Vampirtrace MIPS-IRI-65 PRODUCT.4.0.0.0

User’s Guide PRODUCT.4.0.0.0

PALLAS GmbH
Hermülheimer Straße 10
D–50321 Brühl, Germany

This product includes software developed by the University of California, Berkley and its contributors, and software derived from the Xerox Secure Hash Function.

Contents

Chapter 1

Introduction

1.1 What is Vampirtrace?

The Vampirtrace profiling tool for MPI applications produces tracefiles that can be analyzed with the Vampir performance analysis tool.

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.

Vampirtrace is an add-on for existing MPI implementations; using it merely requires relinking the application with the Vampirtrace 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 Vampirtrace API (see section 5) 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 Vampirtrace 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 Vampirtrace was compiled for: MIPS Irix 6.5 MPI 3.3.0.6 (MPT 1.5.3)

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 Vampirtrace platform list at `http://www.pallas.com/e/products/pdf/Vampirtrace-Platforms.pdf`. 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 Vampirtrace will not work:

```
Datatypes:

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

C constants:

   MPI_CHAR            : 1
   MPI_BYTE            : 27
   MPI_SHORT           : 2
   MPI_INT             : 3
   MPI_FLOAT           : 9
   MPI_DOUBLE          : 10
   MPI_COMM_WORLD      : 1
   MPI_COMM_SELF       : 2

MPI_Status structure and byte offsets of members:

   MPI_STATUS_SIZE     : 6
   MPI_SOURCE          : 0
   MPI_TAG             : 4
   MPI_ERROR           : 8
```

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

The following features are supported:

Feature	Description
Thread-safety	supported, see 1.3
MPI tracing • MPI-IO • MPI One-Sided Communication • MPI-2	3.1 not supported not supported not supported
Single-process tracing	3.2
Subroutine tracing	3.5
Counter tracing	API in 5.7
Automatically Recording Source Location Information	3.4 (requires compiler support)
Manually Recording Source Location Information	API in 5.3
Recording Statistical Information	3.3
Folding Complex Call Trees	6.6
Nonblocking Flushing	MEM-FLUSHBLOCKS

1.3 Multithreading

This version of the Vampirtrace library is thread-safe in the sense that all of their 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.

1.4 About this Manual

This manual describes how to use Vampirtrace. Some of the text is also provided as man pages for easier reading in a shell, e.g. the Vampirtrace API calls (man VT_enter) and the Vampirtrace 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 Vampirtrace 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 Vampirtrace archive in a directory of your choice you need to enter this directory and execute the “install-Vampirtrace” located there.

After asking about the desired read and write permissions this script sets the permissions of the Vampirtrace 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.

It is possible to install different versions of Vampirtrace 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.

In order to use Vampirtrace on a system, you must have a license key from Pallas. The license keys for Pallas products are stored in a plain ASCII file where each line may contain a separate license key. Lines starting with a hash character # are interpreted as comments. The pathname must be made known to Vampirtrace by setting at least one of three environment variables:

VT_ROOT points to the root of the Vampirtrace installation. It is used to find the Vampirtrace include and library files when compiling programs in the example makefile. A license file will be found if placed in $(VT_ROOT)/etc/license.dat. This variable is automatically set by the sourceme scripts, so this is the easiest place to put the license file.

PAL_LICENSEFILE specifies the complete pathname of the license key file. A relative pathname is interpreted starting from the user’s home directory. Both Vampir and Vampirtrace will find their license in this file, so this works well if both are installed on the same machine or the installations and licenses need to be updated seperately.

PAL_ROOT points to the root of the Vampir installation. As Vampirtrace can share a license file with Vampir it will also search $(PAL_ROOT)/etc/license.dat for a valid license.

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

```
Vampirtrace: Could not access license file.
Vampirtrace: Need license for product VT30.
Vampirtrace: platform IA32-LIN, license class Q,
Vampirtrace: hostid 0xa8c02003, network address 192.168.3.32, userid 41D
Vampirtrace: Built for MPICH 1.2.3.
Vampirtrace: Environment variable PAL_ROOT not set.
```

```
Vampirtrace: Environment variable VT_ROOT not set.
Vampirtrace: Environment variable PAL_LICENSEFILE not set.
```

In this case, make sure that the environment variables mentioned above are correctly set and that the files they point to are readable. To acquire a demo license, please visit `http://www.pallas.com/e/products/vampir/download.htm`. For a permanent license, please compile and run the `vtlcheck` executable in the examples directory on every need that you want a license for. Then contact support@pallas.com and include the output that `vtlcheck` printed to stderr.

