分布式动态负载平衡调度的一个通用模型*

陈华平 计永昶 陈国良

(中国科学技术大学计算机系 合肥 230027)
(中国合肥高性能计算中心 合肥 230027)
E-mail:chp@dawnl.cs.ustc.edu.cn

摘要 在大规模并行分布处理系统,特别是网络工作站机群(NOW)系统中,各结点机之间的负载平衡调度是最为常见的关键性问题之一.本文在简单分析了动态负载平衡调度中接收者驱动和发送者驱动这两个常用策略基础上,提出了一个通用的基于混合驱动策略的动态负载平衡调度模型,并给出了曙光1000并行机上的部分实验结果.

关键词 并行分布计算,动态负载平衡,调度模型.
中图法分类号 TP311.1

随着大规模并行分布处理系统,特别是网络工作站机群系统的广泛应用,如何采取有效的调度策略来平衡各结点(机)的负载,从而提高整个系统资源的利用率,已成为人们的研究热点.[1~4]在下面的讨论中,我们假定每个结点上的任务是动态产生的,每个结点的负载大小是动态变化的,因而不考虑静态负载平衡调度方法,只考虑动态负载平衡调度策略.一般说来,动态负载平衡调度可分为集中式调度和分布式调度两大类,前者是由一个调度服务器负责搜集系统负载信息,并由它来决定负载平衡调度方案.集中式调度的主要优点在于实现比较简单,但在结点数较多的大规模并行分布系统中,由于各结点与调度服务器的通讯成为瓶颈,所以调度开销比较大.[1,2,5,6]所以,除非结点数目较少,或者在底层硬件系统中采取比如超级集线器这样的一些特殊实现措施[7],否则在分布存储的并行系统中不大采用集中式平衡调度方法.

分布式调度是根据局部范围内的一些负载信息来进行负载平衡操作,它的最大优点在于具有良好的可扩放性.目前常用的分布式负载平衡调度方法主要有基于随机选择任务移动结点的概率调度算法[8]、根据负载变化差额而基于梯度(Gradient)模型的调度算法[9]以及自适应的近邻契约算法[10]等.但不管哪一种调度算法,它必须解决下列3个问题:什么时候启动负载平衡调度?每次平衡调度的源结点和目标结点是哪些?选择哪些任务进行调度?另外,按一次局部负载平衡调度的启动者来划分,主要可分为接收者驱动和发送者驱动这两大类.在这方面D. L. Eager[11]等做了较多的工作,他们的模拟结果表明,当整个任务负载较重时,接收者驱动策略效率显得好一些.下面我们在分析了这两种策略的基础上,提出一个混合驱动策略,并给出了一个通用的实现模型和一些实验结果.

1 负载平衡调度驱动策略

1.1 接收者驱动策略

这个策略的主要思想是:由空闲结点逐个向邻接结点请求任务,如果请求到任务,则中止请求,否则就继续询问下一个相邻结点.也有可能所有相邻结点都没有满足请求,请求结点就等待,过一段时间后再向相邻结点发出任务请求.很明显,这里等待多长的一个时间段 λ 后再发出任务请求很关键.如果 λ 过短,就会加重忙结点的负担;如果 λ 过长,有可能相邻结点早就有过重负载的任务,这样就浪费了空闲结点的计算资源.λ 本身与任务产生的频率及每个任务的平均执行时间有关,在最后实验比较部分,我们在基于接收者驱动的调度算法中把该参数做成是可调的.

接收者驱动主要有下列几个优点:(1)不需要相互交换负载信息;(2)对于大规模并行计算问题,当每个结点均处

---

* 本文研究得到国家863高科技项目基金和中国科技大学青年基金资助.
作者陈华平,1965年生,在职博士生,副教授,主要研究领域为并行分布处理系统和并行程序设计环境.计永昶,1971年生,博士生,主要研究领域为消息传递系统和并行程序设计环境.陈国良,1938年生,教授,博士导师,主要研究领域为并行分布计算和智能计算.

本文通讯联系人:陈华平,合肥 230027,中国科学技术大学计算机系

本文1996-11-27收到原稿,1997-02-26收到修改稿

于忙状态时，几乎不需要额外调度开销；(3)负载平衡的许多工作由空闲结点来完成，没有给忙结点增加许多额外负担.接收者驱动的主要缺点是：在开始和结束阶段时任务数相对较少，许多任务请求会延迟忙结点的执行.

