中国科学: 信息科学 2021 年 第 51 卷 第 4 期: 565–581
SCIENTIA SINICA Informationis
论文

# 基于路径相似度的并行程序多路径覆盖调度序列排序

潘峰[1], 巩敦卫[1*], 田甜[2], 姚香娟[3], 李吟[4]

1. 中国矿业大学信息与控制工程学院, 徐州 221116
2. 山东建筑大学计算机科学与技术学院, 济南 251100
3. 中国矿业大学数学学院, 徐州 221116
4. 江苏自动化研究所, 连云港 222061
* 通信作者. E-mail: dwgong@vip.163.com

收稿日期: 2019–01–30; 修回日期: 2019–05–20; 接受日期: 2019-07–31; 网络出版日期: 2021–03–12

国家自然科学基金 (批准号: 61773384, 61573362, 61503220) 和国家重点研发计划 (批准号: 2018YFB1003802-01) 资助项目

**摘要** 测试是提高软件可靠性的重要方法. 消息传递并行程序中存在的不确定通信语句, 使得进程执行顺序具有不确定性, 这增加了测试该类程序的难度. 鉴于进程执行顺序对目标路径覆盖难易程度的影响, 本文研究消息传递并行程序多路径覆盖调度序列排序方法, 以提高多路径覆盖测试数据生成的效率. 首先, 在每个调度序列下, 以每个采样的程序输入执行程序, 生成路径覆盖矩阵; 然后, 针对每条目标路径, 分别计算与路径覆盖矩阵中每条路径的相似度, 生成多个路径相似度矩阵; 接着, 基于这些路径相似度矩阵的特征量, 评价调度序列的性能, 并依此对调度序列排序; 最后, 基于调度序列排序集, 使用随机采样法, 生成覆盖所有目标路径的测试数据, 并使用缺陷检测平均百分比 (average percentage of faults detected, APFD) 指标评估调度序列排序集. 将所提方法应用于 9 个基准并行程序中, 并与随机方法和传统方法进行比较. 实验结果表明, 所提方法对路径覆盖率没有影响, 但显著减少了被测程序执行次数和运行时间.

**关键词** 消息传递并行程序, 多路径覆盖, 测试, 调度序列排序, 路径相似度

## 1 引言

测试是提高软件可靠性的重要方法, 但往往消耗很多成本. 在软件开发中, 50% 以上的成本通常用于软件测试 [1]. 各种软件的广泛应用, 使得高性能测试方法的研究日益迫切.

测试软件时, 结构覆盖准则常用于评价测试的充分性, 并引导测试数据的生成. 在诸多结构覆盖准则中, 路径覆盖最为常用, 具有很强的缺陷检测能力 [2]. 所谓路径覆盖, 是指在被测程序输入空间中

**引用格式:** 潘峰, 巩敦卫, 田甜, 等. 基于路径相似度的并行程序多路径覆盖调度序列排序. 中国科学: 信息科学, 2021, 51: 565–581, doi: 10.1360/SSI-2019-0113
Pan F, Gong D W, Tian T, et al. Path similarity-based scheduling sequence sorting for multi-path coverage of parallel programs (in Chinese). Sci Sin Inform, 2021, 51: 565–581, doi: 10.1360/SSI-2019-0113

 

搜索某一程序输入作为测试数据, 使被测程序在该测试数据作为输入时, 能够覆盖给定的目标路径 [3]. 而多路径覆盖是指, 对给定的多条目标路径, 同时找到覆盖这些路径的测试数据 [4].

并行程序是指包含两个或多个并行执行进程的程序. 并行程序具有良好的可移植性、功能强大且执行效率高. 因此, 很多大规模科学或工程计算, 如油气勘探、互联网服务, 以及气象预报等, 通常采用并行程序实现 [5]. 消息传递并行程序是一类重要的并行程序, 通常包含一定数量的通信语句, 用于进程之间传递信息. 消息传递并行程序存在的不确定通信语句, 使得进程执行顺序存在不确定性. 由于进程执行顺序的不确定性, 在给定相同程序输入的情况下, 消息传递并行程序的多次运行, 将得到不同的覆盖路径. 调度序列即指进程执行的顺序. 研究表明, 在不同的调度序列下, 路径覆盖测试数据生成的难易程度不同 [6], 对于多路径覆盖更是如此. 给定一组目标路径, 如果在一个难以覆盖大多数目标路径的调度序列下生成多路径覆盖测试数据, 那么, 效率将大大降低.

鉴于此, 对于包含不确定通信语句的消息传递并行程序, 寻找高性能的调度序列, 并在少数高性能调度序列下生成覆盖所有目标路径的测试数据, 是从调度序列的角度提高多路径覆盖测试效率的重要途径. 因此, 本文针对消息传递并行程序多路径覆盖测试数据生成问题, 研究调度序列排序方法, 基于排序后的调度序列集合, 生成覆盖所有目标路径的测试数据. 思想如下: 首先, 在每个调度序列下, 以每个采样的程序输入执行程序, 生成路径覆盖矩阵; 然后, 对每条目标路径, 分别计算与路径覆盖矩阵中每条路径的相似度, 生成多个路径相似度矩阵; 接着, 基于这些路径相似度矩阵的特征量, 评价调度序列的性能, 并依此对调度序列排序; 最后, 基于调度序列排序集, 使用随机采样法, 生成测试数据覆盖所有目标路径, 并使用缺陷检测平均百分比 (average percentage of faults detected, APFD) 指标评估调度序列排序集.

本文的贡献主要体现在: (1) 给出了多路径覆盖准则下的调度序列排序方法; (2) 为了评估调度序列排序集, 提出了基于调度序列排序集的多路径覆盖测试数据随机采样生成方法.

本文结构如下: 第 2 节评述相关工作; 第 3 节阐述提出的方法, 包括路径相似度矩阵生成、调度序列性能评价和排序、多路径覆盖测试数据生成与评估; 第 4 节是所提方法在基准并行程序的应用和对比实验; 第 5 节总结全文, 并指出需要进一步研究的问题.

## 2 相关工作

本节从如下两个方面, 总结前人的研究工作, 主要包括并行程序测试以及路径覆盖测试. 并在此基础上, 引出本文的研究动机.

### 2.1 并行程序测试

Bianchi 等 [7] 根据执行流之间交互的机制, 把并行系统分为 2 类: 共享内存和消息传递系统. 在共享内存系统中, 执行流通过访问一块共享的内存交互数据; 而在消息传递系统中, 执行流通过交换消息交互数据. Su 等 [8] 将共享内存系统中常见的并发缺陷分为死锁、数据竞争、原子性违背和顺序违背等 4 类.

死锁是常见的并发程序缺陷, 有效检测程序中的死锁能够大大提高程序的可靠性. 为此, Cai 等 [9] 提出一种主动检测并发程序死锁的方法, 试图触发真正的死锁事件. 如果触发了死锁, 它将执行静态分析以识别导致死锁的敏感访问. 为了降低分析死锁的复杂性, Liu 等 [10] 提出一种分解方法, 将并发程序的 Petri 网络模型分解为多个基于进程数量和消息传递语句位置的进程网络, 再通过分析进程网络死锁识别并发程序的死锁. 为了比较 Pthreads 和 Dthreads 在预防死锁、数据竞争和竞争条件上的

优劣, Fei 等[11] 将通信顺序过程作为描述 Pthreads 和 Dthreads 中特定 API 函数的形式模型, 并通过此模型证明了 Dthreads 在消除数据竞争和阻止死锁上优于 Pthreads. 为了检测消息传递并行程序的死锁, Hilbrich 等[12] 提出一种分布式死锁检测算法, 此算法基于消息传递接口锁定状态模型, 用于大规模检测消息传递并行程序的死锁.

针对 happens-before 方法容易隐藏数据竞争缺陷的问题, Cai 等[13] 提出一种准确有效的动态技术, 用于检测被隐藏的数据竞争缺陷, 通过反转可能存在上次执行的 happens-before 关系集合, 暴露隐藏的数据竞争缺陷. 通过分析串行测试的执行场景并生成变量锁依赖关系, Samak 等[14] 提出一种新颖且可扩展的方法, 用于检测多线程程序的原子性违规缺陷. Yin 等[15] 提出一种基于调度约束的抽象细化方法, 用于解决使用有限模型验证算法时, 需要编码大量复杂公式的问题, 并开发了有效的多线程程序验证工具. Yu 等[16] 融合符号执行和模型检验, 针对包含阻塞和非阻塞通信的消息传递并行程序, 开发了首个基于符号执行的消息传递并行程序验证工具.

