松耦合多处理机系统负载平衡技术研究

唐俊奇
（湄洲湾职业技术学院，福建 莆田 351254）

【摘　要】文章主要讨论了在松耦合多处理机系统下，几种基本的平衡算法及其适用面。阐述了两种动态负载平衡的设计思想、终止检测算法并提出在集中工作池和分散式工作池中并行实现的解决方法。

【关键词】松耦合多处理机；动态负载平衡；终止检测算法
【中图分类号】TP393.02　【文献标识码】A　【文章编号】1673-1891(2006)03-0047-07

前言

目前，网络应用正全面向纵深发展，企业上网和政府上网已见成效。随着网络技术的发展，教育信息网和远程教育网络等也得到了普及，尤其是校园网络承担了越来越多的来自教学、科研及管理方面的应用。校园网络中心作为校内局域网络的数据存储发布、流量的管理控制、用户的管理以及内外网络转接控制中心，必须解决因流量增加所带来的一系列问题。传统的网络中心方案中，当流量增加后，一般是升级单一的服务器系统，这往往会造成过高的投入和维护成本，极大地降低了性能价格比。因此，如何优化信息中心系统的性能，以提高整个信息系统的处理能力是人们普遍关心的问题。

1 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，大量的并发访问或数据流量分担到多台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。

2 多处理机系统负载平衡原理

在TCP/IP协议中，数据包含有必要的网络信息，因而在网络缓存或网络平衡的具体实现算法里，数据包的信息很重要。但由于数据包是面向分组的(IP)和面向连接的(TCP)，且经常被分片，没有与应用有关的完整信息，特别是和连接会话相关的状态信息。因此必须从连接的角度看待数据包——从源地址的端口建立到目的地址端口的连接。

平衡考虑的另一个要素就是节点的资源使用状态。由于负载平衡是这类系统的最终目的，那么及时、准确地把握节点负载状况，并根据各个节点当前的资源使用状态动态调整负载平衡的任务分布，是网络动态负载平衡集群系统考虑的另一关键问题。

一般情况下，集群的服务节点可以提供诸如处理器负载，应用系统负载、活跃用户数、可用的网络协议缓存以及其他的资源信息。信息通过高效的消息机制传给平衡器，平衡器监视所有处理节点的状态，主动决定下个任务传给谁。平衡器可以是单个设备，也可以是一组平行或树状分布的设备。

3 几种基本的网络负载平衡算法及其适应面

平衡算法设计的好坏直接决定了集群在负载均衡上的表现，设计不好的算法，会导致集群的负载失衡。一般的平衡算法主要任务是决定如何选择下一个集群节点，然后将新的服务请求转发给它。有些简单平衡方法可以独立使用，有些必须和其它简单或高级方法组合使用。而一个好的负载均衡算法也并不是万能的，它一般只在某些特殊的应用环境下才

收稿日期：2006-04-30
作者简介：唐俊奇(1967-)，男，福建莆田人，副教授、高级工程师。

?1994-2017 China Academic Journal Electronic Publishing House. All rights reserved.　http://www.cnki.net

能发挥最大效用。因此在考察负载均衡算法的同时,也要注意算法本身的适用面,并在采取集群部署的时候根据集群自身的特点进行综合考虑,把不同的算法和技术结合起来使用。

3.1 轮转法

轮转算法是所有调度算法中最简单也最容易实现的一种方法。在一个任务队列里,队列的每个成员(节点)都具有相同的地位,轮转法简单的在这组成员中顺序轮转选择。在负载平衡环境中,均衡器将新的请求轮流发给节点队列中的下一节点,如此连续、周而复始,每个集群的节点都在相等的地位下被轮流选择。这个算法在DNS域名轮询中被广泛使用。

轮转法的活动是可预知的,每个节点被选择的机会是1/N,因此很容易计算出节点的负载分布。轮转法典型的适用于集群中所有节点的处理能力和性能均相同的情况,在实际应用中,一般将它与其他简单方法联合使用时比较有效。

