基于最佳并行度的任务依赖图调度*

杜建成 黄 皓 陈道蓄 谢 立

(南京大学计算机软件新技术国家重点实验室 南京 210093)
(南京大学计算机科学与技术系 南京 210093)

摘要 基于最佳并行度的任务依赖图调度策略充分利用编译时刻所得到的全局信息,采用横向和纵向任务合并,处理节点预分配,静态调度和动态调度相结合、集中式调度和分层调度相结合等措施,是一种简单的、具有较高效率的实用化调度方案.该调度方案能够在尽量压缩调度长度的情况下节约系统资源.

关 键 词 层次任务图,任务依赖图,静态调度,动态调度,最佳并行度.

中图法分类号 TP311

编写并行程序是一件相当复杂和困难的事情,自动并行编译的出现为充分有效地利用各种形式的并行体系结构提供了强大的支持.在进行串行程序的任务并行性分析时,首先要将整个程序划分成若干个模块或任务,然后分析这些任务的读写集,判定是否存在读写冲突,若无冲突,则二者可以并行执行.我们根据模块的自然边界进行任务的划分,这种划分往往可以提供较好的数据局部性.结果得到层次任务图 HTG(hierarchical task graph).下一步的工作便是在并行环境中调度执行 HTG 中的任务.无论是在共享存储环境还是在分布存储环境,都常常采用动态调度和静态调度相结合的方法.静态调度是指在编译时刻所进行的调度,有时也称为预调度,可以利用编译时得到的全局信息进行调度优化.动态调度是指在程序执行过程中所进行的调度,具有较多的灵活性.图 1 是该系统的框架结构.其中:(1) 数据/控制依赖关系分析:分析程序中各个模块之间的依赖关系,分析结果是层次任务图 HTG;(2) 任务粒度和通信代价预估:对各个任务的执行时间和任务间的通信量进行评估,并确定复合任务粒度的上限 UL.

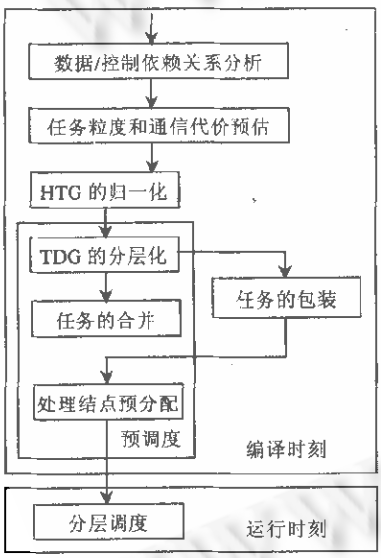

图 1 系统框架

HTG 的归一化:将任务粒度大于 UL 的复合节点的 HTG 图嵌入到初始 HTG 中,最终形成一个单一的任务依赖图 TDG(task dependence graph).

TDG 的分层化:用先广搜索的方法对 TDG 进行分层处理,使得同层任务之间没有依赖关系,可以并行执行,有依赖关系的节点分布在不同层中.结果形成分层任务图 LTG(layout task graph).

任务的合并:由于最佳并行度和可用的处理节点的限制,并且为了实现任务粒度的均衡化,对同一层中的某些任务进行横向合并,同时,为了消除不必要的通信,在不同层中进行纵向合并.

* 本文研究得到国家 863 高科技项目基金资助.作者杜建成,1971 年生,博士,主要研究领域为并行编译.黄皓,1957 年生,副教授,主要研究领域为网络计算.陈道蓄,1949 年生,教授,主要研究领域为并行计算,分布式处理.谢立,1942 年生,教授,博士生导师,主要研究领域为并行计算,分布式处理.

本文通讯联系人:杜建成,南京 210093,南京大学计算机软件新技术国家重点实验室

本文 1998-06-30 收到原稿,1998-10-14 收到修改稿

任务的包装:在每个任务中添加必要的成分,使之成为可以独立编译和运行的单位.

处理节点预分配:让每个任务尽量分配在其前导节点所在的处理节点上,以便压缩通信开销.

分层调度:在程序运行时刻,接受前导任务发来的消息,将下一层的所有任务调度激活.

