软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2016,27(7):1789−1804 [doi: 10.13328/j.cnki.jos.004842]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

一种支持容错的任务并行程序设计模型*

王一拙, 陈 旭, 计卫星, 苏 岩, 王小军, 石 峰

(北京理工大学 计算机学院,北京 100081)

通讯作者: 王一拙, E-mail: frankwyz@bit.edu.cn, http://cs.bit.edu.cn/wangyizhuo

摘 要: 任务并行程序设计模型已成为并行程序设计的主流,其通过发掘任务并行性来提高并行计算机的系统性能.提出一种支持容错的任务并行程序设计模型,将容错技术融入到任务并行程序设计模型中,在保证性能的同时提高系统可靠性.该模型以任务为调度、执行、错误检测与恢复的基本单位,在应用级实现容错支持.采用一种 Buffer-Commit 计算模型支持瞬时错误的检测与恢复;采用应用级无盘检查点实现节点故障类型永久错误的恢复;采用一种支持容错的工作窃取任务调度策略获得动态负载均衡.实验结果表明,该模型以较低的性能开销提供了对硬件错误的容错支持.

关键词: 并行程序设计;容错;任务并行;工作窃取调度;负载均衡

中图法分类号: TP312

中文引用格式: 王一拙,陈旭,计卫星,苏岩,王小军,石峰.一种支持容错的任务并行程序设计模型.软件学报,2016,27(7):1789−1804. http://www.jos.org.cn/1000-9825/4842.htm

英文引用格式: Wang YZ, Chen X, Ji WX, Su Y, Wang XJ, Shi F. Task-Based parallel programming model supporting fault tolerance. Ruan Jian Xue Bao/Journal of Software, 2016,27(7):1789−1804 (in Chinese). http://www.jos.org.cn/1000-9825/4842.htm

Task-Based Parallel Programming Model Supporting Fault Tolerance

WANG Yi-Zhuo, CHEN Xu, JI Wei-Xing, SU Yan, WANG Xiao-Jun, SHI Feng

(School of Computer Science, Beijing Institute of Technology, Beijing 100081, China)

Abstract: Task-Based parallel programming model has become the mainstream parallel programming model to improve the performance of parallel computer systems by exploiting task parallelism. This paper presents a novel task-based parallel programming model which supports hardware fault tolerance. This model incorporates fault tolerance mechanisms into the task-based parallel programming model and aim to improve system performance and reliability. It uses task as the basic unit of scheduling, execution, fault detection and recovery, and supports fault tolerance in the application level. A buffer-commit computation model is used for transient fault tolerance and application-level diskless checkpointing technique is employed for permanent fault tolerance. A work-stealing scheduling scheme supporting fault tolerance is adopted to achieve dynamic load balancing. Experimental results show that the proposed model provides hardware fault tolerance with low performance overhead.

Key words: parallel programming; fault tolerance; task parallelism; work-stealing scheduling; load balancing

计算机系统的硬件体系结构已进入一个并行化的时代,共享存储系统上多核、众核微处理器正得到广泛应用,分布式存储系统上包含的并行处理节点数量越来越多,结构也越来越复杂.硬件体系结构上的并行化发展趋势带来两个显著问题:(1) 如何让软件设计充分发掘硬件的并行处理能力,从而提高系统的性能;(2) 在系统硬件规模不断扩大、复杂度越来越高的情况下,如何保证系统的可靠性.

* 基金项目: 国家自然科学基金(61300011)
Foundation item: National Natural Science Foundation of China (61300011)
收稿时间: 2014-12-31; 修改时间: 2015-03-02; 采用时间: 2015-03-31; jos 在线出版时间: 2015-12-30
CNKI 网络优先出版: 2015-12-30 17:12:42, http://www.cnki.net/kcms/detail/11.2560.TP.20151230.1712.003.html

上述第 1 个问题催生了并行程序设计模型的研究.并行程序设计模型是并行程序从设计到运行的整个软件体系结构的抽象,体现为支持并行程序开发的语言、工具和运行库等,如 OpenMP,MPI,Intel TBB[1],IBM X10[2],Cilk[3]等.并行程序可在指令、循环、任务、线程等不同粒度级别上发掘并行性.指令和循环级的并行性通常在编译优化中加以讨论,任务和线程粒度的并行性则是应用层并行程序设计关注的重点.近年来,任务并行程序设计模型已成为并行程序设计模型的主流[4],如 TBB,X10,Cilk,TPL[5],OpenMP 3.0 等.任务是逻辑上可独立分配的程序执行单元,能用来表示不同的并行模式(数据并行、任务并行、流式并行),这使其与线程相比具有更好的灵活性,更适用于并行程序的设计.

上述第 2 个问题使得容错技术得到越来越多的关注.系统规模的扩大和复杂度的提高,使发生硬件错误的几率不断上升,因此在系统设计中,需要提供针对硬件错误的容错机制来保证系统的可靠性,这在国防、金融、航空航天等领域尤为重要.硬件错误是指因环境因素、物理因素等造成的系统中硬件单元的错误或故障,硬件错误可分为 3 类[6]:瞬时错误(transient fault)、永久错误(permanent fault)和间歇错误(intermittent fault).宇宙射线、电磁干扰等可能引起集成电路器件逻辑状态的翻转,包括单事件翻转(SEU)、多位翻转(MBU)等,这类错误称为瞬时错误,在空间环境中最为常见,是影响航天器可靠性的主要因素之一;电路老化、制造误差、物理损伤等可能造成某个硬件单元的彻底失效,此类错误称为永久错误;间歇错误是指在一段时间内频繁发生的错误,通常是在本身不太稳定的系统上(如老化的系统)由电压或温度波动等引起.本文针对上述几种硬件错误,研究软件实现的容错技术,相比硬件实现的容错技术,软件实现具有成本低、可移植性好、应用灵活等优点.

并行程序设计的发展给容错技术带来了新的机遇和挑战,例如错误检测过程中的冗余执行,在多核时代可由不同处理单元上的工作线程并行完成,与单核时代的串行冗余执行相比大大降低了容错的时间开销,但同时也带来了同步和数据竞争等问题.虽然目前已有一些基于软件的容错技术(这里指对硬件错误的容错技术),但由于错误检测和恢复等的粒度普遍较大,因此已有技术对硬件资源的并行性发掘不够充分.本文研究一种支持容错的任务并行程序设计模型 FT-TPP(fault tolerant task-based parallel programming model),将对上述各种硬件错误的容忍能力融入到基于细粒度任务的并行程序设计模型中,在应用层,以任务为单位进行错误检测与恢复,通过支持容错的工作窃取任务调度等技术充分发掘任务并行性,降低容错开销,同时保证系统的性能和可靠性.

FT-TPP 的主要特点有:

1) 以任务为调度执行、错误检测和恢复的基本单元;
2) 支持硬件错误的检测和恢复,对瞬时错误采用任务粒度的冗余执行和比较进行错误检测,采用线程独立的 Buffer-Commit 机制支持错误恢复,对永久错误采用心跳机制进行检测,采用应用级无盘检查点技术实现错误恢复;
3) 纯软件模型,错误的检测、恢复和任务调度等都由软件实现,无需额外的硬件模块.

1 相关工作

1.1 容错技术

容错技术主要包括 3 个方面:错误检测(fault detection)、错误诊断(fault diagnosis)和错误恢复(fault recovery).错误诊断是对错误的准确定位和类型分析,不在本文的研究范畴.

1.1.1 错误检测

硬件上的瞬时错误可能发生在数据的存储过程或者处理过程中,存储中的瞬时错误通常通过错误校验码(ECC)进行检测甚至恢复,相关技术比较成熟,已得到工业化应用.处理过程的瞬时错误检测主要依赖于重复执行与比较,相关技术通常被称为基于复制的(replication based)错误检测.处理过程被看作位于一个复制域(sphere of replication,简称 SoR)[7]之中,进入 SoR 时,数据和指令被复制成不同的副本,在 SoR 中,不同副本的指令和数据独立执行,并产生各自独立的输出,在离开 SoR 时,对不同副本的输出数据进行比较,如果存在差异,则检测到错误.SoR 是系统中错误检测的逻辑边界,SoR 之外的存储器需要通过 ECC 等技术保证数据的正确性.

处理过程中的瞬时错误检测技术可按照复制对象分为下面几类.

