The Journal of Systems and Software 155 (2019) 130–144

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

In Practice

Genetic algorithm based test data generation for MPI parallel programs with blocking communication

Tian Tian[a], Dunwei Gong[b,*], Fei-Ching Kuo[c,1], Huai Liu[c]

[a] School of Computer Science and Technology, Shandong Jianzhu University, Jinan, Shandong, 250101, PR China
[b] School of Information and Control Engineering, China University of Mining and Technology, Xuzhou, Jiangsu, 221116, PR China
[c] Department of Computer Science and Software Engineering, Swinburne University of Technology, Hawthorn, VIC, 3122, Australia

ARTICLE INFO

Article history:
Received 21 April 2017
Revised 25 March 2019
Accepted 11 April 2019
Available online 12 April 2019

Keywords:
Parallel program
Blocking communication
Path coverage
Test data
Genetic algorithm

ABSTRACT

Parallel computing is one of mainstream techniques for high-performance computation in which MPI parallel programs have gained more and more attention. Genetic algorithms (GAs) have been widely employed in automated test data generation, leading to a major family of search-based software testing techniques. However, previous GA-based methods have limitations when testing MPI parallel programs with blocking communication. In this paper, we focus on the path coverage problem for MPI parallel programs with blocking communication, and formulate the problem as an optimization problem with its decision variable being the program input and the execution order of sending nodes. In addition, we develop target amending strategies for candidates when solving the problem using genetic algorithms. The proposed method is evaluated and compared with several state-of-the-art methods through a series of controlled experiments on five typical programs. The experimental results show that the proposed method can effectively and efficiently generate test data for path coverage.

1. Introduction

A *parallel program* consists of two or more processes, which run in parallel and interact with each other according to the mechanisms of communication and synchronization, to solve scientific or engineering problems. Concurrency is the property differentiating the parallel program from the serial program, in which operations are executed in a fixed sequence.

M**essage- **P**assing **I**nterface* (MPI***) is a communication middleware through which programmers can establish communication among processes in a multiprocessor system. It is one of the most important and widely used parallel programming libraries due to its excellent compatibility and efficiency (Snir et al., 1998). In MPI, sending and receiving messages are crucial for passing messages between two processes. A process that sends a message is regarded as the *sender*, and the one that receives the message is considered as the receiver.

When sender and/or message tags are unspecified in receiving functions, and more than one process tends to send a message to the same receiver simultaneously, and the sending operations of these processes may be completed in different orders, resulting in different execution traces even if the program runs against the same program input, which is the so-called nondeterministic execution of MPI programs. MPI provides two kinds of communication modes: *blocking* and *non-blocking*. On the one hand, the non-blocking mode is beneficial to the execution efficiency of parallel programs. However, MPI programs with the non-blocking pattern typically behave more unpredictable than programs with the blocking counterpart. Meanwhile, the non-blocking mode requires additional auxiliary functions when transmitting messages. On the other hand, the blocking mode program is more logically clear and more controllable than the non-blocking mode, although the blocking mode has drawbacks in the communication and calculation overlap of MPI programs. Various software systems have generally been developed using ***MPI*** in the Blocking-Communication Parallel mode (Barker, 2015). In the rest of this paper, we will call this kind of software ***MPI-BCP*** software (***BCP*** software, for short).

Software testing is a primary software engineering process to guarantee software quality, with its focus on detecting errors in software. One key testing activity is designing test data to run software and observing software behavior during the run. Various studies have shown that *genetic algorithm* (GA) is an effective method of automated generating test data (Malhotra and Khari, 2013; Harman and McMinn, 2010), and code coverage is a sensible

* Corresponding author.
E-mail address: dwgong@vip.163.com (D. Gong).
[1] It is with deep regret and sadness that we report the passing of the third author Fei-Ching Kuo on October 6, 2017.

https://doi.org/10.1016/j.jss.2019.04.049
0164-1212/

NOTATION

p	A parallel program
p^i	The ith process of p
n^i_j	The jth basic unit of p^i
$\mathbb{P}^i$	The given path in p^i
$\mathbb{P}$	The given program path
X	The program input
ndp	A non-deterministic point
ndp_k	The kth non-deterministic point
DNP	The set of non-deterministic points
π_k	The execution sequence of sending nodes corresponding to ndp_k
D_{π_k}	The set of execution sequences of sending nodes corresponding to ndp_k
$\lvert D_{\pi_k}\rvert$	The size of D_{π_k}
π^j_k	The jth execution sequence of sending nodes in D_{π_k}
$\tilde{X}$	The decision variable or a solution
D_X	The domain of X
z	The number of non-deterministic points
$D_{\tilde{X}}$	The domain of $D_{\tilde{X}}$
$\xi^i(\tilde{X})$	The sub-path in p^i traversed by $\tilde{X}$
$\xi(\tilde{X})$	The path traversed by $\tilde{X}$
iL	The length of an encoded input
nkL	The length of encoded execution orders of sending nodes associated with kth ndp
L	The length of an encoded candidate solution
C	The set of encoded candidate solutions
DC	The set of candidate solutions
c_j	The jth encoded candidate solution
a^{nk}_j/b^{nk}_j	The jth digit of execution orders of sending nodes associated with the kth ndp
a^i_j/b^i_j	The jth bit of input encodings
pos	The crossover or mutation position
O	The ordered set that presents the sequence to be amended
$\hat{O}$	The set that contains all the process identifiers associated with ndp corresponding to the sequence to be amended
o^*	The element under examination
o^m	The digit after mutation

parameter in designing the fitness function of GA for software testing.

Tian and Gong applied a traditional genetic algorithm (TGA) (Tian and Gong, 2013) and a co-evolutionary genetic algorithm (CGA) (Tian and Gong, 2016) to generate test data for BCP software. Both methods are code-based coverage testing, with the purpose of generating test data to execute software through a given path. In TGA, test data generation is based on the assumption of correct communication among processes. For CGA, the experiments are controlled such that more than one sender cannot simultaneously send a message to the same receiver to avoid non-deterministic behavior in parallel programs.

In this study, the crucial characteristics of BCP software, such as typical sending/receiving operator and non-deterministic behavior, are fully taken into account. We reformulated the path coverage problem and identified the program input and the execution order of sending nodes as the decision variable. Besides, we developed target amending strategies for candidates at each round of the evolution. Essentially, this paper has the following twofold contributions: (1) formulating the problem of test data generation for path coverage as an optimization problem with its decision variable being the program input and the execution order of sending nodes, and (2) developing target amending strategies for candidates obtained by the crossover and mutation operators when solving the formulated problem using genetic algorithms.

The rest of this paper is organized as follows. The background information of parallel programs is introduced in Section 2. Section 3 presents how to formulate the problem of generating test data and explains our amending strategies for candidates obtained by the crossover and mutation operators in the GA-based method of generating test data for BCP software. The proposed method is evaluated through a series of experiments. The experimental settings, including research questions, object programs, framework of test data generation and experimental design are detailed in Section 4; while the experimental results are analyzed and discussed in Section 5. Section 6 discusses the related work. Finally, Section 7 concludes the whole paper, and identifies potential directions for future research.

2. Background

2.1. MPI

A parallel program is composed of multiple processes that execute the same or different code in parallel. Any process in the program needs to call MPI_Init to initialize the MPI environment at the initial stage, and MPI_Finalize to exit from the MPI environment at the final stage. Other MPI functions, e.g., MPI_Send and MPI_Recv, are called after MPI_Init and before MPI_Finalize. Signatures and parameters of these two methods are given in Tables 1 and 2 (extracted from Snir et al., 1998).

In MPI_Send, “dest” and “tag” must be set explicitly. By contrast, “source” and “tag” in MPI_Recv can be set either explicitly by giving the sender information, or implicitly by using two constants, namely MPI_ANY_TAG and MPI_ANY_SOURCE, which mean that any tag and any sender are acceptable.

2.2. BCP software

Parallel software is the program with m processes executed concurrently and cooperating with each other by communication. A set $\mathrm{p} = \{\mathrm{p}^1, \mathrm{p}^2, \ldots, \mathrm{p}^m\}$ is used to denote the software with these m processes. Behavior of parallel software is often complex, and thus is normally modeled using the control-communication flow graph, which is a directed graph composed of nodes and edges. Definitions of node and edge are given below:

Node: A node is a *basic unit* of a process. When running the program with an input, either all elements contained in the node are executed or none of them is executed. The node associated with MPI_Send function call is named a *sending node*, while the one associated with MPI_Recv is named a *receiving node*. Two processes communicate with each other through their sending/receiving nodes.

Edge: An edge is a link between two nodes. There are two types of edges in the control-communication flow graph. For two nodes, if one node is executed after another of the same process, the edge between these two nodes is called a *control edge*. On the other hand, if two nodes are located in different processes, with one being a *sender* and the other being the corresponding *receiver*, there exists a *communication edge* between these two nodes. Since the control-communication flow graph is a directed graph, every communication edge has a direction from sending node to receiving node.

With the definitions of node and edge, a control-communication flow graph can be defined as follows:

Table 1
MPI_Send.

int MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)	
Parameter	Description
buf	Initial address of the sending buffer
count	Size of the buffer for this MPI_Send operation
datatype	Data type of each element sent. *[Rule:For Process A to transmit data to Process B, the type of data in MPI_Send operation of Process A needs to match that in MPI_Recv operation of Process B.]*
dest	Identifier of the receiver
tag	Message tag
comm	This parameter specifies the scope of communication, that is, which processes can be notified about this MPI_Send operation. Note that MPI_COMM_WORLD is a default scope of communication that includes all processes involved in a parallel program.

Table 2
MPI_Recv.

int MPI_Recv (void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)	
Parameter	Description
buf	Initial address of the receiving buffer
count	size of the buffer for this MPI_Recv operation
datatype	Data type of each element received. *[Rule:For Process A to transmit data to Process B, the type of data in MPI_Send operation of Process A needs to match that in MPI_Recv operation of Process B.]*
source	Identifier of the sender
tag	Message tag
comm	This parameter specifies the scope of communication, that is, which processes can be notified about this MPI_Recv operation.
status	This parameter captures the number and tag of receive data, as well as the sender information if the MPI_Recv operation was successful. It could be a list of errors if this operation failed.

Control-communication flow graph: A control-communication flow graph of parallel software is a directed graph, denoted as $G = \{V, E\}$, where V is the node set, and E is the edge set.

Let us consider one example of BCP software, called Gcd (Krawczyk et al., 1994), which computes the maximal common divisor of three integers. The code of Gcd is given in Figs. 1 and 2, while its control-communication flow graph is shown in Fig. 3. As shown in the graph, Gcd involves three parallel processes, $\mathrm{p} = \{\mathrm{p}^1, \mathrm{p}^2, \mathrm{p}^3\}$. p^1 runs the code in Fig. 1, and p^2 and p^3 have the same function and run the code in Fig. 2. Both processes employ functions in MPI to set up the communication between processes. Though running p^1, p^2, and p^3 in parallel, the program reads three inputs, x_1, x_2, x_3, through p^1, and assigns the calculation task to p^2 and p^3. To show the link between the graph and code, the identifier of a basic unit in Fig. 3 is added to the left side of the code in Figs. 1 and 2.

