Symbolic Verification of MPI Programs with Non-deterministic Synchronizations

Hengbiao Yu[1](✉), Zhenbang Chen[1](✉), Chun Huang[1], and Ji Wang[1,2]

[1] College of Computer, National University of Defense Technology, Changsha, China
{hengbiaoyu,zbchen,chunhuang,wj}@nudt.edu.cn
[2] State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha, China

Abstract. Message Passing Interface (MPI) is the current de-facto standard for developing applications in high-performance computing. MPI allows flexible implementations of message passing operations, which introduces non-deterministic synchronizations that challenge the correctness of MPI programs. We present in this paper a symbolic method for verifying the MPI programs with non-deterministic synchronizations. Insides the method, we propose a path-level modeling method that uses communicating sequential processes (CSP) to precisely encode the non-deterministic synchronizations of an execution path. Furthermore, for the execution paths without non-deterministic message receive operations, we propose an optimization method to reduce the complexity of the CSP models. We have implemented our technique on MPI-SV and evaluated it on 10 real-world MPI programs *w.r.t.* deadlock freedom. The experimental results demonstrate the effectiveness of our verification method.

1 Introduction

Message Passing Interface (MPI) [21] is the most widely used standard for developing applications in high-performance computing (HPC). MPI provides a rich set of *message passing* operations for developers. MPI programs are usually run in many processes spanned on network-connected machines. These distributed processes cooperate by message passings to accomplish a computation task. The development of MPI programs is challenging, and it is highly demanded to have methods and tools to ensure the correctness of MPI programs [8].

Existing approaches for verifying the correctness of MPI programs are mainly dynamic methods [6,24], which run the program under a specific input and verify the correctness of program paths. There are few static verification methods [2,18,19], which abstract the MPI program and verify the correctness of the abstract model. There are also symbolic execution based methods [14,27], which achieve a balance between precision and scalability and provide a bounded verification support for MPI programs.

This work is supported by National Key R&D Program of China (No. 2018YFB0204301) and NSFC Program (No. 61902409, 61632015, 61690203 and 61532007).

For improving the performance further, the MPI standard [7] allows flexible implementations of MPI operations. Especially, the standard send operation can be implemented in "*rendezvous*" mode [1], where the send operation blocks the sending process until the message is received, or the "*eager*" mode [1], where the send operation's completion is independent of the corresponding receive operation and only requires that the sending message has been copied to the local system buffer. If the local system buffer is full, the send operation also blocks. Hence, the synchronizations in MPI programs depend on the MPI implementations, which we call *non-deterministic synchronizations*. Because of this, an MPI program failure (*e.g.*, deadlock) may appear only in some specific MPI implementations, which brings difficulties for developing MPI programs. However, the existing work of verifying MPI programs seldom considers the problem of non-deterministic synchronizations. As far as we know, the work in [1] is the only one, which provides a dynamic verification method that can find potential errors related to non-deterministic synchronizations. However, the method in [1] cannot find the errors that depend on the program input.

Figure 1 shows an MPI program running in two processes. Process P_0 first sends a message to process P_1. Then, if the input x is equal to 'a', P_0 will receive a messages from P_1. On the other hand, if x is 'a', process P_1 will first send a message to P_0. After that, process P_1 receives a message from process P_0. If the input x is 'a' and both the send operations are implemented in the "rendezvous" mode, a deadlock happens, *i.e.*, both P_0 and P_1 block at the send operation and wait for receive operations to receive the message. However, if send operations are implemented in the "eager" model with non-zero-sized local system buffers, there will be no deadlock, despite the input x. Hence, to verify the deadlock freedom of the MPI program, we need to cover both the input space and the non-deterministic synchronizations.

P_0	P_1
`Send(1)`	**if** (x=='a')
if (x=='a')	`Send(0)`
`Recv(1)`	`Recv(0)`

Fig. 1. A motivating example.

In this paper, we present a verification method that covers both the program input and the non-deterministic synchronizations. Our approach is based on the symbolic verification framework of MPI-SV [27]. Specifically, we propose a precise modeling method that utilizes communicating sequential processes (CSP) [17] to encode the non-deterministic synchronizations. The key idea is to use an internal choice to model the blocking or non-blocking of each send operation. Besides, we propose an optimization method for the MPI execution paths without non-deterministic receive operations, which improves the verification's efficiency.

We have implemented our method on MPI-SV [27], and evaluated it on 10 real-world MPI programs, totaling 46638 lines of code (LOC), *w.r.t.* the deadlock freedom property. Our tool successfully verified all the 26 tasks within 90 min, and found that 18 verification tasks have deadlocks due to the non-deterministic synchronizations and 8 tasks are deadlock free. We manually confirmed that all the detected deadlocks are real. These experimental results demonstrate the effectiveness of our approach.

There are following main contributions of this paper.

- A precise method for modeling the non-deterministic synchronizations of an execution path in terms of CSP.
- An optimization method that can reduce the complexity of CSP models for paths having no non-deterministic receive operations.
- A prototype tool and an extensive evaluation on real-world MPI+C programs.

Related Work. There already exist some approaches for verifying MPI programs. MPI-SPIN [18,19] utilizes model checking [5] to verify MPI programs *w.r.t.* LTL properties. However, MPI-SPIN needs manual efforts to build a model in Promela [9]. ParTypes [13] integrates type checking and deductive verification to verify MPI programs against a protocol. ParTypes's verification results hold for any number of processes but may have false positives. Besides, ParTypes only supports MPI programs without non-blocking or wildcard operations. Dynamic verification approaches, *e.g.*, ISP [24] and DAMPI [25], execute the same input multiple times to cover the schedules. MOPPER [6] and the tool in [10] encode the deadlock detection problem under concrete inputs in a SAT and SMT equation, respectively. Hermes [11] integrates dynamic verification and symbolic analysis to verify *multi-path* MPI programs. All these dynamic verification approaches do not support input coverage. MPI-SV [27] integrates symbolic execution and model checking to verify MPI programs *w.r.t.* a given property, but does not consider non-deterministic synchronizations.

