基于分组的并行程序多路径覆盖测试数据进化生成

田 甜，巩敦卫
（中国矿业大学信息与电气工程学院，江苏徐州 221116）

摘 要：尽管并行软件测试已经得到软件工程界的广泛关注，但是，如何高效生成覆盖并行软件多条路径的测试数据，相关的研究还比较少。本文研究消息传递并行程序多路径覆盖测试数据生成问题，并提出基于分组的测试数据进化生成方法。首先根据并行程序包含的进程数、可用的计算资源以及路径相似度，将目标路径分成若干组，并基于每组目标路径，建立多路径覆盖测试数据生成问题的数学模型；然后采用多种群并行遗传算法求解上述模型，使得一次运行遗传算法，生成覆盖所有目标路径的测试数据。性能分析表明，所提出的目标路径分组方法不但能够保证不同组包含的目标路径相差很少，而且同一组的目标路径之间具有很大的相似度。将所提方法应用于4个基准程序的测试中，并与已有方法比较，结果表明，所提方法在保证路径覆盖率的前提下，可大大缩减个体评价次数和耗时。

关键词：软件测试；并行程序；路径覆盖；测试数据；遗传算法
中图分类号：TP201 **文献标志码**：A **文章编号**：2095-2783(2014)04-0441-10

Evolutionary generation of test data for multi-paths coverage of parallel programs by grouping

Tian Tian, Gong Dunwei

(*School of Information and Electrical Engineering, China University of Mining and Technology, Xuzhou, Jiangsu* 221116, *China*)

Abstract: Parallel software testing has attracted wide-ranging attention in the community of software engineering. However, less research is concerned with effectively generating test data for multi-paths coverage of parallel programs. We investigated the problem of generating test data for multi-paths coverage of message-passing parallel programs and proposed a grouping-based method of evolutionarily generating test data. First, target paths were divided into several groups according to the number of processes in a parallel program, available computation resources, and the similarities of target paths, and the mathematical model of generating test data for multi-paths coverage was built based on the target paths belonging to each group. Secondly, a multi-population parallel genetic algorithm was employed to solve the above model, so that the test data covering all target paths could be generated in one run of the genetic algorithm. Performance analysis indicates that the proposed method guarantees not only a small difference in the number of target paths belonging to different groups, but also a great similarity among target paths in the same group. The proposed method was applied to four benchmark programs, and compared with existing methods. The experimental results show that the proposed method greatly reduces the number of evaluated individuals and the consumption time on the promise of meeting the path coverage rate.

Key words: software testing; parallel program; path coverage; test data; genetic algorithm

在国民经济和社会发展中，存在很多需要高效解决的复杂问题。以专用高性能并行机或普通集群系统作为硬件平台，并行计算能够大幅度缩短问题求解时间，并提高求解精度，从而成为主流的问题解决方式。当前，并行计算硬件系统的性能不断提高，运算峰值速度每秒高达几百万甚至上千万亿次，处理器数量多达数十万个[1]，不仅对并行计算硬件系统，而且对该系统上运行并行程序的可靠性，都提出了很高的要求[2]。目前，主要有3种开发并行程序的方式，分别为：①自动并行化已有的串行程序；②使用专门的并行编程语言重新编写并行程序；③基于消息传递环境扩展串行程序[3]。实践表明，采用第①种方法开发的并行程序效率很低，并且涉及到复杂的编译技术；采用第②种方法需要对已有的串行代码做彻底的修改，从头开始编写全新的并行程序，对开发者带来繁重的编程负担；采用第③种方法，即通过使用消息传递环境扩展已有的串行程序，能够减少额外编程负担，并提高并行程序开发的效率，所以该方法成为最常用的并行程序开发方式[4]。典型的消息传递环境有消息传递接口(message passing interface, MPI)[5]，并行虚拟机(parallel virtual machine, PVM)[6]，CM信息传递库以及Express等。其中，MPI有很好的兼容性，且有多个免费实现版本，是目前最重要和最流行的消息传递库。鉴于此，本文研

收稿日期：2014-01-20
基金项目：国家自然科学基金资助项目(61075061, 61203304, 61375067)；高等学校博士学科点专项科研基金资助项目(20100095110006)；江苏省普通高校研究生科研创新计划项目(CXZZ11_0292)
作者简介：田甜(1987—)，女，博士研究生，主要研究方向为并行软件测试
通信联系人：巩敦卫，教授，主要研究方向为智能优化与控制、基于搜索的软件工程，dwgong@vip.163.com

究基于MPI扩展的消息传递并行程序。

到目前为止，大部分针对并行程序测试的研究工作都侧重于检测死锁、资源竞争以及应用程序接口(application program interface，API)是否得到正确使用等。这些问题固然是影响并行程序顺利执行的重要因素，但只考虑这些因素，不能保证并行程序的可靠性和正确性。在通信正确的前提下，即不存在死锁和资源竞争等问题时，如何进行软件测试，以提高并行软件的可信度，是亟需解决的问题。

进行软件测试的核心，是生成有效的测试数据。目前已有很多测试数据生成标准，例如语句覆盖、分支覆盖、条件覆盖、判定-条件覆盖以及全引用覆盖等，且都能够归结为路径覆盖测试数据生成问题。此外，组装测试中的调用对覆盖，以及数据流测试、面向断言的测试和回归测试中的一些问题，也可以归结为路径覆盖问题。因此，路径覆盖成为最常用的测试充分性准则之一[7]。

针对多路径覆盖测试数据生成问题，基于路径相似度对目标路径分组，并基于每一组目标路径，在形成一个子优化问题之后，利用多种群并行遗传算法，使得每个子种群优化求解一个子优化问题，以生成覆盖一组目标路径的测试数据[8]。已有研究表明，上述方法能够高效生成覆盖多条目标路径的测试数据。但该方法基于串行环境实现，不但没有考虑并行计算资源的限制，也没有考虑并行程序多个进程执行的并行性，因此难以高效生成覆盖并行程序多条路径的测试数据。

鉴于此，本文研究并行程序多路径覆盖测试数据生成问题。首先，提出适用于并行程序的目标路径分组方法；然后，基于分组后的目标路径，建立多路径覆盖测试数据生成问题的数学模型；最后，给出用于求解上述模型的多种群并行遗传算法。将所提方法应用于4个基准并行程序的测试中，与已有方法的比较结果表明，本文方法能够高效生成覆盖多条目标路径的测试数据。

1 相关研究工作

1.1 并行程序测试

近年来，并行程序测试逐步得到软件工程界的广泛重视，并取得了一些有价值的研究成果。文献[9]在程序运行过程中，检查程序是否正确使用了MPI的应用程序接口。文献[10]实现了MPI程序的监视工具Umpire，并用于动态监测死锁、不匹配集合操作以及资源耗尽等程序缺陷。文献[11]开发了消息竞争检测工具MPI_Check，通过检查进程间的并发通信，检测并报告所有的竞争条件、与消息竞争有关的代码行、进程号以及消息等[11]。文献[12]研究了可达性测试，并提出一种有效的测试方法，能够保证在不保存已执行序列的前提下，使每个偏序同步序列只执行一次。进一步，文献[13]提出了分布式可达性测试方法，使多个测试序列同时执行，大大减少了执行时间。为了降低测试代价，文献[14]将覆盖准则与可达性测试相结合，依据某一特定的覆盖准则选择测试数据，引导新同步序列的执行，并利用可达性测试，选择合适的同步序列。上述方法都以程序的实际执行为前提，因此，可以归结为并行程序的动态测试。

