gprof, maps, TAU & mpiP

Kent Milfeld

February 28, 2008

THE UNIVERSITY OF TEXAS AT AUSTIN
Texas Advanced Computing Center

THE UNIVERSITY OF TEXAS AT AUSTIN

Login

- login to Rangeer, using your train<xx> account:

  ssh -X train<xx> ranger.tacc.utexas.edu

- Untar the file tau.tar file (in ~train00) into your directory:

  tar xvf ~train00/lab_tau_long.tar
  cd tau_long
  source sourceme.csh   {c shell}
  or
  .   sourceme.sh   {Bourne shell}

***Completed runs are available in lab_tau_long_done.tar.**

gprof

Gprof is a utility for discovering how much time subroutines, library calls and intrinsic functions are using. Code must be "instrumented" (compiled) with "-g" (or some similar option) to provide a symbol table in the executable. There are two subroutine calls and two intrinsic functions calls used in the program **long.f**. Instrument this program and run the executable to obtain a (binary) trace, gmon.out, with timing information. On Linux systems the compiler commands are.

Linux: login3% **ifort -g -p long.f90** **{intel compiler}**
Linux: login4% **pgif90 -g -p long.f90** **{pgi compiler}**

(used "source ./sourceme.csh" (C shell) or " . /sourceme.sh" (Bourne shell) to access ifort)

Execute a.out to make a report:

./a.out
gprof > gprof_report
} Interactive Session

Read the man page on gprof to help make sense of the 2nd part of the listing-- the display information about time spend in parent and children processes.

(The **do_gprof** will perform all of the operation above: just execute "do_gprof". Draw a diagram showing the tree structure of the calling sequence.

Memory Maps

Memory maps provide a list of the layout of the TEXT (code), DATA (initialized variables), and BSS (uninitialized variables) memory regions. It is often used to discover which libraries have been loaded by the linker. Compile the **long.f** program, using an appropriate linker option to produce a "load map" of the program.

Linux: login3% **ifort -Wl,-Map,**map_output **long.f90**

You can use the **do_map** script to execute the above command(s), and produce a map. On a Linux machines it will also produce a vector_ report. Where is the cos library function coming from

TAU

First load up the TAU environment.

% cd tau_long (if you are not already there)
% source sourceme.csh or % . sourceme.sh {for C/Bourne shells, respectively}

The PDT is used to instrument your code; but it is necessary to describe all the component that will be used in the instrumentation (mpi, openmp, profiling, counter [PAPI], etc. But these come in a limited combination. First determine what you want to do (profiling, PAPIcounters, tracing, etc.) and the programming paradigm (mpi, openmp), and the compiler. PDT is a require component:

- PDT
- PROFILE
- MPI
- PAPI
- intel
- pgi
- …

TAU

You can view the available combination by using the command:

% tauTypes

These are called TAU stubs for makefiles.

Look in the sourceme.csh or sourceme.sh file. Here are some of the operations done for you:

unload mvapich
swap pgi intel — **Use Intel Compilers**
load mvapich
module load kojak pdtoolkit tau — **Loads Tau & Papi**
Set the TAU_MAKEFILE environment to the correct stub (full path)
setenv TAU_MAKEFILE …/Makefile.tau-icpc-mpi-papi-pdt
setenv TAU_OPTIONS '-optVerbose …' **{see tau_compiler.sh}**

Sets TAU_MAKEFILE

TAU

If you have a single-file program, you can use the Tau compiler wrapper directly:

instead of

mpif90 foo.f90

use

tau_f90.sh foo.f90

Look in the Makefile, to see how this is done for the matmult.f90 problem.

Now make the matmult executable and submit the job with the commands (look over the job script):

```
% make
% qsub job
```

