一种适用于机群系统的任务动态调度方法*

傅 强 郑纬民

(清华大学计算机科学与技术系 北京 100084)

摘要 任务调度是机群系统上实现并行计算需要解决的重要问题之一.对于在运行中动态产生任务的并行应用程序,由于很难作出准确的任务分配决策,可能导致各个计算结点的任务负载失衡,最终引起整个系统的性能显著下降.因此,需要通过任务再分配来维持负载平衡.该文提出一种任务分配与再分配方法,它通过尽量延迟任务的执行开始时刻,在任务再分配时避免了进程迁移,使得引入的调度开销很小.分析和实验结果表明,该方法在许多情况下能够有效地提高并行程序的运行性能.

关键词 调度,机群系统,并行计算,任务分配,负载平衡.

中图法分类号 TP316

近年来,随着工作站性能价格比的迅速提高和快速局域网的出现,工作站机群系统(Cluster of Workstations)已成为并行计算的一个重要发展方向.在机群系统上进行并行计算,需要软件环境的支持.目前已经开发并广泛应用的并行软件环境有PVM(parallel virtual machine)[1],MPICH(message-passing interface)等.

在基于消息传递模式的机群系统并行计算中,并行任务的调度方案极大地影响了应用程序的运行性能.在实际应用领域中,由于机群系统结构和程序本身的不确定性,往往需要在运行时进行动态调度.但动态调度无法获得任务的负载粒度和相互间的同步与通信关系的准确信息.因此,作出的任务分配往往不够准确,还需要在适当的时候进行任务再分配来维持各个结点上的负载平衡.在一些并行系统如PVM中,任务一经分配就立即执行,要实现任务再分配,只能采取进程迁移的方式.但进程迁移不仅难于在异构结点间实现,而且其开销也十分庞大,甚至抵消负载平衡所带来的性能提高.

本文提出了一种基于消息传递并行环境的"惰性"任务调度方法,它可以不借助进程迁移而实现低开销的任务再分配.其特点有:(1) 采用非阻塞式任务派生方式,缩短响应时间.(2) 采用惰性任务执行机制,只对未开始执行的任务进行再分配,开销很小.(3) 任务分配及再分配时采取本地化算法,不仅进一步减小调度开销,而且在一定情况下还减小了通信开销.(4) 任何任务至多执行一次再分配,不会产生"颠簸"现象.

下面,首先给出算法描述,然后介绍该算法的实现方案,最后给出性能分析和模拟实验的结果.

1 算法描述

"惰性"任务调度方法由任务分配算法和任务执行与再分配机制两部分组成,如图1所示.其中,任务分配算法响应应用程序发出的任务派生申请,作出首次任务分配决定;任务执行机制决定在何时运行何任务,并在一定条件下对已分配的任务进行调整,即任务再分配.

1.1 任务分配算法

实现任务的低开销再分配,实际上是通过"迁移"尚未执行的任务来实现的.这就要求任务在分配后不立即执行,而是进入本地的就绪队列.系统按LIFO(last in first out)顺序从队头取任务执行,只有当该任务结束或因等待未执行任务传送数据而阻塞时才取下一个任务执行.由于单CPU的UNIX系统采用时间片轮换法调度用

* 本文研究得到国防科技预研基金资助.作者**傅强**,1970年生,博士生,主要研究领域为并行处理,负载平衡.**郑纬民**,1946年生,教授,博士生导师,主要研究领域为并行/分布处理.

本文通讯联系人:傅强,北京 100084,清华大学计算机科学与技术系应用教研组

本文1997-11-05收到原稿,1998-01-23收到修改稿

户进程,故这种多个任务串行执行的总时间与并发执行相同.

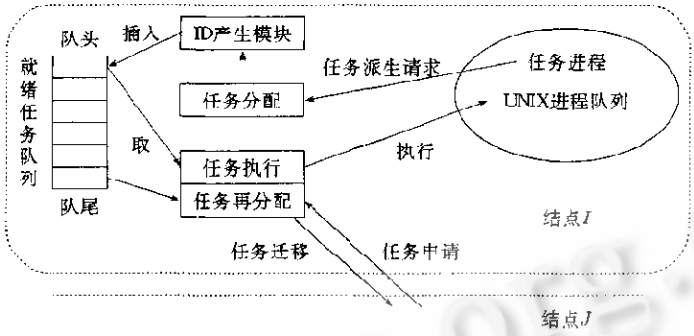

图1

本文采用的任务分配算法流程如下:

```
If (Receive(SPAWN,taskname)) {    // 接到派生任务 taskname 的请求
  ID=GetID(taskname);             // 创建对应 taskname 的唯一标识号
  Insert(Q,ID,HEAD);              // 将 ID 插入本地就绪任务队列 Q 的队头
  Ack(ID);                        // 应答派生任务的请求并返回 ID
}
```

首先将任务分配在本地,目的是减少调度开销,并且使通信尽量本地化.任务分配后不立即执行,但马上响应派生任务的请求,使发出请求的进程能够继续执行.