与动态运行程序不同的是，静态方法不需要程序运行时的信息，可避免程序运行过程中不确定性因素对并行程序测试的影响。文献[15]使用进程依赖网描述Ada并行程序中的依赖关系，并提出并发、同步以及通信依赖的概念。文献[16]通过分析程序代码收集必要的信息，并构建程序的通信图，以检测死锁、竞争条件等消息传递错误。文献[17]采用流敏感的过程间分析方法，检测数据竞争和死锁，并针对大规模复杂并发系统，开发了检测工具RacerX。作为一种重要的静态方法，模型检测方法存在状态空间爆炸问题。为了解决空间爆炸问题，文献[18]提出了动态偏序缩减方法。在此基础上，文献[19]开发了MPI程序的模型检测工具ISP，以检测程序的死锁和局部断言违例[19]。

另外，为了融合静态和动态方法的优势，文献[20]还提出了混合测试方法。该方法首先利用静态分析获得同步或共享变量存取的简要信息；然后，在程序执行过程中，利用这些信息预测没有覆盖分支的行为。

可以看出，无论静态、动态还是混合方法，主要针对并行程序的通信序列进行测试，以发现并发执行导致的死锁、资源竞争以及API使用等问题。鉴于通信是并行程序的重要特征之一，因此，这些研究在一定程度上为并行程序的顺利执行提供了保障。为了测试并行程序的其他元素，以进一步提高并行程序的可靠性，诸多学者借鉴串行程序的测试标准，提出了用于并行程序测试的覆盖准则。

文献[21]针对共享存储并行程序，通过扩展串行程序的测试标准，提出了相应的覆盖准则。文献[22]还将all-du-path覆盖准则应用于共享存储或消息传递并行程序中，并提出了在被测程序中寻找满足条件路径的方法。文献[23]针对消息传递并行程序，分别基于控制和通信流，以及数据和消息传递流，提出了发送节点、接收节点、定义、定义-发送等覆盖准则。进一步，文献[24]在已有工作的基础上，考虑被测程序的集合通信、非阻塞通信等特点，提出了新的覆盖准则。但遗憾的是，它们没有建立测试数据生成问题的数学模型，更没有提出切实可行的测试数据生成方法。

鉴于此，本文研究了高效的并行程序路径覆盖测试数据生成方法，与已有工作具有明显的不同，主要体现在：①本文的重点不在于描述各种覆盖准则，而在于测试数据生成方法；②本文不在于给出目标

路径的选择方法,而在于如何生成覆盖多条给定目标路径的测试数据;③本文不在于考察不确定性对并行程序执行结果的影响,而在于给定进程调度序列下的路径覆盖测试数据生成。

1.2　测试数据生成

到目前为止,学者们已经提出了多种串行程序的测试数据生成方法,总体上可以分为随机法、静态法、动态法以及试探法 4 类。其中,随机法通过对被测程序的输入空间的随机采样,生成覆盖目标路径的测试数据[25]。该方法开销小且简单易行,但具有很大的盲目性。静态法仅通过对程序进行静态的分析和转换,生成满足要求的测试数据,不涉及程序的实际运行[26]。该方法占用的存储空间大,且无法处理输入空间为无穷区间的软件测试。动态法基于程序的实际运行,生成测试数据的过程是确定的,典型的包括直线式程序法[27]、改变变量法[28]以及迭代松弛法[29]。试探法也基于程序的实际运行,但是生成测试数据的过程是不确定的,如基于遗传算法的测试数据生成方法。

采用遗传算法生成复杂软件的测试数据是近年来软件工程界的研究热点之一。文献[30]首先将遗传算法应用于软件测试数据的生成,解决路径覆盖问题。文献[31]和[32]等使用遗传算法,获得满足路径覆盖的测试数据。文献[33]提出了扩展的海明距离,解决含有相同节点,但顺序不同的路径比较问题。文献[34]提出了 3 种基于路径相似程度的适应度函数构造方法,用于指导覆盖目标路径的测试数据生成。文献[35]将搜索算法应用于面向对象软件的测试数据生成,研究测试序列长度对分支覆盖的影响。此外,文献[36]给出控制测试序列长度爆炸的方法。文献[37]通过同时寻找覆盖多条目标路径的测试数据,提高测试数据生成的效率。针对覆盖多条目标路径的测试数据生成问题,文献[8]和[38]从路径编码和分组的角度,基于遗传算法,给出了有效的测试数据生成方法。

但是,上述测试数据生成方法主要适用于串行程序。与串行程序不同的是,并行程序具有并发、通信以及同步等特性,而上述方法并没有考虑这些特性。如果将上述方法直接应用于并行程序的测试,测试数据生成的效率将大大降低。因此,根据并行程序自身的特性,研究高效的测试数据生成方法是非常必要的。

虽然文献[39]对并行程序路径覆盖测试数据生成问题进行了初步研究,并提出了基于遗传算法的测试数据生成方法。但是,该方法运行一次遗传算法,仅能生成覆盖一条目标路径的测试数据,当有多条目标路径时,测试数据生成效率将大大降低。为此,本文进一步考虑了并行程序多路径覆盖测试数据生成方法。具体地讲,将多路径覆盖测试数据生成问题转化为一个多目标优化问题,并通过遗传算法求解该问题,运行一次遗传算法,即能够同时生成覆盖多条目标路径的测试数据。但是,当目标路径很多时,上述转化后的优化问题将包含很多目标函数,导致问题的求解将非常复杂。如果采用已有的进化多目标优化方法求解,将难以甚至不可能得到满足路径覆盖要求的测试数据。

鉴于此,本文研究消息传递并行程序多路径覆盖测试数据生成问题。前已述及,对于串行程序,通过对目标路径分组,能够把一个复杂的多路径覆盖测试数据生成问题转化为多个相对容易求解的子问题,并通过对每一个子问题的求解,生成覆盖所有目标路径的测试数据。

上述方法为多路径覆盖测试数据生成提供了一种解决方案,可提高测试数据生成的效率。但是,该方法并不能直接用于求解并行程序路径覆盖问题。这是因为:①该方法没有考虑并行程序包含的多个进程。由于一个并行程序包含多个进程,因此运行该程序将需要多个计算资源,使得目标路径分组方法应该考虑进程对计算资源的需求。但已有方法没有考虑上述因素。②该方法没有考虑并行程序路径相似度的计算。鉴于并行程序包含多个进程,计算并行程序两条路径的相似度时,应该考虑路径对于不同进程子路径的相似性。但已有方法没有考虑进程对计算并行程序路径相似度的影响。

此外,上述方法基于单机串行环境,求解串行程序路径覆盖测试数据生成问题,还存在如下不足之处:①对于相同的目标路径(集),分组个数与设定的相似度阈值密切相关,相似度阈值越大,得到的分组个数越多。但如何设置合适的相似度阈值,至今尚没有有效的方法。②不同组包含的目标路径条数相差很大,使得生成覆盖不同组目标路径的测试数据需要的计算量相差也很大。所以,采用多种群并行遗传算法生成路径覆盖测试数据的效率有待进一步提高。

基于此,本文针对并行程序多进程并行执行的特点,考虑计算资源负载的均衡,研究用于并行程序的目标路径分组方法,并在此基础上,建立路径覆盖测试数据生成问题的数学模型,采用多种群并行遗传算法求解该模型,以生成期望的测试数据。

2　目标路径分组

2.1　基本概念

记并行程序为 S,由 m 个进程组成 $m>1$,第 i 个进程为 S^i,$i=0,1,\cdots,m-1$。该 m 个进程并行执行,协同完成一个计算任务,S 可以表示为 $S=\{S^0,S^1,\cdots,S^{m-1}\}$。记程序 S 的输入向量为 $\boldsymbol{x}=\{x_1,x_2,\cdots,x_{n_s}\}$,其中,$x_j$ 为第 j 个输入分量,$j=1,2,\cdots,n_s$,n_s 为输入分量的个数。如果 x_j 的取值范围为 D_j,那么,$\boldsymbol{x}$ 的取值范围为 $D=D_1\times D_2\times\cdots\times D_{n_s}$。

