TAU Performance System

(ACTS Workshop LBL)

Sameer Shende, Allen D. Malony

University of Oregon

{sameer, malony}@cs.uoregon.edu

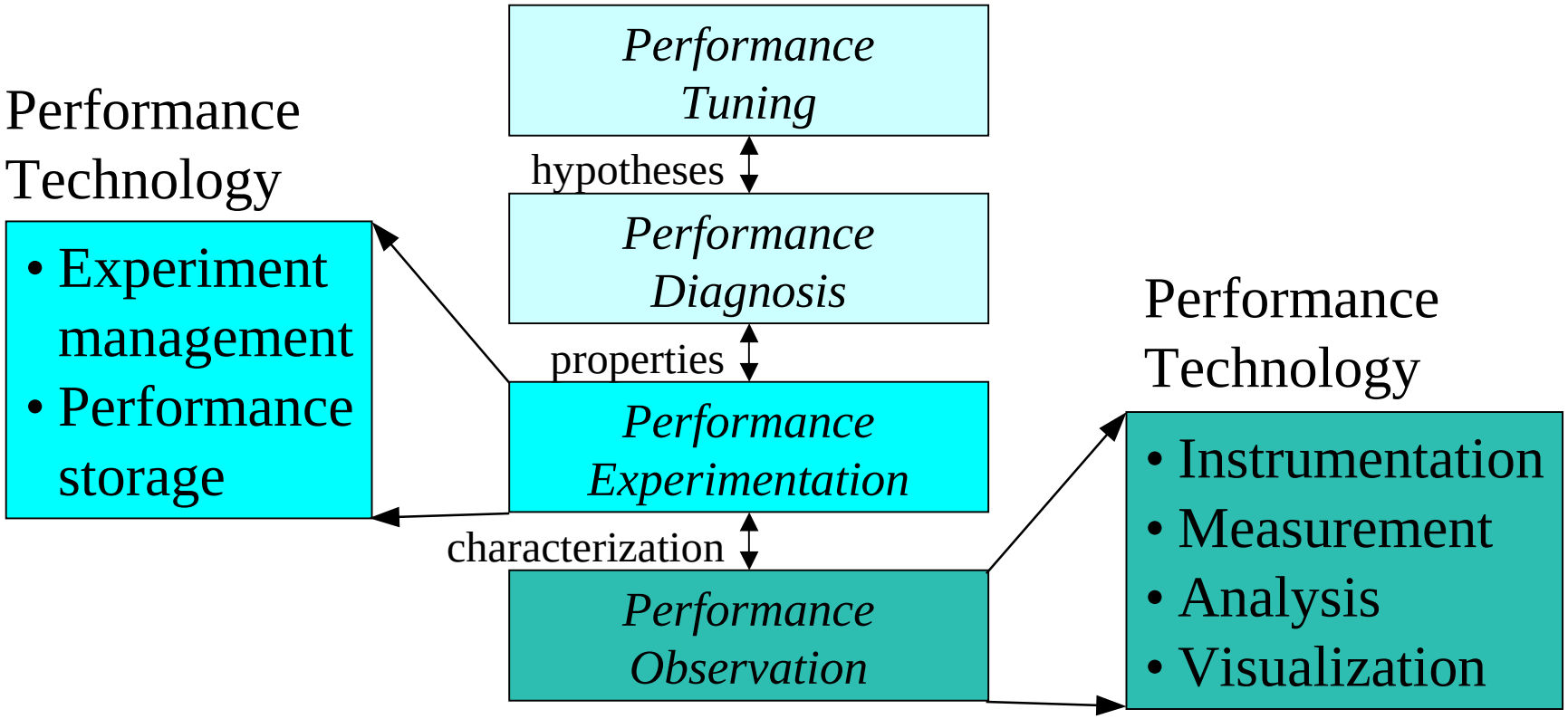
UNIVERSITY OF OREGON

Los Alamos NATIONAL LABORATORY

John von Neumann - Institut für Computing
Zentralinstitut für Angewandte Mathematik
NIC

Research Motivation

- Tools for performance problem solving
  - Empirical-based performance optimization process
  - Performance technology concerns



Outline of Talk

- Performance problem solving
  - Scalability, productivity, and performance technology
  - Application-specific and autonomic performance tools
- TAU parallel performance system and advances
- Performance data management and data mining
  - Performance Data Management Framework (PerfDMF)
  - PerfExplorer
- Multi-experiment case studies
  - Clustering analysis
- Future work and concluding remarks

TAU Performance System

- *T*uning and *A*nalysis *U*tilities (13+ 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
- University of Oregon , Research Center Jülich, LANL

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

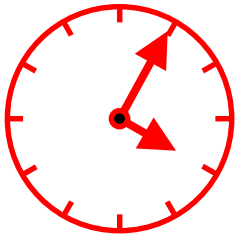
Event Tracing: Instrumentation, Monitor, Trace

CPU A:

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

CPU B:

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



timestamp

MONITOR

Event definition

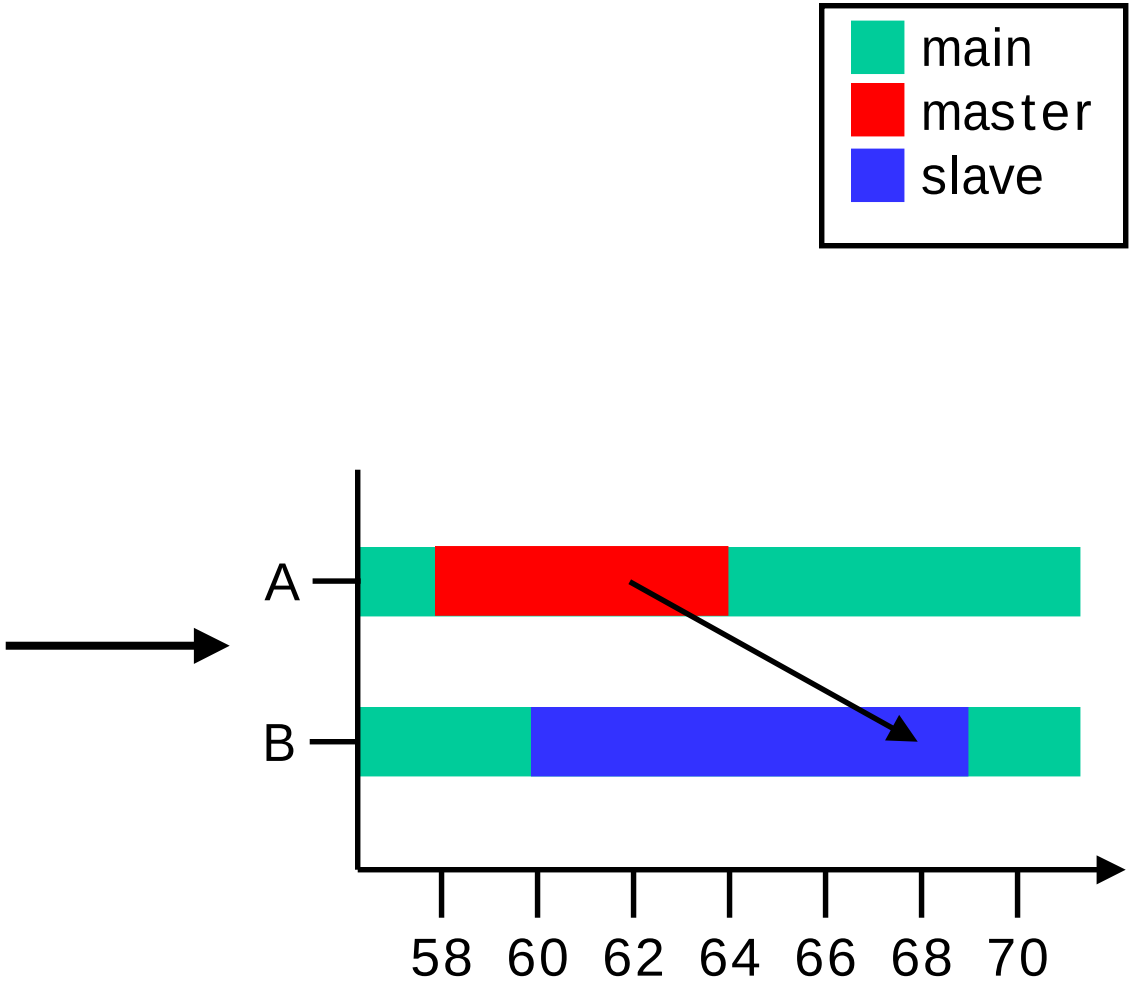
1	master
2	slave
3	...

...			
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	slave
3	...

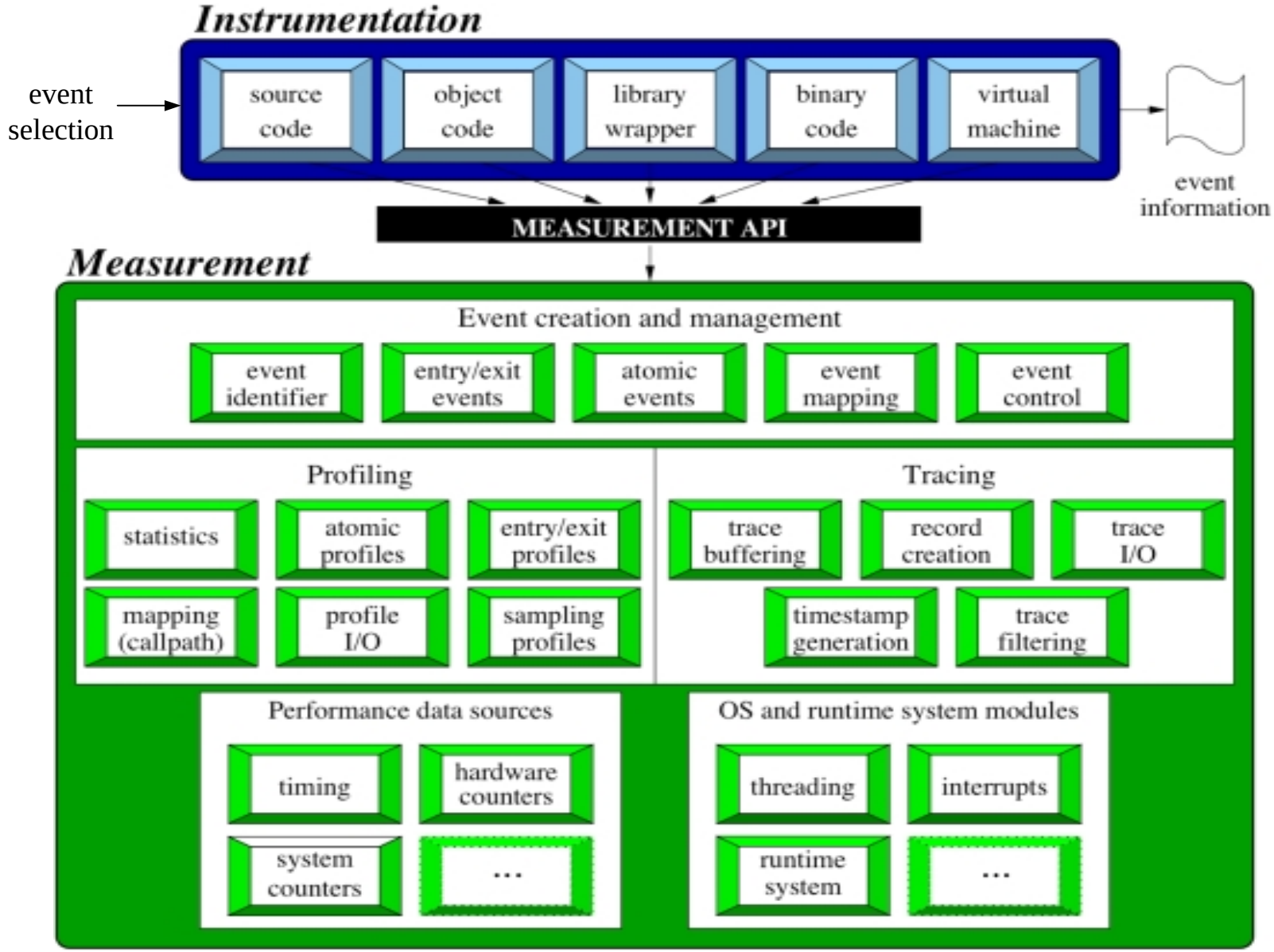
...			
58	A	ENTER	1
60	B	ENTER	2
62	A	SEND	B
64	A	EXIT	1
68	B	RECV	A
69	B	EXIT	2
...			



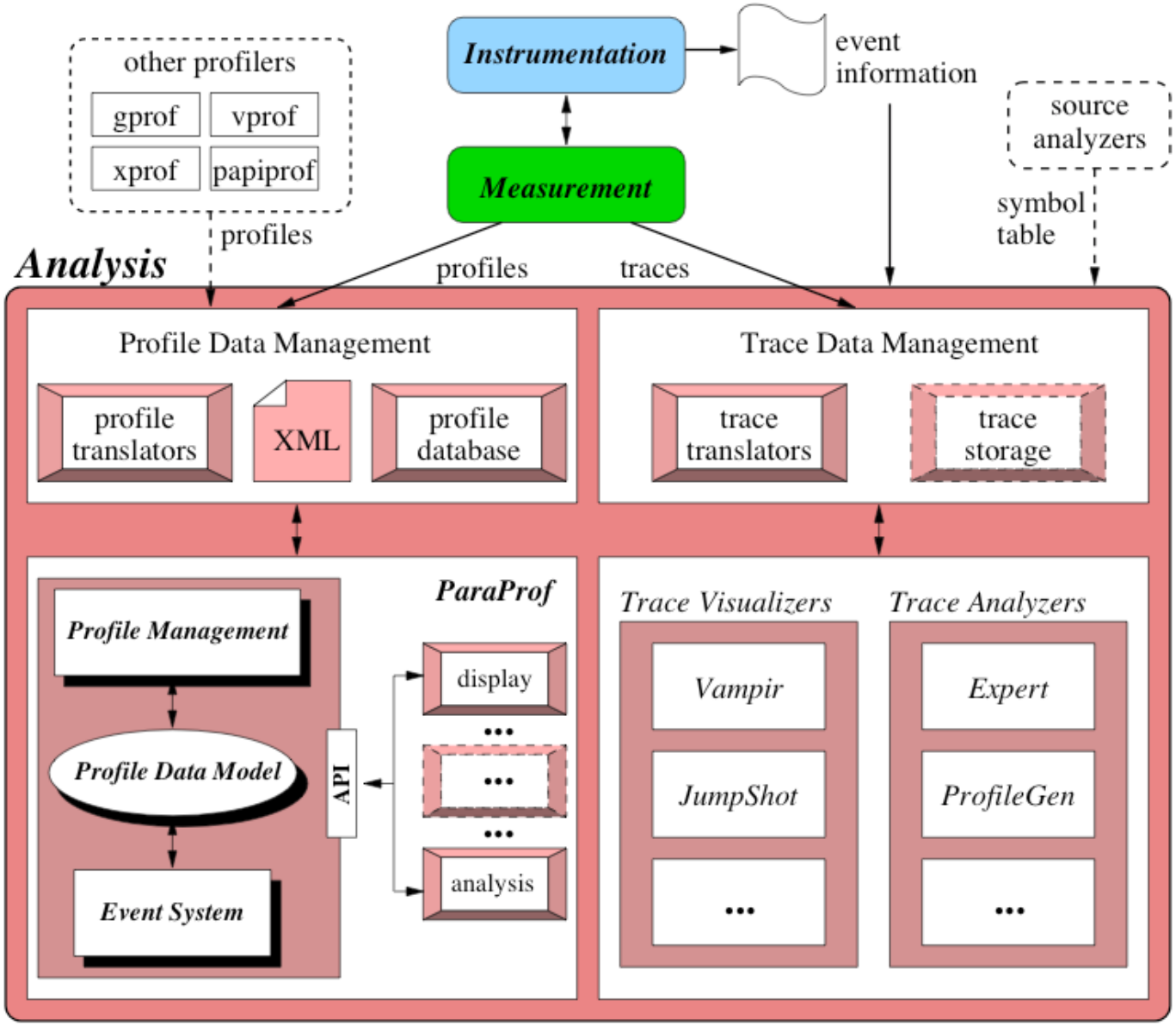
TAU Parallel Performance System Goals

- Multi-level performance instrumentation
  - Multi-language automatic source instrumentation
- Flexible and configurable performance measurement
- Widely-ported parallel performance profiling system
  - Computer system architectures and operating systems
  - Different programming languages and compilers
- Support for multiple parallel programming paradigms
  - Multi-threading, message passing, mixed-mode, hybrid
- Support for performance mapping
- Support for object-oriented and generic programming
- Integration in complex software, systems, applications

TAU Performance System Architecture



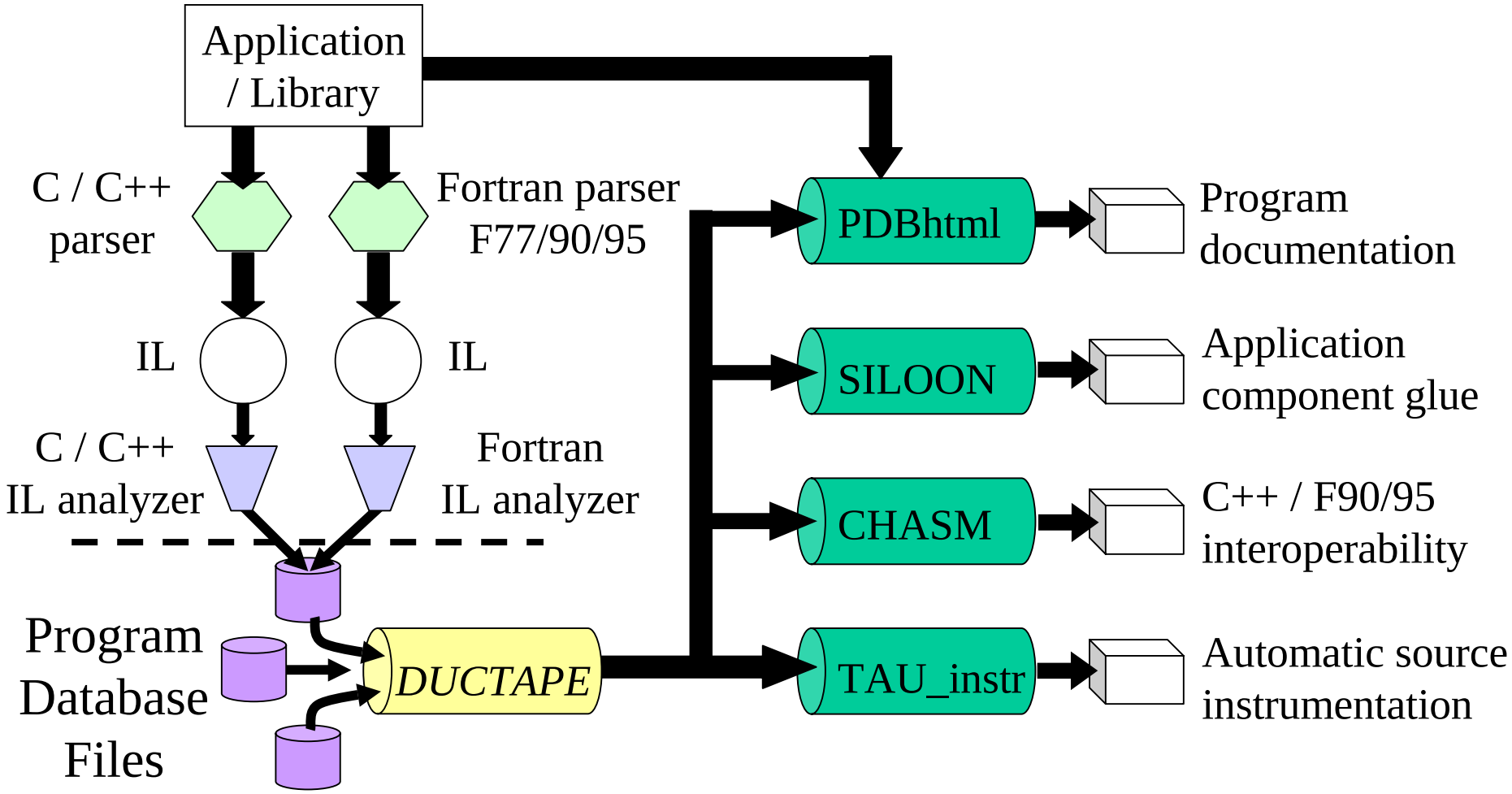
TAU Performance System Architecture



Advances in TAU Instrumentation

- Source instrumentation
  - *Program Database Toolkit* (*PDT*)
    - automated Fortran 90/95 support (Cleanscape Flint parser)
    - statement level support in C/C++ (Fortran soon)
  - *TAU_COMPILER* to automate instrumentation process
  - Automatic proxy generation for component applications
    - automatic CCA component instrumentation
  - Python instrumentation and automatic instrumentation
- Continued integration with dynamic instrumentation
- Update of OpenMP instrumentation (POMP2)
- Selective instrumentation and overhead reduction
- Improvements in performance mapping instrumentation

Program Database Toolkit (PDT)



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 definition of “semantic” entities for mapping
- Support for event groups
- Instrumentation optimization (eliminate instrumentation in lightweight routines)

TAU Instrumentation

- Flexible instrumentation mechanisms at multiple levels
  - Source code
    - manual (TAU API, TAU Component API)
    - automatic
      - C, C++, F77/90/95 (Program Database Toolkit (*PDT*))
      - OpenMP (directive rewriting (*Opari*), *POMP 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*)
  - Proxy Components

Using TAU – A tutorial

- Configuration
- Instrumentation
  - Manual
  - MPI – Wrapper interposition library
  - PDT- Source rewriting for C,C++, F77/90/95
  - OpenMP – Directive rewriting
  - Component based instrumentation – Proxy components
  - Binary Instrumentation
    - DyninstAPI – Runtime Instrumentation/Rewriting binary
    - Java – Runtime instrumentation
    - Python – Runtime instrumentation
- Measurement
- Performance Analysis

TAU Measurement System Configuration

- configure [OPTIONS]

{-c++=<CC>, -cc=<cc>}	Specify C++ and C compilers
{-pthread, -sproc}	Use pthread or SGI sproc threads
-openmp	Use OpenMP threads
-jdk=<dir>	Specify Java instrumentation (JDK)
-opari=<dir>	Specify location of Opari OpenMP tool
-papi=<dir>	Specify location of PAPI
-pdt=<dir>	Specify location of PDT
-dyninst=<dir>	Specify location of DynInst Package
-mpi[inc/lib]=<dir>	Specify MPI library instrumentation
-shmem[inc/lib]=<dir>	Specify PSHMEM library instrumentation
-python[inc/lib]=<dir>	Specify Python instrumentation
-epilog=<dir>	Specify location of EPILOG
-slog2[=<dir>]	Specify location of SLOG2/Jumpshot
-vtf=<dir>	Specify location of VTF3 trace package
-arch=<architecture>	Specify architecture explicitly (bgl,ibm64,ibm64linux…)

