



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

- 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 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

- 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

- MPI Usage Errors
- **Error Classes**
- Avoiding Errors
- Correctness Tools
- Runtime Error Detection
- MUST
- Hands On

- 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:

```
MPI_Send(  
    buf,  
    count,  
    MPI_INTEGER,  
    target,  
    tag,  
    MPI_COMM_WORLD);
```

- Complications
 - Many types of resources
 - Leaks
 - MPI internal limits
- Example:

```
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:

```
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 $T1 = \{\text{MPI_INT}, \text{MPI_INT}\}$

Task 0

```
MPI_Send (buf, 1, T1);
```

Task 1

```
MPI_Recv (buf, 2, MPI_INT);
```

- Matches \Rightarrow type signatures are equal

- Example 3:

- $T1 = \{\text{MPI_INT}, \text{MPI_FLOAT}\}$

- $T2 = \{\text{MPI_INT}, \text{MPI_INT}\}$

Task 0

```
MPI_Send (buf, 1, T1);
```

Task 1

```
MPI_Recv (buf, 1, T2);
```

- Mismatch $\Rightarrow \text{MPI_INT} \neq \text{MPI_FLOAT}$

- Example 4:

- $T1 = \{\text{MPI_INT}, \text{MPI_FLOAT}\}$
- $T2 = \{\text{MPI_INT}, \text{MPI_FLOAT}, \text{MPI_INT}\}$

Task 0

```
MPI_Send (buf, 1, T1);
```

Task 1

```
MPI_Recv (buf, 1, T2);
```

- Matches => MPI allows partial receives

- Example 4:

- $T1 = \{\text{MPI_INT}, \text{MPI_FLOAT}\}$
- $T2 = \{\text{MPI_INT}, \text{MPI_FLOAT}, \text{MPI_INT}\}$

Task 0

```
MPI_Send (buf, 2, T1);
```

Task 1

```
MPI_Recv (buf, 1, T2);
```

- Mismatch => Partial send is not allowed

- 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

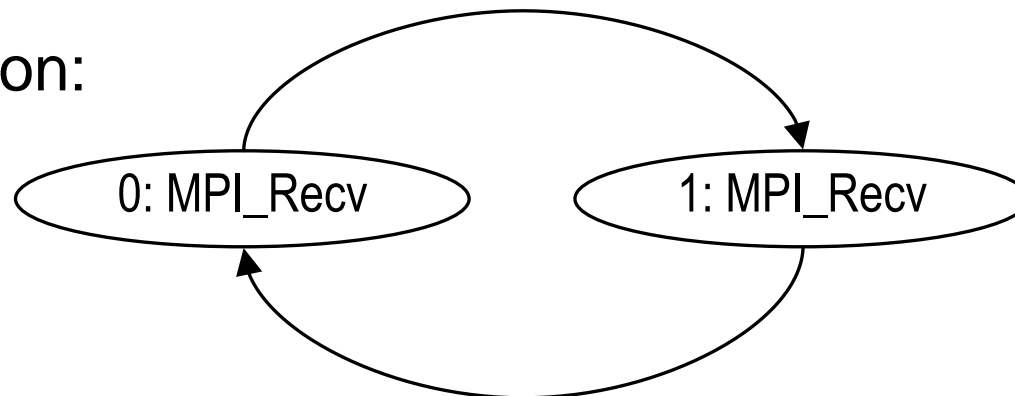
MPI_Recv (from:0);

- 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 \Rightarrow node X has an arc to node Y
- Consider situation from Example 1:



- Visualization:



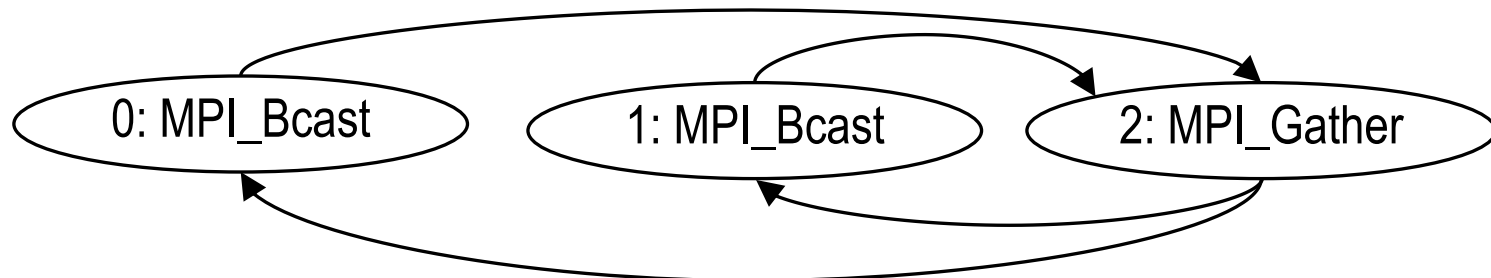
- Deadlock criterion: cycle (For simple cases)

- 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:



- Visualization:

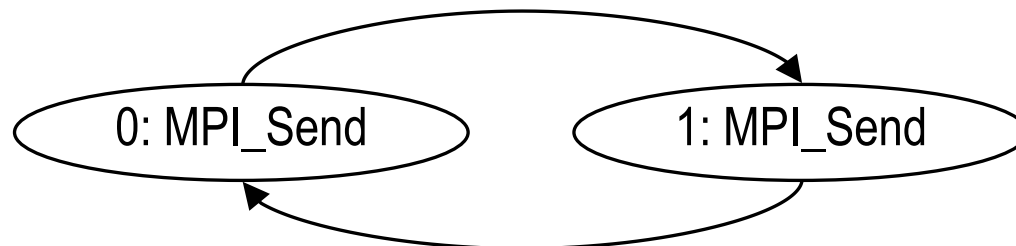


- Deadlock criterion: cycle (Also here)

- What about freedom in semantic?
 - Collectives may not be synchronizing
 - Standard mode send may (or may not) be buffered
- Example 3:



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



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

- Example 3:

Task 0	Task 1	Task 2
<pre>MPI_Send(to:1) MPI_Barrier()</pre>	<pre>MPI_Recv(from:ANY); MPI_Recv(from:2) MPI_Barrier()</pre>	<pre>MPI_Send(to:1) MPI_Barrier()</pre>

- 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

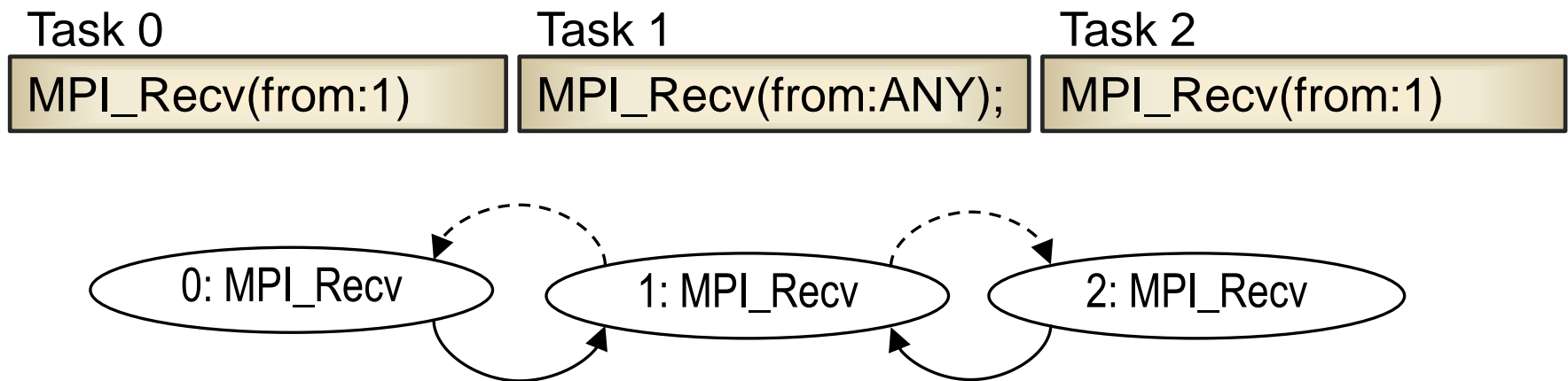
- What about “any” and “some”?
 - MPI_Waitany/Waitsome and wild-card (MPI_ANY_SOURCE) receives have special semantics
 - These wait for at least one out of a set or ranks
 - This is different from the “waits for all” semantic

- Example 4:

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

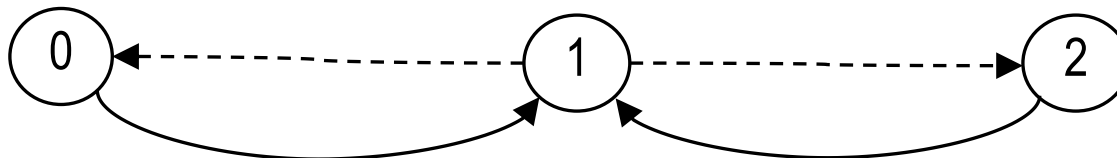
- What happens:
 - No call can progress, Deadlock
 - 0 waits for 1; 1 waits for either 0 or 1; 2 waits for 1

- 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:

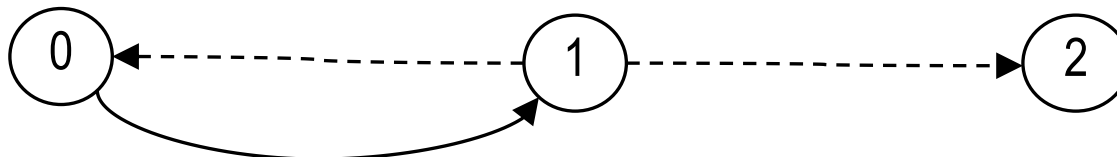


- Deadlock criterion for AND + OR
 - Cycles are necessary but not sufficient
 - A weakened form of a knot (OR-Knot) is the actual criterion
 - Tools can detect it and visualize the core of the deadlock
- Some examples:

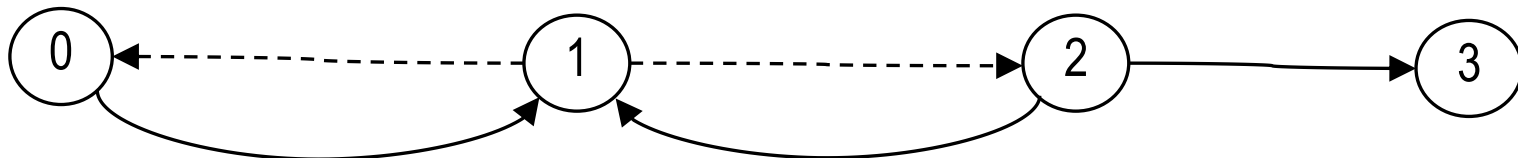
- An OR-Knot (which is also a knot, Deadlock):



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



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



- Description:
 - Erroneous sizes, counts, or displacements
 - Example: During datatype construction or communication
 - Often “off-by-one” errors
- Example (C):

Stride must be 10 for a column

```
/* Create datatype to send a column of 10x10 array */
MPI_Type_vector (
    10,
    1,
    9,
    MPI_INT,
    &newType
    /* #blocks */
    /* #elements per block */
    /* #stride */
    /* old type */
    /* new type */ );
```

- MPI Usage Errors
- Error Classes
- **Avoiding Errors**
- Correctness Tools
- Runtime Error Detection
- MUST
- Hands On

- 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:
 - <http://www.open-mpi.org/projects/mtt/>
 - 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

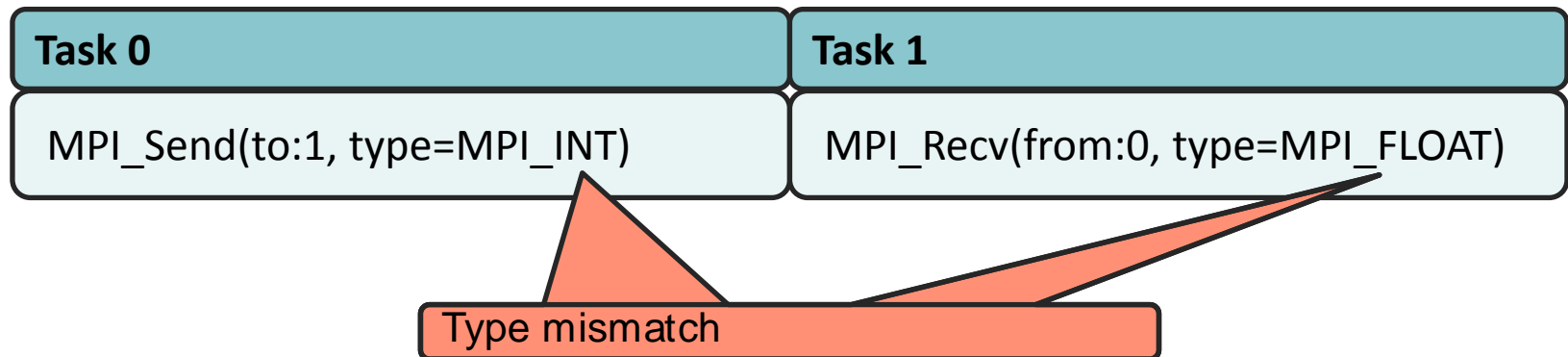
```
MPI_Recv (buf, 5, MPI_INT,  
-1,  
123, MPI_COMM_WORLD,  
&status);
```

“-1” instead of “MPI_ANY_SOURCE”

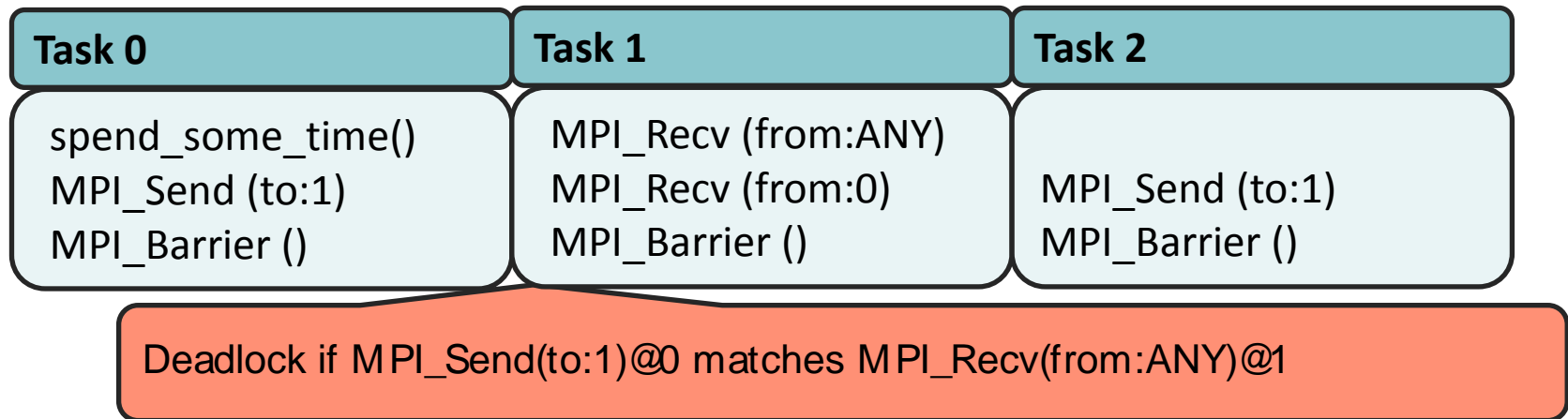
```
if (rank == 1023)  
    crash ();
```

