DOI:10.3969/j.issn.1673-4785.201503013
网络出版地址:http://www.cnki.net/kcms/detail/23.1538.TP.20150413.1505.001.html

基于动态任务调度的 STDS 算法设计研究

刘正
(哈尔滨工程大学 计算机科学与技术学院,黑龙江 哈尔滨 150001)

摘　要:任务调度是计算机多核处理器系统获得高性能的关键,而现有的多核任务调度算法研究,大多侧重于静态调度下的算法优化和负载均衡,对动态调度及动态负载均衡研究较少。针对动态调度,并结合异构多核的特点,提出一种基于核负载均衡的动态任务调度算法 STDS。算法通过合理设定调度粒度,降低调度频率,从而减少调度消耗时间;根据异构多核处理器各核处理性能的差异,设置内核负载上下限值,控制内核负载保持在同一水平,以达到负载均衡效果。算法依据等待时间长短、任务间通信大小和内核负载轻重因素对任务进行实时调度,并可通过实时因子、负载因子等参数设置 3 种因素的影响比重,以满足系统的不同需求。仿真实验显示,在内核数目较多的系统中,STDS 算法更加高效,在保证任务处理速度的同时有较好负载均衡。

关键词:动态任务调度;负载均衡;调度粒度;等待时间;异构多核系统
中图分类号:TP316.4　**文献标志码**:A　**文章编号**:1673-4785(2015)02-0324-09

中文引用格式:刘正. 基于动态任务调度的 STDS 算法设计研究[J]. 智能系统学报, 2015, 10(2): 324-332.
英文引用格式:LIU Zheng. Research on STDS algorithm designing based on dynamic task scheduling[J]. CAAI Transactions on Intelligent Systems, 2015, 10(2): 324-332.

Research on STDS algorithm designing based on dynamic task scheduling

LIU Zheng
(College of Computer Science and Technology, Harbin Engineering University, Harbin 150001, China)

Abstract: Efficient task scheduling is the key for multi-core systems to achieve high performance. However, most of the existing multi-core researches on task scheduling algorithms focus on algorithm optimization and load balancing under the static scheduling rather than dynamic scheduling and dynamic load balancing. Scalable task duplication based scheduling (STDS) is a new dynamic-balanced algorithm based on core load balancing. STDS was put forward specifically for dynamic scheduling and integrating the characteristics of heterogeneous multi-core. It shortens scheduling time by reasonably setting the scheduling granularity and reducing the frequency of scheduling. STDS sets the maximum and minimum ranges of the kernel load according to the different processing performance of each core of the heterogeneous multi-core processor, therefore controlling the core load at the same level and achieving the effect of load balance. The wait time of task and communications between tasks and kernel load will be considered together to pick up the most appropriate task during scheduling. The importance of each element can be adjusted by assigning another value to the real-time factor and load factor enabling it to adapt to various environment. The experimental results verified that the STDS algorithm is more efficient in the system with the most kernels. It also keeps a good load balance while maintaining the speed of task execution processing.

Keywords: dynamic task scheduling; load balancing; scheduling granularity; waiting time; heterogeneous multi-core system

收稿日期:2015-03-09.　**网络出版日期**:2015-04-13.
基金项目:国家自然科学基金资助项目(61003036);中央高校基本科研业务费专项资金资助项目(HEUCF100606).
通信作者:刘正.E-mail:liuzheng528@hrbeu.edu.cn.

动态任务调度可以根据运行的时情况动态地将任务分配到各个内核上,由于需要实时地收集、存储并分析状态信息,动态调度的实施有一

定的系统开销,但这种开销和付出通常是有回报的[1]。多核处理器系统中的任务调度已被证明是 NP 问题[2],除个别特殊情况外,目前尚未有一种多项式时间算法可以求得最优解,只能得到一个最大限度接近最优解的解,因此改进算法的效率并构建任务调度实现机制成为研究重点,很多学者在这方面做了大量工作。

比较经典的调度算法有 Min-Min[3]、Max-Min[4]、MCT[5]、MET[6]算法。Min-Min 算法实现简单、执行速度快。算法首先通过计算待调度任务在任一可用内核上的最早完成时间,取最小值作为该任务的最早完成时间,然后选取所有待调度任务中最早完成时间最小的一个任务进行调度。缺点是如果任务集中存在过多的执行时间比较小的任务,那么大任务将无法得到及时的执行。Max-Min 算法同 Min-Min 算法类似,同样需要计算每一任务的最早完成时间,不同的是 Max-Min 算法首先调度最早完成时间最大的任务,将其分配到对应的内核上。缺点是小任务等待时间过长、影响执行效率,也可能造成负载不均衡。MCT 算法以任务完成时间最早为目标进行调度,每次将到达的任务分配到完成时间最早的内核上,但该算法忽略了任务的执行时间,可能将任务分配到执行时间较长的处理机上运行。MET 算法以任务执行时间最短为目标进行调度,每次将到达的任务分配到执行时间最短的内核上,其缺点是易造成较强内核负载过多任务,导致内核间负载不均衡,降低系统性能。

