



---

Vampirtrace 2.0

Installation and User's Guide

---

Pallas GmbH  
Hermülheimer Straße 10  
D-50321 Brühl, Germany  
<http://www.pallas.com>



Vampir was originally developed by the Zentralinstitut für Mathematik at the Forschungszentrum Jülich.

Vampir is a trademark of the Forschungszentrum Jülich.

This product includes software developed by the University of California, Berkeley and its contributors.

# Contents

Contents .....	i
<b>1 Introduction</b> .....	<b>1</b>
1.1 What is Vampirtrace? .....	1
1.2 What is new in Vampirtrace 2.0? .....	1
1.3 Structure of the Vampirtrace Documentation.....	1
<b>2 Availability and Supported Platforms</b> .....	<b>2</b>
<b>3 Installation and User Setup</b> .....	<b>3</b>
3.1 System Requirements .....	3
3.2 Unpacking the Vampirtrace Distribution .....	3
3.3 The Vampirtrace Licensing Scheme .....	4
3.4 User Setup .....	4
3.5 Testing the Installation .....	4
3.6 Getting Help .....	4
<b>4 How to Use Vampirtrace</b> .....	<b>6</b>
4.1 How to Use Vampirtrace.....	6
4.2 Compiling MPI Programs with Vampirtrace.....	6
4.3 Linking MPI Programs with Vampirtrace .....	6
4.4 Running MPI Programs with Vampirtrace .....	7
4.5 Using the Dummy Libraries .....	7
<b>5 User-level Instrumentation with the API</b> .....	<b>8</b>
5.1 The Vampirtrace API.....	8
5.2 API Version Number.....	8
5.3 Enabling/Disabling Trace Collection .....	8
5.4 Definition of Source Locations .....	9
5.5 User-defined Activities.....	9
5.6 Miscellaneous Functions .....	11
5.7 Vampirtrace Instrumentation Examples .....	11
5.8 C Example: Jacobi Iteration.....	12
5.9 Fortran Example: Jacobi Iteration .....	12
<b>6 Configuring Vampirtrace</b> .....	<b>13</b>
6.1 Specifying a Configuration File .....	13
6.2 Configuration File Format .....	13
6.3 How to Use the Filtering Facility .....	15
6.4 Default Configuration.....	16
<b>7 Tips and Tricks</b> .....	<b>17</b>
7.1 Limiting Tracefile Size .....	17
7.2 Managing Vampirtrace API Calls .....	17
<b>8 Troubleshooting</b> .....	<b>18</b>
8.1 Licensing Problems.....	18
8.2 Can't Find the Tracefile .....	18
8.3 User-Defined Activities Don't Work .....	19
8.4 Messages Are not Shown.....	19
<b>Appendix A Compaq Alpha-Servers</b> .....	<b>A-1</b>
A.1 System Requirements .....	A-1
A.2 Installing Vampirtrace.....	A-1
A.3 Compiling MPI Programs with Vampirtrace.....	A-1
A.4 Linking MPI Programs with Vampirtrace .....	A-1
A.5 Running MPI Programs with Vampirtrace .....	A-1
A.6 Restrictions .....	A-2
<b>Appendix B Cray T3E</b> .....	<b>B-1</b>
B.1 System Requirements .....	B-1
B.2 Installing Vampirtrace.....	B-1
B.3 Compiling MPI Programs with Vampirtrace.....	B-1
B.4 Linking MPI Programs with Vampirtrace .....	B-1
B.5 Running MPI Programs Linked with Vampirtrace .....	B-1

B.6	Support for the shmем Library .....	B-2
B.7	Restrictions .....	B-3
<b>Appendix C Fujitsu VPP Series .....</b>		<b>C-1</b>
C.1	System Requirements .....	C-1
C.2	Installing Vampirtrace .....	C-1
C.3	Compiling MPI Programs with Vampirtrace .....	C-1
C.3.1	Tracing Application Subroutines .....	C-1
C.4	Linking MPI Programs with Vampirtrace .....	C-1
C.5	Running MPI Programs with Vampirtrace .....	C-2
C.6	Restrictions .....	C-2
<b>Appendix D HP Servers and Workstations .....</b>		<b>D-1</b>
D.1	System Requirements .....	D-1
D.2	Installing Vampirtrace .....	D-1
D.3	Compiling MPI Programs with Vampirtrace .....	D-1
D.4	Linking MPI Programs with Vampirtrace .....	D-1
D.5	Running MPI Programs with Vampirtrace .....	D-1
D.6	Restrictions .....	D-1
<b>Appendix E Sun Workstations and Servers .....</b>		<b>E-1</b>
E.1	System Requirements .....	E-1
E.2	Installing Vampirtrace .....	E-1
E.3	Compiling MPI Programs with Vampirtrace .....	E-1
E.4	Linking MPI Programs with Vampirtrace .....	E-1
E.5	Running MPI Programs with Vampirtrace .....	E-2
E.6	Restrictions .....	E-2
<b>Appendix F IBM SP and Workstations .....</b>		<b>F-1</b>
F.1	System Requirements .....	F-1
F.2	Installing Vampirtrace .....	F-1
F.3	Compiling MPI Programs with Vampirtrace .....	F-1
F.4	Linking MPI Programs with Vampirtrace .....	F-1
F.5	Running MPI Programs with Vampirtrace .....	F-1
F.5.1	Recording Source-Code Locations .....	F-2
F.6	Restrictions .....	F-2
<b>Appendix G Intel IA32 Systems and Clusters .....</b>		<b>G-1</b>
G.1	System Requirements .....	G-1
G.2	Installing Vampirtrace .....	G-1
G.3	Compiling MPI Programs with Vampirtrace .....	G-1
G.4	Linking MPI Programs with Vampirtrace .....	G-2
G.5	Running MPI Programs with Vampirtrace .....	G-2
G.6	Restrictions .....	G-2
<b>Appendix H NEC SX Series .....</b>		<b>H-1</b>
H.1	System Requirements .....	H-1
H.2	Identifying the Correct Vampirtrace Library .....	H-1
H.3	Installing Vampirtrace .....	H-1
H.4	Compiling MPI Programs with Vampirtrace .....	H-1
H.4.1	Tracing Application Subroutines .....	H-1
H.5	Linking MPI Programs with Vampirtrace .....	H-2
H.6	Running MPI Programs with Vampirtrace .....	H-2
H.7	Restrictions .....	H-2
<b>Appendix I SGI Origin and Workstations .....</b>		<b>I-1</b>
I.1	System Requirements .....	I-1
I.2	Installing Vampirtrace .....	I-1
I.3	Compiling MPI Programs with Vampirtrace .....	I-1
I.4	Linking MPI Programs with Vampirtrace .....	I-1
I.5	Running MPI Programs with Vampirtrace .....	I-1
I.5.1	Recording Source-Code Locations .....	I-2
I.6	Restrictions .....	I-2

# 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 or the Dimemas performance predictor.

It records all calls to the MPI library and all transmitted messages, and allows to define and record arbitrary user defined events. Instrumentation can be switched on or off at runtime, and a powerful filtering mechanism helps to limit the amount of the generated trace data (see section 6.3).

Vampirtrace is an add-on for existing MPI implementations; using it merely requires relinking the application with the Vampirtrace profiling library (see section 4.2). This will enable the tracing of all calls to MPI routines, as well as of 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 4.4).

## 1.2 What is new in Vampirtrace 2.0?

Vampirtrace 2.0 contains a number of principal new features:

- Complete information about the execution of MPI collective operations is recorded
- Complete information about the execution of MPI-I/O operations can be recorded
- Information about the execution of MPI-2 operations can be recorded

The last two features are only available on a subset of platforms. To take advantage of the new trace information, Vampir and Dimemas 2.5 or later should be used for the analysis.

In addition, several new configuration options have been introduced, most notably `MAX-RECORDS` to limit the tracefile size, and functions to flush trace information during the execution of an application have been added.

On some platforms, the `shmem` communications library is supported.

## 1.3 Structure of the Vampirtrace Documentation

This documentation consists of two principal parts: the main body (sections 1 to 9) discusses the platform-independent Vampirtrace functionality and usage, and the appendices contain detailed information about platform-specific functionality, restrictions and use.

To get the complete picture, it is recommended to read the main body and the appendices that correspond to the systems you intend to use Vampirtrace on.

## 2 Availability and Supported Platforms

Vampirtrace is available for a continuously increasing number of platforms and MPI implementations, including

- Vendor-supplied MPI implementations on most SMP and MPP systems
- MPICH version 1.1.2 on Unix workstations and networks thereof
- MPICH version 1.1.2 on some SMP and MPP systems

The range of supported systems and MPI implementations is growing; for up-to-date information refer to the Pallas WWW page <http://www.pallas.com>.

The Vampirtrace binary distribution and a time-limited evaluation license key can be freely downloaded from the Pallas WWW page.

Inquiries about Vampirtrace should be directed to [info@pallas.com](mailto:info@pallas.com) and bug reports sent to [support@pallas.com](mailto:support@pallas.com).

