Synchronization Error Detection of MPI Programs by Symbolic Execution

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

***Abstract*—Asynchrony based overlapping of computation and communication is commonly used in MPI applications. However, this overlapping introduces *synchronization errors* frequently in asynchronous MPI programming. In this paper, we propose a symbolic execution based method for detecting input-related synchronization errors. The path space of an MPI program is systematically explored, and the related operations of the synchronization errors in the program are checked specifically. In addition, two optimizations are proposed to improve the efficiency. We have implemented our method as a prototype tool based on the symbolic executor Cloud9. The results of the extensive experiments indicate the effectiveness of our method.**

***Keywords*-MPI, Synchronization Error, Symbolic Execution**

I. Introduction

Nowadays, lots of high performance computing (HPC) applications are implemented using Message Passing Interface (MPI). In fact, MPI has been the *de-facto* standard in this field. Given the growing size of clusters, MPI programs are becoming more and more complex, some of which even cost dozens of person-years, with lots of efforts on developments and optimizations.

To optimize MPI applications, one of the major principles is: allowing overlapping of the computation tasks of MPI applications and the communication of the underlying MPI library. This overlapping is usually archived by calling *non-blocking* MPI functions. Unlike blocking calls, non-blocking MPI functions return immediately after called, without waiting until the corresponding buffer being copied out/in. In this way, MPI applications can continue to do computation task, without waiting communications to finish, leaving the data transmission in MPI library to the network controller.

The overlapping of computation and communication could reduce the latency of costly communications. However, it also introduces *synchronization errors* [1], which frequently happen during optimizing MPI programs. The following situation shows a typical synchronization error: the application reuse the buffer of a send/receive call when the send/receive operation has not copied out/in the data in the buffer. When an MPI application overwrites a send buffer before the completion of data transmission by MPI library, the incorrect data may be sent out, or even worse, the data is corrupted, leading to an incorrect computation result or even crashing the MPI computation. Similar things could happen if the MPI application use a receive buffer before MPI library finishes the data receiving and copies the data into the buffer.

To detect synchronization errors, many methods have been proposed. Most existing work uses dynamic program analysis methods [2] [1]. For example, Umpire [2] computes the checksum of the message data in a non-blocking send, and recomputes it in the corresponding completion check calls to detect synchronization errors. SyncChecker [1] checks the temporal order of the related operations of a message buffer at runtime via instrumenting MPI programs. However, dynamic methods need to run the programs with some concrete inputs, which would miss input-related synchronization errors since they do not guarantee input space coverage.

To illustrate such input related errors, take the program in Figure 1 for example, where X is the input and P_0 sends a message to P_1. If $X = 0$ holds, P_1 would use the receiving buffer to do computation. However, this could lead to a synchronization error: if the data transmission of *buff* is not finished, c may get an uninitialized data. On the other hand, if $X = 0$ does not hold, P_0 and P_1 communicate normally.

```
P0                        P1
ISend(to P1, buff, req)   IRecv(from P0, buff, req)
Wait(req)                 if(!X) c=buff[0]
                          Wait(req)
```

Figure 1. An example demonstrating input-related synchronization errors

Therefore, tools that do not provide input space coverage would surely fail to detect this kind of bugs if the program is not fed with particular inputs.

In this paper, we use symbolic execution to reason all the possible inputs of an MPI program. When doing symbolic execution of an MPI program, along each path of the program, the sequence of the related operations of a message buffer will be checked to detect synchronization errors. When a synchronization error is detected, the corresponding program input will be generated automatically, which can be used to help the programmers to localize the error.

A. Contribution

This paper focuses on how to analyze MPI programs with asynchronous communications via symbolic execution, aiming to hunt for synchronization errors. To summarize, this paper makes the following contributions:

- To eliminate the input non-determinism of non-blocking MPI programs, we employ symbolic execution to reason all possible inputs. To avoid exploring all possible interleavings of all MPI processes, we propose to use a round-robin style scheduler, which eliminates unnecessary interleavings without hurting the ability of finding synchronization errors.
- Under the framework of symbolic execution-based analysis, we propose a method to detect synchronization errors. In addition, two optimization methods are proposed to improve the efficiency further.
- A prototype tool of our method has been developed based on Cloud9 [3]. Extensive experiments on real world MPI programs have been conducted. The experimental results indicate the effectiveness and efficiency of our tool.

