一种基于扩展数据流分析的 OpenMP 程序应用级检查点机制

富弘毅 丁 滟 宋 伟 杨学军

(国防科学技术大学并行与分布处理国家重点实验室 长沙 410073)

(国防科学技术大学计算机学院 长沙 410073)

摘 要 随着多核处理器体系结构在高性能计算领域日益广泛的应用,面向共享存储并行程序的容错问题成为研究的热点.近年来,检查点技术已经成为该领域占主导地位的容错机制.目前已有一些针对 OpenMP 程序检查点技术的研究工作,但其中绝大多数解决方案都依赖于特殊的运行时库或硬件平台.该文提出一种编译辅助的 OpenMP 应用级检查点,它是一种平台无关的方案,通过面向 OpenMP 的扩展数据流分析选择那些"必需"的变量保存到检查点映像,从而降低容错的开销,同时通过运行一种非阻塞式的协议维护检查点的全局一致性.文章讨论了该机制的各个关键问题,并通过实验评测以及与同类工作的比较,表明了该文所提出的检查点机制在容错性能方面的优势.

关键词 容错;共享存储;OpenMP;应用级检查点;数据流分析

中图法分类号 TP302 **DOI 号**: 10.3724/SP.J.1016.2010.01809

An Application-Level Checkpointing Based on Extended Data Flow Analysis for OpenMP Programs

FU Hong-Yi DING Yan SONG Wei YANG Xue-Jun

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

(*School of Computer, National University of Defence Technology, Changsha* 410073)

Abstract As the wide application of multi-core processor architecture in the domain of high performance computing, fault tolerance for shared memory parallel programs becomes a hot spot of research. For years, checkpointing has been the dominant fault tolerance technology in this field. Recently, a few research works regarding checkpointing for OpenMP programs have been proposed. However, most of the approaches depend on special libraries or hardware platforms. This paper proposes a compiler-assisted application level checkpointing for OpenMP programs. It is a platform-independent scheme, and through the extended static data flow analysis, it automatically chooses those 'must-be-saved' variables to save in the checkpoint image, to reduce the overhead. It also maintains the global coherence of checkpoints by running a non-block protocol. In this paper, the key issues in the approach are discussed in detail, and the experimental result and the comparison with similar works show the proposed approach achieves promising performance.

Keywords fault tolerance; shared memory; OpenMP; application level checkpointing; data flow analysis

---

收稿日期:2010-08-22.本课题得到国家自然科学基金(60921062,61003087)与国家"八六三"高技术研究发展计划项目基金(2009AA01Z102)资助.**富弘毅**,男,1978 年生,博士研究生,助理工程师,主要研究方向为并行与分布式计算、容错、科学计算. E-mail: fool_20022004@yahoo.com.cn.**丁 滟**,女,1977 年生,博士研究生,助理研究员,主要研究方向为系统安全、分布式计算、可信计算.**宋 伟**,男,1982 年生,博士研究生,主要研究方向为并行与分布式计算、容错、科学计算.**杨学军**,男,1963 年生,教授,博士生导师,主要研究领域为并行体系结构、并行操作系统和并行编译.

1 引 言

近年来,高性能计算机系统的规模和性能都有极大的增长.根据最新的 Top500[1] 排名,目前已经有 3 台超级计算机系统的处理器/处理器核突破了 20 万个.硬件的超高复杂度使可靠性成为一个非常严峻的问题,大规模并行系统的平均无故障时间(mean time between failures)下降到若干小时这个量级.例如,IBM/LLNL ASCI White 系统的实际运行状况表明其平均无故障时间仅为 40h 左右[2],而 Google Cluster 大约每 36h 报告一次结点失效[3].同时,科学计算程序的数据量、计算复杂度和执行时间也在不断增长,其中某些程序,例如蛋白质折叠这样的程序,往往需要数月的执行时间.为了填补系统的低可靠性和应用的长执行时间之间的鸿沟,必须采用适当的容错技术.

随着越来越多的高性能计算机系统开始使用多核处理器进行构建,OpenMP 共享存储编程模型被更加广泛地用于开发并行性,大量的科学计算程序使用 OpenMP 或 OpenMP/MPI 混合编程方式编写.因此,针对 OpenMP 并行程序展开容错技术研究具有相当重要的意义.

本文提出一种编译辅助的、平台无关的半自动化 OpenMP 应用级检查点机制.所谓"半自动化",是指我们提出的技术可以根据用户在程序中指定的若干个候选的检查点位置将程序转换成具有应用级检查点机制的容错程序.与同类工作相比较,本文的创新点在于:

(1)完全地应用级实现,与线程库、操作系统及硬件完全无关;

(2)通过对原始程序的静态数据流分析尽可能地减少检查点保存的数据量,降低检查点保存和恢复开销;

(3)采用非阻塞式检查点保存机制,消除因额外同步引入的运行时开销,同时采用一种非阻塞式一致性协议保证检查点的全局一致性;

(4)根据数据流分析结果和通过 profiling 得到的一些程序运行时信息,可以在程序中优化地选取一些候选位置进行检查点保存,目的在于保证检查点时间间隔满足要求的前提下,使检查点的总数据量尽可能地小.

本文第 2 节简要回顾本领域的相关研究工作;第 3 节提出针对 OpenMP 程序的扩展数据流分析方法,作为检查点机制的理论基础;第 4 节介绍检查点机制并就若干关键问题展开详细讨论;第 5 节介绍一种非阻塞式检查点一致性协议;第 6 节讨论检查点位置的优化选择问题;在第 7 节中给出性能评测以及与同类工作的比较;最后对全文进行总结.

2 相关研究工作

目前针对 OpenMP 程序的容错方法主要可以分为两大类:第一大类着重于提供系统级的解决方案,例如,SaftyNet[4] 和 ReVive[5] 使用专门的硬件来记录本地存储内容的变化以及结点间通过共享存储进行的消息传递,以支持系统计算状态的保存和恢复.这种硬件实现的方案优势在于其较低的时间开销和用户透明性,而其缺陷在于对硬件平台的强依赖性,使其无法得到广泛的应用.文献[6]是一种软件实现的方案,它对 cache 一致性协议进行了扩充,通过定位共享页面的位置来实现全局计算状态的保存.这类解决方案同样依赖于特定的平台.第二大类研究工作注重跨平台的应用能力,提供应用级解决方案.Greg 等人提出了编译辅助的应用级检查点机制[7],并将其集成在 C3 容错编译框架[8-9] 中.C3 能够以源到源转换的方式将普通的 OpenMP 程序转换成嵌入了检查点机制的容错程序,同时通过运行一种阻塞式的检查点一致性协议实现全局计算状态的一致性,但该研究工作尚未针对降低检查点数据量提出实质性的方案.

3 面向 OpenMP 程序的并行数据流分析方法 DAO

3.1 研究动机

减小检查点映像文件对于改善检查点机制的性能至关重要,而这一目标可以通过选择那些必须被保存的程序变量来实现.其依据在于,对于多数程序来说,只需要保存其程序空间中的一部分变量就可以实现正确的计算状态恢复.

在先前的研究工作[10] 中我们有这样的结论,在程序中任意给定的位置 p 上,在进行检查点保存时,变量 x 若必须被保存,则 x 应该满足下面两个条件:

① 程序在 p 之后对 x 的第一次访问是一次引用,并且,

② x 在 p 之前的某个位置上被定值.

也就是说，在位置 p 上进行检查点保存时，所有在 p 处“活跃”的变量都应该被保存.

程序任意点处的活跃变量集合都是其程序空间中全部变量集合的一个子集，因此仅保存活跃变量到检查点中可以有效地降低检查点数据量，从而减少检查点的时间和空间开销.

3.2　基于 OpenMP 并行控制流图的扩展数据流分析

在传统的编译技术中，使用一组数据流方程求解程序中一个基本块入口处的活跃变量集合. 当将这一方法运用于 OpenMP 并行程序时，可能导致错误的分析结果.

