

# MUST (v1.4)

MPI Runtime Error Detection Tool



March 25, 2015

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Installation</b>	<b>3</b>
2.1	Prerequisites to build and use MUST . . . . .	4
2.2	P <sup>n</sup> MPI . . . . .	4
2.3	GTI . . . . .	5
2.4	MUST . . . . .	5
2.5	Environmentals . . . . .	6
<b>3</b>	<b>Usage</b>	<b>6</b>
3.1	Execution . . . . .	6
3.2	Results . . . . .	7
<b>4</b>	<b>Example</b>	<b>7</b>
4.1	Execution with MUST . . . . .	8
4.2	Output File . . . . .	9
<b>5</b>	<b>MUST's Operation Modes</b>	<b>12</b>
5.1	Mode Overview . . . . .	12
5.2	Mode Details . . . . .	13
<b>6</b>	<b>Included Checks</b>	<b>14</b>
<b>7</b>	<b>Optional: MUST Installation with Dyninst</b>	<b>15</b>
<b>8</b>	<b>Integration with DDT</b>	<b>15</b>
8.1	Workflow . . . . .	16
8.2	MUST Command Modifications . . . . .	17
8.3	The Reproduce Mode . . . . .	18
<b>9</b>	<b>Troubleshooting</b>	<b>19</b>
9.1	Issues with Ld-Preload . . . . .	19
9.2	Issues with stackwalkerAPI . . . . .	19
<b>10</b>	<b>Copyright and Contact</b>	<b>20</b>

## 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:

- P<sup>n</sup>MPI
- GTI
- MUST

The P<sup>n</sup>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.

Each MUST installation is built 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 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 7 presents the necessary steps for such an installation.

In general all three packages support 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 2.8 or newer, see `cmake --version`)
- python (required 2.6 or newer, see `python -V`)
- libxml2 with headers (libxml2-dev / libxml2-devel, required)
- graphviz (optional, to generate graphs)
- dyninst (optional, see section 7)
- a browser (optional, to view html output)
- MPI library, used by the application (required)

## 2.2 P<sup>n</sup>MPI

P<sup>n</sup>MPI 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 install
export PATH=<PNMPI-INSTALLATION-DIR>/bin:$PATH
```

In many cases it's 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 P<sup>n</sup>MPI 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`.

## 2.3 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
make install
export PATH=<GTI-INSTALLATION-DIR>/bin:$PATH
```

If you specified extra arguments for the MPI detection when installing P<sup>n</sup>MPI, you must also add these arguments for the CMake call of the GTI configuration. CMake will detect a P<sup>n</sup>MPI installation automatically if P<sup>n</sup>MPI's binary directory is included in the *PATH* environment variable, otherwise provide the P<sup>n</sup>MPI installation directory to CMake with *-DP<sup>n</sup>MPI\_INSTALL\_PREFIX=<PNMPI-INSTALLATION-DIR>*.

## 2.4 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
make install
```

The installation of MUST relies almost completely on the settings specified when installing GTI. CMake will detect the previous GTI installation if GTI's binary directory is included in the *PATH* environment variable, otherwise provide the GTI installation directory to CMake with *-DGTI\_INSTALL\_PREFIX=<GTI-INSTALLATION-DIR>*. 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 with:

```
ctest
```

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

```
ctest -VV -R ^<TEST-NAME>$
```

For the test named *basic*:

```
ctest -VV -R ^basic$
```

## 2.5 Environmentals

To work with MUST, it is sufficient to add `<MUST-INSTALLATION-DIR>/bin` to your *PATH* variable. Binary paths of P<sup>n</sup>MPI and GTI are just needed at installation time.

## 3 Usage

The following two steps allow you to use MUST:

- Replace the *mpirexec* 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 *mpirexec* 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.

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 *mpirexec* command like:

```
mpirexec -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 *mpirexec* with *-np 5*.

For an example where the *mpirexec* command and the switch used to specify the number of process 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
```

*Advice for sysadmins:* Use `cmake -DMPIEXEC=srun` to configure the appropriate `mpiexec` command.

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
```

For some machines it is necessary to list environmental variables (exports) on invocation of `mpiexec`. Use `--must:exportflag` to specify the needed flag. If your `mpiexec` call looks like

```
mpiexec -x LD_LIBRARY_PATH application.exe
```

to ensure, that `LD_LIBRARY_PATH` will be exported on the compute node, use

```
mustrun --must:exportflag '-x' -np 4 application.exe
```

to tell MUST what flag to use.

