高技术通讯 2012 年 第22 卷 第12 期:1243 ~ 1249

doi:10.3772/j.issn.1002-0470.2012.12.005

具有 $O(n)$ 消息复杂度的非阻塞检查点算法①

刘国良②*** 陈蜀宇*

(*重庆大学计算机学院 重庆 400044)

(**重庆市计量质量检测研究院 重庆 401123)

摘 要 为了用检查点设置及回卷恢复技术提高并行分布式系统容错性能时降低设置检查点的时间和空间开销,提出了一种非阻塞协调检查点算法。与传统的两阶段提交算法不同,该算法是单阶段提交算法,可跳过临时检查点阶段直接获得永久检查点,减少了同步控制消息的数量,加快了检查点的形成时间。它通过发送进程排除孤儿消息,实现了并行计算;通过设置检查点算法启动周期,解决中途消息问题。该算法的时间复杂度由通常的 $O(n^2)$ 降低到 $O(n)$,只需要 $n-1$ 个同步消息。

关键词 容错,非阻塞检查点,回卷恢复,单阶段提交算法

0 引 言

检查点算法是通过检查点设置(checkpointing)及回卷恢复(rollback recovery)技术,使分布式并行计算系统发生软件/硬件故障后,恢复到保存在可靠存储设备中的进程状态(检查点)并继续执行,避免程序从头开始执行,从而最大限度地减少因故障带来的计算损失和有效提高系统的可用性和容错能力的方法。对协调检查点算法而言,关键是获得全局一致检查点[1,2],而获得全局一致检查点就是处理系统中的孤儿消息(orphan message)和中途消息(in-transit message)。由于协调检查点算法本身即可避免孤儿消息,故获取全局一致状态就是要消除中途消息。以往的协调检查点系统,如 CoCheck 等,基于通信通道的先进先出(FIFO)假设,采用两两进程之间进行"消息驱赶"来清空通信信道。如果设参与检查点的进程数为 n,则在此过程中共有 $O(n^2)$ 个控制消息需要传递。对于大规模分布式应用而言,进程数量常数以千计,这导致控制消息数量极大[3]。更关键的是,各个进程之间的同步性不可避免地会受到所在结点的负载、操作系统的调度、并行应用算法的自身特点等因素的影响而难于保证[4-6]。这些原因使并行应用运行中的全局同步会产生较长时间的阻塞。针对这种情况,本文提出了一种非阻塞检查点算法,其核心思想是通过发送进程来确保不会产生孤儿消息,不需要接收进程的任何信息,一个进程是否获得检查点与其他进程无关,因此各个进程可以独立地以并行方式获得检查点。同时此算法是单阶段算法,相对传统的两阶段提交算法具有明显的优势,并且代价很低,复杂度为 $O(n)$,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。

1 相关背景知识

检查点算法的回卷恢复原理如图 1 所示。当故障发生时,进程就回卷到最近一致检查点 C_2 继续执行,避免了从 C_0 开始执行。

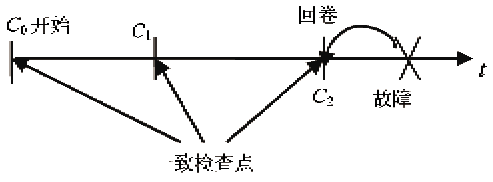

图1 回卷恢复原理

定义 1 全局状态(global state,GS):在一个由 n 个进程构成的分布式程序 $P=\{P_1,P_2,\cdots,P_n\}$ 中,如果每个进程的状态为 $S_i(i=1,2,\cdots,m)$,由

① 重庆市自然科学基金计划(CSTC2008BB2307)资助项目。

② 男,1977 年生,博士生,工程师;研究方向:分布式容错,可信计算;联系人,E-mail:guol_liu@163.com

(收稿日期:2011-03-28)

各个进程状态构成的进程状态集为 $PS = (S_1, S_2, \cdots, S_n)$，进程间通信的信道状态集为 CS，则：全局状态 GS =进程状态集 PS +信道状态集 CS。

