分布式死锁检测算法中伪死锁率的研究和改进*)

吴 堃 戴 菡 周竞扬 陆桑璐 陈道蓄 谢 立

(南京大学计算机软件新技术国家重点实验室 南京大学计算机科学与技术系 南京210093)

摘 要 死锁处理是分布式系统中的关键问题,其中处理死锁最主要的手段为死锁检测。在评价死锁检测算法性能时伪死锁率被视为一项重要指标,故降低伪死锁率对提高算法性能有着促进作用,而目前大多数算法改进对伪死锁率关注较少。本文阐述了伪死锁研究的意义,并对若干种死锁检测算法的伪死锁率进行研究和模拟实验,认为现有的死锁算法可分为两类:环内检测和环无关检测。并分别通过减少冗余消息和本地死锁解决两种改进方法来降低目前算法的伪死锁率,最终实验表明算法性能获得较大提高。

关键词 分布式系统,分布式算法,死锁检测,伪死锁

Research and Improvements on Phantom Deadlock Rate of Deadlock Detection Algorithms

WU Kun DAI Han ZHOU Jing-Yang LU Sang-Lu CHEN Dao-Xu XIE Li

(National Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing 210093)

Abstract Deadlock detection and resolution is a quite important problem in distributed systems. But now all of the deadlock detection algorithms can cause false deadlock. It is necessary to reduce the false deadlock rate to improve the performance of the algorithms. This paper focuses on the research of the false deadlock. We analyze the factors affecting the false deadlock rate, and propose two improvements to reduce the false deadlock rate. To verify our ideas, simulation experiments are used, and the result is quite satisfied.

Keywords Distributed systems, Distributed algorithms, Deadlock detection, False deadlock

1 引言

分布式系统的出现解决了现在用户对计算速度、系统可靠性和成本性需求的快速增长的需求,但死锁问题却是分布式系统一个需要解决的重要问题。在分布式系统中,若某进程在等待其它进程已经占有的资源,则称该进程被其它进程阻塞。当一组进程中的某些进程由于无法访问组中其它进程占用的资源而形成无限期循环阻塞,则产生死锁[6],进程之间的循环阻塞形成的进程环被称为死锁环。

为处理死锁问题,研究人员提出三种死锁处理策略:死锁预防、死锁避免和死锁检测[1]。死锁预防和避免都是在死锁发生前就尽量避免死锁发生的悲观策略,它们的使用会降低系统的并发性以及需要大量的资源和昂贵的计算开销,所以在多数情况下分布式系统中并不采用这两种方法。而死锁检测是在死锁出现后对其进行检测和解决的乐观策略,它能与系统的正常活动并行而避免过度增加系统的负荷,故死锁检测使用最为普遍。

死锁检测方法可分为三类:集中式,分布式和层次式[10]。其中集中式算法存在单点失效问题,即若死锁检测者发生故障,则死锁检测操作失败。层次式死锁检测将站点组织成层次结构,每个站点仅负责探测其子站点相关的死锁,但对于多数死锁环不在本地站点,而是跨越多个站点的情况,层次式死锁检测不太适用。分布式算法可靠性较好,算法相对复杂,但较适于分布式系统中进行的死锁检测,因此研究较多,很多分布式死锁检测算法已被提出[1,2,5,10]。Knapp 将分布式死锁检测算法分为四类[4]:路径推动算法(Path-Pushing),边界跟踪算法(Edge-Chasing),扩散计算(Diffusing Computation)和全局状态检测(Global State Detection)。在本文研究背景部分,将详细介绍这几类算法的特点及其代表算法。

Singhal 在文[3]中指出,分布式死锁检测算法需进一步研究的方向包括算法的正确性(Algorithm Correctness),算法性能(Algorithm Performance),死锁的解决(Deadlock Resolution)等,这些方向都为现在分布式死锁检测解决算法研究人员所关注,并用于评估死锁检测算法。对一个分布式死锁检测算法而言,算法性能是一个重要的评估的标准,一个算法性能好的死锁检测算法应尽可能少地发生伪死锁,同时有较低的死锁持续时间(Deadlock Persistence Time)以及在系统中消息流量较低。本文中将主要关注伪死锁,同时也将死锁持续时间和消息流量作为参考标准。所谓伪死锁是指死锁检测算法发现的死锁在系统中其实并不存在。比如死锁检测算法在发起死锁检测的时候会出现针对同一个死锁在短期内同时发起多个死锁检测的情况(多重启动问题),其中某个检测发现死锁后将死锁解决,但其它检测因为无法获知系统的全局状态信息仍认为该死锁存在而继续检测,假设再次检测到该死锁而再次解除死锁则伪死锁发生。伪死锁的存在会导致正常工作的进程被视为死锁进程而被解决,从而严重影响系统的性能。而死锁持续时间是指一个死锁从形成到成功检测的时间延迟。死锁的长时间存在会导致资源使用的浪费和对用户请求的反应时间的增加[9]。系统消息流量则是指为了发现和解决死锁造成的网络消息负荷,这也会影响算法性能。