从图 1 中可以看出,我们采用了静态调度和动态调度相结合的方法,静态调度(预调度)部分包括 TDG 的分层化、任务的合并和处理节点预分配,分层调度属于动态调度.

1　相关工作

一般形式的优化调度问题都是 NP 难题[1],因此,实用的调度算法都是经验算法.文献[2]给出了一些算法,如 HLF(high level first),CP(critical path),LPT(largest processing time),但是这些算法都基于共享存储模式,不考虑通信代价,不能很好地应用于分布存储系统.

文献[3]给出了一个可伸缩的调度算法 SSS(scalable schedule scheme).SSS 给出一个阈值,在调度长度和并行度之间取得折衷.其中的处理节点竞争算法仅考虑一个前导任务的若干个后继任务竞争处理节点的情况,而没有考虑一组前导任务的若干个后继任务相互竞争的情况.

文献[4]给出了共享存储模式下任务依赖图的调度.它将任务分为并行任务和串行任务两类,串行任务比并行任务具有更高的优先级,以便于消除调度中的瓶颈,但在分布存储系统中,动态调度并行任务很困难.

本文力图解决这样一个问题:如何在尽量压缩调度长度的情况下节约系统资源.本文给出最佳并行度的概念,在调度及通信开销与程序并行性之间取得折衷,通过合并达到任务粒度的均衡化,通过后继任务间处理节点的分配算法来压缩通信开销.实验表明,当存在大量的并行小任务的情况下,本文的调度方案实现了上述目标.

2　预调度

2.1　几个约定

(1) TDG 和 LTG 的约定

- TDG 和 LTG 均是无环有向图(directed acirclic graph,简称 DAG).
- 前导任务与后继任务之间的先后或依赖关系用通信来表达,send 原语为非阻塞通信原语(nonblocking),receive 原语为阻塞通信原语(blocking).
- 任务是非严格的,即后继任务可以在前导任务结束之前被调度,但若后继任务未完全获得所需要的依赖信息,将处于通信等待状态.

(2) 执行环境的约定

分布存储系统中的各个节点同构,任意两个节点之间通信速率相同.

(3) 任务模型

一个任务由以下 4 个部分组成,如图 2 所示.

- receive 区:接受来自前导任务的消息.
- 计算区:任务的主体,完成本任务的所有计算.
- 调度区:向调度器发出消息,指示调度器将后继任务调度激活.
- send 区:向其后继任务发送消息.

图 2　任务模型

并不是所有的任务都包含这 4 个区,例如 TDG 的首任务没有接受区,尾任务没有发送区,LTG 中的每一层只需一个任务有调度区,因为调度器一次可以将下一层的所有任务激活.

(4) 调度模型

采用动态调度和静态调度相接合的策略,动态调度采用集中式方法,由一个调度器利用静态调度的结果,随时接受来自某一任务调度区的消息,对 LTG 实施调度.调度器每次调度 LTG 的一层中的所有任务,直至尾任务执行完毕.

2.2 TDG的分层化

为了便于进行任务的合并和调度等后继工作,首先将 TDG 分层化,使同层中的任务之间没有依赖关系,进而可以并行执行,有依赖关系的节点分布在不同层中.节点 p 依赖于 q,则称 q 为 p 的前导节点,p 为 q 的后继节点.没有前导节点的节点为首节点,没有后继节点的节点为尾节点.算法 1 采用先广搜索的方法对 TDG 遍历,给每个节点一个标记,该标记指出节点的层号,这样就完成了 TDG 的分层化,结果形成分层任务图.

算法 1. TDG 的分层化

```
procedure  LTDG
    Q=nil;
    l=0;
    for I=1 to n
          L(n_i)=l;
    Q=Q∪{n_first};
    L(n_first)=l;
    while (Q<>nil)
        {
         l=l+1;
         q=a node from Q with indgree equal 0;
         Q=Q-{q};
         for (all nodes n adjacent to q)
              {
              L(n)=max(L(n),1);
              Q=Q∪{n};
              delete all arcs incident to this node;
              }
        }
end
```

2.3 任务的合并

2.3.1 最佳并行度

考察如下结构的简单 TDG,如图 3 所示.$T_i(1\le i\le n)$依赖于 T_0,T_0 执行完之后,T_i 可以并行执行.可以申请的处理节点数大于等于 $n+1$,当 T_0 执行到调度区时,向调度器发送消息,调度器接受到消息,将 $T_i(1\le i\le n)$全部调度激活,然后调度器向 T_0 发送消息,报告 $T_i(1\le i\le n)$所在的位置,T_0 进入发送区,向 $T_i(1\le i\le n)$发送消息,$T_i(1\le i\le n)$接受到消息便立即开始执行.T_0,$T_i(1\le i\le n)$和调度器执行时的时序关系如图 4 所示.