定义 2 中途消息(in-transit message)：就分布式系统中的某一全局进程状态 $GS = (S_1, S_2, \cdots, S_n)$ 而言，对于任一消息 m_x，如果其发送者的进程状态 S_i 包含了消息 m_x 的发送事件，而其接收者的进程状态 S_j 却没有包含此消息的接收事件，则消息 m_x 称作中途消息，如图 2 所示。

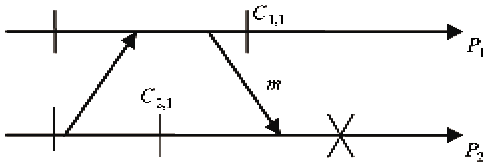

图 2 中途消息

图 2 中，消息 m 即为中途消息，进程 P_1 记录了消息 m 的发送事件，而 P_2 没有记录 m 的接收事件。中途消息的问题是可能导致消息丢失[7,8]，假设进程 P_2 发生故障，回卷到检查点 $C_{2,1}$。在检查点 $C_{1,1}$，进程 P_1 记录消息 m 已经发送，而在检查点 $C_{2,1}$，P_2 并没有消息 m 被接收的记录，这样就导致了消息 m 的丢失。

定义 3 孤儿消息(orphan message)：就分布式系统中的某一全局进程状态 $GS = (S_1, S_2, \cdots, S_n)$ 而言，如果它包含了某一消息的接收事件，但没有包含该消息的发送事件，那么该消息就称作孤儿消息，如图 3 所示。

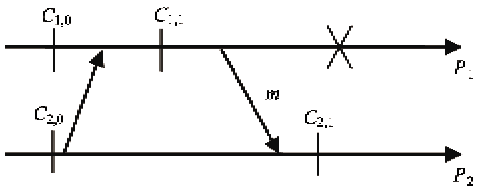

图 3 孤儿消息

图 3 中，对检查点集 $\{C_{1,1}, C_{2,1}\}$ 而言，消息 m 即为孤儿消息，进程 P_2 记录了消息 m 的接收事件，但进程 P_1 没有记录其发送事件。孤儿消息的问题可能导致多米诺(domino)效应[7-10]，假设 P_1 发生故障，回滚到检查点 $C_{1,1}$。回滚使 m 无效，导致 P_2 必须回滚到 $C_{2,0}$，以使接收的 m 无效。如此往复，导致了 domino 效应，使得系统最终回滚到开始处 $(C_{1,0}, C_{2,0})$。

定义 4 全局一致状态(consistent global checkpoint/state)：不存在孤儿消息的全局进程状态 GS[11-14]。

定义 5 强全局一致状态(strongly consistent global checkpoint/state)：既不存在孤儿消息又不存在中途消息的全局进程状态 GS[14]。只有满足强全局一致状态的检查点才可以用于恢复。

2 系统模型

2.1 系统模型

本文提出的检查点算法的应用对象是一些开放的分布式系统，对底层硬件和软件没有特殊要求。系统模型如下：

(1)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；

(2)各结点没有同步的时钟，整个系统也没有全局时钟；

(3)各进程之间仅通过消息进行通信。

2.2 系统定义及相关术语

定义一个分布式并行应用程序 P，$P = \{P_1, P_2, \cdots, P_n\}$，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。我们把启动算法(检查点或回卷恢复算法)的进程叫做启动进程(initiator process)，任何进程都可能成为启动进程，这种设计更符合分布式应用特征。其他进程叫做应用进程(application process)。

为了全文的一致性，我们做如下定义：

S(flag)：发送消息标志，布尔型，表示在一个检查点后是否已发送了消息，值为 1 表示相应进程发送了应用消息，值为 0 表示未发送消息；S_i(flag)表示进程 $P_i(i=1,2,\cdots,n)$ 的发送消息标志。

$C(N)$：检查点序号，整形，每执行一次检查点算法，其值加 1；$C_i(N)$ 表示进程 $P_i(i = 1,2,\cdots,n)$ 的检查点序号。

$S(N)$：发送消息序号，整形，用来惟一标识进程所发送的消息，每发送一条消息，其值加 1；$S_i(N)$ 表示进程 $P_i(i = 1,2,\cdots,n)$ 的消息序号。

