MPI Runtime Error Detection with MUST

At the 30th VI-HPS Tuning Workshop

Joachim Protze
IT Center RWTH Aachen University
January 2019

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

Error more obvious

- 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

Content

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

```
$ cp -r ~nct00009/tutorial/must ~/must-examples
$ source ~/must-examples/load-must.sh
```

Hands-on (part 1)

- Compile with debugflag
- Load must module
- Run interactive debug session

```
$ cd ~/must-examples
$ mpicc -g example.c
$ . ./load-must
$ salloc -t 00:10:00 -n 5 -J debug srun --pty /bin/bash
```

- Launch with mustrun
- Abort with Ctrl + C
- Open MUST_Output.html in browser ("links")

```
$ mustrun –np 4 ./a.out
[MUST] MUST configuration ... centralized checks with fall-back
application crash handling (very slow)
[MUST] Using prebuilt infrastructure at …/modules/mode1-layer2
…
[MUST] Executing application:
============MUST===============
ERROR: MUST detected a deadlock, detailed information is available in
the MUST output file. You should either investigate details with a
debugger or abort, the operation of MUST will stop from now.
===============================
^C
[MUST] Execution finished, inspect "…/MUST_Output.html"!
$ links MUST_Output.html
```

```
$ cp -r ~nct00009/tutorial/must ~/must-examples
$ source ~/must-examples/load-must.sh
```

Hands-on (part 1)

- Fix the identified issue(s)
- Recompile

```
$ vim example.c
$ mpicc -g example.c
```

Some MPI implementations check for errors and abort.

- Launch with mustrun

```
$ mustrun -np 4 ./a.out
[MUST] MUST configuration ... centralized ch     with fall-back
application crash handling (very slow)
[MUST] Using prebuilt infrastructure     …/modules/mode1-layer2
…
[MUST] Executing application:
Invalid datatype, error stack:
MPI_Send(201): MPI_Send(buf=0x7fffe77c58cc, count=2,
dtype=USER<contig>, dest=1, tag=123, comm=0x84000004) failed
MPI_Send(135): Datatype has not been committed
…
Waiting up to 30 seconds for analyses to be finished.
^C
[MUST] Execution finished, inspect "…/MUST_Output.html"!
$ links MUST_Output.html
```

