I.J. Computer Network and Information Security, **2015, 1, 1-8**
Published Online December 2014 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijcnis.2015.01.01

An Algorithm for Static Tracing of Message Passing Interface Programs Using Data Flow Analysis

Alaa I. Elnashar
Department of Computer Science, Faculty of Science, Minia University, Egypt.
College of Computers and Information Technology, Taif University, KSA.
Email: a.ismail@tu.edu.sa

Said F. El-Zoghdy
Department of Mathematics & Computer Science, Faculty of Science, Menoufia University, Egypt.
College of Computers and Information Technology, Taif University, KSA.
Email: elzoghdy@yahoo.com

Abstract—Message Passing Interface (MPI) is a well know paradigm that is widely used in coding explicit parallel programs. MPI programs exchange data among parallel processes using communication routines. Program execution trace depends on the way that its processes are communicated together. For the same program, there are a lot of processes transitions states that may appear due to the nondeterministic features of parallel execution. In this paper we present a new algorithm that statically generates the execution trace of a given MPI program using data flow analysis technique. The performance of the proposed algorithm is evaluated and compared with that of two heuristic techniques that use a random and genetic algorithm approaches to generate trace sequences. The results show that the proposed algorithm scales well with the program size and avoids the problem of processes state explosion which the other techniques suffer from.

Index Terms—Parallel Programming, Message Passing Interface, data and control flow analysis.

I. Introduction

Recently, multiprocessors and supercomputers are widely used due to the numerous advantages of parallelism. In addition, new programming languages and libraries are introduced to enable the users to create and control concurrent processes execution on such systems. Several parallel programming paradigms are used in coding parallel programs such as open MP [14], Parallel Virtual Machine (PVM) [58], and Message Passing Interface (MPI) [61] which is a popular paradigm in writing parallel programs since it provides a portable, efficient, and flexible standard for message passing.

MPI is a library of subroutines that enables programmers to identify and implement explicit parallelism using special constructs. It has been implemented for almost every distributed memory architecture and speed. In message passing paradigm, several separate processes are used to complete the overall computation.

Testing MPI parallel programs is more complicated than that of the sequential ones due to the increased complexity and the additional anomalous effects that may occur due to concurrency and interactions among parallel processes [12]. In general, finding guaranteed ordering of processes is an NP-complete problem [19]. Existing approaches to generate communication traces need to execute the entire parallel applications on full-scale systems, which are time-consuming and expensive [30]. Checking the conflict among the parallel processes in a given trace can be completed in a polynomial-time [19]. This problem is known as the monitoring problem [3, 4]. Another problem of interest is the checking of conflict for all possible interleaving of a given trace. This problem is called the prediction problem. In predictive analysis, a violation is discovered either in the observed trace or in an alternate interleaving of events in that trace.

Exhaustive techniques [32, 35, 42, 48] can be used for extracting the processes ordering within the execution trace by exploring all interleaving of a parallel program by switching processes at synchronization points. These techniques suffer from explosion problem since the number of transitions increase exponentially as the number of processes increases.

Some other techniques use Genetic Algorithms with communication-flow analysis to generate a feasible trace [6].

In this paper we present a new algorithm that statically explores the feasible trace of a given MPI program specification using data and communication flow analysis. The performance of the proposed algorithm is compared with that of both Random and Genetic Algorithms with communication flow analysis techniques. The results show that the proposed algorithm performs well against program size. It avoids the processes state explosion problem which the other techniques suffer from.

This paper is organized as follows: Section 2 presents the related work to the studied problem. Section 3 introduces some basic definitions and concepts. Section 4 presents the proposed algorithm for generating the executable transition sequence among set of parallel processes. Section 5 presents the evaluation of the proposed technique. Finally, section 6 summarizes this paper.

II. Related Work

Several studies [2, 24, 27, 45, 47] and tools like KOJAK [9], Paraver [25], VAMPIR [59] and TAU [54] concern with parallel programs trace collection using instrumentation methods have been proposed. These tools instrument the input program and execute it to get its trace. Some tools such as mpiP [28] collect statistical information only about MPI functions. All of these techniques need to execute the instrumented program, which restricts their usage for analyzing large-scale applications. Static verification [16, 22, 46, 51, and 57] and model checking [11, 31, 37, 41, 43, 60] are two approaches for program tracing and finding bugs in parallel programs. Model checking is an exhaustive search technique and does not scale with program size. Randomized algorithms for model checking like Monte Carlo Model Checking [42] have also been developed. These algorithms use a random walk on the state space to give a probabilistic guarantee of the validity. Randomized depth-first search [35] has been developed to reduce the state space search. A randomized partial order sampling algorithm [33] is used to sample partial orders almost uniformly at random. Race directed random testing [34] uses dynamic analysis tools to identify a set of pairs of statements that could potentially race in a parallel execution. Preissl et al. [44] developed a hybrid approach, with trace analysis to identify inefficient patterns in computation and communication. Strout et al. [36] developed a dataflow analyzer for MPI programs, which models information flow through communication edges.