$C_{i,x}$：表示进程 $P_i(i = 1,2,\cdots,n)$ 的第 x 个检查点。

$M(C)$：控制消息，执行检查点算法使用。

msg：表示进程发送的应用消息；msg_n 表示进程发送的第 n 个应用消息。

本文将算法中用于协调进程行为的各种消息称为控制消息(control message),包括 $M(C)$;进程之间为实现计算目标而进行通信的消息称为应用消息(application message)。

3 检查点算法

3.1 算法描述

我们把进程分为两类,启动检查点算法的进程叫做启动进程(initiator process),其他进程叫做应用进程(application process)。启动进程只是通过控制消息 $M(C)$ 与应用进程进行交互,如果进程数为 n,则只需要 $n-1$ 个控制消息,远低于已有的算法。使获得检查点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。

该算法是一个单阶段非阻塞算法,它允许进程直接获得永久检查点,而不需要获得临时检查点,大大提高了算法的执行速度,且能够保证每次获得的检查点都是全局一致检查点,大大减少了获得检查点的数量。其实现如下:

Initiator process $P_k(k = 1,\cdots,n)$:

if S_k(flag) = 1 /* 最近检查点后,进程 P_k 至少发送了一条消息 */

creates a subprocess by derivation method; /* 通过派生方法创建一个子进程 */

initiator process continues its normal computing; /* 主进程继续运行 */

subprocess takes a forced checkpoint /* 子进程获得检查点文件 */

S_k(flag) = 0; /* 进程 P_k 发送标志 S_k(flag)置 0 */

$C_k(N) = C_k(N) + 1$; /* 进程 P_k 检查点序号加 1 */

sends $< M(C), C_k(N) >$ to all processes;

continues its normal computing;

else

$C_k(N) = C(N) + 1$;

sends $< M(C), C_k(N) >$ to all processes;

continues its normal computing;

Application process $P_i(i = 1,\cdots,n)$:

if P_i receives $< M(C), C_k(N) >$ /* 接收到启动进程 P_k 发来的控制消息 */

if S_i(flag) = 1

creates a subprocess by derivation method; /* 通过派生方法创建子进程 */

application process continues its normal computing; /* 主进程继续运行 */

subprocess takes a forced checkpoint /* 子进程获得检查点文件 */

S_i(flag) = 0;

$C_i(N) = C_i(N) + 1$;

continues its normal computing;

else

$C_i(N) = C_i(N) + 1$;

continues its normal computing;

else if P_i has not yet received $< M(C), C_k(N) >$ && P_i receives $< \text{msg}_n, C_j(N), S_n(N) >$

if $C_i(N) < C_j(N)$

if S_i(flag) = 1

creates a subprocess by derivation method; /* 通过派生方法创建子进程 */

application process continues its normal computing; /* 主进程继续运行 */

subprocess takes a forced checkpoint without waiting for $< M(C), C_k(N) >$; /* 子进程获得检查点文件 */

S_i(flag) = 0;

$C_i(N) = C_i(N) + 1$;

processes the received message msg_n and ignores $< M(C), C_k(N) >$ when received later;

else

$C_i(N) = C_i(N) + 1$;

processes the received message msg_n and ignores $< M(C), C_k(N) >$ when received later;

else

processes the received message msg_n and ignores $< M(C), C_k(N) >$ when received later;

else

continues its normal computing;

3.2 算法相关结论及正确性证明

根据提出的检查点算法,可得出下面的性质。

定理 1:对给定的任意进程 $P_i(i = 1,2,\cdots,n)$,在任意时刻 t,如果发送消息标志满足 $S_i(\text{flag}) = 0$,则在时刻 t,进程 P_i 已发送的所有消息都不是孤儿消息。

证明:根据算法,假设 $C_{i,x}$ 为该进程 P_i 任意检查点,则时刻 t,要么在检查点 $C_{i,x-1}$ 和 $C_{i,x}$ 之间,要么与检查点 $C_{i,x}$ 时刻重叠,要么在检查点 $C_{i,x}$ 和 $C_{i,x+1}$ 之间,如图 4 所示。

