Information and Software Technology 80 (2016) 217–230

Contents lists available at ScienceDirect

Information and Software Technology

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

Reducing scheduling sequences of message-passing parallel programs

Dunwei Gong[a,b,*], Chen Zhang[a], Tian Tian[c], Zheng Li[d]

[a] School of Information and Electrical Engineering, China University of Mining and Technology, Xuzhou, Jiangsu, 221116, P.R. China
[b] School of Electrical Engineering and Information Engineering, LanZhou University of Technology, Lanzhou, Gansu 730000, P.R. China
[c] School of Computer Science and Technology, Shandong Jianzhu University, Jinan, Shandong 250101, P.R. China
[d] College of Information Science and Technology, Beijing University of Chemical Technology, Beijing 100029, P.R. China

ARTICLE INFO

Article history:
Received 24 November 2015
Revised 22 August 2016
Accepted 11 September 2016
Available online 15 September 2016

Keywords:
Software testing
Message-passing parallel program
Reduction of scheduling sequences
Statement coverage
Equivalent class

ABSTRACT

Context: Message-passing parallel programs are commonly used parallel programs. Various scheduling sequences contained in these programs, however, increase the difficulty of testing them. Therefore, reducing scheduling sequences by using appropriate approaches can greatly improve the efficiency of testing these programs.

Objective: This paper focuses on the problem of reducing scheduling sequences of message-passing parallel programs, and presents a novel approach to reducing scheduling sequences.

Method: In this approach, scheduling sequences that affect the target statement are first determined based on the relation between a scheduling sequence and the execution of the target statement. Then, these scheduling sequences are divided into a number of equivalent classes according to the execution of the target statement. Finally, for each scheduling sequence in the same equivalent class, the values of the two proposed indexes are calculated, and the scheduling sequence with the minimal comprehensive value is selected as the one after reduction.

Results: To evaluate the performance of the proposed approach, it is applied to test 12 typical message-passing parallel programs. The experimental results show that the proposed approach reduces 63% scheduling sequences on average. And compared with the method without reduction, and the method with randomly selecting scheduling sequences, the proposed approach shortens 67% and 52% execution time of a program for covering the target statement on average, respectively.

Conclusion: The proposed approach can greatly reduce scheduling sequences, and shorten execution time of a program for covering the target statement, hence improving the efficiency of testing the program.

1. Introduction

Testing is an important way to ensure the correctness of software, and a lot of time consumption increases the cost of software testing. Existing statistics have shown that more than 50% of the total cost in developing software is consumed on testing [1]. Along with broad applications of software testing, there are more and more intense demands for the methods of testing high-performance software [2]. Therefore, it is of considerable necessity to shorten time in testing high-performance software by using appropriate methods.

A parallel program is referred to contain two or more processes with parallel execution [3]. Parallel programs are very popular in real world applications, since most large scale science and engineering computation, such as energy exploration, medicine, and military [4], is often implemented by parallel programs. Generally, parallel programs have good portability, powerful functions, and high efficiency [5]. In addition, almost all vendors engaging in parallel computation provide support for them. Among various parallel programs, message-passing parallel programs are widely used in practice. They are written with FORTRAN, C or other languages.

Test criteria are of considerable importance in software testing, due to guiding the generation of test data, and evaluating the adequacy of testing. So far, various test criteria have been proposed, among which statement coverage is commonly used for structure coverage. Generally, if a given statement (called the target statement) can be executed with a test datum as the input of a program, the test datum is called to cover the target statement.

For serial programs, one execution is enough to judge whether a test datum covers the target statement or not. However, this is not true for message-passing parallel programs. Uncertain ex-

[*] Corresponding author.
E-mail addresses: 340355960@qq.com, dwgong@vip.163.com (D. Gong).

http://dx.doi.org/10.1016/j.infsof.2016.09.003
0950-5849/© 2016 Elsevier B.V. All rights reserved.

ecutions of message-passing parallel programs exist where different coverage results may be caused for the same test datum under different scheduling sequences. Therefore, a test datum cannot be judged to fail to cover the target statement, if the target statement is not executed with the test datum under a scheduling sequence. To this end, the coverage results of the target statement with the same test datum under other scheduling sequences should be further investigated. From this viewpoint, parallel programs consume much more execution time for statement coverage testing than their serial counterparts, indicating considerable necessity of researching methods of shortening execution time for statement coverage testing of parallel programs.

This paper focuses on the problem of reducing scheduling sequences for statement coverage of message-passing parallel programs. To this end, a scheduling sequence that affects the target statement is first defined based on the relation between the scheduling sequence and the execution of the target statement, and a method of seeking these scheduling sequences is given. Following that, the equivalent class of scheduling sequences of the target statement is defined, and a method of forming a number of equivalent classes is presented. Finally, a number of criteria are proposed to evaluate each scheduling sequence in an equivalent class, and the scheduling sequence with the minimal comprehensive value is selected as the one after reduction.

The contributions of this paper are mainly manifested in the following three aspects: (1) presenting a method of seeking scheduling sequences that affect the target statement, (2) proposing a method of forming a series of equivalent classes of scheduling sequences for the target statement, and (3) providing a number of criteria to select an appropriate scheduling sequence.

The remainder of this paper is organized as follows. Section 2 reviews related work. The proposed approach is stated in detail in Section 3 which includes determining scheduling sequences that affect the target statement, forming a series of equivalent classes of scheduling sequences, and selecting an appropriate scheduling sequence from each equivalent class. In Section 4, the applications of the proposed approach in testing several typical message-passing parallel programs and the comparative experiments are provided. Finally, Section 5 summarizes the whole paper, and points out several topics to be further studied.

2. Related work

2.1. Parallel programs testing

Since multiple processes execute simultaneously, parallel programs can make full use of all the hardware resources provided by a system, and improve the efficiency of solving a problem. Parallel programs are, however, subjected to such problems as data race, resource conflict, deadlock, and uncertain execution, to say a few, which greatly increases the difficulty in testing. Fortunately, there has been some valuable research on testing parallel programs.

Christakis et al. built the communication graph of a program by analyzing the source code of this program, and used the graph to detect such defects as deadlock and data conflict [6]. Based on the theory of semantics approximation, Miné analyzed the relations among processes of an embedded parallel program, and employed them to detect defects [7]. In the formal verification tool, TASS, developed by Siegel et al., the correctness of a program is validated by constructing the abstract model of this program, conducting the symbolic execution, and enumerating the whole state space [8]. Given the fact that all the above methods do not actually execute the program under test, they are called the static methods. Model checking is also a representative static method. When it is applied to test parallel programs, the problem of combination explosion, however, appears due to a large number of interactions between processes. To overcome the above drawbacks, Flanagan et al. proposed a method of dynamically reducing partial orders [9]. Based on this method, Vakkalanka et al. developed a model checking tool, ISP, and applied it to seek deadlock in a program [10].

Compared with the static methods, the dynamic methods actually execute a program under test. Krammer et al. checked whether the interfaces of a parallel program are correct or not by executing this program [11]. By using the defect inspection tool developed by Vetter et al., such defects as deadlock, unmatched collective operations, and resource depletion occurring when executing a program can be found [12]. For the testing tool developed by Park et al., it seeks defects in a program by inspecting the communications between processes [13]. In the reachability testing method proposed by Lei et al., each partial order synchronization sequence is executed only once, and the ones having been executed are not saved any longer [14]. Carver et al. proposed a distributed reachability testing method to improve the efficiency of testing by executing multiple test sequences simultaneously [15]. Given the fact that traditional unit testing does not take such problems as deadlock and data race into account, Shivaprasad et al. extended the existing unit testing framework to suit for parallel programs [16]. Hwang et al. obtained a number of synchronous pairs by using reachability testing, and employed them to generate test data that cover statements [17]. For distributed programs, Ferguson et al. utilized a chaining approach to generate test data for covering statements [18]. In addition, Tian et al. employed a co-evolutionary genetic algorithm to generate test data that cover paths [19].

If the static and the dynamic methods are combined together, and employed to test a program, the efficiency of testing will be further improved. Chen et al. presented a combined testing method. In this approach, some basic information of a program is first obtained by using the static analysis, and then utilized to predict the behaviors of the branches not having been covered during the execution of the program [20]. With regard to the method of unit testing for parallel programs proposed by Schimmel et al., the source codes possible to cause data race are first sought by using the static analysis, and then the execution traces of the program are obtained by employing the dynamic methods. Based on them, data race in this program are further inspected [21]. Additionally, Liao et al. proposed a synchronous communication model and its simplified version for message-passing parallel programs. These models can detect such defect as deadlock in a program before and after executing it [22].

Some scholars have proposed several testing criteria for parallel programs based on previous test criteria for serial counterparts. For shared memory parallel programs, Yang et al. expanded the coverage criteria for serial programs to those for their parallel counterparts according to the characteristics of parallel programs [23]. Further, they proposed an approach to seeking paths that satisfy all-du-path coverage, one of coverage criteria for parallel programs [24]. Souza et al. presented such criteria as all-nodes-s coverage, all-nodes-r coverage, all-nodes coverage, all-edges-s coverage, as well as all the edge coverage based on the control flow graph of a program, and all-defs coverage, all-defs-s coverage, all-c-uses coverage, all-p-uses coverage, all-s-uses coverage, all-s-c-uses coverage, as well as all-s-p-uses coverage based on the data flow graph of this program [25]. Alper et al. investigated mutating testing of parallel programs, and presented a novel criterion to judge whether a mutant is killed or not aiming to the uncertain execution of parallel programs [26].

There have been many studies on testing parallel programs. The object of most studies, however, is not message-passing parallel programs. Therefore, these studies cannot be applied directly to testing message-passing parallel programs. From this viewpoint, it is very urgent to research on effective methods for testing parallel programs according to the characteristics of these programs.

