TAU User Guide

TAU User Guide

Updated February 12, 2024, for use with version 2.33.1 or greater.

Table of Contents

List of Figures

List of Tables

TAU preface

TAU Performance System® is a portable profiling and tracing toolkit for performance analysis of parallel programs written in Fortran, C, C++, Java, and Python. TAU (Tuning and Analysis Utilities) is capable of gathering performance information through instrumentation of functions, methods, basic blocks, and statements. The TAU API also provides selection of profiling groups for organizing and controlling instrumentation. Calls to the TAU API are made by probes inserted into the execution of the application via source transformation, compiler directives or by library interposition.

This guide is organized into different sections. Readers wanting to get started right way can skip to the Common Profile Requests section for step-by-step instructions for obtaining difference kinds of performance data. Or browse the starters guide for a quick reference to common TAU commands and variables.

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

Part I. Tau User Guide

Table of Contents

Chapter 1. Tau Instrumentation

1.1. Types of Instrumenation

TAU provides three methods to track the performance of your application. Library interposition using tau_exec, compiler directives or source transformation using PDT. Here is a table that lists the features/ requirement for each method:

Table 1.1. Different methods of instrumenting applications

Method	**Requires recompiling**	**Requires PDT**	**Shows MPI events**	**Routine-level event**	**Low level events (loops, phases, etc...)**	**Throttling to reduce overhead**	**Ability to exclude file from instrumentation**
Interposition			Yes			Yes	
Compiler	Yes		Yes	Yes		Yes	Yes
Source	Yes	Yes	Yes	Yes	Yes	Yes	Yes

The requirements for each method increases as we move down the table: tau_exec only requires a system with shared library support. Compiler based instrumentation requires re-compiling that target application and Source instrumentation aditionally requires PDT. For this reason we often recommend that users start with Library interposition and move down the table if more features are needed.

1.2. Dynamic instrumentation through library pre-loading

Dynamic instrumentation is achieved through library pre-loading. The libraries chosen for pre-loading determine the scope of instrumentation. Some options include tracking MPI, io, memory, cuda, opencl library calls. MPI instrumentation is included by default the others are enabled by command-line options to `tau_exec`. More info at the `tau_exec` manual page. Dynamic instrumentation can be used on both uninstrumented binaries and binaries instrumented via one of the methods below, in this way different layers of instrumentation can be combined.

To use `tau_exec` place this command before the application executable when running the application. In this example IO instrumentation is requested.

```
%> tau_exec -io ./a.out
%> mpirun -np 4 tau_exec -io ./a.out
```

1.3. TAU scripted compilation

1.3.1. Instrumenation

For more detailed profiles, TAU provides two means to compile your application with TAU: through your compiler or through source transformation using PDT.

1.3.2. Compiler Based Instrumentation

TAU provides these scripts: tau_cc.sh, tau_cxx.sh, tau_upc.sh, tau_f77.sh and tau_f90.sh to compile programs. You might use tau_cc.sh to compile a C program by typing:

```
%> module load tau
%> tau_cc.sh -tau_options=-optCompInst samplecprogram.c
```

On machines where a TAU module is not available, you will need to set the tau makefile and/or options. The makefile and options controls how will TAU will compile you application. Use

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

The Makefile can be found in the `/[arch]/lib` directory of your TAU distribution, for example `/x86_64/lib/Makefile.tau-mpi-pdt`.

You can also 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]
%>export TAU_OPTIONS=-optCompInst
%>tau_cc.sh sampleCprogram.c
```

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

1.3.3. Source Based Instrumentation

TAU provides these scripts: tau_cc.sh, tau_cxx.sh, tau_upc.sh, tau_f77.sh and tau_f90.sh to instrument and compile programs. You might use tau_cc.sh to compile a C program by typing:

```
%> module load tau
%> tau_cc.sh samplecprogram.c
```

When setting the TAU_MAKEFILE make sure the Makefile name contains `pdt` because you will need a version of TAU built with PDT.

A list of options for the TAU compiler scripts can be found by typing `man tau_compiler.sh` or in this chapter of the reference guide.

1.3.4. Options to TAU compiler scripts

These are some commonly used options available to the TAU compiler scripts. Either set them via the `TAU_OPTIONS` environment variable or the `-tau_options=` option to `tau_cc.sh`,

`tau_cxx.sh, tau_upc.sh, tau_f77.sh and tau_f90.sh`

`-optVerbose`	Enable verbose output (default: on)
`-optKeepFiles`	Do not remove intermediate files
`-optShared`	Use shared library of TAU (consider when using `tau_exec`

1.4. Selectively Profiling an Application

1.4.1. Custom Profiling

TAU allows you to customize the instrumentation of a program by using a selective instrumentation file. This instrumentation file is used to manually control which parts of the application are profiled and how they are profiled. If you are using one of the TAU compiler wrapper scripts to instrument your application you can use the `-tau_options=-optTauSelectFile=<file>` option to enable selective instrumentation.

Note

Selective instrumentation may be specified at runtime by setting the TAU_SELECT_FILE environment variable to the location of a valid selective instrumentation file in the instrumented application's run environment.

To specify a selective instrumentation file, create a text file and use the following guide to fill it in:

- Wildcards for routine names are specified with the # mark (because * symbols show up in routine signatures.) The # mark is unfortunately the comment character as well, so to specify a leading wildcard, place the entry in quotes.
- Wildcards for file names are specified with * symbols.

Here is a example file:

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

void quicksort(int *, int, int)
# The next line excludes all functions beginning 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

BEGIN_INSTRUMENT_SECTION

# A dynamic phase will break up the profile into phase where
```

```
# each events is recorded according to what phase of the application
# in which it occured.
dynamic phase name="foo1_bar" file="foo.c" line=26 to line=27

# instrument all the outer loops in this routine
loops file="loop_test.cpp" routine="multiply"

# tracks memory allocations/deallocations as well as potential leaks
memory file="foo.f90" routine="INIT"

# tracks the size of read, write and print statements in this routine
io file="foo.f90" routine="RINB"

END_INSTRUMENT_SECTION
```

Selective instrumentation files can be created automatically from `ParaProf` by right clicking on a trial and selecting the `Create Selective Instrumentation File` menu item.

Chapter 2. Profiling

This chapter describes running an instrumented application, generating profile data and analyzing that 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.

2.1. 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. By default, profiles are placed in the current directory. You can also set the TAU_VERBOSE enviroment variable to see the steps the TAU measurement systems takes when your application is running. Example:

```
% setenv TAU_VERBOSE 1
% setenv PROFILEDIR /home/sameer/profiledata/experiment55
% mpirun -np 4 matrix
```

Other environment variables you can set to enable these advanced MPI measurement features are TAU_TRACK_MESSAGE to track MPI message statistics when profiling or messages lines when tracing, and TAU_COMM_MATRIX to generate MPI communication matrix data.

2.2. Reducing Performance Overhead with TAU_THROTTLE

TAU automatically throttles short running functions in an effort to reduce the amount of overhead associated with profiles of such functions. This feature may be turned off by setting the environment variable TAU_THROTTLE to 0. The default rules TAU uses to determine which functions to throttle is: `numcalls > 100000 && usecs/call < 10` which means that if a function executes more than 100000 times and has an inclusive time per call of less than 10 microseconds, then 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_NUMCALLS 2000000
% setenv TAU_THROTTLE_PERCALL  5
```

The changes the values to 2 million and 5 microseconds per call. Functions that are throttled are marked explicitly in there names as THROTTLED.

2.3. Profiling each event callpath

You can enable callpath profiling by setting the environment variable TAU_CALLPATH. In this mode TAU will recorded the each event callpath to the depth set by the TAU_CALLPATH_DEPTH environment variable (default is two). Because instrumentation overhead will increase with the depth of the callpath, you should use the shortest call path that is sufficient.

2.4. Using Hardware Counters for Measurement

Performance counters exist on many modern microprocessors. They can 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 [http://icl.cs.utk.edu/papi/])` package provides a uniform interface to access these performance counters.

To use these counters, you must first find out which PAPI events your system supports. To do so type:

```
%> papi_avail
Available events and hardware information.
--------------------------------------------------------------------------------
Vendor string and code   : AuthenticAMD (2)
Model string and code    : AMD K8 Revision C (15)
CPU Revision             : 2.000000
CPU Megahertz            : 2592.695068
CPU's in this Node       : 4
Nodes in this System     : 1
Total CPU's              : 4
Number Hardware Counters : 4
Max Multiplex Counters   : 32
--------------------------------------------------------------------------------
The following correspond to fields in the PAPI_event_info_t structure.

Name            Code            Avail   Deriv   Description (Note)
PAPI_L1_DCM     0x80000000      Yes     Yes     Level 1 data cache misses
PAPI_L1_ICM     0x80000001      Yes     Yes     Level 1 instruction cache misses
...
```

Next, to test the compatibility between each metric you wish papi to profile, use `papi_event_chooser`:

```
papi/utils> papi_event_chooser PAPI_LD_INS PAPI_SR_INS PAPI_L1_DCM
Test case eventChooser: Available events which can be added with given
events.
-----------------------------------------
Vendor string and code   : GenuineIntel (1)
Model string and code    : Itanium 2 (2)
CPU Revision             : 1.000000
CPU Megahertz            : 1500.000000
CPU's in this Node       : 16
Nodes in this System     : 1
Total CPU's              : 16
Number Hardware Counters : 4
Max Multiplex Counters   : 32
-----------------------------------------
Event PAPI_L1_DCM can't be counted with others
```

Here the event chooser tells us that PAPI_LD_INS, PAPI_SR_INS, and PAPI_L1_DCM are incompatible metrics. Let try again this time removing PAPI_L1_DCM:

```
% papi/utils> papi_event_chooser PAPI_LD_INS PAPI_SR_INS
Test case eventChooser: Available events which can be added with given
events.
-----------------------------------------
Vendor string and code   : GenuineIntel (1)
```

```
Model string and code    : Itanium 2 (2)
CPU Revision             : 1.000000
CPU Megahertz            : 1500.000000
CPU's in this Node       : 16
Nodes in this System     : 1
Total CPU's              : 16
Number Hardware Counters : 4
Max Multiplex Counters   : 32
-------------------------------------------
Usage: eventChooser NATIVE|PRESET evt1 evet2 ...
```

Here the event chooser verifies that PAPI_LD_INS and PAPI_SR_INS are compatible metrics.

Next, make sure that you are using a makefile with `papi` in its name. Then set the environment variable `TAU_METRICS` to a colon delimited list of PAPI metrics you would like to use.

```
setenv TAU_METRICS PAPI_FP_OPS\:PAPI_L1_DCM
```

In addition to PAPI counters, we support TIME (via unix gettimeofday). On Linux and CrayCNL systems, we provide the high resolution LINUXTIMERS metric and on BGL/BGP systems we provide BGLTIMERS and BGPTIMERS.

Chapter 3. Tracing

Typically, profiling shows the distribution of execution time across routines. It can show the code locations associated with specific bottlenecks, but it can 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.

3.1. Generating Event Traces

To enable tracing with TAU, set the environment variable `TAU_TRACE` to 1. Similarly you can enable/disable profile with the `TAU_PROFILE` variable. Just like with profiling, you can set the output directory with a environment variable:

```
% setenv TRACEDIR /users/sameer/tracedata/experiment56
```

This will generate a trace file and an event file for each processor. To merge these files, use the `tau_treemerge.pl` script. If you want to convert TAU trace file into another format use the `tau2otf`, `tau2vtf`, or `tau2slog2` scripts.

Chapter 4. Analyzing Parallel Applications

4.1. Text summary

For a quick view summary of TAU performance, use `pprof` It reads and prints a summary of the TAU data in the current directory. For performance data with multiple metrics, move into one of the directories to get information about that metric:

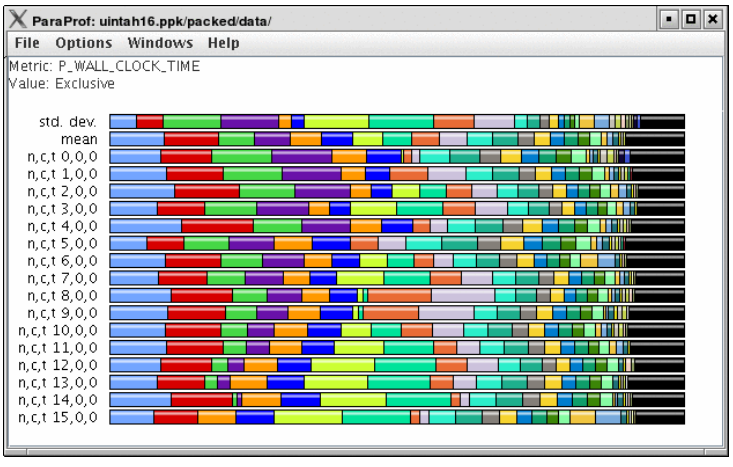
```
%> cd MULTI__P_WALL_CLOCK_TIME
%> pprof
Reading Profile files in profile.*

NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0           24          590           1           1     590963 main
 95.9           26          566           1           2     566911 multiply
 47.3          279          279           1           0     279280 multiply-opt
 44.1          260          260           1           0     260860 multiply-regula
```

4.2. ParaProf

To launch ParaProf, execute paraprof from the command line where the profiles are located. Launching ParaProf will bring up the manager window and a window displaying the profile data as shown below.

Figure 4.1. Main Data Window



For more information see the ParaProf section in the reference guide.

4.3. Jumpshot

To use Argonne's Jumpshot (bundled with TAU), first merge and convert TAU traces to slog2 format:

```
% tau_treemerge.pl
% tau2slog2 tau.trc tau.edf -o tau.slog2
% jumpshot tau.slog2
```

Launching Jumpshot will bring up the main display window showing the entire trace, zoom in to see more detail.

Figure 4.2. Main Data Window

Chapter 5. Quick Reference

`tau_run`
TAU's binary instrumentation tool

`tau_cc.sh -tau_options=-optCompInst / tau_cxx.sh -tau_options=-optCompInst / tau_f90.sh -tau_options=-optCompInst / tau_upc.sh -tau_options=-optCompInst / tau_f77.sh -tau_options=-optCompInst`
(Compiler instrumentation)

`tau_cc.sh / tau_cxx.sh / tau_f90.sh / tau_f77.sh / tau_upc.sh`
(PDT instrumentation)

`TAU_MAKEFILE`
Set instrumentation definition file

`TAU_OPTIONS`
Set instrumentation options

`dynamic phase name='name' file='filename' line=start_line_# to line=end_line_#`
Specify dynamic Phase

`loops file='filename' routine='routine name'`
Instrument outer loops

`memory file='filename' routine='routine name'`
Track memory

`io file='filename' routine='routine name'`
Track IO

`TAU_PROFILE / TAU_TRACE`
Enable profiling and/or tracing

`PROFILEDIR / TRACEDIR`
Set profile/trace output directory

`TAU_CALLPATH=1 / TAU_CALLPATH_DEPTH`
Enable Callpath profiling, set callpath depth

`TAU_THROTTLE=1 / TAU_THROTTLE_NUMCALLS / TAU_THROTTLE_PERCALL`
Enable event throttling, set number of call, percall (us) threshold

`TAU_METRICS`
List of PAPI metrics to profile

`tau_treemerge.pl`
Merge traces to one file

`tau2otf / tau2vtf / tau2slog2`
Trace conversion tools

Chapter 6. Some Common Application Scenario

6.1. Q. What routines account for the most time? How much?

A. Create a flat profile with wallclock time.

Figure 6.1. Flat Profile

Here is how to generate a flat profile with MPI

```
% setenv TAU_MAKEFILE /opt/apps/tau/tau2/x86_64/lib/Makefile.tau-mpi-pdt-pgi

% set path=(/opt/apps/tau/tau2/x86_64/bin $path)
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% qsub  run.job
% paraprof -–pack app.ppk
        Move the app.ppk file to your desktop.

% paraprof app.ppk
```

6.2. Q. What loops account for the most time? How much?

A. Create a flat profile with wallclock time with loop instrumentation.

Figure 6.2. Flat Profile with Loops

Here is how to instrument loops in an application

```
% setenv TAU_MAKEFILE /opt/apps/tau/tau2/x86_64/lib/Makefile.tau-mpi-pdt
% setenv TAU_OPTIONS '-optTauSelectFile=select.tau –optVerbose'
% cat select.tau
  BEGIN_INSTRUMENT_SECTION
  loops routine="#"
  END_INSTRUMENT_SECTION

% set path=(/opt/apps/tau/tau2/x86_64/bin $path)
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% qsub  run.job
```

```
% paraprof --pack app.ppk
        Move the app.ppk file to your desktop.

% paraprof app.ppk
```

6.3. Q. What MFlops am I getting in all loops?

A. Create a flat profile with PAPI_FP_INS/OPS and time with loop instrumentation.

Figure 6.3. MFlops per loop

Here is how to generate a flat profile with FP operations

```
% setenv TAU_MAKEFILE /opt/apps/tau/tau2/x86_64/lib/Makefile.tau-papi-mpi-pdt-pgi
% setenv TAU_OPTIONS '-optTauSelectFile=select.tau –optVerbose'
% cat select.tau
  BEGIN_INSTRUMENT_SECTION
  loops routine="#"
  END_INSTRUMENT_SECTION

% set path=(/opt/apps/tau/tau2/x86_64/bin $path)
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% setenv TAU_METRICS GET_TIME_OF_DAY\:PAPI_FP_INS
% qsub  run.job
% paraprof --pack app.ppk
        Move the app.ppk file to your desktop.
% paraprof app.ppk
  Choose 'Options' -> 'Show Derived Panel' -> Arg 1 = PAPI_FP_INS, Arg 2 =
        GET_TIME_OF_DAY, Operation = Divide -> Apply, close.
```

6.4. Q. Who calls MPI_Barrier() Where?

A. Create a callpath profile with given depth.

Figure 6.4. Callpath Profile

Here is how to generate a callpath profile with MPI

```
% setenv TAU_MAKEFILE
% /opt/apps/tau/tau2/x86_64/lib/Makefile.tau-mpi-pdt
% set path=(/opt/apps/tau/tau2/x86_64/bin $path)
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% setenv TAU_CALLPATH 1
% setenv TAU_CALLPATH_DEPTH 100

% qsub  run.job
% paraprof --pack app.ppk
```

```
        Move the app.ppk file to your desktop.
% paraprof app.ppk
(Windows -> Thread -> Call Graph)
```

6.5. Q. How do I instrument Python Code?

A. Create an python wrapper library.

Here to instrument python code

```
% setenv TAU_MAKEFILE /opt/apps/tau/tau2/x86_64/lib/Makefile.tau-icpc-python-mpi-p
% set path=(/opt/apps/tau/tau2/x86_64/bin $path)
% setenv TAU_OPTIONS '-optShared -optVerbose'
(Python needs shared object based TAU library)
% make F90=tau_f90.sh CXX=tau_cxx.sh CC=tau_cc.sh  (build pyMPI w/TAU)
% cat wrapper.py
  import tau
  def OurMain():
      import App
  tau.run('OurMain()')
Uninstrumented:
% mpirun.lsf /pyMPI-2.4b4/bin/pyMPI ./App.py
Instrumented:
% setenv PYTHONPATH<taudir>/x86_64/<lib>/bindings-python-mpi-pdt-pgi
(same options string as TAU_MAKEFILE)
setenv LD_LIBRARY_PATH <taudir>/x86_64/lib/bindings-icpc-python-mpi-pdt-pgi\:$LD_L
% mpirun –np 4 <dir>/pyMPI-2.4b4-TAU/bin/pyMPI ./wrapper.py
(Instrumented pyMPI with wrapper.py)
```

6.6. Q. What happens in my code at a given time?

A. Create an event trace.

Figure 6.5. Tracing with Vampir

How to create a trace

```
% setenv TAU_MAKEFILE
% /opt/apps/tau/tau2/x86_64/lib/Makefile.tau-mpi-pdt-pgi
% set path=(/opt/apps/tau/tau2/x86_64/bin $path)
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% setenv TAU_TRACE 1
% qsub  run.job
% tau_treemerge.pl
(merges binary traces to create tau.trc and tau.edf files)
JUMPSHOT:
% tau2slog2 tau.trc tau.edf –o app.slog2
% jumpshot app.slog2
   OR
```

```
VAMPIR:
% tau2otf tau.trc tau.edf app.otf –n 4 –z
(4 streams, compressed output trace)
% vampir app.otf
(or vng client with vngd server).
```

6.7. Q. How does my application scale?

A. Examine profiles in PerfExplorer.

Figure 6.6. Scalability chart

How to examine a series of profiles in PerfExplorer

```
% setenv TAU_MAKEFILE /opt/apps/tau/tau2/x86_64/lib/Makefile.tau-mpi-pdt
% set path=(/opt/apps/tau/tau2/x86_64/bin $path)
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% qsub  run1p.job
% paraprof --pack 1p.ppk
% qsub run2p.job
% paraprof --pack 2p.ppk ...and so on.
On your client:
% taudb_configure --create-default
(taudb_configure run without any arguments will prompt for advanced options)
% perfexplorer_configure
(Yes to load schema, defaults)
% paraprof
(load each trial: Right click on trial ->Upload trial to DB
% perfexplorer
(Charts -> Speedup)
```

Part II. ParaProf - User's Manual

Table of Contents

Chapter 7. Introduction

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

Important

ParaProf requires *Oracle / Sun's* Java 1.5 Runtime Environment for basic functionality. Java JOGL (included) 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.

7.1. 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, ompp
                                    snap, perixml, gptl, ipm, google
  --range a-b:c                   Load only profiles from the given range(s) of pr
                                    Seperate individual ids or dash-defined ranges
  -h, --help                      Display this help message

The following options will run only from the console (no GUI will launch):

  --merge <file.gz>               Merges snapshot profiles
  --pack <file>                   Pack the data into packed (.ppk) format
  --dump                          Dump profile data to TAU profile format
  --dumprank <rank>               Dump profile data for <rank> to TAU profile form
  -v, --dumpsummary               Dump derived statistical data to TAU profile for
  --overwrite                     Allow overwriting of profiles
  -o, --oss                       Print profile data in OSS style text output
  -q, --dumpmpisummary            Print high level time and communication summary
  -d, --metadump                  Print profile metadata (works with --dumpmpisumm
  -x, --suppressmetrics           Exclude child calls and exclusive time from --du
  -s, --summary                   Print only summary statistics
                                    (only applies to OSS output)

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.
```

7.2. Supported Formats

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

- **TAU Profiles (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.
- **ParaProf Packed Format (ppk)** - Export format supported by PerfDMF/ParaProf. Typically .ppk.
- **TAU Merged Profiles (snap)** - Merged and snapshot profile format supported by TAU. Typically tauprofile.xml.
- **TAU pprof (pprof)** - Dump Output from TAU's **pprof -d**. Provided for backward compatibility only.
- **DynaProf (dynaprof)** - Output From DynaProf's wallclock and papi probes.
- **mpiP (mpip)** - Output from mpiP.
- **gprof (gprof)** - Output from gprof, see also the --fixnames option.
- **PerfSuite (psrun)** - Output from PerfSuite psrun files.
- **HPM Toolkit (hpm)** - Output from IBM's HPM Toolkit.
- **Cube (cube)** - Output from Kojak Expert tool for use with Cube.
- **Cube3 (cube3)** - Output from Kojak Expert tool for use with Cube3 and Cube4.
- **HPCToolkit (hpc)** - XML data from hpcquick. Typically, the user runs hpcrun, then hpcquick on the resulting binary file.
- **OpenMP Profiler (ompp)** - CSV format from the ompP OpenMP Profiler (http://www.ompp-tool.com). The user must use OMPP_OUTFORMAT=CVS.
- **PERI XML (perixml)** - Output from the PERI data exchange format.
- **General Purpose Timing Library (gptl)** - Output from the General Purpose Timing Library.
- **Paraver (paraver)** - 2D output from the Paraver trace analysis tool from BSC.
- **IPM (ipm)** - Integrated Performance Monitoring format, from NERSC.
- **Google (google)** - Google Profiles.

7.3. 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.
- **--oss** - Outputs profile data in OSS Style. Example:

```
-------------------------------------------------------------------------------
Thread: n,c,t 0,0,0
-------------------------------------------------------------------------------
 excl.secs  excl.%   cum.%    PAPI_TOT_CYC     PAPI_FP_OPS     calls  function
     0.005   56.0%   56.0%        13475345         4194518         1  foo
     0.003   40.1%   96.1%         9682185         4205367         1  bar
         0    3.6%   99.7%          223173           17445         1  baz
   2.2E-05    0.3%  100.0%           14663             206         1  main
```

- **--summary** - Output only summary information for OSS style output.

Chapter 8. Views and Sub-Views

In the past, PerfDMF used a hierarchy of Applications and Experiments to organize Trials. This approach was too rigid, so in TAUdb, trials are organized by dynamic Views. Views are lists of Trials that share a given metadata value. For example, a View could contain all the Trials where the total number of threads is less than 16. Views can also have Sub-Views. For example, it might be useful to have a View of all Trials from a certain machine and then Sub-Views for each executable ran on that machine. Trials can belong to any number of VIews and Sub-Views and new Trials loaded to the database will be sorted into Views automatically.

8.1. To Create a (Sub-)Views

Launch ParaProf and Right click on a database or an existing View and select "Add View" or "Add Sub-View."

Figure 8.1. Add View

In ParaProf and PerfExplorer, Views are marked by the Folder Icon and Trials are now marked with a yellow ball. (The "All Trials" View is created when a database is created.)

This will launch the View Creator window.

Figure 8.2. View Creator Window

Here you can create the rule(s) for which Trials appear in this new View. At the top you can choose to match all of the rules ("and") or to match any of the rules. The "-" or "=" buttons will remove the current rule or add a new one. The first drop down box chooses which metadata field to use. The second box chooses whether the field should be read as a string or a number. Depending on whether it is read as a string or a number, the fourth box will give options on how to compare the metadata field. So to create a View for all trials that have less than 16 threads, select total_threads, read as a string, is less than, 16. Then click Save and give the View a name.

The 'Edit' context menu option on an existing view will allow you to view and alter the view's criteria in the same interface.

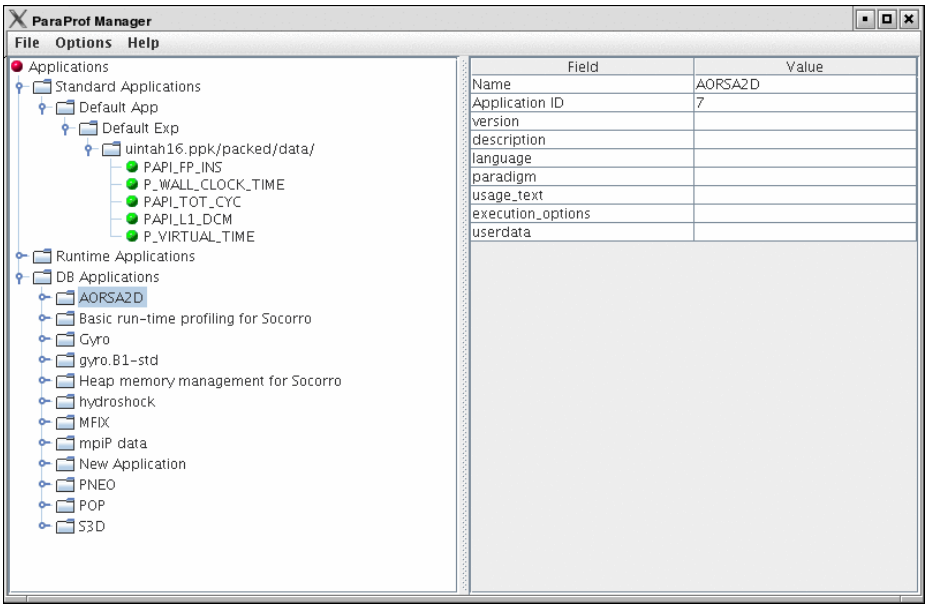
Chapter 9. Profile Data Management

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.

9.1. ParaProf Manager Window

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

Figure 9.1. ParaProf Manager Window

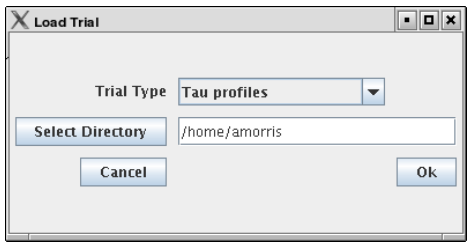


This window is used to manage profile data. The user can upload/download profile data, edit meta-data, launch visual displays, export data, derive new metrics, etc.

9.2. Loading Profiles

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

Figure 9.2. 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.

9.3. Database Interaction

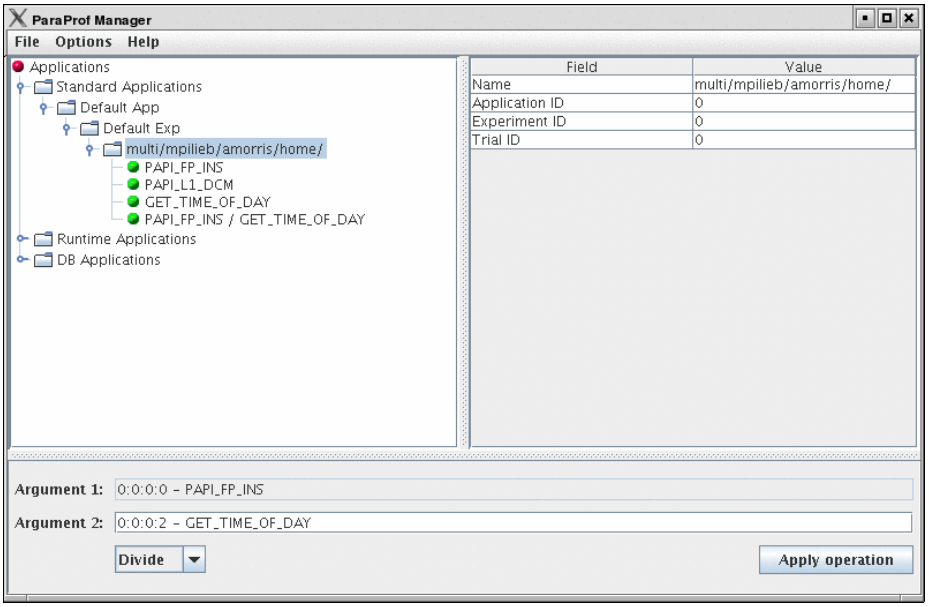
Database interaction is done through the tree view of the ParaProf Manager Window. Applications expand to Experiments, Experiments to Trials, and Trials are loaded directly into ParaProf just as if they were read off disk. Additionally, the meta-data associated with each element is show on the right, as in Figure 9.1, “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".

