TAU Performance System

Workshop on Petascale Architectures and Performance Strategies

4-5pm, Tuesday, July 24, 2007, Snowbird, UT

Sameer S. Shende

sameer@cs.uoregon.edu

http://www.cs.uoregon.edu/research/tau

Performance Research Laboratory

University of Oregon

UNIVERSITY
OF OREGON

Acknowledgements

- Dr. Allen D. Malony, Professor
- Alan Morris, Senior software engineer
- Wyatt Spear, Software engineer
- Scott Biersdorff, Software engineer
- Kevin Huck, Ph.D. student
- Aroon Nataraj, Ph.D. student
- Brad Davidson, Systems administrator

Outline

- Overview of features
- Instrumentation
- Measurement
- Analysis tools
  - Parallel profile analysis (ParaProf)
  - Performance data management (PerfDMF)
  - Performance data mining (PerfExplorer)
- Application examples
- Kernel monitoring and KTAU

Performance Evaluation

- Profiling
  - Presents summary statistics of performance metrics
    - number of times a routine was invoked
    - exclusive, inclusive time/hpm counts spent executing it
    - number of instrumented child routines invoked, etc.
    - structure of invocations (calltrees/callgraphs)
    - memory, message communication sizes also tracked
- Tracing
  - Presents when and where events took place along a global timeline
    - timestamped log of events
    - message communication events (sends/receives) are tracked
      - shows when and where messages were sent
    - large volume of performance data generated leads to more perturbation in the program

Definitions – Profiling

- Profiling
  - Recording of summary information during execution
    - inclusive, exclusive time, # calls, hardware statistics, …
  - Reflects performance behavior of program entities
    - functions, loops, basic blocks
    - user-defined "semantic" entities
  - Very good for low-cost performance assessment
  - Helps to expose performance bottlenecks and hotspots
  - Implemented through
    - sampling: periodic OS interrupts or hardware counter traps
    - instrumentation: direct insertion of measurement code

Definitions – Tracing

- Tracing
  - Recording of information about significant points (events) during program execution
    - entering/exiting code region (function, loop, block, …)
    - thread/process interactions (e.g., send/receive message)
  - Save information in event record
    - timestamp
    - CPU identifier, thread identifier
    - Event type and event-specific information
  - Event trace is a time-sequenced stream of event records
  - Can be used to reconstruct dynamic program behavior
  - Typically requires code instrumentation

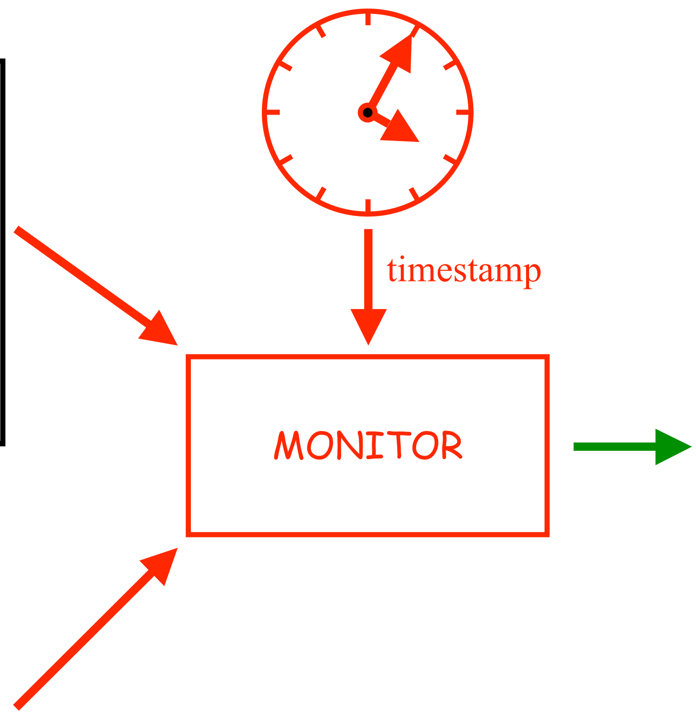
Event Tracing: Instrumentation, Monitor, Trace

CPU A:

```
void master {
  trace(ENTER, 1);
  ...
  trace(SEND, B);
  send(B, tag, buf);
  ...
  trace(EXIT, 1);
}
```

CPU B:

```
void worker {
  trace(ENTER, 2);
  ...
  recv(A, tag, buf);
  trace(RECV, A);
  ...
  trace(EXIT, 2);
}
```



Event definition

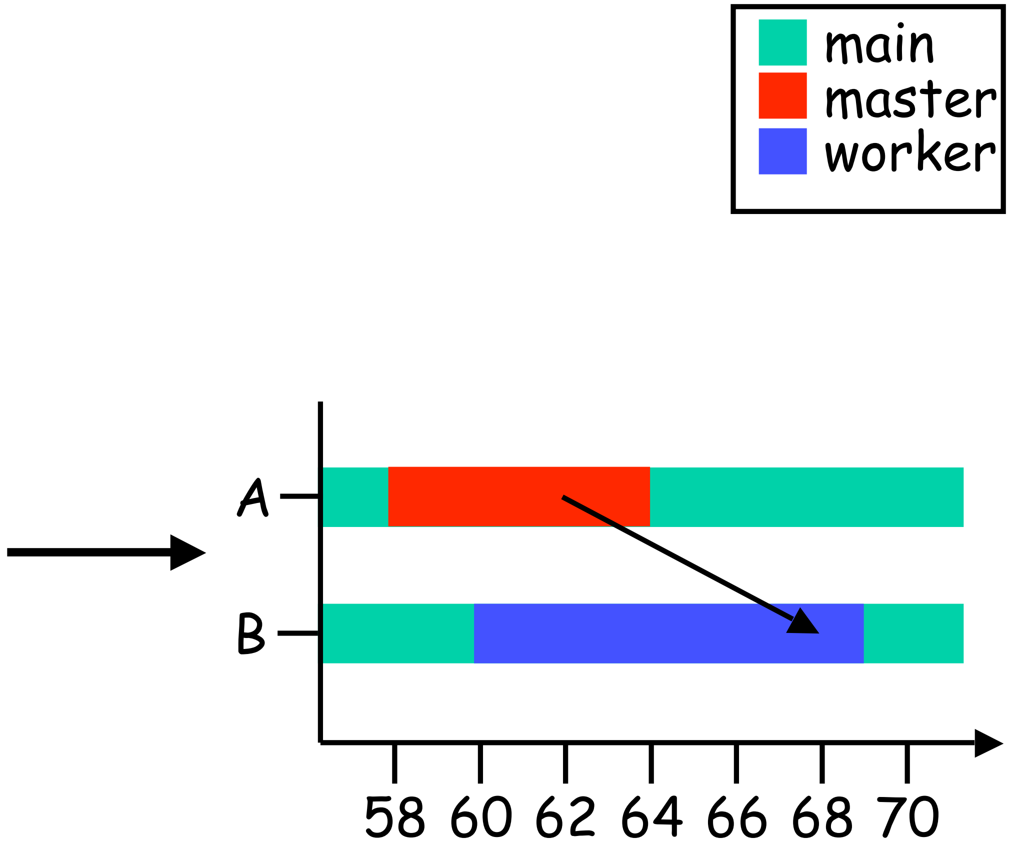
1	master
2	worker
3	...

...			
58	A	ENTER	1
60	B	ENTER	2
62	A	SEND	B
64	A	EXIT	1
68	B	RECV	A
69	B	EXIT	2
...			

Event Tracing: "Timeline" Visualization

1	master
2	worker
3	...

...			
58	A	ENTER	1
60	B	ENTER	2
62	A	SEND	B
64	A	EXIT	1
68	B	RECV	A
69	B	EXIT	2
...			



Steps of Performance Evaluation

- Collect basic routine-level timing profile to determine where most time is being spent
- Collect routine-level hardware counter data to determine types of performance problems
- Collect callpath profiles to determine sequence of events causing performance problems
- Conduct finer-grained profiling and/or tracing to pinpoint performance bottlenecks
  - Loop-level profiling with hardware counters
  - Tracing of communication operations

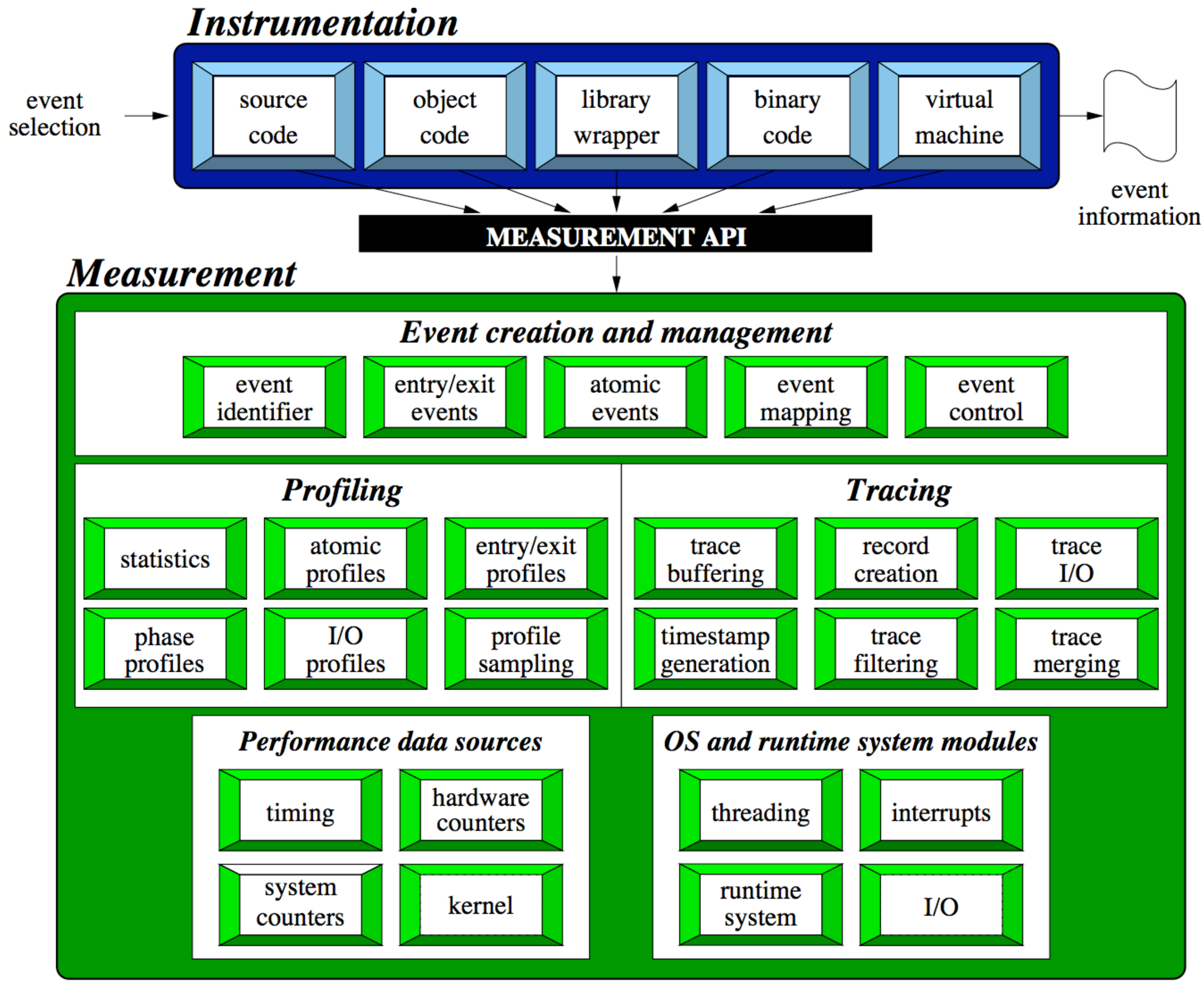
TAU Performance System

- ***T**uning and **A**nalysis **U**tilities (15+ year project effort)*
- **Performance system framework for HPC systems**
  - Integrated, scalable, flexible, and parallel
- Targets a general complex system computation model
  - *Entities*: nodes / contexts / threads
  - *Multi-level*: system / software / parallelism
  - Measurement and analysis abstraction
- **Integrated toolkit for performance problem solving**
  - Instrumentation, measurement, analysis, and visualization
  - Portable performance profiling and tracing facility
  - Performance data management and data mining
- ***Partners***: LLNL, ANL, LANL, Research Center Jülich

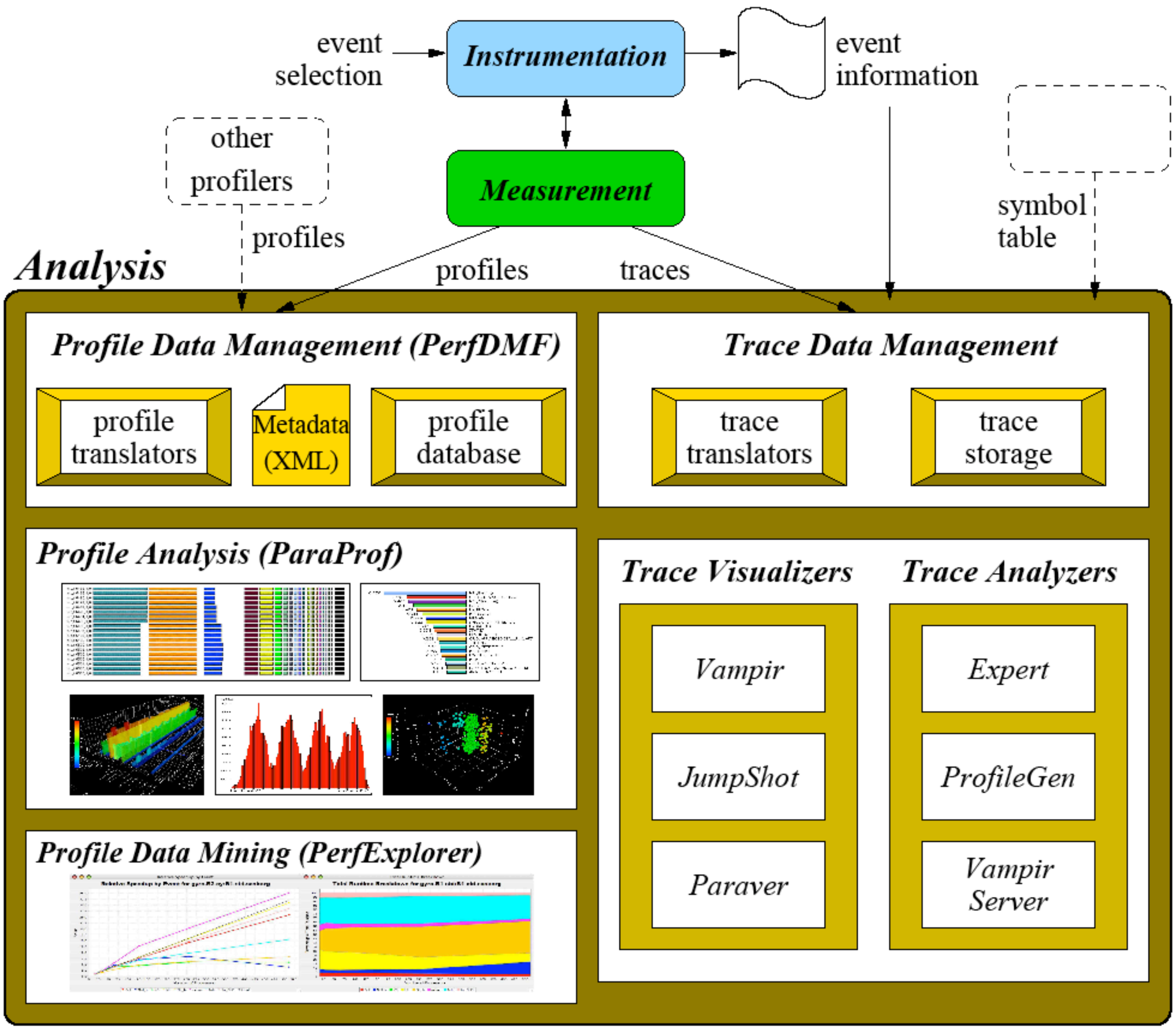
TAU Parallel Performance System Goals

- **Portable (open source) parallel performance system**
  - Computer system architectures and operating systems
  - Different programming languages and compilers
- Multi-level, multi-language performance instrumentation
- **Flexible and configurable performance measurement**
- Support for multiple parallel programming paradigms
  - Multi-threading, message passing, mixed-mode, hybrid, object oriented (generic), component-based
- Support for performance mapping
- Integration of leading performance technology
- **Scalable (very large) parallel performance analysis**

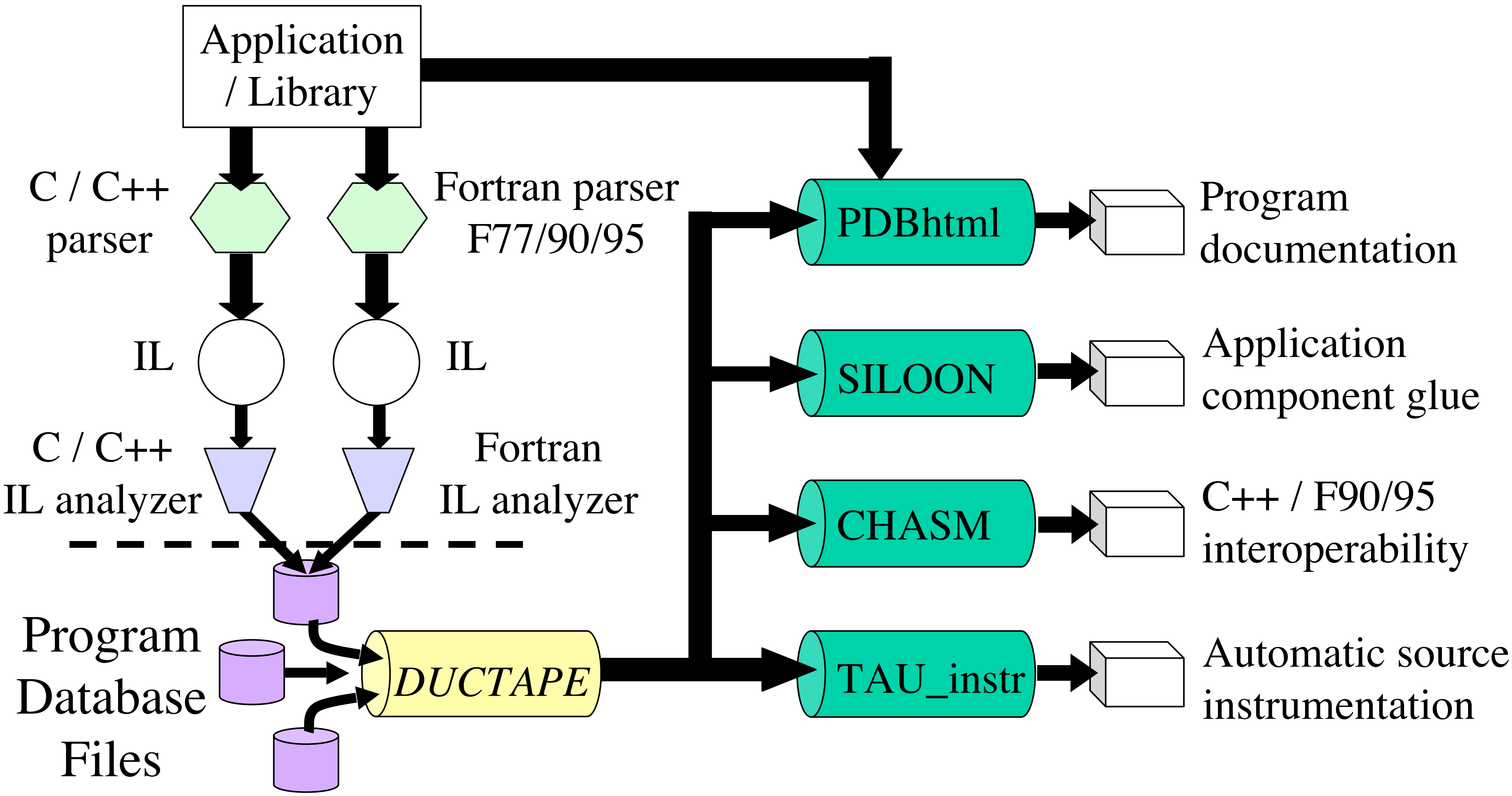
TAU Performance System Architecture



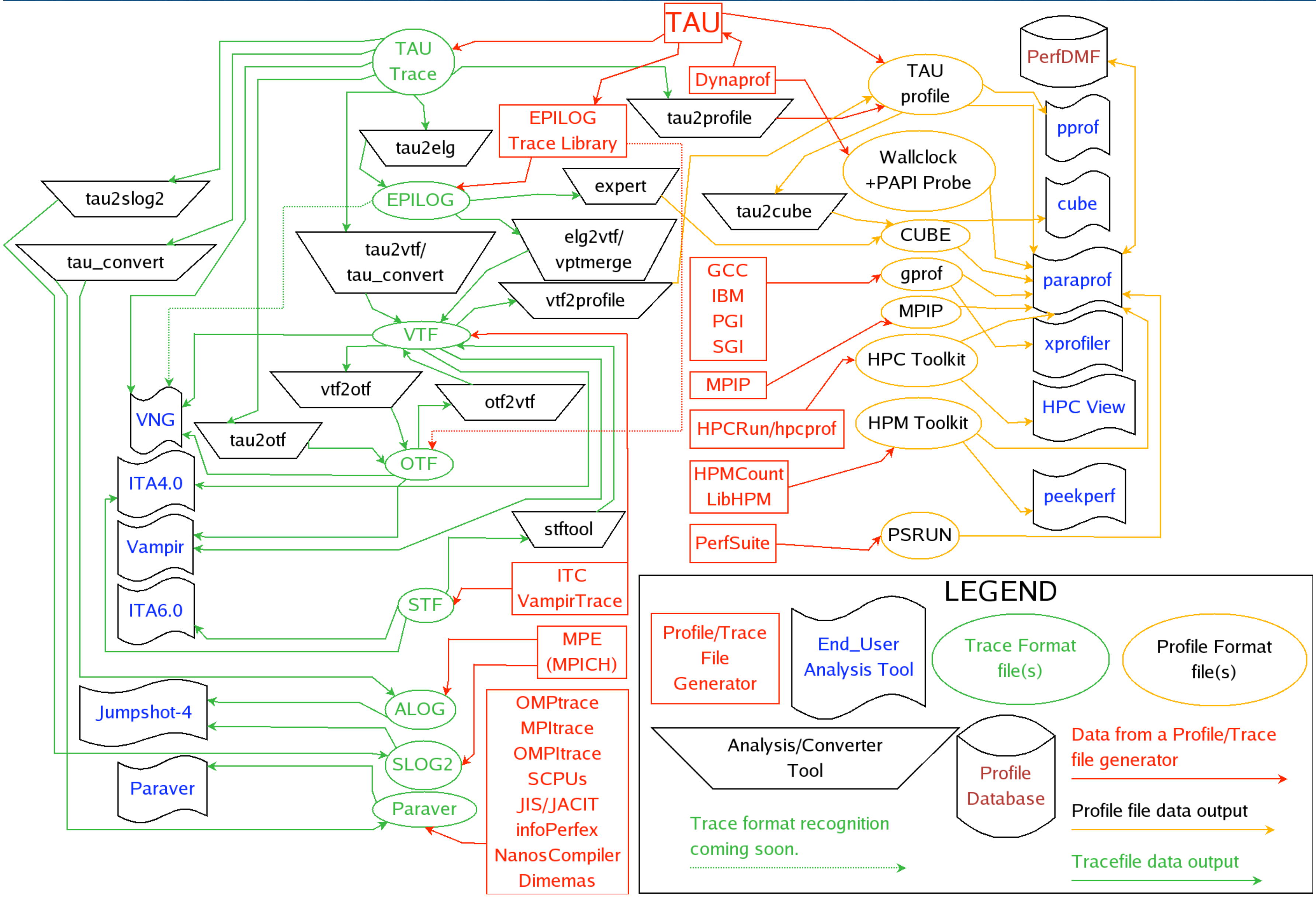
TAU Performance System Architecture



Program Database Toolkit (PDT)