At the time of writing this document, Vampirtrace was available on the following platforms:

Platform	Operating System	Supported MPI	Appendix
Compaq Alpha Servers	Tru64	Compaq MPI	A
Cray T3E	UNICOS/mk	Cray MPT	B
Fujitsu VPP Series	UXV/P	Fujitsu MPI-2	C
HP Servers and Workstations	HP-UX 10, HP-UX 11	HP MPI	D
Sun Servers and Workstations	Solaris 2	MPICH 1.1.2	E
IBM SP and Workstations	AIX 4	PE 2.3	F
Intel IA32 Systems and Clusters	Linux 2, Solaris 2	MPICH 1.1.2	G
NEC SX Series	SUPER-UX 8	NEC MPI	H
SGI Origin and Workstations	Irix 6.5	SGI MPI	I

## 3 Installation and User Setup

### 3.1 System Requirements

Vampirtrace requires MPI to be installed on the target system(s). Depending on the specific platform, either MPICH 1.1.2 or a vendor-supplied MPI system is supported. For details, please refer to “System Requirements” in the appropriate section of the appendix.

### 3.2 Unpacking the Vampirtrace Distribution

The Vampirtrace binary kit is available for download from the Pallas WWW and ftp servers (<http://www.pallas.com> and <ftp://ftp.pallas.com>). It is packaged as a gzipped tar–file containing the Vampirtrace libraries, include files, documentation and examples. On request, Vampirtrace can be obtained on a number of distribution media, including 3.5 inch diskettes and CD-ROMs with ISO9660 filesystem. These distribution media will contain a single, possibly gzipped tar–file. In the following instructions it is assumed that the distribution tar–file has been unzipped and that it is named `/tmp/VT.tar`.

The Vampirtrace directory tree can be installed anywhere in your filesystem(s). Create the intended root directory if it doesn’t already exist, and `cd` into it. It is recommended to use different root directories for Vampir, Dimemas and Vampirtrace. Also, if you want to install Vampirtrace for several platforms into the same filesystem, please use a unique root directory for each platform.

Unpack the distribution file:

```
tar xvfo /tmp/VT.tar
```

This will generate a directory tree like this:

<code>./bin</code>	License checker binary
<code>./doc</code>	This document
<code>./etc</code>	Various support files
<code>./examples</code>	Examples for using the Vampirtrace API
<code>./include</code>	API header files
<code>./lib</code>	Libraries

The shell script `install-Vampirtrace` will perform the installation. Invoke it by typing

```
./install-Vampirtrace
```

in the Vampirtrace root directory and answer the questions issued by the script. You can choose the access permissions for the Vampirtrace files, and on platforms where MPICH is supported, extend the `mpicc` and `mpif77/mpif90` scripts to include support for compiling and linking with Vampirtrace.

For further details please refer to the section “Installing Vampirtrace” in the appendix describing your target platform.

To use Vampirtrace, you need a demo or permanent license key from Pallas. Run the license checker:

```
./bin/VT_lcheck
```

to inquire your system’s license information. If you don’t have a license configured, the tool will print a message like this:

```
Vampirtrace: could not access license file.
Vampirtrace: need license for product VT20, arch XXXX-XXXX,
             hostid 703A8C0, uid D, machine class S.
Vampirtrace: network address 192.168.3.7
Vampirtrace: environment var PAL_ROOT not set.
Vampirtrace: environment var PAL_LICENSEFILE not set.
```

If you're using a workstation network, you must run the license checker on all machines that will run MPI processes, and collect the license checker messages. For larger installations, inquire a network license key for your IP-subnet.

If you don't have a license for Vampirtrace on the current system, contact [support@pallas.com](mailto:support@pallas.com) for a demo or permanent license. Please remember to include the license checker message.

### 3.3 The Vampirtrace Licensing Scheme

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 two environment variables:

<code>PAL_LICENSEFILE</code>	specifies the complete pathname of the license key file. A relative pathname is interpreted starting from the user's home directory.
<code>PAL_ROOT</code>	points to the root of the Vampirtrace installation. The pathname of the license file is assumed to be <code>\$PAL_ROOT/etc/license.dat</code>

If both variables are set, the value of `PAL_LICENSEFILE` will take precedence.



It is usually best to concatenate all license key files for the installed versions of Vampirtrace, Vampir and Dimemas into a single system-wide license file and set `PAL_LICENSEFILE` in the shell initialization to its pathname.

If called without a valid license, or with invalid settings of the above environment variables, Vampirtrace aborts with an error message like that shown in section 3.2 above.

In this case, make sure that the environment variables mentioned above are correctly set. To acquire a demo or permanent license, contact the Pallas user support by sending email to [support@pallas.com](mailto:support@pallas.com).

### 3.4 User Setup

A user must have access to the Vampirtrace directories and the environment variables for the license scheme must be correctly set. It's recommended to include the `./bin` directory in the Vampirtrace tree in the shell search path.

### 3.5 Testing the Installation

To test the installation, try to compile and link the example codes provided in the `./examples` directory tree. Section 5.7 details how to do this. Run each of the examples (both the versions linked with `libVT.a` and `libDT.a`) following the instructions in sections 5.8 and 5.9, and observe whether tracefiles are created. If that is successful, use Vampir and Dimemas to read the respective traces and have a look at the results. If both tools accept the generated traces, your Vampirtrace installation is working.

### 3.6 Getting Help

---

If any problems do occur during installation or use of Vampirtrace, please contact the Pallas support by sending email to [support@pallas.com](mailto:support@pallas.com).

## 4 How to Use Vampirtrace

### 4.1 How to Use Vampirtrace

Using Vampirtrace is straightforward: first, decide whether you want to generate tracefiles for Vampir or Dimemas, and select the appropriate profiling library (`libVT.a` for Vampir, `libDT.a` for Dimemas). Then, 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 or Dimemas, including records of all calls to MPI routines, all point-to-point and collective communication operations and all MP-I/O 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 sections 5 and 5.7) and recompile. This will allow arbitrary user-defined events to be traced; in practice, it is often very useful to record your applications entry and exit to/from subroutines.

The examples in the `./examples` directory show how to instrument C and Fortran code to collect information about application subroutines. On some platforms, subroutine entry and exit can be collected without source-code instrumentation. Please refer to the section “Tracing Application Subroutines” in your platform’s appendix to see what options are available for your system.

The following sections explain how to compile, link and execute MPI applications with Vampirtrace; they are written in general terms and do not include platform dependent information. Please refer to the corresponding sections in your platform’s appendix for additional information.

### 4.2 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++
- `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/Vol/vampirtrace/include`.

On systems supporting MPICH, the extended compile scripts `mpicc`, `mpif77` and `mpif90` can be used with the `-vt` or `-dt` options. These scripts will supply all necessary options to compile and link with Vampirtrace.

### 4.3 Linking MPI Programs with Vampirtrace

The Vampirtrace package contains two different profiling libraries: `libVT.a` produces tracefiles suitable for the Vampir performance analysis tool, while `libDT.a` generates tracefiles for the Dimemas performance predictor. Depending on the intended use of the trace data, your application must be linked with the corresponding library.



The Vampirtrace library (`libVT.a` or `libDT.a`) contains entry points for all MPI routines. They must be linked against your application object files **before** your system’s MPI library. On many systems, this is achieved by specifying `-lVT -lpmi -lmpi` or `-lVT -lmpi`. Depending on the actual MPI library used, additional measures may need to be taken. Please refer to “*Linking MPI Programs with Vampirtrace*” in your platform’s appendix.

With the extended MPICH compile scripts, simply specify `-vt`:

```
mpicc -vt -o test test.o
mpif77 -vt -o ftest ftest.o
```

To link with `libDT.a` for Dimemas traces, specify `-dt` instead of `-vt`.

## 4.4 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:

<code>VT_CONFIG</code>	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 <code>VT_CONFIG_RANK</code> (see below).
<code>VT_CONFIG_RANK</code>	Contains the rank (in <code>MPI_COMM_WORLD</code> ) of the MPI process that reads the Vampirtrace configuration file. The default value is 0. Setting a different 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 resulting tracefile is named `argv[0].bpv` for Vampir tracefiles, and `argv[0].trf` for Dimemas traces, with `argv[0]` being the pathname of the executable image, unless a different name has been specified in the configuration file.

Which MPI process actually writes the tracefile can be influenced by a directive in the configuration file (see section 6.2); by default, it is the same MPI process that reads the configuration file.

## 4.5 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 (and the ordinary MPI library) to create an executable that will not generate traces and not incur the profiling overhead. This library is called `libVTnull.a` and resides in the Vampirtrace library directory.

When using the extended MPICH compile scripts, specifying the `-vtnull` flag will compile and link with the dummy Vampirtrace library.

A separate dummy version of `libDT.a` is not necessary - you can always use `libVTnull.a`.

## 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 application-specific activities. Header files with the necessary parameter, macro and function declarations are provided in the `include` directory: `VT.h` for ANSI C and `VT.inc` for Fortran 77 and Fortran 90. It is strongly recommended to include these headers if any Vampirtrace API routines are to be called.

Vampirtrace is initialized within `MPI_Init()`; don't call any Vampirtrace API routines before `MPI_Init()` or after `MPI_Finalize()`.

The Vampir and Dimemas profiling libraries (`libVT.a` and `libDT.a`) have the same API with the same syntax and semantics, and share the same include files.

