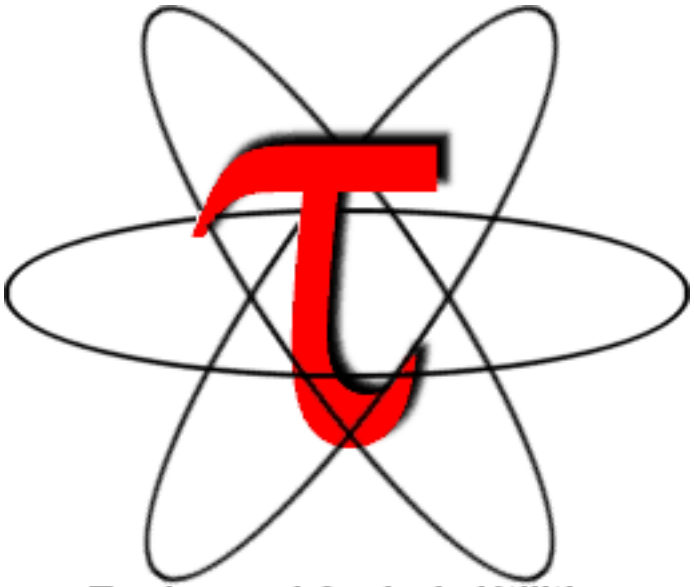
TAU User's Guide



TAU User's Guide

version 2.13

Department of Computer and Information Science,
University of Oregon, OR
Los Alamos National Laboratory, NM
Research Centre Jülich, ZAM, Germany

http://www.cs.uoregon.edu/research/paracomp/tau

Copyright © 1997-2004

Department of Computer and Information Science, University of Oregon
Advanced Computing Laboratory, LANL, NM
Research Centre Jülich, ZAM, Germany

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of University of Oregon (UO) Research Centre Jülich, (ZAM) and Los Alamos National Laboratory (LANL) not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. The University of Oregon, ZAM and LANL make no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

UO, ZAM AND LANL DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE UNIVERSITY OF OREGON, ZAM OR LANL BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Copyright © 2004.

All rights reserved.

CHAPTER 1 *Installation* *1*

Installing TAU ----------2
Examples: ----------11
Platforms Supported ----------13
Software Requirements ----------14

CHAPTER 2 *Compiling* *17*

TAU Stub Makefile ----------18
Enabling and Disabling the Instrumentation ---20
Using TAU with MPI ----------21
Environment Variables ----------21
Application Scenarios ----------22

CHAPTER 3 *Instrumentation* *27*

Automatic Instrumentation of C++, C and F90 source code ----------28
Using TAU with PDT and MPI ----------31
Using TAU with PDT for an F90 MPI app. ----32
Using TAU with PDT and Opari ----------34
Selective Instrumentation ----------35
TAU_REDUCE: A tool for reducing instrumentation overhead ----------37
C++ Measurement API ----------39
TAU Mapping API ----------62
C Measurement API ----------67
Fortran90 Measurement API ----------68
Summary ----------75

CHAPTER 4 *Profiling* *77*

Running the application ----------78
Running an application using DynInstAPI -----78
Using Hardware Performance Counters ------79

Using Multiple Hardware Counters for Measurement ----------86
Running a JAVA application with TAU -------87
Running a Python application with TAU ------88
pprof ----------89
paraprof ----------91

CHAPTER 5 *Tracing* *101*

Generating Event Traces ----------102
Vampir: Visualizing TAU traces ----------103

CHAPTER 6 *Performance Database* *115*

Prerequisites ----------116
Installation ----------116

CHAPTER 7 *Summary* *121*

Software Availability ----------122
Acknowledgments ----------122

CHAPTER 8 *Appendix: Configuration Issues* *123*

Instructions for Installing TAU under Windows **124**

CHAPTER 9 *References* *127*

URLs ----------127

CHAPTER 1 *Installation*

TAU (Tuning and Analysis Utilities) is a portable profiling and tracing toolkit for performance analysis of parallel programs written in Java, C++, C, and Fortran. The model that TAU uses to profile parallel, multi-threaded programs maintains performance data for each thread, context, and node in use by an application. The profiling instrumentation needed to implement the model captures data for functions, methods, basic blocks, and statement execution at these levels. All C++ language features are supported in the TAU profiling instrumentation including templates and namespaces, which is available through an API at the library or application level. The API also provides selection of profiling groups for organizing and controlling instrumentation. The instrumentation can be inserted in the source code using an automatic instrumentor tool based on the Program Database Toolkit (PDT), dynamically using DyninstAPI, at runtime in the Java virtual machine, or manually using the instrumentation API.

TAU's profile visualization tool, paraprof, provides graphical displays of all the performance analysis results, in aggregate and single node/context/thread forms. The user can quickly identify sources of performance bottlenecks in the application using the graphical interface. In addition, TAU can generate event traces that can be displayed with the Vampir or Paraver trace visualization tools.

This chapter discusses installation of the TAU portable profiling package.

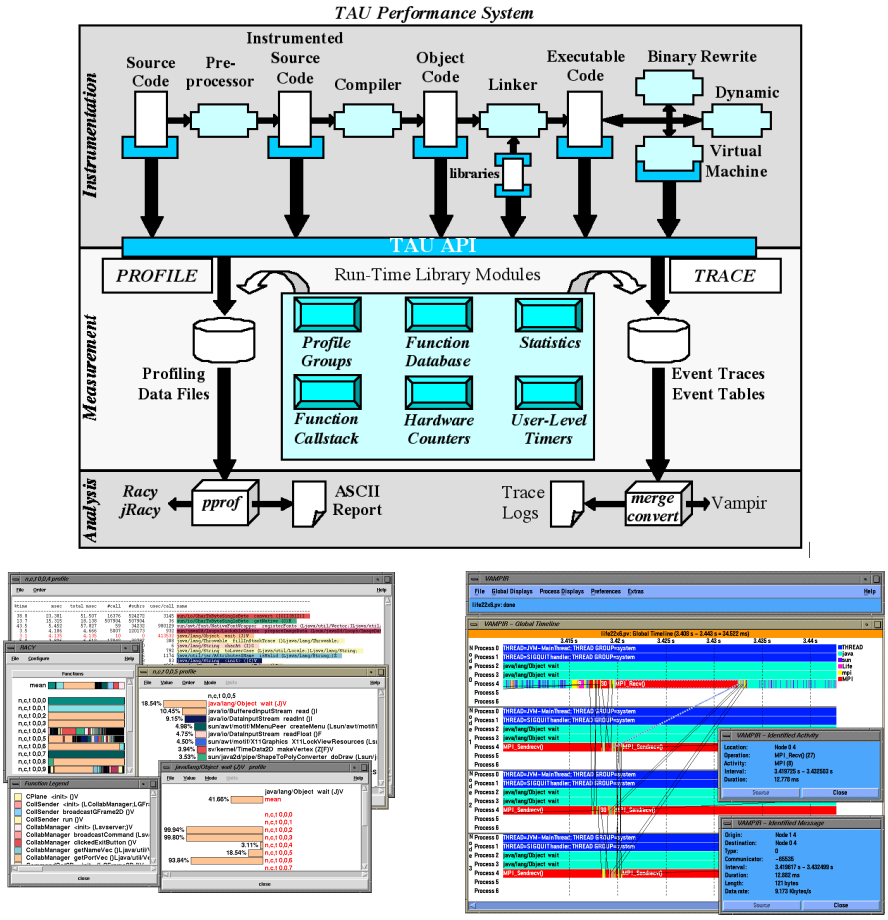


FIGURE 1. Architecture of TAU

Installing TAU

After uncompressing and untarring TAU, the user needs to configure, compile and install the package. This is done by invoking:

```
%./configure
% make install
```

TAU is configured by running the **configure** script with appropriate options that select the profiling and tracing components that are used to build the TAU library. The `configure' shell script attempts to guess correct values for various system-dependent variables used during compilation, and creates the Makefile(s) (one in each subdirectory of the source directory).

The following command-line options are available to configure:

-prefix=<directory>

Specifies the destination directory where the header, library and binary files are copied. By default, these are copied to subdirectories <arch>/bin and <arch>/lib in the TAU root directory.

-arch=<architecture>

Specifies the architecture. If the user does not specify this option, configure determines the architecture. For SGI, the user can specify either of sgi32, sgin32 or sgi64 for 32, n32 or 64 bit compilation modes respectively. The files are installed in the <architecture>/bin and <architecture>/lib directories.

-c++=<C++ compiler>

Specifies the name of the C++ compiler. Supported C++ compilers include KCC (from KAI/Intel), CC (SGI, Sun), g++ (from GNU), FCC (from Fujitsu), xlC (from IBM), guidec++ (from KAI/Intel), cxx (Tru64) and aCC (from HP), c++ (from Apple), icpc and ecpc (from Intel) and pgCC (from PGI).

-cc=<C Compiler>

Specifies the name of the C compiler. Supported C compilers include cc, gcc (from GNU), pgcc (from PGI), fcc (from Fujitsu), xlc (from IBM), and KCC (from KAI/Intel), icc and ecc (from Intel).

-pdt_c++=<C++ Compiler>

Specifies a different C++ compiler for PDT (tau_instrumentor). This is typically used when the library is compiled with a C++ compiler (specified with -c++) and the tau_instrumentor is compiled with a different <pdt_c++> compiler. For e.g.,

```
-c++=pgCC -cc=pgcc -pdt_c++=KCC -openmp ...
```

uses PGI's OpenMP compilers for TAU's library and KCC for tau_instrumentor.

-fortran=<Fortran Compiler>

Specifies the name of the Fortran90 compiler. Valid options are: gnu, sgi, ibm, ibm64, hp, cray, pgi, absoft, fujitsu, sun, kai, nec, hitachi, compaq, and intel.

-pthread

Specifies pthread as the thread package to be used. In the default mode, no thread package is used.

-tulipthread=<directory> -smarts

Specifies SMARTS (Shared Memory Asynchronous Runtime System) as the threads package to be used. <directory> gives the location of the SMARTS root directory. [SMARTS-URL]

-openmp

Specifies OpenMP as the threads package to be used.[OPENMP-URL]

-opari=<dir>

Specifies the location of the Opari OpenMP directive rewriting tool. The use of Opari source-to-source instrumentor in conjunction with TAU exposes OpenMP events for instrumentation. See examples/opari directory. [OPARI-URL] Note: There are two versions of Opari: standalone - (opari-pomp-1.1.tar.gz) and the newer KOJAK - kojak-<ver>.tar.gz opari/ directory. Please upgrade to the KOJAK version (especially if you're using IBM xlf90) and specify -opari=<kojak-dir>/opari while configuring TAU.

-opari_region

Report performance data for only OpenMP regions and not constructs. By default, both regions and constructs are profiled with Opari.

-opari_construct

Report performance data for only OpenMP constructs and not regions. By default, both regions and constructs are profiled with Opari.

-pdt=<directory>

Specifies the location of the installed PDT (Program Database Toolkit) root directory. PDT is used to build **tau_instrumentor**, a C++, C and F90 instrumentation program that automatically inserts TAU annotations in the source code [PDT-URL]. If PDT is configured with a subdirectory option (-compdir=<opt>) then TAU can be configured with the same option by specifying

```
-pdt=<dir> -pdtcompdir=<opt>.
```

-pcl=<directory>

Specifies the location of the installed PCL (Performance Counter Library) root directory. PCL provides a common interface to access hardware performance counters on modern microprocessors. The library supports Sun UltraSparc I/II, PowerPC 604e under AIX, MIPS R10000/12000 under IRIX, Compaq Alpha 21164, 21264 under Tru64Unix and Cray Unicos (T3E) and the Intel Pentium family of microprocessors under Linux. This option specifies the use of hardware performance counters for profiling (instead of time). To measure floating point instructions, set the environment variable **PCL_EVENT** to **PCL_FP_INSTR** (for example). See the section “Using Hardware Performance Counters” in Chapter 4 for details regarding its usage. [PCL-URL]

-papi=<directory>

Specifies the location of the installed PAPI (Performance Data Standard and API) root directory. PCL provides a common interface to access hardware performance counters and timers on modern microprocessors. Most modern CPUs provide on-chip hardware performance counters that can record several events such as the number of instructions issued, floating point operations performed, the number of primary and secondary data and instruction cache misses, etc. To measure floating point instructions, set the environment variable **PAPI_EVENT** to **PAPI_FP_INS**

(for example). This option (by default) specifies the use of hardware performance counters for profiling (instead of time). When used in conjunction with **-PAPIWALLCLOCK** or **-PAPIVIRTUAL**, it specifies the use of wallclock or virtual process timers respectively. See the section "Using Hardware Performance Counters" in Chapter 4 for details regarding its usage. [PAPI-URL]

-PAPIWALLCLOCK

When used in conjunction with the -papi=<dir> option, this option allows TAU to use high resolution, low overhead CPU timers for wallclock time based measurements. This can reduce the TAU overhead for accessing wallclock time for profile and trace measurements. (See NOTE below.)

-PAPIVIRTUAL

When used in conjunction with the -papi=<dir> option, this option allows TAU to use the process virtual time (time spent in the "user" mode) for profile measurements, instead of the default wallclock time. (See NOTE below.)

-CPUTIME

Specifies the use of user+ system time (collectively CPU time) for profile measurements, instead of the default wallclock time. This may be used with multi-threaded programs only under the LINUX operating system which provides bound threads. On other platforms, this option may be used for profiling single-threaded programs only.

-MULTIPLECOUNTERS

Allows TAU to track more than one quantity (multiple hardware counters, CPU-time, wallclock time, etc.) Configure with other options such as -papi=<dir>, -pcl=<dir>, -LINUXTIMERS, -SGITIMERS, -CPUTIME, -PAPIVIRTUAL, etc. See Section "Using Multiple Hardware Counters" in Chapter 4 for detailed instructions on setting the environment variables **COUNTER<1-25>** for this option. If **-MULTIPLECOUNTERS** is used with the **-TRACE** option, tracing employs the **COUNTER1** environment variable for wallclock time.

NOTE: The default measurement option in TAU is to use the wallclock time, which is the total time a program takes to execute, including the time when it is waiting for resources. It is the time measured from a real-time clock. The process virtual time

(**-PAPIVIRTUAL**) is the time spent when the process is actually running. It does not include the time spent when the process is swapped out waiting for CPU or other resources and it does not include the time spent on behalf of the operating system (for executing a system call, for instance). It is the time spent in the "user" mode. The **CPUTIME** on the other hand, includes both the time the process is running (process virtual time) and the time the system is providing services for it (such as executing a system call). It is the sum of the process virtual (user) time and the system time (See *man getrusage()*).

-jdk=<directory>

Specifies the location of the installed Java 2 Development Kit (JDK1.2+) root directory. TAU can profile or trace Java applications without any modifications to the source code, byte-code or the Java virtual machine. See README.JAVA on instructions on using TAU with Java 2 applications. This option should only be used for configuring TAU to use JVMPI for profiling and tracing of Java applications. It should not be used for configuring paraprof, which uses java from the user's path.

-dyninst=<dir>

Specifies the directory where the DynInst dynamic instrumentation package is installed. Using DynInst, a user can invoke **tau_run** to instrument an executable program at runtime or prior to execution by rewriting it. [DYNINST-URL][PARADYN-URL].

-mpiinc=<dir>

Specifies the directory where MPI header files reside (such as mpi.h and mpif.h). This option also generates the TAU MPI wrapper library that instruments MPI routines using the MPI Profiling Interface. See the examples/NPB2.3/config/make.def file for its usage with Fortran and MPI programs. [MPI-URL]

-mpilib=<dir>

Specifies the directory where MPI library files reside. This option should be used in conjunction with the -mpiinc=<dir> option to generate the TAU MPI wrapper library.

-mpilibrary=<lib>

Specifies the use of a different MPI library. By default, TAU uses -lmpi or -lmpich as the MPI library. This option allows the user to specify another library. e.g., -mpilibrary=-lmpi_r for specifying a thread-safe MPI library.

-nocomm

Allows the user to turn off tracking of messages (synchronous/asynchronous) in TAU's MPI wrapper interposition library. Entry and exit events for MPI routines are still tracked. Affects both profiling and tracing.

-epilog=<dir>

Specifies the directory where the EPILOG tracing package [EPILOG-URL] is installed.This option should be used in conjunction with the -TRACE option to generate binary EPILOG traces (instead of binary TAU traces). EPILOG traces can then be used with other tools such as EXPERT. EPILOG comes with its own implementation of the MPI wrapper library and the POMP library used with Opari. Using option overrides TAU's libraries for MPI, and OpenMP.

-pythoninc=<dir>

Specifies the location of the Python include directory. This is the directory where Python.h header file is located. This option enables python bindings to be generated. The user should set the environment variable PYTHONPATH to <TAUROOT>/<ARCH>/lib/bindings-<options> to use a specific version of the TAU Python bindings. By importing package pytau, a user can manually instrument the source code and use the TAU API. On the other hand, by importing tau and using tau.run('<func>'), TAU can automatically generate instrumentation. See examples/python directory for further information.

-pythonlib=<dir>

Specifies the location of the Python lib directory. This is the directory where *.py and *.pyc files (and config directory) are located. This option is mandatory for IBM when Python bindings are used. For other systems, this option may not be specified (but -pythoninc=<dir> needs to be specified).

-PROFILE

This is the default option; it specifies summary profile files to be generated at the end of execution. Profiling generates aggregate statistics (such as the total time spent in routines and statements), and can be used in conjunction with the profile browser **racy** to analyze the performance. Wallclock time is used for profiling program entities.

-PROFILECALLPATH

This option generates call path profiles which shows the time spent in a routine when it is called by another routine in the calling path. “a => b” stands for the time spent in routine “b” when it is invoked by routine “a”. This option is an extension of -PROFILE, the default profiling option. Specifying TAU_CALLPATH_DEPTH environment variable, the user can vary the depth of the callpath. See examples/calltree for further information.

-PROFILESTATS

Specifies the calculation of additional statistics, such as the standard deviation of the exclusive time/counts spent in each profiled block. This option is an extension of -PROFILE, the default profiling option.

-PROFILECOUNTERS

Specifies use of hardware performance counters for profiling under IRIX using the SGI R10000 perfex counter access interface. The use of this option is deprecated in favor of the -pcl=<dir> and -papi=<dir> options described above.

-SGITIMERS

Specifies use of the free running nanosecond resolution on-chip timer on the R10000+. This timer has a lower overhead than the default timer on SGI, and is recommended for SGIs (similar to the -papi=<dir> -PAPIWALLCLOCK options).

-CRAYTIMERS

Specifies use of the free running nanosecond resolution on-chip timer on the CRAY X1 cpu (accessed by the rtc() syscall). This timer has a significantly lower overhead than the default timer on the X1, and is recommended for profiling. Since this timer is not synchronized across different cpus, this option should not be used with the -

TRACE option for tracing a multi-cpu application, where a globally synchronized realtime clock is required.

-LINUXTIMERS

Specifies the use of the free running nanosecond resolution time stamp counter (TSC) on Pentium III+ and Itanium family of processors under Linux. This timer has a lower overhead than the default time and is recommended.

-TRACE

Generates event-trace logs, rather than summary profiles. Traces show when and where an event occurred, in terms of the location in the source code and the process that executed it. Traces can be merged and converted using **tau_merge** and **tau_convert** utilities respectively, and visualized using Vampir, a commercial trace visualization tool. [VAMPIR-URL]

-muse

Specifies the use of MAGNET/MUSE to extract low-level information from the kernel. To use this configuration, Linux kernel has to be patched with MAGNET and MUSE has to be install on the executing machine. Also, magnetd has to be running with the appropriate handlers and filters installed. User can specify package by setting the environment variable TAU_MUSE_PACKAGE. [MUSE-URL]

-noex

Specifies that no exceptions be used while compiling the library. This is relevant for C++.

-useropt=<options-list>

Specifies additional user options such as -g or -I. For multiple options, the options list should be enclosed in a single quote. For example

```
%./configure –useropt='-g –I/usr/local/stl'
```

-help

Lists all the available configure options and quits.

To install ***multiple*** (typical) configurations of TAU at a site, you may use the script 'installtau'. It takes options similar to those described above. It invokes ./configure <opts>; make clean install; to create multiple libraries that may be requested by the users at a site. The installtau script accepts the following options:

```
% installtau -help

TAU Configuration Utility
************************************************
Usage: installtau [OPTIONS]
  where [OPTIONS] are:
-arch=<arch>
-fortran=<compiler>
-cc=<compiler>
-c++=<compiler>
-useropt=<options>
-pdt=<pdtdir>
-papi=<papidir>
-mpiinc=<mpiincdir>
-mpilib=<mpilibdir>
-mpilibrary=<mpilibrary>
-opari=<oparidir>

****************************************************************
```

These options are similar to the options used by the configure script.

