

MPI Runtime Error Detection with MUST

At the 17th VI-HPS Tuning Workshop

Joachim Protze
IT Center RWTH Aachen University
February 2015

How many issues 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.\n", 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
- 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

- 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
- Hands-on

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 safe version of tool

Fixed these errors:

```
#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.\n", rank, size);

    MPI_Finalize ();

    return 0;
}
```

Must detects deadlocks

Rank(s)	Type	Message	From	References
	Error	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).		References of a representative process: reference 1 rank 0: MPI_Recv (1st occurrence) called from: #0 main@example.c:15 reference 2 rank 1: MPI_Recv (1st occurrence) called from: #0 main@example.c:15

Click for graphical representation of the detected deadlock situation.

MUST Outputfile MUST Outputfile
file:///home/pj416018/MUST/example/MUST_Output-files/MUST_Deadlock.html

MUST Deadlock Details, date: Thu Nov 28 13:38:06 2013.

[Back to MUST error report](#)

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

Comm: A
MPI COMM WORLD

Wait-for Graph

```
graph TD; R0[0: MPI_Recv] -- "comm=A, tag=123" --> R1[1: MPI_Recv]; R1 -- "comm=A, tag=123" --> R0;
```

Rank 0 waits for rank 1 and vv.

Call Stack

```
graph TD; Main[main@example.c:15] -- "Ranks: 0-1" --> MPIRecv[MPI_Recv];
```

Simple call stack for this example.

Legend

Active MPI Call

Sub Operation

A A waits for B and C → B
A A waits for B and C → C

A A waits for B or C → B
A A waits for B or C → C

Active a **Pages: Overview**

Active and Relevant Point-to-Point Messages: Callstack-view

Fix1: use asynchronous receive

```
#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);

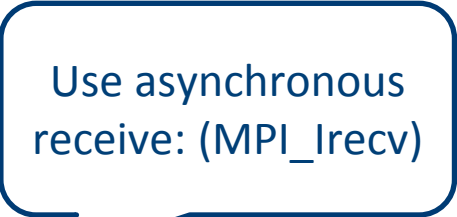
    MPI_Request request;
    MPI_Irecv (buf, 2, MPI_INT, size - rank - 1, 123, MPI_COMM_WORLD, &request);

    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

MUST Outputfile

file:///home/pj416018/MUST/example/MUST_Output.html

MUST Output, starting date: Thu Nov 28 13:50:48 2013.

Rank(s)	Type	Message	References
0	Error	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 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)	<p>Representative location: MPI_Send (1st occurrence) called from: #0 main@example-fix1.c:18</p> <p>References of a representative process: reference 1 rank 0: MPI_Send (1st occurrence) called from: #0 main@example-fix1.c:18 reference 2 rank 1: MPI_irecv (1st occurrence) called from: #0 main@example-fix1.c:16 reference 3 rank 0: MPI_Type_contiguous (1st occurrence) called from: #0 main@example-fix1.c:13</p>
0-1	Error	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})	<p>Representative location: MPI_Send (1st occurrence) called from: #0 main@example-fix1.c:18</p> <p>References of a representative process: reference 1 rank 1: MPI_Type_contiguous (1st occurrence) called from: #0 main@example-fix1.c:13</p>
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 Fortran, based on the following type(s): { MPI_INTEGER}) This communication overlaps with the other communication at position:(contiguous)[0](MPI_INTEGER) A graphical representation of this situation is available in a detailed overlap view (MUST_Output-files/MUST_Overlap_0_0.html) .	<p>Representative location: MPI_Send (1st occurrence) called from: #0 main@example-fix1.c:18</p> <p>References of a representative process: reference 1 rank 0: MPI_irecv (1st occurrence) called from: #0 main@example-fix1.c:16 reference 2 rank 0: MPI_Type_contiguous (1st occurrence) called from: #0 main@example-fix1.c:13</p>
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 Fortran, based on the following type(s): { MPI_INTEGER})	<p>Representative location: MPI_Send (1st occurrence) called from: #0 main@example-fix1.c:18</p> <p>References of a representative process: reference 1 rank 1: MPI_irecv (1st occurrence) called from: #0 main@example-fix1.c:16 reference 2 rank 1: MPI_Type_contiguous (1st occurrence) called from:</p>

Use of uncommitted datatype: **type**

Fix2: use MPI_Type_commit

```
#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



MUST Outputfile
file:///home/pj416018/MUST/example/MUST_Output.html
MUST Output, starting date: Thu Nov 28 13:51:42 2013.

