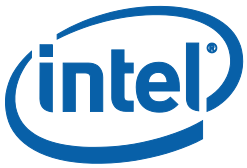


# Tutorial: Detecting Errors with MPI Correctness Checker

## Intel® Trace Analyzer and Collector for Linux* OS

# *Contents*

# Legal Information

# 1. Overview

Intel® Trace Analyzer and Collector enables you to understand MPI application behavior and quickly find bottlenecks to achieve high performance for parallel cluster applications. Intel® Trace Collector generates trace files for MPI applications, while Intel® Trace Analyzer visualizes the MPI application behavior using the generated trace file.

Besides the regular performance analysis, Intel® Trace Analyzer and Collector can perform correctness checking of MPI applications, which can help you:

- Find programming mistakes in the application, including potential portability problems and violations of the MPI standard. Normally, these mistakes do not immediately cause problems, but might when switching to different hardware or a different MPI implementation.
- Detect errors in the execution environment.

| | |
|---|---|
| **About This Tutorial** | This tutorial demonstrates the correctness checking workflow applied to sample MPI applications with the following types of errors:<br>• Data type mismatch<br>• Deadlock<br>You can ultimately apply the same steps to your own application(s). |
| **Estimated Duration** | 10-15 minutes. |
| **Learning Objectives** | After you complete this tutorial, you should be able to:<br>• Run the correctness checker for your own applications<br>• Detect and resolve various errors in your applications |
| **More Resources** | Learn more about the Intel® Trace Analyzer and Collector in the User and Reference Guides available at:<br>• Intel® Trace Collector User and Reference Guide<br>• Intel® Trace Analyzer User and Reference Guide |

You can submit your feedback on the documentation at http://www.intel.com/software/products/softwaredocs_feedback/.

## 1.1. Prerequisites

This section describes the steps you need to do before you start using the Intel® Trace Analyzer and Collector.

### 1.1.1. Required Software

To perform all the steps described in this tutorial, you need the following software installed on your system:

- Intel® compilers
- Intel® MPI Library
- Intel® Trace Analyzer and Collector

All of these products are installed as part of Intel® Parallel Studio XE Cluster Edition.

## 1.1.2. Setting Up the Environment Variables

Set the required environment variables by sourcing the `psxevars.c[sh]` script available at `<install-dir>/parallel_studio_xe_<version>.x.xxx/bin`, where `<install-dir>` is the Intel® Parallel Studio XE Cluster Edition installation directory. For example:

```
$ source psxevars.sh
```

# 2. Detecting and Resolving Errors

To demonstrate the process of correctness checking of MPI applications, this tutorial uses two sample applications that have errors in the source code. All the sample applications eligible for correctness checking are available at: `<install-dir>/examples/checking`. You can use these samples to manually experiment with the functionality using the workflow described here.

Read the topics below to learn how to configure the correctness checker and how to detect and resolve application errors using Intel® Trace Analyzer and Collector.

| | |
|---|---|
| Configuration Options | A list of Intel® Trace Collector configuration options for controlling the correctness checking process. |
| **Example 1:** Data Type Mismatch | A case where the types of data sent and received do not match, while the number of bytes sent is the same. |
| **Example 2:** Deadlock | A case where two processes simultaneously call blocking receive functions, making them unable to call the succeeding sending functions, which causes the so-called deadlock. |

## 2.1. Correctness Checker Configuration Options

The table below lists the environment variables that help you configure the MPI correctness checking. Please, look through them to understand their purpose. They all are used in the examples given in this tutorial.

