高性能并行计算系统检查点技术与应用

孙国忠 李艳红 樊建平

(中国科学院计算技术研究所 中国科学院研究生院 北京 100080)
(sgz@ncic.ac.cn,lyh@ncic.ac.cn,fan@ict.ac.cn)

摘 要 随着高性能并行计算系统规模越来越大，软件和硬件发生故障的概率随之增大，系统的容错性和可靠性已经成为应用可扩展性的主要限制因素。并行检查点技术可以使系统从故障中恢复并减少计算损失，是高性能计算系统重要的容错手段。本文将介绍检查点技术的背景和定义，研究并行检查点协议的分类，检查点存储技术，以及利用这些协议和技术实现的 MPI 并行检查点系统，最后给出对各个关键技术的详细评价及结论。

关键词 高性能计算;消息传递系统;并行检查点;回滚恢复
中图法分类号 TP31

A Survey of Checkpointing Technology and It's Application for High Performance Parallel Systems

Sun Guo-Zhong Li Yan-Hong Fan Jian-Ping

(*Institute of Computing Technology,Chinese Academy of Sciences/Graduate School of the Chinese Academy of Sciences, Beijing 100080*)
(sgz@ncic.ac.cn lyh@ncic.ac.cn fan@ict.ac.cn)

Abstract With the scale of high performance parallel computing systems becoming larger,the fault probability of software and hardware in these systems is increased.As a result, issues of fault tolerance and reliability are becoming limiting factors on application scalability.Parallel checkpointing can help fault system recover from fault and reduce the computing losing,and is an important method for tolerating fault of high performance computing system.This paper will discuss the background and definitions of checkpointing,classify of parallel checkpointing protocols, checkpoint storage technology, and several MPI systems adopting these parallel checkpointing protocols.At last we give appraisement of these key technologies and list our conclusions.

Key words High Performance Computing; Message Passing System; Parallel Checkpointing ; Rollback Recovery

1 引 言

高性能并行计算领域的容错技术由于以下几种情况而越发受到重视。1）在一台高性能计算机系统中，总的处理器数快速增长。如 BlueGene/L 总的处理器有 130,000 个，有证据表明这样的一台机器几个小时就要有一个处理器失效。虽然处理器总数的提高带来了性能提高，但是也提高了故障点的数目。2）大多数并行计算机系统正在从采用昂贵的硬件系统向低成本、由处理器和光纤网络定制组装的 cluster 转变，以及采用 Internet 范围内网格技术来执行程序导致硬件发生故障的概率较高。3）很多科学计算任务被设计成一次运行几天或者几个月，例如 ASCI 的 stockpile certification 程序以及 BlueGene 当中的 ab initio 蛋白质折叠程序将运行几个月。由于应用的运行时间比硬件的平均故障间隔时间(MTBF)长，科学计算程序必须具有对硬件故障的容错技术。采用检查点技术恢复应用运行是一种有效的容错方法。

检查点技术除了实现系统容错，还能协助实现灵活的作业调度。例如，拥有高性能计算系统的气象局要在每天的固定时段加载资源独占作业进行气象预报或者运行紧急作业，需要暂停原来运行的其它作业。因此必须记录原来作业的检查点并在完成紧急作业后恢复运行。

可见，采用检查点技术可以实现系统容错，实现灵活的作业调度以及提高资源利用率。本文将通过对各种并行检查点技术的分析比较，呈现出高性能并行计算系统检查点机制的发展状况，存在的问题和研究前景。

2 背景和定义

检查点技术在各个领域都进行了广泛研究，如硬件级指令重试、分布式共享内存系统、系统调试、实时系统等。本文侧重于高性能并行计算系统，主要包括 MPP、Cluster。这些系统的进程之间通过消息传递实现通信，本文中也称为消息传

本课题得到国家高科技发展计划(863)基金支持(2003AA1Z2070)和中国科学院知识创新工程支持(20036040)