*Advice for sysadmins:* Use `cmake -DMPIEXEC_EXPORTFLAG='-x'` to configure the `mpiexec` export flag appropriate for the system.

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*.

## 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, 456,
41             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 }
/*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

```



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 <a href="#">detailed type mismatch view (MUST Output-files/MUST_Typemismatch_0.html)</a> . 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, 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 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.

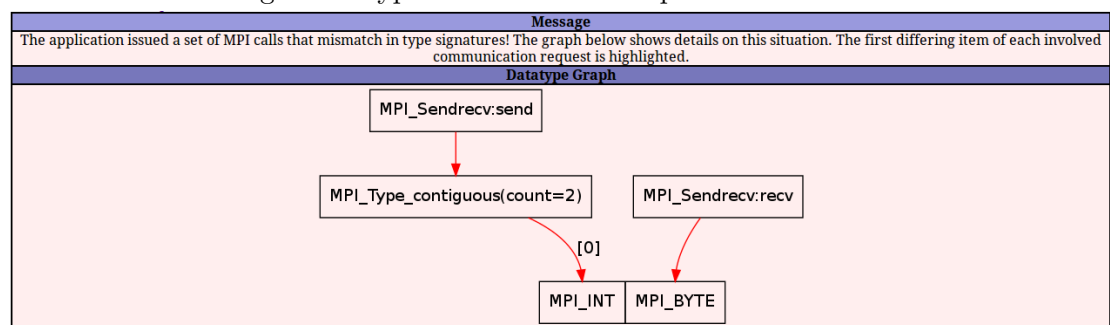


Figure 2: Detail page for the type mismatch in Figure 5.

```
#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 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 Dyninst (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 basetype `MPI_CHAR`.

Figure 2 displays a graphical representation of the type mismatch. The image shows type trees of the involved datatypes. 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 description and graphs are used. In this case all nodes that point 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 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 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 post script viewer of your choice (DOT also supports additional output formats). If MUST was configured with Dyninst (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 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.

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 <a href="#">detailed deadlock view (MUST_Output-files/MUST_Deathlock.html)</a> . 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.