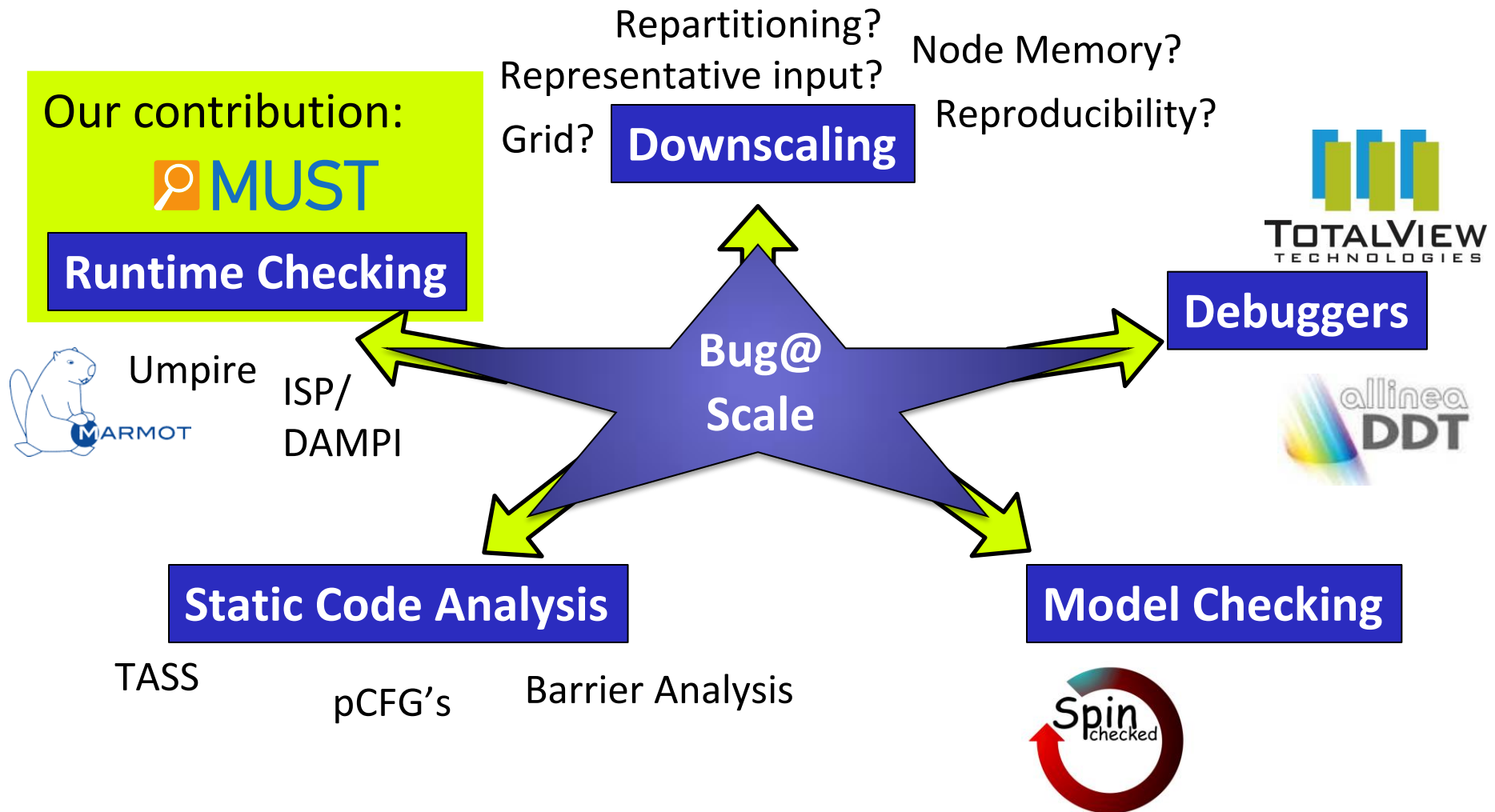
Only works with less than 1024 tasks

- 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**



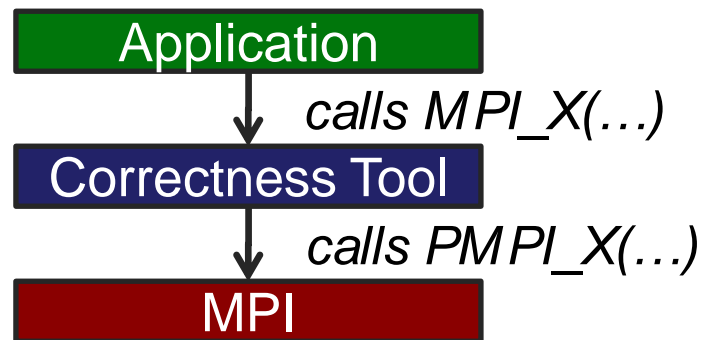
- Formal verification:
 - Extension of runtime error detection
 - Explores *all relevant* interleavings (explore around nondet.)
 - Detects errors that only manifest in some runs
 - Possibly many interleavings to explore
 - E.g.: ISP





- MPI Usage Errors
- Error Classes
- Avoiding Errors
- Correctness Tools
- **Runtime Error Detection**
- MUST
- Hands On

- A MPI wrapper library intercepts all MPI calls



- 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/zih/must>
- Wide range of checks, strength areas:
 - Overlaps in communication buffers
 - Errors with derived datatypes
 - Deadlocks
- Largely distributed, can scale with the application

- C code:

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

Use of uncommitted type

- Tool Output:

Who?

