计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2016.20148445
53(6):1271-1280,2016

基于排队理论的动态任务调度模型及容错

何王全[1] 魏 迪[1] 权建校[1] 吴 伟[1] 漆锋滨[2]
[1](江南计算技术研究所 江苏无锡 214083)
[2](国家并行计算机工程技术研究中心 北京 100080)
(wangquan_he@163.com)

Dynamic Task Scheduling Model and Fault-Tolerant via Queuing Theory

He Wangquan[1], Wei Di[1], Quan Jianxiao[1], Wu Wei[1], and Qi Fengbin[2]
[1](*Jiangnan Institute of Computing Technology*, *Wuxi*, *Jiangsu* 214083)
[2](*National Research Center of Parallel Computer Engineering & Technology*, *Beijing* 100080)

Abstract The design of efficient dynamic task scheduling and fault-tolerant mechanism is an issue of crucial importance in high-performance computing field. Most existing methods, however, can hardly achieve good scalability on large-scale system. In this paper, we propose a scalable dynamic task scheduling model via *N*-level queuing theory, which dramatically reduces the programming burden by providing programmer with concise parallel programming framework. On one hand, we utilize the Poisson process theory to analyze the average wait time of tasks, and then decide the task layers according to threshold. On the other hand, we reduce the fault tolerance overhead using region-aware light-weight degradation model. Experimental results with Micro Benchmark on Bluelight system with 32 768 cores show that our method achieves good scalability when the tasks take 3.4 s on average and the overhead is just 7.2% of traditional model. Running on 16 384 cores, pharmacological application DOCK achieves performance improvement by 34.3% with our scheduling. Moreover, the results of DOCK show our fault-tolerant model achieves 3.75%～5.13% performance improvements over traditional mechanism.

Key words queuing theory; dynamic task scheduling; programming framework; fault-tolerant; light-weight degradation

摘 要 高效的动态任务调度和容错机制是高性能计算面临的挑战之一,已有的方法难以高效扩展到大规模环境.针对该问题,提出了基于 *N* 层排队理论的高可扩展动态任务调度模型,为程序员提供简洁的并行编程框架,有效降低了编程负担;使用泊松过程相关理论分析了任务申请的平均等待时间,通过给定的阈值进行决策分层;结合局部感知的轻量级降级模型,可有效降低大规模并行课题的容错开销,提高系统的可用性.Micro Benchmark 在神威蓝光 32 768 核环境下测试表明,对于平均执行时间为 3.4 s 的短任务,基于 *N* 层排队理论的动态任务调度模型可扩展性很好,调度开销是传统模型的 7.2%;药物软件 DOCK 在 16 384 核环境下的整体性能比该软件原有的任务调度提升 34.3%;局部感知的轻量级降级模型具有故障后损失小的特点,DOCK 的测试表明比传统容错方法执行时间减少 3.75%～5.13%.

收稿日期:2014-12-30;**修回日期**:2015-08-17
基金项目:国家"八六三"高技术研究发展计划基金项目(2012AA010903);计算机体系结构国家重点实验室基金项目(CARCH201403)
This work was supported by the National High Technology Research and Development Program of China (863 Program) (2012AA010903) and the Foundation of State Key Laboratory of Computer Architecture (CARCH201403).

关键词 排队理论;动态任务调度;编程框架;容错;轻量级降级

中图法分类号 TP391

近年来,高性能计算技术发展迅猛,高端并行系统的规模日益庞大,为大规模并行应用课题的高效解算奠定了坚实基础.高性能计算系统可提供强大的计算能力,但其规模和复杂性给并行应用的高效运行带来了极大的挑战,主要体现在可扩展性和容错2个方面.

大规模并行应用可分为数据并行和任务并行2大类.数据并行应用通常根据拥有者计算的原则由数据分布产生计算划分,其基本思想是让左值拥有者进行计算,以减少非本地引用,降低通信开销[1];任务并行应用是本文研究工作的主要对象,它通常将课题分解成众多子任务,对数据集进行分割,通过将任务和对应数据加载到不同计算资源上并行执行[2].任务并行应用广泛存在于药物筛选、基因研究、密码分析、核模拟等领域,多数应用的子任务间无相关性,但子任务的计算量可能存在显著差异,在大规模环境下,高效的负载平衡机制是保证应用性能的关键之一.

高端的并行系统资源数量庞大、设计复杂,很难保证长时间内不出现故障,对于需要运行数日乃至数周的大规模并行应用来说,低开销的容错机制能够降低故障对并行应用的影响,对提升并行应用的健壮性、性能和系统的可用率都具有重要意义.

1 国内外研究现状

1.1 动态任务调度研究现状

动态任务调度是实现负载平衡的重要手段,当今国内外高性能计算领域动态任务调度方法的研究工作主要包含3种类型:

1) 嵌入式.嵌入式调度方法根据应用的特点,在应用程序中实现相应的动态任务调度机制.这种方法需要程序员在精通并行应用特点的基础上,额外开发相应的任务调度模块,以实现应用程序在目标平台上高效的负载平衡[3-4].该方法通用性较差,在大规模环境下,通常需要针对不同应用设计实现相应的动态任务调度模块,程序员开发负担较重.

2) 过程迁移式.针对嵌入式调度方法用户负担较重的缺点,业界又提出了对用户完全透明的过程迁移式调度方法,不需要修改应用代码.当调度系统发现负载不平衡时,就将负载较重进程的任务和数据迁移至负载较轻的进程.这方面的研究工作主要基于Charm++[5]展开,以Menon等人[6]实现的GrapevineLB系统以及Zheng等人[7]提出的层次式调度方式为典型代表.

过程迁移式调度方式的缺点在于需要交换负载信息和任务迁移,开销大,不适合大规模系统.

3) 框架式.框架式调度由并行语言或并行库提供任务调度服务,其设计目标是将任务调度从用户程序中分离出来,将复杂的调度工作交给系统软件,保证任务调度方法的通用性,用户只需对代码进行少量的修改就可以实现任务调度.

