Improving Application Performance Using the TAU Performance System

Sameer Shende, John C. Linford
{sameer, jlinford}@paratools.com

ParaTools, Inc and University of Oregon.

April 4-5, 2013, CG1, NCAR, UCAR

Download slides from:

http://www.paratools.com/sea13

Tutorial Goals

This tutorial is an introduction to portable performance evaluation tools. You should leave here with a better understanding of…

- Concepts and steps involved in performance evaluation
- Understanding key concepts in understanding code performance
- How to collect and analyze data from hardware performance counters (PAPI)
- How to instrument your programs with TAU
- Measurement options provided by TAU
- Environment variables used for choosing metrics, generating performance data
- How to use ParaProf, TAU's profile browser
- General familiarity with TAU use for Fortran, C++, C, and mixed language
- How to generate trace data in different formats

Outline (1)

- **Introduction to the TAU Performance System**
- **Configure your environment, selecting TAU_MAKEFILE**
- **Generating a flat MPI profile**
  - Automatic instrumentation with TAU compiler wrappers
  - Visualization with ParaProf
- **Generating a loop-level profile**
- **Profiling with multiple counters**
  - Computing FLOPS in each loop
- **Generate and visualize a callpath profile**
- **Generate and visualize the communication matrix**
- **Compiler-based Instrumentation (when PDT isn't enough)**

Outline (2)

- **Binary rewriting instrumentation**
- **Generating event traces**
  - Visualization with Vampir
- **Runtime preloading with tau_exec**
- **Techniques for wrapping external libraries**
  - tau_gen_wrapper
  - HDF5 library wrapping
  - Using the TAU I/O and memory wrappers

Outline (3)

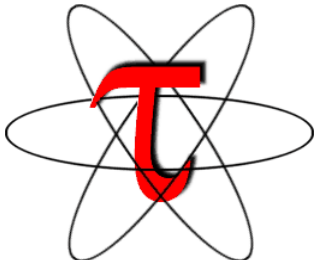
- **Profiling Python applications**
- **Profiling GPU-accelerated applications**
  - GPU performance measuring tools
- **Performance analysis**
  - Snapshots in ParaProf
- **PerfExplorer**
  - Runtime breakdown
  - Scalability
  - Correlation analysis
  - Performance regression testing

Outline (4)

- **Multi-language debugging**
  - Unwind the callstack across Python, C++, and Fortran
- **Memory debugging**
  - Off-by-one errors
  - Zero-byte malloc
  - Incorrect allocation alignment
- **Acknowledgements**
- **Reference**

TAU Performance System®

http://tau.uoregon.edu/



- **Tuning and Analysis Utilities (18+ year project)**
- **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**

Understanding Application Performance using TAU

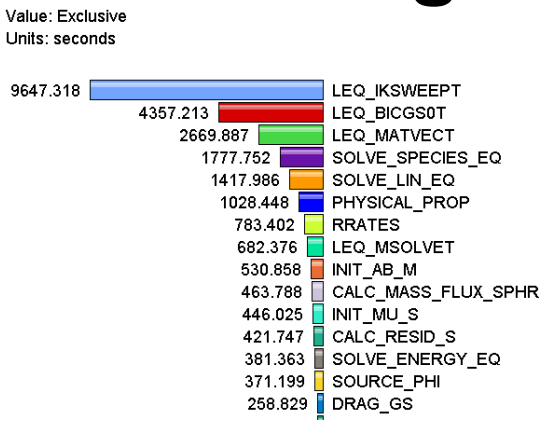
- **How much time** is spent in each application routine and outer *loops*? Within loops, what is the contribution of each *statement*?
- **How many instructions** are executed in these code regions? Floating point, Level 1 and 2 *data cache misses*, hits, branches taken?
- **What is the memory usage**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

What Can TAU Do?

- **Profiling and tracing**
  - Profiling shows you how much (total) time was spent in each routine
  - Tracing shows you *when* the events take place on a timeline
- **Multi-language debugging**
  - Identify the source location of a crash by unwinding the system callstack
  - Identify memory errors (off-by-one, etc.)
- Profiling and tracing can measure **time** as well as **hardware performance counters** (cache misses, instructions) from your CPU
- TAU can **automatically instrument** your source code using a package called PDT for routines, loops, I/O, memory, phases, etc.
- TAU runs on **all HPC platforms** and it is free (BSD style license)
- TAU includes instrumentation, measurement and analysis tools

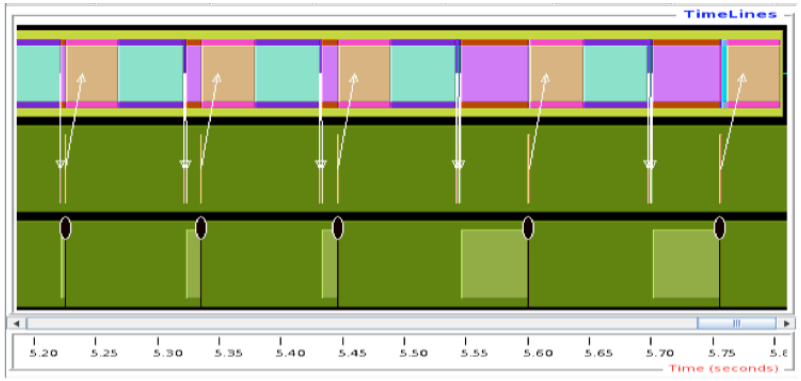
Profiling and Tracing

Profiling



- **Profiling** shows you how much (total) time was spent in each routine

Tracing

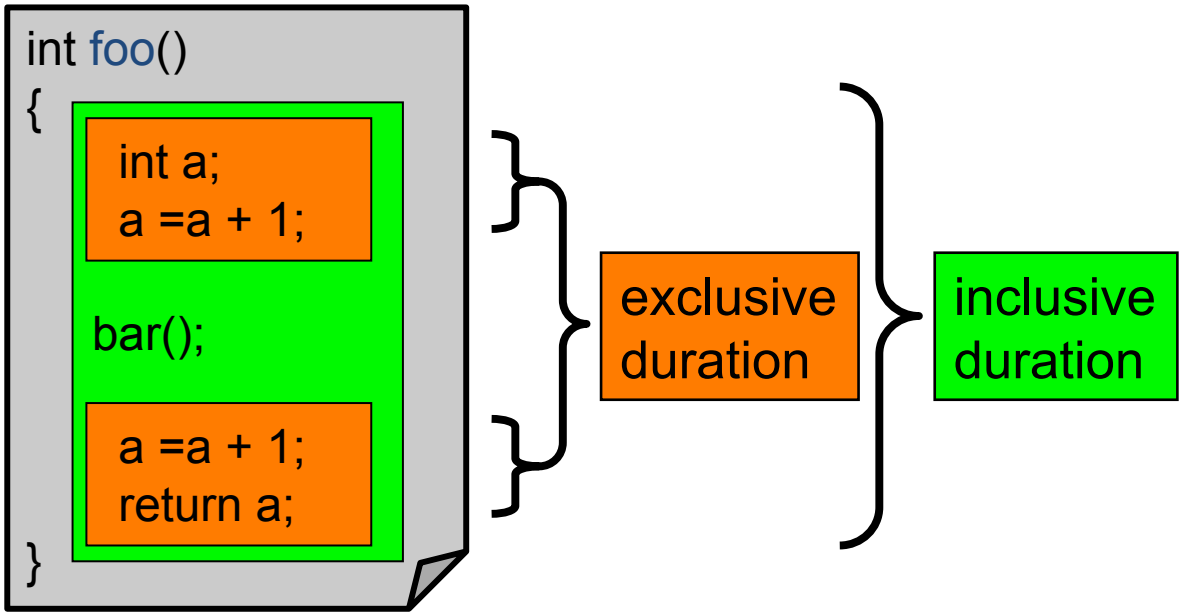


- **Tracing** shows you *when* the events take place on a timeline

- 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



What does TAU support?

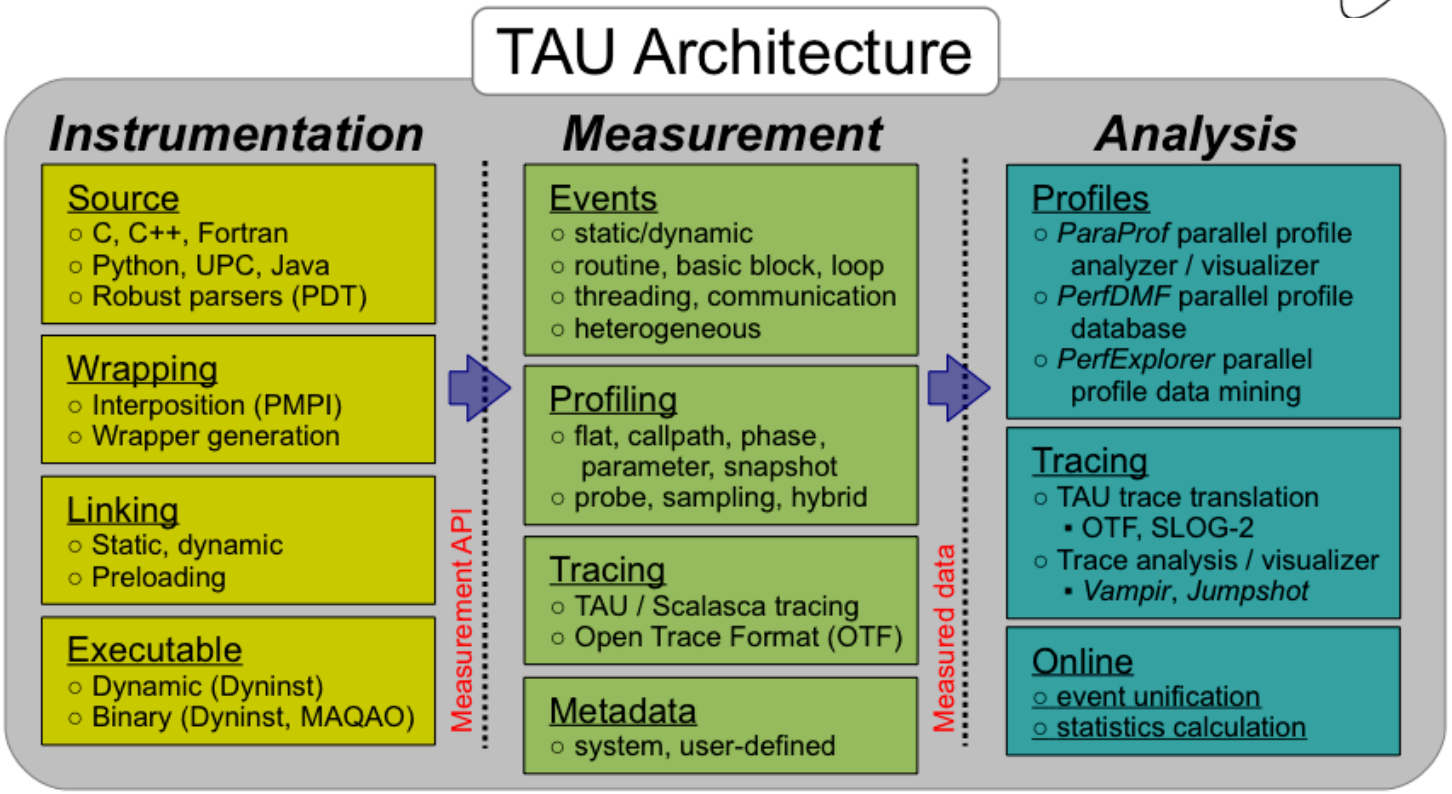
C/C++
Fortran
pthreads
Intel
GNU
MinGW
CUDA
UPC
OpenACC
Intel MIC
LLVM
PGI
Linux
BlueGene
NVIDIA Kepler
OpenCL
GPI
Python
Java
MPI
OpenMP
Cray
Sun
Windows
AIX
Fujitsu
ARM
OS X

Insert yours here

Availability on New Systems

- Intel compilers with Intel MPI on Intel Xeon Phi™ (MIC)
- GPI with Intel Linux x86_64 Infiniband clusters
- IBM BG/Q and Power 7 Linux with IBM XL UPC compilers
- NVIDIA Kepler K20 with CUDA 5.0 with NVCC
- Fujitsu Fortran/C/C++ MPI compilers on the K computer
- PGI compilers with OpenACC support on NVIDIA systems
- Cray CX30 Sandybridge Linux systems with Intel compilers
- Cray CCE compilers with OpenACC support on Cray XK7
- AMD OpenCL libs with GNU on AMD Fusion cluster systems
- MPC compilers on TGCC Curie system (Bull, Linux x86_64)
- GNU compilers on ARM Linux clusters (MontBlanc, BSC)
- Cray CCE compilers with OpenACC on Cray XK6 (K20)
- Microsoft MPI with Mingw compilers under Windows Azure
- LLVM and GNU compilers under Mac OS X, IBM BGQ

The TAU Architecture



TAU Architecture and Workflow

Instrumentation: **Add 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

Measurement: **Profiling or tracing using various metrics**

- 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)

Analysis: **Visualization of profiles and traces**

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

Instrumentation

Direct and indirect performance observation

- Instrumentation invokes performance measurement
- Direct measurement with *probes*
- Indirect measurement with periodic sampling or hardware performance counter overflow interrupts
- Events measure performance data, metadata, context, etc.

User-defined events

- ***Interval*** (start/stop) events to measure exclusive & inclusive duration
- ***Atomic events*** take measurements at a single point
  - Measures total, samples, min/max/mean/std. deviation statistics
- ***Context events*** are atomic events with executing context
  - Measures above statistics for a given calling path

Direct Observation Events

Interval events (begin/end events)

- Measures exclusive & inclusive durations between events
- Metrics monotonically increase
- Example: Wall-clock timer

Atomic events (trigger with data value)

- Used to capture performance data state
- Shows extent of variation of triggered values (min/max/mean)
- Example: heap memory consumed at a particular point

Code events

- Routines, classes, templates
- Statement-level blocks, loops
- Example: for-loop begin/end

Interval and Atomic Events in TAU

xterm

```
NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0        0.187        1,105           1          44     1105659 int main(int, char **) C
 93.2        1,030        1,030           1           0     1030654 MPI_Init()
  5.9        0.879           65          40         320        1637 void func(int, int) C
  4.6           51           51          40           0        1277 MPI_Barrier()
  1.2           13           13         120           0         111 MPI_Recv()
  0.8            9            9           1           0        9328 MPI_Finalize()
  0.0        0.137        0.137         120           0           1 MPI_Send()
  0.0        0.086        0.086          40           0           2 MPI_Bcast()
  0.0        0.002        0.002           1           0           2 MPI_Comm_size()
  0.0        0.001        0.001           1           0           1 MPI_Comm_rank()
---------------------------------------------------------------------------------------

USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0
---------------------------------------------------------------------------------------
NumSamples   MaxValue   MinValue  MeanValue  Std. Dev.  Event Name
---------------------------------------------------------------------------------------
       365  5.138E+04      44.39   3.09E+04  1.234E+04  Heap Memory Used (KB) : Entry
       365  5.138E+04       2064  3.115E+04   1.21E+04  Heap Memory Used (KB) : Exit
        40         40         40         40          0  Message size for broadcast
---------------------------------------------------------------------------------------
                                                                          27,1          1%
```

Interval events show ***duration***

Atomic events (triggered with value) show ***extent of variation*** (min/max/mean)

```
% export TAU_CALLPATH_DEPTH=0
% export TAU_TRACK_HEAP=1
```

Atomic Events and Context Events

xterm

```
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0        0.253        1,106           1          44    1106701 int main(int, char **) C
 93.2        1,031        1,031           1           0    1031311 MPI_Init()
  6.0            1           66          40         320       1650 void func(int, int) C
  5.7           63           63          40           0       1588 MPI_Barrier()
  0.8            9            9           1           0       9119 MPI_Finalize()
  0.1            1            1         120           0         10 MPI_Recv()
  0.0        0.141        0.141         120           0          1 MPI_Send()
  0.0        0.085        0.085          40           0          2 MPI_Bcast()
  0.0        0.001        0.001           1           0          1 MPI_Comm_size()
  0.0            0            0           1           0          0 MPI_Comm_rank()
---------------------------------------------------------------------------------------

USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0
---------------------------------------------------------------------------------------
NumSamples   MaxValue   MinValue  MeanValue  Std. Dev.  Event Name
---------------------------------------------------------------------------------------
        40         40         40         40          0  Message size for broadcast
       365  5.139E+04      44.39  3.091E+04  1.234E+04  Heap Memory Used (KB) : Entry
        40  5.139E+04       3097  3.114E+04  1.227E+04  Heap Memory Used (KB) : Entry : MPI_Barrier()
        40  5.139E+04   1.13E+04  3.134E+04  1.187E+04  Heap Memory Used (KB) : Entry : MPI_Bcast()
         1       2067       2067       2067          0  Heap Memory Used (KB) : Entry : MPI_Comm_rank()
         1       2066       2066       2066          0  Heap Memory Used (KB) : Entry : MPI_Comm_size()
         1  5.139E+04  5.139E+04  5.139E+04  0.0006905  Heap Memory Used (KB) : Entry : MPI_Finalize()
         1      57.56      57.56      57.56          0  Heap Memory Used (KB) : Entry : MPI_Init()
       120  5.139E+04   1.13E+04  3.134E+04  1.187E+04  Heap Memory Used (KB) : Entry : MPI_Recv()
       120  5.139E+04  1.129E+04  3.134E+04  1.187E+04  Heap Memory Used (KB) : Entry : MPI_Send()
         1      44.39      44.39      44.39          0  Heap Memory Used (KB) : Entry : int main(int, char **) C
        40  5.036E+04       2068  3.011E+04  1.227E+04  Heap Memory Used (KB) : Entry : void func(int, int) C
                                                                                  4.9            1%
```

Atomic events

Context events are atomic events with executing context

```
% export TAU_CALLPATH_DEPTH=1
% export TAU_TRACK_HEAP=1
```

Controls depth of executing context shown in profiles

Context Events with Callpath

xterm

```
NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0         0.357        1,114           1          44     1114040 int main(int, char **) C
 92.6         1,031        1,031           1           0     1031066 MPI_Init()
  6.7            72           74          40         320        1865 void func(int, int) C
  0.7             8            8           1           0        8002 MPI_Finalize()
  0.1             1            1         120           0          12 MPI_Recv()
  0.1         0.608        0.608          40           0          15 MPI_Barrier()
  0.0         0.136        0.136         120           0           1 MPI_Send()
  0.0         0.095        0.095          40           0           2 MPI_Bcast()
  0.0         0.001        0.001           1           0           1 MPI_Comm_size()
  0.0             0            0           1           0           0 MPI_Comm_rank()
---------------------------------------------------------------------------------------

USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0
---------------------------------------------------------------------------------------
NumSamples   MaxValue   MinValue  MeanValue  Std. Dev.  Event Name
---------------------------------------------------------------------------------------
       365  5.139E+04      44.39  3.091E+04  1.234E+04  Heap Memory Used (KB) : Entry
         1      44.39      44.39      44.39          0  Heap Memory Used (KB) : Entry : int main(int, char **) C
         1       2068       2068       2068          0  Heap Memory Used (KB) : Entry : int main(int, char **) C => MPI_Comm_rank()
         1       2066       2066       2066          0  Heap Memory Used (KB) : Entry : int main(int, char **) C => MPI_Comm_size()
         1  5.139E+04  5.139E+04  5.139E+04          0  Heap Memory Used (KB) : Entry : int main(int, char **) C => MPI_Finalize()
         1      57.58      57.58      57.58          0  Heap Memory Used (KB) : Entry : int main(int, char **) C => MPI_Init()
        40  5.036E+04       2069  3.011E+04  1.228E+04  Heap Memory Used (KB) : Entry : int main(int, char **) C => void func(int, int) C
        40  5.139E+04       3098  3.114E+04  1.227E+04  Heap Memory Used (KB) : Entry : void func(int, int) C => MPI_Barrier()
        40  5.139E+04   1.13E+04  3.134E+04  1.187E+04  Heap Memory Used (KB) : Entry : void func(int, int) C => MPI_Bcast()
       120  5.139E+04   1.13E+04  3.134E+04  1.187E+04  Heap Memory Used (KB) : Entry : void func(int, int) C => MPI_Recv()
       120  5.139E+04   1.13E+04  3.134E+04  1.187E+04  Heap Memory Used (KB) : Entry : void func(int, int) C => MPI_Send()
       365  5.139E+04       2065  3.116E+04   1.21E+04  Heap Memory Used (KB) : Exit
                                                                                3,7           1%
```

```
% export TAU_CALLPATH_DEPTH=2
% export TAU_TRACK_HEAP=1
```

Callpath shown on context events

Direct Instrumentation Options in TAU

Source Code Instrumentation

- Automatic instrumentation using pre-processor based on static analysis of source code (PDT), creating an instrumented copy
- Compiler generates instrumented object code
- Manual instrumentation

Library Level Instrumentation

- Statically or dynamically linked wrapper libraries
  - MPI, I/O, memory, etc.
- Wrapping external libraries where source is not available

Runtime pre-loading and interception of library calls

Binary Code instrumentation

- Rewrite the binary, runtime instrumentation

Virtual Machine, Interpreter, OS level instrumentation

Hands On

Configure your environment

Get workshop files from
http://www.paratools.com/sea13

```
% tar xvzf workshop.tgz
% module load workshop tau
% cd matmult
% make
```

```
Edit run.job to put in correct project id in -P:
% bsub run.job
Or
% bsub -Is -W 1:00 -n 8 -P <id> -q tutorial $SHELL
% pprof
% paraprof –pack foo.ppk
% paraprof foo.ppk   (either remotely or locally)
```

Using TAU

TAU supports several measurement and thread options

Phase profiling, profiling with hardware counters, MPI library, CUDA…

Each measurement configuration of TAU corresponds to a unique stub makefile and library that is generated when you configure it

To instrument source code automatically using PDT

Choose an appropriate TAU stub makefile in <arch>/lib:

% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt

% export TAU_OPTIONS='-optVerbose …' (see tau_compiler.sh)

% export PATH=/glade/apps/opt/tau/2.22.2-intelpoe-intel-papi/x86_64/bin:$PATH

Use tau_f90.sh, tau_cxx.sh, tau_upc.sh, or tau_cc.sh as F90, C++, UPC, or C compilers respectively:

% mpif90 foo.f90 changes to

% tau_f90.sh foo.f90

Set runtime environment variables, execute application and analyze performance data:

% pprof (for text based profile display)

% paraprof (for GUI)

Choosing TAU_MAKEFILE

```
% ls $TAU/Makefile.*
Makefile.tau
Makefile.tau-icpc-papi-mpi-pdt
Makefile.tau-icpc-papi-mpi-pdt-openmp-opari
Makefile.tau-icpc-papi-mpi-pdt-openmp-opari-scorep
Makefile.tau-icpc-papi-mpi-pdt-scorep
Makefile.tau-icpc-papi-pdt-openmp-opari
Makefile.tau-icpc-papi-pdt-openmp-opari-scorep
```

For an MPI+F90 application with Intel MPI, you may choose

Makefile.tau-icpc-papi-mpi-pdt

- Supports MPI instrumentation & PDT for automatic source instrumentation

```
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt
% tau_f90.sh matrix.f90 -o matrix
% bsub -Is -W 1:00 -n 8 -P SCSG0004 -q tutorial $SHELL
% mpirun.lsf ./matrix

% paraprof
```

Automatic Instrumentation

- **Use TAU's compiler wrappers**
  - Simply replace `CXX` with `tau_cxx.sh, etc.`
  - Automatically instruments source code, links with TAU libraries.
- **Use `tau_cc.sh` for C, `tau_f90.sh` for Fortran, `tau_upc.sh` for UPC, etc.**

Before

```
CXX = mpicxx
F90 = mpif90
CXXFLAGS =
LIBS = -lm
OBJS = f1.o f2.o f3.o … fn.o

app: $(OBJS)
     $(CXX) $(LDFLAGS) $(OBJS) -o $@
     $(LIBS)
.cpp.o:
     $(CXX) $(CXXFLAGS) -c $<
```

