Session 2: MUST
Correctness Checking

Dr. Matthias S. Müller  (RWTH Aachen University)
Tobias Hilbrich   (Technische Universität Dresden)
Joachim Protze (RWTH Aachen University, LLNL)

Email:
mueller@itc.rwth-aachen.de
tobias.hilbrich@tu-dresden.de
protze@itc.rwth-aachen.de
Content

- Motivation
- MPI usage errors
- Examples: Common MPI usage errors
  - Including MUST’s error descriptions
- Correctness tools
- MUST usage
How many errors can you spot in this tiny example?

```
#include <mpi.h>
#include <stdio.h>

int main (int argc, char** argv)
{
    int rank, size, buf[8];

    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    MPI_Datatype type;
    MPI_Type_contiguous (2, MPI_INTEGER, &type);

    MPI_Recv (buf, 2, MPI_INT, size - rank, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    MPI_Send (buf, 2, type, size - rank, 123, MPI_COMM_WORLD);

    printf("Hello, I am rank %d of %d.
", rank, size);

    return 0;
}
```

At least 8 issues in this code example
Content

- Motivation
- MPI usage errors
- Examples: Common MPI usage errors
  - Including MUST’s error descriptions
- Correctness tools
- MUST usage
MPI usage errors

- MPI programming is error prone
- Bugs may manifest as:
  - Crashes
  - Hangs
  - Wrong results
  - Not at all! (Sleeping bugs)
- Tools help to detect these issues
Complications in MPI usage:

- Non-blocking communication
- Persistent communication
- Complex collectives (e.g. Alltoallw)
- Derived datatypes
- Non-contiguous buffers

Error Classes include:

- Incorrect arguments
- Resource errors
- Buffer usage
- Type matching
- Deadlocks
Content

- Motivation
- MPI usage errors
- Examples: Common MPI usage errors
  - Including MUST’s error descriptions
- Correctness tools
- MUST usage
Skipping some errors

- **Missing MPI_Init:**
  - Current release doesn’t start to work, implementation in progress

- **Missing MPI_Finalize:**
  - Current release doesn’t terminate all analyses, work in progress

- **Src/dest rank out of range (size-rank):** leads to crash, use crash save version of tool
MUST: Tool design

Application

MPI Library

Split-comm-world

Force status wrapper

GTI: event forwarding, network

PanMPI Library

Local Analyses

Non-Local Analyses

Rewrite-comm-world

MPI Library

Additional tool ranks
GTI: event forwarding, network

Local Analyses  Non-Local Analyses

0  1  2  3  4  5  6
#include <mpi.h>
#include <stdio.h>

int main (int argc, char** argv)
{
    int rank, size, buf[8];

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    MPI_Datatype type;
    MPI_Type_contiguous (2, MPI_INTEGER, &type);

    MPI_Recv (buf, 2, MPI_INT, size - rank - 1, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send (buf, 2, type, size - rank - 1, 123, MPI_COMM_WORLD);

    printf ("Hello, I am rank %d of %d.
", rank, size);

    MPI_Finalize ();

    return 0;
}
Must detects deadlocks

<table>
<thead>
<tr>
<th>Rank(s)</th>
<th>Type</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td></td>
<td>The application issued a set of MPI calls that can cause a deadlock! A graphical representation of this situation is available in a <a href="MUST_Output-files/MUST_Deadlock.html">detailed deadlock view</a>. References 1-2 list the involved calls (limited to the first 5 calls, further calls may be involved). The application still runs, if the deadlock manifested (e.g. caused a hang on this MPI implementation) you can attach to the involved ranks with a debugger or abort the application (if necessary).</td>
</tr>
</tbody>
</table>

References:
- [MUST Output-file](file:///home/pj416018/MUST/example/MUST_Output.html)
- [MUST Output-files/MUST_Deadlock.html](MUST_Output-files/MUST_Deadlock.html)

- Reference 1 rank 0: MPIRecv (1st occurrence) called from: #0 main@example.c:15
- Reference 2 rank 1: MPIRecv (1st occurrence) called from: #0 main@example.c:15

Click for graphical representation of the detected deadlock situation.
Graphical representation of deadlocks

<table>
<thead>
<tr>
<th>Comm:</th>
<th>MPI COMM WORLD</th>
</tr>
</thead>
</table>

**Message**

The application issued a set of MPI calls that can cause a deadlock! The graphs below show details on this situation. This includes a wait-for graph that shows active wait-for dependencies between the processes that cause the deadlock. Note that this process set only includes processes that cause the deadlock and no further processes. A legend details the wait-for graph components in addition, while a parallel call stack view summarizes the locations of the MPI calls that cause the deadlock. Below these graphs, a message queue graph shows active and unmatched point-to-point communications. This graph only includes operations that could have been intended to match a point-to-point operation that is relevant to the deadlock situation. Finally, a parallel call stack shows the locations of any operation in the parallel call stack. The leaves of this call stack graph show the components of the message queue graph that they span. The application still runs, if the deadlock manifested (e.g. caused a hang on this MPI implementation) you can attach to the involved ranks with a debugger or abort the application (if necessary).