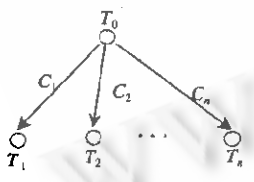

图 3 简单的 TDG

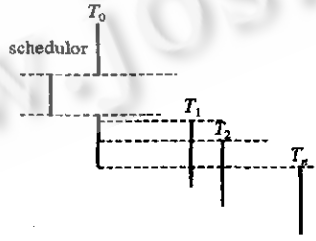

图 4 任务执行时序图

T_0 与 $T_i(1\le i\le n)$之间的通信代价 $C_i=I+S*D_i$,I 为通信启动代价,S 为发送单位数据量的通信代价,D_i 为 T_0 与 $T_i(1\le i\le n)$之间的通信量.调度一个任务的平均开销为 SH,T_0 与 $T_i(1\le i\le n)$的执行开销分别为 t_0 与 $t_i(1\le i\le n)$.为简单起见,假定 D,t_i 分别为常数 D,t,C_i 为常数 $C=I+S*D$,T_0 与 $T_i(1\le i\le n)$之间的通信开销始终存在,则 $T_i(1\le i\le n)$的调度次序不对 TDG 的执行时间产生影响.在这种情况下,图 3 中 TDG 的执行时间为

$$CT=t_0+n*SH+\max_{1\le i\le n}(\sum_{j=1}^{i}C_j+t_i)=t_0+n*SH+\max_{1\le i\le n}(i*C+t)$$
$$=t_0+n*SH+n*C+t=t_0+n*SH+n*I+S*D*n+t. \quad (1)$$

然后将 T_0 的 n 个后继任务 $T_i(1\le i\le n)$均匀合并为 m 个任务 $T'_i(1\le i\le m)$,$m\le n$,$m|n$,此时,每个新任务 $T'_i(1\le i\le m)$包含 n/m 个原任务,因此每个新任务的执行时间相同,均为 $t'=t*n/m$,$T'_i(1\le i\le m)$与 T_0 的通信代价 $C'=I+S*D*n/m$,故合并后 TDG 的完成执行时间为

$$CT'=t_0+m*SH+m*C'+t'=t_0+m*SH+m*(i+S*D*n/m)+t*n/m$$
$$=t_0+m*SH+m*SH+m*I+S*D*n+t*n/m. \quad (2)$$

由式(1),(2)可得:当 $t/(SH+I)\le m\le n$,且 $m|n$ 时,

$$CT'\le CT(3). \quad (3)$$

下面求使得 CT'最小的 m 值.为此,先对式(2)针对 m 求导:

$$(CT')'=(SH+I)-t*n/m2$$

当$(CT')'=0$,即$(SH+I)-t*n/m^2=0$ 时,CT'最小,此时 $m=\sqrt{n*t/(SH+I)}$.结合结论(3),我们给出如下定义:满足 $t/(SH+I)\le m\le n$,$m|n$,且最接近 $\sqrt{n*t/(SH+I)}$ 的 m 为最佳并行度,记为 pm,同时定义最佳粒度 $pt=n*t/pm=\sqrt{n*t/(SH+I)}$.以图 3 为例,若 $n=60$,任务粒度相等,均为 $t=12$,平均调度开销 $SH=2$,任务起动时间 $I=1$,$t_0=10$,$D=1$,$s=1$,求得最佳并行度为 15.事实上,通过表 1 也可以看出,当 $m\le t/(SH+I)=4$ 时,$CT'\le CT$,当 $m=15$ 时,CT'最小.