Chapter 3

How to Use Vampirtrace

3.1 Tracing MPI Applications

Using Vampirtrace 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 Vampir, 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 Vampirtrace API (see section 5) 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 Vampirtrace; if your MPI is different from the one Vampirtrace 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 Vampirtrace

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

Source files that do contain calls to the Vampirtrace 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 Vampirtrace 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 Vampirtrace

The Vampirtrace 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:

```
cc -n32 -mips3 -U__INLINE_INTRINSICS ctest.o -L$(VT_ROOT)/lib32
   -lVT -lmpi -lexc -ldwarf -lnsl -lm -lelf -lpthread -o ctest
```

```
f77 -n32 -mips3 -U__INLINE_INTRINSICS ftest.o -L$(VT_ROOT)/lib32
   -lVT -lmpi -lexc -ldwarf -lnsl -lm -lelf -lpthread -o ftest
```

If your MPI installation is different, then the command may differ and/or you might have to add further libraries manually. In this case it is important that the Vampirtrace library is listed on the command line in front of the MPI libraries, and the binary interface of the MPI libraries must match the one used by Vampirtrace (see section 1.2 and `http://www.pallas.com/e/products/pdf/Vampirtrace-Platforms.pdf` for details).

3.1.3 Running MPI Programs with Vampirtrace

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

VT_CONFIG contains the pathname of a Vampirtrace 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 Vampirtrace 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>.stf.single`

old-style binary Vampir format `<trace>.bvt`

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 Vampirtrace 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 Vampirtrace package was compiled for. If you use a different MPI, then you might have to edit this GNUmakefile. Unless Vampirtrace 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 Vampirtrace. 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. The most up-to-date version of the FAQ is found at `http://www.pallas.com/e/products/pdf/Vampirtrace-FAQ.pdf`.

If this still does not help, the please refer to the chapter 7 “How to Create an Error Report” and send the information mentioned there to support@pallas.com.

3.2 Single-process Tracing

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

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

```
-lVTsp -lexc -ldwarf -lnsl -lm -lelf -lpthread
```

The application must call VT_initialize() and VT_finalize() to generate a trace. Subroutine tracing (3.5) can be used with and without further Vampirtrace API (see chapter 5) calls to actually generate trace events.

3.3 Recording Statistical Information

Vampirtrace 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 4.4.1) 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 Vampirtrace 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.

3.4 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 Vampirtrace:

```
cc -n32 -mips3 -U__INLINE_INTRINSICS -g  -c ctest.c
f77 -n32 -mips3 -U__INLINE_INTRINSICS -g  -c ftest.c
```

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

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 Vampirtrace API can provide similar information but without the performance overhead (see VT_scldef()/VT_thisloc() in section 5.3 for more information). Automatically recording the locations of subroutine calls in the source code is not supported on this platform. Manual instrumentation of the source code with the Vampirtrace API can provide similar information (see VT_scldef()/VT_thisloc() in section 5.3 for more information).

3.5 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 6.6.

3.6 Recording Hardware Performance Information

Vampirtrace 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 Vampirtrace.

This layer is implemented in the `VT_sample.c` source file. It was not possible to provide a precompiled 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 Vampirtrace 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 Vampirtrace 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:

```
cc -n32 -mips3 -U__INLINE_INTRINSICS ctest.o <search path for
   PAPI> -L$(VT_ROOT)/lib32 -lVTsample -lVT -lpapi -lmpi -lexc
   -ldwarf -lnsl -lm -lelf -lpthread <libs required by PAPI> -o
   ctest
```

Then the application must be run with configuration options that enable the counters of interest. Because Vampirtrace 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 6 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 Vampirtrace 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. Vampirtrace 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
```

3.7 Using the Dummy Libraries

Programs containing calls to the Vampirtrace API (see section 5) 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 Vampirtrace library directory. Here’s how a C MPI-application would be linked:

```
cc -n32 -mips3 -U__INLINE_INTRINSICS ctest.o -L$(VT_ROOT)/lib32
   -lVTnull -lmpi  -o ctest
```

Chapter 4

Structured Tracefile Format

4.1 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 Vampirtrace API (see section 5.8) into the executable, or before starting the application by specifying the configuration options discussed in section 6. 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 Vampir 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
- provides precalculated thumbnails of the data that can be displayed without having to load all of the data itself
- 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.

4.2 STF Components

A structured tracefile actually consists of a number of files as shown in the figure 4.1. 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 Vampir to “fast-forward” within the record files; they can be deleted, but that may result in slower operation of Vampir.

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 4.4.1).

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.

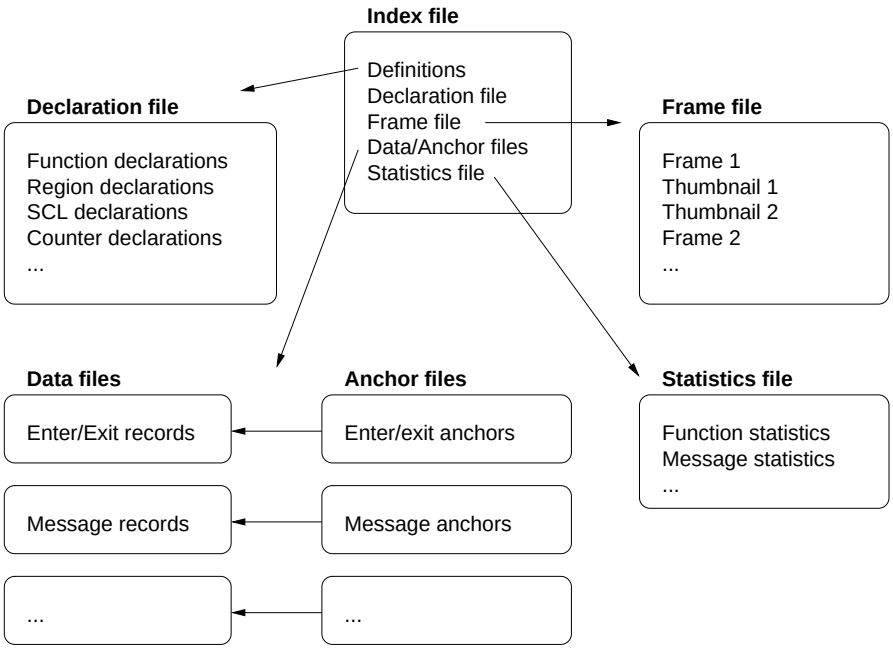


Figure 4.1: STF components

4.3 Single-File STF

As a new option in Vampirtrace, 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 .stf.single. 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 Vampir 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.

4.4 Configuring STF

The two main aspects of the STF behavior that can be configured using directives in the Vampirtrace configuration file or the equivalent environment variables as described in section 6 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. Vampirtrace 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 Vampirtrace 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.

4.4.1 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 Vampirtrace: with a config file, environment variables and command line options. The environment variable VT_CONFIG can be set to the name of a Vampirtrace 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:

```
stfttool example.stf --convert example.avt # ASCII
stfttool example.stf --convert -           # ASCII to stdout
stfttool example.stf --convert - --logfile-format STFSINGLE |
   gzip -c >example.stf.single.gz          # gzipped single-file STF
```

Without the --convert switch one can extract certain parts, but only write VTF:

```
stfttool example.stf --frames 1
         --logfile-name example_frame1.avt # extract frame #1 as ASCII
stfttool 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 and BINARY 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 Vampirtrace when generating the trace.

--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 "--log-filename -" and "--log-fileformat 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.

--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 ".bvt" for binary, ".avt" for ASCII, ".stf" for STF and ".stf.single" 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|BINARY|STF|STFSINGLE]

Specifies the format of the tracefile. ASCII and BINARY are the traditional Vampir file formats where all trace data is written into one file. ASCII is human-readable, whereas BINARY is a more compact machine-readable format.

The Structured Trace File (STF) is a binary format which supports storage of trace data in several files and allows Vampir to analyse the data without loading all of it, so it is more scalable. Writing it is only supported by Vampirtrace 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). STFSINGLE rolls all of these files into one (.stf.single), 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 VT, on in stftool

Several events can only be stored in STF, but not in VTF. Vampirtrace 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 Vampir 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 Vampir.

--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 message when trace file writing starts
- 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

Levels larger than 2 may contain output that only makes sense for the developers of VT.

--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)

4.4.2 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 5

User-level Instrumentation with the API

5.1 The Vampirtrace API

The Vampirtrace 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 Vampirtrace 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 this version of the VT library.

#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.

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 Vampirtrace 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.

5.2 Initialization, Termination and Control

Vampirtrace 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 Vampirtrace is terminated. Thus, it is an error to call Vampirtrace 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 Vampirtrace 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 VT 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 libVTsp.a (single process VT) and no MPI.
If the MPI that VT 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().
Fortran
VTINIT(ierr)
Returns:
error code

int VT_finalize (void)
Finalize VT and underlying communication.
It is not an error to call VT_finalize() twice or after a MPI_Finalize().
Fortran
VTFINI(ierr)
Returns:
error code

int VT_getrank (int * *rank*)
Get process index (same as MPI rank within MPI_COMM_WORLD).
Fortran
VTGETRANK(rank, ierr)
Return values:
rank process index 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( )

void VT_traceon (void)
Turn tracing on for thread if it was disabled, otherwise do nothing.
Cannot enable tracing if "PROCESS/CLUSTER NO" was applied to the process in the configuration.
Fortran
VTTRACEON( )

int VT_tracestate (int * *state*)
Get logging state of current thread.
Set by config options PROCESS/CLUSTER, modified by VT_traceon/off().
There are three states:
- 0 = thread is logging
- 1 = thread is currently not logging
- 2 = logging has been turned off completely