递系统。

在高性能并行计算系统中有两类检查点，单进程检查点(局部检查点)和并行程序检查点(全局检查点)。单进程检查点是将进程足够的状态存到外存的一个文件里，保证从这个外存文件能够恢复进程，使其在检查点正确的继续运行。并行程序检查点由多个单进程检查点组成。高性能并行计算机系统检查点技术就是通过各种协议的设计，利用单进程检查点技术，能切取/恢复并行程序的状态。

2.1 系统模型

一个消息传递系统由一些相互协调合作运行的分布式应用程序进程组成，进程之间通过且仅通过消息传递来通信，且与外界（outside world）通过发送输出消息和接收输入消息进行交互。图 1 给出了一个包含 3 个进程的消息传递系统。

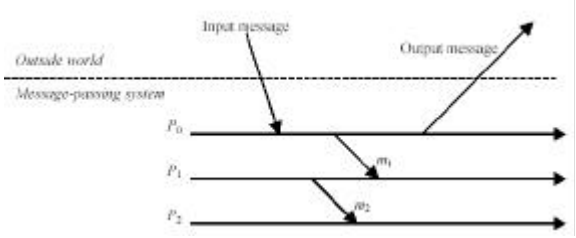


图 1 具有 3 个进程的消息传递系统

2.2 一致的系统状态

一个消息传递系统的全局状态包括所有参与进程的局部状态以及通信通道的状态。一致的系统状态定义为：系统状态中不包含孤立消息。孤立消息定义为：消息的接收事件被记录，但是发送事件却丢失了。图 2 给出了一致状态和非一致状态的例子。(a)是一致状态，表示 m1 已经被发送，但是还没有被接收。m1 称为传递中消息。当传递中消息成为系统全局状态的一部分，这些消息不会导致不一致。(b)是非一致状态，m2 已经被 P2 接收并记录下来，实际 P1 的状态记录 m2 还没有发送。

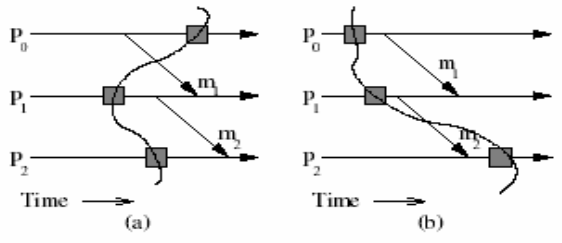


图 2 一致状态和非一致状态说明

一致的全局检查点是指包含在并行程序全局检查点中各个单进程的状态，它描述了该消息传递系统的一致系统状态。

2.3 检查点协议

在基于检查点协议中，每个进程周期或根据某些条件记录局部检查点。检查点内容包括进程数据段、堆栈段、寄存器和环境变量。环境变量包括文件指针、工作目录等进程描述符等内容。

进程可以相互协调记录检查点形成一致的状态，这种方法称为协作式检查点(coordinated checkpointing)，也称为同步(synchronous)检查点。另外各个进程也可以独立记录检查点，在恢复阶段获得一致状态，称为独立式检查点(independent checkpointing)，也称为无协作检查点(uncoordinated checkpointing)，或者异步(asynchronous)检查点。另外一种方法称为消息引导检查点(Communication-induced checkpointing)，也称为类同步(quasi-synchronous)检查点，它强制每一个进程根据收到的捎带在其它应用进程消息中的信息记录检查点。

为减少损失把系统恢复到最近的一致全局检查点，这个一致的全局检查点称为恢复线（Recovery line)。图 3 中，3 个进程开始的检查点集合构成了一个恢复线。

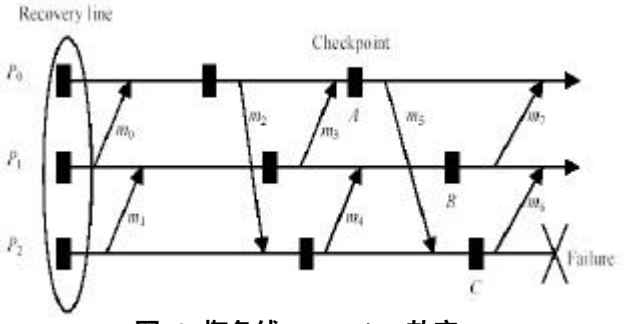


图 3 恢复线、Domino 效应

2.4 Domino 效应

独立式检查点方法容易导致 Domino 效应。图 3 中，假设 P2 发生故障，回滚到检查点 C。回滚使 m6 无效，导致 P1 必须回滚到 B，以使接收到的 m6 无效。P1 的回滚导致使 m7 无效，又使得 P0 回滚。如此往复，导致了 Domino 效应，使得系统回滚到开始处。为了避免 Domino 效应，可以采用协作式检查点或者消息引导检查点，也可以把独立式检查点和事件日志结合起来。

