PPCES 2017

Runtime Correctness Checking Tools

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

IT Center | RWTH AACHEN UNIVERSITY

MPI runtime correctness checking

How many errors 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;
}
```

No MPI_Init before first MPI-call
Fortran type in C
Recv-recv deadlock
Rank0: src=size (out of range)
Type not committed before use
Type not freed before end of main
Send 4 int, recv 2 int: truncation
No MPI_Finalize before end of main

MPI usage errors

- MPI programming is error prone
- Bugs may manifest as:
  - Crashes
  - Hangs
  - Wrong results
  - Not at all! (Sleeping bugs)
- Tools help to detect these issues

MPI usage errors (2)

- Complications in MPI usage:
  - Non-blocking communication
  - Persistent communication
  - Complex collectives (e.g. Alltoallw)
  - Derived datatypes
  - Non-contiguous buffers

- Error Classes include:
  - Incorrect arguments
  - Resource errors
  - Buffer usage
  - Type matching
  - Deadlocks

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

Who?

What?

Where?

Details

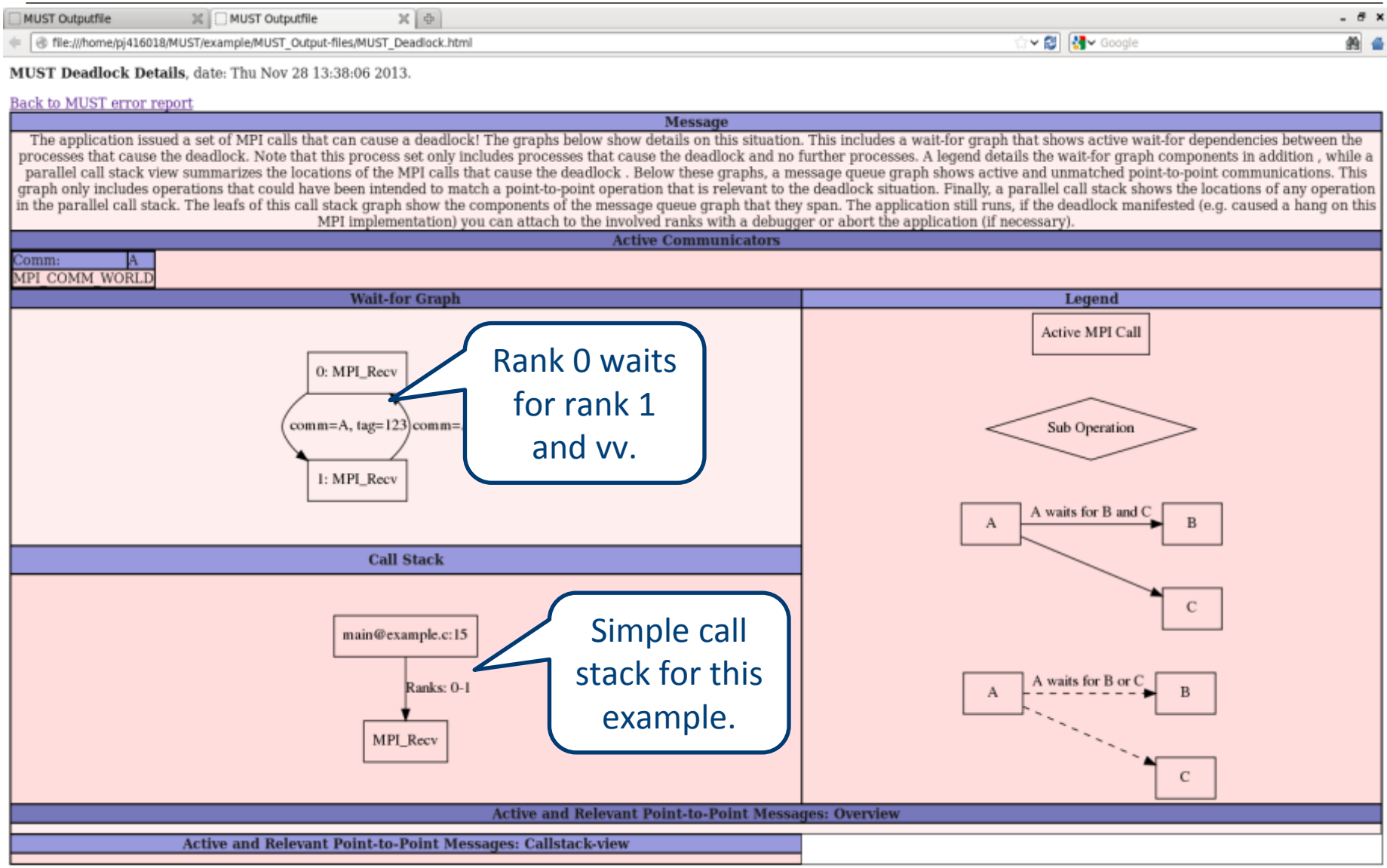
MUST Outputfi

MU Output, starting date: Thu Nov 28 13:38:01 2

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.

Graphical representation of deadlocks



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

MUST Outputfile

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

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

Size of sent message larger than receive buffer

Rank(s)	Type	Messa	om	References
0	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 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 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
0-1	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})	Representative location: **MPI_Send** (1st occurrence) called from: #0 main@example-fix1.c:18	References of a representative process: reference 1 rank 1: **MPI_Type_contiguous** (1st occurrence) called from: #0 main@example-fix1.c:13
0	Error	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: 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).	Representative location: **MPI_Send** (1st occurrence) called from: #0 main@example-fix1.c:18	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
1	Error	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: 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_1_0.html).	Representative location: **MPI_Send** (1st occurrence) called from: #0 main@example-fix1.c:18	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: #0 main@example-fix1.c:13
				References of a representative process:

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_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

Use of uncommited datatype: **`type`**

0-1	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})	Representative location: **MPI_Send** (1st occurrence) called from: #0 main@example-fix1.c:18	References of a representative process: reference 1 rank 1: **MPI_Type_contiguous** (1st occurrence) called from: #0 main@example-fix1.c:13

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

1	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 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, commited at reference 4, 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-fix3.c:19	References of a representative process: reference 1 rank 1: **MPI_Send** (1st occurrence) called from: #0 main@example-fix3.c:19 reference 2 rank 0: **MPI_Irecv** (1st occurrence) called from: #0 main@example-fix3.c:17 reference 3 rank 1: **MPI_Type_contiguous** (1st occurrence) called from: #0 main@example-fix3.c:13 reference 4 rank 1: **MPI_Type_commit** (1st occurrence) called from: #0 main@example-fix3.c:14

Fix3: commit datatype before usage
Fix4: use same C-type for both Integers

