

---

# Intel® Trace Collector 7.0

## User's Guide

---

Copyright © Intel Corporation 1996-2006

All Rights Reserved

Revision: 7.0.3

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



---

## Disclaimer and Legal Information

The information in this manual is subject to change without notice and Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document. This document and the software described in it are furnished under license and may only be used or copied in accordance with the terms of the license. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. The information in this document is provided in connection with Intel products and should not be construed as a commitment by Intel Corporation.

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. Intel products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

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 software described in this document may contain software defects which may cause the product to deviate from published specifications. Current characterized software defects are available on request.

Intel, the Intel logo, Intel SpeedStep, Intel NetBurst, Intel NetStructure, MMX, i386, i486, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Celeron, Intel Centrino, Intel Xeon, Intel XScale, Itanium, Pentium, Pentium II Xeon, Pentium III Xeon, Pentium M, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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

Copyright © Intel Corporation 1996 - 2006.

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 also includes libraries developed and © by SGI and Michael Riepe. They are licensed under the GNU Lesser General Public License (LGPL) and their source code can be found in the "third\_party" directory.



# Contents

<b>Contents</b>	<b>I</b>
<b>1 Introduction</b>	<b>1</b>
1.1 What is the Intel® Trace Collector (ITC)?	1
1.2 System Requirements and Supported Features	1
1.3 Multithreading	3
1.4 About this Manual	3
<b>2 Installation</b>	<b>5</b>
<b>3 How to Use ITC</b>	<b>7</b>
3.1 Tracing MPI Applications	7
3.2 Tracing shmem Programs with ITC	12
3.3 Single-process Tracing	15
3.4 Tracing Application Subroutines	15
3.5 Tracing of Binaries	16
3.6 Multithreaded Tracing	18
3.7 Recording Statistical Information	18
3.8 Recording Source Location Information	19
3.9 Recording Hardware Performance Information	20
3.10 Recording OS Counters	22
3.11 Using the Dummy Libraries	22
3.12 Using the Shared Libraries	22
3.13 Tracing Library Calls	23
<b>4 Time Stamping</b>	<b>27</b>
4.1 Clock Synchronization	28
4.2 Choosing a Timer	28
<b>5 Java Tracing</b>	<b>37</b>



---

5.1	Features	37
5.2	Usage	37
5.3	API	38
<b>6</b>	<b>Tracing of Distributed Applications</b>	<b>41</b>
6.1	Design	41
6.2	Using VTserver	42
6.3	Running without VTserver	43
6.4	Spawning Processes	43
6.5	Tracing Events	44
6.6	Usage	44
6.7	Signals	44
6.8	Examples	44
<b>7</b>	<b>Structured Tracefile Format</b>	<b>47</b>
7.1	Introduction	47
7.2	STF Components	48
7.3	Single-File STF	49
7.4	Configuring STF	49
<b>8</b>	<b>User-level Instrumentation with the API</b>	<b>55</b>
8.1	The ITC API	55
8.2	Initialization, Termination and Control	57
8.3	Defining and Recording Source Locations	62
8.4	Defining and Recording Functions or Regions	65
8.5	Defining and Recording Overlapping Scopes	70
8.6	Defining Groups of Processes	71
8.7	Defining and Recording Counters	73
8.8	Recording Communication Events	75
8.9	Additional API Calls in libVTcs	80
8.10	C++ API	83
<b>9</b>	<b>ITC Configuration</b>	<b>91</b>
9.1	Configuring Intel(R) Trace Collector	91
9.2	Specifying Configuration Options	91
9.3	Configuration Format	91
9.4	Syntax of Parameters	92
9.5	Supported Directives	92

---



## CONTENTS

---

9.6 How to Use the Filtering Facility . . . . .	102
9.7 The Protocol File . . . . .	103
<b>A Copyright and Licenses</b>	<b>105</b>



# Chapter 1

## Introduction

### 1.1 What is the Intel® Trace Collector (ITC)?

The ITC tool for MPI applications produces tracefiles that can be analyzed with the Intel® Trace Analyzer (ITA) performance analysis tool. Some ITC versions are also able to trace non-MPI applications, like Java processes and socket communication in distributed applications. 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.

ITC is an add-on for existing MPI implementations; using it merely requires relinking the application with the ITC profiling library (see section 3.1.1). 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.

To define and trace user-defined events, or to use the profiling control functions, calls to the ITC API (see section 8) have to be inserted into the application's source code. This implies a recompilation of all affected source modules.

A special “dummy” version of the profiling libraries containing empty definitions for all ITC API routines can be used to “switch off” tracing just by relinking (see section 3.1.3).

### 1.2 System Requirements and Supported Features

Java tracing requires a virtual machine that supports the Java Virtual Machine Profiler Interface (JVMPi), which is not part of the Java platform yet. This extension is supported by several implementations already, though. Intel® Trace Collector does not support Java tracing on SGI Altix.

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 ITC 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 ITC will not work:

The following features are supported:



	<i>em64t/impv2</i>	<i>em64t/mpich</i>	<i>ia32/impv2</i>	<i>ia32/mpich</i>	<i>ia64/impv2</i>	<i>ia64/mpich</i>	<i>ia64/mpv</i>
<i>sizeof(MPI_Datatype)</i>	4	4	4	4	4	4	4
<i>sizeof(MPI_Comm)</i>	4	4	4	4	4	4	4
<i>sizeof(MPI_Request)</i>	4	8	4	4	4	8	4
<i>MPI_CHAR</i>	1275068673	1	1275068673	1	1275068673	1	1
<i>MPI_BYTE</i>	1275068685	3	1275068685	3	1275068685	3	27
<i>MPI_SHORT</i>	1275068931	4	1275068931	4	1275068931	4	2
<i>MPI_INT</i>	1275069445	6	1275069445	6	1275069445	6	3
<i>MPI_FLOAT</i>	1275069450	10	1275069450	10	1275069450	10	9
<i>MPI_DOUBLE</i>	1275070475	11	1275070475	11	1275070475	11	10
<i>MPI_COMM_WORLD</i>	1140850688	91	1140850688	91	1140850688	91	1
<i>MPI_COMM_SELF</i>	1140850689	92	1140850689	92	1140850689	92	2
<i>MPI_STATUS_SIZE</i>	5	4	5	4	5	4	6
<i>MPI_SOURCE</i>	8	4	8	4	8	4	0
<i>MPI_TAG</i>	12	8	12	8	12	8	4
<i>MPI_ERROR</i>	16	12	16	12	16	12	8





### 1.3. MULTITHREADING

Feature	Description
Thread-safety	supported, see <a href="#">1.3</a>
MPI tracing	<a href="#">3.1</a>
• IO	ROMIO, <a href="#">3.1.8</a>
• MPI One-Sided Communication	<a href="#">3.1.9</a>
• MPI-2	not supported
• fail-safe	supported ( <a href="#">3.1.7</a> )
• message checking	not supported
SHMEM tracing	<a href="#">3.2</a>
Java tracing	<a href="#">5</a>
Single-process tracing	<a href="#">3.3</a>
Tracing of Distributed Applications	<a href="#">6</a>
Subroutine tracing	<a href="#">3.4</a>
Tracing of Binaries without Recompilation	<a href="#">3.4</a>
Counter tracing	API in <a href="#">8.7</a>
Automatic Counter tracing of OS Activity	<a href="#">3.10</a>
Automatically Recording Source Location Information	<a href="#">3.8</a> (requires compiler support)
Manually Recording Source Location Information	API in <a href="#">8.3</a>
Recording Statistical Information	<a href="#">3.7</a>
Tracing Libraries at Different Levels of Detail	<a href="#">3.13</a>
Nonblocking Flushing	<a href="#">MEM-FLUSHBLOCKS</a>

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

Component	Usage
libVTnull	Dummy implementation of API ( <a href="#">3.11</a> )
libVT	MPI tracing ( <a href="#">3.1</a> ) SHMEM tracing ( <a href="#">3.2</a> )
libVTfs	fail-safe MPI tracing ( <a href="#">3.1.7</a> ) fail-safe SHMEM tracing ( <a href="#">3.2</a> )
libVTcs	Tracing of Distributed Applications and Single-processes ( <a href="#">6</a> , <a href="#">3.3</a> )
libVTjava	Java tracing ( <a href="#">5</a> ) (Distributed Applications and Single Processes)
VT_sample	Automatic Counter tracing with PAPI and getrusage() ( <a href="#">3.9</a> )
stftool	Manipulation of trace files ( <a href="#">7.4.1</a> )
xstftool/expandvtlog.pl	Conversion of trace files into readable format ( <a href="#">7.4.2</a> )
itcinstrument	Tracing of Binaries without Recompilation ( <a href="#">3.4</a> )

## 1.3 Multithreading

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

## 1.4 About this Manual

This manual describes how to use ITC. Some of the text is also provided as man pages for easier reading in a shell, for example, the ITC API calls (man [VT\\_enter](#)) and the ITC 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 ITC terms and names are hyperlinks that take you to the definition of the word. The documentation is platform-specific, which means that the text and even whole sections depend on which features are available or how they work on this platform. If you move between different platforms and something does not work as expected, ensure that you consult the correct documentation.



---

## Chapter 2

# Installation

After unpacking the ITC archive in a directory of your choice read the file 'relnotes.txt' for information on how to install Intel Trace Collector.

In order 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/suppport.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](mailto:tracetools@intel.com).

To acquire a demo license, use Intel Premier Support or contact [tracetools@intel.com](mailto: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). Sourcing the correct file in a shell (with `. itacvars.sh` resp. `source itacvars.csh`) will update `$LD_LIBRARY_PATH` and `$PATH` as well as set additional environment variables.

**VT.ROOT** points to the root of the ITC installation. It is used to find the ITC 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 directory containing the static ITC libraries. It might be useful to create Makefiles or shorter linkage commands when using '-L'.

**VT\_SLIB\_DIR** points to the directory containing the dynamic ITC libraries. It might be useful to create Makefiles or shorter linkage commands when using '-L'.

**VT\_ADD\_LIBS** lists libraries ITC is dependent on and needs to be linked against when using ITC.

**VT\_MPI** points to the currently selected MPI implementation as used by **VT\_LIB\_DIR** and **VT\_SLIB\_DIR**.

The itavars scripts accept an optional argument to specify the desired MPI implementation: impi2, mpich or mpt.

Default install options are `/opt/intel/itc_<platform>_<version>` as install directory and permissions which grant access for all users. `<platform>` includes the CPU type, operating system and often also a qualifier to distinguish versions for different MPIs. It is possible to install several different ITC packages in parallel on the same machine by using different directories. 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 ITC on a cluster of machines you can either install ITC 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 directory. Neither method has a clear advantage when it comes to runtime performance. Root privileges are only needed if writing into the desired install directory requires them.

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 ask the usual questions and install ITC, but in addition to that it will create a file called `itc_<platform>_<version>_SilentInstall.ini` in the current directory or, if that directory 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 ITC without further questions using the install options from that `.ini` file. Only error messages will be printed, all the normal progress messages are suppressed.

## Chapter 3

# How to Use ITC

### 3.1 Tracing MPI Applications

Using ITC for MPI is straightforward: relink your MPI application with the appropriate profiling library and execute it following the usual procedures of your system. This will generate a tracefile suitable for use with ITA, 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, you can instrument the application source code with calls to the ITC API (see section 8) and recompile. 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 ITC; if your MPI is different from the one ITC 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 ITC

Source files without calls to the ITC API can be compiled with the usual methods and without any special precautions.

Source files that do contain calls to the ITC API need to include the appropriate header files: VT.h for C and C++ and VT.inc for Fortran.

To compile these source files, the path to the ITC header files have to be passed to the compiler. On most systems, this is done with the -I flag, say with `-I$(VT.ROOT)/include`.

#### 3.1.2 Linking MPI Programs with ITC

ITC library libVT.a contains entry points for all MPI routines. They have to be linked against your application object files before your system's MPI library, which is achieved as follows:

```
mpicc ctest.o -lVT $(VT.ADD_LIBS) -o ctest
mpif77 ftest.o -lVT $(VT.ADD_LIBS) -o ftest
```

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 ITC library is listed on the command line in front of the MPI libraries. In general, the same ITC library and link line is suitable for all compilers and programming languages.

