TECHNISCHE UNIVERSITÄT DRESDEN

MUST

Center for Information Services and High Performance Computing (ZIH)

MPI Correctness Checking with MUST

Parallel Programming Course, Dresden, 8.- 12. February 2016

Mathias Korepkat (mathias.korepkat@tu-dresden.de

Matthias Lieber (matthias.lieber@tu-dresden.de)

Tobias Hilbrich (tobias.hilbrich@tu-dresden.de)

Joachim Protze (protze@rz.rwth-aachen.de)

ZIH

Center for Information Services & High Performance Computing

Motivation

- MPI programming is error prone
- Portability errors
  (just on some systems, just for some runs)
- Behaviour of an application run:
  - Crash
  - Application hanging
  - Finishes

- Questions:
  - Why crash/hang?
  - Is my result correct?
  - Will my code also give correct results on another system?

Motivation (2)

- C code:

Fortran type in C

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

Use of uncommited type

- Tools:
  - Runtime correctness tools can detect such errors
  - Strength of such tools:
    - o Test for conformance to 600+ page MPI standards
    - o Understand complex calls, e.g., MPI_Alltoallw with:
      - 9 Arguments, including 5 comm sized arrays

MUST – Overview

- MPI runtime error detection tool
- Open source (BSD license)
  https://doc.itc.rwth-aachen.de/display/CCP/Project+MUST
- Wide range of checks, strength areas:
  - Overlaps in communication buffers
  - Errors with derived datatypes
  - Deadlocks
- Largely distributed, can scale with the application

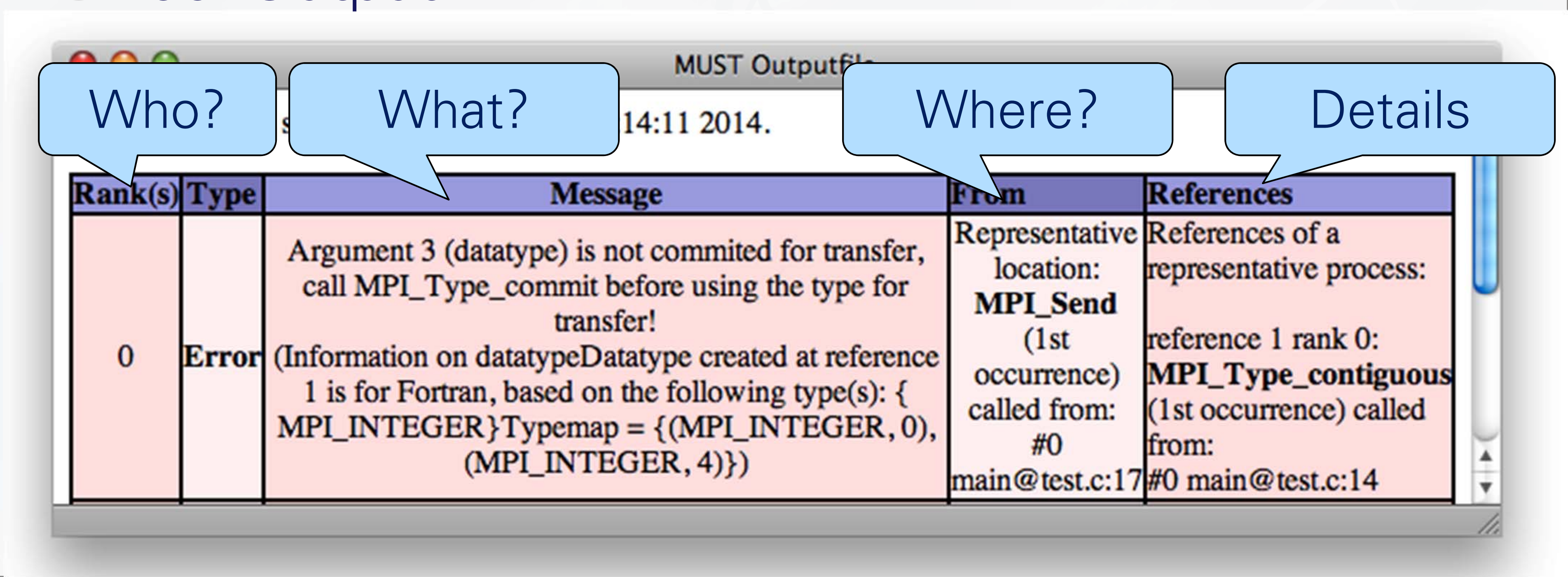
MUST – Correctness Reports

- C code:

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

Use of uncommitted type

- Tool Output:



Rank(s)	Type	Message	From	References
0	Error	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}Typemap = {(MPI_INTEGER, 0), (MPI_INTEGER, 4)})	Representative location: **MPI_Send** (1st occurrence) called from: #0 main@test.c:17	References of a representative process: reference 1 rank 0: **MPI_Type_contiguous** (1st occurrence) called from: #0 main@test.c:14

MUST – Basic Usage

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

```
% mpicc source.c –o exe
% mpirun –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 – With your Code

- Chances are good that you will get:

MUST Output, starting date: Tue Feb 3 08:20:33 2015.

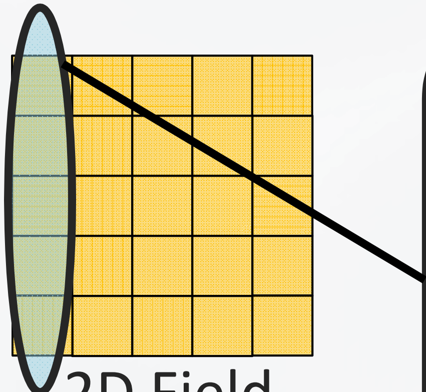
Rank(s)	Type	Message
	Information	MUST detected no MPI usage errors nor any suspicious behavior during this application run.

MUST has completed successfully, end date: Tue Feb 3 08:20:38 2015.

- 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

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

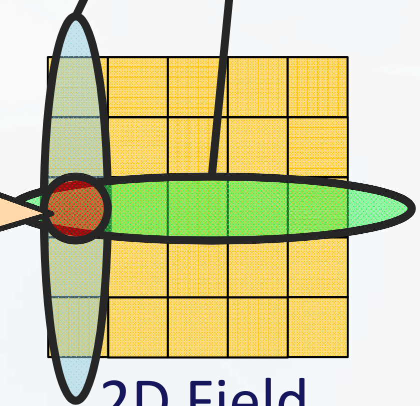
- C code:

```
...
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)

MUST detects the error and pinpoints the user to the exact problem

TECHNISCHE UNIVERSITÄT DRESDEN

ZIH Center for Information Services & High Performance Computing

MUST Usage Example

- Example "mpi_overlap_deadlock_errors.c" :

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

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

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

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

MPI_Finalize ();
```

MUST Usage Example – Apply the Tool

- Runs without any apparent issue with OpenMPI
- Are there any errors?
- Verify with MUST:

```
% mpicc -g mpi_overlap_deadlock_errors.c \
        -o mpi_errors
% mustrun -np 2 mpi_errors
% firefox MUST_Output.html
```

MUST Usage Example – Error 1 Buffer Overlap

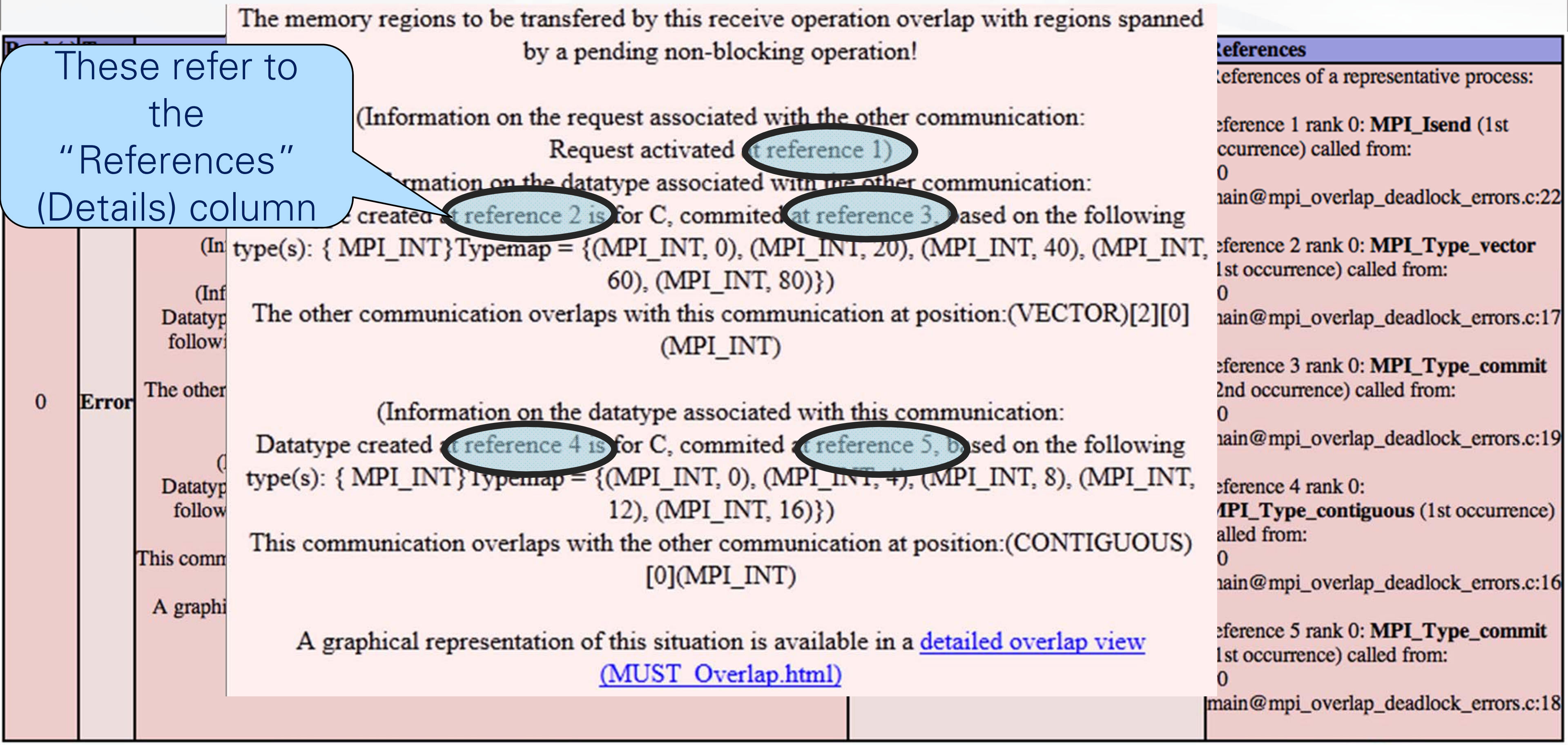
- First error: Overlap in Isend + Recv

Who? What? Where? Details

Rank(s)	Type	Message	From	References
0	Error	The memory regions to be transfered by this receive operation overlap with regions spanned by a pending non-blocking operation! (Information on the request associated with the other communication: Request activated at reference 1) (Information on the datatype associated with the other communication: Datatype created at reference 2 is for C, commited at reference 3, based on the following type(s): { MPI_INT}Typemap = {(MPI_INT, 0), (MPI_INT, 20), (MPI_INT, 40), (MPI_INT, 60), (MPI_INT, 80)}) The other communication overlaps with this communication at position:(vector)[2][0](MPI_INT) (Information on the datatype associated with this communication: Datatype created at reference 4 is for C, commited at reference 5, based on the following type(s): { MPI_INT}Typemap = {(MPI_INT, 0), (MPI_INT, 4), (MPI_INT, 8), (MPI_INT, 12), (MPI_INT, 16)}) This communication overlaps with the other communication at position:(contiguous)[0](MPI_INT) A graphical representation of this situation is available in a detailed overlap view (MUST_Output-files/MUST_Overlap_0_0.html).	