```
#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

Commit the datatype before usage

MUST detects data races in asynchronous communication

MUST Outputfile

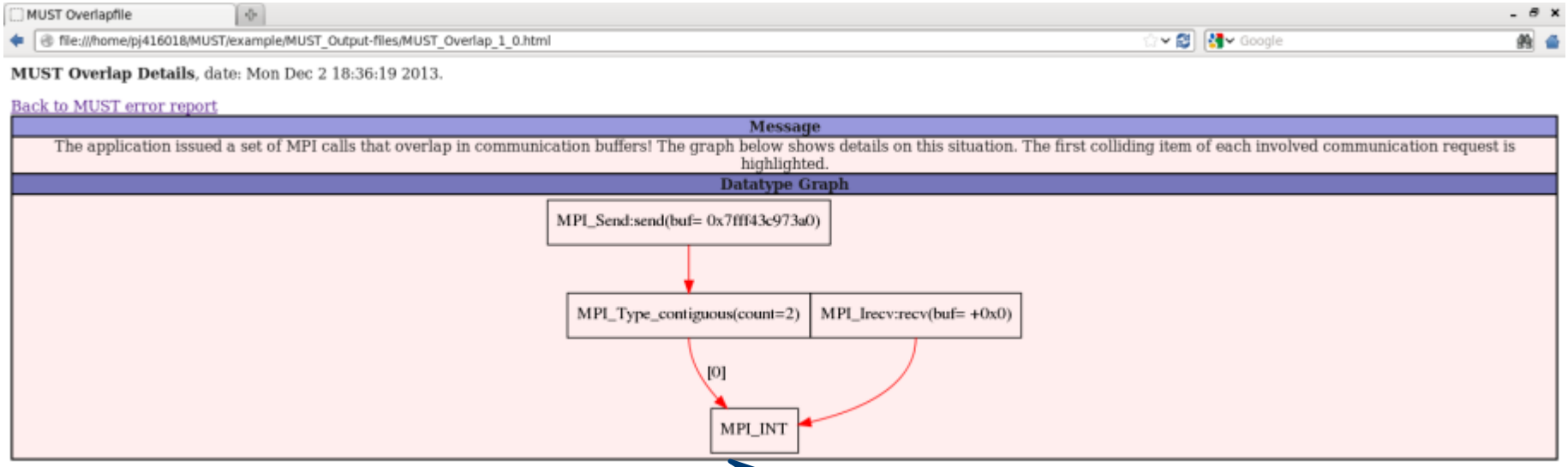
file:///home/pj416018/MUST/example/MUST_Output.html

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

Data race between send and ascynchronous receive operation

Rank(s)	Type	Message		
1	Error	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: 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_1_0.html).	Representative location: **MPI_Send** (1st occurrence) called from: #0 main@example-fix4.c:19	References of a representative process: reference 1 rank 1: **MPI_Irecv** (1st occurrence) called from: #0 main@example-fix4.c:17 reference 2 rank 1: **MPI_Type_contiguous** (1st occurrence) called from: #0 main@example-fix4.c:13 reference 3 rank 1: **MPI_Type_commit** (1st occurrence) called from: #0 main@example-fix4.c:14
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, 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
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-fix4.c:17	References of a representative process: reference 1 rank 1: **MPI_Irecv** (1st occurrence) called from: #0 main@example-fix4.c:17
0	Error	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: 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_0_0.html).	Representative location: **MPI_Send** (1st occurrence) called from: #0 main@example-fix4.c:19	References of a representative process: reference 1 rank 0: **MPI_Irecv** (1st occurrence) called from: #0 main@example-fix4.c:17 reference 2 rank 0: **MPI_Type_contiguous** (1st occurrence) called from: #0 main@example-fix4.c:13 reference 3 rank 0: **MPI_Type_commit** (1st occurrence) called from: #0 main@example-fix4.c:14

Graphical representation of the race condition



Graphical representation of the data race location

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, commited at refere[illegible] 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 sho[illegible] free all MPI resources before calling MPI_Finalize. Listing information for these requests: -Request 1: Request activated at reference 1	Representative location: [illegible] (1st occurrence) called [illegible]	References of a representative process: reference 1 rank 0: **MPI_Irecv** (1st occurrence) called from: [illegible]

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

Leak of user defined datatype object

Unfinished non-blocking receive is resource leak and missing synchronization

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

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

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

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

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

"-1" instead of "MPI_ANY_SOURCE"