1) 指令级容错:该类方法在编译时对原程序中的指令进行复制,并在适当位置,如存储和分支指令处,插入比较指令来检测错误,EDDI[8]和 SWIFT[9]是这类技术的典型代表;
2) 线程级容错:针对 SMT 和 CMP 微处理器体系结构,在两个硬件线程或内核上执行同一个程序的两个拷贝,并在处理器中加入特定缓存用于存放两个线程的执行结果,通过比较执行结果来检测错误,这类技术有 AR-SMT[10],SRT[7]和 CRT[11]等;
3) 应用级容错:此类方法在较高的软件层次上进行复制和比较,如将一个进程复制成多个冗余进程并发执行,然后在程序输出相关的系统调用等位置进行比较和同步,PLR[12]是这类方法的代表.

本文的瞬时错误检测属于应用级容错,对任务进行冗余执行和比较.

永久错误的检测技术可分为两类:一类是微体系结构层的硬件模块故障检测技术,常用于可重构处理器设计中;另一类是分布式系统中节点故障的检测,本文只针对此类永久错误.

1.1.2 错误恢复

错误恢复技术可分为两类:向前错误恢复(forward error recovery,简称 FER)和向后错误恢复(backward error recovery,简称 BER).

- FER 检测到错误后不需要回滚到出错时刻之前的状态重新执行,而是在当前时刻设法更正错误并继续向前执行.冗余是实现 FER 的基本途径,三模冗余(TMR)[13]是一种被广泛应用的 FER 技术,TMR 用 3 个模块执行相同的操作,然后在输出端通过一个多数表决器对数据进行选择,以实现容错的目的;
- BER 在检测到错误后回到错误发生之前的状态重新执行,根据实现方式的不同,BER 技术可分为检查点(checkpointing)和消息日志(message-logging)[14]两种.检查点技术根据保存检查点的内容,可分为系统级检查点和应用级检查点技术[15];根据存储检查点的介质,又可分为基于磁盘的和无盘检查点技术[16].本文对永久错误采用应用级无盘检查点来支持错误恢复.

当然,错误检测和恢复技术不是相互独立的,很多时候是融合在一起的,如,基于算法的容错技术(ABFT)[17]通过专门的算法设计,同时实现错误检测与恢复.

1.2 并行程序设计模型

并行程序设计模型根据其面向的系统体系结构不同,可分为针对共享存储、分布式存储、GPGPU 的模型、PGAS 以及混合模型.针对共享存储系统的并行程序设计模型包括 PThreads 和 OpenMP 等,其并行计算过程用多线程来实现,TBB 和 TPL 等基于任务的并行库也都是将任务调度到多线程上实现并行执行.针对分布式存储系统的并行程序设计模型主要通过消息传递实现,MPI 已成为这方面的工业标准.另外,对大规模的数据并行应用,MapReduce[18]编程模型近年来被广泛应用.针对 GPGPU,目前最常用的编程模型是 CUDA[19]和 OpenCL[20].另外,微软 DirectX 和 Intel ArBB[21]也都提供了适合 GPGPU 的并行编程工具.PGAS 模型及其延伸 APGAS 将共享存储和分布式存储统一起来考虑,给程序员提供一个虚拟的统一的地址空间,典型实现有 UPC[22],X10,Chapel[23]等.针对其他复杂的系统结构,可采用上述混合编程模型,如多核集群系统可采用 MPI+OpenMP 或 MPI+PThreads 等混合编程模型来发掘节点间以及节点内多处理核间两个层次的并行性.

根据实现方式的不同,并行程序设计模型也可分为如下 3 类.

(1) 新的并行编程语言,如 Cilk 以及许多类似于 Cilk 的系统、IBM 的 X10 等.当然,这些语言本身也不是从零开始设计的,也都借助于已有的成熟语言实现,如,Cilk 借助 C 和 Fortran 语言实现,X10 基于 Java 实现;

(2) 现有语言的并行扩展,如 OpenMP 采用编译制导指令为编程人员提供对并行化的完整控制;

(3) 已有语言可直接调用的并行库,如 MPI,TBB,TPL 等.无论哪种实现,最终都体现为对传统编译器的修改以及提供一个新的运行时库.

在现有的并行程序设计模型中,有不少都考虑了对错误容忍的支持,如 Condor[24],LAM/MPI[25],Open MPI[26],FT-MPI[27],Cilk-NOW[28],Satin[29],ATLAS[30]等.在这些系统中,都采用检查点机制支持错误恢复.如,Condor 实现了自己的用户级检查点库,LAM/MPI 采用成熟的 BLCR 模块实现检查点功能.总的来看,现有并行程序设计模型中对容错的考虑主要是针对处理单元的故障,即永久错误,当某个处理单元(节点)发生错误后,

由其他处理单元来完成错误节点上未完成的任务.这些系统中的容错机制和并行程序设计模型本身相互比较独立,相当于是在设计好并行程序设计模型后考虑容错功能,且没有针对瞬时错误的检测和恢复机制.虽然近年来也有研究工作在设计并行程序设计模型的同时考虑了容错[31,32],但都只是一般性的讨论,没有给出详细的容错功能设计和成型系统.

本文在基于任务的并行程序设计模型基础上实现了对瞬时错误、永久错误的检测与恢复,这一改进不是将现有容错技术简单应用于并行程序设计中,而是将任务冗余执行与划分、调度等整体考虑,设计出统一的编程模型.

2 支持容错的任务并行程序设计模型

如何用纯软件的方法高效地实现硬件错误的检测与恢复,是 FT-TPP 的主要设计问题.当然,错误检测与恢复机制与任务并行程序设计模型的任务调度、划分等问题是相互关联的.例如,错误检测的冗余任务影响着调度算法和任务队列的设计,并行调度又对错误检测提供了支持.本节首先介绍 FT-TPP 的基本框架,然后介绍 FT-TPP 的错误检测与恢复机制,之后介绍 FT-TPP 采用的容错工作窃取任务调度等关键技术.

2.1 任务并行程序设计模型框架

FT-TPP 面向多核集群和类似具有两层(节点间和节点内)并行单元结构的系统,是一个基于任务的并行程序设计模型.一般来说,基于任务的并行程序设计模型主要包含以下几个方面的内容:(1) 并行任务的生成与表示,即,任务定义和划分方法;(2) 任务的调度执行,主要涉及工作线程管理和调度器设计;(3) 同步与通信机制,共享存储系统上主要是锁、原子操作等的实现;(4) 对上述几点提供支持和便利的相关数据结构与算法等.FT-TPP 主要在前两个方面区别于已有的任务并行程序设计模型,同步操作等都采用现有技术实现,因此下面分别介绍 FT-TPP 中任务的生成和调度.

2.1.1 任务的生成与表示

本文将任务并行的基本模式分为如下 3 类.

- 水平式的并行(flat parallelism):以往发掘并行性的主要目标——并行循环就属于此类形式,各循环迭代间相互独立,在同一层次上可被调度到任意处理单元上并行执行;
- 递归式的并行(recursive parallelism):主要指分治类型的应用,TBB,Cilk,Map-Reduce 等都针对这种并行形式建立起了很好的并行程序设计模型;
- 不规则的并行(irregular parallelism):某些应用适合用 DAG 图(directed acyclic graph)来表示,图中节点表示任务大小,边表示任务之间的依赖关系.基于 DAG 的任务调度已有广泛研究,这些研究实际上就是在发掘此类并行性.

另外,流水并行是一种特殊的并行形式,有特定的研究方式和方法,在 FT-TPP 中暂不考虑.

类似于 TBB,OpenMP 等,FT-TPP 对水平式并行采用隐式任务生成方式,即:提供简单的接口供程序员使用,具体的任务划分等由编译程序和并行库完成,对不规则并行采用显式任务生成方式,每个任务的定义都由程序员手工完成.

