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

A R T I C L E I N F O

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

A B S T R A C T

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.