### 5.2 API Version Number

The macro `VT_VERSION` (C) and the parameter `VTVERSION` (Fortran 77) define the version of the Vampirtrace API. Value is a 4-digit integer number indicating the release (first two digits) and revision numbers (last two digits), .e.g. 2010 means release 2.0, revision 1.0.

### 5.3 Enabling/Disabling Trace Collection

Vampirtrace starts recording trace data when `MPI_Init()` returns. The data collection can be arbitrarily disabled and re-enabled at runtime by calling the following two functions:

```
void VT_traceoff (void)
SUBROUTINE VTTRACEOFF ()
```

Disables the collection of trace data for the local process.

```
void VT_traceon (void)
SUBROUTINE VTTRACEON ()
```

Re-enables the collection of trace data for the local process.

Please note that these functions affect the calling process only. To switch on/off profiling globally, each MPI process must call the respective API routine.

## 5.4 Definition of Source Locations

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

- State definitions, associating a source location with a state. This will f.i. record the location of a routine in the source-code.
- State changes, associating a source location with the state change. This will record f.i. where a routine is called.
- Communication events, associate 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 codes that must in turn be defined by calls to the `VT_locdef()` routine *before* they are used.

Locations are defined as strings. They either contain a filename and a line number, or are treated as opaque strings, if the `Location-asis` configuration option is set to `on`. The location string may contain at most `VT_STRLLEN` characters, longer location strings are silently truncated.

The following API routines deal with locations:

```
int VT_locdef (int code, char *location)
SUBROUTINE VTLOCDEF (CODE, LOCATION, IERR)
```

Associates the given code with the given location. The codes are positive integers not greater than `VT_MAX_USERCODE` and they must be unique amongst all processes. A return code is passed as the result or in the `IERR` argument with a negative value indicating an error.

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 lcode)
SUBROUTINE VTTHISL (LCODE, IERR)
```

Supplies a location code for the next call to an MPI routine. Please note that `VT_locdef()` must be used to associate the code to an actual location string. The given location code is buffered internally, and attached to the next call to an MPI routine. A return code is passed as the result or in the `IERR` argument with a negative value indicating an error.

## 5.5 User-defined Activities

Vampir can display and analyze general (possibly nested) state changes, relating f.i. to subroutine calls, code blocks and other activities occurring in a process. Vampir has a two-level model of states: a state is referred to by a tuple of names: an *activity* name that identifies a group of states, and the state (or *symbol*) name that in turn identifies a particular state in that group. For instance, all MPI routines are arranged in the activity `MPI`, and are identified by the generic routine name. User-defined states can be arbitrarily arranged into activities.

The Vampirtrace API allows to define user-defined states, and to enter and leave them. In order to reduce the instrumentation overhead, the enter and leave routines identify states by an integer code that is attached to the activity/symbol pair by calling a definition routine first. While all activity definitions are process local, the integer codes must be assigned in a globally consistent manner: if the same code is used for two different activity/symbol pairs in two different processes, one definition overrides the other. Also, codes must be unique within each process.

Optionally, a *source location* can be recorded in a Vampirtrace tracefile. There are two different approaches: one is to define a source location for user-defined state, and the other is to define source code locations for the begin and end events. Please refer to section 5.4 for information on how to define source locations.

To attach a *source location* to a user-defined state, use the `VT_symdefl()` routine and supply a location string containing the name of the source file and the line number separated by a colon, e.g. `interface.c:120`.

To define source code locations for the begin and end events, the integer location code as passed to the `VT_locdef()` routine must be passed to the `VT_beginl()` and `VT_endl()` routines.

The following API routines allow to specify user-defined states:

```
int VT_symdef (int code, char *state, char *activity)
SUBROUTINE VTSYMDEF (CODE, STATE, ACTIVITY, IERR)
int VT_symdefl (int code, char *state, char *activity, char *location)
SUBROUTINE VTSYMDEFL (ICODE, STATE, ACTIVITY, LOCATION, IERR)
```

Defines a numeric code for a user-defined state. The code, the state (symbol) name and the activity name are given as input; a negative value is returned (as function value in C, in the last `IERR` argument in Fortran) to signal an error - such as duplicate definitions for the same code. The `VT_symdefl` and `VTSYMDEFL` routines accept a location code that defines a source location for the state.



The functions are *process-local*; take care to define symbols consistently over all processes!

To enter and leave user-defined states, use

```
int VT_begin (int code)
SUBROUTINE VTBEGIN (ICODE, IERR)
int VT_beginl (int code, int lcode)
SUBROUTINE VTBEGINL (ICODE, LCODE, IERR)
```

Enter the user-defined state specified by `code` or `ICODE`; it must, of course, refer to an activity previously defined by `VT_symdef()` or equivalent. In case of an error, a negative value is returned (as function value in C, in `IERR` in Fortran).

```
int VT_end (int code)
SUBROUTINE VTEND (ICODE, IERR)
int VT_endl (int code, int lcode)
SUBROUTINE VTENDL (ICODE, LCODE, IERR)
```

Leave the user-defined state specified by `code` or `ICODE`; it must, of course, refer to an activity previously defined by `VT_symdef()` or equivalent. Furthermore, the calling process state must have entered the state before and must not have left it already. In case of an error, a negative value is returned (as function value in C, in `IERR` in Fortran).

State codes have to be supplied by the application programmer and must be positive integers not exceeding `VT_MAX_USERCODE` (or `VIMAXCODE` in Fortran). The symbol and activity parameters are character strings which must be no longer than `VT_STRLEN` (or `VTSLEN` in Fortran). Strings longer than that limit will be silently truncated. The limits are defined as macros or parameters in the C and Fortran header files.

## 5.6 Miscellaneous Functions

Vampirtrace limits intrusion into the application run by minimizing the instrumentation overhead. All collected trace data is kept locally in each processor's memory. It is post-processed and saved to disk when the application is about to finish in `MPI_Finalize()`. This scheme can sometimes cause problems, though: memory available to the application is reduced, and no traces are written if an application dies or deadlocks before MPI is finalized. To help in these situations, a new API call is provided that flushes local trace data into a file:

```
int VT_flush(void)
SUBROUTINE VTFLUSH(IERR)
```

Dumps all collected trace records to a temporary file and frees the memory allocated by Vampirtrace. The collection of trace data continues after the routine returns. The function is local to a process – each process can individually dump its trace records. In `MPI_Finalize()`, the temporary files are merged with any in-memory trace data and a normal tracefile is written.

The temporary files are named `<FLUSH-PREFIX>/VT-flush<rank>-<pid>.dat`, where `<FLUSH-PREFIX>` is `"/tmp"` by default, `<rank>` is the rank in `MPI_COMM_WORLD` and `<pid>` is the UNIX process ID. Vampir will visualize the flush event with a special predefined activity/state `"VT_FLUSH"`.

In case of an error, a negative value is returned as function value in C or in `IERR` in Fortran.

Please refer to section 6.2 to learn about the `MAX-RECORDS` configuration option that limits the Vampirtrace memory usage.

## 5.7 Vampirtrace Instrumentation Examples

The `./examples` directory contains a collection of example programs that have been instrumented with the Vampirtrace API. Currently, a Jacobi solver example is provided in a C version (in subdirectory `./jacobi-C`) and in a Fortran 77 version (`./jacobi-Fortran`). Complete information about the particular examples can be found in the `README`-files in the subdirectories.

To compile and link the examples with the supplied Makefiles, you need gnumake (version 3.7 or later, which is available from your friendly GNU archive; as a last resort, try <ftp://prep.ai.mit.edu:/pub/gnu>). Please set the `PAL_ROOT` environment variable to indicate the root directory of your Vampirtrace installation. To make compiling and linking easy, the `palmake` script is provided. It is invoked as

