多处理机工作池方式负载平衡技术在机器人的应用①

Work Pool Load Balancing Technology of Multiprocessor in the Application of Robots

唐俊奇　(湄洲湾职业技术学院 福建 莆田 351254)

摘　要： 文章阐述了利用集中式工作池动态负载平衡和分散式工作池动态负载平衡这两种方法来解决机器人多处理机控制器中各处理器间合理地分配计算问题,以获得最快的可能执行速度,使得控制器中的 CPU 资源能够充分利用并协同工作,以最大限度地提高机器人性能。

关键词： 机器人 负载平衡技术 集中式工作池 分散式工作池

1　引言

作为机器人的核心部分,机器人控制技术已经历了经典控制技术、现代控制技术和智能控制技术的发展过程。由于计算机科学技术及其它相关学科的长足进步,使得机器人的研究在高水平上进行,同时也为机器人控制器的性能提出更高的要求。人们采用了许多方法来提高机器人控制器的性能,最有效的方法就是采用多处理机作并行计算,提高 控制器的计算能力。如:有腿的步行机器人,其控制器采用多 CPU(多处理机)结构(多处理机结构)控制方式[1]即:上、下位机二级分布式结构,上位机负责整个系统管理以及运动学计算、轨迹 规划等. 下位机由多个 CPU(处理机)组成,每个 CPU 控制一个关节运动,这样的控制器工作速度和控制性能明显提高。但是控制器中能否充分利用 CPU 资源协同工作,以最大限度地提高机器人性能则成为机器人性能的瓶颈[2]。

2　负载平衡技术的引入

负载平衡用于在处理器间合理地分配计算,以获得最快的可能执行速度,它有两方面的含义:首先,大量的并发访问时把数据流量分担到多个节点设备上分别处理,减少用户等待响应的时间;其次,单个重负载的运算分担到多台节点设备上做并行处理,每个节点设备处理结束后,将结果汇总,返回给用户,系统处理能力得到大幅度提高。要实现一个好的负载平衡通常有两种方案,一种是静态负载平衡,另外一种是动态负载平衡[3,4]。

2.1 静态负载平衡

静态负载平衡中,需要人工将程序分割成多个可并行执行的部分,并且要保证分割成的各个部分能够均衡地分布到各个 CPU 上运行,也就是说工作量要在多个任务间进行均匀的分配,使得达到高的加速系数[5]。

实际情况中存在许多不能由静态负载平衡解决的问题,比如一个大的循环中,循环的次数是由外部输入的,事先并不知道循环的次数,此时采用静态负载平衡划分策略就很难实现负载平衡且它需要人工干预,显然不可取。

2.2 动态负载平衡

动态负载平衡是在程序的运行过程中来进行任务的分配达到负载平衡的目的。

3　动态负载平衡的引用

动态负载平衡中,任务是在程序运行期间被分配到处理器的。我们把动态负载平衡划分为两类:

- 集中式动态负载平衡
- 分散式动态负载平衡

3.1 集中式动态负载平衡

3.1.1 集中式工作池的设计思想

① 基金项目:湄洲湾职业学院校级科研项目(MZY0707)

© 中国科学院软件研究所 http://www.c-s-a.org.cn

在集中式动态负载平衡中，主进程（主处理器）持有要执行的任务集，任务由主进程发给从进程，从进程完成一个任务后，主进程会请求另一个任务。这种机制称为工作池方法。所谓工作池方法就是：当有处理器处于空闲时就向其分派工作，工作池中拥有所要完成的任务集（池），工作池技术可以用于那些任务很不相同、大小不同的问题。一般最好分配较大或最复杂的任务，如果在计算中分配大任务较晚，完成小任务的从进程会闲呆着，以等待大任务的完成。

当执行期间任务数会发生变化时，也较适合应用工作池技术。在一些应用，特别是在机器人的搜索算法中，一个任务的执行会产生一些新的任务（比如机器在执行足球比赛场境），尽管最终任务数会减少为零以指示计算的完成。可用一个队列存放当前等待的任务，如图1所示。如果所有任务大小相同且同等重要，则可以用简单的先进先出队列；如果某些任务比其他任务更要（如期望更快地得到解），就要首先把这些任务送到从进程。其他的一些信息，如当前的最佳解等，可以用主进程加以保存。