In the control-communication flow graph of Gcd, the 2-5th node of p^1 and the 9th node of p^2 and p^3 are the sending nodes, while the 6-7th node of p^1, the 2-3th node of p^2 and p^3 are the receiving nodes. $< n_9^1, n_{10}^1 >$ is a control edge between n_9^1 and n_{10}^1, and $< n_2^1, n_2^2 >$ is a communication edge between n_2^1 and n_2^2. Every communication edge involves two processes, which are called *sender* and *receiver*, respectively. For example, when p^1 encounters the function of n_2^1, it will be blocked until the message with the destination of p^2 and the tag of 1 is successfully sent. Similarly, when p^2 encounters the function of n_2^2, it will be blocked until the message from p^1 with the tag of 1 is received. In other words, n_2^2 in p^2 matches n_2^1 in p^1. Consequently, a message is transmitted from p^1 to p^2 through cooperation between p^1 and p^2. p^1 is the sender and p^2 is the receiver for the communication edge $< n_2^1, n_2^2 >$.

Communication flow, also known as *communication sequence* between processes depends not only on the program logic in the source code, but also on parameter settings upon MPI_Recv invocation. When the sender information is specified in every call of MPI_Recv methods, the communication flow between processes is fully predetermined. For instance, in the Gcd program, p^1 sends two messages to p^2 and p^3, respectively. The *sender* and *message* tags are both specified during the MPI_Recv operation in p^2 and p^3. These specific tags in MPI_Recv impose a fixed communication flow, that is, $< n_2^1, n_2^2 >, < n_3^1, n_3^2 >, < n_4^1, n_2^3 >, < n_5^1, n_3^3 >$. As a result, the order of messages arrived at the 2-3th node of p^2 and p^3 is deterministic. On the other hand, if the sender information is

```
   int main(int argc, char**argv){
n1 int x1,x2,x3;
n1 MPI_Status status;
n1 FILE *fp=fopen("input.txt","r");
    /*initialize the parallel environment*/
n1  MPI_Init(&argc,&argv);
n1  fscanf(fp,"%d%d%d",&x1,&x2,&x3);
   /*Send x1 and x2 to p2*/
n2  MPI_Send(&x1,1,MPI_INT,2,1,MPI_COMM_WORLD);
n3  MPI_Send(&x2,1,MPI_INT,2,2,MPI_COMM_WORLD);
   /*Send x2 and x3 to p3*/
n4  MPI_Send(&x2,1,MPI_INT,3,1,MPI_COMM_WORLD);
n5  MPI_Send(&x3,1,MPI_INT,3,2,MPI_COMM_WORLD);
   /*Receive results from p2 and p3*/
n6  MPI_Recv(&x1,1,MPI_INT,MPI_ANY_SOURCE,
              MPI_ANY_TAG,MPI_COMM_WORLD,&status);
n7   MPI_Recv(&x2,1,MPI_INT,MPI_ANY_SOURCE,
              MPI_ANY_TAG,MPI_COMM_WORLD,&status);
   /*Calculate the greatest common divisor*/
n8  while(x1!=x2) {
n9    if(x1<x2)
n10      x2=x2-x1;
      else
n11      x1=x1-x2;
n12 }
   /*Print the result and log out from the parallel environment */
n13 printf("The greatest common divisor is %d\n",x1);
n13 MPI_Finalize();
n13 return 0;
    }
```

Fig. 1. Code for the process, p^1, in Gcd.

```
        int main(int argc, char**argv){
n1^2 / n1^3 int x1,x2;
n1^2 / n1^3 MPI_Status status;
        /*Initialize the parallel environment*/
n1^2 / n1^3 MPI_Init(&argc,&argv);
        /*Receive the data from p1*/
n2^2 / n2^3 MPI_Recv(&x1,1,MPI_INT,1,1,MPI_COMM_WORLD,&status);
n3^2 / n3^3 MPI_Recv(&x2,1,MPI_INT,1,2,MPI_COMM_WORLD,&status);
        /*Calculate the greatest common divisor*/
n4^2 / n4^3 while(x1!=x2) {
n5^2 / n5^3     if(x1<x2)
n6^2 / n6^3       x2=x2-x1;
              else
n7^2 / n7^3       x1=x1-x2;
n8^2 / n8^3   }
        /*Send the result to p1*/
n9^2 / n9^3 MPI_Send(&x1,1,MPI_INT,1,1,MPI_COMM_WORLD);
        /*Log out from the parallel environment*/
n10^2 / n10^3 MPI_Finalize();
n10^2 / n10^3 return 0;
        }
```

Fig. 2. Code for the processes, p^2 and p^3, in Gcd.

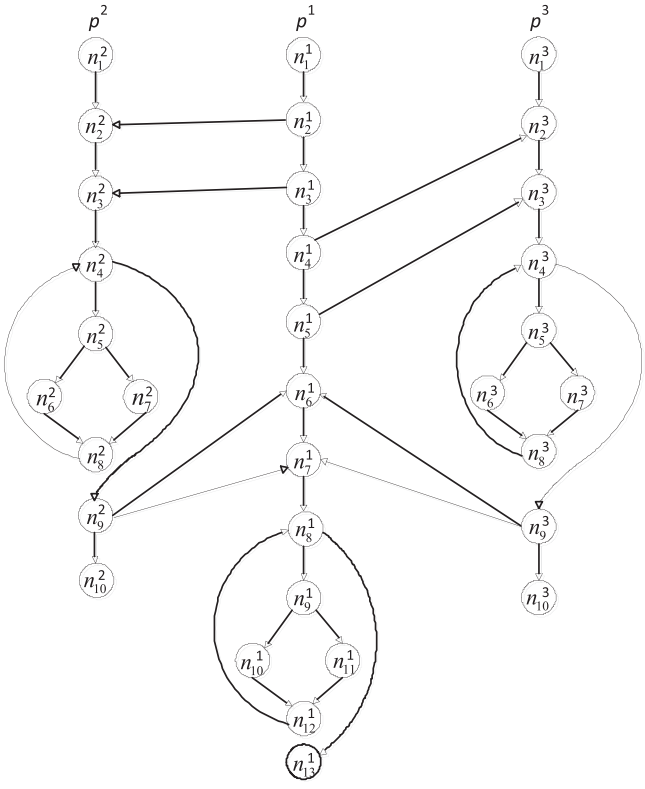

Fig. 3. Control-communication flow graph of program Gcd.

missing in some MPI_Recv calls, the related communication flow is non-deterministic and only revealed at runtime.

For example, Fig. 3 shows that both p^2 and p^3 (processes sharing the same code in Gcd program) have a node (i.e., the 9th node in p^2 and p^3) with 2 outgoing edges to p^1, one edge to n_6^1 and the other edge to n_7^1. This drawing indicates that both receiving nodes (n_6^1 and n_7^1) in p^1 expect to receive messages from one of these two sendings nodes (n_9^2 and n_9^3) without specifying the sender information. Due to such missing information at receiving nodes (n_6^1 and n_7^1), both italicize the phrase "execution sequences of sending nodes", that is, (n_9^2, n_9^3) and (n_9^3, n_9^2) are considered valid to n_6^1 and n_7^1 nodes in p^1. Since processes in parallel software are run in a unpredictable manner, Gcd software is foreseen to experience either one of communication flows, $(< n_9^2, n_6^1 >, < n_9^3, n_7^1 >)$ or $(< n_9^3, n_6^1 >, < n_9^2, n_7^1 >)$ - which flow will be taken is determined at runtime.

BCP software exhibits non-deterministic behavior only when it experiences non-deterministic communication sequences, that is, when two or more receiving nodes in the same process expect messages from more than one sender without specifying the sender information in their MPI_Recv statement. Because it is an important feature of any BCP software, we will call each set of such receiving nodes "a non-determinsitic point in the control-communication flow graph", or simply *"non-determinsitic point"* (abbreviated as *ndp*). We can recognize each of such points easily from the graph because each point is a group of at least 2 receiving nodes in the same process, and each of these nodes is connected with more than one arrows (one arrow per incoming edge).

There could be more than one non-deterministic points, $DNP = \{ndp_1, ndp_2, \ldots, ndp_z\}$, in BCP software, where $z \geq 1$. Each $ndp_i \in DNP$ is associated with a set of execution sequences of sending nodes, denoted as $D_{\pi_k} = \{\pi_k^1, \pi_k^2, \ldots, \pi_k^{|D_{\pi_k}|}\}$. In Gcd, $DNP = \{ndp_1\}$ and $ndp_1 = \{n_6^1, n_7^1\}$. Besides, $|D_{\pi_1}| = 2, \pi_1^1 = (n_9^2, n_9^3)$ and $\pi_1^2 = (n_9^3, n_9^2)$. Hence, $D_{\pi_1} = \{(n_9^2, n_9^3), (n_9^3, n_9^2)\}$.

Coverage testing is about executing software with inputs in order to cover as many components as possible to observe software behavior during the execution. To cover a component (such as statement), software needs to traverse through certain paths that lead to that statement. Definition of *process path* and that of *path* are required to understand our proposed method in this paper.

Process path: Process path is a path associated with a process upon execution of the BCP software. For example, $(n_1^2, n_2^2, n_3^2, n_4^2, n_9^2, n_{10}^2)$, $(n_1^2 n_2^2 n_3^2 n_4^2 n_5^2 n_6^2 n_8^2 n_4^2 n_9^2 n_{10}^2)$ and $(n_1^2 n_2^2 n_3^2 n_4^2 n_5^2 n_7^2 n_8^2 n_4^2 n_9^2 n_{10}^2)$ are three distinct process paths for the process, p^2.

Path: A *path* is an execution trace of the software p when p runs with an input $X \in D_X$. The execution trace includes several nodes, control edges and communication edges. Given that p has m processes executed concurrently, the path traversed by the input X is composed of m *process paths*.

Suppose that the Gcd program runs with $X = (4, 4, 6)$, and the execution order of sending nodes in ndp_1 is $\pi_1^1 = (n_9^2, n_9^3)$. The execution trace after running the software with X will be $\{n_1^1 n_2^1 n_3^1 n_4^1 n_5^1 n_6^1 n_7^1 n_8^1 n_9^1 n_{11}^1 n_{12}^1 n_8^1 n_{13}^1$, $n_1^2 n_2^2 n_3^2 n_4^2 n_9^2 n_{10}^2$, $n_1^3 n_2^3 n_3^3 n_4^3 n_5^3 n_6^3 n_8^3 n_4^3 n_5^3 n_7^3 n_8^3 n_4^3 n_9^3 n_{10}^3\}$. However, if the program runs with the same input but the execution order of the sending nodes is $\pi_1^2 = (n_9^3, n_9^2)$, the process path related to the process p^1 will become $n_1^1 n_2^1 n_3^1 n_4^1 n_5^1 n_6^1 n_7^1 n_8^1 n_9^1 n_{10}^1 n_{12}^1 n_8^1 n_{13}^1$, but other process paths will remain the same.

