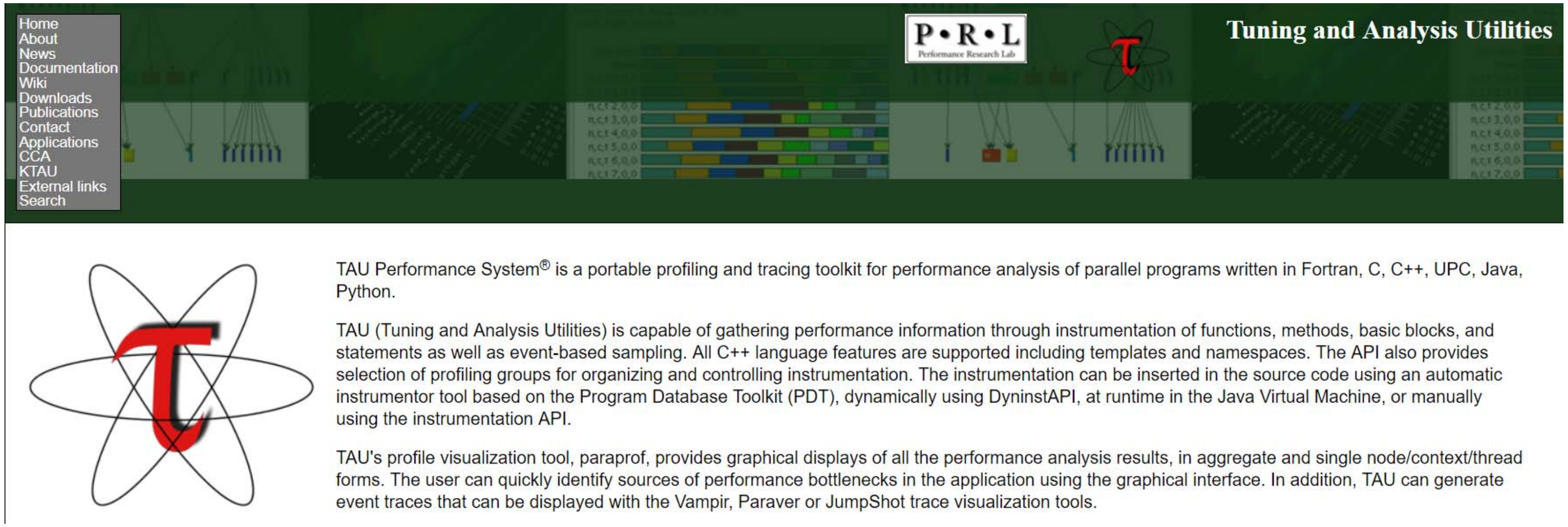
中国科学院大学
University of Chinese Academy of Sciences

The TAU Profiler
Part1：Introduction

人工智能技术学院
缪青海
miaoqh@ucas.ac.cn

TAU



TAU Performance System® is a portable profiling and tracing toolkit for performance analysis of parallel programs written in Fortran, C, C++, UPC, Java, Python.

TAU (Tuning and Analysis Utilities) is capable of gathering performance information through instrumentation of functions, methods, basic blocks, and statements as well as event-based sampling. All C++ language features are supported including templates and namespaces. The API also provides selection of profiling groups for organizing and controlling instrumentation. The instrumentation can be inserted in the source code using an automatic instrumentor tool based on the Program Database Toolkit (PDT), dynamically using DyninstAPI, at runtime in the Java Virtual Machine, or manually using the instrumentation API.

TAU's profile visualization tool, paraprof, provides graphical displays of all the performance analysis results, in aggregate and single node/context/thread forms. The user can quickly identify sources of performance bottlenecks in the application using the graphical interface. In addition, TAU can generate event traces that can be displayed with the Vampir, Paraver or JumpShot trace visualization tools.

- https://www.cs.uoregon.edu/research/tau/home.php

Content

- **Overview**
- Instrumentation
- Profiling
- Tracing
- Analyzing
- Application Scenario

TAU Performance System®

- Tuning and Analysis Utilities (TAU)
- Comprehensive performance profiling and tracing
  - Integrated, scalable, flexible, portable
  - Targets all parallel programming/execution paradigms
- Integrated performance toolkit
  - Instrumentation, measurement, analysis, visualization
  - Widely-ported performance profiling / tracing system
  - Performance data management and data mining
  - Open source (BSD-style license)
- Easy to integrate in application frameworks

What is TAU?

- TAU is a performance evaluation tool. It supports parallel **profiling and tracing**
  - Profiling shows you how much (total) time was spent in each routine;
  - Tracing shows you *when* the events take place in each process along a timeline.

- Profiling and tracing can measure time as well as hardware performance counters (cache misses, instructions) from your CPU

What is TAU?

- TAU can automatically **instrument** your source code using a package called PDT for routines, loops, I/O, memory, phases, etc.
- TAU runs on most HPC platforms and it is free (BSD style license)
- TAU has instrumentation, measurement and analysis tools
  - paraprof is TAU’ s 3D profile browser
- To use TAU’ s automatic source instrumentation, you may set a couple of environment variables and substitute the name of your compiler with a TAU shell script

What Can TAU Do?

- How much time is spent in each application routine and outer *loops*? Within loops, what is the contribution of each *statement*?
- How many instructions are executed in these code regions? Floating point, Level 1 and 2 *data cache misses*, hits, branches taken?
- What is the *peak heap memory* usage of the code? When and where is memory allocated/de-allocated? Are there any memory leaks?

What Can TAU Do?

- How much time does the application spend performing *I/O*? What is the peak read and write *bandwidth* of individual calls, total volume?
- What is the contribution of different *phases* of the program? What is the time wasted/spent waiting for collectives, and I/O operations in Initialization, Computation, I/O phases?
- How does the application *scale*? What is the efficiency, runtime breakdown of performance across different core counts?

Some Features in TAU

- *tau_exec*: A tool to simplify TAU's usage
- PDT: New parsers from ROSE Compiler, LLNL [Dan Quinlan, CASC]
- Power profiling in TAU
- Support for accelerators: CUDA, OpenCL, Intel Xeon Phi Co-processor
- Event based sampling
- OpenMP instrumentation using OpenMP Tools Interface with Intel compilers (OMPT)
- Support for Intel TBB, CILK, and MPC [paratools.com/mpc]
- ParaProf 3D topology displays

What does TAU support?

C/C++ Python UPC OpenCL

CUDA GPI

Fortran MPI

OpenACC Java

pthreads

Intel MIC OpenMP

Intel GNU

LLVM PGI Cray Sun

MinGW

Linux Windows AIX

OS X

NVIDIA Kepler ARM

How does TAU work?

Instrumentation:

- Adds probes to perform measurements
  - Source code instrumentation using pre-processors and compiler scripts
  - Wrapping external libraries (I/O, MPI, Memory, CUDA, OpenCL, pthread)
  - Rewriting the binary executable

How does TAU work?

