# MPISE: Symbolic Execution of MPI Programs

Xianjin Fu*†, Zhenbang Chen*‡, Yufeng Zhang*†, Chun Huang*†, Wei Dong†, and Ji Wang*†
*College of Computer, National University of Defense Technology, Changsha, China
†Science and Technology on Parallel and Distributed Processing Laboratory,
National University of Defense Technology, Changsha, China
Email:{xianjinfu, zbchen, yfzhang, chunhuang, wdong, wj}@nudt.edu.cn
‡Corresponding Author

***Abstract*—Message Passing Interfaces (MPI) plays an important role in parallel computing. Many parallel applications are implemented as MPI programs. The existing methods of bug detection for MPI programs have the shortage of providing both input and non-determinism coverage, leading to missed bugs. In this paper, we employ symbolic execution to ensure the input coverage, and propose an on-the-fly schedule algorithm to reduce the interleaving explorations for non-determinism coverage, while ensuring the soundness and completeness. We have implemented our approach as a tool, called MPISE, which can automatically detect the deadlock and runtime bugs in MPI programs. The results of the experiments on benchmark programs and real world MPI programs indicate that MPISE finds bugs effectively and efficiently. In addition, our tool also provides diagnostic information and replay mechanism to help understand bugs.**

***Keywords*-Message Passing Interfaces; symbolic execution; deadlock detection**

## I. Introduction

In the past decades, Message Passing Interface (MPI) [1] has become the *de facto* standard programming model for parallel programs, especially in the field of high performance computing. A significant part of parallel programs were written with MPI, and many of them are developed in dozens of person-years [2].

Currently, the developers of MPI programs usually use traditional methods to improve the confidence of the programs, such as traditional testing and debugging [3] [4]. In practice, developers may waste a lot of time in testing, but only a small part of behavior of the program is explored. MPI programs have the common features of concurrent systems, including non-determinism, possibility of deadlock, *etc*. These features make the shortage of testing in coverage guarantee more severe. Usually, an MPI program will be run as several individual processes. The nature of non-determinism makes the result of an MPI program depend on the execution order of the statements in different processes. That is to say, an MPI program may behave differently with the same input on different executions. Hence, sometimes it is harder to find the bugs in an MPI program by a specific program execution.

To improve the reliability of MPI programs, many techniques have been proposed. Basically, we can divide the existing work into two categories: static analysis methods [5] [6] [7] and dynamic analysis methods [8]. A static method analyzes an MPI program without actually running it. The analysis can be carried out on code level [6] or model level [5] . Usually, a static method needs to make an abstraction of the MPI program under analysis [5] [7]. Therefore, many static methods suffer from the false alarm problem.

Dynamic methods, such as testing and runtime verification, need to run target MPI programs and utilize the runtime information to do correctness checking [9] [10] [11], online verification [8], debugging [4] [12], *etc*. Traditional testing methods work efficiently in practice by checking the correctness of a run under a given test harness. However, they cannot guarantee the coverage on non-determinism even after many runs of the same program input.

In this paper, we use symbolic execution to reason about all the inputs and try to guarantee the coverage on both input and non-determinism. We symbolically execute the statements in each process of an MPI program to find input-related bugs, especially runtime errors and deadlocks. For the non-determinism brought by the concurrent features, we use an on-the-fly scheduler to reduce the state space to be explored in the analysis, while ensuring the soundness and completeness. Specially, to handle the non-determinism resulted from the wildcard receives in MPI programs, we dynamically match the source of a wildcard receive into all the possible specific sources in a *lazy* style, which avoids the problem of missing bugs. Furthermore, unlike the symbolic execution plus model checking method in [6], which uses an MPI model to simulate the runtime behaviors of MPI library, we use a true MPI library as the model, which enables us to analyze real-world MPI programs.

To summarize, our paper has the following main contributions: firstly, we propose an on-the-fly scheduling algorithm, which can reduce unnecessary interleaving explorations while ensuring the soundness and completeness; secondly, when attacking the non-determinism caused by wildcard receives, we propose lazy matching technique, to avoid blindly matching which may lead to false positives; finally, we have implemented our approach in a tool called MPISE, and conducted extensive experiments to justify its effectiveness and efficiency in finding bugs in MPI programs.

## II. Background and Motivating Example

In this section, we briefly describe symbolic execution and the scope of the MPI APIs we are concerned with, then show how our algorithm works by a motivating example.