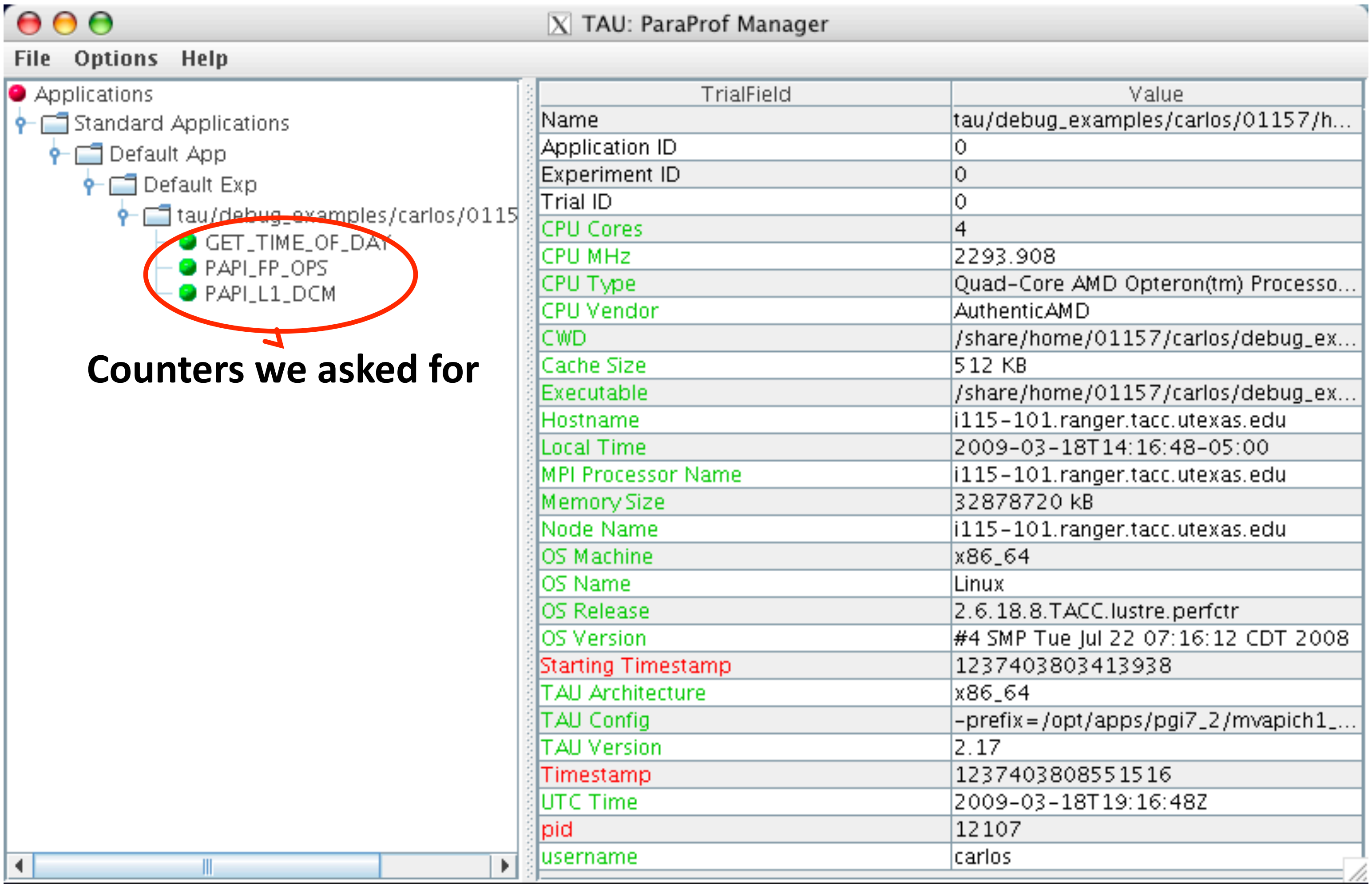
Analyze performance data:

% pprof (for text based profile display)

% paraprof (for GUI)

TAU: ParaProf Manager

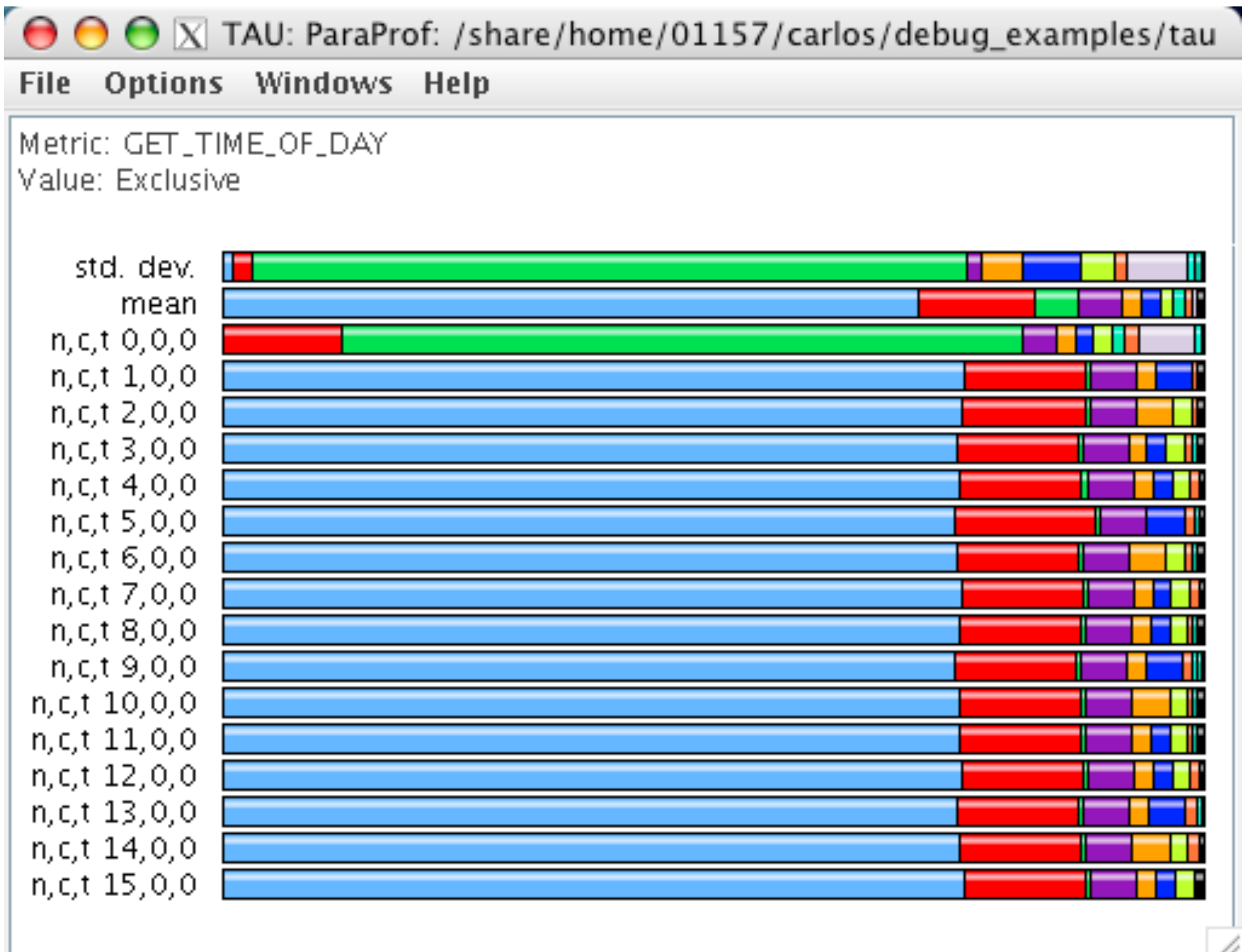
Two windows will appear. This is the manager window, showing the experiment. Note, it has all the details use in the "trial".



