

基于动态任务优先级的网格任务调度算法研究

孟宪福*, 闫玲玲, 刘伟伟

(大连理工大学 计算机科学与技术学院, 辽宁 大连 116024)

摘要: 网格环境下的任务调度是一个 NP 完全问题. 为了确保每一步都能优先调度影响调度长度最大的就绪任务, 提出一种采用动态任务优先级策略的任务调度算法. 在进行任务调度的过程中, 通过动态计算任务图 DAG 的关键路径并有效地利用处理器的空闲时间段来复制任务, 使任务节点之间的通信开销尽可能降低, 进而缩短整个任务图的完成时间. 大量的模拟实验结果表明, 所提的算法在任务完成时间上明显优于 HEFT 算法和 DDS 算法.

关键词: 网格环境; 任务调度; 动态任务优先级; 任务复制; 调度长度

中图分类号: TP393 **文献标志码:** A

0 引言

一般而言, 比较典型的任务调度策略都是基于任务图 DAG (directed acyclic graph, 有向无环图) 的, 这些任务调度策略可以分为静态任务调度策略和动态任务调度策略. 静态任务调度策略是指在任务执行之前依据任务图所提供的基本信息, 计算出任意一个任务在每个处理器上的运行时间和每个任务之间的数据传输时间, 并采用非抢占方式的调度; 与此相反的则被称之为动态任务调度策略. 依据调度策略的差异, 一般将静态任务调度算法分为 4 类, 即基于任务复制的调度算法^[1~3]、基于任务聚类的调度算法^[4~6]、基于随机搜索的调度算法以及表调度算法^[7~9]. 其中, 表调度算法是一种启发式算法, 由于其设计简单、复杂度较低且能够获得较好的次优解而被广泛地研究^[10].

本文采用表调度的思想, 提出基于动态任务优先级的网格任务调度算法, 通过动态计算所有就绪任务的优先级以及利用处理器的空闲时间段来复制父任务, 使当前任务尽可能早地开始执行, 从而缩短整个任务图的执行时间.

1 表调度算法简介

表调度算法的一般处理过程是将任务分配优

优先级后来生成任务调度列表, 在此基础上按顺序从调度列表中取出任务, 并将其调度到最适合该任务的处理器上, 直到列表为空为止. 下面介绍作为本文算法比较对象的两个典型的表调度算法.

1.1 HEFT 算法

HEFT (heterogeneous-earliest-finish-time)^[11] 算法主要分为两个处理步骤, 首先按照节点的静态级 SL ^[12] 的大小逆序形成一个任务调度列表, 出口任务节点的静态级为出口节点任务的计算量, 然后从任务图的出口任务出发, 自下而上递归计算每个任务节点的静态级, 其计算公式为 $SL(n_i) = \overline{w}_i + \max_{n_j \in succ(n_i)} (c_{ij} + SL(n_j))$, 其中 $\overline{w}_i$ 是任务 n_i 在所有处理器上的平均执行时间, c_{ij} 是任务 n_i 和 n_j 之间的通信时间, $SL(n_j)$ 为任务 n_j 的静态级, $succ(n_i)$ 为任务 n_i 所有子任务节点的集合. 在完成对任务调度列表的构造以后, 将调度列表中的各个任务调度到使其完成时间最早的处理器上. 并且, 在不违反任务之间的优先级约束关系的前提下, 该算法还通过将任务调度到两个已经调度结束的任务之间的空闲时间槽上执行来提高调度性能. HEFT 调度算法的时间复杂度为 $O(pn^2)$, 其中 n 是任务节点的个数, p 是处理器节点的个数.

收稿日期: 2010-03-17; 修回日期: 2011-11-10.

基金项目: 国家自然科学基金资助项目(60973014).

作者简介: 孟宪福* (1960-), 男, 副教授, E-mail: xfmeng@dlut.edu.cn.

1.2 采用动态决策路径的任务调度算法

采用动态决策路径的网格任务调度算法(即DSS算法)^[13],利用“先调度、后优化”的调度策略:首先基本的调度方案是采用传统的启发式调度算法来获得,同时任务图DAG也是依据所求得的甘特图来重新生成的,然后计算任务图的决策任务以及决策路径.最后,在能够保证任务之间约束条件的前提下,尽可能地将决策任务分配到处理器的空闲时间槽上提前运行,以缩短整个任务图的执行时间.

此算法的调度性能与初始所选择的启发式调度算法相关是采用动态决策路径的网格任务调度算法的主要问题,除此之外,此算法的时间复杂度也比较高,为所采用的启发式算法的复杂度与 $O(pn^3)$ 之和.