### A. Symbolic execution

Symbolic execution [13] is a program analysis technique originally introduced in the 1970s. The main idea is, rather than using concrete values, symbolic execution uses symbolic values as input values, and keeps tracking the results of numerical operations on symbolic values. Most importantly, symbolic execution uses a constraint of symbolic values, called path condition (PC), to represent a path of a program. At the beginning, the path condition is *true*. When encountering a branch statement, symbolic execution explores both directions of the branch. For exploring one direction, symbolic execution records (*i.e.*, conjunction) the condition $cond$ corresponding to the direction in PC and queries an underlying solver with $PC \wedge cond$ to decide whether this direction is feasible. If the answer is yes, symbolic execution will continue to execute the statements following the direction, and PC is update to be $PC \wedge cond$; otherwise, it means the direction is infeasible, thus symbolic execution backtracks to the branch statement, and starts to explore the other direction. Once symbolic execution reaches the end of a program, the accumulated PC represents the constraints that the inputs need to satisfy to drive the program to the explored path. Therefore, we can consider symbolic execution as a function that computes a set of PCs for a program. Naturally, we can use the PCs of the program to do automatic test generation [14], bug finding [14] [15], verification [16], *etc*.

According to the before explanation, symbolic execution is a precise program analysis technique, because each PC represents a real feasible path of the program under analysis. Therefore, when used for bug finding, symbolic execution does not suffer from the false alarm problem, and the bugs found are real bugs. Whereas, one of the major challenge symbolic execution faces is path space exploration, which is theoretically exponential with the number the branches in the program.

### B. MPI Programs

An MPI program is a program in which some MPI APIs are used. The running of an MPI program usually consists of a number of parallel processes, say $P_0, P_1, ..., P_{n-1}$, that communicate via message passings based on MPI APIs and the supporting platform. The message passing operators we consider in this paper include:

•**Send(dest):** send a message to $P_{dest}$ ($dest = 0, \ldots, n-1$), which is the destination process of the Send operation. Note that only synchronous communications are considered in this paper, so this operation blocks until a matching receive has been posted.

```
1 int main(int argc, char **argv) {
2   int x, y, myrank;
3   MPI_Comm comm = MPI_COMM_WORLD;
4
5   MPI_Init(&argc, &argv);
6   MPI_Comm_rank(comm, &myrank);
7   if (myrank==0) {
8     x = 0;
9     MPI_Ssend(&x,1,MPI_INT,1,99,comm);
10  }
11  else if (myrank==1) {
12    if(strcmp(argv[1],"a"))
13    MPI_Recv(&x,1,MPI_INT,0,99,comm,NULL);
14    else
15    MPI_Recv(&x,1,MPI_INT,MPI_ANY_SOURCE
          ,99,comm,NULL);
16
17    MPI_Recv(&y,1,MPI_INT,2,99,comm,NULL);
18  } else if (myrank==2){
19    x = 20;
20    MPI_Ssend(&x,1,MPI_INT,1,99,comm);
21  }
22  MPI_Finalize();
23  return 0;
24 }
```

Figure 1. Example showing the need for both input and non-determinism coverage

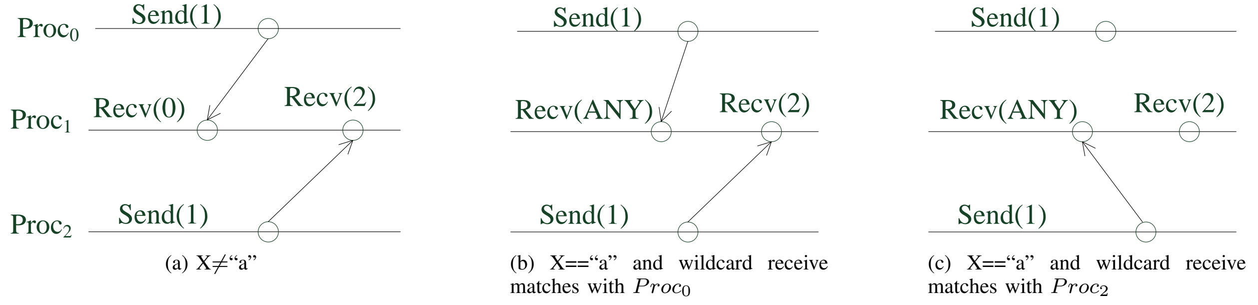
•**Recv(src):** receive a message from $P_{src}$ ($src = 0, \ldots, n-1, ANY$), which is the source process of the Recv operation. Note that the $src$ can take the wildcard value "ANY", which means this Recv operation expects messages from any process. Because Send and Recv are synchronous, a Send/Recv that *fails* to match with a corresponding Recv/Send would result in a *deadlock*.

•**Barrier():** synchronization of all processes, which means the statements of any process should not be issued past this barrier until all the processes are synchronized. Therefor, an MPI program is expected to eventually reach such a state that all the processes reach their barrier calls. If this does not hold, there would be a deadlock.

The preceding three MPI operations are the most important operations we consider in this paper. Actually, they cover the most frequently used *synchronous* communications in MPI programs. However, our implementation covers most of the frequently-used operations, even those collective communication and *non-blocking* ones, except that we do not treat wildcards in these operations, *i.e.*, just symbolic executing it as ordinary functions.

### C. Motivating Example

When an MPI program is fed with inputs to perform a computational task, the bugs of the program may be input-dependent. On the other side, due to the non-determinism feature, even with the same input, one may find that bugs occur "sometimes".




