Intel® Trace Collector 6.0.2.1

User's Guide 6.0.2.1

Intel GmbH
Hermülheimer Straße 8a
D–50321 Brühl, Germany

Contents

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 7) 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

This version of the ITC was compiled for:
EM64T-LIN
Intel MPI 2.0

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.

It is compatible with all other MPI implementations that use the same binary interface. If in doubt, please 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:

```
Datatypes:

   sizeof(MPI_Datatype): 4
   sizeof(MPI_Comm)    : 4
   sizeof(MPI_Request) : 4

C constants:

   MPI_CHAR            : 1275068673
   MPI_BYTE            : 1275068685
   MPI_SHORT           : 1275068931
   MPI_INT             : 1275069445
   MPI_FLOAT           : 1275069450
   MPI_DOUBLE          : 1275070475
   MPI_COMM_WORLD      : 1140850688
   MPI_COMM_SELF       : 1140850689

MPI_Status structure and byte offsets of members:

   MPI_STATUS_SIZE     : 5
   MPI_SOURCE          : 8
   MPI_TAG             : 12
   MPI_ERROR           : 16
```

This output is also found in `examples/mpiconstants.out`.

The following features are supported:

Feature	Description
Thread-safety	supported, see 1.3
MPI tracing • IO • MPI One-Sided Communication • MPI-2 • fail-safe • correctness checking	3.1 not supported not supported not supported supported (3.1.7) not supported
Java tracing	4
Single-process tracing	3.2
Tracing of Distributed Applications	5
Subroutine tracing	3.3
Tracing of Binaries without Recompilation	3.3
Counter tracing	API in 7.7
Automatic Counter tracing of OS Activity	3.9
Automatically Recording Source Location Information	3.7 (requires compiler support)
Manually Recording Source Location Information	API in 7.3
Recording Statistical Information	3.6
Tracing Libraries at Different Levels of Detail	3.12
Nonblocking Flushing	MEM-FLUSHBLOCKS

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 (3.10)
libVT	MPI tracing (3.1)
libVTfs	fail-safe MPI tracing (3.1.7)
libVTcs	Tracing of Distributed Applications and Single-processes (5, 3.2)
libVTjava	Java tracing (4) (Distributed Applications and Single Processes)
VT_sample	Automatic Counter tracing with PAPI and getrusage() (3.8)
stftool	Manipulation of trace files (6.4.1)
xstftool/expandvtlog.pl	Conversion of trace files into readable format (6.4.2)
itcinstrument	Tracing of Binaries without Recompilation (3.3)

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 3.5.

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, e.g. the ITC API calls (man VT_enter) and the ITC configuration (man VT_CONFIG). To access the man pages you must 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, i.e. 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, please ensure that you consult the correct documentation.

Chapter 2

Installation

After unpacking the ITC archive in a directory of your choice you need to enter this directory and execute the script `./install` located there. At this time you must have a license key from Intel. On Intel architectures ITC uses Macrovision's FLEXlm electronic licensing technology (FLEXlm is a registered trademark of Globetrotter Software Inc). This release of the ITC uses version 9.2 of FLEXlm.

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.

The location of this file must be made known to the install command by setting the environment variable INTEL_LICENSE_FILE to the full pathname of the file before the installation is invoked.

For example, in the C-shell, type:

```
    setenv INTEL_LICENSE_FILE /opt/intel/itc/license.dat
```

or in the Bourne shell, type:

```
    INTEL_LICENSE_FILE=/opt/intel/itc/license.dat
    export INTEL_LICENSE_FILE
```

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 should be specified via
the $INTEL_LICENSE_FILE environment variable.

License file(s) used were (in this order):

Please visit http://support.intel.com/support/performancetools/support.htm
```

```
if you require technical assistance.

FLEX_CHECKOUT test failed to acquire license (rc=-1)
```

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

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

After asking about the desired install directory, read and write permissions the install script creates that directory, copies all files and sets the permissions of the ITC files and directories accordingly. It also creates `sourceme.sh` (for shells with Bourne syntax) and `sourceme.csh` (for shells with csh syntax). Sourcing the correct file in a shell (with `. sourceme.sh` resp. `source sourceme.csh`) will set all of the required environment variables.

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 one can either install ITC once in a shared directory which is mounted at the same location on all nodes, or one 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 one 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 7) 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 you should have 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 must 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 must be passed to the compiler. On most systems, this is done with the -I flag, e.g. -I$(VT_ROOT)/include.

3.1.2 Linking MPI Programs with ITC

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

```
mpiicc ctest.o $(LFLAGS) -lVT -ldwarf -lelf -lnsl -lm -
  lpthread -o ctest
```

```
    mpiifort ftest.o $(LFLAGS) -lVT -ldwarf -lelf -lnsl -lm -
      lpthread -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.

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, gcc $<$ 3.0	-lmpigc
Intel®MPI, gcc $>=$ 3.0 or icpc	-lmpiic
mpich 1.2.x	-lpmpich++

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:

```
    mpiifort ftest.o -lVTg77 $(LFLAGS) -lVT -ldwarf -lelf -lnsl
      -lm -lpthread -o ftest
```

In all cases must the binary interface of the MPI libraries 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:

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 don't 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, please 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 in the appendix A 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 should 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 an application must be linked against libVTfs instead of libVT, like this:

```
mpiicc ctest.o $(LFLAGS) -lVTfs -ldwarf -lelf -lnsl -lm -
  lpthread -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. Each hostname must be mapped to an IP address that all processes can connect to. Note that this is not the case if /etc/hosts lists the local hostname as alias for 127.0.0.1 and processes are started on different hosts.

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”.

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 should use blocking MPI calls instead, which is also better for performance.

Premature Abort If all processes remain in MPI for a long time e.g. due to a long data transfer, 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.

3.2 Single-process Tracing

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

Linking is accomplished by adding libVTcs.a and the libraries it needs to the link line:

```
-lVTcs -ldwarf -lelf -lnsl -lm -lpthread
```

The application must 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 5.

Subroutine tracing (3.3) or binary instrumentation (3.3) can be used with and without further ITC API (see chapter 7) 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.

3.3 Tracing Application Subroutines

Function tracing is always possible when using the GNU Compiler suite version 2.95.2 or later. For that the object files that contain functions that are to be traced must be compiled with "-finstrument-function" and VT must 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 3.12.

3.4 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 a MPI application. If the application is not a 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, one 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 must already contain calls to VT_initialize() and VT_finalize() 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

To run itcinstrument you should source the ITC sourceme 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 sourceme 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

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.

--use-debug

Syntax:

Default: on

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

--list

Syntax:

Default: off

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

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

--filter

Syntax: <filter command>

By specifying a command here one can remap the function names as found in the binary into something more useful in the trace file. For example, all functions with a common prefix like FOO_bar could be turned into functions inside a common class (FOO:bar).

The command is executed in a shell. Its standard input consists of one function name per line and for each input line, the command must print either the unmodified original function name or its replacement. To chain several commands together, one can use a shell pipe.

The input function names have nearly the same format as in the --list output above. They contain slightly more information so that the filter command can decide itself how to reduce that information:

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

Example input:

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

Example filter commands:

- sed -e 's/^Application:FOO_\(.*\)/FOO:\1/' (use FOO_ prefix as class)
- sed -e 's!^/home/joe/!!' | sed -e 's!/!:!g' (strip common path, use remaining path as class hierarchy by replacing slash with colon)
- tee /tmp/filterinput.txt (generate a copy of the input which then can be used to test the effect of filter commands more quickly)
- perl -p -e 's/^[^:]*:(([^_]*_){2,})$/$1/ && s/_/:/g;' (for all names which look like Fortran 90 module functions, strip the source file or class, then convert the underscore in the function name into a class separator; this works e.g. for src/foo.f90:foo_bar_)

In the output of the filter command leading and trailing separators are ignored and multiple separators are treated just like a single one, therefore it does not matter that the last example also converts the trailing underscore into a separator.

Beware of meta characters in the filter commands: itcinstrument itself treats any command line argument starting with -- as one of its own command line switches. To avoid that, include the parameter of --filter in single or double quotes. The shell might also expand certain characters, both when calling itcinstrument and when executing the filter. To check which command really gets executed, use --verbose 2. In general the easiest solution is to put the filter commands into a shell script and give the name of the shell script to --filter.

--insert

Syntax: <libname>

Default: libVT for MPI applications

ITC has several libraries that can be used to do different kinds of tracing. For MPI applications the most useful one is libVT, so it is the default. For other applications itcinstrument cannot guess what the user wants to do, so the library which is to be inserted needs to be specified explicitly. "itcinstrument --input <executable>" will list the available choices in this installation of ITC.

--mpi

Syntax: <path to MPI>

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

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

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

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

--profile

Syntax:

Default: off

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

--config

Syntax: <filename>

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

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

--longjmp

Syntax:

Default: off

Profiling an application which uses e.g. setjmp()/longjmp() to transfer control back to a higher level in the call stack requires extra checks after function calls. Otherwise ITC will not notice that setjmp() has returned until the next instrumented function is call.

Another situation where the extra check is needed is when function foo() just consists of a jump instruction via an address pointer into bar() and foo() is instrumented while bar() isn't: then returning from foo() will not be logged immediately.

In applications which do not use long jumps this extra check is redundant and just causes additional overhead, therefore it is disabled by default. If itcinstrument finds calls to setjmp() or sigsetjmp() it enables this check automatically and prints an information message.

If this automatism fails or other functions are used to execute a long jump, then you need to enable the extra check manually with this configuration option.

--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>:∗”.

3.5 Multithreaded Tracing

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

3.6 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 one 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 6.4.1) can convert from the machine-readable format to ASCII text with the same format as in the protocol file with --print-statistics.

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.

3.7 Recording Source Location Information

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

```
mpiicc -g -c ctest.c
mpiifort -g -c ftest.c
```

If your compiler does not support a flag, then search for a similar one. On Linux the compiler must generate 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 doesn't need any special options.

Another requirement is that the compiler must 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.

At runtime Program Counter (PC) tracing must be enabled, either by setting the environment variable VT_PCTRACE to e.g. 5 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 e.g. 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 should 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 7.3 for more information).

3.8 Recording Hardware Performance Information

ITC can sample Operating System values for each process with the getrusage() system call and hardware counters with the Performace 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.

This layer is implemented in the `VT_sample.c` source file. It was not possible to provide a pre-compiled object file, because PAPI was either not available or not installed when this package was prepared. The `VT_sample.o` file can be rebuilt by entering the 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.

The `VT_sample.o` object file must be added 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 -lpapi must also be added, 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:

```
mpiicc ctest.o <search path for PAPI> $(LFLAGS) -lVTsample -
  lVT -lpapi -ldwarf -lelf -lnsl -lm -lpthread <libs required
  by PAPI> -o ctest
```

Then the application must be run 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 e.g. "COUNTER PAPI_* ON" enables all PAPI counters at once. Section 8 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, then ITC must be given 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. The man page of getrusage() or the system manual should be consulted to learn more about these counters:

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

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

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

3.9 Recording OS Counters

Similar to the process specific counters in the previous section, 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.

3.10 Using the Dummy Libraries

Programs containing calls to the ITC API (see section 7) 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:

```
    mpiicc ctest.o $(LFLAGS) -lVTnull  -o ctest
```

3.11 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” 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 started. This can be done either by automatically sourcing the ITC sourceme 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 one can insert the ITC into a MPI binary that was not linked against it. For that to work MPI itself must have been linked dynamically. For mpich, one needs 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, LD_LIBRARY_PATH must be set as described above and in addition to that, the environment variable LD_PRELOAD must be set to "libVT.so".

3.12 Tracing Library Calls

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

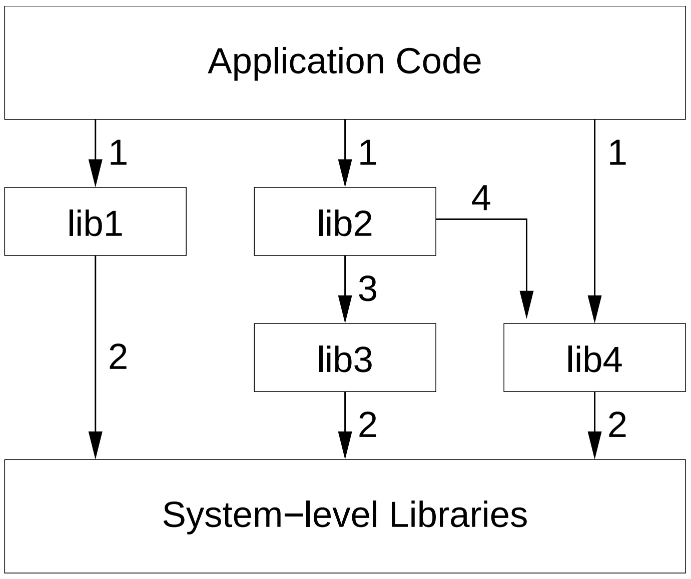

Figure 3.1: General structure of an application using many different libraries.

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 (e.g. 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 ITC 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 ITC. *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 must match a pattern before either folding or unfolding is executed. Section 8.6 has a detailed description of the syntax.

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

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

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

application developer: trace the application with only the top-level calls in lib1, lib2, and lib3

```
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 one could match against a function name prefix that is shared by all library calls in the same library.

Chapter 4

Java Tracing

4.1 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 7 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

