Intel® Trace Collector for Intel® oneAPI

User and Reference Guide

Introduction

Intel® Trace Collector is a tool for tracing MPI applications. It intercepts all MPI calls and generates tracefiles that can be analyzed with Intel® Trace Analyzer for understanding the application behavior. Intel® Trace Collector can also trace non-MPI applications, like socket communication in distributed applications or serial programs. The tool was formerly known as Vampirtrace* (VT), which is why the VT abbreviation is present in the names of some of components and variables.

Before you start using any of the Intel Trace Collector functionality, make sure to set the necessary environment variables using the `setvars` script available in the installation directory (the default installation directory is `/opt/intel/inteloneapi)`.
Linux: Source the `setvars` script.
Windows: Run the `setvars.bat` file.
This will set the required environment variables for compilers, Intel® MPI Library and Intel® Trace Analyzer and Collector, and you will be ready to trace your applications.

This guide documents Intel® Trace Collector 2020. Documentation for older versions of Intel® Trace Analyzer and Collector are available for download only. For a list of available documentation downloads by product version, see Download Documentation for Intel Parallel Studio XE.

Intel® Trace Collector contains the libraries and utilities listed below. Some of them are available only on Linux* OS.

Product Components

Libraries

Component	Description
`libVTnull`	Dummy implementation of API (Tracing User Defined Events).
`libVT`	Library for regular MPI tracing (Tracing Conventional MPI Applications).
`libVTfs`	Library for fail-safe MPI tracing (Tracing Failing MPI Applications).
`libVTim`	Library for tracing MPI load imbalance (Tracing MPI Load Imbalance).
`libVTmc`	Correctness checking library (Correctness Checking).
`libVTcs`	Library for tracing distributed non-MPI applications (Tracing Distributed Non-MPI Applications).
`VT_sample`	Library for automatic counter tracing with PAPI* (Recording Hardware Performance Information).

Utilities

Component	Description
`stftool`	Utility for manipulating trace files (stftool Utility).
`xstftool/expandvtlog.pl`	Utility for conversion of trace files into readable format (Expanded ASCII output of STF Files).
`itcconfig`	Configuration assistant for creating and editing configuration files (Configuring Intel® Trace Collector).
`otf2-to-stf`	Utility for conversion of OTF2 trace files to the STF format. See the *Intel® Trace Analyzer User and Reference Guide* for details.

Compatibility with MPI Implementations

Intel® Trace Analyzer and Collector is built with the Intel® MPI Library, depends on certain MPI constants, and is generally compatible with all MPICH* derivatives. If you use a different MPI implementation or an old version of the supported one, you need to check its compatibility with Intel® Trace Collector. Compile and run the `check_compatibility.c` program located at `<install_dir>/examples`:

Linux* OS

```
$ mpicc -o check_compatibility check_compatibility.c
$ mpirun -n 1 ./check_compatibility
```

Windows* OS

```
> mpicc check_compatibility.c
> mpiexec -n 1 check_compatibility.exe
```

The program outputs whether your MPI implementation is compatible with Intel® Trace Collector. For details on supported implementations, refer to the *Release Notes*.

What's New

Intel® Trace Collector 2020

- Bug fixes.

Intel® Trace Collector 2019

- Added information on tracing Python* applications to Tracing Conventional MPI Applications.

- Added new configuration options VT_START_PAUSED and VT_COMPRESS_TRACE.

Intel® Trace Collector 2018

- Introduced support for OpenSHMEM* applications (Linux* OS only). See Tracing OpenSHMEM Applications for details.
- Removed support of the Intel® Xeon Phi™ coprocessors (formerly code named Knights Corner).
- Removed the macOS* support.
- Removed support for the indexed tracefile format (ITF).

Intel® Trace Collector 2017

- Introduced the `otf2-to-stf` utility for converting OTF2 trace files to the STF format. See the *Intel® Trace Analyzer User and Reference Guide* for details.
- Introduced a new library for collecting MPI load imbalance (Linux* OS only). See Tracing MPI Load Imbalance for details.
- Introduced a new API function VT_registerprefixed.

About This Document

This *User and Reference Guide* provides you with the description of the features of the Intel® Trace Collector. This information is provided in the two main sections:

- User GuideUser Guide – describes the Intel® Trace Collector functionality and provides instructions on how to use its features.
- Intel® Trace Collector ReferenceIntel® Trace Collector Reference – contains the reference information for Intel® Trace Collector.

On Linux* OS, you can get help information in man pages, for example, about the Intel® Trace Collector API calls (`man VT_enter`) and the Intel Trace Collector configuration (`man VT_CONFIG`). The man pages are available in the `<install_dir>/man` directory.

Note

Information in this document is provided with the assumption that you use the latest versions of Intel® C++/Fortran Compiler and Intel® MPI Library, unless another compiler or MPI implementation is specified.

Notational Conventions

The documentation is OS-independent. Linux* OS and Windows* OS may have different styles in passing parameters. This *User and Reference Guide* follows the nomenclature used on the Linux* OS. Here is a list of the most important differences and how they are mapped from the Linux* OS style to the Windows* OS one:

Linux* OS	Microsoft* Windows* OS
`-L<path>`	`-LIBPATH:<path>`
`-l<library>`	`<library>.lib`
`<directory>/<file>`	`<directory>\<file>`

The following conventions are used in this document:

Convention	Explanation	Example
This type style	Document or product names	The term *process* in this documentation implicitly includes thread.
This type style	GUI elements	Click **OK**
`<this type style>`	Placeholders for actual values	`<new_name>`
`This type style`	Commands, arguments, options	`$ mpirun -trace -n 4 myApp`
`THIS_TYPE_STYLE`	Environment variables	Set the `VT_CONFIG` environment variable to the directory that contains the configuration file.
`[ items ]`	Optional items	`[config options]`
`[ item | item ]`	Selectable items separated by vertical bar(s)	`[on|off]`
`$`	Introduces UNIX* commands	`$ ls`
`>`	Introduces Windows* commands	`> cd`

Related Information

Additional information about Intel® Trace Analyzer and Collector is available at: https://software.intel.com/en-us/intel-trace-analyzer/

For a complete list of related documentation visit http://software.intel.com/en-us/articles/intel-trace-analyzer-and-collector-documentation-2018-beta/

Information about Intel® Parallel Studio XE Cluster Edition is available at: https://software.intel.com/en-us/intel-parallel-studio-xe/

Intel® Premier Customer Support is available at: https://premier.intel.com/

User Guide

This section describes the Intel® Trace Collector functionality and provides instructions on how to use its features. See the brief overview of each sub-section in the table below.

Section	Description
Tracing MPI Applications	General instructions on how to trace various types of MPI applications.
Tracing MPI Load Imbalance (Linux* OS)	Information on tracing MPI events that cause application load imbalance.
Tracing User Defined Events	Information on tracing non-MPI user-defined events in MPI applications. You can do this automatically for all application functions, or manually only for selected functions or code regions.
Configuring Intel® Trace Collector	Information on how to configure various aspects of Intel® Trace Collector behavior. Configuration is used to enable and disable some Intel Trace Collector functionality and for filtering trace data.
Filtering Trace Data	Description of the filtering capabilities of Intel® Trace Collector. Use filtering to trace only information of interest reducing the trace file size and making the results easier to analyze.
Recording OpenMP* Regions Information	Instructions on recording the information about the OpenMP* regions in your application into the trace file.
Tracing System Calls (Linux* OS)	Information on tracing system input/output calls.
Collecting Lightweight Statistics	Information on collecting the lightweight statistics about function calls and their communication. Collecting the lightweight statistics is useful for understanding an unknown application.
Recording Source Location Information	Instructions on recording the locations of certain functions in the source code. This enables you to easily navigate the source files when analyzing the trace data.
Recording Hardware Performance	Information on recording the PAPI hardware performance counters.

Information (Linux* OS)	
Recording Operating System Counters	Information on recording operating system counters, which provide useful information about nodes.
Tracing Library Calls	A use case of tracing particular data using various Intel® Trace Collector capabilities. In the example provided, an instrumented application with the use of external libraries is used. Data tracing is presented from the points of view of the application developer and the library developers.
Correctness Checking	Information on the correctness checking capability. The correctness checker can detect deadlocks, data corruption, and errors with MPI parameters, data types, buffers, communicators, point-to-point messages and collective operations.
Tracing Distributed Non-MPI Applications	Information on tracing distributed applications that work without MPI.

Tracing MPI Applications

Tracing MPI Applications

Before tracing your applications, set up the environment variables for compilers, Intel® MPI Library, and Intel® Trace Analyzer and Collector. This can be done by sourcing/running a single file that sets the variables for all products at once.

Linux* OS:

Source the `setvars.[c]sh` script available in the installation directory (the default installation directory is /opt/intel/inteloneapi:

```
$ source setvars.sh
```

Windows* OS:

Run the `setvars.bat` script available in the installation directory (the default installation directory is `C:\Program Files (x86)\inteloneapi`).
After setting up the environment, you are ready to trace your applications. See the instructions below:.

- Tracing Conventional MPI Applications
- Tracing Failing MPI Applications
- Tracing MPI File IO
- Handling of Communicator Names

Tracing Conventional MPI Applications

Before tracing your applications make sure to complete the steps described in the previous section.

Tracing MPI Applications in Python*

To create a trace file for an MPI application written in Python*, use the `-trace` option with `"libVT.so libmpi.so"` argument. For example:

```
$ mpiexec.hydra -trace "libVT.so libmpi.so" -n 2 python helloworld.py
```

To change the trace name, see the `VT_LOGFILE_NAME` variable or refer to the example below:

```
$ export VT_LOGFILE_NAME=helloworld.stf
$ export VT_LOGFILE_FORMAT=SINGLESTF
```

Tracing on Linux* OS

Tracing without rebuilding

The common way to trace an MPI application is to dynamically load the Intel® Trace Collector profiling library during execution. The profiling library then intercepts all MPI calls and generates a trace file. The easiest way to do this is to use the `-trace` option of the `mpirun` command. For example:

```
$ mpirun -trace -n 4 ./myApp
```

If you use your own launch scripts, you can use the `LD_PRELOAD` environment variable to point to the appropriate profiling library (see Product Components). For example, for regular tracing:

```
$ export LD_PRELOAD=libVT.so
$ mpirun -n 4 ./myApp
```

Relinking with profiling library

In some cases you may need to rebuild your application to trace it – for example, if it is statically linked with the MPI library. In this case, use the `-trace` compiler option to link the libraries required for tracing. To generate a trace file, run the application as usual. For example:

```
$ mpiicc -trace myApp.c -o myApp
$ mpirun -n 4 ./myApp
```

If you wish to specify the profiling library, use the `-profile=<profiling_library>` option instead of `-trace`. For the list of available libraries, see Product Components. For example, for the fail-safe tracing library:

```
$ mpiicc -profile=vtfs myApp.c -o myApp
```

Note

The `-trace` and `-profile` options link the selected Intel® Trace Collector library statically. To link it dynamically, use the additional `-dynamic_log` option.

For more details on the options used, see the *Intel® MPI Library* documentation.

Tracing on Windows* OS

To trace an application on Windows* OS, you need to recompile your application and link it with the Intel® Trace Collector profiling library. The `-trace` compiler option helps you do this job.
Do the following:

1. Rebuild your application with the `-trace` compiler option. For example:
   ```
   > mpiicc -trace myApp.c
   ```
2. Run the application as usual:
   ```
   > mpiexec -n 4 myApp.exe
   ```

After running your application, a trace file with the `.stf` extension is created. Open this trace file in Intel® Trace Analyzer to analyze the application behavior. See the *Intel® Trace Analyzer User and Reference Guide* for details.

Tracing Failing MPI Applications

Normally, if an MPI application fails or is aborted, all the trace data collected is lost, because `libVT` needs a working MPI to write the trace file. However, the user might want to use the data collected up to that point. To solve this problem, Intel® Trace Collector and Analyzer provides the `libVTfs` library that enables tracing of failing MPI applications.

Usage Instructions

To trace failing MPI applications, do the following:

Linux* OS

Set the `LD_PRELOAD` environment variable to point to the `libVTfs` library and run the application. For example:

```
$ export LD_PRELOAD=libVTfs.so
$ mpirun -n 4 ./myApp
```

Alternatively, rebuild your application with the static version of the library. For example:

```
$ mpiicc -profile=vtfs myApp.c -o myApp
```

Windows* OS

Relink your application with the `libVTfs` library before the MPI library and run it as usual. To do this, you should create an Intel® MPI Library configuration file that points to the `libVTfs` library. You can do it as follows (administrator privileges may be required):

```
> echo SET PROFILE_PRELIB=%VT_ROOT%\lib\VTfs.lib > %I_MPI_ROOT%\lib\VTfs.conf
> mpiicc -profile=VTfs myApp.c
> mpiexec -n 4 myApp.exe
```

How it Works

Under normal circumstances tracing works like with `libVT`, but communication during trace file writing is done through TCP sockets, so it may take more time than over MPI. In order to establish communication, it needs to know the IP addresses of all the hosts involved. It finds them by looking up the hostname locally on each machine or, if that only yields the 127.0.0.1 local host IP address, falls back to broadcasting hostnames. In the latter case hostname lookup must work consistently in the cluster. In case of a failure, `libVTfs` freezes all MPI processes and then writes a trace file with all trace data.

Possible Failures

Failure	Description
Signals	Includes events inside the application like segmentation faults and floating point errors, and also abort signals sent from outside, like `SIGINT` or `SIGTERM`. Only `SIGKILL` will abort the application without writing a trace because it cannot be caught.
Premature Exit	One or more processes exit without calling `MPI_Finalize()`.
MPI Errors	MPI detects certain errors itself, like communication problems or invalid parameters for MPI functions.
Deadlocks	If Intel® Trace Collector observes no progress for a certain amount of time in any process, it assumes that a deadlock has occurred, stops the application and writes a trace file. You can configure the timeout with `DEADLOCK-TIMEOUT`. "No progress" is defined as "inside the same MPI call". This is only a heuristic and may fail to lead to both false positives and false negatives.
Undetected Deadlock	If the application polls for a message that cannot arrive with `MPI_Test()` or a similar non-blocking function, Intel® Trace Collector still assumes that progress is made and does not stop the application. To avoid this, use blocking MPI calls in the application, which is also better for performance.
Premature Abort	If all processes remain in MPI for a long time due to a long data transfer for instance, then the timeout might be reached. Because the default timeout is five minutes, this is very unlikely. After writing the trace `libVTfs` will try to clean up the MPI application run by sending all processes in the same process group an `INT` signal. This is necessary because certain versions of MPICH* may have spawned child processes which keep running when an application aborts prematurely, but there is a certain risk that the invoking shell also receives this signal and also terminates. If that happens, then it helps to invoke `mpirun` inside a

	remote shell: `rsh localhost 'sh -c "mpirun . . . "'` MPI errors cannot be ignored by installing an error handler. `libVTfs` overrides all requests to install one and uses its own handler instead. This handler stops the application and writes a trace without trying to proceed, otherwise it would be impossible to guarantee that any trace will be written at all. On Windows* OS, not all features of POSIX* signal handling are available. Therefore, `VTfs` on Windows* OS uses some heuristics and may not work as reliably as on Linux* OS. It is not possible to stop a Windows* application run and get a trace file by sending a signal or terminating the job in the Windows task manager.

Tracing OpenSHMEM Applications

Intel® Trace Collector supports tracing of OpenSHMEM* applications on Linux* OS. If an application includes OpenSHMEM functions, Intel Trace Collector traces them and stores as a separate group in the trace file. This support is enabled in the `libVT` library, while the other tracing libraries listed in Product Components do not have this functionality.

Note

You need to have an MPICH-based MPI library available in `LD_LIBRARY_PATH`.

For OpenSHMEM implementations utilizing MPICH-based MPIs, to trace an OpenSHMEM application, follow the steps described in Tracing Conventional MPI Applications. No additional steps required.

For other OpenSHMEM implementations, trace the application as follows:

```
$ LD_PRELOAD=libVT.so:libmpi.so oshrun -n 4 ./shmem-app
```

Note that we do not provide SHMEM launchers within the ITAC package.

Tracing MPI File IO

On Linux* OS, Intel® Trace Collector does not support tracing of ROMIO*, a portable implementation of MPI-IO. Fully standard-compliant implementations of MPI-IO are untested, but might work.

This distinction is necessary because ROMIO normally uses its own request handles (`MPIO_Request`) for functions like `MPI_File_iread()` and expects the application to call `MPIO_Wait()`/`MPIO_Test()`. These two functions are supported if and only if Intel® Trace Collector is compiled with ROMIO support. In that case the wrapper functions for `MPI_File_iread()` are compiled for `MPIO_Requests` and might not work if the MPI and the application use the normal MPI-2 `MPI_Request`.

Applications which avoid the non-blocking IO calls should work either way.

Handling of Communicator Names

By default, Intel® Trace Collector stores names for well-known communicators in the trace: `COMM_WORLD`, `COMM_SELF_#0`, `COMM_SELF_#1` and so on. When new communicators are created, their names are composed of a prefix, a space and the name of the old communicator. For example, calling `MPI_Comm_dup()` on `MPI_COMM_WORLD` will lead to a communicator called `DUP COMM_WORLD`.
There are the following prefixes for MPI functions:

MPI Function	Prefix
`MPI_Comm_create()`	`CREATE`
`MPI_Comm_dup()`	`DUP`
`MPI_Comm_split()`	`SPLIT`
`MPI_Cart_sub()`	`CART_SUB`
`MPI_Cart_create()`	`CART_CREATE`
`MPI_Graph_create()`	`GRAPH_CREATE`
`MPI_Intercomm_merge()`	`MERGE`

`MPI_Intercomm_merge()` is special because the new communicator is derived from two communicators, not just one as in the other functions. The name of the new inter-communicator will be `MERGE <old name 1>/<old name 2>` if the two existing names are different, otherwise it will be just `MERGE <old name>`.
In addition to these automatically generated names, Intel® Trace Collector also intercepts `MPI_Comm_set_name()` and then uses the name provided by the application. Only the last name set with this function is stored in the trace for each communicator. Derived communicators always use the name that is currently set in the old communicator when the new communicator is created.
Intel® Trace Collector does not attempt to synchronize the names set for the same communicator in different processes, therefore the application has to set the same name in all processes to ensure that this name is really used by Intel® Trace Collector.

Tracing MPI Load Imbalance

Normally, tracing of all MPI events results in a large size of the trace file, even for relatively small applications. To reduce the trace file size, but be able to get an impression of the application bottlenecks, you can trace only the MPI functions that cause application load imbalance. That is, an MPI function is traced only if it was idle at some point of the application run, causing the imbalance. This functionality is implemented in the `libVTim` library.
You can enable source code locations tracing to identify the regions in source code that caused the imbalance (see Recording Source Location Information).

To generate an imbalance trace file, link your application with the `libVTim` library, using the `-trace-imbalance` option of `mpirun`, or one of the methods described here. For example:

```
$ mpirun -n 2 -trace-imbalance ./myApp
```

Open the generated `.stf` file to view the results. Intel® Trace Analyzer displays only the regions of MPI idle time. As a consequence, time values for MPI functions are equal to their idle time.

Known Limitations

- This feature is currently available on Linux* OS only.
- Point-to-point communication patterns displayed by Intel Trace Analyzer may be unreliable, because the `libVTim` library skips tracing of certain functions.
- The library traces only those MPI functions that can potentially generate load imbalance. Therefore, all non-blocking operations are not traced.
- The library does not trace user defined events (see Tracing User Defined Events), OpenMP* regions (see Recording OpenMP* Regions Information), or system calls (see Tracing System Calls).
- Intel Trace Analyzer cannot run idealization for trace files generated by `libVTim`.

Tracing User Defined Events

To get more detailed information about your application, you can instrument and trace various user-defined events in your application, including non-MPI function calls. In practice, it is often useful to record entries and exits to/from functions or code regions within larger functions.

Use the following Intel® Trace Collector capabilities:

- Automatic function instrumentation with the compiler
- Manual source code instrumentation with Intel® Trace Collector API

Automatic Function Instrumentation

Using Intel® Compilers

Intel® compilers can automatically instrument all user functions during compilation. At runtime, Intel® Trace Collector records all function entries and exits in the compilation units.

To enable the instrumentation, use the option `-tcollect` (Linux* OS) or `/Qtcollect` (Windows* OS) during compilation. For example:

```
$ mpiicc -tcollect -trace myapp.c
```

The option accepts an argument to specify the collecting library to link against (see Product Components). For example, for fail-safe tracing, select `libVTfs` as follows: `-tcollect=VTfs` (`VT` by default).
To define a particular set of functions to be instrumented, use the `-tcollect-filter <file>` option. `<file>` contains a list of functions followed by `on|off` switcher:

```
func1 on
func2 off
```

If a function is marked `off`, it is not instrumented.

Using GCC*

Similar function tracing is available in the GNU Compiler Collection (gcc*). Object files that contain functions for tracing are compiled with `-finstrument-functions`, for example:

```
$ mpicc -finstrument-functions -trace myapp.c
```

Intel® Trace Collector obtains output about functions in the executable. By default, it starts the shell program `nm -P` to do this, which can be changed with the `NMCMD` configuration option. See NMCMD.

Folding

Function tracing can generate large amounts of trace data. Use folding to disable tracing of calls within certain functions. It enables you to reduce the trace file size and get information only about events of interest. See Tracing Library Calls for details.

C++ Name Demangling

By default Intel® Trace Collector records function names in their mangled form. The `DEMANGLE` configuration option enables automatic demangling of C++ names. See DEMANGLE.

Manual Source Code Instrumentation

Intel® Trace Collector provides the API that enables you to control the profiling library and trace user-defined functions, define groups of processes, define performance counters and record their values. All API functions, parameters, and macros are declared in the header files `VT.h` and `VT.inc` for C/C++ and Fortran, respectively. Include the appropriate header file in your source code when using the Intel® Trace Collector API functions.
Refer to the Intel® Trace Collector API section for detailed description and usage information on the Intel® Trace Collector API.

To compile an application with calls to the Intel® Trace Collector API, pass the header files to the compiler using the `-I` option. For example: `-I$VT_ROOT/include`.

Using the Dummy Libraries

To temporarily disable tracing for the application with calls to the Intel® Trace Collector API, use the dummy library `libVTnull` available in the libraries folder. This way you do not have to remove the API function calls from the source code to run your application without tracing. For instructions on linking, see Tracing Conventional MPI Applications.

Configuring Intel® Trace Collector

When using Intel® Trace Collector, you may want to customize various aspects of its operation and define filters for data tracing. It is achieved through setting up the appropriate configuration options.

You can set up these options in three ways:

- In a configuration file.
- In the corresponding environment variables.
- In the command line when running your application.

For the list of options and their descriptions, see Configuration Options.

Using Configuration File

Intel® Trace Collector configuration file is a plain ASCII file that contains a number of directives in each line and has the `.conf` extension.
For your convenience, Intel® Trace Analyzer and Collector provides a utility called **Configuration Assistant** intended for creating and editing configuration files. However, you can create the configuration file manually using a text editor. For examples and details on syntax, see Configuration Reference.

To run the Configuration Assistant, enter the command:

```
$ itcconfig <trace_file> [<configuration_file>]
```

Note

Configuration Assistant requires a trace file to be passed, therefore you should first trace your application without any settings to use the utility.

If you do not specify the configuration file, the default settings will be used. Edit the file and save it with the `.conf` extension.
To apply the settings, do the following:

1. Set up the `VT_CONFIG` environment variable to point to the full path to your configuration file. For example:
   ```
   $ export VT_CONFIG=/<configuration_file_directory>/my_settings.conf
   ```
2. Set up the `VT_CONFIG_RANK` environment variable to point to the process that reads and parses the configuration file (the default value is `0`).
3. Trace your application as described in Tracing MPI Applications.

Using Environment Variables

Each option has an equivalent environment variable. To set the variables, use the option names, but prefix them with `VT_` and replace hyphens with underscores. For the `SYMBOL`, `STATE` and `ACTIVITY` options you can also list multiple values in one variable (see Filtering Trace Data for details). For example:

```
$ export VT_STATE=* OFF MPI:* ON
```

Note

Environment variables are checked by the process that reads the configuration file after it has parsed the file, so the variables override the configuration file options.

Using Command-Line Options

To specify configuration options in the command line, at runtime use a string of the following syntax as an argument to your application:

```
--itc-args --<configuration_option> <value> --itc-args-end
```

For example, to generate a trace file of the `SINGLESTF` format:

```
$ mpirun -n 4 ./MyApp --itc-args --logfile-format SINGLESTF --itc-args-end
```

Note

Fortran programs are an exception, because Intel Trace Collector has limited access to command line parameters of Fortran programs.

Protocol File

After tracing your application, a protocol file `.prot` is created. The file lists all configuration options with their values used when your application was traced, and other useful information. You can use the protocol file of a particular run as a configuration file to trace the application with the same settings again. See Protocol File for details.

See Also

Configuration Reference
Filtering Trace Data
Configuration Assistant section in the *Intel® Trace Analyzer User and Reference Guide*

Filtering Trace Data

Filtering in Intel® Trace Collector applies specified filters to the trace collection process. This directly reduces the amount of data collected. The filter rules can be defined in a configuration file, in the environment variables or as command line arguments (see Configuring Intel® Trace Collector for details). Filters are evaluated in the order they are listed, and all matching is case-insensitive. For filtering by collective and point-to-point MPI operations the corresponding `mpirun`/`mpiexec` options are also available.
In the following discussion, items within angle brackets (**<** and **>**) are placeholders for actual values, optional items are put within square brackets (`[` and `]`), and alternatives are separated by a vertical bar `|`.

Filtering by Collective and P2P Operations

Use the following `mpirun`/`mpiexec` options at runtime:

- `-trace-collectives` – to collect information only about collective operations
- `-trace-pt2pt` – to collect information only about point-to-point operations

Note

These options are mutually exclusive, use only one option at a time.

An example command line for tracing collective operations may look as follows:

```
$ mpirun -trace -n 4 -trace-collectives ./myApp
```

These options are special cases of filtering by specific functions discussed below. They use predefined configuration files to collect the appropriate operation types. The options override the `VT_CONFIG` environment variable, so you cannot use your own configuration files when using them. If you need to set additional filters, you can set them through individual environment variables.

Filtering by Specific Functions

Basic Function Filtering

Function filtering is accomplished by defining the `STATE`, `SYMBOL` and `ACTIVITY` options. Each option accepts the pattern matching the filtered function, and the filtering rule. The formal definition is as follows:

```
STATE|SYMBOL|ACTIVITY <PATTERN> <RULE>
```

The general option is `STATE`, while `SYMBOL` and `ACTIVITY` are its replacers:

- `SYMBOL` filters functions by their names, regardless of the class. The following definitions are considered equal:
  ```
  SYMBOL                          <PATTERN>                          <RULE>
  STATE **:<PATTERN> <RULE>
  ```
- `ACTIVITY` filters functions by their class names. The following definitions are considered equal:
  ```
  ACTIVITY                          <PATTERN>                          <RULE>
  STATE <PATTERN>:* <RULE>
  ```
- `STATE` can filter functions both by names and their class names.

Pattern should match the name of the function for filtering. You can use the following wildcards:

Wildcard	Description
*	Any number of characters, excluding ":"
**	Any number of characters, including ":"
?	A single character
[ ]	A list of characters

For example the following definition will filter out all functions containing the word `send`, regardless of the class:

```
STATE **send* OFF
```

The basic filter rule should contain one of the following entries:

```
<RULE> = ON | OFF | <trace level> | <skip level>:<trace level>
```

The <trace level> value defines how far up the call stack is traced. The <skip level> value defines how many levels to skip while going up the call stack. This is useful if a function is called within a library, and the library stack can be ignored. Specifying ON turns on tracing with a <trace level> of 1 and a <skip level> of 0, and OFF turns off tracing completely (this is *not* equivalent to 0:0).

Example

In the example below the following events will be traced: all functions in the class Application, all MPI send functions except MPI_Bsend(), and all receive, test and wait functions. All other MPI functions will not be traced.

```
# disable all MPI functions
ACTIVITY MPI OFF
# enable all send functions in MPI
STATE MPI:*send ON
# except MPI_Bsend
SYMBOL MPI_bsend OFF
# enable receive functions
SYMBOL MPI_recv ON
# and all test functions
SYMBOL MPI_test* ON
# and all wait functions, tracing four call levels
SYMBOL MPI_wait* 4
# enable all functions within the Application class
ACTIVITY Application 0
```

Advanced Function Filtering

For function filtering a finer control is also available. Here is a list of additional filter rule entries, which can be used along with the basic rule in any combination:

```
<ENTRYTRIGGER> | <EXITTRIGGER> | <COUNTERSTATE> | <FOLDING> | <CALLER>
```

Here is the specification for each filter entry available:

Entry/Exit Trigger

```
<ENTRYTRIGGER> = entry <TRIGGER>
<EXITTRIGGER> = exit <TRIGGER>
```

Activate a trigger on entry/exit for the matching pattern.

```
<TRIGGER> = [<TRIPLET>] <ACTION> [<ACTION>]
```

Triggers define a set of actions over a set of processes (triplets, see *Filtering by Ranks* below for definition).

```
<ACTION> = traceon | traceoff | restore | none | begin_scope <SCOPE_NAME> |
end_scope <SCOPE_NAME>
```

The action defines what happens to tracing. Using `traceon` or `traceoff` turns tracing on or off, respectively. `begin_scope` and `end_scope` start or end the named scope.

```
<SCOPE_NAME> = [<class name as string>:]<scope name as string>
```

A scope is a user-defined region in the program. See Defining and Recording Scopes.

Counter State

```
<COUNTERSTATE> = counteron | counteroff
```

Counter state turns on or off sampling for the matching pattern. By default, all enabled counters are sampled at every state change. There is no method for controlling which counters are sampled.

Folding

```
<FOLDING> = fold | unfold
```

Enabling folding for a function disables tracing of any functions called by that function. By default, all functions are unfolded.

Caller

```
<CALLER> = caller <PATTERN>
```

Specifying the caller enables tracing only for functions called by the functions matching the pattern.

For details on use of the `FOLDING` and `CALLER` keywords, see Tracing Library Calls.

Filtering by Ranks

Besides filtering by functions, you can also filter the trace data by ranks in `MPI_COMM_WORLD` using the `PROCESS` configuration option. Its value is a comma separated list of Fortran 90-style triplets. The formal definition is as follows:

```
PROCESS <TRIPLET>[,<TRIPLET>,...] on | off
```

Triplet definition is as follows:

```
<TRIPLET> = <LOWER-BOUND>[:<UPPER-BOUND>[:<INCREMENT>] ]
```

The default value for `<UPPER-BOUND>` is the size of `MPI_COMM_WORLD` (`N`) and the default value for `<INCREMENT>` is `1`.
For example, to trace only even ranks and rank 1 use the following triplets: `0:N:2,1:1:1`, where `N` is the total number of processes. All processes are enabled by default, so you have to disable all of them first (`PROCESS 0:N OFF`) before enabling a certain subset again. For SMP clusters, you can also use the `CLUSTER` option to filter for particular SMP nodes.