2.2. Evaluation on the complexity of a program

Evaluating the complexity of a program is very helpful to its testing. To this end, some indexes are employed when evaluation, among which the line of code [27], the number of predicates [28], NPATH complexity [29], the cyclomatic complexity [30], and Halstead complexity [31] are commonly used. Here, the line of code, just as its name, refers to the number of lines of code in a program, and is in general equal to the number of semicolons except those in comments and strings in this program. The number of predicates indicates the number of predicate expressions in conditional statements. For the cyclomatic complexity, it is related to the number of paths traversing a given code. The number of independent paths can be calculated according to the numbers of edges, nodes, and connected parts in the control flow graph of a program, which is the cyclomatic complexity. With respect to Halstead complexity, it only considers the data flow of a program. The attribute values of the program, including length, vocabulary, volume, difficulty, level, cost, time, and bugs, are calculated by the numbers of operands and operators in the program, and the complexity of the program is further evaluated based on these values.

The indexes above have been applied to a variety of software testing. According to Halstead complexity and other indexes, Tian et al. selected the optimal path from a number of paths of a message-passing parallel program [32]. Debbarma et al. chose targets to be tested based on the line of code, the number of predicates, NPATH complexity, and Halstead complexity [33]. In addition, Papadakis et al. evaluated the difficulty of covering targets according to the number of predicates [34]. It can be seen that the difficulty of covering targets can be evaluated based on existing complexity indexes. However, these indexes either are not suitable for evaluating scheduling sequences, or need a lot of changes, since existing methods are mainly for evaluating statements, branches and paths, instead of scheduling sequences focused on in this paper. It is natural that different indexes are required when evaluating different targets. What is more, all the existing methods have not considered communication cost between processes in message-passing parallel programs. Communication between processes, however, is one of very important characteristics of message-passing parallel programs. Therefore, indexes without communication cost are not comprehensive.

2.3. Communication models of parallel programs

At present, various models have been employed to describe communication of parallel programs, among which LogP [35], LogGP [36], LogGPS [37], and LoGPC [38] are commonly used, and LogP is the most basic since the others are the extended versions of LogP. For LogP, it contains four parameters, *L*, *o*, *g*, and *P*, where *L* refers to the maximal delay for passing a piece of message, and *o* is the time consumption of a processor to send or receive a piece of message. These two parameters reflect the inherent properties of parallel programs when passing a piece of messages. *g* is the shortest time interval required to consecutively send or receive messages, and reflects the capability of a processor in processing a network protocol. In addition, *P* represents the number of processors. For LogGP, it has an additional parameter, *G*, to indicate the time interval between bytes when sending a long message. With respect to LogGPS, it further considers the synchronization during passing messages based on LogGP. Finally, LoGPC pays attention to the network congestion.

In addition, BSP is also a commonly used communication model [39]. This model contains three parameters, *P*, *g*, and *L*, where *P* represents the number of processors, *g* is the bandwidth of a communication network, and *L* means the time interval between global synchronization. BSP ignores the time consumption of a processor to send or receive a piece of message, and introduces the mechanism of barrier synchronization. Therefore, it can be regarded to some extent as a simplified version of LogP. C3 is a kind of parallel models with a coarse granularity, and contains such parameters as the number of processors, the number of communication, the communication delay, the size of data for sending, and the network bandwidth. These parameters can reflect the computational complexity and communication congestion of parallel programs [40], to say a few.

However, the models above are hard to be analyzed due to their complexity. In order to reduce the difficulty of analysis, Thakur et al. proposed a simplified model for function communication in message-passing parallel programs, and divided communication time into the following two parts: one is related to data to be passed, and the other is not. They further estimated cost for function communication based on this model [41].

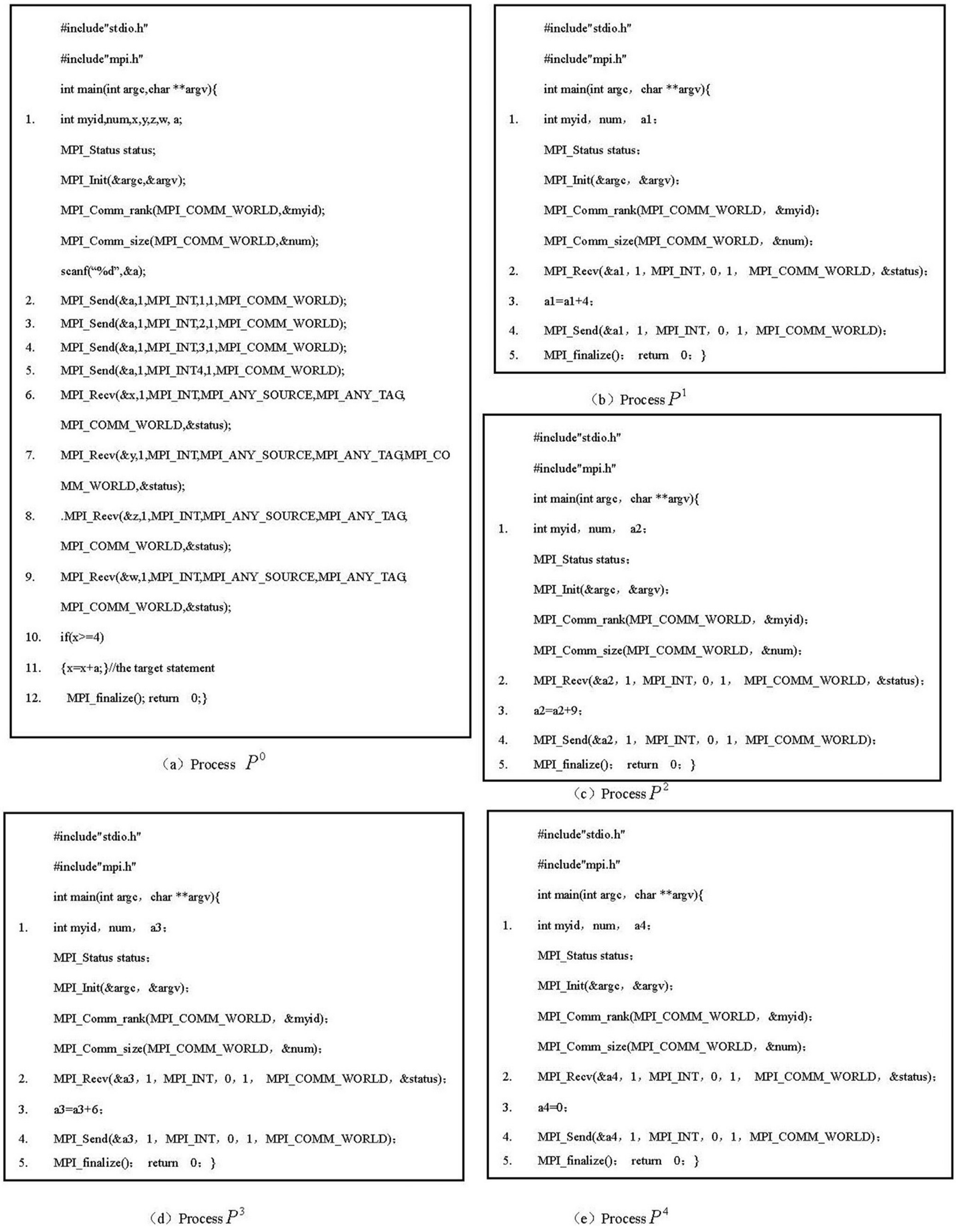
3. The proposed approach

This section presents a method of reducing scheduling sequences. Based on the existing test data, a target statement is expected to be covered according to the scheduling sequences after reduction, so that the cost of executing the program under test can be reduced, and the efficiency of testing the program can be improved. The idea of the proposed approach is as follows. Scheduling sequences that affect the target statement are first determined based on the relation between a scheduling sequence and the execution of the target statement. Then, these scheduling sequences are divided into a number of equivalent classes according to the execution of the target statement. Finally, for each scheduling sequence in the same equivalent class, the values of the two proposed indexes are calculated, and the scheduling sequence with the minimal comprehensive value is selected as the one after reduction. When the target statement is required to cover, the program is executed according to the scheduling sequences after reduction.

It can be seen that determining scheduling sequences that affect the target statement, forming a number of equivalent classes, and selecting a scheduling sequence from each equivalent class are step-by-step related, and three key techniques of this paper. In the following three sections, details of the techniques above will be given, and examples of intuitively demonstrating these details will be provided.

3.1. Determining scheduling sequences that affect the target statement

Unlike serial programs, message-passing parallel programs are featured by various communication statements, such as blocking, non-blocking, and collective communication statements. The execution of a program is uncertain due to different settings of parameters in these communication statements. Taking the message-passing parallel program shown in Fig. 1 as an example, statement 2 in Fig. 1(a) is a sending statement, and the values of its parameters, *dest, tag*, and *comm*, are 1, 1, and MPI_COMM_WORLD, respectively. These mean a message with the value of *tag* being 1 is sent to process 1 in the communication domain of MPI_COMM_WORLD. Similarly, statement 2 in Fig. 1(b) is a receiving statement, whose parameters, *source, tag*, and *comm* have the values of 0, 1, and MPI_COMM_WORLD, respectively. These mean that a message with the value of *tag* being 1 is received from process 0 in the communication domain of MPI_COMM_WORLD. Since parameters, *dest*(*source*), *tag*, and *comm*, of the two statements are matched, the latter can receive the message from the former. For statement 6 in Fig. 1(a), since the value of its parameter, *source*, is MPI_ANY_SOURCE, this statement can receive messages from other



```
        #include"stdio.h"
        #include"mpi.h"
        int main(int argc,char **argv){
1.      int myid,num,x,y,z,w, a;
        MPI_Status status;
        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
        MPI_Comm_size(MPI_COMM_WORLD,&num);
        scanf("%d",&a);
2.      MPI_Send(&a,1,MPI_INT,1,1,MPI_COMM_WORLD);
3.      MPI_Send(&a,1,MPI_INT,2,1,MPI_COMM_WORLD);
4.      MPI_Send(&a,1,MPI_INT,3,1,MPI_COMM_WORLD);
5.      MPI_Send(&a,1,MPI_INT4,1,MPI_COMM_WORLD);
6.      MPI_Recv(&x,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,
        MPI_COMM_WORLD,&status);
7.      MPI_Recv(&y,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_CO
        MM_WORLD,&status);
8.      .MPI_Recv(&z,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,
        MPI_COMM_WORLD,&status);
9.      MPI_Recv(&w,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,
        MPI_COMM_WORLD,&status);
10.     if(x>=4)
11.     {x=x+a;}//the target statement
12.      MPI_finalize(); return   0;}
```