*)基金项目:国家高技术研究发展计划863项目(No. 2001AA113050)。吴 堃 硕士研究生,研究方向为并行处理;戴 菡 硕士研究生,研究方向为并行处理;周竞扬 硕士研究生,研究方向为并行处理。陆桑璐 教授,研究方向为并行处理;陈道蓄 教授,博士生导师,研究方向为并行处理;谢 立 教授,博士生导师,研究方向为并行处理。

为更好地研究死锁检测问题，研究人员引入等待图(Wait-For-Graph，简称 WFG)模型。等待图是一种资源请求的数学模型[4]，其中顶点表示系统中的相关进程，有向边表示两个进程之间有被阻塞关系，如 $A \rightarrow B$ 表示进程 A 被进程 B 阻塞。等待图在死锁的分析和检测解决中应用相当广泛。死锁检测包括发现静态情形(Static Condition)——一旦死锁环形成，那么它将一直存在直至死锁环被打破。另一方面，死锁解决是一个动态过程——它通过删除边和节点来改变等待图。资源等待向等待图增加边，死锁解决删除等待图中的边，两种操作的效果相反。所以若死锁解决不能和死锁检测较好地结合，就可能产生伪死锁(False Deadlock)[9]。

在本文中，对现有的分布式算法进行改进以降低其伪死锁率。文中选择了两种分布式算法作为验证对象，为了体现一般性，选择了一种基于消息的死锁检测算法，同时选择了一种基于移动代理(mobile agent)的死锁检测算法。先通过实验对这两种算法的原来的性能(特别是伪死锁率)进行了评估，然后对这两种算法分别进行了改进，并分析改进结果。

文章第2节介绍研究背景；第3节将对实验算法进行模拟检测并发现伪死锁的一些规律，并提出改进思路；第4节将介绍算法改进后的模拟实验结果；最后给出结论。

2 研究背景

分布式系统是由一些处理进程和资源组成的。处理进程分布在若干个站点上，每个进程(如进程 T)根据某些标准预设一个优先级(如 $P(T)$)。进程间和站点间通过消息或者移动代理进行信息的传递。一般认为在模型中站点内部的进程间通讯要比站点间通讯的网络延时要短。每个站点存在一个站点管理者负责协调本站点内死锁检测的发起。每个处理进程可能同时需要多个资源，只有当所需要的资源都获得(Granted)后，该进程才能进行处理并在处理结束后将资源释放。例如进程 A 申请的资源 R 正被进程 B 占有，进程 A 必须等待进程 B 释放资源 R，称进程 A 被进程 B 阻塞，或者说进程 B 阻塞了进程 A，那么可将 B 表示为 $Block(B)$，将 A 表示为 $Block\_on(A)$。

如引言部分介绍的，Knapp 将分布式死锁检测算法分成四类[4]：

①路径推动算法(Path-pushing algorithms)，这类算法试图在每个站点建立全局等待图。当发起死锁检测时，死锁检测发起站点将自身等待图的拷贝发送给一定数量的邻接站点。这些站点收到后，进行局部拷贝更新然后继续传播。一直重复该过程直到其中某个站点得到足够的信息来判断是否存在死锁。比较典型的路径推动算法有 Obermarck 算法等。该类算法的伪死锁率是所有种类算法中相对较高的。本类算法的优势在于可以通过一次检测就获知死锁的存在和死锁环的组成进程，但由于每次需要传递整个本地等待图，消息量极大。

②边界跟踪算法(Edge-chasing algorithms)，这类算法通过沿着等待图的边界传播一种称为探测消息(如 probe)的特殊信息来检测死锁[6]。当发起者得到一个或多个匹配的探测消息时，它就发现了一个死锁环。典型的边界跟踪算法有 Probe 算法，Misra 算法。该类算法传递的消息量少，伪死锁率低于路径推动算法，但该类算法无法仅通过一次检测就获知死锁环的组成进程，故为了确定解死锁进程需要多次检测。

上述两种算法缺点都很明显，研究人员为优缺互补将两类算法特性合并形成一类新的算法，称为混和算法(Hybrid Algorithms)[1]。混和算法通过只传递本地等待图中与全局死锁相关的部分，与路径推动算法相比信息量大大降低，同时又保证只需一次检测即可发现死锁并获知死锁环相关进程[1]。但混合算法与路径推动算法一样，仍无法克服伪死锁率高的问题。

③扩散计算(Diffusing computation)，在这类算法中，当站点怀疑有死锁发生的时候，站点管理器就通过向与它相关的进程发送查询发起一个扩散进程[6]。发送查询信息，扩散计算就增长；收到回答后，扩散计算就缩减。根据得到的信息，发起者就能检测到死锁的发生。典型的算法有 Chandy、Misra 和 Hass 算法，Hermann 和 Chandy 算法等。与路径推动算法相比，这类算法发起的消息量小，网络负荷小。

④全局状态检测(Global state detection)，这类算法基于 Chandy 和 Lamport 的快照方法，可以不需要暂停当前的计算而通过建立一个一致的全局状态生成一个一致的全局等待图。该类的典型算法有 Bracha 和 Toueg 算法等。由于这类算法引入了系统快照，所以没有与消息延时有关的问题，死锁的发现也更容易[4]。

