基于局部关键路径与截止期限分配的云工作流调度算法

蔡艳婧[1,2] 王 强[1*] 程 实[3]

[1](南通大学电子信息学院 江苏 南通 226019)

[2](江苏商贸职业学院电子与信息学院 江苏 南通 226001)

[3](南通大学计算机科学与技术学院 江苏 南通 226019)

摘 要 为了解决云计算中截止期限约束下的工作流调度代价优化问题,提出一种基于局部关键路径和截止期限分配的工作流任务调度算法。为了满足期限约束,并最小化执行代价,算法将工作流任务的调度过程划分为两个阶段:期限分配阶段和调度资源选择阶段。期限分配阶段定义工作流的局部关键路径,并以递归的方式在局部关键路径上的任务间进行子期限分配;调度资源选择阶段在满足任务子期限的同时,为每个任务选择执行代价最低的资源进行任务调度,以实现调度代价优化。分析算法的时间复杂度,并通过一个算例对算法的实现思路进行了详细阐述。通过科学工作流结构的仿真实验,证明了算法不仅可以满足截止期限约束,而且可以降低工作流任务的执行代价。

关键词 云计算 工作流调度 期限分配 局部关键路径 期限约束 代价优化

中图分类号 TP393 **文献标识码** A DOI:10.3969/j.issn.1000-386x.2019.08.038

CLOUD WORKFLOW SCHEDULING ALGORITHM BASED ON PARTIAL CRITICAL PATH AND DEADLINE DISTRIBUTION

Cai Yanjing[1,2] Wang Qiang[1*] Cheng Shi[3]

[1](*School of Electronic Information, Nantong University, Nantong 226019, Jiangsu, China*)

[2](*School of Electronics and Information, Jiangsu Vocational College of Business, Nantong 226001, Jiangsu, China*)

[3](*College of Computer Science and Technology, Nantong University, Nantong 226019, Jiangsu, China*)

Abstract For optimizing workflow schedulimg cost under deadline constraint in cloud, we presented a workflow scheduling algorithm based on partial critical path and deadline distritution. For meeting the deadline constraint and minimizing the execution cost, our algorithm divided the workflow scheduling process into two steps: the deadline distribution stage and the scheduling resource selection. In the deadline distribution stage, we defined the partial critical path of the workflow and recursively assigned sub-deadlines to the tasks on the partial critical path. In the scheduling resource selection, we assigned the cheapest resource to each task to optimize the scheduling cost while meeting its sub-deadline. The time complexity of our algorithm was analyzed and an example was designed to elaborate the implement idea of our algorithm. Through the simulation experiments of scientific workflow, it is proved that our algorithm can better reduce the workflow execution cost under meeting deadline.

Keywords Cloud environment Workflow scheduling Deadline distribution Partial critical path Deadline constraint Cost optimization

---

收稿日期:2018-12-25。国家自然科学基金项目(61602267);江苏省高校高端研修资助项目(2018GRFX022);南通市科技计划(指导性)项目(MSZ18080);江苏商贸职业学院课题(SY20171212-07)。**蔡艳婧**,副教授,主研领域:大数据,智能算法,移动计算。**王强**,副教授。**程实**,实验师。

0　引　言

工作流结构广泛应用于复杂计算问题建模,云计算特有的按需提供和付即用的定制资源使用方式使其成为调度工作流的有效方法[1]。与传统批任务调度不同,工作流结构的任务具有严格的逻辑执行次序,需要在满足给定 QoS 约束的同时,实现与资源间的映射。工作流调度通常由选择被调度任务和选择提供实例两个阶段构成,两阶段决策对于是否能够满足给定约束和全局调度代价均具有重要影响。传统工作流调度方法仅注重执行效率/时间,忽略了资源使用的费用,此时的调度问题在不同资源和不同调度方案下的执行时间和代价均有所不同。因此,同步考虑用调度时间和代价更加符合云资源的使用环境。

为了解决期限约束时的工作流调度代价优化问题,本文设计的调度方法是将全局期限在所有工作流任务上进行分割,以得到任务的子期限,然后在实例提供时仅满足子期限即可。

1　相关研究

对于工作流调度过程的调度与提供两个阶段[2],给定资源集,调度阶段的目标是决定任务执行的最优序列和与用户相应约束下的任务部署[3];提供阶段的目标是为工作流内的任务选择资源类型和相应资源数量,并为任务执行预留资源[4-5]。相关研究中,底向下分层[6](Down-Buttom Level, DBL)和底向上分层[7](Down-top Level,DTL)算法是典型的基于期限分配的启发式调度算法。前者以 down-top 方式对任务进行划分,后者则以 top-down 方式对任务进行划分。由于工作流可以有向无循环图建模,故 DBL 将任务划分为不同层次,每个层次包含的任务不具有依赖性。而 DTL 将任务划分为不同路径(作为同步任务或简单任务,同步任务定义为拥有一个以上父任务或子任务的任务)。为任务分配期限时,全局期限以正比于每个层次的最小执行时间的方式在各层次间进行分割。然而,在 DBL 算法,首先需要计算最快实例资源,然后再将期限与估算值之差以均匀分布方式在所有层次间进行分配。文献[8]提出了 DET 算法,算法将任务分为两种类型:关键和非关键任务。关键任务利用动态规化进行调度,非关键任务则在关键任务间进行回填式调度,但该算法忽略了任务间的通信时间。

文献[9]提出了云工作流调度算法 PDC,算法将期限以正比于各层次中任务执行时间的方式在层次间进行分割。文献[6,10]提出了最迟完成时间 LFT 算法,该算法也是将期限在各任务间进行分割,并确保工作流在用户定义期限条件下,执行完任务的最早时间。文献[10]提出了局部关键路径算法,算法可以根据任务在工作流中所处的局部关键路径对任务进行分类,同时,期限根据定义的路径进行重分配。然而,算法在每个局部关键路径 PCP 执行后,需要重新计算最迟完成时间,开销较大。文献[6]提出了基于动态代价最小的联合内层技术(Joint Inner Technology,JIT)算法,该算法在期限约束下将联合管道任务集建立为单个任务,以消除任务间的数据传输时间。然而,该算法在任务执行实例的选择上并非最优,有待改进。