9.4. Creating Derived Metrics

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

Figure 9.3. Creating Derived Metrics

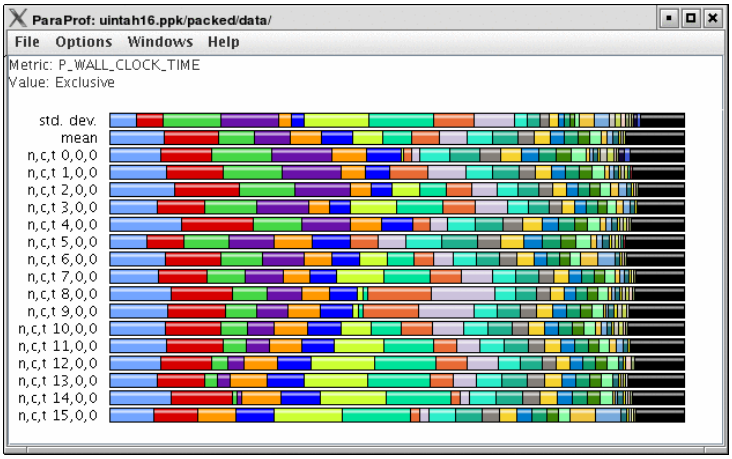


In Figure 9.3, “Creating Derived Metrics”, we have just divided Floating Point Instructions by Wall-clock time, creating FLOPS (Floating Point Operations per Second). The 2nd argument is a user editable text-box and can be filled in with scalar values by using the keyword 'val' (e.g. "val 1.5").

9.5. Main Data Window

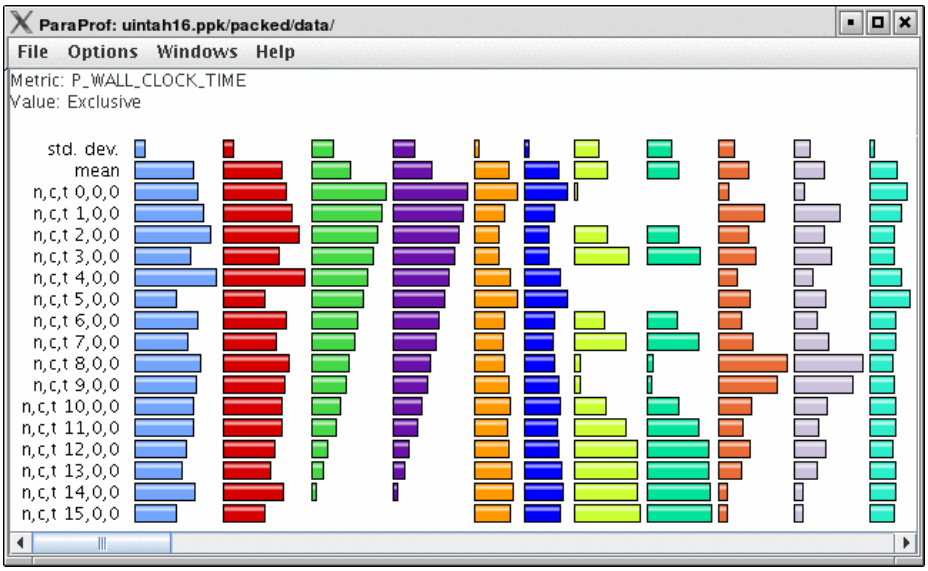
Upon loading a profile, or double-clicking on a metric, the *Main Data Window* will be displayed.

Figure 9.4. 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 9.4, “Main Data Window”, right click on the text *n,c,t, 8,0,0* to launch thread based displays for node 8.

Figure 9.5. Unstacked Bars



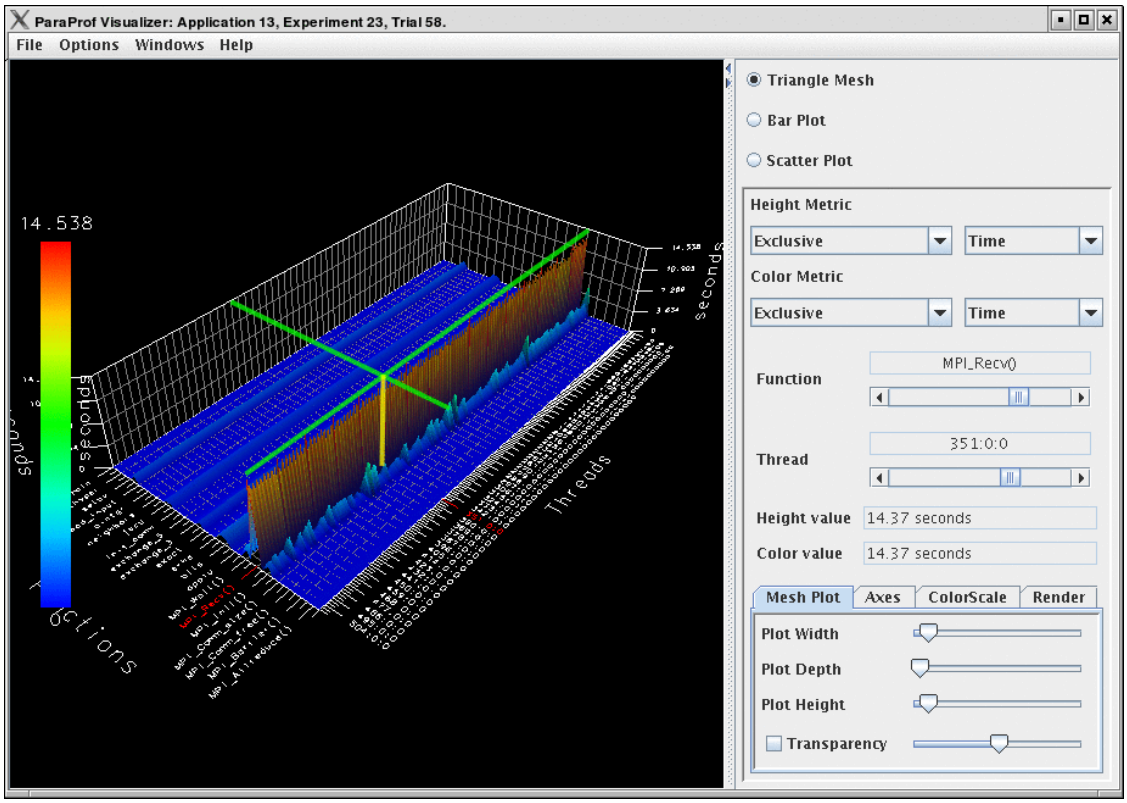
You may also turn off the stacking of bars so that individual functions can be compared across threads in a global display.

Chapter 10. 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.

10.1. Triangle Mesh Plot

Figure 10.1. 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.

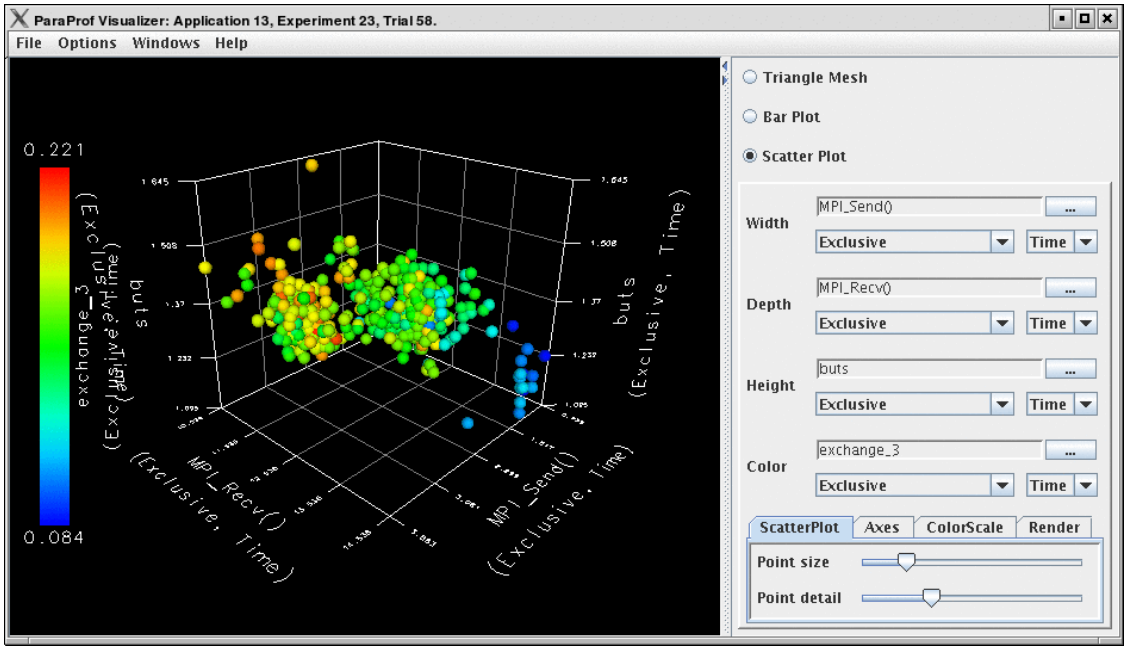
10.2. 3-D Bar Plot

Figure 10.2. 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 10.2, “3-D Mesh Plot” the transparency option is selected, which changes the way in which the selection model operates.

10.3. 3-D Scatter Plot

Figure 10.3. 3-D Scatter Plot

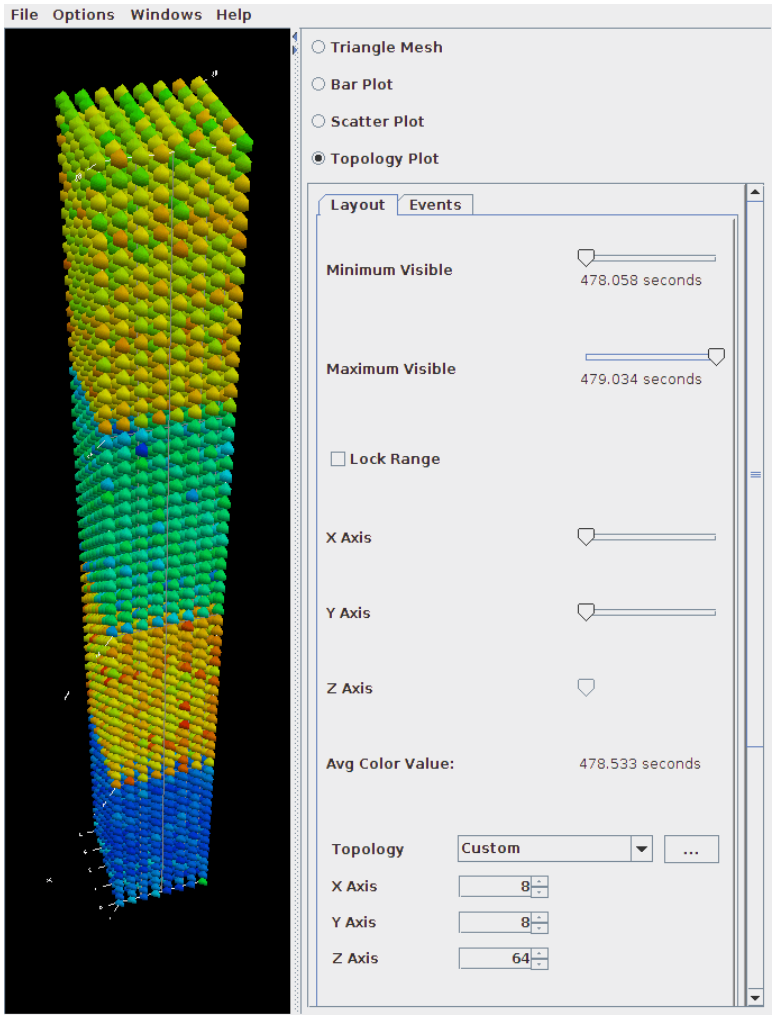


This visualization method plots the value of each thread along up to 4 axes. Each axis represents a different function and metric. This view allows you to discern clustering of values and relationships between functions across threads.

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

10.4. 3-D Topology Plot

Figure 10.4. 3-D Topology Plot



In this visualization, you can either define the layout with a MESP topology definition file or you can fill a rectangular prism of user-defined volume with rank-points in order of rank. For more information, please see the etc/topology directory for additional details on MESP topology definitions.

If the loaded profile is a cube file or a profile from a BGB, then this visualizations groups the threads in two or three dimensional space using topology information supplied by the profile.

When topology metadata is available a trial-specific topological layout may be visualized by selecting Windows->gt;3D Visualization and selecting Topology Plot on the visualization pane.

The layout tab allows control of the layout and display of visualized cores/processes.

Minimum/Maximum Visible (restricts display of nodes with measured values above/below the selected levels). Lock Range causes the sliders to move in unison.

The X/Y/Z Axis sliders allow selection of planes, lines and individual points in the topology for examination of specific values in the display, listed in the Avg. Color Value field.

The topology selection dropdown box allows selection of either trial-specific topologies contained in the metadata, mapped topologies stored in an external file or a custom topology defined by the size of the prism containing the visualized cores. The ... button allows selection of a custom topology mapping file while the map button allows selection of a map file (see <tau2>/etc/topology/README.cray_map for

more information on generating map files).

If a Custom is selected the dimensions of the rectangular prism containing the cores are defined by the X/Y/Z axis control widgets.

The Events tab controls which events are used to define the color values and positions of cores/processes in the display. For trail-specific and Custom topologies only event3(Color) can be changed. For topologies loaded in MESP definition files all four events may be used in calculation of the topology layout. In either case, interval, atomic or metadata values may be used to color or position points in the display.

10.5. 3-D Commication Matrix

Figure 10.5. 3-D Commication Matrix

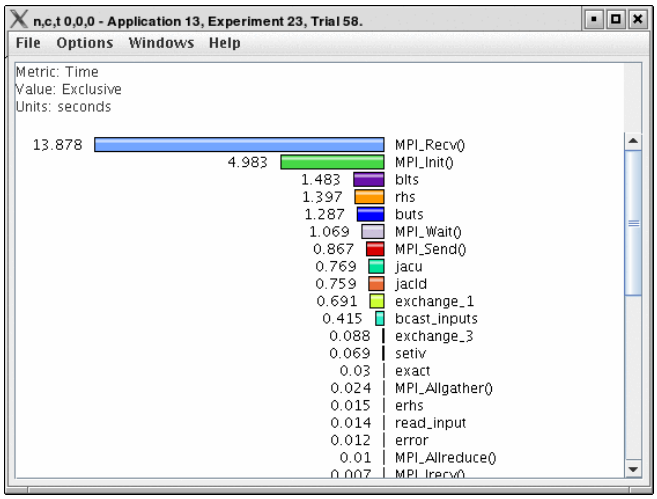
If a Trial has commication information (set TAU_COMM_MATRIX=1 at runtime of your application), then you can launch the 3D Commication window as shown.

Chapter 11. 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.

11.1. Thread Bar Graph

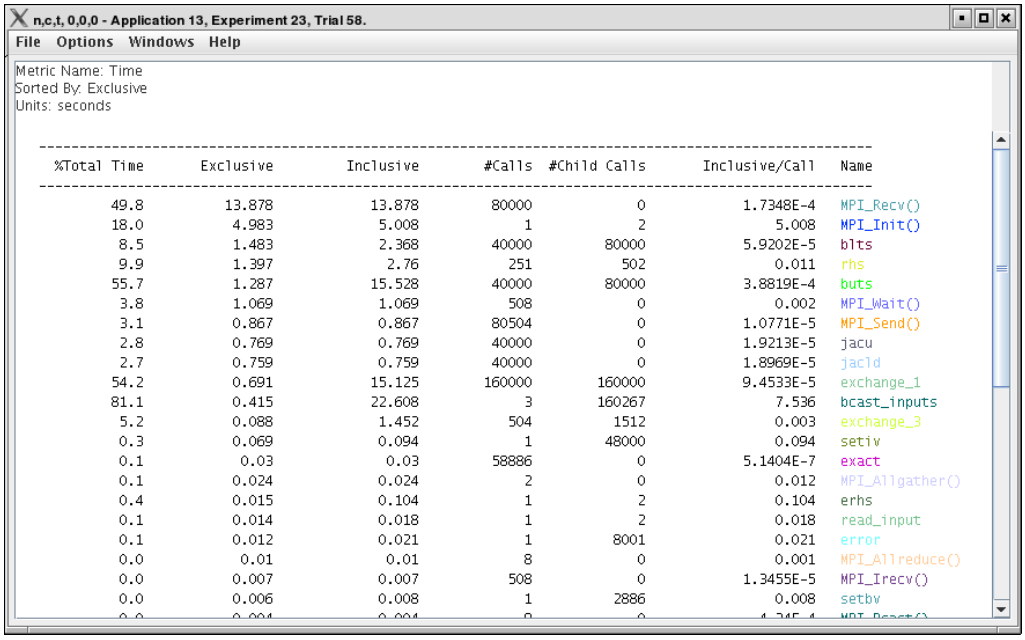
Figure 11.1. Thread Bar Graph



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

11.2. Thread Statistics Text Window

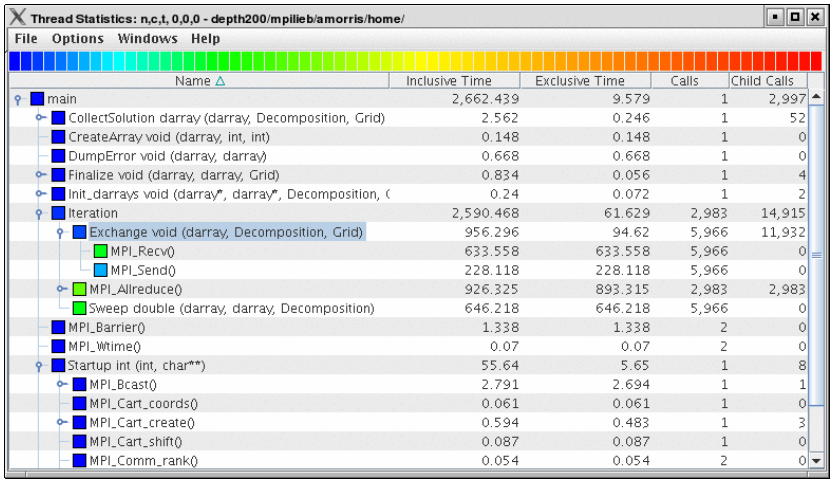
Figure 11.2. Thread Statistics Text Window



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

11.3. Thread Statistics Table

Figure 11.3. Thread Statistics Table, inclusive and 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 11.4. Thread Statistics Table

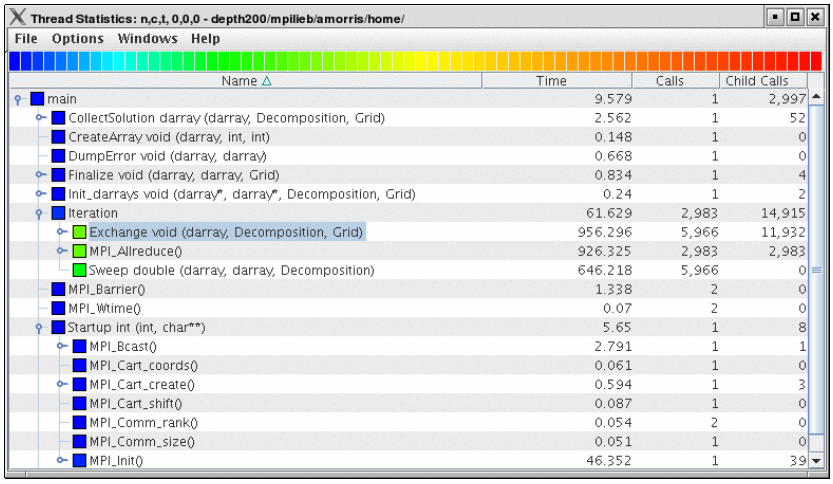
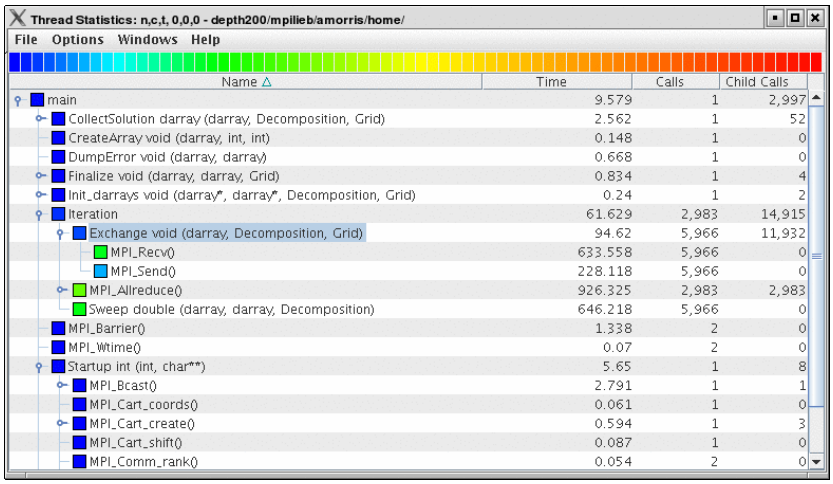


Figure 11.5. Thread Statistics Table

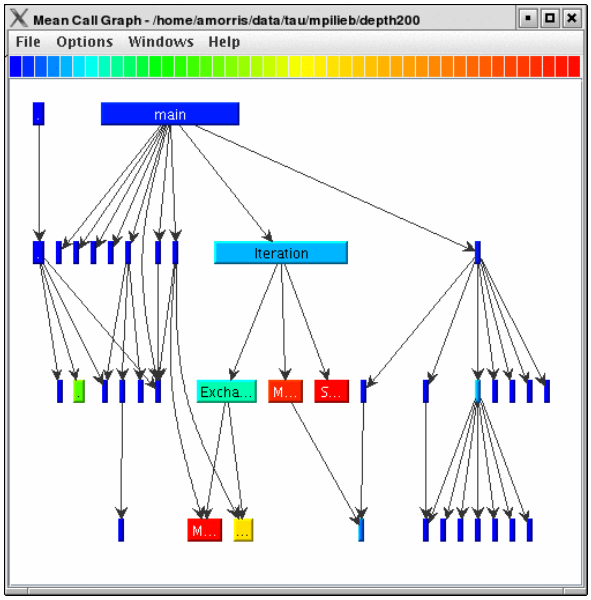


The display can be used in one of two ways, in "inclusive/exclusive" mode, both the inclusive and exclusive values are shown for each path, see Figure 11.3, “Thread Statistics Table, inclusive and exclusive” for an example.

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 11.4, “Thread Statistics Table” and Figure 11.5, “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.

11.4. Call Graph Window

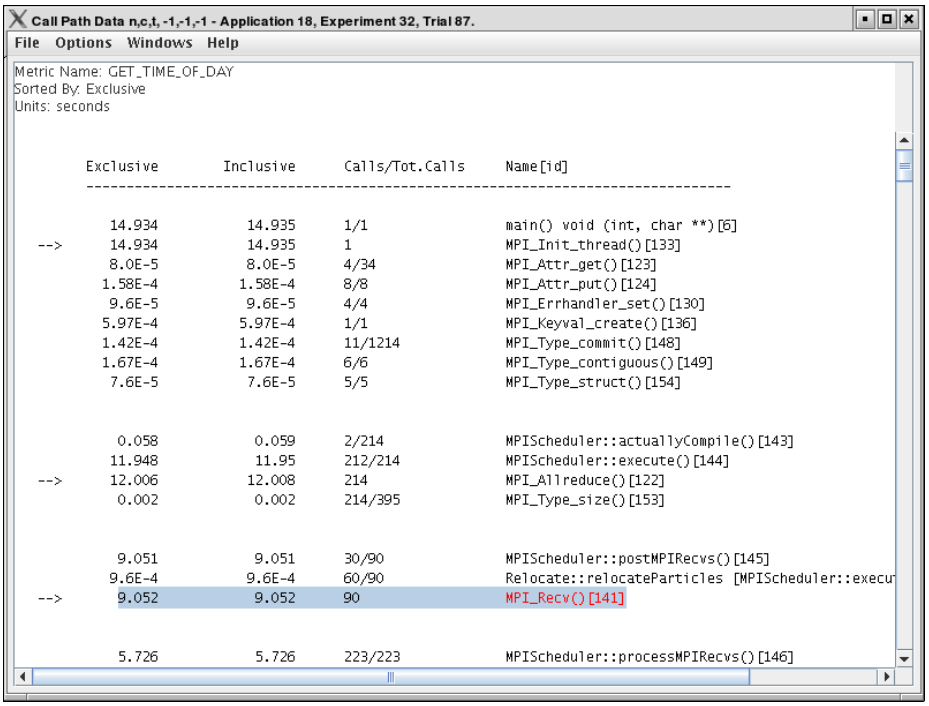
Figure 11.6. Call Graph Window



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.

11.5. Thread Call Path Relations Window

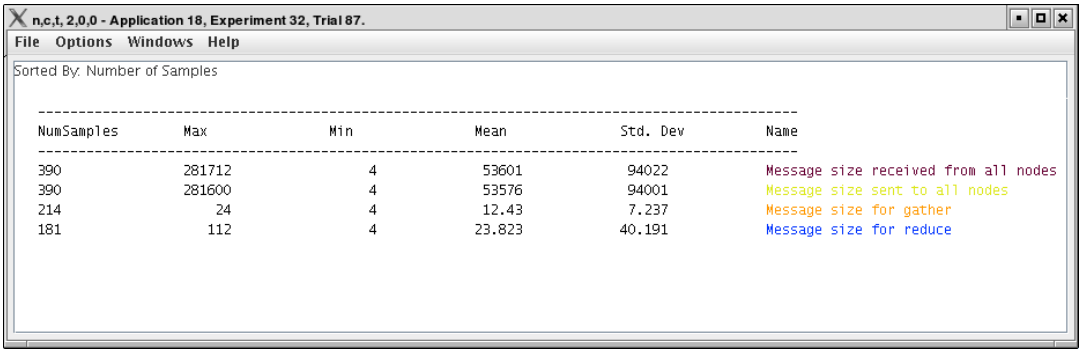
Figure 11.7. Thread Call Path Relations Window



This display shows callpath data in a **gprof** style view. Each function is shown with its immediate parents. For example, Figure 11.7, “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.

11.6. User Event Statistics Window

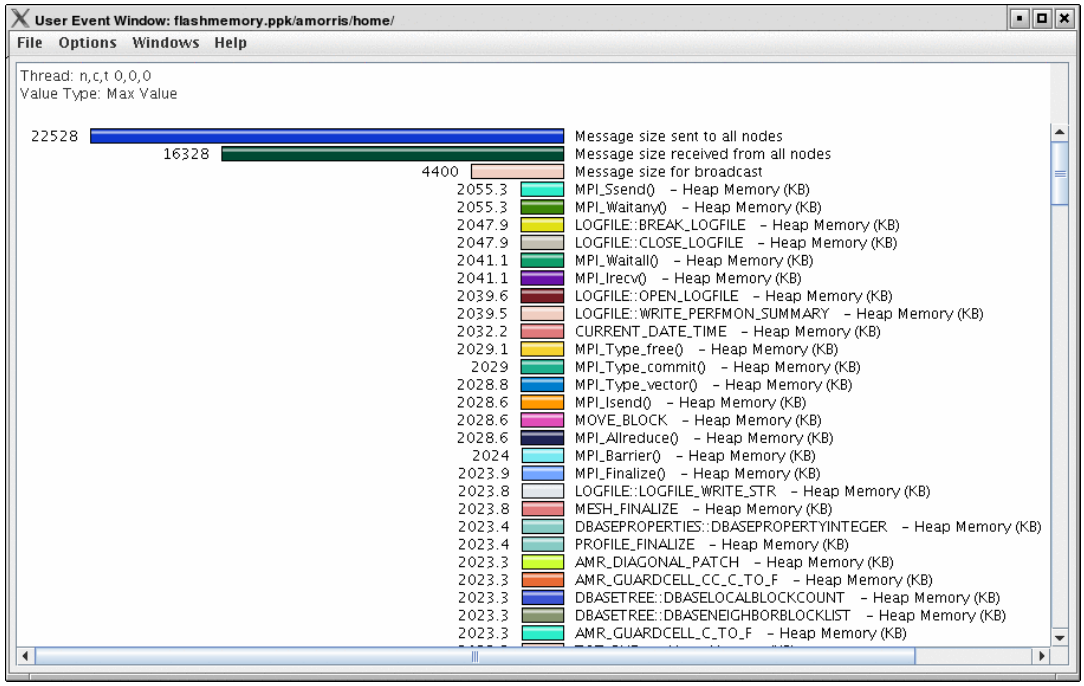
Figure 11.8. User Event Statistics Window



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 15.1, “User Event Bar Graph”

11.7. User Event Thread Bar Chart

Figure 11.9. User Event Thread Bar Chart Window



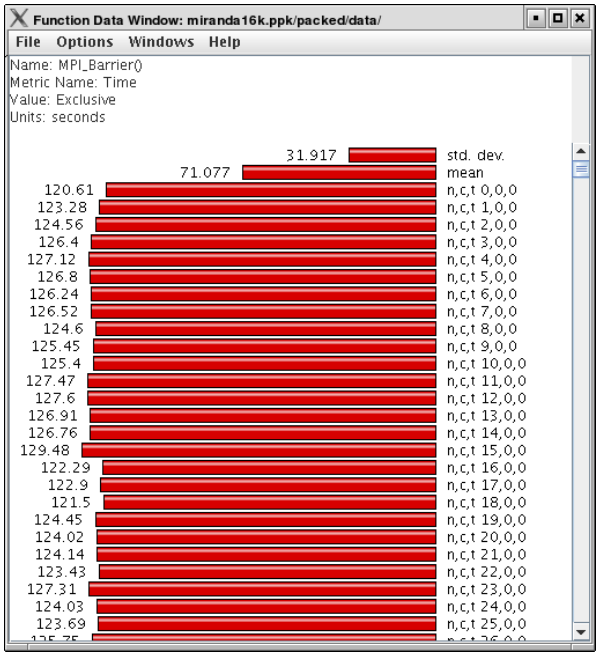
This display shows a particular thread's user defined event statistics as a bar chart. This is the same data from the Section 11.6, “User Event Statistics Window”, in graphical form.