1)节点:考虑程序 S 的第 i 个进程 S^i,其基本执

行单元称为一个节点，使得该节点包含的语句要么都被执行，要么都不被执行。节点可以是一个分支语句的判断条件，也可以是一个循环语句的循环条件，还可以是一条或多条连续语句，或者一条消息发送或接收语句。记 S^i 的第 k 个节点为 n_k^i。如果 n_k^i 是一条消息发送(接收)语句，则称 n_k^i 为消息发送(接收)节点。

2)路径：以输入 $\boldsymbol{x} \in D$ 执行程序 S，$\boldsymbol{x}$ 穿越的节点序列形成一条路径，记为 $p(\boldsymbol{x}) = p^0 p^1 \cdots p^{m-1}$，其中，$p^i$ 为进程 S^i 的节点序列形成的子路径，$|p^i|$ 为子路径 p^i 的长度。

考虑程序 S 的 n 条目标路径 $p_1, p_2, \cdots, p_n$，覆盖这些目标路径的测试数据生成问题可以建模为如下多目标优化问题，即有

$$\max(f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \cdots, f_n(\boldsymbol{x})),$$
$$\text{s.t. } \boldsymbol{x} \in D。 \tag{1}$$

式中，$f_l(\boldsymbol{x})$ 是与第 l 个目标路径相关的目标函数。该函数取得最大值的充要条件是，以 $\boldsymbol{x}$ 作为输入运行程序 S 时穿越的路径恰是第 l 个目标路径。容易看出，式(1)表示的优化问题中，目标函数的个数与目标路径相等。因此，当需要覆盖的目标路径很多时，式(1)将包含很多目标函数，将导致对式(1)的求解非常复杂，说明采用合适的方法，进一步简化本文考虑的测试数据生成问题，是非常必要的。

2.2 路径分组方法

本文将重点阐述用于并行程序的目标路径分组

$$|g_k| = \begin{cases} n/|r/m|+1, & k = 1,2,\cdots,n \bmod |r/m|; \\ n/|r/m|, & k = n \bmod |r/m|+1, n \bmod |r/m|+2, \cdots, |r/m|。 \end{cases} \tag{5}$$

最后，根据目标路径的相似度，确定每组目标路径的构成。为使每组包含的目标路径具有很大的相似度，可根据式(3)，建立反映目标路径相似程度的矩阵，记为 $\boldsymbol{M}(p_1, p_2, \cdots, p_n)$，其表达式为

$$\boldsymbol{M}(p_1, p_2, \cdots, p_n) = \begin{bmatrix} s(p_1,p_1) & s(p_1,p_2) & \cdots & s(p_1,p_n) \\ s(p_2,p_1) & s(p_2,p_2) & \cdots & s(p_2,p_n) \\ \vdots & & & \vdots \\ s(p_n,p_1) & s(p_n,p_2) & \cdots & s(p_n,p_n) \end{bmatrix}。 \tag{6}$$

为了刻画某路径与其他路径的相似度，计算式(6)矩阵每行元素的和。第 k 行元素的和为 $\sum_{k=1}^{n} s(p_j, p_k)$。容易看出，某路径与其他路径的相似度越大，$\sum_{k=1}^{n} s(p_j, p_k)$ 越大。

为了确定 g_1 包含的目标路径，首先选择基准路径，记为 $p_1^b = \arg \max_{j=1,2,\cdots,n} \sum_{k=1}^{n} s(p_j^i, p_k^i)$；然后将所有路径(包含 p_1^b)与 p_1^b 的相似度按照降序排列，并选择与前 $|g_1|$ 个相似度对应的目标路径形成 g_1，且从路径 $p_1, p_2, \cdots, p_n$ 中删除这些路径；最后根据减少的方法。

首先给出路径相似度的度量。对于程序 S 的两条路径 $p_j = p_j^0 p_j^1 \cdots p_j^{m-1}$ 和 $p_k = p_k^0 p_k^1 \cdots p_k^{m-1}$，其中，$p_j^i$ 与 p_k^i 分别为路径 p_j 与 p_k 属于进程 S^i 的子路径，p_j^i 与 p_k^i 的相似度记为 $s(p_j^i, p_k^i)$，可以表示为

$$s(p_j^i, p_k^i) = \frac{|p_j^i \cap p_k^i|}{\max\{|p_j^i|, |p_k^i|\}}。 \tag{2}$$

式中，$p_j^i \cap p_k^i$ 表示 p_j^i 与 p_k^i 从前到后连续相同的节点。考虑程序 S 包含的所有进程，路径 p_j 和 p_k 的相似度可以表示为

$$s(p_j, p_k) = \frac{1}{m} \sum_{i=0}^{m-1} s(p_j^i, p_k^i)。 \tag{3}$$

由式(3)可以看出，对于并行程序而言，两条路径的相似度由该程序包含的所有进程的子路径的相似度决定，因此与串行程序路径相似度的度量有很大差别。

然后，根据可用的计算资源和并行程序包含的进程数，可确定目标路径分组个数和每组包含的目标路径条数。记可用的计算资源数为 r，$r > m$，鉴于程序 S 包含 m 个进程，为保证并行程序的顺利执行，将每 m 个计算资源组成一个计算资源组。为使每组计算资源的负载大体均衡，将 n 条目标路径分为 $|r/m|$ 组，记为 $g_1, g_2, \cdots, g_{|r/m|}$。

如果 $n \bmod |r/m| = 0$，g_k 包含的目标路径条数记为 $|g_k|$，可以表示为

$$|g_k| = n/|r/m|, \quad k = 1,2,\cdots,|r/m|。 \tag{4}$$

否则有：

待分组路径和式(6)得到一个 $n - |g_1|$ 阶相似矩阵。采用与构成 g_1 相同的方法，得到 $g_2, g_3, \cdots, g_{|r/m|}$ 包含的目标路径。记组 g_k 包含的路径为

$$g_k = \{p_{k1}, p_{k2}, \cdots, p_{k|g_k|}\}, k = 1,2,\cdots,|r/m|。$$

3 测试数据生成问题的数学模型

将所有目标路径分成 $|r/m|$ 组之后，基于每组目标路径，形成一个多目标优化问题，从而将原来的含有 n 个目标函数的优化问题(如式(1)所示)转化为 $|r/m|$ 个子优化问题，且第 k 个子优化问题包含 $|g_k|$ 个目标函数，对应于第 k 组目标路径覆盖的测试数据生成。为便于说明，记 g_k 包含的目标路径对应的目标函数为 $f_{k1}(\boldsymbol{x}), f_{k2}(\boldsymbol{x}), \cdots, f_{k|g_k|}(\boldsymbol{x})$，根据式(1)，该子优化问题可以表示为

$$\max(f_{k1}(\boldsymbol{x}), f_{k2}(\boldsymbol{x}), \cdots, f_{k|g_k|}(\boldsymbol{x})),$$
$$\text{s.t. } \boldsymbol{x} \in D。 \tag{7}$$