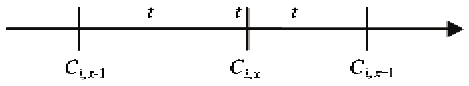

图 4 时刻 t 与检查点 $C_{i,x}$ 间相对位置

讨论:(1)假设 t 位于检查点 $C_{i,x-1}$ 和 $C_{i,x}$ 之间,当发送标志 $S_i(\text{flag}) = 0$,则根据算法可知,在检查点 $C_{i,x-1}$ 到 t 这段时间内,进程 P_i 没有发送任何消息,因而不存在孤儿消息。(2)假设 t 与检查点 $C_{i,x}$ 时刻重叠,当发送标志 $S_i(\text{flag}) = 0$,则根据算法可知,在时刻 t 处,首先获得检查点 $C_{i,x}$,再将标志 $S_i(\text{flag})$ 置0。即记录了进程 P_i 发送的所有消息(在检查点 $C_{i,x-1}$ 和 $C_{i,x}$ 期间),因而不存在孤儿消息。(3)假设 t 位于检查点 $C_{i,x}$ 和 $C_{i,x+1}$ 之间,当发送标志 $S_i(\text{flag}) = 0$,则根据算法可知,在检查点 $C_{i,x}$ 到 t 这段时间内,进程 P_i 没有发送任何消息,因而不存在孤儿消息。综上,定理得证。

定理 2:假如进程 P_k 向进程 P_i 发送应用消息 $< \text{msg}_n, C_k(N), S_n(N) >$,如果进程 P_k 的检查点序号 $C_k(N)$ 与进程 P_i 的检查点序号 $C_i(N)$ 满足 $C_k(N)(=x) > C_i(N)(=x-1)$,则进程 P_k 发送的应用消息 msg_n 不会变成孤儿消息。

证明:当进程 P_i 收到应用消息 $< \text{msg}_n, C_k(N), S_n(N) >$ 后,比较检查点序号 $C_k(N)$ 与 $C_i(N)$,发现 $C_k(N)$ 大于 $C_i(N)$,进程 P_i 知道算法的第 x 次执行已经开始了,并且它会很快收到启动进程的控制消息 $< M(C), C_k(N) >$。进程 P_i 并不等待控制消息 $< M(C), C_k(N) >$,而是根据自身的发送标志 $S_i(\text{flag})$ 来判断是否获得检查点,然后处理收到的应用消息 msg_n,这意味着接收的应用消息 msg_n 没有被接收进程 P_i 记录,根据孤儿消息定义可知,应用消息 msg_n 不会变成孤儿消息。

定理 3:假如进程 P_k 向进程 P_i 发送应用消息 $< \text{msg}_n, C_k(N), S_n(N) >$,如果进程 P_k 的检查点序号 $C_k(N)$ 与进程 P_i 的检查点序号 $C_i(N)$ 满足 $C_k(N) = C_i(N) = x$,则进程 P_k 发送的应用消息 msg_n 不会变成孤儿消息。证明:从略。

定理 4:假如进程 P_k 向进程 P_i 发送应用消息 $< \text{msg}_n, C_k(N), S_n(N) >$,如果进程 P_k 的检查点序号 $C_k(N)$ 与进程 P_i 的检查点序号 $C_i(N)$ 满足 $C_k(N)(=x-1) < C_i(N)(=x)$,则进程 P_k 发送的应用消息 msg_n 不会变成孤儿消息。证明:从略。

算法正确性证明如下:

对启动进程 $P_k(k = 1,2,\cdots,n)$ 而言,假如其发送消息标志 $S_k(\text{flag}) = 1$,即满足 if 条件,那么该进程首先获得检查点,然后把标志 $S_k(\text{flag})$ 置为0,因此,只要检查点获得成功,标志 $S_k(\text{flag})$ 必为0,根据定理1可知,进程 P_k 已发送的所有消息都不是孤儿消息;假如不满足 if 条件,即进入 else 段代码,此时,发送标志 $S_k(\text{flag})$ 必为0,根据定理1可知,进程 P_k 已发送的所有消息都不是孤儿消息。

