Improving Test Data Generation for MPI Program Path Coverage With FERPSO-IMPR and Surrogate-Assisted Models

***Abstract*—Message passing interface (MPI) is a powerful tool for parallel computing, originally designed for high-performance computing on massively parallel computers. In this paper, we combine FERPSO-IMPR (fitness Euclidean distance ratio particle swarm optimizer with information migration-based penalty and population reshaping) and surrogate-assisted models to generate test cases for MPI program path coverage testing. In our proposed method, FERPSO-IMPR employs a dual population strategy to initialize data and calculate fitness. Then, we create a sample set based on the initial data and its fitness. Subsequently, we train the master-slave surrogate models to predict individual fitness. Finally, a small number of elite individuals are selected to execute the program to decide whether to generate the required test data and guide the subsequent evolution process. We apply the proposed method to seven MPI programs and perform experimental comparisons from five directions. Experimental results show that compared with the comparative method, the time consumption of the proposed method is reduced by 33.2%, the number of evaluations is reduced by 38.8%, and the success rate is increased by 7.6%. These results prove that our method can effectively reduce the test data generation cost of MPI programs.**

***Index Terms*—Test data generation, FERPSO-IMPR, surrogate-assisted models, path coverage, MPI program.**

I. Introduction

MPI is used to build parallel computing applications and supports a variety of parallel architectures and operating systems, including shared memory architectures and distributed memory architectures. MPI has become one of the most widely used communication interface standards in the field of parallel computing [1]. Therefore, it is of great significance to study how to test MPI programs.

MPI programs have the characteristics of parallelism, flexibility and non-determinism, which greatly increase the cost and difficulty of testing [2]. To ensure accuracy, we chose the deterministic MPI program because non-determinism can result in the same test data producing different results. In the following sections, unless otherwise specified, the methods discussed are intended for deterministic MPI programs. Previous research proposed a method to select the optimal feasible communication sequence to traverse the same target under the same test data [3]. If combined with this method, the proposed method can also be used for non-deterministic MPI programs.

Previous studies have indicated that software testing expenses make up more than half of the overall software development costs [4]. The complexity of MPI programs is considerably higher than that of regular computer programs, leading to increased testing expenses for MPI programs. In view of this, we focus on how to solve the problem of high MPI program testing costs. Software testing encompasses various coverage standards, such as statement coverage, decision coverage, condition coverage, and path coverage [5]. By employing different coverage criteria during software testing, diverse types of software defects and problems can be identified. Based on this perspective, we mainly study path coverage testing of MPI programs.

There have been many studies on software structure testing. Hausen et al. [6] developed a tool for parallel computing program structure testing that facilitates the generation, evaluation, and replay of test sets. Li et al. [7] introduced an efficient symbolic execution technique for testing MPI applications, addressing the limitations of existing tools. Souza et al. [8] designed a novel MPI-based testing model to address the structural testing problem in concurrent programs. However, these studies focus solely on the characteristics of MPI programs and fail to offer effective methods for generating test data. Xanthakis et al. [9] pioneered the combination of evolutionary algorithms with test data generation, achieving effective testing of path coverage problems and introducing the use of intelligent optimization methods for software test data generation. Tian et al. [10] used a co-evolutionary genetic algorithm to automatically generate test data covering the target path. However, complex MPI programs are computationally expensive due to intra-process computation and inter-process communication. Therefore, it is necessary to propose a computationally efficient method to simplify the generation process of MPI program test data.

Based on the above analysis, we combine FERPSO-IMPR and surrogate-assisted models to generate test data. In the proposed method, we use FERPSO-IMPR to initialize the data and calculate the fitness. Then, we create a sample set based on the initial data and its fitness. Subsequently, we train the master-slave surrogate models to predict individual fitness. Finally, a small number of elite individuals are selected to execute the program to decide whether to generate the required test data and guide the subsequent evolution process. In summary, this paper has the following three contributions:

1) A method for the formation of the model sample set is defined for the problem of generating test data for the MPI program. The quality and diversity of the sample set are critical to the performance of the model. Therefore, the sample set is initialized with as much diversity and exploration as possible. In addition, as the evolutionary process proceeds, the sample set is expanded in terms of both comprehensiveness and data sensitivity.
2) Proposed a method for predicting individual fitness using master-slave surrogate models that are sensitive to excellent data. Previous multi-surrogate models usually use the same type of sample set to train the model and use an averaging method to obtain the predicted values of all models. We divided the models into master and slave categories, with the master model trained using a comprehensive sample set and the slave model trained using a sample set containing only excellent data. The method obtains final predictions by assigning different weights to the master and slave models.
3) Improved on the basis of FERPSO, FERPSO-IMPR is proposed, which is more conducive to test data generation. FERPSO-IMPR adds an information migration-based penalty strategy and a population reshaping strategy on the basis of FERPSO. These two key strategies help the population generate test data as quickly as possible.

This paper is organized as follows: Section II reviews related work. Section III presents the proposed approach. Section IV presents experimental results on seven MPI programs. Section V discusses experimental influencing factors. Finally, Section VI concludes the paper and proposes future research directions.

II. Related Work

A. Preliminary Knowledge

MPI programs typically comprise multiple processes, each operating on individual computing nodes. These processes utilize MPI library functions for effective inter-process communication. An MPI program can be expressed as $M = \{m^0, m^1, \ldots, m^{n-1}\}$, where m^i ($i = 0, 1, \ldots, n\text{-}1$) is the ith process in M, i is the process number of process, and n is the number of processes in M. The input data of program M is usually represented as $X = (x_1, x_2, \ldots, x_k)$, where each variable in X is its jth input data and $j = (0, 1, \ldots, k)$, where k is the number of variables in X. For X, if the range of possible data values for each variable X_j is r_j, then the range of possible data values for X is $R = r_1 \times r_2 \times \ldots \times r_m$.

```
int main(int argc, char *argv[]) { // master
1.  int n = 2; MPI_Comm comm;
2.  MPI_Init(&argc, &argv);
3.  char *s1 = argv[1], *s2 = argv[2];
4.  char *send[2] = {s1, s2};
5.  MPI_Info info;
6.  MPI_Info_create(&info);
7.  MPI_Info_set(info, "host", "localhost");
8.  MPI_Comm_spawn("./worker", …, n, info, &comm);
9.  MPI_Scatter(send, …, comm);
10. int recv[2];
11. MPI_Gather(…, recv, 1, …, comm);
12. int sum = recv [0] + recv [1];
13. printf("The sum is %d\n", sum);
14. MPI_Finalize();
15. return 0;}
```

(a)

```
int main(int argc, char *argv[]) { // worker
1.  MPI_Comm comm;
2.  MPI_Init(&argc, &argv);
3.  MPI_Comm_get_parent(&comm);
4.  char recv[MAXLEN + 1];
5.  MPI_Scatter(..., recv, ..., comm);
6.  recv[MAX_LEN] = '\0';
7.  int len = 0;
8.  int maxlen = 0, curlen = 0; char curstr[MAXLEN + 1];
9.  for (int i = 0; recv[i] != '\0'; i++){
10.     char c = recv[i]; bool repeated = false;
11.     for (int j = 0; curstr[j] != '\0'; j++)
12.         {if (curstr[j] == c) repeated = true;}
13.     if (repeated) { curlen = 0; curstr[0] = '\0';}
14.     curlen++;
15.     strncat(curstr, &c, 1);
16.     if (curlen > maxlen) maxlen = curlen;}
17. MPI_Gather(&maxlen, …, 0, comm);
18. MPI_Finalize();
19. return 0;}
```

(b)

Fig. 1. MPI sample program. (a) Master program code sample. (b) Worker program code sample.

In this section, we illustrate the associated concepts using an MPI example program, as shown in Fig. 1(a) and 1(b). Fig. 1(a) displays the master process code (m^0), which initiates two worker processes (m^1 and m^2) based on the code in Fig. 1(b). The primary function of this program is to compute the sum of the lengths of the longest non-repeating substrings from two given strings.

Node: A node in an MPI program usually refers to a communication primitive or a sequence of executions with the same operation, which is also called a basic execution unit. For a process m^i in M, its lth Node is denoted as n_l^i, which refers to a set of instructions or operations executed in order. As mentioned above, the ninth node in the m^0 can be denoted as n_9^0, and so on for other nodes.

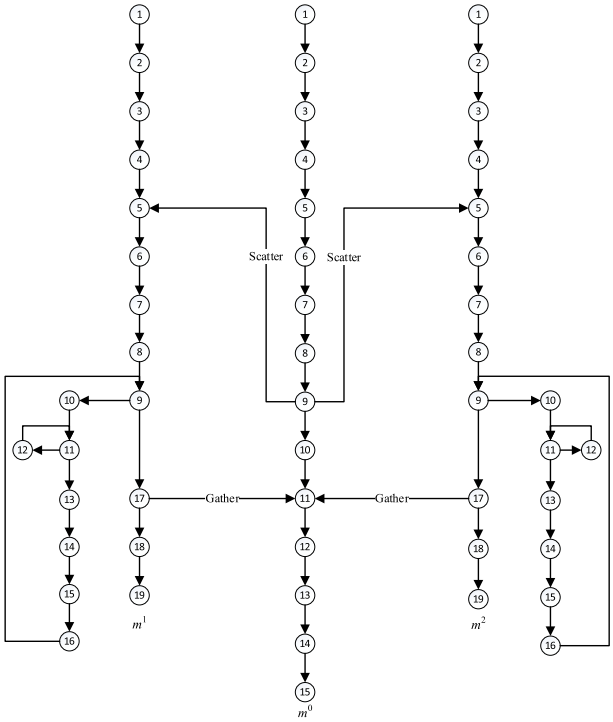

Fig. 2. Control flow diagram of the sample program.

Control-flow graph: A control-flow graph for an MPI program can be expressed as $CFG = \{N_s, ComE_s, ConE_s\}$, where N_s is the set of nodes, $ComE_s$ is the set of communication edges, and $ConE_s$ is the set of control edges. Control edge $ConE$ indicate the execution order among nodes, while communication edge $ComE$ denote communication among different processes. Fig. 2 displays the control flow graph of the example program. Among them, the number represents node N, the edge between two nodes in the same process represents $ConE$, and the edge between m^0, m^1, and m^2 is $ComE$ (for example: the edge from n_9^0 of m^0 to n_5^1 of m^1 is a $ComE$).