Dependent on the platform `$VT_ADD_LIBS` expands to the following

<b>em64t-lin-mpi2:</b>	<code>-ldwarf -lelf -lnsl -lm -ldl -lpthread</code>
<b>em64t-lin-mpich:</b>	<code>-ldwarf -lelf -lnsl -lm -ldl -lpthread</code>
<b>ia32-lin-mpi2:</b>	<code>-ldwarf -lelf -lnsl -lm -ldl -lpthread</code>
<b>ia32-lin-mpich:</b>	<code>-ldwarf -lelf -lnsl -lm -ldl -lpthread</code>
<b>ia64-lin-mpi2:</b>	<code>-ldwarf -lelf -lvtunwind -lnsl -lm -ldl -lpthread</code>
<b>ia64-lin-mpich:</b>	<code>-ldwarf -lelf -lvtunwind -lnsl -lm -ldl -lpthread</code>
<b>ia64-lin-mpt:</b>	<code>-ldwarf -lelf -lvtunwind -lnsl -lm -ldl -lpthread</code>

**Note:** When using Intel compilers, avoid explicitly linking against the math library. In this case, for better performance, it is recommended to *not* specify `'-lm'` and let the Intel compiler use its own and more performant 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 ITC 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 ITC 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 and Intel® C++ compiler 7.1** `-lmpiec`

**Intel® MPI and Intel® C++ compiler ≥ 8.0** `-lmpiic`

**Intel® MPI and gcc < 3.0** `-lmpigc`

**Intel® MPI and gcc ≥ 3.0 and < 3.4** `-lmpigc3`

**Intel® MPI and gcc ≥ 3.4** `-lmpigc4`

**mpich 1.2.x** `-lmpich++`

Another exception are Fortran compilers which are incompatible with the Intel® Fortran compiler that is used for compiling parts of libVT.a. The only system where such an incompatibility has been observed so far is the SGI Altix, where a segmentation fault occurs inside the MPT MPI startup code if Fortran code compiled with `ifort` is added to a Fortran binary which is linked with `g77`. As a workaround for this problem the relevant code is also provided as a library compiled with `g77`. It needs to be added to the link line like this:

```
mpif77 ftest.o -lVTg77 $VT_ADD_LIBS -o ftest
```

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

### 3.1.3 Running MPI Programs with ITC

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



### 3.1. TRACING MPI APPLICATIONS

---

**VT.CONFIG** contains the pathname of an ITC 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 ITC configuration file. The default value is 0. Setting a different value has no effects unless the MPI processes do not share the same filesystem.

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)** `<trace>.stf`

**single-file STF format** `<trace>.single.stf`

**old-style ASCII Vampir format** `<trace>.avt`

A directive in the configuration file (see section Configuration File 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 ITC 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` directory show how to instrument C and Fortran code to collect information about application subroutines. They come with a GNUmakefile that works for the MPI this ITC package was compiled for. If you use a different MPI, then you might have to edit this GNUmakefile. Unless ITC was installed in a private directory, the examples directory needs to be copied because compiling and running the examples requires write permissions.

#### 3.1.5 Trouble Shooting

If generating a trace fails, check first that you can run MPI applications that were linked without ITC. Then ensure that your MPI is indeed compatible with the one this package was compiled for, as described under section 1.2. The FAQ may have further information. If this still does not help, then please submit a report via the Question and Answer Database (QuAD).

#### 3.1.6 Handling of Communicator Names

By default ITC stores names for well-known communicators in the trace: "COMM\_WORLD", "COMM\_SELF\_#0", "COMM\_SELF\_#1", ... 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".

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

MPI\_Intercomm\_merge() is special because the new communicator is derived from two communicators, not just one as in the other functions. The name of the new inter-communicator will be “MERGE <old name 1>/<old name 2>” if the two existing names are different, otherwise it will be just “MERGE <old name>”.

In addition to these automatically generated names ITC 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 which is currently set in the old communicator when the new communicator is created.

ITC 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 ITC.

### 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
```

Under normal circumstances tracing works just like with libVT, but communication during trace file writing is done via 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 ITC 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”.





### 3.1. TRACING MPI APPLICATIONS

---

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 ITC 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 5 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.

#### 3.1.8 Tracing MPI File IO

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. For SGI's MPT (Altix) tracing of ROMIO is only supported if the application avoids ROMIO's non-standard request handles.

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 ITC 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.

#### 3.1.9 Tracing MPI One-Sided Communication

On SGI Altix (MPT) Intel® Trace Collector supports tracing a commonly used subset of the one-sided communication calls in MPI. Nothing needs to be done to enable this tracing, linking as described in 3.1.2 is sufficient. This table lists which functions are traced and which information is recorded:

Information	Functions
function entry and exit	MPI_Win_create() MPI_Win_free() MPI_Win_fence() MPI_Win_lock() MPI_Win_unlock()
function entry and exit, message with same duration as function and size giving the amount of transfered bytes	MPI_Put() MPI_Get() MPI_Accumulate()

## 3.2 Tracing shmem Programs with ITC

On Itanium® systems equipped with Quadrics Switches Intel® Trace Collector supports tracing of SHMEM calls which available through the Quadrics MPI implementation. Tracing programs that use shmem calls is very similar to tracing MPI programs. There is no need to change the source code of the program, relinking it with the ITC library is sufficient:

```
mpicc cshmem.o -lVT $(VT_ADD_LIBS) -lshmem -o cshmem
mpif77 fshmem.o -lVT $(VT_ADD_LIBS) -lshmem -o fshmem
```

ITC uses MPI calls internally, so linking requires the same libraries as in section 3.1.2 and the comments there about choosing the correct MPI still apply. Generating a trace also works as described above.

There are a few caveats:

**program name** If an unmodified shmem program is traced, ITC will initialize itself and MPI when `shmem_init()` is called. Inside this call ITC has no access to the program's arguments and in C there is no global variable or function to obtain this information, therefore the program name will be set to "UNKNOWN". By default, the tracefile will be called "UNKNOWN.stf". PC tracing (section 3.8) would not work because it needs access to the binary. There are several solutions:

- In Fortran, ITC can use global functions to obtain the program name and the user does not need to do anything.
- In C, The application can be changed so that it calls either `VT_initialize()` or `MPI_Init()` before `shmem_init()`, which will set the program name based on the arguments supplied to these functions.
- The environment variable `VT_PROGNAME` or the configuration file setting `PROGRAMME` can be used to pass the executable file name to ITC without changing the source code (see `VT_CONFIG` for details).

**dynamic linking** ITC replaces the shmem functions and calls the original implementation using the dynamic library loader. This means that statically linking the shmem library is not possible and that a shared version is required. This is not a problem because the shared shmem library is the default anyway.

ITC supports the following shmem functions (Fortran functions in uppercase, C in lowercase) and records the information listed in this table:



### 3.2. TRACING SHMEM PROGRAMS WITH ITC

Feature	Description
Information	Functions
start of program	SHMEM_INIT shmem_init
function entry and exit	SHMEM_MY_PE                   shmem_my_pe SHMEM_N_PES                   shmem_n_pes SHMEM_QUIET                   shmem_quiet SHMEM_WAIT                   shmem_wait
collective operation, function entry and exit	SHMEM_BARRIER               shmem_barrier SHMEM_BARRIER_ALL           shmem_barrier_all SHMEM_BROADCAST              shmem_broadcast SHMEM_BROADCAST4             shmem_broadcast32 SHMEM_BROADCAST8             shmem_broadcast64 SHMEM_INT4_MAX_TO_ALL       SHMEM_INT8_MAX_TO_ALL SHMEM_INT4_MIN_TO_ALL       SHMEM_INT8_MIN_TO_ALL SHMEM_INT4_OR_TO_ALL         SHMEM_INT8_OR_TO_ALL SHMEM_INT4_PROD_TO_ALL       SHMEM_INT8_PROD_TO_ALL SHMEM_INT4_SUM_TO_ALL        SHMEM_INT8_SUM_TO_ALL SHMEM_INT4_XOR_TO_ALL        SHMEM_INT8_XOR_TO_ALL SHMEM_REAL4_MAX_TO_ALL       SHMEM_REAL8_MAX_TO_ALL SHMEM_REAL4_MIN_TO_ALL       SHMEM_REAL8_MIN_TO_ALL SHMEM_REAL4_PROD_TO_ALL      SHMEM_REAL8_PROD_TO_ALL SHMEM_REAL4_SUM_TO_ALL       SHMEM_REAL8_SUM_TO_ALL shmem_double_max_to_all       shmem_float_max_to_all shmem_double_min_to_all       shmem_float_min_to_all shmem_double_prod_to_all      shmem_float_prod_to_all shmem_double_sum_to_all       shmem_float_sum_to_all shmem_longdouble_max_to_all shmem_longdouble_min_to_all shmem_longdouble_prod_to_all shmem_longdouble_sum_to_all shmem_short_and_to_all         shmem_int_and_to_all shmem_short_max_to_all         shmem_int_max_to_all shmem_short_min_to_all         shmem_int_min_to_all shmem_short_or_to_all         shmem_int_or_to_all shmem_short_prod_to_all        shmem_int_prod_to_all shmem_short_sum_to_all         shmem_int_sum_to_all shmem_short_xor_to_all         shmem_int_xor_to_all shmem_long_and_to_all         shmem_longlong_and_to_all shmem_long_max_to_all         shmem_longlong_max_to_all shmem_long_min_to_all         shmem_longlong_min_to_all shmem_long_or_to_all         shmem_longlong_or_to_all shmem_long_prod_to_all         shmem_longlong_prod_to_all shmem_long_sum_to_all         shmem_longlong_sum_to_all shmem_long_xor_to_all         shmem_longlong_xor_to_all
single message, function entry and exit	SHMEM_CHARACTER_GET         SHMEM_CHARACTER_PUT SHMEM_DOUBLE_GET             SHMEM_DOUBLE_PUT SHMEM_DOUBLE_IGET            SHMEM_DOUBLE_IPUT SHMEM_INTEGER_GET             SHMEM_INTEGER_PUT

Feature	Description
	SHMEM_INTEGER_IGET SHMEM_INTEGER_IPUT
	SHMEM_GET SHMEM_PUT
	SHMEM_GET32 SHMEM_PUT32
	SHMEM_GET64 SHMEM_PUT64
	SHMEM_GET128 SHMEM_PUT128
	SHMEM_GET4 SHMEM_PUT4
	SHMEM_GET8 SHMEM_PUT8
	SHMEM_IGET SHMEM_IPUT
	SHMEM_IGET32 SHMEM_IPUT32
	SHMEM_IGET64 SHMEM_IPUT64
	SHMEM_IGET128 SHMEM_IPUT128
	SHMEM_IGET4 SHMEM_IPUT4
	SHMEM_IGET8 SHMEM_IPUT8
	SHMEM_IXGET SHMEM_IXPUT
	SHMEM_IXGET4 SHMEM_IXPUT4
	SHMEM_REAL_GET SHMEM_REAL_PUT
	SHMEM_REAL_IGET SHMEM_REAL_IPUT
	shmем_double_get shmем_double_put
	shmем_double_iget shmем_double_iput
	shmем_float_get shmем_float_put
	shmем_float_iget shmем_float_iput
	shmем_get shmем_put
	shmем_get32 shmем_put32
	shmем_get64 shmем_put64
	shmем_get128 shmем_put128
	shmем_getmem shmем_putmem
	shmем_iget shmем_iput
	shmем_iget32 shmем_iput32
	shmем_iget64 shmем_iput64
	shmем_iget128 shmем_iput128
	shmем_int_get shmем_int_put
	shmем_int_iget shmем_int_iput
	shmем_ixget shmем_ixput
	shmем_ixget32 shmем_ixput32
	shmем_long_get shmем_long_put
	shmем_long_iget shmем_long_iput
	shmем_longdouble_get shmем_longdouble_put
	shmем_longdouble_iget shmем_longdouble_iput
	shmем_longlong_get shmем_longlong_put
	shmем_longlong_iget shmем_longlong_iput
	shmем_short_get shmем_short_put
	shmем_short_iget shmем_short_iput
	shmем_double_g shmем_double_p
	shmем_float_g shmем_float_p
	shmем_int_g shmем_int_p
	shmем_long_g shmем_long_p
	shmем_short_g shmем_short_p



### 3.3. SINGLE-PROCESS TRACING

Feature	Description	
pairwise message exchange, function entry and exit	SHMEM_INT4_SWAP	SHMEM_INT8_SWAP
	SHMEM_REAL4_SWAP	SHMEM_REAL8_SWAP
	SHMEM_SWAP	
	shmem_double_swap	shmem_float_swap
	shmem_int_swap	shmem_long_swap
	shmem_longlong_swap	shmem_short_swap
	shmem_swap	
	SHMEM_INT4_FADD	SHMEM_INT8_FADD
	shmem_int_fadd	shmem_long_fadd
	shmem_longlong_fadd	shmem_short_fadd

Different tags are used for messages associated with shmem function calls. This can be used to filter for specific messages:

Function Call	Value of Tag
Put	100000
Get	100001
Swap	100002
Add	100003

## 3.3 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 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 ([3.4](#)) or binary instrumentation ([3.4](#)) can be used with and without further ITC API (see [chapter 8](#)) calls to actually generate trace events.

libVTcs uses the same techniques as fail-safe MPI tracing ([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.

## 3.4 Tracing Application Subroutines

Function tracing is always possible when using 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.

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 3.13](#).

## 3.5 Tracing of Binaries

### Synopsis

