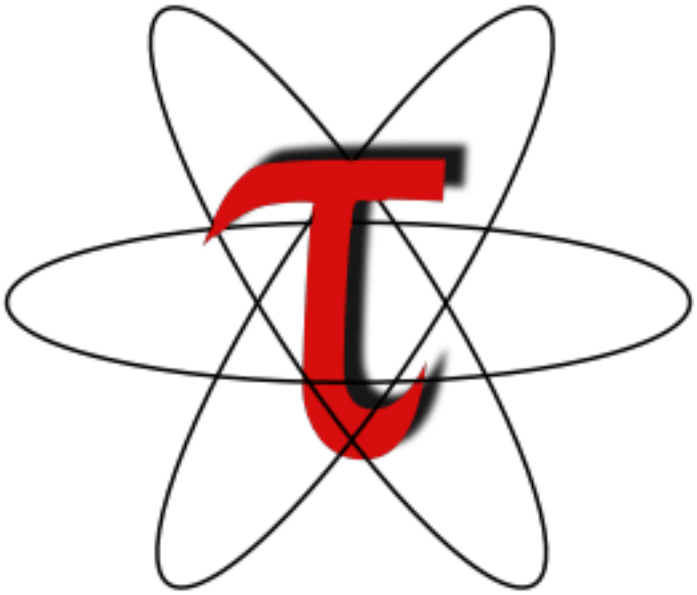
TAU User's Guide



TAU User's Guide

Updated February 26th 2010 for use with version 2.19 or greater.

Table of Contents

List of Figures

List of Tables

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

Chapter 1. Tau Instrumentation

TAU provides three methods to track the performance of your application. Either binary rewriting through Dyninst, through compiler directives or by using PDT to do source transformation. Most projects need a comprehensive picture of where time is spent. The TAU Compiler provides a simple way to automatically instrument an entire project. The TAU Compiler can be used on C, C++, fixed form Fortran, and free form Fortran. 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**
Binary rewriting			Yes			Yes	Yes
Compiler	Yes		Yes	Yes		Yes	Yes
Source	Yes	Yes	Yes	Yes	Yes	Yes	Yes

1.1. Dyninst binary rewriting of applications

This section will cover how to profile your application by rewriting your binary to inserted instrumentation.

The `tau_run` script allows you to instrument an application binary using the Dyninst Tool (requires TAU to be configured with Dyninst). This feature allow instrumentation of already compiled executables without TAU's having to edit the application's code.

To use `tau_run` select the `-o` option to name the rewritten binary:

```
%> tau_run -o a.inst a.out
%> mpirun -np 4 ./a.inst
```

1.2. TAU scripted compilation

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

1.2.1. Compiler Based Instrumentation

TAU provides these scripts: tau_f90.sh, tau_cc.sh, and tau_cxx.sh to instrument and compile Fortran, C, and C++ programs respectively. 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.

1.2.2. Source Based Instrumentation

TAU provides these scripts: tau_f90.sh, tau_cc.sh, and tau_cxx.sh to instrument and compile Fortran, C, and C++ programs respectively. 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.

1.2.3. Option 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_f90.sh, tau_cc.sh, or tau_cxx.sh`

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

1.3. Selectively Profiling an Application

1.3.1. 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 is only available when using source-level instrumentation (PDT).

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` from the `File > Create Selective Instrumentation File` menu item.

Chapter 2. Profiling

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

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
```

These are some other environment variables you can set to enable these advanced MPI measurement features. TAU_TRACK_MESSAGE tracks MPI message statistics when profiling or messages lines when tracing. TAU_COMM_MATRIX generates MPI communication matrix data.

2.2. Reducing Performance Overhead with TAU_THROTTLE

TAU's automatically throttles short running functions in an effort to reduce the amount of overhead associated with profile 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
```

to change the values to 2 million and 5 microseconds per call.

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). Instrumentation overhead will increase with the depth of the callpath so it is a good idea to see if profiling with a short call-path depth 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 provide a uniform interface to access these performance counters.