Measurement:

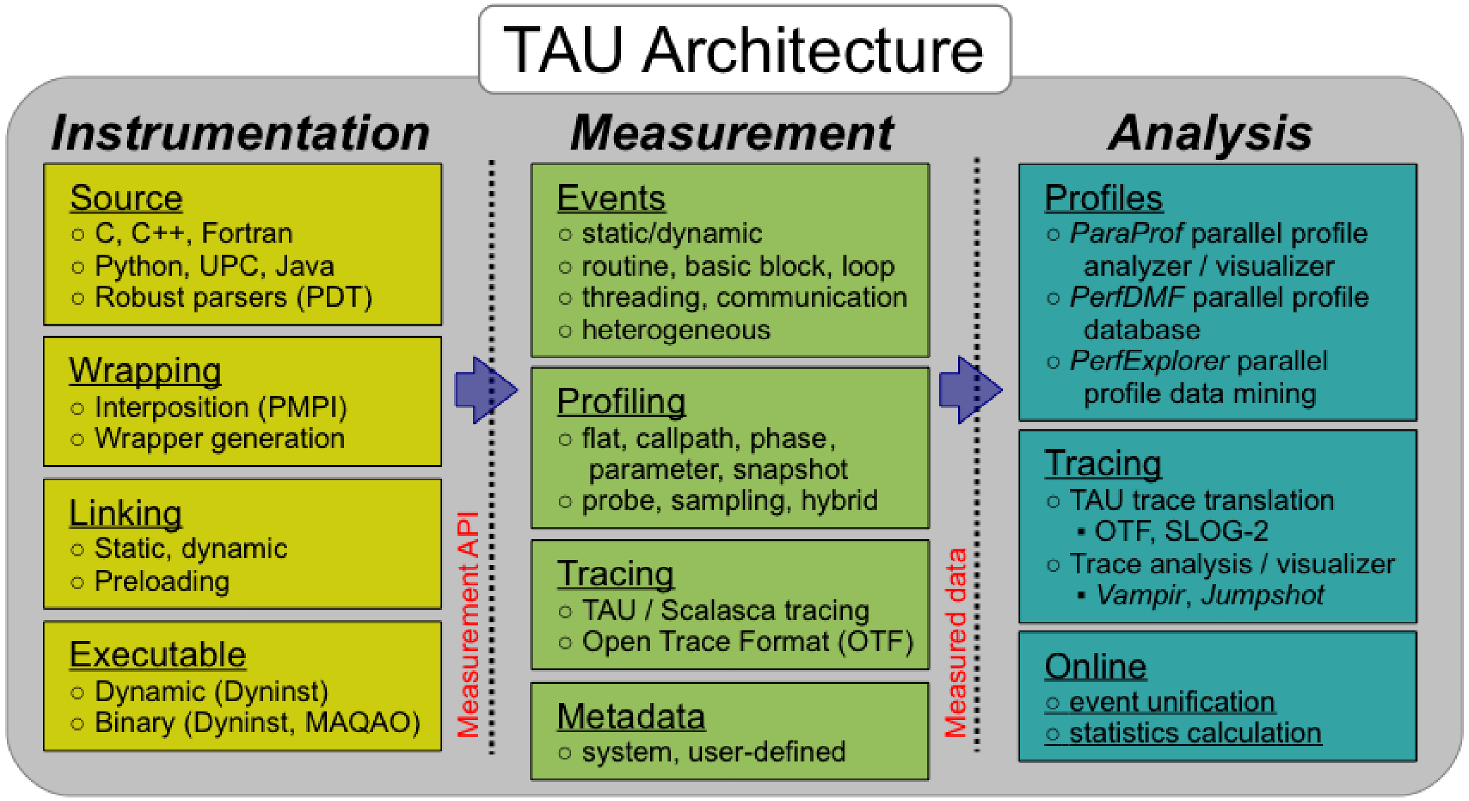
- Profiling or Tracing using **wallclock time** or **hardware counters**:
  - Direct instrumentation (Interval events measure exclusive or inclusive duration)
  - Indirect instrumentation (Sampling measures statement level contribution)
  - Throttling and runtime control of low-level events that execute frequently
  - Per-thread storage of performance data
  - Interface with external packages (e.g. PAPI hw performance counter library)

How does TAU work?

Analysis:

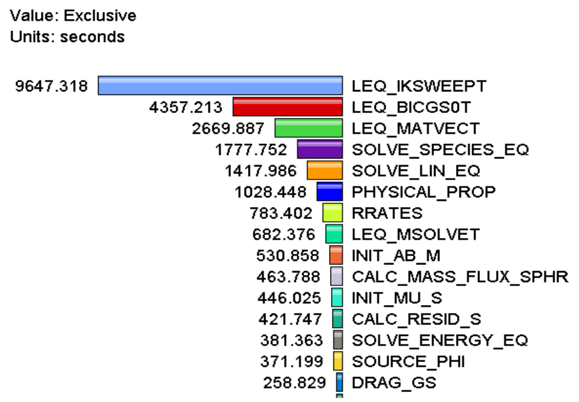
- Visualization of profiles and traces
  - 3D visualization of profile data in paraprof, perfexplorer tools
  - Trace conversion & display in external visualizers (Vampir, Jumpshot, ParaVer)

How does TAU work?

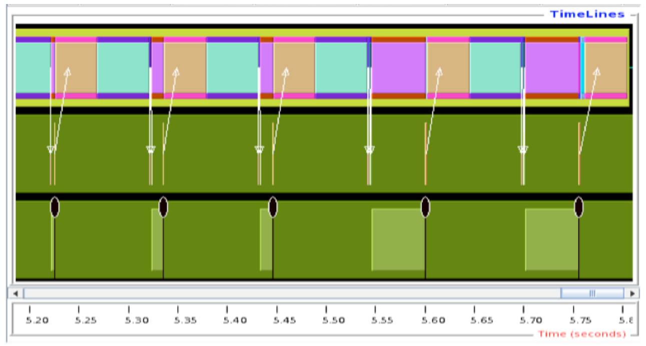


Profiling and Tracing

Profiling



Tracing

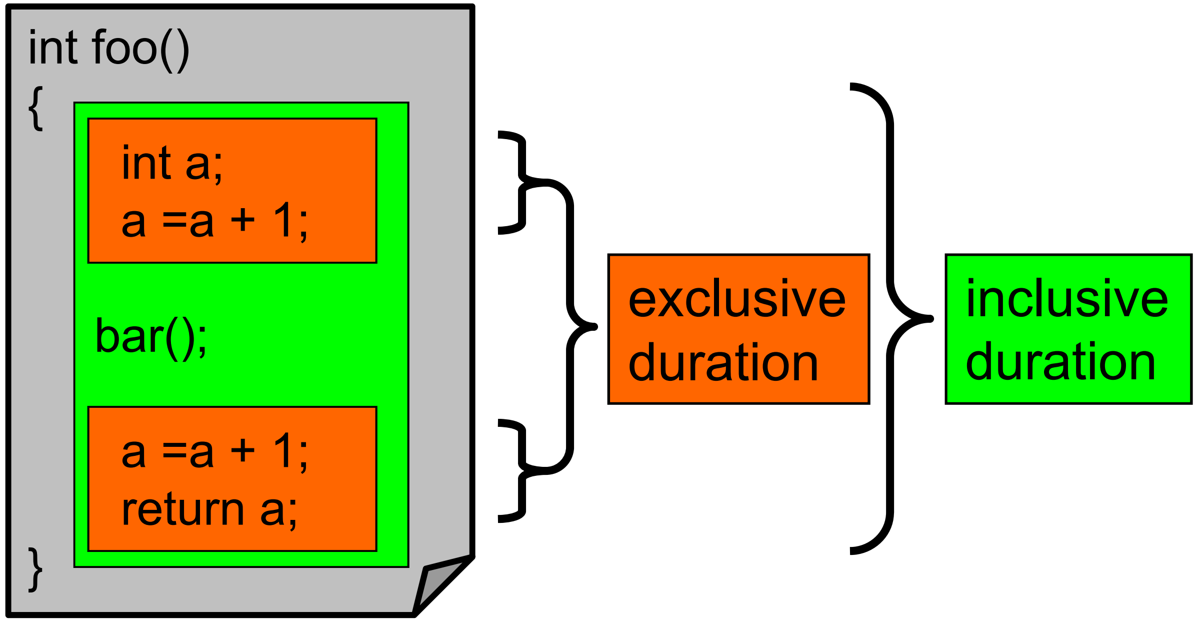


- **Profiling** shows you **how much** (total) time was spent in each routine
- Tracing shows you when the events take place on a timeline