Recording OpenMP* Regions Information

Intel® Trace Collector can record information about OpenMP* regions in your application into trace file.

To collect this information, make sure to do the following:

Linux* OS

1. Your application should be:
   - linked with the Intel implementation of OpenMP. See *User and Reference Guide for the Intel® C++ Compiler* for details.
   - dynamically linked with Intel® MPI Library.
2. Use the `-trace` option of `mpirun` to trace the data.

Note

Using the `LD_PRELOAD` environment variable to trace data will not have the desired effect.

Windows* OS

1. Your application should be:
   - linked with the Intel implementation of OpenMP.
   - dynamically linked with the `VT.dll` library and Intel MPI Library.

   The example command line to compile the application may look as follows:

   ```
   > mpiicc -trace -openmp myapp.c
   ```
2. Make sure the `INTEL_LIBITTNOTIFY64` environment variable contains the full path to the `VT.dll` library.
3. Run your application using the `mpiexec` command to trace the data.

See Also

Tracing Conventional MPI Applications

Tracing System Calls (Linux* OS)

On Linux* OS use this capability to track I/O calls.

By default, system call profiling is disabled. To collect system calls, set the following configuration option (see Configuring Intel® Trace Collector for details):

```
ACTIVITY SYSTEM ON
```

To enable collection of an exact function add the following line into a configuration file:

```
STATE SYSTEM:<func_name> ON
```

Note

Intel® Trace Collector does not collect any information on the amount of data saved or read during these operations.

The following functions are supported:

access	clearerr	close	creat
dup	dup2	fclose	fdopen
feof	ferror	fflush	fgetc
fgetpos	fgets	fileno	fopen
fprintf	fputc	fputs	fread
freopen	fseek	fsetpos	ftell
fwrite	getc	getchar	gets
lseek	lseek64	mkfifo	perror
pipe	poll	printf	putc
putchar	puts	read	readv
remove	rename	rewind	setbuf
setvbuf	sprintf	sync	tmpfile
tmpnam	umask	ungetc	vfprintf
vprintf	vsprintf	write	writev

See Also

Configuring Intel® Trace Collector
ACTIVITY
STATE

Collecting Lightweight Statistics

Intel® Trace Collector can gather and store statistics about the function calls and their communication. These statistics are gathered even if no trace data is collected, so it is a good starting point for trying to understand an unknown application that might produce an unmanageable trace.

Usage Instructions

To collect this light-weight statistics for your application, set the following environment variables before tracing:

```
$ export VT_STATISTICS=ON
```

```
$ export VT_PROCESS=OFF
```

Alternatively, set the `VT_CONFIG` environment variable to point to the configuration file:

```
# Enable statistics gathering
STATISTICS ON
# Do not gather trace data
PROCESS 0:N OFF

$ export VT_CONFIG=<configuration_file_path>/config.conf
```

The statistics is written into the `*.stf` file. Use the `stftool` to convert the data to the ASCII text with `--print-statistics`. For example:

```
$ stftool tracefile.stf --print-statistics
```

TIP

The resulting output has easy-to-process format, so you can use text processing programs and scripts such as awk*, perl*, and Microsoft Excel* for better readability. A perl script `convert-stats` with this capability is provided in the `bin` folder.

Output Format

Each line contains the following information:

- Thread or process
- Function ID
- Receiver (if applicable)
- Message size (if applicable)
- Number of involved processes (if applicable)

And the following statistics:

- Count – number of communications or number of calls as applicable
- Minimum execution time excluding callee times
- Maximum execution time excluding callee times
- Total execution time excluding callee times
- Minimum execution time including callee times
- Maximum execution time including callee times
- Total execution time including callee times

Within each line the fields are separated by colons.

Receiver is set to `0xffffffff` for file operations and to `0xfffffffe` for collective operations. If message size equals `0xffffffff` the only defined value is `0xfffffffe` to mark it as a collective operation.

The message size is the number of bytes sent or received per single message. With collective operations the following values (buckets of message size) are used for individual instances:

Value	Process-local bucket	Is the same value on all processes?
`MPI_Barrier`	0	Yes
`MPI_Bcast`	Broadcast bytes	Yes
`MPI_Gather`	Bytes sent	Yes
`MPI_Gatherv`	Bytes sent	No
`MPI_Scatter`	Bytes received	Yes
`MPI_Scatterv`	Bytes received	No
`MPI_Allgather`	Bytes sent + received	Yes
`MPI_Allgatherv`	Bytes sent + received	No
`MPI_Alltoall`	Bytes sent + received	Yes
`MPI_Alltoallv`	Bytes sent + received	No
`MPI_Reduce`	Bytes sent	Yes
`MPI_Allreduce`	Bytes sent + received	Yes
`MPI_Reduce_Scatter`	Bytes sent + received	Yes
`MPI_Scan`	Bytes sent + received	Yes

Message is set to `0xffffffff` if no message was sent, for example, for non-MPI functions or functions like `MPI_Comm_rank`.

If more than one communication event (message or collective operation) occur in the same function call (for example in `MPI_Waitall`, `MPI_Waitany`, `MPI_Testsome`, `MPI_Sendrecv` etc.), the time in that function is evenly distributed over all communications and counted once for each message or collective operation. Therefore, it is impossible to compute a correct traditional function profile from the data referring to such function instances (for example, those that are involved in more than one message per actual function call). Only the **Total execution time including callee times** and the **Total execution time excluding callee times** can be interpreted similar to the traditional function profile in all cases.

The number of involved processes is negative for received messages. If messages were received from a different process/thread it is `-2`.

Statistics are gathered on the thread level for all MPI functions, and for all functions instrumented through the API or compiler instrumentation.

See Also

Tracing User Defined Events
stftool Utility
Intel® Trace Collector API

Recording Source Location Information

Intel® Trace Collector can automatically record locations of function calls in the source code. To record this information, do the following:

1. Compile the relevant application modules with support for debug information by using the `-g` (Linux* OS) and `/Zi` or `/Z7` (Windows* OS) compiler flags. For example:
```
$ mpiicc -g -c ctest.c
```
2. Enable Program Counter (PC) tracing by setting the environment variable `VT_PCTRACE` to `5` for example:
```
$ export VT_PCTRACE=5
```
Alternatively, set the `VT_CONFIG` variable to the configuration file specifying the following, for example:
```
# trace 4 call levels whenever MPI is used
ACTIVITY MPI 4
# trace one call level in all functions not specified
# explicitly; can also be for example, PCTRACE 5
PCTRACE ON
```
```
$ export VT_CONFIG=<config_file>
```
3. Trace your application as described in Tracing MPI Applications.

`PCTRACE` sets the number of call levels for all functions. To avoid performance issues, `PCTRACE` is disabled by default and should be handled carefully. It is useful to get the initial understanding of the application before recording the source location information.
Manual instrumentation of the source code with the Intel® Trace Collector API can provide similar information but without performance overhead. See Defining and Recording Source Locations for details.

Pay attention that the compiler has to use normal stack frames. This is the default in GCC, but may be disabled with `-fomit-frame-pointer`. If the flag is used, then only the direct caller of MPI or API functions can be found, and asking Intel® Trace Collector to unwind more than one stack level may lead to crashes.
The Intel® compilers do not use normal stack frames by default if optimization is enabled, but they can be enabled with with `-fno-omit-frame-pointer`.

See Also

Configuring Intel® Trace Collector
PCTRACE

Recording Hardware Performance Information (Linux* OS)

On Linux* OS Intel® Trace Collector can sample hardware counters with the Performance Application Programming Interface (PAPI). Because PAPI might not be available on a system, support for it is provided as an additional layer on top of the normal Intel® Trace Collector.

This layer is implemented in the `VT_sample.c` source file. It is a sample file that traces counters available through PAPI High level API.
To record hardware counters, do the following:

1. Adjust the `VT_sample.c` sample with the necessary counters
2. Rebuild the `libVTsample.so` file:
   a. Copy the contents of `<isntall-dir>/slib` directory into your working directory.
   b. Edit the provided `Makefile` to match the local setup.
   c. Build the file using the `make` command.
3. Set the `LD_LIBRARY_PATH` environment variable as follows:
   ```
   $ export LD_LIBRARY_PATH=<path_to_libVTsample>:<path_to_PAPI>
   ```
4. Add `libVTsample.so` to the link line in front of the Intel® Trace Collector library. The link line will look as follows:
   ```
   $ mpiicc ctest.c -L$VT_SLIB_DIR -L. -L$PAPI_ROOT -lVTsample -lVT -lpapi $VT_ADD_LIBS -o ctest
   ```

To view the counters in Intel® Trace Analyzer, use **Counter Timeline**.

Recording Operating System Counters

Similar to recording of process specific counters, Intel® Trace Collector can record operating system counters, which provide information about a node. In contrast to the process specific counters, OS counters are sampled very infrequently by one background thread per node and thus the overhead is very low. The amount of trace data also increases insignificantly.

By default, recording of OS counters is disabled. To enable it, set the configuration option:

```
COUNTER <counter_name> ON
```

Supported Counters

Counter Name	Unit	Comment
`disk_io`	KB/s	Read/write disk IO (any disk in the node).
`net_io`	KB/s	Read/write network IO (any system interface). This might not include the MPI transport layer.
`cpu_idle`	percent	Average percentage of CPU time of all CPUs spent in idle mode.

cpu_sys	percent	Average percentage of CPU time of all CPUs spent in system code.
cpu_usr	percent	Average percentage of CPU time of all CPUs spent in user code.

You can change the delay between recording the current counter values with the configuration option OS-COUNTER-DELAY (by default, 1 second). CPU utilization is calculated by the OS with sampling, therefore a smaller value does not necessarily provide more detailed information. Increasing it could reduce the overhead further, but only slightly because the overhead is hardly measurable already.
These OS counters appear in the trace as normal counters which apply to all processes running on a node. To view the counters in Intel® Trace Analyzer, use **Counter Timeline**.

See Also

Configuring Intel® Trace Collector
COUNTER
OS-COUNTER-DELAY

Tracing Library Calls

If you have an application that makes heavy use of libraries or software components developed independently, you may want to exclude the information not related directly to your application from the trace data. At the same time, the library developer might want to do the opposite – trace only data related to their library.

Intel® Trace Collector provides a capability to turn off tracing for functions at a certain call stack level, that is to fold them. If you want to trace calls within the folded functions, you can unfold them.

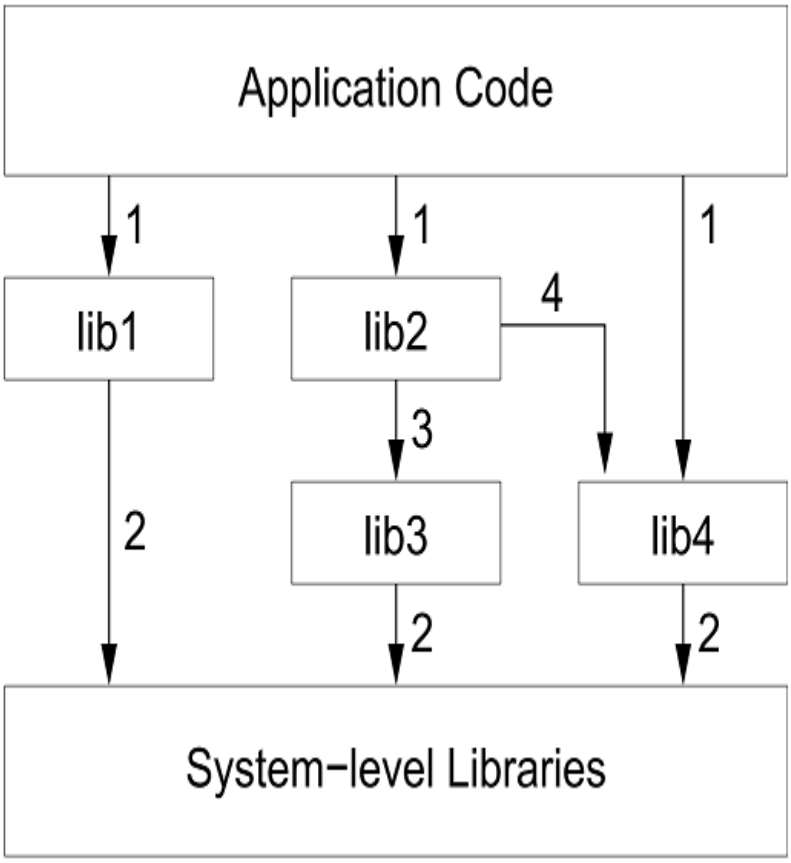
To enable folding, use the FOLD and UNFOLD keywords for the STATE, SYMBOL or ACTIVITY configuration options to select functions for folding by their name (SYMBOL), class (ACTIVITY) or both (STATE). Use the CALLER keyword to specify the function caller. See Filtering Trace Data for details on syntax.

Note

To enable Intel® Trace Collector to profile non-MPI functions, make sure to instrument them using the compiler instrumentation or API. See Tracing User Defined Events.

Below are examples of folding for the application with four additional libraries.

General Structure of an Application Using Multiple Libraries



From the figure above, the following information may be of interest for the application and the library developers:

Application developer

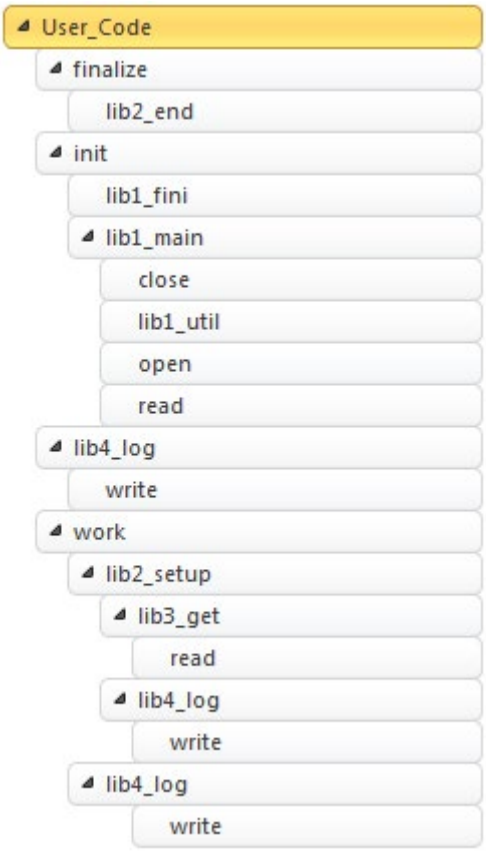
- `lib1`, `lib2`, `lib4` are called by the application. The application developer codes these calls and can change the sequence and parameters to them to improve performance (arrows "1").
- `lib3` is never called directly by the application. The application developer has no way to tailor the use of `lib3`, therefore these calls (arrows "3") are of no interest to him.
- `lib4` is called both directly by the application, and indirectly through `lib2`. Only the direct use of `lib4` can be influenced by the application developer, therefore is of interest to them.

Library developer

The `lib2` developer will need information about the calls from the application, to component libraries (`lib3` and `lib4`), and to system-level services (MPI). They will have no interest in performance data for `lib1`. The `lib1` developer will have no interest in data from `lib2`, `lib3`, and `lib4`.

Examples

In this section folding is illustrated by giving configurations that apply to the example above. The sample `libraries.c` program (available at https://software.intel.com/en-us/product-code-samples) reproduces the same pattern. Its call tree looks as follows (calls are aggregated and sorted by name, therefore the order is not sequential):

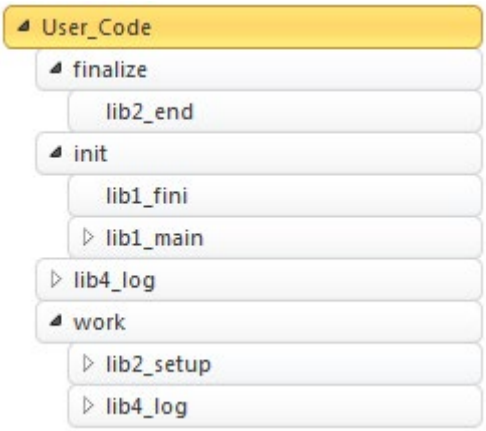


By using the configuration options listed below, different parties can run the same executable to get different traces:

Application developer: Trace the application only with the top-level calls in `lib1`, `lib2`, and `lib4`.
Configuration file: `run_splibraries_app.conf`

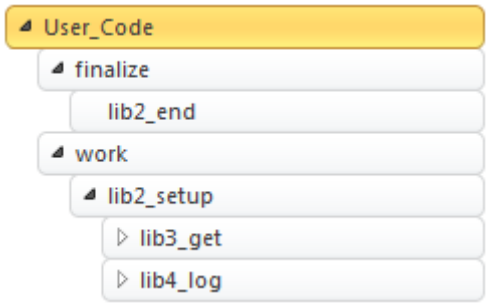
```
STATE lib*:* FOLD
```

Call tree:



lib2 developer: Trace only calls in `lib2`, including its top-level calls
Configuration file: `run_splibraries_lib2.conf`

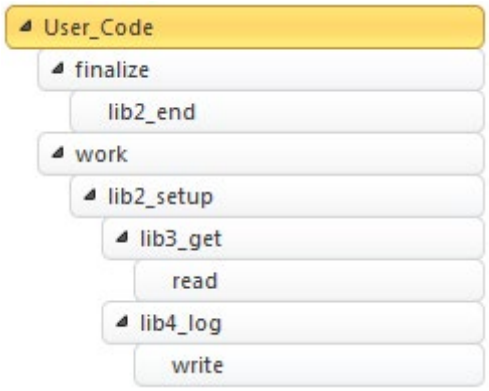
Call tree:



lib2 Developer, detailed view: Trace the top-level calls to `lib2` and all `lib2`, `lib3`, `lib4` and system services invoked by them
Configuration file: `run_splibraries_lib2detail.conf`

```
STATE Application:* FOLD
STATE lib2:* UNFOLD
```

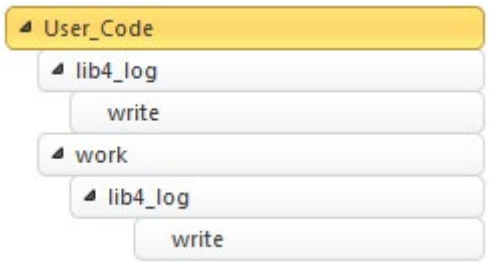
Call tree:



Application and lib4 Developers: Trace the calls in `lib4` only made by the application
Configuration file: `run_splibraries_lib4.conf`

```
STATE *:* FOLD
STATE lib4:* UNFOLD CALLER Application:*
```

Call tree:



It is assumed that the application, library and system calls are instrumented in the way that their classes are different. Alternatively, you can match against the function name prefix that is shared by all library calls in the same library.

Correctness Checking

Correctness Checking

Intel® Trace Collector provides the correctness checking functionality, which addresses the following tasks:

- Finding programming mistakes in the application. They include potential portability problems and violations of the MPI standard, which do not immediately cause problems, but might when switching to different hardware or a different MPI implementation. In this case you are recommended to perform correctness checking interactively on a smaller development cluster, but you can also include it in automated regression testing.
- Detecting errors in the execution environment. In this case use the hardware and software stack on the system that is to be checked.

While doing correctness checking, you should distinguish between error detection that is done automatically by tools, and error analysis that is done by the user to determine the root cause of an error and eventually fix it.

The error detection in Intel® Trace Collector is implemented in the `libVTmc` library, which performs error detection at runtime. To address both of the above scenarios, Intel® Trace Collector supports recording of error reports for later analysis, and interactive debugging at runtime.
The correctness checker prints errors to `stderr` as soon as they are found. You can perform interactive debugging with the help of a traditional debugger: if the application is already running under debugger control, the debugger can stop a process when an error is found. You should manually set a breakpoint in the function `MessageCheckingBreakpoint()`. This function and debug information about it are contained in the Intel® Trace Collector library. Therefore, you can set the breakpoint and inspect the parameters of the function after a process is stopped. The parameters indicate what error occurred.
See the following topics on the usage of correctness checking:

- Correctness Checking of MPI Applications
- Running with Valgrind*
- Configuration
- Analyzing the Results
- Debugger Integration

Correctness Checking of MPI Applications

By default, the `libVTmc` library does not write a tracefile. To perform correctness checking of an MPI application, enable trace collection and link your application with the `libVTmc` library. Do the following:

Linux* OS

1. Switch the `CHECK-TRACING` configuration option to `on` to enable Intel® Trace Collector to record the correctness checking reports to the tracefile. For example:
```
$ export VT_CHECK_TRACING=on
```
2. Run your application with the `-check_mpi` option of `mpirun`. For example:
```
$ mpirun -check_mpi -n 4 ./myApp
```

Windows* OS

1. Relink your application with the `libVTmc` library using the `-check_mpi` compiler option. For example:
```
> mpiicc -check_mpi myApp.c
```
2. Run your application with the `CHECK-TRACING` configuration option enabled. For example:
```
> mpiexec -n 4 myApp.exe --itc-args --check-tracing ON --itc-args-end
```

Use Intel® Trace Analyzer to view correctness checking events.

See Also

CHECK-TRACING

Running with Valgrind* (Linux* OS)

For distributed memory checking (`LOCAL:MEMORY:INITIALIZATION`) and detecting illegal accesses to memory owned by MPI (`LOCAL:MEMORY:ILLEGAL_ACCESS`) it is necessary to run all MPI processes under control of the Valgrind* memory checker (version 3.2.0 or higher). See *http://www.valgrind.org/* for more information.

To run Valgrind, invoke it directly on the main MPI process and add the `mpirun -l` option. This way all output printed by Valgrind is automatically prefixed with the MPI process rank. Intel® Trace Collector detects that `-l` is in effect and then leaves adding the rank prefix to `mpirun` also for Intel® Trace Collector output.

The `LOCAL:MEMORY:ILLEGAL_ACCESS` check causes Valgrind reports not only for illegal application accesses (as desired) but also for Intel® MPI Library own access to the locked memory (not desired, because MPI currently owns it and must read or write it). These reports are normal and the Valgrind suppression file in Intel® Trace Collector lib folder tells Valgrind to not print them, but Valgrind must be notified about it through its `--suppressions` option.

When the MPI executable is given on the command line, an MPI application could be started under Valgrind like this:

```
$ mpirun -check_mpi -l -n <num procs>
$ valgrind --suppressions=$VT_LIB_DIR/impi.supp <application>
...
```

When a wrapper script is used, then it might be possible to trace through the wrapper script by adding the `--trace-children=yes` option, but that could lead to reports about the script interpreter and other programs, so adding Valgrind to the actual invocation of the MPI binary is easier.

Configuration

You can configure manually which errors are checked: all errors have a unique name and are categorized in a hierarchy similar to functions. For example, `LOCAL:MEMORY:OVERLAP` is a local check which ensures that memory is not used twice in concurrent MPI operations. By disabling certain errors you can skip a report about it and reduce the checking overhead.
Use the configuration options listed below. For instructions on how to set them, see Configuring Intel® Trace Collector.

CHECK

Use the `CHECK` configuration option to match against the names of supported errors and turn it on or off, as in the example below. See Correctness Checking Errors for the list of all errors.

```
# Turn all checking off:
# ** matches colons
# * does not
CHECK ** OFF
# Selectively turn on specific checks:
# - All local checks
CHECK LOCAL:** ON
# - Only one global check
CHECK GLOBAL:MSG:DATATYPE:MISMATCH ON
```

PCTRACE

By default, Intel® Trace Collector checks for all errors and tries to provide as much information about them as possible. In particular it does stack unwinding and reports source code information for each level in the call hierarchy. This can be controlled with the `PCTRACE` configuration option. For performance analysis that option is off by default, but for correctness checking with `libVTmc` it is enabled.

DEADLOCK-TIMEOUT

This option controls the same mechanism to detect deadlocks as in `libVTfs`. For interactive use it is recommended to set it to a small value like `10s` to detect deadlocks quickly without having to wait long for the timeout.

DEADLOCK-WARNING

Displays a `GLOBAL:DEADLOCK:NO_PROGRESS` warning if the time spent by MPI processes in their last MPI call exceeds the threshold specified with this option. 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.

VERBOSE

Different levels of verbosity specified with this option have the following effects:

Level	Effect

0	All extra output disabled, only error summary at the end is printed.
1	Adds a summary of configuration options as the application starts (default).
2	Adds a one-line info message at the beginning by each process with host name, process ID and the normal rank prefix. This can be useful if output is redirected into one file per process, because it identifies to which process in the parallel application the output belongs.
3	Adds internal progress messages and a dump of MPI call entry/exit with their parameters and results.

See Also

CHECK
PCTRACE
DEADLOCK-TIMEOUT
DEADLOCK-WARNING
VERBOSE

Analyzing the Results

For interactive debugging, you should start the application so that `stderr` is printed to a console window. Then you can follow which errors are found while the application is running and start analyzing them without having to wait for it to complete. If critical errors are found early, you can abort the run, fix the problem and restart. This ensures a much faster code and test cycle than a post-mortem analysis.
The output for each error varies, depending on the error: only the relevant information is printed, thus avoiding the need to manually skip over irrelevant information. In general, Intel® Trace Collector starts with the error name and then continues with a description of the failure.

For each MPI call involved in the error the MPI parameters are dumped. If PC tracing is enabled (see PCTRACE), Intel® Trace Collector also provides a backtrace of source code locations for each call. For entities like requests, the involved calls include the places where a request was created or activated. This helps to track down errors where the problem is not at the place where it is detected.

Because multiple processes might print errors concurrently, each line is prefixed with a tag that includes the rank of the process in `MPI_COMM_WORLD` which reports the problem. MPI applications which use process spawning or attachment are not supported, therefore that rank is unique.
When the application terminates, Intel® Trace Collector does further error checks (for example, unfree resources, pending messages).

Note

If any process is killed without giving it a chance to clean up (that is, by sending it a `SIGKILL`), this final step is not possible.

Note

Sending a `SIGINT` to `mpiexec` through `kill` or pressing `CTRL-C` will cause Intel® MPI Library to abort all processes with such a hard `SIGKILL`.

Debugger Integration

Debugger Integration

It is necessary to manually set a breakpoint in the function `MessageCheckingBreakpoint()`. Immediately after reporting an error on `stderr` this function is called, so the stack backtrace directly leads to the source code location of the MPI call where the error was detected. In addition to the printed error report, you can also look at the parameters of the `MessageCheckingBreakpoint()` which contain the same information. It is also possible to look at the actual MPI parameters with the debugger because the initial layer of MPI wrappers in `libVTmc` is always compiled with debug information. This can be useful if the application itself lacks debug information or calls MPI with a complex expression or function call as parameter for which the result is not immediately obvious.
The exact methods to set breakpoints depend on the debugger used. Here is some information how it works with specific debuggers. For additional information or other debuggers please refer to the debugger documentation.

The first two debuggers mentioned below can be started by Intel® MPI Library by adding the `-tv` and `-gdb` options to the command line of `mpirun`. Allinea Distributed Debugging Tool* can be reconfigured to attach to MPI jobs that it starts.
Using debuggers like that and Valgrind* are mutually exclusive because the debuggers would try to debug Valgrind, not the actual application. The Valgrind `--db-attach` option does not work out-of-the-box either because each process would try to read from the terminal. One solution that is known to work on some systems for analyzing at least Valgrind reports is to start each process in its own X terminal:

```
$ mpirun -check_mpi -l -n <numprocs> xterm -e bash -c 'valgrind --db-attach=yes
--suppressions=$VT_LIB_DIR/impi.supp <app>; echo press return; read'
```

In that case the Intel® Trace Collector error handling still occurs outside the debugger, so those errors have to be analyzed based on the printed reports.

TotalView* Debugger

For TotalView* Debugger, it is necessary to pay attention that the breakpoint should be set for all processes. There are several ways to automate procedure of setting breakpoints. Mostly it depends on how commonly it is planned to use this automation.

If it is planned to apply it only for the current program, you can create the file `filename.tvd` (file name being the name of the executable) in the working directory in advance and put the following line into it:

```
dfocus gW2 dbreak MessageCheckingBreakpoint
```

Alternatively, you can set the breakpoint in the TotalView* GUI and save breakpoints, which will also create this file and then reuse the settings for further debug sessions with the same executable.

To apply setting this breakpoint for all programs in current working directory, create a file `.tvdrc` with the following lines (or add them if it already exists):

```
proc my_callback {_id} {
   if { $_id == 2 } {
      dfocus p$_id dbreak MessageCheckingBreakpoint
   }
   if { $_id > 2 } {
      dfocus p$_id denable -a
   }
}
dset TV::process_load_callbacks ::my_callback
```

To apply this for all debugging sessions, add these lines to the following file `$HOME/.totalview/tvdrc`.

Run your MPI application as follows:

```
$ mpirun -check_mpi -tv -n <numprocs> <app>
```

See Also

TotalVeiw* Debugger Product Page

GNU* Symbolic Debugger

To automate the procedure of setting breakpoints, GNU* Symbolic Debugger (GDB) supports executing commands automatically. To apply setting this breakpoint for all programs in the current working directory, you can create a file `.gdbinit` with the following lines (or add them if it already exists):

```
set breakpoint pending on
break MessageCheckingBreakpoint
```

Due to the order in which files are processed, placing the same commands in a `.gdbinit` file in the home directory does not work because the main binary is not loaded yet. As a workaround, you can put the following commands into `~/.gdbinit`:

```
define hook-run
# important, output is expected by MPI startup helper
echo Starting program...
# allow pending breakpoint
set breakpoint pending on
# set breakpoint now or as soon as function becomes available
```

```
break MessageCheckingBreakpoint
# restore default behavior
set breakpoint pending auto
end
```

Then start your MPI application as follows:

```
$ mpirun -check_mpi -gdb -n <numprocs> <app>
```

Allinea* Distributed Debugging Tool* (DDT*)

Allinea* Distributed Debugging Tool (DDT) must be configured to run the user application with the necessary Intel libraries preloaded.

Do the following:

1. Go to the **Run** dialog box
2. Select the **Session/Options** menu
3. In the Session/Options menu, choose **Intel MPI Library** and the **Submit through queue or configure own mpirun command option**
4. In the **Submit Command** box enter without line breaks:

   ```
   $ mpirun -genv LD_PRELOAD libVTmc.so -genv VT_DEADLOCK_TIMEOUT 20s -
   genv                    VT_DEADLOCK_WARNING                    25s
   -n NUM_PROCS_TAG DDTPATH_TAG/bin/ddt-debugger
   ```

5. You can leave other boxes empty. Click **OK**.
6. To start the application, press the submit button on DDT's job launch dialog box.
7. When the application is ready, select the **Control/Add Breakpoint** menu and add a breakpoint at the `MessageCheckingBreakpoint` function.
8. Continue to run and debug your application as normal, the program will stop automatically at `MessageCheckingBreakpoint` when an MPI error is detected.

You can use the parallel stack browser to find the processes that are stopped and select any of these processes. The local variables in this function will identify the error type, the number of errors so far, and the error message.

You can also set a condition on this breakpoint from the Breakpoints tab, or **Add Breakpoint** menu, for example, to stop only after 20 errors are reported use a condition of `reportnumber > 20`.