What?

Where?

Details

MUST Outputfile

14:11 2014.

Rank(s)	Type	Message	From	References
0	Error	<p>Argument 3 (datatype) is not committed for transfer, call MPI_Type_commit before using the type for transfer!</p> <p>(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)})</p>	<p>Representative location:</p> <p>MPI_Send</p> <p>(1st occurrence)</p> <p>called from:</p> <p>#0</p> <p>main@test.c:17</p>	<p>References of a representative process:</p> <p>reference 1 rank 0:</p> <p>MPI_Type_contiguous</p> <p>(1st occurrence) called from:</p> <p>#0 main@test.c:14</p>

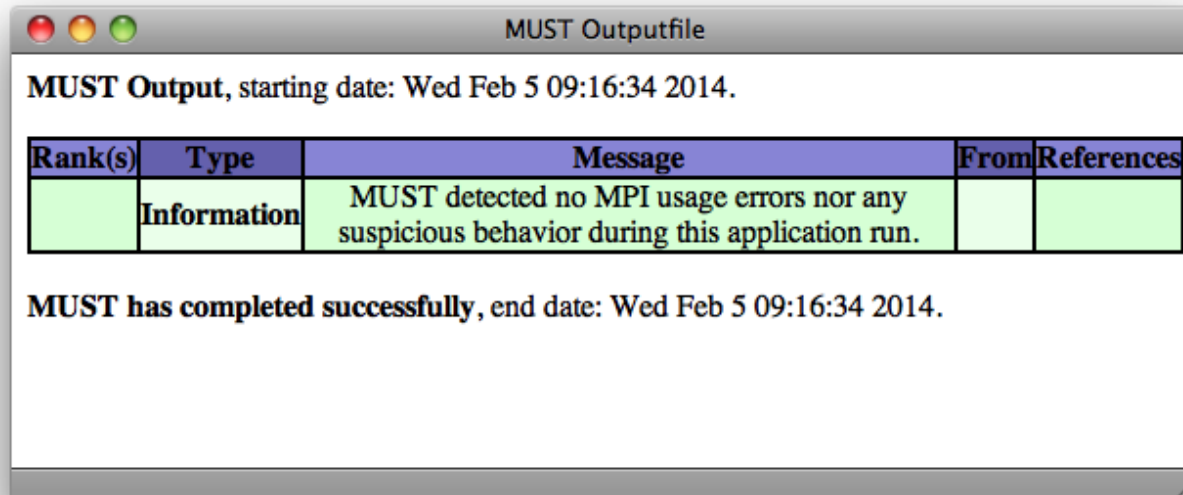
- Apply MUST with an mpiexec wrapper, that's it:

```
% 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 Outputfile

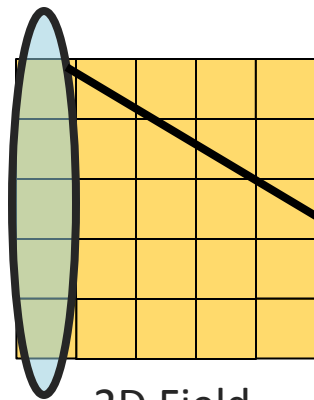
MUST Output, starting date: Wed Feb 5 09:16:34 2014.

Rank(s)	Type	Message	From	References
	Information	MUST detected no MPI usage errors nor any suspicious behavior during this application run.		

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:



2D Field
(of integers)

```
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

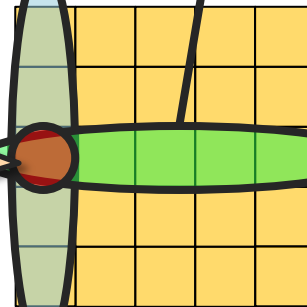
- C Code:

```
...  
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



2D Field
(of integers)

A Tool must:

- Detect the error
- Pinpoint the user to the exact problem

- 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)

Index within block

Block index

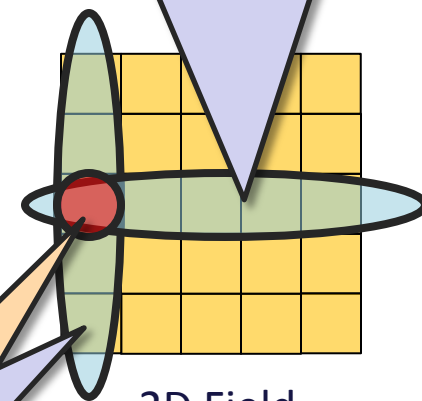
Count in communication call

Vector datatype to span a column

Error: buffer overlap

Contiguous datatype to span a row

2D Field
(of integers)



- 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)
(5)  //1) Create a datatype
(6)  MPI_Type_contiguous (2, MPI_INT, &newType);
(7)  MPI_Type_commit (&newType);
(8)
(9)  //2) Use MPI_Sendrecv to perform a ring communication
(10) MPI_Sendrecv (
(11)      sBuf, 1, newType, (rank+1)%size, 123,
(12)      rBuf, sizeof(int)*2, MPI_BYTE, (rank-1+size) % size, 123,
(13)      MPI_COMM_WORLD, &status);
(14)
(15) //3) Use MPI_Send and MPI_Recv to perform a ring communication
(16) MPI_Send (sBuf, 1, newType, (rank+1)%size, 456,
(17)                                     MPI_COMM_WORLD);
(17) MPI_Recv ( rBuf, sizeof(int)*2, MPI_BYTE, (rank-1+size) % size, 456,
(18)                                     MPI_COMM_WORLD, &status);
(18)
(19) 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
```

