第3期 电 子 学 报 Vol. 53 No.3
2025年3月 ACTA ELECTRONICA SINICA Mar. 2025

分布存储并行程序变异算子有效性评估

田 甜[1],王苗苗[1],李成龙[1],巩敦卫[2*]
(1. 山东建筑大学计算机科学与技术学院,山东济南 250101;2. 青岛科技大学自动化与电子工程学院,山东青岛 266061)

摘 要: 变异测试通过注入变异算子生成变异体模拟软件中的潜在缺陷,是提高软件质量的关键技术. 大量变异体及其执行成本制约着变异测试的研究进展和在工业领域的应用. 选择有效的变异算子是减少变异体数量的主要途径. 针对分布存储并行程序,本文提出变异算子有效性评价准则. 从顽固变异体、崩溃变异体和等价变异体3个方面对变异体进行分类;基于不同变异体对测试数据质量的影响,定义变异算子有效性评价准则,分析不同变异算子的有效性. 实验结果表明,使用本文提出的评价准则能够选择合理的变异算子,基于这些变异算子能够生成更多的有效变异体和尽可能少的无效变异体,在保持变异测试有效性的前提下,平均减少了22.61%的变异体,提高了变异测试效率.

关键词: 分布存储;并行程序;变异测试;变异算子;变异体
基金项目: 国家自然科学基金(No.62102235);山东省自然科学基金(No.ZR2020MF084)
中图分类号: TP311 **文献标识码:** A **文章编号:** 0372-2112(2025)03-0864-14
电子学报URL:http://www.ejournal.org.cn **DOI:**10.12263/DZXB.20240741

Effectiveness Evaluation of Mutation Operators for Distributed Memory Parallel Programs

TIAN Tian[1], WANG Miao-miao[1], LI Cheng-long[1], GONG Dun-wei[2*]
(1. *School of Computer Science and Technology*, *Shandong Jianzhu University*, *Jinan*, *Shandong* 250101, *China*;
2. *College of Automation and Electronic Engineering*, *Qingdao University of Science and Technology*, *Qingdao*, *Shandong* 266061, *China*)

Abstract: Mutation testing is a pivotal technology for enhancing software quality by injecting mutation operators to generate mutant programs that mimic potential defects in software. The substantial number of mutants and the associated execution costs, however, limit the advancement and industrial application of mutation testing. The selection of effective mutation operators is a primary strategy for reducing the volume of mutants. Addressing distributed memory parallel programs, this paper introduces an evaluation criterion for the effectiveness of mutation operators. Mutants are categorized into three types: stubborn mutants, crash mutants, and equivalent mutants. Drawing on the influence of various mutants on the quality of test data, we establish a criterion for evaluating the effectiveness of mutation operators and analyze the performance of different mutation operators. Experimental results demonstrate that the proposed evaluation criterion enables the selection of suitable mutation operators, leading to the generation of a higher number of valid mutants and a minimized number of invalid mutants. Consequently, while preserving the effectiveness of mutation testing, the average number of mutants is decreased by 22.61%, thereby enhancing the efficiency of mutation testing.

Key words: distributed memory; parallel programs; mutation testing; mutation operators; mutants
Foundation Item(s): National Natural Science Foundation of China (No.62102235); Natural Science Foundation of Shandong Province (No.ZR2020MF084)

1 引言

分布存储并行程序包含两个或多个程序流,每个流程在执行期间具有独立的内存. 流程之间通过消息传输机制实现交互. 基于多个流程的并行执行和通信具有较高的求解精度和速度,被广泛用于解决各个领域的复杂难题[1,2],特别是需要处理大规模数据集的科

收稿日期:2024-08-13;修回日期:2024-12-27;责任编辑:孙瑶
*通讯作者:巩敦卫

学研究、生物信息学、气候模拟和实时数据处理等应用中. 提高分布存储并行程序的可靠性至关重要.

变异测试是一种面向缺陷的测试技术. 基于预定义的规则修改原程序以模拟软件中的真实缺陷,预定义的规则称为变异算子,被注入缺陷的程序称为变异体[3]. 作为变异测试的基础,变异算子通常是根据目标语言的使用经验和最常见的故障而设计的[4-6]. 使用相同测试数据执行变异体和原程序,如果输出结果不同,称该变异体被杀死[7]. 研究表明,相比其他结构覆盖测试方法,由变异算子生成的变异体与真实故障耦合性更强[8]. 使用杀死变异体的测试数据构成高充分性测试套件,能提高程序缺陷检测效率. 然而,大量变异体及其执行导致了高昂的测试成本,如何降低测试代价成为变异测试的研究重点.

通过识别和选择有效的变异算子,可以生成具有较高缺陷检测能力的变异体,同时避免生成低效或冗余的变异体,从而优化测试过程,增强测试结果的可靠性[9,10]. 变异算子的评估结果还可以帮助开发人员制定测试策略,比如,为了检测某一缺陷,选择更易导致该类缺陷发生的变异算子,从而有针对性地设计测试用例. 已有研究通常使用变异体数量和测试数据缺陷检测能力选择变异算子[11]. 然而,上述两个因素忽略了变异体的多样性,并且崩溃变异体的存在可能导致测试数据缺陷检测能力不准确. 例如,一个变异算子可能通过引入内存泄漏导致程序崩溃,这种情况下,任意测试用例都能够检测到相关变异体. 此时,这些测试用例的高缺陷检测能力是由变异算子引入的特定错误导致,并不能说明测试用例本身的优秀[12]. 在分布式环境中,由于资源的并发访问和同步管理的复杂性,死锁的出现概率较大,这导致程序崩溃情况尤为频繁[13]. 为了评价分布存储并行程序变异算子,需要考虑变异体的不同类型. 顽固变异体通常代表程序中的一些特殊或边缘情况[14],杀死顽固变异体的测试数据的缺陷检测率更高. 等价变异体与被测程序语义相同. 部分变异算子引入的缺陷可能导致程序崩溃[12],即对应的变异体能被任意测试数据检测到,也就是说,等价变异体和崩溃变异体无法改进测试数据集的质量. 因此,有效的变异算子应能够生成尽可能多的顽固变异体,尽可能少的等价和崩溃变异体.

鉴于此,本文综合考虑三类变异体的影响,研究分布存储并行程序变异算子的有效性,选择合理的变异算子生成变异体,提高分布存储并行程序变异测试效率. 首先,分析顽固变异体、崩溃变异体和等价变异体对改进测试数据集质量的作用,将变异体分类为有效变异体和无效变异体;然后,基于变异算子生成不同变异体的概率构建变异算子的有效性评价准则;最后,选择多个典型并行程序作为被测对象,注入变异算子,根据变异体的运行情况和输出结果,分析不同变异算子生成的变异体类型,使用不同指标评估变异算子,并验证本文提出的评价准则的有效性.

2　相关工作

变异测试的有效性在于其能够模拟程序员在编码过程中可能出现的错误,从而帮助开发者构建更可靠的软件系统. 评估变异算子对改进变异测试过程至关重要. 下面从变异算子设计、选择变异和约减变异体3个方面阐述相关工作.

通过深入分析程序潜在的缺陷,研究者们设计了一系列针对不同程序的变异算子. 针对Java程序的测试,Ghosh[15]定义了两个与移除同步关键字相关的变异算子;Delamaro等人[16]提出了11个能够反映Java语言并发缺陷的变异算子集合,包括删除同步属性、删除同步方法调用和替换同步对象等;Bradbury等人[17]提出了更多的并发变异算子,包括修改并行方法的参数和发生、修改关键字和交换并行对象等,并与并发缺陷模式对应;Wu等人[18]针对Java 8的并发特性设计了新的变异算子集合.

随着变异测试的广泛应用,学者们也尝试将基于变异的测试技术应用到其他并行程序中. Jagannath等人[19]提出了用于测试Actor程序的12个变异算子,涵盖消息、消息约束和线程创建3方面;Moradi moghadam等人[20]提出了针对Akka actor并发模型的变异测试框架μAkka,通过分析真实世界中的Akka actor缺陷,设计了32个专门的变异算子;Cañizares等人[21]设计了针对仿真云和高性能计算环境的变异算子,用于复现分布式系统的通信和死锁错误;Estero-Botaro等人[22]设计了与并发控制流相关的变异算子,主要针对WS-BPEL业务流程;MPI(Message Passing Interface)是常用的分布存储并行程序开发消息传递环境[23],Silva等人[24]分别针对集合通信、点到点通信和全部MPI函数,提出了26个针对MPI函数的变异算子. 本文基于MPI通信环境及其变异算子[24],研究分布存储并行程序变异算子的有效性.