The closest related work is the method in [1], where a two-step verification framework is proposed for MPI programs with non-deterministic synchronizations. They first build an abstract model by supposing all the non-deterministic synchronous operations using the "eager" mode. A post-processing method is then applied to the model to detect the potentially missed deadlocks introduced by the non-deterministic synchronous operations. However, they only consider communications under a specific input and may miss input-related deadlocks. In contrast, our approach covers program inputs, different schedules, and non-deterministic synchronizations.

The rest of this paper is organized as follows. Section 2 briefly introduces MPI programs. Section 3 presents the verification framework. Section 4 details our CSP modeling method. Section 5 gives the implementation and evaluation. We conclude in Sect. 6.

$$\begin{array}{rl} \mathsf{Proc} ::= & \mathbf{var}\ l : \mathbf{T} \mid l := \mathsf{e} \mid \mathsf{Comm} \mid \mathsf{Proc}\ ;\ \mathsf{Proc} \mid \\ & \mathbf{if}\ \mathsf{e}\ \mathsf{Proc}\ \mathbf{else}\ \mathsf{Proc} \mid \mathbf{while}\ \mathsf{e}\ \mathbf{do}\ \mathsf{Proc} \\ \mathsf{Comm} ::= & \mathtt{Ssend(e)} \mid \mathtt{Send(e)} \mid \mathtt{Recv(e)} \mid \mathtt{Recv(*)} \mid \mathtt{Barrier} \mid \\ & \mathtt{ISend(e,r)} \mid \mathtt{IRecv(e,r)} \mid \mathtt{IRecv(*,r)} \mid \mathtt{Wait(r)} \\ & \mathtt{Bcast(e)} \mid \mathtt{Gather(e)} \mid \mathtt{Scatter(e)} \end{array}$$

Fig. 2. Syntax of a core MPI language.

2 MPI Programs

In this section, we first define a core MPI language, including syntax and semantics. Then, we give some MPI related definitions used throughout the remainder of this paper.

2.1 Syntax

MPI is a library of message passing functions that can be used to create parallel applications in different languages, such as C, C++, and Fortran. This paper targets MPI+C programs. We define an MPI program $\mathcal{MP}$ as a *finite* set of MPI processes $\{\mathsf{Proc}_i \mid 0 \leq i \leq n\}$, where there are $n+1$ processes, and each Proc_i is defined by the language in Fig. 2.

Figure 2 gives the syntax of a core MPI language considered in this paper, where $\mathbb{T}$ is a set of types, $\mathbb{N}$ is a set of names, and $\mathbb{E}$ denotes set of expressions. It is worth pointing out that we omit complex language features for brevity, *e.g.*, message related parameters of MPI operations, and pointer operations. Our tool does support real-world MPI+C programs.

The statement **var** l : $\mathbf{T}$ defines variable l with type $\mathbf{T}$ ($\mathbf{T} \in \mathbb{T}$), and statement $l := \mathsf{e}$ assigns the value of expression e ($\mathsf{e} \in \mathbb{E}$) to l. The statement Comm defines an MPI operation, including both blocking and non-blocking message passing operations. An MPI process can be constructed by composing the basic statements using sequence, condition, and loop composition operators.

2.2 Informal Semantics

We present an informal semantics for MPI operations. In terms of the number of evolved MPI processes, we divide the MPI operations into two main groups, *i.e.*, two-sided operations and collective operations.

We first explain the semantics of two-sided operations. The parameter `e` in two-sided MPI operations denotes the destination process's identifier, and the parameter `r` denotes the handler of a non-blocking send or receive operation. `Ssend(e)` is a blocking message send operation that blocks the process until its message has been received by the destination process e, while `ISend(e,r)` is a non-blocking message send operation that returns immediately after being issued. `Send(e)` also sends a message to process `e`, and blocks the process until its sending buffer can be reused. A message from process `e` can be received using

`Recv(e)` and `IRecv(e,r)`. `Recv(e)` will block the process until the message is well received, while `IRecv(e, r)` returns immediately after being issued. Note that `Recv(*)` and `IRecv(*,r)` are blocking and non-blocking wildcard receive operations, which can receive a message from any processes. `wait(r)` is used to ensure the completion of non-blocking operations, *i.e.*, it blocks the process until the non-blocking operation indicated by `r` is completed.

Collective operations include all the processes in the communicator. The parameter `e` represents the identifier of the root process. `Bcast(e)` means that process `e` broadcasts a message to the non-root processes, and the non-root processes are blocked until the messages are received. `Gather(e)` gathers data from all the processes to the root process `e`; hence, the root process is blocked until all the messages are received. `Scatter(e)` scatters the data from process `e` to all the processes, which will be blocked until the messages are received. `Barrier` will block the process until all the processes have called it.

2.3 Definitions

Given an MPI program $\mathcal{MP} = \{\mathsf{Proc}_i \mid 0 \leq i \leq n\}$, we define a global state S of $\mathcal{MP}$ as $(s_0, \ldots, s_n)$, where s_i is the local state of the ith process. The local state s_i is a 4-tuple $(\mathcal{M}, \mathit{Stat}, \mathit{Seq}_i, \mathcal{F})$, where $\mathcal{M}$ is a mapping from variables to values, Stat is the next program statement to be executed, Seq_i records the issued MPI operations of Proc_i, $\mathcal{F}$ is the flag of process status belonging to $\{\mathsf{active}, \mathsf{blocked}, \mathsf{terminated}\}$. An element elem of s_i can be accessed by $s_i.\mathit{elem}$. For a global state S, we use $\mathit{Seq}(S) = \{\mathit{Seq}_i \mid 0 \leq i \leq n\}$ to denote the issued MPI operations of S.

