MPI Runtime Error Detection with MUST

For the 13th VI-HPS Tuning Workshop

Joachim Protze and Felix Münchhalfen
IT Center RWTH Aachen University
February 2014
• MPI Usage Errors
• Error Classes
• Avoiding Errors
• Correctness Tools
• Runtime Error Detection
• MUST
• Hands On
Motivation

• MPI programming is error prone
• Portability errors
  (just on some systems, just for some runs)
• Bugs may manifest as:
  – Crash
  – Application hanging
  – Finishes
• Questions:
  – Why crash/hang?
  – Is my result correct?
  – Will my code also give correct results on another system?

• Tools help to pin-point these bugs
Common MPI Error Classes

- Common syntactic errors:
  - Incorrect arguments
  - Resource usage
  - Lost/Dropped Requests
  - Buffer usage
  - Type-matching
  - Deadlocks

  Tool to use: MUST, Static analysis tool, (Debugger)

- Semantic errors that are correct in terms of MPI standard, but do not match the programmers intent:
  - Displacement/Size/Count errors

  Tool to use: Debugger
MPI Usage Errors (2)

• 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

• MPI Usage Errors
• Error Classes
• Avoiding Errors
• Correctness Tools
• Runtime Error Detection
• MUST
• Hands On
Error Classes – Incorrect Arguments

• Complications
  – Calls with many arguments
  – In Fortran many arguments are of type INTEGER
  – Several restrictions for arguments of some calls
    ⇒ Compilers can’t detect all incorrect arguments

• Example:

```c
MPI_Send(
  buf,
  count,
  MPI_INTEGER,
  target,
  tag,
  MPI_COMM_WORLD);
```
Error Classes – Resource Usage

• Complications
  – Many types of resources
  – Leaks
  – MPI internal limits

• Example:
  
  ```c
  MPI_Comm_dup (MPI_COMM_WORLD, &newComm);
  MPI_Finalize();
  ```
### Complications

- Memory regions passed to MPI must not overlap (except send-send)
- Derived datatypes can span non-contiguous regions
- Collectives can both send and receive

### Example:

```c
MPI_Isend (&(buf[0]), 5 /*count*/, MPI_INT, ...);
MPI_Irecv (&(buf[4]), 5 /*count*/, MPI_INT, ...);
```
• Complications
  – Complex derived types
  – Types match if the signature matches, not their constructors
  – Partial receives

• Example 1:

Task 0

MPI_Send (buf, 1, MPI_INT);

Task 1

MPI_Recv (buf, 1, MPI_INT);

  – Matches => Equal types match
• Example 2:
  – Consider type $T_1 = \{\text{MPI\_INT}, \text{MPI\_INT}\}$

<table>
<thead>
<tr>
<th>Task 0</th>
<th>Task 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MPI_Send (buf, 1, T1);</code></td>
<td><code>MPI_Recv (buf, 2, \text{MPI\_INT});</code></td>
</tr>
</tbody>
</table>

  – Matches => type signatures are equal

• Example 3:
  – $T_1 = \{\text{MPI\_INT}, \text{MPI\_FLOAT}\}$
  – $T_2 = \{\text{MPI\_INT}, \text{MPI\_INT}\}$

<table>
<thead>
<tr>
<th>Task 0</th>
<th>Task 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MPI_Send (buf, 1, T1);</code></td>
<td><code>MPI_Recv (buf, 1, T2);</code></td>
</tr>
</tbody>
</table>

  – Missmatch => MPI\_INT != MPI\_FLOAT
• Example 4:
  – T1 = {MPI_INT, MPI_FLOAT}
  – T2 = {MPI_INT, MPI_FLOAT, MPI_INT}
  - Matches => MPI allows partial receives

• Example 4:
  – T1 = {MPI_INT, MPI_FLOAT}
  – T2 = {MPI_INT, MPI_FLOAT, MPI_INT}
  - Missmatch => Partial send is not allowed
Error Classes – Deadlocks

• Complications:
  – Non-blocking communication
  – Complex completions (Wait{all, any, some})
  – Non-determinism (e.g. MPI_ANY_SOURCE)
  – Choices for MPI implementation (e.g. buffered MPI_Send)
  – Deadlocks may be caused by non-trivial dependencies

• Example 1:

Task 0

MPI_Recv (from:1);

Task 1

MPI_Recv (from:0);

