Detecting MPI Usage Anomalies via Partial Program Symbolic Execution

Fangke Ye, Jisheng Zhao, Vivek Sarkar

KLEE Workshop 2021
Background

• Message Passing Interface (MPI) is a widely-used programming model for distributed-memory parallelism

• MPI programming is error-prone
  • MPI APIs are not expressed as structured program constructs
    • Easy to omit synchronizations for nonblocking communication API calls
  • Pointer aliasing and arithmetic on data buffers for communication for MPI applications written in C/C++
    • General difficulties in parallel programming

• Need tools to help debugging
MPI Usage Anomalies

• True bugs or uncommon coding styles that may lead to bugs

• Anomalies we target
  1. Buffer Type Mismatch
  2. Buffer Data Race
  3. Request Overwriting
  4. Unmatched Wait or Test
  5. Unmatched Point-to-Point (P2P) Call
MPI Usage Anomalies

- True bugs or uncommon coding styles that may lead to bugs

- Anomalies we target
  1. Buffer Type Mismatch
  2. Buffer Data Race
  3. Request Overwriting
  4. Unmatched Wait or Test
  5. Unmatched Point-to-Point (P2P) Call
MPI Usage Anomalies

- True bugs or uncommon coding styles that may lead to bugs

- Anomalies we target
  1. Buffer Type Mismatch
  2. Buffer Data Race
  3. Request Overwriting
  4. Unmatched Wait or Test
  5. Unmatched Point-to-Point (P2P) Call
MPI Usage Anomalies

- True bugs or uncommon coding styles that may lead to bugs

- Anomalies we target
  1. Buffer Type Mismatch
  2. Buffer Data Race
  3. Request Overwriting
  4. Unmatched Wait or Test
  5. Unmatched Point-to-Point (P2P) Call

```c
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...
if (rank == 0) {
    ...
    MPI_Send(sendbuf, 10, MPI_INT, 1, ...);
    ...
} else { // rank == 1
    ...
    MPI_Recv(recvbuf, 10, MPI_INT, 1, ...);
    ...
}
```
MPI Usage Anomalies

- True bugs or uncommon coding styles that may lead to bugs

- Anomalies we target
  1. Buffer Type Mismatch
  2. Buffer Data Race
  3. Request Overwriting
  4. Unmatched Wait or Test
  5. Unmatched Point-to-Point (P2P) Call

```c
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...
if (rank == 0) {
    ...
    MPI_Send(sendbuf, 10, MPI_INT, 1, ...);
    ...
} else { // rank == 1
    ...
    MPI_Recv(recvbuf, 10, MPI_INT, 1, ...);
    ...
}
```
MPI Usage Anomalies

- True bugs or uncommon coding styles that may lead to bugs

- Anomalies we target
  1. Buffer Type Mismatch
  2. Buffer Data Race
  3. Request Overwriting
  4. Unmatched Wait or Test
  5. Unmatched Point-to-Point (P2P) Call
Symbolic Execution for MPI Debugging

• Advantages
  • Precise reasoning of pointers in C/C++
  • Potential better coverage than debugging methods relying on a fixed set of concrete input
  • Better time and space efficiency than dynamically debugging for large-scale parallel programs

• Challenges
  • Modeling MPI API behaviors
  • Scalability
Our Approach

- **C/C++ MPI Application Source Code**
- **Compilation**
  - **Clang**
- **LLVM IR**
  - **LLVM IR + Static Program Information**
  - **Static Analysis Passes**

- **MPI Anomaly Detector**
  - **Intercept**
  - **KLEE**
- **Partial Program Symbolic Execution**
MPI Modeling – Ranks (Process Identifiers)