TrialField	Value
Name	tau/debug_examples/carlos/01157/h...
Application ID	0
Experiment ID	0
Trial ID	0
CPU Cores	4
CPU MHz	2293.908
CPU Type	Quad-Core AMD Opteron(tm) Processo...
CPU Vendor	AuthenticAMD
CWD	/share/home/01157/carlos/debug_ex...
Cache Size	512 KB
Executable	/share/home/01157/carlos/debug_ex...
Hostname	i115-101.ranger.tacc.utexas.edu
Local Time	2009-03-18T14:16:48-05:00
MPI Processor Name	i115-101.ranger.tacc.utexas.edu
Memory Size	32878720 kB
Node Name	i115-101.ranger.tacc.utexas.edu
OS Machine	x86_64
OS Name	Linux
OS Release	2.6.18.8.TACC.lustre.perfctr
OS Version	#4 SMP Tue Jul 22 07:16:12 CDT 2008
Starting Timestamp	1237403803413938
TAU Architecture	x86_64
TAU Config	-prefix=/opt/apps/pgi7_2/mvapich1_...
TAU Version	2.17
Timestamp	1237403808551516
UTC Time	2009-03-18T19:16:48Z
pid	12107
username	carlos

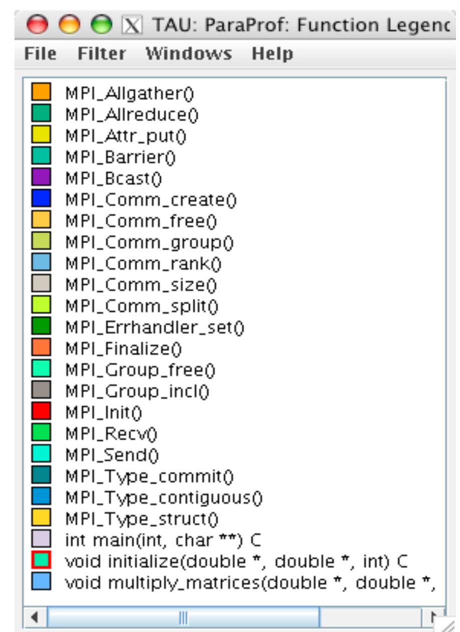
Tau: Metric View

In a second wind, the default “GET_TIME_OF_DAY” profile information will appear. You switch between metrics, by double-clicking on the green bullets next to the names in the manager window (see previous slide).**



