TAU User's Guide

Tuning and Analysis Utilities

TAU User's Guide

version 2.9

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

http://www.acl.lanl.gov/tau

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, Racy, 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 trace visualization tool.

This chapter discusses installation of the TAU portable profiling package.

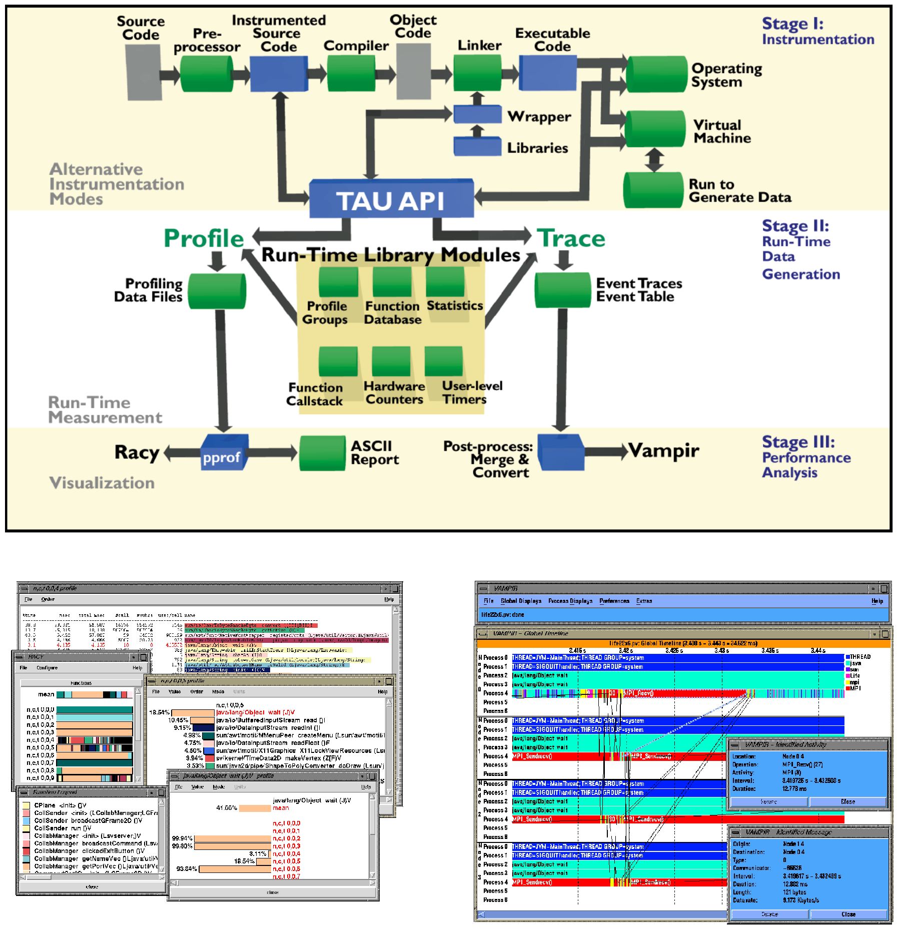


FIGURE 1. **Architecture of TAU**

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 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), CC (SGI, SUN, Cray), g++ (from GNU), FCC (from Fujitsu), 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) and KCC (from KAI).

-pthread

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

-tulipthread=<directory>

Specifies Tulip threads (HPC++) as the threads package to be used as well as the location of the root directory where the package is installed. [TULIP-URL]

-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]

-pdt=<directory>

Specifies the location of the installed PDT (Program Database Toolkit) root directory. PDT is used to build **tau_instrumentor**, a C++ instrumentation program that automatically inserts TAU annotations in the source code. [PDT-URL]

-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). 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. 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 conjunctionwith 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 conjunctionwith 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 wall-clock time. (See NOTE below.)

-CPUTIME

Specifies the use of user+ system time (collectively CPU time) for profile measurements, instead of the default wall-clock 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.

NOTE: The default measurement option in TAU is to use the wallclock time, which is the total time a program takes to execute, inclding 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.

-dyninst=<dir>

Specifies the directory where DynInst dynamic instrumentation package is installed. Using DynInst, a user can invoke **tau_run** to instrument an executable program at runtime. This represents work in progress [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.

-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 analyse the performance. Wallclock time is used for profiling program entities.

-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 nano-second 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).

-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]

-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.

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/jdk1.2
% 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 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
```

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

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

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 and 2000 Series, CC 7.2+, KAI [KAI-URL] KCC and g++/egcs [GNU-URL] compilers are supported.

2. LINUX PCs

On Linux based Intel x86 PC clusters, KAI KCC, g++, egcs (GNU), pgCC (PGI) [PGI-URL], FCC (Fujitsu) [FUJITSU-URL] compilers have been tested. Versions of g++ prior to 2.8.1 need an additional -useropt=-fguiding-decls to be added to the list of configure options. Among the GNU versions, we recommend using gcc-2.95+ g++.

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, g++ can be used.

6. Compaq (DEC Alpha workstations)

On Compaq (DEC) Alpha workstations running Digital Unix, g++ may be used with TAU.

7. LINUX Compaq (DEC Alpha 21164) clusters

On Linux based Alpha workstation clusters, g++ may be used with TAU.

8. Cray T3E

On Cray, KAI KCC and Cray CC compilers have been tested with TAU.

9. Microsoft Windows

On Windows, Microsoft Visual C++ 5.1 and JDK 1.2+ compilers have been tested with TAU.

10. IA-64 Linux.

On IA-64 Linux platform, g++ compiler has been tested with TAU.

TAU has been tested with JDK 1.2 on Solaris, Windows and Linux. On Solaris, we needed to disable support for the JIT compiler by specifying -Djava.compiler=   on the java commandline along with -XrunTAU. On Linux, it worked with or without the JIT compiler.

TAU may work with minor modifications on other platforms.

Software Requirements

1. Tcl/Tk

TAU's GUI **racy** requires Tcl/Tk 7.4/4.0 or better (8.x is recommended). Tcl/Tk is available from Scriptics [TCLTK-URL] as freeware.

2. Xauth

The display must be secure to run racy, the profile browser. Xauthority -- not `xhost+` should be used for secure (authentication based) interaction between the X client and the servers. Refer to the TAU FAQ [TAU-SECURITY-URL] for instructions. Contact your system administrator if your X-server is not configured for generating Xauth cookies.

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_INCLUDE` for the include directories
- `TAU_DEFS` for the defines on the command-line
- `TAU_LIBS` for the TAU 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_DISABLE` for the default TAU stub library for Fortran, and
- `USER_OPT` for any user defined options specified during configuration

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) -lmpi
LDFLAGS         = $(USER_OPT)
MAKEFILE        = Makefile
PRINT           = pr
RM              = /bin/rm -f
TARGET          = matrix
EXTRAOBJS       =
##############################################
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. For 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. For 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`** 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).

For 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 Harware 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. It highlights instrumentation for templates and user defined events.

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.

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.
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.
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).
fortran & f90	- Show how to instrument a simple Fortran 90 (F90) program. A C++ linker needs to be used when linking the fortran application.
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 - 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.

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.

openmpi - Illustrates TAU's support for hybrid exection 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, that requires an extension of the class definition with a TAU pointer and a second scheme of external hash-table lookup that relies on looking at the object address at each method invocation. Both these examples illustrate the use of the TAU Mapping API.

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.

Automatic Instrumentation of C++ sources

tau_instrumentor inserts TAU instrumention macros 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 that uses the .pdb file and the C++ source code to generate an instrumented version of the source code. 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 “*************************************”
```

4. 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.

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_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_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 (say) 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_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);
}
```

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` and `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.

For 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. For 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 on the link command line, the TAU stub makefile variable TAU_DISABLE. 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.

For 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. For 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` subroutines.

For 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 (say) 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 of an instrumented application and 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. For 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.

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

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 provides 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 1. **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

TABLE 1. **Events measured by setting the environment variable `PCL_EVENT` in TAU**

PCL_EVENT	**Event Measured**
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
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

TABLE 1. Events measured by setting the environment variable `PCL_EVENT` in TAU

PCL_EVENT	**Event Measured**
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
PCL_STALL_JUMP	Jump stalls
PCL_STALL_LOAD	Load stalls
PCL_STALL_STORE	Store Stalls
PCL_STALL	Stalls
PCL_MFLOPS	Milions 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-1.1
% make clean install
% cd examples/papi
% setenv PAPI_EVENT PAPI_FP_INS
% a.out
```

TABLE 2. 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

TABLE 2. **Events measured by setting the environment variable PAPI_EVENT in TAU**

PAPI_EVENT	**Event Measured**
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
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)

TABLE 2. **Events measured by setting the environment variable PAPI_EVENT in TAU**

PAPI_EVENT	**Event Measured**
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
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

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

PAPI_EVENT	Event Measured
PAPI_TOT_CYC	Total cycles
PAPI_IPS	Instructions executed per second
PAPI_LST_INS	Total load/store inst. executed
PAPI_SYC_INS	Sync. inst. executed
PAPI_L1_DCH	L1 D Cache Hit
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

TABLE 2. **Events measured by setting the environment variable PAPI_EVENT in TAU**

PAPI_EVENT	**Event Measured**
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
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

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
```

On Solaris, you will need to disable the JIT compiler while using TAU. This is done using the `-Djava.compiler=` option

```
% java -XrunTAU -Djava.compiler=  Pi
```

To disable the JIT compiler. If you're using JAVA HotSpot JVM, you may need the `-classic` flag too.