2.5 与外界交互

一个消息传递系统经常和外界交互接收输入数据或者输出计算结果。如果发生错误，外界不能只依靠回滚实现恢复。例如，一个打印机不能回滚到已经打印了一个字符之前的状态。因此系统在向外界输出结果之前，必须保证即使发生故障，也能从向外界输出后的状态中恢复到输出之前的状态。这通常称为输出提交问题。

类似的，系统从外界接收的消息也不能在恢复中重新生成。恢复机制必须把外界消息保存起来，在需要的时候可以获得这些消息。

2.6 日志协议

在基于日志的协议中，进程不仅记录检查点，也记录与事件相关的足够日志信息。这对于与外界交互频繁的系统来说尤其有意义。有日志的协议可以有效避免 Domino 效应。

根据记录日志事件信息的时机，可以分为三种方法。悲观日志(pessimistic logging)，应用在事件可以被其它进程或外界看到之前，阻塞等待直到把事件相关信息存放到稳定存储上。乐观日志(optimistic logging)，应用不被阻塞，事件相关的日志信息异步记录到稳定存储上。因果日志(causal logging)，结合了以上两种方法的优点。

基于日志的协议以分段确定性(PWD：PieceWise Deterministic)假设作为基础。在这个假设下，恢复机制可识别出进程执行的所有非确定性事件。非确定性事件的例子包括接收消息、接收外界输入、中断等。

2.7 稳定存储

稳定存储是一种存储介质，当断电后它所保存的值不会丢失。稳定存储可以用来记录检查点、事件日志或其他和恢复操作相关的信息。稳定存储经常和实现它的磁盘相混淆，但实质上还有几种稳定存储介质，包括磁带、磁鼓、非易失性内存。

2.8 垃圾回收

检查点内容和事件日志都会占用存储资源。随着进程的执行，更多的恢复信息被收集，其中的一部分会变得无用。垃圾回收是指对这些无用信息的删除以释放存储资源。

垃圾回收的一般方法是识别出恢复线，丢弃恢复线之前事件相关的信息。例如，如果多个进程采用协调记录检查点协议来形成一致状态且总是从每个进程最近的检查点开始恢复，则可以丢弃前面所有的检查点。

3 并行检查点协议分类

通过对消息传递系统中全局一致状态的分析可知，并行检查点设置和恢复技术包括两部分：各进程状态的保存和恢复，以及通信通道状态的保存和恢复。前者与单进程检查点内容密切相关，而对后者的处理协议可分成两大类：检查点协议和日志协议。检查点协议中，进程只记录描述进程状态信息的检查点。日志协议中，进程不仅记录检查点，也记录事件相关的日志信息。

3.1 检查点协议

检查点协议不需要检测、记录以及重新回放事件，因而比日志协议简单。但它不保证已经发生事件的重放，因此不适合频繁和外界交互的应用。

检查点协议分为三种[2,3,8]，协作式检查点，独立式检查点，通信引导式检查点。

3.1.1 协作式检查点

这种方法需要多个进程协调记录检查点以获得一致的全局状态，只需要每个进程在稳定存储上维护一个永久性检查点，降低了存储开销以及避免了垃圾回收操作。它的缺点是在向外界提交输出一个结果时需要协调记录一个全局的检查点，导致提交输出操作有较大延迟。