1.2 发送者驱动策略

这个策略的主要思想是：结点间的任务调度分配由创建任务的结点来执行，至于分配给哪个邻接结点，则主要取决于邻接结点的负载状态，因此，该策略需要交换处理器的负载信息.一个结点有多种方法向邻接结点通知它的负载情况，如定期询问、每当任务数发生变化、接收到执行任务请求、响应请求或者当任务数超过一定阈值时.

发送者驱动的主要优点是：没有过重负载的忙结点，不会被空闲邻接结点所打扰.这一点在系统整个负载较低时尤其重要.发送者驱动的主要缺点是：负载过重的忙结点还要额外增加处理负载平衡调度的负担.

1.3 混合驱动策略

混合驱动策略的主要思想是结合接收者驱动和发送者驱动两者的优点，只有当负载状态发生变化时，才交换负载状态信息，因而减少网络竞争.另外，具有空闲信息的负载消息也被作为任务请求，若不能满足该请求，就记录下请求任务的结点号，以后产生新任务时，就把该任务发送到该空闲的邻接结点上，这样就避免了接收者驱动策略中的反复请求，从而减少通讯开销.同时，为了减少消息传递数量，可在向邻接结点传递任务消息和结果消息时，把自身当前的负载状态消息也附上.因此，每个结点都具有记录邻接结点负载状态信息及相互间任务传递情况的一组属性.

另外，上面3种策略在定位每次平衡调度的源结点或目的结点时，可选择首次适应算法或最佳适应算法.前者只要找到满足平衡调度条件的某个结点即可，不需检查所有邻接结点；而后者每次总是检查比较所有邻接结点，挑选一个最佳的结点来完成这次负载平衡调度.目前许多文献[2,3,5,9]的实验结果分析中并未指出使用的是哪一种.在下面给出的算法及实验结果中使用的是首次适应算法，其主要优点是尽量减少平均调度检测时间，尽早启动调度结点上的任务执行.

2 系统模型及数据结构

假定并行分布系统由若干个结点组成，结点之间通过消息传递系统进行通讯.动态负载平衡调度系统模型如图1所示.在每个结点机上有两类进程：一类是面向用户的应用进程，一个结点机上可能有很多个应用进程，它们构成系统的应用层；还有一类为调度进程，每个结点机只有一个调度进程，它相当于系统的控制层.每个结点的调度进程的运行优先级比应用进程高.它持有一组未计算任务，可把其中某些任务分配给本结点机上的应用进程，或者发送给邻接结点上的调度进程.每个调度进程通过与相邻结点上调度进程的通讯，来为本结点上的应用进程发送/接收任务或结果，所以下面我们提到的应用进程均指本地(结点)应用进程.一般说来，在每个结点的性能相同或差别不大的情况下，每个结点的调度进程是相同的，但这并不是必须的，特别在某些结点的性能比较特殊时，可对这些结点上调度进程的部分参数进行特殊设置.

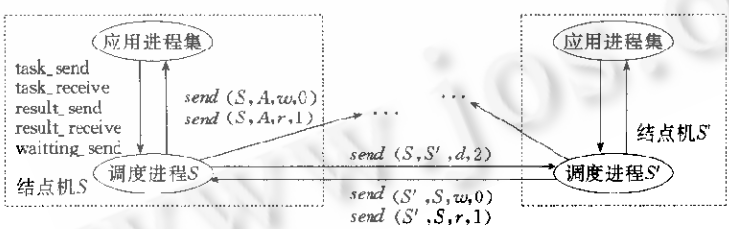

图1 系统调度模型

我们用符号 S,S'和 S''来表示调度进程，用 A,A'和 A''来表示应用进程，由于每个结点上的调度进程是唯一的，所以为了简单起见，我们也可直接用调度进程标识符来表示它所在的结点号，即调度进程 S 所在的结点也可称为结点 S.与调度进程 S 相关的数据结构主要有下列几个：(1)C_s：表示与结点 S 相邻的那些结点上的调度进程集，它也代表了与结点 S 相邻的那些结点集；(2)U_s：表示结点 S 上的应用进程集；(3)W_s：表示结点 S 上还未计算执行的任务集；(4)R_s：表示结点 S 上的需要返回的任务执行结果集；(5)H_s：表示结点 S 上等待执行结果的任务句柄集.