选择变异通过选择部分变异算子减少变异体数[25]. Mathur[26]首次提出在变异测试工具Mothra中排除ASR(Array reference for Scalar variable Replacement)和SCR(Scalar replacement for Conditional Reference)算子,以此降低变异体的数量;Offutt等人[27]通过忽略ASR和SVR(Scalar Variable Replacement)算子,进一步提出了一种成本更低的“2-Selective”变异测试方法;在“2-Selective”的基础上,Offutt等人又开发了两种新的选择策略,即“4-Selective Mutation”和“6-Selective Mutation”策略. 特别是“6-Selective”策略,在平均保持

88.71%测试效率的同时，能够减少大约60%的变异体数量. Ka-zerouni等使用统计方法，选择了RemoveConditionals和AOD（Arithmetic Operator Deletion）作为关键变异算子，降低了约2~3倍的变异分析成本[28]. Zhang等人[29]提出针对深度学习系统的变异算子约减方法，在16个变异算子中选择DR（Data Repetition）、LE（Label Error）和LR（Layer Removal）等11个变异算子. Kim等人[30]提出基于学习的变异体减少技术，通过为特定目标程序学习一个特定的变异模型，使用细粒度变异算子生成少量代表性变异体，从而预测测试套件的变异得分. 上述工作在维持测试数据有效性的情况下能显著提高变异测试的效率，然而，由于无法捕捉消息不匹配、死锁等并行程序的典型缺陷，所提方法不适用于并行程序.

高阶变异和使用机器学习技术生成高质量变异体是约减变异体的有效途径. 宋利等人[31]利用SOM（Self-Organizing Map）神经网络技术对二阶变异体分类，约减冗余变异体；王子健[32]提出了一种多目标差分进化算法，识别更多具有较强包含性的二阶变异体；Fan等人[33]使用SGS（Statement Granularity Sampling）语句粒度采样约减高阶变异体；Ojdanic等人[34]探索了软件进化过程中提交相关变异体的特性，提出了一种基于观察切片的高阶变异体方法来识别相关变异体；Wang等人[35]通过分析原程序、测试数据和变异体的静态特征，来预测变异体缺陷检测能力，进而生成高质量的变异体子集；针对冗余变异体，Sun等人[36]利用与原程序执行状态相似的变异体，提出了一种基于选择符号执行的冗余变异体识别方法；Garg等人[37]基于变异体代码上下文，提出了基于机器学习的变异体静态选择策略；Tian等人[38]预测需要变异的语句和构建代表真实缺陷的高质量变异体. 上述方法通过约减无效的冗余变异体或选择高质量的变异体优化测试过程，显著提高了变异测试效率.

综上所述，选择变异算子子集或变异体子集是约减变异体的主要途径. 然而，已有方法缺乏对分布存储并行程序的关注，大多数方法仍然以生成所有可能的变异体为前提. 为识别有效的变异算子，本文基于不同变异体的检测难度，提出变异算子的评价准则，选择部分变异算子生成有效的变异体，提高分布存储并行程序变异测试效率.

3 研究动机

变异测试通过判断变异体能否被杀死来选取有效的测试数据，变异体的有效性对测试数据集的质量有重要影响. 由于并行和通信等特点，对分布存储并行程序实施变异更容易产生死锁、消息不匹配等崩溃变异体，这类变异体能够被任意测试数据检测到，对改进测试数据集的质量没有积极作用[12]. 因此，有效的变异算子应该生成尽可能少的崩溃变异体.

分布存储并行程序P有n个流程$P=\{P_1,P_2,\cdots,P_n\}$，其中，P_i表示程序的第i个流程. 这些流程并行执行，通过消息发送和接收函数实现交互. 如算法1所示，该程序计算正整数数组的最小值并根据最小值是否小于25分别输出“A”或“B”，其包含5个流程P_1、P_2、P_3、P_4和P_5，每个流程代码的语句编号标注在程序代码左侧. 其中，P_1将数组中的数据分组发送给P_2、P_3和P_4，P_2、P_3和P_4分别处理求局部最小值的任务，并将结果发送给P_1，最终由P_1计算全部数据中的最小值，并发送给P_5，P_5判断最小值是否满足条件，最终输出“A”或“B”. 假设变异算子MO_1将阻塞模式修改为非阻塞模式，MO_2删除发送函数，MO_3删除接收函数. 选择P_1的发送函数（语句14）分别注入变异算子MO_1和MO_2，P_1的接收函数（语句5）注入变异算子MO_3，生成变异体M_1、M_2和M_3，语句14对应的接收操作为P_5的语句2，语句5对应的发送操作为P_3的语句8. 通过执行原程序P和变异体M_1、M_2和M_3，比较最终输出结果是否相同来判断变异体是否被杀死.

算法1 分布存储并行程序

输入：数组buf0[]

输出：“A”或“B”

```
int main(){
if(id==1){//P1
1. Send(buf0, 2);
2. Send(buf0+3, 3);
3. Send(buf0+6,4);
4. Recv(buf1[0],2);
5. Recv(buf1[1],3);
6. Recv(buf1[2], 4);
7. global_min=buf1[0];
8. if (buf1[0]<buf1[1])
9. global_min=buf1[0];
10. else
11. global_min=buf1[1];
12. if(buf1[2]<global_min)
13. global_min=buf1[2];
14. Send(global_min,5);}
if(id==2||id==3||id==4){//P2~P4
1. Recv(t,1);
2. int min;
3. if(t[1]<t[0])
4. min=t[1];
5. else min=t[0];
6. if(t[2]<min)
7. min=t[2];
8. Send(min,1);}
if(id==5){//P5
1. int m;
2. Recv(m,1);
3. if(m<25){
4. print(A);}
5. else print(B);}}
```

分析M_1、M_2和M_3的执行情况. MO_1将阻塞发送模式修改为非阻塞发送模式，即发送操作不需要消息发送完成就立即返回，M_1对应变异操作不会改变P_5语句2的阻塞接收，最终不会影响P_5中语句5的输出，即在任意测试数据下，M_1和原程序P输出相同；M_2对应的变异操作为MO_2删除发送函数. 删除P_1的语句14后，M_2不再向P_5发送数据，P_5在接收语句2持续等待P_1发送

消息,最终导致M_2发生死锁,任意测试数据都能检测到M_2的死锁状态. M_3对应的变异操作为MO_3删除接收函数,即P_1不再接收P_3发送的局部最小值,导致接收变量buf1[1]的值未被定义,为默认值−858 993 460. P_1将默认值发送到P_5,由于默认值小于25,M_3的输出为"A". 因此,只要测试数据最小值小于25,M_3和原程序P输出相同,都为"A". 为了检测变异体M_3,应设计测试用例使得最小值大于等于25. 相比M_1和M_2,M_3的检测难度相对较大.

对于示例程序1,M_1、M_2和M_3分别代表等价变异体、崩溃变异体和顽固变异体. 假设测试数据集$T=\{t_1,t_2\}$由t_1和t_2两条测试数据组成,其中$t_1=\{83,31,92,91,81,20,83,85,8\}$,$t_2=\{43,75,9,89,97,61,2,53,1\}$. 使用$T$分别执行$M_1$、$M_2$和$M_3$,$M_1$计算得到的最小值分别为8和1,均小于25,输出"A",和原程序P输出相同;执行M_2时,P_5在语句2持续等待,程序死锁;M_3的输出为"A",与原程序P相同. 考虑未被杀死的变异体M_1和M_3. 因为M_1和原程序等价,所以任何测试数据都无法杀死它;为了检测变异体M_3,设计新的测试数据$t_3=\{49,91,35,60,46,53,42,56,80\}$,其最小值为35,原程序$P$输出"B",与$M_3$的输出"A"不同,$M_3$被$t_3$杀死. 因此,通过设计杀死顽固变异体$M_3$的测试数据能够改进测试数据集的质量,进而可以检测类似M_3的真实程序故障;因为M_1和原程序等价,M_2能被任意测试数据检测到,故M_1和M_2不能改进测试数据集的质量.

综上所述,针对同一测试对象,不同变异算子生成的变异体不同,对提高测试数据集质量的作用不同. 因此,在评估变异算子时,需要分析变异体的不同类型,生成顽固变异体的变异算子有效性更高,生成崩溃和等价变异体的算子有效性更低. 考虑不同类型变异体对测试数据集质量的不同影响,研究变异算子的有效性,基于此生成有效的变异体.