- Abort with Ctrl + C
- Open MUST_Output.html in browser ("links")
- Start over

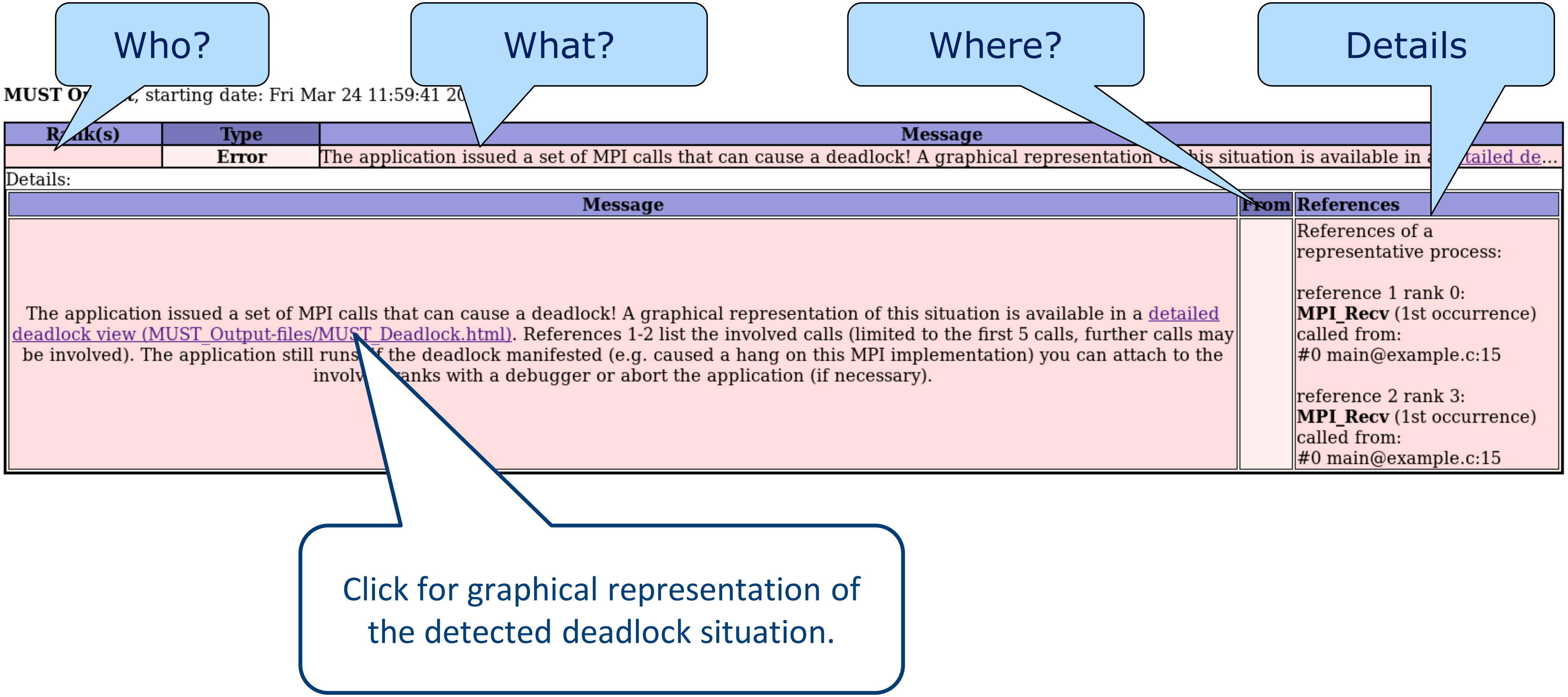
Content

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

Already fixed missing MPI_Init/Finalize:

```
#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



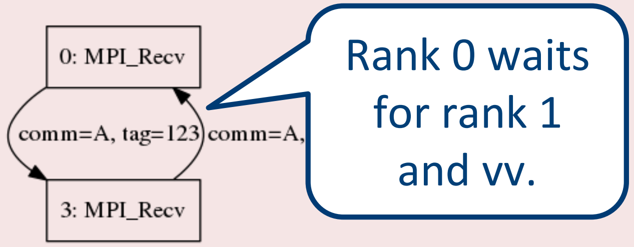
Message

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).

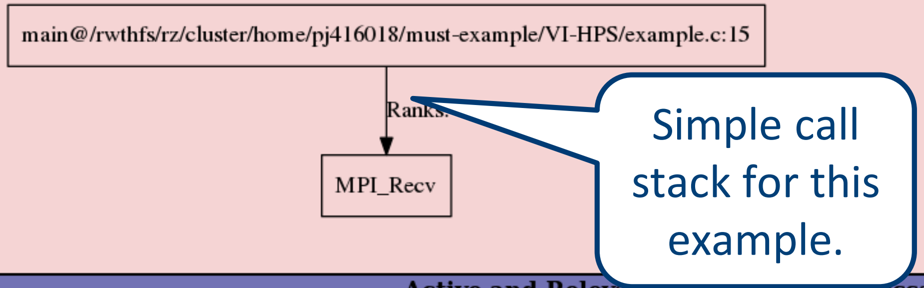
Active Communicators

Comm:	A
MPI_COMM_WORLD	

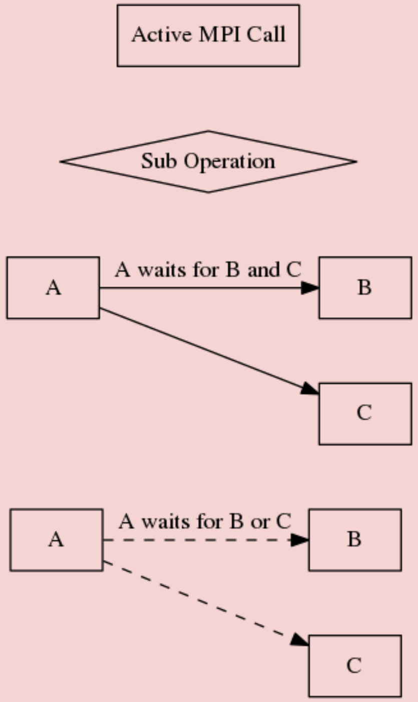
Wait-for Graph



Call Stack



Legend



Active and Relevant Point-to-Point Messages: Overview

Message queue

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

stack

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 transfer buffer sizes / types

Rank(s)	Type	
2(28793)	Error	A receive operation uses a (datatype ... by the send it matches! The first element of the send...

Details:

Message	From	References
A receive operation uses a (datatype,count) pair that can not hold the data transfered 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, based on the following type(s): { MPI_INTEGER}) (Information on receive of count 2 with type:MPI_INT)	Representative location: **MPI_Send** (1st occurrence) called from: #0 main@example-fix1.c:18	References of a representative process: reference 1 rank 2: **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 2: **MPI_Type_contiguous** (1st occurrence) called from: #0 main@example-fix1.c:13

1(28792)	Error	A receive operation uses a (datatype,count) pair that can not hold the data transfered by the send it matches! The first element of the send...
0-3	Error	Argument 3 (datatype) is not commited for transfer, call MPI_Type_commit before using the type for transfer!(Information on datatypeData...
2(28793)	Error	The memory regions to be transfered by this send operation overlap with regions spanned by a pending non-blocking receive operation!(In...
1(28792)	Error	The memory regions to be transfered by this send operation overlap ... receive operation!(In...
3(28795)	Error	The memory regions to be transfered by this send ... receive operation!(In...
3(28795)	Error	A receive operation uses a (datatype,count) pair that can not ... t element of the send...
0(28794)	Error	The memory regions to be transfered by this send operation overla ... receive operation!(In...
