MUST

MPI Runtime Error Detection Tool

Version 1.11.0

June 5, 2025

Contents

1 Introduction

MUST detects usage errors of the Message Passing Interface (MPI) and reports them to the user. As MPI calls are complex and usage errors common, this functionality is extremely helpful for application developers that want to develop correct MPI applications. This includes errors that already manifest as segmentation faults or incorrect results, as well as many errors that are not visible to the application developer or do not manifest on a particular system or MPI implementation.

To detect errors, MUST intercepts the MPI calls issued by the target application and evaluates their arguments. The two main usage scenarios for MUST arise during application development and during porting. When a developer adds new MPI communication calls, MUST can detect newly introduced errors, especially also some that may not manifest in an application crash. Further, before porting an application to a new system, MUST can detect violations to the MPI standard that might manifest on the target system. MUST reports errors in a log file that can be investigated once the execution of the target executable finishes (irrespective of whether the application crashed or not).

2 Installation

The MUST software consists of three individual packages:

- P^{n}MPI
- GTI
- MUST

The P^{n}MPI package provides base infrastructure for the MUST software and intercepts MPI calls of the target application. GTI provides tool infrastructure, while the MUST package contains the actual correctness checks.

Starting with version 1.6, all three packages are contained in a single archive and configured and built at once.

Each MUST installation is built with a specific compiler and MPI library. It should only be used for applications built with the same compiler and MPI library. This is necessary as the behavior of MUST may differ depending on the MPI library. Compilers may be mixed if they are binary compatible.

Building MUST requires CMake for configuration. It is freely available at `http://www.cmake.org/`. You can execute *which cmake* to determine whether a CMake installation is available. If not, contact your system administrator or install a local version, which requires no root privileges. We suggest to use CMake version 3.9 or later (use *cmake --version*) for most functionality. Individual optional features may require even never CMake versions, which is specified in Section 2.1. From CMake version 3.20 on, full functionality is supported.

Further, to augment the MUST output with call stack information, which is very helpful for pinpointing errors, it is possible to utilize Backward or addr2line. In that case, MUST uses Backtrace-cpp or addr2line to read and print stack traces for errors. Section 7 presents the necessary steps for such an installation. Additionally, type-matching and data race analyses can be enabled by the integration of TypeART (see Section 8) and ThreadSanitizer (see Section 3.2), respectively.

MUST supports parallel build. Therefore you may want to append *–j<number of cores>* to the make calls.

2.1 Prerequisites to build and use MUST

- CMake (required 3.13.4 or newer, optional 3.20 or newer for TypeART, see *cmake --version*)
- Python (required 3, see *python -V*)
- libxml2 with headers (libxml2-dev / libxml2-devel, required)
- Graphviz (optional, to generate graphs)
- a browser (optional, to view html output)
- LLVM lit (optional 12 or newer to run tests, see *tests/README.md*)
- LLVM FileCheck (optional to run tests, see *tests/README.md*)
- LLVM-based compiler (optional)
  - version 6 or newer for ThreadSanitizer race analysis, alternatively GCC version 8 or newer
  - version 14, 18, or 19 for TypeART, see Section 8
  - version 15 or newer for RMA data race analysis
- MPI library, used by the application (required)

2.2 Configuring with CMake

All parts of MUST use CMake for configuration. CMake works best with 'out of source' builds, this is what we recommend in the installation steps below. Common CMake options include `-DCMAKE_INSTALL_PREFIX` to set the path to install into if you do not have root or to provide MUST as module environment package. CMake options can be configured with a GUI on many systems by using `ccmake` instead of cmake with all the -D flags listed below.

- When the ccmake gui appears:
- press c to generate options, press e to move on from any messages displayed by cmake.

- edit any options displayed,
- press c to see if there are any new options resulting from the previous round of choices
- repeat until you are happy with the options
- press g to generate the build
- move on to the make step as usual.

2.3 Building MUST

MUST can be built as follows (assuming GNU compilers):

```
tar xzf MUST-v1.11.0.tar.gz
cd MUST-v1.11.0
mkdir BUILD
cd BUILD
CC=$(which gcc) CXX=$(which g++) FC=$(which gfortran) \
cmake ../ \
     -DCMAKE_INSTALL_PREFIX=<MUST-INSTALLATION-DIR> \
     -DCMAKE_BUILD_TYPE=Release
make -j8 install
export PATH=<MUST-INSTALLATION-DIR>/bin:$PATH
```

In many cases, it is essential, to use the plain compilers for *CC*&Co, i.e., not the MPI compiler wrappers. The CMake call will determine your MPI installation in order to configure MUST correctly. If this should fail – or multiple MPIs are available – you can tip the configuration by specifying *–DMPI_C_COMPLIER=<FILE–PATH–TO–MPICC>* as well as *–DMPI_CXX_COMPLIER=<FILE–PATH–TO–MPICXX>* and *–DMPI_Fortran_COMPLIER=<FILE–PATH–TO–MPIF90>* as additional arguments to the *cmake* command. More advanced users can fine-tune the detection by specifying additional variables, consult the comments in *cmakemodules/FindMPI.cmake*. On clusters with special MPI environments, it helps to verify that *MPIEXEC* is set to the right mpiexec command (like srun).