此外, 并发缺陷检测在并发程序调试过程中是非常耗时的. 为了有效降低并发缺陷检测的时间消耗, Wang 等[17] 提出一种基于程序不变体的缺陷检测方法, 以检测一类并发缺陷. 该方法使用组件函数调用图, 并把约简技术应用于不变体, 找到可疑函数并排序, 基于此分析导致并发缺陷的原因. 为了解决缺陷检测与调试过程分离导致的缺陷难以复现的问题, Wu 等[18] 提出一种并发程序行为可视化方法, 该方法静态分析并发程序行为, 为程序员提供全局视图, 帮助他们观察程序的行为并且引导他们注意可疑操作. 为了大幅度提高并发程序分析与测试的效率, Qi 等[19] 采用偏序约简技术, 使基于程序可达图的并发程序切片方法, 在保证切片精度不受损失的前提下显著提高切片效率. 进一步, 基于可达性测试动态框架和变强度组合策略, Qi 等[20] 提出并发程序变强度可达性测试方法, 保证了缺陷检测能力, 极大减少了同步序列的个数, 并在大多数情况下缩短了测试时间.

可以看出, 目前关于消息传递系统测试的研究还很少, 有关并行程序的研究主要集中于共享内存系统. 本文的研究对象是消息传递并行程序, 这有助于丰富消息传递系统测试的研究.

### 2.2 路径覆盖测试

路径覆盖是结构覆盖测试中常用的测试充分性准则. 静态方法、传统动态方法, 以及启发式方法是 3 种路径覆盖测试数据生成方法. 静态方法基于对程序内部结构的分析, 不需要执行程序, 如符号执行和区间算法. 然而, 该方法存在若干困难, 包括: 解决复杂约束、处理过程调用和非序数变量. 传统动态方法通过一些输入执行程序并观察结果, 如随机法、目标导向法和链接法. 传统动态方法的缺点是, 即使存在测试数据, 该方法也不保证找到.

启发式方法使用诸如遗传算法的元启发式优化技术自动生成测试数据. Liao[21] 利用人工鱼群优化, 生成覆盖目标路径的测试数据, 并根据人工鱼能否穿越分离的子路径, 缩减搜索空间, 使得该算法在时间开销、成功率及算法稳定性等方面均具有优越性. 为了最大限度地减少测试用例, Liu 等[22] 提出一种带有收敛速度控制器的进化算法, 具有快速的收敛速度, 并能够在一系列问题上跳出局部最优解. Huang 等[23] 提出一种改进的元启发式算法, 利用启发式信息生成少量测试用例, 以覆盖尽可能多的路径. Panichella 等[24] 提出一种动态多目标排序算法, 基于控制依赖性层次结构动态选择覆盖目标, 用于结构覆盖测试用例生成问题. 为了应对工业网络物理系统测试用例生成和优先化面临的挑战, Arrieta 等[25] 通过定义 4 个目标函数并设计不同的交叉和变异算子, 提出一种多目标测试用例生成和优先化方法, 测试工业网络物理系统.

对于并行程序测试数据生成, 也有一些研究工作. Hwang 等[26] 基于可行同步序列的全序图, 实现并发程序的符号执行, 并使用穷举法生成测试数据. 一个复杂的并行程序包含较多条件语句和同步

序列. 对这种复杂的并行程序, 使用符号执行法, 容易导致路径状态空间爆炸. 为了解决该问题, Ding 等 [27] 基于程序语义和条件计算规则, 生成串行程序测试数据, 再组合这些测试数据, 生成并行程序的测试数据. Souza 等 [28] 还提出一种新的复合方法, 该方法根据特定的结构测试准则, 使用可达性测试引导同步序列的选择. Wang [6] 考虑消息传递并行程序的执行具有不确定性, 提出一种调度序列选择方法, 对于给定的一条目标路径, 在性能最优的调度序列下, 生成覆盖该路径的测试数据. 基于并行程序的路径表示以及定义的等价路径, Tian 等 [29] 曾提出一种并行程序测试数据进化生成方法, 该方法能够高效生成测试数据. 此外, 我们 [30] 还提出一种新颖的消息传递并行程序调度序列约简方法, 根据指标体系选择最优调度序列, 并基于最优调度序列生成测试数据, 以提高单路径覆盖测试数据生成的效率. 以上工作 [6,29,30] 虽然考虑了不确定性, 但都是针对单路径覆盖测试, 而本文研究的是多路径覆盖测试. 为了有效验证并行程序测试工具的效果, Siegel 等 [31] 提供了一个消息传递并行程序的基准测试程序集, 这些基准并行程序被广泛应用于消息传递并行程序测试.

目前, 多路径覆盖的研究很少. 为了生成串行程序多路径覆盖测试数据, Ahmed 等 [4] 首次将此问题转化为多目标优化问题, 以同时生成覆盖多条目标路径测试数据. 针对 Ahmed 方法效率不高的问题, Lv 等 [32] 提出一种基于蜕变关系的多路径覆盖测试数据生成方法, 生成几个测试数据之后, 多次使用蜕变关系生成新的测试数据. 为了生成消息传递并行程序多路径覆盖测试数据, Tian 等 [33] 将此问题建模成一个包含多个目标的优化问题, 以同时生成所有期望的测试数据. 虽然他们考虑了不确定性会影响目标路径的节点顺序, 由此利用了等价路径的概念, 但是, 没有考虑调度序列的性能对目标路径覆盖难易程度的影响.

可以看出, 目前, 在提高消息传递并行程序多路径覆盖测试数据生成效率方面, 尚没有工作从调度序列的角度进行研究. 本文研究调度序列排序方法, 寻找高性能的调度序列, 并在少数高性能调度序列下生成覆盖所有目标路径的测试数据.

## 3 提出的方法

本节提出基于路径相似度的调度序列排序方法, 目的是基于排序后的调度序列生成多路径覆盖测试数据, 提高测试数据生成的效率. 所提方法的思想是: 首先, 在每个调度序列下, 以每个采样的程序输入执行程序, 生成以覆盖路径为元素的路径覆盖矩阵; 然后, 针对每条目标路径, 分别计算与路径覆盖矩阵中每个元素的相似度, 得到多个以路径相似度为元素的路径相似度矩阵; 接着, 基于这些路径相似度矩阵的特征量, 评价调度序列的性能, 并基于此对调度序列排序; 最后, 基于调度序列排序集, 对程序输入空间随机采样, 生成多路径覆盖测试数据, 并使用 APFD 指标评估调度序列排序集.

可以看出, 路径相似度矩阵生成、调度序列性能评价和排序, 以及多路径覆盖测试数据生成与评估是所提方法的关键. 在给出相应的解决策略之前, 为了便于读者理解本文提出的方法, 首先引出一些基本概念.

### 3.1 基本概念

记并行程序为 $S$, 由 $t$ $(t>1)$ 个进程组成, 第 $i$ $(i=0,1,2,\ldots,t-1)$ 个进程为 $S^i$. 这 $t$ 个进程并行执行, $S$ 可以表示为 $S=\{S^0,S^1,\ldots,S^{t-1}\}$. 记 $S$ 的输入向量为 $x=\{x_1,x_2,\ldots,x_{n_s}\}$, 其中, $x_i$ $(i=1,2,\ldots,n_s)$ 为第 $i$ 个输入分量, 取值范围为 $D_{x_i}$, 那么, $x$ 的取值范围为 $D=D_{x_1}\times D_{x_2}\times\cdots\times D_{x_{n_s}}$.

节点可以是消息发送或接收语句, 也可以是分支语句的判断条件或循环语句的循环条件. 记 $S^i$ 的第 $j$ 个节点为 $n_j^i$. 若 $n_j^i$ 是一条消息接收 (发送) 语句, 则称 $n_j^i$ 为消息接收 (发送) 节点.

**表 1 示例程序的调度序列**

**Table 1** Scheduling sequence of the sample program

| Scheduling sequence | $n_5^0$ | $n_6^0$ | $n_7^0$ | Corresponding process execution order | Representation of scheduling sequence |
|---|---|---|---|---|---|
| $SS_1$ | $n_8^1$ | $n_8^2$ | $n_8^3$ | $S^1S^2S^3$ | $(1,2,3)$ |
| $SS_2$ | $n_8^1$ | $n_8^3$ | $n_8^2$ | $S^1S^3S^2$ | $(1,3,2)$ |
| $SS_3$ | $n_8^2$ | $n_8^1$ | $n_8^3$ | $S^2S^1S^3$ | $(2,1,3)$ |
| $SS_4$ | $n_8^2$ | $n_8^3$ | $n_8^1$ | $S^2S^3S^1$ | $(2,3,1)$ |
| $SS_5$ | $n_8^3$ | $n_8^1$ | $n_8^2$ | $S^3S^1S^2$ | $(3,1,2)$ |
| $SS_6$ | $n_8^3$ | $n_8^2$ | $n_8^1$ | $S^3S^2S^1$ | $(3,2,1)$ |

以输入 $x \in D$ 执行 $S$, 穿越 $S^i$ 的节点序列为 $p_i = n_1^i \cdots n_j^i \cdots n_{|S^i|}^i$, 记 $|p_i|$ 为子路径长度, 其中, $|S^i|$ 为 $S^i$ 的节点数, $n_1^i$ 和 $n_{|S^i|}^i$ 分别为 $S^i$ 的入口和出口节点. 多个进程的路径的组合形成并行程序的路径, 那么, $p = p_0p_1 \cdots p_{t-1}$ 为并行程序 $S$ 在 $x$ 作为输入时覆盖的路径.

