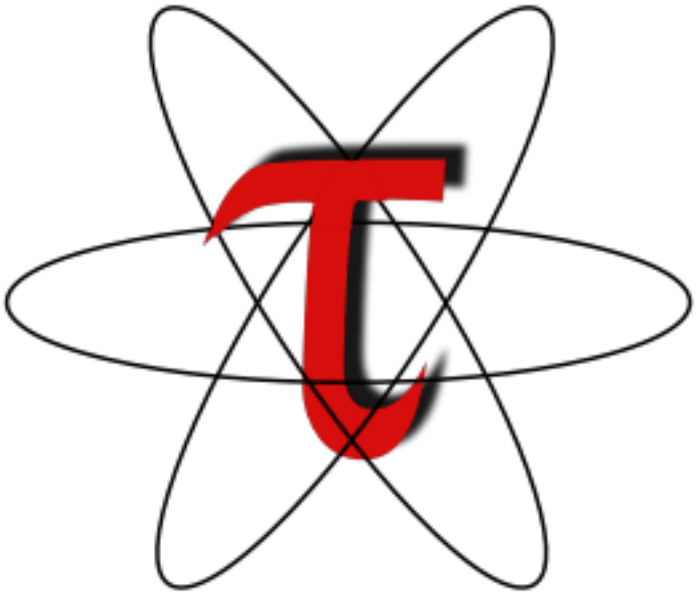
TAU User's Guide



TAU User's Guide

Version 2.15

TAU can be found on the web at: http://www.cs.uoregon.edu/research/tau

Table of Contents

List of Figures

List of Tables

Part Preface. TAU Tutorial

Table of Contents

TAU Tutorial

1. Gather information

Before we began installing PDT and TAU you will find it helpful to gather information about your computing environment. TAU and PDT require both a C and C++ compiler. Furthermore this tutorial uses MPICH. Find out the following information about your computing environment before we began:

- The path of your C compiler
- The path of your C++ compiler

For the remaining of this tutorial we will assume that your C compiler is xlc, your C++ compiler is xlC.

2. Installing PDT

To take advantage of TAU's automatic instrumentation features, you will need to install the Program Database Toolkit (PDT). Download the latest version from the PDT pages and put the tar.gz package in the location that you want to install PDT. For this installation, we will assume that you are using IBM's Fortran and C/C++ compilers, with an mpich installation.

Start by uncompressing the PDT package and moving into the PDT directory.

```
%> tar -xvzf pdtoolkit-3.4.tar.gz
%> cd pdtoolkit-3.4
```

You can get a sense for what options you can configure PDT with by entering:

```
%> ./configure
Program Database Toolkit (PDT) Configuration
----------------------------------------
Looks like a Linux machine ...
Looking for C++ compilers .... done
Usage: ./configure [-KAI|-KCC|-GNU|-CC|-c++|-cxx|-xlC|-pgCC|-icpc|-ecpc]
                   [-arch=ibm64|ibm64linux|IRIXO32|IRIXN32|IRIX64] [-help]
                   [-compdir=<compdir>>]
                   [-enable-old-headers]
                   [-useropt=<options>>]
                   [-prefix=<dir>]
                   [-exec-prefix=<dir>>]
```

We will configure PDT for use the the Fortran xlF, xlc, and xlC compilers. To configure PDT, type

```
%> ./configure -xlC

Program Database Toolkit (PDT) Configuration
----------------------------------------
Looks like a Linux machine ...
Looking for C++ compilers .... done
==> Using /opt/ibmcmp/vacpp/6.0/bin/xlC
Unpacking ppc64/bin ...
```

```
==> ARCH is PPCLINUX
==> PLATFORM is ppc64
==> Default compiler options are -O2
==> Makefiles were configured
==> cparse was configured
==> cxxparse was configured
==> f90parse was configured
==> f95parse was configured

Configuration is complete!

Run "make" and "make install"
Add "/home/users/hoge/pdtoolkit-3.4/ppc64//bin" to your path
```

Add the specified directory to your path. In bash, for example you could enter

```
%> export PATH=$PATH:/home/users/hoge/pdtoolkit-3.4/ppc64/bin
```

Now, build and install PDT. Unless you specify a different location to install PDT, it will be placed in the current working directory.

```
%> make
...
%> make install
```

Now you're ready to proceed with the TAU installation.

3. Installing TAU

Download the latest version of TAU from the TAU home. Place the distribution In the directory that you want to install TAU. Type

```
%> tar -xvzf tau_latest.tar.gz
%> cd tau-2.14.6
```

We will be installing TAU once again assuming that we are using the IBM compilers (xlf, xlc and xlC), and an MPICH installation. Note where your MPICH installation resides, and configure TAU by entering (replacing the MPICH specifics with those in your local system.

```
%> ./configure -c++=xlC -cc=xlc -fortran=ibm \
  -mpiinc=/opt/osshpc/mpich-1.2.5/32/ch_shmem/include \
  -mpilib=/opt/osshpc/mpich-1.2.5/32/ch_shmem/lib \
  -pdt=/home/users/hoge/pdtoolkit-3.4
```

Add the TAU directory to your path and install.

```
%> export PATH=$PATH:/home/users/hoge/tau-2.14.6/ppc64/bin
%> make install
```

TAU is installed, and you're ready to start profiling your code.

4. Automatic instrumentation using TAU Compiler

For this section of the tutorial we will be using the files found in the examples/taututorial directory of the tau distribution. To start, there are two files of note: computePi.cpp and Makefile. computePi.cpp is a C++ program that uses an MPI client-server model to estimate the value of Pi. The server accepts requests for random numbers from the clients, and returns an array of random numbers to the clients. The clients use these values to estimate Pi using a dart-throwing method. When the clients have converged to a satisfactory tolerance, they signal their completion to the server and the program exits.

Build computePi.cpp as you would any c++ mpi application. Test the program in your MPI environment. For mpich, the command might be

```
%> mpirun -np 5 ./computePi
Pi is 3.14226
```

to run the program on 5 nodes. Note that this program requires at least two nodes to be running! Once you've confirmed that the program ran successfully, try timing it to get a sense of how long it takes to run.

```
%> time mpirun -np 5 ./computePi
Pi is 3.14226

real    0m2.012s
user    0m1.570s
sys     0m0.330s
```

Now let us rebuild computePi to be instrumented with tau. First set the environment variable TAU_MAKEFILE to the location of the tau makefile, for example:

```
% export TAU_MAKEFILE=/home/users/hoge/tau2/ia64/lib/Makefile.tau-mpi-pdt
% make
```

Assuming that all goes well, the computePi program will have been automatically built with TAU instrumentation. Run the program as you would any MPI program, i.e.

```
%> mpirun -np 5 ./computePi
Pi is 3.14226
```

TAU generates a profile file for every node the program is run on. You can see these files by doing a directory listing.

```
%> ls profile*
profile.0.0.0  profile.1.0.0  profile.2.0.0
profile.3.0.0  profile.4.0.0
```

Now you're ready to view the output of TAU. If you've added the TAU binary directory to your path you can launch the TAU profile viewer, Paraprof.

```
%> paraprof
```

Enjoy exploring the performance data displayed by Paraprof. A complete description of how to use Paraprof is outside the scope of this document. Please see the Paraprof Manual [http://www.cs.uoregon.edu/research/tau/docs/paraprof/index.html] for more information.

When you ran the instrumented version of computePi you might have noticed that it took significantly longer to run than the non-instrumented version. Let's verify this behavior.

```
%> time mpirun -np 5 ./computePi
real    0m37.750s
user    0m37.370s
sys     0m0.320s
```

On my system, this is an order of magnitude overhead. For multi-processor MPI programs, this is an unacceptable amount of overhead. However, TAU offers a method for dealing with this added overhead, which we'll explore that in the next section.

5. TAU throttle

Tau_THROTTLE is designed to reduce the computational overhead associated with instrumenting a program with TAU. This usually takes the form of selectively instrumented some functions but not others. This can be done manually, but TAU_THROTTLE with do this automatically by helping you develop a criterion to decide which function to instrument.

First gather the profiles into a single profile file,

```
%>pprof -d > pprof.dat
```

To see the combined profile data type:

```
%>pprof -c
```

Looking at the #call column we see that the function computeRandom() is called about 20,000,000 times. It is functions like these that contribute greatly to the overhead associated with instrumenting a program. You see, when a function is entered and exited a small amount of tauinstrument code is executed. When a function is called millions of times even that small amount of code can cause a slow down in execute time.

Let us tell tau not to instrument functions like computeRandom(), this will remove the computational overhead of instrumenting a function that is called 20 millions times. To do this, set these environment variables:

```
%> export TAU_THROTTLE=1
%> export TAU_THROTTLE_NUMCALLS=400000
%> export TAU_THROTTLE_PERCALL=3000
```

This will tell tau not to profile any functions which are called more than 400000 times and their inclusive time per call is less than 3 seconds.

Let us now see how much time it takes to run computePi,

```
%> time mpirun -np 5 ./computePi
Pi is 3.14226

real    0m2.123s
user    0m1.760s
sys     0m0.270s
```

On my machine computePi runs at about 10% overhead this is from 2000% overhead before using TAU_THROTTLE. Not only does TAU_THROTTLE help reduce the overall runtime overhead of in-

strumenting a program, it also, as we will see in the next section, increases the accuracy of the resulting profile data.

6. ParaProf

Paraprof is a tool that shows you a graphical representation of the profiles generated by tau_compiler. Documentation on setting up and using paraprof is outside the scope of this tutorial, see the ParaProf Manual [http://www.cs.uoregon.edu/research/tau/docs/paraprof/index.html]

Here is the results of using TAU_THROTTLE are displayed in paraprof. Notice that before TAU_THROTTLE that the number of calls made to functions other than computeRandom() is obscured. But after TAU_THROTTLE they can be seen clearly.

Figure 1. A graph of the number of calls by function before TAU_THROTTLE

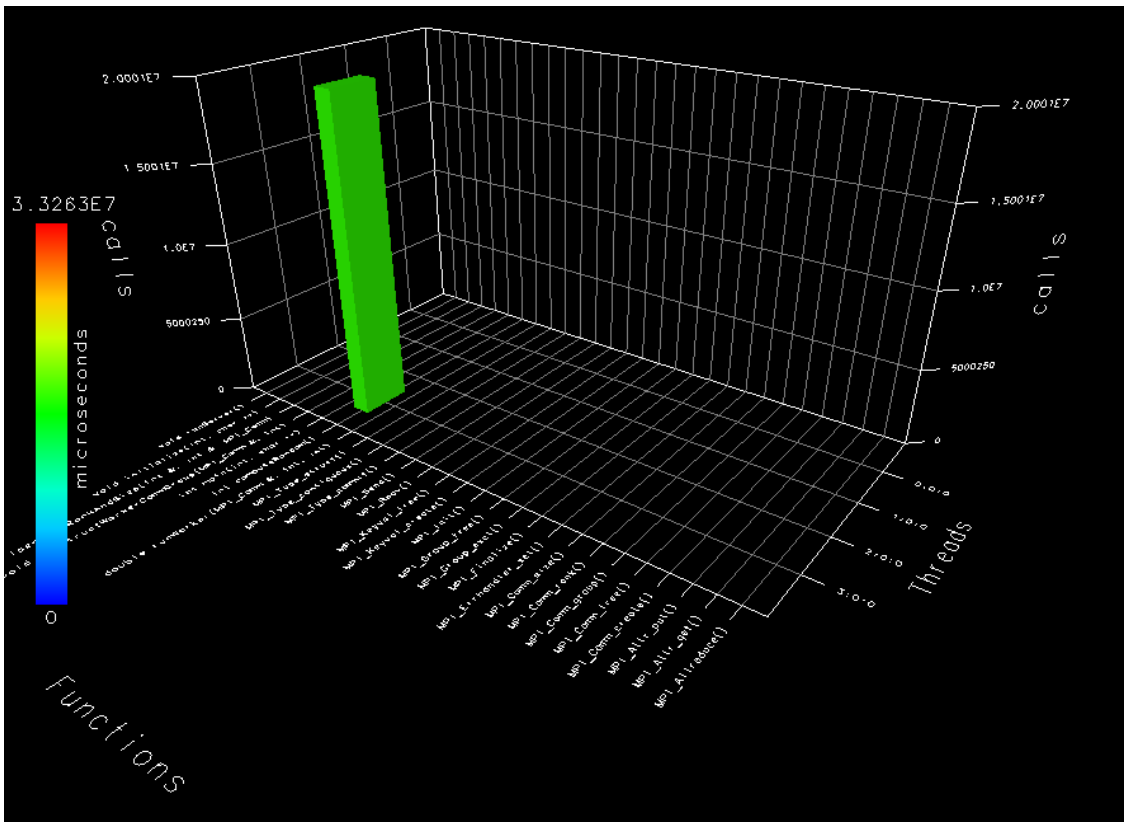
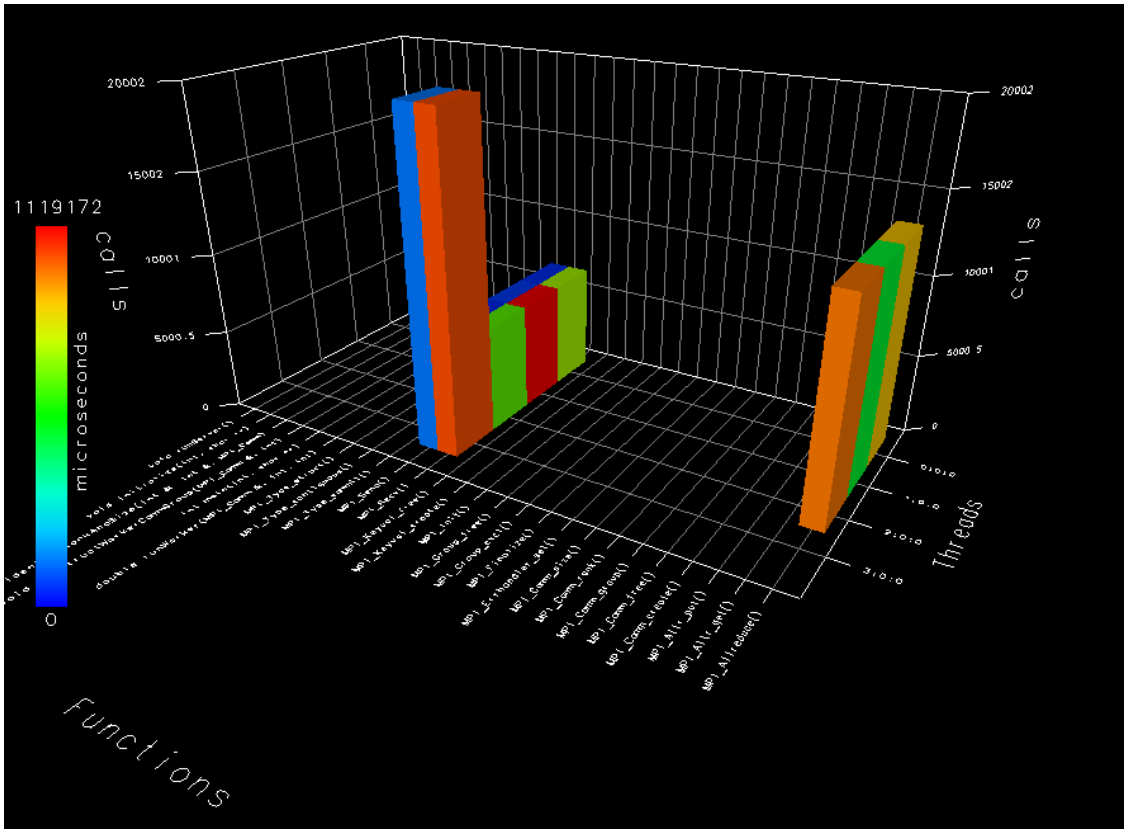


Figure 2. A graph of the number of calls by function after TAU_THROTTLE



Congratulations, you have successfully instrumented a C++ program with tau compiler. Furthermore the you know the basics of TAU_THROTTLE and how it can help reduce the overhead of instrumenting a program. For more information on tau features see the Tau Documentation. [http://www.cs.uoregon.edu/research/tau/docs.php]

Part I. Generating Performance Data

Table of Contents

Chapter 1. Installation

TAU (Tuning and Analysis Utilities) is a portable profiling and tracing toolkit for performance analysis of parallel programs written in Fortran, C++, C, Java and Python. The model that TAU uses to profile parallel, multi-threaded programs maintains performance data for each thread, context, and node in use by an application. The profiling instrumentation needed to implement the model captures data for functions, methods, basic blocks, and statement execution at these levels. All C++ language features are supported in the TAU profiling instrumentation including templates and namespaces, which is available through an API at the library or application level. The API also provides selection of profiling groups for organizing and controlling instrumentation. The instrumentation can be inserted in the source code using an automatic instrumentor tool based on the Program Database Toolkit (PDT), dynamically using DynInstAPI, at runtime in the Java virtual machine, or manually using the instrumentation API. TAU's profile visualization tool, paraprof, provides graphical displays of all the performance analysis results, in aggregate and single node/context/thread forms. The user can quickly identify sources of performance bottlenecks in the application using the graphical interface. In addition, TAU can generate event traces that can be displayed with the Vampir or Paraver trace visualization tools. This chapter discusses installation of the TAU portable profiling package.

1.1. Installing TAU

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

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

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

The following command-line options are available to configure:

1.1.1. Available configuration options

- `-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 IBM BGL, the user should specify bgl as the architecture. For SGI, the user can specify either of sgi32, sgin32 or sgi64 for 32, n32 or 64 bit compilation modes respectively. The files are installed in the <architecture>/bin and <architecture>/lib directories.

- `-c++=<C++ compiler>`

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

PGI).

- `-cc=<C Compiler>`

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

- `-pdt_c++=<C++ Compiler>`

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

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

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

  ```
  -arch=bgl -pdt=/usr/pdtoolkit-3.4 -pdt_c++=xlC -mpi
  ```

  uses PDT, MPI for IBM BG/L and specifies the use of the front-end xlC compiler for building tau_instrumentor.

- `-fortran=<Fortran Compiler>`

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

- `-tag=<Unique Name>`

  Specifies a tag in the name of the stub Makefile and TAU makefiles to uniquely identify the installation. This is useful when more than one MPI library may be used with different versions of compilers. e.g.,

  ```
  % configure -c++=icpc -cc=icc -tag=intel71-vmi   \
              -mpiinc=/vmi2/mpich/include
  ```

- `-pthread`

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

- `-charm=<directory>`

  Specifies charm++ (converse) threads as the thread package to be used.

- `-tulipthread=<directory> -smarts`

  Specifies SMARTS (Shared Memory Asynchronous Runtime System) as the threads package to be used. <directory> gives the location of the SMARTS root directory. Smarts [http://www.cs.uoregon.edu/research/tau/users/smarts.php]

- `-openmp`

  Specifies OpenMP as the threads package to be used. Open MPI [http://www.open-mpi.org/]

- `-opari=<dir>`

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

- `-opari_region`

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

- `-opari_construct`

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

- `-pdt=<directory>`

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

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

- `-pcl=<directory>`

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

  NOTE: If you want to profile multiple PCL counters set the "-MULTIPLECOUNTERS" options as well. And insteed of using the PCL environment variable use COUNTER1, COUNTER2, ... COUNTER25 environment variables to specify the type of counter to profile.. PCL [http://www.fz-juelich.de/zam/PCL/]

- `-papi=<directory>`

  Specifies the location of the installed PAPI (Performance Data Standard and API) root directory. PCL provides a common interface to access hardware performance counters and timers on modern microprocessors. Most modern CPUs provide on-chip hardware performance counters that can record several events such as the number of instructions issued, floating point operations performed, the number of primary and secondary data and instruction cache misses, etc. To measure floating point instructions, set the environment variable PAPI_EVENT to PAPI_FP_INS (for example). This option (by default) specifies the use of hardware performance counters for profiling (instead of time). When used in conjunction with -PAPIWALLCLOCK or -PAPIVIRTUAL, it specifies the use of wallclock or virtual process timers respectively. See the section "Using Hardware Performance Counters" in Chapter 4 for details regarding its usage.

NOTE: If you want to profile multiple PAPI counters set the "-MULTIPLECOUNTERS" options as well. And insteed of using the PAPI_EVENT environment variable use COUNTER1, COUNTER2, ... COUNTER25 environment variables to specify the type of counter to profile. PAPI [http://icl.cs.utk.edu/papi/]

- `-PAPIWALLCLOCK`

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

- `-PAPIVIRTUAL`

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

- `-CPUTIME`

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

- `-MULTIPLECOUNTERS`

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

  NOTE: The default measurement option in TAU is to use the wallclock time, which is the total time a program takes to execute, including the time when it is waiting for resources. It is the time measured from a real-time clock. The process virtual time (-PAPIVIRTUAL) is the time spent when the process is actually running. It does not include the time spent when the process is swapped out waiting for CPU or other resources and it does not include the time spent on behalf of the operating system (for executing a system call, for instance). It is the time spent in the "user" mode. The CPUTIME on the other hand, includes both the time the process is running (process virtual time) and the time the system is providing services for it (such as executing a system call). It is the sum of the process virtual (user) time and the system time (See man getrusage()).

  NOTE: If "-TRACE" and "-MULTIPLECOUNTERS" options are both set the environment variable "COUNTER1" must be set to "GET_TIME_OF_DAY".

- `-jdk=<directory>`

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

- `-dyninst=<dir>`

  Specifies the directory where the DynInst dynamic instrumentation package is installed. Using DynInst, a user can invoke tau_run to instrument an executable program at runtime or prior to execution

by rewriting it. DyninstAPI [http://www.dyninst.org/]PARA-DYN [http://www.paradyn.org/].

- `-vtf=<directory>`

  Specifies the location of the VTF3 trace generation package. TAU's binary traces can be converted to the VTF3 format using tau2vtf, a tool that links with the VTF3 library. The VTF3 format is read by Intel trace analyzer, formerly known as vampir, a commercial trace visualization tool developed by TU. Dresden, Germany.

- `-slog2=<directory>`

  Specifies the location of the SLOG2 SDK trace generation package. TAU's binary traces can be converted to the SLOG2 format using tau2slog2, a tool that uses the SLOG2 SDK. The SLOG2 format is read by the Jumpshot4 trace visualization software, a freely available trace visualizer from Argonne National Laboratories. [Ref: http://www-unix.mcs.anl.gov/perfvis/download/index.htm#slog2sdk]

- `-slog2`

  Specifies the use of the SLOG2 trace generation package and the Jumpshot trace visualizer that is bundled with TAU. Jumpshot v4 and SLOG2 v1.2.5delta are included in the TAU distribution. When the -slog2 flag is specified, tau2slog2 and jumpshot tools are copied to the <tau>/<arch>/<bin> directory. It is important to have a working javac and java (preferably v1.4+) in your path. On linux systems, where /usr/bin/java may be a place holder, you'll need to modify your path accordingly.

- `-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 [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.

- `-mpilibrary=<lib>`

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

- `-shmeminc=<dir>`

  Specifies the directory where shmem.h resides. Specifies the use of the TAU SHMEM interface.

- `-shmemlib=<dir>`

  Specifies the directory where libsma.a resides. Specifies the use of the TAU SHMEM interface.

- `-shmemlibrary=<lib>`

  By default, TAU uses -lsma as the shmem/pshmem library. This option allows the user to specify a different shmem library.

- `-nocomm`

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

- `-epilog=<dir>`

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

- `-MPITRACE`

  Specifies the tracing option and generates event traces for MPI calls and routines that are ancestors of MPI calls in the callstack. This option is useful for generating traces that are converted to the EPILOG trace format. KOJAK's Expert automatic diagnosis tool needs traces with events that call MPI routines. Do not use this option with the -TRACE option.

- `-pythoninc=<dir>`

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

- `-pythonlib=<dir>`

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

- `-PROFILE`

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

- `-PROFILECALLPATH`

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

- `-PROFILEPHASE`

  This option generates phase based profiles. It requires special instrumentation to mark phases in an application (I/O, computation, etc.). Phases can be static or dynamic (different phases for each loop iteration, for instance). See examples/phase/README for further information.

- `-PROFILESTATS`

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

- `-DEPTHLIMIT`

  Allows users to enable instrumentation at runtime based on the depth of a calling routine on a call-stack. The depth is specified using the environment variable TAU_DEPTH_LIMIT. When its value is 1, instrumentation in the top-level routine such as main (in C/C++) or program (in F90) is activated. When it is 2, only routine invoked directly by main and main are recorded. When a routine appears at a depth of 2 and at 10 and we set the limit at 5, then the routine is recorded when its depth is 2, and ignored when its depth is 10 on the calling stack. This can be used with -PROFILECALLPATH to generate a tree of height <h> from the main routine by setting TAU_CALLPATH_DEPTH and TAU_DEPTH_LIMIT variables to <h>.

- `-PROFILEMEMORY`

  Specifies tracking heap memory utilization for each instrumented function. When any function entry takes place, a sample of the heap memory used is taken. This data is stored as user-defined event data in profiles/traces.

- `-PROFILEHEADROOM`

  Specifies tracking memory available in the heap (as opposed to memory utilization tracking in -PROFILEMEMORY). When any function entry takes place, a sample of the memory available (headroom to grow) is taken. This data is stored as user-defined event data in profiles/traces. Please refer to the examples/headroom/README file for a full explanation of these headroom options and the C++/C/F90 API for evaluating the headroom.

- `-COMPENSATE`

  Specifies online compensation of performance perturbation. When this option is used, TAU computes its overhead and subtracts it from the profiles. It can be only used when profiling is chosen. This option works with MULTIPLECOUNTERS as well, but while it is relevant for removing perturbation with wallclock time, it cannot accurately account for perturbation with hardware performance counts (e.g., L1 Data cache misses). See TAU Publication [Europar04] for further information on this option.

- `-PROFILECOUNTERS`

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

- `-SGITIMERS`

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

- `-CRAYTIMERS`

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

- `-LINUXTIMERS`

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

- `-TRACE`

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

- `-muse`

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

- `-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.

1.1.2. tau_setup

tau_setup is a GUI interface to the configure and installtau tools.

1.1.3. installtau script

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

```
% installtau -help

TAU Configuration Utility
************************************************
Usage: installtau [OPTIONS]
  where [OPTIONS] are:
-arch=<arch>
```

```
-fortran=<compiler>
-cc=<compiler>
-c++=<compiler>
-useropt=<options>
-pdt=<pdtdir>
-papi=<papidir>
-mpiinc=<mpiincdir>
-mpilib=<mpilibdir>
-mpilibrary=<mpilibrary>
-opari=<oparidir>
************************************************************************
```

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

1.1.4. Examples:

(See Appendix for POOMA and Windows installation instructions)

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

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

- b) Installing TAU with Java

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

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

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

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

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

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

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

- f) Use multiple hardware performance counters

```
% configure -MULTIPLECOUNTERS -papi=/usr/local/papi \
  -PAPIWALLCLOCK -PAPIVIRTUAL -LINUXTIMERS \
  -mpiinc=/usr/local/mpich/include \
  -mpilib=/usr/local/mpich/ lib/ \
  -pdt=/usr/local/pdtoolkit -useropt=-O2
% setenv COUNTER1 LINUX_TIMERS
% setenv COUNTER2 PAPI_FP_INS
% setenv COUNTER3 PAPI_L1_DCM ...
```

- g) Use TAU with PDT and MPI on IBM BG/L

```
% cd pdtoolkit-3.x
% configure -XLC -exec-prefix=bgl; make clean install
% cd tau-2.x
% configure -mpi -arch=bgl -pdt=/usr/local/pdtoolkit-3.x -pdt_c++=xlC
```

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

1.2. Platforms Supported

TAU has been tested on the following platforms:

- 1. SGI

  On IRIX 6.x based systems, including Indy, Power Challenge, Onyx, Onyx2 and Origin 200, 2000, 3000 Series, CC 7.2+, KAI [http://www.kai.com] KCC and g++ [http://www.gnu.org] compilers are supported. On SGI Altix systems, Intel, and GNU compilers are supported.

- 2. LINUX Clusters

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

- 3. Sun Solaris

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

- 4. IBM

  On IBM SP2 and AIX systems, KAI KCC, KAP/Pro, IBM xlC, xlc, xlf90 and g++ compilers work with TAU. On IBM BG/L, IBM xlC, blrts_xlC, blrts_xlf90, blrts_xlc, and gnu compilers work with TAU. On IBM pSeries Linux, xlC, xlc, xlf90 and gnu compilers work with TAU.

- 5. HP HP-UX

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

- 6. HP Alpha Tru64

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

- 7. NEC SX series vector machines

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

- 8. Cray X1, T3E, SV-1, XT3, RedStorm

  On Cray T3E systems, KAI KCC and Cray CC compilers have been tested with TAU. On Cray SV-1 and X1 systems, Cray CC compilers have been tested with TAU. On Cray XT3, and RedStorm systems, PGI and GNU compilers have been tested with TAU.

- 9. Hitachi SR8000 vector machine

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

- 10. Apple OS X

  On Apple OS X machines, c++ or g++ may be used to compile TAU. Also, IBM's xlf90, xlf and Absoft Fortran 90 compilers for G4/G5 may be used with TAU.

- 11. Fujitsu PRIMEPOWER

  On Fujitsu Power machines, Sun and Fujitsu compilers may be used with TAU.

- 11. Microsoft Window

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

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

1.3. Software Requirements

- 1. Java v 1.4

  TAU's GUI paraprof requires Java v1.3 or better in your path. We recommend Java version 1.4x from Sun. An older Tcl/Tk based browser racy is also included with TAU for compatibility. It requires the executable wish to be in your path. racy is also available in this distribution but support for racy will be gradually phased out. Users are encouraged to use paraprof instead. Paraprof does *not* require - jdk=<dir> option to be specified (which is used for configuring TAU for analyzing Java applications). The java program should be in the user's path.

Chapter 2. Compiling

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

2.1. 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 -arc=sgin32
```

generates

```
Makefile.tau-trace-kcc libtau-trace-kcc.a
in tau-2.x/sgin32/lib
```

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

- `TAU_CXX` - for the C++ compiler
- `TAU_CC` - for the C compiler
- `TAU_F90` - for the F90 compiler
- `TAU_LINKER` - for the Linker
- `TAU_INCLUDE` - for the include directories
- `TAU_DEFS` - for the defines on the command-line
- `TAU_LIBS` - for the TAU static library
- `TAU_SHLIBS` - for the TAU shared object (dynamic library)

- TAU_MPI_INCLUDE - for the directory where MPI header files reside
- TAU_MPI_LIBS - for the TAU MPI library with the MPI libraries for C/C++
- TAU_MPI_FLIBS - for the TAU MPI library with MPI libraries for Fortran
- TAU_FORTRANLIBS - for additional fortran libraries for linking with C++
- TAU_CXXLIBS - for linking with C++ libraries when native f90 linker is used
- TAU_TRACE_INPUT_LIB - for linking with the TAU trace reader library to process binary TAU traces (typically used for making a trace converter)
- TAU_DISABLE - for the default TAU stub library for Fortran
- TAU_USER_OPT - for any user defined options specified during configuration

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

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

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

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

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

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

RM              = /bin/rm -f
TARGET          = matrix
################################################
```

```
all:            $(TARGET)
install:        $(TARGET)
$(TARGET):      $(TARGET).o
      $(CXX) $(LDFLAGS) $(TARGET).o -o $@ $(LIBS)
$(TARGET).o : $(TARGET).cpp
      $(CXX) $(CFLAGS) -c $(TARGET).cpp
clean:
        $(RM) $(TARGET).o $(TARGET)
###############################################
```

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

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

substitute

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

in the makefile above. Also,

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

points to the most recently configured version of the library.

2.2. 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.

2.3. Using TAU with MPI

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

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

   ```
   % ./configure -mpiinc=/usr/local/packages/mpich/include \
   ```

```
-mpilib=/usr/local/packages/mpich/lib/LINUX/ch_pp4 \
-c++=KCC -cc=cc
```

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

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

3. Use the Makefile variables

   `$(TAU_MPI_LIBS)`

   for C/C++ applications and

   `$(TAU_MPI_FLIBS)`

   for Fortran 90 applications, to specify the TAU MPI libraries before the

   `$(TAU_LIBS)`

   in the link command line. Also, use

   `$(TAU_MPI_INCLUDE)`

   in the compiler command line to specifies the MPI include directory to be used. Example:

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

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

2.4. Environment Variables

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

- `PROFILEDIR` - to name the directory that should contain the profile data files and
- `TRACEDIR` - the directory where event traces should be stored.
- `LD_LIBRARY_PATH` - (or LIBPATH for IBM) should include the <tauroot>/<tauarch>/lib directory if TAU is used with JAVA 2 (using the -jdk=<dir> configuration option) or dyninstAPI (using the -dyninst=<dir> configuration option).

Example:

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

Note: TAU also uses the environment variable `PCL_EVENT` and `PAPI_EVENT` to specify the hardware performance counter to be used when -pcl=<dir> or -papi=<dir> configuration options are used, respectively. See Section 4.6, “Using Hardware Performance Counters” for further details.

2.5. Application Scenarios

The TAU `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 API can be used for manually instrumenting a C++ program.

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

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

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

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

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

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

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

`papi` - A matrix multiply example that shows how to use TAU statement level timers for comparing the performance of two algorithms for matrix multiplication. When used with PAPI [http://icl.cs.utk.edu/papi/] or PCL [http://www.fz-juelich.de/zam/PCL/PCLcontent.html], 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).

`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).

`analyze` - Shows the use of tau_analyze, a utility that generates selective instrumentation lists for use with tau_instrumentor based on the analysis of collected program information and a user defined instrumentation scenario. The tau_analyze utility expands on the functionality of the tau_reduce utility. TAU must be configured with -pdt=<dir> option.

`reduce` - Shows the use of tau_reduce, a utility that can read profiles and a set of rules and determine which routines should not be instrumented (for frequently called light-weight routines). See `<tau>/utils/TAU_REDUCE.README` file for further details. It requires configuring TAU with -pdt=<dir> option.

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

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

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

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

`phase` - Shows the use of phase based profiling. Requires configuring TAU with the -PROFILEPHASE option. See the README file in the phase directory for details about the API and an example.

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

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

`NPB2.3` - The NAS Parallel Benchmark 2.3 [http://www.nas.nasa.gov/Software/NPB/] . 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 -mpi-inc=<dir> and -mpilib=<dir> to use this.

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

`dyninstthreads` - The above example with threads.

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

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

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

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

`openmpi` - Illustrates TAU's support for hybrid execution models in the form of MPI for message passing and OpenMP threads. TAU needs to be configured with -mpiinc=<dir> -mpilib=<dir> -openmp options to use this. fork Illustrates how to register a forked process with TAU. TAU provides two options: `TAU_INCLUDE_PARENT_DATA` - and `TAU_EXCLUDE_PARENT_DATA` - which allows the child process to inherit or clear the performance data when the fork takes place.

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

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

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

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

`memory` - TAU can sample memory utilization on some platforms using the getrusage() system call and interrupts. This directory illustrates how sampling can be used to track the maximum resident set size. See the README file in the memory directory for further information.

`malloc` - TAU's malloc and free wrappers can help pinpoint where the memory was allocated/deallocated in a C/C++ program. It can show the size of memory malloc'ed and free'd along with the source file name and line number.

`taucompiler` - using `$(TAU_COMPILER)` in your Makefile before the compiler name invokes tau_compiler.sh - a shell script that instruments and compiles the source file and links in the correct libraries. A Fortran 90 example illustrates its use in the f90 subdirectory.

`userevent` - TAU's user defined events can show context information highlighting the callpath that led to the event. This is supported using the `TAU_REGISTER_CONTEXT_EVENT` and `TAU_CONTEXT_EVENT` calls. It uses the `TAU_CALLPATH_DEPTH` env. variable. This feature works independently of the callpath or phase profiling options, which apply to bracketed entry and exit events - not atomic events. You can disable tracking the callpath at runtime.

`headroom` - TAU's memory headroom evalution options are discussed at length in the examples/headroom/README file. The amount of heap memory that can be allocated at any given point in

the program's execution are tracked in this directory (and three subdirectories - track, here, and available). `-PROFILEHEADROOM` configuration option may be used with these examples.

`mpitrace` - Kojak's Expert tool needs traces that record events that call MPI routines. We track this information at runtime when TAU is configured with the `-MPITRACE` option. This example illustrates its use.

Chapter 3. Tau Compiler

3.1. Introduction

The Tuning and Analysis Utilities (TAU) offers two methods for instrumenting C, C++, and Fortran code for profiling and tracing. The first is to instrument software by hand. While it gives the user complete control over what methods are instrumented, it has several disadvantages; the primary one being that process of inserting and removing code can be time consuming and error prone. The second method is to have TAU automatically instrument your source using the MPI wrapper library and the TAU Compiler.

If you are only interested in time spent in MPI functions, you only need to link your software to the TAU MPI wrapper library. See "Profiling MPI Software using TAU" for more information on this subject. However, most projects need a comprehensive picture of where time is spent. The TAU Compiler provides a simple way to automatically instrument an entire project. The TAU Compiler can be used on C, C++, fixed form Fortran, and free form Fortran.

3.2. Installing TAU Compiler

The TAU compiler comes standard with the TAU distribution, but requires that the Program Database Toolkit (PDT) be installed. TAU relies on the parsers provided by PDT to automatically insert TAU instrumentation into fuctions. Please see "Installing The Program Database Toolkit" for information on how to install PDT. The rest of this section will provide simple installation instructions for installing TAU to profile source code. Please see "Installing The Tuning and Analysis Utilities" for more information on the specific options available for a TAU installation.