(a) Process P^0

```
        #include"stdio.h"
        #include"mpi.h"
        int main(int argc， char **argv){
1.      int myid， num，  a1；
        MPI_Status status；
        MPI_Init(&argc， &argv)；
        MPI_Comm_rank(MPI_COMM_WORLD， &myid)；
        MPI_Comm_size(MPI_COMM_WORLD， &num)；
2.      MPI_Recv(&a1， 1， MPI_INT， 0， 1，  MPI_COMM_WORLD， &status)；
3.      a1=a1+4；
4.      MPI_Send(&a1， 1， MPI_INT， 0， 1， MPI_COMM_WORLD)；
5.      MPI_finalize()；  return   0； }
```

(b) Process P^1

```
        #include"stdio.h"
        #include"mpi.h"
        int main(int argc， char **argv){
1.      int myid， num，  a2；
        MPI_Status status；
        MPI_Init(&argc， &argv)；
        MPI_Comm_rank(MPI_COMM_WORLD， &myid)；
        MPI_Comm_size(MPI_COMM_WORLD， &num)；
2.      MPI_Recv(&a2， 1， MPI_INT， 0， 1，  MPI_COMM_WORLD， &status)；
3.      a2=a2+9；
4.      MPI_Send(&a2， 1， MPI_INT， 0， 1， MPI_COMM_WORLD)；
5.      MPI_finalize()；  return   0； }
```

(c) Process P^2

```
        #include"stdio.h"
        #include"mpi.h"
        int main(int argc， char **argv){
1.      int myid， num，  a3；
        MPI_Status status；
        MPI_Init(&argc， &argv)；
        MPI_Comm_rank(MPI_COMM_WORLD， &myid)；
        MPI_Comm_size(MPI_COMM_WORLD， &num)；
2.      MPI_Recv(&a3， 1， MPI_INT， 0， 1，  MPI_COMM_WORLD， &status)；
3.      a3=a3+6；
4.      MPI_Send(&a3， 1， MPI_INT， 0， 1， MPI_COMM_WORLD)；
5.      MPI_finalize()；  return   0； }
```

(d) Process P^3

```
        #include"stdio.h"
        #include"mpi.h"
        int main(int argc， char **argv){
1.      int myid， num，  a4；
        MPI_Status status；
        MPI_Init(&argc， &argv)；
        MPI_Comm_rank(MPI_COMM_WORLD， &myid)；
        MPI_Comm_size(MPI_COMM_WORLD， &num)；
2.      MPI_Recv(&a4， 1， MPI_INT， 0， 1，  MPI_COMM_WORLD， &status)；
3.      a4=0；
4.      MPI_Send(&a4， 1， MPI_INT， 0， 1， MPI_COMM_WORLD)；
5.      MPI_finalize()；  return   0； }
```

(e) Process P^4

Fig. 1. The example program.

Table 1
The scheduling sequences of the program in Fig. 1.

Scheduling sequences	Message match			
	P_6^0	P_7^0	P_8^0	P_9^0
ss_1	P_4^1	P_4^2	P_4^3	P_4^4
ss_2	P_4^1	P_4^2	P_4^4	P_4^3
ss_3	P_4^1	P_4^3	P_4^2	P_4^4
ss_4	P_4^1	P_4^3	P_4^4	P_4^2
ss_5	P_4^1	P_4^4	P_4^2	P_4^3
ss_6	P_4^1	P_4^4	P_4^3	P_4^2
ss_7	P_4^2	P_4^1	P_4^3	P_4^4
ss_8	P_4^2	P_4^1	P_4^4	P_4^3
ss_9	P_4^2	P_4^3	P_4^1	P_4^4
ss_{10}	P_4^2	P_4^3	P_4^4	P_4^1
ss_{11}	P_4^2	P_4^4	P_4^1	P_4^3
ss_{12}	P_4^2	P_4^4	P_4^3	P_4^1
ss_{13}	P_4^3	P_4^1	P_4^2	P_4^4
ss_{14}	P_4^3	P_4^1	P_4^4	P_4^2
ss_{15}	P_4^3	P_4^2	P_4^1	P_4^4
ss_{16}	P_4^3	P_4^2	P_4^4	P_4^1
ss_{17}	P_4^3	P_4^4	P_4^1	P_4^2
ss_{18}	P_4^3	P_4^4	P_4^2	P_4^1
ss_{19}	P_4^4	P_4^1	P_4^2	P_4^3
ss_{20}	P_4^4	P_4^1	P_4^3	P_4^2
ss_{21}	P_4^4	P_4^2	P_4^1	P_4^3
ss_{22}	P_4^4	P_4^2	P_4^3	P_4^1
ss_{23}	P_4^4	P_4^3	P_4^1	P_4^2
ss_{24}	P_4^4	P_4^3	P_4^2	P_4^1

processes besides statement 4 in Fig. 1(b). For the same reason, the value of *tag* of statement 6 in Fig. 1(a), MPI_ANY_TAG, makes that the statement can receive messages with any value of *tag*.

Given a communication statement, if the values of its parameters make its matched sending or receiving statements uncertain, this statement is called an uncertain communication statement. An uncertain communication statement may be a sending or receiving statement. It is the uncertain communication statements contained in a message-passing parallel program that make the execution of this program uncertain.

The uncertain execution of a message-passing parallel program makes different outputs of the program with the same input under different orders of executing processes. An order of executing processes is called a scheduling sequence. It is clear that a message-passing parallel program has more than one scheduling sequence. Denote a message-passing parallel program as P. It contains $t(t > 1)$ processes, where its ith $(i = 0, 1, \ldots, n-1)$ process is denoted as P^i, and contains N^i uncertain communication statements. Consider the jth uncertain communication statement in process P^i , and denote the number of matched communication statements as N_j^i. Then, at most $N_1^i \cdot N_2^i \cdot \cdots \cdot N_{N^i}^i$ scheduling sequences can be formed based on all these uncertainty communication statements in P^i. If one takes all the t processes into account, more scheduling sequences will be formed, suggesting that the number of scheduling sequences will be greatly increased as the number of processes in a parallel program increases.

Now, the relation between scheduling sequences and uncertain communication statements is explained through the program shown in Fig. 1. For convenient description, the kth statement in process P^i is denoted as P_k^i. From Fig. 1, process P^0 contains four uncertain communication statements, P_6^0, P_7^0, P_8^0, and P_9^0, and the values of their *source* and *tag* are MPI_ANY_SOURCE and MPI_ANY_TAG, respectively. So each of them can receive messages from P_4^1, P_4^2, P_4^3, and P_4^4. For each execution, it can, however, receive messages from only one of these sending statements, and the sending statement is also uncertain. Through enumeration, this program has 24 matches of uncertain communication statements, i.e., the number of scheduling sequences. Table 1 lists all the scheduling sequences of this program, where column 1 is the serial number of a scheduling sequence, row 1 represents four uncertain receiving statements in process P^0, and the rest rows mean the matches of sending statements under the corresponding scheduling sequences.

For a target statement of a program, the execution of the statement is related not only to the input of the program, but also to its scheduling sequences. Therefore, the statement can be executed under one scheduling sequence, but cannot under the other with the same input. This shows that the scheduling sequences of the program affect the execution of the target statement.

For the program shown in Fig. 1, it has only one input variable, a, and its execution process is now analyzed with the value of a being A . With this input, the values of messages sent by P_4^1, P_4^2, P_4^3, and P_4^4 are $A+4$, $A+9$, $A+6$, and 0, respectively. To execute the target statement, P_{11}^0, the value of the message, x, received by P_6^0 is required to be greater than or equal to 4, so that the value of the predicate expression of the conditional statement, P_{10}^0, is true. From Table 1, P_6^0 receives messages sent by P_4^1 under scheduling sequences $ss_1 \sim ss_6$, and $x = A+4$. This statement, however, receives messages from P_4^2 under scheduling sequences $ss_7 \sim ss_{12}$, and the value of x becomes $A+9$. For the same statement, it receives messages sent by P_4^3 under scheduling sequences $ss_{13} \sim ss_{18}$, and $x = A+6$. Under scheduling sequences $ss_{19} \sim ss_{24}$, it, however, receives messages from P_4^4, causing the value of x to be 0.

If the value of A is equal to -1, the value of x will be 8 and 5 under scheduling sequences $ss_7 \sim ss_{12}$ and $ss_{13} \sim ss_{18}$, respectively. Since the value of x is greater than 4, the target statement, P_{11}^0, can be executed under these scheduling sequences. However, x will be equal to 3 and 0 under scheduling sequences $ss_1 \sim ss_6$ and $ss_{19} \sim ss_{24}$, respectively. Due to the value of x being smaller than 4, P_{11}^0 cannot be executed under these scheduling sequences.

If at least one input of a message-passing parallel program can cover a target statement under a scheduling sequence, this scheduling sequence is said to affect the execution of the target statement. It is clear that a message-passing parallel program often contains multiple scheduling sequences, not all of them, however, affect the execution of a target statement of this program. In order to cover the target statement, seeking scheduling sequences that affect the execution of the target statement is of considerable necessity.

Consider the program shown in Fig. 1 again, and the target statement is also P_{11}^0. If one expects to execute P_{11}^0, the value of x received by P_6^0 is required to be greater than or equal to 4. It is clear that as long as the value of the input, a, is big enough, P_{11}^0 will be executed under scheduling sequences $ss_1 \sim ss_{18}$. The value of x, however, is always 0 under scheduling sequences $ss_{19} \sim ss_{24}$, so that the value of the predicate expression of the conditional statement, P_{10}^0, is false, and P_{11}^0 cannot be executed. Thus, the scheduling sequences that affect P_{11}^0 are $ss_1 \sim ss_{18}$.