Information includes Mean and Standard Deviation

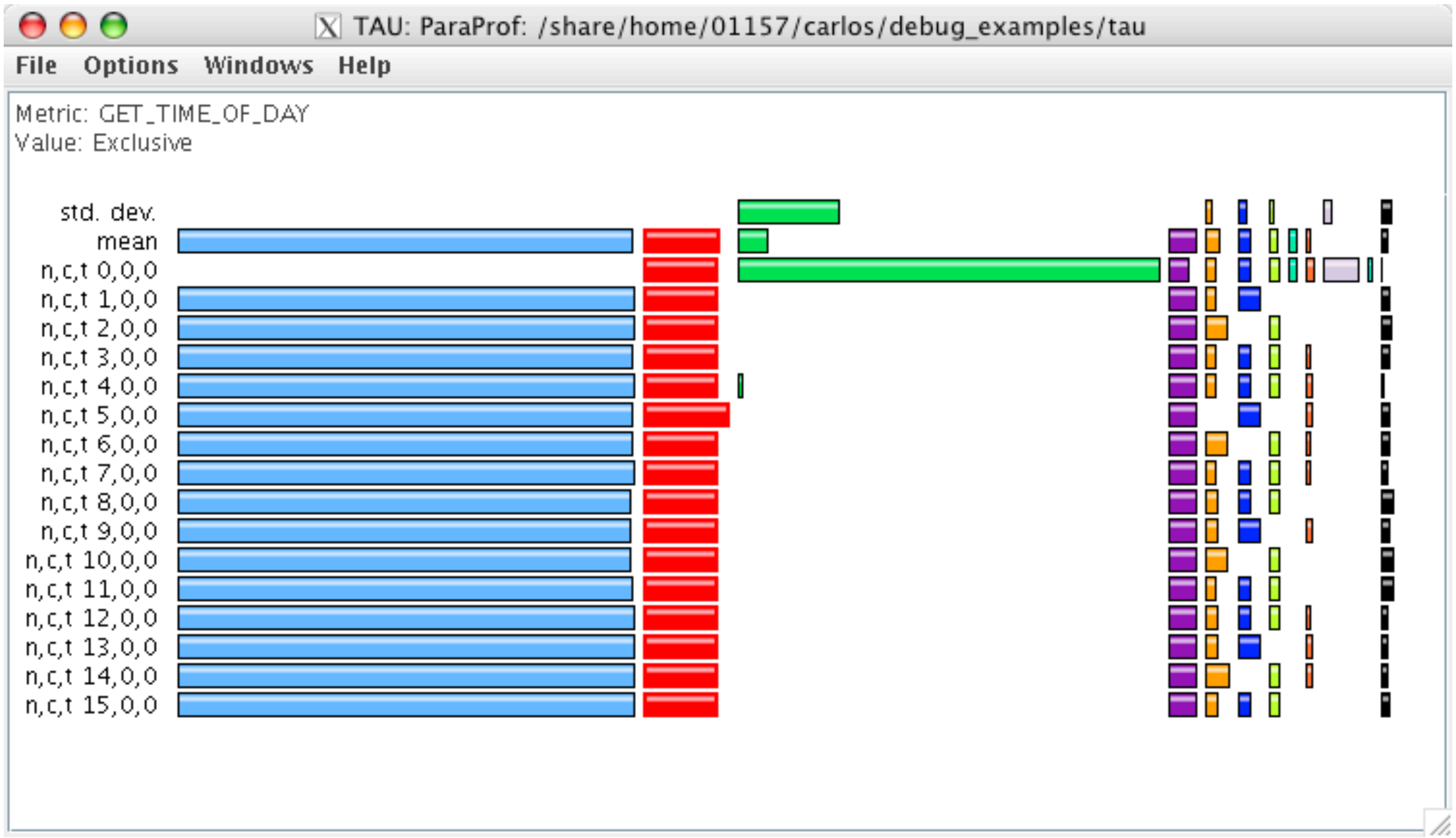
Windows->Function Legend



****This can be a sensitive operation, try several speeds of double clicking.**

Tau: Metric View

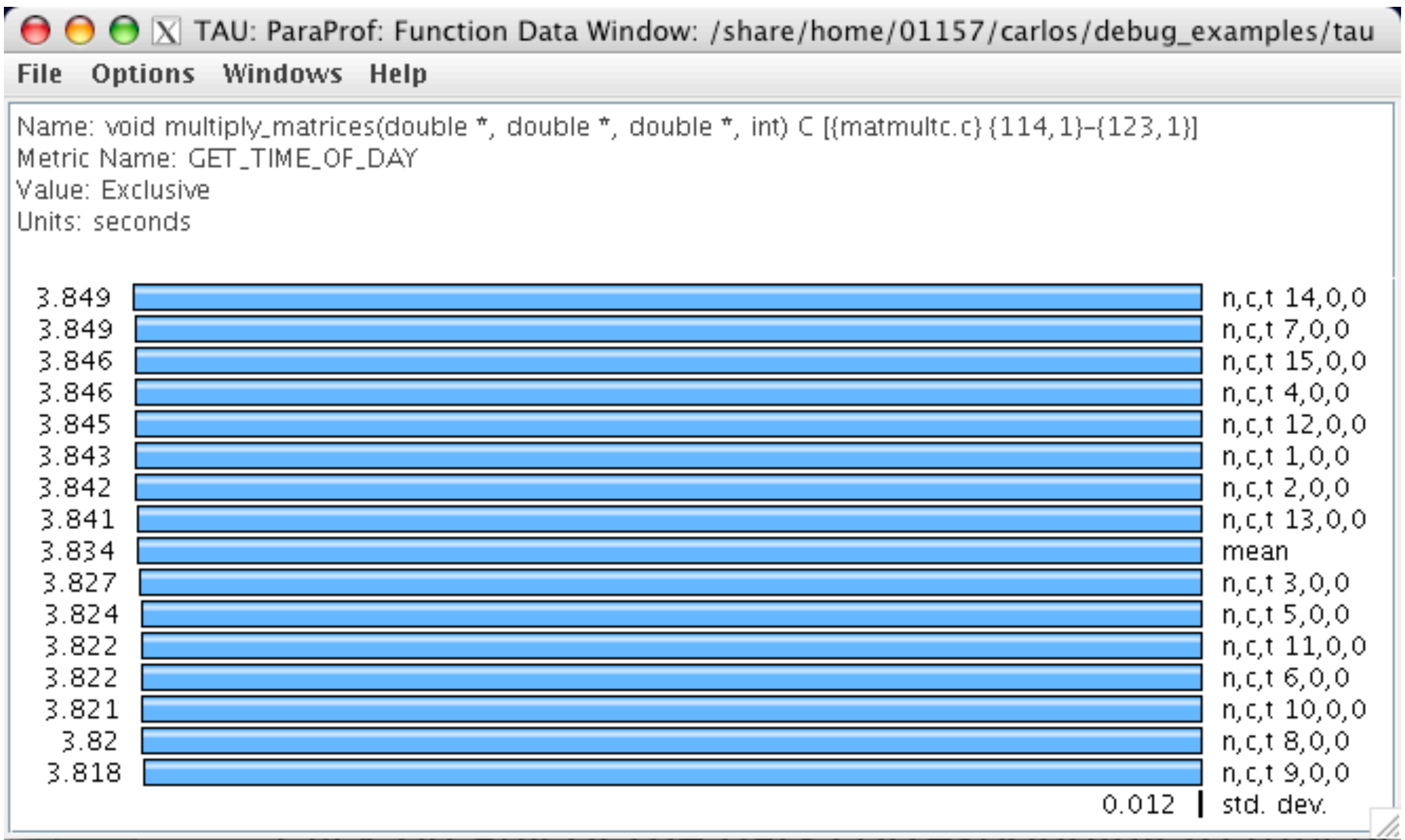
Unstack the bars for clarity: Options -> Stack Bars Together



Hoover over bars with mouse to see time for each function call.