式中，目标函数 $f_{kj}(\boldsymbol{x})$ 与组 g_k 的第 j 条目标路径对应，$j = 1,2,\cdots,|g_k|$，表示 $\boldsymbol{x}$ 穿越的路径 $p(\boldsymbol{x})$ 与目标路径 p_{kj} 的相似度。根据式(2)，$f_{kj}(\boldsymbol{x})$ 可以表示为

$$f_{kj}(\boldsymbol{x})=\frac{1}{m}\sum_{i=0}^{m-1}s(p^i,p_{kj}^i)。\tag{8}$$

式中，p_{kj}^i 为路径 p_{kj} 属于进程 S^i 的子路径，$s(p^i,p_{kj}^i)$ 如式(2)所示。

将目标路径分组之后，多路径覆盖测试数据生成问题的数学模型可以表示为：

$$\begin{cases}\max(f_{11}(\boldsymbol{x}),f_{12}(\boldsymbol{x}),\cdots,f_{1|g_1|}(\boldsymbol{x})),\text{s.t.}\ \boldsymbol{x}\in D;\\ \max(f_{21}(\boldsymbol{x}),f_{22}(x),\cdots,f_{2|g_2|}(\boldsymbol{x})),\text{s.t.}\ \boldsymbol{x}\in D。\\ \vdots\\ \max(f_{|r/m|1}(\boldsymbol{x}),f_{|r/m|2}(\boldsymbol{x}),\cdots,f_{|r/m||g_{|r/m|}|}(\boldsymbol{x})),\\ \text{s.t.}\ \boldsymbol{x}\in D。\end{cases}\tag{9}$$

现在考察式(9)与文献[8]所得模型的关系。

首先，二者均基于分组得到多路径覆盖测试数据生成问题的数学模型，都将一个含有很多目标的优化问题转化为多个含有较少目标的优化问题，都从一定程度上简化了问题求解的难度，提高了测试数据生成的效率。

其次，二者又有很大区别。除了文献[8]和式(9)分别针对串行程序和并行程序之外，二者的区别还体现在如下两个方面。

1)转化后的子优化问题的个数不同。文献[8]中，子优化问题的个数不是事先确定的，而是根据相似度阈值和基准路径得到的。因此，子优化问题的个数依赖于阈值和基准路径的选择。由于基准路径是随机选择的，而阈值的选取又没有规则可循，因此，如果阈值和基准路径选取不合适，将导致子优化问题偏多或偏少，不利于问题的求解。式(9)中，子优化问题的个数由可用的计算资源和程序包含的进程数决定。对于相同的并行程序，计算资源越多，形成的子优化问题越多。这样分组能够充分利用每一个计算资源，提高问题求解的效率。

2)不同子优化问题包含的目标函数个数不同。文献[8]中，子优化问题包含的目标函数个数由相似度阈值和基准路径共同决定，即使有相同的阈值，不同的基准路径也将得到不同的分组。这意味着，文献[8]中不同子优化问题包含的目标函数个数有可能相差很大。如果采用同构的计算资源求解这些子优化问题，需要的时间将相差很大。但是，式(9)中，不同子优化问题包含的目标函数个数相差不超过 1，因此，采用同构的计算资源求解这些子优化问题，需要的时间将相差无几，有助于提高测试数据生成的效率。

4　基于多种群并行遗传算法的测试数据生成

为给出基于多种群并行遗传算法的测试数据生成方法，首先要说明使用遗传算法生成覆盖目标路径的测试数据的一般过程。

应用遗传算法生成测试数据时，需要采用随机或其他方法生成一个初始种群。为了评价进化个体的性能，需要设计合适的适应度函数。评价进化个体的性能时，需要解码该个体，并作为程序的输入以执行被测程序。如果某一测试数据满足预先设定的路径覆盖准则，则算法结束；否则，算法开始如下的迭代过程，即对当前种群实施选择、交叉以及变异等遗传操作，生成新的种群，并采用与评价父代种群的个体相同的方法，对新种群的个体性能进行评价，直到满足种群进化终止准则。进化终止准则是找到满足路径覆盖要求的测试数据，或者达到最大迭代次数。可以看出，种群初始化、适应度函数设计、遗传操作实施以及测试数据编码等，是影响遗传算法性能的几个重要方面。

对于式(9)的 $|r/m|$ 个子问题，本文采用多种群并行遗传算法求解，以生成期望的测试数据。鉴于每个子种群通过进化求解一个子优化问题，因此，并行进化的子种群共有 $|r/m|$ 个。对于每一子种群，本文采用随机方法生成初始种群。此外，采用二进制编码、轮盘赌选择、单点交叉以及单点变异等遗传操作。

考虑用于优化第 k 个子问题的子种群，在不引起混淆的情况下，记测试数据 $\boldsymbol{x}$ 对应该种群个体的适应度函数为 $F_k(\boldsymbol{x})$，则 $F_k(\boldsymbol{x})$ 可以表示为

$$F_k(\boldsymbol{x})=\max\{f_{k1}(\boldsymbol{x}),f_{k2}(\boldsymbol{x}),\cdots,f_{k|g_k|}(\boldsymbol{x})\}。\tag{10}$$

$F_k(\boldsymbol{x})$ 的值越大，说明 $\boldsymbol{x}$ 穿越的路径与组 g_k 的某一条目标路径的相似度越大，$\boldsymbol{x}$ 的性能越好。特别地，当 $\boldsymbol{x}$ 的适应值为 $F_k(\boldsymbol{x})=f_{kj}(\boldsymbol{x})=1$ 时，$\boldsymbol{x}$ 即为覆盖目标路径 p_{kj} 的测试数据。此时，为了高效生成覆盖其他路径的测试数据，将函数 $f_{kj}(\boldsymbol{x})$ 从 $F_k(\boldsymbol{x})$ 中删除，以简化 $F_k(\boldsymbol{x})$。简化后的适应度函数为

$$F_k(\boldsymbol{x})=\max\{f_{k1}(\boldsymbol{x}),\cdots,f_{kj-1}(\boldsymbol{x}),f_{kj+1}(\boldsymbol{x}),\cdots,f_{k|g_k|}(\boldsymbol{x})\}。\tag{11}$$

容易知道，随着期望的测试数据不断生成，适应度函数不断简化，评价个体所需的计算量也不断降低。

本文提出的测试数据生成方法的步骤如下：

步骤 1　按照目标路径分组方法，将 n 条目标路径分成 $|r/m|$ 组，插桩被测程序。

步骤 2　设定算法需要的控制参数值，初始化 $|r/m|$ 个子种群。

步骤 3　判断是否满足种群进化终止条件，若满足，转步骤 7。

步骤 4　解码进化个体，执行被测程序，并判断是否有测试数据穿越目标路径。若有，保存测试数据和被穿越的目标路径，并根据式(11)简化适应度函数。

步骤 5　根据式(10)计算个体适应值。

步骤 6　实施遗传操作，生成子代种群，转步骤 3。

步骤 7　形成并输出测试数据，输出被覆盖的目标路径。

5 性能分析

为了评价本文方法的性能，首先说明采用本文的目标路径分组方法能够使不同组包含的目标路径条数相差很小；然后说明选择目标路径构造不同组的合理性。

由式(4)和(5)可知，每组包含的路径条数最多为 $n/\lfloor r/m \rfloor+1$，最少为 $n/\lfloor r/m \rfloor$。也就是说，任何两组包含的目标路径条数相差均不超过 1。意味着采用本文的目标路径分组方法能够使计算资源的负载大体均等。

为了说明采用本文方法选择目标路径构造不同组的合理性，不失一般性，考察组 g_1 的构造过程。根据路径相似度的定义和相似矩阵，$\sum_{k=1}^{n} s(p_j, p_k)$ 反映了路径 p_j 与其他路径的总体相似度，且 $\sum_{k=1}^{n} s(p_j, p_k)$ 越大，p_j 与其他路径越相似。在目标路径集中选择 $\sum_{k=1}^{n} s(p_j, p_k)$ 最大的路径作为基准路径 p_1^b，可保证 p_1^b 与其他路径的总体相似度最大；选择与 p_1^b 相似度最大的前 $\lfloor g_1 \rfloor$ 条目标路径构成组 g_1，可保证 g_1 中的所有路径都与 p_1^b 具有很大的相似度，或者说，与 p_1^b 最相似的若干条目标路径均在 g_1 中。因此，g_1 中目标路径的相似度很大。对于其他组包含的目标路径，也满足上述条件，意味着本文方法选择目标路径构造不同组，是完全合理的。