调度进程 S 可用布尔函数 $busy(A)$来记录保存本结点上应用进程 A 的状态.若 $busy(A)$为 true，则表示应有进程 A 处于忙状态；若 $busy(A)$为 false，则表示处于空闲状态.可根据 U_s 中应用进程的状态来确定结点 S 的负载状态.若 U_s 中每个应用进程的 *busy* 函数值为 true，那么 S 的负载状态为 high；若 U_s 中每个应用进程的 *busy* 函数值为 false，那么 S 的负载状态为 low；若 U_s 中一部分应用进程的 *busy* 函数值为 true，另一部分应用进程的 *busy* 函数值为 false，那么 S 的负载状态为 moderate.每个调度进程通过交换负载消息，或者通过传递任务消息和结果消息时附带上负载信息来了解邻接结点的负载状态，所以它必须保存邻接结点的一些局部负载信息.

这里要特别指出的是,针对不同的目标机器,具体分布式负载平衡调度模型的负载状态定义方法有所不同.例如,在每个子结点不支持多进程的情况下,有时可根据每个结点的局部任务队栈[12]这一数据结构中的任务数或任务负载大小来决定该结点的负载状态.下面曙光1000上的实验结果就是利用任务队栈这一方法获得的.

每个调度进程 S 可用函数 $nebload(S')$ 来记录保存邻接结点 S' 的负载状态.每当 S 向 S' 传递一个任务后,置 $nebload(S')$ 为 high,即假定 S' 的所有应用进程处于忙状态.另外,每个调度进程 S 还用一个布尔函数 $ldsend(S')$ 来记录 S 与邻接结点 S' 的负载消息传递情况,它的主要目的在于减少重复不必要的负载消息传递.如果 S 最近传递给 S' 的负载状态信息不是 moderate,那么置 $ldsend(S')$ 为 true,否则为 false.如果 S 从 S' 接收到一个任务,那么置 $ldsend(S')$ 为 false,因为根据上面所述,S' 传递一个任务给 S 后,它所保存的 S 的负载状态为 high.

3 主要调度算法及应用接口

每个调度进程主要循环不断地从局部于它的消息队列里取出消息,然后根据不同的消息类型采取相应的操作.调度系统中主要有任务消息、结果消息、负载状态消息和等待消息4种不同类型的消息,在以下算法中,统一用 $send(*,*,*,*)$ 表示一次消息传递.其中,前两个参数分别为该次消息传递的源进程和目的进程,它们可能是调度进程,也可能是应用进程.第3、4个参数分别为消息内容和消息类型,在算法中就简单地用 w,r,d 和 i 分别表示任务消息、结果消息、负载状态消息和等待消息4种类型消息的内容,并且这4种类型的消息分别标识为0,1,2和3.

在算法中,我们用函数 $id(*)$ 来进一步标识任务,其中"*"可取 W_i 中的 w,R_i 中的 r 或 H_i 中的任务句柄 h,用 $a(i)$ 表示生成标识为 i 的任务的应用进程,用 $s(i)$ 表示该应用进程所在结点的调度进程.通过把 $id(*)$ 内容附在任务消息、执行结果消息和等待消息上,就可以把任务执行结果正确返回传递到产生该任务的应用进程上.

3.1 任务消息

调度进程 S 所收到的任务消息 w 有两个来源:本结点上的应用进程,或者是邻接结点上的调度进程.如果该任务消息来自本结点上的应用进程 A,那么 S 对该消息的主要处理算法为:

(1) 如果存在 $S'\in C_i$,$nebload(S')$ 为 low,那么就执行 $send(S,S',w,0)$,并置 $nebload(S')$ 为 high 后结束;否则,转(2).

(2) 如果存在 $A'\in U_i$,$busy(A')$ 为 false,那么执行步骤(2.1)(2.2)后结束;否则转(3).