Representative location: **MPI_Recv** (1st occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:23	References of a representative process: reference 1 rank 0: **MPI_Isend** (1st occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:22 reference 2 rank 0: **MPI_Type_vector** (1st occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:17 reference 3 rank 0: **MPI_Type_commit** (2nd occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:19 reference 4 rank 0: **MPI_Type_contiguous** (1st occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:16 reference 5 rank 0: **MPI_Type_commit** (1st occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:18

MUST Usage Example – Error 1 Buffer Overlap

- First error: Overlap in Isend + Recv



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

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

(Information on the datatype associated with this communication:
Datatype created at reference 4 is for C, commited at reference 5, based on the following type(s): { MPI_INT}Typemap = {(MPI_INT, 0), (MPI_INT, 4), (MPI_INT, 8), (MPI_INT, 12), (MPI_INT, 16)})
This communication overlaps with the other communication at position:(CONTIGUOUS)[0](MPI_INT)

A graphical representation of this situation is available in a [detailed overlap view (MUST_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

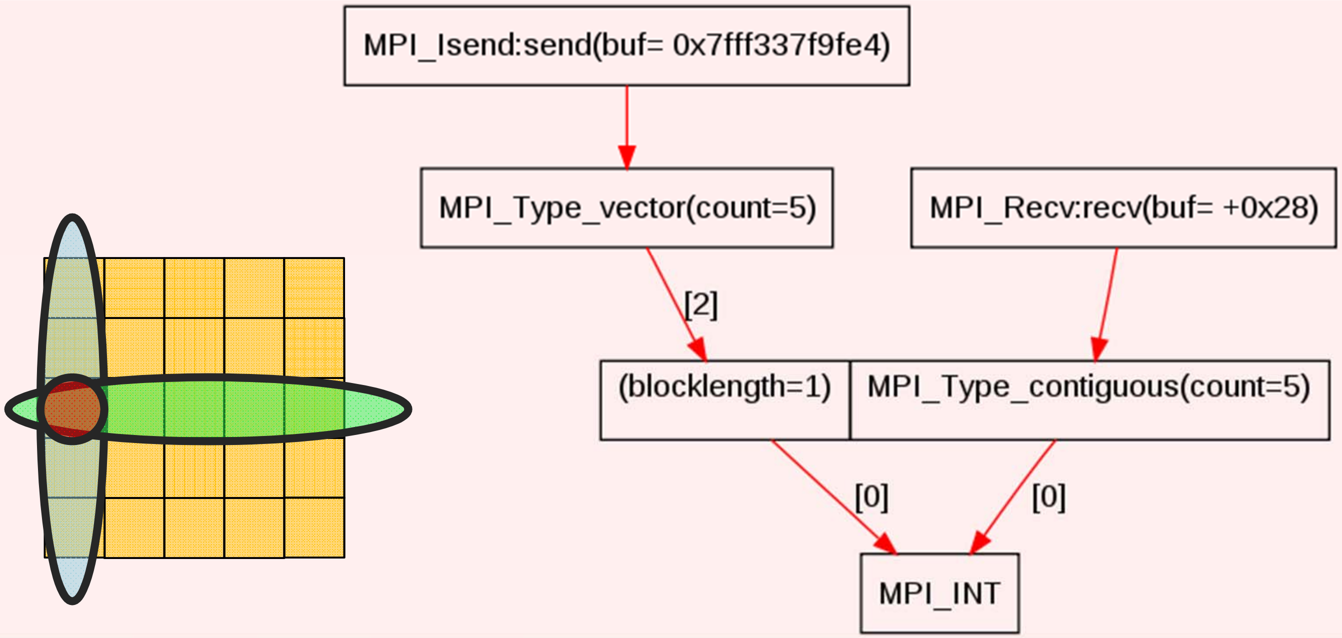
Reference 5 rank 0: **MPI_Type_commit** (1st occurrence) called from:
0
main@mpi_overlap_deadlock_errors.c:18

These refer to the "References" (Details) column

MUST – Example (4)

- Visualization of overlap (MUST_Overlap.html):

Message
The application issued a set of MPI calls that overlap in communication buffers! The graph below shows details on this situation. The first colliding item of each involved communication request is highlighted.