FT-TPP 对水平式并行模式,首先确定集群中各节点的任务分配,设并行循环大小为 N(迭代数),节点数为 p,l_j 和 u_j 分别表示分配给节点 P_j 的部分循环的上下边界索引值,我们用 λ_i 表示节点 P_i(i=1,2,…,p)的处理能力,并将 λ_i 以 λ_1 为标准归一化,例如 λ_1=1,λ_2=2 表示在 P_1 上执行相同工作负载的时间是 P_2 的两倍,分配给各节点的循环边界按如下公式计算:

$$l_1=1;l_{j+1}=u_j+1;u_j=\left\lfloor\frac{\sum_{i=1}^{j}\lambda_i}{\sum_{i=1}^{p}\lambda_i}N\right\rfloor,j=1,2,...,p-1;u_p=N \tag{1}$$

在各节点为实现多个处理核之间的动态负载均衡,先将分配给节点的任务均分给各个处理核,并进一步将

每个核的任务划分成许多小任务,这些小任务保存在各个核(工作线程)的任务队列里,由工作窃取的调度算法在核间动态迁移小任务以达到负载均衡.假设 u 和 l 是分配给某个处理核的循环上下边界,按如下公式进行任务划分,C_i 是划分后的各任务的大小.

$$R_0=u-l,R_{i+1}=R_i-C_i,C_i=\begin{cases}\lceil R_i/2\rceil, R_i\geqslant\theta\\ R_i, \quad R_i<\theta\end{cases} \tag{2}$$

上述划分形成从大到小的任务块,大块先执行,小块后执行.这样,在调度开销和负载均衡之间进行平衡,类似于传统循环调度的 FSS[33]块划分.θ是控制块大小的阈值,与实际应用相关,由用户输入决定.

FT-TPP 对在节点上用多个工作线程执行并行循环目前仅提供如下接口:

*parallel*_for(*iter*,*low*,*up*,min*size*).

iter 是循环控制变量,*low* 和 *up* 是循环边界,min*size* 是划分后的最小任务所包含的循环迭代个数,程序员只需将串行程序中能够并行执行的循环改写成上述形式即可,循环的划分、调度等都由 FT-TPP 完成.在进一步的工作中我们将为 FT-TPP 添加类似于 TBB 的 *parallel_reduce* 等其他并行循环接口.

对递归式并行,FT-TPP 要求程序员定义 *recursive* 类和 *reduce* 类,下面用斐波那契数的计算为例加以说明,对应的两个类定义的伪代码如图 1 所示.

```
class fib_reduce: public ft_tpp:: task{
public:
    long x, y;
    long*result;
    fib_reduce(long*r): result(r){·}
    void run(·){
        *result=x+y;
    }
};

class fib_recursive:public ft_tpp::task{
public:
    long n;
    long*result;
    fib_recursive(long i,long*r): n(i), result(r){·}
    void run(·){
        if (n<CutOff)
            *result=SerialFib(n);
        else {
            fib_reduce c(result);
            fib_recursive a(n−1,&c.x);
            fib_recursive b(n−2,&c.y);
            c.nPreTasks=2;
            a.next=&c; b.next=&c; …
        }
    }
};
```

Fig.1 An example of task definition for recursively parallel program

图 1 递归式并行任务定义示例

fib_recursive 判断递归结束条件,不满足时,生成两个新的 *fib_recursive* 任务对象和一个 *fib_reduce* 任务对象,并设置对象之间的关系,即:将 *c* 的前驱任务计数设为 2,*a* 和 *b* 的后继任务设为 *c*,调度器会取前驱任务计数为 0 的任务调度执行,并在执行完毕后更新其后继任务的该计数器.

对不规则并行,程序员需要将所有任务封装成任务类,生成各个任务对象,并根据应用的 DAG 图设置好任务对象之间的关系;然后,将这些任务对象放入一个任务队列中,调度器会从中取出入度为 0 的 DAG 节点所对应的任务对象分配执行.

2.1.2 任务调度与执行

FT-TPP 是以任务为基本调度单元的并行程序设计模型,任务调度算法的总体目标可概括为:最小化程序执

行时间和最大化资源利用率.为实现这一目标,调度算法在设计时应尽量提高负载均衡、减少调度开销、保持数据局域性.

FT-TPP 调度算法所考虑的系统模型如图 2 所示.一定数量的多核计算节点通过高速网络相连.这样,整个系统可视为两个层次:一是由各计算节点组成的分布式存储层,二是每个节点内部由多个处理核组成的共享存储层.为发掘这两个层次的并行性,FT-TPP 采用层次化的调度框架,在完成初始的静态任务分配后,任务首先在节点内动态调度,以达到节点内各处理核之间的负载均衡;其次,在节点间适时地迁移,以平衡各计算节点的任务量.如图 2 所示,假设程序和数据文件已经预先部署在各个节点上,用户登录某个节点启动该应用程序,我们将这个节点作为主节点.当然,也可指定系统中的某个节点为主节点,用户必须登录该节点来启动计算,如许多高性能计算集群的资源管理节点,这样的节点作为主节点能为任务调度提供许多有用信息,比如当前可用的资源和实时负载情况等.由于主节点要负责任务的初始划分以及对其他工作节点的监测等,在主节点上需要运行一个全局调度器(GS).同时,主节点也作为工作节点之一承担计算任务,每个工作节点都运行一个局部调度器(LS),负责节点内各处理核之间的任务调度以及与全局调度器的信息交互.

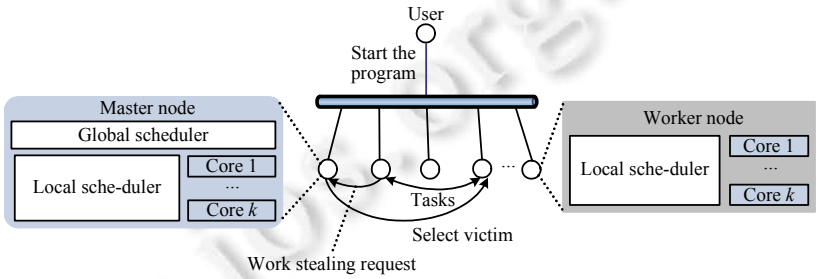


Fig.2 System model for FT-TPP task scheduling

图 2 FT-TPP 任务调度系统模型

FT-TPP 在每个处理核上运行一个工作线程,LS 采用一种支持容错的工作窃取任务调度策略(见第 2.3.1 节)平衡节点内各工作线程的负载.当某个节点上所有工作线程的任务队列都为空时,该节点的 LS 向 GS 所在节点发送工作窃取请求消息;GS 收到该消息后,选择负载最重的节点作为目标节点,并通知该节点把任务发送给工作窃取的请求节点,从而完成一次节点间的工作窃取.

总的来看,FT-TPP 采用的是层次化工作窃取的任务调度,节点内工作窃取采用随机目标选择策略.文献[34]证明了这种随机选择策略在共享存储系统中是高效的.但在分布式存储系统中,节点探测的开销相对较大,随机的目标节点选择可能会造成大量无效的探测,从而导致系统性能的降低.因此,FT-TPP 对节点间工作窃取采用确定的目标节点选择方式,也就是由 GS 直接决定目标节点,而不是由 LS 随机选取.为支持这种集中控制的节点选择,GS 需要了解所有节点的实时任务信息,包括任务的大小和迁移开销等.然而,获取和实时维护这些信息需要耗费大量时间和系统资源,在实际应用中不太可行.因此,我们只采用任务个数来表示节点的负载量,GS 为每个节点设置一个任务计数器,每个节点上的 LS 周期性地向 GS 发送消息来更新该节点的任务计数器.这样,GS 就能掌握各节点的任务数量,从而确定任务量最多的节点为工作窃取的目标节点.

2.2 错误检测与恢复机制

由于间歇错误可看作是周期性发生的瞬时错误,因此这里只讨论瞬时错误和永久错误.

2.2.1 瞬时错误

考虑以往的瞬时错误检测方法,如 SRT,CRT 等,需要微处理器体系结构上的修改,EDDI 和 PLR 虽然不需要修改硬件,但一个是在指令级一个是在进程级实现错误检测,粒度过大或过小都会带来诸多缺点.因此,我们提出任务级的错误检测,任务粒度大小适中,可避免指令级或进程级错误检测的缺点.另外,任务的调度、复制、结果比较等便于软件实现,无需硬件架构上的改动.

为检测一个任务执行过程中的瞬时错误,需要将该任务在两个不同的处理单元(processing element,简称PE)上执行两次,然后对输出数据进行比较:结果如果一致,则认为没有错误发生,当前任务执行完成;结果如果不一致,表示执行过程中有错误发生,任务要由其他 PE 再次执行,并再次进行结果比较.这就要求任务具备可重复执行的能力,也就是一些文献中所说的幂等性(idempotent)[35].另外,与传统幂等执行不同,这里要求任务可同时被两个 PE 重复执行.为满足这一要求,我们采用类似于 SpiceC[36]的计算模型,如图 3 所示.每个 PE 对应一个工作线程,每个工作线程维护一个私有数据空间,共享空间保存任务间的共享数据,对同一个任务的两份拷贝来说,共享空间中保存的就是任务的原始数据.任务由某个工作线程执行的过程如下:首先,将数据从共享空间缓冲到线程私有空间;接下来,线程访问私有空间的数据执行任务,输出数据也都写入私有空间;然后,通过比较任务所对应的两个工作线程的私有空间数据来检测错误;最后,在未检测到错误的情况下,把私有空间中的数据提交到共享空间,如果检测到错误不进行提交操作,将任务调度到其他 PE 上直接重新执行.