4　变异算子评估

4.1　MPI变异算子

分布存储并行程序通过流程之间的消息传递完成协同工作. MPI作为分布存储并行程序的一种重要实现方式,提供了一套跨平台、标准化的消息传递接口. 本文以MPI程序为被测对象,研究面向并行控制和消息传递函数的变异算子[24],如表1所示. 分析这些变异算子的实施对象和变异规则能够发现:

(1) 有些变异算子使用频率低,例如ReplSendRecv、DelDerivDType、DelDetach、ReplComm和ReplStart. Detach函数与动态内存分配和释放相关,但MPI库本身提供了必要的缓冲区管理支持;SendRecv、Derive和Start函数的使用涉及更复杂的编程模式,Sendrecv同时发送和接收消息,在环形或网格通信模式中很有用,但在点对点通信中使用频率低;Derive定义的派生数据类型函数允许使用自定义数据类型进行通信,而程序大多使用基本数据类型进行通信;Start函数通常与复杂的非阻塞通信模式相关;ReplComm涉及替换MPI通信器,由于通信器的创建和管理比较复杂,大多数程序在整个运行期间使用固定的通信器. 因此,这些函数在变异实施对象中出现的频率较低,对应变异算子的使用频率也较低.

(2) 有些变异算子生成崩溃变异体,例如ReplCall替换当前的MPI函数. 假设使用Recv、Gather等替换Send函数,将导致发送函数和接收函数不匹配,引发通信错误,使得任意测试数据都能检测到这些变异体.

(3) 有些变异算子用于非确定通信,例如ReplWait、ReplSource和ReplTag,它们的实施对象分别为用于等待非阻塞请求完成的Wait函数、接收函数中的不确定消息来源和标志. 本文主要研究确定通信,被测对象中不存在非阻塞通信函数和不确定接收函数.

综上,在进行变异算子评估时,排除上述3类共9个变异算子. 将表1中2~18行的剩余17个变异算子作为进一步的研究对象. 此外,有些变异算子生成重复变异体,例如ReplCall和DelCall. 变异算子DelCall对每个MPI函数执行删除操作,而DelBarrier、DelSend、DelRecv和DelFinTask删除Barrier、Send等函数生成相应的变异体,与Delcall生成的变异体存在冗余. 因此,修改DelCall变异算子的变异规则以排除重复变异体. 有些变异算子生成大量变异体,但其中一些变异体是等价的或崩溃的,如ReplArg,该变异算子使用所有可能的变量和参数替换每个MPI函数的变量和参数,在确定执行环境中,对发送和接收函数的消息源和标志进行修改时,若替换的参数值相等会生成等价变异体,不相等则程序崩溃. 因此,修改ReplArg变异算子的变异规则以排除生成的等价和崩溃变异体. 具体算子和规则描述如表1所示.

4.2　基于不同变异体的变异算子评估准则

不同类型的变异体对改进测试数据集质量的贡献不同. 通过分析顽固变异体、崩溃变异体和等价变异体能否提高测试数据集的质量,将这些变异体分类为有效变异体与无效变异体.

顽固变异体指由于测试数据不足而难以杀死的变异体,是提高缺陷检测率的重要因素. 已有研究表明,顽固变异体通常对应程序的一些特殊情况,在测试时经常会发现缺陷. 为了检测顽固变异体,通常需要设计特定的测试数据[14],即杀死顽固变异体的测试数据更容易检测到程序缺陷,从而提高测试数据集的质量. 因此,顽固变异体属于有效变异体.

表1 MPI变异算子[24]

变异算子	描述
ReplRoot	替换Root参数
ReplReduce	替换规约函数,例如,将Reduce替换为Allreduce和Reduce_scatter函数
ReplGather	Gather和Allgather函数的替换
DelBarrier	删除Barrier同步函数
MoveColletiveUpDown	上下移动集合函数
ReplModelSend	发送函数阻塞与非阻塞调用模式的替换
ReplModeSend	发送模式的替换
DelSend	删除发送调用
ReplProbe	使用Probe函数替换接收函数
ReplModelRecv	接收函数阻塞与非阻塞调用模式的替换
DelRecv	删除接收调用
ReplFin	异常终止与正常终止函数的替换
DelFinTask	删除任务终止函数
ChanArg	每个函数中同类型参数的替换
DelCall	删除函数调用,排除其他变异算子已执行的删除操作
InsUnaArg	数值型参数中插入一元运算符
ReplArg	使用已定义的同类型参数替换每个MPI函数的参数,不包括发送和接收函数的消息源和标志参数
ReplSendRecv	替换为SendRecv_replace或其他发送、接收函数
DelDerivDType	删除derived数据类型函数
DelDetach	删除detach函数
ReplComm	替换通信器
ReplStart	Start函数和StartAll函数的替换
ReplWait	替换等待函数,例如,将Wait函数替换为Waitall、Waitany和Waitsome函数
ReplCall	替换为其他MPI函数
ReplSource	替换ANY_SOURCE为其他源
ReplTag	替换ANY_TAG为其他标志

测试数据集中的任意一个测试数据都可以检测到崩溃变异体,所以崩溃变异体不是测试数据发现的[12],对改进测试数据集质量没有积极作用.因此,崩溃变异体属于无效变异体.

等价变异体在功能上和原程序相同,任何测试数据都无法检测到[8].所以等价变异体无法被杀死,即无法评价对应的测试数据集的质量.因此,等价变异体属于无效变异体.

综上分析可知,有效变异体有利于提高测试数据集的质量,而无效变异体往往和测试数据无关,故无法提高测试数据集的质量.顽固变异体属于有效变异体,崩溃变异体和等价变异体属于无效变异体.为提高变异测试的有效性,应选择生成有效变异体占比越高,无效变异体占比越低的变异算子.基于此,给出变异算子有效性评价准则:

(1)变异算子生成顽固变异体的概率越高,有效性越高.变异算子生成的变异体中顽固变异体占比越高,该变异算子的有效性越高.

(2)变异算子生成崩溃变异体的概率越高,有效性越低.变异算子生成的变异体中崩溃变异体占比越高,该变异算子的有效性越低.

(3)变异算子生成等价变异体的概率越高,有效性越低.变异算子生成的变异体中等价变异体占比越高,该变异算子的有效性越低.

基于上述准则,变异算子评估流程分为变异体生成和变异算子评估两个阶段,如图1所示.在变异体生成阶段,向被测程序注入n个变异算子,生成对应的n个变异体集;在变异算子评估阶段,使用相同的测试数据集执行原程序和变异体,根据执行情况和输出结果对变异体分类,根据变异体集中不同变异体的生成概率评估变异算子.

5 实验与结果分析

在本节中,通过对15个不同规模的被测程序的实验分析,给出变异算子的有效性排序,验证本文提出的变异算子评价准则的有效性.首先提出实验研究问题,然后详细介绍了实验设计,包括目标程序、评价指标和具体的实验步骤,最后对实验结果进行详尽分析.

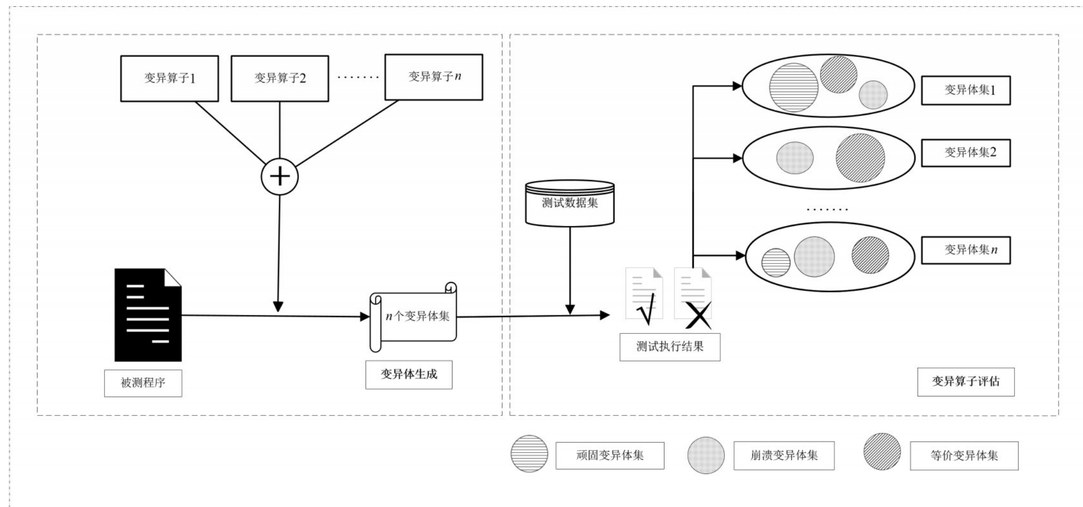

图1　变异算子评估流程

5.1　研究问题

(1)哪些变异算子对分布存储并行程序更有效?