Chapter 12. 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.

12.1. Function Bar Graph

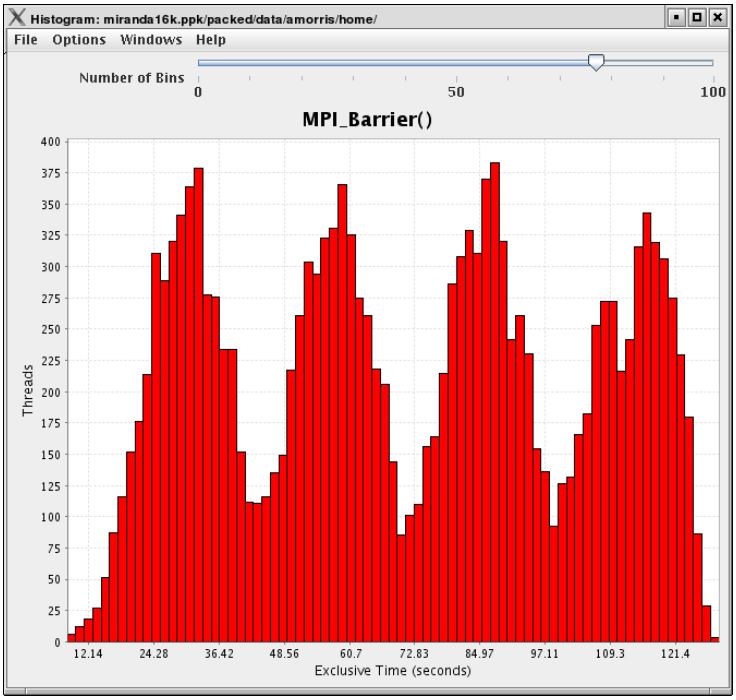
Figure 12.1. Function Bar Graph



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.

12.2. Function Histogram

Figure 12.2. 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.

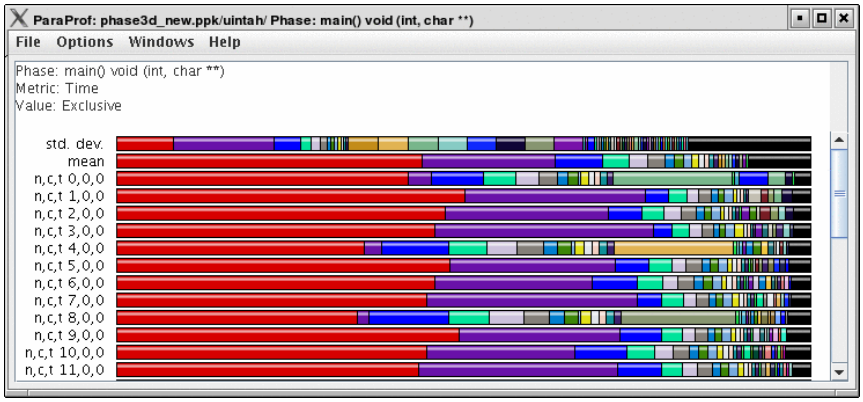
Chapter 13. Phase Based Displays

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.

13.1. Using Phase Based Displays

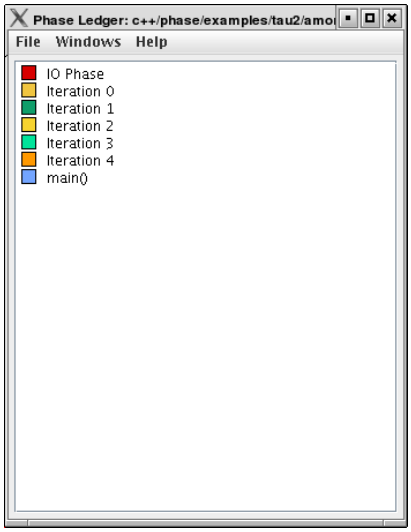
The initial window will default to top level phase, usually *main*

Figure 13.1. 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.

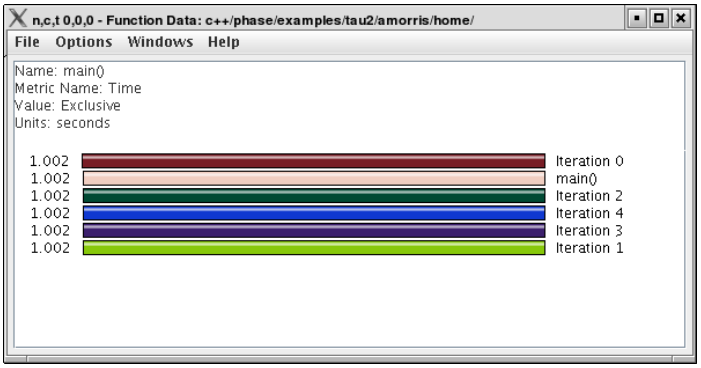
Figure 13.2. Phase Ledger



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 13.3. 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.

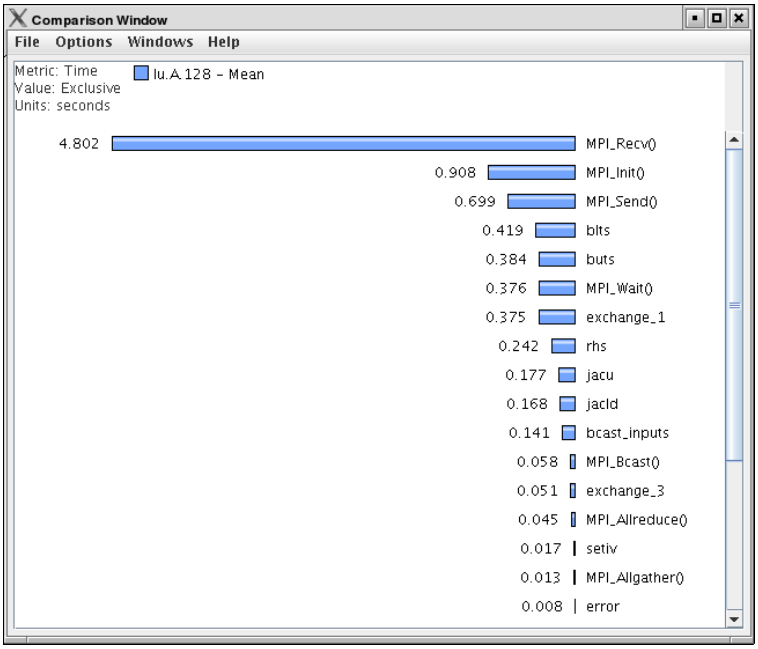
Chapter 14. Comparative Analysis

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.

14.1. 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 14.1. Comparison Window (initial)



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

Figure 14.2. Comparison Window (2 trials)

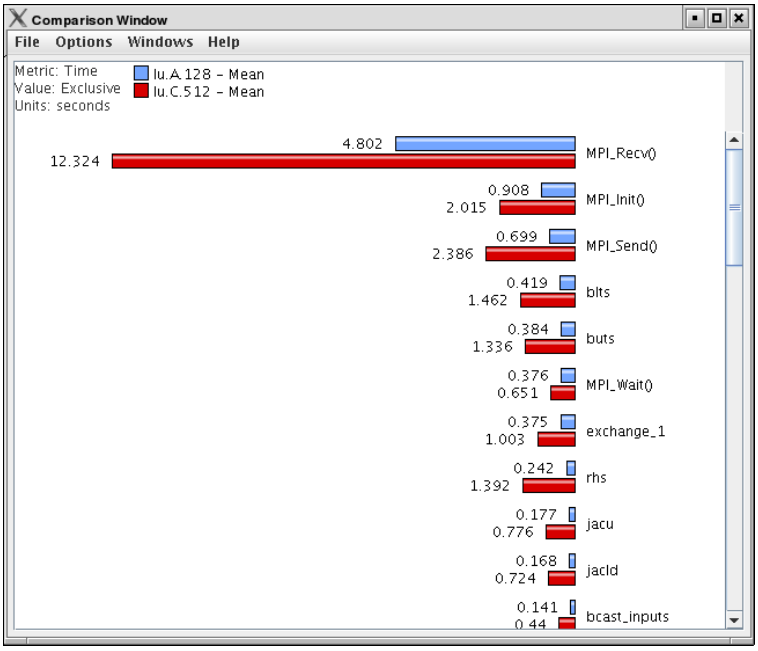
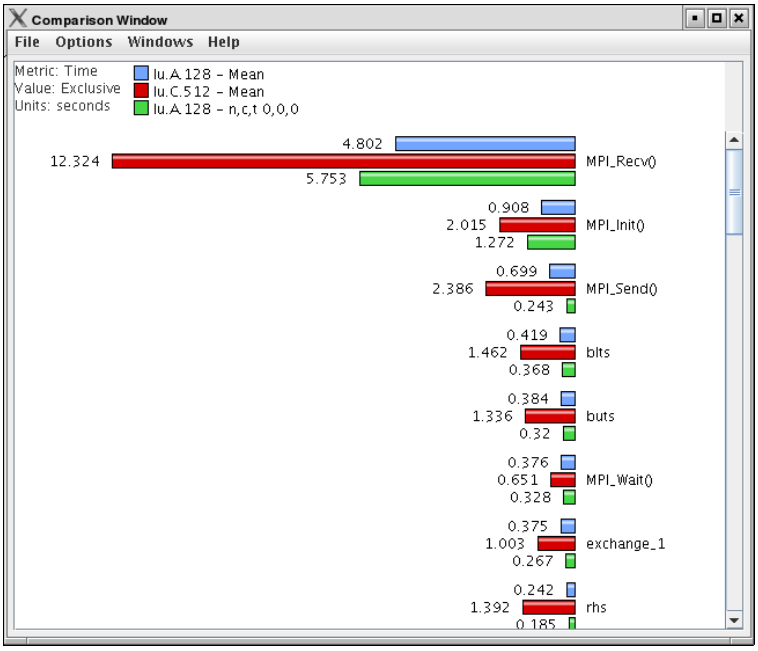


Figure 14.3. Comparison Window (3 threads)

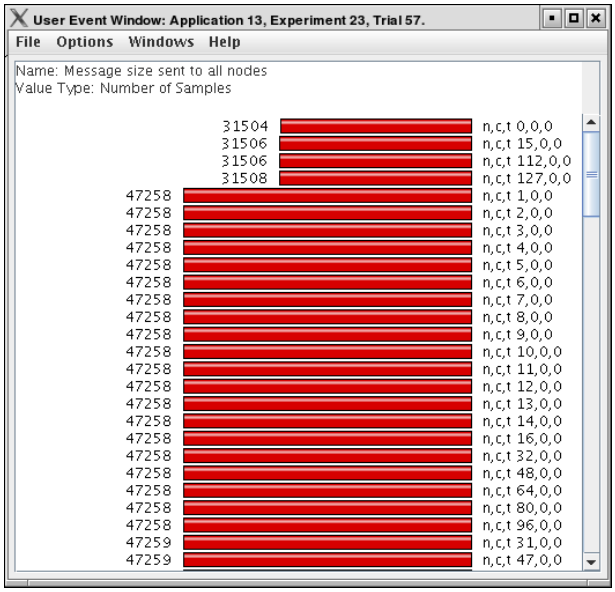


Chapter 15. Miscellaneous Displays

15.1. 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 15.1. User Event Bar Graph



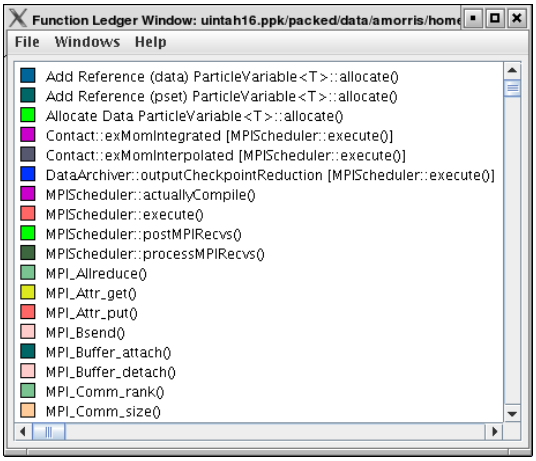
This display graphs the value that the particular user event had for each thread.

15.2. Ledgers

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

15.2.1. Function Ledger

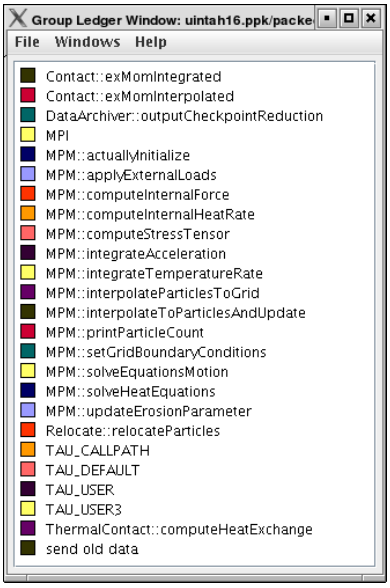
Figure 15.2. Function Ledger



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.

15.2.2. Group Ledger

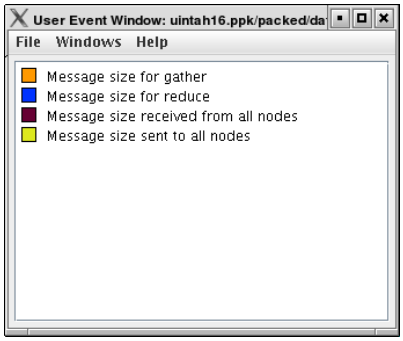
Figure 15.3. Group Ledger



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.

15.2.3. User Event Ledger

Figure 15.4. User Event Ledger

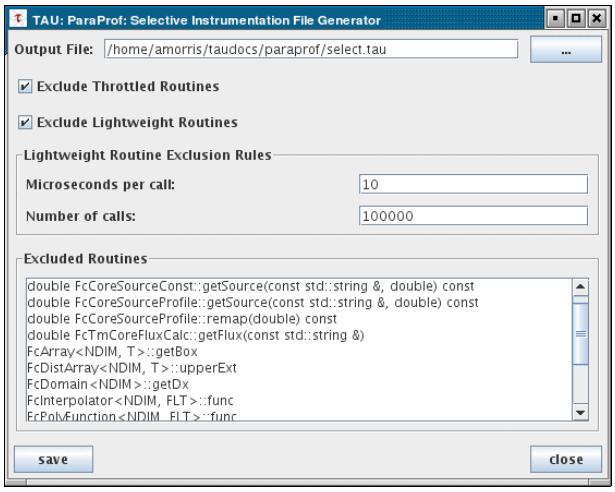


The user event ledger shows each user event along with its current color.

15.3. Selective Instrumentation File Generator

ParaProf can also help you refine your program performance by excluding some functions from instrumentation. You can select rules to determine which function get excluded; both rules must be true for a given function to be excluded. Below each function that will be excluded based on these rules are listed.

Figure 15.5. Selective Instrumentation Dialog



Note

Only the functions profilied in ParaProf can be excluded. If you had previously setup selective instrumentation for this application the functions that where previously excluded will not longer be excluded.

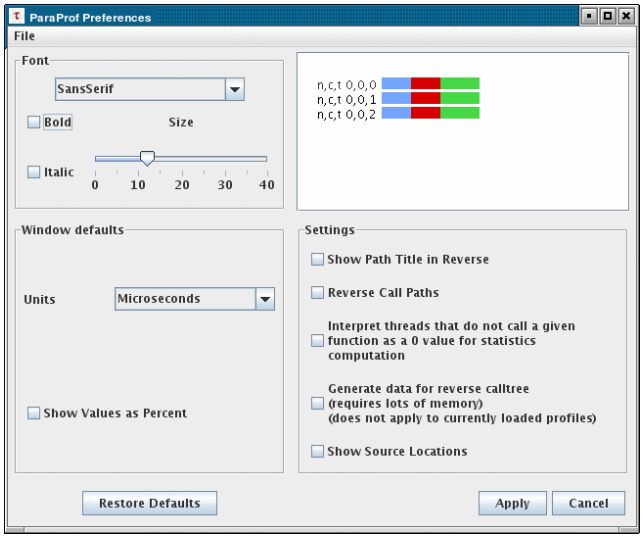
Chapter 16. Preferences

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

16.1. 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 16.1. 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

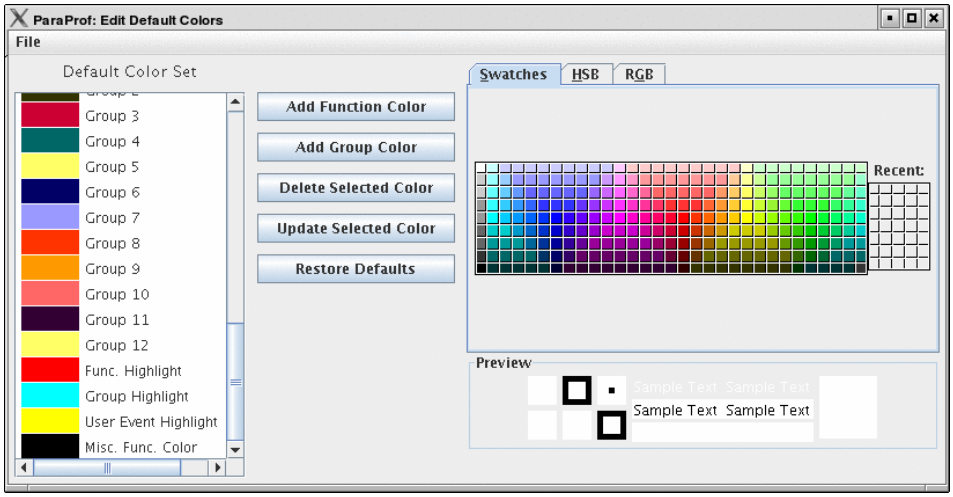
- Show Path Title in Reverse - Path title will normally be shown in normal order (/home/amorris/data/etc). They can be reverse using this option (etc/data/amorris/home). This only affects loaded trials and the titlebars of new windows.
- Reverse Call Paths - This option will immediately change the display 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.
- Generate Reverse Calltree Data - This option will enable the generation of reverse callpath data ne-

cessary for the reverse callpath option of the statistics tree-table window.

- Show Source Locations - This option will enable the display of source code locations in event names.

16.2. Default Colors

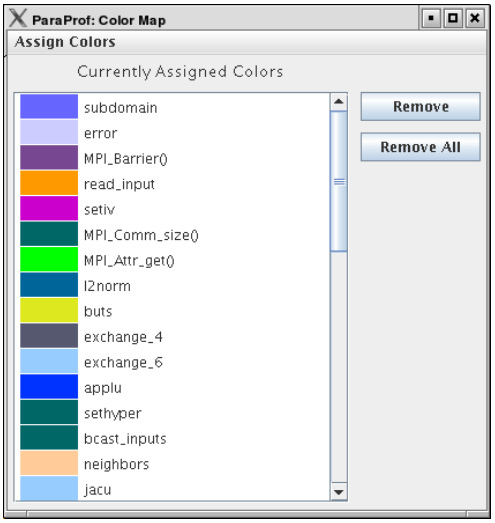
Figure 16.2. Edit Default Colors



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.

16.3. Color Map

Figure 16.3. 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. PerfExplorer - User's Manual

Table of Contents

Chapter 17. 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 18. Installation and Configuration

PerfExplorer uses TAUdb and PerfDMF databases so if you have not already you will need to install TAUdb, see Chapter 28, *Introduction*. After installing and configuring TAU the perfexplorer executable should be available in your `[path to tau]/tau2/[arch]/bin` directory. You will need to run `perfexplorer_configure`, installed at the same location as perfexplorer, to set up your database for use with perfexplorer and to download additional 3rd party jar files perfexplorer requires. When prompted by `perfexplorer_configure` give the name of your PerfDMF or TAUdb database and press Y to agree to download the jar files.

Chapter 19. 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 one or more applications, experiments, views or trials. To view these charts first choose a metric of interest by selecting a trial form the tree on the left. Then optionally choose the `Set Metric of Interest` or `Set Event of Interest` form the `Charts` menu (if you don't, and you need to, you will be prompted). Now you can view a chart by selecting it from the `Charts` menu.

Chapter 20. Cluster Analysis

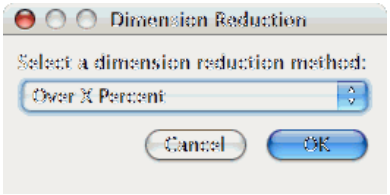
Cluster analysis is a valuable tool for reducing large parallel profiles down to representative groups for investigation. Currently, there are two types of clustering analysis implemented in PerfExplorer. Both *hierarchical* and *k-means* analysis are used to group parallel profiles into common clusters, 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.

20.1. 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".

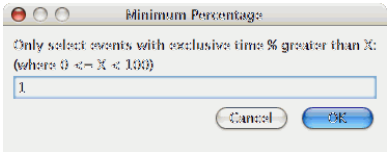
To reduce dimensions, select the "Select Dimension Reduction" item under the "Analysis" main menu bar item. The following dialog will appear:

Figure 20.1. Selecting a dimension reduction method



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

Figure 20.2. Entering a minimum threshold for exclusive percentage



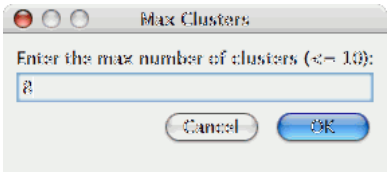
Enter a value, for example "1".

20.2. 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:

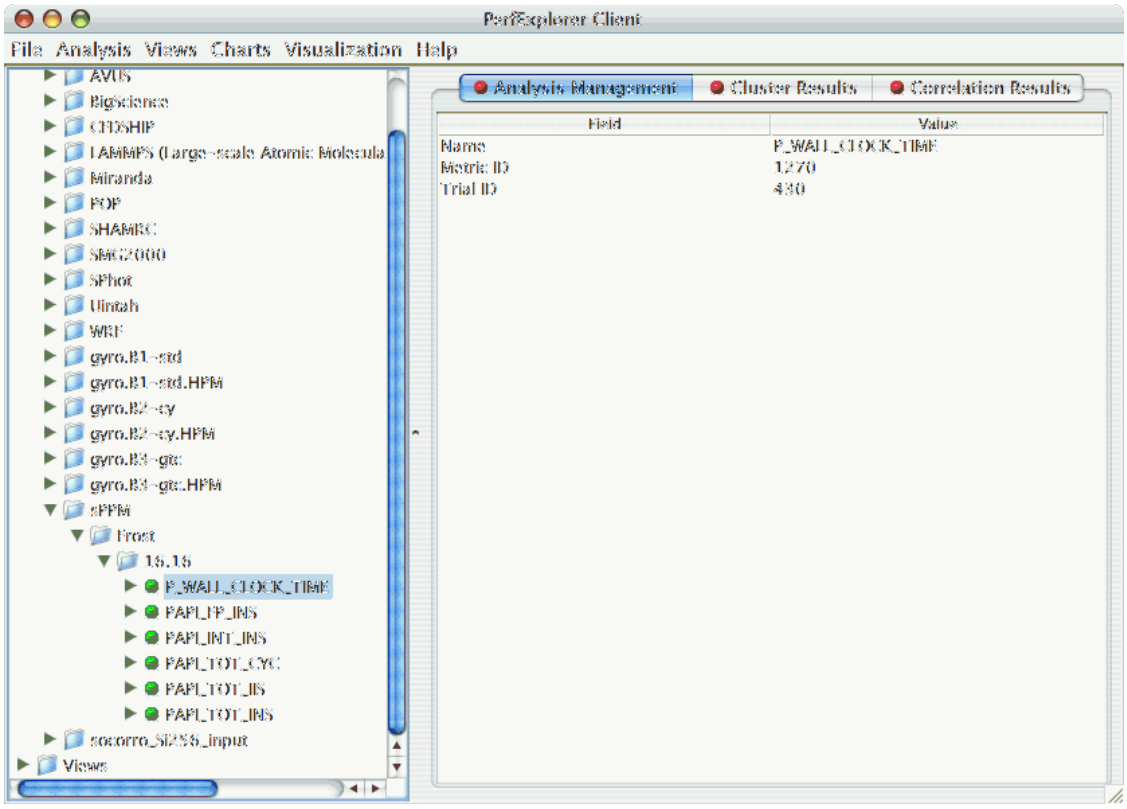
Figure 20.3. Entering a maximum number of clusters



20.3. Performing Cluster Analysis

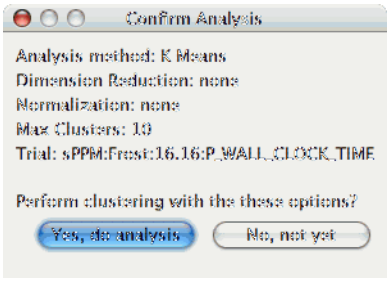
To perform cluster analysis, you first need to select a metric. To select a metric, navigate through the tree of applications, experiments and trials, and expand the trial of interest, showing the available metrics, as shown in the figure below:

Figure 20.4. Selecting a Metric to Cluster



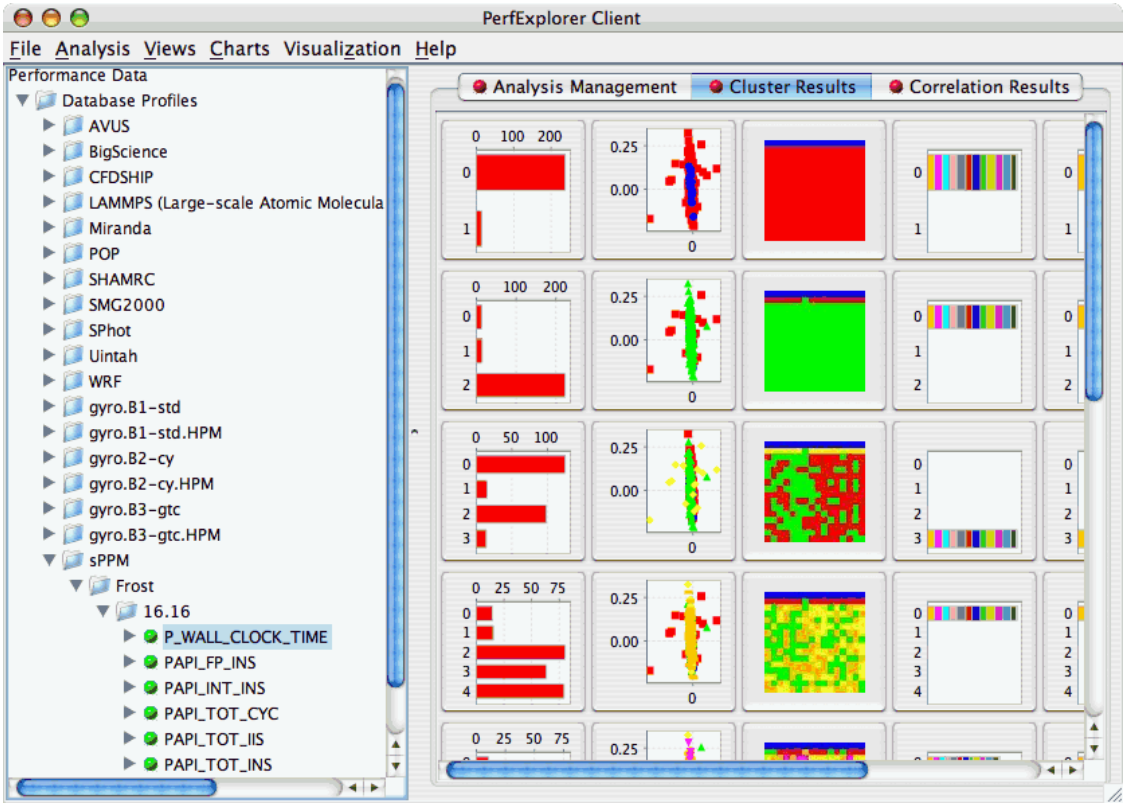
After selecting the metric of interest, select the "Do Clustering" item under the "Analysis" main menu bar item. The following dialog will appear:

Figure 20.5. 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 20.6. 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 minimum values for each event in each cluster, the average values for each event in each cluster, and the maximum 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 20.7. Cluster Membership Histogram