2 相关定义和建模

在具体讨论调度算法之前,首先假设:(1)如图1所示,每个任务之间的数据依赖关系已经确定,并以有向无环图 $G(N_t, e_t)$ 来表示,其中任务节点集合由 N_t 表示,任务节点之间的有向边集合由 e_t 表示,有向边的权值,即任务 n_i 和 n_j 之间的通信时间由 c_{ij} 表示.若 n_i 和 n_j 被调度到同一台处理器上,则 c_{ij} 的值为0.为了不失一般性,如果一个任务图存在多个入口(或出口)节点,则可以将它们用权值为0的边连接到一个伪入口(或出口)的节点上.(2)任何一个处理器都可以执行任何一个任务且每个任务在每台处理器节点上的执行时间是已知的.同时,任何两个处理器之间也都是可以通信的.(3)当一个任务执行完成之后,才能并行地向其所有的子任务发送消息,而只有当任务收到其所有父任务发送来的消息之后它才能开始执行.(4)采用先来先服务的策略来调度在同一个

节点上的任务,并且任务是采用非抢占方式执行的.

为了更好地描述算法,先给出下文中将要使用的几个基本概念.

任务节点的最早可能启动时间定义为 $aest(n_i)$.任务节点的最早可能启动时间实际上就是该任务节点与其所有的父任务节点的通信量都存在的情况下,任务节点的最早可能启动时间.换句话说,在该时间点之前,任务节点的所有父任务都已经执行完毕并且将相应的数据传送到该节点上.计算公式为

$$aest(n_i) = \begin{cases} 0; & n_i \text{ 为入口节点} \\ \max_{n_j \in pred(n_i)} (c_{ji} + aest(n_j) + w_j) \end{cases}$$

其中 w_j 为其父任务 n_j 的执行时间, $pred(n_i)$ 为任务 n_i 所有父任务节点的集合.

任务 n_i 的最晚可能启动时间定义为 $alst(n_i)$.任务节点的最晚可能启动时间是指在不推迟出口节点的最早可能启动时间的前提下,任务节点的最迟启动时间,可通过自下而上遍历整个任务图递归计算得到:

$$alst(n_i) = \begin{cases} east(n_i); & n_i \text{ 为出口节点} \\ \min_{n_j \in succ(n_i)} (alst(n_j) - c_{ij}) - \overline{w}_i \end{cases}$$

任务节点的最早可能启动时间与最晚可能启动时间之差称为时间余量.显然,时间余量越小,相关任务的紧急程度越高.若最早可能启动时间等于最晚可能启动时间,也就是任务节点的时间余量为零,则任务节点为关键任务.关键路径则是由一组关键任务节点组成的,并且路径的入口为整个任务图的入口节点,而路径的出口为整个任务图的出口节点.

任务 n_i 在处理器 p_q 上的启动时间为 $tst(n_i, p_q)$.由前文的假设(3)与(4)可知, $tst(n_i, p_q)$ 主要是受到以下2个因素制约的.

(1)当前处理器的起始可用时间,即最近一次被分配到该处理器上的任务的完成时间,用 $ava(p_q)$ 来表示, $ava(p_q) = ft(n_k)$, $ft(n_k)$ 为任务 n_k 的完成时间, n_k 表示最近一次被调度到处理器 p_q 上的任务节点. $ava(p_q)$ 初始值为0.

(2)任务 n_i 的父任务将信息发送到处理器 p_q 的时间.处理器 p_q 必须接收到任务 n_i 的全部父任务信息且该处理器可用时才能开始执行 n_i ,因此 $tst(n_i, p_q)$ 可表示为

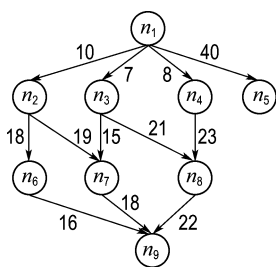


图1 任务图

Fig. 1 Task graph

$$tst(n_i, p_q) = \max(ava(p_q), \max_{n_j \in pred(n_i)} (ft(n_j) + c_{ji})) \quad (1)$$

任务 n_i 在处理器 p_q 上的完成时间为该任务在处理器 p_q 上的启动时间与执行时间之和, 表示为 $tft(n_i, p_q) = tst(n_i, p_q) + \omega_{i,q}$.