For a simple program, such as the one shown in Fig. 1, it is easy to seek scheduling sequences that affect the target statement through static analysis. For a complex program, however, appropriate methods are required to seek these scheduling sequences. In this paper, an approach to combining dynamic execution with static analysis is proposed to seek them.

For an element in the input domain of a program, the program is executed with the element as its input under each scheduling sequence. If there is a scheduling sequence under which a target statement can be executed, it is the scheduling sequence that affects the target statement. After this scheduling sequence has been found, the next one is considered, and judged whether it affects the target statement or not. The process above is repeated, until all the scheduling sequences have been investigated. Following that, execute the program with another element as its input under each scheduling sequence that has not been judged to be the one that affects the target statement, and seek all the other scheduling sequences that affect the target statement. The process above is repeated, until either all the scheduling sequences have been those that affect the target statement, or the program has been executed with all the elements as its input.

The method above is a type of dynamic methods, since it seeks the desired scheduling sequences by executing the program under test. A part of scheduling sequences that affect the target statement can be found by using the dynamic method, the effectiveness of this method, however, is closely related to the input domain of a program. If an inappropriate domain is selected, finding all the desired scheduling sequences of this program will be very hard. On this circumstance, statically analyzing the relation between control statements related to the target statement and scheduling sequences is demanded, so as to determine whether a scheduling sequence affects the execution of the target statement or not. This method is called a static method.

On one hand, the dynamic method can reduce the number of scheduling sequences that have not been judged to be the ones that affect the target statement, so as to alleviate the burden caused by static analysis to some extent. On the other hand, seeking the desired scheduling sequences by using the static method can compensate the defect that the dynamic method cannot find the desired scheduling sequences resulted from improper input domains. However, the effectiveness of the static method is closely related to the complexity of a program and a testers experience. If the program under test is very complex, or the tester is lack of experience, determining the desired scheduling sequences will be very hard, and will result in the following two problems. One is that not all the scheduling sequences that affect the target statement can be sought, the other is that the sought scheduling sequences do not affect the target statement.

3.2. Forming equivalent classes of scheduling sequences

As can be seen from Section 3.1, for the same input of a program, the target statement of this program can be executed under some scheduling sequences, whereas it cannot under others.

For the program shown in Fig. 1, the scheduling sequences that affect the target statement, P_{11}^0, are $ss_1 \sim ss_{18}$. If the value of the input variable, a, is A, the value of x in the predicate expression of P_{10}^0 is $A+4$ under scheduling sequences $ss_1 \sim ss_6$. This means that the value of the predicate expression of P_{10}^0 is same for any input under these scheduling sequences. Correspondingly, P_{11}^0 has the same execution under these scheduling sequences, either be executed or not. The same is true for the other scheduling sequences, such as $ss_7 \sim ss_{12}$ and $ss_{13} \sim ss_{18}$.

Therefore, scheduling sequences that affect the target statement of a program can be divided into a number of classes according to whether the target statement being executed or not for the same input, and the ones in the same class are equivalent. The method of forming these equivalent classes is given in details as follows.

The set consisting of all the scheduling sequences of the parallel program, P, is denoted as S, and the jth statement, P_j^i, in process P^i is selected as the target statement. According to whether a scheduling sequence affecting P_j^i or not, S is divided into the following two classes: one contains scheduling sequences that affect P_j^i, denoted as S_{j1}^i, the other includes scheduling sequences that do not affect P_j^i, denoted as S_{j2}^i. So $S = S_{j1}^i \cup S_{j2}^i$. According to whether P_j^i being executed or not with the same input, S_{j1}^i is further divided into the following m classes:

$$S_{j1}^i = S_{j1}^{i1} \cup S_{j1}^{i2} \cdots \cup S_{j1}^{im}$$
$$S_{j1}^{ih} \cap S_{j1}^{il} = \emptyset, \forall h, l \in \{1, 2, \ldots, m\},\ h \neq l$$

where m is the number of equivalent classes, and S_{j1}^{ik} is the k-th equivalent class of scheduling sequence.

For the program shown in Fig. 1, scheduling sequences $ss_1 \sim ss_6$,$ss_7 \sim ss_{12}$ and $ss_{13} \sim ss_{18}$ form different equivalent classes of the target statement, P_{11}^0. Therefore, scheduling sequences that affect P_{11}^0 can be divided into three equivalent classes.

The approach to seeking equivalent classes of scheduling sequences will be given as follows. When the target statement is in a branch or loop structure, there must be a statement on which the target statement is directly control dependent [42]. To explain the meaning of direct control dependence, assume that s_1 and s_2 are two statements in a program. If the execution of s_2 depends on s_1, s_2 is called control dependent on s_1. Further, if there is no statement, s_3, such that it is located in the path from s_1 to s_2, and s_2 is control dependent on s_3, s_2 is said to be directly dependent on s_1. It is clear that s_1 must be a conditional statement. The statement on which the target one is directly control dependent is called a **target dependence statement**.

A scheduling sequence of a message-passing parallel program can be reflected by its uncertain communication statements and their matched ones. Therefore, scheduling sequences with the same uncertain communication statements and their matched ones of a target dependence statement are equivalent.

In this paper, the relations between uncertain communication statements and their matched ones of a target dependence statement are reflected by the program dependence graph (PDG). Intuitively, PDG is a directed graph, denoted as $G = (N, E)$, where N is the set of vertices, with each vertex corresponding to one or more statements of the program; E is the set of directed edges, with each edge reflecting the dependence relation between statements. The relation can be either control, or data, or communication dependence [43]. Since different kinds of dependence relations are treated in the same way in this paper, they are not further distinguished. The PDG of a program can be obtained based on these dependence relations between statements.

For the program shown in Fig. 1, the PDG of the program under scheduling sequence ss_1 is shown in Fig. 2.

According to the PDG under a scheduling sequence, starting from the vertex corresponding to the target dependence statement, seeking vertices corresponding to all uncertain communication statements and their matched ones along the directed edges in the opposite direction, so as to obtain a vertex sequence. For different scheduling sequences, if the vertex sequences are same, the corresponding scheduling sequences are equivalent. Through the approach above, one can obtain a number of equivalent scheduling sequences.

For the program shown in Fig. 1, the PDG under scheduling sequence ss_2 is demonstrated in Fig. 3. Figs. 2 and 3 show that under these two scheduling sequences, the target dependence statement of P_{11}^0 is P_{10}^0, and the vertex sequences that affect P_{10}^0 are both $\{ P_4^1, P_6^0 \}$. So the two scheduling sequences are equivalent.

If a target statement is in the sequence structure, the execution of this statement is in general not controlled by other statements, indicating that it can be executed under any scheduling sequence that affects it. Therefore, this statement has only one equivalent class of scheduling sequences which contains all the scheduling sequences that affect it.

3.3. Selecting a scheduling sequence from each equivalent class

The target statement can be executed under any scheduling sequence in an equivalent class. Therefore, it is enough to select only one scheduling sequence from each equivalent class and test the program under this scheduling sequence. For equivalent scheduling sequences, the target statements executed under them are same, the execution processes of the program, however, are different.

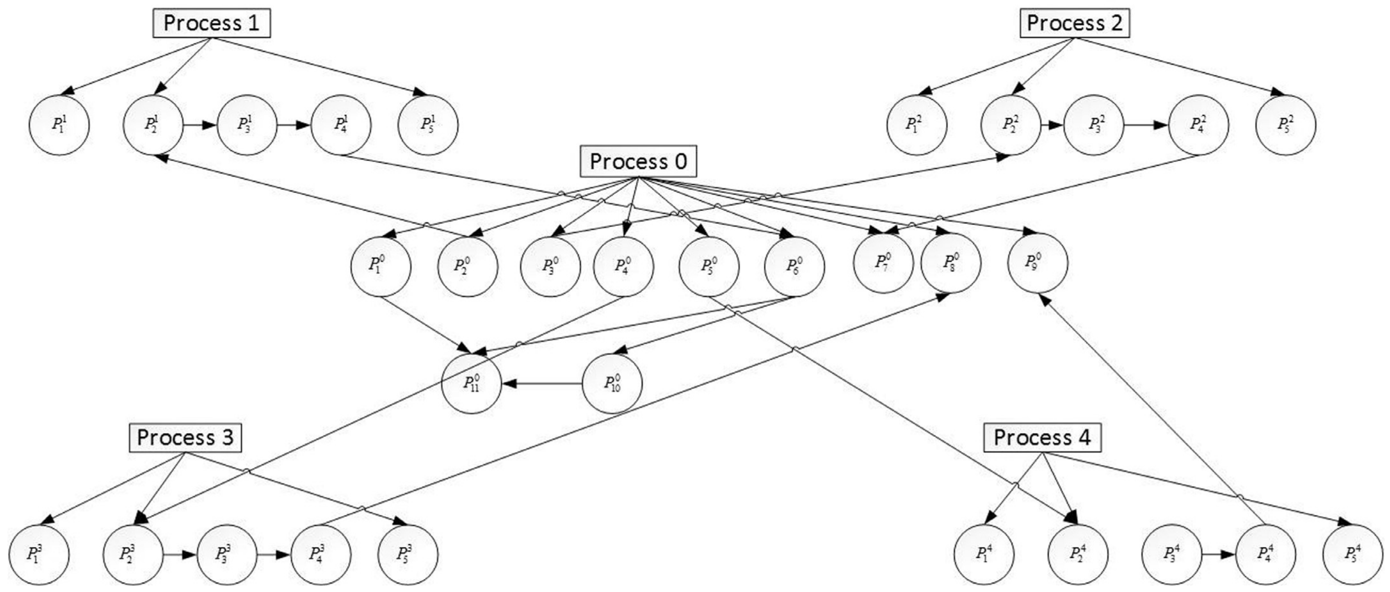


Fig. 2. The PDG of the program above under scheduling sequence ss_1.