  – Deadlock: 0 waits for 1, which waits for 0
How to visualise/understand deadlocks?
- Common approach waiting-for graphs (WFGs)
- One node for each rank
- Rank X waits for rank Y => node X has an arc to node Y

Consider situation from Example 1:

Visualization:

Deadlock criterion: cycle (For simple cases)
• What about collectives?
  – Rank calling coll. operation waits for all tasks to issue a matching call
    ⇒ One arc to each task that did not call a matching call
  – One node potentially has multiple outgoing arcs
  – Multiple arcs means: waits for all of the nodes

• Example 2:

Task 0
MPI_Bcast (WORLD);

Task 1
MPI_Bcast (WORLD);

Task 2
MPI_Gather (WORLD);

• Visualization:

• Deadlock criterion: cycle (Also here)
• What about freedom in semantic?
  – Collectives may not be synchronizing
  – Standard mode send may (or may not) be buffered

• Example 3:

```
Task 0
MPI_Send (to:1);

Task 1
MPI_Send (to:0);
```

• This is a deadlock!
  – These are called “potential” deadlocks
  – Can manifest for some implementations and/or message sizes

• Visualization:
Error Classes – Deadlocks (5)

• What about timely interleaving?
  – Non-deterministic applications
  – Interleaving determines what calls match or are issued
  – Causes bugs that only occur “sometimes”

• Example 3:

<table>
<thead>
<tr>
<th>Task 0</th>
<th>Task 1</th>
<th>Task 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Send(to:1)</td>
<td>MPIRecv(from:ANY);</td>
<td>MPI_Send(to:1)</td>
</tr>
<tr>
<td>MPI_Barrier()</td>
<td>MPIRecv(from:2)</td>
<td>MPI_Barrier()</td>
</tr>
</tbody>
</table>

• What happens:
  – Case A:
    ◆ Recv (from:ANY) matches send from task 0
    ◆ All calls complete
  – Case B:
    ◆ Recv (from:ANY) matches send from task 2
    ◆ Deadlock
• What about “any” and “some”?
  – MPI_Waitany/Waitsome and wild-card (MPI_ANY_SOURCE) receives have special semantics
  – These wait for at least one out of a set or ranks
  – This is different from the “waits for all” semantic

• Example 4:

  Task 0  Task 1  Task 2
  MPI_Recv(from:1)  MPI_Recv(from:ANY);  MPI_Recv(from:1)

• What happens:
  – No call can progress, Deadlock
  – 0 waits for 1; 1 waits for either 0 or 1; 2 waits for 1
• How to visualize the “any/some” semantic?
  – There is the “Waits for all of” wait type => “AND” semantic
  – There is the “Waits for any of” wait type => “OR” semantic
  – Each type gets one type of arcs
    – AND: solid arcs
    – OR: Dashed arcs

• Visualization for Example 4:

<table>
<thead>
<tr>
<th>Task 0</th>
<th>Task 1</th>
<th>Task 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Recv(from:1)</td>
<td>MPI_Recv(from:ANY);</td>
<td>MPI_Recv(from:1)</td>
</tr>
</tbody>
</table>

0: MPI_Recv
1: MPI_Recv
2: MPI_Recv
• Deadlock criterion for AND + OR
  – Cycles are necessary but not sufficient
  – A weakened form of a knot (OR-Knot) is the actual criterion
  – Tools can detect it and visualize the core of the deadlock

• Some examples:
  – An OR-Knot (which is also a knot, Deadlock):

  \[
  \begin{array}{ccc}
  0 & \rightarrow & 1 \\
  1 & \rightarrow & 2 \\
  \end{array}
  \]

  – Cycle but no OR-Knot (Not Deadlocked):

  \[
  \begin{array}{ccc}
  0 & \rightarrow & 1 \\
  1 & \rightarrow & 2 \\
  2 & \rightarrow & 0 \\
  \end{array}
  \]

