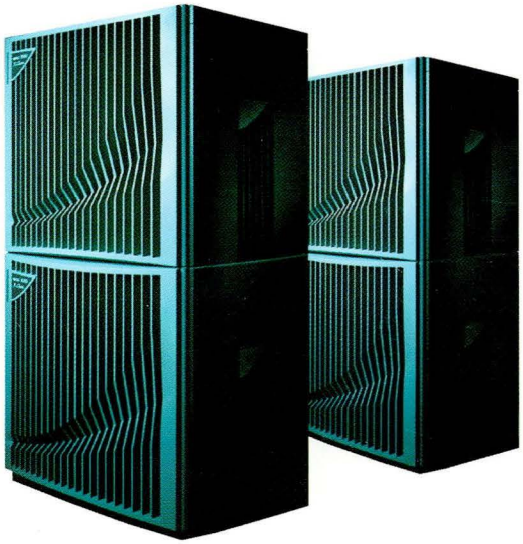
HEWLETT® PACKARD

Exemplar
S-Class and
X-Class Servers



CXtrace User's Guide

Third Edition

Hewlett-Packard Company
Convex Division
3000 Waterview Parkway
P.O. Box 833851
Richardson, TX 75083-3851
United States of America

CXtrace User's Guide

Exemplar S-Class and X-Class Servers

B5639-90003

Third Edition

January 1997

Hewlett-Packard Company
Convex Division
Richardson, Texas
United States of America

CXtrace User's Guide

Exemplar S-Class and X-Class Servers

B5639-90003

Notice

The information contained in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

This entire book is recyclable.

Printed in the United States of America

Revision Information for

CXtrace User's Guide

Exemplar S-Class and X-Class Servers

Edition	Document No.	Description
Third	B5639-90003	Update for release with CXtrace V2.4, January, 1997.
Second	710-029930-002	Update for release with CXtrace V2.1 May, 1996.
First	710-029930-001	Initial release, March, 1994.

Contents

Figures

Tables

How to use this guide

Purpose and audience

This user's guide describes the trace-based performance analysis tool, CXtrace, and explains how to use it to profile or debug timing problems for MPI and PVM applications on parallel systems.

This book is written for experienced programmers who use Exemplar S2000 and X2000 systems to develop and tune C or Fortran 77 programs that employ message-passing techniques.

Organization

Use the following table to select which portions of the book you wish to read:

Table 1 Document organization

To learn...	Read...
How CXtrace works and how to perform the basic steps involved in profiling programs with CXtrace	Chapter 1. "Getting started"
How to use compilation scripts to build and instrument applications for profiling with CXtrace	Chapter 2, "Using compilation scripts"
How to use `xinstrument` to instrument your source code at a finer level of detail	Chapter 3, "Instrumenting code"
How to use the `tv` trace view utility and the `tally` report generator to analyze the profiling data collected	Chapter 4, "Viewing trace files"
How to specify parameters in the .MONITOR file for controlling CXtrace monitoring library operations	Chapter 5, "Using the CXtrace monitoring library"
Detailed information about trace records	Appendix A, "Trace record information"

Notational conventions

Notational conventions used in this book are as follows:

`Bold monospace`

In command examples, text shown in **`bold monospace`** identifies user input that must be typed exactly as shown.

`monospace`

In paragraph text, `monospace` identifies command names, system calls, and data structures and types. In command examples, `monospace` identifies command output, including error messages.

Italic

In paragraph text, *italic* identifies new and important terms and titles of documents.

In command syntax diagrams, *italic* identifies variables that must be supplied by the user. The following command example indicates that definition of the variable *output_file* is optional:

`command` *input_file* [*output_file*]

Associated documents

The following documents contain useful supplementary information applicable to CXtrace:

- *SPP-UX System Administration Guide*, Fifth Edition, Order No. B5655-90002
- *Exemplar Programming Guide*, Fourth Edition, Order No. B5600-90001
- *HP PVM User's Guide*, First Edition, Order No. B5885-90001
- *HP MPI User's Guide*, First Edition, Order No. B6011-90001
- *Exemplar C and Fortran 77 Programmer's Guide*, First Edition, Order No. B5600-90002

Ordering documents

To order the current edition of these or any other Convex Division documents, send requests to:

Hewlett-Packard Company
Convex Division
Customer Service
P.O. Box 833851
Richardson TX 75083-3851 USA

Please include the order number (*xxxxx-9xxxx*) or the exact title of the document.

Technical assistance

If you have questions that are not answered in this book, contact the Hewlett-Packard Convex Technical Assistance Center (TAC) at the following locations:

- Within the continental U.S., call 1 (800) 952-0379.
- From Canada, call 1 (800) 345-2384.

All other locations, contact your local Hewlett-Packard office.

You can also use the `contact` utility, if you would like to report any problems you may have with CXtrace or its associated documentation. For more information refer to the contact(1) man page.

Getting started 1

What is CXtrace?

CXtrace is a performance analysis tool for applications coded in C and/or Fortran 77. Its instrumentation and monitoring systems help you improve your parallel applications by capturing and visualizing execution trace information. CXtrace is particularly useful for analyzing MPI and PVM applications. Its instrumentation, monitoring, and analysis tools are designed to capture, associate, and display messaging events between processes.

CXtrace's main software components include compilation scripts for building and instrumenting applications, a GUI-based source-level instrumentor, a runtime performance monitoring library, and a set of analysis tools that work together to measure and display your program's performance.

- The compilation scripts provide a convenient way to automate the building and instrumenting of applications for profiling with CXtrace.
- The GUI-based interactive source-code instrumentor, `xinstrument`, inserts performance monitoring routines into the application's source code.
- The runtime performance monitoring library, or *monitor*, provides a set of routines that record events and event-related data extracted from the execution of an application. The event data is used by the analysis tools to measure and display various aspects of program performance, such as message-passing overhead, synchronization overhead, and time spent in instrumented loops and routines.
- Using the trace view utility, `tv`, you can zoom and scroll both horizontally and vertically. Events are presented in time order for each process in the application. MPI and PVM messaging events are graphically associated to provide insights into the messaging behavior of the application.

- The report generator utility, `tally`, provides performance statistics for the application, based on the trace file data. These statistics provide insights into the general behavior of the program.

Profiling example

The easiest way to learn how to use CXtrace is to take a small program and profile it. The following example uses a simple MPI program written in C (ping_pong.c) to illustrate the profiling process.

Step 1 Build and instrument the application for profiling with one of the compilation scripts provided in the directory /opt/cxtrace/bin.

```
% /opt/cxtrace/bin/cxtrace-c89 ping_pong.c +e -mpi
```

The above command invokes the `cxtrace-c89` compilation script to compile and instrument the source file ping_pong.c. The script calls the `c89` compiler and passes it the `+e` option. The `-mpi` option specifies that the program is an MPI application, so that the correct instrumentation and monitoring routines are called. The required MPI libraries and include files are also specified by the script. Use the `-pvm` option if you are profiling a PVM application. The default is MPI.

Step 2 Run the instrumented executable to collect profiling data in a trace data file.

```
% a.out -np 2 500
MONITOR (host):Cannot open .MONITOR file. Using
defaults.
Opening /tmp/mon_log.1 as log file
Opening /tmp/mon_log.0 as log file
Buffer Size=262144  Flush level=1  Trace Level=4
No sensor profile specified
ping-pong 500 bytes ...
500 bytes: 54.06 63.06 664.53 usec/msg
500 bytes: 9.25 7.93 0.75 MB/sec
Compiling Trace file...please be patient.
Trace data written to CXTRACE.OUT
```

The command `a.out -np 2 500` runs the instrumented executable and creates two processes. As the program runs, it writes trace data to the file CXTRACE.OUT.

The messages `MONITOR (host):Cannot open .MONITOR file. Using defaults` and `No sensor profile specified` occur because the compilation scripts do not create .MONITOR or profile files. These messages can be ignored. These files are not required by the compilation scripts.

Step 3 Use the `tv` trace view utility or the `tally` statistical report generator to analyze the trace data. To use `tv`, enter:

```
% setenv DISPLAY mydisplay:0.0
% /opt/cxtrace/bin/tv CXTRACE.OUT &
```

The commands above set the X `DISPLAY` environment variable and invoke the `tv` trace view utility on the trace file CXTRACE.OUT. The entire trace file is initially graphed in the window. Select Zoom from the Commands menu to increase the legibility of the graph and to focus on specific regions. The trace view for example program ping_pong.c is shown in Figure 1.