3. Proposed method

3.1. Formulating the problem of generating test data

As mentioned before, execution orders of sending nodes are non-determistic in BCP software. Without considering this factor in the problem formulation like the previous studies, the search will not be precise and can cause more trial-and-error cycles than required to find the right solution (that is, solution always bring software execution through the same path).

Therefore, we reformulate the search problem as follows: seeking *an input* and *an execution order of sending nodes* to drive the software execution through a given path $\mathbb{P} = \{\mathbb{P}^1, \mathbb{P}^2, \ldots, \mathbb{P}^m\}$, where m is the number of process paths in $\mathbb{P}$.

A solution $\tilde{X}$ is composed of the execution sequence of sending node $\pi_k \in D_{\pi_k}$ for each ndp_k and an input $X \in D_X$, and $D_{\tilde{X}} = D_X \cdot \prod_{k=1}^{z} D_{\pi_k}$.

Let a given path be $\mathbb{P} = \{\mathbb{P}^1, \mathbb{P}^2, \ldots, \mathbb{P}^m\}$ and the path traversed by $\tilde{X}$ be $\varepsilon(\tilde{X}) = \{\varepsilon^1(\tilde{X}), \varepsilon^2(\tilde{X}), \ldots, \varepsilon^m(\tilde{X})\}$. The similarity between $\mathbb{P}^i$ and $\varepsilon^i(\tilde{X})$, $\delta(\mathbb{P}^i, \varepsilon^i(\tilde{X}))$, can be defined as follows:

$$\delta(\mathbb{P}^i, \varepsilon^i(\tilde{X})) = \frac{g(\mathbb{P}^i, \varepsilon^i(\tilde{X}))}{\max(|\mathbb{P}^i|, |\varepsilon^i(\tilde{X})|)} \tag{1}$$

where $g(\mathbb{P}^i, \varepsilon^i(\tilde{X}))$ means the number of common nodes between $\mathbb{P}^i$ and $\varepsilon^i(\tilde{X})$ in the same position.

A path is actually an execution order of nodes. Measurement of the path similarity should start from the beginning. Further research has indicated that the more successive common nodes are from the first node, the more closer the candidate solution, $\tilde{X}$, approaches the desired solution. Taking the position of a node into account, the similarity between $\mathbb{P}^i$ and $\varepsilon^i(\tilde{X})$ can be further enhanced as follows:

$$\delta(\mathbb{P}^i, \varepsilon^i(\tilde{X})) = n(\mathbb{P}^i, \varepsilon^i(\tilde{X})) \frac{g(\mathbb{P}^i, \varepsilon^i(\tilde{X}))}{\max(|\mathbb{P}^i|, |\varepsilon^i(\tilde{X})|)} \tag{2}$$

where $n(\mathbb{P}^i, \varepsilon^i(\tilde{X}))$ refers to the number of common successive nodes between $\mathbb{P}^i$ and $\varepsilon^i(\tilde{X})$ from the first node.

Given Formula (2), the similarity between $\mathbb{P}$ and $\varepsilon(\tilde{X})$ can be redesigned as follows:

$$f(\tilde{X}) = \sum_{i=1}^{m} \left(\delta(\mathbb{P}^i, \varepsilon^i(\tilde{X}))\right) \tag{3}$$

Finally, the formulation for the path coverage problem can be defined below as to maximize the similarity between $\mathbb{P}$ and $\varepsilon(\tilde{X})$.

$$\begin{aligned} &\max\ f(\tilde{X}) \\ &s.t.\ \tilde{X} \in D_{\tilde{X}} \end{aligned} \tag{4}$$

It is clear that when $f(\tilde{X})$ reaches the maximal value(i.e., the sum of the length of process paths in $\mathbb{P}$), $\tilde{X}$ will be the *desired solution*.

It can be concluded from Formula (4), the program input and the execution order of sending nodes are selected as the decision variable, and the similarity between the executed path and the given one is defined as the objective function. Together with the constraint that reflects the domain of the program input and that of the execution order, the above problem can be formulated as an optimization problem. It differs the formulation of the problem for serial programs as well as for BCP software given in our earlier work in the decision variable. For the latter, only the program input is regarded as the decision variable when formulating the problem. Even for BCP software, the execution order is not taken into consideration when selecting the decision variable in our earlier work.

3.2. Encoding of candidate solutions

Ways of encoding software inputs have been well studied. There exist standard encoding schemes for primary data types, such as boolean, binary and integer data types. We will follow these standard schemes to encode the input data of software under study. On the other hand, we are not aware of any study which encodes the execution order of sending nodes. We will use a sequence of integers to encode this order, where each integer is the identifier of a process corresponding to a sending node. The length of an encoded candidate solution $L = iL + n1L + n2L + \ldots + nzL$.

In the Gcd program, there are three input variables x_1, x_2 and x_3 of integer type and one non-deterministic point ($ndp_1 = \{n_6^1, n_7^1\}$). Assume that the input range for each variable is [1,32]. We could employ 5 binary bits to encode all 32 values for each input variable. Accordingly, the sum of binary bits for x_1, x_2 and x_3 make up the length of encoded software input; hence $iL = 15$. Besides, 2 single digits are enough to encode π_1 associated with ndp_1, indicating that $n1L = 2$. These decisions will produce 17 digits encoding scheme, where the first 15 digits are the collection of 3 sets of 5-binary bits. In this case, a candidate solution $\tilde{X}$ with the input of (26, 17, 10) and the execution order of (n_9^3, n_9^2) will be encoded as (11010100010101032) where "32" corresponds to π_1.

3.3. Amending illegal candidate solutions

Operators such as crossover and mutation are necessary to GA. As mentioned above, the input of BCP software and the execution order of its sending nodes are encoded in different ways. Besides, the value ranges for these encoded data are different. However, the crossover and mutation operators can possibly result in an illegal candidate solution. This problem will be illustrated in detail below.

Crossover operator: Let two encoded candidate solutions be

$$c_1 = a_1^i a_2^i \ldots a_{iL}^i a_1^{n1} a_2^{n1} \ldots a_{n1L}^{n1} a_1^{n2} a_2^{n2} \ldots a_{n2L}^{n2} \ldots a_1^{nz} a_2^{nz} \ldots a_{nzL}^{nz}$$
$$c_2 = b_1^i b_2^i \ldots b_{iL}^i b_1^{n1} b_2^{n1} \ldots b_{n1L}^{n1} b_1^{n2} b_2^{n2} \ldots b_{n2L}^{n2} \ldots b_1^{nz} b_2^{nz} \ldots b_{nzL}^{nz}$$

where $a_1^i a_2^i \ldots a_{iL}^i$ and $b_1^i b_2^i \ldots b_{iL}^i$ are the input encoding and $a_1^{nk} a_2^{nk} \ldots a_{nkL}^{nk}$ and $b_1^{nk} b_2^{nk} \ldots b_{nkL}^{nk}$ are the encodings of π_k. When *pos* is in the range of $(1, iL)$, after performing the crossover operator, the newly generated two encoded candidate solutions, denoted as c_1' and c_2', are represented as follows:

$$c_1' = a_1^i a_2^i \ldots b_{pos}^i \ldots b_{iL}^i b_1^{n1} b_2^{n1} \ldots b_{n1L}^{n1} b_1^{n2} b_2^{n2} \ldots b_{n2L}^{n2} \ldots b_1^{nz} b_2^{nz} \ldots b_{nzL}^{nz}$$
$$c_2' = b_1^i b_2^i \ldots a_{pos}^i \ldots a_{iL}^i a_1^{n1} a_2^{n1} \ldots a_{n1L}^{n1} a_1^{n2} a_2^{n2} \ldots a_{n1L}^{n2} \ldots a_1^{nz} a_2^{nz} \ldots a_{nzL}^{nz}$$

Similarly, when *pos* falls in the range of $(iL + 1, iL + \sum_{k=1}^{z} njL)$, the crossover will take place in the encoding scheme of an execution order of sending nodes (for example, π_k). As a result, the two newly generated encoded candidate solutions become as follows:

$$c_1'' = a_1^i a_2^i \ldots a_{iL}^i a_1^{n1} a_2^{n1} \ldots a_{n1L}^{n1} \ldots a_1^{nk} a_2^{nk} \ldots b_{pos}^{nk} \ldots b_{nkL}^{nk} \ldots b_1^{nz} b_2^{nz} \ldots b_{nzL}^{nz}$$
$$c_2'' = b_1^i b_2^i \ldots b_{iL}^i b_1^{n1} b_2^{n1} \ldots b_{n1L}^{n1} \ldots b_1^{nk} b_2^{nk} \ldots a_{pos}^{nk} \ldots a_{nkL}^{nk} \ldots a_1^{nz} a_2^{nz} \ldots a_{nzL}^{nz}$$

After the crossover operation, π_k in this two candidate solutions may, i.e., $a_1^{nk} a_2^{nk} \ldots b_{pos}^{nk} \ldots b_{nkL}^{nk}$ in c_1'' and $b_1^{nk} b_2^{nk} \ldots a_{pos}^{nk} \ldots a_{nkL}^{nk}$ in c_2'', contain duplicate sender identifiers.

Our strategy of amending this problem is detailed below.

Take $a_1^{nk} a_2^{nk} \ldots b_{pos}^{nk} \ldots b_{nkL}^{nk}$ in c_1'' as an example. This sequence can be presented as an ordered set, $O = (a_1^{nk}, a_2^{nk}, \ldots, b_{pos}^{nk}, \ldots, b_{nkL}^{nk})$. Let o^* denote the element under examination, $Q = O \backslash \{o*\}$ and examination rule is "the value of o^* does not appear in Q".

A method of **A**mending execut**I**on sequen**C**e of **S**ending nodes caused by crossover(abbreviated as AICS-Crossover) is detailed in Algorithm 1. For each element in O, if o^* is found to be against the examination rule (Lines 5–10), then replace the value of o^* by another process identifier not shown in Q but associated with the kth ndp (Lines 11–13). In this way, a_j^{nk} has a distinct value from other elements in O after amendment. Consequently, AICS-Crossover guarantees that all elements in O will have distinct values.

An example is given as follows. Let us say that two candidate solutions are $(17, 12, 26, n_9^3, n_9^2)$ and $(18, 23, 29, n_9^2, n_9^3)$. After encoding, they will look like the following

$$c_1 = 10001011001101032$$
$$c_2 = 10010101111110123$$

Suppose $pos = 17$, the newly generated two encoded candidate solutions will become:

$$c_1'' = 10001011001101033$$
$$c_2'' = 10010101111110122$$

Algorithm 1 AICS-Crossover.