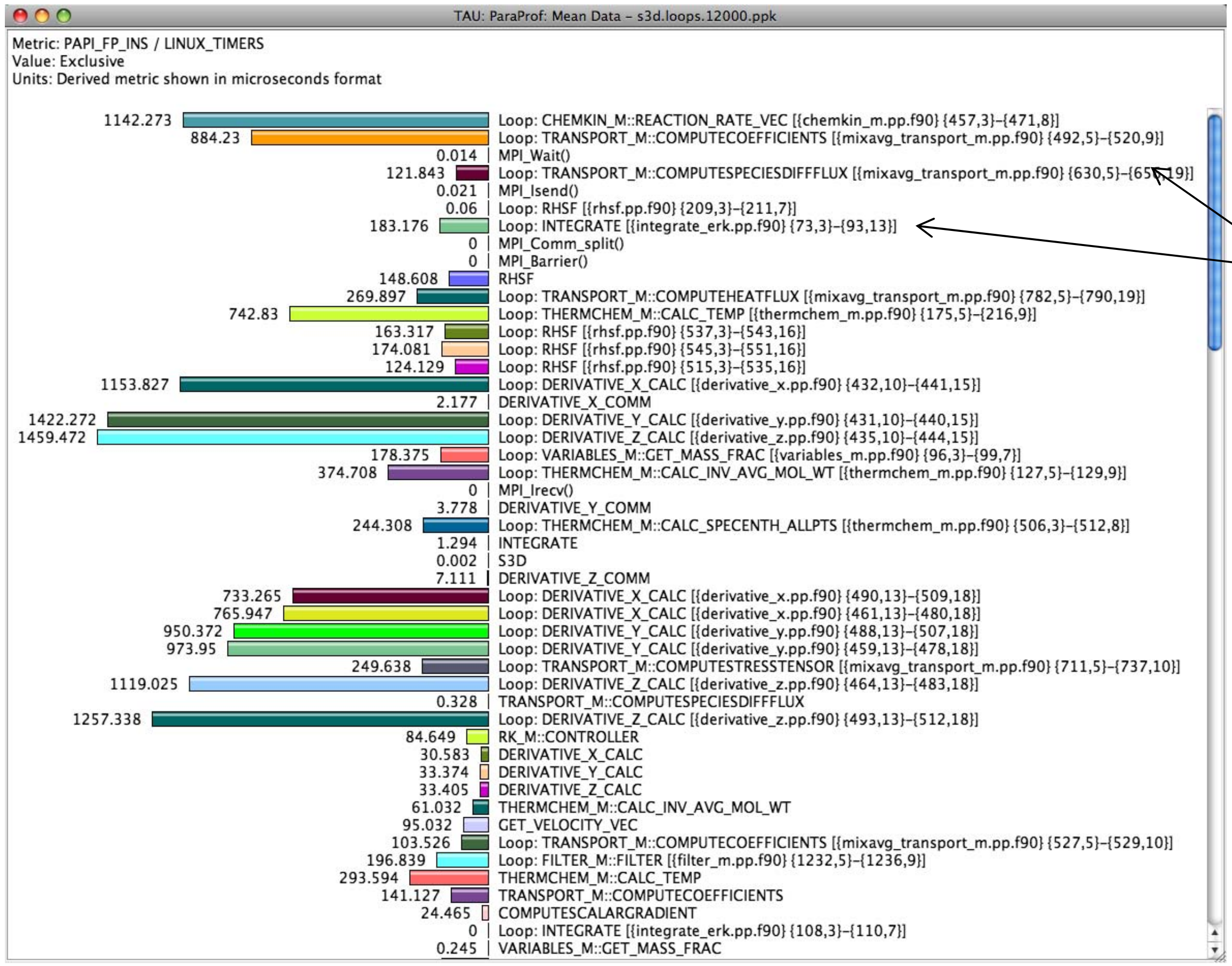
- Metrics can be time or hardware performance counters (cache misses, instructions)
- TAU can automatically instrument your source code using a package called PDT for routines, loops, I/O, memory, phases, etc.

Inclusive vs. Exclusive Measurements

- Performance with respect to code regions
- Exclusive measurements for region only
- Inclusive measurements includes child regions