Examples:

(See Appendix for POOMA & Windows installation instructions)

a) Install TAU using KCC on SGI, with trace and profile options:

```
%./configure -c++=KCC -SGITIMERS -arch=sgi64 -TRACE
     -PROFILE -prefix=/usr/local/packages/tau
```

b) Installing TAU with Java

```
%./configure -c++=g++ -jdk=/usr/local/packages/jdk1.4
% make install
% set path=($path <taudir>/<tauarch>/bin)
% setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:<taudir>/
<tauarch>/lib
% cd examples/java/pi
% java -XrunTAU Pi 200000
% racy
```

c) Use TAU with KCC, and cc on 64 bit SGI systems and use MPI wrapper libraries with SGI's low cost timers and use PDT for automated source code instrumentation. Enable both profiling and tracing.

```
% ./configure -c++=KCC -cc=cc -arch=sgi64 -mpiinc=/
    local/apps/mpich/include -mpilib=/local/apps/mpich/
    lib/IRIX64/ch_p4 -SGITIMERS -pdt=/local/apps/pdt
```

d) Use OpenMP+MPI using KAI's Guide compiler suite, Opari for OpenMP instrumentation and use PAPI for accessing hardware performance counters for profile based measurements.

```
% ./configure -c++=guidec++ -cc=guidec -papi=/usr/
    local/packages/papi -openmp     -mpiinc=/usr/pack-
    ages/mpich/include -mpilib=/usr/packages/mpich/lib
    -opari=/usr/local/opari
```

e) Use CPUTIME measurements for a multi-threaded application using pthreads under LINUX.

```
% configure -pthread -CPUTIME
```

f) Use multiple hardware performance counters

```
% configure -MULTIPLECOUNTERS -papi=/usr/local/papi -
    PAPIWALLCLOCK -PAPIVIRTUAL -LINUXTIMERS -mpiinc=/
    usr/local/mpich/include -mpilib=/usr/local/mpich/
    lib/ -pdt=/usr/local/pdtoolkit -useropt=-O2
% setenv COUNTER1 LINUX_TIMERS
```

```
% setenv COUNTER2 PAPI_FP_INS
% setenv COUNTER3 PAPI_L1_DCM ...
```

NOTE: Also see Section “Application Scenarios” in Chapter 2 (Compiling) for an explanation of simple examples that are included with the TAU distribution.

Platforms Supported

TAU has been tested on the following platforms:

1. SGI

On IRIX 6.x based systems, including Indy, Power Challenge, Onyx, Onyx2 and Origin 200, 2000, 3000 Series, CC 7.2+, KAI [KAI-URL] KCC and g++/egcs [GNU-URL] compilers are supported.

2. LINUX Clusters

On Linux based Intel x86 PC clusters, KAI/Intel’s KCC, g++, egcs (GNU), pgCC (PGI) [PGI-URL], FCC (Fujitsu) [FUJITSU-URL] and icpc/ecpc Intel compilers have been tested. TAU also runs under IA-64, Opteron, PowerPC, Alpha, Apple PowerMac, Sparc and other processors running Linux.

3. Sun Solaris

Sun Workshop Pro 5.0 compilers (CC, F90), KAI KCC, KAP/Pro and GNU g++ work with TAU.

4. IBM AIX

On IBM SP2 and AIX systems, KAI KCC, KAP/Pro, IBM xlC, xlc, xlf90 and g++ compilers work with TAU.

5. HP HP-UX

On HP PA-RISC systems, aCC and g++ can be used.

6. HP Alpha Tru64

On HP Alpha Tru64 machines, cxx and g++, and Guide compilers may be used with TAU.

7. NEC SX series vector machines

On NEC SX-5 systems, NEC c++ may be used with TAU.

8. Cray X1, T3E, SV-1

On Cray T3E systems, KAI KCC and Cray CC compilers have been tested with TAU. On Cray SV-1 and X1 systems, Cray CC compilers have been tested with TAU.

9. Hitachi SR8000 vector machines

On Hitachi machines, Hitachi KCC, g++ and Hitachi cc compilers may be used with TAU.

10. Apple OS X

On Apple OS X machines, c++ or g++ may be used to compile TAU. Also, IBM's xlf90, xlf compilers for G4/G5 may be used with TAU.

11. Microsoft Windows

On Windows, Microsoft Visual C++ 6.0 or higher and JDK 1.2+ compilers have been tested with TAU.

NOTE: TAU has been tested with JDK 1.2, 1.3, 1.4.x under Solaris, SGI, IBM, Linux, and MacOS X.

Software Requirements

1. Java v 1.2

TAU's GUI **paraprof** requires Java v1.2 or better in your path. We recommend Java version 1.4x from Sun. An older Tcl/Tk based browser **racy** is also included with TAU for compatibility. It requires the executable **wish** to be in your path. racy is also available in this distribution but support for racy will be gradually phased out. Users are encouraged to use paraprof instead. Paraprof does *not* require -

jdk=<dir> option to be specified (which is used for configuring TAU for analyzing Java applications). The **java** program should be in the user's path.

CHAPTER 2 *Compiling*

Source-based instrumentation with TAU measurement code requires compilation. At compile time, the TAU system provides several options and configuration alternatives. This chapter explains compilation options to enable profiling or tracing.

TAU Stub Makefile

TAU configuration generates a Makefile stub as well as a library. The Makefile name has the form `Makefile.tau-<options>`, the library name the form `libtau-<options>.a`. For example,

```
%./configure -TRACE -c++=KCC -arch=sgin32
```

generates

```
Makefile.tau-trace-kcc libtau-trace-kcc.a
```

in `tau-2.x/sgin32/lib`

Using different configuration options, several modular libraries can be built and co-exist even in the same architecture. To choose a particular version of the library, the corresponding Makefile stub must be included in the application Makefile. The stub Makefile defines the following variables:

- `TAU_CXX` for the C++ compiler
- `TAU_CC` for the C compiler
- `TAU_F90` for the F90 compiler
- `TAU_LINKER` for the linker
- `TAU_INCLUDE` for the include directories
- `TAU_DEFS` for the defines on the command-line
- `TAU_LIBS` for the TAU static library
- `TAU_SHLIBS` for the TAU shared object (dynamic library)
- `TAU_MPI_INCLUDE` for the directory where MPI header files reside
- `TAU_MPI_LIBS` for the TAU MPI library with the MPI libraries for C/C++
- `TAU_MPI_FLIBS` for the TAU MPI library with MPI libraries for Fortran
- `TAU_FORTRANLIBS` for additional fortran libraries for linking with C++
- `TAU_CXXLIBS` for linking with C++ libraries when native f90 linker is used
- `TAU_TRACE_INPUT_LIB` for linking with the TAU trace reader library to process binary TAU traces (typically used for making a trace converter).
- `TAU_DISABLE` for the default TAU stub library for Fortran, and
- `USER_OPT` for any user defined options specified during configuration

In addition to these options, the stub makefile also contains information about other packages configured with TAU. The stub makefile defines the following variables:

- `PDTDIR` for the location of the PDT root directory
- `OPARIDIR` for the location of the Opari root directory
- `TULIPDIR` for the location of the Tulip root directory
- `PCLDIR` for the location of the PCL root directory
- `PAPIDIR` for the location of the PAPI root directory
- `EPILOGDIR` for the location of the EPILOG root directory
- `JDKDIR` for the location of the JDK root directory
- `DYNINSTDIR` for the location of the DyninstAPI root directory

It should be noted that the TAU library is written in C++. It may be linked with a Fortran or a C object file in two ways. Either the TAU_LINKER (typically C++ compiler) may be used or the native linker (C, F90 compiler) may be used. For Fortran programs that use the C++ linker, the TAU_FORTRANLIBS macro contains additional Fortran libraries that need to be linked in to create the executable. If the F90 linker is used, TAU_CXXLIBS should be added to the link line which links in the necessary C++ libraries.

A typical makefile that uses these Makefile variables is shown below:

TAUROOTDIR = /usr/local/packages/tau-2.x

```
include $(TAUROOTDIR)/sgin32/lib/Makefile.tau-trace-kcc
CXX             = $(TAU_CXX)
CC              = $(TAU_CC)
CFLAGS          = $(TAU_INCLUDE) $(TAU_DEFS)
LIBS            = $(TAU_LIBS) -lm
LDFLAGS         = $(USER_OPT)

RM              = /bin/rm -f
TARGET          = matrix
##############################################
all:            $(TARGET)
install:        $(TARGET)
$(TARGET):      $(TARGET).o
      $(CXX) $(LDFLAGS) $(TARGET).o -o $@ $(LIBS)
$(TARGET).o : $(TARGET).cpp
      $(CXX) $(CFLAGS) -c $(TARGET).cpp
```

```
clean:
        $(RM) $(TARGET).o $(TARGET)
################################################
```

To use a different configuration, simply change the included makefile to some other. For example, for

```
% ./configure -pthread -arch=sgi64
```

substitute

```
include $(TAUROOTDIR)/sgi64/lib/Makefile.tau-pthread
```

in the makefile above. Also,

```
$(TAUROOTDIR)/include/Makefile
```

points to the most recently configured version of the library.

Enabling and Disabling the Instrumentation

Using the TAU stub makefile variable `TAU_DEFS` while compiling C++ and C source code enables profiling (or tracing) instrumentation and generates the performance data files. To disable the instrumentation, `TAU_DEFS` should not be used. In its absence, all the TAU profiling macros defined in the source code for instrumentation purposes are automatically defined to null (the default behavior). Thus, the instrumentation can be retained in the source code, since it has no overhead when it is disabled.

For Fortran however, the instrumentation can be disabled in the program by using the TAU stub makefile variable `TAU_DISABLE` on the link command line. This points to a library that contains empty TAU instrumentation routines.

Using TAU with MPI

TAU MPI wrapper library (libTauMpi.a) uses the MPI Profiling Interface for instrumentation. To use the library,

1. Configure TAU with -mpiinc=<dir> and -mpilib=<dir> command-line options that specify the location of MPI header files and the directory where MPI libraries reside. Example:

```
% ./configure -mpiinc=/usr/local/packages/mpich/
          include -mpilib=/usr/local/packages/mpich/
          lib/LINUX/ch_p4 -c++=KCC -cc=cc
```

2. Include the TAU stub Makefile generated in the application makefile.

```
TAUROOTDIR=/usr/local/packages/tau2
include $(TAUROOTDIR)/i386_linux/Makefile.tau-kcc
```

3. Use the Makefile variables `$(TAU_MPI_LIBS)` for C/C++ applications and `$(TAU_MPI_FLIBS)` for Fortran 90 applications, to specify the TAU MPI libraries before the `$(TAU_LIBS)` in the link command line. Also, use `$(TAU_MPI_INCLUDE)` in the compiler command line to specifies the MPI include directory to be used. Example:

```
CXX      = $(TAU_CXX)
CFLAGS   = $(TAU_INCLUDE) $(TAU_DEFS) $(TAU_MPI_INCLUDE)
LIBS     = $(TAU_MPI_LIBS) $(TAU_LIBS)
```

4. Compile and run the MPI application as usual to generate the performance data.

Environment Variables

When the program has been compiled, it can be executed as it normally would be (for example, using mpirun for an MPI task). TAU generates profile data files or trace files in the current working directory. One file for each context and thread is generated. To better manage different experiments, set the environment variables

- **PROFILEDIR** to name the directory that should contain the profile data files and
- **TRACEDIR** the directory where event traces should be stored.

- **LD_LIBRARY_PATH** (or **LIBPATH** for IBM) should include the <tauroot>/<tauarch>/lib directory if TAU is used with JAVA 2 (using the -jdk=<dir> configuration option) or dyninstAPI (using the -dyninst=<dir> configuration option).

Example:

```
% make
% setenv TRACEDIR /users/foo/tracedata/experiment1
% mpirun –np 4 matrix
```

NOTE: TAU also uses the environment variable **PCL_EVENT** and **PAPI_EVENT** to specify the hardware performance counter to be used when -pcl=<dir> or -papi=<dir> configuration options are used, respectively. See section "Using Hardware Performance Counters" in Chapter 4 for further details.

Application Scenarios

TAU's `examples` directory contains programs that illustrate the use of TAU instrumentation and measurement options.

instrument - This contains a simple C++ example that shows how TAU's API can be used for manually instrumenting a C++ program.

threads - A simple multi-threaded program that shows how the main function of a thread is instrumented. Performance data is generated for each thread of execution. Uses pthread library and TAU must be configured with the -pthread option.

cthreads - Same as threads above, but for a C program. An instrumented C program may be compiled with a C compiler, but needs to be linked with a C++ linker.

sproc - SGI sproc threads example. TAU should be configured with the -sproc option to use this.

pi - An MPI program that calculates the value of pi and e. It highlights the use of TAU's MPI wrapper library. TAU needs to be configured with -mpiinc=<dir> and -mpilib=<dir> to use this.

mpishlib - Demonstrates the use of MPI wrapper library in instrumenting a shared object. The MPI application is instrumented is instrumented as well. TAU needs to be configured with -mpiinc=<dir> and -mpilib=<dir> flags.

python - Instrumentation of a python application can be done automatically or manually using the TAU Python bindings. Two examples, auto.py and manual.py demonstrate this respectively. TAU needs to be configured with-pythoninc=<dir that contains Python.h> option and the user needs to set PYTHONPATH to <taudir>/<arch>/lib to use this feature.

traceinput - To build a trace converter/trace reader application, we provide the TAU trace input library. This directory contains two examples (in c and c++ subdirectories) that illustrate how an application can use the trace input API to read online or post-mortem TAU binary traces. It shows how the user can register routines with the callback interface and how TAU invokes these routines when events take place.

papi - A matrix multiply example that shows how to use TAU statement level timers for comparing the performance of two algorithms for matrix multiplication. When used with PAPI or PCL, this can highlight the cache behaviors of these algorithms. TAU should be configured with -papi=<dir> or -pcl=<dir> and the user should set PAPI_EVENT or PCL_EVENT respective environment variables, to use this.

papithreads - Same as papi, except uses threads to highlight how hardware performance counters may be used in a multi-threaded application. When it is used with PAPI, TAU should be configured with -papi=<dir> -pthread

autoinstrument - Shows the use of Program Database Toolkit (PDT) for automating the insertion of TAU macros in the source code. It requires configuring TAU with the -pdt=<dir> option. The Makefile is modified to illustrate the use of a source to source translator (tau_instrumentor).

reduce - Shows the use of **tau_reduce**, a utility that can read profiles and a set of rules and determine which routines should not be instru-

mented (for frequently called light-weight routines). See <tau>/utils/TAU_REDUCE.README file for further details. It requires configuring TAU with -pdt=<dir> option.

cinstrument - Shows the use of PDT for C. Requires configuring TAU with -pdt=<dir> option.

mixedmode - This example illustrates the use of PDT, hand-instrumentation (for threads), MPI library instrumentation and TAU system call wrapper library instrumentation. Requires configuring TAU with -mpiinc=<dir> -mpilib=<dir> -pdt=<dir> -pthread options.

pdt_mpi - This directory contains C, C++ and F90 examples that illustrate how TAU/PDT can be used with MPI. Requires configuring TAU with -pdt=<dir> -mpiinc=<dir> -mpilib=<dir> options. You may also try this with the -TRACE -epilog=<dir> options to use the EPILOG tracing package (from FZJ).

callpath - Shows the use of callpath profiling. Requires configuring TAU with the -PROFILECALLPATH option. Setting the environment variable TAU_CALLPATH_DEPTH changes the depth of the callpath recorded by TAU. The default value of this variable is 2.

selective - This example illustrates the use of PDT, and selective profiling using profile groups in the tau_instrumentor. Requires configuring TAU with -pdt=<dir> -fortran=<...> options.

NPB2.3 - The NAS Parallel Benchmark 2.3 [NPB-URL]. It shows how to use TAU's MPI wrapper with a manually instrumented Fortran program. LU and SP are the two benchmarks. LU is instrumented completely, while only parts of the SP program are instrumented to contrast the coverage of routines. In both cases MPI level instrumentation is complete. TAU needs to be configured with -mpiinc=<dir> and -mpilib=<dir> to use this.

dyninst - An example that shows the use of DyninstAPI [DYNINST-URL] to insert TAU instrumentation. Using Dyninst, no modifications are needed and tau_run, a runtime instrumentor, inserts TAU calls at routine transitions in the program. [This represents work in progress].

dyninstthreads - The above example with threads.

java/pi - Shows a java program for calculating the value of pi. It illustrates the use of the TAU JVMPI layer for instrumenting a Java program without any modifications to its source code, byte-code or the JVM. It requires a Java 2 compliant JVM and TAU needs to be configured with the -jdk=<dir> option to use this.

java/api The same Pi program as above that illustrates the use of the TAU API. There are subdirectories for C, C++ and F90 to show the differences in instrumentation and Makefiles. TAU needs to be configured with the -openmp option to use this.

openmp - Shows how to manually instrument an OpenMP program using the TAU API. There are subdirectories for C, C++ and F90 to show the differences in instrumentation and Makefiles. TAU needs to be configured with the -openmp option to use this.

opari - Opari is an OpenMP directive rewriting tool that works with TAU. Configure TAU with -opari=<dir> option to use this. This provides detailed instrumentation of OpenMP constructs. There are subdirectories for C++, pdt_f90, and OpenMPI to demonstrate the use of this tool. The pdt_f90 directory contains an example that shows the use of PDT with Opari for a Fortran 90 program.

openmpi - Illustrates TAU's support for hybrid execution models in the form of MPI for message passing and OpenMP threads. TAU needs to be configured with -mpiinc=<dir> -mpilib=<dir> -openmp options to use this.

fork - Illustrates how to register a forked process with TAU. TAU provides two options: TAU_INCLUDE_PARENT_DATA and TAU_EXCLUDE_PARENT_DATA which allows the child process to inherit or clear the performance data when the fork takes place.

mapping - Illustrates two examples in the embedded and external subdirectories. These correspond to profiling at the object level, where the time spent in a method is displayed for a specific object. There are two ways to achieve this using an embedded association. The first method requires an extension of the class definition with a TAU pointer and the second scheme uses external hash-table lookup that

relies on looking at the object address at each method invocation. Both of these examples illustrate the use of the TAU Mapping API.

multicounters - Illustrates the use of multiple measurement options configured simultaneously in TAU. See README file for instructions on setting the env. variables COUNTERS<1-25> for specifying measurements. Requires configuring TAU with -MULTIPLECOUNTERS.

selectiveAccess- Illustrates the use of TAU API for runtime access of TAU performance data. A program can get information about routines executing in its context. This can be used in conjunction with multiple counters.

CHAPTER 3 *Instrumentation*

For TAU instrumentation, macros must be added to the source code to identify routine transitions. It can be done automatically using the C++ instrumentor - **tau_instrumentor**, based on the Program Database Toolkit, manually using the instrumentation API (Application Programmers Interface) or using the **tau_run,** a runtime instrumentor, based on the DynInstAPI dynamic instrumentation package. Python applications can be instrumented automatically by using the tau python package. Java applications can be instrumented automatically by using the JVMPI TAU library.

Automatic Instrumentation of C++, C and F90 source code

tau_instrumentor inserts TAU instrumentation in C++ source code using PDT [PDT-URL].

1. Install pdtoolkit. Refer to the README file in the PDT directory.
   ```
   % ./configure -arch=IRIX64 -KCC
   ```
2. Install TAU using the -pdt configuration option.
   ```
   % ./configure -pdt=/usr/local/packages/pdtoolkit-1.0
         -c++=KCC -arch=sgi64 -SGITIMERS
   % make install
   ```
3. Modify the makefile to invoke cxxparse from PDT which generates a program database file (.pdb) that contains program entities (such as routine locations) and tau_instrumentor which uses the .pdb file and the C++ source code to generate an instrumented version of the source code.
4. **tau_instrumentor** takes the following commandline options:

   ```
   Usage: /apps/tau-2.x/sgi64/bin/tau_instrumentor
   <pdbfile> <sourcefile> [-o <outputfile>] [-noinline]
   [-noinit][-g groupname] [-i headerfile] [-c|-c++|-
   fortran] [-f <instr_req_file> ] [-rn
   <return_keyword>] [-rv <return_void_keyword>]
   ```

   The -noinline option prevents the instrumentation of inlined routines. All routines in the source file can be logically grouped into a TAU group using the -g “groupname” option. An alternate header file can be used (instead of the default Profiler.h using the -i headerfile option. For a C/C++ program, TAU inserts the TAU_INIT call to parse the commandline parameters in main. To prevent this default behavior, specify -noinit on the commandline. The instrumentor can automatically deduce the language of the source file by examining the pdb file. The user can override this behavior using the -c, -c++ or -fortran option by specifying the language associated with the source file. This affects the placement of instrumentation in the source file. To specify a selective instrumentation file, use the -f file option. A selective instrumentation file can contain a list of routines to exclude from instrumentation (one per line bracketed by BEGIN_EXCLUDE_LIST and END_EXCLUDE_LIST). Alternately, you can specify a list of routines that should be included for instrumentation (no other routines are instrumented besides the ones in this list). Such a list is bracketed by BEGIN_INCLUDE_LIST and END_INCLUDE_LIST as shown

in the <taudir>/examples/autoinstrument/select file. The final -rn and -rv <keywords> options to tau_instrumentor allows the user to specify a macro that calls return in a routine with a non-void and a void return type respectively. TAU's timers need to be stopped before a return statement. When the source code contains a macro that calls return, it is important to stop the timers before the macro is invoked in the instrumented source code. These -rn and -rv options help the tau_instrumentor identify the location of subroutine exits.

5. See `examples/autoinstrument/Makefile`. For example, the original makefile

```
CXX             = CC
CFLAGS          =
LIBS            = -lm
TARGET          = klargest
################################################
# Original Rules
################################################
all:            $(TARGET)
$(TARGET):      $(TARGET).o
        $(CXX) $(LDFLAGS) $(TARGET).o -o $@ $(LIBS)
$(TARGET).o : $(TARGET).cpp
        $(CXX) $(CFLAGS) -c $(TARGET).cpp
clean:
        $(RM) $(TARGET).o $(TARGET)
################################################
```

is modified as follows. Some changes are shown in bold font.

```
TAUROOTDIR      = /usr/local/packages/tau2/
include $(TAUROOTDIR)/sgi64/Makefile.tau
CXX             = $(TAU_CXX)
CFLAGS          = $(TAU_INCLUDES) $(TAU_DEFS)
LIBS            = -lm $(TAU_LIBS)
PDTPARSE =$(PDTDIR)/$(CONFIG_ARCH)/bin/cxxparse
TAUINSTR =$(TAUDIR)/$(CONFIG_ARCH)/bin/tau_instrumentor
################################################
# Modified Rules
################################################

all:    $(TARGET) $(PDTPARSE) $(TAUINSTR)
```

```
$(TARGET): $(TARGET).o
      $(CXX) $(LDFLAGS) $(TARGET).o -o $@ $(LIBS)

# Use the instrumented source code to generate the
object code
$(TARGET).o : $(TARGET).inst.cpp
       $(CXX) -c $(CFLAGS) $(TARGET).inst.cpp  -o $(TAR-
GET).o

# Generate the instrumented source from the original
source and the pdb file
$(TARGET).inst.cpp : $(TARGET).pdb $(TARGET).cpp
$(TAUINSTR)
       $(TAUINSTR) $(TARGET).pdb $(TARGET).cpp -o $(TAR-
GET).inst.cpp

# Parse the source file to generate the pdb file
$(TARGET).pdb : $(PDTPARSE) $(TARGET).cpp
        $(PDTPARSE) $(TARGET).cpp $(CFLAGS)

clean:
        $(RM) $(TARGET).o $(TARGET).inst.cpp $(TARGET)
$(TARGET).pdb
################################################
$(PDTPARSE):
  @echo "****************************************"
   @echo "Download and Install Program Database Toolkit "
   @echo "ERROR: Cannot find $(PDTPARSE)"
   @echo "****************************************"
$(TAUINSTR):
   @echo "****************************************"
   @echo "Configure TAU with -pdt=<dir> option to use"
   @echo "C++ instrumentation with PDT"
   @echo "ERROR: Cannot find $(TAUINSTR)"
   @echo "****************************************"
```

6. Compile and execute the application.

The user may also opt to manually insert TAU macros in the source code using the C++ instrumentation API. The following section describes this API in detail.