对应用进程 $P_i(i = 1,2,\cdots,n)$ 而言

(1)假如应用进程 $P_i(i = 1,2,\cdots,n)$ 收到启动进程 $P_k(k = 1,2,\cdots,n)$ 发送的控制消息 $< M(C), C_k(N) >$,则满足第一个 if 条件,类似启动进程,假如其发送消息标志 $S_i(\text{flag}) = 1$,即满足 if 条件,那么该进程首先获得检查点,然后把标志 $S_i(\text{flag})$ 置为0,因此,只要检查点获得成功,标志 $S_i(\text{flag})$ 必为0,根据定理1可知,进程 P_i 已发送的所有消息都不是孤儿消息;假如不满足 if 条件,即进入 else 段代码,此时,发送标志 $S_i(\text{flag})$ 必为0,根据定理1可知,进程 P_i 已发送的所有消息都不是孤儿消息。

(2)假如应用进程 $P_i(i = 1,2,\cdots,n)$ 没有收到启动进程 $P_k(k = 1,2,\cdots,n)$ 发送的控制消息 $< M(C), C_k(N) >$,但是收到了其他进程 $P_j(j = 1,2,\cdots,n)$(包括发送进程 P_k)发送的应用消息 $< \text{msg}_n, C_j(N), S_n(N) >$,即满足 else if 条件,如果条件 $C_i(N) < C_j(N)$ 成立,则进入 if 语句,根据定理2可知,进程 P_j 发送的任何应用消息 $< \text{msg}_n, C_j(N), S_n(N) >$ 均不会变成孤儿消息,即应用进程 P_i 在接收控制消息 $< M(C), C_k(N) >$ 之前接收的任何进程的任意应用消息 $< \text{msg}_n, C_j(N), S_n(N) >$ 均不会变成孤儿消息;如果条件 $C_i(N) < C_j(N)$ 不成立,则进入 else 语句,必满足定理3,定理4,进程 P_j 发送的任何应用消息 $< \text{msg}_n, C_j(N), S_n(N) >$ 均不会变成孤儿消息。

(3)如果既不满足(1)也不满足(2),则程序进入最后一个 else 语句,即进程 P_i 正常执行,不会获得检查点,当然也没有孤儿消息。

综上,对任意的进程 $P_i(i = 1,2,\cdots,n)$,在检查点处,均没有孤儿消息存在,所以,通过该算法获得的检查点集一定是全局一致检查点。

3.3 主存方法

采用文献[15]相似思想,本文也采用主存算法方式来提高获得检查点的速度,当进程需要获得检查点文件时,首先派生一个子进程,然后主进程继续运行,由子进程将自身状态写入磁盘形成检查点文件。算法中描述如下:

```
//takes a forced checkpoint;
creates a subprocess by derivation method; /* 通过派生方法创建一个子进程 */
primary process continues its normal computing; /* 主进程继续运行 */
```

subprocess takes a forced checkpoint /* 子进程获得检查点文件 */

通过主存算法,使用户进程获得检查点的时间从进程状态的磁盘拷贝时间减少到进程派生子进程的时间。

3.4 启动周期

算法启动周期 T 的选择需要综合考虑,如果太短,就会导致算法频繁启动,增加额外负载;如果太长,发生故障后,又会导致丢失的计算量较大;另外还需要有利于回卷恢复算法的实现。我们的算法设系统中任意两进程间传送消息的时间为 $T_{i,j}$,$(i \neq j)$,两进程间传送消息需要的最大时间为 $T_{\max} = \{T_{i,j}, i \neq j\}$,则算法启动周期 T 取略大于 $T_{\max} = \{T_{i,j}, i \neq j\}$。选择该启动周期的理由是为了尽量减少记录中途消息的数量(处理中途消息的常用方法是消息日志方法),由于启动周期大于 $T_{\max}$,所以一个进程 P_i 只需要在它的最近检查点 $C_{i,x}$ 保存检查点间隔 $(C_{i,x-1}, C_{i,x})$ 期间内所发送的消息,如图 5 所示。

