基于簇结构的分层调度器的设计和实现*)

Design and Implementation of a Hierarchical Scheduling Model on Cluster

郭建新 李 冀 魏明亮 陆桑璐 陈贵海 谢 立
(南京大学计算机软件新技术国家重点实验室 南京210093)

Abstract In this paper we present a cooperative scheduling model based on hierarchical cluster structure in a large-scale NOW environment. In our model, the nodes in NOW are stratified according to their physical characteristics, and scheduling is performed across all the layers cooperatively in phases to allocate the batch tasks and interactive tasks efficiently and rationally. In this way, high throughput and low response time are achieved at the same time. The performance evaluation has shown good efficiency.

Keywords NOW, MPI, Co-scheduling, Batch task, Interactive task

1 引言

目前,工作站性能大幅度提高,高速网络技术日益成熟,工作站网络(NOW,Network Of Workstations)得到迅猛的发展。但随着网络环境日趋大规模化,传统方式下基于节点的调度方式显然已不能适应,因为随着节点数目的增多不仅管理更加复杂,而且系统开销也将增加很多,所以我们提出层次调度方式。其主要特点如下:

1)在不同层次上采取不同的调度策略。最底层的调度针对节点采取精细而复杂的协同调度算法,以获得更高的性能;而上层的调度,因为通讯开销等因素,采取粗粒度的调度。基于层次的调度不仅能提高性能而且便于管理。

2)采用协同调度策略。协同调度是NOW环境中常用的调度策略,已经有很多基于协同调度的调度算法实现,比如Max,Dasic[1,2]等。协同调度主要有以下优点:①提高效率。协同调度根据并行任务的各个进程间通讯交互这一特征,尽可能让并行任务的各进程在不同节点上同时被调度运行,使需要通讯的进程总处于相同状态[3],这样避免了进程之间相互等待的时间,也就避免了进程过度的调入调出,同时减少环境上下文切换(Context Swap)带来的开销;②易于编程。SPMD(Single Program Multiple Data)是并行程序设计中用到的最多的编程风格,而协同调度能更有效地调度按照SPMD风格编写的任务[4]。

3)引入交互任务。一般的NOW环境中的调度只针对批任务或实时任务进行调度,我们在层次式协同调度中引入交互任务,也就是当系统满足要求就执行并立即返回结果。它不象实时任务那样有严格的时间要求,也不象批处理任务那样不需要立即返回结果。引入了交互任务程序员就可以象RPC一样来使用系统提供的功能。同时系统中还存在批任务,引入批任务的目的是提高系统的吞吐量。基于层次的协同调度器对两种任务进行合理调度,既增加了系统的吞吐量又减少了交互任务的响应时间。

本文讨论了分层调度器的模型及其各个模块的功能和调度的策略。介绍实验环境和测试结果,并提出今后工作方向。

2 系统结构

2.1 系统简介

调度器的关键问题是调度的粒度、对象和策略,调度的粒度包括任务、进程和线程;调度的对象包括一个簇,一台机器,一个处理器;策略指何时进行调度和如何调度,抢占式还是非抢占式。调度器设计的原则是正确、简单和高效。

本调度器是任务级的调度器,全局调度器针对簇,簇调度器是针对一台机器,都是采用抢占式的调度。

任务性质也是调度器要考虑的,在系统中,我们把任务分为交互任务和批处理任务。对交互任务调度器考虑的是最短响应时间,而对批处理任务调度器考虑的是系统吞吐量的最大化。

*)本文得到国家863高技术项目(863-306-ZT02-03-01)的资助。郭建新 硕士生;李 冀 硕士生;魏明亮 硕士生;陆桑璐 副教授;陈贵海 教授;谢 立 教授,博士生导师,研究领域均为分布式处理和并行计算。

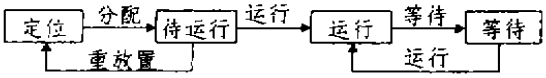

图1 任务状态及变化

我们不考虑进程的动态迁移，因为进程级的动态迁移代价高，并且I/O操作的动态迁移难度也很大，所以我们仅考虑任务的静态重放置，也就是当任务没有启动时才对任务进行必要的迁移，当任务运行后通过控制任务的执行状态来改变系统的负载状况，当系统中某些节点负载过重时，可以通过迁移待运行的任务和挂起已执行的任务来调节这些节点的负载。

任务调度器分为两个层次，全局层和簇层，任务的重放置是在全局层次上的调度，而任务的运行和等待转换是在簇层次上的调度。系统中存在两类任务，批任务和交互任务，我们必须考虑两类任务的特点，在不同层次上对它们进行调度。