记目标路径为 $p^* = p_0^*p_1^*p_2^* \cdots p_{t-1}^*$, $x$ 覆盖的路径为 $p = p_0p_1 \cdots p_{t-1}$, 那么目标路径和覆盖路径的路径相似度 [29] 为

$$\mathrm{sim}(p^*, p) = \frac{1}{t}\sum_{i=0}^{t-1}\frac{|p_i^* \cap p_i|}{\max\{|p_i^*|, |p_i|\}}, \tag{1}$$

其中 $|p_i^* \cap p_i|$ 表示 $p_i^*$ 与 $p_i$ 从前到后连续相同的节点数.

图 1 所示的源程序, 其功能是求 6 个整数的最大公约数. 现在通过这个程序, 说明调度序列的概念. 该程序共有 5 个进程, 分别为 $S^0$, $S^1$, $S^2$, $S^3$ 和 $S^4$. 首先, $S^0$ 把 6 个整数平均分成 3 组, 分别发送给 $S^1$, $S^2$ 和 $S^3$, $S^1$, $S^2$ 和 $S^3$ 分别计算最大公约数, 并把计算结果发送给 $S^0$; 然后, $S^0$ 把接收的前 2 个计算结果发送给 $S^4$, $S^4$ 计算最大公约数后, 把计算结果发送给 $S^0$; 最后, $S^0$ 对接收的第 3 个计算结果和从 $S^4$ 接收的计算结果计算最大公约数, 并将计算结果输出至屏幕.

考虑图 1 所示的消息传递并行程序, 图 1(a) 语句 2 为一个消息发送语句, 其目的地址和标签参数分别为 1 和 1, 表示给进程 1 发送标志为 1 的消息; 图 1(b) 语句 2 为一个消息接收语句, 其源地址和标签参数分别为 0 和 1, 表示接收来自进程 0 的标志为 1 的消息. 由于图 1(b) 语句 2 和图 1(a) 语句 2 的匹配, 因此, 前者能接收后者的消息.

由图 1 可知, 在 $S^0$ 中, 接收语句 $n_5^0$, $n_6^0$ 和 $n_7^0$ 中含有参数 MPI_ANY_SOURCE, 因此, 这 3 条接收语句是不确定接收语句, 即接收语句 $n_5^0$, $n_6^0$ 和 $n_7^0$ 可以和 $S^1$ 的发送语句 $n_8^1$、$S^2$ 的发送语句 $n_8^2$ 和 $S^3$ 的发送语句 $n_8^3$ 任意匹配. 这种发送和接收语句的不确定匹配, 使得消息传递并行程序的进程执行顺序具有不确定性. 比如, 若 $n_5^0$ 和 $n_8^1$, $n_6^0$ 和 $n_8^2$, $n_7^0$ 和 $n_8^3$ 匹配, 则进程执行顺序为 $S^1S^2S^3$; 若 $n_5^0$ 和 $n_8^3$, $n_6^0$ 和 $n_8^1$, $n_7^0$ 和 $n_8^2$ 匹配, 则进程执行顺序为 $S^3S^1S^2$. 相应进程执行顺序如图 2 和 3 所示. 表 1 列出了所有调度序列, 表中第 1 列是调度序列名称, 第 2∼4 列分别是上述 3 个不确定接收语句, 第 5 列为对应调度序列的进程执行顺序, 第 6 列为对应调度序列的表示.

### 3.2 路径相似度矩阵生成

采用合适的方法, 对程序的输入空间采样若干次, 得到一个程序输入集, 记为 $X = \{X_1, X_2, \ldots, X_m\}$, 其中 $m$ 为样本个数. 记所有调度序列形成的集合为 $\mathrm{SS} = \{\mathrm{SS}_1, \mathrm{SS}_2, \ldots, \mathrm{SS}_n\}$, 其中 $n$ 为调度序列的个数. 以 $X_i$ 为程序输入, 在调度序列 $\mathrm{SS}_j$ 下运行程序, 记覆盖的路径为 $q_{ij}$. 在每一调度序列下,

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char** argv){
1.  int myid, num, a, b, c, d, buf[2];
    int input[6] = {2, 4, 4, 4, 4, 104};
    MPI_Status status;
    // Parallel environment initialization
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &num);
    // Send the first number and the second number to process 1
2.  MPI_Send(input, 2, MPI_INT, 1, 1, MPI_COMM_WORLD);
    // Send the third and fourth numbers to process 2
3.  MPI_Send(input+2, 2, MPI_INT, 2, 1, MPI_COMM_WORLD);
    // Send the fifth and sixth numbers to process 3
4.  MPI_Send(input+4, 2, MPI_INT, 3, 1, MPI_COMM_WORLD);
    // Uncertain receiving node, receiving calculation results from Process 1, Process 2, and Process 3
5.  MPI_Recv(&a, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
6.  MPI_Recv(&b, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
7.  MPI_Recv(&c, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    buf[0] = a; buf[1] = b;
    // Send the first and second calculation results to process 4
8.  MPI_Send(buf, 2, MPI_INT, 4, 3, MPI_COMM_WORLD);
    // Receive the calculation result of process 4
9.  MPI_Recv(&d, 1, MPI_INT, 4, 4, MPI_COMM_WORLD, &status);
    // Calculate the greatest common divisor of the third calculation result and the calculation result sent by process 4
10. if (c == 1 || d == 1){
        c = 1;
    }
11. else {
12.     while (c != d){
13.         if (c < d){
                d = d - c;
            }
14.         else{
                c = c - d;
            }
        }
    }
    // Output the result, logout parallel environment
15. printf("The greatest common divisor of the six numbers is %d \n", c);
    MPI_Finalize(); return 0;}
```

(a)

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char** argv){
1.  int myid, num, a, b, buf[2];
    MPI_Status status;
    // Parallel environment initialization
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &num);
    // Receive two numbers sent by process 0
2.  MPI_Recv(buf, 2, MPI_INT, 0, 1,MPI_COMM_WORLD, &status);
    a = buf[0]; b = buf[1];
    // Calculate the greatest common divisor of a and b
3.  if (a == 1 || b == 1){
        a = 1;
    }
4.  else{
5.      while (a != b){
6.          if (a < b){
                b = b - a;
            }
7.          else{
                a = a - b;
            }
        }
    }
    // Send calculation result to process 0, logout parallel environment
8.  MPI_Send(&a, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
9.  MPI_Finalize(); return 0;}
```

(b)