See Also

Allinea* DDT* Product Page

Tracing Distributed Non-MPI Applications

Processes in non-MPI applications or systems are created and communicate using non-standard and varying methods. The communication may be slow or unsuitable for Intel® Trace Collector

communication patterns. Therefore a special version of the Intel® Trace Collector library `libVTcs` was developed that neither relies on MPI nor on the application's communication, but rather implements its own communication layer using TCP/IP. This is why it is called client-server.
The `libVTcs` library allows the generation of executables that work without MPI. Linking is accomplished by adding `libVTcs.a` (`VTcs.lib` on Microsoft* Windows* OS) and the libraries it needs to the link line: `-lVTcs $VT_ADD_LIBS`. The application has to call `VT_initialize()` and `VT_finalize()` to generate a tracefile. Function tracing can be used with and without further Intel® Trace Collector API calls to actually generate trace events.
This section describes the design, implementation and usage of Intel® Trace Collector for distributed applications.

Design

The application has to meet the following requirements:

- The application handles startup and termination of all processes itself. Both startup with a fixed number of processes and dynamic spawning of processes is supported, but spawning processes is an expensive operation and should not be done too frequently.
- For a reliable startup, the application has to gather a short string from every process in one place to bootstrap the TCP/IP communication in Intel® Trace Collector. Alternatively, one process is started first and its string is passed to the others. In this case you can assume that the string is always the same for each program run, but this is less reliable because the string encodes a dynamically chosen port which may change.
- Map the hostname to an IP address that all processes can connect to.

Note

This is not the case if `/etc/hosts` lists the hostname as alias for 127.0.0.1 and processes are started on different hosts. As a workaround for that case the hostname is sent to other processes, which then requires a working name lookup on their host systems.

Intel® Trace Collector for distributed applications consists of a special library (`libVTcs`) that is linked into the application's processes and the `VTserver` executable, which connects to all processes and coordinates the trace file writing. Linking with `libVTcs` is required to keep the overhead of logging events as small as possible, while `VTserver` can be run easily in a different process.
Alternatively, the functionality of the `VTserver` can be accomplished with another API call by one of the processes.

Using VTserver

This is how the application starts, collects trace data and terminates:

1. The application initializes itself and its communication.
2. The application initializes communication between VTserver and processes.
3. Trace data is collected locally by each process.
4. VT data collection is finalized, which moves the data from the processes to the VTserver, where it is written into a file.

5. The application terminates.

The application may iterate several times over points 2 till 4. Looping over 3 and the trace data collection part of 4 are not supported at the moment, because:

- it requires a more complex communication between the application and VTserver
- the startup time for 2 is expected to be sufficiently small
- reusing the existing communication would only work well if the selection of active processes does not change

If the startup time turns out to be unacceptably high, then the protocol between application and Intel® Trace Collector could be revised to support reusing the established communication channels.

Initialize and Finalize

The application has to bootstrap the communication between the VTserver and its clients. This is done as follows:

1. The application server initiates its processes.
2. Each process calls `VT_clientinit()`.
3. `VT_clientinit()` allocates a port for TCP/IP communication with the VTserver or other clients and generates a string which identifies the machine and this port.
4. Each process gets its own string as result of `VT_clientinit()`.
5. The application collects these strings in one place and calls VTserver with all strings as soon as all clients are ready. VT configuration is given to the VTserver as file or through command line options.
6. Each process calls `VT_initialize()` to actually establish communication.
7. The VTserver establishes communication with the processes, then waits for them to finalize the trace data collection.
8. Trace data collection is finalized when all processes have called `VT_finalize()`.
9. Once the VTserver has written the trace file, it quits with a return code indicating success or failure.

Some of the VT API calls may block, especially `VT_initialize()`. Execute them in a separate thread if the process wants to continue. These pending calls can be aborted with `VT_abort()`, for example if another process failed to initialize trace data collection. This failure has to be communicated by the application itself and it also has to terminate the VTserver by sending it a kill signal, because it cannot be guaranteed that all processes and the VTserver will detect all failures that might prevent establishing the communication.

Running without VTserver

Instead of starting VTserver as rank 0 with the contact strings of all application processes, one application process can take over that role. It becomes rank 0 and calls `VT_serverinit()` with the information normally given to VTserver. This changes the application startup only slightly.
A more fundamental change is supported by first starting one process with rank 0 as server, then taking its contact string and passing it to the other processes. These processes then give this string as the initial value of the contact parameter in `VT_clientinit()`. To distinguish this kind of startup from the dynamic spawning of process described in the next section, the prefix **S** needs to be added

by the application before calling `VT_clientinit()`. An example where this kind of startup is useful is a process which preforks several child processes to do some work.
In both cases it may be useful to note that the command line arguments previously passed to VTserver can be given in the `argc/argv` array as described in the documentation of `VT_initialize()`.

Spawning Processes

Spawning new processes is expensive, because it involves setting up TCP communication, clock synchronization, configuration broadcasting, amongst others. Its flexibility is also restricted because it needs to map the new processes into the model of communicators that provide the context for all communication events. This model follows the one used in MPI and implies that only processes inside the same communicator can communicate at all.

For spawned processes, the following model is currently supported: one of the existing processes starts one or more new processes. These processes need to know the contact string of the spawning process and call `VT_clientinit()` with that information; in contrast to the startup model from the previous section, no prefix is used. Then while all spawned processes are inside `VT_clientinit()`, the spawning process calls `VT_attach()` which does all the work required to connect with the new processes.
The results of this operation are:

- a new `VT_COMM_WORLD` which contains all of the spawned processes, but not the spawning process
- a communicator which contains the spawning process and the spawned ones; the spawning process gets it as result from `VT_attach()` and the spawned processes by calling `VT_get_parent()`

The first of these communicators can be used to log communication among the spawned processes, the second for communication with their parent. There's currently no way to log communication with other processes, even if the parent has a communicator that includes them.

Tracing Events

Once a process' call to `VT_initialize()` has completed successfully it can start calling VT API functions that log events. These events will be associated with a time stamp generated by Intel® Trace Collector and with the thread that calls the function.
Should the need arise, then VT API functions could be provided that allow one thread to log events from several different sources instead of just itself.

Event types supported at the moment are those also provided in the normal Intel® Trace Collector, like state changes (`VT_enter()`, `VT_leave()`) and sending and receiving of data (`VT_log_sendmsg()`, `VT_log_recvmsg()`). The resulting trace file is in a format that can be loaded and analyzed with Intel® Trace Analyzer.

Usage

Executables in the application are linked with `-lVTcs` and `$VT_ADD_LIBS`. It is possible to have processes implemented in different languages, as long as they use the same version of the `libVTcs`.
The VTserver has the following synopsis:

```
VTserver <contact infos> [config options]
```

Each contact info is guaranteed to be one word and their order on the command line is irrelevant. The configuration options can be specified on the command line by adding the prefix `--` and listing its arguments after the keyword. This is an example for contacting two processes and writing into the file `example.stf` in STF format:

```
VTserver <contact1> <contact2> --logfile-name example.stf
```

All options can be given as environment variables. The format of the configuration file and the environment variables are described in more detail in the chapter about `VT_CONFIG`.

Signals

`libVTcs` uses the same techniques as fail-safe MPI tracing to handle failures inside the application, therefore it will generate a trace even if the application segfaults or is aborted with Ctrl + C.
When only one process runs into a problem, then `libVTcs` tries to notify the other processes, which then should stop their normal work and enter trace file writing mode. If this fails and the application hangs, then it might still be possible to generate a trace by sending a `SIGINT` to all processes manually.

Examples

There are two examples using MPI as means of communication and process handling. But as they are not linked against the normal Intel® Trace Collector library, tracing of MPI has to be done with Intel Trace Collector API calls.

`clientserver.c` is a full-blown example that simulates and handles various error conditions. It uses threads and fork/exec to run API functions and VTserver concurrently. `simplecs.c` is a stripped down version that is easier to read, but does not check for errors.
The dynamic spawning of processes is demonstrated by `forkcs.c`. It first initializes one process as server with no clients, then forks to create new processes and connects to them with `VT_attach()`. This is repeated recursively. Communication is done through pipes and logged in the new communicators.
`forkcs2.c` is a variation of the previous example which also uses fork and pipes, but creates the additional processes at the beginning without relying on dynamic spawning.
The examples are available at: https://software.intel.com/en-us/product-code-samples

See Also

Intel® Trace Collector API

Intel® Trace Collector Reference

This section contains the reference information for Intel® Trace Collector. See the brief overview of each sub-section in the table below.

Section	Description
API Reference	Detailed information and usage instructions on the Intel® Trace Collector API functions. Related User Guide topic: Tracing User Defined Events
Configuration Reference	Information on the configuration file syntax, protocol file and description of all configuration options supported. Related User Guide topics: Configuring Intel® Trace Collector, Filtering Trace Data
Correctness Checking Errors	The list of supported correctness checking errors and explanation of their detection method. Related User Guide topic: Correctness Checking
Structured Tracefile Format	Information on the structured tracefile format (STF) used by Intel® Trace Collector to store the trace data. The section also describes the single-STF format, lists all STF components and provides information on its configuration.
stftool Utility	Information on manipulating STF trace files using the `stftool` and `xstftool` utilities.
Time Stamping	Description of the time stamps Intel® Trace Collector assigns to events during tracing.
Secure Loading of DLLs on Windows* OS	Information on the security options for the loading of DLLs on Windows* OS.

API Reference

API Reference

The Intel® Trace Collector library provides the user with a number of functions that control the profiling library and record user-defined activities, define groups of processes, define performance

counters and record their values. Header files with the necessary parameter, macro and function declarations are provided in the `include` directory: `VT.h` for ANSI C and C++ and `VT.inc` for Fortran 77 and Fortran 90. It is strongly recommended to include these header files if any Intel® Trace Collector API functions are to be called.
You can also find the description of all available API functions in comments for `VT.h` and in the man pages on Linux* OS (`man VT`).
The Intel® Trace Collector library is thread-safe in the sense that all of its API functions can be called by several threads at the same time. Some API functions can really be executed concurrently, others protect global data with POSIX* mutexes.

Concepts

This section uses the following concepts, essential for understanding the Intel® Trace Collector API:

- **Symbol** – function referred to by its name without the class name. For example: `MPI_Send`.
- **Activity** – set of functions referred to by their class name. For example: `MPI`.
- **State** – function referred to by its full name including the class name (combination of activity and symbol). For example: `MPI:MPI_Send`.
- **State change** – event of entering or leaving a function (state).

For detailed description and examples of these concepts within the framework of the configuration functionality, see Filtering Trace Data.

General Macros and Errors

#define VT_VERSION

API version constant. It is incremented each time the API changes, even if the change does not break compatibility with the existing API. It can be used to determine at compile time how to call the API, like this:

```
#if VT_VERSION > 4000
   do something
#else
   do something different
#endif
```

`VT_version()` provides the same information at runtime.
To check whether the current revision of the API is still compatible with the revision of the API that the application was written against, compare against both `VT_VERSION` and `VT_VERSION_COMPATIBILITY`, as shown below.

#define VT_VERSION_COMPATIBILITY

Oldest API definition, which is still compatible with the current one.

It is set to the current version each time an API change can break programs written for the previous API. For example, a program written for `VT_VERSION 2090` will work with API 3000 if `VT_VERSION_COMPATIBILITY` remained at `2090`. It may even work without modifications when `VT_VERSION_COMPATIBILITY` was increased to `3000`, but this cannot be determined automatically and will require a source code review.
Here is a usage example:

```
#define APP_VT_VERSION 1000 // API version used by APP
#ifdef VT_VERSION_COMPATIBILITY > APP_EXPECTED_VT_VERSION
# error "VT.h is no longer compatible with APP's usage of the API"
#endif
#ifdef VT_VERSION < APP_EXPECTED_VT_VERSION
# error "VT.h is not recent enough for APP"
#endif
```

Suppose you instrumented your C source code for the API with `VT_VERSION` equal to `3100`. Then you could add the following code fragment to detect incompatible changes in the API:

```
#include <VT.h>
#if VT_VERSION_COMPATIBILITY > 3100
# error ITC API is no longer compatible with our calls
#endif
```

Make sure to compare against a fixed number but not `VT_VERSION`, because `VT_VERSION` will always be greater or equal `VT_VERSION_COMPATIBILITY`.

To make the instrumentation work again after such a change, you can either just update the instrumentation to accommodate for the change or even provide different instrumentation that is chosen by the C preprocessor based on the value of `VT_VERSION`.

enum _VT_ErrorCode

Error codes returned by Intel® Trace Collector API.

Enumerator	Description
`VT_OK`	OK
`VT_ERR_NOLICENSE`	No valid license found
`VT_ERR_NOTIMPLEMENTED`	Not implemented
`VT_ERR_NOTINITIALIZED`	Not initialized
`VT_ERR_BADREQUEST`	Invalid request type
`VT_ERR_BADSYMBOLID`	Wrong symbol ID
`VT_ERR_BADSCLID`	Wrong SCL ID
`VT_ERR_BADSCL`	Wrong SCL
`VT_ERR_BADFORMAT`	Wrong format
`VT_ERR_BADKIND`	Wrong kind found

VT_ERR_NOMEMORY	Could not get memory
VT_ERR_BADFILE	Error while handling file
VT_ERR_FLUSH	Error while flushing
VT_ERR_BADARG	Wrong argument
VT_ERR_NOTHREADS	No worker threads
VT_ERR_BADINDEX	Wrong thread index
VT_ERR_COMM	Communication error
VT_ERR_INVT	Intel® Trace Collector API called while inside an Intel® Trace Collector function
VT_ERR_IGNORE	Non-fatal error code

Initialization, Termination and Control

Initialization, Termination and Control

Intel® Trace Collector is automatically initialized within the execution of the `MPI_Init()` function. During the execution of the `MPI_Finalize()` function, the trace data collected in memory or in temporary files is consolidated and written into the permanent trace file(s), and Intel® Trace Collector is terminated. Thus, it is an error to call Intel® Trace Collector API functions before `MPI_Init()` has been executed or after `MPI_Finalize()` has returned.
In non-MPI applications it may be necessary to start and stop Intel® Trace Collector explicitly. These calls also help write programs and libraries that use Intel® Trace Collector without depending on MPI.

`VT_initialize()`, `VT_getrank()`, `VT_finalize()` can be used to write applications or libraries which work both with and without MPI, depending on whether they are linked with `libVT.a` plus MPI or with `libVTcs.a` (distributed tracing) and no MPI.
If the MPI that Intel® Trace Collector was compiled for provides `MPI_Init_thread()`, then `VT_init()` will call `MPI_Init_thread()` with the parameter required set to `MPI_THREAD_FUNNELED`. This is sufficient to initialize multithreaded applications where only the main thread calls MPI. If your application requires a higher thread level, then either use `MPI_Init_thread()` instead of `VT_init()` or (if `VT_init()` is called for example, by your runtime environment) set the environment variable `VT_THREAD_LEVEL` to a value of 0 till 3 to choose thread levels `MPI_THREAD_SINGLE` till `MPI_THREAD_MULTIPLE`.
It is not an error to call `VT_initialize()` twice or after a `MPI_Init()`.
In an MPI application written in C, the program parameters must be passed, because the underlying MPI might require them. Otherwise they are optional, and 0 or a NULL pointer may be used. If parameters are passed, then the number of parameters and the array itself may be modified, either by MPI or Intel® Trace Collector itself.

Intel® Trace Collector assumes that `argv[0]` is the executable name and uses this string to find the executable and as the basename for the default logfile name. Other parameters are ignored unless there are special `--itc-args` parameters.
See the description of the following functions:

- VT_initialize
- VT_finalize
- VT_getrank
- VT_getdescription
- VT_setfinalizecallback
- VT_countsetcallback

The following functions control the tracing of threads in a multithreaded application:

- VT_registerthread
- VT_registernamed
- VT_registerprefixed
- VT_getthrank

The recording of performance data can be controlled on a per-process basis by calls to the `VT_traceon()` and `VT_traceoff()` functions: a thread calling `VT_traceoff()` will no longer record any state changes, MPI communication or counter events. Tracing can be re-enabled by calling the `VT_traceon()` function. The collection of statistics data is not affected by calls to these functions. With the API function `VT_tracestate()` a process can query whether events are currently being recorded.
See the description of functions:

- VT_traceon
- VT_traceoff
- VT_tracestate

With the Intel® Trace Collector configuration mechanisms described in Filtering Trace Data, the recording of state changes can be controlled per symbol or activity. For any defined symbol, the `VT_symstate()` function returns whether data recording for that symbol has been disabled.
Find the function description in the following section:

- VT_symstate

Intel® Trace Collector minimizes the instrumentation overhead by first storing the recorded trace data locally in the memory of each processor and saving it to disk only when the memory buffers are filled up. Calling the `VT_flush()` function forces a process to save the in-memory trace data to disk, and mark the duration of this in the trace. After returning, Intel® Trace Collector continues to work normally.

- VT_flush

Intel® Trace Collector makes its internal clock available to applications, which can be useful to write instrumentation code that works with MPI and non-MPI applications.

For more detailed information, refer to the following sections:

- VT_timestamp

- VT_timestart

VT_initialize

```
int VT_initialize (int * argc, char *** argv)
```

Description

Initializes the Intel® Trace Collector and underlying communication.

Fortran

```
VTINIT(ierr)
```

Parameters

`argc`	a pointer to the number of command line arguments
`argv`	a pointer to the program's command line arguments

Return values

Returns error code

VT_finalize

```
int VT_finalize(void)
```

Description

Finalizes Intel® Trace Collector and underlying communication.

It is not an error to call `VT_finalize()` twice or after a `MPI_Finalize()`.

Fortran

```
VTFINI(ierr)
```

Return values

Returns error code

VT_getrank

```
int VT_getrank(int * rank)
```

Description

Gets process index (same as MPI rank within `MPI_COMM_WORLD`).

Note
This number is not unique in applications with dynamic process spawning.

Fortran

```
VTGETRANK(rank, ierr)
```

Return values

`rank`	stores process index

Returns error code

VT_registerthread

```
int VT_registerthread(int thindex)
```

Description

Registers a new thread with Intel® Trace Collector under the given number.

Threads are numbered starting from 0, which is always the thread that has called `VT_initialize()` or `MPI_Init()`. The call to `VT_registerthread()` is optional, as the thread that uses Intel® Trace Collector without having called `VT_registerthread()` is automatically assigned the lowest free index. If a thread terminates, then its index becomes available again and might be reused for another thread.
Calling `VT_registerthread()` when the thread has been assigned an index already is an error, unless the argument of `VT_registerthread()` is equal to this index. The thread is not (re-)registered in case of an error.

Fortran

```
VTREGISTERTHREAD(thindex, ierr)
```

Parameters

`thindex`	thread number, only used if >= 0

Return values

Returns error codes:

- `VT_ERR_BADINDEX` - thread index is currently assigned to another thread
- `VT_ERR_BADARG` - thread has been assigned a different index already
- `VT_ERR_NOTINITIALIZED` - Intel® Trace Collector has not been initialized yet

VT_registernamed

```
int VT_registernamed (const char * threadname, int thindex)
```

Description

Registers a new thread with Intel® Trace Collector under the given number and name.

Threads with the same number cannot have different names. If you try doing that, the thread uses the number, but not the new name.

Registering a thread twice with different names or numbers is an error. You can add a name to an already registered thread with `VT_registernamed("new name", -1)` if no name has been set before.

Parameters

`threadname`	desired name of the thread, or NULL/empty string if no name wanted
`thindex`	desired thread number, pass negative number to let Intel® Trace Collector pick a number

Return values

Returns error code, see VT_registerthread

VT_registerprefixed

```
int VT_registerprefixed (const char * threadname, int thindex)
```

Description

This functions is identical to VT_registernamed, with the only difference that it appends the process number as a prefix for the thread name. For example, for `threadname = "ThreadA"` and process number 127, the resulting thread name displayed in Intel® Trace Analyzer will be `"P127 ThreadA"`.

Parameters

`threadname`	desired name of the thread, or NULL/empty string if no name wanted
`thindex`	desired thread number, pass negative number to let Intel® Trace Collector pick a number

Return values

Returns error code, see VT_registerthread

VT_getthrank

```
int VT_getthrank (int * thrank)
```

Description

Gets thread index within a process.

Can be assigned either automatically by Intel® Trace Collector, or manually with `VT_registerthread()`.

Fortran

```
VTGETTHRANK(thrank, ierr)
```

Return values

`thrank`	thread index within current thread is stored here

Returns error code

VT_traceon

```
void VT_traceon (void)
```

Description

Turns on tracing for the thread if it was disabled, otherwise does nothing.

Cannot enable tracing if `PROCESS/CLUSTER NO` was applied to the process in the configuration.

Fortran

```
VTTRACEON( )
```

VT_traceoff

```
void VT_traceoff (void)
```

Description

Turns off tracing for the thread if it was enabled, does nothing otherwise.

Fortran

```
VTTRACEOFF( )
```

VT_tracestate

```
int VT_tracestate (int * state)
```

Description

Gets logging state of current thread.

Set by configuration options `PROCESS/CLUSTER`, modified by `VT_traceon/off()`.
There are three states:

- 0 = thread is logging
- 1 = thread is currently not logging
- 2 = logging has been turned off completely

Note
Different threads within one process may be in state 0 and 1 at the same time because `VT_traceon/off()` sets the state of the calling thread, but not for the whole process.

State 2 is set through the configuration option `PROCESS/CLUSTER NO` for the whole process and cannot be changed.

Fortran

```
VTTRACESTATE( state, ierr )
```

Return values

`state`	is set to current state

Returns error code

VT_symstate

```
int VT_symstate (int statehandle, int * on)
```

Description

Gets filter state of one state.

Set by configuration options `SYMBOL`, `ACTIVITY`.

Note
A state may be active even if the thread logging state is `off`.

Fortran

```
VTSYMSTATE( statehandle, on, ierr )
```

Parameters

`statehandle`	result of `VT_funcdef()` or `VT_symdef()`

Return values

`on`	set to 1 if symbol is active

Returns error code

VT_flush

```
int VT_flush (void)
```

Description

Flushes all trace records from memory into the flush file.

The location of the flush file is controlled by options in the configuration file. Flushing will be recorded in the trace file as entering and leaving the state `VT_API:TRACE_FLUSH` with time stamps that indicate the duration of the flushing. Automatic flushing is recorded as `VT_API:AUTO_FLUSH`. Refer to Configuration Options to learn about the `MEM-BLOCKSIZE` and `MEM-MAXBLOCKS` configuration options that control Intel® Trace Collector memory usage.

Fortran

```
VTFLUSH(ierr)
```

Return values

Returns error code

VT_timestamp

```
double VT_timestamp (void)
```

Description

In contrast to previous versions, this time stamp no longer represents seconds. Use `VT_timeofday()` for that instead. The result of `VT_timestamp()` can be copied verbatim and given to other API calls, but nothing else.

Fortran

```
DOUBLE PRECISION VTSTAMP()
```

Return values

Returns an opaque time stamp, or `VT_ERR_NOTINITIALIZED`.

VT_timestart

```
double VT_timestart (void)
```

Description

Writes instrumentation code that works with MPI and non-MPI applications

Fortran

```
DOUBLE PRECISION VTTIMESTART()
```

Return values

Returns point in time in seconds when process started, or `VT_ERR_NOTINITIALIZED`.

VT_setfinalizecallback

```
int VT_setfinalizecallback( VT_Callback_t callback )
```

Description

Sets a callback which is called by the Intel® Trace Collector at the beginning of finalization. This function may use the Intel Trace Collector API to log events.

Only one callback can be stored per process, setting another or `NULL` removes the previous callback.

Parameters

`callback`	a pointer to the callback

Return values

Returns error code

VT_getdescription

```
const char *VT_getdescription( int type )
```

Description

Returns a pointer that describes certain aspects of the library that implements the Intel® Trace Collector API. This call can be used by code that is compatible with any library implementing the API, but nevertheless wants to identify the implementation.

Parameters

`type`	to specify what kind of information is requested (`VT_DESCRIPTION_LIB`, ...)

Return values

Returns error code

VT_countsetcallback

```
int VT_countsetcallback( VT_CountCallback_t callback, void *custom, int ncounters )
```

Description

Sets a callback for counter sampling for the calling thread.

The data provided by the callback is logged with the same time stamp as the event that triggered the callback. The callback must be set for each thread individually. Setting `NULL` disables sampling for the thread.

Parameters

`callback`	address of the callback function or `NULL`
`custom`	opaque data that is passed to the callback function by the Intel® Trace Collector; can be used by the callback function to identify the thread or enabled counters

`ncounters`	upper limit for the number of counters returned by the callback. It is not a problem to specify a number that is larger than the one that will be actually used, because the extra memory provided to the callback function will be reused efficiently.

Return values

Returns error code

Defining and Recording Source Locations

Source locations can be specified and recorded in two different contexts:

- State changes, associating a source location with the state change. This is useful to record where a function has been called, or where a code region begins and ends.
- Communication events, associating a source location with calls to MPI functions, for example, calls to the send/receive or collective communication and I/O functions.

To minimize instrumentation overhead, locations for the state changes and communication events are referred to by integer location handles that can be defined by calling the API function `VT_scldef()`, which will automatically assign a handle. A source location is a pair of a filename and a line number within that file.

VT_scldef

```
int VT_scldef (const char * file, int line_nr, int * sclhandle)
```

Description

Allocates a handle for a source code location (SCL).

Fortran

```
VTSCLDEF(file, line_nr, sclhandle, ierr)
```

Parameters

`file`	file name
`line_nr`	line number in this file, counting from 1

Return values

`sclhandle`	the integer it points to is set by Intel® Trace Collector

Returns error code

Some functions require a location handle, but they all accept `VT_NOSCL` instead of a real handle:

#define VT_NOSCL

Special SCL handle — no location available.

VT_sclstack

```
int VT_sclstack (void * pc, void * stackframe, int skip, int trace, int *
sclhandle)
```

Description

Allocates a handle for a source code location (SCL) handle which refers to the current call stack.

This SCL can then be used in several API calls without having to repeat the stack unwinding each time. Which stack frames are preserved and which are skipped is determined by the `PCTRACE` configuration option, but can be overridden with function parameters.
Special support is available for recording source code locations from inside signal handlers by calling this function with the `pc` and `stackframe` parameters different from `NULL`. Other usages of these special parameters include:

- Remembering the stack frame in those API calls of a library that are invoked directly by the application, then at arbitrary points in the library do stack unwinding based on that stack frame to catch just the application code
- Defining a source code location ID for a specific program counter value

Here is a usage example of this call inside a library that implements a message send:

```
void MySend(struct *msg) {
   int sclhandle;
   VT_sclstack( NULL, NULL, // we use the default stack unwinding
         1,       // MySend() is called directly by the
                  // application code we want to trace:
                  // skip our own source code, but not
                  // more
         -1,      // default PCTRACE setting for size
                  // of recorded stack
         &sclhandle );
   // if an error occurs, we continue with the sclhandle == VT_NOSCL
   // that VT_sclstack() sets
   VT_enter( funchandle,
         sclhandle );
   VT_log_sendmsg( msg->receiver,
                  msg->count,
                  msg->tag,
                  msg->commid,
                  sclhandle );
   // do the send here
```

```
   VT_leave( sclhandle );
}
```

Parameters

`pc`	record the source code of this program counter value as the innermost call location, then continue with normal stack unwinding; `NULL` if only stack unwinding is to be used
`stackframe`	start unwinding at this stack frame, `NULL` for starting with the stack frame of `VT_sclstack()` itself: on Intel® 64 architecture the stack frame is found in the RBP register
`skip`	-1: get the number of stack frames to skip from the `PCTRACE` configuration option 0: first recorded program counter value after the (optional) `pc` address is the return address of the initial stack frame >0: skip the given number of return addresses
`trace`	-1: get the number of stack frames to record from the `PCTRACE` configuration option 0: do not record any source code locations for the call stack: returns an SCL ID for the `pc` address if one is given, otherwise returns `VT_NOSCL` immediately >0: the number of stack frames to record

Return values

`sclhandle`	points to the integer set by Intel® Trace Collector to a valid SCL handle in case of success and `VT_NOSCL` otherwise

Returns error code

Intel® Trace Collector automatically records all available information about MPI calls. On some systems, the source location of these calls is automatically recorded. On the other systems, the source location of MPI calls can be recorded by calling the `VT_thisloc()` function immediately before the call to the MPI function, with no intervening MPI or Intel® Trace Collector API calls.

VT_thisloc

```
int VT_thisloc (int sclhandle)
```

Description

Sets source code location for next activity that is logged by Intel® Trace Collector.

After being logged it is reset to the default behavior again: automatic PC tracing if enabled in the configuration file, and supported or no SCL otherwise.

Fortran

```
VTTHISL(sclhandle, ierr)
```

Parameters

`sclhandle`	handle defined either with `VT_scldef()`

Return values

Returns error code

Defining and Recording Functions or Regions

Defining and Recording Functions or Regions

Intel® Trace Analyzer can display and analyze general (properly nested) state changes, relating to function calls, entry/exit to/from code regions and other events occurring in a process. Intel® Trace Analyzer implements a two-level model of states: a state is referred to by an activity name that identifies a group of states, and the state (or symbol) name that references a particular state in that group. For instance, all MPI functions are part of the activity MPI, and each one is identified by its function name, for instance `MPI_Send` for C and for Fortran.
The Intel® Trace Collector API allows the user to define arbitrary activities and symbols and to record entry and exit to/from them. In order to reduce the instrumentation overhead, symbols are referred to by integer handles that can be managed automatically (using the `VT_funcdef()` interface) or assigned by the user (using the old `VT_symdef()` function). All activities and symbols are defined by each process that uses them, but it is not necessary to define them consistently on all processes (see `UNIFY-SYMBOLS`).
Optionally, information about source locations can be recorded for state enter and exit events by passing a non-null location handle to the `VT_enter()`/`VT_leave()` or `VT_beginl()`/`VT_endl()` functions.

New Interface

To simplify the use of user-defined states, a new interface has been introduced for Intel® Trace Collector. It manages the symbol handles automatically, freeing the user from the task of assigning and keeping track of symbol handles, and has a reduced number of arguments. Furthermore, the performance of the new functions has been optimized, reducing the overhead of recording state changes.

To define a new symbol, first create the respective activity by calling the `VT_classdef()` function. A handle for that activity is returned, and the symbol can be defined with it by calling `VT_funcdef()`. The returned symbol handle is passed, for example, to `VT_enter()` to record a state entry event.

VT_classdef

```
int VT_classdef (const char * classname, int * classhandle)
```

Description
Allocates a handle for a class name.

The `classname` may consist of several components separated by a colon **:**. Leading and trailing colons are ignored. Several colons in a row are treated as just one separator.

Fortran
```
VTCLASSDEF(classname, classhandle, ierr)
```

Parameters

`classname`	name of the class

Return values

`classhandle`	the integer it points to is set by Intel® Trace Collector

Returns error code

VT_funcdef

```
int VT_funcdef (const char * symname, int classhandle, int * statehandle)
```

Description
Allocates a handle for a state.