Rank(s)	Type	Message	References
0	Error	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)	References of a representative process: reference 1 rank 0: MPI_Send (1st occurrence) called from: #0 main@example-fix2.c:19 reference 2 rank 1: MPI_Irecv (1st occurrence) called from: #0 main@example-fix2.c:17 Representative location: MPI_Send (1st occurrence) called from: #0 main@example-fix2.c:19 reference 3 rank 0: MPI_Type_contiguous (1st occurrence) called from: #0 main@example-fix2.c:13 reference 4 rank 0: MPI_Type_commit (1st occurrence) called from: #0 main@example-fix2.c:14
1	Error	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)	References of a representative process: reference 1 rank 1: MPI_Send (1st occurrence) called from: #0 main@example-fix2.c:19 reference 2 rank 0: MPI_Irecv (1st occurrence) called from: #0 main@example-fix2.c:17 Representative location: MPI_Send (1st occurrence) called from: #0 main@example-fix2.c:19 reference 3 rank 1: MPI_Type_contiguous (1st occurrence) called from: #0 main@example-fix2.c:13 reference 4 rank 1: MPI_Type_commit (1st occurrence) called from: #0 main@example-fix2.c:14
		The memory regions to be transferred by this send operation overlap with regions spanned by a pending non-blocking receive operation!	References of a representative process: reference 1 rank 0: MPI_Irecv (1st occurrence) called from: #0 main@example-fix2.c:17

Fix3: use same message size for send and receive

```
#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

MUST Outputfile

file:///home/pj416018/MUST/example/MUST_Output.html

MUST Output, starting date: Mon Dec 2 13:11:12 2013.

Rank(s)	Type	Message	References
1	Error	<p>A send and a receive operation use datatypes that do not match! Mismatch occurs at (contiguous)[0](MPI_INTEGER) 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_1.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)</p>	<p>Representative location: MPI_Send (1st occurrence) called from: #0 main@example-fix3.c:19</p> <p>reference 1 rank 1: MPI_Send (1st occurrence) called from: #0 main@example-fix3.c:19</p> <p>reference 2 rank 0: MPI_Irecv (1st occurrence) called from: #0 main@example-fix3.c:17</p> <p>reference 3 rank 1: MPI_Type_contiguous (1st occurrence) called from: #0 main@example-fix3.c:13</p> <p>reference 4 rank 1: MPI_Type_commit (1st occurrence) called from: #0 main@example-fix3.c:14</p>
0	Error	<p>A send and a receive operation use datatypes that do not match! Mismatch occurs at (contiguous)[0](MPI_INTEGER) 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)</p>	<p>Representative location: MPI_Send (1st occurrence) called from: #0 main@example-fix3.c:19</p> <p>reference 1 rank 0: MPI_Send (1st occurrence) called from: #0 main@example-fix3.c:19</p> <p>reference 2 rank 1: MPI_Irecv (1st occurrence) called from: #0 main@example-fix3.c:17</p> <p>reference 3 rank 0: MPI_Type_contiguous (1st occurrence) called from: #0 main@example-fix3.c:13</p> <p>reference 4 rank 0: MPI_Type_commit (1st occurrence) called from: #0 main@example-fix3.c:14</p>

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

Fix4: use C-datatype constants in C-code

