间EFT

⒀ end

⒁ 将任务分配到使其EFT最小的处理器上调度执行

⒂ End if

⒃ End if

⒄ 从待调度列表中删除任务，更新列表

⒅End While

4 DA

## 4.1 第一节题目（黑体，四号，1.5倍行距，段前0.5行）

### 4.1.1 第一节一级题目（黑体，小四，1.5倍行距，段前0.5行）

## 4.2 第二节题目

### 4.2.1 第二节一级题目

结 论

参 考 文 献

附录A 附录内容名称

攻读硕士学位期间发表学术论文情况

致 谢