上述分析表明，采用本文提出的方法对目标路径分组，既能充分利用可用的计算资源，又使每组包含的目标路径具有很高的相似度，为测试数据的高效生成奠定了基础。

6 在并行程序测试中的应用

本文将所提方法应用于 4 个并行程序测试中，并分别与基于协同进化遗传算法的单路径覆盖测试数据生成方法(记为方法 1)、基于遗传算法的单路径覆盖测试数据生成方法(记为方法 2)以及基于遗传算法的多路径覆盖测试数据生成方法(记为方法 3)进行比较，以评价本文方法的性能。其中，方法 2 和方法 3 均采用遗传算法生成测试数据，方法 2 一次生成覆盖一条路径的测试数据；而方法 3 考虑所有的目标路径，将多路径覆盖测试数据生成问题，建模为一个多目标优化问题，如式(1)所示，并通过运行一次遗传算法，生成覆盖所有目标路径的测试数据。方法 1 根据输入分量与进程子路径的对应关系，将种群划分为多个子种群，且每一子种群仅通过进化，优化与某一进程子路径相关的输入分量。此外，基于所有子种群的优良个体，形成一个合作团体群，以优化程序的所有输入。该方法通过多个子种群和一个合作团体群的交替协同进化，生成覆盖一条目标路径的测试数据。

所有实验均依托曙光 5000A 高性能计算集群。该集群刀片计算节点的硬件配置为 2 个六核 Xeon 5650 CPU、24 GB 内存、1×160 GB SAS 硬盘、Mellanox MTS3600 计算交换机以及 H3C 5120-24P-EI 千兆以太网络交换机；采用的操作系统包括 Linux 和 Windows Compute Cluster Server；集群管理系统为曙光 GridView；并行计算环境为 MPICH。

选择的 4 个被测程序分别为 Gcd1、Gcd2、Index 以及 Matrix。其中，程序 Gcd1 来自文献[23]，用于求取 3 个整数的最大公约数；Gcd2 是在 Gcd1 的基础上扩展得到的，用于求取 4 个整数的最大公约数；Index 是程序 make 的一个功能，用于字符检索；对于程序 Matrix，除了实现矩阵相乘之外，还判断其中一个矩阵中元素之间的关系。程序的基本信息如表 1 所示。

表 1 被测程序的基本信息

名称	输入分量个数	进程个数	发送消息个数	接收消息个数	输入取值范围
Gcd1	3	4	11	9	$[1,64]^3$
Gcd2	4	7	18	18	$[1,64]^4$
Index	15	4	9	9	$[0,128]^{15}$
Matrix	9	3	6	6	$[0,127]^9$

生成覆盖上述程序目标路径的测试数据时，可用的计算资源和种群规模如表 2 所示。

表 2 计算资源和种群规模

程序	计算资源数量	种群规模
Gcd1	12	300
Gcd2	21	600
Index	12	240
Matrix	6	90

除了本文方法和 3 个对比方法采用相同的种群规模之外，其他方法也采用与本文方法相同的遗传操作，即均为轮盘赌选择、单点交叉以及单点变异，且交叉和变异概率分别为 0.9 和 0.3，最大迭代次数为 10 000。

为了评价不同方法生成测试数据的性能，采用进化个体评价次数、耗时、路径覆盖率 3 个性能指标。其中，路径覆盖率是指测试数据覆盖的目标路径占所有目标路径的比值。生成覆盖不同组目标路径的测试数据的耗时相同，本文方法的耗时是指，在所有计算资源组中，生成覆盖满足要求的测试数据需要的最长时间；个体评价次数为求解所有子问题所需的个体评价次数之和。

为了减少随机因素对不同方法性能的影响，针对每一程序，每种方法独立运行 30 次，记录每次运行生成的期望的测试数据、进化个体评价次数以及耗时，并求取进化个体评价次数和耗时的平均值以及路径覆盖率。为了进一步说明本文方法与对比方法在各性能指标上的差异是否显著，对上述指标进行 Mann-Whitney U 非参数检验，并取显著性水平为 0.05。此

外,原假设为两种对比方法的指标值相同,如果某指标的统计测试值 U 小于 0.05,则拒绝原假设,说明两种对比方法在该指标上具有显著差别。

6.1 实例分析

首先,从程序 Gcd1 中选择 9 条路径作为目标路径,该程序的源代码如下所示,共包含 4 个进程。其中,进程 1 的代码为:

```
#include<mpi.h>
#include <string.h>
int main(int argc, char * *argv){
1   int x,y,z;
int myid, numprocs;
MPI_Status status;
MPI_Init(&argc,&argv);
      MPI_Comm_rank(MPI_COMM_WORLD,&myid);
      MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
      scanf("%d%d%d",&x,&y,&z);
2   MPI_Send(&x,1,MPI_INT,1,1,MPI_COMM_WORLD);
3   MPI_Send(&y,1,MPI_INT,1,2,MPI_COMM_WORLD);
4   MPI_Send(&y,1,MPI_INT,2,1,MPI_COMM_WORLD);
5   MPI_Send(&z,1,MPI_INT,2,2,MPI_COMM_WORLD);
6   MPI_Recv(&x,1,MPI_INT,1,1,MPI_COMM_WORLD,&status);
7   MPI_Recv(&y,1,MPI_INT,2,1,MPI_COMM_WORLD,&status);
8  if(x>1&&y>1)
9   {MPI_Send(&x,1,MPI_INT,3,1,MPI_COMM_WORLD);
10   MPI_Send(&y,1,MPI_INT,3,2,MPI_COMM_WORLD);
11   MPI_Recv(&z,1,MPI_INT,3,0,MPI_COMM_WORLD,&status); }
     else
12  { x=0; y=0;z=1;
13   MPI_Send(&x,1,MPI_INT,3,1,MPI_COMM_WORLD);
14   MPI_Send(&y,1,MPI_INT,3,2,MPI_COMM_WORLD); }
15   printf("最大公约数为%d\n",z);
MPI_Finalize();}
```

进程 1 和进程 2 的代码为:

```
#include<mpi.h>
#include <string.h>
int main(int argc, char * *argv){
1   int x,y;
int myid, numprocs;
MPI_Status status;
MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
2   MPI_Recv(&x,1,MPI_INT,0,1,MPI_COMM_WORLD,&status);
3   MPI_Recv(&y,1,MPI_INT,0,2,MPI_COMM_WORLD,&status);
4   while(x! =y)
5   {if(x<y)
6    y=y-x;
else
7   x=x-y;
8   }
9   MPI_Send(&x,1,MPI_INT,0,1,MPI_COMM_WORLD);
10 MPI_Finalize();}
```

进程 3 的代码为:

```
#include<mpi.h>
#include <string.h>
int main(int argc, char * *argv){
1   int x,y,z;
int myid, numprocs;
MPI_Status status;
MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
2   MPI_Recv(&x,1,MPI_INT,0,1,MPI_COMM_WORLD,&status);
3   MPI_Recv(&y,1,MPI_INT,0,2,MPI_COMM_WORLD,&status);
4   if(x>1&&y>1){
5   while(x! =y)
6   {if(x<y)
7    y=y-x;
   else
8    x=x-y;
9   }
10   MPI_Send(&x,1,MPI_INT,0,0,MPI_COMM_WORLD); }
11   MPI_Finalize();}
```