- First error: Overlap in Isend + Recv

Who?

What?

Where?

Details

Rank(s)	Type	Message	From	References
0	Error	<p>The memory regions to be transferred by this receive operation overlap with regions spanned by a pending non-blocking operation!</p> <p>(Information on the request associated with the other communication: Request activated at reference 1)</p> <p>(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)})</p> <p>The other communication overlaps with this communication at position:(vector)[2][0](MPI_INT)</p> <p>(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)})</p> <p>This communication overlaps with the other communication at position:(contiguous)[0](MPI_INT)</p> <p>A graphical representation of this situation is available in a detailed overlap view (MUST Output-files/MUST Overlap 0 0.html).</p>	<p>Representative location: MPI_Recv (1st occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:23</p>	<p>References of a representative process:</p> <p>reference 1 rank 0: MPI_Isend (1st occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:22</p> <p>reference 2 rank 0: MPI_Type_vector (1st occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:17</p> <p>reference 3 rank 0: MPI_Type_commit (2nd occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:19</p> <p>reference 4 rank 0: MPI_Type_contiguous (1st occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:16</p> <p>reference 5 rank 0: MPI_Type_commit (1st occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:18</p>

- 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!

These refer to the “References” (Details) column

(Information on the request associated with the other communication:

Request activated **reference 1)**

(Information on the datatype associated with the other communication:

Datatype created **reference 2** is for C, committed **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 **reference 4** is for C, committed **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 \(MUST Overlap.html\)](#)

References

References of a representative process:

reference 1 rank 0: **MPI_Isend** (1st occurrence) called from: 0
main@mpi_overlap_deadlock_errors.c:22

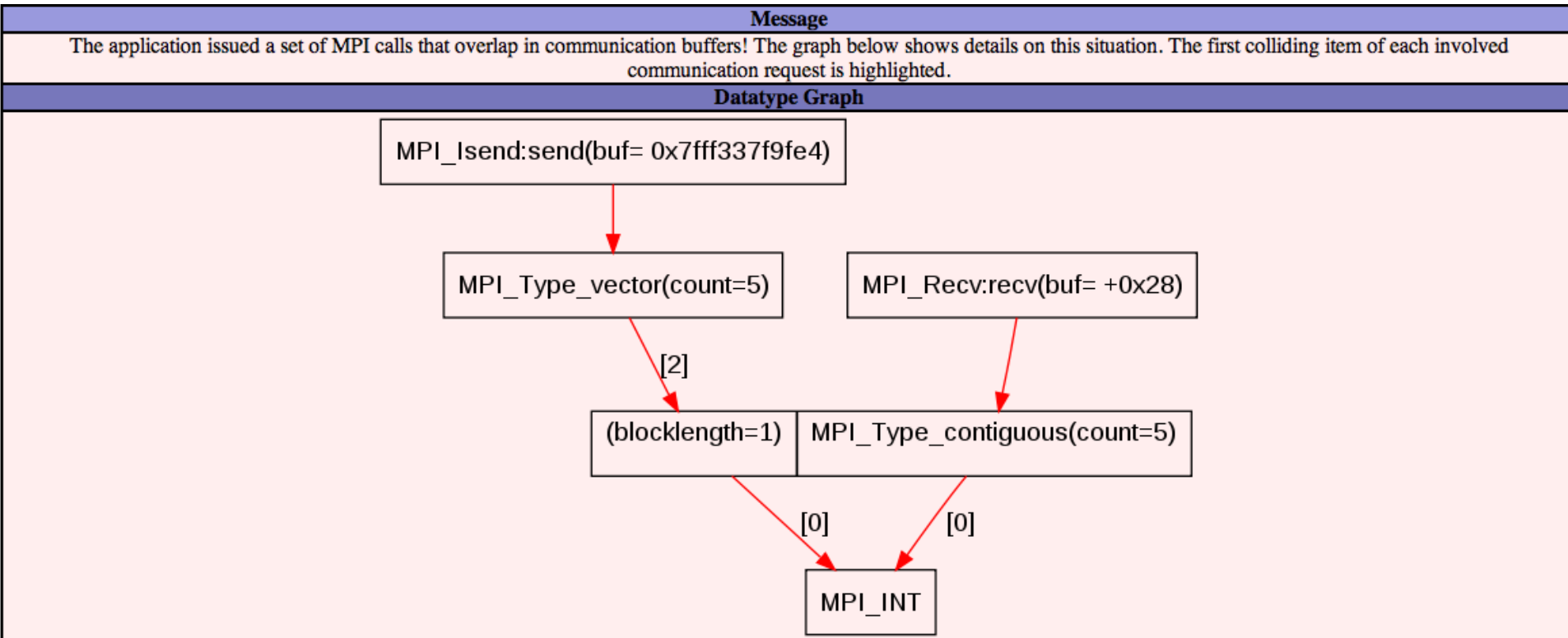
reference 2 rank 0: **MPI_Type_vector** (1st occurrence) called from: 0
main@mpi_overlap_deadlock_errors.c:17