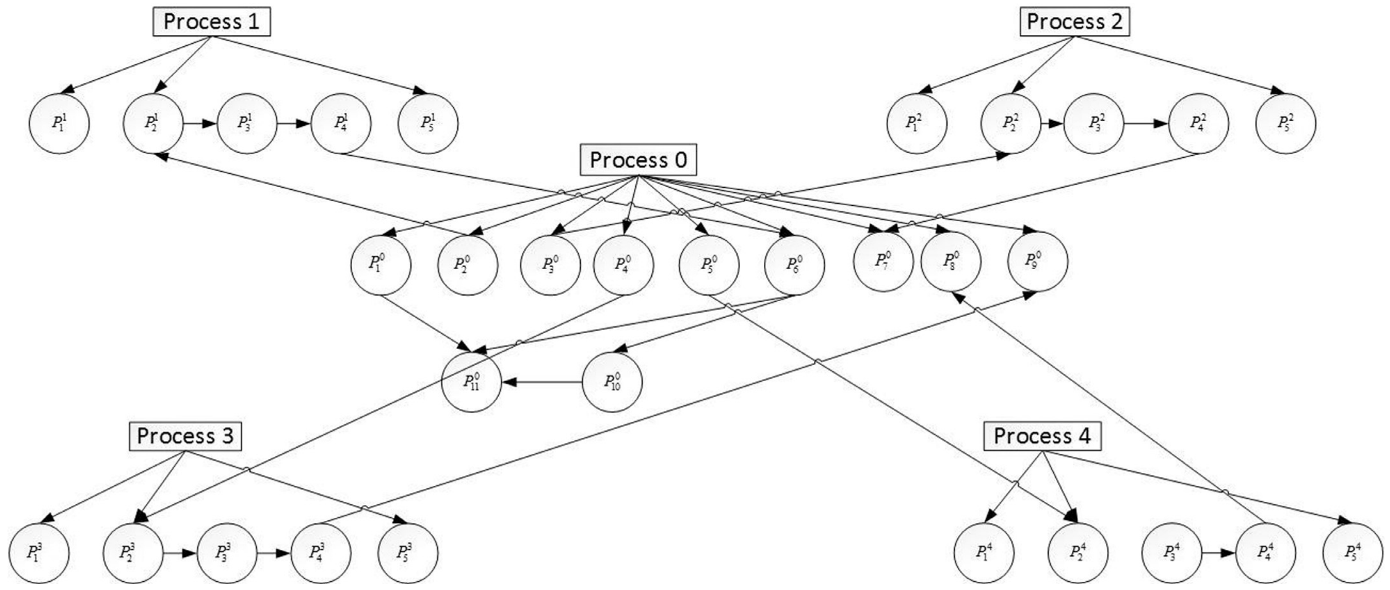


Fig. 3. The PDG of the program above under scheduling sequence ss_2.

On this circumstance, selecting an appropriate scheduling sequence from an equivalent class is very beneficial to increase the efficiency of testing. To this end, this subsection will give a method of selecting a scheduling sequence from an equivalent class.

When selecting a scheduling sequence, execution cost of a message-passing parallel program is investigated. Execution cost can be in general divided into the following two parts: computation and communication cost. Among them, computation cost depends on the line of code and the computation complexity; communication cost is closely related to the frequency of communication between processes and the amount of data passed.

In order to select an appropriate scheduling sequence, an index system is built in this subsection to evaluate execution cost of the program under different scheduling sequences. The index system includes the following two indexes: Halstead measure and average communication cost.

Index 1: Halstead measure of a scheduling sequence

In a message-passing parallel program, statements that have the sequential execution relation may exist either in the same process or in different ones. From the starting statement of the program to the target one, a series of statements that have the sequential execution relation form a statement sequence. Since a message-passing parallel program contains more than one process, a path of the program often contains a number of statement sequences. It is clear that different statement sequences have different complexities, and the statement sequence with the largest complexity often determines cost of executing the path. Therefore, this statement sequence is called a key statement sequence. To reflect the complexity of a statement sequence, Halstead complexity is employed here. It evaluates the complexity of a statement sequence by the numbers of operators and operands, where an operand refers to a variable or constant contained in the statement sequence, and an operator mainly means an arithmetic or Boolean operator.

Denote a statement sequence as *Sq*, its numbers of operators and operands as $N_1(Sq)$ and $N_2(Sq)$, respectively. According to the method in [31], the Halstead length of *Sq*, denoted as $L(Sq)$, can be represented as:

$$L(Sq) = N_1(Sq) + N_2(Sq)$$

For each statement sequence in a path, its Halstead length is calculated. Then, the statement sequence with the largest Halstead length is key. A scheduling sequence may involve more than one path. On this circumstance, the key statement sequence of each path is sought, and the largest Halstead length of these key statement sequences is regarded as Halstead measure of the scheduling sequence.

The program shown in Fig. 1 contains several statement sequences with the largest Halstead length for the target statement, P_{11}^0, under scheduling sequence ss_1, among which one is $P_1^0P_2^0P_2^1P_3^1P_4^1P_6^0P_7^0P_8^0P_9^0P_{10}^0$, and its Halstead length is 6. So Halstead measure of scheduling sequence ss_1 is 6.

Index 2: Average communication cost

For a message-passing parallel program, the time consumption for passing messages between processes is much more than that for executing statements in a process [44]. A scheduling sequence has an influence on passing of messages between processes.Therefore, the goodness of a scheduling sequence can be reflected by its communication cost between processes.

The communication mode of message-passing parallel programs can be divided into the following two categories: point-to-point communication and collective communication. When passing a message with n bytes between processes through point-to-point communication, if the transmission delay is σ, and it takes θ time units to pass each byte, $\sigma + n\theta$ time units will be consumed to execute the communication operation. Given the fact that collective communication is implemented based on point-to-point communication, Thakur et al. estimated communication cost of different collective communication modes through a number of models based on the time consumption of point-to-point communication.

The models built by Thakur et al. describe the time consumption for executing a communication operation. In message passing parallel programs, the implementation of a communication operation requires the cooperation of communication functions in two or more processes. To avoid repeatedly calculating communication cost, communication cost of each function is stipulated as follows. If a sending function and its matched receiving function cooperate to implement a point-to-point communication operation, and the operation consumes $\sigma + n\theta$ time units, communication cost of the sending or receiving function will be equal to $(\sigma + n\theta)/2$ time units. For the case that communication functions in multiple processes cooperate to implement a collective communication operation, communication cost of each function is the ratio of collective communication cost to the number of processes.

Different inputs of a program may execute different communication functions under the same scheduling sequence. Correspondingly, communication cost of this scheduling sequence is not always same. Given this fact, communication cost of a scheduling sequence is reflected by average communication cost for executing the target statement under the scheduling sequence.

In the following, average communication cost of a function is calculated according to the type of a structure that contains the function.

If a communication function is in the sequential structure of a process, it must be executed and produces communication cost once the process is executed. If the function is in the branch structure of a process, executing the process cannot guarantee the execution of the function. Given the fact that the execution of a communication function is random, probability can be employed to describe the possibility of executing the function. In fact, the execution of a communication function is related closely to the value of the predicate expression of the conditional statement that controls its execution. The probability of the value of the predicate expression being true is denoted as p. When the range of the input of a program and that of making the value of the predicate expression true are known, a method of calculating p was given in [45]. Based on the value of p, if communication cost of a function is ε time units, average communication cost of executing the function will be $p \cdot \varepsilon$ time units. If the function is in the loop structure of a process, and the number of loops is assumed as n , average communication cost of executing the function will be $p \cdot n \cdot \varepsilon$.

Assume that N_1, N_2 and N_3 communication functions under a scheduling sequence are in the sequential structure, branch structure, and loop structure, respectively, average communication cost of the ith function in the sequential structure is α_i time units, that of the jth function in the branch structure is β_j time units, and that of the kth function in the loop structure is γ_k time units, then average communication cost of the scheduling sequence is equal to $\sum_{i=1}^{N_1}\alpha_i + \sum_{j=1}^{N_2}\beta_j + \sum_{k=1}^{N_3}\gamma_k$.

For the program shown in Fig. 1, 8 sending statements and their matched receiving ones are required to execute so as to reach the target statement, P_{11}^0, under scheduling sequence ss_1. In other words, 8 point-to-point communication operations should be executed. Meanwhile, since an integer with four bytes is passed in each communication, average communication cost of ss_1 is $8\sigma + 32\theta$.

For a scheduling sequence, its Halstead measure and average communication cost can be calculated according to the methods above. Following that, a comprehensive index value of the scheduling sequence can be obtained through weighted summing the normalized values of the two indexes. Here, the min-max normalization method is employed to map the value, *ix*, of index *IX* into another value, *ix′*, in the range of [0,1] by the following formulation [46], $ix' = (ix - minIX)/(maxIX - minIX)$, where *minIX* and *maxIX* are the minimal and the maximal values of index *IX*, respectively. Both indexes proposed here have an important influence on the execution of the target statement. Which index has a more important influence, however, are related closely to the implementation environment of the program. For different implementation environments of a message-passing parallel program, these two indexes should have different weights.

The approach of reducing scheduling sequences is divided into the following three steps: determining scheduling sequences that affect the target statement, forming equivalent classes of scheduling sequences, and selecting a scheduling sequence from each equivalent class. Each step spends some time in analyzing the program under test. Additionally, if one or several scheduling sequences that affect the target statement cannot be sought, and the existing test data cover the target statement just under these scheduling sequences, the loss of the statement coverage rate will be occurred. We will see it in the next section.

4. Experiments

In this section, the proposed method of reducing scheduling sequences is applied to test several benchmark programs, and its performance is evaluated through a series of experiments. To this end, problems to answer in the experiments are first raised. Following that, basic information of the benchmark programs and the experimental environment are described. Finally, the experimental results are fully demonstrated and deeply analyzed.