协作式检查点有以下几种方法：阻塞式、非阻塞式、基于同步时钟、最少检查点协调方法。

- 阻塞式检查点算法。记录检查点时阻塞所有进程通信。该算法有一个协调器，首先记录一个检查点，然后向其他的进程发送记录检查点请求。当所有进程记录完检查点后恢复运行。
- 非阻塞式检查点算法。记录检查点时不阻塞所有进程通信。分布式快照协议是这种算法的一个例子。协议中，有一个发起者首先记录一个检查点，然后广播发送标记到所有进程作为检查点请求。所有进程接收到第一个标记后马上记录检查点，进程在记录完检查点之后并向其他应用发送消息之前，再向所有进程广播发送一次标记。这个协议假定通信通道是可靠的即先进先出的，解决了可能出现的孤立消息问题。另一个方法是用检查点索引号代替标记。一个进程接收到附在应用消息上的索引号后与本地检查点索引号比较，如果小则记录检查点。
- 基于时钟同步的方法。松散同步时钟可以在几乎同一时间激活所有参与进程的检查点记录操作，而不用特殊的检查点发起者。通过运行最快的节点不断发送同步时间限制时钟偏移在一定范围内。
- 最少检查点协调方法。协作式检查点要求所有的进程参与检查点记录。当系统规模很大时可扩展性成为问题。可以控制只有直接或间接和发起者发生通信关系的进程记录新的检查点，以此减少在一次检查点操作中参与的进程数。

3.1.2 独立式检查点

这种方法中，进程具有极大的自主性来决定记录检查点的时机，例如在进程状态信息量小的时候记录。它的缺点是：1）容易产生 Domino 效应；2）进程可能会记录不属于全局一致状态的无用检查点，导致不必要的开销；3）每个进程要维护多个检查点，必须周期性的激活垃圾回收删除不再使用的检查点；4）不适合和外界交互频繁的应用，因为向外界输出时需要全局的协调操作，导致开销的增加。

为了在恢复时获得一致全局检查点，进程要记录检查点之间的依赖关系。采用如下技术：设 Ci,x

为进程 Pi 的第 x 个检查点，x 称为检查点索引。Ii,x 表示检查点区间，位于 Ci-1,x 和 Ci,x 之间。如图 4 所示。如果 Pi 在区间 Ii,x 向 Pj 发送一个消息 m，它把(i,x)捎带在 m 上，当 Pj 在 Ij,y 区间收到 m，进程 Pj 就记录从 Ii,x 到 Ij,y 的依赖，当 Pj 记录检查点 Cj,y 时把依赖关系存到稳定存储上。在以后的恢复中，通过依赖关系获得一致的全局状态。有两种方法获得一致全局状态，一种是回滚依赖图(rollback-dependency graph)方法，另一种是检查点图(checkpoint graph)方法，不再详述。

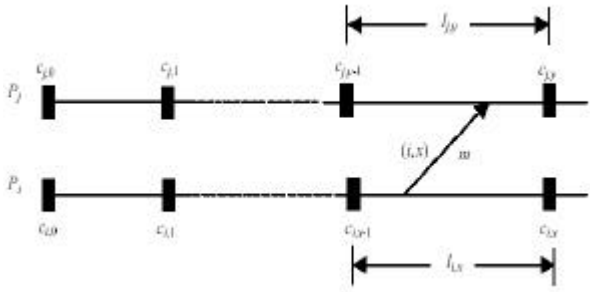


图 4 检查点索引和检查点间隔

3.1.3 通信引导式检查点

在这种方法中，进程要记录两种检查点，一种是进程各自独立记录的，称为局部检查点，另一种是进程被强制记录的，称为强制检查点。进程根据接收到的其他进程消息捎带来的信息决定是否记录强制检查点。和协作式检查点相比，这种方法不需要交换专门的消息。

有两种通信引导检查点方法，基于模型的检查点(model-based checkpointing)和基于索引的检查点(index-based checkpointing)。

- 基于模型的检查点。首先建立一个模型，然后根据捎带在应用消息上的暗示来测试导致不一致状态的模式是否发生。如果发现将发生不期望的模式，则记录强制检查点来避免。也可以通过该方法避免记录无效检查点。
- 基于索引的检查点。系统为所有的局部和强制检查点建立索引，各个进程具有相同索引的检查点形成一致状态。索引捎带在应用消息上来帮助接收者确定是否记录一个强制检查点，例如当捎带的索引比局部检查点索引大时记录一个强制检查点。另外，捎带更多的消息可以减少强制检查点的记录。

