具有 $O(n)$消息复杂度的协调检查点设置算法*

汪东升+, 邵明珑

(清华大学 计算机科学与技术系,北京 100084)

A Cooperative Checkpointing Algorithm with Message Complexity $O(n)$

WANG Dong-Sheng+, SHAO Ming-Long

(Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)

+ Corresponding author: Phn: 86-10-62785592, Fax: 86-10-62771138, E-mail: wds@tsinghua.edu.cn

http://www.tsinghua.edu.cn

Received 2001-09-06; Accepted 2002-08-02

Wang DS, Shao ML. A cooperative checkpointing algorithm with message complexity $O(n)$. *Journal of Software*, 2003,14(1):43~48.

Abstract: The technology of cooperative checkpointing and rollback recovery as an effective method of fault tolerance, has been widely used on the parallel or distributed computer systems, such as cluster of computers. In order to reduce the overhead of time and space, a cooperative checkpointing algorithm based on message counting is given in this paper. While reducing a message complexity during synchronization from $O(n^2)$ to $O(n)$, improving system's efficiency and scalability, this algorithm is also fit for those non-FIFO message passing systems.

Key words: checkpointing; rollback recovery; synchronization; message counting

摘 要: 协调检查点设置及回卷恢复技术作为一种有效的容错手段,已广泛地运用在集群等并行/分布计算机系统中.为了进一步降低协调检查点设置的时间和空间开销,提出了一种基于消息计数的协调检查点设置算法.该算法无须对底层消息通道的FIFO特性进行假设,并使同步阶段引入的控制消息复杂度由通常的$O(n^2)$降低到$O(n)$,有效地提高了系统的效率和扩展性.

关键词: 检查点设置;卷回恢复;同步;消息计数

中图法分类号: TP301 **文献标识码**: A

检查点设置以及回卷恢复(cooperative checkpointing and rollback recovery,简称CRR)技术能够在系统发生软件/硬件故障时,利用保存在可靠存储设备中的进程状态(检查点),将其恢复并继续执行,从而最大限度地减少因故障带来的损失.在大型科学计算程序中,检查点技术可以有效地提高系统的可用性和容错能力.

在并行/分布的环境中,各进程之间由于消息传递而存在着依赖关系.这种依赖关系体现在接收消息的进程必须以发送消息的进程为前提.在底层消息通道不可靠的系统中,所有已被接收的消息必须在另一个进程状态

---

* Supported by the National High-Tech Research and Development Plan of China under Grant No.2001AA111010 (国家高技术研究发展计划); the Assisting Project of Ministry of Education of China for Backbone Teachers of University and College (国家教育部高等院校骨干教师资助计划)

第一作者简介: 汪东升(1966－),男,黑龙江肇东人,博士,副教授,主要研究领域为并行处理,容错计算.

© 中国科学院软件研究所 http://www.jos.org.cn

中表现为已被发送;对于底层通道可靠的系统,已被发送的消息必须在另一个进程状态中表现为被接收.符合上述规定的并行程序中的所有进程状态组成的集合称为"一致性全局状态"[1].

并行/分布式系统中的CRR技术必须获得任务集的"全局一致性状态",才能保证恢复进程的正确执行.根据获得"全局一致性状态"的不同途径,现有的并行/分布系统的 CRR 技术大致可分为独立方式和协调方式两种.独立方式在设置检查点的时候不需要与其他进程进行协调,所以进程执行过程中一次检查点设置带来的开销相对协调方式较小.但在检查点保存内容的多少、检查点文件数目的大小、恢复算法的难易等方面,协调方式较独立方式有很大的优势,这也是它得到广泛运用的主要原因[2].