2.2 全局调度器

(1)*模型结构* 全局调度器的结构如图2。全局任务队列中放的是Globle_Task_info(Task_ID, Require_Processor, Float_mul_number, Custer_ID, Status)，Task_ID是任务的全局唯一编号，Require_Processor为所需要的最小处理器数目，Float_mul_number为任务计算量的估计值。Custer_ID是如果任务已经被分配到了某一簇，Status是本任务的当前状态(wait, run)，当任务被分配到某一簇后其Status为wait，当任务被簇调度器调度后其Status为run，簇任务队列中放的是Cluster_Task_info(Task_ID, Require_Processor, Float_mul_number, Status)，Status是本任务的当前状态(wait, run, stop)，当任务分配到节点上运行后Status为"wait"，当任务被启动后Status为"run"，当任务被挂起时其Status为"stop"。全局调度器和簇调度器通过网络连接，簇调度器向全局调度器发送任务状态改变信息和簇状态信息，全局调度器负责分配任务给簇调度器，并且监测系统当前状况进行负载静态平衡。

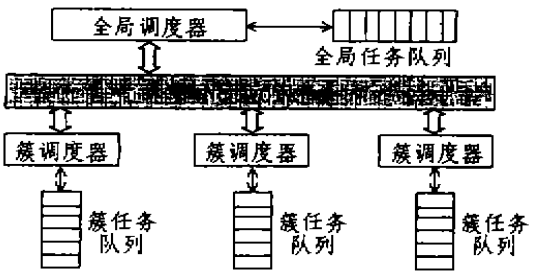

图2 全局调度模型

(2)*调度策略* 用户提交任务给全局调度器，调度器给任务分配全局唯一的编号(Task_ID)，并根据任务性质赋初始优先级，交互任务赋为0，批任务赋为100，然后把任务放入队列，全局调度器根据任务的优先级和各簇的状态进行调度和负载平衡。具体策略如下：

Ⅰ. 当新任务到达时，检查该任务是交互任务还是批任务，如果是交互任务，选择一个没有运行交互任务的簇，如果有多个簇满足要求，则采取最坏最先策略，即选择一个正好能满足要求的簇来执行该任务，如果没有簇满足要求，则在所有的簇中采取最优最先策略，即在所有的簇中选择一个最空闲的簇来执行该任务；如果是批任务，则首先检查是否有簇没有执行交互任务，如果有多个簇满足要求，则采取最坏最先策略，即选择刚好符合要求的簇来执行该任务，如果所有簇都有交互任务，则放入队列。

Ⅰ. 当负载不平衡时，我们的负载平衡策略包括对未执行任务的重置和对已执行的任务的控制。在全局调度器中考虑的是对未执行任务的重置。当接收到所有的簇发送的状态信息后，检测它们状态，如果忙则在该簇的等待队列中选择一优先级最高的任务收回，放入全局任务队列中，并把该任务的优先级提高；如果闲则从全局队列中选择一个优先级最高且未运行的任务，分配给该簇。

2.3 簇调度器

2.3.1 *簇调度器结构* 如图3所示，由四个部分组成，分别是协同调度器、状态收集器、节点和自适应通讯平台。①协同调度器从状态收集器获得簇内接点状态，调度系统中的任务；②状态收集器负责收集簇内节点的状态；③节点模块运行于簇内节点上，负责收集本机器状态，并根据协同调度器发出的调度请求做出相应的动作，其结构如图4所示，由协同调度单元、状态收集单元和自适应通讯单元组成，协同调度单元接收簇调度器发送的控制消息并通过自适应通讯单元对任务进程进行控制，状态收集单元从操作系统获得系统信息，从自适应通讯单元获得系统通讯状况并发送给系统的状态收集器；④自适应通讯平台是调度器、状态收集器同节点通讯的平台，由若干各自适应通讯单元组合而成。