The formal semantics of the language in Fig. 2 can be defined based on the definitions of global and local states. In principle, the semantics of an MPI program is a communicating finite state machine [3] with different buffer sizes determined by the MPI implementations.

3 Symbolic Verification Framework

This section explains the symbolic verification framework based on MPI-SV [27]. Algorithm 1 gives the framework. In principle, this framework combines symbolic execution and model checking in a synergetic manner. The framework employs symbolic execution to extract path-level models from the MPI program. Then, the path-level models are verified by model checking. The results of model checking are also used to prune paths in symbolic execution.

The inputs of the framework are an MPI program and a verification property. The framework's skeleton is a worklist-based symbolic executor [4]. In the beginning, the *worklist* only contains the initial state S_{init}. Then, the framework iteratively selects a state from *worklist* to advance the state for symbolic execution until all the paths are explored or timeout (omitted for brevity). Select at Line 4 can use different search heuristics for state exploration, such as depth-first

Algorithm 1: Symbolic Verification Framework

```
SV-Framework($\mathcal{MP}, \varphi$)
Data: $\mathcal{MP}$ is an MPI program $\{\mathsf{Proc}_i \mid 0 \leq i \leq n\}$ and $\varphi$ is a property
begin
    $worklist \leftarrow \{S_{init}\}$
    while $worklist \neq \emptyset$ do
        $S_c \leftarrow$ Select($worklist$)
        $s_i \leftarrow$ Scheduler($S_c$)
        Execute($S_c, s_i, worklist$)
        if $\forall s_i \in S_c, s_i.\mathcal{F} =$ terminated then
            if containWild($S_c$) then
                $\Gamma \leftarrow$ GenerateCSP($S_c$)
            else
                $\Gamma \leftarrow$ GenerateCSP$^{\#}$($S_c$)
            end
            ModelCheck($\Gamma, \varphi$)
            if $\Gamma \models \varphi$ then
                Prune($worklist, S_c$)
            end
            else if $\Gamma \not\models \varphi$ then
                reportViolation and Exit
            end
        end
    end
end
```

search (DFS) and breadth-first search (BFS). For the state S_c, we select a process for symbolic execution. Scheduler at Line 5 selects the non-blocking process with the smallest rank. Then, Execute will symbolically execute the next statement in the selected process's state s_i, which may add new states to *worklist*. When each process terminates normally in the state S_c (Line 7), we build a CSP model Γ that encodes the equivalent states of S_c, *i.e.*, the states having the same path condition of S_c. Here, if there exists any wildcard receive along the path to S_c, we build the CSP model by `GenerateCSP` (*c.f.* Sect. 4.2); otherwise, we build an optimized model by `GenerateCSP`$^{\#}$ (*c.f.* Sect. 4.3). Then, we verify the CSP model Γ *w.r.t.* φ by model checking (Line 13). If Γ satisfies φ, we prune the equivalent states from *worklist*; otherwise, if the model checker gives a counter-example, we report the counter-example and exit.

Symbolic Execution. The symbolic execution step in the framework is the same as traditional symbolic execution [12] except for the message-passing operations. To get the possible matchings of wildcard receives, the framework executes the non-blocking processes as much as possible. The message operation matchings will be carried out when all the processes are blocked or terminated. The framework matches the message operations *w.r.t.* the happens-before requirements in the MPI standard [7]. When some message operations are matched, the framework will do the symbolic execution of these operations and continue to execute the processes that become active after matching. When a wildcard receive is matched with N send operations, the framework will fork the current state into N states to cover different cases.

P_0	P_1	P_2

`Ssend(1)`	`Recv(*)`
	`Recv(2)`
		`Ssend(1)`

Fig. 3. An illustrating example of MPI-SV.

Example. We use the example in Fig. 3 to show the workflow of symbolic verification. The symbolic executor first executes process P_0, and blocks at `Ssend(1)`. After that the symbolic executor executes P_1 and P_2 in sequence and blocks at `Recv(*)` and `Ssend(1)`, respectively. At this time, all the processes are blocked, symbolic verification handles the message matchings. Clearly, the wildcard receive `Recv(*)` has two matchings, *i.e.*, P_0's `Ssend(1)` and P_2's `Ssend(1)`. The symbolic executor forks two states for the two matchings. Suppose we first explore the state where `Recv(*)` matches P_0's `Ssend(1)`, process P_1 continues to be executed, and blocks at `Recv(2)`. Symbolic verification handles the message matchings again. After matching P_1's `Recv(2)` and P_2's `Ssend(1)`, all the processes are terminated, and the issued MPI operation sequences are $Seq_0 = \langle \texttt{Ssend(1)} \rangle$, $Seq_1 = \langle \texttt{Recv(*)}, \texttt{Recv(2)} \rangle$, and $Seq_2 = \langle \texttt{Ssend(1)} \rangle$. Now, for all the three processes, we generate the CSP processes for each of them and then compose them in parallel to get the CSP model Γ (*c.f.* Sect. 4.2). The model checking of Γ *w.r.t.* deadlock freedom reports a counter-example, *i.e.*, P_1's wildcard receive receives the message from P_2.

Discussion. Our framework integrates symbolic execution and model checking. Scheduler at Line 5 actually employs partial-order reduction (POR) [5] to reduce the full interleavings of different processes. Pure symbolic execution can only verify reachability properties [15]. However, leveraged by the synergy, the framework can verify a larger scope of properties, *i.e.*, temporal properties, because the CSP model encodes the interleavings of the message operations in different processes.

4 CSP Modeling of Non-deterministic Synchronization

4.1 CSP Subset

We utilize a subset of CSP to model an execution path's equivalent communication behaviors, *i.e.*, changing the matchings and interleavings of wildcard receive operations, and the implementations of the non-deterministic synchronizations.