Usually, no extra arguments are needed to configure MUST. You can specify -*DENABLE_TESTS=On* to activate the test suite that is included in MUST. Tests should only be started after installing MUST and can be run from within the build directory with:

```
make check
```

or

```
make build-test
lit tests
```

Make sure to invoke the build target `build-test` before running `lit` directly to build the tests. It is invoked implicitly when using the `check` target.

Some tests will fail even for a correct installation since they document future extensions. You can get a detailed test report for a single test with:

```
make build-test
lit tests/<path/to/test/testname>
```

To run all *BasicChecks* with up to 4 tests in parallel:

```
make build-test
lit -j4 tests/BasicChecks
```

More information on the test suite can be found in *<MUST–SOURCE–DIR>/tests/README.md*.

2.4 Install Prebuilt Configurations

To speed up the tool preparation time, we provide some prebuilt configurations for typical tool usage. These can be installed during building of MUST:

```
make -j8 install install-prebuilds
```

We strongly suggest this step for cluster installations. If prebuilts are not available, MUST will prepare an appropriate configuration during the execution of mustrun.

2.5 Environmentals

To work with MUST, it is sufficient to add *<MUST–INSTALLATION–DIR>/bin* to your *PATH* variable.

3 Usage

The following two steps allow you to use MUST:

- Replace the *mpiexec* command with *mustrun* to execute your application;
- Inspect the result file of the run.

3.1 Execution

The actual execution of an application with MUST is done by replacing the *mpiexec* command with *mustrun*. It performs a code generation step to adapt the MUST tool to your application and will run your application with MUST afterward.

The plain *mustrun* command that we use here is intended for small scale short-running applications and can exhibit very high runtime overhead. Section 5 presents further configurations of MUST that we tested with up to $16,384$ processes. The plain *mustrun* command uses all of MUST's correctness checks and a communication system where one MPI process is used to drive some of these checks. So when submitting a batch job, you should make sure to allocate resources for one additional task. Further, when calling *mustrun* you need to have access to the compilers and MPI utilities that were used to build MUST itself.

A regular *mpiexec* command like:

```
mpiexec -np 4 application.exe
```

Is replaced with:

```
mustrun -np 4 application.exe
```

It will execute your application with four tasks but requires one additional task, i.e. it will actually invoke *mpiexec* with *-np 5*.

For an example where the *mpiexec* command and the switch used to specify the number of processes is named differently:

```
srun -n 4 application.exe
```

You could use the following*mustrun* command:

```
mustrun --must:mpiexec srun --must:np -n  -n 4 application.exe
```

If your machine provides no compilers in batch jobs, you can prepare a run as follows:

```
mustrun --must:mode prepare -np 4 application.exe
```

In your batch job you would then just execute:

```
mustrun --must:mode run -np 4 application.exe
```

The *mustrun* tool provides further switches to modify its behavior, call *mustrun --must:help* for a summary. If you encounter errors during execution, please submit error reports where you use *--must:verbose* as an argument to *mustrun*.

It is not possible to pass arguments to mustrun that contain whitespace characters like spaces or tabs using command line switches. Use environment variables to pass such arguments to *mustrun* (see Section 3.1.1).

3.1.1 Passing arguments by environment variables

You can also use environment variables to pass switches to *mustrun*.

Every command line switch has a corresponding environment variable. These variables are named like the switches according to these rules:

- the two leading hyphens are removed
- all characters are converted to upper case
- colons and hyphens are replaced by underscores

For example the switch `--must:output-dir` can also be set with the environment variable `MUST_OUTPUT_DIR`.

Command line switches take precedence over environment variables. If both are used for the same switch, then the argument passed by the environment variable gets overridden by the command line argument.

Note the placement of quotes in the following example with whitespace in an MUST argument:

```
env "MUST_MPIEXEC=mpiexec --verbose" mustrun -np 4 application.exe
```

3.2 Execution of threaded applications

For support of threaded applications, MUST provides a thread-safe mode:

```
mustrun --must:hybrid -np 4 application.exe
```

Be aware that MUST lifts the required MPI threading level in this case to `MPI_THREAD_MULTIPLE`, while MUST limits the provided threading level without this flag to `MPI_THREAD_SINGLE`.