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

- Visualization of overlap (MUST_Overlap.html):



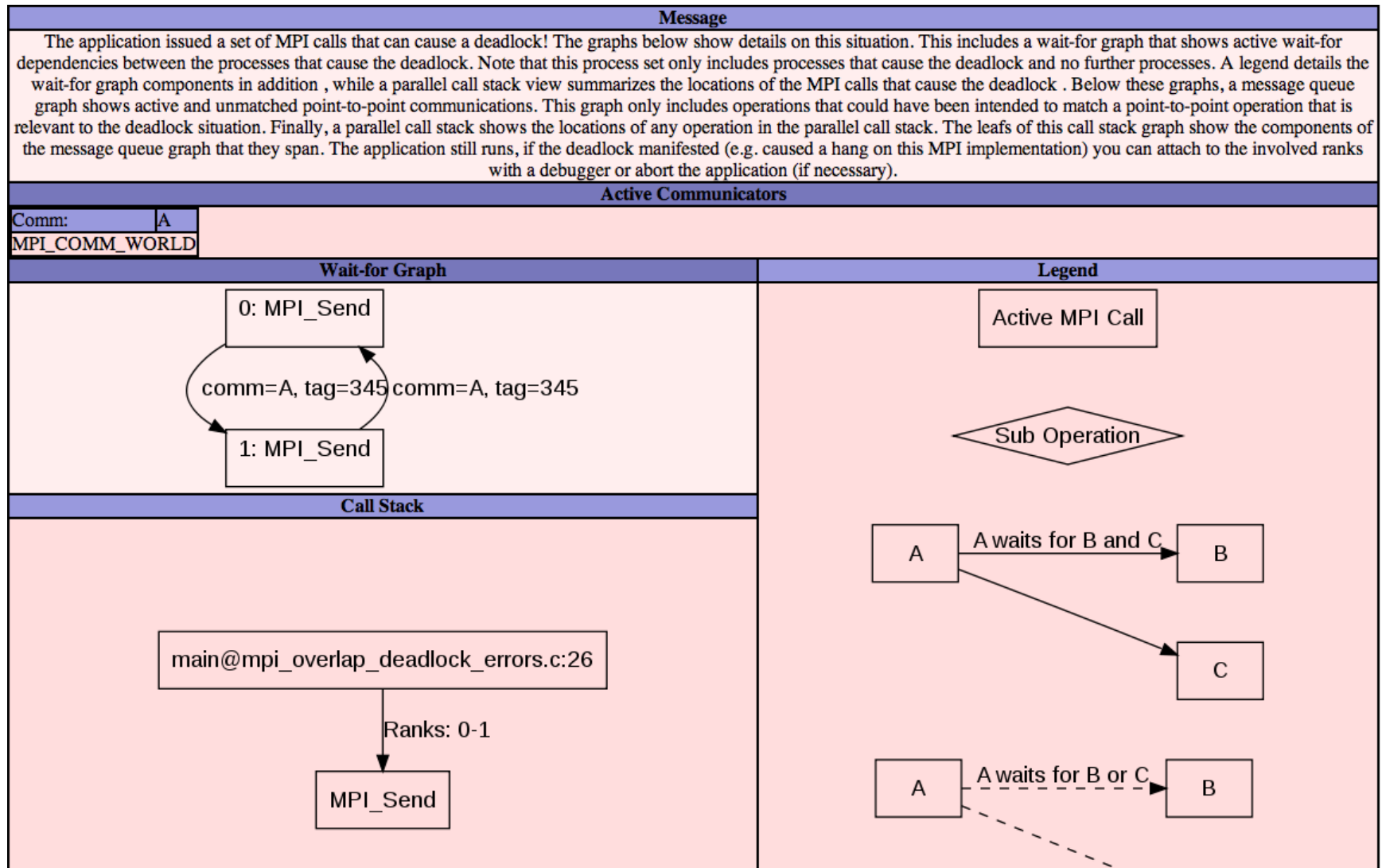
- Warning for unusual values, that match MPI specification:

Rank(s)	Type	Message	From
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

Rank(s)	Type	Message	From	References
	Error	<p>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).</p>		<p>References of a representative process:</p> <p>reference 1 rank 0: MPI_Send (1st occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:26</p> <p>reference 2 rank 1: MPI_Send (1st occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:26</p>

- Visualization of deadlock (MUST_Deadlock.html)



- Third error: Leaked resource (derived datatype)