OpenMP 并行程序的运行遵从 fork-join 模式，其中的串行区由主线程执行，并行区由一组线程(包括主线程)并行执行. 在程序运行的这段时间内，从线程执行的只是程序中被包含在并行区内的那些代码，其操作的数据集合中一部分是与其它线程共享的数据，另一部分是私有访问的数据，并且这两部分之间存在数据交互而引入新的引用定值关系，这是传统的数据流分析方法不能处理的.

针对这一问题，我们提出一种扩展的数据流分析方法，称为 DAO(Data-flow Analysis for Openmp)，利用 OpenMP 并行控制流图，并基于 OpenMP 程序中变量的数据作用域属性进行活跃性分析.

首先我们给出 OpenMP 并行控制流图的定义.

一段串行程序所对应的串行控制流图是一个有向图 $G=\langle V,U\rangle$，其中 V 是结点集合，每个结点代表程序段中一个基本块；U 是弧集合，若对于任意 v_1，$v_2\in V$，程序中描述了 v_1 到 v_2 的转移关系，则弧 $\langle v_1,v_2\rangle\in U$. 该程序中任一代码段 s 对应的串行控制流图是 G 的一个子图，记为 G_s.

OpenMP 程序是由串行区和并行区交替构成的代码序列，其中并行区是由编译指导语句 OMP PARALLEL 和 OMP END PARALLEL 所包围的代码段，而串行区是指未被包含在其中的代码段. OpenMP 程序遵循类似于 fork-join 的并行模型，程序以一个线程(参与执行 OpenMP 程序的任一执行实体被称为一个线程)开始执行，该线程称为主线程. 执行过程中，当主线程遇到一个并行区，则派生 n 个线程，构成一个线程组，并行执行该并行区中所包含的代码. 当该并行区执行完毕后，除主线程外，线程组中其它线程都将退出执行.

设一个 OpenMP 程序包含 u 个并行区，其中第 i 个并行区由 T_i 个线程执行($1\le i\le u$). 设该程序中所有串行区对应的串行控制流子图集合为 $G_S=\{G_S^1,G_S^2,\cdots,G_S^{u+1}\}$，其中 G_S^i 表示第 i 个串行区对应的串行控制流子图($1\le i\le u+1$)；设第 i 个并行区所对应的控制流子图集合为 G_P，且 $G_P=\{G_P^1,G_P^2,\cdots,G_P^u\}$，则基于前述的 OpenMP 并行机制，该程序的 OMP-CFG，记为 $\widetilde{G}$，可表示为三元组 $\langle\widetilde{G}_S,\widetilde{G}_P,U_P\rangle$，其中：

(1) $\widetilde{G}_S$ 描述主线程执行串行区的控制流，其中包含串行区对应的串行控制流子图，即 $\widetilde{G}_S=G_S$；

(2) $\widetilde{G}_P$ 描述线程组执行并行区的情形，其中包含所有并行区在各个线程上被执行的代码对应的串行控制流子图，记第 i 个并行区在线程 t($1\le t\le T_i$)上的控制流子图为 $\widetilde{G}_P^i[t]$，则 $\widetilde{G}_P^i[t]\in G_P^i$；

(3) U_P 为一类特殊的弧，描述进入并行区时创建多个线程以及在离开并行区时线程退出执行的特殊控制流. U_P 包含所有从 G_S^i 的出口结点指向 $\widetilde{G}_P^i[t]$ 的入口结点的弧，以及所有从 $\widetilde{G}_P^i[t]$ 的出口结点指向 G_S^{i+1} 的入口结点的弧，其中 $1\le i\le u$，$1\le t\le T_i$.

在 OpenMP 的并行区中，一般包含有任务共享结构，用于将其中的计算任务并行划分到参与执行的多个线程上. 任务划分的最小单位称为一个任务单元(Unit of work). OpenMP 要求一个任务单元必须是一个结构化块，且任务单元之间不存在任何依赖关系.

若并行区 r 中包含一个任务共享结构 s，分别将 r 中包含在 s 中的代码和 s 外的代码记为 r_s 和 r_s. 则对于任意两个线程 t_1 和 t_2，$\widetilde{G}_P^r[t_1]$ 和 $\widetilde{G}_P^r[t_2]$ 中 r_s 所对应的串行控制流子图相同，但 r_s 所对应的串行控制流子图是否相同，取决于 s 的类型. OpenMP 定义了如下 3 种任务共享结构：

(1) OMP DO. 该结构仅包含一个循环，并对该循环进行并行划分，循环的一次迭代或若干次连续的迭代为一个任务单元；

(2) OMP SECTIONS. 该结构包含若干个由 OMP SECTION 定义的结构化代码块，每一个 OMP SECTION 结构为一个任务单元；

(3) OMP WORKSHARE. 该结构包含若干特定类型的语句(如赋值语句，内建过程的调用语句，FORALL 语句等等)，每一条语句是一个任务单元.

若 s 是一个 OMP DO 结构，则每个线程都执行相同的循环体，只是遍历的循环迭代空间不同. 因此，对于任意两个线程 t_1 和 t_2，$\widetilde{G}_P^r[t_1]$ 和 $\widetilde{G}_P^r[t_2]$ 中 r_s 所对应的串行控制流子图相同，因此 $\widetilde{G}_P^r[t_1]=\widetilde{G}_P^r[t_2]$. 图 1 给出了一个典型的包含 OMP DO 结构 OpenMP 并行区及其对应的 OMP-CFG，其中结点

v_0 和 v_5 构成串行区的串行控制流子图 $\widetilde{G}_P^i[t]$，而结点 v_1，v_2，v_3 和 v_4 构成每个线程上串行控制流子图 $\widetilde{G}_S^i[t](1\leq t\leq T)$，虚线表示 U_P 中的弧.

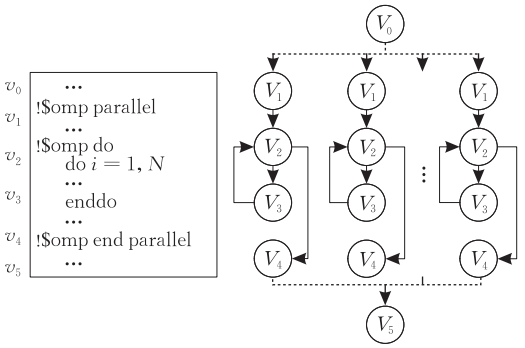


图 1 包含 OMP DO 结构的 OpenMP 并行区及其对应的 OMP-CFG

若 s 是一个 OMP SECTIONS 或 OMP WORKSHARE 结构，则对于任意两个线程 t_1 和 t_2，$\widetilde{G}_P^r[t_1]$ 和 $\widetilde{G}_P^r[t_2]$ 中 r_s 所对应的串行控制流子图是不相同的. s 可以表示为一个任务单元集合，记为 $\{e_1, e_1, \cdots, e_m\}$. 若 s 由 T 个线程执行，则 s 的任一包含 T 个元素的划分 $\Gamma(s)=\{\gamma_1, \gamma_1, \cdots, \gamma_r\}$ 称为 r_s 的一个并行划分，且 γ_i 由线程 $t(1\leq t\leq T)$ 执行. 设 γ_i 中包含 w 个任务单元，分别记为 $e_i^1, e_i^2, \cdots, e_i^w$，则 $\widetilde{G}_P^r[t]$ 中 r_s 对应的串行控制流子图为 $\langle\{v_i^1, v_i^2, \cdots, v_i^w\}, \{\langle v_i^1, v_i^2\rangle, \langle v_i^2, v_i^3\rangle, \cdots, \langle v_i^{w-1}, v_i^w\rangle\}\rangle$，其中 v_i^j 是 e_i^j 所对应的控制流图结点$(1\leq j\leq w)$.

图 2 给出了包含 OMP SECTIONS/OMP WORKSHARE 结构的 OpenMP 并行区及其对应的 OMP-CFG，为了便于表述，其中假设线程数 T 与任务单元数 m 存在关系：$w=m/T$，且任务单元按照书写顺序依次被分配给所有线程.