此四类死锁算法中，路径推动算法和边界追踪算法用于处理资源死锁，而扩散计算和全局状态检测用于处理通讯死锁。由于资源死锁出现较多且研究中较受重视，故本文将重点关注如何降低资源死锁检测算法的伪死锁率。由于混合算法皆具路径推动和边界追踪算法的特性，可视为资源死锁检测算法的代表。故本文选择了混合算法的典型算法 SET 和 MAEDD 作为验证实验对象。下面简单介绍一下两种算法的具体内容。

2.1 SET 算法[1]

该算法通过在收集和传递与全局相关的本地死锁信息(全局死锁候选链)来发现死锁。所谓全局死锁候选链是指如果有一个站点有一个进程记录链表 $S=(T_1,T_2,\cdots,T_n)$，其中 $T_{j+1} \in Block(T_j)$，$1 \leqslant j \leqslant n$，且所有的进程都在同一个站点，那么我们称这个进程记录链表为全局死锁候选链(global candidate)G。其中全局死锁候选链中 T_n 必须被站点外的进程阻塞，同时 T_1 必须阻塞站点外的进程，否则不能称为全局死锁候选链。而一个全局死锁候选链或两个(或两个以上)的全局死锁候选链串连起来称为扩展全局死锁候选链(propagation global candidate)。

全局死锁候选链很可能是一个全局死锁环的一部分。全局死锁候选链是通过如下方法发现的：由死锁检测发起进程沿着本地等待图推进，并按顺序将被阻塞的进程号加入进程记录链表，如遇到某进程未被阻塞则放弃死锁发起检测，否则不断加入被阻塞的进程直到发现某进程被站点外某进程阻塞为止。然后，从发起者沿着本地等待图的反方向推进，同时将在本地等待图中被阻塞进程的进程号加入进程记录链表，操作同前。将正向推进和反向推进获取的进程记录链表合并得到一条全局死锁候选链，并将发现全局死锁候选链合并为一个全局死锁候选链的集合。

死锁检测是在某处理进程等待某资源特定时间后发起。例如，某进程在等待了时间 T 后仍旧没有获得资源，该进程就认为在系统中出现了死锁。于是该进程向站点管理者提出发起死锁检测的请求。站点管理者首先通过本地的等待图来检测本站点内是否存在本地死锁，如果存在，那么就将该本地死锁解除。然后检测是否存在可能成为全局死锁一部分的全局死锁候选链，如果存在将本地的全局死锁候选链(T_1,T_2,

…,T_n)发送给阻塞进程 T_1 的站点。

某站点接收到全局死锁候选链后,首先也获取本站点的全局死锁候选链集合。如果发现本站点的全局死锁候选链集合中有进程的优先级比收到的候选链高,那么丢弃收到的死锁链,否则将本站点的全局候选死锁链和收到的死锁链结合形成扩展全局死锁候选链。如果合并后发现了死锁环,那么找出死锁环中优先级最小的进程,强制该进程放弃已经获得的数据资源;如果没有发现死锁环,则将合并后的扩展全局死锁候选链与死锁检测发起进程类似地传给下个站点。

2.2 MAEDD 算法[5]

MAEDD 算法通过利用移动代理在各站点间收集信息来检测死锁。当某个处理进程等待某个资源一定时间后发起死锁检测。发起者首先检查在自身所处站点中是否存在本地死锁,如存在则先解决本地死锁。如还可能存在全局死锁,发起者则发起一个移动代理。该移动代理中存储着类似等待图的信息 P-List 和 S-List。假设发起者进程的进程号为 P_i,而进程 P_{i+1} 阻塞进程 P_i(根据本地等待图),则将进程 P_{i+1} 的进程号加入 *P-List*,并将它所在的站点号加入 *S-List*。然后移动代理按等待图搜索将阻塞 P_{i+1} 的进程(比如 P_{i+2})再加入 *P-List*。这个过程一直重复直到发现一个本站点中的进程被其它站点的进程(比如 P_N)阻塞。此时将 P_N 所在的站点号加入 *S-List*,将该站点作为目标站点,迁移移动代理。当到达下一个目标后,移动代理进行类似的收集信息,并根据需要更新 *P-List* 和 *S-List*,然后再迁移。当将要加入 *P-List* 中的进程信息中有进程曾在 *P-List* 中出现时,那么表示存在全局死锁。该死锁确定后,将 *P-List* 中的最后一个进程作为死锁解决进程,释放其占有的资源以解除死锁。

为了便于比较,实验时对 MAEDD 算法作了微小改动,使其挑选解除死锁的进程时亦采取类似 SET 算法的选取最低优先级的策略。

3 算法的伪死锁率分析和改进

在设定模拟环境前我们通过一系列实验确定实验参数,使得在该参数下能较为频繁地产生死锁从而高效地验证改进效果。根据结果将模拟实验环境设定如下:系统每个站点上有两个进程;进程的下次请求发起平均延时为2个时间单位;资源平均占有时间为10个时间单位;网络通讯延时为2个时间片。资源占有时间和下次请求发起时间服从 $\alpha=1,\beta=2$(资源平均占有时间或下次请求发起平均延时)的韦伯分布(Weibull Distribution);进程请求的资源号服从平均分布。若进程被阻塞25个时间单位则发起死锁检测。实验中数据资源数为15或20,每次模拟运行5000个时间单位。

