确定性并行技术

周 旭 卢 凯 陈 沉

(国防科技大学计算机学院 长沙 410073)

(国防科技大学并行与分布处理国家重点实验室 长沙 410073)

摘 要 由于执行个体之间的同步、竞争和干扰,并行程序的执行存在着不确定性问题,即程序在相同输入下多次执行可能得到不同的结果.不确定性给并行程序在开发、调试、测试、容错和安全等方面都带来了挑战,严重降低了并行程序的可靠性,阻碍了并行程序的发展.确定性并行技术通过控制并行程序执行个体间的同步、竞争和干扰,使程序的执行结果仅依赖于输入.确定性并行技术能够从根本上解决了目前并行程序存在的诸多问题,提升了并行程序的可靠性,给并行程序的发展带来了新的机遇.文中调查、分析和比较了目前主流的确定性并行技术和方法,分析了弱内存一致性对确定性并行系统的影响,并对未来确定性并行技术的发展趋势做出了展望.

关键词 确定性;并行计算;可靠性;数据竞争

中图法分类号 TP368 **DOI 号** 10.3724/SP.J.1016.2015.00973

Deterministic Multiprocessing

ZHOU Xu LU Kai CHEN Chen

(*College of Computer, National University of Defense Technology, Changsha* 410073)

(*National Key Laboratory of Parallel and Distributed Processing, National University of Defense Technology, Changsha* 410073)

Abstract The executions of parallel programs are nondeterministic due to synchronizations, races and interactions between concurrent executing units such as threads. Nondeterminism indicates that multiple executions of the same program under the same input may produce different outputs. Nondeterminism brings challenges to parallel programs in many aspects such as programming, debugging, testing, fault tolerance and security. As a result, the reliability of parallel programs is greatly reduced, which will hamper the development of parallel computing. Deterministic multiprocessing controls the synchronizations, races and interactions of parallel programs to ensure that the output of a parallel program is only affected by its input. Deterministic multiprocessing is promising to solve the above problems of parallel programs thoroughly. This technique will improve the reliability of parallel programs and brings new opportunities to parallel computing. In this paper, we survey current mainstream deterministic multiprocessing techniques, analyze and compare them. We also analyze how weak memory consistency models affect deterministic multiprocessing systems, and show the prospects of future researches on deterministic multiprocessing.

Keywords determinism; parallel computing; reliability; data race

收稿日期:2014-01-20;最终修改稿收到日期:2014-09-24.本课题得到国家"八六三"高技术研究发展计划项目基金(2012AA01A301,2012AA010901)、新世纪人才基金和国家自然科学基金(61272142,61103082,61402492,61170261,61103193)资助.**周 旭**,男,1985 年生,博士,助理研究员,中国计算机学会(CCF)会员,主要研究方向为并行计算、可靠性和确定性并行技术.E-mail:zhouxu@nudt.edu.cn.**卢 凯**,男,1973 年生,博士,研究员,主要研究领域为操作系统、并行计算和安全.**陈 沉**,男,1985 年生,博士研究生,主要研究方向为程序分析和并行计算.

1 引 言

和传统的串行程序相比，并行程序在给计算性能带来提升的同时，也给程序的开发和维护带来了挑战.并行程序通常由多个并行执行个体(如线程)协同完成一个任务，因此执行个体之间就广泛存在着同步、竞争和干扰的问题，这就导致了并行程序的不确定性，即程序在相同的输入下多次执行，可能产生不同的执行路径和结果.这种不确定性给并行程序在许多方面(如开发、测试、维护、容错和安全等)都带来了新的挑战.

目前，确定性并行技术被认为是应对这个挑战的关键核心技术[1-7].确定性并行技术要求程序在相同的输入下运行时总是能得到相同的执行流程和结果.确定性并行技术的基本思想是控制并行执行个体之间的同步、竞争和干扰，使执行个体之间按照一定的规则和顺序进行交互，从而确保每次执行都能复现这种规则和顺序，使程序即使在不同的环境中执行也能得到相同的结果.在确定性并行技术下，程序的执行结果仅依赖于输入和程序逻辑本身，因此排除了外界环境对于执行结果的影响.

然而目前确定性并行技术的研究还不成熟，存在着诸多问题：例如纯软件实现的确定性并行系统效率普遍低下，可以达到2～10倍的性能开销，这使它们很难在实际应用中被接受[1-2]；而硬件支持的确定性并行系统如今只有模拟结果，在实验室和工业界都不存在真实的硬件确定性系统[1,7].除此之外，确定性并行系统还存在着诸如移植性、稳定性、可扩展性等方面的问题.这些问题制约了确定性并行技术的实用化，亟需解决.

本文分析了引起并行程序不确定性的原因，介绍了确定性并行技术的优势；对目前的确定性并行技术方法做了总结和归类，详细对比了几种主要的确定性并行技术方法，其中也包括我们自己实现的三个确定性算法；并针对弱内存一致性对确定性并行系统的影响做了深入分析.在此基础上，展望了未来确定性并行技术的发展趋势.

2 并行程序的不确定性

并行主要有两种形式，一种是多线程并行，即各个并行执行个体之间共享内存空间；另一种是多进程并行，即各个执行个体之间不共享内存空间，而是通过其他方式(例如管道或者网络)进行通信.然而不管是何种并行形式，都存在着不确定性执行的问题.在本节中，我们将分析引起并行程序不确定性的主要因素及不确定性给程序带来的挑战.

2.1 不确定性因素

在多线程并行中，导致程序不确定性的原因是线程对于共享内存的竞争访问.我们可以将其分为数据竞争和同步竞争两种形式.数据竞争是指并行执行的两个访存操作对同一个内存单元进行访问，它们的执行顺序没有被任何同步语句所限制，并且其中至少一个访存操作为写指令.同步竞争是指两个互斥同步语句同时访问一个同步变量，例如两个加锁操作(lock)同时在加一把锁(L).实际上同步竞争是一种特殊的数据竞争，是线程通过原子指令对同步变量的一种竞争访问.但是因为同步语句的特殊性，有必要将其单列出来.

图1展示了数据竞争和同步竞争引起程序执行不确定性的两段代码.线程T1和线程T2同时进行存款和取款的操作，当两次执行的线程访存交织顺序不同时，代码的最终执行结果就会不同.图1(a)中的代码没有用锁保护，因此并行执行时会产生对共享变量 x 的数据竞争(T1:1和T2:2，T1:2和T2:1，T1:2和T2:2都会分别形成数据竞争).竞争的结果会导致变量 x 最终的值可能为9，10或者11.图1(b)中的代码虽然用锁保护了存款和取款操作的原子性，不会产生对变量 x 的竞争访问，然而两个线程对锁的竞争依然存在，因此不确定性仍然没有消除.如果线程T1先得到锁，就会使客户的存款量达到10以上，因此账户会自动升级为VIP账户.反之，如果线程T2先得到锁，线程T1再执行存款操作时存款量就只有10，因此最终此账户就不会升级为VIP账户.

初始状态：x=10, vip=false;

Thread 1	Thread 2
deposite:	withdraw:
1 $a=x$;	1 $a=x$;
2 $x=a+1$;	2 $x=a-1$;

T1: 1->T1: 2->T2: 1->T2: 2 x: 10
T1: 1->T2: 1->T1: 2->T2: 2 x: 9
T1: 1->T2: 1->T2: 2->T1: 2 x: 11
T2: 1->T2: 2->T1: 1->T1: 2 x: 10

(a)

Thread 1	Thread 2
deposite:	withdraw:
1 lock(l);	1 lock(l);
2 $a=x$;	2 $a=x$;
3 $x=a+1$;	3 $x=a-1$;
4 if(x>10) vip=true;	4 unlock(l)
5 unlock(l);	

T1: lock->T2: lock vip: true
T2: lock->T1: lock vip: false

(b)

图1 并行程序的不确定性

在多进程程序中，同样存在着进程间的交互，因此也可能引发不确定性.例如，MPI消息传递是一

种常见的多进程并行模型.进程通过调用消息发送和接收函数来进行交互.在 MPI 中存在着异步消息传递函数和混杂消息接收函数.异步消息传递函数并不能保证消息传递能够在确定的程序执行点完成,而混杂消息接收函数可以选择接收任意到达的消息,它们都能引起程序执行状态的不确定[8-9].

