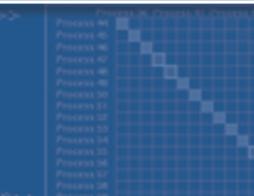




SOFTWARE



0.00 <<time step loop>>
0.00 updatedt
6.62 updateX
372.85 updateLEN
0.00 gene
0.00 <<iteration loop>>
293.65 genBC



PRODUCTIVITY

FAST SOLUTIONS

- PAPI\_L1\_DCM
- PAPI\_L1\_ICM
- PAPI\_L2\_DCM
- PAPI\_L2\_ICM
- PAPI\_L1\_TCM
- PAPI\_L2\_TCM

# TAU PERFORMANCE SYSTEM

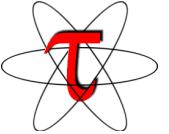
Sameer Shende

Chee Wai Lee, Wyatt Spear, Scott Biersdorff, Suzanne Millstein  
Performance Research Lab

Allen D. Malony, Nick Chaimov, William Voorhees  
Department of Computer and Information Science  
University of Oregon

Presentation at VI-HPS TW9: U. Versailles,  
St-Quentin-en-Yvelines, France, April 25, 2012, 9am,





- Tuning and Analysis Utilities (18+ year project)
- Comprehensive performance profiling and tracing
  - Integrated, scalable, flexible, portable
  - Targets all parallel programming/execution paradigms
- Integrated performance toolkit
  - Instrumentation, measurement, analysis, visualization
  - Widely-ported performance profiling / tracing system
  - Performance data management and data mining
  - Open source (BSD-style license)
- Easy to integrate in application frameworks

*<http://tau.uoregon.edu>*

- TAU is a performance evaluation tool
- It supports parallel profiling and tracing
- Profiling shows you how much (total) time was spent in each routine
- Tracing shows you *when* the events take place in each process along a timeline
- Profiling and tracing can measure time as well as hardware performance counters (cache misses, instructions) from your CPU
- TAU can automatically instrument your source code using a package called PDT for routines, loops, I/O, memory, phases, etc.
- TAU runs on most HPC platforms and it is free (BSD style license)
- TAU has instrumentation, measurement and analysis tools
  - paraprof is TAU's 3D profile browser
- **To use TAU's automatic source instrumentation, you may set a couple of environment variables and substitute the name of your compiler with a TAU shell script**

- How much time is spent in each application routine and outer **loops**? Within loops, what is the contribution of each **statement**?
- How many instructions are executed in these code regions? Floating point, Level 1 and 2 **data cache misses**, hits, branches taken?
- What is the **peak heap memory** usage of the code? When and where is memory allocated/de-allocated? Are there any memory leaks?
- How much time does the application spend performing **I/O**? What is the peak read and write **bandwidth** of individual calls, total volume?
- What is the contribution of different **phases** of the program? What is the time wasted/spent waiting for collectives, and I/O operations in Initialization, Computation, I/O phases?
- How does the application **scale**? What is the efficiency, runtime breakdown of performance across different core counts?

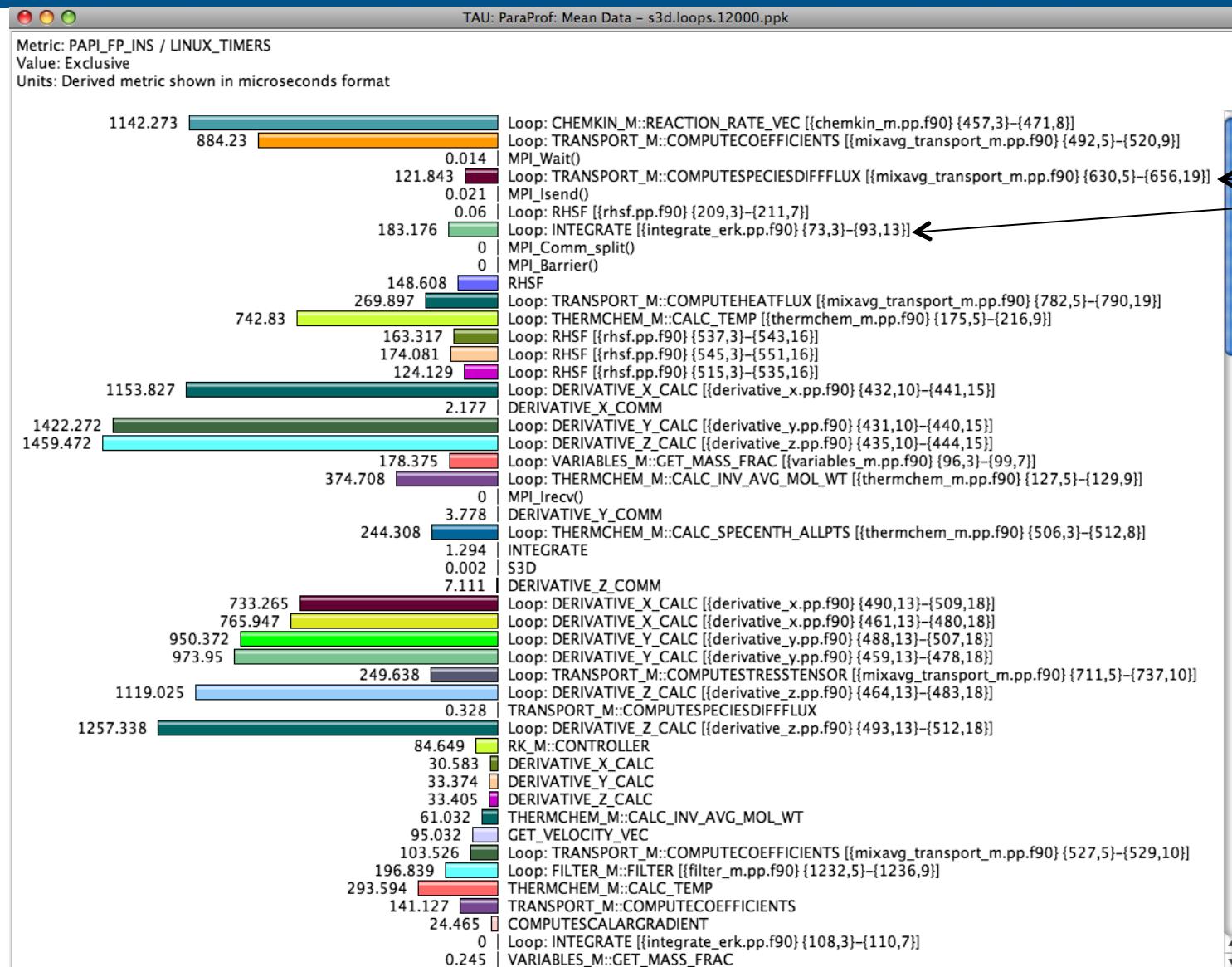
- Uninstrumented MPI code:
  - % ccc\_mprun -n 8 ./a.out
- With TAU:
  - % module load tau
  - % ccc\_mprun -n 8 tau\_exec ./a.out
  - % paraprof
- Shows only MPI data. How can we see application functions?