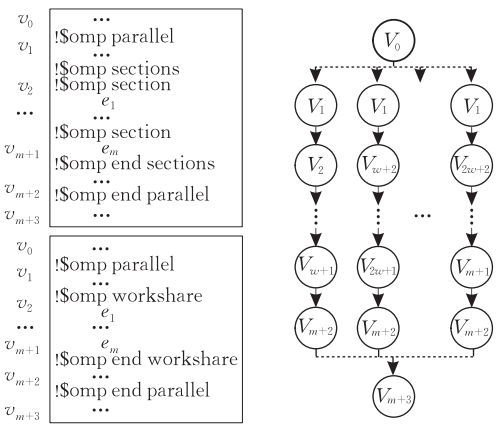


图 2 包含 OMP SECTIONS/OMP WORKSHARE 结构的 OpenMP 并行区及其对应的 OMP-CFG

按照上述定义，一个使用 N 个线程执行的 OpenMP 程序的并行控制流图 G 可以分割为 N 个子图，记为 $G_0, G_1, \cdots, G_{N-1}$. 其中，$G_0$ 描述主线程的执行，而 $G_k(1\leq k<N)$ 描述从线程的执行. 一个典型的 OpenMP 程序及其在 4 个线程上执行时的并行控制流图如图 3 所示.

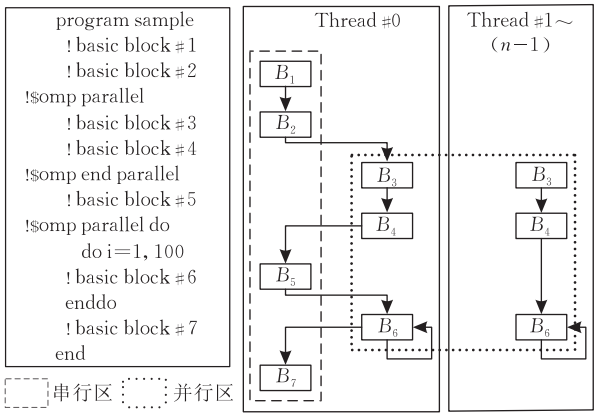


图 3 OpenMP 并行控制流图

如果一个共享变量 x 在并行区 P 中被用作某种类型的私有变量，那么在进入 P 时，会生成并初始化 x 的若干私有副本. 与此同时，x 的共享版本中所包含的值可能会传播到私有副本中. 另一方面，在离开 P 时，其中一个私有副本的值可能会传递给 x 的共享版本. 因此在这两个过程中，存在潜在的对共享版本和私有副本的赋值操作，我们称这些由潜在赋值操作引发的定值-引用关系为隐式引用和隐式定值.

为处理隐式引用和隐式定值，我们在并行控制流图上进行活跃变量分析之前，首先对流图做一次变换，根据每个并行区中变量的数据作用域属性，将其语义中隐含的赋值操作显式地添加到控制流图中该并行区的某些基本块中，使隐式定值和隐式引用能够被包含到这些基本块的 Use 集合和 Def 集合中. 篇幅所限，在这里我们使用一张表来描述如何在一个并行区 P_i 中添加关于变量 x 的潜在赋值操作，如表 1 所示. 其中为了区分 x 的共享副本和私有副本，使用其下标形式 x_i 替换其 x 作为私有副本的所有出现，而下标的值表示 x 在不同并行区中可能具有的不同私有副本.

变换后的 OpenMP 并行控制流图可以用于进行准确的活跃变量分析. 分析过程分别在 G_0 和 G_k 上进行，在 G_0 上可以求解主线程任意一点的活跃变量集合，而在 G_k 上可以求解从线程上任意一点的活跃变量集合. 值得注意的是，在 G_k 中不包含串行区中的任何基本块，所以在其上求解活跃变量集合时，

表 1　赋值操作添加规则

x 在 P_i 中的属性	添加位置	添加操作		
THREADPRIVATE	P_i的入口处	如果 x 在 COPYIN 子句中		$x_i=x$
		x 不在 COPYIN 子句中	$i=1$	$x_i=0$
			$i>1$	$x_i=x$
	P_i的出口处	$x=x_i$(仅在 G_0中添加)		
PRIVATE	P_i的入口处	$x_i=0$		
FIRSTPRIVATE	P_i的入口处	$x_i=x$		
LASTPRIVATE	P_i的出口处	$x=x_i$(仅在 G_0中添加)		

也不会考虑共享变量在串行区中的引用和定值，因此关于共享变量的分析结果是不准确的. 但实际上，共享变量的活跃性在 G_0上已经得到准确的分析，因此可以在 G_k上分析得到的活跃变量集合中删除所有的共享变量.

4　基于 DAO 的 OpenMP 应用极检查点机制

4.1　概　述

在对 OpenMP 并行程序进行静态数据流分析的基础上，我们提出一种编译辅助的 OpenMP 应用级检查点技术，用户只需要在一个 OpenMP 程序中任意指定一些可能需要进行检查点保存的位置(称为候选检查点位置)，就可以按照我们设计的方法将该程序转换成具有应用级检查点保存和恢复机制的容错程序. 并且，转换后的程序可以使用通用编译器和标准 OpenMP 库进行编译链接，生成目标代码，具有很好的可移植性. 我们所提出的应用级检查点机制如图 4 所示.

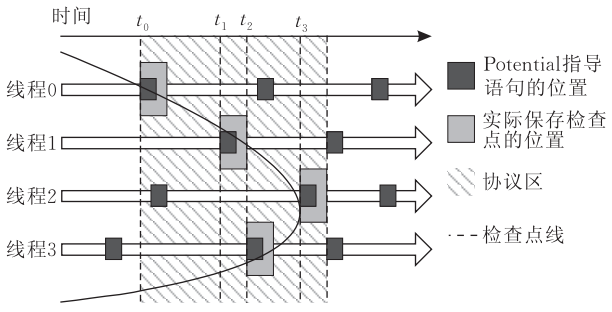


图 4　OpenMP 应用级检查点机制

首先，在程序中由编译指导语句!$PotentialCKPT 指定了若干个候选检查点位置. 程序运行过程中，当某个线程运行到一个 PotentialCKPT 位置时，该线程检查下列两个条件是否满足，如果满足其一则进行检查点保存：

(1)若上一次保存检查点是在 t_1时刻，而运行到当前的候选检查点位置是在 t_2时刻，且 $t_2-t_1-T_{itvl}<\varepsilon$，其中 T_{itvl}是与系统平均无故障时间 MTBF 有关的一个量(其取值将在第 6 节中讨论)；

(2)若当前已经有其它的线程开始进行检查点保存.

在绝大多数情况下，各个线程不会在同一时刻进行检查点保存，总是存在一个先后顺序. 我们称 n 个线程中第一个决定进行检查点保存的线程为本次检查点的发起者. 若发起者在 t_0时刻开始进行检查点保存，而线程组中最后一个保存检查点的线程在 t_{n-1}时刻完成保存，其余 $n-2$ 个线程分别在 t_1，t_2，…，t_{n-2}时刻保存检查点，那么我们称 t_0到 t_{n-1}这段时间为本次检查点的协议期(Protocol Phase)，称图 4 中各个线程保存检查点时刻的连线为检查点线.

用户在程序中指定的检查点候选位置之间不应包含太多的计算任务，以防止出现两个检查点位置之间时间间隔过长，而导致程序无法从一个检查点执行到下一个检查点，而不断地因为故障的发生而回滚的情形. 实际上，用户应该尽可能密集地添加检查点候选位置，而是否要在候选位置上真正进行检查点，则在程序运行时通过上述条件来判断.

如果某个检查点候选位置位于 OMP SECTIONS、OMP WORKSHARE 或 OMP SINGLE 结构中，那么在程序执行过程中并不是所有的线程都会遇到这个检查点候选位置，因此在我们目前的检查点机制中，这种检查点候选位置的出现被认为是不合法的. 实际上，这只是一种保守的策略，我们将在后续的工作中对这一问题进行进一步的分析并给出相应的解决方案，在本文中暂不讨论.