```
./palmake <target>
```

and knows about the following targets:

vtjacobic	C Jacobi program linked with libVT.a
vtjacobif	Fortran Jacobi program linked with libVT.a
dtjacobic	C Jacobi program linked with libDT.a
dtjacobif	Fortran Jacobi program linked with libDT.a

For additional information about the examples, and instructions on how to compile and start them, refer to the following sections and to the `README`-file.

## 5.8 C Example: Jacobi Iteration

This code was adapted from the “Jacobi Iteration - Overlapping communication (sends first)” exercise of the HTML MPI tutorial by William Gropp and Ewing Lusk that is available as <ftp://www.mcs.anl.gov/pub/mpi/mpiexmp.tar.gz>.

For the instrumentation, a setup routine `VT_setup` has been created in the file `VT.c`, and calls to `VT_begin` and `VT_end` have been added to each function of the original program. All the instrumentation code is enclosed in `#ifdef/#endif` brackets and will only be compiled if the macro `USE_VT` is defined.

Start the executable (`vtjacobic` or `dtjacobic`) following the conventions of your system and MPI implementation. It accepts the following command-line arguments:

<code>-print</code>	switch on debugging output
<code>-maxit &lt;iter&gt;</code>	set maximum iteration count
<code>-n &lt;rows&gt;</code>	set number of grid rows to (default 128)
<code>-m &lt;cols&gt;</code>	set number of grid columns (default 128)

The ratio of computation to communication can be influenced by varying the values of `<rows>` and `<cols>` (and thus the grid-size and grid-layout).

## 5.9 Fortran Example: Jacobi Iteration

This code was adapted from the examples for “Using MPI” by William Gropp, Ewing Lusk, and Anthony Skjellum, published by MIT Press (ISBN 0-262-57104-8).

The original version (with the other examples from this fine book) can be retrieved as <ftp://www.mcs.anl.gov/pub/mpi/using/examples.tar>.

To instrument, a setup routine `VTSETUP` has been created in the file `VT.f`, and calls to `VTBEGIN` and `VTEND` have been added to each subprogram of the original program. All the instrumentation code is enclosed in `#ifdef/#endif` brackets and will only be compiled if the macro `USE_VT` is defined.

Start the executable (`vtjacobif` or `dtjacobif`) following the conventions of your system and MPI implementation. It accepts no command-line arguments.

## 6 Configuring Vampirtrace

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

### 6.1 Specifying a Configuration File

As mentioned above in section 4.3, 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!

### 6.2 Configuration File Format

The configuration file is a plain ASCII file containing a number of directives, one per line; any lines starting with the `#` character are being ignored. Within a line, whitespaces separate fields, and the `"` character must be used to quote fields containing whitespaces.

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 `|`, with default values indicated by **fat** type.

The following directives are currently supported:

<code>LOGFILE-NAME &lt;logfile&gt;</code>	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 current working directory of the process writing it.
<code>LOGFILE-RANK &lt;rank&gt;</code>	Determines which process creates and writes the tracefile in <code>MPI_Finalize</code> . Default value is the process reading the configuration file, or the process with rank 0 in <code>MPI_COMM_WORLD</code> .
<code>LOGFILE-PREFIX &lt;directory name&gt;</code>	Specifies the directory of the tracefile. It can be an absolute or relative pathname; in the latter case, it is interpreted relative to the current working directory of the process writing the trace data.
<code>INTERNAL-MSGS [on off]</code>	Specifies whether communication internal to the MPI library should be traced. By specifying <code>on</code> , point-to-point communication that may be used by the collective communication functions will be traced. By default, internal events are not traced. This option does not work with most MPI implementations!
<code>LOCATIONASIS [on off]</code>	Specifies if location strings are handled as opaque objects ( <code>on</code> ) or interpreted as a combination of filename and line number like <code>"init.f:4"</code> ( <code>off</code> ).

AUTOFLUSH [**on**|off]

This parameter is used to enable (**on**) or disable (**off**) the automatic flushing of in-memory trace data to a file when the `MAX-RECORDS` have been collected or the process' memory is exhausted. If set to **on** (default), Vampirtrace will dump all in-memory trace records to a temporary file, free the memory used by Vampirtrace, but continue to collect trace records. The temporary files are named `<FLUSH-PREFIX>/VT-flush<rank>-<pid>.dat`, where `<rank>` is the rank in `MPI_COMM_WORLD` and `<pid>` is the UNIX process ID.

FLUSH-PREFIX <directory name>

Specifies the directory for temporary files. It is best to specify a process-local filesystem. The default value is `/tmp`.

MAX-RECORDS <int value>

Specifies the number of records collected by Vampirtrace in memory before they are being flushed (if `AUTOFLUSH` is **on**), or trace data collection is stopped. By default, an unlimited number of records is collected in memory. This option can be used to reduce Vampirtrace's memory requirements.

PCTRACE [**on**|**off**]

Some platforms support the automatic stack sampling for MPI calls and user-defined events. If enabled, Vampirtrace will sample the stack and record the source location on these platforms.

TRACERANKS <triplets>

Specifies for which processes tracing is to be enabled. This option accepts a comma-separated list of triplets, each one of the form `<start>:<stop>:<incr>` specifying the minimum rank the maximum rank, and the increment to determine a set of processes (similar to Fortran 90 notation). Ranks are interpreted relative to `MPI_COMM_WORLD`. To enable tracing only on odd process ranks, specify the triplet is: "1:N:2", where `N` is the total number of processes.

ACTIVITY <act\_pattern> [**on**|off]

Defines a filter for a given activity class. Two arguments must be provided: a pattern, and **on** or **off**. Patterns are ordinary strings and may contain the wild-card character `*` that matches any number of characters in activity class names. Specifying **on** as second argument enables the tracing of all activities that match the class pattern; **off** likewise prevents tracing of all matching activities.

```
SYMBOL <sym_name> [on|off]
```

Defines a filter for a given symbol name. Two arguments must be provided: a pattern, and `on` or `off`. Patterns are ordinary strings and may contain the wild-card character `*` that matches any number of characters in activity symbol names.

Specifying `on` as second argument enables the tracing of all activities that match the symbol pattern; `off` likewise prevents tracing of all matching activities.

### 6.3 How to Use the Filtering Facility

A single configuration file can continue an arbitrary number of filter directives that are evaluated whenever an activity 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 allows to easily 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
SYMBOL MPI_wait* ON
# enable all activities in the Application class
ACTIVITY Application ON
```

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 `TRACERANKS`. 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 enabling 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.

## 6.4 Default Configuration

If no configuration file has been specified, Vampirtrace runs with the following default setup:

```
LOGFILE-RANK 0
LOGFILE-PREFIX "."
LOGFILE-NAME argv[0].bpv or argv[0].trf
LOCATIONANALYSIS ON
MAX-RECORDS 0
AUTOFLUSH ON
FLUSH-PREFIX "/tmp"
TRACERANKS 0:N:1
PCTRACE OFF
INTERNAL-MSGS OFF
ACTIVITY * ON
```

In effect, all activities in all processes are traced, internal messages are ignored, and the tracefile is written by the process with rank 0 in `MPI_COMM_WORLD`. All trace records are being kept in memory until MPI is finalized.

## 7 Tips and Tricks

### 7.1 Limiting Tracefile Size

Although Vampirtrace uses a compact binary format to store the trace data, tracefile sizes for real-world applications can get immense. To control this, there are basically three options:

- Limit the number of events to be logged by scaling down the application, like f.i., iteration count, number of processes, problem size etc. Quite often, this is not acceptable because reduced input datasets are not available or performance analysis for reduced problems is simply not interesting.
- 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 inconvenient at least.
- Use the `MAX-RECORDS` configuration option to set a limit on the number of trace records. On average, one trace record consumes 40 Bytes, so setting `MAX-RECORDS` to 100000 will limit the amount of trace data to about 4 Mbytes. Vampirtrace will then only record the first 100000 records. It is important to also specify `AUTOFLUSH OFF!`
- Use the 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.

### 7.2 Managing Vampirtrace API Calls

The API routines greatly extend the functionality of Vampirtrace. Alas, 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 to use the C pre-processor (or an equivalent tool for Fortran) to guard all the calls to the Vampirtrace API by `#ifdef` directives. This will allow to easily generate a plain vanilla version and an instrumented version of a program.

## 8 Troubleshooting

The Vampirtrace library must deal with 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 (see section 4.3), 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 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.

To avoid a memory error, try to limit the amount of trace data as explained in section 7.1. The memory requirements of Vampirtrace can be reduced with the `MAX-RECORDS` option. Each record uses about 40 Bytes on average; by setting `MAX-RECORDS` to a value of 25000, Vampirtrace will only use up about a Megabyte of memory. The `AUTOFLUSH` option needs to be enabled if you want to see the whole application execution.

### 8.1 Licensing Problems

As explained in the section about licensing (3.3), Vampirtrace needs a valid license to operate properly. If no license is found, an error message like the one in section 3.2 will be printed, and Vampirtrace terminates the application. Check that the `PAL_ROOT` or `PAL_LICENSEFILE` environment variables are set properly: the same MPI process that reads the configuration file (usually the one with rank 0 in `MPI_COMM_WORLD`) will also try to read the license file. If the problem persists, contact [support@pallas.com](mailto:support@pallas.com); please to include the contents of your license file and the complete text of the license error message.

### 8.2 Can't Find the Tracefile

Unless told otherwise in the configuration file, Vampirtrace will write the trace data to the file `argv[0].bpv`, 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 meddle 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>.bpv`, 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 (see section 4.3) or by a directive in the configuration file (see section 6).

### 8.3 User-Defined Activities Don't Work

In order to minimise 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 are mislabelled in Vampir displays.

### 8.4 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 or received at the wrong place. Similarly, filtering out some of the above mentioned MPI routines has the same effect.



## Appendix A Compaq Alpha-Servers

### A.1 System Requirements

Vampirtrace works on Compaq Alpha systems running Tru64 (alias Digital Unix 5). or compatible. It requires the Compaq MPI version 170 or compatible.

### A.2 Installing Vampirtrace

The shell script `install-Vampirtrace` will perform the installation. First, `cd` into the Vampirtrace root directory, and start the installation by typing

```
./install-Vampirtrace
```

Then answer the questions issued by the script. You can choose the access permissions for the Vampirtrace files.

