并行调试环境中的追踪和重演*

张钦伍

(国防科技大学计算机系 长沙 410073)

张 柳

(军械工程学院维修工程研究所 石家庄 050003)

摘 要 通过传递消息进行通信的并行程序,由于受进程调度和消息等待时间变化的影响,其执行结果可能是不确定的。这导致调试时相继执行不能再现先前的故障,为此,在并行调试环境中引入了追踪和重演机制。本文详细讨论了一种比较先进的追踪和重演算法及其工程实现。

关键词 并行调试,追踪,重演

分类号 TP312

Tracing and Replaying in Parallel Debugging Environments

Zhang Qinwu

(Department of Computer, NUDT, Changsha, 410073)

Zhang Liu

(Maintenance Engineering Institute, Ordnance Engineering College, Shijiazhuang, 050003)

Abstract Executing result of parallel programs which communicate by passing messages, can be nondeterministic due to variations in process scheduling and message latencies. Such nondeterminacy can cause serious problems while debugging: subsequent executions of the program may not reproduce the original bug. Therefore, a tracing and replaying mechanism is introduced into parallel debugging environments. This paper discussed an advanced algorithm for tracing and replaying, and its implementation.

Key words parallel debugging, tracing, replaying

不管哪种并行程序(spmd 或mpmd)执行起来都有多个进程,并且进程之间有通信。根据进程间的通信方法,并行程序又可分为:

(1) 消息传递并行程序。进程之间通过发送和接收消息进行通信。

(2) 数据并行并行程序。进程之间通过共享虚存数据而完成信息交换。

对消息传递并行程序而言,由于一些内在的和外部的原因,一个程序在相同输入之下的两次执行流程正常结束,但结果不同。这就是所谓的“不确定性”(nondeterminacy)。在调试时,它会使程序的相继执行不能再现先前执行产生的故障,即故障“不可重复性”(non_repeatability)。于是以反复执行程序重复再现故障为核心的传统调试技术在消息传递并行程序面前遇到了挑战。

为此,在并行程序调试环境中引入了追踪和重演机制,来确保消息传递并行程序在调试时是可重复执行的,于是,追踪和重演的算法便成了人们关注和研究的焦点。

1 追踪算法

追踪是在并行程序的原始执行过程中,记录一些必要的信息,以便后来能根据这些信息把程序的原

* 1998年4月5日收稿
第一作者:张钦伍,男,1938年生,研究员

始执行重演出来。追踪算法研究的是记录什么和什么时候记录。

1.1 “不确定性”产生的原因

深入分析消息传递机制本身和消息传递并行程序执行的过程,不难发现产生“不确定性”的原因,有内在的也有外部的。

(1) 来自消息传递机制本身的内在原因

消息传递机制中的接收操作(无论是阻塞的还是非阻塞的)通常都可以被设置为如下三种类型:

A: 接收任何进程发来的任何消息; B: 接收指定进程发来的任何消息; C: 接收带有特定标志的消息。

显然,由于C类接收操作接收的消息是明确指定的,所以每次执行中是不会变的,而B类接收操作虽然接收的消息没有明确指定,但这些消息都来自同一个进程,同一个进程中各消息的发送顺序是由该进程的逻辑结构确定的,在相同输入的条件下是不会变的,因此,B类接收操作在相同输入的条件下,每次执行接收的消息也是不变的。问题出在A类接收操作,它对来自不同进程的消息不作辨认,谁先到就接收谁,这就是产生“不确定性”的内因。

(2) 与系统当前运行有关的外部原因

在程序当前的执行中,系统对各进程的调度运行以及造成消息传递中消息等待的时间不可能与上一次程序的执行完全一样,这种变化就有可能(并非一定)改变了某些消息到达A类接收操作的先后顺序。外因通过内因起了作用,使在相同输入的条件下,前后两次执行,由于A类接收操作接收的消息不同而产生不同的执行结果。

为消除“不确定性”,最自然的办法是在原始执行中每个接收操作接收消息时记下自己所接收消息的“特征”,在以后的重复执行中根据所记“特征”使每个接收操作非指定的消息不接。然而,实践很快告诉人们[1,2]: 这种对每一个消息都进行追踪的办法有时时空开销大得使系统无法承受而不能调试,于是,研究的焦点进一步指向了如何减少追踪信息量。

1.2 消息的竞争和竞态消息

虽然“不确定性”是由于A类接收操作接收了不同的消息而产生的,但并不是每个A类接收操作都能引起“不确定性”。一个接收操作能引起“不确定性”的本质条件是: 一组同时在传递的消息,其中任何一个都可能首先到达该接收操作,此时,我们称这组消息是相对该接收操作竞争的。尽管一组竞争消息的每一个都可能首先到达该接收操作,但在一次具体执行中只有一个先到达该接收操作并被接收,后到达的其它消息将被稍后的接收操作接收,我们称这些消息是先到达消息的竞态消息。