De Souza et al. [21] presented Intel Message Checker (IMC) to perform a post-mortem analysis by collecting all information on MPI calls in a trace file. After executing the program, this trace file is analyzed by a separate tool or compared with the results from previous runs [23]. There are some different message-checking tools like MPI-CHECK [17], Umpire [26, 53], and MARMOT [7, 8]. These debuggers are effective in program tracing but still poor to detect semantics-related bugs [40].

Huang et al. [29] presented a static trace simplification technique for reducing the context switches in a parallel program execution trace. The technique constructs a dependence graph model of events; it scales linearly to the trace size and quadratic to the number of nodes in the dependence graph. Sinha et al. [5] presented a method to report a trace violation. The reported interleavings are guaranteed to be feasible in the actual program execution. Park and Sen [10] introduced a technique to detect real atomicity problems in parallel programs. Recently, Kelk et al. [13] employed genetic algorithms to implement an automated system for finding a feasible trace for parallel Java programs such that this trace doesn't contain deadlocks and data races.

A few studies [49, 52] have tried to compute a symbolic expression of the communication patterns for a given parallel program through data flow analysis. Shao et al. proposed a technique named communication sequence to present communication patterns of applications [52]. Ho and Lin described an algorithm for static analysis of communication structures in the programs written in a channel based message passing language [49]. Since these approaches only employ static analysis techniques trying to represent communication patterns of applications, they suffer from intrinsic limitations of static analysis. For example, they cannot deal with program branches, loops and the effects from input parameters.

III. Preliminaries

An MPI program, "**MPIP**" consists of several parallel executing, interacting, and communicating processes, which are concurrently processing a given input to compute the desired output. An MPIP containing np processes can be expressed as $MPIP = \{P_{id}\}$, where $P_{id}, 1 \le id \le np$ represent a parallel process that is identified by its rank, id. Each process consists of a set of sequential statements.

An execution trace $\rho = e_1, e_2, ..., e_n$ is a sequence of events, each of which is an instance of a visible operation during the execution of the parallel program [5].

Processes synchronization is essential when one process must wait for another one to finish before proceeding. Processes synchronization superimposes restrictions on the order of process performing. These restrictions are synchronization rules, which are described by means of synchronization primitives [55] such as "wait" and "send".

Communication flow refers to information exchange among the parallel processes which needs some kind of synchronization either implicit or explicit to guarantee a proper information exchange among these processes [20].

A statement $x \in P_i$ is a communication dependent on a statement $y \in P_j$, if the process P_i sends a message (sending operation) to process P_i (receiving operation) and $i \neq j$.

Happens-before relation [18] can be stated as: given a program execution trace, if a process P_i runs earlier than another process P_j, then P_i is said to be " happens-before" P_j, and is denoted by $P_i \xrightarrow{hb} P_j$. If the happen-before relation ($P_i \xrightarrow{hb} P_j$ or $P_j \xrightarrow{hb} P_i$) is not satisfied between P_i and P_j, the two processes are concurrent and denoted by $P_i \| P_j$. A write (read) access a_i is a key access if there does not exist any other write (read or

write) access a_j within a block such that $a_j \xrightarrow{hb} a_i$ [19].

A Control Flow Graph (CFG) representation for a sequential program SP is a directed graph $G = \{N, E, S, e\}$ where each node $n \in N$ represents a basic block of instructions, each edge $n \to m \in E$ represents a potential flow of control from node n to node m, and there is a unique start node s and a unique exit node e [56].

An MPI-CFG is specified as $CFG_{MPI} = \{V, E, C\}$, where V is the set of nodes in the graph, E is the set of control-flow edges, and C is the set of communication edges in the graph [35].

Data dependence between statements means that the program's computation might be changed if the relative order of statements is reversed [50]. The data dependence information computed by the reaching definitions is stored in the data structures of DU and UD chains. Def-use (DU) Chain links each definition of a variable to all of its possible uses. Use-def (UD) Chain links each use of a variable to a set of its definitions that can reach that use without any other intervening definition [2].