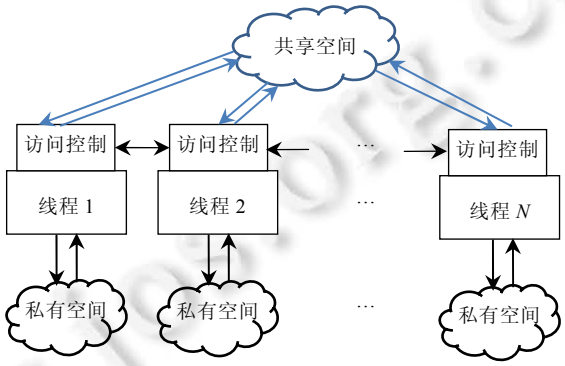


Fig.3 Buffer-Commit computation model

图 3 Buffer-Commit 的计算模型

如何确定 Buffer-Commit 的数据,是这种计算模型的关键问题.任务执行过程中所涉及的数据并不需要都进行缓冲、比较和提交,为尽量减少 Buffer-Commit 的数据量,我们只对任务中有写入操作的数据(任务对象的成员变量和全局变量)进行缓冲.设该部分数据集合为 A,比较提交的数据集合为 B,则 $B \subseteq A$,因为 A 中不作为后继任务输入或者程序输出的数据不需要比较提交,如任务类的私有成员变量(只在该任务中使用的临时数据).

FT-TPP 要求同一任务必须在两个不同的 PE 上执行,这样有利于提高错误的检出率.另外,为提高数据缓冲和提交的效率,当数据量较大时,利用空闲 PE 并行化缓冲和提交过程.

上述方法只能检测任务执行过程中的错误,任务调度、数据缓冲、比较和提交过程中的错误检测不到.针对这些过程中的错误,可以采用下面两种应对策略.

1) 不予考虑,因为这些操作的开销和任务执行开销相比较通常可忽略不计,所以不考虑这些操作中的错误时,上述方法的错误覆盖率仍旧相当好.另外,如果这些操作中的错误引起处理单元的崩溃,这时视为发生永久错误,任务会迁移到其他处理单元上执行;
2) 这些操作在运行时库中实现,可采用其他技术,如 EDDI 等保护起来,相当于逻辑上把实现这些操作的运行时库划分到 SoR 之外.

2.2.2 永久错误

FT-TPP 对永久错误的检测采用心跳检测机制实现.我们把系统中某个计算节点作为主节点,其他作为工作节点,每个工作节点周期性地向主节点发送心跳消息,当主节点超过一定时间未能接收到某个工作节点的消息时,则认为该工作节点发生了故障.

为恢复故障节点上的计算,我们采用无盘检查点技术,节点之间两两配对,每个节点在保存检查点后,将自己的检查点映射到配对节点的内存中.这样,当某个节点发生故障时,配对节点就能根据对方的检查点恢复故障

节点上的计算.这种技术涉及到以下几个关键问题.

1) 如何确定节点间的映射.

总的来说,应该是将相邻节点配对,这里的相邻节点不一定是物理上相邻,应是通信距离最短的两个节点.对异构系统或层次化集群系统,应根据系统结构确定节点映射,并考虑各节点的存储能力和计算能力等.

FT-TPP 目前简单实现了节点间相互保存检查点的映射,直接按节点 ID 顺序两两组合,主节点维护一个节点映射表,记录每个工作节点的检查点保存到了另外哪个工作节点上.

2) 如何最小化检查点的大小.

我们在应用层保存检查点,而不是在系统层保存检查点,这需要在程序设计时考虑保存到检查点中的内容以及何时进行检查点的保存.在基于任务的并行程序设计模型中,主要是考虑如何用最小的数据量保存一个任务的相关信息,这是和应用本身密切相关的.如在分块并行的矩阵乘法中,任务可仅由几个行号或列号来表示.

FT-TPP 的检查点保存各本地任务队列信息以及队列中的任务信息,检查点的保存由队列类型和任务类型数据结构中定义的检测点保存方法完成.各节点按照执行完成的任务数量周期性地保存检查点,检查点保存周期由用户根据实际应用决定.FT-TPP 目前实现中检查点保存的数据由程序员确定,今后将研究自动或半自动(编译器导向)的检查点数据生成方法.

3) 发生错误后,如何重新确定节点映射.

当某个节点崩溃后,与之配对的节点需要将其本地检查点映射到另一个相邻节点上.

FT-TPP 目前按节点 ID 顺序将检查点保存到最邻近的活动节点,例如节点 2 崩溃,之前节点 1 的检查点映射在节点 2 上,则重新将节点 1 的检查点映射到节点 3 上.

虽然 FT-TPP 中目前没有针对主节点故障的恢复措施,但在分布式并行计算领域,这一问题已有一些成熟的解决方案.如:在系统中设置主节点的备份节点,当主节点出现故障时,备份节点充当主节点的角色.

2.3 支持容错的任务调度

2.3.1 容错工作窃取

在多核计算节点内,我们采用工作窃取(work-stealing)的任务调度方法.工作窃取是目前任务并行程序设计中采用的最流行的动态调度方法,在 Intel TBB,Cilk,IBM X10,Microsoft TPL,OpenMP 3.0,Java Concurrency Utilities 等并行程序设计模型中得到广泛应用.基本的工作窃取任务调度如图 4 所示:每个 PE 维护一个任务队列,程序执行过程中产生的任务被从队列底部压入,队列中的任务都是相互独立的,可被并行执行;运行时,每个 PE 从自己任务队列的底部每次取出一个任务执行,当某个 PE 的队列为空时,该 PE 就会从其他 PE 的任务队列中窃取一个或一组任务,以此达到动态负载均衡.通常,空闲 PE 会随机选择一个目标 PE 进行工作窃取,为减小同步开销,任务总是在队列顶部进行窃取.

不同于以往的工作窃取任务调度,在 FT-TPP 中,为支持瞬时错误的检测,我们提出容错工作窃取调度方式,如图所示 5:在同一计算节点内的每两个 PE 共享一个任务队列(task queue)和一个出错任务队列(faulty task queue),任务队列中的每个任务都将被队列所属的两个 PE 执行两次,并比较结果,出错的任务将被压入其他 PE 对的出错任务队列中(图中点线).为缩短错误恢复的时间,出错任务总是被压入该队列的底部,PE 在执行时也总是先检查出错任务队列中是否有任务,然后再检查任务队列.图中 P'和 P''分别表示当前执行此任务的两个 PE.

图 6 用一示例描述容错工作窃取任务调度过程,图中 P_0 和 P_1 共享一个任务队列,其中存在两个任务 C_0 和 C_1.P_0 和 P_1 首先从队列中获取 C_0 执行,执行前,P_0 设置 C_0 的 P'为 0(P_0 的 ID),P_1 设置 C_0 的 P''为 1(P_1 的 ID).也就是说,P_0,P_1 在从任务队列中获取任务执行时分别更新任务的 P',P''标志.另外,每个任务设有一个执行次数标志 e(初始为 0),P_0 或 P_1 执行完一个任务后同步地使 e 加 1,当 e=2 时,表示任务的两次冗余执行完成.任务只有在两次执行完成并比较结果进行相应处理后才会从任务队列中移除.