显然,只有竞态消息才能引起“不确定性”。因此只需追踪竞态消息。

1.3 事件间的关系

在消息传递并行程序的进程中,进行消息传递的是发送操作和接收操作,称之为同步事件。同一个进程中的同步事件发生的顺序是由本进程的逻辑控制结构决定的,不受外部因素的影响。在执行中可用整数1,2,3…给发生的同步事件依次编号,称为事件顺序号。于是,同一进程中的两个事件发生的先后可根据事件顺序号的大小来判断。

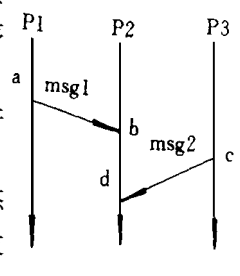

图1

不同进程中的两个事件,能否比较发生的先后,如何比较?首先,有些同步事件之间是没有确定的先后关系的。如图1中,进程P2的b事件与进程P3的c事件,就是不确定的。实际上,消息msg1和msg2相对b是竞争的,msg2是msg1的竞态消息。其次,有些同步事件之间是有确定的先后关系的。最明显的是同一消息的发送事件先于接收事件。借助于这一垮越两个进程的“单向桥梁”,再利用同一进程中事件顺序号的大小和先后关系的传递性,就形成了不同进程中一些事件之间发生先后的确定顺序。如图2中,各事件发生的先后顺序依次为: a→b→e→f→c→d。

1.4 竞态检测算法

只追踪竞态消息的关键技术是如何识别它。由于这种识别只能在执行中动态进行,所以称之为飞行式判定(on_the_fly)[3]。

观察进程 P2 的运行。当 b 事件发生(接收消息 msg1)时,由于此前再没有接收操作接收过消息,因而 msg1 是非竞态的。当 d 事件发生(接收消息 msg2)时,由于前面已有接收操作接收了消息 msg1,此时 msg2 是否与 msg1 发生竞争是需要检测的。为此,只要看 msg2 有无可能先于 msg1 到达 b,也就是检查事件 b 和 c 的发生顺序。若能判定 b 肯定发生在 c 之前(如图 2),那么 msg2 也就肯定不会到达 b,因而是非竞态消息;若 b 和 c 的关系是不确定的(如图 1),c 就有可能发生在 b 之前,msg2 就有可能先于 msg1 到达 b,因而是竞态的。这就是该算法的核心,即每当进程接收消息时检测前一消息的接收事件与当前消息的发送事件之间是否有确定的先后关系。下面是该算法在追踪系统中的实现:

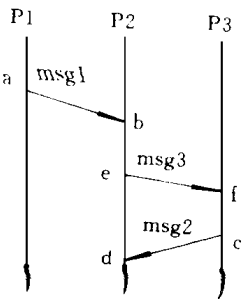

图 2

(1) 在每个进程中,增设两个变量和两个向量:

1) int sym_event_counter= 0,是该进程中的同步事件计数器,初值为 0,每遇到一个接收或发送操作其值增 1。

2) int last_recv= 0,只保存接收操作的事件号。

3) int relation_vector[n],是一关系向量。其中 n 为进程个数。该向量在一定程度反映了不同进程之间事件的顺序关系。对进程 P,relation_vector[P] 是该进程当前同步事件顺序号;relation_vector[i](i≠P)是第 i 个进程中,在 relation_vector[P] 事件发生前发生的同步事件中最大的顺序号。

4) int relation_vector_in[n],该向量只作为临时保存卸出的 relation_vector 向量用。

(2) 在每个发送操作之前,增加下面动作:

1) sym_event_counter= sym_event_counter+ 1;

2) 设当前进程为 P:relation_vector[p]= sym_event_counter;

3) 将向量 relation_vector 的内容附在用户消息之后,以便随消息走。

(3) 在每个接收操作之后,增加下面动作:

1) 从接收的消息中,把随消息带来的发送进程的关系向量卸在 relation_vector_in 中;

2) sym_vevnt- counter= sym_event_counter+ 1;

3) 设当前进程为 P,比较 last_recv 和 relation_vector_in[P],若 last_recv> relation_vector_in[P],则存在竞争,调用追踪子程序追踪该消息,否则,不存在竞争,不追踪;

4) last_recv= sym_event_counter;

5) 若 relation_vector_in[i]> relation_vector[i],

则 relation_vector[i]= relation_vector_in[i](i= 0,1,…,n- 1);

6) relation_vector[P]= sym_event_counter;

