计算机科学 1997Vol.24№.2

并行分布计算中的任务调度问题(二)

陈华平 李 京 陈国良
(中国科学技术大学计算机系 合肥 230027)

在并行程序设计中,SPMD是最为常用的一种编程模式,该模式下的任务调度有其自身的一些特点,本文首先讨论了如何用闭式表达式来决定该执行模式下的最佳处理器数。然后,针对任务图存在的不确定性,介绍了采用动态技术进行任务调度的一般方法。由于动态调度会带来额外开销,因此有效的动态调度方案必须结合实际具体的并行程序和硬件系统。为了减少动态调度开销,对于一些含有不确定性的任务图,也可通过任务图归约技术,先把不确定性转换为确定性,然后再利用静态调度方法,本文我们主要讨论了条件分支语句的静态调度问题。另外,并行循环的调度分配是影响并行程序执行性能的一个重要因素,在本文的最后我们结合独立循环、相关循环、均匀循环和非均匀循环的调度分配问题,着重讨论了基于循环分配的自适应混合调度方法,以及变循环传递相关为循环独立相关的循环展开(unrolling)技术。

5 SPMD执行模式下的任务调度问题

当任务图中包含比较有规律的子任务图时,如任务图中包含有在不同数据上执行相同例程的多个子任务,我们可把这些规律性较强、结构比较固定的多个SPMD任务构成的子任务图作为整个任务图的部件,并把这些部件通过分层组织构成复合任务图。这样,复合任务图的结点可能是一个简单任务,也可能是从层次上可以进一步分解的子任务图。对复合任务图的一种调度办法[2]是先对各子任务图进行调度,由于子任务图一般比较有规律,所以能用闭式表达式来决定每个子任务图应使用的最佳处理器数,然后把各分调度结果合并成一个统一的调度方案。

常用的SPMD执行方式有主从式、树形式和网孔式等,下面我们以主从式为例,说明如何获得最佳处理器数的闭式表达式。主从式的主要思想是由一个主处理器(不妨设为P_0)负责接收需要处理的输入数据,然后把输入数据S划分成等量的k份后分别传送给k个从处理器$P_1,P_2,\cdots,P_k$。设这k个从处理器执行具有相同时间复杂度函数F(x)(其中x=S/k)的k个子任务F,每个子任务计算完后分别把各自的结果R送回给主处理器,由主处理器P_0来完成最后的合并操作。例如求一串数的和、平均、最大值、最小值及并行搜索数组等都是这一类的问题。

设通讯函数为$u(x)=a+bx$,其中a为消息传递启动时间,b为传送单位数据量所需要的时间,x为通讯数据量大小。并且假定消息发送者每次只能发送一个消息,而消息接收者可缓冲接收所有传来的消息。我们假定P_0在一个常数时间g内完成划分预处理操作,在常数时间h内执行完合并操作,并且任务F的返回值数据量$R<<S$。P_0每发送一个消息所需时间为$a+bx$,所以收到最后一个消息的处理器P_k必须在$g+k(a+bx)$时才能开始执行,其中其它在此之前接收到消息的处理器已开始并行执行任务F。P_k执行F(x)段时间后,经过$a+bR$时间把结果传送给P_0,P_0再经过h时间完成合并操作。根据上面所讨论的通讯模型可知,第i个子任务的起始执行时刻为$T(i)=g+(a+bS/k)*i$。

设T(S,k)为整个执行时间,那么$T(S,k)=g+k(a+bx)+f(x)+(a+bR)+h=C+ak+f(x)$,其中$C=g+h+a+bR+bS$,$x=S/k$ (1)。下面的问题就是要选择一个介于1和min(S,m)之间的k值,使得T(S,k)取得最小值。这本身就是一个常见的定量并行粒度大小问题,即如何平衡计算时间和通讯时间。如果一个处理器上分配太多的任务,那么由于失去部分并行性而会降低效率;如果分配太少的任务,那么由于通讯开销也会使效率下降[3,4]。