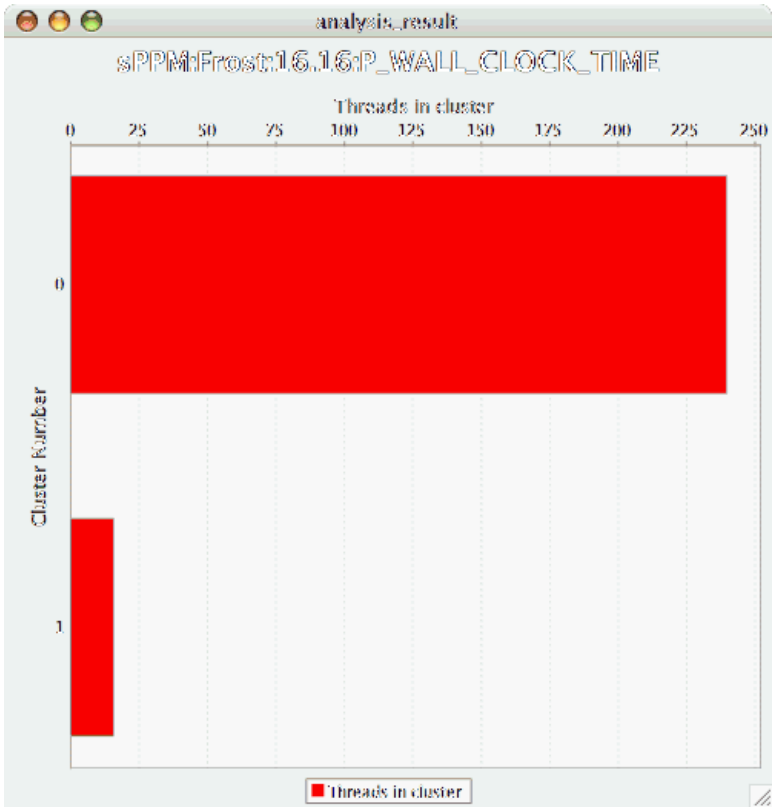


Figure 20.8. Cluster Membership Scatterplot

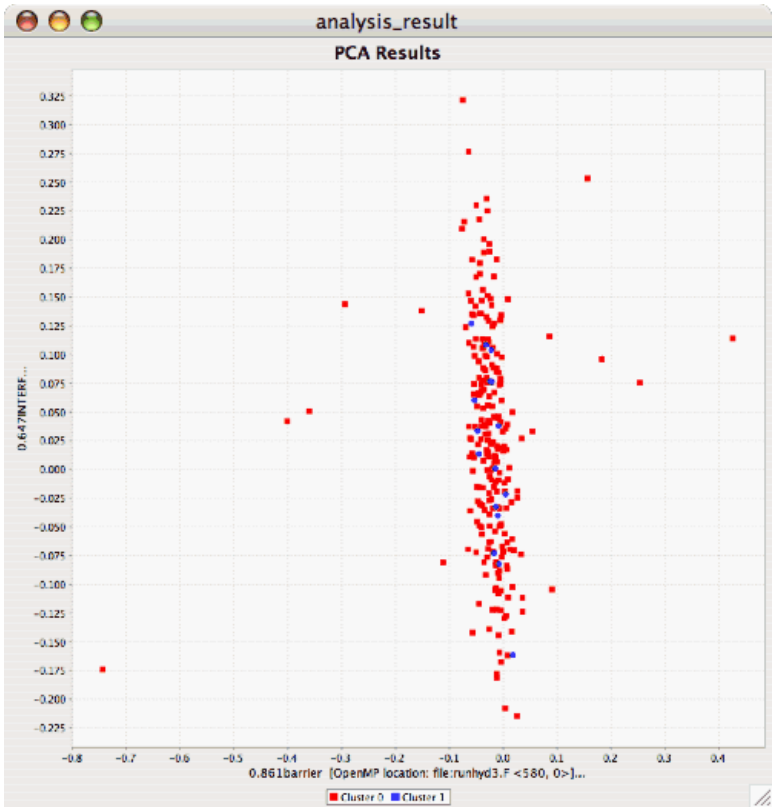


Figure 20.9. Cluster Virtual Topology

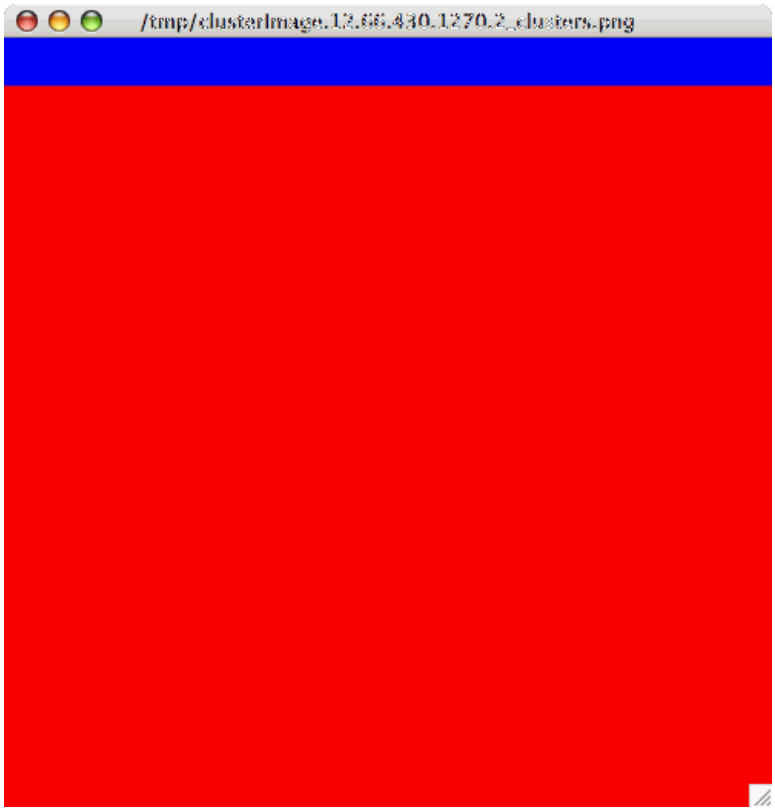
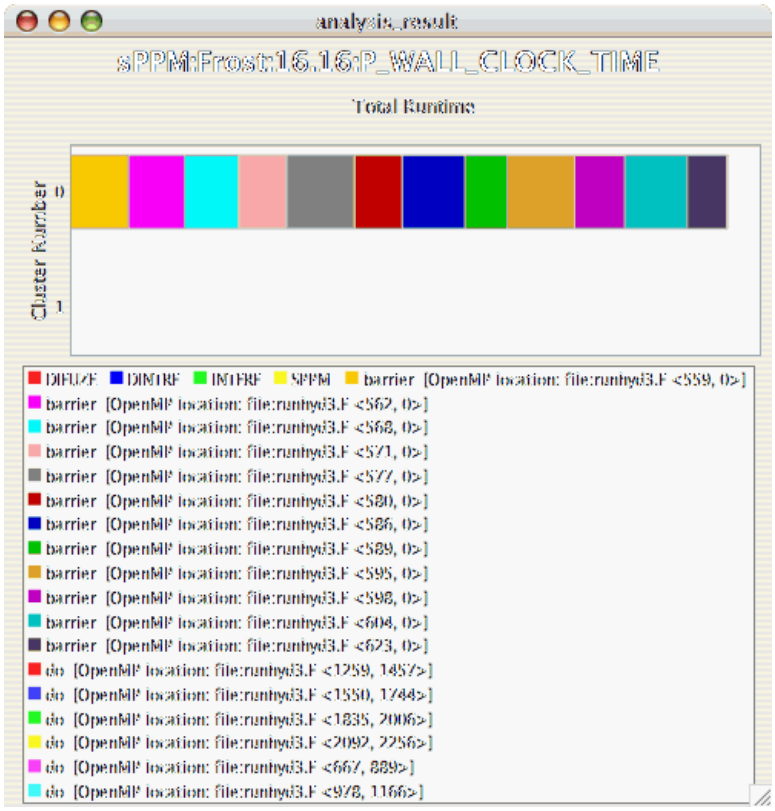


Figure 20.10. Cluster Average Behavior


analysis_result
sPPM:Frost:16.16:P_WALL_CLOCK_TIME
Total Runtime
Cluster Number
0
1
DIFUZE
DIMTRF
IMTRF
SPPM
barrier [OpenMP location: file:runhyd3.F <559, 0>]
barrier [OpenMP location: file:runhyd3.F <562, 0>]
barrier [OpenMP location: file:runhyd3.F <568, 0>]
barrier [OpenMP location: file:runhyd3.F <571, 0>]
barrier [OpenMP location: file:runhyd3.F <577, 0>]
barrier [OpenMP location: file:runhyd3.F <580, 0>]
barrier [OpenMP location: file:runhyd3.F <586, 0>]
barrier [OpenMP location: file:runhyd3.F <589, 0>]
barrier [OpenMP location: file:runhyd3.F <595, 0>]
barrier [OpenMP location: file:runhyd3.F <598, 0>]
barrier [OpenMP location: file:runhyd3.F <604, 0>]
barrier [OpenMP location: file:runhyd3.F <623, 0>]
do [OpenMP location: file:runhyd3.F <1259, 1457>]
do [OpenMP location: file:runhyd3.F <1550, 1748>]
do [OpenMP location: file:runhyd3.F <1835, 2006>]
do [OpenMP location: file:runhyd3.F <2092, 2256>]
do [OpenMP location: file:runhyd3.F <667, 889>]
do [OpenMP location: file:runhyd3.F <978, 1166>]

Chapter 21. Correlation Analysis

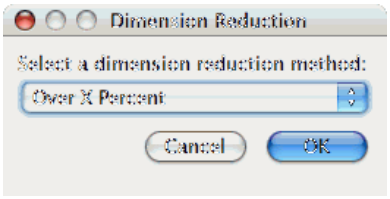
Correlation analysis in PerfExplorer is used to explore relationships between events in a profile. Each event is pairwise plotted with the other events, and a correlation coefficient is calcuated for the relationship. When the events are highly positively correlated (coefficient of close to 1.0) or highly negatively correlated (coefficient close to -1.0), then the relationships will show up as linear groupings in the results. Clusters may also be apparent.

21.1. 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".

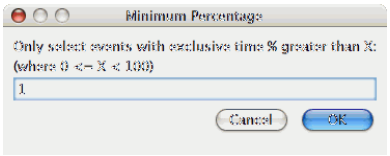
To reduce dimensions, select the "Select Dimension Reduction" item under the "Analysis" main menu bar item. The following dialog will appear:

Figure 21.1. Selecting a dimension reduction method



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

Figure 21.2. Entering a minimum threshold for exclusive percentage

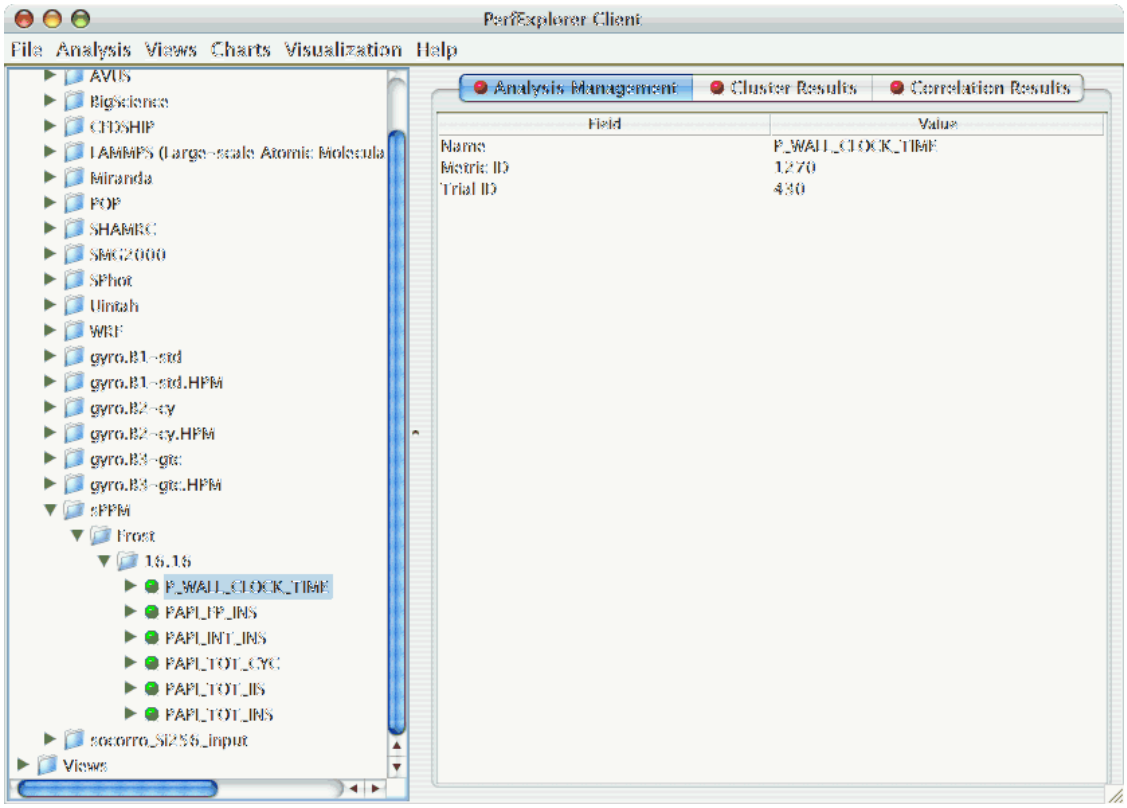


Enter a value, for example "1".

21.2. Performing Correlation Analysis

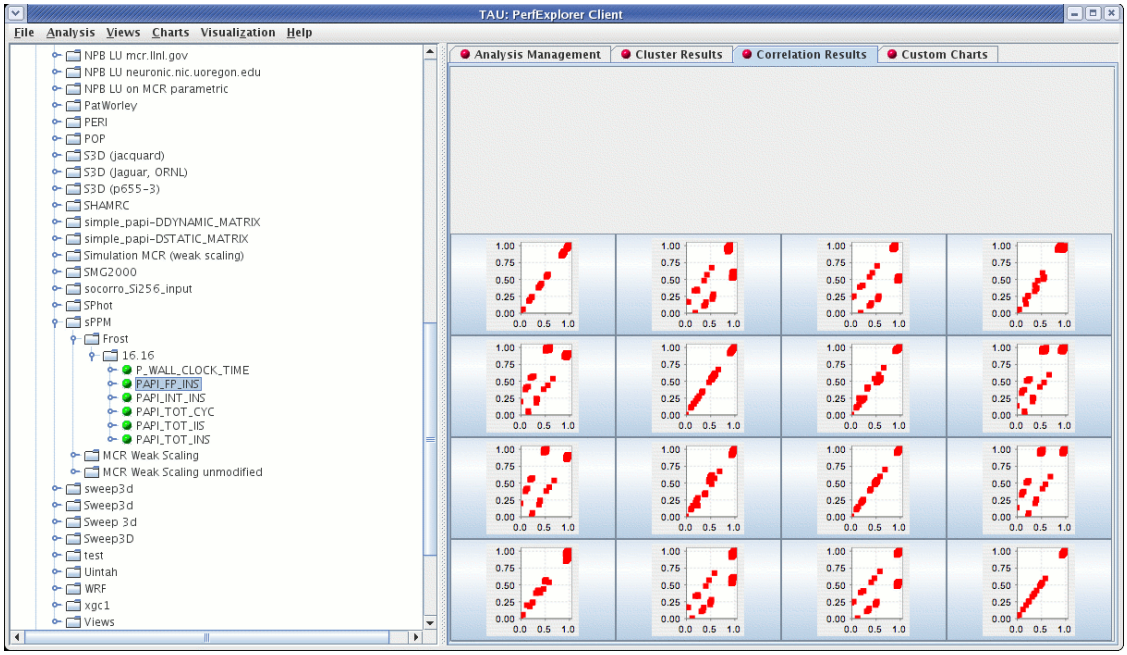
To perform correlation analysis, you first need to select a metric. To select a metric, navigate through the tree of applications, experiments and trials, and expand the trial of interest, showing the available metrics, as shown in the figure below:

Figure 21.3. Selecting a Metric to Cluster



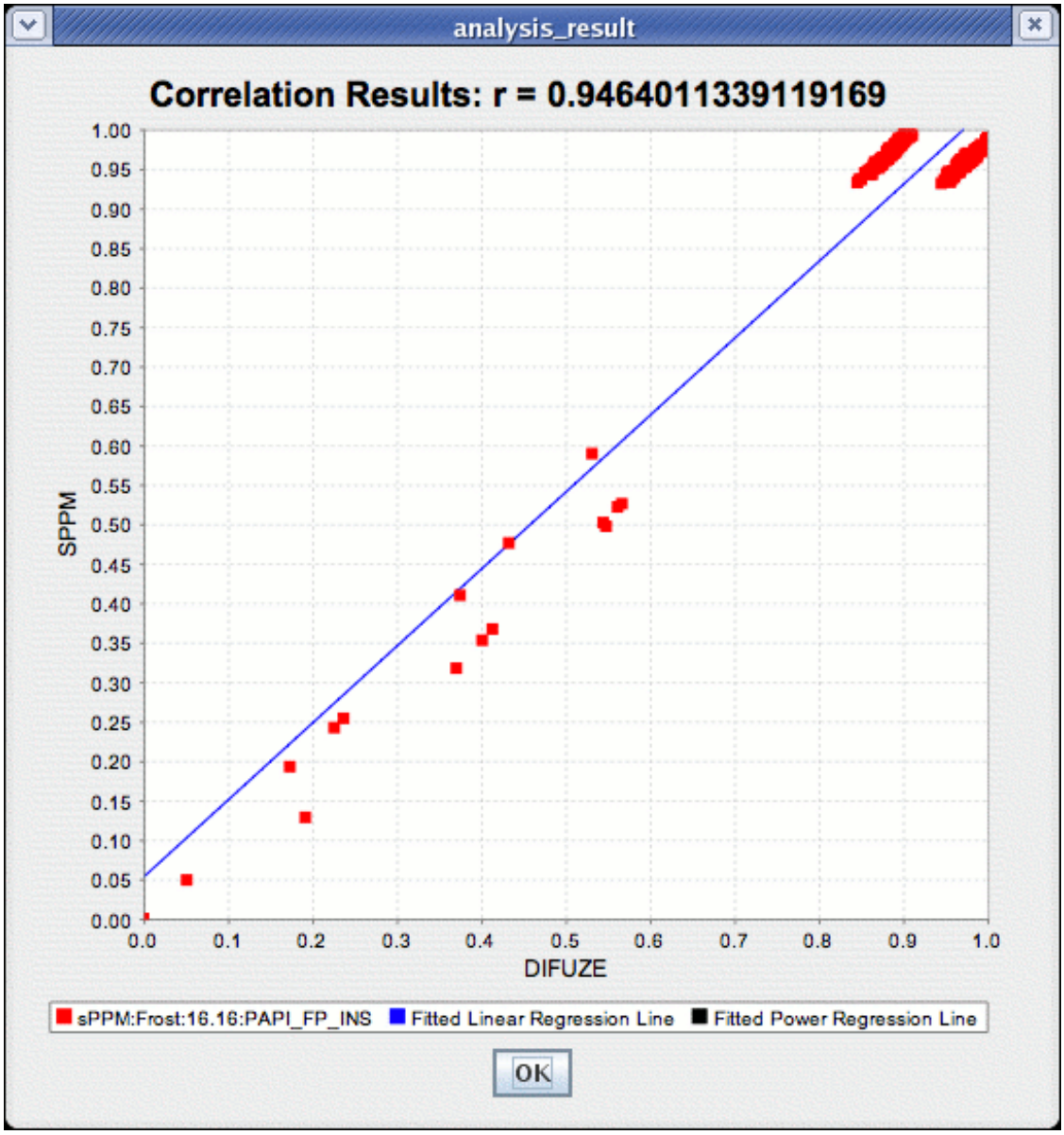
After selecting the metric of interest, select the "Do Correlation Analysis" item under the "Analysis" main menu bar item. A confirmation dialog will appear, and you can either confirm the correlation request or cancel it. After confirming the correlation, the analysis will begin. When the analysis results are available, you can view them in the "Correlation Results" tab.

Figure 21.4. Correlation Results



There are a number of images in the "Correlation Results" window. Each thumbnail represents a pair-wise correlation plot of two events. Clicking on a thumbnail image in the main window will bring up the images, as shown below:

Figure 21.5. Correlation Example


analysis_result
Correlation Results: r = 0.9464011339119169
SPPM
DIFUZE
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
sPPM:Frost:16.16:PAPI_FP_INS
Fitted Linear Regression Line
Fitted Power Regression Line
OK

Chapter 22. Charts

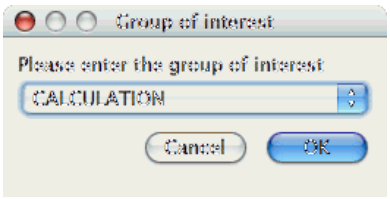
22.1. Setting Parameters

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:

22.1.1. 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.

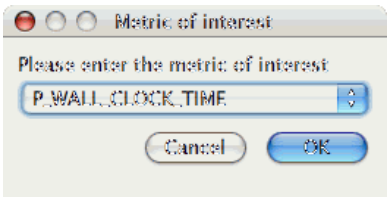
Figure 22.1. Setting Group of Interest



22.1.2. 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.

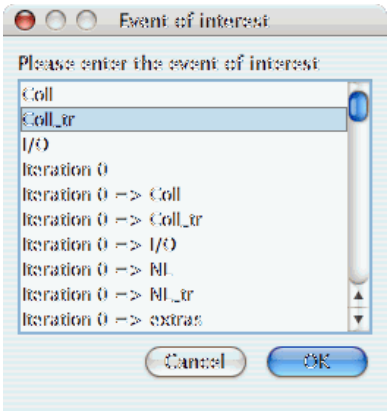
Figure 22.2. Setting Metric of Interest



22.1.3. Event of Interest

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

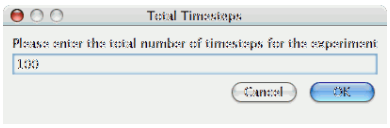
Figure 22.3. Setting Event of Interest



22.1.4. 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 22.4. Setting Timesteps

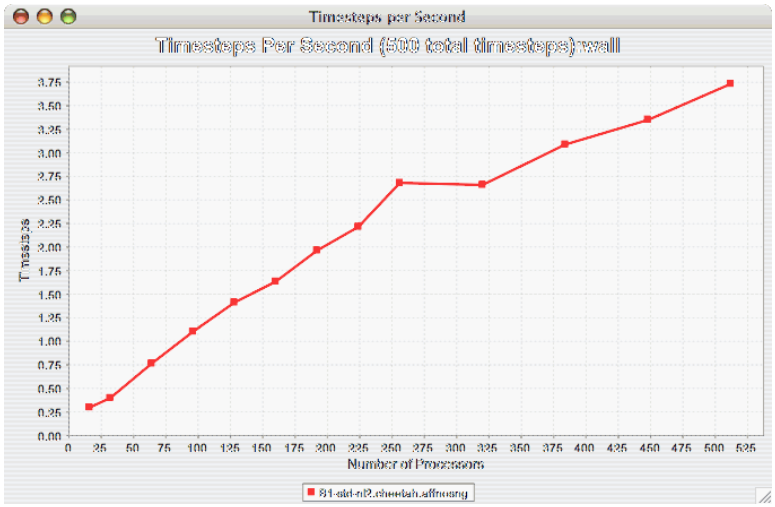


22.2. Standard Chart Types

22.2.1. 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 22.1.4, “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 22.1.2, “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.

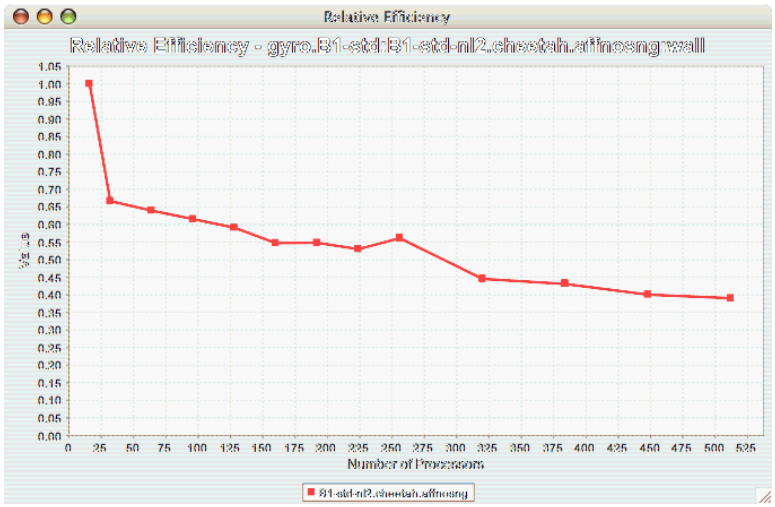
Figure 22.5. Timesteps per Second



22.2.2. 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 22.1.2, “Metric of Interest”). To request this chart, select one experiment or view, and select this chart item under the "Charts" main menu item.

Figure 22.6. Relative Efficiency

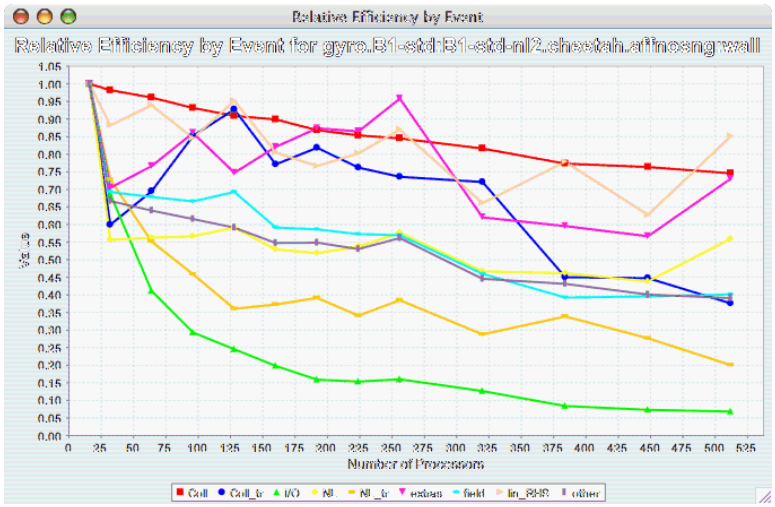


22.2.3. Relative Efficiency by Event

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 22.1.2, “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.

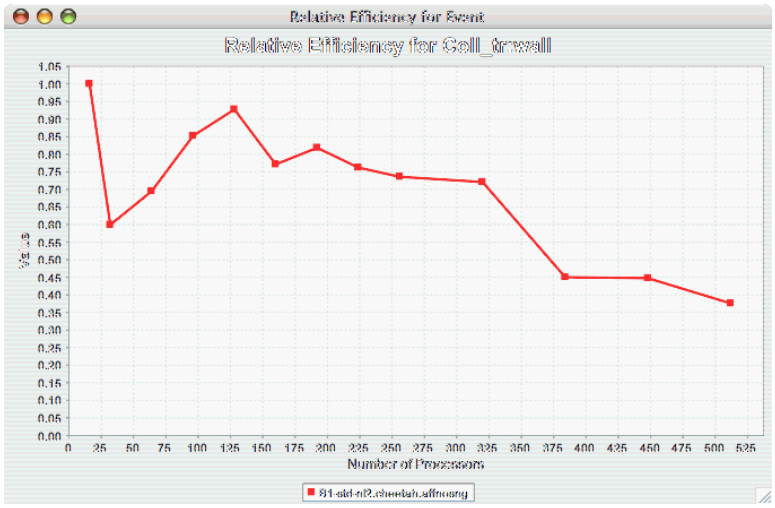
Figure 22.7. Relative Efficiency by Event



22.2.4. 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 22.1.3, “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 22.1.2, “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.

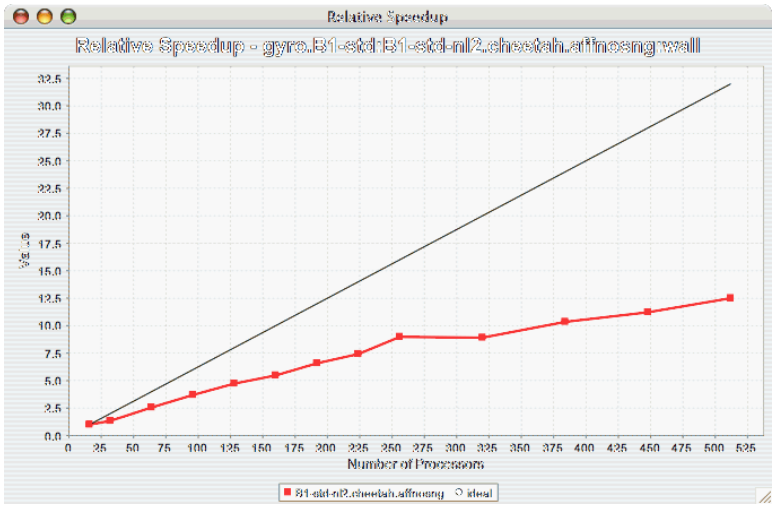
Figure 22.8. Relative Efficiency one Event



22.2.5. 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 22.1.2, “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.

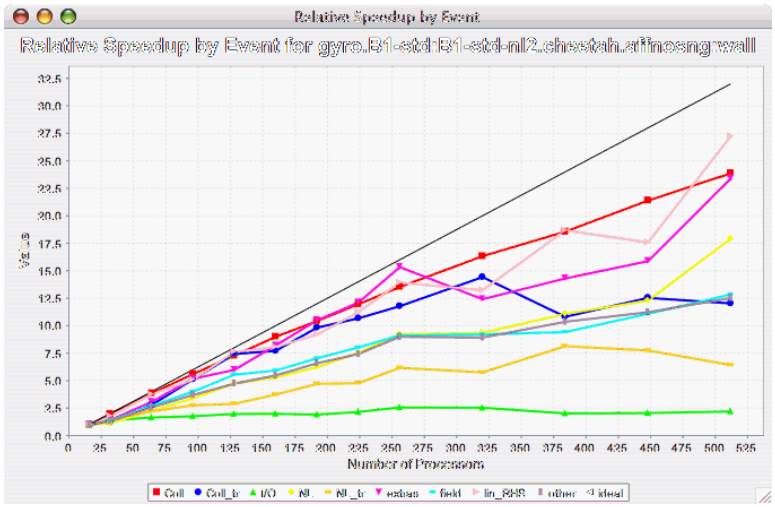
Figure 22.9. Relative Speedup



22.2.6. 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 22.1.2, “Metric of Interest”). To request this chart, select one experiment or view, and select this chart item under the "Charts" main menu item.

