一种基于多处理器任务复制的分簇调度算法*

徐　成，赵林祥，杨志邦
（湖南大学 信息科学与工程学院，长沙 410082）

摘　要：任务调度的优劣是决定并行分布式计算机系统性能好坏的重要因素之一。为优化任务调度，基于一些典型算法（如 LG、PPA 算法等），提出了一种新的任务调度算法。该算法一方面复制满足条件的前驱任务来缩短调度长度；另一方面合理地复制其他前驱任务和合并冗余簇来减少所需处理器的数目。实验表明，该算法在调度长度和所需处理器的数目上优于以上典型算法，并具有更小的时间复杂度，对并行计算机系统性能的提升具有一定的意义。

关键词：任务复制；任务调度；多处理器；分簇复制
中图分类号：TP316　　**文献标志码**：A　　**文章编号**：1001-3695(2012)08-2931-04
doi:10.3969/j.issn.1001-3695.2012.08.034

Scheduling algorithm of multi-processor based on task clustering and duplication

XU Cheng, ZHAO Lin-xiang, YANG Zhi-bang
(*College of Information Science & Engineering*, *Hunan University*, *Changsha* 410082, *China*)

Abstract: The effect of task scheduling is one of the important factors which decide the performance of the parallel distributed computer system. In order to optimize the task scheduling, this paper proposed a novel algorithm based on some typical algorithms (e.g. LG, PPA algorithm etc.). On the one hand, the algorithm replicated precursor tasks which meet the duplication conditions in order to shorten the scheduling length. On the other hand, the algorithm replicated other precursor tasks and merged redundant clusters to reduce the number of the required processors. Simulation results show that the proposed algorithm's performance on scheduling length and the number of the required processors is better than the above typical algorithms'. Furthermore, the proposed algorithm has a smaller time complexity and benefits the performance of the parallel distributed computer system.

Key words: task duplication; task scheduling; multi-processor; clustering and duplication

在分布式并行计算中，任务调度就是根据一定的调度规则和调度策略，把并行程序的任务或构成工作负载的作业，按照一定的执行时序分配到并行分布系统的多个节点上，其目的是减少并行任务间的通信开销和等待时间，使任务的执行时间最短，提高实时任务的响应速度，改善系统负载的均衡性，并且优化系统运行效率等[1]。该问题是一个 NP 完成问题，难以在多项式时间内寻求最优解。任务调度中最为常见的是启发式调度算法。启发式算法通常分为三类[2]：a）基于优先级的调度；b）基于分簇的调度；c）基于复制的调度。

在所有的调度方法中，一个最难处理的问题是在不同处理器之间存在有依赖关系的任务之间的通信开销，这种通信开销在并行程序的执行中不可避免，这种通信开销在不基于任务复制的调度方法中成为了一个巨大的约束。由于在同一个处理器上的通信可以被忽略，所以在实现减少这种开销上，基于任务复制的调度成为一个有效的策略。事实上，已证明基于任务复制的调度在减少通信时间、最小化任务集总体完成时间上优于不基于任务复制的调度方法。现有的基于任务复制的调度算法在任务满足某些特定条件下才能产生最优调度。典型的算法如 Darbha 等人[3]提出的 TDS 算法，该算法需要任务颗粒满足一定条件才能产生最佳调度，并且只考虑与最佳前驱任务合并，而不考虑其他的前驱任务，从而失去产生更短调度的机会，其最优条件形式相对比较复杂，而且是在假设处理器数目不限的条件下的，而现实中处理器的数目往往是有限制要求的。PPA[1]算法是在处理器数目有限的条件下，允许多个前驱任务复制到同一个簇上，去掉了 TDS 算法约束条件，有更广的优化条件，但是与文献[4～7]中算法一样并没考虑专门再通过复制部分父类任务来减少簇数目的情况，在任务间通信频繁的情况下调度结果并不理想。文献[8]中算法对处理器数目有约束和无约束都适合，同时能有效地减少冗余复制，但是其在减少处理器的数目上还不够理想。LG[2]算法同样也去除了 TDS 的算法约束条件，有更广的优化条件，但是仍然没有考虑到将多个前驱放置到同一个簇中，所以在调度长度性能上还不尽理想。针对上述算法存在的不足，本文提出了一种新的算法。