Static data flow analysis is a technique for gathering information about the possible set of values calculated at various points in a sequential program. CFG is used to determine those parts of a program to which a particular value assigned to a variable might propagate. This can be done by generating two sets, $dcu(i)$ and $dpu(i, j)$ [38] for program variables. These two sets are necessary to determine the definitions of every variable in the program and the uses that might be affected by these definitions. The set $dcu(i)$ is the set of all variable definitions for which there are def-clear paths to their c-uses at node i. The set $dpu(i, j)$ is the set of all variable definitions for which there are def-clear paths to their p-uses at edge (i, j) [39]. Using information concerning the location of variable definitions and references, together with the "basic static reach algorithm" [15], the two sets can be determined. The basic static reach algorithm is used to determine the sets reach(i) and avail(i). The set reach(i) is the set of all variable definitions that reach node i . The set avail(i) is the set of all available variables at node i. This set is the union of the set of global definitions at node i together with the set of all definitions that reach this node and are preserved through it. Using these two sets, the sets $dcu(i)$ and $dpu(i, j)$ are constructed from the formula:

$$dcu(i) = reach(i) \cap c - use(i)$$
$$dpu(i, j) = avail(i) \cap p - use(i, j)$$

IV. Tracing Algorithm

In this section we describe our proposed algorithm that finds the feasible trace of a given MPI program specification.

A. MPI program specification

MPI programs are coded in a special manner, in which each process executes the same program with unique data. All parallelism is explicit; the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs. For simplicity; instead of using explicit MPI programs, which are related to a specific programming language, we'll use MPI program specification. An example of MPI program specification is listed in Fig. 1.

```
1.  Begin                       20. if (pid = =3) then
2.  if (pid = =1) then          21. Def  x
3.  Receive (y, 2 )             22. Send (x, 2)
4.  if (y <=1) then             23. Receive (y2, 2)
5.  Def y1 = Use(y)             24. Def y3 = Use(y2)
6.  else                        25. Send (y3, 4)
7.  Def y1 = Use(y)             26. endif
8.  endif                       27. if (pid = =4) then
9.  Send (y1, 2)                28. Receive (y3, 3)
10. Receive (y5 , P5)           29. Def y4 = Use(y3)
11. endif                       30. Send (y4, 5)
12. if (pid = =2) then          31. Endif
13. Receive (x,3 )              32. if (pid = =5) then
14. Def y= Use(x)               33. Receive (y4, 4)
15. Send (y, 1)                 34. Def y5 = Use(y4)
16. Receive (y1, 1)             35. Send (y5, 1)
17. Def y2 = Use(y1)            36. Endif
18. Send (y2, 3)                37. End
19. endif
```

Fig.1. MPI Program Specification Example

The statements "Begin" and "End", lines (1, 37) refer to MPI environment initialization and termination. MPI environment management routines are used for initializing and terminating the MPI environment, querying the environment and identity. Processes in MPI programs are assigned to processors by using a conditional "if statement" depending on a unique identifier that represents the process rank. Each process will be assigned a unique integer rank between 1 and number of processes and identified by the variable "pid". For example, the program specification example contains 5 process ranked from 1 to 5, each process will execute only the block begins with its "if" statement and ending with its corresponding "endif" statement. Any other "if" statement appears in that block (lines 4 -9) and does not depend on the process rank will be treated as the ordinary conditional "if" statement. All arithmetic operations and assignment statements can be expressed as:

Def variable name = **Use** (list of used variable names)

MPI provides several routines used to manage the inter-process communications. In our specification we use two constructs, "Send" and "Receive". Messages are interchanged among the parallel processes via these two constructs, depending on the message variable name and the process rank according to the following syntax:

Send (variable to be sent, destination process rank) and

Receive (variable to be received, source process rank).

B. Design Overview

Our proposed technique consists of six components, "Source Program Scanner", " Sends / Receives Matching

Checker", "Transitions Pairs Builder", "Key Access Finder", "MPI-CFG Generator", and "Tracer" as shown in Fig. 2. Several data structures are used and generated to produce the execution trace.

The "Source Program Scanner" reads the input MPI specification and produces two data structures, "Sends" and "Receives". Each structure contains, for every wait/send, the process rank of the process which includes the wait/send statement, the statement number, variable name that this process waits for/sends to another process, and the number of the other process that will send/receive that variable.

The "Sends / Receives Matching Checker" checks the input program for a properly synchronization. It inspects the generated "Sends" and "Waits" structures to ensure that each "send/ wait" has a corresponding "wait/ send" primitive. If this test failed, a termination with "Error in Specification" will be generated; otherwise the "Transitions Pairs Builder" will be invoked.

