Copyright © 1997-99

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

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

UO, ZAM AND LANL DISCLAIMS ALL WARRANTIES WITH REGARD
to this software, including all implied warranties of
merchantability and fitness. In no event shall the
university of Oregon, ZAM or LANL be liable for any
special, indirect or consequential damages or any
damages whatsoever resulting from loss of use, data or
profits, whether in an action of contract, negligence or
other tortious action, arising out of or in connection
with the use or performance of this software.

Copyright © 1999.

All rights reserved.
Contents

CHAPTER 1  Installation  1
Installing TAU  2
Examples:  5
Instructions for Installing TAU with POOMA:  6
Instructions for Installing TAU under Windows  9
Platforms Supported  12
Software Requirements  14

CHAPTER 2  Compiling  15
TAU stub Makefile  16
Enabling and Disabling the Instrumentation  18
Using TAU with MPI  18
Environment Variables  19

CHAPTER 3  Instrumentation  21
Automatic Instrumentation of C++ sources  22
C++ Instrumentation API  24
TAU Mapping API  33
C Instrumentation API  38
Fortran90 Instrumentation API  39
Summary  45

CHAPTER 4  Profiling  47
Running the application  48
Running a JAVA application with TAU  48
Running an application using DynInstAPI  49
Using Hardware Performance Counters  49
pprof  52
racy  53
CHAPTER 5  

*Tracing*  59

Generating Event Traces  60
Vampir: Visualizing TAU traces  61

CHAPTER 6  

*Summary*  69

Software Availability  70
Acknowledgements  70
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 C++ 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, 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.
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.
Installing TAU

-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. [URL: http://www.acl.lanl.gov/tulip]

-tulipthread=<directory> -smarts
Specifies SMAR TS (Shared Memory Asynchronous Runtime System) as the threads package to be used. <directory> gives the location of the SMAR TS root directory. [URL: http://www.acl.lanl.gov/smarts]

-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. [URL: http://www.acl.lanl.gov/pdtoolkit]

-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. [URL: http://www.fz-juelich.de/zam/PCL]

-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. Currently, the program needs to be re-linked with the libraries, but this restriction will be removed in the future.

[URL: http://www.cs.umd.edu/projects/dyninstAPI/]

-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.

[URL: http://www-unix.mcs.anl.gov/mpi/]

-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> option 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.

-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. [URL http://www.pallas.de ]

-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:

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

Instructions for Installing TAU with POOMA:

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, [URL: http://www.acl.lanl.gov/software/]. 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, [URL:http://www.acl.lanl.gov/software/]. 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 `gmak e; gmake install` as normal.
Instructions for Installing TAU with POOMA:

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 source files to automatically insert profiling code. Makefile.profile includes these extra steps.

7. If you follow these steps, then when a POOMA application 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 -SGITIMERS -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 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.
Installation

Section 1.

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

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

1. Download TAU. TAU is distributed as source and prebuilt libraries for Windows. 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\New-TauLib”, you would now have two new subdirectories of “C:\Program Files\DevStudio\MyProject\NewTauLib” named, “include\Profile” and “src\Profile”.

10 TAU Portable Profiling and Tracing Toolkit User's Guide
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\DIRECTOR\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 your list of libraries in “Project -> Settings -> Link” inside VC++. You must then make sure that the library is in a location known to VC++. You can do this in your “Tools ->Options->Directories->Library files” section of VC++

Section 2.

The Windows port ships with a prebuilt version of pprof which can be used to view your profiling data (See the TAU documentation for more details). Make sure that pprof.exe is in your current path. It can be found in C:\TAU-SOURCE-DIR\windows\bin. Currently, there is no version of Racy for Windows, however, we are 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 contact Robert Ansell-Bell (bertie@cs.uoregon.edu) or send mail to tau-bugs@cs.uoregon.edu.