### A.3 Compiling MPI Programs with Vampirtrace

For programs with calls to Vampirtrace API routines, the path to the Vampirtrace include directory must be supplied to the compiler: use the `-I` flag for the C compiler and the Fortran compilers that support it, e.g.:

```
cc -I/Vol/Vampirtrace/include -I/Vol/mpi/include -c test.c
f77 -I/Vol/VAMPIRtrace/include -I/Vol/mpi/include -c fttest.f
f90 -I/Vol/VAMPIRtrace/include -I/Vol/mpi/include -c fttest.f
```

### A.4 Linking MPI Programs with Vampirtrace

The Vampirtrace package contains two different profiling libraries: `libVT.a` produces tracefiles suitable for the Vampir performance analysis tool, and `libDT.a` generates tracefiles for the Dimemas performance predictor. Depending on the intended use of the trace data, your application must be linked with the correct library.

The following discussion refers to `libVT.a`; the Dimemas trace library is linked in exactly the same way – just replace `libVT.a` by `libDT.a`.

To link an executable that will generate traces, include both the Vampirtrace library `libVT.a` and the profiling library `libpmpi.a` in the link command line, e.g.:

```
cc -o test test.o -L/Vol/VAMPIRtrace/lib -lVT -lpmpi -lmpi -lm
```

Fortran applications need the Fortran wrapper library `libfmpi.a` linked in *before* `libVT.a`:

```
f77 -o fttest fttest.o -L/Vol/VAMPIRtrace/lib -lfmpi -lVT -lpmpi \
-lmpi -lm
f90 -o fttest fttest.o -L/Vol/VAMPIRtrace/lib -lfmpi -lVT -lpmpi \
-lmpi -lm
```

Be sure to obey the order of the various libraries: `libVT.a` must be linked *before* the MPICH libraries, and *after* and the Fortran wrapper library.

To run applications that have been instrumented with calls to the Vampirtrace API without generating a trace, a dummy library `libVTnull.a` is provided. It contains just the Vampirtrace API entry points, and can be linked after the MPI libraries. The extended compile scripts support the `-vtnull` option to link with the dummy library.

### A.5 Running MPI Programs with Vampirtrace

To run an application linked with Vampirtrace, you can use the `dmpi` launcher as with an “ordinary” MPI application.

## A.6 Restrictions

Vampirtrace cannot display the internal point-to-point communication within the MPI implementations.

Vampirtrace is not multi-thread safe.

To get satisfactory tracefiles, the micro-second clock kernel option must be enabled.

## Appendix B Cray T3E

### B.1 System Requirements

Vampirtrace works on Cray T3E systems running UNICOS/mk 2.0.4 or compatible. It requires Message Passing Toolkit 1.2/1.3 or compatible.

### B.2 Installing Vampirtrace

The shell script `install-Vampirtrace` will perform the installation. First, `cd` into the Vampirtrace root directory, and start the installation by typing

```
./install-Vampirtrace
```

Then answer the questions issued by the script. You can choose the access permissions for the Vampirtrace files.

### B.3 Compiling MPI Programs with Vampirtrace

Programs that do not contain calls to the Vampirtrace API can be compiled with the `cc` and `f90` commands, as usual.

For programs with calls to Vampirtrace API routines, the path to the Vampirtrace include directory must be supplied to the compiler with the `-I` flag, e.g.:

```
cc -I/Vol/VAMPIRtrace/include -c test.c
f90 -I/Vol/VAMPIRtrace/include -c ftest.f
```

### B.4 Linking MPI Programs with Vampirtrace

The Vampirtrace package contains two different profiling libraries: `libVT.a` produces tracefiles suitable for the Vampir performance analysis tool, while `libDT.a` generates tracefiles for the Dimemas performance predictor. Depending on the intended use of the trace data, your application must be linked with the correct library.

The following discussion refers to `libVT.a`; the Dimemas trace library is linked in exactly the same way - just replace `libVT.a` by `libDT.a`.

To link an executable that will generate traces from a C or C++ program, include a reference to the Vampirtrace library `libVT.a` in the link command line, e.g.:

```
cc -o test test.o -L/Vol/VAMPIRtrace/lib -lVT -lpmi -lmpi
```

Fortran applications are linked in the same way:

```
f90 -o ftest ftest.o -L/Vol/VAMPIRtrace/lib -lVT -lpmi -lmpi
```

Be sure to put the Vampirtrace library *before* the MPI library in the link command line, and `-lpmi` *before* `-lmpi`. Otherwise, the application will not generate a tracefile.

### B.5 Running MPI Programs Linked with Vampirtrace

To run an application linked with Vampirtrace, you can use any of the methods working for ordinary MPI programs. In particular, both `mpirun` and `mpprun` are known to work.

## B.6 Support for the shmем Library

With the optional SHMEM feature enabled, Vampirtrace can trace the following shmем communication routines:

Routine	Kind	C	Fortran
shmем_my_pe	local	●	●
shmем_n_pes	local	●	●
shmем_quiet	local	●	●
shmем_wait	local	●	●
shmем_get	point-to-point	●	●
SHMEM_GET4	point-to-point		●
SHMEM_GET8	point-to-point		●
shmем_get32	point-to-point	●	
shmем_get64	point-to-point	●	
shmем_iget	point-to-point	●	●
shmем_iput	point-to-point	●	●
shmем_put	point-to-point	●	●
SHMEM_PUT4	point-to-point		●
SHMEM_PUT8	point-to-point		●
shmем_put32	point-to-point	●	
shmем_put64	point-to-point	●	
shmем_swap	point-to-point	●	●
shmем_short_fadd	point-to-point	●	
SHMEM_INT4_FADD	point-to-point		●
shmем_barrier	global	●	●
shmем_barrier_all	global	●	●
shmем_broadcast	global	●	●
SHMEM_BROADCAST4	global		●
shmем_broadcast32	global	●	
shmем_broadcast64	global	●	
SHMEM_BROADCAST8	global		●
SHMEM_INT4_MAX_TO_ALL	global		●
SHMEM_INT4_MIN_TO_ALL	global		●
SHMEM_INT4_SUM_TO_ALL	global		●
SHMEM_INT8_MAX_TO_ALL	global		●
SHMEM_INT8_MIN_TO_ALL	global		●
SHMEM_INT8_SUM_TO_ALL	global		●
SHMEM_REAL4_MAX_TO_ALL	global		●
SHMEM_REAL4_MIN_TO_ALL	global		●
SHMEM_REAL4_SUM_TO_ALL	global		●
SHMEM_REAL8_MAX_TO_ALL	global		●
SHMEM_REAL8_MIN_TO_ALL	global		●
SHMEM_REAL8_SUM_TO_ALL	global		●
shmем_int_max_to_all	global	●	
shmем_int_min_to_all	global	●	
shmем_int_sum_to_all	global	●	
shmем_short_max_to_all	global	●	
shmем_short_min_to_all	global	●	
shmем_short_sum_to_all	global	●	
shmем_float_max_to_all	global	●	
shmем_float_min_to_all	global	●	
shmем_float_sum_to_all	global	●	
shmем_double_max_to_all	global	●	
shmем_double_min_to_all	global	●	
shmем_double_sum_to_all	global	●	

For the local routines, entry and exit into/from the routine is recorded. For the point-to-point routines, the data transfer is recorded in addition. For the global routines, the equivalent MPI-style global communication operation is recorded. It is highly recommended to use Vampir 2.5 to properly display the global routine information. All shmem routines are grouped in the `SHMEM` activity.

A number of changes have to be made to shmem-applications:

- At the beginning of the application, a call to `MPI_Init` must be inserted.
- At the end of the application, a call to `MPI_Finalize` must be inserted.
- The shorthand names (like `BARRIER`) must be replaced by the long ones (like `shmem_barrier_all`).
- For C applications, the CPP definition `#define SHMEM_MACRO_OPT 0` should be inserted before the header file `mpp/shmem.h` is included to inhibit the use of macros. This can be achieved by including `VT.h` *after* `shmem.h`.

With these changes, the application code can be compiled as described in section B.3, linked as shown in section B.4 and run as shown in section B.5.

## B.7 Restrictions

Vampirtrace cannot trace internal messages used by the MPT implementation.



## Appendix C Fujitsu VPP Series

### C.1 System Requirements

Vampirtrace works on the Fujitsu VPP300, VPP700 and VX systems running UXP/V V10L20 or compatible. It requires the Fujitsu MPI-2, version V11L10 with the L98101 update or later. It will *not* work with the old Fujitsu MPI-1 system.

Vampirtrace supports Fujitsu's scalar and vector C and Fortran compilers.

If the `MPI2` feature has been installed, Vampirtrace records all MPI-2 activity in addition to the MPI-1 calls. Also, MPI-I/O activity will be recorded and can be analyzed with Vampir 2.5 or later.

### C.2 Installing Vampirtrace

The shell script `install-Vampirtrace` will perform the installation. First, `cd` into the Vampirtrace root directory, and start the installation by typing

```
./install-Vampirtrace
```

Then answer the questions issued by the script. You can choose the access permissions for the Vampirtrace files.

### C.3 Compiling MPI Programs with Vampirtrace

Programs that do not contain calls to the Vampirtrace API can be compiled with the `cc`, `vcc` and `frt` compilers, as usual.