Note that different threads within one process may be in state 0 and 1 at the same time because VT_traceon/off() sets the state of the calling thread, but not for the whole process.
State 2 is set via config option "PROCESS/CLUSTER NO" for the whole process and cannot be changed.
Fortran
VTTRACESTATE(state, ierr)
Return values:
state is set to current state
Returns:
error code

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

int VT_symstate (int *statehandle*, int * *on*)
Get filter state of one state.
Set by config options SYMBOL, ACTIVITY.
Note that a state may be active even if the thread's logging state is "off".
Fortran
VTSYMSTATE(statehandle, on, ierr)
Parameters:
statehandle result of VT_funcdef() or VT_symdef()
Return values:
on set to 1 if symbol is active
Returns:
error code

Vampirtrace 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, Vampirtrace continues normally.

int VT_flush (void)
Flushes all trace records from memory into the flush file.
The location of the flush file is controlled by options in the config file. Flushing will be recorded in the trace file as entering and leaving the state VT_API:TRACE_FLUSH with time stamps that indicate the duration of the flushing. Automatic flushing is recorded as VT_API:AUTO_FLUSH.
Fortran
VTFLUSH(ierr)
Returns:
error code

Please refer to section 6 to learn about the MEM-BLOCKSIZE and MEM-MAXBLOCKS configuration directives that control Vampirtrace's memory usage.

Vampirtrace 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( )

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

5.3 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)
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 VT
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.

Vampirtrace 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 Vampirtrace API calls.

int VT_thisloc (int *sclhandle*)
Set source code location for next activity that is logged by VT.
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)
Parameters:
sclhandle handle defined either with VT_scldef()
Returns:
error code

5.4 Defining and Recording Functions or Regions

Vampir 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. Vampir 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 Vampirtrace 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.

5.4.1 New Interface

To simplify the use of user-defined states, a new interface has been introduced for Vampirtrace. 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.
Fortran
VTCLASSDEF(classname, classhandle, ierr)
Parameters:
classname name of the class
Return values:
classhandle the int it points to is set by VT
Returns:
error code

int VT_funcdef (const char * *symname*, int *classhandle*, int * *statehandle*)
Allocates a handle for a state.
This is a replacement for VT_symdef() which doesn't require the application to provide a unique numeric handle.
Fortran
VTFUNCDEF(symname, classhandle, statehandle, ierr)
Parameters:
symname name of the symbol
classhandle handle for the class this symbol belongs to, created with VT_classdef(), or VT_NOCLASS, which is the same as "Application"
Return values:
statehandle the int it points to is set by VT
Returns:
error code

#define VT_NOCLASS
special value for VT_funcdef(): put function into the default class ”Application”.

5.4.2 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.

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

5.4.3 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.

Vampirtrace 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 VT 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)

Parameters:

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

Returns:

error code

---

int VT_beginl (int *statehandle*, int *sclhandle*)

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

Fortran

VTBEGINL(statehandle, sclhandle, ierr)

---

int VT_end (int *statehandle*)

Marks the end of a region.

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

Fortran

VTEND(statehandle, ierr)

Parameters:

statehandle obsolete, pass anything you want

Returns:

error code

---

int VT_endl (int *statehandle*, int *sclhandle*)

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

Fortran

VTENDL(statehandle, sclhandle, ierr)

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

Mark the beginning of a function.

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

Fortran

VTENTER(statehandle, sclhandle, ierr)

Parameters:

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

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

Returns:

error code

int VT_leave (int *sclhandle*)

Mark the end of a function.

See also VT_begin().

Fortran

VTLEAVE(sclhandle, ierr)

Parameters:

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

Returns:

error code

int VT_wakeup (void)

Triggers the same additional actions as logging a function call does.

When Vampirtrace 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 Vampirtrace has no chance to execute these actions. To avoid that, the programmer can insert calls to this function into the source code of the long-running function.

Fortran

VTWAKEUP(ierr)

Returns:

error code

5.5 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() respectively 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.