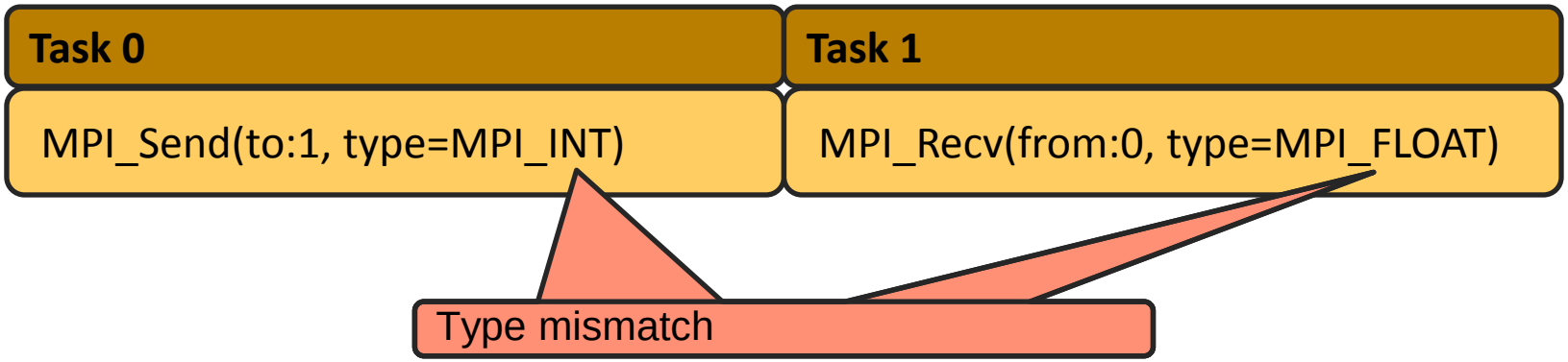
- Model checking:
  - Requires a model of your applications
  - State explosion possible
  - E.g.: MPI-Spin

```
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
  - Limited to the execution branches actually reached during execution
  - 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

Approaches to Remove Bugs (Selection)

Repartitioning?
Representative input?
Node Memory?
Grid?
Reproducibility?

Downscaling

Our contribution:
MUST

Runtime Checking

Debuggers

Bug@ Scale

Umpire
ISP/ DAMPI

Static Code Analysis

TASS
pCFG's
Barrier Analysis

Model Checking

MUST – Basic Usage

- Load MUST module:

```
% module load UNITE must
```

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

```
% $MPICC source.c -o exe
% $MPIRUN -np 4 ./exe
```

- Replace:

```
% $MPICC -g source.c -o exe
% mustrun --must:mpiexec $MPIRUN -np 4 ./exe
```

or:

```
% mustrun -np 4 ./exe
```

- After run: inspect "MUST_Output.html"

MUST - Advanced usage

- MUST scales with application to 10000's processes
  - Use --must:fanin
- Read help:
  - % mustrun --help
- Read documentation:
  - http://www.itc.rwth-aachen.de/must/
- MUST is available for download (BSD license):
  - http://www.itc.rwth-aachen.de/must/
- Feedback via email:
  - must-feedback@lists.rwth-aachen.de

OpenMP runtime correctness checking

OpenMP Correctness checking: ThreadSanitizer – Overview

- Correctness checking for threaded applications
- Integrated in clang and gcc compiler
- Low runtime overhead: 2x – 15x
- Used to find data races in browsers like Chrome and Firefox

OpenMP Correctness checking: ThreadSanitizer – Usage

```
module load archer gcc/6
```

- Compile the program with clang compiler:

C

```
clang -fsanitize=thread -fopenmp -g myprog.c -o myprog
```

C++

```
clang++ -fsanitize=thread -fopenmp -g myprog.cpp
            -o myprog
```

Fortran

```
gfortran -fsanitize=thread -fopenmp -g myprog.f -c
clang -fsanitize=thread -fopenmp -lgfortran myprog.o
            -o myprog
```

- Execute:

```
OMP_NUM_THREADS=4 ./myprog
```

- Understand and correct the detected threading errors
- Edit the source code
- Repeat until no errors reported

OpenMP Correctness checking: ThreadSanitizer – Result Summary

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv) {
4      int a = 0;
5      #pragma omp parallel
6      {
7          if (a < 100) {
8              #pragma omp critical
9              a++;
10         }
11     }
12 }
```

WARNING: ThreadSanitizer: data race

Read of size 4 at 0x7fffffffdcdc by thread T2:

#0 .omp_outlined. race.c:7 (race+0x0000004a6dce)

#1 __kmp_invoke_microtask <null> (libomp_tsan.so)

Previous write of size 4 at 0x7fffffffdcdc by main thread:

#0 .omp_outlined. race.c:9 (race+0x0000004a6e2c)

#1 __kmp_invoke_microtask <null> (libomp_tsan.so)

Thank you for your attention