此外，如果检查点候选位置位于由 omp ordered 和 omp critical 指导语句或 omp set lock 等锁例程定义的临界区中，那么按照我们的检查点机制在该检查点位置上进行检查点保存时可能会出现死锁的情形. 为了避免这种情况的发生，我们必须将位于上述几种临界区中的检查点位置移动到临界区之后. 但是，对于用户自定义的基于 spin lock 的临界区，通过编译时静态分析对其进行准确的识别是非

常困难的，所以我们的方法目前不能够有效地处理这种情况.

4.2 变量的保存与恢复

我们所提出的检查点技术将检查点位置上的活跃变量集合作为检查点保存的数据集合. 按照 DAO 的基本思想，每个检查点位置上有两个活跃变量集合，一个是主线程上的活跃变量集合，其中包括所有的共享变量和主线程的私有变量，该集合只在主线程上保存；另一个是从线程上的活跃变量集合，其中包括线程的私有数据，该集合应该在其所属的从线程上保存.

在进行检查点保存时，对于需要保存的活跃变量集合，我们将其中每个变量以二进制形式依次写入检查点映像文件. 相应地在恢复时，按照写入时的顺序依次读出各个活跃变量的值. 并且在这两个过程中，需要对下面两种情形进行特殊处理.

(1) 如果共享变量 x 在某并行区被定义为私有变量，那么在该并行区内的检查点位置上，活跃变量集合中可能同时包含 x 的共享副本和 x 的私有副本. 在这种情况下，x 的私有副本的值的保存与恢复不必特殊处理，但必须在进入并行区之前将 x 的值写入检查点文件中的一个特殊位置上，此时写入的是 x 的共享副本的值. 而在恢复时，在并行区结束后再从文件中的相应位置上读取 x 的共享副本的值.

(2) 如果变量 x 具有 FIRSTPRIVATE 属性，那么对于两个相邻的并行区来说，如果 x 不在第二个并行区的 COPYIN 子句中，那么在进入该并行区时，x 应该具有第一个并行区结束时的值. 但是如果一个检查点位置位于两个并行区之间，那么只有线程 0 会将该值保存下来. 对于这一问题，我们的解决方案是将除线程 0 外的所有线程上 x 的私有副本在第一个并行区执行完毕时保存到一个专用的位置. 在恢复时，执行一个带有 $COPYIN(x)$ 子句的空并行区，目的是在各个线程上生成 x 的私有副本并使其具有先前保存过的值.

因此检查点映像文件应该包含两部分，分别称为常规区和保留区. 其中，保留区在上述情况中专门用于保存变量的共享版本或私有副本的值.

图 5 显示了上述变量保存与恢复方法的一个示意的代码实现，其中图 5(a)给出了一个包含两个并行循环的示意 OpenMP 程序中，其中指定了两个候选检查点位置；图 5(b)和图 5(c)分别显示了为了在这两个检查点位置上实现变量保存和恢复而加入的代码(粗体部分).

```
    ...
    integer x, y
    common /tp/ y
    threadprivate tp
    ...
!$omp parallel private(x)
    ...
!$PotentialCKPT
    ... = x
!$omp end parallel
    ...
!$PotentialCKPT
    ... = x
!$omp parallel do
    ...
    ... = y
    ...
!$omp end parallel
    ...
```

(a)

```
    ...
    integer x, y
    common /tp/ y
    threadprivate tp
    ...
    ! save x to the reserved
    ! region in CKPT file #1
!$omp parallel private(x)
    ...
    ! save x to the regular
    ! region in CKPT file #1
    ... = x
    ! save y to the reserved
    ! region in CKPT file #2
!$omp end parallel
    ...
    ! save x the regular
    ! region in CKPT file #2
    ... = x
!$omp parallel do
    ...
    ... = y
    ...
!$omp end parallel
    ...
```

(b)

```
    ...
    integer x, y
    common /tp/ y
    threadprivate tp
    ...
!$omp parallel private(x)
    ...
    ! restore x from the regular
    ! region in CKPT file #1
    ... = x
!$omp end parallel
    ! restore x from the reserved
    ! region in CKPT file #1
    ...
    ! restore x from the regular
    ! region in CKPT file #2
    ... = x
!$omp parallel do
    ! restore y from the reserved
    ! region in CKPT file #2
    ...
    ... = y
    ...
!$omp end parallel
    ...
```

(c)

图 5 变量保存与恢复的示意代码

某些科学计算应用呈现出这样的特点，在并行区中每个线程都在一个共享数组的某个子块上进行计算. 在这种情况下，如果各个线程的私有变量仅包含一些标量和规模较小的数组，那么采取上述变量保存和恢复策略是比较低效的，因为主线程要保存太多的数据. 针对这种情形，我们可以考虑采用分布保存的方法，将共享数组分割成若干个子块，分别保存到各个线程上. 这样一来，每个从线程不仅要保存自己的私有状态，还要保存共享数组的一部分. 分布保存共享数组之前应保证所有线程对共享数组有一致的视图，在下一节中我们将讨论这个问题.

4.3 调用栈的保存与恢复

调用栈的保存是通过在程序正常运行过程中记录过程调用和返回操作实现的，这些信息在检查点

保存时将与程序当前所运行到的位置信息一并存入检查点，然后在故障恢复时利用这种信息通过执行程序中的某些语句来重建与原先运行过程中相同的调用栈. 我们称这种恢复调用栈的方法为部分复算.

图 6 说明了部分复算的过程. 其中使用了两个数据结构：过程调用栈(Calling Stack，CS)和恢复跳转栈(Jumping Stack，JS).

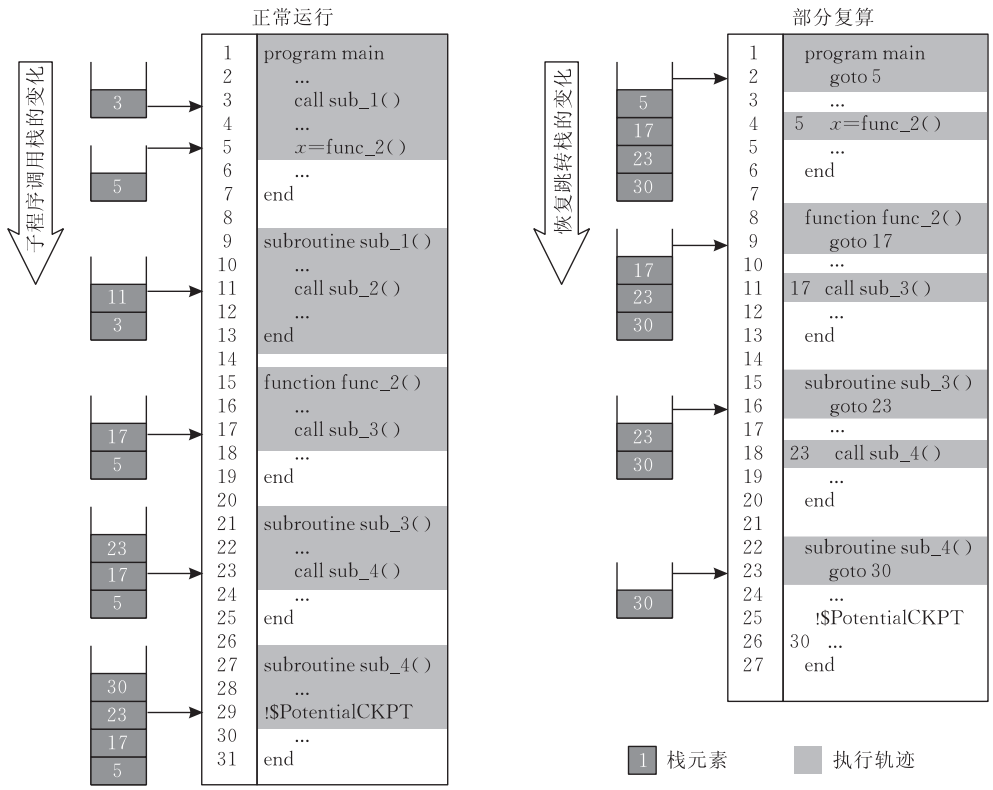


图 6　部分复算过程

CS 是调用栈的一种高级抽象，在程序的正常运行过程中，每进入一个过程时，该过程在调用过程中的位置信息被压入 CS，退出该过程时相应的元素被弹出. 图 6(a)中给出了示意程序在正常运行置检查点保存时 CS 的变化过程.