(a) X≠"a"

(b) X=="a" and wildcard receive matches with $Proc_0$

(c) X=="a" and wildcard receive matches with $Proc_2$

Figure 2. Three cases of the program in Figure 1

Usually, each process of an MPI program has a rank, which is initialized by MPI_Comm_rank at the beginning and used as the identity of the process. We always start from the smallest ranked process and switch to another process until the current process needs synchronization, such as sending or receiving a message. Thus, the switches during symbolic execution happen *on-the-fly*. Specifically, things become more complex when encountering a Recv(ANY) statement, where we need to delay the selection of the corresponding sending process until all the possible sending statements are encountered.

For the MPI program in Figure 1 run in three processes, we start from $Proc_0$, *i.e.*, the process with rank 0. When executing to line 9, a Send is encountered, which means a synchronization is needed. From the send statement, we know it needs to send a message to $Proc_1$. Thus, we switch to $Proc_1$ and do symbolic execution from the beginning. When the branch statement at line 12 is encountered, and $argv[1]$ is symbolic (we suppose it has a symbolic value $X$), the condition $X \neq$ "a" is added to the path condition of the true side and its negation to the false side. We mark here as a backtrack point and has two paths to follow, which are explained as follows:

•**X $\neq$ "a".** If we explore the true side first, the path condition, *i.e.*, $X \neq$ "a", is fed to the solver to check the feasibility of the path. Apparently, the solver will answer yes, thus we can continue the symbolic execution of $Proc_1$. Then, Recv(0) is meet and it is exactly matched with the send in $Proc_0$. Therefore, both processes advance, and $Proc_0$ ends while $Proc_1$ goes to Recv(2). In the same manner, $Proc_1$ gets asleep, we switch to $Proc_2$. Again the two operations matches, the whole execution will end normally, as shown in Figure 2(a).

•**X $==$ "a".** This side is also feasible. The symbolic execution of $Proc_1$ will encounter Recv(ANY), and switches to $Proc_2$. After executing the Send at Line 20, there is no process that can be switched to. All the possible sending processes of the Recv(ANY) in $Proc_1$ are determined. Thus, now we begin to handle the Recv(ANY) by marking it as a backtrack point, and explore by matching it with each possible sending. Suppose we match the Recv(ANY) with the Send of $Proc_0$, we continue to execute $Proc_1$. We encounter another Recv at Line 17 that expects to receive a message from $Proc_2$, then $Proc_1$ and $Proc_2$ advance, and finally the whole execution ends normally, as indicated by Figure 2(b). On the other hand, if the Recv(ANY) is matched with the Send of $Proc_2$, when encountering the Recv in $Proc_1$, symbolic execution will switch to $Proc_2$, but $Proc_2$ has finished. Then, $Proc_0$ and $Proc_1$ can not terminated. Hence, a deadlock is detected, as shown in Figure 2(c).

In summary, the deadlock, which may happen in the program in Figure 1 when run in three processes, can only be encountered when the input is "a" and the Recv(ANY) in the second process is matched with the Send in the third process. By using our approach, MPISE can detect it automatically. The details of our symbolic execution algorithms will be introduced in the next section.

## III. Symbolic execution algorithms

In this section, we will introduce a general framework for symbolic execution of MPI programs first, and then present a scheduling algorithm during the symbolic execution. Furthermore, to attack the non-determinism brought by wildcard receives, we will present a refined scheduling method, which can ensure the exploration of all the possible matches of a wildcard receive.

**Algorithm 1:** Symbolic Execution Framework

1 Search($MP$, $n$, $slist$){
2 $Active = \{P_0, \ldots, P_n\}$ ; $Inactive = \emptyset$;
3 NextProcCandidate= -1; worklist = {initial state};
4 **while** (*worklist is not empty*) **do**
5     $s$ = pick next state;
6     $p$ = *Scheduler*($s$);
7     **if** $p \neq null$ **then**
8         $stmt$ = the next statement of $p$;
9         SE($s$, $p$, $stmt$);
10 }

Algorithm 1 presents a general framework for symbolic execution of MPI programs. Basically, the symbolic execution procedure is a worklist-based algorithm. The input consists of an MPI program, the number of the parallel running processes and the symbolic variables. At the beginning, only the initial state, *i.e.*, composed by the initial states of all the processes, is in the worklist. Then, new states can be derived from the current state and put into the worklist. State exploration is done if there is no state in the

worklist. Clearly, it is hard or even impossible to explore the whole path space. In fact, for the state forking introduced by the concurrent feature, sometimes there is no need to add all the possible successor states to the worklist, which can still capture the behavior of the program precisely in our context. Hence, different from the usual symbolic execution algorithm, in our algorithm, we first select a state from worklist (Line 5, where a search algorithm can be used), then we make a decision (Line 6, the details of which will be given in Section III-A) of which process is scheduled for symbolic execution. Finally, we symbolically execute the next statement of the scheduled process, in which some new states may be generated.