Path: Path refers to the execution path from the start to the end node in an MPI program, showing the execution track of the program. It includes MPI function calls, communication and calculation operations. The sub-path p^i of a process m^i is the path that the data $X \in R$ passes through in that process, and the length of the sub-path is denoted by $|p^i|$. The set of all sub-paths that X passes through in all sub-processes is denoted by $P(X) = \{p^0, p^1, \ldots, p^{n-1}\}$. For the m^0, $p^0 = n_1^0 n_2^0 n_3^0 n_4^0 n_5^0 n_6^0 n_7^0 n_8^0 n_9^0 n_{10}^0 n_{11}^0 n_{12}^0 n_{13}^0 n_{14}^0 n_{15}^0$ is one of its sub-paths.

B. Path Similarity

The optimization problem refers to finding the best solution among many schemes or parameter values under certain conditions. Fitness is commonly used to measure the adaptability of a candidate solution to the optimization problem [11]. Generally, the higher the fitness, the better the solution.

In [10], [12], to assess the fitness of the generated test data, researchers adopted the path similarity function as the fitness function. The calculation of path similarity in MPI program testing serves as a means to assess both MPI program test coverage and testing effectiveness. Since the primary objective is to minimize costs associated with test data generation, rather than researching the selection of a specific path similarity calculation method, we adopt the path similarity function as the fitness function. The specific calculation method is as follows.

For a given target path $P^* = \{p^{*0}, p^{*1}, \ldots, p^{*n-1}\}$, when the path that X passes through in the MPI program is $P(X) = \{p^0, p^1, \ldots, p^{n-1}\}$, the similarity between the two paths can be calculated by (1):

$$Similar(P^*, P(X)) = \frac{1}{n}\sum_{i=0}^{n-1} sim(p^{*i}, p^i) \tag{1}$$

where n is the number of sub-paths, p^{*i} and p^i are the sub-paths in P and $P(X)$ respectively. $sim(p^{*i}, p^i)$ is the function for calculating the sub-path similarity, which is expressed as (2):

$$sim(p^{*i}, p^i) = \frac{|p^{*i} \cap p^i|}{\max\{|p^{*i}|, |p^i|\}} \tag{2}$$

where $|p^{*i} \cap p^i|$ denotes the length of the longest common subsequence starting from the first node, $|p^i|$ denotes the sub-path length, and $\max\{|p^{*i}|, |p^i|\}$ denotes choosing the maximum value between $|p^{*i}|$ and $|p^i|$.

Furthermore, we choose existing methods to select and confirm the target path P^* [13]. Initially, we determine the basic target path set BP_s [14], which encompasses all paths of P^*. The number of basic target paths in BP_s is $|BP_s|$. Subsequently, we utilize the breadth-first search algorithm [15] and EPAT [16] to actively select the feasible basic sub-path set BP_s^i for each sub-process. We then sample BP_s^i to construct the target path and verify its reachability [17]. Ultimately, the target paths meeting the specified conditions are consolidated into a basic target path set.

It can be seen from (1) that the closer $Similar(P^*, P(X))$ is to 1, the better it is. Therefore, the mathematical model for generating test data to achieve path coverage can be expressed as follows:

$$\max_{s.t.X \in R} F(X) = Similar(P^*, P(X)) \tag{3}$$

In summary, we ultimately select (3) as the fitness function of the evolutionary algorithms.

C. Test Data Generation

Xanthakis et al. [9] discussed how genetic algorithms can be applied in software testing, along with their advantages and limitations. Jain et al. [18] explored the challenges of automated test data generation and proposed solutions using heuristic methods. Panichella et al. [19] discussed automated test case generation as a many-objective optimization problem with dynamic selection of the targets and proposed a new approach for generating test cases. In addition, Letko et al. [20] introduced a strategy that combines model checking and random testing to assist developers in conducting more comprehensive program testing. Pargas et al. [21] developed a tool called TGen based on genetic algorithm, which is used to generate test data that meets criteria such as branch coverage, path coverage, etc. However, the

testing methods mentioned above are designed for sequential programs and may not suit the testing of parallel programs.

The concept of parallel programs refers to a computational method that can use computers to perform multiple tasks simultaneously. The purpose of parallel programs is to improve computational efficiency and performance, and solve large and complex problems. Issues like task partitioning, allocation, synchronization, and communication must be taken into account when developing parallel programs. For testing parallel programs, Yang et al. [22] introduced an algorithm and tool based on data flow analysis for generating test data that covers all definition-use paths. Viela et al. [23] proposed a genetic algorithm-based approach, called BioConcST, to support automatic test data generation for parallel programs. Shousha et al. [24] extract all relevant concurrency information from UML models of parallel systems conforming to the UML SPT profile, and then use a genetic algorithm to search for execution sequences that may lead to deadlocks. It is evident that these studies provide advantages in enhancing the test data generation capability of parallel programs.

The main research content of this paper is the path coverage testing problem of MPI programs. Therefore, it is necessary to consider not only the characteristics of the MPI program, but also the cost of testing. Sun et al. [25] integrated the estimation results of the surrogate models into the test data generation process. Gong et al. [13] used surrogate-assisted evolutionary optimization to generate MPI program path coverage test data, thereby achieving the goal of reducing testing costs. Obviously, the above work has contributed to reducing the cost of MPI program test data generation. Based on previous research, we combine FERPSO-IMPR and surrogate-assisted models to generate path coverage test data for MPI programs.

D. Evolutionary Algorithms and Surrogate-Assisted Models

Evolutionary algorithms (EAs) are global optimization methods inspired by natural phenomena, and as well have been intensively studied for many years [26]. Common EAs include genetic algorithms [27], ant colony optimization [28], particle swarm optimization [29], and so on [30].

EAs gradually discover optimal solutions by exploring various candidate solutions, without the need for equations or prior knowledge. Sun et al. [12] employed the PSO algorithm to automatically generate test data and selected a subset of individuals to execute the program, thus reducing the number of program runs. Gong et al. [13] considered the multimodal characteristics of parallel programs, and used FERPSO to generate test data, which can effectively generate high-quality test data.

In view of the multi-modal nature of the test data generation problem and the high performance of FERPSO in solving multimodal optimization problems, we use FERPSO to generate test data. However, for the original FERPSO algorithm [31], as the population size expands, FERPSO can find more optimal solutions, but this leads to FERPSO's efficiency problem. Therefore, we improve on the basis of FERPSO. Next, FERPSO is briefly introduced, and its improvements can be viewed in Section III-F.

FERPSO uses personal bests of particles to create a stable memory-swarm containing the best points found by the population so far. Simultaneously, current particle positions make up an explorer-swarm, which explores the search space broadly. Unlike using a single global best, each particle is drawn towards the fittest and closest neighbor based on the FER (fitness and Euclidean-distance ratio) value. When using FERPSO to generate data, the fitness Euclidean distance ratio of particles needs to be calculated. The specific calculation method is shown in (4):

$$FER^{(j,i)} = \mu \cdot \frac{f(P^j) - f(P^i)}{\|P^j - p^i\|} \tag{4}$$

where $\mu = \|D\|/f(P^g) - f(p^w)$ is a proportionality factor, $\|D\|$ represents the size of the searchable range. For particle i, if there exists a particle j, and the fitness Euclidean distance ratio between these two particles is the largest. Then i will take the individual optimal position of particle j as its own global optimal position, and adjust its own position and velocity according to this position. The specific method of updating position and velocity is shown in (5):

$$\begin{cases} V_i^{t+1} = weight \cdot V_i^t + c_1 \cdot RN_1 \cdot (P_i^t - \mathrm{X}_i^t) + \\ \qquad\qquad c_2 \cdot RN_2 \cdot (P_j^t - \mathrm{X}_i^t) \\ X_i^{t+1} = X_i^t + V_i^{t+1} \end{cases} \tag{5}$$

where t is the number of iterations, *weight* is the inertia weight, c_1 and c_2 are learning factors, RN_1 and RN_2 are two random numbers chosen from the range [0, 1].

While EAs is effective in generating test data, it requires frequent execution of MPI programs, incurring substantial computational costs. To mitigate these costs while ensuring comprehensive test coverage, selecting elite individuals for program execution proves advantageous. Surrogate-assisted model was widely used in the software testing field. Chang et al. [32] and. Christiansen et al. [33] both discussed the use of surrogate models in test data generation and analysis. Therefore, we use surrogate-assisted models to help select elite individuals.

Currently, surrogate-assisted models can be divided into two types: single surrogate model (SSM) and multi-surrogate model. SSM includes Gaussian process regression [34], radial basis function network (RBFN) [35], Kriging model [36] and polynomial regression [37]. Multi-surrogate model has two forms, namely ensemble surrogate models [38], [39] and cluster-based surrogate model [40]. Among them, RBFN is a feedforward neural network that uses the radial basis function as the hidden layer activation function of the neural network. Its training process is simple and fast. It is capable of fitting linear and nonlinear functions efficiently. Since we did not study which model is better. Therefore, this paper chooses RBFN as the basic surrogate-assisted model.

III. Integration of FERPSO-IMPR and Surrogate-Assisted Models into Test Data Generation

A. Overall Framework

In order to reduce the cost of MPI program path coverage testing, we combine FERPSO-IMPR with surrogate-assisted models

Algorithm 1: The framework of the proposed approach

Input: $|BP_s|$, *maxNum* (maximum iterations), *elitSize* (number of elite individuals);
Output: FD_s (final test data);

```
Population initialization;
for d = 1 → |BP_s| do
    Initialize the sample sets sampling_1 and sampling_2, details in Section III-B;
    Execute Algorithm 3 to train master-slave models;
    for i = 1 → maxNum do
        Execute Algorithm 4 to predict individual fitness;
        Select the elite individuals, denoted as elitePop;
        for j = 1 → elitSize do
            Calculate the fitness of elite individual elitPop_j through (3);
            if F(elitPop_j) == 1 then
                Save elitPop_j to FD_s, and exit the loop to proceed to the next data generation;
            end
        end
        Execute Algorithm 2 to expand sample sets sampling_1 and sampling_2;
        Determine whether to update the master-slave models, details in Algorithm 5;
        Evolve the population (Details are given in Sections III-E and III-F);
    end
end
return FD_s;
```

Algorithm 2: Extending sample set

Input: $sampling_1$, $sampling_2$, *eliteSize* (number of elite individuals), *elitePop* (elite individuals);
Output: $sampling_1$, $sampling_2$;

```
for i = 1 → elietSize do
    Add elitePop_i and F(elitePop_i) to sampling_1;
    if F(elitePop_i) >= 0.6 then
        Add elitePop_i and F(elitePop_i) to sampling_2;
    end
end
return sampling_1, sampling_2;
```

Algorithm 3: Construction of master-slave models

Input: $sampling_1$, $sampling_2$, *sn* (number of slave models), *p* (population), *psize* (population size);
Output: RBFN_m, RBFN_s;

```
Create sample set S_sn;
Create 1 RBFN_m master process and sn RBFN_s slave processes;
Send sampling_1 to RBFN_m and train;
for i = 1 → sn do
    S^i_sn ← Ø;
    Sample 1/sn data from sampling_2 to S^i_sn;
    Send S^i_sn to RBFN^i_s and train;
end
return RBFN_m, RBFN_s;
```

Algorithm 4: Application of master-slave models

Input: *p* (population), *psize* (population size), *sn* (number of slave models), RBFN_m, RBFN_s;
Output: F^* (estimated value of fitness);

```
for i = 1 → psize do
    F^i_m = RBFN_m(p_i);
    for j = 1 → sn do
        F^ij_s = RBFN^j_s(p_i)
    end
    Calculate the fitness F^*_i through (6) and add to F^*;
end
return F^*;
```

to generate path coverage test data for MPI programs. Algorithm 1 gives the overall framework of the proposed method.