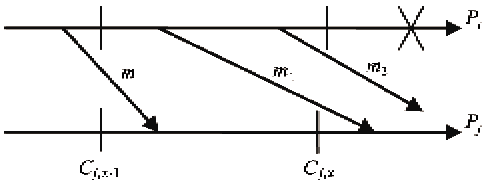

图 5 消息 m 不会成为中途消息

图 5 中,我们考虑两个进程 P_i 和 P_j,由于算法启动周期 T 大于 $T_{\max} = \{T_{i,j}, i \neq j\}$,对进程 P_i 来说,在检查点间隔 $(C_{i,x-2}, C_{i,x-1})$ 期间发送的任意消息 m 都会在进程 P_j 的最近检查点 $C_{j,x}$ 获得之前到达进程 P_j。现在假设进程 P_i 发生故障,两个进程在恢复算法的作用下回卷到它们的最近检查点 $\{C_{i,x}, C_{j,x}\}$ 并重新计算,由于消息 m 在进程 P_j 的检查点 $C_{j,x}$ 之前已经被接收,即 m 不会变成中途消息,所以不需要进行重传。因此在发生进程 P_i 的最近检查点 $C_{i,x}$ 就不需要保存日志信息。相反,图 5 中的消息 m_1 及 m_2 均可能变成中途消息,所以需要在发送进程 P_i 的最近检查点 $C_{i,x}$ 处保存其日志信息,在恢复的时候进行重传,防止中途消息丢失。

3.5 时间复杂度

我们提出的算法,只是启动进程通过控制消息 $M(C)$ 与应用进程进行交互,如果进程数为 n,则只需要 $n-1$ 个控制消息;其他应用进程只是根据自己的发送标识 $S(\text{flag})$ 判断是否获得检查点,不需要其他进程的任何额外消息,时间复杂度为 $O(n)$,远低于传统的 $O(n^2)$。

3.6 算法优势

该算法的优势主要体现在:(1)在检查点计算期间,只有那些发送了应用消息(一条或多条)的进程才会获得检查点,因此,减少了获得检查点的数量;(2)各个进程是否获得检查点,只根据自己的发送标志 $S(\text{flag})$ 进行判断,不需要其他进程的任何消息,各个进程可以独立、同时获得检查点,具有很好的并行特性;(3)该算法是单阶段算法,与传统的两阶段提交算法相比,大大提高了算法的执行速度;(4)该算法是非阻塞算法,获得检查点引入的控制消息复杂度只有 $O(n)$,其代价很低;(5)通过巧妙地设置算法启动周期,很好地解决了中途消息问题,使得回卷恢复变得非常简单。

4 性能评估

为了评估算法,有必要同该领域中一些典型的算法[1,16,17]进行比较,主要从两个方面进行比较:算法是否阻塞及同步控制消息的费用。为了更清楚地说明,首先我们作如下的定义:

C_{send} ——从一个进程发生一条消息到另一个进程的费用;

C_{broad} ——从一个进程广播一条消息到所有进程的费用;

$N_{\min}$ ——需要获得检查点的进程数;

n ——系统中的进程数。

根据上述定义,算法比较如表 1 所示。

表 1 算法比较

算法	是否阻塞	控制消息费用
文献[1]	是	$3 \cdot C_{\text{broad}}$
文献[16]	否	$2 \cdot N_{\min} \cdot C_{\text{send}} + \min(N_{\min} \cdot C_{\text{send}}, C_{\text{broad}})$
文献[17]	是	$2 \cdot C_{\text{broad}} + n \cdot C_{\text{send}}$
本文算法	否	C_{broad}

