

Intel® Trace Collector

Reference Guide

Copyright © 1996–2010 Intel Corporation

All Rights Reserved

Document Number: 318119-004

Revision: 8.0

World Wide Web: <http://www.intel.com>

Revision History

Document Number	Revision Number	Description	Revision Date
318119-001	7.1	Initial draft	/10/12/2007
318119-002	7.2	Sections "Error Detection", "Tracing of Binaries and Binary Instrumentation", "Recording Hardware Performance Information", and "SUPPORTED DIRECTIVES" were updated.	/09/18/2009
318119-003	8.0 beta	Sections "Intended Audience" and "Conventions and Symbols" were added.	/11/03/2009
318119-004	8.0	Section "STF Components" was updated. Sections "Tracing shmem Programs with Intel® Trace Collector" and "Tracing MPI One-Sided Communication" were deleted.	/02/16/2010

Contents

1	Introduction	6
1.1	What is Intel® Trace Collector?	6
1.2	System Requirements and Supported Features	7
1.3	Multithreading	9
1.4	About this Manual	9
1.5	Intended Audience	10
1.6	Conventions and Symbols.....	10
2	Installation.....	11
3	How to Use Intel® Trace Collector.....	13
3.1	Tracing MPI Applications.....	13
3.1.1	Compiling MPI Programs with Intel® Trace Collector	13
3.1.2	Linking MPI Programs with Intel® Trace Collector	13
3.1.3	Running MPI Programs with Intel® Trace Collector.....	15
3.1.4	Examples	15
3.1.5	Troubleshooting	15
3.1.6	Handling of Communicator Names	15
3.1.7	Tracing of Failing MPI Applications	16
3.1.8	Tracing MPI File IO	17
3.2	Single-process Tracing	17
3.3	Compiler-driven Subroutine Instrumentation	18
3.3.1	Intel® Professional Edition Compilers 10.0 and later	18
3.3.2	gcc/g++	18
3.3.3	Folding	18
3.3.4	C++ Name Demangling	18
3.4	Tracing of Binaries and Binary Instrumentation.....	19
3.5	Multithreaded Tracing	25
3.6	Recording Statistical Information	25
3.7	Recording Source Location Information	28
3.7.1	Notes for IA32 and Intel® 64 architectures.....	28
3.8	Recording Hardware Performance Information	29
3.9	Recording OS Counters	30
3.10	Using the Dummy Libraries.....	31
3.11	Using the Shared Libraries.....	31
3.12	Tracing Library Calls	32
4	Correctness Checking	36
4.1	Overview	36
4.2	Usage.....	37
4.2.1	Correctness Checking of MPI Application	37
4.2.2	Running with Valgrind*	37
4.2.3	Configuration.....	38
4.2.4	Analyzing the Results.....	39
4.2.5	Debugger Integration	39
4.3	Error Detection	42
4.3.1	Supported Errors.....	42
4.3.2	How It works.....	43
5	Time Stamping	53
5.1	Clock Synchronization.....	53
5.2	Choosing a Timer	54
5.2.1	gettimeofday/_ftime	58
5.2.2	QueryPerformanceCounter.....	59
5.2.3	CPU Cycle Counter (TSC)	59
5.2.4	MPI_Wtime()	61

5.2.5	High Precision Event Timers (HPET)	61
5.2.6	POSIX* clock_gettime.....	62
6	Tracing of Distributed Applications.....	63
6.1	Design.....	63
6.2	Using VTserver	63
6.2.1	Initialize and Finalize	64
6.3	Running without VTserver	64
6.4	Spawning Processes	65
6.5	Tracing Events.....	65
6.6	Usage.....	65
6.7	Signals	66
6.8	Examples.....	66
7	Structured Tracefile Format	67
7.1	Introduction	67
7.2	STF Components.....	67
7.3	Single-File STF.....	68
7.4	Configuring STF	68
7.4.1	Structured Trace File Manipulation	69
7.4.2	Expanded ASCII output of STF Files	73
8	User-level Instrumentation	75
8.1	The Intel® Trace Collector API.....	75
8.2	Initialization, Termination and Control.....	77
8.3	Defining and Recording Source Locations.....	81
8.4	Defining and Recording Functions or Regions.....	84
8.4.1	New Interface.....	84
8.4.2	Old Interface	85
8.4.3	State Changes	86
8.5	Defining and Recording Overlapping Scopes	90
8.6	Defining Groups of Processes	91
8.7	Defining and Recording Counters.....	93
8.8	Recording Communication Events.....	96
8.9	Additional API Calls in libVTcs	100
8.10	C++ API.....	102
8.10.1	VT_FuncDef Class Reference.....	103
8.10.2	VT_SclDef Class Reference	104
8.10.3	VT_Function Class Reference	104
8.10.4	VT_Region Class Reference.....	106
9	Intel® Trace Collector Configuration	109
9.1	Configuring Intel® Trace Collector.....	109
9.2	Specifying Configuration Options	109
9.3	Configuration Format.....	109
9.4	Syntax of Parameters	110
9.4.1	Time Value.....	110
9.4.2	Boolean Value.....	110
9.4.3	Number of Bytes	110
9.5	Supported Directives	110
9.6	How to Use the Filtering Facility	125
9.7	The Protocol File	127
Appendix A	Copyright and Licenses	129

Disclaimer and Legal Notices

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document 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.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's Web Site.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See http://www.intel.com/products/processor_number for details.

Intel, Itanium, Pentium, VTune, and Xeon are trademarks of Intel Corporation in the U.S. and other countries.

* Other names and brands may be claimed as the property of others.

Copyright © 1996-2010, Intel Corporation. All rights reserved.

This product includes software developed by the University of California, Berkley and its contributors, and software derived from the Xerox Secure Hash Function. It includes software developed by the University of Tennessee, see appendix A for details. It includes libraries developed and © by SGI and others. They are licensed under the GNU Lesser General Public License (LGPL) or Runtime General. Their source code can be downloaded from <ftp://ftp.i kn.intel.com/pub/opensource>.

1 Introduction

1.1 What is Intel® Trace Collector?

Intel® Trace Collector for MPI applications produces tracefiles that can be analyzed with Intel® Trace Analyzer performance analysis tool. Some Intel Trace Collector versions are also able to trace non-MPI applications, like socket communication in distributed applications or plain serial programs. It was formerly known as Vampirtrace*.

In MPI, it records all calls to the MPI library and all transmitted messages, and allows arbitrary user-defined events to be recorded. Instrumentation can be switched on or off at runtime, and a powerful filtering mechanism helps to limit the amount of the generated trace data.

Intel Trace Collector is an add-on for existing MPI implementations; using it merely requires relinking the application with the Intel Trace Collector profiling library (see [Compiling MPI Programs with Intel® Trace Collector](#)). This will enable the tracing of all calls to MPI routines, as well as all explicit message-passing. On some platforms, calls to user-level subroutines and functions will also be recorded.

Shared libraries (see [Using the Shared Libraries](#)) and binary instrumentation (see [Tracing of Binaries and Binary Instrumentation](#)) allow to attach Intel Trace Collector without any explicit linking step.

To define and trace user-defined events, or to use the profiling control functions, calls to the Intel Trace Collector API (see [The Intel® Trace Collector API](#)) have to be inserted into the source code of the application. This implies a recompilation of all affected source modules.

A special dummy version of the profiling libraries containing empty definitions for all Intel Trace Collector API routines can be used to switch off tracing just by relinking (see [Using the Dummy Libraries](#)).

1.2 System Requirements and Supported Features

It is compatible with all other MPI implementations that use the same binary interface. If in doubt, lookup your hardware platform and MPI in the Intel® Trace Collector system requirements list at <http://www.intel.com/software/products/cluster/>.

If your combination is not listed, you can check compatibility yourself by compiling and running the `examples/mpiconstants.c` program with your MPI. If any value of the constants in the output differs from the ones given below, then this version of Intel Trace Collector will not work:

Table 1.1 Constant Values

	intel64-lin- impi32 ia32-lin-mpi32	intel64-lin- mpich	ia32-lin-mpich
<code>sizeof(MPI_Datatype)</code>	4	4	4
<code>sizeof(MPI_Comm)</code>	4	4	4
<code>sizeof(MPI_Request)</code>	4	8	4
<code>MPI_CHAR</code>	1275068673	1	1
<code>MPI_BYTE</code>	1275068685	3	3
<code>MPI_SHORT</code>	1275068931	4	4
<code>MPI_INT</code>	1275069445	6	6
<code>MPI_FLOAT</code>	1275069450	10	10
<code>MPI_DOUBLE</code>	1275070475	11	11
<code>MPI_COMM_WORLD</code>	1140850688	91	91
<code>MPI_COMM_SELF</code>	1140850689	92	92
<code>MPI_STATUS_SIZE</code>	5	4	4
<code>MPI_SOURCE</code>	8	4	4
<code>MPI_TAG</code>	12	8	8
<code>MPI_ERROR</code>	16	12	12

The following features are supported:

Table 1.2 Supported Features

Feature Description	Description
Thread-safety	Section 1.3
MPI tracing <ul style="list-style-type: none"> • IO • MPI-2 • fail-safe • correctness checking 	Section 3.1 ROMIO, Section 3.1.8 not supported Section 3.1.7 Section 4
Single-process tracing	Section 3.2
Tracing of Distributed Applications	Section 6
Subroutine tracing	Section 3.3
Tracing of Binaries without Recompilation	Section 3.4
Counter tracing	API in Section 8.7
Automatic Counter tracing of OS Activity	Section 3.9
Automatically Recording Source Location Information	Section 3.7 (requires compiler support)
Manually Recording Source Location Information	API in Section 8.3
Recording Statistical Information	Section 3.6
Tracing Libraries at Different Levels of Detail	Section 3.12
Nonblocking Flushing	MEM-FLUSHBLOCKS

Most of these features are implemented in the Intel Trace Collector libraries, while some are provided by utilities. Here is a list of what the different components do:

Table 1.3 Usage of Components

Component	Usage
<code>libVTnull</code>	Dummy implementation of API (Section 3.11)
<code>libVT</code>	MPI tracing (Section 3.1)
<code>libVTfs</code>	fail-safe MPI tracing (Section 3.1.7)
<code>libVTmc</code>	correctness checking(Section 4)
<code>libVTcs</code>	Tracing of Distributed Applications and Single-processes (Section 6 , Section 3.3)
<code>VT_sample</code>	Automatic Counter tracing with PAPI and <code>getrusage()</code> (Section 3.9)
<code>stftool</code>	Manipulation of trace files (Section 7.4.1)
<code>xstftool/expandvtlog.pl</code>	Conversion of trace files into readable format (Section 7.4.2)
<code>itcpin</code>	Tracing of Binaries without Recompilation (Section 3.5)

1.3 Multithreading

This version of 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. More information on tracing multithreaded applications is found in [section 3.6](#).

1.4 About this Manual

This manual describes how to use Intel® Trace Collector. On Linux* OS, some text is provided as man pages for easier reading in the shell, for example, the Intel Trace Collector API calls (`man VT_enter`) and the Intel Trace Collector configuration (`man VT_CONFIG`). To access the man pages, follow the instructions in the next chapter.

In the PDF version of the manual all special Intel Trace Collector terms and names are hyperlinks that take you to the definition of the word. The documentation is platform-independent, which means that the text and even whole sections may not be applicable for platforms. Hence, if you move between different platforms certain features may work differently.

This manual tries to cover as much as possible independent on the diverse platforms it supports. Linux and Microsoft Windows* frequently have different styles in passing parameters. In particular Intel® Trace Collector Reference Guide parameters to compilers are not consistent between the two operating systems. This manual follows the nomenclature used on Linux. Here is a list of the most important differences and how they are mapped from Linux style to the Microsoft* Windows* way:

Table 1.4 Differences between Linux* and Microsoft Windows* Styles

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

1.5 Intended Audience

This *Reference Guide* helps you understand the feature of the Intel® Trace Collector and how to use this tool to generate tracefiles.

1.6 Conventions and Symbols

The following conventions are used in this document.

Table 1.5 Conventions and Symbols used in this Document

<i>This type style</i>	Document or product names
This type style	Hyperlinks
<code>This type style</code>	Commands, arguments, options, file names
<code>THIS_TYPE_STYLE</code>	Environment variables
<code><this type style></code>	Placeholders for actual values
<code>[items]</code>	Optional items
<code>{ item item }</code>	Selectable items separated by vertical bar(s)
(SDK only)	For Software Development Kit (SDK) users only

2 Installation

On Linux*: After unpacking the Intel® Trace Collector archive in a folder of your choice read the file `relnotes.txt` for information on how to install Intel Trace Collector. On Microsoft Windows* OS: run the installer file `setup.exe` and follow the instructions given in the installation wizard.

To enable the software, Intel will issue you a license key. The license key is a simple text file containing details of the software that will be enabled. An evaluation license key contains a time limited license.

If called without a valid license, or with invalid settings of the above environment variable, installation aborts with an error message like the following one:

```
Checking for flexlm license
Feature to check out: TRACE_COLLECTOR
Error: A license for ITrColL could not be obtained (-1,359,2).
Is your license file in the right location and readable?
The location of your license file has to be specified via
the $INTEL_LICENSE_FILE environment variable.
License file(s) used were (in this order):
Visit http://support.intel.com/support/performance/tools/support.htm
if you require technical assistance.
FLEX_CHECKOUT test failed to acquire license (rc=-1)
```

License management has to be transparent, but if you have any problems during installation, submit an issue to Intel® Premier Support or send an email to tracetools@intel.com.

To acquire a demo license, use Intel Premier Support or contact tracetools@intel.com. This email address can also be used to find out how to purchase the product. At <http://www.intel.com/software/products/cluster/> you will also find a list of your local sales channel.

The installer creates `itacvars.sh` (for shells with Bourne* syntax) and `itacvars.csh` (for shells with csh syntax) on Linux systems. On Microsoft Windows systems you will find an `itacvars.bat` batch script. Sourcing the correct file in a shell (with `. itacvars.sh`, or `source itacvars.csh` or `itacvars.bat` respectively) will update `$LD_LIBRARY_PATH` and `$PATH` as well as set additional environment variables.

`VT_ROOT` points to the root of the Intel Trace Collector installation. It is used to find the Intel Trace Collector include and library files when compiling programs in the example `makefile` (Example: for using it with `-I$(VT_ROOT)`).

`VT_LIB_DIR` points to the folder containing the static Intel Trace Collector libraries. It might be useful to create Makefiles or shorter linkage commands when using `-L`.

`VT_SLIB_DIR` (`VT_DLL_DIR` on Microsoft Windows) points to the folder containing the dynamic Intel Trace Collector libraries. It might be useful to create Makefiles or shorter linkage commands when using `-L`.

`VT_ADD_LIBS` lists libraries Intel Trace Collector is dependent on and needs to be linked against when using Intel Trace Collector.

`VT_MPI` points to the currently selected MPI implementation as used by `VT_LIB_DIR` and `VT_SLIB_DIR`.

`VT_MPI_DLL` (Microsoft Windows version only) name of the MPI dll to be loaded by an Intel Trace Collector dll. If the Fortran interface of MPI is involved and MPI offers this in a separate dll (for example, MPICH2*), it has to be specified through `VT_FMPI_DLL`.

The `itacvars` scripts accept an optional argument to specify the desired MPI implementation you intend to work with: on Linux `impi2`, `impi3` or `mpich` (for example, `source itacvars.csh impi3` selects Intel® MPI Library version 3), on Microsoft Windows `impi32`, `impi64`, `mpich2_32`, `mpich2_64` (for example, `itacvars.bat impi64` selects the 64-bit version of Intel MPI Library). This mechanism allows you to switch between different MPI implementations. Without an argument the scripts assume the MPI implementation you selected the default at installation time (Intel MPI Library on Microsoft Windows).

It is possible to install several different Intel Trace Collector packages in parallel on the same machine by using different folders. Overwriting an old installation with a new one is not recommended, because this will not ensure that obsolete old files are removed. A single dot `.` can be used to install in the directory where the archive was unpacked.

In order to use Intel Trace Collector on a cluster of machines you can either install Intel Trace Collector once in a shared directory which is mounted at the same location on all nodes, or you can install it separately on each node in a local folder. Neither method has a clear advantage when it comes to runtime performance. Root privileges are only needed if writing into the desired install folder requires them.

For using shared libraries (see [Using the Shared Libraries](#)) or in particular for binary instrumentation (see [Tracing of Binaries and Binary Instrumentation](#)) the installation path must be the same on all nodes.