表 1　并行度与 TDG 的执行时间

m	60	30	20	15	12	10	5	4	3	2	1
CT'	262	184	166	163	166	172	229	262	319	436	793

以上为了简单、方便起见,假定任务粒度和通信开销均相同.在实际应用中,情况不会是这样,但当并行任务数较多,且通信开销小于任务执行开销时,仍然可以近似地认为通信开销均相同,让 t 取任务的平均粒度,简单地定义最佳并行度 $pm=\lfloor\sqrt{n*t/(SH+I)}\rfloor$.总之,最佳并行度给出了调度、通信开销与并行性之间的一个折衷.

需要说明的是,最佳并行度指出了在任务可以并行执行情况下“最好的”并行任务个数,它不能决定任务并行执行是否一定比串行好.这也解释了 pm 的计算公式中没有出现任务通信量的原因.

2.3.2　任务的合并

由于特定执行环境中调度和通信开销的存在,使得任务粒度不能太小.而在原来的 LTG 中常常存在大量的小任务,因此有必要进行任务的合并.我们的任务合并包括两步,先在同层内进行横向合并,然后在不同层间进行纵向合并.

横向合并时,对 LTG 中的每一层,判定该层任务是否存在最佳并行度 pm,若存在,则按 pm 进行任务的合并.如果可申请到的处理节点数 ap 小于 pm,则按 ap 进行任务的合并.合并后的任务粒度应尽量趋于一致,使得该层中的任务并行执行时间最短.由于任务的均匀合并是一个 NP 问题,因此我们给出一个简单的经验化算法,如算法 2 所示.该算法对 LTG 进行操作,任意节点 v 的权为 cw,表示 v 的执行开销,任意边 e 的权为(c,s,d),其中 c 表示通信内容,s 为消息源,d 为消息目的.

纵向合并时,对 LTG 中任一节点,若其出度为 1,并且与之相联系的后继节点入度为 1,则将这两个节点合并成一个节点.

这种合并方法具有简单、可伸缩性好的优点,并且不会破坏原有的 LTG 结构.

任务合并的一个极端情况是,整个 LTG 图被合并成一个节点,这代表了如下一种情况:虽然在串行程序中存在着并行性,但由于可并行任务粒度太小,使得在特定执行环境下并行执行没有意义,因此本系统指示其依然按照串行方式执行.

在进行任务横向合并时,需要知道每个任务的粒度.在编译时刻进行任务粒度分析已成为目前的研究热点[5].但是决定任务大小的一些参数并不都能在编译时刻确定,在这种情况下我们只能进行保守估计.保守估计的不精确性表现在将一些实际上的小任务估计为大任务,这在一定程度上影响了合并后的任务均衡性.我们采用的补救措施是:为 LTG 中某一层提供若干个可能的合并方案,当系统将要执行该层任务时,在大部分情况下关于该层的任务大小信息已经明朗了,系统可以较为容易地选择更加合适的合并结果去执行.

2.3.3 处理节点预分配

LTG 在被执行时所需申请的处理节点的个数由 LTG 中"宽度"最大的层中的并行任务确定,设为 *pnum*,可使用的处理节点的集合记为 *P*.当任务 v_i 与其某个前导任务 v_i^j 分配在同一个处理节点 $p_k(1\le k\le pnum)$上时,记为$(p(v_i^j),v_i),p(v_i^j)=p_k$,其通信开销将减少,因此在调度时应使每一个任务尽量分配在其前导任务所处节点之上,称为任务分配的亲近性.由于一个后继任务可能有若干个前导任务,一个前导任务可能有若干个后继任务,因此在给 LTG 中某一层的所有任务分配处理节点时,就存在选择与竞争两种情况.设合并后的分层任务图的第 t 层有 k 个任务 $v_1,v_2,\ldots,v_k$,$v_i(1\le i\le k)$位于 $t-1$ 层的前导任务集 $PS(v_i)=\{v_j^i|1\le j\le n_i\}$,称为最近前导任务集.算法 3 用一种简单的方式解决 t 层中任务的处理节点预分配问题.对 LTG 中各层依次采用算法 3 进行任务的处理节点预分配.图 5 为 LTG 的一部分,*a*,*b*,*c*,*d* 这 4 个节点的分配方案如表 2 所示.