Basically, for the non-communication statements in an MPI program, the symbolic execution semantics is the same as usual. In the following of this section, we will concentrate on explaining the scheduling of the processes and the handling of the communication operations.

### A. *On-the-fly scheduling*

With the general framework in Algorithm 1, we introduce our scheduler here, aiming to avoid naively exploring the interleavings of all the processes. For each process of an MPI program during symbolic execution, the process is *active* if it is not asleep. Usually, we make a process asleep when the process needs to communicate but the corresponding process is not ready, whose details will be given in Algorithm 3. We maintain the current status of each process via two sets: $Active$ and $Inactive$. At beginning, all the processes are contained in $Active$. If a process is made to be asleep, it will be removed from $Active$ and added to $Inactive$. Because we schedule the processes on-the-fly, we use a global variable $NextProcCandidate$ to denote the index of the next process to symbolically execute. The following Algorithm 2 gives how to do scheduling.

**Algorithm 2:** Scheduling the Next Process for Symbolic Execution

```
1  Scheduler(s){
2  if NextProcCandidate! = −1 and Proc_NextProcCandidate
   is active in s then
3      Next = NextProcCandidate;
4      NextProcCandidate = −1;
5      return Proc_Next;
6  else if Active ≠ ∅ then
7      return the process p′ with the smallest rank in Active;
8  if Inactive ≠ ∅ then
9      Report Deadlock;
10 }
```

First, we check whether there is a next process that needs to be executed and is also active. If there exists one, the process identified by $NextProcCandidate$ will be selected, and the next process global variable is reset (Line 1∼5); otherwise, we return the active process with the smallest rank if exists (Line 6∼7). Finally, if there is no active process that can be scheduled, and the $Inactive$ set is non-empty, *i.e.*, there exists at least one process that does not terminate, we report that a deadlock is found (Line 8∼9).

Now, we explain how to symbolically execute each statement. In Algorithm 3, we mainly give the handling for MPI APIs considered in this paper. The local statements in each process do not influence the other processes, and the symbolic execution of such basic statementsis the same with the traditional approach [14]. Hence, the symbolic execution of local statements is omitted for the sake of space.

In Algorithm 3, Advance($S$) denotes the procedure in which the program counter of each process in $S$ will be advanced, and Match($p$, $q$) denotes the procedure in which the synchronization between $p$ and $q$ happens, *i.e.*, the receiver receives the data sent by the sender, and the program counters of $p$ and $q$ will be both advanced.

If a $Send(dest)$ is encountered and there is a process in $Inactive$ that matches the statement, we move that process from $Inactive$ to $Active$ (Line 6) and advance the two processes (Line 7). If there is no process that can receive the message, we add this process into *Inactive* set (Line 9), and switch to the destination process of the send operation (Line 10). The execution of a receive operation is similar, except that when the receive operation is a wildcard receive, we make the current process asleep (the reason will be explained in Section III-B).

For handling barriers, we use a global variable $mc_b$ to denote the rest processes that need to reach a barrier for a synchronization. When a barrier statement is encountered, if $mc_b$ is empty, we initialize $mc_b$ to be the set containing the rest processes (Line 26) and add the current process into $Inactive$ (Line 27). If $mc_b$ is not empty, we remove the current process from $mc_b$. Then, if $mc_b$ is empty, *i.e.*, all the processes have reached a barrier, we can advance all the processes (Line 31) and make all the processes active (Line 32); otherwise, we add the current process into *Inactive* set (Line 35). When encountering an Exit statement, which means the current process terminates, we remove the current process from $Active$ (Line 38).

In summary, according to the two algorithms, the symbolic execution process will continue to execute the active process with the smallest rank until a preemption happens. From a state in symbolic execution, we do not put all the possible states into the worklist, but only the states generated by the current process. This is the reason why we call it on-the-fly scheduling. Actually, we only explore part of the whole program path space, but without sacrificing the ability of finding deadlock bugs. The correctness of our on-the-fly scheduling algorithms is guaranteed a theorem, along with it's proof is given in the extended version of this paper [17].

**Algorithm 3:** Symbolic Execution of a Statement

```
1  SE(s, p, stmt){
2  switch kindof(stmt) do
3      case Send(dest)
4          if stmt has a matched process q ∈ Inactive then
5              Inactive = Inactive \ {q};
6              Active = Active ∪ {q};
7              Match(p, q);
8          else
9              Inactive = Inactive ∪ {p};
10             NextProcCandidate = dest;
11         return;
12     case Recv(src)
13         if src != MPI_ANY_SOURCE then
14             if stmt has a matched process q ∈ Inactive
               then
15                 Inactive = Inactive \ {q};
16                 Active = Active ∪ {q};
17                 Match(p, q);
18             else
19                 Inactive = Inactive ∪ {p};
20                 NextProcCandidate = src;
21         else
22             Inactive = Inactive ∪ {p};
23         return;
24     case Barrier
25         if mc_b == ∅ then
26             mc_b = {P_0, ..., P_n} \ {p};
27             Inactive = Inactive ∪ {p};
28         else
29             mc_b = mc_b \ {p};
30             if mc_b == ∅ then
31                 Advance({P_0, ..., P_n});
32                 Inactive = ∅;
33                 Active = {P_0, ..., P_n};
34             else
35                 Inactive = Inactive ∪ {p};
36         return;
37     case Exit
38         Active = Active \ {p};
39         return;
40     ...
41 Advance({p});
42 }
```