```
#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;
}
```

Use the integer
datatype intended
for usage in C

MUST detects data races in asynchronous communication

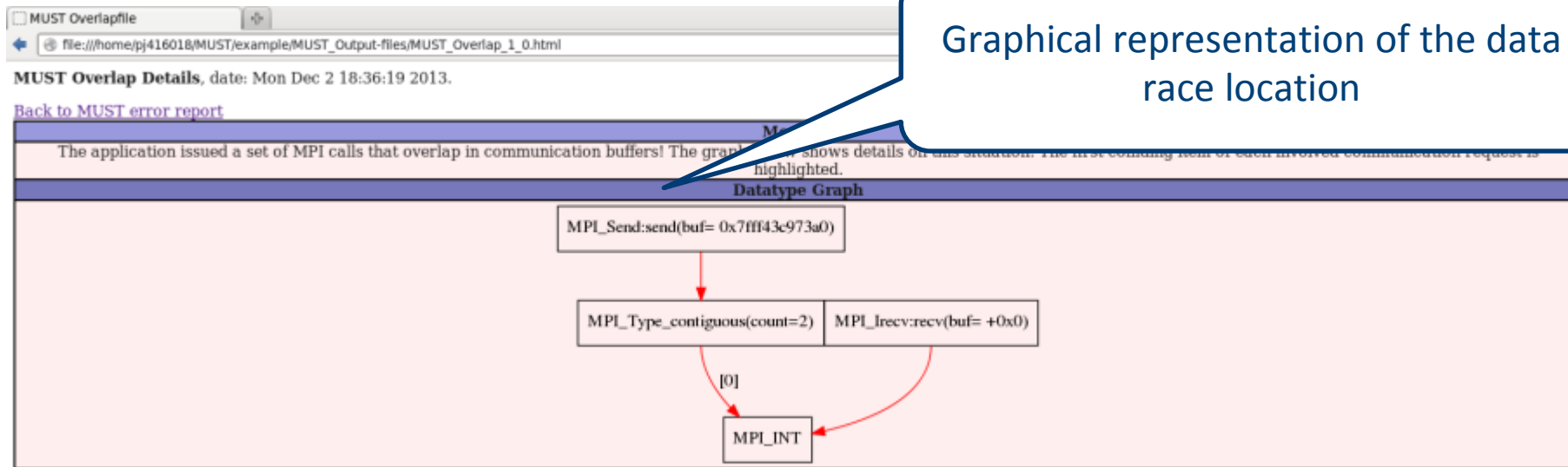
Data race between send and asynchronous receive operation

MUST Outputfile
file:///home/pj416018/MUST/example/MUST_Output.html

MUST Output, starting date: Mon Dec 2 18:36:19 2013.

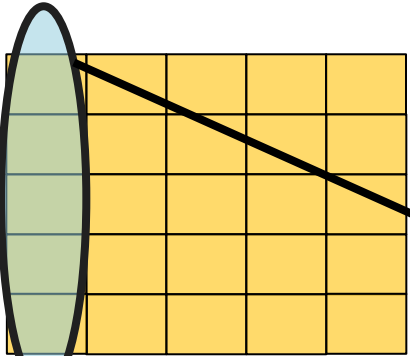
Rank(s)	Type	Message		
1	Error	<p>The memory regions to be transferred by this send operation overlap with regions spanned by a pending non-blocking receive operation!</p> <p>(Information on the request associated with the other communication: Request activated at reference 1)</p> <p>(Information on the datatype associated with the other communication: MPI_INT)</p> <p>The other communication overlaps with this communication at position:(MPI_INT)</p> <p>(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})</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_1_0.html).</p>	<p>Representative location: MPI_Send (1st occurrence) called from: #0 main@example-fix4.c:19</p>	<p>References of a representative process:</p> <p>reference 1 rank 1: MPI_irecv (1st occurrence) called from: #0 main@example-fix4.c:17</p> <p>reference 2 rank 1: MPI_Type_contiguous (1st occurrence) called from: #0 main@example-fix4.c:13</p> <p>reference 3 rank 1: MPI_Type_commit (1st occurrence) called from: #0 main@example-fix4.c:14</p>
0-1	Error	<p>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:</p> <p>-Datatype 1: Datatype created at reference 1 is for C, committed at reference 2, based on the following type(s): { MPI_INT}</p>	<p>Representative location: MPI_Type_contiguous (1st occurrence) called from: #0 main@example-fix4.c:13</p>	<p>References of a representative process:</p> <p>reference 1 rank 1: MPI_Type_contiguous (1st occurrence) called from: #0 main@example-fix4.c:13</p> <p>reference 2 rank 1: MPI_Type_commit (1st occurrence) called from: #0 main@example-fix4.c:14</p>
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_irecv (1st occurrence) called from: #0 main@example-fix4.c:17</p>	<p>References of a representative process:</p> <p>reference 1 rank 1: MPI_irecv (1st occurrence) called from: #0 main@example-fix4.c:17</p>
0	Error	<p>The memory regions to be transferred by this send operation overlap with regions spanned by a pending non-blocking receive 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: MPI_INT)</p> <p>The other communication overlaps with this communication at position:(MPI_INT)</p> <p>(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})</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_Send (1st occurrence) called from: #0 main@example-fix4.c:19</p>	<p>References of a representative process:</p> <p>reference 1 rank 0: MPI_irecv (1st occurrence) called from: #0 main@example-fix4.c:17</p> <p>reference 2 rank 0: MPI_Type_contiguous (1st occurrence) called from: #0 main@example-fix4.c:13</p> <p>reference 3 rank 0: MPI_Type_commit (1st occurrence) called from: #0 main@example-fix4.c:14</p>

Graphical representation of the race condition



Errors with MPI Datatypes – Overview

- 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

Errors with MPI Datatypes – Example

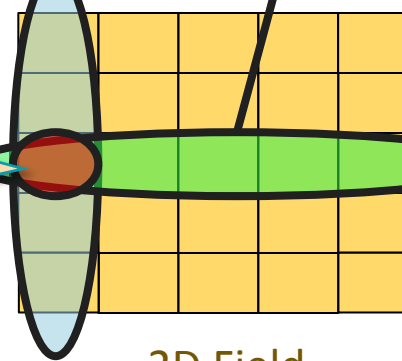
■ C Code:

```
...  
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



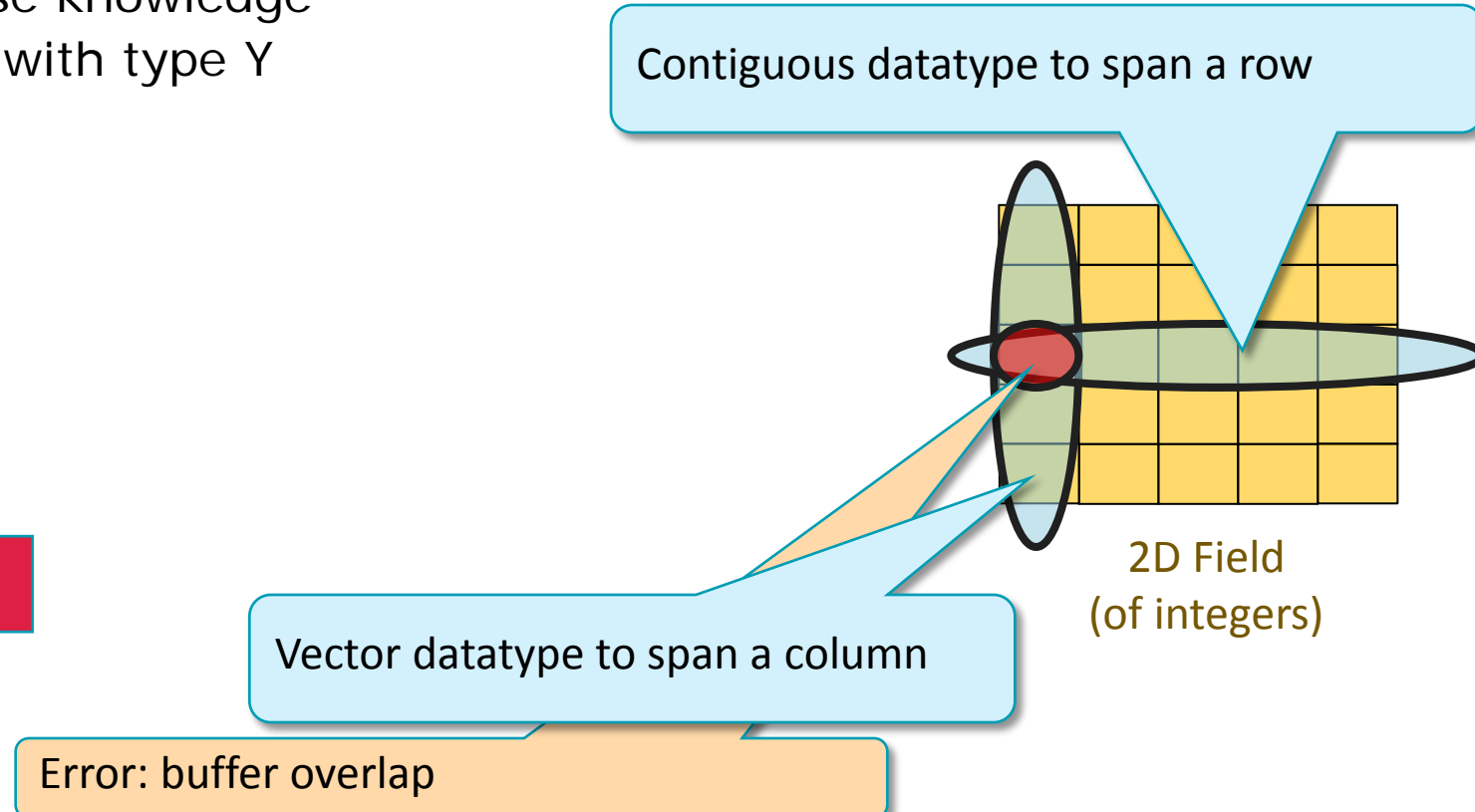
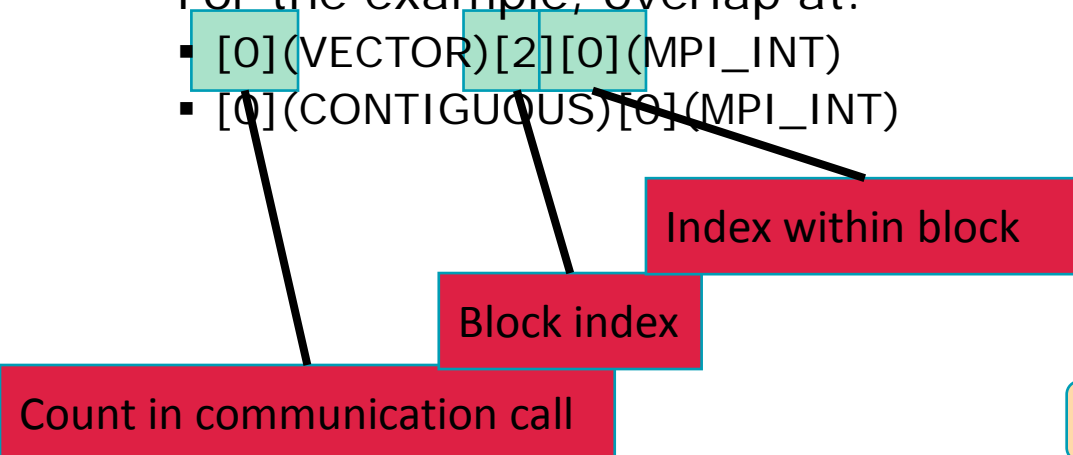
2D Field
(of integers)

A Tool must:

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

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)`



Fix5: use independent memory regions

```
#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;
}
```

Offset points to
independent
memory

MUST detects leaks of user defined objects

MUST Outputfile

file:///home/pj416018/MUST/example/MUST_Output.html

MUST Output, starting date: Thu Nov 28 13:55:26 2013.

Rank(s)	Type	Message	From	References
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}	Representative location: MPI_Type_contiguous (1st occurrence) called from: #0 main@example-fix5.c:13	References of a representative process: reference 1 rank 0: MPI_Type_contiguous (1st occurrence) called from: #0 main@example-fix5.c:13 reference 2 rank 0: MPI_Type_commit (1st occurrence) called from: #0 main@example-fix5.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	Representative location: MPI_Irecv (1st occurrence) called from: #0 main@example-fix5.c:17	References of a representative process: reference 1 rank 0: MPI_Irecv (1st occurrence) called from: #0 main@example-fix5.c:17

MUST has completed successfully, end date: Thu Nov 28 13:55:26 2013.