Building Bridges to Other Tools: TAU



TAU Instrumentation Approach

- Support for *standard* program events
  - Routines, classes and templates
  - Statement-level blocks
- Support for *user-defined* events
  - *Begin/End* events (“user-defined timers”)
  - *Atomic* events (e.g., size of memory allocated/freed)
  - Selection of event statistics
  - Support for hardware performance counters (PAPI)
- Support definition of “semantic” entities for mapping
- Support for event groups (aggregation, selection)
- Instrumentation optimization
  - Eliminate instrumentation in lightweight routines

PAPI

- **P**erformance **A**pplication **P**rogramming **I**nterface
  - The purpose of the PAPI project is to design, standardize and implement a portable and efficient API to access the hardware performance monitor counters found on most modern microprocessors.
- Parallel Tools Consortium project started in 1998
- Developed by University of Tennessee, Knoxville
- http://icl.cs.utk.edu/papi/

TAU Instrumentation Mechanisms

- **Source code**
  - Manual (TAU API, TAU component API)
  - Automatic (robust)
    - C, C++, F77/90/95 (Program Database Toolkit (***PDT***))
    - OpenMP (directive rewriting (*Opari*), *POMP2* spec)
- **Object code**
  - Pre-instrumented libraries (e.g., MPI using *PMPI*)
  - Statically-linked and dynamically-linked
- **Executable code**
  - Dynamic instrumentation (pre-execution) (*DynInstAPI*)
  - Virtual machine instrumentation (e.g., Java using *JVMPI*)
- *`TAU_COMPILER`* to automate instrumentation process

Using TAU: A brief Introduction

- To instrument source code using PDT
  - Choose an appropriate TAU stub makefile in <arch>/lib:

  % setenv TAU_MAKEFILE /usr/tau-2.x/xt3/lib/Makefile.tau-mpi-pdt-pgi

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

  And use tau_f90.sh, tau_cxx.sh or tau_cc.sh as Fortran, C++ or C compilers:

  % mpif90 foo.f90

  changes to

  % tau_f90.sh foo.f90
- Execute application and analyze performance data:

  % pprof (for text based profile display)

  % paraprof (for GUI)

TAU Measurement System Configuration

- configure [OPTIONS]

{-c++=<CC>, -cc=<cc>}	Specify C++ and C compilers
-pdt=<dir>	Specify location of PDT
-opari=<dir>	Specify location of Opari OpenMP tool
-papi=<dir>	Specify location of PAPI
-vampirtrace=<dir>	Specify location of VampirTrace
-mpi[inc/lib]=<dir>	Specify MPI library instrumentation
-dyninst=<dir>	Specify location of DynInst Package
-shmem[inc/lib]=<dir>	Specify PSHMEM library instrumentation
-python[inc/lib]=<dir>	Specify Python instrumentation
-tag=<name>	Specify a unique configuration name
-epilog=<dir>	Specify location of EPILOG
-slog2	Build SLOG2/Jumpshot tracing package
-otf=<dir>	Specify location of OTF trace package
-arch=<architecture>	Specify architecture explicitly (bgl, xt3,ibm64,ibm64linux…)
{-pthread, -sproc}	Use pthread or SGI sproc threads
-openmp	Use OpenMP threads
-jdk=<dir>	Specify Java instrumentation (JDK)
-fortran=[vendor]	Specify Fortran compiler

TAU Measurement System Configuration

- configure [OPTIONS]

-TRACE	Generate binary TAU traces
-PROFILE (default)	Generate profiles (summary)
-PROFILECALLPATH	Generate call path profiles
-PROFILEPHASE	Generate phase based profiles
-PROFILEPARAM	Generate parameter based profiles
-PROFILEMEMORY	Track heap memory for each routine
-PROFILEHEADROOM	Track memory headroom to grow
-MULTIPLECOUNTERS	Use hardware counters + time
-COMPENSATE	Compensate timer overhead
-CPUTIME	Use usertime+system time
-PAPIWALLCLOCK	Use PAPI's wallclock time
-PAPIVIRTUAL	Use PAPI's process virtual time
-SGITIMERS	Use fast IRIX timers
-LINUXTIMERS	Use fast x86 Linux timers

Performance Evaluation Alternatives

Depthlimit profile

Parameter profile

Callpath/ callgraph profile

Flat profile

Phase profile

Trace

Each alternative has:
- one metric/counter
- multiple counters

Volume of performance data

TAU Measurement Configuration – Examples

- ./configure –pdt=/usr/pkgs/pkgs/pdtoolkit-3.11 -mpiinc=/usr/pkgs/mpich/include -mpilib=/usr/pkgs/mpich/lib -mpilibrary=‘-lmpich -L/usr/gm/lib64 -lgm -lpthread -ldl’
  - Configure using PDT and MPI for x86_64 Linux
- ./configure -arch=xt3 -papi=/opt/xt-tools/papi/3.2.1 -mpi -MULTIPLECOUNTERS; make clean install
  - Use PAPI counters (one or more) with C/C++/F90 automatic instrumentation for XT3. Also instrument the MPI library. Use PGI compilers.
- Typically configure multiple measurement libraries
- Each configuration creates a unique <arch>/lib/Makefile.tau<options> stub makefile. It corresponds to the configuration options used. e.g.,
  - /usr/pkgs/tau/x86_64/lib/Makefile.tau-mpi-pdt-pgi
  - /usr/pkgs/tau/x86_64/lib/Makefile.tau-multiplecounters-mpi-papi-pdt-pgi

TAU Measurement Configuration – Examples

% cd /usr/pkgs/tau/x86_64/lib; ls Makefile.*pgi
Makefile.tau-pdt-pgi
Makefile.tau-mpi-pdt-pgi
Makefile.tau-callpath-mpi-pdt-pgi
Makefile.tau-mpi-pdt-trace-pgi
Makefile.tau-mpi-compensate-pdt-pgi
Makefile.tau-multiplecounters-mpi-papi-pdt-pgi
Makefile.tau-multiplecounters-mpi-papi-pdt-trace-pgi
Makefile.tau-mpi-papi-pdt-epilog-trace-pgi
Makefile.tau-pdt-pgi…

- For an MPI+F90 application, you may want to start with:

Makefile.tau-mpi-pdt-pgi

  - Supports MPI instrumentation & PDT for automatic source instrumentation for PGI compilers

Configuration Parameters in Stub Makefiles

- Each TAU stub Makefile resides in <tau>/<arch>/lib directory
- Variables:

TAU_CXX	Specify the C++ compiler used by TAU
TAU_CC, TAU_F90	Specify the C, F90 compilers
TAU_DEFS	Defines used by TAU. Add to CFLAGS
TAU_LDFLAGS	Linker options. Add to LDFLAGS
TAU_INCLUDE	Header files include path. Add to CFLAGS
TAU_LIBS	Statically linked TAU library. Add to LIBS
TAU_SHLIBS	Dynamically linked TAU library
TAU_MPI_LIBS	TAU's MPI wrapper library for C/C++
TAU_MPI_FLIBS	TAU's MPI wrapper library for F90
TAU_FORTRANLIBS	Must be linked in with C++ linker for F90
TAU_CXXLIBS	Must be linked in with F90 linker
TAU_INCLUDE_MEMORY	Use TAU's malloc/free wrapper lib
TAU_DISABLE	TAU's dummy F90 stub library
TAU_COMPILER	Instrument using tau_compiler.sh script

- Each stub makefile encapsulates the parameters that TAU was configured with
- It represents a specific instance of the TAU libraries. TAU scripts use stub makefiles to identify what performance measurements are to be performed.

Automatic Instrumentation

- We now provide compiler wrapper scripts
  - Simply replace `mpxlf90` with `tau_f90.sh`
  - Automatically instruments Fortran source code, links with TAU MPI Wrapper libraries.
- Use `tau_cc.sh` and `tau_cxx.sh` for C/C++

Before

```
CXX = mpCC
F90 = mpxlf90_r
CFLAGS =
LIBS = -lm
OBJS = f1.o f2.o f3.o … fn.o

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

After

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

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

TAU_COMPILER Commandline Options

- See `<taudir>/<arch>/bin/tau_compiler.sh -help`
- Compilation:

  ```
  % mpxlf90 -c foo.f90
  ```

  Changes to

  ```
  % f95parse foo.f90 $(OPT1)
  % tau_instrumentor foo.pdb foo.f90 -o foo.inst.f90 $(OPT2)
  % mpxlf90 -c foo.f90 $(OPT3)
  ```
- Linking:

  ```
  % mpxlf90 foo.o bar.o -o app
  ```

  Changes to

  ```
  % mpxlf90 foo.o bar.o -o app $(OPT4)
  ```
- Where options OPT[1-4] default values may be overridden by the user:

  ```
  F90 = $(TAU_COMPILER) $(MYOPTIONS) mpxlf90
  ```

TAU_COMPILER Options

- Optional parameters for $(TAU_COMPILER): [tau_compiler.sh –help]

-optVerbose	Turn on verbose debugging messages
-optDetectMemoryLeaks	Turn on debugging memory allocations/ de-allocations to track leaks
-optPdtGnuFortranParser	Use gfparse (GNU) instead of f95parse (Cleanscape) for parsing Fortran source code
-optKeepFiles	Does not remove intermediate .pdb and .inst.* files
-optPreProcess	Preprocess Fortran sources before instrumentation
-optTauSelectFile=""	Specify selective instrumentation file for tau_instrumentor
-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)
-optPdtF95Reset=""	Reset options for Fortran parser in PDT (f95parse/gfparse)
-optPdtCOpts=""	Options for C parser in PDT (cparse). Typically $(TAU_MPI_INCLUDE) $(TAU_INCLUDE) $(TAU_DEFS)
-optPdtCxxOpts=""	Options for C++ parser in PDT (cxxparse). Typically $(TAU_MPI_INCLUDE) $(TAU_INCLUDE) $(TAU_DEFS)

...

Overriding Default Options:TAU_COMPILER

```
% cat Makefile
F90 = tau_f90.sh
OBJS = f1.o f2.o f3.o …
LIBS = -Lappdir –lapplib1 –lapplib2 …

app: $(OBJS)
    $(F90) $(OBJS) –o app $(LIBS)
.f90.o:
    $(F90) –c $<
% setenv TAU_OPTIONS '-optVerbose -optTauSelectFile=select.tau
      -optKeepFiles'
% setenv TAU_MAKEFILE <taudir>/x86_64/lib/Makefile.tau-mpi-pdt
```

Optimization of Program Instrumentation

- Need to eliminate instrumentation in frequently executing lightweight routines
- Throttling of events at runtime:

```
% setenv TAU_THROTTLE 1
```

  Turns off instrumentation in routines that execute over 100000 times (TAU_THROTTLE_NUMCALLS) and take less than 10 microseconds of inclusive time per call (TAU_THROTTLE_PERCALL)

- Selective instrumentation file to filter events

```
% tau_instrumentor [options] –f <file>  OR
% setenv TAU_OPTIONS '-optTauSelectFile=tau.txt'
```

- Compensation of local instrumentation overhead

```
% configure -COMPENSATE
```

Selective Instrumentation File

- Specify a list of routines to exclude or include (case sensitive)
- # is a wildcard in a routine name. It cannot appear in the first column.

```
BEGIN_EXCLUDE_LIST
Foo
Bar
D#EMM
END_EXCLUDE_LIST
```

- Specify a list of routines to include for instrumentation

```
BEGIN_INCLUDE_LIST
int main(int, char **)
F1
F3
END_EXCLUDE_LIST
```

- Specify either an include list or an exclude list!

Selective Instrumentation File

- Optionally specify a list of files to exclude or include (case sensitive)
- * and ? may be used as wildcard characters in a file name

```
BEGIN_FILE_EXCLUDE_LIST
f*.f90
Foo?.cpp
END_FILE_EXCLUDE_LIST
```

- Specify a list of routines to include for instrumentation

```
BEGIN_FILE_INCLUDE_LIST
main.cpp
foo.f90
END_FILE_INCLUDE_LIST
```

Selective Instrumentation File

- User instrumentation commands are placed in INSTRUMENT section
- ? and * used as wildcard characters for file name, # for routine name
- \ as escape character for quotes
- Routine entry/exit, arbitrary code insertion
- Outer-loop level instrumentation

```
BEGIN_INSTRUMENT_SECTION
loops file="foo.f90" routine="matrix#"
memory file="foo.f90" routine="#"
io routine="MATRIX"
file="foo.f90" line = 123 code = "  print *, \" In foo\""
exit routine = "int f1()" code = "cout <<\"Out f1\"<<endl;"
END_INSTRUMENT_SECTION
```

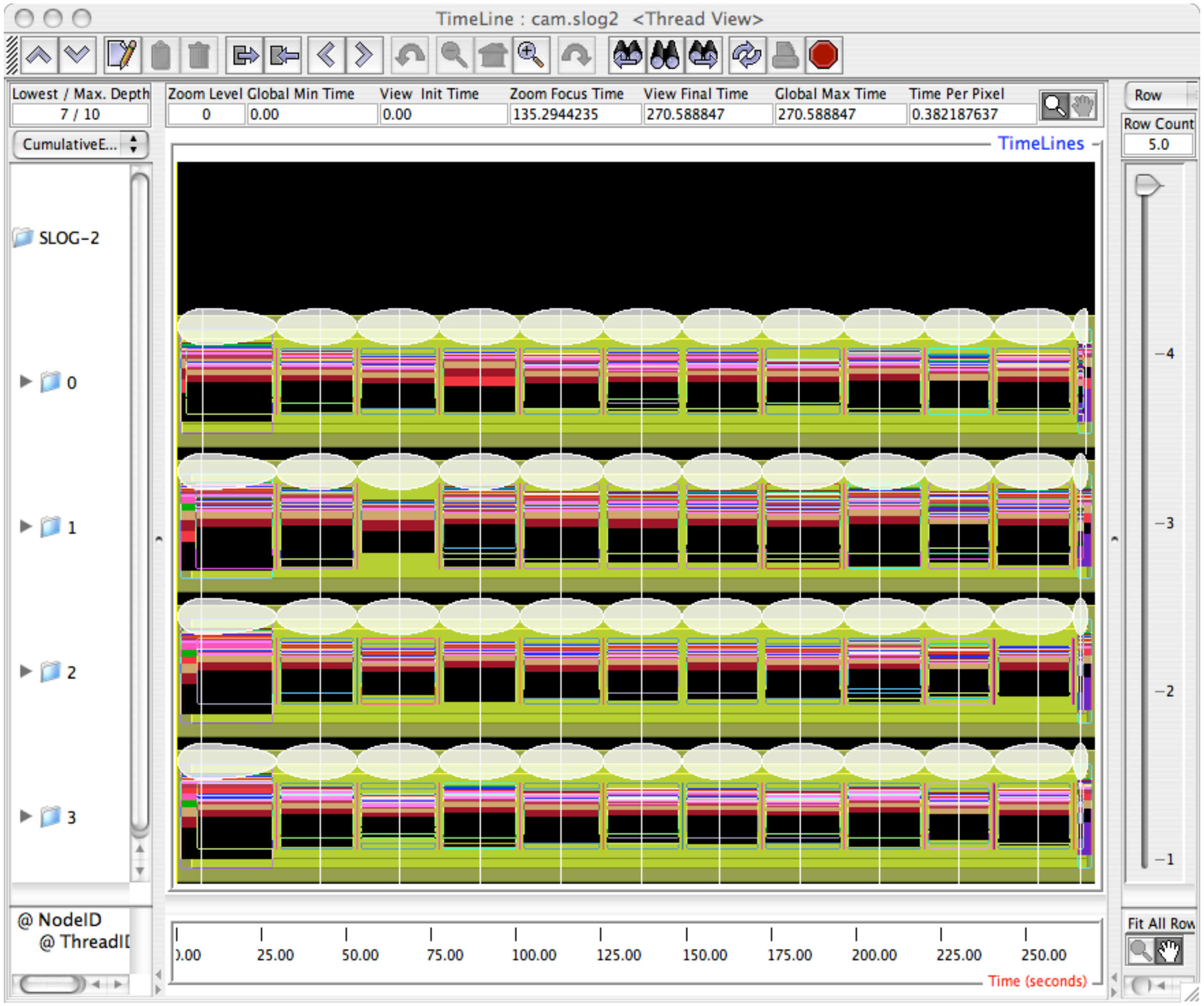
Manual Instrumentation – C/C++ Example

```
#include <TAU.h>
int main(int argc, char **argv)
{
  TAU_START ("big-loop")

  for(int i = 0; i < N ; i++){
    work(i);
  }

  TAU_STOP ("big-loop");

}

% g++ foo.cpp -I<taudir>/include -c
% g++ foo.o -o foo -L<taudir>/<arch>/lib -lTAU
```

Jumpshot

- http://www-unix.mcs.anl.gov/perfvis/software/viewers/index.htm
- Developed at Argonne National Laboratory as part of the MPICH project
- Rusty Lusk, PI
  - Also works with other MPI implementations
  - Jumpshot is bundled with the TAU package
- Java-based tracefile visualization tool for postmortem performance analysis of MPI programs
- Latest version is Jumpshot-4 for SLOG-2 format
  - Scalable level of detail support
  - Timeline and histogram views
  - Scrolling and zooming
  - Search/scan facility

Jumpshot



TimeLine : cam.slog2 <Thread View>
Lowest / Max. Depth
7 / 10
Zoom Level
0
Global Min Time
0.00
View Init Time
0.00
Zoom Focus Time
135.2944235
View Final Time
270.588847
Global Max Time
270.588847
Time Per Pixel
0.382187637
Row
Row Count
5.0
CumulativeE...
TimeLines
SLOG-2
0
1
2
3
@ NodeID
@ ThreadI
Fit All Row
25.00
50.00
75.00
100.00
125.00
150.00
175.00
200.00
225.00
250.00
Time (seconds)

Tracing: Using TAU and Jumpshot

- Configure TAU with -TRACE option:

```
% configure -TRACE -otf=<dir>
  -MULTIPLECOUNTERS -papi=<dir> -mpi
  -pdt=dir …
```

- Set environment variables:

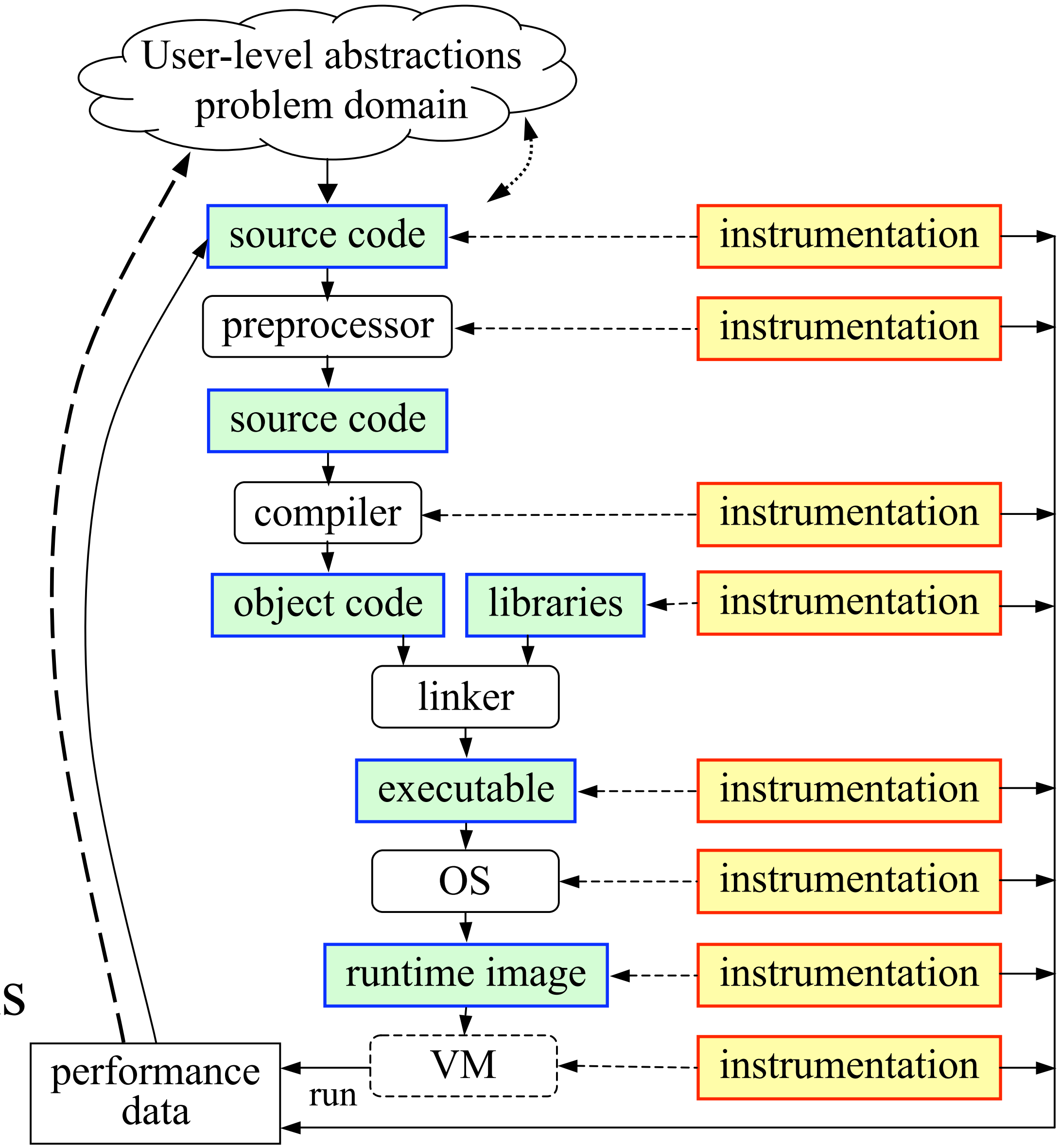
```
% setenv TRACEDIR /p/gm1/<login>/traces
% setenv COUNTER1 GET_TIME_OF_DAY (reqd)
% setenv COUNTER2 PAPI_FP_INS
% setenv COUNTER3 PAPI_TOT_CYC …
```

- Execute application and analyze the traces:

```
% mpirun -np 32 ./a.out [args]

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

Multi-Level Instrumentation and Mapping

- **Multiple interfaces**
- **Information sharing**
  - Between interfaces
- **Event selection**
  - Within/between levels
- **Mapping**
  - Associate performance data with high-level semantic abstractions



TAU Measurement Approach

- **Portable and scalable parallel profiling solution**
  - Multiple profiling types and options
  - Event selection and control (enabling/disabling, throttling)
  - Online profile access and sampling
  - Online performance profile overhead compensation
- **Portable and scalable parallel tracing solution**
  - Trace translation to SLOG2, OTF, EPILOG, and Paraver
  - Trace streams (OTF) and hierarchical trace merging
- Robust timing and hardware performance support
- Multiple counters (hardware, user-defined, system)
- Performance measurement for CCA component software

TAU Measurement Mechanisms

- **Parallel profiling**
  - Function-level, block-level, statement-level
  - Supports user-defined events and mapping events
  - TAU parallel profile stored (dumped) during execution
  - Support for flat, callgraph/callpath, phase profiling
  - Support for memory profiling (headroom, malloc/leaks)
  - Support for tracking I/O (wrappers, Fortran instrumentation of read/write/print calls)
- **Tracing**
  - All profile-level events
  - Inter-process communication events
  - Inclusion of multiple counter data in traced events

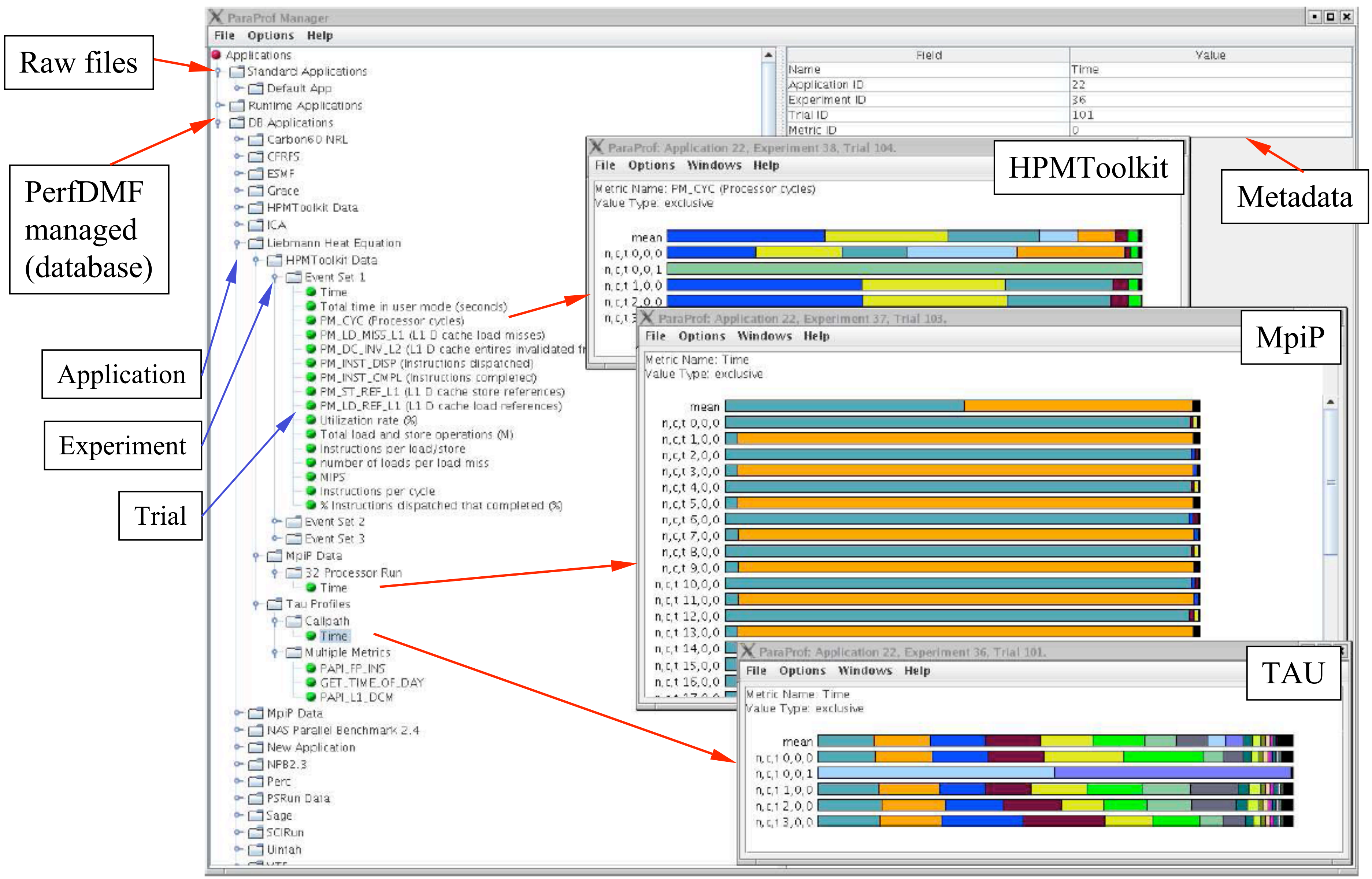
Types of Parallel Performance Profiling

- ***Flat*** profiles
  - Metric (e.g., time) spent in an event (callgraph nodes)
  - Exclusive/inclusive, # of calls, child calls
- ***Callpath*** profiles (***Calldepth*** profiles)
  - Time spent along a calling path (edges in callgraph)
  - "*main=> f1 => f2 => MPI_Send*" (event name)
  - `TAU_CALLPATH_DEPTH` environment variable
- ***Phase*** profiles
  - Flat profiles under a phase (nested phases are allowed)
  - Default "main" phase
  - Supports static or dynamic (per-iteration) phases

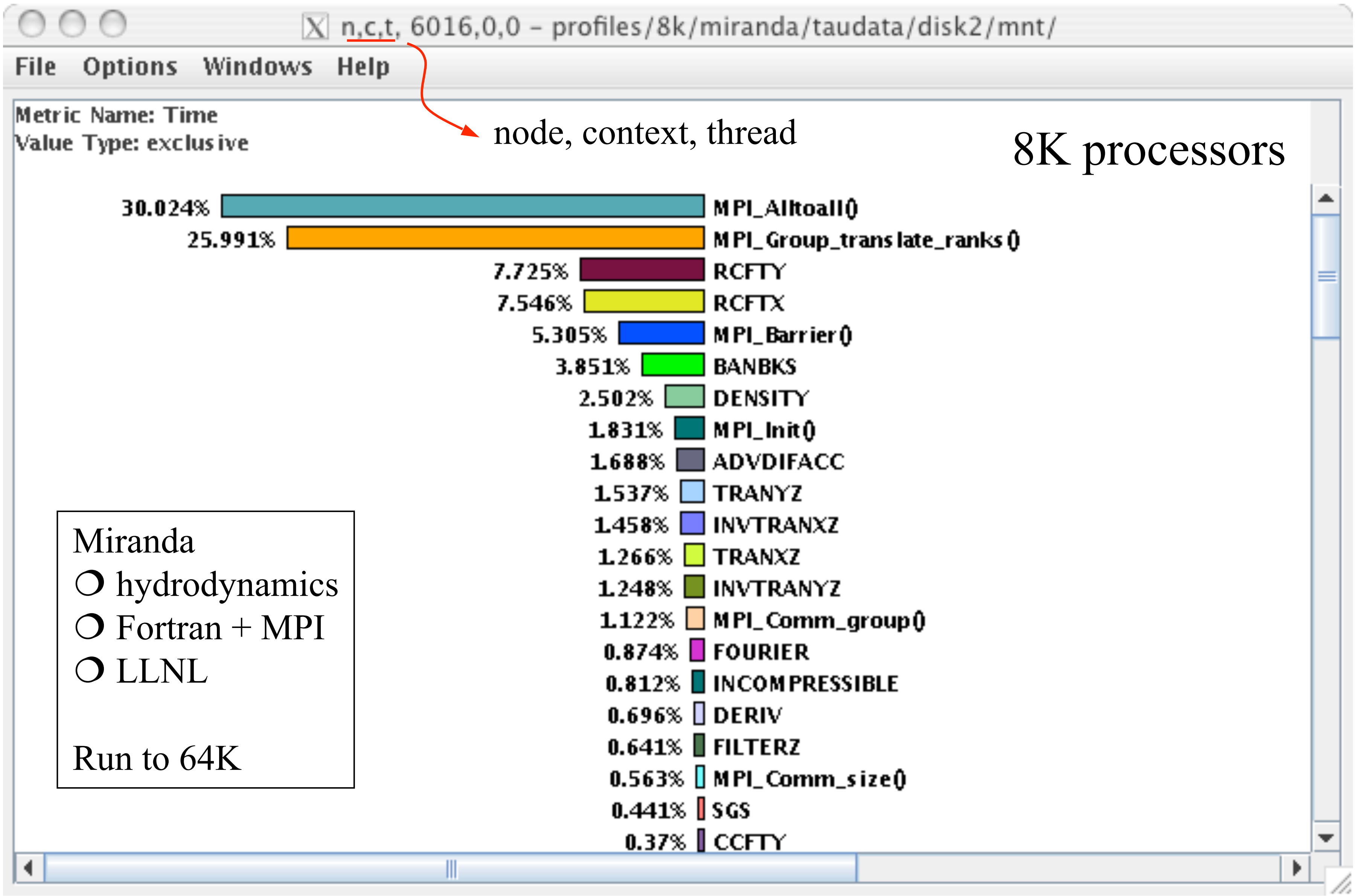
Performance Analysis and Visualization

- Analysis of parallel profile and trace measurement
- **Parallel profile analysis**
  - ***ParaProf***: parallel profile analysis and presentation
  - ***ParaVis***: parallel performance visualization package
  - Profile generation from trace data (***tau2profile***)
- Performance data management framework (***PerfDMF***)
- **Parallel trace analysis**
  - Translation to *VTF* (V3.0), *EPILOG, OTF* formats
  - Integration with *VNG* (Technical University of Dresden)
- Online parallel analysis and visualization
- Integration with *CUBE* browser (KOJAK, UTK, FZJ)

ParaProf Parallel Performance Profile Analysis



ParaProf – Flat Profile (Miranda, BG/L)



node, context, thread

8K processors

Miranda
- hydrodynamics
- Fortran + MPI
- LLNL

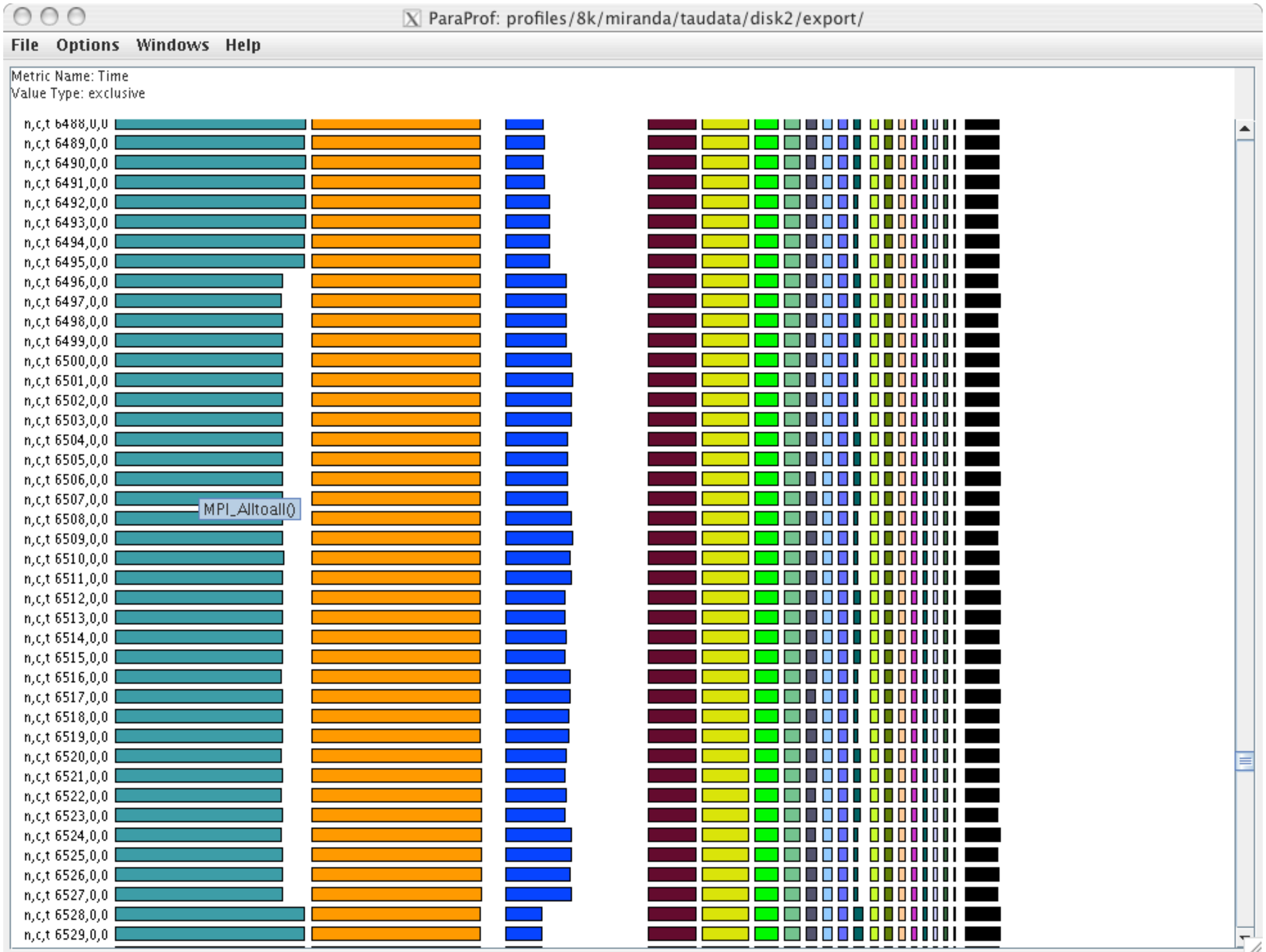
Run to 64K

Terminology – Example

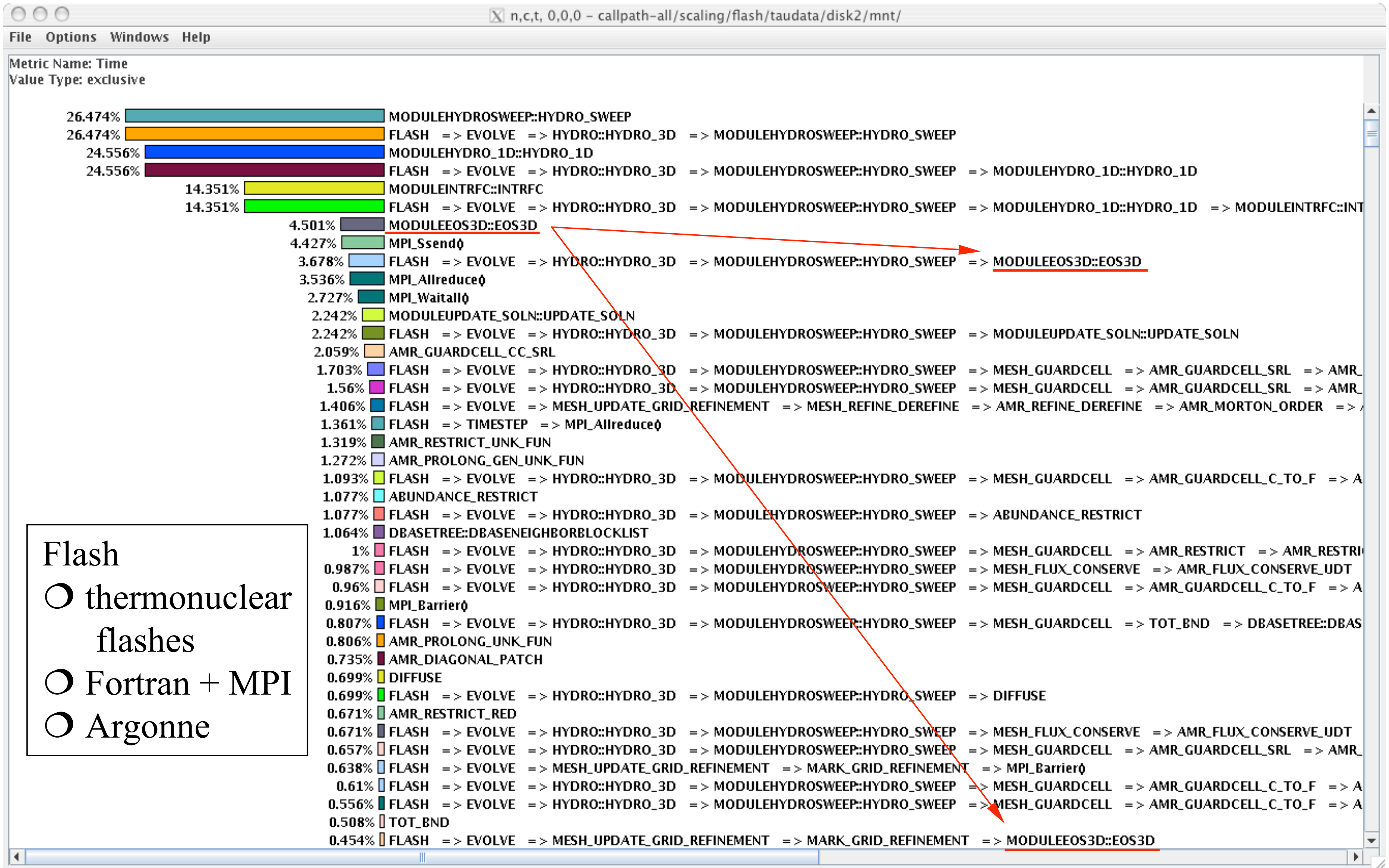
- For routine "int main( )":
- Exclusive time
  - 100-20-50-20=10 secs
- Inclusive time
  - 100 secs
- Calls
  - 1 call
- Subrs (no. of child routines called)
  - 3
- Inclusive time/call
  - 100secs

```
int main( )
{ /* takes 100 secs */

  f1(); /* takes 20 secs */
  f2(); /* takes 50 secs */
  f1(); /* takes 20 secs */

  /* other work */
}

/*
Time can be replaced by  counts
from PAPI e.g., PAPI_FP_OPS. */
```