• Use a symbolic rank
• Fork the execution when rank is tested
  • Add a constraint on rank to the path condition
• Fill receive buffers with unconstrained symbolic values

```c
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
    MPI_Isend(sendbuf, 10, MPI_INT, 1, 0,
              MPI_COMM_WORLD, &req);
    MPI_Test(&req, &flag, MPI_STATUS_IGNORE);
    if (!flag)
        MPI_Wait(&req, MPI_STATUS_IGNORE);
}
```
MPI Modeling – Nonblocking Operations

• Record ongoing nonblocking operations in the state
• Intercept MPI calls to update the records

```c
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
    MPI_Isend(sendbuf, 10, MPI_INT, 1, 0, MPI_COMM_WORLD, &req);
    MPI_Test(&req, &flag, MPI_STATUS_IGNORE);
    if (!flag) MPI_Wait(&req, MPI_STATUS_IGNORE);
}
```
Anomaly Detection

- For anomalies related to nonblocking operations
  - Including Buffer Data Race, Request Overwriting, Unmatched Wait or Test
  - Use nonblocking operation records stored in the execution state
    - Check if the anomaly condition is satisfiable given the path condition

```
MPI_Irecv(recvbuf, 10, MPI_INT, x, 101, MPI_COMM_WORLD, &req[0]);
recvbuf[i] = 0;
```

Path condition: $pc < \text{lrecv, &req[0], recvbuf[40]>}$

If $(pc \land \text{recvbuf} \leq \&\text{recvbuf}[i] < \text{(char *)recvbuf + 40})$ is satisfiable
Anomaly Detection

• For other anomalies
  • Including Buffer Type Mismatch, Unmatched P2P Call
  • Use pre-computed static information to detect them
    • A map from MPI_Datatype to LLVM types
    • All P2P calls that are control dependent on each rank-related branch

```c
uint16_t recvbuf[100];
MPI_Irecv(recvbuf, 10, MPI_LONG, x,
101, MPI_COMM_WORLD, &req[0]);
```

```c
int16_t Int16Ty
int16_t
```

```c
MPI_LONG Int32Ty
```

```c
```
Improving the Scalability of Symbolic Execution

• Observation
  • Usually, a large portion of code in an MPI application is not relevant to communication

• Set limitations on execution
  • Max number of iterations per loop
  • Max fork depth

• Partial program symbolic execution
  • Start new executions at user-specified locations
  • Our implementation: select any function as the entrance of an execution
Memory State Initialization

• The memory state is unknown before entering the entry function
• Lazy initialization \(^1,^2\)
  • Allocate memory at dereferences
  • Fork the execution for multiple possible memory states
    • Eliminate some impossible states using pre-computed whole-program alias analysis results

```c
int a[10];

void f(int *p, int i) {
    p[i] = 0;
}

void g() {
    int b[10];
    f(b, 0);
}

void h() {
    f(a, 0);
}
```

---


Tracking Initialization States for Symbolic Pointers with Shadow Memory

• Not every pointer dereference needs lazy initialization
  • \*p should not trigger lazy initialization

• Use shadow memory to track whether a symbolic pointer value has already been initialized

```c
int f(int **m) {
    int *p = m[0];
    int *q = m[0];
    *q = 0;
    return *p;
}
```
Lazy Initialization with Shadow Memory

**Code**

```c
int f(int **m) {
    int *p = m[0];
    int *q = m[0];
    *q = 0;
    return *p;
}
```

**Application Memory & Local Variables**

- Each pointer-sized memory block has a corresponding shadow memory slot.
- Each local variable has a reference to a metadata cell.

**Shadow Memory**

- Each shadow memory slot stores a reference to a metadata cell.