On Microsoft Windows Compute Cluster Server* most of this is handled by OS-provided mechanisms. For Linux there is a mechanism for unattended mass installations in clusters. It consists of the following steps:

1. Start the install script with the option `--duplicate`. It will install Intel Trace Collector as usual, but in addition to that it will create a file called `itc_<platform>_<version>_SilentInstall.ini` in the current folder or, if that folder is not writable, `/tmp/itc_<platform>_<version>/SilentInstall.ini`. Alternatively you can modify the existing `SilentInstallConfigFile.ini`. It is necessary to acknowledge the End User License Agreement by editing that file and replacing `ITC_EULA=reject` with `ITC_EULA=accept`.
2. Run the install script on each node with the option `--silent <.ini file>`. This will install Intel Trace Collector without further questions using the install options from that `.ini` file. Only error messages will be printed, all the normal progress messages are suppressed.

3 How to Use Intel® Trace Collector

3.1 Tracing MPI Applications

Using Intel® Trace Collector for MPI is straightforward: relink your MPI application with the appropriate profiling library and run it following the usual procedures of your system. This will generate a tracefile suitable for use with Intel® Trace Analyzer, including records of all calls to MPI routines as well as all point-to-point and collective communication operations performed by the application.

If you wish to get more detailed information about your application, Intel Trace Collector provides several methods to instrument your application: Binary instrumentation (see [Tracing of Binaries and Binary Instrumentation](#)), compiler instrumentation (see [Compiler-driven Subroutine Instrumentation](#)) and source code instrumentation through its API (see [The Intel® Trace Collector API](#)). This will allow arbitrary user-defined events to be traced; in practice, it is often very useful to record your applications entry and exit to/from subroutines or regions within large subroutines.

The following sections explain how to compile, link and execute MPI applications with Intel Trace Collector; if your MPI is different from the one Intel Trace Collector was compiled for, or is setup differently, then the paths and options may vary. These sections assume that you know how to compile and run MPI applications on your system, so before trying to follow the instructions below, read the relevant system documentation.

3.1.1 Compiling MPI Programs with Intel® Trace Collector

Source files without calls to the Intel® Trace Collector API can be compiled with the usual methods and without any special precautions.

Source files containing calls to the Intel Trace Collector API need to include the appropriate header files: `VT.h` for C and C++ and `VT.inc` for Fortran.

To compile these source files, you must pass the path to the Intel Trace Collector header files to the compiler. On most systems, this is done with the `-I` option, for example `-I$(VT_ROOT)/include`.

3.1.2 Linking MPI Programs with Intel® Trace Collector

Intel® Trace Collector library `libVT.a` contains entry points for all MPI routines. They have to be linked against your application object files before MPI library of your system. Intel Trace Collector requires a set of additional libraries. For your convenience the `itacvars` scripts set the environment variables `$VT_ADD_LIBS` and `$VT_LIB_DIR` accordingly. The easiest way is to use the compile scripts provided by the Intel® MPI Library: simply use the command line argument `-trace`. It does everything for you. All it requires is having sourced/executed the respective `itacvars` script.

In general, a correct link line is achieved as follows (after sourcing/executing the appropriate `itacvars` script):

Linux* OS:

```
mpicc ctest.o -L$VT_LIB_DIR -lVT $VT_ADD_LIBS -o ctest
mpif77 ftest.o -L$VT_LIB_DIR -lVT $VT_ADD_LIBS -o ftest
```

On Microsoft Windows* OS, you do not need to specify %VT_ADD_LIBS% because the corresponding libraries are automatically pulled in by the VT library:

```
mpiicc ctest.obj /link /LIBPATH:%VT_LIB_DIR% VT.lib
mpiifort ftest.obj /link /LIBPATH:%VT_LIB_DIR% VT.lib
```

If your MPI installation is different, then the command may differ and/or you might have to add further libraries manually. Usually it is important that the Intel Trace Collector library is listed on the command line in front of the MPI libraries: ...-lVT -lmpi.... In general, the same Intel Trace Collector library and link line is suitable for all compilers and programming languages.

Dependent on the platform \$VT_ADD_LIBS expands to the following:

```
em64t-lin-impi32:  -ldwarf -lelf -lvtunwind -lnsl -lm -ldl -lpthread
em64t-lin-mpich:  -ldwarf -lelf -lvtunwind -lnsl -lm -ldl -lpthread
ia32-lin-impi32:  -ldwarf -lelf -lvtunwind -lnsl -lm -ldl -lpthread
ia32-lin-mpich:   -ldwarf -lelf -lvtunwind -lnsl -lm -ldl -lpthread
```

NOTE: For more convenient usage and extended portability it is recommended to use \$VT_ADD_LIBS rather than the expanded list of libraries.

NOTE: When using Intel® Professional Edition Compilers, avoid explicitly linking against the math library. In this case, for better performance, it is recommended not to specify -lm and let the Intel compiler use its own and faster implementation.

One exception from these rules are C++ applications. If they call the C MPI API, then tracing works as described above, but if they use the MPI 2.0 C++ API, then Intel Trace Collector cannot intercept the MPI calls. They have to be mapped to the C function calls first with the help of a MPI implementation specific library which has to be placed in front of the Intel Trace Collector library. The name of that wrapper library depends on the MPI implementation; here is the command line option which needs to be added for some of them:

Intel® MPI Library and gcc* < 3.0: -lmpigc

Intel MPI Library and gcc ≥ 3.0 and < 3.4: -lmpigc3

Intel MPI Library and gcc ≥ 3.4: -lmpigc4

MPICH* 1.2.x: -lmpichxx

In all cases, the binary interface of the MPI libraries has to match the one used by Intel Trace Collector (see [section 1.2](#) for details).

3.1.3 Running MPI Programs with Intel® Trace Collector

MPI programs linked with Intel® Trace Collector as described in the previous sections can be started in the same way as conventional MPI applications. Intel Trace Collector reads two environment variables to access the values of runtime options:

`VT_CONFIG` contains the pathname of an Intel Trace Collector configuration file to be read at MPI initialization time. A relative path is interpreted starting from the working directory of the MPI process specified with `VT_CONFIG_RANK`.

`VT_CONFIG_RANK` contains the rank (in `MPI_COMM_WORLD`) of the MPI process that reads the Intel Trace Collector configuration file. The default value is 0. Setting a different value has no effects unless the MPI processes do not share the same file system. The trace data is stored in memory during the program execution, and written to disk at MPI finalization time. The name of the resulting tracefile depends on the format: the base name `<trace>` is the same as the path name of the executable image, unless a different name has been specified in the configuration file. Then different suffices are used depending on the file format:

Structured Trace Format (STF, the default):	<code><trace>.stf</code>
single-file STF format:	<code><trace>.single.stf</code>
old-style ASCII Vampir* format:	<code><trace>.avt</code>

A directive in the configuration file (see [Configuration Format](#)) can influence which MPI process actually writes the tracefile; by default, it is the same MPI process that reads the configuration file.

If relative path names are used it can be hard to find out where exactly the tracefile was written. Therefore Intel Trace Collector prints an informational message to `stderr` with the file name and the current working directory as soon as writing starts.

3.1.4 Examples

The examples in the `./examples` folder show how to instrument C and Fortran code to collect information about application subroutines. They come with a `GNUmakefile` (on Linux* OS) or Visual Studio* project files (on Microsoft Windows* OS) that work for the MPI this Intel® Trace Collector package was compiled for. If you use a different MPI, then you might have to edit this `GNUmakefile`. Unless Intel Trace Collector was installed in a private folder, the examples folder needs to be copied because compiling and running the examples requires write permissions.

3.1.5 Troubleshooting

If generating a trace fails, check first that you can run MPI applications that were linked without Intel® Trace Collector. Then ensure that your MPI is indeed compatible with the one this package was compiled for, as described in [System Requirements and Supported Features](#). The FAQ may have further information. If this still does not help, then please submit a report through the Question and Answer Database (QuAD).

3.1.6 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`.

Table 3.1 MPI function prefixes

MPI Function	Prefix
<code>MPI_Comm_create()</code>	<code>CREATE</code>
<code>MPI_Comm_dup()</code>	<code>DUP</code>
<code>MPI_Comm_split()</code>	<code>SPLIT</code>
<code>MPI_Cart_sub()</code>	<code>CART_SUB</code>
<code>MPI_Cart_create()</code>	<code>CART_CREATE</code>
<code>MPI_Graph_create()</code>	<code>GRAPH_CREATE</code>
<code>MPI_Intercomm_merge()</code>	<code>MERGE</code>

`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.

3.1.7 Tracing of Failing MPI Applications

Normally if a MPI application fails or is aborted, all trace data collected so far is lost: `libVT` needs a working MPI to write the trace file, but the MPI standard does not guarantee that MPI is still operational after a failure. In practice most MPI implementations just abort the application. To solve this problem, link the application against `libVTfs` instead of `libVT`, like this:

```
mpicc ctest.o -lVTfs $VT_ADD_LIBS -o ctest
```

respectively on Microsoft Windows* OS

```
mpicc ctest.obj VTfs.lib /OUT:ctest
```

Under normal circumstances tracing works just like with `libVT`, but communication during trace file writing is done through TCP sockets, so it may be a bit slower than over MPI. In order to establish communication, it needs to know the IP addresses of all involved hosts. It finds them by looking up the hostname locally on each machine or, if that only yields the 127.0.0.1 localhost 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. Failures that it can catch include:

Signals: these include events inside the applications like segfaults and floating point errors, but 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, then it assumes that a deadlock has occurred, stops the application and writes a trace file. The timeout is configurable with `DEADLOCK-TIMEOUT`. "No progress" is defined as "inside the same MPI call". Obviously this is just 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 then Intel Trace Collector still believes that progress is made and will not stop the application. To avoid this, the application has to use blocking MPI calls instead, 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 Microsoft Windows OS, not all features of POSIX* signal handling are available. Therefore, `VTfs` on Microsoft Windows OS uses some heuristics and may not work as reliably as on Linux* OS. Currently it is not possible to stop a Microsoft Windows application run and get a trace file by sending a signal or terminating the job in the Microsoft Windows task manager.

3.1.8 Tracing MPI File IO

On Linux* OS, only Intel® Trace Collector for MPICH* (and compatible) supports 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.

3.2 Single-process Tracing

Traces of just one process can be generated with the `libVTcs` library, which allows the generation of executables that work without MPI.

Linking is accomplished by adding `libVTcs.a` (`VTcs.lib` on Microsoft Windows*) 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 trace. Additional calls exist in `libVTcs` to also trace distributed applications, that is why it is called client-server. Tracing a single process is just a special case of that mode of operation. Tracing distributed applications is described in more detail in [Section 6](#).

Subroutine tracing ([Section 3.3](#)) or binary instrumentation ([Section 3.4](#)) can be used with and without further Intel Trace Collector API (see [Section 8](#)) calls to actually generate trace events.

`libVTcs` uses the same techniques as fail-safe MPI tracing ([Section 3.1.7](#)) to handle failures inside the application, therefore it will generate a trace even if the application segfaults or is aborted with CTRL-C.

3.3 Compiler-driven Subroutine Instrumentation

3.3.1 Intel® Professional Edition Compilers 10.0 and later

The Intel® Professional Edition Compilers can automatically instrument functions during compilation. At runtime Intel® Trace Collector will record all function entries and exits in those compilation units. Compile time instrumentation is enabled with the option `-tcollect` (Linux* OS) or `/Qtcollect` (Microsoft Windows* OS). The option accepts an argument to specify the collecting library to link against. For example, for non-MPI applications you can select `libVTcs` as follows: `-tcollect=VTcs`. The default value is `VT`. Before you use this option with the Intel compiler, set up Intel Trace Collector through one of its set-up scripts. While for MPI applications, the initialization is done automatically; manual interaction is needed for non-MPI programs. In this case you must ensure that Intel Trace Collector gets properly initialized through its API routines `VT_initialize` or `VT_init`.

3.3.2 gcc/g++

Similar function tracing is possible through the GNU* compiler suite version 2.95.2 or later. Object files that contain functions to be traced are compiled with `-finstrument-function` and VT should be able to obtain output about functions in the executable. By default this is done by starting the shell program `nm -P`, which can be changed with the NMCMD `config` option.

3.3.3 Folding

Function tracing can easily generate large amounts of trace data, especially for object oriented programs. Folding function calls at run-time can help here, as described in [Section 3.12](#).

3.3.4 C++ Name Demangling

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

3.4 Tracing of Binaries and Binary Instrumentation

Synopsis

```
itcpin [<ITC options>] -- <application command line>
```

<ITC options> - described below

<command line> - command and (when running it) its options

Description

The `itcpin` utility program manipulates a binary executable file while it runs. It can:

- insert an Intel® Trace Collector library into the binary as if the executable had been linked against it
- automatically initialize the Intel Trace Collector library, if necessary
- record function entry and exit events, thus allowing more detailed analysis of the user code in an application

Analysis

Without further options `itcpin` will just analyze the executable given in the command line to ensure that it can be instrumented and how. With the `--list` option it will print a list of all functions found inside the executable to stdout. The format of this list is the same as the one used for the `STATE` configuration option and its ON/OFF flag indicates whether tracing of a function would be enabled or not. `--list` can be combined with options that specify a configuration to test their effect without actually running the executable. One relevant options is `--debug`, which groups functions by the source file that contains them. The top-level group is always the binary file containing the function, leading to function names of the format: `<basename binary file>:<basename source file>:<function name>`

C++ names are demangled automatically if `--demangle` is specified. In this case their methods are grouped according to the class hierarchy by default instead of using the normal file oriented grouping. When `--filter` is used, then the filter script is passed all available information and can decide itself which grouping it wants to use.

The function list generated that way is limited to the main executable and those dynamic libraries which are already loaded during startup. Functions in libraries which are opened dynamically during the normal execution of the application will not be found this way.

Running

The executable is run only if the `--run` option is given. Unless instructed otherwise, `itcpin` will insert `libVT` if it finds that the main executable references MPI calls. In all other cases it is necessary to choose which library to insert with the `--insert` option.

The installation path of Intel® Trace Collector must be the same on all nodes (see section [Installation](#)) to ensure that it can find the necessary binaries at run-time.

Invoking `itcpin` on the executable as described in the previous section will print a list of all available libraries with a short description of each one. The Intel Trace Collector documentation also has a full list of all available libraries in the [System Requirements and Supported Features](#) section. `libVTcs` is the one used for ordinary function tracing.

If you want to do MPI tracing and MPI was linked statically into the binary, then it is necessary to point `itcpin` towards a shared version of a matching MPI library with `--mpi`.

Choosing which tracing library to insert and the right MPI library is useful, but not required when just using `--list`: if given, then `itcpin` will hide functions that are internal to those libraries and thus cannot be traced.

Function Tracing

The optional function profiling is enabled with the `--profile` flag. This records the entry and exit for functions in the trace file. Limiting the number of recorded functions is recommended to avoid excessive runtime overhead and reduce the amount of trace data. This can be done with one or more of the following options: `--state`, `--activity`, `--symbol`, `--config`. Alternatively, you can use `folding` to prune the amount of recorded trace data dynamically at runtime; see the section [Tracing Library Calls](#) in the Intel Trace Collector documentation for details.

Initialization

You must initialize Intel Trace Collector libraries before you can use them. In MPI tracing libraries this is done implicitly when `MPI_Init()` is called, so inserting the library is enough. For the other libraries there are two possibilities:

- The application may contain calls to `VT_initialize()` and (for `libVTcs`) `VT_clientinit()/VT_serverinit()`. To get the application linked `libVTnull` can be used. During binary instrumentation all API calls will be redirected into the actual Intel Trace Collector library. This is also useful when tracing MPI applications, for example to store additional data in the trace file.
- If the application contains no call to `VT_initialize()`, then `itcpin` will call `VT_initialize()` as soon as the Intel Trace Collector library gets inserted.

Startup Scripts

It is quite common that MPI applications are started by scripts. It is possible to invoke `itcpin` on the startup script or program loader. `itcpin` will then monitor this initial loader and all commands started by it until it finds the main executable.

When inserting an MPI tracing library (regardless whether it was selected explicitly through `--insert` or not) then the first executable which contains MPI calls is treated as the main executable. If the main executable contains no MPI calls because all of them were moved to a shared library which is only going to be loaded later, then this heuristic fails.

In that case and when inserting other Intel Trace Collector libraries, the `--executable` command line option can be used to specify which executable is to be instrumented.