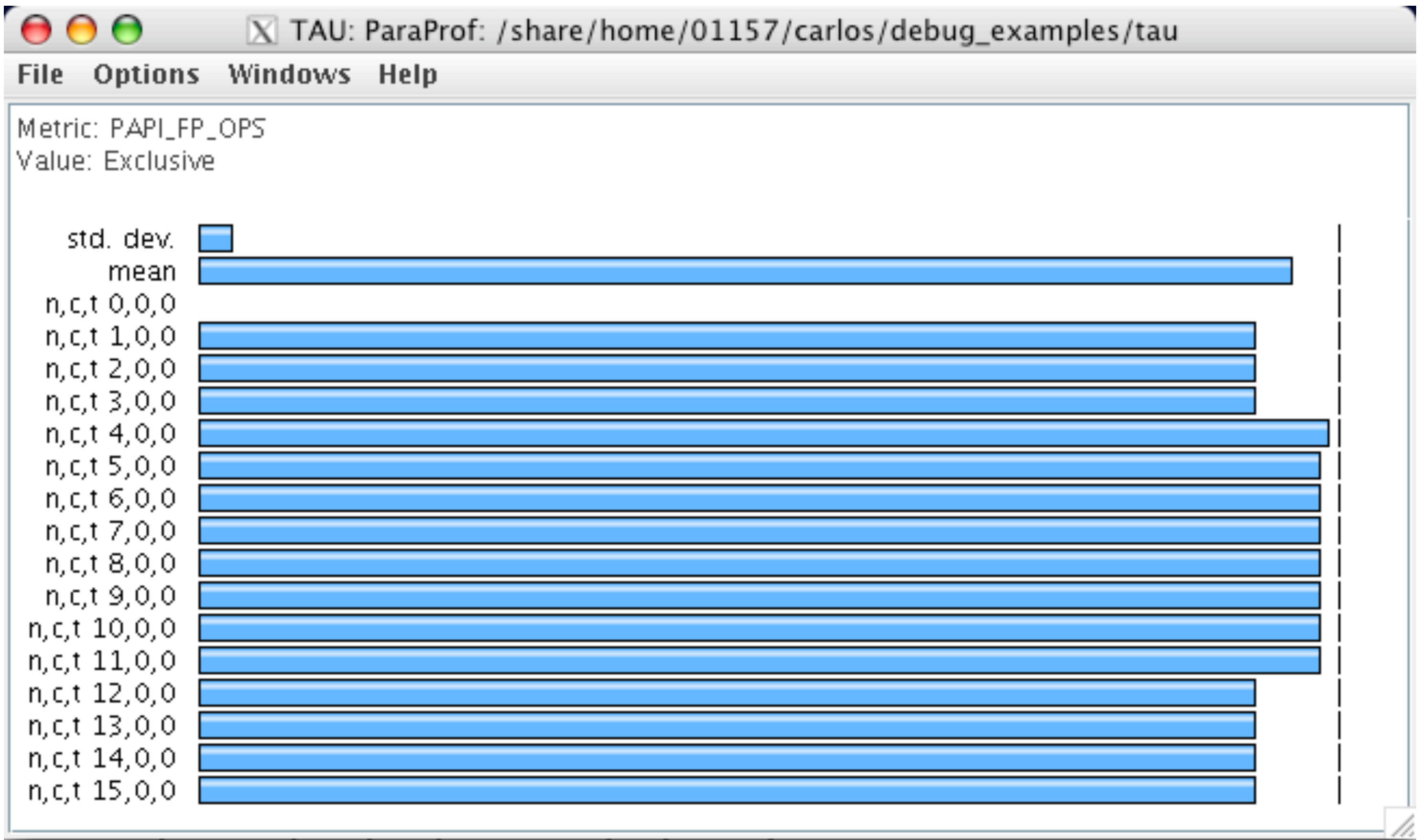
Tau: Function Data Window

Click on any of the bars corresponding to function multiply_matrices. This opens the Function Data Window, which gives a closer look at a single function.



Tau: Float Point OPS

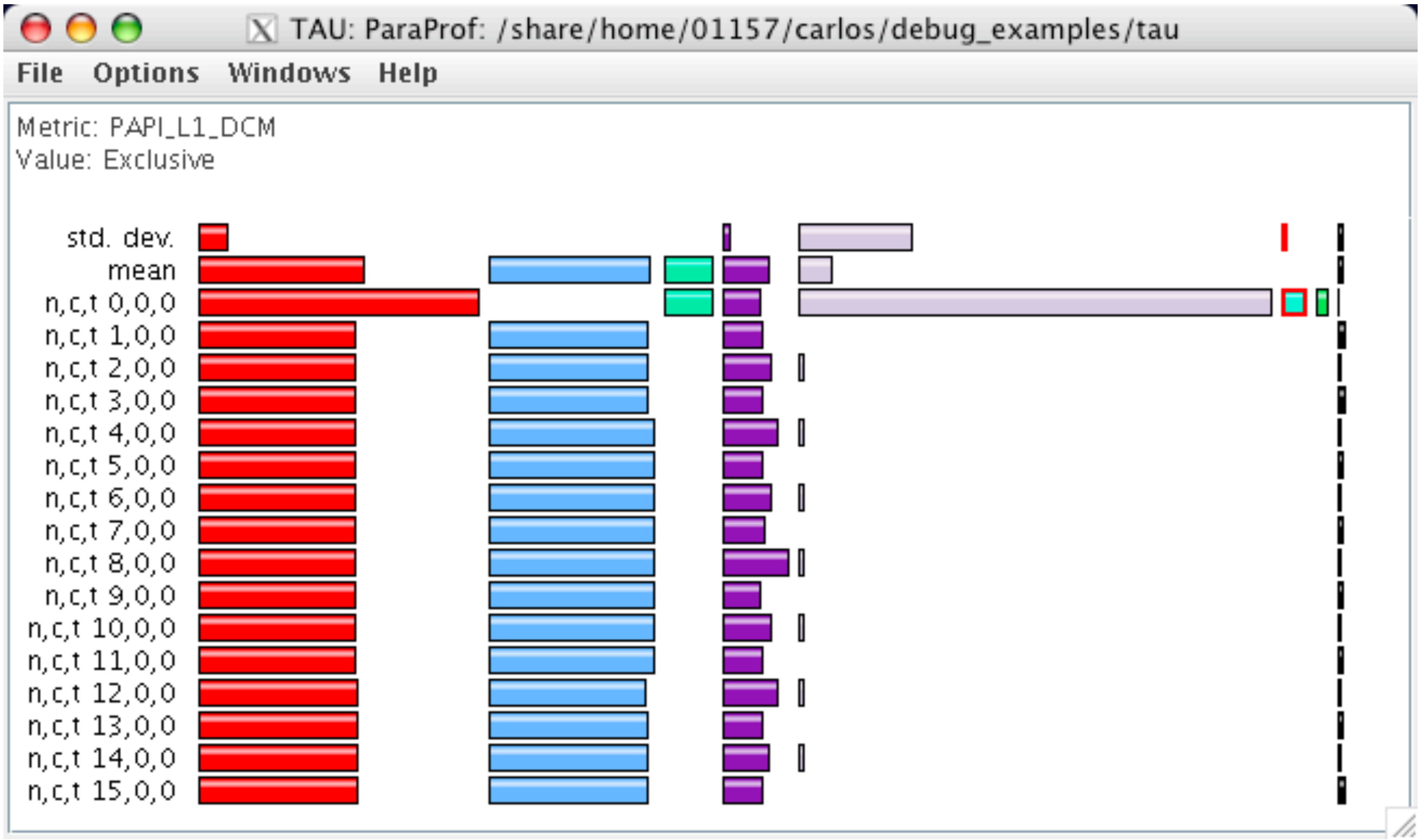
Now, go back and display the PAPI_FP_OPS (double click the PAPI_FP_OPS metric in the manager window. Click a bar corresponding to the function multiply_matrices In the ParaProf Metric Window select Options -> Select Metric -> Exclusive



Note the disparity and grouping of the FLOPS performance.