早期的框架式的任务调度以UPC语言[8-9]为代表,采用循环语法扩充的方式基于数据亲缘性进行任务调度,但存在负载不平衡的问题.

Kumar等人[10]提出的DLBL(dynamic load balancing library)采用面向迭代的调度策略,通过集中仲裁的方式,将迭代所需的数据在进程间迁移以获得良好的负载平衡.Devine等人[11]开发Zoltan系统通过回调函数的形式和用户程序进行交互,并预设了多种负载平衡策略,用户可以自由尝试多种策略,然后根据优化效果选择最优策略.Zhang等人[12]实现的GLB(lifeline-based global load balancing library)在X10语言[13]的基础上提供对用户透明的任务调度服务,用户只需要定义诸如处理算法、任务分割方式、任务归并方式、结果处理方式等要素,在GLB内部,当发现任务执行完毕时,采用Work-Stealing[14]的方式从其他进程窃取任务. Zhang等人[15]实现的AME(anyscale many-task computing engine)以及Krieder等人[16]实现的GeMTC(GPU-enabled many-task computing),主要面向MTC(many-task computing)应用,以应对该类课题在计算资源调度、任务相关性处理、负载平衡、数据管理以及容错等方面对高性能计算系统提出的挑战,AME在计算资源调度、任务相关性处理以及数据管理3个方面做了很好的工作;GeMTC面向具有GPU加速部件的高性能计算平台,用户通过其提供的API接口,实现向加速器高效加载计算任务.Xiao等人[17]针对MTC应用提出了一种应用级的优先级调度算法,给出了针对异构计算环境的粗粒度调度方案,首先分析应用特点将作业分配到不同的资源上,然后在运行时根据作业的负载动态调整

其优先级.

框架式调度只需要用户少量修改程序,但多数已有的框架式调度在大规模环境下,面临可扩展性挑战.

1.2 轻量级容错研究现状

对大型应用来说,需要相应的容错措施来保证程序的健康高效运行,大规模环境下的容错技术是近年来高性能计算领域的研究热点之一.

保留恢复[18]是最常见的容错模型,以检查点为基础.程序正常执行过程中,以一定的间隔在主存或磁盘中记录程序执行的内存映像以及消息日志;当硬件资源出现故障时,通过上述2个要素恢复到距离当前最近的一个检查点重新执行.当前具有代表性的研究成果主要有BLCR[18](Berkeley lab's Linux checkpoint/restart)和SCR[19](scalable checkpoint/restart).BLCR只提供单节点系统级的保留恢复支持,可作为并行运行时系统容错功能开发的基础.SCR的基本思路则是基于简洁API集合提供用户级的保留恢复支持,专注于检查点文件的快速存储以及故障发生时作业的快速恢复,从而保证容错功能的可扩展性.

虽然当今业界对于保留恢复容错模型的研究依然是热点,但是时空开销较大的缺点也极大限制了其应用前景.尤洪涛等人[20]针对任务并行课题提出了一种低开销的降级容错模型:当有个别计算资源出现故障时,丢弃故障资源并回收相关子任务,作业运行不被中断,最终保证所有的子任务均被正确执行.该方法的缺点是有故障时所有进程都要进行处理,性能影响较大.

不难发现,高性能计算领域对动态任务调度方法和容错技术的研究尚属2个独立的研究领域.本文在可扩展动态任务调度架构的基础上提出了轻量级降级容错模型,保证并行程序以较小的容错代价在计算资源出现故障时仍可稳定运行.

2 基于N层排队理论的可扩展动态任务调度模型

对于子任务计算时间不均衡的应用,采用动态任务调度是解决负载平衡问题的有效方法,最常见的执行模型为Master-Slave模型,如图1所示.在该模型中,多个Slave向Master申请任务.

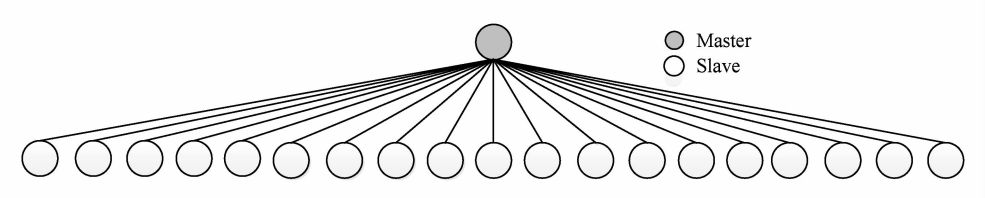

Fig. 1 Classic Master-Slave model.

图1 传统的Master-Slave模型

Master-Slave模型的通信模式为典型的多对一通信,在大规模环境下(进程数量达到数万以上),可扩展性是该模型面临的主要问题.针对该问题,本文提出了基于N层排队理论的动态任务调度模型,并设计了简洁的并行编程框架,可有效降低程序员的编程负担,提高可扩展性,同时在容错方面具有明显的优势.

2.1 N层排队动态任务调度模型

多级Master-Slave模型采用层次式资源分配方法,设置Region-Master(以下简称R-M),每个R-M向上一层申请任务和报告任务完成,并向下一层提供任务动态调度服务,以缓解多对一通信瓶颈问题.4层Master-Slave的调度模型如图2所示,位于顶层的灰色小圈代表Master,位于最下方的白色小圈代表Slave,中间层条形小圈和格子小圈代表不同层次的R-M,计算资源向上级申请任务并报告完成情况,Master管理全局任务池.为了进一步提高任务分配的性能,根据资源数量和实际应用中子任务的执行情况,采用排队论[21]的思想决定模型层数和R-M的数量.