Using TAU with PDT and MPI

To use PDT for source code instrumentation and TAU's MPI wrapper interposition library, modify the default compilation rule as shown in the example below:

```
TAUROOTDIR = /usr/local/packages/tau-2.x
include $(TAUROOTDIR)/include/Makefile
USE_TAU = 1
# Comment above line to disable TAU

CXX             = $(TAU_CXX)
PDTCXXPARSE     = $(PDTDIR)/$(PDTARCHDIR)/bin/cxxparse
TAUINSTR = $(TAUROOTDIR)/$(CONFIG_ARCH)/bin/
          tau_instrumentor

CFLAGS   = $(TAU_INCLUDE) $(TAU_DEFS) $(TAU_MPI_INCLUDE)
LIBS  = $(TAU_MPI_LIBS) $(TAU_LIBS) $(LEXTRA1)
          $(EXTRALIBS) -lm

LDFLAGS  = $(USER_OPT)  $(TAU_LDFLAGS)

##############################################
ifdef USE_TAU
COMP_RULE = $(PDTCXXPARSE) $< $(CFLAGS); \
       $(TAUINSTR) $*.pdb $< -o $*.inst.cpp -g
RING;\
       $(CXX) $(CFLAGS) -c $*.inst.cpp -o $@ ; \
rm -f $*.pdb ;
else
# DISABLE TAU INSTRUMENTATION
TAU_DEFS =
# Don't use TAU MPI wrapper library
TAU_MPI_LIBS = -L/usr/local/lib -lmpich
TAU_LIBS =
TAU_WRAPPER_LIB =
TAU_INCLUDE =
COMP_RULE = $(CXX) $(CFLAGS) -c $< -o $@ ;
endif
########################
TARGET = ring
all:            $(TARGET)
```

```
OBJS = $(TARGET).o
$(TARGET):      $(OBJS)
        $(CXX) $(LDFLAGS) $(OBJS) -o $@ $(LIBS)
# Compilation rule
.cpp.o:
        $(COMP_RULE)
```

Using TAU with PDT for an F90 MPI application

A typical Fortran90 MPI program uses a Makefile that builds object files from a set of Fortran90 source files and links libraries with the objects to build an executable. The original Makefile might look like this:

```
LIBS=<libraries>
OBJS=<list of object files>
app: $(OBJS)
      $(F90) $(LDFLAGS) $(OBJS) -o app
.f90.o:
      $(F90) $(FFLAGS) -c $< -o $@
```

First the user inserts the TAU stub Makefile and defines the Makefile variables.

For e.g.

```
include /usr/local/packages/tau-2.13/rs6000/lib/Make-
file.tau-papiwallclock-multiplecounters-papivirtual-
mpi-papi-pdt

F90= $(TAU_F90)
PDTF90PARSE = $(PDTDIR)/$(PDTARCHDIR)/bin/f95parse
TAUINSTR= $(TAUROOTDIR)/$(CONFIG_ARCH)/bin/
tau_instrumentor
FFLAGS= $(TAU_F90_SUFFIX) $(TAU_MPI_INCLUDE)
LIBS= $(TAU_MPI_FLIBS) $(TAU_LIBS) $(TAU_FORTRANLIBS)
LINKER= $(TAU_LINKER)
```

Next, the compilation rule is changed to first parse the original file using PDT's parser, then instrument the source code and compile the instrumented source code. The PDT script f95parser invokes the new Cleanscape Software International's Flint based F95 parser whereas f90parse invokes the older Mutek Solution's F90 parser. Please refer to PDT's README file for a listing of f90parse and f95parse commandline options.

In case there are any errors in parsing the source code, a fall-back rule is introduced to keep the original compilation rule in place.

```
COMP_RULE= -$(PDTPARSE $< $(FFLAGS) -Mpdt_modules; \
   $(TAUINSTR) $*.pdb $< -o $*.inst.f90 -f select.dat; \
   $(F90) $(FFLAGS) -c $*.inst.f90 -o $@; \
   if [ ! -f $@ ] ; then \
   echo "Error in compiling $*.f90: trying without PDT";
\
   $(F90) $(FFLAGS) -c $< -o $@; \
   fi ; \
   rm -f $*.pdb;
```

The - before $(PDTPARSE) allows make to proceed when there is an error from any command in the rule. The -Mpdt_modules option reads and writes the .mod module files in a directory called pdt_modules (which should be created prior to invoking make) if you're using f90parse. It is not required for f95parse. f95parse can also parse multiple F90 files together when these are specified on the command line. The -o<file.pdb> option should be used for specifying the merged PDB file name if more than one file is parsed at a time. Once a PDB file is generated, it is processed by the next stage in the instrumentation process using tau_instrumentor. The -f select.dat specifies a selective instrumentation list where the user can specify which routines and/or files should be excluded from instrumentation. An optional -g <groupname> argument to tau_instrumentor puts all routines in a given file in a named profile group. Groups can be used to enable or disable the performance instrumentation for a group of logically related routines. The user introduces the compilation rule as below:

```
app: $(OBJS)
        $(LINKER) $(LDFLAGS) $(OBJS) Do app
.f90.o:
        $(COMP_RULE)
```

Using TAU with PDT and Opari

The following example shows the use of PDT and Opari for OpenMP instrumentation in a Makefile.

```
TAUROOTDIR= /usr/local/tau-2.x
include $(TAUROOTDIR)/include/Makefile
CXX= $(TAU_CXX)
CC= $(TAU_CC)
F90= $(TAU_F90)
PDTCPARSE       = $(PDTDIR)/$(PDTARCHDIR)/bin/cparse
PDTF90PARSE     = $(PDTDIR)/$(PDTARCHDIR)/bin/f90parse
TAUINSTR        = $(TAUROOTDIR)/$(CONFIG_ARCH)/bin/
tau_instrumentor
OPARI_TOOL= $(OPARIDIR)/tool/opari
CFLAGS = $(TAU_INCLUDE) $(TAU_DEFS)
FFLAGS        =
LIBS            =  $(TAU_LIBS)  $(TAU_FORTRANLIBS)
$(LEXTRA1)
#LIBS            = $(TAU_DISABLE)  $(TAU_FORTRANLIBS)

LDFLAGS      = $(USER_OPT)
MAKEFILE      = Makefile
TARGET      = mandel
###############################################

install: $(TARGET)
$(TARGET):ppm.o $(TARGET).o mytimer.o opari.tab.o
@echo "****************************************"
@echo "LINKING: "
$(TAU_LINKER) $(LDFLAGS) $(TARGET).o ppm.o mytimer.o
opari.tab.o -o $@ $(LIBS)

$(TARGET).o : $(TARGET).f90 ppm.o
@echo "****************************************"
@echo "Creating $(TARGET).o:"
$(RM) opari.rc
$(OPARI_TOOL) -nosrc -table opari.tab.c $*.f90
$*.pomp.f90
```

```
$(PDTF90PARSE) $*.pomp.f90 -MPDT_MODULES
$(TAUINSTR) $*.pomp.pdb $*.pomp.f90 -o $*.inst.f90
$(F90) $(FFLAGS) -c $*.inst.f90 -o $@

ppm.o : ppm.f90
@echo "*****************************************"
@echo "Creating ppm.o: "
$(PDTF90PARSE) $<
if [ -d PDT_MODULES ] ; then true; \
  else mkdir PDT_MODULES ; fi
if [ ! -f PPM.mod ] ; then true ; \
  else mv PPM.mod PDT_MODULES ; fi

$(TAUINSTR) $*.pdb $< -o $*.inst.f90
$(F90) $(FFLAGS) -c $*.inst.f90 -o $@

opari.tab.o: $(TARGET).o
   @echo "*****************************************"
   @echo "Creating opari.tab.o:"
   $(CC) $(CFLAGS) -c opari.tab.c

mytimer.o : mytimer.c
   @echo "*************************************"
   @echo "Creating mytimer.o:"
   $(PDTCPARSE) $<
   $(TAUINSTR) $*.pdb $< -o $*.inst.c
   $(CC) $(CFLAGS) -c $*.inst.c -o $@
```

Selective Instrumentation

When all routines in a source file are instrumented, frequently executing light-weight routines may cause an instrumentation overhead that distorts the performance data. To reduce the instrumentation overhead, the user can select routines that should not be instrumented and specify this in the selective instrumentation file that can be specified as a commandline argument to tau_instrumentor (-f <file>). The format of this file is shown below:

```
BEGIN_INCLUDE_LIST
main
foo
f12
END_INCLUDE_LIST
```

instruments main, foo and f12 routines only.

```
BEGIN_EXCLUDE_LIST
domain
f1
f2
f4
END_EXCLUDE_LIST
```

excludes routines domain, f1, f2 and f4 from instrumentation.

```
BEGIN_FILE_INCLUDE_LIST
Main.f90
Foo?.f
END_FILE_INCLUDE_LIST
```

specifies that main.f90 and foo?.f files (foo3.f, foo9.f) are instrumented.

```
BEGIN_FILE_EXCLUDE_LIST
*.cpp
bar.f90
END_FILE_EXCLUDE_LIST
```

excludes all files with that end in a .cpp suffix, and bar.f90 from instrumentation. The user should specify either an exclude list or an include list, but not both. Sometimes, it is difficult to build this selective instrumentation file manually. The tool tau_reduce may be used to construct this file automatically.

TAU_REDUCE: A tool for reducing instrumentation overhead

When all routines in a source file are instrumented, frequently executing light-weight routines may cause an instrumentation overhead that distorts the performance data. To reduce the instrumentation overhead, the user can select routines that should not be instrumented and specify this in a selective instrumentation file that can be specified as a commandline argument to tau_instrumentor or tau_run. However, creating lists of routines manually can be tedious. To aid the user in identifying which routines should be removed, tau_reduce may be used.

tau_reduce is an application that will apply a set of user-defined rules to a pprof dump file in order to create a select file that will include an exclude list for selective implementation for TAU. The user must specify the name of the pprof dump file that this application will use. This is done with the -f filename flag. If no rule file is specified, then a single default rule will be applied to the file. This rule is:

```
numcalls > 1000000 & usecs/call < 2,
```

which will exclude all routines that are called at least 1,000,000 times and average less then two microseconds per call. If a rule file is specified, then this rule is not applied. If no output file is specified, then the results will be printed out to the screen.

OPTIONS:

tau_reduce has the following options available at the command line:

-f <filename> : specify **F**ilename of pprof dump file (default: temp.out)

-p : **P**rint out all routines with their attributes (for debugging)

-o <filename> : specify filename for select file **O**utput (default: print to screen)

-r <filename> : specify filename for **R**ule file

-v : **V**erbose mode (for each rule, print out rule and all routines that it excludes)

RULES:

Users can specify a set of rules for tau_reduce to apply. The rules should be specified in a separate file, one rule per line, and the file name should be specified with the appropriate option on the command line. The grammar for a rule is:

```
[GROUPNAME:]FIELD OPERATOR NUMBER.
```

The GROUPNAME followed by the colon (:) is optional. If included, the rule will only be applied to routines that are a member of the group specified. Only one group name can be applied to each rule, and a rule must follow a groupname. If only a groupname is given, then an unrecognized field error will be returned. If the desired effect is to exclude all routines that belong to a certain group, then a trivial rule, such as GROUP:numcalls > -1 may be applied. If a groupname is given, but the data does not contain any groupname data, then an error message will be given, but the rule will still be applied to the date ignoring the groupname specification.

A FIELD is any of the routine attributes listed in the following table:

TABLE 1. Fields used in specifying rules in tau_reduce

ATTRIBUTE NAME	MEANING
numcalls	Number of times this routine is called
numsubrs	Number of subroutines that this routine contains
percent	Percent of total implementation time
cumusec	Inclusive routine running time, in microseconds
count	Exclusive hardware count
totalcount	Inclusive hardware count
stddev	Standard deviation
usecs/call	Microseconds per call
counts/call	Hardware counts per call

Some FIELDS are only available for certain files. If hardware counters are used, then usec, cumusec, usecs/per call are not applicable and a error is reported. The opposite is true if timing data is used rather than hardware counters. Also, stddev is only available for certain files that contain that data (when TAU is configured with -PROFILESTATS). An OPERATOR is any of the following: < (less than), > (greater

than), or = (equals). A NUMBER is any number. A compound rule may be formed by using the & (and) symbol in between two simple rules. There is no “OR” because there is an implied or between two separate simple rules, each on a separate line. (i.e. the compound rule usec < 1000 OR numcalls = 1 is the same as the two simple rules “usec < 1000” and “numcalls = 1”).

EXAMPLES:

```
#exclude all routines that are members of TAU_USER and
have less than
#1000 microseconds
TAU_USER:usec < 1000

#exclude all routines that have less than 1000 microsec-
onds and are
#called only once.
usec < 1000 & numcalls = 1

#exclude all routines that have less than 1000 usecs per
call OR have a percent
#less than 5
usecs/call < 1000
percent < 5
```

NOTE: Any line in the rule file that begins with a # is a comment line. For

clarity, blank lines may be inserted in between rules and will also be ignored.

C++ Measurement API

The API is a set of macros that can be inserted in the C++ source code. An extension of the same API is available to instrument C and Fortran sources. This is discussed later.

At the beginning of each instrumented source file, include the following header

```
#include <Profile/Profiler.h>
```

TAU_PROFILE(function_name, type, group);

```
Arguments:
char *function_name or string& function_name
char *type_name or string& type
TauGroup_t group
```

With TAU_PROFILE, the function function_name is profiled. TAU_PROFILE identifies the function uniquely by the combination of its name and type parameters. Each function is also associated with the group specified. This information can selectively enable or disable instrumentation in a set of profile groups. A function that belongs to the TAU_DEFAULT group is always profiled. Other user defined groups are TAU_USER, TAU_USER1, TAU_USER2, TAU_USER3, TAU_USER4. The top level function in any thread must be profiled using the TAU_DEFAULT group. For details on using selective instrumentation, please refer to the section "Running the application" in Chapter 4.

Example:

```
int main(int argc, char **argv)
{
TAU_PROFILE("main()","int (int, char **)",TAU_DEFAULT);
```

string& CT(variable);

```
Arguments:
<type> variable
```

The CT macro returns the runtime type information string of a variable. This is useful in constructing the type parameter of the TAU_PROFILE macro. For templates, the type information can be constructed using the type of the return and the type of each of the arguments (parameters) of the template. The example in the following macro will clarify this.

TAU_TYPE_STRING(variable, type_string);

```
Arguments:
string & variable;
string & type_string;
```

This macro assigns the string constructed in type_string to the variable. The + operator and the `CT` macro can be used to construct the type string of an object. This is useful in identifying templates uniquely, as shown below.

Example:

```
template<class PLayout>
ostream& operator<<(ostream& out, const Particle-
Base<PLayout>& P) {
 TAU_TYPE_STRING(taustr, "ostream (ostream, " + CT(P) +
     " )" );
  TAU_PROFILE("operator<<()", taustr, TAU_PARTICLE |
     TAU_IO);
...
}
```

When PLayout is instantiated with "`UniformCartesian<3U, double>`", this generates the unique template name:

```
"operator<<() ostream const ParticleBase<UniformCarte-
     sian<3U, double> > )"
```

The following example illustrates the usage of the `CT` macro to extract the name of the class associated with the given object using `CT(*this);`

```
template<class PLayout>
unsigned ParticleBase<PLayout>::GetMessage(Message&
     msg, int node) {
  TAU_TYPE_STRING(taustr, CT(*this) + " unsigned (Mes-
     sage, int)");
  TAU_PROFILE("ParticleBase::GetMessage()", taustr,
     TAU_PARTICLE);
...
}
```

When PLayout is instantiated with "UniformCartesian<3U, double>", this generates the unique template name:

```
"ParticleBase::GetMessage() ParticleBase<UniformCarte-
    sian<3U, double> > unsigned (Message, int)"
```

TAU_PROFILE_TIMER(timer, name, type, group);

```
Arguments:
Profiler timer;
char *name or string& name;
char *type or string& type;
TauGroup_t group;
```

With TAU_PROFILE_TIMER, a group of one or more statements is profiled. This macro has a timer variable as its first argument, and then strings for name and type, as described earlier. It associates the timer to the profile group specified in the last parameter.

```
Example:
template< class T, unsigned Dim >
void BareField<T,Dim>::fillGuardCells(bool reallyFill)
{
  // profiling macros
  TAU_TYPE_STRING(taustr, CT(*this) + " void (bool)" );
  TAU_PROFILE("BareField::fillGuardCells()", taustr,
    TAU_FIELD);

  TAU_PROFILE_TIMER(sendtimer, "fillGuardCells-send",
                    taustr, TAU_FIELD);
  TAU_PROFILE_TIMER(localstimer, "fillGuardCells-
    locals", taustr, TAU_FIELD);
```

TAU_PROFILE_START(timer);

```
Arguments:
Profiler timer;
```

The macro `TAU_PROFILE_START` starts the timer associated with the set of statements that are to be profiled.

TAU_PROFILE_STOP(timer);

```
Arguments:
Profiler timer;
```

The macro `TAU_PROFILE_STOP` stops the timer.

It is important to note that timers can be nested, but not overlapping. TAU detects programming errors that lead to such overlaps at runtime, and prints a warning message.

Example:

```
template< class T, unsigned Dim >
void BareField<T,Dim>::fillGuardCells(bool reallyFill)
{
  // profiling macros
  TAU_TYPE_STRING(taustr, CT(*this) + " void (bool)" );
  TAU_PROFILE("BareField::fillGuardCells()", taustr,
     TAU_FIELD);

  TAU_PROFILE_TIMER(sendtimer, "fillGuardCells-send",
                    taustr, TAU_FIELD);
  TAU_PROFILE_TIMER(localstimer, "fillGuardCells-
     locals", taustr, TAU_FIELD);
// ...
  TAU_PROFILE_START(sendtimer);
    // set up messages to be sent
    Message** mess = new Message*[nprocs];
```

```
    int iproc;
    for (iproc=0; iproc<nprocs; ++iproc) {
      mess[iproc] = NULL;
      recvmsg[iproc] = false; }//... other code
  TAU_PROFILE_STOP(sendtimer);
  ...
}
```

TAU_GLOBAL_TIMER(timer, name, type, group);

```
Arguments:
Profiler timer;
string name, type;
TauGroup_t group;
```

As TAU_PROFILE_TIMER is used within the scope of a block (typically a routine), TAU_GLOBAL_TIMER can be used across different routines in the same file.

TAU_GLOBAL_TIMER_START(timer);

```
Arguments:
Profiler timer;
```

`TAU_GLOBAL_TIMER_START` starts the timer. The timer in this case is declared in the file scope.

TAU_GLOBAL_TIMER_STOP(timer);

```
Arguments:
Profiler timer;
```

`TAU_GLOBAL_TIMER_STOP` stops the timer which is declared in the file scope.

Example:

```
TAU_GLOBAL_TIMER(looptimer, "Loops in foo.cpp", " ",
                    TAU_USER);
void foo()
{
   TAU_GLOBAL_TIMER_START(looptimer);
   for (i=0; i<N; i++) { /* do something */ }
   TAU_GLOBAL_TIMER_STOP(looptimer);
}
void bar()
{
   TAU_GLOBAL_TIMER_START(looptimer);
   for(j=0; j<N; j++) { /* do something */ }
   TAU_GLOBAL_TIMER_STOP(looptimer);
}
```

TAU_PROFILE_SET_GROUP_NAME(groupname);

```
Arguments:
char *groupname;
```

TAU_PROFILE_SET_GROUP_NAME macro allows the user to change the group name associated with the instrumented routine. This macro must be called within the instrumented routine.

```
void foo()
{
  TAU_PROFILE("foo()", "void ()", TAU_USER);
  TAU_PROFILE_SET_GROUP_NAME("Particle");
  /* gives a more meaningful group name */
}
```

TAU_PROFILE_TIMER_SET_GROUP_NAME(timer, groupname);

```
Arguments:
Profiler timer;
char *groupname;
```

TAU_PROFILE_TIMER_SET_GROUP_NAME changes the group name associated with a given timer.

Example:

```
void foo()
{
   TAU_PROFILE_TIMER(looptimer, "foo: loop1", " ",
        TAU_USER);
   TAU_PROFILE_START(looptimer);
   for (int i = 0; i < N; i++) { /* do something */ }
   TAU_PROFILE_STOP(looptimer);
   TAU_PROFILE_TIMER_SET_GROUP_NAME("Field");
}
```

TAU_PROFILE_STMT(statement);

```
Arguments:
statement;
```

TAU_PROFILE_STMT declares a variable that is used only during profiling or for execution of a statement that takes place only when the instrumentation is active. When instrumentation is inactive (i.e., when profiling and tracing are turned off as described in Chapter 2), all macros are defined as null.