(2.1) 执行 $send(S,A',w,0)$,并置 $busy(A')$ 为 true;

(2.2) 如果任给 $A''\in U_i$,$busy(A'')$ 为 true,那么对所有 $S'\in C_i$,执行 $send(S,S',\text{high},2)$ 并置 $ldsend(S')$ 为 true.

(3) 如果存在 $S'\in C_i$,$nebload(S')$ 为 moderate,那么执行 $send(S,S',w,0)$,并置 $nebload(S')$ 为 high 后结束;否则,转(4).

(4) $W_i=W_i\cup\{w\}$;结束.

调度进程 S 首先检查是否能把任务 w 送给负载状态为 low 的邻接结点;若不行,则进一步检查本结点上是否有空闲进程,如果存在的话,S 还检查是否需要把自己的负载状态通知邻接结点;如果不存在空闲进程,那么可把任务送到负载适中的邻接结点上,或者就放在本结点的未计算任务队列 W_i 中.

如果任务消息 w 来自邻接结点上的调度进程 S',那么 S 对该消息的主要处理算法为:

(1) 置 $ldsend(S')$ 为 true.

(2) 如果任给 $A'\in U_i$,$busy(A')$ 为 false,那么执行步骤(2.1)(2.2)后结束;否则转(3).

(2.1) 执行 $send(S,A'',w,0)$,并置 $busy(A'')$ 为 true,其中 $A''\in U_i$.

(2.2) 如果存在 $A'\in U_i$,$busy(A')$ 为 false,那么对所有 $S''\in C_i$,执行 $send(S,S'',\text{moderate},2)$ 和 $ldsend(S'')=\text{false}$.

(3) 如果存在 $S''\in C_i$,$nebload(S'')$ 为 low,那么执行步骤(3.1)(3.2)后结束;否则转(4).

(3.1) 执行 $send(S,S'',w,0)$,并置 $nebload(S'')=\text{high}$;

(3.2) 如果存在 $A'\in U_i$,$busy(A')$ 为 false,那么执行 $send(S,S',\text{moderate},2)$ 并置 $ldsend(S')$ 为 false.

(4) 如果存在 $A'\in U_i$,$busy(A')$ 为 false,那么执行步骤(4.1)(4.2)后结束;否则转(5).

(4.1) 执行 $send(S,A',w,0)$,并置 $busy(A')$ 为 true.

(4.2) 如果存在 $A'\in U_i$,$busy(A')$ 为 false,那么对所有 $S''\in C_i$ 并且 $ldsend(S'')=\text{true}$ 的调度进程 S'',执行 $send(S,S'',\text{moderate},2)$ 并置 $ldsend(S'')$ 为 false;否则,对所有 $S''\in C_i-S'$,执行 $send(S,S'',\text{low},2)$ 并置 $ldsend(S'')$ 为 true;

(5) 如果存在 $S''\in U_i$,$nebload(S'')$ 为 moderate,那么执行(5.1)(5.2)后结束;否则转(6).

(5.1) 执行 $send(S,S'',w,0)$,并置 $nebload(S'')$ 为 high.

(5.2) 如果存在 $A'\in U_i$,$busy(A')$ 为 false,那么执行 $send(S,S',\text{moderate},2)$ 并置 $ldsend(S')$ 为 false.

(6) $W_s=W_s\cup\{w\}$;结束.

上面算法按序检查是否所有本机进程为空闲状态,是否存在一个负载过轻的邻接结点,是否存在一个状态为空闲的进程以及是否存在负载适中的邻接结点.这些检查次序非常关键,它关系到如何最快最有效地获得负载平衡.如果每次调度后 S 的负载状态仍保持适中,那么它就不需要传送负载消息给邻接结点,为此,ldsend 属性记录了最近是否发送了 moderate 负载消息,只有当 $ldsend(*)$ 为 true 时,才发送 moderate 负载消息给邻接结点.

3.2 结果消息

调度进程 S 所收到的结果消息 r 可能来自应用进程,或者来自邻接结点上的调度进程.如果该结果消息来自本结点上的应用进程 A,那么 S 对该消息的主要处理算法为:

(1) 设结果 r 所对应的任务标识 $id(r)$ 为 i,首先根据 $s(i)$ 值执行下列操作后转(2).