After

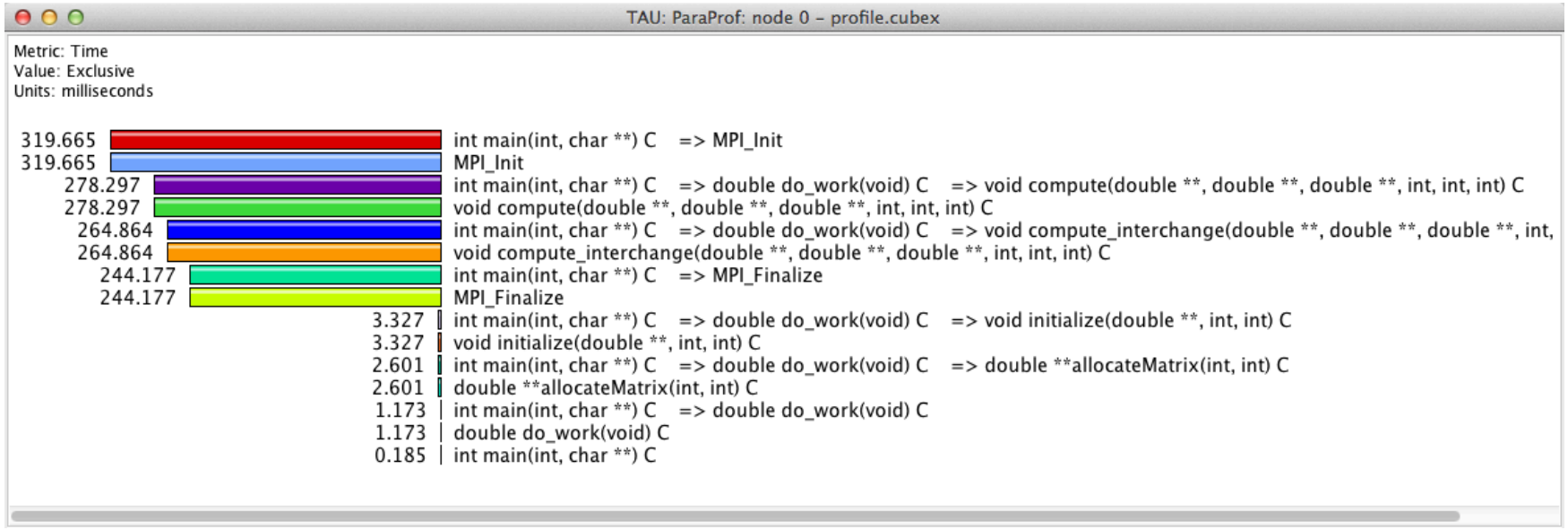
```
CXX = tau_cxx.sh
F90 = tau_f90.sh
CXXFLAGS =
LIBS = -lm
OBJS = f1.o f2.o f3.o … fn.o

app: $(OBJS)
     $(CXX) $(LDFLAGS) $(OBJS) -o $@
     $(LIBS)
.cpp.o:
     $(CXX) $(CXXFLAGS) -c $<
```

Generating a flat profile with MPI

```
% module load workshop tau
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt
% make F90=tau_f90.sh
Or
% tau_f90.sh matmult.f90
% qsub -I -l select:ncpus=4; mpirun.lsf ./a.out
% paraprof
To view. To view the data locally on the workstation,
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.
% paraprof app.ppk

Click on the "node 0" label to see profile for that node. Right
click to see other options. Windows -> 3D Visualization for 3D
window.
```

Routine Level Profile with Score-P

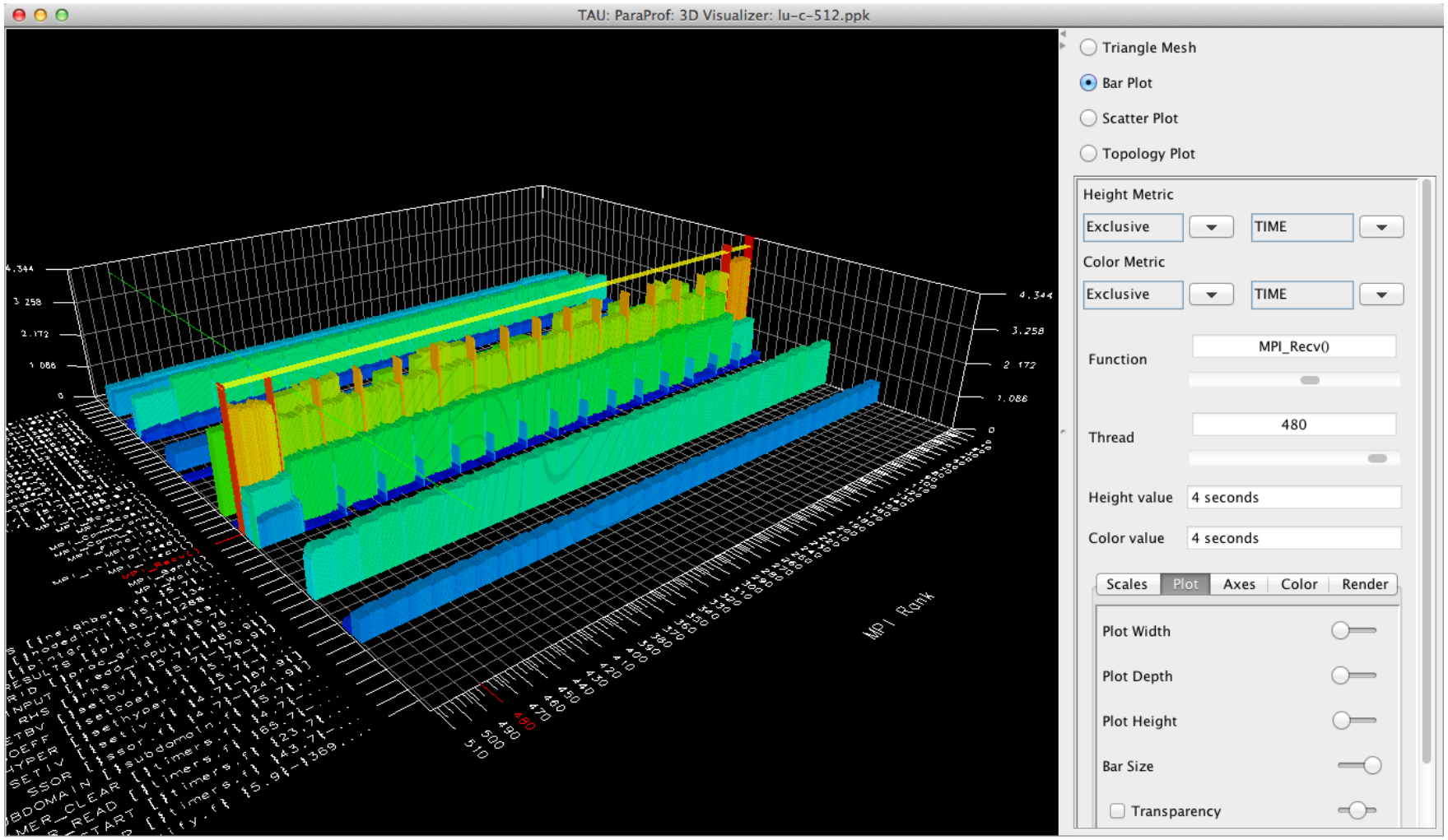
How much time is spent in each application routine?



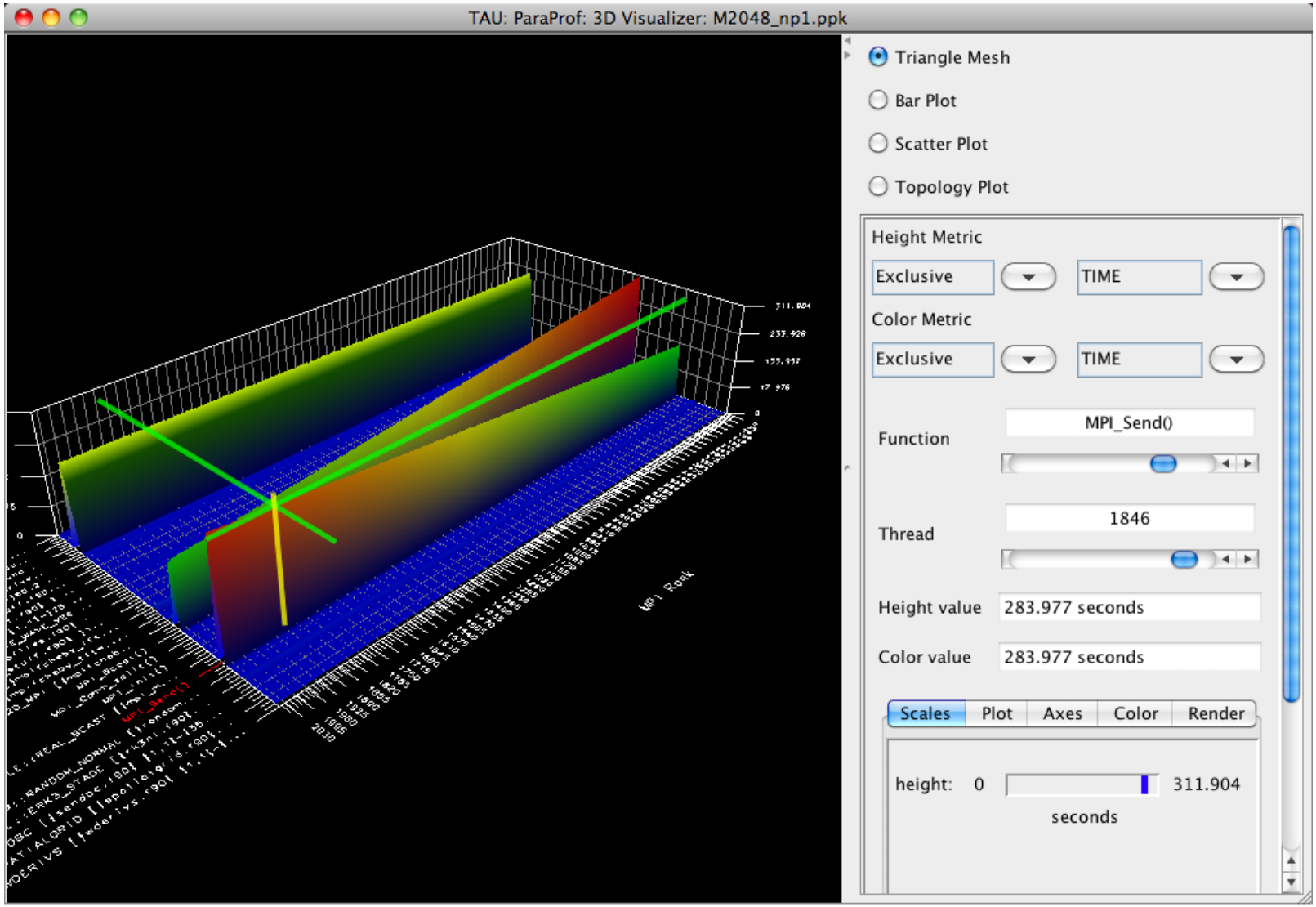
Generating a routine level profile

```
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt-
scorep

% module load tau
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% bsub -Is -W 1:00 -n 8 -P SCSG0004 -q tutorial $SHELL
% mpirun.lsf ./matmult
Execute the program and then launch paraprof or cube:
% cd scorep-<dir> ;
% paraprof profile.cubex
```

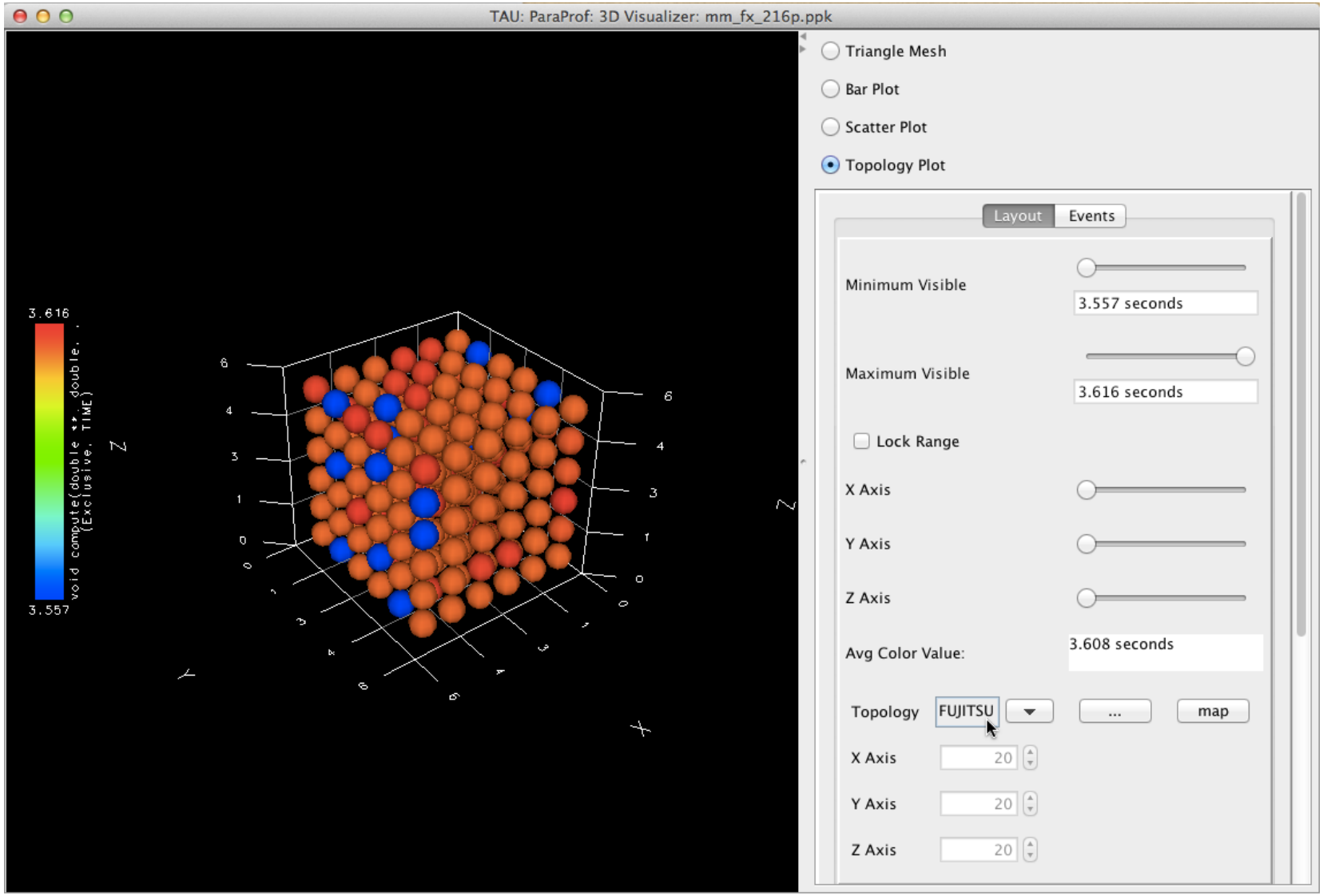
ParaProf 3D Profile Browser



ParaProf


TAU: ParaProf: 3D Visualizer: M2048_np1.ppk
Triangle Mesh
Bar Plot
Scatter Plot
Topology Plot
Height Metric
Exclusive
TIME
Color Metric
Exclusive
TIME
Function
MPI_Send()
Thread
1846
Height value
283.977 seconds
Color value
283.977 seconds
Scales
Plot
Axes
Color
Render
height: 0
311.904
seconds
MPI Rank

ParaProf 3D Topology Display



Generating a loop level profile

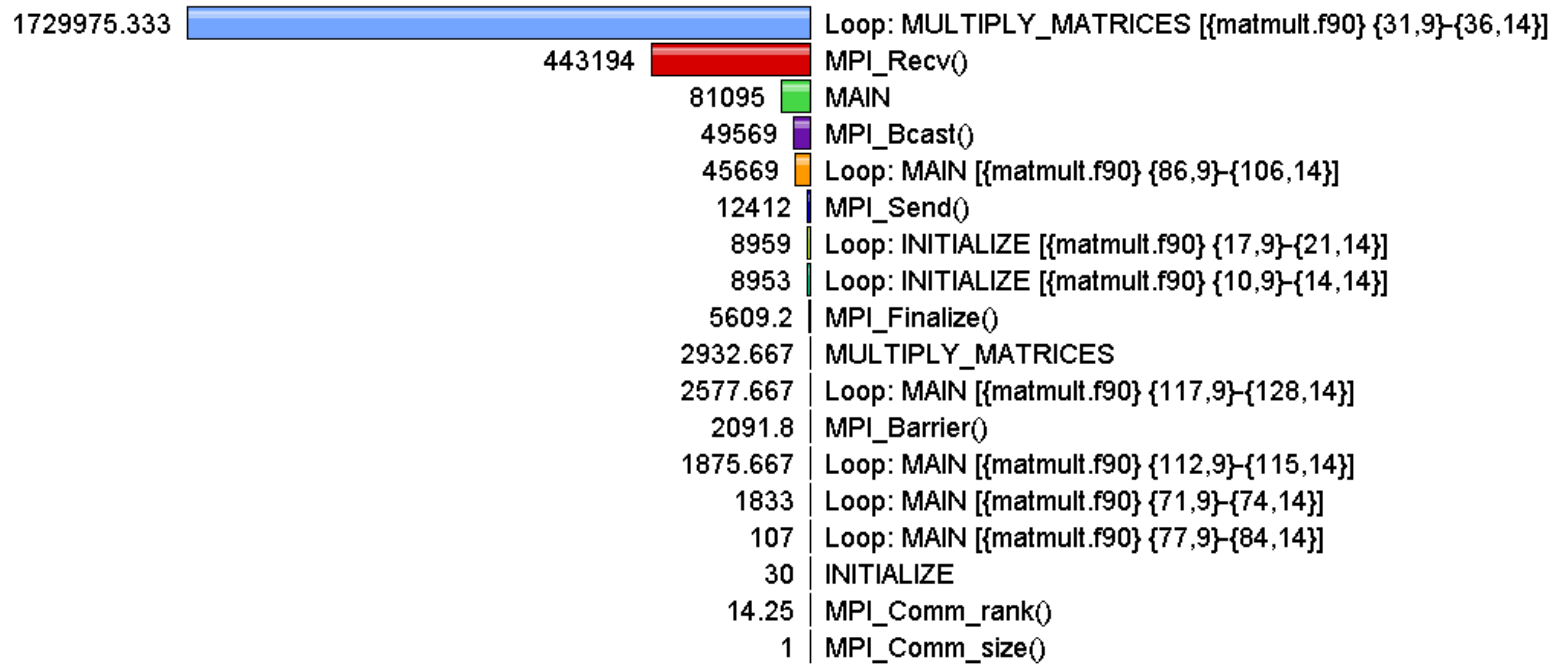
```
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt
% export TAU_OPTIONS='-optTauSelectFile=select.tau -optVerbose'
% cat select.tau
  BEGIN_INSTRUMENT_SECTION
  loops routine="#"
  END_INSTRUMENT_SECTION

% module load tau
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)

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

% paraprof app.ppk
```

Loop Level Instrumentation

Goal: What loops account for the most time? How much?

Flat profile with wallclock time 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()

Pa

Profiling with multiple counters

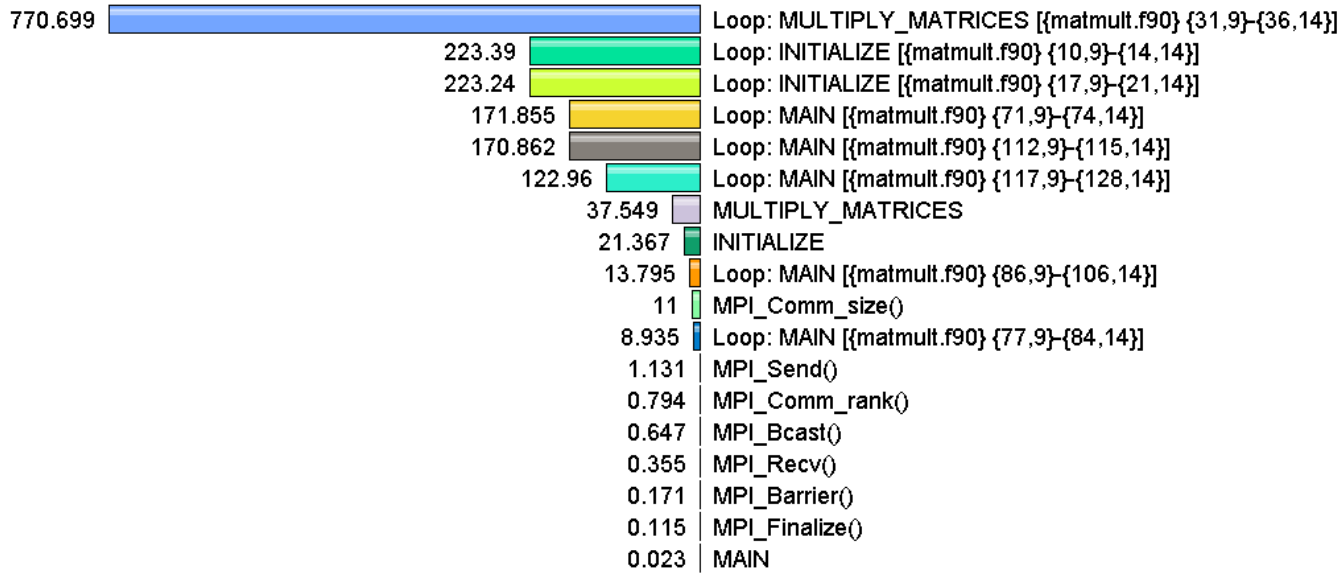
```
% module load workshop tau
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt
% export TAU_OPTIONS='-optTauSelectFile=select.tau -optVerbose'
% cat select.tau
  BEGIN_INSTRUMENT_SECTION
  loops routine="#"
  END_INSTRUMENT_SECTION
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% setenv REQUEST_SUSPEND_HPC_STAT 1
% bsub -Is -W 1:00 -n 8 -P SCSG0004 -q tutorial $SHELL
% export TAU_METRICS=TIME:PAPI_FP_INS:PAPI_L1_DCM
% mpirun.lsf ./matmult
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.
% paraprof app.ppk
  Choose Options -> Show Derived Panel -> Click PAPI_FP_INS,
 Click "/",  Click TIME, Apply, Choose new metric by double
clicking.
```

Computing FLOPS per loop

Goal: What is the execution rate of my loops in MFLOPS?