Let Σ be a *finite* set of *events*, $\mathbb{C}$ be a set of *channels*, and $\mathbf{X}$ be a set of *variables*. Figure 4 gives the syntax of the CSP subset that we used, where P represents a CSP process, $a \in \Sigma$, $c \in \mathbb{C}$, $X \subseteq \Sigma$, $x \in \mathbf{X}$ and *cond* is a boolean expression.

$$P := a \mid P \fatsemi P \mid P \sqcap P \mid P \Box P \mid P \underset{X}{\|} P \mid c?x \rightarrow P \mid c!x \rightarrow P \mid [cond] \rightarrow P \mid \textbf{skip}$$

Fig. 4. The syntax of the CSP subset.

A CSP process can be a single event a or an empty process **skip** that terminates immediately. There exist five operators to compose complex processes, *i.e.*, sequential composition ($\fatsemi$), internal choice ($\sqcap$), external choice ($\Box$), parallel composition with synchronization ($\underset{X}{\|}$) and guarded composition($\rightarrow$). Process $P \fatsemi Q$ executes process P and Q in sequence. The choice of process P and Q executes P or Q, the selection of internal choice ($\sqcap$) is non-deterministic, while the selection of external choice ($\Box$) is made by the environment, *i.e.*, which process is first enabled to execute. $P \underset{X}{\|} Q$ executes the interleaving of P and Q but requires P and Q to synchronize on the events in X. Let PS be a finite set of processes, $\underset{X}{\|} PS$ denotes the parallel composition of all the processes in PS. The guarded composition executes the guard and the guarded process in sequence. Channel operation $c?x$ and $c!x$ represent reading an element from channel c to variable x and writing the value of x to channel c, respectively. The guard $[cond]$ makes the guarded process unable to be executed until the boolean condition *cond* is true. *cond* can be the boolean condition of variables and channels, *e.g.*, $\mathsf{cempty}(c)$ means that channel c is empty.

4.2 CSP Modeling

Algorithm 2 depicts the basic procedure of building a CSP model for a normally terminated path. The input is a normally terminated global state S for an MPI program running with $n+1$ processes, and $Seq(S)=\{Seq_i \mid 0 \le i \le n\}$ contains the recorded message passing operation sequences of the processes, *i.e.*, Seq_i is the sequence of the MPI process Proc_i.

The algorithm generates a CSP process for each MPI process and composes the CSP processes in parallel to construct the whole CSP model. For each MPI process, we use the generation rules defined in Fig. 5 to derive its CSP process P_i'. The generation is to scan the operation sequence backward. For each operation, we generate its CSP model and compose the model with the previously generated model. The basic idea of modeling non-deterministic synchronizations is to use an internal choice between the "rendezvous" mode and the "eager" mode with an infinite buffer for modeling message send operations.

For the standard send operation `MPI_Send`, we allocate a zero-sized channel c_0 and a one-sized channel c_1, and use an internal choice of the channel writings

Algorithm 2: CSP Model Generation for a Terminated State

```
   GenerateCSP($S$)
   Data: A terminated global state $S$, and $Seq(S)=\{Seq_i \mid 0 \le i \le n\}$
1  begin
2      $PS \leftarrow \emptyset$
3      $X \leftarrow \emptyset$
4      for $i \leftarrow 0 \ldots n$ do
5          $P_i \leftarrow$ skip
6          $(Seq_i, P_i, X)_i \rightarrow^* (\langle\rangle, P_i', X')_i$ using the rules in Figure 5
7          $X \leftarrow X'$
8          $PS \leftarrow PS \cup \{P_i'\}$
9      end
10     return $\underset{\{X\}}{\|} PS$
11 end
```

$$\frac{\mathcal{S} = \mathcal{S}' \circ \langle op \rangle \wedge op = \mathsf{Send(j)} \wedge (c_0, c_1) = \mathsf{Chans}(op)}{(\mathcal{S}, P, X)_i \rightarrow (\mathcal{S}', (c_0!x \rightarrow \textbf{skip} \sqcap c_1!x \rightarrow \textbf{skip}) \fatsemi P, X)_i} \quad (\mathsf{Send})$$

$$\frac{\mathcal{S} = \mathcal{S}' \circ \langle op \rangle \wedge op = \mathsf{Bcast(j)} \wedge i = j \wedge c_1 = \mathsf{Chan}(op)}{(\mathcal{S}, P, X)_i \rightarrow (\mathcal{S}', (c_1!x \rightarrow \textbf{skip} \sqcap \mathsf{Bcast}_k) \fatsemi P, X \cup \{\mathsf{Bcast}_k\})_i} \quad (\mathsf{Bcast\text{-}1})$$

$$\frac{\mathcal{S} = \mathcal{S}' \circ \langle op \rangle \wedge op = \mathsf{Bcast(j)} \wedge i \neq j \wedge c_1 = \mathsf{Chan}(op)}{(\mathcal{S}, P, X)_i \rightarrow (\mathcal{S}', ([!\mathsf{cempty}(c_1)] \rightarrow \textbf{skip} \Box \mathsf{Bcast}_k) \fatsemi P, X)_i} \quad (\mathsf{Bcast\text{-}2})$$

$$\frac{\mathcal{S} = \mathcal{S}' \circ \langle op \rangle \wedge op = \mathsf{Gather(j)} \wedge i = j}{(\mathcal{S}, P, X)_i \rightarrow (\mathcal{S}' \circ \langle \mathsf{Send(j)}, \mathsf{Recv(0)}, ..., \mathsf{Recv(n)} \rangle, P, X)_i} \quad (\mathsf{Gather\text{-}1})$$

