第 33 卷 第 7 期 计 算 机 工 程 2007 年 4 月
Vol.33 No.7 Computer Engineering April 2007
·软件技术与数据库· 文章编号：1000—3428(2007)07—0073—02 文献标识码：A 中图分类号：TP393

基于 ProActive 的并行计算任务调度器的研究

董明刚[1,2]，梁正友[1]
（1. 广西大学计算机与电子信息学院，南宁 530004；2. 桂林工学院电子与计算机系，桂林 541004）

摘 要：ProActive 在开发并行计算及网格应用中很受欢迎，但目前还没有可用于基于 ProActive 的并行计算的任务调度器，在对相关技术进行研究后，给出了一个简单有效的任务调度器的实现方案，介绍了其技术细节。将其应用于开发的基于 ProActive 的并行计算支持平台中进行了实验，实验结果表明该调度器是可行和有效的。

关键词：活动对象；调度器；负载平衡；ProActive

Research on Scheduler for ProActive-based Parallel Computing

DONG Minggang [1,2], LIANG Zhengyou [1]
(1. College of Computer and Electronic Information, Guangxi University, Nanning 530004;
2. Department of Electronic and Computer, Guilin University of Technology, Guilin 541004)

【Abstract】ProActive is popular in parallel computing and grid programming, however there isn't a suit scheduler for ProActive-based parallel computing. This paper proposes a scheduler, and all technologies are introduced. Realized scheduler is used in a ProActive-based platform for parallel computing. Experiment demonstrates it is feasible and valid.

【Key words】Active object; Scheduler; Load balance; ProActive

为了克服现有MPI/PVM技术在并行计算上的不足，利用Java技术进行并行分布式计算成了一个研究的热点。ProActive就是其中的一个典型，目前在开发并行计算及网格应用中日益得到重视和应用[1~3]。任务调度是并行分布式计算中的必要环节，调度器是并行计算环境中核心部件之一。由于ProActive的并行计算是基于活动对象(Active Object, AO)的，因此已有的一些调度器不能直接使用，必须根据ProActive的特点重新设计任务调度器。

目前针对 ProActive 的研究主要集中在利用 ProActive 解决具体应用和开发包的功能扩展上，还没有关于任务调度器的研究。因此研究基于 ProActive 的并行计算任务调度器，是十分有有意义的和必要的。

1 ProActive

ProActive 是一个由法国的 INIRA 的 Denis Caromel 教授带领的开发小组开发的适合并行、分布和并发计算，统一框架是具有移动性和安全性的 Java 开源开发包，它是 ObjectWeb consortium 开源中间件的一部分。

有了ProActive，程序员就不需要显式的操作线程对象(如标准的Thread)，以简化在LAN、COW或Grid上应用程序的开发，使用ProActive中间件能方便地开发网格和网络环境下的应用。利用它提供的监控工具IC2D，可以对正在JVM上运行的活动对象进行监控和管理[1]。

2 调度系统的结构及工作过程

采用集中式的调度方式，任务调度是由运行在服务器上的调度器(Scheduler)和动态布署在计算结点上的结点代理(NodeAgent)来共同完成的。

利用ProActive提供的动态部署功能，计算结点的启动和结点代理的部署只需要在服务器上就可轻松完成[2]。调度系统的结构如图 1 所示。

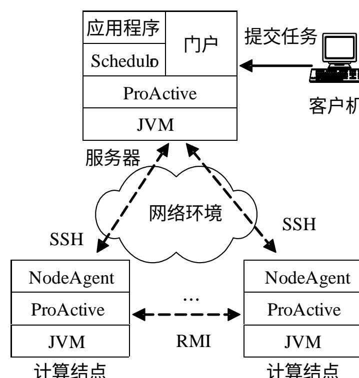

图 1 调度系统的结构

工作过程如下：

(1)服务器结点通过 SSH 连接实现远程计算结点的启动与 NodeAgent 的部署；

(2)NodeAgent 收集各自结点的信息，并返回给调度器；

(3)Scheduler 收到用户请求，根据用户提交的信息选择用户指定的调度算法进行任务调度，生成任务的调度信息；

(4)Scheduler 根据调度信息，用 ProActive 类中提供的静态方法 turnActive(…) 将各子任务分配到各计算结点；

(5)各 NodeAgent 负责本结点上的子任务的执行，并将执行结果送给 Scheduler；

(6)Scheduler 对结果进行归约生成执行结果。

3 任务调度算法

任务调度的主要目的就是要找到一种任务分配方案，使得整个程序的执行时间最少。这种问题绝大多数都是NP完全

基金项目：广西大学科研项目(CC1402, CC1407)
作者简介：董明刚(1977－)，男，硕士生，主研方向：并行分布计算，网格计算；梁正友，副教授、博士
收稿日期：2006-05-17 **E-mail：**d8mg@tom.com

问题，是公认的难题[5]。常见的并行分布式计算可分为独立和协作两种模式。独立模式的调度，相对于协作模式要简单，并且容易得到简单而有效的算法。本节主要研究是独立模式下异构环境中的基于ProActive的动态调度算法。