3.2 日志协议

日志协议把进程的执行看成非确定状态区间序列，每个区间以非确定事件的执行开始。协议假定所有的非确定事件都可以被识别，同时事件相关的信息都可以记录到稳定存储上。另外，进程也记录检查点。

一个进程的状态依赖于非确定事件，但是该事件在恢复阶段不能重新生成，这样的进程称为孤立进程(orphan process)。产生孤立进程将导致系统的不一致状态。日志协议可保证在执行恢复操作时不产生孤立进程从而获得一致的全局状态。

日志协议分为三种[2,5]，悲观日志，乐观日志，因果日志。

3.2.1 悲观日志

悲观日志假定任何非确定事件之后都可能发生故障。最直接的实现方式是在事件影响计算结果之前，事件日志信息都已经存到稳定存储上。

图 5 说明了悲观日志协议。在正常执行时，P0、P1、P2 记录事件日志以保证在恢复时重新生成消息。假设 P1 和 P2 发生错误，然后从检查点 B 和 C 开始恢复执行，在这个过程中，重新生成正常执行时的消息并按原来的顺序发送。一旦恢复完成，这两个进程和 P0 处于一致的全局状态，P0 包含从 P1 发来的消息 m7。

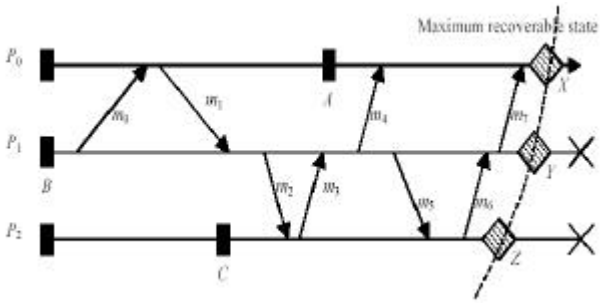


图 5 悲观日志

悲观日志导致很高的开销。可以采用特殊的硬件降低开销。例如，采用非易失性内存作为稳定存储，另外一种形式是采用专门的总线记录日志消息。也可以不采用硬件，如基于发送方消息日志把消息记录在发送方动态内存内。

3.2.2 乐观日志

这种方法中，事件日志信息以易失性日志的形式临时保存，然后周期地存放到稳定存储。它乐观地假设在发生故障之前能记录完日志。因此，应用可继续执行，不必阻塞等待日志保存到稳定存储。和悲观日志相比，乐观日志必须记录多个检查点导致垃圾回收算法相对复杂。另外可能产生孤立进程。

图 6 说明了乐观日志协议。假设 m5 相关的事件日志记录到稳定存储之前 P2 发生故障。此时，P1 变成了孤立进程，必须回滚取消接收 m6 的操作，P1 回滚操作导致 P0 取消接收到 m7 的操作。要正确执行取消操作，必须记录进程消息之间的依赖关系，以保证恢复到最近的全局一致状态。

乐观日志必须记录多个检查点导致垃圾回收算法相对复杂。如 P2 故障导致 P1 从检查点 B 恢复而不是最近的检查点 D 恢复。另外，输出提交需要多个进程协调而延迟输出提交。

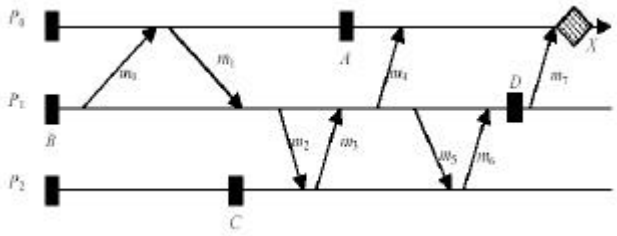


图 6 乐观日志

3.2.3 因果日志

因果日志协议结合了乐观日志和悲观日志的两者的优点。一方面，它避免了同步访问稳定存储的操作；另一方面，它允许每个进程独立地提交输出操作而不产生孤立进程。另外，在恢复时能保证进程回滚到最近的存放在稳定存储上的检查点。但这些优势的获得是以复杂的恢复协议为代价的。