Identifying Potential Bottlenecks



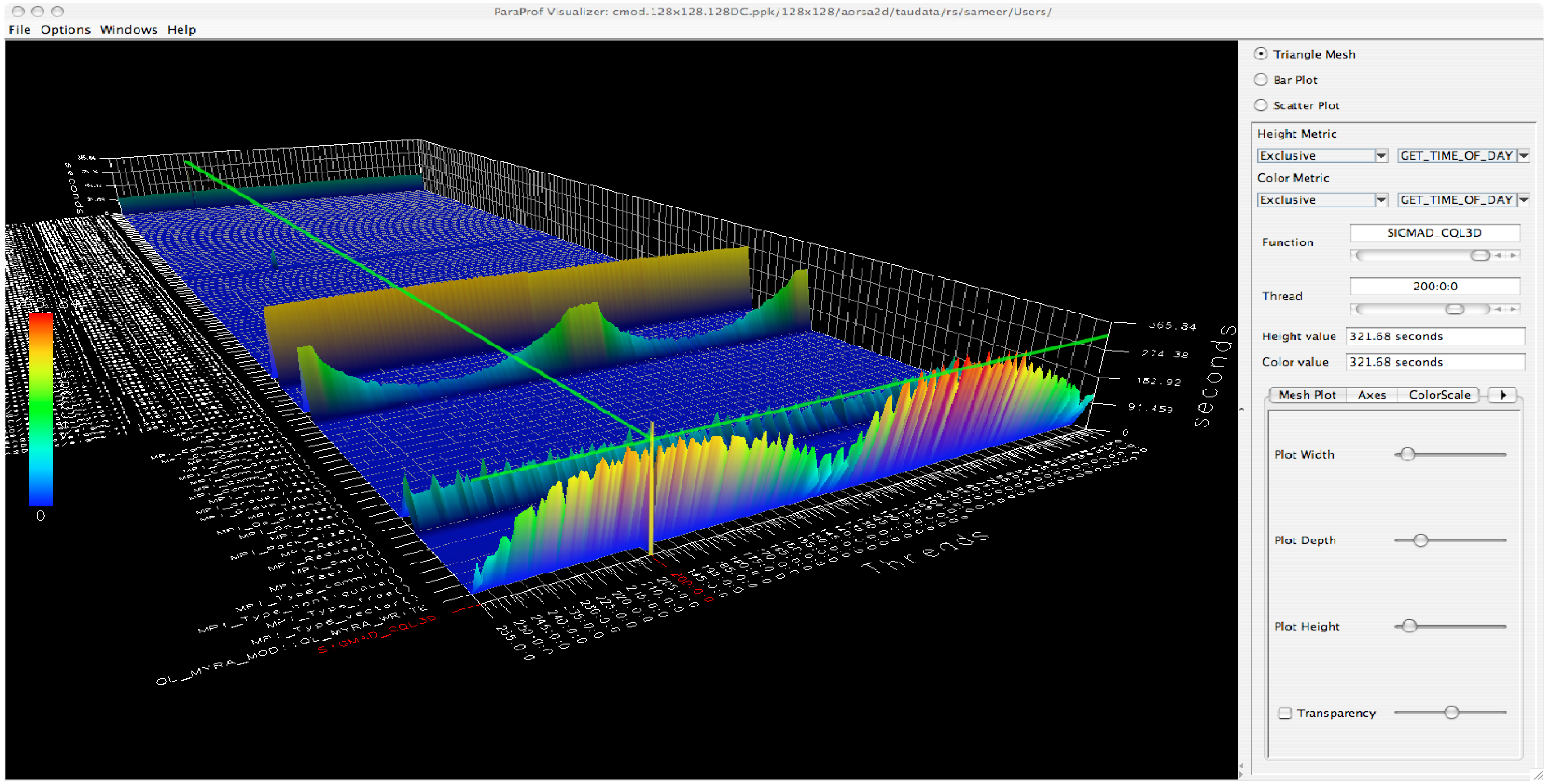
TAU: ParaProf: Mean Data – s3d.loops.12000.ppk

```
Metric: PAPI_FP_INS / LINUX_TIMERS
Value: Exclusive
Units: Derived metric shown in microseconds format

1142.273  Loop: CHEMKIN_M::REACTION_RATE_VEC [{chemkin_m.pp.f90} {457,3}-{471,8}]
  884.23  Loop: TRANSPORT_M::COMPUTECOEFFICIENTS [{mixavg_transport_m.pp.f90} {492,5}-{520,9}]
   0.014  MPI_Wait()
 121.843  Loop: TRANSPORT_M::COMPUTESPECIESDIFFFLUX [{mixavg_transport_m.pp.f90} {630,5}-{65[illegible]19}]
   0.021  MPI_Isend()
    0.06  Loop: RHSF [{rhsf.pp.f90} {209,3}-{211,7}]
 183.176  Loop: INTEGRATE [{integrate_erk.pp.f90} {73,3}-{93,13}]
       0  MPI_Comm_split()
       0  MPI_Barrier()
 148.608  RHSF
 269.897  Loop: TRANSPORT_M::COMPUTEHEATFLUX [{mixavg_transport_m.pp.f90} {782,5}-{790,19}]
  742.83  Loop: THERMCHEM_M::CALC_TEMP [{thermchem_m.pp.f90} {175,5}-{216,9}]
 163.317  Loop: RHSF [{rhsf.pp.f90} {537,3}-{543,16}]
 174.081  Loop: RHSF [{rhsf.pp.f90} {545,3}-{551,16}]
 124.129  Loop: RHSF [{rhsf.pp.f90} {515,3}-{535,16}]
1153.827  Loop: DERIVATIVE_X_CALC [{derivative_x.pp.f90} {432,10}-{441,15}]
   2.177  DERIVATIVE_X_COMM
1422.272  Loop: DERIVATIVE_Y_CALC [{derivative_y.pp.f90} {431,10}-{440,15}]
1459.472  Loop: DERIVATIVE_Z_CALC [{derivative_z.pp.f90} {435,10}-{444,15}]
 178.375  Loop: VARIABLES_M::GET_MASS_FRAC [{variables_m.pp.f90} {96,3}-{99,7}]
 374.708  Loop: THERMCHEM_M::CALC_INV_AVG_MOL_WT [{thermchem_m.pp.f90} {127,5}-{129,9}]
       0  MPI_Irecv()
   3.778  DERIVATIVE_Y_COMM
 244.308  Loop: THERMCHEM_M::CALC_SPECENTH_ALLPTS [{thermchem_m.pp.f90} {506,3}-{512,8}]
   1.294  INTEGRATE
   0.002  S3D
   7.111  DERIVATIVE_Z_COMM
 733.265  Loop: DERIVATIVE_X_CALC [{derivative_x.pp.f90} {490,13}-{509,18}]
 765.947  Loop: DERIVATIVE_X_CALC [{derivative_x.pp.f90} {461,13}-{480,18}]
 950.372  Loop: DERIVATIVE_Y_CALC [{derivative_y.pp.f90} {488,13}-{507,18}]
  973.95  Loop: DERIVATIVE_Y_CALC [{derivative_y.pp.f90} {459,13}-{478,18}]
 249.638  Loop: TRANSPORT_M::COMPUTESTRESSTENSOR [{mixavg_transport_m.pp.f90} {711,5}-{737,10}]
1119.025  Loop: DERIVATIVE_Z_CALC [{derivative_z.pp.f90} {464,13}-{483,18}]
   0.328  TRANSPORT_M::COMPUTESPECIESDIFFFLUX
1257.338  Loop: DERIVATIVE_Z_CALC [{derivative_z.pp.f90} {493,13}-{512,18}]
  84.649  RK_M::CONTROLLER
  30.583  DERIVATIVE_X_CALC
  33.374  DERIVATIVE_Y_CALC
  33.405  DERIVATIVE_Z_CALC
  61.032  THERMCHEM_M::CALC_INV_AVG_MOL_WT
  95.032  GET_VELOCITY_VEC
 103.526  Loop: TRANSPORT_M::COMPUTECOEFFICIENTS [{mixavg_transport_m.pp.f90} {527,5}-{529,10}]
 196.839  Loop: FILTER_M::FILTER [{filter_m.pp.f90} {1232,5}-{1236,9}]
 293.594  THERMCHEM_M::CALC_TEMP
 141.127  TRANSPORT_M::COMPUTECOEFFICIENTS
  24.465  COMPUTESCALARGRADIENT
       0  Loop: INTEGRATE [{integrate_erk.pp.f90} {108,3}-{110,7}]
   0.245  VARIABLES_M::GET_MASS_FRAC
```

low mflops in loops?

ParaProf 3D Profile Browser



ParaProf Visualizer: cmod.128x128.128DC.ppk/128x128/aorsa2d/taudata/rs/sameer/Users/
File Options Windows Help
Threads
seconds