假设 1 各机器都可以执行用户提交的作业。

假设 2 每台机器在同一时刻只允许相同性质的子任务运行。

定义 1 单位任务：由并行程序划分得到的计算量相当的相互独立的具有相同性质的子任务。

定义 2 机器负载：机器上分配的单位子任务的数目。

3.1 静态算法与池算法

静态算法采用计算量按计算力成比例分配的方法，以保证在异构环境下各机器的执行时间平衡。

设在n个计算结点进行调度，α 为用户程序的计算量，α_i 为结点i分配的任务的计算量。有 $\alpha = \alpha_1 + \alpha_2 + \cdots + \alpha_n$， v_i 是计算结点i的计算速度，可利用性能基准软件测试得到。

$$\alpha_i = \frac{m * v_i}{\sum_{k=1}^{n} v_k} * \frac{\alpha}{m} \tag{1}$$

$$N_i = \left\lfloor m * v_i \Big/ \sum_{k=1}^{n} v_k \right\rfloor \tag{2}$$

N_i为结点i上分配的任务数。α / m 为单位任务的计算量。池算法的基本思想是将生成子任务放入任务池，初始时给每个结点分配等量的任务数，执行完成后从任务池中获得子任务，直到所有任务执行完。这两种算法实现较简单，在此不作详细介绍。

3.2 动态算法

动态任务调度是在执行的过程中动态地调整各结点的负载，使得系统中尽可能地达到负载平衡。动态调度算法必须考虑 3 个问题：启动负载平衡调度时间，确定平衡调度的源结点(负载重的结点)和目标结点(负载轻的结点)，选择需要调度的任务(调度的粒度)。当系统中各结点至少有一个任务正在运行时，进行任务迁移并不能在效率上带来什么好处。只要能使各结点处于忙碌状态，获得较好的资源利用率[4]。本算法采用的是在有结点出现空闲时才进行动态调度，为了避免迁移的抖动问题，每次只对空闲结点迁移任务，不考虑负载偏轻的结点。

(1)数学模型

先按静态算法进行子任务的分配，对于结点 j，当第 1 个子任务执行完成后，统计其执行的时间T_j，然后根据以下公式估算每个子任务的工作量：

$M_j = v_j * T_j$

其中，v_j表示j结点的计算速度，M_j表示执行完的单个子任务的工作量。当每一个结点都有一个OA执行完成后

$$M = \sum_{j=1}^{n} (M_j * L_j) \tag{3}$$

L_j为结点j剩余的任务个数。利用文献[5]中的方法以剩余时间作为负载平衡的标准，当有结点空闲时，可按下面公式计算负载平衡标准$\overline{T}$：

$$\overline{T} = M \Big/ \sum_{j=1}^{n} v_j \tag{4}$$

若请求任务的结点为新加入的结点 ，则

$$\overline{T} = \frac{M}{\sum_{j=1}^{n} v_j + v_k} \tag{5}$$

各结点预测自己所需的计算时间

$$P_j = \frac{M_j * L_j}{v_j} \tag{6}$$

L_j 表示的是结点j上未执行的任务数。正常情况下

$\overline{T} \le P_j \le \overline{T} + T_j$

负载偏重

$$P_j > \overline{T} + T_j + m(j,k) \tag{7}$$

负载偏轻

$$P_j < \overline{T} - T_j - m(j,k) \tag{8}$$

其中，m(j,k)表示子任务从结点 j 迁移到 k 所需的时间。各计算结点对于同类子任务(子任务)迁移时间相差不大，可认为它都为 m，对于迁移时间 m，在任务分配时可测得。可迁移的子任务数为

$$K_j = \left\lfloor \frac{P_j - T_j - \overline{T}}{T_j} \right\rfloor$$

$$s = \sum_{j=1}^{n} K_j \tag{9}$$

可接受的最大数目为

$$n = \left\lfloor \frac{\overline{T} * V_k}{M_k + vm} \right\rfloor \tag{10}$$

V_k为空闲结点的计算速度。实际迁移的数目为

$$r = \min\{s, n\} \tag{11}$$

对于其它负载过轻的暂不分配子任务，等其空闲后再进行子任务迁移。

(2)算法的实现

算法描述如下：

```
收集各计算结点信息；
根据用户提交的作业信息按式(1)、式(2)生成分配文件；
根据分配文件，将各子任务(AO)分配到对应结点上执行；
while(任务还未完成){
if (有 AO 执行完毕){
  返回结果，释放 AO；}
if (有结点空闲){
   if (该结点是新加入的结点 ){
根据式(4)计算平衡时间；   }
   else{
根据式(5)计算平衡时间；   }
根据式 (6)、式(11)进行负载平衡，向空闲结点迁移任务；
else{继续执行下一个 AO }     }
收集任务，生成结果。
```

由于AO是弱迁移性[1]，迁移后只能重新开始执行，因此只能迁移计算结点上那些还没有开始执行的AO。

4 实现及实验

我们先前开发了一个Windows平台 下的基于ProActive的并行计算支持平台WPHPC，并利用ProActive开发包开发了可在此平台中运行的 3 个并行应用实例PI、TSP和Matrix乘[6]。利用ProActive按上述思想实现了一个调度器，将此调度器应用在由 100Mbps高速局域网联接起来的 4 台机器(其中一台为服务器)的WPHPC并行计算平台中进行了实验。3 台计算结点(分别用A、B、C表示)利用nbench测得的计算速度(性能基准)分别为 ：5.6, 5.58, 10.62。测试任务为 420*420 的并行矩阵乘，将其划分为 30 个子任务，分别在 2 个计算结点(A、C)，3 个计算结点(A、B、C)对文中的 3 种调度算法进行了测试，测试结果如图 2 所示。

（下转第 91 页）