## B. Lazy matching algorithm

Note that so far, we do not treat wildcard receives. Actually, wildcard receives are one of the major reasons of non-determinism. Clearly, we cannot blindly rewrite a wildcard receive. For example, in Figure 3a, if we force the wildcard receive in $Proc_1$ to receive from $Proc_2$, a deadlock will be reported, which actually will not happen.

In addition, if we rewrite a wildcard receive immediately when we find a possible match, we still may miss bugs. As shown in Figure 3b, if we match the wildcard receive in $Proc_0$ with the send in $Proc_1$, the whole symbolic execution will terminate successfully, thus a deadlock, which will appear when the wildcard receive is matched with the send in $Proc_2$, is missed.

| $Proc_0$ | $Proc_1$ | $Proc_2$ |
|---|---|---|
| Send(1) | Recv(ANY) | local statements |

(a) Blind rewriting of a wildcard receive

| $Proc_0$ | $Proc_1$ | $Proc_2$ |
|---|---|---|
| Recv(ANY) ; Recv(2) | Send(0) | Send(0) |

(b) Eager rewriting of a wildcard receive

Figure 3. Rewriting of a wildcard statement

To solve this problem, we employ a lazy style approach instead of an eager one. That is, we delay the selection of the send candidate of a wildcard receive until the whole symbolic execution procedure blocks. To be detailed, when the symbolic execution encounters a wildcard receive, we would make the current process asleep (Line 22 in Algorithm 3), waiting for all possible senders. When a matched send is found, the current process will also be made asleep, and we switch to the next active process. When there is no process that can be scheduled, *i.e.*, all the processes are in $Inactive$, we match the wildcard receive to each possible matched send by forking a successor state for each one. Thus, Algorithm 2 needs to be refined to handle wildcard receives. The refined parts are given as follows.

**Algorithm 4:** Refined Scheduling

```
1  Scheduler(s){
2  ...
3  if Inactive ≠ ∅ then
4      if Exists a Recv(ANY) process in Inactive then
5          PS = Inactive;
6          for each Recv(ANY) process p ∈ Inactive do
7              for each matched process q ∈ Inactive of p do
8                  Inactive = PS \ {p, q};
9                  Active = {p, q};
10                 AddState(s, p, q);
11         return null;
12     else
13         Report Deadlock;
14 }
```

For each process encountering a wildcard receive in $Inactive$, we add a new state for each of its matched sender processes (Line 10). The AddState($s$, $p$, $q$) denotes a procedure that does the synchronization between $p$ and $q$, advances both $p$ and $q$, and adds the new state to the worklist. Thus, we are exploring all the possible cases of a wildcard receive. If there are multiple Recv(ANY) processes, we are interleaving the matches of all the processes. The proof of correctness of our handling for wildcard receives is ensured

by the interleavings of matches and exploring all the possible matches of a wildcard receive. The proof of the correctness is provided in [17].

## IV. Implementation and Experiments

### A. Implementation

We have implemented our approach as a tool, called MPISE, based on Cloud9 [18], which is a distributed symbolic executor for C programs. Cloud9 enhances KLEE [14] by enabling the support of most POSIX interfaces and parallelism. The architecture of MPISE is shown in Figure 4.

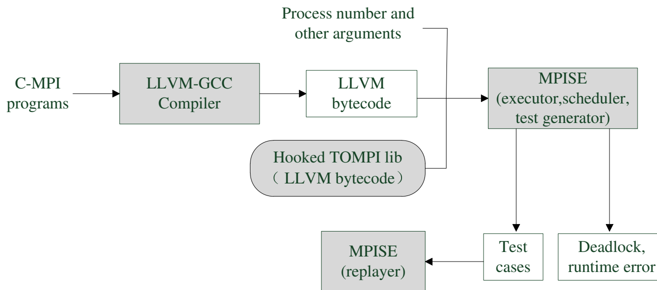



Figure 4. The architecture of MPISE.

A target MPI program written in C is fed into LLVM-GCC compiler to obtain the LLVM bytecode, which will be linked with a pre-compiled library, *i.e.*, TOMPI [19], as well as the POSIX runtime library. Then, the linked executable program will be symbolically executed. Basically, TOMPI is a platform that uses multi-threads to simulate the running of an MPI program. TOMPI provides a subset of MPI interfaces, which contains all the MPI APIs we consider in this paper. By using TOMPI, we can use the support for concurrency in Cloud9 to explore the path space of an MPI program run with a specific number of processes. When a path ends or a deadlock is detected, MPISE records all the information of the path, including the input, the orders of message passings, *etc*, for the *replayer* to reproduce a concrete path.

