Synchronization Error Detection of MPI Programs by Symbolic Execution

Xianjin Fu, Zhenbang Chen, Chun Huang, Wei Dong, and Ji Wang
zbchen@nudt.edu.cn

College of Computer
National University of Defense Technology
Changsha, China
2014.12.02
High Performance Computing

• MPI is widely used for developing HPC applications

• MPI programs are not easy to develop and maintain
MPI Paradigm

• MPI implements a message-passing based parallel programming style
MPI Paradigm

- MPI implements a message-passing based parallel programming style
- Frequently used optimization trick
  - Overlapping of communication and computation
- Asynchronous communication
Synchronization Error

- A buffer is written/read before asynchronously sent out/received
Synchronization Error

• A buffer is written/read before asynchronously sent out/received

Sending Example

```
ISend(..., buff, req)
buff[0] = 1
Wait(req)
```
Synchronization Error

- A buffer is **written/read** before **asynchronously sent out/received**

Sending Example

```
ISend(..., buff, req)
buff[0] = 1
Wait(req)
```

Receiving Example

```
IRrecv(..., buff, req)
c = buff[0]
Wait(req)
```
Synchronization Error

- A buffer is written/read before asynchronously sent out/received
- Incorrect computation and results
- Crash the MPI application
Synchronization Error

- A buffer is written/read before asynchronously sent out/received
- Incorrect computation and results
- Crash the MPI application

*How to detect synchronization errors in MPI programs?*
Existing Approaches

- Dynamic methods
  - Runtime checking: SyncChecker[IPDPS’12], UMPIRE[SC’00], ... 
- Input coverage & Non-determinism
- No static methods
- False alarms
Problem

- How to detect *input-related* synchronization error detection precisely?

\[ \text{P}_0 \]

\[
\text{I} \text{Send}(\text{P}_1, \textbf{buff}, \text{req}) \\
\text{Wait}(\text{req})
\]

\[ \text{P}_1 \]

\[
\text{I} \text{Recv}(\text{P}_0, \textbf{buff}, \text{req}) \\
\text{if} (!X) \; \text{c} = \textbf{buff}[0] \\
\text{Wait}(\text{req})
\]
Problem

- How to detect *input-related* synchronization error detection precisely?

\[ \text{ISend}(P_1, \text{buff}, \text{req}) \]
\[ \text{Wait}(\text{req}) \]
\[ P_0 \]

\[ \text{IRrecv}(P_0, \text{buff}, \text{req}) \]
\[ \text{if} \ (\neg X) \ c = \text{buff}[0] \]
\[ \text{Wait}(\text{req}) \]
\[ P_1 \]

*Symbolic execution based detection*
Symbolic Execution

• A SAT/SMT based program analysis method
  • Execute a program with symbolic values
  • Convert a program into path conditions
  • A precise method
• Usages
  • Test generation, bug finding, etc.
Symbolic Execution

int main(int i, j) {
    if (i > 0) {
        i = i + j
    } else {
        i = i - j
    }
    return i
}
Symbolic Execution

```c
int main(int i, j) {
    if (i > 0) {
        i = i + j
    } else {
        i = i - j
    }
    return i
}
```

Symbolic Calculation

- `i, j ← x_i, x_j`
  - PC: true
- `i, j ← x_i, x_j`
  - PC: `x_i > 0`
- `i, j ← x_i + x_j, x_j`
  - PC: `x_i > 0`
int main(int i, j) {
    if (i > 0) {
        i = i + j
    } else {
        i = i - j
    }
    return i
}

i, j ← \mathbf{x}_i, \mathbf{x}_j
PC: true

Program end
Symbolic Execution

```c
int main(int i, j) {
    if (i > 0) {
        i = i + j
    } else {
        i = i - j
    }
    return i
}
```
Key Idea

- Use symbolic execution to ensure input coverage
- Track the state transition of each transferred buffer
Motivation Example

P₀

\textit{ISend}(P₁, buff₀, req)
Wait(req)

P₁

\textit{IRcv}(P₀, buff₁, req)
if (!X) c = buff₁[0]
Wait(req)