Example:

```
TAU_PROFILE_STMT(T obj;); // T is a template parameter)
TAU_TYPE_STRING(str, "void () " + CT(obj) );
```

TAU_PROFILE_INIT(argc, argv);

```
Arguments:
int argc;
char **argv;
```

TAU_PROFILE_INIT parses the command-line arguments for the names of profile groups that are to be selectively enabled for instrumentation. By default, if this macro is not used, functions belonging to all profile groups are enabled.

Example:

```
int main(int argc, char **argv){
  TAU_PROFILE("main()", "int (int, char **)",
     TAU_DEFAULT);
  TAU_PROFILE_INIT(argc, argv);
...
}
```

TAU_PROFILE_SET_NODE(myNode);

```
Arguments:
int myNode;
```

The TAU_PROFILE_SET_NODE macro sets the node identifier of the executing task for profiling and tracing. Tasks are identified using node, context and thread ids. The profile data files generated will accordingly be named profile.<node>.<context>.<thread>.

TAU_PROFILE_SET_CONTEXT(myContext);

```
Argument:
```

```
int myContext;
```

TAU_PROFILE_SET_CONTEXT sets the context parameter of the executing task for profiling and tracing purposes. This is similar to setting the node parameter with TAU_PROFILE_SET_NODE.

TAU_REGISTER_THREAD();

To register a thread with the profiling system, invoke the TAU_REGISTER_THREAD macro in the run method of the thread prior to executing any other TAU macro. This sets up thread identifiers that are later used by the instrumentation system.

TAU_REGISTER_FORK(nodeid, option);

```
Arguments:
int nodeid;
enum TauFork_t  option;
/* TAU_INCLUDE_PARENT_DATA or TAU_EXCLUDE_PARENT_DATA*/
```

To register a child process obtained from the fork() syscall, invoke the TAU_REGISTER_FORK macro. It takes two parameters, the first is the node id of the child process (typically the process id returned by the fork call or any 0..N-1 range integer). The second parameter specifies whether the performance data for the child process should be derived from the parent at the time of fork (TAU_INCLUDE_PARENT_DATA) or should be independent of its parent at the time of fork (TAU_EXCLUDE_PARENT_DATA). If the process id is used as the node id, before any analysis is done, all profile files should be converted to contiguous node numbers (from 0..N-1). It is highly recommended to use flat contiguous node numbers in this call for profiling and tracing.

Example:

```
   pID = fork();
 if (pID == 0)      {
   printf("Parent : pid returned %d\n", pID)
```

```
 }     else {
// If we'd used the TAU_INCLUDE_PARENT_DATA, we'd get
//  the performance data from the parent in this process
//  as well.
    TAU_REGISTER_FORK(pID, TAU_EXCLUDE_PARENT_DATA);
       printf("Child : pid = %d", pID);
  }
```

TAU_PROFILE_EXIT(message);

```
Argument:
const char * message;
```

TAU_PROFILE_EXIT should be called prior to an error exit from the program so that any profiles or event traces can be dumped to disk before quitting.

Example:

```
if ((ret = open(...)) < 0) {
  TAU_PROFILE_EXIT("ERROR in opening a file");
  perror("open() failed");
  exit(1);
}
```

TAU_PROFILE_TIMER_SET_NAME(t, newname)

```
Arguments:
Profiler timer;
string newname;
```

TAU_PROFILE_TIMER_SET_NAME macro changes the name associated with a timer to the newname argument.

Example:

```
void foo()
{
```

```
  TAU_PROFILE_TIMER(timer1, "foo:loop1", " ", TAU_USER);
  ...
  TAU_PROFILE_TIMER_SET_NAME(timer1, "foo:lines 21-34");
}
```

TAU_PROFILE_TIMER_SET_TYPE(t, newtype)

```
Arguments:
Profiler t;
string newtype;
```

This macro changes the type string associated with the timer. Similar to `TAU_PROFILE_TIMER_SET_NAME`.

TAU_PROFILE_TIMER_SET_GROUP(t, group)

```
Arguments:
Profiler t;
TauGroup_t group;
```

`TAU_PROFILE_TIMER_SET_GROUP` changes the group associated with a timer.

Example:

```
void foo()
{
 TAU_PROFILE_TIMER(t, "foo loop timer", " ", TAU_USER1);
...
 TAU_PROFILE_TIMER_SET_GROUP(t, TAU_USER3);
}
```

TAU_DISABLE_INSTRUMENTATION();

TAU_DISABLE_INSTRUMENTATION macro disables all entry/exit instrumentation within all threads of a context. This allows the user to selectively enable and disable instrumentation in parts of his/her code. It is important to re-enable the instrumentation within the same basic block and scope.

TAU_ENABLE_INSTRUMENTATION();

TAU_ENABLE_INSTRUMENTATION macro re-enables all TAU instrumentation. All instances of functions and statements that occur between the disable/enable section are ignored by TAU. This allows a user to limit the trace size, if the macros are used to disable recording of a set of iterations that have the same characteristics as, for example, the first recorded instance.

Example:

```
main() {
  foo();
  TAU_DISABLE_INSTRUMENTATION();
  for (int i =0; i < N; i++) {
    bar();  // not recorded
  }
  TAU_ENABLE_INSTRUMENTATION();
  bar(); // recorded
}
```

TAU_ENABLE_GROUP(group);

```
Arguments:
TauGroup_t group;
```

TAU_ENABLE_GROUP macro turns on instrumentation in all routines associated with the profile group.

Example:

```
void foo()
{
  TAU_PROFILE("foo()", " ", TAU_USER);

  ...
  TAU_ENABLE_GROUP(TAU_USER);
}
```

TAU_DISABLE_GROUP(group);

```
Arguments:
TauGroup_t group;
```

TAU_DISABLE_GROUP macro turns off instrumentation in all routines associated with the profile group.

Example:

```
void foo()
{
  TAU_PROFILE("foo()", " ", TAU_USER);

  ...
  TAU_DISABLE_GROUP(TAU_USER);
}
```

TAU_GET_PROFILE_GROUP(groupname);

```
Arguments:
string groupname;
```

TAU_GET_PROFILE_GROUP allows the user to dynamically create groups based on strings, rather than use predefined, statically assigned groups such as TAU_USER1, TAU_USER2 etc. This allows names to be associated in creating unique groups that are more meaningful, using names of files or directories for instance.

Example:

```
#define PARTICLES TAU_GET_GROUP("PARTICLES")
void foo()
{
  TAU_PROFILE("foo()", " ", PARTICLES);
}
void bar()
{
  TAU_PROFILE("bar()", " ", PARTICLES);
}
```

TAU_ENABLE_GROUP_NAME(groupname);

```
Arguments:
string groupname;
```

`TAU_ENABLE_GROUP_NAME` macro can turn on the instrumentation associated with routines based on a dynamic group assigned to them. It is important to note that this and the `TAU_DISABLE_GROUP_NAME` macros apply to groups created dynamically using `TAU_GET_PROFILE_GROUP`.

TAU_DISABLE_GROUP_NAME(groupname);

```
Arguments:
string groupname;
```

Similar to TAU_ENABLE_GROUP_NAME, this macro turns off the instrumentation in all routines associated with the dynamic group created using TAU_GET_PROFILE_GROUP.

Example:

```
#define PARTICLES TAU_GET_PROFILE_GROUP("PARTICLES");
void foo()
{
  TAU_DISABLE_GROUP_NAME("PARTICLES");
  /* after some work */
  TAU_ENABLE_GROUP_NAME("PARTICLES");
}
```

TAU_ENABLE_ALL_GROUPS();

This macro turns on instrumentation in all groups

TAU_DISABLE_ALL_GROUPS();

This macro turns off instrumentation in all groups.

Example:

```
void foo()
{
  TAU_DISABLE_ALL_GROUPS();
  TAU_ENABLE_GROUP_NAME("PARTICLES");
```

```
}
```

TAU_REGISTER_EVENT(variable, event_name);

```
Arguments:
TauUserEvent & variable;
char * event_name;
```

TAU can profile user-defined events using `TAU_REGISTER_EVENT`. The meaning of the event is determined by the user.

TAU_EVENT(variable, value);

```
Arguments: TauUserEvent & variable;
double value;
```

`TAU_EVENT` associates a value with some user-defined event. When the event is triggered and this macro is executed, TAU maintains statistics, such as maximum, minimum values, standard deviation, number of samples, etc. for tracking this event.

Example:

```
int ArraySend(int arrayid)
{
  TAU_REGISTER_EVENT(taumsgsize, "Size of message asso-
     ciated with Arrays");
  int size = GetArraySize(arrayid);
  TAU_EVENT(size);
// ...
}
```

TAU_REPORT_STATISTICS();

TAU_REPORT_STATISTICS prints the aggregate statistics of user events across all threads in each node. Typically, this should be called just before the main thread exits.

TAU_REPORT_THREAD_STATISTICS();

TAU_REPORT_THREAD_STATISTICS prints the aggregate, as well as per thread user event statistics. Typically, this should be called just before the main thread exits.

TAU_TRACE_SENDMSG(tag, destination, length);

```
Arguments:
int tag;
int destination;
int length;
```

TAU_TRACE_SENDMSG traces an inter-process message communication when a tagged message is sent to a destination process.

TAU_TRACE_RECVMSG(tag, source, length);

```
Arguments:
int tag;
int source;
int length;
```

TAU_TRACE_RECVMSG traces a receive operation where tag represents the type of the message received from the source process.

Example:

```
if (pid == 0){
  TAU_TRACE_SENDMSG(currCol, sender, ncols * sizeof(T));
  MPI_Send(vctr2, ncols * sizeof(T), MPI_BYTE, sender,
     currCol, MPI_COMM_WORLD);
} else {
  MPI_Recv(&ans, sizeof(T), MPI_BYTE, MPI_ANY_SOURCE,
     MPI_ANY_TAG,MPI_COMM_WORLD, &stat);
  MPI_Get_count(&stat, MPI_BYTE, &recvcount);
  TAU_TRACE_RECVMSG(stat.MPI_TAG, stat.MPI_SOURCE,
     recvcount);
}
```

NOTE: When TAU is configured to use MPI (-mpiinc=<dir> -mpilib=<dir>), the TAU_TRACE_RECVMSG and TAU_TRACE_SENDMSG macros are not required. The wrapper interposition library in $(TAU_MPI_LIBS) uses these macros internally for logging messages.

TAU_DB_DUMP();

TAU_DB_DUMP macro dumps all profile data to disk and records a checkpoint or a snapshot of the profile statistics at that instant. The dump files are named dump.<node>.<context>.<thread>.

TAU_DB_DUMP_PREFIX(prefix);

```
Arguments:
char *prefix;
```

TAU_DB_DUMP_PREFIX macro dumps all profile data to disk and records a checkpoint or a snapshot of the profile statistics at that instant. The dump files are named <prefix>.<node>.<context>.<thread>. If prefix is “profile”, the files are named profile.0.0.0, etc. and may be read by paraprof/pprof tools as the application executes.

TAU_DB_DUMP_INCR();

This is similar to the TAU_DB_DUMP macro but it produces dump files that have a timestamp in their names. This allows the user to record timestamped incremental dumps as the application executes.

TAU_GET_FUNC_NAMES(functionList, numFuncs);

```
Arguments:
char **functionList;
int numFuncs;
```

This macro fills the funcList argument with the list of timer and routine names. It also records the number of routines active in the numFuncs argument.

TAU_DUMP_FUNC_NAMES();

This macro writes the names of active functions to a file named dump_functionnames_<node>.<context>.

TAU_GET_COUNTER_NAMES(counterList, numCounters);

```
Arguments:
```

```
char **counterList;
int numCounters;
```

TAU_GET_COUNTER_NAMES returns the list of counter names and the number of counters used for measurement. When wallclock time is used, the counter name of "default" is returned.

TAU_GET_FUNC_VALS(inFuncs, numRoutines, counterExclusiveValues, counterInclusiveValues, numCalls, numSubrs, counterNames, numOfCounters);

It gets detailed performance data for the list of routines. The user specifies inFuncs and the number of routines; TAU then returns the other arguments with the performance data. counterExclusiveValues and counterInclusiveValues are two dimensional arrays: the first dimension is the routine id and the second is counter id. The value is indexed by these two dimensions. numCalls and numSubrs (or child routines) are one dimensional arrays.

Example:

```
const char **inFuncs;
/* The first dimension is functions, and the second
        dimension is counters */
double **counterExclusiveValues;
double **counterInclusiveValues;
int *numOfCalls;
int *numOfSubRoutines;
const char **counterNames;
int numOfCouns;

TAU_GET_FUNC_NAMES(functionList, numOfFunctions);

/* We are only interested in the first two routines
        that are executing in
   this context. So, we allocate space for two routine
        names and get the
```

```
   performance data for these two routines at runtime.
         */
if(numOfFunctions >=2 ){
  inFuncs = (const char **) malloc(sizeof(const char
         *) * 2);

  inFuncs[0] = functionList[0];
  inFuncs[1] = functionList[1];

  //Just to show consistency.
  TAU_DB_DUMP();

  TAU_GET_FUNC_VALS(inFuncs, 2,
                    counterExclusiveValues,
                    counterInclusiveValues,
                    numOfCalls,
                    numOfSubRoutines,
                    counterNames,
                    numOfCouns);

  TAU_DUMP_FUNC_VALS_INCR(inFuncs, 2);


cout << "@@@@@@@@@@@@@@@" << endl;
cout << "The number of counters is: " << numOfCouns <<
        endl;
cout << "The first counter is: " << counterNames[0] <<
        endl;

cout << "The Exclusive value of: " << inFuncs[0]
     << " is: " << counterExclusiveValues[0][0] <<
        endl;
cout << "The numOfSubRoutines of: " << inFuncs[0]
     << " is: " << numOfSubRoutines[0]
     << endl;


cout << "The Inclusive value of: " << inFuncs[1]
     << " is: " << counterInclusiveValues[1][0]
     << endl;
cout << "The numOfCalls of: " << inFuncs[1]
```

```
      << " is: " << numOfCalls[1]
      << endl;

  cout << "@@@@@@@@@@@@@@@@" << endl;
  }

  TAU_DB_DUMP_INCR();
```

TAU_DUMP_FUNC_VALS(inFuncs, numFuncs);

```
Arguments:
char **inFuncs;
int numFuncs;
```

`TAU_DUMP_FUNC_VALS` writes the data associated with the routines listed in inFuncs to disk. The number of routines is specified by the user in numFuncs.

TAU_DUMP_FUNC_VALS_INCR(inFuncs, numFuncs);

```
Arguments:
char **inFuncs;
int numFuncs;
```

Similar to `TAU_DUMP_FUNC_VALS`. This macro creates an incremental selective dump and dumps the results with a date stamp to the filename such as sel_dump__Thu-Mar-28-16:30:48-2002__.0.0.0. In this manner the previous `TAU_DUMP_FUNC_VALS_INCR(...)` are not overwritten (unless they occur within a second).

TAU Mapping API

TAU allows the user to map performance data of entities from one layer to another in multi-layered software. Mapping is used in profiling (and tracing) both synchronous and asynchronous models of computation. For mapping, the following macros are used. First locate and identify the higher-level statement using the `TAU_MAPPING` macro. Then, associate a function identifier with it using the `TAU_MAPPING_OBJECT`. Associate the high level statement to a FunctionInfo object that will be visible to lower level code, using `TAU_MAPPING_LINK`, and then profile entire blocks using `TAU_MAPPING_PROFILE`. Independent sets of statements can be profiled using `TAU_MAPPING_PROFILE_TIMER`, `TAU_MAPPING_PROFILE_START`, and `TAU_MAPPING_PROFILE_STOP` macros using the FunctionInfo object. The TAU `examples/mapping` directory has two examples (`embedded` and `external`) that illustrate the use of this mapping API for generating object-oriented profiles.

TAU_MAPPING(statement, key);

```
Arguments:
statement ; // any C++ statement
TauGroup_t key; // TAU group/unique key associated
```

`TAU_MAPPING` is used to encapsulate the C++ statement that we want to map to some other layer. The other layer can execute synchronously or asynchronously with respect to this statement. The key corresponds to a number that the lower layer will use to refer to this statement. For example,

```
int main()
{
  Array <2> A(N, N), B(N, N), C(N,N), D(N, N);
  //Original statement:
  A = B + C + D;
  //Instrumented statement:
  TAU_MAPPING(A = B + C + D; , TAU_USER);
...
}
```

TAU_MAPPING_CREATE(name, type, key, groupname, tid);

```
Arguments:
char *name, type, groupname;
TauGroup_t key; // TAU group/unique key associated
int tid; // Thread id
```

TAU_MAPPING_CREATE is similar to TAU_MAPPING but it requires the name, type and group name parameters (as character strings) to be specified. It creates a mapping and associates it with the key that is specified. Later, this key may be specified to retrieve the FunctionInfo object associated with this key for timing purposes. The thread identifier is specified in the tid parameter.

Example:

```
TAU_MAPPING_CREATE("foo()", "void ()",
function_id,"USER", tid);
```

TAU_MAPPING_OBJECT(FuncIdVar);

```
Arguments: FunctionInfo *FuncIdVar;
```

To create storage for an identifier associated with a higher level statement that is mapped using TAU_MAPPING, we use the TAU_MAPPING_OBJECT macro. For example, in the TAU_MAPPING example, the array expressions are created into objects of a class ExpressionKernel, and each statement is an object that is an instance of this class. To embed the identity of the statement we store the mapping object in a data field in this class. This is shown below:

```
 template<class LHS,class Op,class RHS,class EvalTag>
class ExpressionKernel : public Pooma::Iterate_t
{
public:

  typedef ExpressionKernel<LHS,Op,RHS,EvalTag> This_t;
  //
  // Construct from an Expr.
```

```
  // Build the kernel that will evaluate the expression
on the given domain.
  // Acquire locks on the data referred to by the
expression.
  //
  ExpressionKernel(const LHS&,const Op&,const
RHS&,Pooma::Scheduler_t&);


  virtual ~ExpressionKernel();

  //
  // Do the loop.
  //
  virtual void run();

private:

  // The expression we will evaluate.
  LHS lhs_m;
  Op  op_m;
  RHS rhs_m;
  TAU_MAPPING_OBJECT(TauMapFI)
};
```

TAU_MAPPING_LINK(FuncIdVar, Key);

```
Arguments: FunctionInfo *FuncIdVar;
TauGroup_t Key;
```

TAU_MAPPING_LINK creates a link between the object defined in TAU_MAPPING_OBJECT (that identifies a statement) and the actual higher-level statement that is mapped with TAU_MAPPING. The Key argument represents a profile group to which the statement belongs, as specified in the TAU_MAPPING macro argument. For the example of array statements, this link should be created in the constructor of the class that represents the expression. TAU_MAPPING_LINK should be executed before any measurement takes place. It assigns the identifier of the statement to the object to which FuncIdVar refers. For example

```
//
// Constructor
// Input an expression and record it for later use.
//
template<class LHS,class Op,class RHS,class EvalTag>
ExpressionKernel<LHS,Op,RHS,EvalTag>::
ExpressionKernel(const LHS& lhs,const Op& op,const
    RHS&  rhs,      Pooma::Scheduler_t& scheduler) :
    Pooma::Iterate_t(scheduler,        forEachTag(Make-
    Expression<LHS>::make(lhs),  DataBlockTag<Count-
    Blocks>(),SumCombineTag()) +
    forEachTag(MakeExpression<RHS>::make(rhs), Dat-
    aBlockTag<CountBlocks>(),SumCombineTag()), -1),
    lhs_m(lhs), op_m(op), rhs_m(rhs)
{
  TAU_MAPPING_LINK(TauMapFI, TAU_USER)
// .. rest of the constructor
}
```

TAU_MAPPING_PROFILE (FuncIdVar);

`Arguments; FunctionInfo *FuncIdVar;`

The `TAU_MAPPING_PROFILE` macro measures the time and attributes it to the statement mapped in `TAU_MAPPING` macro. It takes as its argument the identifier of the higher level statement that is stored using `TAU_MAPPING_OBJECT` and linked to the statement using `TAU_MAPPING_LINK` macros. `TAU_MAPPING_PROFILE` measures the time spent in the entire block in which it is invoked. For example, if the time spent in the run method of the class does work that must be associated with the higher-level array expression, then, we can instrument it as follows:

```
//
// Evaluate the kernel
// Just tell an InlineEvaluator to do it.
//

template<class LHS,class Op,class RHS,class EvalTag>
void
```

```
ExpressionKernel<LHS,Op,RHS,EvalTag>::run()
{
  TAU_MAPPING_PROFILE(TauMapFI)

  // Just evaluate the expression.
  KernelEvaluator<EvalTag>().evalate(lhs_m,op_m,rhs_m);
  // we could release the locks here or in dtor
}
```

TAU_MAPPING_PROFILE_TIMER(timer, FuncIdVar);

```
Arguments: Profiler timer;
FunctionInfo * FuncIdVar;
```

TAU_MAPPING_PROFILE_TIMER enables timing of individual statements, instead of complete blocks. It will attribute the time to a higher-level statement. The second argument is the identifier of the statement that is obtained after TAU_MAPPING_OBJECT and TAU_MAPPING_LINK have executed. The timer argument in this macro is any variable that is used subsequently to start and stop the timer.

TAU_MAPPING_PROFILE_START(timer, tid);

```
Argument:
Profiler timer;
int tid;
```

TAU_MAPPING_PROFILE_START starts the timer that is created using TAU_MAPPING_PROFILE_TIMER. This will measure the elapsed time in groups of statements, instead of the entire block. A corresponding stop statement stops the timer as described next. The thread identifier is specified in the tid parameter.