- Support for dynamic executables
- Specify the TAU measurement library to be injected
- Simplify the usage of TAU:
  - To instrument:  
`% tau_rewrite -T <options> a.out -o a.inst`
  - To perform measurements, execute the application:  
`% ccc_mprun -n 8 ./a.inst`
  - To analyze the data:  
`% paraprof`

- Simplify the usage of TAU:
  - To instrument an Intel C+MPI program:  
`% tau_rewrite a.out a.inst`
  - To perform measurements, execute the application:  
`% ccc_mprun -n 8 ./a.inst`
  - To analyze the data:  
`% paraprof`

# ParaProf: Mflops Sorted by Exclusive Time

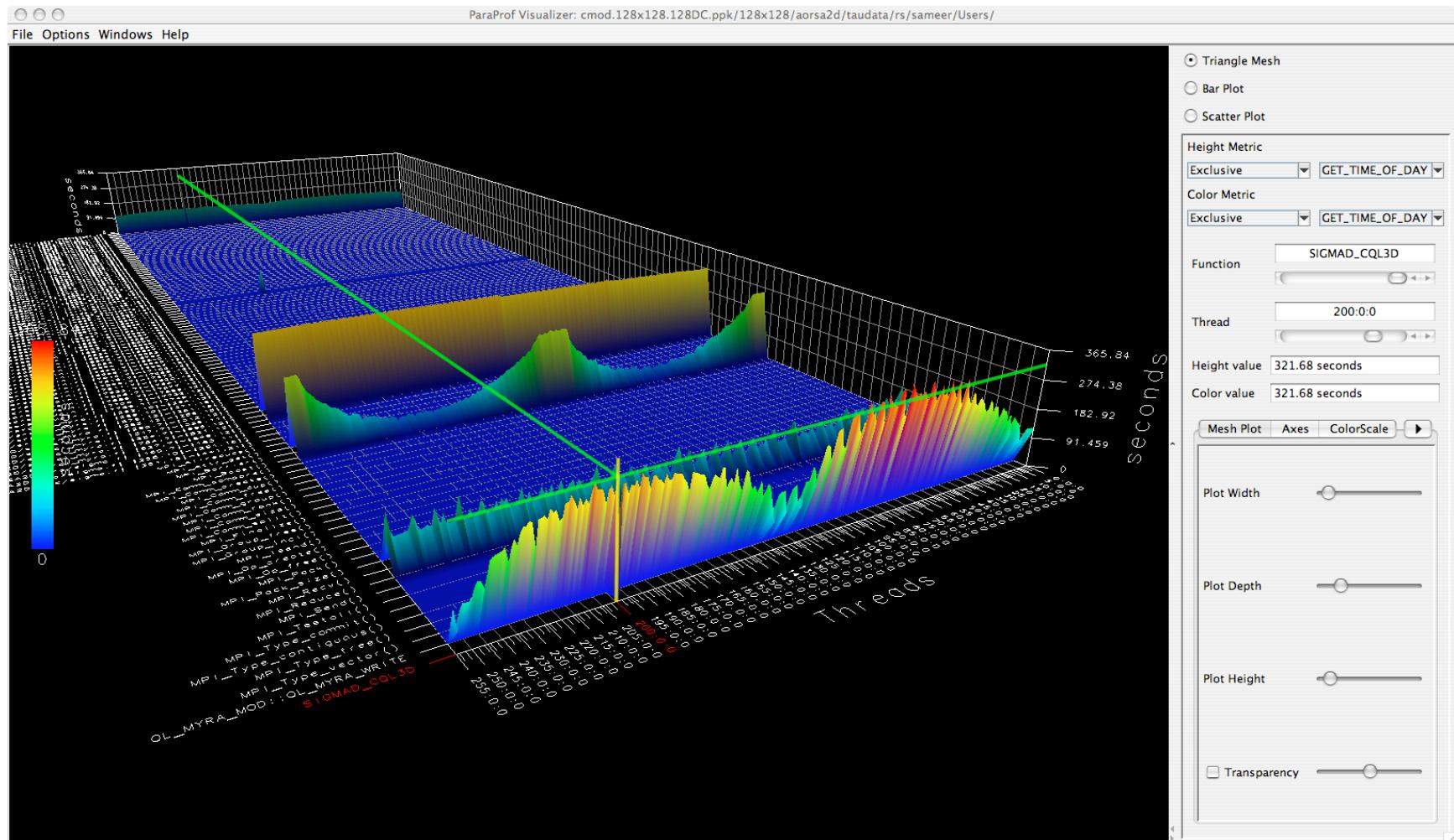
VI-HPS



low mflops in loops?

# Parallel Profile Visualization: ParaProf

VI-HPS



- **Instrumentation:** Adds probes to perform measurements
  - Source code instrumentation using pre-processors and compiler scripts
  - Wrapping external libraries (I/O, MPI, Memory, CUDA, OpenCL, pthread)
  - Rewriting the binary executable
- **Measurement:** Profiling or Tracing using wallclock time or hardware counters
  - Direct instrumentation (Interval events measure exclusive or inclusive duration)
  - Indirect instrumentation (Sampling measures statement level contribution)
  - Throttling and runtime control of low-level events that execute frequently
  - Per-thread storage of performance data
  - Interface with external packages (Scalasca, VampirTrace, Score-P, PAPI)
- **Analysis:** Visualization of profiles and traces
  - 3D visualization of profile data in paraprof, perfexplorer tools
  - Trace conversion & display in external visualizers (Vampir, Jumpshot, ParaVer)

- TAU supports several measurement and thread options
  - Phase profiling, profiling with hardware counters, trace with Score-P...
- Each measurement configuration of TAU corresponds to a unique stub makefile and library that is generated when you configure it
- To instrument source code automatically using PDT
  - Choose an appropriate TAU stub makefile in <arch>/lib directory:  
`% module load tau; export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-mpi-pdt  
% export TAU_OPTIONS=' -optVerbose ...' (see tau_compiler.sh )`  
Use tau\_f90.sh, tau\_cxx.sh or tau\_cc.sh as F90, C++ or C compilers:  
`% mpif90 foo.f90 changes to  
% tau_f90.sh foo.f90`
- Set runtime environment variables, execute application and analyze performance data:
  - `% pprof (for text based profile display)`
  - `% paraprof (for GUI)`