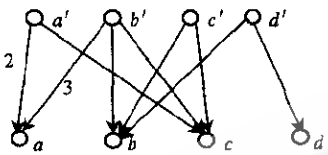

图 5 LTG 的一部分

表 2 处理节点预分配表

排序	$PS(d)=\{d'\}$	$PS(a)=\{a',b'\}$	$PS(c)=\{a',b',c'\}$	$PS(b)=\{b',c',d'\}$
$(p(d'),d)$		$PS(a)=\{a',b'\}$	$PS(c)=\{a',b',c'\}$	$PS(b)=\{b',c'\}$
重排序		$PS(a)=\{a',b'\}$	$PS(b)=\{b',c'\}$	$PS(c)=\{a',b',c'\}$
$(p(b'),a)$			$PS(b)=\{c'\}$	$PS(c)=\{a',c'\}$
$(p(c'),b)$				$PS(c)=\{a'\}$
$(p(a'),c)$				

3 分层调度

分层调度部分可以在编译后端,程序运行前进行,但我们将它推迟到运行时刻,这有以下几个原因:(1) 一开始将执行条件得不到满足的任务调度到各个处理节点上去,浪费大量的存储空间;(2) 在程序执行过程中,尽管这些任务被激活时,发现执行条件得不到满足会立即放弃对 CPU 的占用,但进程间的反复切换开销不容忽视,尤其在任务很多的情况下;(3) 由于很多信息都在预调度部分计算出来,因此分层调度本身的开销很小.

算法 2. LTG 中任务的合并

```
Procedure  LTGmerge
{
    for i=1 to m
      {
        t=第 i 层所有任务执行开销之和 acl第 i 层任务数 tn;
        pm=⌊√(n*t/(SH+I))⌋;
        rp=min(ap,pm);
        at=ac/rp;
        if (tn>rp) {
                Q=第 i 层所有任务按执行开销 c 的降序排列;
                Q'=Q 的末尾的 tn-rp 个元素;
                Q=Q-Q';
                do {q=Q 的首元素;
```

```
                    if (wq≥at) Q=Q-{q}
                      else break; }
                while(Q<>nil);
                while(Q'<>nil)
                  {q'=Q'的首元素;
                   Q'=Q'-{q'}
                   lq=Q 的尾元素;
                   lq=merge(lq,q');
                   if (w_lq<at)
                      将 lq 插入 Q,使 Q 保持执行开销 c 的降序排列;}
                }
        }
}
Function   merge(v,v')
{
   for (每一个指向 v'的边 e(c,s,v'))
        {让 e 指向 v;
         修改 e 的权为 e(c,s,v);}
   for (由 v'发出的边 e(c,v',d))
        {让 e 由 v 发出;
         修改 e 的权为 e(c,v,d);}
   修改 v 的权为 v(cv+cv');
   从 LTG 中删除 v';
   return(v);
}
```

我们的调度模型采用集中式模式,因为集中式调度器的设计较为简单,调度器每次调度 LTG 的一层中的所有任务,调度时刻由上一层中最先结束的任务决定(在 2.1 节中提到,我们的任务是非严格的,这一点允许后继任务在前导任务结束前得到调度.)无论是集中式调度还是分布式调度,都存在着瓶颈问题,但是可以通过仔细的设计来消除这种瓶颈.任务合并和分层调度的结合可以较好地消除这种瓶颈,其中任务合并实现了负载平衡,而分层调度大大压缩了 LTG 与调度器的通信次数,事实上,如果 LTG 有 m 层,则 LTG 只需向调度器发出 $m-1$ 次调度请求.