3.2.1 OpenMP Applications

MUST supports analyzing hybrid applications using MPI and OpenMP, e.g., checking thread level usage. For that, MUST makes use of the OMPT interface, which was introduced in OpenMP 5.0. Thus, in order to use the OpenMP mode, a recent compiler with OMPT support is required (e.g., LLVM/Intel/Cray compiler). The OpenMP mode implicitly enables the hybrid thread-safe mode, it suffices to run the application with

```
mustrun --must:openmp -np 4 application.exe
```

while making sure that the application is built with OpenMP support (`-fopenmp`) and executes with multiple threads (`OMP_NUM_THREADS` > 1).

3.2.2 Data Race Detection

MUST additionally features checks to detect data races involving MPI communication in hybrid programs with the help of ThreadSanitizer[1], which can be enabled by

```
mustrun --must:tsan -np 4 application.exe
```

This requires MUST to be configured with `-DENABLE_TSAN=On` (enabled by default) and the application under test to be compiled with `-fsanitize=thread` (supported by GNU- and LLVM-based compilers). Currently, ThreadSanitizer integration for data race detection is fully supported with the clang compiler from version 6 onwards and gcc version 8 onwards. The application also has to be linked with *<MUST–INSTALLATION–DIR>/lib/libonReportLoader.a* and built with debug info for

[1] `https://clang.llvm.org/docs/ThreadSanitizer.html`

MUST to generate meaningful reports for detected data races. Some linkers may not add unused symbols of the library to the application, in which case it helps to tell the linker to include all symbols from the library

```
$CC app.c -g \
    -Wl,--whole-archive install/must/lib/libonReportLoader.a \
    -Wl,--no-whole-archive \
    --fsanitize=thread
```

There is also a compiler wrapper installed named `must-tsan-cc`, `must-tsan-cxx`, and `must-tsan-fort` for the different languages to link the *libonReportLoader.a* without requiring manual modification of compiler flags.

Using the *libonReportLoader.a* when linking is experimental. If the application is not linked with *libonReportLoader.a*, data races will not be reported in MUST's HTML report but only on standard output.

If MUST is configured with `-DENABLE_STACKTRACE=On` (enabled by default, see Section 7), information from the stacktraces will be used for reports on data races. To check whether MUST is configured properly with ThreadSanitizer support, run

```
make build-test
lit tests/MpiTSan
```

or

```
make check-MpiTSan
```

For the tests however MUST has to be built with either gcc or clang in one of the aforementioned versions.

3.3 Results

MUST stores its results in an HTML file named *MUST_Output.html*. It contains information on all detected issues, including information on where the error occurred. By default the *MUST_Output.html* file is placed in the execution directory. The output directory can be specified via *--must:output-dir <path>*.

Moreover, MUST's output can be split across multiple files by the user. To this end, the application under test may call `MUST_ChangeMessageFile(filename)`, which will result in MUST writing output to `filename`. This requires including the `MUST_Annotations.h` header and adding *<MUST–INSTALLATION–DIR>/include* to the include path for compilation. Note that the filename can be altered multiple times during execution. The filename is interpreted relative to MUST's main output directory.

3.3.1 Filtering Messages

MUST also allows its output to be filtered according to user-defined rules. Filter rules have the following general format: `messageType:MUST_ERROR_TYPE:source`, where `MUST_ERROR_TYPE` determines which type of errors or warnings to filter out (e.g., `MUST_ERROR_TYPEMATCH_MISMATCH`) and `source` is one of the following:

- specific file as `source`: `src:`*`filename.c`*
- specific function as `source`: `func:`*`function_name`*
- every `source`: `*`

For example, the rule `messageType:MUST_WARNING_COMM_NULL:src:main.c` would filter out all of MUST warning regarding `MPI_COMM_NULL` or `NULL` as a communicator handle resulting from the source file `main.c`. The first two type of rules require MUST to use stacktraces, which can be enabled with the *--must:stacktrace* option (see Section 7).

Multiple filter rules may be specified in a file line by line and given to MUST via *--must:filter-file <file>*. Alternatively, the environment variable `MUST_FILTER_FILE` can be set to the file. The error type printed with each message in the report can be used to suppress such error output. For a list of more supported `MUST_ERROR_TYPES` see *<MUST–SOURCE–DIR>/modules/Common/MustEnums.h*.

3.3.2 Output Modes

Instead of printing out HTML, MUST may also log its output to stdout or generate a JSON-file. The output mode can be specified with *--must:output* < *html*|*stdout*|*json* >. The json-mode aims to ease automatic parsing of MUST's output.

An example output for MUST's json-mode:

```
{
'messages':
[
    {
      'type': 'Error',
      'error_id': 'MUST_ERROR_LEAK_DATATYPE',
      'text': 'There are 1 datatypes that are not freed [...] at reference 1',
      'from': {
        'call': 'MPI_Type_contiguous',
        'ranks': [ 0 ],
        'ranks_strided': false
    },
    'references': [{
      'no': 1,
      'call': 'MPI_Type_contiguous',
      'rank': 0,
      'threadid': 0,
      'stacktrace': [

        ]
      }]
    }
]
}
```

3.3.3 Instant Logging

By default, MUST reports the same error occuring on different ranks only once for the sake of clarity. Depending on the MPI library, however, this mode may prevent

MUST from reporting errors that cause the application to crash. MUST can be forced to immediately report errors, *--must:instant-logging <info—warning—error—fatal>* can be used. Note that through this option, the same error occuring on different ranks will only be reported for one of the ranks.

