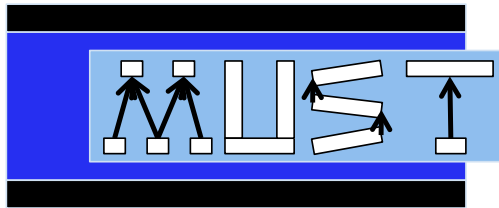


MUST

MPI Runtime Error Detection Tool



April 18, 2012

Contents

1	Introduction	3
2	Installation	3
2.1	P ⁿ MPI	4
2.2	GTI	4
2.3	MUST	5
2.4	Environmentals	5
3	Usage	5
3.1	Execution	5
3.2	Results	6
4	Example	6
4.1	Execution with MUST	7
4.2	Output File	7
5	Included Checks	9
6	Optional: MUST Installation with Dyninst	10
7	Troubleshooting	11
7.1	Issues with Ld-Preload	11
8	Copyright and Contact	11

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 certain system or MPI implementation.

To detect errors, MUST intercepts the MPI calls that are 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:

- PⁿMPI
- GTI
- MUST

The PⁿMPI 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.

Each MUST installation is build with a certain compiler and MPI library. It should only be used for applications that are built with the same pair of 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.

All three packages require 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 your a local version, which requires no root privileges. We suggest to use CMake version 2.8 or later (use *cmake --version*).

Further, in order to augment the MUST output with call stack information, which is very helpful for pinpointing errors, it is possible to utilize Dyninst. In that case MUST uses the Stackwalker API from Dyninst to read and print

stacktraces for errors. As the installation of Dyninst is often non-trivial we suggest this for more experienced users or administrators only. Section 6 presents the necessary steps for such an installation.

2.1 PⁿMPI

PⁿMPI can be build as follows:

```
gunzip pnmpi.tar.gz
tar -xf pnmpi.tar
cd pnmpi
mkdir BUILD
cd BUILD
CC=<C-COMPILER> CXX=<C++-COMPILER> FC=<F90-COMPILER> \
cmake ../ \
    -DCMAKE_INSTALL_PREFIX=<PNMPI-INSTALLATION-DIR> \
    -DCMAKE_BUILD_TYPE=Release
make
make install
```

The CMake call will determine your MPI installation in order to configure PⁿMPI 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 `cmake-modules/FindMPI.cmake`.

Further, PⁿMPI will require a Python installation of version 2.7 or later.

2.2 GTI

GTI can be build as follows:

```
gunzip gti.tar.gz
tar -xf gti.tar
cd gti
mkdir BUILD
cd BUILD
CC=<C-COMPILER> CXX=<C++-COMPILER> FC=<F90-COMPILER> \
cmake ../ \
    -DCMAKE_INSTALL_PREFIX=<GTI-INSTALLATION-DIR> \
    -DCMAKE_BUILD_TYPE=Release \
    -DPnMPI_HOME=<PNMPI-INSTALLATION-DIR>
make
make install
```

If you specified extra arguments for the MPI detection when installing PⁿMPI, you must also add these arguments for the *cmake* call of the GTI configuration. Further, GTI will require developer headers for *libxml2* that should be available on most systems.

2.3 MUST

MUST is built as follows:

```
gunzip must.tar.gz
tar -xf must.tar
cd must
mkdir BUILD
cd BUILD
CC=<C-COMPILER> CXX=<C++-COMPILER> FC=<F90-COMPILER> \
cmake ../ \
    -DCMAKE_INSTALL_PREFIX=<MUST-INSTALLATION-DIR> \
    -DCMAKE_BUILD_TYPE=Release \
    -DGTI_HOME=<GTI-INSTALLATION-DIR>
make
make install
```

The installation of MUST relies almost completely on the settings specified when installing GTI. Usually no extra arguments are needed. You can specify *-DENABLE_TESTS=On* to activate the test suite that is included in MUST.

2.4 Environmentals

You must add *<MUST-INSTALLATION-DIR>/bin* to your *PATH* variable to allow convenient usage of MUST. Further, you should make sure that you load the version of CMake that was used to build MUST.

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 afterwards. It uses a default set of correctness checks and a communication system where one MPI process is used to drive some of MUST's correctness

checks. So when submitting a batch job, you should be 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 4 tasks, but requires one additional task, i.e. it will actually invoke *mpiexec* with *-np 5*.

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. On many machines the *mpiexec* command can differ on batch jobs, add *--must:mpiexec <MPIEXEC>* to the *mustrun* command to override the default. If you encounter errors during execution, please submit error reports where you use *--must:verbose* as an argument to *mustrun*.

3.2 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.