3.2 散列法

散列法也叫哈希法(HASH),通过单射不可逆的HASH函数、按照某种规则将网络请求发往集群节点。哈希法在其他几类平衡算法不是很有效时会显示出特别的威力。例如,在UDP会话的情况下,由于轮转法和其他几类基于连接信息的算法,无法识别出会话的起止标记,会引起应用混乱。而采取基于数据包源地址的哈希映射可以在一定程度上解决这个问题:将具有相同源地址的数据包发给同一服务器节点,这使得基于高层会话的事务可以以适当的方式运行。相对称的是,基于目的地址的哈希调度算法可以用在Web Cache集群中,指向同一个目标站点的访问请求都被负载平衡器发送到同一个Cache服务节点上,以避免页面缺失而带来的更新Cache问题。

3.3 最少连接法

在最少连接法中,平衡器纪录目前所有活跃连接,把下一个新的请求发给当前含有最少连接数的节点。这种算法针对TCP连接进行,但由于不同应用对系统资源的消耗可能差异很大,而连接数无法反映出真实的应用负载,因此在使用重型Web服务器作为集群节点服务时(例如Apache服务器),该算法在平衡负载的效果上要打个折扣。为了减少这个不利的影响,可以对每个节点设置最大的连接数上限(通过阈值设定体现)。

3.4 最低缺失法

在最低缺失法中,平衡器长期记录到各节点的请求情况,把下个请求发给历史上处理请求最少的节点。与最少连接法不同的是,最低缺失记录过去的连接数而不是当前的连接数。

3.5 最快响应法

平衡器记录自身到每一个集群节点的网络响应时间,并将下一个到达的连接请求分配给响应时间最短的节点,这种方法要求使用ICMP包或基于UDP包的专用技术来主动探测各节点。

在大多数基于LAN的集群中,最快响应算法工作的并不是很好,因为LAN中的ICMP包基本上都在10ms内完成回应,体现不出节点之间的差异;如果在WAN上进行平衡的话,响应时间对于用户就近选择服务器而言还是具有现实意义的;而且集群的拓扑越分散这种方法越能体现出效果来。这种方法是高级平衡基于拓扑结构重定向用到的主要方法。

3.6 加权法

加权方法只能与其他方法合用,是它们的一个很好的补充。加权算法根据节点的优先级或当前的负载状况(即权值)来构成负载平衡的多优先级队列,队列中的每个等待处理的连接都具有相同处理等级,这样在同一个队列里可以按照前面的轮转法或者最少连接法进行均衡,而队列之间按照优先级的先后顺序进行均衡处理。在这里权值是基于各节点能力的一个估计值。

上述几种负载平衡算法在不同的场合下能发挥其应有的作用,但是它们都受到了各种条件的制约,无法达到人们预期的效果,为此人们也在不断地探求更好的负载平衡算法以适应不断发展的网络的需要。

4 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

4.1 集中式动态负载平衡

4.1.1 集中式工作池的设计思想

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

?1994-2017 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

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,特别是在搜索算法中，一个任务的执行会产生一些新的任务，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。