Input: O and $\hat{O}$
Output: The amended sequence, O

```
1: Set sign = 0, where sign records whether the element in O needs to be amended
2: for each element o* ∈ O
3:   Set Q = O\{o*}
4:   Set sign=0
5:   for each Q(i) ∈ Q, where Q(i) is the value of the ith element in Q
6:     if Q(i) == o*
7:       Set sign=1 to indicate that o* needs to be amended
8:       break
9:     end_if
10:  end_for
11:  if sign == 1
12:    o* = the first element ∈ Ô\Q
13:  end_if
14: end_for
15: Output O
```

Obviously, there exists an illegal execution order of sending nodes because two sender identifers in one *ndp* cannot be identical but two digits representing these identifies become identical as shown in the 16th to the 17th positions, 33 and 22, in c_1'' and c_2''. Our strategy to solve this problem is as follows. For **33** in c_1'', since the integer in the 16th position, 3, is equal to the one in the 17th position, it needs to be changed to 2, the identifer of p^2, which is another sender associated with ndp_1. Since there is no more digits after the 17th position, the amendment is done and hence c_1' will become:

$$c_1'' = 10001011001101023$$

Similarly, c_2' will be corrected as:

$$c_2'' = 10010101111110132$$

The decoding of c_1'' and c_2'' become $(17, 12, 26, n_9^2, n_9^3)$ and $(18, 23, 29, n_9^3, n_9^2)$, respectively, both showing a legal execution order of sending nodes.

Mutation operator: Let an encoded candidate solution be

$$c_1 = a_1^i a_2^i \dots a_{iL}^i a_1^{n1} a_2^{n1} \dots a_{n1L}^{n1} a_1^{n2} a_2^{n2} \dots a_{n2L}^{n2} \dots a_1^{nz} a_2^{nz} \dots a_{nzL}^{nz}$$

After mutation, the digit in this position is replaced by its allele. As mentioned before, when *pos* falls in the range of $(iL+1, iL+\sum_{j=1}^{z} njL)$, the mutated candidate solution may contain an illegal execution order of sending nodes. A method to **A**mend execut**I**on sequen**C**e of **S**ending nodes caused by mutation (abbreviated as AICS-Mutation) is detailed in Algorithm 2.

Algorithm 2 AICS-Mutation.

Input: o^m, O and $\hat{O}$
Output: The amended sequence, O

```
1: for each element o* ∈ O
2:   if o* == o^m
3:     break
4:   end_if
5: end_for
6: o* = the element ∈ Ô\O
7: Output O
```

Assume c_1''' is created by mutating a_{pos}^{nk} of c_1,

$$c_1''' = a_1^i a_2^i \dots a_{iL}^i a_1^{n1} a_2^{n1} \dots a_{n1L}^{n1} \dots a_1^{nk} a_2^{nk} \dots a_{pos}'^{nk} \dots a_{nkL}^{nk} \dots a_1^{nz} a_2^{nz} \dots a_{nzL}^{nz}$$

where $a_{pos}'^{nk}$ represents the digit after mutation. Further assume $a_1^{nk} a_2^{nk} \dots a_{pos}'^{nk} \dots a_{nkL}^{nk}$ is an illegal encoding of π_k. Lines 1–5 intend to seek for the integer equal to the mutated one, $a_{pos}'^{nk}$, while Line 6 is used to change $a_{pos}'^{nk}$ into a different process identifier, which is associated with the kth *ndp* but not shown in the existing list.

Consider the example of Gcd program. Let the encoded candidate solution be

$$c_1 = 10101111001001032$$

It is decoded as $(21, 28, 18, n_9^3, n_9^2)$. If $pos = 17$, the newly produced one becomes 10101111001001033, where the integer in the 17th position is taken place by its allele, 3. According to our amendment strategy, the integer in the 16th position is equal to the mutated one, 3, and needs to be changed. Since there are only two related senders, p^2 and p^3, associated with ndp_1, the integer in the 16th position is changed into 2, the identifer of p^2. After amendment, the illegal solution 10101111001001033 becomes:

$$c_1''' = 10101111001001023$$

which is decoded as $(21, 28, 18, n_9^2, n_9^3)$.

Given the fact that candidate solutions obtained by the crossover and mutation operators may be illegal due to the involvement of the execution order, target strategies are presented for amending them. After amendment, the resulted candidates are guaranteed to be legal, which can be further employed to run the program under test. Since the existing studies do not involve the execution order in the decision variable, it is of unnecessity to amend candidates obtained by the crossover and mutation operators.

3.4. Pseudo-code of test data generation by using GAs

The pseudo-code of generating test data for BCP software is given in Algorithm 3. The software under test, p, is instrumented. *DC* includes n candidate solutions which are initially generated from the solution domain($D_{\tilde{X}}$). Here, n means the population size in GA approach. A finite iterative process (Lines 5–22) works with the goal of finding out test data to cover the given path. Each candidate solution is used to run p and the traversed path is stored in *EP*. If *EP* contains the given path, $\mathbb{P}$, namely test data to cover the given path are found, the iteration is terminated; otherwise, all solutions in *DC* are encoded to compose *C*. The fitness of each encoded solution c_i is calculated using formula (4). The candidate solutions with high fitness values are retained in *C* by selection (e.g., roulette wheel selection), and the crossover and mutation operators given in Section 3.3 are performed in *C* to generate new encodings. Decoding version of *C* is then replaced the old *DC* in the next iteration. After the iterative process is terminated, whether the desired test data is found is checked. If yes, they are the output; otherwise, there is no output. Algorithm 3 is to search an i**N**put and execut**I**on sequen**C**e of **S**ending nodes for each *ndp* for traversing a target path using GA approach. We will call this algorithm "NICS-GA" hereafter.

3.5. Difference between the proposed and previous methods

Similar to NICS-GA, CGA and TGA are two GA-based methods of generating test data for BCP software. Among these methods, CGA employs co-evolution to generate test data of BCP software with each receiving node explicitly specifying the sender and the message tags. In other words, non-deterministic points are purposely avoided in BCP software, implying that CGA only works for BCP software whose communication among processes is predetermined.

Algorithm 3 NICS-GA.

Input: The software under test, p, and given path, $\mathbb{P}$
Output: test data of covering $\mathbb{P}$

```
Instrument the software under test, p
Set C = {} and DC = {}
Set counter = 1 and sign = 0, where sign records whether the test data covering ℙ is found
Set DC = {dc_1, dc_2, ..., dc_n}, where dc_i is a candidate solution initially generated from the solution domain
while counter <maximum and sign == 0, where the terminal condition of the loop is that the test data that covers the given path are found(sign == 1) or the number of iterations reaches the maximum (counter == maximum)
    for each candidate solution dc_i ∈ DC
    ¡¡ Execute p with dc_i and append the path traversed by dc_i to EP
    end_for
    if ℙ ∈ EP
        sign=1
    end_if
    if sign==0
        Encode each candidate solution dc_i in DC and store the encoding solution c_i in C
        Calculate the fitness for all c_i ∈ C using Formula (4)
        Perform selection in C
        Perform crossover in C with the assistance of Algorithm 1
        Perform mutation in C with the assistance of Algorithm 2
        Empty DC
        Decode each candidate solution c_i in C and store the decoding solution dc_i in DC
    end_if
    Increment counter by 1
end_while
if sign==1
    Output the test data of covering ℙ
end_if
```

Different from CGA, TGA takes the influence of non-deterministic points on the execution of BCP software into consideration. In TGA, the equivalent path is defined, and a path is said to be equivalent to a given path, if and only if the difference in nodes between the path and the given one is caused only by non-deterministic communication. The equivalent path(s) of a given path is(are) first sought for according to *ndps*, and a traditional GA is then employed to generate test data for covering either the given path or any one of its equivalent one(s). Let $\mathbb{P}1$ be a given path, and $\mathbb{P}2$, $\mathbb{P}3, \ldots,$ and $\mathbb{P}n$ be its equivalent ones, then we have

$$\begin{aligned} &\max\{f_1(X), f_2(X) \ldots, f_n(X)\} \\ &s.t.\ X \in D_X \end{aligned} \tag{5}$$

Taking Gcd as an example, if a given path is $\mathbb{P}1 = \{\mathbb{P}1^1, \mathbb{P}1^2, \mathbb{P}1^3\} = \{n_1^1 n_2^1 n_3^1 n_4^1 n_5^1 n_6^1 n_7^1 \mathbf{n_8^1 n_9^1 n_{11}^1 n_{12}^1} n_8^1 n_{13}^1, n_1^2 n_2^2 n_3^2 n_4^2 n_9^2 n_{10}^2, n_1^3 n_2^3 n_3^3 n_4^3 n_5^3 n_6^3 n_8^3 n_4^3 n_5^3 n_7^3 n_8^3 n_4^3 n_9^3 n_{10}^3\}$, its equivalent path will be $\mathbb{P}2 = \{\mathbb{P}2^1, \mathbb{P}2^2, \mathbb{P}2^3\} = \{n_1^1 n_2^1 n_3^1 n_4^1 n_5^1 n_6^1 n_7^1 \mathbf{n_8^1 n_9^1 n_{10}^1 n_{12}^1} n_8^1 n_{13}^1, n_1^2 n_2^2 n_3^2 n_4^2 n_9^2 n_{10}^2, n_1^3 n_2^3 n_3^3 n_4^3 n_5^3 n_6^3 n_8^3 n_4^3 n_5^3 n_7^3 n_8^3 n_4^3 n_9^3 n_{10}^3\}$. Then the path coverage problem can be formulated as follows:

$$\begin{aligned} &\max\{f_1(X), f_2(X)\} \\ &s.t.\ X \in D_X \end{aligned} \tag{6}$$

In this example, TGA will output $X = (4, 4, 6)$ as a desired test datum since the objective in formula (6) reaches to the maximal value, 1, with the datum, meaning that the datum covers either $\mathbb{P}1$ or $\mathbb{P}2$. Due to non-deterministic communication in this program, this datum may not cover the given path, $\mathbb{P}1$, even although it is found.

Let us consider the proposed method. Given the fact that this program has only one non-deterministic point, $ndp_1 = \{n_6^1, n_7^1\}$, the solution, $\tilde{X}$ for Gcd will be $\{X, ndp_1\}$ and $D_{\tilde{X}}$ will be $D_X \cdot D_{ndp_1}$. By adding ndp_1 to the problem formulation, seeking for test data will be more precise than TGA. Instead of outputting $X = (4, 4, 6)$ for the path, $\mathbb{P}1$, the proposed method will output $\{(4, 4, 6), (n_9^2, n_9^3)\}$.

Essentially, the main difference between previous work and this study can be summarized as follows.

(1) The decision variable of the optimization problem formulated in this paper contains two parts, the program input and the execution order of sending nodes, whereas that of previous work includes only the program input.

(2) When employing GA to generate test data, this study develops target amending strategies for illegal candidates produced by the crossover and mutation operators. However, previous work is not involved in these strategies, given the fact that the decision variable does not include the execution order of sending nodes which raises illegal candidates.