3.1 算法中实验关于伪死锁率的发现

首先将 SET 算法和 MAEDD 算法以伪死锁和真死锁比值为指标进行对比实验。实验结果如图1,比较结果显示 MAEDD 算法的伪死锁率相对 SET 算法略低。这是由于死锁检测发起时采用的信息搜索策略不同造成的:以单资源模型为例,在 MAEDD 算法中,死锁检测发起者仅沿着等待图正方向收集信息形成 *P-List*,由于进程一般都在等待另一个进程占有的资源,故每次仅能发现一条 *P-List* 并传递给下一站点。而 SET 算法由于存在沿等待图反向推进的操作,而死锁检测发起者可能阻塞多个进程,故每次可能检测到多个全局死锁候选链。而这些全局死锁候选链极有可能是针对同一死锁环,且都有可能检测到该死锁环,这就导致伪死锁率相对较高。

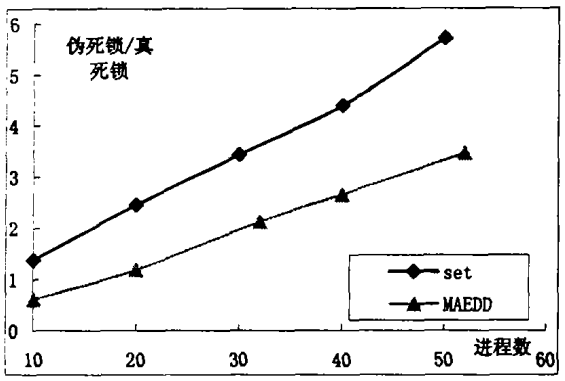

图1 死锁检测实验算法中伪死锁与真死锁比值的比较

同时通过实验和分析,我们认为所有的死锁算法从死锁检测者和死锁发现者的角度可被分为两类:一类是成功的死锁检测都是由死锁环内部的进程发起的,称为环内检测算法(比如边界追踪算法等);另一类是成功的死锁检测可能由死锁环外的进程发起的,称为环无关检测算法(如路径推动算法,混和算法等)。模拟发现,环无关检测算法的伪死锁率要比环内检测算法略高。SET 算法和 MAEDD 算法就属于环无关检测算法,取 MAEDD 为代表进行实验,资源数为20,图中5个点表示从左到右表示进程数分别为10,16,20,26,30。实验最终结果如图2所示,可发现在 MAEDD 算法中成功检测到死锁的死锁检测大多是由死锁环外进程发起的,且其比率随着伪死锁率的增加而快速增加。该现象从概率上亦可加以解释,由于死锁环外进程相对于死锁环内的进程较多,故而死锁检测发起的概率也相对较高。环无关检测算法与环内检测算法相比,死锁环内的进程发起的死锁检测概率相近,而环无关检测算法中死锁环外发起的死锁检测也可能检测到死锁,故针对同一个死锁的死锁检测更多,从而伪死锁率就相对更高。

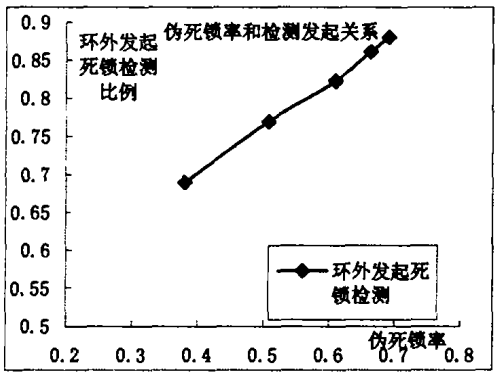

图2 检测发起位置与伪死锁率的关系

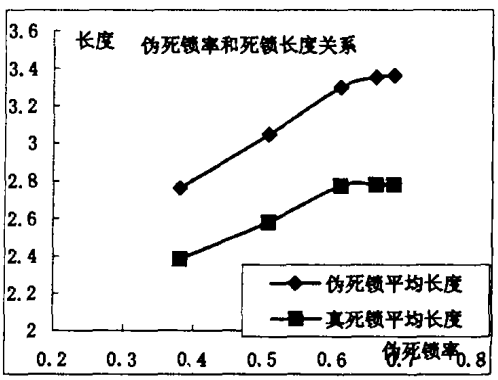

图3 伪死锁率与死锁长度的关系

伪死锁率随着进程的增加而大幅上升除了由于死锁环外发起的死锁检测增多外，另一影响因素为死锁环长度。实验结果显示随着进程的增加，死锁环的长度逐渐增长。以MAEDD为例，设定资源数为15进行实验，结果表示为图3。图中5个点表示从左到右进程数分别为10，16，20，26，30。如图所示随着进程数的增加，死锁的平均长度同步增加(伪死锁率亦同步增加)。图3表明伪死锁平均长度略长于真死锁平均长度，这就意味着越是长的死锁在检测时越可能发生伪死锁。故而进程数的增加导致死锁环长度增长，同时会引起伪死锁率增加。随着进程数到达一定值后，死锁长度开始基本保持不变，而伪死锁率增长幅度同步变小。