### B. Experimental evaluation

We have conducted extensive experiments to validate the effectiveness and scalability of MPISE. Using MPISE to analyze the programs in the Umpire test suite [8], in which each test case is an input-independent program. All the experiments were conducted on a Linux server with 32 cores and 250 GB memory.

In our experiments, we also use ISP and TASS to analyze the programs. The experimental results are displayed in Table I, in which we divide the experimental results into two categories: input independent programs and input dependent ones. For each category, we select programs that can deadlock caused by different reasons, including head to head receive, wait all, receive any, *etc*. For each input dependent program, we generate the input randomly when analyzing the program with ISP, and analyze the program for 10 times, expecting to detect a deadlock. The execution time of analyzing each input dependent program with ISP is the average time of the 10 times of runnings. According to the experimental results, we can conclude as follows:

MPISE can detect the deadlock in all the programs. ISP misses the deadlock for all the input dependent programs. TASS fails to analyze most of programs. Thus, MPISE outperforms ISP and TASS for all the programs in Table I. The reason is, MPISE uses symbolic execution to have an input coverage guarantee, and the scheduling algorithms ensures that any deadlock caused by the MPI operations considered in this paper will not be missed. In addition, we utilize TOMPI and Cloud9 to provide a better environment support for analyzing MPI programs. The reason of the common failure of TASS is that TASS does not support many APIs, such as *fflush*(stdout) of POSIX and MPI_Get_Processor_Name of MPI, and needs manually modifying the analyzed programs.

For each program, the analysis time using MPISE is longer than that of using ISP. The reason is two fold: firstly, we need to symbolically execute the bytecodes including those of the underlying MPI library, *i.e.*, TOMPI. For example, for the input dependent program *barrier-deadlock.c*, the number of the executed instructions is 182304. Secondly, the time used by MPISE includes the linking time of the target program byte code and TOMPI library. In addition, we need to record states and do solving during symbolic execution, which also needs more time than dynamic analysis.

To validate the scalability of MPISE, we use MPISE to analyze three real-world MPI programs, including an MPI program (CPI) for calculating $\pi$ and two C MPI programs (DT and IS) from NSA Parallel Benchmarks (NPB) 3.3 [20] with class S. The lines of code (LOC) of DT is 1.2K and the LOC of IS is 1.4K. MPISE can analyze these three programs successfully, and no deadlock is found. The experimental results are displayed in Figure 5. Because IS can only be run with $2^n$ $(n \geq 1)$ processes, we do not have results for the case of 6 processes.

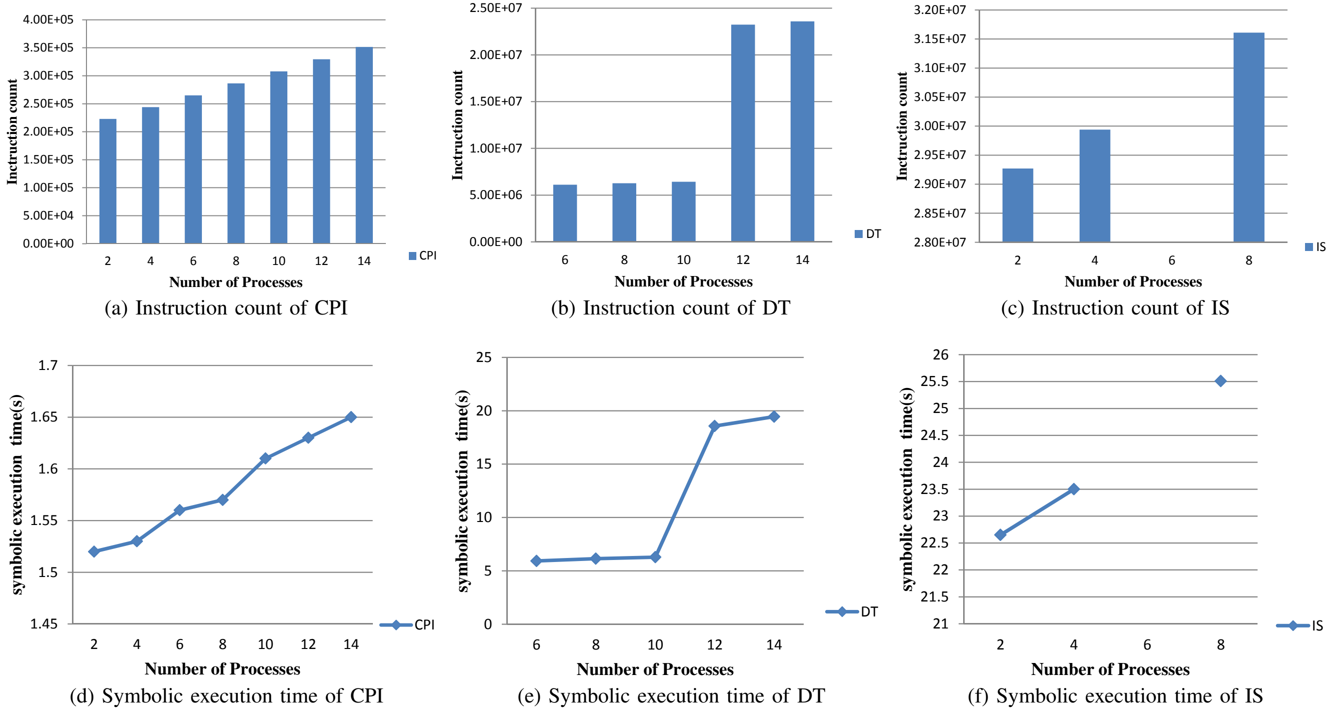
From Figure 5, we can observe that, for all the three programs, the number of the executed instructions and the symbolic execution time do not increase exponentially with respect to the number of processes. It justifies that MPISE avoids the exponential increasing of instructions or symbolic execution time caused by the parallelism by the on-the-fly scheduling algorithms. Note that we make the input of DT symbolic ones, and this program aborts early when fed with input string “BH” and the process number that is less than 12, this explains the sudden rise of both analyze time and instructions in Figure 5(b) and Figure 5(e).