因果日志协议中事件日志记录先发生于进程某一状态。这种先后关系基于 Lamport 的先发生关系[1]，即如果进程 A 的发送事件 a 导致发送消息 m 给进程 B，B 的消息接收事件 b 触发接收，则 a 先于 b。每个进程的事件日志中都包含这种先后依赖关系，一般用一个有向依赖图来描述。进程可以根据依赖关系指导其它进程触发事件或重新发送消息。

3.3 协议比较

表 1 给出了不同协议在几方面内容的比较。

表 1 协议比较

协议 / 比较内容	协作式	独立式	通信引导	悲观日志	乐观日志	因果日志
分段确定性假设	否	否	否	是	是	是
垃圾回收	简单	复杂	复杂	简单	复杂	复杂
进程记录检查点数	1 个	多个	多个	1 个	多个	1 个
Domino 效应	无	可能	无	无	无	无
孤立进程	无	可能	可能	无	可能	无
回滚程度	最近全局检查点	无限制	可能多个检查点	最近检查点	可能多个检查点	最近检查点
恢复复杂性	简单	复杂	复杂	简单	复杂	复杂
输出提交速度	很慢	无法实现	很慢	最快	慢	快

4 检查点存储技术

基于磁盘、非易失性内存稳定设备的检查点技术需要向稳定的存储设备存储大量的检查点，带来很大内容存储开销。因此提出了无盘检查点[6,7]，主要思想是利用内存等非稳定存储减少记录检查点到稳定设备上的开销。它可以恢复部分部件损坏故障，但是当整个系统故障（如突然断电）时无法恢复。有相关研究提出了两层架构方案，一层是实现频繁的无盘检查点，另一层是实现相对较长间隔的稳定存储检查点。

本部分主要阐述无盘检查点相关内容，其中介绍的检查点记录方法也适用于稳定存储介质。

4.1 检查点记录

无盘检查点情况下，检查点都存放在内存中。有三种方法实现检查点记录：(1)简单无盘检查点技术，把进程的地址空间和寄存器内容存放到内存中。 (2)增量检查点技术，当要记录一个检查点时，把进程虚拟地址空间的所有页设置成只读状态，当应用写一个页时，会引发一个异常，检查点系统则在异常处理中把要写的页作一个备份，并把页设成读写状态。(3)Forked(或 copy-on-write)方法，记录检查点时，进程先 clone 自己，这个 clone 就是无盘检查点。当要回滚，进程使用 clone 的状态回写，或者把 clone 当作进程运行。

4.2 检查点编码

在使用稳定存储时，检查点或事件日志可直接存放。在使用非稳定存储的情况下，利用检查点处理器内存来存放所有应用程序处理器内存上的检查点编码。当某些应用程序处理器故障，则它们的检查点可以通过未发生故障的处理器的检查点和编码计算得到并进行恢复。检查点编码有以下方法：

- 简单奇偶方法。假设有一个检查点处理器，有 n 个应用程序处理器，检查点处理器上的字节表示为：

  $B^j_{ckp}= B^j_1 \oplus B^j_2 \oplus \ldots \oplus B^j_n$ (j 表示第 j 个字节)

  该方法能容忍一个处理器故障。当某一个应用程序处理器故障，则可以通过该公式计算出检查点内容，并选择备用处理器进行恢复。
- 镜像。n 个备份处理器能容忍 n 个处理器故障。
- 一维奇偶方法。n 个应用处理器，分成 m 组，每个组提供一个检查点处理器，采用奇偶方法容错，整体可容忍 m 个处理器故障。
- 二维奇偶方法。应用处理器被组织成二维网格的形式。在网格的每一行和每一列都有一个检查点处理器。可以容忍任何两个处理器故障。
- Hamming 码方法。n 个应用处理器，增加 logn 个检查点处理器，可以容忍任意两个处理器故障。
- EVENODD 编码。两个备份处理器，容忍任意两个处理器故障。
- Reed-Solomon 编码。m 个备份处理器能容忍任意 m 个处理器故障。