对于各子任务执行时间均衡的应用,通常采用静态任务调度策略(这类应用不适合本文的模型).而对于各子任务执行时间不均衡的应用,在大规模环境下适合采用多级Master-Slave动态调度模型,该模型的每一层都是一个经典的Master-Slave模型,可以用排队理论描述,Master看作是服务窗口,Slave看作是顾客,Slave向Master申请任务看作是到窗口排队等待服务.泊松分布[22]常用于描述在任意一段固定的时间间隔内,到某公共设施要求给予服务的顾客数量.Master-Slave模型中任务申请是一个时间连续、状态离散的过程,下面论述采用泊松分布来描述任务申请的随机性是合理的.

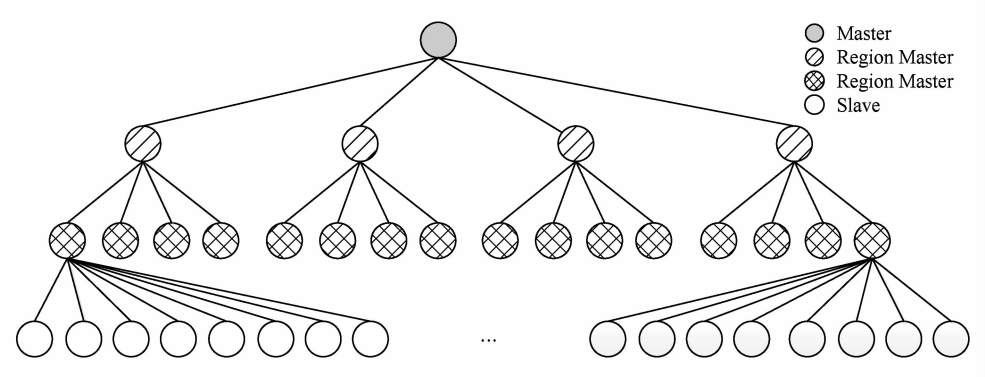

Fig. 2 Four layer Master-Slave model.

图 2 4 层 Master-Slave 模型示意图

定义 1[23]. 若计数过程$\{\xi(t), t\geqslant 0\}$满足下列条件,则称为具有参数$\lambda(\lambda>0)$的泊松过程.

1) $\xi(0)=0$;

2) $\xi(t)$是独立、平稳的增量过程;

3) $\xi(t)$满足:

① 时间区间$[t, t+\Delta t)$内发生 1 次的概率与 Δt 成正比,即 $P\{\xi(t+\Delta t)-\xi(t)=1\}=\lambda\times\Delta t+o(\Delta t)$;

② 时间区间$[t, t+\Delta t)$内发生 2 次以上的概率是 Δt 的高阶无穷小,即 $P\{\xi(t+\Delta t)-\xi(t)\geqslant 2\}=o(\Delta t)$.

任务并行类应用在 0 单位时间内不会出现任务请求,满足定义 1 的条件 1;由任务并行应用的特性可知,子任务间没有相关性,在不重叠的时间间隔内 Slave 向 Master 提交的任务请求数量是相互独立的,没有后效性,因此满足定义 1 的条件 2;Master-Slave 模型中任务申请过程存在网络延迟等因素,在充分小的时间间隔内最多有 1 个任务申请到达,不会或者以极小概率有 2 个或者 2 个以上的任务申请同时到达,而在区间$[t, t+\Delta t)$内有 1 个任务请求到达的概率与时间 t 无关,而与区间长度 Δt 成正比,即满足定义 1 的条件 3. 因此 Master-Slave 模型任务申请满足参数为 λ 的泊松过程,即在$[0, t)$时间内达到 k 个任务请求的概率为

$$P_k(t)=\frac{(\lambda t)^k}{k!}\mathrm{e}^{-\lambda t}.$$

传统的 Master-Slave 模型可以近似地用"M/D/1/∞/P/FCFS"排队模型[21]来描述,其中 P 是进程个数(Master 除分配任务外,自身也参与计算). 当 Slave(顾客)的数量较大,Master 作为服务窗口可能成为性能瓶颈. 当传统的 Master-Slave 模型无法满足动态任务调度的性能要求时,需要调整服务模型的层次,使 W 个进程作为 R-M,由 R-M 向 Master 申请任务和报告任务状态,并同时为下层提供任务动态调度服务,W 取值约为下一层孩子个数的平方根. Master 和 W 个 R-M 可以看作是"M/D/1/∞/W/FCFS",每个 R-M 和大约 P/W 个 Slave 之间可以看作"M/D/1/∞/(P/W)/FCFS"的排队模型. 如果一个 R-M 管理的 Slave 仍过多,难以满足性能要求时进行递归拆分,直到满足要求为止. 本文提出的动态任务调度模型将是基于 N 层排队理论的多级 Master-Slave 模型,每一级看作"M/D/1/∞/W/FCFS"的排队模型. 根据并行应用所使用的资源规模及并行任务的执行时间特性,通常采用 2 层(传统模型)、3 层或 4 层模型,决策的流程如下:

Step1. 使用传统的 Master-Slave 模型进行初始阶段的任务分配;

Step2. Master 对排队情况进行采样统计,采样个数 M 根据各进程第 1 个任务的执行时间来确定;

Step3. 根据采样结果,采用极大似然法拟合确定课题任务申请过程中的排队特性(获取泊松分布的期望 λ 和方差 σ^2);

Step4. 根据任务申请的排队特性,决策动态任务分配模型的层数.

在中小规模的系统上,采用传统的 Master-Slave 2 层模型可以满足要求;在 Peta Flops 级别的大规模系统上,需要采用 3 层模型;而未来的 Exascale 级别的系统,计算核心数量空前庞大,需要采用 4 层以上模型. 基于 N 层排队理论的动态任务调度模型由 2.3 节的并行任务调度框架自动实现,对应用层透明,即应用层不需要关心复杂的 N 层实现细节,由系统实现并行任务调度的高效可扩展,自动适应各种规模的并行环境.