(2)本文提出的准则是否能选择有效的变异算子?

(3)分布存储并行程序选择变异的有效性如何?

(4)影响顽固变异体、崩溃变异体和等价变异体生成的因素有哪些?

5.2　实验设计

选择15个使用扩展MPI标准[39]的C语言程序进行实验研究,它们分别来源于github和已有文献[40~46].对上述程序进行调整,包括修改不确定通信,添加必要的自定义函数,引入复杂的通信关系等,使用的流程数、代码行数、MPI函数个数和变异体数如表2所示,此外,IS在已有程序的基础上增加了新的功能.

表2　被测程序

程序名称	流程数	代码行数	MPI函数个数	变异体数
Vector	4	55	11	63
Prime	5	79	10	86
PI[40]	6	93	14	302
DIT[41]	5	86	14	172
Factorial[42]	4	94	18	333
Gather	4	96	8	82
Array	4	99	8	174
Integrate[43]	4	105	9	263
Graph	5	116	16	312
Mpi_mm	4	128	19	636
Mergesort	4	146	8	79
Pearson	4	262	14	546
Crystal[44]	9	930	36	794
Convex[45]	4	592	34	709
IS[46]	8	1 295	49	1 419

5.2.1　评价指标

采用变异体的杀死概率、变异算子顽固率、变异算子崩溃率和变异算子等价率评估变异算子有效性,有效变异体占比和变异得分验证所提准则的有效性.

变异体的杀死概率(Probability of a Mutant being Killed,PMK)用于衡量生成的变异体的顽固性,如式(1)所示,其中,$|TS_k|$表示杀死变异体的测试数据数,$|TS|$是测试数据集的大小,PMK $\in[0,1]$. PMK值越低,表明该变异体实际被杀死概率越小,即变异体越顽固;PMK值越高,该变异体越容易被杀死.

$$\text{PMK}=\frac{|\text{TS}_k|}{|\text{TS}|}\times 100\% \tag{1}$$

变异算子顽固率(Stubbornness Rate of Mutation Operator,MOSR)用于衡量变异算子生成的顽固变异体的比例,如式(2)所示,其中,$|\text{SM}|$表示产生的顽固变异体的数量,$|M|$是生成的变异体总数,MOSR $\in[0,1]$. MOSR值越高,表明相应的变异算子生成的难杀死变异体越多,变异算子的有效性越高;MOSR值越低,生成的难杀死变异体越少,变异算子有效性越低.

$$\text{MOSR}=\frac{|\text{SM}|}{|M|}\times 100\% \tag{2}$$

变异算子崩溃率(Crash Rate of Mutation Operator,MOCR)用于衡量变异算子生成的崩溃变异体的比例,如式(3)所示,其中,$|\text{CM}|$表示产生的崩溃变异体的数量,MOCR $\in[0,1]$. MOCR值越高,表明相应的变异算子生成的崩溃变异体越多,变异算子的有效性越低;MOCR值越低,变异算子有效性越高.

$$\text{MOCR}=\frac{|\text{CM}|}{|M|}\times 100\% \tag{3}$$

变异算子等价率(Equivalence Rate of Mutation Op-

erator,MOER)用于衡量变异算子生成的等价变异体的比例,如式(4)所示,其中,|EM|表示产生的等价变异体的数量,MOER ∈[0, 1]. MOER值越高,表明相应的变异算子生成的等价变异体越多,变异算子的有效性越低;MOER值越低,变异算子有效性越高.

$$\mathrm{MOER}=\frac{|\mathrm{EM}|}{|M|}\times 100\% \tag{4}$$

有效变异体占比(Proportion of Effective Mutants,PEM)用于衡量有效的变异算子生成的变异体数在所有变异算子生成的变异体数中的占比,如式(5)所示,其中,M_I表示使用第I类变异算子集中的所有变异体生成的变异体数,$|M_{\text{total}}|$表示注入所有变异算子生成的变异体总数,PEM ∈(0, 1). PEM值越小,表明使用第I类变异算子集生成的变异体数越少,减少的变异体数越多;PEM值越大,使用第I类变异算子集约减的变异体数越少.

$$\mathrm{PEM}=\frac{|M_I|}{|M_{\text{toatl}}|}\times 100\% \tag{5}$$

变异得分(Mutation Score, MS)用于度量测试数据集缺陷检测能力,如式(6)所示,其中,$|M_K|$表示被杀死的变异体数,$|M_E|$为生成的变异体中等价变异体的数量,且$|M_K|<|M_{\text{total}}|-|M_E|$,MS ∈[0, 1]. MS越高,说明测试数据集能发现更多缺陷,缺陷检测能力越强;MS越低,测试数据集缺陷检测能力越弱.

$$\mathrm{MS}=\frac{|M_K|}{|M_{\text{total}}|-|M_E|}\times 100\% \tag{6}$$

5.2.2 实验步骤

为回答问题(1),设计实现以下两部分的实验.

首先,生成和执行变异体,判断变异体类型. 第一步,针对每一个MPI函数,注入变异算子,生成变异体集M_o. 第二步,生成满足语句覆盖的测试数据集T,测试数据集规模为5 000. 第三步,使用T中的测试数据执行变异体和原程序. 第四步,分析程序的输出结果,确定变异体的类型. 任一测试下变异体出现运行时崩溃、死锁等行为是崩溃变异体;部分测试下变异体被杀死,部分测试下变异体存活,计算变异体的PMK,根据已有研究和经验设置阈值为0.5[47],PMK小于0.5的变异体为顽固变异体;若执行完所有测试数据后,变异体仍然存活,但是程序中间状态发生改变且状态改变能够传播到程序输出,记为顽固变异体;没有测试数据改变程序中间状态或状态改变不能传播到输出的变异体为等价变异体,基于程序运行和人工分析判定等价变异体. 第五步,统计每个变异算子生成的初始顽固变异体集M_S、杀死变异体集M_K、崩溃变异体集M_C和等价变异体集M_E,其中,M_K不包含M_C. 最后,计算M_K中每个变异体的PMK,将难杀死的变异体添加到M_S中得到最终的顽固变异体集M_S'.

然后,计算指标值,对变异算子排序. 第一步,统计各类变异算子生成的顽固变异体、崩溃变异体和等价变异体数以及变异算子生成的变异体总数,分别计算每类变异算子的有效性评价指标MOSR、MOCR、MOER. 第二步,针对生成顽固变异体的变异算子,计算其生成的变异体的平均PMK,对生成顽固变异体的变异算子进行排序. 第三步,对其他变异算子进行排序,得到最终的变异算子排序表.

为验证问题(2),统计使用A类变异算子生成的变异体数$|M_A|$和所有变异算子生成的变异体总数$|M_o|$,计算A类变异算子生成变异体数在所有变异体数中的占比PEM. 以A类变异算子生成的变异体集为目标,生成杀死该变异体集的测试数据集$|T_A|$,使用$|T_A|$执行原程序和原始变异体集M_o,计算变异得分MS,以验证所提准则的有效性.

为回答问题(3),使用针对Java程序的基于类别的选择变异方法[9]. 选择变异方法选择一组满足变异得分要求的变异算子集作为充分变异算子集,通常要求平均变异得分大于99%. 为了选择合适的变异算子集,按照变异实施对象将MPI变异算子分为3类:参数替换算子(Argument Replacement,AR)、集合通信函数修改算子(Collective communication function Modification,CM)和点对点通信函数修改算子(Point-to-point communication function Modification,PM). AR算子使用其他的合法参数替换程序中的每个参数. 表1中,ReplRoot、ChanArg、InsUnaArg和ReplArg变异算子属于AR. CM算子使用其他形式的集合函数替换现有的集合函数,包括ReplReduce、ReplGather、DelBarrier、MoveColletive-UpDown、DelCall、ReplFin和DelFinTask. PM算子将点对点通信函数修改为其他形式,ReplModelSend、ReplModeSend、DelSend、ReplProbe、ReplModelRecv和DelRecv属于这类算子. 使用每一类变异算子实施变异,计算生成的变异体占比PEM和相应测试数据集的变异得分MS,验证选择变异方法的有效性.

针对研究问题(4),结合程序特点和PIE(Propagation,Infection,and Execution)理论[48],从并行函数所在的位置、变异算子的特点和程序依赖关系,分析影响顽固变异体、崩溃变异体和等价变异体生成的因素.