非并行引起的不确定性.程序中还可能存在其他一些能够引起不确定性的操作,例如对随机数发生器 *rand* 函数的调用.这种不确定性是一种人为的不确定性,是程序逻辑本身所需要的,同时也是一种可控的不确定性.这种不确定性不是由并行所引起的,因此并不是并行程序所固有的,在串行程序中同样存在.由于这种不确定性并不增加程序的开发和维护难度,因此不属于确定性并行技术的研究范畴.

2.2　不确定性的挑战

不确定性给并行程序的开发、维护、测试、容错和安全等许多方面都带来了严峻的挑战.

不确定性对程序开发的挑战.并行程序的开发要比串行程序困难得多.因为线程之间的交织顺序是没有任何限定的,这导致程序员在编写并行程序时不但要考虑程序逻辑,更要注意线程之间所有可能的交织顺序,小心的使用同步语句保证程序并行执行的正确性.这就给普通程序员形成了很高的门槛,即便是熟练的程序员也难免出现各种错误.Lee[4]在其研究中指出,即使对于很小的一段并行算法,经验丰富的程序员也很难一次性写出正确的代码.

不确定性对程序维护的挑战.调试是程序维护的重要方面.不确定性导致并行程序的 bug 不可复现,从而阻碍了调试的进行.传统的最直接的调试方法是:通过多次迭代式的运行程序,追踪 bug 信息,从而一步步逼近 bug 的根源.然而在并行程序中很多 bug 是依赖于特定的线程交织顺序的,这些 bug 因为只有在特定的线程交织顺序下才会出现,因此就具有很强的潜伏性.即便偶尔出现一次,也很难通过重新运行程序来重现 bug,从而导致调试方法失效[1-3,5-6].

不确定性对程序测试的挑战.由于不确定性的影响,并行程序在相同的输入下往往不能得到相同的输出.在并行程序测试时,即使观察到程序在某个输入下得到了正确的输出结果,也不能保证程序在生产环境中在同样的输入下一定能得到同样的结果.因此传统的通过输入对程序进行压力测试的方法在一定程度上会失效.不确定性使得程序的执行不仅依赖于输入的变化,而且还受线程调度等时间相关因素的影响.因此要得到可信的测试结果,就必须对并行程序的交织顺序也进行覆盖.然而线程的交织顺序是不受限制的,这就极大的增加了程序的测试空间,给程序测试带来了挑战[1,5,10].

不确定性对程序容错的挑战.副本容错是一种常用的容错技术,主要使用在一些对可靠性要求很高的领域,经常用来容硬件产生的偶发错.副本容错一般要求同时运行一个程序的多个实例(称为副本),所有运行副本给定同样的输入.在这种情况下即便个别硬件产生了偶发错,只要正常的硬件达到一定的数量,运行在这些硬件上的副本就能得到正确并且一致的输出结果,因此最终我们依然可以得到正确的执行结果.然而不确定性却破坏了副本容错技术在并行程序中的应用.由于不确定性的影响,即使硬件没有产生偶发错,软件仍然有可能产生不一致的输出结果,导致我们不能判断出正确的运行结果[1-3].

不确定性对程序安全的挑战.数据竞争和不确定性导致并行程序面临着一种新的安全漏洞,即并发漏洞[11-13].不同于以往由于输入引起的漏洞,并发漏洞是由特定的线程交织顺序所触发的,不确定性使得这种漏洞具有极强的隐蔽性和不可复现性,从而对漏洞检测和入侵检测等安全领域造成了严峻的挑战.首先,不确定性导致这些安全漏洞很难被现有的漏洞检测工具所发现.其次,漏洞一旦被利用使系统遭到攻击,系统管理员也很难追踪和分析入侵代码行为,不利于漏洞的及时修复.

3　确定性并行技术

确定性并行技术的目标是消除由于并行引起的不确定性,降低并行程序的开发和维护成本,提高并行程序的可靠性,为并行程序的持续发展提供技术支撑.

3.1　相关技术

与确定性并行技术比较相关的两种技术领域是记录-回放技术和数据竞争检测技术.这三种技术的核心都是发现并控制程序中的数据竞争,但是目标和侧重各有不同.数据竞争检测技术侧重于发现程序中的数据竞争;而其他两种技术则侧重于动态地消除数据竞争所引起的不确定性.

数据竞争检测技术通过静态分析或者动态执行

的方式发现可能产生数据竞争的代码位置并报告给程序员,由程序员手动修复数据竞争[14-19].最新的研究还包括了利用数据竞争检测结果自动修复并行错误的技术[20-21].记录-回放技术通过复现数据竞争等不确定性事件来实现确定性执行.记录-回放技术分为两步:首先针对程序在某一输入下的执行进行记录,记录的内容包括所有程序执行中的不确定性事件,主要是数据竞争的顺序;然后在回放阶段根据所记录的日志重放程序的执行,从而确保程序在回放阶段的行为和记录阶段完全一致[9,22-26].与其他两种技术相比,确定性并行技术的功能更为强大,应用更为广泛.确定性并行技术的优势主要体现在以下几个方面.

首先,确定性并行技术支持任意输入下程序的自动确定性执行,应用领域更为广泛.数据竞争检测技术能够检测和自动消除部分数据竞争,却不能实现确定性.这是因为同步竞争也能引起不确定性,图1的例子可以证实这一点.对并行错误的研究同样表明,数据竞争只是引起并行错误的一种主要形式,而不是唯一的形式[27-28].在这种情况下,实现程序的确定性就具有更重要的意义,它可以应用于程序开发、测试和调试等多种领域.记录-回放技术虽然也能实现确定性,但是只能保证程序在特定输入下的确定性,前提是已经记录了程序在该输入下的执行过程.因此,虽然记录-回放技术对程序调试的意义明显,但是却很难应用于开发和测试等领域.

其次,确定性并行技术无需外部日志的支持,使用方便.确定性并行技术实现的是程序内在的确定性,无需外部日志的支持,可以简化系统的部署和使用.与此不同的是,记录-回放技术依赖于外部日志,因此日志和程序必须同时存在才能保证确定性.而维护日志还会产生额外的时间和空间开销,对于某些系统来说,这种日志开销十分显著[25-26,29].

最后,确定性并行技术能够实现首次确定性,即程序在第一次运行时就能保持确定性.而记录-回放技术的首次运行即记录阶段的执行是不确定执行.首次确定性使得确定性并行技术可以解决并行程序在测试和容错等应用领域的困难[10].

然而,确定性并行技术的优势并非说它可以完全代替其他两种技术.事实上,确定性并行技术和其他两种技术的结合会产生更好的效果.例如,利用数据竞争检测技术首先发现程序中的数据竞争,从而减少记录-回放技术或者确定性并行技术的运行时开销[30].

3.2 技术分类

根据实现层次和手段的不同,本文将确定性并行技术分为确定性编程语言、确定性运行时系统、确定性操作系统、确定性编程模型和确定性算法.例如Jade、StreamIt和SHIM属于支持确定性的并行编程语言;Dthreads是一种完全兼容POSIX pthreads的确定性多线程运行时库[6];Determinitor和dOS则属于支持确定性的操作系统[5,31];Grace是一种面向确定性的并行编程模型[32],Blelloch等人[33]则提出了利用基本的可交换操作构建确定性并行算法的理论.确定性并行按照其他标准也有不同的分类方法,例如按照所支持并行级别的不同可以分为线程级确定性系统和进程级确定性系统,按照是否有硬件支持的标准可以分为纯软件确定性系统和硬件支持的确定性系统.而按照实现层次分类的方法最能体现确定性并行系统在技术上的差别,因此本文采用了这种分类法.