Table 2
Basic information of programs under test.

Programs under test	# of input variables	# of processes	# of communication statements
Convex_quadrilateral	4	4	18
Match	5	5	16
Server_client	3	4	6
Server_client_communication	3	4	10
Matrix	16	5	24
Deposit_withdraw	3	4	6
Including	2	6	12
Creator_consumer	2	7	18
Date_swich	6	5	16
Min	125	6	20
Angle	8	12	30
Psd	35	76	290

4.1. Questions to answer

The performance of the proposed method is evaluated by answering the following 3 questions:

Q1: Can scheduling sequences that affect a target statement be found and their equivalent class(es) formed by using the proposed method?

Q2: Can the efficiency of covering a target statement be improved by reducing the scheduling sequences using the proposed method? And, will reducing scheduling sequences result in losses with respect to test thoroughness?

Q3: Can execution cost of a program be reduced by selecting a scheduling sequence from each equivalent class based on the index system proposed in this paper?

To answer these three questions, first of all, randomly select a number of statements from each program under test as the targets, and investigate whether scheduling sequences that affect each target statement can be found and their equivalent classes formed or not by using the proposed method. Then, the scheduling sequences that affect each target statement are reduced by using the proposed method, and the efficiency of covering each target statement is demonstrated by comparing the time consumption for covering the target statement with the same test data before and after reducing scheduling sequences.The corresponding statement coverage rates will also be calculated and compared to reflect that whether reducing scheduling sequences result in losses with respect to test thoroughness. Finally, select a scheduling sequence from each equivalent class by using the proposed index system and the random method, respectively. The time consumption for covering each target statement with the same test data is employed to validate whether the proposed method can reduce execution cost of the program or not.

4.2. The programs under test and the experimental environment

Twelve benchmark programs are selected as the ones under test, and their basic information is listed in Table 2. Among these programs, *Convex_quadrilateral* seeks the smallest three from four angles, and judges whether they can constitute a convex quadrilateral or not. *Match* has the function of matching strings. For *Server_client*, it has the master-slave structure, and its main process deals with messages from its slave processes. With respect to *Server_client_communication*, it is obtained by modifying *Server_client*. For the program, *Matrix*, it has the function of matrix multiplication, and judges the relation between elements of a matrix. Regarding *Deposit_withdraw*, it is a program with the function of depositing and withdrawing. For *Including*, it is from [47] and used to determine the location relation between a point and a polygon. In addition, *Creator_consumer* simulates production and consumption, *Date_swich* inspects information related to an input date, and *Min* seeks the minimal one among several numbers. *Angle* and *Psb* are industrial programs. They are used to recognize graphics and simulate special problems respectively.

The experimental environment is configured as follows. The hardware part includes Intel Core i5 CPU, 500G hard disk, and 4G memory. The software part contains Windows 8.1 operation system, Visual Studio 2013 compiler, and MPI application software, MPICH.

4.3. Experimental results and analysis

(1) Regarding Q1

To answer Q1, all the branches of each program are investigated, and a statement is randomly selected from each branch as the target one. The target statement set of the program is further formed by gathering all the target statements. For each target statement in this set, the method presented in Section 3.1 is employed to determine scheduling sequences that affect it, and the method proposed in Section 3.2 is utilized to form a number of equivalent classes of these scheduling sequences.

Taking *Convex_quadrilateral* as an example, this program has 6 scheduling sequences and 11 target statements. For each target statement, the number of equivalent classes and the reduction rate of scheduling sequences obtained by the proposed approach are listed in Table 3. In this table, column 1 is the serial number of a target statement. Column 2 refers to the number of scheduling sequences that affect the target statement, and all of them are 6. Column 3 means the number of equivalent classes. The last column provides the reduction rate of scheduling sequences. Since only one scheduling sequence is selected from each equivalent class, the number of scheduling sequences after reduction is equal to that of equivalent classes. Consequently, the reduction rate is equal to the ratio of the difference between the number of scheduling sequences that affect the target statement and that of equivalent classes to the former. In addition, the last row also lists the average value of each of columns 2–4.

Table 3 reports that for the program, *Convex_quadrilateral*, (1) different target statements have different reduction rates of scheduling sequences. Among them, there are two statements having the lowest reduction rate, 0. They are target statements 4 and 8. This shows that for these two statements, all the scheduling sequences affect them, and any pair of scheduling sequences is not equivalent. In contrast, there are three target statements (9, 10, and 11) that have the highest reduction rate, 83%. This shows that all of the 6 scheduling sequences that affect them belong to the same equivalent class. (2) The average reduction rate of scheduling sequences for all target statements is 50%. Therefore, the approach proposed in this paper can greatly reduce scheduling sequences of *Convex_quadrilateral*.

Table 3
The reduction results of scheduling sequences of *Convex_quadrilateral*.

No. of a target statement	# of scheduling sequences that affect the target statement	# of equivalent classes	Reduction rate(%)
1	6	3	50
2	6	3	50
3	6	3	50
4	6	6	0
5	6	3	50
6	6	3	50
7	6	3	50
8	6	6	0
9	6	1	83
10	6	1	83
11	6	1	83
Average	6	3	50

Table 4
The reduction results of scheduling sequences of programs under test.

The programs under test	# of scheduling sequences	# of target statements	# of scheduling sequences that affect target statements	# of equivalent classes	Reduction rate(%)
Convex_quadrilateral	6	11	6	3	50
Match	24	8	24	2.5	90
Server_client	6	12	2	1	50
Server_client_communication	6	12	2	1	50
Matrix	24	30	24	4.8	80
Deposit_withdraw	6	4	6	3	50
Including	24	10	24	8	67
Creator_consumer	12	11	12	4.7	61
Date_swich	6	12	6	3	50
Min	120	124	120	3.5	97
Angle	5040	267	5040	1684	67
Psd	120	256	120	62	48
Average	–	–	448.8	148.4	63

For the other programs, the average reduction results of scheduling sequences that affect target statements are listed in Table 4. In this table, column 1 is the programs under test. Column 2 refers to the number of scheduling sequences contained in a program. Column 3 represents the number of target statements. Column 4 means the number of scheduling sequences that affect target statements. Column 5 provides the number of equivalent classes. The last column calculates the reduction rate of scheduling sequences. In addition, this table also lists the reduction results of *Convex_quadrilateral* for convenient comparison.

Table 4 tells that (1) Except *Server_client* and *Server_client_communication* whose a few scheduling sequences do not affect the target statements, any scheduling sequence of the other programs affects the target statements. (2) Different programs have different reduction rates of scheduling sequences. In these programs, the program with the highest reduction rate, as high as 97%, is *Min*. The reason is that for most target statements of this program, scheduling sequences that affect each of these statements can form only one equivalent class. For all the scheduling sequences that affect the target statements, the average number of equivalent classes is only 3.5, reducing a large number of scheduling sequences. (3) For all the programs, the average reduction rate of scheduling sequences is as high as 63%.

The experimental results above show that the approach proposed in this paper can find scheduling sequences that affect the target statement, and form equivalent class(es) of scheduling sequences, thus greatly reducing scheduling sequences under which a program is executed.

(2) Regarding Q2

To answer Q2, first of all, an appropriate method is employed to generate a test data set that can cover the target statements. Here, the method of generating the test data set is as follows. The random method is utilized to generate a test datum. Following that, the program is executed with the test datum under each scheduling sequence, and the executed statements are recorded. If a test datum does not execute statements to be covered, it is redundant, and not put into the test data set. The process above is repeated, until either all the target statements have been covered, or the maximal number of runs has been reached. If there are uncovered target statements, the static analysis is adopted to manually generate test data. The process of generating test data suggests that the test data set contains at least one test datum that can cover each target statement.

Then, for each target statement of a program under test, the time consumption for covering the target statement with the generated test data set is compared between before and after reducing scheduling sequences, so as to demonstrate the efficiency of the proposed approach in covering the target statement. In this group of experiments, the scheduling sequence after reduction is randomly selected from each equivalent class.

Before reducing scheduling sequences, a program under test is executed with a test datum under each scheduling sequence. After reduction, the program is executed with the same test datum under each scheduling sequence after reduction. To eliminate the influence of the experimental environment on the execution of a program, each experiment is repeatedly done 50 times, and the average value of these experimental results is calculated. In each experiment, the scheduling sequences are randomly selected from their equivalent classes, indicating that the scheduling sequences after reduction selected for two experiments may not be same.

The execution time for covering the target statements of *Convex_quadrilateral* is listed in Table 5. In this table, column 1 is the serial number of a target statement. Column 2 represents the time consumption for covering target statements before reducing

Table 5
The time consumption for covering the target statements of *Convex_quadrilateral*.

No. of a target statement	Before reduction(ms)	After reduction(ms)	Time reduction rate (%)
1	0.048	0.053	-10
2	0.056	0.056	0
3	0.394	0.226	43
4	2.475	2.475	0
5	2.818	1.433	49
6	1.435	0.739	49
7	1.73	0.894	48
8	2.075	2.075	0
9	0.594	0.127	79
10	0.266	0.079	70
11	0.032	0.009	72
Average	1.084	0.742	32

Table 6
The time consumption for covering the target statements of the programs under test.

The programs under test	Before reduction(ms)	After reduction(ms)	Time reduction rate (%)
Convex_quadrilateral	1.084	0.742	32
Match	2.227	0.31	86
Server_client	0.119	0.027	77
Server_client_communication	0.149	0.049	67
Matrix	1.42	0.255	82
Deposit_withdraw	0.362	0.186	49
Including	0.738	0.16	78
Creator_consumer	0.862	0.229	73
Date_swich	1.03	0.445	57
Min	5.353	0.136	98
Angle	38.583	15.642	59
Psd	612.821	306.868	48
Average	55.396	27.087	67

scheduling sequences. Column 3 means the time consumption for covering the same target statements after reduction. The last column provides the time reduction rate which is the ratio of the reduced time consumption after reducing scheduling sequences to the time consumption before reduction. In addition, the last row also lists the average value of each of columns 2–4.