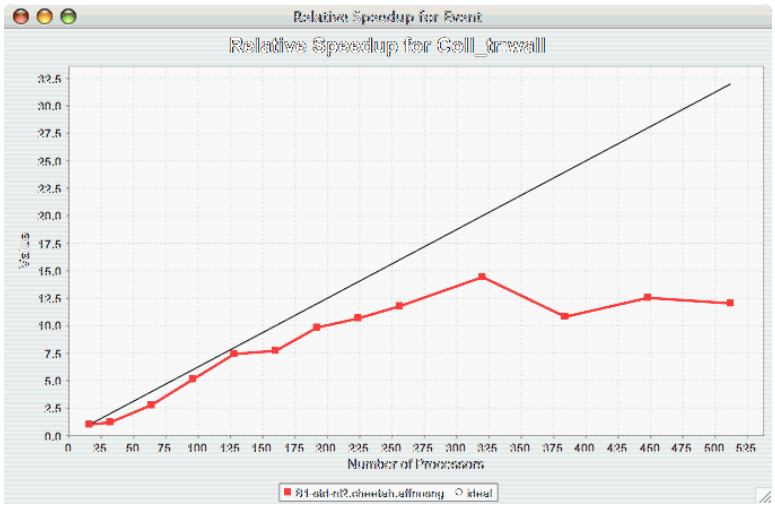
Figure 22.10. Relative Speedup by Event



22.2.7. 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 22.1.3, “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 22.1.2, “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 22.11. Relative Speedup one Event

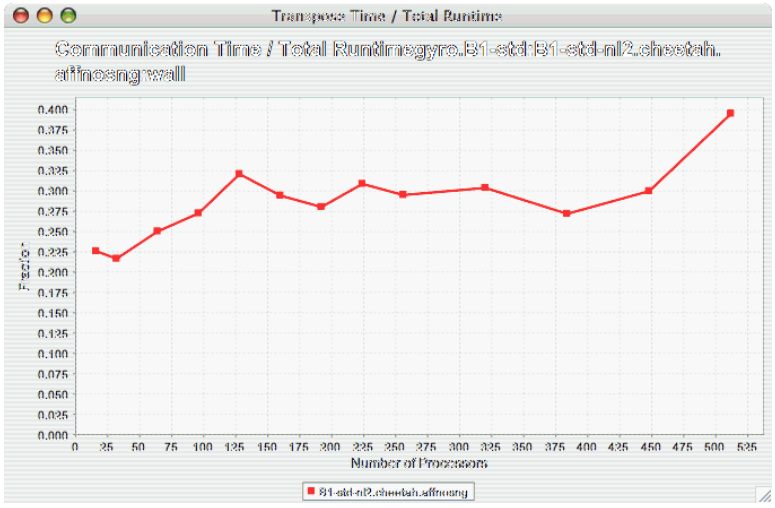


22.2.8. Group % of Total Runtime

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 22.1.1, “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 22.1.2, “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.

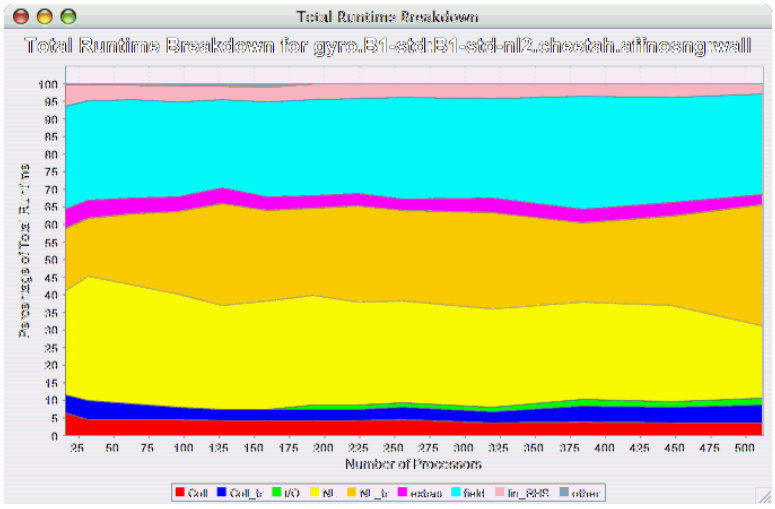
Figure 22.12. Group % of Total Runtime



22.2.9. 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 22.1.2, “Metric of Interest”). To request this chart, select one experiment or view, and select this chart item under the "Charts" main menu item.

Figure 22.13. Runtime Breakdown



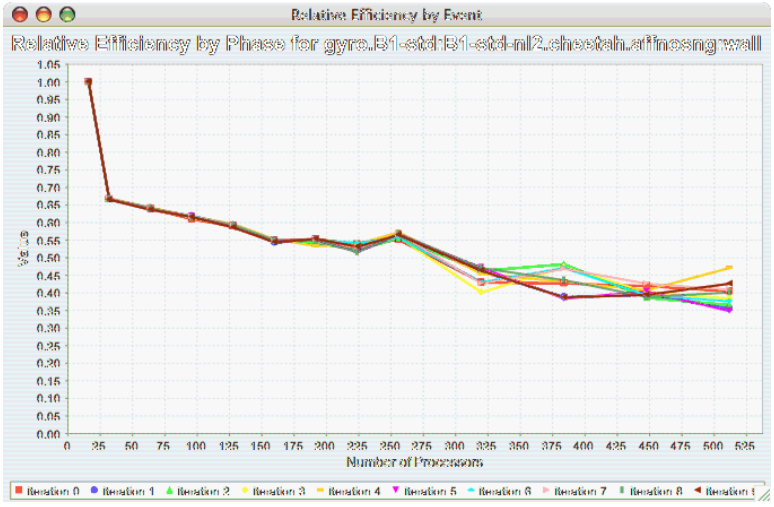
22.3. Phase Chart Types

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.

22.3.1. 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 22.1.2, “Metric of Interest”). To request this chart, select one experiment or view, and select this chart item under the "Charts" main menu item.

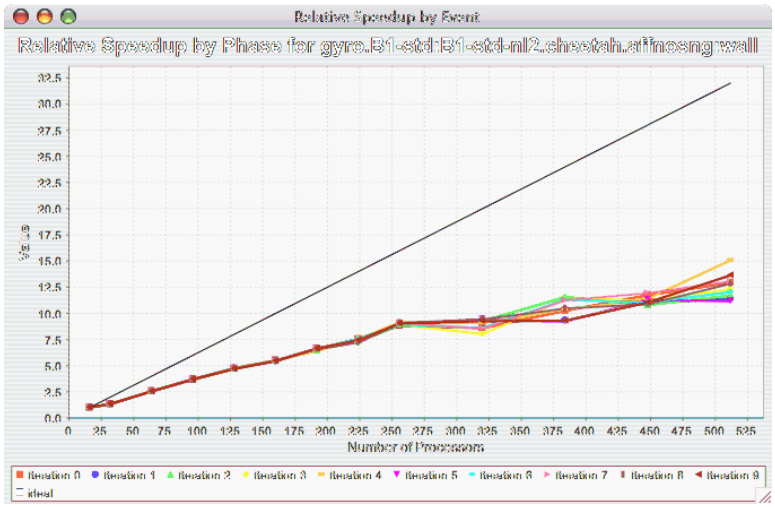
Figure 22.14. Relative Efficiency per Phase



22.3.2. 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 22.1.2, “Metric of Interest”). To request this chart, select one experiment or view, and select this chart item under the "Charts" main menu item.

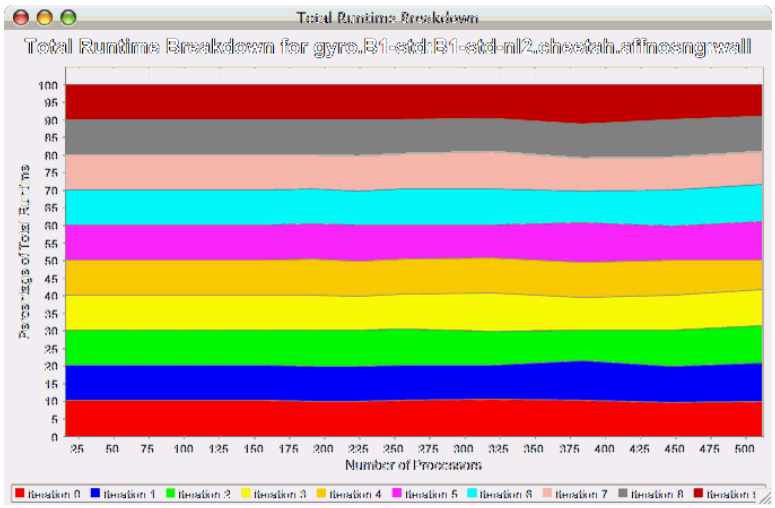
Figure 22.15. Relative Speedup per Phase



22.3.3. 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 22.1.2, “Metric of Interest”). To request this chart, select one experiment or view, and select this chart item under the "Charts" main menu item.

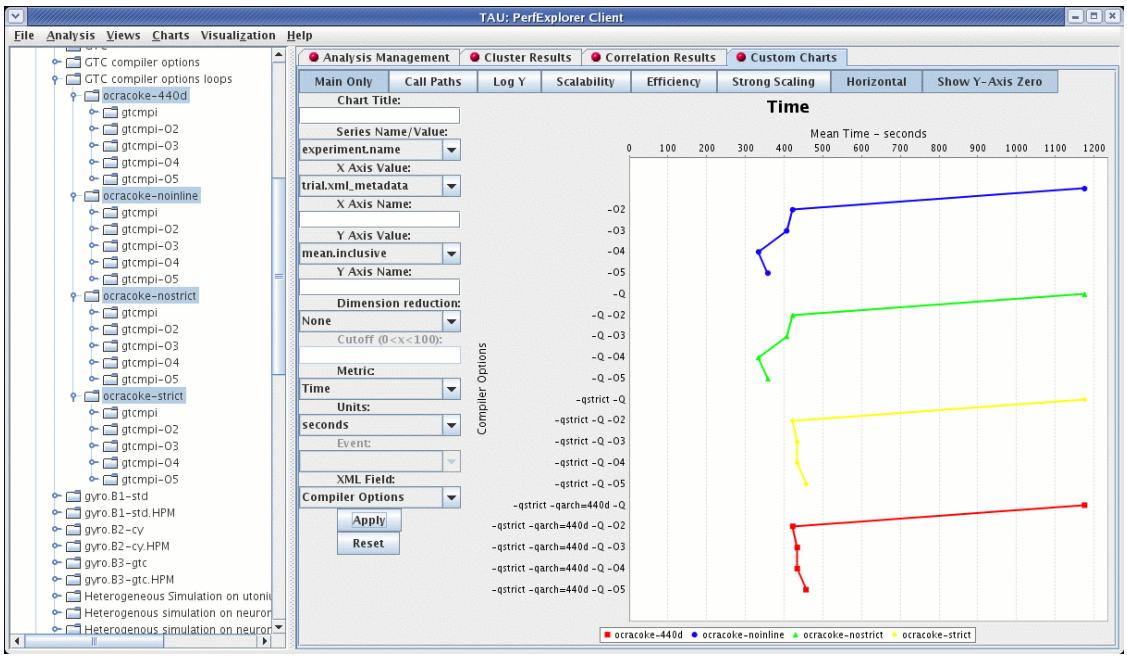
Figure 22.16. Phase Fraction of Total Runtime



Chapter 23. Custom Charts

In addition to the default charts available in the charts menu, there are is a custom chart interface. To access the interface, select the "Custom Charts" tab on in the results pane of the main window, as shown:

Figure 23.1. The Custom Charts Interface



There are a number of controls for the cusotom charts. They are:

- Main Only - When selected, only the main event (the event with the highest inclusive value) will be selected. When deselected, the "Events" control (see below) is activated, and one or all events can be selected.
- Call Paths - When selected, callpath events will be available in the "Events" control (see below).
- Log Y - When selected, the Y axis will be the log of the true value.
- Scalability - When selected, the chart will be interpreted as a speedup chart. The trial with the fewest number of threads of execution will be considered the baseline trial.
- Efficiency - When selected, the chart will be interpreted as a relative efficiency chart. The trial with the fewest number of threads of execution will be considered the baseline trial.
- Strong Scaling - When deselected, the speedup or efficiency chart will be interpreted as a strong scaling study (the workload is the same for all trials). When selected, the button will change to "Weak Scaling", and the chart will be interpreted as a weak scaling study (the workload is proportional to the total number of threads in each trial).
- Horizontal - when selected, the chart X and Y axes will be swapped.
- Show Y-Axis Zero - when selected, the chart will include the value 0. When deselected, the chart

will only show the relevant values for all data points.

- Chart Title - value to use for the chart title
- Series Name/Value - the field to be used to group the data points as a series.
- X Axis Value - the field to use as the X axis value.
- X Axis Name - the name to put in the chart for the value along the X axis.
- Y Axis Value - the field to use as the Y axis value
- Y Axis Name - the name to put in the chart for the value along the X axis.
- Dimension Reduction - whether or not to use dimension reduction. This is only applicable when "Main Only" is disabled.
- Cutoff - when the "Dimension Reduction" is enabled, the cutoff value for selecting "All Events".
- Metric - The metric of interest for the Y axis.
- Units - The unit to be selected for the Y axis.
- Event - The event of interest, or "All Events".
- XML Field - When the X or Y axis is selected to be an XML field, this is the field of interest.
- Apply - build the chart.
- Reset - restore the options back to the default values.

When the chart is generated, it can be saved as a vector image by selecting "File -> Save As Vector Image". The chart can also be saved as a PNG by right clicking on the chart, and selecting "Save As...".

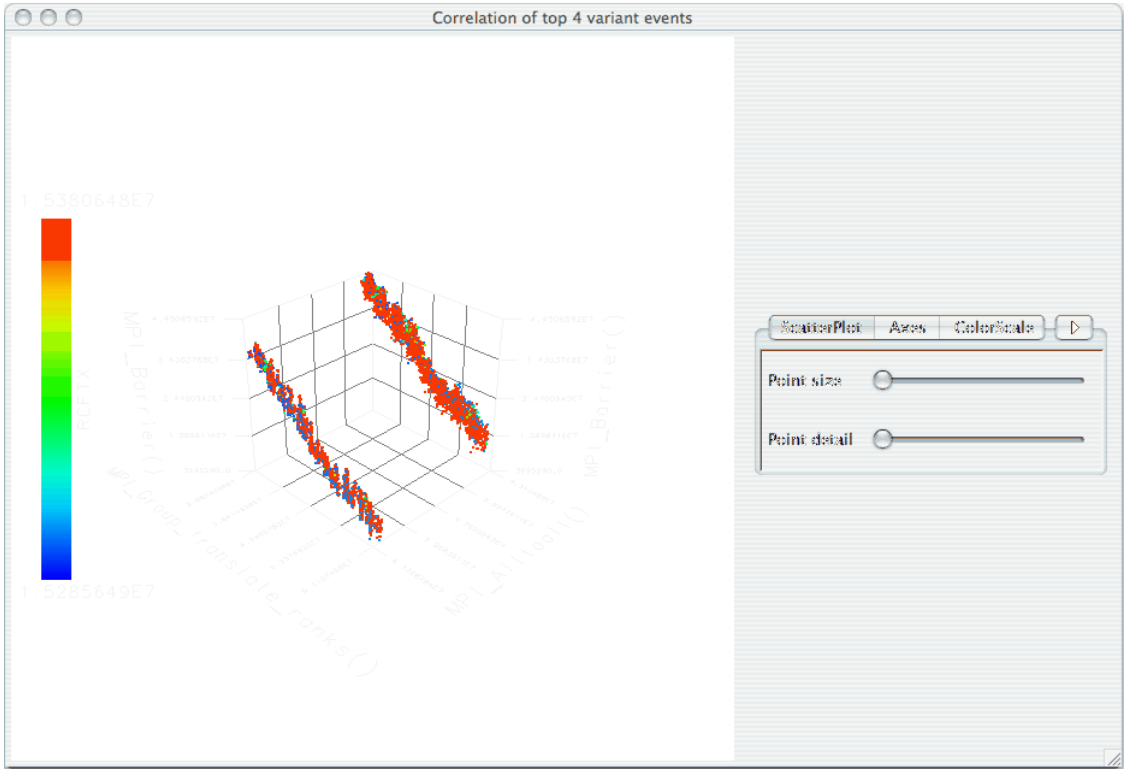
Chapter 24. Visualization

Under the "Visualization" main menu item, there are five types of raw data visualization. The five items are "3D Visualization", "Data Summary", "Create Boxchart", "Create Histogram" and "Create Normal Probability Chart". For the Boxchart, Histogram and Normal Probability Charts, you can either select one metric in the trial (which selects all events by default), or expand the metric and select events of interest.

24.1. 3D Visualization

When the "3D Visualization" is requested, PerfExplorer examines the data to try to determine the most interesting events in the trial. That is, for the selected metric in the selected trial, the database will calculate the weighted relative variance for each event across all threads of execution, in order to find the top four "significant" events. These events are selected by calculating: stddev(exclusive) / (max(exclusive) - min(exclusive)) * exclusive_percentage. After selecting the top four events, they are graphed in an OpenGL window.

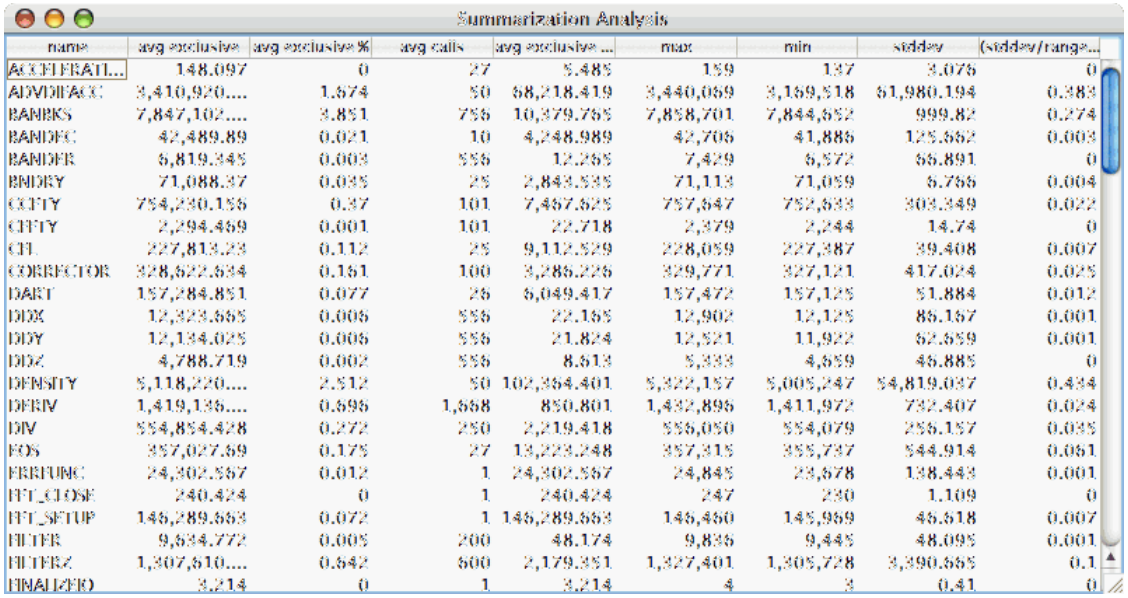
Figure 24.1. 3D Visualization of multivariate data



24.2. Data Summary

In order to see a summary of the performance data in the database, select the "Show Data Summary" item under the "Visualization" main menu item.

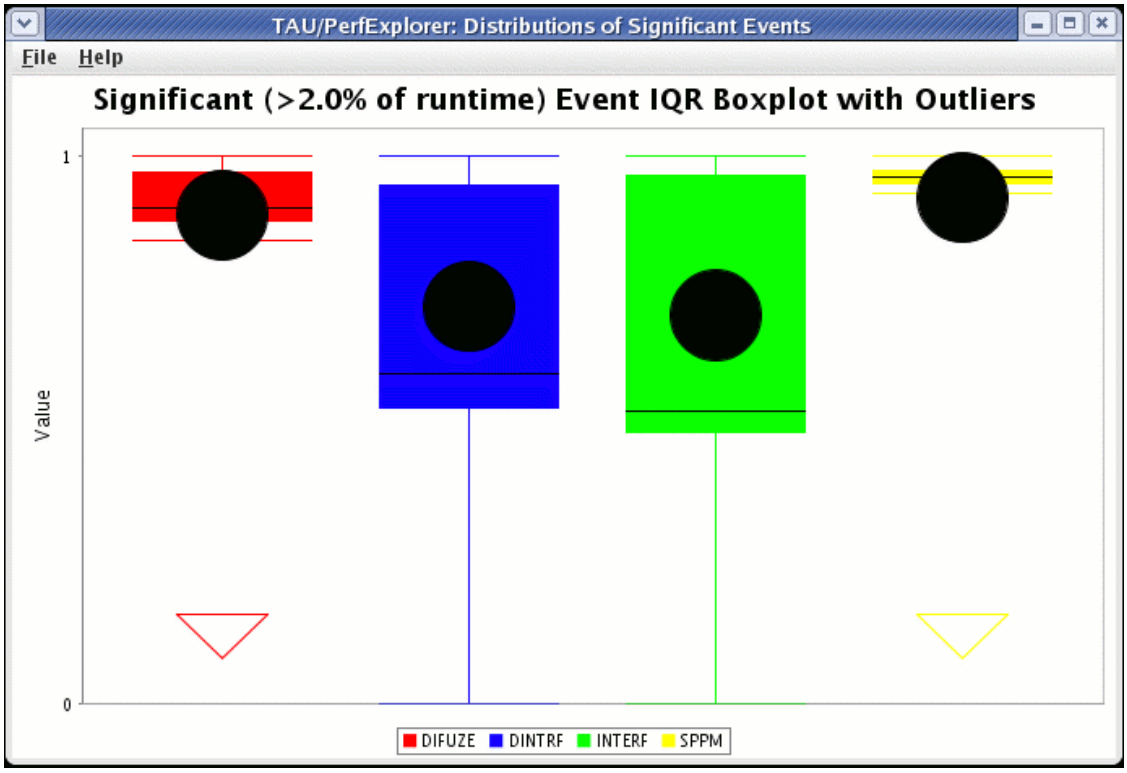
Figure 24.2. Data Summary Window



24.3. Creating a Boxchart

In order to see a boxchart summary of the performance data in the database, select the "Create Boxchart" item under the "Visualization" main menu item.

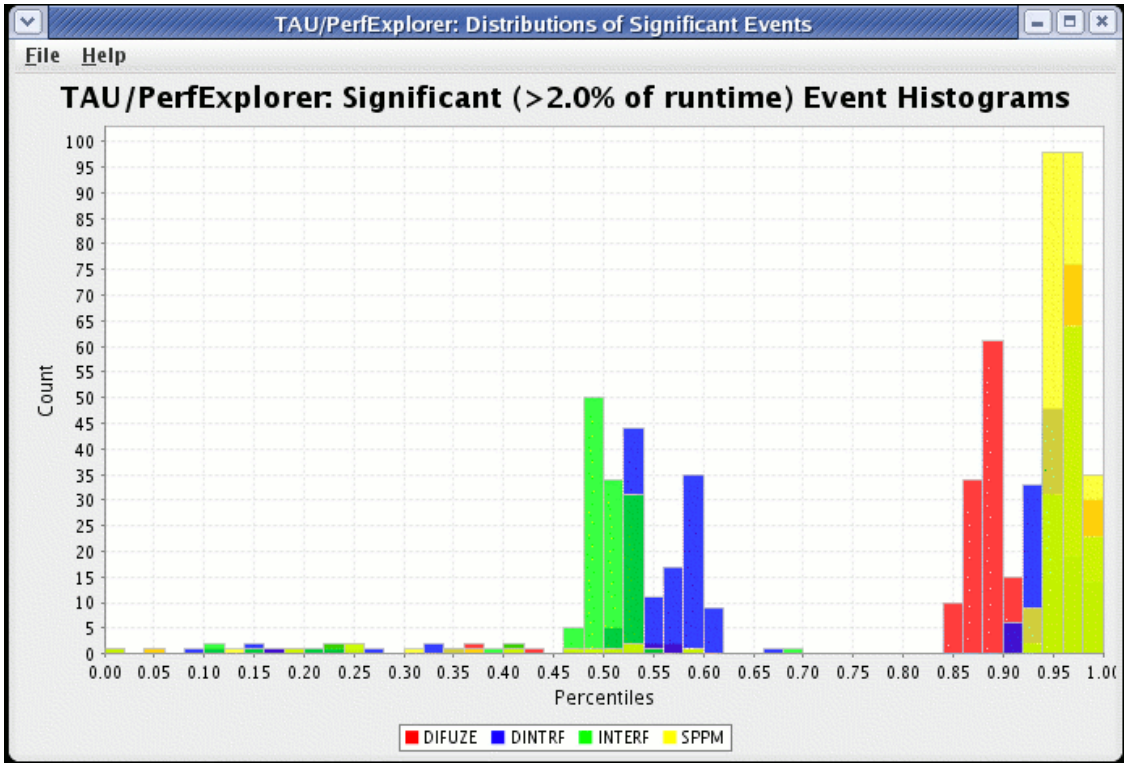
Figure 24.3. Boxchart



24.4. Creating a Histogram

In order to see a histogram summary of the performance data in the database, select the "Create Histogram" item under the "Visualization" main menu item.

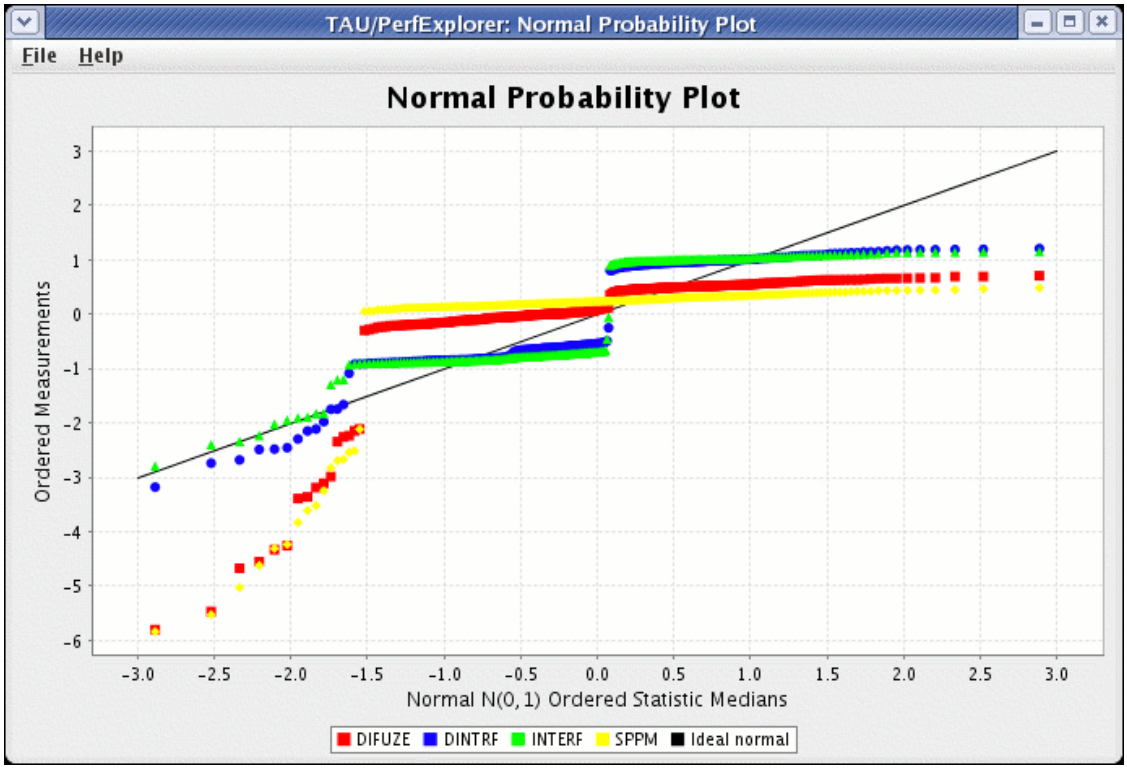
Figure 24.4. Histogram



24.5. Creating a Normal Probability Chart

In order to see a normal probability summary of the performance data in the database, select the "Create NormalProbability" item under the "Visualization" main menu item.

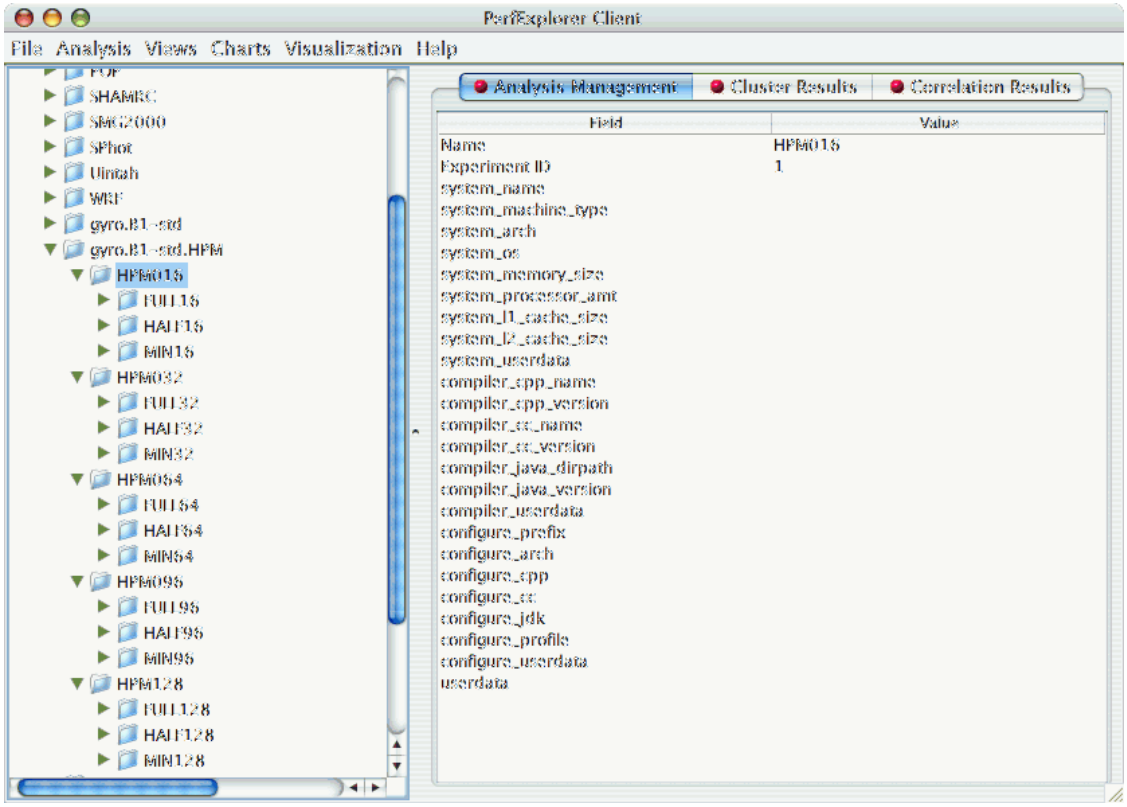
Figure 24.5. Normal Probability


TAU/PerfExplorer: Normal Probability Plot
File Help
Normal Probability Plot
Ordered Measurements
3
2
1
0
-1
-2
-3
-4
-5
-6
-3.0
-2.5
-2.0
-1.5
-1.0
-0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
Normal N(0, 1) Ordered Statistic Medians
DIFUZE
DINTRF
INTERF
SPPM
Ideal normal

Chapter 25. Views

Often times, data is loaded into the database with multiple parametric cross-sections. For example, the charts available in PerfExplorer are primarily designed for scalability analysis, however data might be loaded as a parametric study. For example, in the following example, the data has been loaded with three problem sizes, MIN, HALF and FULL.