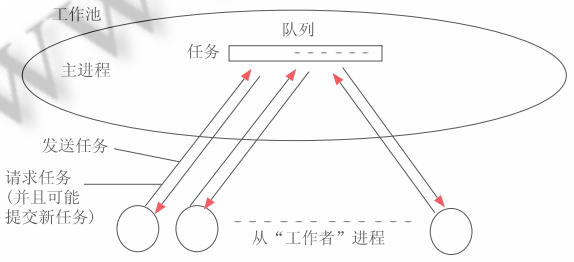

图1　集中式工作池

集中式动态负载平衡的一个最突出的优点是，主进程很容易识别计算会何时终止。对一个计算，如果其中的任务是从任务队列中获取的，则当下面两项都满足计算时就终止：

●任务队列为空。

●每个进程已请求了别的任务，而又没有任何新的任务产生。

3.1.2 集中式工作池中的并行实现

在并行实现中我们将使用一个集中式工作池来存放顶点队列 Vertex_queue[ ]作为任务，每个从进程从顶点队列取得顶点，按照前面图1所示的方式返回新的顶点。对于要确认边和计算距离的从进程，它们需要访问存放图权值的（邻接矩阵或邻接列表）数组和存放当前最小距离的数组（dist[ ]）。如果这些信息由主进程持有，就要向主进程发送消息以访问这些消息，这会导致非常严重的通信开销。由于存放图权值的结构是固定的，可将该结构拷贝到每个从进程中，假定使用的是拷贝的邻接矩阵。现在，我们再假设距离数组 dist[ ]为中央存放的，且与顶点一起整体拷贝，也可单独对距离进行请求。代码可为如下形式：

```
主进程：
while(vertex_queue( ) ! =empty) {
recv(PANY,Source - =Pi); /* 从进程任务请求 */
v=get_vertex_queue( );
send(&v,Pi); /* 发送下个顶点和 */
send(&dist,&n,Pi); /* 和当前距离数组 dist[ ] */
send(&j,&dist[j],PAny,source=Pi); /* 新的距离 */
append_queue(j,dist[j]); /* 增加队列顶点 */
}; /* 和更新距离数组 */
recv(PANY,source=Pi); /* 从进程任务请求 */
send(Pi,termination_tag); /* 消息终止 */
从进程（进程 i）
send(Pmaster); /* 发送请求任务 */
recv(&v,Pmaster,tag); /* 获取顶点数 */
if(tag ! =termination_tag) {
recv(&dist,&n,Pmaster); /* 获取距离数组 dist[ ] */
for (j = 0;j<n;j++) /* 获取下一条边 */
if (w[v][j] ! = infinity) { /* 如果该条边存在 */
        newdist_j = dist[v] + w[v][j];
        if (newdist_j < dist[j]) {
        dist [j] = newdist_j;
send(&j,&dist[j],Pmaster); /* 添加顶点队列 */
}}} /* 发送更新的距离 */
```

很明显，顶点数和距离数组可放在一个消息中发送。还要注意各个从进程或许有不完全一样的距离，因为它们是由不同的从进程连续更新的。

主进程等待任何从进程的请求，但必须对进行请求的指定从进程加以响应。在伪代码中，source = p，

© 中国科学院软件研究所 http://www.c-s-a.org.cn

用来表示消息源。在实际编程系统中,可通过让每个从进程发送其标识(可能作为唯一标记)来确认源。在 MPI 中,能通过读取 MPI_RECV( )例程返回的状态字找到实际的消息源。

集中式工作池中的并行实现虽然克服了上述各种方法的缺点,但由于其自身的原因,也有一个严重的缺点是:主进程一次只能发送一个任务,在初始任务发送后,它只能一次一个地响应新的任务请求,因此,当很多从进程同时请求时就存在着潜在的瓶颈。