0(28794)	Error	A receive operation uses a (datatype,count) pair that can not hold t ... t element of the send...

Size of sent message larger than receive buffer

All detected errors are collapsed for overview – click to expand

Fix2: 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 errors in handling datatypes

Rank(s)	Type	Message
2(17250)	Error	A send and a receive operation use datatypes that do not match! Mismatch occurs at (contiguous)[0](MPI_INTEGER) in the send type and a...

Details:

Message	From	References
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_74088185856002.html)](MUST_Output-files/MUST_Typemismatch_74088185856002.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, based on the following type(s): { MPI_INTEGER}) (Information on receive of count 2 with type:MPI_INT)	Representative location: **MPI_Send** (1st occurrence) called from: #0 main@example-fix2.c:18	References of a representative process: reference 1 rank 2: **MPI_Send** (1st occurrence) called from: #0 main@example-fix2.c:18 reference 2 rank 1: **MPI_Irecv** (1st occurrence) called from: #0 main@example-fix2.c:16 reference 3 rank 2: **MPI_Type_contiguous** (1st occurrence) called from: #0 main@example-fix2.c:13

Rank(s)	Type	Message
0(17249)	Err	that do not m ...](MPI_INTEGER) in the send type and a...
1(17248)	Err	that do not m ...](MPI_INTEGER) in the send type and a...
3(17251)	Err	that do not ...](MPI_INTEGER) in the send type and a...
0-3	Err	ansfer, cal ... transfer!(Information on datatypeData...

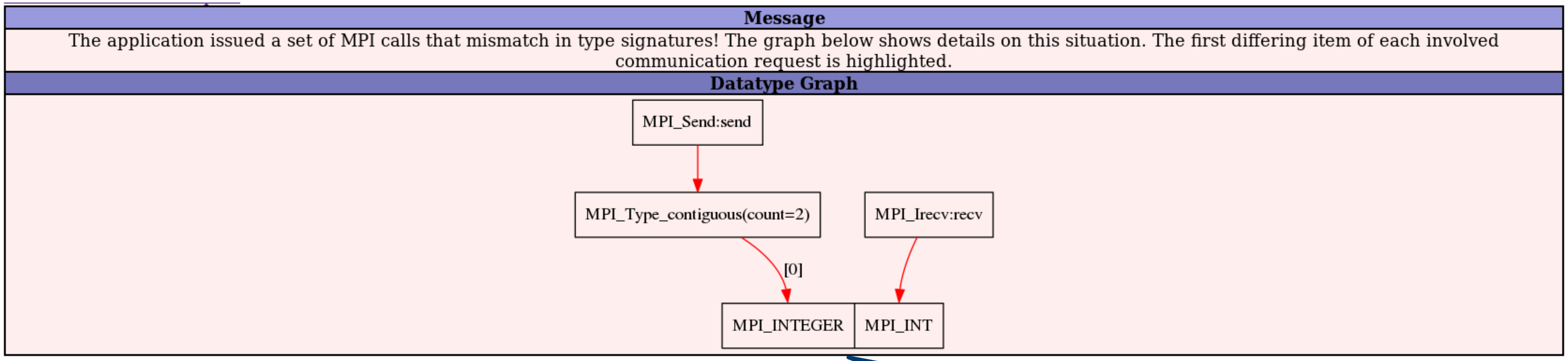
Details:

Message	From	References
Argument 3 (datatype) is not commited for transfer, call MPI_Type_commit before using the type for transfer! (Information on datatypeDatatype created at reference 1 is for Fortran, based on the following type(s): { MPI_INTEGER})	Representative location: **MPI_Send** (1st occurrence) called from: #0 main@example-fix2.c:18	References of a representative process: reference 1 rank 2: **MPI_Type_contiguous** (1st occurrence) called from: #0 main@example-fix2.c:13

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

Use of uncommited datatype: `type`

MUST detects errors in handling datatypes



Graphical representation of the type missmatch

Fix3: use MPI_Type_commit
Fix4: use C integer type