```
% java -classic -XrunTAU -Djava.compiler=  Pi
```

(`java -version` shows the version of JVM. TAU has been tested with JDK 1.2.2 and 1.3 but should work with any release of JDK after 1.2).

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

pprof

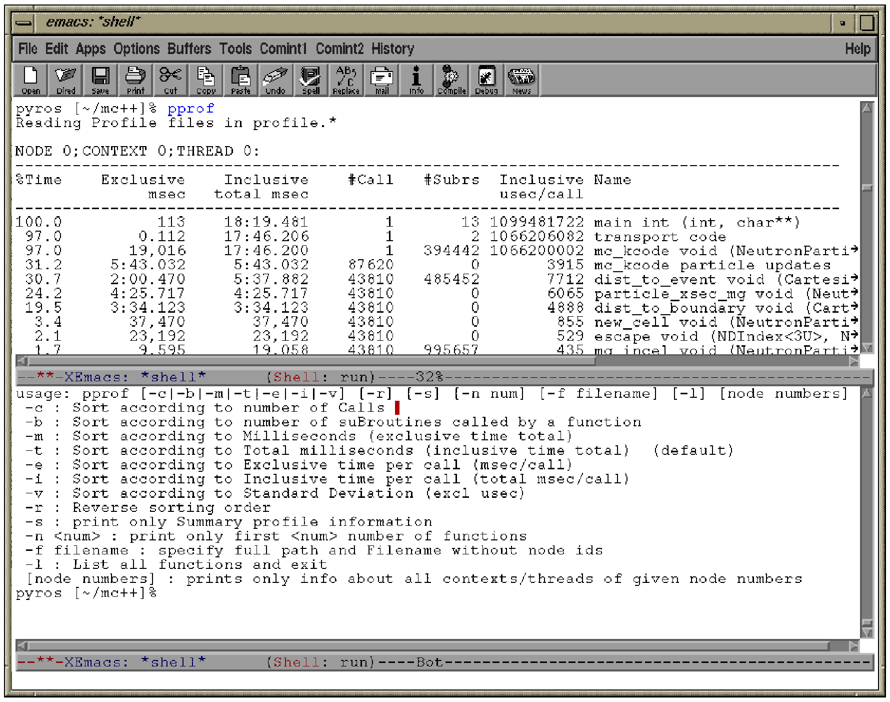
pprof sorts and displays profile data generated by TAU. To view the profile, merely execute pprof in the directory where profile files are located (or set the `PROFILEDIR` environment variable).

```
% 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
```


emacs: *shell*
File Edit Apps Options Buffers Tools Comint1 Comint2 History Help
Open Dired Save Print Cut Copy Paste Undo Spell Replace Mail Info Compile Debug News

```
pyros [~/mc++]% pprof
Reading Profile files in profile.*

NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive     #Call     #Subrs  Inclusive Name
              msec   total msec                        usec/call
---------------------------------------------------------------------------------------
100.0          113    18:19.481         1         13 1099481722 main int (int, char**)
 97.0        0.112    17:46.206         1          2 1066206082 transport code
 97.0       19,016    17:46.200         1     394442 1066200002 mc_kcode void (NeutronParti→
 31.2     5:43.032     5:43.032     87620          0       3915 mc_kcode particle updates
 30.7     2:00.470     5:37.882     43810     485452       7712 dist_to_event void (Cartesi→
 24.2     4:25.717     4:25.717     43810          0       6065 particle_xsec_mg void (Neut→
 19.5     3:34.123     3:34.123     43810          0       4888 dist_to_boundary void (Cart→
  3.4       37,470       37,470     43810          0        855 new_cell void (NeutronParti→
  2.1       23,192       23,192     43810          0        529 escape void (NDIndex<3U>, N→
  1.7        9.595       19.058     43810     995657        435 mg_incel void (NeutronParti→
--**-XEmacs: *shell*        (Shell: run)----32%------------------------------------------
usage: pprof [-c|-b|-m|-t|-e|-i|-v] [-r] [-s] [-n num] [-f filename] [-l] [node numbers]
 -c : Sort according to number of Calls
 -b : Sort according to number of suBroutines called by a function
 -m : Sort according to Milliseconds (exclusive time total)
 -t : Sort according to Total milliseconds (inclusive time total)  (default)
 -e : Sort according to Exclusive time per call (msec/call)
 -i : Sort according to Inclusive time per call (total msec/call)
 -v : Sort according to Standard Deviation (excl usec)
 -r : Reverse sorting order
 -s : print only Summary profile information
 -n <num> : print only first <num> number of functions
 -f filename : specify full path and Filename without node ids
 -l : List all functions and exit
 [node numbers] : prints only info about all contexts/threads of given node numbers
pyros [~/mc++]%
--**-XEmacs: *shell*        (Shell: run)----Bot------------------------------------------
```

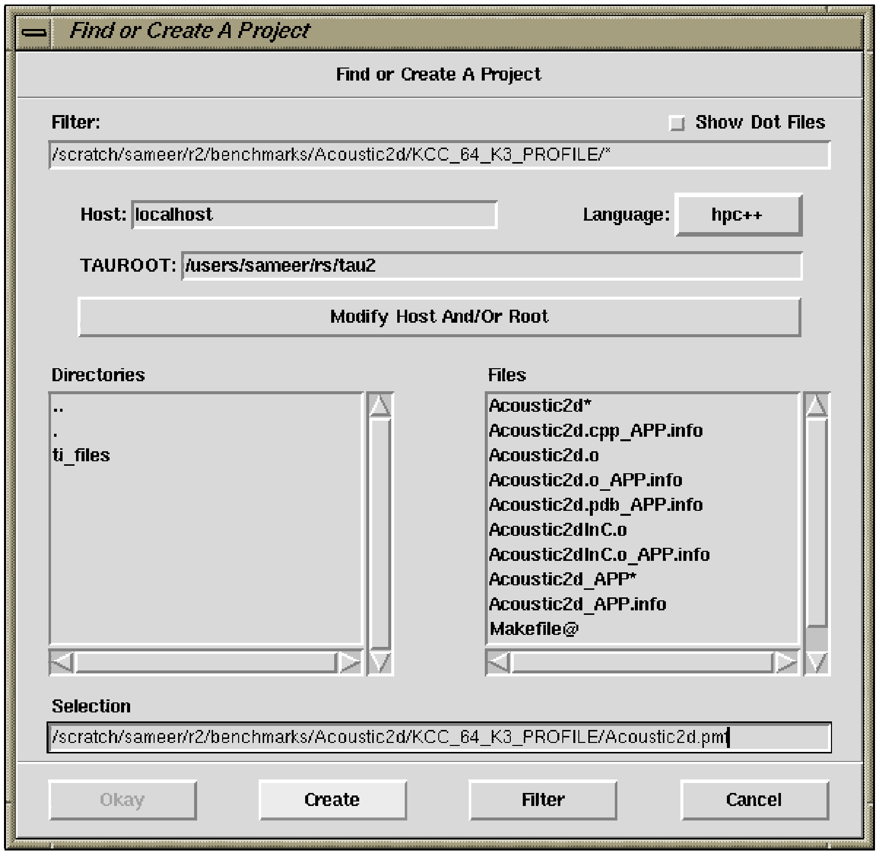
FIGURE 2. pprof in an xemacs window

racy

Racy is the graphical interface to pprof. It shows the profile data in terms of histograms and text displays. Invoke racy in the directory that contains the profile files.

```
% racy
```

In the project management window, select a project by typing in a project file name with a .pmf extension (e.g., `matrix.pmf`), and clicking (the first mouse button) on the “`Create`” button.



Filter: /scratch/sameer/r2/benchmarks/Acoustic2d/KCC_64_K3_PROFILE/*

Host: localhost

TAUROOT: /users/sameer/rs/tau2

Directories:

```
..
.
ti_files
```

Files:

```
Acoustic2d*
Acoustic2d.cpp_APP.info
Acoustic2d.o
Acoustic2d.o_APP.info
Acoustic2d.pdb_APP.info
Acoustic2dInC.o
Acoustic2dInC.o_APP.info
Acoustic2d_APP*
Acoustic2d_APP.info
Makefile@
```

Selection: /scratch/sameer/r2/benchmarks/Acoustic2d/KCC_64_K3_PROFILE/Acoustic2d.pmf

FIGURE 3. **Create a project by creating or choosing an existing project**

NOTE: If the project window does not appear, it is probably due to problems with the security of the X-display. You may not use `xhost +` while using racy. You may need to explicitly turn off xhost by invoking `xhost -` and using Xauthentication. The cookies are generated and stored in the `~/.Xauthority` file. If your server is not configured to generate these cookies, please contact your system administrator for configuring the X display to make it secure. Users may also want to use **ssh**, rather than telnet/rlogin to login to a remote node where racy is invoked. If this is done, the cookies need not be copied to the remote node explicitly by the user. For more information please refer to [TAU-SECURITY-URL].

After the project is created or selected, the main racy window appears.

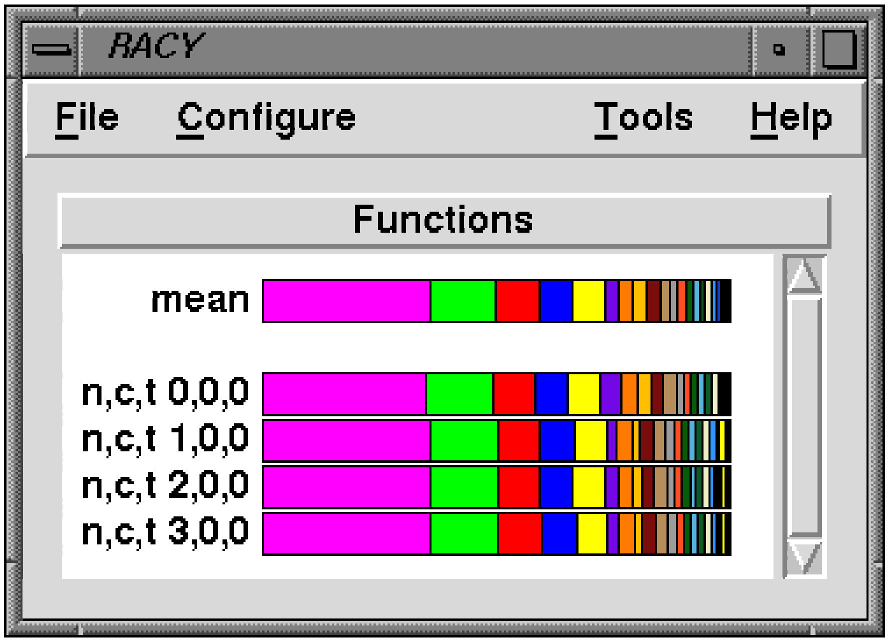
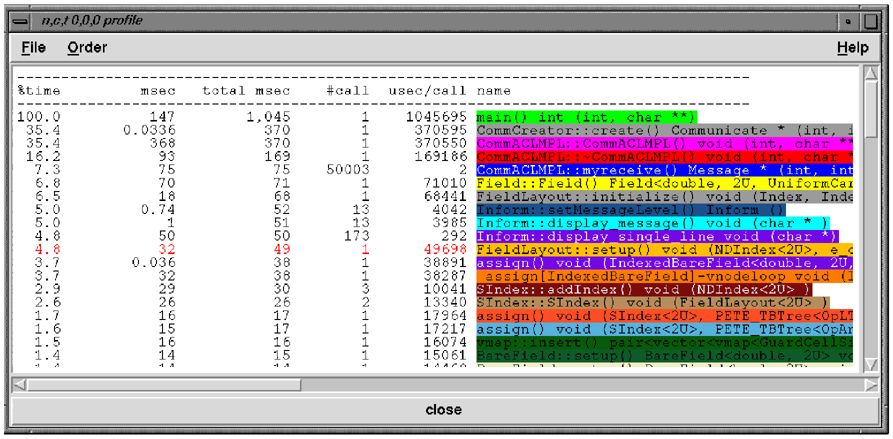


FIGURE 4. **Main racy window showing the profile of functions on different <node>,<context>,<thread>s and the mean profile.**

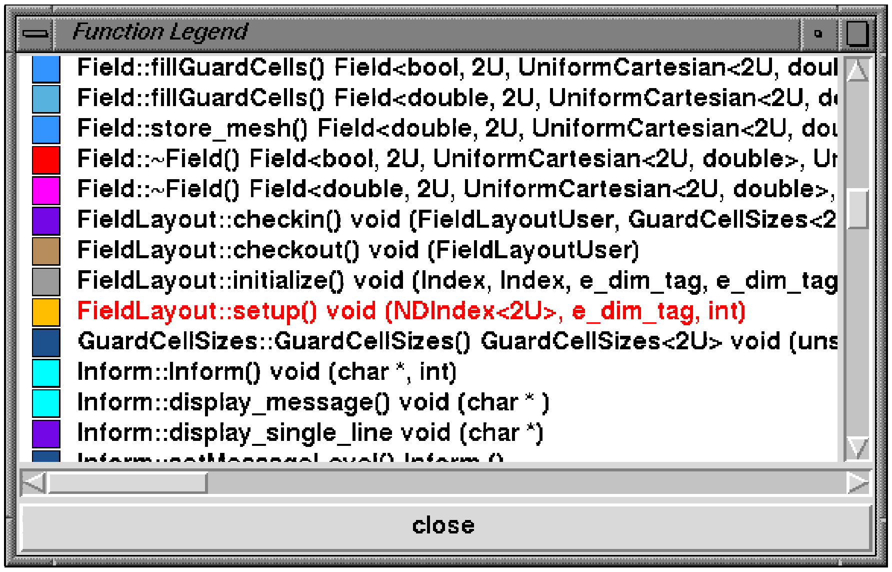
This shows the relative time spent in each function as a horizontal bargraph. Each node, context, thread is represented as a horizontal bar with each function assigned a color. In this main racy window, click middle mouse button over **n,c,t 0,0,0** to see the textual profile of node 0, context 0, thread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



```
---------------------------------------------------------------------------------------
%time         msec   total msec     #call   usec/call name
---------------------------------------------------------------------------------------
100.0          147        1,045         1     1045695 main() int (int, char **)
 35.4       0.0336          370         1      370595 CommCreator::create() Communicate * (int, i
 35.4          368          370         1      370550 CommACLMPL::CommACLMPL() void (int, char **
 16.2           93          169         1      169186 CommACLMPL::~CommACLMPL() void (int, char *
  7.3           75           75     50003           2 CommACLMPL::myreceive() Message * (int, int
  6.8           70           71         1       71010 Field::Field() Field<double, 2U, UniformCar
  6.5           18           68         1       68441 FieldLayout::initialize() void (Index, Inde
  5.0         0.74           52        13        4042 Inform::setMessageLevel() Inform ()
  5.0            1           51        13        3985 Inform::display_message() void (char * )
  4.8           50           50       173         292 Inform::display_single_line void (char *)
  4.8           32           49         1       49698 FieldLayout::setup() void (NDIndex<2U>, e_c
  3.7        0.036           38         1       38891 assign() void (IndexedBareField<double, 2U,
  3.7           32           38         1       38287  assign[IndexedBareField]-vnodeloop void (I
  2.9           29           30         3       10041 SIndex::addIndex() void (NDIndex<2U> )
  2.6           26           26         2       13340 SIndex::SIndex() void (FieldLayout<2U> )
  1.7           16           17         1       17964 assign() void (SIndex<2U>, PETE_TBTree<OpLT
  1.6           15           17         1       17217 assign() void (SIndex<2U>, PETE_TBTree<OpAr
  1.5           16           16         1       16074 vmap::insert() pair<vector<vmap<GuardCellSi
  1.4           14           15         1       15061 BareField::setup() BareField<double, 2U> vo
```

FIGURE 5. **Text shows the detailed profile on n,c,t 0,0,0.**

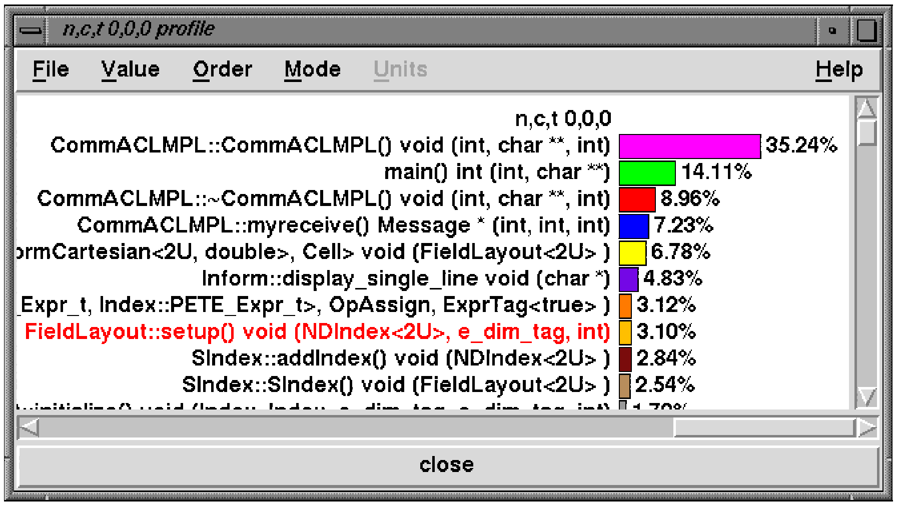
Next, select **Show Function Legend** on the main racy window **File** menu to see the list of functions.



🟦 Field::fillGuardCells() Field<bool, 2U, UniformCartesian<2U, doul
🟦 Field::fillGuardCells() Field<double, 2U, UniformCartesian<2U, d
🟦 Field::store_mesh() Field<double, 2U, UniformCartesian<2U, dou
🟥 Field::~Field() Field<bool, 2U, UniformCartesian<2U, double>, Ur
🟪 Field::~Field() Field<double, 2U, UniformCartesian<2U, double>,
🟪 FieldLayout::checkin() void (FieldLayoutUser, GuardCellSizes<2
🟫 FieldLayout::checkout() void (FieldLayoutUser)
⬜ FieldLayout::initialize() void (Index, Index, e_dim_tag, e_dim_tag
🟧 FieldLayout::setup() void (NDIndex<2U>, e_dim_tag, int)
🟦 GuardCellSizes::GuardCellSizes() GuardCellSizes<2U> void (uns
🟦 Inform::Inform() void (char *, int)
🟦 Inform::display_message() void (char *)
🟪 Inform::display_single_line void (char *)
🟦 Inform::setMessageLevel() Inform ()

FIGURE 6. **The Function Legend window shows the color assigned to each function.**

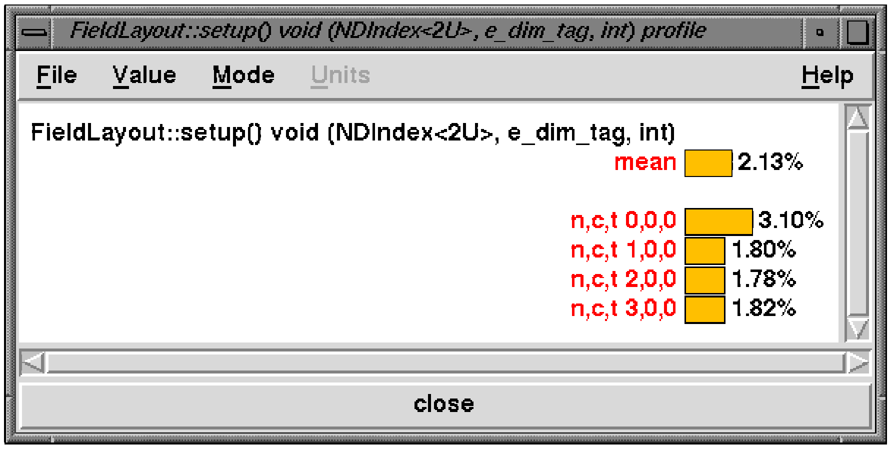
Click the third mouse button over a function to highlight it in the currently open windows. To see the relative function profile on one node, click first mouse button on a node in the racy main window.