1　调度算法模型和基本概念

通常一组并行有序任务可以用一个有向无环图（DAG）表示，从而将任务的调度问题转换为调度 DAG 图的问题。DAG 图可以用数学描述为 $G=(V,E,w,C)$。其中：

$V=\{n_i \mid n_i$ 是有序任务，$i=1,2,3,\cdots,v\}$ 是有序任务集，$v=|V|$ 表示任务集的数目；

收稿日期：2012-01-14；**修回日期**：2012-03-02　　**基金项目**：国家自然科学基金资助项目(60973030)

作者简介：徐成(1962-)，男，湖北蕲春人，教授，主要研究方向为嵌入式系统；赵林祥(1984-)，男，湖南邵阳人，硕士，主要研究方向为嵌入式系统(zlxgongzuo@126.com)；杨志邦(1984-)，湖南邵阳人，博士，主要研究方向为嵌入式系统.

$E=\{e_{ij}|e_{ij}$表示任务 n_i 到任务 n_j 的边$\}$是任务集 V 边的集合,$e=|E|$表示边的个数;

$w=\{w_i|w_i$ 表示任务 n_i 的计算时间开销或权值,$i=1,2,\cdots,v\}$是计算时间开销集合;

$C=\{c_{ij}|c_{ij}$表示任务 n_i 到 n_j 的通信时间开销$\}$,在本文中称任务 n_i 为任务 n_j 的父任务或者直接前驱任务;

$\text{pred}(n_i)=\{n_j|e_{ji}\in E\}$ 表示任务 n_i 的前驱任务集,$|\text{perd}(n_i)|$表示n_i 的前驱任务的数目;

$\text{succ}(n_i)=\{n_j|e_{ij}\in E\}$ 表示任务 n_i 的后驱任务集,$|\text{succ}(n_i)|$表示任务 n_i 的后驱任务的数目。

如果$|\text{pred}(n_i)|=0$,那么 n_i 为开始任务 n_{star},如果$|\text{pred}(n_i)|\geq 2$,那么 n_i 为 join 任务;如果$|\text{succ}(n_i)|=0$,那么 n_i 为结束任务 n_{exit},如果$|\text{succ}(n_i)|\geq 2$,那么 n_i 为 fork 任务。为了不失一般性,本文假定任务集只有一个开始任务和一个结束任务,并对每个任务节点 n_i 作一些相关规定:

a)最早开始时间 $\text{est}(n_i)$,如果 $\text{pred}(n_i)=\varnothing$,那么 $\text{est}(n_i)=0$ 即$\text{est}(n_{stat})=0$。

b)最早完成时间 $\text{ect}(n_i)=\text{est}(n_i)+w(n_i)$。

c)最佳前驱 $\text{fpred}(n_i)=n_j|\text{ect}(n_j)+c(n_j,n_i)>\text{ect}(n_k)+c(n_k,n_i),\forall j,k\in\text{pred}(n_i)$,且$j\neq k$。

d)次最佳驱 $\text{cpred}(n_i)=n_j|\text{ect}(n_j)+c(n_j,n_i)>\text{ect}(n_k)+c(n_k,n_i),j\neq k,\forall j,k\in\text{pred}(n_i)$且$j,k\neq\text{fpred}(n_i)$。

e)MAKESPAN。给定任务集从开始执行到所有任务都完成的时间,即 $\text{MAKESPAN}=\text{ect}(n_{exit})-\text{est}(n_{stat})=\text{ect}(n_{exit})$。

f)PROC_NUM。将给定的任务集,根据一定的调度规则和调度策略全部分配给所需要的处理器数目。

g)S 表示 DAG 图中有最多前驱的任务节点 n_i 的前驱节点数目,$S=\text{MAX\_NUM}(\text{pred}(n_i))$。

将分配到同一个处理器上的任务成为一个簇(cluster),记为 C_i。对于 C_i 中的每个任务 n_k,存在两个值,即 $\text{stc}_i(n_k)$和 $\text{ctc}_i(n_k)$,分别表示在簇 C_i 中任务 n_k 的开始执行时间和结束时间,$ct(C_i)$表示簇 C_i 的完成时间,即簇中最后一个任务的完成时间。如果以任务 n_i 作为簇 C_i 的最后一个任务,且在该簇中任务 n_i 有最早开始执行时间,则 C_i 表示为 $C(n_i)$。假设任务是非抢占性的,同一簇中任务间的通信量为 0,簇中的任务满足如下条件[1]:

a)$\text{ctc}_i(n_k)=\text{stc}_i(n_k)+w(n_k)$;

b)对于任意任务 $n_k,n_j\in C_i$,$\text{ctc}_i(n_k)\leqslant\text{stc}_i(n_j)$或 $\text{ctc}_i(n_j)\leqslant\text{stc}_i(n_k)$;

c)如果 $n_k,n_j\in C_i$ 且 $e_{kj}\in E$,则 $\text{ctc}_i(n_k)\leqslant\text{stc}_i(n_j)$;

d)如果 $n_k\in C_i-C_j,n_j\in C_j$ 且 $e_{kj}\in E$,那么 $\text{ctc}_i(n_k)+c_{kj}\leqslant\text{stc}_j(n_j)$。

2 新调度算法

对于任务 n_a,如果 $\text{pred}(n_a)=\{n_1,n_2,\cdots,n_s\}$,那么 $C(n_a)$则由 $C(n_1),C(n_2),\cdots,C(n_s)$和任务 n_a 通过优化得到。$\text{link}(n_k,n_a)=C(n_k)+c_{ka}(k=1,2,3,\cdots,s)$。同时为了研究的方便,在本文先假设:

$$\text{link}(n_1,n_a)\geqslant\text{link}(n_2,n_a)\geqslant\cdots\text{link}(n_{s+1},n_a)=0 \quad (1)$$

那么 n_1 是 n_a 的最佳前驱任务,n_2 为次最佳前驱任务。此外值得注意的是:在上面的定义中,$\text{pred}(n_a)=\{n_1,n_2,\cdots,n_s\}$,对于任务 n_a 的前驱编号,只是为了说明的方便,在实际的 DAG 图中,对于任务节点 n_a 的前驱任务,根据式(1)用相对应的任务节点相对应地替代 $n_1,n_2,\cdots,n_s$。由于本文算法要首先保证各子任务的最早执行,而对整个任务集来说,保证 n_{exit}具有最早开始时间,这样才可以使整个任务集的 MAKESPAN 最小。所以在构成 $C(n_a)$时,只要能保证 n_a 具有最早开始时间的要求,就要对满足条件的所有 n_a 的父任务进行复制,然后将在不影响整个任务集的 MAKESPAN 的前提下,为了在合并过程中尽可能地减少簇数目而适当对还没被复制的前驱任务进行复制,最后对所有的簇进行合并处理。因此,在基于前驱任务复制的前提下本文算法主要有三个阶段:a)为获得最早开始时间的第一次复制阶段;b)为簇合并服务的第二次复制阶段;c)簇的合并阶段。算法描述如下:

```
algorithm Schedule (V,E,w,C)
begin
est(n1)←0;C(n1)←{n1};
a←2;PrecursorTaskSet←pred(na);
while(a≤|V|)do
while(|PrecursorTaskSet|>0)do
s←(the number of (pred(na)));
S←MAX_NUM(pred(ni))
sortTaskSet(na,pred(na));
endwhile
k← 1;Ctmp(na)← C(n1)Una;k++;
while(ESTtmp(na)+w(nk+1)≤Link(nk+1,na))do
obtain_EST(na);
updateClus(Ctmp(na),ESTtmp(na));
m←(k+2);
while((INTERVAL>0)&& (s≥m))do
if((INTERVAL≥w(nm))&&(succ(nm)=1))
prepare_ Merger(na,nm);
constructClus(Ctmp(na));m++;
else m++;
endwhile
endwhile
clusMerge(C(na));
putout( );
end
```

在算法 Schedule(V,E,w,c)中,首先通过函数 sortTaskSet$(n_a,\text{pred}(n_a))$使任务集得到每个任务的前驱有序任务集;然后为了得到 $C_{tmp}(n_a)$并通过函数 obtain_Est(n_a)对满足条件的前驱任务进行复制操作,当这个过程结束后,为了在合并中能尽可能减少所需簇的数目,将对满足条件的任务进行 prepare_Merger(n_a,n_m)操作;最后对簇进行合并 clusMerge$(C(n_a))$。