```
itcinstrument --input <executable> <options>
              --help
              --version
```

### Description

The itcinstrument utility program manipulates a binary executable file. It can:

- insert an ITC library into the binary as if the executable had been linked against it
- insert code into the executable which records function entry and exit events, thus allowing more detailed analysis of the user code in an application

Without further options itcinstrument will just analyze the executable 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 producing a modified executable. In C++ names are demangled automatically, but only if they follow the current standard which is used by GCC 3.x and newer.

A modified executable is generated only if the "--output" option is given. Without further options, itcinstrument will just insert libVT into an MPI application. If the application is not an MPI application, you need to choose which library to insert with the "--insert" option. Invoking itcinstrument on the binary will print a list of all available libraries with a short description of each one. The ITC documentation also has a full list of all available functions 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 itcinstrument 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 itcinstrument will hide functions that are internal to those libraries and thus cannot be traced.

The optional function profiling is enabled with the "--profile" flag. Limiting the number of instrumented functions is recommended to avoid excessive runtime overhead and 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 or recorded trace data dynamically at runtime; see the section "Tracing Library Calls" in the ITC documentation for details.

### API calls

In order to trace non-MPI applications the applications have to contain calls to `VT_initialize()` and `VT_finalize()` already, to initialize tracing and generate a trace. It is possible to link the binary against libVTnull, the library which provides dummy implementations of all API functions.

itcinstrument will intercept all of the API calls and redirect them into the tracing library which is used by the instrumented binary.

### Installation



### 3.5. TRACING OF BINARIES

---

To run itcinstrument, source the ITC source scripts; they ensure that the required environment variables are set correctly. This is not necessary to run the instrumented binary: although it needs some additional shared libraries, the search path for them gets inserted into the binary itself.

This works as long as these shared libraries are installed in the same directory on all machines where the binary is used. If that is not the case, then you have two options:

- install ITC and ensure that the source script is included before running the instrumented binary
- copy the .so files from ITC's slib directory to the target machines together with the instrumented binary and include this directory in the LD\_LIBRARY\_PATH

#### Restrictions

itcinstrument is available only on IA32 and EM64T.

Note that functions are instrumented, not the location where they are called. This implies that functions found in shared libraries currently cannot be traced. This will be added in later versions of itcinstrument.

Instrumenting static binaries is not supported. The MPI libraries may be linked statically, but in order to insert libVT it is necessary to specify the location of the MPI's shared library with --mpi because libVT needs to call functions contained (perhaps only) in them.

#### Supported Directives

##### --input

**Syntax:** <filename>

Specifies the executable which is to be instrumented or analyzed.

##### --output

**Syntax:** <filename>

Specifies the name of the instrumented executable which shall be generated by itcinstrument.

##### --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 larger than 2 may contain output that only makes sense to the developers of ITC.

##### --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>:\*".

## 3.6 Multithreaded Tracing

To trace multithreaded applications, just link and run as described above. Additional threads will be registered automatically as soon as they call ITC via 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.

## 3.7 Recording Statistical Information

ITC is able to gather and store statistics about the following items:

- function calls
- sent messages
- collective operations

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 unmanagable trace. To run an application in this mode you can either set the environment variables [VT\\_STATISTICS](#) and [VT\\_PROCESS](#) 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 trace in a machine-readable format, but also into the protocol (.prot) file in ASCII format. If the protocol file should ever get lost, then the stftool (see section [7.4.1](#)) can convert from the machine-readable format to ASCII text with the same format as in the protocol file with `--print-statistics`.





### 3.8. RECORDING SOURCE LOCATION INFORMATION

This format was chosen so that text processing programs and scripts such as awk, perl, and Excel can read it. For each type of statistic, the data for each process resp. pair of processes (for messages) is contained in a consecutive block of lines. Beware that ITC is not able to gather statistics by thread: if the application is multithreaded, statistics are still aggregated by process.

A distinctive tag starts each one. The following table describes the data in the protocol file:

Type	Tag	Organization	Available data
Routines	ACTSTATS	By process	Number of calls Minimum execution time (exclusive/inclusive) Maximum execution time (exclusive/inclusive) Total execution time (exclusive/inclusive)
Messages	MSGSTATS	By sending/receiving process	Number of messages Total number of bytes Minimum and maximum size

Within each line, colons separate fields (:). For the three types of statistics, the format is as follows:

Type	Format
Routines	<act>:<sym>:<pid>:<count>: <minexcl>:<maxexcl>:<totalexcl>: <minincl>:<maxincl>:<totalincl>
Messages	<source>:<target>:<count>:<minsize>:<maxsize>:<totalsize>

The fields above have the following definitions:

Field	Description	Type	Units
<act>	Activity name	String	
<sym>	Symbol name	String	
<pid>	MPI task rank	Integer	
<count>	Number of invocations/messages	Integer	
<min/max/totalexcl>	Minimum, maximum, total execution time excluding called routines	Floating Point	Seconds
<min/max/totalincl>	Minimum, maximum, total execution time including called routines	Floating Point	Seconds
<minsize>, <maxsize>	Minimum, maximum message size	Integer	Bytes
<totalsize>	Sum of message sizes	Integer	Bytes
<mintime>, <maxtime>	Minimum, maximum execution time	Floating Point	Seconds
<totaltime>	Total execution time	Floating Point	Seconds

Filter utilities, such as awk and perl, and plotting/spreadsheet packages, like Excel, can process the statistical data easily. In the examples directory an awk script called convert-stats is provided that illustrates how the values in the protocol file 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.

## 3.8 Recording Source Location Information

To record the locations of subroutine calls in the source code automatically, compile the relevant application modules with support for debugging. To do this, use these compiler flags that enable the generation of debug information for ITC:



```
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 e.g.:

```
# 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 ITC API can provide similar information but without the performance overhead (see `VT_scldef()/VT_thisloc()` in section 8.3 for more information).

### 3.8.1 Notes for IA32 and EM64T

On ia32-Linux and EM64T-Linux 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 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 ITC 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`. Support by other compilers for both features is unknown.

## 3.9 Recording Hardware Performance Information

ITC 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 ITC.

The `VT_sample.o` file can be rebuilt by entering the ITC 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.



### 3.9. RECORDING HARDWARE PERFORMANCE INFORMATION

Add the `VT_sample.o` object file to the link line in front of the ITC 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—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 ITC 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. Section 9 describes how to use configuration options.

However, enabling all counters at once is usually a bad idea because logging counters not required for the analysis 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 f.i. because a function runs for a very long time, give ITC 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. ITC 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:

Counter Class: OS		
Counter Name	Unit	Comment
RU_ETIME	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
```

### 3.10 Recording OS Counters

Similar to the process specific counters in the previous section, ITC 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:

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

### 3.11 Using the Dummy Libraries

Programs containing calls to the ITC API (see section 8) 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 ITC library directory. Here’s how a C MPI-application would be linked:

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

### 3.12 Using the Shared Libraries

This version of the ITC also provides all of its libraries as shared objects. They are placed in the “slib” instead of the “lib” directory so that the linker still picks up the normal static libraries by default. Using the static libraries is easier to handle, but in some cases the shared libraries might be useful. They are not officially supported, though.

To use the shared libraries, add the “slib” directory to the command line of your linker. Then ensure that your LD\_LIBRARY\_PATH includes this directory *on all nodes* where the program is

### 3.13. TRACING LIBRARY CALLS

started. This can be done either by automatically sourcing the ITC 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.

On Linux two different bindings for Fortran are supported in the same library. This works fine when linking against the static ITC 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, f.i. 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 the ITC 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"`.

### 3.13 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.

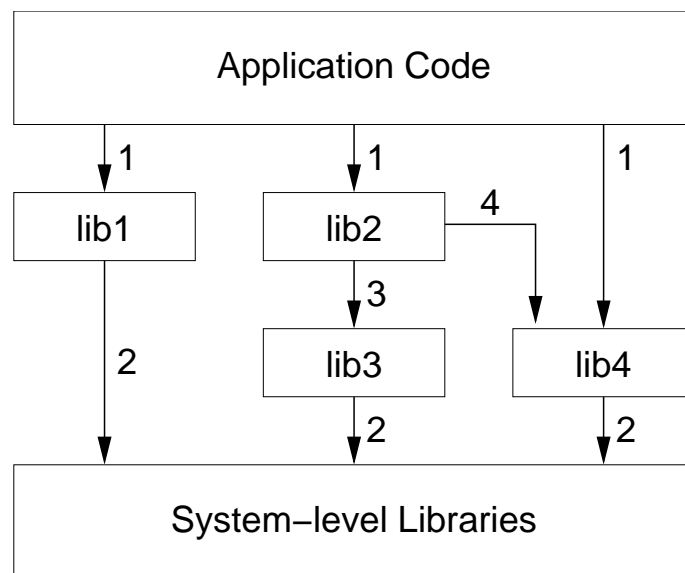


Figure 3.1: General structure of an application using many different libraries.

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

- 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 her.

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(R) 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(R) 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
```



### 3.13. TRACING LIBRARY CALLS

---

```
|  +->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.



## Chapter 4

# Time Stamping

ITC assigns a local time stamp to each event that 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, ITC artificially increments the clock tick. When running ITC 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 ITC uses 51 bits for clock ticks, which is large enough to count  $2^{51}$ ns, which equals  $2^{51}/1e9/60/60/24 \geq 26$  days before the counter overflows. At the same time with a clock of only ms resolution, you can distinguish  $2^{64-51} = 8192$  different events with the same clock tick, which are events with a duration of 0.1us.

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.

## 4.1 Clock Synchronization

By default ITC 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`.

ITC 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 ITC 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 ITC's 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 )
```

### **Java**

```
void timeSync( )
```

## 4.2 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



## 4.2. CHOOSING A TIMER

---

- 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

ITC 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`” one a machine with a good `gettimeofday()` clock:

```
bash$ VT_TIMER=gettimeofday ./timertestcs
performance: 2323603 calls in 5.000s wall clock time = 2.152us/call =
              464720 calls/s
measured clock period/frequency vs. nominal:
  1.000us/1.000MHz vs. 1.000us/1.000MHz
overhead for sampling loop: 758957 clock ticks (= 758.958ms)
  for 10000000 iterations = 0 ticks/iteration
average increase: 2 clock ticks = 2.244us = 0.446MHz
median increase: 2 clock ticks = 2.000us = 0.500MHz
<    0 ticks = 0.00s : 0
<    1 ticks = 1.00us: 0
>=   1 ticks = 1.00us: ##### 2261760
>=  501 ticks = 501.00us: 1
>= 1001 ticks = 1.00ms: 0
...
```

The additional information at the top starts with the performance (and thus overhead) of the timer. The next line compares the measured clock period (calculated as elapsed wall clock time divided by clock ticks in the measurement interval) against the one that the timer is said to have; for `gettimeofday()` this is not useful, but for example `CPU` cycle counters (details below) there might be differences. Similarly, the overhead for an empty loop with a dummy function call is only relevant for a timer like `CPU` cycle counters with a very high precision. For that counter however the



overhead caused by the loop is considerable, so during the measurement of the clock increments ITC 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 ITC 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
```



## 4.2. CHOOSING A TIMER

---

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 ITC 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 behaviour 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 ITC 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 ITC'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 ITC are shown in two different ways: all samples and just those actually used by ITC 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 be as a deviation f.i. below the x-axis and in the other above it.