2.2 模型性能分析

首先讨论模型参数的确定. 作业提交后,首先采用传统的 Master-Slave 模型进行调度,在初始化阶段

Master 主动给每个 Slave 分发 1 个任务,之后就进入任务的自由申请阶段,各 Slave 在完成第一个任务后,主动向 Master 报告任务的完成,并申请新的任务.在任务自由申请阶段,令单位时间内任务申请到达的个数为一个实验样本 ξ_i,选取连续的 M 个样本,采用泊松分布的极大似然估计法[22],可获得期望和方差的估计量:

$$\hat{\lambda}=\frac{1}{M}\sum_{i=1}^{M}\xi_i,$$

$$\hat{\sigma}^2=\frac{1}{M}\sum_{i=1}^{M}(\xi_i-\hat{\lambda})^2.$$

动态任务调度的主要目标是要使计算资源的负载比较平衡,以及 Slave 的任务请求开销 T_{req} 占用的比例尽量小.定义距离 Slave 最近的 R-M 或 Master 为该 Slave 的 Parent,Slave 的任务请求开销 T_{req} 包含任务请求的网上传输时间 T_{msg}、请求到达 Parent 后的排队等待时间 T_{wait}、因 Parent 本地任务池为空时向更上层请求任务的时间 $T_{\text{top\_req}}$、Parent 从本地任务池中分配任务的时间 T_{dispose} 以及任务请求响应的网上传输时间 T_{msg},即:

$$T_{\text{req}}=T_{\text{msg}}+T_{\text{wait}}+T_{\text{top\_req}}+T_{\text{dispose}}+T_{\text{msg}}=2\times T_{\text{msg}}+T_{\text{wait}}+T_{\text{top\_req}}+T_{\text{dispose}}.$$

在 N 层模型的实现中,R-M 采用了预取等优化策略,$T_{\text{top\_req}}$ 的开销占的比例很小,几乎可以忽略,因此 T_{req} 可以近似地表示为

$$T_{\text{req}}\approx 2\times T_{\text{msg}}+T_{\text{wait}}+T_{\text{dispose}}.$$

在给定的系统中,任务请求或响应的网络传输延迟是确定的,即 T_{msg} 可认为是一个常量;T_{dispose} 与管理维护本地任务池有关,取决于 CPU 的计算能力,近似地认为是常量.因此要减少 Slave 的任务请求开销,主要是减少 T_{wait}.为了方便表达,记 Parent 单位时间内可处理请求的个数为 μ,由泊松分布的性质可知,单位时间内到达的任务请求个数的期望值为 λ,令:

$$\rho=\frac{\lambda}{\mu}.$$

当 $\rho>1$ 时,系统的服务能力不足,不能达到稳态,任务分配开销占的比例大,此时,系统必须从 N 层变为 $N+1$ 层以减少任务等待时间 T_{wait}.

当 $\rho\leqslant 1$ 时,系统可到达稳态,文献[22]给出了任务平均等待时间 T_{wait} 可表示为

$$T_{\text{wait}}=\frac{\lambda}{\mu(\mu-\lambda)}.$$

令 T_{exe} 是单个任务的平均执行时间,每个任务可能不同,完全由课题的特征决定,与负载平衡无关.综合上述,可计算出任务分配占作业执行时间的百分比 $percent_{\text{req}}$:

$$percent_{\text{req}}=\frac{T_{\text{req}}}{T_{\text{req}}+T_{\text{exe}}}.$$

对于给定的调度开销阈值 C,若 $percent_{\text{req}}>C$,则说明当前的动态任务调度的开销过大,需要进行递归分层.

2.3 动态任务调度并行编程框架

在很多应用中,并行任务的动态调度由程序员使用消息接口编写,在大规模环境下,简单的 Master-Slave 模型可扩展性差,高效实现动态任务调度对普通程序员来说极具挑战性.为此,本文提出了一种动态任务调度并行编程框架如下所示,适合子任务间无相关性的应用:

```
/* 并行任务调度编程框架 */
while ((task_id=get_task_id(任务总量,检查点
    文件名,通信子))≥0)
{
  do_job(task_id); /* 用户代码 */
}
```

该编程框架以 *get_task_id*()原语的形式提供给程序员,*get_task_id*()返回值代表任务的编号或结束标志(小于 0 表示任务分配结束),检查点文件记录已完成的任务,通信子指出动态任务调度的范围.动态任务调度并行编程框架在软件栈的层次如图 3 所示:

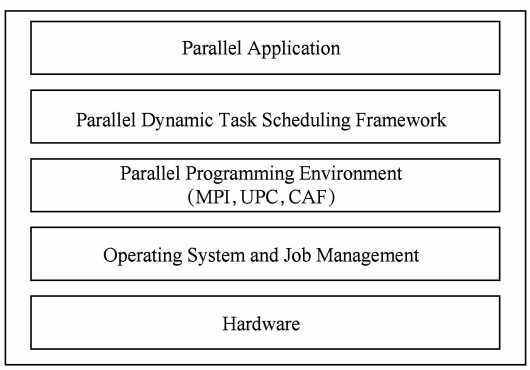

Fig. 3 Parallel dynamic task scheduling framework in the software stack.

图 3 并行任务动态调度编程框架在软件栈中的层次

采用该框架编程非常简洁,复杂的调度工作、检查点均由框架自动完成.框架实现中,所有的进程均参与计算,Master 和 R-M 处理任务请求由轮询线程实现,因此不会带来明显的计算资源损耗.框架中采用全局任务编号代表具体的任务,此方式具有 2 个

特点:1)通信量少,任务申请和完成报告需要使用消息传递的数据非常少,可有效减少系统开销;2)通用性,任务总量确定的任务并行应用均可采用.在实际应用中,并行任务通过函数 f 映射为一个整数;并行任务调度时,计算资源取到任务编号后,再根据任务编号由 f^{-1} 还原出具体的任务.

f:第 i 个任务 $\mapsto$ 任务编号 i,

f^{-1}:任务编号 $i \mapsto$ 第 i 个任务.

f 和 f^{-1} 的规则完全由程序员来定义,操作较为简单.对于单个原始任务的运行时间极短的课题,可以将多个原始任务打包映射为一个任务,以在大规模环境下获得良好的性能.