(●) Triangle Mesh
( ) Bar Plot
( ) Scatter Plot

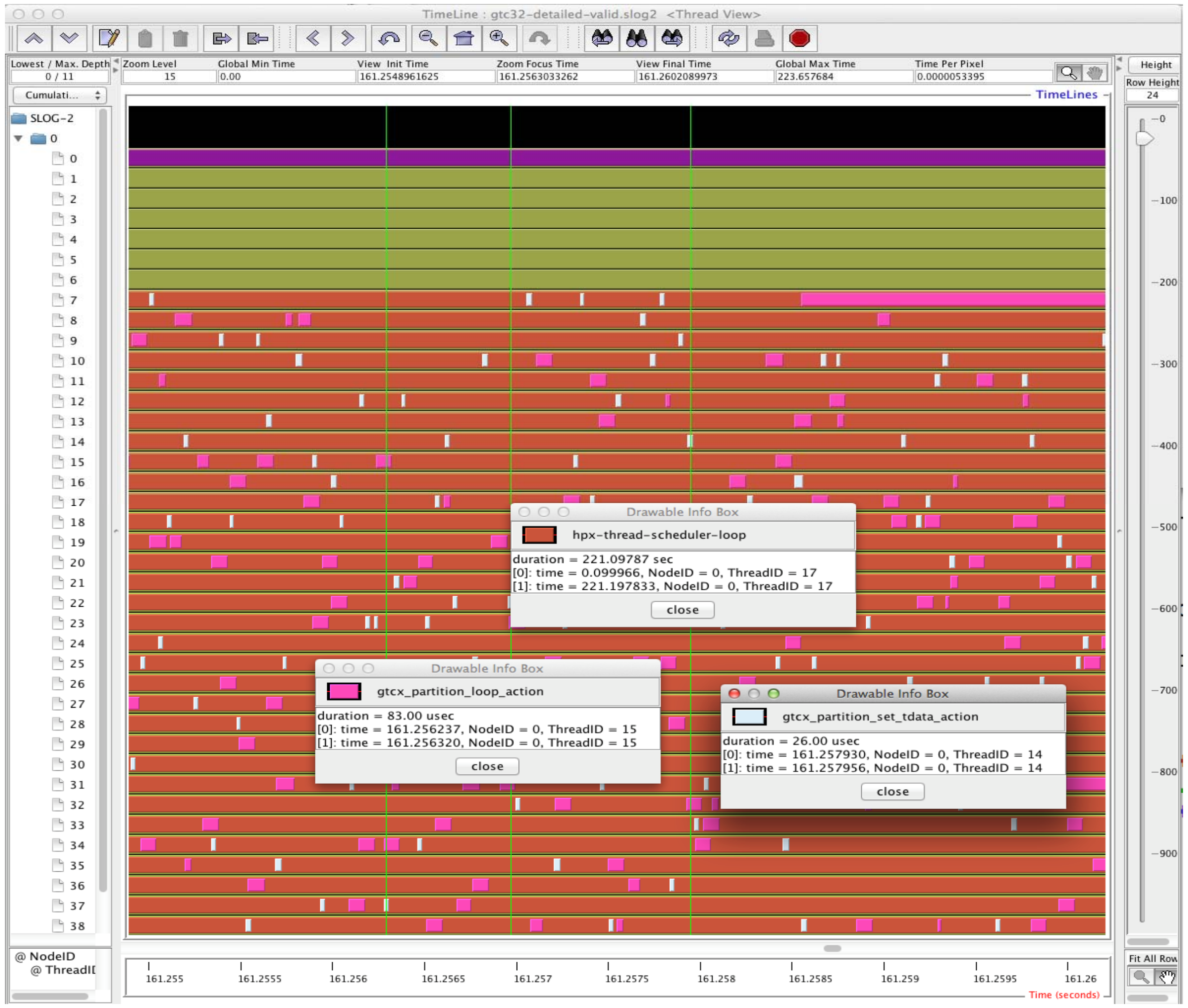
Height Metric: Exclusive, GET_TIME_OF_DAY
Color Metric: Exclusive, GET_TIME_OF_DAY

Function: SICMAD_CQL3D
Thread: 200:0:0
Height value: 321.58 seconds
Color value: 321.58 seconds

Mesh Plot | Axes | ColorScale

Plot Width
Plot Depth
Plot Height
☐ Transparency

Jumpshot Trace Visualizer in TAU



Lowest / Max. Depth	Zoom Level	Global Min Time	View Init Time	Zoom Focus Time	View Final Time	Global Max Time	Time Per Pixel	Row Height
0 / 11	15	0.00	161.2548961625	161.2563033262	161.2602089973	223.657684	0.0000053395	24

Drawable Info Box

hpx-thread-scheduler-loop

```
duration = 221.09787 sec
[0]: time = 0.099966, NodeID = 0, ThreadID = 17
[1]: time = 221.197833, NodeID = 0, ThreadID = 17
```

Drawable Info Box

gtcx_partition_loop_action

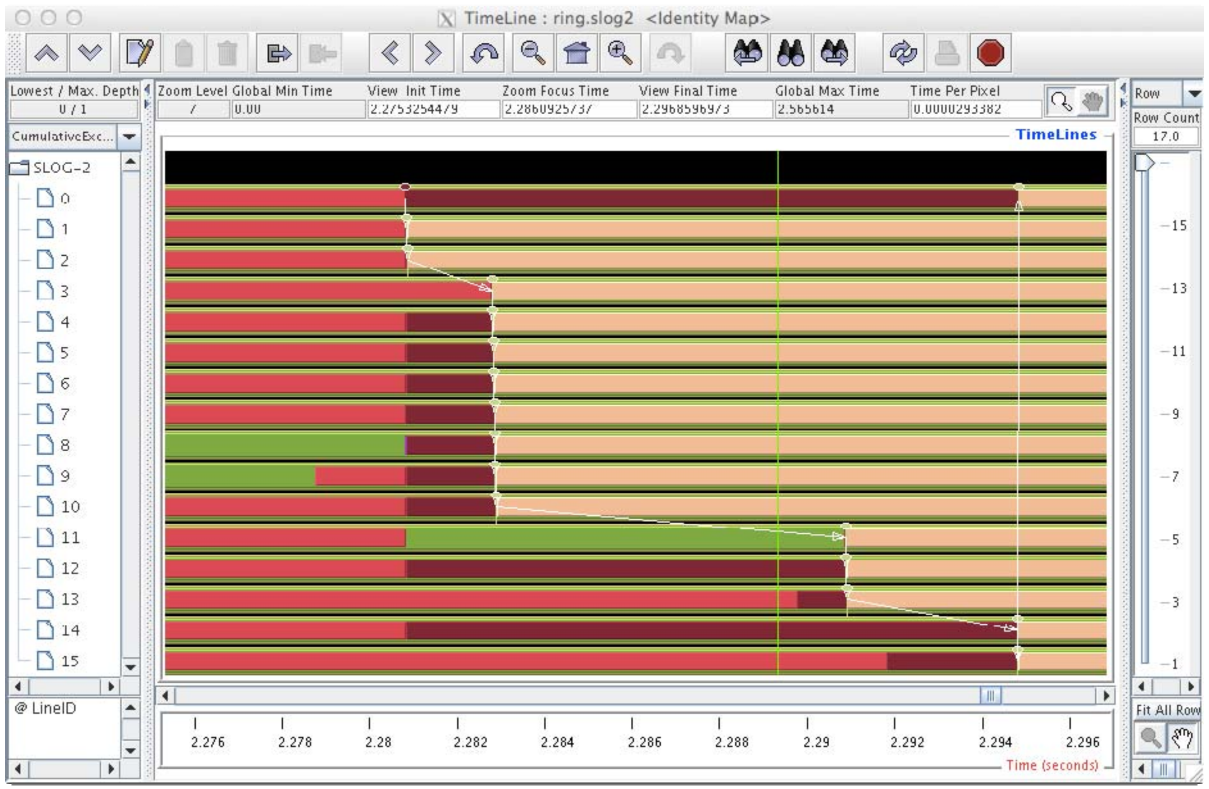
```
duration = 83.00 usec
[0]: time = 161.256237, NodeID = 0, ThreadID = 15
[1]: time = 161.256320, NodeID = 0, ThreadID = 15
```

Drawable Info Box

gtcx_partition_set_tdata_action

```
duration = 26.00 usec
[0]: time = 161.257930, NodeID = 0, ThreadID = 14
[1]: time = 161.257956, NodeID = 0, ThreadID = 14
```

Tracing Communication in Jumpshot


TimeLine : ring.slog2 <Identity Map>

Lowest / Max. Depth	Zoom Level	Global Min Time	View Init Time	Zoom Focus Time	View Final Time	Global Max Time	Time Per Pixel
0 / 1	7	0.00	2.2753254479	2.2860925737	2.2968596973	2.565614	0.0000293382