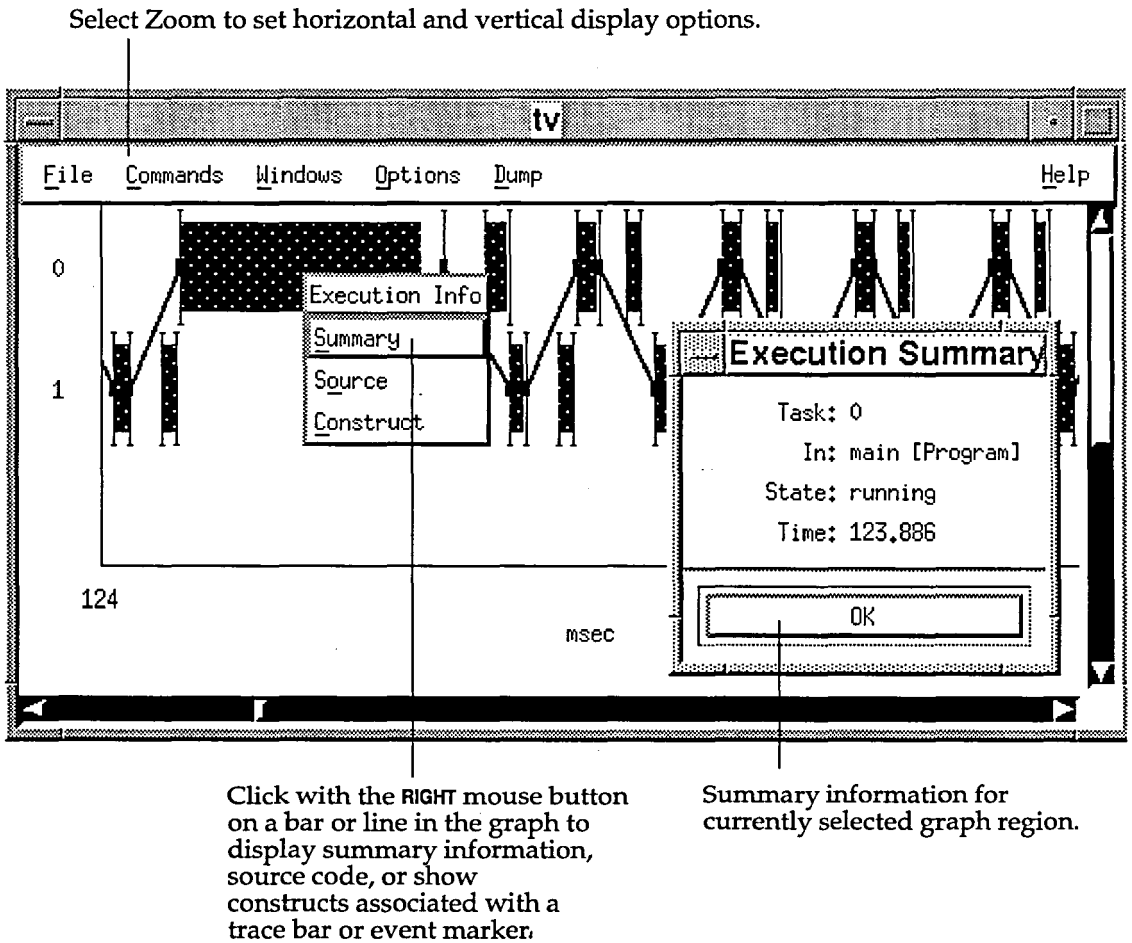


Figure 1 `tv` graph display (zoomed in)

Refer to "Using the trace view tool" on page 34 for more information on using `tv`.

To use `tally`, enter a command similar to the following:

```
% /opt/cxtrace/bin/tally CXTRACE.OUT
```

The above command invokes the `tally` text report generator on the trace file CXTRACE.OUT. The output from `tally` is sent to stdout and written to the files ncpu.summary and tally.summary in the current directory. Refer to "tally" on page 41 for a detailed discussion of the summary information reported by `tally`.

Read Chapter 2, "Using compilation scripts," for more information about building applications with the compilation scripts, including full syntax, options, and examples.

CXtrace process overview

This section describes the steps in the profiling process in more detail. The performance analysis process is illustrated in Figure 2.

The first step is instrumenting your application. This instrumentation produces an application database, an enabling profile, and an instrumented version of your application source code. The application database is used by the analysis tools to correlate performance information back to the application source code. The enabling profile is used to control runtime performance data collection.

Step two is to compile and link your application with the CXtrace runtime monitoring library.

If you are using the compilation scripts to build your application for CXtrace, steps one and two are performed automatically for you. If you wish to profile your code at a finer level of detail, use `xinstrument` to manually instrument your source code. Read Chapter 3, "Instrumenting code," for details on how to use `xinstrument`.

The third step is to execute the application under MPI or PVM. While the application executes, CXtrace's runtime monitoring routines produce a trace data file.

The final step is to analyze the trace data file using `tv` and/or `tally`.

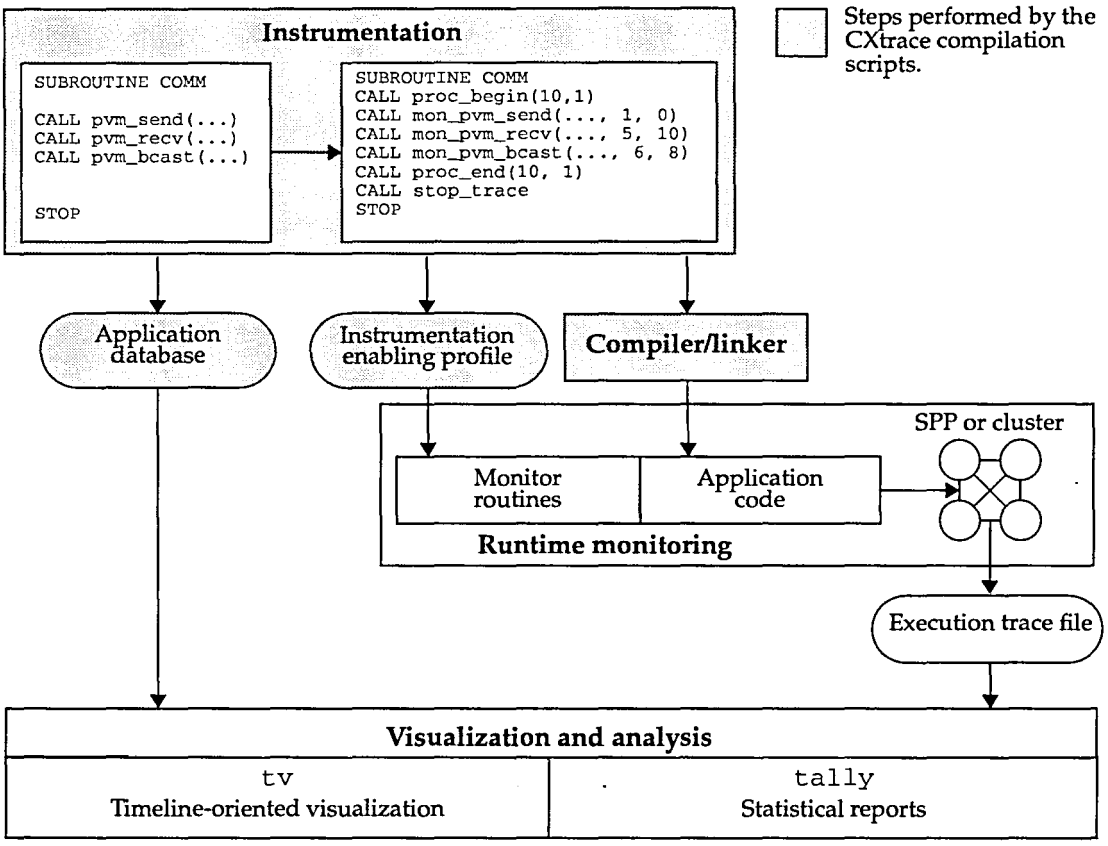


Figure 2 Parallel program execution with CXtrace

Using compilation scripts 2

This chapter describes how to instrument and compile your application for CXtrace using compilation scripts and how to run the instrumented application to generate a trace file.

This method is preferred over the instrumentation methods discussed in Chapter 3, "Instrumenting code."

Overview

The CXtrace compilation scripts are located in the /opt/cxtrace/bin directory. These scripts help to simplify the process of building and instrumenting applications for CXtrace and are recommended for most users of CXtrace.

When using the compilation scripts, the instrumentation and enabling of routines and message-passing constructs is automatically done for you.

By default, all routines or functions containing message-passing constructs (calls to functions in MPI or PVM message-passing libraries) and all message-passing constructs are enabled. If you want to instrument and profile at a finer granularity than what the compilation scripts provide as a default, refer to Chapter 3, "Instrumenting code."

The following compilation scripts are provided:

- `/opt/cxtrace/bin/cxtrace-cc`
- `/opt/cxtrace/bin/cxtrace-c89`
- `/opt/cxtrace/bin/cxtrace-f77`
- `/opt/cxtrace/bin/cxtrace-fort77`
- `/opt/cxtrace/bin/cxtrace-ar`

The C and Fortran 77 compilation scripts:

- Call the appropriate instrumentation routines for your application. These routines instrument the source files given as arguments to the compiler.

- Create an instrumented version of each source file. The instrumented source files are removed after compilation.
- Link in the specified CXtrace runtime monitoring libraries, either PVM or MPI. The default is MPI. The scripts also link in the required message-passing libraries and include files (for example, `-lpvm3` or `-lmpi`).
- Create an application database file for the entire application (APPL_DB). The APPL_DB file is placed in the directory from which you invoked the compilation script. CXtrace also creates a separate .appl_db file for each object file.
- Build an instrumented executable with the default set of constructs enabled.

Script syntax and options

The syntax for using the C and Fortran 77 compilation scripts is shown below:

`/opt/cxtrace/bin/`*script* `{-mpi|-pvm}` *compile_line*

script

The name of the compilation script to use. Valid values are listed below:

`cxtrace-c89`—Passes the *compile_line* to the /opt/ansic/bin/c89 compiler.

`cxtrace-cc`—Passes the *compile_line* to the /opt/ansic/bin/cc compiler.

`cxtrace-f77`—Passes the *compile_line* to the /opt/fortran/bin/f77 compiler.

`cxtrace-fort77`—Passes the *compile_line* to the /opt/fortran/bin/fort77 compiler.

`-mpi`

Links in the required CXtrace runtime monitoring libraries for MPI and the appropriate MPI libraries and include files. This is the default. This option is incompatible with the `-pvm` option.

`-pvm`

Links in the required CXtrace runtime monitoring libraries for PVM and the appropriate PVM libraries and include files. This option is incompatible with the `-mpi` option.

compile_line

The compilation line to pass to the target compiler identified by the compilation script: source files, compiler options, object files, and libraries.

The `cxtrace-ar` script is used in place of `ar` in Makefiles or on the command-line. It merges the application database files for each of the object files passed as arguments to `ar` into *lib*.a.appl_db, where *lib*.a is the target library for `ar`.

The syntax for using the `cxtrace-ar` script is shown below:

`/opt/cxtrace/bin/cxtrace-ar` *ar-options*

Refer to the `ar` manpage for information about the `ar` utility and options.

Before you run the instrumented executable to generate a trace file, you should be aware of the following:

- If your application does not run to completion, a trace file will *not* be generated.
- If your application is large or a large number of constructs are enabled, the generated trace file can potentially be 1 to 2 Gbytes in size.
- Space in /tmp may be required when instrumenting applications. You can use the UNIX `TMPDIR` environment variable to specify a different location for temporary files created by CXtrace.
- Loading large trace data files into `tv` or `tally` and updating the GUI in `tv` when analyzing large files can take a long time.

Examples

The examples in this section show how to compile and instrument executables and archive libraries with the compilation scripts and how to modify a Makefile to use the compilation scripts.

Fortran 77 PVM example

To compile a Fortran 77 PVM application for CXtrace, use a command line similar to the following:

```
% /opt/cxtrace/bin/cxtrace-f77 -pvm spmd.f
```

In the above example, the `cxtrace-f77` compilation script compiles and builds an instrumented PVM executable for profiling with CXtrace. The `-pvm` option is required and ensures that the correct runtime monitoring libraries, message-passing libraries, and include files are specified.