Also, the later two graphs are only useful if ITC 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 ITC to not use those results and thus simulates a default application run where synchronization is only done at the start and end.

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 4.1 shows that in general ITC managed to keep the test results inside a range of plus/minus 1us although it did not use all the information collected with `VT_timesync()`. The clock transformation function in figure 4.2 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 16us. Also, the constrained spline interpolation is superior compared to a simple linear interpolation between the sample points (figure 4.3).

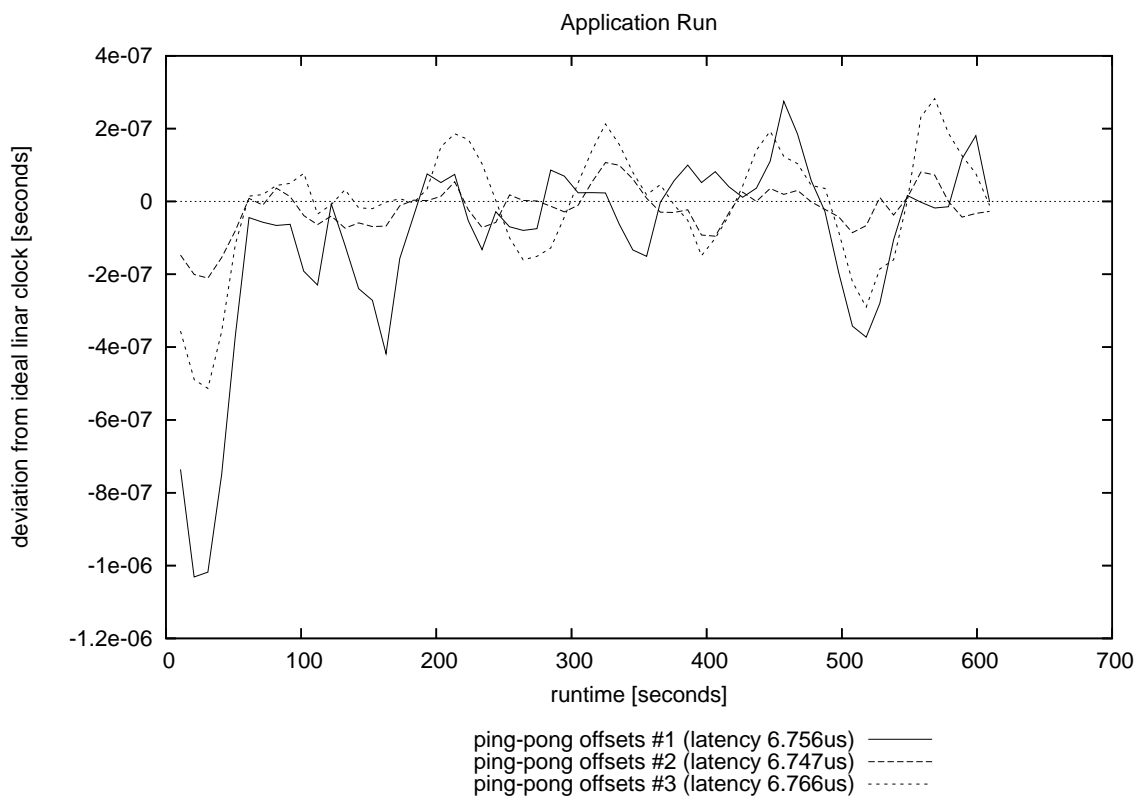


Figure 4.1: CPU timer: application run with non-linear clock correction.

## 4.2. CHOOSING A TIMER

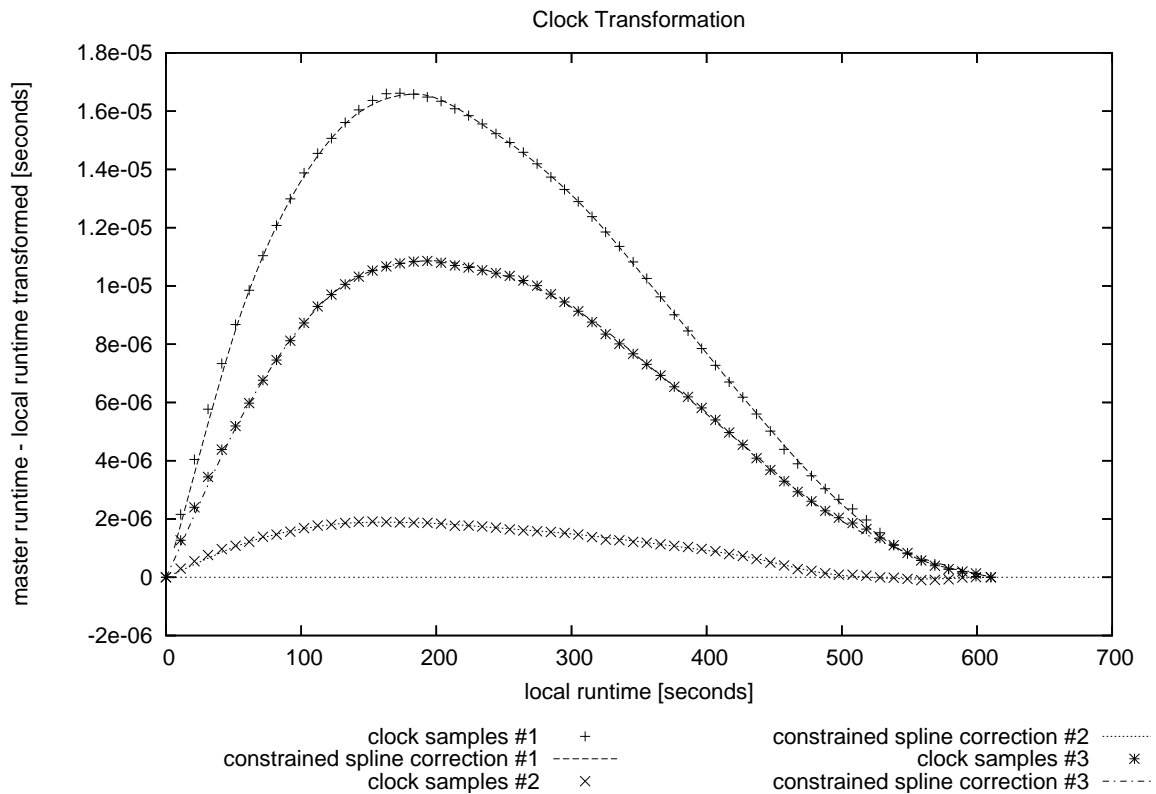


Figure 4.2: CPU timer: clock transformation and the sample points it is based on.

### 4.2.1 gettimeofday

This is the default timer. Its API limits the clock resolution to 1us, but depending on which timer the OS actually uses the clock resolution may be much lower. 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 4.4 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. 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 4.5). However, NTP alone without the additional intermediate synchronization would have led to deviations of nearly 70us.

So the recommendation is to enable NTP, but intermediate clock synchronization by ITC is still needed to achieve good results.

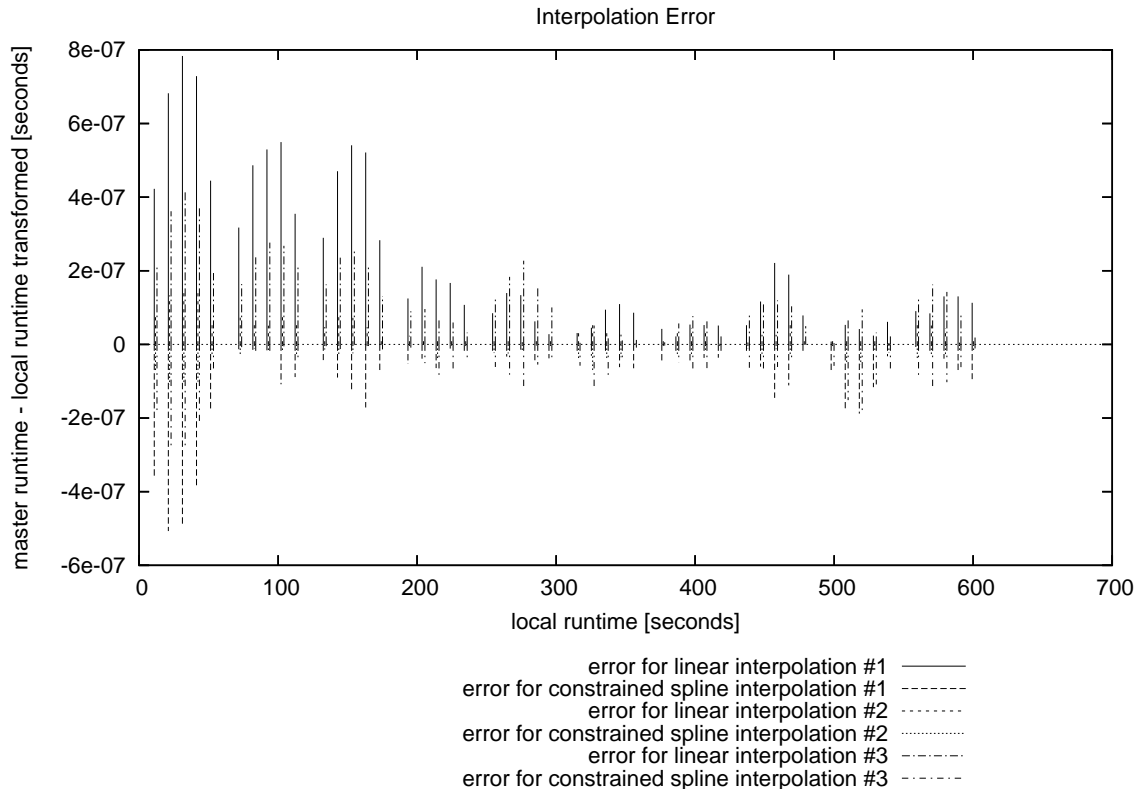


Figure 4.3: CPU timer: error with linear (above x-axis) and non-linear interpolation (below).

## 4.2.2 CPU Cycle Counter (TSC/ITC)

This is a high-resolution counter inside the CPU which counts CPU cycles. This counter is called Timer Stamp Counter (TSC) on x86/EM64T architectures and Interval Time Counter (ITC) on Itanium®. It can be read via 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 4.2.

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 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-FLUSH-BLOCKS and “Recording OS Counters”, 3.10). If the application already is single-threaded, then the additional ITC threads could be disabled to avoid this complication.



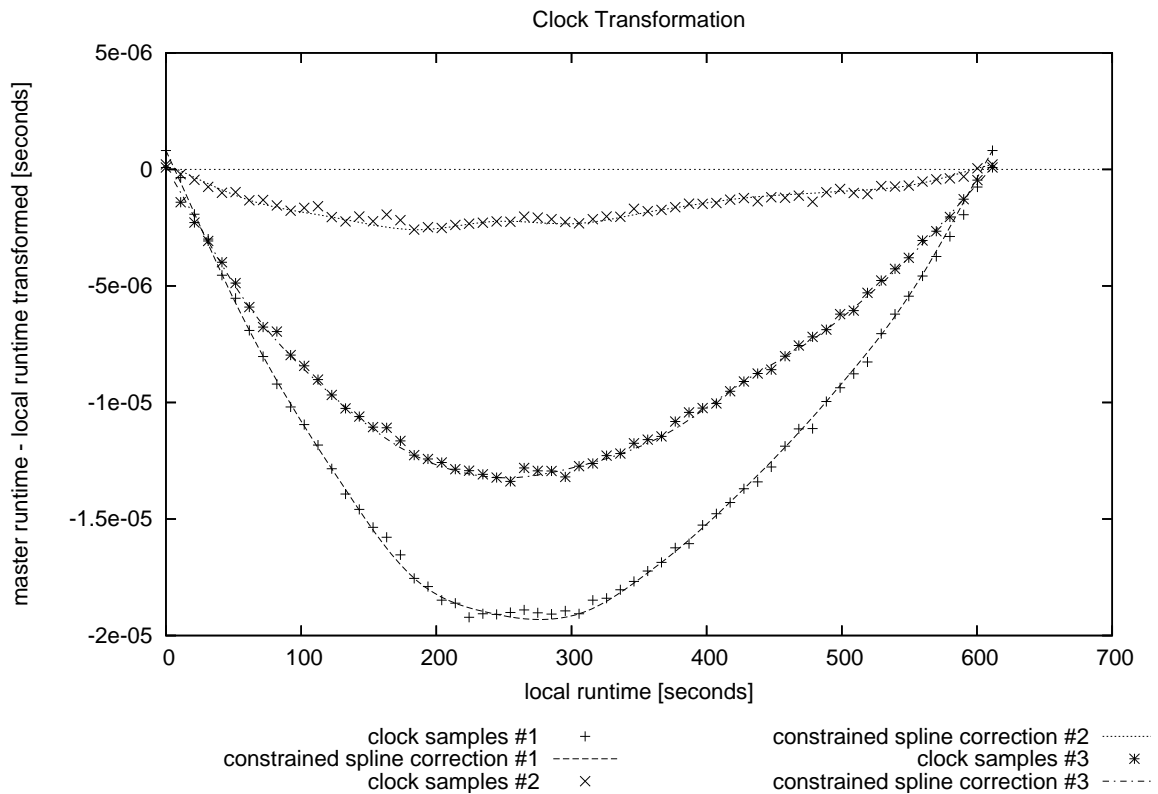


Figure 4.4: gettimeofday() without NTP.