如图 6(a)所示,假设 P_0 在 P_1 之前完成任务 C_0 的执行,P_0 不会等待 P_1 执行完 C_0,而是获取队列中的下一个任务 C_1 继续执行.当 P_1 执行完 C_0 后负责比较结果:如果未检测到错误,则提交数据并将 C_0 从任务队列中移除,如果有新任务生成,这些新任务也由 P_1 负责写入与 P_0 共享的任务队列;如果检测到错误,P_1 将把 C_0 迁移到其他

PE 对(随机选择)的出错任务队列.这种松耦合的冗余执行,最大限度地发掘了任务并行性,

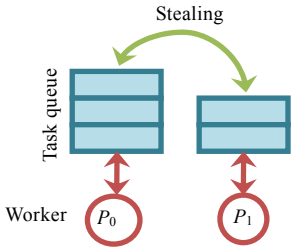


Fig.4 Classic work-stealing

图 4 传统工作窃取任务调度

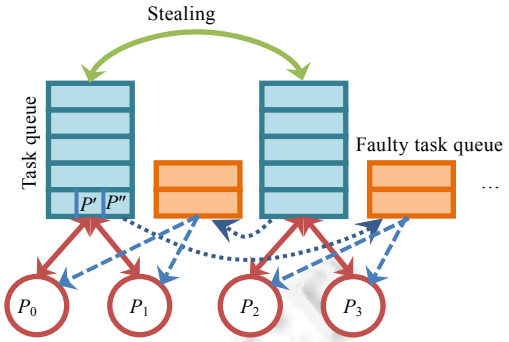


Fig.5 Fault tolerant work-stealing

图 5 容错工作窃取任务调度

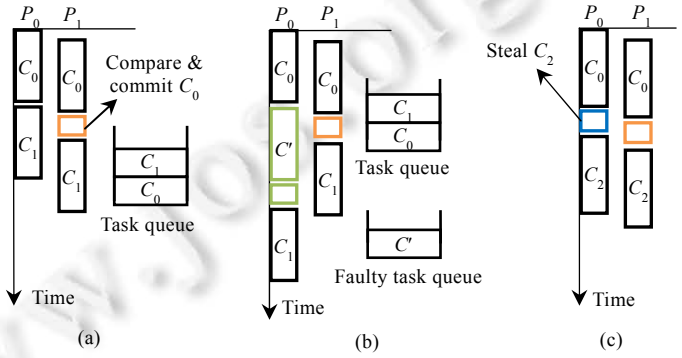


Fig.6 Task queues and scheduling for sample execution

图 6 任务队列及调度执行过程示例

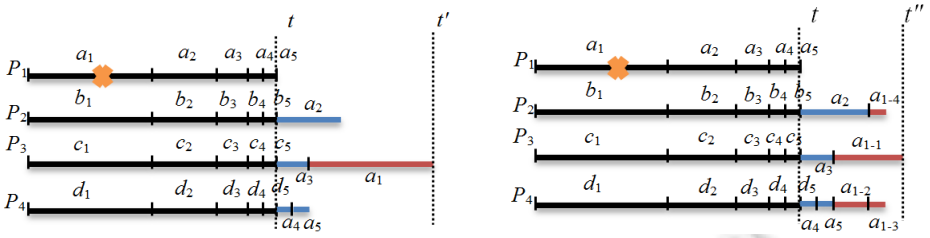
图 6 中,P_0 完成 C_0 后的任务调度有以下 3 种可能的情况.

1) P_0 首先查看与 P_1 共享的出错任务队列,如果其中存在任务,如图 6(b)所示,出错任务队列里有一个任务 C',则 P_0 先执行 C'并进行结果比较和相应处理.优先执行出错任务的好处是能尽快恢复错误;
2) 如果出错任务队列为空,而本地任务队列不为空,如图 6(a)所示,那么 P_0 接着执行 C_1;
3) 如图 6(c)所示,本地任务队列和出错任务队列都为空,P_0 则负责从其他 PE 对的任务队列窃取一个任务执行,P_0 在任务窃取时判断是否所有任务队列都为空,如果是,则有两种可能的情况:
   a) 所有出错任务队列也都为空,则程序结束;
   b) 某个出错任务队列里还有任务,P_0 则获取出错任务执行并比较结果:如果结果比较仍旧不相同,则 P_0 将重新执行一遍该任务并直接提交,不再迁移该出错任务.这样做是为了避免一个出错任务无休止地在出错任务队列中来回迁移.

2.3.2 失败任务的动态划分

针对 FT-TPP 中的错误恢复,我们提出错误块的动态划分策略,如图 7 所示:设有 4 个处理器,块从大到小调度,如果不发生错误,理想情况下 P_1~P_4 同时在 t 时刻结束计算;如果 P_1 在运行 a_1 时发生故障,则分配给 P_1 的任务需要由其他处理器恢复执行.传统方式如图 7(a)所示,P_2~P_4 在结束自身任务后获取 a_2~a_5 执行,错误块 a_1 在 P_3 上完全重新执行,这样,最终程序完成时间为 t'.从图 7(a)可见,这种方式造成了显著的负载不均衡.针对这一点,FT-TPP 在错误恢复时对块 a_1 重新分割,如图 7(b)所示,将 a_1 分割为 $a_{1\text{-}1}$~$a_{1\text{-}4}$,动态调度到 P_2~P_4 上.这样,程序

完成时间变为 t'',早于 t',在 P_2~P_4 上获得较好的负载均衡.另外,不是所有的错误块都应该重新分割,如当块大小较小时,重新分割带来的额外开销可能比负载均衡上的性能提升还要大,因此还需要确定是否进行重新分割.



(a) 传统方式直接 Redo (b) 动态划分后 Redo

Fig.7 An example of the chunk scheduling in the presence of a fault

图 7 出错时的块调度

针对并行循环,设出错任务块 a_1 大小为 R,故障处理单元个数为 p_{crash},我们按照以下公式重新划分 a_1,C_i 是划分后的各块大小.

$$R_0=R, R_{i+1}=R_i-C_i, C_i=\begin{cases}\lceil R_i/p-p_{crash}\rceil, & R_i\geqslant\theta\\ R_i, & R_i<\theta \text{ or } p-p_{crash}=1\end{cases} \tag{3}$$

3 模型实现

我们用 PThread 和 MPICH 实现了 FT-TPP 的运行时库,包括调度器、工作线程和任务队列管理等.由于篇幅所限,这里主要介绍任务定义、执行和调度器的实现.

3.1 任务定义与执行

FT-TPP 对容错的支持对程序员来说是透明的,我们实现了一个源到源的代码转化程序把程序员定义的隐式或显式任务(见第 2.1.1 节)转化为支持容错的任务定义.下面以标准矩阵乘法($C=A\times B$)为例说明转化后的任务,如图 8(a)所示:为支持瞬时错误检测,任务由两个工作线程分别调用 *Execute*1(·)和 *Execute*2(·)执行两次,结果分别写入 c_1 和 c_2,这两个缓冲区在 *Execute*(·)中分配,在 *Commit*(·)中释放.另外,为提高效率,我们用内存池技术管理这些缓冲区,*Compare*(·)比较 c_1 和 c_2 中的数据,*Commit*(·)负责将 c_1 的数据复制到结果 c 中.

```
class Task: public TaskBase {…
  double*a, *b, *c;   //Point to the data of the matrices.
  double*c1, *c2;   //Point to the buffers for the double executions.
  int startrow, size;   //Portions of the matrices computed in this task.
  void Spawn (TaskQueue *tq);
  Task*Execute1(·) { Execute (c1); }
  Task*Execute2(·) { Execute (c2); }
  void Execute(double*c);
  bool Compare(·);
  void Commit(·);
  …
  TaskInfo*Checkpoint(·);
  Task*Recover(TaskInfo*tinf);
};
```

(a) 任务定义

```
while (true) {
  Task*t=LS.Schedule(·);
  if (thread_id%2==0) t→Execute1(·);
  else t→Execute2(·); …
  if (t→tag==2){   //The double executions are finished.
    if (true==t→Compare(·))   //No fault occurs.
      t→Commit(·);
    else
      LS.FaultyTQ_push(t);   //Push t to a faulty task queue.
  }…}
```

(b) 工作线程实现

Fig.8 Pseudocode for task definition and worker thread implementation

图 8 支持容错的任务定义、执行的伪代码示例

以上是 Buffer-Commit 计算模型的简单实现.

为支持永久错误的恢复,图 8(a)的类中定义了两个函数 *Checkpoint* 和 *Recover*,在应用级检查点的保存过程

中,对任务队列中的每个任务调用 *Checkpoint* 保存任务相关信息到 TaskInfo 结构对象中;然后,由局部调度器(LS)将这些 TaskInfo 对象打包成一个检查点数据,检查点数据的传输和分布式计算节点间的任务迁移都由消息传递实现.*Recover* 负责将 TaskInfo 对象还原成本地任务对象并加入任务队列等待执行.

图 8(b)是工作线程的伪代码描述,工作线程每次通过调用 *LS.Schedule*(·)得到一个任务,然后调用相应的执行函数 *Execute*1(·)或 *Execute*2(·),执行完毕后检查任务的 *tag* 标记,判断两次执行是否完成:如果完成,则比较结果进行错误检测和相应处理;否则继续获取下一个可执行的任务,结果的比较等留给双执行中的另一个工作线程来完成.