4.3 针对大规模系统的无盘检查点技术

目前，Oak Ridge 实验室正在研究针对大规模系统(如 BlueGene/L)的无盘检查点技术。研究中

采用一种点对点无盘检查点算法[6]。算法中把进程状态向相邻的多个进程复制，相邻进程执行同样的操作。相邻进程的个数决定进程的容错度。

在进行的试验中，对具体的 FFT 算法使用检查点技术并进行了分析。可以在 FFT 算法的不同阶段采用协作式检查点算法或非协作式检查点算法。

5 采用并行检查点技术的 MPI 系统

检查点可以在两个级别实现：核心级和用户级。核心级实现的缺点是要记录进程的所有状态，检查点内容存储开销大，在大规模系统当中几乎是不可行的。优点是可以直接访问并记录与用户进程相关的核心数据结构。用户级实现的优点是用户可以控制检查点的位置、内容，具有一定自主性。缺点是无法直接访问核心数据结构。

MPI 广泛应用于 MPP 和机群系统等并行系统环境。在 MPI 库实现检查点算法具有用户级实现的特点，另外具有可移植性和透明性的优点。当前有很多采用并行检查点技术的 MPI 系统[8--13]，包括 MPICH-V、MPICH-V2、CoCheck、LAM/MPI、Starfish MPI 等。

1）MPICH-V：该项目利用检查点技术适应节点的动态性，如在网格环境下节点的随时加入和退出。基于非协作检查点悲观日志算法，每一个计算进程和一个 Memory Channel 进程相关联，发送的消息按序记录在 Memory Channel 中，接收消息向 Memory Channel 请求。使带宽利用率减半，极大影响性能。

2）MPICH_V2：同 MPICH-V 相比，也是基于非协作检查点悲观日志算法，但是移去了 Memory Channel，需要更少的可靠的节点来运行必需的服务进程。

3)CoCheck： 检查点机制构建在 MPI 层之上，利用 Condor 库来记录单进程的检查点状态。能为应用提供透明的检查点和进程迁移功能，可以方便的移植到不同的 MPI 实现上。它的基本思想是保证网络上没有正在传递的消息，这样每个进程就可以记录检查点并获得一致的全局状态。实现了协作式检查点算法。

4)LAM/MPI：结合 Berkeley 实验室的核心级进程检查点系统 BLCR，提供检查点操作接口。

LAM/MPI 实现了协作式检查点算法。通过 mpirun 命令，所有的 mpi 进程初始化检查点操作，进程之间开始相互协调以通过本地检查点实现一致的全局检查点。LAM/MPI 通过 Staggered All-to-All 算法[9]保证在记录检查点时通道上没有消息传递，以避免消息事件影响。

5)Starfish MPI：能适应节点动态变化情况，提供内建的检查点功能。使用严格的原子组通信协议实现通信功能，避免了 Cocheck 中使用的消息清除协议。另外 Starfish 支持进程迁移以实现负载均衡。Starfish 支持协作式和非协作式检查点算法。

除了上述和检查点技术相关的 MPI 研究项目，MPICH-GF 研究在网格环境中引入检查点功能。它试图对网格环境下的应用进行容错，实现应用在节点故障后的可恢复计算。

6 评价及结论

采用检查点协议的容错系统中访问稳定存储是主要开销。协作式检查点通过通信来协调记录检查点，而通信开销较小，因此比独立式检查点具有优势：每个进程只需记录一个检查点，垃圾回收简单，不易发生 Domino 效应，容易恢复。通信引导检查点通过原型系统证明当进程数增长时可扩展性不好。在进程随机位置的强制检查点的引入，导致某一应用对稳定存储需求的不可预测性进而影响检查点放置策略。这些缺点影响了它的实际应用。

日志协议应用在和外界交互频繁的系统中具有明显的优势。它可以快速向外界提交输出并且记录输入日志，也能有效处理非确定事件，这在单纯采用检查点协议的系统中是不可能的。可以把协作式检查点和日志协议结合以同时获得两者的优点，如协作式检查点具有的快速恢复和易于垃圾回收及日志协议具有的快速输出提交特点。另外，两者结合使得基于发送方的日志协议不必把易失性的消息日志存放到稳定存储，这样就不需要维护大量日志，使得开销小且实现简单。