4. Experimental study

A series of experiments have been conducted to evaluate the performance of the proposed NICS-GA method. We present the design and settings of the experiments in this section.

4.1. Research questions

In order to evaluate the performance of the proposed method, the following three research questions are raised.

- RQ1 How effectively does the proposed method generate test data for covering a path?
- RQ2 How efficient is the proposed method in finding the desired solution?
- RQ3 How is the effectiveness of test data for path coverage?

These research questions will be answered one by one through the controlled experiments.

4.2. Object softwares

Five BCP-MPI programs were selected as the test objects in the empirical study, as summarized in Table 3. They are either classical parallel programs or their extensions for evaluating parallel program testing techniques. Among these programs, Gcd+ extends Gcd (Krawczyk et al., 1994) by increasing the range of the program input. For Matrix, it multiplies two matrices (Souza et al., 2008), and its complexity is increased by adding other functions. Regarding SUT-MPI, it was implemented through extending and changing a function of the non-trivial benchmark, SkaMPI (Reussner et al, 2002). In addition, Calculating π (Fu et al., 2015) and Parallel Sorting (Rashid and Qureshi, 2006) are two real-world parallel applications. Max-CPI seeks the maximum datum from a series of data followed by calculating the value of π, and

Table 3
Basic information of each object software.

Software name	# of input parameters	# of processes	# of sending operations	# of receiving operations
Gcd+	4	7	18	18
Matrix	9	3	8	8
SUT-MPI	40	10	60	60
Rsort	16	16	51	51
Max-CPI	100	12	42	42

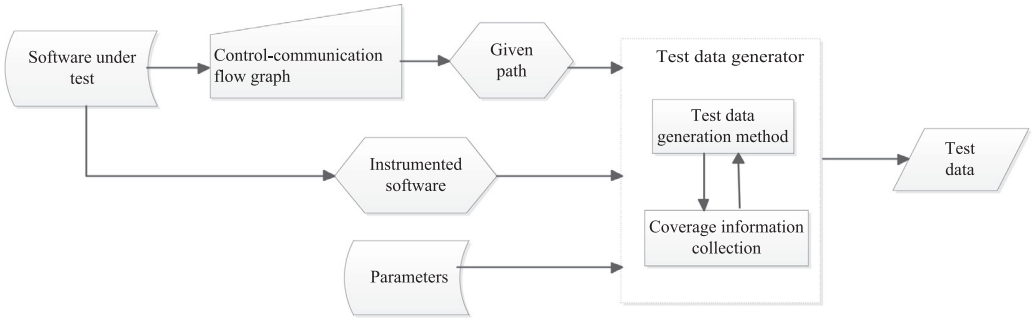

Fig. 4. Framework of test data generation.

Rsort mainly implements the function of parallel rank sorting. For more details of these programs, please refer to the Supplementary Material.

4.3. Variables and measures

4.3.1. Independent variable

The independent variable in this study is the test data generation technique.

NICS-GA proposed in this paper was definitely selected for our experiments. In addition, in view of the current study, TGA and CGA are two existing method of test data generation for BCP softwares. However, as discussed before, CGA is particularly applicable to specific types of BCP programs where sender information is fixed to avoid non-deterministic behavior. Therefore, it is unfair to compare the proposed method with CGA. In this study, we selected the random method and TGA as the baseline technique for comparison.

4.3.2. Dependent variable

If test data that covers the given path are generated by a method, the method is considered as successful. We applied the *success rate* to measure the effectiveness of the methods under study (that is, to answer RQ1). Let r is the total number of experimental runs (which is 20 in this study, as defined in the next section) and r_s is the successful number, then the success rate equals to $r_s/r * 100\%$.

Intuitively speaking, the greater the success rate is, the more effective a method is. Therefore, the success rate can be utilized to evaluate whether a method can effectively generate test data that covers the given path of BCP programs.

For measuring the efficiency (that is, for answering RQ2), we utilized two metrics, namely the *execution time* and *the number of evaluated candidate solutions*, to compare the proposed method and random method. The execution time refers to how much time is consumed in a run of a test data generation method. The number of evaluated candidate solutions means how many candidate solutions have been evaluated in a run. Besides, as mentioned in Section 3, TGA calculates the path similarity several times when evaluating a candidate, and tends to seek a program input that can bring the program execution through one of (equivalent) paths.

Therefore, we employed the number of calculating the path similarity to evaluate TGA and NICS-GA. Obviously, the smaller the values of these three metrics are, the more efficient a method is in finding desired solutions.

Moreover, mutation analysis was employed to answer RQ3. To fulfill this task, a number of changes are inserted into a program to generate its alternative versions according to predefined rules which mean mutation operators. Each alternative version of the program is called a mutant. If a mutant behaves the same as the program under all test data, it is called an equivalent mutant. The effectiveness of test data is evaluated by the mutation score which reflects the percentage of mutants which differ from the program. If a mutant produces an output or covers a path different from the program, it is called a killed mutant. Let M1 be the total number of mutants, M2 be the number of killed mutants, and M3 be the number of equivalent ones, then the mutation score is equal to $M2/(M1 - M3) * 100\%$.

4.4. Framework of test data generation

Test data is generated through the framework shown in Fig. 4. For a program under test, its control-communication flow graph is first constructed. Consequently, a path is selected as the given one. The selected path combined with the instrumented program and required parameters is input to the test data generator. The test data generator includes two parts: the test data generation method and coverage information collector. Here, the test data generation method is responded by one process, while the coverage information collector is responsible for repeatedly calling the instrumented program and collecting coverage information to the method. Correspondingly, the method can judge whether or not the test data that covers the given path are found. For NICS-GA, the fitness of a candidate solution is also calculated.

4.5. Experiment design

For each object program, a path was first chosen, and its feasibility was then checked. If it was feasible, it would be regarded as the given path; otherwise, the above steps were repeated until a feasible one was found. Note that the communication among processes was also investigated when checking the feasibility of a chosen path. Totally, there were 66 given paths for five object programs.

For each given path, and the experimental results of 20-time runs were recorded. Based on them, we calculated the success rate, the average execution time, and the mean number of evaluated candidate solutions and the mean number of calculating path similarities.

Note that parameter settings have an influence on the efficiency of evolutionary test data generation, and there have been no rules for setting parameters to date. A large population size generally results in much computational cost in GA. Therefore, a small population size is often chosen when applying GA to tackle a real-word problem on the premise of guaranteeing its effectiveness. Based on this, if it is complicated, a large population size is generally selected; otherwise, a small one is adopted. Based on this, different population sizes were selected as listed in Table 4 for different paths. GA includes two main operators, crossover and mutation. Among them, crossover is a major operator and mutation is a minor one when searching solutions to a problem. Consequently,

Table 4
Population size.

Software	path	population size
Gcd+	path1-path2	100
	path3-path4	20
Matrix	path1-path2	20
	path3	50
	path4	100
SUT-MPI	path1-path8	50
Rsort	path1-path20	50
Max-CPI	path1-path30	200

crossover is generally employed with a larger probability than mutation. Given the fact that the crossover and mutation probabilities of 0.9 and 0.3 are widely used in the literature, the same settings were adopted in this study. It should be noted that even though these values of parameters have been widely used in various methods of generating test data, they are not necessarily the best. It will be possible to get better results if we set more appropriate values of these parameters (Arcuri and Fraser, 2011; Garousi, 2008). Parameter tuning is not the focus of this paper.

The process of generating testing data using the random method is as follows. A candidate is randomly generated. If the path traversed by the candidate is just the given path, the method will be terminated; otherwise, the above steps are repeated until the maximal number of evaluated candidates is reached.

The Mann-Whitney *U*-test with a significance level of 0.05 was conducted to further verify the statistical significance of the difference in efficiency. The null hypothesis (H_0) for the statistical testing was that for one particular path, NICS-GA had the similar efficiency (either execution time or number of evaluated candidate solutions) to the comparative method. If the *p*-value is smaller than 0.05, H_0 was rejected, implying that the efficiency of NICS-GA was statistically significantly higher than that of the comparative method.

With respected to mutation analysis, a number of mutants were firstly produced according to predefined mutation operators. Five suggested mutation operators (Offutt et al., 1996), ABS, UOI, LCR, AOR, and ROR, were selected for the elements common with serial programs, and Table 5 lists the mutation operators which are special for MPI functions (Silva et al., 2012). Then, the mutation score was calculated. Moreover, there is a lack of approaches to effectively detecting equivalent mutants of MPI programs. Obviously, studies on effectively detecting equivalent mutants are out of the scope of this paper. In this study, equivalent mutants are manually detected and removed.

Table 5
Mutation operators for MPI functions.

Mutation operators	Mutation ways
Del	Delete a function
ReplModeSend	Replace MPI_Send with MPI_Ssend
ReplSource	Change MPI_ANY_SOURCE into other sources
ReplTag	Change MPI_ANY_TAG into other tags
ReplProbe	Replace MPI_Recv with MPI_Prob
ReplArg	Change the argument in a function into the one in other functions
ChanArg	Change the argument in a function into the one in this function
InsUnaArg	Insert an unary operator into arguments

5. Experimental results

5.1. Answer to RQ1

The success rates of NICS-GA and the random method are summarized in Figs. 5 and 6 and those of NICS-GA and TGA are listed in Fig. 7.

It is clearly shown from Figs. 5 and 6 that NICS-GA had the higher success rate than the random method for 66 given paths. Regarding the last two softwares, the success rate of the random method was zero for all the given paths, whereas NICS-GA reached the success rate of 100% and 65% – 100% for *Rsort* and *Max* – *CPI*, respectively. Fig. 7 also verifies that NICS-GA was more effective than TGA when generating test data for covering a given path.

5.2. Answer to RQ2

In the rest of this section, we will focus on the evaluation and comparison of efficiency for the three methods.

Figs. 8–13 report the mean value of a metric under 20 experimental runs. As mentioned in Section 5.1, for each given path of *Rsrot* and *Max* – *CPI*, the random method cannot generate expected test data in each experimental run. Consequently, the mean value of the number of evaluated candidate solutions was 500,000 and 2,000,000 for *Rsrot* and *Max* – *CPI*, respectively. For the sake of space, Figs. 10–13 only list the execution time and the number of evaluated candidate solutions of NICS-GA.