5.3 实验结果与分析

5.3.1 基于所提准则的变异算子评估

分析不同变异算子生成变异体的数量. 图2显示了15个程序中每个变异算子生成的变异体总数,15个程序总共生成5 970个变异体. 其中,InsUnaArg生成1 083个变异体,ReplArg生成3 433个变异体,分别占总变异体的18.14%和57.50%,这是因为这两个变异算子

需要对每个并行函数的多个可能变量进行变异,生成的变异体相对较多. 实验结果表明,不同变异算子生成的变异体数量差异明显. 然而,仅根据变异体的数量来评估变异算子是不够全面的,例如,变异算子 ChanArg 生成了532个变异体,但可能都是无效的等价变异体或崩溃变异体,即ChanArg生成的变异体数虽然多,但变异体的质量可能并不好. 因此,变异算子是否有效,关键是要评估其生成的变异体的质量.

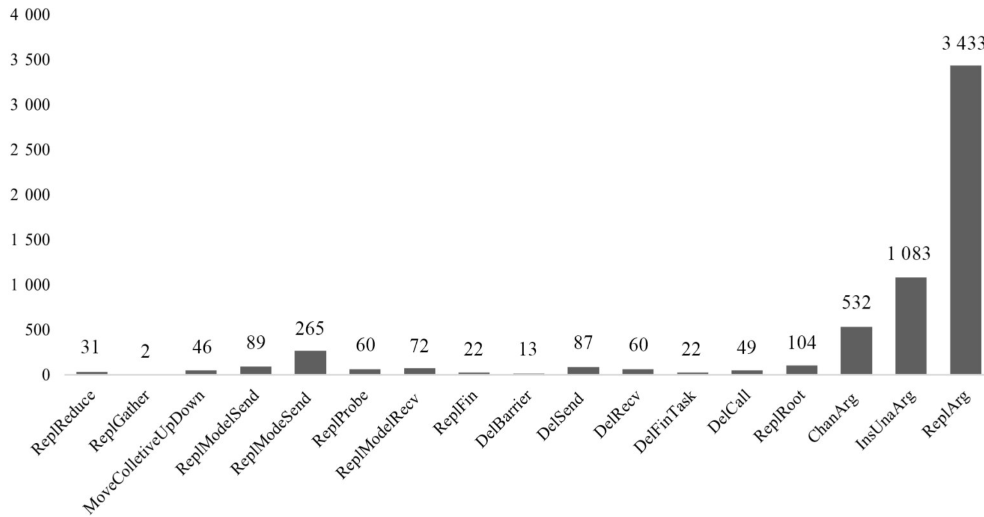

图2　变异体数量

分析所有被测程序中有效的顽固变异体的分布. 表3列出了在所有被测程序中发现的顽固变异体数以及这些顽固变异体的平均被杀死概率PMK,其中在9个程序中发现了顽固变异体,它们被杀死的平均概率范围为7.09%~31.70%. 由表3可知,DIT代码行数仅有86,却生成了43个顽固变异体,而262行代码的Pearson只生成了4个顽固变异体,这表明生成顽固变异体的概率和程序的规模没有必然关系. 在5.3.4节详细讨论不同程序生成顽固变异体数存在差异的原因.

分析不同变异算子生成等价变异体、崩溃变异体和顽固变异体的概率. 表4和图3展示了不同变异算子的MOER、MOCR和MOSR. 除了等价变异体、崩溃变异体和顽固变异体以外,剩余变异体均为易杀死变异体,即该变异体的PMK大于0.5. 加粗显示的为80%以上的MOER、MOCR和非0的MOSR. ReplModelSend和DelBarrier的变异算子等价率MOER最高为100%. 这是因为确定通信的接收模式为阻塞接收,该模式下接收流程持续等待直到消息发送完成,ReplModelSend将

表3　顽固变异体分布

程序名称	顽固变异体数	平均PMK/%
Vector	0	—
Prime	5	24.20
PI	9	24.00
DIT	43	7.09
Factorial	7	14.21
Gather	0	—
Array	0	—
Integrate	0	—
Graph	0	—
Mpi_mm	0	—
Mergesort	4	16.74
Pearson	4	25.82
Crystal	86	31.70
Convex	31	17.30
IS	99	23.79

表4　生成不同变异体概率　　单位:%

变异算子	MOER	MOCR	MOSR
ReplReduce	35.48	64.52	0.00
ReplGather	0.00	**100.00**	0.00
MoveColletiveUpDown	23.91	32.61	0.00
ReplModelSend	**100.00**	0.00	0.00
ReplModeSend	53.58	46.42	0.00
ReplProbe	8.33	20.00	**33.33**
ReplModelRecv	26.39	15.28	**27.78**
ReplFin	13.64	**86.36**	0.00
DelBarrier	**100.00**	0.00	0.00
DelSend	0.00	**100.00**	0.00
DelRecv	8.33	25.00	**31.67**
DelFinTask	13.64	**86.36**	0.00
DelCall	22.45	51.02	0.00
ReplRoot	47.12	17.31	**2.88**
ChanArg	20.86	77.07	0.00
InsUnaArg	14.31	77.01	**1.66**
ReplArg	21.15	48.35	**6.06**

发送函数由阻塞模式改为非阻塞模式，不能改变程序状态；与程序结构相关，DelBarrier变异算子生成的变异体全部与原程序等价，具体原因将在5.3.4节讨论．ReplGather的变异算子崩溃率MOCR为100%．原因是Gather函数用于收集其他流程的数据到指定流程中，而Allgather函数将收集到的数据发送给所有流程．根据变异算子ReplGather的规则，将Gather函数替换为Allgather函数后，某些流程中可能没有对应的接收变量或足够的接收空间，导致程序崩溃．DelSend的MOCR为100%的原因为，在确定通信环境下，删除发送函数一定会导致接收函数的持续等待从而使程序死锁．ReplFin生成崩溃变异体的概率为86.36%．这是由于ReplFin将异常终止函数替换为正常终止函数生成了大量等价变异体，而将正常终止函数替换为异常终止函数会导致程序非正常结束，引发程序崩溃．同理，DelFinTask算子使并行环境非正常结束，生成大量崩溃变异体．在所有变异算子中，ReplProbe的MOSR最大为33.33%，最有可能生成顽固变异体．

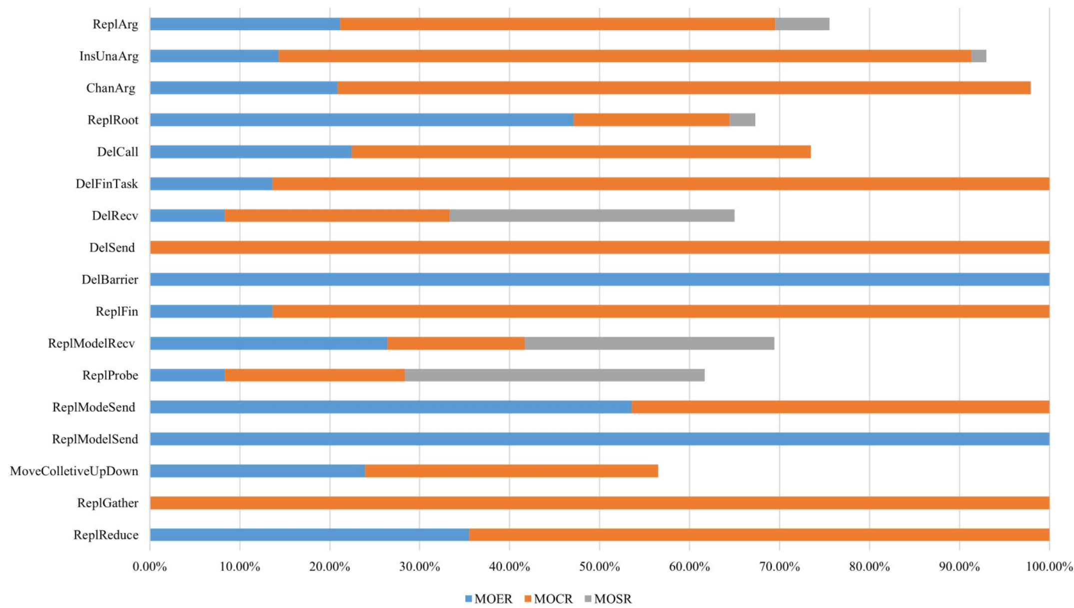

图3 不同变异体概率分布

根据本文所提准则，生成顽固变异体越多，变异算子越有效，生成等价和崩溃变异体越多，变异算子有效性越低．为便于阐述，将变异算子分为A、B、C三类，其中，能生成顽固变异体为A类变异算子．崩溃变异体指在运行过程中异常终止或崩溃的变异体．生成这类变异体的变异算子能够模拟程序中的典型错误，例如，内存泄漏、非法访问、死锁等．等价变异体与原程序语义等价，无法评价测试数据有效性，是阻碍变异测试效率提高的主要原因之一[8]．因此，与等价变异体相比，生成崩溃变异体占比高的变异算子在测试中更有意义，其对应的测试数据能够检测类似的实际缺陷．因此，将变异算子的MOCR高于MOER两倍的归为B类变异算子；其余变异算子主要生成无效的等价变异体，作为C类，即A类变异体算子的有效性最大，B类次之，C类最小．