| Environment Variable | Value | Description |
|---|---|---|
| `VT_DEADLOCK_TIMEOUT <delay>` | `<delay>` - time threshold<br>**Default**: 1m<br>**Examples:**<br>`VT_DEADLOCK_TIMEOUT 1m`<br>`VT_DEADLOCK_TIMEOUT 10s` | If no progress is observed in any process for this amount of time, Intel Trace Collector stops the application and writes a trace file upon reaching this threshold, assuming that a deadlock has occurred.<br>***TIP***<br>For interactive use, set this variable to a small value like “10s” to detect the deadlocks quickly without having to wait long for the timeout. |
| `VT_DEADLOCK_WARNING <delay>` | `<delay>` - time threshold<br>**Default:** 5m<br>**Examples:**<br>`VT_DEADLOCK_WARNING 5m` | Displays a `GLOBAL:DEADLOCK:NO_PROGRESS` warning if the time spent by MPI processes in their last MPI call exceeds the specified threshold. This warning indicates a load imbalance or a deadlock that cannot be detected, which may occur when at least one process polls for progress instead of blocking inside an MPI call. |

| | | |
|---|---|---|
| `VT_CHECK_TRACING <on \| off>` | `<on \| off>`<br>**Default:** off | When set to on, this variable enables you to record all events including any MPI errors found during the run and to create a trace file. |
| `VT_CHECK_MAX_ERRORS <value>` | `<value>` - maximum errors to detect<br>**Default:** 1 | Number of errors that has to be reached by a process before aborting the application. `0` disables the limit. Some errors are fatal and always cause an abort. Errors are counted per-process to avoid the need for communication among processes, as that has several drawbacks, which outweigh the advantage of a global counter. |

# 2.2. Instrumenting an Example with Data Type Mismatch

To experiment with the data type mismatch example, copy the contents of the *<install-dir>*`/itac/examples/checking/global/collective/datatype_mismatch/` directory to your working directory:

```
$ cp -r <install-dir>/itac_latest/examples/checking/global/collective/datatype_mismatch/ ~
$ cd ~/datatype_mismatch
```

Then compile and run the `MPI_Bcast` example located in the directory using the following commands:

```
$ mpiicc -g MPI_Bcast.c -o MPI_Bcast
$ mpirun -n 4 -check_mpi -genv VT_CHECK_MAX_ERRORS 0 MPI_Bcast
```

The command lines above use the following flags:

- `-g` – generate the debugging information in the object file to be able to analyze the source files
- `-check_mpi` – dynamically link the correctness checker library (`VTmc.so`)
- `-genv VT_CHECK_MAX_ERRORS 0` – set the maximum of errors found to unlimited (1 by default)

After running the application you will get the following output:

```
...
[0] ERROR: GLOBAL:COLLECTIVE:DATATYPE:MISMATCH: error
[0] ERROR:    Mismatch found in local rank [1] (global rank [1]),
[0] ERROR:    other processes may also be affected.
[0] ERROR:    No problem found in local rank [0] (same as global rank):
[0] ERROR:       MPI_Bcast(*buffer=0x7fff1066e814, count=1, datatype=MPI_INT,
root=0, comm=MPI_COMM_WORLD)
[0] ERROR:       main
(/checking/global/collective/datatype_mismatch/MPI_Bcast.c:50)
[0] ERROR:    1 elements transferred by peer but 4 expected by
[0] ERROR:    the 3 processes with local ranks [1:3] (same as global ranks):
[0] ERROR:       MPI_Bcast(*buffer=..., count=4, datatype=MPI_CHAR, root=0,
comm=MPI_COMM_WORLD)
[0] ERROR:       main
(/checking/global/collective/datatype_mismatch/MPI_Bcast.c:53)
[0] INFO: GLOBAL:COLLECTIVE:DATATYPE:MISMATCH: found 1 time (1 error + 0
warnings), 0 reports were suppressed
[0] INFO: Found 1 problem (1 error + 0 warnings), 0 reports were suppressed.
```

The highlighted error messages refer to lines 50 and 53 in the `MPI_Bcast.c` source file:

```
...
```

```
39 int main (int argc, char **argv)
40 {
41     int rank, size;
42
43     MPI_Init( &argc, &argv );
44     MPI_Comm_size( MPI_COMM_WORLD, &size );
45     MPI_Comm_rank( MPI_COMM_WORLD, &rank );
46
47     /* error: types do not match */
48     if( !rank ) {
49         int send = 0;
50         MPI_Bcast( &send, 1, MPI_INT, 0, MPI_COMM_WORLD );
51     } else {
52         char recv[4];
53         MPI_Bcast( &recv, 4, MPI_CHAR, 0, MPI_COMM_WORLD );
54     }
55
56     MPI_Finalize( );
57
58     return 0;
59 }
```

The above code example shows a mismatch in the data types within the `MPI_Bcast` function. While you set the sent data type to `int`, the receiver expects a `char`. The number of transferred bytes is the same, so normally this issue is not detected by MPI.