综上所述，可以发现混合检测算法中由于大多数是环外检测，如果能减少针对同一个死锁的死锁检测，那么大多数环外检测就能被消除。

3.2 死锁检测算法的改进

3.2.1 *减少冗余检测改进* 在分布式算法中为降低死锁检测时网络中的信息流量，存在通过将针对同一个死锁环的死锁检测数目减少，从而减轻网络的流量的改进方法。易知被抛弃的死锁检测都属于造成伪死锁的死锁检测，故类似思想可用于降低算法的伪死锁率。

为比较两个死锁检测，需要死锁检测发起者和传送者记录一些信息。系统可为每个死锁检测分配一个独一无二的标识符，检测发起站点记录发起检测的标识符和该死锁检测针对的死锁环等信息。而死锁检测消息中则包含标识符和发起者优先级甚至是发起时间。当某死锁检测 C 传递到某站点时，站点管理者通过查看本地记录信息判断是否曾发起针对同一死锁环的死锁检测，若没有则继续向前传递 C；否则，按预设的消息抛弃策略来处理该检测，若符合抛弃策略则将死锁检测 C 抛弃。如此则可使站点管理者抛弃一部分针对同一个死锁环的检测，该思想反映到SET算法和MAEDD算法中就是分别抛弃SET消息和移动代理。但采用无限制的抛弃策略易出现 A 进程抛弃 B 进程发起的检测，B 进程抛弃 A 进程发起的检测的循环抛弃现象。故抛弃策略的制定需要防止类似现象的发生，最为直观的方法就是若进程收到的死锁检测其发起者的优先级低于接收进程则将该检测抛弃，否则继续传递。如此就避免了循环抛弃的问题，当然也可根据具体的需要制定不同的抛弃规则。

由于死锁检测时死锁环尚未被发现，需要其它方法来确定两个检测是否针对同一死锁环。我们采用如下方法：假设两个死锁检测都传递给同一个站点中的同一个节点，即认为两个检测是针对同一个死锁环。通过比较发起者优先级来改进能降低伪死锁率，能同时降低网络负荷。该改进问题在于由于是根据发起进程的优先级来判断消息是否抛弃，故有可能使先发起的检测但由于发起者自身优先级较低从而导致该检测于途中被抛弃，从而直接导致死锁发现的时间延时增加；又由于对同一个死锁环的判断方法不能保证每次判断完全正确，故有可能使针对不同死锁环的死锁检测被当成针对同一死锁环而抛弃，从而也一定程度上延长了死锁发现的时间延时。故本改进会一定程度上延长死锁持续时间。

而作为基于移动代理的MAEDD算法，由于移动代理本身的特殊性可采取相对特殊的算法改进。代理的存在使检测发起者能容易的定位其产生的移动代理目前的位置。如此可为两个针对同一个死锁环的检测的移动代理合并提供条件。当某站点收到一个死锁检测的移动代理时，首先通过查询本地记录判断是否发起过针对同一个死锁环的移动代理，若不存在则继续传递该代理；若存在则设法将两个代理的信息合并，并抛弃其中一个代理。该改进的优点在于不但减少了网络的信息流量，减少冗余的死锁检测，且充分利用已获取的信息。

3.2.2 *本地解死锁改进* 在分布式算法中，若消息(或移动代理)传递的网络延时较小，整个系统中的信息更新速度将较快。而整个系统的整体状态信息若能更快地更新，则可使分布式算法发现死锁更为准确，从而能降低伪死锁率。在死锁检测中有两个阶段需使用网络通讯，一个是死锁检测消息的传递阶段；另一阶段是死锁发现后通知对应进程释放其占有的资源以达到解死锁的目的，即死锁解决阶段。前者必须在站点间传递，无法在该阶段进行改进，故只能从解决死锁时选择进程的环节加以改进，即采用本地解死锁策略。

在死锁检测算法的死锁解决部分，大多采取全局统一的选择标准来确定死锁解决进程，如选择死锁环中最低优先级的进程等。统一解死锁策略有可能使死锁发现进程和被解决进程位于不同的站点上，从而导致站点间通讯的存在。但假如死锁环发现者将自身选定为死锁解决进程而放弃资源，则可忽略通讯时间，即在发现死锁的同时解决死锁。快速的死锁解决能使尚未发现死锁的死锁检测可能因死锁环的断裂而遇到非阻塞进程，从而放弃死锁检测。举例说明该情况：譬如存在死锁环2→1→3→2(假定每个进程一个站点)，有死锁检测在进程3上发现了死锁，而此时还存在一个死锁检测在进程1上，该检测转移到进程3就可检测出死锁。如果采用统一解死锁策略(选最低优先级进程)，设站点间通讯时间为2个时间单位。当进程1收到解死锁命令时，在进程1上的死锁检测已转移到进程2上，并检测到死锁；但假如进程3本地解死锁，将导致进程1上的死锁检测由于死锁环消失而停止检测。其实类似情况颇多，一般认为死锁环越长，出现类似情况的几率越大。故而可采取使检测到死锁的进程放弃自身占有资源从而解锁改进算法。由于死锁解决的消息不需传递，该改进算法也能从一定程度上减少网络流量。同时由于在死锁检测部分未加修改，可保证死锁被检测到的时间延时与原算法基本相同，不会有太大的增加。