1.2 任务执行与再分配机制

任务执行机制只有在当前运行的任务进程结束或因等待未执行任务传输数据而阻塞时才被激活.它首先查看本地就绪任务队列 Q,若 Q 非空,则取队头任务执行;若 Q 为空,则进行任务再分配:向其他结点申请任务.被申请结点若 Q 非空,则返回队尾任务.

```
If (Endtask(currentTask) or Block(currentTask)) {   // 没有任务进程在运行
  If (not empty(Q))
    currentTask=Get(Q,HEAD);                        // 取队头任务 ID 作为当前运行任务号
  else
    currentTask=RequestTask(otherNode);             // 向其他结点申请任务
  Run(currentTask);                                 // 执行该任务
}
If (Receive(REQUESTTASK))                           // 其他结点申请任务
  If (not empty(Q))
    Ack(Get(Q,TAIL));                               // 返回队尾任务
```

这里,执行机制在运行本地任务和申请其他结点任务时分别采用的是 LIFO 策略和 FIFO(first in first out)策略.任务派生关系可以看成一棵任务树.由于本地执行任务采用 LIFO 方式,可以维持较小的就绪任务集.而"迁移"任务采用 FIFO 方式,则迁移走的任务在任务树上更靠近根结点,也就是说它会派生出更多的任务,从而保护通信的本地化.Blumofe 的分析[2]表明,对于大规模动态并行计算,此种方式能够获得接近线性的加速比.

当结点的本地就绪任务队列为空(结点轻载)时,该结点向其他结点申请任务.这种方式也就是 Receiver-Initiated 方式(RI-调度).由于轻载结点承担了发起任务再分配的开销,因此,RI 方法在系统平均负载较重的情况下能够有效地减少调度开销.[3]

申请其他结点任务的过程就是任务再分配的过程.由于申请后立即执行该任务,就保证了一个任务最多被再分配一次.

2 实现方案

通过考察 PVM 的 API 可以看出,本文所述算法可以在 PVM 系统上加以实现.概略地说,就是接管 PVM 的 3 组原语 PVM_SPAWN,PVM_SEND 和 PVM_RECEIVE.如图 2 所示.

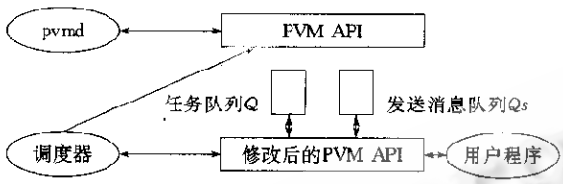

图2

系统引入新的数据结构:本地派生任务队列 Q 和发送消息缓冲队列 Q_S.

对于接管后的 3 组原语描述如下:

• PVM_SPAWN

系统向应用程序返回一个任务标识号 ID,并将其插入队列 Q;

• PVM_SEND

查看队列 Q;

if (目标任务已激活)

　调用 PVM 的消息发送原语进行发送;

else if (Q_S 已满) {

　　挂起;

　　执行本地调度;

}

else

　　插入队列 Q_S;

• PVM_RECEIVE

查看相关结点上的队列 Q;

if (目标任务已激活)

　调用 PVM 的消息接收原语进行接收;

else {

　　挂起;

　　执行本地调度;

}

3 性能分析

任务调度的目的是要达到负载平衡,从而缩短并行程序的运行时间,进而获得更高的加速比.下面,我们通过对比 PVM 系统来讨论本文任务再分配方法的性能.

当不考虑任务间的通信时,设应用程序包含 n 个独立任务,其粒度分别为 $g_i, i \in \{1,\ldots,n\}$.并行系统包括 m 个单 CPU 工作站结点,其计算能力分别为 $c_j, j \in \{1,\ldots,m\}$.

结点 j 单独执行任务 i 的响应时间为 $t_j(g_i)=g_i/c_j$.

PVM 采用 Round-Robin 转轮法一次性分配任务,根据抽屉原理,分配到每个结点上的任务数 $N_j (j \in \{1,\ldots,m\})$ 在 $\left[\left\lfloor \frac{n}{m} \right\rfloor, \left\lceil \frac{n}{m} \right\rceil\right]$ 之间.由于任意两个结点分配到的任务集合的总粒度可能相差很大,导致运行时间

相差很多，不能保证负载平衡. 设结点 j 运行分配到的 N_j 个任务的总时间为 T_j，则并行加速比 S 就是在最快的结点上运行整个应用程序所需时间与并行执行时最晚结束的结点运行时间的比值：

$$S=\frac{\min\limits_{j\in\{1,\cdots,m\}}\left\{\sum\limits_{i\in\{1,\cdots,n\}}t_j(g_i)\right\}}{\max\limits_{j\in\{1,\cdots,m\}}(T_j)},$$

其中分子部分是一个常数，设为 C. 在最坏的情况下，即并行计算中分配在最慢结点上的任务粒度远远大于其他任务时，可得 $S\approx\frac{c_{slow}}{l\cdot c_{fast}}$，其中 c_{fast} 和 c_{slow} 分别为最快和最慢结点的计算能力，l 为分配在最慢结点上的任务粒度与应用程序总粒度之比. 可以看出，当机群系统各结点计算能力相差较大且任务粒度不均时，PVM 可能获得远远小于 1 的最差加速比，也就是说，并行时的性能不如在最快的一个结点上运行的性能.