```
#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, 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 the integer datatype intended for usage in C

Commit the datatype before usage

MUST detects data races in asynchronous communication

Data race between send and ascynchronous receive operation

Rank(s)	Type	
0(1605)	Error	The memory regions to be transfered by thi … receive operation!(In...

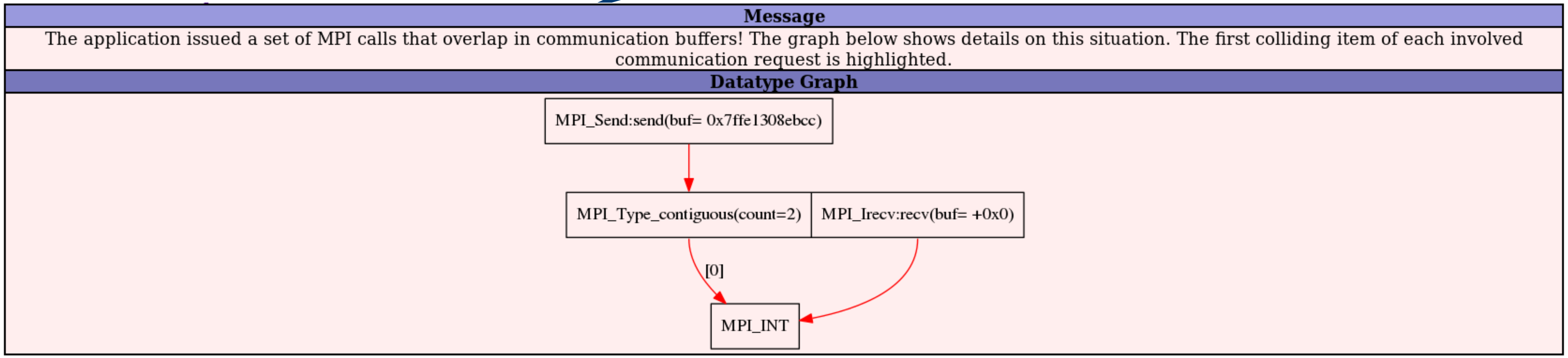
Details:

Message	From	References
The memory regions to be transfered by this send operation overlap with regions spanned by a pending non-blocking receive operation! (Information on the request associated with the other communication: Point-to-point request activated at reference 1) (Information on the datatype associated with the other communication: MPI_INT) The other communication overlaps with this communication at position:(MPI_INT) (Information on the datatype associated with this communication: Datatype created at reference 2 is for C, commited at reference 3, based on the following type(s): { MPI_INT}) This communication overlaps with the other communication at position:(contiguous)[0](MPI_INT) A graphical representation of this situation is available in a detailed overlap view (MUST_Output-files/MUST_Overlap_6893422510080_0.html).	Representative location: **MPI_Send** (1st occurrence) called from: #0 main@example-fix3.c:19	References of a representative process: reference 1 rank 0: **MPI_Irecv** (1st occurrence) called from: #0 main@example-fix3.c:17 reference 2 rank 0: **MPI_Type_contiguous** (1st occurrence) called from: #0 main@example-fix3.c:13 reference 3 rank 0: **MPI_Type_commit** (1st occurrence) called from: #0 main@example-fix3.c:14

3(1610)	Error	The memory regions to be transfered by this send operation overlap with regions spanned by a pending non-blocking receive operation!(In...
2(1608)	Error	The memory regions to be transfered by this send operation overlap with regions spanned by a pending non-blocking receive operation!(In...
1(1606)	Error	The memory regions to be transfered by this send operation overlap with regions spanned by a pending non-blocking receive operation!(In...
0-3	Error	There are 1 datatypes that are not freed when MPI_Finalize was issued, a quality application should free all MPI resources before calling ...
0-3	Error	There are 1 requests that are not freed when MPI_Finalize was issued, a quality application should free all MPI resources before calling M...

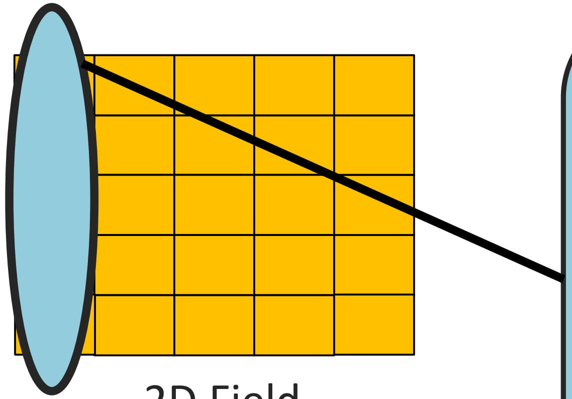
Graphical representation of the race condition