Fortran

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

Parameters:

scopename the name of the scope

classhandle the class this scope belongs to (defined with VT_classdef())

scl1 any kind of SCL as defined with VT_scldef(), or VT_NOSCL

scl2 any kind of SCL as defined with VT_scldef(), or VT_NOSCL

Return values:

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

Returns:

error code

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

Starts a new instance of the scope previously defined with VT_scopedef().

There can be more than one instance of a scope at the same time. In order to have the flexibility to stop an arbitrary instance, VT assigns an intermediate identifier to it which can (but does not have to) be passed to VT_scopeend(). If the application does not need this flexibility, then it can simply pass 0 to VT_scopeend().

Fortran

VTSCOPEBEGIN(scopehandle, scl, seqnr, ierr)

Parameters:

scopehandle the scope as defined by VT_scopedef()

scl in contrast to the static SCL given in the scope definition this 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)
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

5.6 Defining Groups of Processes

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

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

enum VT_Group
Enumeration values:
VT_ME the calling thread/process.
VT_GROUP_THREAD Group of all threads.
VT_GROUP_PROCESS Group of all processes.
VT_GROUP_CLUSTER Group of all clusters.

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

int VT_getprocid (int *procindex*, int * *procid*)
Get global id for process which is identified by process index.
If threads are supported, then this id refers to the group of all threads within the process, otherwise the result is identical to VT_getthreadid(procindex, 0, procid).
Fortran
VTGETPROCID(procindex, procid, ierr)
Parameters:
procindex index of process (0 <= procindex < N)
Return values:
procidpointer to mem place where id is written to
Returns:
error code

The same works for threads:

int VT_getthreadid (int *procindex*, int *thindex*, int * *threadid*)
Get global id for the thread which is identified by the pair of process and thread index.
Fortran
VTGETTHREADID(procindex, thindex, threadid, ierr)
Parameters:
procindex index of process (0 <= procindex < N)
thindex index of thread
Return values:
threadid pointer to mem place where id is written to
Returns:
error code

int VT_groupdef (const char * *name*, int *n_members*, int * *ids*, int * *grouphandle*)
Defines a new group and returns a handle for it.
Groups are distinguished by their name and their members. The order of group members is preserved, which can lead to groups with the same name and same set of members, but different order of these members.
Fortran
VTGROUPDEF(name, n_members, ids[], grouphandle, ierr)
Parameters:
name the name of the group
n_members number of entries in the *ids* array
ids array where each entry is either:
- VT_ME
- VT_GROUP_THREAD
- VT_GROUP_PROCESS
- VT_GROUP_CLUSTER
- result of VT_getthreadid(), VT_getprocid() or VT_groupdef()

Return values:
grouphandle handle for the new group, or old handle if the group was defined already
Returns:
error code

To generate a new group that includes the processes with even ranks in MPI_COMM_WORLD, 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);
```

5.7 Defining and Recording Counters

Vampirtrace 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. A Vampirtrace 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). Furthermore, the upper and lower bounds can be set to assist Vampir in scaling its displays.

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 Vampirtrace 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 VT 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.)

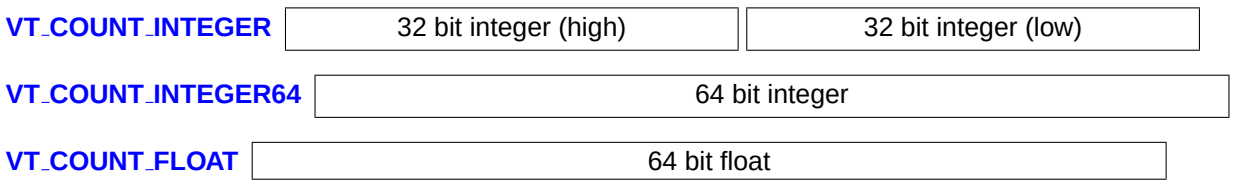
Return values:

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:



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

5.8 Defining Frames

Frames are a new concept implemented in the structured tracefile format (STF) as supported by Vampirtrace (see section 4 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.

Vampirtrace can automatically define frames controlled by the configuration mechanisms described in chapter 6; 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.

Vampirtrace 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)
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 - Vampirtrace 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)
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)
Parameters:
frame_handle handle identifying frame (from VT_framedef)
Returns:
error code

5.9 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;
}
```

5.9.1 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 ()

5.9.1.1 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.

5.9.1.2 Constructor & Destructor Documentation

5.9.1.3 VT_FuncDef::VT_FuncDef (const char * *symname*, const char * *classname*)

5.9.1.4 Member Function Documentation

5.9.1.5 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 VT was not initialized or ran out of memory.

5.9.2 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 ()

5.9.2.1 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().