4 Example

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

```

1 #include <stdio.h>
2 #include <mpi.h>
3
4 int main (int argc, char** argv)
5 {
6     int rank,
7       size,
8       sBuf[2] = {1,2},
9       rBuf[2];
10    MPI_Status status;
11    MPI_Datatype newType;
12
13    MPI_Init(&argc,&argv);
14    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
15    MPI_Comm_size (MPI_COMM_WORLD, &size);

```

```

16
17 //Enough tasks ?
18 if (size < 2)
19 {
20     printf ("This_test_needs_at_least_2_processes!\n");
21     MPI_Finalize();
22     return 1;
23 }
24
25 //Say hello
26 printf ("Hello ,_I_am_rank_%d_of_%d_processes.\n", rank, size);
27
28 //1) Create a datatype
29 MPI_Type_contiguous (2, MPI_INT, &newType);
30 MPI_Type_commit (&newType);
31
32 //2) Use MPI_Sendrecv to perform a ring communication
33 MPI_Sendrecv (
34     sBuf, 1, newType, (rank+1)%size, 123,
35     rBuf, sizeof(int)*2, MPL_BYTE, (rank-1+size) % size, 123,
36     MPI_COMM_WORLD, &status);
37
38 //3) Use MPI_Send and MPI_Recv to perform a ring communication
39 MPI_Send (sBuf, 1, newType, (rank+1)%size, 456, MPI_COMM_WORLD);
40 MPI_Recv (rBuf, sizeof(int)*2, MPL_BYTE, (rank-1+size) % size,
41         456, MPI_COMM_WORLD, &status);
42
43 //Say bye bye
44 printf ("Signing_off ,_rank_%d.\n", rank);
45
46 MPI_Finalize ();
47
48 return 0;
49 }
50 /*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 4 processes, will need resources for 5 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 mis-match (Figure 1)

Rank	Thread	Type	Message	From	References	MPI-Standard Reference
1		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). The send operation was started at reference 1, the receive operation was started at reference 2. (Information on communicator: MPL_COMM_WORLD) (Information on send of count 1 with type:Datatype created at reference 3 is for C, committed 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 from: #0 main@test.c:54 #1 start_main@libc.so	reference 1 rank 0: MPI_Sendrecv from: #0 main@test.c:54 #1 start_main@libc.so reference 2 rank 1: MPI_Sendrecv from: #0 main@test.c:54 #1 start_main@libc.so reference 3 rank 0: MPI_Type_contiguous from: #0 main@test.c:35 #1 start_main@libc.so reference 4 rank 0: MPI_Type_commit from: #0 main@test.c:51 #1 start_main@libc.so	

Figure 1: Type mis-match error report from MUST.

- A send-send deadlock (Figure 2)
- A leaked datatype (Figure 3)

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.

The example shows the specification of the location in the datatype that mismatches. The location `(CONTIGUOUS)[0]` (`MPI_INT`) means that the used datatype is of contiguous kind, the mismatch is within the first child 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 basetype `MPI_CHAR`.

The second error results from the application calling send calls that can lead to deadlock (Figure 2). 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 a 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 2), 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`

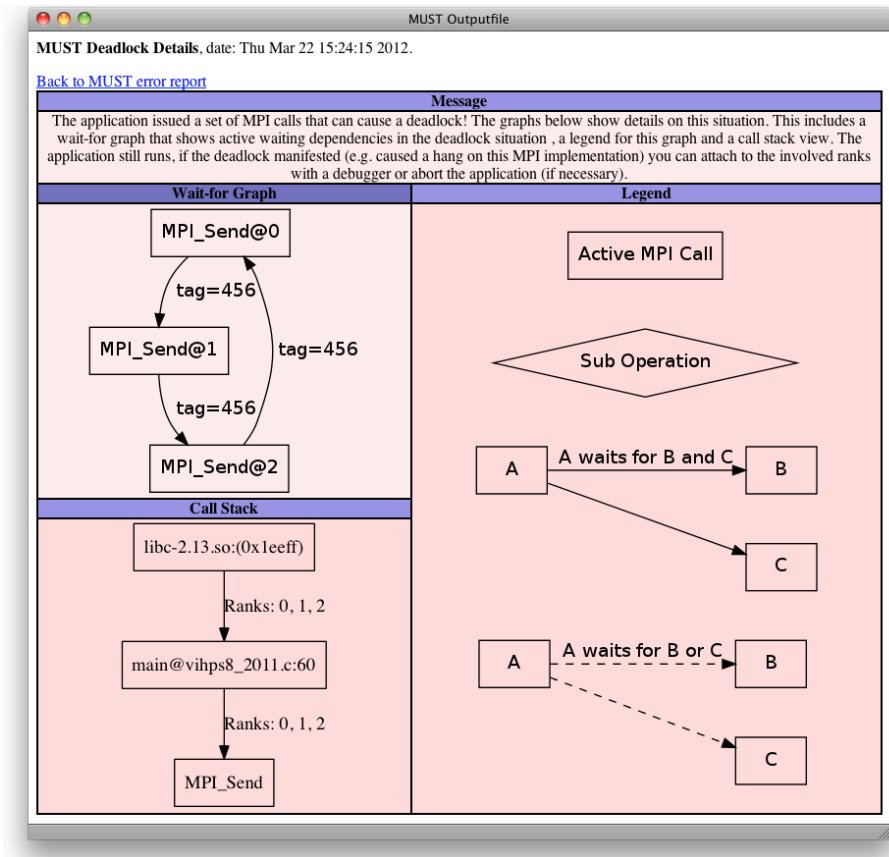


Figure 2: Send-send deadlock report from MUST.

`MUST_Deadlock.dot -o deadlock.ps` after installing this tool. You can open the file `deadlock.ps` with the post script viewer of your choice (DOT also supports additional output formats). If MUST was configured with Dyninst (Section 6), it will also print a parallel call stack in a file called `MUST_DeadlockCallStack.dot`, which Figure 2 shows at the bottom.

Finally, MUST detects that the application leaks MPI resources when calling `MPI_Finalize`, which is a not freed datatype in this case. Applications should free all such resources before invoking `MPI_Finalize`, as leaks are easier to detect in such cases.

5 Included Checks