The `symname` may consist of several components separated by a colon **:**. If that's the case, then these become the parent class(es). Leading and trailing colons are ignored. Several colons in a row are treated as just one separator.
This is a replacement for `VT_symdef()` which doesn't require the application to provide a unique numeric handle.

Fortran
```
VTFUNCDEF(symname, classhandle, statehandle, ierr)
```

Parameters

`symname`	name of the symbol
`classhandle`	handle for the class this symbol belongs to, created with `VT_classdef()`, or `VT_NOCLASS`, which is an alias for "Application" if the `symname` does not contain a class name and ignored otherwise

Return values

`statehandle`	the integer it points to is set by Intel® Trace Collector

Returns error code

#define VT_NOCLASS

Special value for `VT_funcdef()` – put function into the default class Application.

Old Interface

To define a new symbol, first determine which value has to be used for the symbol handle, and then call the `VT_symdef()` function, passing the symbol and activity names, plus the handle value. It is not necessary to define the activity itself. Make sure to not use the same handle value for different symbols.

VT_symdef

```
int VT_symdef (int statehandle, const char * symname, const char * activity)
```

Description
Defines the numeric `statehandle` as shortcut for a state.
This function will become obsolete and should not be used for new code. Both `symname` and `activity` may consist of more than one component, separated by a colon **:**.
Leading and trailing colons are ignored. Several colons in a row are treated as just one separator.

Fortran
```
VTSYMDEF(code, symname, activity, ierr)
```

Parameters

`statehandle`	numeric value chosen by the application
`symname`	name of the symbol
`activity`	name of activity this symbol belongs to

Return values
Returns error code

State Changes

The following functions take a state handle defined with either the new or old interface. Handles defined with the old interface incur a higher overhead in these functions, because they need to be mapped to the real internal handles. Therefore it is better to use the new interface.

Intel® Trace Collector distinguishes between code regions (marked with `VT_begin()`/`VT_end()`) and functions (marked with `VT_enter()`/`VT_leave()`). The difference is only relevant when passing source code locations.

VT_begin

```
int VT_begin (int statehandle)
```

Description
Marks the beginning of a region with the name that was assigned to the symbol.

Regions should be used to subdivide a function into different parts or to mark the location where a function is called.

If automatic tracing of source code locations (PC tracing) is supported, then Intel® Trace Collector will log the location where `VT_begin()` is called as source code location for this region and the location where `VT_end()` is called as SCL for the next part of the calling symbol (which may be a function or another larger region).
If an SCL has been set with `VT_thisloc()`, then this SCL will be used even if PC tracing is supported.
The functions `VT_enter()` and `VT_leave()` have been added that can be used to mark the beginning and end of a function call within the function itself. The difference is that a manual source code location which is given to `VT_leave()` cannot specify where the function call took place, but rather where the function is left.
If PC tracing is enabled, then the `VT_leave` function stores the SCL where the instrumented function was called as SCL for the next part of the calling symbol. In other words, it skips the location where the function is left, which would be recorded if `VT_end()` were used instead.
`VT_begin()` adds an entry to a stack which can be removed only with `VT_end()`.

Fortran
```
VTBEGIN(statehandle, ierr)
```

Parameters

`statehandle`	handle defined either with `VT_symdef()` or `VT_funcdef()`

Return values
Returns error code

VT_beginl

```
int VT_beginl (int statehandle, int sclhandle)
```

Description
Shortcut for `VT_thisloc(sclhandle);VT_begin(statehandle)`.

Fortran
```
VTBEGINL(statehandle, sclhandle, ierr)
```

VT_end

```
int VT_end (int statehandle)
```

Description
Marks the end of a region.

Has to match a `VT_begin()`. The parameter was used to check this, but this is no longer done to simplify instrumentation; now it is safe to pass a 0 instead of the original state handle.

Fortran
```
VTEND(statehandle, ierr)
```

Parameters

`statehandle`	obsolete, pass anything you want

Return values
Returns error code

VT_endl

```
int VT_endl (int statehandle, int sclhandle)
```

Description
Shortcut for `VT_thisloc(sclhandle);VT_end(statehandle)`.

Fortran
```
VTENDL(statehandle, sclhandle, ierr)
```

VT_enter

```
int VT_enter (int statehandle, int sclhandle)
```

Description
Mark the beginning of a function.

Usage similar to `VT_beginl()`. See also `VT_begin()`.

Fortran
```
VTENTER(statehandle, sclhandle, ierr)
```

Parameters

`statehandle`	handle defined either with `VT_symdef()` or `VT_funcdef()`
`sclhandle`	handle, defined by `VT_scldef`. Use `VT_NOSCL` if you don't have a specific value.

Return values
Returns error code

VT_leave

```
int VT_leave (int sclhandle)
```

Description
Mark the end of a function.

See also `VT_begin()`.

Fortran
```
VTLEAVE(sclhandle, ierr)
```

Parameters

sclhandle	handle, defined by VT_scldef. Currently ignored, but is meant to specify the location of exactly where the function was left in the future. Use VT_NOSCL if you don't have a specific value.

Return values
Returns error code

VT_enterstate

```
int VT_enterstate (const char * name, int * statehandle, int * truncated)
```

Description
Defines a state (when invoked the first time) and enters it.

It relies on the caller to provide persistent storage for state handles.

The corresponding call to leave the state again is the normal VT_leave(). VT_leave() must be called if and only if VT_enterstate() returns a zero return code.

```
static int bsend_handle, bsend_truncated;
int bsend_retval;
bsend_retval = VT_enterstate( "MPI:TRANSFER:BSEND", &bsend_handle,
&bsend_truncated );
...
if( !bsend_retval) VT_leave( VT_NOSCL );
```

As demonstrated in this example, one or more colons **:** may be used to specify parent classes of the state, just as in VT_funcdef() and others.
But in contrast to those, VT_enterstate() also treats a slash **/** as special and uses it to log states at a varying degree of detail: depending on the value of DETAILED-STATES (0 = OFF, 1 = ON, 2, 3...), only the part of the name before the first slash is used (DETAILED-STATES 0). For higher values of DETAILED-STATES more components of the name are used and the slashes in the part of the name which is used is treated like the class separator (:).

Examples

1. "MPI:TRANSFER/SEND/COPY" + DETAILED-STATES 0: "MPI:TRANSFER"
2. "MPI:TRANSFER/SEND/COPY" + DETAILED-STATES 1: "MPI:TRANSFER:SEND"
3. "MPI:TRANSFER/SEND/COPY" + DETAILED-STATES >= 2: "MPI:TRANSFER:SEND:COPY"
4. "/MPI:INTERNAL" + DETAILED-STATES 0: "" = not logged
5. "/MPI:INTERNAL" + DETAILED-STATES 1: ":MPI:INTERNAL" = "MPI:INTERNAL"

If (and only if) the configuration option DETAILED-STATES causes the truncation of a certain state name, then entering that state is ignored if the process already is in that state.
Example of trace with DETAILED-STATES 0:

1. enter "MPI:TRANSFER/WAIT" = enter "MPI:TRANSFER"
2. enter "MPI:TRANSFER/COPY" = "MPI:TRANSFER" = ignored by Intel® Trace Collector, return code != 0
3. leave "MPI:TRANSFER/COPY" = ignored by application

4. enter "MPI:TRANSFER/WAIT" = recursive call; ignored too
5. leave "MPI:TRANSFER/WAIT" = ignored by application
6. leave "MPI:TRANSFER/WAIT" = leave "MPI:TRANSFER"

The same trace with DETAILED-STATES 1:

1. enter "MPI:TRANSFER/WAIT" = enter "MPI:TRANSFER:WAIT"
2. enter "MPI:TRANSFER/COPY" = enter "MPI:TRANSFER:COPY"
3. leave "MPI:TRANSFER/COPY" = leave "MPI:TRANSFER:COPY"
4. enter "MPI:TRANSFER/WAIT" = enter "MPI:TRANSFER:WAIT"
5. leave "MPI:TRANSFER/WAIT" = leave "MPI:TRANSFER:WAIT"
6. leave "MPI:TRANSFER/WAIT" = leave "MPI:TRANSFER:WAIT"

Fortran

```
VTENTERSTATE(name, statehandle, truncated, ierr)
```

Parameters

name	the name of the state, with colons and/or slashes as separators as described above

Return values

statehandle	must be initialized to zero before calling this function for the first time, then is set inside the function to the state handle which corresponds to the function which is logged
truncated	set when calling the function for the first time: zero if the full name is logged

Returns zero if state was entered and VT_leave() needs to be called

VT_wakeup

```
int VT_wakeup (void)
```

Description

Triggers the same additional actions as logging a function call, but without actually logging a call.

When Intel® Trace Collector logs a function entry or exit it might also execute other actions, like sampling and logging counter data. If a function runs for a very long time, then Intel® Trace Collector has no chance to execute these actions. To avoid that, the programmer can insert calls to this function into the source code of the long-running function.

Fortran

```
VTWAKEUP(ierr)
```

Return values

Returns error code

Defining and Recording Scopes

Scope is a user-defined region in the source code. In contrast to regions and functions, which are entered and left with `VT_begin`/`VT_end()` or `VT_enter`/`VT_leave()`, scope does not follow the stack based approach. It is possible to start scope **a**, then start scope **b** and stop **a** before **b**, that is they can overlap one another:

```
|---- a -----|
   |------ b ----|
```

VT_scopedef

```
int VT_scopedef (const char * scopename, int classhandle, int scl1, int scl2, int * scopehandle)
```

Description

Define a new scope. A scope is identified by its name and class, like functions. The source code locations that can be associated with it are additional and optional attributes; they can be used to mark a static start and end of the scope in the source.

Like functions, `scopename` may consist of several components separated by a colon `:`.

Fortran

```
VTSCOPEDEF(scopename, classhandle, scl1, scl2, scopehandle, ierr)
```

Parameters

`scopename`	the name of the scope
`classhandle`	the class this scope belongs to (defined with `VT_classdef()`)
`scl1`	any kind of SCL as defined with `VT_scldef()`, or `VT_NOSCL`
`scl2`	any kind of SCL as defined with `VT_scldef()`, or `VT_NOSCL`

Return values

`scopehandle`	set to a numeric handle for the scope, needed by `VT_scopebegin()`

Returns error code

VT_scopebegin

```
int VT_scopebegin (int scopehandle, int scl, int * seqnr)
```

Description

Starts a new instance of the scope previously defined with `VT_scopedef()`.
There can be more than one instance of a scope at the same time. In order to have the flexibility to stop an arbitrary instance, Intel® Trace Collector assigns an intermediate identifier to it which can (but does not have to) be passed to `VT_scopeend()`. If the application does not need this flexibility, then it can simply pass 0 to `VT_scopeend()`.

Fortran

```
VTSCOPEBEGIN(scopehandle, scl, seqnr, ierr)
```

Parameters

`scopehandle`	the scope as defined by `VT_scopedef()`
`scl`	in contrast to the static SCL given in the scope definition this you can vary with each instance; pass `VT_NOSCL` if not needed

Return values

`seqnr`	is set to a number that together with the handle identifies the scope instance; pointer may be `NULL`

Returns error code

VT_scopeend

```
int VT_scopeend (int scopehandle, int seqnr, int scl)
```

Description

Stops a scope that was previously started with `VT_scopebegin()`.

Fortran

```
VTSCOPEEND(scopehandle, seqnr, scl)
```

Parameters

`scopehandle`	identifies the scope that is to be terminated
`seqnr`	0 terminates the most recent scope with the given handle, passing the `seqnr` returned from `VT_scopebegin()` terminates exactly that instance
`scl`	a dynamic SCL for leaving the scope

Defining Groups of Processes

Intel® Trace Collector enables you to define an arbitrary, recursive group structure over the processes of an MPI application, and Intel® Trace Analyzer can display profiling and communication statistics for these groups. Thus, you can start with the top-level groups and walk down the hierarchy, unfolding interesting groups into ever more detail until you arrive at the level of processes or threads.

Groups are defined recursively with a simple bottom-up scheme: the `VT_groupdef()` function builds a new group from a list of already defined groups of processes, returning an integer group handle to identify the newly defined group. The following handles are predefined:

enum VT_Group

Enumerator	Description
`VT_ME`	The calling thread or process
`VT_GROUP_THREAD`	Group of all threads
`VT_GROUP_PROCESS`	Group of all processes
`VT_GROUP_CLUSTER`	Group of all clusters

To refer to non-local processes, the lookup function `VT_getprocid()` translates between ranks in `MPI_COMM_WORLD` and handles that can be used for `VT_groupdef()`.

VT_getprocid

```
int VT_getprocid(int procindex, int * procid)
```

Description

Get global ID for process which is identified by process index.

If threads are supported, then this ID refers to the group of all threads within the process, otherwise the result is identical to `VT_getthreadid(procindex, 0, procid)`.

Fortran

```
VTGETPROCID(procindex, procid, ierr)
```

Parameters

`procindex`	index of process (0 <= `procindex` < N)

Return values

`procidpointer`	pointer to the memory location where the ID is written

Returns error code

The same works for threads.

VT_getthreadid

```
int VT_getthreadid(int procindex, int thindex, int _ threadid)
```

Description

Get global id for the thread which is identified by the pair of process and thread index.

Fortran

```
VTGETTHREADID(procindex, thindex, threadid, ierr)
```

Parameters

`procindex`	index of process (0 <= `procindex` < N)
`thindex`	index of thread

Return values

`threadid`	pointer to the memory location where the ID is written

Returns error code

VT_groupdef

```
int VT_groupdef(const char * name, int n_members, int * ids, int * grouphandle)
```

Description

Defines a new group and returns a handle for it.

Groups are distinguished by their name and their members. The order of group members is preserved, which can lead to groups with the same name and same set of members, but different order of these members.

Fortran

```
VTGROUPDEF(name, n_members, ids[], grouphandle, ierr)
```

Parameters

`name`	the name of the group
`n_members`	number of entries in the ids array
`ids`	array where each entry can be either a `VT_Group` value, or result of `VT_getthreadid()`, `VT_getprocid()` or `VT_groupdef()`

Return values

`grouphandle`	handle for the new group, or old handle if the group has already been defined

Returns error code

To generate a new group that includes the processes with even ranks in `MPI_COMM_WORLD`, you can use the code:

```
int *IDS = malloc(sizeof(*IDS)*(number_procs/2));
int i, even_group;
for( i = 0; i < number_procs; i += 2 )
  VT_getprocid(i, IDS + i/2);
VT_groupdef("Even Group", number_procs/2, IDS, &even_group);
```

If threads are used, then they automatically become part of a group that is formed by all threads inside the same process. The numbering of threads inside this group depends on the order in which threads call the Intel® Trace Collector library because they are registered the first time they invoke the Intel Trace Collector library. The order can be controlled by calling `VT_registerthread()` as the first API function with a positive parameter.

Defining and Recording Counters

Intel® Trace Collector introduces the concept of counters to model numeric performance data that changes over the execution time. Use counters to capture the values of hardware performance counters, or of program variables (iteration counts, convergence rate, etc.) or any other numerical quantity. An Intel® Trace Collector counter is identified by its name, the counter class it belongs to (similar to the two-level symbol naming), and the type of its values (integer or floating-point) and the units that the values are quoted in (Example: MFlop/sec).

A counter can be attached to MPI processes to record process-local data, or to arbitrary groups. When using a group, then each member of the group will have its own instance of the counter and when a process logs a value it will only update the counter value of the instance the process belongs to.

Similar to other Intel® Trace Collector objects, counters are referred to by integer counter handles that are managed automatically by the library.

To define a counter, the class it belongs to needs to be defined by calling `VT_classdef()`. Then, call `VT_countdef()`, and pass the following information:

- Counter name
- Data type

enum VT_CountData

Enumerator	Description
`VT_COUNT_INTEGER`	Counter measures 64 bit integer value, passed to Intel® Trace Collector API as a pair of high and low 32 bit integers
`VT_COUNT_FLOAT`	Counter measures 64 bit floating point value (native format)
`VT_COUNT_INTEGER64`	Counter measures 64 bit integer value (native format)
`VT_COUNT_DATA`	Mask to extract the data format

- Kind of data

enum VT_CountDisplay

Enumerator	Description
VT_COUNT_ABSVAL	Counter are displayed with absolute values
VT_COUNT_RATE	First derivative of counter values is displayed
VT_COUNT_DISPLAY	Mask to extract the display type

- Semantic associated with a sample value

enum VT_CountScope

Enumerator	Description
VT_COUNT_VALID_BEFORE	The value is valid until and at the current time
VT_COUNT_VALID_POINT	The value is valid exactly at the current time, and no value is available before or after it
VT_COUNT_VALID_AFTER	The value is valid at and after the current time
VT_COUNT_VALID_SAMPLE	The value is valid at the current time and samples a curve, so for example, linear interpolation between sample values is possible
VT_COUNT_SCOPE	Mask to extract the scope

- Counter target, that is the process or group of processes it belongs to (VT_GROUP_THREAD for a thread-local counter, VT_GROUP_PROCESS for a process-local counter, or an arbitrary previously defined group handle)
- Lower and upper bounds
- Counter unit (an arbitrary string like FLOP, Mbytes)

VT_countdef

```
int VT_countdef (const char * name, int classhandle, int genre, int target,
const void * bounds, const char * unit, int * counterhandle)
```

Description

Define a counter and get handle for it.

Counters are identified by their name (string) alone.

Fortran

```
VTCOUNTDEF(name, classhandle, genre, target, bounds[], unit, counterhandle,
ierr)
```

Parameters

`name`	string identifying the counter
`classhandle`	class to group counters, handle must have been retrieved by `VT_classdef`
`genre`	bitwise or of one value from `VT_CountScope`, `VT_CountDisplay` and `VT_CountData`
`target`	target which the counter refers to (`VT_ME`, `VT_GROUP_THREAD`, `VT_GROUP_PROCESS`, `VT_GROUP_CLUSTER` or thread/process-id or user-defined group handle).
`bounds`	array of lower and upper bounds (2x 64 bit float, 2x2 32 bit integer, 2x 64 bit integer ->16 byte)
`unit`	string identifying the unit for the counter (like Volt, pints etc.)

Return values

`counterhandle`	handle identifying the defined counter

Returns error code

The integer counters have 64-bit integer values, while the floating-point counters have a value domain of 64-bit IEEE floating point numbers. On systems that have no 64-bit integer type in C, and for Fortran, the 64-bit values are specified using two 32-bit integers. Integers and floats are passed in the native byte order, but for `VT_COUNT_INTEGER` the integer with the higher 32 bits needs to be given first on all platforms:

Counter	Value
VT_COUNT_INTEGER	32 bit integer (high) 32 bit integer (low)
VT_COUNT_INTEGER64	64 bit integer
VT_COUNT_FLOAT	64 bit float

At any time during execution, a process can record a new value for any of the defined counters by calling one of the Intel® Trace Collector API routines described below. To minimize the overhead, it is possible to set the values of several counters with one call by passing an integer array of counter handles and a corresponding array of values. In C, it is possible to mix 64-bit integers and 64-bit floating point values in one value array; in Fortran, the language requires that the value array contains either all integer or all floating point values.

VT_countval

```
int VT_countval(int ncounters, int * handles, void * values)
```

Description

Record counter values.

Values are expected as two 4-byte integers, one 8-byte integer or one 8-byte double, according to the counter it refers to.

Fortran

```
VTCOUNTVAL(ncounters, handles[], values[], ierr)
```

Parameters

`ncounters`	number of counters to be recorded
`handles`	array of ncounters many handles (previously defined by `VT_countdef`)
`values`	array of `ncounters` many values, `value[i]` corresponds to `handles[i]`.

Return Values

Returns error code

The online samples resource contains `counterscopec.c`, which demonstrates all of these facilities.

Recording Communication Events

These are API calls that allow logging of message send and receive and MPI-style collective operations. Because they are modeled after MPI operations, they use the same kind of communicator to define the context for the operation.

enum _VT_CommIDs

Logging send/receive events evaluates the rank local within the active communicator, and matches events only if they take place in the same communicator (in other words, it is the same behavior as in MPI).

Defining new communicators is not supported, but the predefined ones can be used.

Enumerator	Description
`VT_COMM_INVALID`	Invalid ID, do not pass to Intel® Trace Collector
`VT_COMM_WORLD`	Global ranks are the same as local ones

VT_COMM_SELF	Communicator that only contains the active process

VT_log_sendmsg

```
int VT_log_sendmsg(int other_rank, int count, int tag, int commid, int sclhandle)
```

Description

Logs sending of a message.

Fortran

```
VTLOGSENDMSG(other_rank, count, tag, commid, sclhandle, ierr)
```

Parameters

my_rank	rank of the sending process
other_rank	rank of the target process
count	number of bytes sent
tag	tag of the message
commid	numeric ID for the communicator (VT_COMM_WORLD, VT_COMM_SELF, or see VT_commdef())
sclhandle	handle as defined by VT_scldef, or VT_NOSCL

Return values

Returns error code

VT_log_recvmsg

```
int VT_log_recvmsg(int other_rank, int count, int tag, int commid, int sclhandle)
```

Description

Logs receiving of a message.

Fortran

```
VTLOGRECVMSG(other_rank, count, tag, commid, sclhandle, ierr)
```

Parameters

my_rank	rank of the receiving process
other_rank	rank of the source process

`count`	number of bytes sent
`tag`	tag of the message
`commid`	numeric ID for the communicator (`VT_COMM_WORLD`, `VT_COMM_SELF`, or see `VT_commdef()`)
`sclhandle`	handle as defined by `VT_scldef`, or `VT_NOSCL`

Return values

Returns error code

The next three calls require a little extra care, because they generate events that not only have a time stamp, but also a duration. This means that you need to take a time stamp first, then do the operation and finally log the event.

VT_log_msgevent

```
int VT_log_msgevent(int sender, int receiver, int count, int tag, int commid,
double sendts, int sendscl, int recvscl)
```

Description

Logs sending and receiving of a message.

Fortran

```
VTLOGMSGEVENT(sender, receiver, count, tag, commid, sendts, sendscl, recvscl,
ierr)
```

Parameters

`sender`	rank of the sending process
`receiver`	rank of the target process
`count`	number of bytes sent
`tag`	tag of the message
`commid`	numeric ID for the communicator (`VT_COMM_WORLD`, `VT_COMM_SELF`, or see `VT_commdef()`)
`sendts`	time stamp obtained with `VT_timestamp()`
`sendscl`	handle as defined by `VT_scldef()` for the source code location where the message was sent, or `VT_NOSCL`
`recvscl`	the same for the receive location

Return values

Returns error code

VT_log_op

```
int VT_log_op(int opid, int commid, int root, int bsend, int brecv, double startts, int sclhandle)
```

Description

Logs the duration and amount of transferred data of an operation for one process.

Fortran

```
VTLOGOP(opid, commid, root, bsend, brecv, startts, sclhandle, ierr)
```

Parameters

`opid`	id of the operation; must be one of the predefined constants in `enum _VT_OpTypes`
`commid`	numeric ID for the communicator; see `VT_log_sendmsg()` for valid numbers
`root`	rank of the root process in the communicator (ignored for operations without root, must still be valid, though)
`bsend`	bytes sent by process (ignored for operations that send no data)
`brecv`	bytes received by process (ignored for operations that receive no data)
`startts`	the start time of the operation (as returned by `VT_timestamp()`)
`sclhandle`	handle as defined by `VT_scldef`, or `VT_NOSCL`

Return values

Returns error code

VT_log_opevent

```
int VT_log_opevent(int opid, int commid, int root, int numprocs, int _ bsend, int _ brecv, double _ startts, int sclhandle)
```

Description

Logs the duration and amount of transferred data of an operation for all involved processes at once.

Intel® Trace Collector knows which processes send and receive data in each operation. Unused byte counts are ignored when writing the trace, so they can be left uninitialized, but NULL is not allowed as array address even if no entry is used at all.

Fortran

```
VTLOGOPEVENT(opid, commid, root, numprocs, bsend, brecv, startts, sclhandle, ierr)
```

Parameters

`opid`	id of the operation; must be one of the predefined constants in `enum _VT_OpTypes`

commid	numeric ID for the communicator; see VT_log_sendmsg() for valid numbers
root	rank of the root process in the communicator (ignored for operations without root, must still be valid, though)
numprocs	the number of processes in the communicator
bsend	bytes sent by process
brecv	bytes received by process
startts	the start time of the operation (as returned by VT_timestamp())
sclhandle	handle as defined by VT_scldef, or VT_NOSCL

Return values

Returns error code

enum _VT_OpTypes

These are operation IDs that can be passed to VT_log_op().
Their representation in the trace file matches that of the equivalent MPI operation.

User-defined operations are not supported.

Enumerator	Description
VT_OP_INVALID	Undefined operation, should not be passed to Intel® Trace Collector
VT_OP_COUNT	Number of predefined operations
VT_OP_BARRIER VT_OP_IBARRIER VT_OP_BCAST VT_OP_IBCAST VT_OP_GATHER VT_OP_IGATHER VT_OP_GATHERV VT_OP_IGATHERV VT_OP_SCATTER VT_OP_ISCATTER VT_OP_SCATTERV VT_OP_ISCATTERV VT_OP_ALLGATHER VT_OP_IALLGATHER VT_OP_ALLGATHERV VT_OP_IALLGATHERV VT_OP_ALLTOALL VT_OP_IALLTOALL VT_OP_ALLTOALLV VT_OP_IALLTOALLV VT_OP_ALLTOALLW	MPI operation representations

VT_OP_IALLTOALLW VT_OP_NEIGHBOR_ALLGATHER VT_OP_INEIGHBOR_ALLGATHER VT_OP_NEIGHBOR_ALLGATHERV VT_OP_INEIGHBOR_ALLGATHERV VT_OP_NEIGHBOR_ALLTOALL VT_OP_INEIGHBOR_ALLTOALL VT_OP_NEIGHBOR_ALLTOALLV VT_OP_INEIGHBOR_ALLTOALLV VT_OP_NEIGHBOR_ALLTOALLW VT_OP_INEIGHBOR_ALLTOALLW VT_OP_REDUCE VT_OP_REDUCE_LOCAL VT_OP_IREDUCE VT_OP_ALLREDUCE VT_OP_IALLREDUCE VT_OP_REDUCE_SCATTER VT_OP_REDUCE_SCATTER_BLOCK VT_OP_IREDUCE_SCATTER VT_OP_IREDUCE_SCATTER_BLOCK VT_OP_SCAN VT_OP_ISCAN VT_OP_EXSCAN VT_OP_IEXSCAN	

Having a duration also may introduce the problem of having overlapping operations, which has to be taken care of with the following two calls.

VT_begin_unordered

```
int VT_begin_unordered(void)
```

Description

Starts a period with out-of-order events.

Most API functions log events with just one time stamp which is taken when the event is logged. That guarantees strict chronological order of the events.

`VT_log_msgevent()` and `VT_log_opevent()` are logged when the event has finished with a start time taken earlier with `VT_timestamp()`. This can break the chronological order, for example, like in the following two examples:

```
t1: VT_timestamp() "start message"
t2: VT_end() "leave function"
t3: VT_log_msgevent( t1 ) "finish message"

t1: VT_timestamp() "start first message"
t2: VT_timestamp() "start second message"
t3: VT_log_msgevent( t1 ) "finish first message"
t4: VT_log_msgevent( t2 ) "finish second message"
```

In other words, it is okay to just log a complex event if and only if no other event is logged between its start and end in this thread. "logged" in this context includes other complex events that are logged later, but with a start time between the other events start and end time.

In all other cases you have to alert Intel® Trace Collector of the fact that out-of-order events will follow by calling `VT_begin_unordered()` before and `VT_end_unordered()` after these events. When writing the events into the trace file Intel® Trace Collector increases a counter per thread when it sees a `VT_begin_unordered()` and decrease it at a `VT_end_unordered()`. Events are remembered and sorted until the counter reaches zero, or till the end of the data.
This means that:

- unordered periods can be nested
- it is not necessary to close each unordered period at the end of the trace
- but not closing them properly in the middle of a trace will force Intel® Trace Collector to use a lot more memory when writing the trace (proportional to the number of events till the end of the trace).

Fortran

```
VTBEGINUNORDERED(ierr)
```

VT_end_unordered

```
int VT_end_unordered (void)
```

Description

Close a period with out-of-order events that was started with `VT_begin_unordered()`.

Fortran

```
VTENDNORDERED(ierr)
```

Additional API Calls in libVTcs

VT_abort

```
int VT_abort (void)
```

Description

Abort a `VT_initialize()` or `VT_finalize()` call running concurrently in a different thread.
This call will not block, but it might still take a while before the aborted calls actually return. They will return either successfully (if they have completed without aborting) or with an error code.

Return values

`0` if abort request was sent successfully, error code otherwise

VT_clientinit

```
int VT_clientinit(int procid, const char * clientname, const char * contact)
```

Description

Initializes communication in a client/server application.

Must be called before `VT_initialize()` in the client of the application. There are three possibilities:

- client is initialized first, which produces a contact string that must be passed to the server (`*contact == NULL`)
- the server was started first, its contact string is passed to the clients (`*contact == <result of VT_serverinit() with the prefix "S" - this prefix must be added by the application>`)
- a process spawns children dynamically, its contact string is given to its children (`*contact == <result of VT_serverinit() or VT_clientinit()>`)

Parameters

`procid`	All clients must be enumerated by the application. This will become the process id of the local client inside its `VT_COMM_WORLD`. If the VTserver is used, then enumeration must start at 1 because VTserver always gets rank 0. Threads can be enumerated automatically by Intel® Trace Collector or by the client by calling `VT_registerthread()`.
`clientname`	The name of the client. Currently only used for error messages. Copied by Intel® Trace Collector.

Return values

`contact`	Will be set to a string which tells other processes how to contact this process. Guaranteed not to contain spaces. The client may copy this string, but doesn't have to, because Intel® Trace Collector will not free this string until `VT_finalize()` is called.

Returns error code

VT_serverinit

```
int VT_serverinit(const char * servername, int numcontacts, const char * contacts[], const char ** contact)
```

Description

Initializes one process as the server that contacts the other processes and coordinates trace file writing.

The calling process always gets rank 0.

There are two possibilities:

1. Collect all information from the clients and pass them here (`numcontacts >= 0`, `contacts != NULL`)

2. Start the server first, pass its contact string to the clients (`numcontacts` >= `0`, `contacts` == `NULL`)

This call replaces starting the `VTserver` executable in a separate process. Parameters that used to be passed to the `VTserver` to control tracing and trace writing can be passed to `VT_initialize()` instead.

Parameters

`servername`	similar to clientname in `VT_clientinit()`: the name of the server. Currently only used for error messages. Copied by Intel® Trace Collector.
`numcontacts`	number of client processes
`contacts`	contact string for each client process (order is irrelevant); copied by Intel® Trace Collector

Return values

`contact`	will be set to a string which tells spawned children how to contact this server. Guaranteed not to contain spaces. The server may copy this string, but doesn't have to, because Intel® Trace Collector will not free this string until `VT_finalize()` is called. `contact` must have been set to `NULL` before calling this function.

Returns error code

VT_attach

```
int VT_attach(int root, int comm, int numchildren, int * childcomm)
```

Description

Connect to several new processes.