```
                                                  n,c,t 0,0,0
       CommACLMPL::CommACLMPL() void (int, char **, int)  35.24%
                              main() int (int, char **)  14.11%
      CommACLMPL::~CommACLMPL() void (int, char **, int)  8.96%
          CommACLMPL::myreceive() Message * (int, int, int)  7.23%
ormCartesian<2U, double>, Cell> void (FieldLayout<2U> )  6.78%
               Inform::display_single_line void (char *)  4.83%
_Expr_t, Index::PETE_Expr_t>, OpAssign, ExprTag<true> )  3.12%
 FieldLayout::setup() void (NDIndex<2U>, e_dim_tag, int)  3.10%
                 SIndex::addIndex() void (NDIndex<2U> )  2.84%
               SIndex::SIndex() void (FieldLayout<2U> )  2.54%
```

FIGURE 7. **The above node profile shows the relative contribution of the functions on the node 0, context 0, thread 0 .**

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

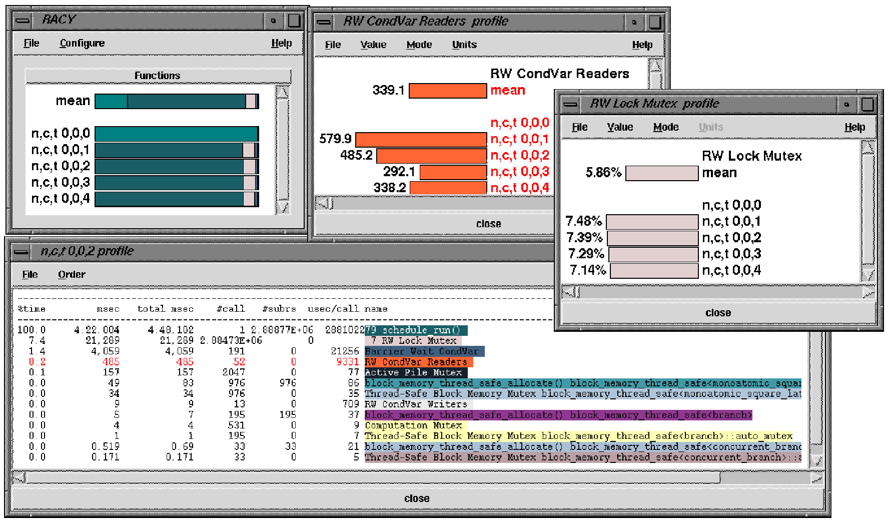

FieldLayout::setup() void (NDIndex<2U>, e_dim_tag, int) profile
File
Value
Mode
Units
Help
close

```
FieldLayout::setup() void (NDIndex<2U>, e_dim_tag, int)
                                   mean      2.13%

                              n,c,t 0,0,0    3.10%
                              n,c,t 1,0,0    1.80%
                              n,c,t 2,0,0    1.78%
                              n,c,t 3,0,0    1.82%
```

FIGURE 8. **The function window shows the profile of the function over all nodes, contexts and threads.**

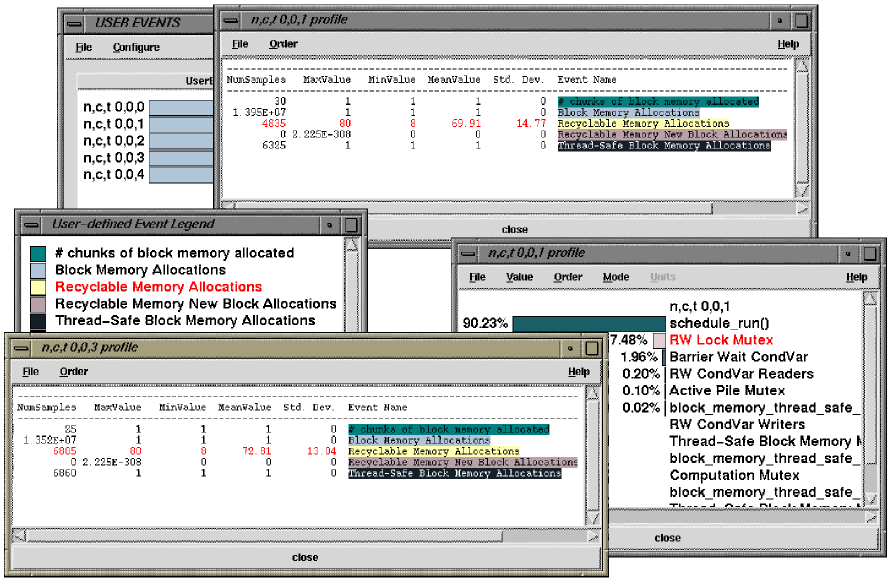
If user-defined events are profiled in addition to routines, then an event window appears with the function profile window. Navigation of user event profiles is identical to the navigation of function profiles as described above. The following figure demonstrates the use of user-defined event profiles from the PaRP project [PARP-URL]



```
---------------------------------------------------------------------------------------------
%time        msec   total msec     #call    #subrs  usec/call name
---------------------------------------------------------------------------------------------
100.0   4:22.004     4:48.102          1 2.88877E+06  288102279 schedule_run()
  7.4     21,289       21,289 2.88473E+06          0         7 RW Lock Mutex
  1.4      4,059        4,059        191         0     21256 Barrier Wait CondVar
  0.2        485          485         52         0      9331 RW CondVar Readers
  0.1        157          157       2047         0        77 Active Pile Mutex
  0.0         49           83        976       976        86 block_memory_thread_safe_allocate() block_memory_thread_safe<monoatomic_squar
  0.0         34           34        976         0        35 Thread-Safe Block Memory Mutex block_memory_thread_safe<monoatomic_square_lat
  0.0          9            9         13         0       709 RW CondVar Writers
  0.0          5            7        195       195        37 block_memory_thread_safe_allocate() block_memory_thread_safe<branch>
  0.0          4            4        531         0         9 Computation Mutex
  0.0          1            1        195         0         7 Thread-Safe Block Memory Mutex block_memory_thread_safe<branch>::auto_mutex
  0.0      0.519         0.69         33        33        21 block_memory_thread_safe_allocate() block_memory_thread_safe<concurrent_branc
  0.0      0.171        0.171         33         0         5 Thread-Safe Block Memory Mutex block_memory_thread_safe<concurrent_branch>::a
```

FIGURE 9. **Racy displays user defined event profiles as well.**

User defined events are also measured and displayed on each node, context and thread



```
----------------------------------------------------------------------------------------
NumSamples   MaxValue   MinValue  MeanValue  Std. Dev.  Event Name
----------------------------------------------------------------------------------------
        30          1          1          1          0  # chunks of block memory allocated
 1.395E+07          1          1          1          0  Block Memory Allocations
      4835         80          8      69.91      14.77  Recyclable Memory Allocations
         0 2.225E-308          0          0          0  Recyclable Memory New Block Allocations
      6325          1          1          1          0  Thread-Safe Block Memory Allocations
```

- 🟩 # chunks of block memory allocated
- 🟦 Block Memory Allocations
- 🟨 Recyclable Memory Allocations
- 🟪 Recyclable Memory New Block Allocations
- ⬛ Thread-Safe Block Memory Allocations

```
n,c,t 0,0,1
90.23%  schedule_run()
 7.48%  RW Lock Mutex
 1.96%  Barrier Wait CondVar
 0.20%  RW CondVar Readers
 0.10%  Active Pile Mutex
 0.02%  block_memory_thread_safe_
        RW CondVar Writers
        Thread-Safe Block Memory M
        block_memory_thread_safe_
        Computation Mutex
        block_memory_thread_safe_
```