TAU_MAPPING_PROFILE_STOP(tid);

```
Arguments:
int tid;
```

TAU_MAPPING_PROFILE_STOP stops the timer associated with the mapped lower-level statements. This is used in conjunction with TAU_MAPPING_PROFILE_TIMER and TAU_MAPPING_PROFILE_START macros. Example:

```
template<class LHS,class Op,class RHS,class EvalTag>
void
ExpressionKernel<LHS,Op,RHS,EvalTag>::run()
{
  TAU_MAPPING_PROFILE_TIMER(timer, TauMapFI);
  printf("ExpressionKernel::run() this = %x\n", this);
  // Just evaluate the expression.

  TAU_MAPPING_PROFILE_START(timer);
      KernelEvaluator<EvalTag>().evaluate(lhs_m, op_m,
          rhs_m);
  TAU_MAPPING_PROFILE_STOP();
  // we could release the locks here instead of in the
dtor.
}
```

This concludes our Mapping section.

C Measurement API

The API for instrumenting C source code is similar to the C++ API. The difference is that `the TAU_PROFILE()` macro is not available for identifying an entire block of code or function. Instead, routine transitions are explicitly specified using `TAU_PROFILE_TIMER()` macro with `TAU_PROFILE_START()` and `TAU_PROFILE_STOP()` macros to indicate the entry and exit from a routine.

Note that, TAU_TYPE_STRING() and CT() macros are not applicable for C. It is important to declare the TAU_PROFILE_TIMER() macro after all the variables have been declared in the function and before the execution of the first C statement.

Example:

```
int main (int argc, char **argv)
{
  int ret;
  pthread_attr_t  attr;
  pthread_t       tid;
  TAU_PROFILE_TIMER(tautimer,"main()", "int (int, char
     **)", TAU_DEFAULT);
  TAU_PROFILE_START(tautimer);
  TAU_PROFILE_INIT(argc, argv);
  TAU_PROFILE_SET_NODE(0);

  pthread_attr_init(&attr);
  printf("Started Main...\n");
  // other statements
  TAU_PROFILE_STOP(tautimer);
  return 0;
}
```

Fortran90 Measurement API

The Fortran90 TAU API allows source code written in Fortran to be instrumented for TAU. This API is comprised of Fortran routines. As explained in Chapter 2, the instrumentation can be disabled in the program by using the TAU stub makefile variable TAU_DISABLE on the link command line. This points to a library that contains empty TAU instrumentation routines.

TAU_PROFILE_INIT()

TAU_PROFILE_INIT routine must be called before any other TAU instrumentation routines. It is called once, in the top level routine (program). It initializes the TAU library.

Example:

```
      PROGRAM SUM_OF_CUBES
      integer profiler(2)
      save profiler

       call TAU_PROFILE_INIT()
```

TAU_PROFILE_TIMER(profiler, name)

```
Arguments:
integer profiler(2)
character name(size)
```

To profile a block of Fortran code, such as a function, subroutine, loop etc., the user must first declare a profiler, which is an integer array of two elements (pointer) with the save attribute, and pass it as the first parameter to the TAU_PROFILE_TIMER subroutine. The second parameter must contain the name of the routine, which is enclosed in a single quote. TAU_PROFILE_TIMER declares the profiler that must be used to profile a block of code. The profiler is used to profile the statements using TAU_PROFILE_START and TAU_PROFILE_STOP as explained later. Example:

```
subroutine bcast_inputs
implicit none
integer profiler(2)
save profiler

include 'mpinpb.h'
include 'applu.incl'

integer IERR

call TAU_PROFILE_TIMER(profiler, 'bcast_inputs')
```

TAU_PROFILE_START(profiler)

```
Arguments:
```

```
integer profiler(2)
```

TAU_PROFILE_START starts the timer for profiling a set of statements. The timer (or the profiler) must be declared using TAU_PROFILE_TIMER routine, prior to using TAU_PROFILE_START.

TAU_PROFILE_STOP(profiler)

```
Arguments:
integer profiler(2)
```

TAU_PROFILE_STOP stops the timer used to profile a set of statements. It is used in conjunction with TAU_PROFILE_TIMER and TAU_PROFILE_START sub-routines.

Example:

```
subroutine setbv
implicit none

include 'applu.incl'
c---------------------------------------------------
c   local variables
c---------------------------------------------------
integer profiler(2)
save profiler
integer i, j, k
integer iglob, jglob

      call TAU_PROFILE_TIMER(profiler, 'setbv')
      call TAU_PROFILE_START(profiler)
c   set the dependent variable values along the top and
c   bottom faces
      do j = 1, ny
       jglob = jpt + j
       do i = 1, nx
         iglob = ipt + i
```

```
      call exact( iglob, jglob, 1, u( 1, i, j, 1 ) )
call exact( iglob, jglob, nz, u( 1, i, j, nz ))
        end do
      end do
      call TAU_PROFILE_STOP(profiler)
      return
      end
```

TAU_PROFILE_SET_NODE(myNode)

```
Arguments:
integer myNode
```

The TAU_PROFILE_SET_NODE macro sets the node identifier of the executing task for profiling and tracing. Tasks are identified using node, context and thread ids. The profile data files generated will accordingly be named profile.<node>.<context>.<thread>.

TAU_PROFILE_SET_CONTEXT(myContext)

```
Argument:
integer myContext
```

TAU_PROFILE_SET_CONTEXT sets the context parameter of the executing task for profiling and tracing purposes. This is similar to setting the node parameter with TAU_PROFILE_SET_NODE.

TAU_PROFILE_REGISTER_THREAD()

To register a thread with the profiling system, invoke the TAU_PROFILE_REGISTER_THREAD routine in the run method of the thread prior to executing any other TAU routine. This sets up thread identifiers that are later used by the instrumentation system.

TAU_DISABLE_INSTRUMENTATION()

TAU_DISABLE_INSTRUMENTATION macro disables all entry/exit instrumentation within all threads of a context. This allows the user to selectively enable and disable instrumentation in parts of his/her code. It is important to re-enable the instrumentation within the same basic block.

TAU_ENABLE_INSTRUMENTATION()

TAU_ENABLE_INSTRUMENTATION macro re-enables all TAU instrumentation. All instances of functions and statements that occur between the disable/enable section are ignored by TAU. This allows a user to limit the trace size, if the macros are used to disable recording of a set of iterations that have the same characteristics as, for example, the first recorded instance.

Example:

```
call TAU_DISABLE_INSTRUMENTATION()
...
call TAU_ENABLE_INSTRUMENTATION()
```

TAU_PROFILE_EXIT(message)

```
Argument:
character message(size)
```

`TAU_PROFILE_EXIT` should be called prior to an error exit from the program so that any profiles or event traces can be dumped to disk before quitting.

Example:

```
call TAU_PROFILE_EXIT('abort called')
```

TAU_REGISTER_EVENT(variable, event_name)

```
Arguments:
int variable(2)
character event_name(size)
```

TAU can profile user-defined events using `TAU_REGISTER_EVENT`. The meaning of the event is determined by the user. The first argument to `TAU_REGISTER_EVENT` is the pointer to an integer array. This array is declared with a save attribute as shown below.

Example:

```
integer eventid(2)
save eventid
call TAU_REGISTER_EVENT(eventid, 'Error in Iteration')
```

TAU_EVENT(variable, value)

```
Arguments:
integer variable(2)
real value
```

`TAU_EVENT` associates a value with some user-defined event. When the event is triggered and this macro is executed, TAU maintains statistics, such as maximum, minimum values, standard deviation, number of samples, etc. for tracking this event.

Example:

```
call TAU_REGISTER_EVENT(taumsgsize, 'Message size')
call TAU_EVENT(size)
```

TAU_REPORT_STATISTICS()

TAU_REPORT_STATISTICS prints the aggregate statistics of user events across all threads in each node. Typically, this should be called just before the main thread exits.

TAU_REPORT_THREAD_STATISTICS()

TAU_REPORT_THREAD_STATISTICS prints the aggregate, as well as per thread user event statistics. Typically, this should be called just before the main thread exits.

TAU_TRACE_SENDMSG(tag, destination, length)

```
Arguments:
integer tag
integer destination
integer length
```

TAU_TRACE_SENDMSG traces an inter-process message communication when a tagged message is sent to a destination process.

TAU_TRACE_RECVMSG(tag, source, length)

```
Arguments:
integer tag
integer source
integer length
```

TAU_TRACE_RECVMSG traces a receive operation where tag represents the type of the message received from the source process.

Summary

In C++, a single macro `TAU_PROFILE`, is sufficient to profile a block of statements. In C and Fortran, the user must use statement level timers to achieve this, using `TAU_PROFILE_TIMER`, `TAU_PROFILE_START` and `TAU_PROFILE_STOP`. Instrumentation of C++ source code can be done manually or by using tau_instrumentor, a tool that can automatically insert TAU annotations in the source code. Implementation of a Fortran 90 instrumentor is in progress.

CHAPTER 4 *Profiling*

This chapter describes running an instrumented application and the generation and subsequent analysis of profile data. Profiling shows the summary statistics of performance metrics that characterize application performance behavior. Examples of performance metrics are the CPU time associated with a routine, the count of the secondary data cache misses associated with a group of statements, the number of times a routine executes, etc.

Running the application

After instrumentation and compilation are completed, the profiled application is run to generate the profile data files. These files can be stored in a directory specified by the environment variable PROFILEDIR as explained in Chapter 2. By default, all instrumented routines and statements are measured. To selectively measure groups of routines and statements, we can use the command-line parameter `--profile` to specify the statements to be profiled. Example:

```
% setenv PROFILEDIR /home/sameer/profiledata/
experiment55
% mpirun -np 4 matrix
```

This profiles all routines

```
% mpirun -np 4 matrix --profile io+field+2
```

The above profiles routines belonging to `TAU_IO`, `TAU_FIELD` and `TAU_USER2` profile groups. For a detailed list of groups, please refer to [TAU-PGROUPS-URL]

Running an application using DynInstAPI

Install DynInstAPI package and refer to the installed directory while configuring TAU. Use **tau_run**, a tool that instruments the application at runtime.

The commandline options accepted by tau_run are:

```
Usage: tau_run [-Xrun<Taulibrary> ][-v][-o outfile] [-f
     <instrumentation file> ] <application> [args]
```

By default, libTAU.so is loaded by tau_run. However, the user can override this and specify another file using the -Xrun<Taulibrary>. In this case lib<Taulibrary>.so will be loaded using LD_LIBRARY_PATH. The -f <instrumentation file> option can be used to specify an exclude/include list of routines and/or files for instrumentation. The list of routines to be excluded from instrumentation is specified, one per line, enclosed by BEGIN_EXCLUDE_LIST and END_EXCLUDE_LIST. Instead of specifying which routines should be excluded, the user can specify the list of routines that are to be instrumented using the include list, one routine name per line, enclosed by BEGIN_INCLUDE_LIST and END_INCLUDE_LIST. Files are

specified using the BEGIN_FILE_INCLUDE_LIST/END_FILE_INCLUDE_LIST and BEGIN_FILE_EXCLUDE_LIST/END_FILE_EXCLUDE_LIST tags. Wildcards * and ? may be used while specifying file names.

Example:

```
BEGIN_EXCLUDE_LIST
void quicksort(int *, int, int)
void sort_5elements(int *)
void interchange(int *, int *)
END_EXCLUDE_LIST

BEGIN_FILE_EXCLUDE_LIST
*.so
END_FILE_EXCLUDE_LIST
```

To use tau_run, TAU is configured with DyninstAPI as shown below:

```
% configure –dyninst=/usr/local/packages/dyninstAPI
% make install
% cd tau/examples/dyninst
% make install
% tau_run klargest 2500 23
% pprof; paraprof
```

Support for new platforms and compilers is being added and this DyninstAPI option is experimental for now.

Using Hardware Performance Counters

Performance counters exist on modern microprocessors. These count hardware performance events such as cache misses, floating point operations, etc. while the program executes on the processor. The Performance Data Standard and API (PAPI, [PAPI-URL]) and Performance Counter Library (PCL, [PCL-URL]) packages provide a uniform interface to access these performance counters. TAU can use either PAPI or PCL to access these hardware performance counters. To do so, download and install PAPI or PCL. Then, configure TAU using the -pcl=<dir> or -papi=<dir> configuration command-line option to specify the location of PCL or PAPI. Build TAU and applications as you normally would (as described in Chapters 2 and 3).

While running the application, set the environment variable **PCL_EVENT** or **PAPI_EVENT** respectively, to specify which hardware performance counter TAU should use while profiling the application. For example to measure the floating point operations in routines using PCL,

```
% ./configure -pcl=/usr/local/packages/pcl-1.2
% setenv PCL_EVENT PCL_FP_INSTR
% mpirun -np 8 application
```

TABLE 2. Events measured by setting the environment variable `PCL_EVENT` in TAU

PCL_EVENT	Event Measured
PCL_L1CACHE_READ	L1 (Level one) cache reads
PCL_L1CACHE_WRITE	L1 cache writes
PCL_L1CACHE_READWRITE	L1 cache reads and writes
PCL_L1CACHE_HIT	L1 cache hits
PCL_L1CACHE_MISS	L1 cache misses
PCL_L1DCACHE_READ	L1 data cache reads
PCL_L1DCACHE_WRITE	L1 data cache writes
PCL_L1DCACHE_READWRITE	L1 data cache reads and writes
PCL_L1DCACHE_HIT	L1 data cache hits
PCL_L1DCACHE_MISS	L1 data cache misses
PCL_L1ICACHE_READ	L1 instruction cache reads
PCL_L1ICACHE_WRITE	L1 instruction cache writes
PCL_L1ICACHE_READWRITE	L1 instruction cache reads and writes
PCL_L1ICACHE_HIT	L1 instruction cache hits
PCL_L1ICACHE_MISS	L1 instruction cache misses
PCL_L2CACHE_READ	L2 (Level two) cache reads
PCL_L2CACHE_WRITE	L2 cache writes
PCL_L2CACHE_READWRITE	L2 cache reads and writes
PCL_L2CACHE_HIT	L2 cache hits
PCL_L2CACHE_MISS	L2 cache misses
PCL_L2DCACHE_READ	L2 data cache reads
PCL_L2DCACHE_WRITE	L2 data cache writes
PCL_L2DCACHE_READWRITE	L2 data cache reads and writes

TABLE 2. Events measured by setting the environment variable `PCL_EVENT` in TAU

PCL_EVENT	Event Measured
PCL_L2DCACHE_HIT	L2 data cache hits
PCL_L2DCACHE_MISS	L2 data cache misses
PCL_L2ICACHE_READ	L2 instruction cache reads
PCL_L2ICACHE_WRITE	L2 instruction cache writes
PCL_L2ICACHE_READWRITE	L2 instruction cache reads and writes
PCL_L2ICACHE_HIT	L2 instruction cache hits
PCL_L2ICACHE_MISS	L2 instruction cache misses
PCL_TLB_HIT	TLB (Translation Lookaside Buffer) hits
PCL_TLB_MISS	TLB misses
PCL_ITLB_HIT	Instruction TLB hits
PCL_ITLB_MISS	Instruction TLB misses
PCL_DTLB_HIT	Data TLB hits
PCL_DTLB_MISS	Data TLB misses
PCL_CYCLES	Cycles
PCL_ELAPSED_CYCLES	Cycles elapsed
PCL_INTEGER_INSTR	Integer instructions executed
PCL_FP_INSTR	Floating point (FP) instructions executed
PCL_LOAD_INSTR	Load instructions executed
PCL_STORE_INSTR	Store instructions executed
PCL_LOADSTORE_INSTR	Loads and stores executed
PCL_INSTR	Instructions executed
PCL_JUMP_SUCCESS	Successful jumps executed
PCL_JUMP_UNSUCCESS	Unsuccessful jumps executed
PCL_JUMP	Jumps executed
PCL_ATOMIC_SUCCESS	Successful atomic instructions executed
PCL_ATOMIC_UNSUCCESS	Unsuccessful atomic instructions executed
PCL_ATOMIC	Atomic instructions executed
PCL_STALL_INTEGER	Integer stalls
PCL_STALL_FP	Floating point stalls

TABLE 2. Events measured by setting the environment variable `PCL_EVENT` in TAU

PCL_EVENT	Event Measured
PCL_STALL_JUMP	Jump stalls
PCL_STALL_LOAD	Load stalls
PCL_STALL_STORE	Store Stalls
PCL_STALL	Stalls
PCL_MFLOPS	Millions of floating point operations/second
PCL_IPC	Instructions executed per cycle
PCL_L1DCACHE_MISSRATE	Level 1 data cache miss rate
PCL_L2DCACHE_MISSRATE	Level 2 data cache miss rate
PCL_MEM_FP_RATIO	Ratio of memory accesses to FP operations

To select floating point instructions for profiling using PAPI, you would:

```
% configure -papi=/usr/local/packages/papi-2.3
% make clean install
% cd examples/papi
% setenv PAPI_EVENT PAPI_FP_INS
% a.out
```

TABLE 3. Events measured by setting the environment variable PAPI_EVENT in TAU

PAPI_EVENT	Event Measured
PAPI_L1_DCM	Level 1 data cache misses
PAPI_L1_ICM	Level 1 instruction cache misses
PAPI_L2_DCM	Level 2 data cache misses
PAPI_L2_ICM	Level 2 instruction cache misses
PAPI_L3_DCM	Level 3 data cache misses
PAPI_L3_ICM	Level 3 instruction cache misses
PAPI_L1_TCM	Level 1 total cache misses

TABLE 3. Events measured by setting the environment variable PAPI_EVENT in TAU

PAPI_EVENT	Event Measured
PAPI_L2_TCM	Level 2 total cache misses
PAPI_L3_TCM	Level 3 total cache misses
PAPI_CA_SNP	Snoops
PAPI_CA_SHR	Request for access to shared cache line (SMP)
PAPI_CA_CLN	Request for access to clean cache line (SMP)
PAPI_CA_INV	Cache Line Invalidation (SMP)
PAPI_CA_ITV	Cache Line Intervention (SMP)
PAPI_L3_LDM	Level 3 load misses
PAPI_L3_STM	Level 3 store misses
PAPI_BRU_IDL	Cycles branch units are idle
PAPI_FXU_IDL	Cycles integer units are idle
PAPI_FPU_IDL	Cycles floating point units are idle
PAPI_LSU_IDL	Cycles load/store units are idle
PAPI_TLB_DM	Data translation lookaside buffer misses
PAPI_TLB_IM	Instruction translation lookaside buffer misses
PAPI_TLB_TL	Total translation lookaside buffer misses
PAPI_L1_LDM	Level 1 load misses
PAPI_L1_STM	Level 1 store misses
PAPI_L2_LDM	Level 2 load misses
PAPI_L2_STM	Level 2 store misses
PAPI_BTAC_M	BTAC miss
PAPI_PRF_DM	Prefetch data instruction caused a miss
PAPI_L3_DCH	Level 3 Data Cache Hit
PAPI_TLB_SD	Translation lookaside buffer shootdowns (SMP)
PAPI_CSR_FAL	Failed store conditional instructions
PAPI_CSR_SUC	Successful store conditional instructions
PAPI_CSR_TOT	Total store conditional instructions
PAPI_MEM_SCY	Cycles Stalled Waiting for Memory Access
PAPI_MEM_RCY	Cycles Stalled Waiting for Memory Read

TABLE 3. Events measured by setting the environment variable PAPI_EVENT in TAU

PAPI_EVENT	Event Measured
PAPI_MEM_WCY	Cycles Stalled Waiting for Memory Write
PAPI_STL_ICY	Cycles with No Instruction Issue
PAPI_FUL_ICY	Cycles with Maximum Instruction Issue
PAPI_STL_CCY	Cycles with No Instruction Completion
PAPI_FUL_CCY	Cycles with Maximum Instruction Completion
PAPI_HW_INT	Hardware interrupts
PAPI_BR_UCN	Unconditional branch instructions executed
PAPI_BR_CN	Conditional branch instructions executed
PAPI_BR_TKN	Conditional branch instructions taken
PAPI_BR_NTK	Conditional branch instructions not taken
PAPI_BR_MSP	Conditional branch instructions mispredicted
PAPI_BR_PRC	Conditional branch instructions correctly predicted
PAPI_FMA_INS	FMA instructions completed
PAPI_TOT_IIS	Total instructions issued
PAPI_TOT_INS	Total instructions executed
PAPI_INT_INS	Integer instructions executed
PAPI_FP_INS	Floating point instructions executed
PAPI_LD_INS	Load instructions executed
PAPI_SR_INS	Store instructions executed
PAPI_BR_INS	Total branch instructions executed
PAPI_VEC_INS	Vector/SIMD instructions executed
PAPI_FLOPS	Floating Point Instructions executed per second
PAPI_RES_STL	Cycles processor is stalled on resource
PAPI_FP_STAL	FP units are stalled
PAPI_TOT_CYC	Total cycles
PAPI_IPS	Instructions executed per second
PAPI_LST_INS	Total load/store instructions executed
PAPI_SYC_INS	Synchronization instructions executed
PAPI_L1_DCH	L1 D Cache Hit

TABLE 3. Events measured by setting the environment variable PAPI_EVENT in TAU

PAPI_EVENT	Event Measured
PAPI_L2_DCH	L2 D Cache Hit
PAPI_L1_DCA	L1 D Cache Access
PAPI_L2_DCA	L2 D Cache Access
PAPI_L3_DCA	L3 D Cache Access
PAPI_L1_DCR	L1 D Cache Read
PAPI_L2_DCR	L2 D Cache Read
PAPI_L3_DCR	L3 D Cache Read
PAPI_L1_DCW	L1 D Cache Write
PAPI_L2_DCW	L2 D Cache Write
PAPI_L3_DCW	L3 D Cache Write
PAPI_L1_ICH	L1 instruction cache hits
PAPI_L2_ICH	L2 instruction cache hits
PAPI_L3_ICH	L3 instruction cache hits
PAPI_L1_ICA	L1 instruction cache accesses
PAPI_L2_ICA	L2 instruction cache accesses
PAPI_L3_ICA	L3 instruction cache accesses
PAPI_L1_ICR	L1 instruction cache reads
PAPI_L2_ICR	L2 instruction cache reads
PAPI_L3_ICR	L3 instruction cache reads
PAPI_L1_ICW	L1 instruction cache writes
PAPI_L2_ICW	L2 instruction cache writes
PAPI_L3_ICW	L3 instruction cache writes
PAPI_L1_TCH	L1 total cache hits
PAPI_L2_TCH	L2 total cache hits
PAPI_L3_TCH	L3 total cache hits
PAPI_L1_TCA	L1 total cache accesses
PAPI_L2_TCA	L2 total cache accesses
PAPI_L3_TCA	L3 total cache accesses
PAPI_L1_TCR	L1 total cache reads

TABLE 3. Events measured by setting the environment variable PAPI_EVENT in TAU

PAPI_EVENT	Event Measured
PAPI_L2_TCR	L2 total cache reads
PAPI_L3_TCR	L3 total cache reads
PAPI_L1_TCW	L1 total cache writes
PAPI_L2_TCW	L2 total cache writes
PAPI_L3_TCW	L3 total cache writes
PAPI_FML_INS	FM ins
PAPI_FAD_INS	FA ins
PAPI_FDV_INS	FD ins
PAPI_FSQ_INS	FSq ins
PAPI_FNV_INS	Finv ins

Using Multiple Hardware Counters for Measurement

TAU can be configured to record more than one hardware performance counter, along with time for each timer and routine. To use this feature, TAU is configured with the -MULTIPLECOUNTERS option. Example:

```
%./configure –MULTIPLECOUNTERS –LINUXTIMERS –CPUTIME –
      papi=/tools/papi-2.3
```

LIST OF COUNTERS:

Set the following values for the COUNTER<1-25> environment variables.