According to Algorithm 1, the first step is to initialize the population. Then, for each basic target path of test data to be generated, the following operations are performed: firstly, initialize the sample set, and use the sample set to train the master-slave surrogate models through Algorithm 3. Next, predict the fitness of individuals within the population using the master-slave surrogate models through Algorithm 4. Subsequently, use the predicted fitness values as input to select elite individuals from the population through Algorithm 7. After that, the elite individual executes the program and reviews the elite individual through (3). If its fitness is 1, save the individual and enter the next target path test data generation process. Otherwise, use Algorithm 2 to expand the sample set and determine whether to update the master-slave surrogate models. Finally evolve the population. Repeat the above process until test data for all target paths in BP_s is generated.

Note that Algorithm 2 is used for expanding the sample set, Algorithm 3 is used to train the master-slave surrogate models, Algorithm 4 is designed for predicting individual fitness, Algorithm 5 is used to determine whether the model needs updating, and Algorithm 7 selects elite individuals. Details of the above algorithms will be given in Sections III-B, III-C, III-D, III-E, and III-F, respectively.

B. Sample Set Formation

In model training, the quality and diversity of the sample set are critical to the performance of the model. Therefore, careful construction and selection of the sample set is a critical step in training accurate and robust models. Additionally, the knowledge generated during the EAs operation proves highly beneficial for the entire testing process. As a result, in this paper, we form this knowledge into a sample set to train the model.

To initialize the sample set, first initialize the dual population (the content of the dual population is in Section III-E). Then, the

individual executes the program and calculates the fitness through (3). Finally, add the initial population data and its fitness to the sample sets $sampling_1$ and $sampling_2$ together. In summary, the diversity and breadth of the dual population ensure the quality and diversity of the initialization sample set.

In addition, the sample set size has a significant impact on model performance and generalization ability. The number of samples in the initial sample set is too small. At this point, the model cannot generalize well to new data, so the sample set needs to be expanded during the evolution process. Algorithm 2 shows the pseudocode for expanding the sample set. During each iteration, elite individuals are selected to execute the program. If the required test cases are not obtained, all elite individuals and their fitness are added to the $sampling_1$, and those individuals with fitness greater than or equal to 0.6 among the elite individuals and their fitness are added to the $sampling_2$. The reason is that $sampling_1$ is used to train the master model, so the diversity of training samples needs to be ensured. $sampling_2$ is used to train the slave model. Training with samples with high fitness can improve the sensitivity of the model to high fitness data, thereby assisting the master model in making more accurate estimates of high fitness data.

C. Master-Slave Surrogate Models

When using EAs to automatically generate test cases, to evaluate the quality of the generated data, we need to execute the corresponding MPI program to calculate the fitness. Executing MPI programs is the process that consumes the most computing resources. In view of this, we have chosen to use the master-slave surrogate models to predict individual fitness and reduce computational consumption.

The master-slave surrogate models take the form of a master model and multiple slave models, as opposed to the previously proposed multi-surrogate models, as follows: we train the master model using $sampling_1$, which contains all the samples, enabling it to comprehensively learn the patterns and regularities of the data. We train the slave model using $sampling_2$, which includes only the excellent data, allowing it to focus more on the features of the excellent data. By this method, the features of different sample sets can be fully utilized to improve the sensitivity of the model to the excellent data and obtain more accurate predictive values for the excellent individuals. Ultimately, the predicted value of the data is the weighted sum of the predicted values of the master and slave models. The final number of slave models for the experiments is 2, because the number of sample sets in the early stages of evolution is small, and too many slave models will result in too small a sample set for the slave model, rendering the slave model useless.

Algorithm 3 provides the pseudocode for constructing the master-slave surrogate models. First, create the sample set S_{sn} and initiate the master-slave model process. Then, send $sampling_1$ to the master model for training. After that, select $1/sn$ data from $sampling_2$, add it to S^i_{sn} ($i = 1, 2, \ldots, sn$) and send it to the corresponding slave model for training. Finally, the trained master-slave surrogate models are obtained.

Algorithm 5: Updating master-slave models

Input: *maxUp* (maximum update frequency), *minUp* (minimum update frequency), *maxNum* (maximum iterations);
Output: $RBFN_m$, $RBFN_s$;

```
for i = 1 → maxNum do
    slope ← (maxUp - minUp)/maxNum;
    upFre ← maxUp - (slope * (maxNum - i));
    if upFre < minUp then
        upFre = minUp;
    end
    if (maxNum - i) % upFre == 0 then
        Execute Algorithm 3 to update the RBFN_m and RBFN_s;
    end
end
return RBFN_m, RBFN_s;
```

When predicting individual fitness, the population is passed into the master-slave surrogate models respectively, and then the final prediction value is obtained by weighting and summing the prediction results of the master-slave surrogate models. The purpose of weighting is to determine the degree of contribution based on the performance and credibility of the model. Algorithm 4 gives the method for predicting individual fitness. F^i_m is the prediction result of the master model for individual i. F^{ij}_s represents the prediction result of individual i from slave model j, and F^i_s is the result set. The final fitness prediction of the individual can be calculated through F^i_m and F^i_s. The calculation formula of fitness weighted summation is shown in (6):

$$F^*_i(F^i_m, F^i_s) = \alpha \cdot F^i_m + \frac{1-\alpha}{sn-1}\sum_{j=1}^{sn} F^{ij}_s \tag{6}$$

where α is the trust control weight. Using α can give different importance to different surrogate models in order to control their contribution. The calculation formula of α is as shown in (7):

$$\alpha = \frac{1}{1+e^{-t}} \tag{7}$$

where t represents the number of iterations. With continuous iteration, α gradually increases from 0.5 to close to 1. The reason for this allocation is as follows: with continuous iteration, the sample set accumulates more and more excellent data. Therefore, the predictive power of the master model strengthens over time, leading to an increase in the corresponding weight value.

As can be seen from Section III-B, when the sample set is too small, the generalization ability of the trained model is low. Therefore, updating the model during runtime becomes crucial. Considering that the predictive ability of the model usually increases with the increase of the sample set, update operations will be more frequent in the beginning and will gradually decrease as the number of iterations increases. Algorithm 5 presents the pseudocode for updating the model. Among them, *maxUp* and *minUp* are used to control the update frequency, which means that within the total number of

Algorithm 6: Select the elite individuals

Input: p (population), F^* (estimated value of fitness), *psize* (population size), *ns* (the number to be selected);
Output: *elitePop* (elite individual set);

```
Create layer sets layer_1, layer_2, layer_3, and layer_4;
for i = 1 → psize do
    Put p_i into the corresponding layer sets according to F_i^*;
end
for i = 1 → ns do
    for j = 1 → 4 do
        if layer_j not empty then
            Randomly select an individual to add to elitePop and removed;
            break;
        end
    end
end
return elitePop;
```

Algorithm 7: Dual population strategy

Input: *psize* (population size), *maxNum* (maximum iterations);
Output: pop_1, pop_2 (final population);

```
Randomly initialize population pop_1;
Initialize population pop_2 using Latin hypercube sampling;
while Number of iterations < maxNum || No test case do
    Execute Algorithm 4 to predict individual fitness;
    prop_1, prop_2 ← Calculate the proportion of outstanding individuals;
    Form elite sets by selecting a proportion of outstanding individuals from their respective populations based on prop_1and prop_2;
    Combine the pop_1 and pop_2 to update the individual and global historical optimal position and optimal fitness;
    Update the pop_1 and pop_2;
end
return pop_1, pop_2;
```

iterations, the update frequency will gradually decrease from *maxUp* to *minUp*.

D. Select Elite Individuals

In EAs, individuals with high fitness usually contain more useful information. Therefore, compared with all individuals executing the program, selecting elite individuals to execute the program can reduce computational costs while ensuring test coverage.

Algorithm 6 gives the pseudocode for selecting elite individuals. First, we divide the population into four layers based on fitness. Those with fitness in the range of [0.9, 1] are in the first layer, those in the range of [0.8, 0.9] are in the second layer, those in the range of [0.7, 0.8] are in the third layer, and those in the range of [*Olim*, 0.7] are in the fourth layer. *Olim* represents the individual distinction factor (refer to Section III-F for details). Then, selection starts from the first layer using a no-put-back sampling method. When there are not enough individuals in the first layer, selection is made to the next layer until enough individuals are selected.

E. Dual Population Application Strategy

In EAs, the diversity and exploration of the population are very important. However, balancing diversity and exploration can be challenging. In view of this, we applied a dual population strategy. One population focuses on broad searches, the other on deep exploration.

As depicted in Algorithm 7, we randomly initialize pop_1 to create diverse initial individuals, and we use Latin hypercube sampling to initialize pop_2, thereby enhancing the population's comprehensiveness. In each iteration, the individual fitness is first obtained, and the proportion of outstanding individuals in population is calculated based on the *loser* (minimum limit). Then, select individuals from the corresponding population to form an elite individual set based on the proportion of outstanding individuals (for example: select $prop_1/(prop_1+prop_2)$*$eliteSize$ outstanding individuals from pop_1). Elite individuals are used to determine whether to generate test cases. After that, we merge pop_1 and pop_2 to update individual and global historical best positions and best fitness. Finally, we employ FERPSO-IMPR to update the population.