Rank(s)	Type	Message	From	References
0-1	Error	<p>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:</p> <p>-Datatype 1: Datatype created at reference 1 is for C, committed 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)}</p> <p>-Datatype 2: Datatype created at reference 3 is for C, committed 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)}</p>	<p>Representative location: MPI_Type_contiguous (1st occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:16</p>	<p>References of a representative process:</p> <p>reference 1 rank 0: MPI_Type_contiguous (1st occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:16</p> <p>reference 2 rank 0: MPI_Type_commit (1st occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:18</p> <p>reference 3 rank 0: MPI_Type_vector (1st occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:17</p> <p>reference 4 rank 0: MPI_Type_commit (2nd occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:19</p>

- Fourth error: Leaked resource (request)
 - Leaked requests often indicate missing synchronization by MPI_Wait/Test

Rank(s)	Type	Message	From	References
0-1	Error	<p>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:</p> <p>-Request 1: Request activated at reference 1</p>	<p>Representative location: MPI_Isend (1st occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:22</p>	<p>References of a representative process: reference 1 rank 0: MPI_Isend (1st occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:22</p>

- 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 /*block length*/, 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)
(20) MPI_Finalize ();
```

Buffer overlap, first MPI_INT of the MPI_Recv overlaps with first MPI_INT in third block of MPI_Isend

- ```

(1) MPI_Init (&argc,&argv);
(2) comm = MPI_COMM_WORLD;
(3) MPI_Comm_rank (comm, &rank);
(4) MPI_Comm_size (comm, &size);
(5) MPI_Type_commit (&colType);
(6) MPI_Type_commit (&rowType);
(7) MPI_Send (arr, 1 /*blocklength*/, 5 /*stride*/, MPI_INT,
(8) rank-1, comm);
(9) MPI_Recv (arr, 1 /*blocklength*/, 5 /*stride*/, MPI_INT,
(10) rank, comm, &status);
(11) MPI_Type_commit (&colType);
(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,
(15) &status);
(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)
(20) MPI_Finalize ();

```
- User forgets to call an MPI\_Wait for the MPI request

- 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 pe
(13) MPI_Isend (&arr[0], 1, colType, (rank+1)
(14) MPI_Recv (&arr[10], 1, rowType, (rank+1), MPI_ANY_SOURCE, 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)
(20) 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)
(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) MPI_Send and MPI_Recv to perform a ring communication
(13) MPI_Send (arr, 0, MPI_INT, (rank+1)%size, 456, comm, &request);
(14) MPI_Recv (arr, 0, MPI_INT, (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)
(20) MPI_Finalize ();
```

Potential for deadlock, MPI\_Send can block (depends on MPI implementation and buffer size)

- 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 perform a ring communication
(17) MPI_Send (arr, 0, MPI_INT, (rank+1)%size, 456, comm);
(18) MPI_Recv (arr, 0, MPI_INT, (rank-1+size) % size, 456, comm,
 &status);
(19)
(20) MPI_Finalize ();
```

User forgot to free MPI Datatypes  
before calling MPI\_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 mpi77)
- Build:

```
% 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
```



- Go to bin directory

```
% cd bin
```

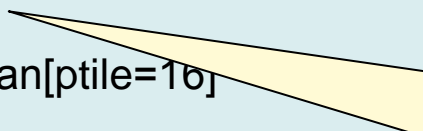
- Create and edit the jobscript

```
cp ../jobscript/marenosturm/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

- 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"!
```

- Open the MUST output: <Browser> MUST\_Output.html

| Rank(s) | Type    | Message                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|---------|---------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0-3     | Warning | <p>You requested 3 threads by OMP_NUM_THREADS=3 but the requested thread level MPI_THREAD_MULTIPLE is not supported by the mpi library but the library provides no thread support. This is ok as long as your application doesn't make use of OpenMP</p> <p>occurrence) called from:<br/>#0 MAIN_@bt.f:90<br/>#1 main@bt.f:319</p>                                                                                                                                                                                                                                                            |
| 0-3     | Error   | <p>There are 1 communicators 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 communicators:</p> <p>-Communicator 1: Communicator created at reference 1 size=4</p> <p>Representative location:<br/><b>MPI_Comm_split</b> (1st occurrence) called from:<br/>#0 MAIN_@bt.f:90<br/>#1 main@bt.f:319</p> <p>References of a representative process:<br/>reference 1 rank 2:<br/><b>MPI_Comm_split</b> (1st occurrence) called from:<br/>#0 MAIN_@bt.f:90<br/>#1 main@bt.f:319</p> |

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 ☺

| From                                                                                                                     |
|--------------------------------------------------------------------------------------------------------------------------|
| Representative location:<br><b>MPI_Init_thread</b> (1st occurrence) called from:<br>#0 MAIN_@bt.f:90<br>#1 main@bt.f:319 |
| Representative location:<br><b>MPI_Comm_split</b> (1st occurrence) called from:<br>#0 MAIN_@bt.f:90<br>#1 main@bt.f:319  |

| Rank(s) | Type        | Message                                                                                    | From | References |
|---------|-------------|--------------------------------------------------------------------------------------------|------|------------|
|         | Information | MUST detected no MPI usage errors nor any suspicious behavior during this application run. |      |            |

- 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