为了求使得T(S,k)值最小的k值,可对(1)式中的k求微商,设使微商等于0的值为k^*。即$a+\partial F/\partial k=0;k^*\in[1..\min(S,m)]$ (2)。我们假设F(x)为多项式复杂度,不妨设$F(x)=x^n=(S/k)^n$,$\partial F/\partial k=-nS^n/k^{n+1}$,由(2)式可得$k^*=\min(\sqrt[n+1]{\frac{nS^n}{a}},m)$ (3)。这个结果的含义也是非常明确的,它说明了:(1)通讯启动开销a对取得最优并行性能有直接影响。在通讯开销较大的计算环境下,所使用的最佳

处理器数目就要少一些，相应的数据并行计算粒度就会大一些；(2)在计算数据量 S 增大的情况下，应选择多一点的处理器来并行执行这些数据；(3)当每个子任务的计算时间复杂度增大时，从并行中也就能获得更大的收益。

对复合任务图最简单的一种调度方案是采用"自顶向下"的方法[2]，即按前面所讨论的时间估计办法先调度上层子任务图，然后依次独立地调度下层子任务图，这种方法实现起来简单，但有时不很有效，因为它忽略了存在于不同子任务图之间的并行性。另一种调度办法是先通过深度优先遍历算法把分层结构的任务图转换成一般任务图，然后采用一些启发信息来进行调度。

6 动态调度

动态调度的基本思想是在并行程序的运行过程中才对任务进行处理器分配，以及决定它们什么时刻开始执行。采用动态调度的主要原因是并行程序所存在的不确定性，一般说来，主要有下列四种不确定因素[8]：(1)并行程序任务中的循环次数事先并不确定；(2)条件分支语句到底执行哪个分支，在程序执行前不能完全了解；(3)每个任务的工作负载大小事先不能确定；(4)任务间的数据通讯量大小只有在运行时才能决定。

虽然动态调度比起静态调度来说更具一般性，但它存在开销大的缺点，这主要包括动态调度所引起的额外通讯延迟、额外内存及动态执行调度算法所需的时间。另外，动态调度有可能导致任务"抖动"，即一个任务可能在处理器之间多次调进调出，消耗更多的时间。这些调度开销本身与并行计算机的硬件及操作系统有关，因此，动态调度的效率除了取决于调度算法本身的好坏以外，还与具体的并行程序和并行计算机的硬件系统有关。下面我们讨论几种常用的动态调度技术。

6.1 无约束 FIFO 调度

最为基本的一种动态调度方法是假定事先对并行程序任务一点都不了解，因此，就使用非常局部的一些信息来平衡各处理器的负载。在这种情形下，可将一个处理器(称为调度处理器)专门用于调度(不妨设为 P_m)，其余每个处理器 $P_i(1\leqslant i\leqslant m-1)$有一个局部的任务等待队列 WQ_i，WQ_i 包含了所有分配到处理器 P_i 上的待执行任务。一个任务在执行时，有可能请求其它任务的服务，这些请求被传送到 P_m 上的调度队列中。调度处理器也按先来先服务的原则给其余处理器分发任务，按什么原则选择执行分发任务的处理器是这类动态调度算法的关键所在。常用的原则是按照等待队列中任务数目，或者按照等待队列中所有任务的总工作负载大小来选择处理器，当很难获得任务工作负载的精确估计时，一般采用第一种选择原则。

无约束 FIFO 调度的最大优点是实现简单，比较适合于通讯延迟开销较低的共享存储并行计算模型，但它通常与最优调度相差甚远。

6.2 平衡约束调度

V. Saletore 在无约束 FIFO 调度思想基础上，提出了平衡约束调度算法，其主要思想是通过等待队列中的任务移动来重新平衡各处理器的工作负载。但是，在分布存储的大规模并行处理系统中，这可能会引起非常大的开销，常用的一个折衷方法是在时间和范围上加以一些限制，即定期进行局部负载平衡调度。每个处理器根据系统的拓扑连接结构，计算它相邻处理器的平均负载，如果该处理器的负载大于平均负载，那么就把部分任务迁移到负载较轻的相邻处理器上。由于每个处理器同时执行近邻平均操作，因此，这些操作会把负载调整传播到整个系统中。

上面动态调度算法的主要优点在于调度是分散的，所以很容易扩放到具有大量处理器的并行系统中，并且定期调整也会减少调度开销。另外，该算法初始时尽快地把所有任务分散到各处理器上，虽然每个任务的执行时间事先无法预测，但调度程序马上会自适应地平衡调整各处理器的负载。