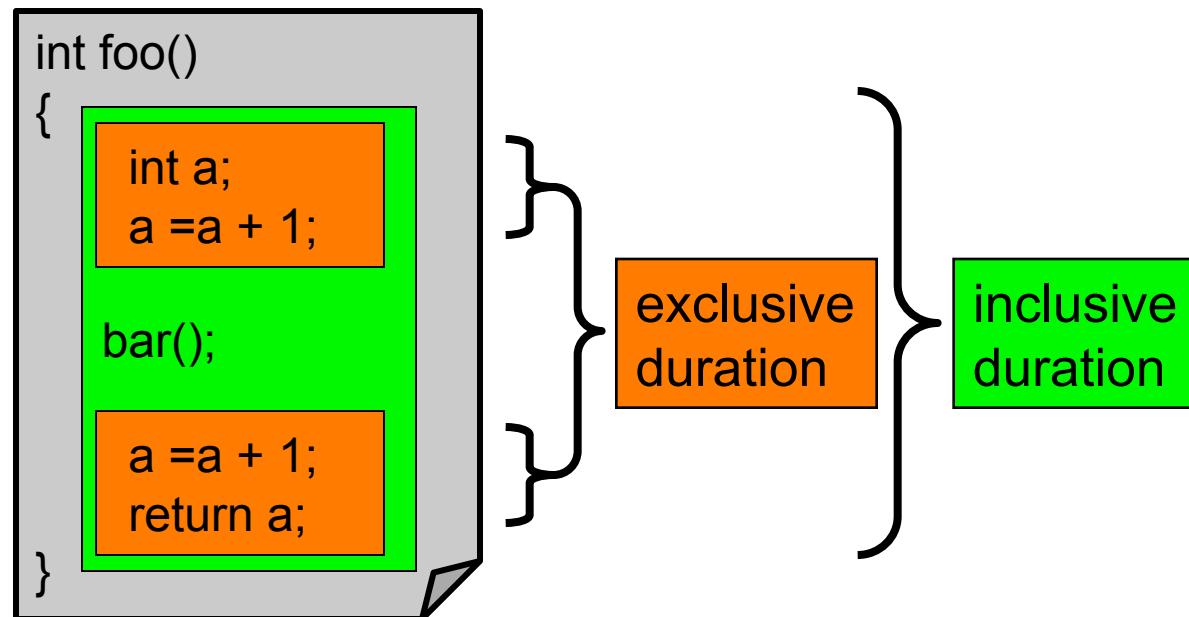
```
% module load tau; cd $TAU_LIB_DIR; ls Makefile.*  
Makefile.tau-icpc-mpi-pdt  
Makefile.tau-icpc-mpi-pdt-openmp-opari  
Makefile.tau-icpc-papi-mpi-pdt  
Makefile.tau-icpc-papi-mpi-pdt-epilog-scalasca-trace  
Makefile.tau-icpc-papi-mpi-pdt-openmp-opari  
Makefile.tau-icpc-papi-mpi-pdt-vampirtrace-trace  
Makefile.tau-phase-icpc-mpi-pdt
```

- For an MPI+F90 application, you may choose **Makefile.tau-icpc-mpi-pdt**
  - Supports MPI instrumentation & PDT for automatic source instrumentation
  - % export TAU\_MAKEFILE=\$TAU/Makefile.tau-icpc-mpi-pdt
  - % tau\_f90.sh matrix.f90 -o matrix
  - % ccc\_mprun -n 8 ./matrix
  - % paraprof

- Supports both direct and indirect performance observation
  - Direct instrumentation of program (system) code (probes)
  - Instrumentation invokes performance measurement
  - Event measurement: performance data, meta-data, context
  - Indirect mode supports sampling based on periodic timer or hardware performance counter overflow based interrupts
- Support for user-defined events
  - **Interval** (Start/Stop) events to measure exclusive & inclusive duration
  - **Atomic events** (Trigger at a single point with data, e.g., heap memory)
    - Measures total, samples, min/max/mean/std. deviation statistics
  - **Context events** (are atomic events with executing context)
    - Measures above statistics for a given calling path

- Event types
  - Interval events (begin/end events)
    - Measures exclusive & inclusive durations between events
    - Metrics monotonically increase
  - Atomic events (trigger with data value)
    - Used to capture performance data state
    - Shows extent of variation of triggered values (min/max/mean)
- Code events
  - Routines, classes, templates
  - Statement-level blocks, loops

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



# Interval Events, Atomic Events in TAU

VI-HPS

NODE 0;CONTEXT 0;THREAD 0:					
%Time	Exclusive msec	Inclusive total msec	#Call	#Subrs	Inclusive Name usec/call
100.0	0.187	1.105	1	44	1105659 int main(int, char **) C
93.2	1.030	1.030	1	0	1030654 MPI_Init()
5.9	0.879	65	40	320	1637 void func(int, int) C
4.6	51	51	40	0	1277 MPI_Barrier()
1.2	13	13	120	0	111 MPI_Recv()
0.8	9	9	1	0	9328 MPI_Finalize()
0.0	0.137	0.137	120	0	1 MPI_Send()
0.0	0.086	0.086	40	0	2 MPI_Bcast()
0.0	0.002	0.002	1	0	2 MPI_Comm_size()
0.0	0.001	0.001	1	0	1 MPI_Comm_rank()

Interval events  
e.g., routines  
(start/stop) show  
**duration**

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

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

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

# Atomic Events, Context Events

VI-HPS

%Time	Exclusive msec	Inclusive total msec	#Call	#Subrs	Inclusive Name usec/call
100.0	0.253	1.106	1	44	1106701 int main(int, char **) C
93.2	1.031	1.031	1	0	1031311 MPI_Init()
6.0	1	66	40	320	1650 void func(int, int) C
5.7	63	63	40	0	1588 MPI_Barrier()
0.8	9	9	1	0	9119 MPI_Finalize()
0.1	1	1	120	0	10 MPI_Recv()
0.0	0.141	0.141	120	0	1 MPI_Send()
0.0	0.085	0.085	40	0	2 MPI_Bcast()
0.0	0.001	0.001	1	0	1 MPI_Comm_size()
0.0	0	0	1	0	0 MPI_Comm_rank()