**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.

### 4.2.3 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() ITC 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 via the MPI API and thus ITC 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.

### 4.2.4 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 ITC does not support it yet.

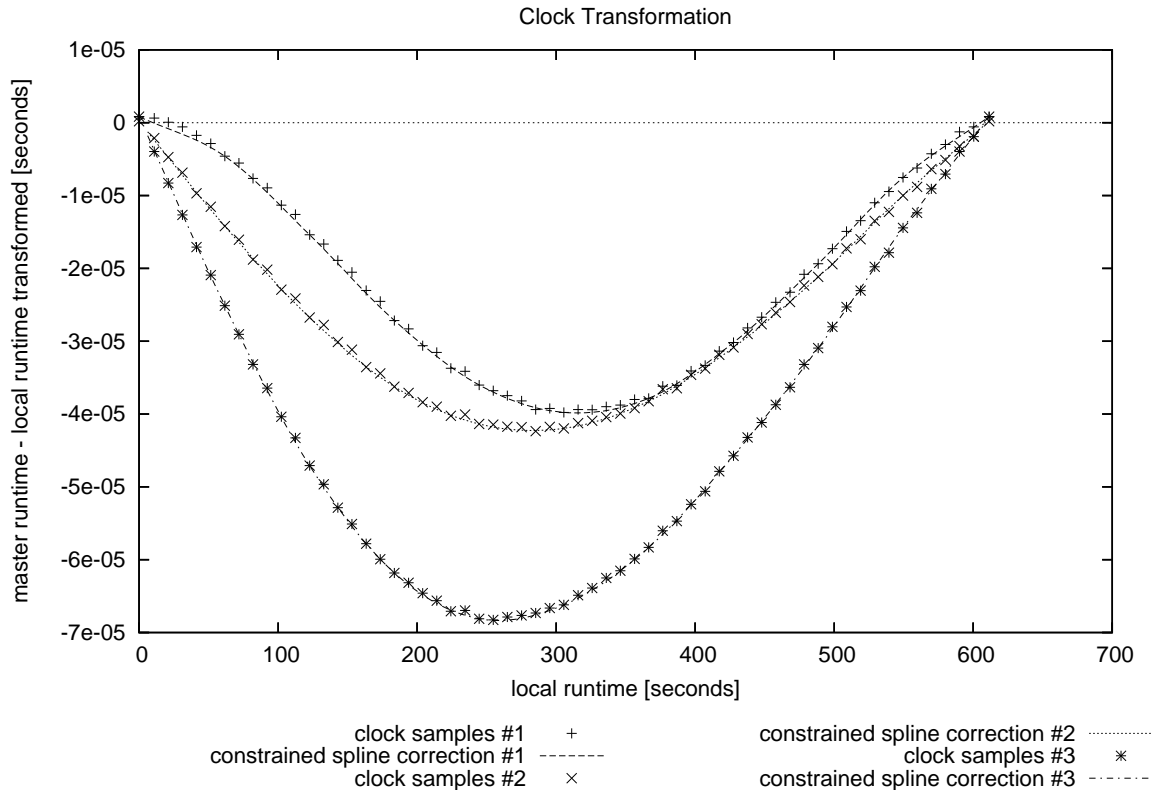


Figure 4.5: `gettimeofday()` with NTP.

#### 4.2.5 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 via this API than through `gettimeofday()`. ITC currently does not support this API.

## Chapter 5

# Java Tracing

### 5.1 Features

Function tracing records all calls to Java or native functions. This is possible without having to modify the Java application in any way by utilizing the Java Virtual Machine Profiler Interface (JVMPi).

All of the ITC API calls described in section 8 are also available and can be used with and without function tracing to log custom events or to mark special code regions. Multiple threads are supported and source code locations in the Java source code can be recorded.

Future extensions might include tracing of:

- monitor operations
- memory management

### 5.2 Usage

Load ITC upon startup of the virtual machine to intercept information about function calls. This is accomplished in Sun compatible Java virtual machines (JVMs) with the following command line options:

```
-XrunVTjava
```

The `LD_LIBRARY_PATH` has to include the `$(VT_ROOT)/slib` directory or the JVM will complain about not being able to find `libVTjava.so`.

ITC will be activated automatically before any user thread is created, and thus will be able to trace the startup of the application, too.

Configuration is done normally by setting `VT_CONFIG` to the name of a config file or other VT environment variables directly, as described in chapter 9. Because there is no unique application name in Java, the default name for the generated trace is "`<name of JVM binary>.stf`" (usually "`java.stf`") and using the `VT_LOGFILE_NAME` variable to override it is advisable.

A full function trace of a Java program can get very large, because of many small functions in the standard classes. Folding these calls so that only the top-level function call is traced helps a lot. This is done with the following entries in a `VT_CONFIG` file:



```
ACTIVITY * FOLD
ACTIVITY "app class*" UNFOLD
```

Replace “app class” with the name of a user class that is to be traced, because they are most likely called from class loader functions. Without such an entry ITC would hide these calls just like all the others done by the standard classes.

By default, function tracing is enabled when starting ITC via `-XrunVTjava`. To disable function tracing, you can use this configuration option:

```
JAVA OFF
```

In both cases all Java threads are recorded under the names used for them by the Java runtime system. In contrast to for example MPI tracing their creation and termination time is not recorded as entering resp. leaving “User\_Code”, because that would be misleading: Java threads enter some system code first. Instead, use function tracing to learn which code is actually executed by a thread.

## 5.3 API

The basic ITC API is made available in Java as static member functions of the class `com.intel.tracecollector.VT`. In order to use this class, include `$(VT_ROOT)/lib` in the `CLASSPATH` and `LD_LIBRARY_PATH`. Initialize ITC either at startup of the virtual machine (as described in the previous section) or with explicit calls to the `VT_initialize()` function.

For tracing of just one process, import this function and the matching `VT_finalize()` from the `com.intel.tracecollector.VTcs` class. In previous releases there used to be a different implementation in `com.pallas.vampirtrace.VTsp` for tracing of single processes. Now this functionality is a subset of what VTcs provides: when calling `VT_initialize()` directly, just the current process is traced. When using the initialization API for distributed tracing, more than one process can contribute to one trace. This is described in section 6.

In contrast to the C and Fortran version no error codes are returned. Instead the result of the function is returned and a `java.lang.Error` exception thrown in case of an error, which should never happen unless the application is using ITC in the wrong way, without having initialized it properly, or a fatal error occurred, so not catching this is legitimate.

The names of the functions have been adapted to the Java naming conventions and all constants are defined as static final members of the VT class. They are listed in the detailed function descriptions given in chapter 8.

Here is an example of a very simple Java program that uses the ITC API:

```
import com.intel.tracecollector.VT;
import com.intel.tracecollector.VTcs;

public class javaapi {
    public static void main(String[] args) {
        int clazz, function, i;
        byte data[] = new byte[10];

        try {
            VT.classDef( "VT not initialized yet" );
        } catch( Error ex ) {
```

```
        System.out.println( ex );
    }

    VTcs.initialize();

    clazz = VT.classDef( "Java API" );
    function = VT.funcDef( "main", clazz );
    VT.begin( function );
    VT.end( function );

    for( i = 0; i < 10; i++ ) {
        data[i] = (byte)i;
    }
    VT.logData( data, VT.NOSCL );

    VTcs.fini();
}
}
```



## Chapter 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 ITC's communication patterns. Therefore a special version of the ITC 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 ITC 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 ITC. 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.

ITC 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 VT server, 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 ITC 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 via 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.





### 6.3. RUNNING WITHOUT VTSERVER

---

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_spawn()` 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_spawn()` 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 VT 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 ITC, 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 ITA tool.

## 6.6 Usage

Executables in the application are linked with `-IVTcs` and the same additional parameters as listed in section 3.3. 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 in the chapter about `VT_CONFIG`.

## 6.7 Signals

`libVTcs` uses the same techniques as fail-safe MPI tracing (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.

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 ITC library, tracing of MPI has to be done with VT API calls.



## 6.8. EXAMPLES

---

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\_spawn(). This is repeated recursively. Communication is done via 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.



## Chapter 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 ITA 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.

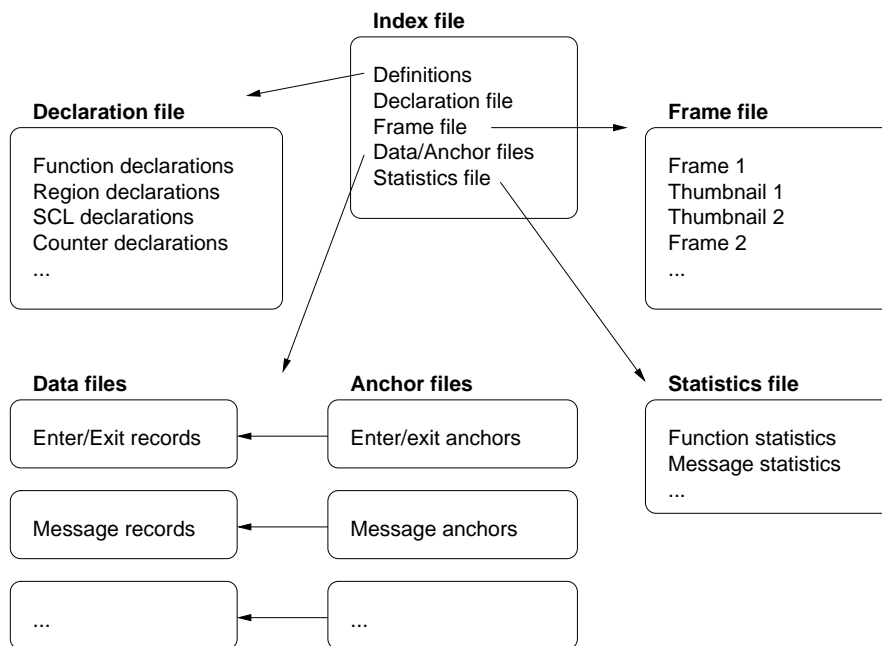


Figure 7.1: STF components

## 7.2 STF Components

A structured tracefile actually consists of a number of files as shown in the figure 7.1. Depending on the 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 global operation file with the name `<trace>.stf.gop`
- one or more process files with the name `<trace>.stf.pr.<index>`
- 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 ITA to “fast-forward” within the record files; they can be deleted, but that may result in slower operation of ITA.

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 section 7.4.1).

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 ITC, 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 ITA 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 ITC configuration file or the equivalent environment variables are also described in [section 9](#).

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 behaviour of `stftool` is configured similarly to ITC: with a config file, environment variables and command line options. The environment variable [VT.CONFIG](#) can be set to the name of a ITC 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-threads

**Syntax:**

**Default:** off

Prints information about each native thread that was encountered by ITC when generating the trace.

### --print-errors

**Syntax:**

**Default:** off

Prints the errors that were found in the application.

### --dump

**Syntax:**





## 7.4. CONFIGURING STF

---

**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]

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 ITA to analyse the data without loading all of it, so it is more scalable. Writing it is only supported by ITC 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 ITC, on in stftool

Several events can only be stored in STF, but not in VTF. ITC 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 ITA 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 ITA.