For programs with calls to Vampirtrace API routines, the path to the Vampirtrace include directory must be supplied to the compiler with the `-I` flag, e.g.:

```
vcc -K ARG,a4 -I/Vol/VAMPIRtrace/include -c test.c
frt -I/Vol/VAMPIRtrace/include -Aab -c fttest.f
```

For C programs, please be sure to specify 4-byte alignment (with the `-K a4` flag), and to put varargs function arguments both on the stack and in registers (with the `-K ARG` flag). For Fortran, specify the `-Aab` flag to ensure correct alignment of `COMMON` data and to refer to dummy arguments by position.

#### C.3.1 Tracing Application Subroutines

For Fortran applications, all calls to application subroutines can be automatically traced without prior instrumentation of the source-code. To enable subroutine tracing, compile with the `-Dl` flag:

```
frt -I/Vol/VAMPIRtrace/include -Aab -Dl -c fttest.f
```

Object modules generated with the `-Dl` option can be linked as usual.

To actually record the subroutine calls in an application run, be sure to enable the `PCTRACE` configuration option (see section 6.2).

### C.4 Linking MPI Programs with Vampirtrace

The Vampirtrace package contains two different profiling libraries: `libVT.a` produces tracefiles suitable for the Vampir performance analysis tool, and `libDT.a` generates tracefiles for the Dimemas performance predictor. Depending on the intended use of the trace data, your application must be linked with the correct library.

The following discussion refers to `libVT.a`; the Dimemas trace library is linked in exactly the same way – just replace `libVT.a` by `libDT.a`.

To link an executable that will generate traces, include both the Vampirtrace library `libVT.a` and the profiling library `libpmpi.a` in the link command line, e.g.:

```
vcc -Wl,-J,-P,-t,-dy -o test test.o -L/Vol/Vampirtrace/lib \  
-lVT -lpmpi -lmpi -lmp -lpx -lself -lgen -lsocket -lnsl -lm
```

Fortran applications need the Fortran wrapper library `libfmpi.a` linked in *before* `libVT.a`:

```
frc -Wl,-J,-P,-t,-dy -o ftest ftest.o -L/Vol/Vampirtrace/lib \  
-lfmpi -lVT -lpmpi -lmpi -lmp -lpx -lself -lgen -lsocket \  
-lnsl -lcvp -lm
```

Be sure to obey the order of the various libraries: `libVT.a` must be linked *before* the MPI libraries, and *after* and the Fortran wrapper library.

To run applications that have been instrumented with calls to the Vampirtrace API without generating a trace, a dummy library `libVTnull.a` is provided. It contains just the Vampirtrace API entry points, and can be linked after the MPI libraries.

## C.5 Running MPI Programs with Vampirtrace

Programs linked against the Vampirtrace libraries are started in the same way as “ordinary” MPI-2 programs with the `mpiexec` launcher. If your MPI-2 version supports launching applications in „limited mode“, this will also work with instrumented applications.

## C.6 Restrictions

Vampirtrace cannot display the internal point-to-point communication within Fujitsu’s MPI-2 implementation.

## Appendix D HP Servers and Workstations

### D.1 System Requirements

Vampirtrace runs on HP servers and workstations running HP-UX 10.20 or HP-UX 11. It requires HP MPI version 1.5 or compatible.

### D.2 Installing Vampirtrace

The shell script `install-Vampirtrace` will perform the installation. First, `cd` into the Vampirtrace root directory, and start the installation by typing

```
./install-Vampirtrace
```

Then answer the questions issued by the script. You can choose the access permissions for the Vampirtrace files.

### D.3 Compiling MPI Programs with Vampirtrace

Programs that do not contain calls to the Vampirtrace API can be compiled as usual.

For programs with calls to Vampirtrace API routines, the path to the Vampirtrace include directory must be supplied to the compiler with the `-I` flag. It is recommended to use the HP-supplied `mpicc`, `mpif77` and `mpif90` compilation scripts., e.g.:

```
mpicc -I/Vol/VAMPIRtrace/include -c test.c
mpif77 -I/Vol/VAMPIRtrace/include -c ftest.f
mpif90 -I/Vol/VAMPIRtrace/include -c ftest.f
```

### D.4 Linking MPI Programs with Vampirtrace

The Vampirtrace package contains two different profiling libraries: `libVT.a` produces tracefiles suitable for the Vampir performance analysis tool, while `libDT.a` generates tracefiles for the Dimemas performance predictor. Depending on the intended use of the trace data, your application must be linked with the correct library.

The following discussion refers to `libVT.a`; the Dimemas trace library is linked in exactly the same way - just replace `libVT.a` by `libDT.a`.

To link an executable that will generate traces from a C or C++ program, include a reference to the Vampirtrace library `libVT.a` in the link command line, e.g.:

```
mpicc -o test test.o -Wl,-L/Vol/VAMPIRtrace/lib -lVT -lm
```

Fortran applications are linked in the same way:

```
mpif90 -o ftest ftest.o -Wl,-L/Vol/VAMPIRtrace/lib -lVT
```

### D.5 Running MPI Programs with Vampirtrace

Applications linked with Vampirtrace can be started in the same way as ordinary MPI applications, e.g. by using the `mpirun` launcher.

### D.6 Restrictions

Vampirtrace cannot display the internal point-to-point messages sent by the HP MPI implementation.



Vampirtrace is not multi-thread safe.

## Appendix E Sun Workstations and Servers

### E.1 System Requirements

Vampirtrace runs on Sun Sparc systems running Solaris 2.6 or higher. It requires MPICH version 1.1.2 or compatible. The Vampirtrace libraries do not depend on a particular MPICH communication device: shared-memory, ch\_p4 and special customized devices will work. Currently, only the 32-bit mode of Solaris is supported.

Vampirtrace works with the C, C++ and Fortran compilers from Sun, and is compatible to the Fujitsu Fortran 90 compiler. The GNU compilers are untested, but should work, also.

The MPICH distribution can be ftp'ed from <ftp://www.mcs.anl.gov/pub/mpi/mpich.tar.gz>.

### E.2 Installing Vampirtrace

The shell script `install-Vampirtrace` will perform the installation. First, `cd` into the vampirtrace root directory, and start the installation by typing

```
./install-Vampirtrace
```

Then answer the questions issued by the script. In particular, you can choose the file access permissions, and extend the MPICH compile scripts `mpicc`, `mpif77` and `mpif90` for use with Vampirtrace.

The install script will optionally patch the MPICH compile and link scripts `mpicc`, `mpif77` and `mpif90` to include support for compiling and linking with Vampirtrace. New options (`-vt`, `-dt` and `-vtnull`) will be available that add include paths and/or libraries to the compile and link command lines. The scripts are installed into the `bin` directory, but you're free to move them anywhere afterwards; it's advisable to either replace the original scripts with them or give them alternate names, like f.i. `mpicc_VT`.

### E.3 Compiling MPI Programs with Vampirtrace

Programs that do not contain calls to the Vampirtrace API can be compiled as usual with MPICH, either with the `mpicc` and `mpif77` scripts or directly with your platform's compilers.

For programs with calls to Vampirtrace API routines, the path to the Vampirtrace include directory must be supplied to the compiler: use the `-I` flag for the C compiler and the Fortran compilers that support it, e.g.:

```
cc -I/Vol/Vampirtrace/include -I/Vol/mpi/include -c test.c
f77 -I/Vol/VAMPIRtrace/include -I/Vol/mpi/include -c ftest.f
```

When using the extended MPICH compile scripts, just specify the `-vt` or `-dt` options:

```
mpicc -vt -c test.c
mpif77 -dt -c ftest.f
mpif90 -dt -c ftest.f
```

The `mpif77` and `mpif90` scripts may create symbolic links `mpif.h` or `VT.inc` referring to the MPI and Vampirtrace include files. These links can easily become outdated or dangling; check that they refer to the correct files if you're having problems compiling or linking Fortran MPI applications. It's best to delete the links after compilation.

### E.4 Linking MPI Programs with Vampirtrace

The Vampirtrace package contains two different profiling libraries: `libVT.a` produces tracefiles suitable for the Vampir performance analysis tool, and `libDT.a` generates tracefiles for the Dimemas performance predictor. Depending on the intended use of the trace data, your application must be linked with the correct library.

The following discussion refers to `libVT.a`; the Dimemas trace library is linked in exactly the same way – just replace `libVT.a` by `libDT.a`.

To link an executable that will generate traces, include both the Vampirtrace library `libVT.a` and the profiling library `libmpich.a` in the link command line, e.g.:

```
cc -o test test.o -L /Vol/mpi/lib/solaris/ch_p4 \  
-L/Vol/VAMPIRtrace/lib -lVT -lmpich -lsocket -lnsl -lm
```

C++ applications need the library `libmpi++.a` in addition:

```
CC -o test test.o -L /Vol/mpi/lib/solaris/ch_p4 \  
-L/Vol/VAMPIRtrace/lib -lmpi++ -lVT -lmpich -lsocket \  
-lnsl -lm
```

Fortran applications need the Fortran wrapper library (usually `libfmpich.a` or `libfmpi.a`) linked in *before* `libVT.a`:

```
f77 -o ftest ftest.o -L /Vol/mpi/lib/solaris/ch_p4 \  
-L/Vol/VAMPIRtrace/lib -lfmpich -lVT -lmpich -lsocket \  
-lnsl -lm
```

Be sure to obey the order of the various libraries: `libVT.a` must be linked *before* the MPICH libraries, and *after* and the Fortran wrapper library. Depending on which device your MPICH installation uses, there will be different system libraries to be linked.