清华大学的石威等[7]提出了一种相关任务图的均衡动态关键路径调度算法,采用动态关键路径技术并均衡考虑关键路径结点和非关键路径结点,优先调度对相关任务图调度长度影响最大的就绪结点,从而极大地缩短任务图的调度长度。李仁发[8]对多核处理器系统任务调度研究进展进行讨论,对不同模型下的多核系统任务调度算法相关研究进行了分析总结,从调度算法分析和调度实现框架 2 个方面探讨了近年来多核任务调度的国内外研究进展情况。文中对任务分配、任务模型优化、调度器实现和任务迁移等当前亟待解决的问题,进行了深入探讨,并指出了下一步主要的研究方向,为多核处理器相关研究提供参考。吉林大学的耿晓中提出了一种动态负载均衡模型[9],将影响多核处理器负载均衡的因素分为 5 类:多核系统的负载均衡环境、用户提交的任务属性、系统的负载评价、系统所采用的调度策略以及系统的调度评价指标,为多核动态调度的负载平衡研究提供了理论指导;徐雨明等[10]将遗传算法和启发式方法有机地结合,根据染色体双螺旋结构模型,提出了一种异构系统中依赖任务调度的双螺旋结构遗传算法。该算法首先采用启发式方法,产生较佳的任务调度优先队列,然后模仿碱基互补配对方法,提高算法的有效性和收敛速度;Vinay 等[11]提出一种多核处理器动态任务调度策略,调度思想是依据任务的执行时间和依赖此任务的任务数量确定此任务优先级,当某个内核空闲时,选择一个优先级最高且为就绪状态的任务分配给该内核。此策略结构简洁,算法复杂度低,但每个内核上只分配一个任务,且调度时需要对数据段加锁以保证数据一致性,当内核数目较多时,经常发生多个内核同时申请分配任务的情况,所以不可避免出现内核空闲等待现象,不能完全发挥多核处理器优势。

文中对文献[11]的动态调度策略进行改进,克服内核可能空闲等待问题,并综合负载均衡策略,提出多核动态任务调度算法(shared task data structure,STDS)。算法通过分析任务等待时间、任务间通信开销和内核负载等因素,确定任务优先级,并据此分配任务。通过仿真实验验证,该算法在保证任务处理速度的同时有较好负载均衡,比较适用于内核数目较多的系统。

1　模型

在多核任务调度研究中,为便于研究理解,通常用抽象的任务调度模型描述任务调度系统。多核任务调度模型通常由系统模型、任务模型、任务调度算法和任务映射图组成。

1.1　系统模型

系统模型是指依据系统硬件环境构建出可进行数学量化分析的数学模型,文中用二元组 SM(system model) = { P, Rate} 表示,其中:SM 表示系统模型;$P = \{ P_0, P_1, \cdots, P_k, \cdots, P_{m-1} \}$ 为处理器内核集合,P_k 表示第 k 个处理器内核,$|P| = m$ 为处理器中内核的总个数。多核处理器根据内核在执行能力方面的差别可划分为同构和异构 2 类,同构多核处理器通过增加同种处理器内核数量提升性能,而异构多核处理器通过集成不同计算能力的内核来优化处理器,实现多核处理器性能发挥的最大化,在能耗、面积等方面有着巨大的优势[12]。文中以有限数目的异构多核处理器为研究基础,集合 P 中处理器内核速度并不完全相同,用 sp_k 表示内核 P_k 的处理速度。

Rate = $\{\cdots, \text{Rate}_{s,t}, \cdots\}$ 表示处理器内核间数据传输速率集合,元素 $\text{Rate}_{s,t}$ 表示处理器内核 P_s 与 P_t 间的数据传输速率。

核间互连技术是多核处理器设计的关键,当前多核处理器核间互连方式主要有总线共享、交叉开关互连和片上网络 3 种。文中对多核处理器核间互连技术不做具体研究,默认任意 2 个处理器内核间均存在数据通信通路,即对于任意的 $s(0 \leqslant s \leqslant m-1)$ 和 $t(0 \leqslant s \leqslant m-1)$, $\text{Rate}_{s,t} \neq 0$。为简化环境模型,假设同一时刻,内核 P_k 只能与一个内核进行通信。

1.2 任务模型

任务模型是描述任务属性和特征的一种数学模型,它包含任务调度过程中的状态和控制等信息。STDS 算法构建了一个可以共享访问的 TDS(task data structure)结构,TDS 由若干数据项构成,每个数据项唯一对应一个任务,数据项记录该任务的相关信息,其定义如图 1 所示。

T_i	T_{is}	T_{id}	T_{idn}	T_{idd}	T_{ia}	T_{ip}	T_{it}	T_{ic}

图 1 数据项定义

Fig.1 Definition of data element

图 1 中,T_i 为任务编号,唯一标识任务; T_{is} 为任务状态,STDS 算法将任务状态分为等待、就绪和已执行 3 种,用 1 表示等待,0 表示就绪,2 表示已执行。如果任务处于就绪态,说明已分配除 CPU 外所有必要资源,允许调度程序将其调度分配;等待状态的任务因需要等待某一事件的发生,比如等待其前驱任务执行完毕,而暂不能被调度程序调度;任务执行完毕后的状态为已执行状态,此状态下,如果其他所有任务都不再需要此任务信息,就将其对应的数据项删除,以节省内存空间;

$T_{id} = \{\cdots, T_j, \cdots\}$: 依赖任务集合,也叫前驱任务集合,任务 T_i 必须在其依赖任务全部执行完成之后执行;

T_{idn}: 依赖任务剩余数量,表示任务 T_i 的依赖任务集合中尚未执行完毕的任务数量,即依赖任务集合 T_{id} 中,还有多少依赖任务没有执行完成。其初值是集合 T_{id} 的长度,即 $T_{idn} = |T_{id}|$, 每当一个依赖任务执行完毕, T_{idn} 值减 1, 直到 $T_{idn} = 0$, 而 $T_{idn} = 0$ 是 $T_{is} = 0$(就绪)的必要条件;