These processes must have been spawned already and need to know the contact string of the root process when calling `VT_clientinit()`.
`comm == VT_COMM_WORLD` is currently not implemented. It has some design problems: if several children want to use `VT_COMM_WORLD` to recursively spawn more processes, then their parents must also call `VT_attach()`, because they are part of this communicator. If the VTserver is part of the initial `VT_COMM_WORLD`, then `VT_attach()` with `VT_COMM_WORLD` won't work, because the VTserver does not know about the spawned processes and never calls `VT_attach()`.

Parameters

`root`	rank of the process that the spawned processes will contact
`comm`	either `VT_COMM_SELF` or `VT_COMM_WORLD`: in the first case root must be `0` and the spawned processes are connected to just the calling process. In the latter case all processes that share this `VT_COMM_WORLD` must call `VT_attach()` and are included in the new communicator. `root` then indicates whose contact infos were given to the children.

`numchildren`	number of children that the spawning processes will wait for

Return values

`childcomm`	an identifier for a new communicator that includes the parent processes in the same order as in their `VT_COMM_WORLD`, followed by the child processes in the order specified by their `procid` argument in `VT_clientinit()`. The spawned processes will have access to this communicator through `VT_get_parent()`.

Returns error code

VT_get_parent

```
int VT_get_parent (int * parentcomm)
```

Description

Returns the communicator that connects the process with its parent, or `VT_COMM_INVALID` if not spawned.

Return values

`parentcomm`	set to the communicator number that can be used to log communication with parents

Returns error code

C++ API

C++ API

These are wrappers around the C API calls which simplify instrumentation of C++ source code and ensure correct tracing if exceptions are used. Because all the member functions are provided as inline functions it is sufficient to include `VT.h` to use these classes with every C++ compiler.
Here are some examples how the C++ API can be used. `nohandles()` uses the simpler interface without storing handles, while `handles()` saves these handles in static instances of the definition classes for later reuse when the function is called again:

```
void nohandles()
{
   VT_Function func( "nohandles", "C++ API", __FILE__, __LINE__ );
}
void handles()
{
   static VT_SclDef scldef( __FILE__, __LINE__ );
      // VT_SCL_DEF_CXX( scldef ) could be used instead
```

```
   static VT_FuncDef funcdef( "handles", "C++ API" );
   VT_Function func( funcdef, scldef );
}
int main( int argc, char **argv )
{
   VT_Region region( "call nohandles()", "main" );
   nohandles();
   region.end();
   handles();
   handles();
   return 0;
}
```

VT_FuncDef Class Reference

Description

Defines a function on request and then remembers the handle.

Can be used to avoid the overhead of defining the function several times in `VT_Function`.

Constructor & Destructor Documentation

`VT_FuncDef (const char *symname, const char *classname)`

Member Function Documentation

int m_handle
Stores the function handle, 0 if not defined yet.

const char *m_symname
Stores the symbol name.

const char *m_classname
Stores the class name.

int GetHandle()
Checks whether the function is already defined or not.

Returns handle as soon as it is available, otherwise 0. Defining the function may be impossible for example, because Intel® Trace Collector was not initialized or ran out of memory.

VT_SclDef Class Reference

Description

Defines a source code location on request and then remembers the handle.

Can be used to avoid the overhead of defining the location several times in `VT_Function`. Best used together with the define `VT_SCL_DEF_CXX()`.

Constructor & Destructor Documentation

```
VT_SclDef( const char *file, int line )
```

Member Function Documentation

int m_handle
Stores the SCL handle, 0 if not defined yet.

const char *m_file
Stores the file name.

int m_line
Stores the line number.

int GetHandle()
Checks whether the SCL is already defined or not.

Returns handle as soon as it is available, else 0. Defining the function may be impossible for example, because Intel® Trace Collector was not initialized or ran out of memory.

#define VT_SCL_DEF_CXX(_sclvar) static VT_SclDef _sclvar(__FILE__, __LINE__)
This preprocessor macro creates a static source code location definition for the current file and line in C++.

VT_Function Class Reference

Description

In C++ an instance of this class should be created at the beginning of a function.

The constructor will then log the function entry, and the destructor the function exit.

Providing a source code location for the function exit manually is not supported, because this source code location would have to define where the function returns to. This cannot be determined at compile time.

Constructor & Destructor Documentation

VT_Function(const char *symname, const char *classname)
Defines the function with `VT_classdef()` and `VT_funcdef()`, then enters it.

This is less efficient than defining the function once and then reusing the handle. Silently ignores errors, like uninitialized Intel® Trace Collector.

Parameters:

`symname`	name of the function
`classname`	the class this function belongs to

VT_Function(const char *symname, const char *classname, const char *file, int line)
The same as the previous constructor, but also stores information about where the function is located in the source code.

Parameters:

`symname`	name of the function
`classname`	the class this function belongs to
`file`	name of source file, may but does not have to include path
`line`	line in this file where function starts

VT_Function(VT_FuncDef &funcdef)
This is a more efficient version which supports defining the function only once.

Parameters:

`funcdef`	this is a reference to the (usually static) instance that defines and remembers the function handle

VT_Function (VT_FuncDef &funcdef, VT_SclDef &scldef)
This is a more efficient version which supports defining the function and source code location only once.

Parameters:

`funcdef`	this is a reference to the (usually static) instance that defines and remembers the function handle
`scldef`	this is a reference to the (usually static) instance that defines and remembers the `scl` handle

~VT_Function()
The destructor marks the function exit.

VT_Region Class Reference

Description

This is similar to `VT_Function`, but should be used to mark regions within a function.
The difference is that source code locations can be provided for the beginning and end of the region, and one instance of this class can be used to mark several regions in one function.

Constructor & Destructor Documentation

VT_Region()
Default constructor. Does not start a region.

VT_Region(const char *symname, const char *classname)
Enters the region upon creation.

VT_Region(const char *symname, const char *classname, const char *file, int line)
The same as the previous constructor, but also stores information about where the region is located in the source code.

VT_Region(VT_FuncDef &funcdef)
This is a more efficient version which supports defining the region only once.

VT_Region(VT_FuncDef &funcdef, VT_SclDef &scldef)
This is a more efficient version which supports defining the region and source code location only once.

~VT_Region()
The destructor marks the region exit.

Member Function Documentation

void begin(const char *symname, const char *classname)
Defines the region with `VT_classdef()` and `VT_funcdef()`, then enters it.
This is less efficient than defining the region once and then reusing the handle. Silently ignores errors, like for example, uninitialized Intel® Trace Collector.

Parameters:

`symname`	the name of the region
`classname`	the class this region belongs to

void begin(const char *symname, const char *classname, const char *file, int line)
The same as the previous `begin()`, but also stores information about where the region is located in the source code.
Parameters:

`symname`	the name of the region
`classname`	the class this region belongs to
`file`	name of source file, may but does not have to include path
`line`	line in this file where region starts

void begin(VT_FuncDef &funcdef)
This is a more efficient version which supports defining the region only once.

Parameters:

`funcdef`	this is a reference to the (usually static) instance that defines and remembers the region handle

void begin(VT_FuncDef &funcdef, VT_SclDef &scldef)
This is a more efficient version which supports defining the region and source code location only once.

Parameters:

`funcdef`	this is a reference to the (usually static) instance that defines and remembers the region handle
`scldef`	this is a reference to the (usually static) instance that defines and remembers the `scl` handle

void end()
Leaves the region.

void end(const char *file, int line)
The same as the previous `end()`, but also stores information about where the region ends in the source code.
Parameters:

`file`	name of source file, may but does not have to include path
`line`	line in this file where region starts

void end (VT_SclDef &scldef)
This is a more efficient version which supports defining the source code location only once.

Parameters:

`scldef`	this is a reference to the (usually static) instance that defines and remembers the `scl` handle.

Configuration Reference

Configuration Reference

This section provides the information on the Intel® Trace Collector configuration:

- Configuration File Format – information on syntax of a configuration file.
- Protocol File – information on the protocol file listing all configuration settings of a particular run.
- Configuration Options – description of all configuration options available.

Configuration File Format

General Syntax

The configuration file is a plain ASCII file with the `.conf` extension containing a number of configuration options with their values, one option per line. Options are evaluated in the order they are listed.
Any line starting with the **#** character is ignored. Within a line, a whitespace separates fields, and double quotation marks **"** are used to quote fields containing whitespace. All text is case-insensitive, except for filenames.

Syntax of Parameters

Apart from having numeric or text values, some configuration options may have one of the values below. See description of particular options for their values.

Time Value

Time values are usually specified as a pair of one floating point value and one character that represents the unit: `c` for microseconds, `l` for milliseconds, `s` for seconds, `m` for minutes, `h` for hours, `d` for days and `w` for weeks. These elementary times are added with a plus sign. For instance, the string `1m+30s` refers to one minute and 30 seconds of execution time.

Boolean Value

Boolean values are set to `on`/`true` to turn something on and `off`/`false` to turn it off. Using only the option name without the `on`/`off` argument is the same as `on`.

Number of Bytes

The amount of bytes can be specified with optional suffices `B`/`KB`/`MB`/`GB`, which multiply the amount in front of them by 1, 1024, 1024^2, 1024^3, respectively. If no suffix is given the number specifies bytes.

Example

Below is an example of a valid configuration file.

```
# This line will be ignored
LOGFILE-NAME trace.stf
CURRENT-DIR                        "My                        Directory/tracing"
MEM-MAXBLOCKS                                                              8KB
OS-COUNTER-DELAY                                                            2s
KEEP-RAW-EVENTS ON
```

Protocol File

The protocol file lists all options with their values used when the program was started and can be used to restart an application with exactly the same options.

The protocol file is generated along with the tracefile, has the same basename and the `.prot` extension. It has the same syntax and entries as a Intel® Trace Collector configuration file.
All options are listed, even if they were not present in the original configuration. This way you can find about for example, the default value of `SYNCED-HOST/CLUSTER` on your machine. Comments tell where the value came from (default, modified by user, default value set explicitly by the user).
Besides the configuration entries, the protocol file contains some entries that are only informative. They are all introduced by the keyword `INFO`. The following information entries are supported:

INFO NUMPROCS

Syntax: *<num>*
Description: Number of processes in `MPI_COMM_WORLD`.

INFO CLUSTERDEF

Syntax: *<name>* `[`*<rank>*`:`*<pid>*`]+`
Description: For clustered systems, the processes with Unix process ID *<pid>* and rank in `MPI_COMM_WORLD` *<rank>* are running on the cluster node *<name>*. There will be one line per cluster node.

INFO PROCESS

Syntax: *<rank>* `"`*<hostname>*`" "`*<IP>*`"` *<pid>*
Description: For each process identified by its MPI *<rank>*, the *<hostname>* as returned by `gethostname()`, the *<pid>* from `getpid()` and all *<IP>* addresses that *<hostname>* translates into with `gethostbyname()` are given. IP addresses are converted to string with `ntoa()` and separated with commas. Both hostname and IP string might be empty, if the information was not available.

INFO BINMODE

Syntax: *<mode>*
Description: Records the floating-point and integer-length execution mode used by the application.

There may be other `INFO` entries that represent statistical data about the program run. Their syntax is explained in the file itself.

Configuration Options

Configuration Options

This topic gives information on the options that you can use to configure Intel® Trace Collector. For instructions on how to set these options, see Configuring Intel® Trace Collector.

- ACTIVITY
- ALTSTACK

- AUTOFLUSH
- CHECK
- CHECK-LEAK-REPORT-SIZE
- CHECK-MAX-DATATYPES
- CHECK-MAX-ERRORS
- CHECK-MAX-PENDING
- CHECK-MAX-REPORTS
- CHECK-MAX-REQUESTS
- CHECK-SUPPRESSION-LIMIT
- CHECK-TIMEOUT
- CHECK-TRACING
- CLUSTER
- COMPRESS-RAW-DATA
- COUNTER
- CURRENT-DIR
- DEADLOCK-TIMEOUT
- DEADLOCK-WARNING
- DEMANGLE
- DETAILED-STATES
- ENTER-USERCODE
- ENVIRONMENT
- EXTENDED-VTF
- FLUSH-PID
- FLUSH-PREFIX
- GROUP
- HANDLE-SIGNALS
- INTERNAL-MPI
- ITFLOGFILE
- KEEP-RAW-EVENTS
- LOGFILE-FORMAT
- LOGFILE-NAME
- LOGFILE-PREFIX
- LOGFILE-RANK
- MEM-BLOCKSIZE

- MEM-FLUSHBLOCKS
- MEM-INFO
- MEM-MAXBLOCKS
- MEM-MINBLOCKS
- MEM-OVERWRITE
- NMCMD
- OS-COUNTER-DELAY
- PCTRACE
- PCTRACE-CACHE
- PCTRACE-FAST
- PLUGIN
- PROCESS
- PROGNAME
- PROTOFILE-NAME
- STATE
- STATISTICS
- STF-PROCS-PER-FILE
- STF-USE-HW-STRUCTURE
- STOPFILE-NAME
- SYMBOL
- SYNC-MAX-DURATION
- SYNC-MAX-MESSAGES
- SYNC-PERIOD
- SYNCED-CLUSTER
- SYNCED-HOST
- TIME-WINDOWS
- TIMER
- TIMER-SKIP
- UNIFY-COUNTERS
- UNIFY-GROUPS
- UNIFY-SCLS
- UNIFY-SYMBOLS
- VERBOSE

ACTIVITY

Syntax

`ACTIVITY <pattern> <filter body>`

Variable

`VT_ACTIVITY`

Default

`on`

Description

A shortcut for `STATE "<pattern>:*"`.

ALTSTACK

Syntax

`ALTSTACK [on|off]`

Variable

`VT_ALTSTACK`

Description

Handling segfaults due to a stack overflow requires that the signal handler runs on an alternative stack, otherwise it will just segfault again, causing the process to terminate.

Because installing an alternative signal handler affects application behavior, it is normally not done. If it is known to work, it is enabled only for MPI correctness checking.

AUTOFLUSH

Syntax

`AUTOFLUSH [on|off]`

Variable

`VT_AUTOFLUSH`

Default

`on`

Description

If enabled (which it is by default), Intel Trace Collector appends blocks that are currently in main memory to one flush file per process. During trace file generation this data is taken from the flush file, so no data is lost. The number of blocks remaining in memory can be controlled with `MEM-MINBLOCKS`.

CHECK

Syntax

`CHECK <pattern><on|off>`

Variable

`VT_CHECK`

Default

`on`

Description

Enables or disables error checks matching the pattern.

CHECK-LEAK-REPORT-SIZE

Syntax

`CHECK-LEAK-REPORT-SIZE <number>`

Variable

`VT_CHECK_LEAK_REPORT_SIZE`

Default

`10`

Description

Determines the number of call locations to include in a summary of leaked requests or data types. By default only the top ten of the calls which have no matching free call are printed.

CHECK-MAX-DATATYPES

Syntax

`CHECK-MAX-DATATYPES <number>`

Variable

`VT_CHECK_MAX_DATATYPES`

Default

`1000`

Description

Each time the total number of currently defined data types exceeds a multiple of this threshold, a `LOCAL:DATATYPE:NOT_FREED` warning is printed with a summary of the calls where those requests were created.

Set this to `0` to disable the warning.

CHECK-MAX-ERRORS

Syntax

`CHECK-MAX-ERRORS <number>`

Variable

`VT_CHECK_MAX_ERRORS`

Default

`1`

Description

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.
Do not ignore errors, because they change the behavior of the application, thus the default value stops immediately when the first such error is found.

CHECK-MAX-PENDING

Syntax

`CHECK-MAX-PENDING <number>`

Variable

`VT_CHECK_MAX_PENDING`

Default

`20`

Description

Upper limit of pending messages that are reported per `GLOBAL:MSG:PENDING` error.

CHECK-MAX-REPORTS

Syntax

`CHECK-MAX-REPORTS <number>`

Variable

`VT_CHECK_MAX_REPORTS`

Default

`0`

Description

Number of reports (regardless whether they contain warnings or errors) that has to be reached by a process before aborting the application. `0` disables the limit. Just as with `CHECK-MAX-ERRORS`, this is a per-process counter.
It is disabled by default because the `CHECK-SUPPRESSION-LIMIT` setting already ensures that each type of error or warning is only reported a limited number of times. Setting `CHECK-MAX-REPORTS` would help to automatically shut down the application, if that is desired.

CHECK-MAX-REQUESTS

Syntax

`CHECK-MAX-REQUESTS <number>`

Variable

`VT_CHECK_MAX_REQUESTS`

Default

`100`

Description

Each time the total number of active requests or inactive persistent requests exceeds a multiple of this threshold, a `LOCAL:REQUEST:NOT_FREED` warning is printed with a summary of the calls where those requests were created.
Set this to `0` to disable just the warning at runtime without also disabling the warnings at the end of the application run. Disable the `LOCAL:REQUEST:NOT_FREED` check to suppress all warnings.

CHECK-SUPPRESSION-LIMIT

Syntax

`CHECK-SUPPRESSION-LIMIT <number>`

Variable

`VT_CHECK_SUPPRESSION_LIMIT`

Default

`10`

Description

Maximum number of times a specific error or warning is reported before suppressing further reports about it. The application continues to run and other problems are still reported. Just as with `CHECK-MAX-ERRORS` these are a per-process counters.

Note
This only counts per error check and does not distinguish between different incarnations of the error in different parts of the application.

CHECK-TIMEOUT

Syntax

`CHECK-TIMEOUT <time>`

Variable

`VT_CHECK_TIMEOUT`

Default

`5s`

Description

After stopping one process because it cannot or is not allowed to continue, the other processes are allowed to continue for this amount of time to see whether they run into other errors.

CHECK-TRACING

Syntax

`CHECK-TRACING [on|off]`

Variable

`VT_CHECK_TRACING`

Default

`off`

Description

By default, no events are recorded and no trace file is written during correctness checking with `libVTmc`. This option enables recording of all events also supported by the normal `libVT` and the writing of a trace file. The trace file also contains the errors found during the run.
In the normal libraries tracing is always enabled.

CLUSTER

Syntax

`CLUSTER <triplets> [on|off|no|discard]`

Variable

`VT_CLUSTER`

Description

Same as `PROCESS`, but filters are based on the host number of each process. Hosts are distinguished by their name as returned by `MPI_Get_processor_name()` and enumerated according to the lowest rank of the MPI processes running on them.

COMPRESS-RAW-DATA

Syntax

`COMPRESS-RAW-DATA [on|off]`

Variable

`VT_COMPRESS_RAW_DATA`

Default

`on` in Intel Trace Collector

Description

The Intel Trace Collector can store raw data in compressed format. The compression runs in the background and does not impact the merge process. By using `COMPRESS-RAW-DATA` option, you can save space in underlying file system and time in transfer over networks.

COUNTER

Syntax

`COUNTER <pattern> [on|off]`

Variable

`VT_COUNTER`

Description

Enables or disables a counter whose name matches the pattern. By default, all counters defined manually are enabled, whereas counters defined and sampled automatically by the Intel Trace Collector are disabled. Those automatic counters are not supported for every platform.

CURRENT-DIR

Syntax

`CURRENT-DIR <directory name>`

Variable

`VT_CURRENT_DIR`

Description

The Intel Trace Collector uses the current working directory of the process that reads the configuration on all processes to resolve relative path names. You can override the current working directory with this option.

DEADLOCK-TIMEOUT

Syntax

`DEADLOCK-TIMEOUT <delay>`

Variable

`VT_DEADLOCK_TIMEOUT`

Default

`1 minute`

Description

If Intel Trace Collector observes no progress for this amount of time in any process, then it assumes that a deadlock has occurred, stops the application and writes a trace file.

As usual, the value may also be specified with units, `1m` for one minute, for example.

DEADLOCK-WARNING

Syntax

`DEADLOCK-WARNING <delay>`

Variable

`VT_DEADLOCK_WARNING`

Default

`5 minutes`

Description

If on average the MPI processes are stuck in their last MPI call for more than this threshold, then a `GLOBAL:DEADLOCK:NO_PROGRESS` warning is generated. This is a sign of a load imbalance or a deadlock which cannot be detected because at least one process polls for progress instead of blocking inside an MPI call.

As usual, the value may also be specified with units, `1m` for one minute, for example.

DEMANGLE

Syntax

`DEMANGLE [on|off]`

Variable

`VT_DEMANGLE`

Default

`off`

Description

Intel® Trace Collector automatically demangles mangled names if this switch is enabled. Name demangling is used in compiler driven instrumentation and in correctness checking reports. Intel® Trace Collector uses `__cxa_demangle` from the C++ ABI or `UnDecorateSymbolName` on Windows* OS. On Linux* OS demangling C++ names only works with the naming scheme used by GCC 3.x and newer compilers.

Note
Some versions of Libstdc++ provide `__cxa_demangle` that does not work properly in all cases.

DETAILED-STATES

Syntax

`DETAILED-STATES [on|off|<level>]`

Variable

`VT_DETAILED_STATES`

Default

`off`

Description

Enables or disables logging of more information in calls to `VT_enterstate()`. That function might be used by certain MPI implementations, runtime systems or applications to log internal states. If that is the case, it will be mentioned in the documentation of those components.
<level> is a positive number, with larger numbers enabling more details:

1. `0` (= `off`) suppresses all additional states
2. `1` (= `on`) enables one level of additional states
3. `2`, `3`, ... enables even more details

ENTER-USERCODE

Syntax

`ENTER-USERCODE [on|off]`

Variable

`VT_ENTER_USERCODE`

Default

`on` in most cases, `off` for Java* function tracing

Description

Usually the Intel Trace Collector enters the `Application:User_Code` state automatically when registering a new thread. This makes little sense when function profiling is enabled, because then the user can choose whether he wants the `main()` function or the entry function of a child thread to be logged or not. Therefore it is always turned off for Java* function tracing. In all other cases it can be turned off manually with this configuration option.
However, without automatically entering this state and without instrumenting functions threads might be outside of any state and thus not visible in the trace although they exist. This may or may not be intended.

ENVIRONMENT

Syntax

`ENVIRONMENT [on|off]`

Variable

`VT_ENVIRONMENT`

Default

`on`

Description

Enables or disables logging of attributes of the runtime environment.

EXTENDED-VTF

Syntax

`EXTENDED-VTF`

Variable

`VT_EXTENDED_VTF`

Default

`off` in Intel Trace Collector, `on` in stftool.

Description

Several events can only be stored in STF, but not in VTF. The Intel Trace Collector libraries default to writing valid VTF trace files and thus skip these events. This option enables writing of non-standard VTF records in ASCII mode that Intel Trace Analyzer would complain about. In the stftool the default is to write these extended records, because the output is more likely to be parsed by scripts rather than by Intel Trace Analyzer.

FLUSH-PID

Syntax

`FLUSH-PID [on|off]`

Variable

`VT_FLUSH_PID`

Default

`on`

Description

The `-<pid>` part in the flush file name is optional and can be disabled with `FLUSH-PID off`.

FLUSH-PREFIX

Syntax

`FLUSH-PREFIX <directory name>`

Variable

`VT_FLUSH_PREFIX`

Default

Content of environment variables or `/tmp`

Description

Specifies the directory of the flush file. It can be an absolute or relative pathname; in the latter case, it is interpreted relative to the current working directory of the process writing it.

On Unix* systems, the flush file of each process is created and immediately removed while the processes keep their file open. This has two effects:

1. if processes get killed prematurely, flush files do not clutter the file system
2. during flushing, the remaining space on the file systems gets less although the file which grows is not visible anymore

The file name is `VT-flush-<program name>_<rank>-<pid>.dat`, with `<rank>` being the rank of the process in `MPI_COMM_WORLD` and `<pid>` the Unix process id.
A good default directory is searched for among the candidates listed below in this order:

1. first folder with more than 512MB
2. failing that, folder with most available space

Candidates (in this order) are the directories referred to with these environment variables and hard-coded directory names:

1. `BIGTEMP`
2. `FASTTEMP`
3. `TMPDIR`
4. `TMP`
5. `TMPVAR`
6. `/work`
7. `/scratch`
8. `/tmp`

GROUP

Syntax

`GROUP <name> <name>|<triplet>[, ...]`

Variable

`VT_GROUP`

Description

This option defines a new group. The members of the group can be other groups or processes enumerated with triplets. Groups are identified by their name. It is possible to refer to automatically generated groups (Example: those for the nodes in the machine), however, groups generated with API functions have to be defined on the process which reads the config to be usable in config groups.

Example

```
GROUP odd 1:N:2
GROUP even 0:N:2
GROUP "odd even" odd,even
```

HANDLE-SIGNALS

Syntax

`HANDLE-SIGNALS <triplets of signal numbers>`

Variable

`VT_HANDLE_SIGNALS`

Default

none in `libVTcs`, all in other fail-safe libs

Description

This option controls whether the Intel Trace Collector replaces a signal handler previously set by the application or runtime system with its own handler. `libVTcs` by default does not override handlers, while the fail-safe MPI tracing libraries do: otherwise they would not be able to log the reason for an abort by MPI.

Using the standard triplet notation, you can both list individual signals (Example: `3`) as well as a whole range of signals (`3,10:100`).

INTERNAL-MPI

Syntax

`INTERNAL-MPI [on|off]`

Variable

`VT_INTERNAL_MPI`

Default

`on`

Description

Allows tracing of events inside the MPI implementation. This is enabled by default, but even then it still requires an MPI implementation which actually records events. The Intel Trace Collector documentation describes in more detail how an MPI implementation might do that.

KEEP-RAW-EVENTS

Syntax

`KEEP-RAW-EVENTS [on|off]`

Variable

`VT_KEEP_RAW_EVENTS`

Default

`off` in Intel Trace Collector

Description

The Intel Trace Collector can merge the final trace from the collected data at the MPI finalization stage. Sometimes it may take much time, especially for large amount of MPI processes and for applications rich of MPI events. This option forces the Intel Trace Collector to store the raw data obtained in each process into the disk without the merge. Then, you can use the merge function offline.

LOGFILE-FORMAT

Syntax

`LOGFILE-FORMAT [ASCII|STF|STFSINGLE|SINGLESTF]`

Variable

`VT_LOGFILE_FORMAT`

Default

`STF`

Description

Specifies the format of the tracefile. ASCII is the traditional Vampir file format where all trace data is written into one file. It is human-readable.

The Structured Trace File (STF) is a binary format which supports storage of trace data in several files and allows Intel Trace Analyzer to analyze the data without loading all of it, so it is more scalable. Writing it is only supported by the Intel Trace Collector at the moment.

One trace in STF format consists of several different files which are referenced by one index file (`.stf`). The advantage is that different processes can write their data in parallel (see `STF-PROCS-PER-FILE`, `STF-USE-HW-STRUCTURE`). `SINGLESTF` rolls all of these files into one (`.single.stf`), which can be read without unpacking them again. However, this format does not support distributed writing, so for large program runs with many processes the generic STF format is better.

LOGFILE-NAME

Syntax

`LOGFILE-NAME` *<file name>*

Variable

`VT_LOGFILE_NAME`

Description

Specifies the name for the tracefile containing all the trace data. Can be an absolute or relative pathname; in the latter case, it is interpreted relative to the log prefix (if set) or the current working directory of the process writing it.

If unspecified, then the name is the name of the program plus `.avt` for ASCII, `.stf` for STF and `.single.stf` for single STF tracefiles. If one of these suffices is used, then they also determine the logfile format, unless the format is specified explicitly.
In the stftool the name has to be specified explicitly, either by using this option or as argument of the `--convert` or `--move` switch.

LOGFILE-PREFIX

Syntax

`LOGFILE-PREFIX` *<directory name>*

Variable

`VT_LOGFILE_PREFIX`

Description

Specifies the directory of the trace or log file. It can be an absolute or relative pathname; in the latter case, it is interpreted relative to the current working directory of the process writing it.

LOGFILE-RANK

Syntax

`LOGFILE-RANK <rank>`

Variable

`VT_LOGFILE_RANK`

Default

`0`

Description

Determines which process creates and writes the tracefile in `MPI_Finalize()`. Default value is the process reading the configuration file, or the process with rank `0` in `MPI_COMM_WORLD`.

MEM-BLOCKSIZE

Syntax

`MEM-BLOCKSIZE <number of bytes>`

Variable

`VT_MEM_BLOCKSIZE`

Default

`64KB`

Description

Intel Trace Collector keeps trace data in chunks of main memory that have this size.

MEM-FLUSHBLOCKS

Syntax

`MEM-FLUSHBLOCKS <number of blocks>`

Variable

`VT_MEM_FLUSHBLOCKS`

Default

`1024`

Description

This option controls when a background thread flushes trace data into the flush file without blocking the application. It has no effect if `AUTOFLUSH` is disabled. Setting this option to a negative value also disables the background flushing.
Flushing is started whenever the number of blocks in memory exceeds this threshold or when a thread needs a new block, but cannot get it without flushing.

If the number of blocks also exceeds `MEM-MAXBLOCKS`, then the application is stopped until the background thread has flushed enough data.

MEM-INFO

Syntax

`MEM-INFO` *`<threshold in bytes>`*

Variable

`VT_MEM_INFO`

Default

`500MB`

Description

If larger than zero, Intel Trace Collector prints a message to stderr each time more than this amount of new data has been recorded. These messages tell how much data was stored in RAM and in the flush file, and can serve as a warning when too much data is recorded.

MEM-MAXBLOCKS

Syntax

`MEM-MAXBLOCKS` *`<maximum number of blocks>`*

Variable

`VT_MEM_MAXBLOCKS`

Default

`4096`

Description

Intel Trace Collector does not allocate more than this number of blocks in main memory. If the maximum number of blocks is filled or allocating new blocks fails, Intel Trace Collector either flushes

some of them onto disk (`AUTOFLUSH`), or overwrites the oldest blocks (`MEM-OVERWRITE`) or stops recording further trace data.

MEM-MINBLOCKS

Syntax

`MEM-MINBLOCKS` *<minimum number of blocks after flush>*

Variable

`VT_MEM_MINBLOCKS`

Default

`0`

Description

When Intel Trace Collector starts to flush some blocks automatically, it can flush all of them (the default) or keep some in memory. The latter may be useful to avoid long delays or unnecessary disk activity.

MEM-OVERWRITE

Syntax

`MEM-OVERWRITE [on|off]`

Variable

`VT_MEM_OVERWRITE`

Default

`off`

Description

If auto flushing is disabled, enabling this option lets Intel Trace Collector overwrite the oldest blocks of trace data with more recent data.

NMCMD

Syntax

`NMCMD` *<command + args>* `"nm -P"`

Variable

`VT_NMCMD`

Description

If function tracing with GCC 2.95.2+'s `-finstrument-functions` is used, Intel Trace Collector is called at function entry/exit. Before logging these events, it has to map from the function's address in the executable to its name.
This is done with the help of an external program, usually nm. If it is not appropriate on your system, you can override the default. The executable's filename (including the path) is appended at the end of the command, and the command is expected to print the result to stdout in the format defined for POSIX.2 nm.