Row Count 17.0

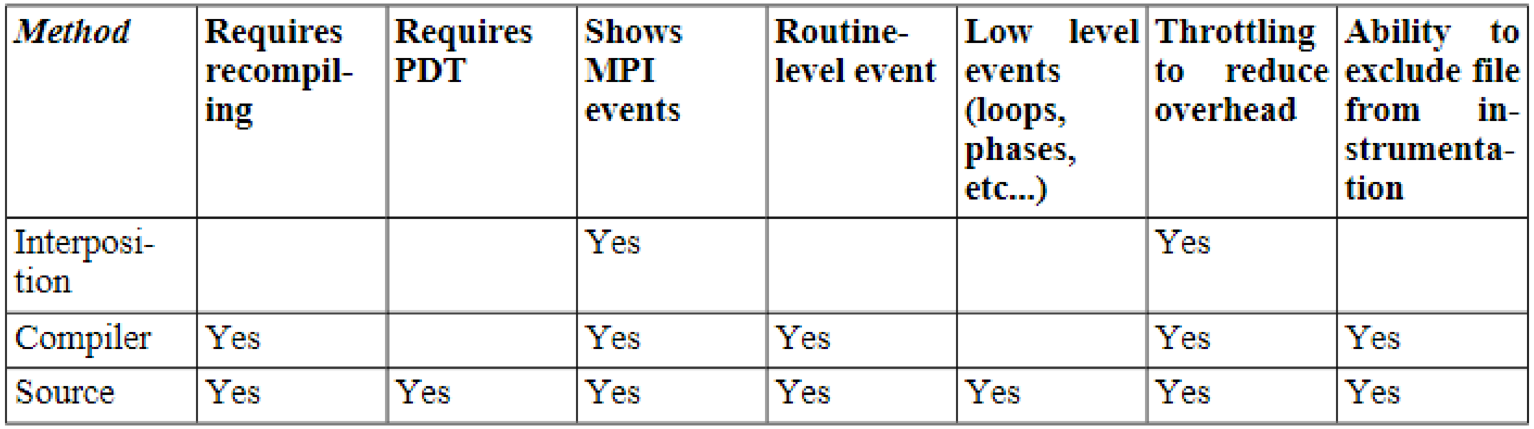
```
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt
% cmake –DCMAKE_CXX_COMPILER=tau_cxx.sh; make –j 8
% export TAU_TRACE=1
% mpirun –np 16 ./a.out ; tau_treemerge.pl; tau2slog2 tau.trc tau.edf –o a.slog2
% jumpshot a.slog2 &
```

Content

- Overview
- **Instrumentation**
- Profiling
- Tracing
- Analyzing
- Application Scenario

Types of Instrumentation

- TAU provides three methods：
  - <1>Library interposition using tau_exec.
  - <2>Compiler directives.
  - <2>Source transformation using PDT.



Method	**Requires recompil-ing**	**Requires PDT**	**Shows MPI events**	**Routine-level event**	**Low level events (loops, phases, etc...)**	**Throttling to reduce overhead**	**Ability to exclude file from in-strumenta-tion**
Interposi-tion			Yes			Yes	
Compiler	Yes		Yes	Yes		Yes	Yes
Source	Yes	Yes	Yes	Yes	Yes	Yes	Yes

<1> Using tau_exec

- Dynamic instrumentation :
  - Achieved through library pre-loading.
  - Can be used on both uninstrumented and instrumented binaries.

- Place tau_exec before the executable:
  - %> tau_exec -io ./a.out
  - %> mpirun -np 4 tau_exec -io ./a.out

TAU scripted compilation

- Tau_exec :
  - Simple but less detailed.

- For more detailed profiles, TAU provides two means to **compile** your application with TAU:
  - Compiler Based Instrumentation
    - through your compiler.
  - Source Based Instrumentation
    - through source transformation using PDT.

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
```

Compiler Based Instrumentation

- On machines where a TAU module is not available, you will need to set the tau makefile and/or options:

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

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

Compiler Based Instrumentation

- 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.

```
%>export TAU_MAKEFILE=[path to tau]/[arch]/lib/[makefile]
%>export TAU_OPTIONS=-optCompInst

%>tau_cc.sh sampleCprogram.c
```

Source 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  samplecprogram.c
```

- NOTE:
  - **When setting the TAU_MAKEFILE make sure the Makefile name contains pdt.**

Options to TAU compiler scripts

- There 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.

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

Content

Profiling

- 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.

Running the Application

- After instrumentation and compilation are completed, the profiled application is run to generate the profile data file.
  - By default, profiles are placed in the current directory or set PROFILEDIR.
  - set the TAU_VERBOSE to see the steps the TAU takes when your application is running.

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

Reducing Overhead

- TAU automatically throttles short running functions in an effort to reduce the amount of overhead.
  - This feature may be turned off by setting the environment variable TAU_THROTTLE =0 .
- The default rules TAU uses to determine which functions to throttle is:
  - numcalls > 100000 && usecs/call < 10

```
% setenv TAU_THROTTLE_NUMCALLS 2000000
% setenv TAU_THROTTLE_PERCALL 5
```

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 environment variable (default is two).

```
TAU_CALLPATH_DEPTH
```

- Because instrumentation overhead will increase with the depth of the callpath, you should use the shortest call path that is sufficient.

Using Hardware Counters

- Performance counters exist on many modern microprocessors.
  - cache misses, floating point operations, etc.

- The **Performance Data Standard and API (PAPI)** package provides a uniform interface to access these performance counters.[http://icl.cs.utk.edu/papi/].
- To use these counters, you must:
  - First, find out which PAPI events your system supports.
  - Next, test the compatibility between each metric you wish papi to profile.
  - Next, make sure that you are using a makefile with papi in its name.

Content

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.

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.
- Set the output directory with a environment variable:

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

Content

Analyzing -- Text summary

- For a quick view summary of TAU performance, use

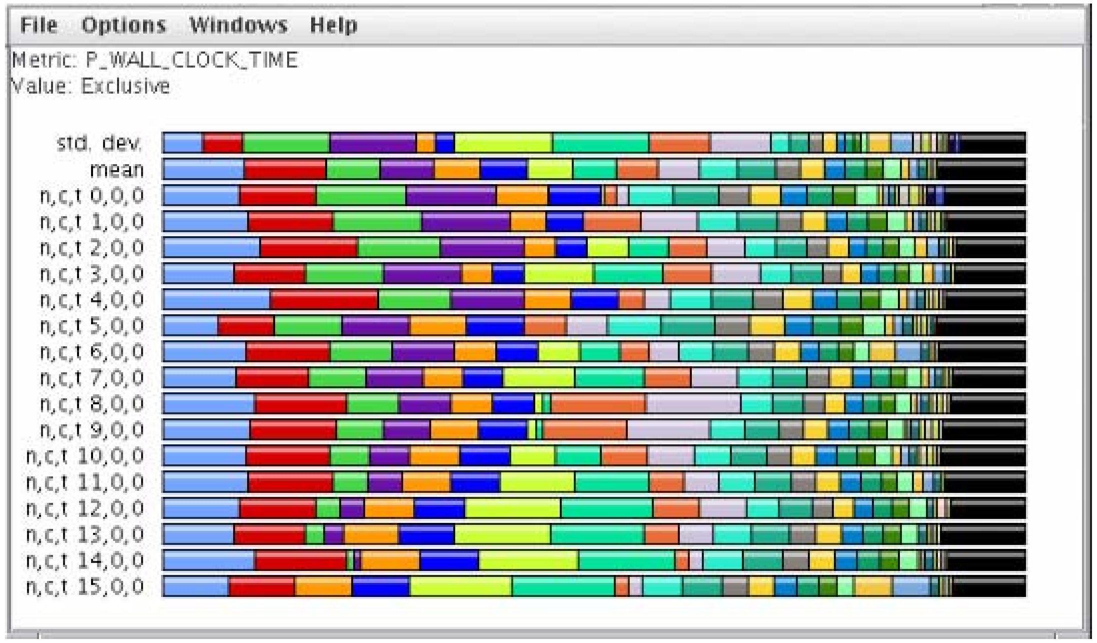
```
%> cd MULTI__P_WALL_CLOCK_TIME
%> pprof
```

- It reads and prints a summary of the TAU data in the current directory.

```
---------------------------------------------------------------------------------------
%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-regul
```

Analyzing -- ParaProf

- To launch ParaProf, execute paraprof from the command line where the profiles are located.