Flat profile with PAPI_FP_INS and time with loop instrumentation:

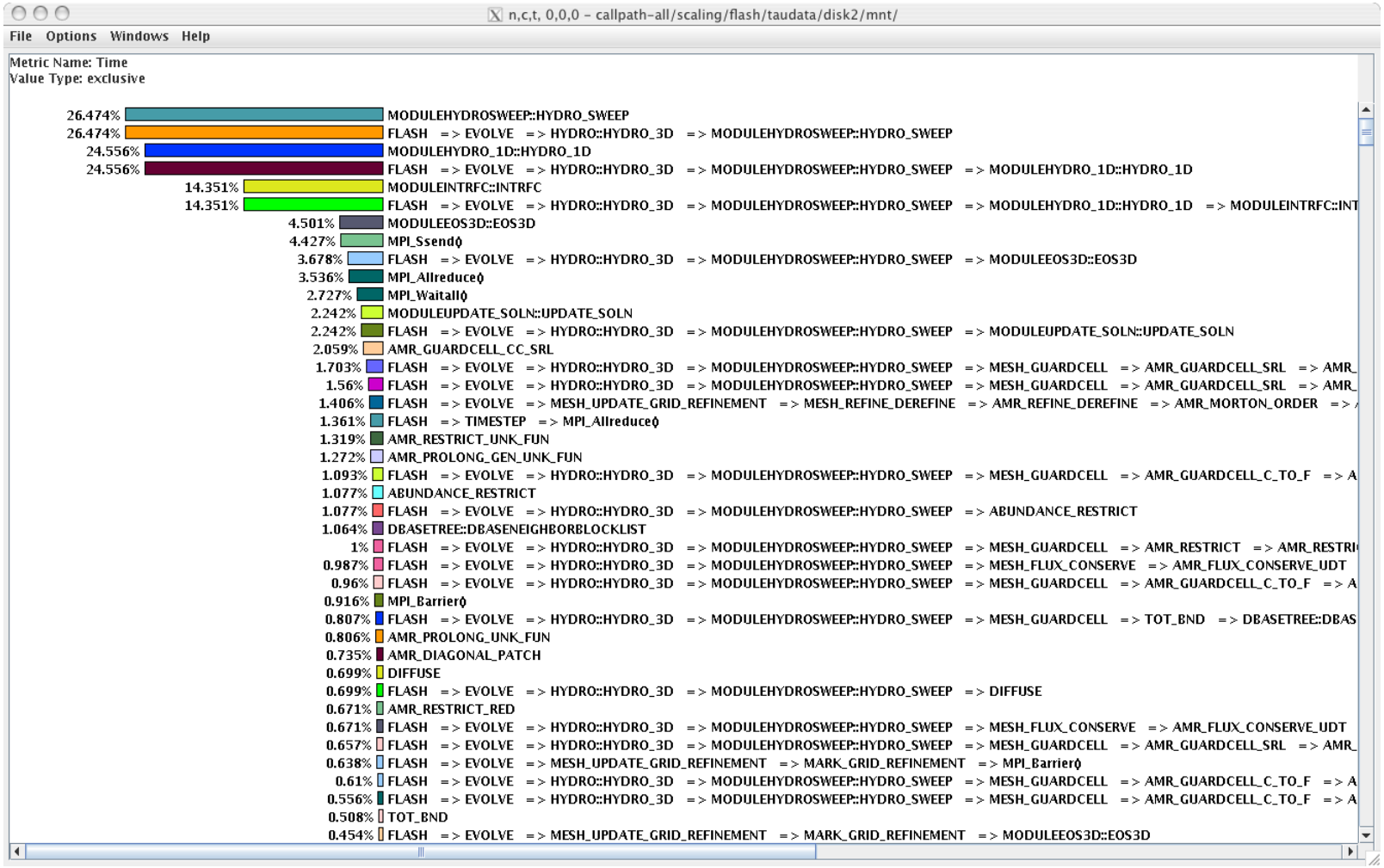
Metric: PAPI_FP_INS / GET_TIME_OF_DAY
Value: Exclusive
Units: Derived metric shown in microseconds format



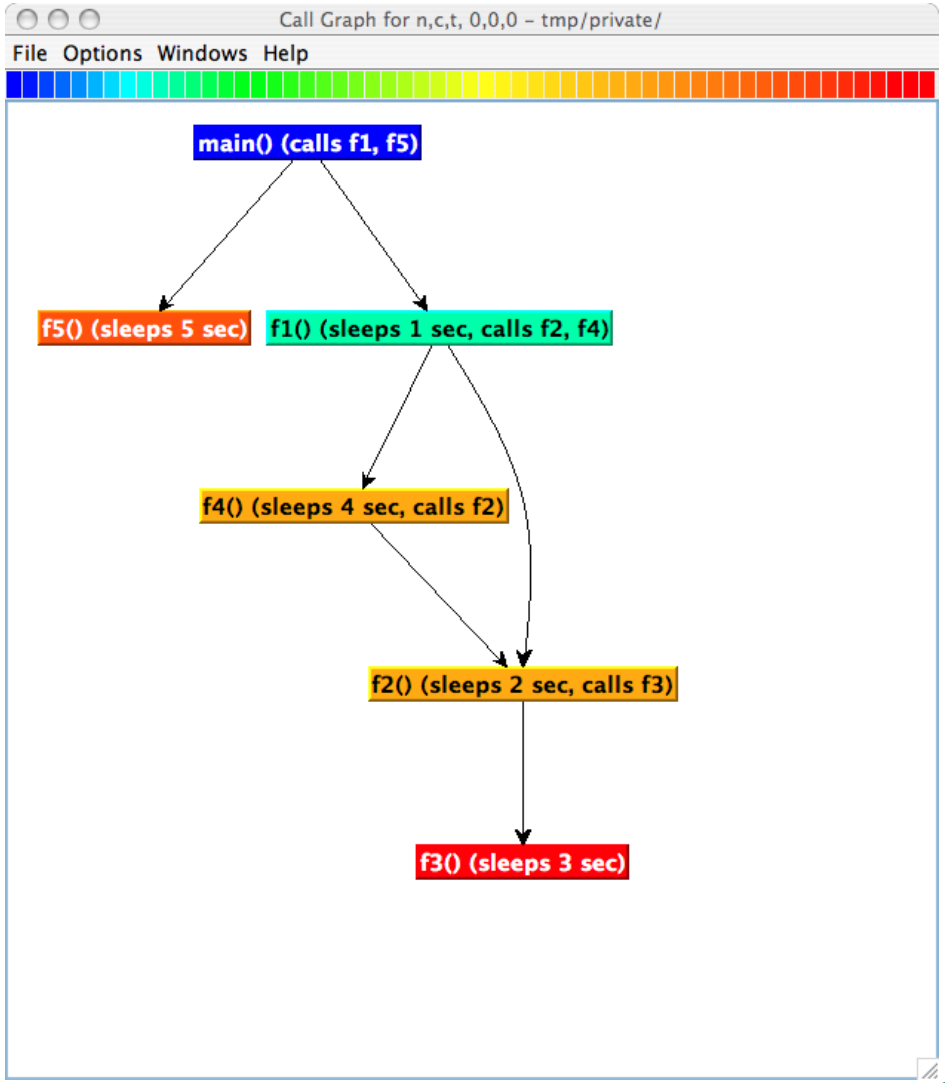
Generate a Callpath Profile

```
% module load workshop tau
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)

% bsub -I -q tutorial …
% export TAU_CALLPATH=1
% export TAU_CALLPATH_DEPTH=100
(truncates all calling paths to a specified depth)
% mpirun.lsf  ./a.out
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.
% paraprof app.ppk
(Windows -> Thread -> Call Graph)
```

Callpath Profile



ParaProf Call Graph Window

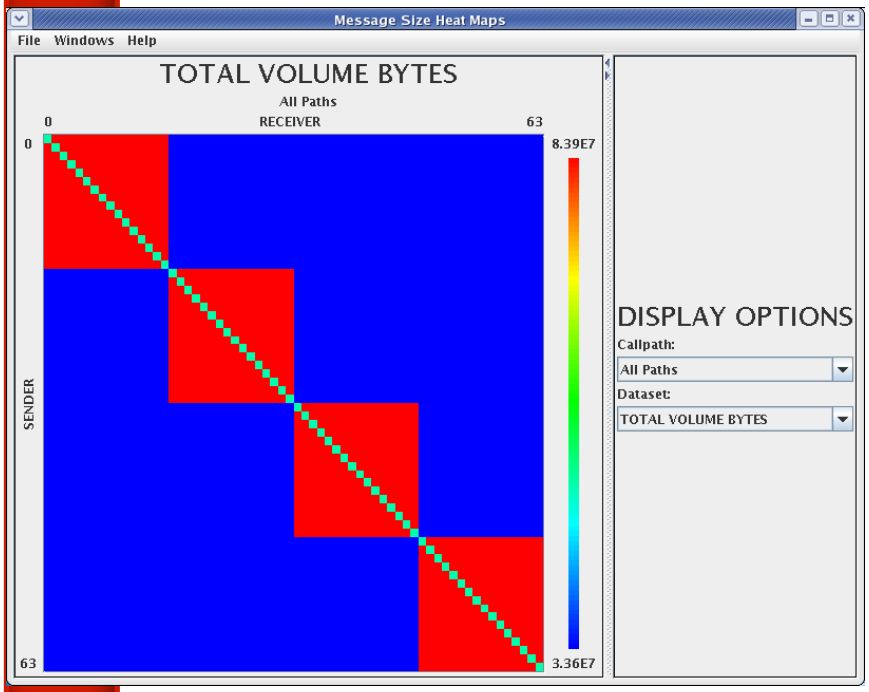


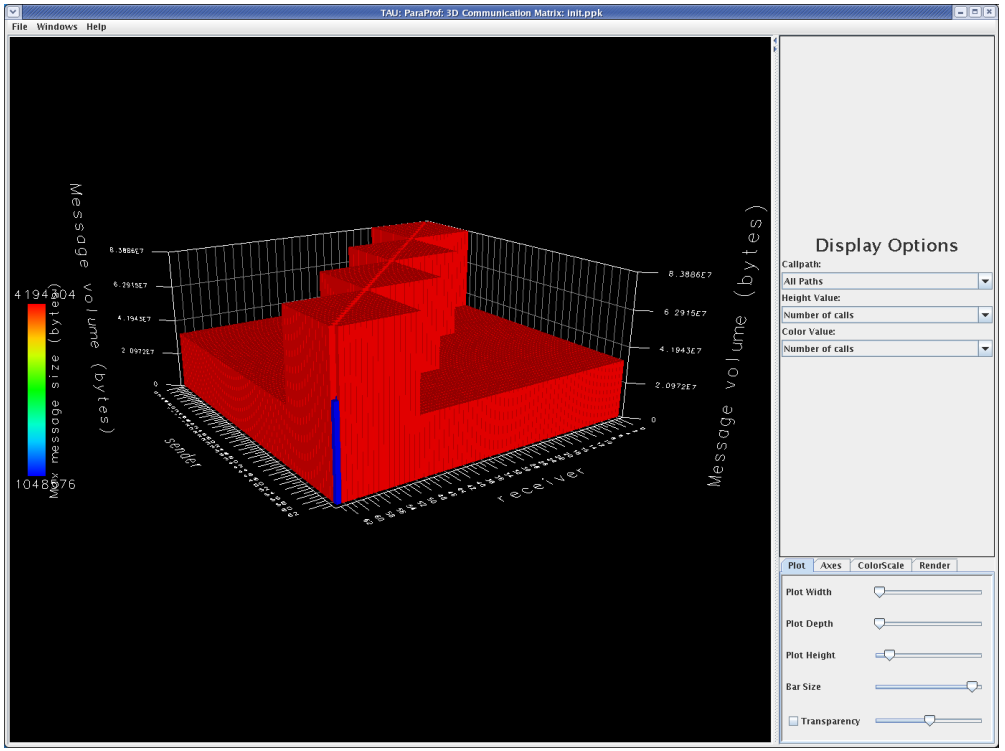
Generating Communication Matrix

```
% module load workshop tau
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)

% qsub -I -l ncpus=4;
% export TAU_COMM_MATRIX=1
% mpirun.lsf  ./a.out

% paraprof
(Windows -> Communication Matrix)
(Windows -> 3D Communication Matrix)
```

Communication Matrix Display

Goal: What is the volume of inter-process communication? Along which calling path?





Compiler-based Instrumentation

- Compiler automatically **emits instrumentation calls** in the object code instead of parsing the source code using PDT
- To enable: export TAU_OPTIONS="-optCompInst"
- Configure TAU with "-bfd=download" for best results

Use Compiler-Based Instrumentation

```
% module load workshop tau
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt
% export TAU_OPTIONS='-optCompInst -optQuiet'

% make CC=tau_cc.sh CXX=tau_cxx.sh F90=tau_f90.sh

NOTE: You may also use the short-hand scripts taucc, tauf90,
taucxx instead of specifying TAU_OPTIONS and using the
traditional tau_<cc,cxx,f90>.sh scripts. These scripts use
compiler-based instrumentation by default.

% make CC=taucc CXX=taucxx F90=tauf90

% mpirun.lsf./a.out
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.
% paraprof app.ppk
```

Binary Rewriting Instrumentation

- Support for both **static and dynamic** executables
- Specify a list of routines to instrument
- Specify the TAU measurement library to be injected
- **Dyninst**:

```
% tau_run -T [tags] a.out -o a.inst
```

- **MAQAO**:

```
% tau_rewrite -T [tags] a.out -o a.inst
```

- **Pebil**:

```
% tau_pebil_rewrite -T [tags] a.out \
  -o a.inst
```

- Execute the application to get measurement data:

```
% mpirun.lsf ./a.inst
```

Use Binary Rewriting Instrumentation

```
% module load workshop tau
% mpif90 -g matmult.f90 -o matmult
% tau_rewrite matmult matmult.i

Or
% tau_rewrite -T icpc,mpi,papi,pdt ./matmult -o matmult.i
% mpirun.lsf ./matmult.i
% paraprof

Score-P:
```

Binary Rewriting with Score-P

```
% mpif90 matmult.f90 -o matmult -g
% tau_rewrite -T scorep ./matmult -o ./matmult.i

% bsub -I …
Uninstrumented:
% mpirun.lsf ./matmult

Instrumented with Score-P:

% mpirun.lsf tau_exec -T scorep -loadlib=/glade/apps/opt/unite/
packages/scorep/1.1.1-intelpoe-intel-papi/lib/libscorep_mpi.so
./matmult.i

% cd scorep-<dir>; paraprof profile.cubex &
% export SCOREP_ENABLE_TRACING=1
% mpirun.lsf tau_exec -T scorep -loadlib=/glade/apps/opt/unite/
packages/scorep/1.1.1-intelpoe-intel-papi/lib/libscorep_mpi.so
./matmult.i
% export PATH=/glade/apps/opt/unite/packages/vampir/8.0.1/bin/:$PATH
% cd scorep-<dir>; vampir traces.otf2 &
```

Compiler-based Instrumentation

```
% module load workshop tau
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt
% export TAU_OPTIONS='-optCompInst -optQuiet'

% make CC=tau_cc.sh CXX=tau_cxx.sh F90=tau_f90.sh

NOTE: You may also use the short-hand scripts taucc, tauf90,
taucxx instead of specifying TAU_OPTIONS and using the
traditional tau_<cc,cxx,f90>.sh scripts. These scripts use
compiler-based instrumentation by default.

% make CC=taucc CXX=taucxx F90=tauf90

% mpirun.lsf./a.out
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.
% paraprof app.ppk
```

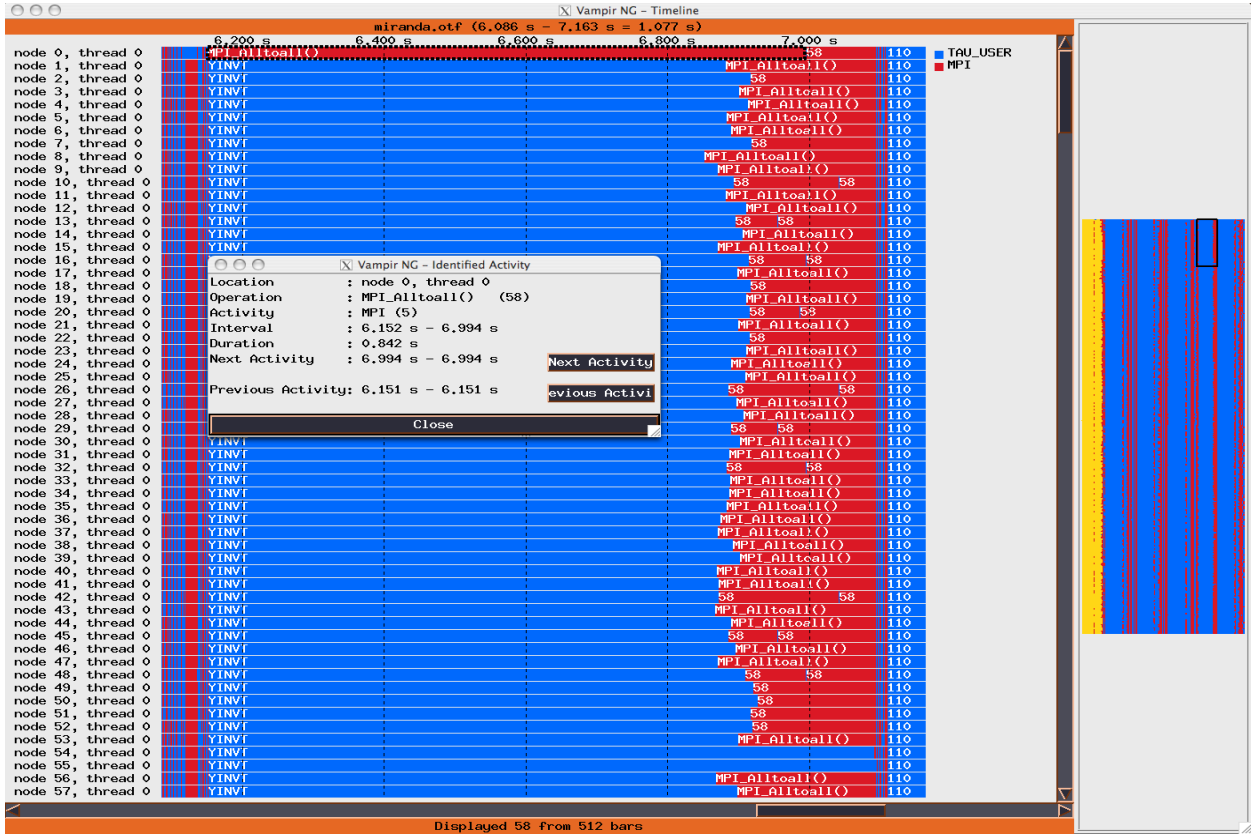
Generating Event Traces

```
% module load workshop tau
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)

% bsub -I
% export TAU_TRACE=1
% mpirun.lsf  ./a.out

Merge and convert the travefiles:
% tau_treemerge.pl
For Vampir (OTF):
% tau2otf tau.trc tau.edf app.otf; vampir app.otf

For Jumpshot (SLOG2):
% tau2slog2 tau.trc tau.edf -o app.slog2; jumpshot app.slog2

For ParaVer:
% tau_convert -paraver tau.trc tau.edf app.prv; paraver app.prv
```

Generating a Trace File

Goal: Identify the temporal aspect of performance. What happens in my code at a given time? When?

Event trace visualized in Vampir/Jumpshot/Paraver



Stencil2D Parallel Profile / Trace in Vampir

Metric: TAUGPU_TIME
Value: Exclusive

Runtime Preloading: tau_exec

- Runtime instrumentation by pre-loading the measurement library
- Works on dynamic executables (default under Linux)
- Can substitute I/O, MPI, SHMEM, CUDA, OpenCL, and memory allocation/deallocation routines with instrumented calls
- Track interval events (e.g., time spent in write()) as well as atomic events (e.g., how much memory was allocated) in wrappers
- Accurately measure I/O and memory usage
- Preload any wrapper interposition library in the context of the executing application

Preloading a TAU Library

```
% ./configure -pdt=<dir> -mpi -papi=<dir>; make install
```
Creates in $TAU:
```
Makefile.tau-papi-mpi-pdt
shared-papi-mpi-pdt/libTAU.so

% ./configure -pdt=<dir> -mpi; make install
```
creates
```
Makefile.tau-icpc-papi-mpi-pdt
shared-mpi-pdt/libTAU.so
```
To explicitly choose preloading of shared-<options>/libTAU.so change:
```
% mpirun.lsf./a.out     to
% mpirun.lsf tau_exec -T <comma_separated_options> ./a.out

% mpirun.lsf tau_exec -T papi,mpi,pdt ./a.out
```
Preloads `$TAU/shared-papi-mpi-pdt/libTAU.so`
```
% mpirun.lsf tau_exec -T papi ./a.out
```
Preloads `$TAU/shared-papi-mpi-pdt/libTAU.so by matching.`
```
% mpirun -np 8 tau_exec -T papi,mpi,pdt -s ./a.out
```
Does not execute the program. Just displays the library that it will preload if executed without the **–s** option.

NOTE: -mpi configuration is selected by default. Use –T serial for Sequential programs.

TAU Execution Command (tau_exec)

Uninstrumented execution

- % mpirun.lsf ./a.out

Track MPI performance

- % mpirun.lsf tau_exec ./a.out

Track POSIX I/O and MPI performance (MPI enabled by default)

- % mpirun.lsf tau_exec –io ./a.out

Track memory operations

- % export TAU_TRACK_MEMORY_LEAKS=1
- % mpirun –np 8 tau_exec –memory_debug ./a.out (bounds check)

Use event based sampling (compile with –g)

- % mpirun –np 8 tau_exec –ebs ./a.out
- Also –ebs_source=<PAPI_COUNTER> -ebs_period=<overflow_count>

Load wrapper interposition library

- % mpirun –np 8 tau_exec –loadlib=<path/libwrapper.so> ./a.out

Track GPGPU operations

- % mpirun –np 8 tau_exec –cupti ./a.out
- % mpirun –np 8 tau_exec –opencl ./a.out

Tags in tau_exec and other tools

```
% cd $TAU; ls Makefile.*
Makefile.tau-icpc-papi-mpi-pdt

% qsub -I -l select:ncpus=4; mpirun.lsf./matrix
% tau_exec -T icpc,mpi,pdt ./a.out

Chooses Makefile.tau-icpc-mpi,pdt and associated libraries.

% tau_exec -T serial,pdt ./a.out
Chooses Makefile.tau-pdt or the shortest Makefile name without -mpi.
```

-T <list_of_tags> is used in several TAU tools:

- tau_run
- tau_rewrite
- tau_exec
- tau_gen_wrapper

Three Instrumentation Techniques for Wrapping External Libraries

Pre-processor based substitution by re-defining a call (e.g., read)

- Tool defined header file with same name <*unistd.h*> takes precedence
- Header redefines a routine as a different routine using macros
- Substitution: *read()* substituted by preprocessor as *tau_read()* at callsite

Preloading a library at runtime

- Library preloaded (*LD_PRELOAD* env var in Linux) in the address space of executing application intercepts calls from a given library
- Tool's wrapper library defines *read(),* gets address of global *read()* symbol (dlsym), internally calls timing calls around call to global read

Linker based substitution

- Wrapper library defines *__wrap_read* which calls *__real_read* and linker is passed *-Wl,-wrap,read* to substitute all references to read from application's object code with the *__wrap_read* defined by the tool

Issues: Preprocessor based substitution

Pre-processor based substitution by re-defining a call

- Compiler replaces read() with tau_read() in the body of the source code

Advantages:

- Simple to instrument
  - Preprocessor based replacement
  - A header file redefines the calls
  - No special linker or runtime flags required

Disadvantages

- Only works for C & C++ for replacing calls in the body of the code.
- Incomplete instrumentation: fails to capture calls in uninstrumented libraries (e.g., libhdf5.a)

Issues: Linker based substitution

Linker based substitution

- Wrapper library defines __wrap_read which calls __real_read and linker is passed -Wl,-wrap, read

Advantages

- Tool can intercept all references to a given call
- Works with static as well as dynamic executables
- No need to recompile the application source code, just re-link the application objects and libraries with the tool wrapper library

Disadvantages

- Wrapping an entire library can lengthen the linker command line with multiple –Wl,-wrap,<func> arguments. It is better to store these arguments in a file and pass the file to the linker
- Approach does not work with un-instrumented binaries

tau_gen_wrapper

Automates creation of wrapper libraries using TAU

Input:

- header file (foo.h)
- library to be wrapped (/path/to/libfoo.a)
- technique for wrapping
  - Preprocessor based redefinition (-d)
  - Runtime preloading (-r)
  - Linker based substitution (-w: default)
- Optional selective instrumentation file (-f select)
  - Exclude list of routines, or
  - Include list of routines

Output:

- wrapper library
- optional *link_options.tau* file (-w), pass –optTauWrapFile=<file> in TAU_OPTIONS environment variable

Design of wrapper generator (tau_gen_wrapper)