Platforms Supported

TAU has been tested on the following platforms:

1. SGI


2. LINUX PCs
Platforms Supported

On Linux based Intel x86 PC clusters, KAI KCC, g++, egcs (GNU), pgCC (PGI) [URL: http://www.pgroup.com], FCC (Fujitsu) [URL: http://www.tools.fujitsu.com/linux/index.shtml] 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 and GNU g++ work with TAU.

4. IBM AIX

On IBM SP2 and AIX systems, KAI KCC 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.

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 can be downloaded from URL: [http://www.scriptics.com](http://www.scriptics.com) 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 URL: [http://www.acl.lanl.gov/tau/docs/faq.html#security](http://www.acl.lanl.gov/tau/docs/faq.html#security) for instructions. Contact your system administrator if your X-server is not configured for generating Xauth cookies.
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:
**TAU stub Makefile**

```
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 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). 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 on the link command line, the TAU stub makefile variable TAU_DISABLE. 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. To better manage different experiments, set the environment variables

- **PROFILEDIR** to name the directory that should contain the profile data files
- **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).

For example:

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

**NOTE:** For instructions on running the program with JAVA, please refer to Chapter 4 [Profiling] Section “Running an Application”.

**NOTE:** TAU also uses the environment variable **PCL_EVENT** to specify the hardware performance counter to be used when `-pcl=<dir>` configure option is used. See section “Using Hardware Performance Counters” in Chapter 4 for further details.
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.
Instrumentation

Automatic Instrumentation of C++ sources

tau_instrumentor inserts TAU instrumentation macros in C++ source code using PDT
[URL: http://www.acl.lanl.gov/pdtoolkit].

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
Automatic Instrumentation of C++ sources

# Modified Rules

## Modified Rules

```bash
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 $(TARGET).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 $(TARGET).inst.cpp

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

clean:
```
$(RM) $(TARGET).o $(TARGET).inst.cpp $(TARGET).pdb
```

## Modified Rules

```bash
$(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 "*******************************************************************************"
```
Instrumentation

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++ Instrumentation 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

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

**TAU_PROFILE** function_name, type, group);

Args:
- 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 to which it belongs. 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:

```cpp
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:

```cpp
template<class PLayout>
ostream& operator<<(ostream& out, const ParticleBase<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<UniformCartesian<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);

```cpp
template<class PLayout>
unsigned ParticleBase<PLayout>::GetMessage(Message& msg, int node) {
    TAU_TYPE_STRING(taustr, CT(*this) + " unsigned (Message, 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<UniformCartesian<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:
```cpp
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:

```c++
template< class T, unsigned Dim >
void BareField<T,Dim>::fillGuardCells(bool reallyFill)```

```c++```
{ // 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, i.e., if this macro is not used, functions belonging to all profile groups are enabled.

Example:

```c
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. TAU uses the HPC++ model of computation, where the tasks are identified using node, context and thread. 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_PROFILE_REGISTER_THREAD();**

To register a thread with the profiling system, invoke the TAU_PROFILE_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_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:

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

**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:

```c
int ArraySend(int arrayid)
{
    TAU_REGISTER_EVENT(taumsgsize, "Size of message associated 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 a 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 macro. 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.

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,

```cpp
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 e.g.:
// 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(MakeExpression<LHS>::make(lhs), DataBlockTag<CountBlocks>(), SumCombineTag()) + 
  forEachTag(MakeExpression<RHS>::make(rhs), DataBlockTag<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_MAPPINGPROFILE(TauMapFI)
   // Just evaluate the expression.
   KernelEvaluator<EvalTag>().evaluate(lhs_m,op_m,rhs_m);
   // we could release the locks here or in dtor
}

TAU_MAPPINGPROFILE_TIMER(timer, FuncIdVar);
Arguments: Profiler timer;
FunctionInfo * FuncIdVar;

TAU_MAPPINGPROFILE_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_MAPPINGOBJECT 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_MAPPINGPROFILE_START(timer, tid);
Argument:
Profiler timer;
int tid;

TAU_MAPPINGPROFILE_START starts the timer that is created using
TAU_MAPPINGPROFILE_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:

```c
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 Instrumentation 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:

```c
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 Instrumentation API

The Fortran90 TAU API allows source code written in Fortran to be instrumented for TAU. This API comprises 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..., 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
dend

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. TAU uses the HPC++ model of computation, where the tasks are identified using node, context and thread. 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_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:

ingenent 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
type of event
integer destination
receiver process
integer length
size of message

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 viz., 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. We are working on a Fortran 90 instrumentor, which should be available in the near future.
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 the performance of an application. 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 \texttt{--profile} to specify the statements to be profiled. For example:

\begin{verbatim}
% setenv PROFILEDIR /home/sameer/profiledata/experiment55
% mpirun -np 4 matrix
\end{verbatim}

This profiles all routines

\begin{verbatim}
% mpirun -np 4 matrix --profile io+field+2
\end{verbatim}

The above profiles routines belonging to TAU\_IO, TAU\_FIELD and TAU\_USER2 profile groups. For a detailed list of groups, please refer to URL: http://www.acl.lanl.gov/tau (Docs->Tutorial->Selective Profiling section)

Running a JAVA application with TAU

Java Applications are profiled/traced using the \texttt{-XrunTAU} commandline parameter as shown below:

\begin{verbatim}
% cd tau/examples/java/pi
% setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:/home/tau/solaris2/lib
% java -XrunTAU Pi
\end{verbatim}

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

\begin{verbatim}
% java -XrunTAU -Djava.compiler= Pi
\end{verbatim}

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

\begin{verbatim}
% java -classic -XrunTAU -Djava.compiler= Pi
\end{verbatim}
Running an application using DynInstAPI

Running the application generates profile files with names having the form profile.<node>.<context>.<thread>. These files can be analyzed using pprof or 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. Performance Counter Library (PCL, URL:http://www.fz-juelich.de/zam/PCL/) package provides a uniform interface to access these performance counters. TAU uses PCL to access these hardware performance counters. To use PCL, download and install PCL. Then, configure TAU using the -pcl=<dir> configuration command-line option to specify the location of PCL. 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 to specify which hardware performance counter TAU should use while profiling the application. For example to measure the floating point operations in routines,

% ./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**

<table>
<thead>
<tr>
<th>PCL_EVENT</th>
<th>Event Measured</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCL_L1CACHE_READ</td>
<td>L1 (Level one) cache reads</td>
</tr>
<tr>
<td>PCL_L1CACHE_WRITE</td>
<td>L1 cache writes</td>
</tr>
<tr>
<td>PCL_L1CACHE_READWRITE</td>
<td>L1 cache reads and writes</td>
</tr>
<tr>
<td>PCL_L1CACHE_HIT</td>
<td>L1 cache hits</td>
</tr>
<tr>
<td>PCL_L1CACHE_MISS</td>
<td>L1 cache misses</td>
</tr>
<tr>
<td>PCL_L1DCACHE_READ</td>
<td>L1 data cache reads</td>