Limitations

The Microsoft Windows* version of `itcpin` has some limitations against Linux* one. Unless the binary to be instrumented comes with a `.pdb` symbol file (that is, has been built with debug information), `itcpin` does not see the symbols inside the main binary and thus cannot detect whether it is an MPI application. In this situation the user has to specify explicitly which Intel Trace Collector library should be inserted (through `--insert`). There is another difference to Linux: the name of the MPI dll the application is linked to has to be specified, either through `--mpi` or through the environment variable `VT_MPI_DLL`. The latter is already done in the `itacvars` script, therefore no special action is needed if the application is linked to the standard MPI dll and `itacvars` has been executed. When running under `itcpin`, it is important that the environment variables `VT_DLL_DIR` and `VT_MPI_DLL` are set on all MPI processes. Depending on your MPI, this might require special flags for `mpiexec` (for example, `mpiexec -env`). For Intel® MPI Library, as an example, after executing the `mpivars` and `itacvars` scripts, the startup command might look like

```
mpiexec -hosts ... -wdir ... -genv VT_DLL_DIR "%VT_DLL_DIR%"  
-genv VT_MPI_DLL "%VT_MPI_DLL%" itcpin --run -- my_app.exe
```

To use `itcpin` on Microsoft Windows OS, you have to disable the McAfee Host Intrusion Prevention* antivirus software.

Supported Directives

--run

Syntax:

Default: `off`

`itcpin` only runs the given executable if this option is used. Otherwise it just analyzes the executable and prints configurable information about it.

--use-debug

Syntax:

Default: `on`

Can be used to disable the usage of debugging information for building the function names in the trace file. By default debugging information is used to find the source file of each function and to group those functions together in the same class.

--list

Syntax:

Default: `off`

Enables printing of all functions found inside the input executable and their tracing state. Function names are listed as they would appear in the trace file:

- class(es) and basic function name are separated by colon(s)
- C++ function names are demangled and the C++ class hierarchy is used if `--demangle` is specified; function parameters are stripped to keep the function names shorter
- functions without such a class or namespace are grouped by source file if that debug information is available; only the basename of the source file is used (`foo.c:bar`)
- all other functions are inside the `Application` default class

--filter

Syntax: `<pattern> <replacement>`

This option allows transforming function names that are encountered in the binary file during execution into more useful names that will appear as predefined functions and function groups in the trace file. For example, all functions with a common prefix `FOO_` (`FOO_bar`) could be turned into functions inside a common group `FOO` (`FOO:bar`).

Please note that this option has changed significantly compared to `itcinstrument` that was available in earlier versions of Intel Trace Collector.

The arguments consist of a pattern that is matched against the input and a replacement expression that controls the transformation into the output.

The input function names have nearly the same format as in the `--list` output above. They contain slightly more information so that a filter can flexibly reduce that information:

- source file names contain the full path, if available
- C++ functions also have their source file as top level class

Example input:

- `/home/joe/src/foo.c:FOO_bar`
- `Application:FOO_bar`
- `/home/joe/src/foo.cpp:app:foo:bar`

Passing `--filter 'Application:FOO_(.)' 'FOO:$1'` would transform `Application:FOO_bar` into `FOO:bar`. The special expression `$1` in the replacement pattern refers to the part of the input that was matched by the first parenthesis in the pattern expression. The single quotes are only needed to protect the expressions from being garbled by the shell, they are not part of the expression. If the pattern or the replacement string contains characters like `$` or `()` which are treated specially by the shell it is recommended to specify the filter expression through an Intel Trace Collector config file: this file would contain lines like

```
FILTER 'Application:FOO_(.)' 'FOO:$1'
```

and would be defined to `itcpin` through `--config`.

You can specify several `--filter` options that are applied from left to right. Try to strip the common prefix `/home/joe/` from the input and then use the remaining directory and file names as groups.

First, throw away the common prefix: `--filter '/home/joe/(.)' '$1'`

Then convert slashes to colons: `--filter '/' ':'`

As another example you might want to handle Fortran 90 module functions by stripping the source file or class, then converting the underscore in the function name into a class separator `--filter '^[^:]*_:(([^_]{2,})$) '$1' --filter '_' ':'`. This works for example, for `src/foo.f90:foo_bar_`.

While the syntax for the pattern expression and the replacement expression is very similar to the Perl* language the underlying regex library is taken from the boost library project (<http://www.boost.org/>), regex version 1.34.1.

NOTE: `--filter` replaces all matched patterns with the replacement, while Perl replaces only the first match by default.

For a detailed reference of the regular expression syntax in the `<pattern>` argument please refer to

http://www.boost.org/doc/libs/1_40_0/libs/regex/doc/html/boost_regex/syntax/perl_syntax.html,

For a detailed reference of the special expressions usable in the `<replacement>` argument (named **Format String** in boost terms) refer to

http://www.boost.org/doc/libs/1_40_0/libs/regex/doc/html/boost_regex/format/perl_format.html.

`--insert`

Syntax: `<libname>`

Default: `libVT` for MPI applications

Intel Trace Collector has several libraries that can be used to do different kinds of tracing. For MPI applications the most useful one is `libVT`, so it is the default. For other applications `itcpin` cannot guess what the user wants to do, so the library which is to be inserted needs to be specified explicitly.

--mpi

Syntax: `<path to MPI>`

If an MPI application is linked statically against MPI, then its executable only contains some of the MPI functions. Several of the functions required by `libVT` may not be present. In this case running the instrumented binary will fail with link errors. `itcpin` tries to detect this failure, but if it happens it will not be able to guess what the MPI is that the application was linked against.

This option provides that information. The MPI installation must have shared libraries which will be searched for in the following places, in this order:

- `<path>`
- `<path>/lib`
- `<path>/lib/shared` and the names (first with version 1.0, then without):
- `libmpich.so`
- `libmpich.so`
- `libmpi.so`
- `libmpi.so`

If `<path>` points towards a file, that file has to be a shared library which implements the PMPI interface and is used directly.

--profile

Syntax:

Default: `off`

Enables function profiling in the instrumented binary. Once enabled, all functions in the executable will be traced. It is recommended to control this to restrict the runtime overhead and the amount of trace data by disabling functions which do not need to be traced (see `--state/symbol/activity` filters).

--config

Syntax: `<filename>`

Specifies an Intel Trace Collector configuration file with `STATE`, `ACTIVITY`, `SYMBOL` configuration options. The syntax of these options is explained in more detail in the documentation of `VT_CONFIG` and the normal pattern matching rules apply.

In this context it only matters whether tracing of a specific function is `ON` or `OFF`. Rule entries given on the command line with `--state`, `--activity`, `--symbol` are evaluated before entries in the configuration file.

--executable

Syntax: `<file pattern>`

Specifies the executable into which the Intel Trace Collector library is to be inserted. When analyzing MPI applications, the default is to pick the first executable which contains MPI calls. Otherwise the first executable invoked by the command line will be instrumented.

The parameter is a file pattern which must match a substring of the full executable name. Multiple file patterns can be separated with `|`. Note that the real name of the executable file is matched against and not the name of a symbolic link through which it might have been invoked.

To find out which executables `itcpin` looks at and why it ignores the right ones, run with verbosity level 3.

--max-threads

Syntax: `<number>`

Default: 100

One internal data structure which tracks threads must have a fixed size to avoid locking. This parameter determines the size of that structure. It must be larger than the number of threads created during the application run and only has to be increased if the run aborts with an error message that instructs to do so.

--verbose

Syntax: `[on|off|<level>]`

Default: on

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

- 0 (= off) disables all output
- 1 (= on) enables only one final message about generating the result
- 2 enables general progress reports by the main process
- 3 enables detailed progress reports by the main process
- 4 the same, but for all processes (if multiple processes are used at all)

Levels higher than two may contain output that only make sense to the developers of Intel Trace Collector.

--state

Syntax: `<pattern> <filter body>`

Default: on

Defines a filter for any state or function that matches the pattern. Patterns are extended shell patterns: they may contain the wild-card 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)
- `java:util:Vector*:*` (all functions contained in Vector classes)
- `*:*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)

`--symbol`

Syntax: `<pattern> <filter body>`

Default: on

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

`--activity`

Syntax: `<pattern> <filter body>`

Default: on

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

3.5 Multithreaded Tracing

To trace multithreaded applications, just link and run as described above. Additional threads will be registered automatically as soon as they call Intel® Trace Collector through MPI wrapper functions or the API. Within each process every thread will have a unique number starting with zero for the master thread.

With the `VT_registerthread()` API function the application developer can control how threads are enumerated. `VT_registernamed()` also supports recording a thread name. `VT_getthrank()` can be used to obtain the thread number that was assigned to a thread.

3.6 Recording Statistical Information

Intel® Trace Collector is able to gather and store statistics about the function calls and their communication. These statistics are gathered even if no trace data is collected, therefore it is a good starting point for trying to understand an unknown application that might produce an unmanageable trace. To run an application in this light weight mode you can either set the environment variables `VT_STATISTICS=ON` and `VT_PROCESS=OFF` or point with `VT_CONFIG` to a file like this:

```
# enable statistics gathering
STATISTICS ON

# no need to gather trace data
```

PROCESS 0:N OFF

The statistics are written into the *.stf file. The stftool (see [Structured Trace File Manipulation](#)) can convert from the machine-readable format to ASCII text with --print-statistics.

The format was chosen so that text processing programs and scripts such as awk*, perl, and Microsoft Excel* can read it. In one line each it prints for each tuple of

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

the following statistics:

- Count; number of communications or number of calls as applicable
- Minimum execution time exclusive callee time
- Maximum execution time exclusive callee time
- Total execution time exclusive callee time
- Minimum execution time inclusive callee time
- Maximum execution time inclusive callee time
- Total execution time inclusive callee time

Within each line the fields are separated by colons :.

Receiver is set to 0xffffffff for file operations and to 0xfffffffef for collective operations. If message size equals 0xffffffff the only defined value is 0xfffffffef to mark it 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:

Table 3.2

	Process-local bucket	Is the same value on all processes?
<code>MPI_Barrier</code>	0	Yes
<code>MPI_Bcast</code>	number of broadcasted bytes	Yes
<code>MPI_Gather</code>	number of bytes sent	Yes
<code>MPI_Gatherv</code>	number of bytes sent	No
<code>MPI_Scatter</code>	number of bytes received	Yes
<code>MPI_Scatterv</code>	number of bytes received	No
<code>MPI_Allgather</code>	number of bytes sent + received	Yes
<code>MPI_Allgatherv</code>	number of bytes sent + received	No
<code>MPI_Alltoall</code>	number of bytes sent + received	Yes
<code>MPI_Alltoallv</code>	number of bytes sent + received	No
<code>MPI_Reduce</code>	number of bytes sent	Yes
<code>MPI_Allreduce</code>	number of bytes sent + received	Yes
<code>MPI_Reduce_Scatter</code>	number of bytes sent + received	Yes
<code>MPI_Scan</code>	number of 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 (as it can happen for instance with `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. This implies that 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 inclusive callee time** and the **Total execution time exclusive callee time** 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 which were instrumented through the API (see [The Intel® Trace Collector API](#)), binary instrumentation (see [Tracing of Binaries and Binary Instrumentation](#)) or compiler driven instrumentation (see [Compiler-driven Subroutine Instrumentation](#)).

Filter utilities, such as awk and perl, and plotting/spreadsheet packages, like Excel, can process the statistical data easily. In the examples directory a perl script called `convert-stats` is provided that illustrates how the values printed by `stftool` (called with the `--dump` and `-print-statistics` options) might be processed: it extracts the total times and transposes the output so that each line has information about one function and all processes instead of one function and process as in the protocol file. It also summarizes the time for all processes. For messages the total message length is printed in a matrix with one row per sender and one column per receiver.

3.7 Recording Source Location Information

Intel® Trace Collector can automatically record the locations of subroutine calls in the source code. Compile the relevant application modules with support for debugging by using these compiler flags that enable the generation of debug information for Intel Trace Collector:

```
mpicc -g -c ctest.c
mpif77 -g -c ftest.c
```

If your compiler does not support a flag, then search for a similar one.

At runtime, enable Program Counter (PC) tracing by either setting the environment variable `VT_PCTRACE` to 5 for example, or by setting `VT_CONFIG` to the name of a configuration file specifying the following, for example:

```
# trace 4 call levels whenever MPI is used

ACTIVITY MPI 4

# trace one call level in all routines not mentioned

# explicitly; could also be for example, PCTRACE 5

PCTRACE ON
```

`PCTRACE` sets the number of call levels for all subroutines that do not have their own setting. Because unwinding the call stack each time a function is called can be very costly and cause considerable runtime overhead, `PCTRACE` is disabled by default and has to be handled with care. It is useful to get an initial understanding of an application which then is followed by a performance analysis without automatic source code locations.

Manual instrumentation of the source code with the Intel Trace Collector API can provide similar information but without the performance overhead (see `VT_scldef()`/`VT_thisloc()` in [Defining and Recording Source Locations](#) for more information).

3.7.1 Notes for IA32 and Intel® 64 architectures

On ia32-Linux* and Intel®64 the compiler generates dwarf-2 debug infos. This is supported by GCC* and was even made the default in GCC 3.1*, but older releases need `-gdwarf-2` to enable that format. The Intel® Professional Edition Compiler also uses it by default since at least version 7.0 and does not need any special options.

Another requirement is that the compiler has to use normal stack frames. This is the default in GCC, but might have been disabled with `-fomit-frame-pointer`. If that 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 compiler does not use normal stack frames by default if optimization is enabled, but it is possible to turn them on with `-fp`. As of version 10.0 of the Intel compiler, this option is deprecated, use `-fno-omit-frame-pointer` instead. Support by other compilers for both features is unknown.

3.8 Recording Hardware Performance Information

On Linux* OS, the Intel® Trace Collector can sample Operating System values for each process with the `getrusage()` system call and hardware counters with the Performance Application Programming Interface (PAPI). Because PAPI and `getrusage()` might not be available on a system, support for both 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 was possible to provide a precompiled object files only for IA32 systems, because PAPI was either not available or not installed when this package was prepared. The `VT_sample.o` file can be rebuilt by entering the Intel Trace Collector lib directory, editing the provided Makefile to match the local setup and then typing `make VT_sample.o`. It is possible to compile `VT_sample.o` without PAPI by removing the line with `HAVE_PAPI` in the provided Makefile. This results in a `VT_sample.o` that only samples `getrusage()` counters, which is probably not as useful as PAPI support.

Add the `VT_sample.o` object file to the link line in front of the Intel Trace Collector library. With the symbolic link from `libVTsample.a` to `VT_sample.o` that is already set in the lib directory it is possible to use `-lVTsample` and the normal linker search rules to include this object file. If it includes PAPI support, then add `-lpapi` also, together with all libraries PAPI itself needs, for example `-lpfm` to find symbols from `libpfm` - please refer to the PAPI documentation for details, which also describes all other aspects of using PAPI. The link line might look like the following one:

```
mpicc ctest.o <search path for PAPI> -lVTsample -lVT -lpapi $VT_ADD_LIBS
<libs required by PAPI> -o ctest
```

Run the application with configuration options that enable the counters of interest. Because Intel Trace Collector cannot tell which ones are interesting, all of them are disabled by default. The configuration option `COUNTER <counter name> ON` enables the counter and accepts wildcards, so that for example `COUNTER PAPI_* ON` enables all PAPI counters at once. [Intel® Trace Collector Configuration](#) describes how to use configuration options.

However, enabling all counters at once is usually a bad idea because logging counters is not required for the analysis but just increases the amount of trace data. Even worse is that many PAPI implementations fail completely with an error in `PAPI_start_counters()` when too many counters are enabled because some of the selected counters are mutually exclusive due to restrictions in the underlying hardware (see PAPI and/or hardware documentation for details).

PAPI counters are sampled at runtime each time a function entry or exit is logged. If this is not sufficient, for example because a function runs for a very long time, give Intel Trace Collector a chance to log data. This is done by inserting calls to `VT_wakeup()` into the source code.

The following Operating System counters are always available, but might not be filled with useful information if the operating system does not maintain them. They are not sampled as often as PAPI counters, because they are unlikely to change as often. Intel Trace Collector only looks at them if 0.1 seconds have passed since last sampling them. This delay is specified in the `VT_sample.c` source code and can be changed by recompiling it. Consult the man page of `getrusage()` or the system manual to learn more about these counters:

Table 3.3

Counter Class: OS		
Counter Name	Unit	Comment
RU_UTIME	s	user time used
RU_STIME	s	system time used
RU_MAXRSS	bytes	maximum resident set size
RU_IXRSS	bytes	integral shared memory size
RU_IDRSS	bytes	integral unshared data size
RU_ISRSS	bytes	integral unshared stack size
RU_MINFLT	#	page reclaims---total vmfaults
RU_MAJFLT	#	page faults
RU_NSWAP	#	swaps
RU_INBLOCK	#	block input operations
RU_OUBLOCK	#	block output operations
RU_MSGSND	#	messages sent
RU_MSGRCV	#	messages received
RU_NSIGNALS	#	signals received
RU_NVCSW	#	voluntary context switches
RU_NIVCSW	#	involuntary context switches

The number of PAPI counters is even larger and not listed here. They depend on the version of PAPI and the CPU. A list of available counters including a short description is usually produced with the command:

```
<PAPI root>/ctests/avail -a
```

3.9 Recording OS Counters

Similar to the process specific counters in the previous section, Intel® Trace Collector can also record some Operating System counters which provide information about a node. In contrast to the process specific counters these counters are sampled only very infrequently by one background thread per node and thus the overhead is very low. The amount of trace data also increases just a little.