Graphical representation of the data race location



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

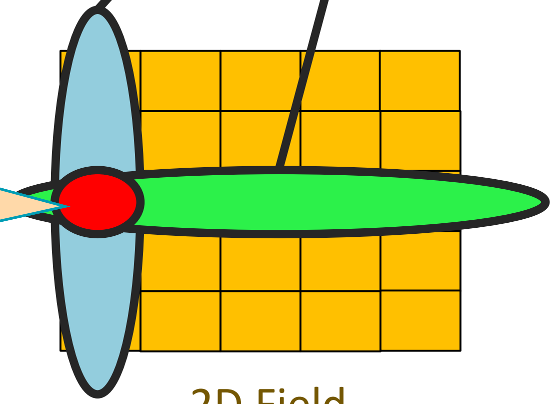
- C Code:

```
...
MPI_Isend(buf, 1 /
         MPI_COMM_WORLD, &request);
MPI_Recv(buf, 1 /
         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

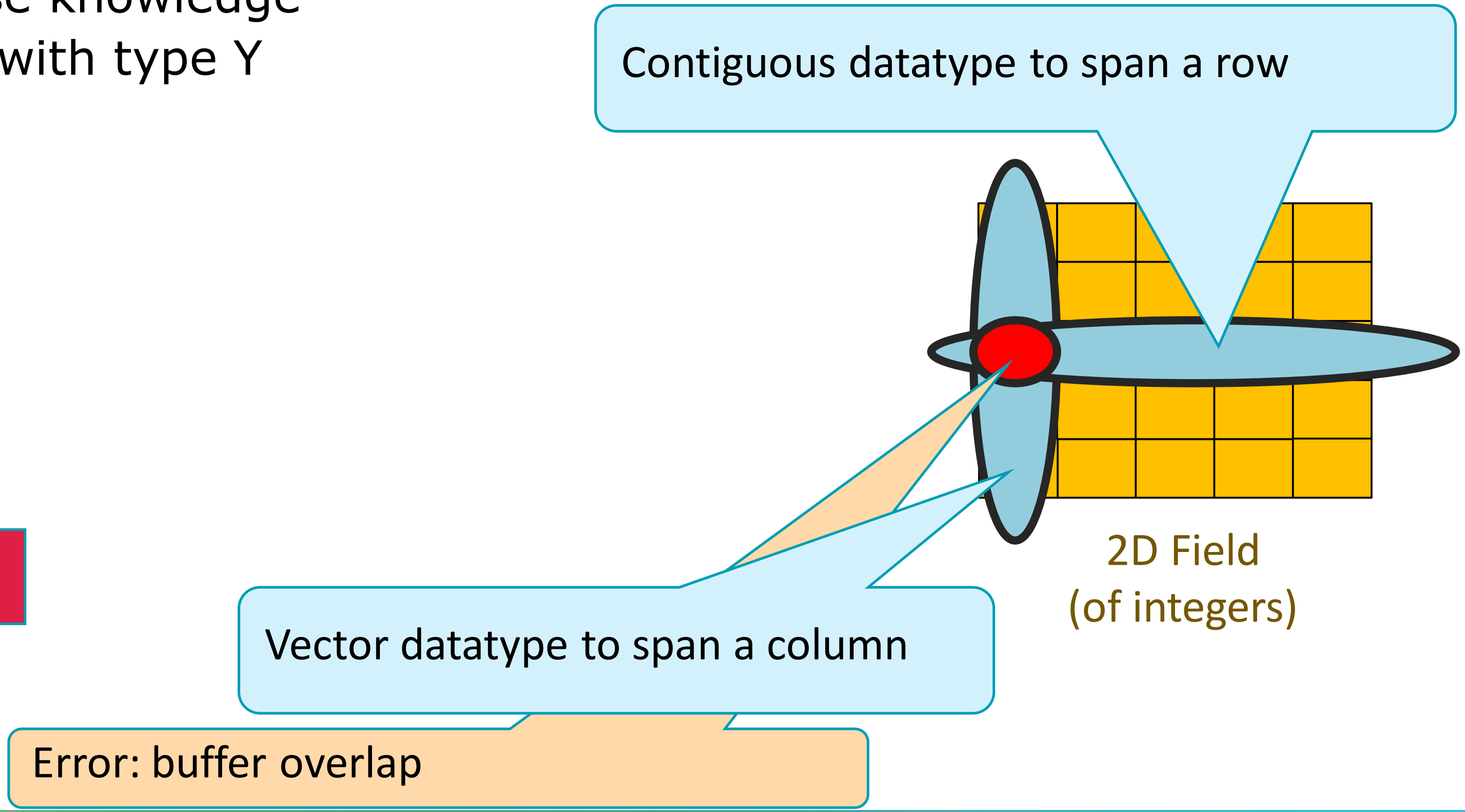
Errors with MPI Datatypes – Error Positions

- 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)

Index within block

Block index

Count in communication call



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

Rank(s)	Type	Message
0-3	Error	There are 1 datatypes that are not freed when MPI_Finalize was issued, a quality application should free all MPI resources before calling ...

Details:

Message	From	References
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, commited at reference 2, based on the following type(s): { MPI_INT}	Representative location: **MPI_Type_contiguous** (1st occurrence) called from: #0 main@example-fix4.c:13	References of a representative process: reference 1 rank 1: **MPI_Type_contiguous** (1st occurrence) called from: #0 main@example-fix4.c:13 reference 2 rank 1: **MPI_Type_commit** (1st occurrence) called from: #0 main@example-fix4.c:14

Rank(s)	Type	Message
0-3	Error	There are 1 requests that are not freed when MPI_Finalize was issued, a qual[illegible]pplication should free all MPI resources before calling M...

Details:

Message	From	References
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: Point-to-point request activated at reference 1	Representative locat[illegible] **MPI_Irecv** (1st occurrence[illegible] called from: #0 main@example-fix4.c:17	References of a representative [illegible]rocess: re[illegible] rank 1: **MPI_Irecv** (1st occur[illegible]led from: #0 main@[illegible]4.c:17

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

Unfinished non-blocking receive is resource leak and missing synchronization

Leak of user defined datatype object

Fix6: Deallocate datatype object
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

Deallocate the created datatype

Finally

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

Details:

Message	From	References
MUST detected no MPI usage errors nor any suspicious behavior during this application run.		

No further error detected

Hopefully this message applies to many applications

Content

- Motivation
- MPI usage errors
- Hands-on
- 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:
  - Can find hidden errors
  - 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);
```