ParaProf – Stacked View (Miranda)



ParaProf – Callpath Profile (Flash)

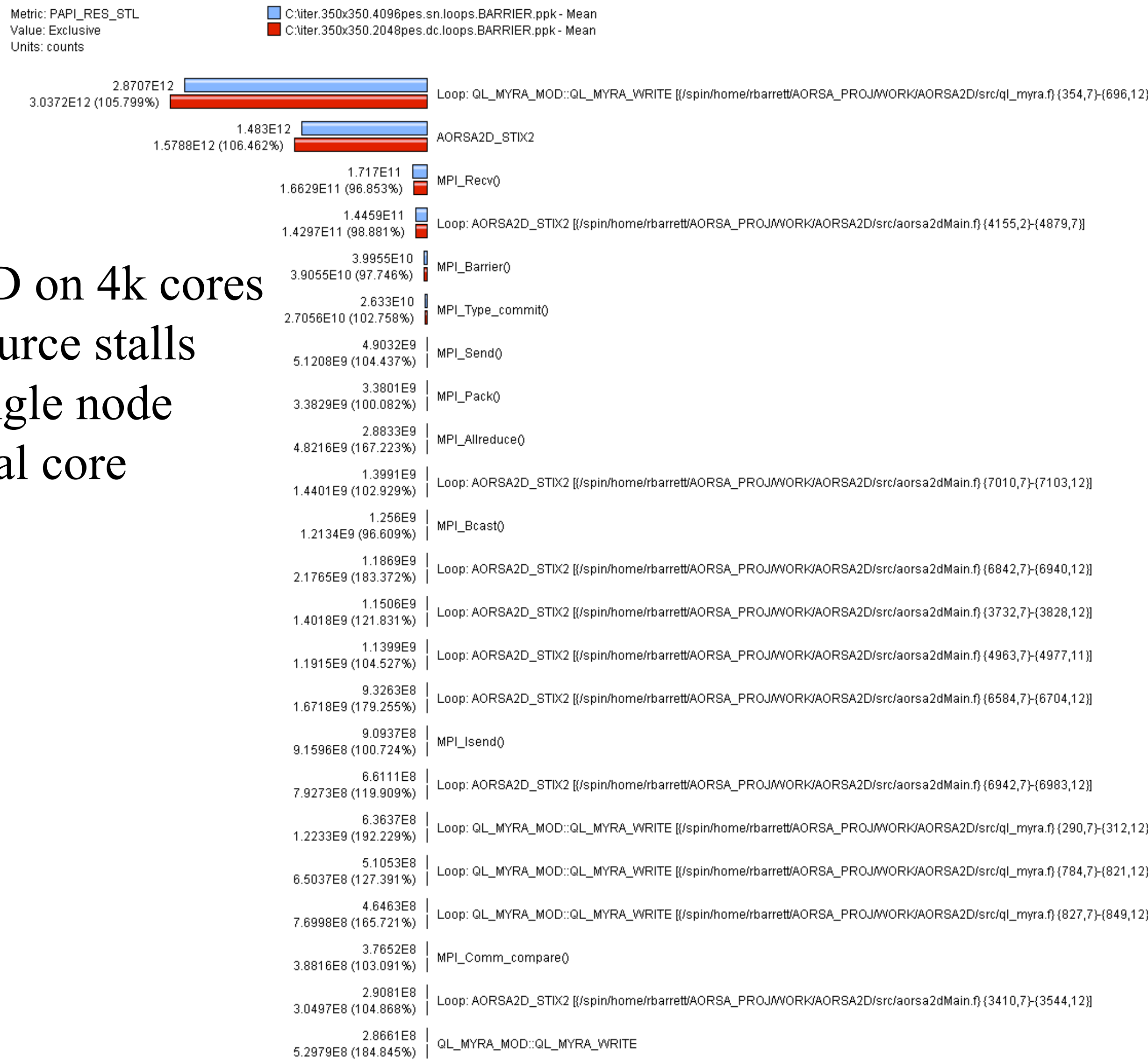


Flash

- thermonuclear flashes
- Fortran + MPI
- Argonne

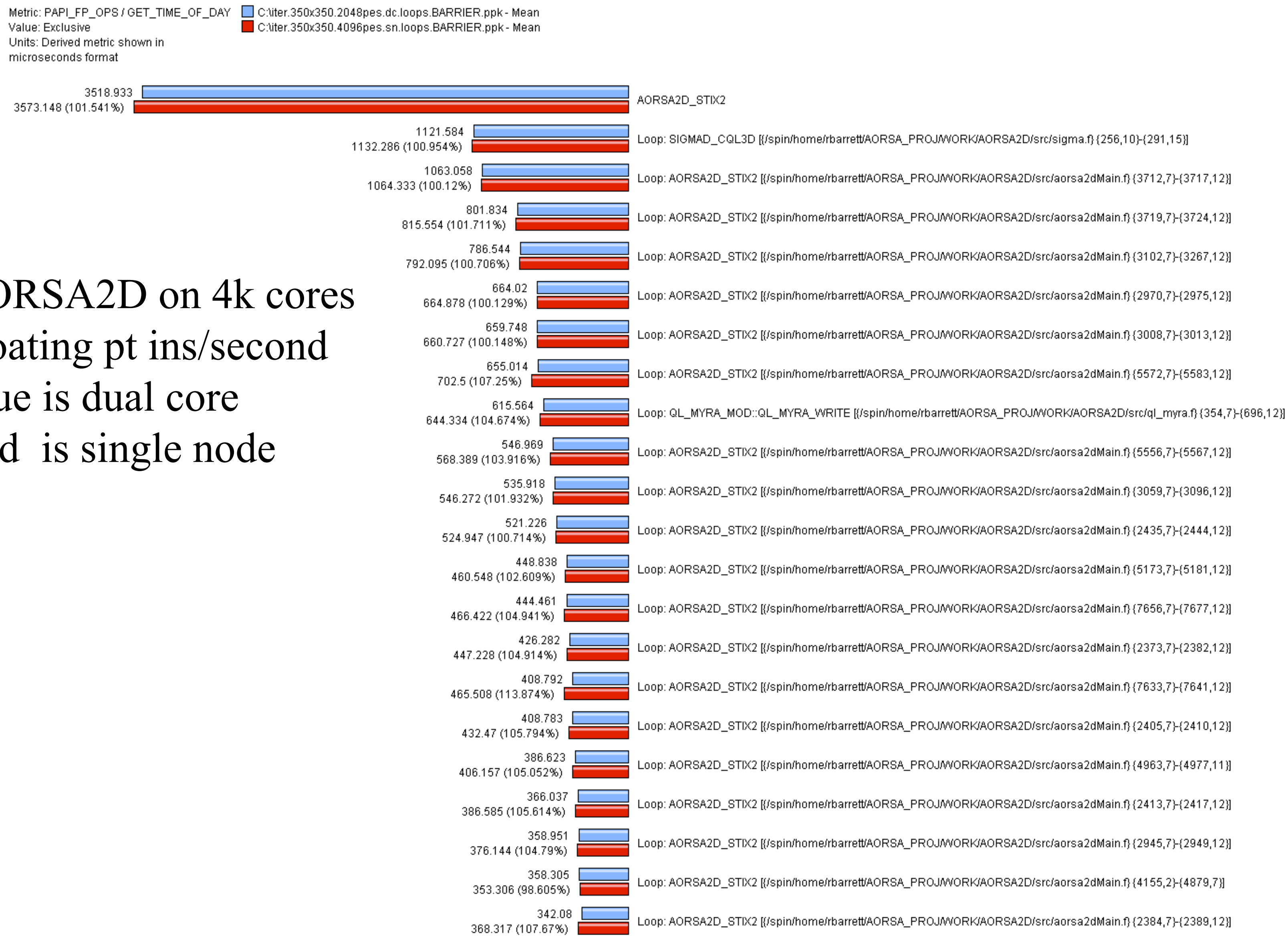
Comparing Effects of MultiCore Processors

- AORSA2D on 4k cores
- PAPI resource stalls
- Blue is single node
- Red is dual core

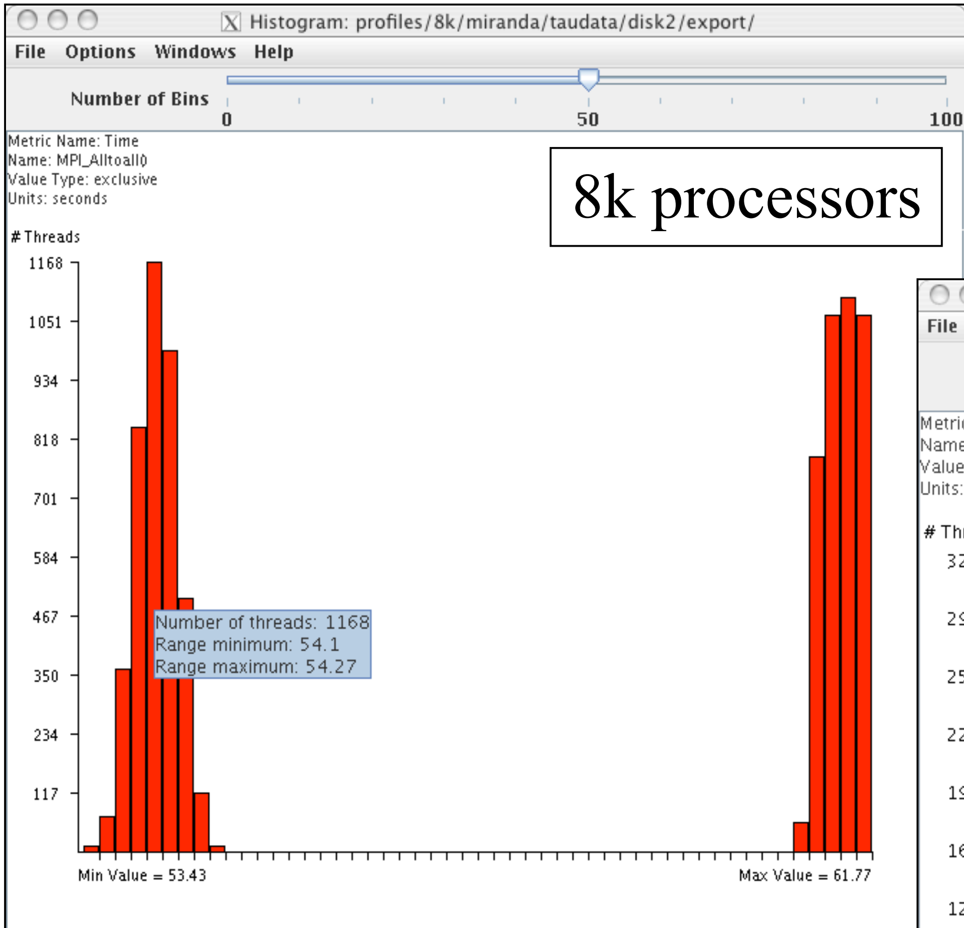


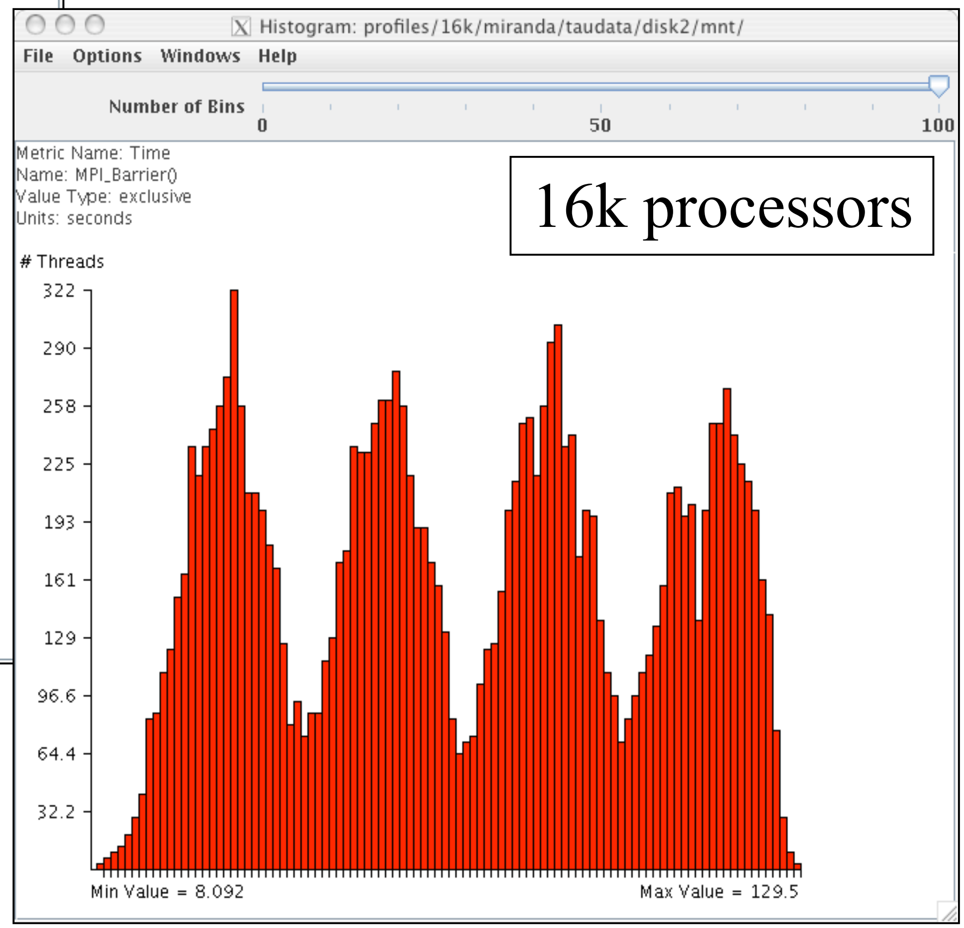
Comparing FLOPS: MultiCore Processors

- AORSA2D on 4k cores
- Floating pt ins/second
- Blue is dual core
- Red is single node

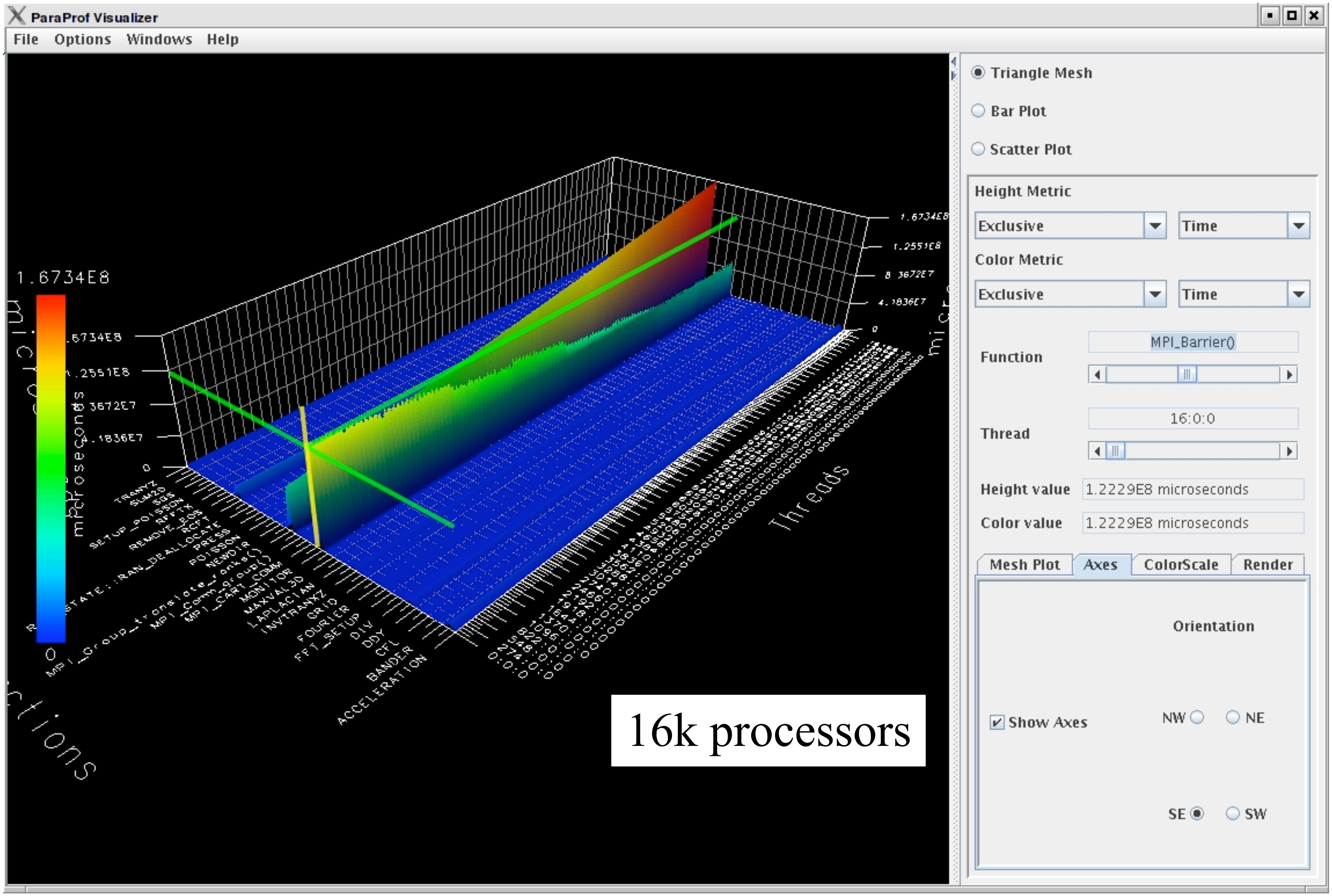


ParaProf – Scalable Histogram View (Miranda)



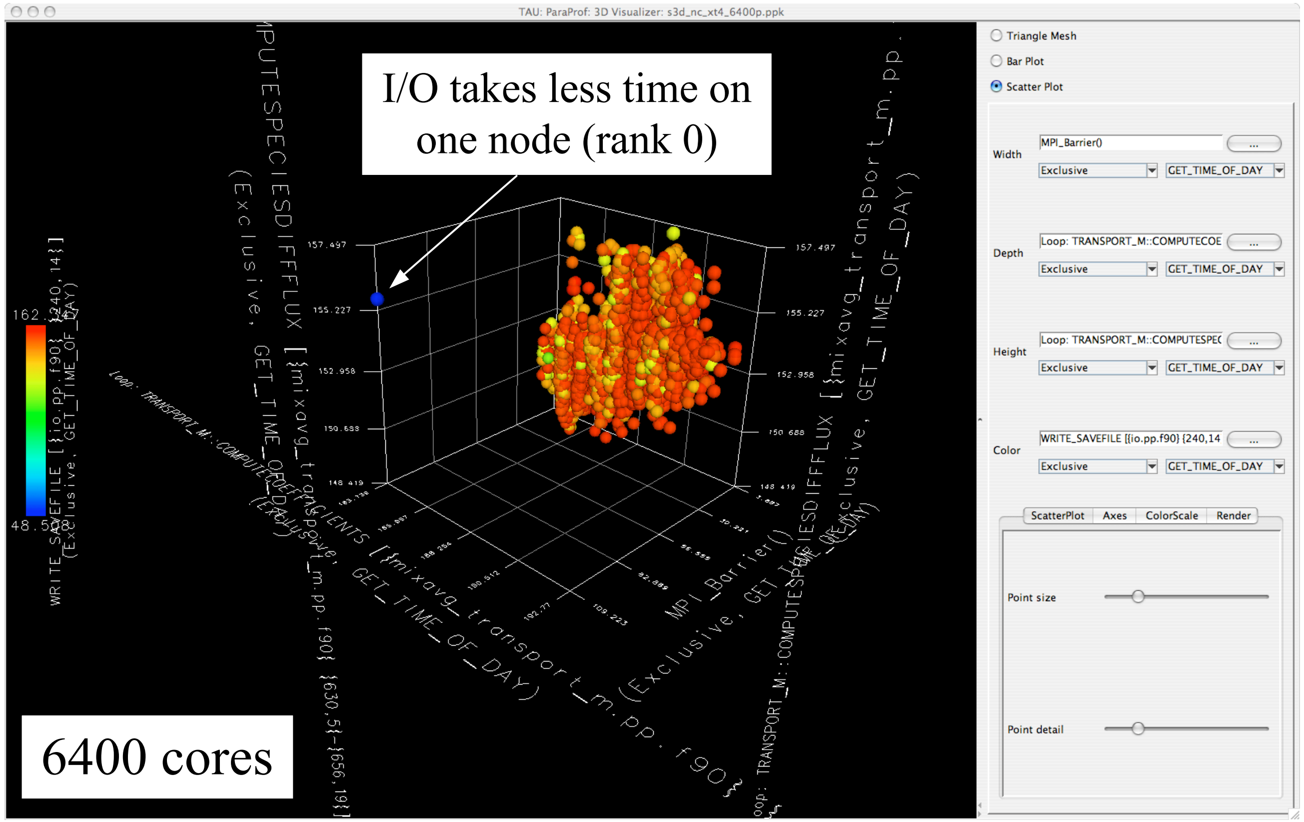


ParaProf – 3D Full Profile (Miranda)