3 任务优先级分配与任务复制策略

本文算法主要分为两个步骤: (1) 计算任务优先级; (2) 为任务节点选择相应的处理器. 以上两个步骤穿插进行. 在为就绪任务节点选择处理器时, 通过复制决策父任务来减少通信开销, 以缩短整个任务图的完成时间.

3.1 动态任务优先级分配策略

本文采用动态优先级进行任务调度, 在每一个调度步中, 都重新计算当前所有就绪任务(即还未被调度的任务)的优先级, 并选择当前优先级最高的任务节点进行调度. 动态任务优先级分配算法如下:

(1) 计算所有任务的 $aest$ 和 $alst$.

(2) 判断就绪任务中是否存在关键任务, 若不存在, 计算所有就绪任务的时间余量, 选择时间余量最小的就绪任务作为当前任务. 若不唯一, 则选择其中子任务数目最多的, 若仍不唯一, 则随机指定, 转(4).

(3) 若关键任务不唯一, 则采用静态级 SL 作为次权值, 选择静态级最大的关键任务作为当前任务.

(4) 判断其父任务节点是否都已调度完毕, 若是, 转(5); 否则, 对其父任务节点按时间余量大小排序并选择时间余量最小的父任务作为当前任务, 若不唯一, 处理方式同上, 重复(4).

(5) 为当前任务指定最高优先级.

在完成一次任务优先级指定之后, 将根据第4章中叙述的过程, 为当前优先级最高的任务节点分配相应的处理器. 对于被分配了最高优先级的任一任务节点, 由于其所有父任务都已经被分配到固定的处理器上, 这样也就完全可以准确确定其执行时间和启动时间, 也就是能够准确获得该任务的 $aest$. 在后续的计算过程中, 也就不必继续使用平均执行时间来进行计算了, 使得此算法能够比较好地适应异构的网格环境.

3.2 任务的冗余复制策略

首先给出以下两个定义.

定义 1 决策父任务. 对于任意 $n_j \in pred(n_i)$, 若 $ft(n_j) + c_{ji} = \max_{n_k \in pred(n_i)} (ft(n_k) + c_{ki})$, 则称 n_j 为 n_i 的决策父任务, 记为 $cp(n_i)$. 若当前任务的父任务数多于一个, 则将 $ft(n_j) + c_{ji}$ 次大的父任务节点记为 $scp(n_i)$.

定义 2 空闲时间槽. 将任务 n_i 在处理器 p_q 上等待其父任务信息传递的这段时间称为空闲时间槽, 记为 $slot(n_i, p_q)$. 显然, $slot(n_i, p_q) = tst(n_i, p_q) - ava(p_q)$.

为了有效地缩短整个任务图的完成时间, 可以尽量将当前任务的父任务节点分配到空闲时间槽上执行, 这样能够减少通信开销, 使当前任务的执行及早启动, 从而能够及时调度后续任务节点. 需要注意的是, 将父任务节点分配到空闲时间槽上执行, 需要满足以下的规则.

规则 1 如果到现在为止任务 n_i 的父任务节点 n_j 从未被调度到处理器节点 p_q 上且 n_j 在处理器 p_q 上满足式(2), 则可以将任务节点 n_j 插入到 $slot(n_i, p_q)$ 时间槽上执行:

$$slot(n_i, p_q) \geq \omega_{j,q} \text{ 且 } tft(n_j, p_q) < tst(n_i, p_q) \quad (2)$$

需要注意的是, 并不是将任何一个满足规则1的父任务分配到空闲时间槽上执行都能使当前任务的开始执行时间得到提前, 由于显著影响任务启动时间的是决策父任务, 应首先考虑复制当前任务的决策父任务节点.

性质 1 若任务 n_i 被调度到处理器 p_q 上, 同时其决策父任务满足规则1, 则利用 $slot(n_i, p_q)$ 时间槽来复制任务 n_i 的决策父任务, 将提前任务 n_i 在处理器 p_q 上的开始执行时间 $tst(n_i, p_q)$.