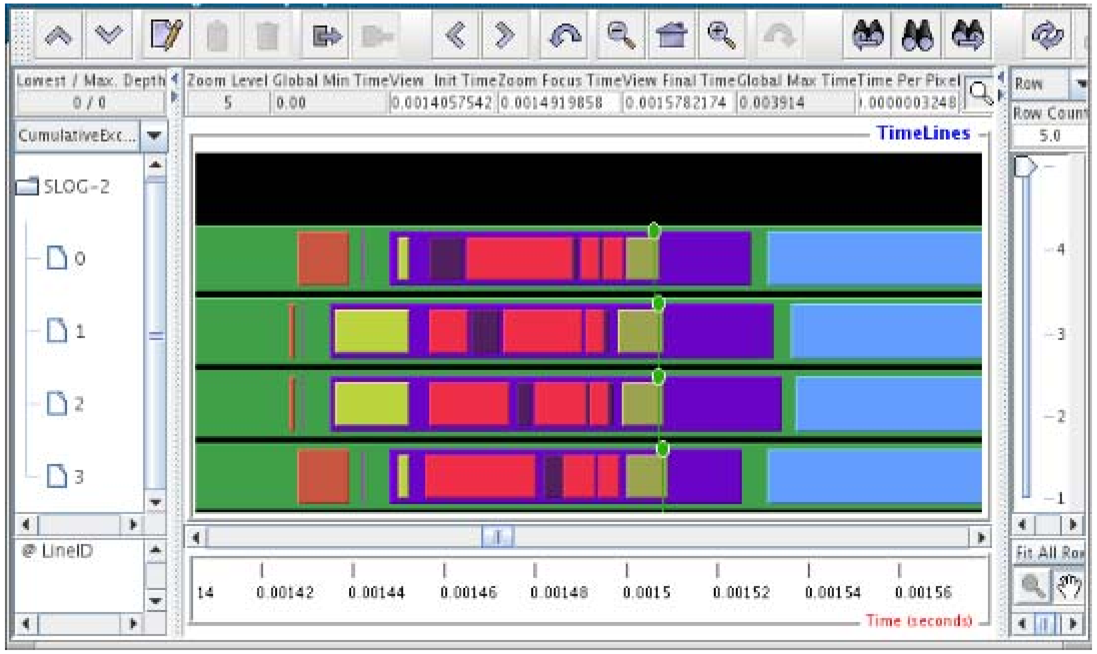


Metric: P_WALL_CLOCK_TIME
Value: Exclusive

Analyzing -- Jumpshot

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

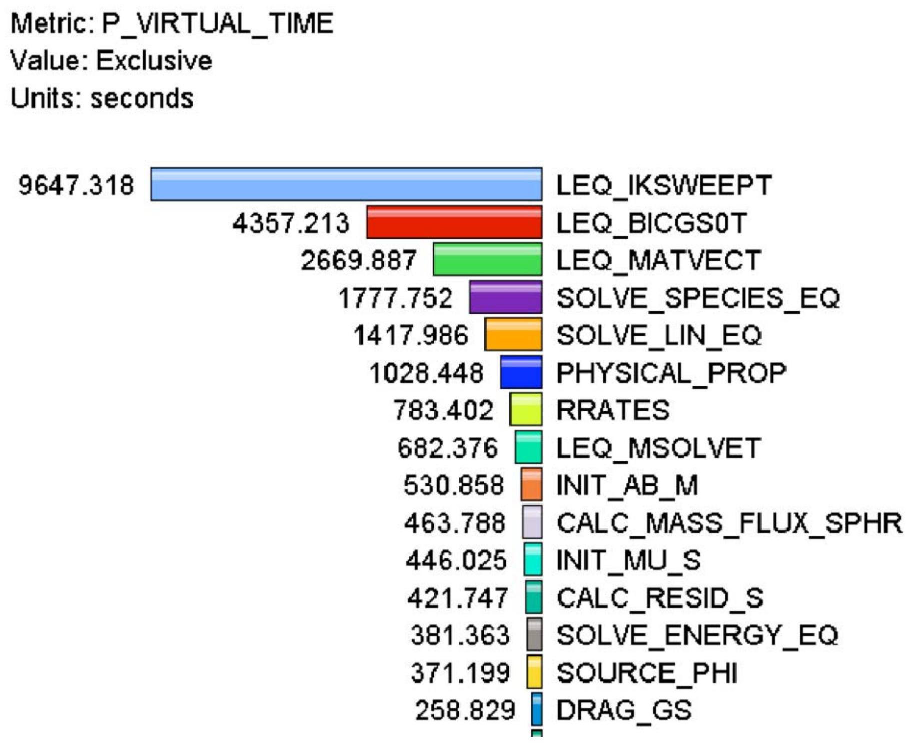
```
% tau_treemerge.pl
% tau2slog2 tau.trc tau.edf -o tau.slog2
% jumpshot tau.slog2
```



Content

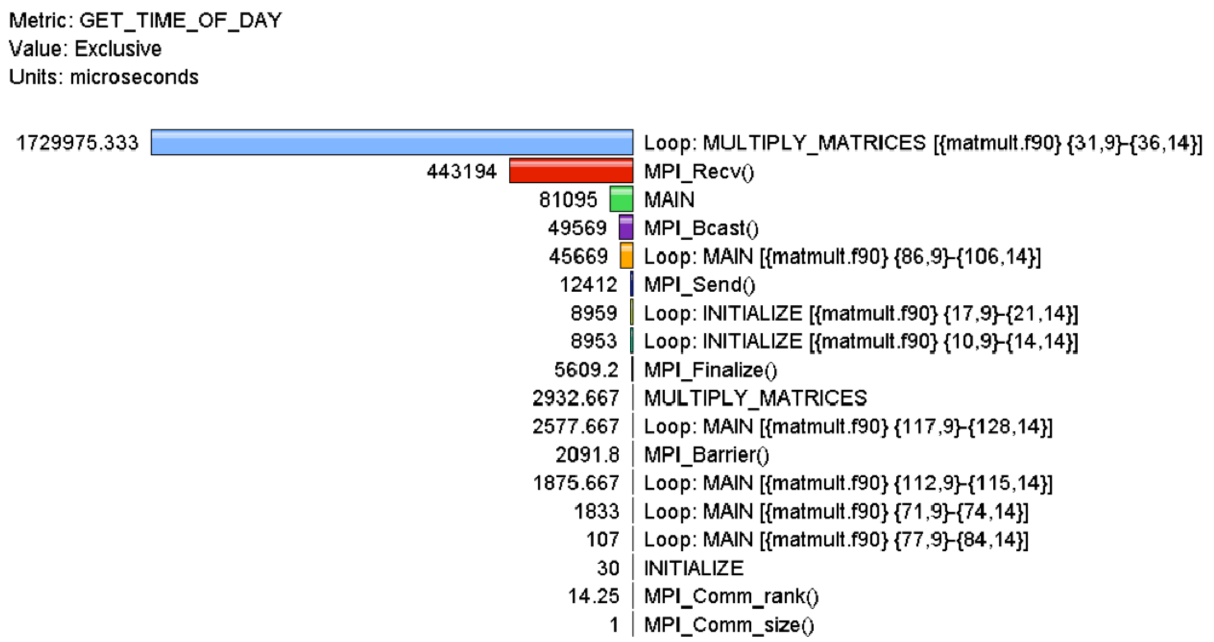
Application Scenario (1)

- Q. What routines account for the most time? How much?
- A. Create a flat profile with wallclock time.