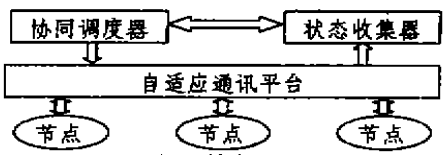

图3 簇调度器

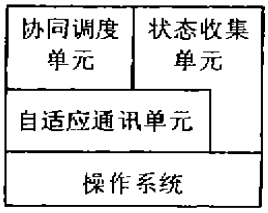

图4 节点模型

自适应通讯平台单元(ACPU,Adaptive Communication Platform Unit)是建立以 MPI 为通讯平台来构造的通讯模型结构,如图5所示。主要由通讯控制器、调度控制器和数据缓冲组成。其中通讯控制器是核心模块,主要功能有:(1)通讯控制器为上层应用程序提供通讯接口。应用程序在调用接收或发送数据的接口时,检测本任务状态是否被挂起,如果状态为被挂起则等待直到状态被恢复为运行状态才继续运行,如果状态为运行则进行要求的操作,可以是发送数据或接收数据。发送数据直接调用 MPI 的发送接口,接收数据时从缓冲读取。之所以不直接通过 MPI 读取数据是因为当本任务挂起时,如果有其他系统任务发送数据给该任务,就可以缓冲数据,这样可以提高系统的效率。(2)通讯控制器监控应用程序的通讯状况。一方面记录节点的总通讯量,并把结果提交给通讯监视器,再由通讯监视器处理后提交给状态收集单元,状态收集器中关于通讯状态的数据就是由通讯控制器产生并通过状态收集单元发送而得到的;另一方面当协同调度单元发送挂起某一任务时,协同调度单元向调度控制器发送需要挂起的任务的任务号,调度控制器根据接收到的任务号决定挂起某一任务,只要设置该任务的通讯状态为挂起,同理当协同调度器释放一任务时,向调度控制器发送需要释放的任务的任务号,调度控制器根据接收到的任务号决定挂起某一任务,并把该任务的通讯状态设置为运行。在系统中我们通过控制任务的通讯来控制任务的执行,因为协同任务之间相互通讯,要求各任务处于相同的状态,即要么都处于运行状态,要么都处于挂起状态。(3)调度控制器接收协同调度单元发送的调度消息,并通过向通讯控制单元发送消息来执行协同调度单元的操作。(4)数据缓冲是用来缓冲进程间通讯的数据,因为当进程被挂起时,其他进程向该进程发送的数据都被存放在数据缓冲中。数据缓冲可以使通讯的进程异步执行,也就是发送数据时不需要接受方确认就可继续执行,有利于提高系统效率。

2.3.2 调度策略 簇调度器是协同调度器,协同调度一般有两类策略:一类是静态的,即给协同任务分配处理器后不进行动态负载平衡,如 Max 和 Grab[4],另一类是动态的,进行动态迁移,如 Dasic[5]。而我们考虑到动态调度的代价大,而静态调度不能满足交互任务的要求,我们采取折衷的策略,通过控制任务的执行状态进行负载平衡,这样既可以提高效率又减少动态迁移的代价。通过引入批任务,可以提高系统的吞吐量。具体策略如下:

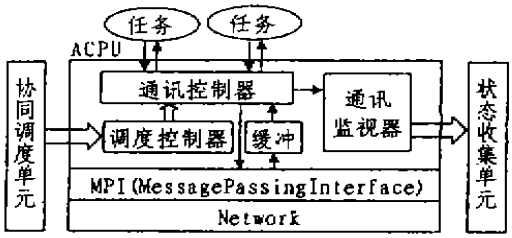

图5 自适应通讯平台单元

Ⅰ.接收全局调度器提交任务,首先判断该任务是交互任务还是批任务,如果是批任务则检查本簇内的节点的负载状况,如果有空闲节点,则把该批任务分配到合适数目的空闲节点,如果没有空闲节点,则把该任务放入等待队列;如果该任务是交互任务,检查本簇内的节点的负载状况,如果有空闲节点,则把任务分配到合适数目的空闲节点,如果没有空闲节点或空闲节点数目不足,则检查是否有批任务在执行,如果有则挂起优先级最低的批任务,并在空闲的节点和被挂起的批任务运行的节点中按最优最先的策略选择合适数目的节点运行该交互任务。

Ⅱ.当簇内有任务执行结束,首先在簇挂起队列中查找被挂起的任务,选择被挂起任务和前一任务执行节点集合的交比重最大的任务,恢复该任务。如果没有被挂起的任务或比重小于阈值,则在等待队列中选择优先级最高的任务执行。

Ⅲ.当批任务被挂起时,增加该批任务的优先级,这样减少了被再挂起的可能性。当有任务到达或有任务完成时,增加等待队列中的任务的优先级,通过增加等待队列中任务的优先级可以避免任务长时间得不到调度,因为全局调度器选择高优先级的任务进行全局的负载平衡,而且簇调度器也会优先选择高优先级的任务执行。

2.3.3 状态收集器 为调度器提供调度的依据,本系统中状态信息有两层含义,一个是节点的状态,另一个是簇的状态,两个状态有各自的收集和管理方式。

(1)节点状态信息。我们用两个参数来表示节点的信息(Computing_State,Communication_State),分别为该节点的计算状态和通讯状态。节点计算状态是该节点的剩余计算能力,我们通过该节点的 CPU 空闲率和 CPU 处理能力的乘积表示;节点的通讯状况是剩余