GET_TIME_OF_DAY --- For the default profiling option using gettimeofday()

SGI_TIMERS --- For -SGITIMERS configuration option under IRIX

CRAY_TIMERS --- For -CRAYTIMERS configuration option under Cray X1.

LINUX_TIMERS --- For -LINUXTIMERS configuration option under Linux

CPU_TIME --- For user+system time from getrusage() call with -CPUTIME

P_WALL_CLOCK_TIME --- For PAPI's WALLCLOCK time using -PAPI-WALLCLOCK

P_VIRTUAL_TIME --- For PAPI's process virtual time using -PAPIVIRTUAL

TAU_MUSE --- For reading counts of Linux OS kernel level events when MAGNET/MUSE is installed and -muse configuration option is enabled. [MUSE-URL]. TAU_MUSE_PACKAGE environment variable has to be set to package name (busy_time, count, etc.)

TAU_MPI_MESSAGE_SIZE --- For tracking message sizes sent by a node for each routine.

and PAPI/PCL options that can be found in Tables 2 & 3. Example:

PCL_FP_INSTR --- For floating point operations using PCL (-pcl=<dir>)

PAPI_FP_INS --- For floating point operations using PAPI (-papi=<dir>)

NOTE: When -MULTIPLECOUNTERS is used with -TRACE option, the tracing library uses the wallclock time from the function specified in the COUNTER1 variable. This should typically point to wallclock time routines (such as GET_TIME_OF_DAY or SGI_TIMERS or LINUX_TIMERS).

Example:

```
% setenv COUNTER1   P_WALL_CLOCK_TIME
% setenv COUNTER2 PAPI_L1_DCM
% setenv COUNTER3 PAPI_FP_INS
```

will produce profile files in directories called MULT_P_WALL_CLOCK_TIME, MULTI__PAPI_L1_DCM, and MULTI_PAPI_FP_INS.

Running a JAVA application with TAU

Java applications are profiled/traced using the `-XrunTAU` command-line parameter as shown below:

```
% cd tau/examples/java/pi
% setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:/home/tau/
solaris2/lib
% java -XrunTAU Pi
```

Running the application generates profile files with names having the form profile.<node>.<context>.<thread>. These files can be analyzed using ***pprof*** or ***paraprof*** (see below).

Running a Python application with TAU

TAU can automatically instrument all Python routines when the **tau** python package is imported. To execute the program, tau.run routine is invoked with the name of the top level Python code. For e.g.,

```
#!/usr/bin/env python

import tau
from time import sleep

def f2():
    print "Inside f2: sleeping for 2 secs..."
    sleep(2)
def f1():
    print "Inside f1, calling f2..."
    f2()

def OurMain():
    f1()

tau.run('OurMain()')
```

instruments routines OurMain(), f1() and f2() although there are no instrumentation calls in the routines. To use this feature, TAU must be configured with the -pythoninc=<dir> option (and -pythonlib=<dir> if running under IBM). Before running the application, the environment variable PYTHONPATH should be set to

include the TAU library directory (where tau.py is stored). Manual instrumentation of Python sources is also possible using the Python API and the **pytau** package. For e.g.,

```
#!/usr/bin/env python

import pytau
from time import sleep

x = pytau.profileTimer("A Sleep for excl 5 secs")
y = pytau.profileTimer("B Sleep for excl 2 secs")
pytau.start(x)
print "Sleeping for 5 secs ..."
sleep(5)
pytau.start(y)
print "Sleeping for 2 secs ..."
sleep(2)
pytau.stop(y)
pytau.dbDump()
pytau.stop(x)
```

shows how two timers x and y are created and used. Note, multiple timers can be nested, but not overlapping. Overlapping timers are detected by TAU at runtime and flagged with a warning (as exclusive time is not defined when timers overlap).

pprof

pprof sorts and displays profile data generated by TAU. To view the profile, merely execute pprof in the directory where profile files are located (or set the `PROFILEDIR` environment variable).

```
% pprof
```

Its usage is explained below:

```
usage: pprof [-c|-b|-m|-t|-e|-i] [-r] [-s] [-n num] [-f
filename] [-l] [node numbers]
  -c : Sort by number of Calls
  -b : Sort by number of suBroutines called by a func-
       tion
```

```
 -m : Sort by Milliseconds (exclusive time total)
 -t : Sort by Total milliseconds (inclusive time total)
      (DEFAULT)
 -e : Sort by Exclusive time per call (msec/call)
 -i : Sort by Inclusive time per call (total msec/call)
 -v : Sort by standard deViation (excl usec)
 -r : Reverse sorting order
 -s : print only Summary profile information
 -n num : print only first num functions
 -f filename : specify full path and Filename without
      node ids
 -l : List all functions and exit
node numbers : prints information about all contexts/
      threads for specified nodes
```

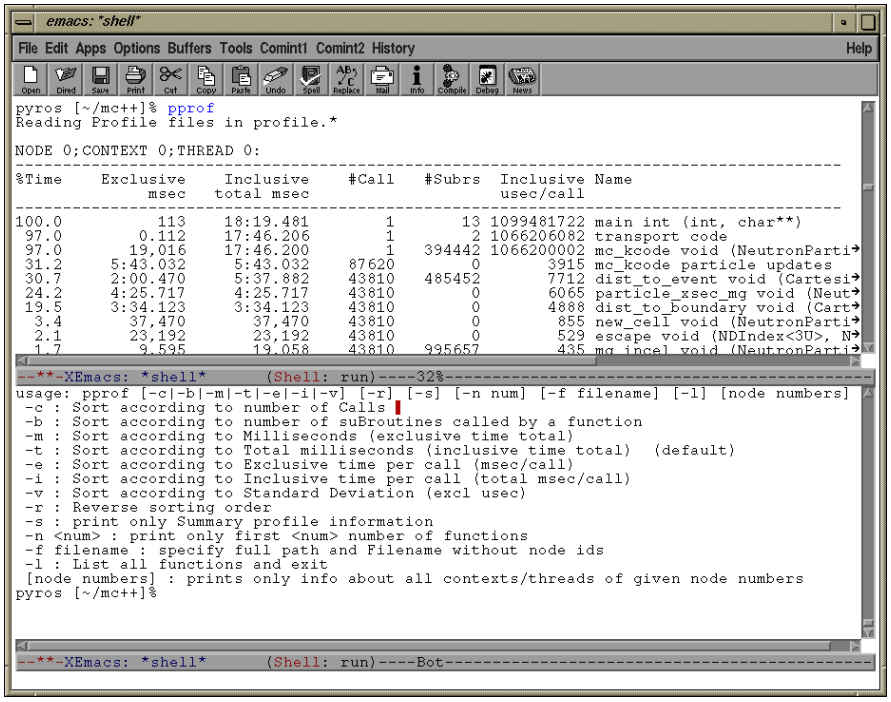


FIGURE 2. pprof in an xemacs window

paraprof

paraprof is the graphical interface to pprof. It shows the profile data in terms of histograms and text displays. paraprof requires that Java version1.2+ be installed and in the user's path. Invoke paraprof in the directory that contains the profile files.

```
% paraprof
```

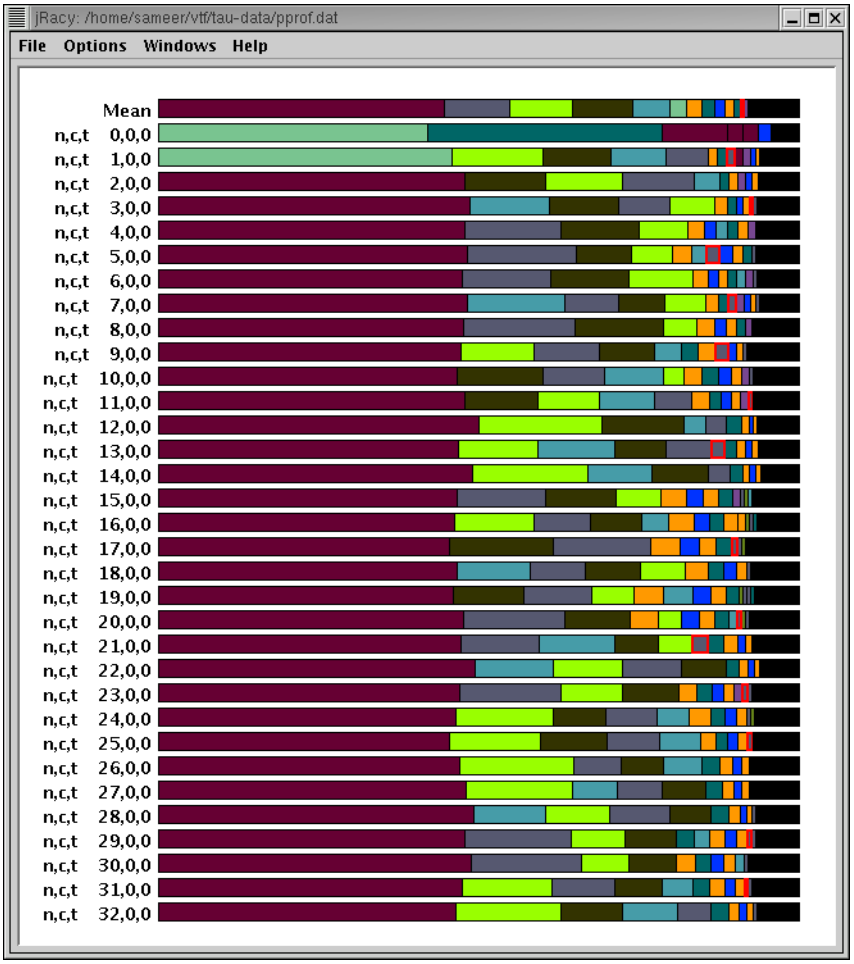


FIGURE 3. paraprof main window

This shows the relative time spent in each function as a horizontal bargraph. Each node, context, thread is represented as a horizontal bar with each function assigned a color. In this main paraprof window, click middle mouse button over say **n,c,t 32,0,0** to see the textual profile of node 32, context 0, thread 0 by selecting **Show Total Statistics Window**.

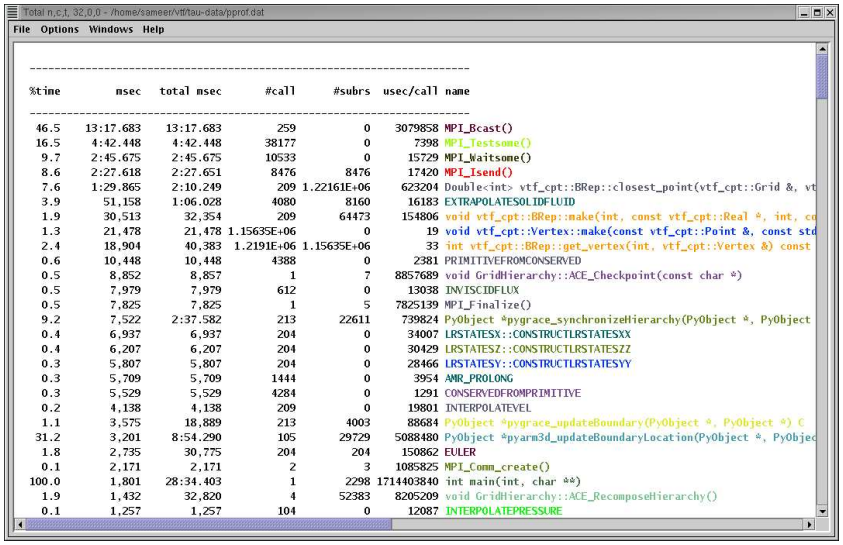


FIGURE 4. Text view of the detailed profile on n,c,t 32,0,0 sorted by exclusive time.

We see the display sorted by exclusive time. The text window shows the inclusive percentage (where main takes 100% inclusive time), the **msec** column shows the exclusive time in milliseconds, the **total msec** column shows the time taken in inclusive milliseconds (inclusive of all child routines called by a given routine). The **#call** column shows the number of calls for the given routine and **#subrs** shows the number of instrumented child subroutines called by a given routine. The **usec/call** column lists the inclusive time per call in microseconds. Finally, the **name** column shows the routine or timer for which the data is presented. This display is similar to the pprof display and the user can sort the performance metrics in a variety of ways. By choosing the **Options** menu item, the user can select the **Sort Order** (**Ascending** or **Descending**), **Select Metric** (**Inclusive**, **Exclusive**, **Number of Calls**, **Number of Subroutines**) or **Adjust** the paraprof colors. By selecting Inclusive as the metric, the routines are sorted by inclusive time as shown in the next figure.

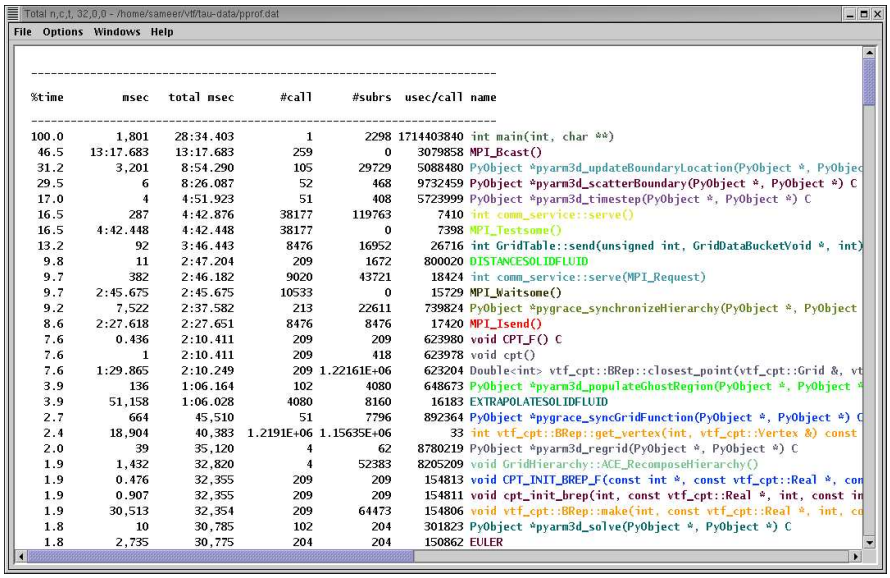


FIGURE 5. Performance data sorted by inclusive time

To see the relative function profile on one node, click the first mouse button on a node in the racy main window.

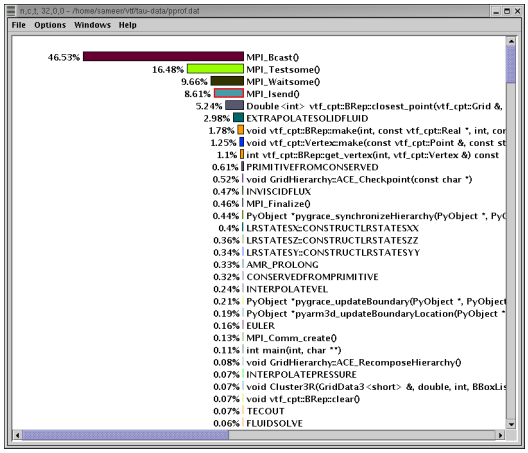


FIGURE 6. Node profile of node 32 sorted by exclusive time

Click, using the first or the third mouse button, on the name of a function, in the main paraprof window, to bring up the function window that shows the profile of the function over all nodes, contexts and threads.

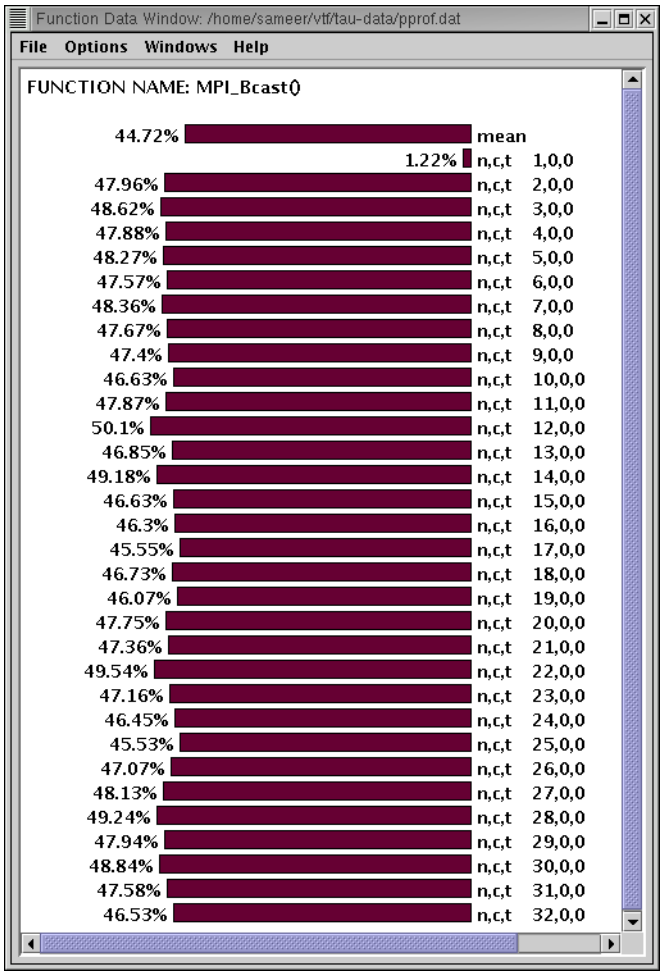


FIGURE 7. The function window shows the profile of the function over all nodes, contexts and threads.

By default exclusive percentages are displayed in this window. To see the actual time taken by the routine, select **Options** menu and choose **Select Value or Per-**

cent followed by **Value**. Then choose **Select Units** to **Seconds** to see the display in seconds. Other options include Select Metric (Exclusive, Inclusive, Number of Calls, Number of Subroutines). We see the window for **MPI_Bcast()** routine in the figure below.

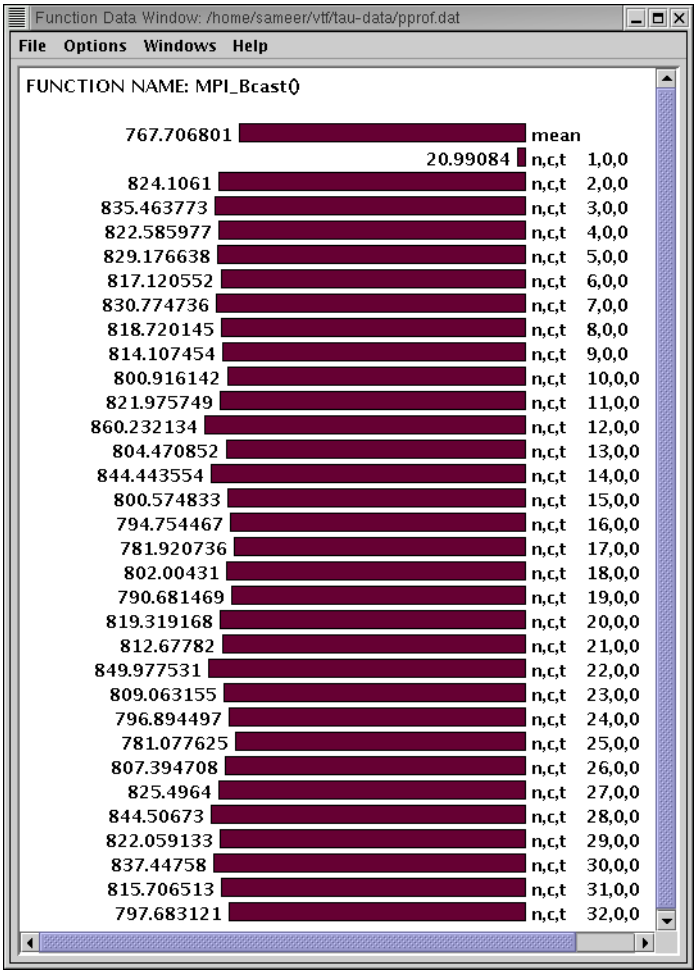


FIGURE 8. Exclusive time spent in seconds in MPI_Bcast over all nodes

By clicking the third mouse button over the colored bars, you can choose **Change Function Color**. The paraprof color selection window allows you to set the color of a function as by choosing the HSB, swatches or RGB tabs as shown in the figures below.