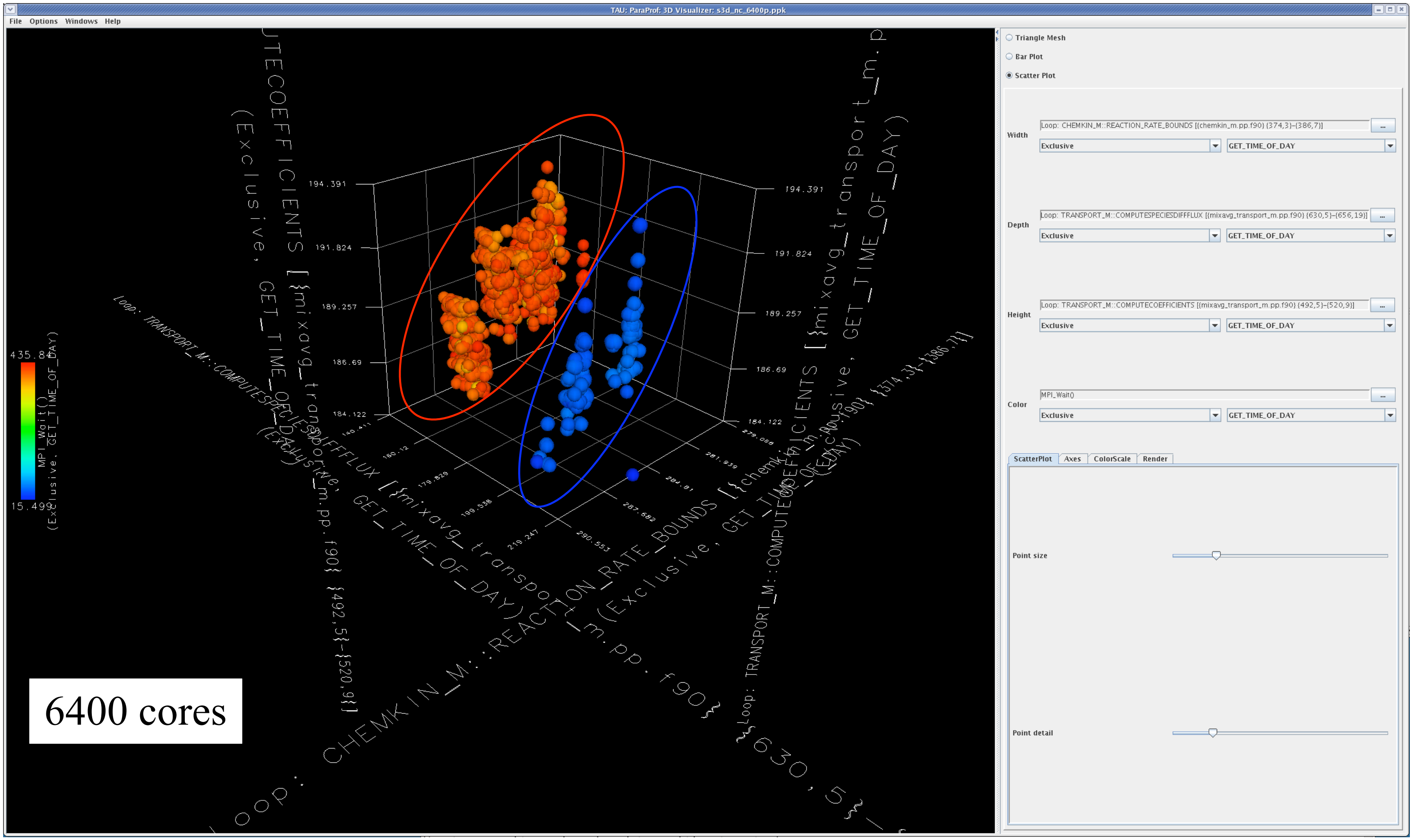
ParaProf – 3D Scatterplot (S3D – XT4 only)

- Each point is a "thread" of execution
- A total of four metrics shown in relation
- ParaVis 3D profile visualization library
  - JOGL



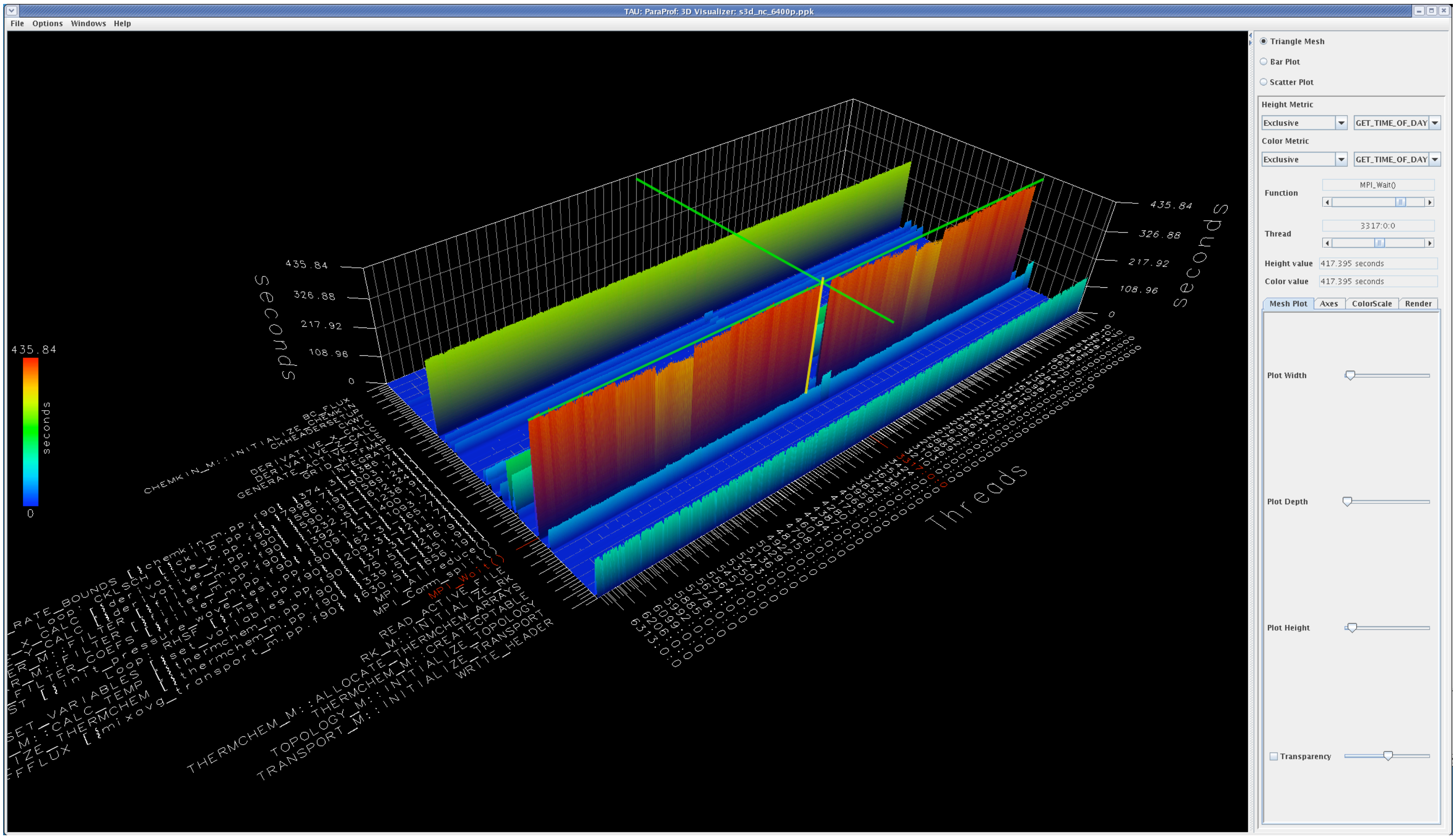
- Events (exclusive time metric)
  - MPI_Barrier(), two loops
  - write operation

S3D Scatter Plot: Visualizing Hybrid XT3+XT4



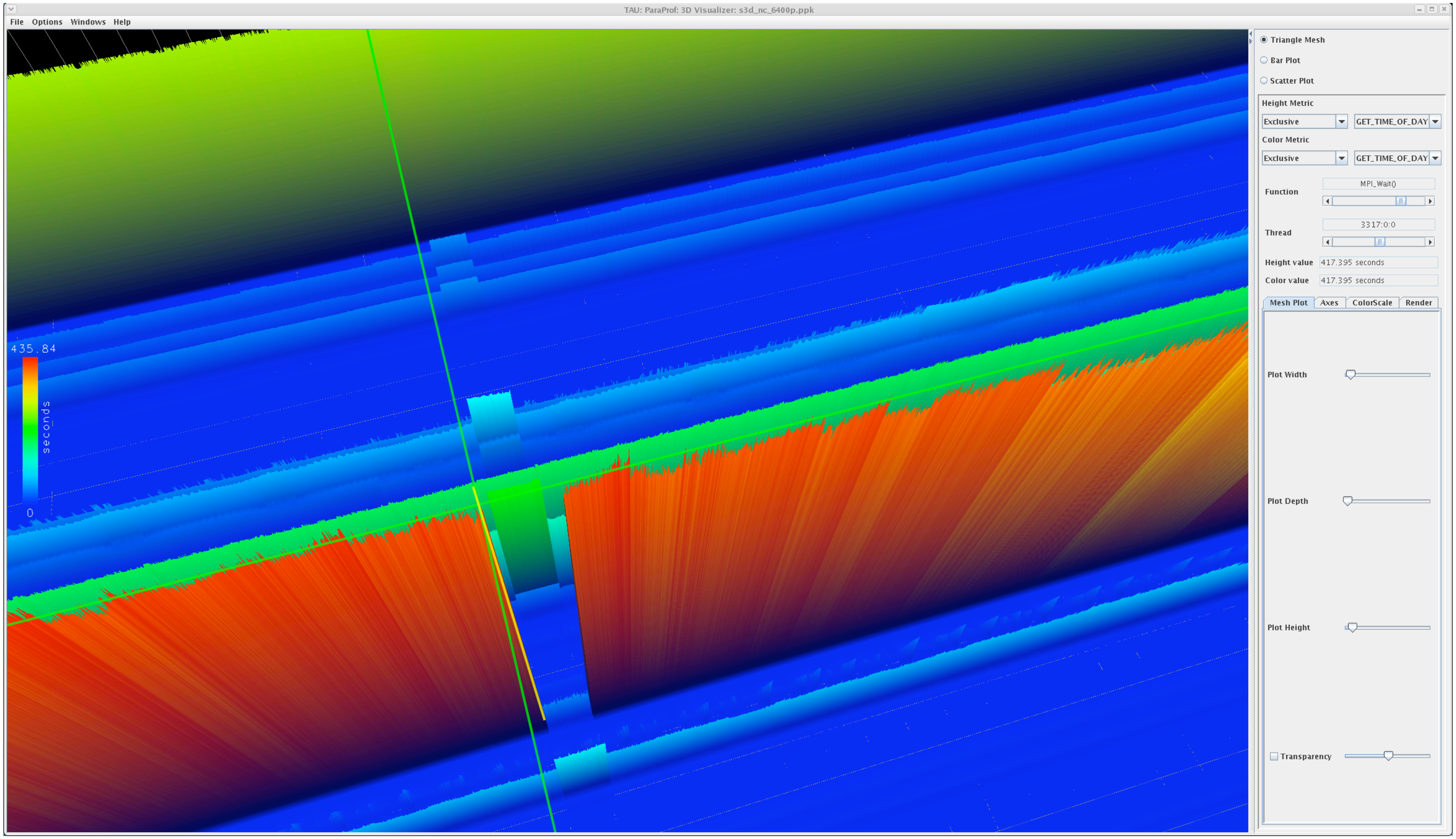
- Red nodes are XT4, blue are XT3

S3D: 6400 cores on XT3+XT4 System (Jaguar)



- Gap represents XT3 nodes

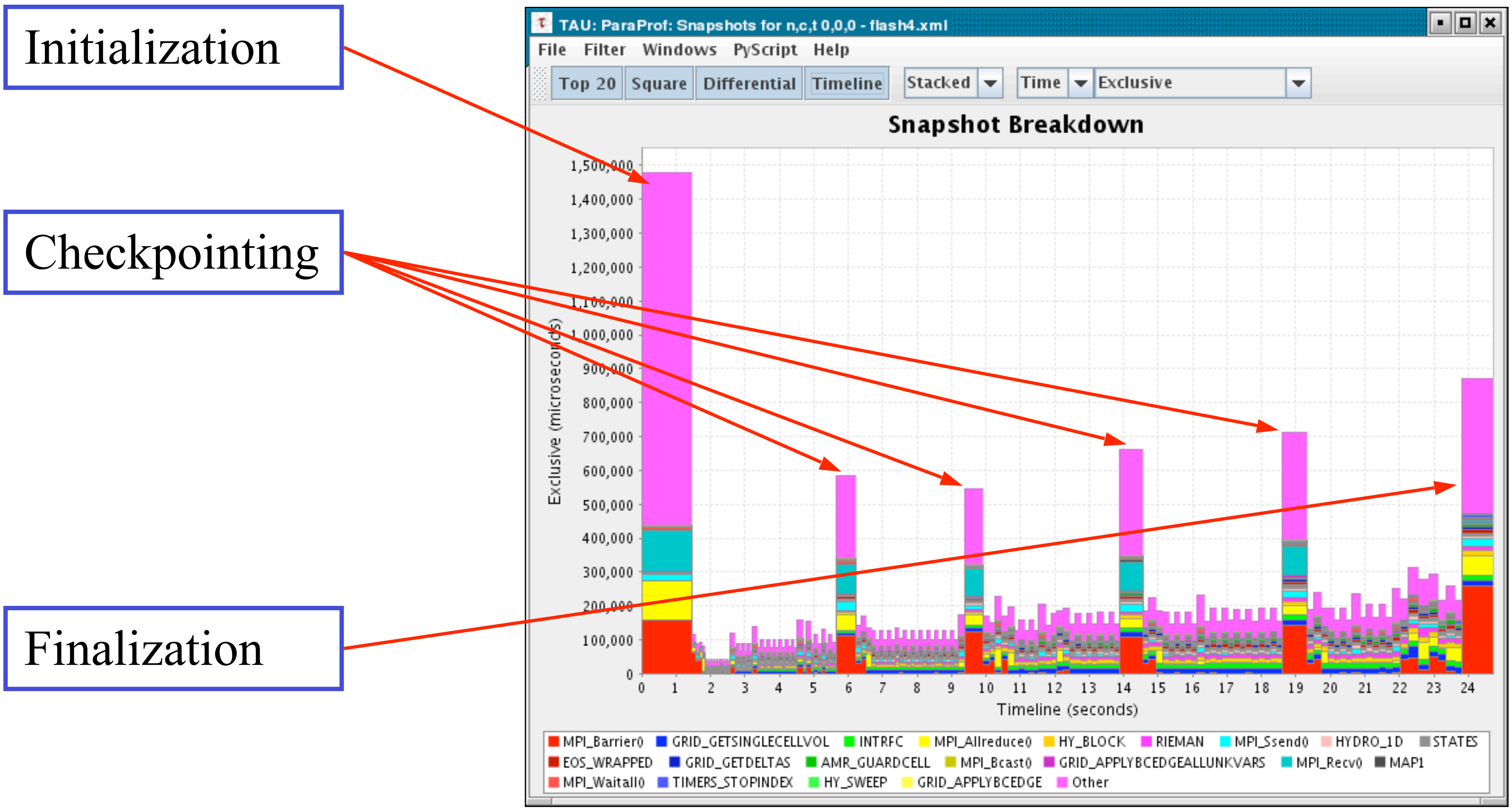
Visualizing S3D Profiles in ParaProf



- Gap represents XT3 nodes
  - MPI_Wait takes less time, other routines take more time

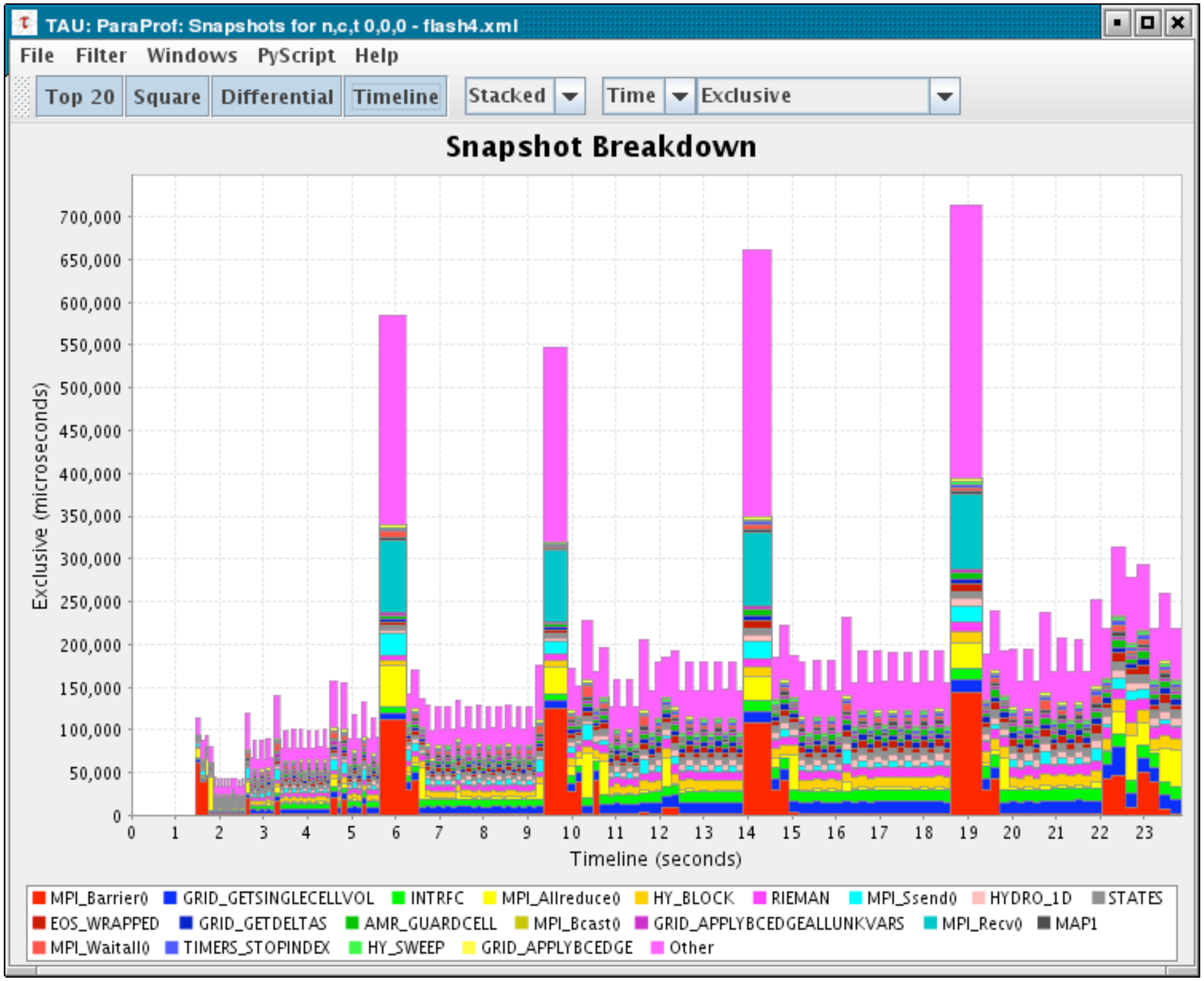
Profile Snapshots in ParaProf

- Profile snapshots are parallel profiles recorded at runtime
- Used to highlight profile changes during execution



