The Journal of Systems and Software 210 (2024) 111962

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Test data generation for covering mutation-based path using MGA for MPI program☆

Xiangying Dang [a], Jinyong Wang [a,*], Dunwei Gong [b], Xiangjuan Yao [c], Changqing Wei [c], Biao Xu [d]

[a] School of Information Engineering (School of Big Data), Xuzhou University of Technology, Xuzhou, 221018, China
[b] College of Automation and Electronic Engineering, Qingdao University of Science and Technology, 266061, China
[c] School of Mathematics, China University of Mining and Technology, Xuzhou 221116, China
[d] Department of Electronic Engineering, Shantou University, Shantou 515063, China

ARTICLE INFO

Keywords:
Message passing interface (MPI)
Software testing
Mutation testing
Multi-population genetic algorithm (MGA)
Test data generation

ABSTRACT

Message Passing Interface (MPI) is a communication protocol used for parallel programming in various languages, valued for its reliability and broad applicability. Mutation testing is a software testing method for systematically simulating software faults. However, the significant number of inserted mutation branches in the program escalates the testing cost. To address this issue, we propose the comprehensive **PMTDGM** framework. The framework first generates mutation-based paths based on the relevancy of mutant branches and the difficulty of mutant branch coverage, followed by the establishment of a multitask model of path coverage. Finally, we employ a multi-population genetic algorithm (MGA) to generate test data. Our experiments, performed on six MPI programs of varying sizes and structures, demonstrate that the mutation-based paths have the small test sets and are easy to cover. Additionally, the multitask model and MGA can significantly improve the efficiency of generating test data and reduce the cost of mutation testing for MPI programs compared to traditional methods.

1. Introduction

The reliability of parallel programs is a persistent concern due to their size, complexity, and application domains (Hager and Wellein, 2010; Pan et al., 2021). Software testing is an effective approach to improve software quality by identifying faults (Hamlet, 1994). However, detecting faults in parallel programs is challenging due to the uncertainty of process scheduling. To address this challenge, we utilize mutation testing as a fault-oriented technique for testing software systems (Souza et al., 2014b). In addition, the traditional methods for obtaining test data are often unsuitable for parallel programs (Gong et al., 2020), which led us to employ MGA to efficiently generate mutation-based test data

MPI programs support point-to-point broadcasting (Akhmetova et al., 2017). These programs typically feature multiple processes running in parallel that communicate with each other via message passing. Processes belong to one or more communication domains, and the characteristics of MPI programs include parallelism, interactivity, synchronization, and uncertainty, all contributing to their problem-solving capabilities. However, specific challenges such as data competition, resource conflicts, and deadlocks can arise in MPI programs, making fault detection even more challenging. Given the challenges of detecting faults in MPI programs, simulation of faults can be achieved by introducing mutations based on mutation operators (Hamlet, 1977; DeMillo et al., 1978), followed by generating test data to identify and kill these mutants, i.e., detect faults.

Mutation operators refer to the set of rules and techniques employed to introduce mutations or faults into a program during mutation testing (Papadakis et al., 2019). The objective is to simulate potential faults in the original program, which can vary from basic changes in the code. Once these mutation operators are applied, the resulting modified program is referred to as a mutant.

Strong mutation testing entails running identical test data on both the original program and its mutants. If a mutant produces different output results compared to the original program, it is defined as "killed" according to the strong mutation testing criteria (Jia and Harman, 2011). However, generating and testing mutants can be a costly process. Finding appropriate test data can be challenging, and the cost of executing the same test data on multiple mutants can also be significant.

To address the limitations of strong mutation testing, weak mutation testing criteria have been proposed. Papadakis et al. (Papadakis and

☆ Editor: Aldeida Aleti.
* Corresponding author.
E-mail address: jinyongw@xzit.edu.cn (J. Wang).

https://doi.org/10.1016/j.jss.2024.111962
Received 25 June 2023; Received in revised form 16 November 2023; Accepted 3 January 2024
Available online 7 January 2024

Malevris, 2011) developed a method for transforming a large number of mutants into "mutant branches", based on the weak mutation testing criteria. This method can reduce the execution costs of mutation testing by generating a new program with a large number of mutant branches. Further, Papadakis et al. (Papadakis and Malevris, 2012) proposed a path selection method to cover all mutant branches, and adopted some strategies to select some paths in a given path set. However, their method can lead to an increase in path complexity and does not necessarily reduce the number of mutant branches. To overcome these challenges, Gong et al. (2017b) and Zhang et al. (2015) analyzed the dominant relationships among mutant branches to reduced their number accordingly, and then generate test data covering these mutant branches by an evolutionary algorithm. The above methods can improve the efficiency of mutation testing. Mutation testing for parallel programs is an active area of research. Gong et al. (2016) proposed a weak mutation testing method for MPI programs, which transformed the task of killing mutants into one of branch coverage.

Structural testing is a traditional method for software testing, and path coverage testing has been a particularly fruitful area of research in this field (Wegener et al., 2001; Souza et al., 2008, 2014b). In general, a path consists of multiple nodes, with each node containing to one or more statements. Test data that covers these paths tend to be of high quality, as they cover more statements (Gong et al., 2021; Dang et al., 2016). Drawing upon this prior research, our research focuses on transforming the problem of killing mutants in parallel programs into one of path coverage in this paper. To achieve this, we design range of strategies for grouping mutant branches into multiple paths.

Mutation testing is a powerful tool that can both simulate faults and assist in generating test data to detect faults (Souza et al., 2014a; Papadakis et al., 2010). There are currently three main families of methods for test data generation (DeMillo and Offutt, 1991; Papadakis and Malevris, 2010; Zhang and Gong, 2013; Dang et al., 2020). Search-based methods, such as genetic algorithms (GAs), are a popular approach to test data generation for mutation testing. GAs are a global evolutionary algorithm that is inspired by the mechanisms of biological evolution (Silva et al., 2017). This method is effective in generating test data by identifying optimal or near-optimal solutions for achieving desired test coverage goals.

Multi-population genetic algorithm (MGA) is a high-performance genetic algorithm that divides a population into subpopulations (Yao and Gong, 2014). The individuals in each subpopulation of MGA have the potential for parallelism and are allowed to migrate from one subpopulation to another (Dang et al., 2021). In this paper, we transform the problem of mutation-based test data generation into an optimization problem with multiple subtasks, and apply MGA to further improve the efficiency of test data generation.

In this paper, we leverage our previous work on serial programs (Dang et al., 2016, 2020, 2021), and apply them to address the unique challenges posed by MPI programs, resulting in the development of a comprehensive framework called **PMTDGM** (**P**ath generation and **M**utation-based **T**est **D**ata **G**eneration by **M**ulti-population genetic algorithm). Our framework involves generating a path set containing mutant branches in the processes, establishing a multitask optimization model of path coverage, and employing an MGA to generate test data.

This paper presents three significant contributions to the field of mutation testing for parallel programs

(1) The establishment of the **PMTDGM** framework, which efficiently facilitates the generation of mutation-based test data for parallel programs while minimizing the cost associated with mutation testing.

(2) The transformation of the mutation testing problem into a path coverage problem for MPI programs, where mutant branches are grouped by generating mutation-based paths. This method reduces the cost of executing large mutant branches and improves the efficiency of mutation testing.

(3) The establishment of a multi-task optimization model and the use of MGA to generate test data efficiently, while achieving a high mutant-killing capability.

The paper is organized as follows. Section 2 introduces the background information. The **PMTDGM** framework is described in Section 3, together with the related concepts and detailed algorithms. In Section 4, the design and settings of the empirical studies are presented. The experimental results are summarized and analyzed in Section 5. Threats to validity are discussed in Section 6. In Section 7, we present the related works. In Section 8, we conclude the paper and highlight some future work.

2. Background

2.1. Message-passing interface (MPI) programs

MPI is a widely used standard for message passing interface, commonly employed in the development of parallel message programs (Pan et al., 2021; Akhmetova et al., 2017; Gropp et al., 1996). In general, an MPI program consists of multiple processes that execute in parallel and communicate with each other through messages. Each process belongs to one or more communication domains, which consist of process groups and communication contexts (Protopopov and Skjellum, 1996).

Fig. 1 includes the source code (the black code) for the MaxTriangle MPI program, which is a modified version of the triangle classification serial program. The "MPI_Send()" and "MPI_Recv()" functions are employed to send and receive messages between different processes. The program consists of four processes, labeled as S^0, S^1, S^2, S^3, and four input variables: x, y, z, w. The primary function of the S^0 process is to send x and y, z and w, and y and w to processes S^1, S^2, and S^3, respectively. Once this data has been sent, S^0 then performs a classification of the triangle based on the values of x, y and z. The role of processes S^0, S^1, S^2, S^3 is to receive the input values from S^0, swap x and y if x is less than y, and then send the resulting values back to S^0.

In general, there are some statements in each process of an MPI program. A statement unit consisting of sequentially executed statements is defined as a node. n^i_j is the jth node of process S^i, such as, n^0_2, n^1_3, n^2_2 in Fig. 1.

A path of an MPI program is composed of the subpaths in multiple processes. Suppose that an input X executes an MPI program under test, and a covered path of executing S^i is denoted as $\vec{g}^i = n^i_0, n^i_1, \ldots, n^i_{|\vec{g}^i|}$, where $|\vec{g}^i|$ is the number of nodes. Thus, the path of the MPI program can be expressed as:

$$\vec{g} = \vec{g}^0 \| \vec{g}^1 \| \ldots \| \vec{g}^i \| \ldots \| \vec{g}^{m-1}$$

where m is the number of processes. For example, an covered path in Fig. 1 can be represented as:

$$\begin{aligned}&\vec{g} = \vec{g}^0 \| \vec{g}^1 \| \vec{g}^2 \| \vec{g}^3 \\ &= n^0_1, n^0_2, n^0_3, n^0_4, n^0_6, n^0_8, n^0_{10}, n^0_{12}, n^0_{13} \| \\ &n^1_1, n^1_2, n^1_3, n^1_4 \| \\ &n^2_1, n^2_2, n^2_4 \| \\ &n^3_1, n^3_2, n^3_3, n^3_4\end{aligned}$$

Subpath $\vec{g}^0, \vec{g}^1, \vec{g}^2, \vec{g}^3$ do not represent the execution sequence in $\vec{g}$ because multiple processes execute in parallel (Tian and Gong, 2016).

By leveraging MPI, developers and researchers can build powerful and robust parallel programs that leverage the scalability and efficiency of message passing communication. However, as these programs become more complex, it becomes increasingly challenging to test them thoroughly and efficiently. The **PMTDGM** framework in this paper offer a powerful new approach to mutation testing for MPI programs, enabling the generation of high-quality test data that can detect potential faults and improve overall software quality and reliability.