实际上, 由于 $tst(n_i, p_q) = \max_{n_j \in pred(n_i)} (ft(n_j) + c_{ji})$, 根据空闲时间槽的定义, 在将决策父任务 n_j 插入到 $slot(n_i, p_q)$ 之后, 假设 n_i 的父任务数多于一个, 则会出现如下两种情况: (1) $tft(cp(n_i), p_q)$ 早于或等于 $scp(n_i)$ 信息到达处理器 p_q 的时间点, 如图 2(a) 所示, 此时 $tst(n_i, p_q)$ 提前到 $ft(scop(n_i)) + c_{i,scop(n_i)}$. (2) $tft(cp(n_i), p_q)$ 晚于 $scp(n_i)$ 信息到达处理器 p_q 的时间点, 如图 2(b) 所示, 由规则1可知, $tft(cp(n_i), p_q)$ 肯定小于

$\max_{n_j \in \text{pred}(n_i)} (ft(n_j) + c_{ji})$, 此时 $tst(n_i, p_q)$ 提前到 $tft(cp(n_i), p_q)$. 若 n_i 的父任务只有一个, 则将其决策父任务安排到空闲时间槽上执行, 这样, 决策父任务在处理器上的完成时间将是该任务的开始执行时间. 如果该任务的父任务不存在, 则不需要进行任何操作. 因此, 无论出现哪种情况, 都将提前 n_i 任务在 p_q 处理器上的执行时间 $tst(n_i, p_q)$.

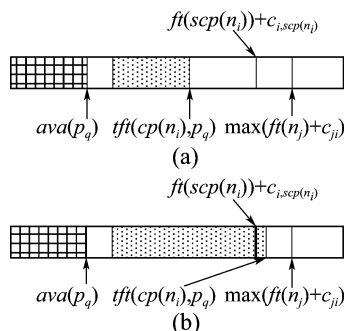


图2 性质1分析示意图

Fig. 2 The analysis sketch map for Property 1

性质2 如果父任务队列中的第 $i (i \geq 2)$ 个任务在被确定为决策父任务之后不再满足规则1, 则无需复制队列中的其他满足规则1的父任务, 因为即使复制这些父任务也不可能提前任务 n_i 在处理器 p_q 上的开始执行时间 $tst(n_i, p_q)$.

实际上, 若任务 n_i 的决策父任务为第 i 个父任务, 则前 $i-1$ 个父任务都已复制完毕. 此时 n_i 的启动时间取决于两种情况: (1) 第 $i-1$ 个父任务在处理器 p_q 上的完成时间. 此时 $slot(n_i, p_q)$ 为零, 无法复制父任务. (2) 第 i 个父任务信息到达的时间. 此时即使复制后续的父任务, 也无法提前第 i 个父任务的信息到达时间, 因此无法提前 $tst(n_i, p_q)$.

4 任务调度算法

4.1 算法描述

首先随机选择一个处理器 p_q 来分配任务节点 n_i , 然后计算当前节点的空闲时间槽和 n_i 节点的 $tst(n_i, p_q)$, 并将 n_i 节点的所有父任务以其数据到达时间的逆序进行排序. 采用3.2节所描述的策略来完成复制过程, 最后计算在该处理器节点上此任务的完成时间. 对所有处理器节点循环执行上述过程, 从中选择使当前任务 n_i 的完成时间最小的处理器节点, 并将任务分配到该处理器上, 而后取消为提前该任务的开始时间而在

其他处理器上冗余映射的父任务, 以防止因冗余复制而推迟其他任务节点的执行. 若存在多个能使该任务最早完成的处理器, 则任意选择一个处理器进行分配. 对所有就绪任务节点的 $aest$ 和 $alst$ 进行更新. 算法描述如下:

- (1) while(未被调度的任务个数 > 0)
- (2) 选择优先级最高的就绪任务节点 n_i .
- (3) foreach ($p_q \in M$)
- (4) 对于任意 $n_j \in \text{pred}(n_i)$, 计算 $ft(n_j) + c_{ji}$, 对父任务根据 $ft(n_j) + c_{ji}$ 进行逆序排序以便构成父任务队列. 显然, 队列中的第一个任务一定是决策父任务.
- (5) 计算任务 n_i 在处理器 p_q 上的 $tst(n_i, p_q)$ 和 $tft(n_i, p_q)$.
- (6) 如果父任务队列中的第一个任务节点 (即决策父任务节点) 满足规则1, 则将其分配到当前空闲时间槽内, 当前决策父任务出队列, 更新 $tst(n_i, p_q)$ 和 $slot(n_i, p_q)$. 重复(6); 若不满足, 转(7).
- (7) 计算 $tft(n_i, p_q)$.
- (8) end foreach
- (9) 将任务 n_i 调度到使 $tft(n_i, p_q)$ 最小的处理器 p_q 上, 更新此处理器的 ava 值.
- (10) 更新所有就绪任务的 $aest$ 和 $alst$.
- (11) end while

由上述算法可知, 选择优先级最高的就绪任务时间复杂度为 $O(n^2)$, 任务复制与调度的时间复杂度为 $O(pn^2)$. 这样, 本文算法的时间复杂度为 $O(pn^2)$, 其中, n 为任务节点的个数, p 为处理器的个数. 在复杂度上本文算法与 HEFT 算法相同, 但低于 DDS 算法.