Download and extract TAU from the TAU pages at The University of Oregon. In the extracted TAU directory, issue the command:

```
% ./configure -c++=<compiler> -cc=<compiler> \
  -fortran=<compiler> -pdt=<pdtdir> -mpi \
  -mpiinc=<dir> -mpilib=<dir> -PROFILE
```

This configures TAU to instrument MPI programs C++, C and Fortran programs, using PDT for automatic instrumentation. If you would like to see what other options are available, you can issue:

```
./configure -help
```

to get a complete listing of the configuration flags.

Configure will give you feedback on its progress. After the configuration is complete, you will want to add the directory indicated by TAU to your path; that is where the TAU Compiler will be located. Now you can enter:

```
% make install
```

to install TAU into the local directory.

3.3. Instrumenting with TAU Compiler

This section describes how the TAU Compiler can be used to instrument Fortran 90, C, and C++ projects. In general, the only necessary step is to replace the compiler used to build your projects with the TAU compiler command.

The combination of the TAU generated Makefile (found in <arch>/lib) and tau_compiler.sh (found in <atch>/bin) makes it particularly easy to instrument projects that use make to control the build. Open your makefile, insert

```
include <tau_dir>/<arch>/lib/Makefile.tau-mpi-pdt
```

in the top of it. This will include all of the header, library, and tool definitions that tau_compiler will need for your compiler, mpi installation, and pdt installation. Note that if you configured TAU with more options than listed in the installation section, the name of the makefile may be slightly different.

Now, find the line that states which compiler is used for compilation. For example, your Makefile might contain the line

```
CXX=g++
```

Replace that line with

```
CXX=$(TAU_COMPILER) g++
```

Perform this step for every Makefile in your project. Build your project as normal. If everything goes well, the TAU Compilere will parse your source files, instrument them with profiling code, save the instrumented files in temporary files, build and link the temporary files, the finally clean up the files. The build process will emit extra information, but the resulting object files and binaries will be completely instrumented. Now, when you run your program, it will write one or more profile files to your working directory. These profiles can be viewed using pprof or paraprof. For more information, see The Paraprof User's Guide.

3.4. Using tau_compiler.sh

If you want to instrument a single file, without using a makefile to handle all of the heavy lifting, you can use tau_compiler.sh by itself to instrument a file. Its syntax is:

```
% tau_compiler.sh <tau_compiler_options> <compiler> \
    <compiler_options>
```

The options available for tau_compiler.sh are:

- `-optVerbose`

  Turn on verbose debugging messages.

- `-optPdtDir=<dir>`

  The PDT architecture directory. Typically `$(PDTDIR)/$(PDTARCHDIR)`.

- `-optPdtF95Opts=<opts>`

Options for Fortran parser in PDT (f95parse).

- `-optPdtF95Reset=<opts>`

  Reset options to the Fortran parser to the given list.

- `-optPdtCOpts=<opts>`

  Options for C parser in PDT (cparse). Typically `$(TAU_MPI_INCLUDE) $(TAU_INCLUDE) $(TAU_DEFS)`.

- `-optPdtCReset=<opts>`

  Reset options to the C parser to the given list

- `-optPdtCxxOpts=<opts>`

  Options for C++ parser in PDT (cxxparse). Typically `$(TAU_MPI_INCLUDE) $(TAU_INCLUDE) $(TAU_DEFS)`.

- `-optPdtCReset=<opts>`

  Reset options to the C++ parser to the given list

- `-optPdtF90Parser=<parser>`

  Specify a different Fortran parser. For e.g., `f90parse` instead of `f95parse`.

- `-optGnuFortranParser=<parser>`

  Specify the GNU `gfortran` Fortran parser `gfparse` instead of `f95parse`

- `-optPdtUser=<opts>`

  Optional arguments for parsing source code.

- `-optTauInstr=<path>`

  Specify location of tau_instrumentor. Typically `$(TAUROOT)/$(CONFIG_ARCH)/bin/tau_instrumentor`.

- `-optPreProcess`

  Preprocess the source code before parsing. Uses /usr/bin/cpp -P by default.

- `-optCPP=<path>`

  Specify an alternative preprocessor and pre-process the sources.

- `-optCPPOpts=<options>`

  Specify additional options to the C pre-processor.

- `-optCPPReset=<options>`

  Reset C preprocessor options to the specified list.

- `-optTauSelectFile=<file>`

Specify selective instrumentation file for tau_instrumentor

- `-optPDBFile=<file>`

  Specify PDB file for tau_instrumentor. Skips parsing stage.

- `-optTau=<opts>`

  Specify options for tau_instrumentor.

- `-optCompile=<opts>`

  Options passed to the compiler. Typically `$(TAU_MPI_INCLUDE) $(TAU_INCLUDE) $(TAU_DEFS)` .

- `-optTauDefs=<opts>`

  Options passed to the compiler by TAU. Typically `$(TAU_DEFS)` .

- `-optTauIncludes=<opts>`

  Options passed to the compiler by TAU. Typically `$(TAU_MPI_INCLUDE) $(TAU_INCLUDE)` .

- `-optReset=<opts>`

  Reset options to the compiler to the given list

- `-optLinking=<opts>`

  Options passed to the linker. Typically `$(TAU_MPI_FLIBS) $(TAU_LIBS) $(TAU_CXXLIBS)` .

- `-optLinkReset=<opts>`

  Reset options to the linker to the given list.

- `-optTauCC=<cc>`

  Specifies the C compiler used by TAU.

- `-optOpariTool=<path/opari>`

  Specifies the location of the Opari tool.

- `-optOpariDir=<path>`

  Specifies the location of the Opari directory.

- `-optOpariOpts=<opts>`

  Specifies optional arguments to the Opari tool.

- `-optOpariReset=<opts>`

  Resets options passed to the Opari tool.

- `-optNoMpi`

Removes `-l*mpi*` libraries during linking (default).

- `-optMpi`

  Does not remove `-l*mpi*` libraries during linking.

- `-optNoRevert`

  Exit on error. Does not revert to the original compilation rule on error.

- `-optRevert`

  Revert to the original compilation rule on error (default).

- `-optKeepFiles`

  Does not remove intermediate `.pdb` and `.inst.*` files.

3.5. TAU scripted compilation

If you wish to avoid the modification of your Makefiles, or are not using Makefiles at all, TAU provides three script that will instrument your code from the command line.

3.5.1. Usage

TAU provides these scripts: tau_f90.sh, tau_cc.sh, and tau_cxx.sh to compile Fortan, C, and C++ programs respectively. These can the found in the tools/src/ directory so you may wish to add it to your path. You might use tau_cc.sh to compile a C program by typing:

```
%>tau_cc.sh -tau_makefile=[path to makefile] \
            -tau_options=[optionx] sampleCprogram.c
```

The Makefile can usually be found in the `/[arch]/lib` directory, for example `/apple/lib/Makefile.tau-pdt`.

tau_cc.sh also has the ability to use a Makefile specified in an environment variable. to run tau_cc.sh so it uses the Makefile specified by environment variable `TAU_MAKEFILE`, type:

```
%>export TAU_MAKEFILE=[path to tau]/[arch]/lib/[makefile].
%>tau_cc.sh sampleCprogram.c
```

Similarly, if you want to set `TAU_COMPILER` options like selective instrumentation use the `TAU_OPTIONS` environment variable.

Chapter 4. Profiling

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

4.1. Running the application

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

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

This profiles all routines

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

The above profiles routines belonging to `TAU_IO, TAU_FIELD and TAU_USER2` profile groups. For a detailed list of groups, please refer to `TAU [http://www.cs.uoregon.edu]`

4.2. Profiling each call to a function

By default TAU profiles the total time (inclusive/exclusive) spend on a given function. Profiling each function call, for application that call some funcition hundred of thousands of times, is impractical since the profile data would grow enormously. But configuring TAU with the `-PROFILEPARAM` option will have TAU profile select function each time they are called. But TAU will also group some of these fuction calls together according to the value of the parameter they are given. For example if a function mpisend(int i) is called 2000 times 1000 times with 512 and 1000 times with 1024 then we will recieve two profile for mpisend() one we it is called with 512 and one when it is called with 1024. This reduces the overhead since we are profiling mpisend() two times not 2000 times.

4.3. Selectively Profiling an Application

TAU allow the users to select which functions to profile within a single application. One way the user can selectively instrument an application is by specifing rules which govern which functions should be profiled.

TAU's environment variable TAU_THROTTLE may be turned on to enable selective instrumentation basied on such rules. TAU uses a default rule of numcalls > 100000 && usecs/call < 10 which means that if a function executes greater than 100000 times and has an inclusive time per call of less than 10 microseconds, that profiling of that function will be disabled after that threshold is reached. To change the values of numcalls and usecs/call the user may optionally set environment variables:

```
% setenv TAU_THROTTLE 1
% setenv TAU_THROTTLE_NUMCALLS 2000000
% setenv TAU_THROTTLE_PERCALL  5
```

to change the values to 2 million and 5 microseconds per call. Throttling is disabled by default and will not take effect unless the TAU_THROTTLE environment variable is set to some value.

For more control over selective instrumentation use the tool "tau_reduce". See tau_reduce

You can also manually control the instrumentation of a program by using the -f <instrumentation file> option. It can be used to specify an exclude/include list of routines and/or files for instrumentation. The list of routines to be excluded from instrumentation is specified, one per line, enclosed by `BEGIN_EXCLUDE_LIST` and `END_EXCLUDE_LIST`. Instead of specifying which routines should be excluded, the user can specify the list of routines that are to be instrumented using the include list, one routine name per line, enclosed by `BEGIN_INCLUDE_LIST` and `END_INCLUDE_LIST`. Similarly, files can be included or excluded with the `BEGIN_FILE_EXCLUDE_LIST`, `END_FILE_EXCLUDE_LIST`, `BEGIN_FILE_INCLUDE_LIST`, and `END_FILE_INCLUDE_LIST` lines. '#' (when the first character of the line) begins a comment. However '#' used within a line specifies a wild-card. For example:

```
#Tell tau to not profile these functions
BEGIN_EXCLUDE_LIST

void quicksort(int *, int, int)
# The next line excludes all functions begining with "sort_" and having arguments
# "int *"
void sort_#(int *)
void interchange(int *, int *)

END_EXCLUDE_LIST

#Exclude these files from profiling
BEGIN_FILE_EXCLUDE_LIST

*.so

END_FILE_EXCLUDE_LIST
```

Manually editing the selective instrumentation file gives you more options. The `BEGIN_INSTRUMENT_SECTION` and `END_INSTRUMENT_SECTION` tags allow more control the instrumenation. Loops in the source can be profiled by specifing a routine in which all loop should be profiled. If this line is in between the SECTION tags each loop inside the multiply routine will be profiled:

```
loops file="loop_test.cpp" routine="multiply"
```

Wildcard can be used to instrument multiple loops. For file names * character can be used to specify any number of character, thus foo* matches foobar, foo2, etc. also for file names ? can match a single charater, ie. foo? matches foo2, fooZ, but not foobar. You can use # as a wildcard for routines, ie. b# matches bar, b2z etc.

Within these tags you can also insert code fragment within the source code by specifing the file and line number, for example:

```
file = "line_test.cpp" line = 9 code = "printf(\"i=%d: \", i);"
```

You may want to add code at the entry and exit of a particular routine, for example:

```
exit routine ="int foo()" code = "cout <<\"exiting foo\"<<endl;"
entry routine ="int foo()" code = "cout <<\"entering foo\"<<endl;"
```

4.4. Running an application using DynInstAPI

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

The commandline options accepted by tau_run are:

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

By default, `libTAU`. so is loaded by tau_run. However, the user can override this and specify another file using the -Xrun<Taulibrary>. In this case lib<Taulibrary>.so will be loaded using `LD_LIBRARY_PATH`.

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

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

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

4.5. Dynamically Instrumenting MPI applications

The tau_laod.sh script allows you to instrument an mpi application at run time using the LD_PRELOAD mechanism. This feature allow instrumentation of already compiled executables without TAU's having to edit the application's code. This is only available on platforms that support LD_PRELOAD. Furthermore since instrumentation is done at runtime the linking is all done dynamically--applications that have static libraries will fail to load them.

To use tau_laod.sh simply place it before the application's executable when calling mpirun:

```
%> mpirun -np 4 tau_load.sh ./a.out
```

For those on AIX systems use the tau_poe application:

```
%> tau_poe ./a.out -procs 4
```

4.6. 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 [http://icl.cs.utk.edu/papi/]) and Performance Counter Library (PCL, PCL [http://www.fz-juelich.de/zam/PCL/]) packages provide a uniform interface to access these performance counters. TAU can use either PAPI or PCL to access these hardware performance counters. To do so, download and install PAPI or PCL. Then, configure TAU using the -pcl=<dir> or -papi=<dir> configuration command-line option to specify the location of PCL or PAPI. Build TAU and applications as you normally would (as described in Chapters 2 and 3). While running the application, set the environment variable PCL_EVENT or PAPI_EVENT respectively, to specify which hardware performance counter TAU should use while profiling the application.

Note

By default, only one counter is tracked at a time. To track more than one counter use -MULTIPLECOUNTERS. See Section 4.7, “Using Multiple Hardware Counters for Measurement” for more details.

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

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

In addition to the following events, you can use native events (see **papi_native**) on a given CPU by setting PAPI_EVENT to PAPI_NATIVE_<event>. For example:

```
% setenv PAPI_EVENT PAPI_NATIVE_PM_BIQ_IDU_FULL_CYC
% a.out
```

Table 4.1. Events measured by setting the environment variable PAPI_EVENT in TAU

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

PAPI_EVENT	EVENT Measured
PAPI_L2_TCM	Level 2 total cache misses
PAPI_L3_TCM	Level 3 total cache misses
PAPI_CA_SNP	Snoops
PAPI_CA_SHR	Request for access to shared cache line (SMP)
PAPI_CA_CLN	Request for access to clean cache line (SMP)
PAPI_CA_INV	Cache Line Invalidation (SMP)
PAPI_CA_ITV	Cache Line Intervention (SMP)
PAPI_L3_LDM	Level 3 load misses
PAPI_L3_STM	Level 3 store misses
PAPI_BRU_IDL	Cycles branch units are idle
PAPI_FXU_IDL	Cycles integer units are idle
PAPI_FPU_IDL	Cycles floating point units are idle
PAPI_LSU_IDL	Cycles load/store units are idle
PAPI_TLB_DM	Data translation lookaside buffer misses
PAPI_TLB_IM	Instruction translation lookaside buffer misses
PAPI_TLB_TL	Total translation lookaside buffer misses
PAPI_L1_LDM	Level 1 load misses
PAPI_L1_STM	Level 1 store misses
PAPI_L2_LDM	Level 2 load misses
PAPI_L2_STM	Level 2 store misses
PAPI_BTAC_M	BTAC miss
PAPI_PRF_DM	Prefetch data instruction caused a miss
PAPI_L3_DCH	Level 3 Data Cache Hit
PAPI_TLB_SD	Translation lookaside buffer shootdowns (SMP)
PAPI_CSR_FAL	Failed store conditional instructions
PAPI_CSR_SUC	Successful store conditional instructions
PAPI_CSR_TOT	Total store conditional instructions
PAPI_MEM_SCY	Cycles Stalled Waiting for Memory Access
PAPI_MEM_RCY	Cycles Stalled Waiting for Memory Read
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_EVENT	**EVENT Measured**
PAPI_FMA_INS	FMA instructions completed
PAPI_TOT_IIS	Total instructions issued
PAPI_TOT_INS	Total instructions executed
PAPI_INT_INS	Integer instructions executed
PAPI_FP_INS	Floating point instructions executed
PAPI_LD_INS	Load instructions executed
PAPI_SR_INS	Store instructions executed
PAPI_BR_INS	Total branch instructions executed
PAPI_VEC_INS	Vector/SIMD instructions executed
PAPI_FLOPS	Floating Point Instructions executed per second
PAPI_RES_STL	Cycles processor is stalled on resource
PAPI_FP_STAL	FP units are stalled
PAPI_TOT_CYC	Total cycles
PAPI_IPS	Instructions executed per second
PAPI_LST_INS	Total load/store instructions executed
PAPI_SYC_INS	Synchronization instructions executed
PAPI_L1_DCH	L1 D Cache Hit
PAPI_L2_DCH	L2 D Cache Hit
PAPI_L1_DCA	L1 D Cache Access
PAPI_L2_DCA	L2 D Cache Access
PAPI_L3_DCA	L3 D Cache Access
PAPI_L1_DCR	L1 D Cache Read
PAPI_L2_DCR	L2 D Cache Read
PAPI_L3_DCR	L3 D Cache Read
PAPI_L1_DCW	L1 D Cache Write
PAPI_L2_DCW	L2 D Cache Write
PAPI_L3_DCW	L3 D Cache Write
PAPI_L1_ICH	L1 instruction cache hits
PAPI_L2_ICH	L2 instruction cache hits
PAPI_L3_ICH	L3 instruction cache hits
PAPI_L1_ICA	L1 instruction cache accesses
PAPI_L2_ICA	L2 instruction cache accesses
PAPI_L3_ICA	L3 instruction cache accesses
PAPI_L1_ICR	L1 instruction cache reads
PAPI_L2_ICR	L2 instruction cache reads
PAPI_L3_ICR	L3 instruction cache reads
PAPI_L1_ICW	L1 instruction cache writes
PAPI_L2_ICW	L2 instruction cache writes
PAPI_L3_ICW	L3 instruction cache writes
PAPI_L1_TCH	L1 total cache hits
PAPI_L2_TCH	L2 total cache hits

PAPI_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

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 4.2. Events measured by setting the environment variable PCL_EVENT in TAU

PCL_EVENT	EVENT Measured
PCL_L1CACHE_READ	L1 (Level one) cache reads
PCL_L1CACHE_WRITE	L1 cache writes
PCL_L1CACHE_READWRITE	L1 cache reads and writes
PCL_L1CACHE_HIT	L1 cache hits
PCL_L1CACHE_MISS	L1 cache misses
PCL_L1DCACHE_READ	L1 data cache reads
PCL_L1DCACHE_WRITE	L1 data cache writes
PCL_L1DCACHE_READWRITE	L1 data cache reads and writes
PCL_L1DCACHE_HIT	L1 data cache hits
PCL_L1DCACHE_MISS	L1 data cache misses
PCL_L1ICACHE_READ	L1 instruction cache reads
PCL_L1ICACHE_WRITE	L1 instruction cache writes
PCL_L1ICACHE_READWRITE	L1 instruction cache reads and writes
PCL_L1ICACHE_HIT	L1 instruction cache hits
PCL_L1ICACHE_MISS	L1 instruction cache misses
PCL_L2CACHE_READ	L2 (Level two) cache reads

PCL_EVENT	EVENT Measured
PCL_L2CACHE_WRITE	L2 cache writes
PCL_L2CACHE_READWRITE	L2 cache reads and writes
PCL_L2CACHE_HIT	L2 cache hits
PCL_L2CACHE_MISS	L2 cache misses
PCL_L2DCACHE_READ	L2 data cache reads
PCL_L2DCACHE_WRITE	L2 data cache writes
PCL_L2DCACHE_READWRITE	L2 data cache reads and writes
PCL_L2DCACHE_HIT	L2 data cache hits
PCL_L2DCACHE_MISS	L2 data cache misses
PCL_L2ICACHE_READ	L2 instruction cache reads
PCL_L2ICACHE_WRITE	L2 instruction cache writes
PCL_L2ICACHE_READWRITE	L2 instruction cache reads and writes
PCL_L2ICACHE_HIT	L2 instruction cache hits
PCL_L2ICACHE_MISS	L2 instruction cache misses
PCL_TLB_HIT	TLB (Translation Lookaside Buffer) hits
PCL_TLB_MISS	TLB misses
PCL_ITLB_HIT	Instruction TLB hits
PCL_ITLB_MISS	Instruction TLB misses
PCL_DTLB_HIT	Data TLB hits
PCL_DTLB_MISS	Data TLB misses
PCL_CYCLES	Cycles
PCL_ELAPSED_CYCLES	Cycles elapsed
PCL_INTEGER_INSTR	Integer instructions executed
PCL_FP_INSTR	Floating point (FP) instructions executed
PCL_LOAD_INSTR	Load instructions executed
PCL_STORE_INSTR	Store instructions executed
PCL_LOADSTORE_INSTR	Loads and stores executed
PCL_INSTR	Instructions executed
PCL_JUMP_SUCCESS	Successful jumps executed
PCL_JUMP_UNSUCCESS	Unsuccessful jumps executed
PCL_JUMP	Jumps executed
PCL_ATOMIC_SUCCESS	Successful atomic instructions executed
PCL_ATOMIC_UNSUCCESS	Unsuccessful atomic instructions executed
PCL_ATOMIC	Atomic instructions executed
PCL_STALL_INTEGER	Integer stalls
PCL_STALL_FP	Floating point stalls
PCL_STALL_JUMP	Jump stalls
PCL_STALL_LOAD	Load stalls
PCL_STALL_STORE	Store Stalls
PCL_STALL	Stalls
PCL_MFLOPS	Millions of floating point operations/second

PCL_EVENT	EVENT Measured
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

4.7. Using Multiple Hardware Counters for Measurement

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

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

LIST OF COUNTERS:

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

- `GET_TIME_OF_DAY` --- For the default profiling option using gettimeofday()
- `SGI_TIMERS` --- For `-SGITIMERS` configuration option under IRIX
- `CRAY_TIMERS` --- For `-CRAYTIMERS` configuration option under Cray X1.
- `LINUX_TIMERS` --- For -LINUXTIMERS configuration option under Linux
- `CPU_TIME` --- For user+system time from getrusage() call with `-CPUTIME`
- `P_WALL_CLOCK_TIME` --- For PAPI's WALLCLOCK time using `-PAPIWALLCLOCK`
- `P_VIRTUAL_TIME` --- For PAPI's process virtual time using `-PAPIVIRTUAL`
- `TAU_MUSE` --- For reading counts of Linux OS kernel level events when MAGNET/MUSE is installed and -muse configuration option is enabled. MUSE [http://public.lanl.gov/radiant/].`TAU_MUSE_PACKAGE` environment variable has to be set to package name (busy_time, count, etc.)
- `TAU_MPI_MESSAGE_SIZE` --- For tracking the cumulative message size for all MPI operations by a node for each routine.

Note

When TAU is configured with -TRACE -MULTIPLECOUNTERS and -papi=<dir> options, the COUNTER1 environment variable must be set to GET_TIME_OF_DAY to allow TAU's tracing module to use a globally synchronized real-time clock for timestamping event records. When we use tracing with hardware performance counters, the counters specified in environment variables COUNTER[2-25] are accessed at routine transitions and logged in the trace file. Use tau2vtf tool to convert TAU traces to VTF3 traces that may be

loaded in the Vampir trace visualization tool.

and PAPI/PCL options that can be found in Table 4.1, “Events measured by setting the environment variable PAPI_EVENT in TAU” and Table 4.2, “Events measured by setting the environment variable PCL_EVENT in TAU”. Example:

- `PCL_FP_INSTR` --- For floating point operations using PCL (-pcl=<dir>)
- `PAPI_FP_INS` --- For floating point operations using PAPI (-papi=<dir>)
- `PAPI_NATIVE_<event>` --- For native papi events using PAPI (-papi=<dir>)

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

Example:

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

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

4.8. Running a Python application with TAU

TAU can automatically instrument all Python routines when the tau python package is imported. Add <TAUROOT>/<ARCH>/lib/bindings-<options> to the PYTHONPATH environment variable in order to use the TAU module.

To execute the program, tau.run routine is invoked with the name of the top level Python code. For e.g.,

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

import tau
from time import sleep

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

def OurMain():
    f1()

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

instruments routines `OurMain(), f1() and f2()` although there are no instrumentation calls in the routines. To use this feature, TAU must be configured with the -pythoninc=<dir> option (and -

pythonlib=<dir> if running under IBM). Before running the application, the environment variable PYTHONPATH should be set to include the TAU library directory (where tau.py is stored). Manual instrumentation of Python sources is also possible using the Python API and the pytau package. For e.g.,

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

import pytau
from time import sleep

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

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

4.9. 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 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
 node numbers : prints information about all contexts/threads
 for specified nodes
```

4.10. Running a JAVA application with TAU

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

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

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

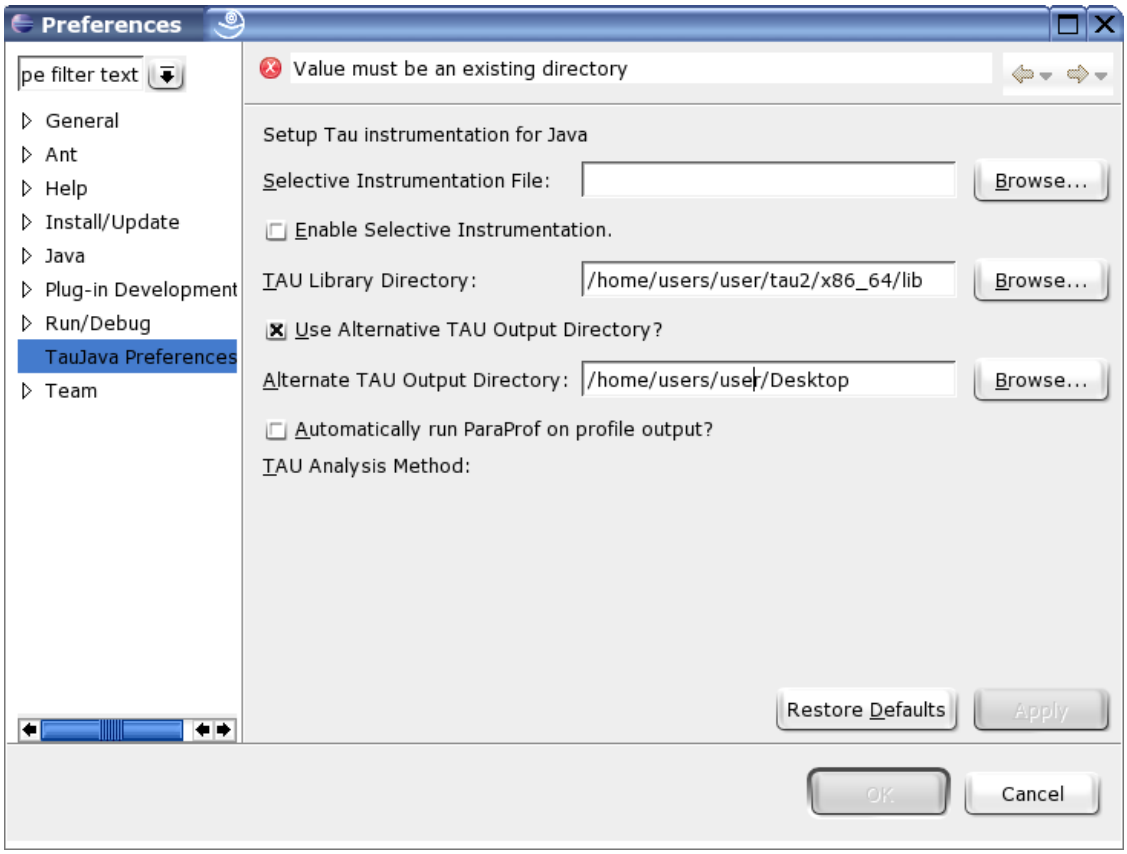
Chapter 5. Eclipse Tau Java System

5.1. Installation

Copy the plugins directory in the tau2/tools/src/taujava directory to the location of your eclipse installation. You may have to restart eclipse if it is running when this is done.

In eclipse go to the Window menu, select Preferences and go to the TauJava Preferences section. Enter the location of the lib directory in the tau installation for your architecture in the Tau Library Directory field. Other options may also be selected at this time.

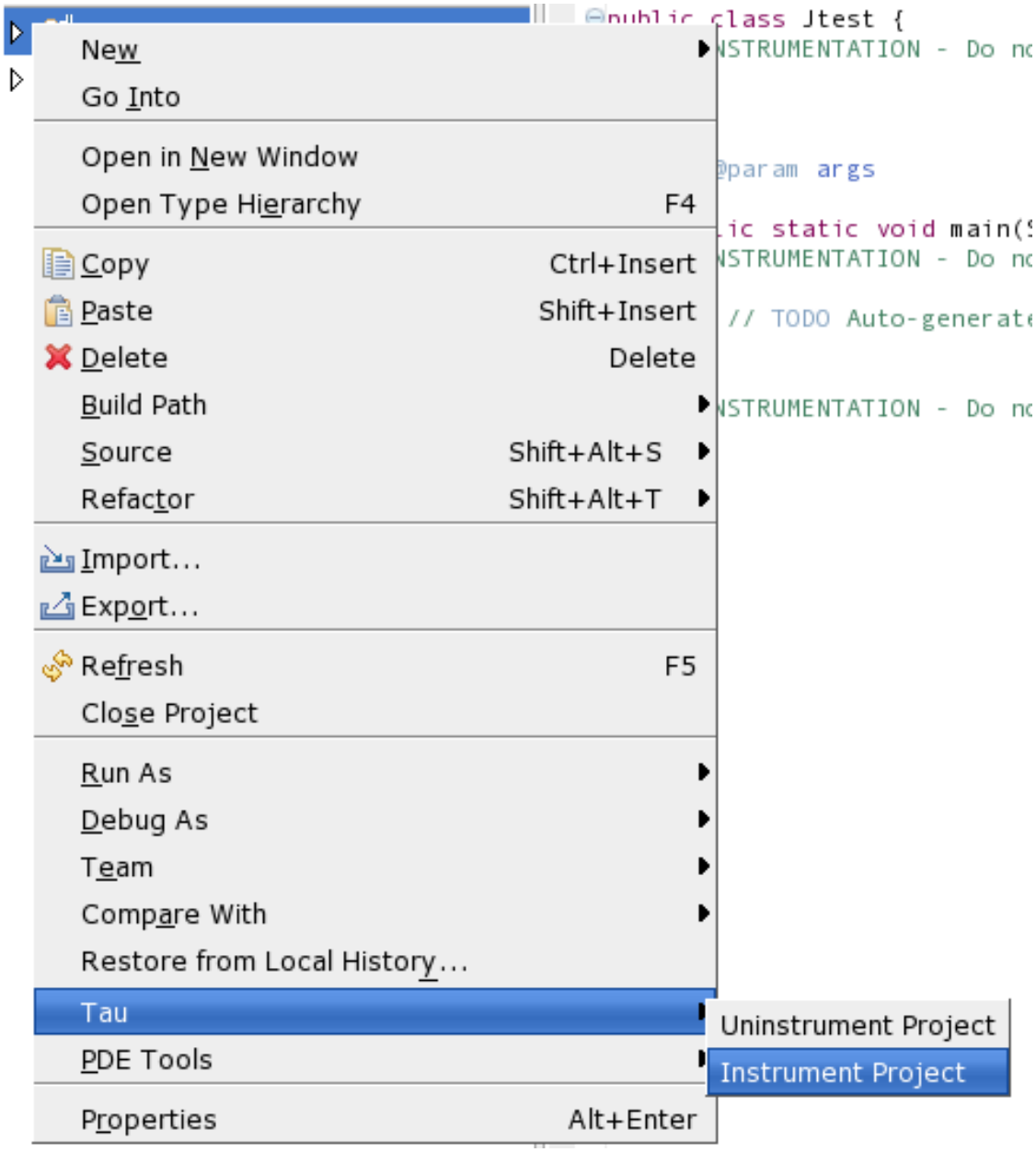
Figure 5.1. TAUJava Options Screen



5.2. Instrumentation

Java programs can be instrumented at the level of full Java projects, packages or individual Java files. From within the Java view simply right click on the element in the package explorer that you wish to instrument select the Tau pop up menu and click on Instrument Project, Package or Java respectively.

Figure 5.2. TAUJava Project Instrumentation



Note that the instrumenter will add the TAU.jar file to the project's class-path the first time any element is instrumented.

Do not perform multiple instrumentations of the same Java file. Do not edit the comments added by the instrumenter or adjust the white space around them. Doing so may prevent the uninstrumenter from working properly.

5.3. Uninstrumentation

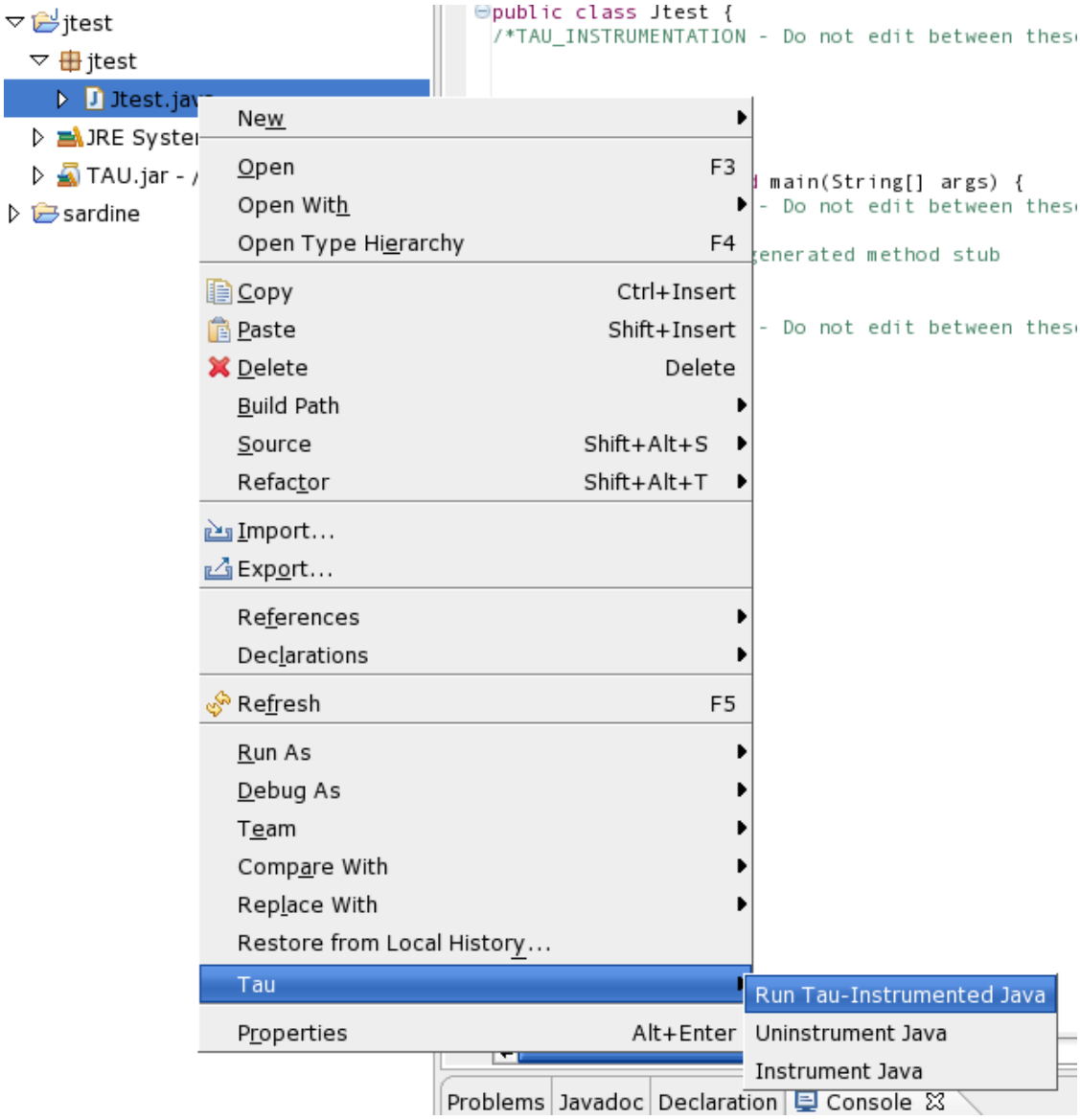
Uninstrumenting a Java project, package or file works just like instrumenting. Just select the uninstrument option instead. Note that the uninstrumenter only removes TAU instrumentation as formatted and commented by the instrumenter. Running the uninstrumenter on code with no TAU instrumentation present has no effect.

5.4. Running Java with TAU

To automatically analyze your instrumented project on a Unix-based system TAU must first be configured with the -JDK option, and any other options you want applied to your trace output. On windows the type of analysis to be conducted, Profile, Call path or Trace, should be selected from the Window, Preferences TauJava Preferences menu.

Once that has been accomplished, right click on the Java file containing the main method you want to run, go to the TAU menu and click on Run Tau-Instrumented Java. The program will run and, by default, the profile and/or trace files will be placed in a timestamped directory, inside a directory indicating the name of the file that was run, in the TAU_Output directory in the home directory of the Java project.

Figure 5.3. TAUJava Running



5.5. Options

The following options are accessible from the Window, Preferences TAUJava Preferences menu.

Use Alternative TAU Output Directory: Causes the TAU_Output directory to be placed in the location specified in the associated field. The internal directory structure of the TAU_Output directory remains unchanged.

Automatically run ParaProf on profile output?: Causes the TAU profile viewer, paraprof, to run on the output of profile and call-path analysis output as soon as the trace files have been produced.

Enable selective instrumentation: Causes Java elements specified in the given selection file to be included or excluded from instrumentation. By default all packages files and methods are included. The file should conform to the TAU file selection format described here.