Nevertheless recording them is turned off by default and needs to be enabled explicitly with the configuration option `COUNTER <counter name> ON`. The supported counters are:

Table 3.4

Counter Class: OS		
Counter Name	Unit	Comment
<code>disk_io</code>	KB/s	read/write disk IO (any disk in the node)
<code>net_io</code>	KB/s	read/write network IO (any system interface). This might not include the MPI transport layer.
<code>cpu_...</code>	percent	average percentage of CPU time of all CPUs spent in...
<code>cpu_idle</code>	percent	... idle mode
<code>cpu_sys</code>	percent	... system code
<code>cpu_usr</code>	percent	... user code

The delay between recording the current counter values can be changed with the configuration option `OS-COUNTER-DELAY`, with a default of one 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.

3.10 Using the Dummy Libraries

Programs containing calls to the Intel® Trace Collector API (see [The Intel® Trace Collector API](#)) can be linked with a dummy version of the profiling libraries to create an executable that will not generate traces and incur a much smaller profiling overhead. This library is called `libVTnull.a` and resides in the Intel Trace Collector library folder. Here is how a C MPI-application would be linked:

```
mpicc ctest.o -lVTnull $VT_ADD_LIBS -o ctest
```

3.11 Using the Shared Libraries

This version of the Intel® Trace Collector also provides all of its libraries as shared objects. Using the static libraries is easier to handle, but in some cases the shared libraries might be useful. They are located in the folder `itac/slib_hmpii` on Linux* OS or `dllnhmpii` on Microsoft Windows* OS. After sourcing the `itacvars` script with the appropriate `hmpii` argument, the path of the shared libraries' folder is contained in the environment variable `VT_SLIB_DIR` (`VT_DLL_DIR` on Microsoft Windows). It is recommended to use this variable in linker flags like `-L$(VT_SLIB_DIR)` in your `Makefile`.

To use the shared libraries on Linux OS, add `-L$(VT_SLIB_DIR)` to the command line of your linker. Then ensure that your `LD_LIBRARY_PATH` includes this folder on all nodes where the program is started. This can be done either by automatically sourcing the `itacvars` scripts in the login scripts of one's account, setting the variable there directly, or by running the program inside a suitable wrapper script. The installation path of the Intel® Trace Collector should be the same on all nodes (see [Installation](#)) to ensure that it can find the necessary binaries at run-time.

Two different bindings for Fortran are supported in the same library. This works fine when linking against the static Intel Trace Collector because the linker automatically picks just the required objects from the library. When using shared libraries, though, it will refuse to generate a binary because it finds unresolved symbols and cannot tell that those are not needed. To solve this, add `-Wl, --allow-shlib-undefined` to the link line. Note that in some distributions of Linux, for example, RedHat Enterprise Linux 3.0*, the linker's support for this option is broken so that it has no effect (ld version 2.14.90.0.4).

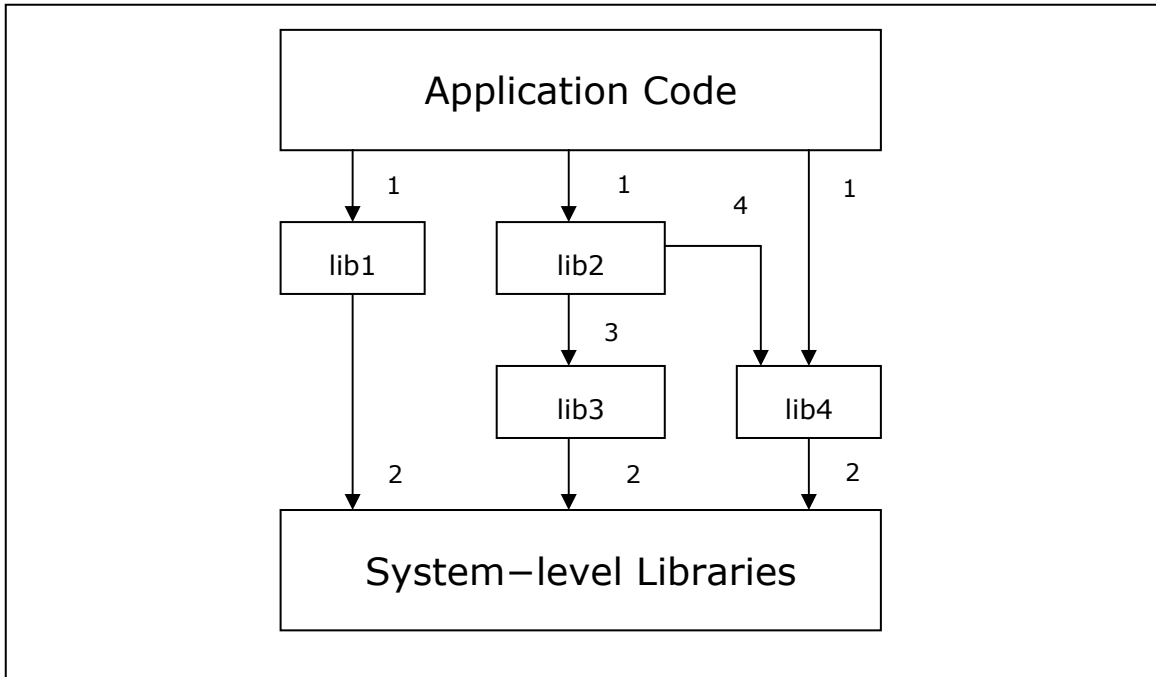
Alternatively you can insert Intel Trace Collector into a MPI binary that was not linked against it. For that to work MPI itself has to be linked dynamically. For MPICH*, you need to configure MPICH with `-enable-sharedlib`, then link the application either with `-shlib` on the command line or the environment variable `MPICH_USE_SHLIB` set to `yes`. When running the dynamically linked MPI application, set `LD_LIBRARY_PATH` as described above and in addition to that, set the environment variable `LD_PRELOAD` to `libVT.so:libpthread.so:libdl.so`. Even more convenient is the simple use of the `-trace` option which is provided by `mpiexec` of the Intel® MPI Library for Linux* OS.

To use the dll versions of Intel Trace Collector on Microsoft Windows you need a full Intel Trace Collector installation on each node. Executing the `itacvars` script with the appropriate MPI argument makes sure that the folder of the Intel Trace Collector dlls (`VT_DLL_DIR`) is added to the `PATH` so that they are found at runtime. On Microsoft Windows OS, the Intel Trace Collector dlls have a special feature: they are not directly linked against MPI, but load the MPI interface dynamically at runtime. Which MPI dlls are to be loaded by Intel Trace Collector is defined through environment variables `VT_MPI_DLL` resp. `VT_FMPI_DLL` for the C resp. Fortran MPI dll. These variables are set through the `itacvars` script to the default dll of the corresponding MPI (for example, `impi.dll` for the Intel® MPI Library for Windows*), but may be changed to a different, but binary compatible MPI dll (for example, `impid.dll`).

3.12 Tracing Library Calls

Suppose you have an application that makes heavy use of libraries or software components which might be developed independently of the application itself. As an application developer the relevant part of the trace are the events inside the application and the top-level calls into the libraries made by the application, but not events inside the libraries. As a library developer the interesting part of a trace are the events inside one's library and how the library functions were called by the application.

Figure 3.1 shows the calling dependencies in a hypothetical application. This is the application developer's view on improving performance:

Figure 3.1 General Structure of an Application Using Many Different Libraries.

- `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 marked as 1)
- `lib3` is never directly called by the application. The application developer has no way to tailor the use of `lib3`. These calls (arrows marked as 3) are therefore of no interest to him, and detailed performance data is not necessary.
- `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, and the information about the indirect calls (arrows marked 4) are not interesting to it.

For the library developer, the performance analysis model is significantly different. Here, the workings of the application are of no concern apart perhaps from call paths that lead into the library. The library developer will need detailed information about the workings of say `lib2`, including the calls from the application, and the calls to component libraries (`lib3` and `lib4`), and to system-level services (MPI). The library developer of `lib2` will have no interest in performance data for `lib1`, and similarly the library developers of `lib1` will have no interest in data from `lib2`, `lib3`, and `lib4`.

If the application and the involved libraries are instrumented to log function calls (either manually or with a compiler), then Intel® Trace Collector supports tracing of the application in a way that just the interesting data is recorded. This is done by writing a filter rule that turns off tracing once a certain function entry has been logged and turns it on again when the same function is left again. This effectively hides all events inside the function. In analogy to the same operation in a graphical tree view this is called **FOLDING** in Intel® Trace Collector. **UNFOLDING** is the corresponding operation that resumes tracing again in a section that otherwise would have been hidden. In contrast to turning tracing on and off with the API calls `VT_traceon()` and `VT_traceoff()`, folding does not log a pseudo-call to `VT_API:TRACEOFF`. Otherwise folding a function that does not call any other function would log more, not less data. It is also not necessary to turn tracing on again explicitly, this is done automatically.

Folding is specified with the **STATE**, **SYMBOL** or **ACTIVITY** configuration options. Shell wildcards are used to select functions by matching against their name (**SYMBOL**), class (**ACTIVITY**) or both (**STATE**). **FOLD** and **UNFOLD** are keywords that trigger folding or unfolding when a matching function is entered. With the **CALLER** keyword one can specify as an additional criteria that the

calling function match a pattern before either folding or unfolding is executed. [Section 9.6](#) has a detailed description of the syntax.

In this section folding is illustrated by giving configurations that apply to the example given above. A C program is provided in `examples/libraries.c` that contains instrumentation calls that log a calltree as it might occur from a program run with library dependencies as in 3.1. Here is an example of call tree for the complete trace (calls were aggregated and sorted by name, therefore the order is not sequential):

```
\->User_Code
  +-->finalize
  |   \->lib2_end
  +-->init
  |   +-->lib1_fini
  |   |   \->lib1_main
  |   |       +-->close
  |   |       +-->lib1_util
  |   |       +-->open
  |   |       \->read
  +-->lib4_log
  |   \->write
  \->work
      +-->lib2_setup
      |   +-->lib3_get
      |   |   \->read
      |   |   \->lib4_log
      |   |       \->write
      |   \->lib4_log
      \->write
```

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

Application Developer: trace the application with only the top-level calls in `lib1`, `lib2`, and `lib4`

```
STATE lib*: * FOLD
\->User_Code
  +-->finalize
  |   \->lib2_end
  +-->init
  |   +-->lib1_fini
  |   |   \->lib1_main
  +-->lib4_log
  \->work
      +-->lib2_setup
      \->lib4_log
```

lib2 Developer: trace everything in `lib2`, plus just the top-level calls it makes

```
STATE *: * FOLD
STATE lib2: * UNFOLD
\->User_Code
  +-->finalize
  |   \->lib2_end
  \->work
      \->lib2_setup
          +-->lib3_get
          \->lib4_log
```

lib2 Developer, detailed view: trace the top-level calls to `lib2` and all `lib2`, `lib3`, `lib4`, and system services invoked by them

```
STATE Application: * FOLD
STATE lib2: * UNFOLD
\->User_Code
  +-->finalize
```

```
| \->lib2_end
\->work
  \->lib2_setup
    +->lib3_get
      | \->read
      \->lib4_log
        \->write
```

Application and lib4 Developers: trace just the calls in lib4 issued by the application

```
STATE *.* FOLD
STATE lib4:* UNFOLD CALLER Application:*
\->User_Code
  +->lib4_log
  | \->write
  \->work
    \->lib4_log
    \->write
```

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

4 Correctness Checking

4.1 Overview

The checking addresses two different concerns:

- Finding programming mistakes in the application which need to be fixed by the application developer. These 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.
- Detecting errors in the execution environment. This is typically done by users of ISV codes or system administrators who just need to know whom they have to ask for help.

In the former case correctness checking is most likely done interactively on a smaller development cluster, but it might also be included in automated regression testing. The second case must use the hardware and software stack on the system that is to be checked.

While doing correctness checking one has to distinguish error detection which is done automatically by tools and error analysis which is done by the user to determine the root cause of the error and eventually fix it.

The error detection in Intel® Trace Collector is implemented in a special library, `libVTmc`, which always does online error detection at runtime of the application. To cover both of the scenarios mentioned above, recording of error reports for later analysis as well as interactive debugging at runtime are both supported. By default `libVTmc` does not write a trace file. Set the `CHECK-TRACING` option to store correctness and performance information to the trace (see [Section 9](#) for details). Use the Intel® Trace Analyzer to view correctness checking events. Take in account that correctness checking requires resources. Do not use the obtained trace for performance analysis.

In some cases special features in Intel® MPI Library are required by `libVTmc`. Therefore this is currently the only MPI for which a `libVTmc` is provided.

The errors are printed to `stderr` as soon as they are found. Interactive debugging is done with the help of a traditional debugger: if the application is already running under debugger control, then the debugger has the possibility to stop a process when an error is found.

Currently it is necessary to manually set a breakpoint in the function `MessageCheckingBreakpoint()`. This function and debug information about it are contained in the Intel Trace Collector library. Therefore it is possible to set the breakpoint and after a process was stopped, to inspect the parameters of the function which describe what error occurred. In later versions it will also be possible to start a debugger at the time when the error is found.

4.2 Usage

4.2.1 Correctness Checking of MPI Application

The first step always is to run the application so that `libVTmc` can intercept all MPI calls. The available methods to achieve this are exactly the same as for normal performance analysis with `libVT` or `libVTfs`:

1. Use `LD_PRELOAD` (Linux* only):

```
mpirexec -genv LD_PRELOAD libVTmc.so -n ...
```
2. Binary instrumentation (see [Tracing of Binaries and Binary Instrumentation](#)).
3. Relinking the application (see [Linking MPI Programs with Intel® Trace Collector](#)).
4. `check` command line option of `mpirexec` (Intel® MPI library only).

To add a correctness checking information to a trace file set `VT_CHECK_TRACING` environment variable. For instance,

```
mpirexec -genv LD_PRELOAD libVTmc.so -genv VT_CHECK_TRACING on -n ...
```

4.2.2 Running with Valgrind*

For distributed memory checking (`LOCAL:MEMORY:INITIALIZATION`) and detecting accesses to memory that is owned by MPI and thus should not be touched by the application (`LOCAL:MEMORY:ILLEGAL_ACCESS`) it is necessary to run all MPI processes under control of the Valgrind* memory checker (Linux* OS only). Valgrind is an open source memory checker which is available for x86 and Intel® 64 Linux machines free of charge from <http://www.valgrind.org/> and/or packaged by Linux distributors; it is not part of the package distributed by Intel and therefore has to be installed separately. Only valgrind versions $\geq 3.2.0$ are supported.

The recommended way of invoking valgrind is to invoke it directly on the main MPI process and to add the `mpirexec -l` option so that all output printed by valgrind is automatically prefixed with the MPI process rank. Intel® Trace Collector detects that `-l` is in effect if the Intel® MPI Library version is 3.0 build 043 or higher and then leaves adding the rank prefix to `mpirexec` also for Intel Trace Collector's own output. One drawback of `-l` is that it leads to a lot of interleaving of lines if processes generate output concurrently; this can be solved by redirecting the output to a file and then either grepping for the output of individual processes (for example, `grep '^0:'` for rank 0) or sorting the output by rank (`sort -n`).