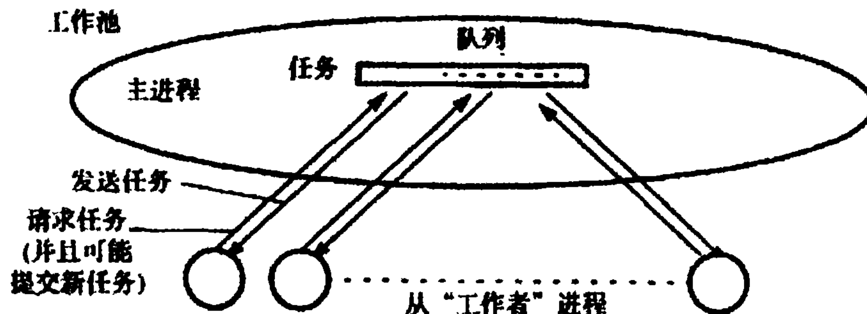

图 1　集中式工作池

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，那么当下面两项都满足足时计算时就终止：

·任务队列为空。

·每个进程己经请求了另一个任务，而又没有任何新的任务产生。

4.1.2 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[]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[] 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); /* request task from slave */
v = get_vertex_queue( );
send(&v, Pi); /*send next vertex and * /
send(&dist, &n, Pi); /*current dist array */
.
send(&j, &dist[j], PAny, source = Pi); /* new distance */
append_queue(j, dist[j]); /* append vertex to queue */
/* and update distance array */
};
recv(PANY, source = Pi); /* request task from slave */
send(Pi, termination_tag); /* termination message */
```

从进程(进程 i)

```
send(Pmaster); /* send request for task */
recv(&v, Pmaster, tag); /* get vertex number */
if(tag ! = termination_tag) {
recv(&dist, &n, Pmaster); /* and dist array */
for (j = 0; j <n; j + + ) /* get next adge */
if (w[v][j] ! = infinity) {/* if an adge*/
newdist_j = dist[v] + w [v][j];
if (newdist_j < dist[j]) {
dist [j] = newdist_j;
send(&j, &dist[j], Pmaster); /* add vertex to queue */
} /* send updated distance */
```

?1994-2017 China Academic Journal Electronic Publishing House. All rights reserved.　http://www.cnki.net

}

}

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，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。

4.2 分散式动态负载平衡

4.2.1 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“(Mo 到 Mn - 1)”中的一个。每个迷你主进程控制一组从进程。对于优化问题，迷你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从进程实际持有工作池的一部分并对这一部分求解。

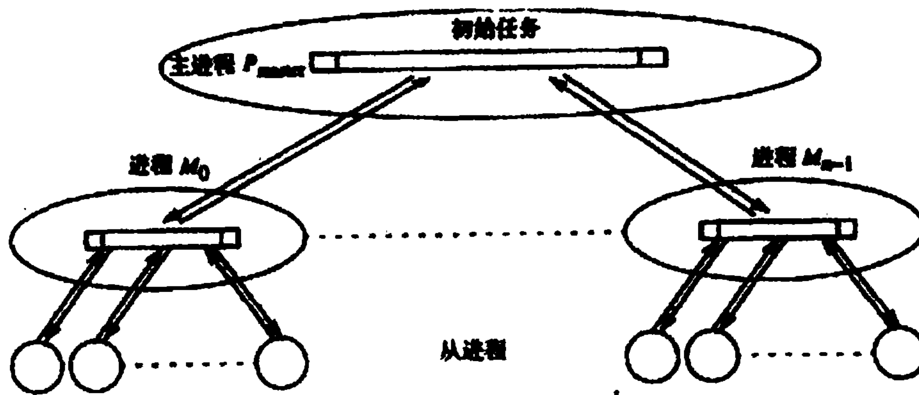

图2 分布式工作池

4.2.2 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：

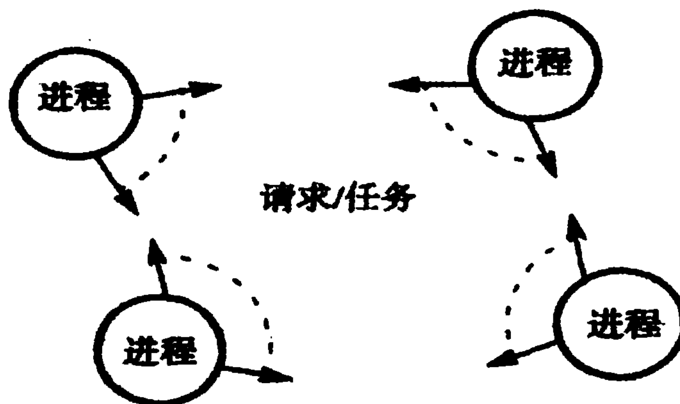

图3 分散式工作池

?1994-2017 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

· 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, ..., 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hk 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:

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

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阈值 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, ..., 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。环形结构知是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。

4.2.3 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

?1994-2017 China Academic Journal Electronic Publishing House. All rights reserved.　　http://www.cnki.net

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。

4.2.4 分布式工作池中的并行实现

有一种分布式工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。

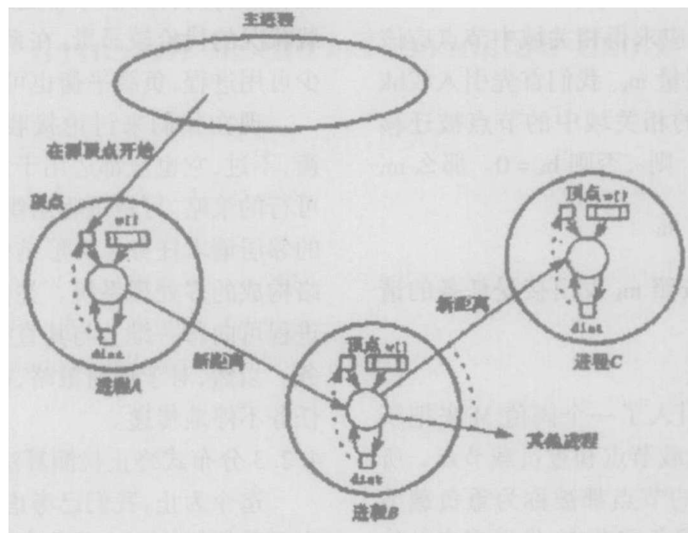

图 4 分布式图搜索

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; /*add to queue/*
} else vertex_queue = = PALSE;
if (vertex_queue = = TRUE) /* start searching around vertex*/
for (j = 0; j <n; j + + ) /* get next adge*/
if (w[j] ! = infinity){
d = dist + w[j];
send(&d, Pj); /* send distance to proc j */
}
```