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

Atomic events

Context events  
=atomic event  
+ executing  
context

% export TAU\_CALLPATH\_DEPTH=1

Controls depth of executing context shown in profiles

% export TAU\_TRACK\_HEAP=1

# Context Events (Default)

VI-HPS

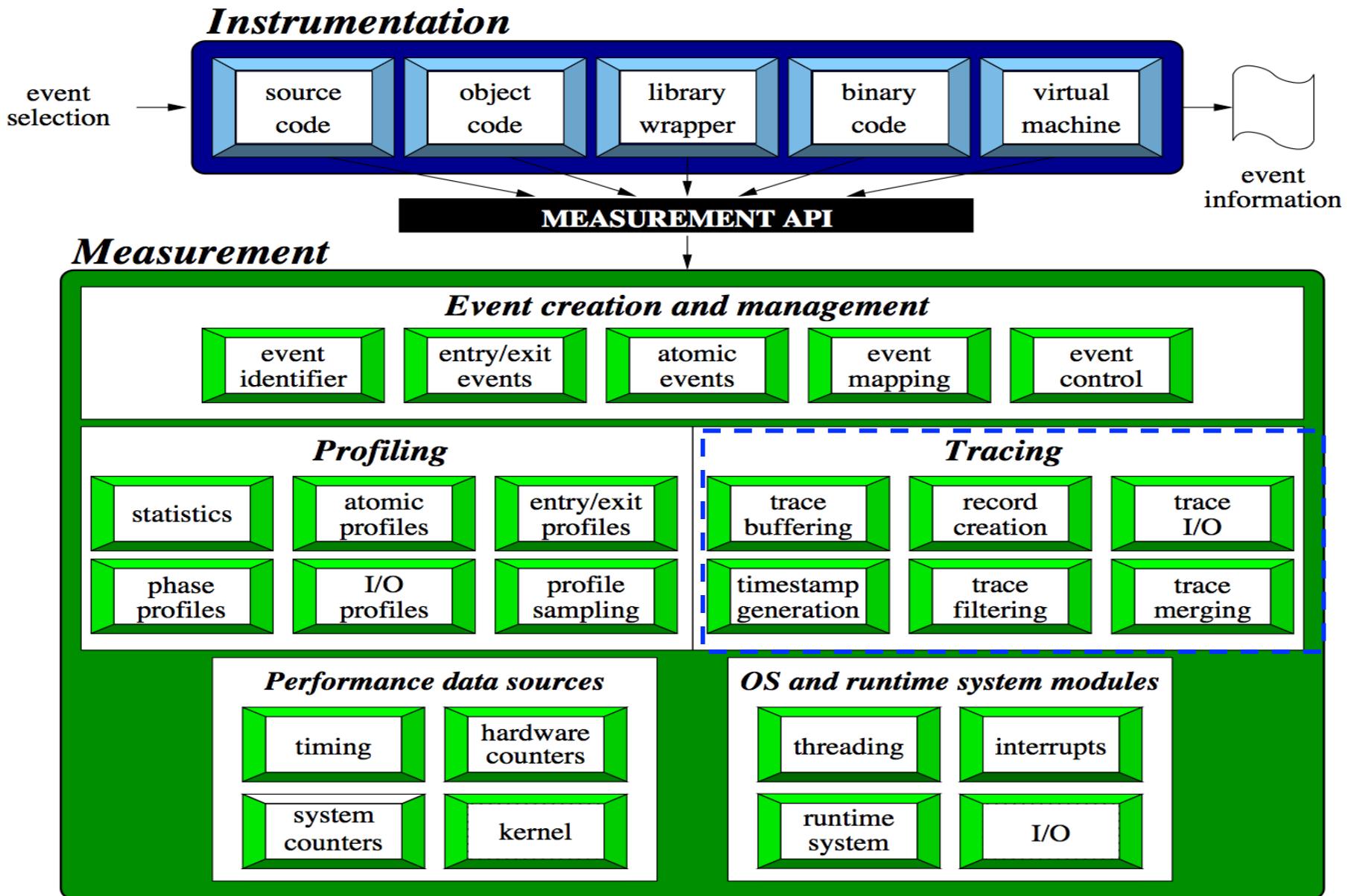
NODE 0:CONTEXT 0;THREAD 0:					
%Time	Exclusive msec	Inclusive total msec	#Call	#Subrs	Inclusive Name usec/call
100.0	0.357	1.114	1	44	1114040 int main(int, char **) C
92.6	1.031	1.031	1	0	1031066 MPI_Init()
6.7	72	74	40	320	1865 void func(int, int) C
0.7	8	8	1	0	8002 MPI_Finalize()
0.1	1	1	120	0	12 MPI_Recv()
0.1	0.608	0.608	40	0	15 MPI_BARRIER()
0.0	0.136	0.136	120	0	1 MPI_Send()
0.0	0.095	0.095	40	0	2 MPI_Bcast()
0.0	0.001	0.001	1	0	1 MPI_Comm_size()
0.0	0	0	1	0	0 MPI_Comm_rank()

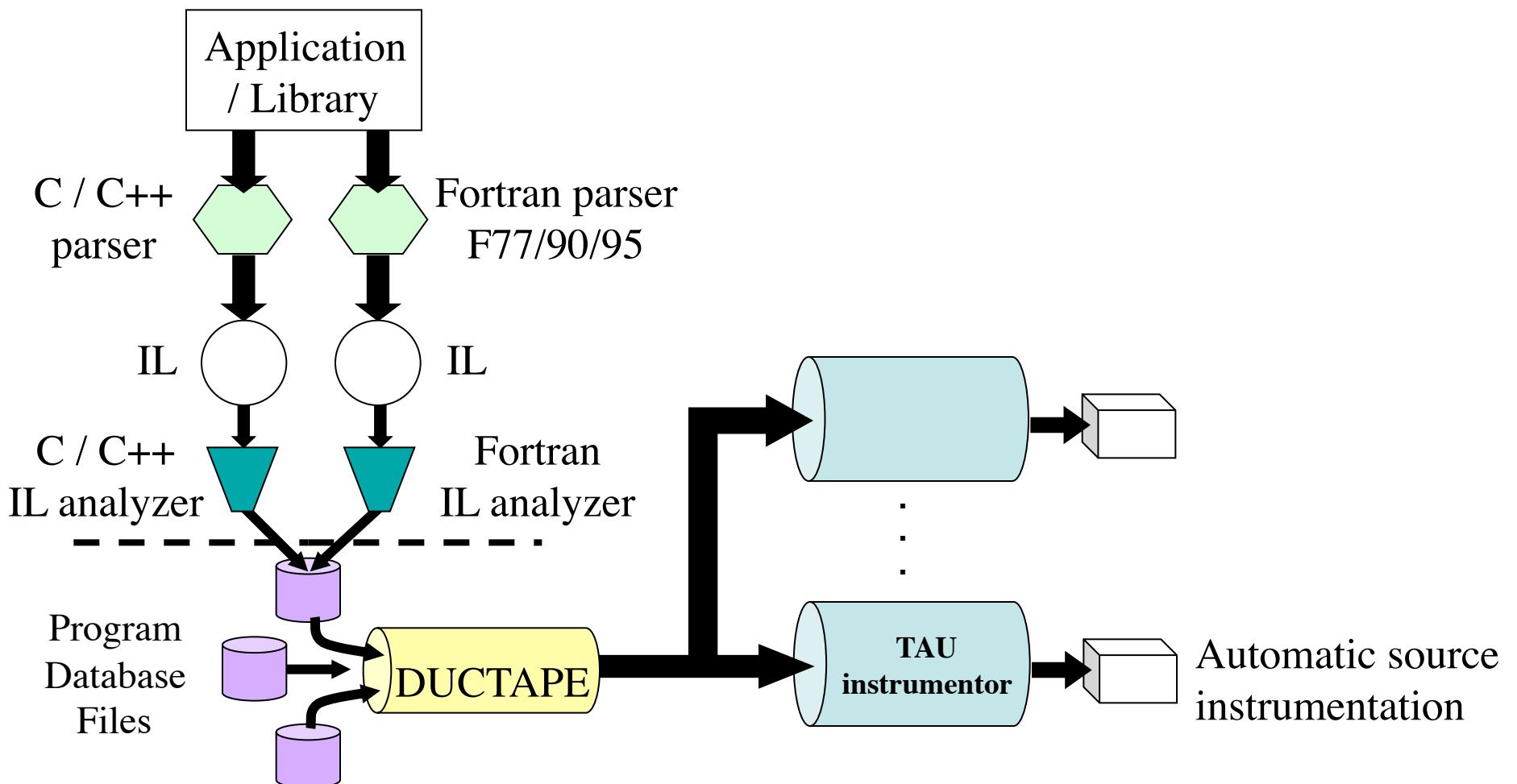
USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0					
NumSamples	MaxValue	MinValue	MeanValue	Std. Dev.	Event Name
365	5.139E+04	44.39	3.091E+04	1.234E+04	Heap Memory Used (KB) : Entry
1	44.39	44.39	44.39	0	Heap Memory Used (KB) : Entry : int main(int, char **) C
1	2068	2068	2068	0	Heap Memory Used (KB)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