To fix the issue:

- in line 52, change the receiver type from `char` array to `int`.
- in line 53, change the MPI data-type argument from `MPI_CHAR` to `MPI_INT`, and the number of received elements to 1.

```
 52         int recv;
 53         MPI_Bcast( &recv, 1, MPI_INT, 0, MPI_COMM_WORLD );
```

To check that you have eliminated the message checking errors, re-compile and re-run the application:

```
...
[0] INFO: Error checking completed without finding any problems.
...
```

# 2.3. Instrumenting an Example with a Deadlock

To experiment with the deadlock example, copy the contents of the *<install-dir>*`/itac/examples/checking/global/deadlock/hard/` directory to your working directory:

```
$ cp -r <install-dir>/itac_latest/examples/checking/global/deadlock/hard/ ~
$ cd ~/hard
```

Compile and run the example with the following commands:

```
$ mpiicc -g MPI_Recv.c -o MPI_Recv
$ mpirun -check_mpi -genv VT_CHECK_TRACING on -genv VT_DEADLOCK_TIMEOUT 20s -
genv VT_DEADLOCK_WARNING 25s -genv VT_PCTRACE on -n 2 MPI_Recv
```

The command lines above use the following flags:

- `-g` – generate the debugging information in the object file to be able to analyze the source files
- `-check_mpi` – dynamically link the correctness checker library (`VTmc.so`)
- `-genv VT_CHECK_TRACING on` – enable writing of the trace file `.stf` for analyzing in Intel® Trace Analyzer (trace file is not written by default with `VTmc.so`)
- `-genv VT_DEADLOCK_TIMEOUT 20s`, `-genv VT_DEADLOCK_WARNING 25s` – see this section for details
- `-genv VT_PCTRACE on` – enable recording of source code locations to the trace file

The resulting output should look as follows:

```
...
[0] ERROR: no progress observed in any process for over 0:20 minutes, aborting
application
[0] WARNING: starting emergency trace file writing
[0] ERROR: GLOBAL:DEADLOCK:HARD: fatal error
[0] ERROR:    Application aborted because no progress was observed for over 0:20
minutes,
[0] ERROR:    check for real deadlock (cycle of processes waiting for data) or
[0] ERROR:    potential deadlock (processes sending data to each other and
getting blocked
[0] ERROR:    because the MPI might wait for the corresponding receive).
[0] ERROR:    [0] no progress observed for over 0:20 minutes, process is
currently in MPI call:
[0] ERROR:       MPI_Recv(*buf=0x7fff447cc494, count=1, datatype=MPI_CHAR,
source=1, tag=100, comm=MPI_COMM_WORLD, *status=0x7fff447cc450)
[0] ERROR:       main (/checking/global/deadlock/hard/MPI_Recv.c:53)
[0] ERROR:    [1] no progress observed for over 0:20 minutes, process is
currently in MPI call:
[0] ERROR:       MPI_Recv(*buf=0x7fffaf31b9a4, count=1, datatype=MPI_CHAR,
source=0, tag=100, comm=MPI_COMM_WORLD, *status=0x7fffaf31b960)
[0] ERROR:       main (/checking/global/deadlock/hard/MPI_Recv.c:53)
[0] INFO: Writing tracefile MPI_Recv.stf in /checking/global/deadlock/hard
[0] INFO: GLOBAL:DEADLOCK:HARD: found 1 time (1 error + 0 warnings), 0 reports
were suppressed
[0] INFO: Found 1 problem (1 error + 0 warnings), 0 reports were suppressed.
...
```