上述代码可简化为:

```
从进程(进程 i)
recv(newdist, PANY);
if (newdist <dist){
dist = newdist; /* start searching around vertex*/
for(j = 1; j <n, j + + ) /* get next adge*/
if(w[j] ! = infinity){
d = dist + w[j];
send(&d, Pj); /* send distance to proc j */
}
```

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

?1994-2017 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

那种情况下,可把一组顶点分配一个处理器上。

5 结束语

综上所述，长期以来人们采用几种基本的负载平衡算法，它们在不同的场合下虽能发挥其应有的作用,但是它们都受到了各种条件的制约,无法达到人们预期的效果；而如今采用工作池方式来实现动态负载平衡算法，可以克服基本的负载平衡算法的缺点,这样我们就可达到预期的效果。

参考文献:

[1]孙强南,孙昱东等编著. 计算机系统结构[M]. 北京:科学出版社,2000.
[2]吴企渊,梁燕编著. 计算机操作系统[M]. 北京:清华大学出版社,2004.
[3]殷兆麟.Java语言程序设计[M]. 北京:高等教育出版社,2002.
[4]陈智勇. 计算机系统结构[M]. 西安:西安电子科技大学出版社,2004.

Loose Couple Multiprocessing Machinesystematic Load Balanced Technical Research

TANG Jun – qi

(Meizhouwan Vocational technology college, Putian Fujian 351254)

Abstract: The article discusses mainly under the much processors system of loose coupling, several kinds of basic balance algorithms and application. Explain two dynamic design philosophy of load balance , stop algorithm of measuring and put forward concentrate job pool and disperse work pool parallel solution.

Key words: Loose couple multiprocessing machine; Development load balance; Termination detection algorithm

(责任编辑:张荣萍)

?1994-2017 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net