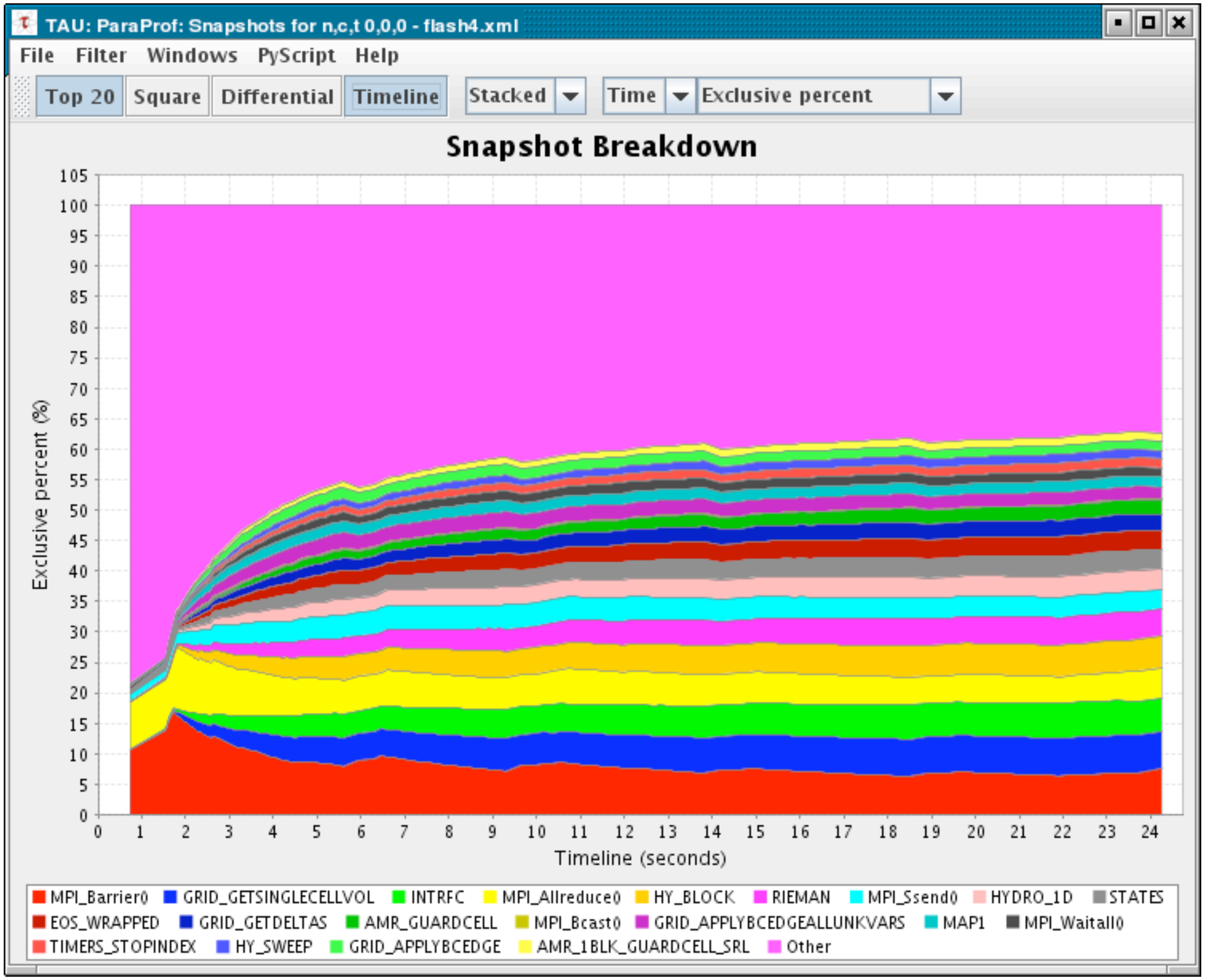
Profile Snapshots in ParaProf

- Filter snapshots (only show main loop iterations)

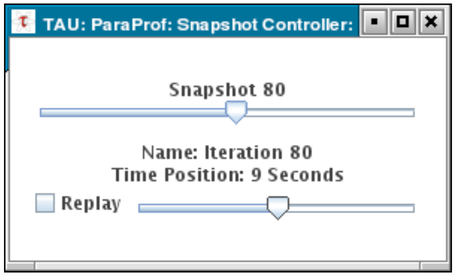


Profile Snapshots in ParaProf

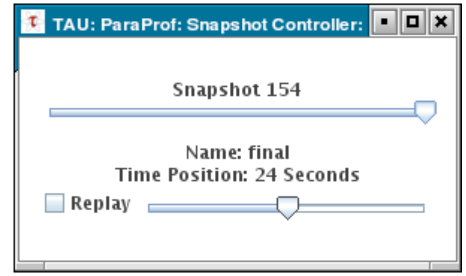
- Breakdown as a percentage

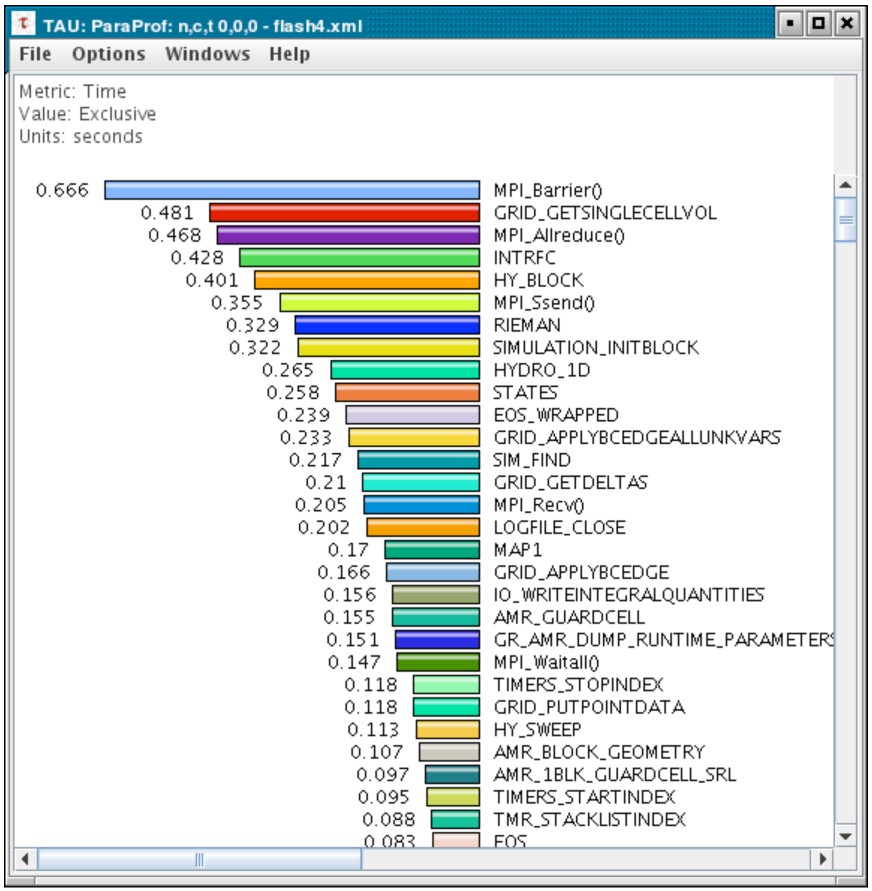


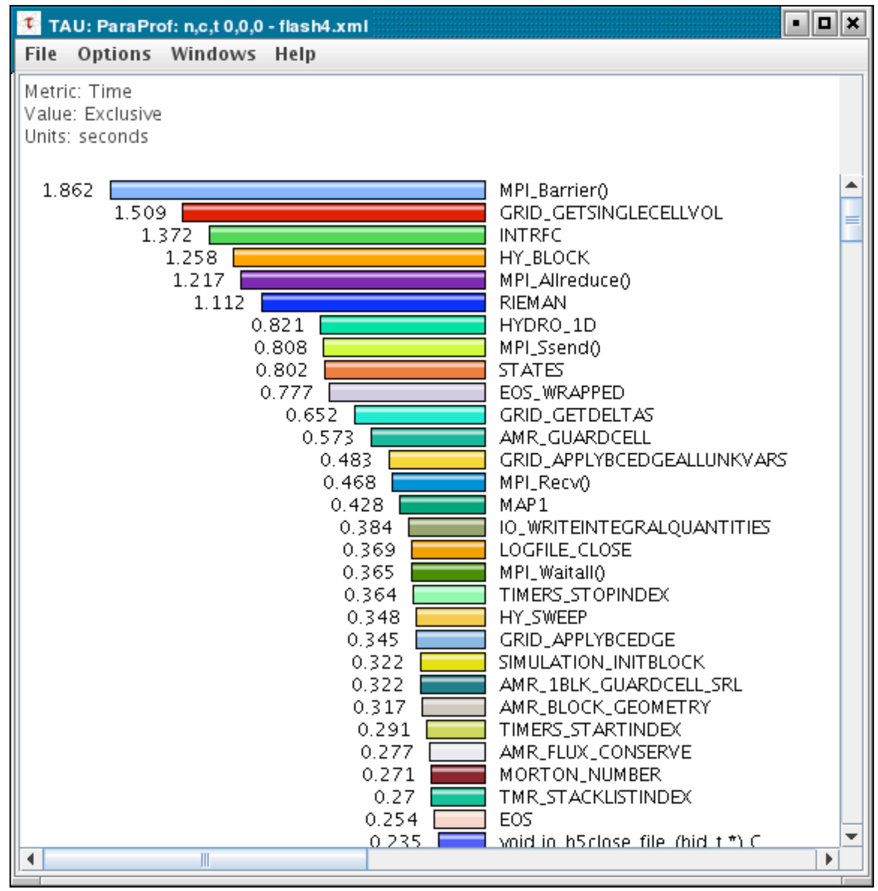
Snapshot replay in ParaProf



All windows dynamically update

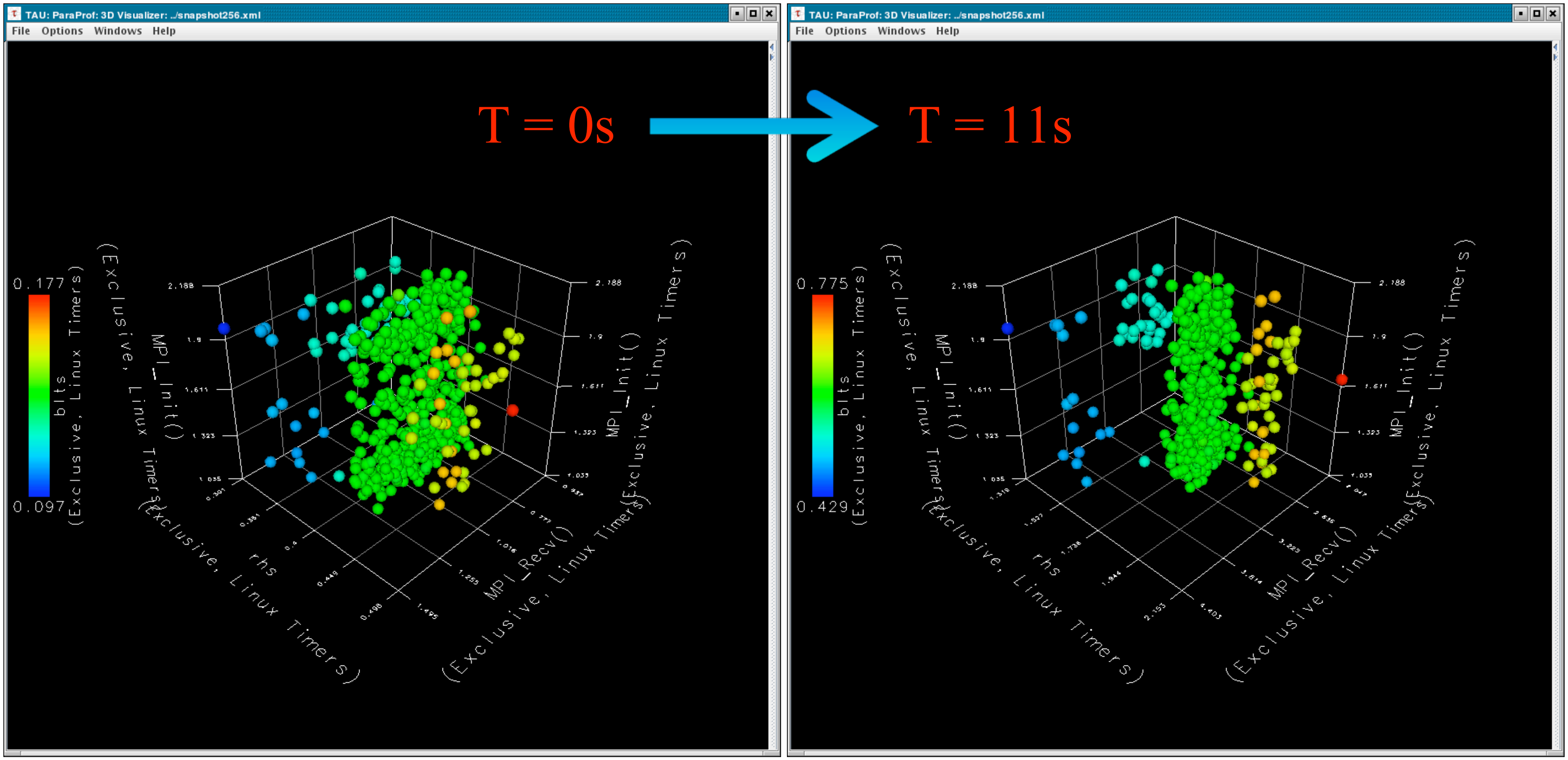




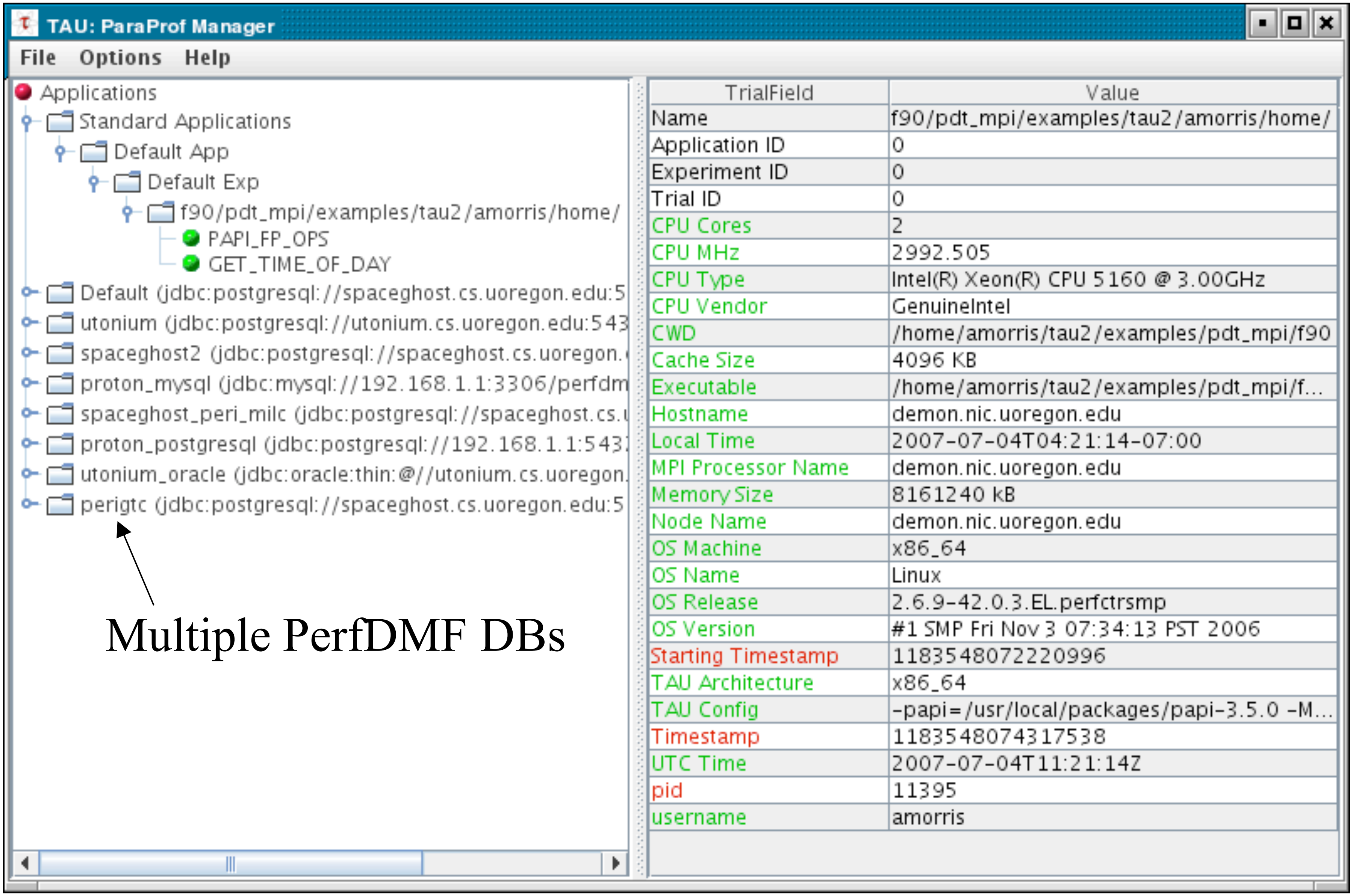


Profile Snapshots in ParaProf

- Follow progression of various displays through time
- 3D scatter plot shown below



New automated metadata collection



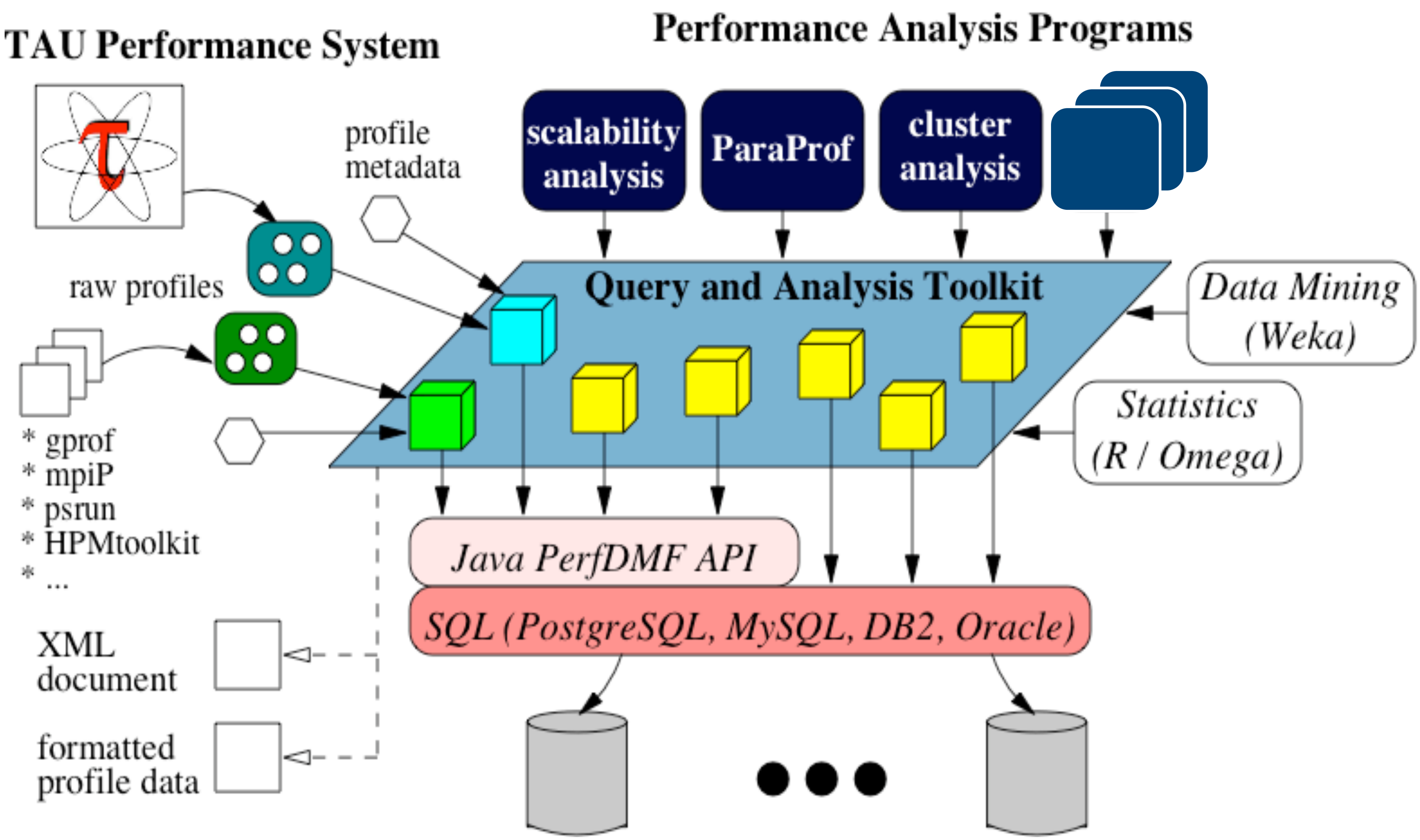
Performance Data Management: Motivation

- Need for robust processing and storage of multiple profile performance data sets
- Avoid developing independent data management solutions
  - Waste of resources
  - Incompatibility among analysis tools
- Goals:
  - Foster multi-experiment performance evaluation
  - Develop a common, reusable foundation of performance data storage, access and sharing
  - A core module in an analysis system, and/or as a central repository of performance data

PerfDMF Approach

- Performance Data Management Framework
- Originally designed to address critical TAU requirements
- Broader goal is to provide an open, flexible framework to support common data management tasks
- Extensible toolkit to promote integration and reuse across available performance tools
  - Supported profile formats:
    TAU, CUBE, Dynaprof, HPC Toolkit, HPM Toolkit, gprof, mpiP, psrun (PerfSuite), others in development
  - Supported DBMS:
    PostgreSQL, MySQL, Oracle, DB2, Derby/Cloudscape

PerfDMF Architecture



K. Huck, A. Malony, R. Bell, A. Morris, "**Design and Implementation of a Parallel Performance Data Management Framework**," ICPP 2005.