$$\frac{\mathcal{S} = \mathcal{S}' \circ \langle op \rangle \wedge op = \mathsf{Gather(j)} \wedge i \neq j}{(\mathcal{S}, P, X)_i \rightarrow (\mathcal{S}' \circ \langle \mathsf{Send(j)} \rangle, P, X)_i} \quad (\mathsf{Gather\text{-}2})$$

$$\frac{\mathcal{S} = \mathcal{S}' \circ \langle op \rangle \wedge op = \mathsf{Scatter(j)} \wedge i = j}{(\mathcal{S}, P, X)_i \rightarrow (\mathcal{S}' \circ \langle \mathsf{Send(0)}, ..., \mathsf{Send(n)}, \mathsf{Recv(j)} \rangle, P, X)_i} \quad (\mathsf{Scatter\text{-}1})$$

$$\frac{\mathcal{S} = \mathcal{S}' \circ \langle op \rangle \wedge op = \mathsf{Scatter(j)} \wedge i \neq j}{(\mathcal{S}, P, X)_i \rightarrow (\mathcal{S}' \circ \langle \mathsf{Recv(j)} \rangle, P, X)_i} \quad (\mathsf{Scatter\text{-}2})$$

Fig. 5. CSP Model Construction Rules. $\mathcal{S}_0 \circ \mathcal{S}_1$ represents the concatenation of two MPI operation sequences. In $(\mathcal{S}, P, X)_i$, $\mathcal{S}$ is the current operation sequence, P is the currently generated CSP model, X is the set of the generated synchronization events, and i denotes the ith process Proc_i. $\mathsf{Chans}(op)$ returns a zero-sized channel and a one-sized channel. $\mathsf{Chan}(op)$ returns a one-sized channel. Besides, $\rightarrow^*$ represents applying the rules zero or multiple times.

to model it (*c.f.*, Rule Send). *Each send operation has its channels.* Hence, each send operation can finish immediately or wait for the message to be received. The internal choice models the non-determinism between the "rendezvous" mode and the "eager" mode with an infinite buffer.

For the collective broadcast operation `MPI_Bcast`, we model it as follows: for the root process (*c.f.*, Rule Bcast-1), we allocate a one-sized channel and use internal choice of a one-sized channel writing and a synchronization event (Bcast_k[1]) to model it; while for the non-root processes (*c.f.*, Rule Bcast-2), we use an external choice of a guarded process and the synchronization event process Bcast_k, where cempty is a boolean function to test whether a channel is empty or not. The external choice selects to execute the guarded process only if the root process has written the one-sized channel.

For other collective operations, *e.g.*, `MPI_Gather` and `MPI_Scatter`, we transform them into a sequence of `MPI_Send` and `MPI_Recv` operations, which preserves the semantics of these collective operations. For the remaining MPI operations, the modeling methods are the same as MPI-SV [27], which are omitted for brevity.

Example. Let's go back to the program in Fig. 1, even though there exist no wildcard operations, we build each terminated path a CSP model. For the false branch $x \neq$ 'a', process P_0 simply sends a message to P_1 using a standard send operation, and process P_1 receives a message from P_0. The issued MPI operation sequences of P_0 and P_1 are $Seq_0 = \langle \texttt{Send(1)} \rangle$ and $Seq_1 = \langle \texttt{Recv(0)} \rangle$, respectively. To model different implementations of the standard send operation, we use internal choice ($\sqcap$) to compose the "rendezvous" mode (corresponding to a zero-sized channel writing) and the "eager" mode that has infinite buffer (corresponding to a one-sized channel writing). Hence, we generate CSP_0 for process P_0, where c_0 is a zero-sized channel and c_1 is a one-sized channel.

$$CSP_0 := c_0!0 \rightarrow \mathbf{skip} \sqcap c_1!1 \rightarrow \mathbf{skip}$$

On the other hand, to model the receive operation, we use external choice ($\Box$) to compose the two possible matched channel reading operations, *i.e.*, generating CSP_1 for process P_1.

$$CSP_1 := c_0?x \rightarrow \mathbf{skip} \Box c_1?x \rightarrow \mathbf{skip}$$

We compose the two CSP models in parallel, *i.e.*, $CSP_0 \parallel CSP_1$, and verify that the model is deadlock free. The result is that the model satisfies the property.

Similarly, for the true branch $x ==$ 'a', the issued MPI operation sequences of P_0 and P_1 are $Seq_0 = \langle \texttt{Send(1)}, \texttt{Recv(1)} \rangle$ and $Seq_1 = \langle \texttt{Send(0)}, \texttt{Recv(0)} \rangle$, respectively. We generate CSP_0' for process P_0, and CSP_1' for process P_1, where c_0 and c_2 are zero-sized channels, c_1 and c_3 are one-sized channels.

$$CSP_0' := (c_0!0 \rightarrow \mathbf{skip} \sqcap c_1!1 \rightarrow \mathbf{skip}) \fatsemi (c_2?x \rightarrow \mathbf{skip} \Box c_3?x \rightarrow \mathbf{skip})$$

$$CSP_1' := (c_2!2 \rightarrow \mathbf{skip} \sqcap c_3!3 \rightarrow \mathbf{skip}) \fatsemi (c_0?x \rightarrow \mathbf{skip} \Box c_1?x \rightarrow \mathbf{skip})$$

When verifying the CSP model $CSP_0' \parallel CSP_1'$ *w.r.t.* deadlock freedom, the model checker gives a counter-example, where both the internal choices select to

[1]We allocate each group of MPI_Bcast operations a unique synchronization event Bcast_k.

write an element to the zero-sized channels, *i.e.*, c_0 and c_2. Hence, our approach detects the deadlock related to non-deterministic synchronizations.

Our CSP modeling method ensures that there will be no deadlock for the "eager" mode with an any-sized finite buffer if the generated model is deadlock-free. We prove this result in Theorem 1, where $\mathsf{CSP_i}(p)$ represents the CSP model built by Algorithm 2 for a normally terminated path p, and $\mathsf{CSP_b}(p)$ denotes the CSP model built by having a finite buffer in "eager" mode.