但该改进亦存在缺点，即可能发生同一死锁环中多个进程同时检测到死锁而导致同时解锁，该问题出现将导致伪死锁率增加。但由于该问题发生几率较小，基本可以忽略不计。

4 实验分析

我们对MAEDD和SET算法应用两种算法改进思想分别加以改进，并通过仿真实验验证改进的结果。由于MAEDD和SET算法在基本思想上相当类似以及篇幅所限，故文中仅具体说明MAEDD算法的结果，同时简略地说明SET算法的改进效果。

4.1 减少冗余检测的改进结果

改进验证实验中采用最基本的改进思想：由发起站点通过比较检测消息中发起者的优先级来判断是否抛弃该检测。实验结果发现伪死锁率有明显降低。但由于采用比较进程优先级这一抛弃策略，导致某些发起较早的检测被抛弃，从而造成了死锁持续时间延长。在资源数为20的情况下，MAEDD算法伪死锁率结果比较如图4，死锁持续时间结果比较如图5所示。

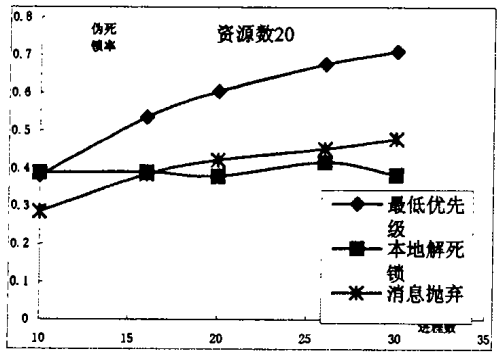

图4 MAEDD 算法改进与原算法的伪死锁率比较

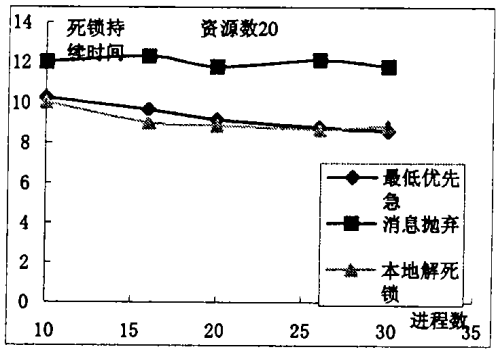

图5 MAEDD 和改进算法的死锁持续时间比较

由图4和图5可知，MAEDD 算法在改进后其伪死锁率降低了10%～20%，且随着进程数增加，伪死锁率降低越多。而死锁持续时间却增加到了12个时间单位，要比原来的死锁持续时间增加了2～3个时间单位。这表明在采用该改进在一定程度上以牺牲死锁持续时间为代价的。当然，在某些对死锁检测的实时性要求不强的系统中该改进是可选方案。假如能够挑选较好的抛弃消息的策略，应该可以使得在死锁持续时间上的增加幅度相当小，从而达到降低伪死锁率的同时保持较低的死锁持续时间的效果。在采用该思想改进 SET 算法后发现伪死锁率也有明显降低，降低幅度也与 MAEDD 近似。

从实验的结果可得出结论：该类改进能有效地降低伪死锁率，并且同时可降低伪死锁率随着进程数增长而增长的幅度。而消息抛弃策略的不同选择，可能造成本地信息处理较为复杂，及死锁持续时间延长的问题。故消息抛弃策略可以根据应用的具体需要进行选择，灵活度较高。

4.2 通过本地解死锁改进的结果

将本地解死锁的改进思想应用到 SET 和 MAEDD 算法后，得到了令人满意的结果。在将 MAEDD 算法改进为本地解死锁后，如果设定资源数为20，改进前后伪死锁率比较如图4所示。如图所示 MAEDD 算法的伪死锁率有较明显的降低，且实验中也发现死锁检测时间并未因此延长(如图5)。这是由于在这个改进中死锁检测环节与原算法没有变化，故死锁检测到的时间延迟将不会改变。

实验中的一种现象值得注意，即在 MAEDD 算法上实现这个改进的效果不仅能较好地降低伪死锁率，而且使其伪死锁率的增长幅度得到了很好的控制(如图4)，而在 SET 算法中虽然较好地降低了伪死锁率，但是在增长幅度控制上效果要稍差。

从实验结果可得出结论：本地解死锁这个改进方法可有效地降低伪死锁率，并对控制某些算法的伪死锁增长幅度有很明显的效果。且能避免死锁检测延时变长而导致算法性能降低的问题。