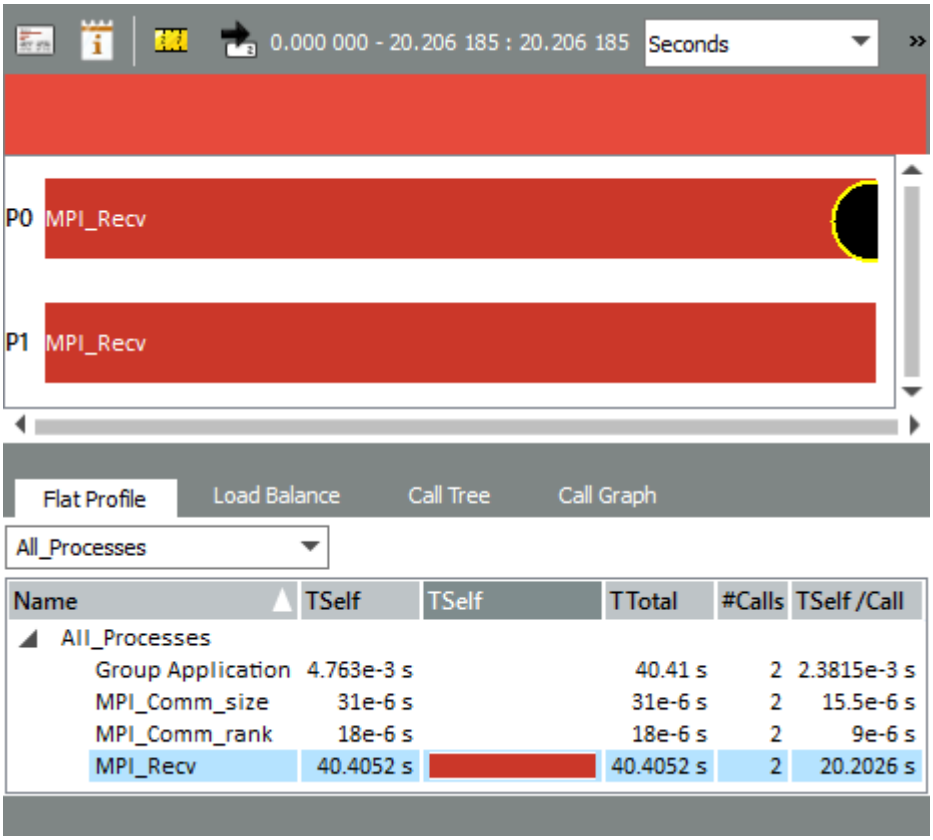
You can observe that the correctness checker reported a deadlock error that needs to be fixed. To dig deeper into the reported problem, analyze the generated `.stf` file in Intel® Trace Analyzer.

## 2.3.1. Analyzing Source Code

You can use the Intel® Trace Analyzer to view the reported deadlock problem. Open the `MPI_Recv.stf` file in Intel® Trace Analyzer:

```
$ traceanalyzer MPI_Recv.stf
```

The trace information may look as follows:

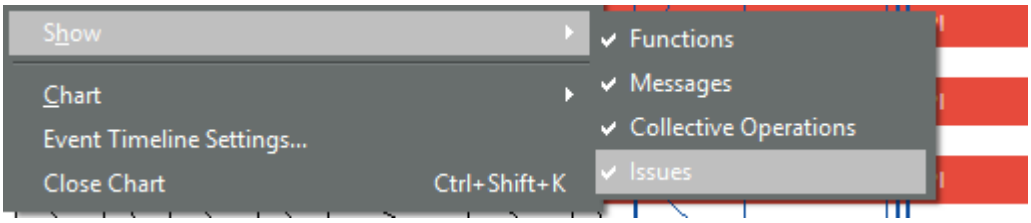



In the **Event Timeline** chart, yellow-bordered circles represent various issues in your application. The color of each circle depends on the type of the particular diagnostic:

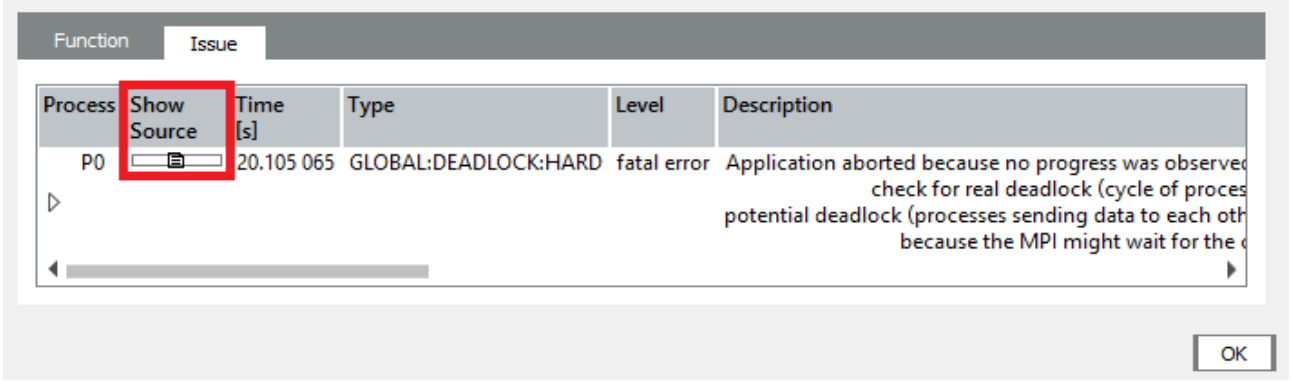
- The black color indicates an error.
- The gray color indicates a warning.

## *TIP*

To suppress error messages and warnings, right-click the **Event Timeline** chart, open the **Show** menu, and uncheck the **Issues** option. The black and gray circles disappear.




To determine which source code line is associated with an error message, right-click the issue on the **Event Timeline** chart and select **Details on Function, Issue** from the context menu. The following dialog will appear:



Click the **Show Source** button shown in the figure above to open the **Source View**. You can see that line 53 is highlighted, which indicates that it causes the deadlock:

```
52      /* waiting for a message that has not been sent */
53      MPI_Recv( &recv, 1, MPI_CHAR, peer, 100, MPI_COMM_WORLD, &status );
54
55      /*
56       * Too late, this code is not going to be reached.
57       * Beware, simply moving this code up would rely on
58       * buffering in the MPI. The correct solution is to
59       * use MPI_Isend() before the receive and MPI_Wait()
60       * afterwards.
61       */
62      send = 0;
63      MPI_Send( &send, 1, MPI_CHAR, peer, 100, MPI_COMM_WORLD );
```

In this example both processes call the blocking `MPI_Recv` function at once, so none of them get to calling the sender function, which causes the deadlock.

## 2.3.2. Resolving the Deadlock

To avoid deadlock situations, you can use the following approaches:

- Reorder MPI communication calls between processes.
- Implement non-blocking calls.
- Use `MPI_Sendrecv` or `MPI_Sendrecv_replace`.
- Use the buffered mode.

The following code section leads to a deadlock in your original application:

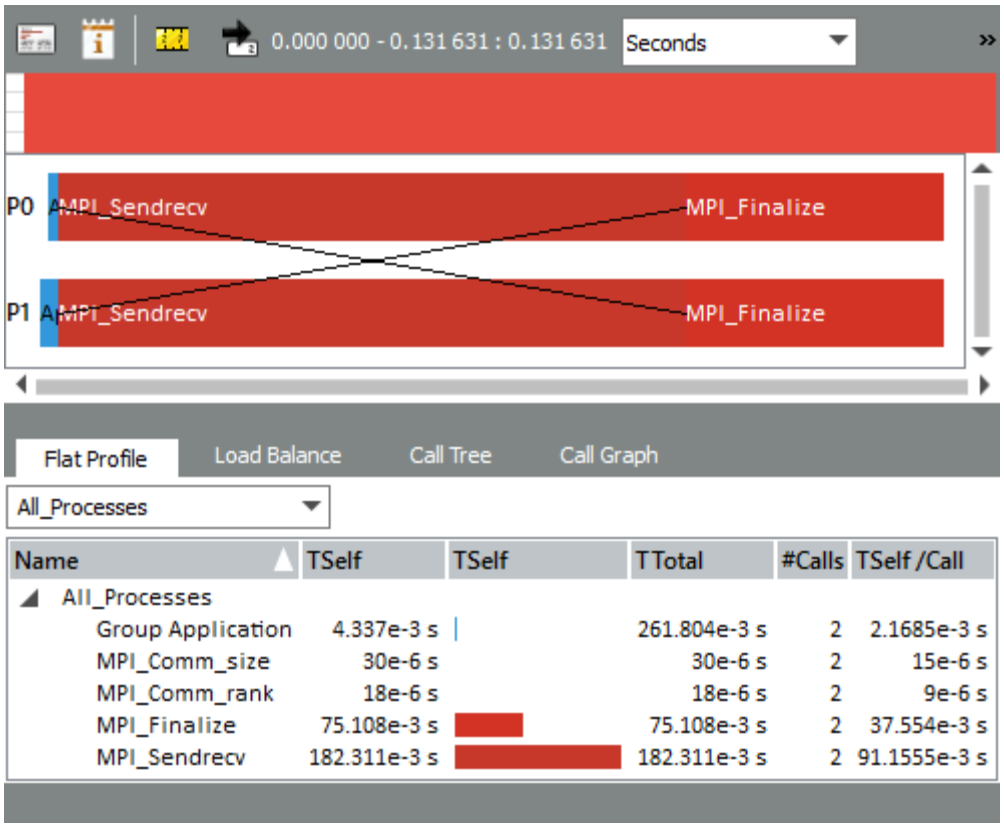
```
...
52      /* waiting for a message that has not been sent */
53      MPI_Recv( &recv, 1, MPI_CHAR, peer, 100, MPI_COMM_WORLD, &status );
54
55      /*
56       * Too late, this code is not going to be reached.
57       * Beware, simply moving this code up would rely on
58       * buffering in the MPI. The correct solution is to
59       * use MPI_Isend() before the receive and MPI_Wait()
60       * afterwards.
61       */
62      send = 0;
63      MPI_Send( &send, 1, MPI_CHAR, peer, 100, MPI_COMM_WORLD );
...
```

To resolve the deadlock for the given example, you need to replace the `MPI_Recv` and `MPI_Send` calls with `MPI_Sendrecv`. Do the following:

1. Remove all lines from 52 to 61.
2. Replace the `MPI_Recv` function call (line 63) with the following `MPI_Sendrecv` call:

   ```
   MPI_Sendrecv( &send, 1, MPI_CHAR, peer, 100, &recv, 1, MPI_CHAR, peer,
   100, MPI_COMM_WORLD, &status );
   ```
3. Save this information into the file: `MPI_Sendrecv.c`.
4. Compile and run the modified application with the same parameters. The resulting output should look as follows:

   ```
     ...
   [0] INFO: Writing tracefile MPI_Sendrecv.stf in
   /checking/global/deadlock/hard
   [0] INFO: Error checking completed without finding any problems.
   ```
5. View the newly generated trace file with Intel® Trace Analyzer to make sure that the deadlock issue has been resolved:




As shown in the figure above, the deadlock problem no longer occurs, and both ranks successfully exchanged the messages.

# 3. Summary

You have completed the *Detecting and Resolving Errors with MPI Correctness Checker* tutorial. The following is the summary of important things to remember when using this functionality to check your MPI application for errors.

| Step | Tutorial Recap | Key Tutorial Take-aways |
|---|---|---|
| **Configuration options overview** | Reviewed the configuration options that control the correctness checking functionality. | You can configure the correctness checking process by adjusting the necessary settings according to your needs. |
| **Resolving the data type mismatch error** | • Ran an application with the data type mismatch error to detect the cause.<br>• Used the correctness checker messages to locate and eliminate the problem. | You can use the correctness checker command-line messages to resolve the reported problems. |
| **Resolving the deadlock error** | • Ran an application with the deadlock error and created its trace file to detect the cause of the problem.<br>• Used the correctness checker messages and Intel® Trace Analyzer Source View to locate and eliminate the problem. | You can:<br>• Store source code locations in trace file to easily find problem causes.<br>• View and analyze the reported problems using Intel® Trace Analyzer GUI. |