Datatype Graph



MUST Usage Example – Warning 1 "Count 0"

- 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

MUST Usage Example – Error 2 Deadlock

- Second Error: potential Deadlock

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_Send** (1st occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:26 reference 2 rank 1: **MPI_Send** (1st occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:26

TECHNISCHE UNIVERSITÄT DRESDEN

ZIH Center for Information Services & High Performance Computing

MUST Usage Example – Error 2 Deadlock (2)

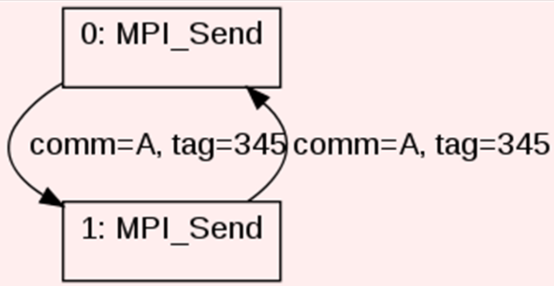
- Visualization of deadlock (MUST_Deadlock.html)

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

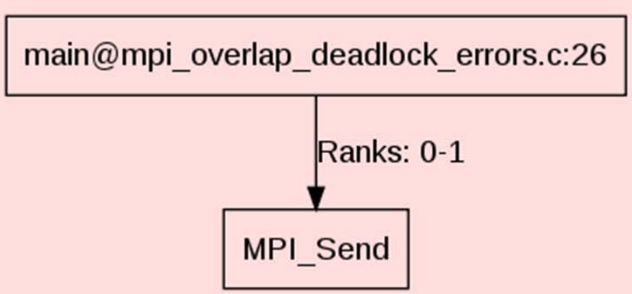
Active Communicators

Comm:	A
MPI_COMM_WORLD	

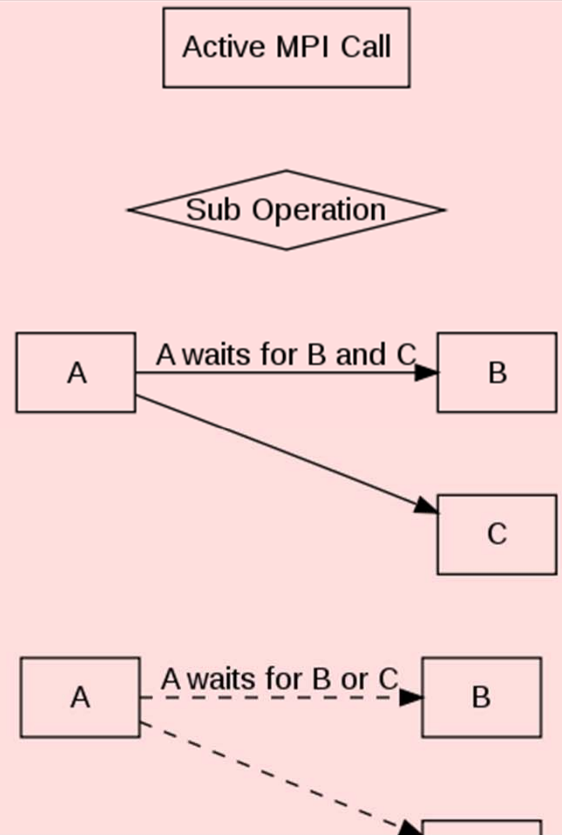
Wait-for Graph



Call Stack



Legend



MUST Usage Example – Error 3 Type Leak

- Third error: Leaked resource (derived datatype)

Rank(s)	Type	Message	From	References
0-1	**Error**	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: -Datatype 1: Datatype created at reference 1 is for C, commited at reference 2, based on the following type(s): { MPI_INT}Typemap = {(MPI_INT, 0), (MPI_INT, 4), (MPI_INT, 8), (MPI_INT, 12), (MPI_INT, 16)} -Datatype 2: Datatype created at reference 3 is for C, commited at reference 4, based on the following type(s): { MPI_INT}Typemap = {(MPI_INT, 0), (MPI_INT, 20), (MPI_INT, 40), (MPI_INT, 60), (MPI_INT, 80)}	Representative location: **MPI_Type_contiguous** (1st occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:16	References of a representative process: reference 1 rank 0: **MPI_Type_contiguous** (1st occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:16 reference 2 rank 0: **MPI_Type_commit** (1st occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:18 reference 3 rank 0: **MPI_Type_vector** (1st occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:17 reference 4 rank 0: **MPI_Type_commit** (2nd occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:19

MUST Usage Example – Error 4 Missing Completion

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

Rank(s)	Type	Message	From	References
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_Isend** (1st occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:22	References of a representative process: reference 1 rank 0: **MPI_Isend** (1st occurrence) called from: #0 main@mpi_overlap_deadlock_errors.c:22

MUST Usage Example – Summary

- Example “mpi_overlap_deadlock_errors.c” :

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

//1) Create some datatypes
MPI_Type_contiguous ( 5, MPI_INT, &r        pe );
          mmit ( &rowType );
          ctor ( 5 /*count*/, 1 /*     cklength*/, 5 /*stride*/, MPI_INT,
                 &colType );
          mmit ( &colTy    e );

//2)        ISend and MPI_Recv to perform a ring commun      ation
MPI_Isend (  &arr[0], 1, colType, (rank+1)%size, 456, comm, &request );
MPI_Recv (  &arr[10], 1, rowType, (rank-1+size) % size, 456, comm,
            &status );

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

MPI_Finalize ();
```