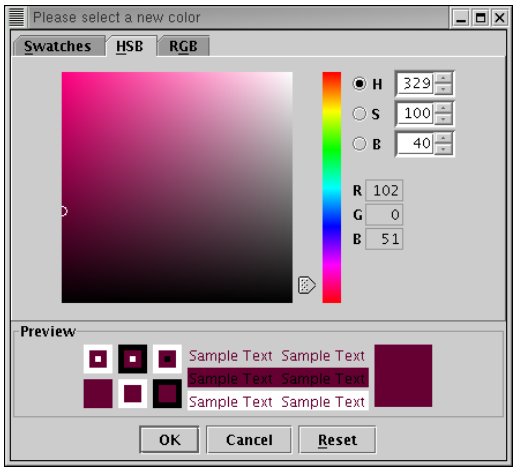


FIGURE 9. Setting function color in paraprof using HSB values

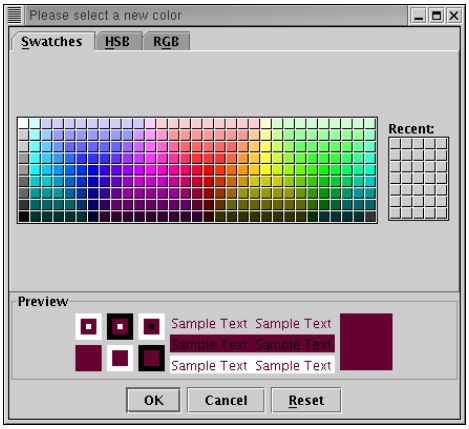


FIGURE 10. Color selection in paraprof

paraprof also provides selection of routines based on profile groups. By selecting **Windows** in the menu in any paraprof window and choosing **Show Group Ledger**, we see the groups as shown in the figure below.

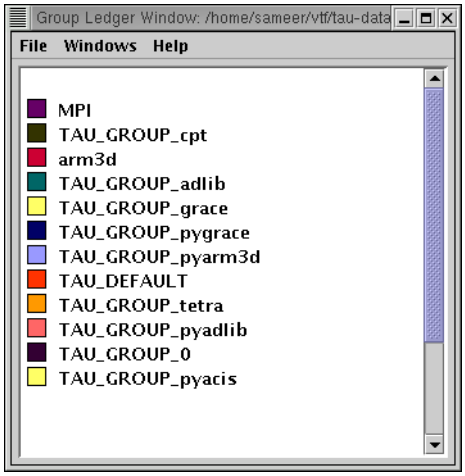


FIGURE 11. paraprof group ledger window

By clicking on a group name, all members of that group are highlighted. Clicking the second or the third mouse button on a group name allows the user to select from **Show This Group Only**, **Show All Groups Except This One**, and **Show All Groups**. The choice applies to all displays. By partitioning the performance data into meaningful groups at the instrumentation phases, we can perform analysis over groups and see the cumulative effect of a group of routines. In the node 32, context 0, thread 0, we can choose to see all elements of the MPI group as shown in the next figure. This is done by selecting the MPI group in the group ledger.

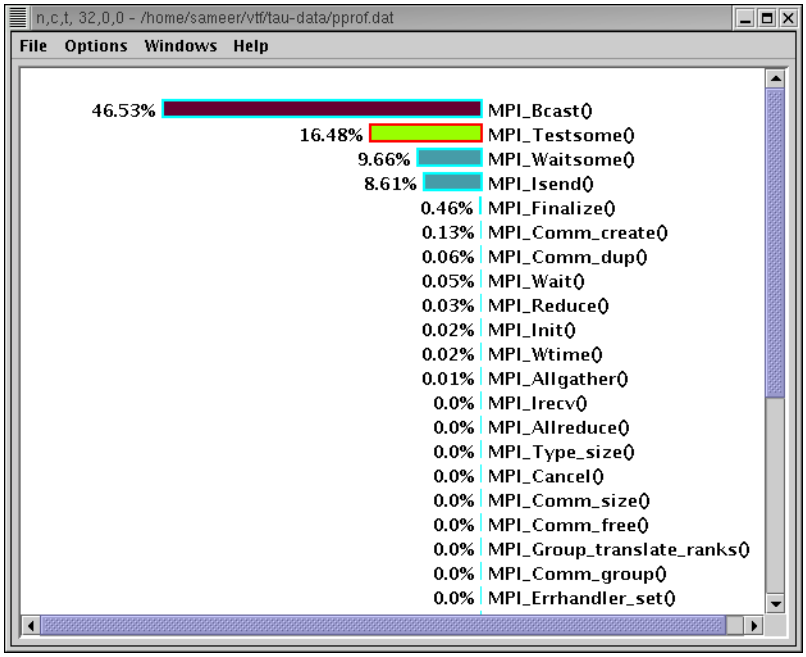


FIGURE 12. Exclusive time spent in all MPI routines

CHAPTER 5 *Tracing*

Typically, profiling shows the distribution of execution time across routines. It can show the code locations associated with specific bottlenecks, but it does not show the temporal aspect of performance variations. Tracing the execution of a parallel program shows when and where an event occurred, in terms of the process that executed it and the location in the source code. This chapter discusses how TAU can be used to generate event traces.

Generating Event Traces

TAU must be configured with the -TRACE option to generate event traces. This can be used in conjunction with -PROFILE to generate both profiles and traces. The traces are stored in a directory specified by the environment variable TRACEDIR, or the current directory, by default. Example:

```
% ./configure -SGITIMERS -arch=sgi64 -TRACE  -c++=KCC
% make clean; make install
% setenv TRACEDIR /users/sameer/tracedata/experiment56
% mpirun -np 4 matrix
```

This generates files named

`tautrace.<node>.<context>.<thread>.trc` and
`events.<node>.edf`

Using the utility **`tau_merge`**, these traces are then merged as shown below:

```
% tau_merge
usage: tau_merge [-a] [-r] inputtraces* (outputtrace|-)
Note: tau_merge assumes edf files are named
events.<nodeid>.edf, and generates a merged edf file
tau.edf
% tau_merge tautrace*.trc matrix.trc
```

This generates matrix.trc as the merged trace file and tau.edf as the merged event description file.

To convert merged or per-thread traces to another trace format, the utility **`tau_convert`** is used as shown below:

```
% tau_convert
usage: tau_convert [-alog | -SDDF | -dump | -paraver [-
t] | -pv | -vampir [-longsymbolbugfix] [-compact] [-
user|-class|-all] [-nocomm]] inputtrc edffile [output-
trc]
 Note: -vampir option assumes multiple threads/node
 Note: -t option used in conjunction with -paraver
option assumes multiple threads/node
```

To view the dump of the trace in text form, use

```
% tau_convert -dump matrix.trc tau.edf
```

tau_convert can also be used to convert traces to the Vampir trace format [VAMPIR-URL]. For single-threaded applications (such as the MPI application above), the `-pv` option is used to generate Vampir traces as follows:

```
% tau_convert -pv matrix.trc tau.edf matrix.pv
% vampir matrix.pv &
```

To convert TAU traces to SDDF or ALOG trace formats, `-SDDF` and `-alog` options may be used. When multiple threads are used on a node (as with `-jdk`, `-pthread` or `-tulipthread` options during `configure`), the `-vampir` option is used to convert the traces to the vampir trace format, as shown below:

```
% tau_convert -vampir smartsapp.trc tau.edf smartsapp.pv
% vampir smartsapp.pv &
```

To convert to the Paraver trace format, use the `-paraver` option for single threaded programs and `-paraver -t` option for multi-threaded programs.

NOTE: To ensure that inter-process communication events are recorded in the traces, in addition to the routine transitions, it is necessary to insert TAU_TRACE_SENDMSG and TAU_TRACE_RECVMSG macro calls in the source code during instrumentation. This is not needed when the TAU MPI Wrapper library is used.

Vampir: Visualizing TAU traces

Vampir is a robust parallel trace visualization tool sold by Pallas GmbH [PALLAS-URL]. It provides a convenient way to graphically analyze the performance characteristics of a parallel application. A variety of graphical displays present important aspects of the application runtime behavior:

- detailed timeline views of events and communication
- statistical analysis of program execution
- statistical analysis of communication operations
- system snapshot and animation

- dynamic calling tree

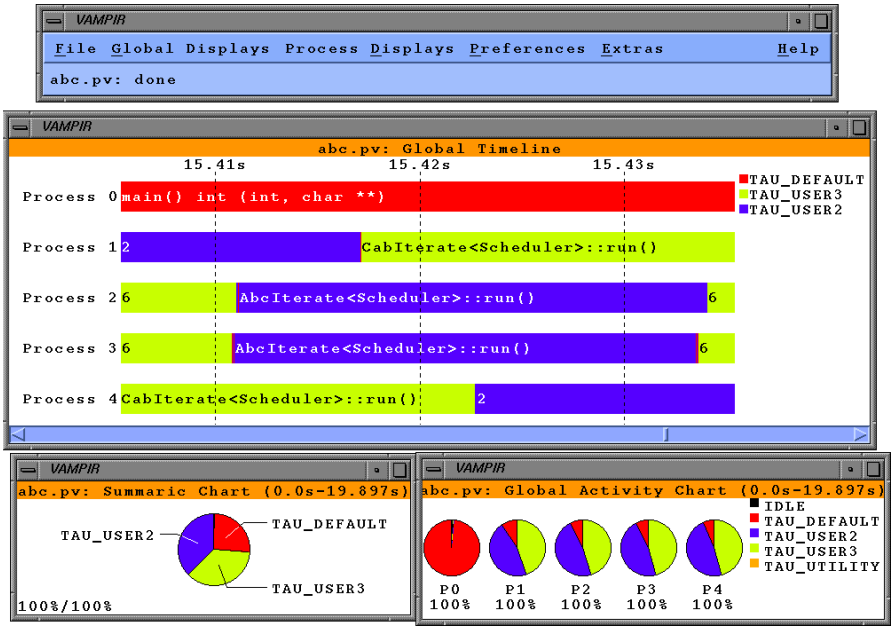


FIGURE 13. Vampir displays space-time diagrams and pie-charts

When interprocess communication is recorded, it shows up as directed line-segments connecting the sending and receiving processes. The details of a message can be obtained by clicking on it.

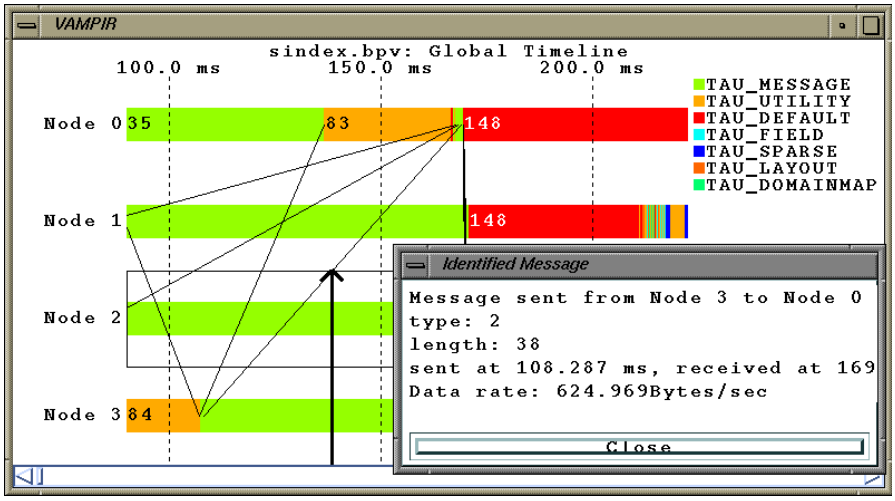


FIGURE 14. Vampir Space-time diagram shows inter-process communication

In Figure 15, “Scheduling work packets in SMARTS,” on page 105, we show how Vampir can be used to display scheduling of work packets or iterates in the Shared Memory Asynchronous Runtime System (SMARTS) [SMARTS-URL]

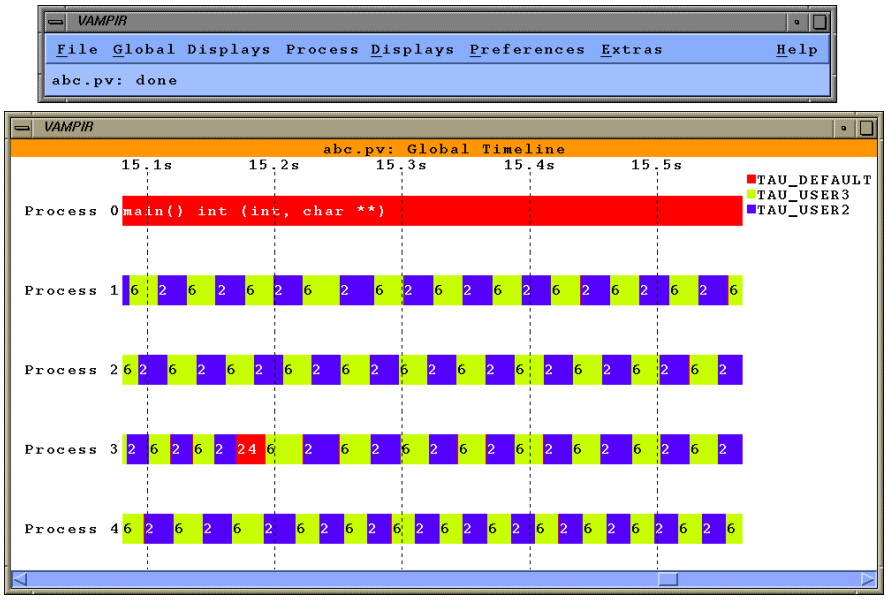


FIGURE 15. Scheduling work packets in SMARTS

In the next figure, we see the symbol legend and the dynamic call tree views provided by Vampir.

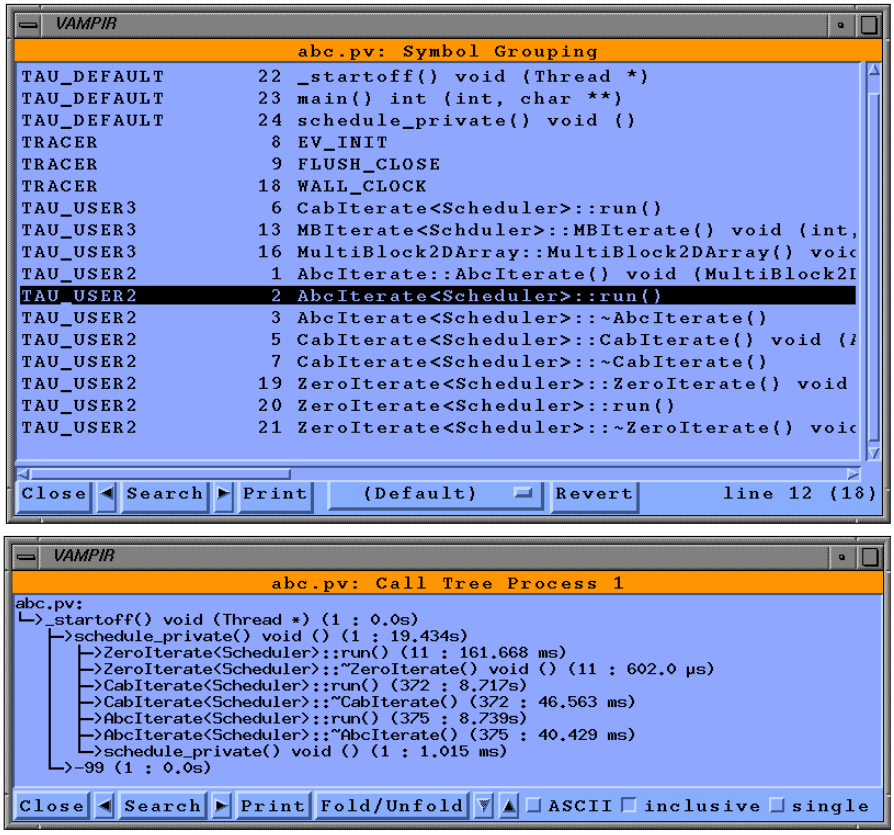


FIGURE 16. Vampir symbol legend and calltree display

Vampir has been used to compare the scheduling policies of the SMARTS package.

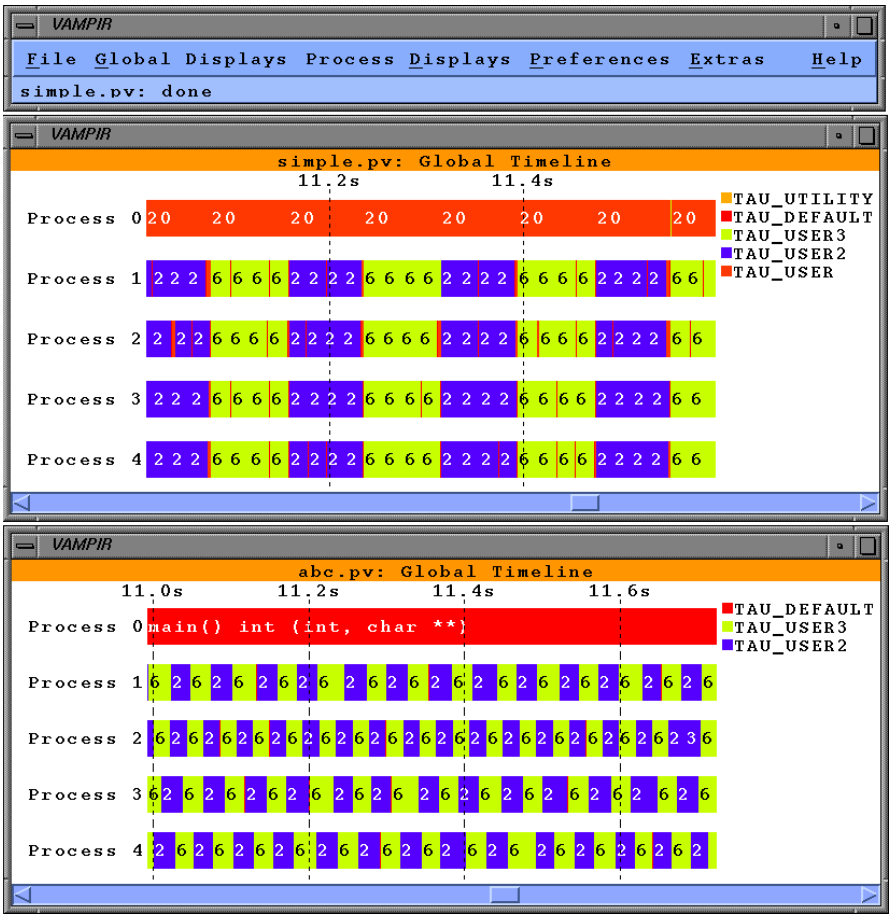


FIGURE 17. Comparing scheduling policies in SMARTS

The following figures illustrate the use of Vampir with Java applications. After converting the traces and invoking Vampir, choose appropriate colors for groups of methods using **Preferences->Colors->Activities** menu in Vampir.

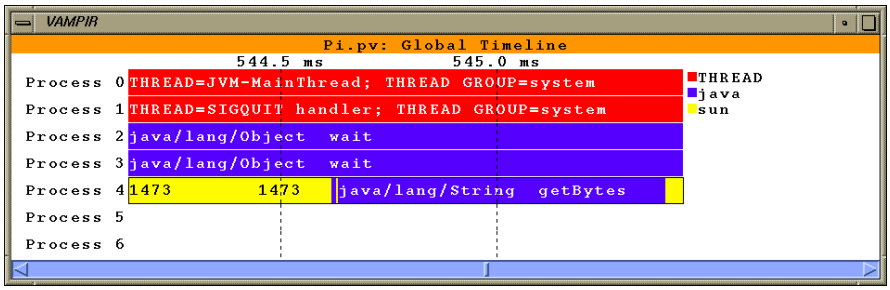


FIGURE 18. Timeline display in Vampir shows the activity (method) that each thread is in wrt time.

Clicking on a process(thread) selects it. Then the user can see the dynamic call tree of the process by choosing the **Process Displays->Call Tree** menu item as shown below.

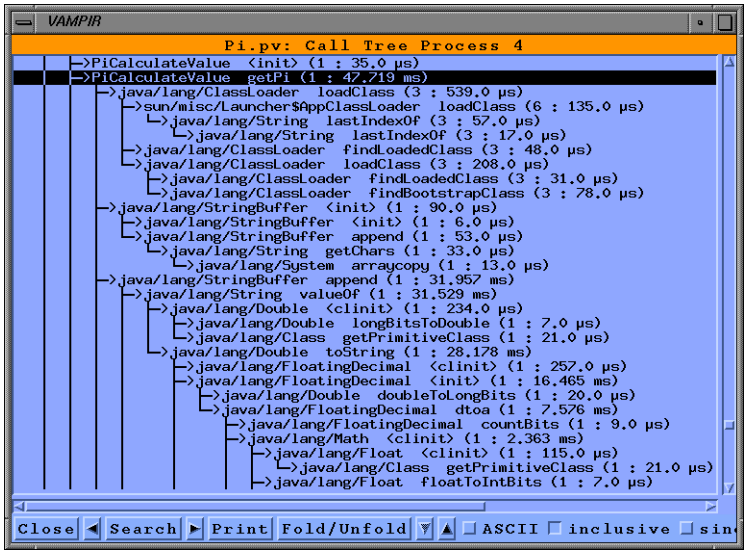


FIGURE 19. Call tree display of a thread shows the dynamic call tree annotated with performance metrics.

Vampir has a rich set of global displays. By choosing the **Global Displays ->Parallelism View** the user can see how many threads participate in an activity belonging to a group at any point in time. All timeline displays support a zoom option where the user can zoom into or out of a section of the trace.