Figure 25.1. Potential scalability data organized as a parametric study

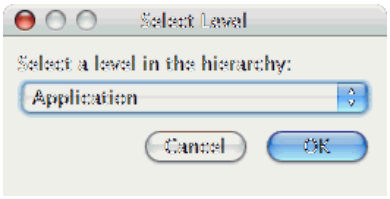


In order to examine this data in a scalability study, it is necessary to reorganize the data. However, it is not necessary to re-load the data. Using views in PerfExplorer, you can re-organize the data based on values in the database.

25.1. Creating Views

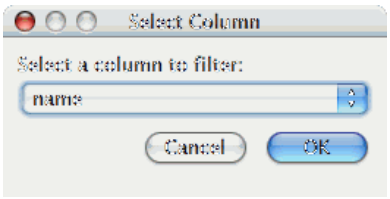
To create a view, select the "Create New View" item under the "Views" main menu item. The first step is to select the table which will form the basis of the view. The three possible values are Application, Experiment and Trial:

Figure 25.2. Selecting a table



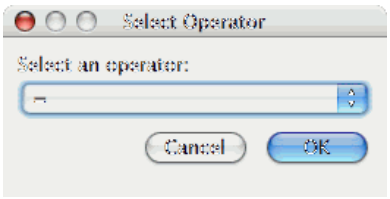
After selecting the table, you need to select the column on which to filter:

Figure 25.3. Selecting a column



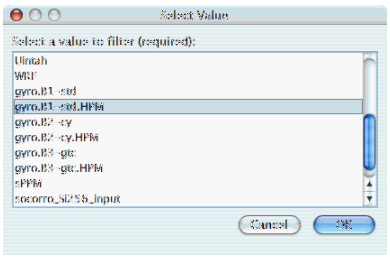
After selecting the column, you need to select the operator for comparing to that column:

Figure 25.4. Selecting an operator



After selecting the operator, you need to select the value for comparing to the column:

Figure 25.5. Selecting a value

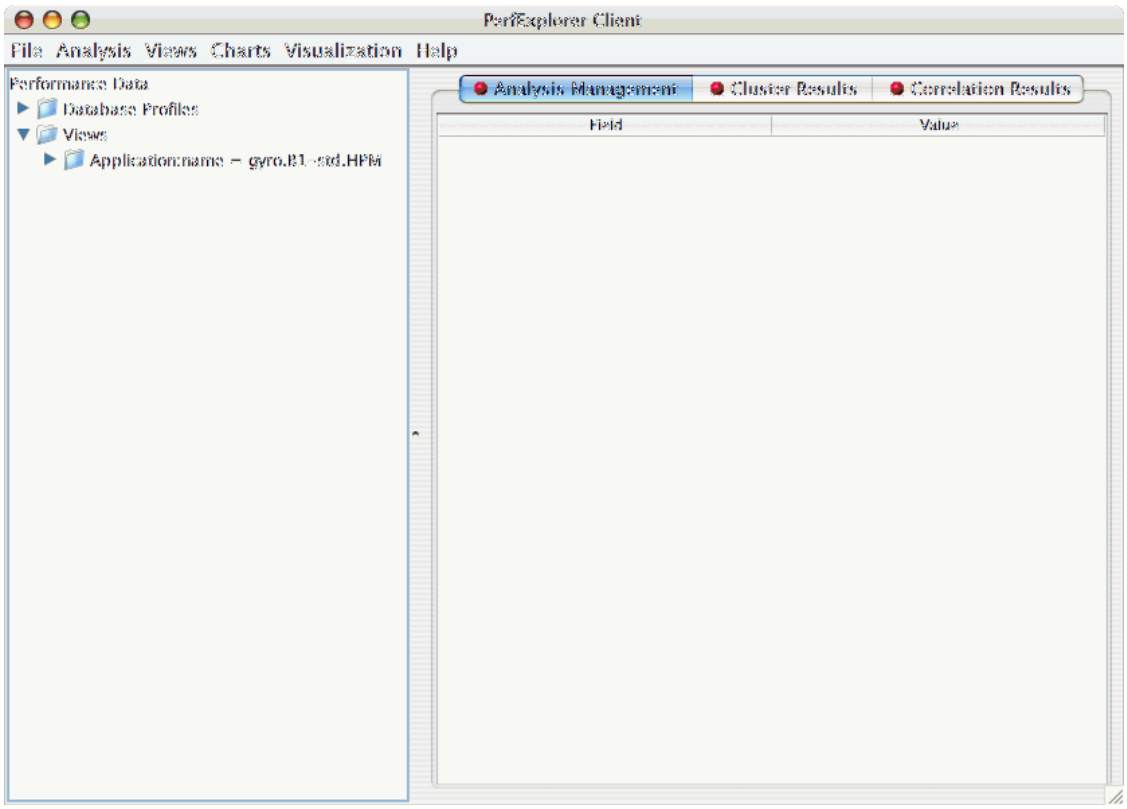


After selecting the value, you need to select a name for the view:

Figure 25.6. Entering a name for the view

After creating the view, you will need to exit PerfExplorer and re-start it to see the view. This is a known problem with the application, and will be fixed in a future release.

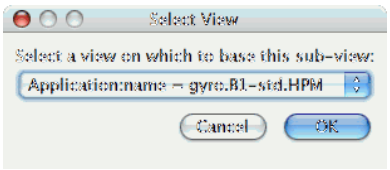
Figure 25.7. The completed view



25.2. Creating Subviews

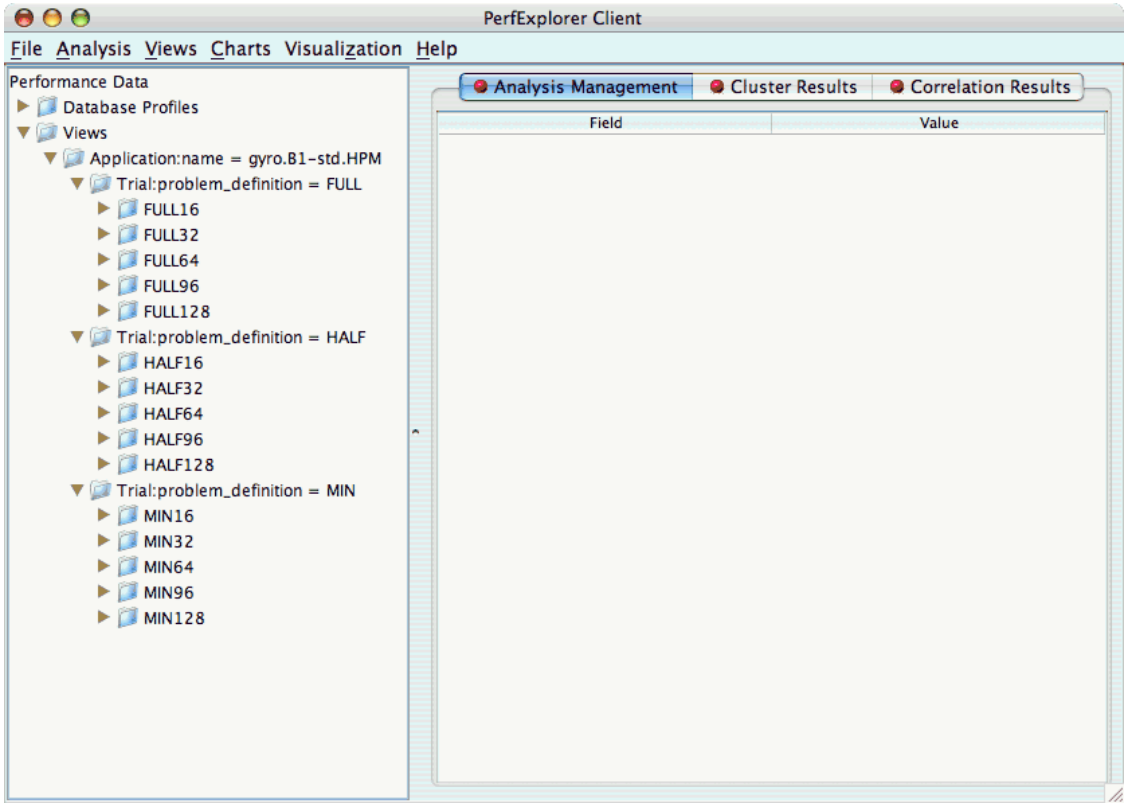
In order to create sub-views, you first need to select the "Create New Sub-View" item from the "Views" main menu item. The first dialog box will prompt you to select the view (or sub-view) to base the new sub-view on:

Figure 25.8. Selecting the base view



After selecting the base view or sub-view, the options for creating the new sub-view are the same as creating a new view. After creating the sub-view, you will need to exit PerfExplorer and re-start it to see the sub-view. This is a known problem with the application, and will be fixed in a future release.

Figure 25.9. Completed sub-views



Chapter 26. Running PerfExplorer Scripts

As of version 2.0, PerfExplorer has officially supported a scripting interface. The scripting interface is useful for adding automation to PerfExplorer. For example, a user can load a trial, perform data reduction, extract out key phases, derive metrics, and plot the result.

26.1. Analysis Components

There are many operations available, including:

- BasicStatisticsOperation
- CopyOperation
- CorrelateEventsWithMetadata
- CorrelationOperation
- DeriveMetricOperation
- DifferenceMetadataOperation
- DifferenceOperation
- DrawBoxChartGraph
- DrawGraph
- DrawMMMGraph
- ExtractCallpathEventOperation
- ExtractEventOperation
- ExtractMetricOperation
- ExtractNonCallpathEventOperation
- ExtractPhasesOperation
- ExtractRankOperation
- KMeansOperation
- LinearRegressionOperation
- LogarithmicOperation
- MergeTrialsOperation
- MetadataClusterOperation
- PCAOperation

- RatioOperation
- ScalabilityOperation
- TopXEvents
- TopXPercentEvents

26.2. Scripting Interface

The scripting interface is in Python, and scripts can be used to build analysis workflows. The Python scripts control the Java classes in the application through the Jython interpreter (http://www.jython.org/). There are two types of components which are useful in building analysis scripts. The first type is the PerformanceResult interface, and the second is the PerformanceAnalysisComponent interface. For documentation on how to use the Java classes, see the javadoc in the perfexplorer source distribution, and the example scripts below. To build the perfexplorer javadoc, type

```
%>./make javadoc
```

in the perfexplorer source directory.

26.3. Example Script

```
from glue import PerformanceResult
from glue import PerformanceAnalysisOperation
from glue import ExtractEventOperation
from glue import Utilities
from glue import BasicStatisticsOperation
from glue import DeriveMetricOperation
from glue import MergeTrialsOperation
from glue import TrialResult
from glue import AbstractResult
from glue import DrawMMMGraph
from edu.uoregon.tau.perfdmf import Trial
from java.util import HashSet
from java.util import ArrayList

True = 1
False = 0

def glue():
        print "doing phase test for gtc on jaguar"
        # load the trial
        Utilities.setSession("perfdmf.demo")
        trial1 = Utilities.getTrial("gtc_bench", "Jaguar Compiler Options", "fasts
        result1 = TrialResult(trial1)

        print "got the data"

        # get the iteration inclusive totals

        events = ArrayList()
        for event in result1.getEvents():
                #if event.find("Iteration") >= 0 and result1.getEventGroupName(eve
                if event.find("Iteration") >= 0 and event.find("=>") < 0:
                        events.add(event)

        extractor = ExtractEventOperation(result1, events)
```

```
extracted = extractor.processData().get(0)

print "extracted phases"

# derive metrics

derivor = DeriveMetricOperation(extracted, "PAPI_L1_TCA", "PAPI_L1_TCM", D
derived = derivor.processData().get(0)
merger = MergeTrialsOperation(extracted)
merger.addInput(derived)
extracted = merger.processData().get(0)
derivor = DeriveMetricOperation(extracted, "PAPI_L1_TCA-PAPI_L1_TCM", "PAP
derived = derivor.processData().get(0)
merger = MergeTrialsOperation(extracted)
merger.addInput(derived)
extracted = merger.processData().get(0)
derivor = DeriveMetricOperation(extracted, "PAPI_L1_TCM", "PAPI_L2_TCM", D
derived = derivor.processData().get(0)
merger = MergeTrialsOperation(extracted)
merger.addInput(derived)
extracted = merger.processData().get(0)
derivor = DeriveMetricOperation(extracted, "PAPI_L1_TCM-PAPI_L2_TCM", "PAP
derived = derivor.processData().get(0)
merger = MergeTrialsOperation(extracted)
merger.addInput(derived)
extracted = merger.processData().get(0)
derivor = DeriveMetricOperation(extracted, "PAPI_FP_INS", "P_WALL_CLOCK_TI
derived = derivor.processData().get(0)
merger = MergeTrialsOperation(extracted)
merger.addInput(derived)
extracted = merger.processData().get(0)
derivor = DeriveMetricOperation(extracted, "PAPI_FP_INS", "PAPI_TOT_INS",
derived = derivor.processData().get(0)
merger = MergeTrialsOperation(extracted)
merger.addInput(derived)
extracted = merger.processData().get(0)

print "derived metrics..."

# get the Statistics
dostats = BasicStatisticsOperation(extracted, False)
stats = dostats.processData()

print "got stats..."

return

for metric in stats.get(0).getMetrics():
        grapher = DrawMMMGraph(stats)
        metrics = HashSet()
        metrics.add(metric)
        grapher.set_metrics(metrics)
        grapher.setTitle("GTC Phase Breakdown: " + metric)
        grapher.setSeriesType(DrawMMMGraph.TRIALNAME);
        grapher.setCategoryType(DrawMMMGraph.EVENTNAME)
        grapher.setValueType(AbstractResult.INCLUSIVE)
        grapher.setXAxisLabel("Iteration")
        grapher.setYAxisLabel("Inclusive " + metric);
        # grapher.setLogYAxis(True)
        grapher.processData()

# graph the significant events in the iteration

subsetevents = ArrayList()
```

```
        subsetevents.add("CHARGEI")
        subsetevents.add("PUSHI")
        subsetevents.add("SHIFTI")

        print "got data..."

        for subsetevent in subsetevents:
                events = ArrayList()
                for event in result1.getEvents():
                        if event.find("Iteration") >= 0 and event.rfind(subseteven
                                events.add(event)

                extractor = ExtractEventOperation(result1, events)
                extracted = extractor.processData().get(0)

                print "extracted phases..."

                # get the Statistics
                dostats = BasicStatisticsOperation(extracted, False)
                stats = dostats.processData()

                print "got stats..."

                for metric in stats.get(0).getMetrics():
                        grapher = DrawMMMGraph(stats)
                        metrics = HashSet()
                        metrics.add(metric)
                        grapher.set_metrics(metrics)
                        grapher.setTitle(subsetevent + ", " + metric)
                        grapher.setSeriesType(DrawMMMGraph.TRIALNAME);
                        grapher.setCategoryType(DrawMMMGraph.EVENTNAME)
                        grapher.setValueType(AbstractResult.INCLUSIVE)
                        # grapher.setLogYAxis(True)
                        grapher.processData()
        return

print "--------------- JPython test script start ------------"
glue()
print "----------------- JPython test script end -------------"
```

Chapter 27. Derived Metrics

Sometimes metrics in a profile need to be combined to create a derived metric. PerfExplorer allows the user to create these using the derived metric expression tab.

27.1. CreatingExpressions

The text box at the top of the tab allows the user to enter an expression. Double clicking on a metric in the "Performance Data" tree will copy that metrics name into the box. If a metric contains any operands, the whole metric must be surrounded by quotes. If the you would like of the metric to be renamed, then you should start the expression with the new name and and equals sign.

If this is the only metric you wish to derive, then select the trial, expression or application where the metric should be derived and then click apply. If you wish to derive many metrics, then click Add to List and create more expressions.

27.2. Selecting Expressions

If you have added multiple expressions, you can select one or many of them to apply. They will be derived from top to bottom. After you have select some, you can select the trial, experiment or application to apply the expression to and then click apply.

27.3. Expression Files

You can also derive metrics using an expression file. An expression file has a single expression per line. To parse the file, select the trial, experiment or application to apply the expressions to; then select File > Parse Expression File and chose the file.

Part IV. TAUdb

Table of Contents

Chapter 28. Introduction

TAUdb (TAU Database), formerly known as 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.

28.1. Prerequisites

1. A supported Database Management System (DBMS). TAUdb currently supports PostgreSQL, MySQL, Oracle, H2, and Derby. For use with the C API, only PostgreSQL is supported (SQLite support is currently being evaluated). Because they are Java only, H2 and Derby can NO be accessed with the C API.
2. Java 1.5.
3. If the C API is desired, a working C compiler is required, along with the following libraries: libpq (PostgreSQL libraries), libxml2, libz, libuuid. These libraries are all commonly installed by default on *NIX systems.

28.2. Installation

The TAUdb 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. (Optionally) Create a database. This step will depend on the user's chosen DBMS.
   - **H2:** Because it is an embedded, file-based DBMS, H2 does **not** require creating the database before configuring TAUdb. TAUdb takes advantage of the "auto-server" capabilities in H2, so multiple clients can connect to the same database at the same time. Users should use the H2 DBMS if they expect to maintain a small to moderate local repository of performance data, and want the convenience of connecting to the database from multiple clients.
   - **Derby:** Because it is an embedded, file-based DBMS, Derby does **not** require creating the database before configuring TAUdb. Be advised that the Derby DBMS does **not** allow multiple clients to connect to the same database. For that reason, we suggest users use the H2 DBMS if a file-based database is desired. Derby support is maintained for backwards compatability.
   - **PostgreSQL:**

     ```
     $ createdb -O taudb taudb
     ```

     Or, from **psql**

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

   - **MySQL:** From the MySQL prompt

     ```
     mysql> create database taudb;
     ```

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

```
create tablespace taudb
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 taudb;
alter user amorris default tablespace taudb;
```

TAUdb is set up to use the Oracle Thin Java driver. You will have to obtain this jar file for your DBMS. In our case, it was ojdbc14.jar.

2. Configure a TAUdb connection. To configure TAUdb, run the **taudb_configure** program 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 already found in the database (as will be the case on initial configuration), the schema will be uploaded. Be sure to specify the correct version of the schema for your DBMS.

   An example session for configuring a database is below. The user is creating an H2 database, with default settings including no username and no password (recommended for file-based databases when security is not an issue).

```
$ taudb_configure
Configuration file NOT found...
a new configuration file will be created.

Welcome to the configuration program for PerfDMF.
This program will prompt you for some information necessary to
ensure the desired behavior for the PerfDMF tools.


You will now be prompted for new values, if desired.  The current
or default values for each prompt are shown in parenthesis.
To accept the current/default value, just press Enter/Return.

Please enter the name of this configuration.
():documentation_example
Please enter the database vendor (oracle, postgresql, mysql, db2,
derby or h2).
(h2):
Please enter the JDBC jar file.
(/Users/khuck/src/tau2/apple/lib/h2.jar):
Please enter the JDBC Driver name.
(org.h2.Driver):
Please enter the path to the database directory.
(/Users/khuck/.ParaProf/documentation_example):
Please enter the database username.
():
Store the database password in CLEAR TEXT in your configuration
file? (y/n):y
Please enter the database password:
Please enter the PerfDMF schema file.
```

```
(/Users/khuck/src/tau2/etc/taudb.sql):

Writing configuration file:
/Users/khuck/.ParaProf/perfdmf.cfg.documentation_example

Now testing your database connection.

Database created, command:
jdbc:h2:/Users/khuck/.ParaProf/documentation_example/perfdmf;AUTO_SERVER=TRUE;c

Uploading Schema: /Users/khuck/src/tau2/etc/taudb.sql
Found /Users/khuck/src/tau2/etc/taudb.sql  ... Loading
Successfully uploaded schema

Database connection successful.
Configuration complete.
```

Chapter 29. Using TAUdb

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

29.1. perfdmf_createapp (deprecated - only supported for older PerfDMF databases)

This utility creates applications with a given name

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

29.2. perfdmf_createexp (deprecated - only supported for older PerfDMF databases)

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

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

29.3. taudb_loadtrial

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

```
$ taudb_loadtrial -h
Usage: perfdmf_loadtrial -a <appName> -x <expName> -n <name>
[options] <files>

Required Arguments:

  -n, --name <text>              Specify the name of the trial
  -a, --applicationname <string> Specify associated application name
                                 for this trial
  -x, --experimentname <string>  Specify associated experiment name
                                 for this trial
              ...or...

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

Optional Arguments:

  -c, --config <name>       Specify the name of the configuration to use
  -g, --configFile <file>   Specify the configuration file to use
                            (overrides -c)
  -f, --filetype <filetype> Specify type of performance data, options
                            are: profiles (default), pprof, dynaprof,
                            mpip, gprof, psrun, hpm, packed, cube,
                            hpc, ompp, snap, perixml, gptl, paraver,
                            ipm, google
```

```
  -t, --trialid <number>    Specify trial ID
  -i, --fixnames            Use the fixnames option for gprof
  -z, --usenull             Include NULL values as 0 for mean
                            calculation
  -r, --reduce <percentage> Aggregate all timers less than percentage
                            as "other"
  -m, --metadata <filename> XML metadata for the trial

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

  perfdmf_loadtrial -a "NPB2.3" -x "parametric" -n "64" par64.ppk
    This will load packed profile par64.ppk into the experiment named
    "parametric" under the application named "NPB2.3" and give the
    trial the name "64".  The application and experiment will be
    created if not found.
```

TAUdb supports a large number of parallel profile formats:

- **TAU Profiles (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.
- **ParaProf Packed Format (ppk)** - Export format supported by PerfDMF/ParaProf. Typically .ppk.
- **TAU Merged Profiles (snap)** - Merged and snapshot profile format supported by TAU. Typically tauprofile.xml.
- **TAU pprof (pprof)** - Dump Output from TAU's **pprof -d**. Provided for backward compatibility only.
- **DynaProf (dynaprof)** - Output From DynaProf's wallclock and papi probes.
- **mpiP (mpip)** - Output from mpiP.
- **gprof (gprof)** - Output from gprof, see also the --fixnames option.
- **PerfSuite (psrun)** - Output from PerfSuite psrun files.
- **HPM Toolkit (hpm)** - Output from IBM's HPM Toolkit.
- **Cube (cube)** - Output from Kojak Expert tool for use with Cube.

- **Cube3 (cube3)** - Output from Kojak Expert tool for use with Cube3 and Cube4.
- **HPCToolkit (hpc)** - XML data from hpcquick. Typically, the user runs hpcrun, then hpcquick on the resulting binary file.
- **OpenMP Profiler (ompp)** - CSV format from the ompP OpenMP Profiler (http://www.ompp-tool.com). The user must use OMPP_OUTFORMAT=CVS.
- **PERI XML (perixml)** - Output from the PERI data exchange format.
- **General Purpose Timing Library (gptl)** - Output from the General Purpose Timing Library.
- **Paraver (paraver)** - 2D output from the Paraver trace analysis tool from BSC.
- **IPM (ipm)** - Integrated Performance Monitoring format, from NERSC.
- **Google (google)** - Google Profiles.

29.4. TAUdb Views

In order to provide flexible data management, the application / experiment / trial hierarchy was removed in the conversion from PerfDMF to TAUdb. In addition, trial metadata was promoted from an XML blob in PerfDMF to queryable tables. Users can now organize their data in arbitrary hierarchies using Views and SubViews. Creating and using Views is outlined in the ParaProf User Manual, in Chapter 2.

Chapter 30. Database Schema

The database schema in TAUdb is designed to flexibly and efficiently store multidimensional parallel performance data. There are 5 dimensions to the actual timer measurements, and 4 dimensions to the counter measurements

Timer dimensions

1. Process and thread of execution
2. Timer source code location (i.e. foo())
3. Metric of interest (i.e. FP_OPS, TIME)
4. Phase of execution (i.e. iteration number, timestamp)
5. Dynamic timer context (i.e. parameter values)

Counter dimensions

1. Process and thread of execution
2. Timer source code location (i.e. foo())
3. Phase of execution (i.e. iteration number, timestamp)
4. Dynamic timer context (i.e. parameter values)