To the best of our knowledge, we are the first to support the symbolic execution of non-blocking MPI programs in such scale, with a real world MPI library rather than a "model", providing a comprehensive framework to analyze MPI programs.

B. Outline

Section II introduces the background and motivates the basic idea of our method. Section III describes the details of the algorithm. Section IV explains our implementation and Section V shows the experimental results. Related work is discussed in Sections VI and the conclusion is drawn in Section VII.

II. Overview

In this section, we introduce the background and the main idea of our method.

A. MPI Programs

An MPI program is a sequential C/Fortran program in which some MPI APIs are used. Usually, an MPI program will be run in terms of a number of parallel processes, say $P_0, P_1, ..., P_{n-1}$, spread on one or multiple machines. Each process runs the same program, and the processes communicate via message passings, using different MPI APIs and the supporting MPI platform. In addition to the synchronous APIs [4], the asynchrony related APIs we consider in this paper include:

- *ISend*(*dest*, *buff*, *req*) - asynchronously send *buff* to P_{dest} $(dest = 0, \ldots, n-1)$, which is the destination process of the *ISend* operation. Different from blocking send operation, this operation is non-blocking, it returns immediately after issued. The third parameter *req* is a flag to indicate whether the data in the *buff* has been sent out by data transfer routines like memcpy.
- *IRecv*(*src*, *buff*, *req*) - asynchronously receive a message from P_{src} $(src = 0, \ldots, n-1, ANY)$ and store it to *buff*. P_{src} is the source process of the *IRecv* operation. Same as *IRecv*, this operation is also non-blocking. The flag *req* indicates whether the message has been well received. Note that P_{src} can take the wildcard value "ANY", which means this receive operation expects a message from any process.
- *Wait*(*req*) - wait until the message operation flagged by *req* is finished. This is a blocking operation, and often used in asynchronous MPI programming to ensure the readiness of the messages being exchanged. For example, the *Wait*(*req*) in the process P_1 in Figure 1 means the process can not continue to execute the following statements until the message in the *IRecv* operation is well received.

The above three APIs are the most frequently used *asynchronous* communciation APIs in MPI programs. In the following of this paper, if the context is clear, we use *ISend*(*buff*, *req*) and *IRecv*(*buff*, *req*) to denote *ISend*(*dest*, *buff*, *req*) and *IRecv*(*src*, *buff*, *req*), respectively.

B. Symbolic execution

Symbolic execution [5] is a program analysis technique with the main idea of using symbolic inputs instead of concrete values to execute a program, tracking the results of the numerical operations on symbolic values. Meanwhile, a symbolic expression, called *path condition* (PC), is also kept during symbolic execution, which holds the conditions that the inputs should satisfy to drive the program along the current path. Initially, PC is $true$. When a branch statement is encountered, both directions of the branch are followed. For one direction, provided that $cond$ is the branch condition, $PC \wedge cond$ is submitted to the underlying solver to check the satisfiability to decide whether this direction is feasible. If yes, PC is updated to $PC \wedge cond$, or else it means that this direction is not feasible, symbolic execution backtracks to explore the other direction. Therefore, the path conditions of a program constitute a symbolic representation of the program considering the input coverage. However, the path space of a program under symbolic execution is exponential with the number of the branch statements in the program, which is a major challenge for symbolic execution based program analysis.

C. Main idea

The basic idea of our method is to use symbolic execution to explore the path space of an MPI program, and track the status of each buffer along each path to ensure that the events related to this buffer happen in a permitted order. If a violation of the order happens, an error is reported, and the report includes the input of the MPI program leading the program to the error.