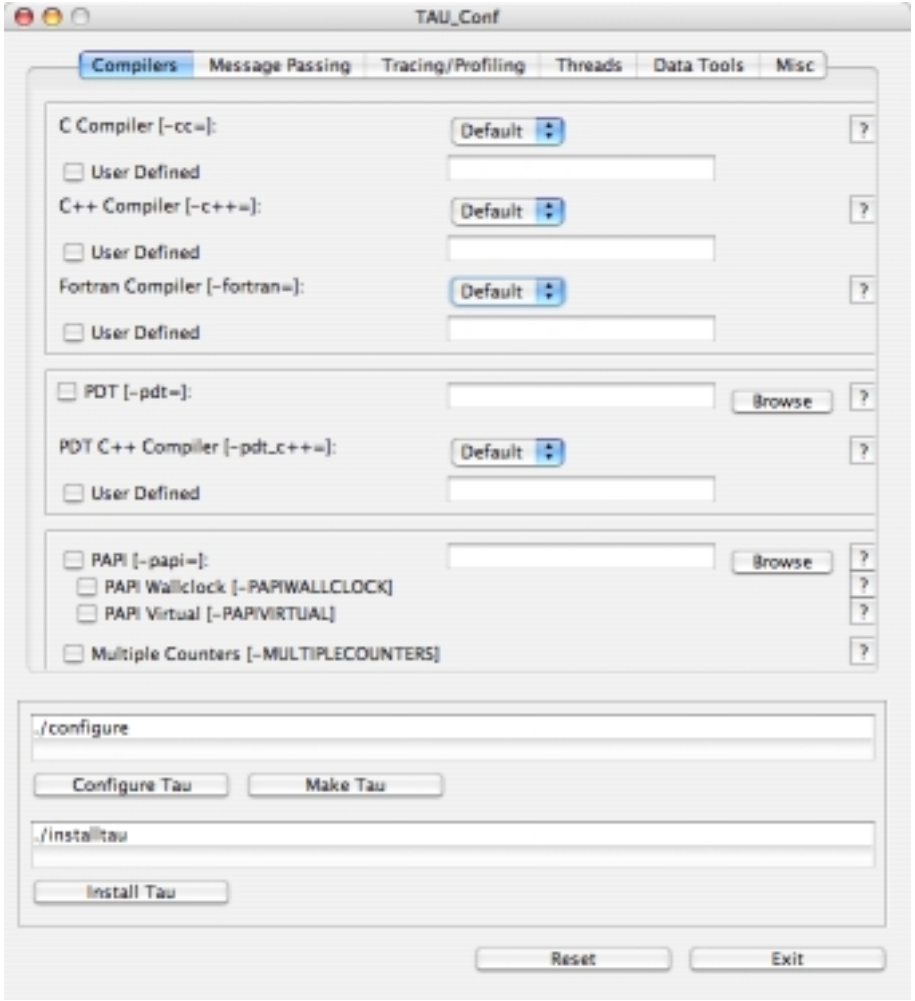
TAU Measurement System Configuration

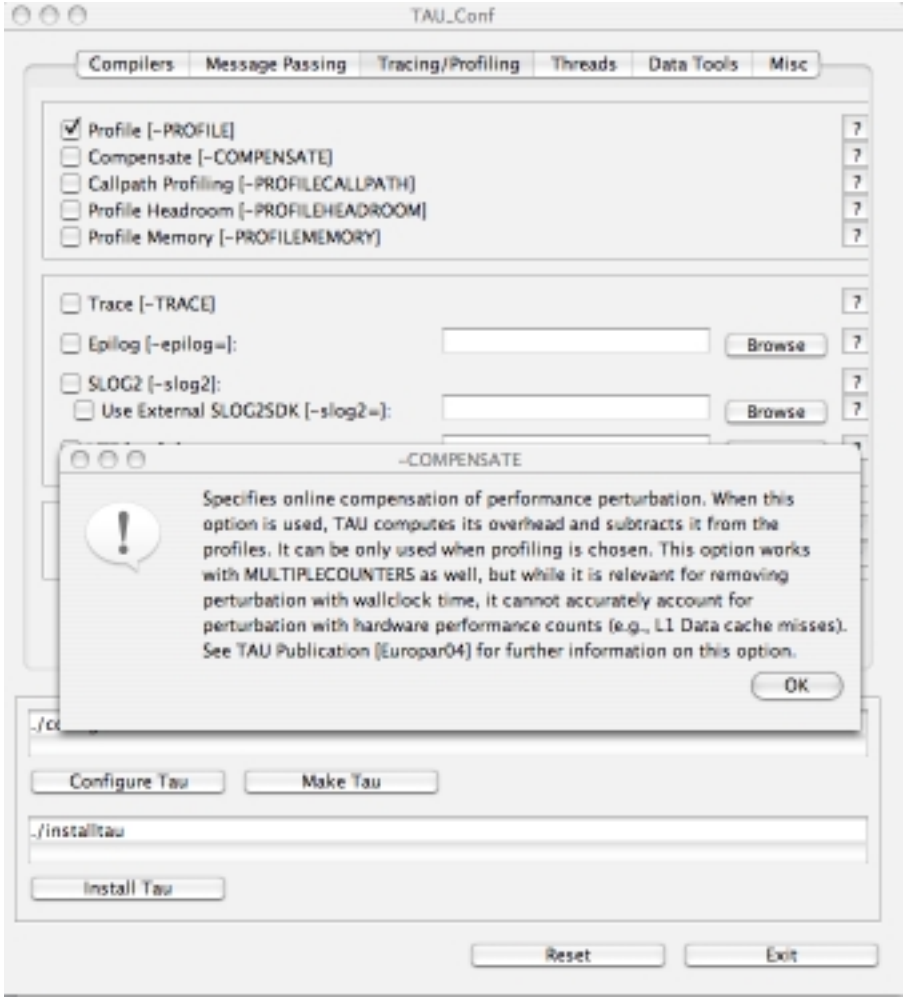
- configure [OPTIONS]
  - -TRACE — Generate binary TAU traces
  - -PROFILE (default) — Generate profiles (summary)
  - -PROFILECALLPATH — Generate call path profiles
  - -PROFILEPHASE — Generate phase based profiles
  - -PROFILEMEMORY — Track heap memory for each routine
  - -PROFILEHEADROOM — Track 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

TAU Measurement Configuration – Examples

- ./configure -c++=xlC_r –pthread
  - Use TAU with xlC_r and pthread library under AIX
  - Enable TAU profiling (default)
- ./configure -TRACE –PROFILE
  - Enable both TAU profiling and tracing
- ./configure -c++=xlC_r -cc=xlc_r
  -papi=/usr/local/packages/papi
  -pdt=/usr/local/pdtoolkit-3.4 –arch=ibm64
  -mpiinc=/usr/lpp/ppe.poe/include
  -mpilib=/usr/lpp/ppe.poe/lib -MULTIPLECOUNTERS
  - Use IBM's xlC_r and xlc_r compilers with PAPI, PDT, MPI packages and multiple counters for measurements
- Typically configure multiple measurement libraries
- Each configuration creates a unique <arch>/lib/Makefile.tau-<options> stub makefile that corresponds to the configuration options specified. E.g.,
  - /san/cca/tau/tau-2.14.7/x86_64/lib/Makefile.tau-icpc-mpi-pdt
  - /san/cca/tau/tau-2.14.7/x86_64/lib/Makefile.tau-icpc-mpi-pdt-trace

TAU_SETUP: A GUI for Installing TAU






```
tau-2.x>./tau_setup
```

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

Note: Not including TAU_DEFS in CFLAGS disables instrumentation in C/C++ programs (TAU_DISABLE for f90).

Using TAU

Step 1: Configure and install TAU:

```
% configure -pdt=<dir> -mpiinc=<dir> -mpilib=<dir>
  -c++=icpc -cc=icc -fortran=intel
% make clean; make install
```

Builds <taudir>/<arch>/lib/Makefile.tau-<options>

```
% set path=($path <taudir>/<arch>/bin)
```

Step 2: Choose target stub Makefile

```
% setenv TAU_MAKEFILE
/san/cca/tau/tau-2.14.7/x86_64/lib/Makefile.tau-icpc-mpi-pdt
% setenv TAU_OPTIONS '-optVerbose -optKeepFiles'
(see tau_compiler.sh for all options)
```

Step 3: Use tau_f90.sh, tau_cxx.sh and tau_cc.sh as the F90, C++ or C compilers respectively.

```
% tau_f90.sh -c app.f90
% tau_f90.sh app.o -o app -lm -lblas
```

Or use these in the application Makefile.

AutoInstrumentation using TAU_COMPILER

- $(TAU_COMPILER) stub Makefile variable in 2.14+ release
- Invokes PDT parser, TAU instrumentor, compiler through **`tau_compiler.sh`** shell script
- Requires minimal changes to application Makefile
  - Compilation rules are not changed
  - User sets TAU_MAKEFILE and TAU_OPTIONS environment variables
  - User renames the compilers
    - F90=xlf90

    to
    - F90= tau_f90.sh
- Passes options from TAU stub Makefile to the four compilation stages
- Uses original compilation command if an error occurs

Tau_[cxx,cc,f90].sh – Improves Integration in Makefiles

OLD

```
include /usr/tau-2.14/include/Makefile
CXX = mpCC
F90 = mpxlf90_r
PDTPARSE = $(PDTDIR)/
         $(PDTARCHDIR)/bin/cxxparse
TAUINSTR = $(TAUROOT)/$(CONFIG_ARCH)/
         bin/tau_instrumentor
CFLAGS = $(TAU_DEFS) $(TAU_INCLUDE)
LIBS = $(TAU_MPI_LIBS) $(TAU_LIBS) -lm
OBJS = f1.o f2.o f3.o … fn.o

app: $(OBJS)
      $(CXX) $(LDFLAGS) $(OBJS) -o $@
      $(LIBS)
.cpp.o:
      $(PDTPARSE) $<
      $(TAUINSTR) $*.pdb $< -o
         $*.i.cpp –f select.dat
      $(CC) $(CFLAGS) -c $*.i.cpp
```

NEW

```
# set TAU_MAKEFILE and TAU_OPTIONS env vars
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 Options

- Optional parameters for $(TAU_COMPILER):
  - -optVerbose Turn on verbose debugging messages
  - -optPdtDir="" PDT architecture directory. Typically $(PDTDIR)/$(PDTARCHDIR)
  - -optPdtF95Opts="" Options for Fortran parser in PDT (f95parse)
  - -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)
  - -optPdtF90Parser="" Specify a different Fortran parser. For e.g., f90parse instead of f95parse
  - -optPdtUser="" Optional arguments for parsing source code
  - -optPDBFile="" Specify [merged] PDB file. Skips parsing phase.
  - -optTauInstr="" Specify location of tau_instrumentor. Typically $(TAUROOT)/$(CONFIG_ARCH)/bin/tau_instrumentor
  - -optTauSelectFile="" Specify selective instrumentation file for tau_instrumentor
  - -optTau="" Specify options for tau_instrumentor
  - -optCompile="" Options passed to the compiler. Typically $(TAU_MPI_INCLUDE) $(TAU_INCLUDE) $(TAU_DEFS)
  - -optLinking="" Options passed to the linker. Typically $(TAU_MPI_FLIBS) $(TAU_LIBS) $(TAU_CXXLIBS)
  - -optNoMpi Removes -l*mpi* libraries during linking (default)
  - -optKeepFiles Does not remove intermediate .pdb and .inst.* files

```
e.g.,
% setenv TAU_OPTIONS '-optTauSelectFile=select.tau –
   optVerbose -optPdtCOpts="-I/home -DFOO" '
% tau_cxx.sh matrix.cpp -o matrix -lm
```

Instrumentation Specification

```
% tau_instrumentor
Usage : tau_instrumentor <pdbfile> <sourcefile> [-o <outputfile>] [-noinline]
[-g groupname] [-i headerfile] [-c|-c++|-fortran] [-f <instr_req_file> ]
For selective instrumentation, use –f option
% tau_instrumentor foo.pdb foo.cpp –o foo.inst.cpp –f selective.dat
% cat selective.dat
# Selective instrumentation: Specify an exclude/include list of routines/files.

BEGIN_EXCLUDE_LIST
void quicksort(int *, int, int)
void sort_5elements(int *)
void interchange(int *, int *)
END_EXCLUDE_LIST

BEGIN_FILE_INCLUDE_LIST
Main.cpp
Foo?.c
*.C
END_FILE_INCLUDE_LIST
# Instruments routines in Main.cpp, Foo?.c and *.C files only
# Use BEGIN_[FILE]_INCLUDE_LIST with END_[FILE]_INCLUDE_LIST
```

tau_reduce: Rule-Based Overhead Analysis

- Analyze the performance data to determine events with high (relative) overhead performance measurements
- Create a select list for excluding those events
- Rule grammar (used in *tau_reduce* tool)

  [GroupName:] Field Operator Number

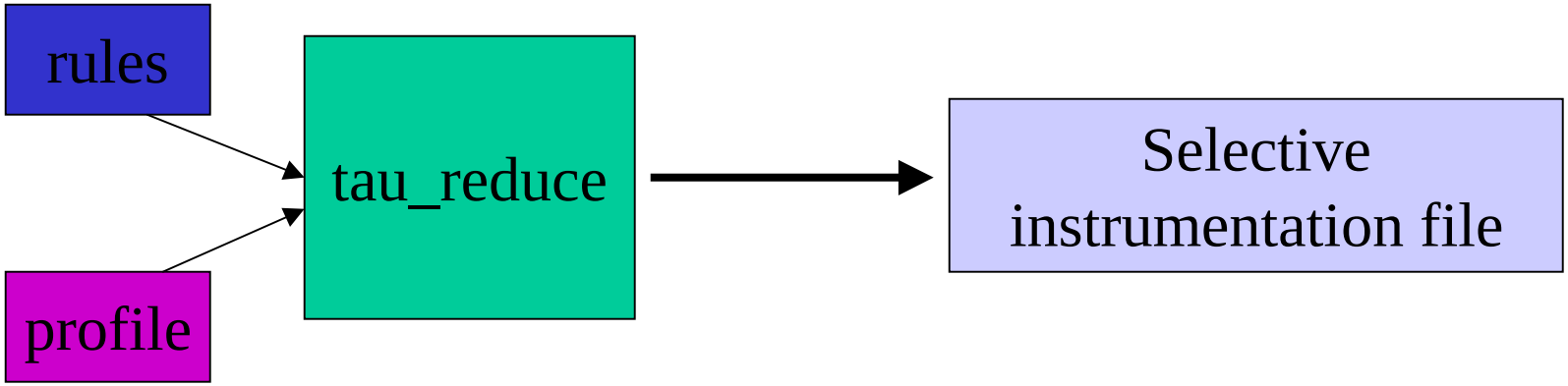
  - *GroupName* indicates rule applies to events in group
  - *Field* is a event metric attribute (from profile statistics)
    - numcalls, numsubs, percent, usec, cumusec, count [PAPI], totalcount, stdev, usecs/call, counts/call
  - *Operator* is one of >, <, or =
  - *Number* is any number
  - Compound rules possible using & between simple rules

Optimizing Instrumentation Overhead: Examples

- #Exclude all events that are members of TAU_USER
  #and use less than 1000 microseconds
  TAU_USER:usec < 1000
- #Exclude all events that have less than 100
  #microseconds and are called only once
  usec < 1000 & numcalls = 1
- #Exclude all events that have less than 1000 usecs per
  #call OR have a (total inclusive) percent less than 5
  usecs/call < 1000
  percent < 5
- Scientific notation can be used
  - **usec>1000 & numcalls>400000 & usecs/call<30 & percent>25**

TAU_REDUCE

- Reads profile files and rules
- Creates selective instrumentation file
  - Specifies which routines should be excluded from instrumentation



Instrumentation of OpenMP Constructs

- **O**penMP **P**ragma **A**nd **R**egion **I**nstrumentor
- Source-to-Source translator to insert POMP calls around OpenMP constructs and API functions
- Done: Supports
  - Fortran77 and Fortran90, OpenMP 2.0
  - C and C++, OpenMP 1.0
  - POMP Extensions
  - EPILOG and TAU POMP implementations
  - Preserves source code information (`#line line file`)
- Work in Progress:
  Investigating standardization through OpenMP Forum
- KOJAK Project website http://icl.cs.utk.edu/kojak

OpenMP API Instrumentation

- Transform
  - **`omp_#_lock()`** → **`pomp_#_lock()`**
  - **`omp_#_nest_lock()`** → **`pomp_#_nest_lock()`**

  `[# = init|destroy|set|unset|test]`

- POMP version
  - Calls omp version internally
  - Can do extra stuff before and after call

Example: !$OMP PARALLEL DO Instrumentation

```
call pomp_parallel_fork(d)
!$OMP PARALLEL other-clauses...
      call pomp_parallel_begin(d)
      call pomp_do_enter(d)
      !$OMP DO schedule-clauses, ordered-clauses,
               lastprivate-clauses
            do loop
      !$OMP END DO NOWAIT
      call pomp_barrier_enter(d)
      !$OMP BARRIER
      call pomp_barrier_exit(d)
      call pomp_do_exit(d)
      call pomp_parallel_end(d)
!$OMP END PARALLEL DO
call pomp_parallel_join(d)
```

Opari Instrumentation: Example

- OpenMP directive instrumentation

```
pomp_for_enter(&omp_rd_2);
#line 252 "stommel.c"
#pragma omp for schedule(static) reduction(+: diff) private(j)
  firstprivate (a1,a2,a3,a4,a5) nowait
for( i=i1;i<=i2;i++) {
  for(j=j1;j<=j2;j++){
    new_psi[i][j]=a1*psi[i+1][j] + a2*psi[i-1][j] + a3*psi[i][j+1]
      + a4*psi[i][j-1] - a5*the_for[i][j];
    diff=diff+fabs(new_psi[i][j]-psi[i][j]);
  }
}
pomp_barrier_enter(&omp_rd_2);
#pragma omp barrier
pomp_barrier_exit(&omp_rd_2);
pomp_for_exit(&omp_rd_2);
#line 261 "stommel.c"
```

Using Opari with TAU

Step I: Configure KOJAK/opari [Download from http://www.fz-juelich.de/zam/kojak/**]**

```
% cd kojak-2.1; cp mf/Makefile.defs.ibm Makefile.defs;
  edit Makefile
% make
```

Builds opari

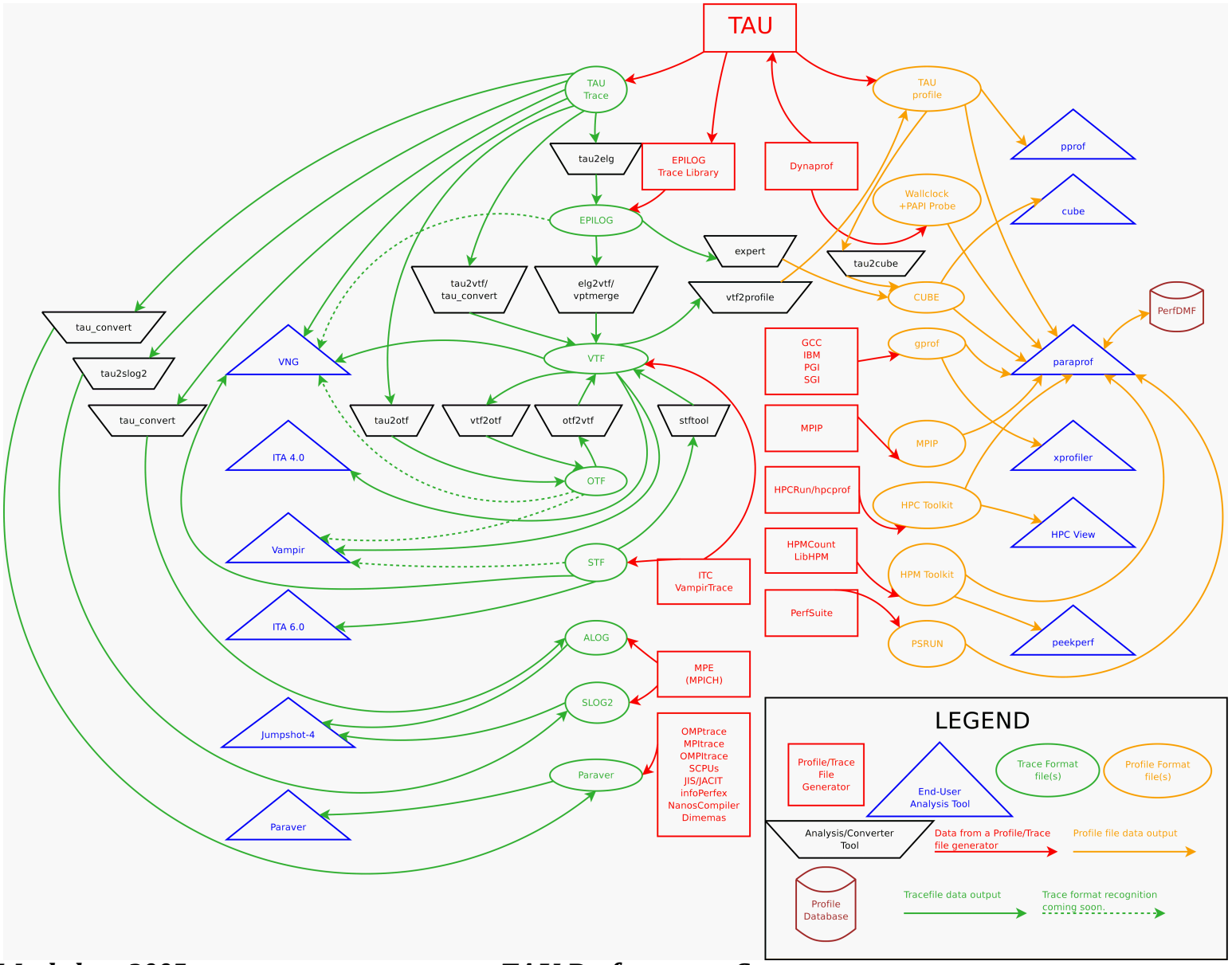
Step II: Configure TAU with Opari (used here with MPI and PDT)

```
% configure –opari=/usr/contrib/TAU/kojak-2.1/opari
  -mpiinc=/usr/lpp/ppe.poe/include
  –mpilib=/usr/lpp/ppe.poe/lib
  –pdt=/usr/contrib/TAU/pdtoolkit-3.4
% make clean; make install
% setenv TAU_MAKEFILE /tau/<arch>/lib/Makefile.tau-…opari-…
% tau_cxx.sh -c foo.cpp
% tau_cxx.sh -c bar.f90
% tau_cxx.sh *.o -o app
```

Advances in TAU Measurement

- Profiling (four types)
  - Memory profiling
    - global heap memory tracking (several options)
  - Callpath profiling and calldepth profiling
    - user-controllable callpath length and calling depth
  - Phase-based profiling
- Tracing
  - Generation of VTF3 / SLOG2 traces files (fully portable)
  - Inclusion of hardware performance counts in trace files
  - Hierarchical trace merging
- Online performance overhead compensation
- Component software proxy generation and monitoring

Building Bridges to Other Tools: TAU



TAU Tracing Enhancements

- Configure TAU with -TRACE –vtf=<dir> -slog2 options

```
% configure –TRACE –vtf=<dir> …
```

  Generates tau_merge, tau2vtf tools in <tau>/<arch>/bin directory