***tau_gen_wrapper* shell script:**

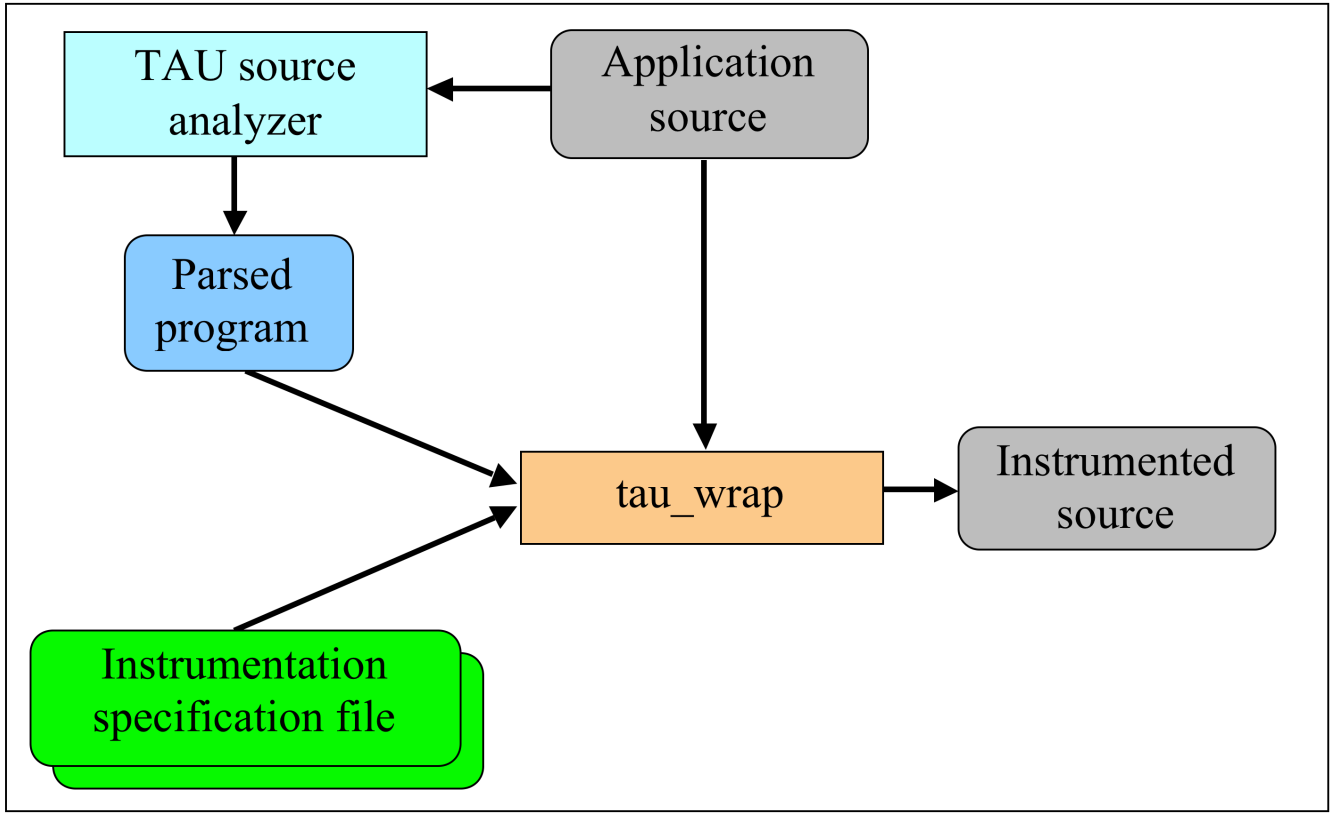
- parses source of header file using static analysis tool Program Database Toolkit (PDT)
- Invokes *tau_wrap*, a tool that generates
  - instrumented wrapper code,
  - an optional *link_options.tau* file (for linker-based substitution, -w)
  - Makefile for compiling the wrapper interposition library
- Builds the wrapper library using make

Use TAU_OPTIONS environment variable to pass location of link_options.tau file using

```
% export TAU_OPTIONS='-optTauWrapFile=<path/to/
link_options.tau> -optVerbose'
```

Use *tau_exec –loadlib=<wrapperlib.so>* to pass location of wrapper library for preloading based substitution

tau_wrap


TAU source analyzer
Application source
Parsed program
tau_wrap
Instrumented source
Instrumentation specification file

HDF5 Library Wrapping

```
[sameer@zorak]$ tau_gen_wrapper hdf5.h /usr/lib/libhdf5.a -f select.tau

Usage : tau_gen_wrapper <header> <library> [-r|-d|-w (default)] [-g
groupname] [-i headerfile] [-c|-c++|-fortran] [-f <instr_spec_file> ]
• instruments using runtime preloading (-r), or -Wl,-wrap linker (-w),
redirection of header file to redefine the wrapped routine (-d)
• instrumentation specification file (select.tau)
• group (hdf5)
• tau_exec loads libhdf5_wrap.so shared library using –
loadlib=<libwrap_pkg.so>
• creates the wrapper/ directory

NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0        0.057            1           1          13       1236 .TAU Application
 70.8        0.875        0.875           1           0        875 hid_t H5Fcreate()
  9.7         0.12         0.12           1           0        120 herr_t H5Fclose()
  6.0        0.074        0.074           1           0         74 hid_t H5Dcreate()
  3.1        0.038        0.038           1           0         38 herr_t H5Dwrite()
  2.6        0.032        0.032           1           0         32 herr_t H5Dclose()
  2.1        0.026        0.026           1           0         26 herr_t
H5check_version()
```

Using POSIX I/O wrapper library

Setting environment variable TAU_OPTIONS=-optTrackIO links in TAU's wrapper interposition library using linker-based substitution

Instrumented application generates bandwidth, volume data

Workflow:

- % export TAU_OPTIONS= '-optTrackIO –optVerbose'
- % export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt
- % make CC=tau_cc.sh CXX=tau_cxx.sh F90=tau_f90.sh
- % mpirun –np 8 ./a.out
- % paraprof

Get additional data regarding individual arguments by setting environment variable TAU_TRACK_IO_PARAMS=1 prior to running

Preloading a wrapper library

Preloading a library at runtime

- Tool defines read(), gets address of global read() symbol (dlsym), internally calls timing calls around call to global read
- *tau_exec* tool uses this mechanism to intercept library calls

Advantages

- No need to re-compile or re-link the application source code
- Drop-in replacement library implemented using LD_PRELOAD environment variable under Linux, Cray CNL, IBM BG/P CNK, Solaris…

Disadvantages

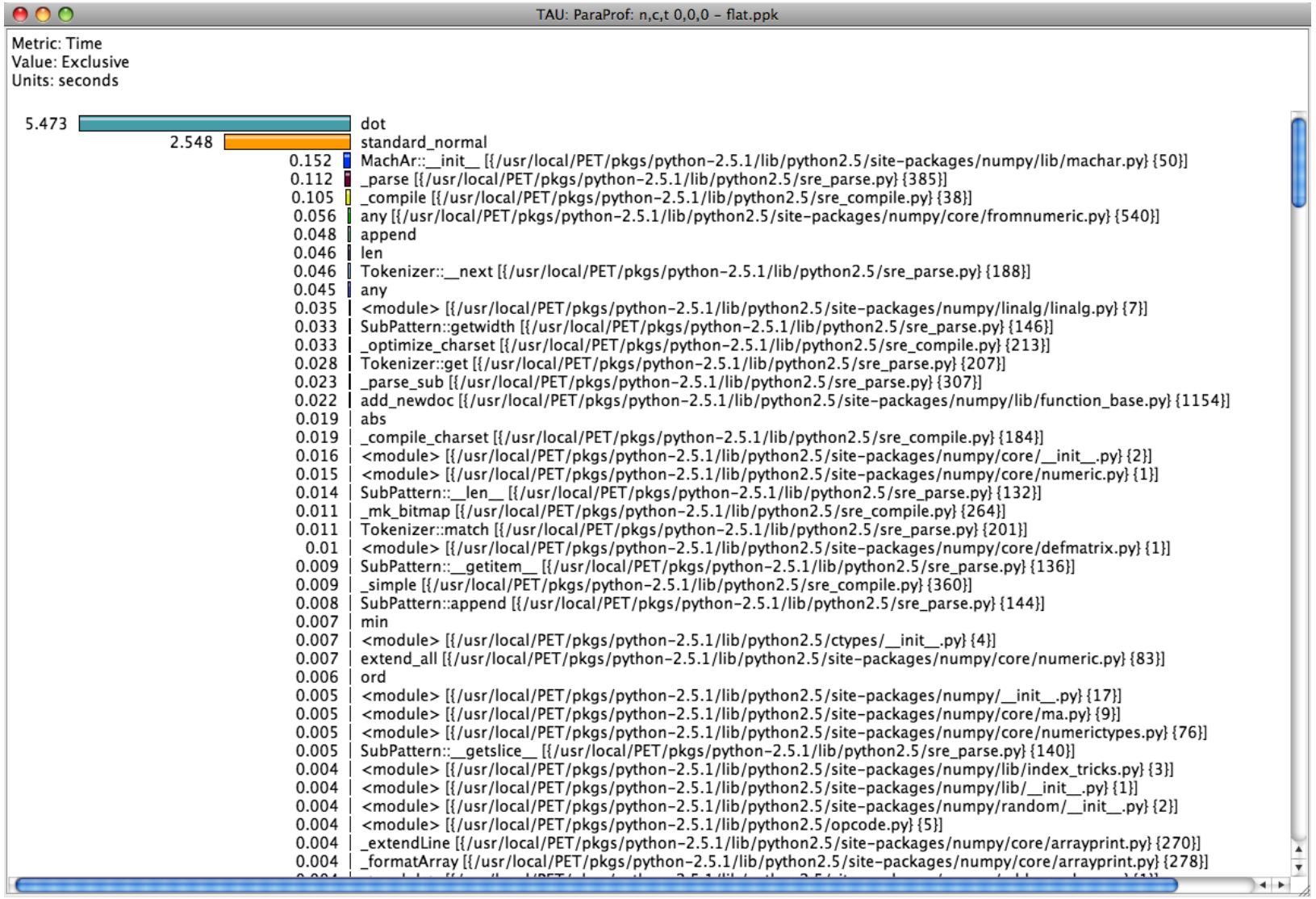
- Only works with dynamic executables. Default compilation mode under Cray XE6 and IBM BG/P is to use static executables
- Not all operating systems support preloading of dynamic shared objects (DSOs)

Profiling Python applications

- Create a top-level Python wrapper
- Launch with tau_exec -T python …

```
% cat wrapper.py

#!/bin/env python
import tau
import sys

def OurMain():
    try:
        import <your_application>
except SystemExit:
        # Intercept the exit call so Tau writes profiles
        print 'SystemExit intercepted by wrapper'

tau.run('OurMain()')

% tau_exec -T python wrapper.py
```

Profiling Python applications

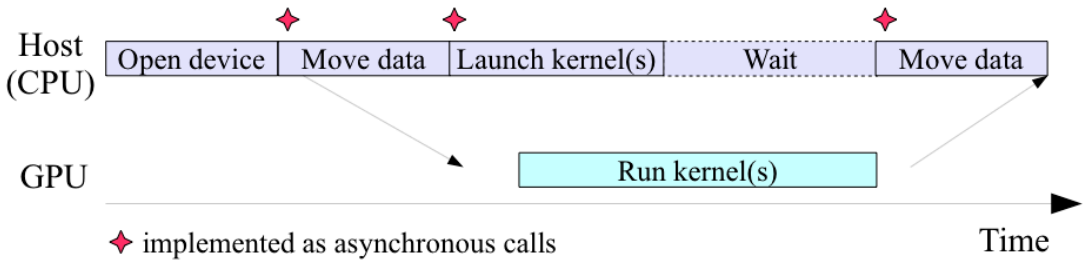


Profiling GPGPU Executions

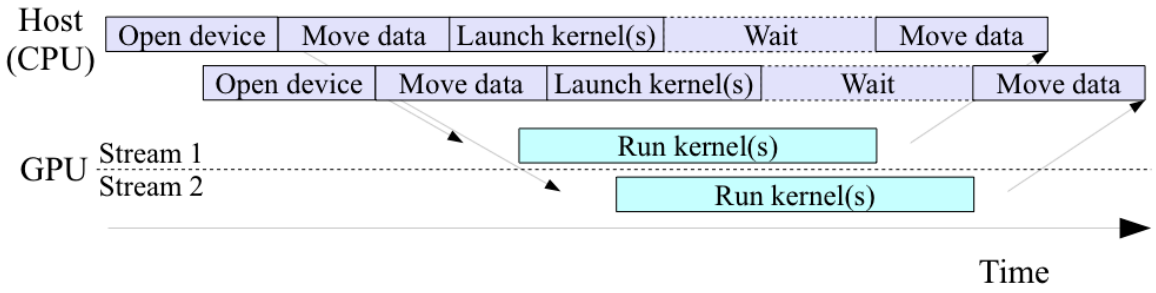
- GPGPU compilers (e.g. CAPS Compilers and PGI) can automatically generate GPGPU code using manual annotation of loop-level constructs and routines
- The loops (and routines for HMPP) are transferred automatically to the GPGPU
- TAU intercepts the runtime library routines and examines the arguments
- Shows events as seen from the host
- Profiles and traces GPGPU execution

Host (CPU) - GPU Scenarios

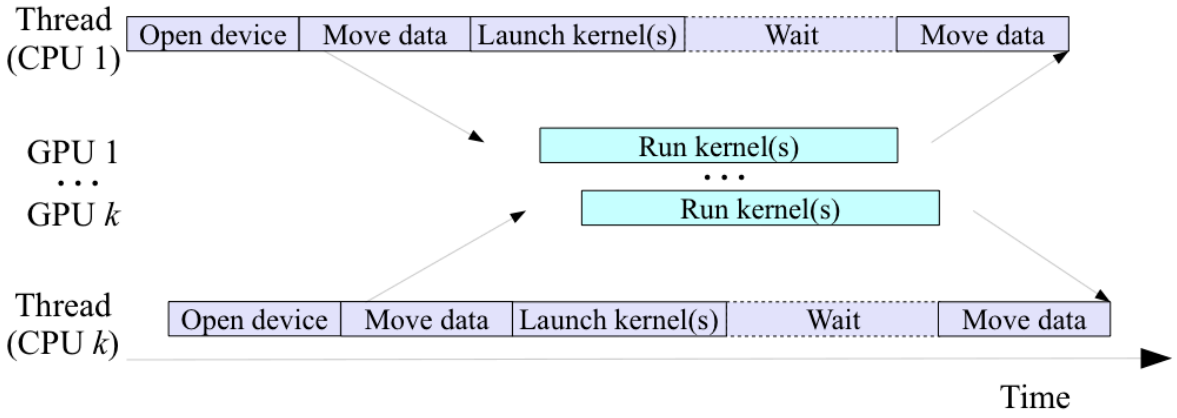
Single GPU



Multi-stream

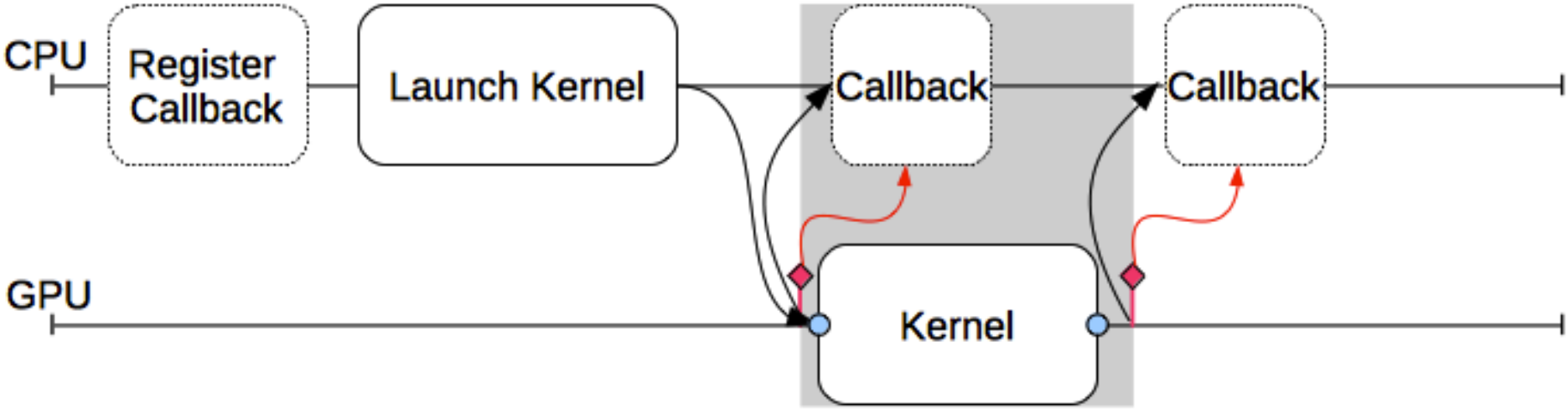


Multi-CPU, Multi-GPU



Host-GPU Measurement – Callback Method

- GPU driver libraries provide callbacks for certain routines and captures measurements
- Measurement tool registers the callbacks and processes performance data
- Application code is not modified



Method Support and Implementation

Synchronous method

- Place instrumentation appropriately around GPU calls (kernel launch, library routine, …)
- Wrap (synchronous) library with performance tool

Event queue method

- Utilize CUDA and OpenCL event support
- Again, need instrumentation to create and insert events in the streams with kernel launch and process events
- Can be implemented with driver library wrapping

Callback method

- Utilize language-level callback support in OpenCL
- Utilize NVIDIA CUDA Performance Tool Interface (CUPTI)
- Need to appropriately register callbacks

GPU Performance Measurement Tools

Support the Host-GPU performance perspective

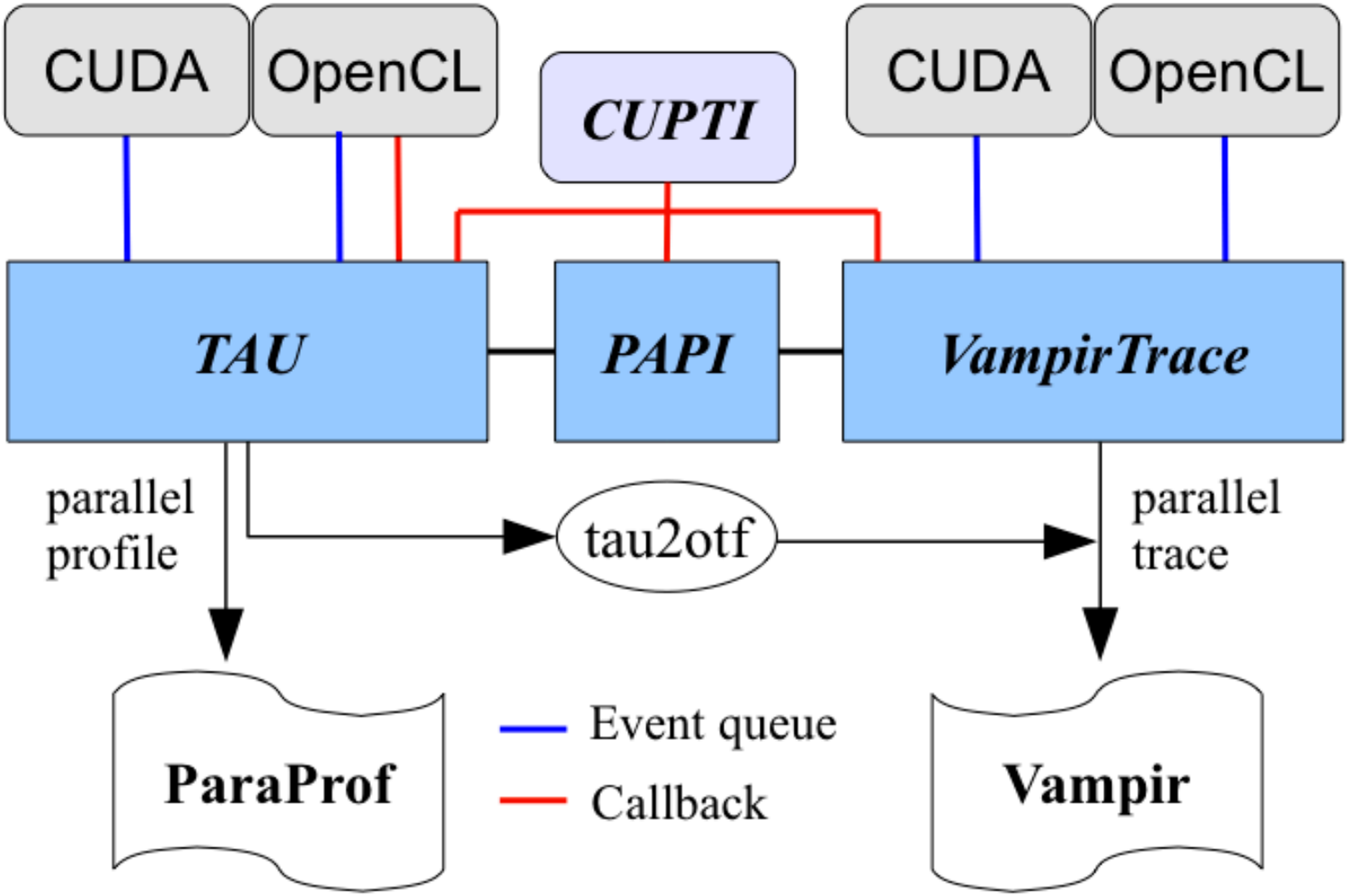
Provide integration with existing measurement system to facilitate tool use

Utilize support in GPU driver library and device

Tools

- TAU performance system
- Vampir
- PAPI
- NVIDIA CUPTI

GPU Performance Tool Interoperability



NVIDIA CUPTI

NVIDIA is developing CUPTI to enable the creation of profiling and tracing tools

Callback API

- Interject tool code at the entry and exist to each CUDA runtime and driver API call

Counter API

- Query, configure, start, stop, and read the counters on CUDA-enabled devices

CUPTI is delivered as a dynamic library

CUPTI is released with CUDA 4.0+

TAU for Heterogeneous Measurement

Multiple performance perspectives

Integrate Host-GPU support in TAU measurement framework

- Enable use of each measurement approach
- Include use of PAPI and CUPTI
- Provide profiling and tracing support

Tutorial

- Use TAU library wrapping of libraries
- Use `tau_exec` to work with binaries

```
% ./a.out    (uninstrumented)
% tau_exec –T serial,cupti –cupti ./a.out
% paraprof
```

Example: SDK simpleMultiGPU

Demonstration of multiple GPU device use

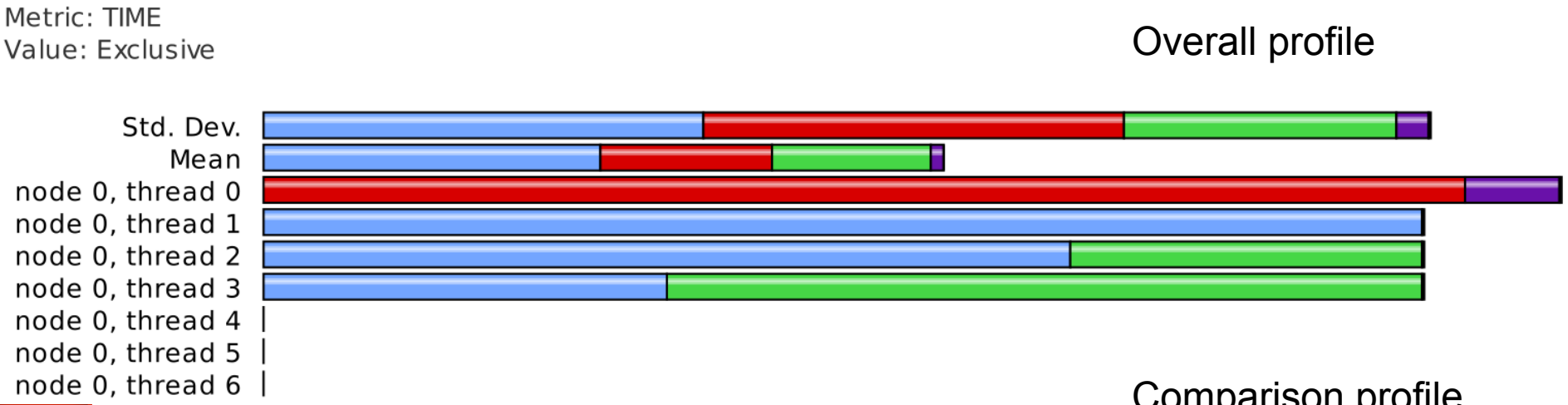
main → ***solverThread*** → ***reduceKernel***