建立一致性状态的关键是消除系统中的孤儿消息(orphan message)和中途消息(transition message).协调检查点设置方式本身即可以避免孤儿消息.对于清除中途消息,现有的协调策略大多数通过同步达到这个目的,比如清华大学的 Charm 系统[3]、慕尼黑大学的 COCHECK 系统[4]、威斯康星大学的 CONDOR 系统[5]等等,这些系统采用两两进程之间进行"消息驱赶",并附以"消息纪录"的机制,清除检查点设置前存在的中途消息.在规模较小的分布式环境中,这是一种简单而有效的做法,但由于协调过程中引入的辅助控制消息数量和进程数之间呈 $O(N^2)$关系,随着进程数量的增多,这些控制消息的数目将会急剧增加,引入不可忽视的时间和空间开销.本文提出的算法也是建立在消息驱赶基础之上.与现有的"同步-驱赶"算法相比,该算法具有以下优点:

- 算法不需要对底层消息通道 FIFO 特性进行假设.我们通过记录进程发送和接收消息的数目,并附以管理进程的协调,使得算法在不满足 FIFO 条件的消息传递通道模型中同样适用.
- 高效.在检查点设置的协调过程中引入的控制消息数量与进程数相关,并且满足线性关系:$N_{系统消息}=2N_{进程数目}$,复杂度为 $N(O)$,即使在规模较大的分布式并行应用中,也不会引入很大的开销.
- 不需要检查点设置中为避免消息丢失而引入的额外"退出时同步"操作.

这些特性使得算法能够满足大规模并行应用程序的需求,以较小的时间、空间开销在分布式计算环境中实现容错功能,从而提高系统的可靠性和可用性.该算法已经成功地应用在我们开发的千亿次集群系统的容错环境中.

本文第 1 节描述与算法相关的系统模型.第 2 节具体描述算法原型及其实现.第 3 节主要给出在实际环境中的测试数据.第 4 节介绍相关工作.第 5 节是结论.

1 系统模型

1.1 系统假设

本文提出的检查点设置算法的应用对象是一些开放的分布式系统,比如工作站集群,对底层硬件和软件没有特殊要求.我们对底层的分布式系统有如下假设:系统由 N 个互联的计算结点组成;各结点之间可以互相通信;各结点没有同步的时钟,整个系统也没有全局时钟;各进程之间仅通过消息进行通信;两进程之间建立的点对点的通信连接称为消息通道;底层消息通道可靠,但无须满足 FIFO 特性.

1.2 系统定义及相关术语

定义一个分布式并行应用程序 P,$P=\{P_1,P_2,\ldots,P_n\}$,其中 n 是该并行应用程序中包含的进程数目,我们称其为计算进程或者用户进程.并行应用程序 P 有时也称作任务集.另外,还有一个协调控制进程 manager,它负责检查点设置时的协调和控制工作.

C_i 表示第 i 次设置的检查点(i=1,…,m),S_i 表示第 i 次同步操作(i=1,…,c),I_k 表示(S_k,S_{k+1})两次同步操作之间的间隔时间(i=1,…,c−1).

本文将检查点设置过程中用于协调进程行为的各种消息称为控制消息(control message)或者协调消息(cooperation message),原来用户进程之间为实现计算目标而进行通信的消息称为用户消息(user message)或者应用消息(application message).

© 中国科学院软件研究所 http://www.jos.org.cn

2 低开销协调式检查点设置算法

协调式检查点设置算法一般有 3 个步骤.首先是采取合适的策略同步各计算进程,清除可能引起全局不一致性的孤儿消息和中途消息.接下来是检查点内容的提取.最后将检查点保存到可靠存储器中,各个阶段都有对应的优化策略.本文主要讨论协调式检查点设置算法中同步算法的设计和实现.

2.1 算法描述

如前假设,分布式并行程序中现有 n 个进程 $P_i(i=1,\dots,n)$.

每个进程都维护一个简单的“进程号-消息个数”结构,用于纪录该进程发出的消息个数和发向的进程,我们称该结构为消息发送表.当进程 P_i 向进程 P_j 发送一个消息时,P_i 就将本进程的消息发送表中 P_j 对应的消息个数加 1.此外,每个进程还有一个消息接收计数器(Msg_counter),统计接收到的所有消息个数.消息发送表和计数器在一次同步操作完成后清零.即消息发送表记录的是 I_k 阶段进程发送的所有应用消息的目的地和对应的发送次数;消息接收计数器则记录在此期间本进程已经收到的所有应用消息的个数.