```
% configure -TRACE -slog2
```

  Generates tau2slog2 and jumpshot v4 tools bundled with TAU in <tau>/<arch>/bin directory

  - Need working javac [v1.4] in your path

- Execute application

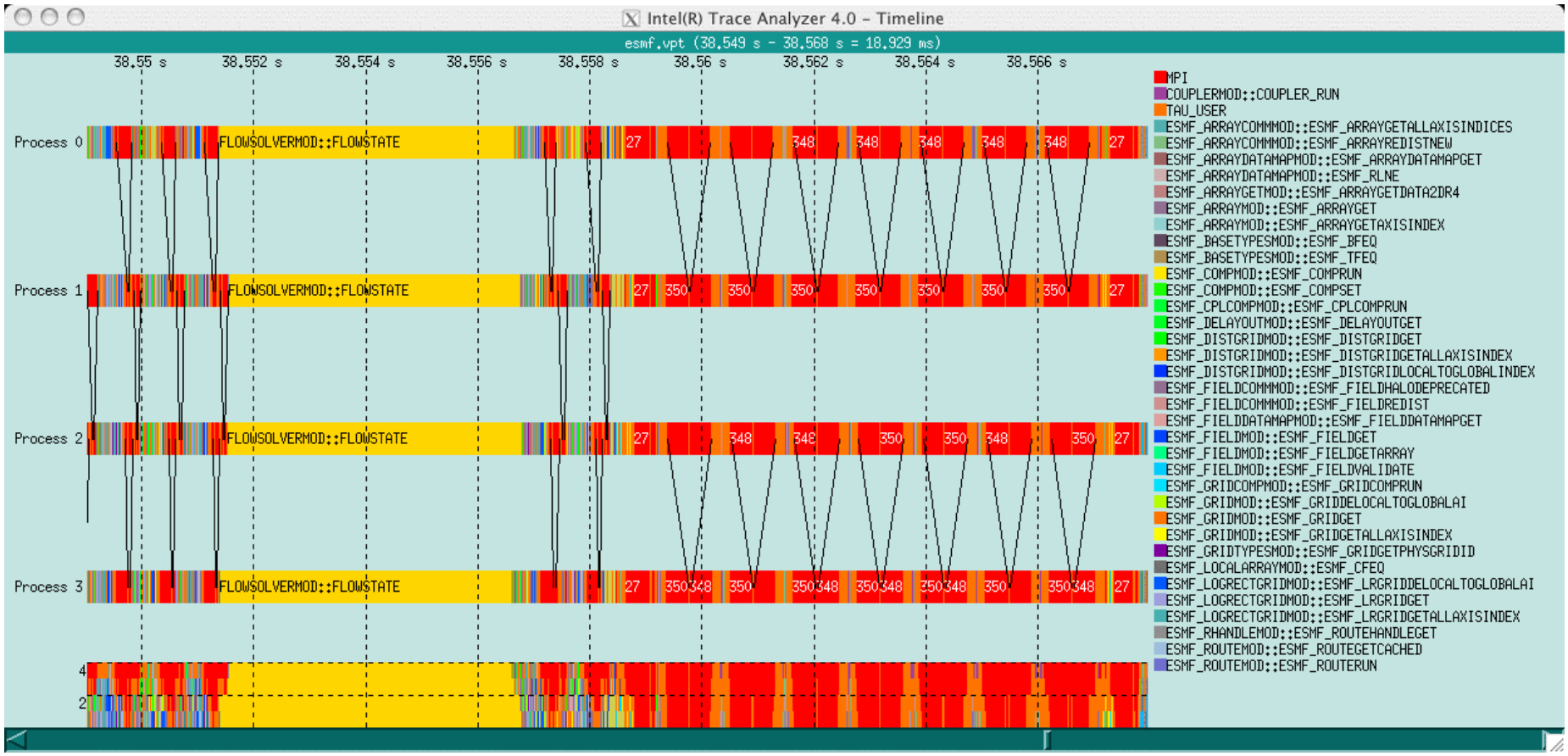
```
% mpirun -np 4 app
```

- Merge and convert trace files to VTF3/SLOG2 format

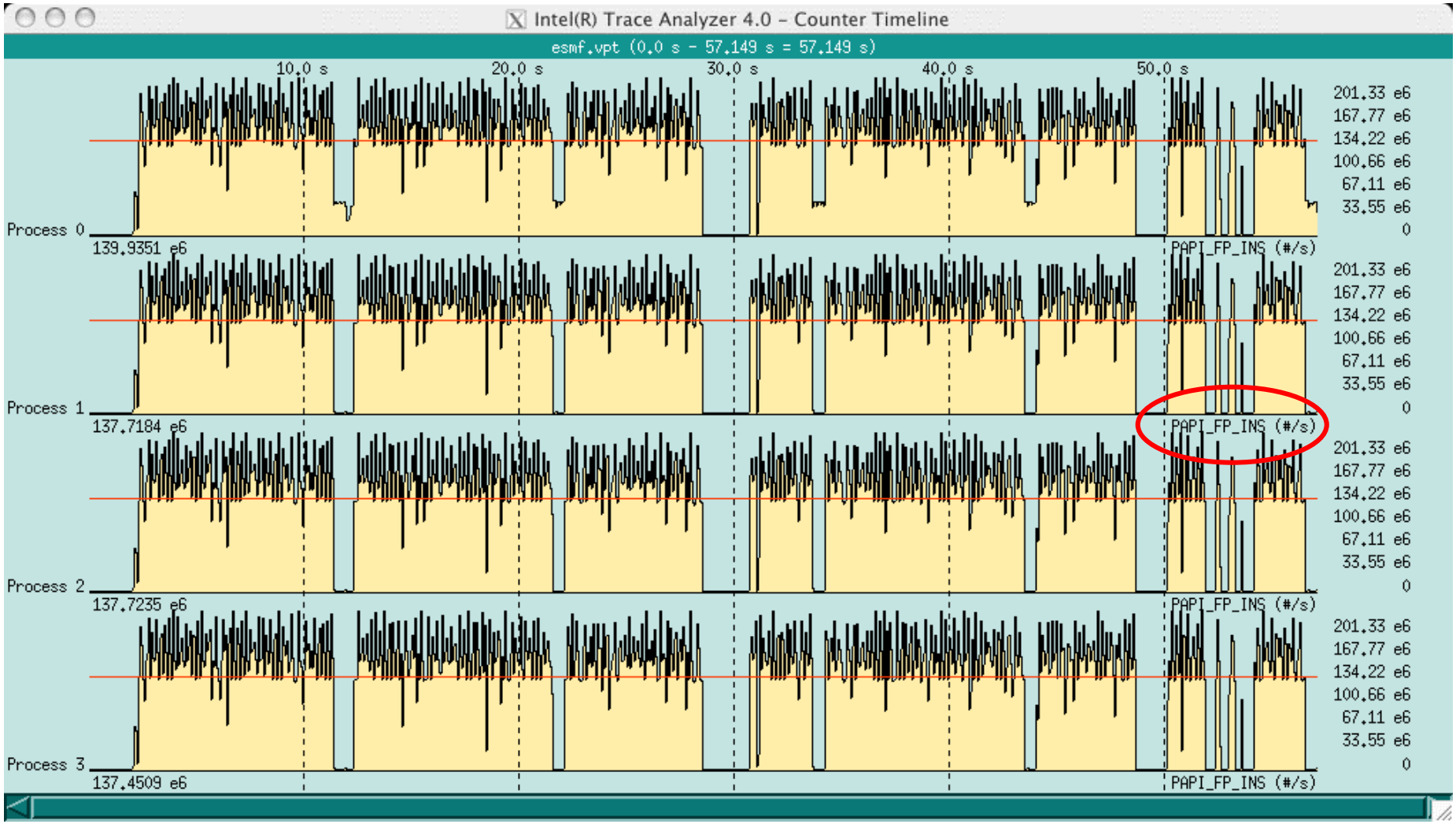
```
% tau_treemerge.pl
% tau2vtf tau.trc tau.edf app.vpt.gz
% traceanalyzer foo.vpt.gz

% tau2slog2 tau.trc tau.edf app.slog2
% jumpshot app.slog2
```

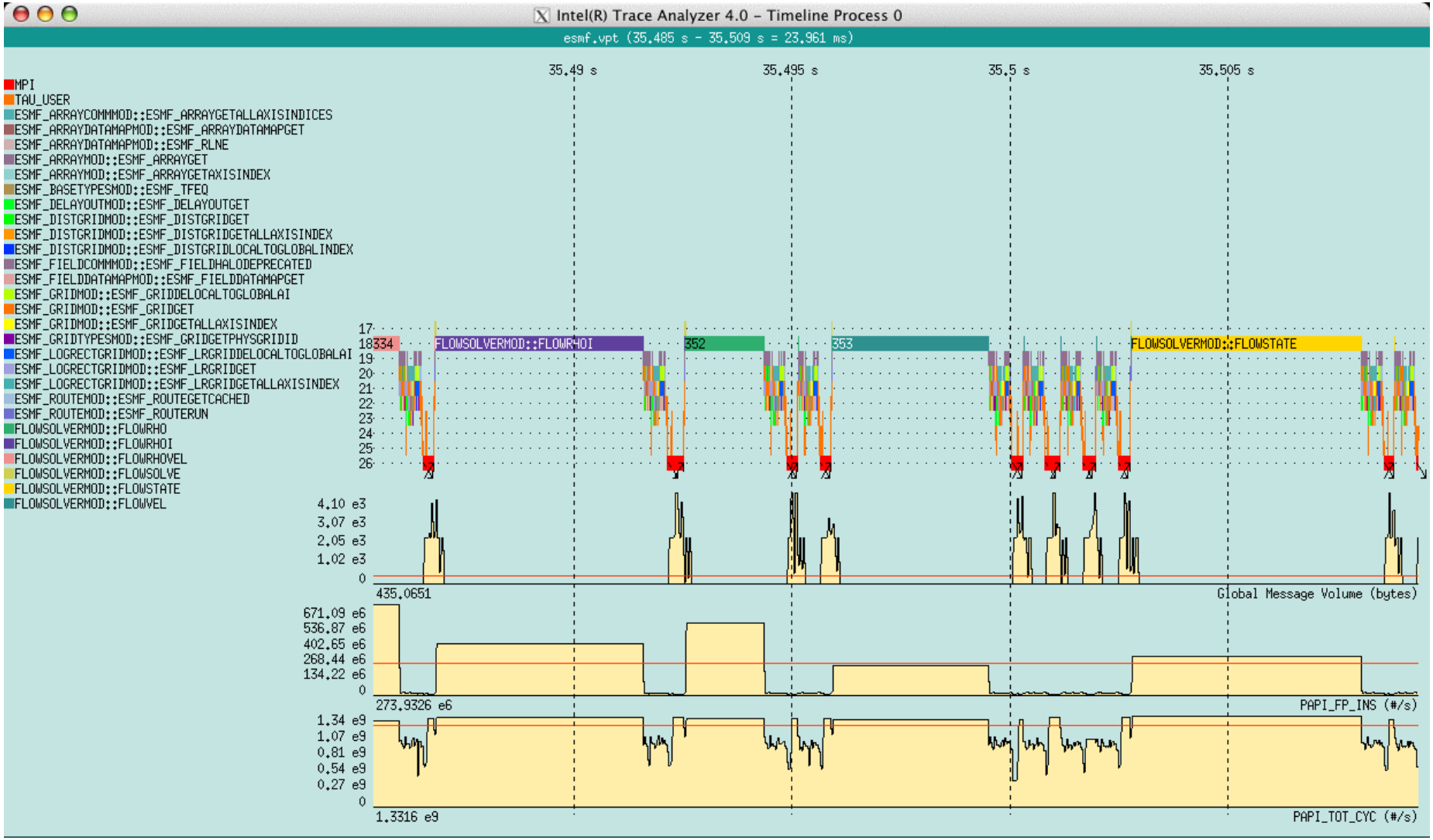
Intel ® Traceanalyzer (Vampir) Global Timeline



Visualizing TAU Traces with Counters/Samples



Visualizing TAU Traces with Counters/Samples



Memory Profiling in TAU

- Configuration option –PROFILEMEMORY
  - Records global heap memory utilization for each function
  - Takes one sample at beginning of each function and associates the sample with function name
- Configuration option -PROFILEHEADROOM
  - Records headroom (amount of free memory to grow) for each function
  - Takes one sample at beginning of each function and associates it with the callstack [TAU_CALLPATH_DEPTH env variable]
- Independent of instrumentation/measurement options selected
- No need to insert macros/calls in the source code
- User defined atomic events appear in profiles/traces

Memory Profiling in TAU

Sorted By: number of userEvents

NumSamples	Max	Min	Mean	Std. Dev	Name
252032	2022.7	1181.2	1534.3	410.04	MODULEHYDRO_1D::HYDRO_1D - Heap Memory (KB)
252032	2022.8	1181.7	1534.3	410.04	MODULEINTRFC::INTRFC - Heap Memory (KB)
104559	2023.2	331.13	1526.6	409.54	MODULEEOS3D::EOS3D - Heap Memory (KB)
63008	2022.7	1182	1534.3	410.01	MODULEUPDATE_SOLN::UPDATE_SOLN - Heap Memory (KB)
55545	2023.3	333.07	1514.2	408.31	DBASETREE::DBASENEIGHBORBLOCKLIST - Heap Memory (KB)
51374	2023	1179.4	1497.7	402.53	AMR_PROLONG_GEN_UNK_FUN - Heap Memory (KB)
42120	2022.7	1187.5	1533.5	409.83	ABUNDANCE_RESTRICT - Heap Memory (KB)
41958	2023	346.12	1514.9	408.39	AMR_RESTRICT_UNK_FUN - Heap Memory (KB)
31832	2022.8	1187.4	1534.1	409.91	AMR_RESTRICT_RED - Heap Memory (KB)
31504	2022.7	1181.8	1534.3	410.04	DIFFUSE - Heap Memory (KB)
26042	2023	1179.2	1501.9	403.61	AMR_PROLONG_UNK_FUN - Heap Memory (KB)

Flash2 code profile (-PROFILEMEMORY) on IBM BlueGene/L [MPI rank 0]

Memory Profiling in TAU

- Instrumentation based observation of global heap memory (not per function)
  - call TAU_TRACK_MEMORY()
  - call TAU_TRACK_MEMORY_HEADROOM()
    - Triggers one sample every 10 secs
  - call TAU_TRACK_MEMORY_HERE()
  - call TAU_TRACK_MEMORY_HEADROOM_HERE()
    - Triggers sample at a specific location in source code
  - call TAU_SET_INTERRUPT_INTERVAL(seconds)
    - To set inter-interrupt interval for sampling
  - call TAU_DISABLE_TRACKING_MEMORY()
  - call TAU_DISABLE_TRACKING_MEMORY_HEADROOM()
    - To turn off recording memory utilization
  - call TAU_ENABLE_TRACKING_MEMORY()
  - call TAU_ENABLE_TRACKING_MEMORY_HEADROOM()
    - To re-enable tracking memory utilization

Profile Measurement – Three Flavors

- Flat profiles
  - Time (or counts) spent in each routine (nodes in callgraph).
  - Exclusive/inclusive time, no. of calls, child calls
  - E.g,: MPI_Send, foo, …
- Callpath Profiles
  - Flat profiles, **plus**
  - Sequence of actions that led to poor performance
  - Time spent along a calling path (edges in callgraph)
  - E.g., “main=> f1 => f2 => MPI_Send” shows the time spent in MPI_Send when called by f2, when f2 is called by f1, when it is called by main. Depth of this callpath = 4 (TAU_CALLPATH_DEPTH environment variable)
- Phase based profiles
  - Flat profiles, **plus**
  - Flat profiles under a phase (nested phases are allowed)
  - Default “main” phase has all phases and routines invoked outside phases
  - Supports static or dynamic (per-iteration) phases
  - E.g., “IO => MPI_Send” is time spent in MPI_Send in IO phase

TAU Timers and Phases

- Static timer
  - Shows time spent in all invocations of a routine (foo)
  - E.g., “foo()” 100 secs, 100 calls
- Dynamic timer
  - Shows time spent in each invocation of a routine
  - E.g., “foo() 3” 4.5 secs, “foo 10” 2 secs (invocations 3 and 10 respectively)
- Static phase
  - Shows time spent in all routines called (directly/indirectly) by a given routine (foo)
  - E.g., “foo() => MPI_Send()” 100 secs, 10 calls shows that a total of 100 secs were spent in MPI_Send() when it was called by foo.
- Dynamic phase
  - Shows time spent in all routines called by a given invocation of a routine.
  - E.g., “foo() 4 => MPI_Send()” 12 secs, shows that 12 secs were spent in MPI_Send when it was called by the 4th invocation of foo.

Advances in TAU Performance Analysis

- Enhanced parallel profile analysis (*ParaProf*)
  - Callpath analysis integration in ParaProf
  - Event callgraph view
- *Performance Data Management Framework* (*PerfDMF*)
  - First release of prototype
- Integration with *Vampir Next Generation* (*VNG*)
  - Online trace analysis
- 3D Performance visualization prototype
- Component performance modeling and QoS

Pprof – Flat Profile (NAS PB LU)

- Intel Linux cluster
- F90 + MPICH
- Profile
  - Node
  - Context
  - Thread
- Events
  - code
  - MPI
- Metric
  - time
- Text display

emacs@neutron.cs.uoregon.edu

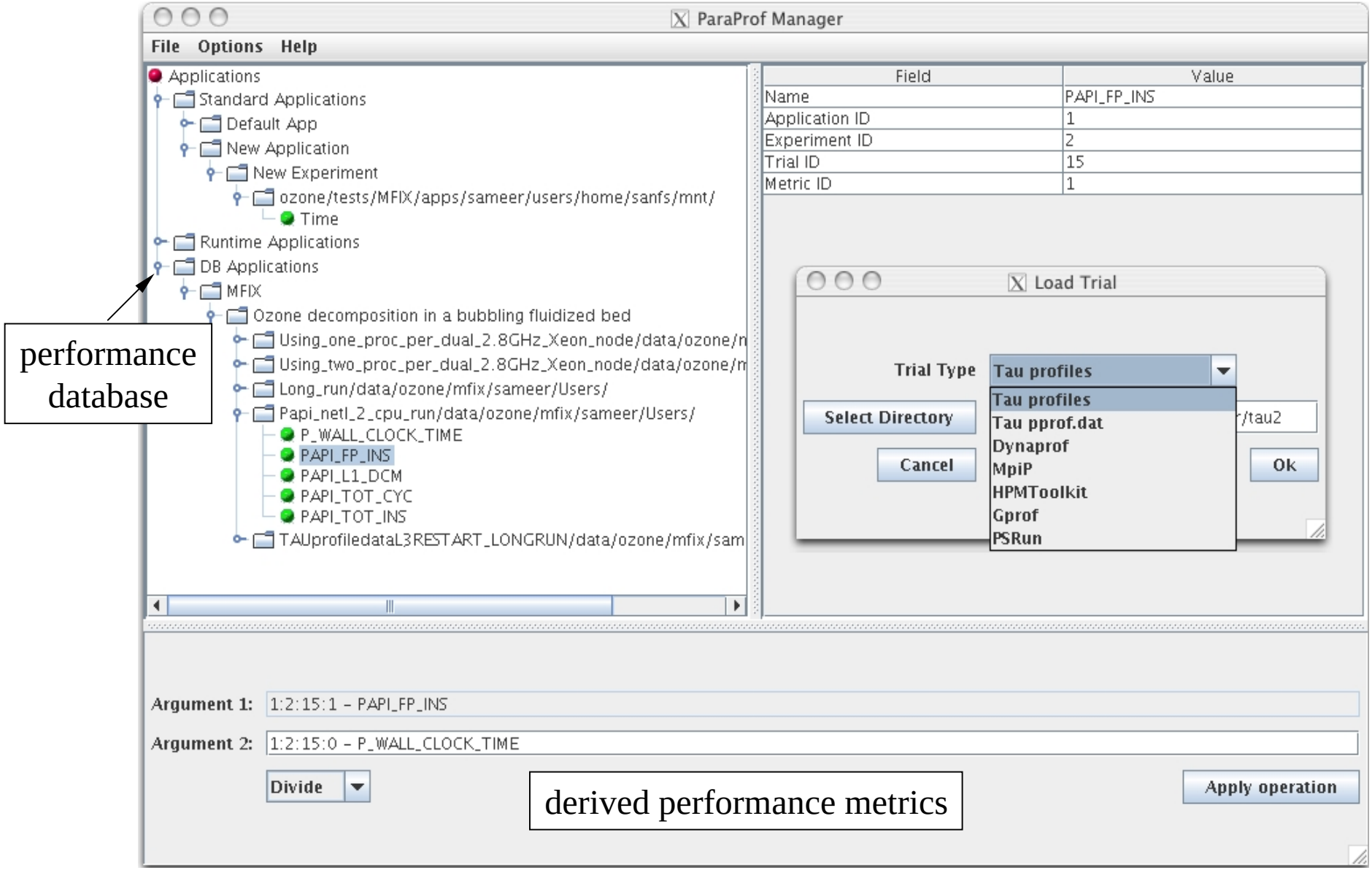
Buffers Files Tools Edit Search Mule Help

Reading Profile files in profile.*

%Time	Exclusive msec	Inclusive total msec	#Call	#Subrs	Inclusive usec/call	Name
100.0	1	3:11.293	1	15	191293269	applu
99.6	3,667	3:10.463	3	37517	63487925	bcast_inputs
67.1	491	2:08.326	37200	37200	3450	exchange_1
44.5	6,461	1:25.159	9300	18600	9157	buts
41.0	1:18.436	1:18.436	18600	0	4217	MPI_Recv()
29.5	6,778	56,407	9300	18600	6065	blts
26.2	50,142	50,142	19204	0	2611	MPI_Send()
16.2	24,451	31,031	301	602	103096	rhs
3.9	7,501	7,501	9300	0	807	jacld
3.4	838	6,594	604	1812	10918	exchange_3
3.4	6,590	6,590	9300	0	709	jacu