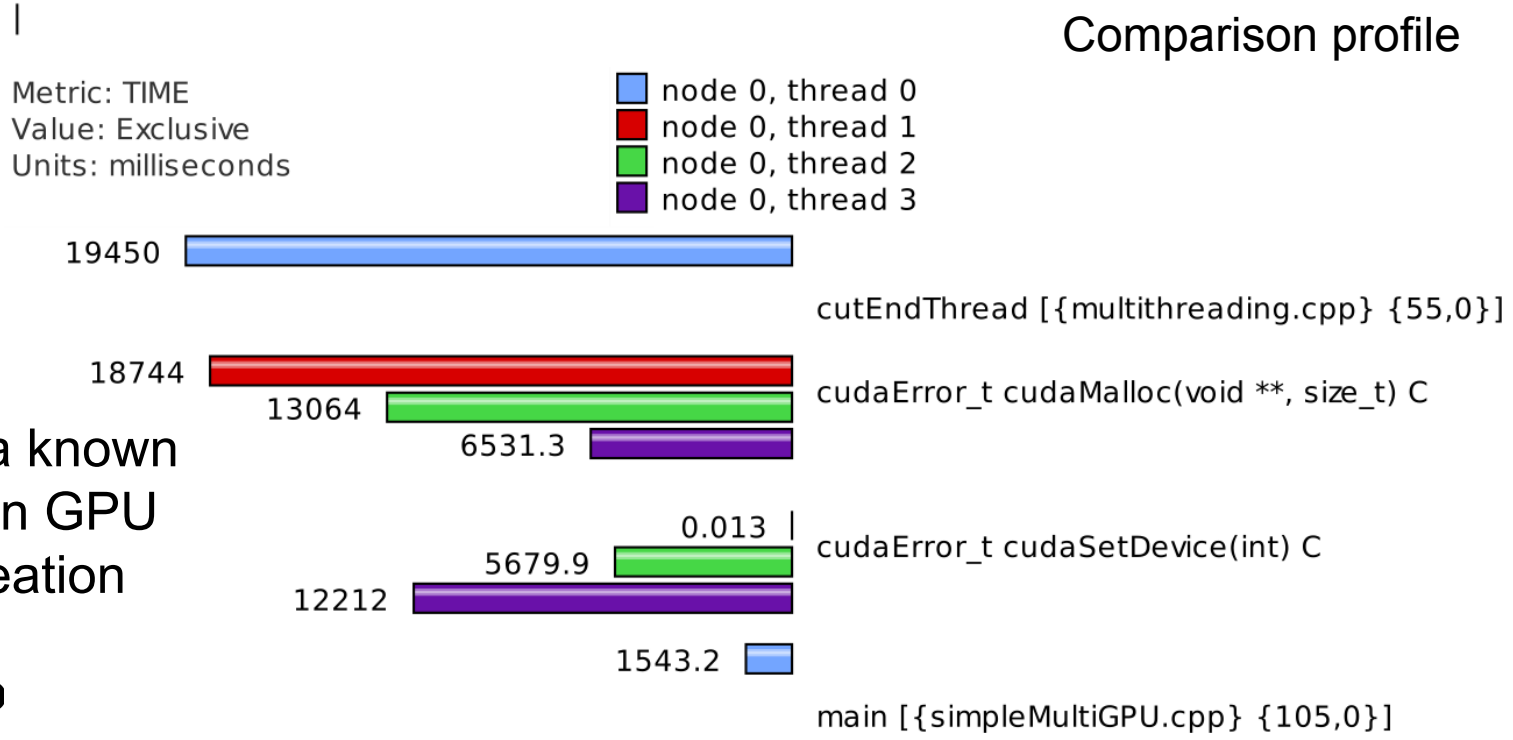
One Keeneland node with three GPUs

Performance profile for:

- One *main* thread
- Three *solverThread* threads
- Three *reduceKernel* "threads"

simpleMultiGPU Profile



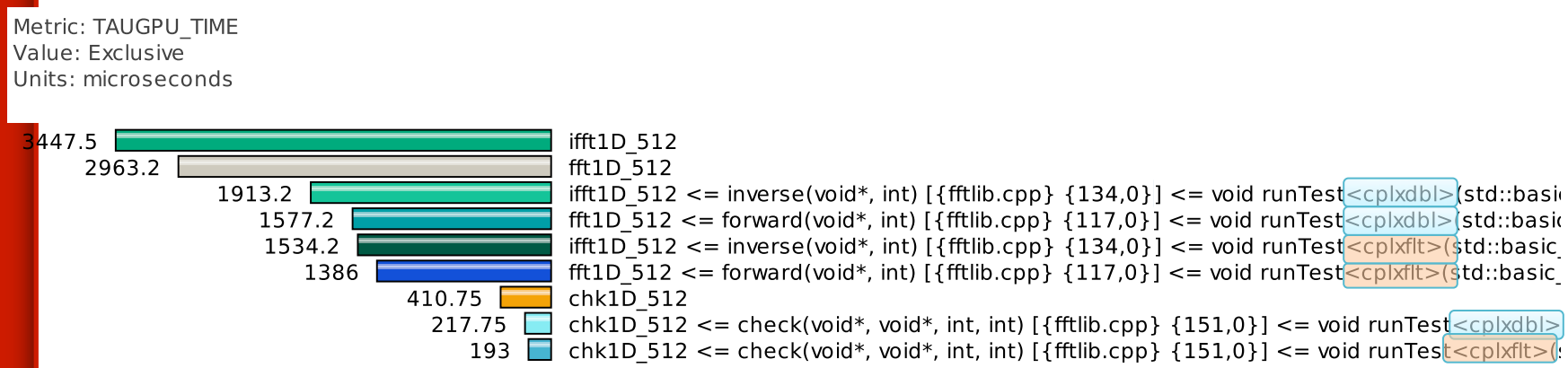


Identified a known overhead in GPU context creation

ParaTo

SHOC FFT Profile with Callsite Info

TAU is able to associate callsite context information with kernel launch so that different kernel calls can be distinguished



Each kernel (ifft1D_512, fft1D_512 and chk1D_512) is broken down by call-site, either during the single precession or double precession step.

Example: SHOC Stencil2D

Compute 2D, 9-point stencil

- Multiple GPUs using MPI
- CUDA and OpenCL versions

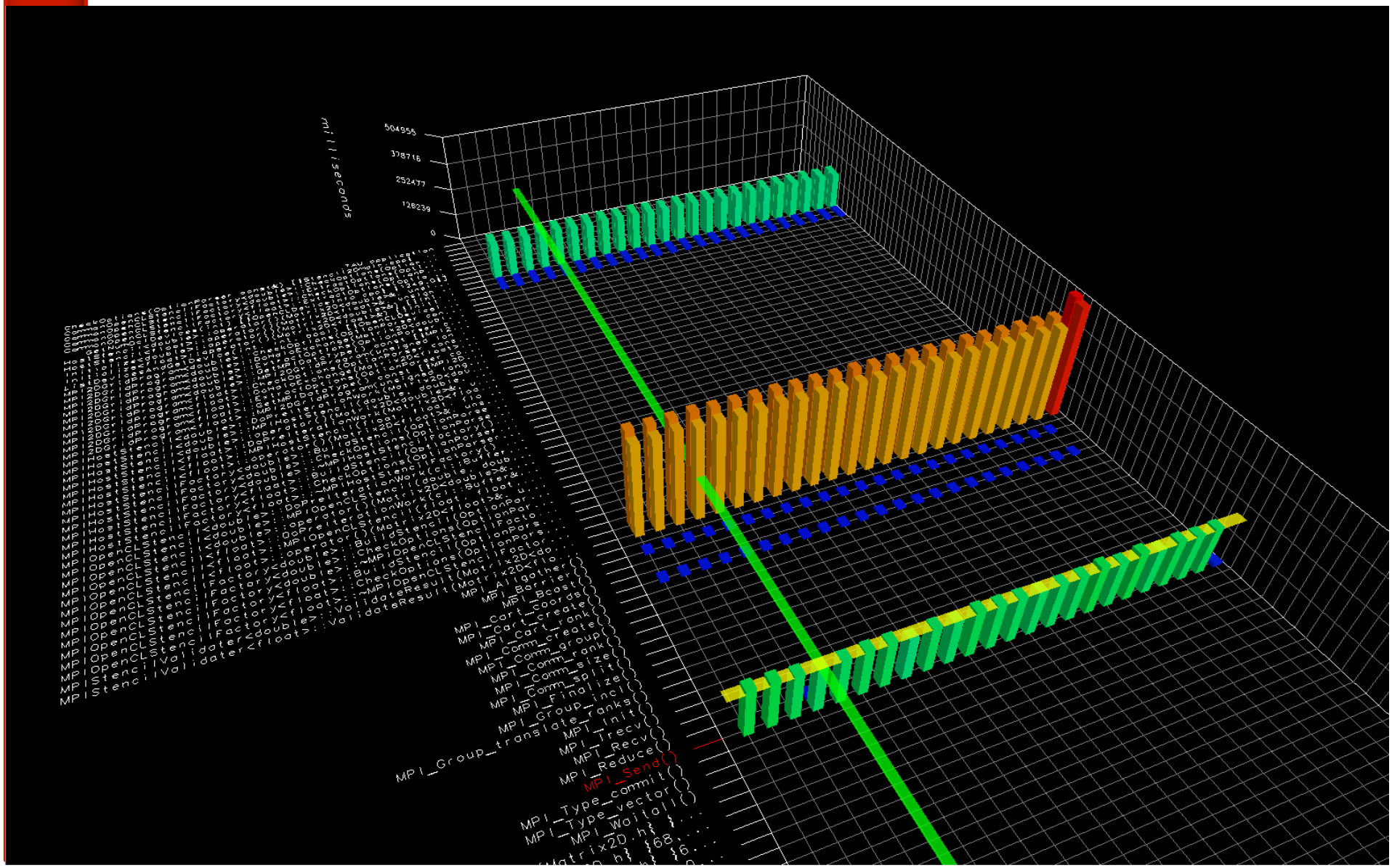
One Keeneland node with 3 GPUs

Eight Keeneland nodes with 24 GPUs

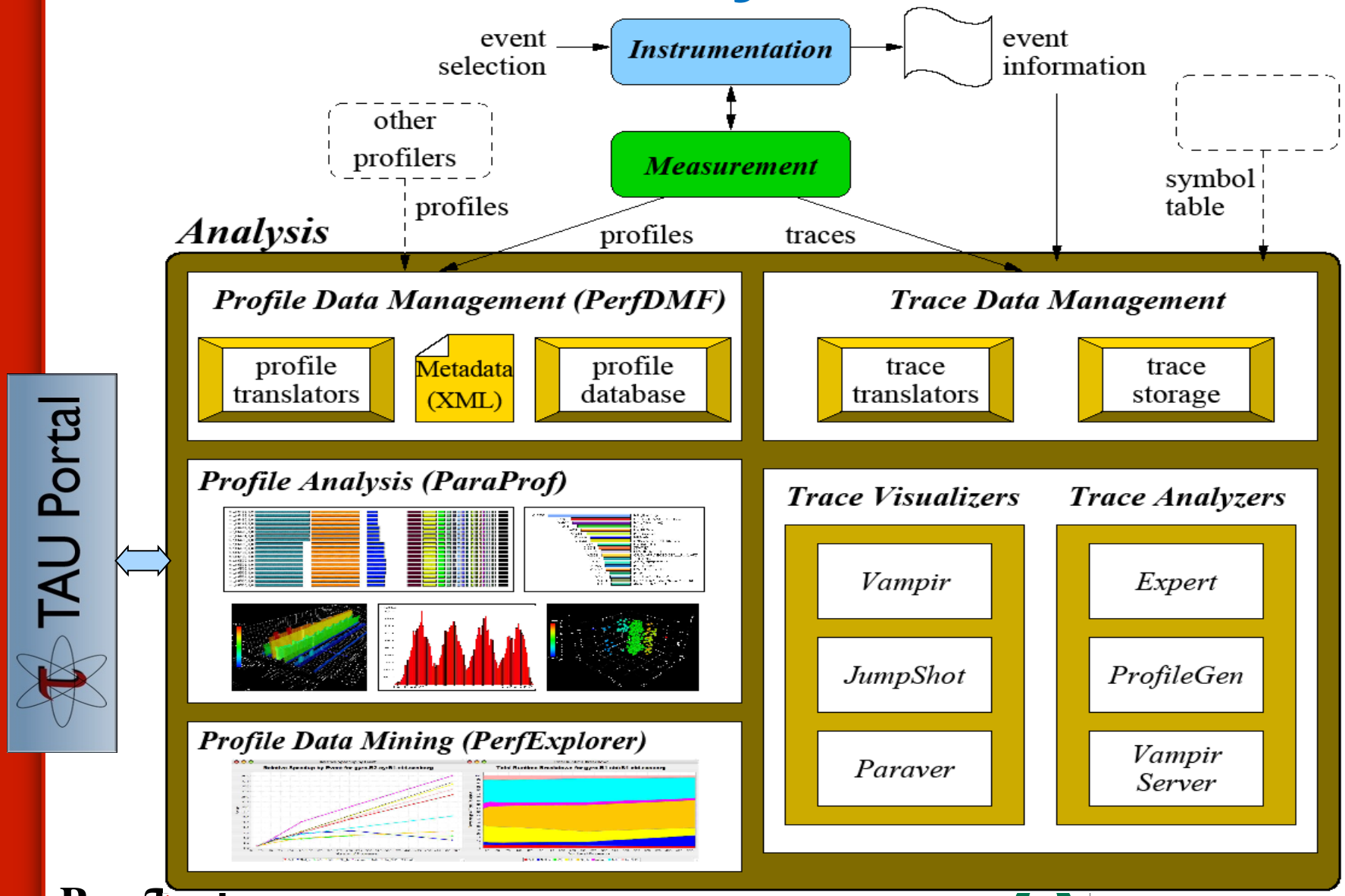
Performance profile and trace

- Application events
- Communication events
- Kernel execution

Stencil2D Parallel Profile



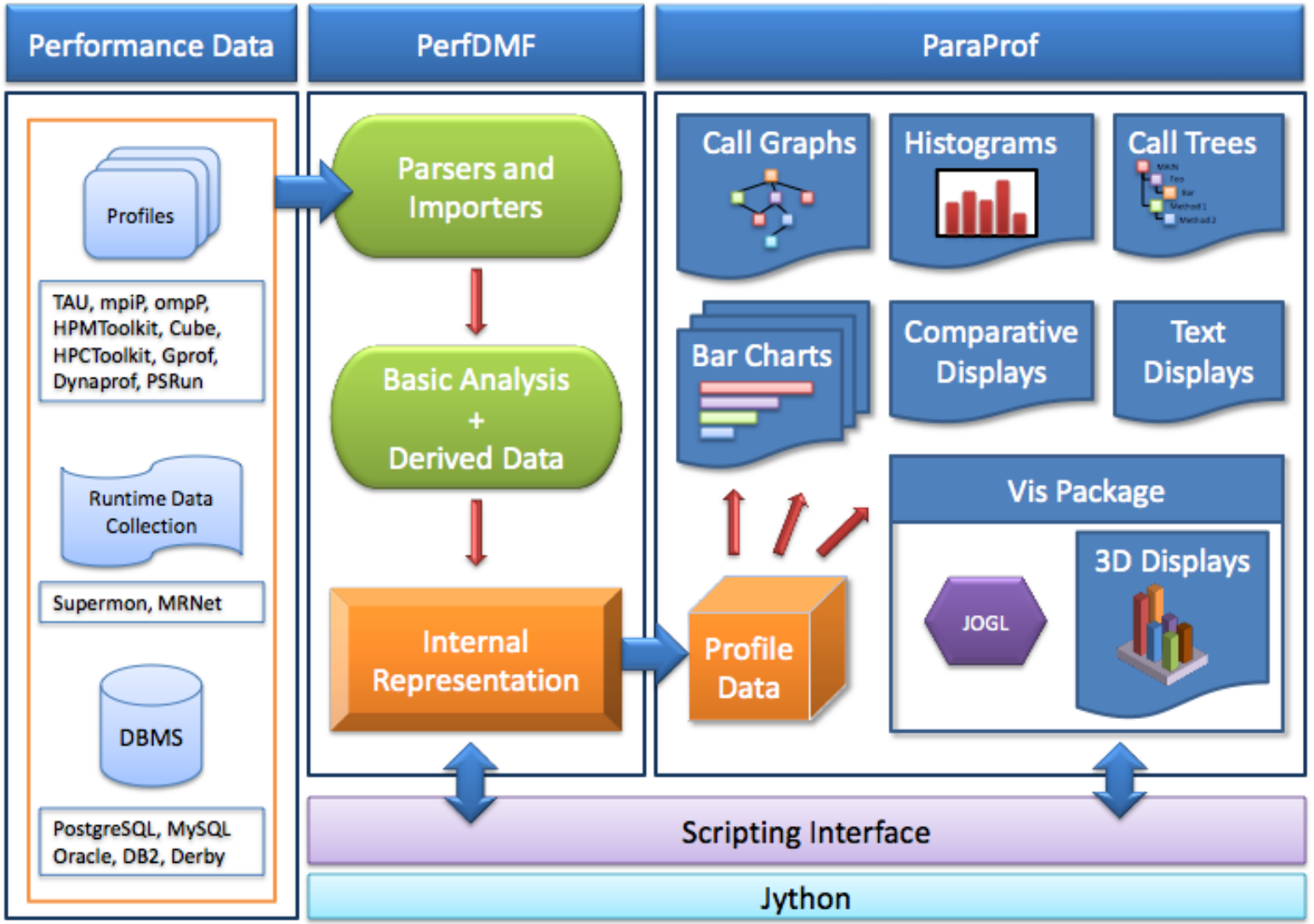
Performance Analysis



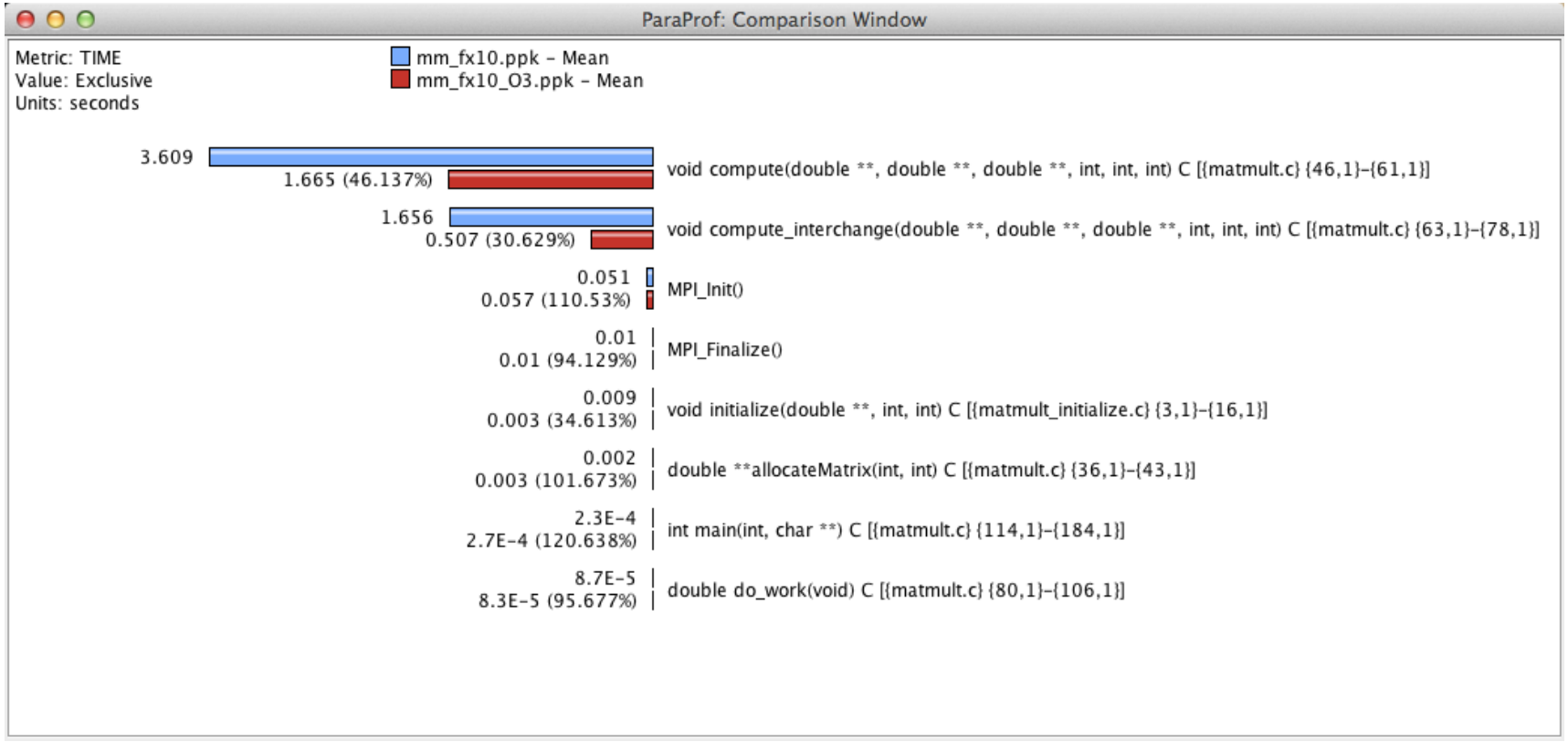
Performance Analysis

- **Parallel profile analysis (ParaProf)**
  - Java-based analysis and visualization tool
  - Support for large-scale parallel profiles
- **Parallel trace analysis**
  - Translation to ParaVer, SLOG-2, OTF formats
  - Integration with Vampir / Vampir Pro (TU Dresden)
  - Profile generation from trace data
- **Performance data management framework (PerfDMF)**
- **Online parallel analysis and visualization**
- **Integration with CUBE browser (Scalasca, FZJ)**