下面看一下采用本文中算法时的情况. 首先可以证明，最早结束的结点 a 的运行时间 T_a 与最晚结束的结点 b 的运行时间 T_b 之差 $\Delta\leqslant t_{slow}(g_{max})$，其中 g_{max} 是应用程序中最大的任务的粒度，$t_{slow}(g_{max})$ 是最慢的结点运行 g_{max} 的时间.

证明：用反证法. 假设 $\Delta>t_{slow}(g_{max})$，则当结点 a 运行结束时，结点 b 正在运行的不可能是最后一个任务（也就是说，队列 Q 中还有未开始运行的任务），这是因为任何任务 x 的执行时间 $t_b(g_x)$ 小于等于 $t_{slow}(g_{max})$. 根据算法，这时，结点 a 将向结点 b 申请运行任务. 这就意味着结点 a 并未运行结束，与前提矛盾. □

上面的结论保证了各结点的运行时间不会相差太大，从而实现了负载平衡.

另外，从加速比来看，由于最快的结点上运行的任务集是总任务集的子集，故运行时间不会超过常数 C. 结合上面结论可得：

$$S>\frac{C}{C+t_{slow}(g_{max})}\approx 1,$$

也就是说，在最坏的情况下，并行运算的时间与在最快的一个结点上串行运行相当.

在模拟实验中，我们将本文中提出的调度方法与 PVM 的任务分配策略进行了性能上的比较. 实验中模拟了以下环境：

(1) 4 个结点的机群系统，各结点的计算能力的比值为 4:2:2:1；

(2) 任务序列为随机产生的树形序列，参数 k 调整任务派生的时间间隔，k 越大，表示任务派生的间隔越长，$k=1$ 表示任务派生的平均时间间隔与任务的平均计算粒度相等；

(3) 任意两个任务间有可能存在通信或同步关系，参数 c 用来调整任务间的通信依赖关系，c 越大，任务间的通信依赖性越强，$c=1$ 表示任意两个任务间都存在通信关系，$c=0$ 反之.

图 3 为模拟实验的测试结果，图中每一个数据点代表一次对比模拟；纵坐标表示使用 PVM 调度策略的总运行时间与采用本文调度方法的总运行时间之比. 为简单起见，实验忽略了网络通信的时延及调度系统的开销.

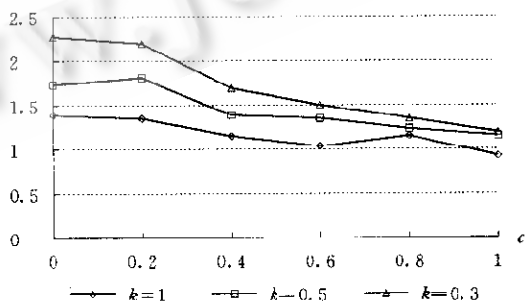

图3

从图中可以看到，使用本文的调度方法可以提高程序的并行性能. 另外，还可以看到，该方法对任务间通信关系较简单、任务分配频度较高的应用程序，效果较为显著.

4 小　结

针对机群系统并行计算中如何实现负载平衡的问题，本文提出了一个低开销的任务调度方法.该方法采用"惰性"任务执行机制，在不影响并行性的前提下尽量延迟就绪任务的实际执行，以便在并行计算过程中对未执行的任务进行重新分配.分析和实验结果表明，该方法在一定的条件下可以达到较好的性能.此外，该方法可利用 PVM 系统加以实现.

参考文献

1 Geist A, Beguelin A, Dongarra *et al*. PVM: Parallel Virtual Machine. Cambridge: The MIT Press, .994. 1～19

2 Blumofe R D, Park D S. Scheduling large-scale parallel computations on networks of workstations. Technical Report, Massachusetts Institute of Technology, 1994. 1～4

3 Willebeek-Lemair H, Reeves A P. Strategies for dynamic load balancing on highly parallel computers. IEEE Transactions on Parallel and Distributed Systems, 1993,4(9):979～993

A Dynamic Task Scheduling Method in Cluster of Workstations

FU Qiang　ZHENG Wei-min

(*Department of Computer Science and Technology, Tsinghua University, Beijing* 100084)

Abstract　Task scheduling is an important issue in the research of parallel computing in cluster of workstations. Because it is difficult to make precise decision of task allocation when running parallel applications that dynamically spawn tasks, load imbalance maybe occur and the performance of whole system will decrease dramatically. So task reallocation is necessary for load balancing. A method for task allocation and reallocation is presented in this paper. By deferring the task's real start-time, it avoids process migration in task reallocation. Therefore, the overhead is greatly decreased. Analysis and experiments show that this method can effectively improve the performance of parallel applications in many cases.

Key words　Scheduling, cluster of workstations, parallel computing, task allocation, load balancing.