由表4可知，生成顽固变异体的变异算子ReplRoot、InsUnaArg、ReplProbe、ReplModelRecv、DelRecv和ReplArg属于A类变异算子．基于不同变异算子生成顽固变异体的平均PMK对A类变异算子进一步排序．表5列出了A类变异算子的平均PMK值．可以看出，ReplModelRecv生成的顽固变异体的平均被杀死概率最低，ReplRoot的平均PMK最高，分别为8.73%和36.55%．PMK值越小，顽固性越高，按变异算子的有效性从高到低依次为ReplModelRecv、InsUnaArg、ReplProbe、DelRecv、ReplArg和ReplRoot．

表5 变异算子顽固分布

变异算子	顽固变异体数	PMK/%
ReplRoot	3	36.55
InsUnaArg	18	18.40
ReplProbe	20	21.51
ReplModelRecv	20	8.73
DelRecv	19	21.75
ReplArg	208	24.61

其次，根据MOCR和MOER对B类和C类变异算子排序．ReplGather和DelSend生成崩溃变异体的概率为100%，为*B*类变异算子．此外，ChanArg、ReplFin、DelFintask和DelCall变异算子的崩溃率MOCR比等价率MOER高两倍以上．这些变异算子属于B类变异算子．其余变异算子生成大量的等价变异体或生成等价

变异体和崩溃变异体的概率相近,归为C类变异算子.表6列出了排序后的变异算子.上述分析可知,如果利用变异测试改进测试数据集质量,注入变异算子时可以优先选择A类变异算子;如果注重并行程序的故障复制、死锁检测等,可以使用A类和B类变异算子.

表6　变异算子排序

类别	变异算子
A	ReplModelRecv, InsUnaArg, ReplProbe, DelRecv, ReplArg, ReplRoot
B	DelSend, ReplGather, ReplFin, DelFintask, ChanArg, DelCall
C	ReplReduce, MoveColletiveUpDown, RepModeSend, DelBarrier, ReplModelSend

进一步分析,A类变异算子中ReplRoot、InsUnaArg和ReplArg都对并行函数实施修改操作,尤其是对函数参数的修改.这表明在确定执行的环境中,对分布存储并行程序的修改操作更容易产生顽固变异体,删除操作更容易产生崩溃变异体.此外,Barrier用于流程消息的同步,在消息发送和接收一致的确定通信中,删除同步函数不会影响程序的执行,因此,删除同步操作会生成大量等价变异体.

5.3.2　评价准则的有效性

表7列出了使用A类变异算子生成的变异体数$|M_A|$、所有变异算子生成的变异体数$|M_o|$、A类算子生成的变异体数占所有变异体数的比例PEM和在所有变异体中的变异得分MS. 分析表7可知,使用A类变异算子后所有程序的平均PEM为77.39%,Vector的PEM最小,为61.90%. Pearson的PEM最大,为90.66%,仅减少了9.34%的变异体生成,出现这一现象的原因是该程序包含的集合通信函数较多.ReplRoot、InsUnaArg和ReplArg这3个A类变异算子使用频率较高,从而生成了大量变异体,导致该程序约减的变异体数最少.利用杀死A类变异算子生成变异体的测试数据T_A,检测所有变异体,变异得分为100%,即使用T_A也可以杀死其余变异算子生成的变异体,这是因为这些变异算子主要生成无效的崩溃变异体和容易杀死的变异体,很容易被检测到.这也表明,针对顽固变异体生成的测试数据能够保证覆盖范围.顽固变异体通常位于程序的核心功能区域或复杂区域,针对顽固变异体的测试数据会覆盖更多的边界情况.因此,它们也能够检测到其他变异体,从而实现较高的变异得分.综上所述,根据所提准则得到的A类变异算子在维持测试数据有效性的前提下,平均约减了22.61%的变异体,验证了本文所提准则的有效性.

5.3.3　选择变异

表8列出了采用参数替换算子集AR、集合通信函数修改算子集CM和点对点通信函数修改算子集PM三类算子集进行变异测试的实验结果.值得注意的是,所有算子集的变异得分均值都大于80%,且崩溃变异体对变异得分的贡献很大.除Pearson程序外,其余14个程序中崩溃变异体的占比均超过55%,其中,Array程序最高,Pearson最低,分别为85.98%和39.39%.

表7　A类变异算子有效性

程序	$\|M_A\|$	$\|M_o\|$	PEM/%	MS/%
Vector	39	63	**61.90**	100
Prime	59	86	68.60	100
PI	233	302	77.15	100
DIG	114	172	66.28	100
Factorial	242	333	72.67	100
Gather	63	82	76.83	100
Array	150	174	86.21	100
Integrate	219	263	83.27	100
Graph	250	312	80.13	100
Mpi_mm	553	636	86.95	100
Mergesort	58	79	73.42	100
Pearson	495	546	**90.66**	100
Crystal	617	794	77.71	100
Convex	536	709	75.60	100
IS	1 184	1 419	83.44	100

AR算子集生成的变异体占比在70%以上,所有程序的变异得分均超过85%,说明AR算子集在缺陷检测方面表现优异,与A类变异算子的实验结果相近.具体地,Vector、Prime、Gather、Array和Graph程序的变异得分达到100%;由于某些顽固变异体未被检测到,DIG程序的变异得分最低.上述结果表明,针对AR的测试数据集可以有效杀死变异体.进一步分析可知,AR算子集包含4个变异算子,其中ReplRoot、InsUnaArg和ReplArg变异算子同样属于A类变异算子,这也验证了本文所提准则在选择变异算子方面的有效性.

CM和PM算子集通过较少的变异体达到了较高的变异得分.由图2所示,变异算子InsUnaArg和ReplArg生成的变异体数分别占总变异体数的18.14%和57.50%,合计达75.64%. CM和PM生成变异体占比较低的原因在于,这两个变异算子集中均不包含InsUnaArg和ReplArg. 此外,由于集合通信函数在被测程序中使用的频率较低,导致使用集合通信函数修改算子集CM生成的变异体数偏少.CM算子集在多个程序中只需不到5%的变异体即可实现超过75%的变异得分.例如,Crystal仅使用0.38%的变异体达到了77.19%的变异得分,但是在被杀死的变异体中,崩溃变异体占73.73%,即测试数据集仅杀死了少量的有效变异体.在Vector、Gather、Array和Mpi_mm中,变异得分均超过90%,主要因为这些程序生成的均为崩溃变异体或易杀

表8 选择变异实验结果 单位:%

程序	崩溃变异体占比	AR		CM		PM	
		PEM	MS	PEM	MS	PEM	MS
Vector	73.68	73.02	100.00	26.98	98.25	—	—
Prime	75.00	76.74	100.00	4.65	85.29	18.60	76.47
PI	58.98	83.11	97.66	3.64	85.16	13.25	91.41
DIG	69.06	75.58	89.93	1.74	73.38	22.67	86.33
Factorial	68.99	79.88	95.74	0.90	83.33	19.22	93.80
Gather	83.33	86.59	100.00	13.41	94.87	—	—
Array	85.98	93.10	100.00	6.90	97.56	—	—
Integrate	67.14	90.11	98.10	1.14	83.33	8.75	92.86
Graph	70.04	88.14	100.00	2.24	89.49	9.62	93.39
Mpi_mm	78.94	90.25	99.12	0.94	94.16	8.81	94.69
Mergesort	69.23	82.28	95.38	7.59	75.38	10.13	84.62
Pearson	39.39	91.76	98.37	5.31	90.20	2.93	95.51
Crystal	73.73	83.88	97.00	0.38	77.19	15.74	92.86
Convex	65.35	81.66	94.34	2.54	89.71	15.80	94.85
IS	84.59	89.01	98.59	3.66	89.22	7.33	94.66

死的变异体.DIG的变异得分最低,为73.38%,结合表3可知,DIG程序包含大量顽固变异体,难以被检测到,导致其变异得分偏低.在PM的实验结果中,"—"表示相应程序不包含点对点通信函数调用.除Prime以外,9个程序的变异得分大于90%,这说明除了崩溃变异体外,PM算子集也生成了一定比例的顽固变异体.Prime变异得分最低的原因在于参数修改变异算子生成的顽固变异体较多,而这未能被面向PM的测试数据集检测到.