Manager 也有类似于消息发送表的结构,但不同的是,manager 将综合所有进程发来的消息发送表,得到任务组中每个用户进程在本次同步操作期间应该接收的消息个数,故将其称为消息接收表.根据 manager 在协调过程中所处的不同地位,我们将其状态分为 IDLE,SYNC 和 EXEC.算法如图 1 所示.

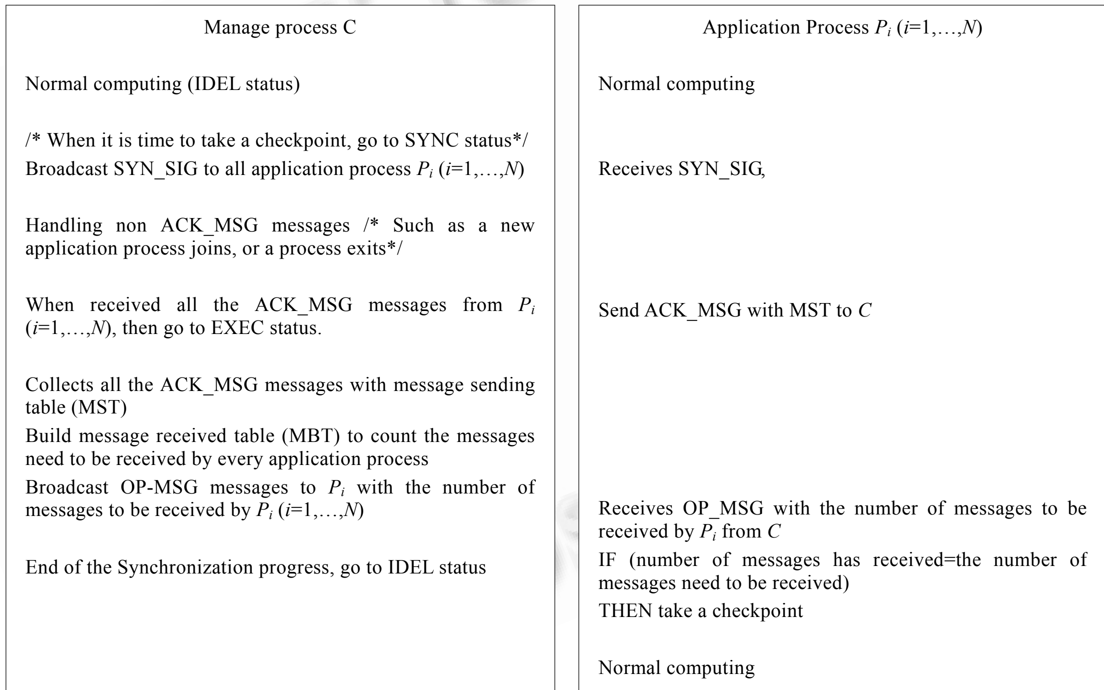

Fig.1 Synchronization algorithm based on message counting
图 1 基于消息计数的同步算法

操作完成后,各进程 $P_i(i=1,\dots,n)$皆处于同步状态,可以进行检查点设置、迁移或恢复操作.

2.2 算法分析

(1) 全局一致性的保证.

由于信道可靠,上述算法可以保证指向某一进程的所有消息接收通道都已被清空,即已经没有“中途消息”.我们可以用图 2 来说明这个过程.