4 Example

As an example consider the following application that contains three MPI usage errors:

```
#include <stdio.h>
#include <mpi.h>

int main (int argc, char** argv)
{
    int rank,
        size,
        sBuf[2] = {1,2},
        rBuf[2];
    MPI_Status status;
    MPI_Datatype newType;

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

    //Enough tasks ?
   if (size < 2)
   {
      printf ("This test needs at least 2 processes!\n");
      MPI_Finalize();
      return 1;
   }

   //Say hello
   printf ("Hello, I am rank %d of %d processes.\n", rank, size);

   //1) Create a datatype
   MPI_Type_contiguous (2, MPI_INT, &newType);
   MPI_Type_commit (&newType);

   //2) Use MPI_Sendrecv to perform a ring communication
   MPI_Sendrecv (
           sBuf, 1, newType, (rank+1)%size, 123,
           rBuf, sizeof(int)*2, MPI_BYTE, (rank-1+size) % size, 123,
           MPI_COMM_WORLD, &status);

   //3) Use MPI_Send and MPI_Recv to perform a ring communication
   MPI_Send (sBuf, 1, newType, (rank+1)%size, 456, MPI_COMM_WORLD);
   MPI_Recv (rBuf, sizeof(int)*2, MPI_BYTE, (rank-1+size) % size, 456,
       MPI_COMM_WORLD, &status);

   //Say bye bye
   printf ("Signing off, rank %d.\n", rank);

    MPI_Finalize ();

    return 0;
}
/*EOF*/
```

4.1 Execution with MUST

A user could set up the environment for MUST, build the application, and run it with the following commands:

```
#Set up environment
export PATH=<MUST-INSTALLATION-DIR>/bin:$PATH

#Compile and link, we rely on the ld-preload mechanism
mpicc example.c -o example.exe -g

#Run with four processes. We will need resources for five tasks!
mustrun -np 4 example.exe
```

4.2 Output File

The output of the run with MUST will be stored in a file named *MUST_Output.html*. For this application MUST will detect three different errors that are:

- A type mismatch (Figure 1)
- A send-send deadlock (Figure 3)
- A leaked datatype (Figure 5)

Figure 1 shows the first error that MUST detects. The error results from the usage of non-matching datatypes, which are an `MPI_INT` and an `MPI_BYTE` of the same size as the integer value. This is not allowed according to the MPI standard. A correct application would use `MPI_INT` for both the send and receive call.

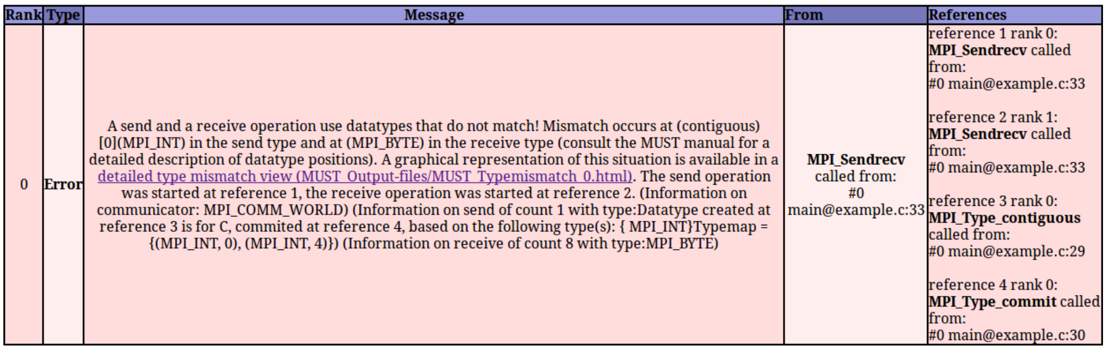
If MUST is configured with stack trace support (Section 7), the right column will list call stacks for all the involved MPI calls, as in Figure 5. Here the error is detected in the `MPI_Sendrecv` call in line 33.

The example shows the specification of the location in the datatype that causes the mismatch. The location `(CONTIGUOUS)[0](MPI_INT)` means that the used datatype is of contiguous kind. The mismatch is within the first element of the contiguous type, which is defined to be a base type, namely `MPI_INT`.

As another example `(VECTOR)[1][2](MPI_CHAR)` would address the third entry of the second block of a vector with base-type `MPI_CHAR`.