The `LOCAL:MEMORY:ILLEGAL_ACCESS` check causes valgrind reports not just for illegal application accesses (as desired) but also for Intel® MPI Library's 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's lib folder tells valgrind to not print them, but valgrind must be told 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:

```
mpirexec -genv LD_PRELOAD libVTmc.so -l -n <num procs>
valgrind --suppressions=$VT_LIB_DIR/mpi.sup <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.

4.2.3 Configuration

Which errors are checked for at runtime is configurable: 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. The `CHECK` configuration option matches against these full names of each supported error and turns it on or off, like this:

```
# turn all checking off:

# ** matches colons

# * does not

CHECK ** OFF

# selectively turn on specific checks:

# - all local checks

CHECK LOCAL:** ON

# - just one global check

CHECK GLOBAL:MSG:DATATYPE:MISMATCH ON
```

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.

Disabling certain errors serves two purposes: first of all it avoids any report about the disabled errors. Then it can also reduce the overhead for checking if it allows Intel Trace Collector to skip certain code or communication.

Another relevant setting is the `DEADLOCK-TIMEOUT`. This 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.

The different levels for the `VERBOSE` configuration of verbosity have the following effects:

- 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

4.2.4 Analyzing the Results

For interactive debugging the application has to be started so that `stderr` is printed to a console window. Then one 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 on one could even 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, 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 at the moment and therefore that rank is unique.

When the application terminates, Intel Trace Collector does further error checks (for example, unfreed 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`.

4.2.5 Debugger Integration

As mentioned earlier, it is currently 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 one 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 which is used. Here is some information how it works with specific debuggers. For additional information or other debuggers please refer to the debugger's user manual.

The first three debuggers mentioned below can be started by Intel® MPI Library by adding the `-tv`, `-gdb` or `-ldb` options to the command line of `mpiexec` or `mpirun`. Allinea's DDT 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's reports is to start each process in its own X terminal:

```
mpiexec -genv LD_PRELOAD libVTmc.so -l -n <numprocs> xterm -e bash -c
'valgrind --db-attach=yes --suppressions=$VT_LIB_DIR/mpi.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.

4.2.5.1 TotalView Technologies TotalView* debugger

For TotalView* it is necessary to pay attention that the breakpoint should be set for all processes. For TotalView 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, one can create the file `filename.tvd` (filename being the name of the executable) in the working directory in advance and put the following line into it:

```
dfocus gw2 dbreak MessageCheckingBreakpoint
```

Alternatively one 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 one can 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, it is necessary to add these lines to the following file `$HOME/.totalview/tvdrc`. Note there is no dot in the name of this file.

4.2.5.2 GDB*, the GNU* Symbolic Debugger

To automate the procedure of setting breakpoints, `gdb*` supports executing commands automatically. To apply setting this breakpoint for all programs in the current working directory one 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 one can put the following commands into `~/.gdbinit` and then start MPI applications with the normal run command:

```
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
```

4.2.5.3 IDB, the Intel® Debugger

For `idb` the following way is used to automate procedure of setting breakpoints. To apply setting this breakpoint for all programs in the current working directory one can create a file `.dbxinit` with the following line (or add them if it already exists):

```
stop MessageCheckingBreakpoint
```

Alternatively one can add this command to a `.dbxinit` file in the home directory, then it is applied to all programs. A warning about not being able to set this breakpoint in programs without `libVTmc` included is normal and can be ignored.

4.2.5.4 Allinea's Distributed Debugging Tool*, DDT*

DDT* must be configured to run the user's application with the necessary Intel libraries preloaded. This is best achieved from the **Run** dialog by selecting the **Session/Options** menu and choosing **Intel MPI Library** along with selecting the **Submit through queue or configure own mpirun command option**. In the **Submit Command** box enter (without line breaks):

```
mpiexec -genv LD_PRELOAD libVTmc.so -genv VT_DEADLOCK_TIMEOUT 20s
-genv VT_DEADLOCK_WARNING 25s -n NUM_PROCS_TAG

DDTPATH_TAG/bin/ddt-debugger
```

You may leave other boxes empty. Click **OK**.

Start the application by pressing the submit button on DDT's job launch dialog. When the application is ready, select the **Control/Add Breakpoint** menu and add a breakpoint at the function `MessageCheckingBreakpoint`.

Continue to run and debug your application as normal, the program will stop automatically at `MessageCheckingBreakpoint` when an MPI error is detected. You may 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 may 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`.

4.3 Error Detection

4.3.1 Supported Errors

Errors fall into two different categories:

Local: checks only need information available in the process itself and thus do not require additional communication between processes

Global: information from other processes is required

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 grace 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.

The types of all supported errors are listed in *Table 4.1* and

Table 4.2. The description provides just a few keywords for each error, a more detailed description can be found in the following sections.

4.3.2 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.

Table 4.1 Supported 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 <code>MPI_Finalize()</code>
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	freeing an active receive request is discouraged
LOCAL:DATATYPE:NOT_FREED	warning	program creates high number of datatypes
LOCAL:BUFFER:INSUFFICIENT_BUFFER	warning	not enough space for buffered send

Table 4.2 Supported 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	<code>MPI_Comm_free()</code> must be called collectively

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

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 communicator¹. 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.

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>

For distributed memory checking and locking memory that the application should not access, Intel Trace Collector interacts with valgrind* (see [Running 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.

4.3.2.1 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. Datatype 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.

4.3.2.2 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.

4.3.2.3 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 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.

4.3.2.4 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 callstacks 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 callstack 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.

4.3.2.5 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 datatypes. 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.

4.3.2.6 Distributed Memory Checking

(LOCAL:MEMORY:INITIALIZATION)

This feature is enabled by default if all processes run under valgrind* (see [Running with 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 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.

4.3.2.7 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.

4.3.2.8 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.

4.3.2.9 Datatype Handling

(LOCAL:DATATYPE)

Unfreed datatypes 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 datatypes 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.

4.3.2.10 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. Example:

```
local/buffered_send/size
```

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. Examples:

```
local/buffered_send/race
```

```
local/buffered_send/policy
```

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. Example:

```
local/buffered_send/deadlock
```

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.

4.3.2.11 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 callstack 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.

4.3.2.12 Checking Message Transmission

(GLOBAL:MSG)

For each application message, another extra message is sent which includes:

- a datatype signature hash code (for `GLOBAL:MSG:DATATYPE:MISMATCH`)
- a checksum of the data (for `GLOBAL:MSG:DATA_TRANSMISSION_CORRUPTED`)
- a 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 callstack of where the message was sent as well as the current callstack where the error is detected.

4.3.2.13 Datatype Mismatches

(GLOBAL*:DATATYPE:MISMATCH)

Datatype mismatches are detected by calculating a hash value of the datatype 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 function² this is very unlikely. This hash function can also be calculated more efficiently than traditional hash functions.

4.3.2.14 Data Modified during Transmission

(GLOBAL*:DATA_TRANSMISSION_CORRUPTED)

After checking that the datatype 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 datatype mismatches.

4.3.2.15 Checking of 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

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>

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 datatype 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.

4.3.2.16 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.

5 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.

Currently 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 μ s.

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.

5.1 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 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.

int VT_timesync(void)

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)`

5.2 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), a test program is provided which can be run to answer 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` program from the examples directory. 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 a the output of `timertestcs` on 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:

Application Run: a graph connecting the offsets derived from the application's message exchanges with straight lines: this shows whether the deviation from the expected zero offset is linear or not; this can be very noisy because outliers are not removed

Clock Transformation: a graph showing 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: a graph comparing 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.

Note that 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. So 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 later 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...
```

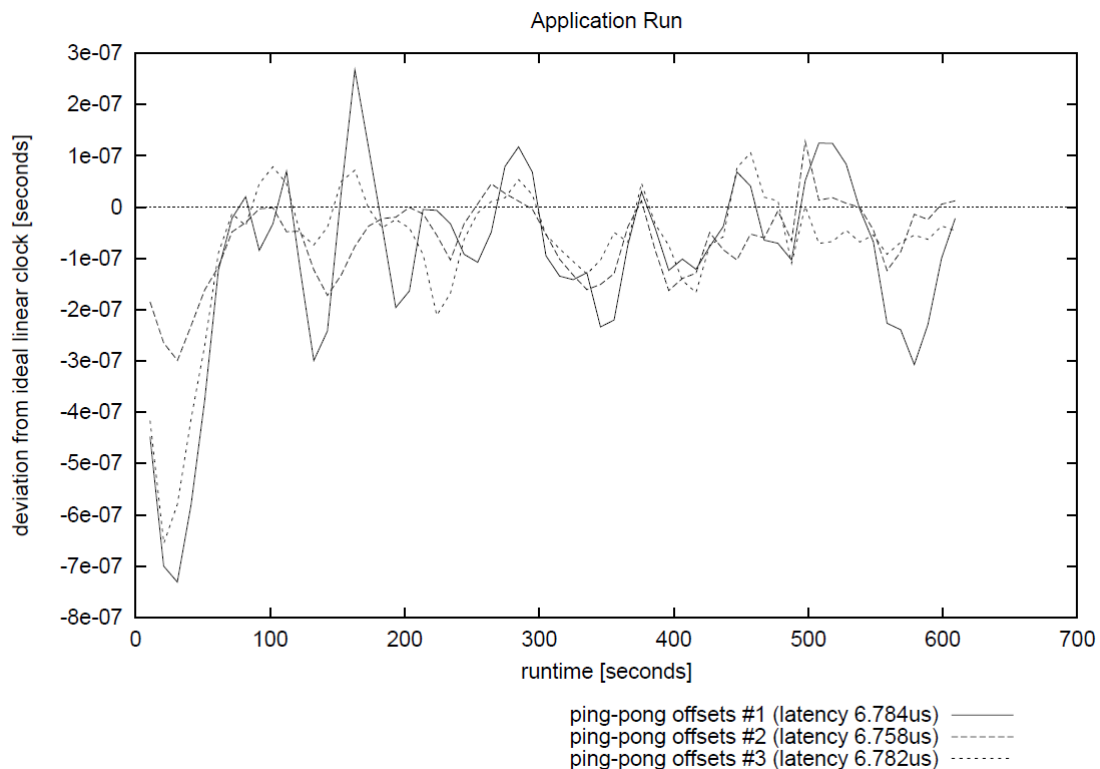
```

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](#) shows that in general Intel Trace Collector managed to keep the test results inside a range of plus-minus 1 μ s 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 μ s. 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.



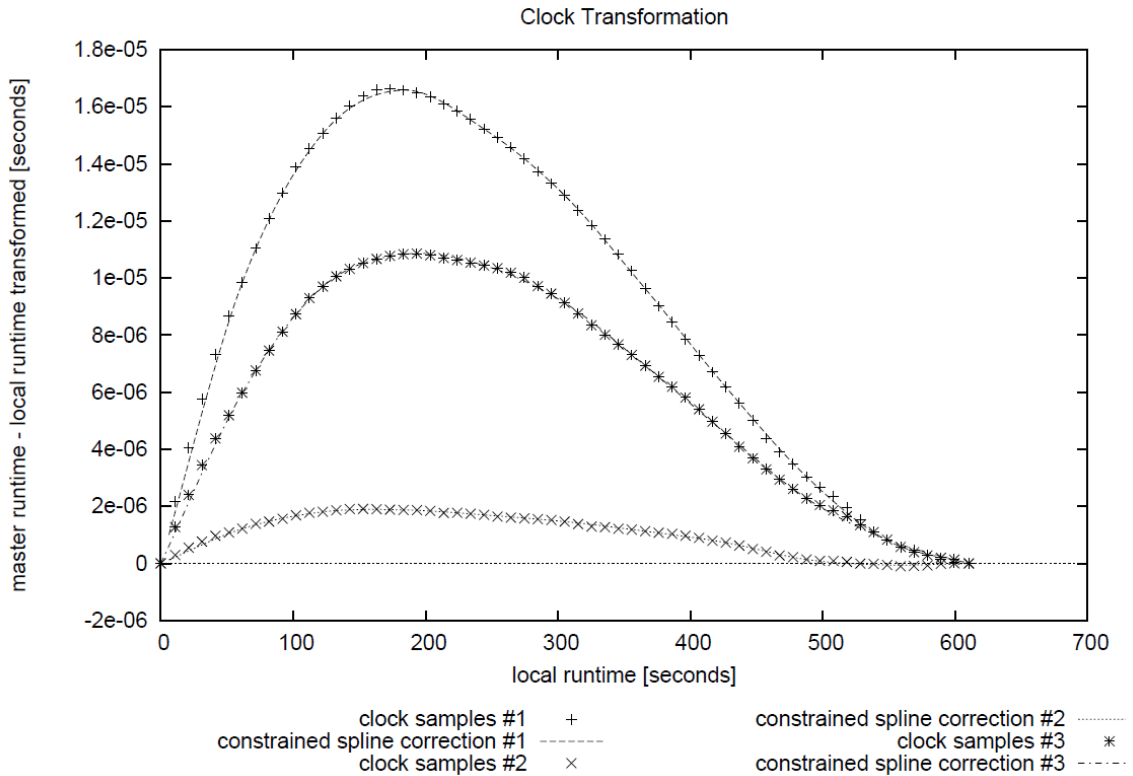
5.2.1 gettimeofday/_ftime

`gettimeofday` is the default timer on Linux* OS with `_ftime` being the equivalent on Microsoft Windows* OS. Its API limits the clock resolution to 1 μ s, 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 μ s.

So the recommendation is to enable NTP, but intermediate clock synchronization by Intel® Trace Collector is still needed to achieve good results.

5.2.2 QueryPerformanceCounter

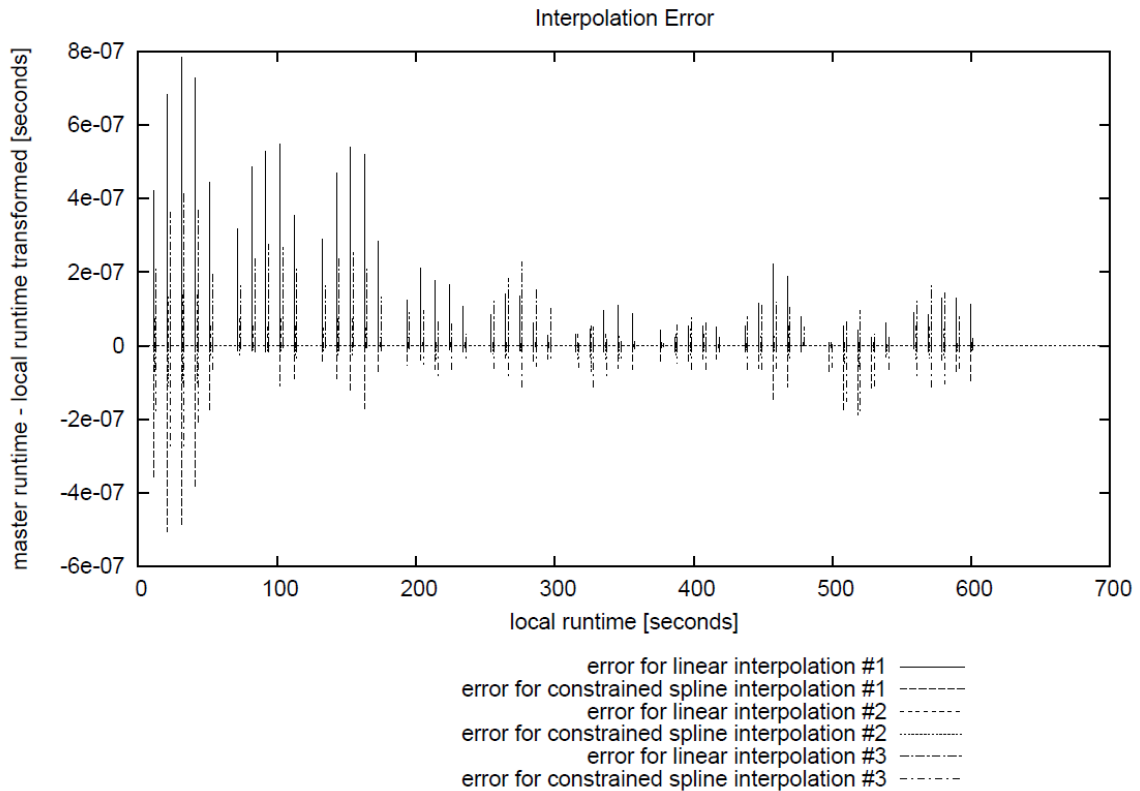
On Microsoft Windows* Intel® Trace Collector uses `QueryPerformanceCounter` as the default timer. As a system function it comes with the same side-effects as `_ftime` but has a higher resolution of around 1 μ s.