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char** argv){
1.  int myid, num, a, b, buf[2];
    MPI_Status status;
    // Parallel environment initialization
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &num);
    // Receive two calculation results sent by process 0
2.  MPI_Recv(buf, 2, MPI_INT, 0, 3, MPI_COMM_WORLD, &status);
    a = buf[0]; b = buf[1];
    // Calculate the greatest common divisor of a and b
3.  if (a == 1 || b == 1){
        a = 1;
    }
4.  else{
5.      while (a != b){
6.          if (a < b){
                b = b - a;
            }
7.          else{
                a = a - b;
            }
        }
    }
    // Send calculation result to process 0, logout parallel environment
8.  MPI_Send(&a, 1, MPI_INT, 0, 4, MPI_COMM_WORLD);
9.  MPI_Finalize(); return 0;}
```

(c)

**图 1　求 6 个整数的最大公约数源程序**

**Figure 1**　A source code of calculating the maximum common divisor for 6 integers. (a) Process $S^0$; (b) process $S^1, S^2, S^3$; (c) process $S^4$

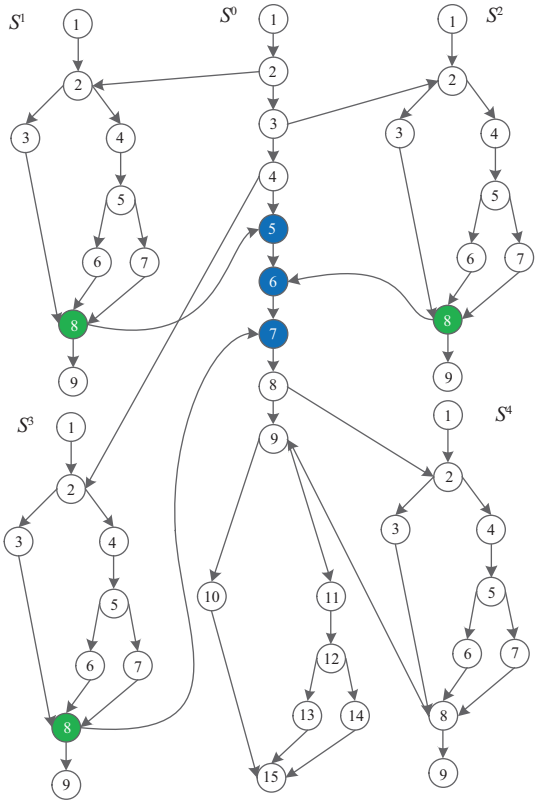


图 2 (网络版彩图) 进程执行顺序 $S^1S^2S^3$ 对应的控制流图

**Figure 2** (Color online) Corresponding control flow graph for the process execution order $S^1S^2S^3$

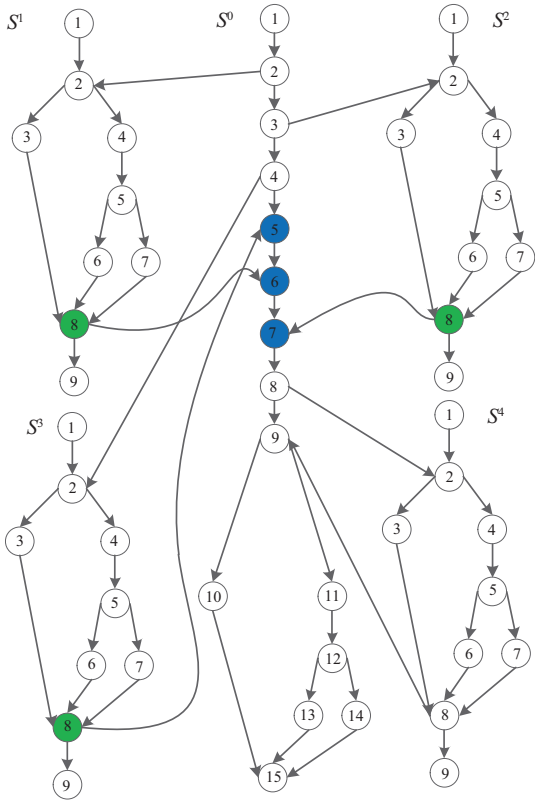


图 3 (网络版彩图) 进程执行顺序 $S^3S^1S^2$ 对应的控制流图

**Figure 3** (Color online) Corresponding control flow graph for the process execution order $S^3S^1S^2$

以每一程序输入运行程序, 能够得到以覆盖路径为元素的路径覆盖矩阵, 记为 $Q$, 那么,

$$Q=\begin{bmatrix} q_{11} & q_{12} & \cdots & q_{1n} \\ q_{21} & q_{22} & \cdots & q_{2n} \\ \cdots & & & \\ q_{m1} & q_{m2} & \cdots & q_{mn} \end{bmatrix}. \tag{2}$$

记所有目标路径形成的集合为 $Q^*=\{q_1^*,q_2^*,\ldots,q_l^*\}$, 其中 $l$ 为目标路径的条数. 考虑目标路径 $q_k^*$, 按式 (1) 计算 $q_k^*$ 与 $q_{ij}$ 的相似度, 记为 $\mathrm{sim}_{ij}^k$. 对于 $Q$ 的所有路径, 得到以 $\mathrm{sim}_{ij}^k$ 为元素的路径相似度矩阵, 记为 $\mathrm{Sim}^k$, 那么,

$$\mathrm{Sim}^k=\begin{bmatrix} \mathrm{sim}_{11}^k & \mathrm{sim}_{12}^k & \cdots & \mathrm{sim}_{1n}^k \\ \mathrm{sim}_{21}^k & \mathrm{sim}_{22}^k & \cdots & \mathrm{sim}_{2n}^k \\ \cdots & & & \\ \mathrm{sim}_{m1}^k & \mathrm{sim}_{m2}^k & \cdots & \mathrm{sim}_{mn}^k \end{bmatrix}. \tag{3}$$

记所有路径相似度矩阵形成的集合为 Sim, 那么, $\mathrm{Sim}=\{\mathrm{Sim}^1,\mathrm{Sim}^2,\ldots,\mathrm{Sim}^l\}$.

### 3.3 调度序列性能评价和排序

考虑路径相似度矩阵 $\mathrm{Sim}^k$, 统计该矩阵第 $j$ 列元素为 1 的个数, 记为 $c_j^k$. $c_j^k$ 越大, 那么, 在调度序列 $\mathrm{SS}_j$ 下, $X$ 中包含更多的测试数据可以覆盖目标路径 $q_k^*$, 因此, 基于该调度序列, 生成测试数据的成功率越高. 评价调度序列 $\mathrm{SS}_j$ 的性能时, 除了 $c_j^k$ 之外, 还考虑 $\mathrm{Sim}^k$ 第 $j$ 列的平均值, 记为 $a_j^k$. $a_j^k$ 越大, 那么, 在调度序列 $\mathrm{SS}_j$ 下, $X$ 中的测试数据覆盖的路径与目标路径 $q_k^*$ 越相似, 因此, 基于该调度序列, 生成测试数据越容易.

由于需要生成覆盖目标路径的测试数据, 因此 $c_j^k$ 比 $a_j^k$ 更重要. 鉴于此, 采用指标 $M^k(\mathrm{SS}_j)$, 评价调度序列 $\mathrm{SS}_j$ 对于目标路径 $q_k^*$ 的性能,

$$M^k(\mathrm{SS}_j) = \alpha \cdot c_j^k + \beta \cdot a_j^k, \tag{4}$$

其中 $\alpha$ 和 $\beta$ 为权重系数, 且满足 $0 < \beta < \alpha < 1$, $\alpha + \beta = 1$.

由式 (4) 可知, $M^k(\mathrm{SS}_j)$ 综合考虑了 $c_j^k$ 和 $a_j^k$, 且该值越大, 那么, 在调度序列 $\mathrm{SS}_j$ 下, 生成覆盖目标路径 $q_k^*$ 测试数据的成功率越高且越容易. 对于所有的目标路径, 采用指标 $M(\mathrm{SS}_j)$, 评价调度序列 $\mathrm{SS}_j$ 的性能, 那么,

$$M(\mathrm{SS}_j) = \frac{1}{l}\sum_{k=1}^{l} M^k(\mathrm{SS}_j). \tag{5}$$

由式 (5) 可知, $M(\mathrm{SS}_j)$ 是调度序列 $\mathrm{SS}_j$ 对所有目标路径性能的平均值, 且该值越大, 那么, 在调度序列 $\mathrm{SS}_j$ 下, 生成覆盖所有目标路径测试数据的成功率越高且越容易. 特别地, 当 $l = 1$ 时, $M(\mathrm{SS}_j) = M^1(\mathrm{SS}_j)$, 此即单路径覆盖调度序列的评价指标 [6]. 容易看出, 这里给出的评价指标是文献 [6] 的推广, 文献 [6] 仅是本文的特例. 对于所有的调度序列, 采用式 (5) 的方法, 能够得到它们的评价结果, 记为 $M$, 那么,

$$M = \{M(\mathrm{SS}_1), M(\mathrm{SS}_2), \ldots, M(\mathrm{SS}_n)\}. \tag{6}$$

记排序后的调度序列形成的集合为 $SS'$. 为了得到该集合, 首先, 将 $M$ 的元素从大到小排序; 然后, 寻找排序后每一元素对应的调度序列, 并按该顺序对调度序列排序.

### 3.4 多路径覆盖测试数据生成与评估

对于需要覆盖的目标路径集, 可能需要基于多个调度序列, 才能覆盖所有目标路径. 因此, 本文基于调度序列排序集中的调度序列, 依次采用随机采样法生成测试数据, 直至所有目标路径被覆盖, 并记录在每个调度序列下覆盖目标路径的个数, 记为 $r_j$ $(j = 1, 2, \ldots, n)$, 再使用 APFD 指标评估调度序列排序集.

具体地说, 首先, 基于调度序列排序集中第 1 个调度序列 $\mathrm{SS}'_1$, 随机采样 Iter 次程序输入, 每次采样后, 在调度序列 $\mathrm{SS}'_1$ 下以采样的程序输入执行被测程序, 若某个目标路径被覆盖, 则 $r_1$ 加 1 并把覆盖的目标路径从目标路径集中删除, 若目标路径集为空, 则停止; 否则, 基于调度序列排序集中下一个调度序列, 重复上述过程, 直至目标路径集为空.

然后, 以调度序列使用比例为 $x$ 轴, 以路径覆盖率为 $y$ 轴, 画折线图, 如图 4 所示. 在调度序列排序集下, 生成覆盖所有目标路径测试数据的 APFD 值即为折线图与 $x$ 轴围成的面积, 计算方法如式 (7) 所示, 式中, APFD 的取值范围为 $[0, 1 - \frac{1}{2n}]$.

$$\mathrm{APFD} = \sum_{i=1}^{l}\frac{r_i}{2nl} + \sum_{i=2}^{l}\sum_{j=1}^{i-1}\frac{r_j}{nl}. \tag{7}$$

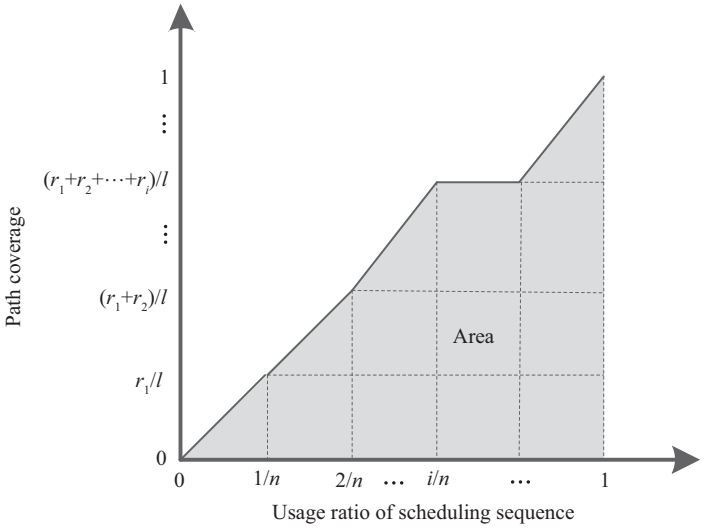


**图 4 调度序列排序集的 APFD 图**

**Figure 4** APFD diagram for the sorted scheduling sequence set

不同调度序列排序集中, APFD 的值越高, 那么, 使用越少的调度序列即可覆盖越多的目标路径, 即基于对应调度序列排序集, 生成多路径覆盖测试数据的效率越高. 基于路径相似度的多路径覆盖调度序列排序的伪代码实现如算法 1 所示.