4.2 算法举例

下面考虑在3个处理器上调度10个任务的过程. 图3给出了 DAG 表示的任务图, 其中任务节点间的通信量由有向边上的数值表示, 每个任务在各处理器上的执行时间由表1给出. 各任务节点的静态级 SL 如表2所示.

下面给出 HEFT 算法、本文算法以及 DDS 算法产生的任务调度结果图, 其中 DDS 算法所采用的初始启发式算法为文献[13]中所使用的 ETF 算法. 图4(a)为 HEFT 算法的调度结果图, 结果为80; 图4(b)为 DDS 算法的调度结果图, 结

果为 97;图 4(c)为本文算法的调度结果图,其中利用斜线标示的任务节点是为了提早后续任务的完成时间而被复制的父任务节点,调度结果为 73,有效缩短了整个任务的调度长度。

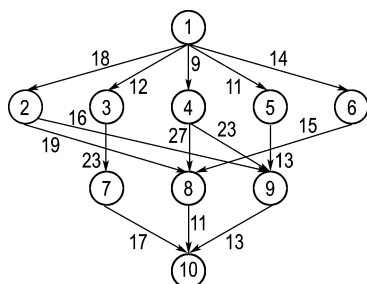


图3 算法任务图

Fig. 3 Task graph in algorithm

表1 执行时间表

Tab. 1 List of execution time

任务节点	p_1	p_2	p_3	任务节点	p_1	p_2	p_3
n_1	14	16	9	n_6	13	16	9
n_2	13	19	18	n_7	7	15	11
n_3	11	13	19	n_8	5	11	14
n_4	13	8	17	n_9	18	12	20
n_5	12	13	10	n_{10}	21	7	16

表2 任务图中各节点的 SL

Tab. 2 SL value of each node in the task graph

任务节点	SL	任务节点	SL
n_1	108	n_6	63
n_2	77	n_7	42
n_3	80	n_8	35
n_4	80	n_9	44
n_5	69	n_{10}	14

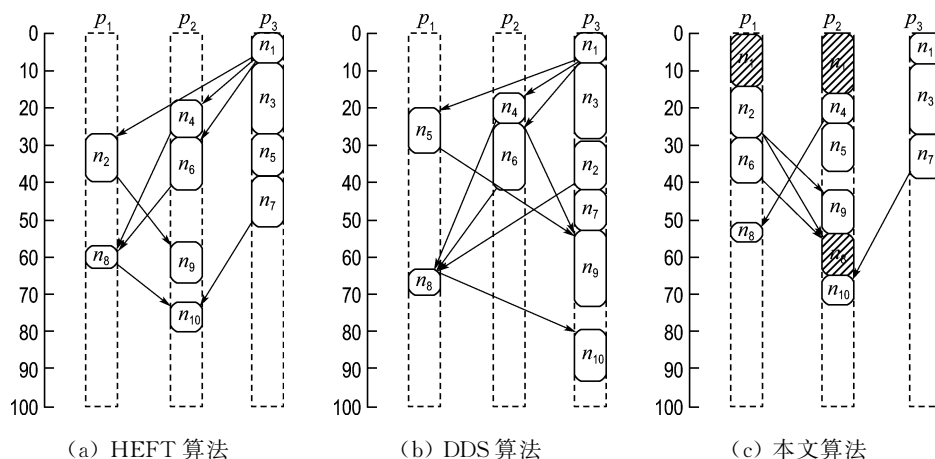


图4 调度结果

Fig. 4 Scheduling results

5 实验结果与性能分析

考虑到 HEFT 算法及 DDS 算法与本文的问题假设基本相同,都充分利用了因任务之间通信量所产生的处理器的空闲时间段,以 HEFT 算法和 DDS 算法作为本文算法的比较对象,以验证本文算法在网格计算环境下进行任务调度的有效性。

5.1 实验环境

首先利用任务图 DAG 生成器来随机生成相应的任务图,并以同样的任务图来比较和分析 3 个算法的性能.为了生成任务图,需要输入的参数如下。

(1) 每个任务图所包含的任务节点的个数 N_t .

(2) 每个任务图中节点出度的最大值 $\max_outdegree$. 0 与 $\max_outdegree$ 之间的一个随机数是每个节点的出度。

(3) 每个任务图中节点权重的最小值 $\min_Node_weight$ 和最大值 $\max_Node_weight$. $\min_Node_weight$ 和 $\max_Node_weight$ 之间的一个随机数是任务图中每个节点的计算量 $Node_weight$.