To be more detailed, given an MPI program P and the number N of the processes that the program needs to be run as, we need to execute P with N processes in a symbolic way. Thus, for an MPI program, its theoretical path space is exponential with both of the numbers of branch statements and the processes in running, which brings significant challenges to symbolic execution. The basic idea of our scheduling method here is to use a round-robin fashion style schedule, *i.e.*, sequentially selecting the *active* process with the smallest rank to symbolic execute, doing process switching when specific statements are encountered. This way of scheduling can reduce the number of interleavings under the condition of ensuring the completeness of the analysis. More details of our scheduling will be given in Section III.

Basing on the symbolic execution of P, we can detect the synchronization errors as follows. For a message buffer in P, we track the following events: (1) the issue of each ISend/IRecv call related to the buffer in the MPI program (ISSUE), (2) the data transmission of the buffer carried out by MPI library (TRANS), (3) the completion check of the data transmission in the MPI program (CHK) and (4) the usage of the buffer in the MPI program (USE). The anticipated event sequence of the buffer should be as follows.

$$\text{ISSUE} \rightarrow \text{TRANS} \rightarrow \text{CHK} \rightarrow \text{USE} \qquad (1)$$

In the sequence, $\rightarrow$ means the "happened-before" relation [6]. Any violation of this anticipated sequence will be reported as a synchronization error. Hence, when doing symbolic execution of P, we interpret the *critical points* in P and MPI library, including the *ISend*/*IRecv* calls, the completion check calls of data transmission such as *Wait*, the data transmission calls such as *memcpy* and the data usage statements. Then, for each path t explored during symbolic execution, we check whether the event sequence of each buffer involved in t conforms to the anticipated sequence in (1), and our checking is buffer-sensitive. In addition, leveraged by symbolic execution, we can also check the runtime errors of the program, including division by zero, buffer overflow, *etc*.

Take the program in Figure 1 for example. Suppose we symbolic execute the program in two processes, *i.e.*, P_0 and P_1, and the input X is made to be a symbolic input, P_0 is executed at the beginning, with respect to our scheduling method. Then, when encountering the *Wait* statement, the execution of P_0 will be switched to P_1 in the case that the data transmission is not finished. Then, when the execution of P_1 reaches the branch statement, the symbolic execution forks two states:

- One represents "$X \neq 0$", along which the path is feasible, and the usage of *buff* will not happen in the next, so no error will be detected.
- The other one represents "$X = 0$", and the path is also feasible. Along the path, there will be a usage of *buff*, but the completion checking (to be done by the Wait(req) statement) has not been done for the buffer, thus an error will be reported. The error report will include a value of the input X, *e.g.*, X is 0.

Finally, the symbolic execution terminates, and a synchronization error is detected.

III. The Method of Synchronization Error Detection

In this section, we first introduce a general framework for the symbolic execution of MPI programs, and then present our algorithm to detect synchronization errors.

A. Symbolic execution framework

The basic procedure of symbolic execution is usually a worklist-based search algorithm. Algorithm 1 shows the main procedure of our symbolic execution framework for MPI programs. The input of the algorithm consists of an MPI program MP, the number of the processes to run n and the symbolic variables $slist$. At the beginning, only the initial state, *i.e.*, composed by the initial states of all the processes, is contained 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.

Because of state forking, we need a strategy for space exploration, such as *depth first search* (DFS) and *breadth first search* (BFS). Because of the parallelism, the non-determinism of an MPI program is not only caused by the inputs, but also the interleavings of the executions of different parallel processes. For a state during symbolic execution, the execution of any process can make the symbolic executor to fork a new state. Thus, the state explosion of the symbolic execution of MPI programs becomes more severe. Clearly, it is usually impossible to explore the whole path space. Actually, in our context, we observe that we do not need exhaustively exploring all schedule, since synchronization errors are not sensitive to interleavings. The reason is that the related events of a buffer usually occur in the same process, hence the happened-before relations are guaranteed by the execution order of local statements.