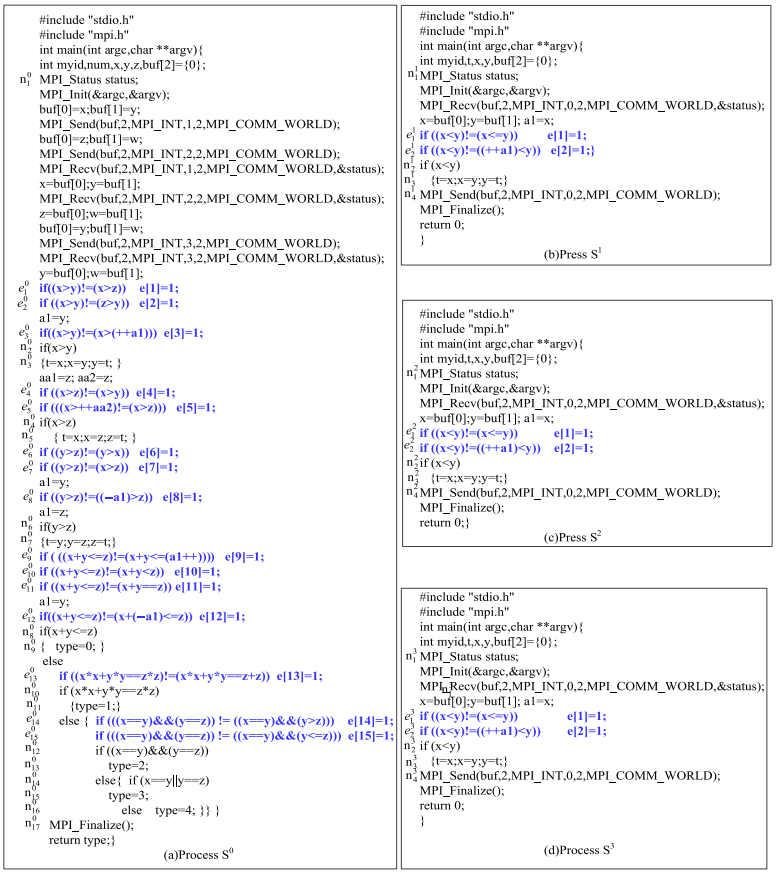


Fig. 1. New MaxTriangle MPI program with mutant branches.

Table 1
Information of mutation operators.

ID	Operator	Description
1	ABS	Absolute value insertion
2	AOR	Arithmetic operator replacement
3	CAR	Constant for array reference replacement
4	CRP	Constant replacement
5	CSR	Constant for scalar variable replacement
6	LCR	Logical connector replacement
7	ROR	Relational operator replacement
8	RSR	RETURN statement replacement
9	SCR	Scalar for constant replacement
10	SAR	Scalar variable for array reference replacement
11	SRC	Source constant replacement
12	SVR	Scalar variable replacement
13	UOI	Unary operator insertion

2.2. Mutant branch

Mutation testing involves intentionally introducing faults into a program by using mutation operators to syntactically modify its statements (Jia and Harman, 2011). Offutt et al. (Offutt and King, 1987) proposed a total of 22 classes mutation operators for C programs, as summarized in Table 1. These mutation operators introduce various types of faults into the code.

A program with only one statement modified is referred to as a first-order mutant (Jia and Harman, 2011), and the corresponding mutation testing is referred to as first-order mutation testing. On the other hand, if there are multiple statements modified in the program, it is called a higher-order mutant (Tokumoto et al., 2016). Furthermore, if a mutant cannot be killed or detected by any test data, it is known as an equivalent mutant (Silva et al., 2017).

Mutation testing techniques can generally be categorized into two types: strong mutation testing and weak mutation testing, based on their ability to meet the requirements for reachability, necessity, and propagation (Howden, 1982; Nishtha et al., 2017). Strong mutation testing criteria satisfy all three of these requirements, while weak mutation testing criteria only meet the requirements for reachability and necessity. While strong mutation testing is more comprehensive, it becomes increasingly expensive as the number of mutants grows, since each mutant is essentially a separate program.

To reduce the cost of strong mutation testing, Papadakis et al. (Papadakis and Malevris, 2011) proposed a technique to transform mutants into "mutant branches" for serial programs based on weak mutation testing criteria. A mutant branch is formed by taking the original statement(denoted as s) in the program under test and its mutated statement (denoted as s'), based on the necessity condition of mutation testing (Gong et al., 2017b). This method involves creating a new expression, denoted as "if $s \neq s'$...", to represent the mutant branch.

As shown in Fig. 1, the blue branches are the mutant branches which are inserted into the MaxTriangle MPI program. When a test datum executing mutant branch e_j^i, if the value of its true branch is 1, e_j^i is defined to be covered or killed under week mutation testing criteria.

2.3. Multi-population genetic algorithm (MGA)

Genetic algorithms (GAs) are a global search method inspired by biological evolution and genetic mutation. These algorithms operate by maintaining a population of potential solutions to a problem and using genetic operators such as selection, crossover, and mutation to mimic the natural process of evolution.

A multi-population genetic algorithm (MGA) is designed to improve the efficiency and effectiveness of GAs by employing multiple sub-populations. The general framework of traditional MGA is illustrated

Algorithm 1 Traditional multi-population genetic algorithm (MGA)

Input: $M = \{M_1, M_2, ..., M_{|M|}\}$ ($|M|$ optimization objectives), Population $X = \{X_1, X_2, ..., X_{|M|}\}$ (a population with $|M|$ subpopulations), ρ (maximum number of iterations)

Output: Optimal solutions

```
Initialize subpopulations X_i, i = 1, 2, ..., |M|;
Evaluate the fitness of each individual in each subpopulation for its own optimization objective;
while Termination criteria are not met do
    for each subpopulation X_i do
        Evaluate the fitness of each individual in each subpopulation;
        Select individuals from X_i based on their fitness to form mating pool;
        Perform crossover and mutation operations on the individuals to generate new individuals;
        Generate the new individuals in population X_i;
    end for
end while
Select the best individuals from all populations as the final solution;
Return Optimal solutions
```

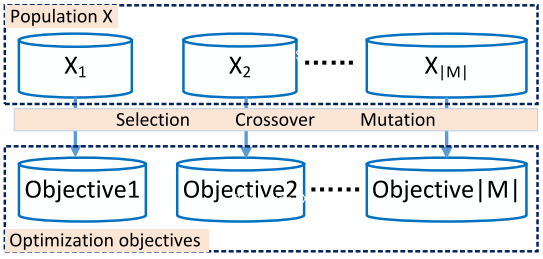


Fig. 2. The general framework of traditional MGA.

in Fig. 2 and the process of MGA is shown in Algorithm 1. The populations consist of multiple subpopulations, with each subpopulation corresponding to its own objective. First, each subpopulation is initialized and parameters are set separately. Then, the best individuals are selected based on their fitness value, and crossover and mutation operations are applied to the individuals in the mating pool for each subpopulation. This evolutionary process continues iteratively until the termination condition is satisfied. Finally, the algorithm outputs the optimal solution for each subpopulation. It is important to note that this framework is a simplified version, and in practical applications, additional adjustments and expansions may be necessary depending on the specific problem.

Yao et al. (Yao and Gong, 2014) proposed an MGA algorithm that utilizes individual information sharing to generate test data for achieving multipath coverage. In their method, MGA integrates migration between subpopulations, allowing individuals from one subpopulation to migrate to another. This introduces genetic diversity and facilitates the sharing of good solutions among subpopulations. In our previous study (Dang et al., 2021), MGA was employed to generate test data specifically aimed at killing stubborn mutants.

3. The proposed method

The proposed **PMTDGM** framework, presented in Fig. 3, is comprised of three main steps:

Step 1: Mutant branches are generated based on previous research (Dang et al., 2020, 2021) and inserted into the MPI program. Following this, **a correlogram of mutant branches** (Definition 3 in Section 3.1) is drawn based on their relevance and difficulty of coverage. Based on the correlogram, the mutation-based paths containing only mutant branches are generated.

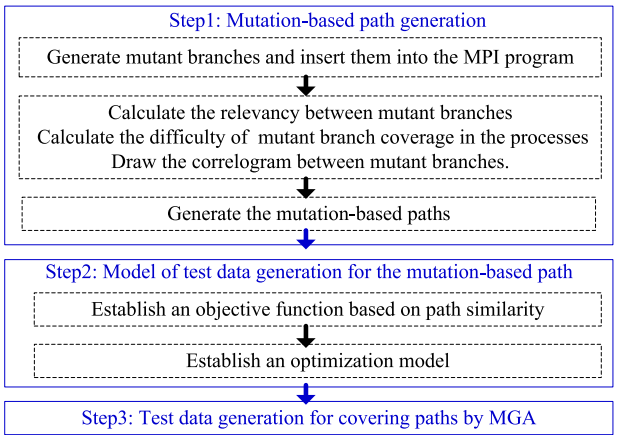


Fig. 3. The framework of the proposed method, **PMTDGM**.

Step 2: A multitask optimization model for multiple, with the objective function being formulated based on path similarity (stated in Eq. (4) in Section 3.3) as the key technique.

Step 3: MGA is employed to generate mutation-based test data by the model formulated in **Step 2** for the MPI program.

3.1. Basic definitions

In general, an MPI program, S, comprises m processes, where the ith process is represented as S^i, $i = 0, 1, \ldots, m-1$. The jth mutant branch in S^i is denoted as $e^i_j, j = 1, 2, \ldots$ and the set of all mutant branches is expressed as:

$$H = \{e^0_1, \ldots, e^0_{|S^0|}, \ldots, e^i_1, \ldots, e^i_{|S^i|}, \ldots, e^{m-1}_1, \ldots, e^{m-1}_{|S^{m-1}|}\},$$

where $|S^i|$ is the number of mutant branches in S^i.

To reflect the possibility of a test datum, X, covering e^i_j, the following random variable is defined:

$$\mu^i_j(X) = \begin{cases} 1 & X \ covers \ e^i_j \\ 0 & otherwise \end{cases} \tag{1}$$

Definition 1 (*The Difficulty of Mutant Branch Coverage*)**.** It can be measured by the probability of covering the mutant branch.

R samples $(x_1, x_2, \ldots, x_R)$ are selected from the input domain of the program. Based on $\mu^i_j(X)$ in Eq. (1), the difficulty of covering e^i_j can be expressed as:

$$Dif(e^i_j) = 1 - \frac{\sum_{X \in \{x_1, x_2, \ldots, x_R\}} \mu^i_j(X)}{R} \tag{2}$$

Eq. (2) shows that the lower the probability of covering e^i_j is, the higher the value of $Dif(e^i_j)$ is.