2　任务调度模型

云工作流模型以有向无循环图 DAG 建模,表示为 $G(T,E)$,T 为 n 个任务 $\{t_1,t_2,\cdots,t_n\}$ 的集合,E 为边集。每条边 $e_{i,j}=(t_i,t_j)$ 表示任务间的执行次序约束,代表 t_i 必须在 t_j 开始前完成执行。对于给定的 DAG,不存在任一父节点的任务称为入口任务 t_{entry},不存在任一子节点的任务称为出口任务 t_{exit}。由于本文的算法要求应用模型具有单一入口和出口任务,故可以分别在任务 DAG 的开始和结尾处增加一个傀儡任务 t_{entry}和 t_{exit}。傀儡任务的执行时间为 0,且与其他任务的连接边上的权值也为 0。

云资源模型由多个云服务提供者组成,每一个均可向用户提供资源。每一个任务 t_i 可由拥有不同 QoS 属性的不同服务提供者的 m_i 种服务 $S_i=\{s_{i,1},s_{i,2},\cdots,s_{i,mi}\}$ 进行处理。本文关注的 QoS 属性为执行时间和代价,服务代价取决于执行时间,即执行时间越短,代价越高。令 $ET(t_i,s)$ 表示在服务 s 上处理 t_i 的估计执行时间,$EC(t_i,s)$ 表示在服务 s 上处理 t_i 的执行代价。令 $TT(e_{i,j},r,s)$ 和 $TC(e_{i,j},r,s)$ 分别表示服务 r(执行任务 t_i)与服务 s(执行任务 t_j)间的边 $e_{i,j}$上的估计数据传输时间和传输代价。

3　算法设计

3.1　算法实现思路

基于局部关键路径和截止期限的云工作流调度算法(Workflow Scheduling Based on Partial Critical Path and Deadline Constraint,WS-PCPDC)包括两个阶段:期

限分配与资源选择。期限分配阶段中,全局任务 DAG 的截止期限在个体任务间进行分配,若每个任务可在其子期限内完成,则整个任务 DAG 可在截止期限内完成。资源选择阶段中,在满足任务子期限的同时,为每个任务选择最优资源完成任务调度,实现代价最优。

3.2 基本定义

对于每个未调度任务 t_i,令 $EST(t_i)$ 表示任务 t_i 的最早开始时间,该时间是未考虑实际执行该任务的资源时得到的时间。显然,无法计算准确的 $EST(t_i)$,由于云环境是异构环境,任务执行时间在不同资源间是变化的。进一步,数据传输时间也取决于所选资源及资源间的传输带宽。任务 t_i 的最小执行时间 $MET(t_i)$ 和最小传输时间可分别定义为:

$$MET(t_i) = \min_{s \in S_i} ET(t_i, s) \tag{1}$$

$$MTT(e_{i,j}) = \min_{r \in S_i, s \in S_j} TT(e_{i,j}, r, s) \tag{2}$$

基于以上定义,最早开始时间可定义为:

$$EST(t_i) = \begin{cases} 0 & t_i = t_{\text{entry}} \\ \max\limits_{t_p \in pred(t_i)} (EST(t_p) + MET(t_p) + MTT(e_{p,i})) & t_i \neq t_{\text{entry}} \end{cases} \tag{3}$$

式中,$pred(t_i)$ 表示 t_i 的父节点任务。

对于每个未调度任务 t_i,令 $LFT(t_i)$ 为整个任务 DAG 在截止时间 D 内保证完成时,任务 t_i 能够完成执行的最迟时间,则:

$$LFT(t_i) = \begin{cases} D & t_i = t_{\text{exit}} \\ \min\limits_{t_c \in succ(t_i)} (LFT(t_c) - MET(t_c) - MTT(e_{i,c})) & t_i \neq t_{\text{exit}} \end{cases} \tag{4}$$

对于每个调度任务 t_i,令 $SS(t_i)$ 为执行 t_i 的所选资源,$AST(t_i)$ 为任务 t_i 在资源上的实际开始时间。

3.3 算法步骤

算法 1 是 WS-PCPDC 算法的伪代码。步骤 3 添加两个傀儡节点至任务 DAG 中,步骤 4 – 步骤 7 计算所需参数值,步骤 8 为节点 t_{entry} 和 t_{exit} 分配子期限,并在步骤 9 中将这两个任务标记为已分配(assigned)节点。已分配节点表明该任务节点已经分配子期限,未分配子期限的节点称为未分配(unassigned)节点。可以看出,t_{exit} 的子期限设置为截止期限 D,说明出口任务必须在 D 内完成。步骤 10 对出口任务调用 AssignParent 算法(分配节点算法),该算法的目标是为输入节点的所有未分配父节点分配子期限,从出口任务 t_{exit} 开始分配即可保证为 DAG 中的所有任务分配子期限。因此,AssignParent 算法负责在所有任务间分配全局截止期限。最后,步骤 11 调用 Planning 算法,用于在满足子期限的情况下为每个任务选择执行资源。

算法 1 WS-PCPDC

(1) Procedure ScheduleWorkflow($G(T,E)$, D)
(2) Request available resource for each task in G //发出资源请求
(3) add t_{entry}, t_{exit} and their correeponding edges to G //添加进出口任务
(4) using Eq. (1) to compute $MET(t_i)$ for each task //为每个任务计算 $MET(t_i)$
(5) using Eq. (2) to compute $MTT(t_i)$ for each edge //为每条边计算 $MTT(t_i)$
(6) using Eq. (3) to compute $EST(t_i)$ for each task in G //为每个任务计算 $EST(t_i)$
(7) using Eq. (4) to compute $LFT(t_i)$ for each task in G //为每个任务计算 $LFT(t_i)$
(8) sub-deadline(t_{entry}) = 0, sub-deadline(t_{exit}) = D //为入口出口任务计算子期限
(9) mark t_{entry} and t_{exit} as assigned //标识入出口任务已调度
(10) call function AssignParent(t_{exit}) //调用 AssignParent(t_{exit})
(11) call function Planning($G(T,E)$) //调用 Planning($G(T,E)$)
(12) end procedure

3.4 AssignParent 算法

AssignParent 算法伪代码如算法 2 所示。算法输入一个已分配节点,并尝试分配子期限至其所有父节点上。AssignParent 算法(分配节点算法)首先需要寻找终止于输入的未分配节点的局部关键路径。