通讯能力，通过节点上的自适应通讯平台单元(ACPU)获得，ACPU 监测并统计任务的通讯量并把结果提交给状态收集器，具体工作流程在下文中将详细论述。我们用节点的计算状态和通讯状态的和来表示该节点的状态，于是得到下面的公式。

$$NodeState_i = Computing\ State_i + Communication\_State_i$$

(2)簇状态信息。簇状态主要考虑节点状态总和以及该簇的任务队列长度。节点状态表示该节点的空闲处理能力，队列长度是指运行队列的长度和等待队列的长度的加权。运行队列长度是该簇中运行的任务的数目，它表示该簇繁忙的程度。等待队列的长度是指将要运行的任务的数目，是潜在的负载，因为等待的任务可能会迁移到其他簇，所以我们用经验值 ρ 加权，ρ 是任务被迁移的可能性，这里取默认值0.5。我们用节点状态的总和与队列的长度的比值来表示簇的状态，于是得到下面的公式，其中 n 表示该簇中的节点的数量，Run_Queue 为该簇的运行队列的长度，Wait_Queue 为该簇的等待队列的长度。

$$ClusterState_j = \sum_{i=1}^{n} NodeState_i / (Run\_Queue_j + Wait\_Queue_j * \rho)$$

(3)状态收集算法。在簇内，采用令牌(Token)，把各节点组成一虚拟的环。由簇服务器周期(Ts)发出状态收集令牌，各节点把状态信息加入令牌并发送给下一节点，依次直到返回到簇服务器。各节点周期采集本节点信息并根据启发算法求出将来一段时间(Tc)本节点可能的状态。簇服务器收到令牌后刷新簇服务器中保存的节点状态信息表并根据收到的所有状态信息计算出本簇的总资源的空闲程度。状态收集过程如下：

Ⅰ. 簇服务器周期(Ts)发起状态收集 Token 并把 Token 发给下一个节点。

Ⅱ. 节点接收到一个 Token，把自己的状态填入 Token，再把状态 Token 发给下一节点。

Ⅲ. 重复上一操作直到簇服务器接收到发出的 Token，簇服务器根据收到的所有状态信息计算出本簇的总资源的空闲程度。

把启发过程放在节点上完成，减轻了簇服务器的负载，并且簇服务器发送状态 Token 的周期和节点上启发预测的时间设为相等的值，这样为任务分配决策时只要以上次收到的各节点的状态信息为依据，因为上次收到的信息是本周期内各节点现在状态的预测。

对于全局状态收集，我们把簇看成一系列的节点，那么由这些簇可以组成一个高一层次的簇，这样全局状态的收集和簇内状态收集一致。

状态收集必须考虑两个问题，一是对系统的负载的影响，二是对系统负载预测的准确程度。影响负载的主要因素是发起状态请求的周期，周期小对系统负载影响大，周期大对系统负载影响小；同样发起状态请求的周期也是影响决定负载预测准确程度的主要因素，周期小将提高准确程度，周期大将降低准确程度。

3 性能测试

3.1 测试环境

系统运行的物理环境如图6。12台 RS6000运行 AIX4.3.1，通过 ATM 交换机连接成星形结构；5台 SunSparc 运行 Solaris2.6，首先连接在100M 交换式集线器上，然后再连上 ATM 交换机；若干台 PC 运行 Linux 或 Windosw NT，通过集线器连接后再和 ATM 交换机相连。我们分层时考虑通讯距离，分簇时考虑节点属性。首先分层，我们以 ATM 为中心，按照距离 ATM 的远近可把机器分为2个部分，一个部分是所有的 RS6000，它们距离 ATM 为1；另一部分是 SunSparc 和 PC，它们距离 ATM 为2。然后分簇，对于 SunSparc 和 PC，我们把性能相近的机器组成一个簇，所以可以分为 SunSparc 和 PC 两个簇。对于 PC，考虑到不同的操作系统，可以把 PC 再分为 Windows 簇和 Linux 簇，因为对于体系结构相同并且运行的操作系统也相同的环境构成的同构的 NOW 环境是比较容易实现的。