F. FERPSO-IMPR

The MPI program path coverage test data generation is a typical multi-modal optimization problem, because for a certain path, there may be multiple similar and qualified test data in the data domain. FERPSO has been widely used in solving multi-modal optimization problems. Although FERPSO has excellent performance, it cannot generate test data as quickly as possible in the pursuit of low-cost test data generation problems and cannot jump out as quickly as possible when it falls into a local optimum.

Based on the above analysis, we propose an improved algorithm for FERPSO, known as FERPSO-IMPR. FERPSO-IMPR adds two key strategies: information migration-based penalty and population reshaping. Key information on FERPSO can be found in Section II-D.

1) Information Migration-Based Penalty: We propose the information migration-based penalty to generate test data as quickly as possible. To distinguish which individuals in population require punishment, we introduce the individual distinction factor *Olim*. If an individual's fitness is lower than *Olim*, we consider it as a punished individual. In the information migration-based penalty, we introduced a concept called Neighbor. For an individual *indi* whose fitness is lower than *Olim*, first divide the population into left and right parts with it as the midpoint. Then, select all outstanding individuals (individuals with fitness greater than *Olim*) in the left and right parts respectively,

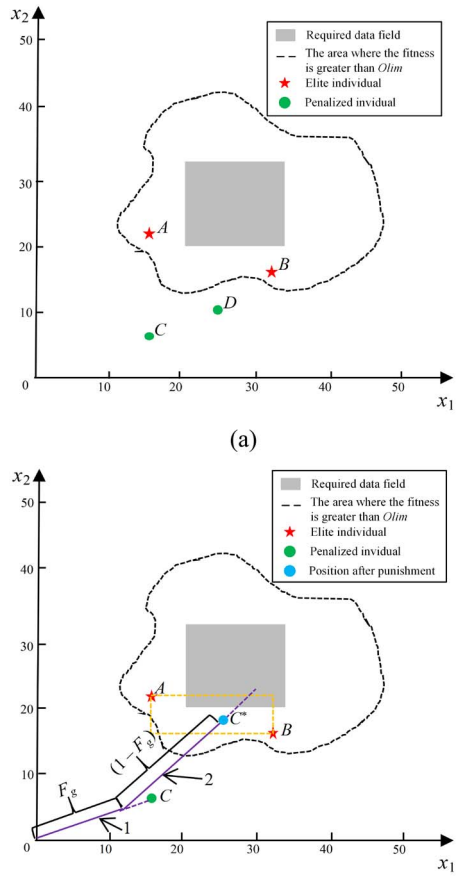

Fig. 3. A sample of punishment based on information transfer. (a) Individual neighbor. (b) Punishment based on information transfer.

recorded as $L = \{lt_1, lt_2, \ldots, lt_{ml}\}$ and $R = \{rt_1, rt_2, \ldots, rt_{mr}\}$, where *ml* and *mr* are the number of outstanding individuals in the left and right parts respectively. In L and R, the individual closest to *indi* is called the neighboring individual of *indi*. Obviously, *indi* has two neighboring individuals, recorded as GN_L and GN_R. Then, $GN = \{GN_L, GN_R\}$ is called the Neighbor of *indi*. It is worth noting that if *ml* or *mr* is 0, GN_L or GN_R should select the individual with the highest fitness in the corresponding part. For example, as shown in Fig. 3(a), the dots represent punished individuals, the star points represent outstanding individuals, and the area within the dotted line is the area where the fitness is greater than *Olim*. It can be seen from the above that the Neighbor of the punished individual C and D are both $GN = \{A, B\}$.

Based on Neighbor, the information migration-based penalty imposed on the punished individual *indi* can be expressed as:

$$indi[i] = F_g \cdot indi[i] + (1 - F_g) \cdot \beta \cdot (GN_L[i] + GN_R[i]) \quad (8)$$

where F_g is the proportion of outstanding individuals in the current population to all individuals in population, and β is the learning rate control parameter, which determines the step length of each movement of the punished individual. β is set to 0.5 in this paper. It is worth noting that the individual penalty is determined by F_g. When F_g is 1, all individuals in population are outstanding individuals and no penalty is required; when F_g is 0, there is a maximum penalty to help individuals escape from the worst situation at this time.

As can be seen from Fig. 3(b), for the penalized individual C and its $GN = \{A, B\}$. Phase 1 is a self-penalizing process, controlled by $F_g \cdot indi[i]$, denoting the deflation of its own data. Phase 2 is the penalty process of Neighbor, controlled by $(1 - F_g) \cdot \beta \cdot (GN_L[i] + GN_R[i])$, indicating the direction and distance that C should migrate.

2) Population Reshaping: In EAs, population evolution sometimes falls into local optimum. The reason for this problem may be that the population initialization is unreasonable, which affects the search process. For test data generation, falling into a local optimum basically means that the data generation failed. Therefore, we propose a population reshaping strategy to jump out of the local optimum as quickly as possible.

The basic idea of the population reshaping strategy is to specifically reorganize or reset the population to a certain extent during the iterative process of the algorithm to increase the diversity and exploration capabilities of the population. Algorithm 8 gives the pseudocode of the population reshaping strategy. The population reshaping strategy is mainly divided into two parts: individual reshaping and population reshaping. The criterion for individual reshaping or population reshaping is the proportion of failed individuals in population. When the number of failed individuals *loserPop* >= *psize* * *lprop*, the population is reinitialized. Otherwise, only the *NPsQ* individuals with the smallest fitness in population are regenerated.

Algorithm 8: Population reshaping

Input: p (population), *psize* (population size), *loser* (minimum limit), *lprop* (minimum ratio), *NPsQ* (number of individual reshaping);
Output: p;

```
loserPop ← 0;
For each individual p_i in p, if the fitness of p_i is less than loser, increment loserPop;
if loserPop >= psize * lprop then
    Initialize p;
else then
    Initialize the NPsQ individuals with the minimum fitness in p;
end
return p;
```

G. Performance Analysis

The performance analysis can be divided into five main parts: sample set formation and expansion, model training and application, dual population strategy, elite individual selection, and FERPSO-IMPR.

It can be known from Section III-B that the performance consumption of initializing the sample set mainly comes from the execution of MPI programs, and its time complexity is recorded

as T(m), where T(m) represents the execution time complexity of different MPI programs. The pseudocode of the expanded sample set is shown in Algorithm 2, and its time complexity is O(*eliteSize*), where *eliteSize* is the number of elite individuals.

Compared with previous multi-surrogate models, the master-slave surrogate models are different in application methods, but does not introduce additional performance costs. Among them, the time complexity of the training model is O($psize*k*m$), and the time complexity of the application model is O($k*psize^2$), where *psize* represents the population size, k represents the number of RBFN center points, and m represents the number of parameters. Since we choose 2 slave models to run in parallel for prediction, the additional performance overhead is very small compared to that of a single model.

The correlation between the dual population strategies adopted in this paper is weak and does not have much communication and computational overhead. Compared with the single-population strategy, the adopted strategy only increases the time complexity of O(*psize*) when selecting elite individuals. Ultimately, the time complexity of both the dual population strategy and the single-population strategy is of the order O(*psize*), and the computational overhead is negligible.

In previous work, the operation of selecting elite individuals is usually to sort the fitness, and its time complexity is O($psize*$log $psize+ns$) in the optimal case, where *psize* represents the population size and *ns* represents the number of elite individuals. The elite individual selection method proposed in this paper is shown in Algorithm 6. Its time complexity is O($psize+ns$) and has better performance.

FERPSO-IMPR is an improved algorithm of FERPSO. It introduces information migration-based penalty and population reshaping, which increases the computational overhead compared to FERPSO. However, these strategies are added to better generate the required test data, and the final performance needs to be evaluated experimentally.

In summary, this paper can efficiently generate MPI program path coverage test data with the lowest possible time complexity and as little performance costs as possible.

IV. Experiment

In this section, we apply the proposed method to the testing process of several complex MPI programs to determine its effectiveness in reducing the cost of MPI program path coverage testing. We begin by presenting the problem and the objectives we aim to achieve. Next, we describe the test environment and the experimental design. Finally, we present the experimental results and engage in examination and discussion.

A. Research Problem

We combine FERPSO-IMPR and surrogate-assisted models to generate test data for MPI program path coverage testing. In the proposed method, we apply a dual population strategy to enhance diversity and exploration and enhance the formation of the sample set. Therefore, we need to verify whether dual population offer more advantages than single population. To expedite the process of generating test data, we introduce an improved FERPSO-IMPR algorithm. Therefore, we need to verify whether FERPSO-IMPR can enhance the exploration capability and convergence speed of the algorithm. Furthermore, we define the sample set formation method and the master-slave surrogate models. Therefore, it is important to verify whether selecting elite individuals can reduce testing costs and whether the master-slave surrogate models can predict individual fitness.

If the previously mentioned method effectively reduced testing costs and improved testing efficiency, it would validate the effectiveness of the proposed approach. Building on this, we have formulated the following research questions and hypotheses.

RQ1: Are dual populations more advantageous?

For RQ1, we compared the dual population strategy with the single population strategy (singlePop) to confirm whether it could enhance the algorithm's exploration ability, accelerate convergence speed, and expedite the discovery of target data.

In the process of generating test data using the proposed method and singlePop, we employed the control variable method to maintain consistency in all strategies except for the population usage method. If the proposed method exhibits higher efficiency and success rates, it demonstrates the advantage of the dual population strategy.

RQ2: Can FERPSO-IMPR generate better individuals earlier?

For RQ2, we proposed FERPSO-IMPR based on FERPSO. But it is unclear whether FERPSO-IMPR successfully compensates for FERPSO's shortcomings. Therefore, we compare the proposed method with FERPSO, which we call ferPSO.

To ensure the experimental accuracy, we use the method of controlling variables to ensure that other strategies are consistent except for the EAs. If the proposed method proves to be more beneficial for producing test data, it can indirectly demonstrate that FERPSO-IMPR generates better individuals earlier.

RQ3: Can the master-slave surrogate models proposed in this paper effectively help to select elite individuals?

For RQ3, considering the wide application of ensemble surrogate model (ESM), we choose to compare with the model using ESM idea. We refer to this comparison method as comModel.

Except for the surrogate-assisted models, other strategies for using comModel are consistent with the original paper. If the master-slave surrogate models have better effects in generating test data, it can indirectly prove that the master-slave surrogate models can effectively help select elite individuals and reduce the testing cost.

RQ4: Can selecting elite individuals to execute programs effectively reduce cost in evaluating individuals?