## 4 实验

本节通过一系列实验, 验证所提方法的有效性. 首先, 提出需要验证的问题; 然后, 介绍被测程序; 接着, 描述实验过程; 最后, 给出实验结果和分析.

### 4.1 需要验证的问题

通过回答如下 2 个问题, 验证所提方法的有效性:

(1) 本文提出的基于路径相似度的多路径覆盖调度序列排序方法的效果如何?

(2) 基于本文所提方法得到的调度序列排序集, 能否提高多路径覆盖测试数据生成的效率?

为了回答这 2 个问题, 首先, 通过比较基于调度序列排序集和调度序列随机排序集生成覆盖所有目标路径测试数据的 APFD 指标, 验证本文提出的多路径覆盖调度序列排序方法的效果; 然后, 通过比较基于调度序列排序集、基于调度序列随机排序集和传统方法生成覆盖所有目标路径测试数据的被测程序执行次数、运行时间, 以及路径覆盖率, 说明采用本文方法生成多路径覆盖测试数据的必要性.

### 4.2 被测程序

选取 9 个基准并行程序作为被测程序, 这些程序的基本信息如表 2 所示, 表中, 程序 T1∼T8 常用于消息传递并行程序路径覆盖测试, 选自文献 [29, 30, 33]; T9 包含不确定通信语句, 选自 FEVS 基准并行程序测试集 [31]. T1 判断 4 个边长值中最大的 3 个能否构成三角形; T2 的功能是两矩阵相乘并寻找结果中最大的 2 个元素; T3 判断点与多边形的位置关系; T4 模拟生产与消费过程; T5 判断 4 个角度值能否构成凸四边形; T6 求取多个数的最小值; T7 统计字符串中数字、字母和其他字符的个数; T8 的功能是字符信息检索; T9 计算三角函数的积分.

**Algorithm 1** Path similarity-based scheduling sequence sorting for multi-path coverage

**for** each $X_i \in X$ **do**
  **for** each $\mathrm{SS}_j \in \mathrm{SS}$ **do**
    Execute the program with input $X_i$ under $\mathrm{SS}_j$;
    $q_{ij}$←the covered path;
  **end for**
**end for**
**for** each $q_k^* \in Q^*$ **do**
  **for** each $q_{ij} \in Q$ **do**
    Calculate $\mathrm{sim}_{ij}^k$;
  **end for**
**end for**
**for** each $\mathrm{Sim}^k \in \mathrm{Sim}$ **do**
  **for** each $\mathrm{Sim}_j^k \in \mathrm{Sim}^k$ **do**
    $c_j^k$←the number of one in $\mathrm{Sim}_j^k$;
    $a_j^k$←the average of $\mathrm{Sim}_j^k$;
    $M^k(\mathrm{SS}_j) \leftarrow \alpha \cdot c_j^k + \beta \cdot a_j^k$;
  **end for**
**end for**
**for** each $\mathrm{SS}_j \in SS$ **do**
  $M(\mathrm{SS}_j) \leftarrow \frac{1}{l} \sum_{k=1}^{l} M^k(\mathrm{SS}_j)$;
**end for**
**for** $i$=1, 2, …, $n$ **do**
  **for** $j$=1, 2, …, $n-i$ **do**
    **if** $M[j]$<$M[j+1]$ **then**
      Swap $M[j]$ and $M[j+1]$;
    **end if**
  **end for**
**end for**
**for** each $M(\mathrm{SS}_j) \in M$ **do**
  Put $\mathrm{SS}_j$ corresponding with $M(\mathrm{SS}_j)$ into $SS'$;
**end for**
**for** $j$=1, 2, …, $n$ **do**
  **while** iter<Iter **do**
    Randomly sample a program input as $x_0$;
    Execute the program with $x_0$ under $\mathrm{SS}_j'$;
    **if** one path in $Q^*$ is covered **then**
      $r_j$++;
      Remove the path from $Q^*$;
    **end if**
    **if** $Q^*$ is empty **then**
      Calculate APFD and terminate;
    **end if**
  **end while**
**end for**

### 4.3 实验过程

实验平台的硬件配置如下: AMD Ryzen 5 处理器、1 T 硬盘和 8 G 内存; 软件采用 Windows 10

**表 2 被测程序的基本信息**

**Table 2** Basic information of programs under test

| ID | Program name | Input space | Number of processes | Number of communication statements | Number of scheduling sequences | Number of target paths |
|---|---|---|---|---|---|---|
| T1 | Max_triangle | $[0,100]^4$ | 5 | 22 | 6 | 20 |
| T2 | Matrix | $[-100,100]^{16}$ | 4 | 12 | 6 | 20 |
| T3 | Including | $[-100,100]^2$ | 5 | 8 | 24 | 25 |
| T4 | Creator_consumer | $[1,100]^2$ | 7 | 18 | 6 | 25 |
| T5 | Convex_quadrilateral | $[0,180]^4$ | 4 | 18 | 6 | 30 |
| T6 | Min | $[-100,100]^{125}$ | 6 | 20 | 120 | 30 |
| T7 | ASCII | $[0,127]^{10}$ | 5 | 8 | 24 | 35 |
| T8 | Index | $[0,127]^{15}$ | 4 | 12 | 6 | 35 |
| T9 | Integrate_mw | $[0,2\pi]^2$ | 20 | 76 | 720 | 40 |

操作系统、Visual Studio 2013 编译器以及消息传递接口标准应用软件 MPICH2.

实验中, 首先, 对于每一个被测程序, 选择一定数量的可达路径形成目标路径集, 保证被选择的路径覆盖所有的分支, 数量如表 2 所示. 设置程序输入集的容量为 $m = 200$, 对程序输入空间随机采样 $m$ 次, 得到程序输入集.

然后, 分析每一个被测程序, 找出不确定接收和发送语句, 那么, 调度序列的个数即为不确定接收语句与不确定发送语句所有可能匹配的个数. 记不确定接收语句个数为 $h$, 则调度序列个数 $n = A_h^h$, 其中, $A_h^h$ 为 $h$ 个整数的全排列个数. 以图 1 的示例程序为例, 不确定接收语句个数为 3, 那么, 调度序列的个数 $n = A_3^3 = 6$; 3 个不确定发送语句分别在进程 1∼3 中, 那么, 这 6 种调度序列可以表示为 $(1,2,3)$, $(1,3,2)$, $(2,1,3)$, $(2,3,1)$, $(3,2,1)$ 和 $(3,1,2)$. 以图 2 的调度序列为例, 此调度序列可表示为 $(1,2,3)$. 为了使消息传递并行程序按此调度序列执行, 只需把不确定接收语句 $n_5^0$, $n_6^0$ 和 $n_7^0$ 的源地址参数改为对应的进程编号 1, 2 和 3.

接着, 使用式 (4) 计算调度序列评价指标时, 设置 $\alpha = 0.6$, $\beta = 0.4$. 本文把基于调度序列随机排序集生成覆盖所有目标路径测试数据的方法称为随机方法. 选择文献 [33] 提出的并行程序多路径覆盖测试数据生成方法作为第 2 个对比方法, 相关的参数设置如下: 最大迭代次数为 1000, 种群规模为 200, 采用轮盘赌选择方法, 单点交叉概率为 0.9, 单点变异概率为 0.3, 并称此方法为传统方法.

最后, 为了排除随机因素对实验结果的影响, 所有实验重复 100 次, 并取这些实验结果的平均值进行对比.

### 4.4 实验结果与分析

#### 4.4.1 关于问题 1

为了回答第 1 个问题, 首先, 使用本文提出的方法排序调度序列集 $SS$, 得到调度序列排序集 $SS'$, 基于 $SS'$ 生成覆盖所有目标路径的测试数据; 然后, 对调度序列集 $SS$ 打乱顺序, 得到调度序列随机排序集 $SS''$, 基于 $SS''$ 生成覆盖所有目标路径的测试数据; 最后, 画出 APFD 图并计算 APFD 指标. 基于调度序列随机排序集 $SS''$ 生成测试数据的方法与基于调度序列排序集 $SS'$ 生成测试数据的方法相同, 这两种方法的 Iter = 1000. 图 5 是 9 个程序 $SS'$ 和 $SS''$ 的 APFD 值. 图 6 是这些程序 $SS'$ 和

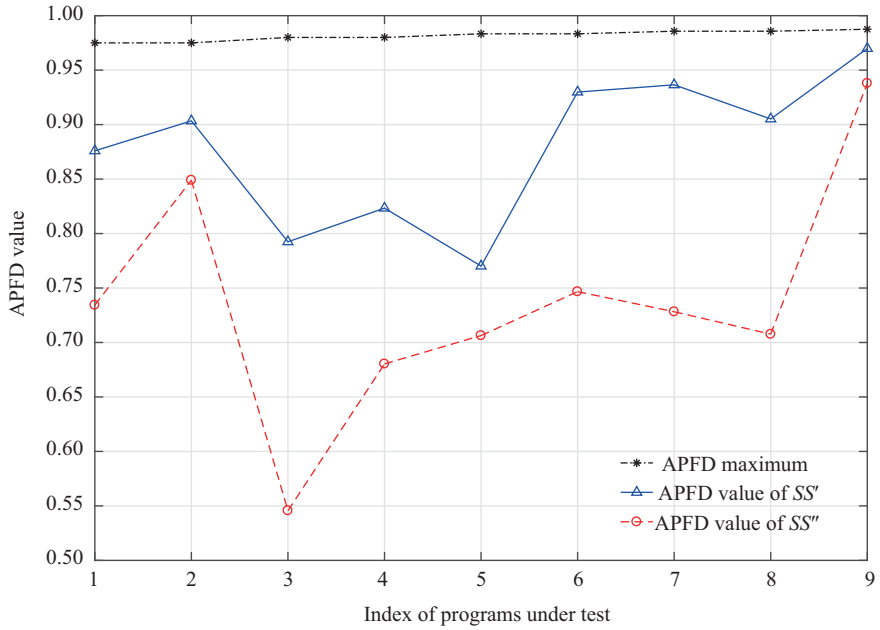


**图 5　(网络版彩图) 9 个程序的 APFD 值**

**Figure 5**　(Color online) APFD values for 9 programs

$SS''$ 的 APFD 图.