OS-COUNTER-DELAY

Syntax

`OS-COUNTER-DELAY` *<delay>*

Variable

`VT_OS_COUNTER_DELAY`

Default

`1 second`

Description

If OS counters have been enabled with the `COUNTER` configuration option, then these counters are sampled every *<delay>* seconds. As usual, the value may also be specified with units, `1m` for one minute, for example.

PCTRACE

Syntax

`PCTRACE [on|off|`*<trace levels>*`|`*<skip levels>*`:`*<trace levels>*`]`

Variable

`VT_PCTRACE`

Default

`off` for performance analysis, `on` otherwise

Description

Some platforms support the automatic stack sampling for MPI calls and user-defined events. Intel Trace Collector then remembers the Program Counter (PC) values on the call stack and translates

them to source code locations based on debug information in the executable. It can sample a certain number of levels (*<trace levels>*) and skip the initial levels (*<skip levels>*). Both values can be in the range of `0` to `15`.
Skipping levels is useful when a function is called from within another library and the source code locations within this library shall be ignored. `ON` is equivalent to `0:1` (no skip levels, one trace level).
The value specified with `PCTRACE` applies to all symbols that are not matched by any filter rule or where the relevant filter rule sets the logging state to `ON`. In other words, an explicit logging state in a filter rule overrides the value given with `PCTRACE`.

PCTRACE-CACHE

Syntax

```
PCTRACE-CACHE [on|off]
```

Variable

```
VT_PCTRACE_CACHE
```

Default

`on`

Description

When the reliable stack unwinding through `libunwind` is used, caching the previous stack back trace can reduce the number of times `libunwind` has to be called later on. When unwinding only a few levels this caching can negatively affect performance, therefore it can be turned off with this option.

PCTRACE-FAST

Syntax

```
PCTRACE-FAST [on|off]
```

Variable

```
VT_PCTRACE_FAST
```

Default

`on` for performance tracing, `off` for correctness checking

Description

Controls whether the fast, but less reliable stack unwinding is used or the slower, but less error-prone unwinding through `libunwind`. The fast unwinding relies on frame pointers, therefore all code must be compiled accordingly for it to work correctly.

PLUGIN

Syntax

`PLUGIN <plugin name>`

Variable

`VT_PLUGIN`

Description

If this option is used, the Intel Trace Collector activates the given plugin after initialization. The plugin takes over responsibility for all function wrappers and normal tracing is disabled. Most of the normal configuration options have no effect. Refer to the documentation of the plugin that you want to use for further information.

PROCESS

Syntax

`PROCESS <triplets> [on|off|no|discard]`

Variable

`VT_PROCESS`

Default

`0:N on`

Description

Specifies for which processes tracing is to be enabled. This option accepts a comma separated list of triplets, each of the form `<start>:<stop>:<incr>` specifying the minimum and maximum rank and the increment to determine a set of processes (similar to the Fortran 90 notation). Ranks are interpreted relative to `MPI_COMM_WORLD`, which means that they start with `0`. The letter `N` can be used as maximum rank and is replaced by the current number of processes. For example, to enable tracing only on odd process ranks, specify `PROCESS 0:N OFF` and `PROCESS 1:N:2 ON`.
A process that is turned off can later turn logging on by calling `VT_traceon()` (and vice versa). Using `no` disables Intel Trace Collector for a process completely to reduce the overhead even further, but also so that even `VT_traceon()` cannot enable tracing.
`discard` is the same as `no`, so data is collected and trace statistics is calculated, but the collected data is not actually written into the trace file. This mode is useful if looking at the statistics is sufficient: in this case there is no need to write the trace data.

PROGNAME

Syntax

`PROGNAME <file name>`

Variable

`VT_PROGNAME`

Description

This option can be used to provide a fallback for the executable name in case the Intel Trace Collector is unable to determine this name from the program arguments. It is also the base name for the trace file.

In Fortran, it may be technically impossible to determine the name of the executable automatically and the Intel Trace Collector may need to read the executable to find source code information (see `PCTRACE` config option). If the file name is unknown and not specified explicitly, `UNKNOWN` is used.

PROTOFILE-NAME

Syntax

`PROTOFILE-NAME <file name>`

Variable

`VT_PROTOFILE_NAME`

Description

Specifies the name for the protocol file containing the config options and (optionally) summary statistics for a program run. It can be an absolute or relative pathname; in the latter case, it is interpreted relative to the current working directory of the process writing it.

If unspecified, the name is the name of the tracefile with the suffix `.prot`.

STATISTICS

Syntax

`STATISTICS [on|off|<hash_size>]`

Variable

`VT_STATISTICS`

Default

`off`

Description

Enables or disables statistics about messages and symbols. These statistics are gathered by the Intel Trace Collector independently from logging them and stored in the tracefile. Apart from `on` and `off`, it allows specifying the hash size used on each collecting thread. For extensively instrumented codes or for codes with a volatile communication pattern, this might be useful to control its performance.

STATE

Syntax

`STATE <pattern> <filter body>`

Variable

`VT_STATE`

Default

`on`

Description

Defines a filter for any state or function that matches the pattern. Patterns are extended shell patterns: they may contain the wildcard characters `*`, `**`, `?` and `[]` to match any number of characters but not the colon, any number of characters including the colon, exactly one character or a list of specific characters. Pattern matching is case insensitive.
The state or function name that the pattern is applied to consists of a class name and the symbol name, separated by a `:` (colon). Deeper class hierarchies as in Java* or C++ may have several class names, also separated by a colon. The colon is special and not matched by the `*` or `?` wildcard. To match it use `**`. The body of the filter may specify the logging state with the same options as PCTRACE. On some platforms further options are supported, as described below.
Valid patterns are:

- `MPI:*` (all MPI functions)
- `*:*send*` (any function that contains "send" inside any class)
- `**:*send*` (any function that contains "send", even if the class actually consists of multiple levels; same as `**send*`)
- `MPI:*send*` (only send functions in MPI)

STF-PROCS-PER-FILE

Syntax

`STF-PROCS-PER-FILE <number of processes>`

Variable

`VT_STF_PROCS_PER_FILE`

Default

`16`

Description

In addition to or instead of combining trace data per node, the number of processes per file can be limited. This helps to restrict the amount of data that has to be loaded when analyzing a sub-set of the processes.

If `STF-USE-HW-STRUCTURE` is enabled, then `STF-PROCS-PER-FILE` has no effect unless it is set to a value smaller than the number of processes running on a node. To get files that contain exactly the data of *<n>* processes, set `STF-USE-HW-STRUCTURE` to `OFF` and `STF-PROCS-PER-FILE` to *<n>*.
In a single-process multi-threaded application trace, this configuration option is used to determine the number of threads per file.

STF-USE-HW-STRUCTURE

Syntax

`STF-USE-HW-STRUCTURE [on|off]`

Variable

`VT_STF_USE_HW_STRUCTURE`

Default

usually `on`

Description

If the STF format is used, trace information can be stored in different files. If this option is enabled, trace data of processes running on the same node are combined in one file for that node. This is enabled by default on most machines, because it both reduces inter-node communication during trace file generation and resembles the access pattern during analysis.lif each process is running on its own node, it is not enabled .

This option can be combined with `STF-PROCS-PER-FILE` to reduce the number of processes whose data is written into the same file even further.

STOPFILE-NAME

Syntax

`STOPFILE-NAME` *<file name>*

Variable

`VT_STOPFILE_NAME`

Description

Specifies the name of a file which indicates that the Intel Trace Collector should stop the application prematurely and write a tracefile. This works only with the fail-safe Intel Trace Collector libraries. On

Linux* systems the same behavior can be achieved by sending the signal `SIGINT` to one of the application processes, but this is not possible on Microsoft* Windows* OS.
If specified, the Intel Trace Collector checks for the existence of such a file from time to time. If detected, the stop file is removed again and the shutdown is initiated.

SYMBOL

Syntax

`SYMBOL <pattern> <filter body>`

Variable

`VT_SYMBOL`

Default

`on`

Description

A shortcut for `STATE "**:<pattern>"`.

SYNC-MAX-DURATION

Syntax:

`SYNC-MAX-DURATION <duration>`

Variable

`VT_SYNC_MAX_DURATION`

Default

`1 minute`

Description

Intel Trace Collector can use either a barrier at the beginning and the end of the program run to take synchronized time stamps on processes or it can use a more advanced algorithm based on statistical analysis of message round-trip times.

This option enables this algorithm by setting the maximum number of seconds that Intel Trace Collector exchanges messages among processes. A value less than or equal to zero disables the statistical algorithm.

The default duration is much longer than actually needed, because usually the maximum number of messages (set through `SYNC-MAX-MESSAGES`)is reached first. This setting mostly acts as a safe-guard against excessive synchronization times, at the cost of potentially reducing the quality of clock synchronization when reaching it and then sending less messages.

SYNC-MAX-MESSAGES

Syntax

`SYNC-MAX-MESSAGES <message number>`

Variable

`VT_SYNC_MAX_MESSAGES`

Default

`100`

Description

If `SYNC-MAX-DURATION` is larger than zero and thus statistical analysis of message round-trip times is done, then this option limits the number of message exchanges.

SYNC-PERIOD

Syntax

`SYNC-PERIOD <duration>`

Variable

`VT_SYNC_PERIOD`

Default

`-1 seconds` = `disabled`

Description

If clock synchronization through message exchanges is enabled (the default), then Intel Trace Collector can be told to do message exchanges during the application run automatically. By default, this is disabled and needs to be enabled by setting this option to a positive time value.

The message exchange is done by a background thread and thus needs a means of communication, which can execute in parallel to the application's communication, therefore it is not supported by the normal MPI tracing library `libVT`.

SYNCED-CLUSTER

Syntax

`SYNCED-CLUSTER [on|off]`

Variable

`VT_SYNCED_CLUSTER`

Default

`off`

Description

Use this setting to override whether Intel Trace Collector treats the clock of all processes anywhere in the cluster as synchronized or not. Whether Intel Trace Collector makes that assumption depends on the selected time source.

SYNCED-HOST

Syntax

`SYNCED-HOST [on|off]`

Variable

`VT_SYNCED_HOST`

Default

`off`

Description

Use this setting to override whether Intel Trace Collector treats the clock of all processes one the same node as synchronized or not. Whether Intel Trace Collector makes that assumption depends on the selected time source.

If `SYNCED-CLUSTER` is on, this option is ignored.

TIME-WINDOWS (Experimental)

Syntax

`TIME-WINDOWS <time_value1>:<time_value2>[,<time_value1:time_value2>]`
See the description of the time format in Time Value.

Variable

`VT_TIME_WINDOWS`

Description

Use the `TIME-WINDOWS` option to set up a time frame within which the Intel® Trace Collector will save the events into the trace file. When `TIME-WINDOWS` is not set, Intel Trace Collector collects the whole trace.
To set several time windows, use the necessary number of time frames separated by commas.

Note
In some cases correct order of messages can be lost and you can get a message about reversed timestamps:

```
[0] Intel® Trace Collector WARNING: message logging: 168 different messages, 0 (0.0%) sends without receive, 5 (3.0%) receives without send, 163 (97.0%) messages with reversed time stamps.
```

To avoid this issue, include the first communication into the first time window. The time of the first communication depends on the application.

Example

```
TIME-WINDOWS 0:1,10:20
```

In this case, Intel Trace Collector will trace the first communication, the events from the beginning to the first second and the events from the 10th to the 20th second.

TIMER

Syntax

`TIMER <timer name or LIST>`

Variable

`VT_TIMER`

Default

`gettimeofday`

Description

Intel Trace Collector can use different sources for time stamps. The availability of the different timers may depend on the actual machine configuration.

To get a full list, link an application with the Intel Trace Collector, then run it with this configuration option set to `LIST`. By setting the verbosity to `2` or higher, you get output for each node in a cluster. If initialization of a certain timer fails, no error messages are printed in this mode and the timer is specified as unavailable. To see error messages, run the program with `TIMER` set to the name of the timer that you want to use.

TIMER-SKIP

Syntax

`TIMER-SKIP <number> 0`

Variable

`VT_TIMER_SKIP`

Description

Number of intermediate clock sample points, which are to be skipped when running the `timertest` program: they then serve as check that the interpolation makes sense.

UNIFY-COUNTERS

Syntax

```
UNIFY-COUNTERS [on|off]
```

Variable

```
VT_UNIFY_COUNTERS
```

Default

```
on
```

Description

Same as `UNIFY-SYMBOLS` for counters.

UNIFY-GROUPS

Syntax

```
UNIFY-GROUPS [on|off]
```

Variable

```
VT_UNIFY_GROUPS
```

Default

```
on
```

Description

Same as `UNIFY-SYMBOLS` for groups.

UNIFY-SCLS

Syntax

```
UNIFY-SCLS [on|off]
```

Variable

```
VT_UNIFY_SCLS
```

Default

`on`

Description

Same as `UNIFY-SYMBOLS` for SCLs.

UNIFY-SYMBOLS

Syntax

`UNIFY-SYMBOLS [on|off]`

Variable

`VT_UNIFY_SYMBOLS`

Default

`on`

Description

During post-processing Intel Trace Collector unifies the ids assigned to symbols on different processes. This step is redundant only if all processes define all symbols in exactly the same order with exactly the same names. As Intel Trace Collector cannot recognize this automatically, the unification can be disabled by the user to reduce the time required for trace file generation. Before using this option, make sure that your program really defines symbols consistently.

VERBOSE

Syntax

`VERBOSE [on|off|<level>]`

Variable

`VT_VERBOSE`

Default

`on`

Description

Enables or disables additional output on stderr. *<level>* is a positive number, with larger numbers enabling more output:

1. 0 (= `off`) disables all output
2. 1 (= `on`) enables only one final message about generating the result
3. 2 enables general progress reports by the main process
4. 3 enables detailed progress reports by the main process

5. 4 the same, but for all processes (if multiple processes are used at all)
6. Levels larger than 2 may contain output that only makes sense to the developers of the Intel Trace Collector.

Correctness Checking Errors

Correctness Checking Errors

This topic introduces the errors types that Intel® Trace Collector supports and explains how it detects them.

- Supported Errors
- How it Works

Supported Errors

Errors fall into two different categories:

- Local errors that need only the information available in the process itself and do not require additional communication between processes
- Global errors that require information from other processes

Another aspect of errors is whether the application can continue after they occurred. Minor problems are reported as warnings and allow the application to continue, but they lead to resource leaks or portability problems. Real errors are invalid operations that can only be skipped to proceed, but this either changes the application semantic (for example, transmission errors) or leads to follow-up errors (for example, skipping an invalid send can lead to a deadlock because of the missing message). Fatal errors cannot be resolved at all and require an application shutdown.

Problems are counted separately per process. Disabled errors are neither reported nor counted, even if they still happen to be detected. The application will be aborted as soon as a certain number of errors are encountered: obviously the first fatal error always requires an abort. Once the number of errors reaches `CHECK-MAX-ERRORS` or the total number of reports (regardless whether they are warnings or errors) reaches `CHECK-MAX-REPORTS` (whatever comes first), the application is aborted. These limits apply to each process separately. Even if one process gets stopped, the other processes are allowed to continue to see whether they run into further errors. The whole application is then aborted after a certain trace period. This timeout can be set through `CHECK-TIMEOUT`.
The default for `CHECK-MAX-ERRORS` is `1` so that the first error already aborts, whereas `CHECK-MAX-REPORTS` is at `100` and thus that many warnings errors are allowed. Setting both values to `0` removes the limits. Setting `CHECK-MAX-REPORTS` to `1` turns the first warning into a reason to abort.
When using an interactive debugger the limits can be set to 0 manually and thus removed, because the user can decide to abort using the normal debugger facilities for application shutdown. If he chooses to continue then Intel® Trace Collector will skip over warnings and non-fatal errors and try to proceed. Fatal errors still force Intel® Trace Collector to abort the application.

See the lists of supported errors (the description provides just a few keywords for each error, a more detailed description can be found in the following sections).

Local Errors

Error Name	Type	Description
LOCAL:EXIT:SIGNAL	Fatal	Process terminated by fatal signal
LOCAL:EXIT:BEFORE_MPI_FINALIZE	Fatal	Process exits without calling MPI_Finalize()
LOCAL:MPI:CALL_FAILED	Depends on MPI and error	MPI itself or wrapper detects an error
LOCAL:MEMORY:OVERLAP	Warning	Multiple MPI operations are started using the same memory
LOCAL:MEMORY:ILLEGAL_MODIFICATION	Error	Data modified while owned by MPI
LOCAL:MEMORY:INACCESSIBLE	Error	Buffer given to MPI cannot be read or written
LOCAL:MEMORY:ILLEGAL_ACCESS	Error	Read or write access to memory currently owned by MPI
LOCAL:MEMORY:INITIALIZATION	Error	Distributed memory checking
LOCAL:REQUEST:ILLEGAL_CALL	Error	Invalid sequence of calls
LOCAL:REQUEST:NOT_FREED	Warning	Program creates suspiciously high number of requests or exits with pending requests
LOCAL:REQUEST:PREMATURE_FREE	Warning	An active request has been freed
LOCAL:DATATYPE:NOT_FREED	Warning	Program creates high number of data types
LOCAL:BUFFER:INSUFFICIENT_BUFFER	Warning	Not enough space for buffered send

Global Errors

Error Name	Type	Description
GLOBAL:MSG/COLLECTIVE:DATATYPE:MISMATCH	Error	The type signature does not match

GLOBAL:MSG/COLLECTIVE:DATA_TRANSMISSION_CORRUPTED	Error	Data modified during transmission
GLOBAL:MSG:PENDING	Warning	Program terminates with unreceived messages
GLOBAL:DEADLOCK:HARD	Fatal	A cycle of processes waiting for each other
GLOBAL:DEADLOCK:POTENTIAL	Fatal[a]	A cycle of processes, one or more in blocking send
GLOBAL:DEADLOCK:NO_PROGRESS	Warning	Warning when application might be stuck
GLOBAL:COLLECTIVE:OPERATION_MISMATCH	Error	Processes enter different collective operations
GLOBAL:COLLECTIVE:SIZE_MISMATCH	Error	More or less data than expected
GLOBAL:COLLECTIVE:REDUCTION_OPERATION_MISMATCH	Error	Reduction operation inconsistent
GLOBAL:COLLECTIVE:ROOT_MISMATCH	Error	Root parameter inconsistent
GLOBAL:COLLECTIVE:INVALID_PARAMETER	Error	Invalid parameter for collective operation
GLOBAL:COLLECTIVE:COMM_FREE_MISMATCH	Warning	`MPI_Comm_free()` must be called collectively

[a] if check is enabled, otherwise it depends on the MPI implementation

How It Works

How It Works

Understanding how Intel® Trace Collector finds the various supported errors is important because it helps to understand what the different configuration options mean, what Intel® Trace Collector can do and what it cannot, and how to interpret the results.

Just as for performance analysis, Intel® Trace Collector intercepts all MPI calls using the MPI profiling interface. It has different wrappers for each MPI call. In these wrappers it can execute additional checks not normally done by the MPI implementation itself.

For global checks Intel® Trace Collector uses two different methods for transmitting the additional information: in collective operations it executes another collective operation before or after the original operation, using the same communicator[1]. For point-to-point communication it sends one additional message over a shadow communicator for each message sent by the application.

In addition to exchanging this extra data through MPI itself, Intel® Trace Collector also creates one background thread per process. These threads are connected to each other through TCP sockets and thus can communicate with each other even while MPI is being used by the main application thread.

For distributed memory checking and locking memory that the application should not access, Intel® Trace Collector interacts with Valgrind* through Valgrind's client request mechanism. Valgrind tracks definedness of memory (that is, whether it was initialized or not) within a process; Intel® Trace Collector extends that mechanism to the whole application by transmitting this additional information between processes using the same methods which also transmit the additional data type information and restoring the correct Valgrind state at the recipient.

Without Valgrind the `LOCAL:MEMORY:ILLEGAL_MODIFICATION` check is limited to reporting write accesses which modified buffers; typically this is detected long after the fact. With Valgrind, memory which the application hands over to MPI is set to "inaccessible" in Valgrind by Intel® Trace Collector and accessibility is restored when ownership is transferred back. In between any access by the application is flagged by Valgrind right at the point where it occurs. Suppressions are used to avoid reports for the required accesses to the locked memory by the MPI library itself.

See Also

Running with Valgrind*

Parameter Checking

(LOCAL:MPI:CALL_FAILED)

Most parameters are checked by the MPI implementation itself. Intel® Trace Collector ensures that the MPI does not abort when it finds an error, but rather reports back the error through a function's result code. Then Intel® Trace Collector looks at the error class and depending on the function where the error occurred decides whether the error has to be considered as a warning or a real error. As a general rule, calls which free resources lead to warnings and everything else is an error. The error report of such a problem includes a stack backtrace (if enabled) and the error message generated by MPI.

To catch MPI errors this way, Intel® Trace Collector overrides any error handlers installed by the application. Errors will always be reported, even if the application or test program sets an error handler to skip over known and/or intentionally bad calls. Because the MPI standard does not guarantee that errors are detected and that proceeding after a detected error is possible, such

programs are not portable and should be fixed. Intel® Trace Collector on the other hand knows that proceeding despite an error is allowed by all supported MPIs and thus none of the parameter errors is considered a hard error.

Communicator handles are checked right at the start of an MPI wrapper by calling an MPI function which is expected to check its arguments for correctness. Data type handles are tracked and then checked by Intel® Trace Collector itself. The extra parameter check is visible when investigating such an error in a debugger and although perhaps unexpected is perfectly normal. It is done to centralize the error checking.

Premature Exit

(LOCAL:EXIT)

Intel® Trace Collector monitors the ways how a process can abort prematurely: otherwise fatal signals are caught in Intel® Trace Collector signal handlers. An `atexit()` handler detects situations where the application or some library decides to quit. `MPI_Abort()` is also intercepted.
This error is presented just like a `LOCAL:MPI:CALL_FAILED`, with the same options for investigating the problem in a debugger. However, these are hard errors and the application cannot continue to run.

Overlapping Memory

(LOCAL:MEMORY:OVERLAP)

Intel® Trace Collector keeps track of memory currently in use by MPI and before starting a new operation, checks that the memory that it references is not in use already.

The MPI standard explicitly transfers ownership of memory to MPI even for send operations. The application is not allowed to read it while a send operation is active and must not setup another send operation which reads it either. The rationale is that the MPI might modify the data in place before sending it and might revert the change afterwards. In practice MPI implementation do not modify the memory, so this is a minor problem and just triggers a warning.

Obviously, writing into the same memory twice in possibly random order or writing into memory which the MPI might read from is a real error. However, detecting these real errors is harder for message receives because the size of the buffer given to MPI might be larger than the actual message: even if buffers overlap, the messages might be small enough to not lead to writes into the same memory. Because the overlap check is done when a send buffer is handed over to MPI, only a warning is generated. The application might be able to continue normally, but the source code should be fixed because under a strict interpretation of the MPI standard using the same buffer twice is already illegal even if the actual messages do not overlap.

Because the problem might be at the place where the memory was given to MPI initially and not where it is reused, Intel® Trace Collector also provides both call stacks.

Detecting Illegal Buffer Modifications

(LOCAL:MEMORY:ILLEGAL_MODIFICATION)

MPI owns the memory that active communication references. The application must not touch it during that time. Illegal writes into buffers that the MPI is asked to send are detected by calculating a checksum of the data immediately before the request is activated and comparing it against a checksum when the send completes. If the checksum is different, someone must have modified the buffer. The reported `LOCAL:MEMORY:ILLEGAL_MODIFICATION` is a real error.
This problem is more common with non-blocking communication because the application gets control back while MPI still owns the buffer and then might accidentally modify the buffer. For non-blocking communication the call stacks of where the send was initiated and where it completed are provided. For persistent requests it is also shown where it was created.

The problem might also occur for blocking communication, for example when the MPI implementation incorrectly modifies the send buffer, the program is multithreaded and writes into it or other communication happens to write into the buffer. In this case only the call stack of the blocking call where the problem was detected gets printed.

Strictly speaking, reads are also illegal because the MPI standard makes no guaranteed about the content of buffers while MPI owns them. Because reads do not modify buffers, such errors are not detected. Writes are also not detected when they happen (which would make debugging a lot easier) but only later when the damage is detected.

Buffer Given to MPI Cannot Be Read or Written

(LOCAL:MEMORY:INACCESSIBLE)

During the check for `LOCAL:MEMORY:ILLEGAL_MODIFICATION` of a send buffer Intel® Trace Collector will read each byte in the buffer once. This works for contiguous as well as non-contiguous data types. If any byte cannot be read because the memory is inaccessible, a `LOCAL:MEMORY:INACCESSIBLE` is reported. This is an error because it is only possible to proceed by skipping the entire operation.
Disabling the `LOCAL:MEMORY:ILLEGAL_MODIFICATION` check also disables the accessibility check and send operations are then treated like receive operations: for receive operations no similar check is performed because the MPI standard does not say explicitly that the whole receive buffer has to be accessible - only the part into which an incoming message actually gets copied must be writable.

Violations of that rule are caught and reported as fatal `LOCAL:EXIT:SIGNAL` errors.

Distributed Memory Checking

(LOCAL:MEMORY:INITIALIZATION)

This feature is enabled by default if all processes run under Valgrind*. If that is not the case, it is disabled. If in doubt, check the configuration summary at the beginning of the run to see whether this feature was enabled or not. There are no Intel® Trace Collector error reports with this type; Valgrind's error reports have to be watched instead to find problems related to memory initialization. See the section "Use of uninitialized values" in Valgrind's user guide for details.

If enabled, then Valgrind's tracking of memory definedness is extended to the whole application. For applications which transmit partially initialized data between processes, this avoids two cases:

- False positive: sending the message with the partially initialized data triggers a valgrind report for send or write system calls at the sender side

- False negative: at the recipient, valgrind incorrectly assumes that all incoming data is completely initialized and thus will not warn if the uninitialized data influences the control flow in the recipient; normally it would report that

To handle the false positive case Valgrind must have been started with the suppression file provided with Intel® Trace Collector. The `local_memory_valgrind` example (available at https://software.intel.com/en-us/product-code-samples) demonstrates both cases.
Turning this feature off is useful if the application is supposed to be written in such a way that it never transmits uninitialized data. In that case Intel® Trace Collector suppression file should not be used because it would suppress warnings at the sender and the `LOCAL:MEMORY:ILLEGAL_ACCESS` must be disabled as it would cause extra valgrind reports.

See Also

Running with Valgrind*

Illegal Memory Access

(LOCAL:MEMORY:ILLEGAL_ACCESS)

This feature depends on valgrind the same way as `LOCAL:MEMORY:INITIALIZATION`. This check goes beyond `LOCAL:MEMORY:ILLEGAL_MODIFICATION` by detecting also reads and reporting them through valgrind at the point where the access happens. Disabling it might improve performance and help if the provided suppression rules do not manage to suppress reports about valid accesses to locked memory.

Request Handling

(LOCAL:REQUEST)

When the program terminates Intel® Trace Collector prints a list of all unfreed MPI requests together with their status. Unfreed requests are usually currently active and application should have checked their status before terminating. Persistent requests can also be passive and need to be freed explicitly with `MPI_Request_free()`.
Not freeing requests blocks resources inside the MPI and can cause application failures. Each time the total number of active requests or inactive persistent requests exceeds another multiple of the `CHECK-MAX-REQUESTS` threshold (that is, after 100, 200, 300, . . . requests) a `LOCAL:REQUEST:NOT_FREED` warning is printed with a summary of the most frequent calls where those requests were created. The number of calls is configured through `CHECK-LEAK-REPORT-SIZE`.
Finalizing the application with pending requests is not an error according to the MPI standard, but is not a good practice and can potentially mask real problems. Therefore a request leak report will be always generated during finalize if at least one request was not freed.

If there are pending receives the check for pending incoming messages is disabled because some or all of them might match with the pending receives.

Active requests that were explicitly deleted with `MPI_Request_free()` will show up in another leak report if they have not completed by the time when the application terminates. Most likely this is due to not having a matching send or receive elsewhere in the application, but it might also be caused by posting and deleting a request and then terminating without giving it sufficient time to complete.

The MPI standard recommends that receive requests are not freed before they have completed. Otherwise it is impossible to determine whether the receive buffer can be read. Although not strictly marked an error in the standard, a `LOCAL:REQUEST:PREMATURE_FREE` warning is reported if the application frees such a request prematurely. For send requests the standard describes a method how the application can determine that it is safe to reuse the buffer, thus this is not reported as an error. In both cases actually deleting the request is deferred in a way which is transparent to the application: at the exit from all MPI calls which communicate with other processes Intel® Trace Collector will check whether any of them has completed and then execute the normal checking that it does at completion of a request (`LOCAL:MEMORY:ILLEGAL_MODIFICATION`) and also keep track of the ownership of the memory (`LOCAL:MEMORY:OVERLAP`).
In addition not freeing a request or freeing it too early, persistent requests also require that calls follow a certain sequence: create the request, start it and check for completion (can be repeated multiple times), delete the request. Starting a request while it is still active is an error which is reported as `LOCAL:REQUEST:ILLEGAL_CALL`. Checking for completion of an inactive persistent request on the other hand is not an error.

Datatype Handling

(LOCAL:DATATYPE)

Unfreed data types can cause the same problems as unfreed requests, so the same kind of leak report is generated for them when their number exceeds `CHECK-MAX-DATATYPES`. However, because not freeing data types is common practice there is no leak report during finalize unless their number exceeds the threshold at that time. That is in contrast to requests which are always reported then.

Buffered Sends

(LOCAL:BUFFER:INSUFFICIENT_BUFFER)

Intel® Trace Collector intercepts all calls related to buffered sends and simulates the worst-case scenario that the application has to be prepared for according to the standard. By default (`GLOBAL:DEADLOCK:POTENTIAL` enabled) it also ensures that the sends do not complete before there is a matching receive.
By doing both it detects several different error scenarios which all can lead to insufficient available buffer errors that might not occur depending on timing and/or MPI implementation aspects:

Buffer Size: The most obvious error is that the application did not reserve enough buffer to store the message(s), perhaps because it did not actually calculate the size with `MPI_Pack_size()` or forgot to add the `MPI_BSEND_OVERHEAD`. This might not show up if the MPI implementation bypasses the buffer, for example, for large messages. See the `local_buffered_send_size` example at the online samples resource.
Race Condition: Memory becomes available again only when the oldest messages are transmitted. It is the responsibility of the application to ensure that this happens in time before the buffer is required again; without suitable synchronization an application might run only because it is lucky and the recipients enter their receives early enough. See the `local_buffered_send_race` and `local_buffered_send_policy` examples at the online samples resource.
Deadlock: `MPI_Buffer_detach()` will block until all messages inside the buffer have been sent. This can lead to the same (potential) deadlocks as normal sends. See the `local_buffered_send_deadlock` example at the online samples resource.

Since it is critical to understand how the buffer is currently being used when a new buffered send does not find enough free space to proceed, the `LOCAL:BUFFER:INSUFFICIENT_BUFFER` error message contains all information about free space, active and completed messages and the corresponding memory ranges. Memory ranges are given using the [*<start address>*, *<end address>*] notation where the *<end address>* is not part of the memory range. For convenience the number of bytes in each range is also printed. For messages this includes the `MPI_BSEND_OVERHEAD`, so even empty messages have a non-zero size.