```
----------------------------------------------------------------------------------------
NumSamples   MaxValue   MinValue  MeanValue  Std. Dev.  Event Name
----------------------------------------------------------------------------------------
        25          1          1          1          0  # chunks of block memory allocated
 1.352E+07          1          1          1          0  Block Memory Allocations
      6805         80          8      72.81      13.04  Recyclable Memory Allocations
         0 2.225E-308          0          0          0  Recyclable Memory New Block Allocations
      6860          1          1          1          0  Thread-Safe Block Memory Allocations
```

FIGURE 10. **User defined events may be used to track memory bugs.**

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 occured, 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. For 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 | -pv |
         -vampir [-compact] [-user|-class|-all]
         [-nocomm]] inputtrc edffile [outputtrc]

  Note: -vampir 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 &
```

`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

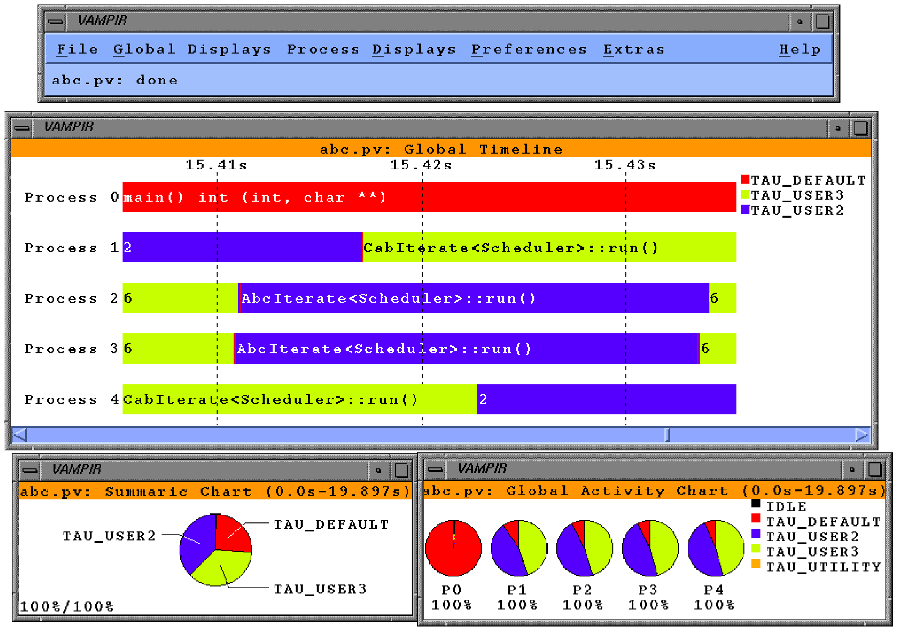
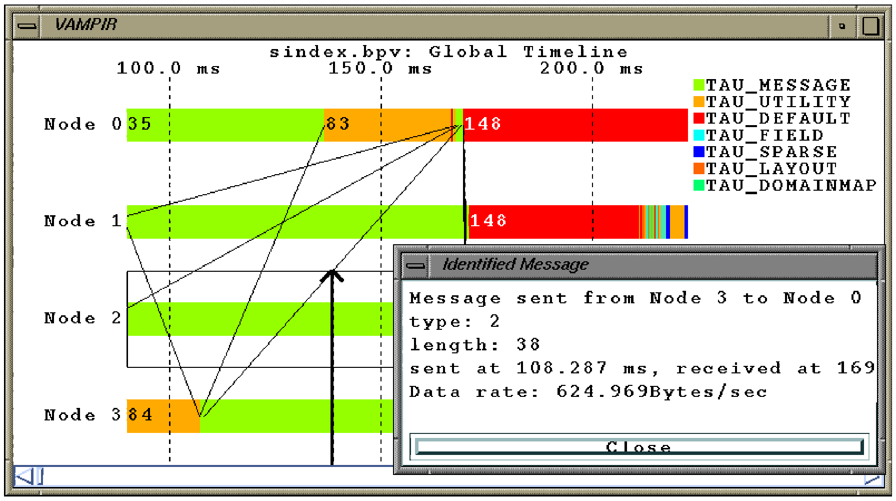

VAMPIR
File Global Displays Process Displays Preferences Extras Help
abc.pv: done
abc.pv: Global Timeline
15.41s
15.42s
15.43s
TAU_DEFAULT
TAU_USER3
TAU_USER2
Process 0 main() int (int, char **)
Process 1 2 CabIterate<Scheduler>::run()
Process 2 6 AbcIterate<Scheduler>::run() 6
Process 3 6 AbcIterate<Scheduler>::run() 6
Process 4 CabIterate<Scheduler>::run() 2
abc.pv: Summaric Chart (0.0s-19.897s)
TAU_USER2
TAU_DEFAULT
TAU_USER3
100%/100%
abc.pv: Global Activity Chart (0.0s-19.897s)
IDLE
TAU_DEFAULT
TAU_USER2
TAU_USER3
TAU_UTILITY
P0 100%
P1 100%
P2 100%
P3 100%
P4 100%

FIGURE 11. **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.



```
Message sent from Node 3 to Node 0
type: 2
length: 38
sent at 108.287 ms, received at 169
Data rate: 624.969Bytes/sec
```

FIGURE 12. **Vampir Space-time diagram shows inter-process communication**

In Figure 13, “Scheduling work packets in SMARTS,” on page 71, 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]

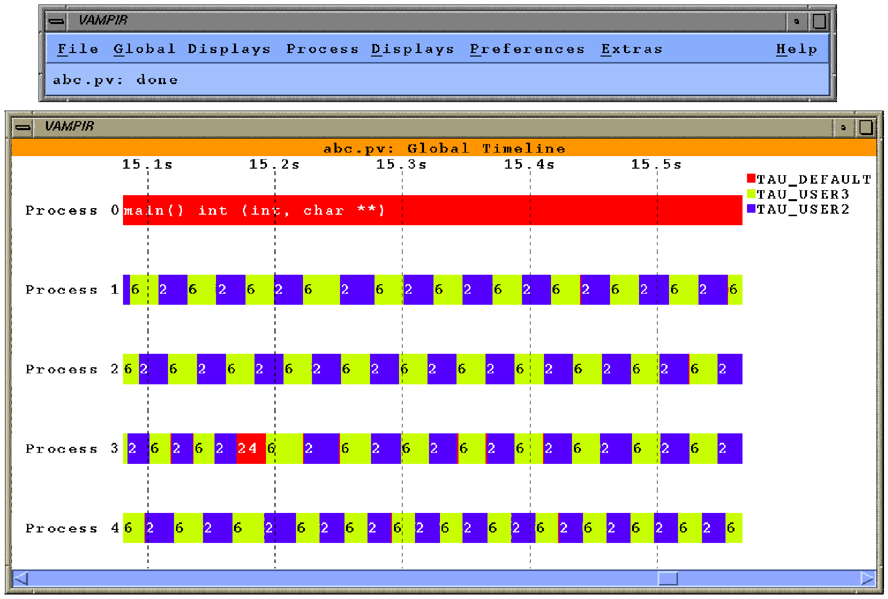
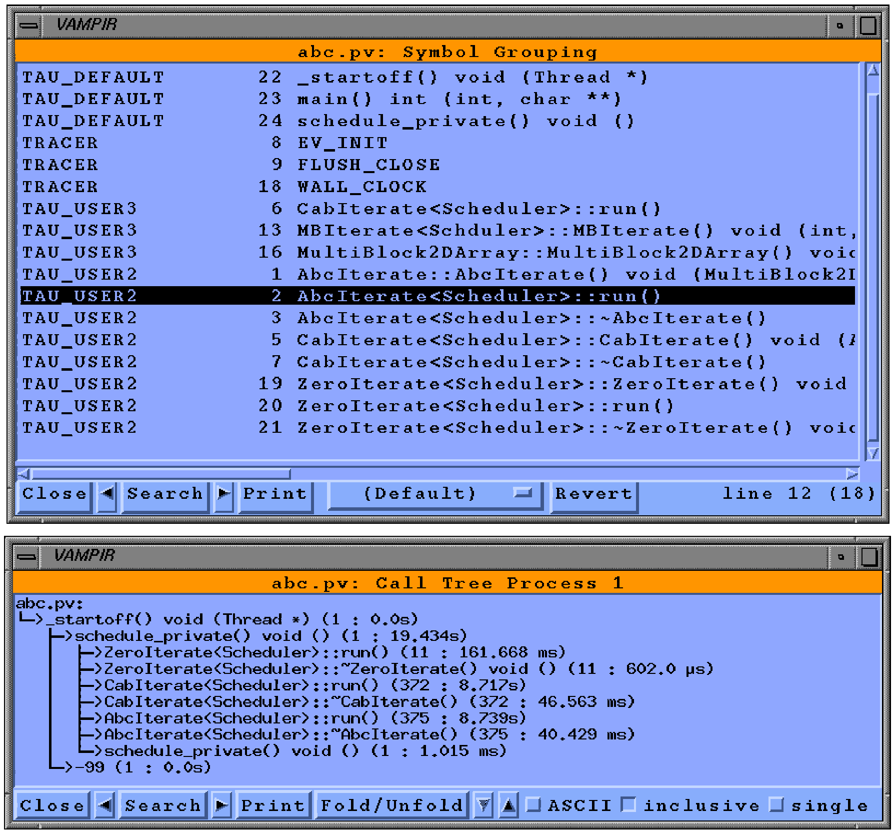