能够看出,即使参数替换类算子AR在变异得分上表现最优,仍没有获得选择变异要求的变异得分,为此可能需要对变异算子进行更细致的分类.此外,选择变异需要多次选择和评估变异算子子集的有效性,这不仅要求设计合理的算子选择策略,还需要反复生成并执行不同算子集的变异体.本文提出的准则通过变异体类型与质量评估变异算子,避免了频繁调整算子组合及其验证带来的资源消耗,同时保证了较高的变异测试有效性.

5.3.4 生成不同变异体的原因分析

上述分析可知,不同变异算子生成不同变异体的概率差异很大,特别是某些变异算子生成大量无效变异体,即等价变异体和崩溃变异体,且某些程序中没有发现顽固变异体.

根据PIE理论,杀死变异体的测试数据需要满足3个条件:能够执行到变异体、执行应当感染程序的内部状态、该状态需要传播到程序输出.执行、感染和传播任意一个实现概率较小,都可能导致变异体难以被检测到.程序语句间存在的数据依赖和控制依赖关系与变异体能否被检测到密切相关[49].

结合程序特点和PIE理论,从并行函数所在位置、变异算子的特点和程序依赖关系三个因素,探讨分布存储并行程序中生成顽固变异体、崩溃变异体和等价变异体的原因.

(1)并行函数所在位置.程序中语句所处位置包括顺序结构、分支结构和循环结构.位于顺序结构、确定执行次数的循环结构和条件易满足的分支结构下的变异语句,执行概率大,生成顽固变异体的概率小.在Vector、Gather、Mpi_mm等未产生顽固变异体的被测程序中,并行函数大多处于顺序结构下,变异语句容易被执行到,相应地,大多数变异体容易被杀死.DelBarrier变异算子生成的变异体全部与原程序等价.通过分析被测程序,发现Barrier函数都位于传统的顺序语句中,且没有不确定的消息传递.删除Barrier函数对流程通信和程序执行没有影响,对应的变异体与原程序等价.

(2)变异算子特点.变异算子主要是对并行相关函数的修改和删除,不同操作对程序的影响不同.以算法1为例,使用ReplModelSend变异算子生成的变异体M_1为等价变异体,DelSend生成的变异体M_2为崩溃变异体,表明相较于修改操作,删除操作有更大的概率生成崩溃变异体.特别地,由于确定通信环境,流程间消息发送和接收对象是确定的,如果修改发送或接收函数中的目标流程号或者标志参数,大概率会导致由通信错误引起的程序死锁,即修改通信函数变量值的相关变异算子容易生成大量的崩溃变异体.在实验研究中,ChanArg变异算子替换发送函数中的接收流程号和消息标志,生成崩溃变异体的概率为77.07%.

(3)程序依赖关系.分布存储并行程序存在数据依赖、控制依赖和通信依赖关系.控制依赖关系决定了语

句的执行顺序,数据依赖关系描述变量之间是否存在定义-使用关系,通信依赖关系描述消息发送和接收函数间的关系[50]. 条件语句和控制依赖关系密切相关,控制依赖关系决定程序执行的路径和结果,因此,如果变异语句位于分支结构中,执行到该语句需要满足相应的条件,生成顽固变异体概率更高. 由于数据依赖和通信依赖关系,修改发送函数的发送变量和接收函数的接收变量,不会改变原程序的执行路径,原程序和变异体程序执行路径相似性大,需要更高质量的测试数据才能发现错误,即生成顽固变异体的概率更高. 具体地,变异算子 ReplRoot、InsUnaArg 和 ReplArg 修改函数参数,ReplModelRecv、ReplProbe 和 DelRecv 这 3 个变异算子都是删除或替换接收函数,生成的变异体可能改变程序数据依赖和通信依赖关系,因而生成顽固变异体的概率较高. 特别地,ReplModelRecv 将阻塞接收修改为非阻塞接收,导致接收函数能否接收到消息是不确定的,这种不确定使变异体的检测难度增加.

6 结论

为降低分布存储并行程序变异测试的代价,本文提出变异算子有效性评价准则,选择合理的变异算子. 综合考虑不同变异算子生成顽固变异体、崩溃变异体和等价变异体的数量,基于所提准则定义变异算子的评价指标,通过这些指标对多个变异算子进行排序. 此外,从并行函数所在位置、变异算子特点和程序依赖关系三方面分析生成不同变异体的原因,对实验结果进一步分析. 综合考虑顽固、崩溃和等价变异体的测试方法,不仅适用于分布存储并行程序,也为其他程序变异测试提供参考. 具体而言,顽固变异体、崩溃变异体和等价变异体的特性不仅局限于分布存储并行程序,在串行程序、共享存储程序和其他复杂程序中同样存在,通过分析不同类型变异体的生成和检测情况,所提准则能够为不同程序的变异测试提供有效的优化手段. 针对非确定通信的分布存储并行程序,流程间执行顺序的不确定导致无法判断变异体被杀死是由程序输入数据造成的. 综合考虑不同流程顺序下的输出结果是分布存储并行程序变异测试需要进一步研究的工作.

参考文献

[1] GU H F, ZHANG J N, CHEN M S, et al. Specification-driven conformance checking for virtual/silicon devices using mutation testing[J]. IEEE Transactions on Computers, 2021, 70(3): 400-413.

[2] SOUZA S R S, BRITO M A S, SILVA R A, et al. Research in concurrent software testing: A systematic review[C]//Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging. New York: ACM, 2011: 1-5.

[3] JIN K X, LANO K. Design and classification of mutation operators for OCL specification[C]//Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings. New York: ACM, 2022: 852-861.

[4] BASILE D, TER BEEK M H, LAZREG S, et al. Static detection of equivalent mutants in real-time model-based mutation testing[J]. Empirical Software Engineering, 2022, 27(7): 160.

[5] AHMED Z, SCHWASS E, HERBOLD S, et al. A new perspective on the competent programmer hypothesis through the reproduction of real faults with repeated mutations[J]. Software Testing, Verification and Reliability, 2024, 34(3): e1874.

[6] HUMBATOVA N, JAHANGIROVA G, TONELLA P. DeepCrime: From real faults to mutation testing tool for deep learning[C]//2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). Piscataway: IEEE, 2023: 68-72.

[7] 田甜, 巩敦卫. 并发程序变异测试研究综述[J]. 电子学报, 2020, 48(11): 2267-2277.
TIAN T, GONG D W. Survey on mutation testing of concurrent programs[J]. Acta Electronica Sinica, 2020, 48(11): 2267-2277. (in Chinese)

[8] KAUFMAN S J, FEATHERMAN R, ALVIN J, et al. Prioritizing mutants to guide mutation testing[C]//2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE). Piscataway: IEEE, 2022: 1743-1754.

[9] GLIGORIC M, ZHANG L M, PEREIRA C, et al. Selective mutation testing for concurrent code[C]//Proceedings of the 2013 International Symposium on Software Testing and Analysis. New York: ACM, 2013: 224-234.

[10] DU H, PALEPU V K, JONES J A. Ripples of a mutation: An empirical study of propagation effects in mutation testing[C]//Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. New York: ACM, 2024: 1-13.

[11] OFFUTT A J, LEE A, ROTHERMEL G, et al. An experimental determination of sufficient mutant operators[J]. ACM Transactions on Software Engineering and Methodology, 1996, 5(2): 99-118.

[12] DU H, PALEPU V K, JONES J A, et al. To kill a mutant: An empirical study of mutation testing kills[C]//Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. New York: ACM, 2023: 715-726.

[13] LI J X, ZHANG Y M, LU S, et al. Performance bug analysis and detection for distributed storage and computing

systems[J]. ACM Transactions on Storage, 2023, 19(3): 1-33.

[14] CHEKAM T T, PAPADAKIS M, CORDY M, et al. Killing stubborn mutants with symbolic execution[J]. ACM Transactions on Software Engineering and Methodology, 2021, 30(2): 1-23.

[15] GHOSH S. Towards measurement of testability of concurrent object-oriented programs using fault insertion: A preliminary investigation[C]//Proceedings of Second IEEE International Workshop on Source Code Analysis and Manipulation. Piscataway: IEEE, 2002: 17-25.