## V. Related Work

There are already some existing work for improving the reliability of MPI programs [2]. Generally, they often fall

Table I
EXPERIMENTAL RESULTS

| | Program | ISP | | TASS | | MPISE | |
|---|---|---|---|---|---|---|---|
| | | Result | Time(s) | Result | Time(s) | Result | Time(s) |
| Input Indep-endent | anysrc-deadlock.c | Deadlock | 0.126 | Fail | 1.299 | Deadlock | 1.59 |
| | basic-deadlock.c | Deadlock | 0.022 | Fail | 1.227 | Deadlock | 1.46 |
| | collect-misorder.c | Deadlock | 0.022 | Fail | 0.424 | Deadlock | 1.48 |
| | waitall-deadlock.c | Deadlock | 0.024 | Fail | 1.349 | Deadlock | 1.49 |
| | bcast-deadlock.c | Deadlock | 0.021 | Fail | 0.493 | Deadlock | 1.40 |
| | complex-deadlock.c | Deadlock | 0.023 | Fail | 1.323 | Deadlock | 1.46 |
| | waitall-deadlock2.c | Deadlock | 0.024 | Fail | 1.349 | Deadlock | 1.48 |
| Input Depe-ndent | barrier-deadlock.c | No | 0.061 | Fail | 0.863 | Deadlock | 1.71 |
| | head-to-head.c | No | 0.022 | Fail | 1.542 | Deadlock | 1.67 |
| | rr-deadlock.c | No | 0.022 | Fail | 1.244 | Deadlock | 1.67 |
| | recv-any-deadlock.c | No | 0.022 | Deadlock | 1.705 | Deadlock | 1.70 |
| | cond-bcast.c | No | 0.021 | No | 1.410 | Deadlock | 1.63 |
| | collect-misorder.c | No | 0.023 | Deadlock | 1.682 | Deadlock | 1.85 |
| | waitall-deadlock3.c | No | 0.104 | Fail | 1.314 | Deadlock | 1.78 |




(a) Instruction count of CPI (b) Instruction count of DT (c) Instruction count of IS

(d) Symbolic execution time of CPI (e) Symbolic execution time of DT (f) Symbolic execution time of IS

Figure 5. The experimental results under different numbers of processes

into one of the following two categories: debugging and testing methods, and verification methods.

Debugging and testing tools often scale well, but depend on concrete inputs to run MPI programs, expecting to find or locate bugs. Debugging tools such as TotalView [12] and DDT [4] are often effective when the bugs can be replayed consistently. Whereas, for MPI programs, reproducing a concurrency bug caused by non-determinism is itself a challenging problem. Another kind of tools, such as Marmot [9], the Intel Trace Analyzer and Collector [10] and MUST [21], intercept MPI calls at runtime and record the running information of an MPI program, and check runtime errors, deadlock or analyze performance bottlenecks based on the recorded runtime information. These tools often need to recompile or relink MPI programs, and also depend on the inputs and the scheduling of each running.

Another line of tools are verification tools. Dynamic verification tools, such as ISP [8] , provide a coverage guarantee over the space of MPI non-determinism. When two or more matches of a non-deterministic operation, such as wildcard receive is detected, the program will be re-executed, and each running using a specific match. Based on ISP, in [22], a SAT-based predicative method is proposed to improve the generation of the schedules for revealing the deadlock errors in MPI programs. These tools can find the bug relying on a particular choice when non-deterministic

operations are encountered, but also depend on the inputs that are fed to run the program. TASS [6] tackles the limitation by using symbolic execution to reason about all the inputs of an MPI program, but its feasibility is limited by the simple MPI model used, which is justified in Section IV-B.