5.2.3 CPU Cycle Counter (TSC)

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:

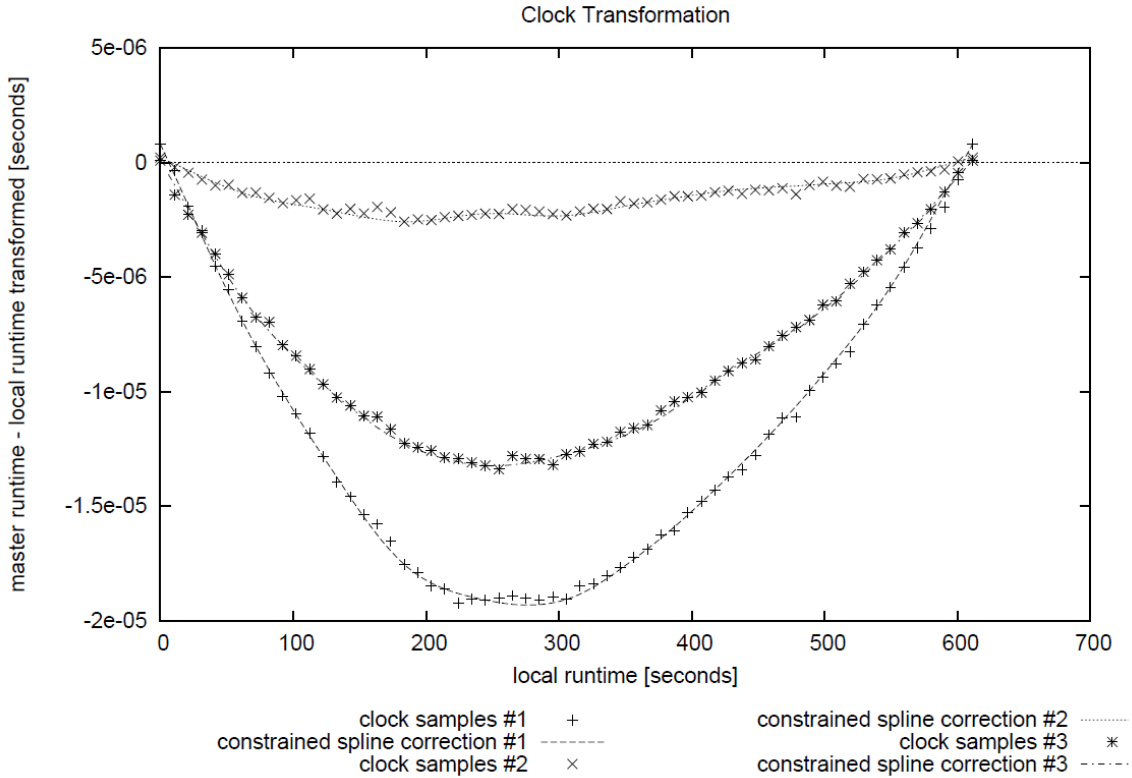
Multi-CPU machines: the counter is CPU-specific, so if threads migrate from one CPU to another the clock that ITC reads might jump arbitrarily. ITC cannot compensate this as it would have to identify the current CPU and read the register in one atomic operation, which cannot be done from user space without 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 ITC 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 ITC itself (see [MEM-FLUSHBLOCKS](#) and [Section 3.10](#)). If the application already is single-threaded, then the additional ITC 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 `gettimeofday()` without NTP



5.2.4 MPI_Wtime()

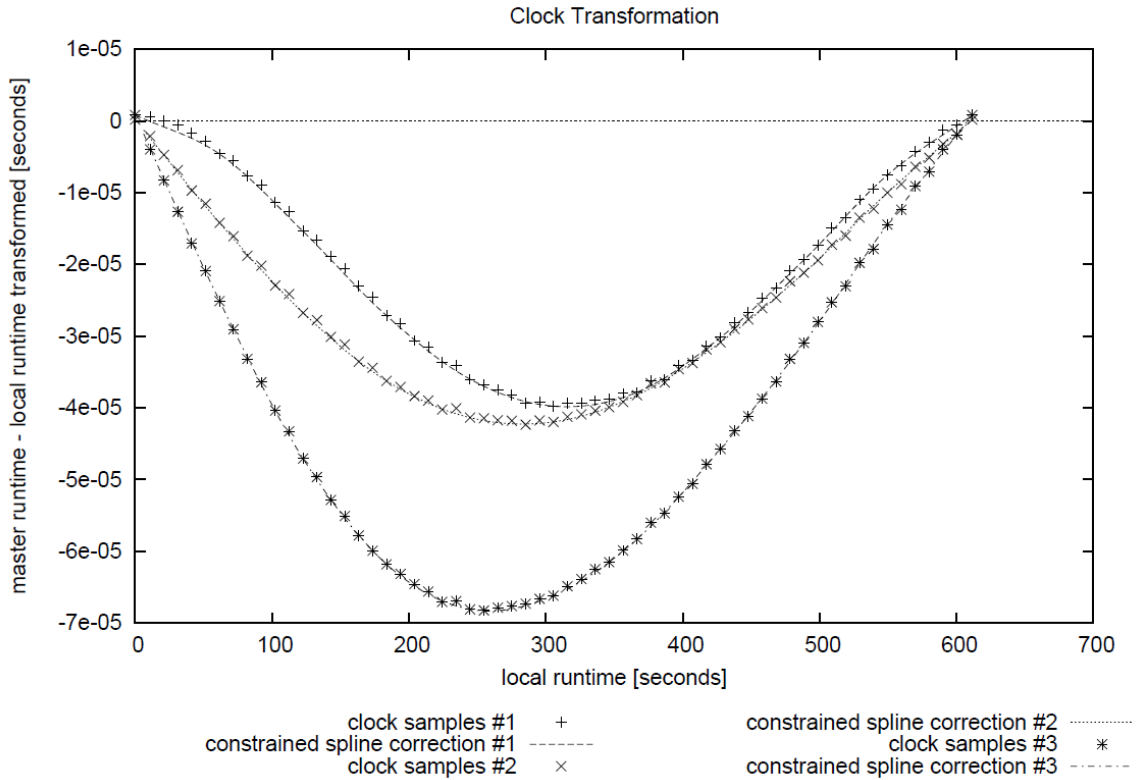
This timer is provided by the MPI implementation. In general this is simply a wrapper around `gettimeofday()` and then using it instead of `gettimeofday()` only has disadvantages: with `gettimeofday()` Intel® Trace Collector knows that processes running on the same node share the same clock and thus 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 (as seen in some versions of MPICH*).

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.

5.2.5 High Precision Event Timers (HPET)

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 `gettimeofday()` with NTP



5.2.6 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 currently does not support this API.

6 Tracing of Distributed 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 was developed that neither relies on MPI nor on the application's communication, but rather implements its own communication layer using TCP/IP.

This chapter describes the design, implementation and usage of Intel Trace Collector for distributed applications. This is work in progress, so this chapter also contains comments about possible extensions and feedback is welcome.

6.1 Design

The following conditions have to be met by the application:

- 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 that 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.

6.2 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.

6.2.1 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.

6.3 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()`.

6.4 Spawning Processes

Spawning new processes is expensive, because it involves setting up TCP communication, clock synchronization, configuration broadcasting, amongst others. It's 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.

6.5 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 the standard Intel® Trace Analyzer tool.

6.6 Usage

Executables in the application are linked with `-lVTcs` and the same additional parameters as listed in [Single-process Tracing](#). 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 config 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 config file and environment variables are described in more detail in the chapter about `VT_CONFIG`.

6.7 Signals

`libVTcs` uses the same techniques as fail-safe MPI tracing (see [Tracing of Failing MPI Applications](#)) 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.

6.8 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 VT 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 resp. 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.

7 Structured Tracefile Format

7.1 Introduction

The Structured Trace File Format (STF) is a format that stores data in several physical files by default. This chapter explains the motivation for this change and provides the technical background to configure and work with the new 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. Provide mechanisms to start at a coarser level of display and then resolve the display into more detailed information. Also, 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 Structured Tracefile Format (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.

7.2 STF Components

A structured tracefile actually consists of a number of files, which can be interpreted as one conceptual data set. See the approximate structure in the [Figure 7.1](#). Depending on the organization of actual files, the following component files will be written, with `<trace>` being the tracefile name that can be automatically determined or set by the `LOGFILE-NAME` directive:

- one index file with the name `<trace>.stf`
- one record declaration file with the name `<trace>.stf.dcl`
- one statistics file with the name `<trace>.stf.sts`
- one message file with the name `<trace>.stf.msg`
- one collective operation file with the name `<trace>.stf.cop`

- one or more process files with the name `<trace>.stf.*.<index>` (where `*` is one of the symbols, `f`, `i`, `s`, `c`, or `x`)
- for the above three kinds of files, one anchor file each with the added extension `.anc`

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 (see [Structured Trace File Manipulation](#)).

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.

7.3 Single-File STF

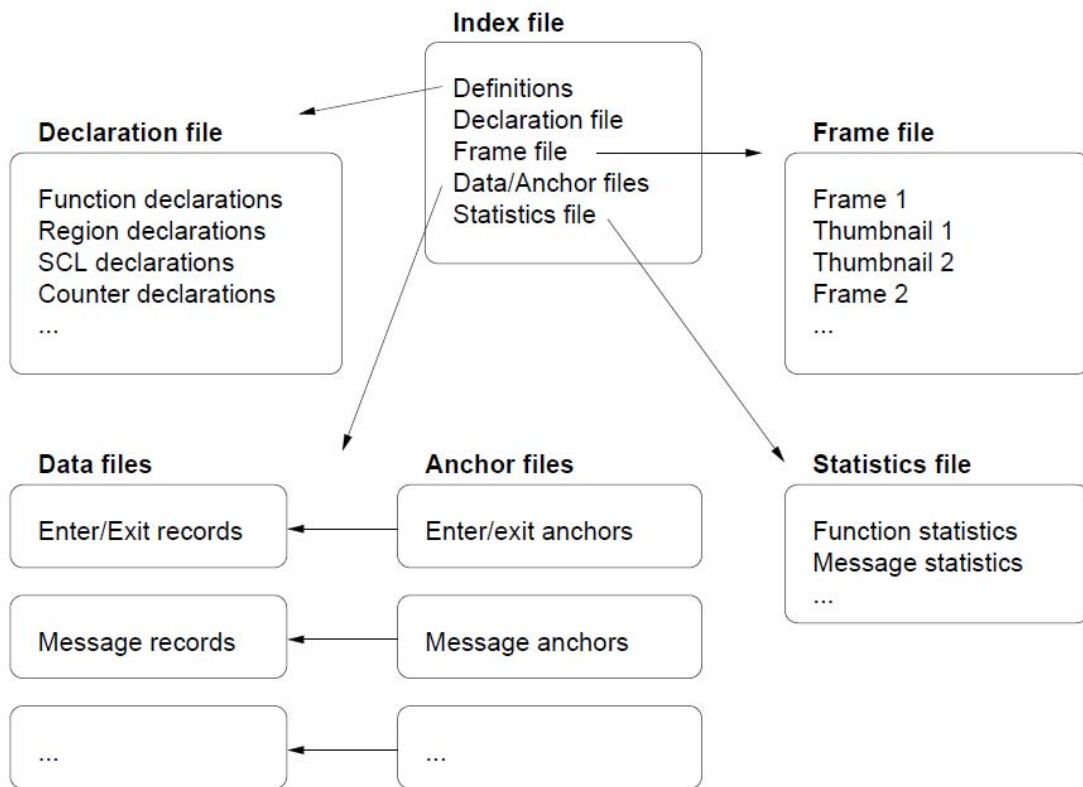
As a new option in Intel® Trace Collector, the trace data can be saved in the single-file STF format. This format is selected by specifying the `LOGFILE-FORMAT STFSINGLE` configuration directive, and it causes all the component files of an STF trace to be combined into one file with the extension `.single.stf`. 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.

7.4 Configuring STF

The STF behavior which can be configured using directives in the Intel® Trace Collector configuration file or the equivalent environment variables are also described in [Intel® Trace Collector Configuration](#).

Figure 7.1 STF Components



To determine the file layout, the following options can be used:

`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

`STF-CHUNKSIZE <bytes>` determines at which intervals the anchors are set

All of these options are explained in more detail in the `VT_CONFIG` chapter.

7.4.1 Structured Trace File Manipulation

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 config 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 config options on the command line.

All config 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 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 config file and environment variables are described in more detail in the documentation for `VT_CONFIG`.

Supported Directives

`--convert`

Syntax: [`<filename>`]

Default: `off`

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.

`--move`

Syntax: [`<file/dirname>`]

Default: `off`

Moves the given file without otherwise changing it. The target can be a directory.

`--remove`

Syntax:

Default: `off`

Removes the given file and all of its components.

--print-files

Syntax:

Default: *off*

List 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-statistics

Syntax:

Default: *off*

Prints the precomputed statistics of the input file to stdout.

--print-reports

Syntax:

Default: *off*

Prints the Message Checker reports of the input file to stdout.

--print-threads

Syntax:

Default: *off*

Prints information about each native thread that was encountered by Intel® Trace Collector when generating the trace.

--print-errors

Syntax:

Default: *off*

Prints the errors that were found in the application.

--dump

Syntax:

Default: *off*

This is a shortcut for `--logfile-name -` and `--logfile-format ASCII`, that is, it prints the trace data to `stdout`.

--request

Syntax: *<type>, <thread triplets>, <categories>, <window>*

This option restricts the data that is written into the new trace to that which 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 not optional 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 uniquely identify 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.

`--ticks`

Syntax:

Default: `off`

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.

`--logfile-name`

Syntax: `<file name>`

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-format`

Syntax: `[ASCII|STF|STFSINGLE|SINGLESTF]`

Default: `STF`

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 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.

`--extended-vtf`

Syntax:**Default:** `off` in Intel® Trace Collector, `on` in `stftool`

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 Intel Trace Analyzer.

--matched-vtf**Syntax:****Default:** `off`

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.

--verbose**Syntax:** `[on|off|<level>]`**Default:** `on`

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

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

See also `VT_CONFIG(3)`

7.4.2 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
```

8 User-level Instrumentation

8.1 The Intel® Trace Collector API

The Intel® Trace Collector library provides the user with a number of routines 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 routines are to be called.

```
#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.

This 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
```

```
#ifndef VT_VERSION < APP_EXPECTED_VT_VERSION
# error "VT.h is not recent enough for APP"
#endif
```

enum _VT_ErrorCode

error codes returned by Intel® Trace Collector API.

Enumeration values:

`VT_OK` OK.

`VT_ERR_NOLICENSE` no valid license found.

`VT_ERR_NOTIMPLEMENTED` Not (yet?) 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_NOTTHREADS` no worker threads.

`VT_ERR_BADINDEX` wrong thread index.

`VT_ERR_COMM` communication error.

`VT_ERR_INVNT` Intel Trace Collector API called while inside an Intel Trace Collector function.

`VT_ERR_IGNORE` non-fatal error code.

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
```

Of course, breaking compatibility that way will be avoided at all costs. Beware of comparing against a fixed number and 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`.

8.2 Initialization, Termination and Control

Intel® Trace Collector is automatically initialized within the execution of the `MPI_Init()` routine. During the execution of the `MPI_Finalize()` routine, 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 to write programs and libraries that use VT without depending on MPI.

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

Initialization of Intel Trace Collector and underlying communication.

`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 a MPI application written in C the program's parameters must be passed, because the underlying MPI might require them. Otherwise they are optional and 0 resp. 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's name and uses this string to find the executable and as the basename for the default logfile name. Other parameters are ignored unless there is the special `--tracecollector-args` parameters: then all following parameters are interpreted as configuration options, written with a double hyphen as prefix and a hyphen instead of underscores (for example, `--tracecollector-args --logfile-format BINARY --logfile-prefix /tmp`). These parameters are then removed from the `argv` array, but not freed. To continue with the program's normal parameters, `--tracecollector-args-end` may be used. There may be more than one block of Intel Trace Collector arguments on the command line.

Fortran:

```
VTINIT(ierr)
```

Parameters:

`argc` a pointer to the number of command line arguments

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

Returns error code.

`int VT_finalize(void)`

Finalize Intel Trace Collector and underlying communication.

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

Fortran:

`VTFINI(ierr)`

Returns error code

`int VT_getrank(int * rank)`

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

Beware that this number is not unique in applications with dynamic process spawning.

Fortran:

`VTGETRANK(rank, ierr)`

Return values:

`rank` process index is stored here

Returns error code

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

`int VT_registerthread(int thindex)`

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()` resp. `MPI_Init()`. The call to `VT_registerthread()` is optional: a 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`

Returns error code:

- `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 wasn't initialized yet

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

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 that, the thread uses the number, but not the new name.

Trying to register a thread twice with different names or numbers is an error. One 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

Returns error code, see `VT_registerthread()`

`int VT_getthrank (int * thrank)`

Get thread index within process.

Either assigned 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

The recording of performance data can be controlled on a per-process basis by calls to the `VT_traceon()` and `VT_traceoff()` routines: 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()` routine. The collection of statistics data is not affected by calls to these routines. With the API routine `VT_tracestate()` a process can query whether events are currently being recorded.