**Wait-for Graph**

Rank 0 waits for rank 1 and vv.

**Call Stack**

Simple call stack for this example.
Fix1: use asynchronous receive

```c
#include <mpi.h>
#include <stdio.h>

int main (int argc, char** argv)
{
    int rank, size, buf[8];

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    MPI_Datatype type;
    MPI_Type_contiguous (2, MPI_INTEGER, &type);

    MPI_Request request;
    MPI_Irecv (buf, 2, MPI_INT, size - rank - 1, 123, MPI_COMM_WORLD, &request);

    MPI_Send (buf, 2, type, size - rank - 1, 123, MPI_COMM_WORLD);

    printf ("Hello, I am rank %d of %d.\n", rank, size);

    MPI_Finalize ();
    return 0;
}
```

Use asynchronous receive: (MPI_Irecv)
MUST detects errors in handling datatypes

<table>
<thead>
<tr>
<th>Rank(s)</th>
<th>Type</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Error</td>
<td>A receive operation uses a (datatype,count) pair that can not hold the data transferred by the send it matches! The first element of the send that did not fit into the receive operation is at (contiguous)<a href="MPI_INTEGER">0</a> in the send type (consult the MUST manual for a detailed description of datatype positions). The send operation was started at reference 1, the receive operation was started at reference 2. Information on communicator: MPI_COMM_WORLD) Information on send of count 2 with type:Datatype created at reference 3 is for Fortran, based on the following type(s): { MPI_INTEGER}) (Information on receive of count 2 with type:MPI_INT)</td>
</tr>
<tr>
<td>0-1</td>
<td>Error</td>
<td>Argument 3 (datatype) is not commited for transfer, call MPI_Type_commit before using the type for transfer! (Information on datatypeDatatype created at reference 1 is for Fortran, based on the following type(s): { MPI_INTEGER})</td>
</tr>
<tr>
<td>0</td>
<td>Error</td>
<td>The memory regions to be transfered by this send operation overlap with regions spanned by a pending non-blocking receive operation! (Information on the request associated with the other communication: Request activated at reference 1) (Information on the datatype associated with the other communication: MPI_INT) The other communication overlaps with this communication at position:(MPI_INT) (Information on the datatype associated with this communication: Datatype created at reference 2 is for Fortran, based on the following type(s): { MPI_INTEGER}) This communication overlaps with the other communication at position:(contiguous)<a href="MPI_INTEGER">0</a> A graphical representation of this situation is available in a detailed overlap view (MUST_Output_files/MUST_Overlap_0_0.html)</td>
</tr>
<tr>
<td>1</td>
<td>Error</td>
<td>The memory regions to be transfered by this send operation overlap with regions spanned by a pending non-blocking receive operation! (Information on the request associated with the other communication: Request activated at reference 1) (Information on the datatype associated with the other communication: MPI_INT) The other communication overlaps with this communication at position:(MPI_INT) (Information on the datatype associated with this communication: Datatype created at reference 2 is for Fortran, based on the following type(s): { MPI_INTEGER}) This communication overlaps with the other communication at position:(contiguous)<a href="MPI_INTEGER">0</a> A graphical representation of this situation is available in a detailed overlap view (MUST_Output_files/MUST_Overlap_1_0.html)</td>
</tr>
</tbody>
</table>

Use of uncommited datatype: `type`
Fix2: use MPI_Type_commit

```c
#include <mpi.h>
#include <stdio.h>

int main (int argc, char** argv)
{
    int rank, size, buf[8];

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    MPI_Datatype type;
    MPI_Type_contiguous (2, MPI_INTEGER, &type);
    MPI_Type_commit (&type);

    MPI_Request request;
    MPI_Irecv (buf, 2, MPI_INT, size - rank - 1, 123, MPI_COMM_WORLD, &request);

    MPI_Send (buf, 2, type, size - rank - 1, 123, MPI_COMM_WORLD);

    printf ("Hello, I am rank %d of %d.\n", rank, size);

    MPI_Finalize ();

    return 0;
}
```

Commit the datatype before usage
MUST detects errors in transfer buffer sizes / types