Therefore, in our framework, we employ a round-robin style scheduler to fix a schedule for MPI programs to be symbolic executed. The scheduler is a non-preemptive one, *i.e.*, a process would keep being executed until it encounters blocking MPI calls (the handling of which can be referred to [4]) or other explicit preemption points such as *sleep*. The second rule of this scheduler is: always pick the process with the smallest rank to symbolic execute, *i.e.*, if there are n processes and the process P_i is blocked, the next process to be executed is $P_{((i+1) \mod n)}$ if $P_{((i+1) \mod n)}$ is not blocked; otherwise, the scheduler continues to check the next process until an active process is encountered. In

Algorithm 1: Symbolic Execution Framework

```
  Input: program $MP$, process number $n$, symbolic
         value list $slist$
1 Search($MP$, $n$, $slist$){
2 worklist = {initial state};
3 while (worklist is not empty) do
4     $s$ = pick next state;
5     $p$ = Scheduler($s$);
6     if $p \neq null$ then
7         $stmt$ = the next statement of $p$;
8         SE($s$, $p$, $stmt$);
9 }
```

Algorithm 1, after we select a state from worklist (Line 4, where a search algorithm can be used), we decide which process is scheduled for symbolic execution (Line 5), *i.e.*, pick a process according to the aforementioned round-robin scheduler. Finally, we symbolic execute the next statement of the scheduled process (Line 8), in which some new states may be generated. And the details of SE in line 8 would be given in the next subsection.

For the statements that do not affect synchronization errors, such as non-communicative statements and synchronous communication statements, the symbolic execution semantics can be referred to [7] [4]. In the following of this section, we will concentrate on the handling of synchronization error related events and the optimizations.

B. Synchronization error checking while symbolic execution

We use the notion originated from object-orientation specification, called *typestate* [8], to name the specification we use in detecting synchronization errors. A typestate usually specifies the requirement of the operation sequence of an object according the current state of the object. A typestate property is usually described using a finite state machine (FSM). Therefore, for a message buffer here, we also use an FSM in Figure 2 to specify its typestate property.

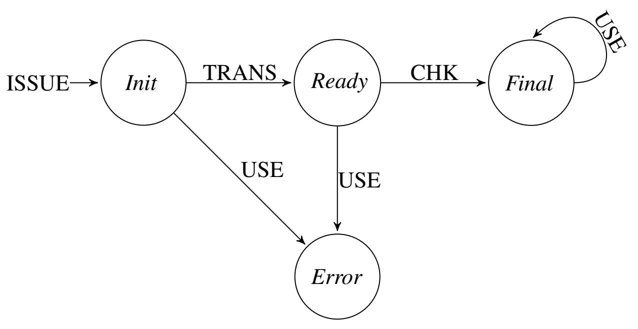

Figure 2. Typestate specification of a buffer

As indicated in Figure 2, the usage of the buffer at the *Init* state or the *Ready* state would lead to the error state. Only when the completion checking of data transmission is done, the data usage is safe. Even when the data in the buffer is transferred but without a completion checking, we consider the usage of the data as an error. The reason is that: considering current existing MPI implementations, it would be possible that there exists a case such that the data usage can happen before the completion checking of the data transmission.

Algorithm 2 gives how to handle different statements during symbolic execution, where $inter(b, a, n)$ represents the intersection of the memory area of the buffer b and the memory area of size n starting from the address a, and $in(a, b)$ represents that the address a falls into the memory area of the buffer b. We only present the operations related to synchronization error detection. The rest operations, such as symbolic arithmetic operations and state forking, are omitted for the sake of space.

As indicated in Subsection II-C, we intercept MPI asynchronous sends/receive calls, data transmission checking calls and data transmission calls. For intercepting the data usages of a buffer, we check at each memory access point, since our implementation works on the virtual machine level. To record the status of each buffer, we use a global map, *i.e.*, the *map* variable in Algorithm 2, to record the related information of a buffer, including the communication mark (*req*), the current state (*state*) and the request type (*type*).

To be more detailed, we mark a buffer to be initialized (*Init*, Line 6) after its asynchronous Send/Recv call being issued, and then the buffer goes to ready state (*Ready*, Lines 11 & 13) if its data transmission (*e.g.*, memcpy) is finished by MPI library. The buffer goes to the final state (*Final*) after the corresponding completion checking call is executed (Line 17). Note that MPI_Test can also confirm whether a communication is finished, the handling of which is similar to that of Wait.