算法 2 AssignParent

(1) Procedure AssignParents(t)
(2) while (t has an unassigned parent) do //若 t 有未调度父节点
(3) $PCP \leftarrow$ null, $t_i \leftarrow t$ //局部关键路径置空
(4) while (there exists an unassigned parent of t_i) do //若仍有未调度父节点
(5) add CriticalParent(t_i) to the beginning of PCP //添加任务的关键父节点至 PCP 的队首
(6) $t_i \leftarrow$ CriticalParent(t_i) //提取当前任务
(7) call function AssginPath(PCP) //调用函数 AssginPath(PCP)
(8) for all ($t_i \in PCP$) do //每个局部关键路径上的任务更新 EST 和 LFT
(9) update EST for all unassigned successors of t_i //更新所有未调度子任务的 EST
(10) update LFT for all unassigned predecessors of t_i //更新所有未调度父任务的 LFT
(11) end procedure

以下定义关键父节点的概念:

定义 1　任务 t_i 的关键父节点为数据到达时间最迟的未分配父节点,即:满足下式的 t_i 的父节点 t_p:

$$\max_{t_p \in pred(t_i)} (EST(t_p) + MET(t_p) + MTT(e_{p,i}))$$

定义 2　任务节点 t_i 的局部关键路径为:

1) 若 t_i 不存在任一未分配父节点,则为空;

2) 若 t_i 存在任一未分配父节点,则由其关键父节点 t_p 和 t_p 的局部关键路径组成。

算法 2 由输入节点开始,沿关键父节点直到到达没有未分配父节点的节点任务,以形成一条局部关键路径。第一次调用该算法时,由 t_{exit} 开始,向回追溯其关键父节点,直到到达 t_{entry}。因此,算法可以找到穿越整个 DAG 的全局关键路径。然后,算法调用 AssignPath 算法(分配路径算法),该算法接收一条路径(任务节点序列)作为输入,在任务的最迟完成时间内分配子期限至路径上的每个节点上。当子期限分配至任务后,其未分配后继节点的 *EST* 和其未分配父节点的 *LFT* 可能发生改变(根据式(3)和式(4))。基于此原因,算法需要在下一次循环中更新路径上所有任务的这两个值。最后,算法通过递归调用 AssignParent 为局部关键路径上的每个任务节点的父节点分配子期限。

3.5　AssignPath 算法

AssignPath 算法接受一条路径作为输入,分配子期限至其上的每个任务节点。本文设计三种分配策略,试图为路径创建预计调度方案,并使用算法为路径上的任务分配子期限。由于仅是估计调度方案而非实际方案,故未考虑资源的可用时间。

1) 最优策略。该策略试图寻找在最迟完成时间内执行路径上任务的代价最小调度,然后利用该最佳调度分配子期限至路径上的任务。策略过程如算法 3 所示。该算法基于回溯法,从路径上的第一个任务开始,向最后一个任务遍历,在每一步中为当前任务选择下一个更慢的资源,即步骤 5。因此,对于每个任务的资源是从最快至最慢进行检测。如果没有可用未检测资源剩余,或分配当前任务 t 至下一个更慢资源 s 是不可行分配,则算法回溯至路径的前一任务,并为其选择另一资源,即步骤 6 - 步骤 8。如果任务 t 能够在其最迟完成时间内在资源 s 上完成,即 $EST(t) + ET(t,s) \leqslant LFT(t)$,则定义为一次可行分配。

步骤 9 - 步骤 10 中,算法检测当前任务是否为路径上的最后一个任务,且当前分配是否拥有比最优分配更低的代价,若满足,在步骤 11 中设置当前调度为最优 best 调度。While 循环结束后,步骤 18 检测是否找到最优调度,由于路径上某些任务可能在其 LFT 内得到调度,所有可能存在最优调度不存在的情形。如果不存在最优调度,则在步骤 19 中将任务的 *EST* + *MET* 值作为路径上的任务的子期限值分配,否则,根据最优调度 best 为路径 path 上的所有任务设置 *EST* 和分配子期限。

最后,可能存在额外时间,即最后一个任务的 *LFT* 与其分配的子期限之间存在差值,可将其添加至路径上的任务的子期限上。当该额外时间值小于最小值时,可将其添加至最后一个任务的子期限上。若该值较大,可以正比于传输时间减去执行时间的方式将其分配至路径上的任务。

算法 3　Optimized PathAssigning Algorithm(最优策略算法)

```
(1)  Procedure AssignPath(Path)
(2)  best←null                          //最优集合先置空
(3)  t←first task on the path
                                        //提取当前路径的第一个任务
(4)  while (t is not null) do
(5)  s←next slower service ∈ S_t
                                        //提取下一个最慢的服务提供者
(6)  if (s = assigning t to s is not admissible) then
                                        //若无法分配完成
(7)  t←previous task on the path and continue while loop
                                        //继续在该路径上寻找
(8)  end if
(9)  if (t is the last on the path) then
                                        //若该任务为路径上的最后一个任务
(10) set this schedule as best          //设置该调度解为最优
(11) end if
(12) t←next task on the path            //提取路径上的下一任务
(13) end while
(14) if (best is null) then             //若最优集为空
(15) set sub-deadline(t) = EST(t) + MET(t) for all tasks t on the path
                                        //更新子期限
(16) else
(17) set EST and sub-deadline according to best for all tasks ∈ path
                                        //以最优解为基础设置 EST 和子期限
(18) end if
(19) mark all tasks of the path as assgined
                                        //标记任务是否调度
(20) end procedure
```

2) 代价降低策略。该策略是一种近似最优的贪婪方法,即策略试图以多项式时间寻找一个较优解(不一定为最优解)。策略首先将最快资源分配至路径上的每个任务,显然该分配是代价最高解。然后,试图在不超过任一任务的 *LFT* 的情况下,通过分配代价更低(也更慢)的资源至任务来降低代价。为了决定

哪些任务需要重分配至代价更低的资源,需要计算代价降低率 CDR,定义为:

$$CDR = \frac{TEC(t_i,cs) - TEC(t_i,ns)}{TET(t_i,ns) - TET(t_i,cs)} \quad (5)$$

式中:cs 为已分配至任务 t_i 的当前资源,ns 为比当前资源执行 t_i 更慢的资源。任务 t 在资源 s 上的总执行时间 $TET(t,s)$为在资源 s 上的执行时间 + 任务 t 与其路径上的父节点与子节点间的总传输时间(除不存在父/子节点的第一个和最后一个任务节点)。任务 t 在资源 s 上的总执行代价 $TEC(t,s)$ 的定义方式与上类似。

t_i 的 CDR 可以衡量一个单位时间的代价可以换来多少执行代价的降低。若 t^* 被选择使得它有最大 CDR 值,则该任务是可替换的,即将其分配至下一个更慢资源是一个可行分配。最后,t^* 的当前资源可更改为下一个更慢资源。过程如算法 4 所示。