4.2 Usage

ITC must be loaded 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 must include the $(VT_ROOT)/slib directory or the JVM will complain about not being able to find libVTcsjava.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 8. Because there is no unique application name in Java, the default name for the generated trace is “<name of JVM binary>”.stf (i.e. 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
```

“app class” must be replaced 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, one 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 e.g. 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 function tracing should be used to learn which code is actually executed by a thread.

4.3 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 the CLASSPATH and LD_LIBRARY_PATH must both include $(VT_ROOT)/lib. ITC must be initialized 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, this function and the matching VT_finalize() must be imported 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 5.

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 occured, 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 7.

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 5

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.

5.1 Design

The following conditions must 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 shouldn't be done too frequently.

- For a reliable startup, the application must gather a short string from every process in one place to boostrap the TCP/IP communication in ITC. Alternatively one process is started first and its string must be passed to the others. In this case one 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.

- The hostname must be mapped 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.

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 acomplished with another API call by one of the processes.

5.2 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.

5.2.1 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.

Some of the VT API calls may block, especially VT_initialize(). They should be executed in a separate thread if the process wants to continue. These pending calls can be aborted with VT_abort(), e.g. if another process failed to initialize trace data collection. This failure should 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.

5.3 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” must 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().

5.4 Spawning Processes

Spawning new processes is expensive, because it involves setting up TCP communication, clock synchronization, configuration broadcasting etc. 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.

5.5 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.

5.6 Usage

Executables in the application must be linked with -lVTcs and the same additional parameters as listed in section 3.2. 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.

5.7 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.

5.8 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.

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 6

Structured Tracefile Format

6.1 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. Mechanisms must be provided to start at a coarser level of display and then resolve the display into more detailed information.

A coarse view of the data will be represented by *frames*, which cover different parts of the trace data and provide previews for these parts, the so called *thumbnails*. Usually several frames exist in one trace and the user will be able to navigate through the frames and select one or more to request additional detailed information. The subdivision of a trace into frames can occur along three principal dimensions:

along the time axis different frames represent different time intervals.

along the task/thread axis different frames represent different threads or processes

along the kind of trace data a frame can contain any combination of the following categories of data: state changes, collective operations, point-to-point messages, counter values, and finally file I/O data (for MPI-I/O, if supported).

For any application, the subdivision of trace data into frames can be defined at runtime by compiling calls to the frame definition routines in the ITC API (see section 7.8) into the executable, or before starting the application by specifying the configuration options discussed in section 8. It is important to point out that frames are independent of the physical storing of data in files, which is controlled by another set of configuration options.

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.

6.2 STF Components

A structured tracefile actually consists of a number of files as shown in the figure 6.1. Depending on the number of frames and their distribution to 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 frame file with the name <trace>.stf.frm
- 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.

Please 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 6.4.1).

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.

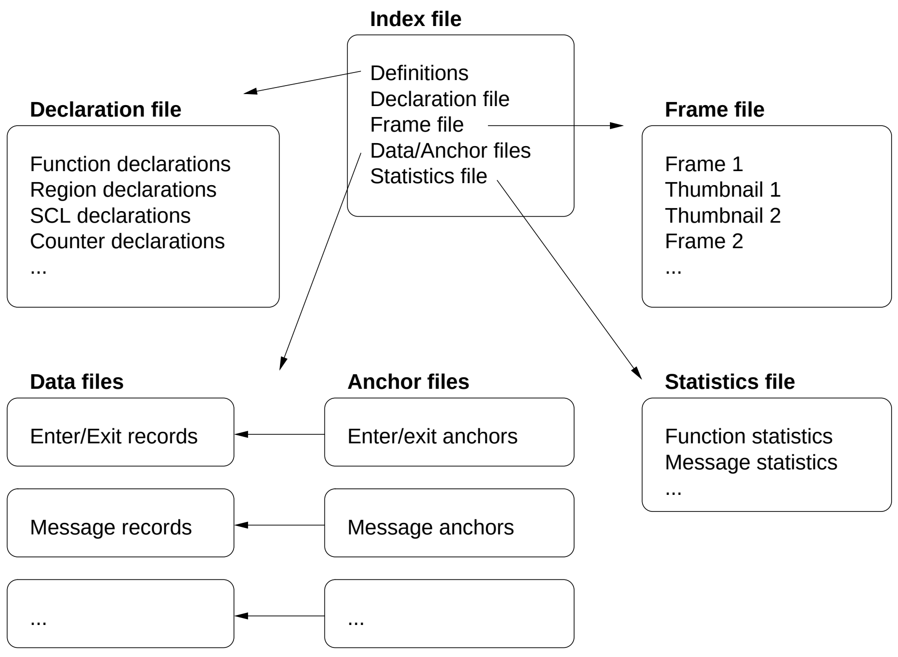

Figure 6.1: STF components

6.3 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 frame structure is preserved, as are the precomputed thumbnails. 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 penality on parallel read accesses to the same file.

6.4 Configuring STF

The two main aspects of the STF behavior that can be configured using directives in the ITC configuration file or the equivalent environment variables as described in section 8 are:

Frame definition: frames can be defined by a regular subdivision of the process and execution time space, and also depend on the hardware structure of the machine (where all of the processes are running on the same node in one frame).

Mapping to files: frames are just a logical concept, and need not coincide with the set of files actually written. ITC allows the event data to be partitioned in the process files by blocking, or coinciding with the hardware structure, such that events from processes running on the same node end up in one file.

The most important mechanisms for defining frames supported in ITC are:

FRAME-USE-HW-STRUCTURE combines all processes running on the same node into the same frame

PROCS-PER-FRAME <number> limits the number of processes that can be put in a frame

SECONDS-PER-FRAME <timespec> divides frames by time so that no frame corresponds to more than <timespec> of execution

FRAMES-PER-RUNTIME <num> it adapts the duration so that the given number of frames is achieved.

DATA-PER-FRAME <sizespec> divides frames in time whenever the data collected by all processes exceeds the given freshold

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.

6.4.1 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)
- manipulate frames in the file (--redo-frames)
- list frames, thumbnails, statistics (--print-frames, --print-thumbnails, --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 overriden 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 --frames 1
         --logfile-name example_frame1.avt # extract frame #1 as ASCII
stftool example.stf --request 1s:5s
         --logfile-name example_1s5s.bvt   # extract interval as binary
```

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 similiar 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-frames

Syntax:

Default: off

Prints a list of all frames in the input file to stdout.

--print-thumbnails

Syntax:

Default: off

Prints the precomputed thumbnails of each frame in the input file to stdout. Implies PRINT-FRAMES.

--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.

--redo-frames

Syntax:

Default: off

Modifies the frames of the STF file without copying it. By default it will keep all frames in the file, but recalculate their thumbnails. You can control which frames are kept with the FRAMES filter options and add new ones with FRAME.

--dump

Syntax:

Default: off

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

--frames

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

Default: 0:N = all

With this option you can extract some of the predefined frames from the input file. By default all frames are enabled, but if you use this option then only those listed explicitly are extracted. The first form enables frames by their number, while the second one matches against either the type or label of a frame. The second form overrides the first, and a filter that matches the label of a frame overrides a filter that matches the type.

If the stftool is used to recalculate frames, then this option specifies which frames are preserved.

--request

Syntax: ”<type>”, <thread triplets>, <categories>, <duration>, <window>

This option has the same arguments as the --frame option below, but in contrast to defining a new frame, it restricts the data that is written into the new trace to that which matches the arguments. This option can be used more than once and then data matching any request is written. In addition to those categories mentioned for a frame, ERRORS and REQUESTS are also supported categories.

--ticks

Syntax:

Default: off

Setting this option to ’on’ lets stftool interpret all timestamps as ticks (rather than seconds, miliseconds etc). Given time values are converted into seconds and then truncated (floor).

--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 must 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.

--frame

Syntax: "<type>", <thread triplets>, <categories>, <duration>, <window>

This option defines a new frame for certain categories and threads. The <duration> corresponds to SECONDS-PER-FRAME, but the value is valid for this frame type alone. If a window is given (in the form <timespec>:<timespec> with at least one unit descriptor), frames are created only inside 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, i.e. use "10s" instead of just "10". The <type> can be any kind of string in single or double quotation marks, but it should uniquely identify the kind of data combined into this frame. Valid <categories> are FUNCTIONS, SCOPES, OPENMP, FILEIO, COUNTERS, MESSAGES, COLLOPS.