Tau: L1 Cache Data Misses

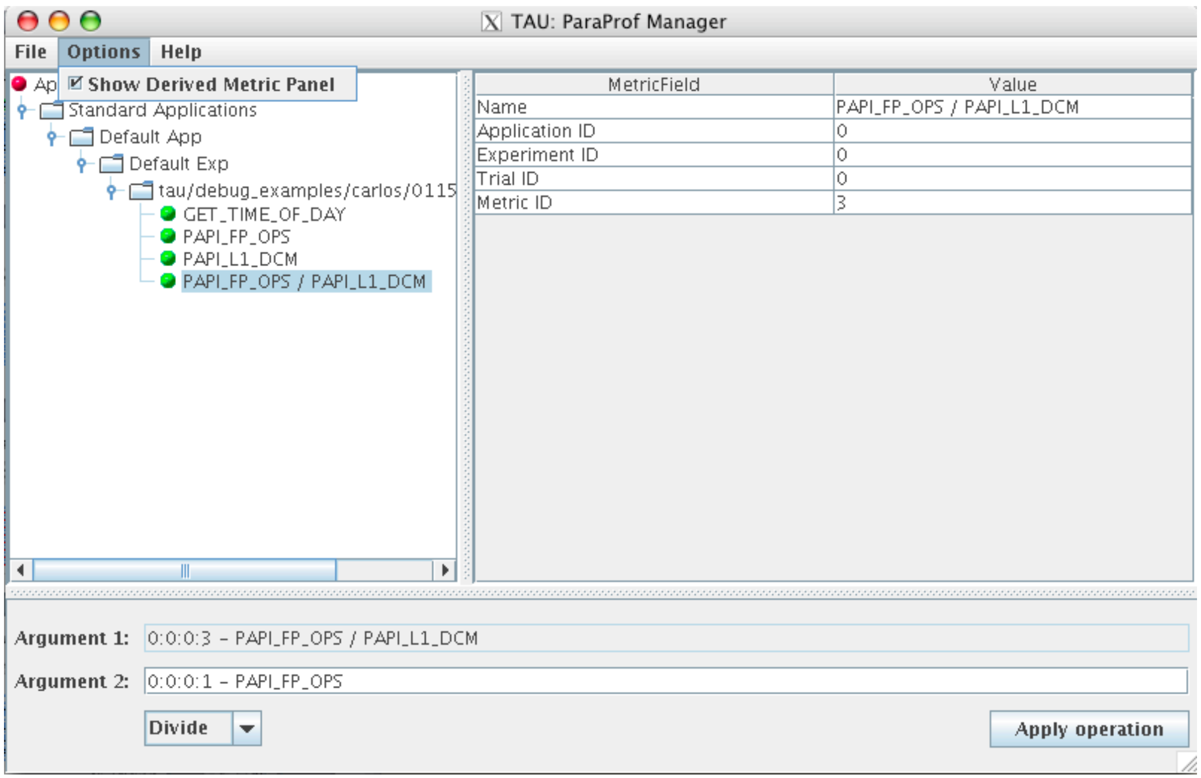
Now, go back and display the PAPI_FP_OPS (double click this metric in the manager window. Select Options -> Stack Bars Together to see differences better.



Now, click a bar corresponding to the function multiply_matrices to see just the matmult results. -- These look similar to the FP_OPS profiles.