3.2 分散式动态负载平衡

3.2.1 分散式动态负载平衡的设计思想

由于集中式工作池的一个严重的缺点使得主进程一次只能发送一个任务,在初始任务发送后,它只能一次一个地响应新的任务请求,因此,当很多从进程同时请求时就存在着潜在的瓶颈。如果从进程很少且任务又是计算密集型的,则集中式工作池是会令人满意的,但对于粒度任务和有很多从进程的情况,则把工作池分布在多个地点将会更合适。

图 2 所示为分布工作池。这里,主进程已将初始的工作池分布几个部分,并且将每一部分发送给一组"M 主进程(Mo 到 Mn-1)"中的一个。每个 M 主进程控制一组从进程。对于优化问题,M 主进程会找到本地最优,然后将其返回给主进程,用内部结点分割工作,就可形成一棵树,这是将一个任务等分成子任务的基本方法。对于一棵二叉树,可在树的每一层进程把任务的一半送给一棵子树,而把另一半送给另一棵子树。另一种分布方法则是让人进程实际持有工作池的一部分并对这一部分求解。

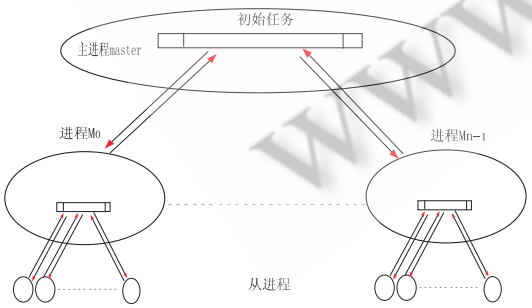

图 2　分布式工作池

3.2.2 全分布式工作池

一旦进程分配了工作负载,它又产生自己的任务,就存在进程间相互执行任务的可能性,如图 3 所示。任务可按如下方法传递:

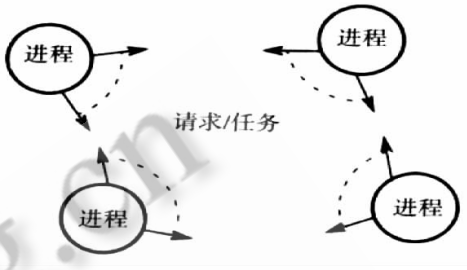

图 3　分散式工作池

● 由接收者启动方法。

接收者启动策略与发送者启动策略除了是由轻负载节点启动,要求其它节点把任务发送给它之外,其它基本相同。

接收者启动策略同样引入 M 以区分轻、重负载节点,引入相关域以确定交互范围。

在启动时,所有节点开始执行计算任务,经过一段时间之后,一旦某个节点发现自身成为轻负载节点,就试图在它的相关域中均匀地分布负载。具体地:设该轻负载节点的负载为 l_p,相关域中有 K 个节点,其负载分别为 $l_1,\ldots\ldots,l_k$,则平均负载 L_{avg} 为:

$$L_{avg}=\frac{1}{K+1}(l_p+\sum_{K=1}^{K}l_k)$$

为了达到均匀分布,应求得相关域中节点应该传递给轻负载节点的负载量 m_k。我们首先引入权 h_k 以避免负载从负载更轻的相关域中的节点被迁移到该节点。如果 $L_{avg}<l_k$,则 ,否则 $h_k=0$。那么 m_k 为:

$$m_k=(L_{avg}-l_p)g_k/\sum_{K=1}^{K}h_k$$

随后该节点就可以按照 m_k 发出接受任务的请求了。

● 由发送者启动方法。

发送者启动策略也引入了一个阀值 M 来把所有的处理节点划分成轻负载节点和重负载节点,所有当前剩余负载 t > M 的节点都被称为重负载节点,t < M 的节点被称为轻负载节点。发送者启动策略还需要为每个节点定义一个相关域,节点只与它的相关域中的节点进行交互和任务传递。一个直观的相关域的定义是把所有与之相邻的节点作为相关域。