(4) 通信量与计算量的比值 CCR. 若一个任务图的 CCR 比较低,则可认为是计算密集型的应用;否则,可认为是通信密集型的应用。

(5) 处理器的利用率 m . $m = P/N_t$, P 为当前可以利用的处理器个数, $m \leq 1$.

(6) 每个任务图中边的权重的最小值

$\min Edge_weight$ 和最大值 $\max Edge_weight$. $\min Edge_weight$ 和 $\max Edge_weight$ 之间的一个随机数是任务图中的节点与节点之间的通信量 $Edge_weight$. 总的计算量 $Total_comp$ 和 CCR 之乘积是任务图中的总的通信量 $Total_comm$.

在本实验中所生成的任务图的参数如下:

$N_t = \{10, 20, 40, 80, 120, 200, 250, 300, 400, 500\}$

$CCR = \{0.1, 0.5, 1.0, 2.0, 5.0, 10.0, 12.0, 15.0\}$

$Node_weight = [1, 30]$

$\max_outdegree = \{3, 4, 5, 6, 7, 8\}$

$m = \{0.05, 0.10, 0.15, 0.20, 0.25, 0.30, 0.35, 0.40, 0.45, 0.50\}$

$Edge_weight = [1, 300]$

5.2 性能分析指标

一个任务调度算法性能好坏的重要判断标准就是整个任务图的完成时间. 由于每个任务图的构成是不同的, 应该对调度长度进行标准化处理, 以便能够客观和准确地对调度算法的性能进行比较.

定义 3 调度长度比率 $SLR^{[14]}$. 其计算公式为 $SLR = \frac{Makespans}{CPEC}$, 其中 $Makespans$ 是整个任务图的完成时间, 也就是任务图的出口节点任务的完成时间. $CPEC$ 是位于关键路径上的所有任务的计算量之和, 而每个任务的计算量是以其在每个处理器上执行时间的最小值来确定的. 显然 $SLR \geq 1$, 且 SLR 越小说明此调度算法的所有任务的完成时间越短, 其性能也就越好.

定义 4 性能提高百分比^[14]. 其计算公式如下:

$$Improvement_Percentage = \frac{(Comparison_SLR - \text{本文算法_}SLR)}{Comparison_SLR} * 100\%$$

其中 $Comparison_SLR$ 为比较对象的 SLR 值, 本文算法 $_SLR$ 为本文算法的 SLR 值. 显然, 如果 $Improvement_Percentage$ 为负, 则说明本文算法的性能低于比较对象算法, 如果为正, 则说明本文算法性能高于比较对象算法, 正值越大, 说明本文算法的优越性越大.

5.3 性能分析

在网络任务调度过程中, 任务之间的通信量以及计算量对调度性能都产生直接影响. 为此, 分

析调度算法性能需要考虑的一个重要因素就是作为总的通信量和总的计算量之比的 CCR. 与此同时, 一个好的调度算法应该具有良好的可扩展性, 也就是问题规模的变化不应该明显影响调度算法的性能. 基于以上的考虑, 本文将实验分为两部分: 第一部分实验主要测试调度算法的性能受 CCR 变化的影响; 第二部分实验主要测试任务数量的增长对调度算法性能所产生的影响.

在第一部分实验中, 考虑在处理器数分别等于 20、30、40、50 时调度 200 个任务的情况. 利用任务图 DAG 生成器来生成 3 000 个不同的 DAG, 这些 DAG 具有不同的任务节点的出度、权重和边的权重等. 在实验中, 对所有这些任务图 DAG 进行调度以得到平均实验结果, 如图 5 所示.

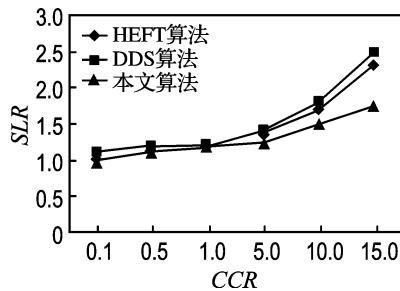
由实验结果可以看出, 由于 HEFT 算法没有采用任务复制策略来降低任务间的通信延迟, 很多能够产生更短调度的机会被错过了, 这种情况在 CCR 比较大时尤为明显. 而 DDS 算法在重新调度决策路径上的任务节点时, 由于可以在处理器节点之间迁移决策父任务, 这就使得原来与其位于同一个处理器上的子任务之间的通信量不再为零, 从而导致延缓任务的开始执行时间, 并使后续调度到此处理器上的任务的启动时间也被延迟了, 使得无法有效缩短整个任务图的调度长度. 而且, 随着 CCR 的增大, 该问题尤为明显. 在这种情况下, 初始所选的启发式调度算法在很大程度上决定了算法的调度性能.