All of the arguments are optional and default to "unnamed frame", 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.

--thumbnail

Syntax: <pattern> [on|off]

Default: on

Enables or disables those thumbnails whose name matches the pattern.

--message-thumb-size

Syntax: <size>

Default: 32

This option limits the size of the "Sent Message Statistics" thumbnail in the x and y directions. Without this limit the thumbnail would require space proportional to the number of processes squared, which does not scale for large number of processes.

SEE ALSO

VT_CONFIG(3)

6.4.2 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
- --frames: extract trace data of certain frames
- --matched-vtf: put information about complex events like messages and collective operations into one line

Description

The xstftool is a simple wrapper around the stftool and the expandvtlog.pl Perl script which tells the 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 7

User-level Instrumentation with the API

7.1 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, and finally define and create frames. 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. Therefore you should check against VT_VERSION_COMPATIBILITY to determine whether your program is compatible with the ITC library or to compile differently.

#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 should be checked.

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 symbold id.
VT_ERR_BADSCLID	wrong SCL id.
VT_ERR_BADSCL	wrong SCL.
VT_ERR_BADFORMAT	wrong format.
VT_ERR_BADKIND	Wrong kind found.
VT_ERR_NOMEMORY	Could not get memory.
VT_ERR_BADFILE	Error while handling file.
VT_ERR_FLUSH	Error while flushing.
VT_ERR_BADARG	wrong argument.
VT_ERR_NOTHREADS	no worker threads.
VT_ERR_BADINDEX	wrong thread index.
VT_ERR_COMM	communication error.
VT_ERR_INVT	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
```

Of course, breaking compatibility that way will be avoided at all costs. Beware that you must compare 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, one 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.

7.2 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 $>=$ 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.

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

VTTRACEOFF( )

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

VTTRACEON( )

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

VTTRACESTATE(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

Please refer to section 8 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 monotonously increasing time stamps that measure seconds, or VT_ERR_NOTINITIALIZED.
Time stamps are not guaranteed to be synchronized between processes. Within each process they are always larger than the value returned by VT_timestart().
Fortran
DOUBLE PRECISION VTSTAMP( )
Java
double timeStamp( )

double VT_timestart (void)
Returns point in time when process started, or VT_ERR_NOTINITIALIZED.
Fortran
DOUBLE PRECISION VTTIMESTART( )
Java
double timeStart( )

7.3 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, e.g. calls to the send/receive or collective communication and I/O routines.

To minimize instrumentation overhead, locations for the state changes and communication events are referred to by integer location handles that can be defined by calling the new API routine VT_scldef(), which will automatically assign a handle. The old API routine VT_locdef() which required the user to assign a handle value has been removed. A source location is a pair of a filename and a line number within that file.

int VT_scldef (const char ∗ *file*, int *line_nr*, int ∗ *sclhandle*)
Allocates a handle for a source code location (SCL).
Fortran
VTSCLDEF(file, line_nr, sclhandle, 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:
sclhandle 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.

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

7.4 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, e.g. 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 must be 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.

7.4.1 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”.

7.4.2 Old Interface

To define a new symbol, first determine which value should 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. Care must be taken to avoid using 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

7.4.3 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 must 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 were 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 *sclhandle*)

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

Fortran

VTENDL(statehandle, sclhandle, ierr)

Java

void end(int statehandle, int sclhandle)

int VT_enter (int *statehandle*, int *sclhandle*)

Mark the beginning of a function.

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

Fortran

VTENTER(statehandle, sclhandle, ierr)

Java

void enter(int statehandle, int sclhandle)

Parameters:

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

sclhandle handle, defined by VT_scldef. Use VT_NOSCL if you don't have a specific value.

Returns:

error code

int VT_leave (int *sclhandle*)

Mark the end of a function.

See also VT_begin().

Fortran

VTLEAVE(sclhandle, ierr)

Java

void leave(int sclhandle)

Parameters:

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

Returns:

error code

int VT_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

7.5 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 it which can (but does not have to) be passed to VT_scopeend(). If the application does not need this flexibility, then it can simply pass 0 to VT_scopeend().

Fortran

VTSCOPEBEGIN(scopehandle, scl, seqnr, ierr)

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 one 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

7.6 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 $<=$ 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, one 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.

7.7 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 (e.g. 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 must have been 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
VTCOUNTDEF(name, classhandle, genre, target, bounds[], unit, counterhandle, ierr)

Parameters:

name string identifying the counter

classhandle class to group counters, handle must have been retrieved by VT_classdef

genre bitwise or of one value from VT_CountScope, VT_CountDisplay and VT_-CountData

target target which the counter refers to (VT_ME, VT_GROUP_THREAD, VT_GROUP_PROCESS, VT_GROUP_CLUSTER or thread/process-id or user-defined group handle).

bounds array of lower and upper bounds (2x 64 bit float, 2x2 32 bit integer, 2x 64 bit integer -> 16 byte)

unit string identifying the unit for the counter (like Volt, pints etc.)

Return values:

counterhandle handle identifying the defined counter

Returns:

error code

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

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

At any time during execution, a process can record a new value for any of the defined counters by calling one of the 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 `counterscopec.c`, which demonstrates all of these facilities.

7.8 Defining Frames

Frames are a new concept implemented in the structured tracefile format (STF) as supported by ITC (see section 6 for details about STF). A frame is a subset of a trace file, identified by any combination of the following

- a time interval (start and end time), defined by calling VT_framebegin() resp. VT_frameend()
- a subset of threads, defined by who calls VT_framedef()
- a subset of data categories:

enum _VT_Categories
Enumeration values:

VT_CAT_ANY_DATA	special value that matches everything.
VT_CAT_FUNCTIONS	function entry/exits (strictly stack oriented).
VT_CAT_SCOPES	scope start/ends (may overlap).
VT_CAT_OPENMP	OpenMP support.
VT_CAT_FILEIO	MPI-IO.
VT_CAT_COUNTERS	counter values.
VT_CAT_MESSAGES	one-to-one communication.
VT_CAT_COLLOPS	communication among more than two threads.

ITC can automatically define frames controlled by the configuration mechanisms described in chapter 8; it is, however, also possible to define frames and create instances of them with API calls. For most applications, this will not be necessary—please use the configuration mechanisms since the proper use of the frame API is quite complex.

A frame set is identified by its name and either belongs to (and contains) all threads of a process or just those threads that define it:

enum _VT_FrameScope
Enumeration values:

VT_FRAME_PROCESS	register all threads of process.
VT_FRAME_THREAD	register calling thread only.

ITC automatically assigns a frame handle for future reference to the newly defined frame set. After definition, the processes can now create frames belonging to the frame set by starting data collection into the frame and ending it later. Since a frame cannot contain disjoint time intervals for any of its processes, starting to collect data into a frame creates a new instance of it, which will be completed when the process ends data collection. Each frame is identified by a label string passed with the frame start call.

The frame definition is handled by the VT_framedef() routine:

int VT_framedef (const char ∗ *type*, int *categories*, int *target*, int ∗ *frame_handle*)
Define a frame.
Fortran
VTFRAMEDEF(type, categories, target, frame_handle, ierr)
Java
int frameDef(final String type, int categories, int target)
Parameters:

type string which uniquely identifies frame (must not be NULL)

categories Ored values representing categories (enum _VT_Categories) to be held in frame

target VT_FRAME_PROCESS for registering all threads in a process at once; subsequent calls to VT_framebegin/end must be done only once per frame instance (the entry- and exit-points must be indicated by one thread only). VT_FRAME_THREAD for registering each thread individualy; subsequent calls to VT_framebegin/end must be done by each thread.

Return values:

frame_handle handle is stored here

Returns:
error code

To start collecting data into a frame (and thus create a new frame instance), call the VT_framebegin() routine, passing a label to identify the newly created frame instance. To end an active frame instance, call the VT_frameend() routine. Please note that all processes in a frame have to call VT_framebegin() and VT_frameend() in a loosely synchronous way to create a new instance - ITC will match up the first calls to VT_framebegin() to start the first instance, then the first calls to VT_frameend() to stop that instance, starting over with the second calls to VT_framebegin() etc. It is possible to have arbitrary overlaps between frames with different frame handles. If the frame set was defined with VT_FRAME_THREAD, then every thread in each participating process must call these functions.

int VT_framebegin (const char ∗ *label*, int *frame_handle*)

Let a given frame begin for calling thread (VT_FRAME_THREAD) or the whole process (VT_FRAME_PROCESS).

For all threads in the frame this function must be called in the same order and equally often.

Fortran

VTFRAMEBEGIN(label, framehandle, ierr)

Java

void frameBegin(final String label, int frame_handle)

Parameters:

label string which identifies frame instance (may be empty)

frame_handle handle identifying frame (from VT_framedef)

Returns:

error code

int VT_frameend (int *frame_handle*)

Let a given frame end for calling thread (VT_FRAME_THREAD) or the whole process (VT_FRAME_PROCESS).

For all threads in frame this function must be called in the same order and equally often and must follow a VT_framebegin.

Fortran

VTFRAMEEND(frame_handle, ierr)

Java

void frameEnd(int frame_handle)

Parameters:

frame_handle handle identifying frame (from VT_framedef)

Returns:

error code

7.9 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_scldef, or VT_NOSCL

Returns:

error code

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

Logs receiving of a message.

Fortran

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

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_scldef, or VT_NOSCL

Returns:

error code

The next three calls require a little extra care, because they generate events that not only have a time stamp, but also a duration. This means that one needs 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 transfered 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 transfered 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_scldef, or VT_NOSCL

Returns:

error code

enum _VT_OpTypes

These are operation ids that can be passed to VT_log_op().

Their representation in the trace file matches that of the equivalent MPI operation.

User-defined operations are currently not supported.

Enumeration values:

VT_OP_INVALID	undefined operation, should not be passed to 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 one has to alert ITC of the fact that out-of-order events will follow by calling VT_begin_unordered() before and VT_end_unnordered() 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()

7.10 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 >= 0, contacts != NULL)

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

This call replaces starting the VTserver executable in a seperate 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

7.11 C++ API

These are wrappers around the C API calls which simplify instrumentation of C++ source code and ensure correct tracing if exceptions are used. Because all the member functions are provided as inline functions it is sufficient to include VT.h to use these classes with every C++ compiler.

Here are some examples how the C++ API can be used. `nohandles()` uses the simpler interface without storing handles, while `handles()` saves these handles in static instances of the definition classes for later reuse when the function is called again:

```
void nohandles()
{
    VT_Function func( "nohandles", "C++ API", __FILE__, __LINE__ );
}
void handles()
{
    static VT_SclDef scldef( __FILE__, __LINE__ );
        // VT_SCL_DEF_CXX( scldef ) could be used instead
    static VT_FuncDef funcdef( "handles", "C++ API" );
    VT_Function func( funcdef, scldef );
}
int main( int argc, char **argv )
{
    VT_Region region( "call nohandles()", "main" );
    nohandles();
    region.end();
    handles();
    handles();
    return 0;
}
```

7.11.1 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 ()

7.11.1.1 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.

7.11.1.2 Constructor & Destructor Documentation

7.11.1.3 VT_FuncDef::VT_FuncDef (const char ∗ *symname*, const char ∗ *classname*)

7.11.1.4 Member Function Documentation

7.11.1.5 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.

7.11.2 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 ()

7.11.2.1 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().

7.11.2.2 Constructor & Destructor Documentation

7.11.2.3 VT_SclDef::VT_SclDef (const char ∗ *file*, int *line*)

7.11.2.4 Member Function Documentation

7.11.2.5 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

7.11.3 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 ()

7.11.3.1 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.

7.11.3.2 Constructor & Destructor Documentation

7.11.3.3 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

7.11.3.4 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

7.11.3.5 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

7.11.3.6 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

7.11.3.7 VT_Function::~VT_Function ()

the destructor marks the function exit.

7.11.4 VT_Region Class Reference

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

Public Methods

- void begin (const char ∗symname, const char ∗classname)
- void begin (const char ∗symname, const char ∗classname, const char ∗file, int line)
- void begin (VT_FuncDef &funcdef)
- void begin (VT_FuncDef &funcdef, VT_SclDef &scldef)
- void end ()
- void end (const char ∗file, int line)
- void end (VT_SclDef &scldef)
- VT_Region ()
- VT_Region (const char ∗symname, const char ∗classname)
- VT_Region (const char ∗symname, const char ∗classname, const char ∗file, int line)
- VT_Region (VT_FuncDef &funcdef)
- VT_Region (VT_FuncDef &funcdef, VT_SclDef &scldef)
- ∼VT_Region ()

7.11.4.1 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.

7.11.4.2 Constructor & Destructor Documentation

7.11.4.3 VT_Region::VT_Region ()

The default constructor does not start the region yet.

7.11.4.4 VT_Region::VT_Region (const char ∗ *symname*, const char ∗ *classname*)

Enter region when it is created.

7.11.4.5 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.

7.11.4.6 VT_Region::VT_Region (VT_FuncDef & *funcdef*)

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

7.11.4.7 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.

7.11.4.8 VT_Region::∼VT_Region ()

the destructor marks the region exit.

7.11.4.9 Member Function Documentation

7.11.4.10 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

7.11.4.11 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

7.11.4.12 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

7.11.4.13 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

7.11.4.14 void VT_Region::end ()

Leaves the region.

7.11.4.15 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

7.11.4.16 void VT_Region::end (VT_SclDef & *scldef*)

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

Parameters:

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

Chapter 8

ITC Configuration

8.1 Configuring ITC

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

8.2 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 must 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 an 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.

8.3 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 must 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.

8.4 Syntax of Parameters

8.4.1 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.

8.4.2 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”.

8.4.3 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.

8.5 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 must 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 tracefile. 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

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.

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.

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 a 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

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 rules 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, i.e. 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 "on", so data is collected and thumbnails and statistics will be calculated, but the collected data is not actually written into the trace file. This mode is useful if looking at frames and the statistics contained in their thumbnails 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_hostname() 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, than 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 refered 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 atttributes of the runtime environment.

STATISTICS

Syntax: [on|off]

Variable: VT_STATISTICS

Default: off

Enables or disables statistics about messages, OpenMP regions 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 one 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 writen 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.

FRAME-USE-HW-STRUCTURE

Syntax: [on|off]

Variable: VT_FRAME_USE_HW_STRUCTURE

Default: usually on

When writing STF, then frames provide precalculated thumbnails of trace data. One frame covers a time interval and a set of processes. You can configure frames independently from the physical layout of the data, but the config options to do that are very similar. This config options corresponds to STF-USE-HW-STRUCTURE.

This option can be combined with PROCS-PER-FRAME.

FRAME-GROUP

Syntax: <group name>

Variable: VT_FRAME_GROUP

This option overrides FRAME-USE-HW-STRUCTURE: instead of using the hardware structure, a group is taken and for each of its members a set of frames is generated. For example, if group ”odd_even” contains groups ”odd” with all processes having an odd process rank and ”even” with the other processes, the FRAME-GROUP ”odd_even” will create frames labeled ”odd” and ”even” covering the two set of processes.

The group may have been defined with the GROUP configuration option, with the API call VT_groupdef() or be one of the predefined groups. However, no distinction is made between threads and processes in these groups: if a thread is listed, then the whole process is inside the corresponding frame.

This option can be combined with PROCS-PER-FRAME.

PROCS-PER-FRAME

Syntax: <number of processes>

Variable: VT_PROCS_PER_FRAME

Default: 16

In addition or instead of calculating frames per node, the number of processes per frame can be limited. Setting it to 0 is the same as setting it to unlimited.

ALL-PROCS-FRAME

Syntax: [on|off]

Variable: VT_ALL_PROCS_FRAME

Default: on

By default one frame called ”all processes” will be created, covering all processes with duration divided exactly like the others, thus giving an overview of the whole machine at each time and simplifying the task of selecting all processes. If this and the ”all” frame mentioned below both cover the whole program run, then only the ”all” frame is generated.

ALL-FRAME

Syntax: [on|off]

Variable: VT_ALL_FRAME

Default: on

By default one frame called ”all” will be created, covering the whole program run and without division in time. Its thumbnails serve as an overview of the the whole program.

SECONDS-PER-FRAME

Syntax: <duration>

Variable: VT_SECONDS_PER_FRAME

Default: 10 minutes

Most frames that are created with config options are divided into parts no longer than this time limit; only the ”all” frame always covers the whole program run, and frames created via the ITC API at runtime are not modified either. <duration> has the usual format for a time value.

FRAMES-PER-RUNTIME

Syntax: <number>

Variable: VT_FRAMES_PER_RUNTIME

Default: 1

Both SECONDS-PER-FRAME and FRAMES-PER-RUNTIME divide the whole program run into frames of equal length. While SECONDS-PER-FRAME results in frames of equal duration, FRAMES-PER-RUNTIME produces the same, fixed number of frames for each program run. ITC will look at both options and pick the larger number of frames, so the default of 1 for FRAMES-PER-RUNTIME basically disables this feature.

Even with SECONDS-PER-FRAME larger than the program's runtime and FRAMES-PER-RUNTIME set to 1 there may be more than one frame if FRAME-USE-HW-STRUCTURE or PROCS-PER-FRAME produce frames for specific processes.

DATA-PER-FRAME

Syntax: <number of bytes>

Variable: VT_DATA_PER_FRAME

Default: 80MB

One advantage of frames is that they allow selective loading of trace data, but this only works well if frames don't include too much trace data. Having frames that include equal amounts of data (and thus events) also helps to zoom into regions of high activity.

This option sets an upper limit for the amount of data in main memory that is stored in (and thus needs to be loaded from) frames with the same time intervals. For applications which log many events the default values usually lead to shorter frames than specified in SECONDS-PER-FRAME. Setting SECONDS-PER-FRAME to an even higher value leads to frames that are generated by their amount of data as the only criterion.

Note that the data is stored more efficiently in STF files, so the resulting number of frames will be higher than the final trace file size divided by the specified amount of data per frame.

The advantages of looking at data in memory are that communication between processes during trace file writing can be avoided and that the resulting frames are tailored for tools that may have to load them completely for analysis.

FRAMES-MAXNUM

Syntax: <number>

Variable: VT_FRAMES_MAXNUM

Default: 0

This option sets an upper limit for the total number of frames generated by ITC. Because the current release of ITA does not need frames, writing any frame is disabled by default.

Increasing the number enables writing of frames. Then the it can be used as a safeguard against unintentionally generating a large number of frames, which can happen e.g. with a low value for SECONDS-PER-FRAME and a long program run. It has no effect on frames created via the ITC API.

FRAME

Syntax: "<type>", <thread triplets>, <categories>, <duration>, <window>

Variable: VT_FRAME

This option defines a new frame for certain categories and threads. The <duration> corresponds to SECONDS-PER-FRAME, but the value is valid for this frame type alone. If a window is given (in the form <timespec>:<timespec> with at least one unit descriptor), frames are created only inside 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, i.e. use "10s" instead of just "10". The <type> can be any kind of string in single or double quotation marks, but it should uniquely identify the kind of data combined into this frame. Valid <categories> are FUNCTIONS, SCOPES, OPENMP, FILEIO, COUNTERS, MESSAGES, COLLOPS.

All of the arguments are optional and default to "unnamed frame", 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.

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 (e.g. those for the nodes in the machine), however, groups generated with API functions must 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
```