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

- Constants and integer values

Rank	Thread	Type	Message	From	References	MPI-Standard Reference
		Error	<p>There are 1 datatypes that are not freed when MPI_Finalize was issued, a quality application should free all MPI resources before calling MPI_Finalize. Listing information for these datatypes:</p> <p>-Datatype 1: Datatype created at reference 1 is for C, committed at reference 2, based on the following type(s): { MPI_INT }Typemap = {(MPI_INT, 0), (MPI_INT, 4)}</p>		<p>reference 1 rank 2: MPI_Type_contiguous from: #0 main@test.c:50 #1 start_main@libc.so</p> <p>reference 2 rank 2: MPI_Type_commit from: #0 main@test.c:51 #1 start_main@libc.so</p>	
		Error	<p>There are 1 datatypes that are not freed when MPI_Finalize was issued, a quality application should free all MPI resources before calling MPI_Finalize. Listing information for these datatypes:</p> <p>-Datatype 1: Datatype created at reference 1 is for C, committed at reference 2, based on the following type(s): { MPI_INT }Typemap = {(MPI_INT, 0), (MPI_INT, 4)}</p>		<p>reference 1 rank 1: MPI_Type_contiguous from: #0 main@test.c:50 #1 start_main@libc.so</p> <p>reference 2 rank 1: MPI_Type_commit from: #0 main@test.c:51 #1 start_main@libc.so</p>	

Figure 3: Resource leak report from MUST.

- 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

6 Optional: MUST Installation with Dyninst

In order to install MUST with Dyninst support a full Dyninst installation or a separate installation of the Dyninst Stackwalker API is needed. This usually requires an installation of libdwarf. Installation instructions for these can be found on the Dyninst website¹. We suggest to install libdwarf as a shared library (*--enable-shared* during its configure).

After a successful installation of the Stackwalker API it is necessary to configure MUST to use this installation. Use the following CMake variables:

¹<http://www.dyninst.org/>

- **-DUSE_CALLPATH=On** Enables the feature
- **-DCALLPATH_STACKWALKER_HOME=** Should point to the directory used for Stackwalker API installation (i.e. prefix given to its configure)
- **-DCALLPATH_STACKWALKER_PLATFORM=** Usually *x86_64-unknown-linux2.4* depends on your platform
- **-DCALLPATH_STACKWALKER_EXTRA_LIBRARIES=** Additional libraries that are needed, if libdwarf was built statically you will need to add an absolute filepath to this lib here

Afterwards run *make* and *make install* to build and install MUST. When running MUST no additional steps are needed. However, the stackwalker library will only be able to extract source file names and line numbers if the application was built with the debugging flag *-g*. Otherwise, it will list symbol addresses and library names instead.

Note that MUST expects that the shared libraries for the Stackwalker API and libdwarf (if built as a shared library) are in the `LD_LIBRARY_PATH`.

7 Troubleshooting

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

7.1 Issues with Ld-Preload

In order to use MUST, your application must be linked against the core library of PⁿMPI. 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 PⁿMPI 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 PⁿMPI library first and the MPI library afterwards.

8 Copyright and Contact

MUST is distributed under a BSD style license, for details see the file `LICENSE.txt` in its package. Also, MUST uses parts of the callpath library from LLNL, its also uses a BSD style license, which can be found in the file `modules/Callpath/LICENSE`. Further, MUST uses parts of LLNL's adept utils which have a BSD style license too, it is listed in the respective source files.

Contact tobias.hilbrich@tu-dresden.de for bug reports, feedback, and feature requests.