2.6	4,989	4,989	608	0	8206	MPI_Wait()
0.2	0.44	400	1	4	400081	init_comm
0.2	398	399	1	39	399634	MPI_Init()
0.1	140	247	1	47616	247086	setiv
0.1	131	131	57252	0	2	exact
0.1	89	103	1	2	103168	erhs
0.1	0.966	96	1	2	96458	read_input
0.0	95	95	9	0	10603	MPI_Bcast()
0.0	26	44	1	7937	44878	error
0.0	24	24	608	0	40	MPI_Irecv()
0.0	15	15	1	5	15630	MPI_Finalize()
0.0	4	12	1	1700	12335	setbv
0.0	7	8	3	3	2893	l2norm
0.0	3	3	8	0	491	MPI_Allreduce()
0.0	1	3	1	6	3874	pintgr
0.0	1	1	1	0	1007	MPI_Barrier()
0.0	0.116	0.837	1	4	837	exchange_4
0.0	0.512	0.512	1	0	512	MPI_Keyval_create()
0.0	0.121	0.353	1	2	353	exchange_5
0.0	0.024	0.191	1	2	191	exchange_6
0.0	0.103	0.103	6	0	17	MPI_Type_contiguous()

--:-- NPB_LU.out (Fundamental)--L8--Top----

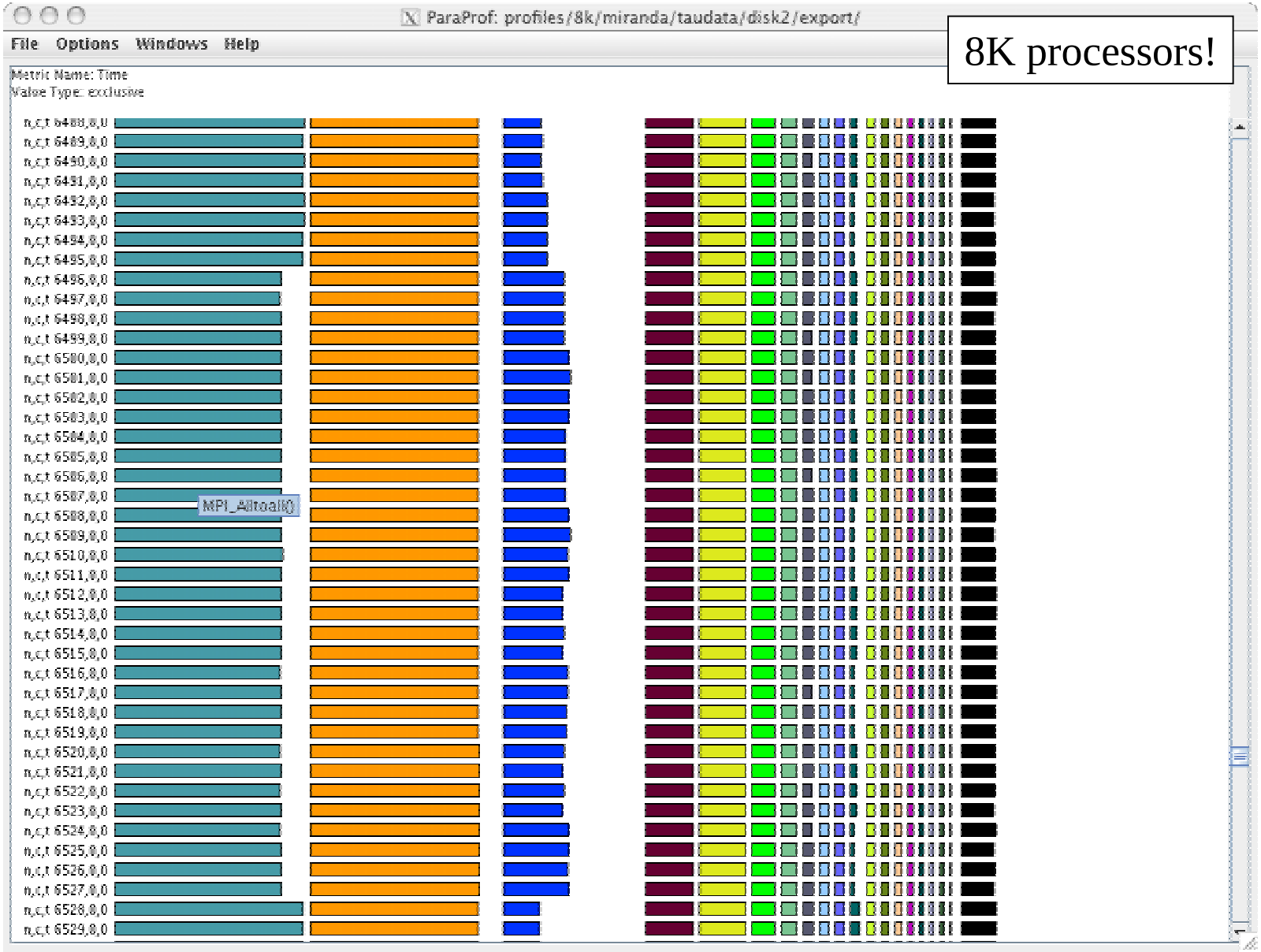
ParaProf – Manager Window



performance database

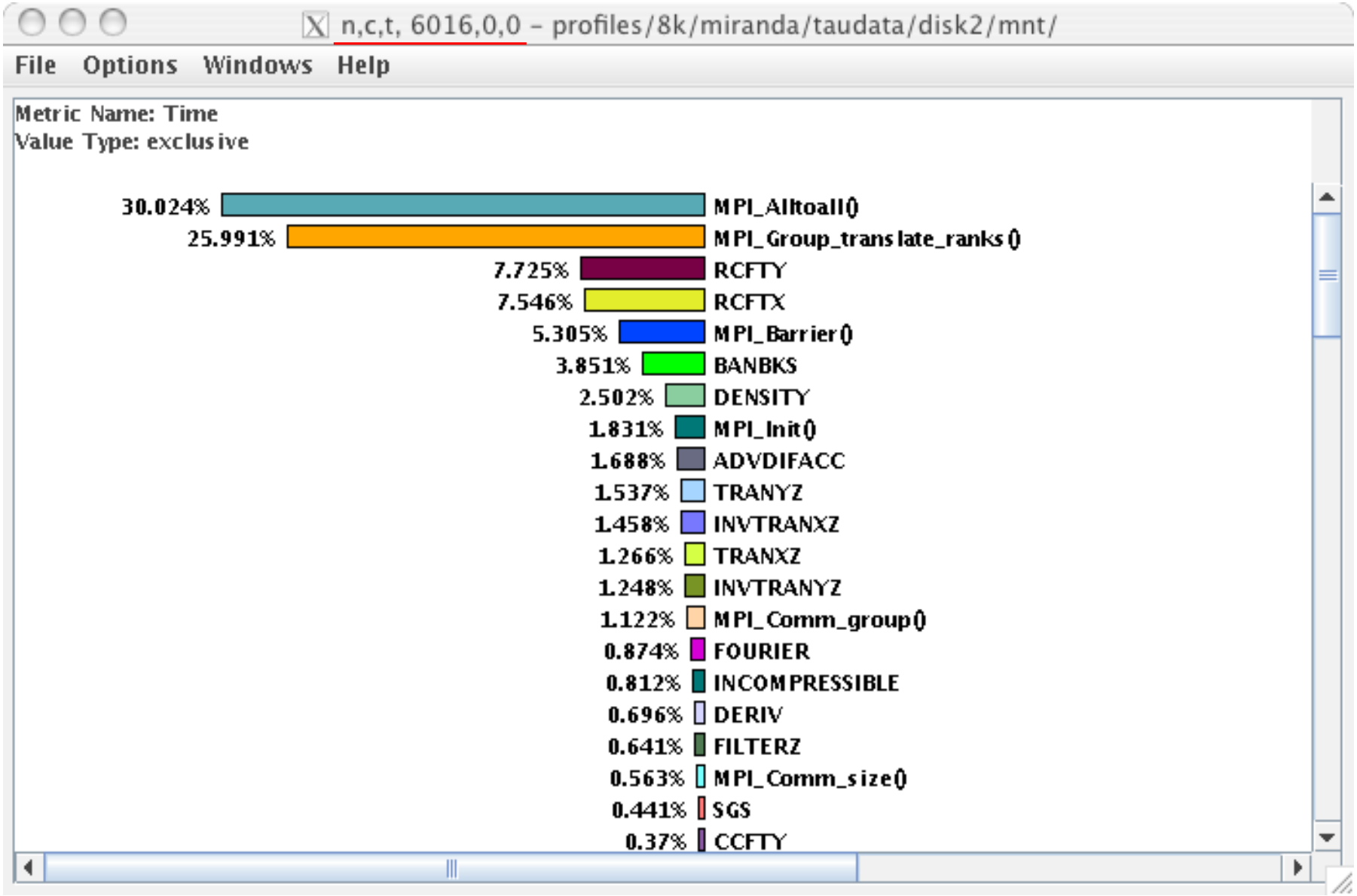
derived performance metrics

ParaProf – Full Profile (Miranda)



8K processors!

ParaProf– Flat Profile (Miranda)



ParaProf– Callpath Profile (Flash)

n,c,t, 0,0,0 - callpath-all/scaling/flash/taudata/disk2/mnt/

File Options Windows Help

Metric Name: Time
Value Type: exclusive

26.474%	MODULEHYDROSWEEP::HYDRO_SWEEP
26.474%	FLASH => EVOLVE => HYDRO::HYDRO_3D => MODULEHYDROSWEEP::HYDRO_SWEEP
24.556%	MODULEHYDRO_1D::HYDRO_1D
24.556%	FLASH => EVOLVE => HYDRO::HYDRO_3D => MODULEHYDROSWEEP::HYDRO_SWEEP => MODULEHYDRO_1D::HYDRO_1D
14.351%	MODULEINTRFC::INTRFC
14.351%	FLASH => EVOLVE => HYDRO::HYDRO_3D => MODULEHYDROSWEEP::HYDRO_SWEEP => MODULEHYDRO_1D::HYDRO_1D => MODULEINTRFC::INT
4.501%	MODULEEOS3D::EOS3D
4.427%	MPI_Ssend()
3.678%	FLASH => EVOLVE => HYDRO::HYDRO_3D => MODULEHYDROSWEEP::HYDRO_SWEEP => MODULEEOS3D::EOS3D
3.536%	MPI_Allreduce()
2.727%	MPI_Waitall()
2.242%	MODULEUPDATE_SOLN::UPDATE_SOLN
2.242%	FLASH => EVOLVE => HYDRO::HYDRO_3D => MODULEHYDROSWEEP::HYDRO_SWEEP => MODULEUPDATE_SOLN::UPDATE_SOLN
2.059%	AMR_GUARDCELL_CC_SRL
1.703%	FLASH => EVOLVE => HYDRO::HYDRO_3D => MODULEHYDROSWEEP::HYDRO_SWEEP => MESH_GUARDCELL => AMR_GUARDCELL_SRL => AMR_
1.56%	FLASH => EVOLVE => HYDRO::HYDRO_3D => MODULEHYDROSWEEP::HYDRO_SWEEP => MESH_GUARDCELL => AMR_GUARDCELL_SRL => AMR_
1.406%	FLASH => EVOLVE => MESH_UPDATE_GRID_REFINEMENT => MESH_REFINE_DEREFINE => AMR_REFINE_DEREFINE => AMR_MORTON_ORDER =>
1.361%	FLASH => TIMESTEP => MPI_Allreduce()
1.319%	AMR_RESTRICT_UNK_FUN
1.272%	AMR_PROLONG_GEN_UNK_FUN
1.093%	FLASH => EVOLVE => HYDRO::HYDRO_3D => MODULEHYDROSWEEP::HYDRO_SWEEP => MESH_GUARDCELL => AMR_GUARDCELL_C_TO_F => A
1.077%	ABUNDANCE_RESTRICT
1.077%	FLASH => EVOLVE => HYDRO::HYDRO_3D => MODULEHYDROSWEEP::HYDRO_SWEEP => ABUNDANCE_RESTRICT
1.064%	DBASETREE::DBASENEIGHBORBLOCKLIST
1%	FLASH => EVOLVE => HYDRO::HYDRO_3D => MODULEHYDROSWEEP::HYDRO_SWEEP => MESH_GUARDCELL => AMR_RESTRICT => AMR_RESTRI
0.987%	FLASH => EVOLVE => HYDRO::HYDRO_3D => MODULEHYDROSWEEP::HYDRO_SWEEP => MESH_FLUX_CONSERVE => AMR_FLUX_CONSERVE_UDT
0.96%	FLASH => EVOLVE => HYDRO::HYDRO_3D => MODULEHYDROSWEEP::HYDRO_SWEEP => MESH_GUARDCELL => AMR_GUARDCELL_C_TO_F => A
0.916%	MPI_Barrier()
0.807%	FLASH => EVOLVE => HYDRO::HYDRO_3D => MODULEHYDROSWEEP::HYDRO_SWEEP => MESH_GUARDCELL => TOT_BND => DBASETREE::DBAS
0.806%	AMR_PROLONG_UNK_FUN
0.735%	AMR_DIAGONAL_PATCH
0.699%	DIFFUSE
0.699%	FLASH => EVOLVE => HYDRO::HYDRO_3D => MODULEHYDROSWEEP::HYDRO_SWEEP => DIFFUSE
0.671%	AMR_RESTRICT_RED
0.671%	FLASH => EVOLVE => HYDRO::HYDRO_3D => MODULEHYDROSWEEP::HYDRO_SWEEP => MESH_FLUX_CONSERVE => AMR_FLUX_CONSERVE_UDT
0.657%	FLASH => EVOLVE => HYDRO::HYDRO_3D => MODULEHYDROSWEEP::HYDRO_SWEEP => MESH_GUARDCELL => AMR_GUARDCELL_SRL => AMR_
0.638%	FLASH => EVOLVE => MESH_UPDATE_GRID_REFINEMENT => MARK_GRID_REFINEMENT => MPI_Barrier()
0.61%	FLASH => EVOLVE => HYDRO::HYDRO_3D => MODULEHYDROSWEEP::HYDRO_SWEEP => MESH_GUARDCELL => AMR_GUARDCELL_C_TO_F => A
0.556%	FLASH => EVOLVE => HYDRO::HYDRO_3D => MODULEHYDROSWEEP::HYDRO_SWEEP => MESH_GUARDCELL => AMR_GUARDCELL_C_TO_F => A
0.508%	TOT_BND
0.454%	FLASH => EVOLVE => MESH_UPDATE_GRID_REFINEMENT => MARK_GRID_REFINEMENT => MODULEEOS3D::EOS3D

ParaProf– Callpath Profile (ESMF)

21-level callpath

Function Data Window: compensatecallpath/esmf/sameer/Users/

File Options Windows Help

Metric Name: Time

Name: ESMF_APPLICATIONWRAPPER => ESMF_GRIDCOMPMOD::ESMF_GRIDCOMPRUN => ESMF_COMPMOD::ESMF_COMPRUN => void c_esmc_ftablecallentrypointvm(ESMC_VM **, ESMC_VMPlan **, void **, void **, ESMC_FTable **, char *, int *, int *, int) C => void *vmachine::vmachine_enter(vmplan &, void *(*)(void *, void *), void *) vmachine => void *vmachine_spawn(void *) => void *ESMC_FTableCallEntryPointVMHop(void *, void *) C => int ESMC_FTable::ESMC_FTableCallVFuncPtr(char *, ESMC_VM *, int *) ESMC_FTable => COUPLEDFLOWMOD::COUPLEDFLOW_RUN => ESMF_CPLCOMPMOD::ESMF_CPLCOMPRUN => ESMF_COMPMOD::ESMF_COMPRUN => void c_esmc_ftablecallentrypointvm(ESMC_VM **, ESMC_VMPlan **, void **, void **, ESMC_FTable **, char *, int *, int *, int) C => void *vmachine::vmachine_enter(vmplan &, void *(*)(void *, void *), void *) vmachine => void *vmachine_spawn(void *) => void *ESMC_FTableCallEntryPointVMHop(void *, void *) C => int ESMC_FTable::ESMC_FTableCallVFuncPtr(char *, ESMC_VM *, int *) ESMC_FTable => COUPLERMOD::COUPLER_RUN => ESMF_FIELDCOMMMOD::ESMF_FIELDREDIST => ESMF_ARRAYCOMMMOD::ESMF_ARRAYREDISTNEW => ESMF_ROUTEMOD::ESMF_ROUTERUN => void c_esmc_routerunla(ESMC_Route **, ESMC_LocalArray **, ESMC_LocalArray **, int *) C => int ESMC_Route::ESMC_RouteRun(void *, void *, ESMC_DataKind) => int ESMC_DELayout::ESMC_DELayoutExchange(void **, void **, void **, void **, int, int, int, int, ESMC_Logical) => int ESMC_DELayout::ESMC_DELayoutCopy(void **, void **, int, int, int, ESMC_Logical) => void vmachine::vmachine_recv(void *, int, int) vmachine => MPI_Recv()

Value Type: exclusive

10.7487%	mean
11.2785%	n,c,t 1,0,0
10.9582%	n,c,t 3,0,0
10.4453%	n,c,t 2,0,0
10.3146%	n,c,t 0,0,0

Gprof Style Callpath View in Paraprof (SAGE)