从表 1 中可知,文献[1,17]是阻塞算法,因此只有当进程获得了永久检查点后才能进行正常计算。在控制消息费用方面,文献[16]采用的是两阶段提交方式,首先一个进程发送 2 个控制消息来获得临时检查点,系统负载为 $2 \cdot N_{\min} \cdot C_{\text{send}}$,在第二阶段从临时检查点转变为永久检查点,消息负载为 $\min(N_{\min} \cdot C_{\text{send}}, C_{\text{broad}})$,因此总费用为 $2 \cdot N_{\min} \cdot$

$C_{send} + \min(N_{min} \cdot C_{send}, C_{broad})$。同理,其他算法费用可以计算,如表 1 所示。由于我们的算法是单阶段算法,即直接获得永久检查点,并且只需要广播一条请求消息,因此总费用是 C_{broad}。

算法的另一个重要方面是,当进程数较多时,算法是否适用。这个问题主要归结于进程数与控制消息数的关系,即进程数较多时,算法的负载应该比较低。我们同文献[16,17]进行比较,其中 C_{broad} 等于 $n \times C_{send}$,对文献[16],假设平均获得检查点进程数为总进程的 80%,结果如图 6 所示。

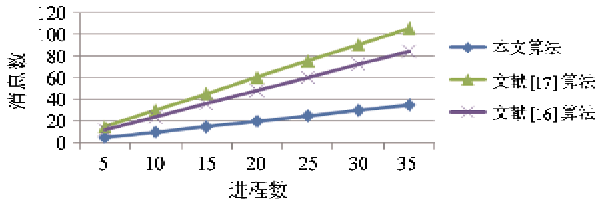

图 6 进程数量与消息数量

从图 6 可以看出,随着进程数量的增长,需要的控制消息数也随之增长,三种算法中的协调控制消息数都是按线性规律增长。设系统中进程数为 n,文献[16]中算法大约需要 $3n \cdot 80\%$ 条控制消息,文献[17]需要 $3n$ 条控制消息,而本文检查点算法只需要 $n-1$ 条控制消息消息。

算法的执行时间也很重要。在节点间带宽为 10Mbps,操作系统为 Windows XP,每个节点均只包含一个进程的条件下,通过仿真实验,我们获得的进程数量与算法的执行时间如图 7 所示,其中时间单位为毫秒。

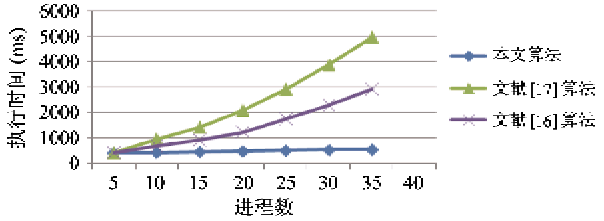

图 7 进程数与执行时间

从图 7 中可以看出,本文算法的执行时间明显优于文献[16,17]的算法,算法的执行时间开销基本上均在毫秒级,与并行程序的运行时间相比是微不足道的。随着并行进程规模的扩大,检查点的时间开销增长很慢,基本是按线性增长,能够适应大规模的分布式并行程序。

5 结 论

本文提出了一个非阻塞协调检查点算法,接收消息进程不必担心接收到的消息是否会变成孤儿消息,孤儿消息的排除是通过发送消息进程来实现的。算法内在的并行特性,提高了算法的执行效率。算法是单阶段提交算法,直接获得永久检查点,跳过了临时检查点阶段,减少了控制消息的数量,加快了检查点的形成时间。通过设置检查点算法启动周期,解决了中途消息问题。算法的时间复杂度由通常的 $O(n^2)$ 降低到 $O(n)$。实验验证了该算法的有效性,该算法能够适应大规模的分布式并行应用。今后将在更大规模系统环境下对此算法进行实验,进一步验证其有效性,同时考虑将检查点算法与操作系统的调度算法结合起来,以提高系统的容错能力。

参考文献

[1] Monnet S, Morin C, Badrinath R. Hybrid checkpointing for parallel applications in cluster federations. In: Proceedings of the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid , Chicago, USA, 2004.773-782

[2] Gupta B, Rahimi S, Ahmad R. A new roll-forward checkpointing recovery mechanism for cluster federation. *International Journal of Computer Science and Network Security*, 2006,6(11):292-298

