# Symbolic Execution of MPI Programs

Xianjin Fu*†, Zhenbang Chen†, Hengbiao Yu*†, Chun Huang†, Wei Dong†, and Ji Wang*†
*State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha, China
†College of Computer, National University of Defense Technology, Changsha, China
Email: {xianjinfu, zbchen}@nudt.edu.cn, fishzqyhb@gmail.com, {chunhuang, wdong, wj}@nudt.edu.cn

***Abstract*—MPI is widely used in high performance computing. In this extended abstract, we report our current status of analyzing MPI programs. Our method can provide coverage of both input and non-determinism for MPI programs with mixed blocking and non-blocking operations. In addition, to improve the scalability further, a deadlock-oriented guiding method for symbolic execution is proposed. We have implemented our methods, and the preliminary experimental results are promising.**

## I. Introduction

Message Passing Interface (MPI) is the *de facto* standard for developing applications in high performance computing. A lot of MPI programs are deployed to run on hundreds or even thousands of machines. The programs on these machines communicate by message passing to accomplish a parallel task. Because of the high parallelism, MPI programs are *non-deterministic*, and it is hard to ensure the correctness of MPI programs. The typical errors of MPI programs include deadlock, data type mismatch, resource error, *etc*.

The existing methods for finding the errors in MPI programs are mainly dynamic methods, which fall into two categories: correctness checking methods and verification methods. Correctness checking tools, such as Intel Message Checker [1], record the runtime information of an MPI program, and detect runtime errors, deadlocks or performance bottlenecks based on the recorded information. Hence, these tools may miss the errors that only happen under a specific case in non-determinism. On the other hand, dynamic verification tools, such as ISP [2], can explore different cases of the non-determinism caused by non-deterministic communication operations in MPI under a given input. However, both of these two types of methods cannot provide an input coverage, which results in missing input-related errors.

Symbolic execution [3] can systematically explore the path space of a program to have the input coverage. Hence, our basic idea is to use symbolic execution to analyze MPI programs. In this extended abstract, we report the current status of using symbolic execution to analyze MPI programs, providing coverage of *both* input space and non-determinism.

## II. MPI Programs

An MPI program will be run in terms of multiple independent processes on different machines. A process $P$ can use the communication APIs provided by the MPI platform (*e.g.* MPICH2) for communications. Non-blocking (or asynchronous) communication APIs are frequently used in MPI programs to improve the performance. The following three are representative APIs related to asynchronous communications.

- *ISend*(*dest*, *req*) - asynchronously send a piece of data to the process $P_{dest}$. Note, because this operation is a non-blocking operation, it returns immediately after being issued. The second parameter *req* is the indicator of the status of this operation.
- *IRecv*(*src*, *req*) - asynchronously receive a message from $P_{src}$. Note that $src$ can take the wildcard value "*", which means this operation expects a message from any process, and the receiving of any message finishes the operation. Hence, wildcard receives may cause non-determinism.
- *Wait*(*req*) - block until the message operation indicated by *req* is finished. This blocking operation is often used to ensure the readiness of the messages being exchanged.

Actually, many other non-blocking or blocking APIs can be composed by these three. In addition to the non-blocking APIs, there are blocking (or synchronous) communication APIs in MPI, which wait until the message exchanging is finished. In this extended abstract, we use *Send*(*dest*) and *Recv*(*src*) to represent the blocking sending and receiving APIs, respectively.

## III. Challenges for the Symbolic Execution of MPI Programs

There are two challenges for the symbolic execution of MPI programs:

1) Guarantee the coverage of the non-determinism caused by the non-deterministic operations in MPI programs.
2) Improve the scalability of the symbolic execution.

The first challenge is to ensure the soundness of analyzing MPI programs. Consider the following MPI program, in which *x* is an input variable and a deadlock error exists.

| $P_0$ | $P_1$ | $P_2$ |
|---|---|---|
| *Send*(1) | **if** (*x* == 'a') *Recv*(0); **else** *Recv*(*); *Recv*(2) | *Send*(1) |

No deadlock happens when *x* is 'a'. When *x* is not 'a', the deadlock will happen when the *Recv*(*) receives the message from $P_2$, because after it $P_1$ expects a message from $P_2$ and $P_0$ expects $P_1$ to receive its message. Hence, in principle, a deadlock may depend not only on the input, but also on the matchings of the wildcard receives. We can use symbolic execution to provide the input coverage, but *how to explore all the matchings of wildcard receives* is challenging, especially when considering non-blocking communications.

For the second challenge, symbolic execution has the path explosion problem, which challenges the scalability of any

analysis based on symbolic execution. Hence, the problem is inherited when we use symbolic execution. However, the problem becomes more challenging when analyzing MPI programs. Because the running of an MPI program consists of multiple processes, the execution of any statement in one process forks a new state during symbolic execution. Hence, the path space is exponential with the number of processes. Furthermore, the path space is also exponential with the number of the non-deterministic MPI operations during symbolic execution. Therefore, how to improve the scalability of the symbolic execution of MPI programs is challenging.