算法 4 低价降低策略算法

(1) Procedure AssignPath(path)
(2) cur←assign the fastest service to each task of the path //将最慢服务分配给路径上的每个任务得到一个初始解
(3) compute $CDR(t_i)$ for each task of the path by Eq.(5) //计算任务 $CDR(t_i)$
(4) repeat
(5) t^*←null
(6) for all ($t_i \in$ path) do //判定路径的 CDR 值的大小
(7) if ($CDR(t_i) > CDR(t^*)$ and t_i is replaceable) then
(8) $t^* \leftarrow t_i$
(9) end if
(10) end for
(11) if (t^* is not null) then
(12) update cur by assigning t^* to the next slower service //分配次最慢的服务器更新 cur
(13) update $CDR(t^*)$ //更新 CDR
(14) end if
(15) until (t^* is null) //循环至集合为空
(16) if (there is an inadmissible assignment in cur) then //若在 cur 集合中存在一个不允许的调度解
(17) set sub-deadline(t) = $EST(t) + MET(t)$ for all tasks t on the path //更新子期限
(18) else
(19) set EST and sub-deadline according to cur for all tasks ∈ path //以最优解为基础设置 EST 和子期限
(20) end if
(21) mark all tasks of the path as assgined //标记调度任务

3) 公平策略。该策略以公平的方式为路径上的任务节点分配子期限。首先,策略将路径上的每个任务分配至最慢的资源上。然后,从第一个任务至最后一个任务,在不超过任务的 LFT 的情况下,策略以下一个更慢资源替换已分配资源。该过程迭代执行至无法替换为止。策略过程如算法 5 所示,最差情况下,repeat-until循环将执行 m 次,因此,算法时间复杂度为 $O(lm)$。

算法 5 Fair PathAssigning Algorithm

(1) Procedure AssignPath(Path)
(2) cur←assign the fastest service to each task of the path //将最慢服务分配给路径上的每个任务得到一个初始解
(3) for all ($t_i \in$ path) do //对于所有路径上的任务
(4) if (assigning t_i→next service is admissible) then //若将任务调度至下一个服务器可行
(5) update cur by assigning t_i→next slower service //更新 cur 集合
(6) until (no change is done) //直到无法进一步改进
(7) if (there is an inadmissible assignment in cur) then //若存在不可行解
(8) set sub-deadline(t) = $EST(t) + MET(t)$ for all tasks t on the path //更新子期限
(9) else
(10) set EST and sub-deadline according to cur for all tasks ∈ path //以最优解为基础设置 EST 和子期限
(11) mark all tasks of the path as assgined //标记已调度任务

3.6 Planning 算法

Planning 算法(规划算法)即为算法的第二个阶段(资源选择阶段),其目标是为每个任务选择最优资源,在确保满足截止期限的同时,以最小化执行代价调度任务。在期限分配阶段,每个任务被分配一个子期限。如果可以调度每个任务,使得任务在其分配子期限内完成,则整个 DAG 可在截止期限内完成。该阶段算法尝试以贪婪策略通过制定局部最优决策创建全局最优调度解。在每个阶段中,算法选择一个就绪任务,即该任务的所有父任务已经完成调度,然后调度至满足其子期限的价格最低的资源上执行。因此,对于就绪任务 t_i 的所选资源 $SS(t_i)$满足:

$$\min EC(t_i,s) + \sum_{t_p \in pred(t_i)} TC(e_{p,i},SS(t_p),s) \quad s \in S_i$$

约束条件为:

$$AST(t_i,s) + ET(t_i,s) \leqslant sub - deadline(t_i) \quad (6)$$

式中:t_i 在 s 上的实际开始时间 $AST(t_i,s)$为 t_i 的父节点数据到达资源 s 的最迟时间与资源 s 上可用时槽开

始时间的相对大值。

当没有资源可子期限内完成任务 t_i 时(由于子期限仅是估算调度,并未考虑资源上的实际可用空闲时间槽),则选择完成时间最小的资源 $SS(t_i)$,即满足:

$$\min AST(t_i,s) + ET(t_i,s) \tag{7}$$

Planning 算法的伪代码如算法 6 所示。

算法 6　Planning

(1)　Procedure Planning($G(T,E)$)
(2)　Queue←t_{entry}　　//将入口任务输入队列
(3)　while (Queue is not empty) do　　//若队列不为空
(4)　t←delete first task from Queue　　//删除队列首任务
(5)　query available time slots for each service from *CRP*　　//查询资源的可用时隙
(6)　compute $SS(t)$ according to Eq. (6) and (7)　　//计算 SS(t)
(7)　*AST*←the actual start time of t on $SS(t)$　　//更新 AST
(8)　make advance reservation of t on $SS(t)$　　//提前保留
(9)　for all ($t_c \in$ children of t) do　　//对于所有子任务
(10)　if (all parent of t_c are scheduled) then　　//若所有父节点已被调度
(11)　add t_c to Queue　　//添加至队列

3.7　算法时间复杂度分析

假设调度的任务 DAG $G(T,E)$ 包含 n 个任务和 e 条边,可用资源数量为 m,入口任务与出口任务间的最长路径的长度为 l。由于 G 为有向无循环图,则最大边数量为 $(n-1)(n-2)/2$,因此可假设 $e \approx O(n^2)$。调度算法中,步骤 4 计算 *MET* 的时间复杂度为 $O(nm) = O(n^3)$,步骤 5 计算 *MTT* 的时间复杂度为 $O(em^2) = O(n^2m)$,步骤 6 计算 *EST* 的时间复杂度为 $O(n+e) = O(n^2)$,步骤 7 计算 *LFT* 的时间复杂度为 $O(n+e) = O(n^2)$,步骤 11 的 Planning 算法的时间复杂度为 $O(nme) = O(n^3m)$。

对于 Planning 算法,需要为每个任务尝试所有资源以寻找满足子期限的代价最低的资源。每一次尝试中,需要计算任务在资源上的实际开始时间,该过程需要考虑所有父任务及连接边,故时间复杂度为 $O(nme)$。