6.3 成本约束调度

成本约束调度的主要思想是不仅考虑处理器之间的负载平衡，而且还针对不同的具体并行系统，进一步根据某些成本指标进行任务调度。例如，在基于网络环境的并行分布系统中，通讯延迟开销较大，虽然通过负载平衡调度能获得一些收益，但有可能这些收益还远抵不上由于任务迁移所带来的通讯开销。因此，成本约束调度算法首先根据平衡约束条件确定需要迁移的候选任务，但至于该候选任务到底迁移与否，还要进一步通过检查任务迁移所引起的某些成本开销来确定。除了通讯成本指标外，还有内存开销及磁盘操作成本等指标，有时为了使某项成本最低，甚至不惜以负载不平衡为代价。例如，有时我们不管每个处理器的工作负载如何，而尽量要求每个处理器的 I/O 操作数量比较平均。

7 条件分支的静态调度

虽然动态调度比起静态调度来说更具一般性，但它存在额外开销大，并且有可能导致任务"抖动"的缺点。因此，在并行分布计算中，经常利用任务图归约技术[4][5]，把任务图中诸如条件分支语句这类不确定性问题逐步转化为可用静态调度方法来解决的确定性问题。

一个并行程序可看作是一组任务以及这些任务之间的控制流和数据流，对于包含条件分支的任务图，我们也可用两个无环有向图来表示并行程序的行为，一个是分支图 $BG=(V,E_b)$，另一个是前趋图 $HG=(V,E_p)$，其中 V 是代表任务的顶点集合，E_b 是分支图的边，E_p 是前趋图的边。形成不同条件分支的任务有许多共性，这些共性包括各分支与程序中其它任务的通讯方式。另外，不同分支的计算量可能是差不多的，如果我们能找出具有某一共同特性的任务集，那么可用一个任务来代替这个集合，从而简化任务图。最佳的话，可以完全消除与条件分支有关的不确定性。

7.1 任务图归约

如果任务结点 u 和 v 在分支图 BG 中满足条件：$IMS(u)=IMS(v)$，$IMP(u)=IMP(v)$，那么称结点 u 和 v 是相似的。条件分支语句的各分支结点之间存在下面三种类型的差别：(1)前趋图中分支结点的后继集和前趋集；(2)前趋图中分支结点与后继结点或前趋结点之间的通讯成本；(3)分支结点的执行计算量。针对这三种类型的差别，我们可引入容差参数 α、β 和 γ，如果一组相似的分支结点之间的差别范围在这三个容差参数之内，就可把这些结点作为等同结点对待。

定义 1　给定容差参数 α、β 和 γ，如果相似结点 u 和 v 在前趋图中满足下列条件：

(1)$\max(|IMP(u)-IMP(v)|,|IMP(v)-IMP(u)|,|IMS(u)-IMS(v)|,|IMS(v)-IMS(u)|)\leqslant\alpha$。

(2)$\forall x((x,u)\in E_p,(x,v)\in E_p)$，那么 $abs(D(x,u)-D(x,v))\leqslant\beta$；

$\forall y((u,y)\in E_p,(v,y)\in E_p)$，那么 $abs(D(u,y)-D(v,y))\leqslant\beta$。

(3)$abs(W_u-W_v)\leqslant\gamma$。

那么，称任务结点 u 和 v 是等同的。

容差参数规定了作为等同结点看待的相似结点之间的差量上界，决定了相似结点的等同关系程度，在分支图和前趋图中用单一结点来替换每个等同结点集，所得到的归约分支图和归约前趋图表示了相当小的归约程序模型。

定义 2　一个执行实例是一个任务子集 EI，该任务子集被选来执行某些输入，并且对每个相似结点集来说，该任务子集最多只能包含其中的一个结点。即如果 u，v 不是相似结点，那么至少存在一个包含 u 和 v 执行实例。

根据上面定义，给定一个分支图 $G=(V,E_b)$，$u,v\in V$，如果 u 与 v 相似，那么不存在同时包含 u，v 的执行实例。设 $SCHED(EI_u)$ 是对于包含任务 u 的一个执行实例 EI_u 的最优调度，如果 u 与 v 等同，那么可通过用任务 v 替换 EI_u 中的 u，来获得包含 v 的一个执行实例 EI_v 的最优调度 $SCHED(EI_v)$[5]。对分支图中的每一等同结点集，可选择等同集中的任一结点，移去所有其它与之等同的结点，这样把每一等同结点集归约成一个结点。