Theorem 1. *If* $\mathsf{CSP_i}(p)$ *is deadlock free,* $\mathsf{CSP_b}(p)$ *is deadlock free.*

Proof. The key idea of the proof is to use the refinement relation of CSP in stable-failures semantics [17]. The stable-failures semantic model of a process P is $(\mathcal{T}(P), \mathcal{F}(P))$, where $\mathcal{T}(P)$ contains the traces of P, and $\mathcal{F}(P)$ contains the elements formed as (s, X), which represents that P refuses to execute any event in X after executing the trace s.

The only different between $\mathsf{CSP_i}(p)$ and $\mathsf{CSP_b}(p)$ is the modeling of send operations. For $\mathsf{CSP_i}(p)$, any send operation can block or finish immediately, due to the infinite buffer. However, for $\mathsf{CSP_b}(p)$, a send operation's behavior depends on the size of the buffer when it selects "eager" mode. If the buffer is full, the send operation blocks; otherwise, it can finish immediately. Hence, for a send operation *op*, if $M_i(op)$ and $M_b(op)$ represent the models built by Algorithm 2 with infinite and finite buffers, respectively, we have $\mathcal{F}(M_i(op)) \supseteq \mathcal{F}(M_b(op))$ and $\mathcal{T}(M_i(op)) \supseteq \mathcal{T}(M_b(op))$. Because the modeling of all the remaining operations is same between $\mathsf{CSP_i}(p)$ and $\mathsf{CSP_b}(p)$, we can have $\mathcal{F}(\mathsf{CSP_i}(p))) \supseteq \mathcal{F}(\mathsf{CSP_b}(p))$ and $\mathcal{T}(\mathsf{CSP_i}(p))) \supseteq \mathcal{T}(\mathsf{CSP_b}(p))$.

Hence, we can prove that $\mathsf{CSP_b}(p)$ is a stable-failures refinement of $\mathsf{CSP_i}(p)$, which implies the theorem.

4.3 Optimization

For the execution paths without wildcard receives, we can optimize the CSP model construction by only considering the "rendezvous" mode. The intuition is that a model with more synchronizations tends to have a deadlock. The correctness of the optimization is ensured by Theorem 2, where $\mathsf{CSP_r}(p)$ represents the CSP model built by Algorithm 2 for a normally terminated path p considering only "rendezvous" mode.

Theorem 2. *Given an execution path p along which there are no wildcard receive operations,* $\mathsf{CSP_r}(p)$ *is deadlock free if and only if* $\mathsf{CSP_i}(p)$ *is deadlock free.*

Proof. The CSP modeling also complies with the happens-before requirements of the MPI standard [7]. Especially, the *non-overtaken rules* [24] are the ones that motivate the optimization. The rules require that:

- if there exist two send operations of a process that send messages to the same process and both can match a receive operation, the receive operation should receive the message of the first issued send operation;

– if a process has two receive operations that can match the same send operation, the send operation's message should be received by the first issued receive operation.

As a result, the message operation matchings are deterministic for the paths having no wildcard receive operations, despite the implementations of send operations.

With respect to the semantic definition [17] of internal choice, we can have that $\mathsf{CSP}_\mathsf{r}(p)$ is a stable-failure refinement of $\mathsf{CSP}_\mathsf{i}(p)$. Hence, if $\mathsf{CSP}_\mathsf{i}(p)$ is deadlock free, $\mathsf{CSP}_\mathsf{r}(p)$ is deadlock free. Next, we prove that the deadlock freedom of $\mathsf{CSP}_\mathsf{r}(p)$ implies that $\mathsf{CSP}_\mathsf{i}(p)$ is deadlock free by contradiction. Suppose $\mathsf{CSP}_\mathsf{r}(p)$ is deadlock free but there exists a deadlock in $\mathsf{CSP}_\mathsf{i}(p)$, and the deadlock happens after executing the trace s_d, *i.e.*, $\mathsf{CSP}_\mathsf{i}(p)$ refuses to execute any event after executing s_d. Due to the non-overtaken rules and the assumption of no wildcard receives, the message matchings are deterministic. Hence, $\mathsf{CSP}_\mathsf{r}(p)$ can also executes s_d. Besides, the deadlock of $\mathsf{CSP}_\mathsf{i}(p)$ means that $\mathsf{CSP}_\mathsf{r}(p)$ cannot execute any event either. So, $\mathsf{CSP}_\mathsf{r}(p)$ also deadlocks, which contradicts with the assumption.

In summary, the theorem holds.

According to Theorem 2, when an execution path has no wildcard receive operations, we only need to treat the send operations as blocking operations for the verification of deadlock freedom. Such optimization can significantly reduce the complexity of CSP models because the state space of the original CSP model increases exponentially *w.r.t.* the number of send operations. We omit the optimization from the construction rules for brevity.

Example. We still take the program in Fig. 1 for example. For the terminated path p of the true branch $x ==$ 'a', the issued MPI operations of process P_0 and P_1 are $Seq_0 = \langle \texttt{Send(1)}, \texttt{Recv(1)} \rangle$ and $Seq_1 = \langle \texttt{Send(0)}, \texttt{Recv(0)} \rangle$, respectively. Since p has no wildcard receive operations, we use the optimization method to build the CSP model. We build the following two CSP processes, *i.e.*, CSP_0 and CSP_1, where channel c_0 and c_1 are zero-sized channels.

$$CSP_0 := c_0!x \rightarrow c_1?x \rightarrow \textbf{skip}$$

$$CSP_1 := c_1!x \rightarrow c_0?x \rightarrow \textbf{skip}$$

When verifying the optimized model $CSP_0 \parallel CSP_1$ *w.r.t.* deadlock freedom, we can successfully detect the deadlock, because the standard send operations are implemented in "rendezvous" mode. Compared with the CSP model generated by the rules in Fig. 5, the optimized model is much simpler.