3 结合 N 层动态任务调度模型的轻量级容错

降级模型如图 4 所示,它将应用程序分为课题初始化、子任务并行计算、资源重构和结果处理 3 个阶段.阶段 2 是程序运行的主体,可以采用降级的方式进行容错,当有计算资源故障时,在线隔离故障资源并回收故障节点的任务,重新分配给正常的节点进行计算.

尤洪涛等人[20]提出的降级模型在计算资源出现故障时,需要中断所有进程的执行并等待容错完成.我们对降级模型进行了改进,实现了局部感知的轻量级容错,当有故障发生时,只需要通知少量相关进程,其他进程的执行不受影响.

3.1 局部感知的轻量级降级模型

在基于 N 层排队的动态任务调度模型中,经过逻辑划分的每个 Region 实际上是一个相对独立的任务调度子系统,R-M 能够掌握所辖区域内进程的任务分配情况.结合动态任务调度模型,我们提出了基于局部故障感知技术的轻量级降级模型,将故障的影响有效控制在较小范围内.

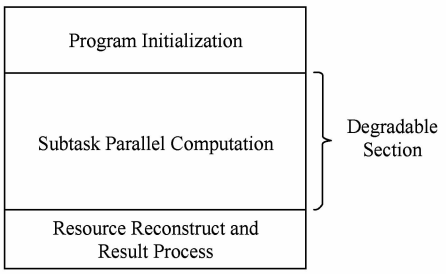

Fig. 4 Fault-tolerant model of degradation.

图 4 降级容错模型

结合 3 层动态任务调度的轻量级容错模型可以用图 5 描述,在降级区内计算资源发生故障后,相应的容错措施可以分为 3 类:

1) 当系统检测到 Slave 出现故障,只需通知 Region 内的所有进程,故障隔离后,由 R-M 回收已分配给故障资源但尚未完成的任务,并重新分配给健康资源计算,如图 5(a)所示;

2) 当系统检测到 R-M 出现故障,只需通知 Master 和 R-M,故障隔离后,Region 内重新选举新的 R-M,由 Master 回收已分配给故障资源但尚未完成的任务,并重新分配给健康资源计算,如图 5(b)所示;

3) 当系统检测到 Master 出现故障,需通知所有的 R-M 和 Master 所在 Region 内的进程,故障隔离后,重新选举新的 Master 和 R-M,由新 Master 回收已分配给故障资源但尚未完成的任务,并重新分配给健康资源计算,如图 5(c)所示.

当子任务并行计算完毕后,需要进行资源的重构,完成作业的后续处理工作.

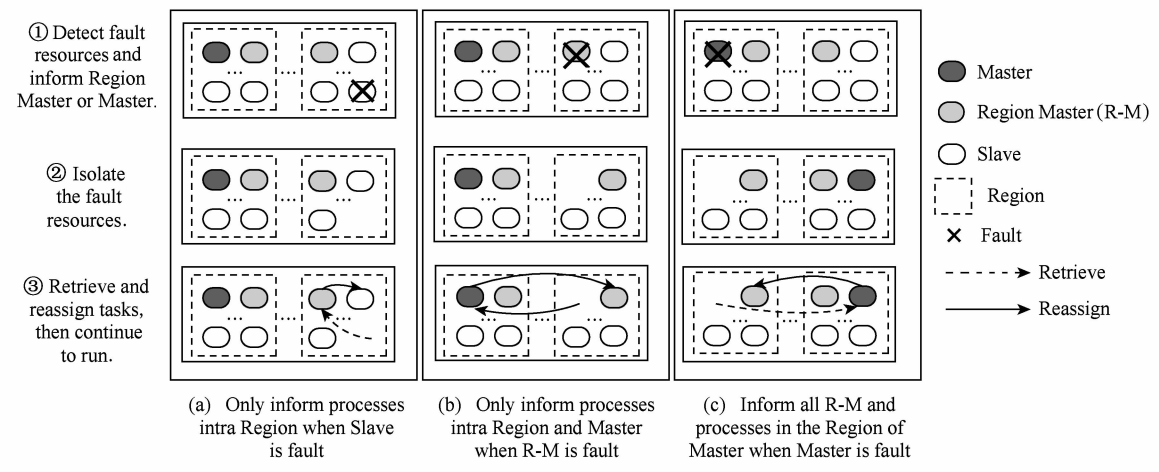

Fig. 5 Light-weight degradation model with dynamic task scheduling.

图 5 结合动态任务调度的轻量级降级模型

3.2 轻量级降级模型性能分析

本文提出的轻量级降级模型,与故障后采用整个作业回卷的方式相比,作业的损失明显要小.假设作业由 P 个进程执行,共有 n 个子任务,执行过程中共发生了 g 次故障,采用降级模型的损失为

$$L_1 = \sum_{i=1}^{g}(D_i \times A_i + R_i \times N_i),$$

其中,D_i 是第 i 次降级的处理时间,A_i 是第 i 次降级处理影响的进程个数,R_i 是第 i 次降级后作业继续运行的时间,N_i 是第 i 次降级减少的进程数量.

故障发生后,若采用回卷的容错方式,损失的期望值为

$$L_2 = \sum_{i=1}^{g}(\text{正在执行任务的损失} + \text{作业退出时间} + \text{重新提交时间} + \text{初始化时间}) \approx \sum_{i=1}^{g}\left(\frac{\text{单个任务的平均执行时间}}{2} \times P + T_{\text{quit}} + T_{\text{sub}} + T_{\text{init}}\right) \approx g \times \left(\frac{\sum_{i=1}^{n} T_{\text{exe}}^{i}}{2 \times n} \times P + T_{\text{quit}} + T_{\text{sub}} + T_{\text{init}}\right).$$

对大规模环境来说,$T_{\text{quit}} + T_{\text{sub}} + T_{\text{init}}$ 的时间比较长,回卷会影响作业中所有计算资源执行,因此 L_1 会明显小于 L_2.