Makefile example

This example is based on a sample Makefile that is shipped with HP PVM examples in the directory /opt/pvm3/examples. Only a few changes are needed to modify the Makefile so that the executables are instrumented for profiling with CXtrace:

- Change the CC line so that the cxtrace-cc compilation script is called in place of the /opt/ansic/bin/cc compiler.
- Change the F77 line so that the cxtrace-f77 compilation script is called in place of the /opt/ansic/bin/f77 compiler.
- Add the -pvm option to the CFLAGS and FFLAGS lines so that the correct runtime monitoring libraries are linked in (in this case, PVM).

```
PVM_ROOT     = /opt/pvm3
PVM_ARCH     = CSPP
PVMIDIR      = $(PVM_ROOT)/include
PVMLDIR      = $(PVM_ROOT)/lib/$(PVM_ARCH)
PVMLIB       = -Wl,-lgpvm3 -Wl,-lpvm3 -Wl,-lcnx_syscall /lib/libail.sl
PVMFLIB      = -Wl,-lfpvm3 $(PVMLIB)

OPT          = +O0
CC           = /opt/cxtrace/bin/cxtrace-cc
CFLAGS       = $(OPT) -pvm -I$(PVMIDIR) -Wl,-L$(PVMLDIR)
F77          = /opt/cxtrace/bin/cxtrace-f77
FFLAGS       = $(OPT) -pvm +es +U77 -I$(PVMIDIR) -Wl,-L$(PVMLDIR)

C_PROGS = gexample hello hello_other master1 nntime slave1 spmd \
                                        timing timing_slave
F_PROGS = fgexample fmaster1 fslave1 fspmd hitc hitc_slave mm3_m mm3_w testall

default: hello hello_other

all: c-all f-all

clean:
        rm -f $(C_PROGS) $(F_PROGS)
hello: hello.c
        $(CC) $(CFLAGS) -o $@ hello.c $(PVMLIB)
hello_other: hello_other.c
        $(CC) $(CFLAGS) -o $@ hello_other.c $(PVMLIB)
timing_slave: timing_slave.c
        $(CC) $(CFLAGS) -o $@ timing_slave.c $(PVMLIB)
timing: timing.c
        $(CC) $(CFLAGS) -o $@ timing.c $(PVMLIB)
spmd: spmd.c
        $(CC) $(CFLAGS) -o $@ spmd.c $(PVMLIB)
        $(F77) $(FFLAGS) -o $@ spmd.f $(PVMFLIB)
master1: master1.c
        $(CC) $(CFLAGS) -o $@ master1.c $(PVMLIB)
```

```
slave1: slave1.c
        $(CC) $(CFLAGS) -o $@ slave1.c $(PVMLIB)
fmaster1: master1.f
        $(F77) $(FFLAGS) -o $@ master1.f $(PVMFLIB)
fslave1: slave1.f
        $(F77) $(FFLAGS) -o $@ slave1.f $(PVMFLIB)
testall: testall.f
        $(F77) $(FFLAGS) -o $@ testall.f $(PVMFLIB)
hitc: hitc.f
        $(F77) $(FFLAGS) -o $@ hitc.f $(PVMFLIB)
hitc_slave: hitc_slave.f
        $(F77) $(FFLAGS) -o $@ hitc_slave.f $(PVMFLIB)
gexample: gexample.c
        $(CC) $(CFLAGS) -o $@ gexample.c $(PVMLIB)
fgexample: gexample.f
        $(F77) $(FFLAGS) -o $@ gexample.f $(PVMFLIB)
nntime: nntime.c
        $(CC) $(CFLAGS) -D$(PVM_ARCH) -DPVM -o $@ nntime.c $(PVMLIB)
mm3_m: mm3_m.f
        $(F77) $(FFLAGS) -o $@ mm3_m.f $(PVMFLIB)
mm3_w: mm3_w.f
        $(F77) $(FFLAGS) -o $@ mm3_w.f $(PVMFLIB)
```


Archive library example

The following example shows how to build an archive library that is instrumented for CXtrace and link it with instrumented object files to build an instrumented executable.

Assume the application is an MPI application written in C containing three files: file1.c, file2.c, and file3.c. To create an archive library that is instrumented for CXtrace that contains file1.c and file2.c, enter the following commands:

```
% /opt/cxtrace/bin/cxtrace-cc  -mpi -c file1.c file2.c
% /opt/cxtrace/bin/cxtrace-ar  r  mylib.a file1.o file2.o
```

To compile and instrument file3.c and link it with the instrumented archive library you just created, enter the following command:

```
% /opt/cxtrace/bin/cxtrace-cc -mpi file3.c mylib.a
```

In the above example, the `-mpi` is not required because MPI is the default platform.

Instrumenting code 3

When you instrument your source files, CXtrace inserts calls to monitoring routines into your source code. These calls replace or surround instrumentable constructs, such as loops and MPI or PVM function calls that send or receive messages or perform synchronization operations.

This chapter provides information on how to use `xinstrument` to instrument your source code at a finer granularity than that provided by the compilation scripts described in Chapter 2. For most users of CXtrace, the compilation scripts are the preferred method for instrumenting code.

The following topics are covered:

- Instrumentation files
- Invoking `xinstrument` and its command-line options
- Instrumentor specifics
- Instrumentor preprocessing
- Using `xinstrument`
- `xinstrument` windows and menus

Press **F1** in the `xinstrument` main window to view a summary of this information.

Instrumentation files

In addition to inserting instrumentation at appropriate locations in program code, `xinstrument` generates two important files: an *application database* and an *instrument-enabling profile.*

Application database file

The application database file, named APPL_DB by default, contains information about the static structure of an application's source code. The database file is incorporated at the beginning of the trace file produced by executing the instrumented application.

The analysis tools use this information to relate traced events to instrumented constructs in the source code. A *construct* is a loop, routine, or language-specific keyword the instrumentor recognizes as instrumentable.

Instrument-enabling profile file

The instrument-enabling profile is a table of flags, one for each construct in the application database. This profile is used in the following ways:

- By the instrumentor, to select the constructs to be instrumented.
- By the monitor, to select the instrumented constructs to be traced.

Only those constructs whose flags are true in the profile are traced. By creating a profile file that contains this information, you can change the instrumentation of your application without rebuilding it. Profile files are created with `xinstrument` and have a default extension of .pro. Read the section "Profile menu" on page 22 for instructions.

If a profile file does not exist, the default set of constructs is enabled for tracing. By default, routines containing message-passing constructs and all message-passing constructs are enabled.

Invoking xinstrument

The `xinstrument` program coordinates source code instrumentation with minimal user-intervention. An X-based interface facilitates the instrumentation of source files as well as program constructs within those files. You can save instrumentation specifications in a file, called a *profile*, for later use.

You should invoke `xinstrument` from your source code directory. Syntax and options for invoking `xinstrument` are shown below:

```
% /opt/cxtrace/bin/xinstrument [options]
```

options can be any of the following:

`-adb` *path*

Specifies the application database file to load from *path* (default is ./inst/APPL_DB).

`-oadb` *path*

Specifies the output application database file to write to *path* (default file name is ./inst/APPL_DB).

`-overwrite`

Overwrites the existing application database file.

`-help`

Prints usage options and exits.

`-output` *dir_path*

Specifies the directory name where to write instrumented files (default directory is ./inst).

`-origin` *dir_path*

Specifies the full path name for the root directory of the source tree (default is current working directory).

`-platform [c-pvm|fortran-pvm|fortran-mpi|c-mpi]`

Specifies which platform to use to load the files that follow this argument. The platform specifies the source language and whether PVM or MPI is used. The default is `c-pvm`. This option is specific to `xinstrument` and may appear more than once in a command line.

`-enable` *file*

Specifies an initial instrumentation profile file for the files that follow this argument. If this argument is not specified, the default profile is used. If a file is specified, it must refer to a profile file previously created with `xinstrument`. This option may appear more than once in a command line and affects the instrumentation of the files that follow this option.

`-verbose`

Prints file names to be instrumented.

`-run_pp`

Runs a language-specific preprocessor (`/opt/ansic/lbin/cpp` by default).

`-no_pp`

Does not run a preprocessor.

`-pp_command` *command*

Runs preprocessor *command* on the source files.

`-pp_options` *switches*

Passes *switches* to specified preprocessor for source files.

Command-line *options* that `xinstrument` accepts also include the following X-toolkit options:

`-fn` *font*

Specifies the font to use instead of the default font. The default is `fixed`.

`-bg` *color*

Specifies the background color.

`-fg` *color*

Specifies the color of the text (foreground color).

`-display` [*host*]:*server*[.*screen*]

Specifies the workstation (*host*) to display `xinstrument` windows.

Refer to your X Window System documentation for more information on X-toolkit options.

Application-specific X resources for CXtrace are specified in the file /opt/cxtrace/newconfig/X11/CXtrace.

Instrumentor specifics

Use `xinstrument`, the interactive GUI-based instrumentor, to refine an application's instrumentation profile.

`xinstrument` uses the application database to determine which files to consider for instrumentation. The database contains information about application source files for use by the analysis tools to relate the trace file to the source code. If no database is specified, the `APPL_DB` file in the output directory is used.

`xinstrument` does not write uninstrumented files into the output directory. You must copy them after the directory has been created by selecting Copy File(s) from the File menu.

`xinstrument` does not require all source files to be stored in the same directory. For multiple paths, include the `-origin` option to define the root of a multidirectory source tree.

You must reload files into the database whenever you modify them. Although it is possible to instrument a modified file without reloading it into the database, doing so may cause CXtrace to exit abruptly.

Instrumentor preprocessing

In order to instrument C source code, all macro definitions must first be resolved so that the instrumentor knows what constructs are instrumentable. The C instrumentor relies on a preprocessor, usually `cpp`, to provide this information.

Preprocessing using source code and makefiles

In CXtrace, all preprocessing occurs at instrumentation time, not at compile time. For example, if your code uses `#ifdef` statements and you do not specify the `-D` flag as a preprocessor argument (because you rely on having it in your makefile), CXtrace may exit abruptly if you continue using it.