Recent PerfDMF Development

- Integration of XML metadata for each profile
  - Common Profile Attributes
  - Thread/process specific Profile Attributes
  - Automatic collection of runtime information
  - Any other data the user wants to collect can be added
    - Build information
    - Job submission information
  - Two methods for acquiring metadata:
    - TAU_METADATA() call from application
    - Optional XML file added when saving profile to PerfDMF
  - TAU Metadata XML schema is simple, easy to generate from scripting tools (no XML libraries required)

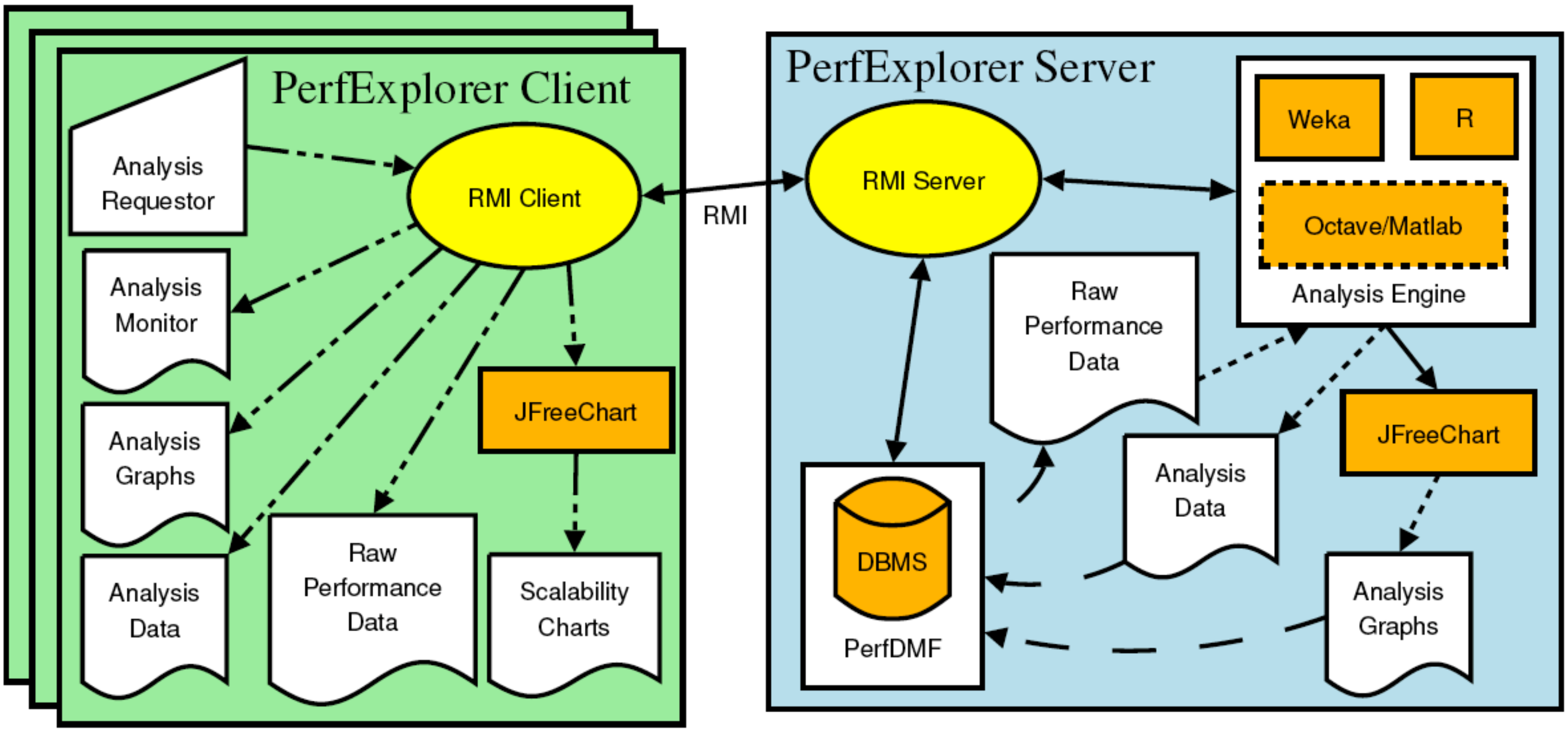
Performance Data Mining (Objectives)

- Conduct parallel performance analysis process
  - In a systematic, collaborative and reusable manner
  - Manage performance complexity
  - Discover performance relationship and properties
  - Automate process
- Multi-experiment performance analysis
- Large-scale performance data reduction
  - Summarize characteristics of large processor runs
- Implement extensible analysis framework
  - Abstraction / automation of data mining operations
  - Interface to existing analysis and data mining tools

Performance Data Mining (PerfExplorer)

- Performance knowledge discovery framework
  - Data mining analysis applied to parallel performance data
    - comparative, clustering, correlation, dimension reduction, …
  - Use the existing TAU infrastructure
    - TAU performance profiles, PerfDMF
  - Client-server based system architecture
- Technology integration
  - Java API and toolkit for portability
  - PerfDMF
  - R-project/Omegahat, Octave/Matlab statistical analysis
  - WEKA data mining package
  - JFreeChart for visualization, vector output (EPS, SVG)

Performance Data Mining (PerfExplorer)



K. Huck and A. Malony, "**PerfExplorer: A Performance Data Mining Framework For Large-Scale Parallel Computing**," SC 2005.

PerfExplorer Analysis Methods

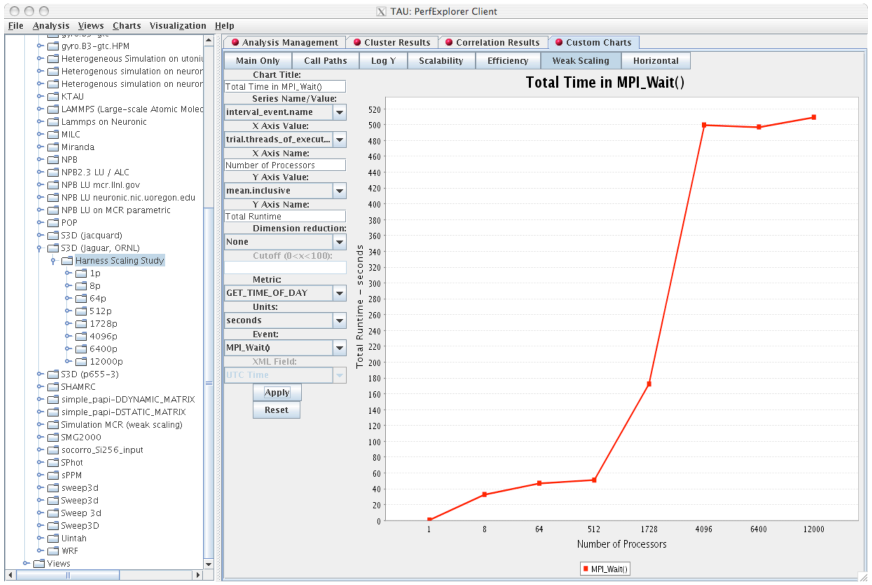
- Data summaries, distributions, scatterplots
- Clustering
  - *k*-means
  - Hierarchical
- Correlation analysis
- Dimension reduction
  - PCA
  - Random linear projection
  - Thresholds
- Comparative analysis
- Data management views

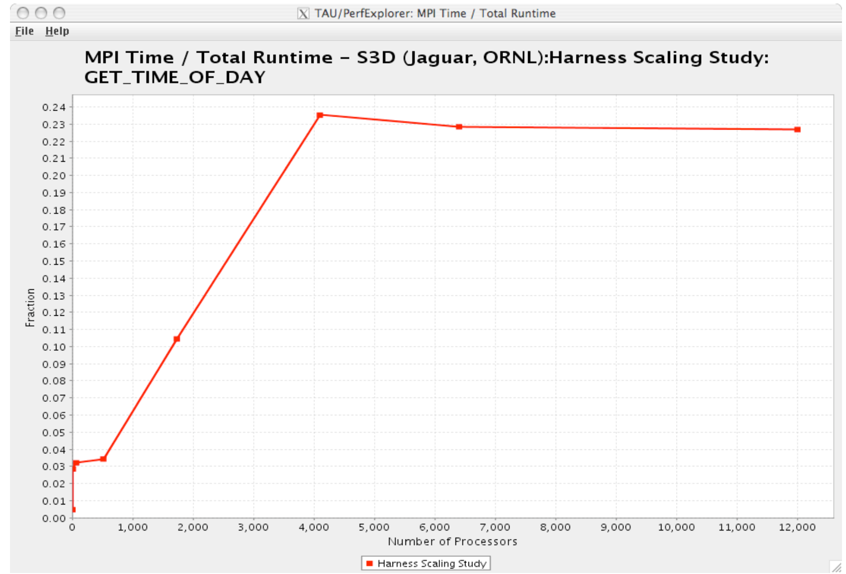
PerfDMF and the TAU Portal

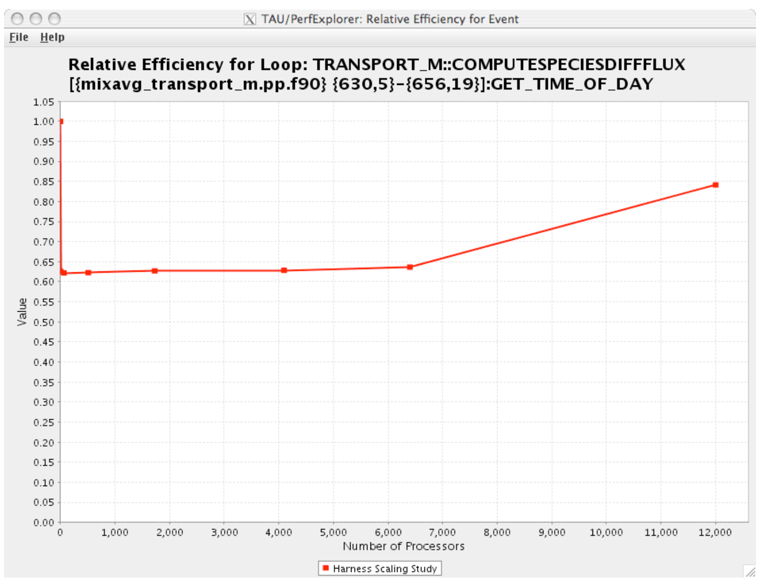
- Development of the TAU portal
  - Common repository for collaborative data sharing
  - Profile uploading, downloading, user management
  - Paraprof, PerfExplorer can be launched from the portal using Java Web Start (no TAU installation required)
- Portal URL

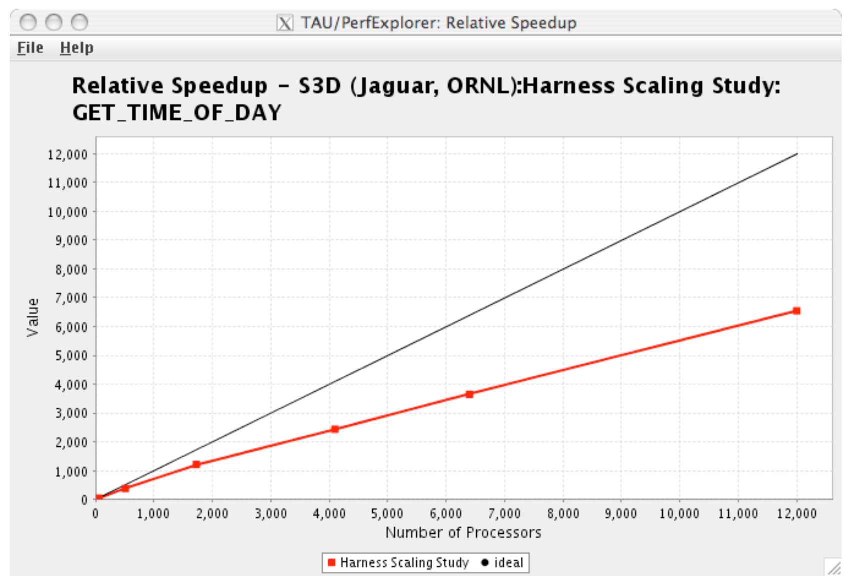
  http://tau.nic.uoregon.edu

PerfExplorer: Cross Experiment Analysis for S3D

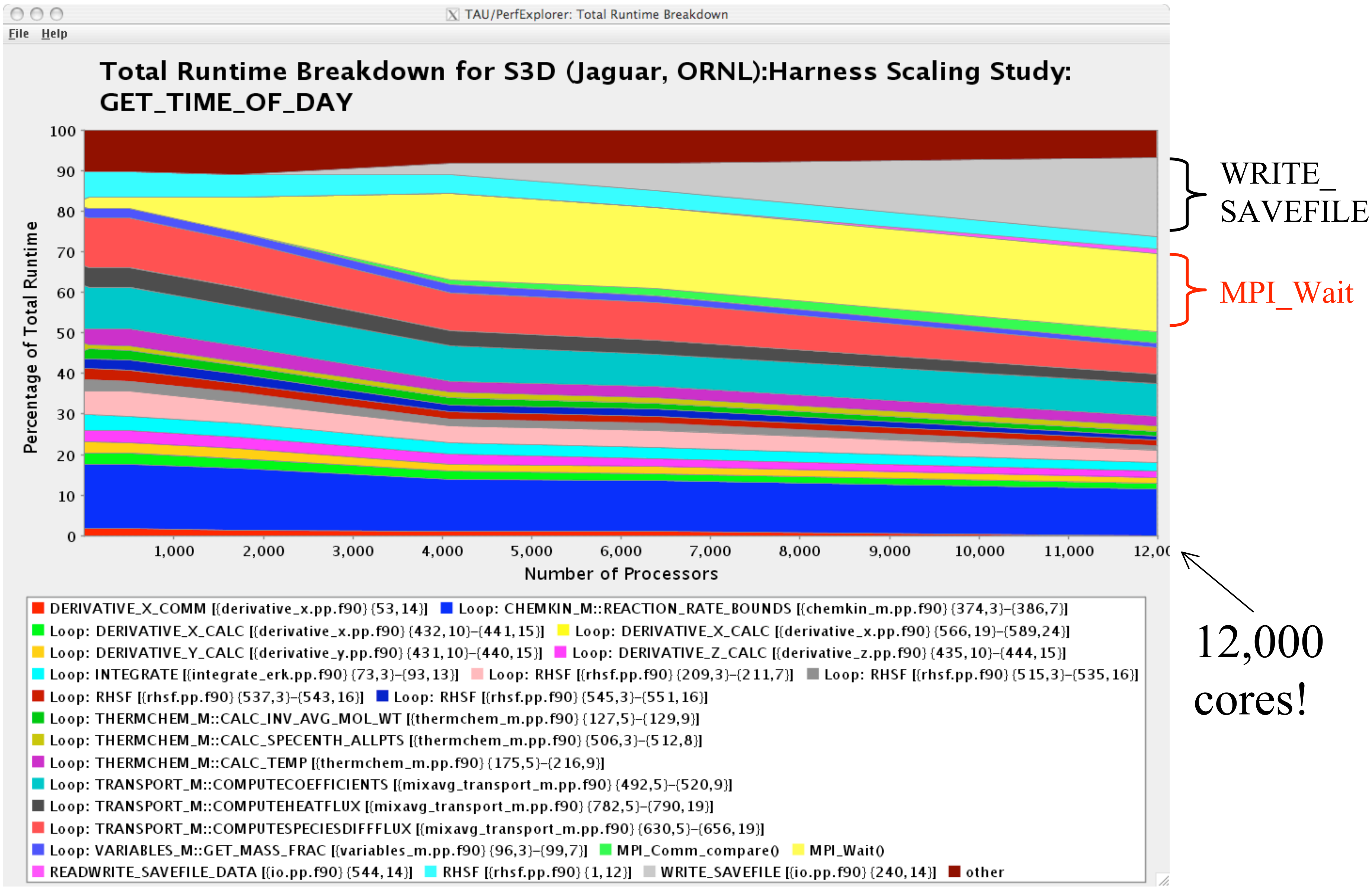








PerfExplorer: S3D Total Runtime Breakdown

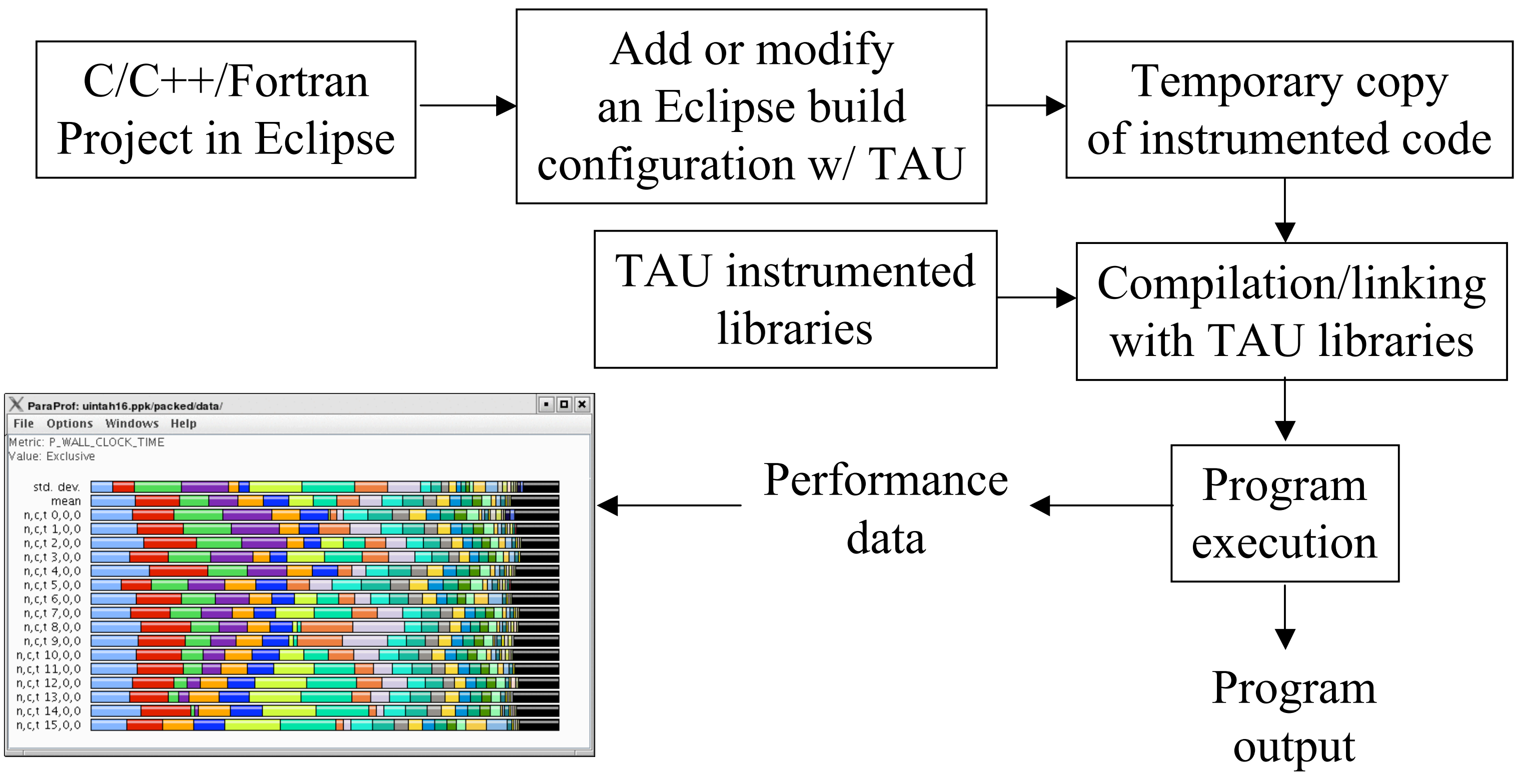


TAU Plug-Ins for Eclipse: Motivation

- High performance software development environments
  - Tools may be complicated to use
  - Interfaces and mechanisms differ between platforms / OS
- Integrated development environments
  - Consistent development environment
  - Numerous enhancements to development process
  - Standard in industrial software development
- Integrated performance analysis
  - Tools limited to single platform or programming language
  - Rarely compatible with 3rd party analysis tools
  - Little or no support for parallel projects