$T_{idd} = \{\cdots, \text{Data}_{i,j}, \cdots\}$: 与依赖任务通信数据量集合,记录任务 T_i 与其依赖任务 T_{id} 通信的数据大小, $\text{Data}_{i,j}$ 表示任务 T_i 与依赖任务 T_j 通信的数据大小,如果任务 T_i、T_j 分别分配到内核 P_s、P_t 上,则任务 T_i 与 T_j 的通信时间就可用 $\text{Data}_{i,j}/\text{Rate}_{s,t}$ 表示;

$T_{ia} = \{\cdots, T_j, \cdots\}$: 依赖任务 T_i 的任务集合,也叫后继任务集合,只有 T_i 执行完成后,其后继任务才有机会变为就绪状态,从而调度执行。当 T_i 完成后, 需要将集合 T_{ia} 中所有任务的 T_{idn} 属性值减一;

T_{ip}: 任务优先级,表示任务优先调度程度,任务优先级越高,被优先调度的可能性越大。任务优先级取决于任务就绪时间、通信开销以及内核负载等因素。系统运行过程中,就绪时间和内核负载等是动态变化的,所以任务优先级不是某一定值,它随着系统状态的改变而动态变化;

T_{it}: 任务就绪时间,即任务满足调度条件变为就绪状态的时间。如果用 t 表示现在时间,则等待时间就是$(t - T_{it})$, 为避免存在长时间未能调度执行的就绪任务,算法应优先调度等待时间长的任务;

T_{ic}: 任务映射,表示任务和内核的对应关系,即任务 T_i 分配到哪个内核上执行。STDS 算法用 -1 表示任务尚未分配内核资源,用整数 0,1,…,$m-1$ 分别表示在内核 P_0, P_1,…, P_{m-1} 上执行。

1.3 调度器模型

文中设计的调度器模型采用集中式调度模式,如图 2 所示。图中指定了一个内核专门执行调度程序,称为中心调度器,中心调度器负责收集调度信息和执行任务调度,其余内核只负责执行任务。

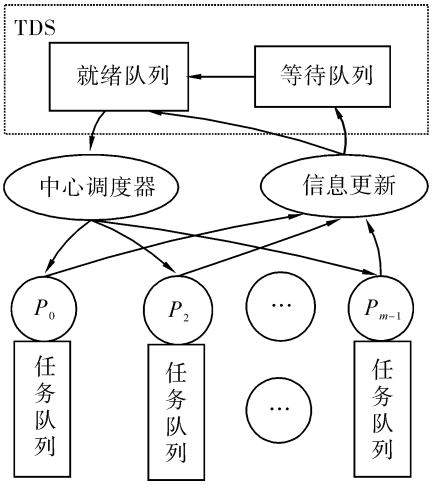

图 2 调度器模型

Fig.2 Scheduler model

建立维护等待队列和就绪队列 2 个全局任务列表,调度器在任务调度时只需要检索就绪任务队列,节省了调度时间开销,等待队列中的任务满足就绪条件时可以变为就绪态等待调度。每个内核都有一个可以缓存一定数目任务的局部队列,当内核需要调度任务时直接从缓存中获取;而当局部队列中的任务过少时开始请求调度器为局部队列分配任务,

降低了调度器调度频率,而且中心调度器与各内核可以并行地运行。全局调度队列和局部调度队列结合的方式在整体上是全局队列方式,与集中式调度策略配合使用易于实现负载均衡;在底层实现上是局部队列方式,降低了系统对队列共享性要求,克服了集中式调度模式的性能瓶颈问题。

文中通过重载和轻载阀值确定调度时机,当内核处于轻载状态时请求任务调度,处于重载时不允许再向内核分配任务,这种策略在保证一定负载均衡效果的同时提高了内核执行效率。

2　任务优先级

将任务分配到最合适的内核上是任务调度的核心问题,而任务优先级计算是任务分配的关键,任务优先级表明任务被优先调度程度。在多核环境下,各内核的状态不尽相同,STDS 算法为评价任务与内核的适合程度,首先计算任务相对于一个确定内核的优先级 T_{ipk},然后取其在所有内核上的最大值作为任务优先级 T_{ip},表示为式(1):

$$T_{ip} = \max_{0 \leqslant k \leqslant m-1} T_{ipk} \tag{1}$$

式中:m 为内核数量,T_{ipk} 表示任务 T_i 相对于内核 P_k 的优先级。STDS 算法在计算 T_{ipk} 时,综合考虑等待时间、任务间通信开销和内核负载状态因素。等待时间为当前时间与就绪时间的差值,任务间通信开销是任务 T_i 与其所有依赖任务 T_{id} 通信时间之和,而内核负载状态用于实现内核负载均衡。

多核处理器的负载均衡要求避免出现一个或多个内核负载较低甚至空闲,而其他内核处于高负载状态的情况,从而充分发挥多核处理器优势。STDS 算法使用任务队列长度 $|TL_k|$ 作为内核负载指标,内核的任务队列越长,说明分配的任务越多,所以其负载越大。