\(x \leftarrow x_i\)
PC: true

Thursday, December 25, 14
Motivation Example

\[ \text{P}_0 \]
\[ \text{ISend(P}_1, \text{buff}_0, \text{req}) \]
\[ \text{Wait(req)} \]

\[ \text{P}_1 \]
\[ \text{IRrecv(P}_0, \text{buff}_1, \text{req}) \]
\[ \text{if (!X) c = buff}_1[0] \]
\[ \text{Wait(req)} \]

\[ x \leftarrow x_i \]
\[ \text{PC: true} \]

\[ x \leftarrow x_i \]
\[ \text{PC: true} \]

\[ \text{buff}_0 \]

\[ \text{PC: true} \]

\[ \text{Init} \rightarrow \text{TRANS} \rightarrow \text{Ready} \rightarrow \text{CHK} \rightarrow \text{Final} \]

\[ \text{USE} \]

\[ \text{USE} \]
Motivation Example

\begin{align*}
\text{P}_0 & \quad \text{ISend}(\text{P}_1, \text{buff}_0, \text{req}) \\
& \quad \text{Wait}(\text{req}) \\
\text{P}_1 & \quad \text{IRecv}(\text{P}_0, \text{buff}_1, \text{req}) \\
& \quad \text{if } (!\text{X}) \text{ c } = \text{buff}_1[0] \\
& \quad \text{Wait}(\text{req})
\end{align*}

\begin{align*}
\text{x } & \leftarrow \text{x}_i \\
\text{PC: true}
\end{align*}
Motivation Example

P_0

ISend(P_1, \textit{buff}_0, \textit{req})

Wait(\textit{req})

P_1

IRcv(P_0, \textit{buff}_1, \textit{req})

if (!X) c = \textit{buff}_1[0]

Wait(\textit{req})

\texttt{x} \leftarrow \texttt{x}_i

PC: true

\texttt{x} \leftarrow \texttt{x}_i

PC: true

\textit{buff}_1
Motivation Example

$P_0$

$\text{Send}(P_1, \textit{buff}_0, \textit{req})$

$\text{Wait(}\textit{req})$

$P_1$

$\text{Recv}(P_0, \textit{buff}_1, \textit{req})$

$\text{if } (!X) \text{ c = buff}_1[0]$

$\text{Wait(}\textit{req})$

$x \leftarrow x_i$

$\text{PC: true}$

$x \leftarrow x_i$

$\text{PC: true}$

$x \leftarrow x_i$

$\text{PC: } x_i \neq 0$

$\textit{buff}_1$
Motivation Example

**P₀**

\( \text{ISend}(P₀, \text{buff}_0, \text{req}) \)

\( \text{Wait}(\text{req}) \)

**P₁**

\( \text{IRcv}(P₀, \text{buff}_1, \text{req}) \)

if \((!X)\) \(c = \text{buff}_1[0]\)

\( \text{Wait}(\text{req}) \)

\[ \begin{align*}
   & x \leftarrow x_i \\
   & \text{PC: true}
\end{align*} \]

\[ \begin{align*}
   & x \leftarrow x_i \\
   & \text{PC: true}
\end{align*} \]

\[ x \leftarrow x_i \]

\[ \text{PC: } x_i \neq 0 \]

\( \text{buff}_1 \)
Motivation Example

P₀

ISend(P₁, buff₀, req)
Wait(req)

P₁

IRecv(P₀, buff₁, req)
if (!X) c = buff₁[0]
Wait(req)

Thursday, December 25, 14
Motivation Example

P₀

| lSend(P₁, buff₀, req) | Wait(req) |

P₁

| lRecv(P₀, buff₁, req) | if (!X) c = buff₁[0] | Wait(req) |

A synchronization error is detected
Internals of Our Method

• Symbolic execution framework
• Fixed number of processes
• A round-robin style schedule
• A process is preempted when being blocked
• Synchronization error checking as a dynamic typestate analysis
Two Optimizations

• Optimization 1
  • Only track the buffers used on application level

• Optimization 2
  • Remove the buffer from checking list when its state reaches the final state
Implementation

- Cloud9 based implementation
- A multi-thread MPI library (*azequiaMPI*) as the environment model for MPI
# Experiments

- MPI Programs

<table>
<thead>
<tr>
<th>Programs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>change-send-buffer</td>
<td>Programs from Umpire benchmark</td>
</tr>
<tr>
<td>vector-isend</td>
<td></td>
</tr>
<tr>
<td>noerror-wait</td>
<td></td>
</tr>
<tr>
<td>irecv-isend</td>
<td></td>
</tr>
<tr>
<td>athena 4.0</td>
<td>Astrophysical magneto hydrodynamics</td>
</tr>
<tr>
<td>heat-errors</td>
<td>The equation of heat conduction</td>
</tr>
<tr>
<td>IS</td>
<td>Integer sort from NPB</td>
</tr>
<tr>
<td>Program</td>
<td>#proc</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>change-send-buffer</td>
<td>2</td>
</tr>
<tr>
<td>vector-isend</td>
<td>2</td>
</tr>
<tr>
<td>noerror-wait*</td>
<td>2</td>
</tr>
<tr>
<td>irecv-isend*</td>
<td>2</td>
</tr>
<tr>
<td>athena4.0*</td>
<td>2</td>
</tr>
<tr>
<td>heat-errors</td>
<td>2</td>
</tr>
<tr>
<td>IS*</td>
<td>16</td>
</tr>
</tbody>
</table>