由图 5 和 6 可知:

(1) 对于 9 个程序, 基于调度序列排序集 $SS'$ 得到的 APFD 值都比基于调度序列随机排序集 $SS''$ 得到的 APFD 值高. 这说明, 相比于 $SS''$, 基于 $SS'$ 中更少的调度序列即可覆盖所有目标路径, 因此 $SS'$ 的效果比 $SS''$ 更好.

(2) 对于 T1, T3, T4, T6~T9, $SS'$ 的 APFD 图的收敛速度明显比 $SS''$ 更快, 这说明, 若要覆盖所有目标路径, 基于 $SS'$ 需要的调度序列个数明显少于基于 $SS''$ 需要的调度序列个数, 因此 $SS'$ 的效果明显优于 $SS''$. 但对于 T2 和 T5, $SS'$ 和 $SS''$ 的 APFD 图的收敛速度接近. 可能的原因是: 目标路径集中部分目标路径的覆盖难易程度与调度序列无关, 此时本文提出的排序方法与不排序时的效果是相近的.

(3) 对于 T1, T2, T4, T5, T7~T9, $SS'$ 和 $SS''$ 的 APFD 图可以收敛到最大值 1; 但对于 T3 和 T6, $SS'$ 和 $SS''$ 的 APFD 图的最大值都没有收敛到最大值 1. 这说明, 本文方法不会影响路径覆盖率.

(4) 若与理论上的 APFD 最大值比较, 对于 T1, T2, T6~T9, $SS'$ 的 APFD 值已非常接近理论最大值, 这说明基于 $SS'$ 中极少数的调度序列即可覆盖所有目标路径; 除了 T9, $SS''$ 的 APFD 值与理论最大值差距较大. 因此 $SS'$ 的效果更好.

(5) 对于 T1, T3, T4, T6~T8, $SS'$ 的 APFD 值明显高于 $SS'$ 的 APFD 值, 这说明基于 $SS'$ 中更少的调度序列即可覆盖所有目标路径. 因此 $SS'$ 的效果更好.

以上分析说明, 通过与不排序的调度序列相比, 本文提出的基于路径相似度的多路径覆盖调度序列排序方法得到的调度序列排序集的效果更好; 同时, 基于调度序列排序集中少数调度序列, 生成的测试数据即可覆盖所有目标路径, 这有助于降低多路径覆盖测试数据生成的计算消耗.

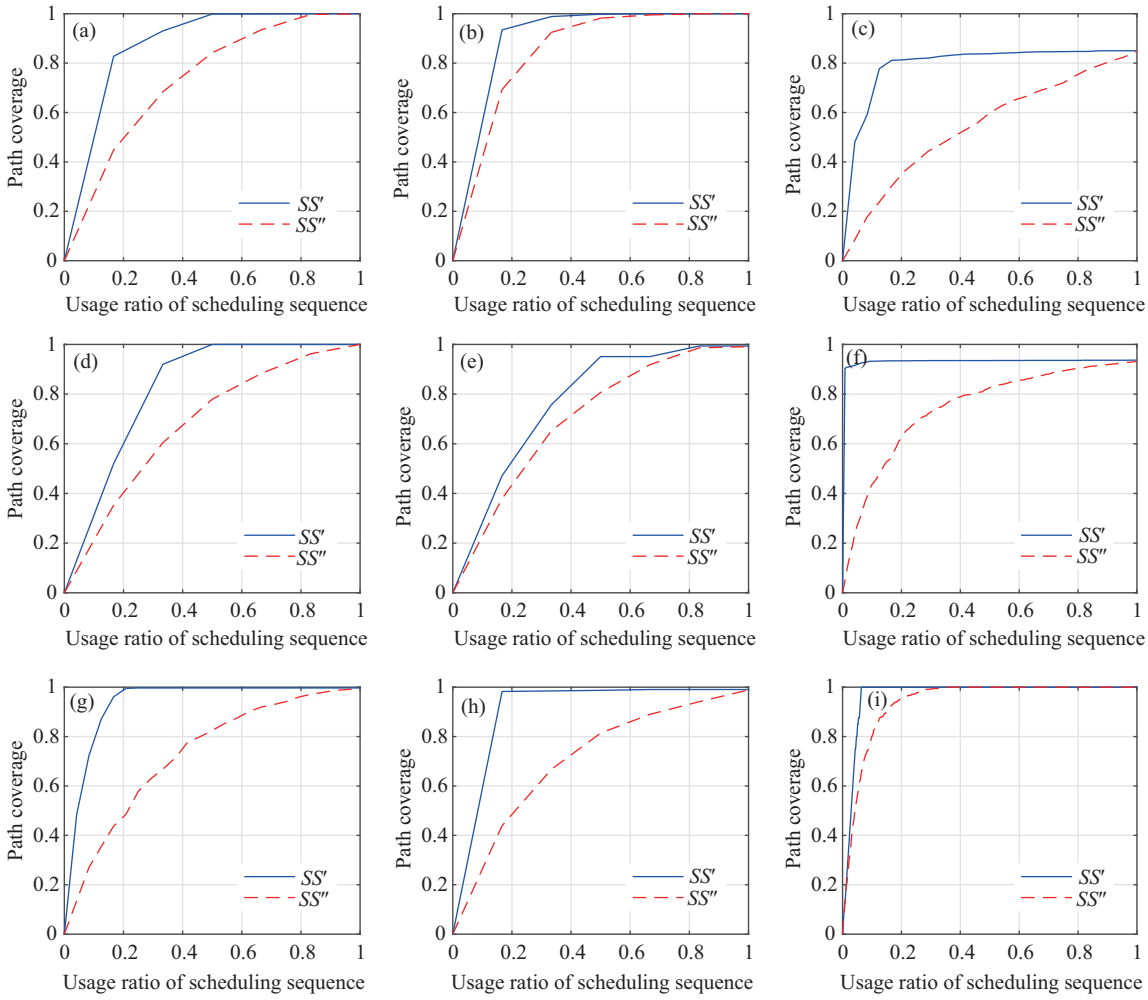


**图 6　(网络版彩图) 9 个程序的 APFD 图**

**Figure 6**　(Color online) APFD diagrams for 9 programs. (a) T1; (b) T2; (c) T3; (d) T4; (e) T5; (f) T6; (g) T7; (h) T8; (i) T9

### 4.4.2　关于问题 2

为了回答第 2 个问题, 首先, 设置 Iter = 1000, 基于调度序列排序集 $SS'$ 和调度序列随机排序集 $SS''$, 生成覆盖所有目标路径的测试数据, 记录被测程序执行次数、运行时间, 以及路径覆盖率; 接着, 使用文献 [33] 提出的方法生成覆盖所有目标路径的测试数据, 记录被测程序执行次数、运行时间, 以及路径覆盖率; 最后, 比较 3 种方法的被测程序执行次数、运行时间, 以及路径覆盖率. 被测程序执行次数越低、运行时间越少, 以及路径覆盖率越高, 那么对应方法的测试数据生成效率越高. 表 3 是本文方法与其他方法的 3 种指标对比结果.

本文采用 Mann-Whitney U 非参数假设检验方法, 确定本文方法与其他方法的差异是否显著, 显著性水平取 0.05. “+” 代表两种方法的指标差异明显, “−” 代表两种方法的指标没有显著差异. 表 4 是本文方法与其他方法在 3 个指标上的非参数假设检验结果.

由表 3 和 4 可知:

**表 3　3 种方法的指标对比**

**Table 3**　Comparison of indicators of the three methods

| ID | Number of executions | | | Running time (ms) | | | Path coverage (%) | | |
|---|---|---|---|---|---|---|---|---|---|
| | Our method | Random method | Traditional method | Our method | Random method | Traditional method | Our method | Random method | Traditional method |
| T1 | 2187.63 | 3353.94 | 67450.89 | 9.74 | 14.88 | 362.62 | 100.00 | 99.90 | 99.80 |
| T2 | 1487.18 | 2236.95 | 5558.08 | 6.43 | 9.47 | 29.80 | 100.00 | 100.00 | 100.00 |
| T3 | 23816.22 | 24000.00 | 200000.00 | 87.07 | 94.84 | 1374.49 | 85.00 | 84.96 | 76.56 |
| T4 | 2039.84 | 3943.19 | 7432.90 | 20.03 | 32.98 | 78.69 | 100.00 | 100.00 | 100.00 |
| T5 | 4517.17 | 4517.63 | 10486.54 | 18.54 | 18.91 | 61.10 | 99.37 | 99.10 | 100.00 |
| T6 | 89365.24 | 111150.90 | 182321.03 | 864.83 | 1240.81 | 1664.04 | 93.70 | 93.13 | 92.90 |
| T7 | 6254.51 | 19175.22 | 199631.86 | 27.97 | 82.45 | 1480.67 | 99.69 | 99.57 | 86.63 |
| T8 | 2895.76 | 4562.81 | 3235.73 | 11.84 | 19.69 | 17.30 | 99.06 | 98.89 | 100.00 |
| T9 | 80121.07 | 110431.28 | 190251.24 | 1144.67 | 1495.93 | 3224.13 | 100.00 | 100.00 | 100.00 |

**表 4　非参数假设检验结果**

**Table 4**　Non-parametric hypothesis test results

| ID | Number of executions | | Running time | | Path coverage | |
|---|---|---|---|---|---|---|
| | Our method and random method | Our method and traditional method | Our method and random method | Our method and traditional method | Our method and random method | Our method and traditional method |
| T1 | + | + | + | + | − | − |
| T2 | + | + | + | + | − | − |
| T3 | − | + | + | + | − | + |
| T4 | + | + | + | + | − | − |
| T5 | − | + | − | + | − | + |
| T6 | − | + | + | + | − | − |
| T7 | + | + | + | + | − | + |
| T8 | + | − | + | + | − | + |
| T9 | + | + | + | + | − | − |

(1) 对于被测程序执行次数, 本文方法比随机方法和传统方法都少. 除了 T3 和 T5, 本文方法与随机方法在数值上相差很多; 通过检验结果可以看出, 除了 T3, T5 和 T6, 本文方法与随机方法有显著差异. 这说明, 本文方法的被测程序执行次数明显比随机方法的少. 同时, 本文方法与传统方法在数值上相差很多; 通过检验结果可以看出, 除了 T8, 本文方法与传统方法有显著差异. 这说明, 本文方法的被测程序执行次数明显比传统方法的少. 因此, 本文方法在被测程序执行次数上显著优于随机方法和传统方法.

(2) 对于运行时间, 本文方法比随机方法和传统方法都少. 除了 T5, 本文方法与随机方法在数值上相差很大; 通过检验结果可以看出, 除了 T5, 本文方法与随机方法有显著差异. 这说明, 本文方法的运行时间明显比随机方法的少. 同时, 本文方法与传统方法在数值上相差很大; 通过检验结果可以看出, 本文方法与传统方法有显著差异. 这说明, 本文方法的运行时间明显比传统方法的少. 因此, 本文

方法在运行时间上显著优于随机方法和传统方法.

(3) 对于路径覆盖率, 本文方法与随机方法和传统方法相差不大. 除了 T2, T4 和 T9, 本文方法在数值上比随机方法大; 但是检验结果显示, 本文方法与随机方法无显著差异. 这说明, 本文方法与随机方法在路径覆盖率上效果相近. 同时, 对于 T1, T3, T6 和 T7, 本文方法在数值上比传统方法大; 通过检验结果可以看出, 对于 T3 和 T7, 本文方法显著优于传统方法. 对于 T5 和 T8, 本文方法在数值上比传统方法小; 通过检验结果可以看出, 对于 T5 和 T8, 传统方法显著优于本文方法. 这说明, 在路径覆盖率上, 本文方法与传统方法效果相近. 因此, 本文方法与随机方法和传统方法在路径覆盖率上差异不大.

以上分析说明, 使用本文方法生成多路径覆盖测试数据时, 被测程序执行次数和运行时间更少, 但在路径覆盖率上与其他方法差别不大. 因此, 本文方法在不降低路径覆盖率的前提下, 显著减少了被测程序执行次数和运行时间, 从而大大提高了多路径覆盖测试数据生成的效率.

综合问题 1 和 2 的分析, 可以得到如下结论: 基于排序后调度序列集合中的少数高性能调度序列, 能高效地生成覆盖多路径的测试数据, 从而从调度序列的角度大大提高了多路径覆盖测试的效率.