Table 5 shows that (1) for target statements 2, 4, and 8, the time reduction rate is 0, suggesting that execution time of this program is not shortened by reducing scheduling sequences. There are two possible reasons. One is that scheduling sequences that affect the target statements are not reduced by using the proposed approach; the other is that this program is executed in a special order of scheduling sequences after reduction, causing the target statements to be covered early, thus the advantage of reducing scheduling sequences cannot been fully demonstrated. (2) The time reduction rate of target statement 1 is negative, indicating that execution time of this program is not shortened, but increased by reducing scheduling sequences. This may be due to the uncertain execution of the program. (3) Among these 11 target statements, target statement 9 has the maximal time reduction rate, 79%, which is equivalent to saving four-fifths of the time consumption. (4) For all target statements, the average time reduction rate is 32% after reducing scheduling sequences, meaning that reducing scheduling sequences is helpful to shorten execution time for covering the target statements of *Convex_quadrilateral*.

For the other programs, the average time consumption for covering the target statements is listed in Table 6. In this table, except column 1 that represents the programs under test, columns 2–4 have the same meaning as those in Table 5. In addition, this table also lists the time consumption of *Convex_quadrilateral* for convenient comparison.

Table 6 demonstrates that (1) the time reduction rates of different programs are different. Among these programs, *Min* has the maximal time reduction rate, 98%. The reason is that for most target statements of this program, only one equivalent class of scheduling sequences is formed, which makes the program executed with a test datum under 120 scheduling sequences before reduction; whereas it is executed with the same test datum under only 1 scheduling sequence after reduction. The program with the minimal time reduction rate, 32%, is *Convex_quadrilateral*. (2) For all these 12 programs, the average time reduction rate is 67%, meaning that more than half of execution time of a program for covering target statements can be shortened by reducing scheduling sequences using the proposed approach.

The experimental results above show that the proposed approach of reducing scheduling sequences can greatly shorten execution time of a program for covering the target statement, hence improving the efficiency of testing the program.

To illustrate whether reducing scheduling sequences results in losses with respect to test thoroughness or not, the statement coverage rates before and after reducing scheduling sequences are also calculated through experiments. The statement coverage rate is the ratio of the number of target statements being covered to that of all the target statements. For the programs under test, their statement coverage rates before and after reducing scheduling sequences are listed in Table 7.

Table 7 demonstrates that the statement coverage rates before and after reducing scheduling sequences are all 1 but program *Angle*. This means that reducing scheduling sequences slightly reduces the statement coverage rate of program *Angle*, and has a faint negative influence on its test thoroughness. The possible reason is that not all the scheduling sequences that affect the target statement can be sought, and the generated test data cover the target statement just under the scheduling sequences not being sought. As a result, several target statements are not covered after reducing scheduling sequences. But, as can be seen from Table 7, reducing scheduling sequences does not have any negative influence on most programs test thoroughness, and program *Angle*

Table 7
The statement coverage rate of the programs under test.

The programs under test	Before reduction	After reduction
Convex_quadrilateral	1	1
Match	1	1
Server_client	1	1
Server_client_communication	1	1
Matrix	1	1
Deposit_withdraw	1	1
Including	1	1
Creator_consumer	1	1
Date_swich	1	1
Min	1	1
Angle	1	0.993
Psd	1	1

Table 8
Communication time for passing a piece of message.

The size of a message(Kb)	Communication time(μs)
1	4.87
32	23.07
64	33.16
128	59.79
256	93.82

Table 9
The value of each index and the comprehensive value of a scheduling sequence of *Including*.

No. of a scheduling sequence	Halstead measure	Average communication cost(μs)	The comprehensive value
1	46	40.8	0
2	46	61.2	0.8
3	78	61.2	1

Table 10
The time consumption for covering the target statement under a scheduling sequence selected by different approaches (for *Including*).

No. of a target statement	Random selection (μs)	The proposed selecting approach (μs)	The time reduction rate (%)
1	42	7	83
2	278	59	79
3	16	9	44
4	278	54	81
5	46	7	85
6	267	53	80
Average	154.5	31.5	80

statement coverage rate only has a bit less of 0.007. This problem can be solved by either generating additional test data, or increasing the accuracy in seeking scheduling sequences. We can conclude that reducing scheduling sequences basically has no negative influence on the test thoroughness.

(3) Regarding Q3

To answer Q3, the values of indexes of each scheduling sequence that affects the target statement is first calculated. Two indexes are considered in this paper, among which the second is average communication cost.

When calculating average communication cost of a scheduling sequence, the values of parameters σ and θ are required to known. Here, the method of *Ping-Pong* is employed to get them with its idea being as follows [48]. Process *i* first sends a message with *n* bytes to process *j*, and waits for another message sent by process *j*. Meanwhile, process *j* executes a receiving statement, and once this process receives the message sent by process *i*, it will return another message to process *i*. The process above is repeated several times, and the time consumption for passing a piece of message is calculated for each time, then its average time consumption is calculated. The time consumption for each communication is equal to a half of the average above. The time consumption for passing a piece of message with different lengths can be obtained by changing the number of bytes contained in this message.

Communication time for passing a piece of message is listed in Table 8 based on the experimental environment provided in this paper. In this table, column 1 represents the number of bytes contained in a piece of message. Column 2 is communication time for passing a piece of message. Based on these data, the values of σ and θ can be obtained by the first-order linear fitting, i.e.,$\sigma = 10.2, \theta = 3.32 \times 10^{-4}$. That is to say, the delay of point-to-point communication is 10.2 μs, and the time consumption for passing a byte is 3.32×10^{-4} μs.

Taking the first equivalent class of target statement 1 of program *Including* as an example. The value of each index and the comprehensive value of each scheduling sequence in this equivalent class are listed in Table 9. When calculating the comprehensive value of a scheduling sequence, the weights of Halstead measure and average communication cost are set to 0.2 and 0.8, respectively, based on the experimental environment of this paper.

The less the comprehensive value of a scheduling sequence, the better the scheduling sequence. So, the first scheduling sequence is selected from this equivalent class. For the other equivalent classes of target statement 1, the comprehensive value of each scheduling sequence is also calculated, and the scheduling sequence with the minimal value is selected. Further, for the other target statements of the program, the same method is employed to select scheduling sequences.

To evaluate the rationality of indexes proposed in this paper, a scheduling sequence is selected from each equivalent class that affects the target statement by using the approach proposed in this paper, and the time consumption for covering the target statement under the selected scheduling sequence is calculated, and compared with that under a scheduling sequence randomly selected from the equivalent class. For each target statement, the program is repeatedly executed 50 times, the time consumption is recorded, and the average is then calculated.

Taking program *Including* as an example, the time consumption for covering the target statement under a scheduling sequence selected by different methods is listed in Table 10. In this table, column 1 is the serial number of a target statement. Column 2 means the time consumption when randomly selecting a scheduling sequence from each equivalent class. Column 3 refers to the time consumption when selecting a scheduling sequence from each equivalent class by using the approach proposed in this paper. The last column is the time reduction rate, whose calculation method has been provided before.

Table 10 tells that for program *Including*, (1) the one with the maximal reduction rate of 85% is target statement 5, indicating that more than four-fifths of the time consumption can be saved under the scheduling sequence selected from each equivalent class by using the approach proposed in this paper. (2) Target statement 3 has the minimal time reduction rate. Even so, its value is also close to 50%. Therefore, execution time of this program can be shortened nearly half. (3) For all these target statements, the average time reduction rate is 80%, suggesting that execution time can be greatly saved by executing the program under the scheduling sequences selected by the proposed approach.

It is worth noting that the number of target statements when answering Q3 is less than that when answering Q2. The reason is explained as follows. When evaluating a scheduling sequence,

Table 11
The time consumption for covering the target statements under the scheduling sequences selected by different approaches (for all programs).

The programs under test	# of target statements	Random selection (μs)	The proposed selecting approach (μs)	The time reduction rate (%)
Server_client	4	25.5	19.5	24
Server_client_communication	4	39	7.5	81
Deposit_withdraw	3	743	439	41
Including	6	927	189	80
Creator_consumer	5	689	193	72
Date_swich	6	2167	1619	25
Angle	217	151	105	30
Psd	186	64,779	52,106	20
Average	–	8690	6834	52

some scheduling sequences in the same equivalent class have the same comprehensive value. On this circumstance, the proposed approach to selecting scheduling sequences will degenerate to randomly select, and have the same selection results as the random selection method. So only the target statements for which the proposed approach is suitable are listed in Table 10. The same is true for the target statements in Table 11.

For the other programs, the average time consumption for covering the target statements under the scheduling sequences selected by different approaches is listed in Table 11. In this table, except that column 1 represents the programs under test, and column 2 refers to the number of target statements, the other columns have the same meaning as those in Table 10. In addition, this table also lists the time consumption of program *Including* for convenient comparison.

Table 11 shows that (1) different programs have different time reduction rates when selecting scheduling sequences by using the proposed method. Among these programs, *Server_client_communication* has the largest time reduction rate of 81%, indicating that more than three-fourths of execution time of this program can be shortened. The program with the minimal reduction rate is *Psd*. Even so, one-fifth of execution time of this program is saved. (2) For all the programs listed in this table, the average time reduction rate is 46%, suggesting that the proposed method can save nearly half of execute time of the random selection method.

The experimental results above show that selecting a scheduling sequence from each equivalent class by using the proposed index system can greatly reduce execution cost of a program.

By answering the three questions raised in Section 4.1, the following conclusion can be drawn: the proposed approach to reducing scheduling sequences can greatly shorten execution time of a program for covering the target statement, hence improving the efficiency of testing the program.

5. Conclusions

This paper focuses on the problem of testing message-passing parallel programs, a kind of widely used parallel programs. The efficiency of testing message-passing parallel programs is expected to increase by reducing scheduling sequences. The approach presented in this paper can be divided into the following three steps. Scheduling sequences that affect the target statement are first determined. Following that, these scheduling sequences are divided into a number of equivalent classes. Finally, a scheduling sequence is selected from each equivalent class. The efficiency of testing a parallel program can be greatly improved when executing the program under the scheduling sequences after reduction.