A mutant branch with a high difficulty of coverage is difficult to cover, i.e., ($Dif(e^i_j)$ close to 1), indicating that it may only be executed under specific and rare conditions. On the other hand, a mutant branch with a low difficulty of coverage is easy to cover, i.e.,($Dif(e^i_j)$ close to 0), indicating that it can be executed under most circumstances.

In an MPI program, some mutant branches within the same process may be relevant, while other mutant branches in different processes may also be relevant, as processes communicate with each other through communication statements.

Definition 2 (*The Relevancy Between Mutant Branches*)**.** It is the degree of association or correlation between different mutant branches.

The relevancy between mutant branches represents the probability of executing a mutant branch when another mutant branch has already been executed. We can use the conditional distribution law of $\mu_j^i(X)$ and $\mu_{j'}^{i'}(X)$ to represent the relevancy between e_j^i and $e_{j'}^{i'}$. The probability of $\mu_{j'}^{i'}(X)$ being 1 when $\mu_j^i(X)$=1 is represented as

$$\begin{aligned}\alpha_{(e_j^i,e_{j'}^{i'})} &= probability(\mu_{j'}^{i'}(X)|\mu_j^i(X)=1)\\ &= \frac{\sum_{X\in\{x_1,x_2,\ldots,x_R|\mu_j^i=1\}}\mu_{j'}^{i'}(X)}{\sum_{X\in\{x_1,x_2,\ldots,x_R\}}\mu_j^i(X)}\end{aligned} \tag{3}$$

Eq. (3) shows that the coverage of $e_{j'}^{i'}$ is always accompanied by the coverage of e_j^i.

Note that when $\alpha_{(e_j^i,e_{j'}^{i'})}=1$ and $e_j^i \neq e^{i'}{}_{j'}$, if e_j^i is covered, $e_{j'}^{i'}$ must also be covered. Therefore, $e_{j'}^{i'}$ can be deleted from the program under test without affecting the program's behavior, because any input that triggers e_j^i will also trigger $e_{j'}^{i'}$.

Their relevant matrix of mutant branches is expressed as:

$$\Lambda = \begin{bmatrix} \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots\alpha_{(e_1^i,e_1^i)}\cdots\alpha_{(e_1^i,e^{i'}{}_1)}\cdots\alpha_{(e_1^i,e^{i'}{}_{|S^{i'}|})}\cdots & & & & & & & & & & \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots\alpha_{(e_j^i,e_1^i)}\cdots\alpha_{(e_j^i,e^{i'}{}_1)}\cdots\alpha_{(e_j^i,e^{i'}{}_{|S^{i'}|})}\cdots & & & & & & & & & & \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots\alpha_{(e^i{}_{|S^i|},e_1^i)}\cdots\alpha_{(e^i{}_{|S^i|},e^{i'}{}_1)}\cdots\alpha_{(e^i{}_{|S^i|},e^{i'}{}_{|S^{i'}|})}\cdots & & & & & & & & & & \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \end{bmatrix}$$

Definition 3 (*The Correlogram Between Mutant Branches*)**.** It is a directed graph $G(V,E,W)$, where $V=\{\ldots,e_j^i,\ldots,e_{j'}^{i'},\ldots\}$ is a set of vertices, $E=\{\ldots,\langle e_j^i,e_{j'}^{i'}\rangle,\ldots\}$ is a set of directed edges, and W is a set of weights.

In this correlogram, each vertex corresponds to a mutant branch e_j^i, and each edge $\langle e_j^i,e_{j'}^{i'}\rangle$ represents a directed connection from e_j^i to $e_{j'}^{i'}$. The weight of $\langle e_j^i,e_{j'}^{i'}\rangle$ represents the relevancy between $e_j^i,e_{j'}^{i'}$.

Definition 4 (*The Mutation-Based Path of MPI Program*)**.** A mutation-based path in an MPI program is a sequence of mutant branches that represents a potential execution path through the program. Each node in this path corresponds to a mutant branch.

When an input executes process S^i in an MPI program, a series of covering mutant branches, $e_1^i,\ldots,e_j^i,\ldots,e_{|g_i|}^i$, are covered, forming a subpath, i.e., $g^i=e_1^i,\ldots,e_j^i,\ldots,e_{|g_i|}^i$, where $|g^i|$ is the number of mutant branches, and also called the length of subpath g^i.

Furthermore, a mutation-based path of the MPI program containing m processes, is expressed as $g=g^0\|g^1\|\ldots\|g^i\|\ldots\|g^{m-1}$.

Note that the mutation-based paths differ from traditional paths in that they only contain mutant branches (faults). Mutation-based paths are a form of testing that focus on faulty paths by introducing mutant branches to a program.

3.2. The mutation-based path generation

Our method of mutation-based path generation is outlined in **Algorithm 2**. We have set two goals. The first goal is to generate a small path set that includes all the mutant branches, as shown in lines 3 to 5; this strategy helps reduce the cost associated with test data generation. The second goal is to generate paths that are easier to cover, as demonstrated in lines 9 to 11 and lines 30 to 31; this strategy contributes to the efficiency of generating test data.

Line 3 of **Algorithm 2** shows that the mutant branches are sorted based on the difficulty of mutant branch coverage, and are represented by an ordered set H. Notably, the first element of H corresponds to the hardest mutant branch to cover. Then, in Line 5, to obtain a small path set, we pick mutant branches that are harder to cover as the benchmark mutant branches, and then generate the paths by selecting the mutant branches that are highly relevant to the benchmarks. With this strategy, the first goal (i.e., obtaining a small path set) can be achieved.

Algorithm 2 The mutation-based path generation

Input: H (a mutant branch set)
Output: $Q=\{g_1,g_2,\ldots,g_{|Q|}\}$(the path set)
1: Set $V=\emptyset$, $E=\emptyset$, $W=\emptyset$, and $G(V,E,W)=\emptyset$
2: Calculate $\alpha_{(e_j^i,e_{j'}^{i'})}$ of all mutant branches and obtain Λ
3: Calculate $Dif(e_j^i)$ of all mutant branches and obtain the sorted set, H
4: **repeat**
5: Pick e_j^i that is most difficult to cover from H as the benchmark, and put it into $V=\{e_j^i\}$
6: Set a threshold be Th
7: Record $\alpha_{(e_j^i,e_{j_{k1}}^{i_{k1}})}$ based on benchmark e_j^i from Λ, where $i_{k1}=0,1,\ldots,m-1$; $j_{k1}=1,2,\ldots;j_{k1}\neq j$;
8: **repeat**
9: **if** $\alpha_{(e_j^i,e_{j_{k1}}^{i_{k1}})}>Th$ **then**
10: Put $e_{j_{k1}}^{i_{k1}}$ into $V=V\cup\{e_j^i,e_{j_{k1}}^{i_{k1}}\}$
11: **end if**
12: **until** all mutant branches that are strongly related to e_j^i are placed into $V=\{e_j^i,e_{j_{k1}}^{i_{k1}},\ldots\}$
13: Generate V
14: **repeat**
15: Calculate $\alpha_{(e_j^i,e_{j_{k1}}^{i_{k1}})}$ in $V=\{e_j^i,e_{j_{k1}}^{i_{k1}},\ldots\}$
16: **if** $\alpha_{(e_j^i,e_{j_{k1}}^{i_{k1}})}\neq 0$ **then**
17: Obtain $E_{e_j^i}=E_{e_j^i}\cup\{<e_j^i,e_{j_{k1}}^{i_{k1}}>\}$ and record the weight of the edge in W
18: **end if**
19: **until** all the relevancy between vertices are calculated
20: Generate E and W
21: Generate $G(V,E,W)$
22: **repeat**
23: **if** There are no edges between vertices e_j^i and $e_{j'}^{i'}$ in G **then**
24: Divide e_j^i and $e_{j'}^{i'}$ into different groups, $group_{e_j^i}$ and $group_{e_{j'}^{i'}}$
25: **end if**
26: **if** e_j^i ($e_{j'}^{i'}$) has an edge with a vertex in G **then**
27: this vertex goes into groups $group_{e_j^i}$ ($group_{e_{j'}^{i'}}$)
28: **end if**
29: **until** all vertices in G are judged
30: Obtain the paths based on $group_{e_j^i}$ ($group_{e_{j'}^{i'}}$)
31: Select $g_{e_j^i}$ with the fewest nodes from the feasible paths, and put it into Q
32: Delete mutant branches in $g_{e_j^i}$ from H
33: **until** $H=\emptyset$
34: **Return** Q

Example. We select different statements from the MaxTriangle MPI program (in Fig. 1) to mutate, and obtain 55 non-equivalent mutants. These mutants are then transformed into mutant branches. Next, we set the sample size $R=3000$ and calculate the relevancy between mutant branches based on Eq. (3). If the relevancy of two mutant branches is 1, one of the mutant branches is deleted as they are covered by the same test data. Following this process, only twenty-one mutant branches remain, which are displayed in Fig. 1. We then calculated the difficulty of coverage for each of these twenty-one mutant branches

Table 2
Difficulty of mutant branch coverage.

ID	e_j^i	Dif	ID	e_j^i	Dif	ID	e_j^i	Dif
1	e_{14}^0	0.9997	8	e_1^1	0.9842	15	e_4^0	0.6793
2	e_{13}^0	0.9989	9	e_2^2	0.9842	16	e_1^0	0.6740
3	e_{15}^0	0.9891	10	e_2^3	0.9840	17	e_2^0	0.6737
4	e_1^3	0.9876	11	e_1^2	0.9817	18	e_9^0	0.6695
5	e_3^0	0.9873	12	e_{10}^0	0.9767	19	e_6^0	0.6695
6	e_5^0	0.9848	13	e_{12}^0	0.9758	20	e_{11}^0	0.5453
7	e_2^1	0.9847	14	e_8^0	0.9685	21	e_7^0	0.3387

and sorted them accordingly. The values of $Dif(e_j^i)$ and order for each mutant branch are presented in Table 2, with e_{14}^0 being the hardest mutant branch to cover. The ordered set, H, is

$$H=\{e_{14}^0,e_{13}^0,e_{15}^0,e_1^3,e_3^0,e_5^0,e_2^1,e_1^1,e_2^2,e_2^3,e_1^2,e_{10}^0,e_{12}^0,e_8^0,e_4^0,e_1^0,e_2^0,e_9^0,e_6^0,e_{11}^0,e_7^0\}$$

3.2.1. The correlogram generation

The methods of generating the correlogram between mutant branches are shown in lines 4 to 21 of **Algorithm 1**. Firstly, the most difficult-to-cover mutant branches are picked one by one from H as benchmarks. Subsequently, a correlogram is generated based on a benchmark and Λ.