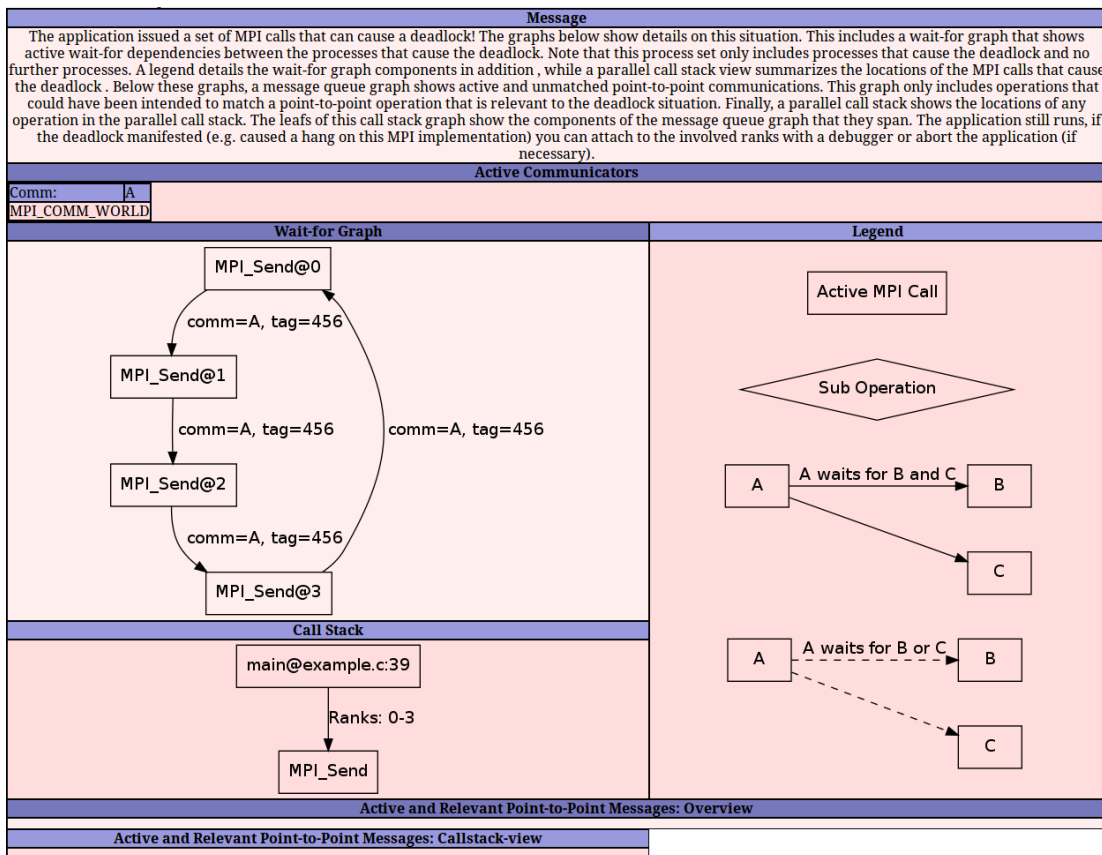


Figure 4: Deadlock view for the send-send deadlock.

<b>Error</b>	<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 0:  <b>MPI_Type_contiguous</b>  called from:  #0 main@example.c:29</p> <p>reference 2 rank 0:  <b>MPI_Type_commit</b> called  from:  #0 main@example.c:30</p>
--------------	--	---

Figure 5: Resource leak report from MUST.

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 crashes
- 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 nodesize  $Y$  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 crashes. 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.

## 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

## 7 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 website<sup>1</sup>. We tested the integration of dyninst in versions 8.1.2. For some systems we identified issues for the older version of dyninst, that are listed in Section 9.2. We suggest to install libdwarf as a shared library (`--enable-shared` during its configure).

As Dyninst's build support is currently limited to GNU compilers, you should build your application and the tool with binary compatible compilers. To build dyninst with compilers other than GNU, make sure to set the variables `CC`, `CXX`, `GXX`, `LD` and `LINKER` for both, the configure and the make step (this is not supported).

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

- `-DUSE_CALLPATH=On` Enables the feature
- `-DSTACKWALKER_INSTALL_PREFIX=` Should point to the directory used for Stackwalker API installation (i.e. prefix given to its configure)
- `-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 libdwarf (if built as a shared library) and libelf are in the `LD_LIBRARY_PATH`.

## 8 Integration with DDT

MUST enables integration with Allinea DDT in order to highlight the MPI usage errors that it detects within a full-fledged parallel debugger, such as Allinea DDT. This allows a closer investigation of the surroundings of an MPI usage error, e.g., to understand

---

<sup>1</sup><http://www.dyninst.org/>

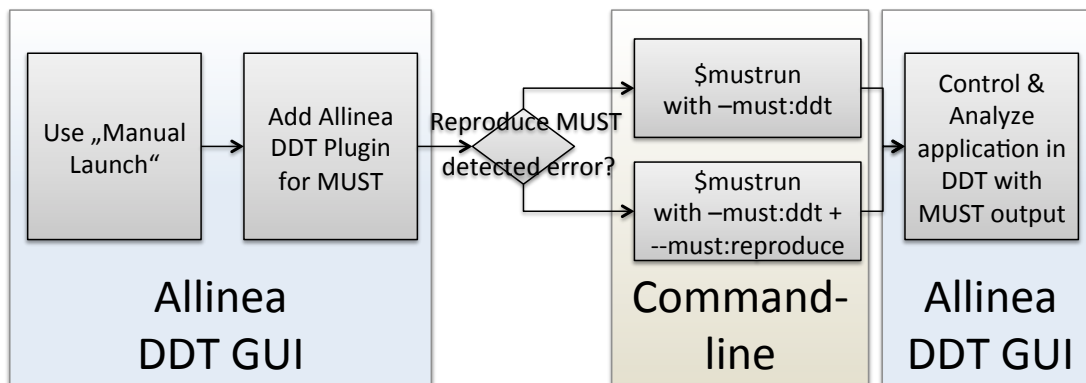


Figure 6: Workflow to combine Allinea DDT with MUST.