`void VT_traceoff (void)`

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

Fortran:

`VTTTRACEOFF()`

`void VT_traceon (void)`

Turn tracing on for thread if it was disabled, otherwise do nothing.

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

Fortran:

```
VTTRACEON( )
```

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

Get logging state of current thread.

Set by `config` 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 that 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 `config` 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

With the Intel Trace Collector configuration mechanisms described in chapter `VT_CONFIG`, the recording of state changes can be controlled per symbol or activity. For any defined symbol, the `VT_symstate()` routine returns whether data recording for that symbol has been disabled.

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

Get filter state of one state.

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

Note that a state may be active even if the thread's 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

Intel Trace Collector minimizes the instrumentation overhead by first storing the recorded trace data locally in each processor's memory and saving it to disk only when the memory buffers are filled up. Calling the `VT_flush()` routine 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 normally.

```
int VT_flush (void)
```

Flushes all trace records from memory into the flush file.

The location of the flush file is controlled by options in the `config` 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`.

Fortran:

```
VTFLUSH(ierr)
```

Returns error code.

Refer to [Section 9](#) to learn about the `MEM-BLOCKSIZE` and `MEM-MAXBLOCKS` configuration directives that control Intel Trace Collector memory usage.

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.

```
double VT_timestamp (void)
```

Returns an opaque time stamp, or `VT_ERR_NOTINITIALIZED`.

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()
```

```
double VT_timestart (void)
```

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

Fortran:

```
DOUBLE PRECISION VTTIMESTART()
```

8.3 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 routine has been called, or where a code region begins and ends.

Communication events, associating a source location with calls to MPI routines, for example, calls to the send/receive or collective communication and I/O routines.

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 new API routine `VT_scldef()`, which will automatically assign a handle. The old API routine `VT_locdef()` which required the user to assign a handle value has been removed. A source location is a pair of a filename and a line number within that file.

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

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

Fortran:

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

Parameters:

`file` file name

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

Return values:

`sclhandle` the int 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.

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

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 an example of the usage of this call inside a library which 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 IA32 the stack frame is found in the EBP register, on Intel® 64 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 the int it points to is 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 remaining systems, the source location of MPI calls can be recorded by calling the `VT_thisloc()` routine immediately before the call to the MPI routine, with no intervening MPI or Intel Trace Collector API calls.

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

Set 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 config file and supported or no SCL otherwise.

Fortran:

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

Parameters:

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

Returns error code.

8.4 Defining and Recording Functions or Regions

Intel® Trace Analyzer can display and analyze general (properly nested) state changes, relating to subroutine calls, entry/exit to/from code regions and other activities 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 routines are part of the activity MPI, and each one is identified by its routine 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()` routine). All activities and symbols are defined by each process that uses them, but it is no longer 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_begin1()/VT_end1()` routines.

8.4.1 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 routines has been optimized, reducing the overhead of recording state changes.

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

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

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 int it points to is set by Intel Trace Collector

Returns error code.

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

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` doesn't contain a class name and ignored otherwise

Return values:

`statehandle` the int 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".

8.4.2 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()` routine, passing the symbol and activity names, plus the handle value.

It is not necessary to define the activity itself. Take care not to use the same handle value for different symbols.

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

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

Returns error code.

8.4.3 State Changes

The following routines 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, so that support for the old interface may eventually be removed.

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:

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

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.

NOTE: If automatic tracing of source code locations (aka 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).

NOTE: If a 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. So currently it has to be ignored until the trace file format can store this additional information.

If PC tracing is enabled, then the `VT_leave` routine 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 with (and only with) `VT_end()`.

Fortran:

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

Parameters:

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

Returns error code

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

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

Fortran:

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

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

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

Returns error code.

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

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

Fortran:

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

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

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.

Returns error code

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

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.

Returns error code.

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

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:

- `"MPI:TRANSFER/SEND/COPY" + DETAILED-STATES 0: "MPI:TRANSFER"`
- `"MPI:TRANSFER/SEND/COPY" + DETAILED-STATES 1: "MPI:TRANSFER:SEND"`
- `"MPI:TRANSFER/SEND/COPY" + DETAILED-STATES >= 2: "MPI:TRANSFER:SEND:COPY"`

- `"/MPI:INTERNAL" + DETAILED-STATES 0: "" = not logged`
- `"/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`:

- `enter "MPI:TRANSFER/WAIT" = enter "MPI:TRANSFER"`
- `enter "MPI:TRANSFER/COPY" = "MPI:TRANSFER" = ignored by Intel Trace Collector, return code != 0`
- `leave "MPI:TRANSFER/COPY" = ignored by application`
- `enter "MPI:TRANSFER/WAIT" = recursive call; ignored too`
- `leave "MPI:TRANSFER/WAIT" = ignored by application`
- `leave "MPI:TRANSFER/WAIT" = leave "MPI:TRANSFER"`

Same trace with `DETAILED-STATES 1`:

- `enter "MPI:TRANSFER/WAIT" = enter "MPI:TRANSFER:WAIT"`
- `enter "MPI:TRANSFER/COPY" = enter "MPI:TRANSFER:COPY"`
- `leave "MPI:TRANSFER/COPY" = leave "MPI:TRANSFER:COPY"`
- `enter "MPI:TRANSFER/WAIT" = enter "MPI:TRANSFER:WAIT"`
- `leave "MPI:TRANSFER/WAIT" = leave "MPI:TRANSFER:WAIT"`
- `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

```
int VT_wakeup (void)
```

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)
```

Returns error code

8.5 Defining and Recording Overlapping Scopes

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

In contrast to a state, which is entered and left with `VT_begin/VT_end()` rather than with `VT_enter/VT_leave()`, a scope does not follow a stack based approach. It is possible to start a scope **a**, then start scope **b** and stop **a** before **b**:

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

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

As functions, the 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

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

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.

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

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

8.6 Defining Groups of Processes

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

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

```
enum VT_Group
```

Enumeration values:

`VT_ME` the calling thread/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 routine `VT_getprocid()` translates between ranks in `MPI_COMM_WORLD` and handles that can be used for `VT_groupdef()`:

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

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 ($0 \leq \text{procindex} < N$)

Return values:

`procidpointer` to mem place where id is written to

Returns error code.

The same works for threads:

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

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 ($0 \leq \text{procindex} < N$)

`thindex` index of thread

Return values:

`threadid` pointer to mem place where id is written to

Returns error code.

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

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 is either:

- `VT_ME`
- `VT_GROUP_THREAD`
- `VT_GROUP_PROCESS`

- `VT_GROUP_CLUSTER`
- result of `VT_getthreadid()`, `VT_getprocid()` or `VT_groupdef()`

Return values:

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

Returns error code.

To generate a new group that includes the processes with even ranks in `MPI_COMM_WORLD`, you can 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 VT because they are registered the first time they invoke VT. The order can be controlled by calling `VT_registerthread()` as the first API function with a positive parameter.

8.7 Defining and Recording Counters

Intel® Trace Collector introduces the concept of counters to model numeric performance data that changes over the execution time. Counters can be used to capture the values of hardware performance counters, or of program variables (iteration counts, convergence rate, ...) 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:

- the counter name
- the data type

```
enum VT_CountData
```

Enumeration values:

`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.

- the kind of data

enum VT_CountDisplay

Enumeration values:

`VT_COUNT_ABSVAL` counter shall be displayed with absolute values.

`VT_COUNT_RATE` first derivative of counter values shall be displayed.

`VT_COUNT_DISPLAY` mask to extract the display type.

- the semantic associated with a sample value

enum VT_CountScope

Enumeration values:

`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.

- the counter's 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)
- the lower and upper bounds
- the counter's unit (an arbitrary string like FLOP, Mbytes)

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

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 int 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:

Figure 8.1

<code>VT_COUNT_INTEGER</code>	32 bit integer (high)	32 bit integer (low)
<code>VT_COUNT_INTEGER64</code>	64 bit integer	
<code>VT_COUNT_FLOAT</code>	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.

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

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]`.

Returns error code.

The examples directory contains `counterscope.c`, which demonstrates all of these facilities.

8.8 Recording Communication Events

These are API calls that allow logging of message send and receive and MPI-style collective operations. Because they are modelled 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 process rank local within the active communicator, and matches events only if they are taking place in the same communicator (in other words, it is the same behavior as in MPI).

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

Enumeration values:

`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.

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

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`

Returns error code.

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

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_commdf()`)

`sclhandle` handle as defined by `VT_scldef`, or `VT_NOSCL`

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.

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

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_commdf()`)

`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

Returns error code.

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

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`

Returns error code.

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

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`

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 currently not supported.

Enumeration values:

`VT_OP_INVALID` undefined operation, should not be passed to Intel Trace Collector.

`VT_OP_BARRIER`

`VT_OP_BCAST`

`VT_OP_GATHER`

`VT_OP_GATHERV`

`VT_OP_SCATTER`

`VT_OP_SCATTERV`

`VT_OP_ALLGATHER`

`VT_OP_ALLGATHERV`

`VT_OP_ALLTOALL`

`VT_OP_ALLTOALLV`

`VT_OP_REDUCE`

`VT_OP_ALLREDUCE`

`VT_OP_REDUCE_SCATTER`

`VT_OP_SCAN`

`VT_OP_COUNT` number of predefined operations.

Having a duration also introduces the problem of (possibly) having overlapping operations, which has to be taken care of with the following two calls:

```
int VT_begin_unordered(void)
```

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)

int VT_end_unordered (void)
```

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

Fortran:

```
VTENDUNORDERED(ierr)
```

8.9 Additional API Calls in libVTcs

```
int VT_abort (void)
```

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.

Returns:

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

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

Initializes communication in a client/server application.

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

1. client is initialized first, which produces a contact string that must be passed to the server (`*contact == NULL`)
2. 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>`)

3. 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

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

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 infos 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.

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

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.

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

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.

8.10 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;
}
```

8.10.1 VT_FuncDef Class Reference

Defines a function on request and then remembers the handle.

Public Methods

- `VT_FuncDef (const char *symname, const char *classname)`
- `int GetHandle()`

8.10.1.1 Detailed 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`.

8.10.1.2 Constructor & Destructor Documentation

8.10.1.3 VT_FuncDef::VT_FuncDef (const char _symname, const char _classname)

8.10.1.4 Member Function Documentation

8.10.1.5 int VT_FuncDef::GetHandle ()

Checks whether the function is defined already 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.

8.10.2 VT_SclDef Class Reference

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

Public Methods

- `VT_SclDef (const char _file, int line)`
- `int GetHandle ()`

8.10.2.1 Detailed 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()`.

8.10.2.2 Constructor & Destructor Documentation

8.10.2.3 VT_SclDef::VT_SclDef (const char _file, int line)

8.10.2.4 Member Function Documentation

8.10.2.5 int VT_SclDef::GetHandle ()