```
Metric Name: Time
Sorted By: exclusive
Units: seconds

     Exclusive    Inclusive    Calls/Tot.Calls                Name[id]
     ---------------------------------------------------------------------------

     1.8584       1.8584       1196/13188                     TOKEN_MODULE::TOKEN_GS_I  [521]
     0.584        0.584        234/13188                      TOKEN_MODULE::TOKEN_GS_L  [544]
     25.0819      25.0819      11758/13188                    TOKEN_MODULE::TOKEN_GS_R8  [734]
 --> 27.5242      27.5242      13188                          MPI_Waitall()  [525]


     17.9579      39.1657      156/156                        DERIVATIVE_MODULE::DERIVATIVES_NOFACE  [841]
 --> 17.9579      39.1657      156                            DERIVATIVE_MODULE::DERIVATIVES_FACE  [843]
     0.0156       0.0195       312/312                        TIMER_MODULE::TIMERSET  [77]
     0.1133       9.1269       2340/2340                      MESSAGE_MODULE::CLONE_GET_R8  [808]
     0.1602       11.4608      4056/4056                      MESSAGE_MODULE::CLONE_PUT_R8  [850]
     0.0059       0.6006       117/117                        MESSAGE_MODULE::CLONE_PUT_I  [856]


     14.1151      21.6209      5/5                            MATRIX_MODULE::MCGDS  [1443]
 --> 14.1151      21.6209      5                              MATRIX_MODULE::CSR_CG_SOLVER  [1470]
     0.0654       1.2617       1005/1005                      TOKEN_MODULE::TOKEN_GET_R8  [769]
     0.0557       5.2714       1005/1005                      TOKEN_MODULE::TOKEN_REDUCTION_R8_S  [1475]
     0.0703       0.9726       1000/1000                      TOKEN_MODULE::TOKEN_REDUCTION_R8_V  [208]
```

ParaProf – Phase Profile (MFIX)

In 51st iteration, time spent in MPI_Waitall was 85.81 secs

Total time spent in MPI_Waitall was 4137.9 secs across all 92 iterations

Call Path Data n,c,t, 0,0,0 – phase/mfix/taudata/disk2/mnt/

File Options Windows Help

Metric Name: Time
Sorted By: exclusive
Units: seconds

65.217	65.217	21232.0/1345134.0	ITERATE	48[7116]
55.321	55.321	17888.0/1345134.0	ITERATE	49[7252]
51.351	51.351	16592.0/1345134.0	ITERATE	50[7388]
85.81	85.81	28208.0/1345134.0	ITERATE	51[7524]
75.069	75.069	24384.0/1345134.0	ITERATE	52[7670]
78.938	78.938	25728.0/1345134.0	ITERATE	53[7806]
69.684	69.684	23104.0/1345134.0	ITERATE	54[7942]
58.461	58.461	19072.0/1345134.0	ITERATE	55[8080]
85.117	85.117	27856.0/1345134.0	ITERATE	56[8216]
47.885	47.885	15504.0/1345134.0	ITERATE	57[8354]
46.436	46.436	14816.0/1345134.0	ITERATE	58[8490]
46.242	46.242	14752.0/1345134.0	ITERATE	59[8626]
45.728	45.728	14640.0/1345134.0	ITERATE	60[8772]
45.244	45.244	14656.0/1345134.0	ITERATE	61[8908]
45.283	45.283	14416.0/1345134.0	ITERATE	62[9044]
61.168	61.168	20032.0/1345134.0	ITERATE	63[9180]
46.992	46.992	15600.0/1345134.0	ITERATE	64[9326]
47.01	47.01	15792.0/1345134.0	ITERATE	65[9462]
44.046	44.046	14608.0/1345134.0	ITERATE	66[9598]
47.424	47.424	15584.0/1345134.0	ITERATE	67[9734]
41.176	41.176	13472.0/1345134.0	ITERATE	68[9870]
51.488	51.488	16880.0/1345134.0	ITERATE	69[10016]
43.714	43.714	14480.0/1345134.0	ITERATE	70[10152]
46.175	46.175	15152.0/1345134.0	ITERATE	71[10288]
45.348	45.348	14864.0/1345134.0	ITERATE	72[10424]
38.728	38.728	12848.0/1345134.0	ITERATE	73[10560]
46	46	15008.0/1345134.0	ITERATE	74[10706]
52.453	52.453	17008.0/1345134.0	ITERATE	75[10842]
44.341	44.341	14496.0/1345134.0	ITERATE	76[10978]
44.288	44.288	14240.0/1345134.0	ITERATE	77[11116]
58.298	58.298	18736.0/1345134.0	ITERATE	78[11252]
48.099	48.099	15584.0/1345134.0	ITERATE	79[11388]
45.351	45.351	14480.0/1345134.0	ITERATE	80[11534]
48.512	48.512	15824.0/1345134.0	ITERATE	81[11670]
41.185	41.185	13408.0/1345134.0	ITERATE	82[11806]
34.789	34.789	11248.0/1345134.0	ITERATE	83[11944]
34.061	34.061	10944.0/1345134.0	ITERATE	84[12080]
33.843	33.843	10960.0/1345134.0	ITERATE	85[12216]
33.182	33.182	10848.0/1345134.0	ITERATE	86[12362]