Application Scenario (2)

- Q. What loops account for the most time? How much?
- A. Create a flat profile with loop instrumentation.

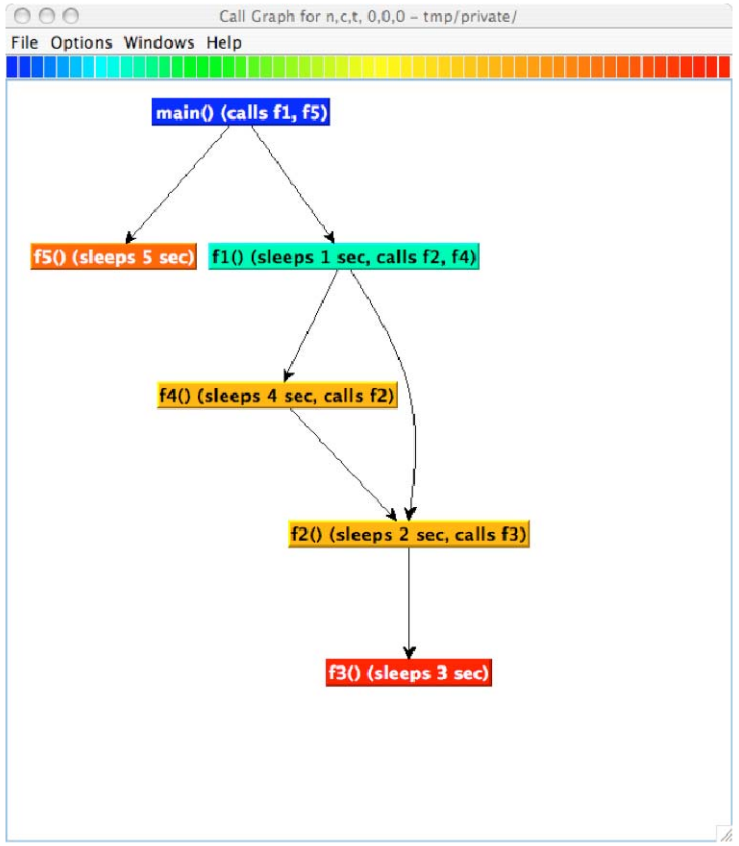


Metric: GET_TIME_OF_DAY
Value: Exclusive
Units: microseconds

1729975.333	Loop: MULTIPLY_MATRICES [{matmult.f90} {31,9}-{36,14}]
443194	MPI_Recv()
81095	MAIN
49569	MPI_Bcast()
45669	Loop: MAIN [{matmult.f90} {86,9}-{106,14}]
12412	MPI_Send()
8959	Loop: INITIALIZE [{matmult.f90} {17,9}-{21,14}]
8953	Loop: INITIALIZE [{matmult.f90} {10,9}-{14,14}]
5609.2	MPI_Finalize()
2932.667	MULTIPLY_MATRICES
2577.667	Loop: MAIN [{matmult.f90} {117,9}-{128,14}]
2091.8	MPI_Barrier()
1875.667	Loop: MAIN [{matmult.f90} {112,9}-{115,14}]
1833	Loop: MAIN [{matmult.f90} {71,9}-{74,14}]
107	Loop: MAIN [{matmult.f90} {77,9}-{84,14}]
30	INITIALIZE
14.25	MPI_Comm_rank()
1	MPI_Comm_size()

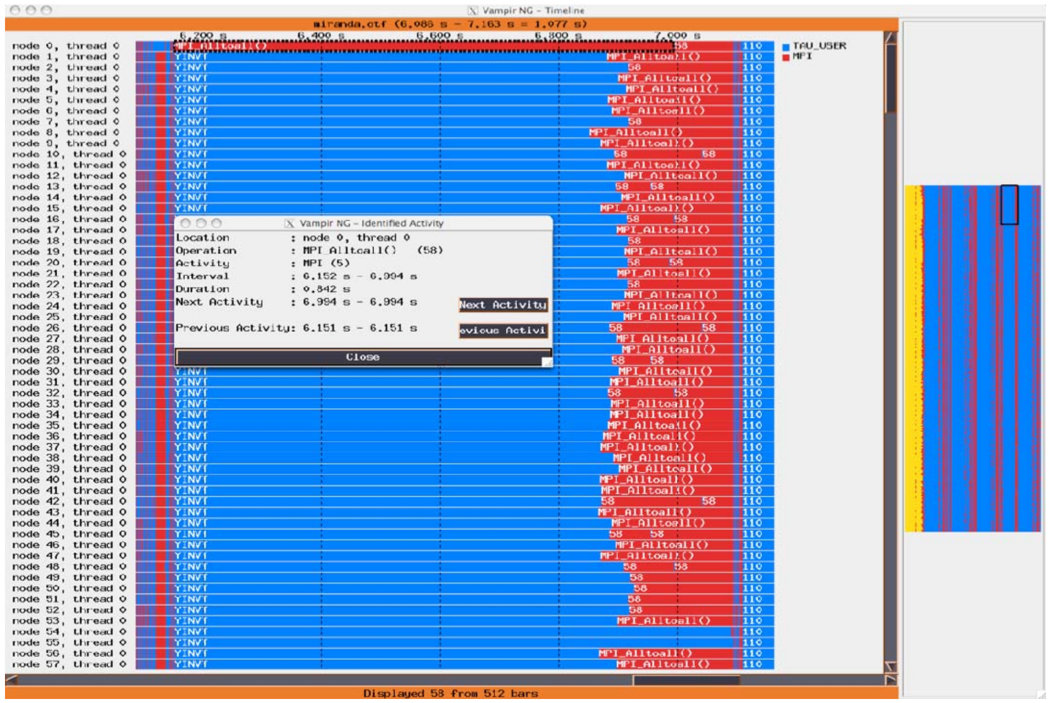
Application Scenario (3)

- Q. Who calls MPI_Barrier() Where?
- A. Create a callpath profile with given depth.


Call Graph for n,c,t, 0,0,0 – tmp/private/
File Options Windows Help
main() (calls f1, f5)
f5() (sleeps 5 sec)
f1() (sleeps 1 sec, calls f2, f4)
f4() (sleeps 4 sec, calls f2)
f2() (sleeps 2 sec, calls f3)
f3() (sleeps 3 sec)

Application Scenario (4)

- Q. What happens in my code at a given time ?
- A. Create an event tree.



```
Location          : node 0, thread 0
Operation         : MPI_Alltoall()   (58)
Activity          : MPI (5)
Interval          : 6.152 s - 6.994 s
Duration          : 0.842 s
Next Activity     : 6.994 s - 6.994 s

Previous Activity: 6.151 s - 6.151 s
```

Application Scenarios

- Q. How do I instrument Python Code?
- A. Create an python wrapper library.

- Q. How does my application scale?
- A. Examine profiles in PerfExplorer.

- Q. What MFlops am I getting in all loops?
- A. Create a flat profile with PAPI_FP_INS/OPS and time with loop instrumentation.

Using TAU: Simplest Case

- Uninstrumented code:
  - % mpirun –np 8 ./a.out
- With TAU:
  - % mpirun –np 8 tau_exec ./a.out
  - % paraprof

references

- **TAU User Guide**
  - https://www.cs.uoregon.edu/research/tau/docs/newguide/bk01pt01.html
- TAU Website:
  - http://tau.uoregon.edu
- Sameer Shende, *TAU Performance System,* Performance Research Lab, University of Oregon.
  - http://tau.uoregon.edu/tau.ppt