Checks whether the `scl` is defined already 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)
```

This preprocessor macro creates a static source code location definition for the current file and line in C++.

Parameters:

`_sclvar` name of the static variable which is created

8.10.3 VT_Function Class Reference

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

Public Methods

- `VT_Function(const char *symname, const char *classname)`
- `VT_Function(const char *symname, const char *classname, const char _file, int line)`
- `VT_Function(VT_FuncDef &funcdef)`
- `VT_Function(VT_FuncDef &funcdef, VT_SclDef &scldef)`
- `~VT_Function()`

8.10.3.1 Detailed 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.

8.10.3.2 8.10.3.2 Constructor & Destructor Documentation

8.10.3.3 8.10.3.3 VT_Function::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 for example, uninitialized Intel® Trace Collector.

Parameters:

`symname` the name of the function

`classname` the class this function belongs to

8.10.3.4 VT_Function::VT_Function(const char * symname, const char * classname, const char * file, int line)

Same as previous constructor, but also stores information about where the function is located in the source code.

Parameters:

`symname` the 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

8.10.3.5 VT_Function::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

8.10.3.6 VT_Function::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

8.10.3.7 VT_Function::~VT_Function ()

The destructor marks the function exit.

8.10.4 VT_Region Class Reference

This is similar to `VT_Function`, but should be used to mark regions within a function.

Public Methods

- `void begin(const char *symname, const char *classname)`
- `void begin(const char *symname, const char *classname, const char *file, int line)`
- `void begin(VT_FuncDef &funcdef)`
- `void begin(VT_FuncDef &funcdef, VT_SclDef &scldef)`
- `void end()`
- `void end(const char _file, int line)`
- `void end(VT_SclDef &scldef)`
- `VT_Region()`
- `VT_Region(const char *symname, const char *classname)`
- `VT_Region(const char *symname, const char *classname, const char *file, int line)`
- `VT_Region(VT_FuncDef &funcdef)`
- `VT_Region(VT_FuncDef &funcdef, VT_SclDef &scldef)`
- `~VT_Region()`

8.10.4.1 Detailed 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.

8.10.4.2 Constructor & Destructor Documentation

8.10.4.3 VT_Region::VT_Region()

The default constructor does not start the region yet.

8.10.4.4 VT_Region::VT_Region(const char * symname, const char * classname)

Enter region when it is created.

8.10.4.5 VT_Region::VT_Region(const char * symname, const char * classname, const char * file, int line)

Same as previous constructor, but also stores information about where the region is located in the source code.

8.10.4.6 VT_Region::VT_Region(VT_FuncDef & funcdef)

This is a more efficient version which supports defining the region only once.

8.10.4.7 VT_Region::VT_Region(VT_FuncDef & funcdef, VT_SciDef & scldef)

This is a more efficient version which supports defining the region and source code location only once.

8.10.4.8 VT_Region::~~VT_Region()

the destructor marks the region exit.

8.10.4.9 Member Function Documentation

8.10.4.10 void VT_Region::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

8.10.4.11 void VT_Region::begin(const char * symname, const char * classname, const char * file, int line)

Same as 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

8.10.4.12 void VT_Region::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

8.10.4.13 void VT_Region::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

8.10.4.14 void VT_Region::end()

Leaves the region.

8.10.4.15 void VT_Region::end(const char * file, int line)

Same as 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

8.10.4.16 void VT_Region::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.

9 Intel® Trace Collector Configuration

9.1 Configuring Intel® Trace Collector

With a configuration file, the user can customize various aspects of Intel® Trace Collector operation and define trace data filters.

9.2 Specifying Configuration Options

The environment variable `VT_CONFIG` can be set to the name of an Intel® Trace Collector configuration file. If this file exists, it is read and parsed by the process specified with `VT_CONFIG_RANK` (or 0 as default). The values of `VT_CONFIG` have to be consistent over all processes, although it need not be set for all of them. A relative path is interpreted as starting from the current working directory; an absolute path is safer, because `mpirun` may start your processes in a different folder than you would expect.

In addition to specifying options in a `config` file, all options have an equivalent environment variable. These variables are checked by the process that reads the `config` file after it has parsed the file, so the variables override the `config` file options. Some options like `SYMBOL` may appear several times in the `config` file. A variable may contain line breaks to achieve the same effect.

The environment variable names are listed below in square brackets `[]` in front of the `config` option. Their names are always the same as the options, but with the prefix `T_` and hyphens replaced with underscores.

Finally, it is also possible to specify configuration options on the command line of a program. The only exception is Fortran programs, because Intel Trace Collector's access to command line parameters is limited there. To avoid conflicts between Intel Trace Collector parameters and normal application parameters, only parameters following the special `--itc-args` are interpreted by Intel Trace Collector. To continue with the application's normal parameters, `--itc-args-end` may be used. There may be more than one block of Intel Trace Collector arguments on the command line.

9.3 Configuration Format

The configuration file is a plain ASCII file containing a number of directives, one per line; any line starting with the `#` character is ignored. Within a line, whitespace separates fields, and double quotation marks are to be used to quote fields containing whitespace. Each directive consists of an identifier followed by arguments. With the exception of filenames, all text is case-insensitive. In the following discussion, items within angle brackets (`<` and `>`) denote arbitrary case-insensitive field values, and alternatives are put within square brackets (`[` and `]`) and separated by a vertical bar `|`.

Default values are given in round brackets after the argument template, unless the default is too complex to be given in a few words. In this case the text explains the default value. In general the default values are chosen so that features that increase the amount of trace data have to be

enabled explicitly. Memory handling options default to keeping all trace records in memory until the application is finalized.

9.4 Syntax of Parameters

9.4.1 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 + sign. For instance, the string **1m+30s** refers to one minute and 30 seconds of execution time.

9.4.2 Boolean Value

Boolean values are set to **on/true** to turn something on and **off/false** to turn it off. Just using the name of the option without the **on/off** argument is the same as **on**.

9.4.3 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 with 1/1024/1024²/1024³. If no suffix is given the number specifies bytes.

9.5 Supported Directives

LOGFILE-NAME

Syntax: *<file name>*

Variable: **VT_LOGFILE_NAME**

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.

PROGNAME

Syntax: *<file name>*

Variable: **VT_PROGNAME**

This option can be used to provide a fallback for the executable name in case of Intel® Trace Collector not being able 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 Intel Trace Collector may need to read the executable to find source code information (see `PCTRACE` config option). `UNKNOWN` is used if the file name is unknown and not specified explicitly.

LOGFILE-FORMAT

Syntax: `[ASCII | STF | STFSINGLE | SINGLESTF]`

Variable: `VT_LOGFILE_FORMAT`

Default: `STF`

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 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.

EXTENDED-VTF

Syntax:

Variable: `VT_EXTENDED_VTF`

Default: `off` in Intel® Trace Collector, `on` in `stftool`

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 Intel Trace Analyzer.

PROTOFILE-NAME

Syntax: `<file name>`

Variable: `VT_PROTOFILE_NAME`

Specifies the name for the protocol file containing the config options and (optionally) summary statistics for a program run. 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, then the name is the name of the tracefile with the suffix `.prot`.

LOGFILE-PREFIX

Syntax: `<directory name>`

Variable: `VT_LOGFILE_PREFIX`

Specifies the directory of the trace or log file. 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.

CURRENT-DIR

Syntax: *<directory name>*

Variable: `VT_CURRENT_DIR`

Intel® Trace Collector will use 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.

VERBOSE

Syntax: `[on|off|<level>]`

Variable: `VT_VERBOSE`

Default: `on`

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

- 0 (= `off`) disables all output
- 1 (= `on`) enables only one final message about generating the result
- 2 enables general progress reports by the main process
- 3 enables detailed progress reports by the main process
- 4 the same, but for all processes (if multiple processes are used at all)
- Levels larger than 2 may contain output that only makes sense to the developers of Intel® Trace Collector.

LOGFILE-RANK

Syntax: *<rank>*

Variable: `VT_LOGFILE_RANK`

Default: `0`

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`.

STOPFILE-NAME

Syntax: *<file name>*

Variable: `VT_STOPFILE_NAME`

Specifies the name of a file which indicates that 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*.

If specified, 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.

PLUGIN

Syntax: *<plugin name>*

Variable: `VT_PLUGIN`

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

CHECK

Syntax: *<pattern>* *<on/off>*

Variable: `VT_CHECK`

Default: `on`

Enables or disables error checks matching the pattern.

CHECK-MAX-ERRORS

Syntax: *<number>*

Variable: `VT_CHECK_MAX_ERRORS`

Default: `1`

Number of errors that has to be reached by a process before aborting the application. `0` disables the limit. Some errors are fatal and always cause an abort. Errors are counted per-process to avoid the need for communication among processes, as that has several drawbacks which outweigh the advantage of a global counter.

Errors usually should not be ignored because they change the behavior of the application, thus the default value stops immediately when the first such error is found.

CHECK-TRACING

Syntax: *<on/off>*

Variable: `VT_CHECK_TRACING`

Default: `off`

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

In the normal libraries tracing is always enabled.

CHECK-MAX-REPORTS

Syntax: *<number>*

Variable: `VT_CHECK_MAX_REPORTS`

Default: 0

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 also automatically shut down the application, if that is desired.

CHECK-SUPPRESSION-LIMIT

Syntax: `<number>`

Variable: `VT_CHECK_SUPPRESSION_LIMIT`

Default: 10

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 will still be reported. Just as with `CHECK-MAX-ERRORS` these are a per-process counters.

Note that 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: `<time>`

Variable: `VT_CHECK_TIMEOUT`

Default: 5s

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-MAX-PENDING

Syntax: `<number>`

Variable: `VT_CHECK_MAX_PENDING`

Default: 20

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

CHECK-MAX-REQUESTS

Syntax: `<number>`

Variable: `VT_CHECK_MAX_REQUESTS`

Default: 100

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-MAX-DATATYPES

Syntax: `<number>`

Variable: `VT_CHECK_MAX_DATATYPES`

Default: `1000`

Each time the total number of currently defined datatypes 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-LEAK-REPORT-SIZE

Syntax: `<number>`

Variable: `VT_CHECK_LEAK_REPORT_SIZE`

Default: `10`

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

DETAILED-STATES

Syntax: `[on|off|<level>]`

Variable: `VT_DETAILED_STATES`

Default: `off`

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:

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

ENTER-USERCODE

Syntax: `[on|off]`

Variable: `VT_ENTER_USERCODE`

Default: on in most cases, off for Java* function tracing

Usually Intel® Trace Collector enters the `Application:User_Code` state automatically when registering a new thread. This make 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.

COUNTER

Syntax: `<pattern> [on|off]`

Variable: `VT_COUNTER`

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 Intel® Trace Collector are disabled. Those automatic counters are not supported for every platform.

INTERNAL-MPI

Syntax: `[on|off]`

Variable: `VT_INTERNAL_MPI`

Default: `on`

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.

PCTRACE

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

Variable: `VT_PCTRACE`

Default: `off` for performance analysis, `on` otherwise

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-FAST

Syntax: `[on|off]`

Variable: `VT_PCTRACE_FAST`

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

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.

PCTRACE-CACHE

Syntax: `[on|off]`

Variable: `VT_PCTRACE_CACHE`

Default: `on`

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.

PROCESS

Syntax: `<triplets> [on|off|no|discard]`

Variable: `VT_PROCESS`

Default: `0:N on`

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. F.i. 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 will be 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.

CLUSTER

Syntax: `<triplets> [on|off|no|discard]`

Variable: `VT_CLUSTER`

Same as `PROCESS`, but filters 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.

MEM-BLOCKSIZE

Syntax: `<number of bytes>`

Variable: `VT_MEM_BLOCKSIZE`

Default: `64KB`

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

MEM-MAXBLOCKS

Syntax: *<maximum number of blocks>*

Variable: `VT_MEM_MAXBLOCKS`

Default: `4096`

Intel® Trace Collector will never allocate more than this number of blocks in main memory. If the maximum number of blocks is filled or allocating new blocks fails, then Intel Trace Collector will either flush some of them onto disk (`AUTOFLUSH`), overwrite the oldest blocks (`MEM-OVERWRITE`) or stop recording further trace data.

MEM-MINBLOCKS

Syntax: *<minimum number of blocks after flush>*

Variable: `VT_MEM_MINBLOCKS`

Default: `0`

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

MEM-INFO

Syntax: *<threshold in bytes>*

Variable: `VT_MEM_INFO`

Default: `500MB`

If larger than zero, then Intel® Trace Collector will print 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.

AUTOFLUSH

Syntax: `[on|off]`

Variable: `VT_AUTOFLUSH`

Default: `on`

If enabled (which it is by default), then Intel® Trace Collector will append 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`.

MEM-FLUSHBLOCKS

Syntax: *<number of blocks>*

Variable: `VT_MEM_FLUSHBLOCKS`

Default: `1024`

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-OVERWRITE

Syntax: `[on|off]`

Variable: `VT_MEM_OVERWRITE`

Default: `off`

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

FLUSH-PREFIX

Syntax: `<directory name>`

Variable: `VT_FLUSH_PREFIX`

Default: content of `env` variables or `/tmp`

Specifies the directory of the flush file. 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 will be created and immediately removed while the processes keep their file open. This has two effects:

- flush files do not clutter the file system if processes get killed prematurely
- during flushing, the remaining space on the file systems gets less although the file which grows is not visible any more

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:

- first folder with more than 512MB
- 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:

- `BIGTEMP`
- `FASTTEMP`
- `TMPDIR`
- `TMP`
- `TMPVAR`
- `/work`
- `/scratch`

- `/tmp`

FLUSH-PID

Syntax: `[on|off]`

Variable: `VT_FLUSH_PID`

Default: `on`

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

ENVIRONMENT

Syntax: `[on|off]`

Variable: `VT_ENVIRONMENT`

Default: `on`

Enables or disables logging of attributes of the runtime environment.

STATISTICS

Syntax: `[on|off|<hash_size>]`

Variable: `VT_STATISTICS`

Default: `off`

Enables or disables statistics about messages and symbols. These statistics are gathered by 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.

STF-USE-HW-STRUCTURE

Syntax: `[on|off]`

Variable: `VT_STF_USE_HW_STRUCTURE`

Default: usually `on`

If the STF format is used, then trace information can be stored in different files. If this option is enabled, then 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. It is not enabled if each process is running on its own node.

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.

STF-PROCS-PER-FILE

Syntax: `<number of processes>`

Variable: `VT_STF_PROCS_PER_FILE`

Default: `16`

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 each 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, multithreaded application trace this configuration option is used to determine the number of threads per file.

STF-CHUNKSIZE

Syntax: `<number of bytes>`

Variable: `VT_STF_CHUNKSIZE`

Default: `64KB`

Intel® Trace Collector uses so called anchors to navigate in STF files. This value determines how many bytes of trace data are written into a file before setting the next anchor. Using a low number allows more accurate access during analysis, but increases the overhead for storing and handling anchors.

GROUP

Syntax: `<name> <name>|<triplet>[, ...]`

Variable: `VT_GROUP`

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
```

OS-COUNTER-DELAY

Syntax: `<delay>`

Variable: `VT_OS_COUNTER_DELAY`

Default: `1 second`

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

DEADLOCK-TIMEOUT

Syntax: *<delay>*

Variable: `VT_DEADLOCK_TIMEOUT`

Default: `1 minute`

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: *<delay>*

Variable: `VT_DEADLOCK_WARNING`

Default: `5 minutes`

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.

HANDLE-SIGNALS

Syntax: *<triplets of signal numbers>*

Variable: `VT_HANDLE_SIGNALS`

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

This option controls whether 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`).

ALTSTACK

Syntax: `[on|off]`

Variable: `VT_ALTSTACK`

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 be terminated.

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

TIMER

Syntax: *<timer name or LIST>*

Variable: `VT_TIMER`

Default: `gettimeofday`

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 against 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. In this mode no error messages are printed if initialization of a certain timer fails, it is simply listed 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: `<number> 0`

Variable: `VT_TIMER_SKIP`

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

SYNC-MAX-DURATION

Syntax: `<duration>`

Variable: `VT_SYNC_MAX_DURATION`

Default: `1 minute`

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 or equal 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`) will be 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: `<message number>`

Variable: `VT_SYNC_MAX_MESSAGES`

Default: `100`

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: `<duration>`

Variable: `VT_SYNC_PERIOD`

Default: `-1 seconds = disabled`

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: `[on|off]`

Variable: `VT_SYNCED_CLUSTER`

Default: `off`

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: `[on|off]`

Variable: `VT_SYNCED_HOST`

Default: `off`

Use this setting to override whether Intel® Trace Collector treats the clock of all processes on 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, then this option is ignored.

NMCMD

Syntax: `<command + args> "nm -P"`

Variable: `VT_NMCMD`

If function tracing with GCC 2.95.2+'s `-finstrument-function` is used, then Intel® Trace Collector will be 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`. You can override the default if it is not appropriate on your system. 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`.

UNIFY-SYMBOLS

Syntax: `[on|off]`

Variable: `VT_UNIFY_SYMBOLS`

Default: `on`

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

UNIFY-SCLS

Syntax: `[on|off]`

Variable: `VT_UNIFY_SCLS`

Default: `on`

Same as `UNIFY-SYMBOLS` for SCLs.

UNIFY-GROUPS

Syntax: `[on|off]`

Variable: `VT_UNIFY_GROUPS`

Default: `on`

Same as `UNIFY-SYMBOLS` for groups.

UNIFY-COUNTERS

Syntax: `[on|off]`

Variable: `VT_UNIFY_COUNTERS`

Default: `on`

Same as `UNIFY-SYMBOLS` for counters.

9.6 How to Use the Filtering Facility

A single configuration file can contain an arbitrary number of filter directives that are evaluated whenever a state is defined. Since they are evaluated in the same order as specified in the configuration file, the last filter matching a state determines whether it is traced or not. This scheme makes it easily possible to focus on a small set of activities without having to specify complex matching patterns. Being able to turn entire activities (groups of states) on or off helps to limit the number of filter directives. All matching is case-insensitive.

Example:

```
# disable all MPI activities
ACTIVITY MPI OFF

# enable all send routines in MPI
STATE MPI:*send ON

# except MPI_Bsend
```

```

SYMBOL MPI_bsend OFF

# enable receives

SYMBOL MPI_recv ON

# and all test routines

SYMBOL MPI_test* ON

# and all wait routines, recording locations of four calling levels

SYMBOL MPI_wait* 4

# enable all activities in the Application class, without locations

ACTIVITY Application 0

```

In effect, all activities in the class Application, all MPI send routines except `MPI_Bsend()`, and all receive, test and wait routines will be traced. All other MPI routines will not be traced.

Because inserting line break into environment variables can be difficult, the `SYMBOL/STATE/ACTIVITY` rules also support multiple entries per line or environment variable, as in:

```
SYMBOL MPI_* off MPI_Barrier on
```

Besides filtering specific activities or states it is also possible to filter by process ranks in `MPI_COMM_WORLD`. This can be done with the configuration file directive `PROCESS`. The value of this option is a comma separated list of Fortran 90-style triplets. The formal definition is as follows:

```

<PARAMETER-LIST> := <TRIPLET>[, <TRIPLET>, ...]

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

```

The default value for `<UPPER-BOUND>` is `N` (equals size of `MPI_COMM_WORLD`) and the default value for `<INCREMENT>` is `1`.

For instance changing tracing only on even process ranks and on process 1 the triplet list is: `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, the `CLUSTER` filter option can be used to filter for particular SMP nodes.

The `STATE/ACTIVITY/SYMBOL` rule body may offer even finer control over tracing depending on the features available on the current platform:

- Special filter rules make it possible to turn tracing on and off during runtime when certain states (aka functions) are entered or left. In contrast to `VT_traceon/off()` no changes in the source code are required for this. So called actions are triggered by entering or leaving a state and executed before the state is logged.
- If folding is enabled for a function, then this function is traced, but not any of those that it calls. If you want to see one of these functions, then unfold it.
- Counter sampling can be disabled for states.

Here's the formal specification:

```

<SCLRANGE> := on|off|<trace>|<skip>:<trace>

<PATTERN> := <state or function wild-card as defined for STATE>

```

```

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

<ACTION> := traceon|traceoff|restore|none|

begin_scope <SCOPE_NAME>|end_scope <SCOPE_NAME>

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

<ENTRYTRIGGER> := entry <TRIGGER>

<EXITTRIGGER> := exit <TRIGGER>

<COUNTERSTATE> := counteron|counteroff

<FOLDING> := fold|unfold

<CALLER> := caller <PATTERN>

<RULEENTRY> := <SCLRANGE>|<ENTRYTRIGGER>|<EXITTRIGGER>|

<COUNTERSTATE>|<FOLDING>|<CALLER>

```

The filter body of a filter may still consist of a `<SCLRANGE>` which is valid for every instance of the state (as described above), but also of a counter state specification, an `<ENTRYTRIGGER>` which is checked each time the state is entered and an `<EXITTRIGGER>` for leaving it. The caller pattern, if given, is an additional criterion for the calling function that has to match before the entry, exit or folding actions are executed. The body may have any combination of these entries, separated by commas, as long as no entry is given more than once per rule.

Counter sampling can generate a lot of data, and some of it may not be relevant for every function. By default all enabled counters are sampled whenever a state change occurs. The `COUNTERON/OFF` rule entry modifies this for those states that match the pattern. There is no control over which counters are sampled on a per-state basis, though, you can only enable or disable sampling completely per state. This example disables counter sampling in any state, then enables it again for MPI functions:

```

SYMBOL * COUNTEROFF

ACTIVITY MPI COUNTERON

```

9.7 The Protocol File

The protocol file has the same syntax and entries as a Intel® Trace Collector configuration file. Its extension is `.prot`, with the basename being the same as the tracefile. It lists all options with their values used when the program was started, thus it can be used to restart an application with exactly the same options.

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 currently supported:

INFO NUMPROCS

Syntax: *<num>*

Number of processes in `MPI_COMM_WORLD`.

INFO CLUSTERDEF

Syntax: *<name>* [*<rank>*:*<pid>*]+

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>*

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>*

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.

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:

```
Copyright (c) 1992-2007 The University of Tennessee. All rights reserved.
```

```
$COPYRIGHT$
```

```
Additional copyrights may follow
```

```
$HEADER$
```

```
Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
```

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer listed in this license in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

```
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```