```
# Any line beginning with a # is a comment and will be disregarded.
#
# If an entry is both included and excluded inclusion will take precedence.
#
# Entries in INCLUDE or EXCLUDE lists may use * as a wildcard character.
#
# If an EXCLUDE_LIST is specified, the methods in the list will not be
# instrumented.
#
BEGIN_EXCLUDE_LIST
*main*
END_EXCLUDE_LIST
#
# If an INCLUDE_LIST is specified, only the methods in the list will be
# instrumented.
#
BEGIN_INCLUDE_LIST
*get*
*set*
END_INCLUDE_LIST
#
# TAU also accepts FILE_INCLUDE/EXCLUDE lists.  These may be specified with
# the wildcard character # to exclude/include multiple files.
# These options may be used in conjunction with the routine INCLUDE/EXCLUDE
# lists as shown above.
#
BEGIN_FILE_INCLUDE_LIST
foo.java
hello#.java
END_FILE_INCLUDE_LIST
#
BEGIN_FILE_EXCLUDE_LIST
bar.java
END_FILE_EXCLUDE_LIST
# Note that the order of the individual sections does not matter
# and not all of the sections need to be included.  Each section
# must be closed.
```

Chapter 6. Eclipse PTP / CDT plugin System

6.1. Installation

Be certain that the PTP/CDT/FDT plugin is installed and running properly in your eclipse installation.

Copy the plugins folder in the tauptp directory to the location of your eclipse installation. You may have to restart eclipse if it is running when this is done.

In eclipse go to the Window menu, select Preferences and go to the TAU Preferences section. Enter the location of the desired architecture directory in your tau installation in the Tau Arch Directory field. Select a makefile either by choosing one from the dropdown menu or by selecting 'Specify Makefile Manually' from the menu and entering a TAU makefile in the field below. Normally to generate auto-instrumented code the tau makefile selected should have been generated with the '-pdt' flag. Other options may also be selected at this time.

6.2. Adding a Tau configuration

To add a TAU Build configuration to a Managed Make C, C++ or FDT Make project right click on the project and select Add Tau Configuration. You will be presented with a list of existing configurations. The configuration you select will be copied and modified to automatically apply TAU instrumentation to the resulting program. A parenthetical statement indicating the TAU makefile type used with the new TAU build will be appended to the name of the configuration. TAU Configurations generated in this way may be manipulated as standard Eclipse build configurations.

6.3. Running A TAU Instrumented Binary

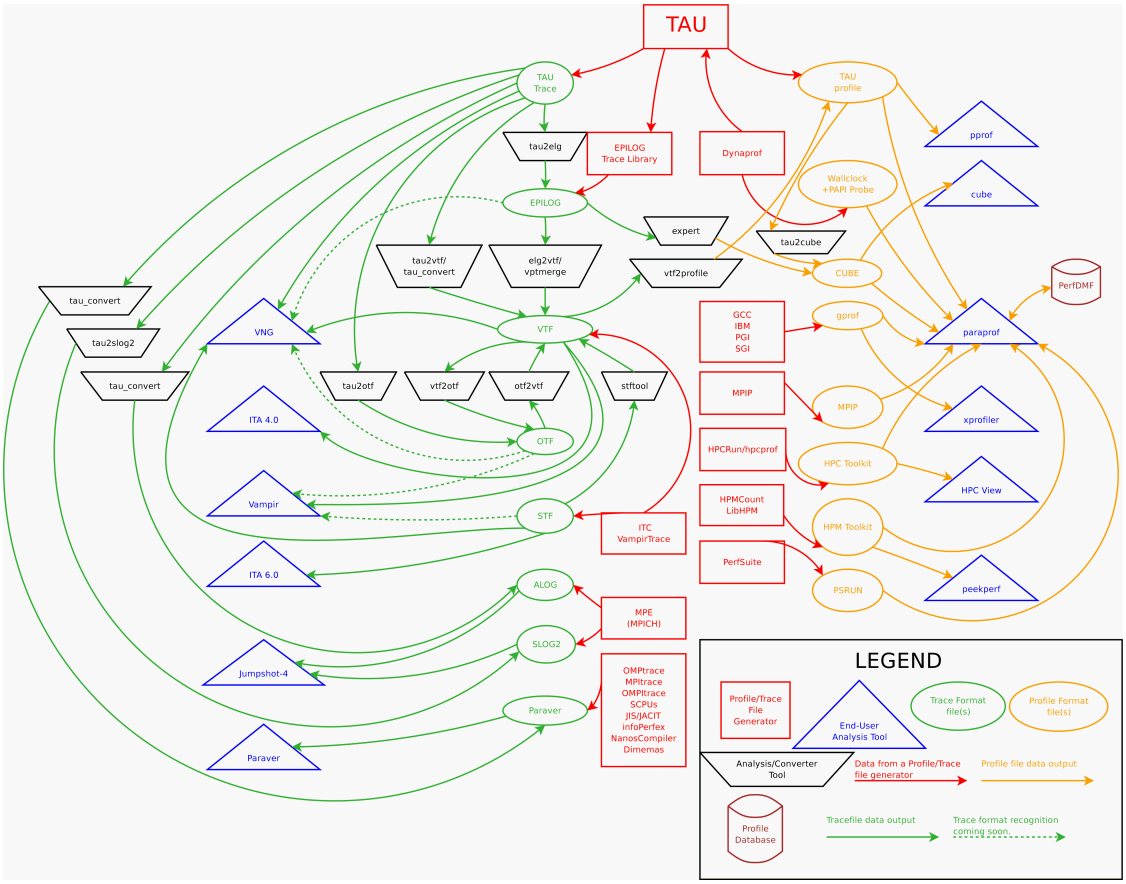
Once a program has been compiled with TAU instrumentation you may run it by right clicking on the resulting binary and selecting TAU Run. The program will run and, by default, the profile and/or trace files will be placed in a timestamped directory, inside a directory indicating the name of the file that was run, in the TAU_Output directory in the home directory of the Java project. Note that you may also run a TAU instrumented binary using Eclipse's standard run system, but any trace or profile output will be placed directly in the project's home directory. In either case the resulting run configuration may be manipulated as standard Eclipse run configurations.

Chapter 7. Tracing

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

Figure 1 show the possible interactions between different trace file formats.

Figure 7.1. Performance Data IO Chart



7.1. 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. The environment variables `TAU_TRACEFILE` may be used to specify the name of Vampir trace file. When this variable is set, trace files are automatically merged and the tau2vtf is invoked to convert the merged trace file to VTF3 trace format. This conversion takes place on node 0, thread 0. The intermediate trace files are deleted. To retain the trace files, the user can set the environment variable `TAU_KEEP_TRACEFILES` to true. When `TAU_TRACEFILE` is not specified, the user needs to merge and convert the traces as below. Ex-

ample:

```
% ./configure -arch=sgi64 -TRACE  -mpi -vtf=/usr/local/vtf3-1.34 -slog2
% 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] [-n] [-e eventedf*]
                 [-m mergededf] 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.

The utility tau_treemerge.pl may be used to generate the trace files in a hierarchical manner. It generates tau.trc and tau.edf files.

```
% tau_treemerge.pl
% tau_treemerge.pl -help
tau_treemerge.pl [-n <break amount> ]
```

tau_treemerge.pl can take an optional argument (with -n <value>) to specify the maximum number of trace files to merge in each invocation of tau_merge. If we need to merge 2000 trace files and if the maximum number of open files specified by unix is 250, tau_treemerge.pl will incrementally merge the trace files so as not to exceed the number of open file descriptors. This is important for the IBM Blue-Gene/L machine where such restrictions are present on the front-end node.

To convert merged or per-thread traces to another trace format, the utilities tau_convert, tau2vtf, or tau2slog2 are used as shown below:

```
% tau2vtf
Usage: tau2vtf <TAU trace> <edf file> <out file> [-a|-fa]
               [-nomessage] [-v]
 -a         : ASCII VTF3 file format
 -fa        : FAST ASCII VTF3 file format
 -nomessage : Suppress printing of message information in the trace
 -v         : Verbose
 Default trace format of <out file> is VTF3 binary
 e.g.,
 tau2vtf merged.trc tau.edf app.vpt.gz
% tau2vtf matrix.trc tau.edf matrix.vpt.gz
% vampir matrix.vpt.gz
```

To generate slog2 trace files that may be visualized using Jumpshot, we recommend using the slog2 SDK and Jumpshot bundled with TAU.

```
% configure -slog2 -TRACE ...
% tau2slog2
tau2slog2 converts a TAU formatted trace file to the SLOG2 format
        for Jumpshot trace visualizer
Usage: tau2slog2 <tau_tracefile> <edf_file> -o <slog_tracefile>
For e.g.,
% tau2slog2 app.trc tau.edf -o app.slog2
```

To generate traces that may be visualized using Vampir, we recommend using tau2vtf over the older tau_convert tool. tau2vtf can produce binary traces with user-defined events (hardware performance counters from PAPI etc.) while tau_convert cannot do this. Binary traces load faster in Vampir.

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

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

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

tau_convert can also be used to convert traces to the Vampir [http://www.vampir-ng.de/] trace format. 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.vpt.gz &
```

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

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

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

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

Vampir format traces may be converted to TAU profiles using the vtf2profile tool.

```
% vtf2profile -f matrix.vpt.gz -p profiledatadir
% vtf2profile
Usage: vtf2profile [options]
*************************HELP*************************
* '-h' --display this help text.                     *
* '-c' --open command line interface.                *
* '-f' --used as -f <VTF File> where                  *
*        VTF File is the name of the trace file       *
*        to be converted to TAU profiles.             *
* '-p' --used as -p <path> where 'path' is the relative  *
*        path to the directory where profiles are to  *
*        stored.                                      *
* '-i' --used as -i <from> <to> where 'from' and 'to' are*
*        integers to mark the desired profiling interval.*
******************************************************
```

7.2. TAU Trace Format Reader Library

7.2.1. Tau Reader Usage

7.2.1.1. SYNOPSIS

An API for reading data from TAU tracefiles

See TAU_tf.h

7.2.1.2. DESCRIPTION

The TAU Trace Format Reader system, defined in TAU_tf.h, operates primarily via a series C/C++ of callback methods, each representing a data type contained in a TAU tracefile. The TAU trace is opened with a call to the function: Ttf_FileHandleT Ttf_OpenFileForInput(const char *name, const char *edf); Where *name and *edf are the locations of the TAU trace and event definition files to be read respectively. The TtfFileHandleT returned is then used to access the TAU trace. e.g: fh = Ttf_OpenFileForInput(argv[1], argv[2]);

The callback methods are stored in a callback table, a struct of the type Ttf_Callbacks. The Ttf_Callbacks struct contains entities representing each of the 11 defined callbacks: Ttf_DefClkPeriodT DefClkPeriod; Ttf_DefThreadT DefThread; Ttf_DefStateGroupT DefStateGroup; Ttf_DefStateT DefState; Ttf_EndTraceT EndTrace; Ttf_EnterStateT EnterState; Ttf_LeaveStateT LeaveState; Ttf_SendMessageT SendMessage; Ttf_RecvMessageT RecvMessage; Ttf_DefUserEvent DefUserEvent; Ttf_EventTrigger EventTrigger; The struct also contains "void* UserData;" which is used as an argument to each of the callback functions.

The trace data relevant to a callback function's associated TAU event type are delivered to the function via its arguments. Additionally the user data argument may be used to pass in data defined elsewhere in the trace reading process, before the callback functions are invoked. The userData argument is often used to provide the file handler for functions to which the read trace events are being written. The userData argument must be recast to its original type before use. For example, the following callback function, compliant with the definition of Ttf_DefClkPeriodT, receives the clock period information from the TAU tracefile, in addition to the userData that specifies the the location of sprintf's target.

```
int ClockPeriod (void* userData, double clkPeriod ) {
    sprintf((char*)userData,"Clock period %g\n", clkPeriod);
```

```
    return 0;
}
```

The callback table should be initialized by associating each of its elements with a complementary callback function. As the trace file is read the contents of each entry will be passed to the corresponding callback function. Alternatively if an element of the callback table is set to 0 no action will be taken when its associated entry type is encountered. Each element of the callback table must be initialized to a viable function or 0. For example, to create a callback table, tb, and initialize the DefClkPeriod element with the function defined above one would use the following:

```
Ttf_CallbacksT cb;
cb.DefClkPeriod = ClockPeriod;
```

Once the callback table is initialized the tracefile reading may commence. This is done via the Ttf_ReadNumEvents function defined in TAU_tf.h. Because Ttf_ReadNumEvents requires trace files to be read in chunks of events of the number specified when the function is called it is common practice to read a tracefile by enclosing TtfReadNumEvents in a loop which terminates when there are no records left to read (i.e. the return value of TtF_ReadNumEvents is 0). The EndTrace callback function may also set a flag that breaks out of the loop. For example, using the Ttf_FileHandleT fh and Ttf_CallbacksT cb the following would process the entire tracefile specified within fh:

```
do{
    recs_read = Ttf_ReadNumEvents(fh,cb, 1024);
}
while (recs_read >=0);
```

In some circumstances it may be convenient to parse the trace files more than once using different callback tables and methods. For example, this technique is often useful when all of the the initialization records must be registered and processed before the event records.

When the processing of a tracefile is complete the tracefile should be closed with the Ttf_CloseFile(Ttf_FileHandleT fileHandle); function. eg:

```
Ttf_CloseFile(fh);
```

7.2.2. Callback API

7.2.2.1. int Ttf_DefClkPeriodT(userData, clkPeriod);

Arguments: void userData, double clkPeriod*

Returns: int status

This method is called when the trace reader encounters the definition of the clock period of the trace being read. It is called with the user defined argument userData and the clock period, clkPeriod, defined in the TAU trace. It should return 0 upon successful completion.

7.2.2.2. int Ttf_EndTraceT(userData,nodeToken,threadToken);

*Arguments: void *userData, unsigned int nodeToken, unsigned int threadToken*

Returns: int status

This method is called when an EOF is encountered in a tracefile. It is called with the user defined argument userData and the numeric ID of the node and thread, nodeToken and threadToken respectively, where the trace has ended. Note that the full trace has not concluded until the end of each node/thread combination has been reached. It should return 0 upon successful completion.

7.2.2.3. int Ttf_DefStateGroupT(userData, stateGroupToken, stateGroupName);

*Arguments: void *userData, unsigned int stateGroupToken, const char *stateGroupName*

Returns: int status

This method is called when the trace reader encounters a state group definition. It is called with the user defined argument userData the numeric ID of the state group being defined, stateGroupToken, and the name of the group being defined, stateGroupName. It should return 0 upon successful completion.

7.2.2.4. int Ttf_DefStateT(userData, stateToken, stateName, stateGroupToken);

*Arguments: void *userData, unsigned int stateToken, const char *stateName, unsigned int stateGroupToken*

Returns: int status

This method is called when the trace reader encounters a state definition. A state generally represents a programmatic function. It is called with the user defined argument userData, the numeric ID of the state being defined, stateToken, the name of the state being defined, stateName, and the numeric group ID of the state being defined, stateGroupToken. It should return 0 upon successful completion.

7.2.2.5. int Ttf_DefUserEvent(userData, userEventToken, userEventName, monotonicallyIncreasing);

*Arguments: void *userData, unsigned int userEventToken, const char *userEventName, int monotonicallyIncreasing*

Returns: int status

This method is called when the trace reader encounters a user defined event definition. It is called with the user defined argument userData, the numeric ID of the user defined event, userEventToken, the name of the user defined event, userEventName, and monotonicallyIncreasing a numeric indicator of if the user defined event is monotonically Increasing. If monotonicallyIncreasing is greater than 0 the user defined event's value will increase monotonically. If it is 0 then it will not be. It should return 0 upon successful completion.

7.2.2.6. int Ttf_EnterStateT(userData, time, nodeToken, threadToken, stateToken);

Arguments: void userData, double time, unsigned int nodeToken, unsigned int threadToken, unsigned int stateToken*

Returns: int status

This method is called when the trace reader encounters a state entry event. It is called with the user defined argument userData, the time of the state entry, time, the numeric ID of the node and thread

where the entry is taking place, nodeToken and threadToken respectively, and the numeric ID of the state that has been entered. It should return 0 upon successful completion.

7.2.2.7. int Ttf_LeaveStateT(userData, time, nodeToken, threadToken);

Arguments: void userData, double time, unsigned int nodeToken, unsigned int threadToken*

Returns: int status

This method is called when the trace reader encounters a state exit event. It is called with the user defined argument userData, the time of the state exit, time and the numeric IDs of the node and thread where the exit is taking place, nodeToken and threadToken respectively. It should return 0 upon successful completion.

7.2.2.8. int Ttf_SendMessageT(userData, time, sourceNodeToken, sourceThreadToken, destinationNodeToken, destinationThreadToken, messageSize, int messageTag);

Arguments: void userData, double time, unsigned int sourceNodeToken, unsigned int sourceThreadToken, unsigned int destinationNodeToken, unsigned int destinationThreadToken, unsigned int messageSize, unsigned int messageTag*

Returns: int status

This method is called when the trace reader encounters a message send event. It is called with the user defined argument userData, the time of the transmission, time, the numeric IDs of the node and thread from which the message was sent, sourceNodeToken and sourceThreadToken respectively, the numeric IDs of the node and thread to which the message was sent, destinationNodeToken and destinationThreadToken respectively, the size of the message, messageSize, and the numeric ID of the message, messageTag. It should return 0 upon successful completion.

7.2.2.9. int Ttf_RecvMessageT(userData, time, sourceNodeToken, sourceThreadToken, destinationNodeToken, destinationThreadToken, messageSize, int messageTag);

Arguments: void userData, double time, unsigned int sourceNodeToken, unsigned int sourceThreadToken, unsigned int destinationNodeToken, unsigned int destinationThreadToken, unsigned int messageSize, unsigned int messageTag*

Returns: int status

This method is called when the trace reader encounters a message receive event. It is called with the user defined argument userData, the time of the receipt, time, the numeric IDs of the node and thread from which the message was sent, sourceNodeToken and sourceThreadToken respectively, the numeric IDs of the node and thread to which the message was sent, destinationNodeToken and destinationThreadToken respectively, the size of the message, messageSize, and the numeric ID of the message, messageTag. It should return 0 upon successful completion.

7.2.2.10. int Ttf_EventTrigger(userData, time, nodeToken, threadToken, userEventToken, userEventValue);

*Arguments: void *userData, double time, unsigned int nodeToken, unsigned int threadToken, unsigned int userEventToken, long long userEventValue*

Returns: int status

This method is called when the trace reader encounters a user defined event trigger. It is called with the user defined argument data userData, the time of the event trigger, time, the numeric IDs of the node and thread where the event was triggered, nodeToken and threadToken respectively, the numeric ID of the user defined event triggered, userEventToken and the value recorded by the user defined event, userEventValue. It should return 0 upon successful completion.

7.2.3. TauReader API

7.2.3.1. Ttf_FileHandleT TtfOpenFileForInput(name, edf);

*Arguments: const char *name, const char *edf*

Returns: Ttf_FileHandleT fileHandle

Given the full name of the TAU trace file, name, and the corresponding event file, edf, and returns the virtual file handle that represents the trace in its entirety.

7.2.3.2. int Ttf_AbsSeek(handle, eventPosition);

Arguments: Ttf_FileHandleT handle, int eventPosition

Returns: int position

Given a Ttf_fileHandleT object, handle, this function will move to the nth event in the associated tracefile where n = the input int eventPosition. A negative position indicates to start from the tail of the event stream. The position will be returned if the operation is successful, otherwise it will return 0.

7.2.3.3. int Ttf_RelSeek(handle, plusMinusNumEvents);

Arguments: Ttf_FileHandleT handle, int plusMinusNumEvents

Returns: int position

Given a Ttf_fileHandleT object, handle, this function will shift the current position by a number of events equal to the input int plusMinusNumEvents. The new position will be returned if the operation is successful, otherwise it will return 0.

7.2.3.4. int Ttf_ReadNumEvents(fileHandle,callbacks, numberOfEvents);

Arguments: Ttf_FileHandleT fileHandle, Ttf_CallbacksT callbacks, int numberOfEvents

Returns: int numEventsRead

Given a Ttf_FileHandleT, handle, a fully initialized Ttf_CallbacksT struct, callbacks, and an integer indicating the number of events to read, numberOfEvents, this function will read the number of events indicated starting at the current position of the file handle while advancing the current position of the handle by that number. Each event read will be sent to the appropriate callback function specified in the callbacks. When successful this function returns the number of events read. This may be 0 or less than

the number specified if there are fewer remaining events to be read than numberOfEvents requests. If an error is encountered it will return -1.

7.2.3.5. Ttf_FileHandleT Ttf_CloseFile(fileHandle);

Arguments: Ttf_FileHandleT fileHandle

Returns: Ttf_FileHandleT fileHandle

When the tracefile reading is complete the file should be closed with this function.

Chapter 8. Tools

Name

vtf2profile -- Generate a TAU profile set from a vampir trace file

`vtf2profile [ -p profile ][ -i interval_start interval_end ][ -c ][ -h ]{ -f tracefile }`

Description

vtf2profile is created when TAU is configured with the -vtf=<vtf_dir> option. This tool converts a VTF trace file (*.vpt) to a tau profile set (profile.A.B.C where A, B and C are the node, context and thread numbers respectively).

The vtf file to be read is specified in the command line by the -f flag followed by the file's location. The VTF tracefile specified may be in gzipped form, eg app.vpt.gz. -p is similarly used to specify the relative path to the directory where the profile files should be stored. If no output directory is specified the current directory will be used. A contiguous interval within the vtf file may be selected for conversion by using the -i flag followed by two integers, representing the timestamp of the start and end of the desired interval respectively. The entire vtf file is converted if no interval is given.

Options

`-f tracefile` -Specify the Vampir tracefile to be converted.

`-p profile` -Specify the location where the profile file(s) should be written.

`-i interval_start interval_end` -Limit the profile produced to the specified interval within the vampir trace file.

`-c` -Opens a command line interface for the program.

`-h` -Displays a help message.

Examples

To convert a vampir tracefile, trace.vpt, to an equivalent TAU profile, use the following:

```
vtf2profile -f trace.vpt
```

To produce a TAU profile in the ./profiles directory representing only the events from the start of the tracefile to timestamp 6000, use:

```
vtf2profile -f trace.vpt -p ./profiles -i 0 6000
```

See Also

tau2vtf, tau2profile

Name

tau2vtf -- convert TAU tracefiles to vampir tracefiles

`tau2vtf` [`-nomessage`] [`-v`] [[`-a`] | [`-fa`]] { *`tau_tracefile`* } { *`tau_eventfile`* } { *`vtf_tracefile`* }

Description

This program is generated when TAU is configured with the -vtf=<vtf_dir> option.

The tau2vtf trace converter takes a single tau_tracefile (*.trc) and tau_eventfile (*.edf) and produces a corresponding vtf_tracefile (*.vtf). The input files and output file must be specified in that order. Multi-file TAU traces must be merged before conversion.

The default output file format is VTF3 binary. If the output filename is given as the .vpt.gz type, rather than .vpt, the output file will be gzipped. There are two additional output format options. The command line argument '-a' produces the vtf file output in ASCII VTF3 format. The command line argument '-fa' produces the vtf file output in the FAST ASCII VTF3 format. Note that these arguments are mutually exclusive.

Options

`-nomessage` Suppresses printing of message information in the trace.

`-v` Verbose mode sends trace event descriptions to the standard output as they are converted.

`-a` Print the vtf file output in the human-readable VTF3 ASCII format

`-fa` Print the vtf file in the simplified human-readable FAST ASCII VTF3 format

Examples

The program must be run with the tau trace, tau event and vtf output files specified in the command line in that order. Any additional arguments follow. The following will produce a VTF, app.vpt, from the TAU trace and event files merged.trc and tau.edf trace file:

```
tau2vtf merged.trc tau.edf app.vpt
```

The following will convert merged.trc and tau.edf to a gzipped FAST ASCII vampir tracefile app.vpt.gz, with message events omitted:

```
tau2vtf merged.trc tau.edf app.vpt.gz -nomessage -fa
```

See Also

vtf2profile, tau2profile, tau_merge, tau_convert

Name

tau2profile -- convert TAU tracefiles to TAU profile files

`tau2vprofile [ -d directory ] [ -s snapshot_interval ] { tau_tracefile } { tau_eventfile }`

Description

This program is generated when TAU is configured with the -TRACE option.

The tau2profile converter takes a single tau_tracefile (*.trc) and tau_eventfile (*.edf) and produces a corresponding series of profile files. The input files must be specified in that order, with optinal parameters coming afterward. Multi-file TAU traces must be merged before conversion.

Options

`-d` Output profile files to the specified 'directory' rather than the current directory.

`-s` Output a profile snapshot showing the state of the profile data accumulated from the trace every 'snapshot_interval' time units. The snapshot profiles are placed sequentially in directories labled 'snapshot_n' where 'n' is an integer ranging from 0 to to the total number of snapshots -1.

Examples

The program must be run with the tau trace and tau event files specified in the command line in that order. Any additional arguments follow. The following will produce a profile file array, from the TAU trace and event files merged.trc and tau.edf trace file:

```
tau2profile merged.trc tau.edf
```

The following will convert merged.trc and tau.edf to a series of profiles one directory higher. It will also produce a profile snapshot every 250,000 time units:

```
tau2profile merged.trc tau.edf -d ./.. -s 250000
```

See Also

vtf2profile, tau2vtf, tau2otf, tau_merge, tau_convert

Name

tau2elg -- convert TAU tracefiles to Epilog tracefiles

`tau2elg [ -nomessage ] [ -v ] { tau_tracefile } { tau_eventfile } { elg_tracefile }`

Description

This program is generated when TAU is configured with the -epilog=<epilog_dir> option.

The tau2elg trace converter takes a tau trace file (*.trc) and event definition file (*.edf) and produces a corresponding epilog binary trace file (*.elg). Multi-file TAU traces must be merged before conversion.

Options

`-nomessage` Suppresses printing of message information in the trace.

`-v` Verbose mode sends trace event descriptions to the standard output as they are converted.

Examples

The program must be run with the tau trace, tau event and elg output files specified in the command line in that order. Any additional arguments follow. The following would convert merged.trc and tau.edf to the Epilog tracefile app.elg, with message events omitted:

```
./tau2vtf merged.trc tau.edf app.elg -nomessage
```

See Also

tau_merge

Name

tau2slog2 -- convert TAU tracefiles to SLOG2 tracefiles

`tau2slog2` { *`tau_tracefile`* } { *`tau_eventfile`* } { `-o` *`slog2_tracefile`* }

Description

This program is generated when TAU is configured with the -slog2 or -slog2=<slog2_dir> option.

The tau2slog2 trace converter takes a single tau trace file (*.trc) and event definition file (*.edf) and produces a corresponding slog2 binary trace file (*.slog2).

The tau2slog2 converter is called from the command line with the locations of the tau trace and event files. These arguments must be followed by the -o flag and the name of the slog2 file to be written. tau2slog 2 accepts no other arguments.

Examples

A typical invocation of the converter, to create app.slog2, is as follows:

```
tau2slog2 app.trc tau.edf -o app.slog2
```

See Also

tau_merge, tau_convert

Name

tau2otf -- convert TAU tracefiles to OTF tracefiles for Vampir/VNG

`tau2otf` [`-n` *`streams`*] [`-nomessage`] [`-v`]

Description

This program is generated when TAU is configured with the -otf=<otf_dir> option. The tau2otf trace converter takes a TAU formatted tracefile (*.trc) and a TAU event description file (*.edf) and produces an output trace file in the Open Trace Format (OTF). The user may specify the number of output streams for OTF. The input files and output file must be specified in that order. TAU traces should be merged using tau_merge prior to conversion.

Options

`-n` streams Specifies the number of output streams (default is 1). `-nomessage` Suppresses printing of message information in the trace. `-v` Verbose mode sends trace event descriptions to the standard output as they are converted.

Examples

The program must be run with the tau trace, tau event and otf output files specified in the command line in that order. Any additional arguments follow. The following will produced an OTF file, a pp.otf and other related event and definition files, from the TAU trace and event files merged.trc and tau.edf trace file:

```
tau2otf merged.trc tau.edf app.otf
```

See Also

tau2vtf(1), tau2profile(1), vtf2profile(1), tau_merge(1), tau_convert(1)

Name

tau_merge -- combine multiple node and or thread TAU tracefiles into a merged tracefile

`tau_merge` [`-a`] [`-r`] [`-n`] [`-e` *`eventfile_list`*] [`-m` *`output_eventfile`*] { *`trace-file_list`* } [{ *`output_tracefile`* } | { `-` }]

Description

tau_merge is generated when TAU is configured with the -TRACE option.

This tool assembles a set of tau trace and event files from multiple multiple nodes or threads across a program's execution into a single unified trace file. Many TAU trace file tools operate on merged trace files.

Minimally, tau_merge must be invoked with a list of unmerged trace files followed by the desired name of the merged trace file or the - flag to send the output to the standard out. Typically the list can be designated by giving the shared name of the trace files to be merged followed by desired range of thread or node designators in brackets or the wild card character '*' to encompass variable thread and node designations in the filename (trace.A.B.C.trc where A, B and C are the node, context and thread numbers respectively). For example tautrace.*.trc would represent all tracefiles in a given directory while tautrace.[0-5].0.0.trc would represent the tracefiles of nodes 0 through 5 with context 0 and thread 0.

tau_merge will generate the specified merged trace file and an event definition file, tau.edf by default.

The event definition file can be given an alternative name by using the '-m' flag followed by the desired filename. A list of event definition files to be merged can be designated explicitly by using the '-e' flag followed by a list of unmerged .edf files, specified in the same manner as the trace file list.

If computational resources are insufficient to merge all trace and event files simultaneously the process may be undertaken hierarchically. Corresponding subsets of the tracefiles and eventfiles may be merged in sequence to produce a smaller set of files that can then be to merged into a singular fully merged tracefile and eventfile. E.g. for a 100 node trace, trace sets 1-10, 11-20, ..., 91-100 could be merged into traces 1a, 2a, ..., 10a. Then 1a-10a could be merged to create a fully merged tracefile.

Options

`-e` eventfile_list explicitly define the eventfiles to be merged

`-m` output_eventfile explicitly name the merged eventfile to be created

`-` send the merged tracefile to the standard out

`-a` adjust earliest timestamp time to zero

`-r` do not reassemble long events

`-n` do not block waiting for new events. By default tau_merge will block and wait for new events to be appended if a tracefile is incomplete. This command allows offline merging of (potentially) incomplete tracefiles.

Examples

To merge all TAU tracefiles into app.trc and produce a merged tau.edf eventfile:

```
tau_merge *.trc app.trc
```

To merge all eventfiles 0-255 into ev0_255merged.edf and TAU tracefiles for nodes 0-255 into the standard out:

```
tau_merge -e events.[0-255].edf -m ev0_255merged.edf \
  tautrace.[0-255].*.trc -
```

To merge eventfiles 0, 5 and seven info ev057.edf and tau tracefiles for nodes 0, 5 and 7 with context and thread 0 into app.trc:

```
tau_merge -e events.0.edf events.5.edf events.7.edf -m ev057.edf \
  tautrace.0.0.0.trc tautrace.5.0.0.trc tautrace.7.0.0.trc app.trc
```

See Also

tau_convert

tau2profile

tau2vtf

tau2elg

tau2slog2

Name

tau_convert -- convert TAU tracefiles into various alternative trace formats

`tau_convert` [[`-alog`] | [`-SSDF`] | [`-dump`] | [`-paraver` [-t]] | [`-pv`] | [`-vampir` [`-longsymbolbugfix`] [`-compact`] [[`-user`] | [`-class`] | [`-all`]] [`-nocomm`]]] [*`outputtrc`*] { *`inputtrc`* } { *`edffile`* }

Description

tau_convert is generated when TAU is configured with the -TRACE option.

This program requires specification of a TAU tracefile and eventfile. It will convert the given TAU traces to the ASCII-based trace format specified in the first argument. The conversion type specification may be followed by additional options specific to the conversion type. It defaults to the single threaded vampir format if no other format is specified. tau_convert also accepts specification of an output file as the last argument. If none is given it prints the converted data to the standard out.

Options

`-alog` convert TAU tracefile into the alog format (This format is deprecated. The SLOG2 format is recommended.)

`-SDDF` convert TAU tracefile into the SDDF format

`-dump` convert TAU tracefile into multi-column human readable text

`-paraver` convert TAU tracefile into paraver format

`-t` indicate conversion of multi threaded TAU trace into paraver format

`-pv` convert single threaded TAU tracefile into vampir format (all -vampir options apply) (default)

`-vampir` convert multi threaded TAU tracefile into vampir format

`-longsymbolbugfix` make the first characters of long, similar identifier strings unique to avoid a bug in vampir

`-compact` abbreviate individual event entries

`-all` compact all entries (default)

`-user` compact user entries only

`-class` compact class entries only

`-nocomm` disregard communication events

`[outputtrc]` specify the name of the output tracefile to be produced

Examples

To print the contents of a TAU tracefile to the screen:

```
tau_convert -dump app.trc tau.edf
```

To convert a merged, threaded TAU tracefile to paraver format:

```
tau_convert -paraver -t app.trc tau.edf app.pv
```

See Also

tau_merge, tau2vtf, tau2profile, tau2slog2

Name

tau_reduce -- generates selective instrumentation rules based on profile data

`tau_reduce { -f filename } [ -n ] [ -r filename ] [ -o filename ] [ -v ] [ -p ]`

Description

tau_reduce is an application that will apply a set of user-defined rules to a pprof dump file (**pprof -d**) in order to create a select file that will include an exclude list for selective implementation for TAU. The user must specify the name of the pprof dump file that this application will use. This is done with the -f filename flag. If no rule file is specified, then a single default rule will be applied to the file. This rule is: numcalls > 1000000 & usecs/call < 2, which will exclude all routines that are called at least 1,000,000 times and average less then two microseconds per call. If a rule file is specified, then this rule is not applied. If no output file is specified, then the results will be printed out to the screen.

Rules

Users can specify a set of rules for tau_reduce to apply. The rules should be specified in a separate file, one rule per line, and the file name should be specifed with the appropriate option on the command line. The grammar for a rule is: [GROUPNAME:]FIELD OPERATOR NUMBER. The GROUPNAME followed by the colon (:) is optional. If included, the rule will only be applied to routines that are a member of the group specified. Only one group name can be applied to each rule, and a rule must follow a groupname. If only a groupname is given, then an unrecognized field error will be returned. If the desired effect is to exclude all routines that belong to a certain group, then a trivial rule, such as GROUP:numcalls > -1 may be applied. If a groupnameis given, but the data does not contain any groupname data, then then an error message will be given, but the rule will still be applied to the date ignoring the groupname specification. A FIELD is any of the routine attributes listed in the following table:

Table 8.1. Selection Attributes

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

Some FIELDS are only available for certain files. If hardware counters are used, then usec, cumusec, usecs/per call are not applicable and a error is reported. The opposite is true if timing data is used rather than hardware counters. Also, stddev is only available for certain files that contain that data.

An OPERATOR is any of the following: < (less than), > (greater than), or = (equals).

A NUMBER is any number.

A compound rule may be formed by using the & (and) symbol in between two simple rules. There is no "OR" because there is an implied or between two separate simple rules, each on a separate line. (ie the compound rule usec < 1000 OR numcalls = 1 is the same as the two simple rules "usec < 1000" and "numcalls = 1").

Rule Examples

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

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

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

NOTE: Any line in the rule file that begins with a # is a comment line. For clarity, blank lines may be inserted in between rules and will also be ignored.

Options

`-f` filename specify filename of pprof dump file

`-p` print out all functions with their attributes

`-o` filename specify filename for select file output (default: print to screen

`-r` filename specify filename for rule file

`-v` verbose mode (for each rule, print out rule and all functions that it excludes)

Examples

To print to the screen the selective instrumentation list for the paraprof dump file app.prf with default selection rules use:

```
tau_reduce -f app.prf
```

To create a selection file, app.sel, from the paraprof dump file app.prf using rules specified in foo.rlf use:

```
tau_reduce -f app.prf -r foo.rlf -o app.sel
```

See Also

Name

tau_ompcheck -- Completes uncompleted do/for/parallel omp directives

`tau_ompcheck` { *`pdbfile`* } { *`sourcefile`* } [`-o` *`outfile`*][`-v`][`-d`]

Description

Finds uncompleted do/for omp directives and inserts closing directives for each one uncompleted. do/for directives are expected immediately before a do/for loop. Closing directives are then placed immediately following the same do/for loop.

Options

`pdbfile` A pdbfile generated from the source file you wish to check. This pdbfile must contain comments from which the omp directives are gathered. See pdbcomment for information on how to obtain comment from a pdbfile.

`sourcefile` A fortran, C or C++ source file to analyzed.

`-o` write the output to the specified outfile.

`-v`verbose output, will say which directive where added.

`-d` debuging information, we suggest you pipe this unrestrained output to a file.

Examples

To check file: source.f90 do: (you will need pdtoolkit/<arch>/bin and tau/utils/ in your path).

```
%>f95parse source.f90
%>pdbcomment source.pdb > source.comment.pdb
%>tau_omp source.comment.pdb source.f90 -o source.chk.f90
```

See Also

f95parse pdbcomment

Name

tau_poe -- Instruments a MPI application while it is being executed with poe.

`tau_poe` [`-XrunTAUsh-` *`tauOptions`*] { *`applcation`* } [*`poe options`*]

Description

This tool dynamically instruments a mpi application by loading a specific mpi library file.

Options

`tauOptions` To instrument a mpi application a specific TAU library file is loaded when the application is executed. To select which library is loaded use this option. The library files are build according to the options set when TAU is configured. The library file that have been build and thus available for use are in the [TAU_HOME]/[arch]/lib directory. The file are listed as libTAUsh-*.so where * is the instrumentation options. For example to use the libTAUsh-pdt-openmp-opari.so file let the comman line option be -XrunTAUsh-pdt-openmp-opari.

Examples

Instrument a.out wit the currently configured options and then run it on four nodes:

```
%>tau_poe ./a.out -procs 4
```

Select the libTAUsh-mpi.so library to instrument a.out with:

```
%>tau_poe -XrunTAUsh-mpi ./a.out -procs 4
```

Part II. ParaProf

Table of Contents

Chapter 9. Introduction

ParaProf is a portable, scalable performance analysis tool included with the TAU distribution.

Important

ParaProf requires Java 1.3 for basic functionality. Java 1.4 is required for 3d visualization and image export. Additionally, OpenGL is required for 3d visualization.

Note

Most windows in ParaProf can export bitmap (png/jpg) and vector (svg/eps) images to disk (png/jpg) or print directly to a printer. This are available through the *File* menu.

9.1. Using ParaProf from the command line

ParaProf is a java program that is run from the supplied **paraprof** script (**paraprof.bat** for windows binary release).