<table>
<thead>
<tr>
<th>Rank(s)</th>
<th>Type</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Error</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Error</td>
<td></td>
</tr>
</tbody>
</table>

A receive operation uses a (datatype,count) pair that can not hold the data transferred by the send it matches! The first element of the send that did not fit into the receive operation is at (contiguous)[0]([MPI_INTEGER]) in the send type (consult the MUST manual for a detailed description of datatype positions). The send operation was started at reference 1, the receive operation was started at reference 2. (Information on communicator: MPI_COMM_WORLD) (Information on send of count 2 with type:Datatype created at reference 3 is for Fortran, committed at reference 4, based on the following type(s): {MPI_INTEGER}) (Information on receive of count 2 with type:MPI_INT)

The memory regions to be transferred by this send operation overlap with regions spanned by a pending non-blocking receive operation!

(Information on the request associated with the other communication: Request activated at reference 1) (Information on the datatype associated with the other communication:)

Size of sent message larger than receive buffer
Fix3: use same message size for send and receive

```c
#include <mpi.h>
#include <stdio.h>

int main (int argc, char** argv)
{
    int rank, size, buf[8];

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    MPI_Datatype type;
    MPI_Type_contiguous (2, MPI_INTEGER, &type);
    MPI_Type_commit (&type);

    MPI_Request request;
    MPI_Irecv (buf, 2, MPI_INT, size - rank - 1, 123, MPI_COMM_WORLD, &request);

    MPI_Send (buf, 1, type, size - rank - 1, 123, MPI_COMM_WORLD);

    printf ("Hello, I am rank %d of %d.\n", rank, size);

    MPI_Finalize ();

    return 0;
}
```

Reduce the message size
MUST detects use of wrong argument values

Use of Fortran type in C, datatype mismatch between sender and receiver

<table>
<thead>
<tr>
<th>Rank(s)</th>
<th>Type</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Error</td>
<td>A send and a receive operation use datatypes that do not match! Mismatch occurs at (contiguous) and (MPI_INT) in the send type and at (MPI_INT) in the receive type (consult the MUST manual for a detailed description of datatype positions). A graphical representation of this situation is available in a detailed type mismatch view (MUST_Output files/MUST_Typemismatch.html). The send operation was started at reference 1, the receive operation was started at reference 2. (Information on communicator: MPI_COMM_WORLD) (Information on send of count 1 with type:Datatype created at reference 3 is for Fortran, committed at reference 4, based on the following type(s): { MPI_INTEGER}) (Information on receive of count 2 with type:MPI_INT)</td>
</tr>
<tr>
<td>0</td>
<td>Error</td>
<td>A send and a receive operation use datatypes that do not match! Mismatch occurs at (contiguous) and (MPI_INT) in the send type and at (MPI_INT) in the receive type (consult the MUST manual for a detailed description of datatype positions). A graphical representation of this situation is available in a detailed type mismatch view (MUST_Output files/MUST_Typemismatch_0.html). The send operation was started at reference 1, the receive operation was started at reference 2. (Information on communicator: MPI_COMM_WORLD) (Information on send of count 1 with type:Datatype created at reference 3 is for Fortran, committed at reference 4, based on the following type(s): { MPI_INTEGER}) (Information on receive of count 2 with type:MPI_INT)</td>
</tr>
</tbody>
</table>

The memory regions to be transferred by this send operation overlap with regions spanned by a pending non-blocking receive operation!
#include <mpi.h>
#include <stdio.h>

int main (int argc, char** argv)
{
  int rank, size, buf[8];

  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &size);

  MPI_Datatype type;
  MPI_Type_contiguous (2, MPI_INT, &type);
  MPI_Type_commit (&type);

  MPI_Request request;
  MPI_Irecv (buf, 2, MPI_INT, size - rank - 1, 123, MPI_COMM_WORLD, &request);

  MPI_Send (buf, 1, type, size - rank - 1, 123, MPI_COMM_WORLD);

  printf("Hello, I am rank %d of %d.\n", rank, size);

  MPI_Finalize ();

  return 0;
}
MUST detects data races in asynchronous communication

<table>
<thead>
<tr>
<th>Rank(s)</th>
<th>Type</th>
<th>Message</th>
<th>References of a representative process:</th>
</tr>
</thead>
</table>
| 1       | Error | The memory regions to be transferred by this send operation overlap with regions spanned by a pending non-blocking receive operation!  
  