7.2 MPA 算法

MPA(Multi-Phase Approach)算法[6][7]的主要思想是通过下列四个步骤，在归约任务图的基础上获得原始不确定任务图的静态调度方案。

(1)产生覆盖所有任务的执行实例集。这一步首先估算每个任务的概率，然后产生覆盖所有任务的执行实例集 SetEI。概率为 1 的任务包含在所有执行实例中，优先考虑具有最高概率的实例。当执行实例数目较少时，有可能考虑所有的执行实例，但一般情况下，产生所有执行实例是不可行的。其实，也根本不需要产生所有实例，只要产生的执行实例集 SetEI 满足每个任务至少在这些实例中出现一次这个条件即可。

(2)构造每个实例任务图。这一步主要为上面产生的每个执行实例构造实例任务图。对产生的每个实例 EI，实例任务图 $ITG=(EI,E_{ie})$ 是前趋图 HG 的子图，ITG 给出了该实例的任务以及它们之间的先后关系，其中 $EI\subseteq V$，$E_{ie}\subseteq E_p$。如果 $u,v\in EI$ 并且 $(u,v)\in E_p$，那么 $(u,v)\in E_{ie}$。ITG 中任务结点的工作负载及任务间的通讯量保持与 HG 中的值相同。

(3)调度每个实例任务图。对(2)中产生的每个确定的实例任务图，可独立地采用某一静态调度算法来对它调度，产生以 Gantt 图形式表示的调度方案。每个实例 EI_g 的 Gantt 图 GC_g 用两个函数 $P_g(u)$ 和 $T_g(u)$ 来表示，其中 $u\in EI_g$。如果 $P_g(u)=-1$，那么表示 u 不属于相应的执行实例 EI_g。与每个 Gantt 图有关的量还有 prob(g)，它表示 EI_g 出现的概率，其计算方法为 $prob(g)=\Pi P(u,v)$，其中(u，

$v)\in E_b$ 并且 $u,v\in GC_g$，$P(u,v)$为分支图中执行(u,v)条件分支的概率。

(4)合并各分调度为统一的调度方案。这一步的主要功能是把(3)中产生的 Gantt 图合并成一个统一的调度方案，合并过程主要有两步，第一步是进行处理器分配，可用概率相加，并以一个任务在同一处理器上的出现次数来决定任务分配；第二步是计算任务的执行时序，即每个任务 u 在相应的处理器上的执行时序。设 u，v 是两个任务，如果在第一步中 u 和 v 被分配到同一处理器上，对于这两个任务的执行次序，可通过检查(3)中这两个任务在同一处理器上的所有 Gantt 图来获得。如果两个任务在不同的 Gantt 图上执行次序不同，那么可通过一个权重函数来决定它们在统一的调度中的执行次序，该权重函数为 Gantt 图出现的概率之和。如果在所有的 Gantt 图中，两个结点均未出现在同一处理器上，并且它们之间是存在先后关系的，那么它们在统一调度中的关系由原来的前趋图所决定。

8 并行循环的调度

在并行程序设计中，如何高效地并行执行循环语句是最为常见的一个问题。一般说来，如果并行循环语句的执行结果与每次循环迭代的执行次序无关，那么称该并行循环是独立的，否则就是相关的。如果并行循环语句的每次循环迭代的执行时间相同，那么称该并行循环是均匀的，否则就是不均匀的。均匀的独立并行循环调度比较简单，只要采用静态的循环分配技术就可平衡各处理器的负载。下面我们主要讨论不均匀循环及相关循环的调度方法。

8.1 不均匀循环的自适应混合调度

对于不均匀的独立并行循环，每次循环迭代所需执行时间不同，并且每次迭代的执行时间也只有在运行时才知道，那么单采用静态调度技术就很难在编译时把整个工作负载均衡地分配给各处理器，所以应该说采用动态调度方案更灵活些[3]。但动态调度必须以延迟、额外内存及执行调度所需时间等额外开销为代价。有一种减少动态调度开销的办法，那就是采用混合调度[9]，其主要思想是在不破坏各处理器负载平衡的前提下，先在编译时静态分配尽可能多的任务给各处理器，剩余负载则采用动态调度的办法进行分配；在动态调度过程中，并且根据当前执行动态调度算法的开销，自适应地调整每个循环块的最小迭代次数。