为实现负载均衡引入调度粒度,内核 P_k 的调度粒度定义为一次调度过程中为 P_k 分配的任务数量,这里的一次调度过程是指一个内核请求调度,调度算法为其分配任务的过程,在实际运行中,可能出现调度算法一次性处理多个内核调度请求,调度的任务数量等于为每个内核分配的任务数量之和。粒度大小要根据系统模型而定,调度粒度过大,不能充分发挥动态调度优势,而粒度过小,会引发频繁调度,增大调度程序运行时间开销,降低处理器效率。对于异构多核处理器,调度粒度与内核处理速度是正比关系。STDS 算法将调度粒度定义为式(2):

$$l_k = l \cdot \mathrm{sp}_k, 0 \leqslant k \leqslant m-1 \tag{2}$$

式中:l_k 表示内核 P_k 的调度粒度,l 表示粒度因子,sp_k 表示内核 P_k 的处理速度。对于一个实际的处理器系统,内核速度是确定的已知量,调度粒度 l_k 的大小,可以通过粒度因子 l 调节,粒度因子与具体的运行状况有关,它起到将内核的计算速度转换为内核调度任务数量功能。

STDS 算法通过限制内核任务队列长度实现负载均衡,为有效发挥调度队列作用,算法引入 2 个负载因子,分别确定队列的上限和下限,具体定义如式(3)~(5):

$$l_k\_\max = l_k(1 + \delta_1) \tag{3}$$

$$l_k\_\min = l_k(1 - \delta_2) \tag{4}$$

$$\delta_1 + \delta_2 = 1 \tag{5}$$

$$0 \leqslant k \leqslant m - 1$$

式中:δ_1 是负载上限因子,$l_k\_\max$ 表示内核 P_k 的任务队列长度上限,δ_2 是负载下限因子,$l_k\_\min$ 表示内核 P_k 的任务队列长度下限。当 $|Tl_k| \leqslant l_k\_\min$ 时,内核 P_k 请求调度程序分配任务,而当 $|Tl_k| \geqslant l_k\_\max$ 时,就不允许向内核 P_k 上分配任务,从而保证内核 P_k 的任务队列长度满足:$l_k\_\min \leqslant |Tl_k| \leqslant l_k\_\max$。需要说明的是,在一些极端情况下,比如当有多个内核同时请求分配任务,或者暂时没有任务可分配时,调度程序无法及时响应内核任务调度请求,而内核继续从其任务队列上取任务执行,此时会出现 $|Tl_k| < l_k\_\min$ 的情况,但只要调度粒度设置合理,这种情况不会频繁出现,所以不影响整体负载均衡。内核 P_k 的任务队列长度上限与下限之差 $l_k\_\max - l_k\_\min = l_k(\delta_1 + \delta_1)$,而式(5)设置约束条件 $\delta_1 + \delta_2 = 1$,从而使得 $l_k\_\max - l_k\_\min = l_k$,内核 P_k 在其任务队列长度为 $l_k\_\min$ 时请求分配任务,调度程序响应请求并为 P_k 分配 l_k(调度粒度)个任务,此时内核 P_k 的任务队列长度为其上限值 $l_k\_\max$。

上述阐述了计算任务优先级时的内核队列上下限问题,式(6)~(10)则是对任务优先级计算时的等待时间、通信开销和内核负载均衡因素的描述。STDS 算法在计算任务优先级时,综合考虑等待时间、通信开销和内核负载均衡因素,式(1)中 T_{ipk} 表示任务 T_i 分配到内核 P_k 适合程度,其计算公式为:

$$T_{ipk} = (\mathrm{PW}_i + \mathrm{PC}_{ik}) \cdot l_k j \tag{6}$$

$$\mathrm{PW}_i = \beta(t - T_{it}) \tag{7}$$

$$\mathrm{PC}_{ik} = \frac{\sum_{0 \leqslant s \leqslant m-1} C_{is}}{m \cdot C_{ik}} \tag{8}$$

$$l_k = \frac{l_{k\_\max} - |Tl_k|}{l_{k\_\max} - l_{k\_\min}} \tag{9}$$

$$C_{ik}=\sum_{j\in T_{idd}\&\&s\neq k}\frac{\mathrm{Data}_{i,j}}{\mathrm{Rate}_{s,k}} \tag{10}$$

$0\leqslant k\leqslant m-1,0\leqslant s\leqslant m-1,0\leqslant i<|T_{\mathrm{List}}|$

式中:PW_i代表任务 T_i的等待时间所占优先级比重,PC_{ik}代表通信时间所占优先级比重,L_k为内核 P_k的负载状态。t 是当前时间,$t-T_{it}$表示任务 T_i的等待时间,等待时间越长的任务优先级相对越高,同等条件下,调度程序优先调度等待时间长的任务,避免存在就绪任务长时间等待的"饥饿"现象,β 是实时性因子,用来调节等待时间在优先级中所占比重。C_{ik}表示任务 T_i的通信时间,STDS 算法假设同一内核内的任务通信开销忽略不计,C_{ik}值即任务 T_i与除内核 P_k以外的其他内核 P_s上依赖任务通信的时间之和,用 $\sum_{j\in T_{idd}\&\&s\neq k}\frac{\mathrm{Data}_{i,j}}{\mathrm{Rate}_{s,k}}$ 表示,下标 s 即任务 T_j的 T_{jc}属性值,$\frac{\sum_{0\leqslant s\leqslant m-1}C_{is}}{m}$ 表示任务 T_j分配给内核 P_s。$\frac{\sum_{0\leqslant s\leqslant m-1}C_{is}}{m}$ 表示平均通信时间,通信时间越小,PC_{ik}值越大,优先级也相应越大,对于 C_{ik}相同的任务,若任务的 $\sum_{0\leqslant s\leqslant m-1}C_{is}$ 值较大,则说明此任务如果分配到内核 P_k上节省的通信开销更多,相应的其优先级也较高。在实际计算时,因为任务 T_i的依赖任务已经全部分配执行,所以若分配在内核 P_k上,通信开销是个定值,因此 PC_{ik}只需计算一次。$\frac{l_{k_\max}-|TL_k|}{l_{k_\max}-l_{k_\min}}$ 反应了内核的负载状态,$|TL_k|$是内核 P_k当前任务队列长度,当 $|TL_k|=l_{k_\max}$时其值为 0,从而 $T_{ipk}=0$ 为最小值,则当前状态下不能向内核 P_k分派任务。在一次调度过程中,调度程序可能需要响应多个内核的调度请求,对于 $|TL_k|$较小的内核,其 L_k较大,从而实现优先将任务分配到负载较轻的内核,并进而达到负载均衡效果。