(1.1) 如果 $s(i)=S$,即该结果所对应的任务由 S 本结点上的应用进程 $a(i)$ 产生,那么进一步判别是否存在 $h\in H_s: id(h)=i$,如果存在,那么执行 $H_s=H_s-\{h\}$ 和 $send(S,a(i),r,1)$;如果不存在,那么只要执行 $R_k=R_k\cup\{r\}$ 即可;

(1.2) 如果 $s(i)\neq S$,即该结果所对应的任务由其它结点上的应用进程产生,那么执行 $send(S,s(i),r,1)$;

(2) 如果存在 $w\in W_s$,那么执行 $send(S,A,w,0)$ 和 $W_s=W_s-\{w\}$ 后结束;否则转(3).

(3) 置 $busy(A)$ 为 false;根据下列不同条件执行后结束.

(3.1) 如果任给 $A'\in U_s$,$busy(A')$ 为 false,那么对所有 $S'\in C_s$,执行 $send(S,S',\text{low},2)$ 并置 $ldsend(S')$ 为 true;

(3.2) 如果存在 $A''\in U_s$,$busy(A'')$ 为 true,那么对所有满足 $ldsend(S')=\text{true}$ 的 $S'(\in C_s)$,执行 $send(S,S',\text{moderate},2)$ 并置 $ldsend(S')$ 为 false.

如果该结果消息来自邻接调度进程,那么处理步骤与上面所描述的算法类似,只是不需检查任务队列 W_s,S 的负载状态也不发生变化.

3.3 负载消息

调度进程 S 所收到的负载消息 d 只可能来自邻接结点上的调度进程 S',S 对该负载消息的主要处理步骤为:如果 $d\neq\text{high}$ 并且存在 $w\in W_s$,那么执行 $send(S,S',w,0)$,$W_s=W_s-\{w\}$,$nebload(S')=\text{high}$;否则,只要通过执行 $nebload(S')=d$ 来保存传送过来的负载状态即可.

3.4 等待消息

调度进程 S 所收到的等待消息只可能来自本结点上的应用进程 A,消息参数主要为任务标识符 i.S 对该消息的主要处理步骤为:如果存在 $r\in R_s: id(r)=i$,那么执行 $R_s=R_s-\{r\}$,$send(S,A,r,1)$;否则只要执行 $H_s=H_s\cup\{h\ id(h)=i\}$ 即可.

3.5 应用接口

应用进程主要通过下列几个函数调用来操作调度进程:(1)send_task:发送一个任务到调度进程,由它来对该任务进行调度分配;(2)receive_task:通过调度进程从 W_s 中接收一个任务;(3)send_rcsult:把任务执行结果传送给调度进程,由调度进程负责把该结果返回到产生该任务的应用进程上;(4)receive_result:接收该应用进程先前所产生任务的执行结果,如没取到,应用进程就被挂起,直到结果到达;(5)send_waiting:与 receive_result 类似,只是异步接收,即如没取到结果,应用进程不中止,只是把等待结果消息传给调度进程.另外,在主从式的并行程序设计中,一个任务 w 必须接收到儿子任务 w_c 的执行结果后才能继续运行,可以通过把儿子任务 w_c 的执行结果直接返回到 w 的父亲任务 w_f 的办法,来减少由于结果等待而引起的应用进程阻塞.

4 实验结果

我们在"中国合肥高性能计算中心"的曙光 1000 并行机上对上述 3 种调度策略进行了一些性能分析比较.曙光 1000 并行机中分布存储的的 32 个计算结点采用 mesh 连接,相互之间通过消息传递系统进行通讯与同步.我们选择的测试程序 $mgsort(*)$ 为四路归并排序,当每个子任务(相当于一个数据划分段)中排序元素数目不大于 20 万时,就直接使用快速排序,否则就继续划分动态生成多个子任务来执行.每个结点机 P_i 均有一个局部队栈[12](Queue-Stack) QS_i,本地执行时从队栈顶部取任务,给其它结点机传送任务时从栈底取,由于栈底任务的并行粒度较大,这样可以尽量减少负载平衡调度次数.