5 Experimental Evaluation

In this section, we first introduce the implementation of our approach. Then, we give the experimental setup. Finally, we present the experimental results.

5.1 Implementation

We have implemented our approach on MPI-SV [27], whose basic procedure is to compile an MPI+C program to LLVM bytecode using Clang, and then link it with a pre-compiled multi-threaded MPI library AzequiaMPI [16] to generate the input for verification. We implemented our path-level CSP modeling method as a module within the symbolic execution engine, which will be invoked if the generated path has wildcard receive operations or non-deterministic blocking operations. We adopt the state-of-the-art CSP model checker PAT [22] to verify the path-level CSP models *w.r.t.* deadlock freedom.

5.2 Setup

Table 1 lists the programs analyzed in our experiments. All the programs are real-world open source MPI programs. `DTG` is a testing program from a PhD dissertation [23]. `Integrate_mw` and `Diffusion2d` come from the FEVS benchmark suite [20]. `Integrate_mw`[2] calculates the integrals of trigonometric functions, and `Diffusion2d` is a parallel solver for two-dimensional diffusion equation. `Gauss_elim` is an MPI implementation for gaussian elimination used in [26]. We downloaded `Pingpong`, `Mandelbrot`, and `Image_manip` from github. `Pingpong` is a testing program for communication performance. `Mandelbrot` parallel draws the mandelbrot set for a bitmap. `Image_manip` is an MPI program for image manipulations, *e.g.*, shifting, rotating and scaling. The remaining three programs are large parallel applications. `Depsolver` is a parallel multi-material 3D electrostatic solver, `Kfray` is a ray tracing program that can create realistic images, and `ClustalW` is a popular tool for aligning multiple gene sequences.

We experiment on a laptop with 8G memory and 2.0 GHz cores. The operating system is Ubuntu 14.04. We set the time threshold as 90 min for each verification task. The conducted experiments are to answer the following two questions:

- Effectiveness: Can our tool effectively verify real-world MPI programs having non-deterministic synchronizations *w.r.t.* deadlock freedom?
- Optimization: Can the optimization for modeling the paths without wildcard receives reduce the cost of CSP model checking?

5.3 Experimental Results

Table 2 lists the verification results for the programs with different numbers of processes. To evaluate our approach, we verify each program under 6, 8, and 10 processes. The first column **Program** shows the program names. Column **#i** ($i \in \{6, 8, 10\}$) indicates the number of running processes. A verification task consists of a program and the number of running processes. Column **Deadlock** indicates whether a task is deadlock-free, where **no** denotes that our tool successfully

[2] `Integrat_mw` is adopted from [6], in which a static schedule is employed.

Table 1. The programs in the experiments.

Program	LOC	Brief description
DTG	90	Dependence transition group
Integrate_mw	181	Integral computing
Diffusion2d	197	Simulation of diffusion equation
Gauss_elim	341	Gaussian elimination
Pingpong	220	Comm performance testing
Mandelbrot	268	Mandelbrot set drawing
Image_manip	360	Image manipulation
DepSolver	8988	Multimaterial electrostatic solver
Kfray	12728	KF-Ray parallel raytracer
ClustalW	23265	Multiple sequence alignment
Total	**46638**	**10 open source programs**

verified that the program is deadlock-free under the number of processes, and **yes** denotes that a deadlock exists. The column **Time(s)** gives the verification time. Table 3 lists the results for DTG and Pingpong that are developed under a fixed number of processes, where column **#Procs** gives the number of processes.

Table 2. Results for programs with variable number of processes.

Program	Deadlock			Time(s)		
	#6	#8	#10	#6	#8	#10
Integrate_mw	**No**	**No**	**No**	10.8	54.1	3783.4
Diffusion2d	**Yes**	**Yes**	**Yes**	20.3	12.9	13.1
Gauss_elim	**Yes**	**Yes**	**Yes**	16.1	21.1	28.6
Mandelbrot	**Yes**	**Yes**	**Yes**	11.7	12.6	14.1
Image_mani	**Yes**	**Yes**	**Yes**	11.6	13.3	15.4
Depsolver	**Yes**	**Yes**	**Yes**	127.9	204.8	322.8
Kfray	**Yes**	**Yes**	**Yes**	46.1	51.3	52.7
Clustalw	**No**	**No**	**No**	68.3	154.1	3651.2

Table 3. Results for programs with fixed number of processes.

Program	# Procs	Deadlock	Time(s)
DTG	5	**No**	7.8
Pingpong	2	**No**	343.8

To the best of our knowledge, there exist no verification tools that can cover both the inputs and the schedules of the MPI program with non-deterministic synchronizations. Hence, we evaluate our tool directly on real-world programs. Our tool successfully verified all the tasks within 90 min, *i.e.*, deadlock for 18 tasks, and no deadlock in 8 tasks. We manually confirmed that the detected deadlocks are real and are caused by the non-deterministic synchronizations. Even for the large tasks, *e.g.*, the last three programs running in 10 processes, Our tool can successfully verify them, demonstrating the scalability of our technique.

We evaluate the effectiveness of our optimization on the programs having no wildcard receive operations, *i.e.*, `DTG*`[3] and `Pingpong`. Table 4 gives the detailed results. The first column **PATH** is the label for the execution paths of symbolic execution, and the suffix of the program indicates the execution path's index. Column **Time(s)** gives the time consumption for verifying the corresponding CSP models and column **#States** gives the explored states in the CSP model. **CSP** and **CSP**$^{\#}$ represent the default modeling method and the optimized modeling method, respectively. Column **Speedup** shows the speedups of optimization. In terms of time consumption for model checking the path-level CSP models, **CSP**$^{\#}$ achieves an average 3x speedup. On the other hand, the number of states that need to be explored is reduced significantly by the optimization, *i.e.*, **CSP**$^{\#}$ explores at least 2.4x fewer states than **CSP**. These results indicate that our optimization for the paths without wildcard receive operations can effectively reduce the verification complexity of path-level CSP models.