From Figs. 8–13 as well as Table 6, we can observe that NICS-GA normally spent shorter execution time and used fewer number of evaluated candidate solutions than the random method. In other words, the NICS-GA was more efficient than the random method. The results of the statistical testing are summarized in Table 6. In

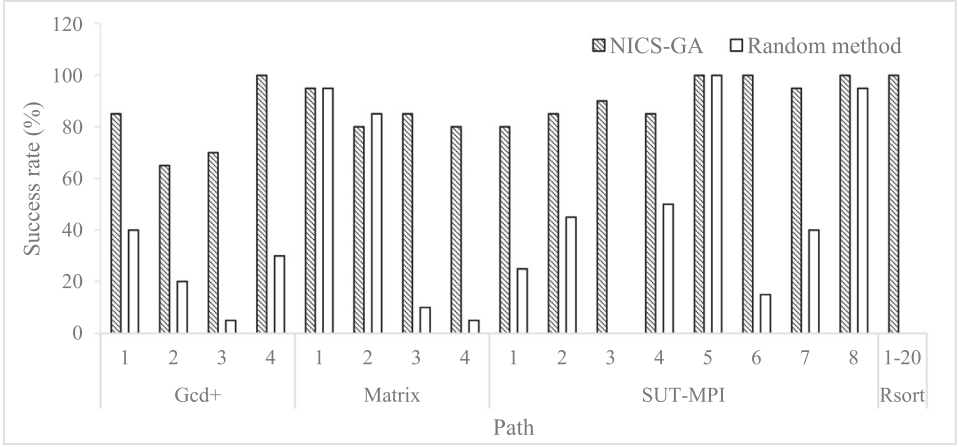

Fig. 5. Success rate(%) of NICS-GA and the random method on Gcd+, Matrix, SUT-MPI and Rsort.

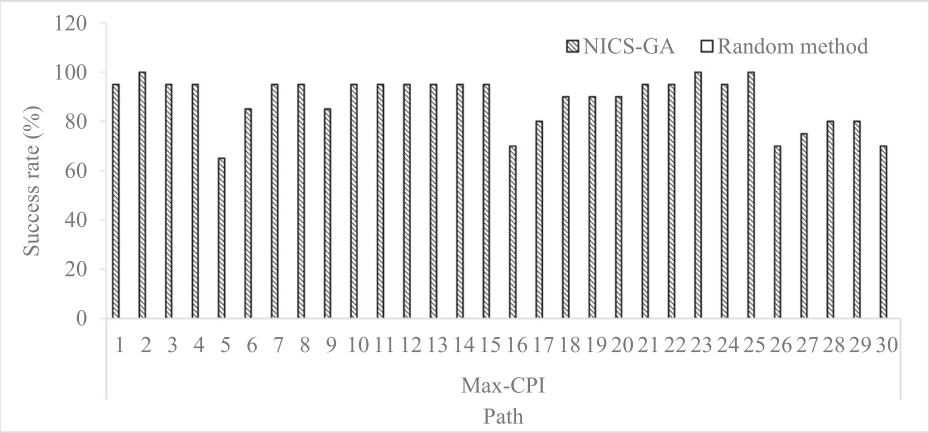

Fig. 6. Success rate(%) of NICS-GA and the random method on Max-CPI.

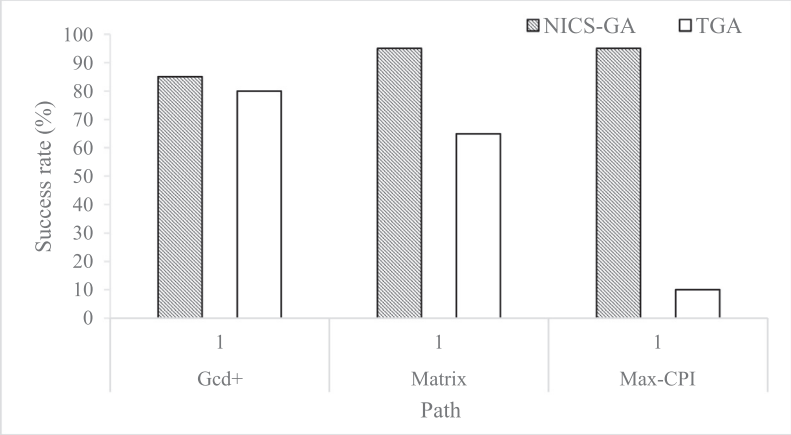

Fig. 7. Success rate(%) of NICS-GA and TGA.

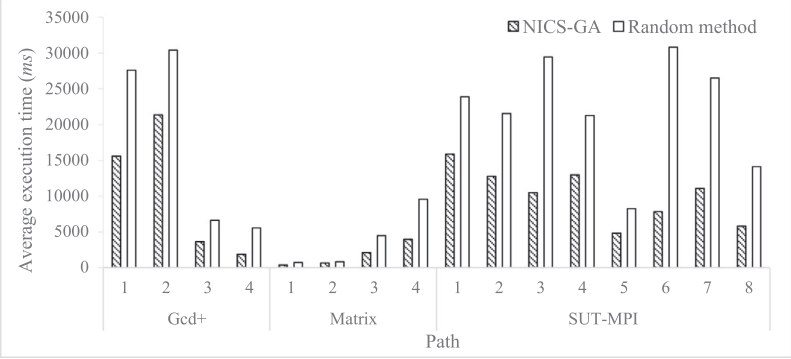

Fig. 8. Average execution time(ms) for NICS-GA and the random method on Gcd+, Matrix and SUT-MPI.

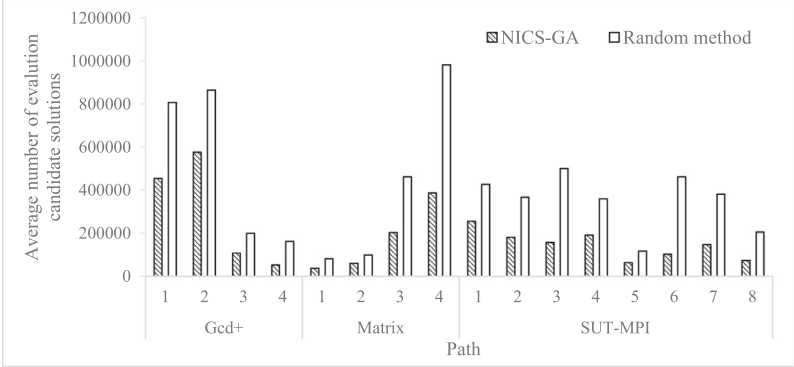

Fig. 9. Average number of evaluated candidate solutions for NICS-GA and the random method on Gcd+, Matrix and SUT-MPI.

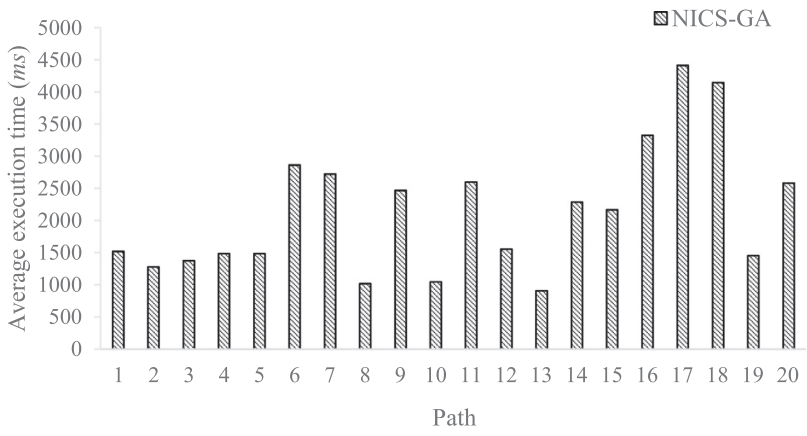

Fig. 10. Average execution time(ms) for NICS-GA on Rsort.

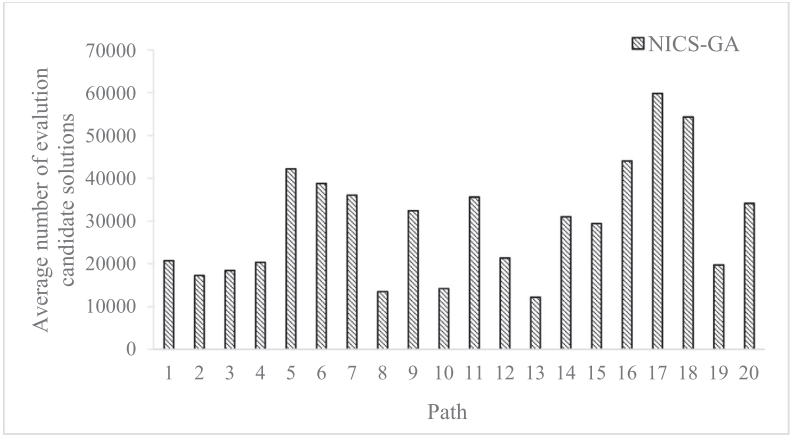

Fig. 11. Average number of evaluated candidate solutions for NICS-GA on Rsort.

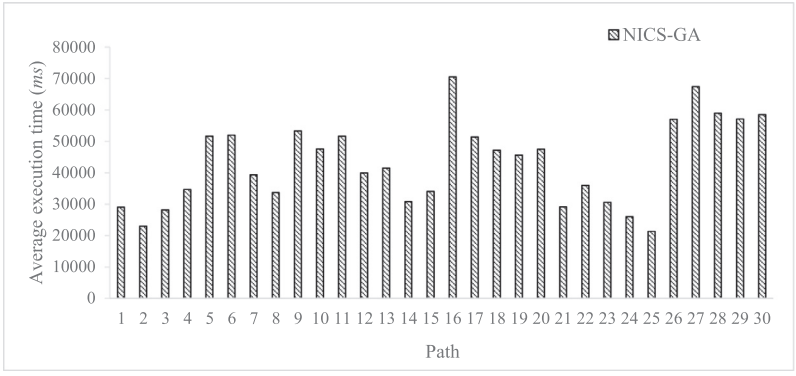

Fig. 12. Average execution time(ms) for NICS-GA on Max-CPI.

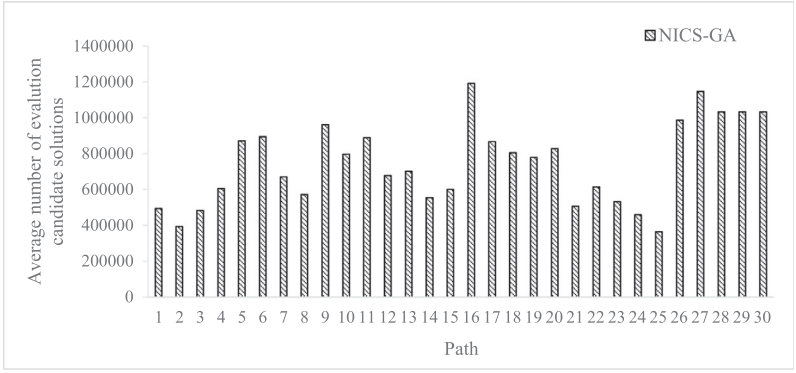

Fig. 13. Average number of evaluated candidate solutions for NICS-GA on Max-CPI.

Table 6
p-values of statistical testing for comparing the efficiency between NICS-GA and the random method.