To evaluate the performance of the proposed approach, it is applied to test 12 typical message-passing parallel programs, and compared with other approaches. The experimental results show that the proposed approach can greatly shorten execution time of a program for covering the target statement, hence improving the efficiency of testing the program.

It is worth mentioning that the approach to reducing scheduling sequences presented in this paper considers only one target statement. In practice, a program, especially a complex program, often has a lot of statements required to test. If multiple target statements are simultaneously considered, scheduling sequences that affect these target statements are different from those when considering only one target statement, and are not a simple union of scheduling sequences that affect each target statement. On this circumstance, determining scheduling sequences that affect multiple target statements, and further classifying them so as to reduce scheduling sequences is a very challenging topic. In addition, if proper approaches are utilized to reflect the complexity of an equivalent class, and further the equivalent class with the minimal complexity is selected so as to reduce scheduling sequences or prioritize scheduling sequences by using the presented approach, the efficiency of testing message-passing parallel programs will be greatly improved. Unfortunately, there is no approach to evaluating the complexity of an equivalent class so far. In the future, we will further research these topics.

Acknowledgements

This work is jointly supported by National Natural Science Foundation of China with grant No. 61375067 and No. 61503220.

References

[1] B. Beizer, Software Testing Techniques, Van Nostrand Reinhold, 1990.
[2] J.C. Munson, Software Engineering Measurement, CRC Press, 2003.
[3] S.R. Souza, M.A.S. Brito, R.A. Silva, P.S.L. Souza, E. Zaluska, Research in Concurrent Software Testing: A Systematic Review, in: Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, ACM, 2011, pp. 1–5.
[4] G.L. Chen, Parallel Computing: Structure Algorithm Programming, Higher Education Press, 2011.
[5] Z.H. Du, S.L. Li, Y. Chen, P. Liu, Parallel programming technology of high performance computing—MPI parallel programming, Beijing: Tsinghua University press, 2001.
[6] M. Christakis, K. Sagonas, Detection of asynchronous message passing errors using static analysis, Springer, 2011, pp. 5–18.
[7] A. Miné, Static analysis of run-time errors in embedded critical parallel C programs, in: European Symposium on Programming, Springer, 2011, pp. 398–418.
[8] S.F. Siegel, T.K. Zirkel, Automatic formal verification of MPI-based parallel programs, ACM SIGPLAN Notices 46 (8) (2011) 309–310.
[9] C. Flanagan, P. Godefroid, Dynamic partial-order reduction for model checking software, ACM SIGPLAN Notices 40 (1) (2005) 110–121.
[10] S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R.M. Kirby, R. Thakur, W. Gropp, Implementing efficient dynamic formal verification methods for MPI programs, in: European Parallel Virtual Machine/Message Passing Interface Usersí¯ Group Meeting, Springer, 2008, pp. 248–256.
[11] B. Krammer, M.M. Resch, Correctness Checking of MPI One-Sided Communication Using Marmot, Springer, 2006, pp. 105–114.
[12] J.S. Vetter, B.R. De Supinski, Dynamic Software Testing of MPI Applications with Umpire, in: Supercomputing, ACM/IEEE 2000 Conference, IEEE, 2000, p. 51.
[13] M.Y. Park, S.J. Shim, Y.K. Jun, H.R. Park, Mpirace-check: Detection of message races in MPI programs, in: International Conference on Grid and Pervasive Computing, Springer, 2007, pp. 322–333.

[14] Y. Lei, R.H. Carver, Reachability testing of concurrent programs, IEEE Trans. Softw. Eng. 32 (6) (2006) 382–403.
[15] R.H. Carver, Y. Lei, Distributed reachability testing of concurrent programs, Concurrency Comput. 22 (18) (2010) 2445–2466.
[16] S. Shivaprasad, N. Prasad, Unit testing concurrent java programs, Int. J. Comput. Appl. 67 (10) (2013) 41–46.
[17] G.H. Hwang, H.Y. Lin, S.Y. Lin, C.S. Lin, Statement-coverage testing for nondeterministic concurrent programs, in: International Symposium on Theoretical Aspects of Software Engineering, IEEE, 2012, pp. 263–266.
[18] R. Ferguson, B. Korel, Generating test data for distributed software using the chaining approach, Inf. Softw. Technol. 38 (5) (1996) 343–353.
[19] T. Tian, D.W. Gong, Test data generation for path coverage of message-passing parallel programs based on co-evolutionary genetic algorithms, Autom. Softw. Eng. (2014) 1–32.
[20] Q. Chen, L. Wang, Z. Yang, S.D. Stoller, HAVE: detecting atomicity violations via integrated dynamic and static analysis, in: International Conference on Fundamental Approaches to Software Engineering, Springer, 2009, pp. 425–439.
[21] J. Schimmel, K. Molitorisz, A. Jannesari, W.F. Tichy, Automatic generation of parallel unit tests, in: Proceedings of the 8th International Workshop on Automation of Software Test, IEEE Press, 2013, pp. 40–46.
[22] M.X. Liao, Z.H. Fan, Deadlock detection in basic models of MPI synchronization communication programs, Acta Electronica Sinica 36 (2) (2008) 402–407.
[23] C.S.D. Yang, L.L. Pollock, All-uses testing of shared memory parallel programs, Softw. Testing, Verification Reliab. 13 (1) (2003) 3–24.
[24] C.S.D. Yang, A.L. Souter, L.L. Pollock, All-du-path coverage for parallel programs, in: ACM SIGSOFT Software Engineering Notes, Vol. 23, ACM, 1998, pp. 153–162.
[25] S. Souza, S.R. Vergilio, P. Souza, A. Simao, A.C. Hausen, Structural testing criteria for message-passing parallel programs, Concurrency Comput. 20 (16) (2008) 1893–1916.
[26] A. Sen, M.S. Abadir, Coverage metrics for verification of concurrent SystemC designs using mutation testing, in: High Level Design Validation and Test Workshop (HLDVT), IEEE, 2010, pp. 75–81.
[27] S.D. Conte, H.E. Dunsmore, V.Y. Shen, Software Engineering Metrics and Models, Benjamin-Cummings Publishing Co., Inc., 1986.
[28] J. Ferrer, F. Chicano, E. Alba, Estimating software testing complexity, Inf. Softw. Technol. 55 (12) (2013) 2125–2139.
[29] B.A. Nejmeh, NPATH: a measure of execution path complexity and its applications, Commun. ACM 31 (2) (1988) 188–200.
[30] T.J. McCabe, A complexity measure, IEEE Trans. Softw. Eng. (4) (1976) 308–320.
[31] M.H. Halstead, Elements of Software Science (Operating and Programming Systems Series), Elsevier New York, 1977.
[32] T. Tian, D.W. Gong, Evolutionary generation of test data for path coverage through selecting target paths based on coverage difficulty, J. Zhejiang Univ. (Eng. Sci.) 5 (2014) 22–30.
[33] M.K. Debbarma, N. Kar, A. Saha, Static and dynamic software metrics complexity analysis in regression testing, in: 2012 International Conference on Computer Communication and Informatics (ICCCI), IEEE, 2012, pp. 1–6.
[34] M. Papadakis, N. Malevris, Mutation based test case generation via a path selection strategy, Inf. Softw. Technol. 54 (9) (2012) 915–932.
[35] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, T. Von Eicken, LogP: Towards a realistic model of parallel computation, in: ACM Sigplan Notices, Vol. 28, ACM, 1993, pp. 1–12.
[36] A. Alexandrov, M.F. Ionescu, K.E. Schauser, C. Scheiman, LogGP: incorporating long messages into the LogP modelí¬one step closer towards a realistic model for parallel computation, in: Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures, ACM, 1995, pp. 95–105.
[37] F. Ino, N. Fujimoto, K. Hagihara, LogGPS: a parallel computational model for synchronization analysis, in: ACM SIGPLAN Notices, Vol. 36, ACM, 2001, pp. 133–142.
[38] C.A. Moritz, M.I. Frank, LoGPC: Modeling network contention in message-passing programs, in: ACM SIGMETRICS Performance Evaluation Review, Vol. 26, ACM, 1998, pp. 254–263.
[39] T.H. Cormen, M.T. Goodrich, A bridging model for parallel computation, communication, and i/o, ACM Comput. Surv. (CSUR) 28 (4es) (1996) 208.
[40] S.E. Hambrusch, A. Khokhar, et al., C 3: An architecture-independent model for coarse-grained parallel machines, in: IEEE Symposium on Parallel & Distributed Processing, IEEE, 1994, pp. 544–551.
[41] R. Thakur, R. Rabenseifner, W. Gropp, Optimization of collective communication operations in MPICH, Int. J. High Perform. Comput. Appl. 19 (1) (2005) 49–66.
[42] K.J. Ottenstein, L.M. Ottenstein, The program dependence graph in a software development environment, in: ACM Sigplan Notices, Vol. 19, ACM, 1984, pp. 177–184.
[43] J. Cheng, Slicing concurrent programs-AGraph-Theoretical approach, Autom. Algorithmic Debugging (1993) 223–240.
[44] D. Pan, Researeh on exact model and complexity for task assignment problem in multi-core environment, Dalian University of Technology, 2009 Master's thesis.
[45] C.Q. Zhong, Software test data generation and defect detection based on dominant relationship between statements, China University of Mining and Technology, 2014 Master's thesis.
[46] L.P. Ma, The study and use of modern statistical analysis methods(three): Standardization of statistical data-dimensionless method 03 (2000) 34–35.
[47] G.L. Chen, H. An, L. Chen, et al., Parallel Algorithm Practice, Higher Education Press, 2004.
[48] H. Liu, B. Dai, Y. Zhang, W.-B. Zhang, Simple logp model's parameters simulate, J. Univ. Electron. Sci. Technol. China 34 (2) (2005) 229–232.