Figure 3 contains sample source code and a makefile that builds an uninstrumented application containing preprocessor macros.

```
/* Source for lucky.c */
main() {
int  i;
#ifdef  TEST
for (i=0; i< 52; i++) {
    compute_lucky_numbers();
}
#endif
}
compute_lucky_numbers() {
< source deleted for brevity >
}

# Makefile for lucky.c
lucky: lucky.c
  cc -DTEST -o lucky lucky.c
```

Figure 3 Source containing preprocessor macros

Preprocessing at the command line

To define preprocessing information on the command line for the source code shown in the example above, use the `-pp_options` flag:

```
% xinstrument -pp_options -DTEST lucky.c
```

You can also specify preprocessor options from the Options menu in the `xinstrument` main window.

For C files, the default preprocessor is `/opt/ansic/lbin/cpp`. To override this default, use the `-pp_command` option. In `xinstrument`, you can also use the Options menu to set another preprocessor. For Fortran files, there is no default preprocessor. If you do not need a preprocessor, use the `-no_pp` option.

Using xinstrument

This section describes basic steps for using `xinstrument`.

To use `xinstrument`:

Step 1 Make sure your X `DISPLAY` environment variable is set, then invoke `xinstrument` from the shell. For example:

```
% setenv DISPLAY mydisplay:0.0
% /opt/cxtrace/bin/xinstrument
```

The main `xinstrument` window displays. An application database file was not specified from the command line, so the File table is initially empty.

Step 2 Select Load Module(s) from the File menu to open the Load Module(s) dialog.

Step 3 Use the Platform option menu to specify the language and type of message-passing libraries used by your application. Choices are C-PVM, C-MPI, Fortran-PVM, or Fortran-MPI.

Step 4 Select the files in the application you want to instrument, then choose Done. Refer to "Load Module(s)" on page 21 for instructions on loading files.

Step 5 Select Copy File(s) from the file menu to copy uninstrumented source files, such as header files, to the instrumentation directory. The default instrumentation directory is ./inst.

Step 6 Use the Profile menu options to enable and disable types of constructs for profiling.

You can also double-click on a file name in the list to open a construct tree window displaying all instrumentable constructs for that source file. From this window you can enable or disable specific constructs within that file.

Step 7 Select Save from the Profile menu to save the application instrumentation profile to a file. Use a .pro extension when naming the profile file.

Step 8 Highlight all the files to be instrumented in the File list in the main window, then click on the Instrument button to instrument the selected files.

Step 9 Select Exit from the File menu to exit `xinstrument`.

Step 10 Use a text editor to create a file named .MONITOR.

The .MONITOR initialization file specifies the location of the profile file you just created, as well as other parameters. Refer to "Parameters" on page 48 and "Example .MONITOR file" on page 50. The .MONITOR file must reside in the directory from which the instrumented application is run.

Step 11 Change to the directory containing the instrumented source files and build the application.

You must compile and link the instrumented source files with the CXtrace monitoring libraries /opt/cxtrace/lib/libpvmcxtrace.a or /opt/cxtrace/lib/libmpicxtrace.a (`-lpvmcxtrace` or `-lmpicxtrace`), in order to create an instrumented executable.

xinstrument windows and menus

The main window of `xinstrument` contains three selections:

- Menu bar
- Files list
- Instrument button

Figure 4 illustrates the `xinstrument` main window.

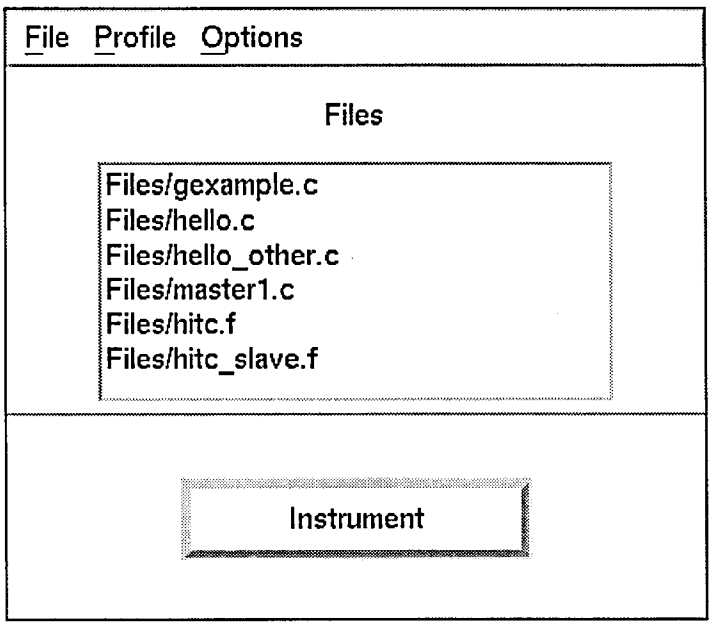


Figure 4 `xinstrument` main window

The Menu bar contains the following selections:

- File
- Profile
- Options

File menu

The File menu contains the Use Database, Load File(s), Copy File(s), and Exit selections.

Use Database

The Use Database selection allows you to load files for instrumentation that are defined in an existing application database file using the following steps:

Step 1 Select a directory by double clicking on a path name.

Step 2 Select a file by clicking on the file name.

Step 3 Choose OK to load the database file.

Step 4 Choose Cancel to close the window.

Load Module(s)

The Load Module(s) selection allows you to select and load source files for instrumentation, using the following steps:

Step 1 Select a directory by double clicking on a path name.

Step 2 Select a single file by clicking on the file name. Select multiple files by holding down CTRL and clicking on file names. Select a group of files by clicking on one file name and dragging the pointer over files you want to load.

Step 3 Select the required platform (C-PVM, C-MPI, Fortran-PVM, or Fortran-MPI) for the selected files by clicking on the Platform option menu.

Step 4 Choose Load to load the files.

Step 5 Repeat steps one through four to load additional files.

Step 6 Choose Done to close the window.

Copy File(s)

The Copy File(s) selection allows you to copy uninstrumented files into the instrumentation (origin) directory using the following steps.

Step 1 Select a directory by double clicking on a path name.

Step 2 Select a single file by clicking on the file name. Select multiple files by holding down CTRL and clicking on file names. Select a group of files by clicking on one file name and dragging the pointer over files you want to load.

Step 3 Choose Copy to copy the selected files into the instrumentation directory.

Step 4 Repeat steps one through three to copy additional files.

Step 5 Choose Done to close the window.

Exit

The Exit selection terminates `xinstrument`.

Profile menu

Selections from this menu apply to files highlighted in the Files field of the main window. Options and their functions are:

Disable All
: Disables all listed constructs.

Enable All
: Enables all constructs.

Set Default
: Enables only the default constructs.

Enable By Type
: Enables specific constructs.

Disable By Type
: Disables specific constructs.

Load
: Loads defined profile.

Save
: Saves current profile.

Options menu

The Options menu contains selections that deal with the output directory and preprocessor actions.

Set Output Directory

Specifies where to save the instrumented files. Changes to the output directory do not take effect until you click on OK or press RETURN.

Set Preprocessor command

Sets the preprocessor commands. If set, also select Run Preprocessor from this menu.

Set Preprocessor Options

Sets the preprocessor switches. If set, also select Run Preprocessor from this menu.

Run Preprocessor

Runs the preprocessor when you click on the Instrument button. It uses information provided in the Set Preprocessor Options. Once set, a highlighted toggle to the left of this selection appears.

Files list

Files to instrument

The middle panel of the main window shows all of the files in the application database. To select the files you want to instrument, follow these steps:

Step 1 Select files by clicking on the file name. Select multiple files by holding down CTRL and clicking on file names.

Step 2 Choose a profile for the selected files.

Step 3 Repeat steps one and two to profile all files.

Step 4 Select files to instrument.

Step 5 Click on the Instrument button.

Displaying constructs

Within each file are a number of constructs. You can display the constructs to be instrumented by double clicking on the file of interest. Doing so brings up a window with all instrumentable constructs for that file, as shown in Figure 5.

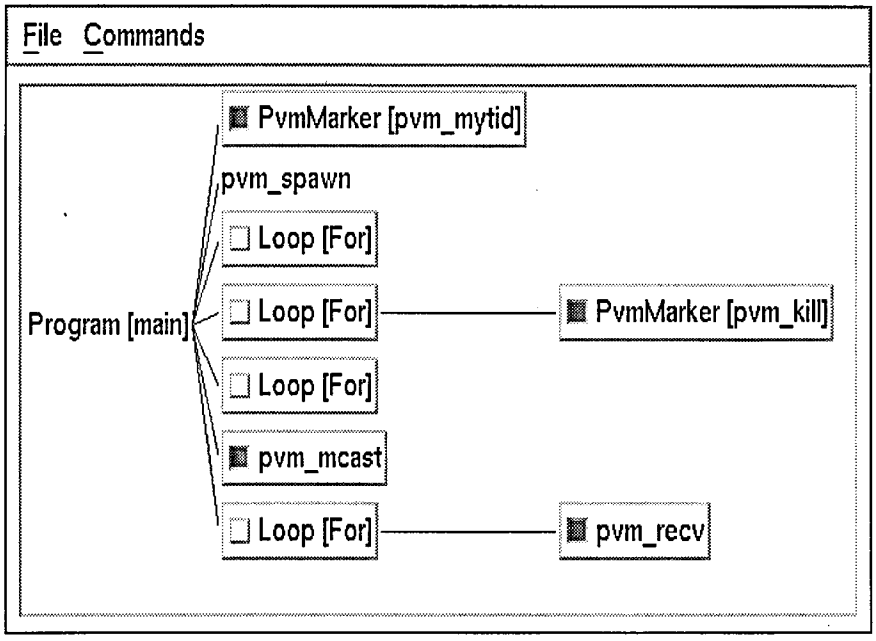


Figure 5 Constructs displayed

Clicking the left mouse button on a construct enables or disables that construct. Enabled constructs display a highlighted toggle button and are instrumented when the file is instrumented. Clicking the right mouse button on a construct displays its source code.

To quickly enable and view groups of constructs, use the dialog boxes avalilable through the Commands menu. You enable or view specific (or all) constructs by clicking the left mouse button on the appropriate toggle. Figure 6 shows the Enable Constructs dialog.