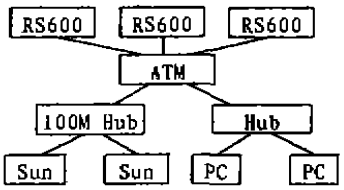

图6 实验物理环境

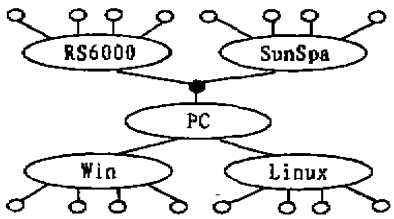

图7 簇结构图

根据以上的讨论，我们把系统中的机器分为4个簇，每一个簇再确定一个簇服务器，负责节点的加入和删除以及本簇和其他簇的信息交换和协调工作。这样系统可以逻辑上构成如图所示的簇结构，分为三个层次：PC 簇、RS6000簇和 SunSparc 簇为第一个层次，在 PC 簇下的 Windows 簇和 Linux 簇为第二层次，每个簇下的节点是第三个层次。结构如图7所示，根据这种

层次结构我们的调度器分为两个层次，全局调度器和簇调度器。在本文第二部分已经做了具体介绍。

3.2 测试结果

我们在系统中测试两个例子，一个是计算素数的交互任务，另一个是解密的批任务。下面分别给出加速比，进行负载平衡和不进行负载平衡时交互任务响应时间的比较。

测试环境：在由6台 RS6000通过155兆 ATM 连接的簇。

测试1 计算素数，计算1到12345678之间的素数的数目，是一个交互任务，主程序提交任务请求后等待结果。

测试2 解密任务，对经过 Unix 中通用加密库函数加密的数据进行解密，采取穷举法，主程序提交任务后立刻结束。

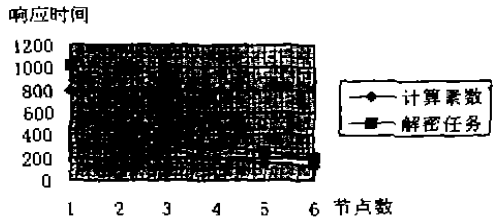

图8 响应时间和节点数的关系

1)加速比测试。图8是节点数目和响应时间的关系图，两个任务分别执行。由图中可以看出随着节点的数目增加，响应时间减小，而加速比(响应时间/节点数)也跟着减小，主要原因是当节点增加，启动时间也增加，而且启动时间是由参与节点中最慢的节点确定，当节点数目过多时，启动时间对加速比的影响不可忽略，这也是我们把性能类似的机器组成簇的原因。

2)进行负载平衡和不进行负载平衡时交互任务响应时间比较。图9是交互任务进行负载平衡和不进行负载平衡的响应时间的比较。假设批任务和交互任务同时分布在相同的节点上，通过控制批任务的执行来控制交互任务的执行。由图9可以看出进行负载平衡减少了交互任务响应时间。当节点数为1时，没有进行负载平衡的测试，因为1个节点时的两个任务是串行任务，不可能进行负载平衡。

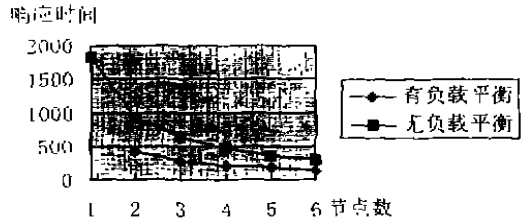

图9 负载平衡的比较

结论 本文给出了基于层次的调度模型的设计和实现，并讨论了设计过程中相关的要点，最后给出了测试结果，结果表明基于层次的调度策略是有效的。因为采用了集中式的管理方式，所以系统的稳定性不高，特别是全局调度器，当全局调度器崩溃后系统也将崩溃。今后我们将做全局调度器的备份调度器，对调度信息进行备份，这样就可以增加系统的健壮性。

参考文献

1 陆桑璐，谢立. 基于簇结构的负载平衡模型——簇平衡. 计算机研究与发展，1998(9)

2 Ousterout J K. Scheduling Techniques for Concurrent Systems. In: Proc. of the 3rd Intl. Conf. on Distributed Computing Systems (ICDCS). IEEE Press, Oct. 1982. 22～33

3 Feitelson D G, Rudolph L. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. Journal of Parallel and Distributed Computing, 1992(Dec.): 306～318

4 da Silva F A B, Scherson I D. Improvements in Parallel Job Scheduling Using Gang Service. Parallel Architectures, Algorithms, and Networks, 1999, (I-SPAN'99). Proceedings Fourth International Symposium. 1999. 268～273

5 陆桑璐，谢立. 一个动态自适应的迁移和协同调度模型. 软件学报，1997，8(10)：752～759

6 Franke H, Pattnaik P, Rudolph L. Gang scheduling for the IBM SP-2 workstation cluster. System Sciences, 1997. Proceedings of the Thirtieth Hwaii International Conference. on Volume. 1. 1997. 630～631