综上所述,任务优先级值的计算,是对等待时间、通信开销和内核负载 3 个因素综合考量的过程,其结果可能不是单一因素中最优的,但一定是综合考虑全部 3 种因素后最优的选择。

3 算法实现

在上节中详细介绍了任务优先级的计算,本节将介绍 STDS 算法的具体实现过程。

为了使任务调度算法得到较理想的实现,STDS 算法的研究基于以下假设条件:

1)假设任务间的依赖关系固定不变;

2)同一内核上任务间通信开销忽略不计;

3)任务不可抢占。

系统运行时首先将任务初始化,构造 TDS,由于算法中大多计算只针对就绪任务,特别是任务分配时,所以为了节约计算开销,算法初始化时会生成一个就绪任务列表和一个等待任务列表。当内核 P_k任务队列长度 $|Tl_k|\leqslant l_{k_min}$时,就向调度程序请求分配任务,调度程序遍历就绪列表,计算相对于内核 P_k的优先级,选择优先级 T_{ip}值最大的任务分配到 P_k上,重复执行此过程,直到内核 P_k任务队列长度 $|Tl_k|\geqslant l_k$。若同时有多个内核请求任务分配,调度程序分配任务时需要响应这几个内核的请求,计算优先级时需要计算相对于这几个内核的优先级,然后取最大值作为最终优先级,调度程序选择优先级最大的任务分配到对应的内核。

值得注意的是在动态调度过程中,为了保证数据实时有效,STDS 算法在每个任务分配完成或执行结束时对数据进行更新。当任务 T_i分配给内核 P_k后,需要将任务 T_i对应数据项中的 T_{ic}置为 k,此时内核 P_k负载也相应改变;当任务 T_i执行完成时,如果任务 T_j依赖任务 T_i,需要将任务 T_j的 T_{idn}值减 1,若减 1 后 $T_{idn}=0$,将任务 T_j状态置为就绪态并从等待列表移到就绪列表。为了降低复杂度节省时间,STDS 算法采用先记录再批量执行的方式,即在任务执行期间暂时将完成的任务编号记录,在任务调度程序执行时批量更新数据;如果任务 T_i执行完毕且所有依赖 T_i的任务也执行完毕,就将任务 T_i的信息从 TDS 中删除,从而释放内存节省空间。

如果就绪列表为空,调度程序不会响应内核的任务分配请求,当就绪列表和等待列表全为空时,说明任务全部执行完毕,调度结束。

STDS 算法实现步骤:

Procedure STDS(SM,TDS,P_{List},T_{Lexe})/ * SM 表示处理器内核系统;TDS 是所有任务信息;P_{List} 是需要分配任务的内核列表,T_{Lexe} 是执行完毕但还未进行信息更新的任务列表 * /

1){for each $T_i\in T_{\mathrm{Lexe}}$

2) { $T_{ia}[\,]\leftarrow T_i$;

3) for each $T_j\in T_{ia}$

4) $T_{jdn}\leftarrow T_{jdn}-1$; / * 执行完毕的任务信息更新,将其后继任务的依赖任务数量减一 * /

}/ * end for each $T_i\in T_{\mathrm{Lexe}}$ * /

5)$\mathrm{TL}_{\mathrm{ready}}[\,]\leftarrow$TDS;/ * 就绪任务放入就绪队列 * /

6) for each $T_i\in TL_{\mathrm{ready}}$

7){ for each $P_k\in P_{\mathrm{List}}$

8)　C_{ik}←式(10);/* 任务 T_i 若在内核 P_k 上与前驱任务的通信开销,只需计算一次 */

}/* end for each $T_i \in TL_{ready}$ */

9) while($TL_{ready} \neq \varnothing$ and $P_{List} \neq \varnothing$)

10) {　for each $T_i \in TL_{ready}$

11)　　{　PW_i←式(7);

12)　　for each $P_k \in P_{List}$

13)　　{　PC_{ik}←式(8);

14)　　L_k←式(9);

15)　　T_{ipk}←式(6);

}/* end for each $P_k \in P_{List}$ */

16)　　$T_{ip} = \max\{T_{ipk}\}$;

}/* end for each $T_i \in TL_{ready}$ */

17)　T_j←$\max\{T_{ip}\}$;/* 选出最大优先级的任务,设 T_j 就是选出的任务,且其对应内核为 P_s,即 $T_{jp} = T_{jps}$ */

18)　$TL_s \leftarrow TL_s + \{T_j\}$;/* 任务 T_j 分配给内核 P_s */

19)　$TL_{ready} \leftarrow TL_{ready} - \{T_j\}$;

20)　$T_{jc} = s$;

21)　if $|TL_s| = l_{s\_max}$ then $P_{List} \leftarrow P_{List} - \{P_s\}$;