AssignParent 算法(分配节点算法)为递归过程。第一次在出口任务上调用并在所有 DAG 任务上进行自调用。算法拥有一个 while 循环(步骤 2 - 步骤 14)处理每个节点的入边,故算法将处理所有 DAG 的边。在 while 循环内部,首先需要计算局部关键路径,其时间复杂度为 $O(l)$。然后,算法调用 AssignPath,其时间复杂度取决于所选策略。由于 AssignPath 的时间复杂度取决于 l 和 m,将其考虑为 $g(l,m)$,故 AssignParent 的时间复杂度为 $O(el + eg(l,m))$。对于 AssignPath 算法(分配路径算法),最快的为 Fair 策略,时间复杂度为 $O(lm)$,因此可忽略时间复杂度的 *el* 部分,时间消耗为 $eg(l,m)$。如果替换 e,则时间复杂度为 $O(n^2g(l,m))$。在分配子期限后,AssignParent 需要更新任务的所有未分配子节点的 *EST* 和所有未分配父节点的 *LFT*。最差情况下,一个节点拥有 $n-1$ 个未分配子节点和父节点,因此更新所有任务的 *EST* 和 *LFT* 的时间复杂度为 $O(n^2)$。

在三种不同的 AssignPath 策略下,AssignParent 的时间复杂度分别为 $O(n^2l^m)$、$O(n^2l^2m)$ 和 $O(n^2lm)$。由于 l 是入口任务至出口任务间最长路径的长度,故其最大值为 n(此时为线性 DAG)。此时,AssignParent 的时间复杂度分别为 $O(nm)$、$O(n^4m)$ 和 $O(n^3m)$,这也是整个算法的时间复杂度。

4　算例分析

以图 1 所示 DAG 对算法工作原理进行阐述。算例包括 9 个任务 t_1 至 t_9 和两个傀儡任务 t_{entry} 和 t_{exit}。三种资源 $S_{i,1}$、$S_{i,2}$ 和 $S_{i,3}$,能以不同的 QoS 能力执行任务。表 1 给出任务在不同资源上的执行时间和执行代价。可以看到,对于任务而言,资源越快,代价越高。图 1 中,每条边上的数值既表示数据传输时间,也表示对应传输代价。例如:t_2 与 t_5 间的数据传输时间为 2,则用户的传输代价也为 2,与两个任务所选资源无关。设置 DAG 的全局截止期限为 35。

表 1　执行时间和执行代价

资源	时间	代价	资源	时间	代价	资源	时间	代价
$S_{1,1}$	6	10	$S_{4,1}$	8	10	$S_{7,1}$	5	8
$S_{1,2}$	8	8	$S_{4,2}$	12	5	$S_{7,2}$	9	6
$S_{1,3}$	10	5	$S_{4,3}$	15	4	$S_{7,3}$	12	4
$S_{2,1}$	5	8	$S_{5,1}$	6	9	$S_{8,1}$	5	10
$S_{2,2}$	8	5	$S_{5,2}$	9	8	$S_{8,2}$	8	8
$S_{2,3}$	12	3	$S_{5,3}$	12	5	$S_{8,3}$	10	5
$S_{3,1}$	4	4	$S_{6,1}$	8	12	$S_{9,1}$	6	8
$S_{3,2}$	7	3	$S_{6,2}$	12	6	$S_{9,2}$	12	5
$S_{3,3}$	10	1	$S_{6,3}$	20	4	$S_{9,3}$	15	3

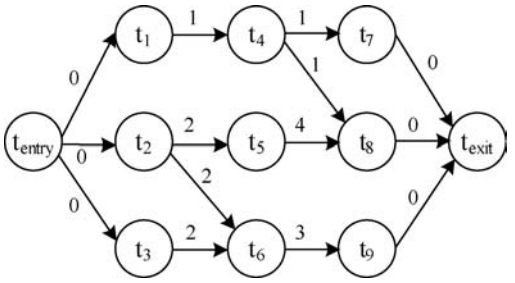


图 1 任务 DAG

利用 WS-PCPDC 算法调度图 1 的 DAG，首先需要将所有任务分配至最快资源上计算 *EST* 和 *LFT*。然后，算法设置 t_{entry} 和 t_{exit} 的子期限分别为 0 和 35，并标记两个任务为已分配。接下来算法需要调用 AssignParent 和 Planning，以下作出讨论。

4.1 调用 AssignParent 算法

首先，在 t_{exit} 上调度 AssignParent 算法，由于该任务有 3 个父节点，步骤 2 的 while 循环将执行三次，将之称为 Step1 至 Step3，后文进行讨论。进一步，需要选择路径分配策略，以最优策略 Optimized 为例，每一步得到的任务的 *EST*、*LFT* 和子期限 *DL* 如表 2 所示。标记 * 表示该值相比前一步骤已发生改变。

Step1 首先，AssignParent 追踪 t_{exit} 的局部关键父节点寻找其局部关键路径，为 $t_2 - t_6 - t_9$。然后，调用 AssignPath 算法（分配路径算法）分配子期限至这些任务。对于以上三个任务共有 27 种可能的资源分配，其中，$S_{2,3} - t_2$、$S_{6,2} - t_6$ 和 $S_{9,1} - t_9$ 为拥有最小代价的最优可行分配，该分配用于决定每个任务的子期限值。下一步即是更新这些任务的所有未分配子节点的 *EST*，即 t_5 和 t_8，及未分配父节点的 *LFT*，即 t_3。变化后的值如表 2 的 Step1。最后一步是在路径上的所有任务上递归调用 AssignParent。由于 t_2 和 t_9 没有未分配父节点，在 Step1.1 中仅需在 t_6 上调用 AssignParent。

Step1.1 当在 t_6 上调用 AssignParent 时，首先寻找该任务的局部关键路径，即 t_3。然后，调用 AssignPath 寻找 t_3 的最优分配，即 $S_{3,3}$。由于 t_3 没有未分配子节点或父节点，Step1 完成。

Step2 现在，回到任务 t_{exit}，AssignParent 试图寻找下一条该任务的局部关键路径，即 $t_5 - t_8$。然后，调用 AssignPath，考虑这两个任务的所有 9 种可能分配，并选择最优可行分配，即 $S_{5,1} - t_5$，$S_{8,3} - t_8$。这两个任务没有未分配子节点，但算法需要更新其未分配父节点的 *LFT*，即 t_1 和 t_4。最后，算法在路径上的所有任务上调用 AssignParent，t_5 没有未分配父节点，因此在 Step2.1 中仅考虑 t_8。