Deadlocks

(GLOBAL:DEADLOCK)

Deadlocks are detected through a heuristic: the background thread in each process cooperates with the MPI wrappers to detect that the process is stuck in a certain MPI call. That alone is not an error because some other processes might still make progress. Therefore the background threads communicate if at least one process appears to be stuck. If all processes are stuck, this is treated as a deadlock. The timeout after which a process and thus the application is considered as stuck is configurable with `DEADLOCK-TIMEOUT`.
The timeout defaults to one minute which should be long enough to ensure that even very long running MPI operations are not incorrectly detected as being stuck. In applications which are known to execute correct MPI calls much faster, it is advisable to decrease this timeout to detect a deadlock sooner.

This heuristic fails if the application is using non-blocking calls like `MPI_Test()` to poll for completion of an operation which can no longer complete. This case is covered by another heuristic: if the average time spent inside the last MPI call of each process exceeds the `DEADLOCK-WARNING` threshold, then a `GLOBAL:DEADLOCK:NO_PROGRESS` warning is printed, but the application is allowed to continue because the same high average blocking time also occurs in correct application with a high load imbalance. For the same reason the warning threshold is also higher than the hard deadlock timeout.
To help analyzing the deadlock, Intel® Trace Collector prints the call stack of all process. A real hard deadlock exists if there is a cycle of processes waiting for data from the previous process in the cycle. This data dependency can be an explicit `MPI_Recv()`, but also a collective operation like `MPI_Reduce()`.
If message are involved in the cycle, then it might help to replace send or receive calls with their non-blocking variant. If a collective operation prevents one process from reaching a message send that another process is waiting for, then reordering the message send and the collective operation in the first process would fix the problem.

Another reason could be messages which were accidentally sent to the wrong process. This can be checked in debuggers which support that by looking at the pending message queues. In the future Intel® Trace Collector might also support visualizing the program run in Intel® Trace Analyzer in case of an error. This would help to find messages which were not only sent to the wrong process, but also received by that processes and thus do not show up in the pending message queue.

In addition to the real hard deadlock from which the application cannot recover MPI applications might also contain potential deadlocks: the MPI standard does not guarantee that a blocking send returns unless the recipient calls a matching receive. In the simplest case of a head-to-head send with two processes, both enter a send and then the receive for the message that the peer just sent. This deadlocks unless the MPI buffers the message completely and returns from the send without waiting for the corresponding receive.

Because this relies on undocumented behavior of MPI implementations this is a hard to detect portability problem. Intel® Trace Collector detects these `GLOBAL:DEADLOCK:POTENTIAL` errors by turning each normal send into a synchronous send. The MPI standard then guarantees that the send blocks until the corresponding receive is at least started. Send requests are also converted to their synchronous counterparts; they block in the call which waits for completion. With these changes any potential deadlock automatically leads to a real deadlock at runtime and will be handled as described above. To distinguish between the two types, check whether any process is stuck in a send function. Due to this way of detecting it, even the normally non-critical potential deadlocks do not allow the application to proceed.

Checking Message Transmission

(GLOBAL:MSG)

For each application message, another extra message is sent which includes:

- Data type signature hash code (for `GLOBAL:MSG:DATATYPE:MISMATCH`)
- Checksum of the data (for `GLOBAL:MSG:DATA_TRANSMISSION_CORRUPTED`)
- Stack backtrace for the place where the message was sent (for both of these errors and also for `GLOBAL:MSG:PENDING`)

Only disabling of all of these three errors avoids the overhead for the extra messages.

Buffered messages which are not received lead to a resource leak. They are detected each time a communicator is freed or (if a communicator does not get freed) when the application terminates.

The information provided includes a call stack of where the message was sent as well as the current call stack where the error is detected.

Datatype Mismatches

(GLOBAL:*:DATATYPE:MISMATCH)

Data type mismatches are detected by calculating a hash value of the data type signature and comparing that hash value: if the hash values are different, the type signatures must have been different too and an error is reported. Because the information about the full type signature at the sender is not available, it has to be deduced from the function call parameters and/or source code locations where the data is transmitted.

If the hash values are identical, then there is some small chance that the signatures were different although no error is reported. Because of the choice of a very advanced hash function[2] this is very unlikely. This hash function can also be calculated more efficiently than traditional hash functions.

Data Modified during Transmission

(GLOBAL:*:DATA_TRANSMISSION_CORRUPTED)

After checking that the data type signatures in a point-to-point message transfer or collective data gather/scatter operation at sender and receiver match, Intel ®Trace Collector also verifies that the data was transferred correctly by comparing additional checksums that are calculated inside the sending and receiving process. This adds another end-to-end data integrity check which will fail if

any of the components involved in the data transmission malfunctioned (MPI layer, device drivers, hardware).

In cases where this `GLOBAL:*:DATA_TRANSMISSION_CORRUPTED` error is obviously the result of some other error, it is not reported separately. This currently works for truncated message receives and data type mismatches.

Checking Collective Operations

(GLOBAL:COLLECTIVE)

Checking correct usage of collective operations is easier than checking messages. At the beginning of each operation, Intel® Trace Collector broadcasts the same data from rank #0 of the communicator. This data includes:

- Type of the operation
- Root (zero if not applicable)
- Reduction type (predefined types only)

Now all involved processes check these parameters against their own parameters and report an error in case of a mismatch. If the type is the same, for collective operations with a root process that rank and for reduce operations the reduction operation are also checked. The `GLOBAL:COLLECTIVE:REDUCTION_OPERATION_MISMATCH` error can only be detected for predefined reduction operation because it is impossible to verify whether the program code associated with a custom reduction operation has the same semantic on all processes. After this step depending on the operation different other parameters are also shared between the processes and checked.
Invalid parameters like `MPI_DATATYPE_NULL` where a valid data type is required are detected while checking the parameters. They are reported as one `GLOBAL:COLLECTIVE:INVALID_PARAMETER` error with a description of the parameter which is invalid in each process. This leads to less output than printing one error for each process.
If any of these checks fails, the original operation is not executed on any process. Therefore proceeding is possible, but application semantic will be affected.

Freeing Communicators

(GLOBAL:COLLECTIVE:COMM_FREE_MISMATCH)

A mistake related to `MPI_Comm_free()` is freeing them in different orders on the involved processes. The MPI standard specifies that `MPI_Comm_free()` must be entered by the processes in the communicator collectively. Some MPIs including Intel® MPI Library deadlock if this rule is broken, whereas others implement `MPI_Comm_free()` as a local call with no communication.
To ensure that this error is detected all the time, Intel® Trace Collector treats `MPI_Comm_free()` just like the other collective operations. There is no special error message for `GLOBAL:COLLECTIVE:COMM_FREE_MISMATCH`, it will be reported as a mismatch between collective calls (`GLOBAL:COLLECTIVE:OPERATION_MISMATCH`) or a deadlock, so `GLOBAL:COLLECTIVE:COMM_FREE_MISMATCH` just refers to the check which enables or disables this test, not a specific error instance.

Structured Tracefile Format

Structured Tracefile Format

The Structured Trace File Format (STF) is a format that stores data in several physical files by default. This chapter describes the structure of this format and provides the technical background to configure and work with the STF format. It is safe to skip over this chapter because all configuration options that control writing of STF have reasonable default values.

The development of STF was motivated by the observation that the conventional approach of handling trace data in a single trace file is not suitable for large applications or systems, where the trace file can quickly grow into the tens of gigabytes range. On the display side, such huge amounts of data cannot be squeezed into one display at once. There should be mechanisms to enable one to start at a coarser level and then dive into details. Additionally, the ability to request and inspect only parts of the data becomes essential with the amount of trace data growing.

These requirements necessitate a more powerful data organization than the previous Intel® Trace Analyzer tracefile format can provide. In response to this, the STF has been developed. The aim of the STF is to provide a file format which:

- Can arbitrarily be partitioned into several files, each one containing a specific subset of the data
- Allows fast random access and easy extraction of data
- Is extensible, portable, and upward compatible
- Is clearly defined and structured
- Can efficiently exploit parallelism for reading and writing
- Is as compact as possible

The traditional tracefile format is only suitable for small applications, and cannot efficiently be written in parallel. Also, it was designed for reading the entire file at once, rather than for extracting arbitrary data. The structured tracefile implements these new requirements, with the ability to store large amounts of data in a more compact form.

STF Components

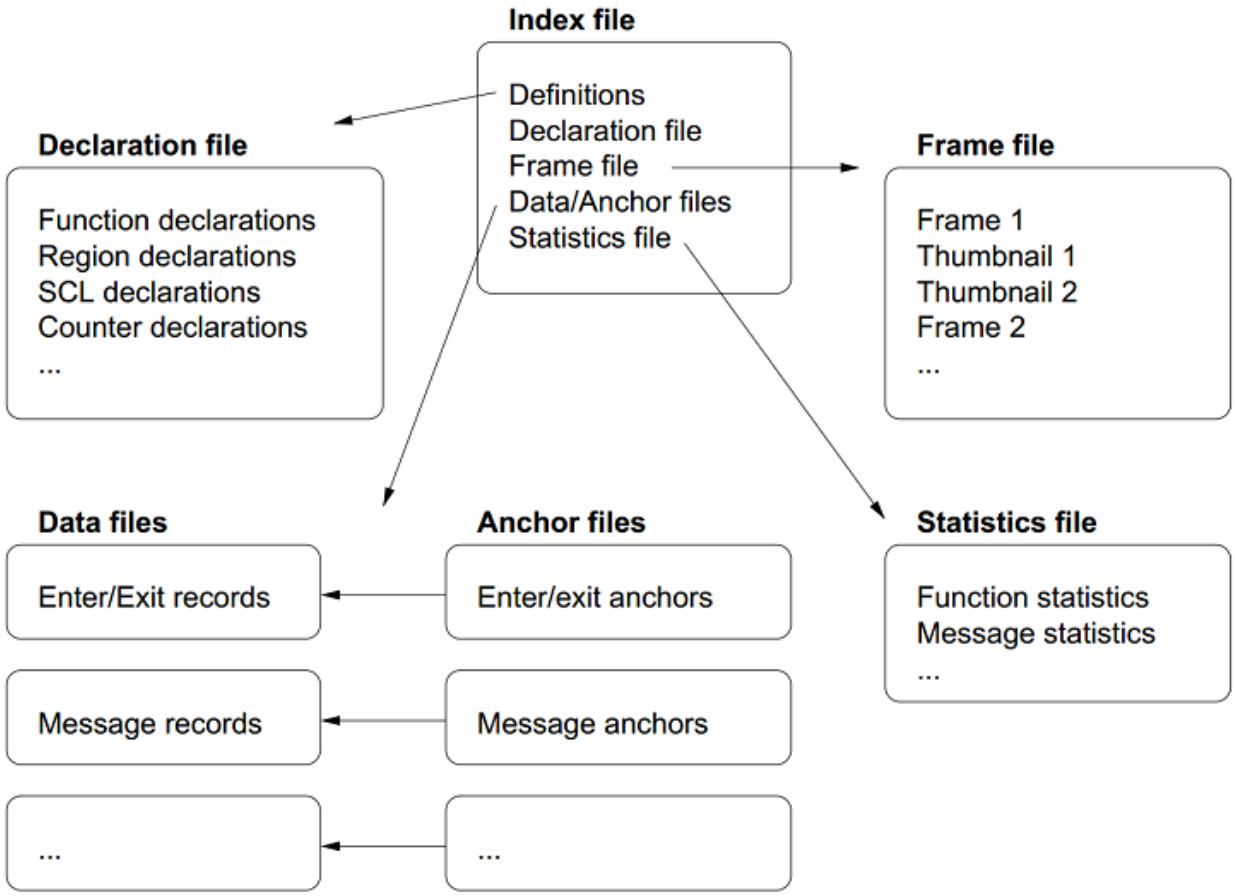
A structured tracefile consists of a number of files, which can be interpreted as one conceptual data set. See the approximate structure in the figure below. Depending on the organization of actual files, the following component files will be written:

- Index file `<trace>.stf`
- Record declaration file `<trace>.stf.dcl`
- Statistics file `<trace>.stf.sts`
- Message file `<trace>.stf.msg`
- Collective operation file `<trace>.stf.cop`

- Process file(s) *<trace>*.stf.*.*<index>* (where * is one of the symbols, *f*, *i*, *s*, *c*, *r*, or *x*)
- For the above three kinds of files, one anchor file each with the extension .anc

<trace> is the tracefile name, which is determined automatically or set in the LOGFILE-NAME configuration option.

STF Components



Additionally, five data files may be created for the given trace. These files are Summary Data files. They have common name <trace>.stf.sum.<suffix> (where suffix is one of fnc, cop, msg, cnt, or rep) and formally are not a part of the trace. You can use these files as additional input for Intel® Trace Analyzer. For details of Summary Data usage, see *Intel® Trace Analyzer User and Reference Guide*.

The records for routine entry/exit and counters are contained in the process files. The anchor files are used by Intel® Trace Analyzer to fast-forward within the record files; they can be deleted, but that may result in slower operation of Intel® Trace Analyzer.

Make sure that you use different names for traces from different runs; otherwise you will experience difficulties in identifying which process files belong to an index file, and which ones are left over from a previous run. To catch all component files, use the stftool with the --remove option to delete a STF file, or put the files into single-file STF format for transmission or archival with the stftool --convert option.

The number of actual process files will depend on the setting of the STF-USE-HW-STRUCTURE and STF-PROCS-PER-FILE configuration options described below.

See Also

stftool Utility

Single-File STF

Intel® Trace Collector can save the trace data in the single-file STF format. This format is selected by specifying the `LOGFILE-FORMAT STFSINGLE` configuration option, and it causes all the component files of an STF trace to be combined into one file with the extension `.single.stf`. The logical structure is preserved. The drawback of the single-file STF format is that no I/O parallelism can be exploited when writing the tracefile.
Reading it for analysis with Intel® Trace Analyzer is only marginally slower than the normal STF format, unless the operating system imposes a performance penalty on parallel read accesses to the same file.

See Also

Configuring Intel® Trace Collector

Configuring STF

To determine the file layout, you can use the following configuration options:

- `STF-USE-HW-STRUCTURE` will save the local events for all processes running on the same node into one process file
- `STF-PROCS-PER-FILE` *`<number>`* limits the number of processes whose events can be written in a single process file

All of these options are explained in more detail in the Configuration Reference section.

stftool Utility

stftool Utility

Synopsis

```
stftool <input file> <config options>
--help
--version
```

Description

The `stftool` utility program reads a structured trace file (STF) in normal or single-file format. It can perform various operations with this file:

- Extract all or a subset of the trace data (default)
- Convert the file format without modifying the content (`--convert`)
- List the components of the file (`--print-files`)
- Remove all components (`--remove`)
- Rename or move the file (`--move`)
- List statistics (`--print-statistics`)

The output and behavior of `stftool` is configured similarly to Intel® Trace Collector: with a configuration file, environment variables, and command-line options. The environment variable `VT_CONFIG` can be set to the name of an Intel® Trace Collector configuration file. If the file exists and is readable, then it is parsed first. Its settings are overridden with environment variables, which in turn are overridden by configuration options on the command line.
All configuration options can be specified on the command line by adding the prefix '`--`' and listing its arguments after the keyword. The output format is derived automatically from the suffix of the output file. You can write to `stdout` by using '`-`' as the filename; this defaults to writing ASCII VTF*.
These are examples of converting the entire file into different formats:

```
stftool example.stf --convert example.avt # ASCII
stftool example.stf --convert - # ASCII to stdout
stftool example.stf --convert - --logfile-format SINGLESTF | gzip -c
>example.single.stf.gz # gzipped single-file STF
```

Without the `--convert` switch one can extract certain parts, but only write VTF:

```
stftool example.stf --request 1s:5s --logfile-name example_1s5s.avt # extract
interval as ASCII
```

All options can be given as environment variables. The format of the configuration file and environment variables are described in more detail in the documentation in the Configuration Reference section.

stftool Utility Options

convert

Syntax: `--convert [<filename>]`
Default: `off`
Description: Converts the entire file into the file format specified with `--logfile-format` or the filename suffix. Options that normally select a subset of the trace data are ignored when this low-level conversion is done. Without this flag writing is restricted to ASCII format, while this flag can also be used to copy any kind of STF trace.

delete-raw-data

Syntax: `--delete-raw-data`
Default: `off`
Description: Sub-option to `--merge`. Deletes or removes the given raw trace after merging.

dump

Syntax: `--dump`
Default: `off`
Description: A shortcut for `--logfile-name -` and `--logfile-format ASCII`, that is, it prints the trace data to `stdout`.

extended-vtf

Syntax: `--extended-vtf`
Default: `off` in Intel Trace Collector, `on` in `stftool`
Description: Several events can only be stored in `STF`, but not in `VTF`. Intel® Trace Collector libraries default to writing valid VTF trace files and thus skip these events. This option enables writing of non-standard VTF records in ASCII mode that Intel® Trace Analyzer would complain about. In the `stftool` the default is to write these extended records, because the output is more likely to be parsed by scripts rather than by the Intel Trace Analyzer.

logfile-format

Syntax: `--logfile-format [ASCII|STF|STFSINGLE|SINGLESTF]`
Default: `STF`
Description: Specifies the format of the tracefile. ASCII is the traditional Vampir* file format where all trace data is written into one file. It is human-readable.

The Structured Trace File (STF) is a binary format which supports storage of trace data in several files and allows Intel® Trace Analyzer to analyze the data without loading all of it, so it is more scalable. Writing it is only supported by Intel® Trace Collector.

One trace in STF format consists of several different files which are referenced by one index file (`.stf`). The advantage is that different processes can write their data in parallel (see STF-PROCS-PER-FILE, STF-USE-HW-STRUCTURE). `SINGLESTF` rolls all of these files into one (`.single.stf`), which can be read without unpacking them again. However, this format does not support distributed writing, so for large program runs with many processes the generic STF format is better.

logfile-name

Syntax: `--logfile-name` *<file name>*
Description: Specifies the name for the tracefile containing all the trace data. Can be an absolute or relative pathname; in the latter case, it is interpreted relative to the log prefix (if set) or the current working directory of the process writing it.

If unspecified, then the name is the name of the program plus `.avt` for ASCII, `.stf` for STF and `.single.stf` for single STF tracefiles. If one of these suffices is used, then they also determine the logfile format, unless the format is specified explicitly.
In the `stftool` the name has to be specified explicitly, either by using this option or as argument of the `--convert` or `--move` switch.

matched-vtf

Syntax: `--matched-vtf`
Default: `off`
Description: When converting from `STF` to `ASCII-VTF` communication records are usually split up into conventional `VTF` records. If this option is enabled, an extended format is written, which puts all information about the communication into a single line.

merge

Syntax: `--merge [<merged trace name>]`
Default: `off`
Description: Merges the given raw trace. When you use the `--merge` option with the `--delete-raw-data` option, such configuration deletes the given raw trace after merging. When you use `--merge` option with the `--sumdata` option, such configuration creates additional Summary Data files for the given unmerged trace.

move

Syntax: `--move [<file/dirname>]`
Default: `off`
Description: Moves the given file without otherwise changing it. The target can be a directory.

print-errors

Syntax: `--print-errors`
Default: `off`
Description: Prints the errors that were found in the application.

print-files

Syntax: `--print-files`
Default: `off`
Description: Lists all components that are part of the given STF file, including their size. This is similar to `ls -l`, but also works with single-file STF.

print-reports

Syntax: `--print-reports`
Default: `off`
Description: Prints the Message Checker reports of the input file to `stdout`.

print-statistics

Syntax: `--print-statistics`
Default: `off`
Description: Prints the precomputed statistics of the input file to `stdout`.

print-threads

Syntax: `--print-threads`
Default: `off`
Description: Prints information about each native thread that was encountered by the Intel® Trace Collector when generating the trace.

remove

Syntax: `--remove`
Default: `off`
Description: Removes the given file and all of its components.

request

Syntax: `--request <type>, <thread triplets>, <categories>, <window>`
Description: Restricts the data written into the new trace to the one that matches the arguments. If a window is given (in the form `<timespec>:<timespec>` with at least one unit descriptor), data is restricted to this time interval. It has the usual format of a time value, with one exception: the unit for seconds **s** is required to distinguish it from a thread triplet; in other words, use `10s` instead of just `10`. The `<type>` can be any kind of string in single or double quotation marks, but it has to identify uniquely the kind of data. Valid `<categories>` are `FUNCTIONS`, `SCOPES`, `FILEIO`, `COUNTERS`, `MESSAGES`, `COLLOPS`, `ERRORS` and `REQUESTS`.
All of the arguments are optional and default to all threads, all categories and the whole time interval. They can be separated by commas or spaces and it is possible to mix them as desired. This option can be used more than once and then data matching any request is written.

sumdata

Syntax: `--sumdata <output trace name>`
Default: `off`
Description: Forces creation of additional Summary Data files for the given trace.

You can use the `--sumdat` option with or without `--merge` option. Thus, there can be the following three scenarios:

1. `--merge <output trace>` Merges the given unmerged trace and creates output merged trace.
2. `--sumdata <output trace>` Creates Summary Data files for the given merged trace.

Note

In this scenario, only Summary Data files is created. No output trace is generated.

---

3. `--merge --sumdata <output trace>` Merges the given unmerged trace; creates output merged trace and the Summary Data files for this output trace.

ticks

Syntax: `--ticks`
Default: `off`
Description: Setting this option to `on` lets `stftool` interpret all timestamps as ticks (rather than seconds, milliseconds and so on). Given time values are converted into seconds and then truncated (floor). The clock ticks are based on the nominal clock period specified by the `CLKPERIOD` header, just as the time stamps printed by the `stftool` for events.

verbose

Syntax: `--verbose [on|off|<level>]`
Default: `on`
Description: Enables or disables additional output on `stderr`. `<level>` is a positive number, with larger numbers enabling more output:

1. `0` (= `off`) disables all output
2. `1` (= `on`) enables only one final message about generating the result
3. `2` enables general progress reports by the main process
4. `3` enables detailed progress reports by the main process
5. `4` the same, but for all processes (if multiple processes are used at all)

Levels higher than `2` may contain output that only makes sense to the developers of Intel® Trace Collector.

Expanded ASCII output of STF Files

Synopsis

```
xstftool <STF file> [stftool options]
```

Valid options are those that work together with `stftool --dump`, the most important ones being:

- `--request`: extract a subset of the data
- `--matched-vtf`: put information about complex events like messages and collective operations into one line

Description

The `xstftool` is a simple wrapper around the `stftool` and the `expandvtlog.pl` Perl* script which tells the `stftool` to dump a given Structured Trace Format (STF) file in ASCII format and uses the script as a filter to make the output more readable.
It is intended to be used for doing custom analysis of trace data with scripts that parse the output to extract information not provided by the existing tools, or for situations where a few shell commands provide the desired information more quickly than a graphical analysis tool.

Output

The output has the format of the ASCII Vampir* Trace Format (VTF), but entities like function names are not represented by integer numbers that cannot be understood without remembering their definitions, but rather inserted into each record. The CPU numbers that encode process and thread ranks resp. groups are also expanded.

Examples

The following examples compare the output of `stftool --dump` with the expanded output of `xstftool`:

- definition of a group

```
DEFGROUP 2147942402 "All_Processes" NMEMBS 2 2147483649 2147483650
DEFGROUP All_Processes NMEMBS 2 "Process_0" "Process_2"
```

- a counter sample on thread 2 of the first process

```
8629175798 SAMP CPU 131074 DEF 6 UINT 8 3897889661
8629175798 SAMP CPU 2:1 DEF "PERF_DATA:PAPI_TOT_INS" UINT 8 3897889661
```

Time Stamping

Time Stamping

Intel® Trace Collector assigns a local time stamp to each event it records. A time stamp consists of two parts which together guarantee that each time stamp is unique:

Clock Tick counts how often the timing source incremented since the start of the run.

Event Counter is incremented for each time stamp which happens to have the same clock tick as the previous time stamp. In the unlikely situation that the event counter overflows, Intel® Trace Collector artificially increments the clock tick. When running Intel® Trace Collector with `VERBOSE > 2`, it will print the maximum number of events on the same clock tick during the whole application run. A non-zero number implies that the clock resolution was too low to distinguish different events.
Both counters are stored in a 64-bit unsigned integer with the event counter in the low-order bits. Legacy applications can still convert time stamps as found in a trace file to seconds by multiplying the time stamp with the nominal clock period defined in the trace file header: if the event counter is zero, this will not incur any error at all. Otherwise the error is most likely still very small. The accurate solution however is to shift the time stamp by the amount specified as event bits in the trace header (and thus removing the event counter), then multiplying with the nominal clock period and 2 to the power of event bits.

Intel® Trace Collector uses 51 bits for clock ticks, which is large enough to count 2^{51}ns, which equals to more than 26 days before the counter overflows. At the same time with a clock of only ms resolution, you can distinguish 8192 different events with the same clock tick, which are events with duration of 0.1 µs.

Before writing the events into the global trace file, local time stamps are replaced with global ones by modifying their clock tick. A situation where time stamps with different local clock ticks fall on the same global clock tick is avoided by ensuring that global clock ticks are always larger than local ones. The nominal clock period in the trace file is chosen so that it is sufficiently small to capture the offsets between nodes as well as the clock correction: both leads to fractions of the real clock period and rounding errors would be incurred when storing the trace with the real clock period. The real clock period might be hard to figure out exactly anyway. Also, the clock ticks are scaled so that the whole run takes exactly as long as determined with `gettimeofday()` on the master process.

Clock Synchronization

By default, Intel® Trace Collector synchronizes the different clocks at the start and at the end of a program run by exchanging messages in a fashion similar to the Network Time Protocol (NTP): one process is treated as the master and its clock becomes the global clock of the whole application run. During clock synchronization, the master process receives a message from a child process and replies by sending its current time stamp. The child process then stores that time stamp together with its own local send and receive time stamps. One message is exchanged with each child, then the cycles starts again with the first child until `SYNC-MAX-MESSAGES` have been exchanged between master and each child or the total duration of the synchronization exceeds `SYNC-MAX-DURATION`.
Intel® Trace Collector can handle timers which are already synchronized among all process on a node (`SYNCED-HOST`) and then only does the message exchange between nodes. If the clock is even synchronized across the whole cluster (`SYNCED-CLUSTER`), then no synchronization is done by Intel® Trace Collector at all.
The gathered data of one message exchange session is used by the child processes to calculate the offset between its clock and the master clock: it is assumed that the duration of messages with equal size is equally fast in both directions, so that the average of local send and receive time coincides

with the master time stamp in the middle of the message exchange. To reduce the noise, the 10% message pairs with the highest local round-trip time are ignored because those are the ones which most likely suffered from not running either process in time to react in a timely fashion or other external delays.

With clock synchronization at the start and the end, Intel® Trace Collector clock correction uses a linear transformation; that is a scaling local clock ticks and shifting them, which is calculated by linear regression of all available sample data. If the application also calls `VT_timesync()` during the run, then clock correction is done with a piece-wise interpolation: the data of each message exchange session is condensed into one pair of local and master time by averaging all data points, then a constrained spline is constructed which goes through all of the condensed points and has a contiguous first derivative at each of these joints.

VT_timesync

```
int VT_timesync(void)
```

Description

Gathers data needed for clock synchronization.

This is a collective call, so all processes which were started together must call this function or it will block.

This function does not work if processes were spawned dynamically.

Fortran

```
VTTIMESYNC(ierr)
```

Choosing a Timer

Choosing a Timer

A good timer has the following properties:

- High resolution (one order of magnitude higher than the resolution of the events that are to be traced)
- Low overhead
- Linearly increasing values for a long period of time (at least for the duration of a program run); in particular it should not jump forwards or backwards

Intel® Trace Collector supports several different timers. Because the quality of these timers depends on factors which are hard to predict (like specific OS bugs, available hardware and so on), you can run a test program if you want to find answers to the following questions:

- What is the resolution of a timer?
- What is its overhead?
- How well does clock synchronization work with the default linear transformation?
- If it does not work well, how often does the application have to synchronize to achieve good non-linear interpolation?

To test the quality of each timer, link the `timerperformance.c` sample program (available at the online samples resource). The `makefile` already has a target `vttimertest` (linked against `libVT` and MPI) and for `timertestcs` (linked against `libVTcs` and no MPI). Use the MPI version if you have MPI, because `libVT` supports all the normal timers from `libVTcs` plus `MPI_Wtime` and because only the MPI version can test whether the clock increases linearly by time-stamping message exchanges.
To get a list of supported timers, run with the configuration option `TIMER` set to `LIST`. This can be done easily by setting the `VT_TIMER` environment variable. The subsections below have more information about possible choices, but not all of them may be available on each system.
To test an individual timer, run the binary with `TIMER` set to the name of the timer to be tested. It will repeatedly acquire time stamps and then for each process (`vttimertest`) or the current machine (`timertestcs`) print a histogram of the clock increments observed. A good timer has most increments close or equal to the minimum clock increment that it can measure. Bad clocks have a very high minimum clock increment (a bad resolution) or only occasionally increment by a smaller amount.
Here is the output of `timertestcs` one a machine with a good `gettimeofday()` clock:

```
bash$ VT_TIMER=gettimeofday ./timertestcs
performance: 2323603 calls in 5.000s wall clock time = 2.152us/call =
464720 calls/s
measured clock period/frequency vs. nominal:
1.000us/1.000MHz vs. 1.000us/1.000MHz
overhead for sampling loop: 758957 clock ticks (= 758.958ms)
for 10000000 iterations = 0 ticks/iteration
average increase: 2 clock ticks = 2.244us = 0.446MHz
median increase: 2 clock ticks = 2.000us = 0.500MHz
< 0 ticks = 0.00s : 0
< 1 ticks = 1.00us: 0
>= 1 ticks = 1.00us: ################### 2261760
>= 501 ticks = 501.00us: 1
>= 1001 ticks = 1.00ms: 0
...
```

The additional information at the top starts with the performance (and thus overhead) of the timer. The next line compares the measured clock period (calculated as elapsed wall clock time divided by clock ticks in the measurement interval) against the one that the timer is said to have; for `gettimeofday()` this is not useful, but for example CPU cycle counters (details below) there might be differences. Similarly, the overhead for an empty loop with a dummy function call is only relevant for a timer like CPU cycle counters with a very high precision. For that counter however the overhead caused by the loop is considerable, so during the measurement of the clock increments Intel® Trace Collector subtracts the loop overhead.
Here is an example with the CPU cycle counter as timer:

```
bash$ VT_TIMER=CPU ./timertestcs
performance: 3432873 calls in 5.000s wall clock time = 1.457us/call =
```

```
686535 calls/s
measured clock period/frequency vs. nominal:
0.418ns/2392.218MHz vs. 0.418ns/2392.356MHz
overhead for sampling loop: 1913800372 clock ticks (= 800.011ms)
for 10000000 iterations = 191 ticks/iteration
average increase: 3476 clock ticks = 1.453us = 0.688MHz
median increase: 3473 clock ticks = 1.452us = 0.689MHz
< 0 ticks = 0.00s : 0
< 1 ticks = 0.42ns: 0
>= 1 ticks = 0.42ns: 0
>= 501 ticks = 209.43ns: 0
>= 1001 ticks = 418.44ns: 0
>= 1501 ticks = 627.45ns: 0
>= 2001 ticks = 836.46ns: 0
>= 2501 ticks = 1.05us: 0
>= 3001 ticks = 1.25us: ################### 3282286
>= 3501 ticks = 1.46us: 587
>= 4001 ticks = 1.67us: 8
>= 4501 ticks = 1.88us: 1
>= 5001 ticks = 2.09us: 869
```

Testing whether the timer increases linearly is more difficult. It is done by comparing the send and receive time stamps of ping-pong message exchanges between two processes after Intel® Trace Collector has applied its time synchronization algorithm to them: the algorithm will scale and shift the time stamps based on the assumption that data transfer in both directions is equally fast. So if the synchronization works, the average difference between the duration of messages in one direction minus the duration of the replies has to be zero. The visualization of the trace `timertest.stf` should show equilateral triangles.
If the timer increases linearly, then one set of correction parameters applies to the whole trace. If it does not, then clock synchronization might be good in one part of the trace and bad in another or even more obvious, be biased towards one process in one part with a positive difference and biased towards the other in another part with a negative difference. In either case tweaking the correction parameters would fix the time stamps of one data exchange, but just worsen the time stamps of another.