/* 内核 P_s 分配任务结束 */

22)　if $TL_{ready} \neq \varnothing$ then return; /* 没有就绪任务 */

}/* end for while */

}

其中 1)~5)是算法对执行完毕但未进行数据更新的任务进行处理,将其后继任务的 T_{idn} 属性值减 1,且减为 0 时放入就绪队列等待调度。算法中有两重循环,外层循环遍历 T_{Lexe} 列表,内层循环遍历后继任务列表,所以时间复杂度为 $O(n) \times O(\log n) = O(n\log n)$。而任务更新发生在每次任务分配时,次数大约为任务数量与调度粒子比值即 $n/1$,而在具体应用环境中调度粒子为定值,所以数据更新总时间复杂度为 $O(n^2 \log n)$。6~20 是进行任务分配,其中优先级计算是关键。计算每个任务相对于每个请求分配任务内核的优先级,其时间复杂度为 $O(mn)$,其中 m 是内核数量,n 是就绪任务数量。而算法最终需要将任务全部分配完成,所以任务分配的总时间复杂度为 $O(mn) \times O(n) = O(mn^2)$。因此 STDS 算法的时间复杂度为 $O(n^2 \log n) + O(mn^2) = \text{Max}(O(n^2 \log n), O(mn^2))$,而经典 Min-Min 算法的时间复杂度为 $O(mn^2)$[13],两者处于同一量级。

4　实验

任务调度目前还没有形成统一规范的性能测试方案[14],大多数对调度算法的研究都是采用随机 DAG 图生成器来生成各种不同形状的 DAG 图集合作为任务调度算法的测试用例。

文中借助 TGFF 工具生成随机任务图,并以此完成 TDS 初始化,用内核上任务长度表示负载情况,通过任务总执行时间和负载均衡等衡量算法优劣,在 Simics[15]模拟平台上对本文算法性能进行模拟实验测试。为了测试本文提出的算法对并行任务调度问题的求解效果,采用下面 2 类实验来验证。

4.1　参数对调度算法的影响

STDS 算法用到的参数有:粒度因子 l、负载因子 δ_1 和 δ_2、实时性因子 β。l 的大小决定一次调度过程分配的任务数量,进而决定调度频度;l、δ_1 和 δ_2 共同影响着负载均衡;β 主要调节算法实时性。下面分别通过实验数据说明参数 l、δ_1、δ_2、β 对 STDS 算法性能的影响。

1)调度粒度对 STDS 算法调度频度和负载影响

通过 TGFF 工具分别生成具有 3 000 个任务和 5 000 个任务的随机任务图,测试这 2 类任务在不同调度粒度下的调度效果,为屏蔽其他因素影响,设置参数 $\beta = 0$, $l_k\_\min = l_k(1 - \delta_2) = 2$,结果如图 3 所示。由图 3 可以看出,调度次数随着调度粒度增大而减小,进而降低 STDS 调度算法消耗时间;粒度较小时调度次数减少幅度较大,而在较大粒度情况下效果较弱。

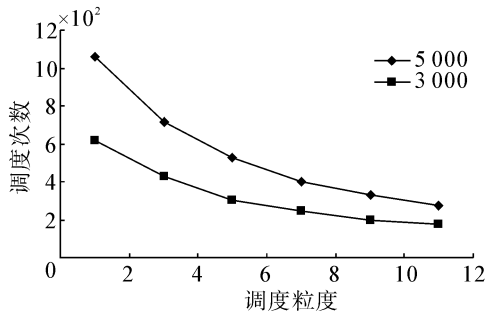

图 3　调度粒度对调度次数影响

Fig.3　Change of scheduling times in the situation of different scheduling granularities

虽然较大粒度下节省调度时间,但并不是 l 取最大值时 STDS 算法性能最优,l 取极大值时,调度频度最小,但此时对于内核 P_k 的请求,STDS

算法会将全部就绪任务分配给 P_k，从而造成各内核上负载差别较大。表 1 说明了粒度对负载均衡的影响，在调度过程中对内核负载进行采样，每次采样时，用此时内核上任务队列长度占所有内核任务队列长度之和的比例，评价负载均衡效果。表 1 中 P_0、P_1、P_2的处理速度为 1 GHz，P_3的处理速度为 2 GHz，据 STDS 算法设定，P_3的负载应为 P_0、P_1、P_2的 2 倍。从表 1 可以看出，负载均衡效果随调度粒度的增大而降低。

表 1 调度粒度对负载均衡影响

Table 1 Change of load balancing in the situation of different scheduling granularities

调度粒度	P_0/%	P_1/%	P_2/%	P_3/%
2	19.61	20.44	20.37	39.58
8	20.11	19.80	21.19	38.90
14	22.51	19.13	20.91	37.45

2）负载因子对算法性能影响

δ_1 是负载上限因子，δ_2 是负载下限因子，δ_1 和 δ_2 满足公式(5)约束条件：$\delta_1 + \delta_2 = 1$，当 δ_2 确定时，δ_1 的值也是确定的。δ_2 的大小决定了内核 P_k 调度队列长度下限 l_k_min 的大小，l_k_min 较大时，内核 P_k 的负载也相对较大，然而此时会引发另一个问题，虽然内核 P_k 负载达到下限 l_k_min 并请求分配任务，但此刻 P_k 依然有较多任务待执行，当就绪任务不是很充足时，容易发生 STDS 算法频繁响应分配请求。图 4 是在 δ_2 取不同值时的调度统计结果，固定调度粒度 l=6，可以看出，调度次数随着 δ_2 的增大而增大。