[16] DELAMARO M, PEZZÈ M, VINCENZI A M R. Mutant operators for testing concurrent Java programs[C]//Anais do XV Simpósio Brasileiro de Engenharia de Software (SBES 2001). Raleigh: Sociedade Brasileira de Computação, 2001: 272-285.

[17] BRADBURY J S, CORDY J R, DINGEL J. Mutation operators for concurrent Java (J2SE 5.0)[C]//Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006). Piscataway: IEEE, 2006: 11.

[18] WU X X, ZHENG W, SHI Z, et al. Concurrency bug-oriented mutation operators design for Java[C]//2018 IEEE International Conference on Progress in Informatics and Computing (PIC). Piscataway: IEEE, 2018: 364-369.

[19] JAGANNATH V, GLIGORIC M, LAUTERBURG S, et al. Mutation operators for actor systems[C]//2010 Third International Conference on Software Testing, Verification, and Validation Workshops. Piscataway: IEEE, 2010: 157-162.

[20] MORADI MOGHADAM M, BAGHERZADEH M, KHATCHADOURIAN R, et al. Akka: Mutation testing for actor concurrency in akka using real-world bugs[C]//Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. New York: ACM, 2023: 262-274.

[21] CAÑIZARES P C, NÚÑEZ A, DE LARA J. OUTRIDER: Optimizing the mUtation Testing pRocess in Distributed EnviRonments[J]. Procedia Computer Science, 2017, 108: 505-514.

[22] ESTERO-BOTARO A, PALOMO-LOZANO F, MEDINA-BULO I. Mutation operators for WS-BPEL 2.0[C]//Proceedings of the 21th International Conference on Software & Systems Engineering and Their Applications. Paris: Genie Lo giciel, 2008: 1-7.

[23] SNIR M, OTTO S, STEVEN H, et al. MPI-the Complete Reference: The MPI Core[M]. Cambridge: MIT Press, 1998.

[24] SILVA R A, DO ROCIO SENGER DE SOUZA S, DE SOUZA P S L. Mutation operators for concurrent programs in MPI[C]//2012 13th Latin American Test Workshop (LATW). Piscataway: IEEE, 2012: 1-6.

[25] KURTZ B, AMMANN P, OFFUTT J, et al. Analyzing the validity of selective mutation with dominator mutants[C]//Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM, 2016: 571-582.

[26] MATHUR A P. Performance, effectiveness, and reliability issues in software testing[C]//Proceedings The Fifteenth Annual International Computer Software & Applications Conference. Piscataway: IEEE, 2002: 604-605.

[27] OFFUTT A J, ROTHERMEL G, ZAPF C. An experimental evaluation of selective mutation[C]//Proceedings of 1993 15th International Conference on Software Engineering. Piscataway: IEEE, 1993: 100-107.

[28] KAZEROUNI A M, DAVIS J C, BASAK A, et al. Fast and accurate incremental feedback for students' software tests using selective mutation analysis[J]. Journal of Systems and Software, 2021, 175: 1-18.

[29] ZHANG S Y, WANG X Y, FENG L C, et al. Mutation operator reduction for deep learning system[C]//2021 7th International Symposium on System and Software Reliability (ISSSR). Piscataway: IEEE, 2021: 32-37.

[30] KIM Y, HONG S. Learning-based mutant reduction using fine-grained mutation operators[C]//2022 IEEE Conference on Software Testing, Verification and Validation (ICST). Piscataway: IEEE, 2022: 464.

[31] 宋利, 刘靖. 基于SOM神经网络的二阶变异体约简方法[J]. 软件学报, 2019, 30(5): 1464-1480.
SONG L, LIU J. Second-order mutant reduction based on SOM neural network[J]. Journal of Software, 2019, 30(5): 1464-1480. (in Chinese)

[32] 王子健. 基于多目标差分进化算法的二阶变异体约简方法研究[D]. 呼和浩特: 内蒙古大学, 2022.
WANG Z J. Research on Second-Order Variation Reduction Method Based on Multi-Objective Differential Evolution Algorithm[D]. Hohhot: Inner Mongolia University, 2022. (in Chinese)

[33] FAN L X, LI Z, LIU H Y, et al. SGS: Mutant reduction for higher-order mutation-based fault localization[C]//2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC). Piscataway: IEEE, 2023: 870-875.

[34] OJDANIC M, SOREMEKUN E, DEGIOVANNI R, et al. Mutation testing in evolving systems: Studying the relevance of mutants to code evolution[J]. ACM Transactions on Software Engineering and Methodology, 2023, 32(1): 1-39.

[35] WANG X Y, ZHANG S Y, LIU F X, et al. MQP: Mutants quality prediction for cost-effective mutation test-

ing[C]//2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C). Piscataway: IEEE, 2021: 45-50.
[36] SUN C G, FU A, GUO X L, et al. ReMuSSE: A redundant mutant identification technique based on selective symbolic execution[J]. IEEE Transactions on Reliability, 2022, 71(1): 415-428.
[37] GARG A, OJDANIC M, DEGIOVANNI R, et al. Cerebro: Static subsuming mutant selection[J]. IEEE Transactions on Software Engineering, 2023, 49(1): 24-43.
[38] TIAN Z, CHEN J J, ZHU Q H, et al. Learning to construct better mutation faults[C]//Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. New York: ACM, 2022: 1-13.
[39] BOSCH J, CARLOS Á, JIMÉNEZ-GONZÁLEZ D, et al. Asynchronous runtime with distributed manager for task-based programming models[J]. Parallel Computing, 2020, 97: 102664.
[40] FU X J, CHEN Z B, ZHANG Y F, et al. MPISE: Symbolic execution of MPI programs[C]//2015 IEEE 16th International Symposium on High Assurance Systems Engineering. Piscataway: IEEE, 2015: 181-188.
[41] CHEN Z B, YU H B, FU X J, et al. MPI-SV: A symbolic verifier for MPI programs[C]//2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). Piscataway: IEEE, 2020: 93-96.
[42] SIEGEL S F, ZIRKEL T K. TASS: The toolkit for accurate scientific software[J]. Mathematics in Computer Science, 2011, 5(4): 395-426.
[43] SIEGEL S F, ZIRKEL T K. FEVS: A functional equivalence verification suite for high-performance scientific computing[J]. Mathematics in Computer Science, 2011, 5(4): 427-435.
[44] Headquarters ASC. CORAL benchmarks[EB/OL]. (2017-12-01)[2024-8-13]. https://asc.llnl.gov/coral-2-benchmarks.
[45] CHEN G, HONG A, SHAN J. The Parallel Algorithm[M]. Beijing: Higher Educaion Press, 2004: 353-355.
[46] DUNBAR J. NAS[CP/OL]. (2023-02-24)[2024-8-13]. https://www.nas.nasa.gov/publications/npb.html.
[47] DANG X Y, YAO X J, GONG D W, et al. Efficiently generating test data to kill stubborn mutants by dynamically reducing the search domain[J]. IEEE Transactions on Reliability, 2020, 69(1): 334-348.
[48] KIM J, JEON J, HONG S, et al. Predictive mutation analysis via the natural language channel in source code[J]. ACM Transactions on Software Engineering and Methodology, 2022, 31(4): 1-27.
[49] 田甜, 邵阳阳, 王苗苗, 等. 基于程序依赖关系的变异体生成策略[J]. 计算机应用, 2024, 44(9): 2863-2870.
TIAN T, SHAO Y Y, WANG M M, et al. Mutant generation strategy based on program dependencies[J]. Journal of Computer Applications, 2024, 44(9): 2863-2870. (in Chinese)
[50] LALLCHANDANI J T, MALL R. Computation of dynamic slices for object-oriented concurrent programs[C]//12th Asia-Pacific Software Engineering Conference (APSEC’05). Piscataway: IEEE, 2005: 8-15.

作者简介

田　甜　女,1987年出生于山东德州. 山东建筑大学计算机科学与技术学院副教授,硕士生导师. 主要研究方向为程序分析与测试、智能软件工程等.

E-mail: tian_tiantian@126.com

李成龙　男,1988年出生于山东济南. 山东建筑大学计算机科学与技术学院副教授,硕士生导师. 主要研究方向为计算机视觉、智能软件测试等.

E-mail: Chenglongli_sdu@163.com

王苗苗　女,2000年出生于河南济源. 山东建筑大学计算机科学与技术学院硕士研究生. 主要研究方向为变异测试.

E-mail: wmm_bee@163.com

巩敦卫　男,1970年出生于江苏铜山. 青岛科技大学自动化与电子工程学院教授,博士生导师. 主要研究方向为多目标优化、智能软件工程等.

E-mail: dwgong@vip.163.com