在簇的形成中,首先要计算每个任务的 est,因此要计算 V 次,其对每个任务的前驱任务复制中,最多复制 S 个。由于在对簇的初步形成过程中是由为获得最早开始时间而对前驱任务的复制和基于为簇合并服务的前驱任务复制二阶段构成,所以簇的形成时间复杂度是 $O(S^2v)$,此外簇的合并时间复杂度是 $O(v)$,总体算法的时间复杂度是 $O(S^2v)+O(v)=O(S^2v)$。表 1 给出了本文算法与 LG、PPA 的时间复杂度的比较。相对而言,S^2 一般都要远远小于 v 和$|E|$,所以本文算法与 LG、PPA 比较起来时间复杂度更小。

表 1　三种算法的时间复杂度对比

	算法		
	LG	PPA	本文算法
时间复杂度	$O(\|E\|Logv)$	$O(v^2)$	$O(S^2v)$

2.1　为获得最早开始时间的第一次复制阶段

如图 1(a)所示,任务 n_a 的全部前驱任务先按照式(1)排列,构成前驱任务簇集,设整形变量 $k=0$, 然后将 n_a 和最佳前驱任务所在簇 $C(n_1)$ 合并为临时簇 C_{tmp},$k++$,如图 1(b)所示。假设任务 n_a 在 C_{tmp} 中不需要等待其他簇中前驱任务通信时的临时最早开始时间 $EST_{tmp}(n_a)$ 的值如下:

$$\begin{cases} EST_{tmp}(n_a)=ct(c_1) & \text{if} \quad k=1 \\ EST_{tmp}(n_a)=(ct(C_1)+\sum_{x=z,x++}^{x\leqslant k} w(n_x)) & \text{if} \quad k\geqslant 2 \end{cases} \tag{2}$$

$$EST_{tmp}(n_a)+w(n_{k+1})\leqslant \text{link}(n_{k+1},n_a) \tag{3}$$

假设任务 n_a 在 C_{tmp} 中需要等待其他簇中前驱任务通信时的临时最早开始时间为 $est_{tmp}(n_a)$。$EST_{tmp}(n_a)$ 与 $est_{tmp}(n_a)$ 的关系为

$$est_{tmp}(n_a)=\max(EST_{tmp}(n_a),\text{link}(n_{k+1},n_a)) \tag{4}$$

如果满足式(3),那么就将 n_{k+1} 复制到 C_{tmp} 中,处于 n_k 与 n_a 之间,并且 $k++$,此时 C_{tmp} 更新为新的临时簇;然后判断式(3)是否成立,如果仍然满足式(3),那么就继续将 n_{k+1} 复制到 C_{tmp} 中,位于 n_k 与 n_a 之间,$k++$,并通过 updateClus(C_{tmp}(n_a),$EST_{tmp}(n_a)$)更新 C_{tmp} 和 $EST_{tmp}(n_a)$。不断循环上述操作,对其他前驱任务进行判断,如图 1(c)所示,直到式(3)不成立退出。当上述过程结束后,k、$EST_{tmp}(n_a)$、$est_{tmp}(n_a)$ 均为定值,此时 $est_{tmp}(n_a)=est(n_a)$。