© 中国科学院软件研究所 http://www.jos.org.cn

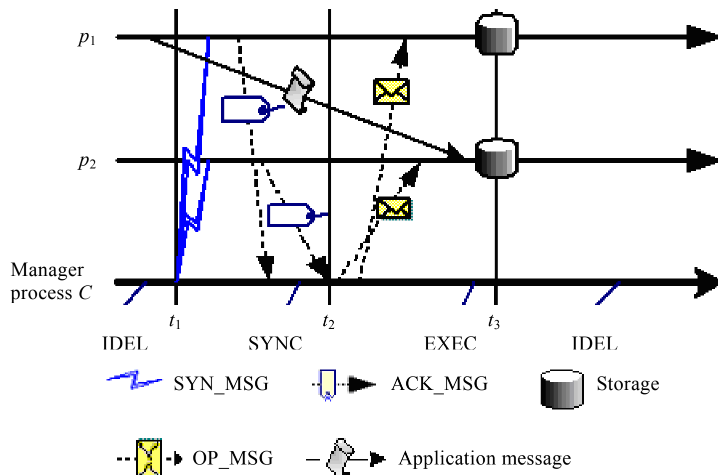

Fig.2　Synchronization procedure based on message counting
图 2　基于消息计数的同步过程

各种协调控制消息、信号如图 2 所示.存储设备保存本次检查点文件,带箭头的线是进程执行时间轴.控制进程在同步过程中把本次检查点设置期间应该接收到的消息数量告知各用户进程,用户进程利用该信息"清空"指向自己的消息通道.检查点设置操作推迟到 t_3,此刻所有应用消息都已经被相应进程接收并记录下来.t_1 时刻前发送的应用消息不可能穿越检查线(t_3 时刻),即系统中消除了导致"全局不一致"性的中途消息(transit message).此时的检查点集合满足"全局一致性"条件,由其组成的检查线能保证恢复过程的正确执行.

从图 1、图 2 也可以看出,同步过程引入的协调控制消息数量与并行应用程序的规模相关,设并行程序中包含 N 个进程,那么控制消息数量为 $2N$,复杂度为 $O(N)$,控制消息数量和并行程序规模呈线性关系.

(2) 避免了对底层消息通道 FIFO 特性的假设.

本算法的"消息驱赶"实际上是利用了在 I_k 期间,进程收到的消息数目必须等于发向它的消息数目这一简单而直接的判断准则.Manager 进程通过收集、综合用户进程的发送表,得到在 I_k 期间发向各进程的消息总数.显然,如果每一个进程在此期间接收到的消息和发向它的消息数目相等,那么可以肯定系统中已不存在中途消息,这就达到了"驱赶"的目的.

在驱赶过程中,并没有像以往的系统那样依赖于某个特定的消息(比如用于标志通道清空的驱赶消息)来判断过程的结束,而是通过检查接收和发送的消息数量是否一致来完成,所以,在非 FIFO 信道中,这种算法同样适用.

(3) 减少消息驱赶带来的系统开销.

在以往的同步算法中,没有消息发送表和接收表的结构.用户进程在接收到 manager 发出的同步信号以后,向其他所有进程发送一个特殊的清空消息 CLS_MSG,然后持续接收发向自己的应用消息直至收到所有其他进程发来的 CLS_MSG.利用消息通信层的 FIFO 语义.我们可以知道,此时发向该进程的所有消息通道已被清空,系统中再没有中途消息存在,各进程的状态集合满足"全局一致性"条件,可以进行检查点设置、迁移或者恢复操作.这种方法实现简单,不足之处在于数量为 $O(N^2)$级的清空消息 CLS_MSG 将给系统带来很大的通信量.当 N 增长的时候,同步时的控制消息量将随着规模的扩大,以平方的量级增长,在时间和空间上引入了很大的开销.

在提出的"辅以消息计数"的同步算法中,我们通过在控制信息中夹带少量信息来达到减少整个系统中通信次数的目的.在这个算法中,同步消息个数的增加是 $O(N)$级的,每个 ACK_MSG,OP_MSG 都携带一定的信息量,并通过 manager 的协调来完成同步操作.

(4) 避免了退出前同步的操作.

在基于 PVM 通信库的检查点设置和卷回恢复应用中,需要另外一些辅助机制,以协助系统实现容错功能.为了完整地保存 PVM 进程的信息,我们必须先退出 PVM 系统,将用户进程作为一个普通进程保存,然后再重新

© 中国科学院软件研究所　http://www.jos.org.cn

加入 PVM.这种“退出再加入”的方法改变了用户进程的 PVM 标识.在这种情况下,利用信道的 FIFO 语义进行消息驱赶的算法会造成任务结束前发出的消息不能被正确发送到接收任务这样的隐患.