Leak of user defined
datatype object

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

Fix6: Deallocate datatype object

```
#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

MUST Outputfile

file:///home/pj416018/MUST/example/MUST_Output.html

MUST Output, starting date: Thu Nov 28 13:55:49 2013.

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_irecv (1st occurrence) called from: #0 main@example-fix6.c:17</p>	<p>References of a representative process: reference 1 rank 0: MPI_irecv (1st occurrence) called from: #0 main@example-fix6.c:17</p>

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

Remaining unfinished
asynchronous receive

Fix7: use MPI_Wait to finish asynchronous communication

```
#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

Finally



MUST Outputfile

file:///home/pj416018/MUST/example/MUST_Output.html

MUST Output, starting date: Thu Nov 28 13:56:03 2013.

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: Thu Nov 28 13:56:03 2013.

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
- Hands-on

Tool Overview – Approaches Techniques

- 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);
```

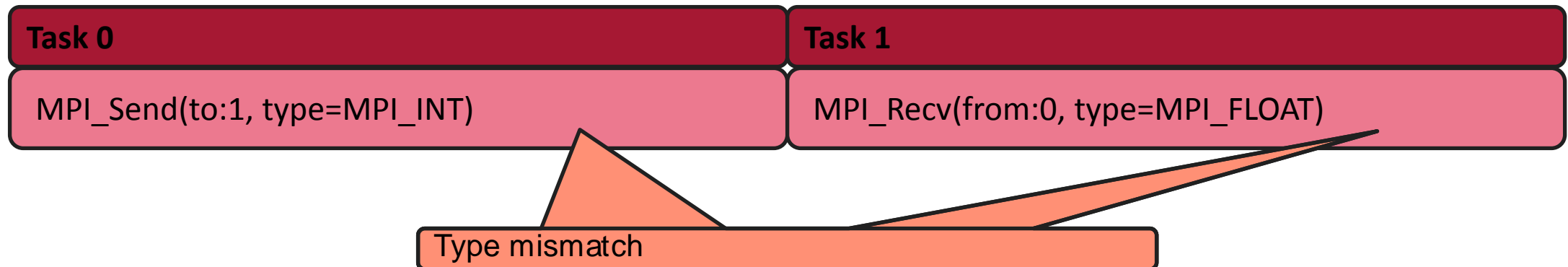
“-1” instead of “MPI_ANY_SOURCE”