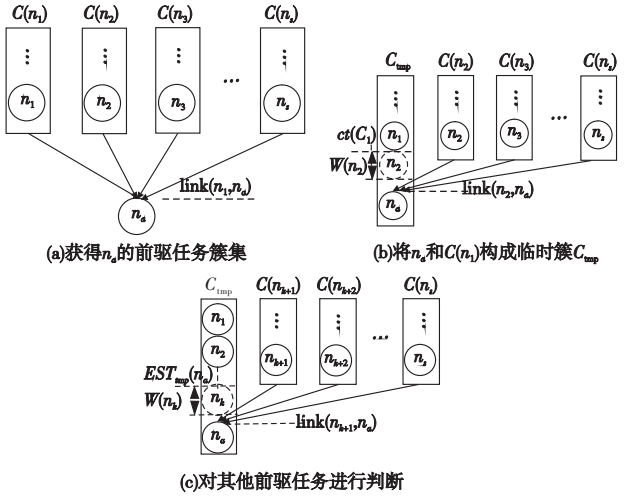


图1　为获得最早开始时间的第一次任务复制阶段

由于第一阶段是对满足条件的前驱任务进行复制,其原理就是让临时簇在因为获得被复制任务的执行时间所付出的代价小于由于该被复制前驱任务不被复制而产生通信的代价。所以对满足式(3)的前驱进行复制,能使任务 n_a 获得更短的执行时间。如果对于满足条件而复制的父节点数目越多,$est(n_a)$ 越小,在最理想的情况下,任务 n_a 的所有前驱任务都可以复制到同一个簇上,在最不理想的条件下,只能复制最佳前驱任务。此时 n_a 同样有最早开始执行时间。

2.2　为簇合并服务的第二次复制阶段

接下来将进行为簇合并服务的第二次任务复制阶段,经过阶段一之后,此时任务 n_a 已经具有最早执行时间,设在簇 C_{tmp} 中,任务 n_a 的前一个任务是 n_{key},$ct(n_{key})$ 是任务 n_{key} 在该 C_{tmp} 中的完成时间,设 INTERVAL 为任务 n_{key} 与任务 n_a 之间的时间空闲,,INTERVAL 的初值如下:

$$\text{INTERVAL}=(est(n_a)-EST_{tmp}(n_a)) \tag{5}$$

很明显,INTERVAL≥0。由于任务 n_{k+1} 与 n_s 之间的任务在基于获得最早任务开始时间的过程中并没有被复制,在本文中统称这些任务为次考虑前驱任务。设置一个整形变量 m,令 $m=k+2$,首先从任务 n_m 开始:

$$\text{INTERVAL}\geqslant w(n_m)\&\&\quad succ(n_m)=1 \tag{6}$$

关键步骤:如图 2(a)所示,如果满足式(6),那么就将任务 n_m 复制到 C_{tmp} 中,n_m 的位置位于 n_{key} 与 n_a 之间,并执行式(7),此时任务 n_m 变成了 n_{key},更新 C_{tmp} 及 n_{key},然后 $m++$。

$$\text{INTERVAL}=\text{INTERVAL}-w(n_m) \tag{7}$$

如果不满足条件式(6),如图 2(b)所示,那么同样 $m++$,继续执行关键步骤,直到 $m>s$ 或者 INTERVAL=0;退出循环,此时为簇合并服务的第二次任务复制阶段结束。

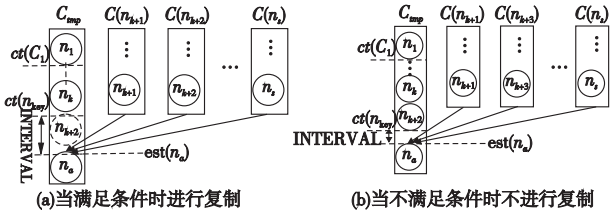


图2　为簇合并服务的第二次任务复制阶段

由于 INTERVAL 实际上是 n_a 在 C_{tmp} 中的时间空闲,如果不满足式(6),那么就不复制次考虑前驱任务,这样 $est(n_a)$ 的值保持不变;如果满足式(6),由于复制次考虑前驱任务,只是将复制的任务填充 INTERVAL,并没有影响到 $est(n_a)$,所以不论是否复制次考虑前驱任务,都不会影响 $est(n_a)$。对于次考虑前驱任务中 $succ(n_i)=1$ 的任务,如果这些任务不被复制到子簇中,那么在构成簇的过程中将构成一个单独的簇,并且在簇的复制过程中将无法被合并,所以对于这种任务,尽可能地被复制到子簇中,但是考虑到被复制后不能影响到子节点的最早开始时间,所以只有满足式(6)才既可以减少簇的数目,又不影响子节点的最早开始时间。由于 n_{exit} 是最后一个任务,所以不会影响 $est(n_{exit})$,这样整个任务集的 MAKESPAN 就不会受影响,并且经过簇的合并过程之后,保留的簇的数目越少即所需的处理器数目也就越少。