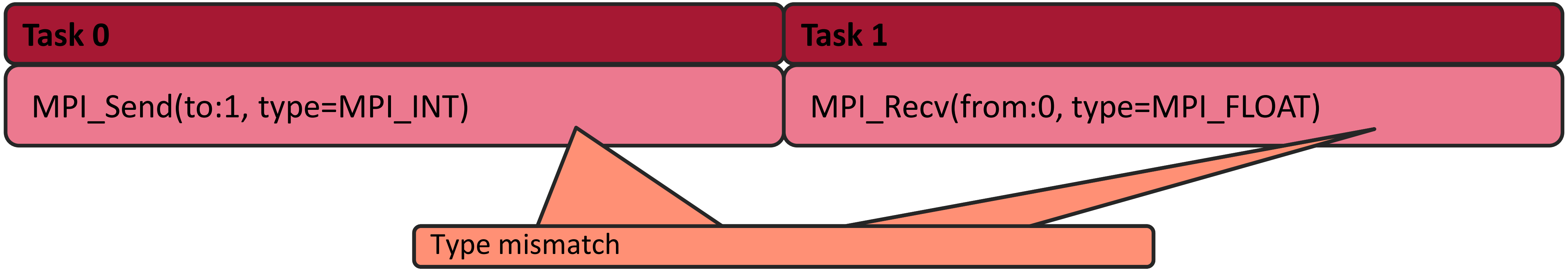
"-1" instead of "MPI_ANY_SOURCE"

```
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

Task 0	Task 1	Task 2
spend_some_time() MPI_Send (to:1) MPI_Barrier ()	MPI_Recv (from:ANY) MPI_Recv (from:0) MPI_Barrier ()	MPI_Send (to:1) MPI_Barrier ()

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

Content

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

MUST – Basic Usage

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

```
% mpicc source.c –o exe
% $MPIRUN –n 4 ./exe
```

```
% mpicc -g source.c –o exe
% mustrun --must:mpiexec $MPIRUN –n 4 ./exe
```

or simply

```
% mustrun –n 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 slower (details later)

MUST – Usage on frontend - backend machines

- Compile and run using a batch script

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

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

- If you see messages about missing dot on the backend, run on frontend:

```
% mustrun --must:dot
```

- 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 -np 40 ./exe --must:fanin 8
```

- Get info about the number of processes:

```
% mustrun -np 40 ./exe --must:fanin 8 --must:info
```

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

MUST – Multithreading Support

- By default, MUST supports up to MPI_THREAD_FUNNELED
- For higher threading levels:

```
% mustrun -np 40 ./exe --must:hybrid
```

- This will raise the required level to MPI_THREAD_MULTIPLE!
- Get info about the resources needed:

```
% mustrun -np 40 ./exe --must:hybrid --must:info
```

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

MUST – Summary

- 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

Thank You