When running the MPI `vttimertest` with two or more processes it will do a short burst of data exchanges between each pair of processes, then sleep for 10 seconds. This cycle is repeated for a total runtime of 30 seconds. This total duration can be modified by giving the number of seconds as command line parameter. Another argument on the command line also overrides the duration of the sleep. After `MPI_Finalize()` the main process will read the resulting trace file and print statistics about the message exchanges: for each pair of processes and each burst of message exchanges, the average offset between the two processes is given. Ideally these offsets will be close to zero, so at the end the pair of processes with the highest absolute clock offset between sender and receiver will be printed:

```

maximum clock offset during run:
```

```
1 <-> 2 374.738ns (latency 6.752us)
to produce graph showing trace timing, run: gnuplot timertest.gnuplot
```

If the value is much smaller than the message latency, then clock correction worked well throughout the whole program run and can be trusted to accurately time individual messages.

Running the test program for a short interval is useful to test whether the NTP-like message exchange works in principle, but to get realistic results you have to run the test for several minutes. If a timer is used which is synchronized within a node, then you should run with one process per node because Intel® Trace Collector would use the same clock correction for all processes on the same node anyway. Running with multiple processes per node in this case would only be useful to check whether the timer really is synchronized within the node.

To better understand the behavior of large runs, several data files and one command file for `gnuplot` are generated. Running `gnuplot` as indicated above will produce several graphs:

Graph	Description
Application Run	Connects the offsets derived from the application's message exchanges with straight lines: it shows whether the deviation from the expected zero offset is linear or not; it can be very noisy because outliers are not removed
Clock Transformation	Shows the clock samples that Intel® Trace Collector itself took at the application start, end and in `VT_timesync()` and what the transformation from local clock ticks to global clock ticks looks like
Interpolation Error	Compares a simple linear interpolation of Intel® Trace Collector's sample data against the non-linear constrained spline interpolation: at each sample point, the absolute delta between measured time offset and the corresponding interpolated value is shown above the x-axis (for linear interpolation) and below (for splines)
Raw Clock Samples	For the first three message exchanges of each process, the raw clock samples taken by Intel® Trace Collector are shown in two different ways: all samples and just those actually used by Intel® Trace Collector after removing outliers. In these displays the height of the error bars corresponds to the round-trip time of each sample measured on the master. If communication works reliably, most samples should have the same round-trip time.

The graphs use different coordinate systems: the first one uses global time for both axis; the latter two have local time on the x-axis and a delta in global time on the y-axis. Thus although the same error will show up in all of them, in one graph it will appear as a deviation for example below the x-axis and in the other above it.

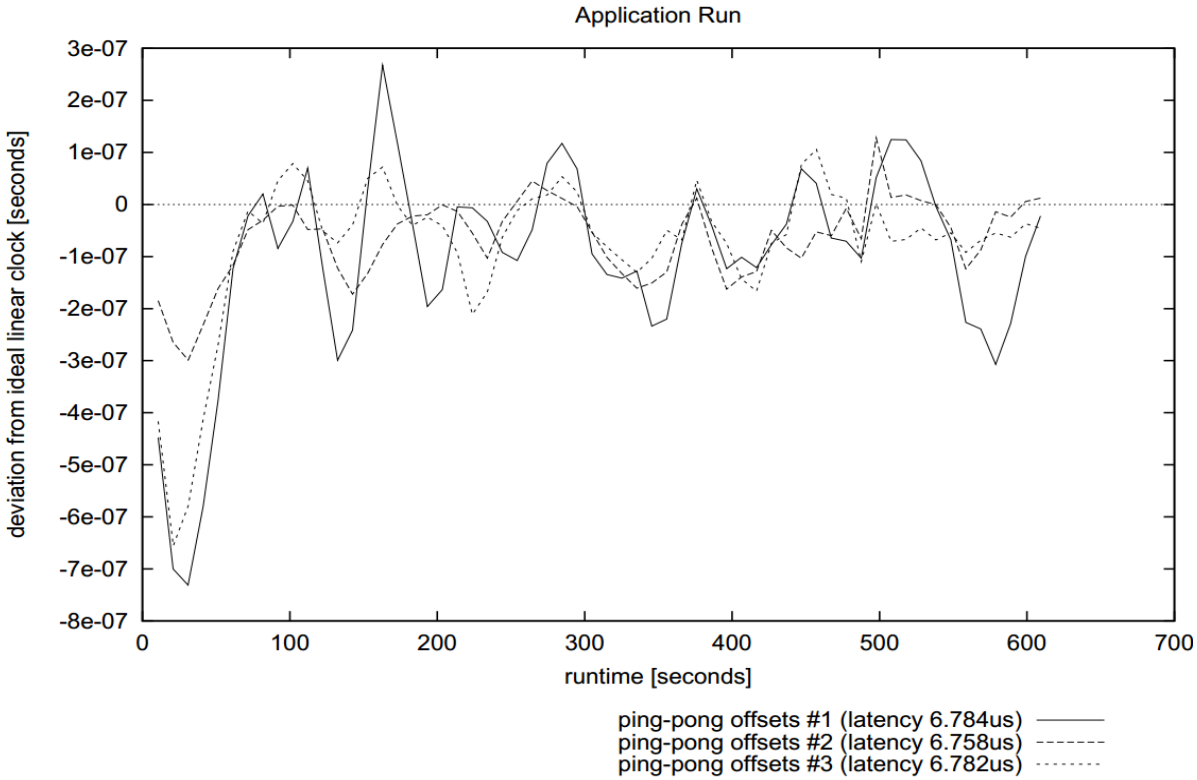
Also, the latter two graphs are only useful if Intel® Trace Collector really uses non-linear interpolation which is not the case if all intermediate clock samples are skipped: although the test program causes a clock synchronization before each message exchange by calling `VT_timesync()`, at the same time it tells Intel® Trace Collector to not use those results and thus simulates a default application run where synchronization is only done at the start and end.
This can be overridden by setting the `TIMER-SKIP` configuration option or `VT_TIMER_SKIP` environment variable to a small integer value: it chooses how often the result of a `VT_timesync()`

is ignored before using a sample for non-linear clock correction. The skipped samples serve as checks that the interpolation is sound.
In the following figures the test program was run using the CPU timer source, with a total runtime of 10 minutes and skipping 5 samples:

```
bash$ VT_TIMER_SKIP=5 VT_TIMER=CPU mpirun -np 4 timertest 600

...

[0 (node0)] performance: 115750510 calls in 5.000s wall

clock time = 43.197ns/call = 23149574 calls/s

...

0. recording messages 0 <-> 1...
0. recording messages 0 <-> 2...
0. recording messages 0 <-> 3...
0. recording messages 1 <-> 2...
0. recording messages 1 <-> 3...
0. recording messages 2 <-> 3...
1. recording messages 0 <-> 1...
...
maximum clock offset during run:
0 <-> 1 -1.031us (latency 6.756us)
```

The application run in Figure 5.1 below shows that in general Intel® Trace Collector managed to keep the test results inside a range of plus-minus 1µs although it did not use all the information collected with `VT_timesync()`. The clock transformation function in Figure 5.2 is non-linear for all three child processes and interpolates the intermediate samples well. Using a linear interpolation between start and end would have led to deviations in the middle of more than 16 µs. Also, the constrained spline interpolation is superior compared to a simple linear interpolation between the sample points (Figure 5.3).

Figure 5.1 CPU Timer: Application Run with Non-linear Clock Correction

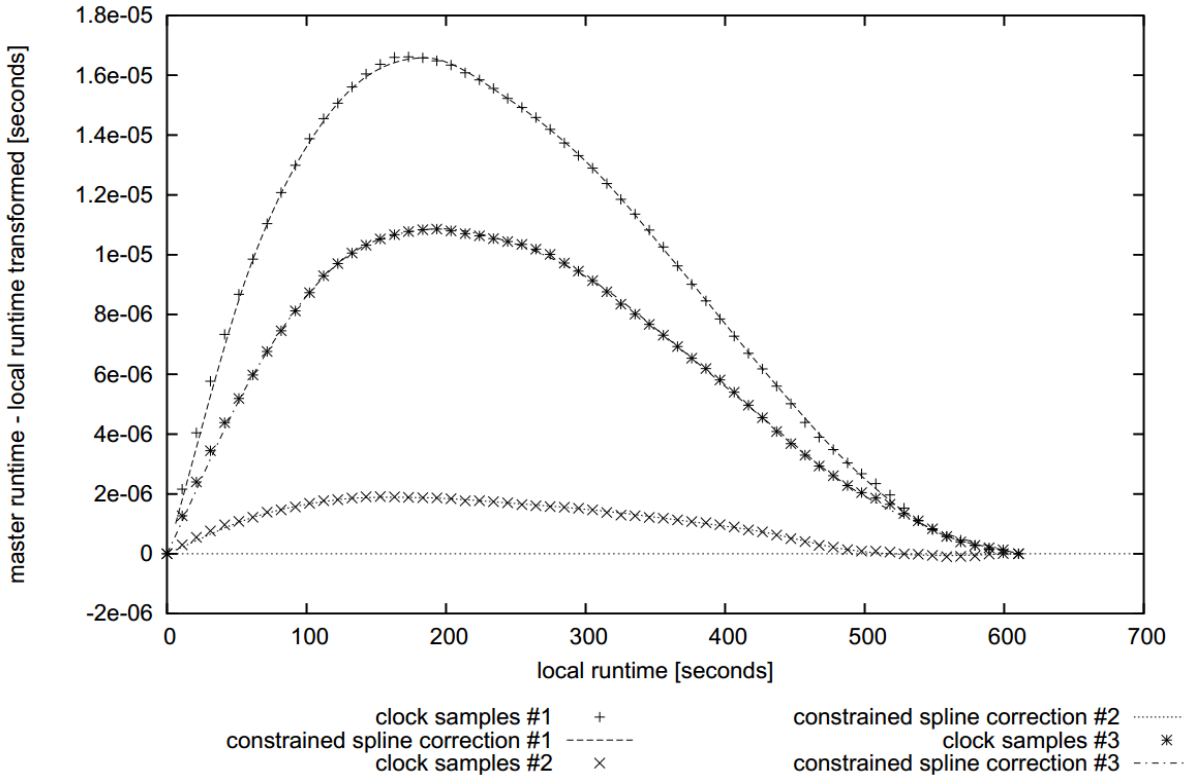


gettimeofday/_ftime

`gettimeofday` is the default timer on Linux* OS with `_ftime` being the equivalent on Microsoft* Windows* OS. Its API limits the clock resolution to 1µs, but depending on which timer the OS actually uses the clock resolution may be much lower (`_ftime` usually shows a resolution of only 1 millisecond). It is implemented as a system call; therefore it has a higher overhead than other timers. In theory the advantage of this call is that the OS can make better use of the available hardware, so this timer should be stable over time even if NTP is not running. However, Figure 5.4 shows that in practice at least on that system quite a high deviation between different nodes occurred during the run.

If NTP is running, then the clock of each node might be modified by the NTP daemon in a non-linear way. NTP should not cause jumps, only accelerate or slow down the system time.

Figure 5.2 CPU Timer: clock Transformation and the Sample Points It Is Based on



However, even decreasing system time stamps have been observed on some systems. This may or may not have been due to NTP.

Due to the clock synchronization at runtime enabling NTP did not make the result worse than it is without NTP (Figure 5.5). However, NTP alone without the additional intermediate synchronization would have led to deviations of nearly 70 µs.

So the recommendation is to enable NTP, but intermediate clock synchronization by Intel® Trace Collector is still needed to achieve good results.

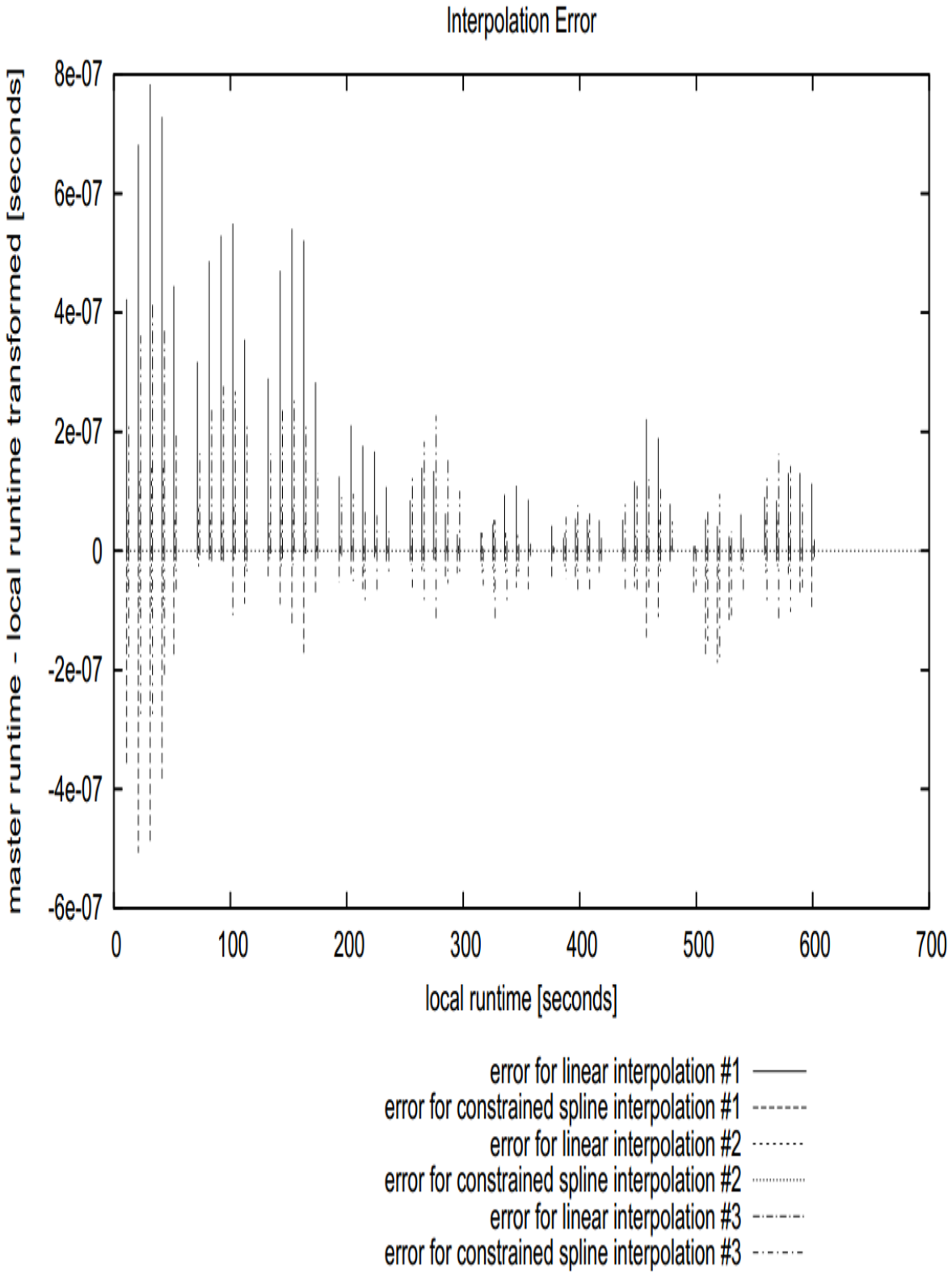
QueryPerformanceCounter

On Microsoft* Windows* OS, Intel® Trace Collector uses `QueryPerformanceCounter` as the default timer. As a system function it comes with the same side-effects as `_ftime` but has a higher resolution of around 1 µs.

CPU Cycle Counter

This is a high-resolution counter inside the CPU which counts CPU cycles. This counter is called Timer Stamp Counter (TSC) on x86/Intel®64 architectures. It can be read through an assembler instruction, so the overhead is much lower than `gettimeofday()`. On the other hand, these counters were never meant to measure long time intervals, so the clock speed also varies a lot, as seen earlier in Figure 5.2.

Figure 5.3 CPU Timer: Error with Linear (above x-axis) and Non-linear Interpolation (below)



Additional complications are:

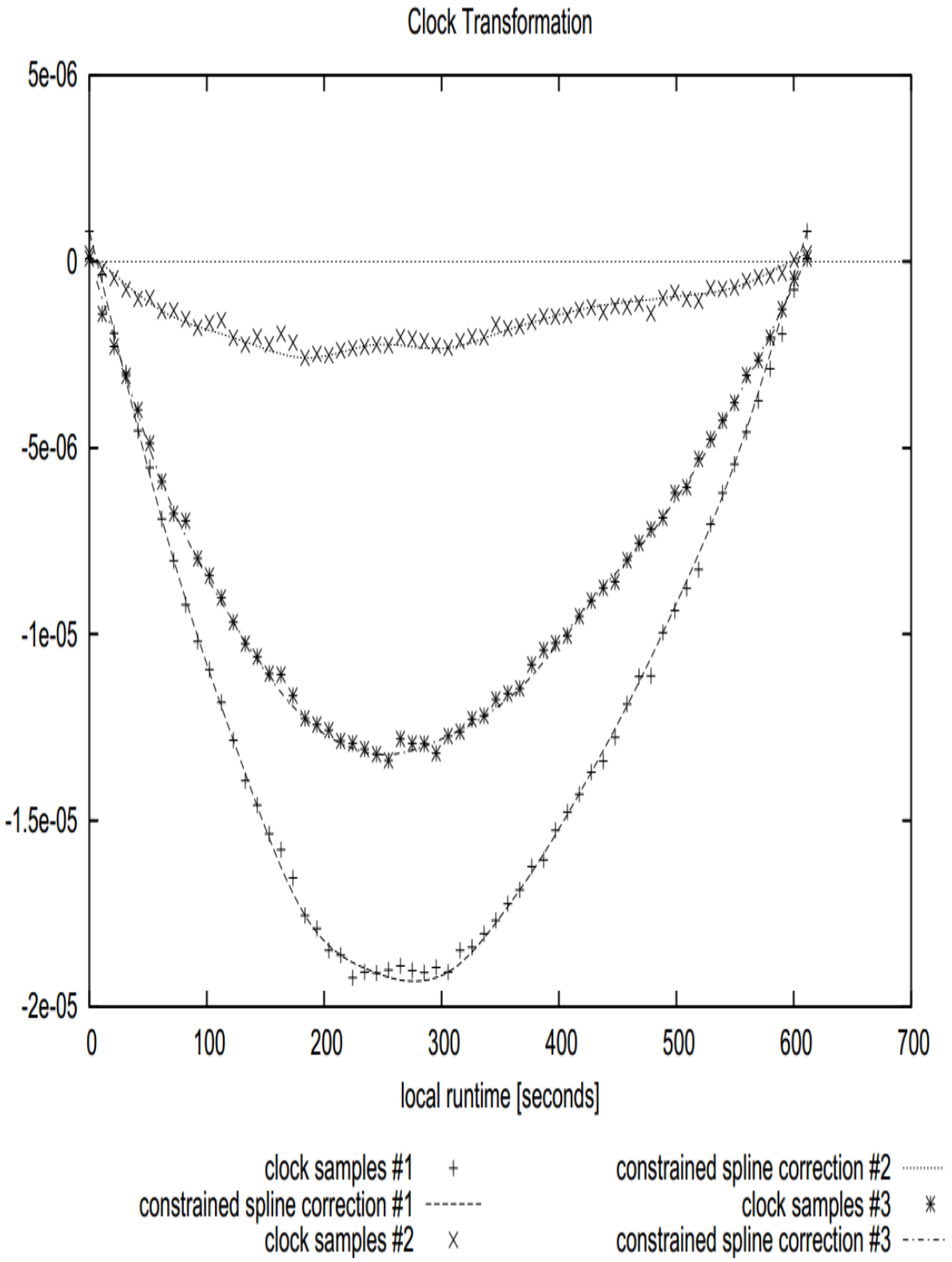
Multi-CPU machines: the counter is CPU-specific, so if threads migrate from one CPU to another the clock that Intel® Trace Collector reads might jump arbitrarily. Intel® Trace Collector cannot compensate this as it would have to identify the current CPU and read the register in one atomic operation, which cannot be done from user space without considerable overhead.

CPU cycle counters might still be useful on multi-CPU systems: Linux* OS tries to set the registers of all CPUs to the same value when it boots. If all CPUs receive their clock pulse from the same source their counters do not drift apart later on and it does not matter on which CPU a thread reads the CPU register, the value will be the same one each.

This problem could be addressed by locking threads onto a specific CPU, but that could have an adverse effect on application performance and thus is not supported by Intel® Trace Collector itself. If done by the application or some other component, then care has to be taken that all threads in a process run on the same CPU, including those created by Intel® Trace Collector itself. If the application already is single-threaded, then the additional Intel® Trace Collector threads could be disabled to avoid this complication.

Frequency scaling: power-saving mode might lead to a change in the frequency of the cycle count register during the run and thus a non-linear clock drift. Machines meant for HPC probably do not support frequency scaling or will not enter power-saving mode. Even then, on Intel CPUs, TSC often continues to run at the original frequency.

Figure 5.4 getimeofday() without NTP



See Also

MEM-FLUSHBLOCKS
Recording OS Counters

Normalized CPU Cycle Counter

The CPU timer described in CPU Cycle Counter is applicable for homogenous systems only. Specifically, the CPU frequency should match across the systems.

For heterogeneous systems with different CPU frequencies, a special *Normalized CPU timer* (`VT_TIMER=CPU_Norm`) can be used. This timer is based on the *Timer Stamp Counter* (TSC) CPU ticks as well as the original CPU timer (`VT_TIMER=CPU`). The normalized timer converts the local CPU ticks into microseconds on the fly to allow usage of TSC on heterogeneous systems.

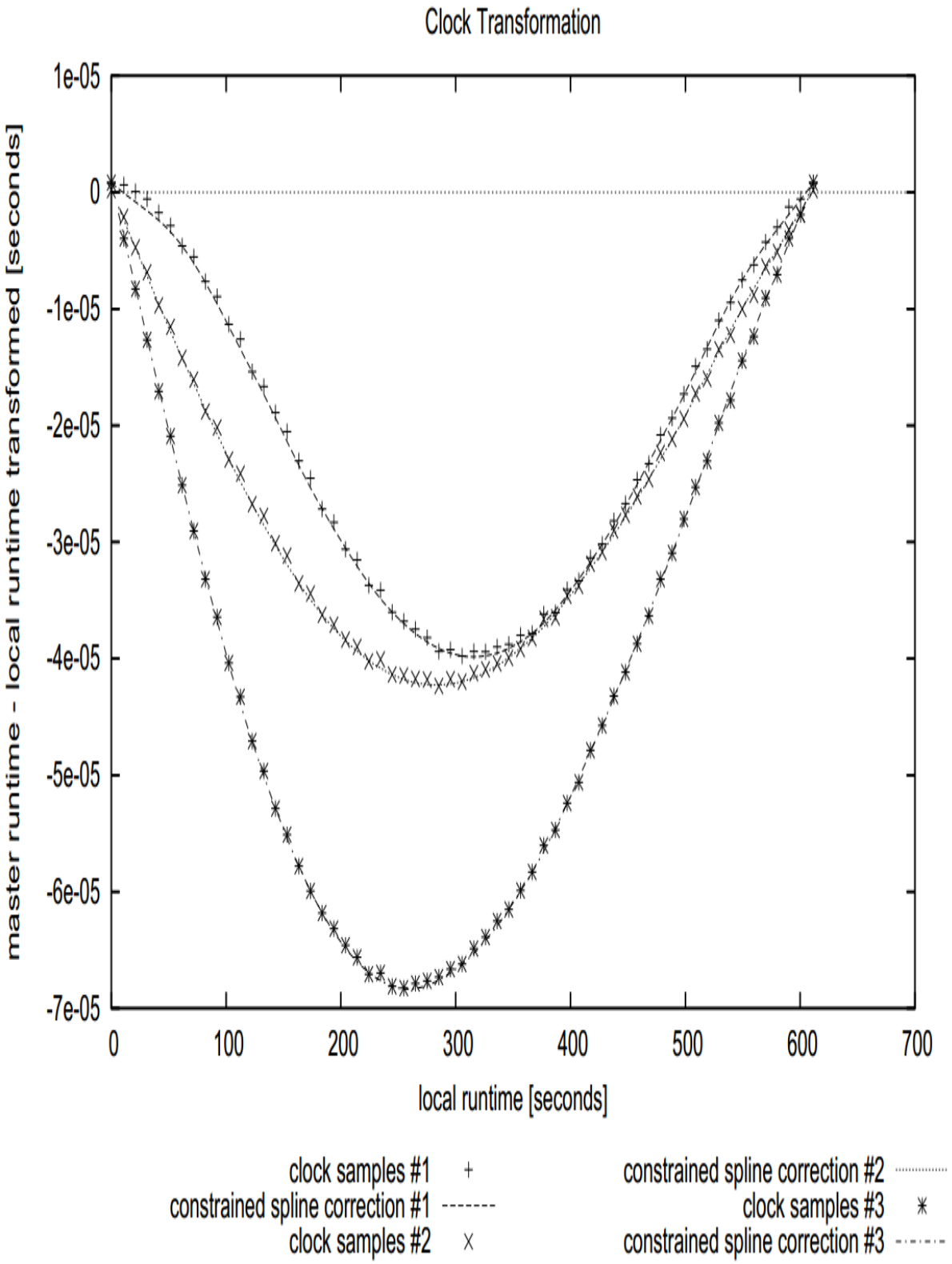
MPI_Wtime()

This timer is provided by the MPI implementation. In general this is simply a wrapper around `gettimeofday()` and then using it instead of `gettimeofday()` only has disadvantages: with `gettimeofday()` Intel® Trace Collector knows that processes running on the same node share the same clock and thus does not synchronize between them. The same information cannot be obtained through the MPI API and thus Intel® Trace Collector is conservative and assumes that clock synchronization is needed. This can be overridden with the `SYNCED-HOST` configuration option. Another disadvantage is increased overhead and potentially implementation errors in MPI.
If the MPI has access to a better timer source (for example a global clock in the underlying communication hardware), then using this timer would be advantageous.

High Precision Event Timers

This is a hardware timer source designed by Intel as replacement for the real time clock (RTC) hardware commonly found in PC boards. Availability and support for it in BIOS and OS is still very limited, therefore Intel® Trace Collector does not support it yet.

Figure 5.5 getimeofday() with NTP



POSIX* clock_gettime

This is another API specified by the Single Unix Specification and POSIX*. It offers a monotonic system clock which is not affected (for good or bad) by NTP, but the current implementation in Linux*/`glibc` does not provide better timing through this API than through `gettimeofday()`. Intel® Trace Collector does not support this API.

Secure Loading of Dynamic Link Libraries* on Windows* OS

To improve security protections on Microsoft Windows* OS, Intel® Trace Collector provides the enhanced security options for the loading of Dynamic-Link Libraries*. You can enable the secure DLL loading mode, as well as define a set of directories in which the library will attempt to locate an external DLL.

The security options are placed in the `HKEY_LOCAL_MACHINE\Software\Intel\ITAC` protected Windows* registry key. The location prevents the options from being changed with non-administrative privileges.

SecureDynamicLibraryLoading

Select the secure DLL loading mode.

Syntax

`SecureDynamicLibraryLoading=`*`<value>`*

Arguments

`<value>`	Binary indicator
`enable | yes | on | 1`	Enable the secure DLL loading mode
`disable | no | off | 0`	Disable the secure DLL loading mode. This is the default value

Description

Use `HKEY_LOCAL_MACHINE\Software\Intel\ITAC` registry key to define the `SecureDynamicLibraryLoading` registry entry. Set this entry to enable the secure DLL loading mode.

VT_MPI_DLL and VT_FMPI_DLL

Specify the MPI library to be used in the secure DLL loading mode.

Syntax

`VT_MPI_DLL=<library>`
`VT_FMPI_DLL=<library>`

Arguments

<library>	Specify the name of the library to be loaded

Description

In the secure DLL loading mode, the library changes the default-defined set of directories to locate DLLs. Therefore, the current working directory and the directories that are listed in the `PATH` environment variable may be ignored. To select a specific MPI library to be loaded, define the `VT_MPI_DLL` and `VT_FMPI_DLL` entries of the `HKEY_LOCAL_MACHINE\Software\Intel\ITAC` registry key. Specify the full path to the MPI library.

Note

The `VT_MPI_DLL` and `VT_FMPI_DLL` environment variables have no effect in the secure DLL loading mode.

SecurePath

Specify a set of directories to locate an external DLL.

Syntax

`SecurePath=`*`<path>[;<path>[...]]`*

Arguments

<path>	Specify paths to directories. The paths must be separated with a semicolon *;*.

Description

Use `HKEY_LOCAL_MACHINE\Software\Intel\ITAC` registry key to define the `SecurePath` registry entry. Set this entry to specify a set of directories to locate loaded DLLs in the secure DLL loading mode. Use a safe set of directories instead of some publicly writable directories to avoid insecure library loading.

Note

Use this option if the static tracing library `VT*.lib` is linked into the executable or if the tracing library is unable to load a DLL in the secure DLL loading mode. The option has no effect if the secure DLL loading mode is turned off.

Appendix A Copyright and Licenses

The MPI datatype hash code was developed by Julien Langou and George Bosilca, University of Tennessee, and is used with permission under the following license:

[1] This is similar to the method described in "Collective Error Detection for MPI Collective Operations", Chris Falzone, Anthony Chan, Ewing Lusk, William Gropp, *http://www.mcs.anl.gov/~gropp/bib/papers/2005/collective-checking.pdf*

[2] "Hash functions for MPI datatypes", Julien Langou, George Bosilca, Graham Fagg, Jack Dongarra, *http://www.cs.utk.edu/~library/TechReports/2005/ut-cs-05-552.pdf*

Notices and Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary

>Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.