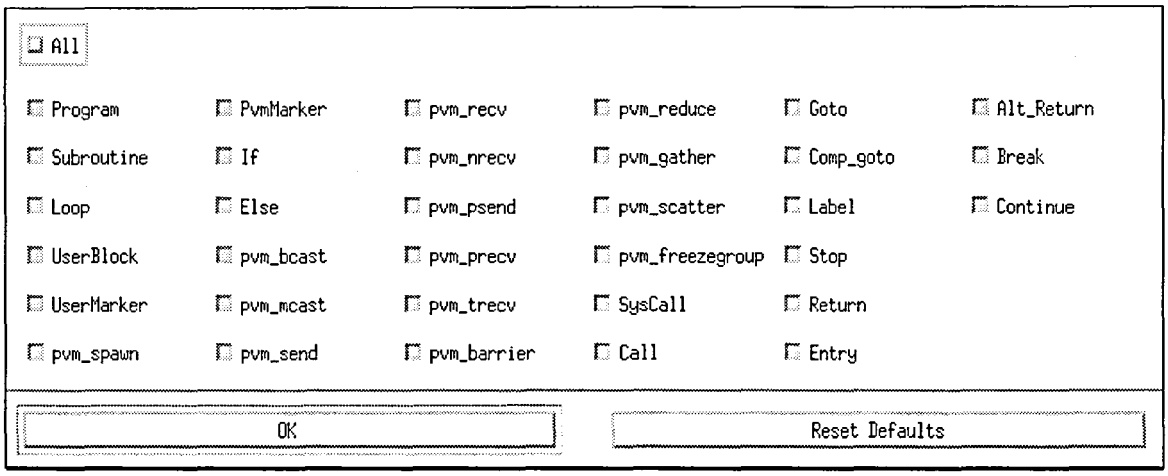


Figure 6 Enable Constructs dialog

By default:

- Message-passing and group operations and subroutines that contain these operations are selected.
- Loops and subroutines that do not contain message-passing or group operations are not selected.

Instrument button

Clicking on Instrument causes all selected files to be instrumented. The resulting files (instrumented source files and application database file) are stored in the output directory with the same name as the original source file.

Instrumentor directives

The automatic instrumentation provided by `xinstrument` may not be sufficient for your purposes. For instance, you may want to:

- Instrument constructs that `xinstrument` does not recognize
- Instrument phases of your program's execution by hand
- Selectively instrument a large loop

To handle these situations, CXtrace provides several instrumentor directives that you can add to the uninstrumented source code. You insert instrumentor directives into a program as subroutine calls (for Fortran source files) or function calls (for C source files).

When the instrumentor processes these directives, it looks for these special directive routines and replaces them with calls to monitor routines. For example, you might use the Fortran call

```
CALL insert_marker('Starting FFT')
```

to mark the beginning of a computation. The instrumentor replaces this call in the instrumented code with

```
CALL fpoint_marker(0,11)
```

The text string is stored in the application database and accessed by the key '11.'

When the instrumented program is executed, the monitor routine generates a `MARKER` trace record, which is displayed in the `tv` trace window.

You can compile and run uninstrumented source code containing instrumentor directives if the code is linked with the monitor libraries. Instrumentor directive routines have no effect in uninstrumented code.

Trace control

The `begin_trace` and `end_trace` directives turn tracing on and off. While the automatic instrumentor selects certain constructs to be traced, it does not allow you to specify general conditions for tracing. However, inserting `begin_trace` and `end_trace` directives allows you to trace when something unusual happens; for example, inside `IF` statements, or when a value gets particularly large.

If two `begin_traces` are called without an intervening `end_trace`, the second call has no effect.

These directives do not take any arguments.

Flush control

The `flush_trace` directive causes the monitor routines of the calling process to immediately write its buffered trace records to disk and clear its event buffer. Each process stores the records generated by the monitoring routines in a memory buffer.

Although the monitor automatically flushes when the record buffer is full, it may occur at an inconvenient time. Inserting `flush_trace` directives prevent undue perturbation, especially when the record buffer is small.

This directive does not take any arguments.

User-defined code blocks

The `begin_block` and `end_block` directives treat a segment of a subroutine as a subroutine itself. The analysis tools collect data for this region like a subroutine. You use these directives to provide more details, by instrumenting small blocks within an instrumented subroutine, or fewer details, by creating blocks around calls to subroutines and deselecting them.

These directives take a single character string argument, which is the name you associate with the block. The string arguments in a pair of calls should match, as shown in this example:

```
CALL begin_block ('Initialize')
.
.
.
CALL end_block ('Initialize')
```

You can nest blocks, but do not overlap them. Structure blocks in such a way that statements inside can be replaced by subroutines. For example, the following constructs are *not* allowed:

```
CALL begin_block('Bad Block')
IF (i .EQ. 1) THEN
...
CALL end_block('Bad Block')
...
END IF
```

User-defined code point

When you execute this directive, the `insert_marker` directive inserts a time-stamped marker event into the trace buffer. This directive takes a single character string argument, which is the name of the marker. For example:

```
CALL insert_marker('test')
```

The marker is displayed in the `tv` window as a vertical line.

Instrumentor limitations

The following limitations exist when instrumenting code.

Limitations when using labels

The instrumentors do not account for labels that can cause problems. For example, when the instrumentor sees a Fortran END statement, it inserts a few monitor routines before that statement. See Figure 7.

Source code

```
      send_id = pvmfsend(...)
   END IF
END
```

Instrumented code

```
      send_id = mon_pvmfsend(...)
   END IF
CALL stop_trace
CALL mon_term
CALL proc_end(3,0)
END
```

Figure 7 Instrumenting an END statement

If a label precedes the END statement, these calls are not inserted between the label and the END statement, but rather after the line preceding the END statement. This can result in the instrumented program hanging, because a GOTO statement elsewhere in the code goes directly to the END statement without executing these routines. This situation is illustrated in Figure 8.

Source code

```
         send_id = pvmfsend(...)
      END IF
10 END
```

Instrumented code

```
         send_id = mon_pvmfsend(...)
      END IF
   CALL stop_trace
   CALL mon_term
   CALL proc_end(3,0)
10 END
```

Figure 8 Faulty instrumentation of an END statement

To be safe, use labels with CONTINUE statements and have code continue on subsequent lines, as shown in Figure 9.

Source code

```
          send_id = pvmfsend(...)
       END IF
10  CONTINUE
    END
```

Instrumented code

```
       send_id = mon_pvmfsend(...)
    END IF
10  CONTINUE
    CALL stop_trace
    CALL mon_term
    CALL proc_end(3,0)
    END
```

Figure 9 Using a CONTINUE statement with a label

Limitations when compiling

The instrumentor inserts calls in such a way that may result in them being placed where they are never reached. For example, the three calls in the previous examples are inserted before both STOP and END statements. If a STOP precedes an END statement in the source code, the calls following the STOP are never instrumented.

Restrictions on Fortran DO loops

Restrictions on Fortran DO loops are:

- No intermediate CONTINUE statement can appear in a DO loop before the terminating CONTINUE statement.
- All DO loops must terminate with either a CONTINUE or an ENDDO statement.

The following DO loops do not instrument properly:

```
          DO 100 i=1,10
          j = j + 110
          CONTINUE
100       CONTINUE

          DO 100 i=1,10
100       j = j + 1
```

Information related to PVM/GSM

You should be aware of several items related to the interaction between CXtrace and PVM/GSM.

- If any processes within a PVM/GSM application are executed with instrumentation, all processes must be instrumented.
- All hosts in a PVM/GSM cluster must contain the name of the host running the master PVM daemon in their .rhosts file. To compile the trace files generated by remote PVM/GSM processes, the master CXtrace process must be able to run a remote shell to each remote host. Generally, this condition is met to satisfy PVM/GSM's requirement to be able to log in to remote machines to start PVM/GSM daemons. However, unlike PVM/GSM, CXtrace has no capability to prompt for a password if the .rhosts entry does not exist.
- Processes in a CXtrace application that call `pvm_spawn` or are spawned by `pvm_spawn` must be instrumented.

Viewing trace files

4

The `tv` and `tally` programs allow you to examine data collected by the monitoring routines.

- `tv` provides a static view of the trace file that you can scroll and zoom both horizontally and vertically.
- `tally` collects and tabulates statistics that reflect the cumulative activity of the program.

The monitor library automatically sorts the trace file and places it into a default file called CXTRACE.OUT.*taskid* for use with the analysis tools (unless you specified another name in your .MONITOR file), where *taskid* is the task identifier (tid) of the process.

Using the trace view tool

`tv` provides a static view of trace file that you can scroll and zoom.

Invoking `tv`

To invoke `tv`, enter

```
% /opt/cxtrace/bin/tv tracefile
```

where *tracefile* is the sorted trace file (default name is CXTRACE.OUT.*taskid*). Refer to Figure 10 to see an example of the main `tv` window.

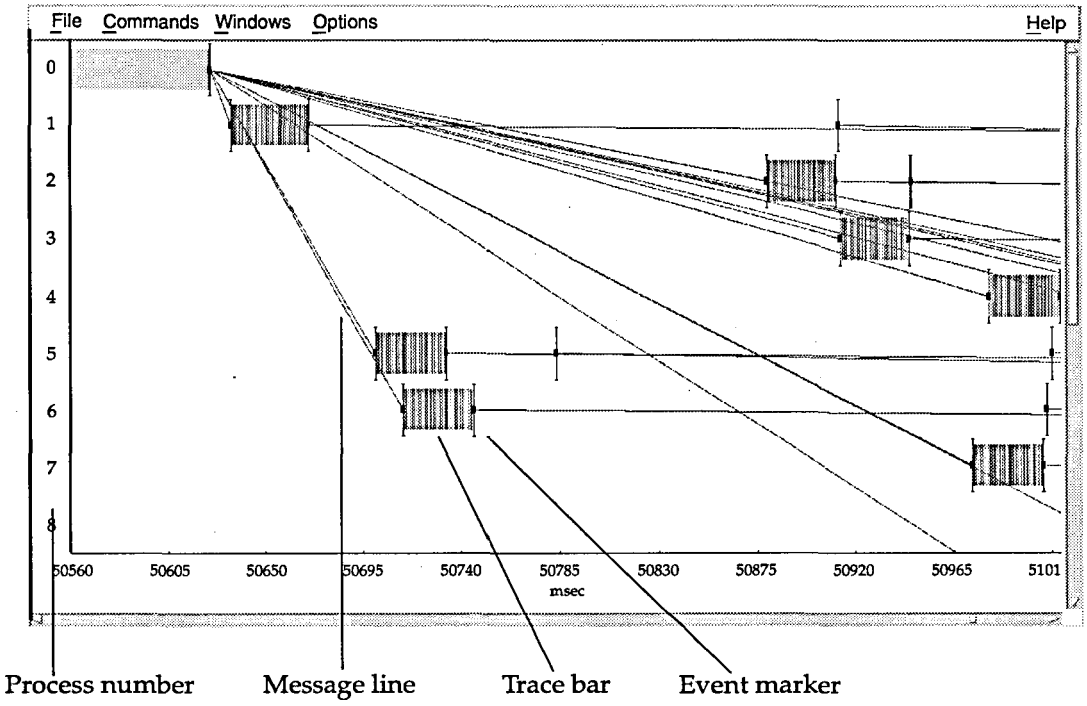


Figure 10 `tv` main window

Use the scrollbars to scroll through the view. The following sections discuss options for selecting messages and trace bars and the information received with both.