在所有的这些确定性技术中,确定性运行时系统在灵活性、移植性、兼容性、通用性等方面都具有优势,是确定性并行领域最重要的一个研究分支.确定性运行时系统是指以编译器加运行时库的方式实现的确定性并行系统.确定性运行时系统简单易用,只需要将原来的并行程序重新编译和链接(个别系统只需要重新链接)即可,可以较好地兼容现有并行程序.确定性运行时系统部署在操作系统和应用程序之间,因此又具有较好的移植性.目前确定性运行时系统是确定性并行系统的主流实现方式.

3.3 技术对比

本节从性能、确定性、可扩展性、兼容性、稳定性和可移植性等方面对现有的主流确定性技术进行对比.注意,这里的稳定性是指系统抵御代码或者输入的微小变化对确定性扰动的能力[34-35].为了能够直观的比较各种确定性技术,我们采取了量化评分的方式.首先我们假设存在理想的确定性系统,它必须在性能和可扩展性方面接近非确定性并行实现(例如接近pthreads库的性能);在确定性方面必须能够实现执行流程和结果的完全一致;在兼容性方面,必须能够保证在不修改程序代码的情况下完全兼容现有并行程序;在稳定性方面,其确定性不受代码或者输入微小扰动的影响;在可移植性方面,必须能够保证确定性在各种硬件平台和操作系统之间能够方便地移植.

我们以理想的确定性系统为基准,对上述每个指标进行量化评分(评分数值为离散的1,2,3).如

果确定性系统在某个指标上达到或接近理想确定性系统的标准，则其在这个指标上的量化评分为 3 分. 如果确定性系统在某个指标上完全没有达到要求，则其量化评分为 1 分. 而如果确定性系统在某个指标上部分达到了要求，其量化评分就为 2 分. 在这个标准上，我们根据文献中报告的数据对现有的主流确定性系统进行了评分，结果如表 1 所示. 其中 pthreads 为标准的 POSIX 线程库，它除了不具有确定性之外，其他方面都是线程级确定性并行系统的理想标准.

表 1　主流确定性并行系统的分类和量化比较

确定性系统	分类		量化比较					
	实现层次	并行级别	确定性	性能	可扩展性	兼容性	稳定性	可移植性
DMP	运行时系统/硬件支持	线程级	3	1	1	3	1	3
Kendo	运行时系统	线程级	2	3	2	3	2	3*
CoreDet	运行时系统	线程级	3	1	2	3*	1	3
RCDC	运行时系统/硬件支持	线程级	3	1	2	3*	1	3
Dthreads	运行时系统	线程级	3	2	2	2	3	3
dOS	操作系统	进程级/线程级	3	1	1	3	1	1
Clavin	运行时系统/硬件支持	线程级	3*	3	3	3	1	1
Determinitor	操作系统	进程级	3	3	2	1	3	1
Tern	运行时系统	线程级	2	2	2	3	3	3*
Peregrine	运行时系统	线程级	3	2	2	3	3	3*
DDOS	操作系统	进程级/线程级	3	1	1	3	1	2
FPDet	**运行时系统**	**线程级**	**3**	**1**	**1**	**3**	**1**	**3**
DMPI	**运行时系统**	**进程级**	**2**	**3**	**2**	**3**	**2**	**3**
RFDet	**运行时系统**	**线程级**	**3**	**3**	**2**	**2**	**2**	**3**
pthreads	运行时系统	线程级	1	3	3	3	3	3

注意，表 1 的主要作用是对比各种确定性系统，方便发现系统的问题，其中的量化评分并不具有绝对意义. 随着技术的改进，一个原本接近理想标准的系统也可能变为完全没有达到理想标准. 因此，这些评分只有在当前技术条件下比较同类型的确定性系统时才有意义. 另外，如无特殊说明，表中的性能评分都是针对算法的软件实现，例如 DMP 算法就存在软件和硬件两种实现方式. 而 Calvin 系统只有硬件实现方式，因此表 1 中只有它的性能评分是针对硬件实现.

表 1 中带星号的数据表示确定性系统在某个指标上达到该评分时受某些非关键因素影响. 例如 Kendo 的可移植性受硬件性能计数器的影响；而 CoreDet 和 RCDC 的兼容性可能会受到弱内存一致性的影响；Cavlin 由于采用硬件实现，一般只能保证整个系统的确定性，而人们一般更加关注单个程序的确定性；Tern 和 Peregrine 的可移植性受所记录的线程调度的影响，如果新平台缺乏原先记录的线程调用，将不能够按照原来的确定性运行程序.

从以上对比可以看出，目前确定性系统或多或少都存在一些问题，尤其是在性能方面. 对于通用确定性系统来说，目前还不存在真实的硬件实现，纯软件实现的方法性能开销又非常大. 此外，确定性并行技术的研究对于进程级并行的支持还不够. 针对这些问题，我们基于现有工作对确定性技术做了一些研究和优化，表 1 中加粗的三项是我们的研究成果. 从整体上讲，目前确定性并行技术离实用化还存在差距，需要做更加深入的研究.

4　确定性技术分析

本节我们通过介绍几种主流的确定性并行方法，深入讨论确定性并行技术的实现细节，分析这些技术的特点和不足，在此基础上分析弱内存一致性对确定性技术的影响.

4.1　轮转法

Devietti 等人最早提出了一种轮转法来实现通用多线程程序的确定性，称之为 DMP[1] 算法 (Deterministic Multiprocessing). DMP 算法按照执行指令数将程序执行分成“轮”，每一轮每个线程只允许执行一定数量的指令，指令执行完则该轮结束. 一轮执行又可以分为并行阶段和串行阶段，在每一轮开始时，程序首先进入并行阶段. 这时只要线程之间的访存不发生冲突，就可以一直执行到该轮结束；一旦检测到线程的访存与其他线程发生冲突，则强制线程暂停执行，等待进入串行阶段解决访存冲突. 当所有线程都因为发生访存冲突或者执行完该轮的指令数而暂停时，程序就会进入串行执行阶段. 在串行阶段线程根据 ID 号的顺序依次轮流执行. 由于并行阶段不存在访存冲突，而串行阶段线程按照确定

的顺序执行,于是就以确定的方式解决了原本的访存冲突,消除了不确定性.

为了检测访存冲突,DMP 将程序内存以"块"为粒度进行划分,每一块设置一个拥有权标志位,标记当前拥有该内存块的线程.在每一轮的并行执行中,线程只能根据内存块的拥有权标记来访问内存:当线程拥有内存块时,该线程就可以对该内存块进行读写访问,其他线程不能对该内存块进行读写操作;当内存块不被任何线程拥有时(内存块的状态为共享态),任何线程可以对该内存块进行读操作,而不能写.内存块的状态在并行阶段中不会发生改变,直到程序进入串行阶段.进入串行阶段后,由于每次只有一个线程对内存进行独占访问,当前执行的线程就可以访问任何内存块,同时根据访存指令修改内存块的状态:当线程写内存块时,内存块变为独占状态,并归该线程拥有,以方便后续该线程对于内存块的读写;当线程读内存块时,内存块的状态变为共享态,以方便其他线程后续对该内存块的读.

DMP 算法虽然能够保证确定性,但是其串行执行的方法却损失了并行度.一种极端的情况是线程因为一条访存冲突指令进入串行阶段,但是串行阶段执行的其他指令都不发生访存冲突.因此,纯软件的 DMP 实现有 2～10 倍的开销.针对这一点我们对 DMP 算法进行了改进,实现了全并行的确定性多线程算法 FPDet[35].FPDet 仍采用轮转法执行多线程程序,但是和 DMP 算法不同,当发生访存冲突时 FPDet 算法并不强制程序进入串行阶段,而是通过在线程之间交换内存块的拥有权来解决访存冲突,从而使没有发生访存冲突的指令可以尽可能的并行执行.这种方法在一定程度上提升了 DMP 算法的性能,但性能开销仍然有 2～8 倍[1,6].