For RQ4, we select elite individuals to execute the program in order to obtain the required test cases faster. But the efficiency of methods without elite selection is unknown. Therefore, we chose to compare with methods using only EAs to determine whether the elite individual selection is effective. We refer to methods that only use evolutionary algorithms as nonElite.

The experiment still uses the method of controlling variables. By demonstrating higher efficiency, the proposed method can prove that executing the program for a few individuals with good estimation can effectively lower the cost of MPI program path coverage testing.

TABLE I
BASIC INFORMATION OF THE PROGRAMS UNDER TEST

Program	Processes	Communication Primitives	Lines of Code	$\|BP_s\|$
Mpi_Dijkstra	4	27	560	20
Convex	4	49	569	15
QR_value	3	34	575	11
Cjacobi	4	97	721	17
Heat	3	39	613	19
DepSolver	4	33	8988	227
ClustalW	4	352	23256	468

RQ5: Can the proposed method improve the efficiency of generating test cases?

For RQ5, in order to verify whether the combination of all strategies is still effective, we compare it with the existing method [13]. The existing method is denoted as SAEO.

The control variable method is not used in the comparison, and test cases are generated according to their own optimal ways. If superior efficiency is exhibited by the proposed method in evaluation, it serves as evidence that the approach can enhance the efficiency of test case generation.

B. Program Under Test

In this section, we select seven open source MPI programs as test objects, and Table I provides details on these seven MPI programs. Among them, Mpi_Dijkstra represents a parallel implementation of the Dijkstra shortest path algorithm. Convex, QR_value, and Cjacobi are three MPI programs. Convex can be utilized to compute the convex hull of a set of points on a plane, which defines the smallest convex polygon encompassing these points. QR_value is a program designed for calculating the eigenvalues of a matrix, while Cjacobi is a numerical method employed for diagonal zing symmetric matrices. Additionally, Heat serves as a parallel solver for the heat equation [41]. As for DepSolver and ClustalW, both of these programs originate from [42]. DepSolver is a 3D static analysis program capable of handling multiple media simultaneously, while ClustalW is a widely used tool for conducting multiple gene sequence alignments.

The objective of this paper is to minimize the expenses related to software test generation. Therefore, it is crucial to perform testing from multiple and representative perspectives. As can be seen from Table I, the test programs mentioned have differences in various aspects, so the selected test programs have good representativeness.

C. Experimental Setting

The hardware configuration used in the experiment is Intel® Core™ i7-8750H CPU, 16.0GB RAM, 512G SSD hard disk. The software configuration is Windows 10 operating system and Microsoft Visual Studio 2013 as well as MPI+C/C++ programming languages.

To ensure the stability and efficiency of the FERPSO-IMPR algorithm, the parameters of the FERPSO part are configured based on the values specified in the original paper [43]. Among them, the value of w decreases linearly from 0.9 to 0.4, the population size is 30, the two learning factors c_1 and c_2 are both set to 1.49, *Vmax* is set according to the domain of the program input, and *MaxNum* is equal to 1,000.

We evaluate experimental results using three metrics: time consumption, number of evaluations, and success rate. Time consumption measures the total time taken to generate test cases for all target paths. The number of evaluations counts how often the test program runs post-test case generation. The success rate represents the percentage of runs that successfully produce required test cases. For instance, an 80% success rate means 80 out of 100 runs are successful. To mitigate potential errors and biases, we independently run each program 30 times and then calculate the average time consumption and number of evaluations.

D. Key Parameter Settings

Before conducting the test, some parameters need to be confirmed to ensure efficient test case generation. These parameters include *Olim*, *loser*, *lprop*, and *NPsQ*. To control variables and ensure accuracy, an initial value will be given to the remaining parameters. The initial values are $Olim = 0.8$, $loser = 0.2$, $lprop = 1$, $NPsQ = 1$, and $eliteSize = 10$.

We select time consumption, number of evaluations, and success rate as evaluation criteria. Among them, lower time consumption and fewer evaluations are better, and a higher success rate is better. To comprehensively assess these three indicators, we employ the weighted average method. Proceed as follows:

1) Assign the weight of each parameter according to its importance. Since this paper focuses on reducing the testing cost, time consumption is the most important, followed by success rate, and then number of evaluations. The weights are $w_1 = 0.5$, $w_2 = 0.2$, and $w_3 = 0.3$.
2) The time consumption and the number of evaluations have been transformed into numbers between 0 and 1 using the extreme difference normalization method.
3) The calculation method of the comprehensive evaluation value is to add the normalized values of each parameter multiplied by their weights, that is:

$$score = w_1 \cdot (1 - TC) + w_2 \cdot (1 - FE) + w_3 \cdot SR \quad (9)$$

where *TC* is the normalized value of time consumption, *FE* is the normalized value of number of evaluations, *SR* is the normalized value of success rate. Finally, compare according to the comprehensive evaluation value, the higher the evaluation value, the better the data.

In this paper, we use Convex as an example to illustrate the steps for confirming the optimal parameter values. For Convex, we select 15 paths to create the basic target path set. Different parameters have distinct value ranges based on their meanings. For example, the value range of *Olim* is [0.1,0.9], the value range of *loser* is [0,0.9], the value range of *lprop* is [0.1,0.9], and the value range of *NPsQ* is [1,15].

For the given basic target path, follow the steps in Section III-A for test case generation. Repeat this process until test data is generated for all the target paths within the basic target path set. For each parameter value, this operation is performed 20 times, and then the time consumption, evaluation numbers, and success rate are calculated and transformed into a composite evaluation value.

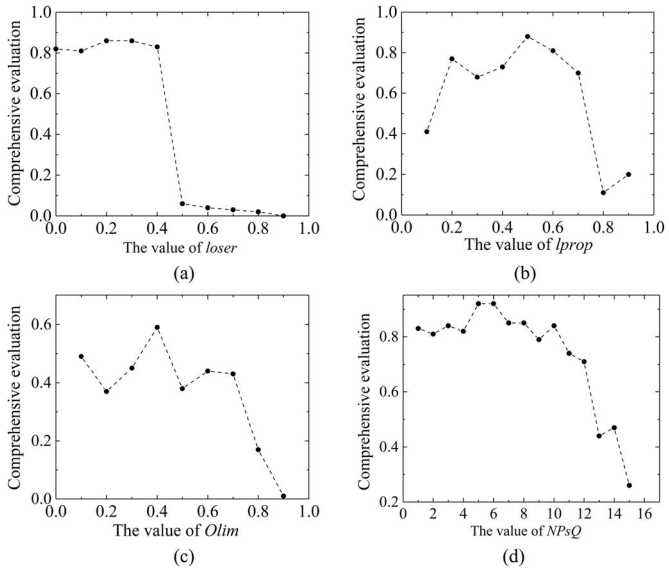

Fig. 4. Comprehensive evaluation of each parameter. (a) Results for different values of *loser*. (b) Results for different values of *lprop*. (c) Results for different values of *Olim*. (d) Results for different values of *NPsQ*.

Fig. 4(a) displays the experimental results for the *loser* in the range of values with a step size of 0.1. It is clear that the comprehensively evaluate value fluctuates gently in the interval [0, 0.4] and has the highest value at 0.2. From 0.5 onward, the evaluation value plummets. Therefore, it is reasonable to set the optimal value of *loser* to 0.2.

Fig. 4(b) displays the experimental results of *lprop* with a step size of 0.1. There is an upward trend between 0.1 and 0.5, followed by a decline from 0.5 to 0.8, and a slight rebound at 0.9. However, the highest value is observed at 0.5. Hence, setting the optimal value for *lprop* as 0.5 is reasonable.

Fig. 4(c) displays the experimental results for *Olim*, with a chosen step size of 0.1. The trend of the data is similar to that of *lprop*, with a trend of lifting and then decreasing. Obviously, the data reach their highest value at 0.4 Therefore, it is reasonable to set the optimal value of *Olim* to 0.4.

As depicted in Fig. 4(d), the experimental results of *NPsQ* increase slowly between 1 and 6, and then start to decrease. And the values 5 and 6 have the same comprehensively evaluate value. Since, *NPsQ* is the number of reshaping individuals, *NPsQ* should be taken as small as possible. Therefore, it is reasonable to set the optimal value of *NPsQ* to 5.

For other tested programs, except for Convex, we are following the above process to determine the optimal values of parameters. The values of parameters in each tested function are being shown in Table II. Considering that setting optimal parameter values is also costly, the average value in the last column of Table II can be taken as the parameter value to improve versatility and reduce test cost.

E. Experimental Results and Analysis

The experimental results for RQ1-RQ5 are given and analyzed in this section.

1) RQ1: Are Dual Populations More Advantageous?: To solve RQ1, we obtain the average time consumption for covering the entire target path set of the test program 30 times by comparing the dual population strategy with singlePop. Furthermore, to facilitate a more effective comparison of the efficiency difference between the two methods, we compute the average number of evaluations for 30 successful tests. Table III presents the experimental data for the two methods. It displays the average time consumption of the proposed method in row 3 and the average time consumption of singlePop in row 4. Row 6 provides the average number of evaluations for the proposed method, while row 7 presents the average number of evaluations for singlePop. Rows 5 and 8 indicate the lift rate of the proposed method for the corresponding metrics, respectively. The lift rate is calculated as follows:

$$Lrate = \frac{(num - num^*)}{num} \cdot 100\% \tag{10}$$

where *num* denotes the average value of the comparison method and num^* denotes the average value of the proposed method. In the following, the lift rates are calculated from (10).

First, the comparison of average time consumption between the proposed method and singlePop is shown in rows 3-5 of Table III. The data reveals the following: 1) The average time consumption of the proposed method is lower than the singlePop in all programs. The average time consumption of QR_value is the smallest, 17.8s and 19.5s, respectively, and the average time consumption of ClustalW is the largest, 936532.3s and 1113850.5s, respectively. 2) The proposed method has a positive lift rate over the singlePop. Among them, DepSolver has the largest lift rate of 29.7%, and QR_value has the smallest lift rate of 8.7%. For all programs, the average lift rate of the proposed method is 19.8%. The proposed method demonstrated its capability in reducing the time required for test case generation.

Second, the difference in the average number of evaluations between the two methods is shown in rows 6-8 of Table III. The data reveals the following: 1) The average number of evaluations for the proposed method are lower than those for singlePop. QR_value has the fewest evaluations among the two methods, with 182.7 and 213.4, respectively, while ClustalW has the highest number of evaluations, with 15715 and 18445, respectively. Across all programs, the average number of evaluations for the proposed method and singlePop are 4590.1 and 6214.4, respectively. 2) The lift rates vary among different programs, but all are positive. Among them, DepSolver shows the most significant lift rate of 46.3%, while QR_value exhibits the smallest lift rate of 14.3%. For both methods, the average lift rate across all programs is 23.6%. It is evident that the proposed method holds a clear advantage.