The colors indicated in the `tv` window have no significant meaning. Each color represents the execution of a different procedure. For example, if the color is assigned to subroutine `foo`, whenever `foo` is executing, the execution trace bar is blue.

The initial color assignment is arbitrary. You may change colors using the color editor (described in the following sections).

Selecting messages

To select a message, with the cursor on a message line, press the right mouse button for message options:

- Summary
- Sender Source
- Receiver Source
- Sender Construct
- Receiver Construct

The right mouse button brings up menu options and the left mouse button defaults to the first menu option, Summary.

Summary

The summary option shows the sending and receiving task, message type and size, transmission time, and throughput, as shown in Figure 11.

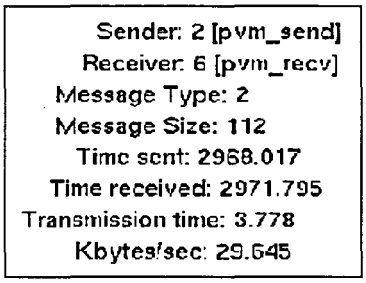


Figure 11 Summary dialog for messages

Sender Source

The sender source option displays source code that generated the message. The specific line of code is marked with an I-beam cursor in the left margin, as shown in Figure 12.

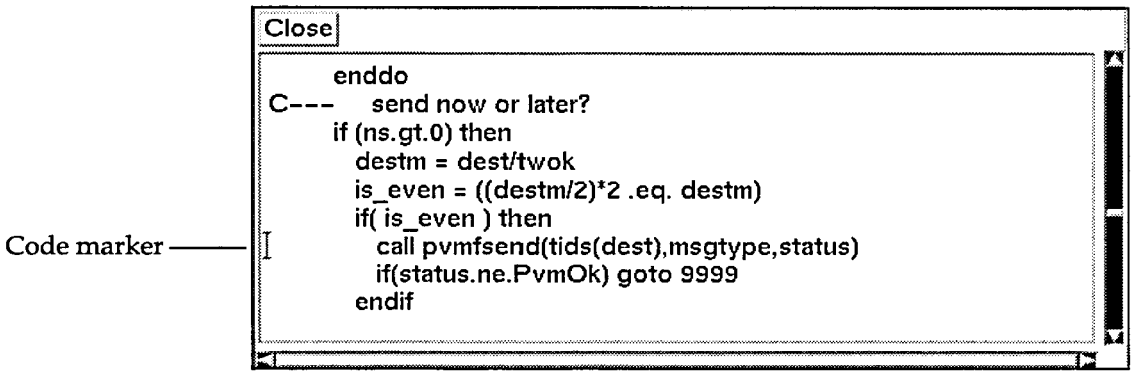


Figure 12 Source code for sender

Receiver Source

This option displays the source code that received the message. The specific line of code is marked with an I-beam cursor in the left margin. An example is shown in Figure 12.

Sender Construct

This option displays a construct tree with the sending construct's border highlighted. Clicking the right mouse button on a construct name displays its source code. See Figure 13.

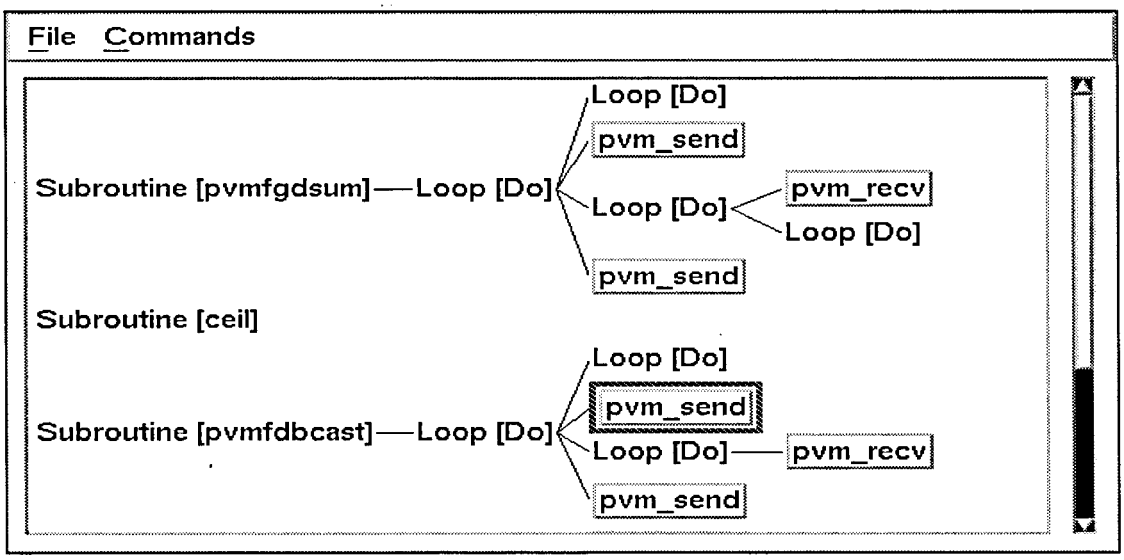


Figure 13 Construct tree for sender

To view groups of constructs, use the View Constructs dialog, available through the Commands menu. You view specific constructs (or all constructs) by highlighting the appropriate toggle, as shown in Figure 14.

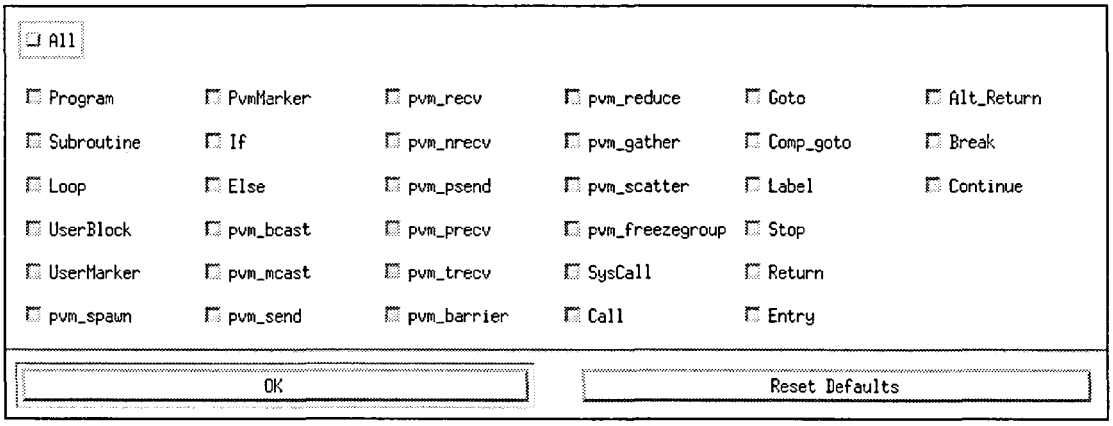


Figure 14 View Constructs dialog

Receiver Construct

This option displays a construct tree with the receiving construct's border highlighted. Clicking the right mouse button on a construct name displays its source code, as shown in Figure 12.

Selecting execution trace bars

With the cursor on a trace bar, press the right mouse button for message options:

- Summary
- Source
- Construct

The right mouse button brings up menu options and the left mouse button defaults to the first menu option (Summary).

Summary

This option shows the current procedure, state of execution, and time. If execution is blocked, the blocking call is reported. See Figure 15.

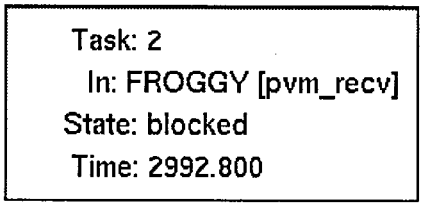


Figure 15 Summary dialog for execution trace

Source

This option displays source code of the currently executing procedure. If execution is blocked, the source displays the call on which the process is waiting.

Construct

This option displays a construct tree with the current construct's border highlighted. Clicking the right mouse button on a construct name displays its source code.

Color editor

CXtrace provides a color editor for modifying colors assigned to the various constructs of an application. You might find it useful to assign similar routines the same color or change colors to heighten contrast. Figure 16 shows the color editor control panel.

Invoke the color editor from a construct window. On a construct, press SHIFT-right mouse button to display the construct's color, then press CTRL-right mouse button to invoke the color editor.

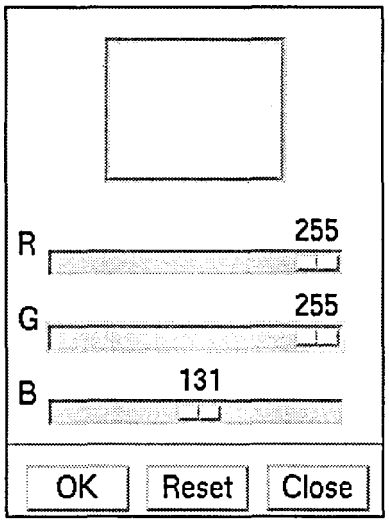


Figure 16 Color editor control panel

Using the left mouse button, click on a slider to modify the red, green, and blue values (range is 0 to 255). You can also click in the gray area of a slide bar to change it incrementally by 25. When you are finished, choose Reset to implement the change.

Not all colors may be changed. A warning box appears if:

- You attempt to change a fixed color.
- A construct does not have a color to change.

Zooming a view

From the main window, click on Command and select Zoom to choose the horizontal and vertical zooming options. If you change the settings and want to return to the default zoom, click on the appropriate reset button. Refer to Figure 17.