5.9.2.2 Constructor & Destructor Documentation

5.9.2.3 VT_SclDef::VT_SclDef (const char * *file*, int *line*)

5.9.2.4 Member Function Documentation

5.9.2.5 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 VT 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

5.9.3 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 ()

5.9.3.1 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.

5.9.3.2 Constructor & Destructor Documentation

5.9.3.3 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 VT.

Parameters:

symname the name of the function

classname the class this function belongs to

5.9.3.4 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

5.9.3.5 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

5.9.3.6 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

5.9.3.7 VT_Function::~VT_Function ()

the destructor marks the function exit.

5.9.4 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 ()

5.9.4.1 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.

5.9.4.2 Constructor & Destructor Documentation

5.9.4.3 VT_Region::VT_Region ()

The default constructor does not start the region yet.

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

Enter region when it is created.

5.9.4.5 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.

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

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

5.9.4.7 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.

5.9.4.8 VT_Region::~VT_Region ()

the destructor marks the region exit.

5.9.4.9 Member Function Documentation

5.9.4.10 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 VT.

Parameters:

symname the name of the region

classname the class this region belongs to

5.9.4.11 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

5.9.4.12 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

5.9.4.13 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

5.9.4.14 void VT_Region::end ()

Leaves the region.

5.9.4.15 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

5.9.4.16 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 6

Vampirtrace Configuration

6.1 Configuring Vampirtrace

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

6.2 Specifying a Configuration File

The environment variable VT_CONFIG can be set to the name of a Vampirtrace 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!

6.3 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.

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

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

6.4 Syntax of Parameters

6.4.1 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.

6.4.2 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”.

6.4.3 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.

6.5 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 ”.bvt” for binary, ”.avt” for ASCII, ”.stf” for STF and ”.stf.single” 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 VT 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 Vampirtrace 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|BINARY|STF|STFSINGLE]

Variable: VT_LOGFILE_FORMAT

Specifies the format of the tracefile. ASCII and BINARY are the traditional Vampir file formats where all trace data is written into one file. ASCII is human-readable, whereas BINARY is a more compact machine-readable format.

The Structured Trace File (STF) is a binary format which supports storage of trace data in several files and allows Vampir to analyse the data without loading all of it, so it is more scalable. Writing it is only supported by Vampirtrace 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). STFSINGLE rolls all of these files into one (.stf.single), 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 VT, on in stftool

Several events can only be stored in STF, but not in VTF. Vampirtrace 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 Vampir 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 Vampir.

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

VT 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 message when trace file writing starts
- 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

Levels larger than 2 may contain output that only makes sense for the developers of VT.

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.

SYMBOL

Syntax: <pattern> <filter body>

Variable: VT_SYMBOL

Default: on

Defines a filter for any symbol that matches the pattern. Patterns are ordinary strings and may contain the wild-card character * that matches any number of characters.

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.

ACTIVITY

Syntax: <pattern> <filter body>

Variable: VT_ACTIVITY

Default: on

Defines a filter for any symbol within the activity that is matched by the pattern.

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 Vampirtrace are disabled. Those automatic counters are not supported for every platform.

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. Vampirtrace 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>). 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 Vampirtrace 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

Vampirtrace keeps trace data in chunks of main memory that have this size.

MEM-MAXBLOCKS

Syntax: <maximum number of blocks>

Variable: VT_MEM_MAXBLOCKS

Default: 0

Vampirtrace will never allocate more than this number of blocks in main memory. 0 is the default and allows an unlimited number of blocks. If the maximum number of blocks is filled or allocating new blocks fails, then Vampirtrace 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 Vampirtrace 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 Vampirtrace 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 (with the default setting) 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 Vampirtrace 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: max. int

By default the thread that triggers the flushing does this work itself and thus is blocked while the data is written. Setting this option enables flushing in the background. It has no effect if AUTOFLUSH is disabled. 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.

MEM-OVERWRITE

Syntax: [on|off]

Variable: VT_MEM_OVERWRITE

Default: off

If auto flushing is disabled, then enabling this lets Vampirtrace 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. The file name there 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.

The default is the value of the first, non-empty environment variable in this list that points to a directory, or ”/tmp”:

- BIGTEMP
- FASTTEMP
- TMPDIR
- TMP
- TMPVAR

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 Vampirtrace 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 VT 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 VT (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: VT 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 VT 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>.

STF-CHUNKSIZE

Syntax: <number of bytes>

Variable: VT_STF_CHUNKSIZE

Default: 64KB

VT 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 VT 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. VT 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: 500

This option sets an upper limit for the total number of frames generated by VT. It is a safeguard against using config options that lead to an unexpectedly high (and thus unmanagable) 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 VT 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.