  – OR-Knot but not a knot (Deadlock):

  \[
  \begin{array}{ccc}
  0 & \rightarrow & 1 \\
  1 & \rightarrow & 2 \\
  2 & \rightarrow & 0 \\
  0 & \rightarrow & 3 \\
  \end{array}
  \]
• Description:
  – Erroneous sizes, counts, or displacements
    • Example: During datatype construction or communication
    • Often “off-by-one” errors
  • Example (C):

```c
/* Create datatype to send a column of 10x10 array */
MPI_Type_vector (  
  10,              /* #blocks */  
  1,               /* #elements per block */  
  9,               /* #stride */  
  MPI_INT,         /* old type */  
  &newType         /* new type */  
);"
• MPI Usage Errors
• Error Classes
• **Avoiding Errors**
• Correctness Tools
• Runtime Error Detection
• MUST
• Hands On
Avoiding Errors

- The bugs you don’t introduce are the best one:
  - Think, don’t hack
  - Comment your code
  - Confirm consistency with asserts
  - Consider a verbose mode of your application
  - Use unit testing, or at least provide test cases
  - Set up nightly builds
    - MPI Testing Tool:
    - Ctest & Dashboards:
      - http://www.vtk.org/Wiki/CMake_Testing_With_CTest
• MPI Usage Errors
• Error Classes
• Avoiding Errors
• **Correctness Tools**
• Runtime Error Detection
• MUST
• Hands On
• **Debuggers:**
  - **Helpful to pinpoint any error**
  - Finding the root cause may be hard
  - Won’t detect sleeping errors
  - E.g.: gdb, TotalView, Allinea 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

```c
MPI_Recv (buf, 5, MPI_INT, -1, 123, MPI_COMM_WORLD, &status);
```

"-1" instead of "MPI_ANY_SOURCE"

```c
if (rank == 1023)
crash ();
```

Only works with less than 1024 tasks
• 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 relevant* interleavings (explore around nondet.)
  – Detects errors that only manifest in some runs
  – Possibly many interleavings to explore
  – E.g.: ISP

```
Task 0
spend_some_time()
MPI_Send (to:1)
MPI_Barrier ()

Task 1
MPI_Recv (from:ANY)
MPI_Recv (from:0)
MPI_Barrier ()

Task 2
MPI_Send (to:1)
MPI_Barrier ()
```

Deadlock if MPI_Send(to:1)@0 matches MPI_Recv(from:ANY)@1
Our contribution: MUST

Runtime Checking

Approaches to Remove Bugs (Selection)

Downscaling

Repertioning? Reproducibility?
Representative input? Node Memory?
Grid?

Static Code Analysis

TASS pCFG’s

Barrier Analysis

Model Checking

Debuggers

Umpire ISP/ DAMPI

MARMOT
- MPI Usage Errors
- Error Classes
- Avoiding Errors
- Correctness Tools
- **Runtime Error Detection**
- MUST
- Hands On
• A MPI wrapper library intercepts all MPI calls

![Diagram]

Application \(\rightarrow\) \(\text{calls MPI}_X(\ldots)\)

Correctness Tool \(\rightarrow\) \(\text{calls PMPI}_X(\ldots)\)

MPI

• Checks analyse the intercepted calls
  – Local checks require data from just one task
    – E.g.: invalid arguments, resource usage errors
  – Non-local checks require data from multiple task
    – E.g.: type matching, collective verification, deadlock detection
• **Workflow:**
  - Attach tool to target application (Link library to application)
  - Configure tool
    - Enable/disable correctness checks
    - Select output type
    - Enable potential integrations (e.g. with debugger)
  - Run application
    - Usually a regular mpirun
    - Non-local checks may require extra resources, e.g. extra tasks
  - Analyze correctness report
    - May even be available if the application crashes
  - Correct bugs and rerun for verification
• MPI Usage Errors
• Error Classes
• Avoiding Errors
• Correctness Tools
• Runtime Error Detection
• MUST
• Hands On
• MPI runtime error detection tool
• Open source (BSD license)
  http://tu-dresden.de/zh/must
• Wide range of checks, strength areas:
  – Overlaps in communication buffers
  – Errors with derived datatypes
  – Deadlocks
• Largely distributed, can scale with the application
MUST – Correctness Reports

• C code:

```c
MPI_Type_contiguous (2, MPI_INTEGER, &newtype);
MPI_Send (buf, count, newtype, target, tag, MPI_COMM_WORLD);
```

• Tool Output:

Use of uncommitted type

Who? What? Where? Details

<table>
<thead>
<tr>
<th>Rank(s)</th>
<th>Type</th>
<th>Message</th>
<th>From</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Error</td>
<td>Argument 3 (datatype) is not committed 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}Typemap = {(MPI_INTEGER, 0), (MPI_INTEGER, 4)})</td>
<td></td>
</tr>
</tbody>
</table>

References:

- References of a representative process:
  - MPI_Send (1st occurrence) called from:
    - #0 main@test.c:17
  - MPI_Type_contiguous (1st occurrence) called from:
    - #0 main@test.c:14
• Apply MUST with an mpiexec wrapper, that’s it:

% mpicc source.c -o exe
% mpiexec -np 4 ./exe

% mpicc -g source.c -o exe
% mustrun -np 4 ./exe

• After run: inspect “MUST_Output.html”
• “mustrun” (default config.) uses an extra process:
  – I.e.: “mustrun –np 4 …” will use 5 processes
  – Allocate the extra resource in batch jobs!
  – Default configuration tolerates application crash; BUT is very slow (details later)
• Chances are good that you will get:

  ![MUST Output File]

  MUST Output, starting date: Wed Feb 5 09:16:34 2014.

<table>
<thead>
<tr>
<th>Rank(s)</th>
<th>Type</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Information</td>
<td>MUST detected no MPI usage errors nor any suspicious behavior during this application run.</td>
</tr>
</tbody>
</table>

  MUST has completed successfully, end date: Wed Feb 5 09:16:34 2014.

• Congratulations you appear to use MPI correctly!

• Consider:
  – Different process counts or inputs can still yield errors
  – Errors may only be visible on some machines
  – Integrate MUST into your regular testing
• Derived datatypes use constructors, example:

```c
MPI_Type_vector (  
  NumRows /*count*/,  
  1 /*blocklength*/,  
  NumColumns /*stride*/,  
  MPI_INT /*oldtype*/,  
  &newType);
```

• Errors that involve datatypes can be complex:
  – Need to be detected correctly
  – Need to be visualized
Errors with MPI Datatypes – Example

• C Code:

```c
MPI_Isend(buf, 1 /*count*/ , vectortype , target, tag, MPI_COMM_WORLD, &request);
MPI_Recv(buf, 1 /*count*/ , columntype , target, tag, MPI_COMM_WORLD, &status);
MPI_Wait (&request, &status);

...```

• Memory:

Error: buffer overlap
MPI_Isend reads, MPI_Recv writes at the same time

A Tool must:
- Detect the error
- Pinpoint the user to the exact problem
Errors with MPI Datatypes – Error Positions

• How to point to an error in a derived datatype?
  – Derived types can span wide areas of memory
  – Understanding errors requires precise knowledge
  – E.g., not sufficient: Type X overlaps with type Y

• Example:

• We use path expressions to point to error positions
  – For the example, overlap at:
    • [0](VECTOR)[2][0](MPI_INT)
    • [0](CONTIGUOUS)[0](MPI_INT)
• Example “vihps13_2014.c”:

1. MPI_Init (&argc,&argv);
2. MPI_Comm_rank (MPI_COMM_WORLD, &rank);
3. MPI_Comm_size (MPI_COMM_WORLD, &size);
4. //1) Create a datatype
5. MPI_Type_contiguous (2, MPI_INT, &newType);
6. MPI_Type_commit (&newType);
7. //2) Use MPI_Sendrecv to perform a ring communication
8. MPI_Sendrecv (
9. sBuf, 1, newType, (rank+1)%size, 123,
10. rBuf, sizeof(int)*2, MPI_BYTE, (rank-1+size) % size, 123,
11. MPI_COMM_WORLD, &status);
12. //3) Use MPI_Send and MPI_Recv to perform a ring communication
13. MPI_Send (sBuf, 1, newType, (rank+1)%size, 456,
14. MPI_COMM_WORLD);
15. MPI_Recv (rBuf, sizeof(int)*2, MPI_BYTE, (rank-1+size) % size, 456,
16. MPI_COMM_WORLD, &status);
17. MPI_Finalize ();
• Runs without any apparent issue with OpenMPI
• Are there any errors?

• Verify with MUST:

% mpicc -g vihps13_2014.c \ 
   -o vihps13_2014.exe
% mustrun -np 4 vihps13_2014.exe
% firefox MUST_Output.html
**MUST Usage Example – Error 1: Buffer Overlap**