由图 6(a)可知, 如果处理器的可用率 m 比较小同时 CCR 也比较小时, 则本文算法与 HEFT 算法在性能上差别不大. 其原因是如果 CCR 比较小, 则表示任务之间的通信开销很小, 也就是处理器节点的空闲时间槽很少, 这样, 基本上不存在满足复制条件的父任务. 然而, 随着 CCR 的增大, 任务之间的通信开销也开始增加, 处理器节点的空闲时间槽也大幅增加, 这样本文策略的优越性也就显现出来了. 由图 6(b)可知, 随着处理器数目的增多以及 m 的增大, 任务复制策略开始发挥作用, 从而降低了任务之间的通信开销.

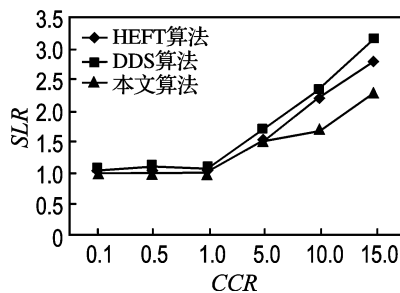
第二部分实验主要验证任务个数的增加对调度算法性能的影响. 设 $CCR = 12$, 利用任务图 DAG 生成器共生成 3 000 个不同的 DAG, 这些 DAG 具有不同的任务节点出度、权重和边的权重

等. 图7给出了平均实验结果, 其中处理器的个数 $P = mN_i$, 而处理器的可用率 m 为 $0.15 \sim 0.40$ 的随机值.

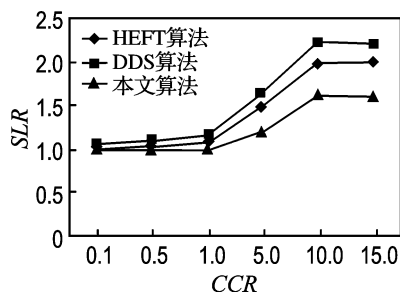
由图7不难看出, 3种算法的性能都随着任务个数的增加而呈现出不同程度的下降, 但无论针对何种任务图 DAG, 本文算法的性能始终优于其他两种算法, 说明本文算法具有很好的可扩展性.