Step2.1 当在任务 t_8 上调用 AssignParent 时，寻找其局部关键路径，即 $t_1 - t_4$。然后，调用 AssignPath，计算该路径的最优可行分配，即 $S_{1,3} - t_1$、$S_{4,2} - t_4$。这两个任务没有未分配父节点，算法需要更新 t_4 的子节点的 *EST*，即 t_7。由于 t_1 和 t_4 没有未分配父节点，Step2 停止。

Step3 在最后一步中，AssginParent 寻找 t_{exit} 的最后一条局部关键路径，即 t_7。AssignPath 寻找最优可行分配为 $S_{7,2} - t_7$，由于没有未分配父节点或子节点，算法停止。

4.2 调用 Planing 算法

在该算例中，Planning 仅将每个任务调度至 AssignParent 算法计算得到的相同资源上。原因在于：数据传输时间是固定的，且资源是完全可用的。这两个假设使得 AssignPath 得到的估计调度方案是实际的调度方案。所选资源如表 2 所示。总时间为 35，总代价为 64，包括执行代价 48 和数据传输代价 16。

表 2 算法每一步的详细计算结果

任务	Initial		Step1			Step1.1			Step2			Step2.1			Step3		
	EST	LFT	EST	LFT	DL	EST	LFT	DL	EST	LFT	DL	EST	LFT	DL	EST	LFT	DL
t_1	0	20	0	20	–	0	20	–	0	15*	–	0	11*	10*	0	11	10
t_2	0	16	0	12*	12*	0	12	12	0	12	12	0	12	12	0	12	12
t_3	0	16	0	12*	–	0	12	10*	0	12	10	0	12	10	0	12	10
t_4	7	29	7	29	–	7	29	–	7	24*	–	11*	24	23*	11	24	23
t_5	7	26	14*	26	–	14	26	–	14	21*	20*	14	21	20	14	21	20
t_6	7	26	14*	26*	26*	14	26	26	14	26	26	14	26	26	14	26	26
t_7	16	35	16	35	–	16	35	–	16	35	–	24*	35	–	24	35	33
t_8	17	35	24*	35	–	24	35	–	24*	35	34*	24	35	34	24	35	34
t_9	18	35	29*	35*	35*	29	35	35	29	35	35	29	35	35	29	35	35

5　仿真分析

5.1　实验配置

为了评估算法性能,本节设计仿真实验对算法进行测试。利用工作流仿真工具包 WorkflowSim 对算法进行实验分析。在 Workflow 平台上配置 10 个异构数据中心,每个数据中心随机配置 10 ~ 100 个资源节点,资源处理能力与代价配置参考 Amazon EC2,单个数据中心的资源拥有相同的处理器速率,数据中心内的资源处理能力约定最快为最慢的 10 倍。数据中心内部的资源间的带宽随机分布于[100 Mbit/s,512 Mbit/s]之间,数据传输代价正比于带宽,即带宽越高,代价越高。

同时,实验为了测试任务规模对算法性能的影响,配置了三种规模的任务数量,小规模为 30 个任务,中规模为 100 个任务,大规模为 1 000 个任务。使用五种不同科学领域中的合成工作流结构作为数据源,包括:(1) Montage 工作流:天文学;(2) Epigenomics 工作流:生物学;(3) SIPHT:生物信息学;(4) LIGO 工作流:引力物理学;(5) CyberShake 工作流:地震学。其结构如图 2 所示[12]。不同工作流形式在其任务关联、数据聚合、数据分布及数据重分布等组成方面均有所不同。Montage 工作流的任务以 I/O 密集型为主,对 CPU 处理能力的要求相对较低,且串行任务结构极少。Epigenomics 工作流的任务以计算密集型为主,且对内存要求较多,串行任务也较多。SIPHT 工作流与 Epigenomics 同为生物学工作流形式,任务类型相似,但 SIPHT 工作流结构更为复杂,串行任务极少。LIGO 工作流的任务多以 CPU 计算密集型为主,且拥有较多内存需求,拥有大量较短的串行任务。CyberShake 任务以数据密集型为主,同时拥有较大内存需求和 CPU 计算能力请求。

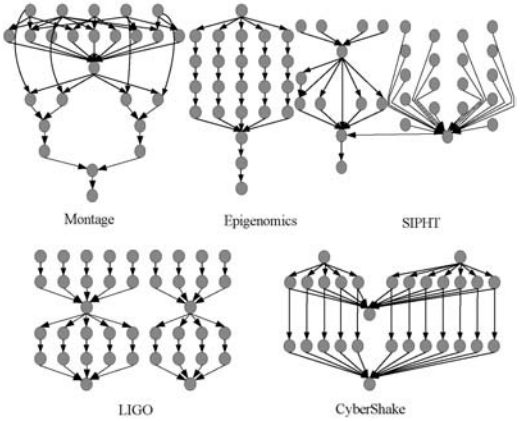


图 2　工作流结构图

5.2　实验结果

利用标准化调度长度 makespan(NM)和标准化代价 cost(NC)对算法性能进行衡量:

$$NM = \frac{schedule\ makespan}{M_{\mathrm{HEFT}}}$$

$$NC = \frac{total\quad schedule\quad cost}{C_{\mathrm{C}}}$$

式中:M_{HEFT}表示利用质构最早完成时间算法 HEFT[13]得到的调度长度,C_{C} 为将所有任务调度至代价最低资源上的调度代价。

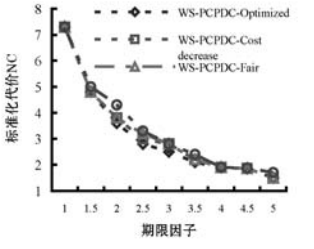
为了评估算法性能,需要将截止期限分配至整个工作流任务。显然,该截止期限须大于或等于 HEFT 算法得到的调度长度。为了设置截止期限,定义一个截止期限因子 α,并设置工作流的期限为其到达时间加上 αM_{HEFT}。实验中 α 值的取值范围为[1,5]。选取的基准算法为 MDP 算法[10]。