Compared with the existing work, MPISE provides coverage guarantee on both input and non-determinism by symbolically executing MPI programs and using an on-the-fly scheduling algorithm to handle non-determinism. In addition, MPISE uses a realistic MPI library, *i.e.*, TOMPI [19], to be the MPI model. Therefore, more realistic MPI programs can be analyzed automatically by MPISE, without modifying the programs manually.

## VI. Conclusion

MPI plays a significant role in parallel programming. To improve the reliability of MPI applications, we propose MPISE in this paper to use symbolic execution to analyze MPI programs, targeting to find the bugs of an MPI program automatically. Existing work on analyzing MPI programs suffers problems in different aspects, such as scalability, feasibility and input or non-determinism coverage. We employ symbolic execution to tackle the input coverage problem, and propose an on-the-fly algorithm to reduce the interleaving explorations for non-determinism coverage, while ensuring the soundness and completeness. We have implemented a prototype of MPISE as an adoption of Cloud9, and conducted extensive experiments. The experimental results show that MPISE can find bugs effectively and efficiently. MPISE also provides diagnostic information and utilities to help people understand a bug.

For future work, there are several aspects. In one aspect, we plan to support non-blocking MPI operations, which are widely used in nowadays MPI programs. In another aspect, we want to refine our MPI model further, *e.g.*, using a more realistic library, to improve the precision of symbolic execution. Finally, we are also concerned with improving the scalability of MPISE and the analysis of production-level MPI programs.

## Acknowledgements

This work is supported by the National 973 project 2014CB340703, and the National NSFC projects (Nos. 61120106006, 61272140 and 61472440).

## References

[1] Message Passing Interface Forum, "MPI: A message-passing interface standard,version 2.2." http://www.mpi-forum.org/docs/, 2009.

[2] G. Gopalakrishnan, R. M. Kirby, S. Siegel, R. Thakur, W. Gropp, E. Lusk, B. R. De Supinski, M. Schulz, and G. Bronevetsky, "Formal analysis of MPI-based parallel programs," *Commun. ACM*, vol. 54, no. 12, pp. 82–91, Dec. 2011.

[3] "The GNU debugger." http://www.gnu.org/software/gdb/.

[4] "Allinea DDT." http://www.allinea.com/products/ddt/.

[5] S. F. Siegel, "Verifying parallel programs with MPI-spin," in *PVM/MPI 07*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 13–14.

[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] M. M. Strout, B. Kreaseck, and P. D. Hovland, "Data-flow analysis for MPI programs," in *ICPP '06*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 175–184.

[8] 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*, ser. Lecture Notes in Computer Science, A. Gupta and S. Malik, Eds. Springer Berlin Heidelberg, 2008, vol. 5123, pp. 66–79.

[9] B. Krammer, K. Bidmon, M. S. Müller, and M. M. Resch, "Marmot: An MPI analysis and checking tool," in *PARCO*, ser. Advances in Parallel Computing, G. R. Joubert, W. E. Nagel, F. J. Peters, and W. V. Walter, Eds., vol. 13. Elsevier, 2003, pp. 493–500.

[10] P. Ohly and W. Krotz-Vogel, "Automated MPI correctness checking what if there was a magic option?" in *8th LCI International Conference on High-Performance Clustered Computing*, South Lake Tahoe, California, USA, May 2007.

[11] J. S. Vetter and B. R. de Supinski, "Dynamic software testing of mpi applications with umpire," in *Supercomputing '00*. Washington, DC, USA: IEEE Computer Society, 2000.

[12] "Totalview Software." http://www.roguewave.com/products/totalview.

[13] J.King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[14] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI '08*. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224.

[15] P. Godefroid, M. Levin, D. Molnar *et al.*, "Automated whitebox fuzz testing," in *NDSS '08*, 2008.

[16] X. Deng, J. Lee, and R. Bogor/Kiasan, "A k-bounded symbolic execution for checking strong heap properties of open systems," in *ASE '06*, 2006, pp. 157–166.

[17] X. Fu, Z. Chen, Y. Zhang, C. Huang, and J. Wang, "MPISE: Symbolic execution of MPI programs," http://arxiv.org/abs/1403.4813, 2014.

[18] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *Proceedings of the Sixth Conference on Computer Systems*. New York, NY, USA: ACM, 2011, pp. 183–198.

[19] E. Demaine, "A threads-only MPI implementation for the development of parallel programs," in *Proceedings of the 11th International Symposium on High Performance Computing Systems*, 1997, pp. 153–163.

[20] "NSA parallel benchmarks." http://www.nas.nasa.gov/Resources/Software/npb.html.

[21] T. Hilbrich, J. Protze, M. Schulz, B. R. de Supinski, and M. S. Müller, "MPI runtime error detection with must: Advances in deadlock detection," in *SC '12*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 30:1–30:11.

[22] 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.