THUMBNAIL

Syntax: <pattern> [on|off]

Variable: VT_THUMBNAIL

Default: on

Enables or disables those thumbnails whose name matches the pattern.

MESSAGE-THUMB-SIZE

Syntax: <size>

Variable: VT_MESSAGE_THUMB_SIZE

Default: 32

This option limits the size of the "Sent Message Statistics" thumbnail in the x and y directions. Without this limit the thumbnail would require space proportional to the number of processes squared, which does not scale for large number of processes.

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, e.g. 1m for one minute.

SYNC-DURATION

Syntax: <duration>

Variable: VT_SYNC_DURATION

Default: -1 for MPI applications unless mpich is used, 2 seconds otherwise

ITC usually uses a barrier at the beginning and the end of the program run to take synchronized time stamps on processes. This method may fail if a barrier does not synchronize the processes well enough. In this case an advanced algorithm based on statistical analysis of message round-trip times might yield better results. It also requires several seconds of message exchanges at the beginning and the end of the program run, though. <duration> has the usual format of time values in ITC.

This options enables this algorithm by setting the number of seconds that ITC exchanges messages among processes. A value less or equal zero disables the statistical algorithm. A good number of seconds to start with is 10.

SYNCED-CLUSTER

Syntax: [on|off]

Variable: VT_SYNCED_CLUSTER

If enabled, then ITC assumes that processes running on any host use the same clock and does no clock synchronization itself, unless you explicitly enable the statistical sampling algorithm by setting SYNC-DURATION.

The default value of this option is taken from the MPI attribute MPI_WTIME_IS_GLOBAL if ITC uses MPI_Wtime() as clock.

SYNCED-HOST

Syntax: [on|off]

Variable: VT_SYNCED_HOST

Default: on

If enabled, then ITC assumes that processes running on the same host use the same clock and only synchronizes the clocks of different hosts. Because clock synchronization cannot achieve perfect results avoiding it whenever possible is desirable.

Currently this option is enabled by default on all platforms. If the MPI attribute MPI_WTIME_IS_GLOBAL is set to true, then this config option is irrelevant and the result of MPI_Wtime() is taken as it is without any clock correction.

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 must 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.