30.1. SQL for TAUdb

Below is the SQL schema definition for TAUdb.

```
/*****************************/
/* CREATE THE STATIC TABLES */
/*****************************/

CREATE TABLE schema_version (
 version     INT NOT NULL,
 description VARCHAR NOT NULL
);
/* IF THE SCHEMA IS MODIFIED, INCREMENT THIS VALUE */
/* 0 = PERFDMF (ORIGINAL) */
/* 1 = TAUDB (APRIL, 2012) */
/*VALUES (1, 'TAUdb redesign from Spring, 2012');*/
INSERT INTO schema_version (version, description)
  VALUES (2, 'Changes after Nov. 9, 2012 release');

/* These are our supported parsers. */
CREATE TABLE data_source (
 id          INT UNIQUE NOT NULL,
 name        VARCHAR NOT NULL,
 description VARCHAR
);
```

```
INSERT INTO data_source (name,id,description)
  VALUES ('ppk',0,'TAU Packed profiles (TAU)');
INSERT INTO data_source (name,id,description)
  VALUES ('TAU profiles',1,'TAU profiles (TAU)');
INSERT INTO data_source (name,id,description)
  VALUES ('DynaProf',2,'PAPI DynaProf profiles (UTK)');
INSERT INTO data_source (name,id,description)
  VALUES ('mpiP',3,'mpiP: Lightweight, Scalable MPI Profiling (Vetter, Chambreau)'
INSERT INTO data_source (name,id,description)
  VALUES ('HPM',4,'HPM Toolkit profiles (IBM)');
INSERT INTO data_source (name,id,description)
  VALUES ('gprof',5,'gprof profiles (GNU)');
INSERT INTO data_source (name,id,description)
  VALUES ('psrun',6,'PerfSuite psrun profiles (NCSA)');
INSERT INTO data_source (name,id,description)
  VALUES ('pprof',7,'TAU pprof.dat output (TAU)');
INSERT INTO data_source (name,id,description)
  VALUES ('Cube',8,'Cube data (FZJ)');
INSERT INTO data_source (name,id,description)
  VALUES ('HPCToolkit',9,'HPC Toolkit profiles (Rice Univ.)');
INSERT INTO data_source (name,id,description)
  VALUES ('SNAP',10,'TAU Snapshot profiles (TAU)');
INSERT INTO data_source (name,id,description)
  VALUES ('OMPP',11,'OpenMP Profiler profiles (Fuerlinger)');
INSERT INTO data_source (name,id,description)
  VALUES ('PERIXML',12,'Data Exchange Format (PERI)');
INSERT INTO data_source (name,id,description)
  VALUES ('GPTL',13,'General Purpose Timing Library (ORNL)');
INSERT INTO data_source (name,id,description)
  VALUES ('Paraver',14,'Paraver profiles (BSC)');
INSERT INTO data_source (name,id,description)
  VALUES ('IPM',15,'Integrated Performance Monitoring (NERSC)');
INSERT INTO data_source (name,id,description)
  VALUES ('Google',16,'Google profiles (Google)');
INSERT INTO data_source (name,id,description)
  VALUES ('Cube3',17,'Cube 3D profiles (FZJ)');
INSERT INTO data_source (name,id,description)
  VALUES ('Gyro',100,'Self-timing profiles from Gyro application');
INSERT INTO data_source (name,id,description)
  VALUES ('GAMESS',101,'Self-timing profiles from GAMESS application');
INSERT INTO data_source (name,id,description)
  VALUES ('Other',999,'Other profiles');

/* threads make it convenient to identify timer values.
   Special values for thread_index:
   -1 mean (nulls ignored)
   -2 total
   -3 stddev (nulls ignored)
   -4 min
   -5 max
   -6 mean (nulls are 0 value)
   -7 stddev (nulls are 0 value)
*/

CREATE TABLE derived_thread_type (
 id INT NOT NULL,
 name VARCHAR NOT NULL,
 description VARCHAR NOT NULL
);

INSERT INTO derived_thread_type (id, name, description)
  VALUES (-1, 'MEAN', 'MEAN (nulls ignored)');
INSERT INTO derived_thread_type (id, name, description)
  VALUES (-2, 'TOTAL', 'TOTAL');
```

```
INSERT INTO derived_thread_type (id, name, description)
  VALUES (-3, 'STDDEV', 'STDDEV (nulls ignored)');
INSERT INTO derived_thread_type (id, name, description)
  VALUES (-4, 'MIN', 'MIN');
INSERT INTO derived_thread_type (id, name, description)
  VALUES (-5, 'MAX', 'MAX');
INSERT INTO derived_thread_type (id, name, description)
  VALUES (-6, 'MEAN', 'MEAN (nulls are 0 value)');
INSERT INTO derived_thread_type (id, name, description)
  VALUES (-7, 'STDDEV', 'STDDEV (nulls are 0 value)');

/*************************/
/* CREATE THE TRIAL TABLE */
/*************************/

/* trials are the top level table */

CREATE TABLE trial (
 id                  SERIAL NOT NULL PRIMARY KEY,
 name                VARCHAR,
 /* where did this data come from? */
 data_source         INT,
 /* number of processes */
 node_count          INT,
 /* legacy values - these are actually "max" values - i.e. not all nodes have
  * this many threads */
 contexts_per_node   INT,
 /* how many threads per node? */
 threads_per_context INT,
 /* total number of threads */
 total_threads       INT,
 /* reference to the data source table. */
 FOREIGN KEY(data_source) REFERENCES data_source(id)
   ON DELETE NO ACTION ON UPDATE NO ACTION
);

/*****************************/
/* CREATE THE DATA DIMENSIONS */
/*****************************/

/* threads are the "location" dimension */

CREATE TABLE thread (
 id           SERIAL NOT NULL PRIMARY KEY,
 /* trial this thread belongs to */
 trial        INT NOT NULL,
 /* process rank, really */
 node_rank    INT NOT NULL,
 /* legacy value */
 context_rank INT NOT NULL,
 /* thread rank relative to the process */
 thread_rank  INT NOT NULL,
 /* thread index from 0 to N-1 */
 thread_index INT NOT NULL,
 FOREIGN KEY(trial) REFERENCES trial(id) ON DELETE
   NO ACTION ON UPDATE NO ACTION
);

/* metrics are things like num_calls, num_subroutines, TIME, PAPI
   counters, and derived metrics. */

CREATE TABLE metric (
 id      SERIAL NOT NULL PRIMARY KEY,
 /* trial this value belongs to */
```

```
 trial   INT NOT NULL,
 /* name of the metric */
 name    VARCHAR NOT NULL,
 /* if this metric is derived by one of the tools */
 derived BOOLEAN NOT NULL DEFAULT FALSE,
 FOREIGN KEY(trial) REFERENCES trial(id)
   ON DELETE NO ACTION ON UPDATE NO ACTION
);

/* timers are timers, capturing some interval value.  For callpath or
   phase profiles, the parent refers to the calling function or phase. */

CREATE TABLE timer (
 id                SERIAL NOT NULL PRIMARY KEY,
 /* trial this value belongs to */
 trial             INT NOT NULL,
 /* name of the timer */
 name              VARCHAR NOT NULL,
 /* short name of the timer - without source or parameter info */
 short_name        VARCHAR NOT NULL,
 /* filename */
 source_file       VARCHAR,
 /* line number of the start of the block of code */
 line_number       INT,
 /* line number of the end of the block of code */
 line_number_end   INT,
 /* column number of the start of the block of code */
 column_number     INT,
 /* column number of the end of the block of code */
 column_number_end INT,
 FOREIGN KEY(trial) REFERENCES trial(id)
   ON DELETE NO ACTION ON UPDATE NO ACTION
);

/* timer index on the trial and name columns */
CREATE INDEX timer_trial_index on timer (trial, name);

/***********************************/
/* CREATE THE TIMER RELATED TABLES */
/***********************************/

/* timer groups are the groups such as TAU_DEFAULT,
   MPI, OPENMP, TAU_PHASE, TAU_CALLPATH, TAU_PARAM, etc.
   This mapping table allows for NxN mappings between timers
   and groups */

CREATE TABLE timer_group (
 timer INT,
 group_name  VARCHAR NOT NULL,
 FOREIGN KEY(timer) REFERENCES timer(id)
   ON DELETE NO ACTION ON UPDATE NO ACTION
);

/* index for faster queries into groups */
CREATE INDEX timer_group_index on timer_group (timer, group_name);

/* timer parameters are parameter based profile values.
 * an example is foo (x,y) where x=4 and y=10. In that example,
 * timer would be the index of the timer with the
 * name 'foo (x,y) <x>=<4> <y>=<10>'. This table would have two
 * entries, one for the x value and one for the y value. */

CREATE TABLE timer_parameter (
 timer     INT,
```

```
 parameter_name  VARCHAR NOT NULL,
 parameter_value VARCHAR NOT NULL,
 FOREIGN KEY(timer) REFERENCES timer(id)
   ON DELETE NO ACTION ON UPDATE NO ACTION
);

/* timer callpath have the information about the call graph in a trial.
 * If the profile is "flat", these will all have no parents. Otherwise,
 * the parent points to a node in the callgraph, the calling timer
 * (function). */

CREATE TABLE timer_callpath (
 id        SERIAL NOT NULL PRIMARY KEY,
 /* what timer is this? */
 timer     INT NOT NULL,
 /* what is the parent timer? */
 parent    INT,
 FOREIGN KEY(timer) REFERENCES timer(id)
   ON DELETE NO ACTION ON UPDATE NO ACTION,
 FOREIGN KEY(parent) REFERENCES timer_callpath(id)
   ON DELETE NO ACTION ON UPDATE NO ACTION
);

/* By definition, profiles have no time data. However, there are a few
 * examples where time ranges make sense, such as tracking call stacks
 * or associating metadata to a particular phase. The time_range table
 * is used to give other measurements a time context. The iteration
 * start and end can be used to indicate which loop iterations or
 * calls to a function are relevant for this time range. */

CREATE TABLE time_range (
 id SERIAL NOT NULL PRIMARY KEY,
 /* starting iteration */
 iteration_start INT NOT NULL,
 /* ending iteration. */
 iteration_end INT,
 /* starting timestamp */
 time_start BIGINT NOT NULL,
 /* ending timestamp. */
 time_end BIGINT
);

/* timer_call_data records have the dynamic information for when a node
 * in the callgraph is visited by a thread. If you are tracking dynamic
 * callstacks, you would use the time_range field. If you are storing
 * snapshot data, you would use the time_range field. */

CREATE TABLE timer_call_data (
 id          SERIAL NOT NULL PRIMARY KEY,
 /* what callgraph node is this? */
 timer_callpath       INT NOT NULL,
 /* what thread is this? */
 thread      INT NOT NULL,
 /* how many times this timer was called */
 calls       INT,
 /* how many subroutines this timer called */
 subroutines INT,
 /* what is the time_range? this is for supporting snapshots */
 time_range  INT,
 FOREIGN KEY(timer_callpath) REFERENCES timer_callpath(id)
   ON DELETE NO ACTION ON UPDATE NO ACTION,
 FOREIGN KEY(thread) REFERENCES thread(id)
   ON DELETE NO ACTION ON UPDATE NO ACTION,
 FOREIGN KEY(time_range) REFERENCES time_range(id)
```

```
   ON DELETE NO ACTION ON UPDATE NO ACTION
);

/* timer values have the timer of one timer
   on one thread for one metric, at one location on the callgraph. */

CREATE TABLE timer_value (
 /* what node in the callgraph and thread is this? */
 timer_call_data       INT NOT NULL,
 /* what metric is this? */
 metric                INT NOT NULL,
 /* The inclusive value for this timer */
 inclusive_value       DOUBLE PRECISION,
 /* The exclusive value for this timer */
 exclusive_value       DOUBLE PRECISION,
 /* The inclusive percent for this timer */
 inclusive_percent     DOUBLE PRECISION,
 /* The exclusive percent for this timer */
 exclusive_percent     DOUBLE PRECISION,
 /* The variance for this timer */
 sum_exclusive_squared DOUBLE PRECISION,
 FOREIGN KEY(timer_call_data) REFERENCES timer_call_data(id)
   ON DELETE NO ACTION ON UPDATE NO ACTION,
 FOREIGN KEY(metric) REFERENCES metric(id)
   ON DELETE NO ACTION ON UPDATE NO ACTION
);

/* one metric, one thread, one timer */
CREATE INDEX timer_value_index on timer_value (timer_call_data, metric);

/*************************************/
/* CREATE THE COUNTER RELATED TABLES */
/*************************************/

/* counters measure some counted value. */

CREATE TABLE counter (
 id          SERIAL      NOT NULL PRIMARY KEY,
 trial       INT         NOT NULL,
 name        VARCHAR        NOT NULL,
 FOREIGN KEY(trial) REFERENCES trial(id)
   ON DELETE NO ACTION ON UPDATE NO ACTION
);

/* counter index on the trial and name columns */
CREATE INDEX counter_trial_index on counter (trial, name);

CREATE TABLE counter_value (
 /* what counter is this? */
 counter             INT NOT NULL,
 /* where in the callgraph? */
 timer_callpath      INT,
 /* what thread is this? */
 thread              INT NOT NULL,
 /* The total number of samples */
 sample_count        INT,
 /* The maximum value seen */
 maximum_value       DOUBLE PRECISION,
 /* The minimum value seen */
 minimum_value       DOUBLE PRECISION,
 /* The mean value seen */
 mean_value          DOUBLE PRECISION,
 /* The variance for this counter */
 standard_deviation DOUBLE PRECISION,
```

```
 FOREIGN KEY(counter) REFERENCES counter(id)
   ON DELETE NO ACTION ON UPDATE NO ACTION,
 FOREIGN KEY(timer_callpath) REFERENCES timer_callpath(id)
   ON DELETE NO ACTION ON UPDATE NO ACTION,
 FOREIGN KEY(thread) REFERENCES thread(id)
   ON DELETE NO ACTION ON UPDATE NO ACTION
);

/* one thread, one counter */
CREATE INDEX counter_value_index on counter_value (counter, thread);

/*************************************/
/* CREATE THE METADATA RELATED TABLES */
/*************************************/

/* primary metadata is metadata that is not nested, does not
   contain unique data for each thread. */

CREATE TABLE primary_metadata (
 trial    INT NOT NULL,
 name     VARCHAR NOT NULL,
 value    VARCHAR,
 FOREIGN KEY(trial) REFERENCES trial(id)
   ON DELETE NO ACTION ON UPDATE NO ACTION
);

/* create an index for faster queries against the primary_metadata table */
CREATE INDEX primary_metadata_index on primary_metadata (trial, name);

/* secondary metadata is metadata that could be nested, could
   contain unique data for each thread, and could be an array. */

CREATE TABLE secondary_metadata (
 id       VARCHAR NOT NULL PRIMARY KEY,
 /* trial this value belongs to */
 trial    INT NOT NULL,
 /* this metadata value could be associated with a thread */
 thread   INT,
 /* this metadata value could be associated with a timer that happened */
 timer_callpath    INT,
 /* which call to the context timer was this? */
 time_range    INT,
 /* this metadata value could be a nested structure */
 parent   VARCHAR,
 /* the name of the metadata field */
 name     VARCHAR NOT NULL,
 /* the value of the metadata field */
 value    VARCHAR,
 /* this metadata value could be an array - so tokenize it */
 is_array BOOLEAN DEFAULT FALSE,
 FOREIGN KEY(trial) REFERENCES trial(id)
   ON DELETE NO ACTION ON UPDATE NO ACTION,
 FOREIGN KEY(thread) REFERENCES thread(id)
   ON DELETE NO ACTION ON UPDATE NO ACTION,
 FOREIGN KEY(timer_callpath) REFERENCES timer_callpath(id)
   ON DELETE NO ACTION ON UPDATE NO ACTION,
 FOREIGN KEY(parent) REFERENCES secondary_metadata(id)
   ON DELETE NO ACTION ON UPDATE NO ACTION,
 FOREIGN KEY(time_range) REFERENCES time_range(id)
   ON DELETE NO ACTION ON UPDATE NO ACTION
);

/* create an index for faster queries against the secondary_metadata table */
CREATE INDEX secondary_metadata_index on secondary_metadata
```

```
   (trial, name, thread, parent);

/************************************/
/* CREATE THE METADATA RELATED TABLES */
/************************************/

/* this is the view table, which organizes and filters trials */
create table taudb_view (
    id                    SERIAL            NOT NULL    PRIMARY KEY,
    /* views can be nested */
    parent                INTEGER            NULL,
    /* name of the view */
    name                 VARCHAR    NOT NULL,
    /* view conjoin type for parameters */
    conjoin                VARCHAR    NOT NULL,
    FOREIGN KEY (parent) REFERENCES taudb_view(id)
      ON DELETE CASCADE ON UPDATE CASCADE
);

create table taudb_view_parameter (
    /* the view ID */
    taudb_view            INTEGER    NOT NULL,
    /* the table name for the where clause */
    table_name            VARCHAR    NOT NULL,
    /* the column name for the where clause.
       If the table_name is one of the metadata tables, this is the
       value of the "name" column */
    column_name            VARCHAR    NOT NULL,
    /* the operator for the where clause */
    operator            VARCHAR    NOT NULL,
    /* the value for the where clause */
    value                VARCHAR    NOT NULL,
    FOREIGN KEY (taudb_view) REFERENCES taudb_view(id)
      ON DELETE CASCADE ON UPDATE CASCADE
);

/* simple view of all trials */
INSERT INTO taudb_view (parent, name, conjoin)
    VALUES (NULL, 'All Trials', 'and');
/* must have a parameter or else the sub views for this view
   do not work correctly*/
INSERT INTO taudb_view_parameter
    (taudb_view, table_name, column_name, operator, value)
        VALUES (1, 'trial', 'total_threads', '>', '-1');

/* the application and experiment columns are not used in the
   latest schema, but keeping them makes the code in
   PerfExplorer simpler. */
create table analysis_settings (
    id                  SERIAL          NOT NULL    PRIMARY KEY,
    taudb_view          INTEGER         NULL,
    application         INTEGER         NULL,
    experiment          INTEGER         NULL,
    trial               INTEGER         NULL,
    metric              INTEGER         NULL,
    method              VARCHAR(255)    NOT NULL,
    dimension_reduction VARCHAR(255)    NOT NULL,
    normalization       VARCHAR(255)    NOT NULL,
    FOREIGN KEY (taudb_view) REFERENCES taudb_view(id)
        ON DELETE CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (trial) REFERENCES trial(id)
        ON DELETE CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (metric) REFERENCES metric(id)
        ON DELETE CASCADE ON UPDATE CASCADE
```

```
);

create table analysis_result (
    id                  SERIAL          NOT NULL    PRIMARY KEY,
    analysis_settings   INTEGER         NOT NULL,
    description         VARCHAR(255)    NOT NULL,
    thumbnail_size      INTEGER         NULL,
    image_size          INTEGER         NULL,
    thumbnail           BYTEA           NULL,
    image               BYTEA           NULL,
    result_type         INTEGER         NOT NULL
);

 /* Performance indexes! */
create index trial_name_index on trial(name);
create index timer_name_index on timer(name);
CREATE INDEX timer_callpath_parent on timer_callpath(parent);
CREATE INDEX thread_trial on thread(trial);
CREATE INDEX timer_call_data_timer_callpath on
    timer_call_data(timer_callpath);
CREATE INDEX counter_name_index on counter(name);
CREATE INDEX timer_call_data_thread on timer_call_data(thread);

/* SHORT TERM FIX! These views make sure that charts
   (mostly) work... for now. */

DROP VIEW IF EXISTS interval_location_profile;
DROP VIEW IF EXISTS interval_mean_summary;
DROP VIEW IF EXISTS interval_total_summary;
DROP VIEW IF EXISTS interval_event_value;
DROP VIEW IF EXISTS interval_event;
DROP VIEW IF EXISTS atomic_location_profile;
DROP VIEW IF EXISTS atomic_mean_summary;
DROP VIEW IF EXISTS atomic_total_summary;
DROP VIEW IF EXISTS atomic_event_value;
DROP VIEW IF EXISTS atomic_event;

CREATE OR REPLACE VIEW interval_event
(id, trial, name, group_name, source_file, line_number, line_number_end)
AS
SELECT tcp.id, t.trial, t.name, tg.group_name,
t.source_file, t.line_number, t.line_number_end
FROM timer_callpath tcp
INNER JOIN timer t ON tcp.timer = t.id
INNER JOIN timer_group tg ON tg.timer = t.id;

CREATE OR REPLACE VIEW interval_event_value
(interval_event, node, context, thread, metric, inclusive_percentage,
inclusive, exclusive_percentage, exclusive, call, subroutines,
inclusive_per_call, sum_exclusive_squared)
AS SELECT tcd.timer_callpath, t.node_rank, t.context_rank,
t.thread_rank, tv.metric, tv.inclusive_percent,
tv.inclusive_value, tv.exclusive_percent, tv.exclusive_value, tcd.calls,
tcd.subroutines, tv.inclusive_value / tcd.calls, tv.sum_exclusive_squared
FROM timer_value tv
INNER JOIN timer_call_data tcd on tv.timer_call_data = tcd.id
INNER JOIN thread t on tcd.thread = t.id;

CREATE OR REPLACE VIEW interval_location_profile
AS SELECT * from interval_event_value WHERE thread >= 0;

CREATE OR REPLACE VIEW interval_total_summary
AS SELECT * from interval_event_value WHERE thread = -2;
```

```
CREATE OR REPLACE VIEW interval_mean_summary
AS SELECT * from interval_event_value WHERE thread = -1;


CREATE OR REPLACE VIEW atomic_event
(id, trial, name, group_name, source_file, line_number)
AS SELECT c.id, c.trial, c.name, NULL, NULL, NULL
FROM counter c;

CREATE OR REPLACE VIEW atomic_event_value
(atomic_event, node, context, thread, sample_count,
maximum_value, minimum_value, mean_value, standard_deviation)
AS SELECT cv.counter, t.node_rank, t.context_rank, t.thread_rank,
cv.sample_count, cv.maximum_value, cv.minimum_value, cv.mean_value,
cv.standard_deviation FROM counter_value cv
INNER JOIN thread t ON cv.thread = t.id;

CREATE OR REPLACE VIEW atomic_location_profile
AS SELECT * FROM atomic_event_value WHERE thread >= 0;

CREATE OR REPLACE VIEW atomic_total_summary
AS SELECT * FROM atomic_event_value WHERE thread = -2;

CREATE OR REPLACE VIEW atomic_mean_summary
AS SELECT * FROM atomic_event_value WHERE thread >= -1;
```

Chapter 31. TAUdb C API

31.1. TAUdb C API Overview

The C API for TAUdb is currently under development, but there is a beta version of the API available. The API provides the following capabilities:

- Loading trials from the database
- Inserting trials into the database
- Parsing TAU profile files

31.2. TAUdb C Structures

The C structures are roughly organized as a tree, with a trial object at the root.

- **taudb_trial:** A top-level structure which contains the collections of all the performance data dimensions.
- **taudb_primary_metadata:** Name/value pairs which describe the properties of the trial.
- **taudb_secondary_metadata:** Name/value pairs which describe the properties of the trial. Unlike primary_metadata values, secondary_metadata objects can have complex value types. They are also associated with a measurement context - a thread of execution, a timer, a timestamp, an iteration, etc.
- **taudb_thread:** A structure which represents a thread of execution in the parallel measurement.
- **taudb_time_range:** A structure which holds a time-range value of beginning and ending iteration numbers or timestamps.
- **taudb_metric:** A structure which represents a unit of measurement, such as TIME, FP_OPS, L1_DCM, etc.
- **taudb_timer:** A structure which represents a region of code. For example, a phase, a function, a loop, a basic block, or even a line of code.
- **taudb_timer_parameter:** A structure which represents parameter values, when parameter based profiling is used.
- **taudb_timer_group:** A structure which represents a semantic grouping of timers, such as "I/O", "MPI", "OpenMP", etc.
- **taudb_timer_callpath:** A structure which represents a node in the dynamic callpath tree. Timer_callpaths with a null parent are either top level timers, or a timers in a flat profile.
- **taudb_timer_call_data:** A structure which represents a tuple between a thread of execution and a node on the timer callpath tree.
- **taudb_timer_value:** A structure which represents a tuple between a timer_call_data object and a metric. The timer_value contains the measurement of one metric for one timer on one thread of execution.

- **taudb_counter:** A structure which represents a counter in the profile. For example, the number of bytes transferred on an MPI_Send() timer.
- **taudb_counter_value:** A structure which represents a counter measurement on one thread of execution.

Below are the object definitions, from the TAUdb C header file.

```
#ifndef TAUDB_STRUCTS_H
#define TAUDB_STRUCTS_H 1

#include "time.h"
#include "uthash.h"
#include "taudb_structs.h"

#if defined __TAUDB_POSTGRESQL__
#include "libpq-fe.h"
#elif defined __TAUDB_SQLITE__
#include "sqlite3.h"
#endif

#ifndef boolean
#define TRUE  1
#define FALSE 0
typedef int boolean;
#endif

typedef struct taudb_prepared_statement {
 char* name;
 UT_hash_handle hh; /* hash index for hashing by name */
} TAUDB_PREPARED_STATEMENT;

/* forward declarations to ease objects that need to know about
 * each other and have doubly-linked relationships */

struct taudb_timer_call_data;
struct taudb_timer_value;
struct taudb_timer_callpath;
struct taudb_timer_group;
struct taudb_timer_parameter;
struct taudb_timer;
struct taudb_counter_value;
struct taudb_counter;
struct taudb_primary_metadata;
struct taudb_secondary_metadata;
struct taudb_time_range;
struct taudb_thread;
struct taudb_metric;
struct taudb_trial;
struct perfdmf_experiment;
struct perfdmf_application;

typedef struct taudb_configuration {
  char* jdbc_db_type;    /* to identify DBMS vendor.
                          * postgresql, mysql, h2, derby, etc. */
  char* db_hostname;     /* server host name */
  char* db_portnum;      /* server port number */
  char* db_dbname;       /* the database name at the server */
  char* db_schemaprefix; /* the schema prefix. This is appended to
                          * all table names for some DBMSs */
  char* db_username;     /* the database username */
```

```
  char* db_password;     /* the database password for username */
  char* db_schemafile;   /* full or relative path to the schema file,
                          * used for configuration, not used in C API */
} TAUDB_CONFIGURATION;

typedef enum taudb_database_schema_version {
  TAUDB_2005_SCHEMA,
  TAUDB_2012_SCHEMA
} TAUDB_SCHEMA_VERSION;

typedef struct taudb_data_source {
 int id;
 char* name;
 char*description;
 UT_hash_handle hh1; /* hash index for hashing by id */
 UT_hash_handle hh2; /* hash index for hashing by name */
} TAUDB_DATA_SOURCE;

typedef struct taudb_connection {
  TAUDB_CONFIGURATION *configuration;
#if defined __TAUDB_POSTGRESQL__
  PGconn *connection;
  PGresult *res;
  TAUDB_PREPARED_STATEMENT *statements;
#elif defined __TAUDB_SQLITE__
  sqlite3 *connection;
  sqlite3_stmt *ppStmt;
  int rc;
#endif
  TAUDB_SCHEMA_VERSION schema_version;
  boolean inTransaction;
  boolean inPortal;
  TAUDB_DATA_SOURCE* data_sources_by_id;
  TAUDB_DATA_SOURCE* data_sources_by_name;
} TAUDB_CONNECTION;

/* these are the derived thread indexes. */

#define TAUDB_MEAN_WITHOUT_NULLS -1
#define TAUDB_TOTAL -2
#define TAUDB_STDDEV_WITHOUT_NULLS -3
#define TAUDB_MIN -4
#define TAUDB_MAX -5
#define TAUDB_MEAN_WITH_NULLS -6
#define TAUDB_STDDEV_WITH_NULLS -7

/* trials are the top level structure */

typedef struct taudb_trial {
 /* actual data from the database */
 int id;
 char* name;
 struct taudb_data_source* data_source;
 int node_count;            /* i.e. number of processes. */
 int contexts_per_node;     /* rarely used, usually 1. */
 int threads_per_context;   /* max number of threads per process
                             * (can be less on individual processes) */
 int total_threads;         /* total number of threads */
 /* arrays of data for this trial */
 struct taudb_metric* metrics_by_id;
 struct taudb_metric* metrics_by_name;
 struct taudb_thread* threads;
 struct taudb_time_range* time_ranges;
 struct taudb_timer* timers_by_id;
```

```
 struct taudb_timer* timers_by_name;
 struct taudb_timer_group* timer_groups;
 struct taudb_timer_callpath* timer_callpaths_by_id;
 struct taudb_timer_callpath* timer_callpaths_by_name;
 struct taudb_timer_call_data* timer_call_data_by_id;
 struct taudb_timer_call_data* timer_call_data_by_key;
 struct taudb_counter* counters_by_id;
 struct taudb_counter* counters_by_name;
 struct taudb_counter_value* counter_values;
 struct taudb_primary_metadata* primary_metadata;
 struct taudb_secondary_metadata* secondary_metadata;
 struct taudb_secondary_metadata* secondary_metadata_by_key;
} TAUDB_TRIAL;

/*********************************************/
/* data dimensions */
/*********************************************/

/* thread represents one physical & logical
 * location for a measurement. */

typedef struct taudb_thread {
 int id; /* database id, also key to hash */
 struct taudb_trial* trial;
 int node_rank;    /* which process does this thread belong to? */
 int context_rank; /* which context? USUALLY 0 */
 int thread_rank;  /* what is this thread's rank in the process */
 int index;        /* what is this threads OVERALL index?
                    * ranges from 0 to trial.thread_count-1 */
 struct taudb_secondary_metadata* secondary_metadata;
 UT_hash_handle hh;
} TAUDB_THREAD;

/* metrics are things like TIME, PAPI counters, and derived metrics. */

typedef struct taudb_metric {
 int id; /* database value, also key to hash */
 char* name; /* key to hash hh2 */
 boolean derived;  /* was this metric measured, or created by a
                    * post-processing tool? */
 UT_hash_handle hh1; /* hash index for hashing by id */
 UT_hash_handle hh2; /* hash index for hashing by name */
} TAUDB_METRIC;

/* Time ranges are ways to delimit the profile data within time ranges.
   They are also useful for secondary metadata which is associated with
   a specific call to a function. */

typedef struct taudb_time_range {
 int id; /* database value, also key to hash */
 int iteration_start;
 int iteration_end;
 uint64_t time_start;
 uint64_t time_end;  /* was this metric measured,
                      * or created by a post-processing tool? */
 UT_hash_handle hh;
} TAUDB_TIME_RANGE;

/* timers are interval timers, capturing some interval value.
 * For callpath or phase profiles, the parent refers to the calling
 * function or phase.  Timers can also be sample locations, or
 * phases (dynamic or static), or sample aggregations (intermediate) */

typedef struct taudb_timer {
```

```
 int id; /* database value, also key to hash */
 struct taudb_trial* trial;  /* pointer back to trial - NOTE: Necessary? */
 char* name;  /* the full timer name, can have file, line, etc. */
 char* short_name;  /* just the function name, for example */
 char* source_file;  /* what source file does this function live in? */
 int line_number;  /* what line does the timer start on? */
 int line_number_end;  /* what line does the timer end on? */
 int column_number;  /* what column number does the timer start on? */
 int column_number_end;  /* what column number does the timer end on? */
 struct taudb_timer_group* groups; /* hash of groups,
                                    * using group hash handle hh2 */
 struct taudb_timer_parameter* parameters; /* array of parameters */
 UT_hash_handle trial_hash_by_id;  /* hash key for id lookup */
 UT_hash_handle trial_hash_by_name;  /* hash key for name lookup
                                      * in temporary hash */
 UT_hash_handle group_hash_by_name;  /* hash key for name lookup
                                      * in timer group */
} TAUDB_TIMER;

/*********************************************/
/* timer related structures  */
/*********************************************/

/* timer groups are the groups such as tau_default,
   mpi, openmp, tau_phase, tau_callpath, tau_param, etc.
   this mapping table allows for nxn mappings between timers
   and groups */

typedef struct taudb_timer_group {
 char* name;
 struct taudb_timer* timers;   /* hash of timers,
                                * using timer hash handle hh3 */
 UT_hash_handle trial_hash_by_name;  // hash handle for trial
 UT_hash_handle timer_hash_by_name;  // hash handle for timers
} TAUDB_TIMER_GROUP;

/* timer parameters are parameter based profile values.
   an example is foo (x,y) where x=4 and y=10. in that example,
   timer would be the index of the timer with the
   name 'foo (x,y) <x>=<4> <y>=<10>'. this table would have two
   entries, one for the x value and one for the y value.
   The parameter can also be a phase / iteration index.
*/

typedef struct taudb_timer_parameter {
 char* name;
 char* value;
 UT_hash_handle hh;
} TAUDB_TIMER_PARAMETER;

/* callpath objects contain the merged dynamic callgraph tree seen
 * during execution */

typedef struct taudb_timer_callpath {
 int id; /* link back to database, and hash key */
 struct taudb_timer* timer; /* which timer is this? */
 struct taudb_timer_callpath *parent; /* callgraph parent */
 char* name; /* a string which has the aggregated callpath. */
 UT_hash_handle hh1; /* hash key for hash by id */
 UT_hash_handle hh2; /* hash key for name (a => b => c...) lookup */
} TAUDB_TIMER_CALLPATH;

/* timer_call_data objects are observations of a node of the callgraph
   for one of the threads. */
```

```
typedef struct taudb_call_data_key {
 struct taudb_timer_callpath *timer_callpath; /* link back to database */
 struct taudb_thread *thread; /* link back to database, roundabout way */
 char* timestamp; /* timestamp in case we are in a snapshot or something */
} TAUDB_TIMER_CALL_DATA_KEY;

typedef struct taudb_timer_call_data {
 int id; /* link back to database */
 TAUDB_TIMER_CALL_DATA_KEY key; /* hash table key */
 int calls;  /* number of times this timer was seen */
 int subroutines;  /* number of timers this timer calls */
 struct taudb_timer_value* timer_values;
 UT_hash_handle hh1;
 UT_hash_handle hh2;
} TAUDB_TIMER_CALL_DATA;

/* finally, timer_values are specific measurements during one of the
   observations of the node of the callgraph on a thread. */

typedef struct taudb_timer_value {
 struct taudb_metric* metric;   /* which metric is this? */
 double inclusive;              /* the inclusive value of this metric */
 double exclusive;              /* the exclusive value of this metric */
 double inclusive_percentage;   /* the inclusive percentage of
                                   * total time of the application */
 double exclusive_percentage;   /* the exclusive percentage of
                                   * total time of the application */
 double sum_exclusive_squared;  /* how much variance did we see
                                   * every time we measured this timer? */
 char *key; /* hash table key - metric name */
 UT_hash_handle hh;
} TAUDB_TIMER_VALUE;

/*******************************************/
/* counter related structures  */
/*******************************************/

/* counters measure some counted value. An example would be MPI message size
 * for an MPI_Send.  */

typedef struct taudb_counter {
 int id; /* database reference */
 struct taudb_trial* trial;
 char* name;
 UT_hash_handle hh1; /* hash key for hashing by id */
 UT_hash_handle hh2; /* hash key for hashing by name */
} TAUDB_COUNTER;

/* counters are atomic counters, not just interval timers */

typedef struct taudb_counter_value_key {
 struct taudb_counter* counter; /* the counter we are measuring */
 struct taudb_thread* thread;   /* where this measurement is */
 struct taudb_timer_callpath* context; /* the calling context (can be null) */
 char* timestamp; /* timestamp in case we are in a snapshot or something */
} TAUDB_COUNTER_VALUE_KEY;

typedef struct taudb_counter_value {
 TAUDB_COUNTER_VALUE_KEY key;
 int sample_count;          /* how many times did we see take this count? */
 double maximum_value;      /* what was the max value we saw? */
 double minimum_value;      /* what was the min value we saw? */
 double mean_value;         /* what was the average value we saw? */
```

```
 double standard_deviation; /* how much variance was there? */
 UT_hash_handle hh1; /* hash key for hashing by key */
} TAUDB_COUNTER_VALUE;

/*****************************************/
/* metadata related structures  */
/*****************************************/

/* primary metadata is metadata that is not nested, does not
   contain unique data for each thread. */

typedef struct taudb_primary_metadata {
 char* name;
 char* value;
 UT_hash_handle hh; /* uses the name as the key */
} TAUDB_PRIMARY_METADATA;

/* primary metadata is metadata that could be nested, could
   contain unique data for each thread, and could be an array. */

typedef struct taudb_secondary_metadata_key {
 struct taudb_timer_callpath *timer_callpath; /* link back to database */
 struct taudb_thread *thread; /* link back to database, roundabout way */
 struct taudb_secondary_metadata* parent; /* self-referencing */
 struct taudb_time_range* time_range;
 char* name;
} TAUDB_SECONDARY_METADATA_KEY;

typedef struct taudb_secondary_metadata {
 char* id; /* link back to database */
 TAUDB_SECONDARY_METADATA_KEY key;
 int num_values; /* can have arrays of data */
 char** value;
 int child_count;
 struct taudb_secondary_metadata* children; /* self-referencing  */
 UT_hash_handle hh; /* uses the id as a compound key */
 UT_hash_handle hh2; /* uses the key as a compound key */
} TAUDB_SECONDARY_METADATA;

/* these are for supporting the older schema */

typedef struct perfdmf_experiment {
 int id;
 char* name;
 struct taudb_primary_metadata* primary_metadata;
} PERFDMF_EXPERIMENT;

typedef struct perfdmf_application {
 int id;
 char* name;
 struct taudb_primary_metadata* primary_metadata;
} PERFDMF_APPLICATION;

#endif /* TAUDB_STRUCTS_H */
```