33.165	33.165	10752.0/1345134.0	ITERATE	87[12498]
29.992	29.992	9632.0/1345134.0	ITERATE	88[12634]
28.337	28.337	9136.0/1345134.0	ITERATE	89[12770]
35.926	35.926	11488.0/1345134.0	ITERATE	90[12906]
36.238	36.238	11648.0/1345134.0	ITERATE	91[13052]
27.385	27.385	8896.0/1345134.0	ITERATE	92[13188]
--> 4137.9	4137.9	1345134.0	MPI_Waitall()[121]	

dynamic phases one per interation

ParaProf - Statistics Table (Uintah)

Thread Statistics: n,c,t, 3,0,0 - /home/amorris/data/packed/uintah16.ppk

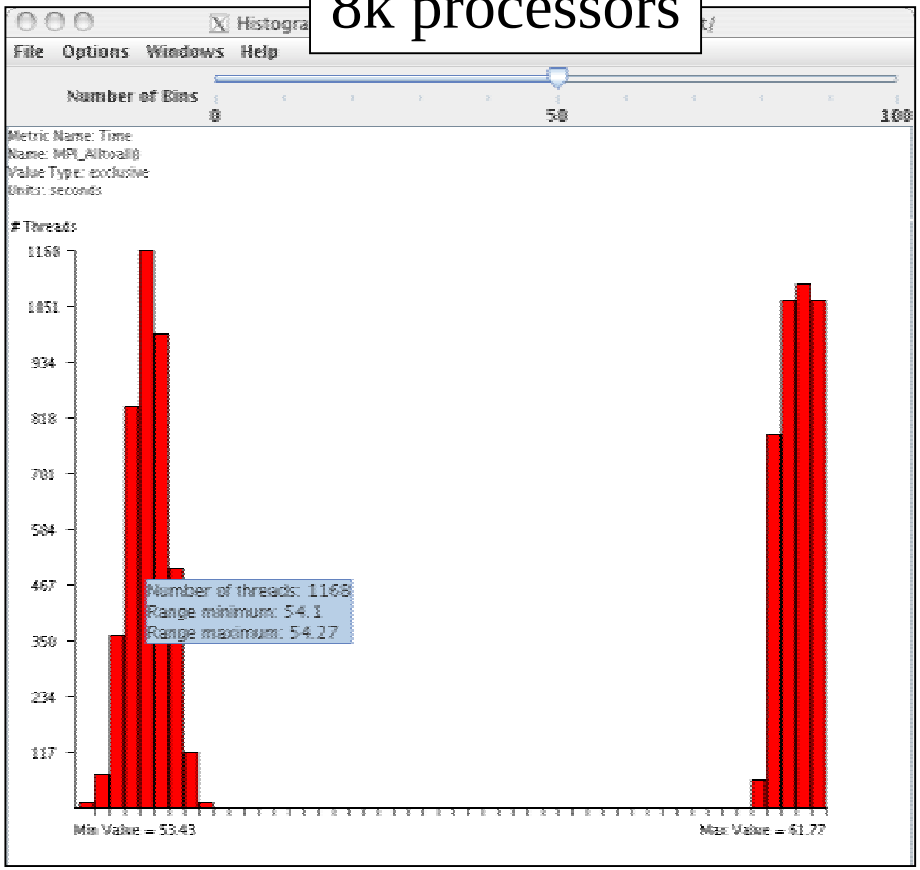
File Options Windows Help

Name	P_WALL_CLOCK_TIME	Calls	Child Calls
main() void (int, char **)	0.015	1	14
Uintah::ProcessorGroup *Uintah::Parallel::getRootProcessor(	0	1	0
Uintah::SimpleSimulationController &Uintah::SimpleSimulatior	0	1	0
Uintah::SimulationController &Uintah::SimulationController::Si	0	1	0
bool Uintah::Parallel::usingMPI()	0	1	0
int Uintah::Parallel::getMPIRank()	0	1	0
void Uintah::OnDemandDataWarehouse::~OnDemandDataW	0	2	0
void Uintah::Parallel::determineIfRunningUnderMPI(int, char	0.002	1	0
void Uintah::Parallel::finalizeManager(Uintah::Parallel::Circur	0.011	1	1
void Uintah::Parallel::initializeManager(int &, char **&, const	0.001	1	3
MPI_Comm_rank()	0	1	0
MPI_Comm_size()	0	1	0
MPI_Init_thread()	6.327	1	39
void Uintah::Parallel::noThreading()	0	1	0
void Uintah::SimpleSimulationController::run() Uintah::Simple	0.074	1	154
MPIScheduler::actuallyCompile()	0.109	2	44
MPIScheduler::execute()	27.68	11	3,460
MPI_Reduce()	0.001	40	40
Uintah::DataWarehouse::ScrubMode Uintah::OnDemandD	0	21	0
Uintah::OnDemandDataWarehouse &Uintah::OnDemandD	0	11	0
bool Uintah::OnDemandDataWarehouse::timestepAborte	0	10	0
bool Uintah::OnDemandDataWarehouse::timestepRestar	0	10	0
bool Uintah::SimpleSimulationController::needRecompile	0	10	0
void Uintah::OnDemandDataWarehouse::get(Uintah::Red	0.001	10	30
void Uintah::OnDemandDataWarehouse::override(const U	0.001	20	40

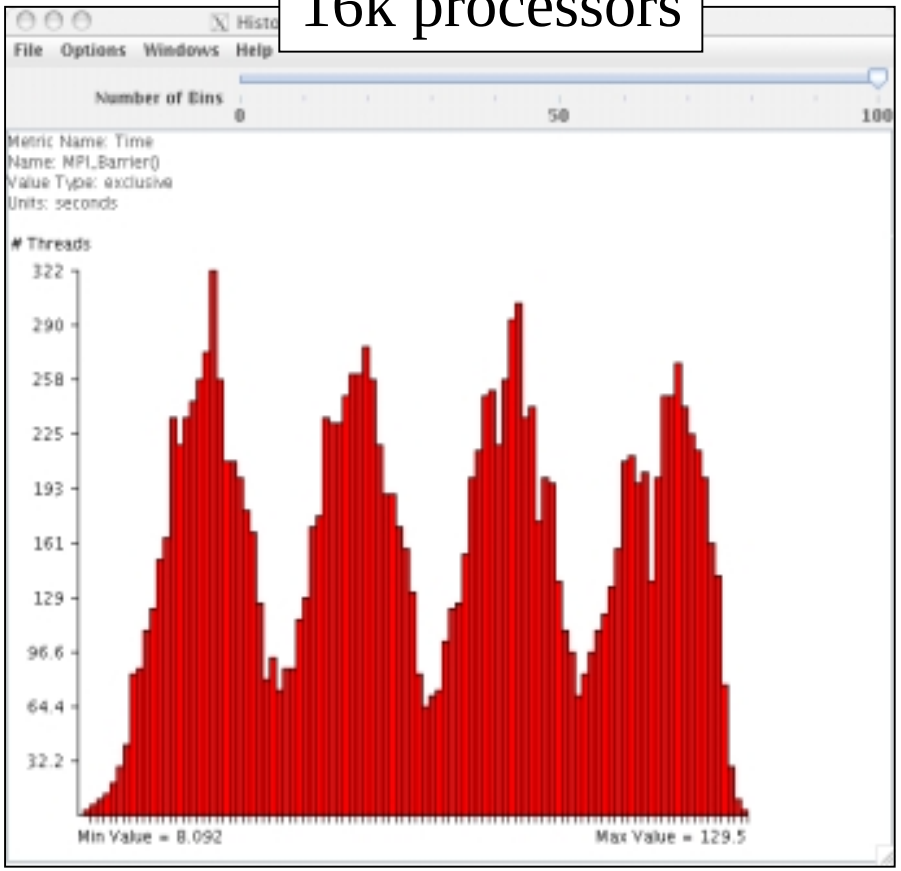
ParaProf – Histogram View (Miranda)

- Scalable 2D displays

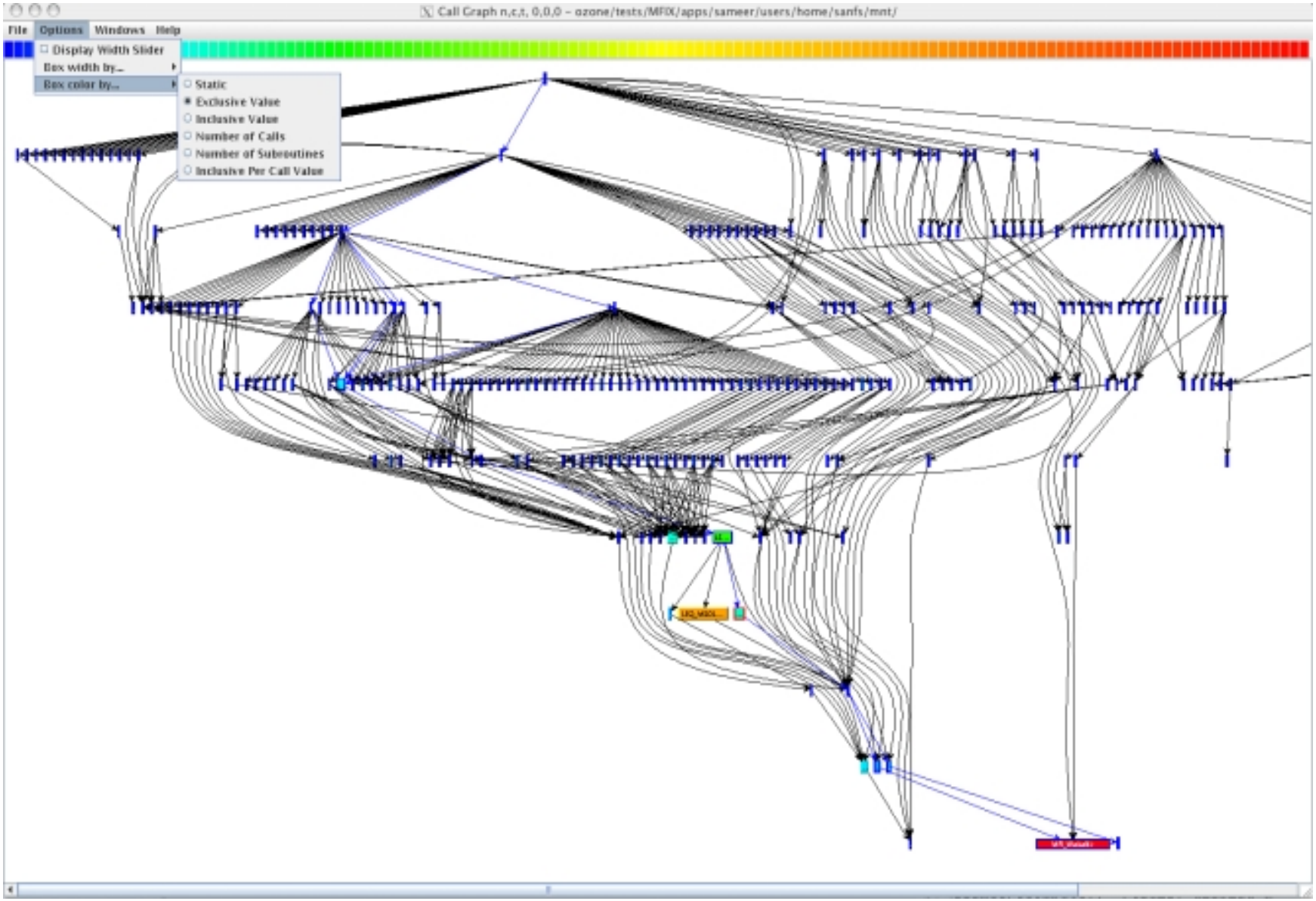
8k processors



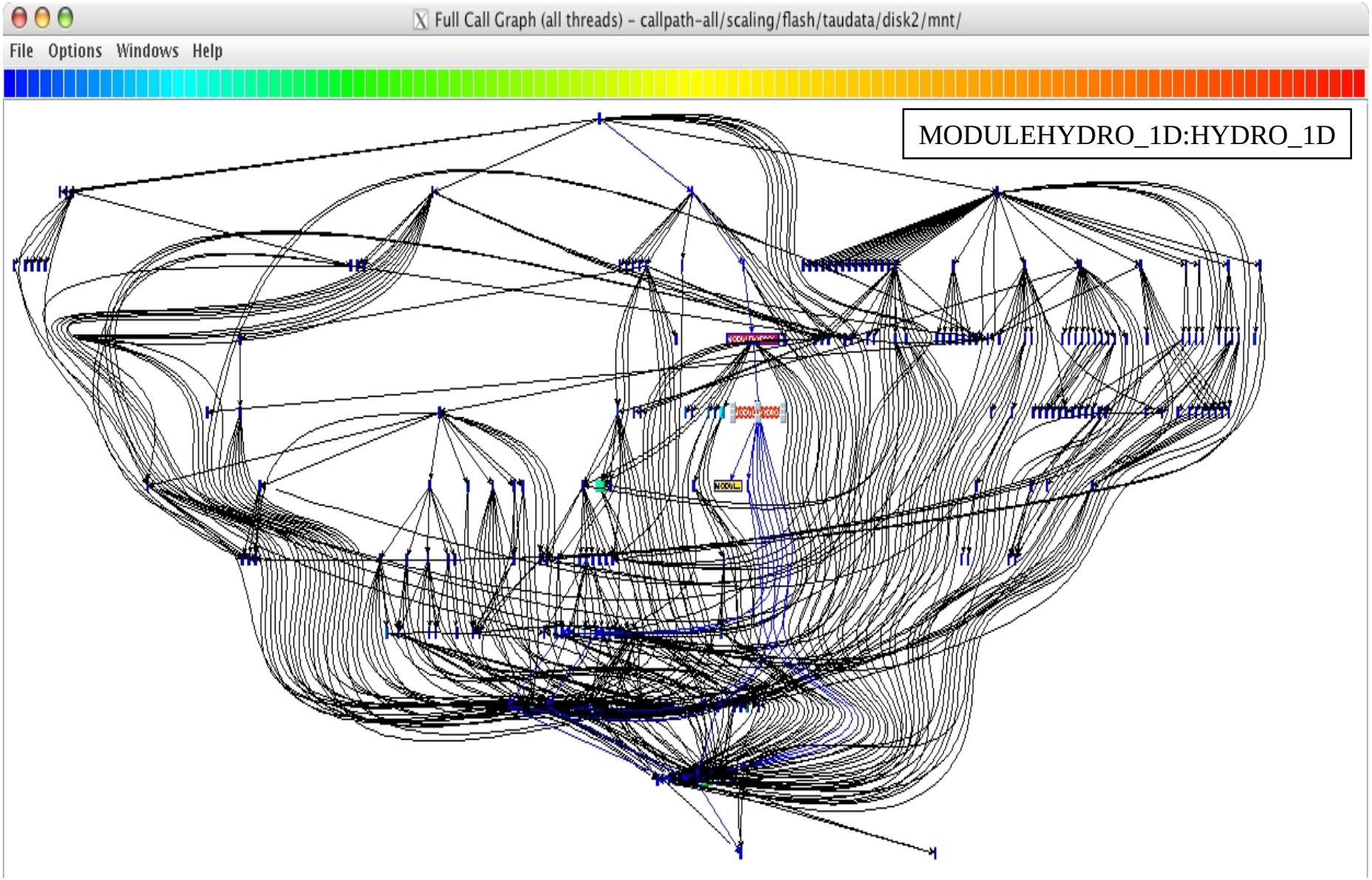
16k processors



ParaProf –Callgraph View (MFIX)

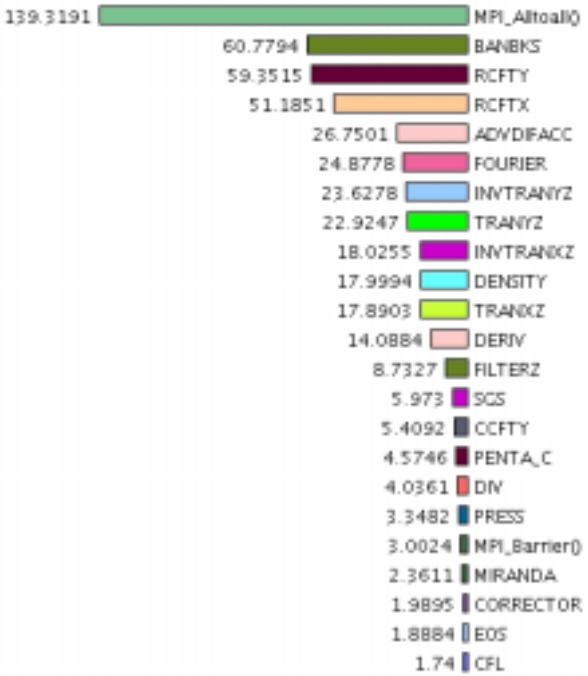


ParaProf – Callpath Highlighting (Flash)

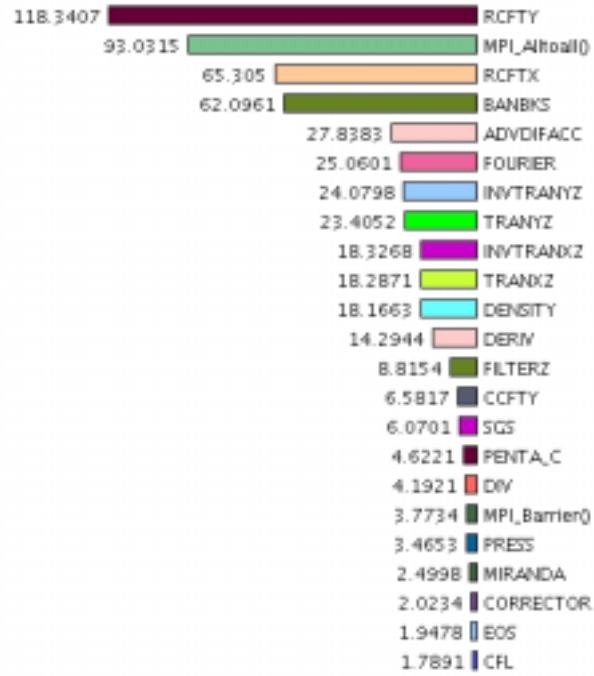


Profiling of Miranda on BG/L

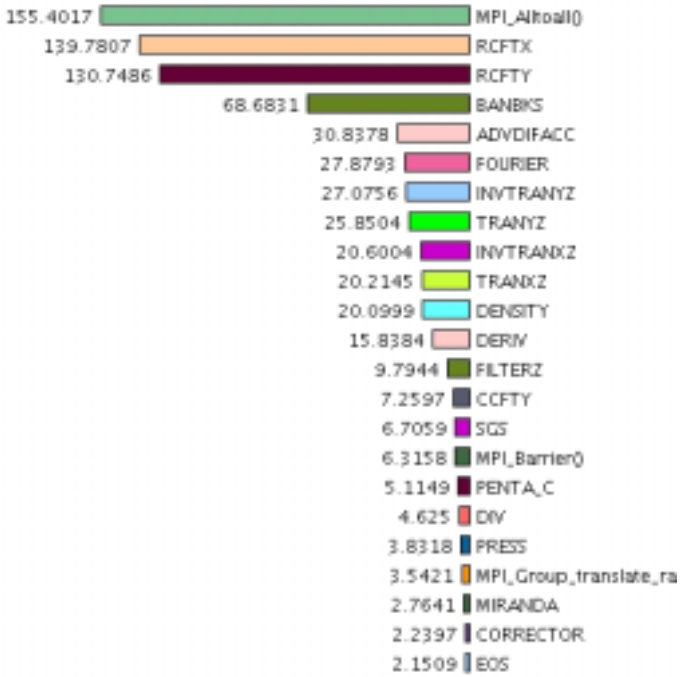
- Profile code performance (automatic instrumentation)
- Scaling studies (problem size, number of processors)



128 Nodes



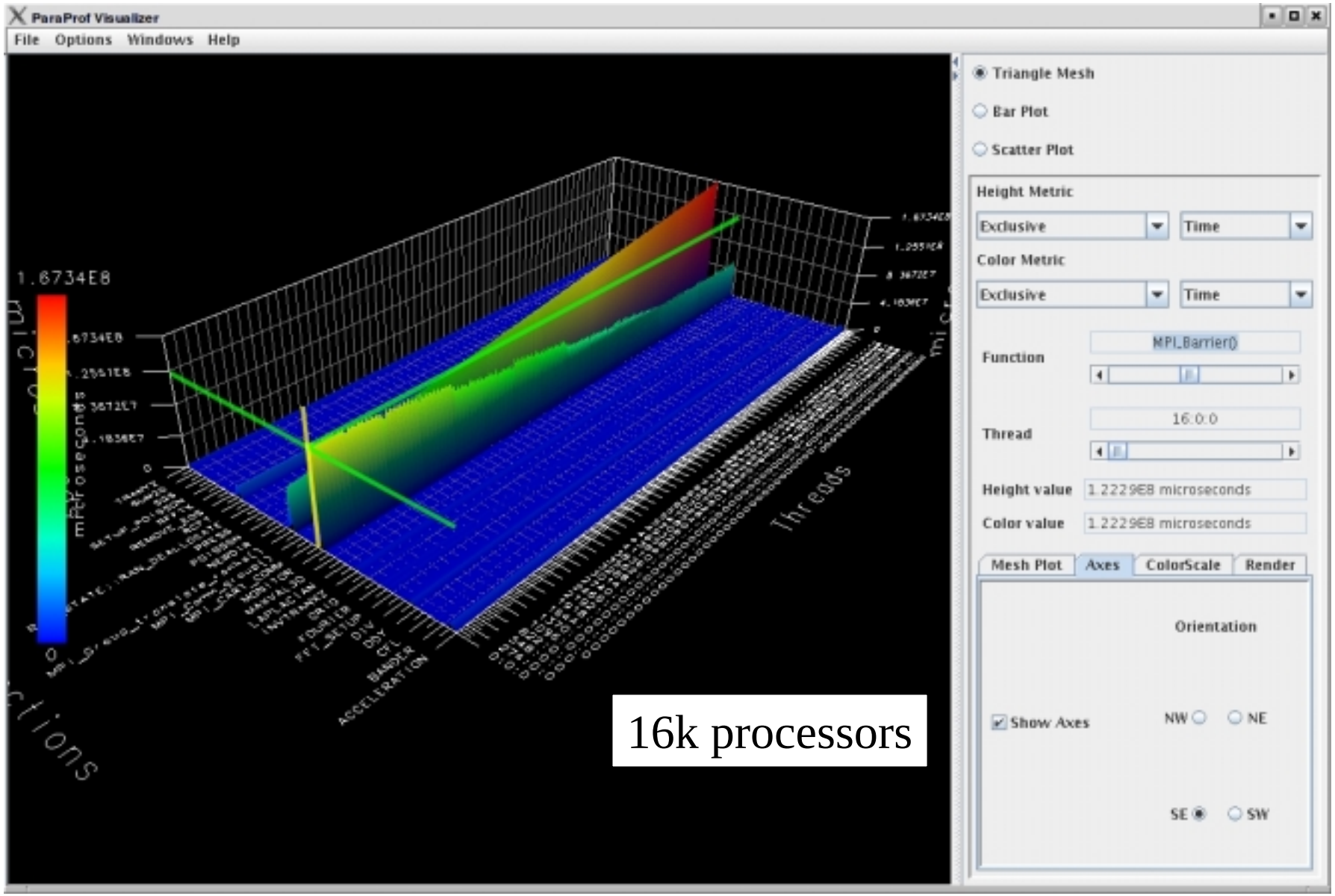
512 Nodes



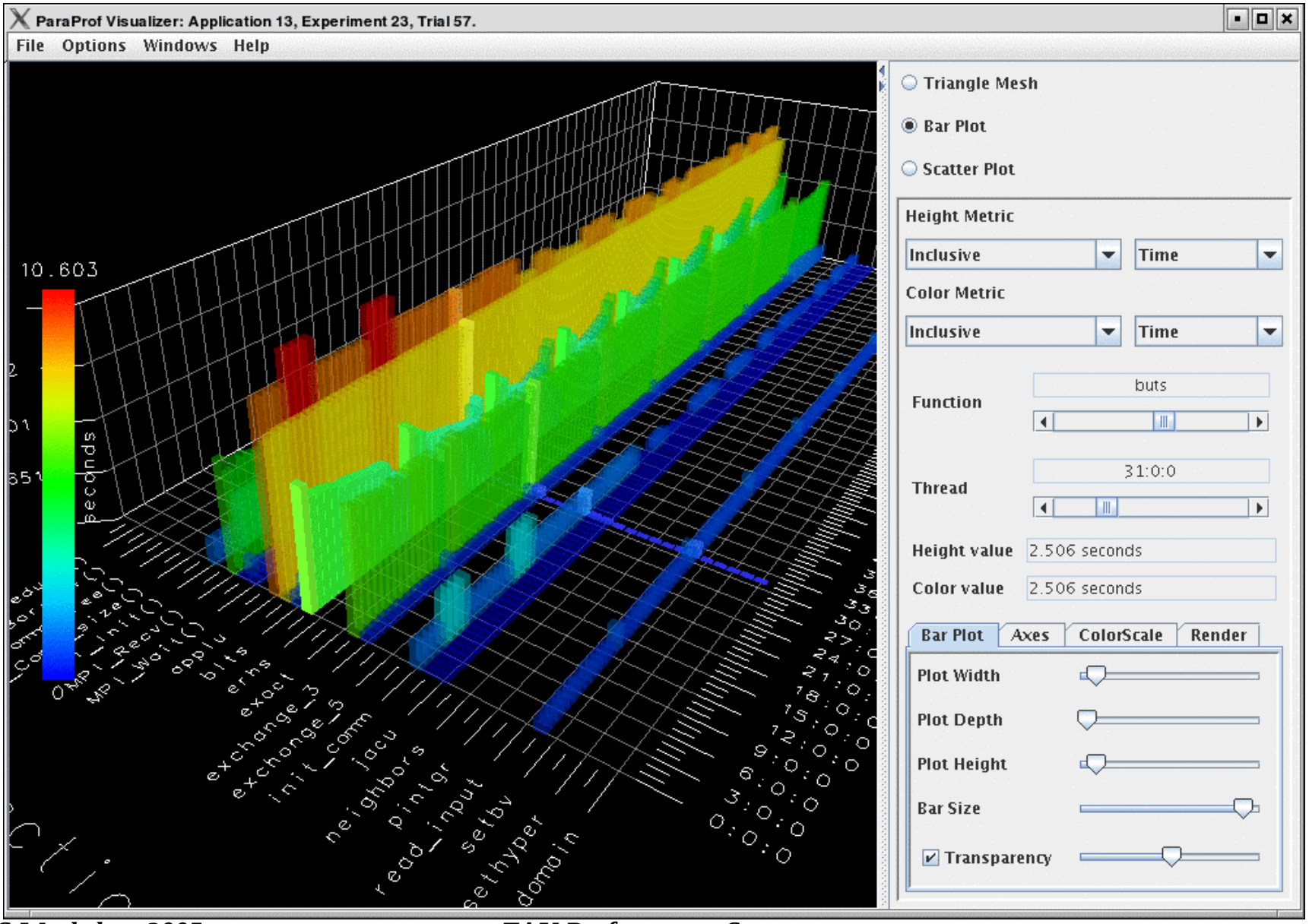
1024 Nodes

- Run on 8K,16K and 32K processors!

ParaProf – 3D Full Profile (Miranda)

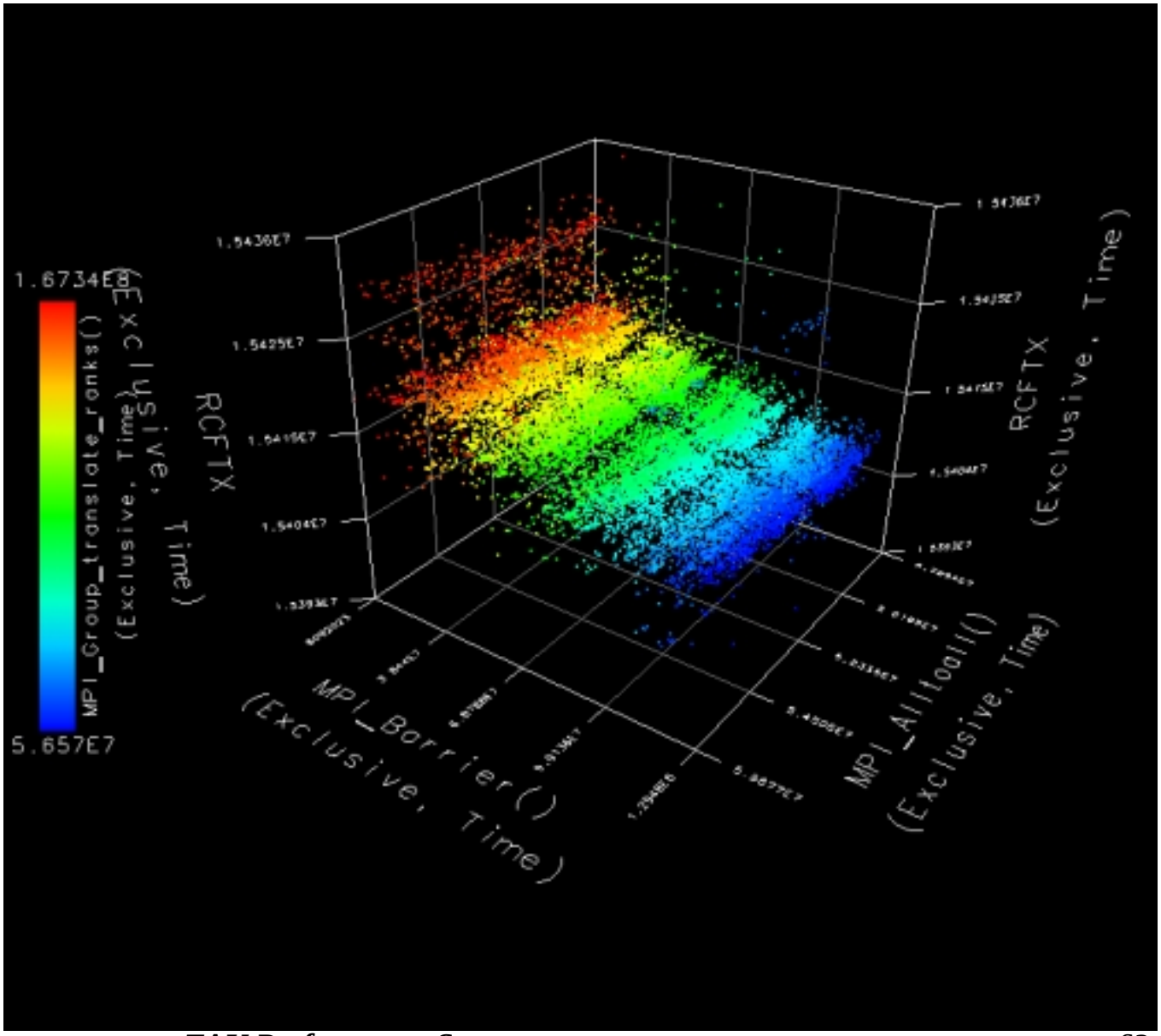


ParaProf Bar Plot (Zoom in/out +/-)



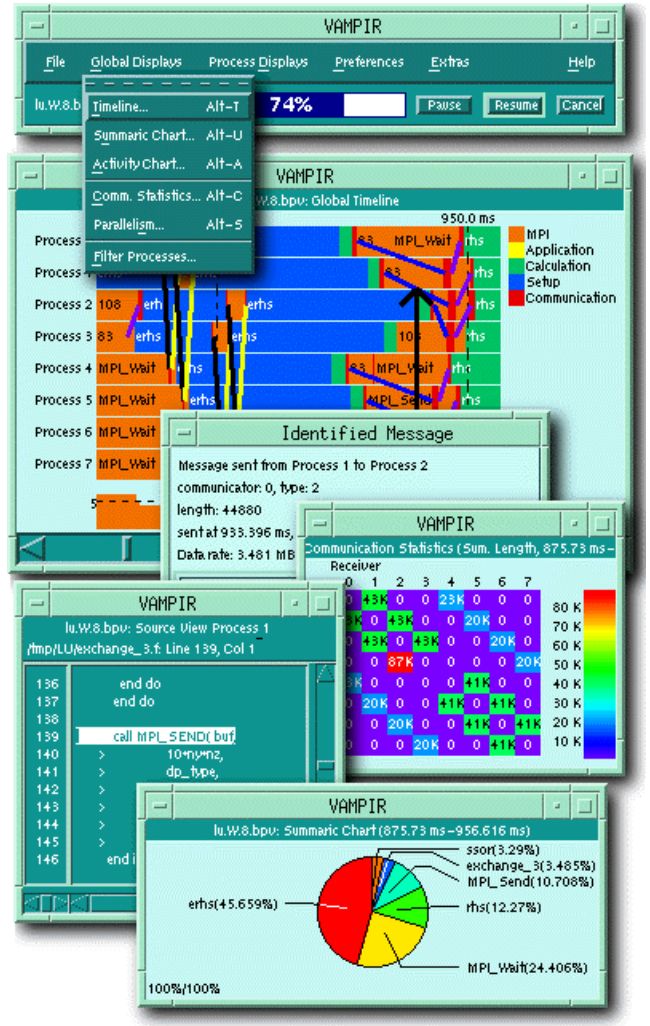
ParaProf – 3D Scatterplot (Miranda)

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



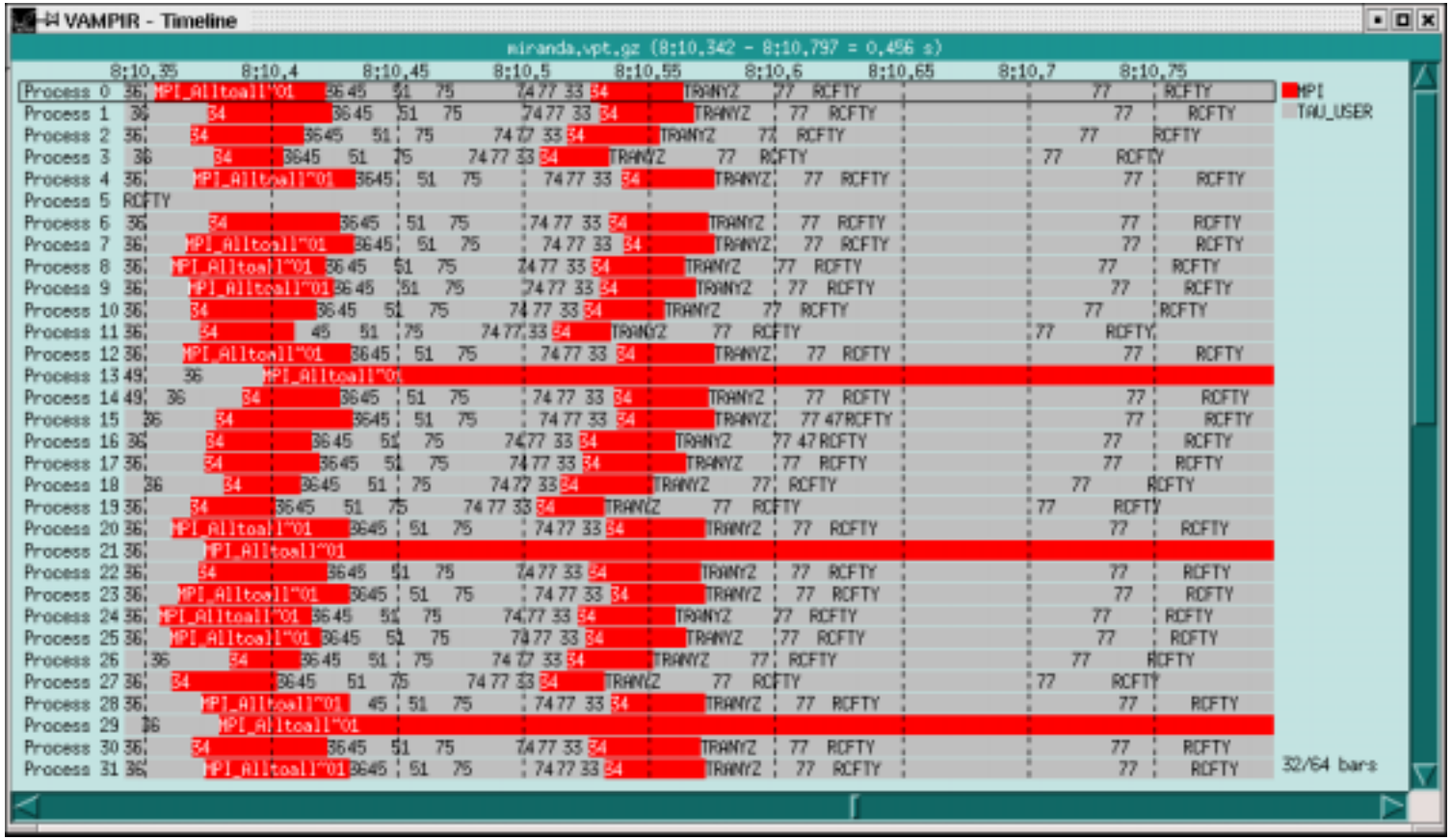
Vampir Trace Visualizer/Intel ® TraceAnalyzer 4

- Visualization and Analysis of MPI Programs
- Originally developed by Forschungszentrum Jülich
- Current development by Technical University Dresden, Germany
- Distributed by Intel
- **`http://www.vampir-ng.de`**

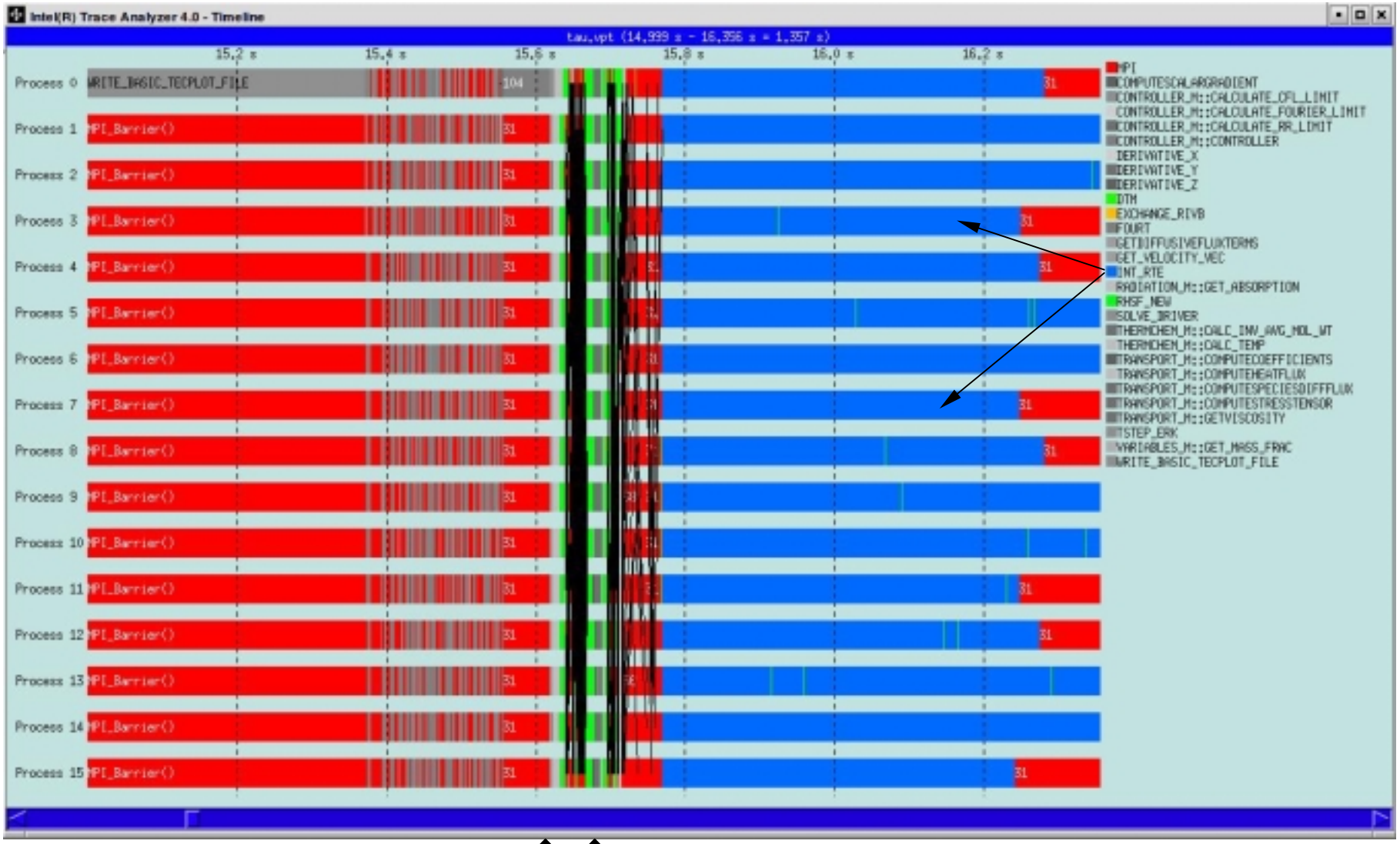


Performance Tracing on Miranda

- Use TAU to generate VTF3 traces for Vampir analysis
  - MPI calls with HW counter information (not shown)
  - Detailed code behavior to focus optimization efforts

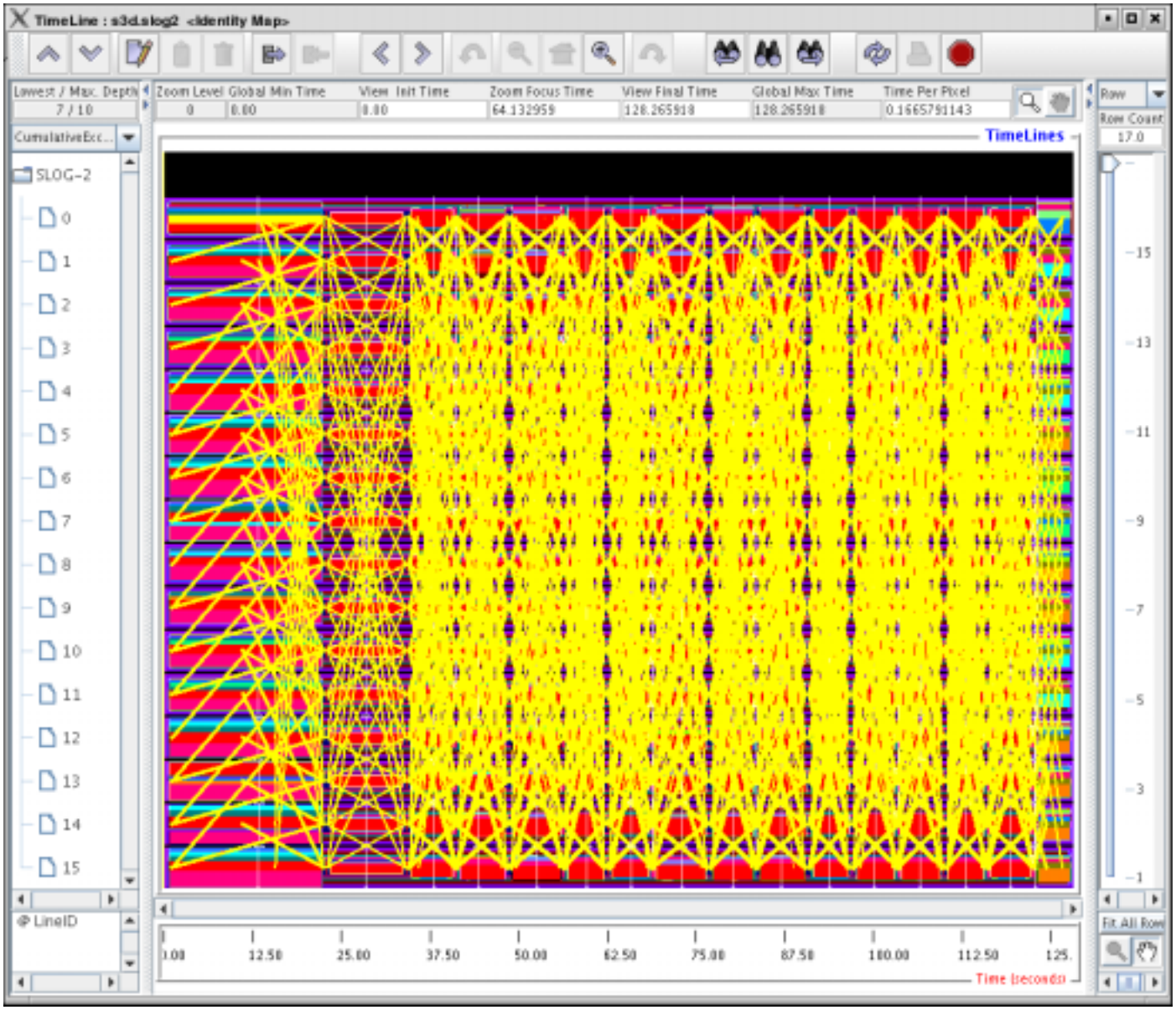


S3D on Lemieux (tau2vtf, Vampir)

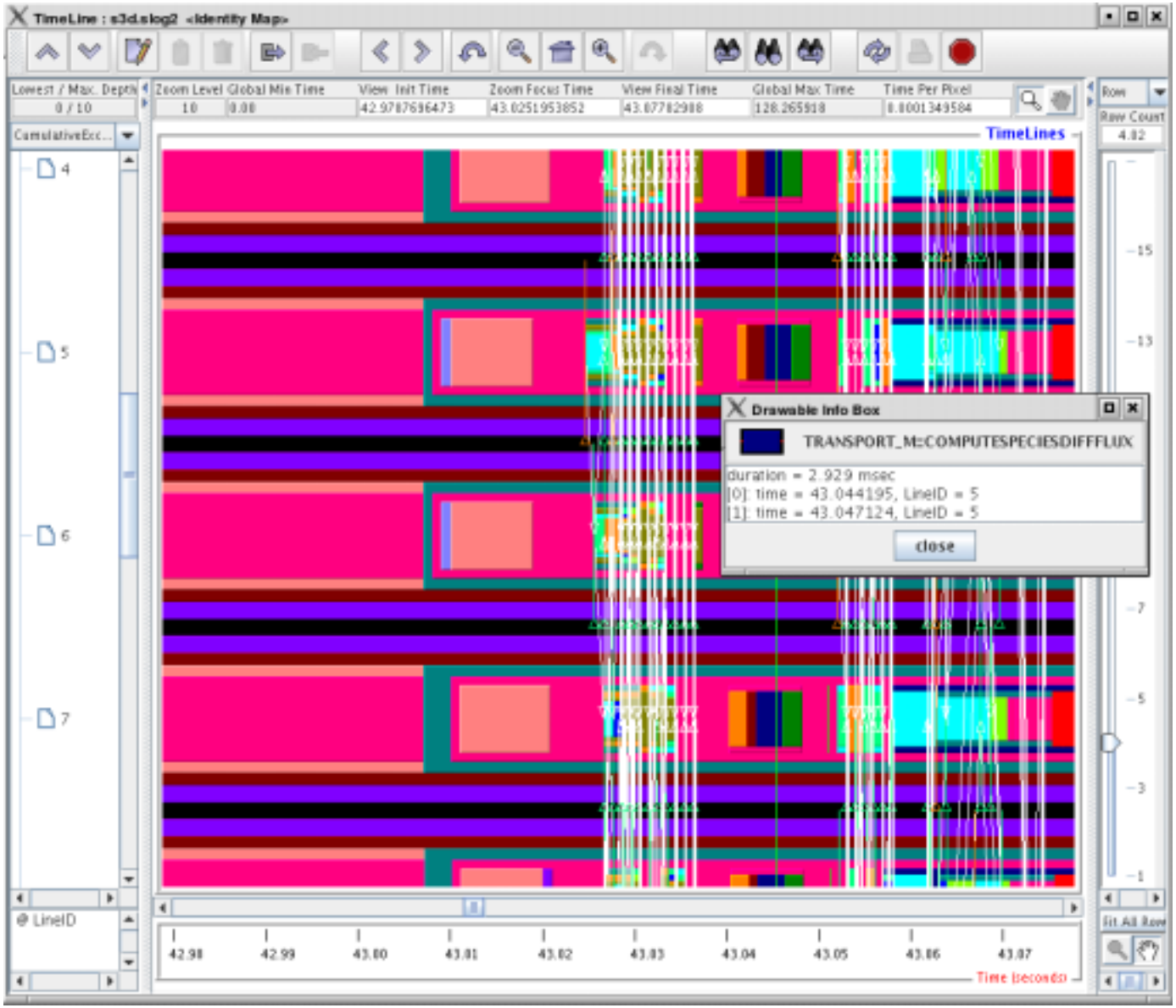


S3D on Lemieux (Zoomed)

Jumpshot Trace Visualizer [ANL] (S3D)



Jumpshot Trace Visualizer (S3D on Tru64)



TAU Performance System Status

- Computing platforms (selected)
  - IBM SP/pSeries/BGL, SGI Altix/Origin, Cray T3E/SV-1/X1/XT3, HP (Compaq) SC (Tru64), Sun, Hitachi SR8000, NEC SX-5/6, Linux clusters (IA-32/64, Alpha, PPC, PA-RISC, Power, Opteron), Apple (G4/5, OS X), Windows