Adding TAU to Eclipse

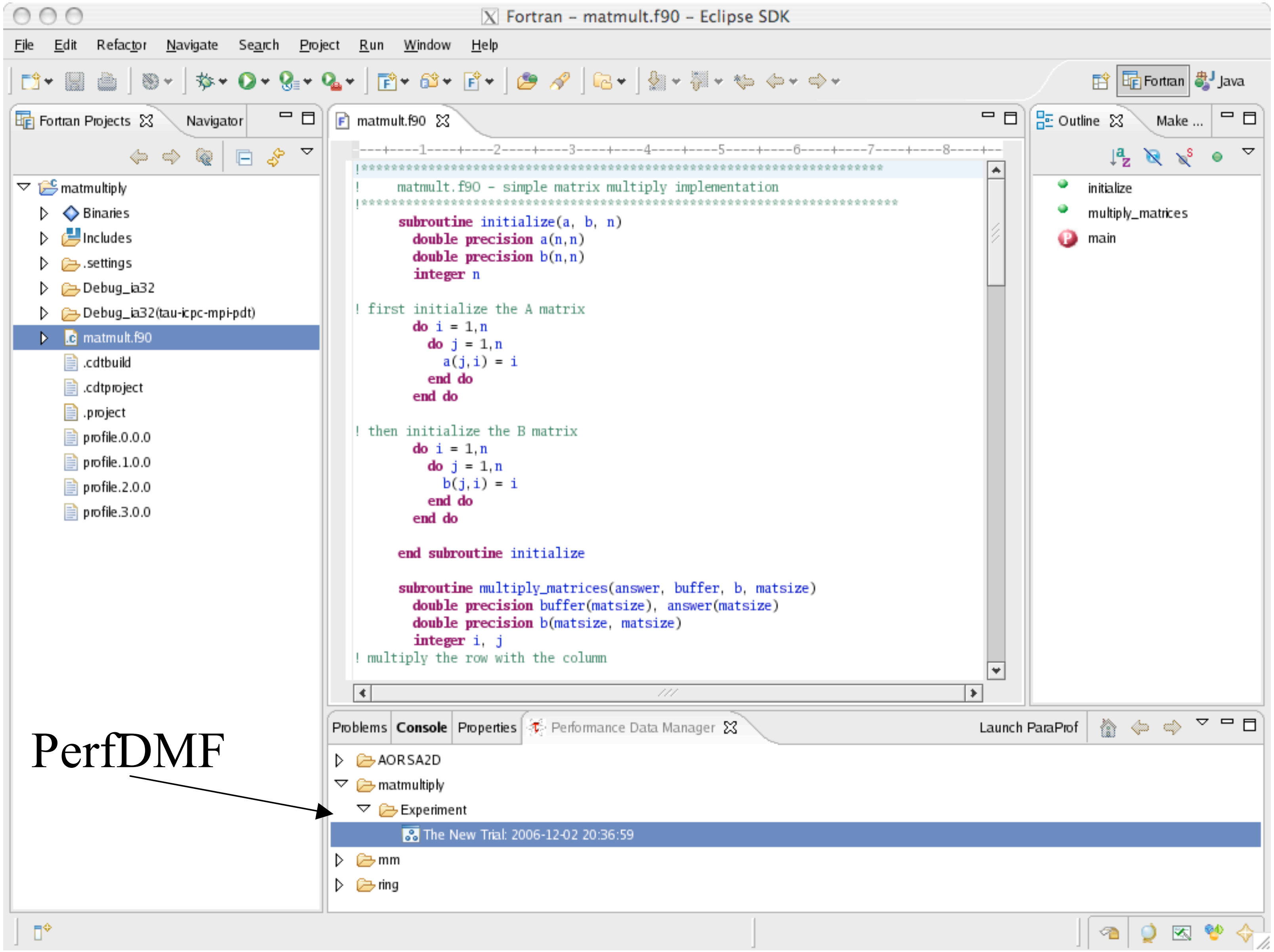
- Provide an interface for configuring TAU's automatic instrumentation within Eclipse's build system
- Manage runtime configuration settings and environment variables for execution of TAU instrumented programs



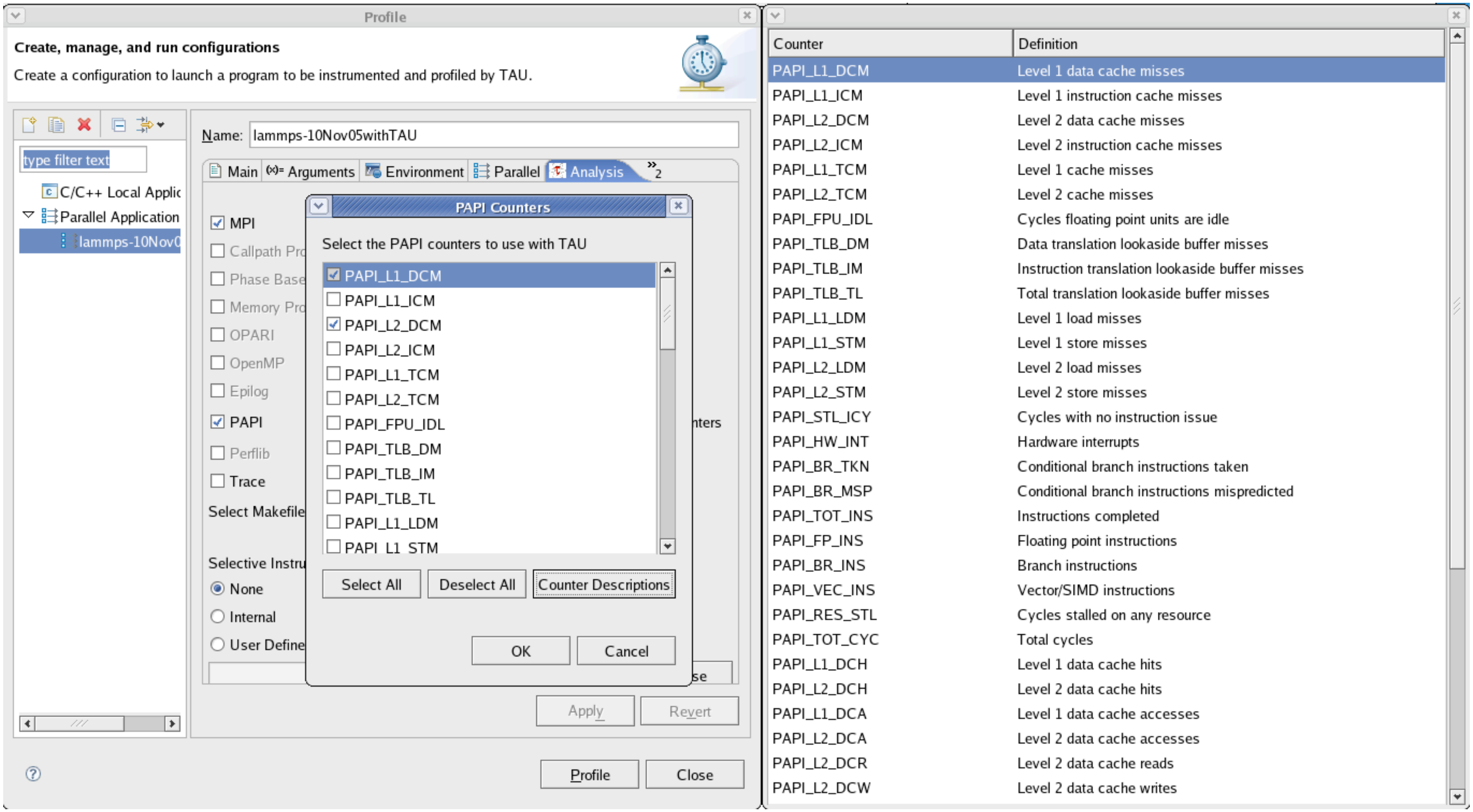
TAU Eclipse Plug-In Features

- Performance data collection
  - Graphical selection of TAU stub makefiles and compiler options
  - Automatic instrumentation, compilation and execution of target C, C++ or Fortran projects
  - Selective instrumentation via source editor and source outline views
  - Full integration with the Parallel Tools Platform (PTP) parallel launch system for performance data collection from parallel jobs launched within Eclipse
- Performance data management
  - Automatically place profile output in a PerfDMF database or upload to TAU-Portal
  - Launch ParaProf on profile data collected in Eclipse, with performance counters linked back to the Eclipse source editor

TAU Eclipse Plug-In Features



Choosing PAPI Counters with TAU's in Eclipse



Future Plug-In Development

- Integration of additional TAU components
  - Automatic selective instrumentation based on previous experimental results
  - Trace format conversion from within Eclipse
- Trace and profile visualization within Eclipse
- Scalability testing interface
- Additional user interface enhancements

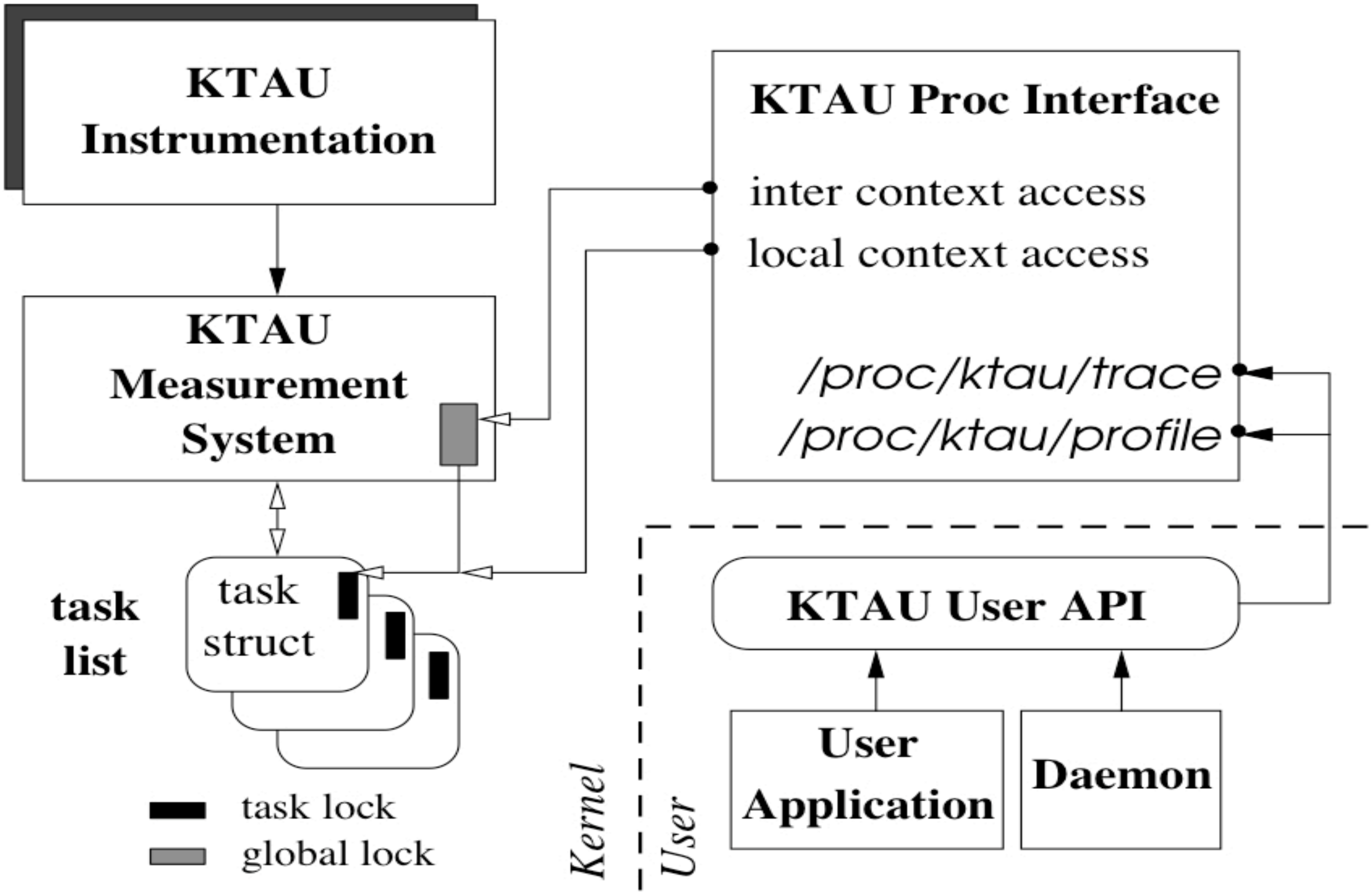
KTAU Project

- Trend toward Extremely Large Scales
  - System-level influences are increasingly dominant performance bottleneck contributors
  - Application sensitivity at scale to the system (e.g., OS noise)
  - Complex I/O path and subsystems another example
  - Isolating system-level factors non-trivial
- OS Kernel instrumentation and measurement is important to understanding system-level influences
- But can we closely correlate observed application and OS performance?
- KTAU / TAU (Part of the ANL/UO ZeptoOS Project)
  - Integrated methodology and framework to measure whole-system performance

Applying KTAU+TAU

- How does ***real*** OS-noise affect ***real*** applications on target platforms?
  - Requires a tightly coupled performance measurement & analysis approach provided by KTAU+TAU
  - Provides an estimate of application slowdown due to Noise (and in particular, different noise-components - IRQ, scheduling, etc)
  - Can empower both application and the middleware and OS communities.
  - A. Nataraj, A. Morris, A. Malony, M. Sottile, P. Beckman, "The Ghost in the Machine : Observing the Effects of Kernel Operation on Parallel Application Performance", SC'07.
- Measuring and analyzing complex, multi-component I/O subsystems in systems like BG(L/P) (work in progress).

KTAU System Architecture



A. Nataraj, A. Malony, S. Shende, and A. Morris, "**Kernel-level Measurement for Integrated Performance Views: the KTAU Project**," *Cluster 2006*, distinguished paper.

Support Acknowledgements

- US Department of Energy (DOE)
  - Office of Science
    - MICS, Argonne National Lab
  - ASC/NNSA
    - University of Utah ASC/NNSA Level 1
    - ASC/NNSA, Lawrence Livermore National Lab
- US Department of Defense (DoD)
- NSF Software and Tools for High-End Computing
- Research Centre Juelich
- TU Dresden
- Los Alamos National Laboratory
- ParaTools, Inc.

TAU Transport Substrate - Motivations

- Transport Substrate
  - Enables movement of measurement-related data
  - TAU, in the past, has relied on shared file-system
- Some Modes of Performance Observation
  - Offline / Post-mortem observation and analysis
    - least requirements for a specialized transport
  - Online observation
    - long running applications, especially at scale
    - dumping to file-system can be suboptimal
  - Online observation with feedback into application
    - in addition, requires that the transport is bi-directional
- Performance observation problems and requirements are a function of the mode

Requirements

- Improve performance of transport
  - NFS can be slow and variable
  - Specialization and remoting of FS-operations to front-end
- Data Reduction
  - At scale, cost of moving data too high
  - Sample in different domain (node-wise, event-wise)
- Control
  - Selection of events, measurement technique, target nodes
  - What data to output, how often and in what form?
  - Feedback into the measurement system, feedback into application
- Online, distributed processing of generated performance data
  - Use compute resource of transport nodes
  - Global performance analyses within the topology
  - Distribute statistical analyses
- Scalability, most important - All of above at very large scales

Approach and Prototypes

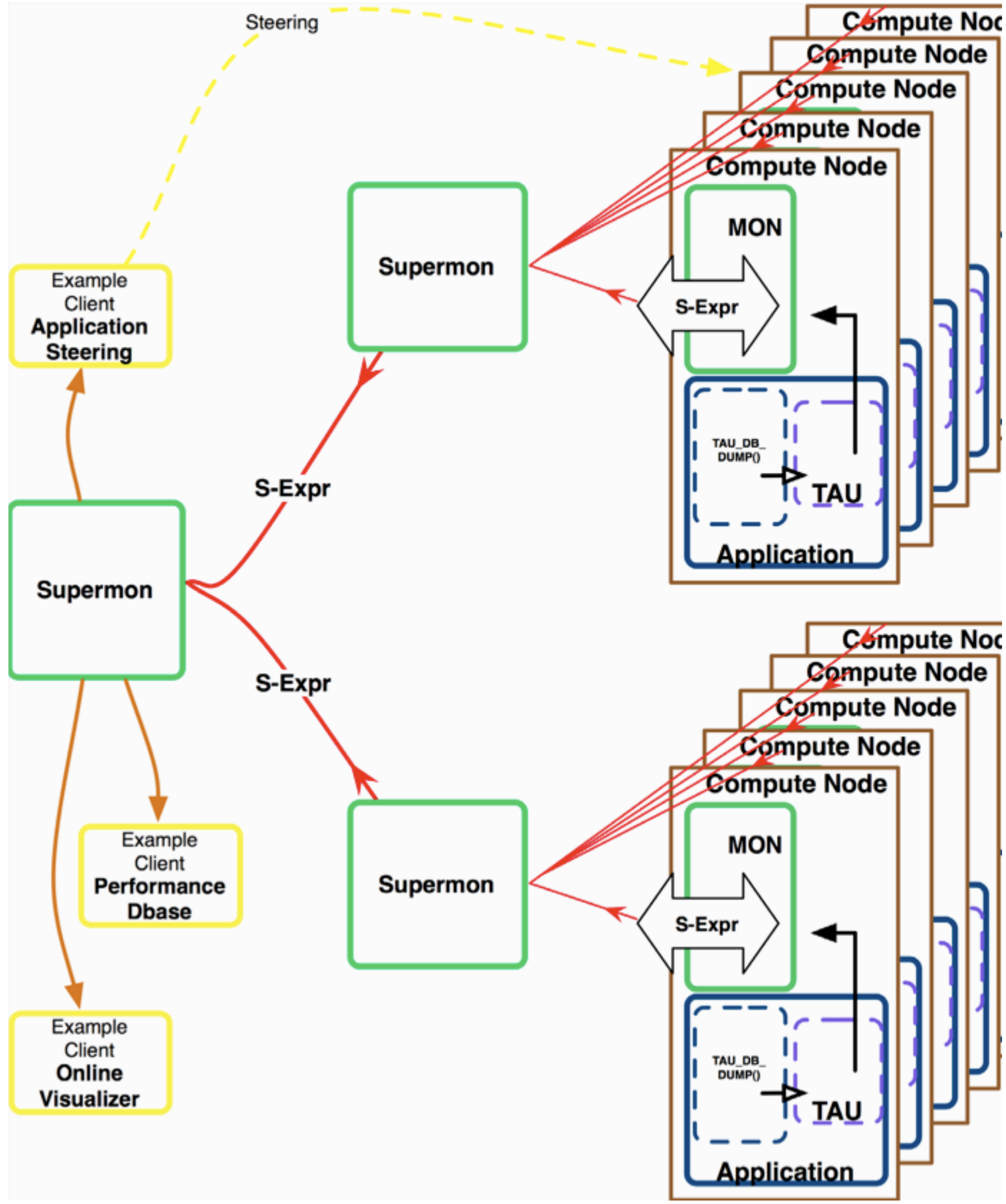
- Measurement and measured data transport de-coupled
  - Earlier, no such clear distinction in TAU
- Created abstraction to separate and hide transport
  - *TauOutput*
- Did not create a custom transport for TAU(as yet)
  - Use existing monitoring/transport capabilities
- TAUover: Supermon (Sottile and Minnich, LANL) and MRNET (Arnold and Miller, UWisc)
- A. Nataraj, M.Sottile, A. Morris, A. Malony, S. Shende "TAUoverSupermon: Low-overhead Online Parallel Performance Monitoring", Europar'07.

Rationale

- Moved away from NFS
- Separation of concerns
  - Scalability, portability, robustness
  - Addressed independent of TAU
- Re-use existing technologies where appropriate
- Multiple bindings
  - Use different solutions best suited to particular platform
- Implementation speed
  - Easy, fast to create adapter that binds to existing transport

Substrate Architecture - High-level

- Components
  - Front-End (FE)
  - Intermediate Nodes
  - Back-End (BE)
- NFS, Supermon, MRNet API
- Push-Pull model of data retrieval
- Figure shows ***ToS*** high-level view



Substrate Architecture - Back-End

- Application calls into TAU
  - Per-Iteration explicit call to output routine
  - Periodic calls using alarm
- TauOutput object invoked
  - Configuration specific: compile or runtime
  - One per thread
- TauOutput mimics subset of FS-style operations
  - Avoids changes to TAU code
  - If required rest of TAU can be made aware of output type
- Non-blocking *recv* for control
- Back-end pushes, Sink pulls