**--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 larger than 2 may contain output that only makes sense to the developers of ITC.

**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 expandvtflog.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
```



## Chapter 8

# User-level Instrumentation with the API

### 8.1 The ITC API

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

**enum \_VT\_ErrorCode**

error codes returned by ITC API.

**Enumeration values:**

**VT\_OK** OK.

**VT\_ERR\_NOLICENSE** no valid license found.

**VT\_ERR\_NOTIMPLEMENTED** Not (yet?) implemented.

**VT\_ERR\_NOTINITIALIZED** Not initialised.

**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** ITC API called while inside an ITC 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
```



## 8.2. INITIALIZATION, TERMINATION AND CONTROL

---

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

ITC 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 ITC is terminated. Thus, it is an error to call ITC 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 ITC explicitly. These calls also help to write programs and libraries that use VT without depending on MPI.

**int VT\_initialize (int \* *argc*, char \*\*\* *argv*)**

Initialize ITC 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 ITC 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 e.g. 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 ITC itself.

ITC 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 (e.g. `-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 ITC arguments on the command line.

**Fortran**

```
VTINIT( ierr )
```

**Java**

```
void initialize( )
```

**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 ITC and underlying communication.

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

**Fortran**

```
VTFINI( ierr )
```

**Java**

```
void fini( )
```

**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 )
```

**Java**

```
int getRank()
```

**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 ITC 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 ITC 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.

**Java**

```
void registerThread( int thindex )
```

**Parameters:**

**thindex** thread number, only used if  $\geq 0$

**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](#) - ITC wasn't initialized yet

**int VT\_registernamed (const char \* threadname, int thindex)**

Registers a new thread with ITC 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\\_registerthread\("new name", -1\)](#) if no name has been set before.

**Java**

```
void registerThread( final String threadname, int thindex )
```

**Parameters:**

**threadname** desired name of the thread, or NULL/empty string if no name wanted

**thindex** desired thread number, pass negative number to let ITC pick a number

**Returns:**

error code, see [VT\\_registerthread\(\)](#)

**int VT\_getthrank (int \* *thrank*)**

Get thread index within process.

Either assigned automatically by ITC or manually with [VT\\_registerthread\(\)](#).

**Fortran**

VTGETTHRANK( *thrank*, *ierr* )

**Java**

int getThreadRank()

**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( )

**Java**

void traceOff()

**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**

VTTTRACEON( )

**Java**

void traceOn()

**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 via config option "[PROCESS/CLUSTER NO](#)" for the whole process and cannot be changed.

**Fortran**

```
VTTTRACESTATE( state, ierr )
```

**Java**

```
int traceState()
```

**Return values:**

**state** is set to current state

**Returns:**

error code

With the ITC 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 )
```

**Java**

```
int symState( int statehandle )
```

**Parameters:**

**statehandle** result of [VT\\_funcdef\(\)](#) or [VT\\_symdef\(\)](#)

**Return values:**

**on** set to 1 if symbol is active

**Returns:**

error code

ITC 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, ITC 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 )

**Java**

void flush( )

**Returns:**

error code

Refer to section 9 to learn about the [MEM-BLOCKSIZE](#) and [MEM-MAXBLOCKS](#) configuration directives that control ITC's memory usage.

ITC 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( )

**Java**

double timeStamp( )

**double VT\_timestart (void)**

Returns point in time in seconds when process started, or [VT\\_ERR\\_NOTINITIALIZED](#).

**Fortran**

DOUBLE PRECISION VTTIMESTART( )

**Java**

double timeStart( )

## 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 \* *schandle*)**

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

**Fortran**

VT\_SCLDEF( file, line\_nr, schandle, ierr )

**Java**

int sclDef( final String file, int line\_nr )

**Parameters:**

***file*** file name

***line\_nr*** line number in this file, counting from 1

**Return values:**

***schandle*** the int it points to is set by ITC

**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, ,
```

**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 EM64T in the RBP register, on Itanium(R) in the BSP 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 ITC to a valid SCL handle in case of success and **VT\_NOSCL** otherwise

**Returns:**

error code

ITC 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 ITC API calls.

**int VT\_thisloc (int sclhandle)**

Set source code location for next activity that is logged by ITC.

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

**Fortran**

VTTHISL( sclhandle, ierr )

**Java**

void thisLoc( int sclhandle )

**Parameters:**

**sclhandle** handle defined either with [VT\\_scldef\(\)](#)

**Returns:**

error code

## 8.4 Defining and Recording Functions or Regions

ITA 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. ITA 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 MPI\_SEND for Fortran.

The ITC 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\\_beginl\(\)/VT\\_endl\(\)](#) routines.

### 8.4.1 New Interface

To simplify the use of user-defined states, a new interface has been introduced for ITC. 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 f.i. 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 )
```

**Java**

```
int classDef( final String classname )
```

**Parameters:**

***classname*** name of the class

**Return values:**

***classhandle*** the int it points to is set by ITC

**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 )
```

**Java**

```
int funcDef( final String symname, int classhandle )
```

**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 ITC

**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.

ITC 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.

**Notes:**

If automatic tracing of source code locations (aka PC tracing) is supported, then ITC will log the location where `VT_begin()` is called as source code location for this region and the location where `VT_end()` is called as SCL for the next part of the calling symbol (which may be a function or another, larger region).

If 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 )
```

**Java**

```
void begin( int statehandle )
```

**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 )
```

**Java**

```
void begin( int statehandle, int sclhandle )
```

**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 )
```

**Java**

```
void end( int statehandle )
```

**Parameters:**

**statehandle** obsolete, pass anything you want

**Returns:**

error code

**int VT\_endl (int *statehandle*, int *schandle*)**

Shortcut for [VT\\_thisloc\( \*schandle\* \); VT\\_end\( \*statehandle\* \)](#).

**Fortran**

VTENDL( *statehandle*, *schandle*, *ierr* )

**Java**

void end( int *statehandle*, int *schandle* )

**int VT\_enter (int *statehandle*, int *schandle*)**

Mark the beginning of a function.

Usage similar to [VT\\_beginl\(\)](#). See also [VT\\_begin\(\)](#).

**Fortran**

VTENTER( *statehandle*, *schandle*, *ierr* )

**Java**

void enter( int *statehandle*, int *schandle* )

**Parameters:**

***statehandle*** handle defined either with [VT\\_symdef\(\)](#) or [VT\\_funcdef\(\)](#)

***schandle*** handle, defined by [VT\\_scldf](#). Use [VT\\_NOSCL](#) if you don't have a specific value.

**Returns:**

error code

**int VT\_leave (int *schandle*)**

Mark the end of a function.

See also [VT\\_begin\(\)](#).

**Fortran**

VTLEAVE( *schandle*, *ierr* )

**Java**

void leave( int *schandle* )

**Parameters:**

***schandle*** handle, defined by [VT\\_scldf](#). 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\_wakeup (void)**

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

When ITC 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 ITC 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* )

**Java**

void wakeup()

**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()` resp. `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 )
```

### Java

```
int scopeDef( final String scopename, int classhandle, int scl1, int scl2 )
```

### 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, ITC assigns an intermediate identifier to 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 )
```

### Java

```
int scopeBegin( int scopehandle, int scl )
```

### 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 )
Java
  void scopeEnd( int scopehandle, int seqnr, int 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

ITC makes it possible to define an arbitrary, recursive group structure over the processes of an MPI application, and ITA 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 )
Java
  int getProcid( int procindex );
Parameters:
  procindex index of process (0 <= 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 )
```

**Java**

```
int getThreadID( int procindex, int thindex )
```

**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 )
```

**Java**

```
int groupDef( final String name, int ids[] )
```

**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

ITC 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 ITC 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 ITC 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
    ITC 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 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 e.g. 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**

```
VT_COUNTDEF( 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:**

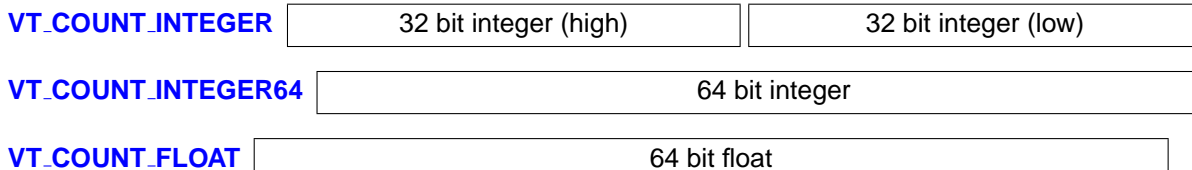
error code

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:





## 8.8. RECORDING COMMUNICATION EVENTS



At any time during execution, a process can record a new value for any of the defined counters by calling one of the ITC 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 behaviour 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 ITC.

**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 )

**Java**

void logSendmsg( int other\_rank, int count, int tag, int commid, int sclhandle )

**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](#))

***sclhandle*** handle as defined by [VT\\_scldf](#), 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 )

**Java**

void logRecvmsg( int other\_rank, int count, int tag, int commid, int sclhandle )

**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](#))

***sclhandle*** handle as defined by [VT\\_scldf](#), 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 )

**Java**

void logMsgEvent( int sender, int receiver, int count, int tag, int commid, double sendts, int sendscl, int recvscl )

**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](#))

**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 )

**Java**

void logOp( int opid, int commid, int root, int bsend, int brecv, double startts, int sclhandle )

**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.

ITC 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* )

**Java**

void logOpEvent( int *opid*, int *commid*, int *root*, int *numprocs*, int *bSEND*[], int *bRECV*[], double *startts*[], int *sclhandle* )

**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\\_scldf](#), 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 ITC.

**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, e.g. 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 ITC 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 ITC 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 ITC 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 )
```

**Java**

```
void beginUnordered()
```

**int VT\_end\_unordered (void)**

Close a period with out-of-order events that was started with [VT\\_begin\\_unordered\(\)](#).

**Fortran**

```
VTENDUNORDERED( ierr )
```

**Java**

```
void endUnordered()
```

## 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, else error code



**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 ITC or by the client by calling [VT\\_registerthread\(\)](#).

***clientname*** The name of the client. Currently only used for error messages. Copied by ITC.

**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 ITC 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  $\geq$  0, contacts  $\neq$  NULL)
2. start the server first, pass its contact string to the clients (numcontacts  $\geq$  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 ITC.

**numcontacts** number of client processes

**contacts** contact string for each client process (order is irrelevant); copied by ITC

**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 ITC 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 via `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. `nhandles()` 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, else 0. Defining the function may be impossible e.g. because ITC 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 e.g. because ITC 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 Constructor & Destructor Documentation

#### 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 e.g. uninitialized ITC.

**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](#) &scldf)
- void [end](#) ()
- void [end](#) (const char \*file, int line)
- void [end](#) ([VT\\_SclDef](#) &scldf)
- [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](#) &scldf)
- [~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\_SclDef & 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 e.g. uninitialized ITC.

**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\_SciDef & *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





## Chapter 9

# ITC Configuration

### 9.1 Configuring Intel(R) Trace Collector

With a configuration file, the user can customize various aspects of ITC's operation and define trace data filters.

### 9.2 Specifying Configuration Options

The environment variable `VT_CONFIG` can be set to the name of an ITC 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 directory than you'd 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 "VT\_" and hyphens replaced with underscores.

Finally, it is also possible to specify configuration options on the command line of a program. The only exception are Fortran programs (because ITC's access to command line parameters is limited there). To avoid conflicts between ITC's parameters and normal application parameters, only parameters following the special `-itc-args` are interpreted by ITC. To continue with the application's normal parameters, `-itc-args-end` may be used. There may be more than one block of ITC 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 have 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  $1/1024/1024^2/1024^3$ . 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 ITC 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 ITC 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](#)

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 ITA to analyse the data without loading all of it, so it is more scalable. Writing it is only supported by ITC 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 ITC, on in stftool

Several events can only be stored in STF, but not in VTF. ITC 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 ITA 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 ITA.

**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](#)

ITC 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 ITC.

#### 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.

#### PLUGIN

**Syntax:** <plugin name>

**Variable:** VT\_PLUGIN

If this option is used, then ITC 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.

#### DETAILED-STATES

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

**Variable:** VT\_DETAILED\_STATES

**Default:** on

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 ITC 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.



## 9.5. SUPPORTED DIRECTIVES

---

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 ITC 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 ITC documentation describes in more detail how an MPI implementation might do that.