</tr>
<tr>
<td>PCL_L1DCACHE_WRITE</td>
<td>L1 data cache writes</td>
</tr>
<tr>
<td>PCL_L1DCACHE_READWRITE</td>
<td>L1 data cache reads and writes</td>
</tr>
<tr>
<td>PCL_L1DCACHE_HIT</td>
<td>L1 data cache hits</td>
</tr>
<tr>
<td>PCL_L1DCACHE_MISS</td>
<td>L1 data cache misses</td>
</tr>
<tr>
<td>PCL_L1ICACHE_READ</td>
<td>L1 instruction cache reads</td>
</tr>
<tr>
<td>PCL_L1ICACHE_WRITE</td>
<td>L1 instruction cache writes</td>
</tr>
<tr>
<td>PCL_L1ICACHE_READWRITE</td>
<td>L1 instruction cache reads and writes</td>
</tr>
<tr>
<td>PCL_L1ICACHE_HIT</td>
<td>L1 instruction cache hits</td>
</tr>
<tr>
<td>PCL_L1ICACHE_MISS</td>
<td>L1 instruction cache misses</td>
</tr>
<tr>
<td>PCL_L2CACHE_READ</td>
<td>L2 (Level two) cache reads</td>
</tr>
<tr>
<td>PCL_L2CACHE_WRITE</td>
<td>L2 cache writes</td>
</tr>
<tr>
<td>PCL_L2CACHE_READWRITE</td>
<td>L2 cache reads and writes</td>
</tr>
<tr>
<td>PCL_L2CACHE_HIT</td>
<td>L2 cache hits</td>
</tr>
<tr>
<td>PCL_L2CACHE_MISS</td>
<td>L2 cache misses</td>
</tr>
<tr>
<td>PCL_L2DCACHE_READ</td>
<td>L2 data cache reads</td>
</tr>
<tr>
<td>PCL_L2DCACHE_WRITE</td>
<td>L2 data cache writes</td>
</tr>
<tr>
<td>PCL_L2DCACHE_READWRITE</td>
<td>L2 data cache reads and writes</td>
</tr>
<tr>
<td>PCL_L2DCACHE_HIT</td>
<td>L2 data cache hits</td>
</tr>
<tr>
<td>PCL_L2DCACHE_MISS</td>
<td>L2 data cache misses</td>
</tr>
<tr>
<td>PCL_L2ICACHE_READ</td>
<td>L2 instruction cache reads</td>
</tr>
<tr>
<td>PCL_L2ICACHE_WRITE</td>
<td>L2 instruction cache writes</td>
</tr>
<tr>
<td>PCL_L2ICACHE_READWRITE</td>
<td>L2 instruction cache reads and writes</td>
</tr>
</tbody>
</table>
TABLE 1. Events measured by setting the environment variable PCL_EVENT in TAU

<table>
<thead>
<tr>
<th>PCL_EVENT</th>
<th>Event Measured</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCL_L2ICACHE_HIT</td>
<td>L2 instruction cache hits</td>
</tr>
<tr>
<td>PCL_L2ICACHE_MISS</td>
<td>L2 instruction cache misses</td>
</tr>
<tr>
<td>PCL_TLB_HIT</td>
<td>TLB (Translation Lookaside Buffer) hits</td>
</tr>
<tr>
<td>PCL_TLB_MISS</td>
<td>TLB misses</td>
</tr>
<tr>
<td>PCL_ITLB_HIT</td>
<td>Instruction TLB hits</td>
</tr>
<tr>
<td>PCL_ITLB_MISS</td>
<td>Instruction TLB misses</td>
</tr>
<tr>
<td>PCL_DTLB_HIT</td>
<td>Data TLB hits</td>
</tr>
<tr>
<td>PCL_DTLB_MISS</td>
<td>Data TLB misses</td>
</tr>
<tr>
<td>PCL_CYCLES</td>
<td>Cycles</td>
</tr>
<tr>
<td>PCL_ELAPSED_CYCLES</td>
<td>Cycles elapsed</td>
</tr>
<tr>
<td>PCL_INTEGER_INSTR</td>
<td>Integer instructions executed</td>
</tr>
<tr>
<td>PCL_FP_INSTR</td>
<td>Floating point (FP) instructions executed</td>
</tr>
<tr>
<td>PCL_LOAD_INSTR</td>
<td>Load instructions executed</td>
</tr>
<tr>
<td>PCL_STORE_INSTR</td>
<td>Store instructions executed</td>
</tr>
<tr>
<td>PCL_LOADSTORE_INSTR</td>
<td>Loads and stores executed</td>
</tr>
<tr>
<td>PCL_INSTR</td>
<td>Instructions executed</td>
</tr>
<tr>
<td>PCL_JUMP_SUCCESS</td>
<td>Successful jumps executed</td>
</tr>
<tr>
<td>PCL_JUMP_UNSUCCESS</td>
<td>Unsuccessful jumps executed</td>
</tr>
<tr>
<td>PCL_JUMP</td>
<td>Jumps executed</td>
</tr>
<tr>
<td>PCL_ATOMIC_SUCCESS</td>
<td>Successful atomic instructions executed</td>
</tr>
<tr>
<td>PCL_ATOMIC_UNSUCCESS</td>
<td>Unsuccessful atomic instructions executed</td>
</tr>
<tr>
<td>PCL_ATOMIC</td>
<td>Atomic instructions executed</td>
</tr>
<tr>
<td>PCLSTALL_INTEGER</td>
<td>Integer stalls</td>
</tr>
<tr>
<td>PCLSTALL_FP</td>
<td>Floating point stalls</td>
</tr>
<tr>
<td>PCLSTALL_JUMP</td>
<td>Jump stalls</td>
</tr>
<tr>
<td>PCLSTALL_LOAD</td>
<td>Load stalls</td>
</tr>
<tr>
<td>PCLSTALL_STORE</td>
<td>Store Stalls</td>