DMP 算法和 FPDet 算法都实现了强确定性和顺序一致性.顺序一致性使它们能够很好地兼容已有程序,运行时系统的实现方式使其具有可移植性.然而,由于需要监控每条访存指令,这两种算法的性能和可扩展性都比较差.DMP 和 FPDet 算法都需要静态配置两个基本的参数:即轮长度 R 和内存块大小 B.轮长度以指令数为计量单位,程序执行 R 条指令为一轮.由于这两个静态配置参数的影响,这两种算法都不具有稳定性.当程序的指令数发生微小变化时,例如程序员插入一条 printf 函数调用,就会导致 printf 调用之后每一轮的指令位置发生变化,造成程序执行结果的巨大变化.同理,由于内存块大小的配置参数,在输入发生微小变化时同样会导致执行结果的巨大变化.因此,静态配置的轮长度和数据块大小等参数使它们实现的确定性很难抵御代码或者输入的微小改变所带来的扰动,因此不具有稳定性.另外,这两个算法的实现复杂度较低,而软件实现方法对访存指令的插桩相对于算法本身开销较大,因此从性能和算法复杂度上考虑这两种算法更适合于硬件实现.

4.2 弱确定性算法

经典轮转法的性能问题很大程度上是由处理数据竞争引起的.Olszewski 等人[3]从另一个角度出发,假设程序中不存在数据竞争,于是提出了弱确定性的概念.所谓弱确定性是指系统只保证程序同步语句的执行顺序是具有确定性的.为了实现弱确定性,Olszewski 等人设计了 Kendo 算法.算法的思想是利用逻辑时钟来对线程的加锁机会排序,由于逻辑时钟的快慢仅依赖于程序代码执行的数量,因此能够保证程序加锁顺序是确定的,而这等价于程序同步语句的执行顺序是确定的.Kendo 算法的关键问题是弱确定性的基本假设在实际情况中很难成立.由于各种原因(程序员的疏忽等),绝大多数多线程程序都存在数据竞争,对于这些程序,Kendo 算法不能直接保证其确定性,而是必须要求程序员首先对所有的数据竞争代码进行加锁,将数据竞争转换为同步竞争,这样才能保证确定性.然而精确的定位和消除程序中的数据竞争本身就是一件非常困难的事情,现有的数据竞争检测工具无法完全解决误报和漏报的问题[36-37].

Kendo 通过牺牲一部分确定性来换取性能的提升.实验显示,它的性能开销只有不到 16%[3].Kendo 算法之所以能够提升性能,主要是因为在弱确定性的需求下系统无需控制每条访存指令,而只需处理同步语句即可.由于同步语句的数量相对于普通访存指令的数量要少很多,因此会极大地降低开销.除此之外,Kendo 算法不改变原有程序的内存一致性模型,因此能够很好地兼容已有程序.Kendo 算法的可扩展性受限于线程对逻辑时钟的等待,即每次同步操作都要等待线程的逻辑时钟变为最小,任何线程的延迟都会导致等待时间的加长,在线程数增加时这种等待开销会急剧增加.利用硬件性能计数器来实现逻辑时钟使得 Kendo 算法的稳定性和移植性都受到一定的影响[38].类似实现弱确定性的系统还包括 Tern.由于 Tern 系统的特点是实现了稳定性,因此我们将在 4.6 节重点介绍.

4.3 弱化内存一致性

为了解决确定性技术的性能问题,学术界提出了

通过弱化内存一致性来提升并行度的方法. Bergan 等人在 DMP 算法的基础上做了进一步改进，实现了 CoreDet[2] 和 RCDC[39] 确定性系统. CoreDet 系统首次使用了 Total Store Order(TSO)弱内存一致性模型来优化确定性系统的性能. CoreDet 使用和 DMP 同样的方法将程序执行分轮，每一轮执行一定数量的指令. 不同的是，在每一轮中每个线程都工作在共享内存的本地副本中. 确切地说，当一个线程需要写一块共享内存时，系统首先为该线程创建这个内存块的本地副本，然后将内容写到本地副本中. 随后该线程对于这块内存的读写都在本地副本中进行. 如果线程只对内存进行读操作就不需要创建本地副本，直接在共享内存中进行. CoreDet 在每一轮结束时会将本地副本中的内存修改合并到共享内存中，然后清除本地副本. 由于本地副本的机制，线程之间就不会存在 R/W 和 W/R 冲突. 而所有的 W/W 型访存冲突都被集中在每一轮结束时的内存修改合并阶段. 在这个阶段，各个线程依次按照 ID 号顺序合并内存修改，因此也可以保证确定性.

在 CoreDet 系统中，由于使用了 TSO 弱内存一致性模型，原本需要在串行阶段执行的冲突代码可以在并行阶段执行. 因此就提升了系统的并行度，对系统可扩展性的提升有着重要的作用. 需要注意的是，在 CoreDet 系统中，每轮的结束阶段仍然有一小段串行执行，这是为了执行原子指令. 原子指令由于其特殊性，必须直接在共享内存上执行. 因此，在遇到原子指令时，线程需要提前结束本轮执行，等所有线程提交了本轮的内存修改之后，按线程号顺序依次执行剩余部分的指令.

RCDC 在 CoreDet 的基础之上进一步弱化了内存一致性. RCDC 使用的内存一致性模型为 DRF(Data Race Free)放松一致性模型[39]. 在 CoreDet 系统中，所有的原子指令是不能并行执行的. 而 RCDC 观察到，有些原子指令是可以并行执行的. 例如两个线程分别执行 lock L1 和 lock L2，由于所加的锁不同，因此可以并行执行而不会引起不确定性. RCDC 通过追踪由于线程同步所形成的 happens-before 关系，对于不存在 happens-before 关系的加锁操作可以在不影响确定性的情况下令其并行执行.

CoreDet 和 RCDC 算法通过弱化内存一致性来提升并行度，其优点是具有强确定性、较好的可扩展性和较好的可移植性. 但是弱化内存一致性可能导致兼容性问题. 如果原本程序的正确性是建立在顺序一致性的基础上，那么弱化内存一致性将可能导致程序不能正确运行. 在性能方面，由于没有改变轮转法使用编译插桩的实现方式，这两种方法的性能几乎没有提升，仍然有 2～8 倍的性能开销[2]. 同时这两个系统采用了和 DMP 算法同样的指令分轮和内存分块方法，因此稳定性也没有改善[34].

Calvin[7] 同样采用了 TSO 弱内存一致性，从算法上来讲和 CoreDet 算法一致，但是完全基于硬件实现. 因此其确定性、性能、可扩展性都比较好. 由于需要特殊的硬件支持，因此它存在比较严重的兼容性问题，即不能兼容已有的二进制程序；另一方面，如果有编译器的支持，Calvin 系统可以兼容具有源码的程序. Calvin 系统的不足是稳定性(仍然由轮长度等参数引起)和可移植性较差，在没有商用硬件的情况下这种可移植性问题尤为严重.

4.4　隔离通信法

Liu 等人[6] 通过完全隔离线程通信的方法开发了 Dthreads 确定性系统. Dthreads 突破了纯软件确定性系统实现手段的瓶颈，通过完全隔离线程的内存空间，有效化解了 CoreDet 系统的编译插桩开销，并成为一段时间内最高效的纯软件确定性系统[6]. 与 CoreDet 等系统采用编译插桩控制访存冲突不同的是，Dthreads 采用了页面保护机制来隔离线程内存空间，从而消除访存冲突. 首先，Dthreads 将多线程程序的线程全部转换为轻量级进程，这样每个线程就可以拥有独立的页表，从而实现对共享内存不同的访问权限. 其次，Dthreads 在线程首次修改共享内存时做 Copy-on-Write，为每个线程生成共享内存的本地副本，从而避免直接的访存冲突. 这些副本以页面为单位组织，称为双胞胎页面(Twin-Page). 在每次需要将本地内存修改提交到共享内存时，通过逐字节比较原始页面和双胞胎页面就可以获得线程的内存修改. Dthreads 所采用的这种控制访存冲突的方法取得了明显的性能提升，主要原因是它消除了对读写指令的编译插桩开销.