JS 是 CS 的逆向栈，也就是说，JS 的栈顶元素是 CS 的栈底元素，反之亦然. 在恢复计算开始之前，程序的控制流首先跳转到 JS 栈顶元素所指向的一条过程调用语句，然后 JS 栈顶元素被弹出，程序控制流此时进入被调程序，再重复这一过程，直至控制流最终转到紧跟在检查点之后的那条语句.

图 6(b)中用阴影部分标识出了在两次运行过程总的程序执行轨迹，显然在部分复算过程中只执行了非常少的几条语句，绝大部分语句被跳过. 但仅执行这些语句已经重现了程序正常运行至检查点之前的过程调用顺序，从而重建了与先前的运行过程中相同的程序调用栈.

基于上述思想，我们给出 CS 生成和部分复算的算法，其中 CS 生成算法要求程序的每条过程调用语句和 PotentialCKPT 指导语句之后的语句具有唯一的标签.

算法 1.　过程调用栈生成.

输入：候选检查点位置

输出：程序运行至各个候选检查点位置时的 CS 所构成的 CS 数组

1. 在程序起始处初始化一个临时 CS 和一个 CS 数组；

2. 对于程序运行过程中实现如下操作：

2.1. 每遇到一条过程调用语句，就将该调用语句的标签压入临时 CS；

2.2. 从某个过程返回时，将临时 CS 的栈顶元素弹出；

2.3. 每遇到一条 PotentialCKPT 指导语句，就将该指导语句之后的那条语句的标签压入临时 CS，然后将当前临时 CS 复制一份，作为当前 PotentialCKPT 指导语句的 CS 加 CS 数组. 最后将临时 CS 的栈顶元素弹出；

2.4. 若程序运行结束则算法结束.

算法 2.　部分复算.

输入：CS 数组

输出：恢复计算之前的程序控制流跳转过程

1. 从检查点映像中读取 CS；

2. 对于程序运行过程中实现如下操作：

2.1. 每进入一个过程(包括主程序),就在其第一条执行性语句之前控制流跳转到由 CS 栈顶元素指向的语句,然后将 CS 的栈顶元素弹出;

2.2. 若 CS 已空则算法结束.

图 7 给出了调用栈保存和恢复的示意代码.

```
    program main
      if restarted goto cs.top
      ...
      call tmp_cs.push('lb_1')
lb_1  call sub_1()
      ...
      call tmp_cs.push('lb_2')
lb_2  x = func_2()
      ...
    end

    subroutine sub_1()
      if restarted
        call cs.pop()
        goto cs.top
      end if
      ...
      call tmp_cs.push('lb_3')
lb_3  call sub_2()
      ...
      call tmp_cs.pop()
    end

    function func_2()
      if restarted
        call cs.pop()
        goto cs.top
      end if
      ...
      call tmp_cs.push('lb_4')
lb_4  call sub_3()
      ...
      call tmp_cs.pop()
    end
```

```
    subroutine sub_3()
      if restarted
        call cs.pop()
        goto cs.top
      end if
      ...
      call tmp_cs.push('lb_5')
lb_5  call sub_4()
      ...
      call tmp_cs.pop()
    end

    subroutine sub_4()
      if restarted
        call cs.pop()
        goto cs.top
      end if
      ...
      call tmp_cs.push('lb_6')
      cs(i) = tmp_cs
      call tmp_cs.pop()
!$CheckHere
lb_6  ...
    end
```

图 7 调用栈保存与恢复的示意代码

4.4 OpenMP 并行环境的保存与恢复

OpenMP 程序运行到某个检查点位置时的并行环境是关于当时并行机制的一个描述,其中包括程序可见的并行环境和程序不可见的并行环境两部分.

程序可见的并行环境包括活跃的线程个数、并行区使用的调度策略、变量的数据作用域属性等等.这些环境状态在程序中是以指导语句/子句和过程调用的形式被定义的,因此可以将它们视为特殊的变量进行保存和恢复.

程序不可见的并行环境是指 OpenMP 并行库的内部状态,由于我们不能访问并行库的源码,所以不能对其进行分析以便显式地进行状态的保存.但是,我们可以利用可见的环境状态来重现 OpenMP 并行库的内部状态.在上一节讨论调用栈的恢复时,我们使用基于跳转的方法来恢复检查点保存时的过程调用关系.实际上只要在跳转的过程中依次执行所有对并行环境进行设置的过程调用语句,并且重新执行检查点所在的并行区的定义语句,就可以在恢复调用栈的同时重现检查点保存时的并行环境.

5 非阻塞式检查点一致性协议

5.1 检查点的不一致性

如前所述,所有线程并不是在同一时刻进行检查点的保存,因此可能会出现如图 8 所示的情形.

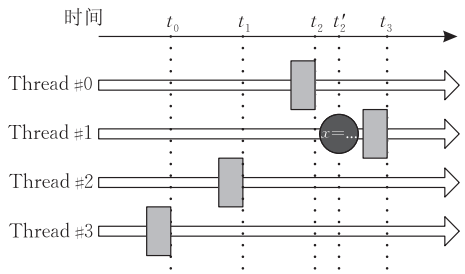


图 8 OpenMP 程序中一次检查点保存

在图 8 中,4 个线程分别在 t_0、t_1、t_2 和 t_3 时刻完成检查点的保存,在时间上的先后顺序为线程 3→线程 2→线程 0→线程 1.并且在 t_2'($t_2 < t_2' < t_3$)时刻,线程 1 对共享变量 x 进行了更新.在这种情况下,首先我们考虑由线程 0 负责共享计算状态的保存的情形.当线程 0 在 t_2 时刻完成共享计算状态的保存时,共享变量 x 处于未被更新的状态.但是,线程 1 在 t_3 时刻进行检查点保存之前的计算状态中,x 应该是已经被更新的状态.对于这种情形,我们称线程 0 和线程 1 所保存的检查点是不一致的.

维护检查点一致性的一种做法是各个线程在进行检查点的保存之前先进行一次同步,这就是阻塞式检查点的基本思想.虽然阻塞式检查点可以有效地实现计算状态的一致性,但是如果采用软件实现,其缺陷也是显著的,主要包括两个方面.首先是阻塞式检查点会在程序中引入额外的同步开销.对于负载不平衡的程序,这种同步开销在有些情况下有可能是非常严重的.其次,额外的同步操作可能会在程序中造成死锁.例如线程 m 在同步等待其它线程以便进行检查点保存的同时,另一个线程 n 可能在另一个同步点上等待线程 m.

因此我们提出一种非阻塞式检查点,在避免各个线程在检查点保存之前进行同步的前提下维护检查点的一致性.

5.2 协议描述

我们所提出的一致性协议基本思想如下:

(1)令主线程专门负责共享计算状态中标量的保存,对于共享数组则可以由主线程专门保存,也可以由所有线程分布保存,但要保证任何一个数据单元都仅被一个线程保存.

(2) 共享计算状态仅在每个线程都对其具有相同的视图之后才进行保存. 也就是说，当所有的线程都对共享变量完成更新之后，才开始进行共享计算状态的保存.

(3) 在对共享计算状态进行保存的同时，必须保证对所有读写共享变量访问的独占性，当一个线程完成私有计算状态的保存后直至共享计算状态被保存之前，该线程不能访问共享计算状态.

因此在这种协议之下，检查点的协议期从发起者开始进行私有计算状态的保存时开始，到主线程完成共享计算状态的保存时结束. 我们使用一个全局共享标志 CKPT_Stared 来表示程序目前是否处于协议期，初值为 FALSE，并定义其访问接口 Get_CKPT_Flag()、Set_CKPT_Flag()和 Clear_CKPT_Flag()，分别实现 CKPT_Started 的读取、置 1 和清 0. 此外，我们还使用一个全局共享的计数器 CKPT_Counter 来记录各个线程检查点保存的进度，初值为 0，相应地定义其访问接口 CKPT_Counter_Read()、CKPT_Counter_Set()和 CKPT_Counter_Dec()，分别进行计数器值的读取、设置和减 1 操作. CKPT_Started 和 CKPT_Counter 的访问都是互斥的. 在不考虑共享数组分布保存的情况下，非阻塞式一致性协议描述如下.