**First error: Overlap in Isend + Recv**

<table>
<thead>
<tr>
<th>Rank(s)</th>
<th>Type</th>
<th>Message</th>
<th>From</th>
<th>References</th>
</tr>
</thead>
</table>
| 0       | Error  | The memory regions to be transferred by this receive operation overlap with regions spanned by a pending non-blocking operation!  
Information on the request associated with the other communication:  
Request activated at reference 1  
Information on the datatype associated with the other communication:  
Datatype created at reference 2 is for C, committed at reference 3, based on the following type(s): {MPI_INT}Typemap = {(MPI_INT, 0), (MPI_INT, 20), (MPI_INT, 40), (MPI_INT, 60), (MPI_INT, 80)}  
The other communication overlaps with this communication at position:(vector)[2] [0](MPI_INT)  
Information on the datatype associated with this communication:  
Datatype created at reference 4 is for C, committed at reference 5, based on the following type(s): {MPI_INT}Typemap = {(MPI_INT, 0), (MPI_INT, 4), (MPI_INT, 8), (MPI_INT, 12), (MPI_INT, 16)}  
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 (MUST_Output-files/MUST_Overlap_0_0.html). | Representative location:  
**MPI_Rece** (1st occurrence) called from:  
#0  
main@mpi_overlap_deadlock_errors.c:23  
References of a representative process:  
reference 1 rank 0: MPI_Isend (1st occurrence) called from:  
#0  
main@mpi_overlap_deadlock_errors.c:22  
reference 2 rank 0: MPI_Type_vector (1st occurrence) called from:  
#0  
main@mpi_overlap_deadlock_errors.c:17  
reference 3 rank 0: MPI_Type_commit (2nd occurrence) called from:  
#0  
main@mpi_overlap_deadlock_errors.c:19  
reference 4 rank 0:  
**MPI_Type_contiguous** (1st occurrence) called from:  
#0  
main@mpi_overlap_deadlock_errors.c:16  
reference 5 rank 0: MPI_Type_commit (1st occurrence) called from:  
#0  
main@mpi_overlap_deadlock_errors.c:18 |
MUST Usage Example – Error 1: Buffer Overlap

- First error: Overlap in Isend + Recv

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

(Information on the request associated with the other communication:
Request activated at reference 1
Information on the datatype associated with the other communication:
Datatype created at reference 2 is for C, committed at reference 3 based on the following type(s): { MPI_INT } Typemap = { (MPI_INT, 0), (MPI_INT, 20), (MPI_INT, 40), (MPI_INT, 60), (MPI_INT, 80) }

The other communication overlaps with this communication at position: (VECTOR)[2][0] (MPI_INT)

(Information on the datatype associated with this communication:
Datatype created at reference 4 is for C, committed at reference 5, 6 based on the following type(s): { MPI_INT } Typemap = { (MPI_INT, 0), (MPI_INT, 4), (MPI_INT, 8), (MPI_INT, 12), (MPI_INT, 16) }

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 (MUST_Overlap.html)

These refer to the “References” (Details) column
Visualization of overlap (MUSTOverlap.html):

The application issued a set of MPI calls that overlap in communication buffers! The graph below shows details on this situation. The first colliding item of each involved communication request is highlighted.
Warning for unusual values, that match MPI specification:

<table>
<thead>
<tr>
<th>Rank(s)</th>
<th>Type</th>
<th>Message</th>
<th>From</th>
</tr>
</thead>
</table>
| 0-1     | Warning| Argument 2 (count) is zero, which is correct but unusual! | Representative location:  
MPI_Send (1st occurrence) called from:  
#0 main@mpi_overlap_deadlock_errors.c:26 |
• **Second Error: potential Deadlock**

<table>
<thead>
<tr>
<th>Rank(s)</th>
<th>Type</th>
<th>Message</th>
<th>From</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Error</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 detailed deadlock view (MUST Output-files/MUST_Deadlock.html). 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>
<td>References of a representative process:</td>
<td>reference 1 rank 0: MPI_Send (1st occurrence) called from: #0 <a href="mailto:main@mpi_overlap_deadlock_errors.c">main@mpi_overlap_deadlock_errors.c</a>:26 reference 2 rank 1: MPI_Send (1st occurrence) called from: #0 <a href="mailto:main@mpi_overlap_deadlock_errors.c">main@mpi_overlap_deadlock_errors.c</a>:26</td>
</tr>
</tbody>
</table>
Visualization of deadlock (MUST_Deadlock.html)

- 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 leafs 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).