Figure 2 displays a graphical representation of the type mismatch. The image shows type trees of the involved data types. For a correct type match, both trees should share all their leaves. For a clearer view, matching leaves are hidden. The path to the first clash is highlighted in red. For derived types, the node labels display the count/blocklength value, used in the declaration of the type, while the edge label (corresponding to the path expression) gives the index of the block/blockitem, that leads to the first clash.

For communication buffers that access the same memory address concurrently ("buffer overlap"), similar descriptions and graphs are used. In this case, all nodes that point



Rank	Type	Message	From	References
0	Error	A send and a receive operation use datatypes that do not match! Mismatch occurs at (contiguous)[0](MPI_INT) in the send type and at (MPI_BYTE) 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 C, commited at reference 4, based on the following type(s): { MPI_INT}Typemap = {(MPI_INT, 0), (MPI_INT, 4)}) (Information on receive of count 8 with type:MPI_BYTE)	**MPI_Sendrecv** called from: #0 main@example.c:33	reference 1 rank 0: **MPI_Sendrecv** called from: #0 main@example.c:33 reference 2 rank 1: **MPI_Sendrecv** called from: #0 main@example.c:33 reference 3 rank 0: **MPI_Type_contiguous** called from: #0 main@example.c:29 reference 4 rank 0: **MPI_Type_commit** called from: #0 main@example.c:30

Figure 1: Type mismatch error report from MUST.

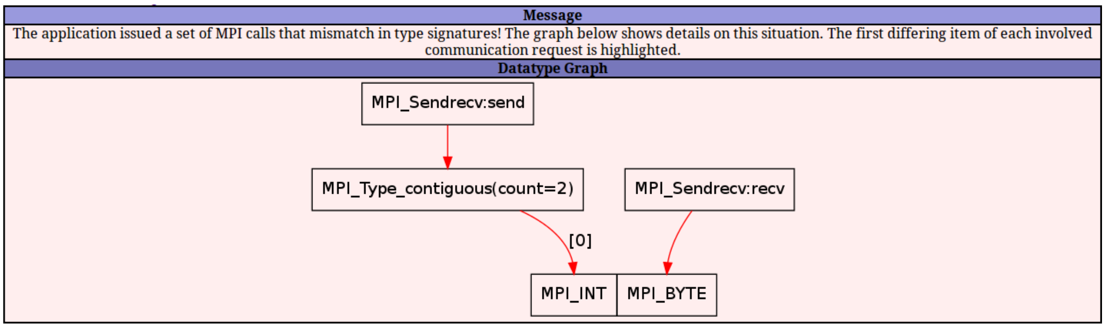


Figure 2: Detail page for the type mismatch in Figure 5.

to distinct memory addresses are hidden, as the focus lies on the representation of the memory overlap.

The second error results from the application calling send calls that can lead to deadlock (Figure 3). Each task issues one call to `MPI_Send` while no matching receive is available. This can cause deadlock. However, as such calls would be buffered for most MPI implementations, this is a deadlock that only manifests for some message sizes or MPI implementations.

If MUST detects a deadlock, it provides visualization for its core, i.e., the set of MPI calls of which at least one call has to be modified or replaced. It stores a wait-for graph representation of this core in a file named *MUST_Deadlock.dot*. If available, MUST automatically translates this file into an image and provides a deadlock view (Figure 4), which shows the task dependencies and a parallel call stack. This graph file uses the DOT language of the *Graphviz* package. If a graphviz installation was available when MUST was installed, it automatically visualizes the graph. Otherwise, you can visualize it by issuing *dot -Tps MUST_Deadlock.dot -o deadlock.ps* after installing this tool. You can open the file *deadlock.ps* with the postscript viewer of your choice (DOT also supports additional output formats). If MUST was configured with stack trace support (Section 7), it will also print a parallel call stack in a file called *MUST_DeadlockCallStack.dot*, which Figure 4 shows at the bottom. This stack includes any MPI call that was referred to in the wait-for graph. Especially if processes use non-blocking communications, this call stack

Rank	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 1-4 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).		reference 1 rank 0: **MPI_Send** called from: #0 main@example.c:39 reference 2 rank 1: **MPI_Send** called from: #0 main@example.c:39 reference 3 rank 2: **MPI_Send** called from: #0 main@example.c:39 reference 4 rank 3: **MPI_Send** called from: #0 main@example.c:39

Figure 3: Send-send deadlock report from MUST, basic report.