Lines 4 to 13 of **Algorithm 2** depict the strategies for obtaining the vertices V of the correlogram, G. Let us assume that mutant branch e_j^i is the most difficult to cover, and it is selected as the mutant branch benchmark. Then, we put it into the set of vertices. Next, we scan the row that e_j^i belongs to in Λ and record the relevancy between e_j^i and other mutant branches, $e_{j'}^{i'}$. We set a threshold value, Th. If $\alpha_{(e_j^i,e_{j'}^{i'})} > Th$, $e_{j'}^{i'}$ is placed into V. Then, we continue to compare Th with the relevancy between $e_{j'}^{i'}$ and other mutant branches. In this way, all mutant branches that are strongly related to e_j^i are placed into V. In general, if some mutant branches with high relevancy are combined as the paths, the paths should be easy to cover. With these strategy, the second goal (i.e., generating paths that are easier to cover) can be achieved.

Lines 14 to 21 of **Algorithm 2** depict the strategies for obtaining E, W and G. If $\alpha_{(e_j^i,e_{jk1}^{ik1})} \neq 0$, edge $\langle e_j^i, e_{jk1}^{ik1}\rangle$ goes to a set of edges E. The weight, $\alpha_{(e_j^i,e_{j'}^{i'})}$, of this edge goes into a set of weights W. Finally, we can obtain a correlogram $G(V,E,W)$ based on e_j^i.

Next, based on the example in Fig. 1, we explain the process of generating the correlogram based on the benchmark mutant branches.

Example. e_3^0 is selected from H as the benchmark. Set Th=0.45. We scan the row in Λ that e_3^0 belongs to, and record $\alpha_{e_3^0,e_1^0}$ and $\alpha_{e_3^0,e_6^0}$, which are greater than Th. Then, we scan the rows that e_1^0 and e_6^0 belong to, and record $\alpha_{e_1^0,e_4^0}, \alpha_{e_1^0,e_6^0}, \alpha_{e_1^0,e_7^0}, \alpha_{e_6^0,e_2^0}$ ($> Th$). In this way, we can generate $V_{e_3^0} = \{e_1^0, e_2^0, e_3^0, e_4^0, e_6^0, e_7^0, e_8^0\}$. Based on $V_{e_3^0}$ and Λ, certain branches exhibit zero correlation with each other ($\alpha_{e_2^0,e_1^0} = 0$, $\alpha_{e_6^0,e_7^0} = 0$ and $\alpha_{e_6^0,e_8^0} = 0$), resulting in their inability to form edges. Thus, we obtain the set of edges,

$$E_{e_3^0} = \{\langle e_3^0,e_1^0\rangle, \langle e_3^0,e_6^0\rangle, \langle e_1^0,e_4^0\rangle, \langle e_1^0,e_6^0\rangle, \langle e_1^0,e_7^0\rangle, \langle e_6^0,e_2^0\rangle, \langle e_2^0,e_7^0\rangle, \langle e_2^0,e_4^0\rangle, \langle e_8^0,e_7^0\rangle, \langle e_4^0,e_7^0\rangle, \langle e_3^0,e_7^0\rangle, \langle e_3^0,e_8^0\rangle, \langle e_3^0,e_2^0\rangle, \langle e_3^0,e_4^0\rangle, \langle e_2^0,e_8^0\rangle, \langle e_4^0,e_8^0\rangle, \langle e_1^0,e_8^0\rangle, \langle e_4^0,e_6^0\rangle\}$$

Based on $V_{e_3^0}$ and $E_{e_3^0}$, we generate a correlogram, G, shown in Fig. 4. Solid lines denote directed edges with relevancy (weights) greater than or equal to Th. In contrast, some dashed edges are drawn to represent relevancy between certain mutant branches that is less than Th.

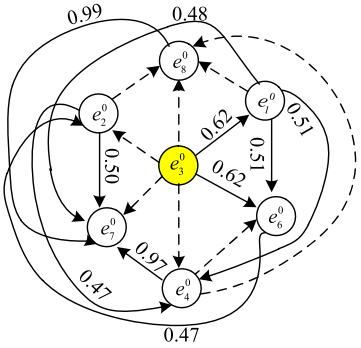


Fig. 4. The generated correlogram.

3.2.2. The path generation

Generally, there are multiple paths obtained based on a correlogram. We select a path that is relatively easy to cover, as it facilitates the generation of test data. The strategies shown in lines 22 to 34 of **Algorithm 2**.

Lines 23 to 29 describe the process of judging whether there exists an edge between any two vertices, e_j^i and $e_{j'}^{i'}$ in G. If there is no edge between them, they will be put into different groups. Next, for each vertex in group $group_{e_j^i}$ or $group_{e_{j'}^{i'}}$, if it has an edge with a vertex in G, this vertex will move into $group_{e_j^i}$ or $group_{e_{j'}^{i'}}$, until all vertices in G have been judged.

Lines 30 to 32 indicate that based on $group_{e_j^i}$, we can obtain several paths according to the order of mutant branches in their processes. Then, we delete non-feasible paths based on Dang et al. (2016). Subsequently, the path with the fewest nodes from the feasible paths is selected as the generated path, denoted as $g_{e_j^i}$, and $g_{e_j^i}$ is put into path set Q. With these strategy, the second goal (i.e., generating paths that are easier to cover) can be also met. Meanwhile, the mutant branches in $g_{e_j^i}$ are deleted from H.

Next, if $H \neq \emptyset$, the most difficult mutant branch to cover is selected as the new benchmark. We repeat the previous steps (lines 4 to 33). If $H = \emptyset$, the path set, $Q = \{g_1, \ldots, g_{e_j^i}, \ldots, g_{|Q|}\}$, is generated, where $|Q|$ is the number of paths.

Example. Based on solid edges between any two vertices of G in Fig. 4, we obtain four groups, i.e., $\{e_1^0, e_3^0, e_4^0, e_6^0\}$, $\{e_2^0, e_3^0, e_4^0, e_6^0\}$, $\{e_1^0, e_3^0, e_4^0, e_7^0, e_8^0\}$, $\{e_2^0, e_3^0, e_4^0, e_7^0, e_8^0\}$. According to these four groups, we can find out two feasible paths, "$e_1^0, e_3^0, e_6^0\|0\|0\|0$" and "$e_2^0, e_3^0, e_4^0, e_7^0, e_8^0\|0\|0\|0$". After that, we select "$e_1^0, e_3^0, e_6^0\|0\|0\|0$ " with fewer nodes as the generated path, i.e., $g_{e_3^0} = e_1^0, e_3^0, e_6^0\|0\|0\|0$. Meanwhile, e_1^0, e_3^0, e_6^0 contained in $g_{e_3^0}$ are deleted from H. Next, the new benchmark is selected from the updated H, and we repeat the above steps. In this way, we obtain the path set $Q = \{g_{e_{14}^0}, g_{e_{13}^0}, g_{e_{15}^0}, g_{e_3^0}, g_{e_5^0}, g_{e_2^1}, g_{e_2^2}, g_{e_2^3}, g_{e_1^2}, g_{e_{10}^0}, g_{e_{12}^0}\}$, which contains all mutant branches, listed in Table 3.

3.3. Model of test data generation for covering the mutation-based path

To generate test data for covering the mutation-based paths in Q, the problem of the multiple path coverage is transformed into a multitask optimization one.

Assume that path $g_l = g_l^0\|\ldots\|g_l^i\|\ldots\|g_l^{m-1}$ is one of the target paths in Q. When an input (a test datum), X, executes an MPI program, we can obtain a covered path, $\vec{g} = \vec{g}^0\|\ldots\|\vec{g}^i\|\ldots\|\vec{g}^{m-1}$. Based on Gong et al. (2021), the similarity between g_l and $\vec{g}$ is defined as the objective function, which can be expressed as:

$$f_l(X) = \frac{\sum_{i=0}^{m-1} \frac{|\vec{g}^i \Delta g_l^i|}{\max(|\vec{g}^i|,|g_l^i|)}}{m} \quad (4)$$

Table 3
The generated paths.

ID	Benchmarks	Path
1	e^0_{14}	$g_{e^0_{14}} = e^0_{14}\|\|e^1_1\|\|0\|\|0$
2	e^0_{13}	$g_{e^0_{13}} = e^0_2, e^0_6, e^0_{13}\|\|0\|\|0\|\|e^3_1$
3	e^0_{15}	$g_{e^0_{15}} = e^0_2, e^0_7, e^0_{15}\|\|0\|\|0\|\|0$
4	e^0_3	$g_{e^0_3} = e^0_1, e^0_3, e^0_6\|\|0\|\|0\|\|0$
5	e^0_5	$g_{e^0_5} = e^0_2, e^0_4, e^0_5, e^0_7, e^0_9, e^0_{11}\|\|0\|\|0\|\|0$
6	e^1_2	$g_{e^1_2} = e^0_1, e^0_4, e^0_7, e^0_8\|\|e^1_2\|\|0\|\|0$
7	e^2_2	$g_{e^2_2} = e^0_7, e^0_9, e^0_{11}\|\|0\|\|e^2_2\|\|0$
8	e^3_2	$g_{e^3_2} = e^0_7, e^0_9, e^0_{11}\|\|0\|\|0\|\|e^3_2$
9	e^2_1	$g_{e^2_1} = e^0_7\|\|0\|\|e^2_1\|\|0$
10	e^0_{10}	$g_{e^0_{10}} = e^0_7, e^0_9, e^0_{10}\|\|0\|\|0\|\|0$
11	e^0_{12}	$g_{e^0_{12}} = e^0_7, e^0_{12}\|\|0\|\|0\|\|0$

Here $|\vec{g}^i \Delta g^i_l|$ represents the number of consecutive identical nodes in $\vec{g}^i$ and g^i_l from front to back, and m represents the number of processes.

Eq. (4) demonstrates that the closer $f_l(X)$ is to its maximum value, the closer the coverage path aligns with the target path. Hence, finding test data to cover the path can be formulated as a problem of maximizing $f_l(X)$, expressed as:

$$\begin{cases} \max f_l(X) \\ s.t. \quad X \in D \end{cases} \tag{5}$$

Where D is the input domain of the program under test. The multitask optimization model of test data generation for covering $g_1, g_2, \cdots, g_{|Q|}$ in Q, is expressed as:

$$\begin{cases} \max f_1(X) \\ s.t. \quad X \in D \end{cases}, \quad \begin{cases} \max f_2(X) \\ s.t. \quad X \in D \end{cases}, \quad \cdots \quad \begin{cases} \max f_{|Q|}(X) \\ s.t. \quad X \in D \end{cases} \tag{6}$$

3.4. Test data generation based on MGA

To solve the multitasking problem stated in Eq. (6), we refer to our previous research (Dang et al., 2021). However, the method we propose differs from prior methods. Specifically, we generate test data for covering multipaths using a multi-population genetic algorithm (MGA) outlined in **Algorithm 3**.