设循环迭代总次数为 N，$E(i)$表示执行第 i 次循环迭代的时间，也叫第 i 次循环的工作负载，那么整个循环执行时间为 $E=\sum_{i=1}^{N}E(i)$，分别用 E_{max}、E_{avg}和 E_{min}表示每次循环迭代所需的最大、平均和最小执行时间。我们把分配给处理器的一组连续的循环迭代称作(任务)块，用 B_j 表示第 j 个块，块中所包含的循环迭代次数称为该块的长度。如果 T 是产生的整个块数，I 是循环迭代空间，显然有 $I=\bigcup_{j=1}^{T}B_j$。自适应混合调度算法[9][8]首先把 B_1 分配给处理器 P_1，B_2 分配给处理器 P_2，…，以此类推，每 m 个连续块构成一个回合。最早的头 m 个块是编译时静态分配的，构成第 0 回合。剩下的块采用动态调度的办法分配，每进入下一回合就对剩余的循环迭代分块，块的大小根据当前循环迭代空间的大小自动调整，最后一回合把剩余的所有循环迭代分配完。由于分配块长度是按指数下降的[9]，所以回合数一般不会很多。

定义 3　设 $E(P_i)$为处理器 P_i 的工作负载，如果某一调度使得每个回合满足条件 $MAX\{(E(P_i)-E(P_j)|i,j=1,2,\cdots,m\}\leqslant E_{max}$，那么称该调度为平衡调度。

由于调度程序放在互斥区中，各处理器必须等待进入互斥区才能执行调度程序。若减少分块总数 T，那么进入互斥区的次数减少，也就减少了处理器因进入互斥区而花费的等待时间，从而降低整个调度开销。从这一点上看，必须增大每个块的长度以减少块总数 T，但这样各处理器之间负载不平衡的机会就大大增加。所以关键在于每个回合中如何选择合适的块长度。

可以按照式子 $\sum_{i=s}^{s+q-1}E(i)<\frac{E}{m}\leqslant\sum_{i=s}^{s+q}E(i)$来确定 T 的大小，即给空闲处理器分配 q 个连续的循环迭代，其中 $s(\in I)$为某一块 $B_j(1\leqslant j\leqslant T)$的循环开始点，E 为整个并行循环的负载大小，E/m 相当于每个处理器的平均负载。式(2)的后一部分确保给每个空闲处理器分配足够的负载，以免它再次取任务；前一部分确保不给空闲处理器分配太大的工作量，以免引起各处理器之间的负载不平衡。根据上式，给处理器分配长度介于 1 和 $\frac{E/m}{E_{max}}$ 的一个块总是安全的[9][1]，但由于这个块长度小于平均负载，所以至少还得再分配一次。从 $\frac{E/m}{E_{max}}$ 开始，分配的块越大，超过平均负载的机会就越多，再次分配的可能性就越小，但破坏负载平衡的机会也越大，有许多在这两者之间进行折衷的办法[9]。

设每次动态调度开销为 O_i，如果每次循环迭代

的平均执行时间 E_{avg} 小于 O_j（特别是 $E_{avg} \ll O_j$）时，很明显最后几个回合的块长度不能简单地按平衡条件来计算，否则的话，各处理器负载平衡所获得的收益远抵不上由于各处理器的动态调度所带来的额外开销。可选择 $L_{min}=[O_j/E_{avg}]$ 作为每个块的最小迭代次数，至于 E_{avg} 可在静态块执行完后求得，O_j 可在第一次动态调度后求得。

8.2 相关循环的调度

一般说来，循环体中每个任务之间的数据相关性可分为两大类：循环独立（loop-independent）相关和循环传递（loop-carried）相关。如果数据相关只限于本次循环迭代，而与其它循环迭代的数据无关，那么称这种数据相关性是循环独立的，显然，具有循环独立相关性的循环语句的执行结果与循环迭代的执行次序无关。对于循环独立相关的循环语句可用一般任务图来表示，然后使用前面介绍的调度技术进行调度。