When dealing with buffer usages, *i.e.*, memory loads and stores, we find that the reads of an ISend buffer do not affect the results. Whereas, for IRecv buffers, both memory loads and memory stores cause errors. Hence, we report an error when an unchecked ISend buffer is written, or an unchecked IRecv buffer is read or written (Lines 22 & 27).

Note that for simplicity, we only deal with continues data type in Algorithm 2. However, we can deal with non-continues data types by attaching a data type mask with each buffer. The mask indicates the displacements of the sub data types in the non-continues data type. When checking synchronization error at a load/store, we are concerned with not only the memory area, but also the data type mask, which is checked to ensure that the memory access indeed hits a meaningful part of the buffer.

However, the execution of an MPI program may have thousands of memory accesses, leading to costly overheads

Algorithm 2: Symbolic Execution of Statements

```
1  SE($s$, $p$, $stmt$){
2  switch $kindof(stmt)$ do
3      ...
4      case ISend(buff, req) or IRecv(buf, req)
5          map(buff).req = req;
6          map(buff).state = Init;
7          map(buff).type = SEND/RECV; /* resp. */
8          return;
9      case dataTransfer(dest, src, n)
10         if $\exists v \in map \bullet v$.type=SEND$\wedge inter(v$.buff$, src, n) \neq \emptyset$ then
11             $v$.state = Ready;
12         if $\exists v \in map \bullet v$.type=RECV$\wedge inter(v$.buff$, dest, n) \neq \emptyset$ then
13             $v$.state = Ready;
14         return;
15     case Wait(req)
16         if $\exists v \in map \bullet v$.req $= req$ then
17             $v$.state = Final;
18         return;
19     case Load(addr)
20         if $\exists v \in map \bullet in(addr, v$.buff$) \wedge v$.type $=$ RECV then
21             if $v$.state $== Ready$ or $v$.state $== Init$ then
22                 Report an error;
23         return;
24     case Store(addr)
25         if $\exists v \in map \bullet in(addr, v$.buff$)$ then
26             if $v$.state $== Ready$ or $v$.state $== Init$ then
27                 Report an error;
28         return;
29     ...
30 ...
31 return;
32 }
```

because of our analysis. In addition, when dealing with Wait, Load and Store, we need to travel the global map, whose cost is linear to the map size. Hence, we introduce two optimizations: one for reducing unnecessary memory access checking, and the other for reducing map size, which will be introduced in the next sub-section.

C. Optimizations

During the symbolic execution of an MPI application, if we apply synchronization error checking to each memory access made by the application or MPI library, there would be a significant overhead. We observe that only the memory accesses of the MPI applications are related to synchronization errors. Hence, if we found that a memory access is issued by MPI library, we simple execute it without any checking, since the memory access does not affect any synchronization error.

Clearly, in Algorithm 2, each time when we interpret a Wait or a memory access statement, the checking cost is linear to the size of the global map. We observe that: if a buffer is already at *Final* state, *i.e.*, the data transmission is finished and checked, either reading or writing of the buffer will not lead to synchronization errors. Hence, we can delete the corresponding items of the buffers that are at *Final* state in the global map, to reduce the checking cost.

The effectiveness of these two optimizations will be justified in Section V.

D. Discussion

Our method does not fully handle the non-determinism resulted by the receive from any operations in MPI programs, especially when there are asynchronous communications in the program. Hence, in principle, our analysis is *not sound* but complete. However, when only synchronous communications exist, the method in [4] can be used, which ensures the soundness.

Since we employ symbolic execution to detect synchronization errors, we sacrifice part of the scalability. We argue that: first, the sacrificed part is compensated by the detection of input-related errors; second, synchronization errors are not very sensitive to parallelism, which alleviates the state explosion problem.

Note that the overlapping of the computation and communication in MPI applications can also be achieved by RDMA (Remote Direct Memory Access), such as InfiniBand [9]. Since RDMA functions show no difference with the non-blocking ones discussed in the paper, we believe our method can also be applied to the MPI programs using RDMA functions.