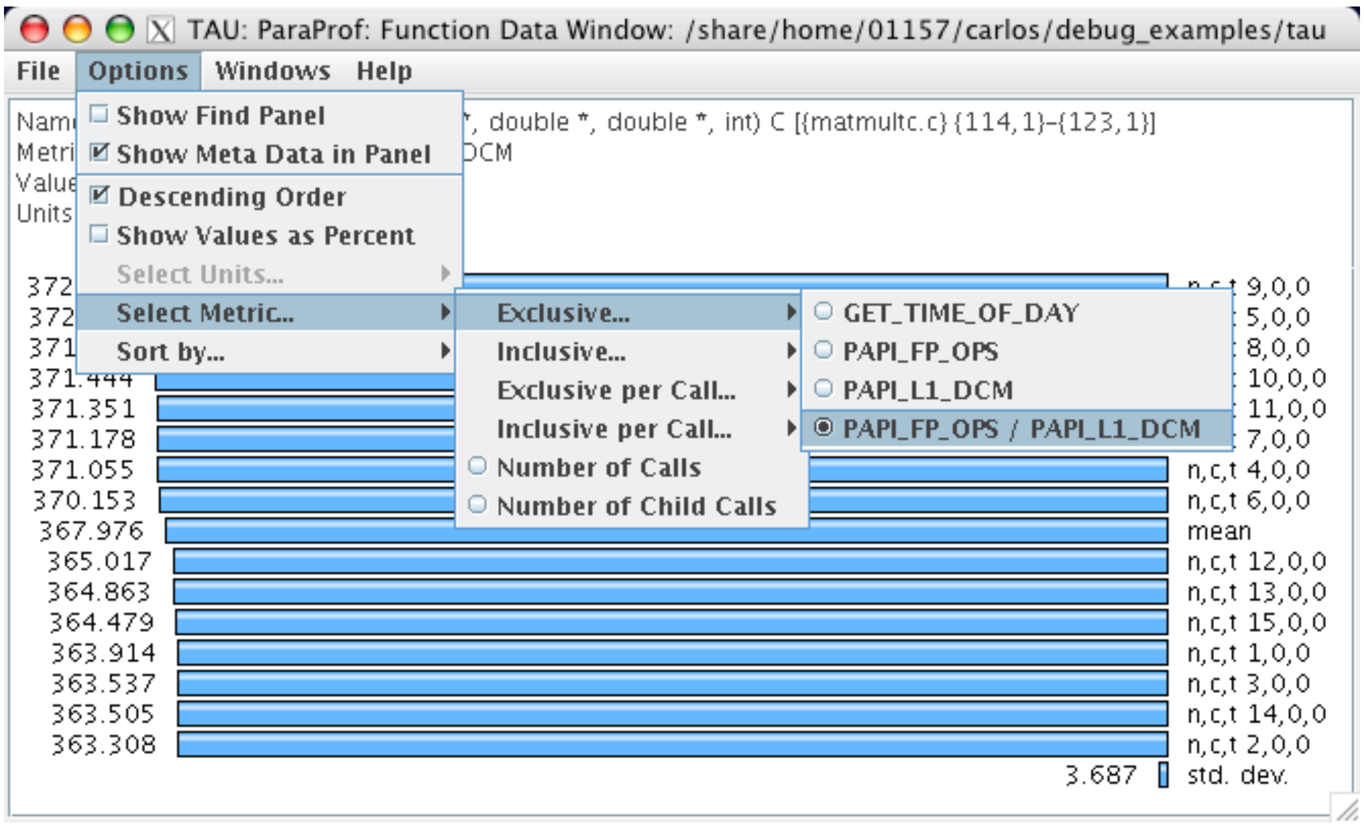
Derived Metrics

- ParaProf Manager Window -> Options -> Show Derived Metrics Panel
- Select Argument 1 (PAPI_L1_DCM) and Argument 2 (PAPI_FP_OPS)
- Select Operation (Division) & Apply



Derived Metrics (Cont.)

- Select a Function
- Function Data Window -> Options -> Select Metric -> Exclusive -> …



Callgraph

Important, save your present Tau data: mkdir save; cp –R MULTI* save

To trace the function calls within the program, follow the same process as before, but use the following **TAU_MAKEFILE**:

`Makefile.tau-callpath-mpi-pdt-pgi`

Here are the commands:

```
% source sourceme_callpath.csh
%    .   sourceme_callpath.sh
% make clean
% make
% qsub job_callpath
```

`cd down to call_path_* and execute paraprof:`

In the Metric View Window, two new options will be available under:

Windows ➔ Thread ➔ Call Graph

Windows ➔ Thread ➔ Call Path Relations

Verify the calling structure of the call tree is similar to the gprof tree.

mpiP

Load the mpiP module:

```
% module load mpiP
% source sourceme_mpip.csh   {C shell}
% source sourceme_mpip.sh    {Bourne shell}
```

Link the static library before any others:

```
% mpif90 -g -L$TACC_MPIP_LIB \
  -lmpiP -lbfd -liberty ./matmultf.f90
```

Try the compilation above with the map option (-Wl,-Map mapout), with and without MPIp, and determine which MPI library is satisfying the code's MPI calls:

```
% mpif90 -Wl,-Map map_mpip -g -L$TACC_MPIP_LIB \
         -lmpiP -lbfd -liberty matmultf.f90

% mpif90 -Wl,-Map map_nompip    matmultf.f90
```

mpiP

Set environmental variables controlling the mpiP output in the job_mpip script (for C shell):

```
% setenv MPIP '-t 10 -k 2' {See job_mpip script.}
```

In this case:

`-t 10` ➔ **only callsites with time > 10% MPI time reported**
`-k 2` ➔ **set callsite stack traceback depth to 2**

Run program through the queue as usual.

```
% qsub job_mpip
```

Display results.

```
% mpipview <app_name><unique number>.mpiP
```

mpiP

In the slave nodes, which collective and point-2-points MPI calls take the most time? What does the master spend mode of its time doing in MPI?