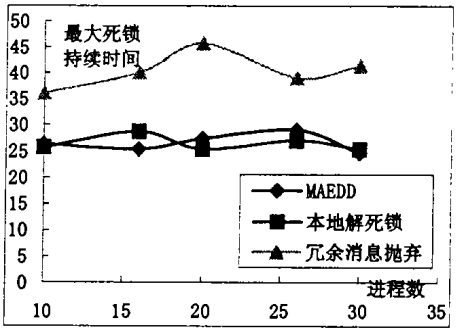

图6 MAEDD 死锁平均最大持续时间

如图4所示，两种改进应用在 MAEDD 算法上效果可以看出有所差异。首先，在进程数较少(意味着站点也较少)的情况下，采用本地解死锁这种改进方法的效果并不是很明显；相对而言采用冗余检测抛弃的效果较好。而当进程数增加到特定程度，采用本地解死锁的效果较优。其次，采用本地解死锁改进后，伪死锁率增长幅度明显变缓，而采用消息抛弃的改进的增长幅度虽然也变缓，但是效果比本地解死锁略差。但本地解死锁只能较明显地降低某些算法的增长幅度，某些算法效果与冗余检测抛弃类似。结果表明两种改进各有特点，可根据不同的需求加以使用。当然如果将两种改进结合起来对原算法进行改进，伪死锁率将更低，但比使用单一改进仅有少许降低，故我们认为意义不大而不再将数据列出。

而作为算法改进，如果影响了算法发现死锁的能力，那么该改进也是不成功的。从理论上分析，由于在冗余检测抛弃策略中抛弃的检测都是将可能检测到伪死锁的检测，同时在本地解死锁中，死锁检测部分没有改变，故两种算法改进应该不会影响算法的死锁发现能力。我们通过测量最大死锁持续时间来检验改进对真死锁检测的影响，结果如图6所示。MAEDD 算法应用时，死锁的最大持续时间是一个有限值，故能发现所有存在的死锁。而采用本地解死锁的思想改进算法后，死锁的最大持续时间基本不变，这不仅证明了本地解死锁能避免死锁持续时间增加，且仍能发现所有死锁；采用冗余消息抛弃的思想改进算法后，虽然死锁最大持续时间增加，但仍能在有限时间内发现死锁，故算法改进对真死锁发现无不良影响。由此可知，本文中提出的两种算法改进能够保持原算法对真死锁检测的能力。

结论 本文首先研究了死锁检测算法中引起伪死锁率偏高的具体原因，并试图通过对现有死锁检测算法的改进降低算法的伪死锁率。研究结果表明，检测算法是否支持死锁环外发起的检测能在很大程度上影响伪死锁率，而在支持环外发起检测的算法中环外发起的死锁检测占所有死锁检测的较大部分，这些检测的存在也在一定程度上造成了网络信息量过大而影响算法性能。本文提出了两种算法改进的思想，首先采用成熟的降低网络信息量的改进方法来减少冗余检测，该改进不但能降低伪死锁率，且能在一定程度上降低伪死锁率的增长幅度。但消息抛弃策略的选择将影响算法性能或造成本地计算负荷一定程度的增加。其次，采用了改进解死锁策略来降低伪死锁率的方法在一定程度上减少整个系统中死锁检测的数量，从而达到降低伪死锁率的目的。该改进也能达到降低伪死锁增幅的目的，但效果因算法而异。

实际上，从算法检测发起后再对其中的冗余检测进行处

理较为消极，若能在死锁检测发起前就积极防止冗余检测的发生(即解决多重启动问题)，将更有效地降低伪死锁率而提高算法性能。所以，我们进一步的研究将重点关注如何解决多重启动问题。

参 考 文 献

1 Choudhary A N. Cost of Distributed Deadlock Detection: A Performance Study. In: Proc. the 6th Intl. Conf. on Data Engineering, 1990. 174～181

2 Rubio J M M, Lipez P, Dutao J. FC3D: Flow Control-based Distributed Deadlock Detection Mechanism for True Fully Adaptive Routing in Wormhole Network. IEEE Trans. on Parallel and Distributed Systems, Aug. 2003, 14(8): 765～779

3 Ekmecic I, Tartalja T, Milutinovic V. EM³: A contribution to taxonomy of heterogeneous computing systems. IEEE Computers, Dec. 1995. 68～70

4 Knapp E. Deadlock Detection in Distributed Databases. ACM Computing Surveys, Dec. 1987, 19(4): 79～100

5 Cao J, Zhou J Y, Zhu W W, Chen D X, Lu J. A Mobile Agent Enabled Approach for Distributed Deadlock Detection. In: Proc. Of GCC 2004, Wu Han, 2004

6 Wu J. Distributed System Design. CRC Press LLC, 1999

7 Chandy K M, Misra J. A distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems. In: Proc. Of ACM SIGACT-SIGOPS Symp. on Principles of Dist. Computer, Ottawa, Canada, Aug. 1982

8 Chandy K M, Misra J, Haas L M. Distributed Deadlock Detection. ACM Trans. on Computer Systems, May 1983, 1: 114～156

9 Singhal M. Deadlock Detection in Distributed Systems. IEEE Computer, 1989, 22(11): 37～48

10 Obermarck R. Distributed Deadlock Detection Algorithm. ACM Trans. On Database Systems, 1982, 7: 187～208

(上接第168页)