</tr>
<tr>
<td>PCLSTALL</td>
<td>Stalls</td>
</tr>
<tr>
<td>PCL_MFLOPS</td>
<td>Millions of floating point operations/second</td>
</tr>
</tbody>
</table>

TAU Portable Profiling and Tracing Toolkit User's Guide 51
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 function
  -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
[ ] : prints information about all contexts/threads for specified nodes

<table>
<thead>
<tr>
<th>PCL_EVENT</th>
<th>Event Measured</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCL_IPC</td>
<td>Instructions executed per cycle</td>
</tr>
<tr>
<td>PCL_L1DCACHE_MISSRATE</td>
<td>Level 1 data cache miss rate</td>
</tr>
<tr>
<td>PCL_L2DCACHE_MISSRATE</td>
<td>Level 2 data cache miss rate</td>
</tr>
<tr>
<td>PCL_MEM_FP_RATIO</td>
<td>Ratio of memory accesses to FP operations</td>
</tr>
</tbody>
</table>
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.
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:

URL: http://www.acl.lanl.gov/tau/docs/faq.html#security

After the project is created or selected, the main racy window appears.
This shows the relative time spent in each function as a horizontal bar graph. 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.

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

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.
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 (URL: http://www.csi.uoregon.edu/parp)

FIGURE 8. Racy displays user defined event profiles as well.

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

FIGURE 9. User defined events may be used to track memory bugs.
This chapter discusses how TAU can be used to generate event traces.

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

tau_convert can also be used to convert traces to the Vampir trace format (URL: http://www.pallas.de). 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 &
```

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

This describes the types of visualizations of trace data that Vampir provides. Event traces can show, using a global timeline display, when routine transitions take place on different processes.
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:
Vampir is used to display scheduling of work packets or iterates in SMARS, the Shared Memory Asynchronous Runtime System (URL: http://www.acl.lanl.gov/smarts):

FIGURE 12. Scheduling work packets in SMARS
In the next figure, we see the symbol legend and the dynamic call tree.

**FIGURE 13.** Vampir symbol legend and calltree display
Vampir has been used to compare the scheduling policies of the SMAR TS package.

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 14. Comparing scheduling policies in SMAR TS

FIGURE 15. 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.

![Call tree display of a thread showing the dynamic call tree annotated with performance metrics.](image)

**FIGURE 16.** 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.
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 18. Summaric chart and activity chart global displays highlight the groups that take the most time using pie charts and histograms respectively.
This chapter discusses the availability of the software and related issues.
Summary

Software Availability

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

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

TAU was also released on the ACL Fall 1998 CD-ROM available from:

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:

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

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 LANL ASCI Level 3 grants.