When using the extended compile scripts, simply specify `-vt`:

```
mpicc -vt -o test test.o  
mpif77 -vt -o ftest ftest.o  
mpif90 -vt -o ftest ftest.o
```

To link with `libDT.a` for Dimemas traces, specify `-dt` instead of `-vt`.

To run applications that have been instrumented with calls to the Vampirtrace API without generating a trace, a dummy library `libVTnull.a` is provided. It contains just the Vampirtrace API entry points, and can be linked after the MPI libraries. The extended compile scripts support the `-vtnull` option to link with the dummy library.

## E.5 Running MPI Programs with Vampirtrace

Programs linked against the Vampirtrace libraries are started in the same way as “ordinary” MPICH programs with the `mpirun` or `mpiexec` launchers. There are no Vampirtrace runtime flags, but the library will examine the environment variables `VT_CONFIG` and `VT_CONFIG_RANK` that specify a Vampirtrace configuration file (see section 6.1), and of course `PAL_ROOT` and `PAL_LICENSEFILE`. Take care: not all versions of `mpirun` and `mpiexec` copy the environment of your shell to the MPI processes – you may have to modify the shell startup files to set these environment variables.

## E.6 Restrictions

Vampirtrace cannot display the internal point-to-point communication within the MPICH implementation.

Vampirtrace does not support the 64-bit mode of Solaris.

Vampirtrace is not multi-thread safe.

## Appendix F IBM SP and Workstations

### F.1 System Requirements

Vampirtrace runs on the IBM SP parallel systems and on IBM RS/6000 workstations running AIX 4.3 or compatible. It requires the IBM Parallel Environment for AIX, version 2 release 3 or compatible. It is independent from the communication device used by POE, and thus does not require the High Performance Switch.

### F.2 Installing Vampirtrace

The shell script `install-Vampirtrace` will perform the installation. First, `cd` into the Vampirtrace root directory, and start the installation by typing

```
./install-Vampirtrace
```

Then answer the questions issued by the script. You can choose the access permissions for the Vampirtrace files.

### F.3 Compiling MPI Programs with Vampirtrace

Programs that do not contain calls to the Vampirtrace API can be compiled as usual.

For programs with calls to Vampirtrace API routines, the path to the Vampirtrace include directory must be supplied to the compiler with the `-I` flag, e.g.:

```
mpcc -I/Vol/VAMPIRtrace/include -c test.c
mpxlf -I/Vol/VAMPIRtrace/include -c fttest.f
```

### F.4 Linking MPI Programs with Vampirtrace

The Vampirtrace package contains two different profiling libraries: `libVT.a` produces tracefiles suitable for the Vampir performance analysis tool, and `libDT.a` generates tracefiles for the Dimemas performance predictor. Depending on the intended use of the trace data, your application must be linked with the correct library.

The following discussion refers to `libVT.a`; the Dimemas trace library is linked in exactly the same way – just replace `libVT.a` by `libDT.a`.

To link an executable that will generate traces, include the Vampirtrace library `libVT.a` in the link command line, and append the `ld` library:

```
mpcc -o test test.o -L/Vol/VAMPIRtrace/lib -lVT -lld -lm
```

Fortran applications can be linked in the same way:

```
mpxlf -o fttest fttest.o -L/Vol/VAMPIRtrace/lib -lVT -lld
```

To run applications that have been instrumented with calls to the Vampirtrace API without generating a trace, a dummy library `libVTnull.a` is provided. It contains just the Vampirtrace API entry points, and can be linked after the MPI libraries.

### F.5 Running MPI Programs with Vampirtrace

Programs linked against the Vampirtrace libraries are started in the same way as “ordinary” POE programs with the `poe` application launcher.

### F.5.1 Recording Source–Code Locations

If the `PCTRACE` configuration option (see section 6.2) is enabled, Vampirtrace will record the source–code locations of all calls to MPI routines, of all point–to–point messages and collective operations, and of all calls to the `VT_BEGIN` and `VT_END` calls of the Vampirtrace API. This information will enable Vampir to highlight state changes and communication operations in the source–code display.

### F.6 Restrictions

Vampirtrace cannot record point-to-point communication internal to IBM’s MPI library.

Vampirtrace is currently not multi–thread safe.

## Appendix G Intel IA32 Systems and Clusters

### G.1 System Requirements

Vampirtrace runs on Intel IA32 systems running Solaris 86 2.6 or higher, or Linux with kernel version 2.2.x and support for glibc. It requires MPICH version 1.1.2 or compatible. The Vampirtrace libraries do not depend on a particular MPICH communication device: shared-memory, `ch_p4` and special customized devices will work. Thus, Vampirtrace can be used on a variety of different cluster architectures based on IA32 nodes.

Vampirtrace works with the usual C and C++ compilers, and the GNU and PGI Fortran compilers for Linux, and the Sun Fortran compilers for Solaris 86.

The MPICH distribution can be ftp'ed from <ftp://www.mcs.anl.gov/pub/mpi/mpich.tar.gz>.

### G.2 Installing Vampirtrace

The shell script `install-Vampirtrace` will perform the installation. First, `cd` into the Vampirtrace root directory, and start the installation by typing

```
./install-Vampirtrace
```

Then answer the questions issued by the script. In particular, you can choose the file access permissions, and extend the MPICH compile scripts `mpicc`, `mpif77` and `mpif90` for use with Vampirtrace.

The install script will optionally patch the MPICH compile and link scripts `mpicc`, `mpif77` and `mpif90` to include support for compiling and linking with Vampirtrace. New options (`-vt`, `-dt` and `-vtnull`) will be available that add include paths and/or libraries to the compile and link command lines. The scripts are installed into the `bin` directory, but you're free to move them anywhere afterwards; it's advisable to either replace the original scripts with them or give them alternate names, like `f.i. mpicc_VT`.

### G.3 Compiling MPI Programs with Vampirtrace

Programs that do not contain calls to the Vampirtrace API can be compiled as usual with MPICH, either with the `mpicc` and `mpif77` scripts or directly with your platform's compilers.

For programs with calls to Vampirtrace API routines, the path to the Vampirtrace include directory must be supplied to the compiler: use the `-I` flag for the C compiler and the Fortran compilers that support it, e.g.:

```
cc -I/Vol/Vampirtrace/include -I/Vol/mpi/include -c test.c
g++ -I/Vol/VAMPIRtrace/include -I/Vol/mpi/include -c cctest.cc
f77 -I/Vol/VAMPIRtrace/include -I/Vol/mpi/include -c fttest.f
pgf77 -I/Vol/VAMPIRtrace/include -I/Vol/mpi/include -c fttest.f
```

If the `-I` option doesn't work with your Fortran compiler, copy (or link) the include file `VT.inc` into the compilation directory.

When using the extended MPICH compile scripts, just specify the `-vt` or `-dt` options:

```
mpicc -vt -c test.c
mpif77 -dt -c fttest.f
```

The `mpif77` and `mpif90` scripts may create symbolic links `mpif.h` or `VT.inc` referring to the MPI and Vampirtrace include files. These links can easily become outdated or dangling; check that they refer to the correct files if you're having problems compiling or linking Fortran MPI applications. It's best to delete the links after compilation.

## G.4 Linking MPI Programs with Vampirtrace

The Vampirtrace package contains two different profiling libraries: `libVT.a` produces tracefiles suitable for the Vampir performance analysis tool, and `libDT.a` generates tracefiles for the Dimemas performance predictor. Depending on the intended use of the trace data, your application must be linked with the correct library.

The following discussion refers to `libVT.a`; the Dimemas trace library is linked in exactly the same way – just replace `libVT.a` by `libDT.a`.

To link an executable that will generate traces, include both the Vampirtrace library `libVT.a` and the profiling library `libmpich.a` in the link command line, e.g.:

```
cc -o test test.o -L /Vol/mpi/lib/linux/ch_p4 \
-L/Vol/VAMPIRtrace/lib -lVT -lmpich
```

C++ applications need the library `libmpi++.a` in addition:

```
g++ -o test test.o -L /Vol/mpi/lib/linux/ch_p4 \
-L/Vol/VAMPIRtrace/lib -lmpi++ -lVT -lmpich
```

Fortran applications need the Fortran wrapper library (usually `libfmpich.a` or `libfmpi.a`) linked in *before* `libVT.a`:

```
f77 -o ftest ftest.o -L /Vol/mpi/lib/LINUX/ch_p4 \
-L/Vol/VAMPIRtrace/lib -lfmpich -lVT -lmpich
```

or

```
pgf77 -o ftest ftest.o -L /Vol/mpi/lib/LINUX/ch_p4 \
-L/Vol/VAMPIRtrace/lib -lfmpich -lVT -lmpich
```

Take care to generate and select the matching `libfmpich.a` for your Fortran compiler – the MPI entry points are different.

Be sure to obey the order of the various libraries: `libVT.a` must be linked *before* the MPICH libraries, and *after* and the Fortran wrapper library.