由不同循环迭代之间的数据传递引起的任务之间的相关称为循环传递相关，很难用一般任务图来表示，它比循环独立相关更复杂。一般地，可用循环任务图来表示循环体中各任务之间的循环传递相关和循环独立相关这两种数据相关性，但是，直接调度体现相关循环的循环任务图比较困难。经常采用的办法是先通过循环展开技术[8]，把循环任务图转换成一般任务图形式的复制任务图，然后再利用一般任务图的调度算法来调度该复制任务图，这里如何选择循环展开向量是问题的关键[9]。

8.2.1 循环任务图 用循环向量 $I=(i_1,i_2,\cdots,i_k)$ 来表示包含在 k 重循环中的一组任务的一个执行实例，它代表了一组循环控制变量的值，循环向量中的元素按外层循环到内层循环的次序排列，第 j 个元素表示第 j 层循环正处于第 i_j 次迭代。根据循环向量，我们可定义两个任务之间每个相关关系的距离向量。

定义 4 设 q 和 r 是包含在 k 重循环中的两个任务，如果迭代 I_r 中任务 r 的执行依赖于迭代 I_q 中的任务 q，那么这个相关关系的距离向量为 $D=I_r-I_q=(d_1,d_2,\cdots,d_k)$。

定义 5 设 q 和 r 是包含在 k 重循环中的两个任务，用(D,C)表示迭代 I_r 中的任务 r 与迭代 I_q 中的任务 q 之间的相关对 DP(q,r)(dependency pair)，其中 D 为距离向量，C 表示任务 r 与任务 q 之间的通讯量。

显然，两个任务循环传递相关的充要条件为 $D \neq 0$。任务 r 可能与任务 q 不止一个相关对，我们把它们之间的所有相关对构成的集合称为这两个任务之间的相关集 DPS(q,r)。这里要指出的是上面两个定义不排除 q 与 r 相同，q=r 反映了同一个任务在不同循环迭代之间的数据传递相关关系。

循环任务图是一个带权有向图 $LG=(V,LE)$，其中 V 是结点集，LE 是边集，其中 V 的定义同一般任务图相同，但 LE 中的边权是任务 q 与任务 r 之间的相关集 DPS(q,r)，它不但体现了两个任务间的通讯量和一般的循环独立相关性，还体现了它们之间的循环传递相关性。在循环任务图中，同一任务的循环传递相关是一个环。

8.2.2 循环展开 是把一个循环的某些迭代过程替换成非迭代的直接代码，它的主要目的就是展示循环传递相关，以便能使几个循环迭代重叠执行。可用 $U=(u_1,u_2,\cdots,u_k)$ 表示循环展开向量，用 $B=(b_1,b_2,\cdots,b_n)$ 表示循环的上界向量。如果一个上界为 b 的单循环被展开 u 次，那么原循环体就复制成 u+1 份，每次复制时还需相应调整循环控制变量，循环步长也乘上 u+1。当一个具有上界向量 $(b_1,b_2,\cdots,b_k)$ 的 k 重嵌套循环用展开向量 $U=(u_1,u_2,\cdots,u_k)$ 来展开时，那么有 $\prod_{i=1}^{k}(u_i+1)$ 份循环体被复制，其中第 i 层循环的步长值乘上 u_i+1。

一般地，我们可按照下列方式定义复制任务图 $G_u=(V_u,E_u)$：G_u 代表用展开向量 U 展开循环体后的无环任务图，G_u 的结点集 V_u 是按照上面所述的展开过程产生的，它的结点总数 $|V_u|$ 为 $|V| * \prod_{i=1}^{k}(u_i+1)$，其中每个产生结点的权与源结点的权相同。$G_u$ 的边集 E_u 代表了原来 LG 中的循环独立相关和循环传递相关，其中后者经过循环展开后在 G_u 中也变成了循环独立相关，G_u 中的边权为原来循环任务图中边权的通讯量部分。