Software	Path	Execution time	Number of evaluated candidate solutions
Gcd+	path1	0.010	0.002
	path2	0.372	0.007
	path3	0.007	< 0.001
	path4	< 0.001	< 0.001
Matrix	path1	0.011	0.008
	path2	0.079	0.030
	path3	0.001	< 0.001
	path4	0.001	< 0.001
SUT-MPI	path1	0.094	0.001
	path2	0.011	0.001
	path3	< 0.001	< 0.001
	path4	0.070	0.008
	path5	0.028	0.023
	path6	< 0.001	< 0.001
	path7	< 0.001	< 0.001
	path8	0.003	0.001
Rsort	path1–path20	< 0.001	< 0.001
Max-CPI	path1–path4	< 0.001	< 0.001
	path5	0.105	< 0.001
	path6	0.007	< 0.001
	path7–path8	< 0.001	< 0.001
	path9	0.001	< 0.001
	path10	0.001	< 0.001
	path11-path15	< 0.001	< 0.001
	path16	0.279	< 0.001
	path17	0.007	< 0.001
	path18	0.001	< 0.001
	path19	< 0.001	< 0.001
	path20	0.001	< 0.001
	path21–25	< 0.001	< 0.001
	path26	0.030	< 0.001
	path27	0.105	< 0.001
	path28	0.030	< 0.001
	path29	0.007	< 0.001
	path30	0.105	< 0.001

the table, "< 0.001" means that the *p*-value is very small (smaller than 0.001).

Based on Table 6, we can observe that NICS-GA significantly outperformed the random method on most of paths in terms of the execution time, and there was no significant difference on Gcd+-path2, Matrix-path2, path1 and 4 of SUT-MPI and path5, 16, 27 and 30 of Max-CPI. With respect to the number of evaluated candidate solutions, NICS-GA had significantly higher performance than the random method on 66 paths.

Since the difference between TGA and NICS-GA is the number of calculating the path similarity and success rate, which can be embodied by only one given path, path1 of each program among Gcd+, Matrix, and Max-CPI was selected to explain the performance of NICS-GA and TGA. Fig. 14 and Table 7 exactly show that there was no significant difference in terms of the execution time between NICS-GA and TGA, and Fig. 15 and Table 7 report that NICS-GA consumed less computation cost than TGA for two paths.

In summary, NICS-GA was more effective and efficient than the random method and TGA for generating test data that covers given paths.

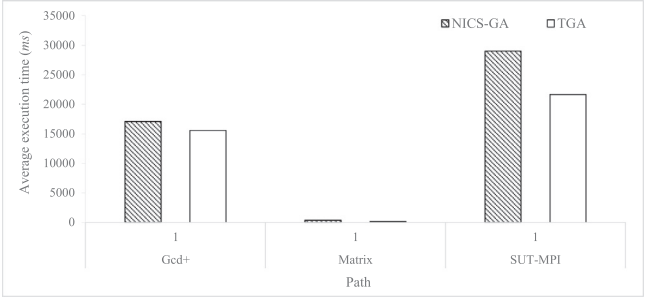

Fig. 14. Average execution time(ms) for NICS-GA and TGA.

5.3. Answer to RQ3

In order to answer RQ3, 2887 mutants were generated for Gcd+, Rsort and Max-CPI. We performed mutation analysis on these three programs based on a test datum that traverses a path of Gcd+ and Rsort, respectively, and six test data of covering six paths

Table 7
p-values of statistical testing for comparing the efficiency between NICS-GA and TGA.

Software	Path	Execution time	Number of evaluated candidate solutions
Gcd+	path1	0.685	< 0.001
Matrix	path1	0.766	0.279
Max-CPI	path1	0.871	< 0.001

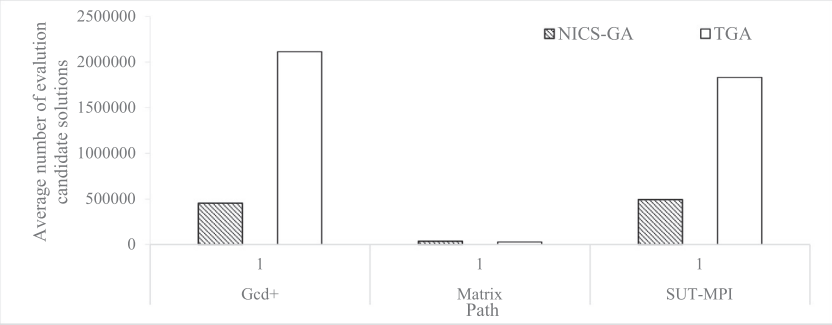

Fig. 15. Average number of evaluated candidate solutions for NICS-GA and TGA.

Table 8
Results of mutation analysis.

Program	# of mutants	# of equivalent mutants	mutation score(%)
Gcd+	578	88	93.1
Rsort	1113	249	97.4
Max-CPI	1196	298	96.1

of Max-CPI. Table 8 lists the high mutation scores of 93.1%, 97.4% and 96.1%. The reason is that a test datum includes an input and an execution order of sending nodes. Meanwhile, a path is composed of multiple process paths which cover information related to not only the control flow but the communication flow. Consequently, the variations in a program can be successfully detected by comparing the outputs and the traversed paths of the program and those of its mutants. Therefore, test data for path coverage can reveal faults resulted from computation and communication associated with parallel program characteristics, suggesting that test data for path coverage were effective in software testing.

6. Related work

6.1. Test data generation based on genetic algorithms

Search-based optimization algorithms have been widely used for studying software engineering (Harman et al., 2012). A lot of work has also been conducted to use various search techniques in test data generation (Malhotra and Khari, 2013; Harman and McMinn, 2010; Ali et al., 2010; Wu et al., 2015). Xanthakis et al. (1992) is a pioneer who employed GAs to automatically generate test data and established the foundation for the application of GAs in test data generation. Rauf and Anwar (2010) proposed the method of generating test data by using GAs for programs with graphic user interface. Arcuri (2010) applied search-based algorithms to automatically generate test data for the popular object-oriented software, and studied the influence of the length of a test sequence on the branch coverage. Rao and Govindarajulu (2015) applied genetic GAs to generate test data for killing mutants in mutation testing. Moreover, in the view that test data with high diversity are likely to improve the completeness and quality of software testing, Bueno et al. (2014) incorporated GAs with other meta-heuristics to generate diversity-oriented test data. 100% path coverage is a very stringent requirement and often infeasible. Some researchers studied the problem of multiple paths coverage. Ahmed and Hermadi (2008) obtained the desired test data for covering multiple paths by one population, while McMinn et al. (2006) utilized several sub-populations to seek for test data that covers multiple paths, aiming at improving the efficiency of generating test data. For the problem of many paths coverage, Gong et al. (2011) divided many paths into several groups according to path similarity and made use of a GA to generate test data covering the paths contained in each group.

The aforementioned methods mainly focused on serial programs. Since these methods omitted the unique characteristics of parallel programs, the problem formulation in them for test data generation is unsuitable for testing parallel programs. Furthermore, information among processes of a parallel program was not fully taken into account during generating test data. Consequently, the effectiveness and the efficiency of these methods in generating test data will be drastically reduced if they are applied directly to a parallel program. Different from their approaches, our method takes account of the features of a parallel program to generate test data.

6.2. Parallel program testing

Due to the popularity of parallel programs, seeking for approaches to effectively and efficiently testing parallel programs has attracted increasing attention of many researchers. Krammer and Resch (2006) inspected improper usages of MPI occurred while running a program. Vetter and de Supinski (2000) developed a verification tool for a MPI program, and applied it to detect certain defects such as resource exhaustion, deadlock and mismatched collective operation. Also for MPI programs, Fu et al. (2015) adopted the symbolic execution to detect the similar defects. Christakis and Sagonas (2011) collected useful information by analyzing source code and built the communication graph to detect defects related to message passing, such as deadlock and race condition. In addition, Vakkalanka et al. (2008) presented a model-checking tool for MPI programs, and Leungwattanakit et al. (2014) proposed cache-based model checking to detect deadlock and non-captured exception for distributed parallel programs. All these studies attempted to reveal problems caused by the concurrent execution, and to test communication sequences in a parallel program in order to detect certain defects such as deadlock, resource race and incorrect usage of MPI. Different from these researches, this study aims to cover paths using refined GA operators and fitness function.

Researchers have proposed various methods for finding a feasible path, for which test cases can be found to cover a given code element of a parallel program, such as a node, an edge, a statement, etc. Bao et al. (2009) employed the event graph of a program to find a feasible path. Furthermore, Katayama et al. (1997) combined the *event graph* with *interactions among units* of a concurrent program to find a feasible path of this program. In addition, Wong et al. (2005) exploited the reachability graph to generate a test sequence for covering all nodes or edges. Huang et al. (2013) developed CLAP which records thread paths at runtime and computes the related memory dependencies offline for replaying concurrency bugs. This method can be fallen into path testing techniques, but it is different from NICS-GA which evolutionarily generates test data to cover a given path by

repeatedly running a software product. Aiming to multi-threaded C programs, Rabinovitz and Grumberg (2005) detected bugs related to safety properties by confining the number of interleavings among threads for property checking. Similarly, NICS-GA also takes interleavings among processes into consideration when generating test data of MPI-BCP programs.

Many traditional coverage criteria for serial programs are not suitable or sufficient for testing message passing parallel software. Souza et al. (2008) redfined these existing criteria, such as the all-nodes, all-edges, all-defs, all-c-uses, and all-p-uses criteria, by incorporating the control/communication flows and data/message flows. They further extended the available criteria to cover types of communications, such as collective and unblocking communications (Souza et al., 2014). Unfortunately, neither the problem formulation for test data generation nor the corresponding method of generating test data has been given yet. In this paper, we improve the way of finding test inputs and execution orders of sending nodes in order to cover a given path $\mathbb{P}$, regardless of which coverage criterion is used to decide paths (such as $\mathbb{P}$) for coverage.

6.3. Applications of genetic algorithms in parallel program testing

In recent years, GAs have been brought to testing a parallel or concurrent program. Some thread interleavings can reveal concurrent faults in Java programs. In order to generate different interleavings, the synchronization primitives (e.g., yield()) can be seeded into a program to cause context switches, which is called noise injection. Eytani (2006) formulated the problem of noise injection as an optimization problem, and employed a genetic algorithm to seek for appropriate positions for injecting noises into the program. Hrubá et al. (2012) also utilized a GA to search for proper types of noises and parameterize each noise type. Alba et al. (2008) applied a GA in the model checking to detect deadlocks. Steenbuck and Fraser (2013) proposed a concurrency-based coverage to reveal real-world concurrency bugs, and utilized a genetic algorithm to generate the desired test cases for the proposed criteria.

More recently, some work has been conducted on testing BCP software by using GAs. Based on unrealistic assumptions, the problem of path coverage was described as generating test data to cover one of target paths. The method (TGA) of solving the problem by using GA was hereby proposed. CGA is a method of employing the co-evolution of two kinds of populations to generate test data that meet path coverage. This method was designed specifically for BCP software without non-determinism. Therefore, the problems formulated in these two methods (TGA and CGA) and the corresponding strategies of solving them are unsuitable for general BCP software. By contrast, this paper has proposed solutions to address these issues.