- Active Communicators

- Wait-for Graph

- Legend

- Call Stack
### MUST Usage Example – Error 3 Type Leak

- **Third error: Leaked resource (derived datatype)**

<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></td>
<td></td>
<td>References of a representative process:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>There are 2 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:</td>
<td></td>
<td>reference 1 rank 0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Datatype 1: Datatype created at reference 1 is for C, commited at reference 2, based on the following type(s): { MPI_INT } Typemap = { (MPI_INT, 0), (MPI_INT, 4), (MPI_INT, 8), (MPI_INT, 12), (MPI_INT, 16) }</td>
<td></td>
<td>MPI_Type_contiguous (1st occurrence) called from:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Datatype 2: Datatype created at reference 3 is for C, commited at reference 4, based on the following type(s): { MPI_INT } Typemap = { (MPI_INT, 0), (MPI_INT, 20), (MPI_INT, 40), (MPI_INT, 60), (MPI_INT, 80) }</td>
<td></td>
<td>#0 <a href="mailto:main@mpi_overlap_deadlock_errors.c">main@mpi_overlap_deadlock_errors.c</a>:16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>reference 2 rank 0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MPI_Type_commit (1st occurrence) called from:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#0 <a href="mailto:main@mpi_overlap_deadlock_errors.c">main@mpi_overlap_deadlock_errors.c</a>:18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>reference 3 rank 0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MPI_Type_vector (1st occurrence) called from:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#0 <a href="mailto:main@mpi_overlap_deadlock_errors.c">main@mpi_overlap_deadlock_errors.c</a>:16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>reference 4 rank 0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MPI_Type_commit (2nd occurrence) called from:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#0 <a href="mailto:main@mpi_overlap_deadlock_errors.c">main@mpi_overlap_deadlock_errors.c</a>:19</td>
</tr>
</tbody>
</table>
- Fourth error: Leaked resource (request)
  - Leaked requests often indicate missing synchronization by MPI_Wait/Test

<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>Representative location: MPI_Isend (1st occurrence) called from: #0 <a href="mailto:main@mpi_overlap_deadlock_errors.c">main@mpi_overlap_deadlock_errors.c</a>:22</td>
<td>References of a representative process: reference 1 rank 0: MPI_Isend (1st occurrence) called from: #0 <a href="mailto:main@mpi_overlap_deadlock_errors.c">main@mpi_overlap_deadlock_errors.c</a>:22</td>
</tr>
</tbody>
</table>
Example “mpi_overlap_deadlock_errors.c”:

1. `MPI_Init (&argc, &argv);`
2. `comm = MPI_COMM_WORLD;`  
3. `MPI_Comm_rank (comm, &rank);`
4. `MPI_Comm_size (comm, &size);`

//1) Create some datatypes
5. `MPI_Type_contiguous (5, MPI_INT, &rowType);`
6. `MPI_Type_commit (&rowType);`
7. `MPI_Type_vector (5 /*count*/, 1 /*blocklength*/, 5 /*stride*/, MPI_INT, &colType);`
8. `MPI_Type_commit (&colType);`

//2) Use MPI_Isend and MPI_Recv to perform a ring communication
9. `MPI_Isend (&arr[0], 1, colType, (rank+1)%size, 456, comm, &request);`
10. `MPI_Recv (&arr[10], 1, rowType, (rank-1+size) % size, 456, comm, &status);`

//3) Use MPI_Send and MPI_Recv to acknowledge recv
11. `MPI_Send (arr, 0, MPI_INT, (rank-1+size) % size, 345, comm);`
12. `MPI_Recv (arr, 0, MPI_INT, (rank+1)%size, 345, comm, &status);`

13. `MPI_Finalize ();`
• Example “mpi_overlap_deadlock_errors.c” :

(1)    MPI_Init ( &argc,&argv );
(2)    comm = MPI_COMM_WORLD;
(3)    MPI_Comm_rank ( comm, &rank );
(4)    MPI_Comm_size ( comm, &size );

(5) (6)    //1) Create some datatypes
(7)    MPI_Type_contiguous ( 5, MPI_INT, &rowType );
(8)    MPI_Type_commit ( &rowType );
(9)    MPI_Type_vector ( 5 /*count*/, 1 /*blocklength*/, 5 /*stride*/, MPI_INT, &colType );
(10)   MPI_Type_commit ( &colType );

(11)   (12)    //2) Use MPI_Isend and MPI_Recv to perform a ring communication
(13)   MPI_Isend ( &arr[0], 1, colType, (rank+1)%size, 456, comm, &request );
(14)   MPI_Recv ( &arr[10], 1, rowType, (rank-1+size) % size, 456, comm, &status );

(15)   (16)    //3) Use MPI_Send and MPI_Recv to acknowledge recv
(17)   MPI_Send ( arr, 0, MPI_INT, (rank-1+size) % size, 345, comm);
(18)   MPI_Recv ( arr, 0, MPI_INT, (rank+1)%size, 345, comm, &status );