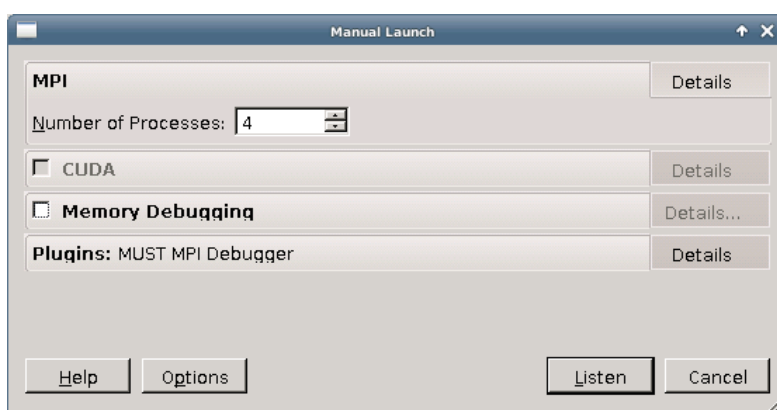


Figure 7: Plugin selection in Allinea DDT's manual launch mode.

how a specific execution state of an application issues the incorrect MPI call. The user can investigate all arguments and variables that influence the MPI call, which will often simplify error diagnosis.

*Advice:* This MUST release includes all the features that the integration with Allinea DDT requires. However, to the best of our knowledge, current production Allinea DDT releases do not yet carry the extensions that enable the integration within the debugger. If you want to use this functionality, please contact [must-feedback@lists.rwth-aachen.de](mailto:must-feedback@lists.rwth-aachen.de)

## 8.1 Workflow

Figure 6 summarizes the workflow that enables an integration of Allinea DDT with MUST. The workflow starts with initiating the Allinea DDT GUI and selecting the so-called “Manual Launch” mode within the GUI. In this mode, the user starts the target application manually, rather than letting DDT start the application. Thus,



the application can conveniently be started with *mustrun* instead of *mpiexec*. In the configuration for the manual launch mode, the user must select the MUST plugin for Allinea DDT, which is highlighted in Figure 7. Once configured, the “Listen” button lets Allinea DDT wait until the target application is started. From this point, the workflow continues on a command line interface or within a bash shell script. The next step is to start the application with MUST attached. Figure 6 highlights a choice in whether to *reproduce* MPI usage errors from a previous application run, or to start a regular run. The need for this distinction is detailed subsequently. Depending on the choice for reproducing errors, the user starts the target application with *mustrun* and adds either the flags `--must:ddt` or `--must:ddt --must:reproduce`, e.g.:

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

Once this command is initiated, the application starts to execute and Allinea DDT automatically connects to the application. The execution is then controlled within the Allinea DDT GUI, as in a regular Allinea DDT use case. The only distinction is that all correctness reports (detected MPI usage errors or warnings) are reported within the Allinea DDT GUI. Additionally, these reports trigger breakpoints that allow the user a detailed investigation of the surroundings of these errors.

## 8.2 MUST Command Modifications

The integration between Allinea DDT and MUST is controlled with four switches of the *mustrun* command:

- `--must:ddt`
  - Enables the integration with a connection to an instance of the Allinea DDT GUI.
- `--must:reproduce`
  - See reproduce mode description in the next section.
- `--must:capture`
  - See reproduce mode description in the next section.
- `--must:precmdddt`
  - By default MUST uses the command line tool *ddt-client*, which is provided by the Allinea DDT installation, to connect with the debugger GUI.
  - If this command is not in the default path or uses a different name, then this switch allows the user to specify the correct command to connect to the debugger.

### 8.3 The Reproduce Mode

In the regular integration mode (`--must:ddt`), the MUST tool can report most types of correctness errors directly to Allinea DDT. However, MUST detects some types of correctness errors—e.g., type matching errors or deadlocks—in an asynchronous and offloaded manner. These errors can not immediately be reported to Allinea DDT. For that case, MUST automatically creates a specialized error report that stores information on the correctness reports. A subsequent run with MUST and Allinea DDT can then visualize and highlight these errors (with `--must:reproduce`).