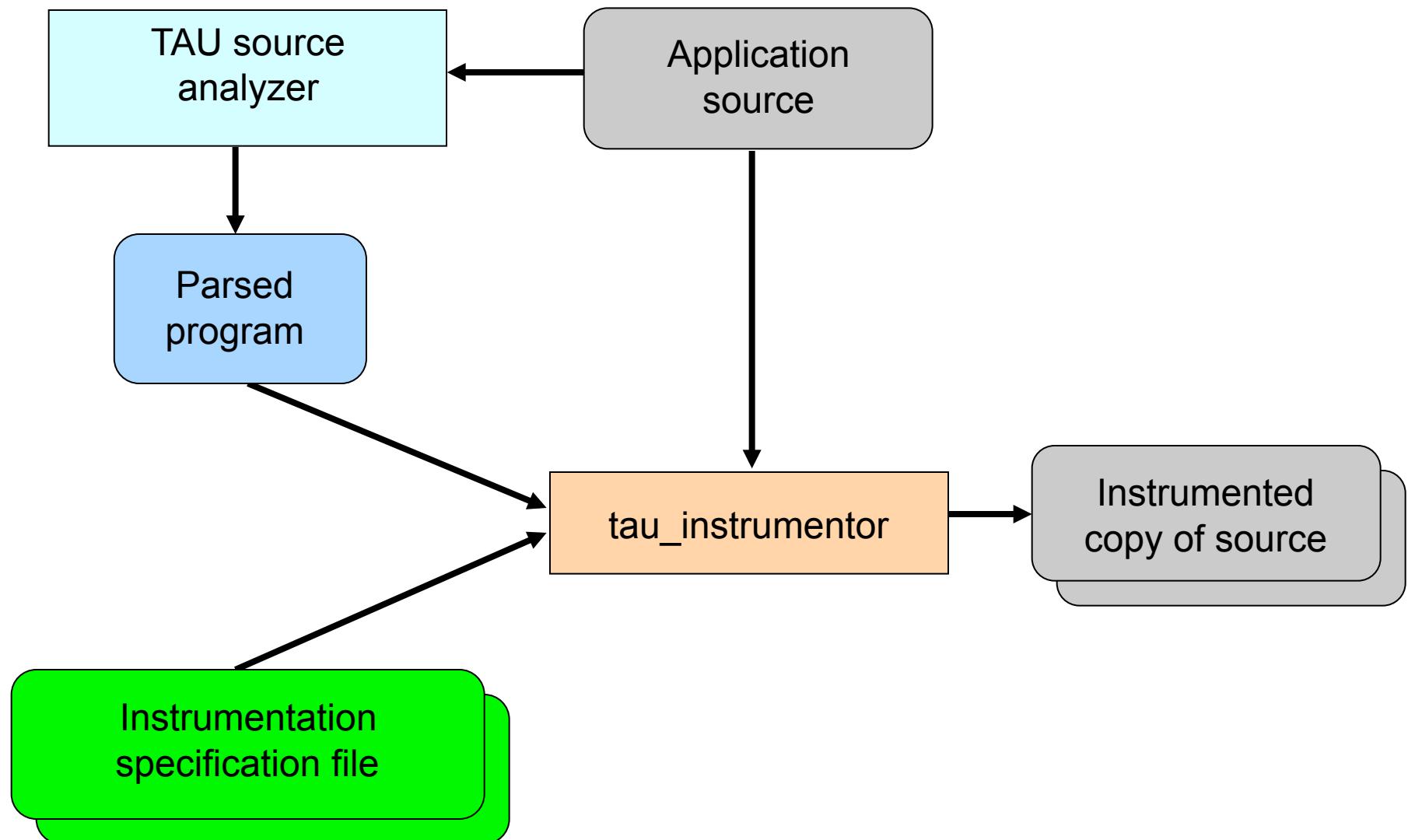
```
% export TAU_CALLPATH_DEPTH=2
% export TAU_TRACK_HEAP=1
```

3.7  
Context event  
=atomic event  
+ executing  
context



- Source Code Instrumentation
  - Manual instrumentation
  - Automatic instrumentation using pre-processor based on static analysis of source code (PDT), creating an instrumented copy
  - Compiler generates instrumented object code
- Library Level Instrumentation
  - Wrapper libraries for standard MPI libraries using PMPI interface
  - Wrapping external libraries where source is not available
- Runtime pre-loading and interception of library calls
- Binary Code instrumentation
  - Rewrite the binary, runtime instrumentation
- Virtual Machine, Interpreter, OS level instrumentation



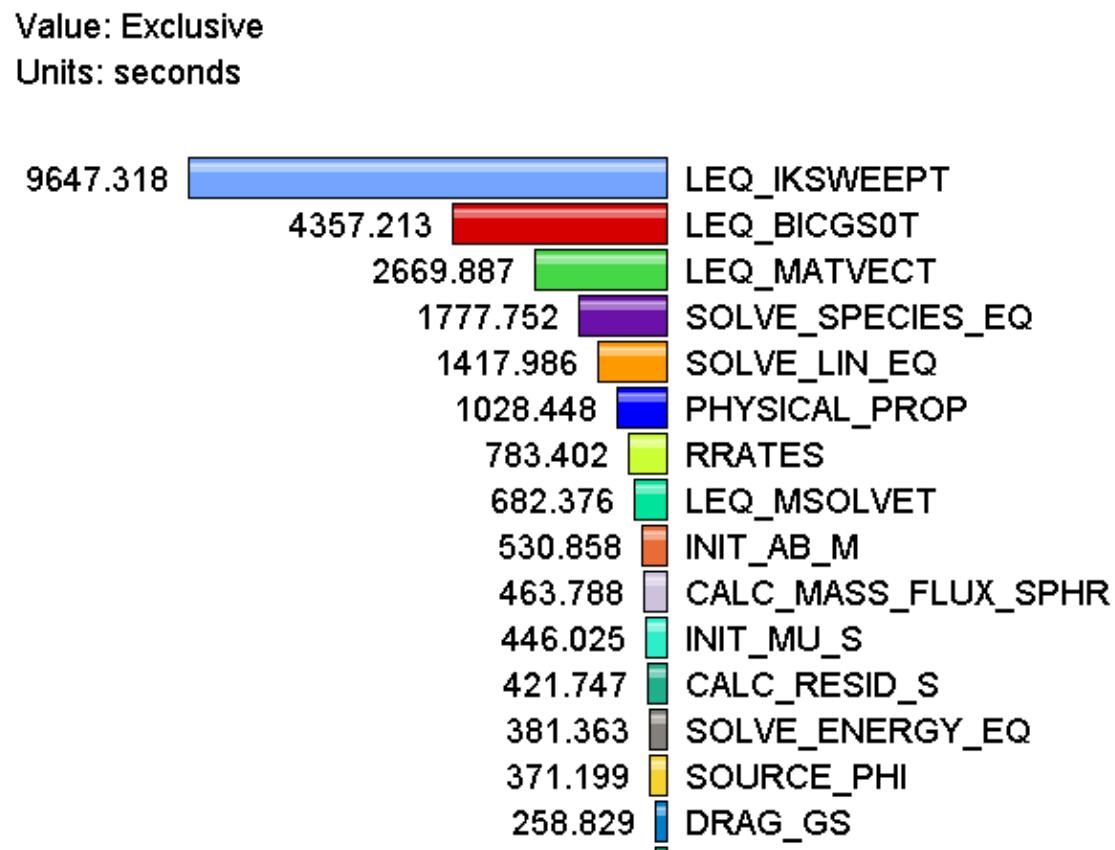


- To instrument source code using PDT
  - Choose an appropriate TAU stub makefile from  
\$TAU\_LIB\_DIR/Makefile.tau\*:  
(typically, *arch*=i386\_linux, x86\_64, craycnl, bgq, cygwin ... and  
*taudir*=/usr/local/packages/tau on LiveDVD)  
  

```
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-mpi-pdt
% make CC=tau_cc.sh CXX=tau_cxx.sh F90=tau_f90.sh
```
- Execute application and analyze performance data:  
  

```
% pprof (for text based profile display)
% paraprof (for GUI)
```

- How much time is spent in each application routine?