7. Conclusion

Parallel computing, along with the popularity of high-performance parallel machines and computer clusters, becomes a mainstream way for problem solving. Correspondingly, in order to guarantee the reliability of parallel programs, the testing of parallel programs has attracted more and more attention.

The problem of generating test data for covering a path of BCP software was investigated in this paper. According to the characteristics of BCP software, the software input and the execution order of sending nodes were considered together as the decision variable for this problem. In addition, the specific crossover and mutation operators were designed for the genetic algorithm to solve the above problem, with the aim of effectively and efficiently generating test data. The proposed method, namely NICS-GA, was applied to test five typical BCP software programs, and the experimental results showed that as compared with the baseline random method, the proposed NICS-GA method achieved the higher success rate with the smaller number of evaluated candidate solutions and the shorter execution time, implying the higher effectiveness and efficiency.

A number of randomly selected paths were employed to evaluate the test data generation method presented in this paper. A more comprehensive experimental evaluation can be conducted by using more appropriate methods of selecting paths. For example, path complexity can be considered when selecting paths, which is a topic to be addressed in the future. In addition, this study focused on BCP software, and simultaneously searched for the input of the program and the execution orders of sending nodes at non-deterministic points. When the unblocking communication mode is used, the non-deterministic behavior of a parallel program is related not only to senders, but also to their corresponding receivers. Under this circumstance, the problem of generating test data and the development of appropriate strategies of solving the problem to produce the desired test data would be more complicated, and thus are another potential research topics for the future work.

Acknowledgment

This work is jointly supported by National Natural Science Foundation of China with Grant Nos. 61503220 and 61773384.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.jss.2019.04.049.

References

Ahmed, M.A., Hermadi, I., 2008. GA-based multiple paths test data generator. Comput. Oper. Res. 35 (10), 3107–3124.
Alba, E., Chicano, F., Ferreira, M., Gomez-Pulido, J., 2008. Finding deadlocks in large concurrent java programs using genetic algorithms. In: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation, pp. 1735–1742.
Ali, S., Briand, L.C., Hemmati, H., Panesar-Walawege, R.K., 2010. A systematic review of the application and empirical investigation of search-based test case generation. IEEE Trans. Softw. Eng. 36 (6), 742–762.
Arcuri, A., 2010. Longer is better: on the role of test sequence length in software testing. In: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation, pp. 469–478.
Arcuri, A., Fraser, G., 2011. On parameter tuning in search based software engineering. In: Proceedings of International Symposium on Search Based Software Engineering. Springer, Berlin, Heidelberg, pp. 33–47.
Bao, X., Zhang, N., Ding, Z., 2009. Test case generation of concurrent programs based on event graph. In: Proceedings of the 5th International Joint Conference on INC, IMS and IDC, pp. 143–149.
Barker, B., 2015. Message passing interface (mpi)[c]//workshop: High performance computing on stampede 262.
Bueno, P.M.S., Jino, M., Wong, W.E., 2014. Diversity oriented test data generation using metaheuristic search techniques. Inf. Sci. 259, 490–509.
Christakis, M., Sagonas, K.F., 2011. Detection of asynchronous message passing errors using static analysis. In: Proceedings of the 13th International Symposium on Practical Aspects of Declarative Languages, pp. 5–18.
Eytani, Y., 2006. Concurrent java test generation as a search problem. Electron Notes Theor. Comput. Sci. 144 (4), 57–72.
Fu, X., Chen, Z., Zhang, Y., Huang, C., Dong, W., Wang, J., 2015. MPISE: symbolic execution of MPI programs. In: Proceedings of the 2015 IEEE 16th International Symposium on High Assurance Systems Engineering (HASE), pp. 181–188.
Garousi, V., 2008. Empirical analysis of a genetic algorithm-based stress test technique. In: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation. ACM, pp. 1743–1750.
Gong, D.W., Zhang, W., Yao, X., 2011. Evolutionary generation of test data for many paths coverage based on grouping. J. Syst. Softw. 84 (12), 2222–2233.
Harman, M., Mansouri, S.A., Zhang, Y., 2012. Search-based software engineering: trends, techniques and applications. ACM Comput Surv 45 (1). 11:1-11:61
Harman, M., McMinn, P., 2010. A theoretical and empirical study of search based testing: local, global and hybrid search. IEEE Trans. Softw. Eng. 36 (2), 226–247.
Hrubá, V., Krěna, B., Letko, Z., Ur, S., Vojnar, T., 2012. Testing of concurrent programs using genetic algorithms. In: Proceedings of the 4th International Conference on Search Based Software Engineering, pp. 152–167.
Huang, J., Zhang, C., Dolby, J., 2013. CLAP: recording local executions to reproduce concurrency failures. ACM Sigplan Not. 48 (6), 141–152.

Katayama, T., Furukawa, Z., Ushijima, K., 1997. A test-case generation method for concurrent programs including task-types. In: Proceedings of Joint 4th International Computer Science Conference and 4th Asia Pacific Software Engineering Conference, pp. 485–494.
Krammer, B., Resch, M.M., 2006. Correctness checking of MPI one-sided communication using marmot. In: Proceedings of the 13th European PVM/MPI Users Group Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface, ser. EuroPVM/MPI06, pp. 105–114.
Krawczyk, H., Wiszniewski, B., Mork, H., 1994. Classification of Software Defects in Parallel Programs. Faculty of Electronics, Technical University of Gdansk, Poland Tech. rep. 2.
Leungwattanakit, W., Artho, C., Hagiya, M., Tanabe, Y., Yamamoto, M., Takahashi, K., 2014. Modular software model checking for distributed systems. IEEE Trans. Softw. Eng. 40 (5), 483–501.
Malhotra, R., Khari, M., 2013. Heuristic search-based approach for automated test data generation. A survey. Int. J. Bio-Insp. Comput. 5 (1), 1–18.
McMinn, P., Harman, M., Binkley, D., Tonella, P., 2006. The species per path approach to search-based test data generation. In: Proceedings of the 2006 International Symposium on Software Testing and Analysis, pp. 13–24.
Offutt, A.J., Lee, A., Rothermel, G., 1996. An experimental determination of sufficient mutant operators. ACM Trans. Softw. Eng. Method. 5 (2), 99–118.
Rabinovitz, I., Grumberg, O., 2005. Bounded model checking of concurrent programs. In: Proceedings of International Conference on Computer Aided Verification. Springer, Berlin, Heidelberg, pp. 82–97.
Rao, C.P.G., Govindarajulu, P., 2015. Genetic algorithm for automatic generation of representative test suite for mutation testing. Int. J. Comput. Sci. Netw. Secur. 15 (2), 11–17.
Rashid, H., Qureshi, K., 2006. A practical performance comparison of parallel sorting algorithms on homogeneous network of workstations. WSEAS Trans. Comput. 5 (7), 1606–1610.
Rauf, A., Anwar, S., 2010. Automated GUI test coverage analysis using GA. In: Proceedings of the 7th International Conference on Information Technology, pp. 1057–1062.
Reussner, R., Sanders, P., Traff, J.L., 2002. SKaMPI: a comprehensive benchmark for public benchmarking of MPI. Sci. Program. 10 (1), 55–65.
Silva, R.A., Souza, S.R.S., Souza, P.S.L., 2012. Mutation operators for concurrent programs in MPI. In: Proceedings of the 13th Latin American Test Workshop (LATW), pp. 1–6.
Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J., 1998. MPI- the complete reference, volume 1. The MPI Core, 2nd MIT Press, Cambridge, MA, USA.
Souza, P.S.L., Souza, S.R.S., Zaluska, E., 2014. Structural testing for message-passing concurrent programs: an extended test model. Concurr. Comput. 26 (1), 21–50.
Souza, S.R.S., Vergilio, S.R., Souza, P.S.L., Simao, A.S., Hausen, A.C., 2008. Structural testing criteria for message-passing parallel programs. Concurr. Comput. 20 (16), 1893–1916.
Steenbuck, S., Fraser, G., 2013. Generating unit tests for concurrent classes. In: Proceedings of the 2013 IEEE 6th International Conference on Software Testing, Verification and Validation, pp. 144–153.
Tian, T., Gong, D.W., 2013. Model of test data generation for path coverage of messeage-passing parallel programs and its evolution-based solution. Chin. J. Comput. 36 (11), 2212–2223.
Tian, T., Gong, D.W., 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.
Vakkalanka, S., Delisi, M., Gopalakrishnan, G., Kirby, R., Thakur, R., Gropp, W., 2008. Implementing efficient dynamic formal verification methods for MPI programs. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface, pp. 248–256.
Vetter, J.S., de Supinski, B.R., 2000. Dynamic software testing of MPI applications with umpire. In: Proceedings of the 2000 ACM/IEEE conference on Supercomputing, Vol. 51. 1-51:10
Wong, W.E., Lei, Y., Ma, X., 2005. Effective generation of test sequences for structural testing of concurrent programs. In: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems, pp. 539–548.
Wu, H., Nie, C., Kuo, F.C., Leung, H., Colbourn, C.J., 2015. A discrete particle swarm optimization for covering array generation. IEEE Trans. Evol. Comput. 19 (4), 575–591.
Xanthakis, S., Ellis, C., Skourlas, C., 1992. Application of genetic algorithms to software testing. In: Proceedings of the 1992 International Conference on Software Engineering and Its Applications, pp. 625–636.

Tian Tian received the Ph.D. degree in control theory and control engineering from China University of Mining and Technology in 2014. She is an associate professor in the School of Computer Science and Technology, Shandong Jianzhu University. Her main research interest is parallel program testing.

Dunwei Gong received the Ph.D. degree in control theory and control engineering from China University of Mining and Technology in 1999. He is a professor in the School of Information and Control Engineering, China University of Mining and Technology. His main research interest is search-based software engineering.

Fei-Ching Kuo was a senior lecturer at the Department of Computer Science and Software Engineering, Swinburne University of Technology, Australia. She received her Bachelor of Science Honors in Computer Science and Ph.D. in Software Engineering, both from Swinburne University of Technology, Australia. She was a lecturer at University of Wollongong, Australia. She was also the Program Committee Chair for the 10th International Conference on Quality Software 2010 (QSIC'10) and Guest Editor of a Special Issue for the Journal of Systems and Software, special issue for Software Practice and Experience, and special issue for International Journal of Software Engineering and Knowledge Engineering. Her research interests included software analysis, testing and debugging.

Huai Liu is a lecturer in Department of Computer Science and Software Engineering at Swinburne University of Technology, Melbourne, Australia. He received his Bachelor of Engineering in Physioelectronic Technology and Masters of Engineering in Communications and Information Systems at Nankai University, China, in 1998 and 2003 respectively, and a Ph.D. degree in Software Engineering from Swinburne University of Technology, Australia, in 2008. Dr. Liu has worked as a Lecturer at Victoria University and a Research Fellow at RMIT University. Prior to working in Higher Education he worked as an engineer in the IT industry. Dr Liu's research interests include software testing and reliability, services and cloud computing, and end-user software engineering.