3.2 调度器实现

FT-TPP 的调度器包括全局调度器 GS 和局部调度器 LS,GS 和 LS 实现的伪代码描述如图 9 所示.GS 根据任务并行模式进行初始划分,并设置各工作节点的任务计数器,然后进入消息循环等待来自 LS 的消息.当接到工作窃取请求时,GS 根据各节点的任务计数选择任务量最大的节点作为目标节点,此时,应考虑是否值得进行任务窃取,我们通过阈值 T_{ws} 判断:当目标节点的任务量大于 T_{ws} 时,GS 发送 *VICTIM* 消息(带有请求节点 ID)给所选择的目标节点启动任务窃取过程;否则,不进行任务窃取.另外,如果所有任务计数都为 0,则发送 *TERMINATION* 消息进行计算终止的判定.

```
GS:
InitPartition(·);
InitTaskCount(·);
while (true) {
  ReceiveMsg(msg);
  switch(msg) {
    STEAL_REQ:
      if (all the task counters are zero)
        SendMsg(TERMINATION); …
      else {
        Select a victim P_v, c_v=max{c_1,c_2,…,c_p}.
        if (c_v>T_ws)
          SendMsg(VICTIM); …}
    UPDATE_TC:
      Update task counter and SendMsg(TC_UPDATED);
  …}
}
```

```
LS:
if (Δ>T_f) {Update(γ); Δ=0;}
if (γ=0) {
  SendMsg(STEAL_REQ);
  Suspend the current thread w_i, …}
if (FQ_i!=NULL) Pop a task t from FQ_i.
else if (Q_i!=NULL) Pop a task t from Q_i.
else t=Random_steal(·);   //steal a task t.
if (t!=NULL) return t; …
------------------------------------------------------------
while (true) {
  ReceiveMsg(msg);
  switch(msg) {
    VICTIM:
      if (γ>T_s) {
        SendMsg(TASK);
        ReceiveMsg(TASK_RECEIVED); …}
    TASK:
      Update(γ);
      SendMsg (TASK_RECEIVED);
      Resume a worker thread,...
    TERMINATION:
      Exit worker threads w_1,w_2,…,w_n.
      Break;
    CHECKPOINT:
      Save the checkpoint received.
  …}
}
```

Fig.9 Pseudocode for the schedulers (GS and LS)

图 9 调度器实现的伪代码描述(GS 和 LS)

LS 维护节点内部各任务队列,在这些任务队列之间平衡负载,并定期发送任务计数给 GS.如图 9 所示,每个工作线程空闲时调用虚线上方的 LS 功能,即 *LS.Schedule*(·),虚线下方是一个专门的线程负责与 GS 的消息通信.FQ_i 和 Q_i 分别是出错任务队列和任务队列.先看虚线上方方法,设节点当前任务总数为 γ,我们通过变量 Δ 记录任务数的变化次数,用来控制节点任务计数的更新频率,γ 在任务生成时加 1,结束时减 1,Δ 在生成和结束时都加 1.γ 和 Δ 的访问用原子操作实现.当 Δ 大于阈值 T_f 时,将当前节点的任务总数 γ 发送给 GS,通过调整阈值 T_f 的大小可控制向 GS 发送任务计数更新消息的频率.当 γ 等于 0 时,节点上的任务队列都为空,LS 发送 *STEAL_REQ* 消息给 GS,然后等待接收消息.接收到的消息可能是从目标节点发送过来的任务,也可能是计算终止消息.接下来看虚

线下方 LS 的消息处理循环,LS 接到 *VICTIM* 消息后要判断本地任务队列里有没有多的任务:若有,则从队列尾部取出一个任务打包发送给请求节点;若无,则不用专门通知请求节点,因为马上该节点就会向 GS 发送工作窃取请求,而此时,该节点应是任务数最多的节点,因此,GS 会很快判断出所有任务结束,并给各个节点发送计算终止消息.原则上,出错队列中的任务如果在本节点内被所有 PE 对执行后还不能得到一致结果,则应迁移到其他节点执行,但在目前 FT-TPP 的实现中,为使队列管理等不过于复杂,我们不允许出错任务在节点间迁移.

4 实验与分析

4.1 实验平台与测试程序

为测试 FT-TPP 的容错能力和对程序性能的影响,我们基于 FT-TPP 分别实现了表 1 中的应用,这些应用涵盖了 FT-TPP 支持的 3 种任务并行模式,Fib,Nq 和 Ms 属于递归式并行,MM 属于水平式并行,St 本身可实现为 3 种模式中的任意一种.这里,我们对 Strassen 算法递归几次后形成的任务 DAG 实现为不规则并行模式.

实验在一个 16 节点的多核集群上进行,各节点基本配置为:2.4GHz Intel Xeon E5620 处理器(支持 8 个硬件线程),12G 内存,Linux 内核 3.4.

Table 1 Benchmark applications

表 1 测试程序

名称	描述
Fib (*n*)	递归地计算斐波那契数列的第 *n* 项值
Nq (*n*)	求解 *n* 皇后问题
MM (*n*)	标准矩阵乘法的并行实现,对最外层循环并行化(*n*×*n* double matrix)
Ms (*n*)	对长度为 *n* 的整数类型数据进行并行归并排序
St (*n*)	Strassen 快速矩阵乘法(*n*×*n* double matrix)[37]

4.2 容错性能分析

为检验 FT-TPP 在瞬时错误检测与恢复方面的效果,我们用 Pin 工具[38]在测试用例运行过程中注入 SEU(单位翻转)错误,错误注入后的程序行为可分为 4 类:(1) 检测到该瞬时错误并进行错误恢复(detected);(2) 发生段错误退出程序(seg fault);(3) 程序运行超时(timeout),亦即受错误影响的线程在所有任务都结束后始终不能退出,可能陷入了一个死循环;(4) 程序正常结束并输出正确结果(benign).每个测试程序各运行 1 000 次,为控制程序运行时间,我们在本实验中采用较小的输入参数,实验统计上述 4 种程序行为发生的比例,结果如图 10 所示.

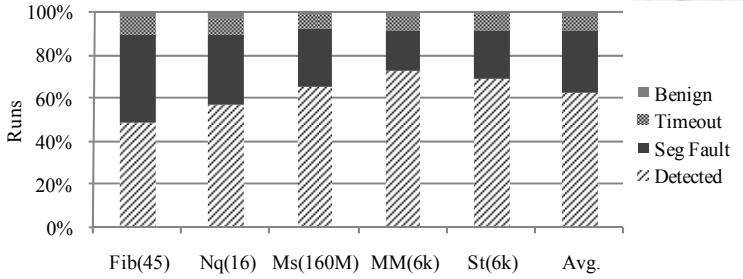


Fig.10 Fault detection distribution of FT-TPP with transient faults

图 10 FT-TPP 瞬时错误检测与恢复情况分布

平均来看,有 62%的错误被检测和恢复,28%的错误引起段错误,还有 6.8%的超时错误.另外,虽然有 38%的错误未被检测到,但由于 FT-TPP 的动态任务迁移和永久错误检测与恢复机制,所有程序最终都成功完成了计算任务.从这一点来看,FT-TPP 实际提供了 100%的错误覆盖率.但在理论上,FT-TPP 的错误覆盖率不可能是 100%,因为程序可能在下述两种发生概率极低的情况下输出错误结果:(1) 两次瞬时错误发生在同一任务两次执行过程中的相同位置;(2) 错误发生在所有任务队列都为空且出错任务队列里的任务最后一次直接执行并提交的过

程中(见第 3.3.1 节).

为检验 FT-TPP 在永久错误容忍方面的效果,我们在程序运行过程中通过 kill 相关计算进程来模拟节点故障,我们进行了两次实验:第 1 次实验在每个程序运行过程中随机终止一个节点上的计算进程,第 2 次随机终止两个节点上的计算进程.实验中,每个应用分别执行 100 次并计算平均执行时间,结果如图 11 所示.为便于比较,各执行时间以无错误情况下的执行时间为标准归一化,结果可见:单节点故障情况下程序执行时间平均比无故障情况增加了 2.76%,双节点故障情况下程序执行时间平均增加了 5.6%.总的来看,FT-TPP 能很快恢复故障节点上的任务,并成功完成所有计算任务.这归功于 FT-TPP 所采用的无盘检查点机制和动态任务调度,故障节点上未完成的任务从检查点恢复由调度器分配给各处理单元并行执行.