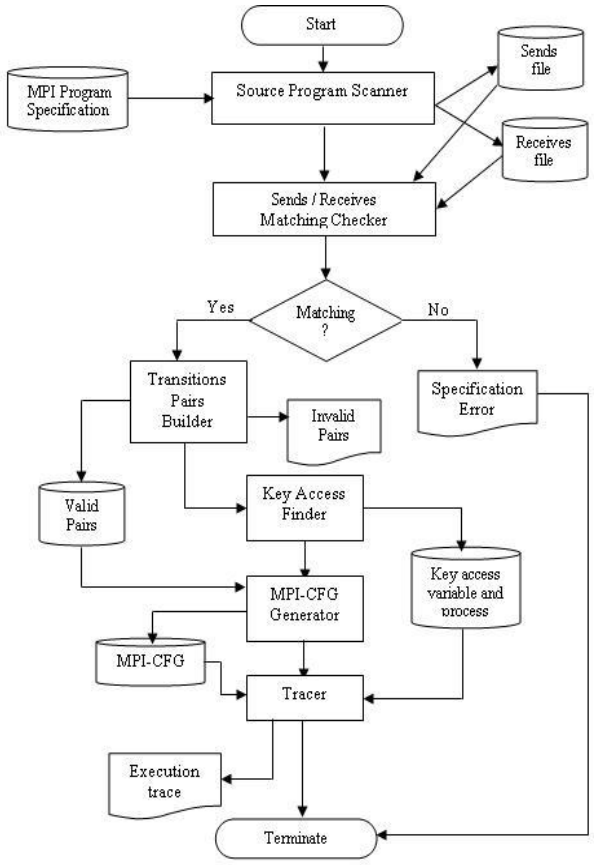

Fig. 2. Design Overview

The "Transitions Pairs Builder" uses the information gathered from the previous two phases to produce "Valid Transition Pairs" data structure which contains the valid transition pairs from one process to another one. Each entry in this structure contains two processes ranks. The trace length increases the number of valid transition pairs by one. Also, it generates a list of all "Invalid Transition Pairs". Tables 1.a and 1.b show both "Valid Transition Pairs" and "Invalid Transition Pairs" for the program example listed in Fig.1. The number of valid transition pairs is 7 and hence the trace length will be 8. 14 invalid transition pairs are recorded for the program example.

Table 1. Transition Pairs

a- Valid Transition Pairs	
From	**To**
1	2
2	1
2	3
3	2
3	4
4	5
5	1
b- Invalid Transition Pairs	
From	**To**
1	3
1	4
1	5
2	4
2	5
3	1
3	5
4	1
4	2
4	3
5	2
5	3
5	4

After the length of the trace is determined, the "Key Access Finder" is invoked to determine which process will start passing messages and also the key access variable that is not sent to this process from any one. The start process will be the first node in the execution trace. In our example the start process is process 3 and the key access variable is "x" (line 21) since there is no write "happens before" its definition. So the process containing this variable should be selected as the first one that starts the message passing scenario.

Then the "MPI-CFG Generator" builds the control flow graph of the input specifications by computing both the set $dcu(i)$ and $dpu(i, j)$ described in section 3. It also creates the required edges connecting graph nodes. Edges are classified into sequential, parallel and synchronization edges. Sequential edges represent the ordinary flow within a process. Parallel edges reflect the parallel nature of MPI programs. Synchronization edges are created to address synchronization and data dependence among the running processes. Both the start process and key access variable are also differentiated from the other processes and variables. Fig. 3 shows the MPI-CFG of the program specification example listed in Fig. 1.

The "Execution Tracer" traverses the generated MPI-CFG starting with the annotated key access and start process to produce the execution trace. The generated trace for our example represented by process' ranks will be (3 2 1 2 3 4 5 1). To confirm that the generated trace is feasible, we converted the program specification into an explicit c++ MPI code and then profiled its dynamic execution using Jumpshot [1] as shown in Fig. 4 which proves the trace feasibility. The main operations of the proposed technique are listed in Fig. 5.