1.5 追踪子程序

追踪子程序的任务是在追踪文件中记录竞态消息的必要信息,以便重演时利用。究竟要记录哪些信息,这与重演的算法有关,通常有下面四个信息便足够了。

· 接收者的进程号(或任务号)和接收事件的顺序号;

· 发送者的进程号(或任务号)和发送事件的顺序号。

因为重演时要使那些竞态消息的发送和接收按追踪时的具体情况相对固定,这是通过给发送操作和接收操作加特定标志实现的,尽管在这一点上不同的消息传递系统略有差别,但上面四个信息用于改写追踪文件和形成特定标志是足够的。

2 重演

重演是强制消息传递并行程序按其原始执行重复执行,即消息的发送者和实际接收者与原始执行完全一样,以便调试时,故障可再现。其实,重演时真正需要强制的只有追踪了的消息,即追踪文件中记录的那些消息,没被追踪的消息不用强制就会自动正确重复执行。

2.1 追踪文件的整理

重演系统在对源程序中竞争消息的发送操作和接收操作进行强制性改造时,所需的信息都在追踪文件中记着,每当需要时自然要到这些文件中去读取。

用 TF_i(i= 0, 1, …, n- 1)表示前述追踪算法所产生的追踪文件。每个文件都是按相应进程中接收事件顺序号的增序形成的,若依次从 TF_i 中读出每次所记的接收事件顺序号组成一个序列,那么它应该是该进程中依次产生的全部接收事件顺序号序列的一个子序列。这说明 TF_i 文件的结构特别适合重演系统识别相应进程中一个接收操作是不是追踪消息的接收操作,因为此时只需从相应的 TF_i 中依次读取信息而不涉及别的 TF_i。但重演系统也必须识别一个进程中的发送操作是不是一个追踪消息的发送操作,此时所需信息在相应的 TF_i 中是找不到的,而是分散在其它的 TF_i 中。

为使重演系统有效地工作,首先要对 TF_i(i= 0, 1, …, n- 1)进行一番整理,形成另一组文件,记作 CTF_i(i= 0, 1, …, n- 1)。CTF_i 与 TF_i 的内容基本一样,只是其记录的顺序是按相应进程中发送事件顺序号的增序形成的。若依次从 CTF_i 中读出每次所记的发送事件顺序号组成一个序列,那么它应该是相应进程中依次产生的全部发送事件顺序号序列的一个子序列。

2.2 重演算法

这里讨论的重演算法仍然是以端口方式消息传递系统(如 PVM)为基础。在这种系统中,实现消息发送和接收准确对应的办法通常是给发送操作和接收操作同时加特定标志,即明确指定消息类型参数,不用通配符,因此,为使竞态消息的发送和接收按追踪执行时准确对应,只要利用进程号(或任务号)和事件顺序号形成特定标志,给消息的发送操作和接收操作同时加上便可。由于有 TF_i 和 CTF_i 两个文件,很容易判断哪些发送和接收操作应该加,哪些不应该加。下面是重演算法的实现:

(1) 在每个进程中增设如下几个变量:

1) int counter= 0,是该进程中同步事件计数器,初值为0,每遇到一个发送或接收操作时其值增1。

2) int CTF_i_SPN, CTF_i_SESN, CTF_i_RPN, CTF_i_RESN,存放从 CTF_i 中一次读取的信息,分别是发送进程号,发送事件顺序号,接收进程号,接收事件顺序号。程序开始时先读取一次。

3) int TF_i_SPN, TF_i_SESN, TF_i_RPN, TF_i_RESN,存放从 TF_i 一次读取的信息,其它同上。

(2) 在每个发送操作之前,增加下面动作:

1) counter= counter+ 1;

2) 若 counter= CTF_i_SESN,则 a) 形成特定标志(比如就用 CTF_i_RESN),给发送操作加上; b) 从 CTF_i 中读出下一个记录。

(3) 在每个接收事件之前,增加下面动作:

1) counter= counter+ 1;

2) 若 counter= TF_i_RESN,则 a) 形成特定标志(比如就用 TF_i_RESN),给接收操作加上; b) 从 TF_i 中读出下一个记录。

3 结束语

这里论述的追踪和重演算法,从工程实现的角度看对端口式通信(如 PVM)是切实可行的。在邮箱式通信中,情况要复杂许多,要实现类似的算法,则要适当修改邮箱机制。

参考文献

1 Curtis R, Wittie L. BugNet: A debugging system for parallel programming environment. in Proc. 3rd Int. conf. Distrib. comput. syst, Miumi, FL, 394～399, Qct, 1982

2 Thomas J LeBlanc, MellorCrummey J M. Debugging Parallel programs with Instant Replay. IEEE Trans. on computers, 1987, C36 (4): 471～482

3 Netzer R H B, Miller B P. Optimal Tracing and Replay for Debugging Message passing parallel programs. IEEE, 1992