    (Information on the request associated with the other communication: Request activated at reference 1)  
    (Information on the datatype associated with the other communication: MPI_INT)  
    The other communication overlaps with this communication at position:(MPI_INT)  
    (Information on the datatype associated with this communication: Datatype created at reference 2 is for C, committed at reference 3, based on the following type(s): { MPI_INT })  
    This communication overlaps with the other communication at position:(contiguous)[0](MPI_INT)  
    A graphical representation of this situation is available in a detailed overlap view (MUSTOutputsfiles/MUST_Overlap_1_0.html). | reference 1 rank 1: MPI_Irecv (1st occurrence) called from:  
    #0 main@example-fix4.c:17  
    reference 2 rank 1: MPI_Type_contiguous (1st occurrence) called from:  
    #0 main@example-fix4.c:13  
    reference 3 rank 1: MPI_Type_commit (1st occurrence) called from:  
    #0 main@example-fix4.c:14 |
| 0-1     | Error | There are 1 datatypes that are not freed when MPI_Finalize was issued, a quality application should free all MPI resources before calling MPI_Finalize. Listing information for these datatypes:  
  
    - Datatype 1: Datatype created at reference 1 is for C, committed at reference 2, based on the following type(s): { MPI_INT }  
  
  | Representatives of a representative process:  
    reference 1 rank 1: MPI_Type_contiguous (1st occurrence) called from:  
    #0 main@example-fix4.c:13  
    reference 2 rank 1: MPI_Type_commit (1st occurrence) called from:  
    #0 main@example-fix4.c:14 |
| 0-1     | Error | There are 1 requests that are not freed when MPI_Finalize was issued, a quality application should free all MPI resources before calling MPI_Finalize. Listing information for these requests:  
  
    - Request 1: Request activated at reference 1  
  
  | Representatives of a representative process:  
    reference 1 rank 1: MPI_Irecv (1st occurrence) called from:  
    #0 main@example-fix4.c:17  
    reference 2 rank 0: MPI_Type_contiguous (1st occurrence) called from:  
    #0 main@example-fix4.c:13  
    reference 3 rank 0: MPI_Type_commit (1st occurrence) called from:  
    #0 main@example-fix4.c:14 |
| 0       | Error | The memory regions to be transferred by this send operation overlap with regions spanned by a pending non-blocking receive operation!  
  
    (Information on the request associated with the other communication: Request activated at reference 1)  
    (Information on the datatype associated with the other communication: MPI_INT)  
    The other communication overlaps with this communication at position:(MPI_INT)  
    (Information on the datatype associated with this communication: Datatype created at reference 2 is for C, committed at reference 3, based on the following type(s): { MPI_INT })  
    This communication overlaps with the other communication at position:(contiguous)[0](MPI_INT)  
    A graphical representation of this situation is available in a detailed overlap view (MUSTOutputsfiles/MUST_Overlap_0_0.html). | reference 1 rank 0: MPI_Irecv (1st occurrence) called from:  
    #0 main@example-fix4.c:17  
    reference 2 rank 0: MPI_Type_contiguous (1st occurrence) called from:  
    #0 main@example-fix4.c:13  
    reference 3 rank 0: MPI_Type_commit (1st occurrence) called from:  
    #0 main@example-fix4.c:14 |

MUST has completed successfully, end date: Mon Dec 2 18:36:20 2013.
Graphical representation of the race condition

Graphical representation of the data race location
Fix5: use independent memory regions

```c
#include <mpi.h>
#include <stdio.h>

int main (int argc, char** argv)
{
    int rank, size, buf[8];

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    MPI_Datatype type;
    MPI_Type_contiguous (2, MPI_INTEGER, &type);
    MPI_Type_commit (&type);

    MPI_Request request;
    MPI_Irecv (buf, 2, MPI_INT, size - rank - 1, 123, MPI_COMM_WORLD, &request);

    MPI_Send (buf + 4, 1, type, size - rank - 1, 123, MPI_COMM_WORLD);

    printf ("Hello, I am rank %d of %d.\n", rank, size);

    MPI_Finalize ();

    return 0;
}
```
MUST detects leaks of user defined objects

- User defined objects include:
  - MPI_Comms (even by MPI_Comm_dup)
  - MPI_Datatypes
  - MPI_Groups

Leak of user defined datatype object
# Fix6: Deallocate datatype object

```c
#include <mpi.h>
#include <stdio.h>

int main (int argc, char** argv)
{
    int rank, size, buf[8];

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    MPI_Datatype type;
    MPI_Type_contiguous (2, MPI_INT, &type);
    MPI_Type_commit (&type);

    MPI_Request request;
    MPI_Irecv (buf, 2, MPI_INT, size - rank - 1, 123, MPI_COMM_WORLD, &request);

    MPI_Send (buf + 4, 1, type, size - rank - 1, 123, MPI_COMM_WORLD);

    printf ("Hello, I am rank %d of %d.\n", rank, size);
    MPI_Type_free (&type);

    MPI_Finalize ();

    return 0;
}
```