- Programming languages
  - C, C++, Fortran 77/90/95, HPF, Java, Python
- Thread libraries (selected)
  - pthreads, SGI sproc, Java,Windows, OpenMP
- Compilers (selected)
  - Intel, PGI, GNU, Fujitsu, Sun, PathScale, SGI, Cray, IBM, HP, NEC, Absoft, Lahey, Nagware

Project Affiliations (selected)

- Center for Simulation of Accidental Fires and Explosion
  - University of Utah, ASCI ASAP Center, C-SAFE
  - Uintah Computational Framework (UCF) (C++)
- Center for Simulation of Dynamic Response of Materials
  - California Institute of Technology, ASCI ASAP Center
  - Virtual Testshock Facility (VTF) (Python, Fortran 90)
- Earth Systems Modeling Framework (ESMF)
  - NSF, NOAA, DOE, NASA, …
  - Instrumentation for ESMF framework and applications
  - C, C++, and Fortran 95 code modules
  - MPI wrapper library for MPI calls

Project Affiliations (selected) (continued)

- Lawrence Livermore National Lab
  - Hydrodynamics (Miranda)
- Sandia National Lab and Los Alamos National Lab
  - DOE CCTTSS SciDAC project
  - Common component architecture (CCA) integration
- Argonne National Lab
  - Jumpshot SLOG2 SDK project
  - ZeptoOS - scalable components for petascale architectures
  - KTAU - integration of TAU infrastructure in Linux kernel
- Oak Ridge National Lab
  - Contribution to the Joule Report: S3D, AORSA3D

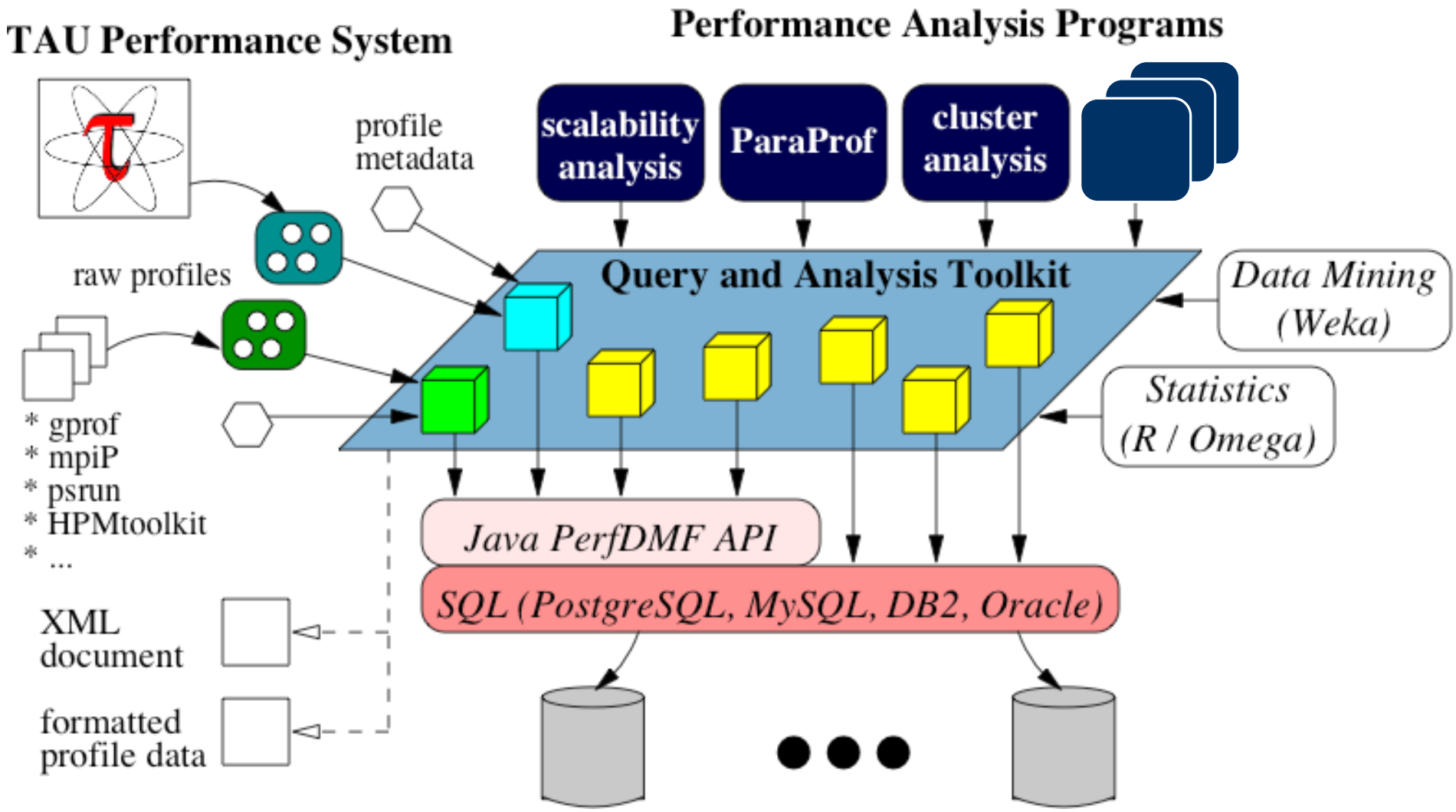
Important Questions for Application Developers

- How does performance vary with different compilers?
- Is poor performance correlated with certain OS features?
- Has a recent change caused unanticipated performance?
- How does performance vary with MPI variants?
- Why is one application version faster than another?
- What is the reason for the observed scaling behavior?
- Did two runs exhibit similar performance?
- How are performance data related to application events?
- Which machines will run my code the fastest and why?
- Which benchmarks predict my code performance best?

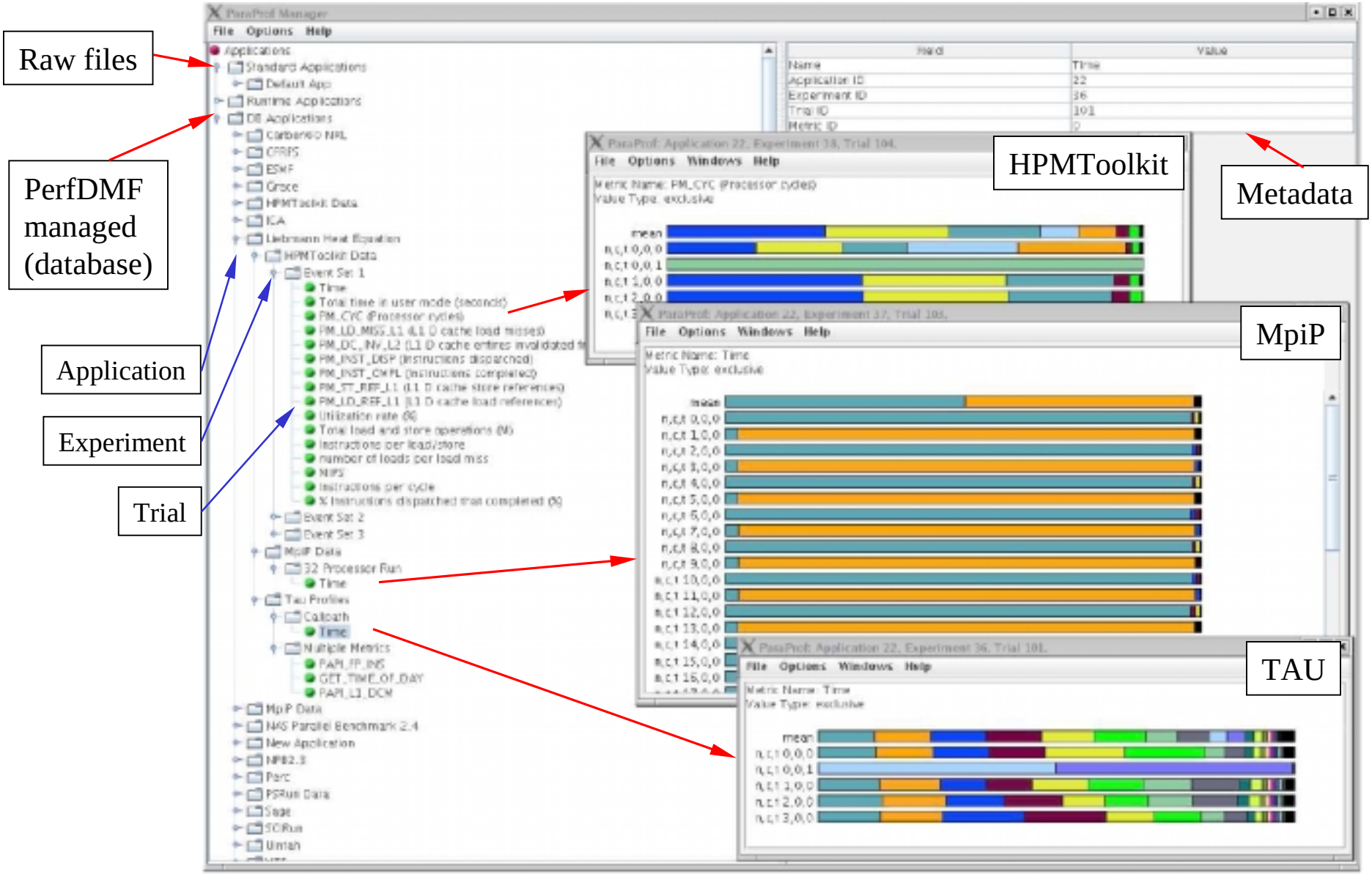
Performance Problem Solving Goals

- Answer questions at multiple levels of interest
  - Data from low-level measurements and simulations
    - use to predict application performance
  - High-level performance data spanning dimensions
    - machine, applications, code revisions, data sets
    - examine broad performance trends
- Discover general correlations application performance and features of their external environment
- Develop methods to predict application performance on lower-level metrics
- Discover performance correlations between a small set of benchmarks and a collection of applications that represent a typical workload for a given system

Performance Data Management Framework



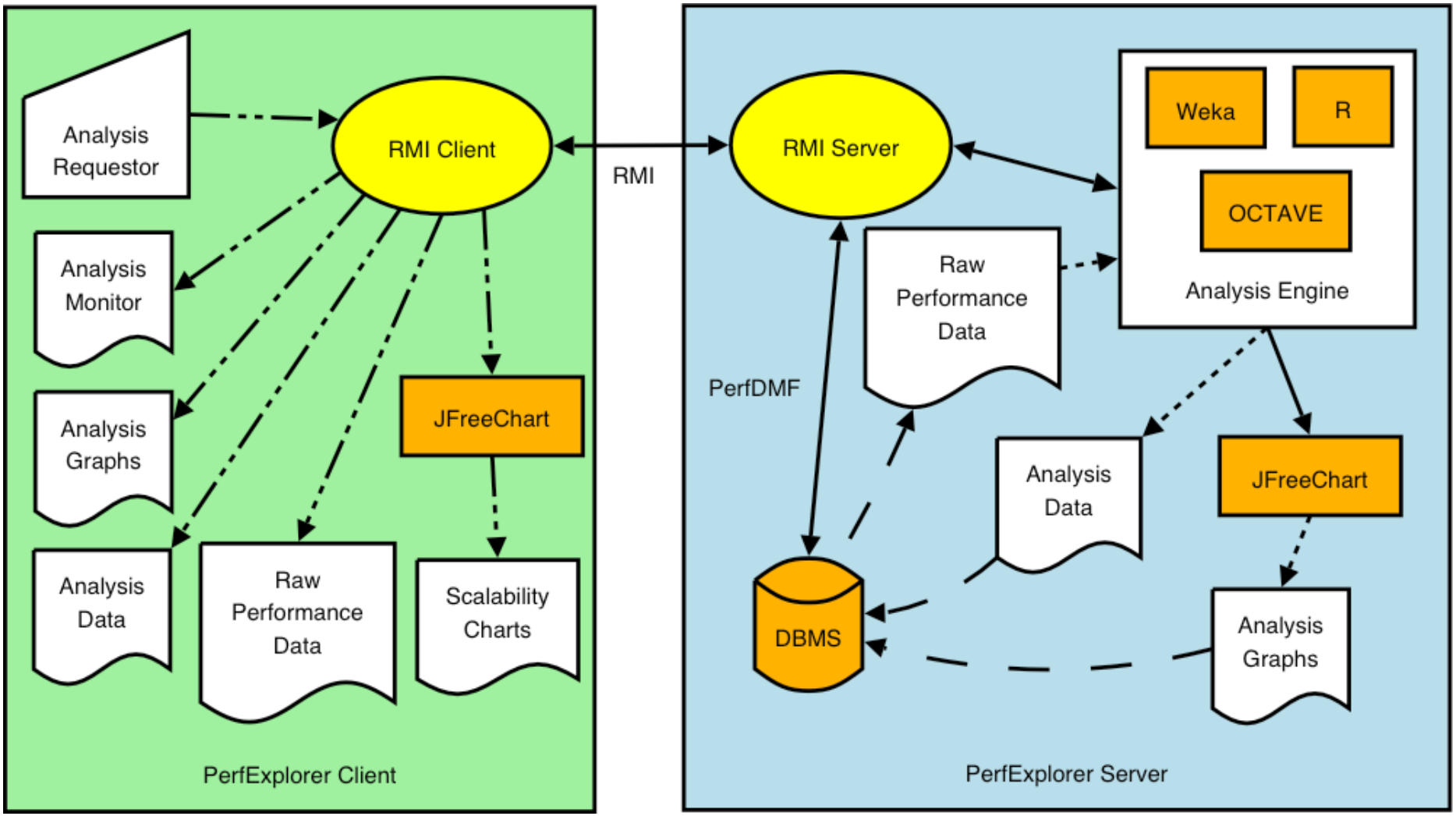
ParaProf Performance Profile Analysis



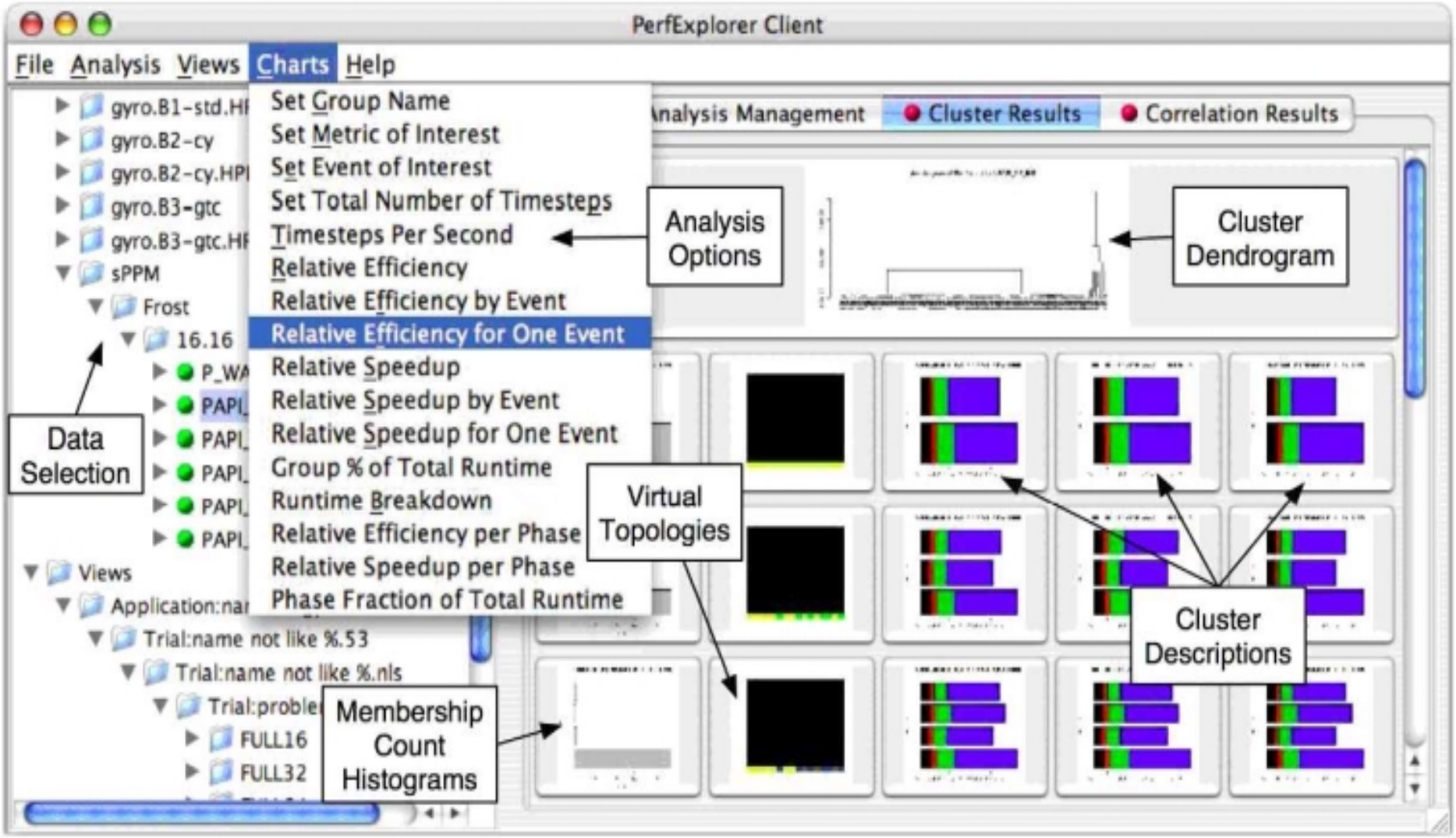
PerfExplorer

- Performance knowledge discovery framework
  - Use the existing TAU infrastructure
    - TAU instrumentation data, PerfDMF
  - Client-server based system architecture
  - Data mining analysis applied to parallel performance data
    - comparative, clustering, correlation, dimension reduction, ...
- Technology integration
  - Relational DatabaseManagement Systems (RDBMS)
  - Java API and toolkit
  - R-project / Omegahat statistical analysis
  - WEKA data mining package
  - Web-based client

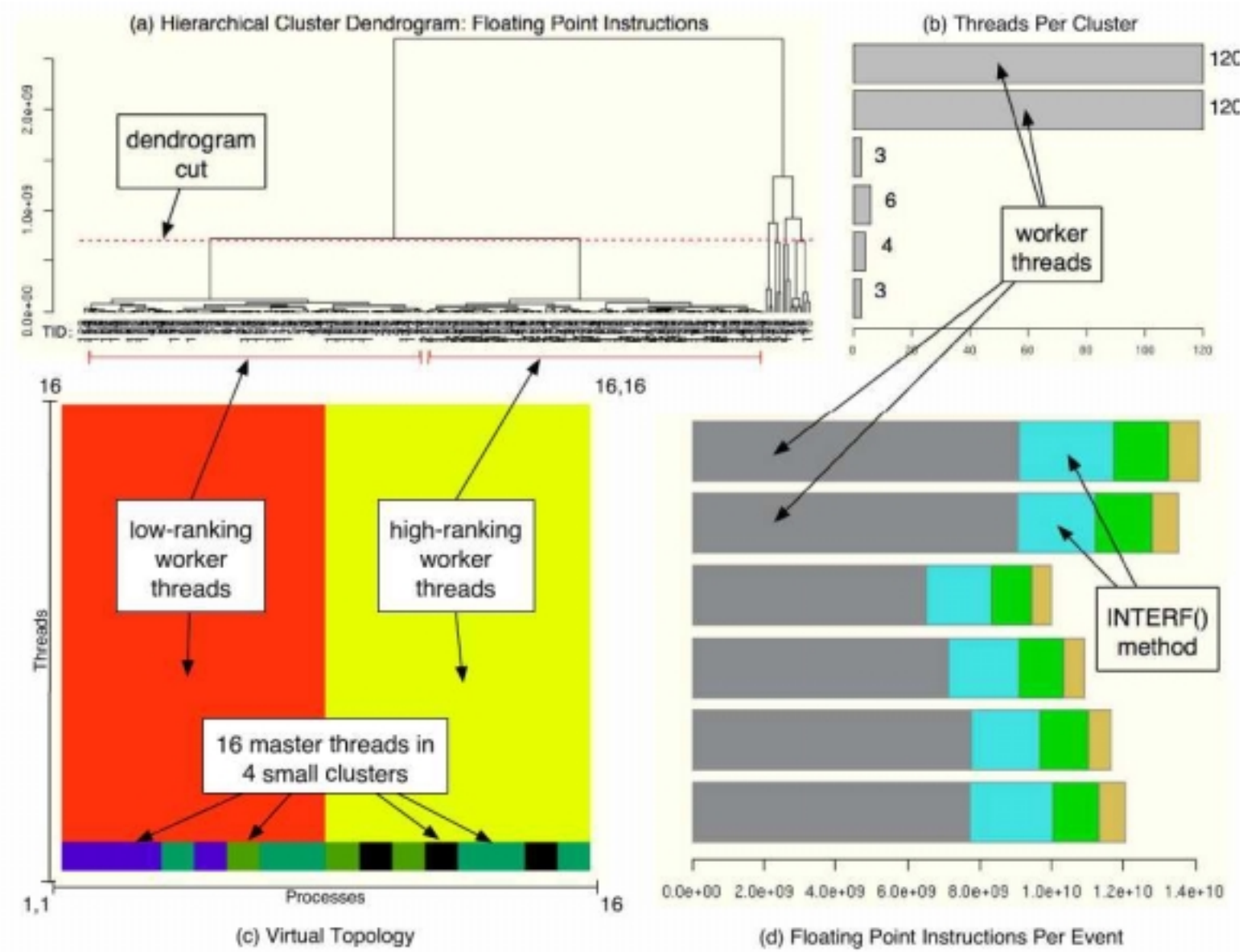
PerfExplorer Architecture



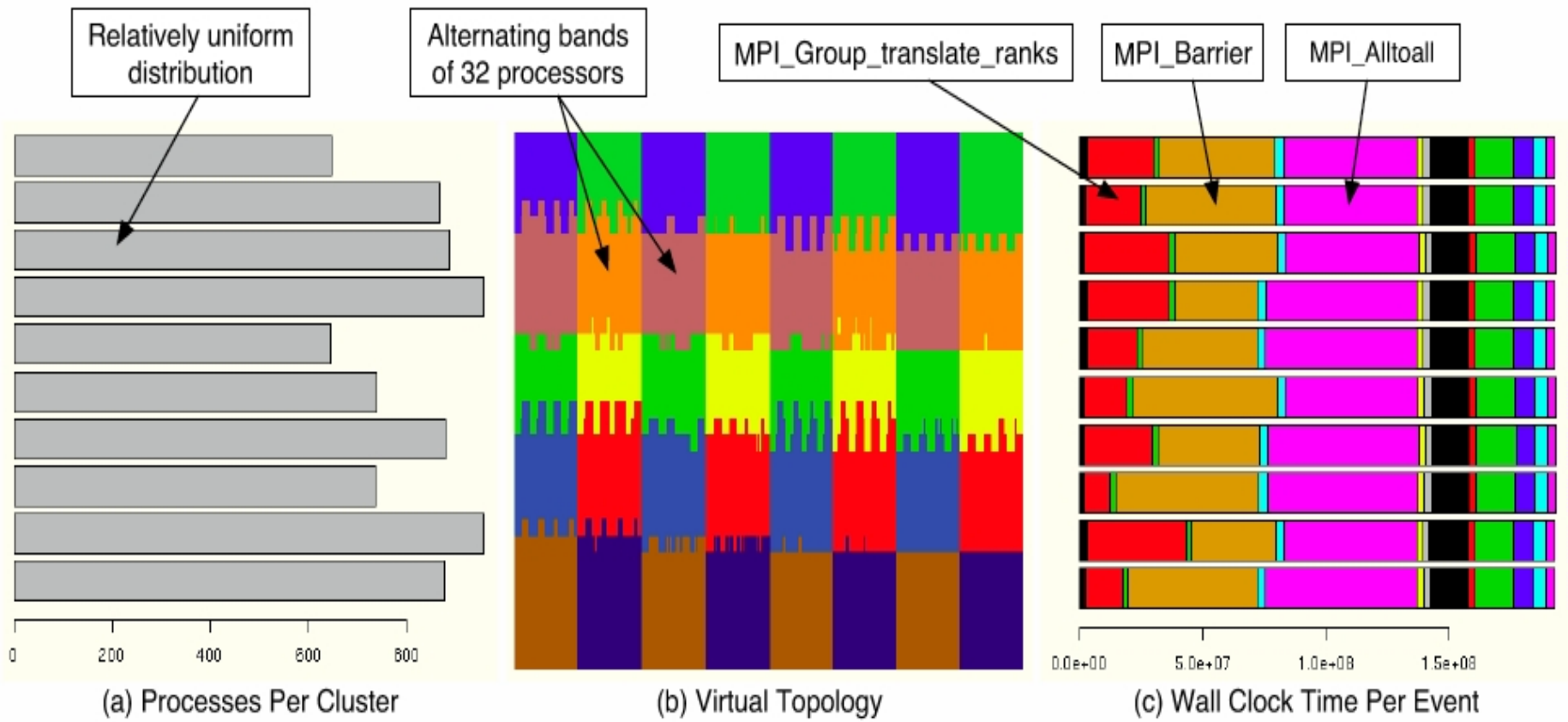
PerfExplorer Client GUI



Hierarchical and K-means Clustering (sPPM)



Miranda Clustering on 16K Processors



PERC Tool Requirements and Evaluation

- Performance Evaluation Research Center (PERC)
  - DOE SciDAC
  - Evaluation methods/tools for high-end parallel systems
- PERC tools study (led by ORNL, Pat Worley)
  - In-depth performance analysis of select applications