需要注意的是，与 CoreDet 等轮转法系统不同的是，Dthreads 每轮的长度不是由执行指令数决定的. Dthreads 规定，线程在遇到任何一条同步指令时结束当前轮的执行. 这样做主要是因为 Dthreads 要和 POSIX pthreads 的使用接口完全兼容，因此抛弃了编译插桩的方法. 这样 Dthreads 就只需要重新实现 pthreads 库函数，而应用程序仅需要重新链接 Dthreads 库来获取确定性. 显然，这种实现方式的劣势是可能造成线程之间在每轮执行中的负载不均

衡.例如,如果两个线程在进入下一个同步语句前执行的指令数差别很大,就会造成其中一个线程等待时间过长.对于那些具有典型不规则同步的程序来说,Dthreads 可能造成较大的性能开销.

Dthreads 系统极大地提升了纯软件确定性系统的性能,使得平均性能开销不超过 2 倍,但是某些程序开销仍然可达 8 倍[40].Dthreads 系统能够提升性能的主要原因有两方面:首先,它采用页面保护机制来隔离线程之间的访存冲突,避免了 CoreDet 等软件算法的插桩开销;其次,它的指令分块粒度较大,极大地减少了线程间的全局同步次数,降低了同步开销.Dthreads 同时具有强确定性、稳定性和较好的可移植性.Dthreads 系统的可扩展性受限于全局同步所引起的负载均衡问题.同时,由于使用了 TSO 弱内存一致性模型,并且内存一致性较 CoreDet 更弱,因此导致了更为严重的兼容性问题,例如它明确不支持 Ad hoc 同步方式[6].

4.5 全局同步问题

以上介绍的强确定性系统存在一个共同的问题:那就是依赖于全局同步来实现确定性.全局同步是指由确定性系统插入到程序执行中的一种额外的同步点,它要求所有线程在这个同步点中暂停执行,以方便系统处理不确定性.全局同步使确定性的实现变得简单,但同时也带来了性能和正确性方面的问题.全局同步问题根源在于它并非程序固有的同步策略,而是由确定性系统引入的额外的同步策略.因此全局同步是没有任何程序语义的,它的存在仅仅是为了实现确定性,和线程之间的通信需求毫无关系.当线程本身的同步策略和确定性系统所引入的全局同步产生冲突时,就有可能产生正确性和性能方面的问题.

为了消除全局同步,我们提出了一种 DLRC 弱内存一致性模型,并在此基础上实现了一种免全局同步的确定性并行算法 RFDet[40].确定性算法的关键是限制数据竞争,在没有全局同步的情况下,我们必须保证能够通过程序本身的同步来限制数据竞争.而根据 C++内存一致性模型,如果两个访存操作之间具有 happens-before 时序关系,后一个操作就必须能够看到前一个操作对内存的更新[41],这是保证 C++程序正确性最基本的内存一致性.因此,为了限制数据竞争,我们就将内存一致性弱化到这个程度,并且保证任何指令仅能够看到和它具有 happens-before 时序关系并且发生在它之前的指令对内存的修改,这就是我们提出的 DLRC 内存一致性模型.根据这个模型,我们就可以阻止所有通过共享内存访问形成的线程间通信,将这些通信全部延迟到线程同步的时候.这样一来,我们就将原本程序中的数据竞争都限制在线程同步点上,从而可以在不引入任何额外同步的前提下处理数据竞争.

DLRC 内存一致性模型的效果是数据竞争的结果仅依赖于同步竞争的结果,因此程序的确定性仅和同步语句的执行顺序相关.在 DLRC 内存一致性模型的基础上,我们通过 Kendo 算法保证同步语句执行顺序的确定性,于是实现了一个高效、免全局同步的确定性系统 RFDet.实验显示 RFDet 的性能比同样是纯软件实现但使用了全局同步的 Dthreads 性能提升了近 1 倍.

和 Dthreads 系统相比,RFDet 系统最大的特点是消除了全局同步.对于具有不规则同步的程序来说,Dthreads 方法的全局同步会引起线程之间的负载不均衡,严重的还会导致死锁,而 RFDet 算法由于完全基于线程原有同步实现确定性,因此更能发挥程序细粒度的并行,从而取得更好的性能.RFDet 系统实现了强确定性,平均性能开销不到 50%[40].此外纯软件运行时系统的实现方式使其具有可移植性.不足之处是 RFDet 系统增加了很多写操作,一定程度上影响了可扩展性.其抗代码扰动的能力,即稳定性,受限于 Kendo 算法中逻辑时钟受代码微小变化的影响.同样,因为采用了弱内存一致性模型,RFDet 不支持 Ad hoc 同步方式,降低了兼容性.

4.6 稳定确定性

DMP 等基于轮转法的确定性系统存在一个问题,即不具有稳定性.对于传统的并行执行模型(如 pthreads)来说,当输入发生微小变化时,只要不是关键数据的改变,程序一般都会走类似的执行路径,偏差不会太大.但是由于确定性系统的干扰,程序的稳定性可能会被破坏,即当程序的输入发生微小变化时,程序执行路径的变化可能会被放大,造成执行结果的显著变化[42-43].也就是说确定性系统在保证确定性的同时,可能将程序的计算过程变成一个混沌系统.Yang 等人提出了稳定确定性的概念,并通过 Tern 算法解决了稳定性的问题[34].Tern 算法会首先对程序执行进行记忆,并以〈输入,调度〉二元组的形式存储在数据库中.在以后的执行中,系统会首先在数据库中依据输入条件匹配调度二元组.注意二元组中的输入是一个输入条件,只要符合这个输入条件就会匹配上,因此实际上代表了一组输入.一旦匹配上,系统就会使用相应的调度去执行程序.否

则，系统以正常的方式执行程序，记录其执行路径和输入条件，并将其存放到数据库中. 由于 Tern 不支持数据竞争(类似于 Kendo)，Peregrine 算法[44]在 Tern 算法的基础上做了进一步改进，使其可以支持数据竞争. Bergan 等人[43]进一步将调度精确化简，保证能够覆盖到所有的输入集合.

稳定性是多线程程序应该具备的一个重要特性[42]，然而由于确定性系统强加的调度限制，程序的稳定性被破坏了. Yang 等人所提出的稳定确定性方法从一定程度上解决了这个问题，但是他们设计的系统复杂度太高，部署和使用不方便，其致命的问题是输入不能匹配调度数据库的情况，这对于商用软件来说是不可接受的. 因此未来有必要对确定性系统的稳定性做进一步研究.

Tern 和 Peregrine 两个系统最大的优点是具有好的稳定性. 这两种方法不改变原有多线程程序的调度方式，而是从原有多线程程序的调度集合中选择其中的一部分作为自己的调度集合. 由于在一定程序上缩小了调度集合，因此这种方法具有较好的稳定性. 其次这两个系统采用纯软件的方法实现，不改变程序原有的内存一致性模型，因此也具有较好的兼容性和可移植性. 在性能方面，两个系统在输入不匹配时性能开销不可容忍，可以达到上百倍[34,43-44]. 由于需要强制线程按照特定的调度执行，可扩展性并不能达到理想标准. 另外，Tern 系统不解决数据竞争问题，因此仅实现了弱确定性.