## Solution: Generating a flat profile with MPI



```
% module load tau  
% export TAU_MAKEFILE=$TAU_LIB_DIR/Makefile.tau-icpc-mpi-pdt  
% make F90=tau_f90.sh
```

Or

```
% tau_f90.sh matmult.f90  
% ccc_mprun -n 8 ./a.out  
% paraprof
```

To view. To view the data locally on the workstation,

```
% paraprof --pack app.ppk
```

Move the app.ppk file to your desktop.

```
% paraprof app.ppk
```

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

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

## Before

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

app: $(OBJS)
    $(CXX) $(LDFLAGS) $(OBJS) -o $@
    $(LIBS)

.cpp.o:
    $(CXX) $(CXXFLAGS) -c $<
```

## After

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

app: $(OBJS)
    $(CXX) $(LDFLAGS) $(OBJS) -o $@
    $(LIBS)

.cpp.o:
    $(CXX) $(CXXFLAGS) -c $<
```

- Go to tutorial directory

```
% cd tutorial/NPB3.3-MZ-MPI /
```

- Activate the TAU compiler wrappers

```
% vim config/make.def  
      #F77 = mpif77  
      F77 = tau_f90.sh  
% module load tau  
% export TAU_MAKEFILE=$TAU_LIB_DIR/Makefile.tau-  
icpc-mpi-pdt-openmp-opari
```

- Re-compile

```
% make clean; make bt-mz CLASS=W NPROCS=4
```

Reconsider number  
of ranks and buffer  
sizes when running  
the LiveDVD on a  
laptop computer.

```
% ccc_mprun run.msub  
% paraprof &
```

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

- Compilation:

```
% mpif90 -c foo.f90
```

Changes to

```
% gfpars e foo.f90 $(OPT1)  
% tau_instrumentor foo.pdb foo.f90 -o foo.inst.f90 $(OPT2)  
% mpif90 -c foo.inst.f90 -o foo.o $(OPT3)
```

- Linking:

```
% mpif90 foo.o bar.o -o app
```

Changes to

```
% mpif90 foo.o bar.o -o app <taulibs> $(OPT4)
```

- Where options OPT[1-4] default values may be overridden by the user:

**F90 = tau\_f90.sh**

# Compile-Time Environment Variables



- Optional parameters for the TAU\_OPTIONS environment variable:

% tau_compiler.sh	
-optVerbose	Turn on verbose debugging messages
-optComplInst	Use compiler based instrumentation
-optNoComplInst	Do not revert to compiler instrumentation if source instrumentation fails.
-optTrackIO	Wrap POSIX I/O call and calculates vol/bw of I/O operations (Requires TAU to be configured with <code>-iowrapper</code> )
-optKeepFiles	Does not remove intermediate .pdb and .inst.* files
-optPreProcess	Preprocess Fortran sources before instrumentation
-optTauSelectFile=" <i>&lt;file&gt;</i> "	Specify selective instrumentation file for <i>tau_instrumentor</i>
-optTauWrapFile=" <i>&lt;file&gt;</i> "	Specify path to <i>link_options.tau</i> generated by <i>tau_gen_wrapper</i>
-optHeaderInst	Enable Instrumentation of headers
-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) ...

- If your Fortran code uses free format in .f files (fixed is default for .f), you may use:  
`% export TAU_OPTIONS='-optPdtF95Opts="-R free" -optVerbose'`
- To use the compiler based instrumentation instead of PDT (source-based):  
`% export TAU_OPTIONS='-optComplInst -optVerbose'`
- If your Fortran code uses C preprocessor directives (#include, #ifdef, #endif):  
`% export TAU_OPTIONS='-optPreProcess -optVerbose -optDetectMemoryLeaks'`
- To use an instrumentation specification file:  
`% export TAU_OPTIONS='-optTauSelectFile=select.tau -optVerbose -optPreProcess'`  
`% cat select.tau`  
BEGIN\_INSTRUMENT\_SECTION  
loops routine="#"  
# this statement instruments all outer loops in all routines. # is wildcard as well as comment in first column.  
END\_INSTRUMENT\_SECTION

## Runtime Environment Variables in TAU

Environment Variable	Default	Description
TAU_TRACE	0	Setting to 1 turns on tracing
TAU_CALLPATH	0	Setting to 1 turns on callpath profiling
TAU_TRACK_MEMORY_LEAKS	0	Setting to 1 turns on leak detection (for use with tau_exec -memory ./a.out)
TAU_TRACK_HEAP or TAU_TRACK_HEADROOM	0	Setting to 1 turns on tracking heap memory/headroom at routine entry & exit using context events (e.g., Heap at Entry: main=>foo=>bar)
TAU_CALLPATH_DEPTH	2	Specifies depth of callpath. Setting to 0 generates no callpath or routine information, setting to 1 generates flat profile and context events have just parent information (e.g., Heap Entry: foo)
TAU_TRACK_IO_PARAMS	0	Setting to 1 with –optTrackIO or tau_exec –io captures arguments of I/O calls
TAU_SAMPLING	1	Generates sample based profiles
TAU_COMM_MATRIX	0	Setting to 1 generates communication matrix display using context events
TAU_THROTTLE	1	Setting to 0 turns off throttling. Enabled by default to remove instrumentation in lightweight routines that are called frequently
TAU_THROTTLE_NUMCALLS	100000	Specifies the number of calls before testing for throttling
TAU_THROTTLE_PERCALL	10	Specifies value in microseconds. Throttle a routine if it is called over 100000 times and takes less than 10 usec of inclusive time per call
TAU_COMPENSATE	0	Setting to 1 enables runtime compensation of instrumentation overhead
TAU_PROFILE_FORMAT	Profile	Setting to “merged” generates a single file. “snapshot” generates xml format
TAU_METRICS	TIME	Setting to a comma separated list generates other metrics. (e.g., TIME:P_VIRTUAL_TIME:PAPI_FP_INS:PAPI_NATIVE_<event>\:<subevent>)