FIGURE 13. Scheduling work packets in SMARTS

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

.



```
VAMPIR
abc.pv: Symbol Grouping
TAU_DEFAULT        22 _startoff() void (Thread *)
TAU_DEFAULT        23 main() int (int, char **)
TAU_DEFAULT        24 schedule_private() void ()
TRACER              8 EV_INIT
TRACER              9 FLUSH_CLOSE
TRACER             18 WALL_CLOCK
TAU_USER3           6 CabIterate<Scheduler>::run()
TAU_USER3          13 MBIterate<Schduler>::MBIterate() void (int,
TAU_USER3          16 MultiBlock2DArray::MultiBlock2DArray() voic
TAU_USER2           1 AbcIterate::AbcIterate() void (MultiBlock2I
TAU_USER2           2 AbcIterate<Scheduler>::run()
TAU_USER2           3 AbcIterate<Scheduler>::~AbcIterate()
TAU_USER2           5 CabIterate<Scheduler>::CabIterate() void (I
TAU_USER2           7 CabIterate<Scheduler>::~CabIterate()
TAU_USER2          19 ZeroIterate<Scheduler>::ZeroIterate() void
TAU_USER2          20 ZeroIterate<Scheduler>::run()
TAU_USER2          21 ZeroIterate<Scheduler>::~ZeroIterate() voic
Close ◄ Search ► Print  (Default)  Revert        line 12 (18)
```

```
VAMPIR
abc.pv: Call Tree Process 1
abc.pv:
└─>_startoff() void (Thread *) (1 : 0.0s)
    ├─>schedule_private() void () (1 : 19.434s)
    │   ├─>ZeroIterate<Scheduler>::run() (11 : 161.668 ms)
    │   ├─>ZeroIterate<Scheduler>::~ZeroIterate() void () (11 : 602.0 µs)
    │   ├─>CabIterate<Scheduler>::run() (372 : 8.717s)
    │   ├─>CabIterate<Scheduler>::~CabIterate() (372 : 46.563 ms)
    │   ├─>AbcIterate<Scheduler>::run() (375 : 8.739s)
    │   ├─>AbcIterate<Scheduler>::~AbcIterate() (375 : 40.429 ms)
    │   └─>schedule_private() void () (1 : 1.015 ms)
    └─>-99 (1 : 0.0s)
Close ◄ Search ► Print Fold/Unfold ▼ ▲ ☐ASCII ☐inclusive ☐single
```

FIGURE 14. **Vampir symbol legend and calltree display**

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

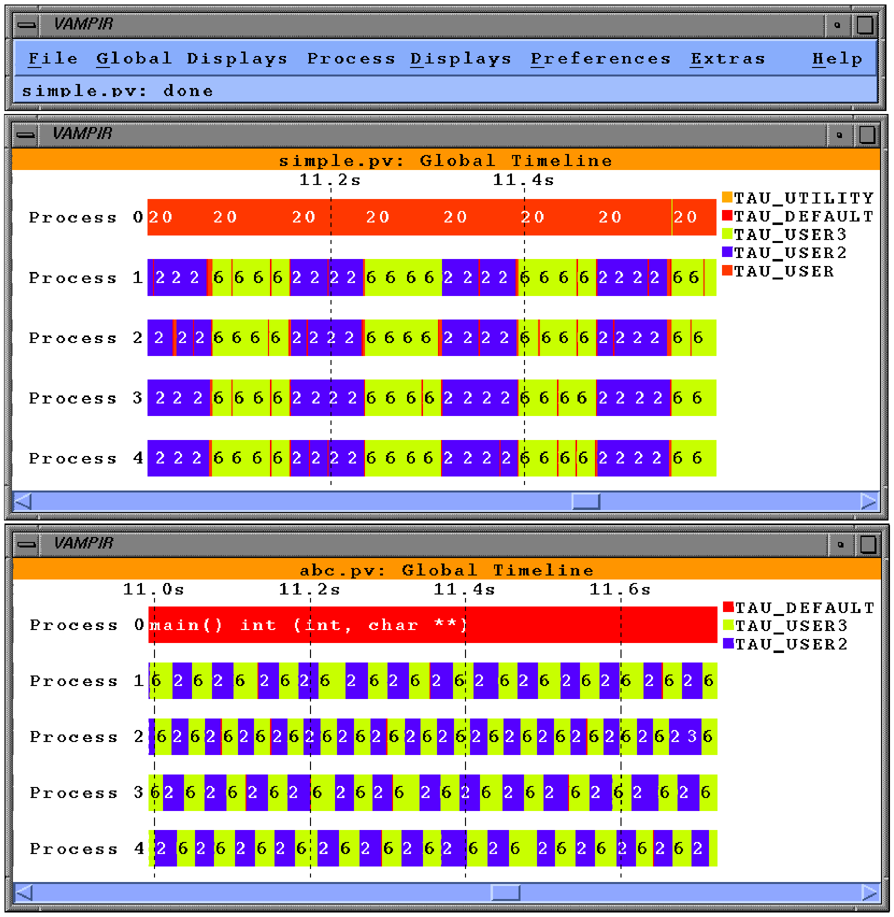


FIGURE 15. Comparing scheduling policies in SMARTS

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.

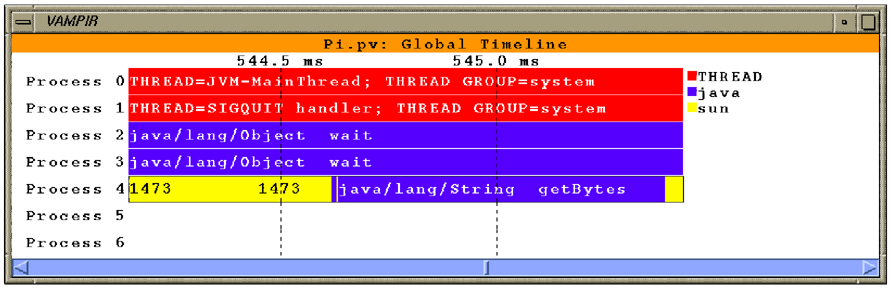
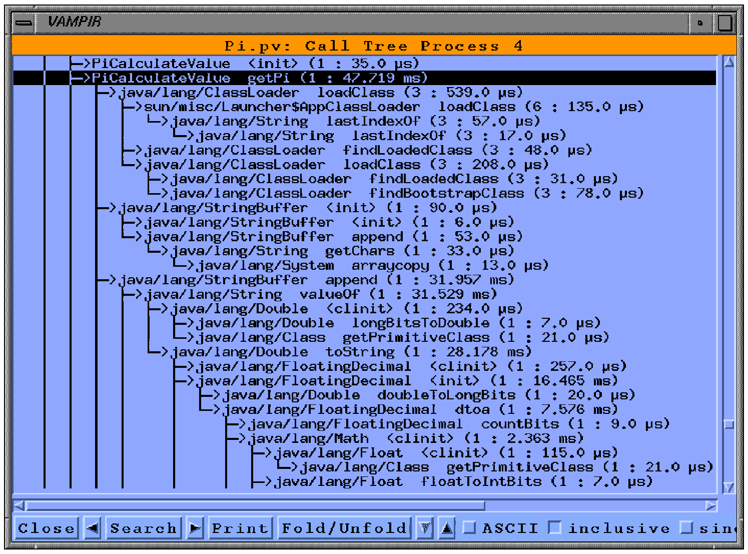


FIGURE 16. **Timeline display in Vampir shows the activity (method) that each thread is in wrt time.**

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.



```
|   |--->PiCalculateValue  <init> (1 : 35.0 µs)
|   |--->PiCalculateValue  getPi (1 : 47.719 ms)
|   |   |-->java/lang/ClassLoader  loadClass (3 : 539.0 µs)
|   |   |   |-->sun/misc/Launcher$AppClassLoader  loadClass (6 : 135.0 µs)
|   |   |   |   └->java/lang/String  lastIndexOf (3 : 57.0 µs)
|   |   |   |      └->java/lang/String  lastIndexOf (3 : 17.0 µs)
|   |   |   |-->java/lang/ClassLoader  findLoadedClass (3 : 48.0 µs)
|   |   |   └-->java/lang/ClassLoader  loadClass (3 : 208.0 µs)
|   |   |       |-->java/lang/ClassLoader  findLoadedClass (3 : 31.0 µs)
|   |   |       └-->java/lang/ClassLoader  findBootstrapClass (3 : 78.0 µs)
|   |   |-->java/lang/StringBuffer  <init> (1 : 90.0 µs)
|   |   |   |-->java/lang/StringBuffer  <init> (1 : 6.0 µs)
|   |   |   └-->java/lang/StringBuffer  append (1 : 53.0 µs)
|   |   |       └->java/lang/String  getChars (1 : 33.0 µs)
|   |   |          └->java/lang/System  arraycopy (1 : 13.0 µs)
|   |   |-->java/lang/StringBuffer  append (1 : 31.957 ms)
|   |   |   |-->java/lang/String  valueOf (1 : 31.529 ms)
|   |   |   |   |-->java/lang/Double  <clinit> (1 : 234.0 µs)
|   |   |   |   |   |-->java/lang/Double  longBitsToDouble (1 : 7.0 µs)
|   |   |   |   |   └-->java/lang/Class  getPrimitiveClass (1 : 21.0 µs)
|   |   |   |   └-->java/lang/Double  toString (1 : 28.178 ms)
|   |   |   |       |-->java/lang/FloatingDecimal  <clinit> (1 : 257.0 µs)
|   |   |   |       |-->java/lang/FloatingDecimal  <init> (1 : 16.465 ms)
|   |   |   |       |   |-->java/lang/Double  doubleToLongBits (1 : 20.0 µs)
|   |   |   |       |   └-->java/lang/FloatingDecimal  dtoa (1 : 7.576 ms)
|   |   |   |       |       |-->java/lang/FloatingDecimal  countBits (1 : 9.0 µs)
|   |   |   |       |       |-->java/lang/Math  <clinit> (1 : 2.363 ms)
|   |   |   |       |       |   |-->java/lang/Float  <clinit> (1 : 115.0 µs)
|   |   |   |       |       |   |   └->java/lang/Class  getPrimitiveClass (1 : 21.0 µs)
|   |   |   |       |       |   |-->java/lang/Float  floatToIntBits (1 : 7.0 µs)
```

FIGURE 17. 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.