```
% paraprof --help
Usage: paraprof [options] <files/directory>

Options:

  -f, --filetype <filetype>   Specify type of performance data,
                                options are: profiles (default), pprof,
                                dynaprof, mpip, gprof, psrun, hpm,
                                packed, cube, hpc

  -h, --help                  Display this help message
  -p                          Use `pprof` to compute derived data
  -i, --fixnames              Use the fixnames option for gprof

  --pack <file>               Pack the data into packed (.ppk) format
                                (does not launch ParaProf GUI)
  --dump                      Dump profile data to TAU profile format
                                (does not launch ParaProf GUI)

Notes:
  For the TAU profiles type, you can specify either a specific set of
  profile files on the commandline, or you can specify a directory
  (by default the current directory).  The specified directory will
  be searched for profile.*.*.* files, or, in the case of
  multiple counters, directories named MULTI_* containing profile data.
```

9.2. Supported Formats

ParaProf can load profile date from many sources. The types currently supported are:

- **TAU Profiles** - Output from the TAU measurement library, these files generally take the form of `profile.X.X.X`, one for each node/context/thread combination. When multiple counters are used, each metric is located in a directory prefixed with "MULTI__". To launch ParaProf with all the

metrics, simply launch it from the root of the MULTI__ directories.

- **pprof** - Dump Output from TAU's **pprof -d**. Provided for backward compatibility only.
- **DynaProf** - Output From DynaProf's wallclock and papi probes.
- **mpiP** - Output from mpiP.
- **gprof** - Output from gprof, see also the --fixnames option.
- **HPM Toolkit** - Output from HPM Toolkit.
- **ParaProf Packed Format** - Export format supported by PerfDMF/ParaProf. Typically .ppk.
- **Cube** - Output from Kojak Expert tool for use with Cube.
- **HPCToolkit** - XML data from hpcquick. Typically, the user runs hpcrun, then hpcquick on the resulting binary file.

9.3. Command line options

In addition to specifying the profile format, the user can also specify the following options

- **--fixnames** - Use the fixnames option for gprof. When C and Fortran code are mixed, the C routines have to be mapped to either .function or function_. Strip the leading period or trailing underscore, if it is there.
- **--pack <file>** - Rather than load the data and launch the GUI, pack the data into the specified file.
- **--dump** - Rather than load the data and launch the GUI, dump the data to TAU Profiles. This can be used to convert supported formats to TAU Profiles.

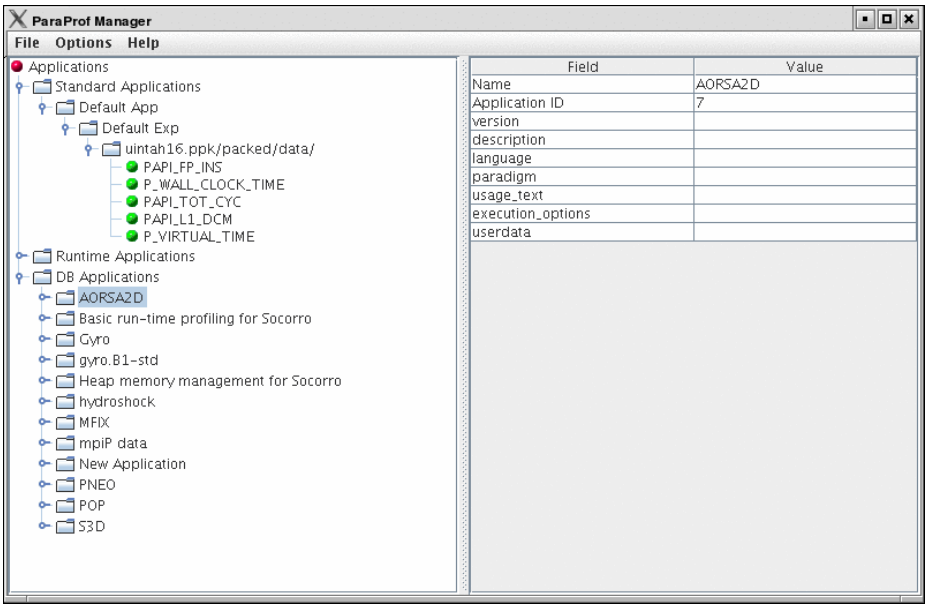
Chapter 10. Profile Data Management

ParaProf uses *PerfDMF* to manage profile data. This enables it to read the various profile formats as well as store and retrieve them from a database.

10.1. ParaProf Manager Window

Upon launching ParaProf, the user is greeted with the ParaProf Manager Window.

Figure 10.1. ParaProf Manager Window

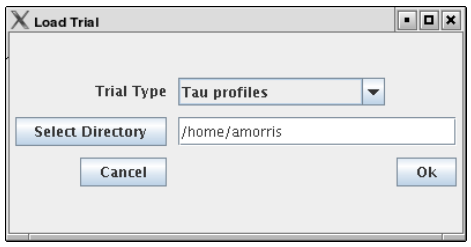


This window is used to manage profile data. The user can upload/download profile data, edit meta-data, launch visual displays, export data, derive new metrics, etc.

10.2. Loading Profiles

To load profile data, select File->Open, or right click on the Application's tree and select "Add Trial".

Figure 10.2. Loading Profile Data



Select the type of data from the "Trial Type" drop-down box. For TAU Profiles, select a directory, for other types, files.

10.3. Database Interaction

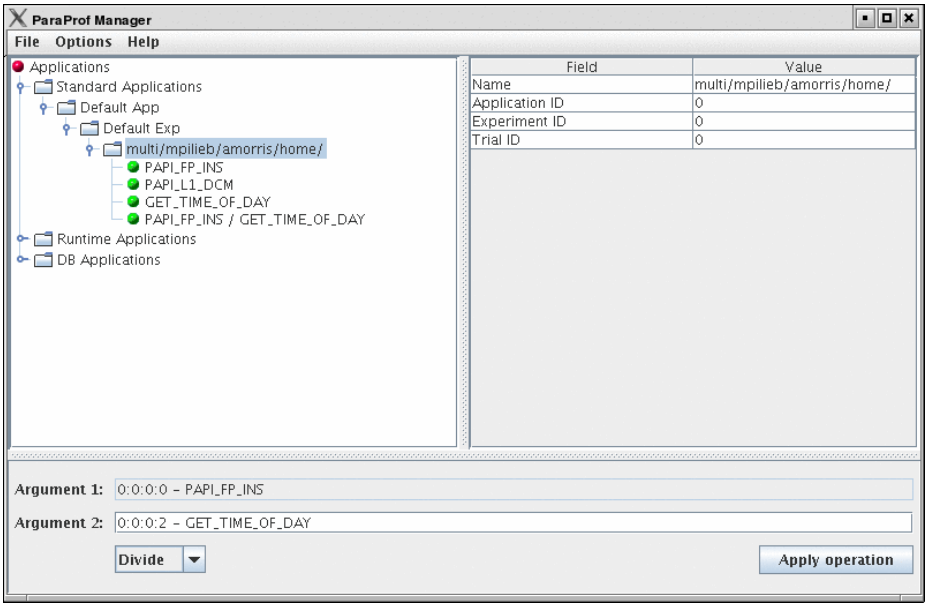
Database interaction is done through the tree view of the ParaProf Manager Window. Applications expand to Experiments, Experiments to Trials, and Trials are loaded directly into ParaProf just as if they were read off disk. Additionally, the meta-data associated with each element is show on the right, as in Figure 10.1, “ParaProf Manager Window”. A trial can be exported by right clicking on it and selecting "Export as Packed Profile".

New trials can be uploaded to the database by either right-clicking on an entity in the database and selecting "Add Trial", or by right-clicking on an Application/Experiment/Trial hierarchy from the "Standard Applications" and selecting "Upload Application/Experiment/Trial to DB".

10.4. Creating Derived Metrics

ParaProf can created derived metrics using the *Derived Metric Panel*, available from the *Options* menu of the ParaProf Manager Window.

Figure 10.3. Creating Derived Metrics

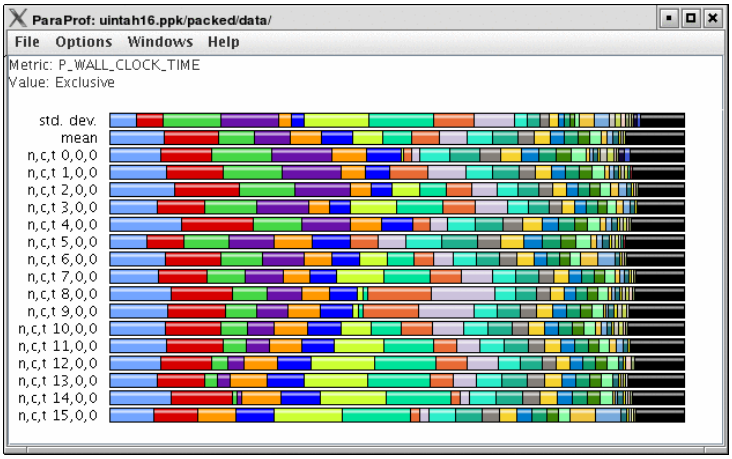


In Figure 10.3, “Creating Derived Metrics”, we have just divided Floating Point Instructions by Wall-clock time, creating FLOPS (Floating Point Operations per Second). The 2nd argument is a user editable text-box and can be filled in with scalar values by using the keyword 'val' (e.g. "val 1.5").

10.5. Main Data Window

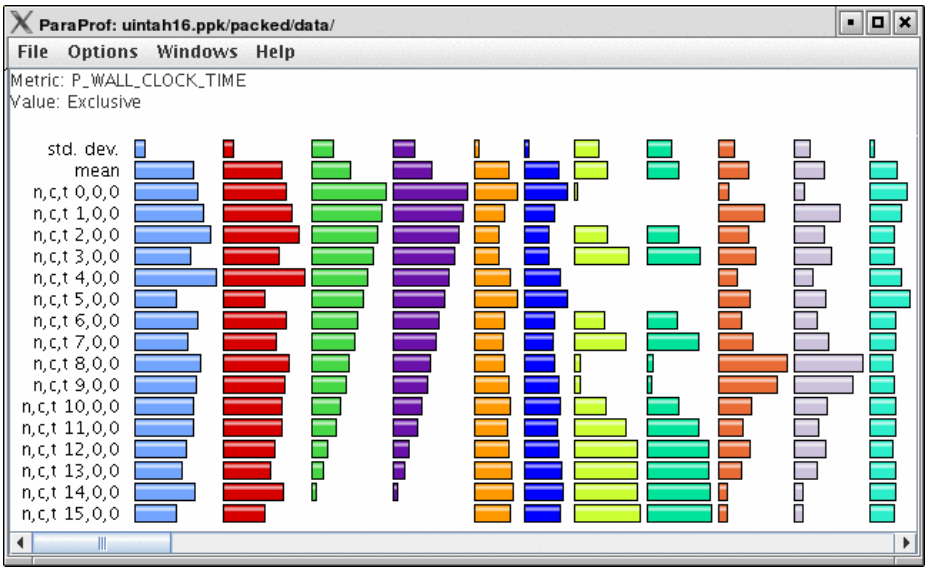
Upon loading a profile, or double-clicking on a metric, the *Main Data Window* will be displayed.

Figure 10.4. Main Data Window



This window shows each thread as well as statistics as a combined bar graph. Each function is represented by a different color (though possibly cycled). From anywhere in ParaProf, you can right-click on objects representing threads or functions to launch displays associated with those objects. For example, in Figure 10.4, “Main Data Window”, right click on the text *n,c,t, 8,0,0* to launch thread based displays for node 8.

Figure 10.5. Unstacked Bars



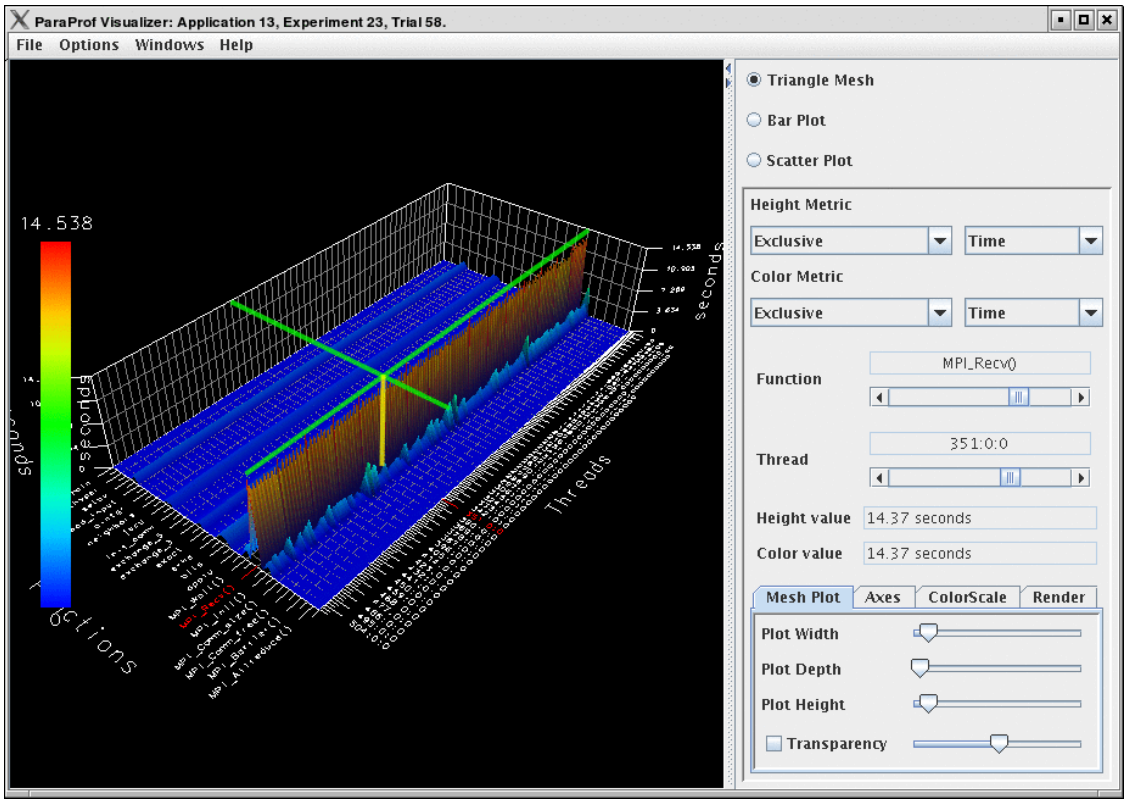
You may also turn off the stacking of bars so that individual functions can be compared across threads in a global display.

Chapter 11. 3-D Visualization

ParaProf displays massive parallel profiles through the use of OpenGL hardware acceleration through the *3D Visualization* window. Each window is fully configurable with rotation, translation, and zooming capabilities. Rotation is accomplished by holding the left mouse button down and dragging the mouse. Translation is done likewise with the right mouse button. Zooming is done with the mousewheel and the + and - keyboard buttons.

11.1. Triangle Mesh Plot

Figure 11.1. Triangle Mesh Plot

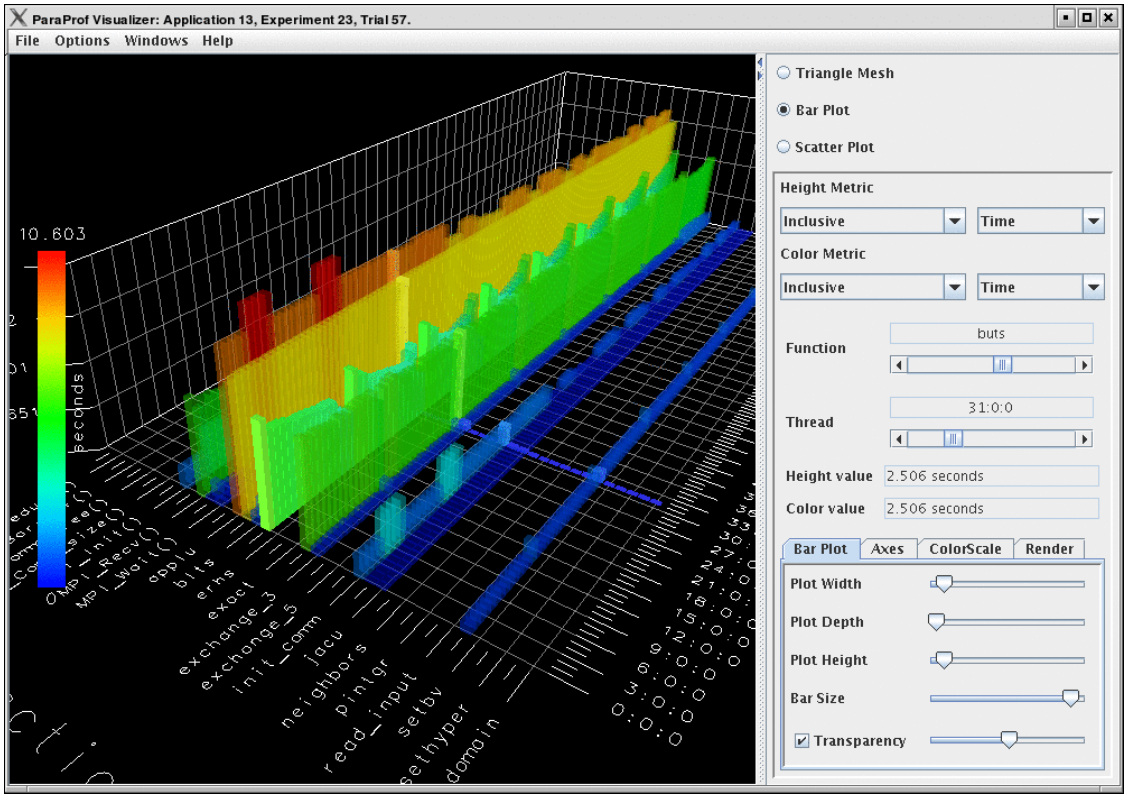


This visualization method shows two metrics for all functions, all threads. The height represents one chosen metric, and the color, another. These are selected from the drop-down boxes on the right.

To pinpoint a specific value in the plot, move the *Function* and *Thread* sliders to cycle through the available functions/threads. The values for the two metrics, in this case for `MPI_Recv()` on `Node 351`, the value is `14.37` seconds.

11.2. 3-D Bar Plot

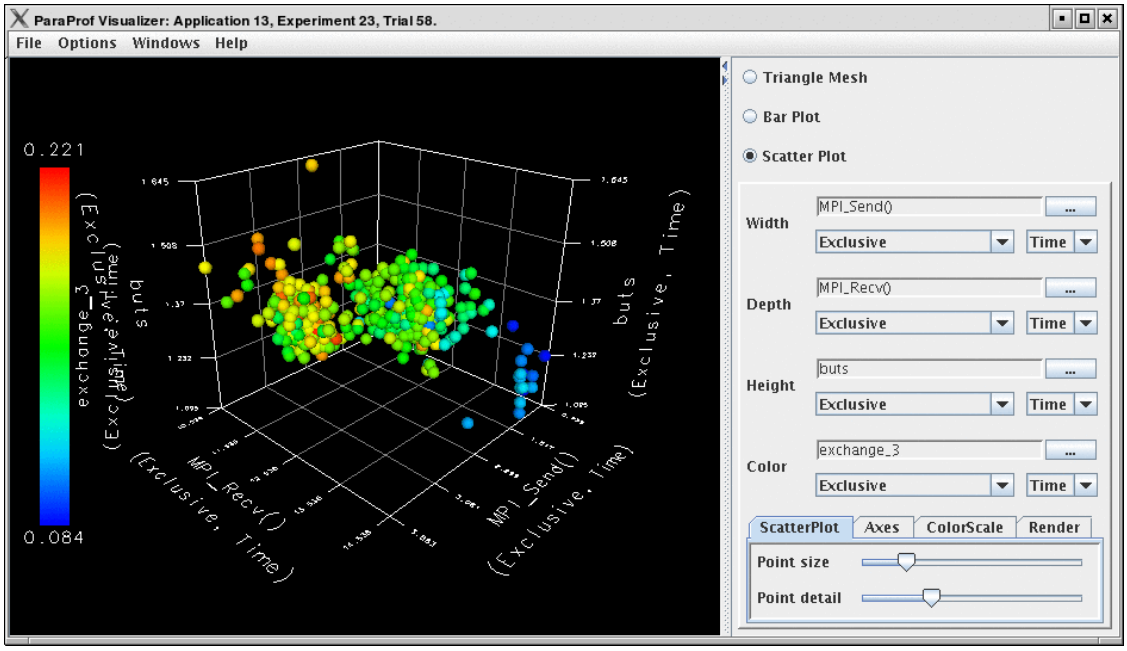
Figure 11.2. 3-D Mesh Plot



This visualization method is similar to the triangle mesh plot. It simply displays the data using 3d bars instead of a mesh. The controls works the same. Note that in Figure 11.2, “3-D Mesh Plot” the transparency option is selected, which changes the way in which the selection model operates.

11.3. 3-D Scatter Plot

Figure 11.3. 3-D Scatter Plot



This visualization method plots the value of each thread along up to 4 axes (each a different function/metric). This view allows you to discern clustering of values and relationships between functions across threads.

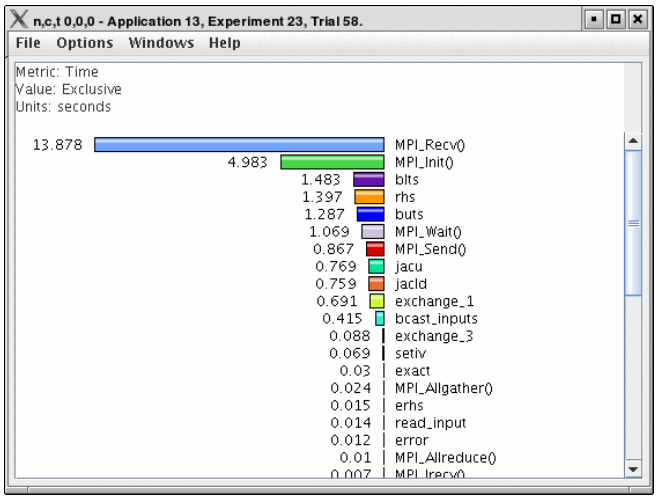
Select functions using the button for each dimension, then select a metric. A single function across 4 metrics could be used, for example.

Chapter 12. Thread Based Displays

ParaProf displays several windows that show data for one thread of execution. In addition to per thread values, the users may also select *mean* or *standard deviation* as the "thread" to display. In this mode, the mean or standard deviation of the values across the threads will be used as the value.

12.1. Thread Bar Graph

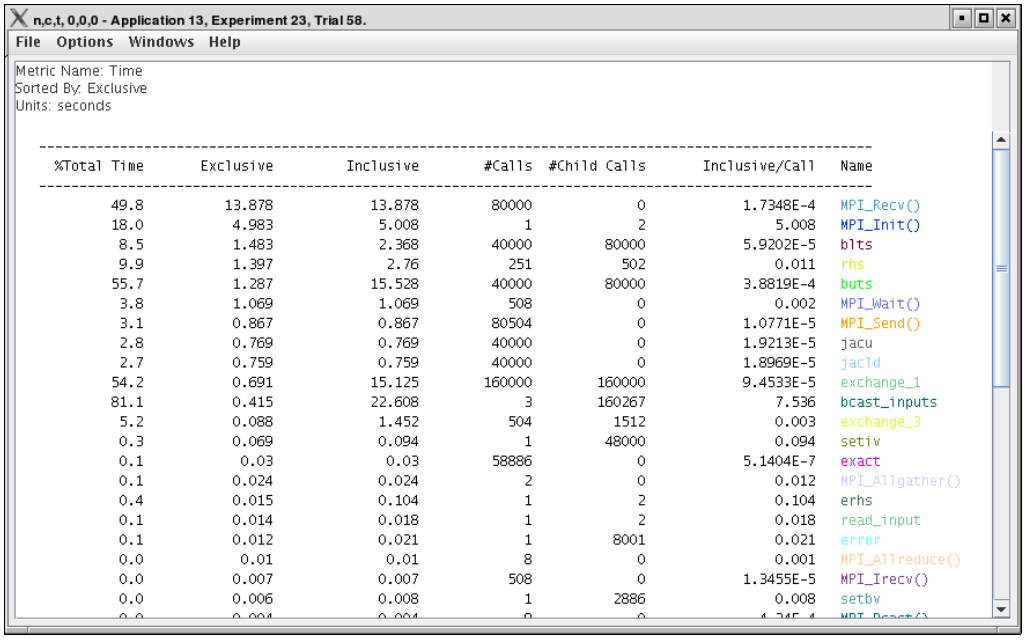
Figure 12.1. Thread Bar Graph



This display graphs each function on a particular thread for comparison. The metric, units, and sort order can be changed from the *Options* menu.

12.2. Thread Statistics Text Window

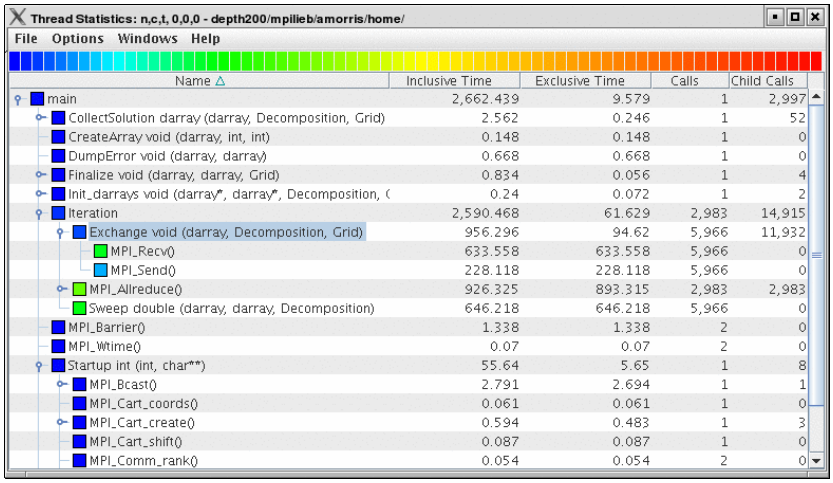
Figure 12.2. Thread Statistics Text Window



This display shows a **pprof** style text view of the data.

12.3. Thread Statistics Table

Figure 12.3. Thread Statistics Table, inclusive/exclusive



This display shows the callpath data in a table. Each callpath can be traced from root to leaf by opening each node in the tree view. A colorscale immediately draws attention to "hot spots", areas that contain highest values.

Figure 12.4. Thread Statistics Table

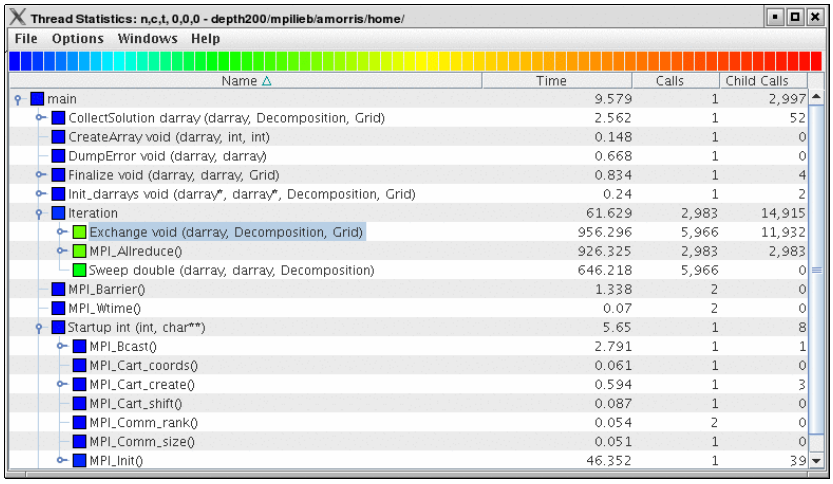
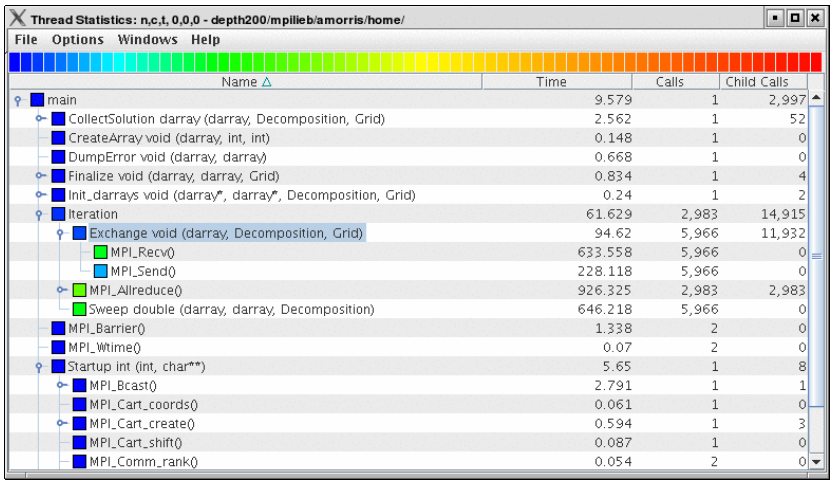


Figure 12.5. Thread Statistics Table

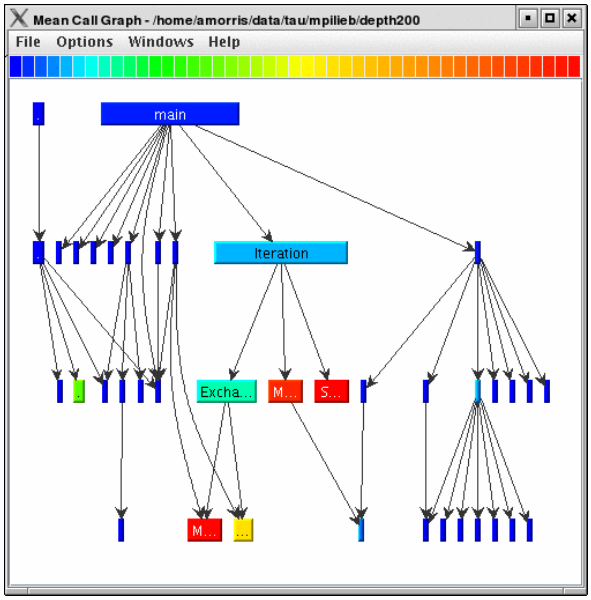


The display can be used in one of two ways, in "inclusive/exclusive" mode, both the inclusive and exclusive values are shown for each path, see Figure 12.3, “Thread Statistics Table, inclusive/exclusive” for an example.

When this option is off, the inclusive value for a node is show when it is closed, and the exclusive value is shown when it is open. This allows the user to more easily see where the time is spent since the total time for the application will always be represented in one column. See Figure 12.4, “Thread Statistics Table” and Figure 12.5, “Thread Statistics Table” for examples. This display also functions as a regular statistics table without callpath data. The data can be sorted by columns by clicking on the column heading. When multiple metrics are available, you can add and remove columns for the display using the menu.

12.4. Call Graph Window

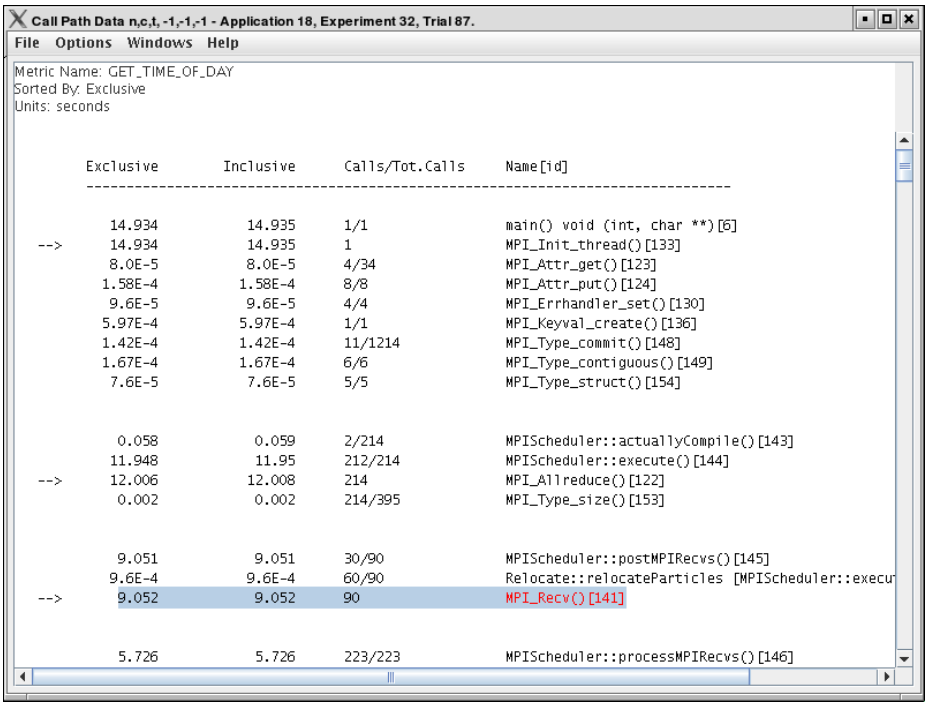
Figure 12.6. Call Graph Window



This display shows callpath data in a graph using two metrics, one determines the width, the other the color. The full name of the function as well as the two values (color and width) are displayed in a tooltip when hovering over a box. By clicking on a box, the actual ancestors and descendants for that function and their paths (arrows) will be highlighted with blue. This allows you to see which functions are called by which other functions since the interplay of multiple paths may obscure it.

12.5. Thread Call Path Relations Window

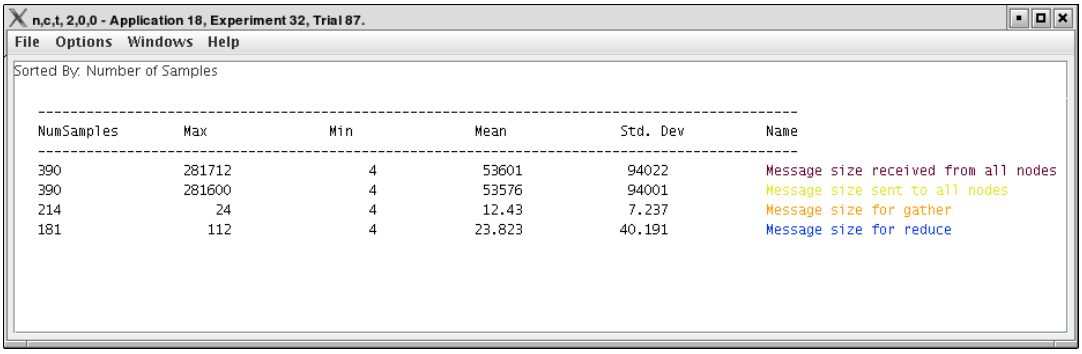
Figure 12.7. Thread Call Path Relations Window



This display shows callpath data in a **gprof** style view. Each function is shown with its immediate parents. For example, Figure 12.7, “Thread Call Path Relations Window” shows that `MPI_Recv()` is call from two places for a total of `9.052` seconds. Most of that time comes from the 30 calls when `MPI_Recv()` is called by `MPIScheduler::postMPIRecvs()`. The other 60 calls do not amount to much time.

12.6. User Event Statistics Window

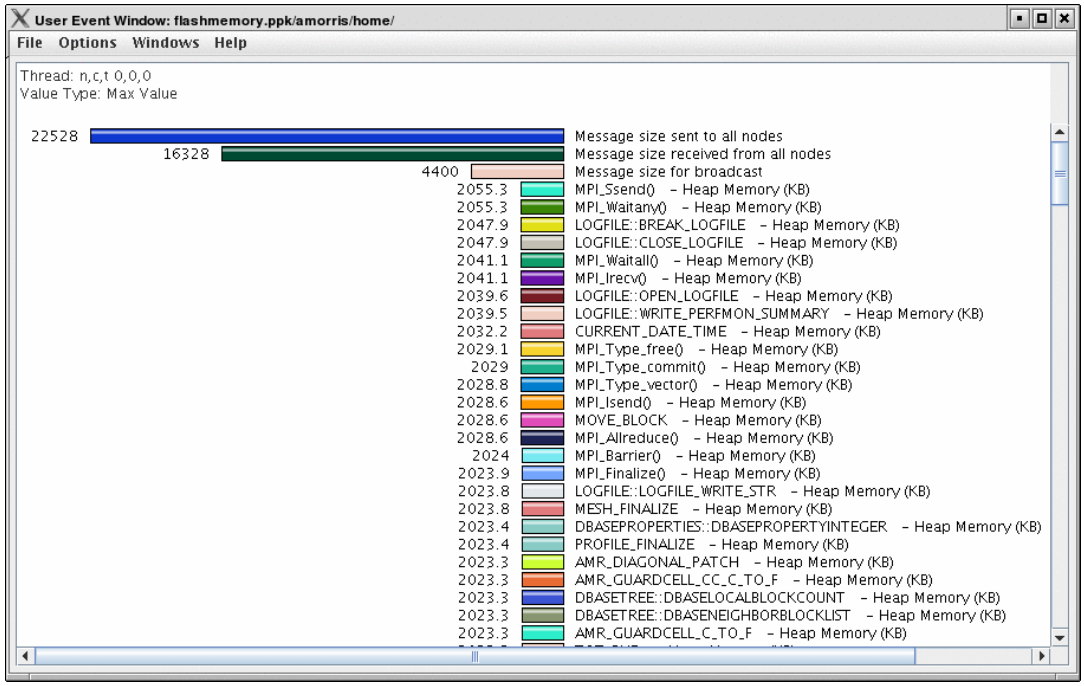
Figure 12.8. User Event Statistics Window



This display shows a **pprof** style text view of the user event data. Right clicking on a User Event will give you the option to open a Bar Graph for that particular User Event across all threads. See Section 16.1, “User Event Bar Graph”

12.7. User Event Thread Bar Chart

Figure 12.9. User Event Thread Bar Chart Window



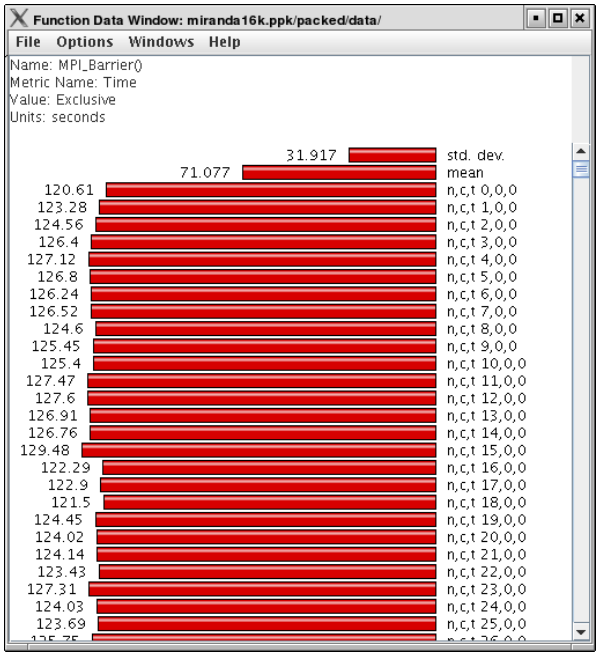
This display shows a particular thread's user defined event statistics as a bar chart. This is the same data from the Section 12.6, “User Event Statistics Window”, in graphical form.

Chapter 13. Function Based Displays

ParaProf has two displays for showing a single function across all threads of execution. This chapter describes the Function Bar Graph Window and the Function Histogram Window.

13.1. Function Bar Graph

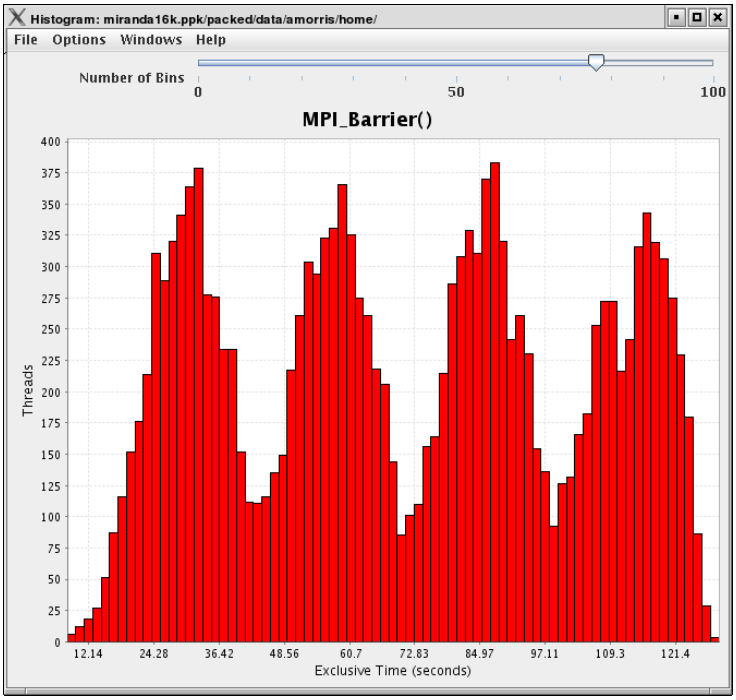
Figure 13.1. Function Bar Graph



This display graphs the values that the particular function had for each thread along with the mean and standard deviation across the threads. You may also change the units and metric displayed from the *Options* menu.

13.2. Function Histogram

Figure 13.2. Function Histogram



This display shows a histogram of each thread's value for the given function. Hover the mouse over a given bar to see the range minimum and maximum and how many threads fell into that range. You may also change the units and metric displayed from the *Options* menu.

You may also dynamically change how many bins are used (1-100) in the histogram. This option is available from the *Options* menu. Changing the number of bins can dramatically change the shape of the histogram, play around with it to get a feel for the true distribution of the data.

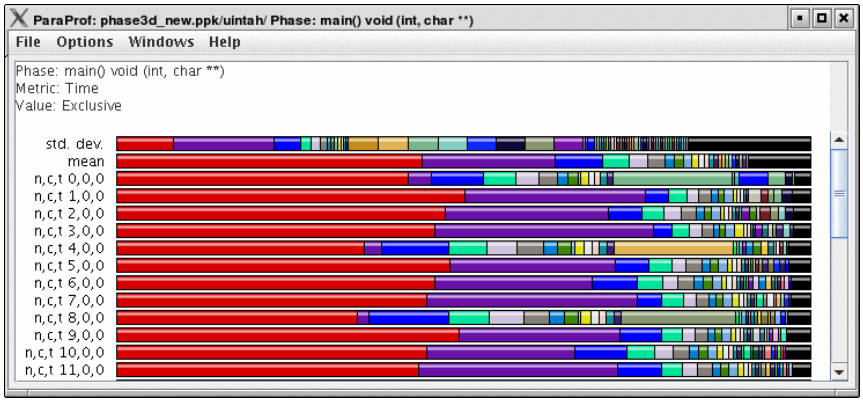
Chapter 14. Phase Based Displays

When a profile contains phase data, ParaProf will automatically run in phase mode. Most displays will show data for a particular phase. This phase will be displayed in teh top left corner in the meta data panel.

14.1. Using Phase Based Displays

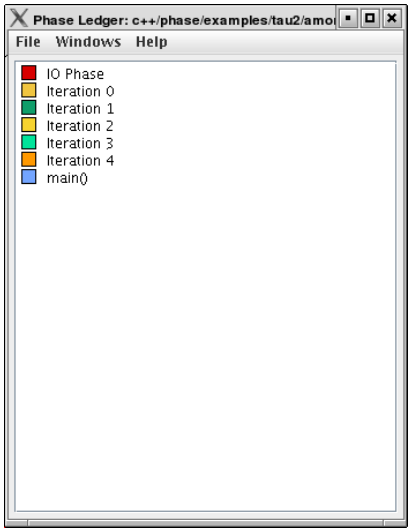
The initial window will default to top level phase, usually *main*

Figure 14.1. Initial Phase Display



To access other phases, either right click on the phase and select, "Open Profile for this Phase", or go to the *Phase Ledger* and select it there.

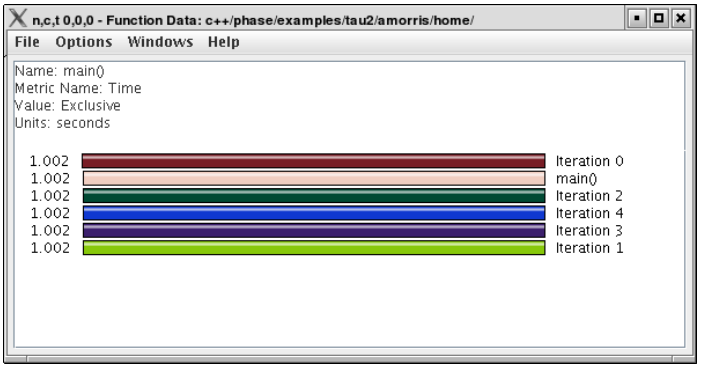
Figure 14.2. Phase Ledger



ParaProf can also display a particular function's value across all of the phases. To do so, right click on a

function and select, "Show Function Data over Phases".

Figure 14.3. Function Data over Phases



Because Phase information is implemented as callpaths, many of the callpath displays will show phase data as well. For example, the Call Path Text Window is useful for showing how functions behave across phases.

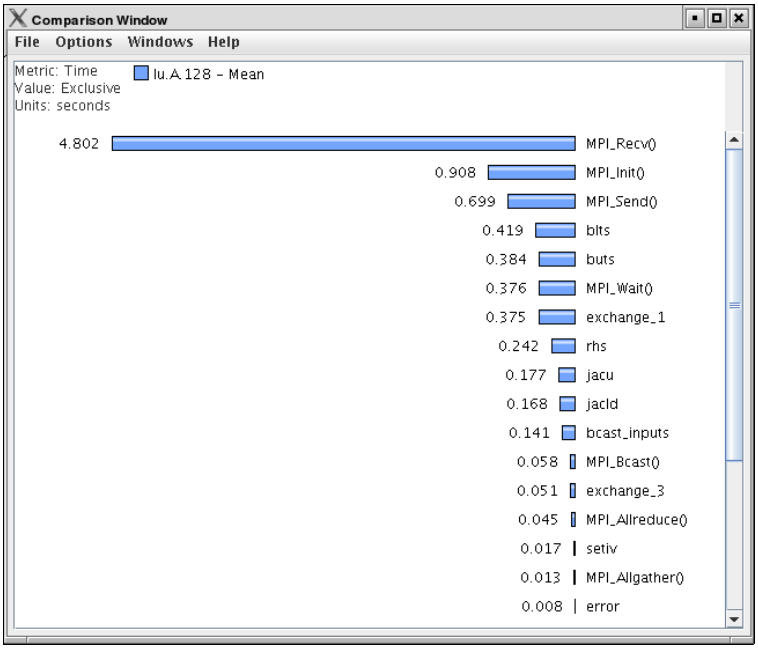
Chapter 15. Comparative Analysis

ParaProf can perform cross-thread and cross-trial anaylsis. In this way, you can compare two or more trials and/or threads in a single display.

15.1. Using Comparitive Analysis

Comparative analysis in ParaProf is based on individual threads of execution. There is a maximum of one Comparison window for a given ParaProf session. To add threads to the window, right click on them and select "Add Thread to Comparison Window". The Comparison Window will pop up with the thread selected. Note that "mean" and "std. dev." are considered threads for this any most other purposes.

Figure 15.1. Comparison Window (initial)



Add additional threads, from any trial, by the same means.

Figure 15.2. Comparison Window (2 trials)

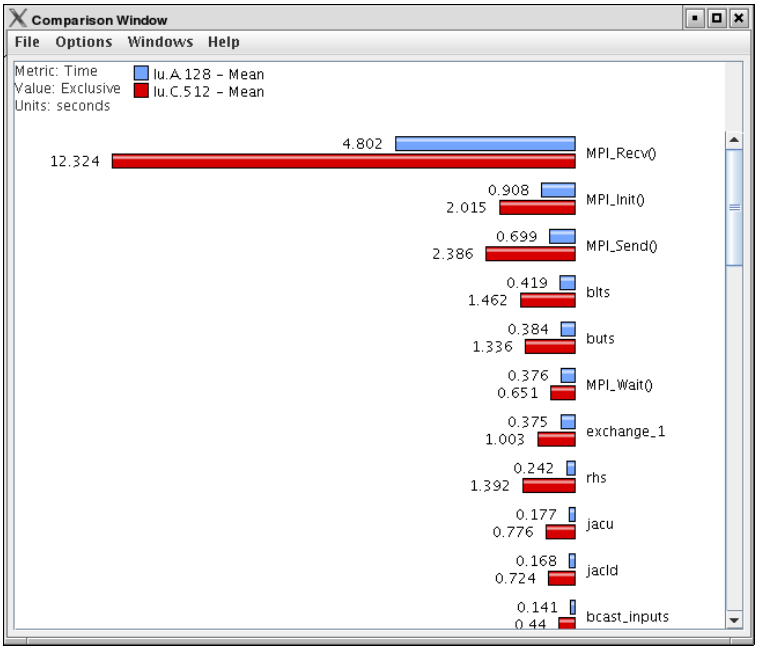
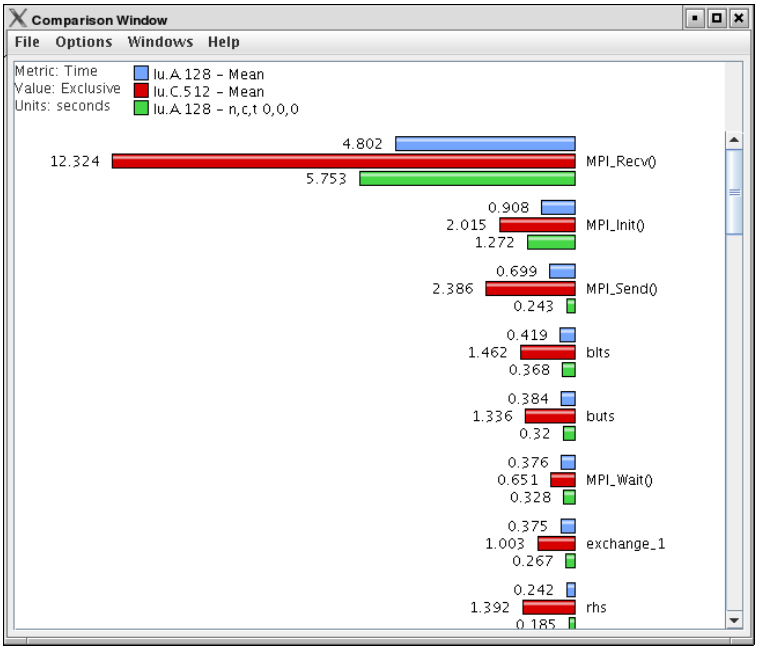


Figure 15.3. Comparison Window (3 threads)

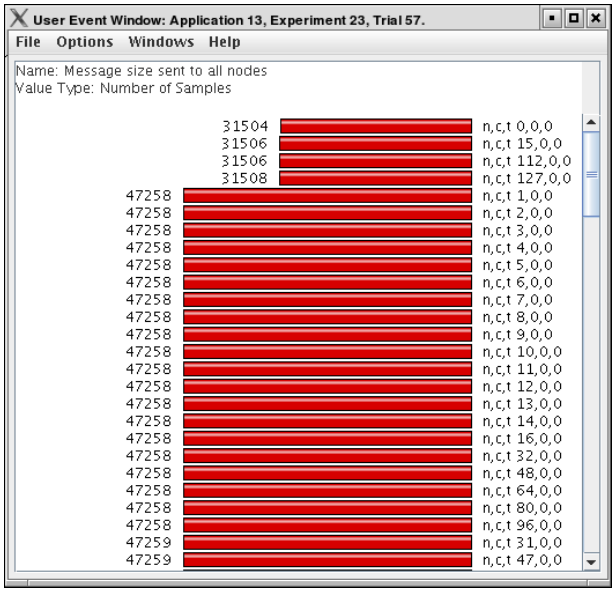


Chapter 16. Miscellaneous Displays

16.1. User Event Bar Graph

In addition to displaying the text statistics for User Defined Events, ParaProf can also graph a particular User Event across all threads.

Figure 16.1. User Event Bar Graph



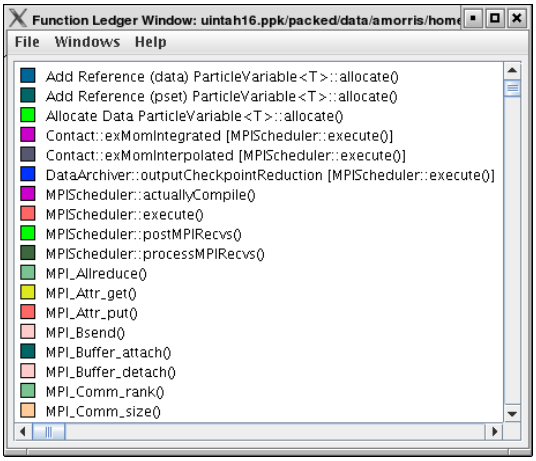
This display graphs the value that the particular user event had for each thread.

16.2. Ledgers

ParaProf has three ledgers that show the functions, groups, and user events.

16.2.1. Function Ledger

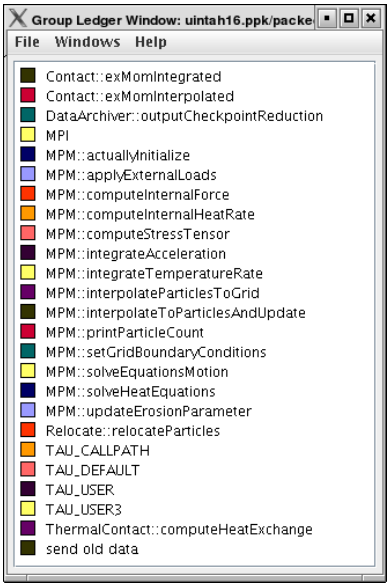
Figure 16.2. Function Ledger



The function ledger shows each function along with its current color. As with other displays showing functions, you may right-click on a function to launch other function-specific displays.

16.2.2. Group Ledger

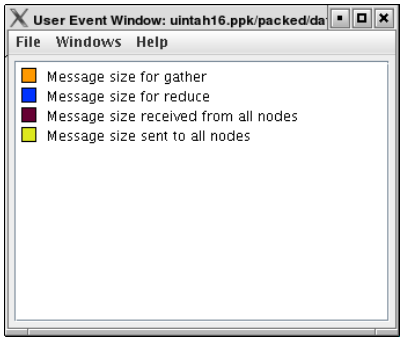
Figure 16.3. Group Ledger



The group ledger shows each group along with its current color. This ledger is especially important because it gives you the ability to mask all of the other displays based on group membership. For example, you can right-click on the `MPI` group and select "Show This Group Only" and all of the windows will now mask to only those functions which are members of the `MPI` group. You may also mask by the inverse by selecting "Show All Groups Except This One" to mask out a particular group.

16.2.3. User Event Ledger

Figure 16.4. User Event Ledger



The user event ledger shows each user event along with its current color.

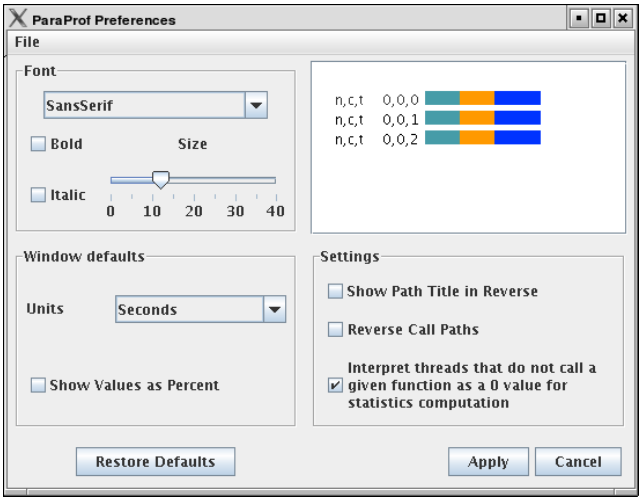
Chapter 17. Preferences

Preferences are modified from the ParaProf Preferences Window, launched from the File menu. Preferences are saved between sessions in the `.ParaProf/ParaProf.prefs`

17.1. Preferences Window

In addition to displaying the text statistics for User Defined Events, ParaProf can also graph a particular User Event across all threads.

Figure 17.1. ParaProf Preferences Window



The preferences window allows the user to modify the behavior and display style of ParaProf's windows. The font size affects bar height, a sample display is shown in the upper-right.

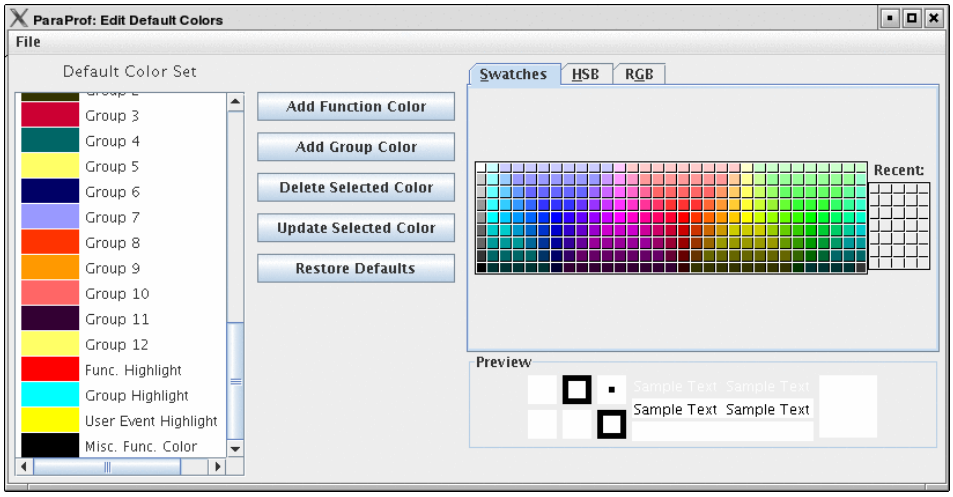
The Window defaults section will determine the initial settings for new windows. You may change the initial units selection and whether you want values displayed as percentages or as raw values.

The Settings section controls the following

- Show Path Title in Reverse - Path title will normally be shown in normal order (/home/amorris/data/etc). They can be reverse using this option (etc/data/amorris/home). This only affects loaded trials and the titlebars of new windows.
- Reverse Call Paths - This option will immediately change the display of all callpath functions between `Root => Leaf` and `Leaf <= Root`.
- Statistics Computation - Turning this option on causes the mean computation to take the sum of value for a function across all threads and divide it by the total number of threads. With this option off the sum will only be divided by the number of threads that actively participated in the sum. This way the user can control whether or not threads which do not call a particular function are consider as a `0` in the computation of statistics.

17.2. Default Colors

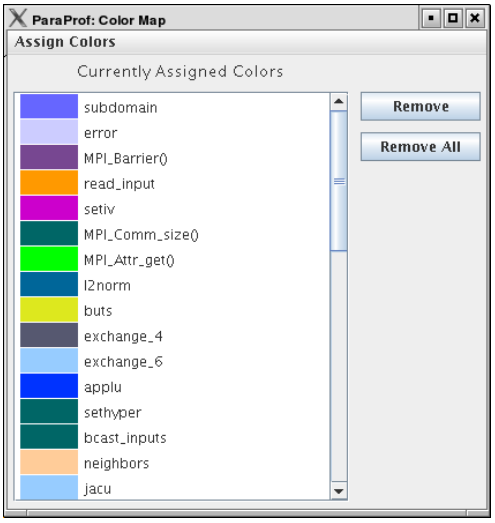
Figure 17.2. Edit Default Colors



The default color editor changes how colors are distributed to functions whose color has not been specifically assigned. It is accessible from the File menu of the Preferences Window.

17.3. Color Map

Figure 17.3. Color Map



The color map shows specifically assigned colors. These values are used across all trials loaded so that the user can identify a particular function across multiple trials. In order to map an entire trial's function set, Select "Assign Defaults from ->" and select a loaded trial.

Individual functions can be assigned a particular color by clicking on them in any of the other ParaProf

Windows.

Part III. PerfDMF

Table of Contents

Chapter 18. Introduction

PerfDMF (Performance Data Management Framework) is a an API/Toolkit that sits atop a DBMS to manage and analyze performance data. The API is available in its native Java form as well as C.

18.1. Prerequisites

1. A supported database (currently, PostgreSQL, MySQL, or Oracle).
2. Java 1.4.

18.2. Installation

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

1. Create a database. This step will depend on the user's chosen database.
   - **PostgreSQL:**

     ```
     $ createdb -O perfdmf perfdmf
     ```

     Or, from **psql**

     ```
     psql=# create database perfdmf with owner = perfdmf;
     ```

   - **MySQL:** From the MySQL prompt

     ```
     mysql> create database perfdmf;
     ```

   - **Oracle:** It is recommended that you create a tablespace for perfdmf:

     ```
     create tablespace perfdmf
     datafile '/path/to/somewhere' size 500m reuse;
     ```

     Then, create a user that has this tablespace as default:

     ```
     create user amorris identified by db;
     grant create session to amorris;
     grant create table to amorris;
     grant create sequence to amorris;
     grant create trigger to amorris;
     alter user amorris quota unlimited on perfdmf;
     alter user amorris default tablespace perfdmf;
     ```

     PerfDMF is set up to use the Oracle Thin Java driver. You will have to obtain this jar file for your database. In our case, it was ojdbc14.jar
2. Configure PerfDMF. To configure PerfDMF, run the **perfdmf_configure** from the TAU bin directory.

The configuration program will prompt the user for several values. The default values will work for most users. When configuration is complete, it will connect to the database and test the configuration. If the configuration is valid and the schema is not found (as will be the case on initial configuration), the schema will be uploaded. Be sure to specify the correct schema for your database.

Chapter 19. Using PerfDMF

The easiest way to interact with PerfDMF is to use ParaProf which provides a GUI interface to all of the database information. In addition, the following commandline utilities are provided.

19.1. perfdmf_createapp

This utility creates applications with a given name