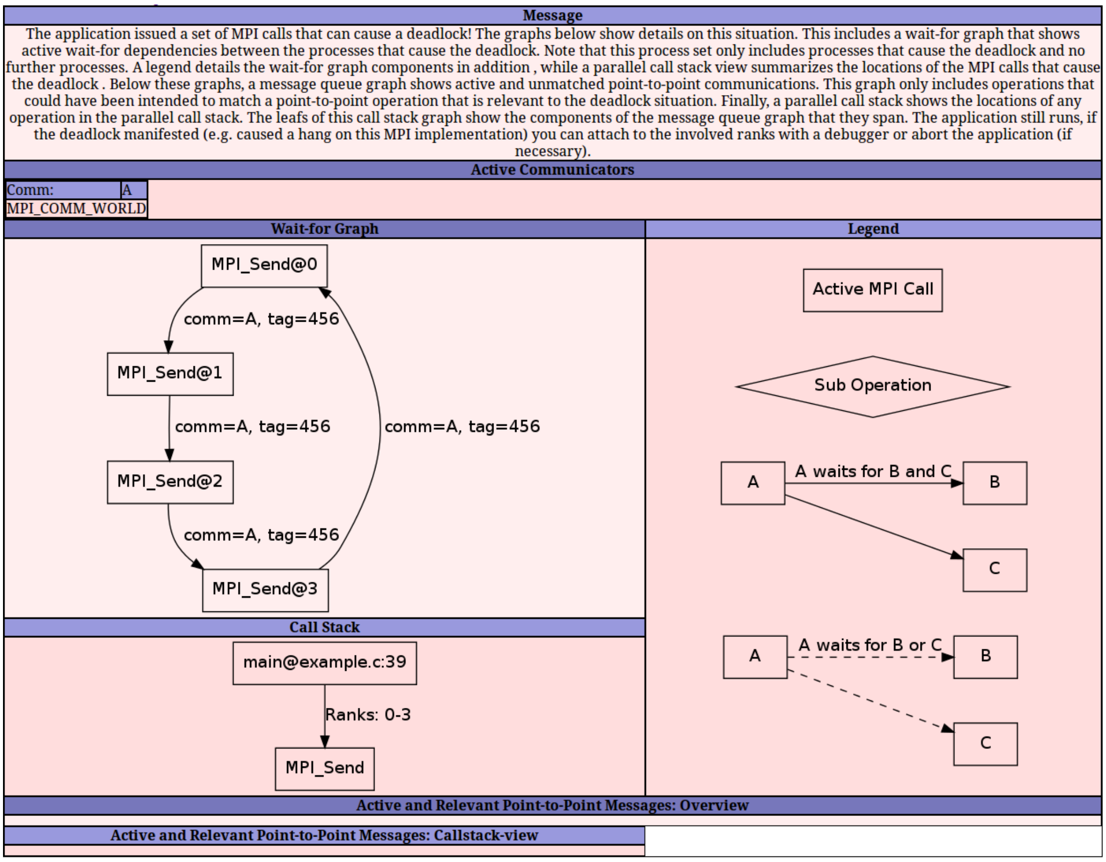


Figure 4: Deadlock view for the send-send deadlock.

	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}Typemap = {(MPI_INT, 0), (MPI_INT, 4)}		reference 1 rank 0: **MPI_Type_contiguous** called from: #0 main@example.c:29 reference 2 rank 0: **MPI_Type_commit** called from: #0 main@example.c:30

Figure 5: Resource leak report from MUST.

may include multiple MPI calls for each process.

Further graphs in the deadlock view show information about the message matching state to highlight any call that might have been intended to match a blocked point-to-point call. Since no outstanding point-to-point message exists in the deadlock situation of Figure 3, these graphs are empty.

Finally, MUST detects that the application leaks MPI resources when calling `MPI_Finalize`. In particular, this is a datatype created with an `MPI_contiguous` call. Applications should free all such resources before invoking `MPI_Finalize`, as harmful leaks are easier to detect in such cases.

5 MUST's Operation Modes

MUST's analysis of all MPI calls causes runtime overhead. As a result, it is important to adapt its configuration such that its overhead stays acceptable. While its default configuration (*mustrun* without additional switches) is easy to use, more advanced configurations may be required. MUST's overhead primarily results from:

- Correctness checks that require information from multiple processes, and
- A communication mode that allows MUST to detect MPI usage errors even if the application crashes.

MUST can use more than one additional process to run expensive correctness checks, while a shared memory based communication mode allows MUST to tolerate application crashes with limited runtime overhead.

5.1 Mode Overview

MUST provides the following operation modes that adapt its overhead to the target use-case:

1. **(Default) Very slow, Centralized, application may crash:**
   - Command line: *mustrun -np X exe*
   - One extra process for correctness checking
   - All checks enabled
   - Detects errors even if application crashes

- Very slow, for short running tests at < 32 processes

2. **Fast, centralized, application does not crash:**
   - Command line: *mustrun -np X --must:nocrash exe*
   - One extra process for correctness checking
   - All checks enabled
   - Detects errors only if the application does not crash
   - Limited scalability, use for < 100 processes

3. **Fast, centralized, application may crash:**
   - Command line: *mustrun -np X --must:nodesize Y exe*
   - Number of extra processes: $1 + \lceil \frac{X}{Y-1} \rceil$
   - All checks enabled
   - Detects errors even if application crashs
   - Limited scalability, use for < 100 processes
   - Requires shared memory communication (Available on most linux based clusters)