表 3 给出了算法的标准化调度长度小于期限因子的平均比例,可以看出,算法均可以在期限约束内完成所有工作流调度,即使期限较紧的情况下(更小的期限因子取值)。对于 LIGO 和 CyberShake 工作流,两种算法几乎利用了所有可用的期限使得执行代价达到最小,即平均差值比例小于 1%。Montage 工作流几乎是同样的情况,而对于 Epigenomics 和 SIPHT 工作流,MDP 算法拥有更高的平均差值比例,分别是中规模 Epigenomics 工作流中的 3.07% 和小规模 SPIHT 工作流中的 5.99%。而本文的三种算法在小规模 SPIHT 中也均有相对更高的差值比例。

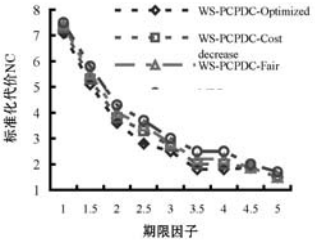
表 3　标准化调度长度小于期限因子的平均比例

工作流类型	规模	WS-PCPDC			MDP
		Optimized	DC	Fair	
CyberShake	小	0.51	0.36	0.67	0.93
	中	0.17	0.01	0.18	0.06
	大	0.34	0.39	0.21	0.32
Epigenomics	小	0.19	0.19	0.40	2.94
	中	1.17	1.36	1.72	3.07
	大	0.52	1.11	1.06	2.09
LIGO	小	0.20	0.39	0.21	0.47
	中	0.08	0.09	0.18	0.15
	大	0.05	0.30	0.25	0.49
Montage	小	0.49	0.41	0.61	0.46
	中	0.03	0.74	0.39	0.88
	大	0.21	0.87	0.35	1.17
SPIHT	小	2.24	3.53	3.10	5.99
	中	1.19	0.89	1.14	2.44
	大	0.04	0.16	0.05	0.40

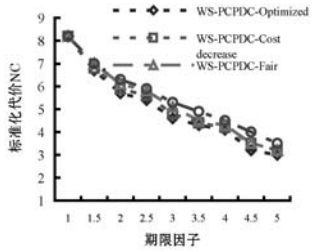
图 3 给出了调度算法得到的执行代价情况。大致上,中小规模工作流拥有类似结果,两类算法在较宽松期限下(期限因子为 5)拥有基本相同的标准化代价值(约为 2),这表明当将期限从 M_{HEFT} 增加到 5 倍时,对于中小规模工作流标准化代价降低幅度约小于两倍 C_{C},除了中规模 Montage 例外。对于大规模工作流则拥有完全不同的结果,仅有 SIPHT 工作流维持与中小规模工作流相同的结果,而 Montage 工作流拥有最差的性能表现。这表明拥有大数量任务的大规模工作流中,工作流任务间的结构特征比中小规模工作流更加影响任务调度过程。图 3 还表明,Optimized 在三种策略中及所有工作流类型中拥有最佳的性能表现,即最小的代价,也优于参考算法 MDP。



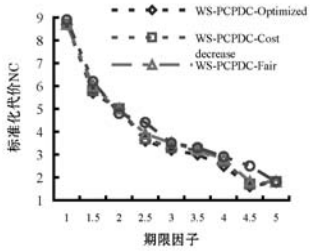
(a) 小规模 CyberShake



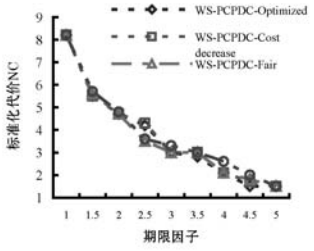
(b) 中规模 CyberShake



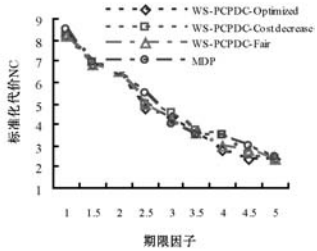
(c) 大规模 CyberShake



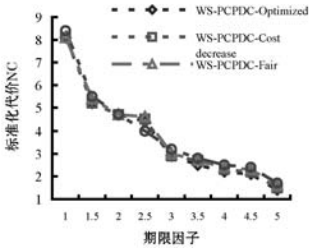
(d) 小规模 Epigenomics



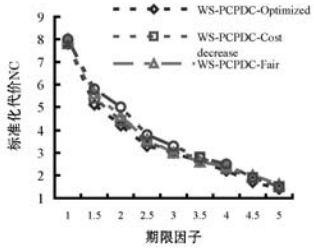
(e) 中规模 Epigenomics



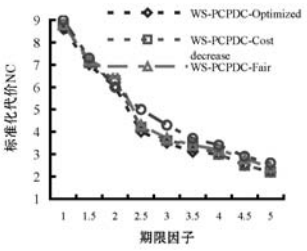
(f) 大规模 Epigenomics



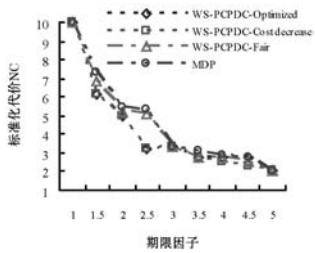
(g) 小规模 LIGO



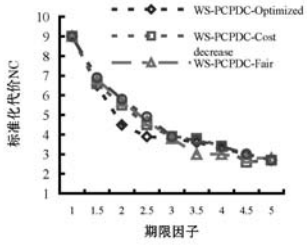
(h) 中规模 LIGO



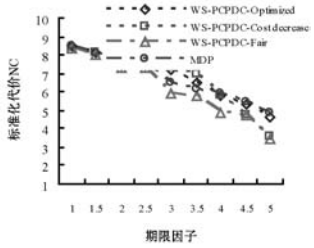
(i) 大规模 LIGO



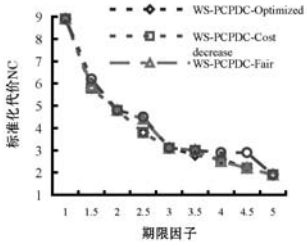
(j) 小规模 Montage



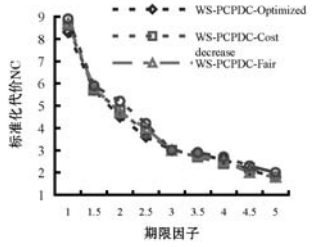
(k) 中规模 Montage



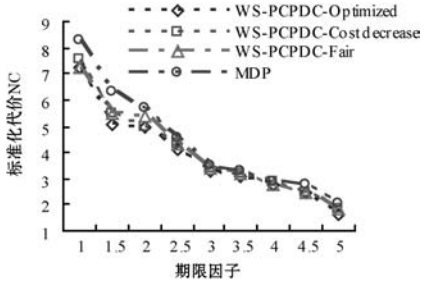
(l) 大规模 Montage



(m) 小规模 SPIHT



(n) 中规模 SPIHT



(o) 大规模 SPIHT

图 3 标准化代价情况

对于 CyberShake、LIGO 和 SIPHT 工作流,利用 Optimized策略的 WS-PCPDC 算法拥有最佳性能,而 Cost Decrease 策略则拥有近似表现。Fair 策略拥有最差的性能,但仍然比 MDP 算法表现更优。表 4 给出 WS-PCPDC 算法比较 MDP 算法的性能优势。