8.2.3 调度循环任务图 由于展开循环传递相关就会允许部分循环迭代在执行时间上重叠，所以循环展开可以开发存在于不同迭代之间的并行性，其基本步骤为：(1)采用某展开向量 U 展开循环，以便在复制任务图 G_u 中展示迭代间的相关性；(2)采用一般任务图的调度方法把 G_u 调度到某目标机器上；(3)对于没展开的循环迭代，按照(2)产生的调度方案逐个调度。那么到底展开哪些循环，展开多少次才能获得最佳的性能提高？一种直接了当的办法是完全展开所有循环，但当上界向量中元素较大时这不太可行。实际上有时只要展开部分循环就可

开放分布式处理中的交易服务*)

龚 俭
(东南大学计算机科学与工程系 南京 210096)

摘 要 Trading is the fundamental function of open distributed processing systems to compose an open cooperative system. It is a kind of improvement of network directory service and binder service in RPC, and has more comprehensive formal semantic description capability. The basic functions of trading service is introduced in the paper, and its working mechanism and required operations are discussed as well.

关键词 ODP, Trading, Trader.

大规模互连网络的出现及其与并行工程、远程教育等新型分布式应用的结合，已开始迅速地影响现实的生产活动和社会活动。这种结合促使分布式应用的设计与实现从全局模型方式转向集成方式，即分布式应用的处理元素以标准的形式进行交互合作，并支持一致的语义，这就是开放分布式处理的含义。开放分布式处理(ODP)的基本参考模型为开放的分布式应用提供了一个体系结构框架，以支持系

获得较大的性能提高，所以关键在于如何选择最优的循环展开向量。

8.2.4 如何确定最佳循环展开向量 确定一个最优展开向量 $U=(u_1,u_2,\cdots,u_k)$ 这本身是一个组合优化问题，虽然问题空间是 k 维离散的，但穷尽搜索是不可能的，所以必须利用一些优化启发信息。不管使用哪一种优化搜索方法，关键在于选择目标函数，以及如何选定 U 的初始值。

如果把 $D_{max}=(g_1,g_2,\cdots,g_n)$ 定义为一个循环任务图的最大距离向量，其中 $g_i=\max\{d_i(d_i$ 为 DP 中 D 的第 i 个分量)}，那么 D_{max} 也是展示所有隐藏的循环传递相关的最小展开向量，所以可把 D_{max} 作为最优搜索的循环展开向量初始值 U_0。

设 x_i 为执行第 i 层循环迭代一次所需时间，那么 x_n 为执行一次 G_n 的时间，x_1 为执行一次最外层循环所需时间。设 y_i 为第 i 层循环的两个相继迭代之间的通讯延迟。Rewini[10][11]给出了目标函数 $F=\left(\prod_{i=1}^{n}\frac{1}{u_i+1}\right)\left(x_k+\sum_{i=1}^{n}y_i\right)$。至于参数 x_k 和 y_i，把任务图 G_n 调度到某一目标机器上时，x_k 就是所获调度结果的调度长度。另外，根据调度所产生的 Gantt 图，我们可得到第 i 层循环的相继迭代之间的通讯延迟 $y_i(1\leqslant i\leqslant k)$。

参考文献

[1] 陈华平、李京、陈国良，并行分布计算中的任务调度问题(一)，《计算机科学》，24(1)1997
[2] 陈华平、陈国良，SMPD 模式下的任务调度问题，《计算机科学》，23(6)1996
[3] D. Towsley, Allocating programs containing branches and loops within a multiple processor system, IEEE Trans. on Soft. Eng. SE-12(10) 1986
[4] H. El-Rewini and H. Ali, Scheduling conditional branching using representative task graphs, Journal of Combinatorial Mathematics and Combinatorial Computing. 10, 1991
[5] H. El-Rewini and H. Ali, Scheduling parallel programs containing conditional branching, Journal of Parallel and Distributed Computing, 1994
[6] V. Saletore et al., Scheduling non-uniform parallel loops on distributed memory machines, Proc. HICSS-26, Maui, Hawau, 1993
[7] H. El-Rewini and H. Ali, How many times should a loop be unrolled?, Technical Report, Dept. of Computer Science, University of Nebraska at Omaha, 1993
[8] H. El-Rewini and T. Lewis, Schedule-driven loop unrolling for parallel processors, Proc. the 24th Hawaii Intl. Conf. on System Science, Kauai, Hawaii, January, 1991

---

*)本文的研究内容受江苏省青年科技基金课题(BQ94002)资助。