4. **Distributed, application does not crash:**
   - Command line: *mustrun -np X --must:distributed [--must:fanin Z] exe*
   - Network of extra processes:
     - Layer 0: $A = \lceil \frac{X}{Z} \rceil$
     - Layer 1: $B = \lceil \frac{A}{Z} \rceil$
     - ...
     - Layer k: 1
   - If you need to reduce overheads, you can disable MUST's distributed deadlock detection with *--must:nodl*
   - Detects errors only if the application does not crash
   - Tested with $16,384$ processes

5. **Distributed, application may crash:**
   - Command line:
     mustrun -np X --must:distributed --must:nodesize Y [--must:fanin Z] exe
   - Network of extra processes:
     - Layer 0: $A = \lceil \frac{X}{Y-1} \rceil$
     - Layer 1: $B = \lceil \frac{A}{Z} \rceil$
     - Layer 2: $C = \lceil \frac{B}{Z} \rceil$

  - ...
  - Layer k: 1
- If you need to reduce overheads, you can disable MUST's distributed deadlock detection with *--must:nodl*
- Tested with 4,096 processes
- Requires shared memory communication (Available on most linux based clusters)

5.2 Mode Details

For any non-demanding (short and small scale) use case, we suggest operation Mode 1 (*mustrun -np X exe*), since it is always available and easy to use.

For more extensive application runs at moderate scale (< 100 processes) users should either use Mode 2 (*mustrun -np X --must:nocrash exe*) or Mode 3 (*mustrun -np X --must:nodesize Y exe*). While Mode 2 assumes that the application does not crash, Mode 3 uses a shared memory communication (Linux message queues) to tolerate application crashes. Besides the limited availability of this communication mechanism (most Linux-based systems), it requires more than one extra process to operate. The user needs to specify a node size Y that is a divisor of the number of cores available within each compute node. MUST then uses one tool process per $Y - 1$ application processes. It is important that the resource manager distributes MPI ranks in node-core order. That is, it fills each node completely and with successive ranks. The use of the *--must:fillnodes* switch to the *mustrun* command may help if the total number of MPI ranks does not fill all allocated nodes causing the resource manager to not fill nodes completely.

By adding the *--must:info* switch to any *mustrun* command, the user may retrieve additional information on the number of application tasks, tool tasks, and required nodes without running or preparing a MUST run. This provides valuable information to prepare batch job allocations.

Modes 4 (*mustrun -np X --must:distributed [--must:fanin Z] exe*) and 5 (*mustrun -np X --must:distributed --must:nodesize Y [--must:fanin Z] exe*) are intended for application runs at scale (> 100 processes, where we tested MUST with up to 16,384 processes). Both modes use a tree network to run several correctness checks, which increase their demand for extra computing cores. Again Mode 4 assumes that the application does not crash, while Mode 5 uses a shared memory communication to tolerate application crashs. Mode 5 comes with the same restrictions and allocation assumptions as Mode 3. For both modes, the user may specify the *--must:fanin Z* switch which controls the ratio of application to extra tool processes. The default value is 16, higher values may increase MUST's overhead, while lower values may reduce its overhead. Experience with MUST's distributed deadlock detection shows that it scales to an order of 16,384 processes but can double MUST's overhead. If MUST's overhead is too high for your use-case, you can add the switch *--must:nodl* to disable the distributed deadlock detection for Modes 4 and 5.

Rank(s)	Type	Message
1	**Information**	The following MPI functions were used but are not supported by MUST: MPI_Mprobe,MPI_Mrecv

Details:

Message	From	References
The following MPI functions were used but are not supported by MUST: MPI_Mprobe MPI_Mrecv	Representative location: **MPI_Finalize** (1st occurrence) called from: #0 main@Mrecv.c:39	References of a representative process: reference 1 rank 1: **MPI_Mprobe** (1st occurrence) called from: #0 main@Mrecv.c:35 reference 2 rank 1: **MPI_Mrecv** (1st occurrence) called from: #0 main@Mrecv.c:36

Figure 6: MUST report on unsupported MPI functions `MPI_Mprobe` and `MPI_Mrecv`.

6 Included Checks

MUST currently provides correctness checks for the following classes of errors:

- Constants and integer values
- Communicator usage
- Datatype usage
- Group usage
- Operation usage
- Request usage
- Leak checks (MPI resources not freed before calling `MPI_Finalize`)
- Type mis-matches
- Overlapping buffers passed to MPI
- Deadlocks resulting from MPI calls
- Data races involving MPI calls

6.1 Identifying use of unsupported MPI functions

MUST informs about the usage of MPI functions in the application for which no analysis is offered yet. For an example report of unsupported MPI functions, see Figure 6.