算法 3. 同层任务处理节点预分配

```
Procedure   proallocate(t)
{
  Q=v1,v2,...,vk,按最近前导任务集数目增序的对列;
   P=P-{t-1 层中使用的所有处理节点};
   while(Q<>nil){
        v=Q 的首元素;
        Q=Q-{v};
        If (PS(v)<>nil){
             If (v^j∈PS(v)&&v 与 v^j 之间的通信开销最大)
               (P(v^j),v);}
        else {p=P 中的一个空闲处理节点;
             P=P-{p};
```

```
        (p,v);}
    for (Q 的每个元素 v')
        {if (v'∈PS(v')
          PS(v')=PS(v')-{v'};}
    对 Q 中元素按最近前导任务集数目增序排队;
}
```

4 实例研究

为了测试以上调度算法的性能,需要较多的处理节点,由于条件所限,我们主要采用模拟的方法.我们在一台 RS6000 工作站上用 21 个进程模拟 21 个处理节点,每个进程具有相同的优先级,为了使它们能够分得相同个数的时间片,当一个进程“空闲”时,即未被调度器赋予计算任务时,也让其处于计算状态,借助 PVM 来完成节点之间的通信,通过这种方式来模拟一个同构的计算环境.在该环境中,调度一个任务的平均开销为 30112μs,通信启动时间为 239μs.我们设计了两组实验,第 1 组用来对最佳并行度进行测试,第 2 组对整个调度算法的性能进行评估.

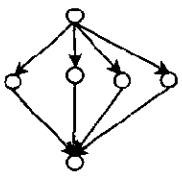

图 6 FMM 任务图

实验 1. 两个 $N*N$ 的浮点矩阵 AB 差乘(float matrix multiply,简称 FMM),并行执行时的任务图如图 6 所示.表 3~6 分别是 60*60,80*80,100*100,120*120 的矩阵差乘结果,时间以μs 为单位,取 10 次执行结果的平均值,以下均同.

表 3 60*60FMM

并行任务个数	1(串行)	2	3	5	6	10
执行时间	955 703	597 312	382 280	227 547	220 543	455 095
加速比	1	1.6	2.5	4.2	4.3	2.1
理论 *pm*				5		
实际 *pm*				6		

表 4 80*80 FMM

并行任务个数	1(串行)	2	4	6	8	10
执行时间	2 262 960	1 432 227	685 733	435 176	348 141	452 584
加速比	1	1.6	3.3	5.2	6.5	5.0
理论 *pm*				8		
实际 *pm*				8		

表 5 100*100 FMM

并行任务个数	1(串行)	2	4	5	10	20
执行时间	4 428 980	2 952 654	1 342 115	1 029 995	560 630	820 188
加速比	1	1.5	3.3	4.3	7.9	5.4
理论 *pm*				12		
实际 *pm*				10		

表 6 120*120 FMM

并行任务个数	1(串行)	4	6	10	12	15	20
执行时间	7 229 985	2 259 370	1 390 380	860 712	777 418	860 822	881 705
加速比	1	3.2	5.2	8.4	9.3	8.4	8.2
理论 *pm*				15			
实际 *pm*				12			

从以上几组实验可以看出,理论 *pm* 和实际 *pm* 吻合得较好,说明最佳并行度能够反映并行性与调度及通信开销的折衷,这一点从图 7 可以清楚地看出来.

实验 2. 为了对整个算法的性能进行评估,我们设计了塔形任务依赖图,如图 8 所示.图中每一层均比上

一层多两个任务,相邻两层形成一个完全二分图,任务的粒度控制在 5*104~21*104μs 之间,任务间的通信量控制在 1~5k byte 之间,整个任务依赖图包括 6 层,共计 36 个任务.对该任务图的一种简单调度方法(SS)是:尽量利用可以得到的资源,即为每一个可以执行的任务分配一个处理节点,由于处理节点数 21(一个节点被调度器占用)大于最大的任务宽度 11,故该条件能够得到满足.我们针对该任务图做了 20 组实验,表 7 给出了 3 组实验结果,每组实验将基于最佳并行度的调度方法(optimum degree parallelism-based scheduling,简称 OPBS)与上述的简单调度方法(SS)作了比较.表中可用 p 表示允许使用的处理节点数,实用 p 表示实际使用的处理节点数,均不包括调度节点.任务的串行执行时间等于该任务图中任务粒度之和.从表中可以看出,OPBS 方法要比 SS 方法效率高,并且可以尽可能地节约处理器资源.附带说明一下,两种调度算法的加速比随处理节点个数的增多提高不大,主要是因为任务粒度不均匀,某一层任务的完成时刻常常由最大粒度任务的完成时刻所制约.而在实验 1 中,由于任务较为均匀,所以在一定范围内,加速比随处理节点个数的增多而提高很快.