根据表 2 的计算资源数,将目标路径分为 3 组,每组包含 3 条目标路径,如表 3 所示。表中,“‖”用于分隔属于不同进程的子路径。

表 3 目标路径分组

<table>
<tr><th>组</th><th>目标路径</th></tr>
<tr><td>g_1</td><td>$p_{11}=n_1^0 n_2^0 n_3^0 n_4^0 n_5^0 n_6^0 n_7^0 n_8^0 n_9^0 n_{10}^0 n_{11}^0 n_{15}^0 \Vert n_1^1 n_2^1 n_3^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_9^1 n_{10}^1 \Vert n_1^2 n_2^2 n_3^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_7^2 n_8^2 n_4^2 n_5^2 n_7^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_7^2 n_8^2 n_4^2 n_9^2 n_{10}^2 \Vert n_1^3 n_2^3 n_3^3 n_4^3 n_5^3 n_9^3 n_{10}^3 n_{11}^3$
$p_{12}=n_1^0 n_2^0 n_3^0 n_4^0 n_5^0 n_6^0 n_7^0 n_8^0 n_9^0 n_{10}^0 n_{11}^0 n_{15}^0 \Vert n_1^1 n_2^1 n_3^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_9^1 n_{10}^1 \Vert n_1^2 n_2^2 n_3^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_6^2 n_4^2 n_9^2 n_{10}^2 \Vert n_1^3 n_2^3 n_3^3 n_4^3 n_5^3 n_9^3 n_{10}^3 n_{11}^3$
$p_{13}=n_1^0 n_2^0 n_3^0 n_4^0 n_5^0 n_6^0 n_7^0 n_8^0 n_9^0 n_{10}^0 n_{11}^0 n_{15}^0 \Vert n_1^1 n_2^1 n_3^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_9^1 n_{10}^1 \Vert n_1^2 n_2^2 n_3^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_7^2 n_8^2 n_4^2 n_5^2 n_7^2 n_8^2 n_4^2 n_5^2 n_7^2 n_8^2 n_4^2 n_5^2 n_7^2 n_8^2 n_4^2 n_5^2 n_7^2 n_8^2 n_4^2 n_9^2 n_{10}^2 \Vert n_1^3 n_2^3 n_3^3 n_4^3 n_5^3 n_6^3 n_7^3 n_9^3 n_5^3 n_6^3 n_8^3 n_9^3 n_5^3 n_{10}^3 n_{11}^3$</td></tr>
<tr><td>g_2</td><td>$p_{21}=n_1^0 n_2^0 n_3^0 n_4^0 n_5^0 n_6^0 n_7^0 n_8^0 n_{12}^0 n_{13}^0 n_{14}^0 n_{15}^0 \Vert n_1^1 n_2^1 n_3^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_9^1 n_{10}^1 \Vert n_1^2 n_2^2 n_3^2 n_4^2 n_5^2 n_7^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_7^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_7^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_9^2 n_{10}^2 \Vert n_1^3 n_2^3 n_3^3 n_4^3 n_{11}^3$
$p_{22}=n_1^0 n_2^0 n_3^0 n_4^0 n_5^0 n_6^0 n_7^0 n_8^0 n_{12}^0 n_{13}^0 n_{14}^0 n_{15}^0 \Vert n_1^1 n_2^1 n_3^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_9^1 n_{10}^1 \Vert n_1^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_7^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_7^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_9^2 n_{10}^2 \Vert n_1^3 n_2^3 n_3^3 n_4^3 n_{11}^3$
$p_{23}=n_1^0 n_2^0 n_3^0 n_4^0 n_5^0 n_6^0 n_7^0 n_8^0 n_{12}^0 n_{13}^0 n_{14}^0 n_{15}^0 \Vert n_1^1 n_2^1 n_3^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_9^1 n_{10}^1 \Vert n_1^2 n_2^2 n_3^2 n_4^2 n_5^2 n_7^2 n_8^2 n_4^2 n_9^2 n_{10}^2 \Vert n_1^3 n_2^3 n_3^3 n_4^3 n_{11}^3$</td></tr>
<tr><td>g_3</td><td>$p_{31}=n_1^0 n_2^0 n_3^0 n_4^0 n_5^0 n_6^0 n_7^0 n_8^0 n_{12}^0 n_{13}^0 n_{14}^0 n_{15}^0 \Vert n_1^1 n_2^1 n_3^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_9^1 n_{10}^1 \Vert n_1^2 n_2^2 n_3^2 n_4^2 n_5^2 n_7^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_7^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_7^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_9^2 n_{10}^2 \Vert n_1^3 n_2^3 n_3^3 n_4^3 n_{11}^3$
$p_{32}=n_1^0 n_2^0 n_3^0 n_4^0 n_5^0 n_6^0 n_7^0 n_8^0 n_{12}^0 n_{13}^0 n_{14}^0 n_{15}^0 \Vert n_1^1 n_2^1 n_3^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_9^1 n_{10}^1 \Vert n_1^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_7^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_5^2 n_7^2 n_8^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_9^2 n_{10}^2 \Vert n_1^3 n_2^3 n_3^3 n_4^3 n_{11}^3$
$p_{33}=n_1^0 n_2^0 n_3^0 n_4^0 n_5^0 n_6^0 n_7^0 n_8^0 n_{12}^0 n_{13}^0 n_{14}^0 n_{15}^0 \Vert n_1^1 n_2^1 n_3^1 n_4^1 n_5^1 n_7^1 n_8^1 n_4^1 n_5^1 n_6^1 n_8^1 n_4^1 n_9^1 n_{10}^1 \Vert n_1^2 n_2^2 n_3^2 n_4^2 n_5^2 n_7^2 n_8^2 n_4^2 n_9^2 n_{10}^2 \Vert n_1^3 n_2^3 n_3^3 n_4^3 n_{11}^3$</td></tr>
</table>

根据表 3 的目标路径分组和式(9),可得到如下 3 个子优化问题:

$$\begin{cases}\max(f_{11}(\boldsymbol{x}),f_{12}(\boldsymbol{x}),f_{13}(\boldsymbol{x})),\text{s. t. } \boldsymbol{x}\in D;\\ \max(f_{21}(\boldsymbol{x}),f_{22}(\boldsymbol{x}),f_{23}(\boldsymbol{x})),\text{s. t. } \boldsymbol{x}\in D;\\ \max(f_{31}(\boldsymbol{x}),f_{32}(\boldsymbol{x}),f_{33}(\boldsymbol{x})),\text{s. t. } \boldsymbol{x}\in D。\end{cases} \quad (12)$$

由式(8)可得:

$$f_{kj}(x)=\frac{1}{4}\sum_{i=0}^{3}s(p^i,p_{kj}^i)k,j=1,2,3,$$

$$D=D_1\times D_2\times D_3。$$

采用 3 种群并行遗传算法,分别求解上述 3 个子优化问题时,各子种群的适应度函数分别为:

$$\begin{cases}F_1(\boldsymbol{x})=\max\{f_{11}(\boldsymbol{x}),f_{12}(\boldsymbol{x}),f_{13}(\boldsymbol{x})\};\\ F_2(\boldsymbol{x})=\max\{f_{21}(\boldsymbol{x}),f_{22}(\boldsymbol{x}),f_{23}(\boldsymbol{x})\};\\ F_3(\boldsymbol{x})=\max\{f_{31}(\boldsymbol{x}),f_{32}(\boldsymbol{x}),f_{33}(\boldsymbol{x})\}。\end{cases} \quad (13)$$

为了说明本文所提方法,以某一次实验为例,给出第一个子问题的求解过程。

当子种群 1 进化到第 592 代时,生成了覆盖目标路径 p_{13} 的测试数据,适应度函数 $F_1(\boldsymbol{x})$ 简化为

$$F_1(\boldsymbol{x})=\max\{f_{11}(\boldsymbol{x}),f_{12}(\boldsymbol{x})\}。\quad (14)$$