ParaProf Profile Analysis Framework

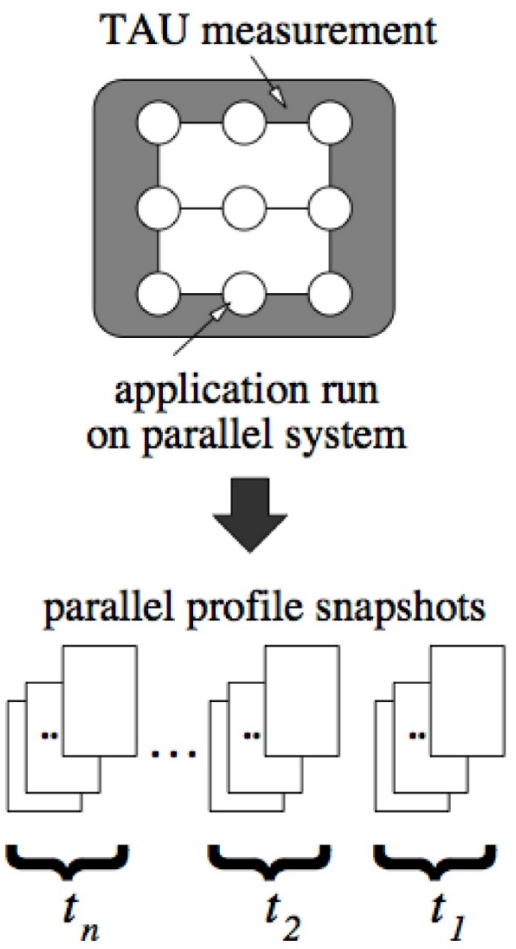


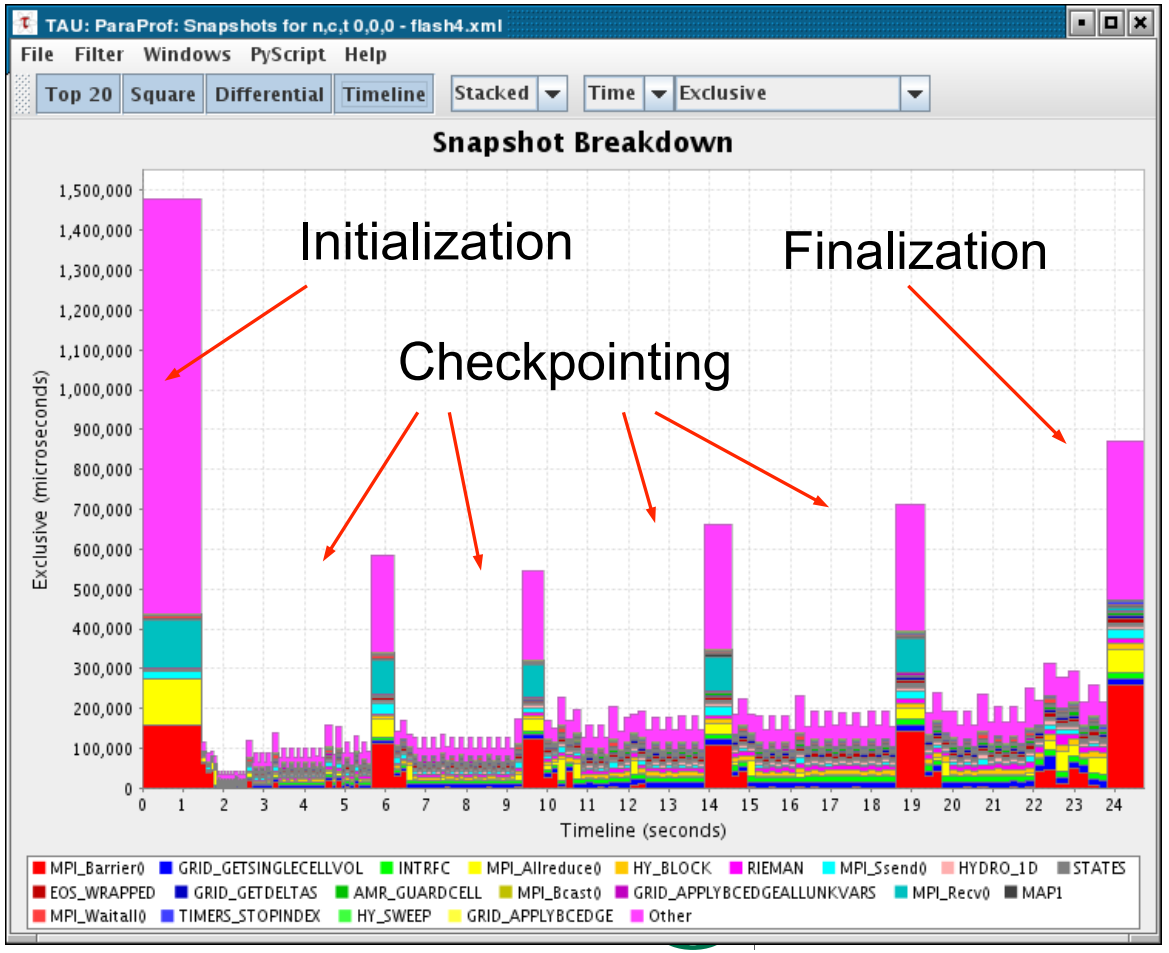
ParaProf Profile Comparison Window



Profile Snapshots in ParaProf

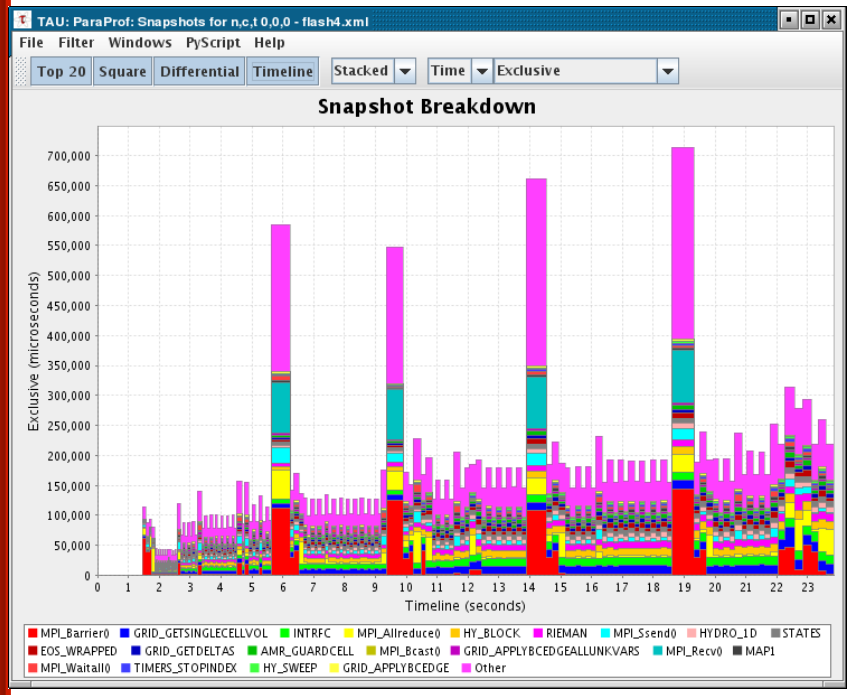
- Profile snapshots are profiles recorded at runtime
- Shows performance profile dynamics (all types allowed)