采用轻量级的降级模型可以进行实时资源调配.当进程数为 P 的作业正在运行,需要调配出 Q 个进程的计算资源供其他作业使用时,只需要采用软件措施标记拟调配的资源为"故障"状态,可以在不停止作业的情况下划走需要的资源,损失的期望值为 L_3.采用先停止作业再划走资源的方式,损失的期望值为 L_4.

$$L_3 = \frac{\sum_{i=1}^{n} T_{\text{exe}}^{i}}{2 \times n} \times Q,$$

$$L_4 = \frac{\sum_{i=1}^{n} T_{\text{exe}}^{i}}{2 \times n} \times P + T_{\text{quit}} + T_{\text{adj}} + T_{\text{sub}} + T_{\text{init}},$$

其中,T_{adj} 是管理员调整资源的时间.

通常情况下,P 通常是 Q 的数倍,因此 L_3 明显比 L_4 小,从理论上来看采用降级模型具有明显的优势.

4 实验结果

为了验证本文提出的动态任务调度方法和轻量级容错技术的有效性,利用国家超算济南中心的神威蓝光计算机系统进行了大规模测试,该系统每个节点由一颗申威-1600 16 核 CPU 构成(运行频率 1 GHz),配备 8 GB 内存,运行 Linux 操作系统,节点间采用 Infiniband QDR 网络连接.我们使用了神威蓝光计算机系统 32 768 核的计算资源进行测试.

4.1 Micro Benchmark 测试

为了方便地验证本文模型的有效性,我们编写了 Micro Benchmark,并在神威蓝光 32 768 核的环境下进行了验证.该 Micro Benchmark 的并行任务执行时间限定在[L,$L+S$](单位 s)的范围内,由随机数产生,任务之间相互独立,平均每个核执行 100 个任务.

表 1 提供的测试数据可知,对 2 种不同计算时长的子任务进行了测试,其中子任务计算时间 2~5 s(平均 3.4 s)属于极短任务,采用传统 Master-Slave 模型,任务调度的开销占执行时间的 50%以上.采用本文的模型,选取调度开销的阈值 $C=10\%$,根据模型的分析和决策,使用 3 层实现,R-M 的数量取 128,任务调度开销占测试程序执行时间的 6.88%,调度开销仅为传统模型的 7.2%;AME[15] 在 16 384 核环境下平均执行 4 s 的任务,调度开销超过 10%.子任务计算时间为 60~180 s 属于中等任务,采用传统 Master-Slave 模型,调度开销为 2.81%,本文模型的开销仅为 0.32%.

在任务自由申请阶段,设 10 ms 内任务申请到达的个数为一个实验样本 ξ_i,选取连续样本若干个,采用泊松分布的极大似然估计法,能够得到参数 λ 的估计量 $\hat{\lambda}$ 以及方差的估计量 $\hat{\sigma}^2$,如表 1 所示,理论方差与实际统计方差相差小于 10%,说明用排队理论描述我们的模型是合理、有效的.

Table 1 Estimate Value of λ, σ^2, and Time Costs for Different Task-size

表 1 不同规模任务的 λ、σ^2 估计以及模型开销对比

Scale	Average Time of Single Task/s	Sampling Interval/ms	Number of Samples	$\hat{\lambda}$	$\hat{\sigma}^2$	Percentage of Scheduling of Textual Model/%	Percentage of Scheduling of Master-Slave/%
Tasks with 2~5 s ($L=2$,$S=3$)	3.4	10	1 000	97	104.9	6.88	53.01
Tasks with 60~80 s ($L=60$,$S=120$)	119.2	10	20 000	3	3.11	0.32	2.81

图 6 给出了任务执行时间为 2～5 s 的情况下 N 层排队模型与传统 Master-Slave 模型的可扩展性对比. 测试数据表明,采用传统 Master-Slave 模型,任务调度开销随着进程数的增加上升非常快,说明 Master 是明显的热点,到 32 768 进程时任务申请已经成为主要开销. 采用 N 层排队模型,可以有效规避热点,从 1 024 进程到 32 768 进程任务调度开销增加不明显,表明该方法可以有效扩展到大规模环境.

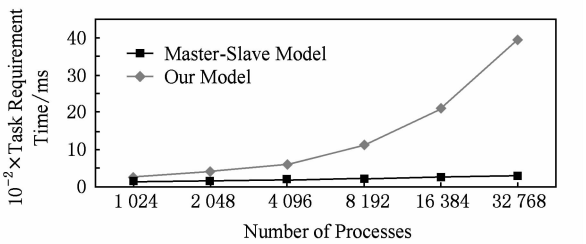

Fig. 6 Comparison of task requirement time costs for 2～5 s tasks.

图 6 2～5 s 极短随机任务的任务申请开销对比

4.2 实际应用测试结果

我们对实际应用 DOCK[24] 进行了测试. DOCK 是药物设计领域应用十分广泛的分子对接计算模拟软件,已经成为药物发现的核心工具之一,该软件采用嵌入式的动态任务调度方法. 原始 DOCK 程序的结果回收由主进程承担,规模较大时主进程成为瓶颈,在测试之前我们对结果处理进行了优化.

测试算例使用了 24 万分子规模的化合物数据库与疾病靶标进行分子对接模拟,每个靶标与分子的相互作用能(它们之间的自由能)计算就是一个计算任务,课题总共需要计算 24 万次,单个任务的计算时间 14～801 s,平均 204 s.