# Usage Scenarios: Loop Level Instrumentation

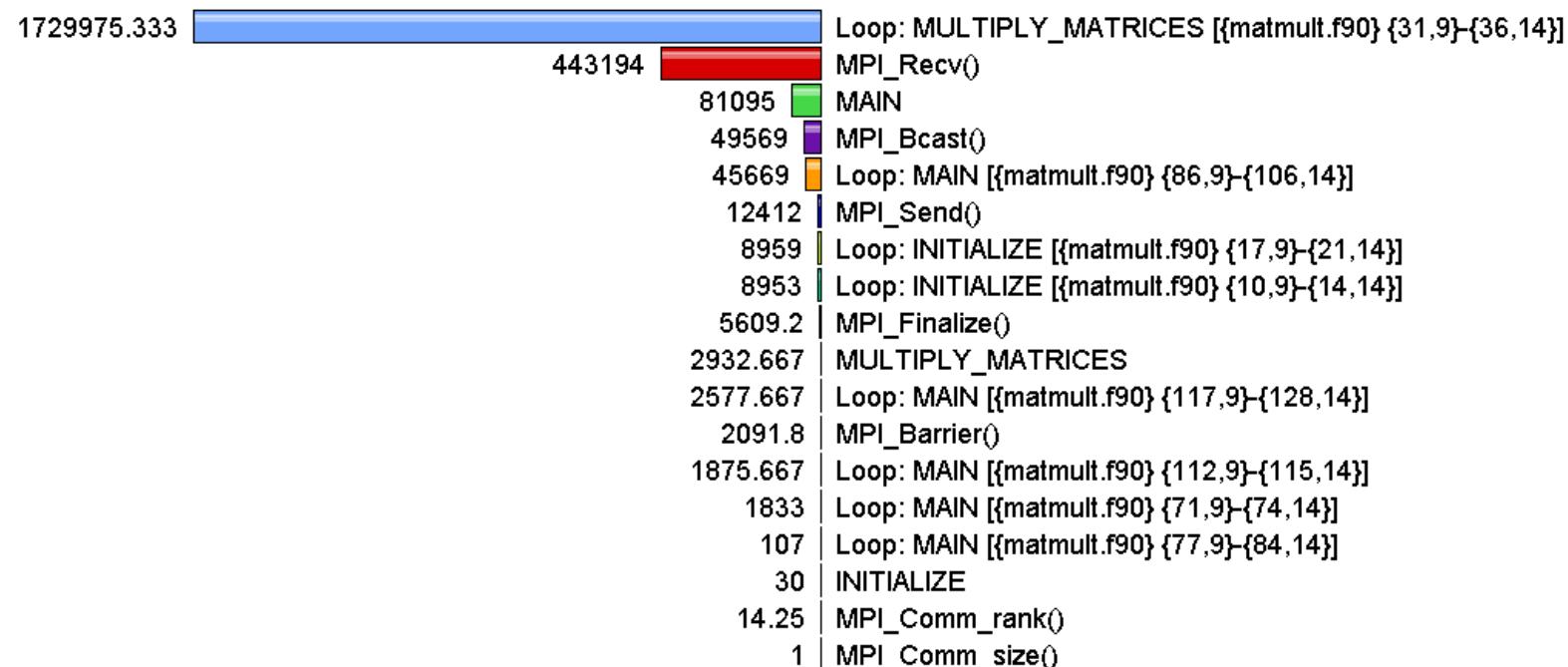
VI-HPS

- Goal: What loops account for the most time? How much?
- Flat profile with wallclock time with loop instrumentation:

Metric: GET\_TIME\_OF\_DAY

Value: Exclusive

Units: microseconds



## Solution: Generating a loop level profile

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

% module load tau
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% ccc_mprun -n 8 ./a.out
% paraprof --pack app.ppk
Move the app.ppk file to your desktop.

% paraprof app.ppk
```

# Computing Floating Point Instructions Executed Per Second in Loops

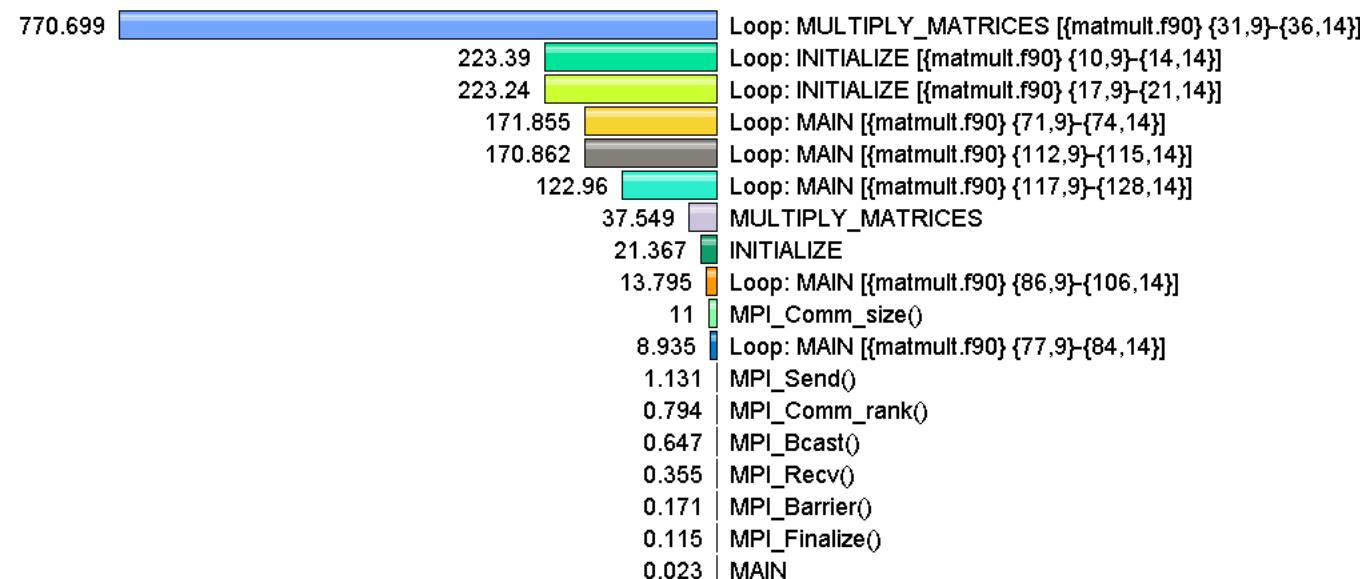
VI-HPS

- Goal: What execution rate do my application loops get in mflops?
- Flat profile with PAPI\_FP\_INS and time with loop instrumentation:

Metric: PAPI\_FP\_INS / GET\_TIME\_OF\_DAY

Value: Exclusive

Units: Derived metric shown in microseconds format



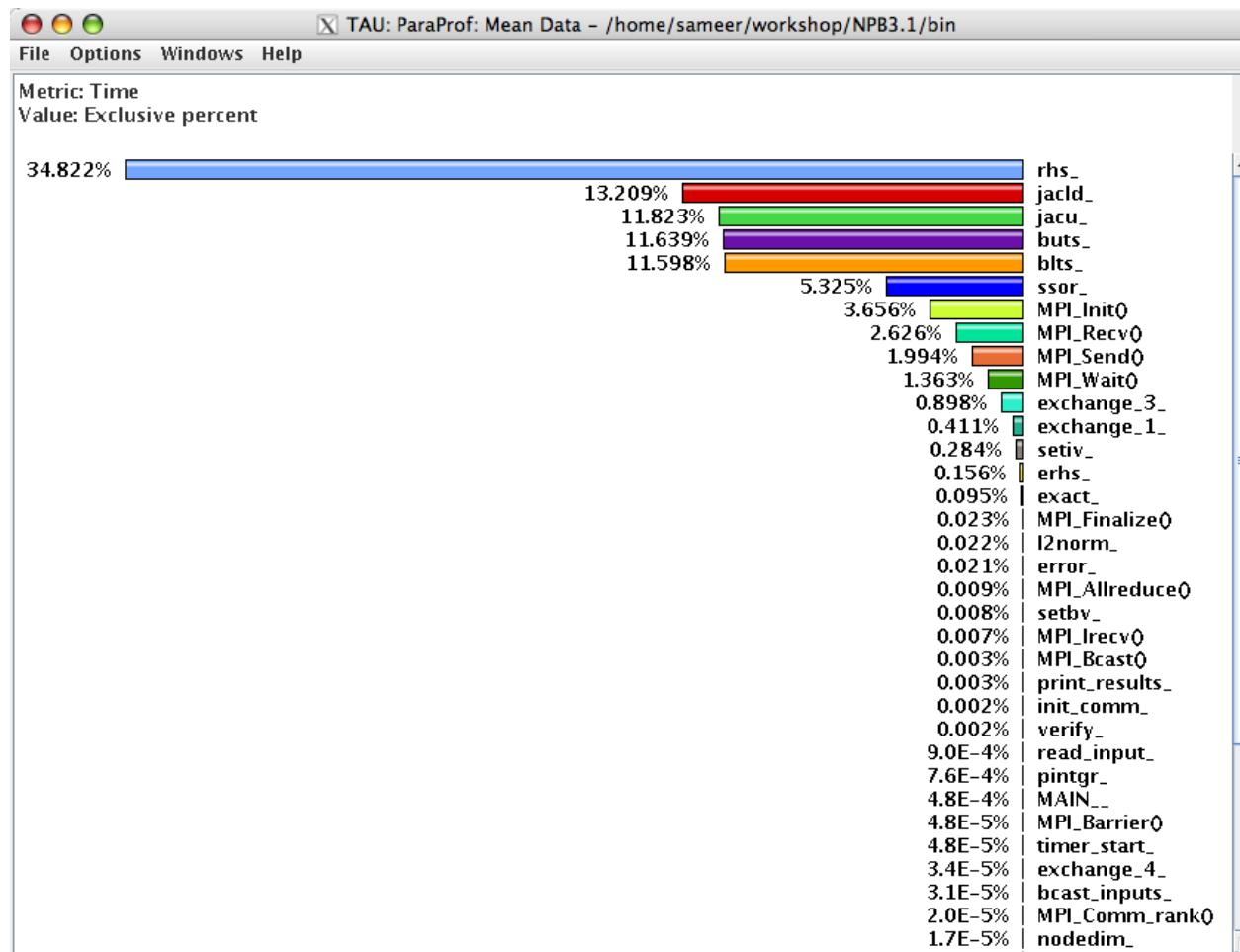
## Generate a PAPI profile with 2 or more counters



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

% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% export TAU_METRICS=TIME:PAPI_FP_INS:PAPI_L1_DCM
% ccc_mprun -n 8 ./a.out
% paraprof --pack app.ppk
Move the app.ppk file to your desktop.
% paraprof app.ppk
Choose Options -> Show Derived Panel -> Click PAPI_FP_INS,
Click "/", Click TIME, Apply, Choose new metric by double clicking.
```