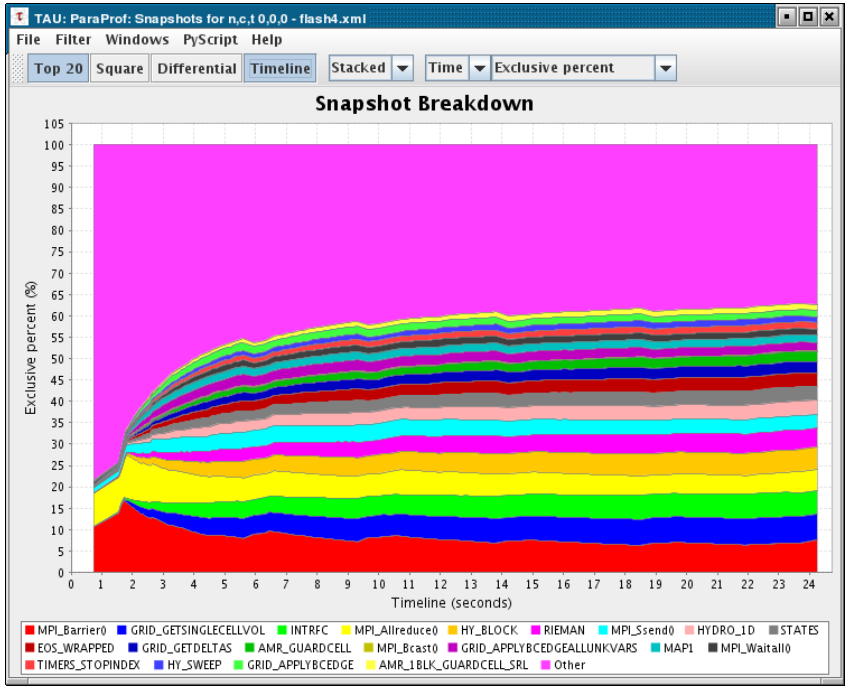


Profile Snapshot Views

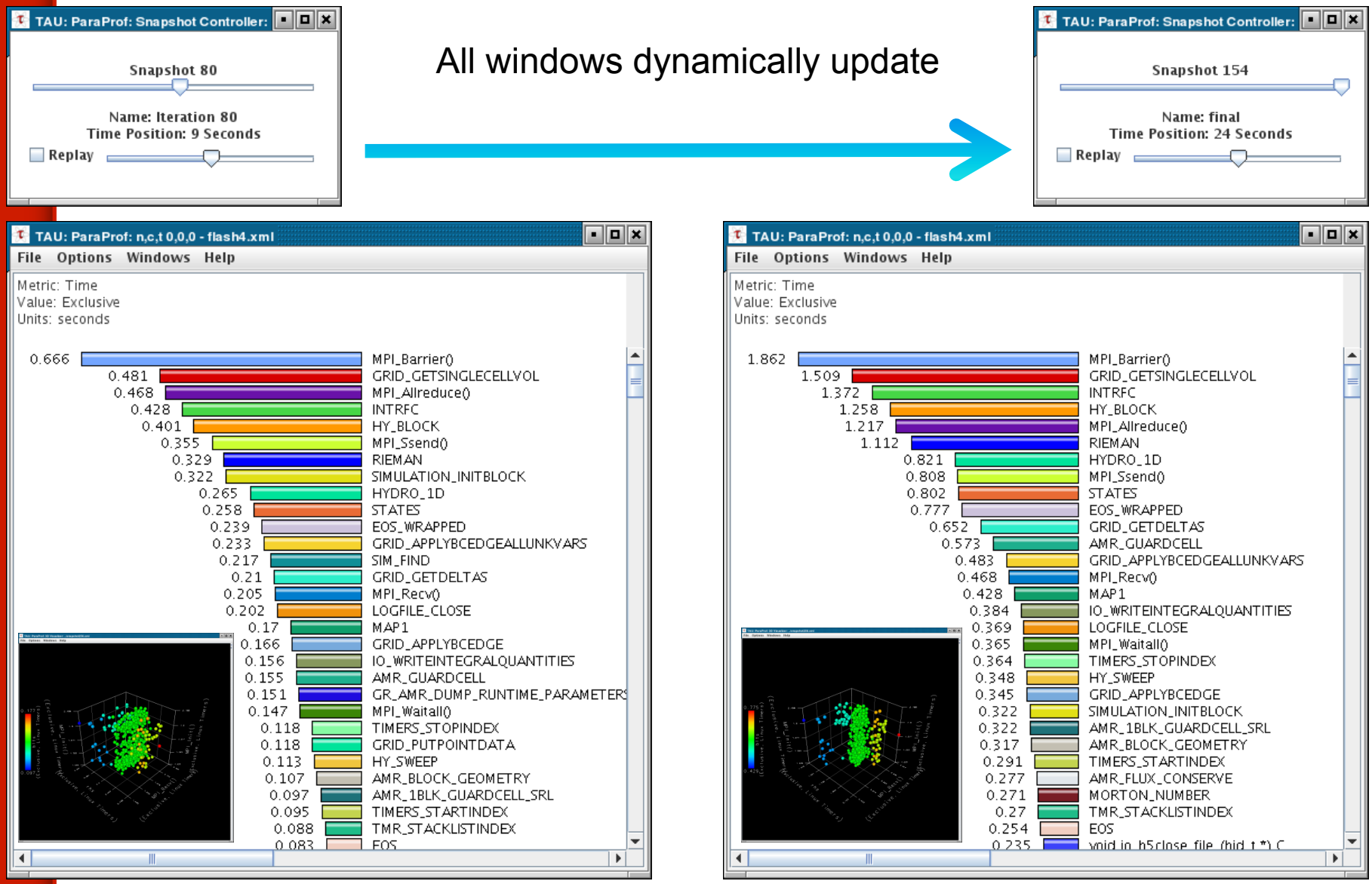
Percentage breakdown



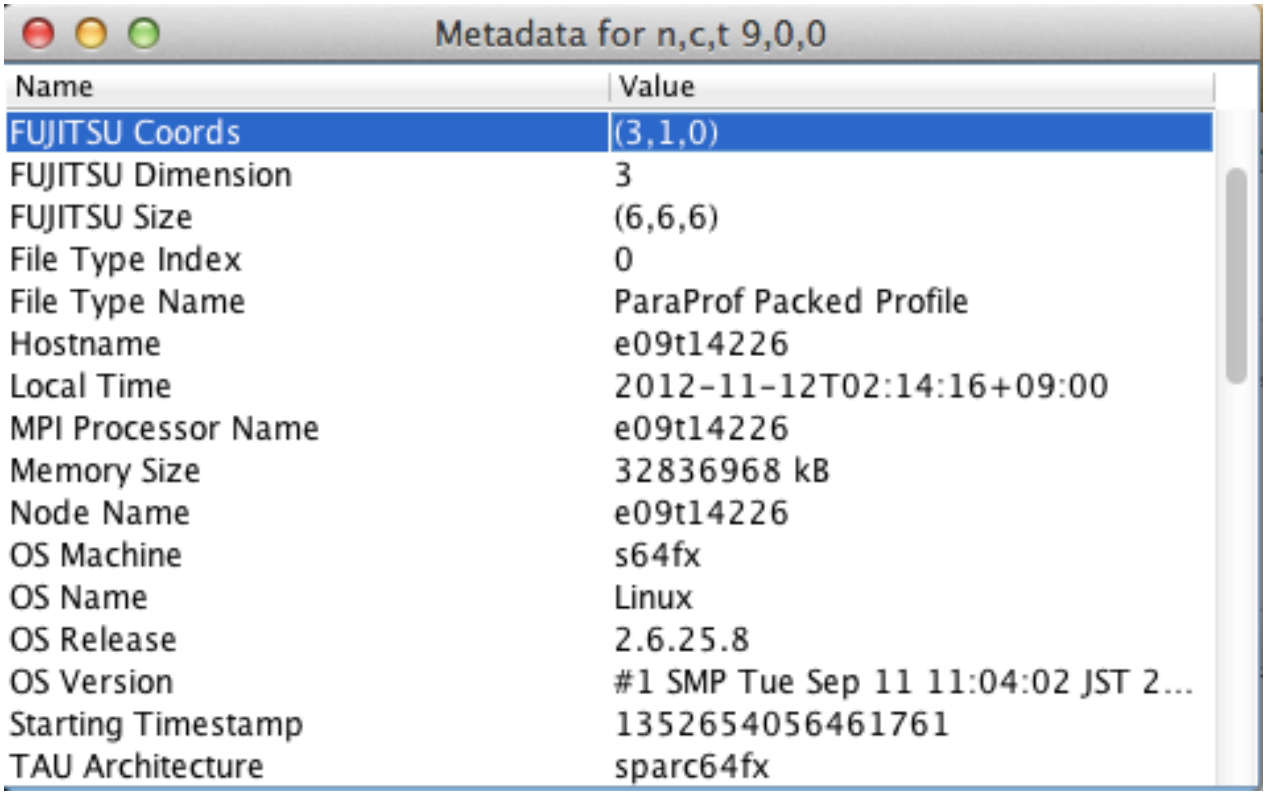
Only show main loop



Snapshot Replay in ParaProf

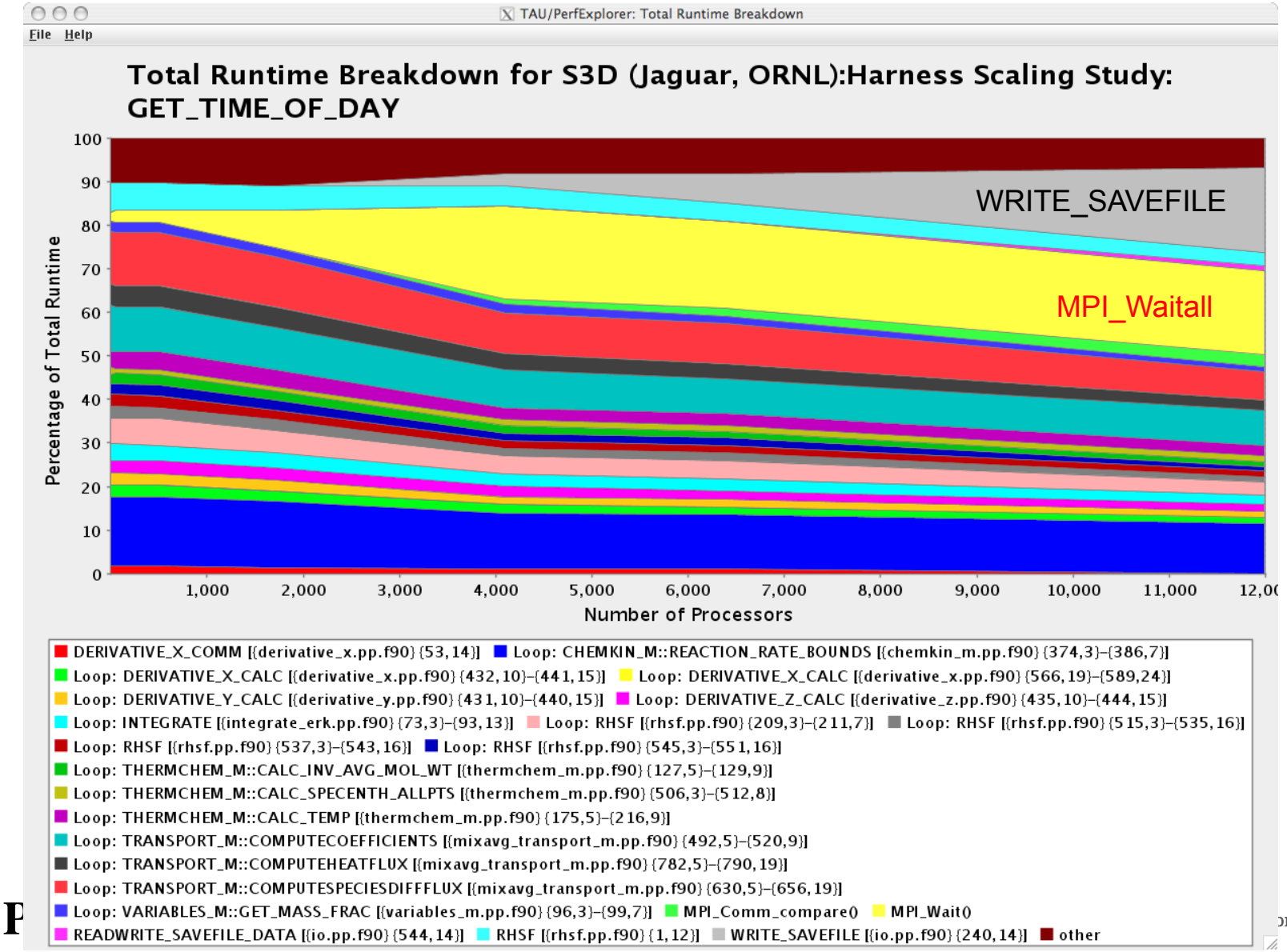


ParaProf Metadata Window

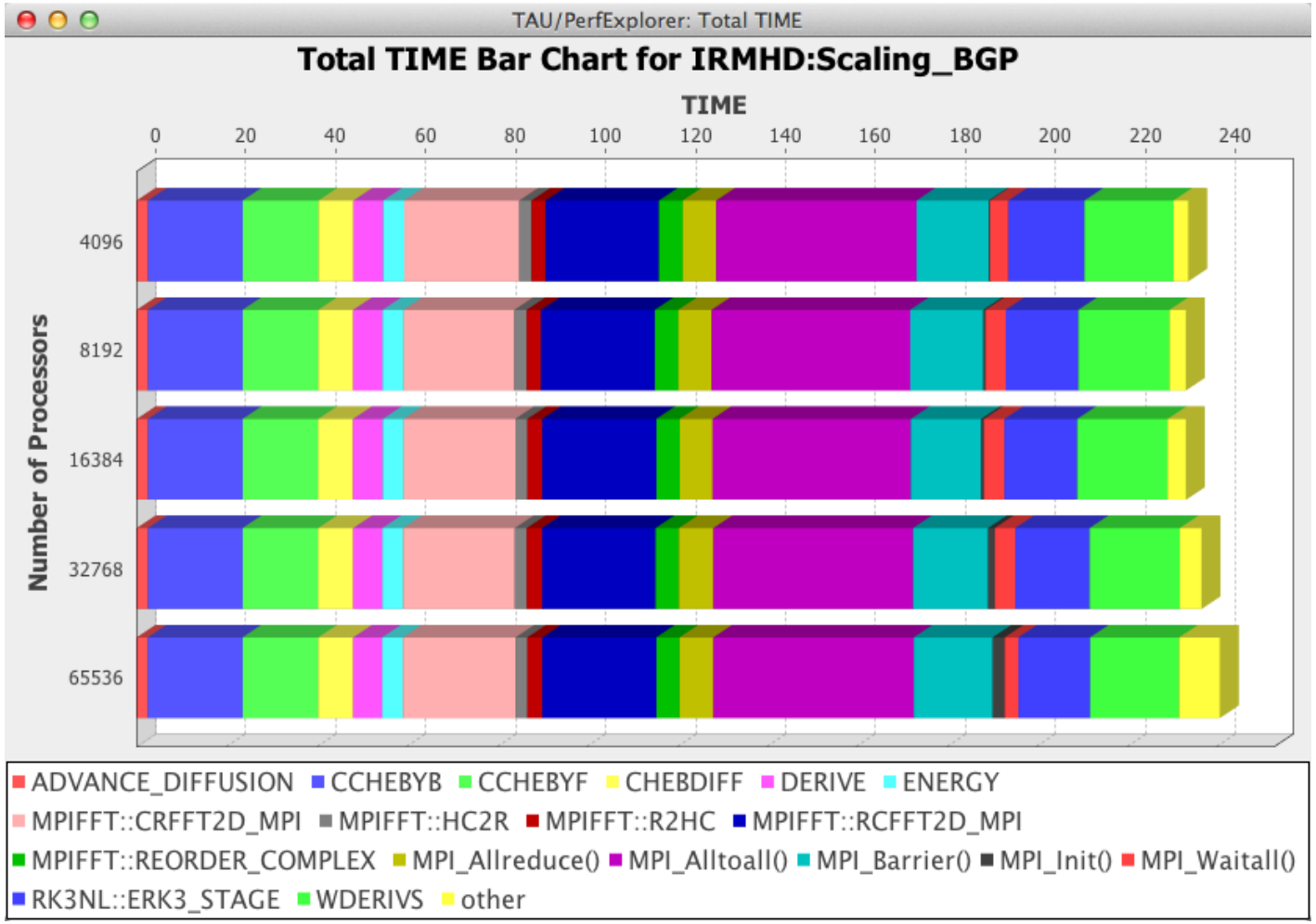


% pjsub –interact –L node="6x6x6"
% mpirun.lsf –n 216 ./a.out

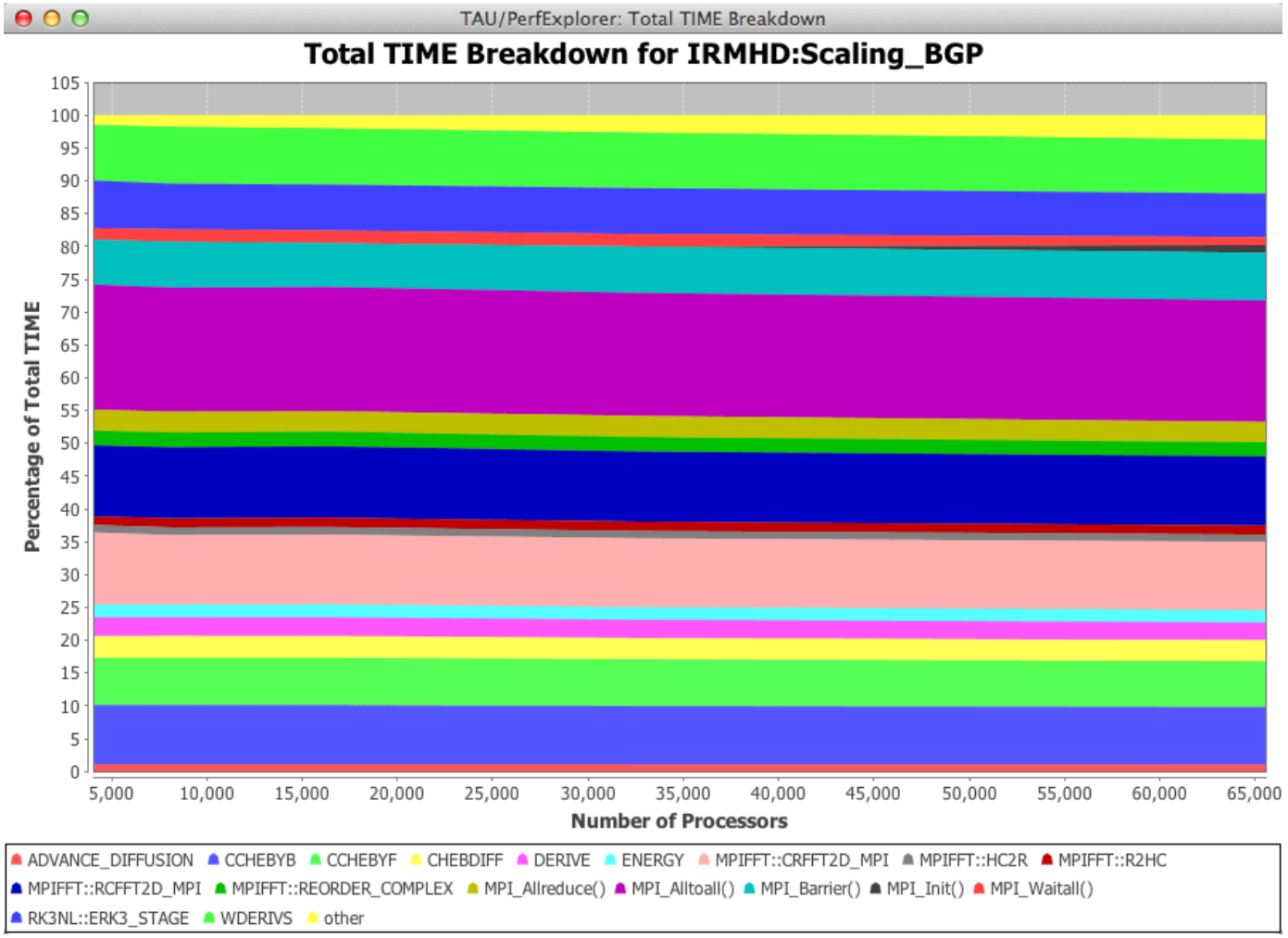
PerfExplorer – Runtime Breakdown



Evaluate Scalability

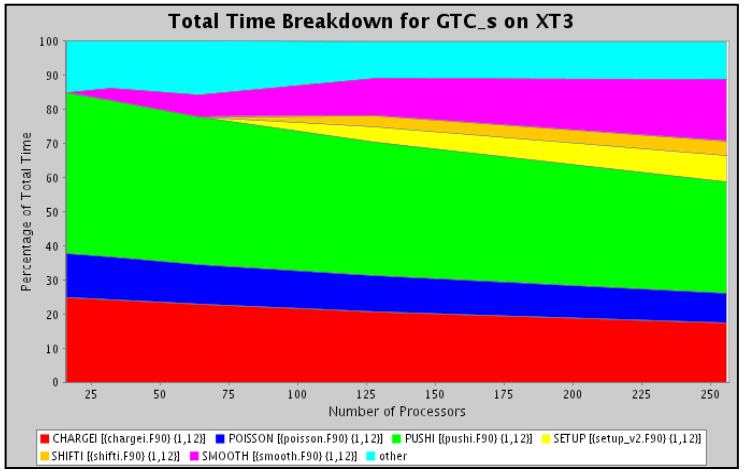


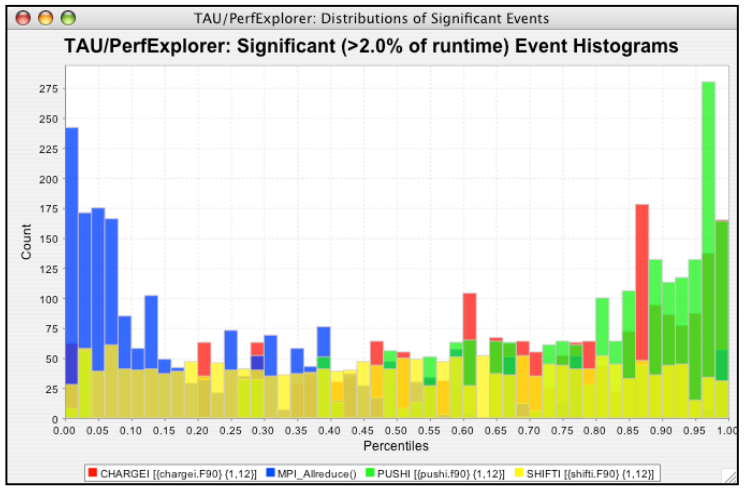
Runtime Breakdown



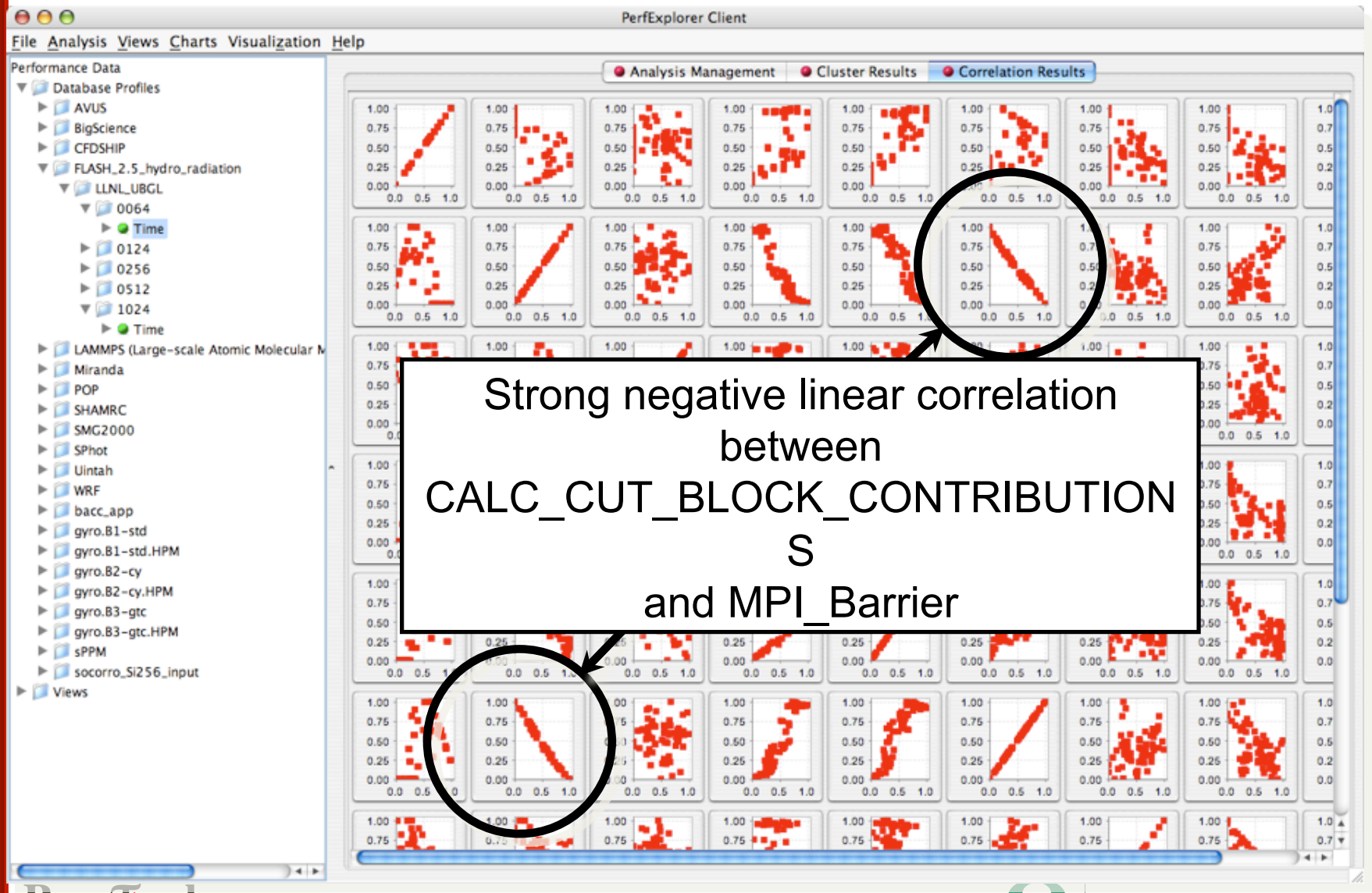
PerfExplorer – Relative Comparisons

- Total execution time
- Timesteps per second
- Relative efficiency
- Relative efficiency per event
- Relative speedup
- Relative speedup per event
- Group fraction of total
- Runtime breakdown
- Correlate events with total runtime
- Relative efficiency per phase
- Relative speedup per phase
- Distribution visualizations



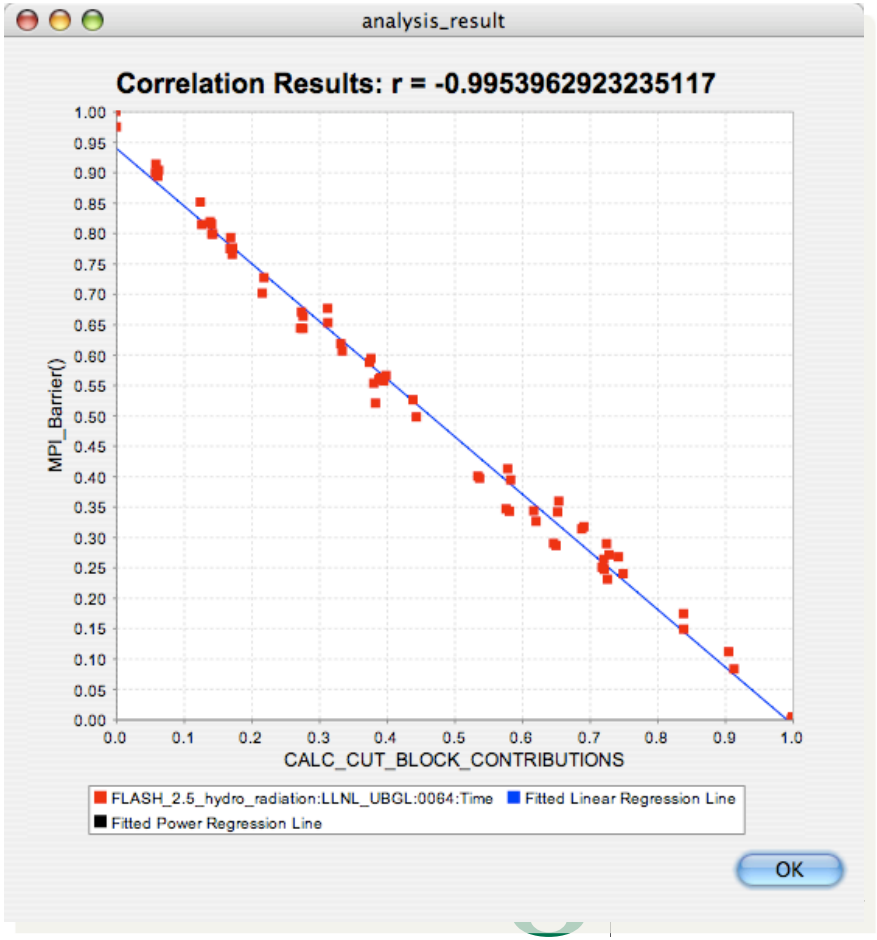


PerfExplorer – Correlation Analysis

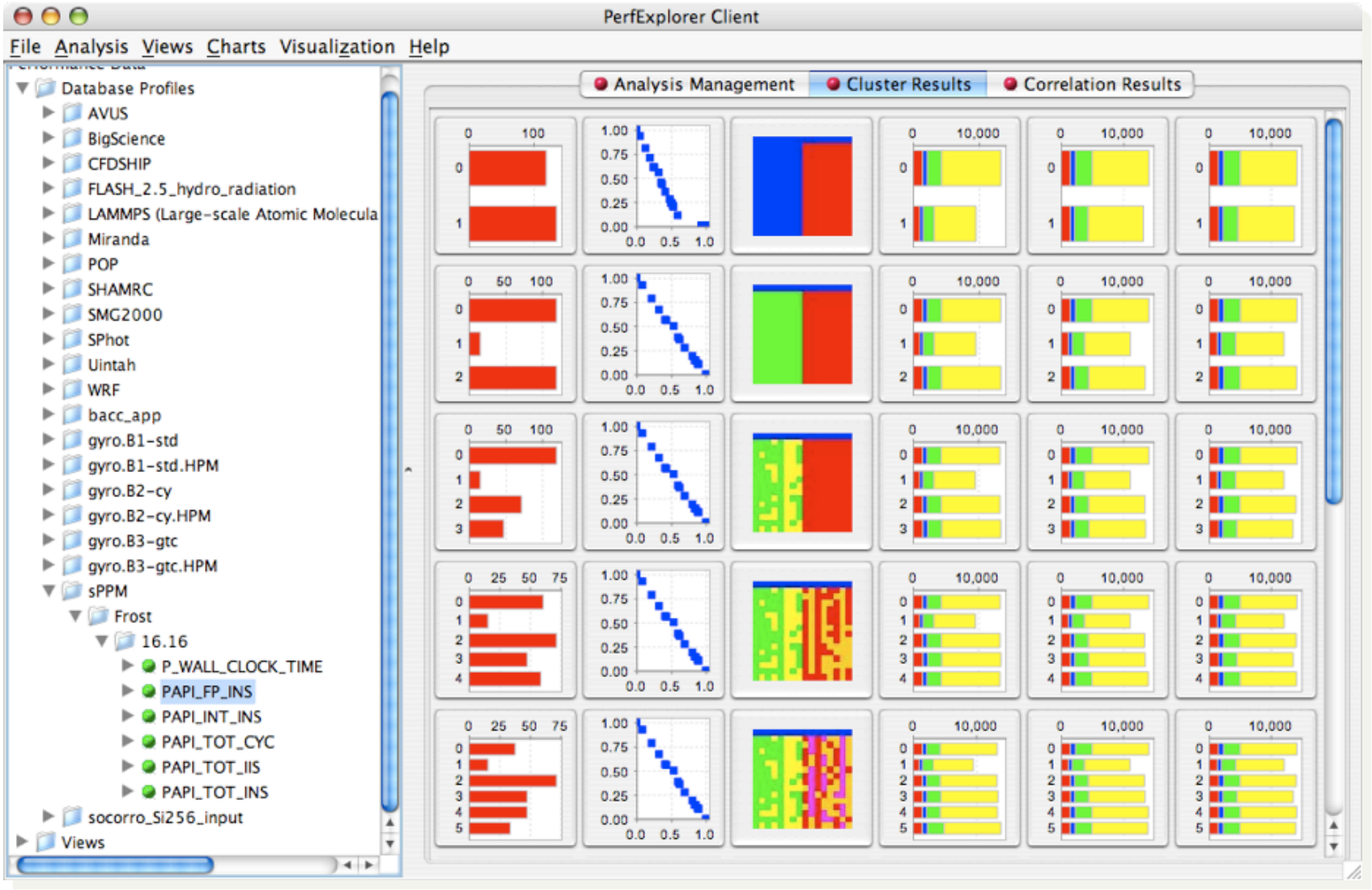


PerfExplorer – Correlation Analysis

-0.995 indicates strong, negative relationship. As CALC_CUT_BLOCK_CONTRIBUTIONS() increases in execution time, MPI_Barrier() decreases