外我们也给出一个常用的高阶函数 *filter*。

3.4.1 *高阶函数 map* 这是一个与 *mapTree* 类似的高阶函数，但它作用在列表而不是二叉树上。它将一个作为参数的函数应用于一个列表中的每个元素，从而产生另一个列表。该函数的定义如下：

```
map::(a→b)→[a]→[b]
map f [ ]=[ ]
map f (a:as) = f a:map f as
```

例如：*map* (\x→x+1)[1,2,3]的结果为[2,3,4]。

换句话说，*map* 将一个函数的定义域扩展到相应的列表上。

3.4.2 *高阶函数 filter* *filter* 的功能是将列表中不满足过滤条件的元素过滤掉，这里，过滤条件是一个谓词(即值域是布尔类型的函数)，因此，它也是高阶函数。

```
filter::(a→Bool)→[a]→[a]
filter p[ ]=[ ]
filter p (a:as)| p a=a:filter p as
               | otherwise = filter p as
```

例如，*filter* (\x→x>3)[2,3,4,5]得到[4,5]。

3.4.3 *列表内涵特性* 在数学中，我们已经知道如何利用集合的内涵特性来描述集合的定义。例如我们可以用{$x|x \in N, x>10$}来表示大于10的自然数的集合，而不必逐个列出各个元素。Haskell 语言的列表内涵特性，可以像集合论中通过内涵的方法表示集合那样，让我们通过内涵的方法来表示列表，例如上面的表示大于10的自然数的集合，在 Hascall 中可以用列表[x|x←[0..],x>10]来表示。这里 x←[0..],x>10起到了内涵的作用。利用这种机制可以很方便地在现有列表的基础上定义新的列表。

一个列表内涵通常由三部分组成，分别是：

(1)产生器(generator)，即源列表集，也就是一个元素的选择范围，它是一个已知的列表。

(2)过滤器(filter)，规定了元素的选择条件，即如何从一个已知的列表中选择元素，选择哪些元素。

(3)转换(transformation)，即如何将从源列表集中选择的元素转换为希望生成的新列表的元素，可以理解为新列表中元素的表达形式。

例如，在列表内涵表达式[(x,True)|x←[1 .. 100],even x,x>15]中：x←[1..100]称为产生器，even x,x>15称为过滤器，(x,True)则称为新列表的转换。

借助于列表内涵，我们可以给出 *map* 和 *filter* 的更为简洁的定义如下：

```
map f xs=[f x|x←xs]
filter p xs=[x|x←xs,p x]
```

3.4.4 *树及其搜索* 前面我们只是局限于二叉树来讨论相应的应用，事实上，我们很容易定义一棵普通的树并且实现其上的搜索算法。下面定义的 *Gtree* 就是一棵普通的多态类型树：

```
data Gtree a=Gtree {root_item::a
                   ,children ::[Gtree a]}
```

利用上面定义的高阶函数 *map* 以及列表内涵特性，我们只用下面的两行，就可以给出深度优先搜索函数 *depth_first* 的类型说明及其实现。

```
depth_first::Gtree a→[a]
depth_first(Gtree n cs)
=n:[m|df←map depth_first cs,m←df]
```

比较一下在强制式语言中与此相关的概念和算法的定义，我们不难看出这样的实现是多么的简洁和优美！

结束语 本文简要阐述了 Haskell 语言及其高阶特性，通过一些简单的例子揭示了如何利用其高阶特性来编写程序模版。

综合上面的论述可以看出，使用高阶函数有以下优点：(1)定义简洁；(2)定义容易理解；(3)定义容易修改；(4)代码可重用性强。

本文中的程序，均已在 Haskell 的解释器 Hugs98(versionNov2003)和其编译器 GHC6.2.1上调试编译通过，真心希望能有更多的国内同行来使用该语言，编写出简洁漂亮的程序。

参 考 文 献

1 Pang J, Callaghan P, Luo Z. An approach to verification of domain properties based on LF. TYPES 2002 Workshop. Netherlands

2 Callaghan P C, Luo Z, Pang J. Object languages in a type-theoretic meta-framework. Workshop of Proof Transformation and Presentation and Proof Complexities (PTP'01), Italia, 2001

3 Callaghan P C. Functional Programming. Unpublished lecture notes. UK, 2003

4 Thompson S. Haskell: The Craft of Functional Programming, Second Edition. Addison-Wesley, 1999. ISBN 0-201-34275-8

5 Bird R. Introduction to Functional Programming using Haskell, 2nd edition. Prentice Hall Press, 1998, ISBN: 0-13-484346-0

6 Jones S P. Haskell 98 Language and Libraries. Cambridge University Press, 2003, Hardback, ISBN: 0521826144

7 Structuring Depth First Search Algorithms in Haskell. In: King D, Launchbury J, eds. Proc. ACM Principles of Programming Languages, San Francisco, 1995

8 Haskell 98 report. http://www.haskell.org