### JAVA

**Syntax:** [on|off]

**Variable:** VT\_JAVA

**Default:** on in libVTjava, off in libVTsocket

This option controls Java function tracing. It serves as a master switch that - when turned off - avoids the overhead of function tracing completely.

### PCTRACE

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

**Variable:** VT\_PCTRACE

**Default:** off for performance analysis, enabled otherwise

Some platforms support the automatic stack sampling for MPI calls and user-defined events. ITC 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.

### 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 ITC 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

ITC 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

ITC 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 ITC 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 ITC 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 ITC 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 ITC 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 ITC 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 directory with more than 512MB
- failing that, directory 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]

**Variable:** `VT_STATISTICS`

**Default:** off

Enables or disables statistics about messages and symbols. These statistics are gathered by ITC independently from logging them and written to the protocol file, so you can get statistics in a machine-readable ASCII format without generating or loading the complete trace file.

**DYNAMIC-STATS**

**Syntax:** <filename> [<triplet>]

**Variable:** `VT_DYNAMIC_STATS`

This option is only available if **STATISTICS** was enabled in the initial ITC configuration file.

Each time `VT_confsync()` is called, the current statistics can be written into a separate file. The number of times that `VT_confsync()` is called are counted by ITC (starting with 1) and if the filename contains one or more "d", then they are replaced by this counter value.

It is not necessary to make the filename unique like that, though: ITC will remove the file before writing into it, so you can read old statistics while the application is running without getting parts of the file overwritten with new statistics (on Unix an application like "more" may have the old file open, while ITC is writing into another file with the same name).

The optional triplet specifies which instances of `VT_confsync()` will create this statistics file. It is always counting from the current instance forward, so 1:1:1 refers to the next (and only the next) instance of `VT_confsync()`. If you omit this parameter, then the statistics file will be written each time `VT_confsync()` is called.

`VT_confsync()` will generate the statistics file at the beginning and at the end, so if you set your breakpoint into `VT_confbreak()`, the statistic file will be up-to-date if it was enabled for the current instance of `VT_confsync()`, and if it was disabled and is enabled by changing the configuration the file will have been updated when `VT_confsync()` completes.

**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 analysing 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

ITC 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:** 5 minutes

If ITC 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.

## 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 ITC 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").

## TIMER

**Syntax:** <timer name or LIST>

**Variable:** `VT_TIMER`

**Default:** `gettimeofday`

ITC 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 ITC, 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.

## SYNC-MAX-DURATION

**Syntax:** <duration>

**Variable:** `VT_SYNC_MAX_DURATION`

**Default:** 1 minute

ITC 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 ITC 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 via `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 via message exchanges is enabled (the default), then ITC 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.



## 9.5. SUPPORTED DIRECTIVES

---

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 ITC treats the clock of all processes anywhere in the cluster as synchronized or not. Whether ITC makes that assumption depends on the selected time source.

### SYNCED-HOST

**Syntax:** [on|off]

**Variable:** `VT_SYNCED_HOST`

**Default:** on

Use this setting to override whether ITC treats the clock of all processes on the same node as synchronized or not. Whether ITC 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 ITC 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 ITC 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 ITC 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
```

Beside 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:



## 9.7. THE PROTOCOL FILE

- 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 criteria 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 ITC 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 f.i. 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) 1992-2006 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.
```

# Index

- activity
  - command line switch definition, [18](#)
- convert
  - command line switch definition, [50](#)
- dump
  - command line switch definition, [50](#)
- extended-vtf
  - command line switch definition, [51](#)
- input
  - command line switch definition, [17](#)
- logfile-format
  - command line switch definition, [51](#)
- logfile-name
  - command line switch definition, [51](#)
- matched-vtf
  - command line switch definition, [52](#)
- move
  - command line switch definition, [50](#)
- output
  - command line switch definition, [17](#)
- print-errors
  - command line switch definition, [50](#)
- print-files
  - command line switch definition, [50](#)
- print-statistics
  - command line switch definition, [50](#)
- print-threads
  - command line switch definition, [50](#)
- remove
  - command line switch definition, [50](#)
- request
  - command line switch definition, [51](#)
- state
  - command line switch definition, [17](#)
- symbol
  - command line switch definition, [18](#)
- ticks
  - command line switch definition, [51](#)
- verbose
  - command line switch definition, [17](#), [52](#)
- ~VT\_Function
  - VT\_Function, [86](#)
- ~VT\_Region
  - VT\_Region, [88](#)
- \_VT\_CommIDs
  - VT.h, [75](#)
- \_VT\_ErrorCode
  - VT.h, [56](#)
- \_VT\_OpTypes
  - VT.h, [79](#)
- AUTOFLUSH
  - config directive definition, [96](#)
- begin
  - VT\_Region, [88](#)
- CLUSTER
  - config directive definition, [96](#)
- COUNTER
  - config directive definition, [95](#)
- CURRENT-DIR
  - config directive definition, [93](#)
- DEADLOCK-TIMEOUT
  - config directive definition, [99](#)
- DETAILED-STATES
  - config directive definition, [94](#)
- DYNAMIC-STATS
  - config directive definition, [98](#)
- end
  - VT\_Region, [88](#), [89](#)
- ENTER-USERCODE
  - config directive definition, [94](#)
- ENVIRONMENT
  - config directive definition, [98](#)
- environment variable
  - VT\_ADD\_LIBS, [6](#)
  - VT\_CONFIG, [9](#)
  - VT\_CONFIG\_RANK, [9](#)
  - VT\_LIB\_DIR, [6](#)
  - VT\_MPI, [6](#)
  - VT\_ROOT, [5](#)
  - VT\_SLIB\_DIR, [6](#)
- EXTENDED-VTF
  - config directive definition, [93](#)
- FLUSH-PID
  - config directive definition, [98](#)



- FLUSH-PREFIX
  - config directive definition, [97](#)
- GetHandle
  - VT\_FuncDef, [84](#)
  - VT\_SclDef, [85](#)
- GROUP
  - config directive definition, [99](#)
- HANDLE-SIGNALS
  - config directive definition, [100](#)
- INFO BINMODE
  - config directive definition, [104](#)
- INFO CLUSTERDEF
  - config directive definition, [104](#)
- INFO NUMPROCS
  - config directive definition, [104](#)
- INFO PROCESS
  - config directive definition, [104](#)
- INTERNAL-MPI
  - config directive definition, [95](#)
- JAVA
  - config directive definition, [95](#)
- LOGFILE-FORMAT
  - config directive definition, [93](#)
- LOGFILE-NAME
  - config directive definition, [92](#)
- LOGFILE-PREFIX
  - config directive definition, [93](#)
- LOGFILE-RANK
  - config directive definition, [94](#)
- MEM-BLOCKSIZE
  - config directive definition, [96](#)
- MEM-FLUSHBLOCKS
  - config directive definition, [97](#)
- MEM-INFO
  - config directive definition, [96](#)
- MEM-MAXBLOCKS
  - config directive definition, [96](#)
- MEM-MINBLOCKS
  - config directive definition, [96](#)
- MEM-OVERWRITE
  - config directive definition, [97](#)
- NMCMMD
  - config directive definition, [101](#)
- OS-COUNTER-DELAY
  - config directive definition, [99](#)
- PCTRACE
  - config directive definition, [95](#)
- PLUGIN
  - config directive definition, [94](#)
- PROCESS
  - config directive definition, [95](#)
- PROGNAME
  - config directive definition, [93](#)
- PROTOFILE-NAME
  - config directive definition, [93](#)
- STATISTICS
  - config directive definition, [98](#)
- STF-CHUNKSIZE
  - config directive definition, [99](#)
- STF-PROCS-PER-FILE
  - config directive definition, [99](#)
- STF-USE-HW-STRUCTURE
  - config directive definition, [98](#)
- SYNC-MAX-DURATION
  - config directive definition, [100](#)
- SYNC-MAX-MESSAGES
  - config directive definition, [100](#)
- SYNC-PERIOD
  - config directive definition, [100](#)
- SYNCED-CLUSTER
  - config directive definition, [101](#)
- SYNCED-HOST
  - config directive definition, [101](#)
- TIMER
  - config directive definition, [100](#)
- UNIFY-COUNTERS
  - config directive definition, [102](#)
- UNIFY-GROUPS
  - config directive definition, [101](#)
- UNIFY-SCLS
  - config directive definition, [101](#)
- UNIFY-SYMBOLS
  - config directive definition, [101](#)
- VERBOSE
  - config directive definition, [94](#)
- VT.h
  - \_VT\_CommIDs, [75](#)
  - \_VT\_ErrorCode, [56](#)
  - \_VT\_OpTypes, [79](#)
  - VT\_abort, [80](#)
  - VT\_attach, [83](#)
  - VT\_begin, [68](#)
  - VT\_begin\_unordered, [80](#)
  - VT\_beginl, [68](#)
  - VT\_classdef, [66](#)
  - VT\_clientinit, [81](#)

- VT\_COMM\_INVALID, 75
- VT\_COMM\_SELF, 75
- VT\_COMM\_WORLD, 75
- VT\_COUNT\_ABSVAL, 73
- VT\_COUNT\_DATA, 73
- VT\_COUNT\_DISPLAY, 73
- VT\_COUNT\_FLOAT, 73
- VT\_COUNT\_INTEGER, 73
- VT\_COUNT\_INTEGER64, 73
- VT\_COUNT\_RATE, 73
- VT\_COUNT\_SCOPE, 74
- VT\_COUNT\_VALID\_AFTER, 74
- VT\_COUNT\_VALID\_BEFORE, 74
- VT\_COUNT\_VALID\_POINT, 74
- VT\_COUNT\_VALID\_SAMPLE, 74
- VT\_CountData, 73
- VT\_countdef, 74
- VT\_CountDisplay, 73
- VT\_CountScope, 74
- VT\_countval, 75
- VT\_end, 68
- VT\_end\_unordered, 80
- VT\_endl, 69
- VT\_enter, 69
- VT\_ERR\_BADARG, 56
- VT\_ERR\_BADFILE, 56
- VT\_ERR\_BADFORMAT, 56
- VT\_ERR\_BADINDEX, 56
- VT\_ERR\_BADKIND, 56
- VT\_ERR\_BADREQUEST, 56
- VT\_ERR\_BADSCL, 56
- VT\_ERR\_BADSCLID, 56
- VT\_ERR\_BADSYMBOLID, 56
- VT\_ERR\_COMM, 56
- VT\_ERR\_FLUSH, 56
- VT\_ERR\_IGNORE, 56
- VT\_ERR\_INVNT, 56
- VT\_ERR\_NOLICENSE, 56
- VT\_ERR\_NOMEMORY, 56
- VT\_ERR\_NOTHREADS, 56
- VT\_ERR\_NOTIMPLEMENTED, 56
- VT\_ERR\_NOTINITIALIZED, 56
- VT\_finalize, 58
- VT\_flush, 62
- VT\_funcdef, 66
- VT\_get\_parent, 83
- VT\_getprocid, 71
- VT\_getrank, 59
- VT\_getthrank, 60
- VT\_getthreadid, 72
- VT\_Group, 71
- VT\_GROUP\_CLUSTER, 71
- VT\_GROUP\_PROCESS, 71
- VT\_GROUP\_THREAD, 71
- VT\_groupdef, 72
- VT\_initialize, 58
- VT\_leave, 69
- VT\_log\_msgevent, 77
- VT\_log\_op, 77
- VT\_log\_opevent, 78
- VT\_log\_recvmsg, 76
- VT\_log\_sendmsg, 76
- VT\_ME, 71
- VT\_NOCLASS, 66
- VT\_NOSCL, 63
- VT\_OK, 56
- VT\_OP\_ALLGATHER, 79
- VT\_OP\_ALLGATHERV, 79
- VT\_OP\_ALLREDUCE, 79
- VT\_OP\_ALLTOALL, 79
- VT\_OP\_ALLTOALLV, 79
- VT\_OP\_BARRIER, 79
- VT\_OP\_BCAST, 79
- VT\_OP\_COUNT, 79
- VT\_OP\_GATHER, 79
- VT\_OP\_GATHERV, 79
- VT\_OP\_INVALID, 79
- VT\_OP\_REDUCE, 79
- VT\_OP\_REDUCE\_SCATTER, 79
- VT\_OP\_SCAN, 79
- VT\_OP\_SCATTER, 79
- VT\_OP\_SCATTERV, 79
- VT\_registernamed, 59
- VT\_registerthread, 59
- VT\_SCL\_DEF\_CXX, 85
- VT\_scldef, 63
- VT\_sclstack, 64
- VT\_scopebegin, 70
- VT\_scopedef, 70
- VT\_scopeend, 71
- VT\_serverinit, 82
- VT\_symdef, 67
- VT\_symstate, 61
- VT\_thisloc, 65
- VT\_timestamp, 62
- VT\_timestart, 62
- VT\_timesync, 28
- VT\_traceoff, 60
- VT\_traceon, 60
- VT\_tracestate, 61
- VT\_VERSION, 55
- VT\_VERSION\_COMPATIBILITY, 56
- VT\_wakeup, 69
- VT\_AUTOFLUSH
  - env variable definition, 96
- VT\_CLUSTER
  - env variable definition, 96
- VT\_COUNTER