Finally, it can be seen from rows 9-11 of Table III: 1) The success rate of the proposed method exceeds or equals that of singlePop, with success rates of 100% for Mpi_Dijkstra, QR_value, and Heat using the proposed method. On average, the success rates for all programs with the proposed method and singlePop are 95.2% and 88.3%, respectively. 2) Cjacobi exhibits the largest difference in success rate at 15.2%, while the difference for Mpi_Dijkstra and Heat is 0%. Across both methods, the average

TABLE II
PARAMETER SELECTION VALUES

Program	Mpi_Dijkstra	Convex	QR_value	Cjacobi	Heat	DepSolver	ClustalW	Average
loser	0.1	0.2	0.5	0.7	0.8	0.6	0.6	0.5
lprop	0.2	0.5	0.6	0.8	0.7	0.8	0.7	0.61
Olim	0.4	0.4	0.3	0.5	0.6	0.7	0.6	0.5
NPsQ	2	5	5	7	6	3	4	4.57

TABLE III
EXPERIMENTAL COMPARISON OF USE STRATEGIES AMONG DIFFERENT POPULATIONS

1	Program	Mpi_Dijkstra	Convex	QR_value	Cjacobi	Heat	DepSolver	ClustalW	Average
2	$\|BP_s\|$	20	15	11	17	19	227	468	111
3	Proposed/s	33.6	28.1	17.8	38.7	1000.9	30315.4	936532.3	138280.9
4	singlePop/s	38.1	33.6	19.5	52.8	1422.2	43167.7	1113850.5	165512.1
5	Lifting rate/%	11.8	16.3	8.7	26.7	29.6	29.7	15.9	19.8
6	Proposed	251.9	6014.5	182.7	2940.8	390	6636	15715	4590.1
7	singlePop	311.1	7875	213.4	3752.1	528.8	12376	18445	6214.4
8	Lifting rate/%	19	23.6	14.3	21.6	26.2	46.3	14.8	23.6
9	Proposed/%	100	90	100	96.7	100	95	85	95.2
10	singlePop/%	100	75	96.7	81.5	100	90	75	88.3
11	Difference/%	0	15	3.3	15.2	0	5	10	6.9
12	Proposed_variance	0.5890	7.5728	0.2434	4.0593	57.1973	5724.1145	146401.4635	21742.1771
13	singlePop_variance	5.4101	18.9112	1.3690	7.71936	110.8796	6020.9921	196106.6698	28895.9930

difference among all programs is 6.9%. This indicates that the proposed method is highly efficient in generating test data.

To assess whether the evaluation indicators might be significantly influenced by excessive randomness in the experimental results of the two methods, we measure the accuracy and stability of these methods using the standard deviation. Data stability is indicated by a smaller standard deviation, whereas a larger standard deviation implies greater data dispersion. The calculation of the standard deviation is presented in rows 12-13 of Table III.

Rows 12-13 of Table III reveal that the standard deviation values by the proposed method across all programs are smaller than those for the singlePop, with notable differences in values. Therefore, combined with the standard deviations, it can be seen that the dual population has higher stability than the singlePop in generating path coverage test data, and basically there is no sudden high and low test efficiency.

Based on the analysis above, we can conclude that the utilization of the dual population strategy offers significant advantages in generating parallel program path coverage test data.

2) RQ2: Can FERPSO-IMPR Generate Better Individuals Earlier?: To solve RQ 2, we employ FERPSO-IMPR and ferPSO to generate test cases for seven programs, respectively. We then calculate the corresponding metrics based on the results. Table IV presents the comparison results, where each row conveys the same meaning as in Table III. More information can be found above.

Rows 3-5 of Table IV present the experimental results of FERPSO-IMPR and FERPSO in all programs. 1) The proposed method outperforms ferPSO in terms of average time consumption for all programs, with QR_value being the least time-consuming program at 17.5s and 19.1s, respectively. The average time consumption for all programs under both methods is 130979.9s and 163609.0s, respectively. 2) The proposed method also exhibits a higher lift rate in average time consumption than ferPSO, with DepSolver showing the largest enhancement of 45.2%. The other programs demonstrate lift rates ranging from 8.4% to 23.8%. When considering all programs, the proposed method achieves an average lift rate of 20.4%.

The comparative data for the two methods in terms of the average number of evaluations are presented in rows 6-8 of Table IV. 1) QR_value exhibits the smallest difference in average number of evaluations between the two methods, with values of 174.3 and 212.1, respectively. In contrast, ClustalW shows the largest difference in average number of evaluations, with values of 15250.7 and 20067.8, respectively. For all programs, the average number of evaluations for the two methods are 4768.0 and 6729.6, respectively. 2) Regarding the difference in the lift rate, QR_value demonstrates the smallest difference of 17.8%, while Heat exhibits the largest difference of 51.1%. The average lift rate is 30.1%.

Finally, when observed from the perspective of success rates, as shown in rows 9-11 of Table IV: 1) The proposed method achieves a 100% success rate in four programs, while the lowest success rate is 82% for Convex. In contrast, the ferPSO method fails to attain a 100% success rate in any of the programs, with the lowest being 75% for Heat. The average success rates for the two methods are 94.5% and 85.8%, respectively. 2) The difference in success rates varies among the tested programs. The success rates of DepSolver and ClustalW are the same for both methods, while Heat exhibits the largest difference of 25% between the two methods. The average success rate improvement of the proposed method is 8.7%.

The proposed method offers greater stability than ferPSO in the majority of the tested programs, as indicated in Table IV. In summary, using FERPSO-IMPR can improve exploration capabilities and convergence speed, and generate test data faster.

TABLE IV
EXPERIMENTAL COMPARISON BETWEEN THE FERPSO-IMPR AND THE ORDINARY ALGORITHM

1	Program	Mpi_Dijkstra	Convex	QR_value	Cjacobi	Heat	DepSolver	ClustalW	Average
2	$\|BP_s\|$	20	15	11	17	19	227	468	111
3	Proposed/s	34.0	28.7	17.5	59.9	955.9	40909.2	874854.7	130979.9
4	ferPSO/s	39.0	35.3	19.1	71.5	1255.8	74683.0	1069159.8	163609.0
5	Lifting rate/%	12.8	18.6	8.4	16.2	23.8	45.2	18.1	20.4
6	Proposed	227.9	5573	174.3	4265.5	525	7360	15250.7	4768.0
7	ferPSO	315.4	8375.5	212.1	5331.6	1075	11730	20067.8	6729.6
8	Lifting rate/%	27.7	33.4	17.8	19.9	51.1	37.2	24.0	30.1
9	Proposed/%	100	82	100	100	100	95	85	94.5
10	ferPSO/%	96	78.9	93.7	77.5	75	95	85	85.8
11	Difference/%	4	3.1	6.3	22.5	25	0	0	8.7
12	Proposed_variance	0.5513	7.2458	0.2669	12.2490	33.7458	8279.9070	117141.7057	17925.0959
13	FERPSO_variance	7.0219	17.9764	0.8248	12.4279	155.1477	24634.0406	160387.1332	26459.2246

TABLE V
EXPERIMENTAL COMPARISON BETWEEN DIFFERENT SURROGATE-ASSISTED MODELS

1	Program	Mpi_Dijkstra	Convex	QR_value	Cjacobi	Heat	DepSolver	ClustalW	Average
2	$\|BP_s\|$	20	15	11	17	19	227	468	111
3	Proposed/s	32.3	34.9	17.6	46.6	910.9	33532.9	851546.9	126588.8
4	comModel/s	37.0	65.8	19.0	76.7	1788.0	55318.0	1122640.5	168563.5
5	Lifting rate/%	12.7	46.9	7.3	39.2	49.0	39.2	24.1	31.2
6	Proposed	203	3352.7	222.2	3817.4	862.5	4534.2	14547.3	3934.2
7	comModel	248.3	6323.9	274.4	5605.9	1847.5	10448.4	18023.9	6110.3
8	Lifting rate/%	18.2	46.9	19.0	31.9	53.3	56.6	19.2	35.0
9	Proposed/%	96.8	100	100	82	75	95	78.9	89.6
10	comModel/%	88.5	100	68	82	66.7	90	75	81.4
11	Difference/%	8.3	0	32	0	8.3	5	3.9	8.2
12	Proposed_variance	0.3415	9.2780	0.1887	4.8575	129.2124	10832.3970	67551.4911	11218.2
13	Model_variance	5.9980	34.1001	2.3578	15.5471	565.5454	25984.6390	218065.4230	34953.3

3) RQ3: Can the Master-Slave Surrogate Models Proposed in This Paper Effectively Help to Select Elite Individuals?: To solve RQ3, we compare the proposed master-slave surrogate models with the existing comModel. The detailed experimental data is shown in Table V. The meaning of each row in the table is consistent with Table III. For details, please see above.

Firstly, rows 3-5 of Table V show the data on time consumption. The data reveals the following: 1) The time consumption of test data generation using the master-slave surrogate models is lower than that of comModel. Among them, QR_value has the smallest time consumption under both methods, which is 17.6s and 19.0s, respectively. the time consumption of ClustalW has the highest time consumption, which is 851546.9s and 1122640.5s, respectively. For all programs, the average time consumption is 126588.8s and 168563.5s, respectively. 2) The test case generation speed of master-slave surrogate models is higher than that of comModel for all of them. The degree of enhancement varies among different programs, and the enhancement spans from 7.3% to 49%. For all programs, the average improvement is 31.2%.

Secondly, the data on the average number of evaluations is shown in rows 6-8 of Table V. 1) Mpi_Dijkstra exhibits the lowest average number of evaluations, with values of 203 and 248.3, respectively. In contrast, ClustalW consistently has the highest average number of evaluations, amounting to 14547.3 and 18023.9, respectively. The average across all programs is 3934.2 and 6110.3, respectively. 2) The lift rate of the number of evaluations by the proposed method varies for different programs. The average efficiency improvement is 35%.

Finally, the data on the success rate of the two methods in generating test cases are in rows 9-11 of Table V, and the results are as follows: 1) When compared to comModel, the proposed method achieves a significantly higher success rate. The average success rates for the two methods are 89.6% and 81.4%, respectively. 2) The success rates of the two methods exhibit significant variations across different programs. Among them, the largest difference in success rates is observed in generating QR_value test cases, with a margin of 32%. The average difference across all programs is 8.2%.