## 5 总结

对于包含不确定通信语句的消息传递并行程序, 本文寻找高性能调度序列, 并在少数高性能调度序列下运行程序, 从调度序列的角度提高多路径覆盖测试的效率. 为此, 本文针对消息传递并行程序多路径覆盖测试数据生成问题, 研究调度序列排序方法, 给出了多路径覆盖准则下的调度序列排序方法, 为了评估调度序列排序结果, 提出了基于调度序列排序集的多路径覆盖测试数据随机采样生成方法, 目的在于提高多路径覆盖测试数据生成的效率.

在提出的方法中, 首先, 采样程序输入并生成路径覆盖矩阵; 然后, 计算每条目标路径与路径覆盖矩阵中每条路径的相似度, 生成多个路径相似度矩阵; 接着, 基于路径相似度矩阵的特征量, 评价调度序列的性能并对调度序列排序; 最后, 基于调度序列排序集, 对程序输入空间随机采样, 生成覆盖所有目标路径的测试数据, 并使用 APFD 指标评估调度序列排序集.

本文利用 9 个具有代表性的基准并行程序, 并与随机方法和传统方法比较, 验证了所提方法的有效性. 实验结果表明: (1) 使用本文所提方法得到的调度序列排序集的 APFD 指标很高, 这表明基于调度序列排序集中少数的高性能调度序列, 生成的测试数据即可覆盖所有目标路径; (2) 相比于随机方法和传统方法, 使用本文方法生成覆盖所有目标路径的测试数据, 能够在不降低路径覆盖率的前提下, 显著减少被测程序执行次数和运行时间, 从而提高多路径覆盖测试的效率.

值得说明的是, 本文没有使用启发式方法生成测试数据, 因此测试数据生成的效率有待进一步提高. 如何基于调度序列排序集, 使用启发式方法生成多路径覆盖测试数据, 是需要进一步研究的课题.

## 参考文献


1 Beizer B. Software Testing Techniques. New Delhi: Dreamtech Press, 2002. 3

2 Xie X Y, Xu B W, Shi L, et al. Genetic test case generation for path-oriented testing. J Softw, 2009, 20: 3117–3136

3 Shan J H, Wang J, Qi Z C. Survey on path-wise automatic generation of test data. Acta Electron Sin, 2004, 32: 109–113

4 Ahmed M A, Hermadi I. GA-based multiple paths test data generator. Comput Oper Res, 2008, 35: 3107–3124

5 Chen G L. Parallel Computing — Structure, Algorithms, Programming. Beijing: Higher Education Press, 2011. 4–6 [陈国良. 并行计算 — 结构、算法、编程. 北京: 高等教育出版社, 2011. 4–6

6 Wang J X. Similarity-based testing for message passing parallel programs. Dissertation for M.S. Degree. Xuzhou: China University of Mining and Technology, 2017
7 Bianchi F A, Margara A, Pezze M. A survey of recent trends in testing concurrent software systems. IEEE Trans Softw Eng, 2018, 44: 747–783
8 Su X H, Yu Z, Wang T T, et al. A survey on exposing, detecting and avoiding concurrency bugs. Chin J Comput, 2015, 38: 2215–2233
9 Cai Y, Lu Q. Dynamic testing for deadlocks via constraints. IEEE Trans Softw Eng, 2016, 42: 825–842
10 Liu W, Wang L, Du Y Y, et al. Deadlock property analysis of concurrent programs based on Petri net structure. Int J Parallel Prog, 2017, 45: 879–898
11 Fei Y, Zhu H B, Wu X, et al. Comparative modelling and verification of Pthreads and Dthreads. J Softw Evol Proc, 2018, 30: e1919
12 Hilbrich T, de Supinski B R, Nagel W E, et al. Distributed wait state tracking for runtime MPI deadlock detection. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Denver, 2013. 1–12
13 Cai Y, Cao L W. Effective and precise dynamic detection of hidden races for Java programs. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, Bergamo, 2015. 450–461
14 Samak M, Ramanathan M K. Synthesizing tests for detecting atomicity violations. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, Bergamo, 2015. 131–142
15 Yin L Z, Dong W, Liu W W, et al. On scheduling constraint abstraction for multi-threaded program verification. IEEE Trans Softw Eng, 2018. doi: 10.1109/TSE.2018.2864122
16 Yu H B. Combining symbolic execution and model checking to verify MPI programs. In: Proceedings of the 40th International Conference on Software Engineering, Gothenburg, 2018. 527–530
17 Wang R, Ding Z H, Gui N, et al. Detecting bugs of concurrent programs with program invariants. IEEE Trans Rel, 2017, 66: 425–439
18 Wu X Q, Wei J. Visual debugging concurrent programs with event structure. J Softw, 2014, 25: 457–471
19 Qi X F, Xu X J, Jiang Z L, et al. Slicing concurrent programs based on program reachability graphs with partial-order reduction. Chin J Comput, 2014, 37: 568–579
20 Qi X F, He J, Wang P, et al. Variable strength combinatorial testing of concurrent programs. Front Comput Sci, 2016, 10: 631–643
21 Liao W Z. Test data generation based on automatic division of path. Acta Electron Sin, 2016, 44: 2254–2261
22 Liu F Q, Huang H, Hao Z F. Evolutionary algorithm with convergence speed controller for automated software test data generation problem. In: Proceedings of IEEE Congress on Evolutionary Computation, San Sebastian, 2017. 869–875
23 Huang H, Liu F Q, Zhuo X Y, et al. Differential evolution based on self-adaptive fitness function for automated test case generation. IEEE Comput Intell Mag, 2017, 12: 46–55
24 Panichella A, Kifetew F M, Tonella P. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. IEEE Trans Softw Eng, 2018, 44: 122–158
25 Arrieta A, Wang S, Markiegi U, et al. Employing multi-objective search to enhance reactive test case generation and prioritization for testing industrial cyber-physical systems. IEEE Trans Ind Inf, 2018, 14: 1055–1066
26 Hwang G H, Lin H Y, Lin S Y, et al. Statement-coverage testing for concurrent programs in reachability testing. J Inf Sci Eng, 2014, 30: 1095–1113