4.7　确定性多进程

目前大多数确定性并行方法都是针对单个多线程程序，然而多进程程序的确定性也不容忽视. 很多程序都会采用多进程的并行方式来提升安全性，例如 Apache 服务器和 Chrome 浏览器.

Bergan 等人[31]设计了 dOS 宣布支持多进程程序的确定性. dOS 是对 Linux 操作系统的一种扩展，它可以保证 Linux 系统上一组进程的确定性. 每一个确定性的进程组被称为一个 DPG(Deterministic Process Group). 由于有来自于操作系统内核的支持，dOS 可以控制进程间通信，同时屏蔽了来自于操作系统内部状态变化对应用程序的影响. 进程内部由于线程竞争引起的不确定性使用 DMP 算法解决. 进程组与外部其他进程或者进程组的通信接口通过 SHIM 层实现，这个 SHIM 层负责记录外部进程与进程组交互时产生的不确定性事件. 这样就可以通过记录-回放技术重现进程组执行. 最近，Bergan 等人又将 dOS 扩展，实现了可以支持分布式环境的确定性系统 DDOS[45].

这些确定性操作系统功能强大，但因为实现在操作系统级别，因此移植性较差，部署和使用极不方便. 同时，dOS 和 DDOS 的开销问题非常严重[31,45]，这是因为一方面它们采用了开销较大的 DMP 算法，另一方面，它们并没有采用一层抽象级别来屏蔽底层细节，因此需要处理进程之间的各种交互操作，带来较大的开销. Determinitor 也是一个支持确定性的操作系统，它采用了新的编程模型. 在这种编程模型下，线程之间只能通过特定的系统调用进行交互，使线程在正常执行时不会发生访存冲突，因此其性能较好. 该系统另一个优点是程序本身已经定义了所有访存操作的顺序，不需要外部运行时系统的干预，因此程序的稳定性好. 然而新型的编程模型导致在该系统下编程方式的改变，从而造成系统完全不兼容现有的多线程程序[5].

与这些直接支持确定性多进程的方法不同，我们通过在 MPI 编程模型上实现确定性消息传递技术(DMPI)来支持多进程的确定性[8]. DMPI 利用逻辑时钟控制消息的发送和接受，强制异步消息传递操作在确定数量个逻辑时钟之后完成消息传输，同时强制混杂消息接收函数只接收具有最小逻辑时钟的消息，最终消除了这两个不确定性操作对应用程序状态的影响.

DMPI 系统的性能开销主要来自于两个方面，一是实现逻辑时钟所产生的插桩开销，二是消息传递操作额外的等待开销. 由于插桩粒度较大，并且消息传递操作相对于普通指令执行数量很少，因此 DMPI 系统的总体性能开销并不大，平均性能开销仅有 14%. 同时 MPI 是介于操作系统和应用程序之间的运行时系统，对于现有的 MPI 程序具有很好的兼容性和可移植性. DMPI 的可扩展性和稳定性都受限于系统对逻辑时钟的依赖，这是因为逻辑时钟比较会牵扯到所有系统中的进程，具有全局同步的特性. 而逻辑时钟计数结果会受到指令数微小改变的影响. 最后，DMPI 没有实现强确定性，它不支持 MPI 中一些不常用的非确定性操作，如单边通信. MPI 是一种通用的编程模型，其应用十分广泛，因此对 MPI 的确定性支持具有重要的应用价值.

4.8　内存一致性和确定性

在共享内存的多核体系结构中，内存一致性定义了内存修改可见性的延迟程度. Lamport 最早给出了顺序一致性的定义：并行程序的执行结果等价于某种串行执行方式的结果，并且对于每个执行单

位(线程)来说,其执行指令的顺序和串行等价执行中这些指令的顺序完全一致[46].如果用内存修改的可见性来解释,顺序一致性等价于线程对内存的修改会对所有其他线程立即可见.顺序一致性是最直接的一种内存一致性模型,也是程序员最容易接受的内存一致性模型.然而顺序一致性要求每个访存指令都必须等待前一个访存指令到达主存后才能开始,因此其性能较低.为了优化访存性能,现代体系结构一般通过弱化顺序一致性来提升性能.常用的内存一致性模型包括 TSO(Total Store Order)、PSO(Partial Store Order)、RC(Release Consistency)和 LRC(Lazy Release Consistency)等.

弱内存一致性通过延迟线程写操作对其他线程的可见性来提升性能.在 TSO 模型中,本地 CPU 核所写入的值存放在一个写缓冲区中,写缓冲区中的值对远端 CPU 核是不可见的,只有当写缓冲区满或者程序执行了 fence 指令后写缓冲区中的值才会被写入到主存中,这时其他 CPU 核才可以看到这些值.

内存一致性对确定性技术有着重要影响.首先,弱内存一致性能够有效地减少线程间的访存冲突,而确定性系统的开销主要来自于处理线程间的访存冲突,因此弱一致性可以降低确定性系统的开销.例如在顺序一致性模型下,一个线程的写操作 W 是立即可见的,因此就会和其他线程对于同一个内存位置的读 R 或者写 W' 发生冲突,引起不确定性.正是因为这样,DMP 等算法必须控制每一个访存操作,安排这些访存操作的执行顺序,避免不确定性的发生.然而同样的代码在 TSO 模型下就会有所不同,由于线程的写 W 并不是立即可见的,因此即便其他线程有对同一个内存的读 R 或者写 W',它们之间也不会产生冲突.在 CoreDet 系统中,写操作被延迟到每轮执行的末尾才提交到主存,这样就只需在提交内存修改时安排所有写操作的顺序,因此会极大的提升并行度.理论上讲,内存一致性越弱,确定性并行系统需要安排的访存顺序就越少,就越容易获得较高的性能.然而这并不是绝对的,系统的性能还和具体的实现技术相关.除了性能,弱内存一致性还能改善确定性系统的稳定性.例如在 Dthreads 系统中,各个线程对内存的修改只有在线程同步时才会合并到共享内存,因此其确定性并不受输入和代码微小扰动的影响.

弱内存一致性为确定性并行性能优化带来契机的同时也给可编程性和兼容性带来了困难.在多线程编程中,程序员最容易接受的是顺序一致性,程序员在编写程序时会自然认为对一个变量的修改会立即可见.如果确定性系统不能保证程序员的这种期望,就可能造成程序执行的错误.因此弱内存一致性一般都要求程序员增加额外的 fence 指令来强制内存修改必须在某一个点之后可见,这无疑会增加程序员的负担.另一方面,按照顺序一致性模型编写的程序一般不能在弱内存一致性的体系结构中正确执行,更进一步,支持较强内存一致性的程序一般不能在具有较弱内存一致性的体系结构中正确运行.这是因为程序中可能不存在合理的 fence,因此产生了向后兼容的问题.和体系结构面临的问题一样,使用弱内存一致性的确定性并行系统也存在着这种兼容性和可编程性的问题.例如 Dthreads 的弱内存一致性模型就不支持 Ad hoc 同步语句.

即便是这样,弱内存一致性仍然是目前确定性并行技术的主流[47-49].这是因为性能和稳定性是影响确定性并行系统实用性的关键,而可编程性和兼容性可以通过其他方法来弥补.例如,可以设计新的编程模型,使程序员在编写多线程程序时并不直接考虑线程同步,这样就能屏蔽弱内存一致性的问题.另一方面,也可以利用一些自动化的工具来实现对传统代码的兼容.例如 Liu 等人[50]提出了一种自动根据弱内存一致性模型为程序加 fence 的方法,类似的方法也可以用在确定性系统中.

弱内存一致性在确定性多线程的实现中占有重要地位.如图 2 所示,目前确定性并行技术已经探讨了 SC、TSO 等内存一致性模型.我们也在 RFDet 算法的实现中尝试了 LRC 内存一致性.初步结果显示,在 LRC 内存一致性模型下,确定性并行能取得更好的性能,但同时也会将系统的兼容性降低.其他的一些内存一致性模型如 PSO 还没有被任何确定性技术所使用.目前哪种弱内存一致性模型最适合确定性并行还不得而知,很有可能需要为确定性并行设计一种全新的内存一致性模型来同时解决性能和兼容性方面的问题,未来需要在这方面做深入探讨.