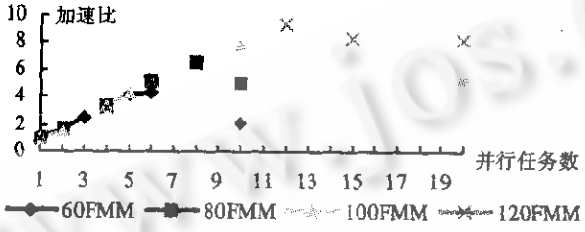

图7　并行任务数与加速比的关系

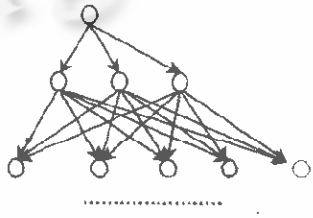

图 8　塔形任务图(前 3 层)

表 7　塔形任务图的调度结果

实验组别	串行执行	并行执行						
		可用 p	实用 p		执行时间		加速比	
			OPBS	SS	OPBS	SS	OPBS	SS
1	4842936	4	4	4	1 992 325	2 466 785	2.43	1.96
	4842936	6	6	6	1 971 023	2 337 802	2.45	2.17
	4842936	8	7	8	1 968 599	2 308 842	2.46	2.09
	4842936	11	7	11	1 967 943	2 084 355	2.46	2.32
2	6124533	4	4	4	2 662 840	3 310 558	2.30	1.85
	6124533	6	6	6	2 617 321	2 958 711	2.34	2.07
	6124533	8	7	8	2 171 820	2 902 622	2.72	2.11
	6124533	11	7	11	2 164 545	2 551 889	2.83	2.39
3	5407265	4	4	4	2 128 844	2 457 847	2.51	2.20
	5407265	6	6	6	2 032 806	2 340 807	2.66	2.31
	5407265	8	7	8	1 945 057	2 207 043	2.78	2.45
	5407265	11	7	11	1 952 081	2 216 092	2.77	2.44

5　结束语

本文提出的基于最佳并行度的任务依赖图调度算法,首先确定并行任务的最佳并行度,通过合并算法达到同层任务粒度的均衡化,通过处理节点的预分配压缩通信开销,实现在尽量压缩调度长度的情况下,尽量节约系统资源的目的.实验表明,在存在大量的并行小任务的情况下,本文的调度方案实现了预定目标.

参考文献

1　Graham R L, Lawler E L, Rinnooy Kan A H G. Optimization and approximation in deterministic sequencing and scheduling: a survey. In: Annals Discrete Mathematics. Amsterdam, Netherlands: North-Holland Publishing Company, 1979. 287~326

2　Wang Q, Cheng K H. List scheduling and parallel tasks. Information Processing Letters, 1991,37(5):78~87

3　Pande S, Agrawal D R, Mauney J. A scalable scheduling scheme for functional parallelism on distributed memory multiprocessor

systems. IEEE Transactions on Parallel and Distributed Systems, 1995,6(4):388~398

4 Polychronopodlos C D. Parallel Programming and Compilers. Boston: Kluwer Academic Publishers Group, 1988. 83~111

5 Liu Y A, Gomez G. Automatic accurate time-bound analysis for high-level languages. Technical Report, TR508, Indiana University, 1998

Optimum Degree of Parallelism-based Task Dependence Graph Scheduling Scheme

DU Jian-cheng HUANG Hao CHEN Dao-xu XIE Li

(*State Key Laboratory for Novel Software Technology Nanjing University Nanjing* 210093)

(*Department of Computer Science and Technology Nanjing University Nanjing* 210093)

Abstract Optimum degree of parallelism-based task dependence graph scheduling scheme fully utilizes the global information collected at compile-time, employs the techniques such as task merging in horizontal and vertical directions, processors pre-allocation, combination of static and dynamic scheduling, and integration of centralized scheduling and layer-scheduling. It is a simple, practical and effective scheduling method which addresses the problem of how to both reduce the execution time of programs and economize on processor resources.

Key words Hierarchical task graph, task dependence graph, static scheduling, dynamic scheduling, optimum degree of parallelism.