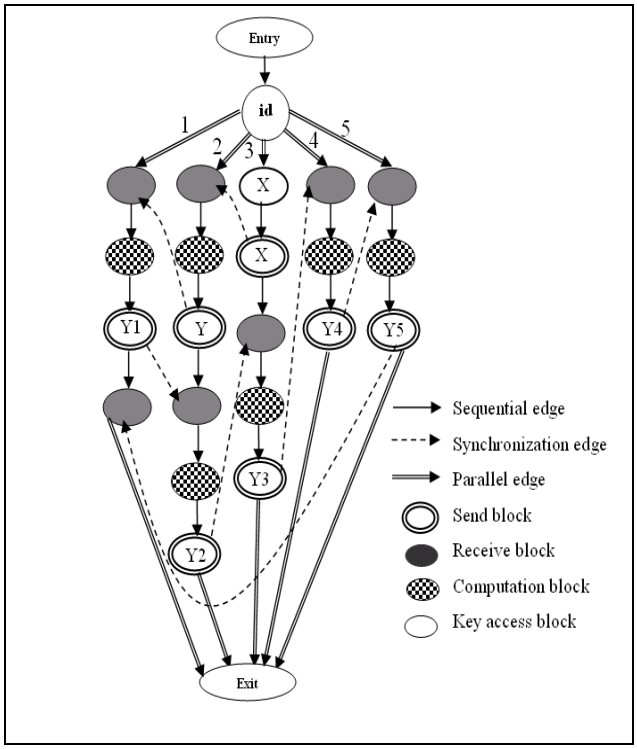

Fig. 3. MPI Control Flow Graph MPI-CFG

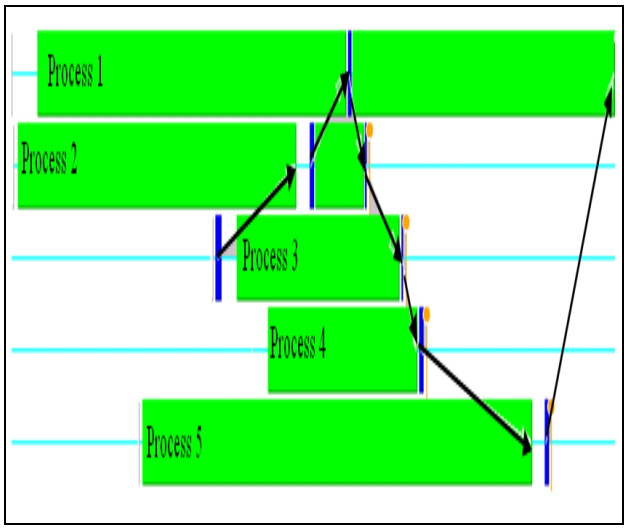

Fig. 4. Jumpshot Profiling

1. Trace $\rho = \varphi$
2. Trace_length $|\rho| = 0$
3. Scan "Waits" and "Sends" to detect key access and start process
   - 3.1 k_var = Key_access_var
   - 3.2 def_process = start process
   - 3.3 use_process = NULL
   - 3.4 def_var = k_var
   - 3.5 $\rho = \rho +$ start process
   - 3.6 $|\rho| = |\rho| + 1$
   - 3.7 $a_1 =$ start process
4. Expected trace length $|\rho_0|$ = number of valid transition pairs + 1
5. For each record in def-use file do
   - 5.1 If($\exists e: a_{j-1} \rightarrow a_j, 2 \le j \le \rho_0$) then
     - 5.1.1 use_process = a_j
     - 5.1.2 $\rho = \rho +$ use_process
     - 5.1.3 $|\rho| = |\rho| + 1$
     - 5.1.4 a_j = def_process
     - 5.1.5 increase j

     else
       get next record

   endif

   endfor
6. if ($|\rho| < |\rho_0|$) then
   - 6.1 ρ is a partial trace

   else /* $|\rho| = |\rho_0|$ */
   - 6.1 ρ is the execution trace

   endif

Fig. 5. Tracing Algorithm

V. Efficiency Evaluation

The proposed algorithm performance was compared with that of the two heuristic techniques introduced in [6]. The two techniques apply communication-flow analysis to generate all def-use pairs among the set of parallel processes. Then, one of them applies a genetic algorithm to generate a set of valid sequences of processes transition. The second one generates the set of valid sequences randomly. The generated set of valid sequences of transitions is passed to the feasibility checker to find the executable one. The results of the experiments showed that genetic algorithm is more efficient than the random technique in generating the executable transition sequence since it reduces the probability of state explosion. The two techniques still suffer from state explosion since the number of states exponentially increases as the number of processes increases as shown in Fig. 6.

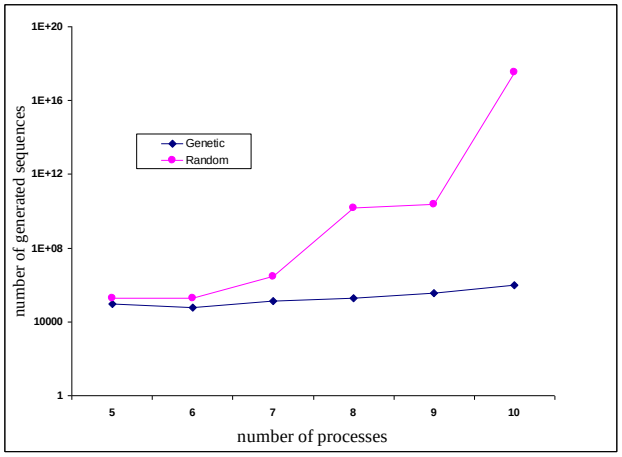

Fig. 6. Number of Transitions Sequences Generated by Random and Genetic Algorithm Techniques

Beside the problem of explosion, the generated sequences by the two techniques satisfy only the "def-use" pairs between each two successive nodes and hence many traces may be produced. Another pass searches for the sequence that does not only satisfy the "def-use" pairs, but also satisfies the synchronization primitives among all synchronized processes. If it fails to find such a sequence, a new set of sequences is generated. So, no guarantee that the required trace will be obtained.