由于利用磁介质记录检查点延迟开销大，非易失性内存价格昂贵导致容量上的限制，因此采用无盘检查点技术可以弥补这两面的不足。另外采用两层架构方案，可以对整个系统如断电导致的严重故障进行容错。

并行计算系统中主要使用 MPI 库开发应用。在 MPI 库层实现检查点具有透明性和可移植性等优点，是实现系统容错的一种有效途径。

在大规模并行系统上采用检查点技术实现系统容错仍然存在有待解决的问题，比如削减由检查点引入的时间开销；压缩检查点占用空间，减小空间开销；优化协议，减小传输负载和通信开销。目前的许多研究项目正致力于对这些问题的解决。

参 考 文 献

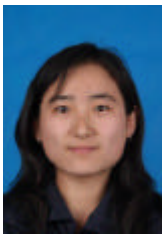
1 Chandy, K.M. and Lamport, L.. Distributed Snapshots: Determining Global States of Distributed Systems, *ACM Transactions on Computer Systems*, 3(1):63-75, 1985.

2 E.N. Elnozahy, L. Alvisi, Yi.-Min. Wang,D.B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems, ACM Computing 34, No. 3, September 2002, p. 375–408.

3 Adnan Agbaria,Ari Freund, Roy Friedman. Evaluating Distributed Checkpointing Protocols. Proceedings of the 23rd International

Conference on Distributed Computing Systems (ICDCS’ 03)
4 J. S. Plank.Efficient Checkpointing on MIMD Architectures.Doctor Dissertion,Princeton 1993.
5 L. Alvisi,K Marzullo. Message Logging:pessimistic,optimistic ,causal and optimal. IEEE Trans. Softw. Eng. v24, n2, 1998,149–159.
6 Christian Engelmann and Al Geist. A Diskless Checkpointing Algorithm for Super-scale Architectures Applied to the Fast Fourier Transform. Proceedings Of The International Workshop On Challenges Of Large Applications In Distributed Environments,IEEE,June 2003,p.47-52
7 J. S. Plank, Y. Kim, and J. J. Dongarra. Fault tolerant matrix operations for networks of workstations using diskless checkpointing. *Parallel and Distributed Computing*, 43(2):125–138, 1997.
8 Aur´ elien Bouteiller, Pierre Lemarinier, G´ eraud Krawezik, Franck Cappello. Coordinated checkpoint versus message log for fault tolerant MPI. 1 Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER ’ 03),p.242-250
9 Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine.The LAM/MPI Checkpoint/Restart Framework : System-Initiated Checkpointing. In Proceedings, 17th Annual International Symposium on High Performance Computing Systems and Applications, Quebec, Canada, pages 277-283, May 2003.
10 Adnan M. Agbaria,Roy Friedman.Starfish:Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. In 8th IEEE International Symposium on High Performance Distributed Computing, pages 167-176. IEEE, 1999.
11 Georg Stellner.CoCheck: Checkpointing and Process Migration for MPI. Proceedings of IPPS '96,IEEE, p.526-531
12 George Bosilca, Aurelien Bouteiller, Franck Cappello,etal. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. Proceedings of the 2002 ACM/IEEE conference on Supercomputing,P.1-18
13 Aur´ elien Bouteiller, Franck Cappello y, Thomas H erault. MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on PessimisticSender Based Message Logging. SC’ 03, November 15-21, 2003,
14 L. Alvisi, E. Elnozahy, S. Rao, S. A. Husain, and A. D. Mel. An analysis of communication induced checkpointing. In 29th Symposium on Fault-Tolerant Computing (FTCS’ 99). IEEE CS Press, june 1999.

孙国忠　男，1973年生，在读博士研究生，主要研究方向为高性能计算、计算机系统软件。



李艳红　女，1979 年生，在读硕士研究生，主要研究方向为高性能计算、机群操作系统。

樊建平 男，1963 年生，博士，研究员，博导，主要研究方向为高性能计算机系统结构、计算机系统软件。