2.3　簇的合并阶段

在基于复制的前期条件下,经过前面两个阶段,接着对所有的簇进行合并处理。合并的前提条件是:

a)如果一个簇 $C(n_i)$ 所有的任务都被另外一个簇 $C(n_j)$ 所包含。

b)如果 $C(n_i)$ 被合并到 $C(n_j)$ 中,对于 $n_k\in succ(n_i)$,合并不会影响 $C(n_k)$,即不影响 $est(n_k)$。

只有符合合并条件才进行合并。由于 n_{exit} 是最后一个任务,所以不会影响 $est(n_{exit})$,这样整个任务集的 MAKESPAN 就不会受影响。

3　实例说明以及实验对比

本章将用本文算法与传统算法(LG、PPA)在实验环境相

同的前提下对具体的 DAG 图(图 3)进行调度,并以 MAKESPAN 和 PROC_NUM 两方面的性能来作对比,对比结果如图 4 所示。此外针对不同任务数目的 DAG 图,对三种算法进行同样的实验对比,得出的结果如表 2 所示。

表 2 不同 DAG 图的三种算法比较

算法		节点数						
		10	15	21	32	36	41	50
LG	MAKESPAN	21	23	30	34	42	46	60
	PROC_NUM	5	5	6	9	10	10	12
PPA	MAKESPAN	20	22	25	27	32	35	52
	PROC_NUM	5	5	5	6	8	8	10
本文	MAKESPAN	20	22	23	27	31	33	46
算法	PROC_NUM	3	4	4	5	6	7	8

通过表 2 可以看出,随着 DAG 图中任务节点的增加,本文算法比 LG 和 PPA 算法在 MAKESPAN 和 PROC_NUM 性能上的优势越明显。

图 3 中,圆圈代表任务节点,其圆的上半圆表示任务 n_i 节点编号 i,下半圆表示任务的计算时间开销,有向边上的数字表示节点和节点的通信值,边的方向表示节点的前后关系,其中节点 1 是入口节点,节点 10 是出口节点。对于任务节点 1,$\text{est}(n_1)=0$,$\text{ect}(n_1)=w(n_1)=3$;对于节点 2、3、4、5,其只有一个前驱节点 1,节点 1 也可看成这些节点的最佳前驱节点,所以直接将节点 1 复制并与子节点结合;对于节点 6,其最佳前驱节点是 2,所以优先将节点 6 与 $C(n_2)$ 簇结合在一起,形 C_{tmp},$\text{est}_{\text{tmp}}(n_6)=\max(\text{EST}_{\text{tmp}}(n_6),\text{link}(n_3,n_6))$,此时节点 3 仍满足式(3),即 $\text{EST}_{\text{tmp}}(n_6)+w(n_3)\leqslant\text{link}(n_3,n_6)$,所以 $\text{est}_{\text{tmp}}(n_6)$ 可以因为复制任务 n_3 而变得更小。复制节点 n_3,置于 n_2 与 n_6 之间,此时由于节点 n_4 不满足式(3),$\text{est}_{\text{tmp}}(n_6)=\max(\text{EST}_{\text{tmp}}(n_6),\text{link}(n_4,n_6))$ 等于 $\text{est}(n_6)$,所以这个时候 $\text{est}_{\text{tmp}}(n_6)$ 最小。接下来考虑为簇合并服务的第二次复制。这里的 INTERVAL 初值是 $(\text{est}(n_6)-\text{EST}_{\text{tmp}}(n_6))$,$n_{\text{key}}$ 是 n_3,经过判断满足式(6),将 n_5 复制到 C_{tmp} 中,同时更新 C_{tmp},此时 n_5 成为 n_{key},已经不满足式(6),所以复制过程结束。对于节点 n_6,其簇 $C(n_6)=\{n_1,n_2,n_3,n_5,n_6\}$;同样对于节 n_7、n_8、n_9、n_{10},就可以采取类似 $C(n_6)$ 的形成步骤。$C(n_7)$、$C(n_8)$、$C(n_9)$、$C(n_{10})$ 其最后分簇结果如表 3 所示。