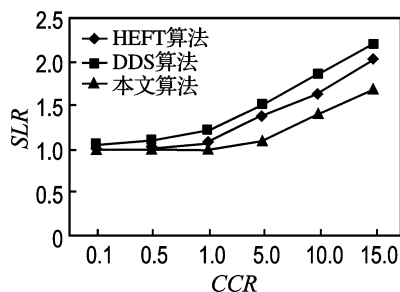
(a) $m=0.10, N_t=200$



(b) $m=0.15, N_t=200$



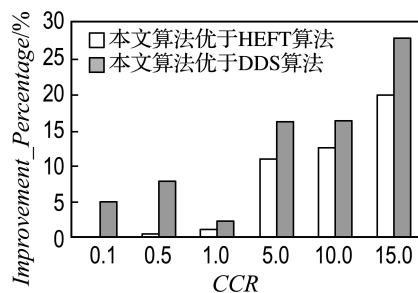
(c) $m=0.20, N_t=200$



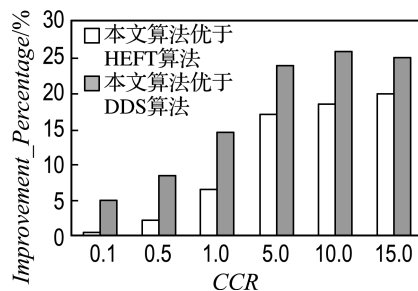
(d) $m=0.25, N_t=200$

图5 CCR对算法性能的影响

Fig. 5 Impacts of CCR on the performance of algorithms



(a) $m=0.10, N_t=200$



(b) $m=0.25, N_t=200$

图6 性能提高比随CCR的变化

Fig. 6 Performance improvement with the changes of CCR

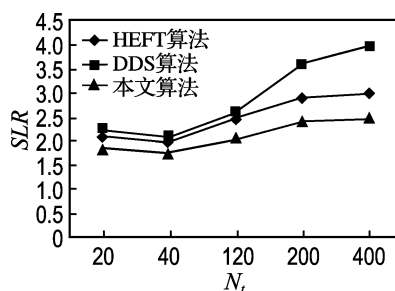


图7 任务节点个数变化对算法性能的影响

Fig. 7 Impacts of the task number on the performance of the algorithms

6 结 论

本文提出了基于动态任务优先级的任务调度算法. 此算法对关键任务节点与非关键任务节点进行了综合考虑, 对任务图的完成时间影响最大的就绪任务进行优先调度. 在此基础上, 通过冗余地将决策父任务调度到子任务所在处理器节点的空闲时间槽上执行以减少任务之间的通信开销, 从而能够有效地缩短整个任务图的完成时间.

参考文献:

- [1] KRUATRACHUE B, LEWIS T G. Duplication scheduling heuristics (DSH): A new precedence task

- scheduler for parallel processor systems [R]. Corvallis: Oregon State University, 1987
- [2] CHUNG Y C, RANKA S. **Application and Performance Analysis of a Compile-time Optimization Approach for List Scheduling Algorithms on Distributed Memory Multiprocessors** [C]. Minneapolis: IEEE Computer Society Press, 1992:512-521
- [3] AHMAD I, KWOK Y K. On exploiting task duplication in parallel programs scheduling [J]. **IEEE Transactions on Parallel and Distributed Systems**, 1998, **9**(9):872-892
- [4] 杜晓丽, 蒋昌俊, 徐国荣, 等. 一种基于模糊聚类的网格 DAG 任务图调度算法 [J]. **软件学报**, 2006, **17**(11):2277-2288
- [5] YANG T, GERASOULIS A. DSC: Scheduling parallel tasks on an unbounded number of processors [J]. **IEEE Transactions on Parallel and Distributed Systems**, 1994, **5**(9):951-967
- [6] KWOK Y K, AHMAD I. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors [J]. **IEEE Transactions on Parallel and Distributed Systems**, 1996, **7**(5):506-521
- [7] SIH G C, LEE E A, INC Q, *et al.* A compile-time scheduling heuristic for interconnection constrained heterogeneous processor architectures [J]. **IEEE Transactions on Parallel and Distributed Systems**, 1993, **4**(2):75-87
- [8] ADMA T L, CHANDY K M, DICKSON J. A comparison of list scheduling for parallel processing systems [J]. **Communications of the ACM**, 1974, **17**(12):685-690
- [9] WU M Y, GAJSKI D D. Hypertool: A programming aid for message passing systems [J]. **IEEE Transactions on Parallel and Distributed Systems**, 1990, **1**(3):330-343
- [10] 石威, 郑伟民. 相关任务图的均衡动态关键路径调度算法 [J]. **计算机学报**, 2001, **24**(9):991-997
- [11] TOPCUOGLU H, HARIRI S, WU M. Performance-effective and low complexity task scheduling for heterogeneous computing [J]. **IEEE Transactions on Parallel and Distributed Systems**, 2002, **13**(3):260-274
- [12] GRAHAM R, LAWLER E, LENSTRAJ K, *et al.* Optimization and approximation in deterministic sequencing and scheduling: A survey [J]. **Annals of Discrete Mathematics**, 1979, **5**(2):287-326
- [13] 林剑柠, 吴慧中. 一种基于动态决策路径的网格任务调度算法 [J]. **计算机研究与发展**, 2008, **45**(5):841-847
- [14] BASKIYAR S, DICKINSON C. Scheduling directed a-cyclic task graphs on a bounded set of heterogeneous processors using task duplication [J]. **Journal of Parallel and Distributed Computing**, 2005, **65**(5):911-921

Research on task scheduling algorithm in grid computing systems based on dynamic task priority

MENG Xian-fu*, YAN Ling-ling, LIU Wei-wei

(School of Computer Science and Technology, Dalian University of Technology, Dalian 116024, China)

Abstract: Task scheduling is a NP-complete problem in the grid environment. To ensure that the task most significantly affects the makespan can be scheduled in each scheduling step, the task scheduling algorithm by using dynamic task priority is proposed. The critical path of directed acyclic graph (DAG) is dynamically determined and the idle time slots of nodes are effectively utilized to replicate task for reducing the communication overhead and shortening the overall execution time. Extensive experiments are carried out and the research results show that the proposed algorithm outperforms the HEFT algorithm and the DDS algorithm in execution time.

Key words: grid environment; task scheduling; dynamic task priority; task duplication; scheduling length