当该子种群进化到第 996 代时,找到覆盖目标路径 p_{11} 的测试数据,适应度函数 $F_1(\boldsymbol{x})$ 进一步简化为

$$F_1(\boldsymbol{x})=\max f_{12}(\boldsymbol{x})。\quad (15)$$

当该子种群进化到第 2 002 代时,覆盖路径 p_{12} 的测试数据也被找到,生成了所有满足路径覆盖要求的测试数据,因此终止该子种群的进化。

其他两个子问题的求解过程与第一个子问题完全相同,不再赘述。

在程序 Gcd2 中选择 6 条路径作为目标路径,根据表 2 的计算资源数,将该 6 条目标路径分为 3 组,每组包含 2 条目标路径。除了本文方法之外,还采用方法 3 生成覆盖程序 Gcd1 和 Gcd2 的多条目标路径的测试数据,结果如表 4 所示。表中,上标“+”和“0”分别表示两种方法关于某指标具有和不具有显著性差异。

由表 4 可知:①对于程序 Gcd1,两种方法的路径覆盖率均为 100%,但本文方法的个体评价次数和耗

表 4 程序 Gcd1 和 Gcd2 的实验结果

<table>
<tr><th rowspan="2">程序</th><th rowspan="2">方法</th><th colspan="3">个体评价次数</th><th colspan="3">消耗时间/s</th><th colspan="3">路径覆盖率/%</th></tr>
<tr><th>均值</th><th>标准差</th><th>U</th><th>均值</th><th>标准差</th><th>U</th><th>均值</th><th>标准差</th><th>U</th></tr>
<tr><td rowspan="2">Gcd1</td><td>本文方法</td><td>1 981 749.0</td><td>855 294.4</td><td rowspan="2">0^+</td><td>6.85</td><td>3.004</td><td rowspan="2">0.008^+</td><td>100</td><td>0</td><td rowspan="2">1^0</td></tr>
<tr><td>方法 3</td><td>3 413 948.0</td><td>2 107 706.0</td><td>16.82</td><td>11.901</td><td>100</td><td>0</td></tr>
<tr><td rowspan="2">Gcd2</td><td>本文方法</td><td>3 971 109.0</td><td>1 356 617.0</td><td rowspan="2">0.003^+</td><td>52.66</td><td>7.137</td><td rowspan="2">0^+</td><td>82.2</td><td>18.010</td><td rowspan="2">0.414^+</td></tr>
<tr><td>方法 3</td><td>5 472 575.0</td><td>969 013.8</td><td>220.12</td><td>38.572</td><td>77.8</td><td>20.215</td></tr>
</table>

时仅分别为方法 3 的 58%和 13%，此外，由统计测试值可以看出，该两种方法在个体评价次数和消耗时间上的差异是显著的；②对于程序 Gcd2，本文方法的路径覆盖率略高于方法 3，但统计测试值表明，该两种方法针对该指标的差异并不显著。本文方法的个体评价次数和消耗时间分别为方法 3 的 25%和 73%。

上述实验结果表明，对于程序 Gcd1 和 Gcd2，本文方法在路径覆盖率、个体评价次数以及耗时等指标方面优于方法 3。

6.2　其他程序实验结果

在程序 Index 和 Matrix 中，分别选择 9 和 12 条路径作为目标路径。采用本文方法生成测试数据时，首先根据该两个程序包含的进程数（分别为 4 和 3）以及可用的计算资源数（分别为 12 和 6），应用本文的目标路径分组方法，将目标路径分别分成 3 和 2 组；然后，利用基于多种群并行遗传算法的测试数据生成方法，生成期望的测试数据。与其他 3 种方法的对比实验结果如表 5 所示。

表 5　其他程序实验结果

程序	方法	评价次数			消耗时间/s			路径覆盖率/%		
		均值	标准差	U	均值	标准差	U	均值	标准差	U
Index	本文方法	1 370 933.0	180 811.5		12.13	3.373		84.07	17.18	
	方法 1	12 801 295.0	2 036 631	0^+	18.47	3.492	0^+	76.29	19.07	0.111^0
	方法 2	13 505 216.0	3 028 263	0^+	177.30	41.005	0^+	74.44	18.94	0.086^0
	方法 3	2 270 400.0	272 057	0^+	36.17	4.828	0^+	77.40	10.07	0.233^0
Matrix	本文方法	461 428.5	151 424		3.12	0.988		95.83	4.68	
	方法 1	5 175 954.0	1 051 187	0^+	6.50	1.471	0^+	94.04	5.05	0.278^0
	方法 2	3 851 628.0	958 003.2	0^+	46.25	11.510	0^+	93.88	3.74	0.061^0
	方法 3	906 824.9	557 325.2	0^+	8.98	4.755	0^+	98.05	3.58	0.053^0

由表 5 可以看出：

1）采用本文方法生成测试数据，需要的个体评价次数和耗时均是最少的。对于程序 Index 而言，本文方法的个体评价次数只有方法 1 和 2 的 11%左右，约占方法 3 的 60%；耗时分别为其他 3 种方法的 66%、7%以及 34%。关于程序 Matrix，本文方法需要的个体评价次数和耗时分别为方法 1 的 9%和 48%，方法 2 的 11%和 7%以及方法 3 的 51%和 35%。针对上述两个程序的统计测试值表明，本文方法显著优于另外 3 种方法。

2）本文方法的路径覆盖率与其他 3 种方法没有显著差别。关于程序 Index，本文方法的路径覆盖率最高；关于程序 Matrix，本文方法的路径覆盖率高于方法 1 和 2，略低于方法 3，但统计测试值表明，对于这两个程序，本文方法与其他 3 种方法的路径覆盖率没有显著差异。

上述实验结果表明，与其他方法相比，本文方法在保证路径覆盖率的前提下，能够减少生成测试数据所需的个体评价次数和耗时。

7　结　论

本文研究消息传递并行程序多路径覆盖测试数据生成问题。为了降低问题求解的难度，将目标路径分成若干组，在分组过程中，充分考虑了并行程序包含的进程数、可用的计算资源数以及路径相似度。为了提高测试数据生成效率，采用多种群并行遗传算法，并生成所有满足路径覆盖要求的测试数据。理论分析表明，采用本文的分组方法能够使每组的目标路径具有很大的相似度，并可均衡计算资源的负载。将所提方法应用于 4 个并行程序测试中，实验结果表明，采用本文方法生成测试数据，不但可保证路径覆盖率，而且需要的个体评价次数和耗时也很少。

本文的主要贡献体现在如下 4 个方面：①提出了适用于并行程序的目标路径分组方法；②建立了并行程序多路径覆盖测试数据生成问题的数学模型；③设计了基于多种群并行遗传算法的测试数据生成方法；④通过在基准并行程序中的应用，验证了所提方法能够高效生成覆盖多路径的测试数据。

本文通过固定进程调度序列，使得并行程序的执行过程是确定的，从而忽略了不确定性对测试数据生成的影响。但是，不确定性是并行程序一个非常重要的特征。当考虑程序执行的不确定性时，多路径覆盖测试数据生成方法与本文提出的方法有很大区别。如何进行目标路径分组，建立测试数据生成问题的数学模型以及高效生成测试数据，以同时覆盖多条目标路径，是需要进一步研究的问题。

[参考文献] (References)

[1] 张军华，臧胜涛，单联瑜，等. 高性能计算的发展现状及趋势[J]. 石油地球物理勘探，2010，45(6)：918-925.
Zhang Junhua, Zang Shengtao, Shan Lianyu, et al. Current situation and trend of high performance computing [J]. Phys Prospect Petrol, 2010, 45(6): 918-925. (in Chinese)

[2] 黄永勤，金利峰，刘耀. 高性能计算机的可靠性技术现状与趋势[J]. 计算机研究与发展，2010，47(4)：589-594.
Huang Yongqin, Jin Lifeng, Liu Yao. Current situation and trend of reliability technology in high performance computers [J]. J Comput Res Dev, 2010, 47(4): 589-594. (in Chinese)