Deallocate the created datatype
MUST detects unfinished asynchronous communication

<table>
<thead>
<tr>
<th>Rank(s)</th>
<th>Type</th>
<th>Message</th>
<th>From</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1</td>
<td>Error</td>
<td>There are 1 requests that are not freed when MPI_Finalize was issued, a quality application should free all MPI resources before calling MPI_Finalize. Listing information for these requests: Request 1: Request activated at reference 1</td>
<td></td>
<td>References of a representative process: Reference 1 rank 0: MPI_Irecv (1st occurrence) called from: #0 <a href="mailto:main@example-fix6.c">main@example-fix6.c</a>:17</td>
</tr>
</tbody>
</table>

MUST has completed successfully, end date: Thu Nov 28 13:55:49 2013.

Remaining unfinished asynchronous receive
Fix8: use MPI_Wait to finish asynchronous communication

```c
#include <mpi.h>
#include <stdio.h>

int main (int argc, char** argv)
{
    int rank, size, buf[8];

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    MPI_Datatype type;
    MPI_Type_contiguous (2, MPI_INT, &type);
    MPI_Type_commit (&type);

    MPI_Request request;
    MPI_Irecv (buf, 2, MPI_INT, size - rank - 1, 123, MPI_COMM_WORLD, &request);

    MPI_Send (buf + 4, 1, type, size - rank - 1, 123, MPI_COMM_WORLD);

    MPI_Wait (&request, MPI_STATUS_IGNORE);

    printf ("Hello, I am rank %d of %d.\n", rank, size);
    MPI_Type_free (&type);

    MPI_Finalize ();

    return 0;
}
```

Finish the asynchronous communication
No further error detected

Hopefully this message applies to many applications
Content

- Motivation
- MPI usage errors
- Examples: Common MPI usage errors
  - Including MUST’s error descriptions
- Correctness tools
- MUST usage
Classes of Correctness Tools

- **Debuggers:**
  - Helpful to pinpoint any error
  - Finding the root cause may be very hard
  - Won’t detect sleeping errors
  - E.g.: gdb, TotalView, Alinea DDT

- **Static Analysis:**
  - Compilers and Source analyzers
  - Typically: type and expression errors
  - E.g.: MPI-Check

- **Model checking:**
  - Requires a model of your applications
  - State explosion possible
  - E.g.: MPI-Spin
Strategies of Correctness Tools

• Runtime error detection:
  - Inspect MPI calls at runtime
  - Limited to the timely interleaving that is observed
  - Causes overhead during application run
  - E.g.: Intel Trace Analyzer, Umpire, Marmot, MUST

• Formal verification:
  - Extension of runtime error detection
  - Explores ALL possible timely interleavings
  - Can detect potential deadlocks or type missmatches that would otherwise not occur in the presence of a tool
  - For non-deterministic applications exponential exploration space
  - E.g.: ISP
Content

- Motivation
- MPI usage errors
- Examples: Common MPI usage errors
  - Including MUST’s error descriptions
- Correctness tools
- MUST usage
MUST Usage

1) Compile and link application as usual
   - Link against the shared version of the MPI lib (Usually default)

2) Replace “mpiexec” with “mustrun”
   - E.g.: mustrun –np 4 myApp.exe input.txt output.txt

3) Inspect “MUST_Output.html” in run directory
   - “MUST_Output/MUST_Deadlock.dot” exists in case of deadlock
   - Visualize with: dot –Tps MUST_Deadlock.dot –o deadlock.ps

• The mustrun script will use an extra process for non-local checks (Invisible to application)
• I.e.: “mustrun –np 4 …” will issue a “mpirun –np 5 …”
• Make sure to allocate the extra task in batch jobs
MUST - Features

- Local checks:
  - Integer validation
  - Integrity checks (pointers valid, etc.)
  - Operation, Request, Communicator, Datatype, Group usage
  - Resource leak detection
  - Memory overlap checks

- Non-local checks:
  - Collective verification
  - Lost message detection
  - Type matching (For P2P and collectives)
  - Deadlock detection (with root cause visualization)
MUST - Features: Scalability

- Local checks largely scalable

- Non-local checks:
  - Current default uses a central process
    - This process is an MPI task taken from the application
    - Limited scalability ~100 tasks (Depending on application)
  - Distributed analysis available (tested with 10k tasks)
    - Uses more extra tasks (10%-100%)

- Recommended: Logging to an HTML file

- Uses a scalable tool infrastructure

- Tool configuration happens at execution time