IV. Implementation

We have implemented our method in a prototype tool, called MPISE, based on Cloud9 [3], which is a KLEE [7] enhanced symbolic executor for C programs with more complete POSIX environment support and the parallelism of symbolic execution. Figure 3 shows the framework of MPISE.

There are two major components of our tool related to this paper: the hooked MPI library and the analyzer embedded in the symbolic execution engine. The former mainly includes our interceptions of MPI APIs for symbolic execution, while

Table I
EXPERIMENTAL RESULTS

Program	#proc	Error	#ins	LS checks		memIns time(s)	
				no opt	opt	no opt	opt
change-send-buffer	2	send	29990456	4322220	17960	3.25	3.11
vector-isend	2	send	30008052	4346419	42091	3.25	3.14
noerror-wait*	2	recv	30074743	4323203	18943	3.35	3.30
irecv-isend*	2	recv	30082632	4325064	20606	3.09	3.07
athena4.0*	2	recv	37285238	5559386	1249438	11.86	7.80
heat-errors	2	send	31373864	4411571	103350	4.96	3.30
IS*	16	recv	71232393	11117094	5384417	13.68	13.8

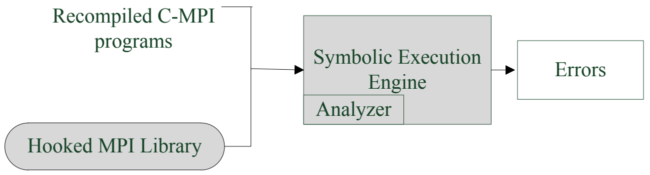

Figure 3. The framework of MPISE

the latter analyzes the usages of each buffer to see if there is a synchronization error according to Algorithm 2.

Same as our previous work [4], we also employ a multi-thread MPI library as the MPI "model" for symbolic execution. We use azequiaMPI [10], which is an MPI platform on which an MPI program is run in a multi-thread manner. An MPI program will be compiled into LLVM [11] bytecode first. Then, the generated bytecode will be linked with azequiaMPI library bytecode to form a multi-threaded version, which should generate the same result as that of running the MPI program in parallel. The path space of the multi-threaded version will be explored by the symbolic execution engine to detect synchronization errors. When a synchronization error is found, MPISE records all the information, including the input, the sequences of message passings, *etc*.

For the analyzer, when implementing the symbolic execution semantics of each instruction in an MPI program, we implement the synchronization error checking algorithm (*c.f.* Algorithm 2) at the related instructions, to analyze the usage of each message buffer.

When one feeds an MPI program to MPISE, he/she needs to tell MPISE the inputs that the analysis needs to cover (thanks to KLEE, one can also provide "symbolic arguments" to tell MPISE without code modifications). Then, the program will be executed in a specific number of processes. For each path during symbolic execution, the analyzer checks the existence of any synchronization error. At the same time, the runtime errors in the MPI programs, such as division by zero and array index out of bound, can also be detected.

Based on the prototype, we have conducted the experiments of typical MPI programs to justify the effectiveness. The results and discussions will be shown in the next Section.

V. EXPERIMENTAL RESULTS

During experiments, we manually marked some inputs as symbolic inputs. Table I shows the experimental results, where we use 7 programs to evaluate MPISE: (1) `change-send-buffer`, `vector-isend`, `noerror-wait` and `irecv-isend`, which are from Umpire [2]; (2) `athena` [12], which is an MPI application for astrophysical magneto hydrodynamics; (3) `heat-error` from [13], which implements the equation of heat conduction. (4) `IS` in NAS Parallel Benchmarks (NPB) [14]. In Table I, if a program is marked with '*', it indicates that the synchronization error in the program is injected. The reason of why we only choose IS from NPB benchmarks is that: `IS` and `DT` are the only two programs in NPB written in C, while `DT` has no non-blocking communications. The first 6 programs in Table I are run with 2 processes, and the last one with 16 processes. All the experiments were conducted on a Linux server with 32 cores and 250 GB memory.