Let the number of subpopulations of MGA be $|Q|$. Each subpopulation, $X^l = \{x^l_1, x^l_2, \cdots, x^l_{Size}\}$ $(l = 1, 2, \ldots, |Q|))$, contains $Size$ individuals, and x^l_k is the kth $(k = 1, 2, \ldots, Size)$ individual. X^l is responsible for generating test data for covering path g_l.

Line 1 of **Algorithm 3** shows that each subpopulation, X^l is initialized. Lines 5 to 12 demonstrate that during each iteration, each individual X^l decides not only whether it is the optimal solution of the lth subtask but also whether it is optimal for other subtasks. Furthermore, lines 13 to 17 illustrate how genetic operations (selection, crossover, and mutation) are utilized for X^l when there is no individual covering path g_l.

In line 14, the fitness function is defined as:

$$fit_l(X) = f_l(X) \tag{7}$$

If an individual x^l_k in X^l makes $fit(x^l_k)$ =1, that is, x^l_k can cover path g_l, and it is the optimal solution of the lth subtasks. Thus, subpopulation X^l stops evolving.

4. Empirical study

To verify the validity of the proposed framework, **PMTDGM**, three groups of experiments are designed.

Algorithm 3 Test data generation for covering multipaths by MGA

Input: $Q = \{g_1, g_2, ..., g_{|Q|}\}$ (a path set), Pop (a population with $|Q|$ subpopulations), ρ (maximum number of iterations)
Output: T(test set)

```
1: Initialize subpopulation X^l = {x^l_1, x^l_2, ···, x^l_Size}, l = 1, 2, ···, |Q|
2: Set count=1
3: while count ≤ ρ or |Q| ≠ 0 do
4:     Set l = 1
5:     for l = 1 to |Q| do
6:         each x^l_k (k = 1, 2, ···, Size) run the program under test
7:         if x^l_k can cover g_l then
8:             Stop the evolution of X^l and save test datum, x^l_k
9:         else
10:            if one of subpopulations can cover g_l(j = 1, 2, ···, |Q|, j ≠ l) then
11:                Stop the evolution and save test datum
12:            end if
13:            if no subpopulation can cover g_l then
14:                Calculate fitness fit_l(X^l)
15:                Perform selection, crossover, and mutation;
16:                Generate new individuals
17:            end if
18:        end if
19:    end for
20:    count = count+1
21: end while
22: Return T
```

4.1. Research questions

RQ1: Are the sizes of the test sets generated by **PMTDGM** small?
The size of the test sets may vary depending on the size of the paths being tested. To obtain a small test set for an MPI program, one of the key technologies in **PMTDGM** is generating the paths by selecting the most "hard-to-kill" mutant branches as the benchmarks. As a comparison, we select some paths by the different methods to generate the test data. Alternatively, test data is generated without covering the paths.

RQ2: To what extent can our paths improve the efficiency of generating mutation-based test data?
The efficiency of generating mutation-based test data improves significantly if the path is easy to cover. The paths generated by **PMTDGM** are designed to be easily covered, contributing to their effectiveness. To evaluate the performance of our paths, we select various paths obtained by different methods and compare the efficiency of generating test data that can cover these paths.

RQ3: To what extent does the multitask method improve the efficiency of generating test data?
To cover the multipaths, we use a multi-tasking method called MGA, where each subpopulation in MGA completes a different subtask. By using MGA, we aim to increase the efficiency of generating test data for mutation testing. In comparison, we also employ the random method (RD) and single population genetic algorithm (SGA) to generate test data.

4.2. Experimental setup

Our experiments are performed on a machine running Microsoft Windows 10, equipped with 2 Intel(R) Core(TM) i5 CPUs and 4 GB of memory. The programming language is MPI+C/C++.

We select six MPI programs to be tested, each with distinct application domains, data types, logic structures, functionalities, and scales. Table 4 provides basic information about these programs, including the number of inputs, processes, and communication statements.

Table 4
Information about the programs under test.

ID	Programs	NO. of inputs	NO. of processes	NO. of communication statements	Function
$S1$	MaxTriangle	2	5	32	Triangle classification
$S2$	Gcd	3	4	15	Greatest common divisor
$S3$	Matrix	2	4	12	Matrix multiplication
$S4$	Compare	6	3	4	Compare array relationships
$S5$	Index	10	5	8	Index with condition
$S6$	Including	2	4	24	Positional relationship of points and polygons

Table 5
Number of tested statements and mutant branches.

ID	NO. of statements	NO. of mutants	NO. of mutant branches
$S1$	9	27	21
$S2$	8	24	19
$S3$	7	25	20
$S4$	12	37	31
$S5$	14	53	42
$S6$	21	85	68

In the experiments, the tested statements are selected based on their statement type, lines of code, and program structure (Gong et al., 2017a). Table 1 displays the mutation operators utilized in the experiments. The mutant branches obtained after removing redundant and equivalent mutants are displayed in the last column of Table 5.

Previous studies have shown that automatic tools may not always be precise in determining equivalent mutants (Jia and H.M., 2008; Kintis et al., 2017). Hence, we manually generated the mutants based on references and our previous research (Dang et al., 2020, 2021; Yao et al., 2014).

4.3. Experimental process

4.3.1. Procedure for RQ1

To compare the sizes of test sets, four methods are designed.

PMTDGM: The proposed method presented in Section 3. It selects the most "hard-to-kill" mutant branches as benchmarks to obtain the path set, Q.

EPMTDG: The mutant branches that are **E**asy to cover are selected as benchmarks, and the **P**ath set, Q_e, is generated based on the method outlined in Section 3.1. The size of Q_e is $|Q_e|$. Finally, **M**utation-based **T**est **D**ata are **G**enerated to cover Q_e.

RPMTDG: This method **R**andomly selects a **P**ath set, and **M**utation-based **T**est **D**ata are **G**enerated to cover it. Twenty feasible path sets containing all the mutant branches are generated, and three are randomly selected (i.e., Q_1, Q_2, and Q_3). The sizes of these path sets are less than $|Q|$ ($|Q_1| < |Q|$), equal to $|Q|$ ($|Q_2| = |Q|$), and greater than $|Q|$ ($|Q_3| > |Q|$).

TMTDG: This is a **T**raditional **M**utation-based **T**est **D**ata **G**eneration method, where test data covering mutant branches are directly generated without covering the paths.

4.3.2. Procedure for RQ2

To validate the effectiveness of Q generated by **PMTDGM**, we generate test data to cover Q_e, Q_1, Q_2, and Q_3. To eliminate algorithmic randomness, the MGA is run independently 30 times for each path set, and the results are averaged. We compare two indexes: time consumption and the number of iterations. Generally, a path that requires less time and fewer iterations to cover is considered easier to cover.

The number of subpopulations in MGA equals the size of the path set, and the number of individuals in each subpopulation is 5. The genetic operations are roulette selection, single point crossover, and single point mutation. The crossover and mutation probabilities are 0.9 and 0.3, respectively. The algorithm has two termination criteria: one is when the desired test data for the target paths are generated, while the other is when the population has evolved for the maximum number of iterations (3000).

Table 6
Test set sizes obtained by different methods.

ID	PMTDGM	EPMTDG	RPMTDG			TMTDG
	Q	Q_e	Q_1	Q_2	Q_3	
S1	11	12	10	11	12	12
S2	12	15	11	12	15	14
S3	9	12	9	11	12	9
S4	13	16	12	13	16	16
S5	14	17	14	15	18	14
S6	19	23	18	19	24	20

4.3.3. Procedure for RQ3

In this experiment, we compare MGA in **PMTDGM** with RD and SGA. The effectiveness of the three methods is measured using three metrics: success rate of test data generation (SR), time consumption, and the number of iterations. To eliminate the randomness of the algorithms, we run each algorithm independently 30 times and take the average of the results.

SR (Success Rate) denotes the ratio of the number of times that the test data is successfully generated to the total number of times the algorithm is run.

Note that the population number of SGA is 1, and all other parameters are identical to those in MGA. The parameter settings of MGA remain the same as in the second experiment.

5. Experimental results

5.1. Answer to RQ1: The size of test sets

Referring to the example $S1$ in Fig. 1, the paths obtained by the three methods are as follows:

Based on **PMTDGM**,

$$Q = \{g_{e_{14}^0}, g_{e_{13}^0}, g_{e_{15}^0}, g_{e_3^0}, g_{e_5^0}, g_{e_2^1}, g_{e_2^2}, g_{e_2^3}, g_{e_1^2}, g_{e_{10}^0}, g_{e_{12}^0}\}$$

are shown in Table 3.

Based on **EPMTDG**, we first select e_7^0, which is easiest to cover, to draw the correlogram of mutant branches, and generate Q_e:

$$\begin{aligned}
Q_e = \{&\{e_1^0, e_4^0, e_7^0 \| 0 \| 0 \| e_2^3\},\\
&\{e_2^0, e_4^0, e_7^0 \| 0 \| 0 \| e_1^3\},\\
&\{e_2^0, e_4^0, e_5^0, e_7^0, e_9^0, e_{11}^0 \| 0 \| e_1^2 \| 0\},\\
&\{e_7^0 \| 0 \| e_2^2 \| 0\},\\
&\{e_7^0, e_9^0, e_{10}^0 \| 0 \| 0 \| e_2^3\}\\
&\{e_2^0, e_4^0, e_7^0, e_{12}^0 \| 0 \| e_1^2 \| e_2^3\},\\
&\{e_1^0, e_5^0, e_7^0, e_8^0 \| e_2^1 \| 0 \| 0\},\\
&\{e_2^0, e_{11}^0 \| e_1^1 \| 0 \| 0\},\\
&\{e_2^0, e_3^0, e_7^0, e_8^0, e_{15}^0 \| 0 \| 0 \| e_1^3\},\\
&\{e_{14}^0 \| e_1^1 \| 0 \| 0\}\\
&\{e_1^0, e_7^0, e_{13}^0 \| 0 \| 0 \| 0\},\\
&\{e_6^0, e_{12}^0 \| 0 \| 0 \| e_1^3\}\}
\end{aligned}$$

Based on **RPMTDG**, we obtain paths Q_1, Q_2 and Q_3, $|Q_1| < |Q|$, $|Q_2| = |Q|$, $|Q_3| > |Q|$, shown in Fig. 5. Note that each path set must contain all 21 mutant branches.