During experiments, we manually marked some input-related synchronization errors. Hence, using optimizations, we can reduce the times of memory checkings significantly, which justifies the effectiveness of the first optimization. However, the second optimization, i.e., reducing map size, its effect is the execution time is dominated by the execution semantics of each instruction in an MPI program, so the optimizations do not have an impressive effect. The reason is that there are not many messages transferred in the program. The reason is the execution time is dominated by the execution time of load/store instructions in the last two columns, which are the only two programs in NPB written in C, and they have no non-blocking communications. The first 6 programs are from NAS Parallel Benchmarks (NPB) [14]. In the next two columns in “LS checks” show the count of the checks when handling Load/Store instructions related to the detected synchronization error. The column “#proc” shows the number of the processes run for each program. The third column shows the type of the buffer usage of each message buffer.

As shown in Table I, MPISE can not only detect the four synchronization errors successfully, but also the two in the test cases of Umpire. The second column in “LS checks” shows the execution time of load/store instructions in the last two columns, which are from [13], which implements the equation of heat conductivity in astrophysical magneto hydrodynamics; (3) heat-errors, which are from Umpire [2]; (2) vector-isend, which implements the equation of heat conductivity in astrophysical magneto hydrodynamics; (3) heat-errors; (4) irecv-isend*, which is a synchronization error according to Algorithm 2 at the related instructions, to analyze the passing of information, including the input, the sequences of message passings, and the usage of each message buffer. (c.f. [4]).

Same as our previous work [4], we also employ a multi-threaded version of azequiaMPI library bytecode to form a multi-threaded application. The next two columns in “LS checks” show the count of the checks when handling Load/Store instructions related to the detected synchronization error. The column “#proc” shows the number of the processes run for each program. The third column shows the type of the buffer usage of each message buffer. For each path during symbolic execution, the analyzer checks the existence of any synchronization error. When one feeds an MPI program to MPISE, he/she runs the MPI program in parallel. The path space of programs changes constantly. 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 bytecode. 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, and the usage of each message buffer.

Table I, if a program is marked with '*', it indicates that the synchronization error in the program is injected. The reason is the execution time is dominated by the execution time of load/store instructions in the last two columns, which are the only two programs in NPB written in C, and they have no non-blocking communications. The first 6 programs are from NAS Parallel Benchmarks (NPB) [14]. In the next two columns in “LS checks” show the count of the checks when handling Load/Store instructions related to the detected synchronization error. The column “#proc” shows the number of the processes run for each program. The third column shows the type of the buffer usage of each message buffer.

As shown in Table I, MPISE can not only detect the four synchronization errors successfully, but also the two in the test cases of Umpire. The second column in “LS checks” shows the execution time of load/store instructions in the last two columns, which are from [13], which implements the equation of heat conductivity in astrophysical magneto hydrodynamics; (3) heat-errors; (4) irecv-isend*, which is a synchronization error according to Algorithm 2 at the related instructions, to analyze the passing of information, including the input, the sequences of message passings, and the usage of each message buffer. (c.f. [4]).

Same as our previous work [4], we also employ a multi-threaded version of azequiaMPI library bytecode to form a multi-threaded application. The next two columns in “LS checks” show the count of the checks when handling Load/Store instructions related to the detected synchronization error. The column “#proc” shows the number of the processes run for each program. The third column shows the type of the buffer usage of each message buffer. For each path during symbolic execution, the analyzer checks the existence of any synchronization error. When one feeds an MPI program to MPISE, he/she runs the MPI program in parallel. The path space of programs changes constantly. 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 bytecode. 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, and the usage of each message buffer.

Table I, if a program is marked with '*', it indicates that the synchronization error in the program is injected. The reason is the execution time is dominated by the execution time of load/store instructions in the last two columns, which are the only two programs in NPB written in C, and they have no non-blocking communications. The first 6 programs are from NAS Parallel Benchmarks (NPB) [14]. In the next two columns in “LS checks” show the count of the checks when handling Load/Store instructions related to the detected synchronization error. The column “#proc” shows the number of the processes run for each program. The third column shows the type of the buffer usage of each message buffer.
Results (2/2)

- Analyzing NPB IS with different numbers of processes
Conclusion

- A symbolic execution based method for detecting synchronization errors in MPI programs
- Two optimizations
- A prototype and the experiments on real-world MPI programs
Work in progress

- Sound method for analyzing asynchronous MPI programs
- Applications on more real-world MPI programs
- Improvements and optimizations on tool
Thank you!

Q&A