```
$ perfdmf_createapp -n "New Application"
Created Application, ID: 24
```

19.2. perfdmf_createapp

This utility creates experiments with a given name, under a specified application

```
$ perfdmf_createexp -a 24 -n "New Experiment"
Created Experiment, ID: 38
```

19.3. perfdmf_loadtrial

This utility uploads a trial to the database with a given name, under a specified experiment

```
Usage: perfdmf_loadtrial -e <experiment id> -n <name>
                                    [options] <files>

Required Arguments:

  -e, --experimentid <number>    Specify associated experiment
                                       ID for the trial
  -n, --name <text>              Specify the name of the trial

Optional Arguments:

  -f, --filetype <filetype>   Specify type of performance data,
                               options are: profiles (default), pprof,
                               dynaprof, mpip, gprof, psrun, hpm,
                               packed, cube, hpc
  -t, --trialid <number>         Specify trial ID
  -i, --fixnames                 Use the fixnames option for gprof

Notes:
  For the TAU profiles type, you can specify either a specific set of
  profile files on the commandline, or you can specify a directory
  (by default the current directory).  The specified directory will
  be searched for profile.*.*.* files, or, in the case of
  multiple counters, directories named MULTI_* containing profile data.

Examples:

  perfdmf_loadtrial -e 12 -n "Batch 001"
    This will load profile.* (or multiple counters directories MULTI_*)
    into experiment 12 and give the trial the name "Batch 001"

  perfdmf_loadtrial -e 12 -n "HPM data 01" perfhpm*
```

```
This will load perfhpm* files of type HPMToolkit into experiment
12 and give the trial the name "HPM data 01"
```

Part IV. PerfExplorer

Table of Contents

Chapter 20. Introduction

PerfExplorer is a framework for parallel performance data mining and knowledge discovery. The framework architecture enables the development and integration of data mining operations that will be applied to large-scale parallel performance profiles.

The overall goal of the PerfExplorer project is to create a software to integrate sophisticated data mining techniques in the analysis of large-scale parallel performance data.

PerfExplorer supports clustering, summarization, association, regression, and correlation. Cluster analysis is the process of organizing data points into logically similar groupings, called clusters. Summarization is the process of describing the similarities within, and dissimilarities between, the discovered clusters. Association is the process of finding relationships in the data. One such method of association is regression analysis, the process of finding independent and dependent correlated variables in the data. In addition, comparative analysis extends these operations to compare results from different experiments, for instance, as part of a parametric study.

In addition to the data mining operations available, the user may optionally choose to perform comparative analysis. The types of charts available include time-steps per second, relative efficiency and speedup of the entire application, relative efficiency and speedup of one event, relative efficiency and speedup for all events, relative efficiency and speedup for all phases and runtime breakdown of the application by event or by phase. In addition, when the events are grouped together, such as in the case of communication routines, yet another chart shows the percentage of total runtime spent in that group of events. These analyses can be conducted across different combinations of parallel profiles and across phases within an execution.

Chapter 21. Installation and Configuration

PerfExplorer uses PerfDMF databases so if you have not already you will need to install PerfDMF, see Chapter 18, *Introduction*. To configure PerfExplorer move to the `tools/src/PerfExplorer/` directory in you TAU distribution. Type:

```
%>./configure
```

If you haven't already done so for other TAU tools, add `[path to tau]/tau2/apple/bin` to your path.

The following command-line options are available to configure:

21.1. Available configuration options

- `-engine=<analysis engine>`

  Specifies the data-mining engine to use. The supported options include weka and R.

- `-rroot=<directory>`

  Specifies the directory where R is installed. Specifically, it should be the directory where the `bin`, `include`, `lib`, `library` and `share` directories are located.

- `-objectport=<available network port>`

  Specifies the port that the PerfExplorer server should use, when running PerfExplorer in client-server mode. Select an available network port, and make sure that other appropriate network configurations are made (firewalls, etc.). The default port is 9999.

- `-registryport=<available network port>`

  Specifies the port that the rmiregistry should use, when ruining PerfExplorer in client-server mode. Select an available network port, and make sure that other appropriate network configurations are made (firewalls, etc.). The default port is 1099.

- `-server=<server name>`

  Specifies the fully qualified domain name of the server where PerfExplorer is run, when running PerfExplorer in client-server mode.

Chapter 22. Running PerfExplorer

To run PerfExplorer type:

```
%>perfexplorer
```

When PerfExplorer loads you will see on the left window all the experiments that where loaded into PerfDMF. You can select which performance data you are interested by navigating the tree structure. PerfExplorer will allow you to run analysis operations on these experiments. Also the cluster analysis results are visible on the right side of the window. Various types of comparative analysis are available from the drop down menu selected.

To run an analysis operation, first select the metric of interest form the experiments on the left. Then perform the operation by selecting it from the `Analysis` menu. If you would like you can set the `clustering method, dimension reduction, normalization method and the number of clusters` from the same menu.

The options under the Charts menu provide analysis over an entire trial. To view these charts first choose a metric of interest by selecting a trial form the tree on the left. Then choose the `Set Metric of Interest` or `Set Event of Interest` form the `Charts` menu. Now you can view a chart by selecting it from the `Charts` menu.

Chapter 23. Cluster Analysis

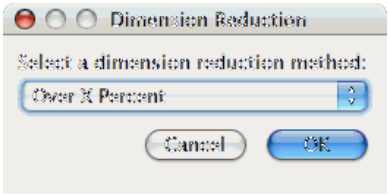
Cluster analysis is a valuable tool for reducing large parallel profiles down to representative groups for investigation. Currently, there are two types of clustering analysis implemented in PerfExplorer. Both *hierarchical* and *k-means* analysis are used to group parallel profiles into common clusters, and then the clusters are summarized. Initially, we used similarity measures computed on a single parallel profile as input to the clustering algorithms, although other forms of input are possible. Here, the performance data is organized into multi-dimensional vectors for analysis. Each vector represents one parallel thread (or process) of execution in the profile. Each dimension in the vector represents an event that was profiled in the application. Events can be any sub-region of code, including libraries, functions, loops, basic blocks or even individual lines of code. In simple clustering examples, each vector represents only one metric of measurement. For our purposes, some dissimilarity value, such as *Euclidean* or *Manhattan* distance, is computed on the vectors. As discussed later, we have tested hierarchical and k-means cluster analysis in PerfExplorer on profiles with over 32K threads of execution with few difficulties.

23.1. Dimension Reduction

Often, many hundreds of events are instrumented when profile data is collected. Clustering works best with dimensions less than 10, so dimension reduction is often necessary to get meaningful results. Currently, there is only one type of dimension reduction available in PerfExplorer. To reduce dimensions, the user specifies a minimum exclusive percentage for an event to be considered "significant".

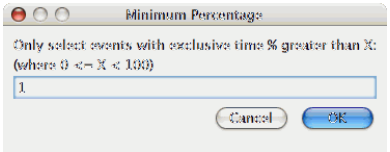
To reduce dimensions, select the "Select Dimension Reduction" item under the "Analysis" main menu bar item. The following dialog will appear:

Figure 23.1. Selecting a dimension reduction method



Select "Over X Percent". The following dialog will appear:

Figure 23.2. Entering a minimum threshold for exclusive percentage



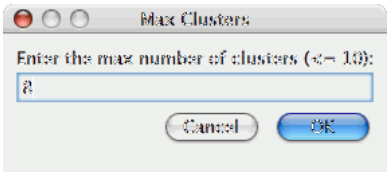
Enter a value, for example "1".

23.2. Max Number of Clusters

By default, PerfExplorer will attempt k-means clustering with values of k from 2 to 10. To change the maximum number of clusters, select the "Set Maximum Number of Clusters" item under the "Analysis"

main menu item. The following dialog will appear:

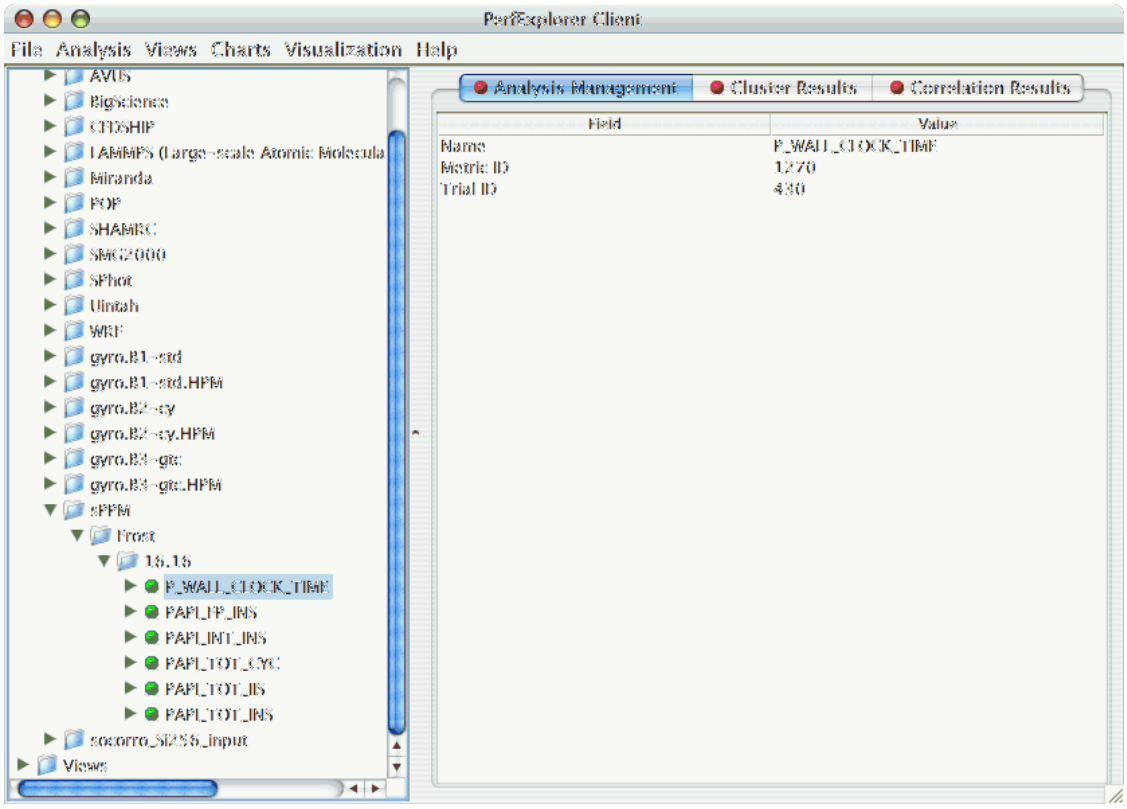
Figure 23.3. Entering a maximum number of clusters



23.3. Performing Cluster Analysis

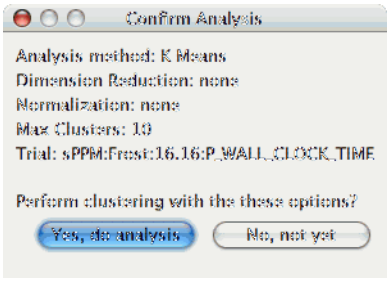
To perform cluster analysis, you first need to select a metric. To select a metric, navigate through the tree of applications, experiments and trials, and expand the trial of interest, showing the available metrics, as shown in the figure below:

Figure 23.4. Selecting a Metric to Cluster



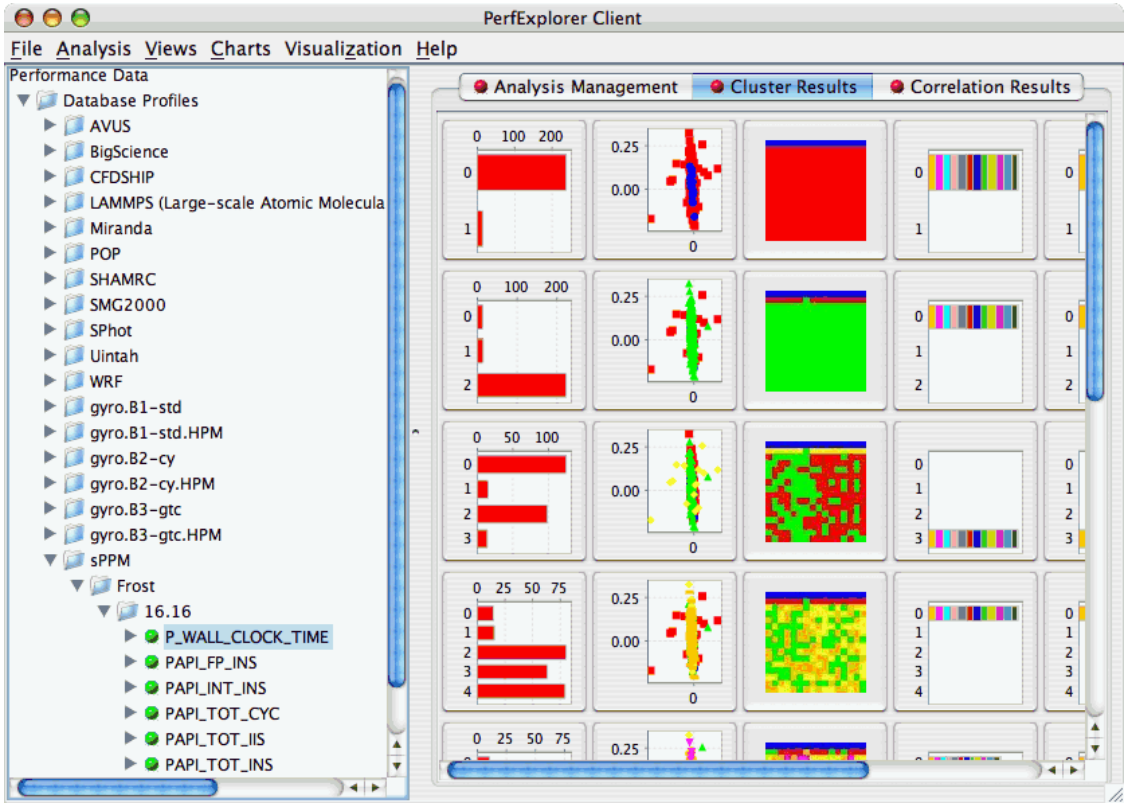
After selecting the metric of interest, select the "Do Clustering" item under the "Analysis" main menu bar item. The following dialog will appear:

Figure 23.5. Confirm Clustering Options



After confirming the clustering, the clustering will begin. When the clustering results are available, you can view them in the "Cluster Results" tab.

Figure 23.6. Cluster Results



There are a number of images in the "Cluster Results" window. From left to right, the windows indicate the cluster membership histogram, a PCA scatterplot showing the cluster memberships, a virtual topology of the parallel machine, the average values for each event in each cluster, the maximum values for each event in each cluster, and the minimum values for each event in each cluster. Clicking on a thumbnail image in the main window will bring up the images, as shown below:

Figure 23.7. Cluster Membership Histogram

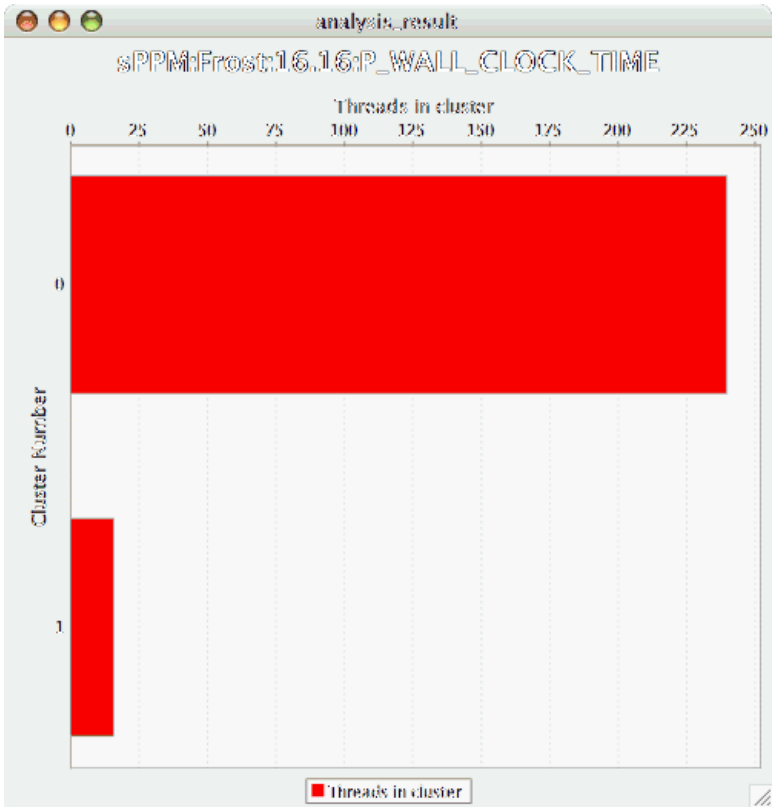


Figure 23.8. Cluster Membership Scatterplot

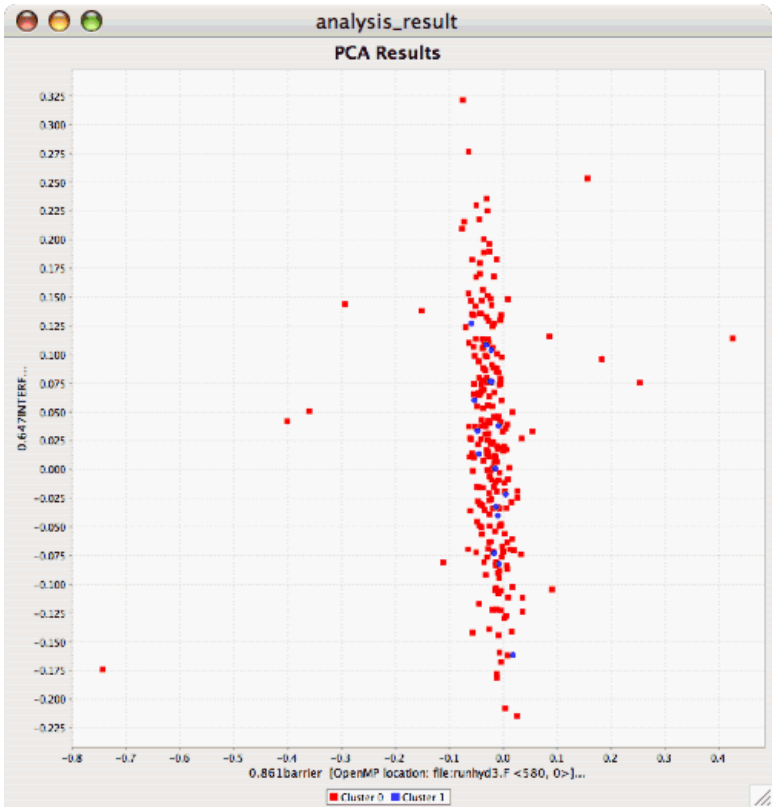


Figure 23.9. Cluster Virtual Topology

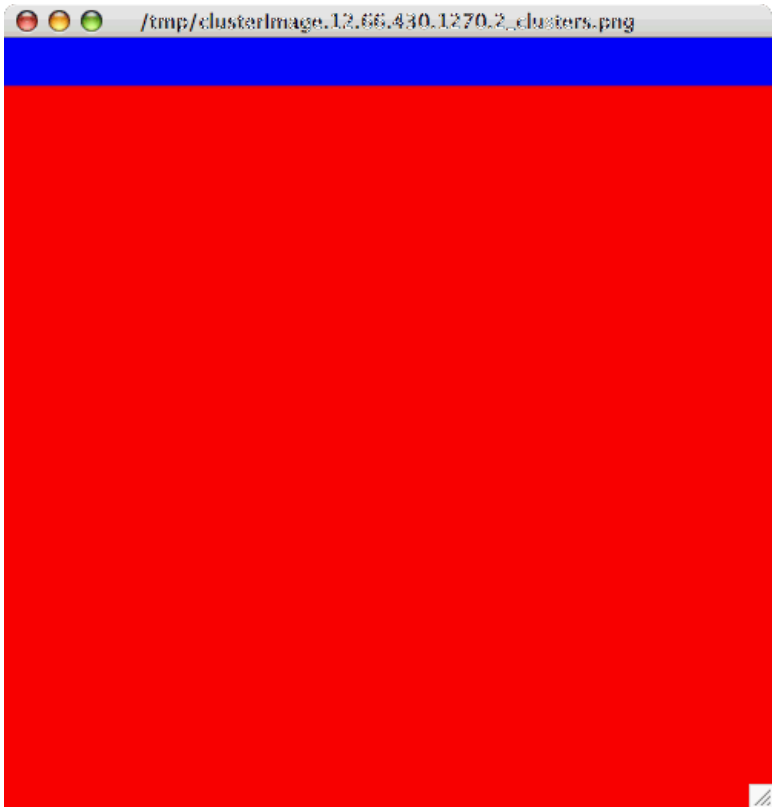
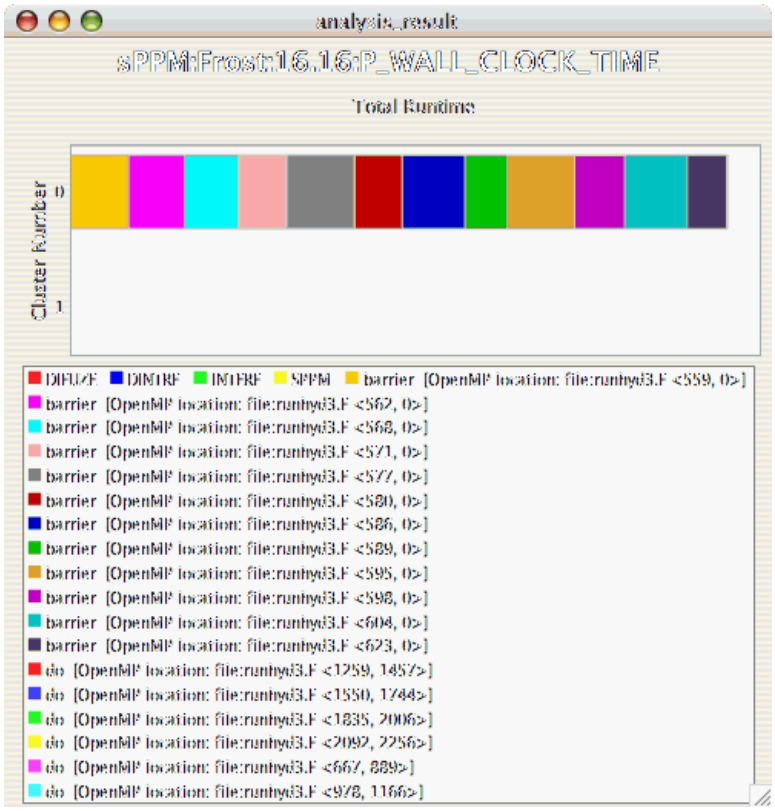


Figure 23.10. Cluster Average Behavior


analysis_result
sPPM:Frost:16.16:P_WALL_CLOCK_TIME
Total Runtime
Cluster Number
0
1
DIFUZE
DIMTRF
IMTRF
SPPM
barrier [OpenMP location: file:runtyd3.F <559, 0>]
barrier [OpenMP location: file:runtyd3.F <562, 0>]
barrier [OpenMP location: file:runtyd3.F <568, 0>]
barrier [OpenMP location: file:runtyd3.F <571, 0>]
barrier [OpenMP location: file:runtyd3.F <577, 0>]
barrier [OpenMP location: file:runtyd3.F <580, 0>]
barrier [OpenMP location: file:runtyd3.F <586, 0>]
barrier [OpenMP location: file:runtyd3.F <589, 0>]
barrier [OpenMP location: file:runtyd3.F <595, 0>]
barrier [OpenMP location: file:runtyd3.F <598, 0>]
barrier [OpenMP location: file:runtyd3.F <604, 0>]
barrier [OpenMP location: file:runtyd3.F <623, 0>]
do [OpenMP location: file:runtyd3.F <1259, 1457>]
do [OpenMP location: file:runtyd3.F <1550, 1748>]
do [OpenMP location: file:runtyd3.F <1835, 2006>]
do [OpenMP location: file:runtyd3.F <2092, 2256>]
do [OpenMP location: file:runtyd3.F <667, 889>]
do [OpenMP location: file:runtyd3.F <978, 1166>]

Chapter 24. Charts

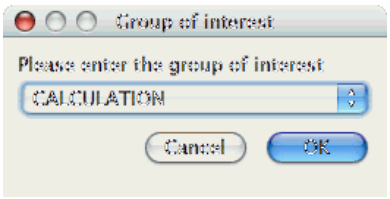
24.1. Setting Parameters

There are a few parameters which need to be set when doing comparisons between trials in the database. If any necessary setting is not configured before requesting a chart, you will be prompted to set the value. The following settings may be necessary for the various charts available:

24.1.1. Group of Interest

TAU events are often associated with common groups, such as "MPI", "TRANSPOSE", etc. This value is used for showing what fraction of runtime that this group of events contributed to the total runtime.

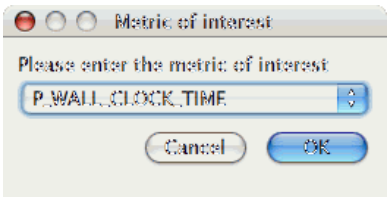
Figure 24.1. Setting Group of Interest



24.1.2. Metric of Interest

Profiles may contain many metrics gathered for a single trial. This selects which of the available metrics the user is interested in.

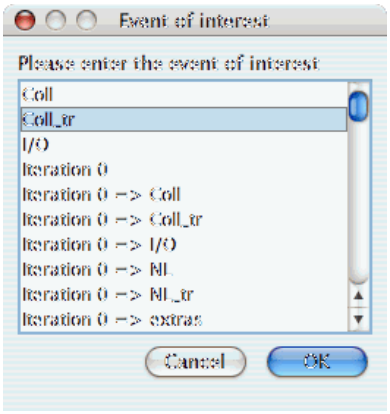
Figure 24.2. Setting Metric of Interest



24.1.3. Event of Interest

Some charts examine events in isolation. This setting configures which event to examine.

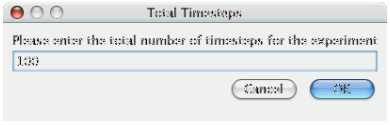
Figure 24.3. Setting Event of Interest



24.1.4. Total Number of Timesteps

One chart, the "Timesteps per second" chart, will calculate the number of timesteps completed per second. This setting configures that value.

Figure 24.4. Setting Timesteps

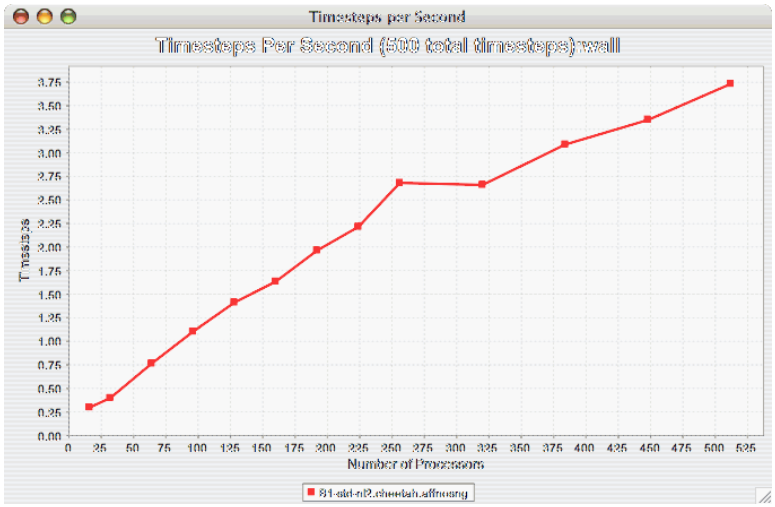


24.2. Standard Chart Types

24.2.1. Timesteps Per Second

The Timesteps Per Second chart shows how an application scales as it relates to time-to-solution. If the timesteps are not already set, you will be prompted to enter the total number of timesteps in the trial (see Section 24.1.4, “Total Number of Timesteps”). If there is more than one metric to choose from, you may be prompted to select the metric of interest (see Section 24.1.2, “Metric of Interest”). To request this chart, select one or more experiments or one view, and select this chart item under the "Charts" main menu item.

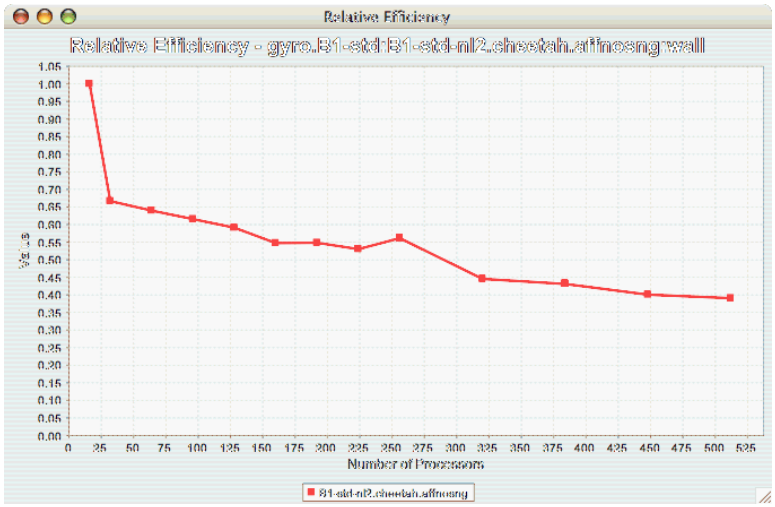
Figure 24.5. Timesteps per Second



24.2.2. Relative Efficiency

The Relative Efficiency chart shows how an application scales with respect to relative efficiency. That is, as the number of processors increases by a factor, the time to solution is expected to decrease by the same factor (with ideal scaling). The fraction between the expected scaling and the actual scaling is the relative efficiency. If there is more than one metric to choose from, you may be prompted to select the metric of interest (see Section 24.1.2, “Metric of Interest”). To request this chart, select one experiment or view, and select this chart item under the "Charts" main menu item.

Figure 24.6. Relative Efficiency

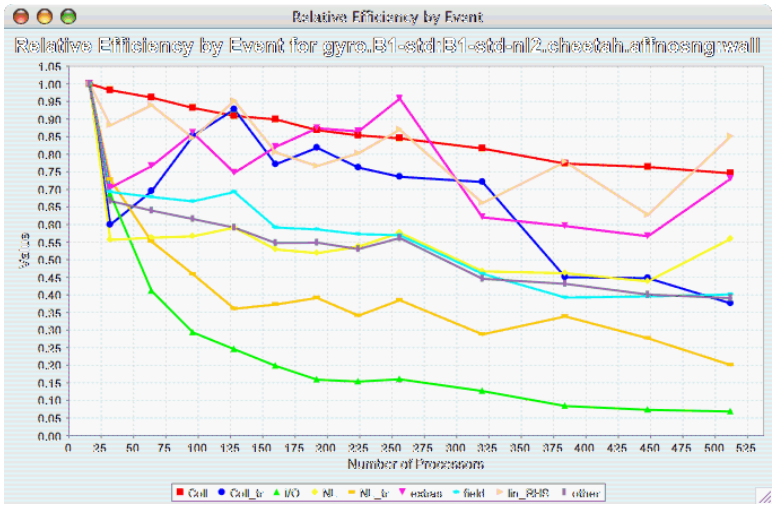


24.2.3. Relative Efficiency by Event

The Relative Efficiency By Event chart shows how each event in an application scales with respect to relative efficiency. That is, as the number of processors increases by a factor, the time to solution is expected to decrease by the same factor (with ideal scaling). The fraction between the expected scaling and the actual scaling is the relative efficiency. If there is more than one metric to choose from, you may be prompted to select the metric of interest (see Section 24.1.2, “Metric of Interest”). To request this chart,

select one or more experiments or one view, and select this chart item under the "Charts" main menu item.

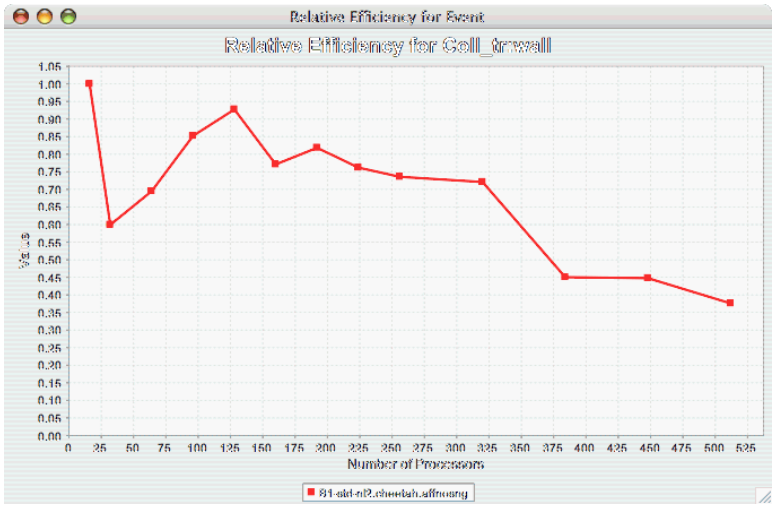
Figure 24.7. Relative Efficiency by Event



24.2.4. Relative Efficiency for One Event

The Relative Efficiency for One Event chart shows how one event from an application scales with respect to relative efficiency. That is, as the number of processors increases by a factor, the time to solution is expected to decrease by the same factor (with ideal scaling). The fraction between the expected scaling and the actual scaling is the relative efficiency. If there is more than one event to choose from, and you have not yet selected an event of interest, you may be prompted to select the event of interest (see Section 24.1.3, “Event of Interest”). If there is more than one metric to choose from, you may be prompted to select the metric of interest (see Section 24.1.2, “Metric of Interest”). To request this chart, select one or more experiments or one view, and select this chart item under the "Charts" main menu item.

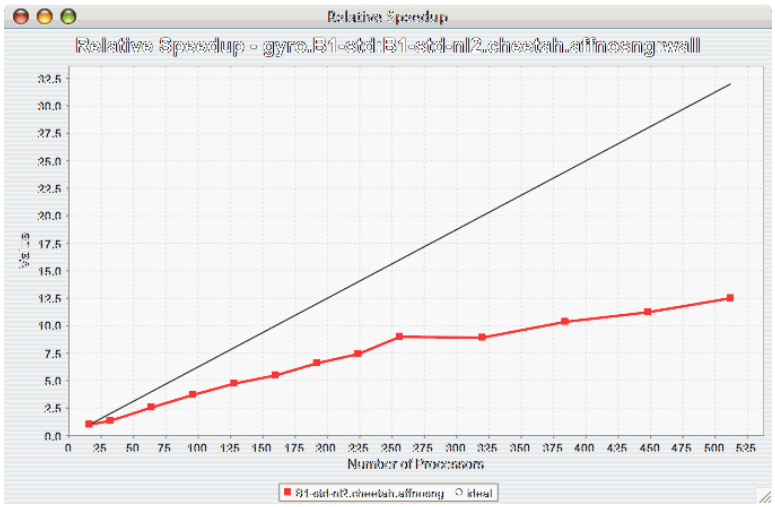
Figure 24.8. Relative Efficiency one Event



24.2.5. Relative Speedup

The Relative Speedup chart shows how an application scales with respect to relative speedup. That is, as the number of processors increases by a factor, the speedup is expected to increase by the same factor (with ideal scaling). The ideal speedup is charted, along with the actual speedup for the application. If there is more than one metric to choose from, you may be prompted to select the metric of interest (see Section 24.1.2, “Metric of Interest”). To request this chart, select one or more experiments or one view, and select this chart item under the "Charts" main menu item.

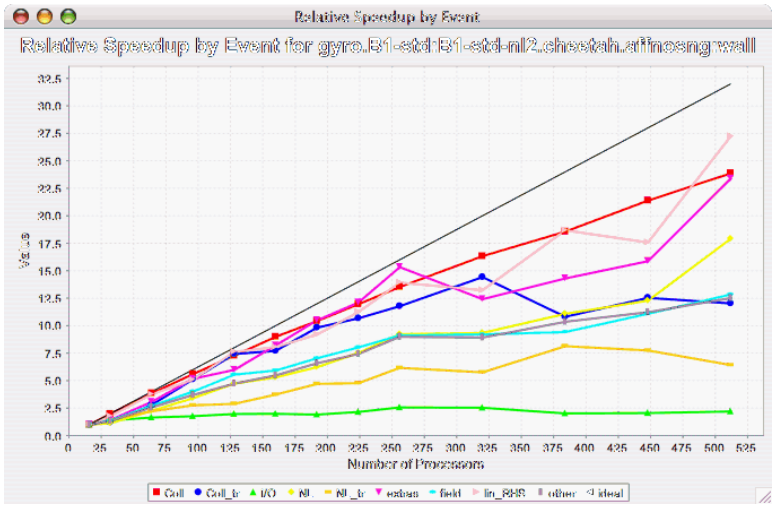
Figure 24.9. Relative Speedup



24.2.6. Relative Speedup by Event

The Relative Speedup By Event chart shows how the events in an application scale with respect to relative speedup. That is, as the number of processors increases by a factor, the speedup is expected to increase by the same factor (with ideal scaling). The ideal speedup is charted, along with the actual speedup for the application. If there is more than one metric to choose from, you may be prompted to select the metric of interest (see Section 24.1.2, “Metric of Interest”). To request this chart, select one experiment or view, and select this chart item under the "Charts" main menu item.

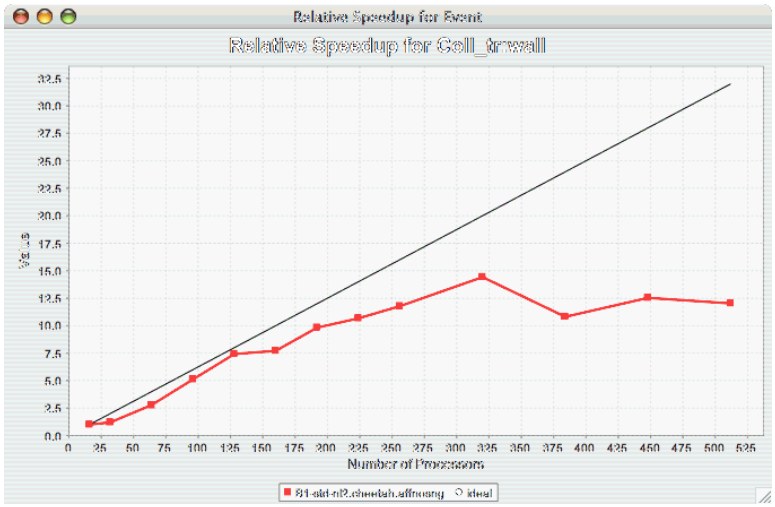
Figure 24.10. Relative Speedup by Event



24.2.7. Relative Speedup for One Event

The Relative Speedup for One Event chart shows how one event in an application scales with respect to relative speedup. That is, as the number of processors increases by a factor, the speedup is expected to increase by the same factor (with ideal scaling). The ideal speedup is charted, along with the actual speedup for the application. If there is more than one event to choose from, and you have not yet selected an event of interest, you may be prompted to select the event of interest (see Section 24.1.3, “Event of Interest”). If there is more than one metric to choose from, you may be prompted to select the metric of interest (see Section 24.1.2, “Metric of Interest”). To request this chart, select one or more experiments or one view, and select this chart item under the "Charts" main menu item.

Figure 24.11. Relative Speedup one Event

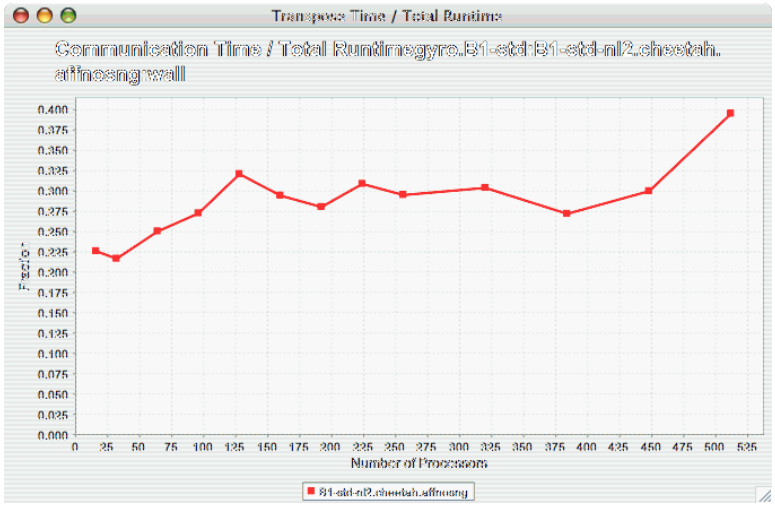


24.2.8. Group % of Total Runtime

The Group % of Total Runtime chart shows how the fraction of the total runtime for one group of events changes as the number of processors increases. If there is more than one group to choose from, and you have not yet selected a group of interest, you may be prompted to select the group of interest (see Sec-

tion 24.1.1, “Group of Interest”). If there is more than one metric to choose from, you may be prompted to select the metric of interest (see Section 24.1.2, “Metric of Interest”). To request this chart, select one or more experiments or one view, and select this chart item under the "Charts" main menu item.

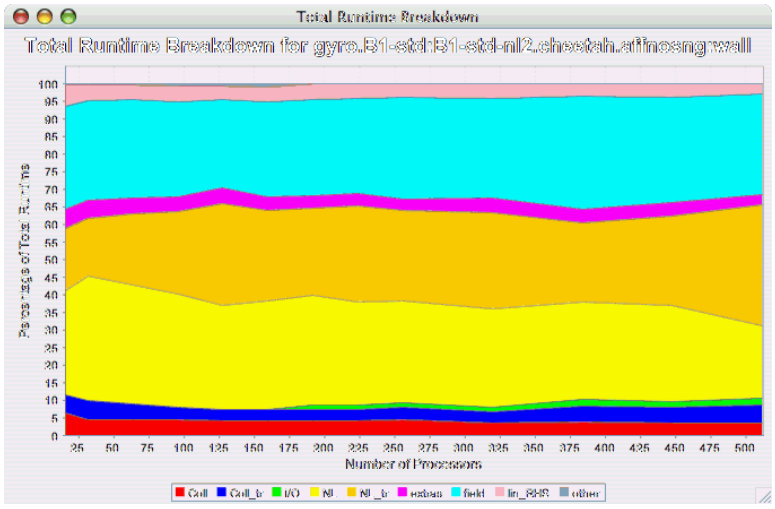
Figure 24.12. Group % of Total Runtime



24.2.9. Runtime Breakdown

The Runtime Breakdown chart shows the fraction of the total runtime for all events in the application, and how the fraction changes as the number of processors increases. If there is more than one metric to choose from, you may be prompted to select the metric of interest (see Section 24.1.2, “Metric of Interest”). To request this chart, select one experiment or view, and select this chart item under the "Charts" main menu item.

Figure 24.13. Runtime Breakdown



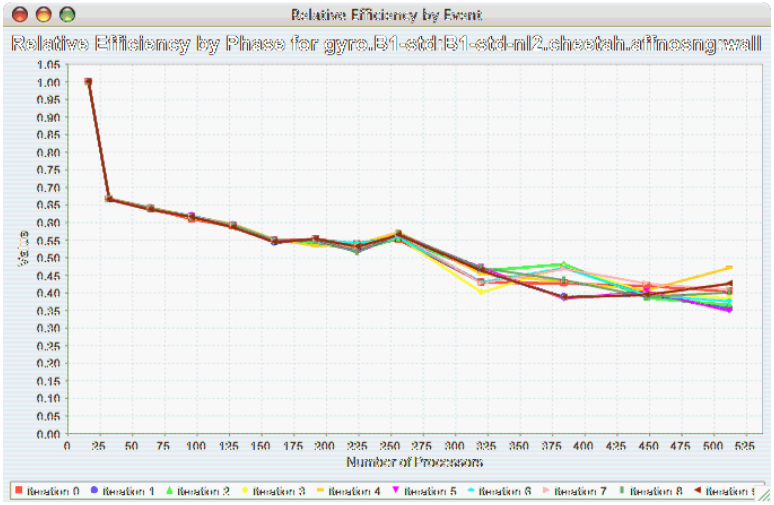
24.3. Phase Chart Types

TAU now provides the ability to break down profiles with respect to phases of execution. One such application would be to collect separate statistics for each timestep, or group of timesteps. In order to visualize the variance between the phases of execution, a number of phase-based charts are available.

24.3.1. Relative Efficiency per Phase

The Relative Efficiency Per Phase chart shows the relative efficiency for each phase, as the number of processors increases. If there is more than one metric to choose from, you may be prompted to select the metric of interest (see Section 24.1.2, “Metric of Interest”). To request this chart, select one experiment or view, and select this chart item under the "Charts" main menu item.

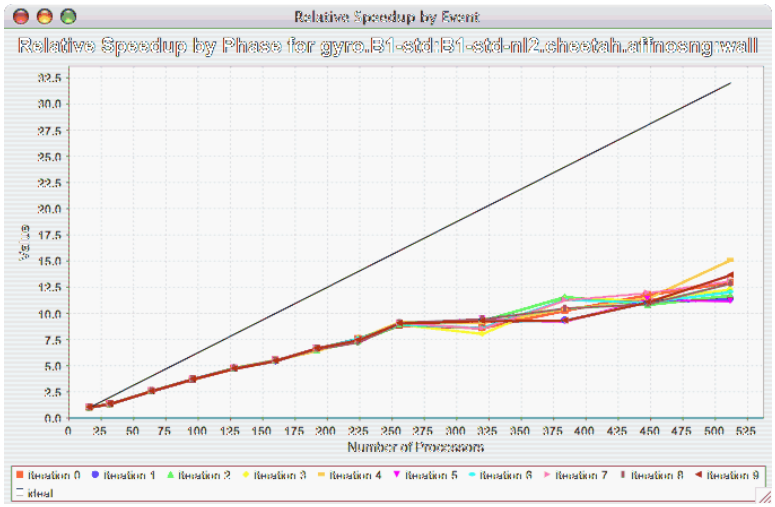
Figure 24.14. Relative Efficiency per Phase



24.3.2. Relative Speedup per Phase

The Relative Speedup Per Phase chart shows the relative speedup for each phase, as the number of processors increases. If there is more than one metric to choose from, you may be prompted to select the metric of interest (see Section 24.1.2, “Metric of Interest”). To request this chart, select one experiment or view, and select this chart item under the "Charts" main menu item.

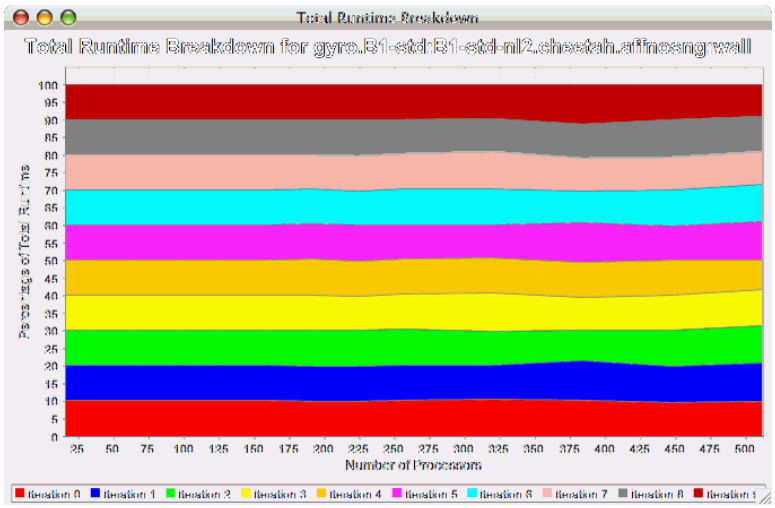
Figure 24.15. Relative Speedup per Phase



24.3.3. Phase Fraction of Total Runtime

The Phase Fraction of Total Runtime chart shows the breakdown of the execution by phases, and shows how that breakdown changes as the number of processors increases. If there is more than one metric to choose from, you may be prompted to select the metric of interest (see Section 24.1.2, “Metric of Interest”). To request this chart, select one experiment or view, and select this chart item under the "Charts" main menu item.

Figure 24.16. Phase Fraction of Total Runtime



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.

1. Software Availability

TAU Portable Profiling and Tracing Toolkit may be downloaded as freeware from the following website TAU [http://www.cs.uoregon.edu/research/tau]:

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

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

```
tau-bugs@cs.uoregon.edu
```

Technical papers about TAU can be downloaded from the TAU Publications homepage at TAU-PUBS [http://www.cs.uoregon.edu/research/tau/pubs.php]

2. Acknowledgments

The TAU development team wishes to thank the U.S. Government, Department of Energy, and the National Science Foundation for their support of the TAU project under the DOE MICS office contracts, University of Utah ASC subcontract, ASC Level 3, and NSF grants.

Part V. appendices

Table of Contents

TAU Instrumentation API

Introduction

- **C++**

  The C++ 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.

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

  ```
  #include <TAU.h>
  ```

- **C**

  The API for instrumenting C source code is similar to the C++ API. The primary 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:

  ```
  #include <TAU.h>

  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;
  }
  ```

- **Fortran 77/90/95**

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

Timers

- **Static timers**

  These are commonly used in most profilers where all invocations of a routine are recorded. The name and group registration takes place when the timer is created (typically the first time a routine is entered). A given timer is started and stopped at routine entry and exit points. A user defined timer can also measure the time spent in a group of statements. Timers may be nested but they may not overlap. The performance data generated can typically answer questions such as: *what is the total time spent in MPI_Send() across all invocations?*

- **Dynamic timers**

  To record the execution of each invocation of a routine, TAU provides dynamic timers where a unique name may be constructed for a dynamic timer for each iteration by embedding the iteration count in it. It uses the start/stop calls around the code to be examined, similar to static timers. The performance data generated can typically answer questions such as:*what is the time spent in the routine foo() in iterations 24, 25, and 40?*

- **Static phases**

  An application typically goes through several phases in its execution. To track the performance of the application based on phases, TAU provides static and dynamic phase profiling. A profile based on phases highlights the context in which a routine is called. An application has a default phase within which other routines and phases are invoked. A phase based profile shows the time spent in a routine when it was in a given phase. So, if a set of instrumented routines are called directly or indirectly by a phase, we'd see the time spent in each of those routines under the given phase. Since phases may be nested, a routine may belong to only one phase. When more than one phase is active for a given routine, the closest ancestor phase of a routine along its callstack is its phase for that invocation. The performance data generated can answer questions such as: *what is the total time spent in MPI_Send() when it was invoked in all invocations of the IO (IO => MPI_Send()) phase?*

- **Dynamic phases**

  Dynamic phases borrow from dynamic timers and static phases to create performance data for all routines that are invoked in a given invocation of a phase. If we instrument a routine as a dynamic phase, creating a unique name for each of its invocations (by embedding the invocation count in the name), we can examine the time spent in all routines and child phases invoked directly or indirectly from the given phase. The performance data generated can typically answer questions such as: *what is the total time spent in MPI_Send() when it was invoked directly or indirectly in iteration 24?* Dynamic phases are useful for tracking per-iteration profiles for an adaptive computation where iterations may differ in their execution times.

- **Callpaths**

  In phase-based profiles, we see the relationship between routines and parent phases. Phase profiles do not show the calling structure between different routines as is represented in a callgraph. To do so, TAU provides callpath profiling capabilities where the time spent in a routine along an edge of a callgraph is captured. Callpath profiles present the full flat profiles of routines (or nodes in the callgraph), as well as routines along a callpath. A callpath is represented syntactically as a list of routines separated by a delimiter. The maximum depth of a callpath is controlled by an environment variable.

- **User-defined Events**

  Besides timers and phases that measure the time spent between a pair of start and stop calls in the code, TAU also provides support for user-defined atomic events. After an event is registered with a

name, it may be triggered with a value at a given point in the source code. At the application level, we can use user-defined events to track the progress of the simulation by keeping track of application specific parameters that explain program dynamics, for example, the number of iterations required for convergence of a solver at each time step, or the number of cells in each iteration of an adaptive mesh refinement application.

Name

TAU_PROFILE -- Profile a C++ function

```
TAU_PROFILE(function_name, type, group);
char* or string& function_name;
char* or string& type;
TauGroup_t group;
```

Description

TAU_PROFILE profiles a function. This macro defines the function and takes care of the timer start and stop as well. The timer will stop when the macro goes out of scope (as in C++ destruction).

Example

```
int foo(char *str) {
  TAU_PROFILE(foo","int (char *)",TAU_DEFAULT);
  ...
}
```

See Also

TAU_PROFILE_TIMER

Name

TAU_PROFILE_TIMER -- Defines a static timer.

C/C++:

```
TAU_PROFILE_TIMER(timer, function_name, type, group);
Profiler timer;
char* or string& function_name;
char* or string& type;
TauGroup_t group;
```

Fortran:

```
TAU_PROFILE_TIMER(profiler, name);
integer profiler(2);
character name(size);
```

Description

C/C++ :

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.

Fortran :

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

Example

C/C++ :

```
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);
 ...
}
```

Fortran :

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

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

interger IERR

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

See Also

TAU_PROFILE_TIMER_DYNAMIC, TAU_PROFILE_START, TAU_PROFILE_STOP

Name

TAU_PROFILE_START -- Starts a timer.

C/C++:

```
TAU_PROFILE_START(timer);
Profiler timer;
```

Fortran:

```
TAU_PROFILE_START(profiler);
integer profiler(2);
```

Description

Starts the timer given by *timer*

Example

C/C++ :

```
int foo(int a) {
  TAU_PROFILE_TIMER(timer, "foo", "int (int)", TAU_USER);
  TAU_PROFILE_START(timer);
  ...
  TAU_PROFILE_STOP(timer);
  return a;
}
```

Fortran :

```
subroutine F1()
  integer profiler(2) / 0, 0 /
  save    profiler

  call TAU_PROFILE_TIMER(profiler,'f1()')
  call TAU_PROFILE_START(profiler)
  ...
  call TAU_PROFILE_STOP(profiler)
end
```

See Also

TAU_PROFILE_TIMER, TAU_PROFILE_STOP

Name

TAU_PROFILE_STOP -- Stops a timer.

C/C++:

```
TAU_PROFILE_STOP(timer);
Profiler timer;
```

Fortran:

```
TAU_PROFILE_STOP(profiler);
integer profiler(2);
```

Description

Stops the timer given by *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/C++ :

```
int foo(int a) {
  TAU_PROFILE_TIMER(timer, "foo", "int (int)", TAU_USER);
  TAU_PROFILE_START(timer);
  ...
  TAU_PROFILE_STOP(timer);
  return a;
}
```

Fortran :

```
subroutine F1()
  integer profiler(2) / 0, 0 /
  save    profiler

  call TAU_PROFILE_TIMER(profiler,'f1()')
  call TAU_PROFILE_START(profiler)
  ...
  call TAU_PROFILE_STOP(profiler)
end
```

See Also

TAU_PROFILE_TIMER, TAU_PROFILE_START

Name

TAU_PROFILE_TIMER_DYNAMIC -- Defines a dynamic timer.

C/C++:

```
TAU_PROFILE_TIMER_DYNAMIC(timer, function_name, type, group);
Profiler timer;
char* or string& function_name;
char* or string& type;
TauGroup_t group;
```

Fortran:

```
TAU_PROFILE_TIMER_DYNAMIC(profiler, name);
integer profiler(2);
character name(size);
```

Description

TAU_PROFILE_TIMER_DYNAMIC operates similar to TAU_PROFILE_TIMER except that the timer is created each time the statement is invoked. This way, the name of the timer can be different for each execution.

Example

C/C++ :

```
int main(int argc, char **argv) {
  int i;
  TAU_PROFILE_TIMER(t,"main()", "", TAU_DEFAULT);
  TAU_PROFILE_SET_NODE(0);
  TAU_PROFILE_START(t);

  for (i=0; i&5; i++) {
    char buf[32];
    sprintf(buf, "Iteration %d", i);

    TAU_PROFILE_TIMER_DYNAMIC(timer, buf, "", TAU_USER);
    TAU_PROFILE_START(timer);
    printf("Iteration %d\n", i);
    f1();

    TAU_PROFILE_STOP(timer);
  }
  return 0;
}
```

Fortran :

```
subroutine ITERATION(val)
  integer val
  character(13) cvar
  integer profiler(2) / 0, 0 /
```

```
  save profiler

  print *, "Iteration ", val

  write (cvar,'(a9,i2)') 'Iteration', val
  call TAU_PROFILE_TIMER_DYNAMIC(profiler, cvar)
  call TAU_PROFILE_START(profiler)

  call F1()
  call TAU_PROFILE_STOP(profiler)
  return
end
```

See Also

TAU_PROFILE_TIMER, TAU_PROFILE_START, TAU_PROFILE_STOP

Name

TAU_PROFILE_DECLARE_TIMER -- Declares a timer for C

C:

```
TAU_PROFILE_DECLARE_TIMER(timer);
Profiler timer;
```

Description

Because C89 does not allow mixed code and declarations, `TAU_PROFILE_TIMER` can only be used once in a function. To declare two timers in a C function, use `TAU_PROFILE_DECLARE_TIMER` and `TAU_PROFILE_CREATE_TIMER`.

Example

C:

```
int f1(void) {
  TAU_PROFILE_DECLARE_TIMER(t1);
  TAU_PROFILE_DECLARE_TIMER(t2);

  TAU_PROFILE_CREATE_TIMER(t1, "timer1", "", TAU_USER);
  TAU_PROFILE_CREATE_TIMER(t2, "timer2", "", TAU_USER);

  TAU_PROFILE_START(t1);
  ...
  TAU_PROFILE_START(t2);
  ...
  TAU_PROFILE_STOP(t2);
  TAU_PROFILE_STOP(t1);
  return 0;
}
```

See Also

TAU_PROFILE_CREATE_TIMER

Name

TAU_PROFILE_CREATE_TIMER -- Creates a timer for C

C:

```
TAU_PROFILE_CREATE_TIMER(timer);
Profiler timer;
```

Description

Because C89 does not allow mixed code and declarations, TAU_PROFILE_TIMER can only be used once in a function. To declare two timers in a C function, use TAU_PROFILE_DECLARE_TIMER and TAU_PROFILE_CREATE_TIMER.

Example

C:

```
int f1(void) {
  TAU_PROFILE_DECLARE_TIMER(t1);
  TAU_PROFILE_DECLARE_TIMER(t2);

  TAU_PROFILE_CREATE_TIMER(t1, "timer1", "", TAU_USER);
  TAU_PROFILE_CREATE_TIMER(t2, "timer2", "", TAU_USER);

  TAU_PROFILE_START(t1);
  ...
  TAU_PROFILE_START(t2);
  ...
  TAU_PROFILE_STOP(t2);
  TAU_PROFILE_STOP(t1);
  return 0;
}
```

See Also

TAU_PROFILE_DECLARE_TIMER, TAU_PROFILE_START, TAU_PROFILE_STOP

Name

TAU_GLOBAL_TIMER -- Declares a global timer

C/C++:

```
TAU_GLOBAL_TIMER(timer, function_name, type, group);
Profiler timer;
char* or string& function_name;
char* or string& type;
TauGroup_t group;
```

Description

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

Example

C/C++:

```
/* f1.c */

TAU_GLOBAL_TIMER(globalTimer, "global timer", "", TAU_USER);

/* f2.c */

TAU_GLOBAL_TIMER_EXTERNAL(globalTimer);
int foo(void) {
  TAU_GLOBAL_TIMER_START(globalTimer);
  /* ... */
  TAU_GLOBAL_TIMER_STOP();
}
```

See Also

TAU_GLOBAL_TIMER_EXTERNAL, TAU_GLOBAL_TIMER_START, TAU_GLOBAL_TIMER_STOP

Name

TAU_GLOBAL_TIMER_EXTERNAL -- Declares a global timer from an external compilation unit

C/C++:

```
TAU_GLOBAL_TIMER_EXTERNAL(timer);
Profiler timer;
```

Description

TAU_GLOBAL_TIMER_EXTERNAL allows you to access a timer defined in another compilation unit.

Example

C/C++:

```
/* f1.c */

TAU_GLOBAL_TIMER(globalTimer, "global timer", "", TAU_USER);

/* f2.c */

TAU_GLOBAL_TIMER_EXTERNAL(globalTimer);
int foo(void) {
  TAU_GLOBAL_TIMER_START(globalTimer);
  /* ... */
  TAU_GLOBAL_TIMER_STOP();
}
```

See Also

TAU_GLOBAL_TIMER, TAU_GLOBAL_TIMER_START, TAU_GLOBAL_TIMER_STOP

Name

TAU_GLOBAL_TIMER_START -- Starts a global timer

C/C++:

```
TAU_GLOBAL_TIMER_START(timer);
Profiler timer;
```

Description

TAU_GLOBAL_TIMER_START starts a global timer.

Example

C/C++:

```
/* f1.c */

TAU_GLOBAL_TIMER(globalTimer, "global timer", "", TAU_USER);

/* f2.c */

TAU_GLOBAL_TIMER_EXTERNAL(globalTimer);
int foo(void) {
  TAU_GLOBAL_TIMER_START(globalTimer);
  /* ... */
  TAU_GLOBAL_TIMER_STOP();
}
```

See Also

TAU_GLOBAL_TIMER, TAU_GLOBAL_TIMER_EXTERNAL, TAU_GLOBAL_TIMER_STOP

Name

TAU_GLOBAL_TIMER_STOP -- Stops a global timer

C/C++:

TAU_GLOBAL_TIMER_STOP();

Description

TAU_GLOBAL_TIMER_STOP stops a global timer.

Example

C/C++:

```
/* f1.c */

TAU_GLOBAL_TIMER(globalTimer, "global timer", "", TAU_USER);

/* f2.c */

TAU_GLOBAL_TIMER_EXTERNAL(globalTimer);
int foo(void) {
  TAU_GLOBAL_TIMER_START(globalTimer);
  /* ... */
  TAU_GLOBAL_TIMER_STOP();
}
```

See Also

TAU_GLOBAL_TIMER, TAU_GLOBAL_TIMER_EXTERNAL, TAU_GLOBAL_TIMER_START

Name

TAU_PHASE -- Profile a C++ function as a phase

```
TAU_PHASE(function_name, type, group);
char* or string& function_name;
char* or string& type;
TauGroup_t group;
```

Description

TAU_PHASE profiles a function as a phase. This macro defines the function and takes care of the timer start and stop as well. The timer will stop when the macro goes out of scope (as in C++ destruction).

Example

```
int foo(char *str) {
  TAU_PHASE(foo","int (char *)",TAU_DEFAULT);
  ...
}
```

See Also

TAU_PHASE_CREATE_DYNAMIC, TAU_PHASE_CREATE_STATIC

Name

TAU_PHASE_CREATE_DYNAMIC -- Defines a dynamic phase.

C/C++:

```
TAU_PHASE_CREATE_DYNAMIC(phase, function_name, type, group);
Phase phase;
char* or string& function_name;
char* or string& type;
TauGroup_t group;
```

Fortran:

```
TAU_PHASE_CREATE_DYNAMIC(phase, name);
integer phase(2);
character name(size);
```

Description

TAU_PHASE_CREATE_DYNAMIC creates a dynamic phase. The name of the timer can be different for each execution.

Example

C/C++ :

```
int main(int argc, char **argv) {
  int i;
  TAU_PROFILE_TIMER(t,"main()", "", TAU_DEFAULT);
  TAU_PROFILE_SET_NODE(0);
  TAU_PROFILE_START(t);

  for (i=0; i&5; i++) {
    char buf[32];
    sprintf(buf, "Iteration %d", i);

    TAU_PHASE_CREATE_DYNAMIC(timer, buf, "", TAU_USER);
    TAU_PHASE_START(timer);
    printf("Iteration %d\n", i);
    f1();

    TAU_PHASE_STOP(timer);
  }
  return 0;
}
```

Fortran :

```
subroutine ITERATION(val)
  integer val
  character(13) cvar
  integer profiler(2) / 0, 0 /
  save profiler
```

```
  print *, "Iteration ", val

  write (cvar,'(a9,i2)') 'Iteration', val
  call TAU_PHASE_CREATE_DYNAMIC(profiler, cvar)
  call TAU_PHASE_START(profiler)

  call F1()
  call TAU_PHASE_STOP(profiler)
  return
end
```

See Also

TAU_PHASE_CREATE_STATIC, TAU_PHASE_START, TAU_PHASE_STOP

Name

TAU_PHASE_CREATE_STATIC -- Defines a static phase.

C/C++:

```
TAU_PHASE_CREATE_STATIC(phase, function_name, type, group);
Phase phase;
char* or string& function_name;
char* or string& type;
TauGroup_t group;
```

Fortran:

```
TAU_PHASE_CREATE_STATIC(phase, name);
integer phase(2);
character name(size);
```

Description

TAU_PHASE_CREATE_STATIC creates a static phase. Static phases (and timers) are more efficient than dynamic ones because the function registration only takes place once.

Example

C/C++ :

```
int f2(void)
{
  TAU_PHASE_CREATE_STATIC(t2,"IO Phase", "", TAU_USER);
  TAU_PHASE_START(t2);
  input();
  output();
  TAU_PHASE_STOP(t2);
  return 0;
}
```

Fortran :

```
subroutine F2()

  integer phase(2) / 0, 0 /
  save    phase

  call TAU_PHASE_CREATE_STATIC(phase,'IO Phase')
  call TAU_PHASE_START(phase)

  call INPUT()
  call OUTPUT()

  call TAU_PHASE_STOP(phase)
end
```

See Also

TAU_PHASE_CREATE_DYNAMIC, TAU_PHASE_START, TAU_PHASE_STOP

Name

TAU_PHASE_START -- Enters a phase.

C/C++:

```
TAU_PHASE_START(phase);
Phase phase;
```

Fortran:

```
TAU_PHASE_START(phase);
integer phase(2);
```

Description

TAU_PHASE_START enters a phase. Phases can be nested, but not overlapped.

Example

C/C++:

```
int f2(void)
{
  TAU_PHASE_CREATE_STATIC(t2,"IO Phase", "", TAU_USER);
  TAU_PHASE_START(t2);
  input();
  output();
  TAU_PHASE_STOP(t2);
  return 0;
}
```

Fortran :

```
subroutine F2()

  integer phase(2) / 0, 0 /
  save    phase

  call TAU_PHASE_CREATE_STATIC(phase,'IO Phase')
  call TAU_PHASE_START(phase)

  call INPUT()
  call OUTPUT()

  call TAU_PHASE_STOP(phase)
end
```

See Also

TAU_PHASE_CREATE_STATIC, TAU_PHASE_CREATE_DYNAMIC, TAU_PHASE_STOP

Name

TAU_PHASE_STOP -- Exits a phase.

C/C++:

```
TAU_PHASE_STOP(phase);
Phase phase;
```

Fortran:

```
TAU_PHASE_STOP(phase);
integer phase(2);
```

Description

TAU_PHASE_STOP exits a phase. Phases can be nested, but not overlapped.

Example

C/C++ :

```
int f2(void)
{
  TAU_PHASE_CREATE_STATIC(t2,"IO Phase", "", TAU_USER);
  TAU_PHASE_START(t2);
  input();
  output();
  TAU_PHASE_STOP(t2);
  return 0;
}
```

Fortran :

```
subroutine F2()

  integer phase(2) / 0, 0 /
  save    phase

  call TAU_PHASE_CREATE_STATIC(phase,'IO Phase')
  call TAU_PHASE_START(phase)

  call INPUT()
  call OUTPUT()

  call TAU_PHASE_STOP(phase)
end
```

See Also

TAU_PHASE_CREATE_STATIC, TAU_PHASE_CREATE_DYNAMIC, TAU_PHASE_START

Name

TAU_GLOBAL_PHASE -- Declares a global phase

C/C++:

```
TAU_GLOBAL_PHASE(phase, function_name, type, group);
Phase phase;
char* or string& function_name;
char* or string& type;
TauGroup_t group;
```

Description

Declares a global phase to be used in multiple compilation units.

Example

C/C++:

```
/* f1.c */

TAU_GLOBAL_PHASE(globalPhase, "global phase", "", TAU_USER);

/* f2.c */

int bar(void) {
  TAU_GLOBAL_PHASE_START(globalPhase);
  /* ... */
  TAU_GLOBAL_PHASE_STOP(globalPhase);
}
```

See Also

TAU_GLOBAL_PHASE_EXTERNAL, TAU_GLOBAL_PHASE_START, TAU_GLOBAL_PHASE_STOP

Name

TAU_GLOBAL_PHASE_EXTERNAL -- Declares a global phase from an external compilation unit

C/C++:

```
TAU_GLOBAL_PHASE_EXTERNAL(timer);
Profiler timer;
```

Description

TAU_GLOBAL_PHASE_EXTERNAL allows you to access a phase defined in another compilation unit.

Example

C/C++:

```
/* f1.c */

TAU_GLOBAL_PHASE(globalPhase, "global phase", "", TAU_USER);

/* f2.c */

int bar(void) {
  TAU_GLOBAL_PHASE_START(globalPhase);
  /* ... */
  TAU_GLOBAL_PHASE_STOP(globalPhase);
}
```

See Also

TAU_GLOBAL_PHASE, TAU_GLOBAL_PHASE_START, TAU_GLOBAL_PHASE_STOP

Name

TAU_GLOBAL_PHASE_START -- Starts a global phase

C/C++:

```
TAU_GLOBAL_PHASE_START(phase);
Phase phase;
```

Description

TAU_GLOBAL_PHASE_START starts a global phase.

Example

C/C++:

```
/* f1.c */

TAU_GLOBAL_PHASE(globalPhase, "global phase", "", TAU_USER);

/* f2.c */

int bar(void) {
  TAU_GLOBAL_PHASE_START(globalPhase);
  /* ... */
  TAU_GLOBAL_PHASE_STOP(globalPhase);
}
```

See Also

TAU_GLOBAL_PHASE, TAU_GLOBAL_PHASE_EXTERNAL, TAU_GLOBAL_PHASE_STOP

Name

TAU_GLOBAL_PHASE_STOP -- Stops a global phase

C/C++:

```
TAU_GLOBAL_PHASE_STOP(phase);
Phase phase;
```

Description

TAU_GLOBAL_PHASE_STOP stops a global phase.

Example

C/C++:

```
/* f1.c */

TAU_GLOBAL_PHASE(globalPhase, "global phase", "", TAU_USER);

/* f2.c */

int bar(void) {
  TAU_GLOBAL_PHASE_STOP(globalPhase);
  /* ... */
  TAU_GLOBAL_PHASE_STOP(globalPhase);
}
```

See Also

TAU_GLOBAL_PHASE, TAU_GLOBAL_PHASE_EXTERNAL, TAU_GLOBAL_PHASE_START

Name

TAU_PROFILE_EXIT -- Alerts the profiling system to an exit call

C/C++:

```
TAU_PROFILE_EXIT(message);
const char * message;
```

Fortran:

```
TAU_PROFILE_EXIT(message);
character message(size);
```

Description

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/C++ :

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

Fortran :

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

See Also

TAU_DB_DUMP

Name

TAU_REGISTER_THREAD -- Register a thread with the profiling system

C/C++:

```
TAU_REGISTER_THREAD();
```

Fortran:

```
TAU_REGISTER_THREAD();
```

Description

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.

Example

C/C++ :

```
void * threaded_func(void *data) {
  TAU_REGISTER_THREAD();
  { /**** NOTE WE START ANOTHER BLOCK IN THREAD */
    TAU_PROFILE_TIMER(tautimer, "threaded_func()", "int ()",
                      TAU_DEFAULT);
    TAU_PROFILE_START(tautimer);
    work(); /* work done by this thread */
    TAU_PROFILE_STOP(tautimer);
  }
  return NULL;
}
```

Fortran :

```
call TAU_REGISTER_THREAD()
```

Caveat

PDT based tau_instrumentor does not insert `TAU_REGISTER_THREAD` calls, they must be inserted manually

Name

TAU_PROFILE_SET_NODE -- Informs the measurement system of the node id

C/C++:

```
TAU_PROFILE_SET_NODE(node);
int node;
```

Fortran:

```
TAU_PROFILE_SET_NODE(node);
integer node;
```

Description

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>. Note that it is not necessary to call TAU_PROFILE_SET_NODE when using the TAU MPI wrapper library.

Example

C/C++ :

```
int main (int argc, char **argv) {
  int ret, i;
  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);
  /* ... */
  TAU_PROFILE_STOP(tautimer);
  return 0;
}
```

Fortran :

```
     PROGRAM SUM_OF_CUBES
       integer profiler(2) / 0, 0 /
         save profiler
      INTEGER :: H, T, U
        call TAU_PROFILE_INIT()
        call TAU_PROFILE_TIMER(profiler, 'PROGRAM SUM_OF_CUBES')
        call TAU_PROFILE_START(profiler)
        call TAU_PROFILE_SET_NODE(0)
      ! This program prints all 3-digit numbers that
      ! equal the sum of the cubes of their digits.
      DO H = 1, 9
        DO T = 0, 9
          DO U = 0, 9
```

```
      IF (100*H + 10*T + U == H**3 + T**3 + U**3) THEN
         PRINT "(3I1)", H, T, U
      ENDIF
      END DO
    END DO
  END DO
  call TAU_PROFILE_STOP(profiler)
  END PROGRAM SUM_OF_CUBES
```

See Also

TAU_PROFILE_SET_CONTEXT

Name

TAU_PROFILE_SET_CONTEXT -- Informs the measurement system of the context id

C/C++:

```
TAU_PROFILE_SET_CONTEXT(context);
int context;
```

Fortran:

```
TAU_PROFILE_SET_CONTEXT(context);
integer context;
```

Description

The TAU_PROFILE_SET_CONTEXT macro sets the context identifier of the executing task for profiling and tracing. Tasks are identified using context, context and thread ids. The profile data files generated will accordingly be named profile.<context>.<context>.<thread>. Note that it is not necessary to call TAU_PROFILE_SET_CONTEXT when using the TAU MPI wrapper library.

Example

C/C++ :

```
int main (int argc, char **argv) {
  int ret, i;
  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);
  TAU_PROFILE_SET_CONTEXT(1);
  /* ... */
  TAU_PROFILE_STOP(tautimer);
  return 0;
}
```

Fortran :

```
     PROGRAM SUM_OF_CUBES
       integer profiler(2) / 0, 0 /
        save profiler
      INTEGER :: H, T, U
        call TAU_PROFILE_INIT()
        call TAU_PROFILE_TIMER(profiler, 'PROGRAM SUM_OF_CUBES')
        call TAU_PROFILE_START(profiler)
        call TAU_PROFILE_SET_NODE(0)
        call TAU_PROFILE_SET_CONTEXT(1)
      ! This program prints all 3-digit numbers that
      ! equal the sum of the cubes of their digits.
      DO H = 1, 9
```

```
  DO T = 0, 9
    DO U = 0, 9
    IF (100*H + 10*T + U == H**3 + T**3 + U**3) THEN
       PRINT "(3I1)", H, T, U
    ENDIF
    END DO
  END DO
END DO
call TAU_PROFILE_STOP(profiler)
END PROGRAM SUM_OF_CUBES
```

See Also

TAU_PROFILE_SET_NODE

Name

TAU_REGISTER_FORK -- Informs the measurement system that a fork has taken place

C/C++:

```
TAU_REGISTER_FORK(pid, option);
int pid;
enum TauFork_t option;
```

Description

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

C/C++:

```
pID = fork();
if (pID == 0) {
  printf("Parent : pid returned %d\n", pID)
}  else {
  // If we'd used the TAU_INCLUDE_PARENT_DATA, we 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);
}
```

Name

TAU_REGISTER_EVENT -- Registers a user event

C/C++:

```
TAU_REGISTER_EVENT(variable, event_name);
TauUserEvent variable;
char *event_name;
```

Fortran:

```
TAU_REGISTER_EVENT(variable, event_name);
int variable(2);
character event_name(size);
```

Description

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

C/C++ :

```
int user_square(int count) {
  TAU_REGISTER_EVENT(ue1, "UserSquare Event");
  TAU_EVENT(ue1, count * count);
  return 0;
}
```

Fortran :

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

See Also

TAU_EVENT, TAU_REGISTER_CONTEXT_EVENT, TAU_REPORT_STATISTICS, TAU_REPORT_THREAD_STATISTICS, TAU_GET_EVENT_NAMES, TAU_GET_EVENT_VALS

Name

TAU_EVENT -- Triggers a user event

C/C++:

```
TAU_EVENT(variable, value);
TauUserEvent variable;
double value;
```

Fortran:

```
TAU_EVENT(variable, value);
integer variable(2);
real value;
```

Description

Triggers an event that was registered with `TAU_REGISTER_EVENT`.

Example

C/C++ :

```
int user_square(int count) {
  TAU_REGISTER_EVENT(ue1, "UserSquare Event");
  TAU_EVENT(ue1, count * count);
  return 0;
}
```

Fortran :

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

See Also

TAU_REGISTER_EVENT

Name

TAU_REGISTER_CONTEXT_EVENT -- Registers a context event

C/C++:

```
TAU_REGISTER_CONTEXT_EVENT(variable, event_name);
TauUserEvent variable;
char *event_name;
```

Fortran:

```
TAU_REGISTER_CONTEXT_EVENT(variable, event_name);
int variable(2);
character event_name(size);
```

Description

Creates a context event with name. A context event appends the names of routines executing on the call-stack to the name specified by the user. Whenver a context event is triggered, the callstack is examined to determine the context of execution. Starting from the parent function where the event is triggered, TAU walks up the callstack to a depth specified by the user in the environment variable `TAU_CALLPATH_DEPTH` . If this environment variable is not specified, TAU uses 2 as the default depth. For e.g., if the user registers a context event with the name "memory used" and specifies 3 as the callpath depth, and if the event is triggered in two locations (in routine a, when it was called by b, when it was called by c, and in routine h, when it was called by g, when it was called by i), then, we'd see the user defined event information for "memory used: c() => b() => a()" and "memory used: i() => g() => h()".

Example

C/C++ :

```
int f2(void)
{
  static int count = 0;
  count ++;
  TAU_PROFILE("f2()", "(sleeps 2 sec, calls f3)", TAU_USER);
  TAU_REGISTER_CONTEXT_EVENT(event, "Iteration count");
/*
  if (count == 2)
    TAU_DISABLE_CONTEXT_EVENT(event);
*/
  printf("Inside f2: sleeps 2 sec, calls f3\n");

  TAU_CONTEXT_EVENT(event, 232+count);
  sleep(2);
  f3();
  return 0;
}
```

Fortran :

```
subroutine foo(id)
  integer id

  integer profiler(2) / 0, 0 /
  integer maev(2) / 0, 0 /
  integer mdev(2) / 0, 0 /
  save profiler, maev, mdev

  integer :: ierr
  integer :: h, t, u
  INTEGER, ALLOCATABLE :: STORAGEARY(:)
  DOUBLEPRECISION   edata

  call TAU_PROFILE_TIMER(profiler, 'FOO')
  call TAU_PROFILE_START(profiler)
  call TAU_PROFILE_SET_NODE(0)

  call TAU_REGISTER_CONTEXT_EVENT(maev, "STORAGEARY Alloc [cubes.f:20]")
  call TAU_REGISTER_CONTEXT_EVENT(mdev, "STORAGEARY Dealloc [cubes.f:37]")

  allocate(STORAGEARY(1:999), STAT=IERR)
  edata = SIZE(STORAGEARY)*sizeof(INTEGER)
  call TAU_CONTEXT_EVENT(maev, edata)
  ...
  deallocate(STORAGEARY)
  edata = SIZE(STORAGEARY)*sizeof(INTEGER)
  call TAU_CONTEXT_EVENT(mdev, edata)
  call TAU_PROFILE_STOP(profiler)
  end subroutine foo
```

See Also

TAU_CONTEXT_EVENT, TAU_ENABLE_CONTEXT_EVENT, TAU_DISABLE_CONTEXT_EVENT, TAU_REGISTER_EVENT, TAU_REPORT_STATISTICS, TAU_REPORT_THREAD_STATISTICS, TAU_GET_EVENT_NAMES, TAU_GET_EVENT_VALS

Name

TAU_CONTEXT_EVENT -- Triggers a context event

C/C++:

```
TAU_CONTEXT_EVENT(variable, value);
TauUserEvent variable;
double value;
```

Fortran:

```
TAU_CONTEXT_EVENT(variable, value);
integer variable(2);
real value;
```

Description

Triggers a context event. A context event associates the name with the list of routines along the callstack. A context event tracks information like a user defined event and TAU records the maxima, minima, mean, std. deviation and the number of samples for each context event. A context event helps distinguish the data supplied by the user based on the location where an event occurs and the sequence of actions (routine/timer invocations) that preceeded the event. The depth of the the callstack embedded in the context event's name is specified by the user in the environment variable TAU_CALLPATH_DEPTH. If this variable is not specified, TAU uses a default depth of 2.

Example

C/C++ :

```
int f2(void)
{
  static int count = 0;
  count ++;
  TAU_PROFILE("f2()", "(sleeps 2 sec, calls f3)", TAU_USER);
  TAU_REGISTER_CONTEXT_EVENT(event, "Iteration count");
/*
  if (count == 2)
    TAU_DISABLE_CONTEXT_EVENT(event);
*/
  printf("Inside f2: sleeps 2 sec, calls f3\n");

  TAU_CONTEXT_EVENT(event, 232+count);
  sleep(2);
  f3();
  return 0;
}
```

Fortran :

```
integer memevent(2) / 0, 0 /
save memevent
call TAU_REGISTER_CONTEXT_EVENT(memevent, "STORAGEARY mem allocated')
```

```
call TAU_CONTEXT_EVENT(memevent, SIZEOF(STORAGEARY)*sizeof(INTEGER))
```

See Also

TAU_REGISTER_CONTEXT_EVENT

Name

TAU_ENABLE_CONTEXT_EVENT -- Enable a context event

C/C++:

```
TAU_ENABLE_CONTEXT_EVENT(event);
TauUserEvent event;
```

Description

Enables a context event.

Example

C/C++ :

```
int f2(void) {
  static int count = 0;
  count ++;
  TAU_PROFILE("f2()", "(sleeps 2 sec, calls f3)", TAU_USER);
  TAU_REGISTER_CONTEXT_EVENT(event, "Iteration count");

  if (count == 2)
    TAU_DISABLE_CONTEXT_EVENT(event);
  else
    TAU_ENABLE_CONTEXT_EVENT(event);

  printf("Inside f2: sleeps 2 sec, calls f3\n");

  TAU_CONTEXT_EVENT(event, 232+count);
  sleep(2);
  f3();
  return 0;
}
```

See Also

TAU_REGISTER_CONTEXT_EVENT, TAU_DISABLE_CONTEXT_EVENT

Name

TAU_DISABLE_CONTEXT_EVENT -- Disable a context event

C/C++:

```
TAU_DISABLE_CONTEXT_EVENT(event);
TauUserEvent event;
```

Description

Disables a context event.

Example

C/C++ :

```
int f2(void) {
  static int count = 0;
  count ++;
  TAU_PROFILE("f2()", "(sleeps 2 sec, calls f3)", TAU_USER);
  TAU_REGISTER_CONTEXT_EVENT(event, "Iteration count");

  if (count == 2)
    TAU_DISABLE_CONTEXT_EVENT(event);
  else
    TAU_ENABLE_CONTEXT_EVENT(event);

  printf("Inside f2: sleeps 2 sec, calls f3\n");

  TAU_CONTEXT_EVENT(event, 232+count);
  sleep(2);
  f3();
  return 0;
}
```

See Also

TAU_REGISTER_CONTEXT_EVENT, TAU_ENABLE_CONTEXT_EVENT

Name

TAU_EVENT_SET_NAME -- Sets the name of an event

C/C++:

```
TAU_EVENT_SET_NAME(event, name);
TauUserEvent event;
const char *name;
```

Description

Changes the name of an event.

Example

C/C++ :

```
TAU_EVENT_SET_NAME(event, "new name");
```

See Also

TAU_REGISTER_EVENT

Name

TAU_EVENT_DISABLE_MAX -- Disables tracking of maximum statistic for a given event

C/C++:

```
TAU_EVENT_DISABLE_MAX(event);
TauUserEvent event;
```

Description

Disables tracking of maximum statistic for a given event

Example

C/C++ :

```
TAU_EVENT_DISABLE_MAX(event);
```

See Also

TAU_REGISTER_EVENT

Name

TAU_EVENT_DISABLE_MEAN -- Disables tracking of mean statistic for a given event

C/C++:

```
TAU_EVENT_DISABLE_MEAN(event);
TauUserEvent event;
```

Description

Disables tracking of mean statistic for a given event

Example

C/C++ :

```
TAU_EVENT_DISABLE_MEAN(event);
```

See Also

TAU_REGISTER_EVENT

Name

TAU_EVENT_DISABLE_MIN -- Disables tracking of minimum statistic for a given event

C/C++:

```
TAU_EVENT_DISABLE_MIN(event);
TauUserEvent event;
```

Description

Disables tracking of minimum statistic for a given event

Example

C/C++ :

```
TAU_EVENT_DISABLE_MIN(event);
```

See Also

TAU_REGISTER_EVENT

Name

TAU_EVENT_DISABLE_STDDEV -- Disables tracking of standard deviation statistic for a given event

C/C++:

```
TAU_EVENT_DISABLE_STDDEV(event);
TauUserEvent event;
```

Description

Disables tracking of standard deviation statistic for a given event

Example

C/C++ :

```
TAU_EVENT_DISABLE_STDDEV(event);
```

See Also

TAU_REGISTER_EVENT

Name

TAU_REPORT_STATISTICS -- Outputs statistics

C/C++:

```
TAU_REPORT_STATISTICS();
```

Fortran:

```
TAU_REPORT_STATISTICS();
```

Description

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.

Example

C/C++ :

```
TAU_REPORT_STATISTICS();
```

Fortran :

```
call TAU_REPORT_STATISTICS()
```

See Also

TAU_REGISTER_EVENT, TAU_REGISTER_CONTEXT_EVENT, TAU_REPORT_THREAD_STATISTICS

Name

TAU_REPORT_THREAD_STATISTICS -- Outputs statistics, plus thread statistics

C/C++:

```
TAU_REPORT_THREAD_STATISTICS();
```

Fortran:

```
TAU_REPORT_THREAD_STATISTICS();
```

Description

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.

Example

C/C++ :

```
TAU_REPORT_THREAD_STATISTICS();
```

Fortran :

```
call TAU_REPORT_THREAD_STATISTICS()
```

See Also

TAU_REGISTER_EVENT, TAU_REGISTER_CONTEXT_EVENT, TAU_REPORT_STATISTICS

Name

TAU_ENABLE_INSTRUMENTATION -- Enables instrumentation

C/C++:

```
TAU_ENABLE_INSTRUMENTATION();
```

Fortran:

```
TAU_ENABLE_INSTRUMENTATION();
```

Description

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

Example

C/C++ :

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

Fortran :

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

See Also

TAU_DISABLE_INSTRUMENTATION, TAU_ENABLE_GROUP, TAU_DISABLE_GROUP, TAU_INIT, TAU_PROFILE_INIT

Name

TAU_DISABLE_INSTRUMENTATION -- Disables instrumentation

C/C++:

```
TAU_DISABLE_INSTRUMENTATION();
```

Fortran:

```
TAU_DISABLE_INSTRUMENTATION();
```

Description

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.

Example

C/C++ :

```
int main(int argc, char **argv) {
  foo();
  TAU_DISABLE_INSTRUMENTATION();
  for (int i =0; i < N; i++) {
    bar();  // not recorded
  }
  TAU_DISABLE_INSTRUMENTATION();
  bar(); // recorded
}
```

Fortran :

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

See Also

TAU_ENABLE_INSTRUMENTATION, TAU_ENABLE_GROUP, TAU_DISABLE_GROUP, TAU_INIT, TAU_PROFILE_INIT

Name

TAU_ENABLE_GROUP -- Enables tracking of a given group

C/C++:

```
TAU_ENABLE_GROUP(group);
TauGroup_t group;
```

Fortran:

```
TAU_ENABLE_GROUP(group);
integer group;
```

Description

Enables the instrumentation for a given group. By default, it is already on.

Example

C/C++ :

```
void foo() {
  TAU_PROFILE("foo()", " ", TAU_USER);
  ...
  TAU_ENABLE_GROUP(TAU_USER);
}
```

Fortran :

```
  include 'Profile/TauFAPI.h'
  call TAU_ENABLE_GROUP(TAU_USER)
```

See Also

TAU_ENABLE_INSTRUMENTATION, TAU_DISABLE_INSTRUMENTATION, TAU_DISABLE_GROUP, TAU_INIT, TAU_PROFILE_INIT

Name

TAU_DISABLE_GROUP -- Disables tracking of a given group

C/C++:

```
TAU_DISABLE_GROUP(group);
TauGroup_t group;
```

Fortran:

```
TAU_DISABLE_GROUP(group);
integer group;
```

Description

Disables the instrumentation for a given group. By default, it is on.

Example

C/C++ :

```
void foo() {
  TAU_PROFILE("foo()", " ", TAU_USER);
  ...
  TAU_DISABLE_GROUP(TAU_USER);
}
```

Fortran :

```
  include 'Profile/TauFAPI.h'
  call TAU_DISABLE_GROUP(TAU_USER)
```

See Also

TAU_ENABLE_INSTRUMENTATION, TAU_DISABLE_INSTRUMENTATION, TAU_ENABLE_GROUP, TAU_INIT, TAU_PROFILE_INIT

Name

TAU_PROFILE_TIMER_SET_GROUP -- Change the group of a timer

C/C++:

```
TAU_PROFILE_TIMER_SET_GROUP(timer, group);
Profiler timer;
TauGroup_t group;
```

Description

`TAU_PROFILE_TIMER_SET_GROUP` changes the group associated with a timer.

Example

C/C++:

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

See Also

TAU_PROFILE_TIMER, TAU_PROFILE_TIMER_SET_GROUP_NAME

Name

TAU_PROFILE_TIMER_SET_GROUP_NAME -- Changes the group name for a timer

C/C++:

```
TAU_PROFILE_TIMER_SET_GROUP_NAME(timer, groupname);
Profiler timer;
char *groupname;
```

Description

TAU_PROFILE_TIMER_SET_GROUP_NAME changes the group name associated with a given timer.

Example

C/C++ :

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

See Also

TAU_PROFILE_TIMER, TAU_PROFILE_TIMER_SET_GROUP

Name

TAU_PROFILE_TIMER_SET_NAME -- Changes the name of a timer

C/C++:

```
TAU_PROFILE_TIMER_SET_NAME(timer, newname);
Profiler timer;
string newname;
```

Description

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

Example

C/C++ :

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

See Also

TAU_PROFILE_TIMER

Name

TAU_PROFILE_TIMER_SET_TYPE -- Changes the type of a timer

C/C++:

```
TAU_PROFILE_TIMER_SET_TYPE(timer, newname);
Profiler timer;
string newname;
```

Description

TAU_PROFILE_TIMER_SET_TYPE macro changes the type associated with a timer to the newname argument.

Example

C/C++ :

```
void foo() {
  TAU_PROFILE_TIMER(timer1, "foo", "int", TAU_USER);
  ...
  TAU_PROFILE_TIMER_SET_TYPE(timer1, "long");
}
```

See Also

TAU_PROFILE_TIMER

Name

TAU_PROFILE_SET_GROUP_NAME -- Changes the group name of a profiled section

C/C++:

```
TAU_PROFILE_SET_GROUP_NAME(groupname);
char *groupname;
```

Description

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

Example

C/C++ :

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

See Also

TAU_PROFILE

Name

TAU_INIT -- Processes command-line arguments for selective instrumentation

C/C++:

```
TAU_INIT(argc, argv);
int *argc;
char ***argv;
```

Description

TAU_INIT parses and removes 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. TAU_INIT differs from TAU_PROFILE_INIT only in the argument types.

Example

C/C++:

```
int main(int argc, char **argv) {
  TAU_PROFILE("main()", "int (int, char **)", TAU_GROUP_12);
  TAU_INIT(&argc, &argv);
  ...
}
```

```
% ./a.out --profile 12+14
```

See Also

TAU_PROFILE_INIT

Name

TAU_PROFILE_INIT -- Processes command-line arguments for selective instrumentation

C/C++:

```
TAU_PROFILE_INIT(argc, argv);
int argc;
char **argv;
```

Fortran:

```
TAU_PROFILE_INIT();
```

Description

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. TAU_INIT differs from TAU_PROFILE_INIT only in the argument types.

Example

C/C++ :

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

% ./a.out --profile 12+14
```

Fortran :

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

  call TAU_PROFILE_INIT()
  ...
```

See Also

TAU_INIT

Name

TAU_GET_PROFILE_GROUP -- Creates groups based on names

C/C++:

```
TAU_GET_PROFILE_GROUP(groupname);
char *groupname;
```

Description

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

Example

C/C++:

```
#define PARTICLES TAU_GET_PROFILE_GROUP("PARTICLES")

void foo() {
  TAU_PROFILE("foo()", " ", PARTICLES);
}

void bar() {
  TAU_PROFILE("bar()", " ", PARTICLES);
}
```

See Also

TAU_ENABLE_GROUP_NAME, TAU_DISABLE_GROUP_NAME, TAU_ENABLE_ALL_GROUPS, TAU_DISABLE_ALL_GROUPS

Name

TAU_ENABLE_GROUP_NAME -- Enables a group based on name

C/C++:

```
TAU_ENABLE_GROUP_NAME(groupname);
char *groupname;
```

Fortran:

```
TAU_ENABLE_GROUP_NAME(groupname);
character groupname(size);
```

Description

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

Example

C/C++ :

```
/* tau_instrumentor was invoked with -g DTM for a set of files */
TAU_DISABLE_GROUP_NAME("DTM");
dtm_routines();
/* disable and then re-enable the group with the name DTM */
TAU_ENABLE_GROUP_NAME("DTM");
```

Fortran :

```
! tau_instrumentor was invoked with -g DTM for this file
    call TAU_PROFILE_TIMER(profiler, "ITERATE>DTM")

    call TAU_DISABLE_GROUP_NAME("DTM")
! Disable, then re-enable DTM group
    call TAU_ENABLE_GROUP_NAME("DTM")
```

See Also

TAU_GET_PROFILE_GROUP, TAU_DISABLE_GROUP_NAME, TAU_ENABLE_ALL_GROUPS, TAU_DISABLE_ALL_GROUPS

Name

TAU_DISABLE_GROUP_NAME -- Disables a group based on name

C/C++:

```
TAU_DISABLE_GROUP_NAME(groupname);
char *groupname;
```

Fortran:

```
TAU_DISABLE_GROUP_NAME(groupname);
character groupname(size);
```

Description

Similar to TAU_ENABLE_GROUP_NAME , this macro turns off the instrumentation in all routines associated with the dynamic group created using the tau_instrumentor -g <group_name> argument.

Example

C/C++ :

```
/* tau_instrumentor was invoked with -g DTM for a set of files */
TAU_DISABLE_GROUP_NAME("DTM");
dtm_routines();
/* disable and then re-enable the group with the name DTM */
TAU_ENABLE_GROUP_NAME("DTM");
```

Fortran :

```
! tau_instrumentor was invoked with -g DTM for this file
    call TAU_PROFILE_TIMER(profiler, "ITERATE>DTM")

    call TAU_DISABLE_GROUP_NAME("DTM")
! Disable, then re-enable DTM group
    call TAU_ENABLE_GROUP_NAME("DTM")
```

See Also

TAU_GET_PROFILE_GROUP, TAU_ENABLE_GROUP_NAME, TAU_ENABLE_ALL_GROUPS, TAU_DISABLE_ALL_GROUPS

Name

TAU_ENABLE_ALL_GROUPS -- Enables instrumentation in all groups

C/C++:

```
TAU_ENABLE_ALL_GROUPS();
```

Fortran:

```
TAU_ENABLE_ALL_GROUPS();
```

Description

This macro turns on instrumentation in all groups

Example

C/C++ :

```
TAU_ENABLE_ALL_GROUPS();
```

Fortran :

```
call TAU_ENABLE_ALL_GROUPS();
```

See Also

TAU_GET_PROFILE_GROUP, TAU_ENABLE_GROUP_NAME, TAU_DISABLE_GROUP_NAME, TAU_DISABLE_ALL_GROUPS

Name

TAU_DISABLE_ALL_GROUPS -- Disables instrumentation in all groups

C/C++:

```
TAU_DISABLE_ALL_GROUPS();
```

Fortran:

```
TAU_DISABLE_ALL_GROUPS();
```

Description

This macro turns off instrumentation in all groups.

Example

C/C++ :

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

Fortran :

```
call TAU_DISABLE_ALL_GROUPS();
```

See Also

TAU_GET_PROFILE_GROUP, TAU_ENABLE_GROUP_NAME, TAU_DISABLE_GROUP_NAME, TAU_ENABLE_ALL_GROUPS

Name

TAU_GET_EVENT_NAMES -- Gets the registered user events.

C/C++:

```
TAU_GET_EVENT_NAMES(eventList, numEvents);
const char ***eventList;
int *numEvents;
```

Description

Retrieves user event names for all user-defined events

Example

C/C++:

```
const char **eventList;
int numEvents;

TAU_GET_EVENT_NAMES(eventList, numEvents);

cout << "numEvents: " << numEvents << endl;
```

See Also

TAU_REGISTER_EVENT, TAU_REGISTER_CONTEXT_EVENT, TAU_GET_EVENT_VALS

Name

TAU_GET_EVENT_VALS -- Gets user event data for given user events.

C/C++:

```
TAU_GET_EVENT_VALS(inUserEvents, numUserEvents, numEvents, max, min,
mean, sumSqe);
const char **inUserEvents;
int numUserEvents;
int **numEvents;
double **max;
double **min;
double **mean;
double **sumSqe;
```

Description

Retrieves user defined event data for the specified user defined events. The list of events are specified by the first parameter (eventList) and the user specifies the number of events in the second parameter (numUserEvents). TAU returns the number of times the event was invoked in the numUserEvents. The max, min, mean values are returned in the following parameters. TAU computes the sum of squares of the given event and returns this value in the next argument (sumSqe).

Example

C/C++ :

```
  const char **eventList;
  int numEvents;

  TAU_GET_EVENT_NAMES(eventList, numEvents);

  cout << "numEvents: " << numEvents << endl;

  if (numEvents > 0) {
    int *numSamples;
    double *max;
    double *min;
    double *mean;
    double *sumSqr;

    TAU_GET_EVENT_VALS(eventList, numEvents, numSamples,
      max, min, mean, sumSqr);
    for (int i=0; i<numEvents; i++) {
      cout << "------------------\n";
      cout << "User Event:        " << eventList[i] << endl;
      cout << "Number of Samples: " << numSamples[i] << endl;
      cout << "Maximum Value:     " << max[i] << endl;
      cout << "Minimum Value:     " << min[i] << endl;
      cout << "Mean Value:        " << mean[i] << endl;
      cout << "Sum Squared:       " << sumSqr[i] << endl;
    }
  }
}
```

See Also

TAU_REGISTER_EVENT, TAU_REGISTER_CONTEXT_EVENT, TAU_GET_EVENT_NAMES

Name

TAU_GET_COUNTER_NAMES -- Gets the counter names

C/C++:

```
TAU_GET_COUNTER_NAMES(counterList, numCounters);
char **counterList;
int numCounters;
```

Description

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

Example

C/C++ :

```
int numOfCounters;
const char ** counterList;

TAU_GET_COUNTER_NAMES(counterList, numOfCounters);

for(int j=0;j<numOfCounters;j++){
  cout << "The counter names so far are: " << counterList[j] << endl;
}
```

See Also

TAU_GET_FUNC_NAMES, TAU_GET_FUNC_VALS

Name

TAU_GET_FUNC_NAMES -- Gets the function names

C/C++:

```
TAU_GET_FUNC_NAMES(functionList, numFuncs);
char **functionList;
int numFuncs;
```

Description

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

Example

C/C++ :

```
const char ** functionList;
int numOfFunctions;

TAU_GET_FUNC_NAMES(functionList, numOfFunctions);

for(int i=0;i<numOfFunctions;i++){
  cout << "This function names so far are: " << functionList[i] << endl;
}
```

See Also

TAU_GET_COUNTER_NAMES, TAU_GET_FUNC_VALS, TAU_DUMP_FUNC_NAMES, TAU_DUMP_FUNC_VALS

Name

TAU_GET_FUNC_VALS -- Gets detailed performance data for given functions

C/C++:

```
TAU_GET_FUNC_VALS(inFuncs, numOfFuncs, counterExclusiveValues, counterInclusiveValues, numOfCalls, numOfSubRoutines, counterNames, numOfCounters, tid);
const char **inFuncs;
int numOfFuncs;
double ***counterExclusiveValues;
double ***counterInclusiveValues;
int **numOfCalls;
int **numOfSubRoutines;
const char ***counterNames;
int *numOfCounters;
int tid;
```

Description

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

Example

C/C++ :

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

TAU_GET_FUNC_NAMES(functionList, numOfFunctions);

/* We are only interested in the first two routines
that are executing in this context. So, we allocate
space for two routine names and get the performance
data for these two routines at runtime. */
if (numOfFunctions >=2 ) {
  inFuncs = (const char **) malloc(sizeof(const char *) * 2);

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

  //Just to show consistency.
  TAU_DB_DUMP();

  TAU_GET_FUNC_VALS(inFuncs, 2,
  counterExclusiveValues,
```

```
  counterInclusiveValues,
  numOfCalls,
  numOfSubRoutines,
  counterNames,
  numOfCouns);

  TAU_DUMP_FUNC_VALS_INCR(inFuncs, 2);


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

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


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

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

TAU_DB_DUMP_INCR();
```

See Also

TAU_GET_COUNTER_NAMES, TAU_GET_FUNC_NAMES, TAU_DUMP_FUNC_NAMES, TAU_DUMP_FUNC_VALS

Name

TAU_ENABLE_TRACKING_MEMORY -- Enables memory tracking

C/C++:

```
TAU_ENABLE_TRACKING_MEMORY();
```

Fortran:

```
TAU_ENABLE_TRACKING_MEMORY();
```

Description

Enables tracking of the heap memory utilization in the program. TAU takes a sample of the heap memory utilized (as reported by the mallinfo system call) and associates it with a single global user defined event. An interrupt is generated every 10 seconds and the value of the heap memory used is recorded in the user defined event. The inter-interrupt interval (default of 10 seconds) may be set by the user using the call `TAU_SET_INTERRUPT_INTERVAL`.

Example

C/C++ :

```
TAU_ENABLE_TRACKING_MEMORY();
```

Fortran :

```
call TAU_ENABLE_TRACKING_MEMORY()
```

See Also

TAU_DISABLE_TRACKING_MEMORY, TAU_SET_INTERRUPT_INTERVAL, TAU_TRACK_MEMORY, TAU_TRACK_MEMORY_HERE

Name

TAU_DISABLE_TRACKING_MEMORY -- Disables memory tracking

C/C++:

```
TAU_DISABLE_TRACKING_MEMORY();
```

Fortran:

```
TAU_DISABLE_TRACKING_MEMORY();
```

Description

Disables tracking of heap memory utilization. This call may be used in sections of code where TAU should not interrupt the execution to periodically track the heap memory utilization.

Example

C/C++ :

```
TAU_DISABLE_TRACKING_MEMORY();
```

Fortran :

```
call TAU_DISABLE_TRACKING_MEMORY()
```

See Also

TAU_ENABLE_TRACKING_MEMORY, TAU_SET_INTERRUPT_INTERVAL, TAU_TRACK_MEMORY, TAU_TRACK_MEMORY_HERE

Name

TAU_TRACK_MEMORY -- Initializes memory tracking system

C/C++:

```
TAU_TRACK_MEMORY();
```

Fortran:

```
TAU_TRACK_MEMORY();
```

Description

For memory profiling, there are two modes of operation: 1) the user explicitly inserts TAU_TRACK_MEMORY_HERE() calls in the source code and the memory event is triggered at those locations, and 2) the user enables tracking memory by calling TAU_TRACK_MEMORY() and an interrupt is generated every 10 seconds and the memory event is triggered with the current value. Also, this interrupt interval can be changed by calling TAU_SET_INTERRUPT_INTERVAL(value). The tracking of memory events in both cases can be explictly enabled or disabled by calling the macros TAU_ENABLE_TRACKING_MEMORY() or TAU_DISABLE_TRACKING_MEMORY() respectively.

Example

C/C++ :

```
TAU_TRACK_MEMORY();
```

Fortran :

```
call TAU_TRACK_MEMORY()
```

See Also

TAU_ENABLE_TRACKING_MEMORY, TAU_DISABLE_TRACKING_MEMORY, TAU_SET_INTERRUPT_INTERVAL, TAU_TRACK_MEMORY_HERE, TAU_TRACK_MEMORY_HEADROOM

Name

TAU_TRACK_MEMORY_HERE -- Triggers memory tracking at a given execution point

C/C++:

```
TAU_TRACK_MEMORY_HERE();
```

Fortran:

```
TAU_TRACK_MEMORY_HERE();
```

Description

Triggers memory tracking at a given execution point

Example

C/C++ :

```
int main(int argc, char **argv) {
  TAU_PROFILE("main()", " ", TAU_DEFAULT);
  TAU_PROFILE_SET_NODE(0);

  TAU_TRACK_MEMORY_HERE();

  int *x = new int[5*1024*1024];
  TAU_TRACK_MEMORY_HERE();
  return 0;
}
```

Fortran :

```
INTEGER, ALLOCATABLE :: STORAGEARY(:)
allocate(STORAGEARY(1:999), STAT=IERR)

! if we wish to record a sample of the heap memory
! utilization at this point, invoke the following call:
call TAU_TRACK_MEMORY_HERE()
```

See Also

TAU_TRACK_MEMORY

Name

TAU_ENABLE_TRACKING_MEMORY_HEADROOM -- Enables memory headroom tracking

C/C++:

```
TAU_ENABLE_TRACKING_MEMORY_HEADROOM();
```

Fortran:

```
TAU_ENABLE_TRACKING_MEMORY_HEADROOM();
```

Description

`TAU_ENABLE_TRACKING_MEMORY_HEADROOM()` enables memory headroom tracking after a `TAU_DISABLE_TRACKING_MEMORY_HEADROOM()`.

Example

C/C++ :

```
TAU_DISABLE_TRACKING_MEMORY_HEADROOM();
/* do some work */
...
/* re-enable tracking memory headroom */
TAU_ENABLE_TRACKING_MEMORY_HEADROOM();
```

Fortran :

```
call TAU_ENABLE_TRACKING_MEMORY_HEADROOM();
```

See Also

TAU_TRACK_MEMORY_HEADROOM, TAU_DISABLE_TRACKING_MEMORY_HEADROOM, TAU_TRACK_MEMORY_HEADROOM_HERE, TAU_SET_INTERRUPT_INTERVAL

Name

TAU_DISABLE_TRACKING_MEMORY_HEADROOM -- Disables memory headroom tracking

C/C++:

```
TAU_DISABLE_TRACKING_MEMORY_HEADROOM();
```

Fortran:

```
TAU_DISABLE_TRACKING_MEMORY_HEADROOM();
```

Description

`TAU_DISABLE_TRACKING_MEMORY_HEADROOM()` disables memory headroom tracking.

Example

C/C++ :

```
TAU_DISABLE_TRACKING_MEMORY_HEADROOM();
```

Fortran :

```
call TAU_DISABLE_TRACKING_MEMORY_HEADROOM()
```

See Also

TAU_TRACK_MEMORY_HEADROOM, TAU_ENABLE_TRACKING_MEMORY_HEADROOM, TAU_TRACK_MEMORY_HEADROOM_HERE, TAU_SET_INTERRUPT_INTERVAL

Name

TAU_TRACK_MEMORY_HEADROOM -- Track the headroom (amount of memory for a process to grow) by periodically interrupting the program

C/C++:

```
TAU_TRACK_MEMORY_HEADROOM();
```

Fortran:

```
TAU_TRACK_MEMORY_HEADROOM();
```

Description

Tracks the amount of memory available for the process before it runs out of free memory on the heap. This call sets up a signal handler that is invoked every 10 seconds by an interrupt (this interval may be altered by using the `TAU_SET_INTERRUPT_INTERVAL` call). Inside the interrupt handler, TAU evaluates how much memory it can allocate and associates it with the callstack using the TAU context events (See TAU_REGISTER_CONTEXT_EVENT). The user can vary the size of the callstack by setting the environment variable `TAU_CALLPATH_DEPTH` (default is 2). This call is useful on machines like IBM BG/L where no virtual memory (or paging using the swap space) is present. The amount of heap memory available to the program is limited by the amount of available physical memory. TAU executes a series of malloc calls with a granularity of 1MB and determines the amount of memory available for the program to grow.

Example

C/C++ :

```
TAU_TRACK_MEMORY_HEADROOM();
```

Fortran :

```
call TAU_TRACK_MEMORY_HEADROOM()
```

See Also

TAU_TRACK_MEMORY, TAU_SET_INTERRUPT_INTERVAL, TAU_ENABLE_TRACKING_MEMORY_HEADROOM, TAU_DISABLE_TRACKING_MEMORY_HEADROOM, TAU_TRACK_MEMORY_HEADROOM_HERE

Name

TAU_TRACK_MEMORY_HEADROOM_HERE -- Takes a sample of the amount of memory available at a given point.

C/C++:

```
TAU_TRACK_MEMORY_HEADROOM_HERE();
```

Fortran:

```
TAU_TRACK_MEMORY_HEADROOM_HERE();
```

Description

Instead of relying on a periodic interrupt to track the amount of memory available to grow, this call may be used to take a sample at a given location in the source code. Context events are used to track the amount of memory headroom.

Example

C/C++ :

```
ary = new double [1024*1024*50];
TAU_TRACK_MEMORY_HEADROOM_HERE();
```

Fortran :

```
INTEGER, ALLOCATABLE :: STORAGEARY(:)
allocate(STORAGEARY(1:999), STAT=IERR)
TAU_TRACK_MEMORY_HEADROOM_HERE();
```

See Also

TAU_TRACK_MEMORY_HEADROOM

Name

TAU_SET_INTERRUPT_INTERVAL -- Change the inter-interrupt interval for tracking memory and headroom

C/C++:

```
TAU_SET_INTERRUPT_INTERVAL(value);
int value;
```

Fortran:

```
TAU_SET_INTERRUPT_INTERVAL(value);
integer value;
```

Description

Set the interrupt interval for tracking memory and headroom (See TAU_TRACK_MEMORY and TAU_TRACK_MEMORY_HEADROOM). By default an inter-interrupt interval of 10 seconds is used in TAU. This call allows the user to set it to a different value specified by the argument value.

Example

C/C++ :

```
TAU_SET_INTERRUPT_INTERVAL(2)
/* invokes the interrupt handler for memory every 2s */
```

Fortran :

```
call TAU_SET_INTERRUPT_INTERVAL(2)
```

See Also

TAU_TRACK_MEMORY, TAU_TRACK_MEMORY_HEADROOM

Name

CT -- Returns the type information for a variable

C/C++:

```
CT(variable);
<type> variable;
```

Description

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.

Example

C/C++ :

```
TAU_PROFILE("foo::memberfunc()", CT(*this), TAU_DEFAULT);
```

See Also

TAU_PROFILE, TAU_PROFILE_TIMER, TAU_TYPE_STRING

Name

TAU_TYPE_STRING -- Creates a type string

C++:

```
TAU_TYPE_STRING(variable, type_string);
string &variable;
string &type_string;
```

Description

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

C++ :

```
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);

```
template<class PLayout>
unsigned ParticleBase<PLayout7>::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)
```

See Also

CT, TAU_PROFILE, TAU_PROFILE_TIMER

Name

TAU_DB_DUMP -- Dumps the profile database to disk

C/C++:

```
TAU_DB_DUMP();
```

Fortran:

```
TAU_DB_DUMP();
```

Description

Dumps the profile database to disk. The format of the files is the same as regular profiles, they are simply prefixed with "dump" instead of "profile".

Example

C/C++ :

```
TAU_DB_DUMP();
```

Fortran :

```
call TAU_DB_DUMP()
```

See Also

TAU_DB_DUMP_PREFIX, TAU_DB_DUMP_INCR, TAU_DUMP_FUNC_NAMES, TAU_DUMP_FUNC_VALS, TAU_DUMP_FUNC_VALS_INCR, TAU_DB_PURGE, TAU_PROFILE_EXIT

Name

TAU_DB_DUMP_INCR -- Dumps profile database into timestamped profiles on disk

C/C++:

```
TAU_DB_DUMP_INCR();
```

Description

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

Example

C/C++ :

```
TAU_DB_DUMP_INCR();
```

See Also

TAU_DB_DUMP, TAU_DB_DUMP_PREFIX, TAU_DUMP_FUNC_NAMES, TAU_DUMP_FUNC_VALS, TAU_DUMP_FUNC_VALS_INCR, TAU_DB_PURGE, TAU_PROFILE_EXIT

Name

TAU_DB_DUMP_PREFIX -- Dumps the profile database into profile files with a given prefix

C/C++:

```
TAU_DB_DUMP_PREFIX(prefix);
char *prefix;
```

Fortran:

```
TAU_DB_DUMP_PREFIX(prefix);
character prefix(size);
```

Description

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

Example

C/C++ :

```
TAU_DB_DUMP_PREFIX("prefix");
```

Fortran :

```
call TAU_DB_DUMP_PREFIX("prefix")
```

See Also

TAU_DB_DUMP

Name

TAU_DB_PURGE -- Purges the performance data.

C/C++:

```
TAU_DB_PURGE();
```

Description

Purges the performance data collected so far.

Example

C/C++ :

```
TAU_DB_PURGE();
```

See Also

TAU_DB_DUMP

Name

TAU_DUMP_FUNC_NAMES -- Dumps function names to disk

C/C++:

```
TAU_DUMP_FUNC_NAMES();
```

Description

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

Example

C/C++ :

```
TAU_DUMP_FUNC_NAMES();
```

See Also

TAU_DB_DUMP, TAU_DUMP_FUNC_VALS, TAU_DUMP_FUNC_VALS_INCR

Name

TAU_DUMP_FUNC_VALS -- Dumps performance data for given functions to disk.

C/C++:

```
TAU_DUMP_FUNC_VALS(inFuncs, numFuncs);
char **inFuncs;
int numFuncs;
```

Description

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

Example

C/C++ :

See Also

TAU_DB_DUMP, TAU_DUMP_FUNC_NAMES, TAU_DUMP_FUNC_VALS_INCR

Name

TAU_DUMP_FUNC_VALS_INCR -- Dumps function values with a timestamp

C/C++:

```
TAU_DUMP_FUNC_VALS_INCR(inFuncs, numFuncs);
char **inFuncs;
int numFuncs;
```

Description

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

Example

C/C++ :

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

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

TAU_DUMP_FUNC_VALS(inFuncs, 2);
```

See Also

TAU_DB_DUMP, TAU_DUMP_FUNC_NAMES, TAU_DUMP_FUNC_VALS

Name

TAU_PROFILE_STMT -- Executes a statement only when TAU is used.

C/C++:

```
TAU_PROFILE_STMT(statement);
statement statement;
```

Description

TAU_PROFILE_STMT executes a statement, or 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

C/C++ :

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

Name

TAU_PROFILE_CALLSTACK -- Generates a callstack trace at a given location.

C/C++:

```
TAU_PROFILE_CALLSTACK();
```

Description

When TAU is configured with `-PROFILECALLSTACK` configuration option, and this call is invoked, a callpath trace is generated. A GUI for viewing this trace is included in TAU's utils/csUI directory. This option is deprecated.

Example

C/C++ :

```
TAU_PROFILE_CALLSTACK();
```

Name

TAU_TRACE_RECVMSG -- Traces a receive operation

C/C++:

```
TAU_TRACE_RECVMSG(tag, source, length);
int tag;
int source;
int length;
```

Fortran:

```
TAU_TRACE_RECVMSG(tag, source, length);
integer tag;
integer source;
integer length;
```

Description

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

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

```
$(TAU_MPI_LIBS)
```

uses these macros internally for logging messages.

Example

C/C++ :

```
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);
}
```

Fortran :

```
call TAU_TRACE_RECVMSG(tag, source, length)
call TAU_TRACE_SENDMSG(tag, destination, length)
```

See Also

TAU_TRACE_SENDMSG

Name

TAU_TRACE_SENDMSG -- Traces a receive operation

C/C++:

```
TAU_TRACE_SENDMSG(tag, source, length);
int tag;
int source;
int length;
```

Fortran:

```
TAU_TRACE_SENDMSG(tag, source, length);
integer tag;
integer source;
integer length;
```

Description

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

NOTE: When TAU is configured to use MPI (-mpiinc=<dir> -mpilib=<dir>), the TAU_TRACE_SENDMSG and TAU_TRACE_SENDMSG macros are not required. The wrapper interposition library in

```
$(TAU_MPI_LIBS)
```

uses these macros internally for logging messages.

Example

C/C++ :

```
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);
}
```

Fortran :

```
call TAU_TRACE_RECVMSG(tag, source, length)
call TAU_TRACE_SENDMSG(tag, destination, length)
```

See Also

TAU_TRACE_RECVMSG

TAU Mapping API

Introduction

TAU allows the user to map performance data of entities from one layer to another in multi-layered software. Mapping is used in profiling (and tracing) both synchronous and asynchronous models of computation.

For mapping, the following macros are used. First locate and identify the higher-level statement using the TAU_MAPPING macro. Then, associate a function identifier with it using the TAU_MAPPING_OBJECT. Associate the high level statement to a FunctionInfo object that will be visible to lower level code, using TAU_MAPPING_LINK, and then profile entire blocks using TAU_MAPPING_PROFILE. Independent sets of statements can be profiled using TAU_MAPPING_PROFILE_TIMER, TAU_MAPPING_PROFILE_START, and TAU_MAPPING_PROFILE_STOP macros using the FunctionInfo object.

The TAU `examples/mapping` directory has two examples (embedded and external) that illustrate the use of this mapping API for generating object-oriented profiles.

Name

TAU_MAPPING -- Encapsulates a C++ statement for profiling

C/C++:

```
TAU_MAPPING(statement, key);
statement statement;
TauGroup_t key;
```

Description

TAU_MAPPING is used to encapsulate a C++ statement as a timer. A timer will be made, named by the statment, and will profile the statement. The key given can be used with TAU_MAPPING_LINK to retrieve the timer.

Example

C/C++ :

```
int main(int argc, char **argv) {
  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);
  ...
}
```

See Also

TAU_MAPPING_CREATE, TAU_MAPPING_LINK

Name

TAU_MAPPING_CREATE -- Creates a mapping

C/C++:

```
TAU_MAPPING_CREATE(name, type, groupname, key, tid);
char *name;
char *type;
char *groupname;
unsigned long key;
int tid;
```

Description

TAU_MAPPING_CREATE creates a mapping and associates it with the key that is specified. Later, this key may be used to retrieve the FunctionInfo object associated with this key for timing purposes. The thread identifier is specified in the `tid` parameter.

Example

C/C++:

```
class MyClass {
  public:
    MyClass() {
      TAU_MAPPING_LINK(runtimer, TAU_USER);
    }
    ~MyClass() {}

    void Run(void) {
      TAU_MAPPING_PROFILE(runtimer); // For one object
      TAU_PROFILE("MyClass::Run()", " void (void)", TAU_USER1);

      cout <<"Sleeping for 2 secs..."<<endl;
      sleep(2);
    }
  private:
    TAU_MAPPING_OBJECT(runtimer)  // EMBEDDED ASSOCIATION
};

int main(int argc, char **argv) {
  TAU_PROFILE_INIT(argc, argv);
  TAU_PROFILE("main()", "int (int, char **)", TAU_DEFAULT);
  MyClass x, y, z;
  TAU_MAPPING_CREATE("MyClass::Run() for object a", " " , TAU_USER,
                     "TAU_USER", 0);
  MyClass a;
  TAU_PROFILE_SET_NODE(0);
  cout <<"Inside main"<<endl;

  a.Run();
  x.Run();
  y.Run();
}
```

See Also

TAU_MAPPING_LINK, TAU_MAPPING_OBJECT, TAU_MAPPING_PROFILE

Name

TAU_MAPPING_LINK -- Creates a mapping link

C/C++:

```
TAU_MAPPING_LINK(FuncIdVar, Key);
FunctionInfo FuncIdVar;
unsigned long Key;
```

Description

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

Example

C/C++ :

```
class MyClass {
  public:
    MyClass() { }
    ~MyClass() { }

    void Run(void) {
      TAU_MAPPING_OBJECT(runtimer)
      TAU_MAPPING_LINK(runtimer, (unsigned long) this);
      TAU_MAPPING_PROFILE(runtimer); // For one object
      TAU_PROFILE("MyClass::Run()", " void (void)", TAU_USER1);

      /* ... */
    }
};

int main(int argc, char **argv) {
  TAU_PROFILE_INIT(argc, argv);
  TAU_PROFILE("main()", "int (int, char **)", TAU_DEFAULT);
  MyClass x, y, z;
  MyClass a;
  TAU_MAPPING_CREATE("MyClass::Run() for object a", " " ,
                     (TauGroup_t) &a, "TAU_USER", 0);
  TAU_MAPPING_CREATE("MyClass::Run() for object x", " " ,
                     (TauGroup_t) &x, "TAU_USER", 0);
  TAU_PROFILE_SET_NODE(0);
  cout <<"Inside main"<<endl;

  a.Run();
  x.Run();
  y.Run();
```

See Also

TAU_MAPPING_CREATE, TAU_MAPPING_OBJECT, TAU_MAPPING_PROFILE

Name

TAU_MAPPING_OBJECT -- Declares a mapping object

C/C++:

```
TAU_MAPPING_OBJECT(FuncIdVar);
FunctionInfo FuncIdVar;
```

Description

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:

Example

C/C++ :

```
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)
};
```

See Also

TAU_MAPPING_CREATE, TAU_MAPPING_LINK, TAU_MAPPING_PROFILE

Name

TAU_MAPPING_PROFILE -- Profiles a block based on a mapping

C/C++:

```
TAU_MAPPING_PROFILE(FuncIdVar);
FunctionInfo *FuncIdVar;
```

Description

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:

Example

C/C++ :

```
// 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
}
```

See Also

TAU_MAPPING_CREATE, TAU_MAPPING_LINK, TAU_MAPPING_OBJECT

Name

TAU_MAPPING_PROFILE_START -- Starts a mapping timer

C/C++:

```
TAU_MAPPING_PROFILE_START(timer, tid);
Profiler timer;
int tid;
```

Description

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.

Example

C/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 = 4854\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.
}
```

See Also

TAU_MAPPING_PROFILE_TIMER, TAU_MAPPING_PROFILE_STOP

Name

TAU_MAPPING_PROFILE_STOP -- Stops a mapping timer

C/C++:

```
TAU_MAPPING_PROFILE_STOP(timer, tid);
Profiler timer;
int tid;
```

Description

TAU_MAPPING_PROFILE_STOP stops 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.

Example

C/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 = 4854\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.
}
```

See Also

TAU_MAPPING_PROFILE_TIMER, TAU_MAPPING_PROFILE_START

Name

TAU_MAPPING_PROFILE_TIMER -- Declares a mapping timer

C/C++:

```
TAU_MAPPING_PROFILE_TIMER(timer, FuncIdVar);
Profiler timer;
FunctionInfo *FuncIdVar;
```

Description

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.

Example

C/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 = 4854\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.
}
```

See Also

TAU_MAPPING_LINK, TAU_MAPPING_OBJECT, TAU_MAPPING_PROFILE_START, TAU_MAPPING_PROFILE_STOP

Appendix A. Environment Variables

Table A.1. TAU Environment Variables

VARIABLE NAME	DESCRIPTION
PAPI_EVENT	Sets the hardware counter to use when TAU is configured with -PAPI. See Section 4.6, “Using Hardware Performance Counters”
PCL_EVENT	Sets the hardware counter to use when TAU is configured with -PCL. See Section 4.6, “Using Hardware Performance Counters”
PROFILEDIR	Selectively measure groups of routines and statements. Use with -profile command line option. See Chapter 4, *Profiling*
TAU_CALLPATH_DEPTH	Sets the depth of the callpath profiling. Use with -PROFILECALLPATH TAU configuration option. See Section 1.1, “Installing TAU”
TAU_COMPENSATE_ITERATIONS	Set the number of iterations TAU uses to estimate the mesurment overhead. A larger number of iteration will increases profiling precision (default 1000).
TAU_KEEP_TRACEFILES	Retains the intermediate trace files. Use with -TRACE TAU configuration option. See Section 7.1, “Generating Event Traces”
TAU_MUSE_PACKAGE	Sets the MAGNET/MUSE package name. Use with the -muse TAU configuration option. See Section 4.7, “Using Multiple Hardware Counters for Measurement”
TAU_THROTTLE	Enables the runtime throttling of events that are lightweight. See Section 4.3, “Selectively Profiling an Application”
TAU_THROTTLE_NUMCALLS	Set the maximum number of calls that will be profiled for any function when TAU_THROTTLE is enabled. See Section 4.3, “Selectively Profiling an Application”
TAU_THROTTLE_PERCALL	Set the minimum inclusive time (in milliseconds) a fuction has to have to be instrumented when TAU_THROTTLE is enabled. See Section 4.3, “Selectively Profiling an Application”
TAU_TRACEFILE	Specifies the name of Vampir trace file. Use with -TRACE TAU configuration option. See Section 7.1, “Generating Event Traces”
TRACEDIR	Specifies the directory where trace file are to be stored. See Section 7.1, “Generating Event Traces”