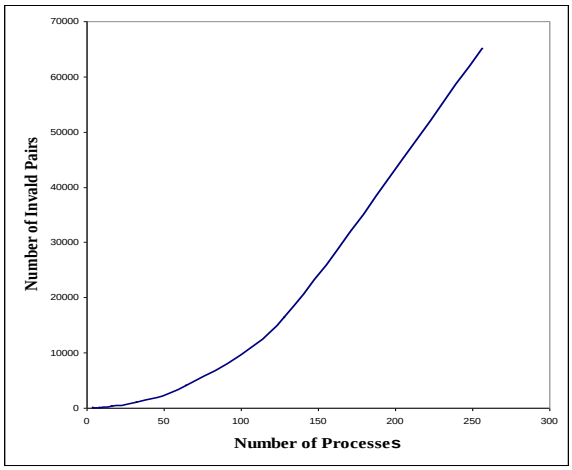

Fig.7. Invalid Pairs State Space Explosion

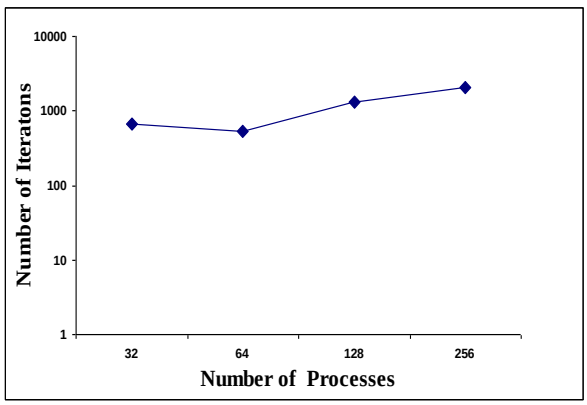

Fig. 8. Number of Iterations for Trace Generation

The proposed algorithm overcomes all the previous shortages arising in the heuristic techniques since it does not depend on heuristic generation of trace sequences. Fig. 7 shows that the number of invalid pairs, that should be avoided not to appear in the execution trace, exponentially increases as the number of processes increases.

Also, the proposed algorithm does not use such pairs; so it scales well with the number of processes and reduces the probability of state explosion as shown in Fig. 8.

VI. Conclusion And Future Work

In this paper, we presented a new algorithm for finding the execution trace of a given MPI program. This algorithm modifies the ordinary data flow analysis technique that is used in case of sequential programs to accommodate with the parallel nature of MPI programs. Unlike the heuristic approaches, the proposed algorithm builds its own control flow graph that represents the considered parallel program. This graph represents not only the control and data flow within the program, but also the inter-processes communication. Then the technique traverses this graph to generate the required trace. Another consideration that should be taken into account is that the proposed technique can deal with medium size programs since it does not suffer from state explosion as in the case of heuristic approaches. In future, we aim to modify the proposed technique to handle all MPI processes communication constructs.

References

[1] A. Chan D. Ashton, R. Lusk, and W. Gropp, Jumpshot-4 Users Guide, Mathematics and Computer Science Division, Argonne National Laboratory July 11, 2007.

[2] A. Faraj and X. Yuan. Communication characteristics in the NAS parallel benchmarks. In International Conference onParallel and Distributed Computing Systems, 2002.

[3] A. Farzan and P. Madhusudan, “Monitoring atomicity in concurrent programs,” in CAV, pp. 52–65, 2008

[4] A. Farzan and P. Madhusudan, “The complexity of predicting atomicity violations,” in TACAS, pp. 155–169, 2009

[5] A. Sinha, S. Malik, C. Wang, and A. Gupta. Predictive analysis for detecting serializability violations through Trace Segmentation. 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE). 2011.

[6] Ahmed S Ghiduk , Alaa I. Elnashar Automatic Generation of Valid Parallel-Processes Transition Using Genetic Algorithms and Communication-Flow Analysis, Int.J.Computer Technology & Applications,Vol 5 (3),973-982, 2014

[7] B. Krammer, K. Bidmon, M. S. M ¨uller and M. M. Resch. MARMOT: An MPI Analysis and Checking Tool. In PARCO 2003, Dresden, Germany, September 2003

[8] B. Krammer, M. S. M ¨uller, and M. M. Resch. MPI Application Development Using the Analysis Tool MARMOT. In ICCS 2004, volume LNCS 3038, pp. 464–471. Springer, 2004.