(19)    MPI_Finalize ( );

User forgets to call an MPI_Wait for the MPI request
Example “mpi_overlap_deadlock_errors.c”:

(1)  

\[
\text{MPI\_Init ( \&argc, \&argv );}
\]

\[
\text{comm = MPI\_COMM\_WORLD;}
\]

\[
\text{MPI\_Comm\_rank ( comm, \&rank );}
\]

\[
\text{MPI\_Comm\_size ( comm, \&size );}
\]

(2)  

\[
\text{//1) Create some datatypes}
\]

\[
\text{MPI\_Type\_contiguous ( 5, MPI\_INT, \&rowType );}
\]

\[
\text{MPI\_Type\_commit ( \&rowType );}
\]

\[
\text{MPI\_Type\_vector ( 5 /*count*/, 1 /*blocklength*/, 5 /*stride*/, MPI\_INT, \&colType );}
\]

(3)  

\[
\text{MPI\_Type\_commit ( \&colType );}
\]

(4)  

\[
\text{//2) Use MPI\_ISend and MPI\_Recv to perform a ring communication}
\]

\[
\text{MPI\_Isend ( \&arr[0], 1, colType, (rank+1) \% size, 456, comm, \&request );}
\]

\[
\text{MPI\_Recv ( \&arr[10], 1, rowType, (rank-1+size) \% size, 456, comm, \&status );}
\]

(5)  

\[
\text{//3) Use MPI\_Send and MPI\_Recv to acknowledge recv}
\]

\[
\text{MPI\_Send ( arr, 0, MPI\_INT, (rank-1+size) \% size, 345, comm);}
\]

\[
\text{MPI\_Recv ( arr, 0, MPI\_INT, (rank+1)%size, 345, comm, \&status );}
\]

(6)  

\[
\text{MPI\_Finalize ( );}
\]

Send/recv count are 0, is this intended?
Example “mpi_overlap_deadlock_errors.c”:

1. `MPI_Init ( &argc, &argv );`
2. `comm = MPI_COMM_WORLD;`
3. `MPI_Comm_rank ( comm, &rank );`
4. `MPI_Comm_size ( comm, &size );`
5. `//1) Create some datatypes`
   6. `MPI_Type_contiguous ( 5, MPI_INT, &rowType );`
   7. `MPI_Type_commit ( &rowType );`
   8. `MPI_Type_vector ( 5 /*count*/, 1 /*blocklength*/, 5 /*stride*/, MPI_INT, &colType );`
   9. `MPI_Type_commit ( &colType );`
10. `//2) Use MPI_Isend and MPI_Recv to perform a ring communication`
    11. `MPI_Isend ( &arr[0], 1, colType, (rank+1)%size, 456, comm, &request );`
    12. `MPI_Recv ( &arr[10], 1, rowType, (rank-1+size) % size, 456, comm, &status );`
13. `//3) Use MPI_Send and MPI_Recv to acknowledge recv`
    14. `MPI_Send ( arr, 0, MPI_INT, (rank-1+size) % size, 345, comm );`
    15. `MPI_Recv ( arr, 0, MPI_INT, (rank+1)%size, 345, comm, &status );`
16. `MPI_Finalize ();`

Potential for deadlock, `MPI_Send` can block (depends on MPI implementation and buffer size)
• Example “mpi_overlap_deadlock_errors.c”:

1) \texttt{MPI\textunderscore Init} ( \&argc, \&argv );
2) \texttt{comm = MPI\textunderscore COMM\textunderscore WORLD;}
3) \texttt{MPI\textunderscore Comm\textunderscore rank} ( \texttt{comm, \&rank} );
4) \texttt{MPI\textunderscore Comm\textunderscore size} ( \texttt{comm, \&size} );
5) 

6) //1) Create some datatypes
7) \texttt{MPI\textunderscore Type\textunderscore contiguous} ( 5, MPI\textunderscore INT, \&rowType );
8) \texttt{MPI\textunderscore Type\textunderscore commit} ( \&rowType );
9) \texttt{MPI\textunderscore Type\textunderscore vector} ( 5 /*count*/, 1 /*blocklength*/, 5 /*stride*/, MPI\textunderscore INT, \&colType );
10) \texttt{MPI\textunderscore Type\textunderscore commit} ( \&colType );
11)