每个线程在 PotentialCKPT 位置上根据 CKPT_Started 的值进行下列操作：

(1) 如果 CKPT_Started=FALSE，且最近一次检查点据当前时刻的时间间隔不超过 T_{itvl}，则当前位置上不需要进行检查点保存.

(2) 否则，按照下列步骤进行检查点保存：

① 如果 CKPT_Started=FALSE，则该线程成为本次检查点的发起者，此时将 CKPT_Started 置为 TRUE，将 CKPT_Counter 置为总线程数；

② 开始进行私有计算状态的保存，完成后将 CKPT_Counter 减 1.

(3) 根据线程的角色完成下列操作：

① 主线程检查 CKPT_Counter 的值，如果非 0 则说明还有线程未完成检查点的保存，此时主线程循环等待. 若为 0 则说明所有线程都已完成了私有计算状态的保存，此时主线程开始进行共享计算状态的保存，完成后将 CKPT_Started 置为 FALSE. 至此，协议期结束.

② 从线程继续进行后续计算，在下一次共享变量的定值语句之前检查 CKPT_Counter 的值，若非 0 则循环等待.

6　OpenMP 应用级检查点位置的优化选取

为了降低检查点技术在程序运行中引入的额外开销，一方面我们可以采用第 3 节所讨论的数据流分析方法降低单次检查点的保存开销，另一方面我们还可以减少总的检查点保存次数，从而降低多次检查点保存的总开销.

在先前我们所讨论的应用级检查点技术中，程序在某个用户指定的候选检查点位置上是否确实进行检查点保存取决于运行至该检查点时的时间戳信息. 这种方案可以满足 MTBF 的时间约束，但没有考虑多次检查点保存的总开销. 在最坏的情况下，如果每次程序决定进行检查点保存的位置上都存在很大的活跃变量集合，那么检查点保存的总开销可能仍是无法接受的. 针对这一问题，在本节中我们提出一种优化的检查点选择方法，在用户指定的若干个候选检查点位置中选择一部分作为实际进行检查点保存的位置. 选择的原则上是在满足 MTBF 约束的前提下，使总的保存数据量尽可能地小，从而进一步降低检查点保存开销.

6.1　问题抽象

假设用户在程序中指定了 N 个检查点候选位置，要从中选择 $M(M\leq N)$ 个作为实际进行检查点保存的位置，称为入选位置. 定义下列符号表示：

(1) 令 C_i 表示第 i 个候选位置，其中 $1\leq i\leq N$. C_i 具有下列域：

① $C_i.time$ 表示程序运行到第 i 个候选位置时相对于程序开始运行时刻的时间；

② $C_i.shared$ 表示第 i 个检查点位置上活跃的共享变量的总数据量；

③ $C_i.private$ 表示第 i 个检查点位置上活跃的私有变量的总数据量；

④ $C_i.size$ 表示第 i 个检查点候选位置上的数据保存量. 若程序在 k 个线程上运行，则有

$$C_i.size=C_i.shared+k\cdot C_i.private;$$

(2) 令 x_i 表示第 i 个候选位置是否入选(相应取值为 1 和 0).

接下来我们首先讨论检查点位置选取的前提，也就是检查点间隔时间应该满足 MTBF 约束.

具体来说，在给定 MTBF 的情况下，入选位置中任意两个相邻位置之间的时间间隔应当不大于一个与 MTBF 有关的阈值 T_{itvl}. 该约束条件表示为

$$\text{if } (x_i \cdot x_j = 1) \text{ and } \left(\sum_{l=i}^{j} x_l = 2\right)$$

$$\text{then } C_j.time - C_j.time \leq T_{itvl}.$$

下面我们讨论如何根据 MTBF 确定 T_{itvl} 的取值.

根据可靠性理论[11],硬件失效率是与时间无关的常数,记为 λ,并且有 $\lambda=\frac{1}{MTBF}$.

系统发生故障的时刻 t 是服从参数为 λ 的指数分布的随机变量,其概率密度函数为

$$f(t)=\lambda e^{-\lambda t}.$$

分布函数为

$$F(t)=\int_0^t \lambda e^{-\lambda t} dt = 1-e^{-\lambda t}.$$

令 $R(t)$ 表示程序的运行在时间段 $[0,t]$ 内不发生故障的概率(假设 0 时刻系统处于无故障状态),则有

$$R(t) = 1-F(t) = e^{-\lambda t}.$$

上式的物理含义也可以理解为程序从 0 时刻开始能够以概率 $R(t)$ 运行至 t 时刻而不发生故障.

如果选取的 M 个检查点中某两个检查点 C_j 和 C_{j+1} 之间的时间间隔 $\delta t=t_{j+1}-t_j$ 过长,那么如果有故障发生在 C_j 和 C_{j+1} 之间,程序回滚到 C_j 之后,能够运行 δt 时间而不发生故障的概率很低,也就是 $R(\delta t)$ 的值很小,那么该程序可能需要相当多次的反复回滚才能到达 C_{j+1}. 因此 T_{itvl} 的取值应该使 $R(T_{itvl})$ 的值足够大,才能避免反复回滚的情况发生. 例如令 T_{itvl} 满足如下约束:

$$R(T_{itvl})=0.99 \Rightarrow T_{itvl}=\frac{\ln 0.99}{\lambda},$$

即

$$e^{-\lambda T_{itvl}}=0.99 \Rightarrow T_{itvl}=\frac{\ln 0.99}{\lambda}.$$

在可靠性为 99.9999% 的系统上,这个阈值约为 3h.

$R(t)$ 是单调减函数,因此若任意两个检查点 C_j 和 C_{j+1} 之间的时间间隔 $\delta t=t_{j+1}-t_j<T_{itvl}$,则 $R(\delta t)>R(T_{itvl})$,也就是说只要检查点间隔小于这个阈值,程序就能以更高的概率从 C_j 运行到 C_{j+1} 而不发生故障.

此外,为了保证检查点间隔小于 T_{itvl},从 N 个候选检查点位置中应该选取的实际进行检查点保存的入选位置数 M 应该满足

$$M \geq \frac{T}{T_{itvl}}.$$

基于这些讨论,检查点位置选取问题可以抽象为 0-1 整数规划模型如下:

$$\min\left\{\sum_{i=1}^{N} C_i.size \cdot x_i\right\}$$

$$\text{s.t.}\begin{cases} \sum_{i=0}^{N} x_i = M \\ M \geq \dfrac{T}{T_{itvl}} \\ \text{若 } (x_i \cdot x_j = 1) \text{ 且 } \left(\sum_{l=i}^{j} x_l = 2\right) \\ \qquad \text{则 } C_j.time - C_j.time \leq T_{itvl} \\ x_i = 0 \text{ 或 } 1 \end{cases}.$$

6.2 问题求解

我们采用过滤隐枚举法对上述模型求解,算法如下.

算法 3. 检查点位置的优化选取.

输入:候选位置数 N、时间间隔阈值 T_{itvl}、检查点候选位置 $C_i(1\leq i\leq N)$

输出:入选位置数 M、最优目标函数值 Z^*、最优解 x_i^* $(1\leq i\leq N)$

1. 选择一个可行解 x_i,计算目标函数值 Z;
2. 置过滤条件 $\sum_{i=1}^{N} C_i.size \cdot x_i \leq Z$;
3. 重复本步骤直至没有未考察的枚举点
   选取一个未考察的枚举点 x_i,计算目标函数值 Z';
   if $Z'<Z$, then
      if x_i 满足所有约束条件 then
         $x^*=x_i$;
         $Z=Z'$;
      end if
   end if
4. $Z^*=Z$.