When using the extended compile scripts, simply specify `-vt`:

```
mpicc -vt -o test test.o
mpif77 -vt -o ftest ftest.o
mpif90 -vt -o ftest ftest.o
```

To link with `libDT.a` for Dimemas traces, specify `-dt` instead of `-vt`.

To run applications that have been instrumented with calls to the Vampirtrace API without generating a trace, a dummy library `libVTnull.a` is provided. It contains just the Vampirtrace API entry points, and can be linked after the MPICH libraries. The extended compile scripts support the `-vtnull` option to link with the dummy library.

## G.5 Running MPI Programs with Vampirtrace

Programs linked against the Vampirtrace libraries are started in the same way as “ordinary” MPICH programs with the `mpirun` or `mpiexec` launchers. There are no Vampirtrace runtime flags, but the library will examine the environment variables `VT_CONFIG` and `VT_CONFIG_RANK` that specify a Vampirtrace configuration file (see section 6.1), and of course `PAL_ROOT` and `PAL_LICENSEFILE`. Take care: not all versions of `mpirun` and `mpiexec` copy the environment of your shell to the MPI processes – you may have to modify the shell startup files to set these environment variables.

## G.6 Restrictions

---

Vampirtrace cannot display the internal point-to-point communication within the MPICH implementation.



# Appendix H NEC SX Series

## H.1 System Requirements

This release of Vampirtrace runs on NEC SX-4 systems. It requires Super-UX version 8.1 or compatible and a matching MPI-SX MPI package supplied by NEC. For the automatic subroutine tracing (see section ) to work, the Fortran 77 compiler f77sx revision 170 or newer is needed.

## H.2 Identifying the Correct Vampirtrace Library

Which Vampirtrace library to link depends on the floating-point mode and numerical storage size used for compilation and linking.

The floating-point mode is specified by the `-float0`, `-float1`, or `-float2` flags of the Fortran compiler (`-hfloat0`, `-hfloat1`, or `-hfloat2` of the C compiler), or by the value of the environment variable `FIMOD`. The numerical storage size is specified by the `-Nw` (32-bit) or `-w` (64-bit) flags of the Fortran compiler. The following table shows which Vampirtrace library to pick; you have to prepend the path to the Vampirtrace root directory, of course.

Language	Num. Storage Size	float0	float1	float2
C	4 bytes	lib0/libVT.a	lib1/libVT.a	lib2/libVT.a
		lib0/libDT.a	lib1/libDT.a	lib2/libDT.a
Fortran	4 bytes	lib0/libVT.a	lib1/libVT.a	-
		lib0/libDT.a	lib1/libDT.a	-
	8 bytes	lib0/libVTw.a	lib1/libVTw.a	lib2/libVTw.a
		lib0/libDTw.a	lib1/libDTw.a	lib2/libDTw.a

## H.3 Installing Vampirtrace

The shell script `install-Vampirtrace` will perform the installation. First, `cd` into the Vampirtrace root directory, and start the installation by typing

```
./install-Vampirtrace
```

Then answer the questions issued by the script. You can choose the access permissions for the Vampirtrace files.

## H.4 Compiling MPI Programs with Vampirtrace

Programs that do not contain calls to the Vampirtrace API can be compiled as usual.

For programs with calls to Vampirtrace API routines, the path to the Vampirtrace include directory must be supplied to the compiler with the `-I` flag, e.g.:

```
cc -I/Vol/VAMPIRtrace/include -hfloat0 -c test.c
f77sx -I/Vol/VAMPIRtrace/include -float0 -c ftest.f
```

### H.4.1 Tracing Application Subroutines

For Fortran applications, all calls to application subroutines can be automatically traced without prior instrumentation of the source-code. To enable subroutine tracing, compile with the `-flowtrace` flag:

```
f77sx -flowtrace -I/Vol/VAMPIRtrace/include -float0 -c ftest.f
```

To link object files that have been compiled with the `-flowtrace` option, `-flowtrace` must be specified, also. The linker may complain about the symbol `_PVWORK_COMMON` to be multiply defined; this warning can be safely ignored, or disabled with the `-l-t` flag.

To actually record the subroutine calls, be sure to enable the `PCTRACE` configuration option (see section 6.2).

## H.5 Linking MPI Programs with Vampirtrace

The Vampirtrace package contains two different profiling libraries: `libVT.a` produces tracefiles suitable for the Vampir performance analysis tool, and `libDT.a` generates tracefiles for the Dimemas performance predictor. Depending on the intended use of the trace data, your application must be linked with the correct library.

The following discussion refers to `libVT.a`; the Dimemas trace library is linked in exactly the same way – just replace `libVT.a` by `libDT.a` (or `libVTw.a` by `libDTw.a`, respectively).

To create an executable that will generate traces, include the matching Vampirtrace library `libVT.a` (or `libVTw.a`) in the link command line, e.g.:

```
cc -hfloat0 -o test test.o /Vol/VAMPIRtrace/lib0/libVT.a \  
-lpmpi -lmpi -li77sx -lld -lm  
  
f77sx -hfloat0 -o ftest ftest.o /Vol/VAMPIRtrace/lib0/libVT.a \  
-l-lpmpi -l-lmpi -l-llld
```

Be sure to put the Vampirtrace library *before* the MPI library in the link command line, and to link the version corresponding to the application's floating point mode.

To run applications that have been instrumented with calls to the Vampirtrace API without generating a trace, a dummy library `libVTnull.a` is provided. It contains just the Vampirtrace API entry points, and can be linked after the MPI libraries.

## H.6 Running MPI Programs with Vampirtrace

Programs linked against the Vampirtrace libraries are started in the same way as “ordinary” MPI-SX programs with the `mpirun` application launcher.

## H.7 Restrictions

Vampirtrace supports all three floating point modes `float0`, `float1` and `float2`, and both 4-byte and 8-byte numerical storage units in Fortran (`-Nw` and `-w` flags) for the `float0` and `float1` modes. C programs are currently limited to the default 32-bit integer mode; *don't* use the `-int64` C compiler flag.

Vampirtrace cannot record point-to-point communication internal to the MPI-SX library.

## Appendix I SGI Origin and Workstations

### I.1 System Requirements

Vampirtrace runs on MIPS-based SGI machines, specifically on the Origin and on Workstations. It requires Irix 6.5 or compatible, and version 3.1 or compatible of the SGI MPI implementation. Both 64 and n32 ABIs are supported. The 32 and o32 ABIs are not supported.

### I.2 Installing Vampirtrace

The shell script `install-Vampirtrace` will perform the installation. First, `cd` into the Vampirtrace root directory, and start the installation by typing

```
./install-Vampirtrace
```

Then answer the questions issued by the script. You can choose the access permissions for the Vampirtrace files.

### I.3 Compiling MPI Programs with Vampirtrace

Programs that do not contain calls to the Vampirtrace API can be compiled as usual.

For programs with calls to Vampirtrace API routines, the path to the Vampirtrace include directory must be supplied to the compiler with the `-I` flag, e.g.:

```
cc -n32 -I/Vol/VAMPIRtrace/include -c test.c
f77 -64 -I/Vol/VAMPIRtrace/include -c fttest.f
```

### I.4 Linking MPI Programs with Vampirtrace

The Vampirtrace package contains two different profiling libraries: `libVT.a` produces tracefiles suitable for the Vampir performance analysis tool, and `libDT.a` generates tracefiles for the Dimemas performance predictor. Depending on the intended use of the trace data, your application must be linked with the correct library. Furthermore, a set of libraries is provided for both the 64 and n32 ABIs. These libraries can be found in the `lib/lib64` and `lib/libn32` directories, respectively.

The following discussion refers to `libVT.a`; the Dimemas trace library is linked in exactly the same way – just replace `libVT.a` by `libDT.a`.

To link an executable that will generate traces, include the Vampirtrace library `libVT.a` in the link command line, and append the Dwarf, ELF and exc libraries, e.g. (in n32 mode):

```
cc -n32 -o test test.o -L/Vol/VAMPIRtrace/lib/libn32 -lVT -lmpi \
-lldwarf -lself -lexc -lm
```

Fortran applications can be linked in the same way (in 64 mode):

```
f77 -64 -o fttest fttest.o -L/Vol/VAMPIRtrace/lib/lib64 -lVT -lmpi \
-lldwarf -lself -lexc -lm
```

Be sure to obey the order of the various libraries: `libVT.a` must be linked *before* the MPI library.

To run applications that have been instrumented with calls to the Vampirtrace API without generating a trace, a dummy library `libVTnull.a` is provided. It contains just the Vampirtrace API entry points, and can be linked after the MPI libraries.

### I.5 Running MPI Programs with Vampirtrace

Programs linked against the Vampirtrace libraries are started in the same way as “ordinary” SGI-MPI programs with the `mpirun` application launcher.

### **I.5.1 Recording Source-Code Locations**

If the `PCTRACE` configuration option (see section 6.2) is enabled, Vampirtrace will record the source-code locations of all calls to MPI routines, of all point-to-point messages and collective operations, and of all calls to the `VT_BEGIN` and `VT_END` calls of the Vampirtrace API. This information will enable Vampir to highlight state changes and communication operations in the source-code display.

## **I.6 Restrictions**

Vampirtrace cannot record point-to-point communication internal to SGI’s MPI library.

Vampirtrace is currently not multi-thread safe.