在启动时,所有节点开始执行计算任务。在执行

© 中国科学院软件研究所 http://www.c-s-a.org.cn

一段时间之后,节点就开始检查其自身是否是重负载节点。如果是重负载节点,则它就试图在相关域中均匀地分布任务。具体地:设该重负载节点的负载为 l_p,相关域中有 K 个节点,其负载分别为 $l_1,\ldots,l_k$,则平均负载 L_{avg} 为:

$$L_{avg} = \frac{1}{K+1}(l_p + \sum_{i=1}^{K} l_i)$$

为了达到均匀分布,应求得重负载节点应该传递给每个相关域中节点的负载量 m_k。我们首先引入 h_k 以避免负载被迁移到相关域中负载最重的重负载节点。如果 $L_{avg} > l_k$,则 ,否则 $h_k = 0$。那么 m_k 为

$$m_k = [(l_p - L_{avg})h_k / \sum_{i=1}^{K} h_i]$$

随后该节点就可以按照 m_k 向各个相关节点发送任务了。

在接收者启动方法中,一个进程向它选择的其他进程请求任务。典型地,当一个进程有很少或没有任务执行中,会向其他进程请求任务。已经表示该方法在高系统负载时会工作得很好。在发送者启动方法中,一个进程向它选择的其他的进程发送任务。在这种方法中,典型的是一个负载很重的进程会向愿意接收的其他进程传递一些它的任务。已经表明这种方法在整修系统负载较轻时工作得较好。另一种选择是将两种方法结合起来。不幸的是,确定进程负载状况的代价较昂贵。在系统负载非常重时,由于缺少可用进程,负载平衡也可能会很能实现。

现在我们来讨论接收者启动环境中的负载平衡,不过,它也全都适用于发送者启动方法。有几种可行的策略。可将进程组织成一个环,进程向其最近的邻居请求任务。环形结构适合一个用环形互连网络构成的多处理系统。类似地,在一个超立方体中,进程可向每一维上与其直接相连的一个进程请求任务。当然,对于任何策略,都要小心不要让已接收的任务不停地传递。

3.2.3 分布式终止检测算法

迄今为止,我们已考虑了任务的分配,现在我们来看看如何终止这些分布式任务。我们讨论一下终止条件。

当计算是分布式的时候,识别计算是否已经结束是很困难的,除非是一个进程可得到的一个解的问题。通常在时间 t 的分布终止需要满足如下条件:

在时间 t,对于所有进程,存在有特定应用的本地终止条件。

在时间 t,进程间没有消息在传送。

这些终止条件和集中式负载平衡系统中的那些相比,它们之间的细微差别是必须考虑传送中的消息。对于分布式终止系统,第二个条件是必要的,因为传送中的消息也许会重新启动一个已终止的进程。可以想象一下一个进程已到达它的本地终止条件并准备终止,而此时有另一进程正向它发送一条消息。通常第一个条件相对容易识别,只要每个进程在满足其本地终止条件时向主进程发送一条消息便可。不过第二个条件就较难识别。消息在进程间传送的时间预先是不知道的,可以等待足够长的时间以便传送中的消息到达,但这种方法是不受欢迎的,而且使代码在不同的结构上不可移植。

3.2.4 分布式工作池中的并行实现

有一种分布式作池方法能用于我们的求解问题。任务队列,下面的从进程代码中的 vertex_queue [ ]也可以是分布式的。有一种方便的方法是:让从进程 I 只围绕顶点 I 搜索,如果顶点 I 在队列中存在,就让进程 I 拥有顶点 I 的队列项。换句话说,队列中有一个元素专门用来存放顶点 I,该项在进程 I 中。数组 dist[ ]也分布在进程中间,以便进程 I 保存当前到顶点 I 的最小距离,为了确认顶点 I 的边,进程 I 还需存储顶点 I 的邻接矩阵/列表。