## IV. Methods and Implementation

We present in this section how the two challenges are coped with. Then, our implementation is briefly introduced.

### A. Coverage of Non-determinism

Our key observation of MPI programs is that the different processes of the program run independently without sharing memory. Hence, unlike multi-threaded programs, there is no data race errors in MPI application programs. This observation implies that the occurrence of an error does not depend on the sequence of the executions of the local statements in different processes. Thus, we use a *round-robin scheduler* to linearize the running of an MPI program under a given number of processes. A process is preempted when encountering a blocking operation such as *Send* or *Recv*. If a process is preempted, we turn to the next non-blocked process with the smallest rank, till all the processes are blocked.

The key idea of guaranteeing the coverage of non-determinism is called *lazy matching*. During symbolic execution, when MPI operations are encountered, the operations are recorded, but not executed. The symbolic execution procedure tries to advance the non-blocking processes until all the processes are blocked. Then, we match the recorded communication operations, and then do the symbolic execution of the matched operations. When there are multiple matches for a wildcard receive, we explore all of the matches by forking a new state for each. In this way, the potential matchings of wildcard receives can be systematically explored.

Note that, lazy matching permits the execution of the operations in an MPI program out of statement order. However, the "non-overtaking" rule required by MPI standard are preserved. For example, when two *send* operations (maybe non-blocking) in the same process are matched with a *receive* operation in another process, the first *send* operation in the program order should be matched with the *receive* operation.

### B. Scalability

We tackle the path explosion problem as follows. First, the *round-robin* scheduler avoids exploring all the interleavings of different processes. Inspired by the idea of partial order reduction, we only execute the local statements of the processes in one sequence, while ensuring the soundness. Second, in order to accelerate the discovery of deadlocks, we propose a deadlock-oriented guiding method for symbolic execution. We convert the guiding problem to a deadlock model checking problem, and leverage a state-of-the-art model checker to help the symbolic executor to find deadlocks faster.

### C. Implementation

Based on Cloud9 symbolic executor [4], we have implemented the proposed method for analyzing C MPI programs. We use a multi-threaded MPI library as the environment model for symbolic execution. Now, our prototype supports the analysis of the C MPI programs in which there exist both blocking and non-blocking operations. We can detect the errors including deadlock, runtime errors, synchronization error, *etc*.

## V. preliminary Experimental Results

Table I gives the preliminary experimental results of deadlock detection. The programs in the table are from [5], except *as-deadlock*, which is from [2]. For *as-deadlock*, guiding is slower than the case of no guiding, since the guiding method needs to explore at least two paths to detect a deadlock when the first path does not hit the deadlock.

TABLE I
Preliminary Experimental results

| Programs | #procs | #MPI calls | #path | Deadlock | Time(s) | |
|---|---|---|---|---|---|---|
| | | | | | guide | no guide |
| *GaussElim* | 4 | 40 | 48 | no | 17.5 | 17.3 |
| *Floyd* | 2 | 34 | 1 | no | 8.2 | 7.6 |
| *DTG* | 5 | 16 | 3 | no | 8.8 | 6.9 |
| *as-deadlock* | 3 | 15 | 2 | yes | 6.7* | 6.0* |
| *Heat* | 2 | 34 | 2 | no | 11.4 | 10.3 |
| | 5 | 85 | 300 | yes | 11.8* | 88.0* |

*mark means that the time is from start to the first found deadlock.

## VI. Related Work

The closest related work is TASS [6], which extracts a model from an MPI program and symbolic executes the model. The feasibility of TASS is limited by its simple MPI environment model. In addition, TASS does not support the analysis of asynchronous MPI programs. Similar to TASS, our previous work [7] is also targeting the MPI programs in which only synchronous operations exist.


## Acknowledgment

This work is supported in part by National 973 Program (2014CB340703) and National Natural Science Foundation (61120106006, 61272140, 61472440) of China.


## References


[1] J. DeSouza, B. Kuhn, B. R. de Supinski, V. Samofalov, S. Zheltov, and S. Bratanov, "Automated, scalable debugging of mpi programs with intel message checker," in *SE-HPCS*, 2005.

[2] S. Vakkalanka, G. Gopalakrishnan, and R. Kirby, "Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings," in *CAV '08*, 2008, pp. 66–79.

[3] J.King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[4] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *EuroSys '11*, 2011, pp. 183–198.

[5] V. Forejt, D. Kroening, G. Narayanaswamy, and S. Sharma, "Precise predictive analysis for discovering communication deadlocks in MPI programs," in *FM '14*, 2014, pp. 263–278.

[6] S. Siegel and T. Zirkel, "TASS: The toolkit for accurate scientific software," *Mathematics in Computer Science*, vol. 5, no. 4, pp. 395–426, 2011.

[7] X. Fu, Z. Chen, Y. Zhang, C. Huang, and J. Wang, "MPISE: Symbolic execution of MPI programs," in *HASE '15*, 2015.