表 3 各任务节点的分簇结果

任务节点	fpred	est	ect	$C(n_i)$
n_1		0	3	1
n_2	n_1	3	7	12
n_3	n_1	3	6	13
n_4	n_1	3	7	14
n_5	n_1	3	5	15
n_6	n_2	12	16	12356
n_7	n_4	9	15	147
n_8	n_4	8	11	148
n_9	n_3	9	12	139
n_{10}	n_7	18	20	147810

其中在任务分簇之后,根据合并条件,对生成的簇进行第三阶段的合并操作,经过簇的合并,最后保留了 $C(n_6)$、$C(n_9)$、$C(n_{10})$。

对于图 3 的 DAG 图,本文算法与两种典型算法(LG、PPA)的调度结果对比如图 4 所示,$P_i(i=1,2,3,\cdots)$ 表示所需处理器的编号,$n_i(i=1,2,3,\cdots)$ 表示任务节点,箭头表示不同簇(处理器)之间的通信,箭头上的数字表示通信代价,其中黑色阴影部分表示处理器的空闲(等待)时间。对于图 3 中的特定 DAG 图,从图 4 中可以看到,本文的调度算法在 MAKES-PAN 和 PROC_NUM 上优于 LG 算法,与 PPA 算法的调度长度一致,在 PROC_NUM 上优于 PPA 算法。

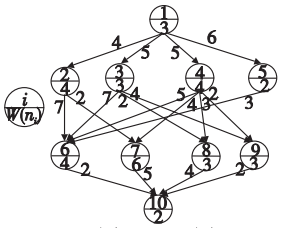
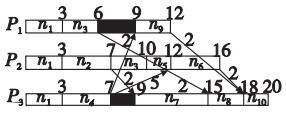
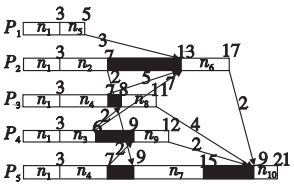


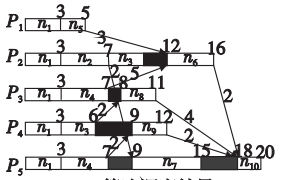
图3 DAG图



(a)本文算法调度结果



(b)LG算法调度结果



(c)PPA算法调度结果

图4 三种算法的调度对比

4 结束语

本文针对典型算法中存在的不足,提出了一种新的调度算法,该算法为了使 MAKESPAN 以及簇的数目尽可能少而进行了两个阶段的前驱任务复制,所以能够在 MAKESPAN 和处理器的数目上优于 LG 和 PPA 算法,具有更小的时间复杂度 $O(S^2v)$,对并行计算中任务调度性能的提高具有一定的意义。

参考文献:

[1] 周双娥,袁由光,熊兵周,等. 基于任务复制的处理器预分配算法[J]. 计算机学报,2004,27(2):216-22.

[2] LIN Wei-ming, GU Qiu-yan. An efficient clustering-based task scheduling algorithm for parallel programs with task duplication[J]. Journal of Information Science and Engineering, 2007, 23(2): 589-604.

[3] DARBHA S, AGRAWAL D P. Optimal scheduling algorithm for distributed-memory machines[J]. IEEE Trans on Parallel and Distributed Systems, 1998, 9(1): 87-95.

[4] 何琨,赵勇,黄文奇. 基于任务复制的分簇与调度算法[J]. 计算机学报,2008,31(5):733-740.

[5] RUAN You-lin, LIU Gan, ZHU Guang-xi, *et al*. An optimal scheduling algorithm based on task duplication [J]. Journal of Systems Engineering and Electronics, 2005, 16(2): 445-450.

[6] EBAID A, AMMAR R, RAJASEKARAN S, *et al*. Task clustering & scheduling with duplication using recursive critical path approach (RCPA)[C]//Proc of IEEE Signal Processing and Information Technology. 2010: 15-18.

[7] GENG Xiao-zhong, XU Gao-chao, WANG Dan, *et al*. A task scheduling algorithm based on multi-core processors[C]//Proc of International Conference on Mechatronic Science, Electric Engineering and Computer. 2011: 942-945.

[8] SHI K S, CHA M J, JANG M S, *et al*. Task scheduling algorithm using minimized duplications in homogeneous systems[J]. Journal of Parallel Distributed Computing, 2008, 68(8): 1146-1156.