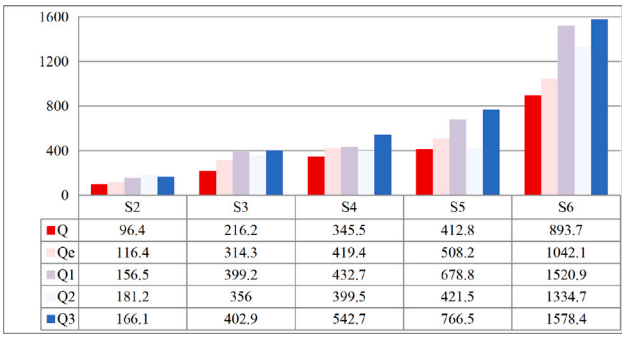
Further, we obtain test sets covering Q, Q_e, Q_1, Q_2, Q_3 respectively for $S1$. Table 6 lists the sizes of the test sets.

$Q_1 = \{\{e_2^0, e_4^0, e_7^0, e_{13}^0 \| 0 \| 0 \| 0\},$
$\{e_1^0, e_4^0, e_5^0, e_7^0, e_9^0, e_{11}^0 \| 0 \| 0 \| e_2^3\},$
$\{e_1^0, e_6^0, e_{11}^0 \| 0 \| e_1^2 \| 0\},$
$\{e_7^0, e_9^0, e_{10}^0 \| 0 \| 0 \| e_1^3\},$
$\{e_7^0, e_8^0 \| 0 \| 0 \| e_2^3\},$
$\{e_2^0, e_{12}^0, e_{15}^0 \| e_1^1 \| 0 \| 0\},$
$\{e_2^0, e_6^0 \| 0 \| e_2^2 \| 0\},$
$\{e_1^0, e_4^0, e_7^0, e_8^0, e_{12}^0 \| e_2^1 \| 0 \| 0\},$
$\{e_2^0, e_3^0, e_4^0, e_7^0, e_8^0 \| 0 \| e_1^2 \| 0\},$
$\{e_{14}^0 \| e_1^1 \| 0 \| 0\}\}$

$Q_2 = \{\{e_{14}^0 \| e_1^1 \| 0 \| 0\},$
$\{e_1^0, e_4^0, e_7^0, e_8^0 \| e_2^1 \| 0 \| e_2^3\},$
$\{e_1^0, e_3^0, e_6^0, e_{10}^0 \| 0 \| 0 \| 0\},$
$\{e_1^0, e_6^0, e_{13}^0 \| 0 \| 0 \| 0\},$
$\{e_7^0, e_6^0, e_{10}^0 \| 0 \| e_2^2 \| 0\},$
$\{e_7^0, e_8^0 \| 0 \| e_1^2 \| e_2^3\},$
$\{e_7^0, e_{12}^0 \| 0 \| e_2^1 \| 0\},$
$\{e_2^0, e_6^0 \| e_2^1 \| 0 \| e_1^3\},$
$\{e_2^0, e_{11}^0 \| e_1^1 \| e_1^1 \| 0\},$
$\{e_1^0, e_4^0, e_5^0, e_6^0 \| e_1^1 \| 0 \| 0\},$
$\{e_2^0, e_7^0, e_{15}^0 \| 0 \| 0 \| 0\}\}$

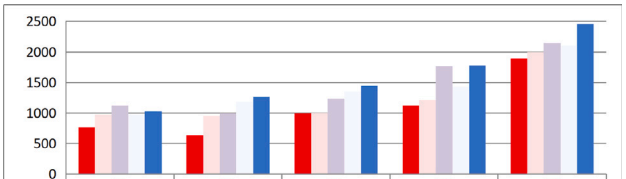
$Q_3 = \{\{e_2^0, e_4^0, e_7^0 \| 0 \| 0 \| e_2^3\},$
$\{e_1^0, e_4^0, e_7^0 \| 0 \| 0 \| e_1^3\},$
$\{e_7^0 \| 0 \| e_2^2 \| 0\},$
$\{e_2^0, e_4^0, e_7^0, e_{12}^0 \| 0 \| e_1^2 \| 0\},$
$\{e_1^0, e_4^0, e_5^0, e_7^0, e_8^0 \| e_2^1 \| 0 \| 0\},$
$\{e_2^0, e_{11}^0 \| e_1^1 \| 0 \| 0\},$
$\{e_2^0, e_3^0, e_7^0, e_8^0, e_{15}^0 \| 0 \| 0 \| 0\},$
$\{e_{14}^0 \| e_1^1 \| 0 \| 0\},$
$\{e_7^0, e_{13}^0 \| 0 \| 0 \| 0\},$
$\{e_2^0, e_4^0, e_5^0, e_7^0, e_9^0, e_{11}^0 \| 0 \| e_1^2 \| 0\},$
$\{e_2^0, e_6^0, e_{12}^0 \| 0 \| 0 \| e_1^3\},$
$\{e_7^0, e_9^0, e_{10}^0 \| 0 \| 0 \| e_2^3\}\}$

Fig. 5. The path sets generated by **RPMTDG**.



	S2	S3	S4	S5	S6
Q	96.4	216.2	345.5	412.8	893.7
Qe	116.4	314.3	419.4	508.2	1042.1
Q1	156.5	399.2	432.7	678.8	1520.9
Q2	181.2	356	399.5	421.5	1334.7
Q3	166.1	402.9	542.7	766.5	1578.4

Fig. 6. Mean of time consumption for covering the path sets.



	S2	S3	S4	S5	S6
Q	771.4	634.3	997.1	1123.4	1897.5
Qe	979.3	949.8	1005.3	1213.6	2001.9
Q1	1120.6	999.3	1231.5	1767.2	2145.1
Q2	987.8	1180.4	1344.1	1432.2	2101.9
Q3	1026.5	1260.3	1450.4	1777.6	2456.4

Fig. 7. Mean of iterations for covering the path sets.

Table 7
The cost of test data generation for covering paths by the different methods for S1.

Sets	Time consumption			Iterations		
	Max	Min	Mean	Max	Min	Mean
Q	369.6	10.3	78.9	2510.1	17.3	604.1
Q_e	384.7	23.5	99.3	2434.1	16.4	999.4
Q_1	410.9	112.3	183.0	1994.0	154.6	1016.3
Q_2	864.5	102.9	225.8	2271.5	112.9	937.9
Q_3	388.6	85.3	162.3	2606.7	75.4	1379.9

As depicted in Fig. 5 for $S1$, it is evident that the path set size, Q, obtained by **PMTDGM** is comparatively smaller. Likewise, the sizes of test sets for other programs ($S2$-$S6$) are shown in Table 6. As can be seen, the test sets produced by Q of **PMTDGM** are much smaller in size compared to Q_e, Q_3, and **TMTDG** for all the programs.

It is worth noting that **RPMTDG**, as a random method of path generation, introduces a large amount of randomness, which may cause it to generate smaller set sizes than **PMTDGM**. In this experiment, we use randomly generated path sets Q_1, Q_2, provided that $Q_1| < |Q|, |Q_2| = |Q|$. These path sets are not only used to compare the size of the test sets in this experiment, but also for the second set of experiments, when the size of the path set Q_1, Q_2 generated by **RPMTDG** is comparable to Q of **PMTDGM**, we intend to verify that the greater efficiency of generating test data that covers Q compared to covering other paths.

5.2. Answer to RQ2: Efficiency of the path sets generated by PMTDGM

Referring to $S1$ again, Table 7 shows the minimum, maximum, and mean values of time consumption and iterations times for different path sets. It can be seen that Q_e and Q have the minimum time and the number of iterations when generating test sets. However, Q_e has a larger test set than Q, as shown in Table 6. Moreover, comparing Q_1, Q_2 and Q_3, Q achieve reductions of 56.9%, 65.1%, and 51.41% in terms of time consumption, and it decrease by 40.6%, 35.6%, and 56.2% in terms of iterations times. Note that, $|Q_1|$ =10 of $S1$ is the smallest, but the paths in Q_1 are difficult to cover. In summary, our path Q demonstrates superiority in terms of time consumption and the number of iterations.

For the other five programs ($S2$-$S6$), the mean values of time consumption and the number of iterations are illustrated in Fig. 6 and Fig. 7. By comparing Q_e, Q_1, Q_2 and Q_3, it can be concluded that Q has less time consumption and fewer the number of iterations.

To verify the significant superiority of Q, the Mann–Whitney U test is adopted based on the data in Fig. 6 and Fig. 7. Let the significance level of the U test be 0.05. The results of the Mann–Whitney U test indicate that Q is significantly superior to Q_1, Q_2 and Q_3 in terms of time consumption and iterations times for six MPI programs.

In summary, the experimental results show that our path sets(Q), generated by **PMTDGM**, outperform the path sets obtained by different methods, not only due to the shorter time consumption but also because of fewer iterations times when generating test data. In other words, our path sets are easy to cover. As a result, our paths are beneficial to improving the efficiency of mutation-based test data generation.

Table 8
SR of generating test data covering Q by three methods for all programs.

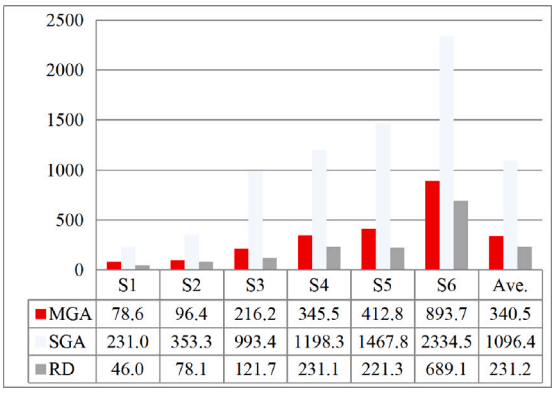
ID	MGA			SGA			RD		
	Smax	Smin	Smean	Smax	Smin	Smean	Smax	Smin	Smean
S1	100	93.33	98.79	100	93.33	98.79	100	46.67	78.79
S2	100	90.00	98.05	100	93.33	98.88	100	43.33	75.55
S3	100	86.67	96.67	100	86.67	96.67	100	46.67	85.19
S4	100	83.33	95.13	100	93.33	94.87	96.67	33.33	70.56
S5	100	83.33	95.00	100	83.33	95.00	96.67	0	69.76
S6	100	83.33	95.96	100	83.33	94.39	100	0	69.00
Ave.	100	86.67	96.60	100	88.89	96.43	98.89	37.78	74.81

5.3. Answer to RQ3: Efficiency of the multitask method by *PMTDGM*

Table 8 presents the success rate (SR) of test data generation for covering Q based on MGA, SGA, and RD for all programs under test. In the table, "Smax" represents the maximum value of SR for finding the test data, "Smin" represents the minimum value of SR, and "Smean" represents the average value of SR of all paths.