27 Ding Z H, Zhang K, Hu J L. A rigorous approach towards test case generation. Inf Sci, 2008, 178: 4057–4079
28 Souza S R S, Souza P S L, Brito M A S, et al. Empirical evaluation of a new composite approach to the coverage criteria and reachability testing of concurrent programs. Softw Test Verif Reliab, 2015, 25: 310–332
29 Tian T, Gong D W. Model of test data generation for path coverage of message-passing parallel programs and its evolution-based solution. Chin J Comput, 2013, 36: 2212–2223
30 Gong D W, Zhang C, Tian T, et al. Reducing scheduling sequences of message-passing parallel programs. Inf Softw Tech, 2016, 80: 217–230
31 Siegel S F, Zirkel T K. FEVS: a functional equivalence verification suite for high-performance scientific computing. Math Comput Sci, 2011, 5: 427–435

32 Lv X W, Huang S, Hui Z W, et al. Test cases generation for multiple paths based on PSO algorithm with metamorphic relations. IET Softw, 2018, 12: 306–317
33 Tian T, Gong D W. Evolutionary generation approach of test data for multiple paths coverage of message-passing parallel programs. Chin J Electron, 2014, 23: 291–296

# Path similarity-based scheduling sequence sorting for multi-path coverage of parallel programs

Feng PAN[1], Dunwei GONG[1*], Tian TIAN[2], Xiangjuan YAO[3] & Yin LI[4]

1. *School of Information and Control Engineering, China University of Mining and Technology, Xuzhou* 221116*, China*;
2. *School of Computer Science and Technology, Shandong Jianzhu University, Jinan* 251100*, China*;
3. *School of Mathematics, China University of Mining and Technology, Xuzhou* 221116*, China*;
4. *Jiangsu Automation Research Institute, Lianyungang* 222061*, China*
* Corresponding author. E-mail: dwgong@vip.163.com

**Abstract** Testing is an important way of improving the reliability of a software product. Uncertain communication statements in the message-passing parallel program can lead to uncertainty in the order of the execution of processes, which increases the difficulty of testing. Taking into consideration the influence of an uncertain process-execution order on the difficulty of covering target paths, in this paper, we focus on a method for sorting scheduling sequences in a message-passing parallel program to enable multi-path coverage and improve the efficiency of generating test data for multi-path coverage. To fulfill this task, we first execute a parallel program on each sample in each scheduling sequence to generate a path coverage matrix. Then, we produce a number of path similarity matrixes based on the similarity of each path in the coverage matrix and for each target. Next, we evaluate each scheduling sequence with respect to the characteristics of the similarity matrixes, based on which we sort the scheduling sequences. Finally, we generate test data for multi-path coverage by performing a random sampling of the sorted scheduling sequence set, which are evaluated using the APFD index. We applied the proposed method to nine benchmark parallel programs, and compared the performances of the random and traditional methods. The experimental results indicate that the proposed method has no effect on path coverage, but significantly reduces the number of program executions during the test and running time.

**Keywords** message-passing parallel program, multi-path coverage, testing, scheduling sequence sorting, path similarity

**Feng PAN** was born in 1995. He received a B.S. degree from Xuzhou University of Technology in 2013. Currently, he is an M.S. candidate in control science and engineering at the School of Information and Control Engineering, China University of Mining and Technology. His main research area is software testing.

**Dunwei GONG** was born in 1970. He received his Ph.D. degree from China University of Mining and Technology. Currently, he is a professor at China University of Mining and Technology. His main research areas are search-based software engineering and intelligent optimization.

**Tian TIAN** was born in 1987. She received her Ph.D. degree from China University of Mining and Technology. Currently, she is an associate professor at Shandong Jianzhu University. Her research interest mainly focuses on testing of parallel software.

**Xiangjuan YAO** was born in 1975. She received her Ph.D. degree from China University of Mining and Technology. Currently, she is a professor at China University of Mining and Technology. Her research interest includes search-based software engineering.