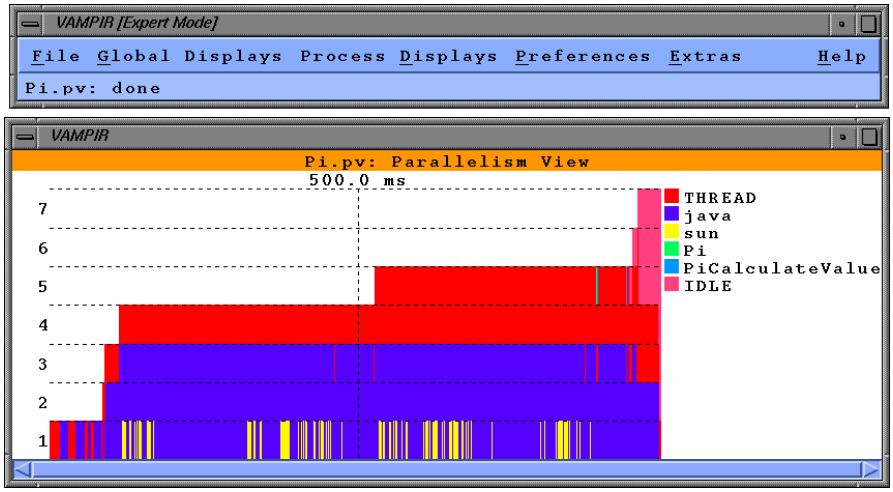


FIGURE 20. Parallelism view

By choosing other global displays such as **Summary chart** or **Activity chart**, the user can see a global summary of the time spent in different groups of methods as shown in the following figure.

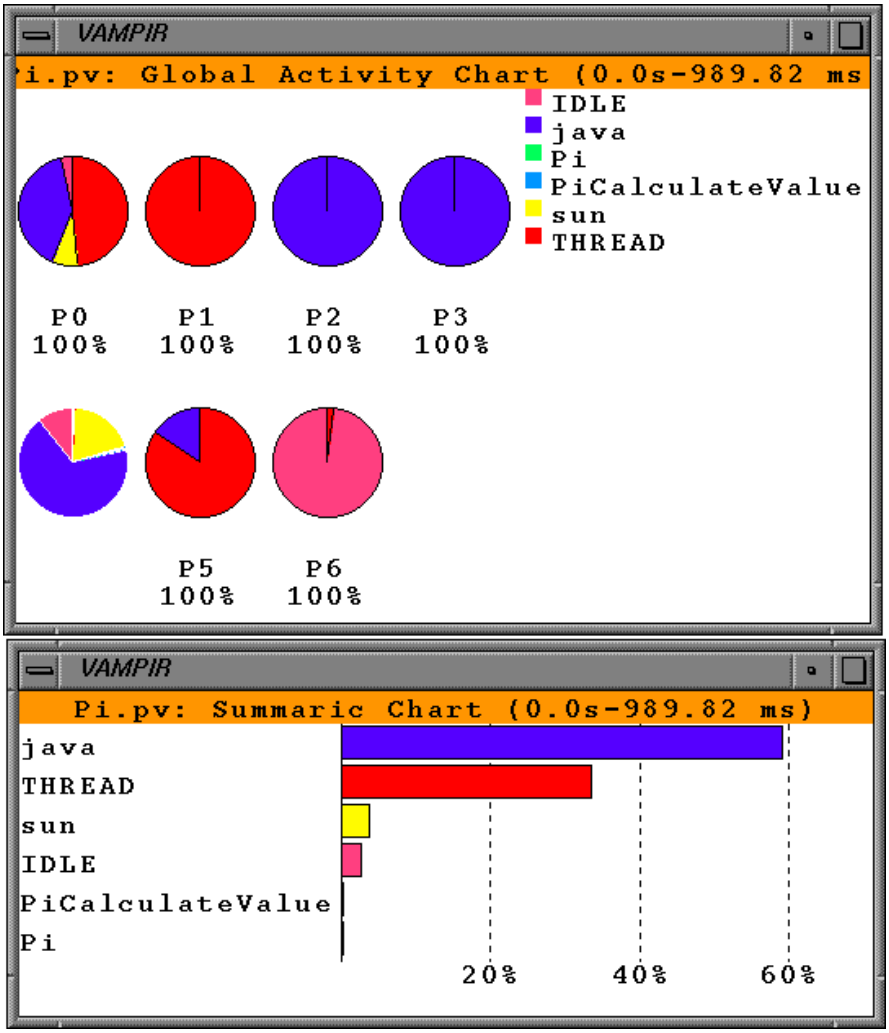


FIGURE 21. Summaric chart and activity chart global displays highlight the groups that take the most time using pie charts and histograms respectively.

Hybrid execution models can be traced in TAU by enabling support for the appropriate message passing model and thread package. One example of such a mixed

model program is shown in the following figure. It shows a trace of an OpenMP+MPI (OpenMPI) program that uses OpenMP threads for loop-level parallelism and MPI for inter-context message communication. The figure shows a timeline display.

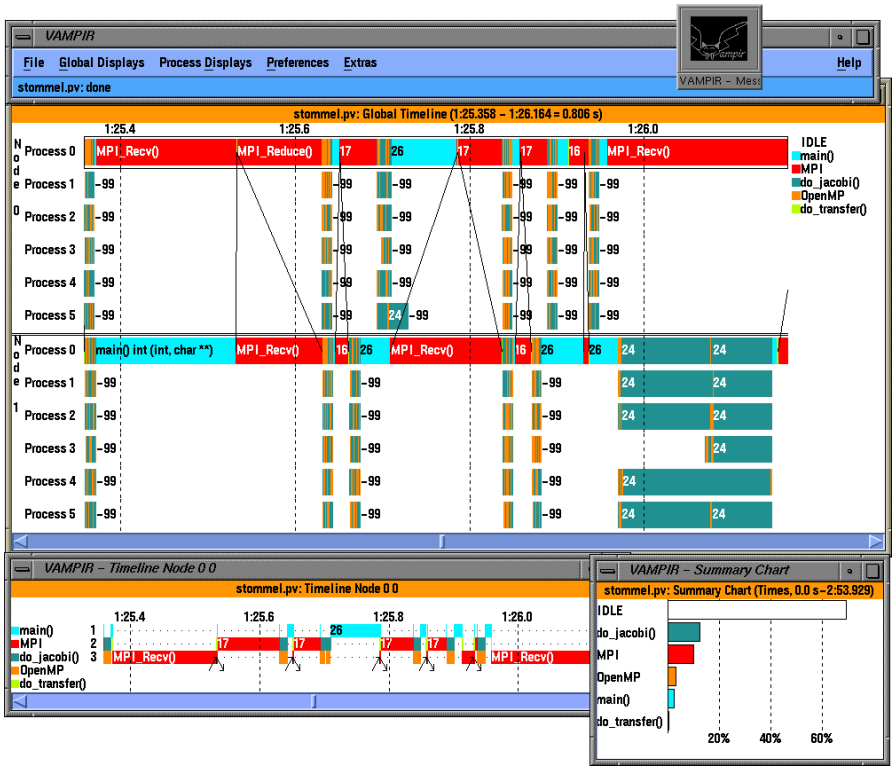


FIGURE 22. Tracing an OpenMPI application with TAU

Another example of mixed model programming is shown below. It shows an mpiJava [MPIJAVA-URL] program that uses the message passing interface (MPI) for inter-node communication and uses Java threads within each node for computation.

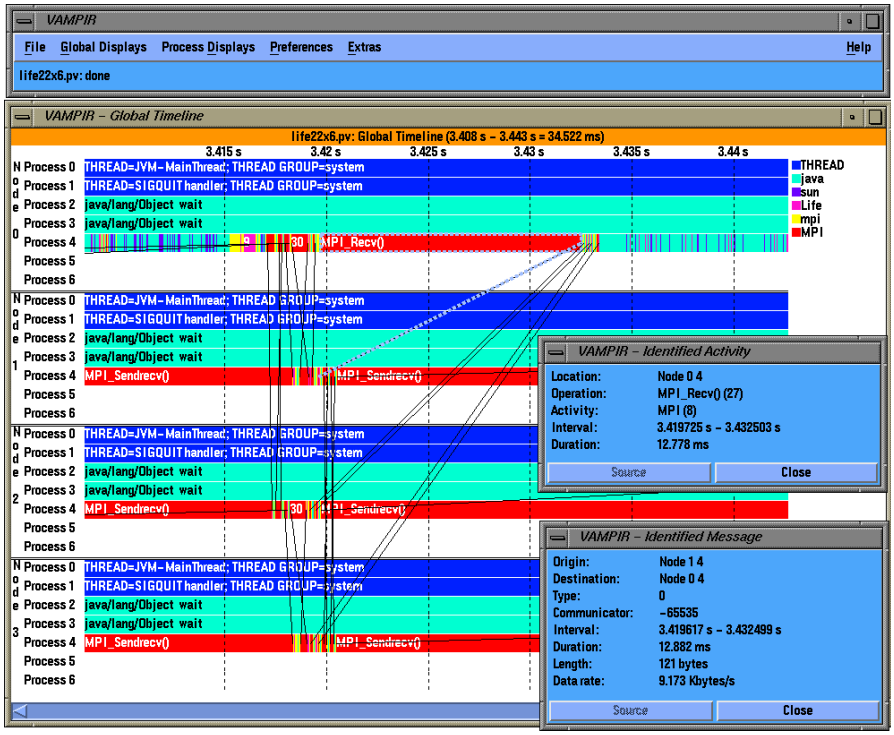


FIGURE 23. Tracing hybrid (mixed-model) execution models with MPI and Java.

CHAPTER 6

Performance Database

PerfDB is a performance database tool related to the TAU framework. The PerfDB database is designed to store and provide access to TAU profile data. A number of utility programs have been written in Java to load the data into PerfDB and to query the data. With PerfDB, users can perform performance analyses such as regression analysis, scalability analysis across multiple trials, and so on. An unlimited number of comparative analyses are available through the PerfDB toolkit. Work is being done to provide the user with standard analysis tools, and an API has been developed to access the data with standard Java classes.

Prerequisites

1. PostgreSQL 7.0 (or an equivalent DBMS)
   PerfDB requires a DataBase Management System (DBMS). It has been tested with both PostgreSQL and MySQL databases. The default database is PostgreSQL (http://www.postgresql.org).
2. Java 1.4
   The PerfDB utilities and API are written in Java.

Installation

PerfDB is installed as part of the standard TAU release. Shell scripts are installed in the TAU bin directory to run the configuration and loading utilities. It is assumed that the user has installed TAU and run TAU's configure and 'make install'.

Create a database

Once a DBMS is installed, a database needs to be created. This database can be called anything the user likes - perfdb is the default. If the DBMS is PostgreSQL, the command from the shell prompt is:

```
% createdb perfdb
```

If the user is in psql, the command is:

```
psql=# create database perfdb
```

Other DBMS are similar.

Build PerfDB

Change directory to the $TAUROOT/tools/src/perfdb directory, and issue the command:

```
% make
```

Configuration

PerfDB needs to be configured for the first time use. To configure PerfDB, run the command (assuming $TAUROOT/$arch/bin is in your path):

```
% perfdb_configure
```

The configure program will prompt the user for several values. The default values in all cases will work for 99.9% of users. Unless you need to specify something other than the default values (for example, if you are using MySQL instead of PostgreSQL, you need to specify a different JDBC .jar file, driver name, and database vendor). The only value for which there is no default is the database username. Enter the name of a database user which has administrative access. Because the utilities use the 'COPY' command with PostgreSQL, the user requires administrative access. Once the configuration program collects the information, it prompts for the user's database password, and connects to the database to test the configuration. If the configuration is valid, the database schema is loaded into the database.

Loading database schema

If the configuration ended successfully, then the database schema was loaded at the end of the configuration process. If problems occurred, then it may be necessary to load the database schema later. If that is the case, then the program to load the schema is:

```
%  perfdb_loadschema
```

Loading application data

To load application data, simply run the perfdb_loadapp program which takes a parameter <-x | --xmlfile> filename : the name of the application data file.

The xmlfile passed in is the application data file. A sample application data file is $TAUROOT/tools/src/perfdb/data/App_Info.xml. For e.g.,

```
% perfdb_loadapp –x App_Info.xml
```

The application loader will load the application, and return the ID of the application in the database.

Loading experiment data

To load experiment data, simply run the perfdb_loadexp program. It takes two parameters:

<-x | --xmlfile> filename: the name of the experiment data file.

<-a | --applicationid> value: the value of the application ID

The xmlfile passed in is the experiment data file. A sample experiment data file is $TAUROOT/tools/src/perfdb/data/Exp_Info.xml. For e.g.,

```
% perfdb_loadexp –x Exp_Info.xml – a 1
```

The experiment loader will load the experiment, and return the ID of the experiment in the database.

Translating TAU profiles

TAU data needs to be translated to XML in order to be loaded into the database. This is a simple operation, performed with the perfdb_translate program. There are several parameters for perfdb_translate:

<-s | --sourcefile> filename: the name of the TAU pprof dump format data file (created by pprof -d)

<-d | --destinationfile> filename: the name of the XML output file

<-a | --applicationid> value: the value of the application ID

<-e | --experimentid> value: the value of the experiment ID

For e.g.,

```
% pprof –d > pprof.dat
% perfdb_translate –s pprof.dat –d pprof.xml –a 1 –e 1
```

Loading translated trial data

Once the data has been translated, it can be loaded into the database. The data is loaded by running the perfdb_loadtrial command. It takes the following parameters:

<-x | --xmlfile> filename: the name of the translated trial data file

< -t | --trialid> value: the value of an existing trial ID

<-p | --problemfile> filename: the name of a problem definition file

perfdb_loadtrial can be run two ways. When creating a new trial, the user calls perfdb_loadtrial with an optional problem definition file. The problem definition file is a user-defined XML file that describes the trial data. An example problem definition file is in the data directory. For e.g.,

```
% perfdb_loadtrial –x pprof.xml –p sample_problem.xml
```

If the user is adding trial data to an existing trial (due to multiple metrics recorded during the run with TAU), then the problem file is omitted, and the trial ID is passed in:

```
% perfdb_loadtrial –t 1
```

Once the data has been loaded into the database, analysis can be performed. See the ${TAUROOT}/tools/src/dms/README file for more details.

CHAPTER 7 *Summary*

The TAU performance framework and toolkit is an ongoing research and development project. The TAU Portable Profiling and Tracing Toolkit described in this document represents functionality present in the current software release. All available software should be considered research software available to the community under the BSD style license.

Software Availability

TAU Portable Profiling and Tracing Toolkit may be downloaded as freeware from the following website [TAU-URL]:

`http://www.cs.uoregon.edu/research/paracomp/tau`

For more information, please refer to the documentation section at the above URL. Bug reports and comments may be sent to:

`tau-bugs@cs.uoregon.edu`

Technical papers about TAU can be downloaded from the TAU Publications home-page at [TAU-PUBS-URL]

Acknowledgments

The TAU development team wishes to thank the U.S. Government, Department of Energy for their support of the TAU project under the DOE-2000, DOE MICS office contract, University of Utah ASCI subcontract, and ASCI Level 3 grants.

CHAPTER 8

Appendix: Configuration Issues

Instructions for Installing TAU under Windows

Supported Systems: Windows9x/NT.

Compiler: Microsoft Visual C++ Version 5.0 - Service Pack 3, or above.

NOTE: Service Pack 3 MUST be installed... it contains required bug fixes.

Section1.

The following steps detail how to build TAU libraries on Windows9x/NT.

For illustrative purposes, we assume that the TAU root directory is: “C:\TAU-SOURCE-DIR”.

1. Download TAU. TAU is distributed as source and prebuilt libraries forWindows. If you wish to use the prebuilt libraries, skip to steps 25 and 26.
2. Open Microsoft Visual C++ ... henceforth referred to as VC++.
3. i) If you wish to create a dynamic library proceed to step 4.
   ii) If you wish to create a static library proceed to step 12.
4. Creating a dynamic library allows you to profile Java code using Sun’s JDK1.2+.
5. From the “File” menu in VC++, select “New”.
6. Click on the “Projects” tab.
7. Select “Win32 Dynamic-Linked Library”.
8. Type in a name for your new library.
9. Make sure that the radio button on the right of the new project window is set to “Create a new workspace”.
10. Click “OK”
11. Please skip to step 18 below.
12. From the “File” menu in VC++, select “New”.
13. Click on the “Projects” tab.
14. Select “Win32 Static Library”.
15. Type in a name for your new library.
16. Make sure that the radio button on the right of the new project window is set to “Create a new workspace”.
17. Click “OK”

18. Open Windows Explorer, and, from the TAU source you downloaded, copy the C:\TAU-SOURCE-DIR\include\Profile and C:\TAU-SOURCE-DIR\src\Profile directories to your new project directory. For example, if you new project was located in "C:\Program Files\DevStudio\MyProjects\NewTauLib", you would now have two new subdirectories of "C:\Program Files\DevStudio\MyProject\NewTauLib" named, "include\Profile" and "src\Profile".
19. Now, back in VC++, from the "Project" menu, select "Add To Project" and click on "Files". Move to your new "src\Profile" directory and select the following list of files: (holding down the control key whilst clicking so that you can select more than one file)
    FunctionInfo.cpp
    Profiler.cpp
    RtsLayer.cpp
    RtsThread.cpp
    TauJava.cpp
    TauMapping.cpp
    UserEvent.cpp
    WindowsThreadLayer.cpp

    Now click OK.
20. From the "Project" again, select "Settings" and then click on the "C/C++" tab.
21. Make sure that the Category in "General" and in the "Preprocessor definitions:" box, add the following defines: (separated by commas)
    TAU_WINDOWS TAU_DOT_H_LESS_HEADERS PROFILING_ON

    If you want to profile a Java application, also add:
    JAVA

    Click "OK"
22. From the "Tools" menu, select "Options". Click on the "Directories" tab. Make sure that the "Show directories for:" field has "Include files" selected. Now add a new include directory named "C:\YOUR_PROJECT_DIRECTORY\include". Thus, our above example would be: "C:\Program Files\DevStudio\MyProjects \NewTauLib\include". Also add the include directories for jvmpi.h and jni_md.h. These are typically in "C:\JAVA_ROOT_DIR\include" and "C:\JAVA_ROOT_DIR\include\win32". Thus, when done, you should have three new include directories listed. Now click "OK".
23. Now, from the "Build" menu, select "Build PROJECT_NAME.dll (or .lib)"

24. Ignoring warnings, you should now have a library file in your project debug directory.
25. If you created a dll for use with Java, you only need to make sure that the dll is in a location that can be found by Java when it is running. The command to profile your Java application is: java -XrunTAU “Java Application Name” “Application parameters”. The default TAU.dll for use with a Java app. is provided in: “C:\TAU-SOURCE-DIR\windows\lib”. If, when building your dll from the source, you named it something other than TAU.dll, you can either rename it, or replace “TAU” in “java -XrunTAU” with your dll name.
26. If you created a static library, you will need to include a reference to it in when you build your application. You can do this by adding the library file to you list of libraries in “Project -> Settings -> Link” inside VC++. You must then make sure that the library is in a location known to VC++. You can do this in your “Tools ->Options->Directories->Library files” section of VC++

Section 2.

The Windows port ships with a prebuilt version of pprof which can be used to view your profiling data (See the TAU documentation for more details). Make sure that pprof.exe is in your current path. It can be found in C:\TAU-SOURCE-DIR\windows\bin. Currently, there is no version of Racy for Windows, however, we are rewriting Racy in Java and will soon have it running on the Windows platform.

For information on how to profile your C/C++ and Java code, please see the TAU documentation.

For more information on the Windows port of TAU please send mail to `tau-bugs@cs.uoregon.edu`.

CHAPTER 9 *References*

URLs

[TAU-URL]	**http://www.cs.uoregon.edu/research/paracomp/tau**
[TAU-PUBS-URL]	**http://www.cs.uoregon.edu/research/paracomp/tau/papers.html**
[TAU-PGROUPS-URL]	**http://www.acl.lanl.gov/tau/docs/selective.html**
[KAI-URL]	**http://www.kai.com**
[GNU-URL]	**http://www.gnu.org**
[PGI-URL]	**http://www.pgroup.com**
[FUJITSU-URL]	**http://www.tools.fujitsu.com/linux/index.shtml**
[TCLTK-URL]	**http://www.scriptics.com**
[NPB-URL]	**http://www.nas.nasa.gov/Software/NPB/**

[DYNINST-URL]	**http://www.cs.umd.edu/projects/dyninstAPI/**
[PARADYN-URL]	**http://www.cs.wisc.edu/~paradyn/**
[PAPI-URL]	**http://icl.cs.utk.edu/projects/papi/**
[PCL-URL]	**http://www.fz-juelich.de/zam/PCL/**
[PARP-URL]	**http://www.csi.uoregon.edu/projects/parp/**
[VAMPIR-URL]	**http://www.pallas.de/pages/vampir.htm**
[PALLAS-URL]	**http://www.pallas.de**
[POOMA-URL]	**http://www.acl.lanl.gov/pooma**
[SMARTS-URL]	**http://www.acl.lanl.gov/smarts**
[TULIP-URL]	**http://www.acl.lanl.gov/tulip**
[ACL-SW-URL]	**http://www.acl.lanl.gov/software/**
[OPENMP-URL]	**http://www.openmp.org**
[MUSE-URL]	**http://public.lanl.gov/radiant/**
[OPARI-URL]	**http://www.fz-juelich.de/zam/kojak/opari**
[EPILOG-URL]	**http://www.fz-juelich.de/zam/kojak**
[PDT-URL]	**http://www.acl.lanl.gov/pdtoolkit**
[MPI-URL]	**http://www-unix.mcs.anl.gov/mpi/**
[MPIJAVA-URL]	**http://www.npac.syr.edu/projects/pcrc/HPJava/mpiJava.html**