From the above experimental results, it can be seen that the proposed master-slave surrogate models can achieve the purpose of reducing testing costs on the premise of efficiently generating test cases. This indirectly proves that the proposed method of using the master-slave surrogate models can effectively select elite individuals.

4) RQ4: Can Selecting Elite Individuals to Execute Programs Effectively Reduce Cost in Evaluating Individuals?: For RQ4, the proposed method is compared with the nonElite to determine whether the elite selection method is effective. The experimental data is shown in Table VI. The meaning in the table is the same as Table III. See the above text for details.

First, it is clear from rows 3-5 of Table VI: 1) The proposed method has lower time consumption than nonElite in all programs. Among them, the program with the least average time

TABLE VI
EXPERIMENTAL COMPARISON BETWEEN METHODS OF SELECTING ELITE INDIVIDUALS AND THOSE OF NOT SELECTING ELITE INDIVIDUALS

1	Program	Mpi_Dijkstra	Convex	QR_value	Cjacobi	Heat	DepSolver	ClustalW	Average
2	$\|BP_s\|$	20	15	11	17	19	227	468	111
3	Proposed/s	33.2	28.13	17.4	41.1	931.4	33799.2	829309.3	123451.3
4	nonElite/s	36.5	30.39	20.3	54.8	1298.9	59872.3	1086391.6	163957.8
5	Lifting rate/%	9.0	7.4	14.2	25.0	28.2	43.5	26.3	21.9
6	Proposed	182.9	4834.1	182.7	3433.3	491.0	4255	14907.5	4020.9
7	nonElite	214.1	5440.1	220.5	6791.1	984.1	10695	20240.0	6369.2
8	Lifting rate/%	14.6	11.1	17.1	49.4	50.1	60.2	26.3	32.6
9	Proposed/%	100	100	100	94	95	95	85	95.5
10	nonElite/%	86	100	100	91	90	95	80	91.7
11	Difference/%	14	0	0	3	5	0	5	3.8
12	Proposed_variance	0.5175	5.0765	0.0971	3.9439	60.9296	17025.1347	74595.2390	13098.7
13	nonElite_variance	6.7956	8.7699	1.9131	7.3446	142.3872	16637.4791	156885.4351	24812.8

TABLE VII
EXPERIMENTAL COMPARISON BETWEEN DIFFERENT METHODS

1	Program	Mpi_Dijkstra	Convex	QR_value	Cjacobi	Heat	DepSolver	ClustalW	Average
2	$\|BP_s\|$	20	15	11	17	19	227	468	111
3	Proposed/s	33.1	33.0	17.3	58.8	774.3	33791.2	811254.8	120851.7
4	SAEO/s	36.8	84.9	19.7	82.7	1477.5	55735.5	1210113.6	181078.6
5	Lifting rate/%	10.1	61.1	12.2	28.8	45.5	39.9	32.9	33.2
6	Proposed	239.5	5230.9	191.1	4187.1	445	4041.4	13785.0	4017.1
7	SAEO	304.5	7399.1	293.3	5719.6	2062.5	8674.2	19082.5	6219.3
8	Lifting rate/%	21.3	29.3	34.8	26.7	78.4	53.4	27.7	38.8
9	Proposed/%	91	100	100	82	100	90	80	91.8
10	SAEO/%	82	100	86	72	80	90	80	84.2
11	Difference/%	9	0	14	10	20	0	0	7.6
12	Proposed_variance	0.3918	12.0855	0.2426	11.2179	65.4241	5194.3230	76684.6510	11709.7
13	SAEO_variance	4.7125	39.0510	1.9760	24.7512	425.7232	12810.9525	198146.8589	30207.7

consumption is QR_value, which are 17.4s and 20.3s, respectively. ClustalW has the highest time consumption, which are 829309.3s and 1086391.6s, respectively. The average time consumption of all programs under the two methods are 123451.3s and 163957.8s, respectively. 2) DepSolver has the largest improvement, which is 43.5%. The improvement of other programs is between 7.4% and 28.2%. Considering seven programs, the proposed method achieves an average lift rate of 21.9%.

Secondly, the comparison data of the two methods in average number of evaluations are shown in rows 6-8 of Table VI. 1) The proposed method holds a clear advantage in reducing number of evaluations. Among them, Mpi_Dijkstra has the smallest difference in average number of evaluations, which are 182.9 and 214.1, respectively. ClustalW has the largest difference in average number of evaluations, which are 14907.5 and 20240.0, respectively. For all programs, the average number of evaluations of the two methods is 4020.9 and 6369.2, respectively. 2) In terms of the difference in lift rate, Convex has the smallest difference of 11.1%, and DepSolver has the largest difference of 60.2%. The comprehensive average lift rate is 32.6%.

Finally, regarding the success rate, as shown in rows 9-11 of Table VI. 1) The proposed method has a success rate of 100% in three programs of all programs, and the lowest success rate is 85% for ClustalW. The nonElite method has a 100% success rate in two programs, and the lowest success rate is 80% for ClustalW. The average success rates of the two methods are 95.5% and 91.7%, respectively. 2) The difference in success rate between different programs is different. Convex, QR_value and DepSolver have the same success rate on both methods. Mpi_Dijkstra has the largest difference between the two methods, which is 14%. The average success rate improvement of the proposed method is 3.8%.

The proposed method has higher stability than the nonElite in generating path coverage test data, as seen from the smaller data standard deviation values for all programs under test in rows 12-13 of Table VI. This shows that executing programs for individuals with good estimates can effectively reduce the cost of individual evaluation.

5) RQ5: Can the Proposed Method Improve the Efficiency of Generating Test Cases?: To address RQ5, we compare the existing SAEO with the proposed method. The detailed experimental data is presented in Table VII. The meaning of each row in this table is consistent with Table III. For details, please refer to the above text.

First, rows 3-5 in Table VII give the average time consumption of the proposed model and SAEO. 1) The average time consumption of the proposed model is lower than that of SAEO. The average time consumption of the two methods is 120851.7s and 181078.6s, respectively. 2) Our proposed model is more rapid in generating test cases than SAEO The degree of lift rate varies for different programs, and the lift span is 10.1%-61.1%. Among them, Convex has the highest lift rate, which is 61.1%. An average lift rate increase of 33.2% has been achieved for all programs.

Secondly, rows 6-8 of Table VII show the average number of evaluations for both methods. 1) QR_value has the least average number of evaluations under the two methods, which are 191.1 and 293.3, respectively. The average number of evaluations are still the highest for ClustalW, which are 13785.0 and 19082.5, respectively. The average number of evaluations of all programs is 4017.1 and 6219.3, respectively. 2) For different programs, the lift rate of the proposed method in number of evaluations is different. Among them, Mpi_Dijkstra has the least improvement, which is 21.3%. Heat has the highest improvement, which is 78.4%. The average efficiency improvement is 38.8%.

Finally, rows 9-11 of Table VII show the data on the success rate of the two methods in generating test cases. The data reveals the following: 1) The success rate of the proposed method is significantly higher than that of SAEO. Among them, the proposed method achieved a 100% success rate in the Convex, QR_value and Heat. The SAEO only achieved a 100% success rate in the Convex. The average success rates of the two methods are 91.8% and 84.2%, respectively. 2) The difference in success rates between the two methods in different programs is large. Of these, the difference in the success rate of Heat is the largest, reaching 20%. The average difference for all programs is 7.6%.

Based on the aforementioned experimental results, it is observed that the proposed method achieved the objective of reducing test costs while efficiently generating test cases. This indirectly demonstrated that the proposed method improved the efficiency of test case generation.

In summary, from all the questions above, it can be seen that 1) the dual population strategy is advantageous in generating test data for path coverage of a parallel program, 2) the improved FERPSO-IMPR can generate better individuals earlier, 3) using the master-slave surrogate models can effectively select elite individuals, 4) selecting a small number of elite individuals to execute program can effectively reduce cost in assessing individuals, 5) the proposed method is beneficial to improve the efficiency of generating test data. In addition, the above five parts can be combined separately to match different programs. Therefore, the proposed method has high scalability.

V. Threats to Validity

During the experiment, we found two main factors that may affect the experimental results: system factors and random factors.

System factors are mainly caused by the parameters required in the algorithm. For example: If the value of *Olim* is set too high, it will lead to the scarcity of excellent individuals, thus hindering effective information migration. To alleviate this problem, we determined the optimal value of *Olim* through multiple experiments. Similarly, the settings of other parameters such as *loser*, *lprop*, and *NPsQ* were also determined based on the results of multiple experiments. In addition, the parameter selection of FERPSO refers to the standard settings of the original paper [44]. Judging from the experimental results, the settings of these parameters are reasonable.

Random factors are usually caused by external causes. Take the representativeness of the tested program as an example: if the selected program lacks representativeness, the experimental results will be difficult to apply to other programs. Therefore, we consider several aspects such as the number of program code lines, the number of communication primitives, and the number of target paths. In addition, the performance of experimental equipment will also affect the experimental results. To minimize this effect, we performed 30 independent tests in each experiment and averaged the results.

VI. Conclusion and Future Work

We combine FERPSO-IMPR and surrogate-assisted models to generate test cases for MPI program path coverage testing. This method potentially significantly enhances the efficiency of automated test sample generation. In this method, FERPSO-IMPR employs a dual population strategy to initialize test data and calculate fitness. Then, we create a sample set based on the initial data and its fitness. Subsequently, we train master-slave surrogate models to predict individual fitness. Finally, we select a small number of elite individuals to execute the program, obtaining the true fitness. This true fitness is used to determine whether to generate the required test data and guide the subsequent process.

In order to evaluate the effectiveness of our proposed method, we conduct tests on seven MPI programs from five different perspectives. The conclusive results clearly indicate that the approach outlined in the paper effectively reduces the cost of path coverage testing for parallel programs and enhances testing efficiency, rendering it highly competitive.

However, it's important to note that the proposed method includes many preset parameters. Determining the optimal values for these parameters can be a resource-consuming process, which may increase testing costs. Therefore, a key direction that needs to be addressed is reducing the number of parameter values in the method or achieving adaptive parameter values. In addition, the current research direction of this paper is test case generation, but the type of test cases that better detect program defects is very important. In view of this, studying how to combine test case generation and test case selection to better solve software testing problems is necessary.

References

[1] I. B. Peng, S. Markidis, R. Gioiosa, G. Kestor, and E. Laure, "MPI streams for HPC applications," *New Front. High Perform. Comput. Big Data*, vol. 30, p. 75, 2017.