SYNC-DURATION

Syntax: <duration>

Variable: VT_SYNC_DURATION

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

Vampirtrace 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 VT.

This options enables this algorithm by setting the number of seconds that Vampirtrace 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 Vampirtrace 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 Vampirtrace uses MPI_Wtime() as clock.

SYNCED-HOST

Syntax: [on|off]

Variable: VT_SYNCED_HOST

Default: on

If enabled, then Vampirtrace 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 VT 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 Vampirtrace 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 Vampirtrace 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.

6.6 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
SYMBOL 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 ACTIVITY/SYMBOL rule body may offer even finer control over tracing depending on the features available on the current platform:

On the SX-5 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. For example, if foo:bar first calls foo:bar1 and then foo:bar2, then

```
ACTIVITY foo FOLD
```

will only trace the call to foo:bar and ignore the other two function calls. One can unfold certain functions:

```
ACTIVITY foo FOLD
SYMBOL bar2 UNFOLD
```

will trace foo:bar, ignore the call to foo:bar1 (because foo:bar is folded) and unfold again when foo:bar2 is called.

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

Here’s the formal specification:

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

<ACTION>       := traceon|traceoff|restore|none
<TRIGGER>      := [<TRIPLET>] <ACTION>
<ENTRYTRIGGER> := entry <TRIGGER>
<EXITTRIGGER>  := exit  <TRIGGER>
<COUNTERSTATE> := counteron|counteroff
<FOLDING>      := fold|unfold
<RULEENTRY>    := <SCLRANGE>|<ENTRYTRIGGER>|<EXITTRIGGER>|<COUNTERSTATE>|<FOLDING>
```

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 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
```

For each state, only one action for entering and one action for leaving is active, and as usual rules later in the config file overwrite the actions specified in previous rules. The optional triplet specifies for which instances of this state the specified action is triggered. For example, ”SYMBOL MPI_Barrier ENTRY 1:9:2 TRACEON” performs the equivalent of a VT_traceon() the first, third, 9th time MPI_Barrier() is called. Without the triplet, the action is always triggered. Calling a recursive function counts as one instance of this function.

Currently supported actions are TRACEON (same effect as VT_traceon()), TRACEOFF (VT_traceoff()), RESTORE (only valid as an exit action, restores the logging state that was ac-

tive when the state was entered) and NONE (removes actions specified in earlier filter rules).

Here's a complex example that shows several legal combinations:

```
ACTIVITY MPI ON, ENTRY 1 TRACEON, EXIT 2 TRACEOFF
ACTIVITY MPI ENTRY 1 TRACEON , OFF , EXIT 2 TRACEOFF
ACTIVITY MPI ENTRY 1 TRACEON, EXIT 2 TRACEOFF OFF
SYMBOL   MPI_* ON
SYMBOL   MPI_* 0
SYMBOL   "Function with spaces" 1:5
SYMBOL   MPI_Comm_rank ENTRY 1:2 TRACEOFF
ACTIVITY MPI EXIT 1:2:2 RESTORE
ACTIVITY MPI ENTRY TRACEOFF

# reset actions
ACTIVITY   * ENTRY NONE, EXIT NONE
# all processes log MPI, even if tracing is off when they
# are entered, and restore the previous state
ACTIVITY MPI ENTRY TRACEON, EXIT RESTORE
# dummy rule which is overwritten
SYMBOL MPI_Barrier ENTRY 1:N:1 TRACEON
# second barrier turns on, third turns off
SYMBOL MPI_Barrier ENTRY 2 TRACEON
SYMBOL MPI_Barrier EXIT 3 TRACEOFF
# every second instance of function_a turns on logging, starting with 2
SYMBOL "function_a" ENTRY 2:N:2 TRACEON
```

The following rules are illegal for various reasons:

```
# RESTORE not valid for ENTRY
ACTIVITY MPI ENTRY 1 RESTORE
# more than one entry trigger
ACTIVITY MPI ENTRY 1 TRACEON, ENTRY 2 TRACEOFF
# more than one logging state definition
ACTIVITY MPI ON, OFF, 1:5
# invalid action
ACTIVITY MPI ENTRY 1 XYZ - invalid text
```

6.7 The Protocol File

The protocol file has the same syntax and entries as a Vampirtrace 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.

Chapter 7

How to Create an Error Report

In this chapter you will find some helpful information on how to create an error or problem report for Vampirtrace. We really encourage you to provide us with the required information. This will keep the turnaround time for your problem report as low as possible.

If this is a new issue there are several things that you can do right now to help us solve your problem. Start by gathering information about your unique environment:

Operating system version, architecture, compiler version and command line, shell variables, parallel run-time version, program version, command line and stdout, and the run-time environment in which you encountered the problem.