表 4 WS-PCPDC 算法比较 MDP 算法的性能优势

工作流类型	小规模	中规模	大规模
CyberShake	5.56	8.13	9.04
Epigenomics	6.46	3.75	2.82
LIGO	3.54	6.68	10.83
Montage	8.38	5.44	0.04
SPIHT	7.13	9.32	12.26

对于 Epigenomics 工作流,MDP 在某些情况下具有比 WS-PCPDC 更好的性能,在大中规模情况下可以得到更小的平均代价降低幅度,主要原因是 Epigenomics 工作流结构中拥有多个并行线性任务。初始状态下,WS-PCPDC 寻找工作流的关键路径时,工作流拥有多个入口任务,一个是并行管道任务,其他三个为工作流的终端任务。WS-PCPDC 试图为这条关键路径寻找最优调度时,未考虑第一个和第六个任务间的并行性。若考虑并行性,需分配最长的子期限到这四个任务上,由于此时可以留下更多的空闲时间,全局代价也可以得到降低。

大规模 Montage 在所有算法上均拥有最差性能,即截止期限的增加并未带来代价降低幅度的增加。在大规模 Montage 中,当增加期限至 5 倍时,代价降低约为初始值的一半。进一步,利用 Optimized 的 WS-PCPDC 算法从小规模至大规模工作流中有所降低,尤其在大规模工作流中,其性能差于 MDP。原因在于,对于 Montage 结构,其全局关键路径由 9 个任务组成,需优先为该路径分配子期限。对于小规模工作流,所分配的子期限在资源选择阶段得以保留。然而,对于大规模工作流,全局关键路径上第三个任务前的很多任务会在资源选择阶段被调度到更慢的资源上,这会导致关键路径上的第三个任务无法按时完成,并会将这推迟传导至其子节点,从而降低最终的调度性能。Fair 在大规模工作流中拥有较好性能,比较 MDP 的平均代价降低比例为 12.07,该策略在中规模工作流中也有较好性能。

6　结　语

为了满足期限约束,最小化执行代价,提出一种基于局部关键路径和截止期限分配的工作流调度算法。算法通过定义工作流的局部关键路径,以递归方式在局部关键路径上的任务间进行子期限分配,并在调度资源选择上满足任务子期限的同时,为每个任务选择执行代价最低的调度,实现调度代价优化。

参　考　文　献

[1] Foster I, Chard K, Tuecke S. The Discovery Cloud: Accelerating and Democratizing Research on a Global Scale[C]// IEEE International Conference on Cloud Engineering. IEEE, 2016:68 - 77.

[2] 王岩, 汪晋宽, 韩英华. 面向云工作流的两阶段资源调度方法[J]. 华南理工大学学报 (自然科学版), 2017, 45(1):80 - 87.

[3] 于新征, 蒋哲远. 面向服务的云工作流模型与调度研究[J]. 微电子学与计算机, 2016, 33(7):44 - 48.

[4] Genez T A L, Bittencourt L F, Madeira E R M. Workflow scheduling for SaaS/PaaS cloud providers considering two SLA levels[J]. Network Operations & Management Symposium IEEE, 2016, 14(5):906 - 912.

[5] Chard R, Chard K, Bubendorfer K, et al. Cost-Aware Cloud Provisioning[C]//IEEE, International Conference on E-Science. IEEE, 2015:136 - 144.

[6] Abrishami S, Naghibzadeh M, Epema D H J, et al. Deadline-constrained workflow scheduling algorithms for Infrastructure;as a Service Clouds[J]. Future Generation Computer Systems, 2013, 29(1):158 - 169.

[7] Tan W A, Guang-Zhen L U, Sun Y. Workflow scheduling optimization based on concurrent level[J]. Computer Integrated Manufacturing Systems, 2014, 20(5):1070 - 1077.

[8] Wu F, Wu Q, Tan Y, et al. Schedule Compaction and Deadline Constrained DAG Scheduling for IaaS Cloud[C]// International Conference on Cloud Computing and Big Data in Asia. Springer International Publishing, 2015:16 - 28.

[9] Arabnejad V, Bubendorfer K. Cost Effective and Deadline Constrained Scientific Workflow Scheduling for Commercial Clouds[C]//IEEE, International Symposium on Network Computing and Applications. IEEE, 2016:106 - 113.

[10] Sahni J, Vidyarthi D. A Cost-Effective Deadline-Constrained Dynamic Scheduling Algorithm for Scientific Workflows in a Cloud Environment[J]. IEEE Transactions on Cloud Computing, 2015, 34(2):1 - 10.

[11] Sun T, Xiao C, Xu X, et al. An Improved Budget-Deadline Constrained Workflow Scheduling Algorithm on Heterogeneous Resources[C]// IEEE International Conference on Cyber Security & Cloud Computing. 2017:23 - 29.

[12] Juve G, Chervenak A, Deelman E, et al. Characterizing and profiling scientific workflows[J], Future Generation Computer Systems, 2013,29(3):682 - 692.

[13] Chopra N, Singh S. HEFT based workflow scheduling algorithm for cost optimization within deadline in hybrid clouds [C]//Fourth International Conference on Computing, Communications and Networking. IEEE, 2013:1 - 6.

(上接第 213 页)

[11] Deng T, Deng Y, Shi Y, et al. Research on Improved Locally Linear Embedding Algorithm[J]. Communications in Computer & Information Science, 2014, 472:88 - 92.

[12] Huang R S. Information Technology in an Improved Supervised Locally Linear Embedding for Recognizing Speech Emotion[J]. Advanced Materials Research, 2014, 1014: 375 - 378.