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

Potential deadlock: MPI_Send may block (depends on MPI implementation and buffer size)

User forgets to call an MPI_Wait for the MPI request

Send/recv count are 0, is this intended?

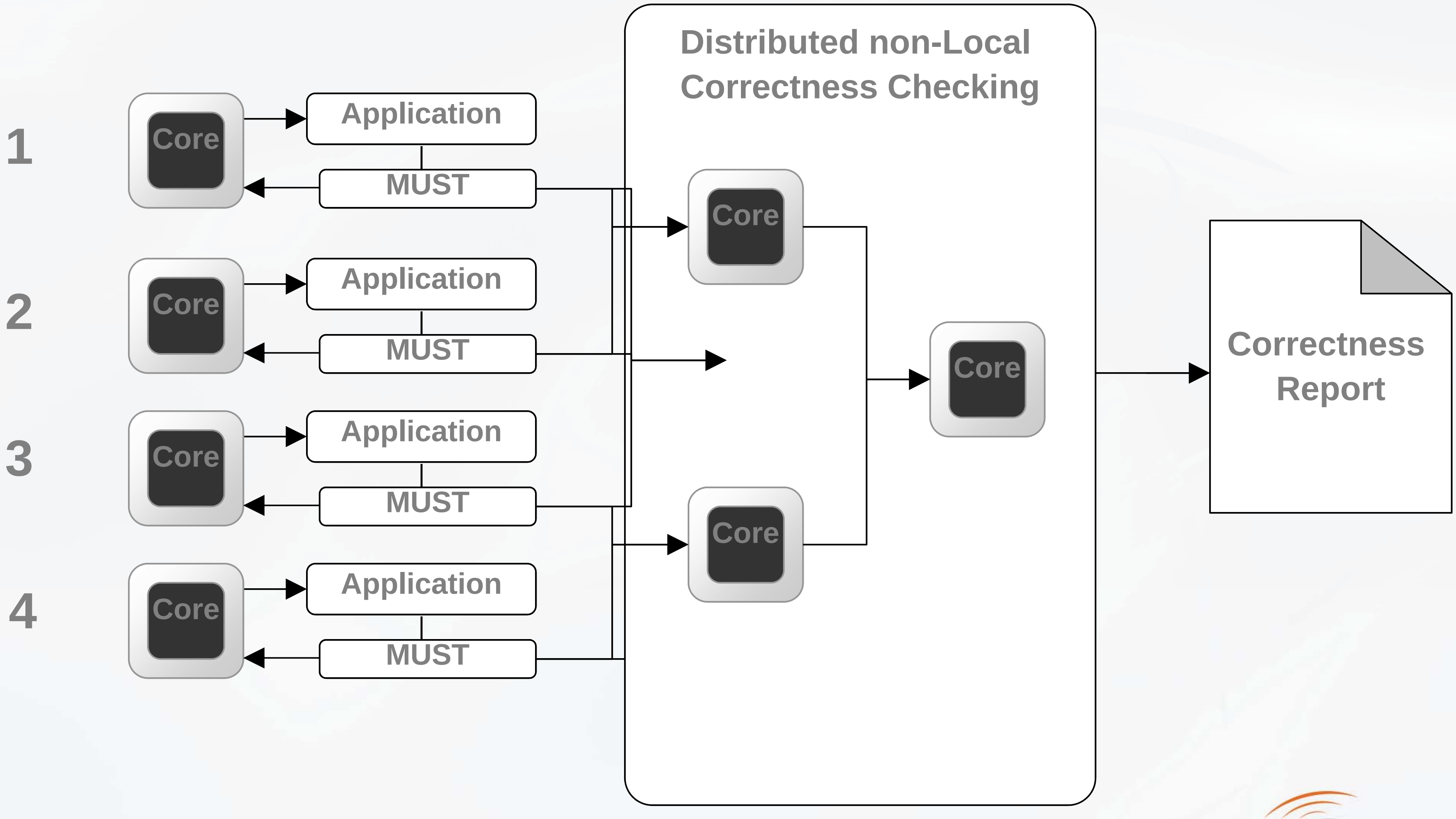
User forgot to free MPI Datatypes before calling MPI_Finalize

Scalability – Operation Modes

- MUST causes overhead at runtime
- MUST expects application crash at any time
  - MUST's communication must tolerate crashes
- Basic operation modes (centralized):

Centralized, application known to crash	Centralized, application does not crash
mustrun -np X exe + All checks enabled + Requires only one extra process - Very slow => use for small test cases at < 32 processes	*mustrun -np X --must:nocrash exe* + All checks enabled + Requires only one extra process - Application must not crash or hang - Use for < 100 processes

Scalability – Distributed Correctness Checking



Scalability – Advanced Operation Modes

Distributed, no crash	Centralized, crash	Distributed, crash
mustrun -np X --must:fanin Z exe • Uses tree network: Layer 0: X ranks Layer 1: ceil(X/Z) ranks … Layer k: 1 rank ~ 10.000 process scale • Use “--must:nodl” to disable deadlock detection towards reduced overhead	*mustrun -np X --must:nodesize Y exe* • Three layer network: Layer 0: X Layer 1: ceil(X/(Y-1)) Layer 2: 1 + < 100 processes + All checks - Currently not on all systems	*mustrun -np X --must:nodesize Y --must:fanin Z exe* • Uses tree network: Layer 0: X Layer 1: A=ceil(X/(Y-1)) Layer 2: B=ceil(A/Z) … Layer k: 1 + ~ 10.000 process scale

Scalability – "--must:info"

- Use "--must:info" to learn about a configuration:

```
% mustrun --must:info \
          --must:fanin 16 \
          --must:nodesize 12 \
          -np 1024
[MUST] MUST configuration ... distributed checks
       with application crash handling
[MUST] Required total number of processes ... 1125
[MUST] Number of application processes ... 1024
[MUST] Number of tool processes ... 101
[MUST] Total number of required nodes ... 94
[MUST] Tool layers sizes ... 1024:94:6:1
```

Configuration type

Tree layout

Number of compute nodes

Total number processes used