You should also document the steps that lead up to the unusual behavior you are seeing, take screen captures and Save Window to Files. You may also want to take a snapshot of your work: source code, makefiles, executables, DLLs and corefiles.

7.1 Vampirtrace Problem Report Form and Instructions

Copy and paste the form listed below into your email editor. Then, complete and return the form to support@pallas.com.

Document just one problem on a form. Remove or replace all data fields with a selection or with contents. All data requested below is helpful to us, though not all is necessary to solve each problem. Supply as much data as you can. If your problem involves Vampirtrace execution, attach or FTP a reproducible example.

```
To: support@pallas.com

Subject: <copy value of Synopsis field, below  .  .  .  .  .  .  .  .  >

Submitter-Id:  <primary contact's *simplest* E-mail address (one line)>
Originator:    <originator's name .  .  .  .  .  .  .  .  . (one line)>
Confidential:  <[ no | yes ]   .  .  .  .  .  .  .  .  .  . (one line)>
Synopsis:      <synopsis of the problem .  .  .  .  .  .  . (one line)>
Category:      vampirtrace
Class:         <[ sw-bug | doc-bug | change-request | support ] (1 ln)>
Release:
```

```
Environment:

System:     <‘uname -a‘ output .  .  .  .  .  .  .  .  .  .  .  .  >
Platform:   <machine make&model, processor, etc. .  .  .  .  .  .  >
OS:         <OS version and patch level .  .  .  .  .  .  .  .  .  >
ToolChain:  <compiler version, linker version, etc. .  .  .  .  .  >
Libraries:  <versions of parallel runtimes or standard libraries   .  >

Description:
            <precise description of the problem  .  .  .  .  .  . (multiple lines)

How-To-Repeat:

            <step-by-step: how to reproduce the problem   .  .  . (multiple lines)

Fix:
            <how to correct or work around the problem, if known  (multiple lines)
```

7.2 How to Prepare and Send Your Example

Create a directory named repro and place your problem files in it. Add the following files to repro:

- Include the target executable. A test program or code fragment is preferable to a large amount of production code.
- Build the executable statically. If possible, use `-v`.
- Show the compiler version used.
- Show the `compile/build` session (`stdin/stdout/stderr`).
- Include sources (not always necessary, but usually helpful).
- Include any special libraries or input files required.
- Describe the problem. Please be very specific.

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 Vampirtrace 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 Vampirtrace 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 Vampirtrace 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, Vampirtrace 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. Vampirtrace 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 Vampirtrace, but can disrupt the application when a process must wait for flushing of trace data. To avoid this, Vampirtrace 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 Vampirtrace.

In order to understand how much memory is currently in use, Vampirtrace 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, Vampirtrace 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 Vampirtrace API calls

The API routines greatly extend the functionality of Vampirtrace. Unfortunately, manually instrumenting the application source code with the Vampirtrace API makes code maintenance harder. An application that contains calls to the Vampirtrace API requires the Vampirtrace 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 Vampirtrace 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 Vampirtrace 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 Vampir. If a program fails, `MPI_Finalize()` is never executed, and no Vampir tracefile is written.

A.1.6 Troubleshooting

The Vampirtrace 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 a Vampirtrace 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, Vampirtrace 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 Vampirtrace 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 Vampirtrace 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, Vampirtrace 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 Vampirtrace 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 Vampirtrace 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, Vampirtrace 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, Vampir might complain about duplicate activities, or activities may be mis-labeled in Vampir displays.

A.1.9 Messages are not shown

In order for messages to be indicated in the Vampir 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 Vampir, 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 Vampirtrace support MPI-IO?

MPI-IO statistics can be investigated in Vampir with the display (Global Displays:I/O Events Statistics). The display option is available in all Vampir 3.x versions. If a Vampitrace file contains MPI-IO trace data, this option can be used to display it. If a trace file does not include MPI-IO data, then there is nothing to be displayed.

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

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

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

A.1.11 Does Vampirtrace support ROMIO?

The `MPIO_Request` structure as used in ROMIO will not be supported in Vampirtrace, as it does not comform to the MPI-2 standard for parallel I/O. If an MPI implementation advances to compliance with standard `MPI_Request` structures it can be considered for MPI-I/O trace support in a subsequent release of Vampirtrace.

We would encourage users who need to trace I/O relevant information with Vampirtrace to use the MPI-2 standard for parallel I/O.

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, so you have to install this package from your Linux installation media.

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 Vampirtrace and ScaMPI

In addition to page 4 in the Vampirtrace 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 Vampirtrace library for C should be suited for generating the trace info.

Index