As shown in Table I, MPISE can not only detect the four injected *input-related* synchronization errors successfully, but also the two in the test cases of Umpire. The second column shows the number of the processes run for each program. The third column shows the type of the buffer related to the detected synchronisation error. The column "# ins" shows the count of the executed instructions of each application. The next two columns in "LS checks" show the count of the checks when handling Load/Store instructions without (no opt) or with (opt) optimizations. The last two columns in "memIns time(s)" show the total time used by all the load/store instructions without (no opt) or with (opt) optimizations, which includes the time for checking synchronization errors. Hence, using optimizations, we can reduce the times of memory checkings significantly, which justifies the effectiveness of the first optimization. However, for the second optimization, *i.e.*, reducing map size, its effectiveness is not significant in the experiments, and the reason is that there are not many messages transferred in the analyzed programs. In addition, according to the execution time of load/store instructions in the last two columns, the optimizations do not have an impressive effect. The reason is the execution time is dominated by the execution of instructions. The saved memory checkings for detecting synchronization errors do not need much time.

For more large-scale experiments, we employ `IS` from NPB. Figure 4 and Figure 5 show the results of analyzing `IS` with different numbers of processes.

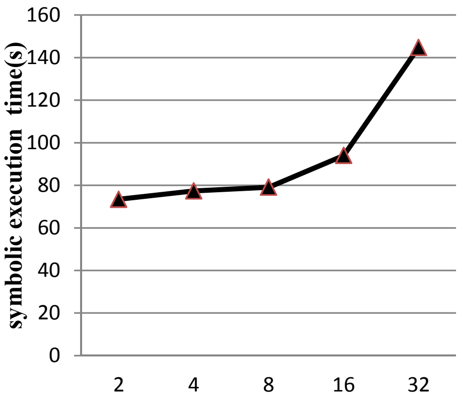

Figure 4. The time of analyzing NPB IS

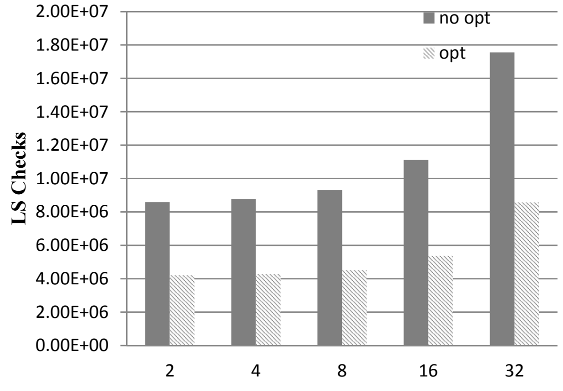

Figure 5. The number of the Load/Store checks in analyzing NPB IS

As shown in Figure 4, the analyze time does not increase exponentially with the increasing of the process number, which validates the effectiveness of the scheduling method we use. For the count of checks, the results in Figure 5 indicate that we can nearly reduce half of the checks by employing the optimizations.

Another interesting result is about the send synchronization error reported by the analysis of `heat-errors` program. After manual checking, we find that the buffer of the send operation related to the error will never be modified before being sent out. The reason is that the program uses non-blocking send and blocking receive operations to form an implicit synchronization, which has the same effect as that of a *Wait* operation, but does not use a *Wait* operation explicitly. To avoid reporting this kinds of "false" errors, we need to reason the program executions, which is leaved to be the future work. However, we also advice to avoid this kind of tricky program skills in asynchronous MPI programming.

VI. Related Work

To get MPI programs right, many techniques and tools has been proposed. Existing methods basically fall into the following three categories: static analysis, runtime checking and verification methods.

There is very few static analysis work for MPI programs. Greg proposes *parallel* control flow graph (pCFG) in [15] to capture the interaction behaviors of an MPI program with arbitrary number of processes. Hence, analysis activities can be carried out basing on pCFG, such as getting the communication topology of an MPI program. However, the static analysis based on pCFG is hard to be automated.