To use these counters, First find out which PAPI events your system supports, 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 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 there is an incompatible in the choice of these three metrics: PAPI_LD_INS, PAPI_SR_INS, and PAPI_L1_DCM. 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 ...
```

event chooser verifies that PAPI_LD_INS and PAPI_SR_INS compatible metrics.

Next, make sure that you a 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, 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 does not show the temporal aspect of performance variations. Tracing the execution of a parallel program shows when and where an event occurred, in terms of the process that executed it and the location in the source code. This chapter discusses how TAU can be used to generate event traces.

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 a 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 Application

4.1. Text summary

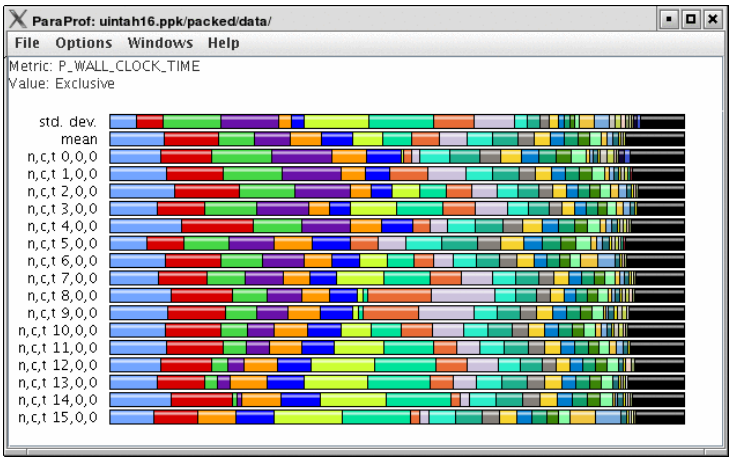
For a quick view summary of TAU performance use `pprof` It reads and print a summary of the TAU data in the current directory. For performance data with multiple metrics move into one of the directories to get information about that metric:

```
%> 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 use 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.

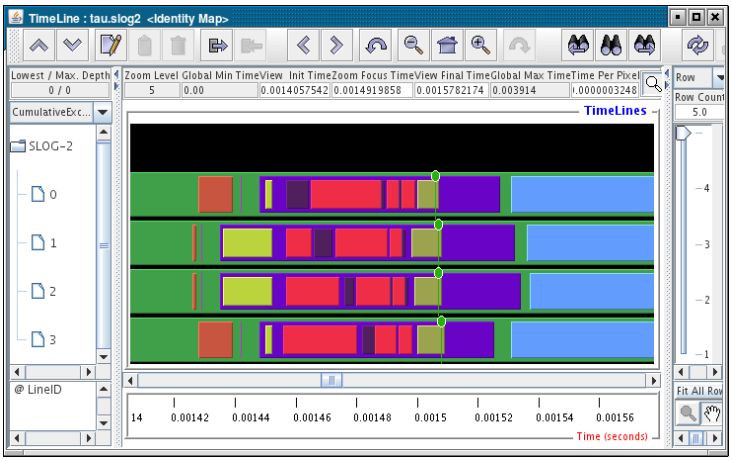
4.3. Jumpshot

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`
Compiler wrappers (Compiler instrumentation)

`tau_cc.sh / tau_cxx.sh / tau_f90.sh`
Compiler wrappers (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

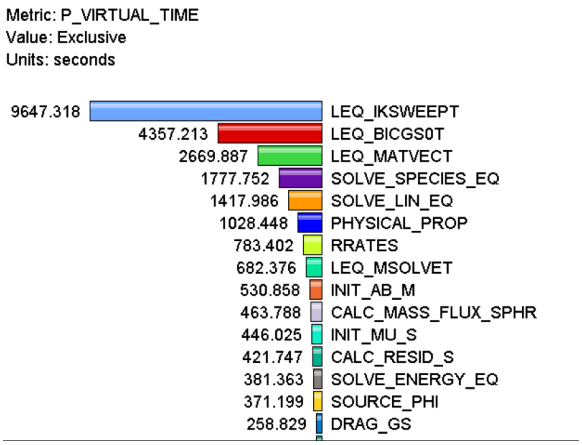
`tau2otf / tau2vtf / tau2slog2`
Trace conversion tools

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



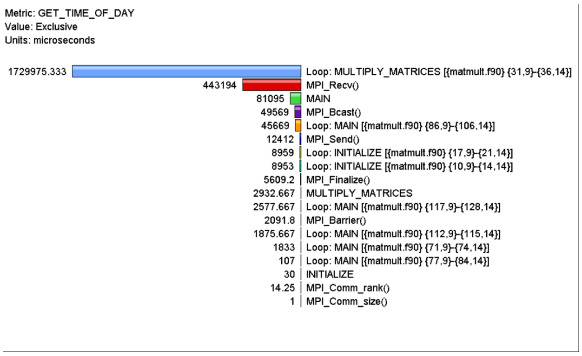
Here is how to generate a flat profile with MPI

```
% setenv TAU_MAKEFILE /opt/apps/tau/tau-2.17.1/x86_64/lib/Makefile.tau-mpi-pdt-pgi

% set path=(/opt/apps/tau/tau-2.17.1/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



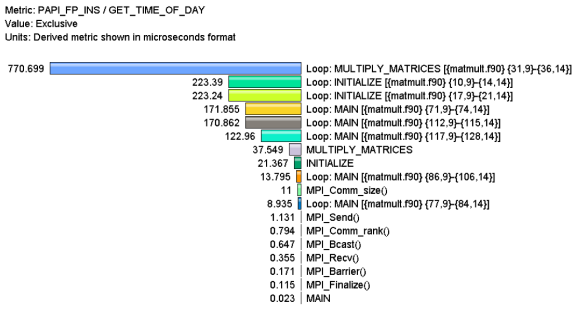
Here is how to instrument loops in an application