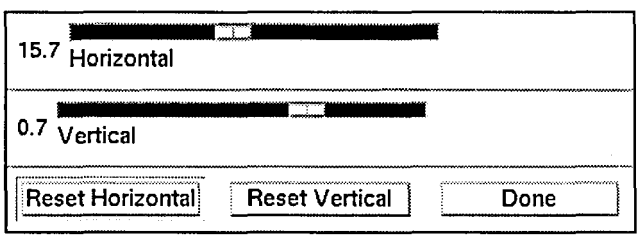


Figure 17 Zoom control panel

Viewing console information

Select Console from the Windows menu to open a text window. It displays summary information regarding your actions during your `tv` session. You might want to disable the Summary dialogs if you keep this window open. The information provided in both is similar. Figure 18 shows the Console window.

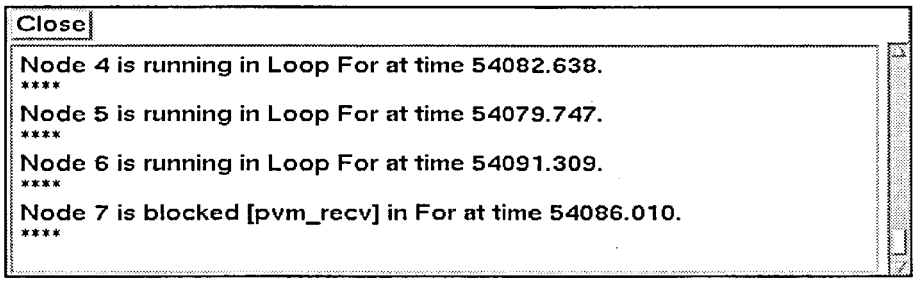


Figure 18 Console window

Choosing view options

The View submenu contains the following options:

Show Messages

Toggles display of message lines.

Show Message_Type

Toggles display of message type notations.

Show Event

Toggles display of event marks.

Show Summary Dialogs

Toggles display of dialogs that report summary information for messages and trace bars.

tally

The `tally` program generates a list of resource-use statistics on process-by-process and routine-by-routine bases. These statistics help identify inefficient sections of code, which can then be examined more closely with `tv`.

The only input to `tally` is a sorted trace file. `tally` uses the application database to relate identifiers in the trace file to constructs in the source program. `tally` sends its results to standard output as well as to the ncpu.summary and tally.summary files in the current directory.

Invoking `tally`

Invoke `tally` with the following command:

```
% /opt/cxtrace/bin/tally tracefile
```

where *tracefile* is the sorted trace file.

Defining the output

`tally` produces a file that contains several tables of statistics. Each row in the tables is a list of numbers or strings separated by tabs. Tables are preceded by a title and separated by a blank line.

tally.summary file

Table 2 presents data per function executing the program. Entries in this table are sorted by the lifetime of each routine.

Table 2 Routine summary table

Column	Meaning
Routine	Routine index and name of subroutine.
Lifetime	Amount of time taken to execute instructions in this function (excluding the functions called from this function).
Busy time	Amount of time function performed useful work (amount of time not spent in communication).
Communication index	Indicates function's contribution to the total communication time of the program. The lower this value, the lower the impact on the program of reducing this function's communication characteristics.
Global blocking	Amount of time routine spent in a global blocking operation.

Table 2 Routine summary table —**(continued)**

Column	Meaning
Send blocking	Amount of time routine spent in a send operation.
Receive blocking	Amount of time routine spent in a receive operation.
Percentage communication	Amount of total execution time routine spent in communication.

Table 3 provides information per process.

Table 3 Node summary table

Column	Meaning
Node	Number of process.
Percentage communication	Amount of total execution time processor spent in communication.
Lifetime	Amount of time spent executing the program.
Busy time	Amount of time spent not performing communication-related work.
Global blocking	Amount of time spent in a global blocking operation.
Send blocking	Amount of time spent in a send operation.
Receive blocking	Amount of time spent in a receive operation.

The remaining information provides statistics for routines that perform communications. Statistics are presented in individual tables, each having entries similar to the ones above.

NCPU.summary file

The first table provides the normalized CPU (NCPU) statistics. The NCPU for a given subroutine and a given *k* is the amount of CPU time used by that subroutine when *k* processors are busy, divided by *k*. If a subroutine spends a lot of time executing when only a few processes are busy, this may indicate that the routine inhibits parallelization and is a bottleneck.

The second table provides routine concurrency statistics. Routine concurrency indicates the amount of time spent by each subroutine when *k* copies were executing simultaneously. If a routine never has more than a few copies running simultaneously, it may indicate that the routine is inherently sequential. This

property differs from that of inhibiting parallelism for all subroutines, as described above with the NCPU statistics.

The next section contains sample output from `tally`.

Sample `tally` output

Example output from invoking `tally` is listed below.

```
ROUTINE SUMMARY
```

Routine	Busy Time	Global Blocking	Send Blocking	Recv Blocking	Life Time	% Commn	Comm. Index
1 transpose	1229.399	0.000	246.186	64.887	540.472	57.555	0.387
2 transpose_node	100.659	0.000	0.186	160.826	261.671	61.532	0.200
3 bin_to_dec	0.508	0.000	0.000	0.000	0.508	0.000	0.000
4 dec_to_bin	0.128	0.000	0.000	0.000	0.128	0.000	0.000
5 <rest..>	0.117	0.000	0.000	0.000	0.117	0.000	0.000

NODE SUMMARY

Node	Busy Time	Global Blocking	Send Blocking	Recv Blocking	Life Time	% Commn
0	37.443	0.000	34.241	28.154	99.838	62.496
1	39.927	0.000	35.575	28.342	103.844	61.550
2	39.560	0.000	24.994	33.395	97.949	59.611
3	44.796	0.000	29.530	27.495	101.821	56.005
4	39.289	0.000	30.399	28.216	97.904	59.869
...						

STATISTICS FOR ROUTINE transpose1

Node	Busy Time	Global Blocking	Send Blocking	Recv Blocking	Life Time	% Commn
0	24.754	0.000	34.220	8.191	67.165	63.144
1	27.263	0.000	35.555	8.232	71.050	61.628
2	26.861	0.000	24.967	13.341	65.169	58.782
3	32.139	0.000	29.509	7.381	69.029	53.441
4	26.605	0.000	30.374	8.156	65.135	59.154

...

STATISTICS FOR ROUTINE transpose_node

Node	Busy Time	Global Blocking	Send Blocking	Recv Blocking	Life Time	% Commn
0	12.599	0.000	0.021	19.963	32.583	61.332
1	12.573	0.000	0.020	20.110	32.703	61.553
2	12.605	0.000	0.027	20.054	32.686	61.436
3	12.562	0.000	0.021	20.114	32.697	61.580
4	12.589	0.000	0.025	20.060	32.674	61.470

...

NCPU STATISTIC

Routine	1	2	3	4	5	6	7	8
transpose1	5.342	13.957	7.038	9.436	5.806	8.677	5.325	2.363
transpose_node	0.289	0.017	0.091	0.110	0.063	0.062	0.042	12.329
bin_to_dec	0.000	0.012	0.029	0.032	0.014	0.015	0.008	0.005
<rest..>	0.004	0.003	0.001	0.001	0.001	0.001	0.004	0.006
dec_to_bin	0.000	0.000	0.000	0.000	0.005	0.001	0.004	0.007
<flush>	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

ROUTINE CONCURRENCY STATISTIC

Routine	1	2	3	4	5	6	7	8
transpose1	5.667	14.060	7.109	9.631	5.914	8.584	5.257	2.236
transpose_node	0.599	0.171	0.363	0.006	0.057	0.058	0.092	12.166
bin_to_dec	0.386	0.061	0.000	0.000	0.000	0.000	0.000	0.000
dec_to_bin	0.050	0.039	0.000	0.000	0.000	0.000	0.000	0.000
<rest..>	0.038	0.002	0.000	0.000	0.000	0.000	0.005	0.005
<flush>	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Using the CXtrace monitoring library

5

The runtime performance monitoring library, or monitor, contains a set of routines that function as event recorders by measuring the behavior of the instrumented program on a parallel machine. These event recorders are placed by the instrumentor into the instrumented code; they are responsible for generating the trace file that will be used by the analysis tools.

Figure 19 illustrates the basic function of event recorders during the monitoring phase. Event recorders write records into a buffer that is only intermittently written to disk. Because writing the buffer to disk is time consuming and can affect the execution pattern of the application, CXtrace allows you to specify the buffer size and empty the buffer by setting values in the .MONITOR initialization file.

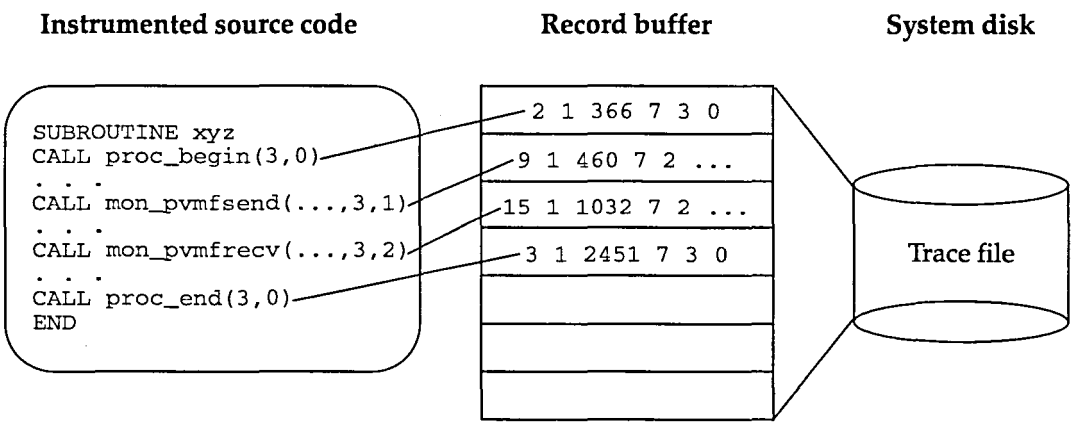


Figure 19 Inserted event recorders generating trace records

Parameters

Several parameters control the monitor's operation and are set in the .MONITOR initialization file. It must reside in the directory from which the instrumented application is run. All parameters have default settings and may appear in any order in the file. Refer to the MONITOR(5) man page.