- env variable definition, 95
- VT\_CURRENT\_DIR
  - env variable definition, 93
- VT\_DEADLOCK\_TIMEOUT
  - env variable definition, 99
- VT\_DETAILED\_STATES
  - env variable definition, 94
- VT\_DYNAMIC\_STATS
  - env variable definition, 98
- VT\_ENTER\_USERCODE
  - env variable definition, 94
- VT\_ENVIRONMENT
  - env variable definition, 98
- VT\_EXTENDED\_VTF
  - env variable definition, 93
- VT\_FLUSH\_PID
  - env variable definition, 98
- VT\_FLUSH\_PREFIX
  - env variable definition, 97
- VT\_GROUP
  - env variable definition, 99
- VT\_HANDLE\_SIGNALS
  - env variable definition, 100
- VT\_INTERNAL\_MPI
  - env variable definition, 95
- VT\_JAVA
  - env variable definition, 95
- VT\_LOGFILE\_FORMAT
  - env variable definition, 93
- VT\_LOGFILE\_NAME
  - env variable definition, 92
- VT\_LOGFILE\_PREFIX
  - env variable definition, 93
- VT\_LOGFILE\_RANK
  - env variable definition, 94
- VT\_MEM\_BLOCKSIZE
  - env variable definition, 96
- VT\_MEM\_FLUSHBLOCKS
  - env variable definition, 97
- VT\_MEM\_INFO
  - env variable definition, 96
- VT\_MEM\_MAXBLOCKS
  - env variable definition, 96
- VT\_MEM\_MINBLOCKS
  - env variable definition, 96
- VT\_MEM\_OVERWRITE
  - env variable definition, 97
- VT\_NMCMD
  - env variable definition, 101
- VT\_OS\_COUNTER\_DELAY
  - env variable definition, 99
- VT\_PCTRACE
  - env variable definition, 95
- VT\_PLUGIN
  - env variable definition, 94
- VT\_PROCESS
  - env variable definition, 95
- VT\_PROGNAME
  - env variable definition, 93
- VT\_PROTOFILE\_NAME
  - env variable definition, 93
- VT\_STATISTICS
  - env variable definition, 98
- VT\_STF\_CHUNKSIZE
  - env variable definition, 99
- VT\_STF\_PROCS\_PER\_FILE
  - env variable definition, 99
- VT\_STF\_USE\_HW\_STRUCTURE
  - env variable definition, 98
- VT\_SYNC\_MAX\_DURATION
  - env variable definition, 100
- VT\_SYNC\_MAX\_MESSAGES
  - env variable definition, 100
- VT\_SYNC\_PERIOD
  - env variable definition, 100
- VT\_SYNCED\_CLUSTER
  - env variable definition, 101
- VT\_SYNCED\_HOST
  - env variable definition, 101
- VT\_TIMER
  - env variable definition, 100
- VT\_UNIFY\_COUNTERS
  - env variable definition, 102
- VT\_UNIFY\_GROUPS
  - env variable definition, 101
- VT\_UNIFY\_SCLS
  - env variable definition, 101
- VT\_UNIFY\_SYMBOLS
  - env variable definition, 101
- VT\_VERBOSE
  - env variable definition, 94
- VT\_ADD\_LIBS
  - environment variable, 6
- VT\_CONFIG
  - environment variable, 9
- VT\_CONFIG\_RANK
  - environment variable, 9
- VT\_LIB\_DIR
  - environment variable, 6
- VT\_MPI
  - environment variable, 6
- VT\_ROOT
  - environment variable, 5
- VT\_SLIB\_DIR
  - environment variable, 6
- VT\_abort
  - VT.h, 80
- VT\_attach

- VT.h, [83](#)
- VT\_begin
  - VT.h, [68](#)
- VT\_begin\_unordered
  - VT.h, [80](#)
- VT\_beginl
  - VT.h, [68](#)
- VT\_classdef
  - VT.h, [66](#)
- VT\_clientinit
  - VT.h, [81](#)
- VT\_COMM\_INVALID
  - VT.h, [75](#)
- VT\_COMM\_SELF
  - VT.h, [75](#)
- VT\_COMM\_WORLD
  - VT.h, [75](#)
- VT\_COUNT\_ABSVAL
  - VT.h, [73](#)
- VT\_COUNT\_DATA
  - VT.h, [73](#)
- VT\_COUNT\_DISPLAY
  - VT.h, [73](#)
- VT\_COUNT\_FLOAT
  - VT.h, [73](#)
- VT\_COUNT\_INTEGER
  - VT.h, [73](#)
- VT\_COUNT\_INTEGER64
  - VT.h, [73](#)
- VT\_COUNT\_RATE
  - VT.h, [73](#)
- VT\_COUNT\_SCOPE
  - VT.h, [74](#)
- VT\_COUNT\_VALID\_AFTER
  - VT.h, [74](#)
- VT\_COUNT\_VALID\_BEFORE
  - VT.h, [74](#)
- VT\_COUNT\_VALID\_POINT
  - VT.h, [74](#)
- VT\_COUNT\_VALID\_SAMPLE
  - VT.h, [74](#)
- VT\_CountData
  - VT.h, [73](#)
- VT\_countdef
  - VT.h, [74](#)
- VT\_CountDisplay
  - VT.h, [73](#)
- VT\_CountScope
  - VT.h, [74](#)
- VT\_countval
  - VT.h, [75](#)
- VT\_end
  - VT.h, [68](#)
- VT\_end\_unordered
  - VT.h, [80](#)
- VT\_endl
  - VT.h, [69](#)
- VT\_enter
  - VT.h, [69](#)
- VT\_ERR\_BADARG
  - VT.h, [56](#)
- VT\_ERR\_BADFILE
  - VT.h, [56](#)
- VT\_ERR\_BADFORMAT
  - VT.h, [56](#)
- VT\_ERR\_BADINDEX
  - VT.h, [56](#)
- VT\_ERR\_BADKIND
  - VT.h, [56](#)
- VT\_ERR\_BADREQUEST
  - VT.h, [56](#)
- VT\_ERR\_BADSCL
  - VT.h, [56](#)
- VT\_ERR\_BADSCLID
  - VT.h, [56](#)
- VT\_ERR\_BADSYMBOLID
  - VT.h, [56](#)
- VT\_ERR\_COMM
  - VT.h, [56](#)
- VT\_ERR\_FLUSH
  - VT.h, [56](#)
- VT\_ERR\_IGNORE
  - VT.h, [56](#)
- VT\_ERR\_INVT
  - VT.h, [56](#)
- VT\_ERR\_NOLICENSE
  - VT.h, [56](#)
- VT\_ERR\_NOMEMORY
  - VT.h, [56](#)
- VT\_ERR\_NOTHEADS
  - VT.h, [56](#)
- VT\_ERR\_NOTIMPLEMENTED
  - VT.h, [56](#)
- VT\_ERR\_NOTINITIALIZED
  - VT.h, [56](#)
- VT\_finalize
  - VT.h, [58](#)
- VT\_flush
  - VT.h, [62](#)
- VT\_FuncDef
  - VT\_FuncDef, [84](#)
- VT\_FuncDef, [84](#)
  - GetHandle, [84](#)
  - VT\_FuncDef, [84](#)
- VT\_funcdef
  - VT.h, [66](#)
- VT\_Function, [85](#)
  - ~VT\_Function, [86](#)

VT\_Function, [86](#)  
VT\_get\_parent  
  [VT.h, 83](#)  
VT\_getprocid  
  [VT.h, 71](#)  
VT\_getrank  
  [VT.h, 59](#)  
VT\_getthrank  
  [VT.h, 60](#)  
VT\_getthreadid  
  [VT.h, 72](#)  
VT\_Group  
  [VT.h, 71](#)  
VT\_GROUP\_CLUSTER  
  [VT.h, 71](#)  
VT\_GROUP\_PROCESS  
  [VT.h, 71](#)  
VT\_GROUP\_THREAD  
  [VT.h, 71](#)  
VT\_groupdef  
  [VT.h, 72](#)  
VT\_initialize  
  [VT.h, 58](#)  
VT\_leave  
  [VT.h, 69](#)  
VT\_log\_msgevent  
  [VT.h, 77](#)  
VT\_log\_op  
  [VT.h, 77](#)  
VT\_log\_opevent  
  [VT.h, 78](#)  
VT\_log\_recvmsg  
  [VT.h, 76](#)  
VT\_log\_sendmsg  
  [VT.h, 76](#)  
VT\_ME  
  [VT.h, 71](#)  
VT\_NOCLASS  
  [VT.h, 66](#)  
VT\_NOSCL  
  [VT.h, 63](#)  
VT\_OK  
  [VT.h, 56](#)  
VT\_OP\_ALLGATHER  
  [VT.h, 79](#)  
VT\_OP\_ALLGATHERV  
  [VT.h, 79](#)  
VT\_OP\_ALLREDUCE  
  [VT.h, 79](#)  
VT\_OP\_ALLTOALL  
  [VT.h, 79](#)  
VT\_OP\_ALLTOALLV  
  [VT.h, 79](#)  
VT\_OP\_BARRIER  
  [VT.h, 79](#)  
VT\_OP\_BCAST  
  [VT.h, 79](#)  
VT\_OP\_COUNT  
  [VT.h, 79](#)  
VT\_OP\_GATHER  
  [VT.h, 79](#)  
VT\_OP\_GATHERV  
  [VT.h, 79](#)  
VT\_OP\_INVALID  
  [VT.h, 79](#)  
VT\_OP\_REDUCE  
  [VT.h, 79](#)  
VT\_OP\_REDUCE\_SCATTER  
  [VT.h, 79](#)  
VT\_OP\_SCAN  
  [VT.h, 79](#)  
VT\_OP\_SCATTER  
  [VT.h, 79](#)  
VT\_OP\_SCATTERV  
  [VT.h, 79](#)  
VT\_Region, [87](#)  
  ~VT\_Region, [88](#)  
  begin, [88](#)  
  end, [88, 89](#)  
  VT\_Region, [87](#)  
VT\_registernamed  
  [VT.h, 59](#)  
VT\_registerthread  
  [VT.h, 59](#)  
VT\_SCL\_DEF\_CXX  
  [VT.h, 85](#)  
VT\_SclDef  
  VT\_SclDef, [85](#)  
VT\_SclDef, [85](#)  
  GetHandle, [85](#)  
  VT\_SclDef, [85](#)  
VT\_scldef  
  [VT.h, 63](#)  
VT\_sclstack  
  [VT.h, 64](#)  
VT\_scopebegin  
  [VT.h, 70](#)  
VT\_scopedef  
  [VT.h, 70](#)  
VT\_scopeend  
  [VT.h, 71](#)  
VT\_serverinit  
  [VT.h, 82](#)  
VT\_symdef  
  [VT.h, 67](#)  
VT\_symstate  
  [VT.h, 61](#)  
VT\_thisloc



VT.h, [65](#)  
VT\_timestamp  
    VT.h, [62](#)  
VT\_timestart  
    VT.h, [62](#)  
VT\_timesync  
    VT.h, [28](#)  
VT\_traceoff  
    VT.h, [60](#)  
VT\_traceon  
    VT.h, [60](#)  
VT\_tracestate  
    VT.h, [61](#)  
VT\_VERSION  
    VT.h, [55](#)  
VT\_VERSION\_COMPATIBILITY  
    VT.h, [56](#)  
VT\_wakeup  
    VT.h, [69](#)