比如,任务 *B* 在建立了 n–1 个检查点以后结束,并且在执行段 n–1 向任务 *A* 发送消息 msg.此时,映射机制将用户给出的 *A* 的初始任务号 tid *A*, 0 映射为执行段 n–1 时 *A* 的实际任务号 tid *A*, n–1,并向任务 tid *A*, n–1 转发 msg.假定任务 *A* 直到执行段 n 才调用 pvm_recv 接收消息 msg.在第 n 次检查点设置的协调式同步过程中,由于任务 *B* 在执行段 n–1 中结束而未参与同步,msg 并未作为中途消息被驱赶,并被 tid *A*, n–1 提前接收.因此,当任务 *A* 任务号变为 tid *A*, n 以后,就无法收到 msg 了.

一般的解决办法是,在任务结束之前发起一次额外的紧协调式同步,待所有由此任务发出的中途消息都被提前接收时,再结束任务.在采用了“基于消息纪录的同步算法”以后,由于有发送表和接收表,我们可以避免这一次额外的同步操作.进程结束时向管理进程发送的退出消息可以携带发送表,manger 利用它更新接收表,这样,在下一次的同步过程中,该消息就会被作为中途消息驱赶,从而避免上文提到的消息丢失现象.

3 性能测试

测试环境为具有 4 个节点(每个节点是一个含有 4 个 XEON 处理器的 SMP 计算机,每个处理器的主频为 700MHz,内存为 1G)的集群系统,节点间用 100M 快速以太网连接起来,操作系统为 Redhat6.2(kernel version is 2.4.4).测试用例为通信密集型的分型计算程序和计算密集型的矩阵幂程序.除了测试每次检查点所用的开销之外,我们还分别考察了随着程序规模(进程数目)的扩大,同步开销的增长情况,如图 3 所示.这里的同步开销(synchronization overhead)是指图 1 中 SYNC 和 EXEC 两部分所用时间之和.

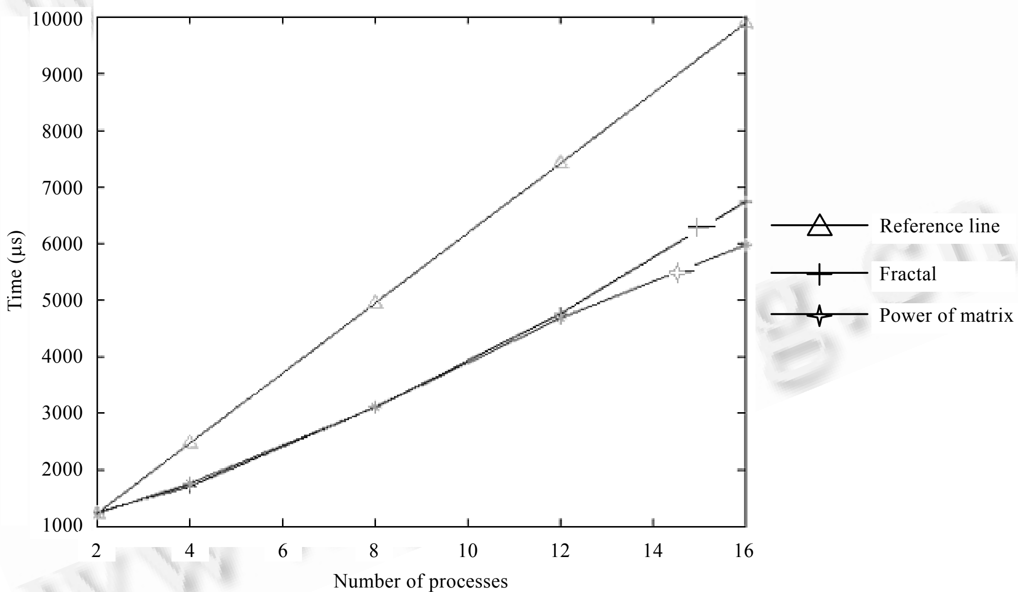

Fig.3 Curve of the process number vs. synchronization overhead
图 3 进程数量与同步开销的关系曲线