7 实验评测

7.1 实验设置

我们在一台 32 结点系统上对本文提出的应用级检查点机制进行了性能评测. 该系统每个结点包含两个 4 核 Itanium® 2 处理器,支持软件实现的全局分布共享存储、高速互连以及硬件实现的取-加操作. 实验测试程序选用 NPB OMP 3.2 测试程序包[12]中的 4 个程序:BT、CG、FT 和 MG.

实验中首先使用数据集规模 C 测试了 3 组执行时间,用于评估检查点保存及恢复的时间开销. 第 1 组是原始版本的各个测试程序的执行时间,在下一节的图表中用 Clean 表示;第 2 组是各测试程序在执行过程中进行一次检查点保存情况下的总执行时

间，用 $ALC(0)$ 表示；最后一组执行时间包括一次检查点保存和一次从该检查点的恢复计算，用 $ALC(1)$ 表示.

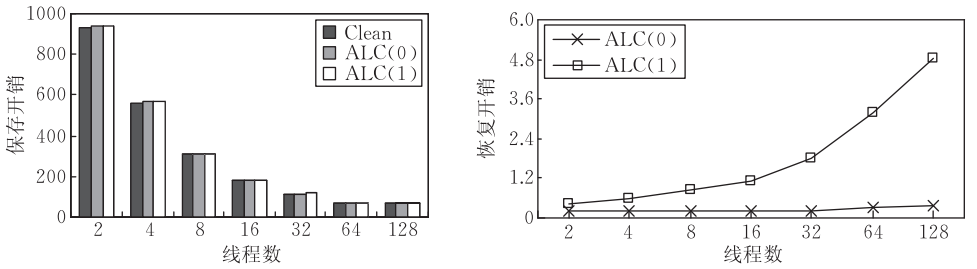
接下来，我们使用数据集规模 S、W、A 和 B 测试了两组执行时间，分别用于计算进行一次检查点保存和一次故障恢复的时间，并与 C3 系统的这两个指标进行比较.

此外，在实验中还记录了各个程序在数据集规模 B 下的检查点映像文件大小，并与 C3 系统的检查点映像文件大小进行比较.

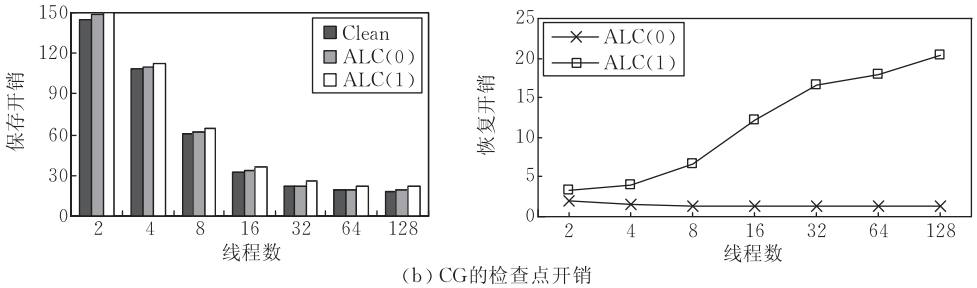
最后，在实验中还对本文所提出的检查点位置优化选择方法进行了评测.

7.2　实验结果

图 9 和图 10 显示了 4 个测试程序的检查点保存和恢复开销. 可以看出，我们所提出的应用级检查点技术在目标系统上的检查点保存开销是比较低的，在使用不同线程数运行程序时都保持在 5% 以内. 由于文件的写入操作在数据被写入缓冲区后就可以返回，因此开销只和数据量有关. 但是恢复时间相对于程序运行时间的百分比随着线程个数的增加而增加，这是因为文件的读操作需要在数据“真正”被读入后才能返回，而当线程数量比较多时每个结点上运行的线程数也比较多，由此引起的 I/O 竞争导致了较大的恢复开销.

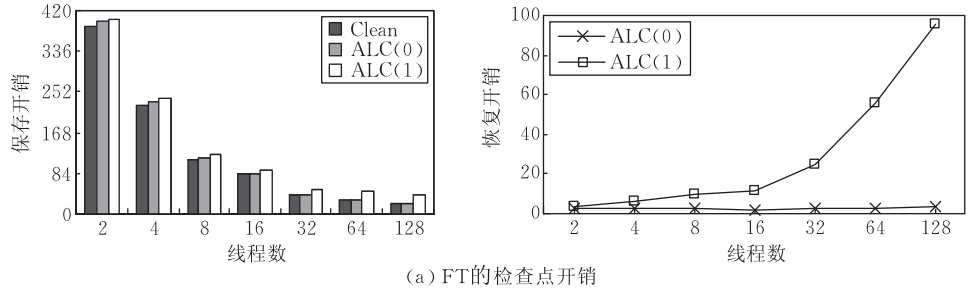


(a) BT的检查点开销

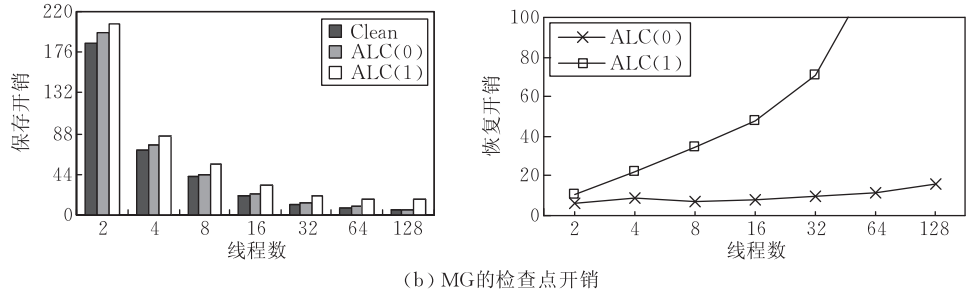


(b) CG的检查点开销

图　9



(a) FT的检查点开销



(b) MG的检查点开销

图　10

如图 10 所示，由于 FT 和 MG 两个程序的检查点数据量非常大，而当线程数超过 32 时，程序的运行时间又比较短，因此恢复开销与运行时间的百分比激增. 实际上，如果线程调度机制能够保证每个结点上只运行一个线程，那么恢复开销也可以保持在较低的水平.

表 2 和表 3 列出了我们所提出的检查点机制(用 ALC 表示)与 C3 系统在时间开销方面的比较情况，其中 C3 的性能数据来自文献[13-14]. 表中存在空白项是因为我们无法获得 C3 系统在相应指标上的数据.

表 2 与 C3 系统的检查点保存开销比较

		保存开销							
		CLASS S		CLASS W		CLASS A		CLASS B	
BT	ALC	1.3E-4s	0.61%	0.024s	1.22%	0.07s	0.19%	0.21s	0.13%
	C3	3.2E-05s	0.03%	0.074s	1.63%	1.2s	0.60%	17s	2.02%
CG	ALC	0.007s	4.61%	0.042s	5.38%	0.13s	8.44%	0.38s	0.92%
	C3	0.012s	24.00%	0.053s	23.54%	0.15s	8.15%	1.6s	2.07%
FT	ALC	0.011s	7.06%	0.02s	7.73%	0.15s	6.63%	1.32s	2.35%
	C3	0.035s	24.71%	0.07s	23.66%	1.1s	18.87%		
MG	ALC	0.0013s	37.66%	0.104s	46.55%	0.67s	37.65%	0.67s	7.53%
	C3	0.0075s	102.75%	0.030s	7.16%	1.8s	53.74%	2s	12.66%

表 3 与 C3 系统的检查点恢复开销比较

		恢复开销							
		CLASS S		CLASS W		CLASS A		CLASS B	
BT	ALC	4.1E-04s	1.87%	0.069s	3.50%	0.15s	0.44%	0.72s	0.46%
	C3	−8.6E-05s	−0.07%	0.067s	1.48%	0.41s	0.21%	7.3s	0.87%
CG	ALC	0.013s	8.73%	0.083s	10.68%	0.22s	14.70%	0.71s	1.69%
	C3	0.012s	22.96%	0.028s	12.70%	0.14s	7.60%	0.45s	0.59%
FT	ALC	0.016s	11.18%	0.03s	11.01%	0.20s	8.59%	2.23s	3.96%
	C3	0.022s	15.21%	0.036s	12.08%	0.68s	11.30%		
MG	ALC	0.0021s	59.53%	0.221s	98.30%	1.04s	58.11%	0.543s	6.06%
	C3	0.0092s	126.94%	0.011s	2.67%	0.087s	2.54%	0.046s	0.28%