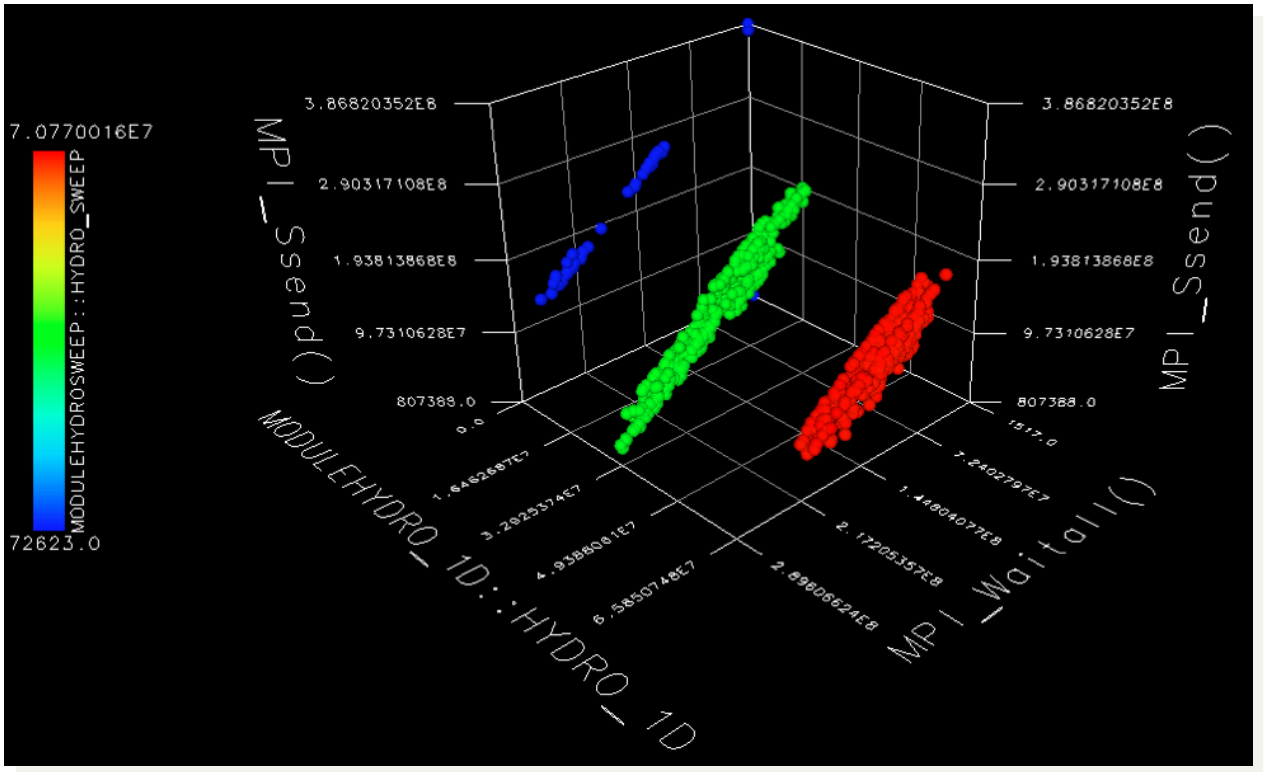
PerfExplorer – Cluster Analysis



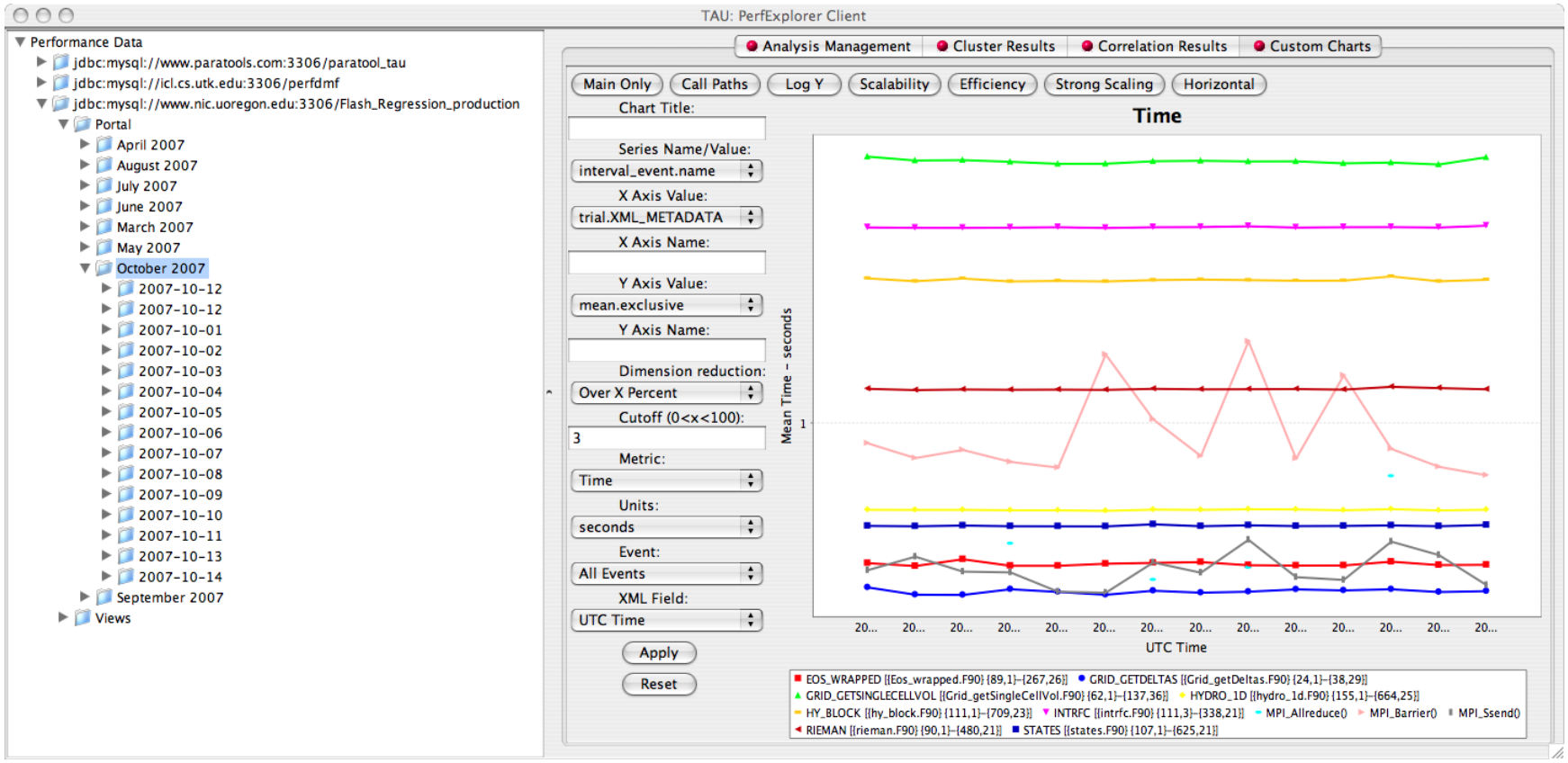
PerfExplorer – Cluster Analysis

Four significant events automatically selected

Clusters and correlations are visible



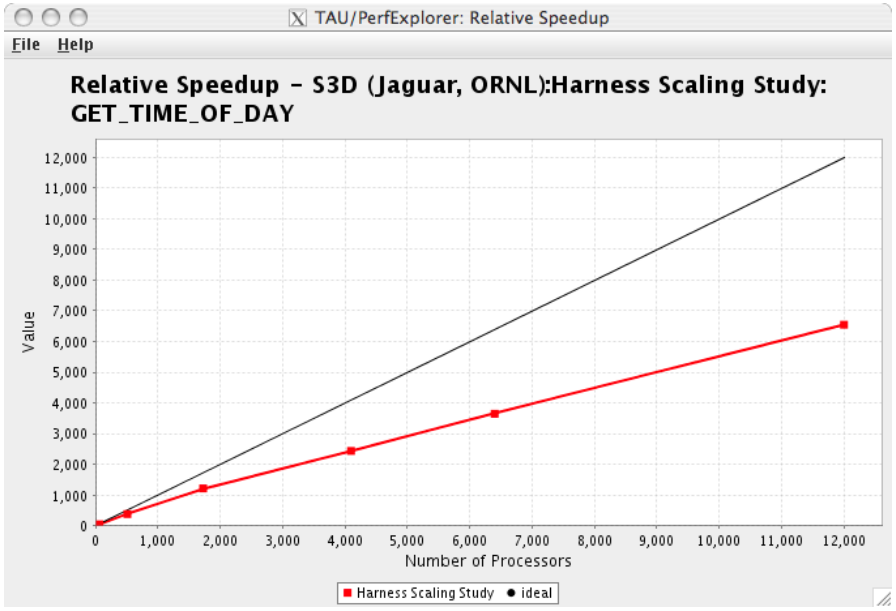
PerfExplorer – Performance Regression

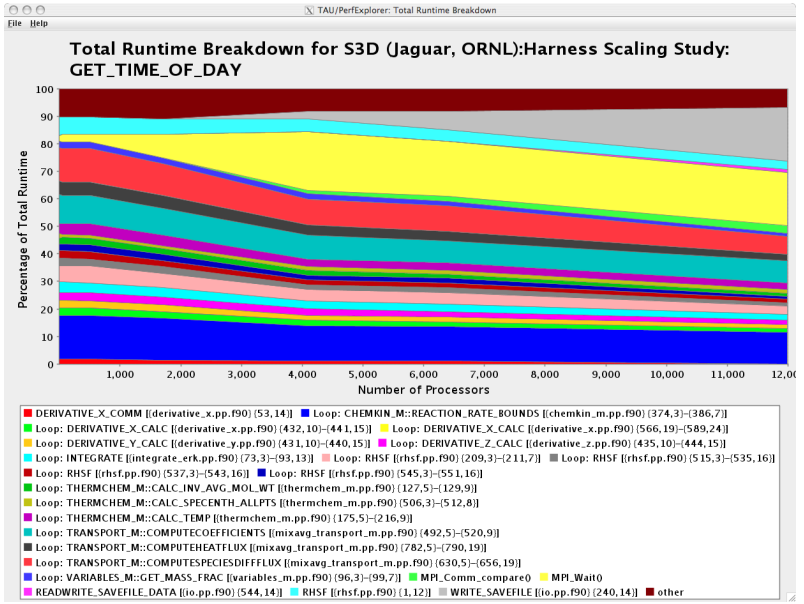


Evaluate Scalability

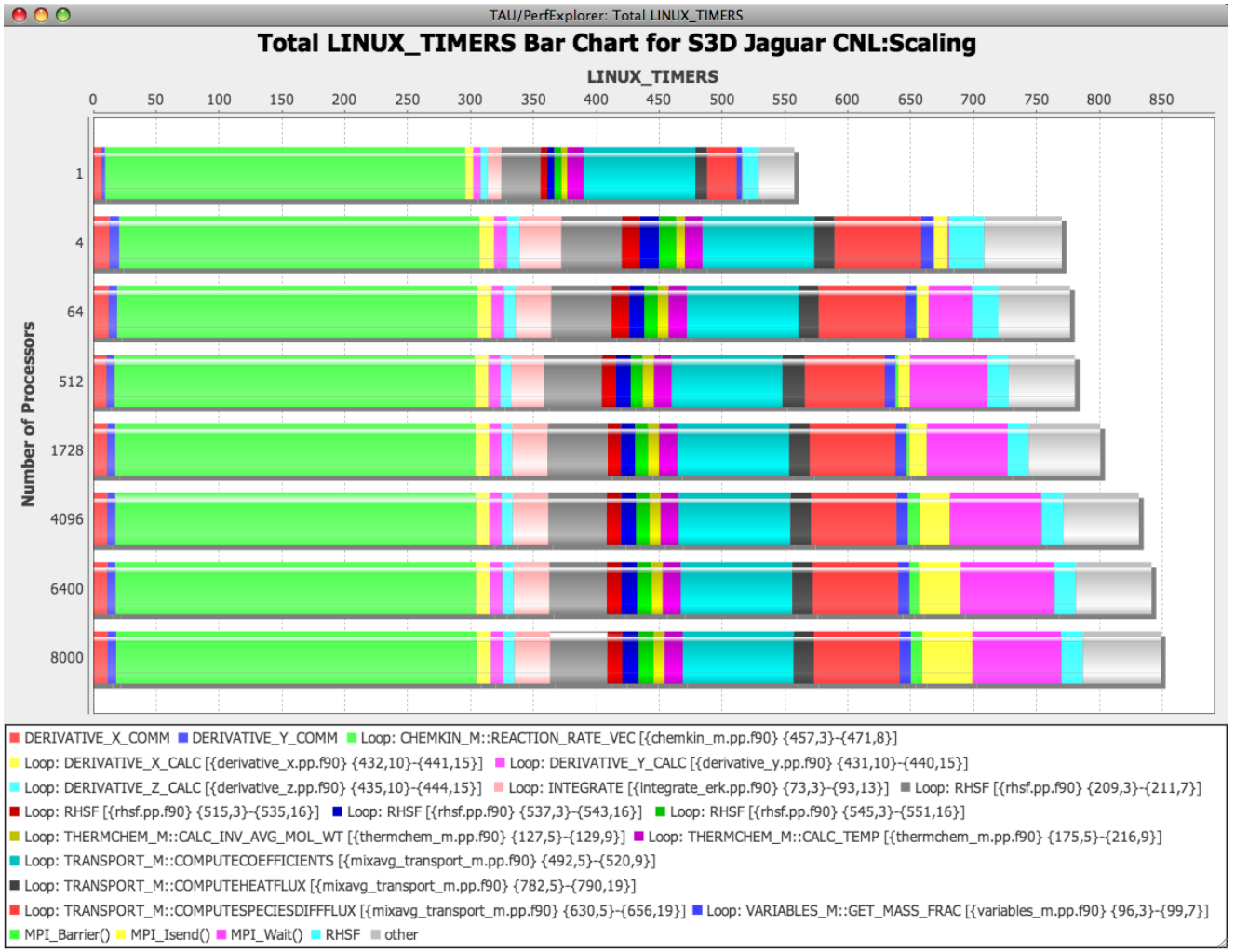
Goal: How does my application scale? What bottlenecks at what CPU counts?

Load profiles in PerfDMF database and examine with PerfExplorer

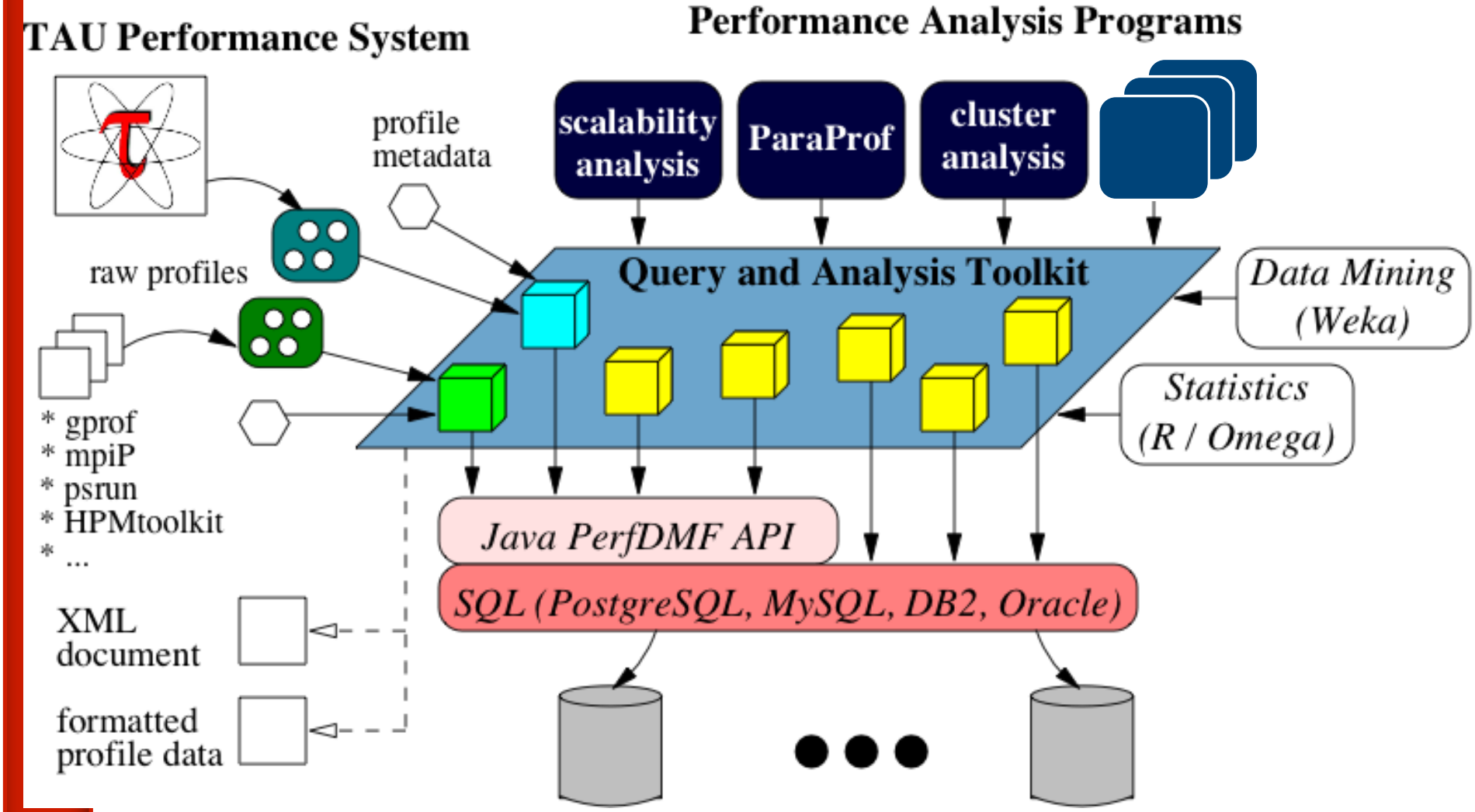




Usage Scenarios: Evaluate Scalability



TAUdb: Framework for Managing Performance Data



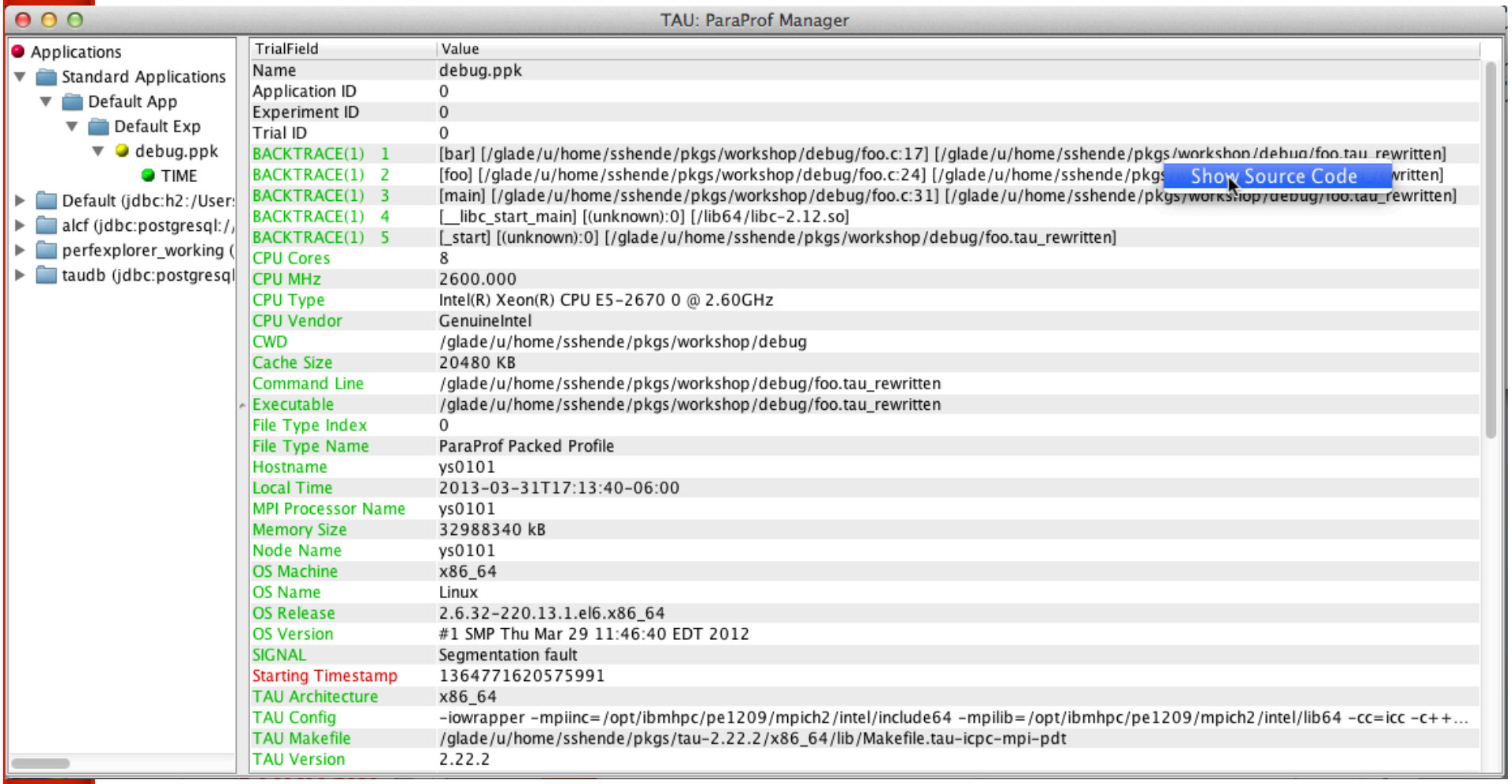
Evaluate Scalability using PerfExplorer Charts

```
% module load workshop tau
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt
% 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
% perfexplorer_configure
(Enter, Yes to load jars, schema, defaults)
% paraprof
(load each trial: DB -> Add Trial -> Type (Paraprof Packed
Profile) -> OK, OR use taudb_configure on the commandline)
% perfexplorer
(Charts -> Speedup)
```

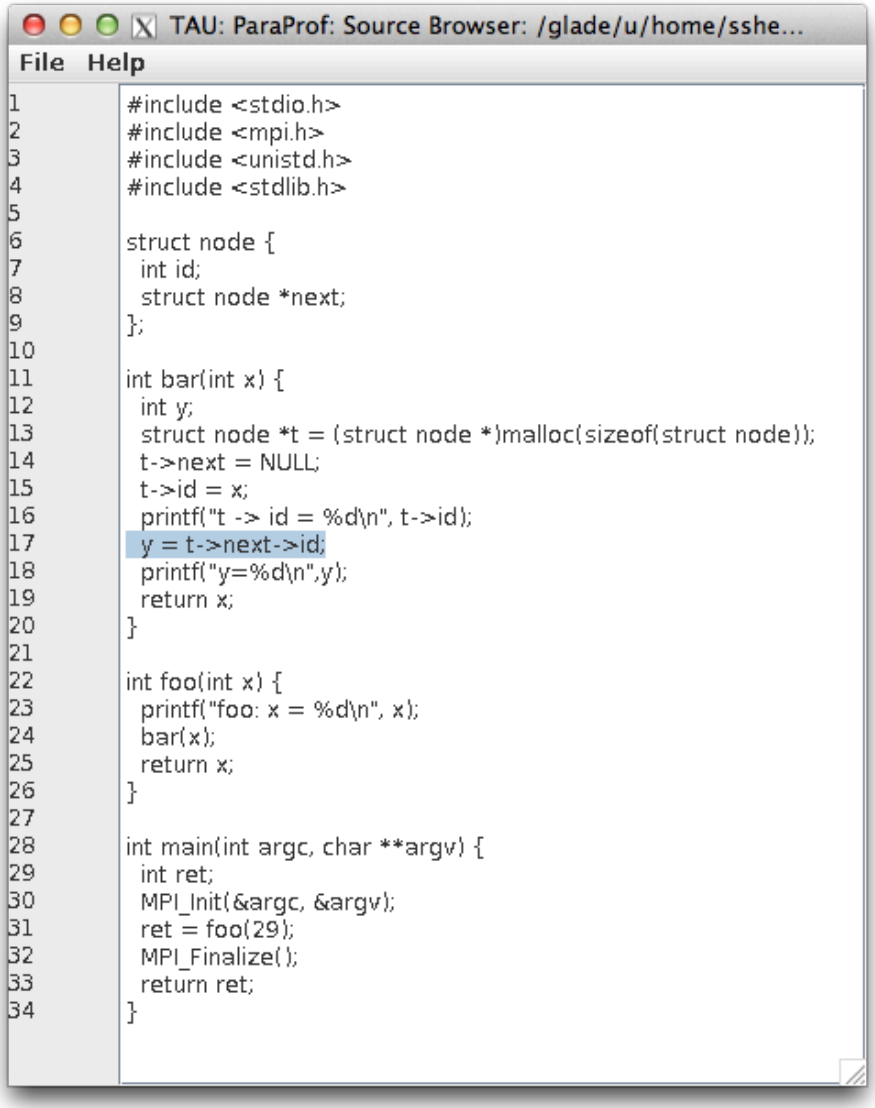
Multi-language Application Debugging

```
% module load workshop tau
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt
% export TAU_OPTIONS='-optMemDbg -optVerbose'
% make F90=tau_f90.sh CC=tau_cc.sh CXX=tau_cxx.sh
% bsub -Is -W 1:00 -n 8 -P SCSG0004 -q tutorial $SHELL

% export TAU_MEMDBG_PROTECT_ABOVE=1
% export TAU_MEMDBG_PROTECT_BELOW=1
% export TAU_MEMDBG_PROTECT_FREE=1
% mpirun.lsf ./matmult
% paraprof
```

Multi-language Application Debugging



Location of segmentation violation



Memory Leak Detection

```
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt
% export PATH=$TAU/../bin:$PATH
% export TAU_OPTIONS='-optMemDbg -optVerbose'
% make F90=tau_f90.sh CC=tau_cc.sh CXX=tau_cxx.sh
% bsub -Is -W 1:00 -n 8 -P SCSG0004 -q tutorial $SHELL

% export TAU_TRACK_MEMORY_LEAKS=1
% mpirun.lsf ./matmult
% paraprof
```

Multi-language Memory Leak Detection

TAU: ParaProf: Context Events for: node 0 - memleak.ppk

Name	Total	NumSamples	MaxValue	MinValue	MeanValue	Std. Dev.
Heap Allocate	5,000,033	2	5,000,001	32	2,500,016.5	2,499,984.5
Heap Allocate <file=simple.c, line=15>	180	3	80	48	60	14.236
Heap Allocate <file=simple.c, line=23>	180	1	180	180	180	0
Heap Free <file=simple.c, line=18>	80	1	80	80	80	0
Heap Free <file=simple.c, line=25>	180	1	180	180	180	0
Heap Memory Used (KB)	4,884.829	8	4,883.196	0.047	610.604	1,614.888
▼ int foo(int) C [{simple.c} {36,1}-{44,1}]						
▼ int bar(int) C [{simple.c} {7,1}-{28,1}]						
Heap Allocate <file=simple.c, line=23>	180	1	180	180	180	0
Heap Free <file=simple.c, line=25>	180	1	180	180	180	0
▼ int g(int) C [{simple.c} {30,1}-{34,1}]						
▼ int bar(int) C [{simple.c} {7,1}-{28,1}]						
Heap Allocate <file=simple.c, line=15>	180	3	80	48	60	14.236
Heap Free <file=simple.c, line=18>	80	1	80	80	80	0
MEMORY LEAK! Heap Allocate <file=simple.c, line=15>	100	2	52	48	50	2
▼ int main(int, char **) C [{simple.c} {45,1}-{55,1}]						
▼ MPI_Finalize()						
Heap Allocate	5,000,033	2	5,000,001	32	2,500,016.5	2,499,984.5
MEMORY LEAK! Heap Allocate	5,000,033	2	5,000,001	32	2,500,016.5	2,499,984.5

Support Acknowledgments

US Department of Energy (DOE)

- Office of Science contracts
- SciDAC, LBL contracts
- LLNL-LANL-SNL ASC/NNSA contract
- Battelle, PNNL contract
- ANL, ORNL contract

Department of Defense (DoD)

- PETTT, HPCMP

National Science Foundation (NSF)

- Glassbox, SI-2

University of Tennessee, Knoxville

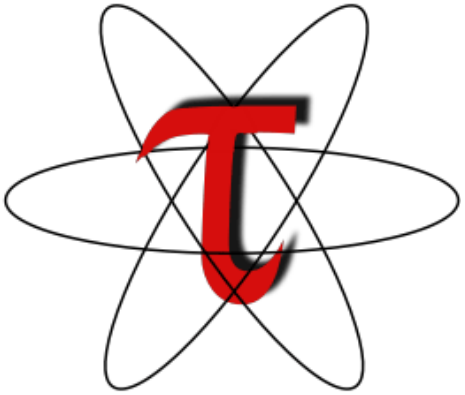
T.U. Dresden, GWT

Juelich Supercomputing Center

And a special thanks to UCAR!

UNIVERSITY OF OREGON

Download TAU from U. Oregon



http://tau.uoregon.edu

http://www.hpclinux.com [LiveDVD]

Free download, open source, BSD license

Reference

Using TAU on Yellowstone

- **Configuration of PDT:**
  - wget http://tau.uoregon.edu/pdt_lite.tgz
  - ./configure –prefix=<dir>; make ; make install
- **Configuration of TAU:**
  - wget http://tau.uoregon.edu/tau.tgz
  - ./configure –pdt=<dir> -c++=icpc –cc=icc –fortran=intel -prefix=<dir> -papi=<dir>
  - make install
- **Compiling:**
  - export TAU_MAKEFILE=<taudir>/x86_64/lib/Makefile.tau-icpc-papi-mpi-pdt
  - make CC=tau_cc.sh CXX=tau_cxx.sh F90=tau_f90.sh

Compile-Time Options

Optional parameters for the TAU_OPTIONS environment variable:
% tau_compiler.sh

-optVerbose	Turn on verbose debugging messages
-optCompInst	Use compiler based instrumentation
-optNoCompInst	Do not revert to compiler instrumentation if source instrumentation fails.
-optTrackIO	Wrap POSIX I/O call and calculates vol/bw of I/O operations (Requires TAU to be configured with *–iowrapper*)
-optMemDbg	Runtime bounds checking (see TAU_MEMDBG_* env vars)
-optKeepFiles	Does not remove intermediate .pdb and .inst.* files
-optPreProcess	Preprocess sources (OpenMP, Fortran) before instrumentation
-optTauSelectFile=”<file>"	Specify selective instrumentation file for *tau_instrumentor*
-optTauWrapFile=”<file>"	Specify path to *link_options.tau* generated by *tau_gen_wrapper*
-optHeaderInst	Enable Instrumentation of headers
-optTrackUPCR	Track UPC runtime layer routines (used with tau_upc.sh)
-optLinking=""	Options passed to the linker. Typically $(TAU_MPI_FLIBS) $(TAU_LIBS) $(TAU_CXXLIBS)
-optCompile=""	Options passed to the compiler. Typically $(TAU_MPI_INCLUDE) $(TAU_INCLUDE) $(TAU_DEFS)
-optPdtF95Opts=""	Add options for Fortran parser in PDT (f95parse/gfparse) …

Runtime Environment Variables

Environment Variable	Default	Description
TAU_TRACE	0	Setting to 1 turns on tracing
TAU_CALLPATH	0	Setting to 1 turns on callpath profiling
TAU_TRACK_MEMORY_LEAKS	0	Setting to 1 turns on leak detection (for use with –optMemDbg or tau_exec)
TAU_MEMDBG_PROTECT_ABOVE	0	Setting to 1 turns on bounds checking for dynamically allocated arrays. (Use with –optMemDbg or tau_exec –memory_debug).
TAU_CALLPATH_DEPTH	2	Specifies depth of callpath. Setting to 0 generates no callpath or routine information, setting to 1 generates flat profile and context events have just parent information (e.g., Heap Entry: foo)
TAU_TRACK_IO_PARAMS	0	Setting to 1 with –optTrackIO or tau_exec –io captures arguments of I/O calls
TAU_TRACK_SIGNALS	0	Setting to 1 generate debugging callstack info when a program crashes
TAU_COMM_MATRIX	0	Setting to 1 generates communication matrix display using context events
TAU_THROTTLE	1	Setting to 0 turns off throttling. Enabled by default to remove instrumentation in lightweight routines that are called frequently
TAU_THROTTLE_NUMCALLS	100000	Specifies the number of calls before testing for throttling
TAU_THROTTLE_PERCALL	10	Specifies value in microseconds. Throttle a routine if it is called over 100000 times and takes less than 10 usec of inclusive time per call
TAU_COMPENSATE	0	Setting to 1 enables runtime compensation of instrumentation overhead
TAU_PROFILE_FORMAT	Profile	Setting to “merged” generates a single file. “snapshot” generates xml format
TAU_METRICS	TIME	Setting to a comma separated list generates other metrics. (e.g., TIME:P_VIRTUAL_TIME:PAPI_FP_INS:PAPI_NATIVE_<event>\\:<subevent>)

Compiling Fortran Codes with TAU

If your Fortran code uses free format in .f files (fixed is default for .f), you may use:

% export TAU_OPTIONS= '-optPdtF95Opts="-R free" -optVerbose '

To use the compiler based instrumentation instead of PDT (source-based):

% export TAU_OPTIONS= '-optCompInst -optVerbose'

If your Fortran code uses C preprocessor directives (#include, #ifdef, #endif):

% export TAU_OPTIONS= '-optPreProcess -optVerbose -optDetectMemoryLeaks'

To use an instrumentation specification file:

% export TAU_OPTIONS= '-optTauSelectFile=select.tau -optVerbose -optPreProcess'

% cat select.tau

```
BEGIN_EXCLUDE_LIST
FOO
END_EXCLUDE_LIST

BEGIN_INSTRUMENT_SECTION
loops  routine="#"
# this statement instruments all outer loops in all routines. # is wildcard as well as comment in first column.
END_INSTRUMENT_SECTION
```