31.3. TAUdb C API

```
#ifndef TAUDB_API_H
#define TAUDB_API_H 1
```

```
#include "taudb_structs.h"

/* when a "get" function is called, this global has the number of
   top-level objects that are returned. */
extern int taudb_numItems;

/* the database version */
extern enum taudb_database_schema_version taudb_version;

/* to connect to the database */
extern TAUDB_CONNECTION* taudb_connect_config(char* config_name);
extern TAUDB_CONNECTION* taudb_connect_config_file(char* config_file_name);

/* test the connection status */
extern int taudb_check_connection(TAUDB_CONNECTION* connection);

/* disconnect from the database */
extern int taudb_disconnect(TAUDB_CONNECTION* connection);

/*********************************************/
/* query functions */
/*********************************************/

/* functions to support the old database schema - avoid these if you can */
extern PERFDMF_APPLICATION*
    perfdmf_query_applications(TAUDB_CONNECTION* connection);
extern PERFDMF_EXPERIMENT*
    perfdmf_query_experiments(TAUDB_CONNECTION* connection,
        PERFDMF_APPLICATION* application);
extern PERFDMF_APPLICATION*
    perfdmf_query_application(TAUDB_CONNECTION* connection, char* name);
extern PERFDMF_EXPERIMENT*
    perfdmf_query_experiment(TAUDB_CONNECTION* connection,
        PERFDMF_APPLICATION* application, char* name);
extern TAUDB_TRIAL* perfdmf_query_trials(TAUDB_CONNECTION* connection,
    PERFDMF_EXPERIMENT* experiment);

/* get the data sources */
extern TAUDB_DATA_SOURCE*
    taudb_query_data_sources(TAUDB_CONNECTION* connection);
extern TAUDB_DATA_SOURCE*
    taudb_get_data_source_by_id(TAUDB_DATA_SOURCE* data_sources,
        const int id);
extern TAUDB_DATA_SOURCE*
    taudb_get_data_source_by_name(TAUDB_DATA_SOURCE* data_sources,
        const char* name);

/* using the properties set in the filter, find a set of trials */
extern TAUDB_TRIAL*
    taudb_query_trials(TAUDB_CONNECTION* connection, boolean complete,
        TAUDB_TRIAL* filter);
extern TAUDB_PRIMARY_METADATA*
    taudb_query_primary_metadata(TAUDB_CONNECTION* connection,
        TAUDB_TRIAL* filter);
extern TAUDB_PRIMARY_METADATA*
    taudb_get_primary_metadata_by_name(TAUDB_PRIMARY_METADATA* primary_metadata,
        const char* name);
extern TAUDB_SECONDARY_METADATA*
    taudb_query_secondary_metadata(TAUDB_CONNECTION* connection,
        TAUDB_TRIAL* filter);

/* get the threads for a trial */
extern TAUDB_THREAD*
    taudb_query_threads(TAUDB_CONNECTION* connection, TAUDB_TRIAL* trial);
```

```
extern TAUDB_THREAD*
    taudb_query_derived_threads(TAUDB_CONNECTION* connection,
        TAUDB_TRIAL* trial);
extern TAUDB_THREAD*
    taudb_get_thread(TAUDB_THREAD* threads, int thread_index);
extern int taudb_get_total_threads(TAUDB_THREAD* threads);

/* get the metrics for a trial */
extern TAUDB_METRIC*
    taudb_query_metrics(TAUDB_CONNECTION* connection, TAUDB_TRIAL* trial);
extern TAUDB_METRIC*
    taudb_get_metric_by_name(TAUDB_METRIC* metrics, const char* name);
extern TAUDB_METRIC*
    taudb_get_metric_by_id(TAUDB_METRIC* metrics, const int id);

/* get the time_ranges for a trial */
extern TAUDB_TIME_RANGE*
    taudb_query_time_range(TAUDB_CONNECTION* connection,
        TAUDB_TRIAL* trial);
extern TAUDB_TIME_RANGE*
    taudb_get_time_range(TAUDB_TIME_RANGE* time_ranges, const int id);

/* get the timers for a trial */
extern TAUDB_TIMER*
    taudb_query_timers(TAUDB_CONNECTION* connection, TAUDB_TRIAL* trial);
extern TAUDB_TIMER*
    taudb_get_timer_by_id(TAUDB_TIMER* timers, int id);
extern TAUDB_TIMER*
    taudb_get_trial_timer_by_name(TAUDB_TIMER* timers, const char* id);
extern TAUDB_TIMER*
    taudb_get_trial_timer_by_name(TAUDB_TIMER* timers, const char* id);
extern TAUDB_TIMER_GROUP*
    taudb_query_timer_groups(TAUDB_CONNECTION* connection,
        TAUDB_TRIAL* trial);
extern void
    taudb_parse_timer_group_names(TAUDB_TRIAL* trial, TAUDB_TIMER* timer,
        char* group_names);
extern TAUDB_TIMER_GROUP*
    taudb_get_timer_group_from_trial_by_name(TAUDB_TIMER_GROUP* timers,
        const char* name);
extern TAUDB_TIMER_GROUP*
    taudb_get_timer_group_from_timer_by_name(TAUDB_TIMER_GROUP* timers,
        const char* name);
extern TAUDB_TIMER_CALLPATH*
    taudb_query_timer_callpaths(TAUDB_CONNECTION* connection,
        TAUDB_TRIAL* trial, TAUDB_TIMER* timer);
extern TAUDB_TIMER_CALLPATH*
    taudb_get_timer_callpath_by_id(TAUDB_TIMER_CALLPATH* timers, int id);
extern TAUDB_TIMER_CALLPATH*
    taudb_get_timer_callpath_by_name(TAUDB_TIMER_CALLPATH* timers,
        const char* id);
extern TAUDB_TIMER_CALLPATH*
    taudb_query_all_timer_callpaths(TAUDB_CONNECTION* connection,
        TAUDB_TRIAL* trial);
extern char* taudb_get_callpath_string(TAUDB_TIMER_CALLPATH* timer_callpath);

/* get the counters for a trial */
extern TAUDB_COUNTER*
    taudb_query_counters(TAUDB_CONNECTION* connection, TAUDB_TRIAL* trial);
extern TAUDB_COUNTER*
    taudb_get_counter_by_id(TAUDB_COUNTER* counters, int id);
extern TAUDB_COUNTER*
    taudb_get_counter_by_name(TAUDB_COUNTER* counters, const char* id);
extern TAUDB_COUNTER_VALUE*
```

```
    taudb_query_counter_values(TAUDB_CONNECTION* connection,
        TAUDB_TRIAL* trial);
TAUDB_COUNTER_VALUE*
    taudb_get_counter_value(TAUDB_COUNTER_VALUE* counter_values,
        TAUDB_COUNTER* counter, TAUDB_THREAD* thread,
        TAUDB_TIMER_CALLPATH* context, char* timestamp);

/* get the timer call data for a trial */
extern TAUDB_TIMER_CALL_DATA*
    taudb_query_timer_call_data(TAUDB_CONNECTION* connection,
        TAUDB_TRIAL* trial, TAUDB_TIMER_CALLPATH* timer_callpath,
        TAUDB_THREAD* thread);
extern TAUDB_TIMER_CALL_DATA*
    taudb_query_all_timer_call_data(TAUDB_CONNECTION* connection,
        TAUDB_TRIAL* trial);
extern TAUDB_TIMER_CALL_DATA*
    taudb_query_timer_call_data_stats(TAUDB_CONNECTION* connection,
        TAUDB_TRIAL* trial, TAUDB_TIMER_CALLPATH* timer_callpath,
        TAUDB_THREAD* thread);
extern TAUDB_TIMER_CALL_DATA*
    taudb_query_all_timer_call_data_stats(TAUDB_CONNECTION* connection,
        TAUDB_TRIAL* trial);
extern TAUDB_TIMER_CALL_DATA*
    taudb_get_timer_call_data_by_id(TAUDB_TIMER_CALL_DATA* timer_call_data,
        int id);
extern TAUDB_TIMER_CALL_DATA*
    taudb_get_timer_call_data_by_key(TAUDB_TIMER_CALL_DATA* timer_call_data,
        TAUDB_TIMER_CALLPATH* callpath, TAUDB_THREAD* thread, char* timestamp);

/* get the timer values for a trial */
extern TAUDB_TIMER_VALUE*
    taudb_query_timer_values(TAUDB_CONNECTION* connection,
        TAUDB_TRIAL* trial, TAUDB_TIMER_CALLPATH* timer_callpath,
        TAUDB_THREAD* thread, TAUDB_METRIC* metric);
extern TAUDB_TIMER_VALUE*
    taudb_query_timer_stats(TAUDB_CONNECTION* connection,
        TAUDB_TRIAL* trial, TAUDB_TIMER_CALLPATH* timer_callpath,
        TAUDB_THREAD* thread, TAUDB_METRIC* metric);
extern TAUDB_TIMER_VALUE*
    taudb_query_all_timer_values(TAUDB_CONNECTION* connection,
        TAUDB_TRIAL* trial);
extern TAUDB_TIMER_VALUE*
    taudb_query_all_timer_stats(TAUDB_CONNECTION* connection,
        TAUDB_TRIAL* trial);
extern TAUDB_TIMER_VALUE*
    taudb_get_timer_value(TAUDB_TIMER_CALL_DATA* timer_call_data,
        TAUDB_METRIC* metric);

/* find main */
extern TAUDB_TIMER*
    taudb_query_main_timer(TAUDB_CONNECTION* connection, TAUDB_TRIAL* trial);

/* save everything */
extern void taudb_save_trial(TAUDB_CONNECTION* connection,
    TAUDB_TRIAL* trial, boolean update, boolean cascade);
extern void taudb_save_threads(TAUDB_CONNECTION* connection,
    TAUDB_TRIAL* trial, boolean update);
extern void taudb_save_metrics(TAUDB_CONNECTION* connection,
    TAUDB_TRIAL* trial, boolean update);
extern void taudb_save_timers(TAUDB_CONNECTION* connection,
    TAUDB_TRIAL* trial, boolean update);
extern void taudb_save_time_ranges(TAUDB_CONNECTION* connection,
    TAUDB_TRIAL* trial, boolean update);
extern void taudb_save_timer_groups(TAUDB_CONNECTION* connection,
```

```
    TAUDB_TRIAL* trial, boolean update);
extern void taudb_save_timer_parameters(TAUDB_CONNECTION* connection,
    TAUDB_TRIAL* trial, boolean update);
extern void taudb_save_timer_callpaths(TAUDB_CONNECTION* connection,
    TAUDB_TRIAL* trial, boolean update);
extern void taudb_save_timer_call_data(TAUDB_CONNECTION* connection,
    TAUDB_TRIAL* trial, boolean update);
extern void taudb_save_timer_values(TAUDB_CONNECTION* connection,
    TAUDB_TRIAL* trial, boolean update);
extern void taudb_save_counters(TAUDB_CONNECTION* connection,
    TAUDB_TRIAL* trial, boolean update);
extern void taudb_save_counter_values(TAUDB_CONNECTION* connection,
    TAUDB_TRIAL* trial, boolean update);
extern void taudb_save_primary_metadata(TAUDB_CONNECTION* connection,
    TAUDB_TRIAL* trial, boolean update);
extern void taudb_save_secondary_metadata(TAUDB_CONNECTION* connection,
    TAUDB_TRIAL* trial, boolean update);

/*********************************************/
/* memory functions */
/*********************************************/

extern char* taudb_strdup(const char* in_string);
extern TAUDB_TRIAL* taudb_create_trials(int count);
extern TAUDB_METRIC*             taudb_create_metrics(int count);
extern TAUDB_TIME_RANGE*         taudb_create_time_ranges(int count);
extern TAUDB_THREAD*             taudb_create_threads(int count);
extern TAUDB_SECONDARY_METADATA* taudb_create_secondary_metadata(int count);
extern TAUDB_PRIMARY_METADATA*   taudb_create_primary_metadata(int count);
extern TAUDB_PRIMARY_METADATA*   taudb_resize_primary_metadata(int count,
    TAUDB_PRIMARY_METADATA* old_primary_metadata);
extern TAUDB_COUNTER*            taudb_create_counters(int count);
extern TAUDB_COUNTER_VALUE*      taudb_create_counter_values(int count);
extern TAUDB_TIMER*              taudb_create_timers(int count);
extern TAUDB_TIMER_PARAMETER*    taudb_create_timer_parameters(int count);
extern TAUDB_TIMER_GROUP*        taudb_create_timer_groups(int count);
extern TAUDB_TIMER_GROUP*        taudb_resize_timer_groups(int count,
    TAUDB_TIMER_GROUP* old_groups);
extern TAUDB_TIMER_CALLPATH*     taudb_create_timer_callpaths(int count);
extern TAUDB_TIMER_CALL_DATA*    taudb_create_timer_call_data(int count);
extern TAUDB_TIMER_VALUE*        taudb_create_timer_values(int count);

extern void taudb_delete_trials(TAUDB_TRIAL* trials, int count);

/*********************************************/
/* Adding objects to the hierarchy */
/*********************************************/

extern void taudb_add_metric_to_trial(TAUDB_TRIAL* trial,
    TAUDB_METRIC* metric);
extern void taudb_add_time_range_to_trial(TAUDB_TRIAL* trial,
    TAUDB_TIME_RANGE* time_range);
extern void taudb_add_thread_to_trial(TAUDB_TRIAL* trial,
    TAUDB_THREAD* thread);
extern void taudb_add_secondary_metadata_to_trial(TAUDB_TRIAL* trial,
    TAUDB_SECONDARY_METADATA* secondary_metadata);
extern void taudb_add_secondary_metadata_to_secondary_metadata
    (TAUDB_SECONDARY_METADATA* parent, TAUDB_SECONDARY_METADATA* child);
extern void taudb_add_primary_metadata_to_trial(TAUDB_TRIAL* trial,
    TAUDB_PRIMARY_METADATA* primary_metadata);
extern void taudb_add_counter_to_trial(TAUDB_TRIAL* trial,
    TAUDB_COUNTER* counter);
extern void taudb_add_counter_value_to_trial(TAUDB_TRIAL* trial,
    TAUDB_COUNTER_VALUE* counter_value);
```

```
extern void taudb_add_timer_to_trial(TAUDB_TRIAL* trial,
    TAUDB_TIMER* timer);
extern void taudb_add_timer_parameter_to_trial(TAUDB_TRIAL* trial,
    TAUDB_TIMER_PARAMETER* timer_parameter);
extern void taudb_add_timer_group_to_trial(TAUDB_TRIAL* trial,
    TAUDB_TIMER_GROUP* timer_group);
extern void taudb_add_timer_to_timer_group(TAUDB_TIMER_GROUP* timer_group,
    TAUDB_TIMER* timer);
extern void taudb_add_timer_callpath_to_trial(TAUDB_TRIAL* trial,
    TAUDB_TIMER_CALLPATH* timer_callpath);
extern void taudb_add_timer_call_data_to_trial(TAUDB_TRIAL* trial,
    TAUDB_TIMER_CALL_DATA* timer_call_data);
extern void taudb_add_timer_value_to_timer_call_data
    (TAUDB_TIMER_CALL_DATA* timer_call_data, TAUDB_TIMER_VALUE* timer_value);

/* Profile parsers */
extern TAUDB_TRIAL* taudb_parse_tau_profiles(const char* directory_name);

/* Analysis routines */
extern void taudb_compute_statistics(TAUDB_TRIAL* trial);

/* iterators */
extern TAUDB_DATA_SOURCE*
    taudb_next_data_source_by_name_from_connection
        (TAUDB_DATA_SOURCE* current);
extern TAUDB_DATA_SOURCE*
    taudb_next_data_source_by_id_from_connection
        (TAUDB_DATA_SOURCE* current);
extern TAUDB_THREAD*
    taudb_next_thread_by_index_from_trial(TAUDB_THREAD* current);
extern TAUDB_METRIC*
    taudb_next_metric_by_name_from_trial(TAUDB_METRIC* current);
extern TAUDB_METRIC*
    taudb_next_metric_by_id_from_trial(TAUDB_METRIC* current);
extern TAUDB_TIME_RANGE*
    taudb_next_time_range_by_id_from_trial(TAUDB_TIME_RANGE* current);
extern TAUDB_TIMER*
    taudb_next_timer_by_name_from_trial(TAUDB_TIMER* current);
extern TAUDB_TIMER*
    taudb_next_timer_by_id_from_trial(TAUDB_TIMER* current);
extern TAUDB_TIMER*
    taudb_next_timer_by_name_from_group(TAUDB_TIMER* current);
extern TAUDB_TIMER_GROUP*
    taudb_next_timer_group_by_name_from_trial
        (TAUDB_TIMER_GROUP* current);
extern TAUDB_TIMER_GROUP*
    taudb_next_timer_group_by_name_from_timer
        (TAUDB_TIMER_GROUP* current);
extern TAUDB_TIMER_PARAMETER*
    taudb_next_timer_parameter_by_name_from_timer
        (TAUDB_TIMER_PARAMETER* current);
extern TAUDB_TIMER_CALLPATH*
    taudb_next_timer_callpath_by_name_from_trial
        (TAUDB_TIMER_CALLPATH* current);
extern TAUDB_TIMER_CALLPATH*
    taudb_next_timer_callpath_by_id_from_trial
        (TAUDB_TIMER_CALLPATH* current);
extern TAUDB_TIMER_CALL_DATA*
    taudb_next_timer_call_data_by_key_from_trial
        (TAUDB_TIMER_CALL_DATA* current);
extern TAUDB_TIMER_CALL_DATA*
    taudb_next_timer_call_data_by_id_from_trial
        (TAUDB_TIMER_CALL_DATA* current);
extern TAUDB_TIMER_VALUE*
```

```
    taudb_next_timer_value_by_metric_from_timer_call_data
        (TAUDB_TIMER_VALUE* current);
extern TAUDB_COUNTER*
    taudb_next_counter_by_name_from_trial(TAUDB_COUNTER* current);
extern TAUDB_COUNTER*
    taudb_next_counter_by_id_from_trial(TAUDB_COUNTER* current);
extern TAUDB_COUNTER_VALUE*
    taudb_next_counter_value_by_key_from_trial(TAUDB_COUNTER_VALUE* current);
extern TAUDB_PRIMARY_METADATA*
    taudb_next_primary_metadata_by_name_from_trial
        (TAUDB_PRIMARY_METADATA* current);
extern TAUDB_SECONDARY_METADATA*
    taudb_next_secondary_metadata_by_key_from_trial
        (TAUDB_SECONDARY_METADATA* current);
extern TAUDB_SECONDARY_METADATA*
    taudb_next_secondary_metadata_by_id_from_trial
        (TAUDB_SECONDARY_METADATA* current);

#endif /* TAUDB_API_H */
```

31.4. TAUdb C API Examples

31.4.1. Creating a trial and inserting into the database

```
#include "taudb_api.h"
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <dirent.h>
#include "dump_functions.h"

int main (int argc, char** argv) {
  TAUDB_CONNECTION* connection = NULL;
  if (argc >= 2) {
    connection = taudb_connect_config(argv[1]);
  } else {
    fprintf(stderr, "Please specify a TAUdb config file.\n");
    exit(1);
  }
  printf("Checking connection...\n");
  taudb_check_connection(connection);

  // create a trial
  TAUDB_TRIAL* trial = taudb_create_trials(1);
  trial->name = taudb_strdup("TEST TRIAL");
  // set the data source to "other"
  trial->data_source = taudb_get_data_source_by_id(
      taudb_query_data_sources(connection), 999);

  // create some metadata
  TAUDB_PRIMARY_METADATA* pm = taudb_create_primary_metadata(1);
  pm->name = taudb_strdup("Application");
  pm->value = taudb_strdup("Test Application");
  taudb_add_primary_metadata_to_trial(trial, pm);

  pm = taudb_create_primary_metadata(1);
  pm->name = taudb_strdup("Start Time");
  pm->value = taudb_strdup("2012-11-07 12:30:00");
  taudb_add_primary_metadata_to_trial(trial, pm);
```

```
// alternatively, you can allocate the primary metadata in blocks
pm = taudb_create_primary_metadata(10);
pm[0].name = taudb_strdup("ClientID");
pm[0].value = taudb_strdup("joe_user");
taudb_add_primary_metadata_to_trial(trial, &(pm[0]));
pm[1].name = taudb_strdup("hostname");
pm[1].value = taudb_strdup("hopper04");
taudb_add_primary_metadata_to_trial(trial, &(pm[1]));
pm[2].name = taudb_strdup("Operating System");
pm[2].value = taudb_strdup("Linux");
taudb_add_primary_metadata_to_trial(trial, &(pm[2]));
pm[3].name = taudb_strdup("Release");
pm[3].value = taudb_strdup("2.6.32.36-0.5-default");
taudb_add_primary_metadata_to_trial(trial, &(pm[3]));
pm[4].name = taudb_strdup("Machine");
pm[4].value = taudb_strdup("Hopper.nersc.gov");
taudb_add_primary_metadata_to_trial(trial, &(pm[4]));
pm[5].name = taudb_strdup("CPU Cache Size");
pm[5].value = taudb_strdup("512 KB");
taudb_add_primary_metadata_to_trial(trial, &(pm[5]));
pm[6].name = taudb_strdup("CPU Clock Frequency");
pm[6].value = taudb_strdup("800.000 MHz");
taudb_add_primary_metadata_to_trial(trial, &(pm[6]));
pm[7].name = taudb_strdup("CPU Model");
pm[7].value = taudb_strdup("Quad-Core AMD Opteron(tm) Processor 8378");
taudb_add_primary_metadata_to_trial(trial, &(pm[7]));

// create a metric
TAUDB_METRIC* metric = taudb_create_metrics(1);
metric->name = taudb_strdup("TIME");
taudb_add_metric_to_trial(trial, metric);

// create a thread
TAUDB_THREAD* thread = taudb_create_threads(1);
thread->node_rank = 1;
thread->context_rank = 1;
thread->thread_rank = 1;
thread->index = 1;
taudb_add_thread_to_trial(trial, thread);

// create a timer, timer_callpath, timer_call_data, timer_value
TAUDB_TIMER_GROUP* timer_group = taudb_create_timer_groups(1);
TAUDB_TIMER* timer = taudb_create_timers(1);
TAUDB_TIMER_CALLPATH* timer_callpath = taudb_create_timer_callpaths(1);
TAUDB_TIMER_CALL_DATA* timer_call_data = taudb_create_timer_call_data(1);
TAUDB_TIMER_VALUE* timer_value = taudb_create_timer_values(1);

timer->name = taudb_strdup(
    "int main(int, char **) [{kernel.c} {134,1}-{207,1}]");
timer->short_name = taudb_strdup("main");
timer->source_file = taudb_strdup("kernel.c");
timer->line_number = 134;
timer->column_number = 1;
timer->line_number_end = 207;
timer->column_number_end = 1;
taudb_add_timer_to_trial(trial, timer);

timer_group->name = taudb_strdup("TAU_DEFAULT");
taudb_add_timer_group_to_trial(trial, timer_group);
taudb_add_timer_to_timer_group(timer_group, timer);

timer_callpath->timer = timer;
timer_callpath->parent = NULL;
```

```
  taudb_add_timer_callpath_to_trial(trial, timer_callpath);

  timer_call_data->key.timer_callpath = timer_callpath;
  timer_call_data->key.thread = thread;
  timer_call_data->calls = 1;
  timer_call_data->subroutines = 0;
  taudb_add_timer_call_data_to_trial(trial, timer_call_data);

  timer_value->metric = metric;
  // 5 seconds, or 5 million microseconds
  timer_value->inclusive = 5000000;
  timer_value->exclusive = 5000000;
  timer_value->inclusive_percentage = 100.0;
  timer_value->exclusive_percentage = 100.0;
  timer_value->sum_exclusive_squared = 0.0;
  taudb_add_timer_value_to_timer_call_data(timer_call_data, timer_value);

  // compute stats
  printf("Computing Stats...\n");
  taudb_compute_statistics(trial);

  // save the trial!
  printf("Testing inserts...\n");
  boolean update = FALSE;
  boolean cascade = TRUE;
  taudb_save_trial(connection, trial, update, cascade);

  printf("Disconnecting...\n");
  taudb_disconnect(connection);
  printf("Done.\n");
  return 0;
}
```

31.4.2. Querying a trial from the database

```
#include "taudb_api.h"
#include <stdio.h>
#include <string.h>

void dump_metadata(TAUDB_PRIMARY_METADATA *metadata) {
   printf("%d metadata fields:\n", HASH_COUNT(metadata));
   TAUDB_PRIMARY_METADATA * current;
   for(current = metadata; current != NULL;
       current = taudb_next_primary_metadata_by_name_from_trial(current)) {
     printf("  %s = %s\n", current->name, current->value);
   }
}

void dump_secondary_metadata(TAUDB_SECONDARY_METADATA *metadata) {
   printf("%d secondary metadata fields:\n", HASH_COUNT(metadata));
   TAUDB_SECONDARY_METADATA * current;
   for(current = metadata; current != NULL;
       current = taudb_next_secondary_metadata_by_key_from_trial(current)) {
     printf("  %s = %s\n", current->key.name, current->value[0]);
   }
}

void dump_trial(TAUDB_CONNECTION* connection, TAUDB_TRIAL* filter,
                boolean haveTrial) {
   TAUDB_TRIAL* trial;
```

```
   if (haveTrial) {
     trial = filter;
   } else {
     trial = taudb_query_trials(connection, FALSE, filter);
   }
   TAUDB_TIMER* timer = taudb_query_main_timer(connection, trial);
   printf("Trial name: '%s', id: %d, main: '%s'\n\n",
          trial->name, trial->id, timer->name);
}

int main (int argc, char** argv) {
   printf("Connecting...\n");
   TAUDB_CONNECTION* connection = NULL;
   if (argc >= 2) {
     connection = taudb_connect_config(argv[1]);
   } else {
     fprintf(stderr, "Please specify a TAUdb config file.\n");
     exit(1);
   }
   printf("Checking connection...\n");
   taudb_check_connection(connection);
   printf("Testing queries...\n");

   int t;

   // test the "find trials" method to populate the trial
   TAUDB_TRIAL* filter = taudb_create_trials(1);
   filter->id = atoi(argv[2]);
   TAUDB_TRIAL* trials = taudb_query_trials(connection, TRUE, filter);
   int numTrials = taudb_numItems;
   for (t = 0 ; t < numTrials ; t = t+1) {
      printf("  Trial name: '%s', id: %d\n",
                 trials[t].name, trials[t].id);
      dump_metadata(trials[t].primary_metadata);
      dump_secondary_metadata(trials[t].secondary_metadata);
      dump_trial(connection, &(trials[t]), TRUE);
   }

   printf("Disconnecting...\n");
   taudb_disconnect(connection);
   printf("Done.\n");
   return 0;
}
```