[3] Almasi G S, Gottlieb A. Highly Parallel Computing [M]. Menlo Park: The Benjamin Cummings Publishing Company, 1989.
[4] 陈国良，安虹，陈崚，等. 并行算法实践[M]. 北京：高等教育出版社，2004.
Chen Guoliang, An Hong, Chen Ling, et al. Parallel Algorithm Practice [M]. Beijing: Higher Education Press, 2004. (in Chinese)
[5] Snir M, Otto S, Huss-Lederman S, et al. MPI: the Complete Reference [M]. Cambridge: MIT Press, 1998.
[6] Geist G A, Kohl J A, Papadopoulos P M, et al. Beyond PVM 3.4: what we've learned what's next, and why [C]//15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface. Berlin: Springer, 1997: 116-126.
[7] 单锦辉，姜瑛，孙萍. 软件测试研究进展[J]. 北京大学学报:自然科学版，2005，41(1)：134-145.
Shan Jinhui, Jiang Ying, Sun Ping. Research progress of software testing [J]. Acta Sci Nat Univ Pek, 2005, 41(1): 134-145. (in Chinese)
[8] Gong Dunwei, Zhang Wanqiu, Yao Xiangjuan. Evolutionary generation of test data for many paths coverage based on grouping [J]. J Syst Software, 2011, 84(12): 2222-2233.
[9] Krammer B, Resch M M. Correctness checking of MPI one-sided communication using marmot [J]. Lect Notes Comput Sci, 2006, 4192: 105-114.
[10] Vetter J S, Supinski B R. Dynamic software testing of MPI applications with umpire [C]//Supercomputing ACM/IEEE Conference. New York: IEEE, 2000: 70-79.
[11] Park M Y, Shim S J, Jun Y K, et al. Mpirace-check: detection of message races in MPI programs[C]//2nd International Conference on Grid and Pervasive Computing. Berlin: Springer, 2007: 322-333.
[12] Yu Lei, Carver R H. Reachability testing of concurrent programs [J]. IEEE Trans Software Eng, 2006, 32 (6): 382-403.
[13] Carver R H, Yu Lei. Distributed reachability testing of concurrent programs [J]. Concurr Comput-Pract E, 2010, 22(18): 2445-2466.
[14] Souza S, Souza P, Machado M. Using coverage and reachability testing to improve concurrent program testing quality [C]//23rd International Conference on Software Engineering and Knowledge Engineering. Miami, USA, 2011: 207-212.
[15] Cheng J. Slicing concurrent programs-A graph theoretical approach [C]//1st International Workshop on Automated and Algorithmic Debugging. Berlin: Springer, 1993: 223-240.
[16] Christakis M, Sagonas K. Detection of asynchronous message passing errors using static analysis[C]//13th International Conference on Practical Aspects of Declarative Language. Berlin: Springer, 2011: 5-18.
[17] Engler D, Ashcraft K. Racerx: effective, static detection of race conditions and deadlocks [C]//ACM SIGOPS Operating Systems Review. New York: ACM, 2003, 37(5): 237-252.
[18] Flanagan C, Godefroid P. Dynamic partial-order reduction for model checking software [J]. ACM Sigplan Notices. 2005, 40(1): 110-121.
[19] Vakkalanka S, DeLisi M, Gopalakrishnan G, et al. Implementing efficient dynamic formal verification methods for MPI programs [C]//15th European PVM/MPI Users' Group Meeting. Dublin, Ireland, 2008: 248-256.
[20] Chen Qichang, Wang Liqiang, Yang Zijiang, et al. HAVE: detecting atomicity violations via integrated dynamic and static analysis [C]//12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software. Berlin: Springer, 2009: 425-439.
[21] Yang C S D, Pollock L L. All-uses testing of shared memory parallel programs [J]. Software Test Verif Reliab, 2003, 13(1): 3-24.
[22] Yang C S D, Souter A L, Pollock L L. All-du-path coverage for parallel programs [C]//ACM International Symposium on Software Testing and Analysis. New York, USA, 1998: 153-162.
[23] Souza S R S, Vergilio S R, Souza P S L. Structural testing criteria for message-passing parallel programs [J]. Concurr Comput-Pract E, 2008, 20(16): 1893-1916.
[24] Souza P S L, Souza S R S, Zaluska E. Structural testing for message-passing concurrent programs: an extended test model [J]. Concurr Comput-Pract E, 2014, 26(1): 21-50.
[25] Chen T Y, Kuo F C, Merkel R G, et al. Adaptive random testing: the art of test case diversity [J]. J Syst Software, 2010, 83(1): 60-66.
[26] 王志言，刘椿年. 区间算术在软件测试中的应用[J]. 软件学报，1998，9(6)：438-443.
Wang Zhiyan, Liu Chunnian. The application of interval computation in software testing [J]. J Software, 1998, 9(6): 438-443. (in Chinese)
[27] Miller W, Spooner D L. Automatic generation of floating-point test data [J]. IEEE Trans Software Eng, 1976, 2(3): 223-226.
[28] Korel B. Automated software test data generation [J]. IEEE Trans Software Eng, 1990, 16(8): 870-879.
[29] Gupta N, Mathur A P, Soffa M L. Automated test data generation using an iterative relaxation method [J]. ACM SIGSOFT Software Engineering Notes. 1998, 23 (6): 231-244.
[30] Xanthakis S, Ellis C, Skourlas C, et al. Application of genetic algorithms to software testing [C]//5th International Conference on Software Engineering and Applications. Toulouse, France, 1992: 625-636.
[31] Ahmed M A, Hermadi I. GA-based multiple paths test data generator [J]. Comput Operat Res, 2008, 35 (10): 3107-3124.
[32] Bueno P, Jino M. Automatic test data generation for program path using genetic algorithms [J]. Int J Software Eng Knowl Eng, 2002, 12(6): 691-709.
[33] Lin J C, Yeh P L. Using genetic algorithms for test case generation in path testing [C]//9th Asian Test Symposium. New York: IEEE, 2000: 241-246.
[34] 谢晓园，徐宝文，史亮，聂长海. 面向路径覆盖的演化测试用例生成技术[J]. 软件学报. 2009，20(12)：3117-3136.
Xie Xiaoyuan, Xu Baowen, Shi Liang, et al. Genetic test case generation for path-oriented testing [J]. J Software, 2009, 20(12): 3117-3136. (in Chinese)
[35] Arcuri A. Longer is better: on the role of test sequence length in software testing [C]//3rd International Conference on Software Testing, Verification and Validation. New York, USA, 2010: 469-478.
[36] Fraser G, Arcuri A. It is not the length that matters, it is how you control it [C]//IEEE International Conference on Software Testing, Verification and Validation. New York, USA, 2011: 150-159.
[37] McMinn P, Harman M, Binkley D, et al. The species per path approach to search-based test data generation [C]//International Symposium on Software Testing and Analysis. New York, USA, 2006: 13-24.
[38] 巩敦卫，张岩. 一种新的多路径覆盖测试数据进化生成方法[J]. 电子学报，2010，38(6)：1299-1304.
Gong Dunwei, Zhang Yan. Novel evolutionary generation approach to test data for multiple paths coverage [J]. Acta Electron Sin, 2010, 38(6): 1299-1304. (in Chinese)
[39] 田甜，巩敦卫. 消息传递并行程序的路径覆盖测试数据生成问题的数学模型及其进化求解方法[J]. 计算机学报，2013，36(11)：2212-2223.
Tian Tian, Gong Dunwei. Model of test data generation for path coverage of message-passing parallel programs and its evolution-based solution [J]. Chin J Comput, 2013, 36(11): 2212-2223. (in Chinese)