表 4 与 C3 系统的检查点映像文件大小比较

	文件大小					
	ALC	C3				
	CLASS B	CLASS S	CLASS W	CLASS A	CLASS B	
BT	30.4MB	2.7MB	17.6MB	306MB	1.2GB	
CG	37.3MB	3.1MB	17.8MB	60MB	428MB	
FT	257.6MB	13.5MB	26.9MB	419MB		
MG	129.4MB	1.5MB	8.1MB	438MB	438MB	

表 2 列出了两种检查点机制的检查点保存开销，给出了绝对时间和相对于总执行时间的百分比. 从中可以看出在 15 对比较数据中(较优者以粗体表示)，本文所提出的检查点机制胜出 12 对，说明在检查点保存开销方面，显著优于 C3 系统. 得益于本文所提出的 DAO 数据流分析方法，检查点中排除了相当多的大数组，因此在时间开销上表现出了相当的优势.

表 3 列出了恢复开销的比较情况，在这方面本文所提出的方法与 C3 系统的性能相当. 在 15 对比较数据中，本文所提出的方法胜出 7 对，并且还有一点比较显著的结果，即对于 MG 程序来说，两种机制的恢复时间相差非常大. 其原因还是在于我们的检查点机制在实验中采用的同步 I/O 读，这在很大程度上增加了恢复开销.

表 4 给出了检查点映像文件大小的比较情况，从中也可以看出我们所提出的 DAO 数据流分析方法在减小检查点映像文件方面的实际效果. 此外，共享数据由各个线程分割保存也是检查点文件较小的一个原因.

在检查点优化设置实验中，我们在每个测试程序中指定了 8 个检查点候选位置. 由于测试程序的运行时间都远小于目标系统的 MTBF，所以在进行实验时根据每个测试程序原始版本的运行时间分别指定了一个合适的 T_{itvl} 值，使优化算法能够为每个程序产生 5 个检查点入选位置. 在这 5 个位置上进行检查点保存得到的平均检查点文件大小在图 11 中用白色竖条表示. 为了比较，我们还在 8 个候选位置中随机挑选了 5 个位置进行检查点保存，得到的平均检查点文件大小用深灰色竖条表示.

图 11 给出了归一化后的比较结果，从中可以看出，经过优化选择的检查点设置相对于优化前的任意设置节省了约 15%～30%的数据保存量.

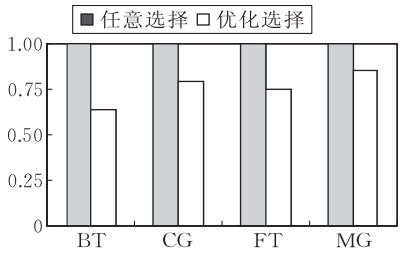


图 11　检查点优化设置的收益

8　结束语

本文针对 OpenMP 程序的应用级容错技术展开研究，提出了一种应用级检查点机制. 文章详细讨论了检查点机制的各个关键问题，并给出了实验评估以及与同类工作的比较. 本文提出的 OpenMP 扩展数据流分析方法 DAO 对于减少检查点数据量非常有效，能够显著地改善检查点机制的性能.

目前，对 DAO 的研究还在进一步进行中，主要针对的问题是通过对数组访问轨迹的分析实现更精确的活跃变量集合求解.

参　考　文　献

[1] TOP500 Supercomputing Sites. http://www.top500.org/, 2010

[2] Reed D, Lu C, Mendes C. Reliability challenges in large systems. Future Generation Computer Systems, 2006, 22(3): 293-302

[3] Bosilca G, Bouteiller A, Cappello F et al. MPICH-V: Toward a scalable fault tolerant mpi for volatile nodes//Proceedings of the ACM/IEEE Conference on Supercomputing. Los Alamitos, California, USA, 2002: 1-18

[4] Sorin D, Martin M, Hill M, Wood D. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery//Proceedings of the International Symposium on Computer Architecture. Anchorage, Alaska, USA, 2002: 123

[5] Zhang Z, Prvulovic M, Torrellas J. ReVive: Cost-effective architectural support for rollback recovery in shared memory multiprocessors//Proceedings of the International Symposium on Computer Architecture. Anchorage, Alaska, USA, 2002: 111

[6] Dieter W, Lumpp J. A user-level checkpointing library for POSIX threads programs//Proceedings of the 1999 Symposium on Fault-Tolerant Computing Systems. Madison, Wisconsin, USA, 1999: 224-227

[7] Bronevetsky G, Marques D, Pingali K et al. Application-level checkpointing for shared memory programs. SIGARCH Computer Architecture News, 2004, 32(5): 235-247

[8] Bronevetsky G, Marques D, Pingali K et al. Automated application-level checkpointing of mpi programs//Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. San Diego, California, USA, 2003: 84-94

[9] Bronevetsky G, Marques D, Pingali K et al. C3: A system for automating application-level checkpointing of mpi program//Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing. College Station, Texas, USA, 2003: 357-373

[10] Yang X, Du Y, Wang P et al. The fault tolerant parallel algorithm: The parallel recomputing based failure recovery//Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques. Brasov, Romania, 2007: 199-212

[11] Liu P. Fundamentals of Reliability Engineering. Beijing: China Metrology Publishing House, 2002(in Chinese)
(刘品. 可靠性工程基础. 北京：中国计量出版社，2002)

[12] Bailey D, Harris T, Saphir W et al. The NAS parallel benchmarks 2.0. California: NASA Ames Research Center, Technical Report: NAS-95-020, 1995

[13] Bronevetsky G, Pingali K, Stodghill P. Experimental evaluation of application-level checkpointing for openmp programs//Proceedings of the 20th Annual International Conference on Supercomputing. Cairns, Queensland, Australia, 2006: 2-13

[14] Bronevetsky G. Portable Checkpointing for Parallel Applications[Ph. D. dissertation]. Ithaca, New York, USA: Cornell University, 2007

FU Hong-Yi, born in 1978, Ph. D. candidate, assistant engineer. His research interests focus on parallel and distributed systems, fault tolerance, and scientific computing.

DING Yan, Ph. D. candidate, assistant professor. Her research interests focus on trusted computing and system security.

SONG Wei, born in 1982, Ph. D. candidate. His research interests include parallel computer system architecture, fault tolerance, and scientific computing.

YANG Xue-Jun, born in 1963, professor, Ph. D. supervisor. His research interests include supercomputer architecture, parallel and distributed operating system, parallel language and compiler.

Background

This research work is addressing the reliability issue of large scale parallel computing systems, and focused on checkpointing, which is widely used in the domain. For long-time-running scientific programs, periodically saving checkpoints and restart from a checkpoint upon a failure induce considerable time overhead. So far the research works concerning this are majorly done for message passing parallel programs, trying to lower the overhead for saving and restoring from checkpoints. This work is considering the checkpointing approach for shared memory parallel programs. The authors use an extended parallel data flow analysis to reduce the data amount of checkpoint as possible, to shorten the time spent for saving and restoring.

This research work is support by the National Natural Science Foundation of China, with the project ＃60621003. The project is named as 'The Key Techniquees for TFLOPS High Performance Computing'. It's focused on processor architecture for high performance computing, structure-awared internetwork, and scalable parallel algorithm and system software. The group has made multiple achievements, including the programming model and compiler techniques for streaming processor architecture and for GPU architecture, and the fast failure recovery scheme based on parallel re-computing. All these works have been published in several top ranking conferences such as ISCA, PACT, PPoPP, and ICDCS.

This research work is dedicated for providing application level fault tolerance solution for shared memory systems and OpenMP programs, to improve the reliability of parallel computing systems built on emerging multi-core processors. This work plays an important role in the project 'The Key Techniques for TFLOPS High Performance Computing'.