  - Evaluation performance analysis requirements
  - Test tool functionality and ease of use
- Applications
  - Start with fusion code – GYRO
  - Repeat with other PERC benchmarks
  - Continue with SciDAC codes

Primary Evaluation Machines

- *Phoenix* (ORNL – Cray X1)
  - 512 multi-streaming vector processors
- *Ram* (ORNL – SGI Altix (1.5 GHz Itanium2))
  - 256 total processors
- *TeraGrid*
  - ~7,738 total processors on 15 machines at 9 sites
- *Cheetah* (ORNL – p690 cluster (1.3 GHz, HPS))
  - 864 total processors on 27 compute nodes
- *Seaborg* (NERSC – IBM SP3)
  - 6080 total processors on 380 compute nodes

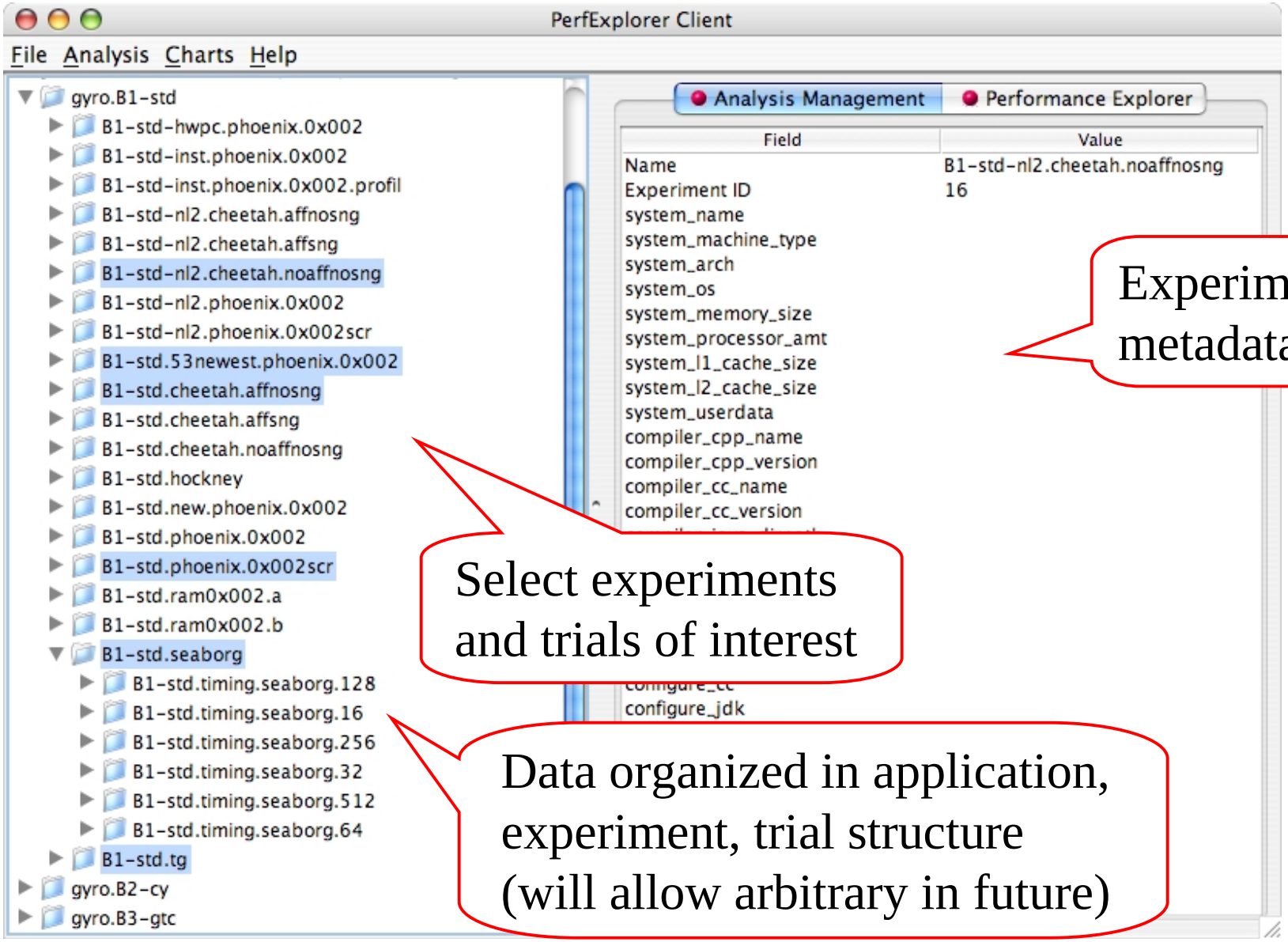
GYRO Execution Parameters

- Three benchmark problems
  - ***B1-std*** : 16n processors, 500 timesteps
  - ***B2-cy*** : 16n processors, 1000 timesteps
  - ***B3-gtc*** : 64n processors, 100 timesteps (very large)
- Test different methods to evaluate nonlinear terms:
  - Direct method
  - FFT (“nl2” for B1 and B2, “nl1” for B3)
- Task affinity enabled/disabled (p690 only)
- Memory affinity enabled/disabled (p690 only)
- Filesystem location (Cray X1 only)

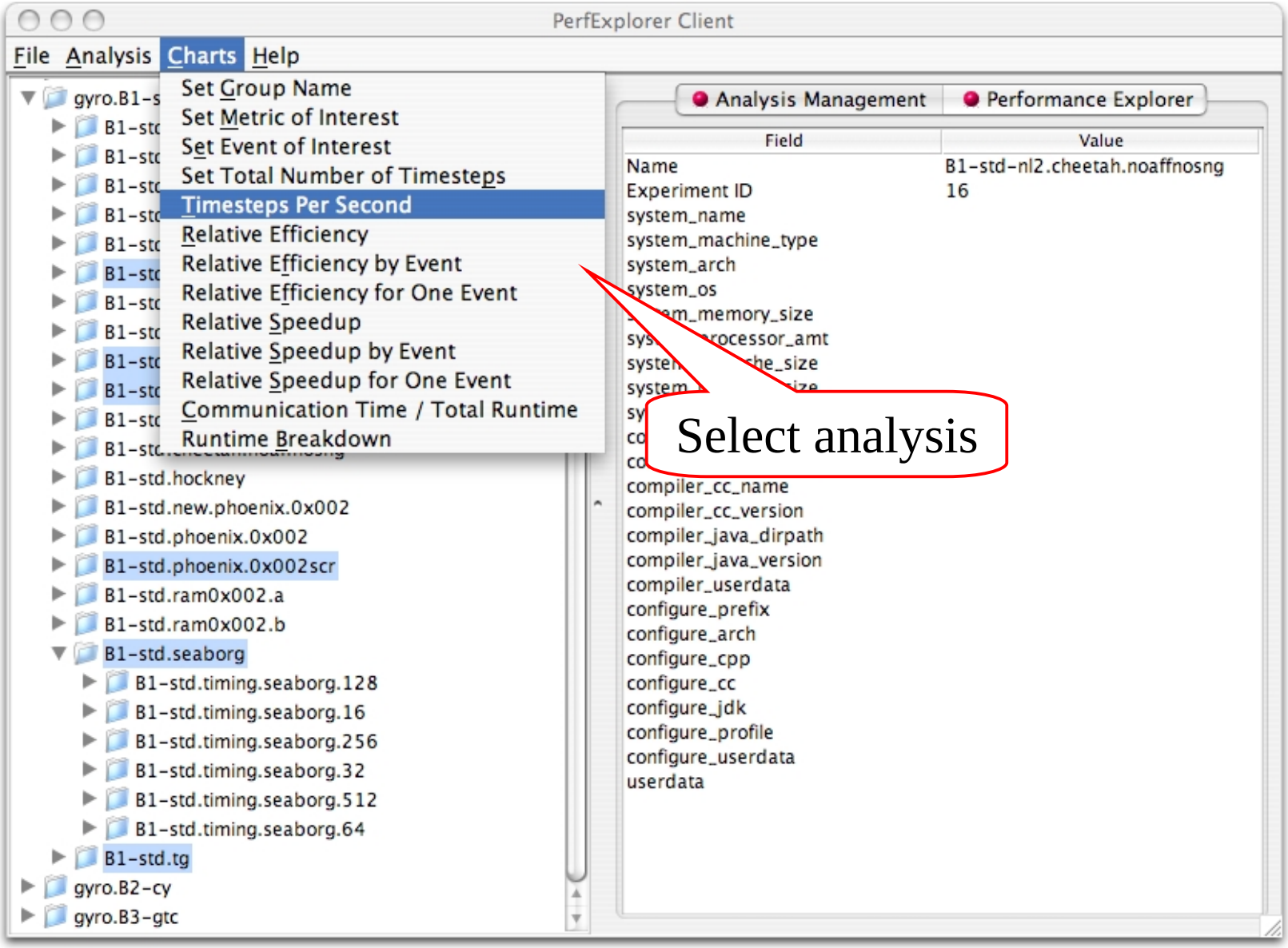
PerfExplorer Analysis of Self-Instrumented Data

- PerfExplorer
  - Focus on comparative analysis
  - Apply to PERC tool evaluation study
- Look at user timer data
  - Aggregate data
    - no per process data
    - process clustering analysis is not applicable
  - Timings output every *N* timesteps
    - some phase analysis possible
- Goal
  - Recreate manually generated performance reports

PerfExplorer Interface

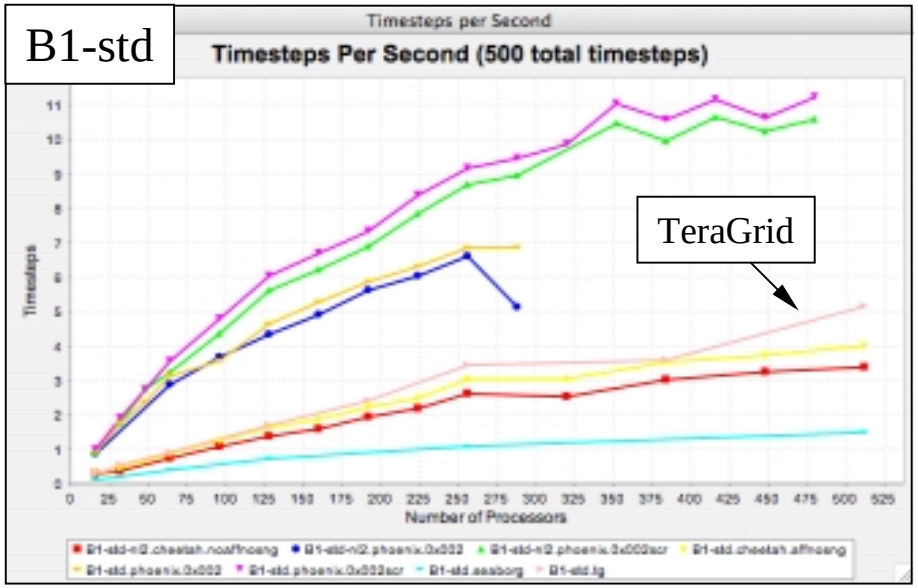


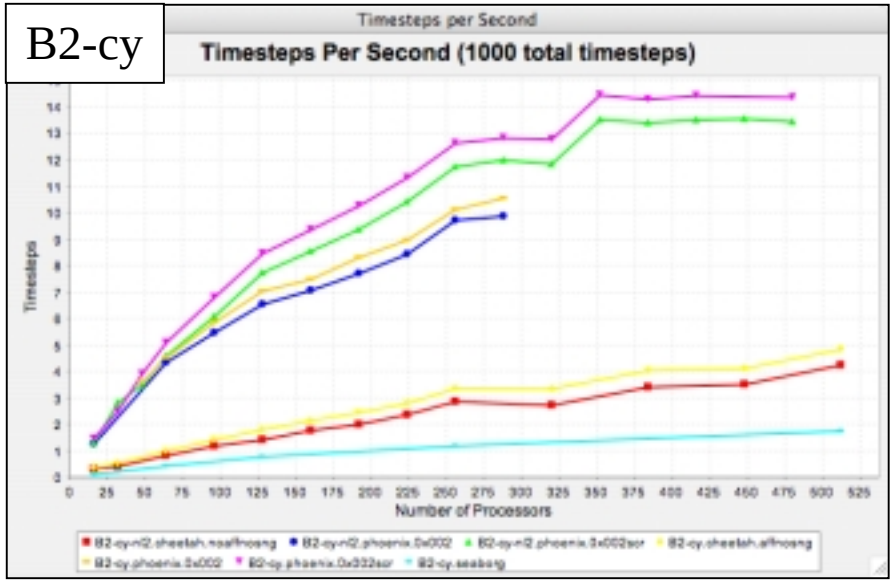
PerfExplorer Interface

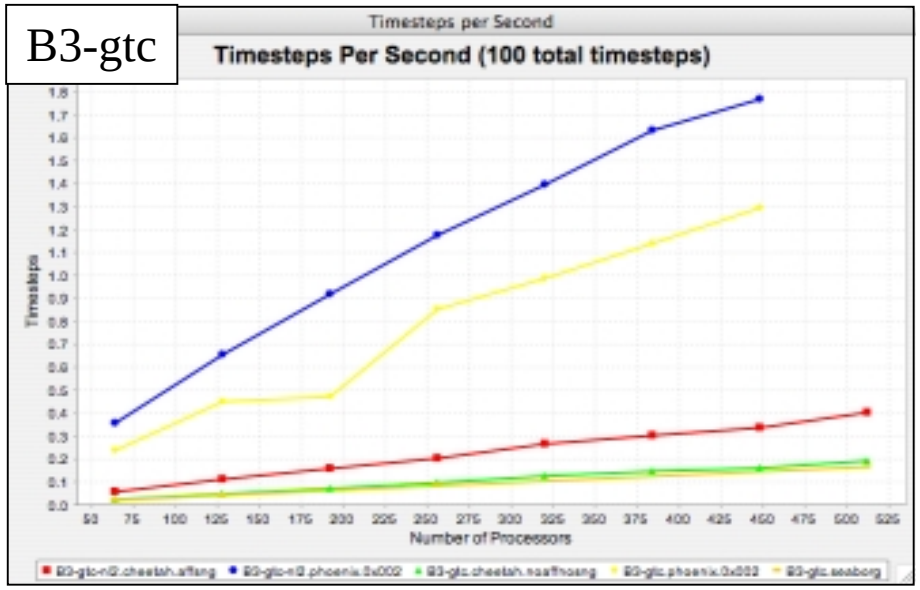


Timesteps per Second

- Cray X1 is the fastest to solution in all 3 tests
- FFT (nl2) improves time for B3-gtc only
- TeraGrid faster than p690 for B1-std?
- Plots generated automatically

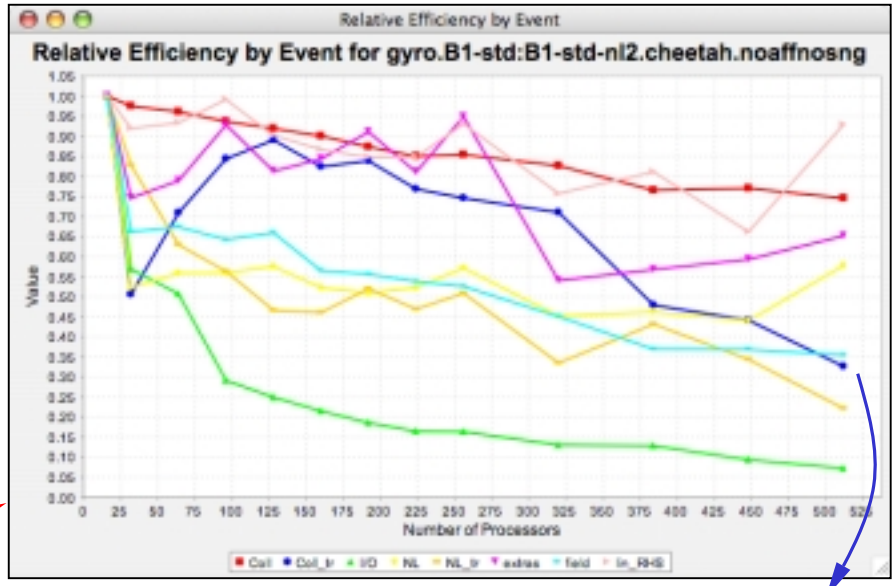


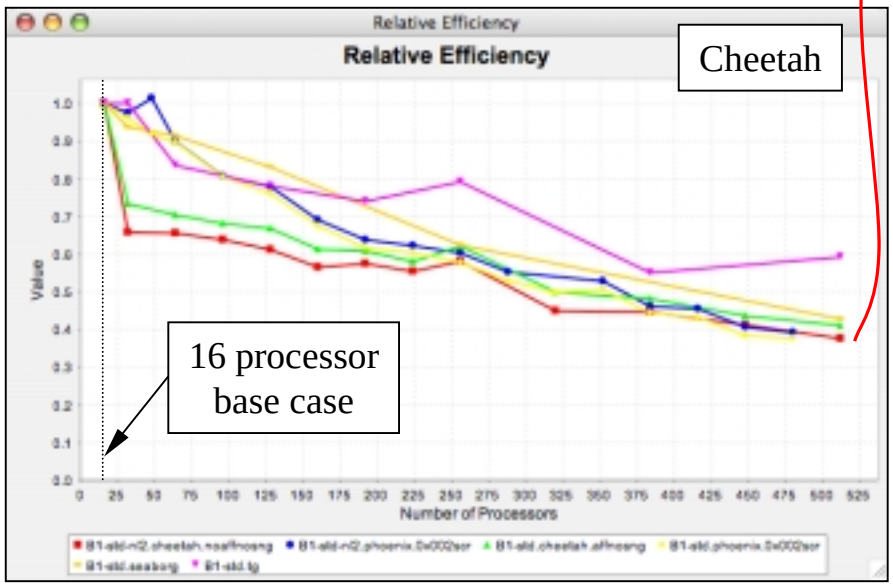


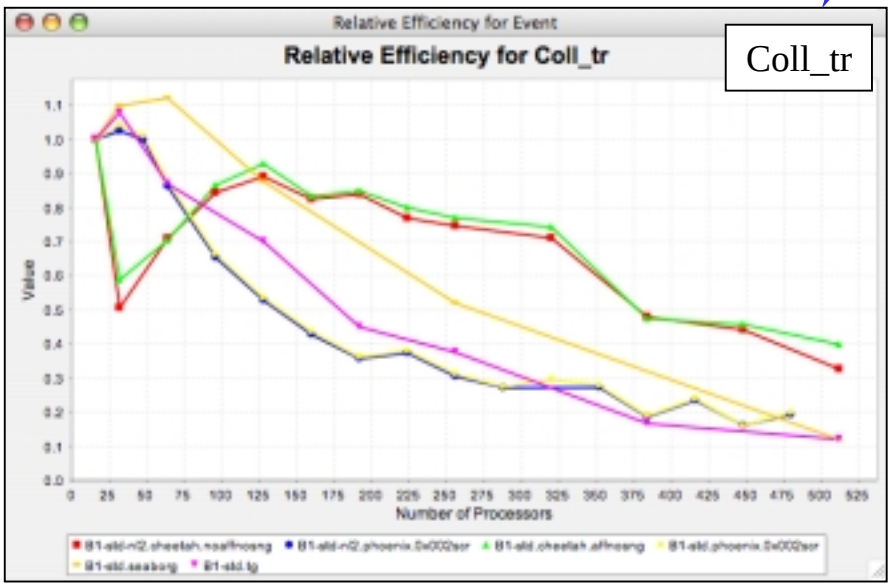


Relative Efficiency (B1-std)

- By experiment (B1-std)
  - Total runtime (Cheetah (red))
- By event for one experiment
  - Coll_tr (blue) is significant
- By experiment for one event
  - Shows how Coll_tr behaves for all experiments







Current and Future Work

- ParaProf
  - Developing phase-based performance displays
- PerfDMF
  - Adding new database backends and distributed support
  - Building support for user-created tables
- PerfExplorer
  - Extending comparative and clustering analysis
  - Adding new data mining capabilities
  - Building in scripting support
- Performance regression testing tool (PerfRegress)
- Integrate in Eclipse Parallel Tool Project (PTP)

Concluding Discussion

- Performance tools must be used effectively
- More intelligent performance systems for productive use
  - Evolve to application-specific performance technology
  - Deal with scale by “full range” performance exploration
  - Autonomic and integrated tools
  - Knowledge-based and knowledge-driven process
- Performance observation methods do not necessarily need to change in a fundamental sense
  - More automatically controlled and efficiently use
- Develop next-generation tools and deliver to community
- Open source with support by ParaTools, Inc.
- http://www.cs.uoregon.edu/research/tau

Hands-On Session

- Login to odin.cs.indiana.edu and get software

% cp /san/cca/tau/tautraining.tar.gz .

% tar zxf tautraining.tar.gz

- Follow instructions in the README file

Support Acknowledgements

- Department of Energy (DOE)
  - Office of Science contracts
  - University of Utah ASCI Level 1 sub-contract
  - ASC/NNSA Level 3 contract
- NSF
  - High-End Computing Grant
- Research Centre Juelich
  - John von Neumann Institute
  - Dr. Bernd Mohr
- Los Alamos National Laboratory

QuickTime™ and a
TIFF (Uncompressed) decompressor
are needed to see this picture.