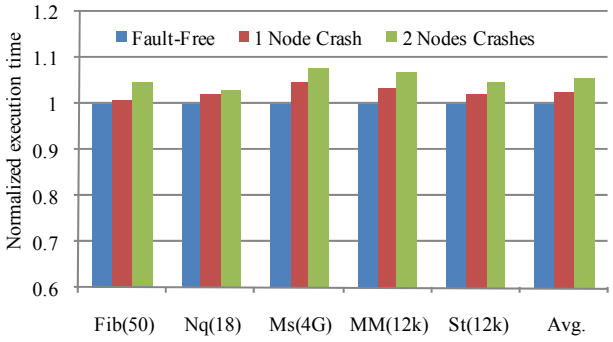


Fig.11 Performance of FT-TPP with permanent faults

图 11 FT-TPP 永久错误检测与恢复的性能

4.3 开销分析

为评估 FT-TPP 的容错开销,我们将其实现中与容错相关的部分剥离出去,形成一个普通的不支持容错的任务并行程序设计模型,简记为 NoFT-TPP,比较 FT-TPP 与 NoFT-TPP 就可对 FT-TPP 的容错开销有一个总体了解.NoFT-TPP 实际上与文献[39]工作类似,是一个层次化工作窃取任务调度框架.我们将表 1 中的应用基于 FT-TPP 和 NoFT-TPP 分别实现,对每个测试程序各执行 40 次并计算平均执行时间,运行过程中不引入任何错误,结果如图 12 所示.

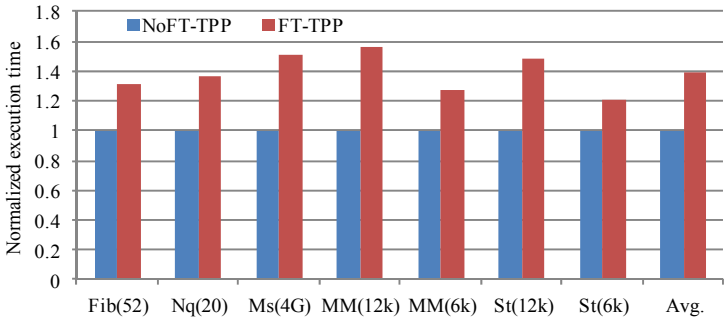


Fig.12 Performance comparison of FT-TPP with NoFT-TPP

图 12 FT-TPP 与 NoFT-TPP 的性能比较

各执行时间以 NoFT-TPP 为标准归一化,与 NoFT-TPP 相比,FT-TPP 执行时间平均增加了 39%.也就是说,FT-TPP 总体容错开销大约为 39%.这些开销主要来自于 Buffer-Commit、检查点相关操作、冗余执行等几个方面.

Buffer-Commit 和检查点的开销与任务输出的数据量相关,对于 Fib(n)和 Nq(n),每个任务仅输出少量数据,而对 MM(n),Ms(n)和 St(n)而言,每个任务会产生大量数据,因此在图 12 中,Fib(52)和 Nq(20)的 FT-TPP 容错开销低于 Ms(4G),MM(12k)和 St(12k).FT-TPP 对每个任务要执行 2 次,而 NoFT-TPP 对每个任务只执行 1 次,因此理

论上,FT-TPP 的容错开销应高于 100%,但在图 12 中,FT-TPP 的最大容错开销仅为 57%,见 MM(12k).这主要归功于 FT-TPP 对任务之间并行性的充分发掘,在处理单元数量充足的情况下,冗余任务也被均衡地调度到各个处理单元上并行执行,细粒度的任务并行和高效的负载均衡策略使得执行时间并不加倍.另外,比较图 12 中 MM(12k)与 MM(6k),St(12k)与 St(6k)的执行时间可知:数据量的减少能显著降低 FT-TPP 的容错开销,St(6k)的容错开销只有 21%,这与 St(12k)(48%)相比降低了超过一半.这是因为,我们在实验中观察到:对较小的数据量,在 MM(n)和 St(n)的执行过程中,有的处理单元由于得不到任务,长时间处于空闲状态,FT-TPP 的冗余任务执行使其资源利用率在这种情况下高于 NoFT-TPP.

为进一步分析 FT-TPP 瞬时错误检测的性能开销,即,本文 Buffer-Commit 计算模型与容错工作窃取调度策略的效果,我们将 FT-TPP 和 NoFT-TPP 简化为共享存储模式,即,去除其中节点间消息传递相关部分.另外,对 FT-TPP 去除检查点相关操作,然后在实验平台的一个计算节点上运行各测试程序,比较简化后的 FT-TPP 与 NoFT-TPP,分别记为 FT-TPP-S 与 NoFT-TPP-S(S 表示针对共享存储).结果如图 13 所示:平均来看,FT-TPP 以 27.4%的性能开销提供了瞬时错误的检测能力.

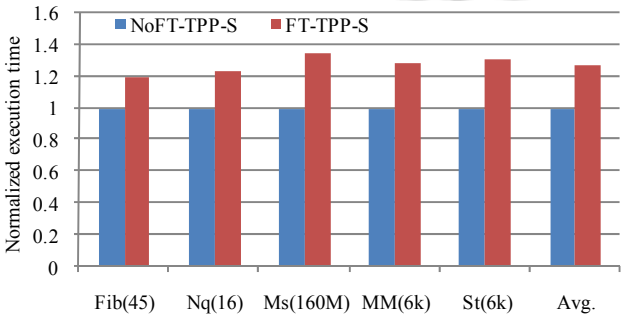


Fig.13 Performance comparison of FT-TPP-S with NoFT-TPP-S

图 13 FT-TPP-S 与 NoFT-TPP-S 的性能比较

5 总 结

本文提出了一种支持容错的任务并行程序设计模型 FT-TPP,FT-TPP 在发掘任务级并行性以提高系统性能的同时,实现了任务粒度的错误检测和恢复机制,以保证系统可靠性.FT-TPP 目前支持水平、递归和不规则并行这 3 种并行模式,调度策略采用分层工作窃取任务调度,主要采用以下关键技术实现容错支持:Buffer-Commit 的计算模型、容错工作窃取任务调度以及应用级无盘检查点.

进一步的工作将围绕以下几个方面进行:(1) 考虑对流水并行模式的支持;(2) 探讨编译器辅助进行的检查点任务数据生成方法;(3) 优化运行时库的实现减小实际性能开销;(4) 将本文容错工作窃取任务调度等技术应用于 TBB 等现有并行程序设计语言和运行库中.

References:

[1] Reinders J. Intel Threading Building Blocks. 2015. https://www.amazon.com/Intel-Threading-Building-Blocks-Parallelism/dp/0596514808

[2] Charles P, Grothoff C, Saraswat V, Donawa C, Kielstra A, Ebcioglu K, von Praun C, Sarkar V. X10: An object-oriented approach to non-uniform cluster computing. In: Johnson R, ed. Proc. of the 20th Annual ACM SIGPLAN Conf. on Object-Oriented Programming. New York: ACM Press, 2005. 519−538. [doi: 10.1145/1103845.1094852]

[3] Frigo M, Leiserson CE, Randall KH. The implementation of the cilk-5 multithreaded language. In: Berman AM, ed. Proc. of the ACM SIGPLAN'98 Conf. on Programming Language Design and Implementation. New York: ACM Press, 1998. 212−223. [doi: 10.1145/277650.277725]

[4] Wang L, Cui HM, Chen L, Feng XB. Research on task parallel programming model. Ruan Jian Xue Bao/Journal of Software, 2013, 24(1):77−90 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4339.htm [doi: 10.3724/SP.J.1001.2013.04339]

[5] Leijen D, Schulte W, Burckhardt S. The design of a task parallel library. In: Arora S, ed. Proc. of the 24th Annual ACM SIGPLAN Conf. on Object-Oriented Programming. New York: ACM Press, 2009. 227−242. [doi: 10.1145/1640089.1640106]

[6] Yang C, Orailoglu A. Full fault resilience and relaxed synchronization requirements at the cache-memory interface. IEEE Trans. on VLSI System, 2011,19(11):1996−2009. [doi: 10.1109/TVLSI.2010.2067230]

[7] Reinhardt SK, Mukherjee SS. Transient fault detection via simultaneous multithreading. In: Berenbaum A, ed. Proc. of the 27th Annual Int'l Symp. on Computer Architecture. New York: ACM Press, 2000. 25−36. [doi: 10.1145/339647.339652]