In summary to be able to show and breakpoint for all usage errors of an MPI application, we suggest the following two run workflow:

- First run to store all errors in a report for reproducing them:

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

- Second run to visualize all reports in Allinea DDT:

```
mustrun --must:ddt --must:reproduce -np 4 application.exe
```

For simplicity, the first run can also be executed without the presence of Allinea DDT, since the second run will highlight all reports of MUST already. The additional switch `--must:capture` specifies that MUST should write out all correctness reports into a file that allows them to be shown in the Allinea DDT GUI in a subsequent run. An example to use this mode for the first run is:

- First run to store all errors in a report for reproducing them (without starting and connecting to Allinea DDT):

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

The two execution workflow also serves a further purpose. When MUST detects an MPI usage error, it reports associated information. If the error involves MPI resources such as communicators, requests, datatypes, and so forth, MUST details which MPI calls created and modified these resources. At the time when MUST detects the error, these calls are already completed, i.e., MUST could not trigger a breakpoint in the parallel debugger for them. However, with the reproduce mode, MUST will breakpoint for all calls that MUST associates with a usage error. As an example, consider that MUST detects a type mismatch between calls to `MPI_Send` and `MPI_Recv`, which use an MPI derived datatype created with a call to `MPI_Type_contiguous`. In the reproduce mode, MUST will issue breakpoints in the debugger when the datatype is created (at the `MPI_Type_contiguous` calls), when the datatype is committed (at the `MPI_Type_commit` calls), and when the mismatch occurs (at `MPI_Send` and `MPI_Recv`). This fine grained reporting allows the user to investigate all the MPI calls that are associated with an MPI usage error. In our example the MPI usage error might be a consequence of an incorrectly constructed MPI datatype, which could be investigated with the breakpoints that are triggered when `MPI_Type_contiguous` gets issued. Consequentially, for the Allinea DDT integration, we suggest users to always use the two run workflow with `--must:capture` in the first run and `--must:ddt --must:reproduce` in the second run.

## 9 Troubleshooting

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

### 9.1 Issues with Ld-Preload

In order to use MUST, your application must be linked against the core library of P<sup>n</sup>MPI. 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<sup>n</sup>MPI 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<sup>n</sup>MPI library first and the MPI library afterwards.

### 9.2 Issues with stackwalkerAPI

**boost related error while build of MUST with dyninst-8:**

```
error: boost/bar_foo.hpp: File not found
```

Solution for boost installation outside of /usr:

Edit *mustsrc/modules/Callpath/CMakeLists.txt* insert around line 21:

```
include_directories(/<BOOST_INSTALL_PREFIX>/include)
```

or add */jBOOST\_INSTALL\_PREFIXj/include* to *CXX\_INCLUDE\_PATH* env variable

**SEGFAULT on execution of *mustrun*:**

```
rank 0 (of 4), pid 12345 caught signal nr 11
```

without any of the WARNINGS listed below. This issue affects almost every installation.

Solution:

Edit *src/dyninst/symtabAPI/src/Object-elf.C* around line 2069 and replace:

```
if(secNumber >= 1 && secNumber <= regions_.size()) {
```

by

```
if(secNumber >= 0 && secNumber < regions_.size()) {
```

... and rebuild / install dyninst

**SEGFAULT on execution of *mustrun*:**

```
rank 0 (of 4), pid 12345 caught signal nr 11
```

without any of the WARNINGS listed below and after fixing the issue above.

**Solution:**

Make sure that you use the same compiler family for building dyninst, P<sup>n</sup>MPI, GTI, MUST and your application!

***mustrun* reports missing library:**

```
WARNING: Can't load module libcallpathModule.so (Error libdwarf.so:  
cannot open shared object file: No such file or directory)
```

**Solution:**

Add `<LIBDWARF-INSTALLATION-DIR>/lib` to `LD_LIBRARY_PATH` (`libdwarf.so` should be located there).

## 10 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 [must-feedback@lists.rwth-aachen.de](mailto:must-feedback@lists.rwth-aachen.de) for bug reports, feedback, and feature requests.