12) //2) Use \texttt{MPI\textunderscore ISend} and \texttt{MPI\textunderscore Recv} to perform a ring communication
13) \texttt{MPI\textunderscore Isend} ( \&arr[0], 1, colType, (rank+1)\%size, 456, comm, \&request );
14) \texttt{MPI\textunderscore Recv} ( \&arr[10], 1, rowType, (rank-1+size) \% size, 456, comm, \&status );
15)

16) //3) Use \texttt{MPI\textunderscore Send} and \texttt{MPI\textunderscore Recv} to acknowledge recv
17) \texttt{MPI\textunderscore Send} ( arr, 0, MPI\textunderscore INT, (rank-1+size), 345, comm );
18) \texttt{MPI\textunderscore Recv} ( arr, 0, MPI\textunderscore INT, (rank+1)\%size, 345, comm, \&status );
19)

20) \texttt{MPI\textunderscore Finalize} ();

\textbf{User forgot to free MPI Datatypes before calling MPI\textunderscore Finalize}
• MPI Usage Errors
• Error Classes
• Avoiding Errors
• Correctness Tools
• Runtime Error Detection
• MUST
• Hands On
• Go into the NPB directory
• Edit config/make.def
• Disable any other tool (i.e. use mpif77)
• Build:

```bash
% make bt-mz NPROCS=4 CLASS=B
============================================================================
= NAS PARALLEL BENCHMARKS 3.3 =
= MPI+OpenMP Multi-Zone Versions =
= F77 =
============================================================================

cd BT-MZ; make CLASS=B NPROCS=4
make[1]: Entering directory
...

mpi77 -O3 -g -openmp -o ../../../bin/bt-mz.B.4 bt.o initialize.o ...
make[1]: Leaving directory
Hands On - Prepare Job

• Go to bin directory

% cd bin

• Create and edit the jobscript

cp ../*.jobscript/marenostrum/run.must.lsf ./
vim run.must.lsf

• Jobscript:

...  
#BSUB -n 16  
#BSUB -R "span[ptile=16]"  
...  
export OMP_NUM_THREADS=3  
module load UNITE must  
module list  
mustrun -np 4 ./bt-mz_B.4

MUST needs one extra process!  
We use 4 processes * 3 threads + 1 tool process
Hands On – Executing with MUST

• Submit the jobscript:

```
bsub < run.must.lsf
```

• Job output should read:

```
[MUST] MUST configuration ... centralized checks with fall-back application crash handling (very slow)
[MUST] Information: overwriting old intermediate data in directory “(...)must_temp”!
[MUST] Weaver ... success
[MUST] Code generation ... success
...
[MUST] Generating P^nMPI configuration ... success
[MUST] Search for preloaded P^nMPI ... not found ... success
[MUST] Executing application: NAS Parallel Benchmarks (NPB3.3-MZ-MPI) - BT-MZ MPI+OpenMP Benchmark
...
Total number of threads:     16  ( 3.0 threads/process)
Calculated speedup =     11.97

Time step   1
...
Verification Successful
...
[MUST] Execution finished, inspect “(...)MUST_Output.html”!
```
OpenMPI-1.5 has by default no thread support. BT-MZ should evaluate the “provided” thread level and don’t use threads.

Resource leak:
A communicator created with MPI_Comm_split is not free
We use an external lib for stacktraces
This lib has no support for Intel compiler
  - But: in most cases it’s compatible to icc compiled C applications
  - You may load the must/intel+stackwalker module for C applications
Ifort compiled FORTRAN applications lead to segfault
  - Use the default module for fortran applications
  - Use GNU compiler to build your application and load the must/GNU+stackwalker module
Supposed your application has no faults you won’t need stacktraces 😊
Conclusions

• Many types of MPI usage errors
  – Some errors may only manifest sometimes
  – Consequences of some errors may be “invisible”
  – Some errors can only manifest on some systems/MPIs

• Use MPI correctness tools

• Runtime error detection with MUST
  – Provides various correctness checks
  – Verifies type matching
  – Detects deadlocks
  – Verifies collectives
• MUST is a runtime MPI error detection tool
• Usage:
  – Compile & link as always
  – Use “mustrun” instead of “mpirun”
  – Keep in mind to allocate at least 1 extra task in batch jobs
    – “--must:info” for details on task usage
  – Add “--must:nocrash” if your application does not crash
  – Open “MUST_Output.html” after the run completed/crashed