[3] Gupta B, Rahimi S, Yang Y X. A novel roll-back mechanism for performance enhancement of asynchronous checkpointing and recovery. Informatica (Slovenia), 2007,31(1): 1-13

[4] Elnozahy N, Alvisi L, Wang Y M, et al. A Survey of Rollback-recovery protocols in message-passing system. *ACM Computing Surveys*, 2002,34(3): 375-408

[5] Nishanth C, Suman K M. A two level Cache based checkpointing and Rollback recovery scheme using multiple epochs. Texas: TR-Texas A&M University, 2006.102-108

[6] Bosilca G, Delmas R, Dongarra J, et al. Algorithm-based fault tolerance applied to high performance computing. *Journal Parallel Distributed Computer*, 2009, 69(4): 410-416

[7] Jim S, Paul W. Applying low-overhead rollback- recovery to wide area distributed query processing:[Technical Report CS-TR-861]. New South Wales: University of Newcastle, 2004,10-17

[8] Gupta S K, Chauhan R K, Kumar P. Backward error recovery protocols in distributed mobile systems: A survey.

Journal of Theoretical and Applied Information Technology, 2008, 30(4): 225-240

[9] Claudia R, Cristian G, Lorena A. Blocking and non-blocking checkpointing and rollback recovery for networks-on-chip. In: Proceedings of 208 IEEE International Symposium on Dependable and Secure Nanocomputing, AK, USA, 2008. 72-78

[10] Manivannan D. Checkpointing and rollback recovery in distributed systems: Existing solutions, open issues and proposed solutions. In: Proceedings of the 12th WSEAS International Conference on Systems, Heraklion, Crete Island, Greece, 2008. 22-24

[11] Raphael Y C, Andrei G. Checkpointing-based rollback recovery for parallel applications on the InteGrade grid middleware. In: Proceeding of the 2nd Workshop on Middleware for Grid Computing, New York, USA, 2004. 35-40

[12] Janakiraman G, Tamir Y. Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputers. In: Proceeding of the 13th Symposium on Reliable Distributed Systems, California, USA, 1994. 42-51

[13] Gupta B, Rahimi S, Ziping L. Design of high performance distributed snapshot recovery algorithms for ring networks. *Journal of Computing and Information Technology*, 2008, 16(1): 23-28

[14] Mani K, Chan D, Leslie L. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 1985, 3(1): 63-75

[15] Wei XH, Ju JB. SFT: A consistent checkpointing algorithm with short freezing time. *Journal of Computers*, 1999, 22(6): 645-649

[16] Cao G, Singhal M. Mutable checkpoints: a new checkpointing approach for mobile computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 2001, 12(2): 157-172

[17] Wang D S, Shao M L. A cooperative checkpointing algorithm with message complexity $O(n)$. *Journal of Software*, 2003, 14(1): 43-48

A non-blocking checkpointing algorithm with message complexity $O(n)$

Liu Guoliang[*,**], Chen shuyu[*]

([*] College of Computer Science, Chongqing University, Chongqing 400044)

([**] Chongqing Academy of Metrology and Quality Inspection, Chongqing 401123)

Abstract

In order to reduce the time and space overheads for setting checkpoints when using the checkpointing-rollback recovery technique to enhance the error-tolerance performance of parallel and distributed systems, a non-blocking cooperative checkpointing algorithm is proposed. It is a one-phase method. Different from the conventional (two-phase) approach, it jumps over the temporary checkpoint phase to obtain permanent checkpoints to reduce the number of controlled messages and the time of checkpoint forming. The proposed checkpointing algorithm allows processes to take permanent checkpoints directly, without taking temporary checkpoints. The character of the algorithm contributes to its speed of execution. Orphan messages are eliminated by sender processes and in-transit messages are eliminated by the checkpointing interval and retransmission mechanism. Its time complexity for obtaining checkpoints is $O(n)$, not the ordinary $O(n^2)$.

Key words: fault tolerance, non-blocking checkpointing, rollback recovery, single phase commit algorithm