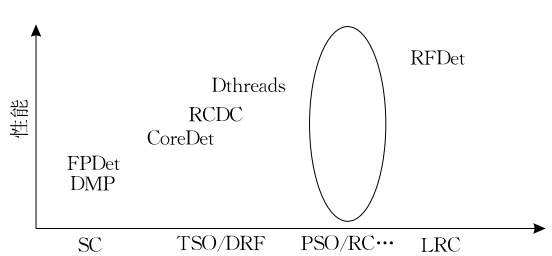


图 2 内存一致性和确定性

5　确定性并行发展趋势展望

确定性并行技术作为一种并行程序的解决方案，上接应用程序，下接体系结构，因此它的发展同时受两方面因素的主导：一是体系结构的发展；二是应用程序的需求. 目前异构和大规模并行是体系结构的两个重要发展趋势；而上层应用程序则要求尽可能的发挥体系结构的性能，同时方便用户编程. 受这些因素的影响，目前确定性并行技术存在着向大规模并行、异构和新型编程模型等方向发展的趋势.

5.1　大规模并行支持

受科学计算和云计算等应用的影响，并行计算正在向大规模并行和分布式并行等方面发展. 针对这一趋势，确定性并行技术不能仅仅停留在支持多线程并行. 在大规模并行计算领域中，由于程序并行度极高，更加容易出现问题，因此更加迫切需要确定性并行技术的支持. 目前国际上已经开始研究面向分布式环境的确定性系统. 然而由于并行度很大，这种系统的开销也会急速上升. 例如，Bergan 等人[45]实现的面向分布式系统的确定性并行系统 DDOS 的性能开销可高达 20 倍，远远不能满足实用需求. 支持大规模并行程序的确定性技术能够应用于云计算和科学计算等领域，例如应用在天河超级计算机中[51]，为大规模并行计算和云计算的可靠性提升提供技术支撑，是未来的一个发展趋势.

5.2　异构体系结构支持

目前异构体系结构也是一个重要的发展趋势，因此确定性并行技术必须能够支持异构体系结构，否则将很难适应未来硬件的发展. 目前 Jooybar 等人已经研究了 GPU 的确定性并行技术[52]. 然而，异构并非只有 GPU 一种形式，其实现层次也可以多元化. 例如在通用的 OpenCL 语言级别上实现确定性，这样便于屏蔽底层硬件的差别，同时具有较好的移植性. 另外，由于异构体系结构中各个执行单元之间的速率和指令集可能存在差别，因此也会给确定性并行技术带来新的挑战.

5.3　确定性编程模型

目前大多数确定性系统都是对现有多线程模型的修补. 由于多线程模型本身的限制，可能导致确定性系统在性能等方面的问题. 因此有些研究者倾向于设计新的编程模型来从根本上改善确定性并行技术. 我们在一些确定性系统中能看到新编程模型的影子，例如 Determinitor 系统[5]，Grace[11] 系统等. 最近，Merrifield 等人[53]又提出了利用版本控制模型来实现确定性的方法，取得了较好的实验效果. 设计全新的编程模型能够有效地避开数据竞争的问题，同时能够让程序员专注于算法逻辑，具有很强的吸引力. 因此，设计高效通用的确定性并行编程模型也是未来确定性并行技术的一个发展趋势.

6　结束语

并行程序的执行具有不确定性，这给并行程序带来了严峻的挑战，已经成为影响并行程序可靠性的关键问题. 确定性并行技术给我们带来了很多期待：通过保证并行程序在相同的输入下总是产生相同的输出，确定性并行技术能够同时解决并行程序在开发、测试、调试、容错和安全等方面的问题. 然而目前确定性并行技术在很多方面，尤其是性能方面还难以满足实用性的需求. 本文介绍了目前主流的一些确定性并行技术方法，包括我们自己实现的三个确定性算法，分析和对比了这些方法的优缺点，讨论了弱内存一致性对确定性并行技术的影响，并对未来确定性并行技术的发展趋势作出了展望.

参　考　文　献

[1] Joseph D, Brandon L, Luis C, Mark O. DMP: Deterministic shared memory multiprocessing//Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems. Washington, USA, 2009: 85-96

[2] Bergan T, Anderson O, Devietti J, et al. CoreDet: A compiler and runtime system for deterministic multithreaded execution//Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems. Pittsburgh, USA, 2010: 53-64

[3] Olszewski M, Ansel J, Amarasinghe S. Kendo: efficient deterministic multithreading in software//Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems. Washington, USA, 2009: 97-108

[4] Lee E A. The problem with threads. Computer, 2006, 39(5): 33-42

[5] Amittai A, Shu-Chun W, Sen H, Bryan F. Efficient system-enforced deterministic parallelism//Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation. Vancouver, Canada, 2010: 1-16

[6] Liu T, Curtsinger C, Berger E D. DTHREADS: Efficient deterministic multithreading//Proceedings of the 22nd ACM Symposium on Operating Systems Principles. Cascais, Portugal, 2011: 327-336

[7] Hower D R, Dudnik P, Hill M D, Wood D A. Calvin: Deterministic or not? Free will to choose//Proceedings of the 17th International Symposium on High Performance Computer Architecture, Washington, USA, 2011: 333-334

[8] Zhou X, Lu K, Wang X, et al. Deterministic message passing for distributed parallel computing. IEICE Transactions on Information and Systems, 2013, E96-D(5): 1068-1077

[9] de Kergommeaux J, Ronsse M, De Bosschere K. MPL: Efficient record/replay of nondeterministic features of message passing libraries//Dongarra J, Luque E, Margalef T eds. Recent Advances in Parallel Virtual Machine and Message Passing Interface. Lecture Notes in Computer Science 1697. Berlin Heidelberg: Springer-Verlag, 1999: 141-148

[10] Bergan T, Devietti J, Hunt N, Ceze L. The deterministic execution hammer: How well does it actually pound nails?// Proceedings of the 2nd Workshop on Determinism and Correctness in Parallel Programming. Newport Beach, California, USA, 2011: 56-63

[11] Yang J, Cui A, Stolfo S, Sethumadhavan S. Concurrency attacks//Proceedings of the 4th USENIX Workshop on Hot Topics in Parallelism (HOTPAR'12). Berkeley, USA, 2012: 15-30

[12] Sender S, Vidergar A. Concurrency attacks in web applications//Proceedings of the Blackhat. Las Vegas, USA, 2008: 56-62

[13] Watson R N M. Exploiting concurrency vulnerabilities in system call wrappers//Proceedings of the 1st USENIX Workshop on Offensive Technologies. Berkeley, USA, 2007: 21-28

[14] Boehm H-J. Position paper: Nondeterminism is unavoidable, but data races are pure evil//Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multi-Core and Many-Core Scalability. New York, USA, 2012: 9-14

[15] Benjamin W, David D, Peter M C, et al. Parallelizing data race detection//Proceedings of the 18th Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013). Houston, USA, 2013: 27-38

[16] Park C-S, Sen K, Iancu C. Scaling data race detection for partitioned global address space programs//Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS 2013). Eugene, USA, 2013: 47-58

[17] Eslamimehr M, Palsberg J. Race directed scheduling of concurrent programs//Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2014). Orlando, USA, 2014: 301-314

[18] Tingting Y, Witawas S, Rothermel G. SimRT: An automated framework to support regression testing for data races// Proceedings of the 2014 International Conference on Software Engineering (ICSE 2014). Hyderabad, India, 2014: 48-59

[19] Wu Zhendong, Lu Kai, Wang Xiaoping, Zhou Xu. Collaborative technique for concurrency bug detection. International Journal of Parallel Programming, 2014, 3(1): 1-26