8.6 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.

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:

- 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 you must unfold it.

- Counter sampling can be disabled for states (and in a similar way for OpenMP regions).

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 must 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 or when an OpenMP region starts or ends. 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
```

8.7 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

FAQ - Frequently asked questions

This chapter is divided into a more general and a platform specific part. Please refer to the appropriate part for your questions.

A.1 General questions

A.1.1 Interpretation of a version number

The version numbers consist of a string and four separate numbers:

```
<product> <major>.<minor>.<release>.<internal>
```

The `<product>` string is necessary because there are many different distributions of ITC which are all numbered independently of each other. Some of these distributions are:

PRODUCT: the official product version

ASCI/VGV: a version produced for the Advanced Simulation and Computing Initiative

NEC: Vampirtrace/SX, as distributed by NEC

COMPAQ: Vampirtrace/SC, an enhanced version produced for HP (formerly Compaq)

`<major>` and `<minor>` are incremented if and only if new features are added. Together they can serve as a label for the functionality of a product, as in ”release 3.1 has feature *xyz* that was not found in 3.0”. `<major>` is only incremented after significant changes.

`<internal>` is incremented each time a new package is prepared and ensures that two files with different content also have different versions. It is called ”internal” because it is incremented even if the package is not released to the end customer.

Once the package has been released to the customer, the `<internal>` counter is reset to `0` and the `<release>` counter is incremented. From this rule follows that the version with the highest `<internal>` counter is the one delivered to the customer, and that this counter is not necessarily zero. In general, two packages with the same `<major>.<minor>` version, but a different `<release>` number only differ in the number of bug fixes, so one could say ”release 3.1.0 has bug *abd* which is fixed in 3.1.1”.

A.1.2 How to limit the tracefile size

Although ITC uses a compact binary format to store the trace data, tracefile sizes for real-world applications can get immense. The best approach it to limit the number of events to be logged by scaling down the application, like f.i. iteration count, number of processes, problem size etc. This also shortens the time required to run a test. Quite often, this is not acceptable because reduced input datasets are not available or performance analysis for reduced problems is simply not interesting. In that case there are basically four other options:

- Enable trace data collection for a subset of the application's runtime only: by inserting calls to VT_traceoff() and VT_traceon(), an application programmer can easily limit the profiling to *interesting* parts of an application or a subset of iterations. This will require recompilation of (a subset of) the application though, which may not be possible, or at least inconvenient.
- If the application has a complex call graph e.g. due to automatic function tracing, then folding of functions can prune the call tree a lot at run-time and thus cut down the trace file size. This feature is not supported by all ITC versions.
- Use the activity/symbol filtering mechanism to limit the set of logged events. For this the application doesn't need to be changed in any way. However, the user must have an idea of which events are *interesting* enough to be traced, and which events can be discarded. As every MPI routine call generates roughly the same amount of trace data the possible reduction in data volume is quite high: concentrate on the calls actually communicating data, and don't trace the administrative MPI routines.
- Use the process or node or time interval filters to limit data collection to a subset of processes.

A.1.3 How to limit the memory consumption

During the application run, ITC first stores trace data in memory buffers. There are two options that control the allocation of these buffers: MEM-BLOCKSIZE specifies the size of each memory block in bytes, and MEM-MAXBLOCKS determines the maximum number of memory blocks. ITC will not exceed the memory limits set by `MEM-BLOCKSIZE*MEM-MAXBLOCKS`. When this trace data memory is exhausted, one of three actions are taken:

- If the AUTOFLUSH option is enabled (the default), the collected trace data is flushed to disk, and the trace collection continues. The spill files are automatically merged when the application finalizes, so that all records will appear in the tracefile.
- If AUTOFLUSH is disabled and MEM-OVERWRITE is enabled, the trace buffers will be overwritten from the beginning, in effect recording the last n records.
- Else, the trace collection will be stopped, in effect collecting the first n records.

Placing trace data in main memory can slow down the application if it needs the memory itself. Setting MEM-MAXBLOCKS puts a hard limit on the amount of memory used by ITC, but can disrupt the application when a process must wait for flushing of trace data. To avoid this, ITC can be told to start flushing earlier in the background with the MEM-FLUSHBLOCKS option. This option is only available in more recent thread-safe versions of ITC.

In order to understand how much memory is currently in use, ITC can add counter data to the trace:

Counter Class: VT_BUFFERING		
Counter Name	Unit	Comment
data_in_ram	bytes	amount of trace data stored in main memory
data_in_file	bytes	amount of trace data stored in flush file
flush_active	boolean	unequal zero if background flushing is active

If enabled, each process will store its own values for these counters in the trace each time they change. This makes it possible to take the effect of buffer handling into account when doing the analysis of the trace. These counters are not enabled by default. It is necessary to add the following lines to a configuration file (see usage of VT_CONFIG) to enable each counter:

```
COUNTER data_in_ram  ON
COUNTER data_in_file ON
COUNTER flush_active ON
```

At runtime, ITC also provides feedback on the amount of data collected: with the default setting of 500MB for the MEM-INFO configuration option a message is printed each time more than this amount of new data is recorded by a process. The value is chosen so that the message serves as a warning when the amount of trace data exceeds the amount that can usually be handled without problems. In order to use it as a kind of progress report a much lower value would be more appropriate.

A.1.4 How to manage ITC API calls

The API routines greatly extend the functionality of ITC. Unfortunately, manually instrumenting the application source code with the ITC API makes code maintenance harder. An application that contains calls to the ITC API requires the ITC library to link and incurs a certain profiling overhead. The *dummy* API library `libVTnull.a` helps in this situation: all the API calls map to empty subroutines, and no trace data is ever gathered if an application is linked to it. Still, the extraneous function calls remain and may cause a slight overhead.

It is recommended that the C pre-processor (or an equivalent tool for Fortran) be used to guard all the calls to the ITC API by `#ifdef` directives. This will allow easy generation of a plain vanilla version and an instrumented version of a program.

A.1.5 What happens if a program fails ?

The ITC library stores trace data first in buffers in the application memory, and then in flush files (one per MPI process) when the buffers have been filled. In normal operation, the library will merge the trace data from each process during execution of the `MPI_Finalize()` routine, and write the trace data into a single tracefile suitable for input to ITA. If a program fails, `MPI_Finalize()` is never executed, and no ITA tracefile is written.

A.1.6 Troubleshooting

The ITC library can report four basic error classes:

1. Setup errors
2. Invalid configuration file format
3. Erroneous use of the API routines

4. Insufficient memory

The first category includes invalid settings of the VT_ environment variables, failure to open the specified tracefile etc. A warning message is printed, the library ignores the erroneous setup and tries to continue with default settings.

For the second class, a warning message is printed, the faulty configuration file line is ignored, and the parser continues with the next line.

When an ITC API routine is called with invalid parameters, a negative value is returned (as a function result in C, in the error parameter in Fortran), and operation continues. Invoking any API routines before `MPI_Init()` or after `MPI_Finalize()` is considered erroneous, and the call is silently ignored.

An insufficient memory error can occur during execution of an API routine or within any MPI routine if tracing is enabled. In the first case, an error code (`VT_ENOMEM` or `VTENOMEM`) is returned to the calling process; in any case, ITC prints an error message and attempts to continue by disabling the collection of trace data. Within `MPI_Finalize()`, the library will try to generate a tracefile from the data gathered before the *insufficient memory error*.

Although ITC tries to handle out-of-memory situations gracefully, library calls in the application might not be as tolerant, or the operating system does not handle such a situation well enough. To avoid a memory error in the first place, try to limit the amount of trace data as explained in the section “Limiting Memory Consumption” (A.1.3). The memory requirements of ITC can be reduced with the MEM-BLOCKSIZE and MEM-MAXBLOCKS config options. The AUTOFLUSH option needs to remain enabled if you want to see a trace of the whole application run.

A.1.7 Can’t find the tracefile

Unless told otherwise in the configuration file, ITC will write the trace data to the file `argv[0].stf`, with `argv[0]` being the application name in the command line (same as `getarg(0)` in Fortran). Note that your MPI library or the MPI execution script may interfere with `argv[0]`, and that only the process actually writing the tracefile (usually the one with rank 0 in `MPI_COMM_WORLD`) will look at it. A relative pathname will be interpreted relative to that process’ current working directory.

You can however change the tracefile name with the LOGFILE-NAME directive in a configuration file.

If it turns out that ITC can’t create the specified tracefile, it will attempt to write to the file `/tmp/VT-<pid>.stf`, with `<pid>` being the Unix process id of the tracefile-writing MPI process.

In any case, an information message with the actual tracefile name will be printed by ITC within `MPI_Finalize()`.

On systems where not all processes see the same files, be sure to look for the tracefile in the correct process’ filesystem. You can influence which process will write the file by setting an environment variable or by a directive in the configuration file.

A.1.8 User-defined activities don’t work

In order to minimize the instrumentation overhead, ITC does not check for global consistency of the activity codes specified by calls to VT_symdef() or `VTSYMDEF()`. It is the user’s responsibility to ensure that

- The same code is used for the same activity in all processes

- Two different symbols never share the same code

If these rules are violated, ITA might complain about duplicate activities, or activities may be mis-labeled in ITA displays.

A.1.9 Messages are not shown

In order for messages to be indicated in the ITA displays, both the calls to the sending and the receiving MPI routine must have been traced. For nonblocking receives, the call to the MPI wait or test routine that *did complete* the receive request must be logged.

If tracing has been disabled during runtime it can happen that for some messages, either the sending or the receiving call has not been traced. As a consequence, these messages are not shown by ITA, and other messages can appear to be sent to or received at the wrong place. Similarly, filtering out some of the above mentioned MPI routines has the same effect.

A.1.10 Does ITC support MPI-IO?

MPI-IO statistics can be investigated in ITA with the display (Global Displays:I/O Events Statistics). The display option is available in all ITA 4.x versions. If a tracefile contains MPI-IO trace data, this option can be used to display it. If a tracefile does not include MPI-IO data, then there is nothing to be displayed.

ITC only supports standard MPI-IO, that is if the according MPI release implements the full and standard compliant MPI-IO functionality.

Platforms on which ITC can record MPI-IO trace data:

- IBM AIX 5.1 (Product 3.0 and above)
- Sparc Solaris 2.8 (Product 3.0 and above)
- Intel Itanium with SGI MPT 1.8 (Product 3.1 and above)
- Fujitsu VPP
- Hitachi SR8000
- NEC SX

A.1.11 Bad Clock Resolution

If the clock resolution is very low, i.e. the timer function returns the same value for a long period of time, then many events will be recorded on the same time stamp and analysis of such a trace becomes very hard. In particular the Global Timeline becomes useless.

ITC 4.0.2 will issue a warning like “minimum clock increment 1e-3s is very high, please fix system setup to obtain better traces” if it detects this. The minimum clock increment is always stored in the trace file infos, because the timer base also listed there may be lower than the real value.

This problem was observed on some versions of Tru64 and certain Red Hat EL3.0 kernel versions for Itanium. The following releases (see `/etc/r*release*` on a Red Hat derived system) are definitely affected:

- Rocks release 3.1.0 (Matterhorn)

These are not:

- Rocks release 3.2.0
- Red Hat Enterprise Linux AS release 3 (Taroon Update 1), kernel 2.4.21-9.EL
- Red Hat Linux Advanced Server release 2.1AS (Derry), kernel 2.4.18-e.40smp

A.2 Platfrom specific questions

A.2.1 Linux: Can't find libelf

If you compile your MPI program on Linux you may run into the following linker problem.

```
/usr/bin/ld: cannot find -lelf
```

This means that the linker cannot find the libelf.a library. Some distributions don't install this library by default. In some older ITC distributions it wasn't included either, so you had to install this package from your Linux installation media. But now this error should no longer occur because now a version of libelf is included in the same directory as libVT itself.

A.2.2 AIX 5.1: Undefined symbol

If you get the following error message on IBM AIX 5.1 when linking a MPI program:

```
mpxlf90 -o hello hello.f -L$PAL_ROOT/lib -lVT -lld
** hello   === End of Compilation 1 ===
1501-510  Compilation successful for file hello.f.
ld: 0711-317 ERROR: Undefined symbol: mpi_status_ignore
ld: 0711-317 ERROR: Undefined symbol: mpi_statuses_ignore
ld: 0711-345 Use the -bloadmap or -bnoquiet option
to obtain more information.
```

Please compile/link with `mpxlf90_r` or `mpxlf_r` which use an updated version of MPI.

A.2.3 ITC and ScaMPI

In addition to page 4 in the ITC UserGuide, we have to admit that there is an exception to the rule that libVT.a has to be included before the systems MPI libraries. If you use the Scali MPI implementation ScaMPI than you need to use

```
-lfmpi -lVT -lmpi
```

This is necessary because the ScaMPI Fortran library libfmpi.so is a Fortran wrapper to the MPI functions in the libmpi.so library. The libmpi.so library have weak symbols on `MPI_*` with true functions `PMPI_` to support an external trace library. Since the functions in libfmpi.so (`mpi_*`) calls the `MPI_*` functions, the ITC library for C should be suited for generating the trace info.

A.2.4 ITC and Quadrics MPI

When writing a large trace of a Quadrics MPI run you may get errors like “THRD: elan3_alloc: Exhausted ALLOC” or “elan_baseInit: Failed to allocate vaddr space”. These occur because ITC sends many messages which Quadrics MPI considers as small and thus buffers them without waiting for the recipient. Eventually this overflows the available buffers. Some versions of Quadrics MPI also had a memory handling bug.

There are several independent solutions to this problem which all work by configuring ITC or MPI via environment variables. They are listed here in the order in which they should be tried:

1. VT_MEM_BLOCKSIZE=128KB - increases the chunk size used by ITC so that Quadrics MPI switches to a blocking send mode
2. LIBELAN_TPORT_BIGMSG=32768 - decreases the threshold in Quadrics MPI to achieve the same result, but may also have a negative effect on application performance
3. MPI_USE_LIBELAN_SUB=0 - disables usage of Elan library in Quadrics MPI and thus avoids the problematic code altogether

A.2.5 Error: Unsupported Architecture

We do not test or validate the Intel® Trace Collector on systems using non-Intel processors. Because of potential architectural differences, we cannot ensure that crucial performance results are correct. Therefore, rather than allow test or validations that could lead to potentially incorrect results, we prevent our tool from running on systems using non-Intel processors.

Index