根据我们的安排,算法可按如下方式进行。由一协调进程激活搜索,将源顶点装载到适当的进程。在我们的例子中,A 是第一个搜索的顶点。首先激活指派给 A 的进程,该进程立即在顶点周围开始搜索,找出到相连顶点的距离;然后,把该距离送到相应的进程。到顶点 j 的距离将被送到进程 j,以与它当前存储的值相比较,如果当前存储的值较大就被替换。在例中,到顶点 B 的距离与负责顶点 B 的进程联系。按这种方式,在搜索中将更新所有的最小距离,如果 d[I]的内容改变,进程 I 就要被重新激活,再次搜索。图 4 显示出了消息传递的情况,注意消息传递分布在

© 中国科学院软件研究所 http://www.c-s-a.org.cn

许多从进程间,而不是集中在主进程上。

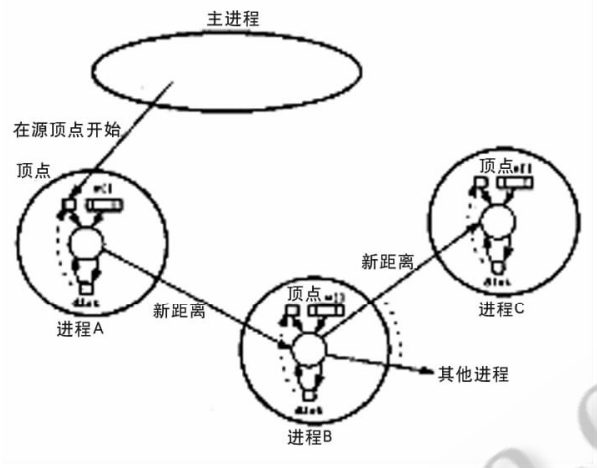

图 4　分布式图搜索

从进程的代码段可为如下形式:

```
从进程(进程 I)
recv(newdist,PANY);
if (newdist < dist) {
    dist = newdist;
vertex_queue = TRUE;      /* 加到队列中 */
 }else vertex_queue  = = PALSE;
if (vertex_queue = = TRUE)      /* 开始搜索周围的顶点 */
     for (j = 0; j < n; j + + )      /* 获取下一条边 */
          if (w[j] ! = infinity) {
           d = dist + w[j];
  send(&d,Pj); }   /* 传送距离到进程 j 中 */
```

上述代码可简化为:

```
从进程(进程 i)
recv(newdist,PANY);
if (newdist < dist) {
dist = newdist;          /* 开始搜索周围的顶点 */
     for(j = 1;j < n,j + + )      /* 获取下一条边 *
          if(w[j] ! = infinity) {
           d = dist + w[j];
send(&d,Pj);      /* 传送距离到进程 j 中 */
}
```

需要用一种机制来重复这些动作,并在所有进程空闲时终止,该机制必须处理传输中的消息。最简单的解决方法是利用同步消息传递,其中一个进程只有在对方收到消息后才能继续运行。

值得注意的是,一个进程只有在其顶点放入顶点队列之后才是活动的。有可能很多进程不是活动的,从而导致一种低效的解决方案。如果将一个顶点分到每个处理器,则该方法对大图也是不实用的,在那种情况下,可把一组顶点分配一个处理器上。

4　结束语

综上所述,采用工作池方式来实现动态负载平衡算法,可以克服了其他算法的缺点,使得机器人控制器中能够真正实现各处理机系统的负载平衡,充分利用 CPU 资源,达到我们预期的目的。

参考文献:

1 普杰信,严学高. 机器人分布式计算机控制系统结构的性能分析. 机器人,1994,16(3):144 – 149.

2 唐俊奇. 多处理机系统 Cache 共享数据乒乓效应的研究. 莆田学院学报,2006,13(2):51 – 54.

3 孙强南,孙昱东等编著. 计算机系统结构. 北京:科学出版社,2000.

4 殷兆麟. Java 语言程序设计. 北京:高等教育出版社,2002.

5 TMASEVIC M., AND V. MILUTINOVIC(1993), The Cache Coherence Problem in Shard – MemoryMultiprocessor:Hardware Solutions, IEEE CS Press,Los Alamitos,California.

© 中国科学院软件研究所 http://www.c-s-a.org.cn