[2] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Comput.*, vol. 22, no. 6, pp. 789–828, 1996.

[3] B. Sun, J. Wang, D. Gong, and T. Tian, "Scheduling sequence selection for generating test data to cover paths of MPI programs," *Inf. Softw. Technol.*, vol. 114, pp. 190–203, 2019.

[4] P. Xiao, B. Liu, and S. Wang, "Feedback-based integrated prediction: Defect prediction based on feedback from software testing process," *J. Syst. Softw.*, vol. 143, pp. 159–171, 2018.

[5] H. Alaqail and S. Ahmed, "Overview of software testing standard ISO/IEC/IEEE 29119," *Int. J. Comput. Sci. Netw. Secur.*, vol. 18, no. 2, pp. 112–116, 2018.

[6] A. C. Hausen, S. R. Vergilio, S. R. Souza, P. S. Souza, and A. S. Simao, "A tool for structural testing of MPI programs," in *Proc. 8th IEEE Latin-Amer. Test Workshop (LATW)*, 2007, pp. 1–6.

[7] H. Li, Z. Chen, and R. Gupta, "Efficient concolic testing of MPI applications," in *Proc. 28th Int. Conf. Compiler Constr.*, New York, NY, USA: ACM, 2019, pp. 193–204.

[8] P. S. Souza, S. R. Souza, and E. Zaluska, "Structural testing for message-passing concurrent programs: An extended test model," *Concurrency Comput.: Pract. Experience*, vol. 26, no. 1, pp. 21–50, 2014.

[9] S. Xanthakis, C. Ellis, C. Skourlas, A. L. Gall, S. Katsikas, and K. Karapoulios, "Application of genetic algorithms to software testing," in *Proc. 5th Int. Conf. Softw. Eng. Appl. (SEDA)*, Piscataway, NJ, USA: IEEE, 1992, pp. 625–636.

[10] T. Tian and D. Gong, "Test data generation for path coverage of message-passing parallel programs based on co-evolutionary genetic algorithms," *Autom. Softw. Eng.*, vol. 23, pp. 469–500, 2016.

[11] G.-G. Wang and Y. Tan, "Improving metaheuristic algorithms with information feedback models," *IEEE Trans. Cybern.*, vol. 49, no. 2, pp. 542–555, Feb. 2019.

[12] B. Sun, D. Gong, and X. Yao, "Integrating DSGEO into test case generation for path coverage of MPI programs," *Inf. Softw. Technol.*, vol. 153, 2023, Art. no. 107068.

[13] D. Gong, B. Sun, X. Yao, and T. Tian, "Test data generation for path coverage of MPI programs using SAEO," *ACM Trans. Softw. Eng. Method.*, vol. 30, no. 2, pp. 1–37, 2021.

[14] J. Yan and J. Zhang, "An efficient method to generate feasible paths for basis path testing," *Inf. Process. Lett.*, vol. 107, no. 3–4, pp. 87–92, 2008.

[15] R. Zhou and E. A. Hansen, "Breadth-first heuristic search," *Artif. Intell.*, vol. 170, no. 4–5, pp. 385–408, 2006.

[16] Z. Xu and J. Zhang, "A test data generation tool for unit testing of C programs," in *Proc. 6th Int. Conf. Qual. Softw. (QSIC)*, Piscataway, NJ, USA: IEEE, 2006, pp. 107–116.

[17] M. Berkelaar, K. Eikland, and P. Notebaert, "lpsolve: Open source (mixed-integer) linear programming system," *Eindhoven Univ. Technol.*, Eindhoven, vol. 63, 2004.

[18] N. Jain and R. Porwal, "Automated test data generation applying heuristic approaches—A survey," in *Proc. Comput. Society India Softw. Eng.*, Singapore: Springer, 2019, pp. 699–708.

[19] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Trans. Softw. Eng.*, vol. 44, no. 2, pp. 122–158, Feb. 2018.

[20] Z. Letko, "Sophisticated testing of concurrent programs," in *Proc. 2nd Int. Symp. Search Based Softw. Eng. (SSBSE)*, Benevento, Italy. Piscataway, NJ, USA: IEEE, 2010, pp. 36–39.

[21] R. P. Pargas, M. J. Harrold, and R. R. Peck, "Test-data generation using genetic algorithms," *Softw. Test. Verification Rel.*, vol. 9, no. 4, pp. 263–282, 1999.

[22] C.-S. D. Yang, A. L. Souter, and L. L. Pollock, "All-du-path coverage for parallel programs," *ACM SIGSOFT Softw. Eng. Notes*, vol. 23, no. 2, pp. 153–162, 1998.

[23] R. F. Vilela, V. H. Pinto, T. E. Colanzi, and S. R. Souza, "Bio-inspired optimization of test data generation for concurrent software," in *Proc. Int. Symp. Search Based Softw. Eng. (SSBSE)*, Tallinn, Estonia: Springer, 2019, pp. 121–136.

[24] M. Shousha, L. C. Briand, and Y. Labiche, "A UML/SPT model analysis methodology for concurrent systems based on genetic algorithms," in *Proc. MoDELS*, 2008, pp. 475–489.

[25] B. Sun, D. Gong, T. Tian, and X. Yao, "Integrating an ensemble surrogate model's estimation into test data generation," *IEEE Trans. Softw. Eng.*, vol. 48, no. 4, pp. 1336–1350, Apr. 2022.

[26] C. Zhu, L. Xu, and E. D. Goodman, "Generalization of Pareto-optimality for many-objective evolutionary optimization," *IEEE Trans. Evol. Comput.*, vol. 20, no. 2, pp. 299–315, Apr. 2016.

[27] S. Mirjalili, "Evolutionary algorithms and neural networks," in *Studies in Computational Intelligence*, vol. 780, Berlin, Germany: Springer, 2019.

[28] M. Dorigo, M. Birattari, and T. Stutzle, "Ant colony optimization," *IEEE Comput. Intell. Mag.*, vol. 1, no. 4, pp. 28–39, Nov. 2006.

[29] R. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," in *Proc. 6th Int. Symp. Micro Mach. Human Sci. (MHS)*, Piscataway, NJ, USA: IEEE, 1995, pp. 39–43.

[30] Z. Lin, B. Tang, W. Sun, J. Li, Y. Li, and Z. Su, "Position estimation based on the intelligent optimization algorithm," in *Proc. IEEE 6th Int. Conf. Signal Image Process. (ICSIP)*, Piscataway, NJ, USA: IEEE, 2021, pp. 568–572.

[31] X. Li, "A multimodal particle swarm optimizer based on fitness Euclidean-distance ratio," in *Proc. 9th Annu. Conf. Genet. Evol. Comput. (GECCO)*, New York, NY, USA: ACM, 2007, pp. 78–85.

[32] R. Chang, S. Sankaranarayanan, G. Jiang, and F. Ivancic, "*Software testing using machine learning*," U.S. Patent 8,924,938, Google Patent, 2014.

[33] H. Christiansen and C. M. Dahmcke, "A machine learning approach to test data generation: A case study in evaluation of gene finders," in *Proc. Int. Workshop Mach. Learn. Data Mining Pattern Recognit. (MLDM)*, Berlin, Heidelberg, Germany: Springer, 2007, pp. 742–755.

[34] J. Luo, A. Gupta, Y.-S. Ong, and Z. Wang, "Evolutionary optimization of expensive multiobjective problems with co-sub-Pareto front Gaussian process surrogates," *IEEE Trans. Cybern.*, vol. 49, no. 5, pp. 1708–1721, May 2019.

[35] G. Li, Q. Zhang, J. Sun, and Z. Han, "Radial basis function assisted optimization method with batch infill sampling criterion for expensive optimization," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Piscataway, NJ, USA: IEEE, 2019, pp. 1664–1671.

[36] T. Chugh, Y. Jin, K. Miettinen, J. Hakanen, and K. Sindhya, "A surrogate-assisted reference vector guided evolutionary algorithm for computationally expensive many-objective optimization," *IEEE Trans. Evol. Comput.*, vol. 22, no. 1, pp. 129–142, Feb. 2018.

[37] Y. Liu, W. Chen, J. Hu, X. Zheng, and Y. Shi, "Ensemble of surrogates with an evolutionary multi-agent system," in *Proc. IEEE 21st Int. Conf. Comput. Supported Cooperative Work Des. (CSCWD)*, Piscataway, NJ, USA: IEEE, 2017, pp. 521–525.

[38] W. Luo, R. Yi, B. Yang, and P. Xu, "Surrogate-assisted evolutionary framework for data-driven dynamic optimization," *IEEE Trans. Emerg. Topics Comput. Intell.*, vol. 3, no. 2, pp. 137–150, Apr. 2019.

[39] H. Wang, Y. Jin, C. Sun, and J. Doherty, "Offline data-driven evolutionary optimization using selective surrogate ensembles," *IEEE Trans. Evol. Comput.*, vol. 23, no. 2, pp. 203–216, Apr. 2019.

[40] Y. Jin and B. Sendhoff, "Reducing fitness evaluations using clustering techniques and neural network ensembles," in *Proc. Genet. Evol. Comput. Conf. (GECCO)*, Seattle, WA, USA: Springer, 2004, pp. 688–699.

[41] M. Müller, B. de Supinski, G. Gopalakrishnan, T. Hilbrich, and D. Lecomber, "*Dealing with MPI bugs at scale: Best practices, automatic detection, debugging, and formal verification*," Slides Presented in this Tutorial, Integrating Presentations from Dresden, Allinea, LLNL, and Utah. Accessed: Jan. 23, 2024. [Online]. Available: http://www.cs.utah.edu/fv/publications/sc11_with_handson.pptx

[42] H. Yu, "Combining symbolic execution and model checking to verify MPI programs," in *Proc. 40th Int. Conf. Softw. Eng.: Companion Proc. (ICSE)*, New York, NY, USA: ACM, 2018, pp. 527–530.

[43] A. Windisch, S. Wappler, and J. Wegener, "Applying particle swarm optimization to software testing," in *Proc. 9th Annu. Conf. Genetic Evol. Comput. (GECCO)*, New York, NY, USA: ACM, 2007, pp. 1121–1128.

[44] D. Guo, Y. Jin, J. Ding, and T. Chai, "Heterogeneous ensemble-based infill criterion for evolutionary multiobjective optimization of expensive problems," *IEEE Trans. Cybern.*, vol. 49, no. 3, pp. 1012–1025, Mar. 2019.