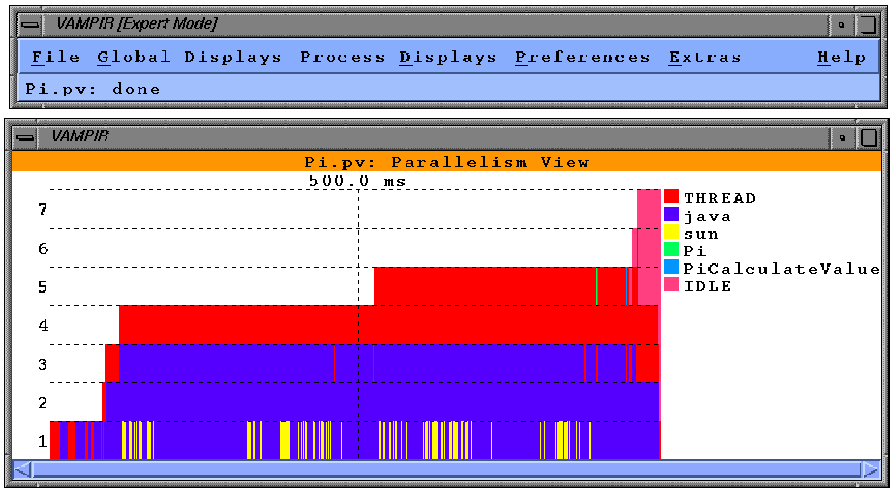


FIGURE 18. **Parallelism view**

By choosing other global displays such as **Summaric 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.

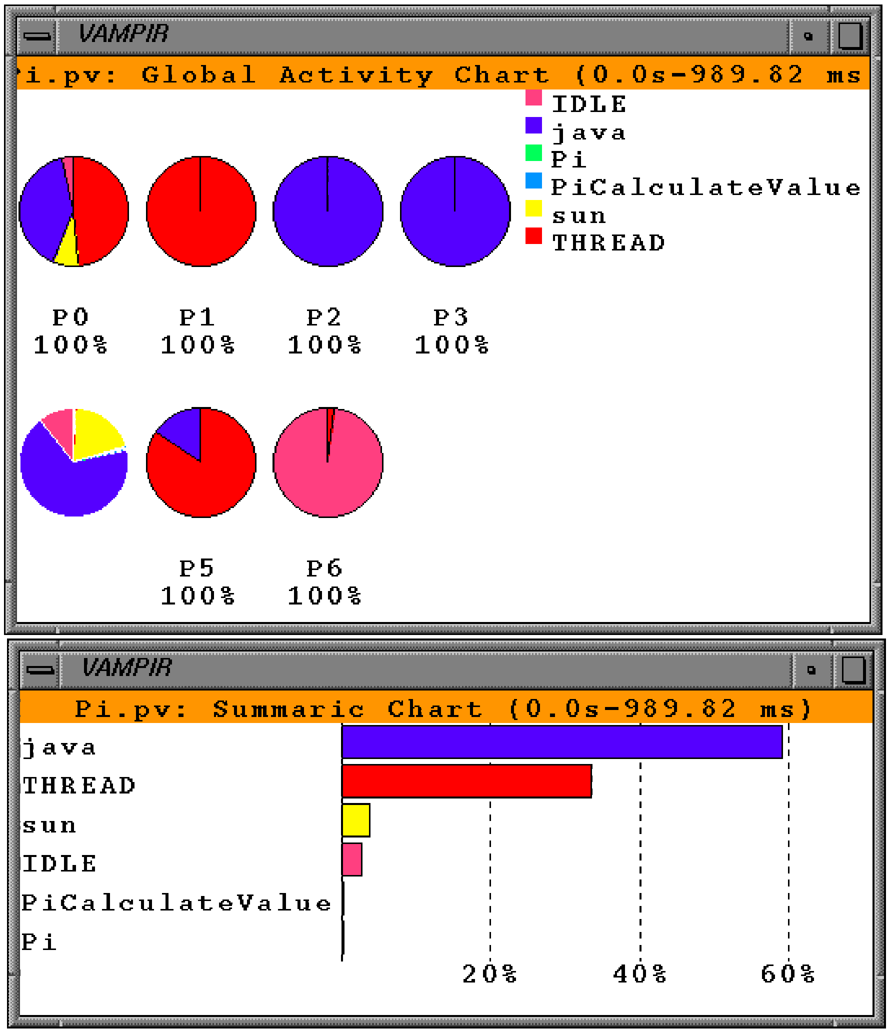


FIGURE 19. **Summaric chart and activity chart global displays highlight the groups that take the most time using pie charts and histograms respectively.**

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.

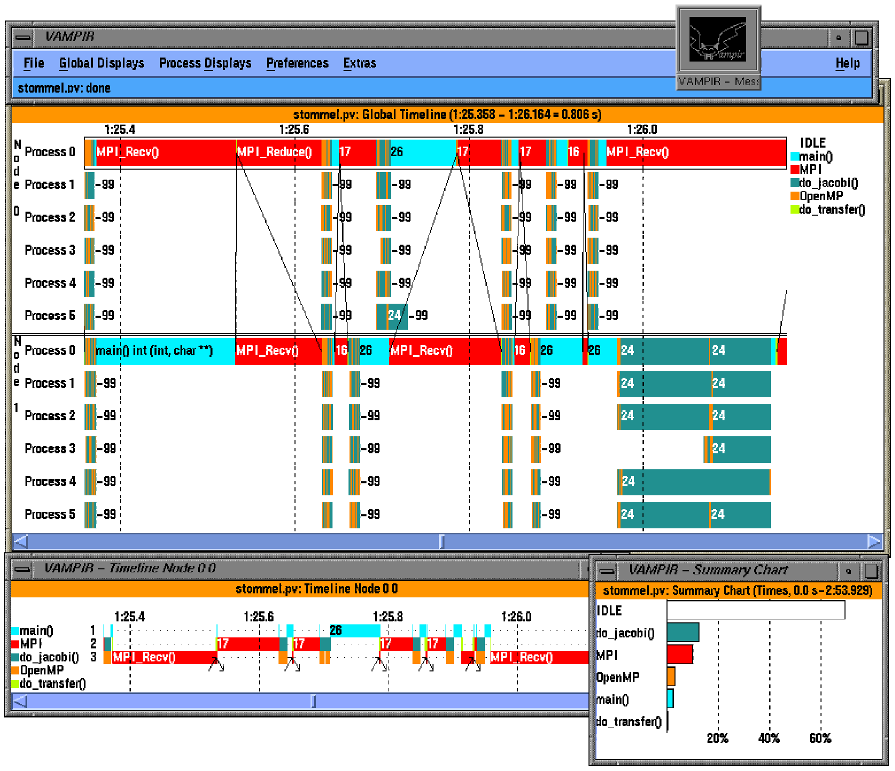
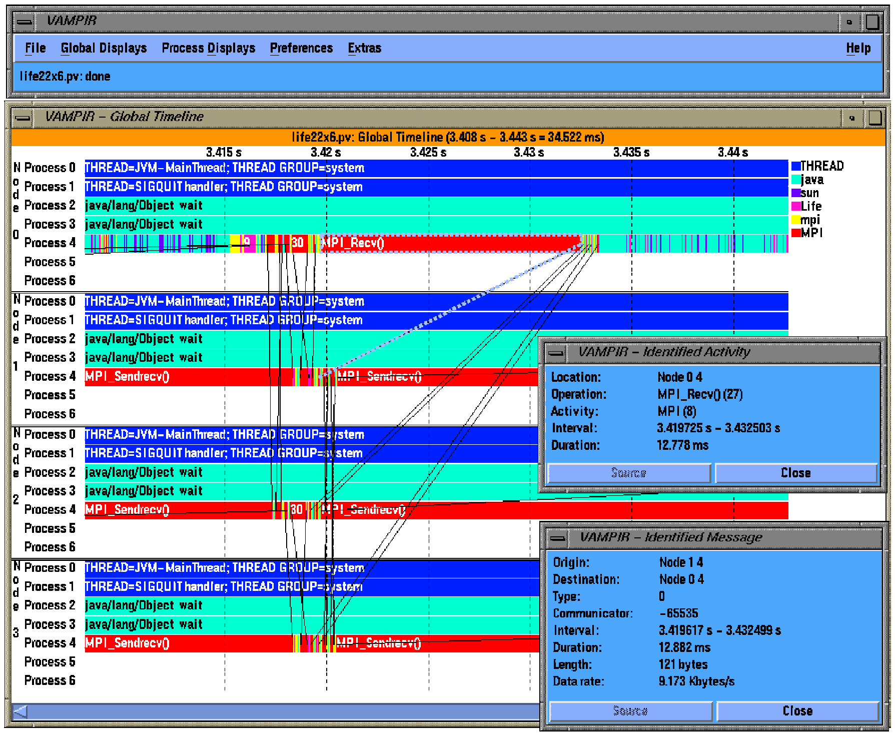


FIGURE 20. **Tracing an OpenMPI application with TAU**

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.