**Metadata**

- Each metadata cell stores a boolean value that indicates whether lazy allocation is needed.
Lazy Initialization with Shadow Memory

```c
int f(int **m) {
    int *p = m[0];
    int *q = m[0];
    *q = 0;
    return *p;
}
```
Lazy Initialization with Shadow Memory

```c
int f(int **m) {
    int *p = m[0];
    int *q = m[0];
    *q = 0;
    return *p;
}
```
Lazy Initialization with Shadow Memory

```
int f(int **m) {
    int *p = m[0];
    int *q = m[0];
    *q = 0;
    return *p;
}
```
Lazy Initialization with Shadow Memory

Code

```c
int f(int **m) {
    int *p = m[0];
    int *q = m[0];
    *q = 0;
    return *p;
}
```

Application Memory & Local Variables

Shadow Memory

Metadata
Lazy Initialization with Shadow Memory

```c
int f(int **m) {
    int *p = m[0];
    int *q = m[0];
    *q = 0;
    return *p;
}
```
Lazy Initialization with Shadow Memory

int f(int **m) {
    int *p = m[0];
    int *q = m[0];
    *q = 0;
    return *p;
}
Evaluation

• Benchmarks
  • 2 real-world applications: AMG2013, Athena
  • 2 Benchmarks from the NAS Parallel Benchmark Suite: NPB.IS, NPB.DT
  • A library implemented with MPI: OpenFFT
  • A benchmark application used in previous work: Sort

• Comparison
  • Our approach: partial program symbolic execution (PSE)
  • Static analysis tool: MPI-Checker
  • Dynamic analysis tool: MUST

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMG2013</td>
<td>74,901</td>
</tr>
<tr>
<td>Athena</td>
<td>63,012</td>
</tr>
<tr>
<td>NPB.IS</td>
<td>6,498</td>
</tr>
<tr>
<td>NPB.DT</td>
<td>711</td>
</tr>
<tr>
<td>OpenFFT</td>
<td>892</td>
</tr>
<tr>
<td>Sort</td>
<td>127</td>
</tr>
</tbody>
</table>

2 Alexander Droste, Michael Kuhn, and Thomas Ludwig. MPI-Checker: Static Analysis for MPI. LLVM ’15.
3 Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. MPI Runtime Error Detection with MUST: Advances in Deadlock Detection. SC ’12.
## Evaluation – Effectiveness

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Anomaly Type</th>
<th>Number of Anomalies</th>
<th>Number of Anomalies Reported</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMG2013</td>
<td>Request Overwriting</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Unmatched Wait</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Athena</td>
<td>Buffer Data Race</td>
<td>4</td>
<td>4 (TP)</td>
</tr>
<tr>
<td></td>
<td>Request Overwriting</td>
<td>8</td>
<td>8 (TP)</td>
</tr>
<tr>
<td>Sort</td>
<td>Buffer Data Race</td>
<td>1</td>
<td>1 (TP)</td>
</tr>
</tbody>
</table>

**Number of Anomalies Reported** for PSE, MPI-Checker, and MUST.

- **TP** = True Positive
- **FP** = False Positive
- **N/A** = Not Supported
## Evaluation – Performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time (s)</th>
<th>Memory Usage (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PSE</td>
<td>MPI-Checker</td>
</tr>
<tr>
<td>AMG2013</td>
<td>28.57*</td>
<td>96.70</td>
</tr>
<tr>
<td>Athena</td>
<td>1960.14*</td>
<td>27.13</td>
</tr>
<tr>
<td>OpenFFT</td>
<td>75.21*</td>
<td>4.76</td>
</tr>
<tr>
<td>NPB.IS</td>
<td>206.21**</td>
<td>1.00</td>
</tr>
<tr>
<td>NPB.DT</td>
<td>9.54**</td>
<td>1.60</td>
</tr>
<tr>
<td>Sort</td>
<td>0.11**</td>
<td>0.39</td>
</tr>
</tbody>
</table>

* Starting from a non-main function.
** Starting from the main function with concrete input and symbolic ranks.
Conclusion

C/C++ MPI Application Source Code → Clang → LLVM IR

MPI Anomaly Detector

Intercept

KLEE

Partial Program Symbolic Execution

LLVM IR + Static Program Information → Static Analysis Passes

Source code available at
https://github.com/fkye/PSE-MPI