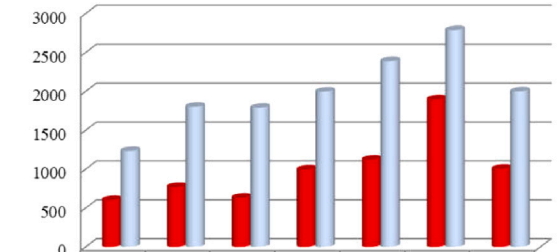
The last row of Table 8 lists the average success rates of all paths, denoted as Ave. For instance, when MGA is run 30 times, among the 11 paths in Q, eight paths find the test data, and "Smax" is 100% (30/30*100%). A path can successfully finds test data 28 times, and "Smin" is 93.33% (28/30*100%). Two paths successfully find test data 29 times, so the success rate is 96.67%(29/30*100%). Therefore, "Smean" is 98.79% (8*100%+2*96.67%+1*93.33%) for the 11 paths in Q, which is listed in the first row of Table 8.

Figs. 8 and 9 illustrate the time consumption and the number of iterations of RD, SGA, and MGA in **PMTDGM**. In the experiment, to eliminate randomness, we run each algorithm 30 times when generating test data and take the average of these results.



	S1	S2	S3	S4	S5	S6	Ave.
MGA	78.6	96.4	216.2	345.5	412.8	893.7	340.5
SGA	231.0	353.3	993.4	1198.3	1467.8	2334.5	1096.4
RD	46.0	78.1	121.7	231.1	221.3	689.1	231.2

Fig. 8. Time consumption by the different methods.



	S1	S2	S3	S4	S5	S6	Ave.
MGA	604.1	771.4	634.3	997.1	1123.4	1897.5	1004.6
SGA	1234.4	1804.6	1790.5	1998.4	2389.5	2789.2	2001.1

Fig. 9. Iteration times by the different methods.

In terms of time consumption, Fig. 8 shows that SGA takes 1096.4 ms, which is 3.21 times greater as MGA (340.5 ms). It can be observed that RD takes the shortest time to generate test data for each program, however, its SR values are the lowest (refer to Table 8).

Note that the evolution strategy of MGA and SGA is similar, as both evolve individuals iterate until test data is found. Hence, we compare their the number of iterations, which are shown in Fig. 9. On average, SGA requires 2001.1 iterations, which is 1.99 times greater than that of MGA (1004.6). These results demonstrate that MGA has better performance than SGA when generating test data in PMTDGM.

Let the significance level of the Mann–Whitney U test be 0.05. The results of Mann–Whitney U show that MGA in **PMTDGM** is significantly superior to RD and SGA in terms of time consumption and the number of iterations.

In summary, we can answer RQ3 by stating that the multitask method (MGA) in PMTDGM improves the efficiency of test data generation based on the results of the experiments.

6. Threats to validity

The first threat to the validity arises from the uncertainty surrounding the size of the data sample when calculating the difficulty of mutant branch coverage and the relevancy between mutant branches. In the experiments, a total of R samples are selected from the input domain of the program. If the value of R is too small, the calculated difficulty of mutant branch coverage and relevancy metrics may be unreasonable and unreliable, while a value of R that is too large may increase the computational complexity of the algorithm. Therefore, in the study, the value of R was determined based on the number of mutant branches and complexity of the programs under test.

The second threat pertains to the generated correlogram, which may vary depending on the chosen value of threshold Th for a given program under test. In the study, the value of Th is determined based on experimental results and experience. However, determining the optimal threshold value is beyond the scope of the current study.

Finally, the potential threats related to the parameters of the algorithms, such as the population size of MGA in **PMTDGM**, and crossover and mutation probabilities in genetic algorithms. Currently, there are no established methods for determining appropriate values for these parameters. In our experiments, reasonable parameter values are selected based on prior research and experience.

7. Related work

7.1. Parallel program testing

Parallel programs pose a greater variety of potential risks and errors than serial programs. Common issues with parallel programs include state races, data races, and deadlocks, making software testing of parallel programs significantly more challenging.

Software testing for parallel programs can be performed using one of three methods, namely static analysis, dynamic analysis, and a mixed-method approach (Hamlet, 1994; Souza et al., 2014b). Static analysis involves analyzing the code without executing the program under test to identify scenarios that may lead to errors during execution. For example, Engler et al. (Engler and Ashcraft, 2003) developed RacerX, a tool for detecting data races and deadlocks in large-scale concurrent systems using flow-sensitive interprocedural analysis. Christakis et al. (Christakis and Sagonas, 2003), on the other hand, built a communication graph for a program by scanning its code to detect message passing errors such as deadlocks and race conditions.

Model checking is another heavy-weight static checking method that involves searching for all possible program behaviors to determine whether the program fails, based on a model of the program under test. Koppol et al. (2002) modeled a parallel program as a symbolic transformation system and proposed incremental testing methods. Flanagan et al. (Flanagan and Godefroid, 2005) proposed a method of dynamic partial-order reduction for model checking software, while Vakkalanka et al. (2008) developed ISP, a model detection tool for MPI programs, building on this approach.

Dynamic testing involves executing the program under test, controlling its execution process, and predicting its behavior or detecting potential faults. Carver et al. (Carver and Lei, 2010) proposed a distributed reachability testing method for concurrent programs, which allows multiple test sequences to be executed simultaneously to reduce test time. Souza et al. (2013) developed coverage testing criteria specifically for parallel programs by utilizing their unique characteristics.

7.2. Mutation testing

Mutation testing is a fault-based testing technique. A program that contains one or more mutant statements becomes a mutant. If a test distinguishes the behavior of a mutant from the source program, we say that the mutant is killed; otherwise, we say that it is alive (Harman et al., 2011).

Mutation scores can be used to measure the effectiveness of a test set and its ability to detect faults (Dave and Agrawal, 2016). The higher the mutation score is, the stronger the capability of a test suite to detect faults is. Generally speaking, identifying equivalent mutants is a manual task that is partially addressed through static heuristics. The idea (Grun et al., 2009) is that mutants that are not killed by the tests but are capable of causing differences in the program state are likely to be killable.

A large number of mutants can lead to high test costs, which is the main challenge of practical application for mutation testing. Mutant reduction strategies aim to select representative subsets from given sets of mutants, including sampling mutation, selection mutation, and high order mutation.

Sampling mutation is a mutant reduction technique that involves randomly selecting a subset of mutants for testing instead of testing all mutants. Zhang et al. (2010) found that selecting 50% of the mutants from the mutant set and applying corresponding test data was able to kill over 99% of the mutants. In addition, by selectively applying a subset of the available mutation operators, the generation of mutants can be optimized, which helps to reduce the cost of mutation testing while maintaining its effectiveness. For instance, Offutt et al. (1996) proposed a basic set of mutation operators that includes five operator sets: LCR, ROR, AOR, ABS, UOI.

In recent years, higher-order mutation testing has gained attention due to its ability to not only reflect complex faults in real software, but also reduce the number of mutants. Jia et al. (Jia and Harman, 2008) introduced the concepts of high-order mutants and first-order mutants, and used higher order mutation testing to construct subtle faults.

There are also many mutation testing tools available for parallel programs. Kusano et al. (Kusano and Wang, 2013) developed a tool for generating high-quality mutants for multi-threaded programs. To reduce the execution time of mutation testing, Gligoric et al. (2010) created mutation testing tools based on the information of executing multi-threaded code. Wu et al. (Wu and Kaiser, 2011) designed mutation operators related to concurrency and synchronization for the Java language. Additionally, Bradbury et al. (2006) created some new mutation operators for object-oriented programming using Java language.

Efficiently generating test data is a crucial aspect of mutation testing. Currently, there are three main families of methods for mutation-based test data generation: constraint-based test generation (CBT) (DeMillo and Offutt, 1991), dynamic symbolic execution (Papadakis and Malevris, 2010), and search-based test generation (Zhang and Gong, 2013; Dang et al., 2020). A popular search-based approach is the use of evolutionary algorithms. Souza et al. (2014a) reviewed 19 major papers related to mutation-based test data generation and found that genetic algorithms were used in six of them. Fraser et al. (Fraser and Zeller, 2012) employed a genetic algorithm to generate test data for detecting faults in Java object-oriented classes.

8. Conclusions and future work

This paper addresses the challenge of mutation testing for parallel programs and presents a new testing framework, **PMTDGM**, to improve the efficiency of software testing for these programs. To evaluate the performance of **PMTDGM**, we tested six MPI programs. Results showed that **PMTDGM** generated relatively small test sets, and the efficiency of test data generation was high. MGA in **PMTDGM** also improved the performance of test data generation. These findings suggest that the mutation-based path generation method and the evolutionary algorithm significantly enhance the effectiveness and efficiency of mutation testing for parallel programs.

However, there are still some areas for improvement. The proposed method is based on traditional mutation operators of serial programs and does not mutate communication statements in parallel programs. We can incorporate special mutation operators designed for parallel programs based on other researchers (Sen, 2009; Silva et al., 2012). Finally, we will explore the use of other artificial intelligence methods to generate test data for detecting faults in parallel programs with a goal of improving their reliability.

CRediT authorship contribution statement

Xiangying Dang: Conceptualization, Methodology, Software, Investigation, Formal analysis, Writing – original draft. **Jinyong Wang:** Data curation, Writing – original draft. **Dunwei Gong:** Investigation, Supervision, Funding acquisition, Resources. **Xiangjuan Yao:** Resources, Supervision, Visualization, Writing – review & editing. **Changqing Wei:** Resources, Supervision, Writing – review & editing. **Biao Xu:** Visualization, Software, Validation, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This work is jointly supported by National Natural Science Foundation of China (No. 42230704), Major Project of Natural Science Research of the Jiangsu Higher Education Institutions of China under Grant (No. 21KJA520006), Xuzhou Science and Technology Plan Project under Grant (No. KC21007), and Natural Science Foundation of Guangdong Province of China under Grant (No. 2021A1515011709).

References

Akhmetova, D., Cebamanos, L., Iakymchuk, R., Rotaru, T., Rahn, M., Markidis, S., Laure, E., Bartsch, V., Simmendinger, C., 2017. Interoperability of Gaspi and Mpi in Large Scale Scientific Applications. Springer.

Bradbury, J.S., Cordy, J.R., Dingel, J., 2006. Mutation Operators for Concurrent Java (J2SE 5.0). IEEE.

Carver, R., Lei, Y., 2010. Distributed reachability testing of concurrent programs. Concurr. Comput.: Pract. Exper. 22 (18), 2445–2466.