仍然取调度开销的阈值 $C=10\%$,表 2 给出了并行任务调度的测试结果. 从表 2 看出,在 1 024 进程下,本文提出的并行框架采用 2 层调度模型实现,性能比原嵌入式调度方法提高 4.9%,主要得益于网上传送的消息量少;在 16 384 进程下,本文提出的并行框架经决策采用 3 层调度模型,R-M 的数量为 128,缓解了 Master 端的压力,比原嵌入式调度方法提高 34.3%. 课题从 1 024 扩展到 16 384 进程的加速比如图 7 所示,DOCK 原始的嵌入式任务调度方法加速比为 12.37,采用本文的调度方法加速比达到了 15.84,接近线性. 根据理论分析可以预测,在更大的环境下采用多层调度模型将有更大的优势.

Table 2 Test Result of DOCK Application

表 2 DOCK 应用的测试结果

Number of Processes	Time Consumption of Original Task Scheduling/s	Time Consumption of Textual Model/s	Speedup Ratio
1 024	50 181.2	47 837.2	1.049
2 048	26 163.9	24 406.7	1.072
4 096	13 671.2	12 141.4	1.126
8 192	7 339.0	6 055.3	1.212
16 384	4 055.9	3 020.0	1.343

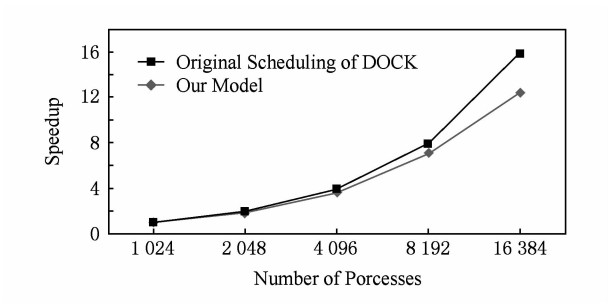

Fig. 7 The speedup of DOCK application.

图 7 DOCK 应用的加速比

表 3 给出了 16 384 进程下局部感知的降级效果,并与故障后回卷执行的时间进行了对比,运行过程中的故障均采用手工制造的方式产生. Slave 故障的处理开销是 5.2 ms,影响 127 进程,相比无故障执行,运行时间增加了 11.8 s;R-M 故障的处理开销是 61.1 ms,影响 127 进程(未出故障的 R-M 和 Master),相比无故障执行,运行时间增加了 19.6 s;Master 故障的处理时间较长,处理开销是 6.5 s,影响影响 127 进程(所有的 R-M),总执行时间增加了 55.5 s. 采用故障后回卷执行,比无故障运行的结果增加了 175.6 s(任何一个点故障对回卷来说是同等的,因此回卷只测试了一次),其中正在执行的任务大约损失 100 s,课题的中止、课题的重新提交和应用初始化的时间之和大约是 75 s. 从测试数据看,局部感知的降级措施相对于故障后回卷,课题执行时间减少 3.75%～5.13%. 若单个任务的执行时间长,回卷执行的损失更大,局部感知的降级措施的优势将更为明显.

表 4 给出了 16 384 进程环境、作业正在执行情况下,动态划走 4 096 个进程计算资源的开销对比. 划走计算资源,通常需要停止作业,完成资源整理后再重新提交作业,本作业所有计算资源上正在执行的任务将重新执行,在大规模环境下,作业停止时间、资源整理时间、作业重新提交的初始化时间都不

短.采用本文的降级模型,可以进行人工造错,在作业不停的情况下划走需要的资源,主要损失是被划走资源上正执行的任务需重新执行,其他进程的执行不受影响.

Table 3 Fault-tolerant Test for 16 384 Processes (Artificial Fault when Executing 1 500 s)

表 3 16 384 进程环境下的容错测试(程序执行 1 500 s 时人工造错)

Type of Fault-tolerant	Light-weight Degradation Model/s	Roll-back Model/s	Percentage Reduction of Degradation vs Roll-back/%
Slave Fault	3 031.8	3 195.6	5.13
R-M Fault	3 039.6	3 195.6	4.88
Master Fault	3 075.5	3 195.6	3.75

Table 4 Comparison of Time Costs between Two Methods when the Resources are Changed

表 4 资源变动情况下 2 种方法的开销对比测试

Execution Time with Light-weight Degradation Model/s	Execution Time with Resubmission after Job Stopped and Resources Changed/s	Percentage Reduction/%
3 056.7	3 226.3	5.26

5 结束语

本文针对大规模环境下并行任务动态调度的可扩展性和容错问题,提出了基于 N 层排队理论的高可扩展动态任务调度模型和方法.测试数据表明,该模型可以有效扩展到大规模环境,相比传统的 Master-Slave 模型具有明显的优势,可以满足 Peta Flops 级别系统下的应用需求,并可以推广到未来的 Exascale 级别的系统;配套的并行编程框架能有效减轻程序员的负担,并将任务调度产生的消息量通信开销降至最低.

结合高可扩展动态任务调度方法,提出了基于局部感知技术的优化降级模型,当发生故障时只影响部分进程的执行,有效降低容错开销.测试数据表明,与传统的容错方法相比具有较为明显的优势.

目前调度模型的分层是根据前 M 个采样来决策的,下一步我们拟在运行过程中对模型的分层决策进行动态调整.

致谢 我们向对本文工作给予指导和帮助的江南计算技术研究所的龚道永、宋长明、李祖华等人表示衷心地感谢!