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

Only works with less than 1024 tasks

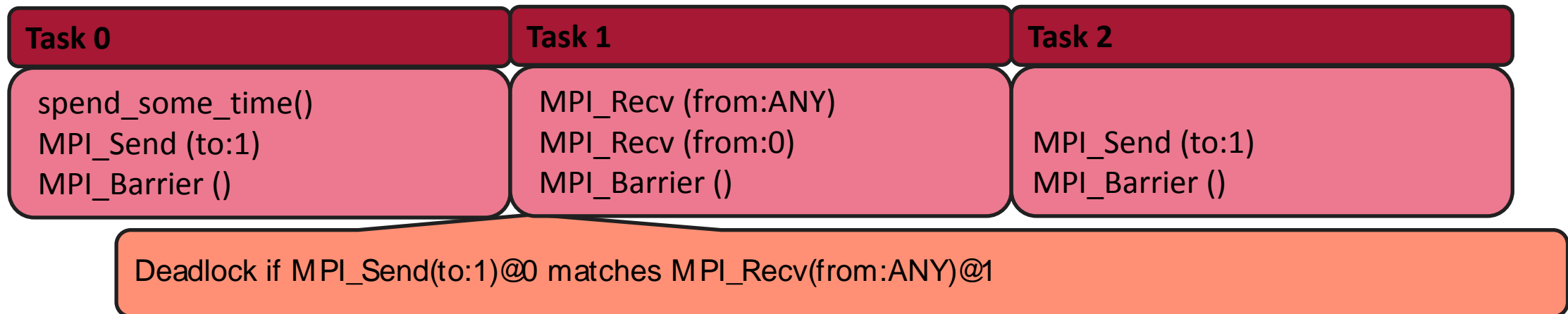
Tool Overview – Approaches Techniques (2)

- 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

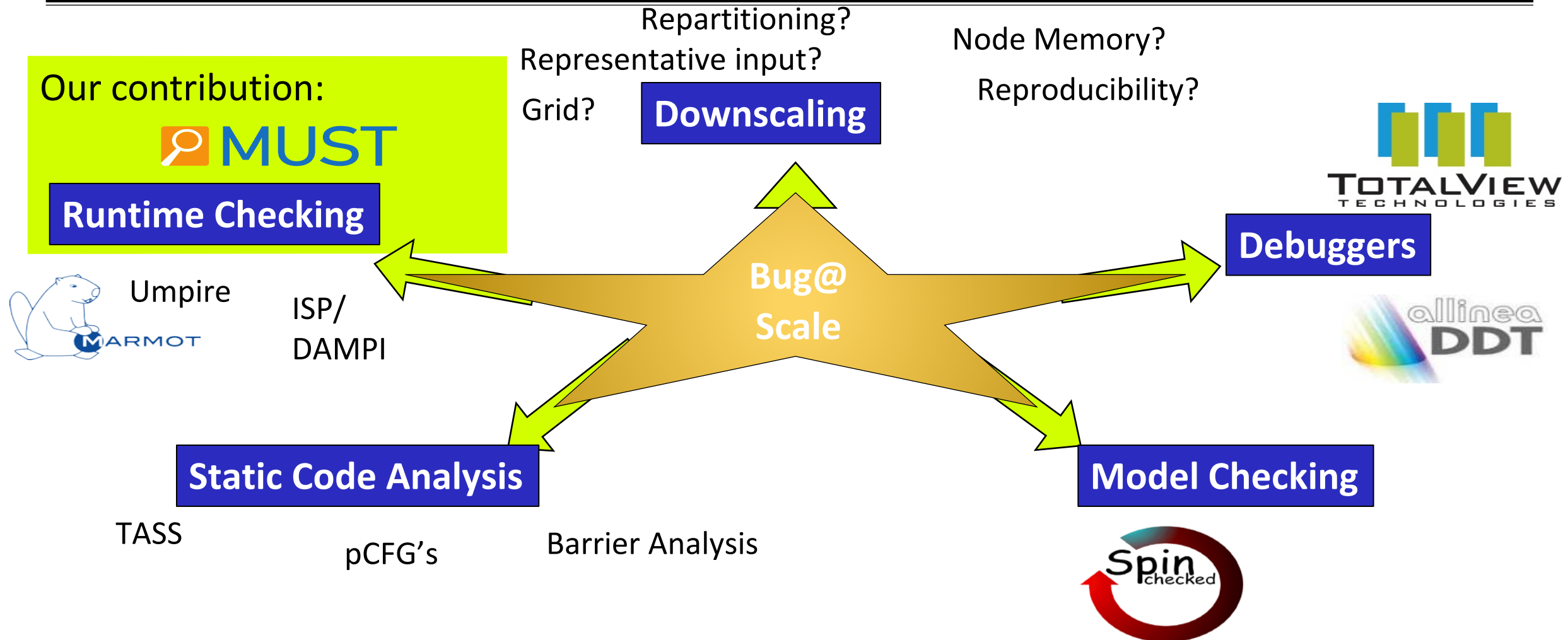


Tool Overview – Approaches Techniques (3)

- 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



Approaches to Remove Bugs (Selection)



Content

- Motivation
- MPI usage errors
- Examples: Common MPI usage errors
 - Including MUST's error descriptions
- Correctness tools
- **MUST usage**
- Hands-on



MUST – Overview

- MPI runtime error detection tool
- Open source (BSD license)
<http://www.itc.rwth-aachen.de/MUST/>
- Wide range of checks, strength areas:
 - Overlaps in communication buffers
 - Errors with derived datatypes
 - Deadlocks
- Largely distributed, able to scale with the application

MUST – Basic Usage

- 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)

MUST – Usage on frontend - backend machines

- Prepare on frontend

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

```
% mpicc -g source.c -o exe  
% mustrun --must:mpiexec mpiexec -np 4 ./exe \  
  --must:mode prepare
```

- Submit a job to the backend:

```
% mustrun --must:mpiexec mpiexec -np 4 ./exe \  
  --must:mode run
```

- Open MUST_Output.html with a browser

MUST – At Scale (highly recommended for >10 processes)

- Provide a branching factor (fan-in) for the tree infrastructure:

```
% mustrun --must:mpiexec mpiexec -np 4 ./exe --must:fanin 8
```

- Get info about the number of processes:

```
% mustrun --must:mpiexec mpiexec -np 4 ./exe --must:fanin 8 \  
  --must:info
```

→ This will give you the number of processes needed with tool attached

Content

- Motivation
- MPI usage errors
- Examples: Common MPI usage errors
 - Including MUST's error descriptions
- Correctness tools
- MUST usage
- Hands-on

Hands On – Build for MUST

- Go into the NPB directory
- Edit config/make.def
- Disable any other tool (i.e. use mpif77, unset PREP)
- Use intel or gnu tool chain
- Build:

```
COMPFLAGS = -openmp -g -extend-source -dynamic # intel
```

```
...
```

```
MPIF77 = ftn
```

```
% module switch PrgEnv-cray/5.2.40 PrgEnv-intel
```

```
% module use ~xhjp/.modules
```

```
% module load must-intel
```

```
% make clean
```

```
% make bt-mz NPROCS=6 CLASS=C
```

```
=====
```

```
=   NAS PARALLEL BENCHMARKS 3.3   =
```

```
=   MPI + OpenMP Multi-Zone Versions   =
```

```
=   F77                               =
```

```
=====
```

```
cd BT-MZ; make CLASS=C NPROCS=6
```

```
make[1]: Entering directory
```

```
...
```

```
ftn -O3 -g -openmp -extend-source -o ../bin/bt-mz_C.6
```

```
bt_scorep_user.o ...
```


Hands On – Prepare MUST

- Go to the bin directory:

```
% cd bin
```

- On machines with dedicated serial frontend nodes and backend nodes you need to prepare MUST on the frontend

```
% mustrun -n 4 -d 6 ./bt-mz_C.4 --must:mode prepare
[MUST] MUST configuration ... centralized checks with fall-back
application crash handling (very slow)
[MUST] Information: overwriting old intermediate data in directory
"/zhome/academic/HLRS/xhp/xhpjp/NPB3.3-MZ-MPI/bin/must_temp"!
[MUST] Weaver ... success
[MUST] Code generation ... success
[MUST] Build file generation ... success
[MUST] Configuring intermediate build ... success
[MUST] Building intermediate sources ... success
[MUST] Installing intermediate modules ... success
[MUST] Generating P^nMPI configuration ... success
%
```

Hands On - Prepare Job

- Create and edit the jobscript

```
cp ../jobscript/hornet/run.pbs ./must.pbs  
vim must.pbs
```

- Jobscript:

```
#PBS -q R_vihps04  
...  
module switch PrgEnv-cray PrgEnv-intel  
module use ~xhpjp/.modules  
module load must-intel  
...  
export OMP_NUM_THREADS=6  
CLASS=C  
NPROCS=6  
...  
mustrun --must:mpiexec aprun -n $NPROCS -d $OMP_NUM_THREADS $EXE --must:mode run
```

MUST needs one extra process!
We use 6 processes * 6 threads + 1 tool
process

Hands On – Executing with MUST

- Submit the jobscript:

```
qsub must.pbs
```

- Job output should read:

```
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"!
```

BT – MUST Results

- Open the MUST output: <Browser> MUST_Output.html

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

BT-MZ should evaluate the "provided" thread level and don't use threads.

Resource leak:
A communicator created with MPI_Comm_split is not freed

Stacktraces in MUST

- 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
 - Nevertheless, the must-intel module is built without stacktrace support
- Ifort compiled FORTRAN applications lead to segfault:
 - Use MUST w/o stacktraces for fortran applications
 - Use GNU compiler to build your application and use MUST w/ stacktraces
- Supposed your application has no faults you won't need stacktraces ☺

From
Representative location: MPI_Init_thread (1st occurrence) called from: #0 MAIN_@bt.f:90 #1 main@bt.f:319
Representative location: MPI_Comm_split (1st occurrence) called from: #0 MAIN_@bt.f:90 #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.		

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

Thank You