```
% setenv TAU_MAKEFILE /opt/apps/tau/tau-2.17.1/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/tau-2.17.1/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



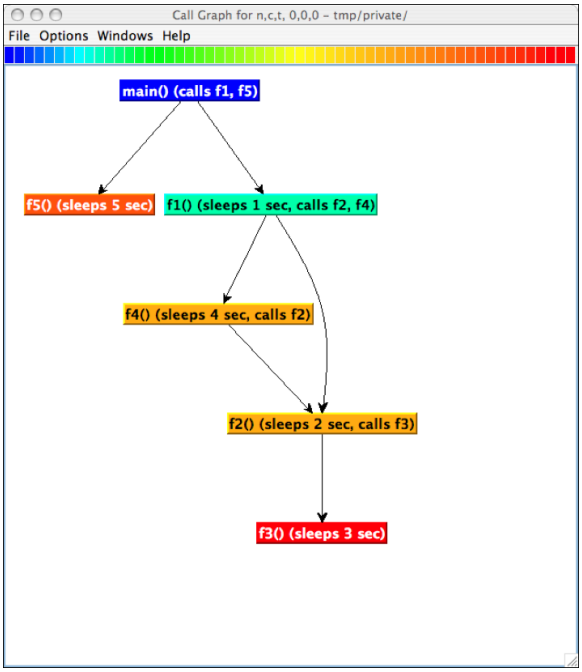
Here is how to generate a flat profile with FP operations

```
% setenv TAU_MAKEFILE /opt/apps/tau/tau-2.17.1/x86_64/lib/Makefile.tau-papi-mpi-pd
% setenv TAU_OPTIONS '-optTauSelectFile=select.tau –optVerbose'
% cat select.tau
  BEGIN_INSTRUMENT_SECTION
  loops routine="#"
  END_INSTRUMENT_SECTION

% set path=(/opt/apps/tau/tau-2.17.1/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/tau-2.17.1/x86_64/lib/Makefile.tau-mpi-pdt
% set path=(/opt/apps/tau/tau-2.17.1/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.

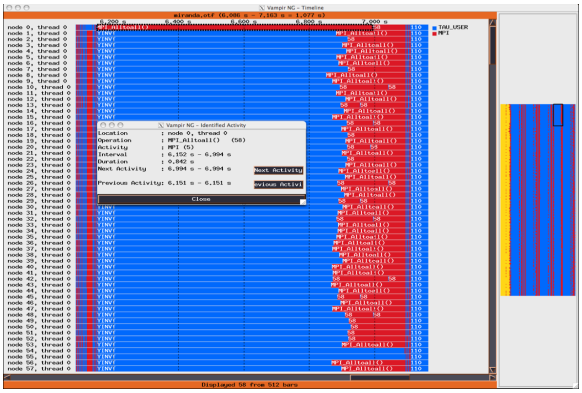
Here to instrument python code

```
% setenv TAU_MAKEFILE /opt/apps/tau/tau-2.17.1/x86_64/lib/Makefile.tau-icpc-python
% set path=(/opt/apps/tau/tau-2.17.1/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



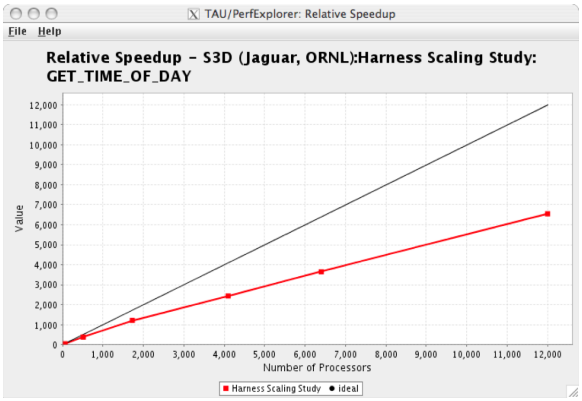
How to create a trace

```
% setenv TAU_MAKEFILE
% /opt/apps/tau/tau-2.17.1/x86_64/lib/Makefile.tau-mpi-pdt-pgi
% set path=(/opt/apps/tau/tau-2.17.1/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/tau-2.17.1/x86_64/lib/Makefile.tau-mpi-pdt
% set path=(/opt/apps/tau/tau-2.17.1/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:
% perfdmf_configure
(Choose derby, blank user/password, yes to save password, defaults)
% perfexplorer_configure
(Yes to load schema, defaults)
% paraprof
(load each trial: Right click on trial ->Upload trial to DB
% perfexplorer
(Charts -> Speedup)
```