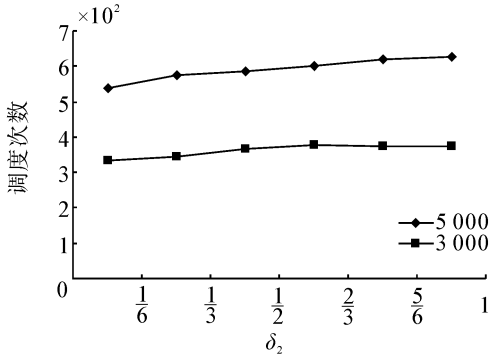

图 4 负载因子对调度次数影响

Fig.4 Change of scheduling times in the situation of different load factor

3）实时性因子对算法性能影响

STDS 算法设立实时性因子主要是解决就绪任务长时间得不到内核资源的“饥饿”现象。由式(6)可知，任务优先级是任务等待时间、任务间通信和内核负载 3 种因素共同决定的，通过实时性因子可以调节等待时间因素的重要程度，实时性因子反映了系统对等待时间的容忍程度。表 2 是在 β 取不同值时对 STDS 算法平均等待时间和总运行时间的影响，其中 AWT(average of wait time)表示平均等待时间，此时设置 l=6，δ_1=1/3，δ_2=2/3 以消除其他因素影响。由表 2 可以看出，随着 β 值的增大，等待时间最大的 100 个任务的平均等待时间减小，而全部任务的平均等待时间的趋势不能确定，因为此时通信开销不是最小的，所以算法总执行时间增大，从而会造成某些任务的等待时间增大。在 β 取较大值(>2.0)时，STDS 算法性能基本不再变化，此时 STDS 算法近似于先来先服务(FIFO)算法。所以 β 应取满足系统对实时性要求前提下的最小值。

表 2 实时性因子对算法性能影响

Table 2 Change of performance in the situation of different real-time factors

β	AWT of Top 100/ms	AWT of All/ms	运行时间/ms
0	7 603.11	689.446	32 562
0.4	6 752.75	1 084.344	34 250
0.8	6 179.30	901.595	34 922
1.0	4 838.11	882.158	35 125
2.0	4 517.32	911.572	35 756
5.0	4 497.84	926.485	35 998

综上所述，参数会对算法的调度时间、调度次数、负载均衡效果以及实时性产生影响，对于不同应用环境，应适当调整参数大小，在满足系统要求前提下充分发挥算法性能。对于实时性要求不严格系统，可以减少实时因子大小，降低总调度时间；对于规模较大系统，应增大调度粒度在减小调度次数，尽量减少调度器的调度压力，适当增大负载下限因子，保证在调度器调度不及时情况下，内核依然有充足的任务可以执行；反之亦然。本文没有明确参数值的具体大小，此工作需要通过大量具体的实际实验完成。

4.2 算法对比实验

为有效评价 STDS 算法性能，将其与 Min-Min 算法和文献[11]的 Vinay 算法进行比较。对于每个固定的内核数，通过 TGFF 工具随机产生 10 组5 000 节点的 DAG 图，记录其调度执行时间，然后求出其平均值。由于内核数目改变，系统模型发生变化，所以实验时适当调节 STDS 算法参数，以充分发挥算法性能，表 3 是不同内核数目时 STDS 算法参数取值，内核数目较多时，增大调度粒度 l，虽然负载均衡

效果有所下降,但降低了调度次数。图 5 是 STDS 算法、Min-Min 和 Vinay 算法在不同内核数目下执行时间的比较结果。

由图 5 可以看出在内核数目较少时,Vinay 算法比较高效,这是因为 Vinay 算法实现简单,算法复杂度低,每个内核分配一个任务,不需要考虑负载均衡等问题。随着内核数目增加,STDS 算法优势逐步体现,这是因为 Vinay 算法一次调度分配一个任务,调度较频繁,内核数目较多时,会发生多个内核同时请求调度,但同一时刻只有一个内核可以调度成功,其他内核不得不等待。STDS 算法为每个内核设置一个任务队列,每次调度会分配若干个任务,这样虽然增大了算法复杂度,但减少了调度次数且最大限度避免了内核等待现象,所以总时间较小。STDS 算法比较适合内核数目较多的系统,但值得注意的是在内核数目较多的系统中,对任务量的要求也较大,否则难以充分发挥多核系统和 STDS 算法性能。

表 3　不同内核数目时的参数值

Table 3　Value of parameters in the situation of different quantity of cores

内核数目	l	δ_1	δ_2	β
2	3	1/3	2/3	0.1
4	4	1/2	1/2	0.1
8	6	1/3	2/3	0.1
16	6	1/2	1/2	0.1
32	10	3/10	7/10	0.1

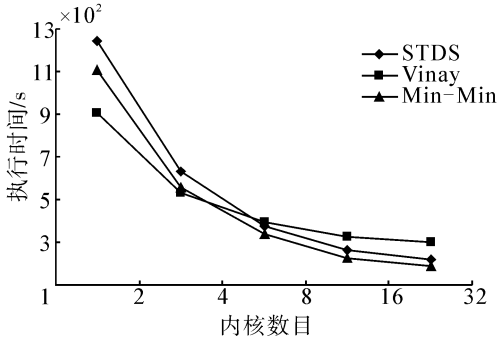

图 5　STDS、Min-Min、Vinay 算法执行时间

Fig.5　Comparison of running time among STDS, Min-Min andVinay

STDS 算法复杂度与 Min-Min 算法相当,但调度运行时间比 Min-Min 算法稍长,原因是 Min-Min 算法将运行时间作为其唯一考虑因素,优先选取最早完成时间最小的任务进行调度。但 Min-Min 算法中执行时间较长的任务得不到及时执行,图 6 是 STDS 算法与 Min-Min 算法等待时间的对比结果。