VAMPIR – Identified Activity

Location: Node 0 4
Operation: MPI_Recv() (27)
Activity: MPI (8)
Interval: 3.419725 s – 3.432503 s
Duration: 12.778 ms

Source Close

VAMPIR – Identified Message

Origin: Node 1 4
Destination: Node 0 4
Type: 0
Communicator: -65535
Interval: 3.419617 s – 3.432499 s
Duration: 12.882 ms
Length: 121 bytes
Data rate: 9.173 Kbytes/s

Source Close

FIGURE 21. Tracing hybrid (mixed-model) execution models with MPI and Java.

CHAPTER 6 *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.acl.lanl.gov/tau`

TAU was also released on the ACL Fall 1998 and 1999 CD-ROMs available from [ACL-SW-URL]:

`http://www.acl.lanl.gov/software`

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 homepage at [TAU-PUBS-URL]

Acknowledgements

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 and ASCI Level 3 grants.

University of Oregon

Forschungszentrum
Jülich GmbH

Los Alamos
NATIONAL LABORATORY

DOE 2000

CHAPTER 7

Appendix : Configuration Issues

Instructions for Installing TAU with POOMA:

POOMA (Parallel Object Oriented Methods and Applications) is an object-oriented framework for building high-performance computational science applications. [POOMA-URL]

SMARTS (Scalable Multithreaded Asynchronous Runtime System) is a run-time thread system for parallel execution on SMP systems[SMARTS-URL].

To build the POOMA library with profiling (using TAU and PDT), but no parallelism.

1. First make sure you have already installed the PDT package (Program Database Toolkit is a source code analysis package). PDT is available on the ACL CD-ROM in the pdtoolkit-1.1 directory, or from the ACL software web page [ACL-SW-URL]. Install the version of PDT. See the PDT INSTALL instructions for information on how to install PDT.
2. Then make sure you have already built and installed the TAU package. TAU is available on the ACL CD-ROM in the tau-2.7 directory, or from the ACL software web page [ACL-SW-URL]. Build the version of TAU without SMARTS or any other threads, and install it. POOMA can be built with profiling for all platforms where TAU can be compiled. See the TAU INSTALL instructions for information on how to build and install TAU.
3. Set the TAUDIR environment variable to the location of the top- level of the TAU installation, and the PDTDIR environment variable to the location of the PDT installation.
4. Follow the instructions above for building POOMA, but make sure you include the --profile flag when you run configure. You do not need to change the settings for the location of TAU in the config/arch/*.conf files if you set TAUDIR and PDTDIR; the configure script will use the config/arch/*.conf settings only if they are not set. A typical configure line that builds an optimized POOMA with profiling is

```
    configure <basic option list> --opt --profile
```
5. Then, set the POOMASUITE variable, and do ‘gmake; gmake install’ as normal.

6. To build POOMA applications in the examples/ subdirectories, follow the same steps as explained above when just compiling POOMA, except instead of using Makefile.user, compile with Makefile.profile. The TAU profiling mechanism requires extra steps to pre-process the the source files to automatically insert profiling code. Makefile.profile includes these extra steps.
7. If you follow these steps, then when a POOMA application it should produce a file or set of files profile.N.N.N. You can use the 'pprof' or 'racy' tools from the TAU installation to print and browse the profiling information.

To build the POOMA library with parallelism (using SMARTS), and profiling (using TAU):

WARNING: MIGHT NOT WORK EXCEPT UNDER IRIX 6.X AND KCC 3.3d OR LATER

Using both SMARTS and TAU together is a little bit more complicated that just building POOMA with one or the other of these packages, due to the fact that there is a slight circular dependency between SMARTS and TAU: TAU uses the thread capability from SMARTS, and SMARTS uses the profiling capability from TAU. So to build POOMA with both SMARTS and TAU, you should follow these steps exactly in the order they are given.

1. The first step is to unpack or copy the SMARTS, TAU and PDT packages to locations where they can be built and installed. For illustrative purposes, we'll say these packages are unpacked to `/usr/local/smarts-1.0.1`, `/usr/local/tau-2.7` and `/usr/local/pdtoolkit-1.1` respectively.
2. Next, install PDT. For example, on 64-bit SGI machines, you would cd to `/usr/local/pdtoolkit-1.1` and say:

   ```
   % ./configure -KCC -arch=IRIX64
   ```

   If you were building the packages with GNU or SGI CC, you would substitute -KCC with -GNU or -CC respectively. If you were building for SGI N32 or O32 architectures, you would substitute -arch=IRIX64 with -arch=IRIXN32 or -arch=IRIXO32 respectively. See the PDT installation instructions included in its INSTALL file for further details. Linux does not need any -arch configuration parameter to be specified. TAU uses PDT to build a source code instrumentor to instrument POOMA sources for profiling.
3. Next, configure, build, and install TAU. For example, on 64-bit SGI machines, you would cd to /usr/local/tau-2.7 and say:

```
% ./configure -arch=sgi64 -c++=KCC -cc=cc -SGITIM-
  ERS -smarts -tulipthread=/usr/local/smarts-1.0.1
  \
  -pdt=/usr/local/pdtoolkit-1.1
```

on Linux machines, you would say:

```
% ./configure -arch=linux -c++=KCC -cc=cc -smarts
  \  -tulipthread=/usr/local/smarts-1.0.1 \
  -pdt=/usr/local/pdtoolkit-1.1
```

to configure TAU for the specified architecture. To build and install TAU, you would say :

```
% gmake install
```

Here, the -arch flag selects the architecture (use -arch=linux for Linux builds), the -c++ and -cc flags select the compiler to use, -SGITIMERS is an sgi-specific flag to use fast sgi-only timer routines, and the -smarts and -tulipthread flags are used to indicate where to find the SMARTS headers. Then, this builds and installs TAU. Since no -prefix=<dir> flag was given, this installs TAU in the default location, which is the current directory (/usr/local/tau-2.7). See the TAU installation instructions in its INSTALL file for further details.

4. Now, configure, build, and install SMARTS. cd to the smarts-1.0.1 directory, and do the following (this example is for building SMARTS on a 64-bit SGI platform):

```
% ./configure --with-arch=iris4d --prefix \
  /usr/local/smarts-1.0.1 --with-taudir=/usr/
  local/tau-2.7 --enable-64bit --enable-profile
% gmake
% gmake install
```

Here, the --with-arch flag select the architecture to build for (use --with-arch=i386-linux for Linux), the --prefix flag selects where to install SMARTS after it has been built (in this case, it is in the same build directory), and the --with-taudir and --enable-profile flags indicate that the library should use TAU from the given location. --enable-64bit is only needed for 64-bit compilation on SGI's.

5. After TAU and SMARTS are built and installed, you can compile POOMA. Go to the pooma-2.2.0 directory, and do the following:

```
% setenv TAUDIR /usr/local/tau-2.7 (or where it was
   installed)
% setenv PDTDIR /usr/local/pdtoolkit-1.1 (or where
   it was installed)
% setenv SMARTSDIR /usr/local/smarts-1.0.1 (or
   where it was installed)
% ./configure --arch SGI64KCC --parallel --profile
   --suite PP --ex <other opts>
% setenv POOMASUITE PP
% gmake
```

Here, TAUDIR and PDT indicate where to find installed TAU and PDT components, and SMARTSDIR indicates where to find the installed SMARTS library. When these are set, run the POOMA configuration script as described for the basic case, but make sure to include the --parallel and --profile flags. After this, you set the POOMASUITE variable as is done in the previous descriptions of building the POOMA library, and run make.

6. To build POOMA applications in the examples/ subdirectories, follow the same steps as explained above when just compiling POOMA, except instead of using Makefile.user, compile with Makefile.profile. The TAU profiling mechanism requires extra steps to preprocess the the source files to automatically insert profiling code. Makefile.profile includes these extra steps.
7. If you follow these steps, then when a POOMA application it should produce a file or set of files profile.N.N.N. You can use the ‘pprof’ or ‘racy’ tools from the TAU installation to print and browse the profiling information. TAU can profile multithreaded runs as well as serial runs, and keeps track of what code was executed in what thread.

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 assmue 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 know 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 re-writing 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 8 References

URLs

[TAU-URL]	**http://www.acl.lanl.gov/tau**
[TAU-PUBS-URL]	**http://www.cs.uoregon.edu/research/paracomp/tau/papers.html**
[TAU-PGROUPS-URL]	**http://www.acl.lanl.gov/tau/docs/selec-tive.html**
[TAU-SECURITY-URL]	**http://www.acl.lanl.gov/tau/docs/faq.html#security**
[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
[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