The following sections discuss these parameters in detail.

TRACE_LEVEL

The TRACE_LEVEL parameter determines when the monitor generates trace records. A lower value produces fewer records. Four possible levels are possible:

Level 0

Generate TRACE_BEGIN, TRACE_END, BLOCK_BEGIN, BLOCK_END, and MARKER records.

Level 1

Generate level 0 records plus PROC_BEGIN and PROC_END records.

Level 2

Generate level 1 records plus GLOBAL_BEGIN and GLOBAL_END records.

Level 3

Generate level 2 records plus message-passing records (default).

With the default set at level 3, the monitor generates all records unless specified otherwise. Some records are generated even if you specify a negative level. Those records are: MON_BEGIN and MON_END. Refer to Appendix A, "Trace record information,"for the mapping of instrumented source code constructs to trace records to TRACE_LEVEL settings.

BUFFER_SIZE

The BUFFER_SIZE parameter sets the number of bytes to be allocated in RAM for the temporary storage of trace data. It is important not to set this value too low or too high for the application. If BUFFER_SIZE is very small, the monitor flushes data often and affects overall performance. If BUFFER_SIZE is very large, the monitor may compete with the application for memory resources. The default buffer size is 256 kbytes.

APPL_DB_FILE

The instrumentors encapsulate information about the structure of an application program into an application database file that is subsequently included in the trace file by the monitor. By default, the instrumentors store this information in a file named `APPL_DB` located in the directory containing the instrumented code. Use this parameter to specify a new name if you change the application database name.

FLUSH_MODE

The monitor provides two policies for flushing performance data from memory to disk:

- If `FLUSH_MODE` is nonzero (the default), a task flushes its buffer when full. Emptying buffers this way can cause irregularities in the program's execution, especially if the data buffer is small and flushing is slow.
- If `FLUSH_MODE` is 0, flushing is under user control. The task does not flush its buffer until you call `flush_trace` or the program terminates. The task stops collecting data until the buffer is emptied. This allows you to specify when data should be flushed but increases the chance of losing data if the buffer becomes full before calling `flush_trace`.

H_TRACE_FILE

This parameter specifies a trace file name and contains the events generated by the instrumented program. The default is `CXTRACE.OUT.`*taskid*, where *taskid* is the task identifier (tid) of the process.

PROFILE

The profile is a table of Boolean flags with an entry for each construct in the application. While the default is to monitor all instrumented constructs, you can select a subset by specifying a profile.

To specify a profile, follow these three steps.

Step 1 Load your modules and set them in `xinstrument`.

Step 2 Select Save in the Profile menu (the current profile is not created automatically).

Step 3 Enter a file name followed by a .pro extension.

For this option to work, the specified profile must reside in your home directory on all machines where processes might run. The .MONITOR file is not required, only the .pro file. If it does not exist, then the profile does not take effect for processes running on that host.

For the master process, the file must be located in the current working directory when the process is launched in order to be read correctly. If there are different host and node programs, you must save the profile while both modules are loaded into `xinstrument`.

The `TRACE_LEVEL` parameter takes precedence over a profile setting.

Example .MONITOR file

Lines in the .MONITOR file have the form:

parameter_name`:`*parameter_value*

Lines beginning with a pound sign (#) denote comments. Figure 20 contains an example .MONITOR file

```
#Amount of data to save before flushing
BUFFER_SIZE:32768
#FLUSH_MODE=0 -- Don't flush until I say so
#FLUSH_MODE=1 -- Flush when buffer fills up
FLUSH_MODE:1
#When to generate trace records
TRACE_LEVEL:0
#Name of trace file
H_TRACE_FILE:WORK.OUT
#Name of profile to use
PROFILE:flush01.pro
```

Figure 20 Example .MONITOR file

Trace record information

A

This appendix contains information about trace records. The first section shows trace records generated with monitor name routines. The second section lists trace records along with their identifiers, thresholds, and formats. MPI trace records are not covered in this appendix.

Trace records

Table 4 shows the trace records generated by various source code constructs, along with the names of the monitor routines that produce the trace records. Only constructs listed in the table produce trace records.

Many trace records are produced conditionally. Conditions are listed in the fifth column of the table.

Table 4 Generated trace records

Source code construct	Monitor event recorder	Trace record	Level	Conditions

Beginning of program	`mon_init`
	`start_trace`
	`fproc_begin`
	`proc_begin`
		`MON_BEGIN`
		`TRACE_BEGIN`
		`PROC_BEGIN`
		`PROC_BEGIN`
			
			0
			1
			1
				Only node 0 sends this.

End of program	`stop_trace` `mon_term`	`TRACE_END` `MON_END`	0	

`STOP` statement	`fproc_end`
	`proc_end`
	`stop_trace`
	`mon_term`
		`PROC_END`
		`PROC_END`
		`TRACE_END`
		`MON_END`
			1
			1
			0	

Return statement	`proc_end`	`PROC_END`	1	
Call to `begin_trace`	`start_trace`	`TRACE_BEGIN`	0	

Table 4 Generated trace records (continued)

Source code construct	Monitor event recorder	Trace record	Level	Conditions
Call to `end_trace`	`stop_trace`	`TRACE_END`	0	
Beginning of function or subroutine	`proc_begin`	`PROC_BEGIN`	1	
End of function or subroutine	`proc_end`	`PROC_END`	1	
Call to `begin_block`	`block_begin`	`BLOCK_BEGIN`	0	
Call to `end_block`	`block_end`	`BLOCK_END`	0	
Program loops: `DO, DO WHILE` in Fortran `for, while, do` in C	`block_begin` `block_end`	`BLOCK_BEGIN` `BLOCK_END`	0	
Call to `insert_marker`	`point_marker`	`MARKER`	0	
Call to `pvm_barrier`, `pvmfbarrier`, `pvm_freezegroup`, or `pvmffreezegroup`	`global_start` `global_end`	`GLOBAL_BEGIN` `GLOBAL_END`	2 2	
Call to `pvm_send` or `pvmfsend`	`mon_pvm_send` `mon_pvmfsend`	`SYNC_SEND_BLK` `SYNC_SEND_UNBLK`	3 3	
Call to `pvm_recv` or `pvmfrecv`	`mon_pvm_recv` `mon_pvmfrecv`	`SYNC_RECV_BLK` `SYNC_RECV_UNBLK`	3 3	

Call to `pvm_nrecv` or `pvmfnrecv`	`mon_pvm_nrecv`
	`mon_pvmfnrecv`
		`PVM_PROBE`
		
		`ASYNC_RECV`
			3
			
			3
				If no message available.
				
				If message received.

Call to `flush_trace`	`flushtrace`	N/A		

Table 4 Generated trace records **(continued)**

Source code construct	Monitor event recorder	Trace record	Level	Conditions
Calls to pvm_addhosts pvmfaddhost pvm_delhosts pvmfdelhost pvm_joingroup pvmfjoingroup pvm_kill pvmfkill pvm_lvgroup pvmflvgroup pvm_mytid pvmfmytid pvm_sendsig pvmfsendsig	point_marker ("PVM Marker")	MARKER		
Call to pvm_probe or pvmfprobe	mon_pvm_probe mon_pvmfprobe	PVM_PROBE	3	
Call to pvm_spawn or pvmfspawn	mon_pvm_spawn mon_pvmfspawn	MARKER		Not user selectable.

Explaining trace records

This section contains the following information on trace records:

- A list of records with numerical identifier, threshold, and format type.
- The format of each record.
- The meaning of each record.

List of records

Table 5 lists trace records. In the table:

- The first column lists record names.
- The second column shows the numerical identifier. This number appears in the trace file instead of the record name.
- The third column indicates how large the `TRACE_LEVEL` parameter must be in order for the record to be generated. Those records produced regardless of the value have no entry.
- The fourth column indicates the format of the record. Format key is

  S
  : Short format

  C
  : Code block format

  M
  : Message format

  SM
  : Short message format

Table 5 List of trace records

Name	ID	Level	Format
TRACE_BEGIN	0	0	S
TRACE_END	1	0	S
PROC_BEGIN	2	1	C
PROC_END	3	1	C
BLOCK_BEGIN	4	0	C
BLOCK_END	5	0	C
MARKER	6	0	C
GLOBAL_BEGIN	7	2	C
GLOBAL_END	8	2	C
SYNC_SEND_BLK	10	3	M
SYNC_SEND_UNBLK	11	3	SM
ASYNC_RECV	15	3	M
SYNC_RECV_BLK	16	3	SM
SYNC_RECV_UNBLK	17	3	M
MON_BEGIN	20		C
MON_END	21		S
PVM_PROBE	28	3	S

Record format

There are several trace record formats. Each consists of several fields printed on one line in the trace file.

Short format

Short format consists of four fields:

- Trace record identifier
- Time of event (seconds)
- Time of event (microseconds)
- Process where event occurred

Code block format

Code block format consists of the fields in the short format plus the following fields:

- File identifier
- Object identifier

Two identifiers are used with the application database to relate trace file events to source code constructs.

Message format

Message format consists of the fields in the short format plus the following fields:

- Other process participating in message
- Type of message
- Size of message
- File identifier
- Object identifier

File and object identifiers are used with the application database to relate trace file events to source code constructs.

Short message format

Short message format consists of the fields in the short format plus the following fields:

- File identifier
- Object identifier

File and object identifiers are used with the application database to relate trace file events to source code constructs.

Meaning of records

The analysis tools read the trace file to find out when a process began a new code block, when it was blocked, when it sent a message, and when it received a message.

Entered block

Entered block trace records indicate that a process is entering (or reentering) a different code block: `PROC_BEGIN`, `PROC_END`, `BLOCK_BEGIN`, and `BLOCK_END`.

Started blocking

Started blocking trace records indicate that a process has started blocking: `GLOBAL_BEGIN`, `SYNC_SEND_BLK`, and `SYNC_RECV_BLK`.

Finished blocking

Finished blocking trace records indicate that a process has finished blocking: `GLOBAL_END`, `SYNC_SEND_UNBLK`, and `SYNC_RECV_UNBLK`.

Sent message

Sent message trace record indicates that a process has sent a message: `SYNC_SEND_BLK`.

Received message

Received message trace records indicate that a process has received a message: `ASYNC_RECV` and `SYNC_RECV_UNBLK`.

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

U

V

W

X

B5639-90003