Table 4. Results for optimization.

PATH	Time(s)			#States		
	CSP	CSP$^{\#}$	Speedup	CSP	CSP$^{\#}$	Speedup
DTG_1^*	13.8	7.9	1.8	386	32	12.1
$\mathsf{Pingpong}_1$	11.5	10.1	1.1	133	55	2.4
$\mathsf{Pingpong}_2$	88.2	16.5	5.3	2413	967	2.5
$\mathsf{Pingpong}_3$	63.2	14.7	4.3	1813	727	2.5
$\mathsf{Pingpong}_4$	37.5	14.3	2.6	1213	487	2.5

6 Conclusion and Future Work

This paper has presented an approach for verifying MPI programs with non-deterministic synchronization features. We enhance the symbolic verification by proposing a precise method for modeling the non-deterministic synchronizations of an execution path in terms of CSP. To improve the scalability, for the execution paths without wildcard receive operations, we give an optimization to reduce the complexity of CSP models. We have implemented our approach as a

[3]`DTG*` is the version that replaces the wildcard receives by deterministic receives.

prototype tool and extensively evaluated it on real-world MPI+C programs. The experimental results demonstrate the effectiveness of our approach. Our future work mainly includes two directions: (1) reducing the complexity of CSP models for paths having wildcard receive operations; and (2) applying our tool to verify temporal safety properties related to non-deterministic synchronizations.

References

1. Böhm, S., Meca, O., Jančar, P.: State-space reduction of non-deterministically synchronizing systems applicable to deadlock detection in MPI. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 102–118. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_7
2. Botbol, V., Chailloux, E., Le Gall, T.: Static analysis of communicating processes using symbolic transducers. In: Bouajjani, A., Monniaux, D. (eds.) VMCAI 2017. LNCS, vol. 10145, pp. 73–90. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-52234-0_5
3. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM **30**, 323–342 (1983)
4. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, pp. 209–224 (2008)
5. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
6. Forejt, V., Kroening, D., Narayanaswamy, G., Sharma, S.: Precise predictive analysis for discovering communication deadlocks in MPI programs. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 263–278. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06410-9_19
7. MPI Forum: MPI: a message-passing interface standard version 3.0 (2012). http://mpi-forum.org
8. Gopalakrishnan, G., et al.: Report of the HPC correctness summit, 25–26 January 2017, Washington, DC (2017). https://science.energy.gov/~/media/ascr/pdf/programdocuments/docs/2017/HPC_Correctness_Report.pdf
9. Holzmann, G.J.: Promela manual pages (2012). http://spinroot.com/spin/Man/promela.html
10. Huang, Yu., Mercer, E.: Detecting MPI zero buffer incompatibility by SMT encoding. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 219–233. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17524-9_16
11. Khanna, D., Sharma, S., Rodríguez, C., Purandare, R.: Dynamic symbolic verification of MPI programs. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E. (eds.) FM 2018. LNCS, vol. 10951, pp. 466–484. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95582-7_28
12. King, J.: Symbolic execution and program testing. Commun. ACM **19**, 385–394 (1976)
13. López, H.A., et al.: Protocol-based verification of message-passing parallel programs. In: OOPSLA, pp. 280–298 (2015)
14. Luo, Z., Zheng, M., Siegel, S.F.: Verification of MPI programs using CIVL. In: EuroMPI, pp. 6:1–6:11 (2017)
15. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems - Specification. Springer, New York (1992). https://doi.org/10.1007/978-1-4612-0931-7

16. Rico-Gallego, J.-A., Díaz-Martín, J.-C.: Performance evaluation of thread-based MPI in shared memory. In: Cotronis, Y., Danalis, A., Nikolopoulos, D.S., Dongarra, J. (eds.) EuroMPI 2011. LNCS, vol. 6960, pp. 337–338. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24449-0_42
17. Roscoe, B.: The Theory and Practice of Concurrency. Prentice-Hall, Upper Saddle River (2005)
18. Siegel, S.F.: Model checking nonblocking MPI programs. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 44–58. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-69738-1_3
19. Siegel, S.F.: Verifying parallel programs with MPI-Spin. In: Cappello, F., Herault, T., Dongarra, J. (eds.) EuroPVM/MPI 2007. LNCS, vol. 4757, pp. 13–14. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75416-9_8
20. Siegel, S.F., Zirkel, T.K.: FEVS: a functional equivalence verification suite for high-performance scientific computing. Math. Comput. Sci. **5**, 427–435 (2011)
21. Snir, M.: MPI-The Complete Reference: The MPI Core, vol. 1. MIT Press, Cambridge (1998)
22. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: towards flexible verification under fairness. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 709–714. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_59
23. Vakkalanka, S.: Efficient dynamic verification algorithms for MPI applications. Ph.D. thesis, The University of Utah (2010)
24. Vakkalanka, S., Gopalakrishnan, G., Kirby, R.M.: Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 66–79. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_9
25. Vo, A., Aananthakrishnan, S., Gopalakrishnan, G., De Supinski, B.R., Schulz, M., Bronevetsky, G.: A scalable and distributed dynamic formal verifier for MPI programs. In: SC, pp. 1–10 (2010)
26. Xue, R., et al.: MPIWiz: subgroup reproducible replay of MPI applications. ACM SIGPLAN Not. **44**, 251–260 (2009)
27. Yu, H., et al.: Symbolic verification of message passing interface programs. In: 42nd International Conference on Software Engineering, ICSE 2020, Seoul, South Korea, 27 June–19 July 2020, pp. 1248–1260 (2020)