从测试结果中可以看出,因为同步引入的时间开销基本上在毫秒级,所以它同并行程序的运行时间相比是微不足道的.随着并行程序规模的扩大,同步开销的增长在实际测试中低于线性增长函数,结果比较令人满意.需要指出的是,整个检查点设置的开销还应包括保存检查点时读写磁盘的时间.为了进一步提高整个检查点设置的效率,我们利用了自行开发的“高可用性网络内存”技术代替传统的写磁盘操作,有效地降低了检查点设置时的 I/O 操作实践.相关工作将另文阐述.

© 中国科学院软件研究所 http://www.jos.org.cn

4 相关工作

协调检查点设置以及回卷恢复技术已经广泛地应用在很多并行容错系统中,比如慕尼黑理工大学信息学院的 CoCheck[4,6]、俄勒冈科学技术研究院的 MIPS[7]、威斯康星大学计算机系的 Condor[5]等等.它们都使用一种简单的“同步-暂停”算法来协调进程行为,在通信密集的应用程序中,有些效率上的损失.本算法虽然延续了这种设计思路,但因为对同步过程作了优化,所以在实际应用中效率较高,适于规模较大的并行应用.

此外,在协调检查点设置过程中,还可以使用另外一些算法.在韩国 Sejong 大学和汉城国立大学提出的“用户程序控制的协调式算法”中,作者对分布式应用中各进程之间的通信关系作了分类,由应用程序根据不同的通信模型,采用最适宜的算法[8].由于协调算法的针对性很强,所以能得到比较好的性能,但是,通信模式的获取需要一定程度的程序员参与,同时,为了探测“多米诺”深度,需要辅助的检测工具,这加大了实现的复杂度.新南威尔士大学和惠普公司提出的协调算法为分布式应用程序中的每个组成进程添加了两个用于通信和管理的辅助线程[9],故而能够将原来“集中”进行的同步过程在时间上“分散”开来,从而减小了同步协调的过程对计算进程的影响,但是它极大地增加了回卷的深度(rollback distance).

References:

[1] Chandy KM, Lamport L. Distributed snapshots: determining global states of distributed systems. ACM Transactions of Computer Systems, 1985,3(1):63~75.

[2] Elnozahy EN, Johnson DB, Wang YM. A survey of rollback-recovery protocols in message passing systems. Technical Report, CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, 1996.

[3] Wang DS, Shen MM, Zheng WM, Pei D. Checkpoint-Based rollback recovery and process recovery. Journal of Software, 1999,10(1):68~73 (in Chinese with English abstract).

[4] Tannenbaum T, Litzkow M. The condor distributed processing system. Dr. Dobb’s Journal, 1995,20(2):40~48.

[5] Pruyue SJ. Resource management and checkpointing for PVM. In: Hermes PP, ed. Proceedings of the 2nd Europe PVM User Group Meeting. 1995. 131~136. http://wwwbode.informatik.tu-muenchen.de/~cocheck/EPVMUG95.ps.gz.

[6] Stellner G. CoCheck: checkpointing and process migration for MPI. 1996. http://ipdps.eece.unm.edu/1996/PAPERS/S13/STELLNER/STELLNER.PDF.

[7] Casas J, Clark D, Galbiati P, Konuru R, Otto S, Prouty R, Walpole J. MIST: PVM with transparent migration and checkpointing. 1995. http://www.cse.ogi.edu/DISC/projects/mist.

[8] Park T, Yeom HY. Application controlled checkpointing coordination for fault-tolerant distributed computing systems. Parallel Computing, 2000,26(4):467~482.

[9] OuYang JS, Maheshwari, P. Supporting cost-effective fault tolerance in distributed message-passing applications with file operations. Journal of Supercomputing, 1999,14(3):207~232.

附中文参考文献:

[3] 汪东升,沈美明,郑纬民,裴丹.一种基于检查点的卷回恢复与进程迁移系统.软件学报,1999,10(1):68~73.

© 中国科学院软件研究所 http://www.jos.org.cn