参 考 文 献

[1] Wang Yiran, Chen Li, Feng Xiaobing, et al. Global partial replicate computation partitioning [J]. Journal of Computer Research and Development, 2006, 43(12): 2158-2165 (in Chinese)
(王轶然,陈莉,冯晓兵,等. 全局部分重复计算划分[J]. 计算机研究与发展, 2006, 43(12): 2158-2165)

[2] Wang Lei, Cui Huimin, Chen Li, et al. Research on task parallel programming model [J]. Journal of Software, 2013, 24(1): 77-90 (in Chinese)
(王蕾,崔慧敏,陈莉,等. 任务并行编程模型研究与进展[J]. 软件学报, 2013, 24(1): 77-90)

[3] Streitz F H, Glosli J N, Patel M V, et al. 100+TFlop solidification simulations on BlueGene/L [EB/OL]. [2014-11-02]. http://sc05.supercomp.org/schedule/pdf/pap307.pdf

[4] Koziar C, Reilein R, Runger G. Load imbalance aspects in atmosphere simulations [J]. International Journal of Computational Science and Engineering, 2005, 1(2): 215-225

[5] Kale L V. CHARM++: A portable concurrent object oriented system based on C++[C] //Proc of OOPSLA 1993. New York: ACM, 1993: 91-108

[6] Menon H, Kalé L. A distributed dynamic load balancer for iterative applications [C] //Proc of IEEE/ACM SC13. New York: ACM, 2013: 1-11

[7] Zheng G, Meneses E, Bhatele A, et al. Hierarchical load balancing for Charm++ applications on large supercomputers [C] //Proc of the 39th Int Conf on Parallel Processing Workshops. Los Alamitos, CA: IEEE Computer Society, 2010: 436-444

[8] LBNL, UC Berkeley. Berkeley UPC-Unified Parallel C [EB/OL]. [2014-11-02]. http://upc.lbl.gov

[9] Chen Li, Huo Wei, Lu Xingjing, et al. Parallel programming languages on multi-core and many-core architectures [J]. Information Technology Letter, 2012, 10(1): 23-40 (in Chinese)
(陈莉,霍伟,卢兴敬,等. 多核/众核系统上的并行编程语言[J]. 信息技术快报, 2012, 10(1): 23-40)

[10] Kumar R, Tullsen D M, Ranganathan P, et al. Single-ISA heterogeneous multi-core architectures for multi-threaded workload performance [J]. Isca Proc of Annual International Symposium on Computer Architecture, 2004, 32(2): 64

[11] Devine K, Boman E, Heaphy R, et al. Zoltan data management services for parallel dynamic applications [J]. Computing in Science & Engineering, 2002, 4(2): 90-96

[12] Zhang W, Tardieu O, Grove D, et al. GLB: Lifeline-based global load balancing library in X10 [C] //Proc of the 1st Workshop on Parallel Programming for Analytics Applications. New York: ACM, 2014: 31-40

[13] Tardieu O, Herta B, Cunningham D, et al. X10 at Petascale [C] //Proc of the 17th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming. New York: ACM, 2012: 267-276

[14] Berger E, Browne J C. Scalable load distribution and load balancing for dynamic parallel programs [EB/OL]. [2014-11-02]. http://web. engr. illinois. edu/~lumetta/wcbc99/wcbc-99-beb. pdf

[15] Zhang Z, Katz D S, Ripeanu M, et al. AME: An anyscale many-task computing engine [C] //Proc of the 6th Workshop on Workflows in Support of Large-Scale Science. New York: ACM, 2011: 137-146

[16] Krieder S J, Wozniak J M, Armstrong T, et al. Design and evaluation of the GeMTC framework for GPU-enabled many-task computing [C] //Proc of HPDC 2014. New York: ACM, 2014: 153-164

[17] Xiao J, Zhang Y, Chen S, et al. An application-level scheduling with task bundling approach for many-task computing in heterogeneous environments [C] //Proc of the 9th IFIP Int Conf. Berlin: Springer, 2012: 1-13

[18] Hargrove P H, Duell J C. Berkeley lab checkpoint/restart (blcr) for Linux clusters [J]. Journal of Physics: Conference Series, 2006, 46(1): 494-499

[19] Moody A, Bronevetsky G, Mohror K, et al. Design, modeling, and evaluation of a scalable multi-level checkpointing system [C] //Proc of IEEE/ACM SC'10. Piscataway, NJ: IEEE, 2010: 1-11

[20] You Hongtao, Jiang Xiaocheng, Chen Zuoning. Design of degrade based on dynamic job assignment [J]. Microcomputer Information, 2006, 22(30): 72-75 (in Chinese)
(尤洪涛, 姜小成, 陈左宁. 基于动态任务划分的降级机制[J]. 微计算机信息, 2006, 22(30): 72-75)

[21] Tang Yinghui, Tang Xiaowo. Basis and Analysis Technology of Queuing Theory [M]. Beijing: Science Press, 2006 (in Chinese)
(唐应辉, 唐小我. 排队论基础与分析技术[M]. 北京: 科学出版社, 2006)

[22] Department of Mathematics and Mechanics at Zhongshan University. Probability Theory and Mathematical Statistics [M]. Beijing: Higher Education Press, 1985 (in Chinese)
(中山大学数学力学系. 概率论及数理统计[M]. 北京: 高等教育出版社, 1985)

[23] Liu Cihua. Stochastic Processes [M]. 2nd ed. Wuhan: Huazhong University of Science and Technology Press, 2005 (in Chinese)
(刘次华. 随机过程[M]. 2 版. 武汉: 华中科技大学出版社, 2005)

[24] UCSF. The official UCSF DOCK Web-site: DOCK 6 [EB/OL]. [2014-11-02]. http://dock. compbio. ucsf. edu/DOCK_6/index. htm

He Wangquan, born in 1975. PhD candidate and senior engineer. Member of China Computer Federation. His main research interests include parallel programming language design, compiler optimization and runtime system.

Wei Di, born in 1984. Master and engineer. His main research interests include parallel progamming language design, runtime system and comunication system design (dididi888@chinaren. com).

Quan Jianxiao, born in 1983. Master and engineer. His main research interests include parallel progamming language design, complier optimization and runtime system (brightsky2007@163. com).

Wu Wei, born in 1984. Master and engineer. His main research interests include compiler design, compiler optimization, and parallel programming (ww7tc@sina. com).

Qi Fengbin, born in 1966. Senior engineer and PhD supervisor. Senior member of China Computer Federation. His main research interests include high performance computing architecture, compiler optimization and parallel algorithm (qifb116@sina. com).