6.2 MPI 4.0 Support

Starting with version 1.9, MUST includes basic MPI 4.0 support. Currently, MPI 4.0 support is limited to correctness checks for Partitioned Point-to-Point Communication and the tracking of resources (e.g. communicators) created through MPI 4.0 functions. However, explicitly not yet covered with analyses are

- MPI Sessions
- New function variants `MPI_XXX_c` using `MPI_Count`
- Persistent collectives

In particular this also means, MUST requires the usage of the *World Model* (`MPI_Init[_thread]` and `MPI_COMM_WORLD`) by the application.

7 Stack Trace Information in MUST Reports

MUST relies on external libraries to generate source code information included in MUST reports. The *--must:stacktrace* switch allows selecting the stack trace mechanism when launching mustrun. Since collecting stack traces can be costly and introduce significant runtime overhead, MUST will not collect stack traces by default. Therefore, the default of this setting is *none*. To enable stack traces based on backward-cpp, use *--must:stacktrace backward*. For a light stack tracing alternative that only resolves the first stack frame and relies on `addr2line`, use *--must:stacktrace addr2line*.

Since MUST version 1.8, MUST is configured with backward-cpp support enabled by default. To install MUST *without* backward-cpp support, the CMake variable **-DUSE_BACKWARD=Off** must be explicitly set during the configuration of MUST. Backward-cpp can work with different libraries to unwind the call stack[2] and to read the debug information from the binary[3]. The backward-cpp CMake configuration will automatically detect and select available debugging libraries.

Since MUST version 1.11, MUST is configured with addr2line support enabled if available. To install MUST *without* addr2line support, the CMake variable **-ENABLE_ADDR2LINE_RESOLUTION=Off** must be explicitly set during the configuration of MUST.

8 TypeART Integration

MUST's native type mis-match detection focuses on distributed mis-matches, e.g., a send and a matching receive call using two incompatible data types. In order to additionally check for local type mis-matches, MUST provides optional integration of the type and memory allocation tracker TypeART[4].

8.1 MUST Preparation

By default, an existing and installed version of TypeART will be used for MUST. However, if TypeART is not installed at this system, the TypeART sources will be included as submodule and built from source. TypeART support requires an LLVM-based compiler in version 14, 18, or 19, and can be enabled during the configuration

[2] `https://github.com/bombela/backward-cpp/#libraries-to-unwind-the-stack`
[3] `https://github.com/bombela/backward-cpp/#libraries-to-read-the-debug-info`
[4] `https://github.com/tudasc/TypeART`

with **-DENABLE_TYPEART=On**. MUST will automatically detect and enable TypeART support if applicable during the configuration.

8.2 Application Preparation

As TypeART relies on code instrumentation, its compiler wrapper needs to be used for compilation. For detailed information on requirements on and preparation of the application to test see *<MUST–SOURCE–DIR>/externals/typeart/README.md*. In case TypeART was built from source as a submodule, the needed wrappers can be found under *<MUST–INSTALLATION–DIR>/bin*, i.e, if you followed the suggetion in Section 2.5 the wrappers will be available without any other manual tweaking.

8.3 Execution

If MUST was configured with TypeART support and the application was built accordingly, MUST's TypeART checks can be enabled via the option *--must:typeart*. To check whether MUST is configured properly for TypeART checks, run

```
make build-test
lit tests/TypeArt
```

or

```
make check-TypeArt
```

9 RMA Data Race Detection

There is an experimental data race detection mode for MPI RMA provided in MUST. It partly relies on ThreadSanitizer for the execution and is currently only tested with Clang >= 15. The application itself must be compiled with ThreadSanitizer support using the `libonReportLoader.a` as described in Section 3.2, e.g., via the compiler wrappter `must-tsan-cc`, to make it work.

MUST's RMA race detection checks can be enabled via the option *--must:rma-race*.

10 Troubleshooting

The following lists currently known problems or issues and potential workarounds.

10.1 Issues with LD-PRELOAD

In order to use MUST, your application must be linked against the core library of P^nMPI. Per default, MUST will add this library at execution time by using the ld-preload mechanism. If this causes issues, you can use the following command to manually link the P^nMPI library:

```
mpicc source.c -L<PNMPI-INSTALLATION-DIR>/lib  \
        -lpnmpi -o application.exe
```

Important: if you manually link against the MPI library, you must add the P^nMPI library first and the MPI library afterwards.

11 Copyright and Contact

MUST is distributed under a BSD style license. For details, see the file LICENSE.txt in its package. MUST uses parts of external code, mostly distributed under BSD style license. In any case the license is indicated in the source file, and in the external directories, a LICENSE file can be found. Finally, P^nMPI is distributed under LGPL license. The license file is located in externals/GTI/externals/PnMPI/LICENSE.

Contact *must-feedback@lists.rwth-aachen.de* for bug reports, feedback, and feature requests.