[8] Oh N, Shirvani PP, McCluskey EJ. Error detection by duplicated instructions in super-scalar processors. IEEE Trans. on Reliability, 2002,51(1):63−75. [doi: 10.1109/24.994913]

[9] Reis GA, Chang J, Vachharajani N, Rangan R, August DI. SWIFT: Software implemented fault tolerance. In: Conte T, ed. Proc. of the Int'l Symp. on Code Generation and Optimization. Washington: IEEE Computer Society, 2005. 243−254. [doi: 10.1109/CGO.2005.34]

[10] Rotenberg E. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In: Proc. of the 29th Annual Int'l Symp. on Fault-Tolerant Computing. Madison: IEEE, 1999. 84−91. [doi: 10.1109/FTCS.1999.781037]

[11] Mukherjee SS, Kontz M, Reinhardt SK. Detailed design and evaluation of redundant multithreading alternatives. In: Skadron K, ed. Proc. of the 29th Annual Int'l Symp. on Computer Architecture. Washington: IEEE Computer Society, 2002. 99−110. [doi: 10.1109/ISCA.2002.1003566]

[12] Shye A, Blomstedt J, Moseley T, Reddi VJ, Connors DA. PLR: A software approach to transient fault tolerance for multicore architectures. IEEE Trans. on Dependable and Secure Computing, 2009,6(2):135−148. [doi: 10.1109/TDSC.2008.62]

[13] Randell B, Lee P, Treleaven PC. Reliability issues in computing system design. ACM Computing Surveys, 1978,10(2):123−165. [doi: 10.1145/356725.356729]

[14] Du YF. The study and analysis on fault-tolerant parallel algorithm [Ph.D. Thesis]. Changsha: National University of Defense Technology, 2008 (in Chinese with English abstract).

[15] Bronevetsky G, Marques D, Pingali K, Szwed PK, Schulz M. Application-Level checkpointing for shared memory programs. In: Proc. of the 11th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. New York: ACM Press, 2004. 235−247. [doi: 10.1145/1024393.1024421]

[16] Chen Z. Adaptive checkpointing. Journal of Communications, 2010,5(1):81−87. [doi: 10.4304/jcm.5.1.81-87]

[17] Huang KH, Abraham JA. Algorithm-Based fault tolerance for matrix operations. IEEE Trans. on Computers, 1984,33(6):518−528. [doi: 10.1109/TC.1984.1676475]

[18] Dean J, Ghemawat S. Mapreduce: Simplified data processing on large clusters. Communications of the ACM, 2008,51(1):107−113. [doi: 10.1145/1327452.1327492]

[19] NVIDIA Corp. NVIDIA CUDA. 2010. http://developer.nvidia.com/object/cuda.html/

[20] Khronos. OpenCL: The open standard for parallel programming of heterogeneous systems. 2010. http://www.khronos.org/opencl/

[21] Sophisticated Library for Vector Parallelism. Intel array building blocks. 2010. http://software.intel.com/en-us/articles/intel-array-building-blocks/

[22] El-Ghazawi TA, Carlson WW, Draper JM. UPC Language Specification v1.1.1. 2003.

[23] Cray Inc. The chapel language specification version 0.4. Technical Report, Cray Inc., 2005.

[24] Tannenbaum T, Jim B, Litzkow M, Livny M. Checkpoint and migration of Unix processes in the condor distributed processing system. Technical Report, 1346, University of Wisconsin-Madison Computer Sciences, 1997.

[25] Burns G, Daoud R, Vaigl J. LAM: An open cluster environment for MPI. In: Proc. of the Supercomputing Symp. Ontario: University of Toronto, 1994. 379−386.

[26] Gabriel E, Fagg GE, Bosilca G, Angskun T, Dongarra JJ, Squyres JM, Sahay V, Kambadur P, Barrett B, Lumsdaine A, Castain RH, Daniel DJ, Graham RL, Woodall TS. Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Kranzlmuller D, ed. Proc. of the 11th European PVM/MPI Users' Group Meeting. Berlin: Springer-Verlag, 2004. 97−104. [doi: 10.1007/978-3-540-30218-6_19]

[27] Fagg GE, Gabriel E, Chen Z, Angskun T, Bosilca G, Pjesivac-Grbovic J, Dongarra JJ. Process fault-tolerance: Semantics, design and applications for high performance computing. Int'l Journal of High Performance Computing Applications, 2005,19(4):465−477. [doi: 10.1177/1094342005056137]

[28] Blumofe RD. Executing multithreaded programs efficiently [Ph.D. Thesis]. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1995.

[29] van Nieuwpoort RV, Wrzesińska G, Jacobs CJH, Bal HE. Satin: A high-level and efficient grid programming model. ACM Trans. on Programming Languages and Systems, 2010,32(3):39. [doi: 10.1145/1709093.1709096]

[30] Baldeschwieler JE, Blumofe RD, Brewer EA. Atlas: An infrastructure for global computing. In: Proc. of the 7th Workshop on ACMSIGOPS European Workshop. New York: ATLAS, 1996. 165−172. http://dl.acm.org/citation.cfm?id=504482&dl=ACM&coll=DL&CFID=632100309&CFTOKEN=67677564

[31] Taura K, Kaneda K, Endo T, Yonezawa A. Phoenix: A parallel programming model for accommodating dynamically joining/leaving resources. In: Eigenmann R, ed. Proc. of the 9th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. New York: ACM Press, 2003. 216−229. [doi: 10.1145/781498.781533]

[32] Bahi JM, Hakem M, Mazouzi K. Reliable parallel programming model for distributed computing environments. In: Lin HX, ed. Proc. of the Euro-Par Workshops. Berlin: Springer-Verlag, 2010. 162−171. [doi: 10.1007/978-3-642-14122-5_20]

[33] Flynn-Hummel S, Schonberg E, Flynn LE. Factoring: A method for scheduling parallel loops. Communications of the ACM, 1992, 35(8):90−101. [doi: 10.1145/135226.135232]

[34] Blumofe RD, Leiserson CE. Scheduling multithreaded computations by work stealing. Journal of the ACM, 1999,46(5):720−748. [doi: 10.1145/324133.324234]

[35] Zhang W, Kruijf M, Li A, Lu S, Sankaralingam K. ConAir: Featherweight concurrency bug recovery via single-threaded idempotent execution. In: Sarkar V, ed. Proc. of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. New York: ACM Press, 2013. 113−126. [doi: 10.1145/2451116.2451129]

[36] Feng M, Gupta R, Hu Y. SpiceC: Scalable parallelism via implicit copying and explicit commit. In: Cascaval C, ed. Proc. of the 16th ACM Symp. on Principles and Practice of Parallel Programming. New York: ACM Press, 2011. 69−80. [doi: 10.1145/2038037.1941564]

[37] Strassen V. Gaussian elimination is not optimal. Numerische Mathematik, 1969,14(3):354−356.

[38] Luk CK, Lowney PG, Hazelwood KM. Pin: Building customized program analysis tools with dynamic instrumentation. In: Sarkar V, ed. Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation. New York: ACM Press, 2005. 190−200. [doi: 10.1145/1065010.1065034]

[39] Quintin JN, Wagner F. Hierarchical work-stealing. In: D'Ambra P, ed. Proc. of the 16th Int'l Euro-Par Conf. Berlin: Springer-Verlag, 2010. 217−229. [doi: 10.1007/978-3-642-15277-1_21]

附中文参考文献:

[4] 王蕾,崔慧敏,陈莉,冯晓兵.任务并行编程模型研究与进展.软件学报,2013,24(1):77−90. http://www.jos.org.cn/1000-9825/4339.htm [doi: 10.3724/SP.J.1001.2013.04339]

[14] 杜云飞.容错并行算法的研究与分析[博士学位论文].长沙:国防科学技术大学,2008.

王一拙(1979－),男,陕西西安人,博士,讲师,CCF 会员,主要研究领域为计算机体系结构,并行程序设计.

苏岩(1980－),男,博士生,主要研究领域为计算机体系结构.

陈旭(1983－),男,博士生,主要研究领域为计算机体系结构.

王小军(1979－),男,讲师,主要研究领域为计算机体系结构,嵌入式系统.

计卫星(1980－),男,博士,副教授,CCF 会员,主要研究领域为计算机体系结构,并行计算,嵌入式系统.

石峰(1961－),男,博士,教授,博士生导师,主要研究领域为计算机体系结构,嵌入式系统.