Christakis, M., Sagonas, K., 2003. Detection of Asynchronous Message Passing Errors Using Static Analysis. ACM.

Dang, X.Y., Gong, D.W., Yao, X.J., 2016. Feasible path generation of weak mutation testing based on statistical analysis. Chin. J. Comput. Sci. 39 (11), 2355–2371.

Dang, X.Y., Gong, D.W., Y.Yao, X., 2021. Enhancement of mutation testing via fuzzy clustering and multi-population genetic algorithm. IEEE Trans. Softw. Eng..

Dang, X., Yao, X., Gong, D.W., 2020. Efficiently generating test data to kill stubborn mutants by dynamically reducing the search domain. IEEE Trans. Reliab. 69 (1), 334–348.

Dave, M., Agrawal, R., 2016. Mutation Testing and Test Data Generation Approaches: A Review. Springer and Singapore.

DeMillo, R.A., Lipton, R.J., Sayward, F.G., 1978. Hints on test data selection: Help for the practicing programmer. Computer 11 (4), 34–41.

DeMillo, R.A., Offutt, A.J., 1991. Constraint-based automatic test data generation. IEEE Trans. Softw. Eng. 17 (9), 900–910.

Engler, D., Ashcraft, K., 2003. Racerx: Effective, Static Detection of Race Conditions and Deadlocks. ACM.

Flanagan, C., Godefroid, P., 2005. Dynamic Partial-Order Reduction for Model Checking Software. ACM.

Fraser, G., Zeller, A., 2012. Mutation-Driven Generation of Unit Tests and Oracles, Vol. 38, No. 2. pp. 278–292.

Gligoric, M., Jagannath, V., Marinov, D., 2010. MuTMuT: Efficient Exploration for Mutation Testing of Multithreaded Code. IEEE.

Gong, D.W., Chen, Y.W., Tian, T., 2016. Weak mutation testing and its transformation for message passing parallel programs. J. Softw. 27 (8), 2008–2024.

Gong, D.W., Pan, F., Tian, T., Yang, S., Meng, F.L., 2020. A feedback-directed method of evolutionary test data generation for parallel programs. Inf. Softw. Technol. 124 (2), 6317–6332.

Gong, D.W., Qin, B., Tian, T., 2017a. Selecting objects to be mutated based on statement importance. Acta electronica sinica 45 (6), 1518–1522.

Gong, D., Sun, B., Yao, X., Tian, T., 2021. Test data generation for path coverage of mpi programs using saeo. ACM Trans. Softw. Eng. Methodol. 30 (2), 1–37.

Gong, D., Zhang, G.J., Yao, X.J., Fan, L.M., 2017b. Mutant reduction based on dominance relation for weak mutation testing. Inf. Softw. Technol. 81 (4), 82–96.

Gropp, W., Lusk, E., Doss, N., Skjellum, A., 1996. A high-performance,portable implementation of the MPI message passing interface standard. Parallel Comput. 22 (6), 789–828.

Grun, B., Schuler, D., Zeller, A., 2009. The Impact of Equivalent Mutants. IEEE.

Hager, G., Wellein, G., 2010. Introduction to high performance computing for scientists and engineers. Computer 11 (4), 34–41.

Hamlet, R.G., 1977. Testing programs with the aid of a compiler. IEEE Trans. Softw. Eng. 4, 279–290.

Hamlet, R.G., 1994. Softw. qual., software process, and software testing. Adv. Comput. 41 (8), 191–229.

Harman, M., Jia, Y., Langdon, W.B., 2011. Strong Higher Order Mutationbased Test Data Generation. ACM.

Howden, W.E., 1982. Weak mutation testing and completeness of test sets. IEEE Trans. Softw. Eng. 8 (4), 371–379.

Jia, Y., Harman, M., 2008. Constructing Subtle Faults Using Higher Order Mutation Testing. IEEE.
Jia, Y., Harman, M., 2011. An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. 37 (5), 649–678.
Jia, Y., H.M., 2008. MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language. IEEE.
Kintis, M., Papadakis, M., Papadopoulos, A., 2017. How effective are mutation testing tools ? an empirical analysis of java mutation testing tools with manual analysis and real faults. Empir. Softw. Eng. 8 (1), 1–38.
Koppol, P., Carver, R., Tai, K., 2002. Incremental integration testing of concurrent programs. IEEE Trans. Softw. Eng. 28 (6), 607–623.
Kusano, M., Wang, C., 2013. CCmutator: A Mutation Generator for Concurrency Constructs in Multithreaded C/C++ Applications. IEEE and ACM.
Nishtha, J., Bharti, S., Shweta, R., 2017. Systematic literature review on search based mutation testing. e-Inf. Softw. Eng. 11 (1), 59–76.
Offutt, A.J., King, K.N., 1987. A fortran 77 interpreter for mutation analysis. Autom. Softw. Eng. 22 (7), 177–188.
Offutt, A., Lee, A., Rothermel, G., Untch, R., Zapf, C., 1996. An Experimental Determination of Sufficient Mutant Operators. ACM.
Pan, F., Gong, G.W., Tian, T., Yao, X.J., Li, Y., 2021. Path similarity-based scheduling sequence sorting for multi-path coverage of parallel programs. Sci. Sinica Inf. 51 (4), 565–581.
Papadakis, M., Kintis, M., Zhang, J., 2019. Mutation testing advances: an analysis and survey. Adv. Comput. 112, 275–378.
Papadakis, M., Malevris, N., 2010. Automatic Mutation Test Case Generation Via Dynamic Symbolic Execution. IEEE.
Papadakis, M., Malevris, N., 2011. Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing. Softw. Qual. 19 (4), 691–723.
Papadakis, M., Malevris, N., 2012. Mutation based test case generation via a path selection strategy. Inf. Softw. Technol. 54 (9), 915–932.
Papadakis, M., Malevris, N., Kallia, M., 2010. Towards Automating the Generation of Mutation Tests. ACM.
Protopopov, B.V., Skjellum, A., 1996. A multithreaded message passing interface (MPI) architecture: Performance and program issues. J. Parallel Distrib. Comput. 61 (4), 449–466.
Sen, A., 2009. Mutation Operators for Concurrent System C Designs. IEEE.
Silva, R.A., Souza, S.R.S., De-Souza, P.S.L., 2017. A systematic review on search based mutation testing. Inf. Softw. Technol. 81 (1), 19–35.
Silva, R.A., Souza, S.R.S., Souza, P.S.L., 2012. Mutation Operators for Concurrent System C Designs. IEEE.
Souza, F.C., Papadakis, M., Durelli, V.H.S., Delamaro, M.E., 2014a. Test Data Generation Techniques for Mutation Testing: A Systematic Mapping. Luxembourg:University of Luxembourg Interdisciplinary Center for Security, Reliability and Trust.
Souza, P., Souza, S., Zaluska, E., 2013. Structural testing for message-passing concurrent programs an extended test model. Concurr. Comput.: Pract. Exper. 25 (18), 149–158.
Souza, P.S., Souza, S.R., Zaluska, E., 2014b. Structural testing for message-passing concurrent programs: an extended test model. Concurr. Comput.: Pract. Exper. 26 (1), 21–50.
Souza, S., Vergilio, S.R., Souza, P., Simao, A., Hausen, A.C., 2008. Structural testing criteria for message passing parallel programs. Concurr. Comput.: Pract. Exper. 20 (16), 1893–1916.
Tian, T., Gong, D., 2016. Test data generation for path coverage of message-passing parallel programs based on co-evolutionary genetic algorithms. Autom. Softw. Eng. 23 (3), 469–500.
Tokumoto, S., Yoshida, H., Sakamoto, K., Honiden, S., Mu, V.M., 2016. MuVM: Higher Order Mutation Analysis Virtual Machine for C. IEEE International Symposium on Software Reliability Engineering.
Vakkalanka, S., De Lisi, M., Gopalakrishnan, G., Kirby, R., Thakur, R., Gropp, W., 2008. Implementing Efficient Dynamic Formal Verification Methods for MPI Programse. Springer.
Wegener, J., Bareseland, A., Sthamer, H., 2001. Evolutionary test environment for automatic structural testing. Inf. Softw. Technol. 43 (14), 841–851.
Wu, L.L., Kaiser, G.E., 2011. Empirical Study of Concurrency Mutation Operators for Java. Department of Computer Science and Columbia University.
Yao, X.J., Gong, D.W., 2014. Genetic algorithm-based test data generation for multiple paths via individual sharing. Comput. Intell. Neurosci. 29 (1), 1–13.
Yao, X.J., Harman, M., Jia, Y., 2014. A Study of Equivalent and Stubborn Mutation Operators Using Human Analysis of Equivalence. ACM.
Zhang, Y., Gong, D.W., 2013. Evolutionary generation of test data for paths coverage based on scarce data capturing. Chinese J. Comput. 36 (36), 2429–2440.
Zhang, G.J., Gong, D.W., Yao, X.J., 2015. Test case generation based on mutation analysis and set evolution. Chinese J. Comput. 38 (11), 2318–2331.
Zhang, L., Hou, S., Hu, J., Tao, X., Mei, H., 2010. Is Operator-Based Mutant Selection Superior to Random Mutant Selection? ACM.

Xiangying Dang received the Ph.D. degree in control theory and control engineering from the China University of Mining and Technology in 2020. She is a professor at the School of Information Engineering (School of Big Data), Xuzhou University of Technology. Her primary research interests encompass software engineering-based search, mutation testing and analysis.

Jinyong Wang received the Ph.D. degree in software engineering from the Nanjing University of Aeronautics and Astronautics in 2022. He is with the School of Information Engineering (School of Big Data), Xuzhou University of Technology. His research interests include formal methods, intelligent algorithms, and AI safety.

Dunwei Gong received the Ph.D. degree in control theory and control engineering from the China University of Mining and Technology in 1999. He is a professor at the School of Information Science and Technology, Qingdao University of Science and Technology. His primary research interests focus on intelligent optimization and control.

Xiangjuan Yao received the Ph.D. degree in control theory and control engineering from the China University of Mining and Technology in 2011. She is a professor at the School of Mathematics, the China University of Mining and Technology. Her primary areas of research involve intelligent optimization and search-based software testing.

Changqing Wei received the master's degree in Applied Mathematics from China University of Mining and Technology in 2020. Now he is a PhD student in the School of Mathematics, China University of Mining and Technology. His main research interests include search-based software testing.

Biao Xu received the Ph.D. degree in control theory and control engineering from the China University of Mining and Technology in 2019. He is a lecturer at the College of Engineering, Shantou University. His primary research interests encompass multiobjective evolutionary optimization and dynamic scheduling.