Another kind of tools, such as SyncChecker [1], Marmot [16], the Intel Trace Analyzer and Collector [17] and MUST [18], intercept the runtime MPI calls and record the running information of an MPI program. Then, using runtime information, runtime errors, synchronization errors, deadlocks or performance bottlenecks can be analyzed. These tools often need to recompile or relink MPI programs, and depend on the inputs and the scheduling of each running. Tools such as ISP [19], can handle the non-determinism caused by parallelism by controlling the schedules of an MPI program under a specific input. In [13], a SAT-based predicative method is proposed to improve the generation of the schedules for revealing the deadlock errors in MPI programs. Park *et al.* propose in [20] a sampling based dynamic analysis method for detecting data races in hybrid parallel programming. But still, input-related errors can be missed for these methods.

For verification tools, TASS [21] also tackles the problem by using symbolic execution to provide input coverage of an MPI program. A model of the program is first constructed, and the symbolic execution is done on the model to detect errors or verify the equivalence between a sequential program and its parallel version. TASS does not support the analysis of asynchronous MPI programs, thus it does not consider the detection of synchronization errors. In addition, as far as we know, the programs that can be analyzed by TASS are pretty small-scale, and only a subset of C syntax and restricted environment are supported. It is hard to apply TASS on real world MPI applications, such as NPB and athena in this paper.

VII. Conclusion and Future Work

Asynchronous MPI programming results synchronization errors frequently. Detecting such errors is hard due to the non-determinism caused by inputs and parallelism. In this paper, we propose a symbolic execution based method to detect the input-related synchronization errors in MPI programs. Basing on the symbolic execution of an MPI program, we can systematically check each path of the program with respect to the typestate property of message buffers. Two optimizations are also presented to improve the efficiency, especially when analyzing large MPI programs. We have implemented our analysis method in a prototype tool. The results of the experiments on representative MPI programs indicate the effectiveness and the efficiency of our method.

The future work lies in two aspects: first, until now, our analysis is not sound, since the receive from any operations under asynchronous communications are not handled systematically; second, the usages of our tool on more real world MPI applications will be appreciated.

Acknowledgements

This work is supported by the National 973 project 2014CB340703, and the National NSFC projects (Nos. 61120106006, 61272140 and 61472440). We would like to thank the authors of Cloud9 and KLEE for the wonderful platform. Especially Stefan Bucur, for his useful suggestions.

References

[1] Z. Chen, X. Li, J.-Y. Chen, H. Zhong, and F. Qin, "Syncchecker: Detecting synchronization errors between mpi applications and libraries," in *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '12. IEEE Computer Society, 2012, pp. 342–353.

[2] J. S. Vetter and B. R. de Supinski, "Dynamic software testing of mpi applications with umpire," in *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '00. Washington, DC, USA: IEEE Computer Society, 2000.

[3] 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*, ser. EuroSys '11. New York, NY, USA: ACM, 2011, pp. 183–198.

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

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

[6] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[7] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI '08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224.

[8] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Trans. Softw. Eng.*, vol. 12, no. 1, pp. 157–171, Jan. 1986.

[9] "Infiniband trade association," http://www.infinibandta.com.

[10] "AzequiaMPI site," gim.unex.es/azequiampi.

[11] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, ser. CGO '04, Palo Alto, California, Mar 2004.

[12] "Athena site," https://trac.princeton.edu/Athena.

[13] V. Forejt, D. Kroening, G. Narayanaswamy, and S. Sharma, "Precise predictive analysis for discovering communication deadlocks in MPI programs," in *Proceedings of the 19th International Symposium Formal Methods*, ser. FM '14, C. B. Jones, P. Pihlajasaari, and J. Sun, Eds., vol. 8442. Springer, 2014, pp. 263–278.

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

[15] G. Bronevetsky, "Communication-sensitive static dataflow for parallel message passing applications," in *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '09. IEEE Computer Society, 2009, pp. 1–12.

[16] 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.

[17] 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.

[18] 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 *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 30:1–30:11.

[19] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby, "Isp: A tool for model checking mpi programs," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 285–286.

[20] C.-S. Park, K. Sen, and C. Iancu, "Scaling data race detection for partitioned global address space programs," in *Proceedings of the 27th International Conference on Supercomputing*, ser. ICS '13. ACM, 2013, pp. 47–58.

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