[9] B. Mohr and F. Wolf. KOJAK–A tool set for automatic performance analysis of parallel programs. In Euro-Par, 2003.
[10] Chang-Seo Park and Koushik Sen, "Randomized Active Atomicity Violation Detection in Concurrent Programs" Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, Pages 135-145 ACM New York, NY, USA, 2008
[11] D. Bruening. Systematic testing of multithreaded Java programs. Master's thesis, MIT, 1999.
[12] D. Kranzlmüller. Event Graph Analysis for Parallel Program Testing. PhD thesis, GUP Linz, Joh. Kepler University Linz, http://www.gup.uni-linz.ac.at/~dk/thesis. 2000.
[13] David Kelk, Kevin Jalbert and Jeremy S. Bradbury, "Automatically Repairing Concurrency Bugs with ARC", Lecture Notes in Computer Science, Multicore Software Engineering, Performance, and Tools International Conference, MUSEPAT 2013, St. Petersburg, Russia, Volume 8063, pp 73-84, 2013
[14] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation", In Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, pp. 97–104, September 2004.
[15] F. E. Allen and J. Cocke " A Program Data Flow Analysis Procedure," Communications of the ACM, vol. 9, p.137-147, 1976.
[16] G. Holzmann. The Spin model checker. IEEE Transactions on Software Engineering, 23(5):279–295, 1997.
[17] G. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou, "MPI-CHECK: a tool for checking Fortran 90 MPI programs", Concurrency and Computation: Practice and Experience, Volume 15, pp 93-100, 2003.
[18] Guy Martin Tchamgoue, Kyong Hoon Kim, and Yong-Kee Jun, Verification of Data Races in Concurrent Interrupt Handlers, International Journal of Distributed Sensor Networks, Volume 2013, 2013
[19] H. D. Park, and Y. K. Jun. First Race Detection in Parallel Program with Random Synchronization using Trace Information. International Journal of Software Engineering and Its Applications. 7, No.5, 2013, pp. 65-76.
[20] I Puaut. Operating systems - process management (SGP). Master's degree in computer science, http://www.irisa.fr/alf/downloads/puaut/Systeme/Lectures SGPEnglish.pdf. 2013.
[21] J. DeSouza, B. Kuhn, and B. R. de Supinski, "Automated, scalable debugging of MPI programs with Intel message checker", In Proceedings of the 2nd international workshop on Software engineering for high performance computing system applications, Volume 4, pp 78–82, ACM Press, NY, USA, 2005.
[22] J. E. M. Clarke, O. Grumberg, and D. A. Peled. Model checking. MIT Press, 1999.
[23] J. Huselius," Debugging Parallel Systems: A State of the Art Report", MRTC Report no. 63, September 2002.
[24] J. Kim and D. J. Lilja. Characterization of communication patterns in message-passing parallel scientific application programs. In CANPC, pages 202–216, 1998.
[25] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris DiP: A parallel program development environment. In EuroPar'96, pages 665–674, 1996.
[26] J. S. Vetter and B. R. de Supinski. Dynamic software testing of MPI applications with Umpire. In Supercomputing, pages 51–51. ACM/IEEE, 2000.
[27] J. S. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. In IPDPS, pages 853–865, 2002.
[28] J. S. Vetter and M. O. McCracken. Statistical scalability analysis of communication operations in distributed applications. In PPoPP, pages 123–132, 2001.
[29] Jeff Huang and Charles Zhang " ,An Efficient Static Trace Simplification Technique for Debugging Concurrent Programs", Lecture Notes in Computer Science Volume 6887, pp 163-179, 2011
[30] Jidong Zhai, Tianwei Sheng, Jiangzhou He, Wenguang Chen, Weimin Zheng, FACT: fast communication trace collection for parallel applications through program slicing, Proceedings of the Conference on High Performance Computing Networking, Storage and Analysi, 1 – 12, 2009
[31] K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. Int. Journal on Software Tools for Technology Transfer, 2(4):366–381, 2000.
[32] K. Sen and G. Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In Haifa verification conference 2006 (HVC'06), Lecture Notes in Computer Science. Springer, 2006.
[33] K. Sen. Effective random testing of concurrent programs. In 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07). 2007.
[34] K. Sen. Race directed random testing of concurrent programs. In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08), 2008.
[35] M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In 29th International Conference on Software Engineering (ICSE), pages 3–12. IEEE, 2007.
[36] M. M. Strout, B. Kreaseck, and P. D. Hovland, "Data-flow analysis for MPI programs." in Proceedings of the International Conference on Parallel Processing, pp. 175–184, 2006
[37] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In ACM Symposium on Programming Language Design and Implementation (PLDI'07), 2007.
[38] M. R. Girgis and M. R. Woodward " An Integrated System for Program Testing Using Weak Mutation and Data Flow Analysis", Proceedings of Eights International Conference on Software Engineering , IEEE Computer Society, p. 313-319, 1985.
[39] M. R. Girgis "Using Symbolic Execution and Data Flow Criteria to Aid Test Data Selection", software testing, verification and reliability, v. 3, p.101-113, 1993.
[40] N. Nethercote and J. Seward "Valgrind: A framework for heavyweight dynamic binary instrumentation", In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), San Diego, California, USA, 2007.
[41] P. Godefroid. Model checking for programming languages using verisoft. In 24th Symposium on Principles of Programming Languages, pages 174–186, 1997.
[42] R. Grosu and S. A. Smolka. Monte carlo model checking. In 11th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS