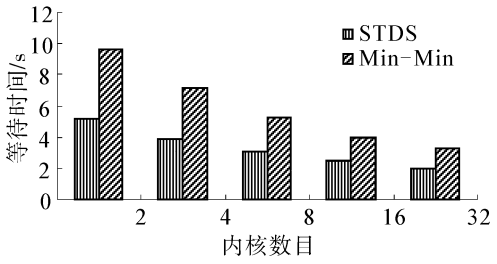

图 6　STDS 算法与 Min-Min 算法等待时间对

Fig.6　Comparison of waiting time between STDS and Min-Min

可以看出 STDS 算法中任务最大等待时间明显小于 Min-Min 算法。由此也可以得出 STDS 算法的另一大优势,就是可以通过调整参数取值,从而适应不同的系统要求。为了排除偶然因素影响,本文通过生成另外 9 组不同结构的 3 000 个节点与 5 000 个节点的 DAG 图,进行相同的实验,实验结果与上述描述吻合。

5　结束语

文中研究了异构多核动态任务调度算法,结合设计的调度模型,提出了动态任务调度 STDS 算法。该算法分析了调度粒度对任务调度的影响可以根据当前内核负载和任务状态调整任务优先级,从而将高优先级的任务合理高效地分配到内核,并通过限定内核任务队列的上下限值达到负载均衡的目的。通过引入实时性因子使任务总能及时得到执行处理,减少了过长的等待时间。实验表明,STDS 算法在内核较大情况下效率较高,且能够保持负载均衡。

参考文献:

[1] GENG X, XU G, ZHANG Y. Dynamic load balancing scheduling model based on multi-core processor[C]//2010 Fifth International Conference on Frontier of Computer Science and Technology (FCST). [S.l.],2010: 398-403.

[2] GRAY M R, JOHNSON D S. Computers and intractability: a guide to the theory of NP-completeness[M]. New York: W H Freeman and Company, 1979: 92-115.

[3] JIN H, CHEN H, CHEN J, et al. Real-time strategy and practice in service grid[C]//Proceedings of the 28th Annual International on Computer Software and Applications Conference, 2004. 2004: 161-166.

[4] HE X S, SUN X H, VON LASZEWSKI G . A QoS guided scheduling algorithm for grid computing[J]. Journal of Computer Science and Technology, 2003,18(4): 442-451.

[5] FREUND R F, GHERRITY M, AMBROSIUS S, et al. Scheduling resources in multi-user, heterogeneous, compu-

ting environments with SmartNet[C]//Proceedings on Heterogeneous Computing Workshop. , 1998: 184-199.

[6]ARMSTRONG R, HENSGEN D, KIDD T. The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions[C]//Proceedings on Heterogeneous Computing Workshop. , 1998: 79-87.

[7]石威, 郑纬民. 相关任务图的均衡动态关键路径调度算法[J]. 计算机学报, 2001, 24(9): 991-997.
SHI Wei, ZHENG Weimin. The balanced dynamic critical path scheduling algorithm of dependent task graph[J]. Chinese Journal of Computers, 2001, 24(9): 991-997.

[8]李仁发, 刘彦, 徐成. 多处理器片上系统任务调度研究进展评述[J]. 计算机研究与发展, 2008, 45(9): 1620-1629.
LI Renfa, LIU Yan, XU Cheng. A survey of task scheduling research progress on multiprocessor system-on-chip [J]. Journal of Computer Research and Development, 2008, 45(9): 1620-1629.

[9]耿晓中. 基于多核分布式环境下的任务调度关键技术研究[D]. 吉林: 吉林大学, 2013: .
GENG Xiaozhong. Research on key techniques of task scheduling based on multi-core distributed environment[D]. Jilin: Jilin University, 2013.

[10]徐雨明,李肯立. 异构系统中 DAG 任务调度的双螺旋结构遗传算法[J]. 计算机研究与发展,2014,51(6): 1240-1252.
XU Yuming, LI Kenli. A structure genetic algorithm for task scheduling double-helix on heterogeneous computing systems [J]. Journal of Computer Research and Development, 2014, 51(6): 1240-1252.

[11]VAIDYA V G, RANADIVE P, SAH S. Dynamic scheduler for multi-core systems[C]// International Conference on Software Technology and Engineering, 2010 2nd. Puerto Rico: IEEE, 2010, 1: V1-13-V1-16.

[12]陈锐忠,齐德昱,林伟伟.一种面向非对称多核处理器的综合性调度算法. 软件学报, 2013, 24(2): 34-3357.
CHEN Ruozhong, QI Deyu, LIN Weiwei. Comprehensive scheduling algorithm for asymmetric multi-core processors [J]. Journal of Software, 2013, 24(2): 34-3357.

[12]邓晓衡, 卢锡城, 王怀民. iVCE 中基于可信评价的资源调度研究[J]. 计算机学报, 2007, 30: 1750-1762.
DENG Xiaoheng, LU Xicheng, WANG Huaimin. Study on trust evaluation based resource scheduling in iVCE[J]. Chinese Journal of Computers, 2007, 30: 1750-1762.

[12]KWOK Y K, AHMAD I. Benchmarking the task graph scheduling algorithms[C]//Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International... and Symposium on Parallel and Distributed Processing 1998. IEEE, 1998: 531-537.

[12]FORSGREN D, ESKILSON J, CHRISTENSSON M, et al. Simics:a full system simulation platform[J]. IEEE Computer, 2002, 35(2): 50-58.

作者简介:

刘正, 男,1983 年生,讲师,博士研究生,主要研究方向为性能优化、信息安全与智能信息处理。