[20] Jin G, Song L, Zhang W, et al. Automated atomicity-violation fixing//Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. San Jose, USA, 2011: 389-400

[21] Volos H, Tack A J, Swift M M, Lu S. Applying transactional memory to concurrency bugs. ACM SIGARCH Computer Architecture News, 2012, 47(4): 211-222

[22] Veeraraghavan K, Lee D, Wester B, et al. Doubleplay: Parallelizing sequential logging and replay. ACM Transactions on Computer Systems, 2012, 30(1): 1-24

[23] Subhraveti D, Nieh J. Record and transplay: Partial checkpointing for replay debugging across heterogeneous systems// Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems. San Jose, USA, 2011: 109-120

[24] LeBlanc T J, Mellor-Crummey J M. Debugging parallel programs with instant replay. IEEE Transactions on Computers, 1987, 100(4): 471-482

[25] Lee D, Wester B, Veeraraghavan K, et al. Respec: Efficient online multiprocessor replay via speculation and external determinism//Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems. New York, USA, 2010: 77-90

[26] Montesinos P, Hicks M, King S T, Torrellas J. Capo: A software-hardware interface for practical deterministic multiprocessor replay//Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems. Washington, USA, 2009: 73-84

[27] Lu S, Park S, Seo E, Zhou Y. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics//Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, USA, 2008: 329-339

[28] Lu S, Tucek J, Qin F, Zhou Y. AVIO: Detecting atomicity violations via access interleaving invariants//Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems. San Jose, USA, 2006: 37-48

[29] Patil H, Pereira C, Stallcup M, et al. Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs//Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization. Toronto, Canada, 2010: 2-11

[30] Lee D, Chen P M, Flinn J, Narayanasamy S. Chimera: Hybrid program analysis for determinism//Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation. Beijing, China, 2012: 463-474

[31] Bergan T, Hunt N, Ceze L, Gribble S D. Deterministic process groups in dOS//Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation. Vancouver, Canada, 2010: 1-16

[32] Berger E D, Yang T, Liu T, Novark G. Grace: Safe multithreaded programming for C/C++//Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications. Orlando, USA, 2009: 81-96

[33] Blelloch G E, Fineman J T, Gibbons P B, Shun J. Internally deterministic parallel algorithms can be fast. ACM SIGPLAN Notices, 2012, 3(4): 181-192

[34] Cui H, Wu J, Yang J. Stable deterministic multithreading through schedule memorization//Proceedings of the 9th USENIX Conference on Operating System Design and Implementation. Cascais, Portugal, 2010: 12-24

[35] Zhou X, Lu K, Wang X, Li X. Exploiting parallelism in deterministic shared memory multiprocessing. Journal of Parallel and Distributed Computing, 2012, 72(5): 716-727

[36] Petrov B, Vechev M, Sridharan M, Dolby J. Race detection for web applications//Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. Beijing, China, 2012: 251-262

[37] Kasikci B, Zamfir C, Candea G. Data races vs. data race bugs: Telling the difference with portend//Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). London, UK, 2012: 185-198

[38] Weaver V M, McKee S A. Can hardware performance counters be trusted?//Proceedings of the IEEE International Symposium on Workload Characterization. Seattle, USA, 2008: 141-150

[39] Devietti J, Nelson J, Bergan T, et al. RCDC: A relaxed consistency deterministic computer//Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems. Newport Beach, USA, 2011: 67-78

[40] Lu K, Zhou X, Bergan T, Wang X. Efficient deterministic multithreading without global barriers//Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2014). Orlando, USA, 2014: 287-300

[41] Boehm H-J, Adve S V. Foundations of the C++ concurrency memory model//Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation. Tucson, USA, 2008: 68-78

[42] Yang J, Cui H, Wu J, et al. Determinism is not enough: Making parallel programs reliable with stable multithreading. Communications of the ACM, 2013, 57(3): 58-69

[43] Bergan T, Ceze L, Grossman D. Input-covering schedules for multithreaded programs//Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. Indianapolis, USA, 2013: 677-692

[44] Cui H, Wu J, Gallagher J, et al. Efficient deterministic multithreading through schedule relaxation//Proceedings of the 23rd ACM Symposium on Operating Systems Principles. Cascais, Portugal, 2011: 337-351

[45] Hunt N, Bergan T, Ceze L, Gribble S D. DDOS: Taming nondeterminism in distributed systems//Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems. Houston, USA, 2013: 499-508

[46] Adve S V, Gharachorloo K. Shared memory consistency models: A tutorial. Computer, 1996, 29(12): 66-76

[47] Cui H, Simsa J, Li H, et al. Parrot: A practical runtime for deterministic, stable, and reliable threads//Proceedings of the 24th ACM Symposium on Operating Systems Principles. Farmington, USA, 2013: 388-405

[48] Nguyen D, Lenharth A, Pingali K. Deterministic Galois: On-demand, portable and parameterless//Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems. Salt Lake City, USA, 2014: 499-512

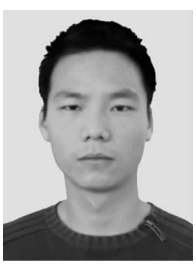
[49] Segulja C, Abdelrahman T S. What is the cost of determinism?//Proceedings of the Workshop on Determinism and Correctness in Parallel Programming. Salt Lake City, USA, 2014: 26-30

[50] Liu F, Nedev N, Prisadnikov N, et al. Dynamic synthesis for relaxed memory models. ACM SIGPLAN Notices, 2012, 47(6): 429-440

[51] Yang X-J, Liao X-K, Lu K, et al. The TianHe-1A supercomputer: Its hardware and software. Journal of Computer Science and Technology, 2011, 26(3): 344-351

[52] Jooybar H, Fung W W, O'Connor M, et al. Gpudet: A deterministic GPU architecture//Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, USA, 2013: 1-12

[53] Merrifield T, Eriksson J. Conversion: Multi-version concurrency control for main memory segments//Proceeding of the 8th ACM European Conference on Computer Systems. Prague, Czech Republic, 2013: 127-139



ZHOU Xu, born in 1985, Ph. D., assistant professor. His research interests include parallel computing software reliability and deterministic multiprocessing.

LU Kai, born in 1973, Ph. D., professor, Ph. D. supervisor. His research interests include operating system, parallel computing and security.

CHEN Chen, born in 1985, Ph. D. candidate. His research interests include program analysis and parallel computing.

Background

Multicore processors have taken the place of uniprocessors, and become today's mainstream CPUs. In the future, CPU will accommodate more cores, which will lead us to the many-core era. However, as parallelism increases, the cost for software development and maintenance is also increased obviously. Due to the factors such as data races, synchronization races and message races, a parallel program may produce different results in different executions. This nondeterminism is the natural property of parallel programs, which makes many tasks in parallel programs—such as programming, debugging, testing, intrusion analysis and fault tolerance—much more difficult than their sequential counterparts.

To solve this problem, researches proposed the technique of deterministic multiprocessing recently. The ultimate aim of deterministic multiprocessing is to ensure that the execution of a parallel program always produce the same output if it is provided with the same input. As a result, the complexity of parallel program executions will be simplified to the level of sequential programs, which will greatly reduce the cost of parallel programming.

This paper introduces the research field of deterministic multiprocessing. We start by introducing the sources of nondeterminism in parallel programs, and comparing deterministic multiprocessing with other similar techniques. Then we discuss several mainstream deterministic multiprocessing techniques and compare their differences. After that, we analyze the impact of weak memory consistency on deterministic multithreading. Finally, we foresee the future directions in the field of deterministic multiprocessing.

This work is partially supported by the National High Technology Research and Development Program (863 Program) of China under Grant Nos. 2012AA01A301 and 2012AA010901, by the Program for New Century Excellent Talents in University and by the National Natural Science Foundation of China under Grant Nos. 61272142, 61103082, 61402492, 61170261 and 61103193.