2005), volume 3440 of LNCS, pages 271–286, 2005.
[43] R. H. Carver and Y. Lei. A general model for reachability testing of concurrent programs. In 6th International Conference on Formal Engineering Methods (ICFEM'04), volume 3308 of LNCS, pages 76–98, 2004.
[44] R. Preissl, M. Schulz, D. Kranzlm ¨uller, B. R. de Supinski, and D. J. Quinlan. Transforming MPI source code based on communication patterns. Future Generation Comp. Syst. Vol. 26, No. 1, 2010, pp. 147–154.
[45] R. Zamani and A. Afsahi. Communication characteristics of message-passing scientific and engineering applications. In International Conference on Parallel and Distributed Computing Systems, 2005.
[46] S. Burckhardt, R. Alur, and M. M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In CM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI), pages 12–21, 2007.
[47] S. Chodnekar, V. Srinivasan, A. S. Vaidya, A. Sivasubramaniam, and C. R. Das. Towards a communication characterization methodology for parallel applications. In HPCA, 1997.
[48] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In International symposium on Software testing and analysis (ISSTA), pages 157–168. ACM Press, 2006.
[49] S. Ho and N. Lin. Static analysis of communication structures in parallel programs. In International Computer Symposium, 2002.
[50] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst., 12(1):26–60, 1990.
[51] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In ACM SIGPLAN 2004 conference on Programming language design and implementation (PLDI), pages 14–24. ACM, 2004.
[52] S. Shao, A. K. Jones, and R. G. Melhem. A compiler-based communication analysis approach for multiprocessor systems. In IPDPS, 2006.
[53] S. Sharma, G. Gopalakrishnan, and R. M. Kirby. A survey of MPI related debuggers and tools. 2007.
[54] S. Shende and A. D. Malony. TAU: The tau parallel performance system. International Journal of High Performance Computing Applications, 20(2), 2006.
[55] Saito, N. Synchronization Mechanisms for Parallel Processing. Lecture Notes in Computer Science Volume 143. 1982, pp. 1-22.
[56] Shires. D, Pollock. L,Sprenkle. S, rogram Flow Graph Construction For Static Analysis of MPI Programs, International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 1999, June 28 - Junlly 1, 1999, Las Vegas, Nevada, USA, 1847-1853
[57] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. SIGPLAN Not., 39(6):1–13, 2004.
[58] V. S. Sunderam, "PVM: A framework for parallel distributed computing", Concurrency: Practice & Experience, Volume 2, Number 4, pp 315–339, Dec. 1990. 33
[59] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. Supercomputer, 12(1), Jan. 1996.
[60] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In 15th International Conference on Automated Software Engineering (ASE). IEEE, 2000.
[61] Y. Aoyama J. Nakano "Practical MPI Programming", International Technical Support Organization, IBM Coorporation SG24-5380-00, August 1999. 34

Authors' Profiles

Alaa I. Elnashar was born in Minia, Egypt, in November 5, 1967. He received his B.Sc. and M.Sc. from Faculty of Science, Department of Mathematics (Math. & Comp. Science), and Ph.D. from Faculty of Science, Department of Computer Science, Minia University, Egypt, in 1988, 1994 and 2005. He is an associate professor in Faculty of Science, Computer Science Dept., Minia University, Egypt. Dr. Elnashar was a postdoctoral fellow at Kanazawa University, Japan. His research interests are in the area of Software Engineering, Software Testing, Parallel programming and Genetic Algorithms. Now, Dr Elnashar is an associate professor, Department of Information Technology, College of Computers and Information Technology, Taif University, Saudi Arabia

Dr. Said Fathy El-Zoghdy Was born in El-Menoufia, Egypt, in 1970. He received the BSc degree in pure Mathematics and Computer Sciences in 1993, and MSc degree for his work in computer science in 1997, all from the Faculty of Science, Menoufia, Shebin El-Koom, Egypt. In 2004, he received his Ph. D. in Computer Science from the Institute of Information Sciences and Electronics, University of Tsukuba, Japan. From 1994 to 1997, he was a demonstrator of computer science at the Faculty of Science, Menoufia University, Egypt. From December 1997 to March 2000, he was an assistant lecturer of computer science at the same place. From April 2000 to March 2004, he was a Ph. D. candidate at the Institute of Information Sciences and Electronics, University of Tsukuba, Japan, where he was conducting research on aspects of load balancing in distributed and parallel computer systems. From April 2004 to 2007, he worked as a lecturer of computer science, Faculty of Science, Menoufia University, Egypt. From 2007 until now, he is working as an assistant professor of computer science at the Faculty of Computers and Information Systems, Taif University, Kingdom of Saudi Arabia. His research interests are in load balancing in parallel and distributed systems, Grid computing, performance evaluation, network security and cryptography.

How to cite this paper: Alaa I. Elnashar, Said F. El-Zoghdy,"An Algorithm for Static Tracing of Message Passing Interface Programs Using Data Flow Analysis", IJCNIS, vol.7, no.1, pp.1-8, 2015. DOI: 10.5815/ijcnis.2015.01.01