每个结点 P_i 的负载状态可根据 QS_i 中的数据元素数目 Y_i 来确定.如果 $Y_i=0$,即 QS_i 为空,那么 P_i 的负载状态为 low;如果 $Y_i\geqslant\alpha*m/n$,那么 P_i 的负载状态为 high,其中 m 为总的排序元素数目,n 为当前所使用的处理机结点数

目，$\alpha(>1)$为调整因子，它与负载平衡调度频率成反比，在我们的测试实验中置 $\alpha=2$；如果 $0<Y_i<\alpha * m/n$，那么 P_i 的负载状态为 moderate. 图 2 是曙光 1000 上测试程序 *mgsort*(*m*)在不同结点数目下采用不同驱动策略所获得的执行加速比，其中 $m=64\times10^4$. 由于通讯/计算比值较大，所以整个加速性能不是很高.

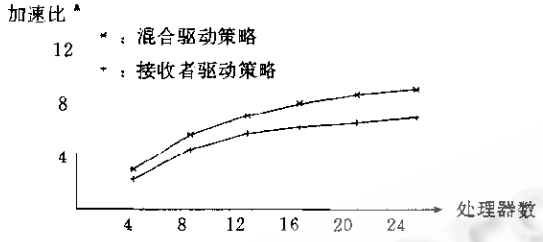

图2　曙光1000上不同驱动策略的加速比

参考文献

1 Hesham El-Rewini, Theodore G Lewis, Hesham H Ali. Task scheduling. Englewood Cliffs, New Jersey, PTR Prentice Hall, 1994

2 Joosen W, Pollet J. The efficient management of task clusters in a cynamic load balancer. In: Proceedings of the International Conference'94 on Parallel Distributed Systems. Hsinchu, Taiwan, Dec. 1994. 19～21

3 Joosen W *et al*. Design and implementation of an experimental load balancing environment. In: Proceedings of the European Spring Conference. May, 1991

4 Chen Hua-ping, Li Jing, Chen Guo-liang. Task scheduling in parallel distributed computing (Part 2). Journal of Computer Science, 1997,24(2):23～27

5 Feng M D, Yuen C K. Dynamic load balancing on a distributed system. In: Proceedings of the 6th Symposium on Parallel and Distributed Processing. Dallas, Texas, Oct. 1994. 26～29

6 Chen Hua-ping, Lin Hong, Chen Guo-liang. Heuristic task scheduling in parallel distributed computing. Computer Reseach and Development, 1997,34(Supplementary Issue):74～78

7 Gene Eu Jan, Ming-Bo Lin. Effective load balancing on highly parallel multicomputers based on superconcentrators. In: Proceedings of the International Conference'94 on Parallel Distributed Systems. Hsinchu, Taiwan, Dec. 1994. 19～21

8 Soumen Chakrabarti *et al*. Randomized load balancing for tree-structured computation. In: Proceedings of the International Conference'94 on Parallel Distributed Systems. Hsinchu, Taiwan, Dec. 1994. 19～21

9 Lin F C H, Keller R M. The gradient model load balancing method. IEEE Transactions on Software Engineering, 1987,SE-13(1):32～38

10 Shu W, Kale L V. A dynamic scheduling strategy for the Chare-Kernel system. In: Proceedings of Supercomputing'89. Reno, Nevada, Nov. 1989. 389～398

11 Eager D L *et al*. A comparison of receiver-initiated and send initiated adaptive load sharing. Performance Evaluation, 1986,6: 53～68

12 Chen Hua-ping, Chen Guo-ling. Mechanism of parallel inference on MIMD models. Mini-Micro Systems, 1996,17(4):7～11

A Universal Model of Distributed Dynamic Load Balancing

CHEN Hua-ping　JI Yong-chang　CHEN Guo-liang

(*Department of Computer Science　University of Science and Technology of China　Hefei*　230027)
(*China High Performance Computing Center(Hefei)*　*Hefei*　230027)

Abstract　Load balancing among processors is a critical problem on a massively parallel and distributed system, especially on a network of workstations. This paper has analyzed the receiver-initiated and sender-initiated strategies to be commonly used, and then proposed a universal model of dynamic load balancing based on the mixed strategy. Lastly, some experiment results on DAWN-1000 have been given.

Key words　Parallel distributed computing, dynamic load balancing, universal scheduling model.

Class number　TP311.1