- Use the compiler to automatically emit instrumentation calls in the object code instead of parsing the source code using PDT.



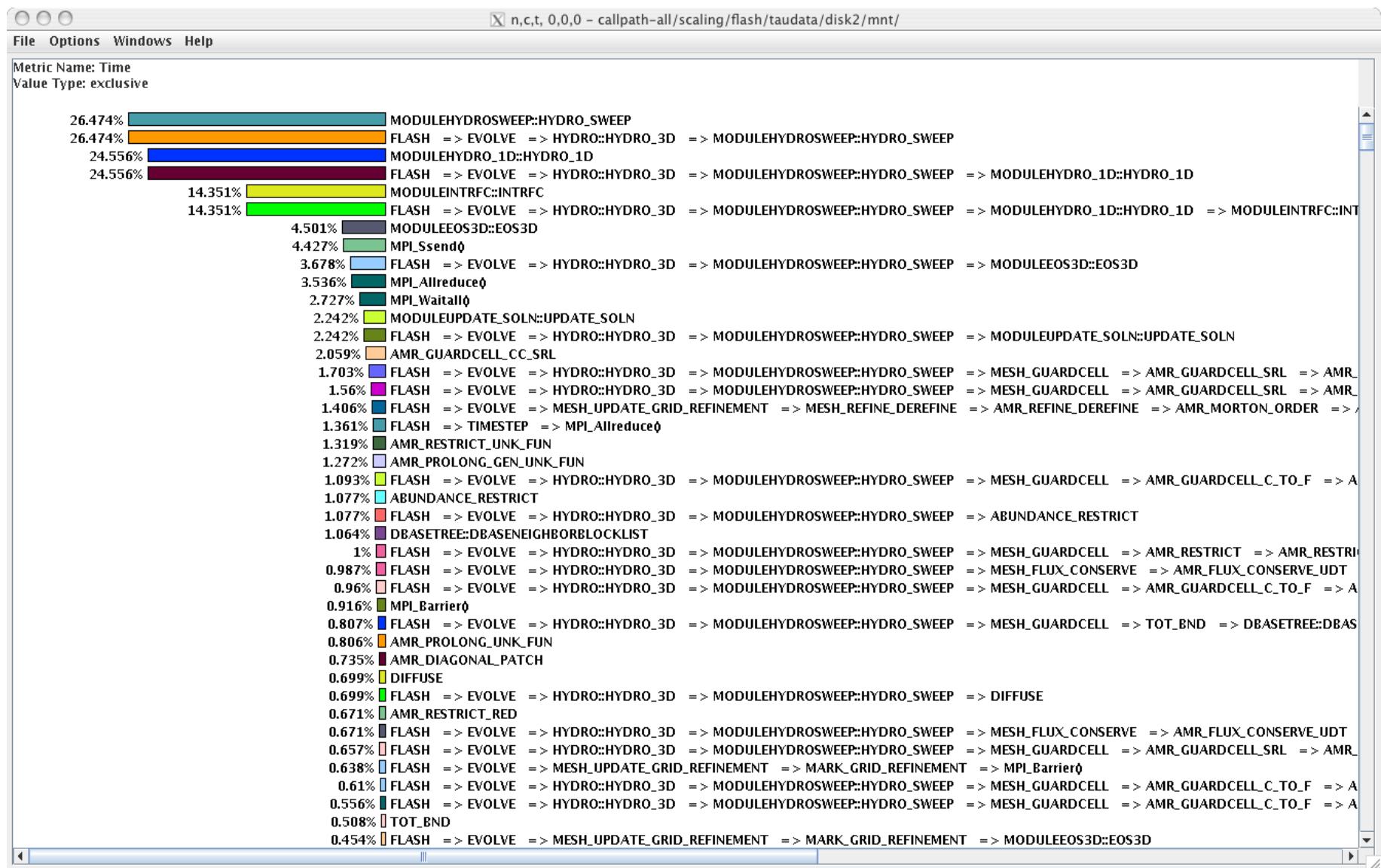
```
% export TAU_MAKEFILE=$TAU_LIB_DIR/Makefile.tau-icpc-mpi-pdt  
% export TAU_OPTIONS='-optCompInst -optQuiet'  
  
% make CC=tau_cc.sh CXX=tau_cxx.sh F90=tau_f90.sh
```

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

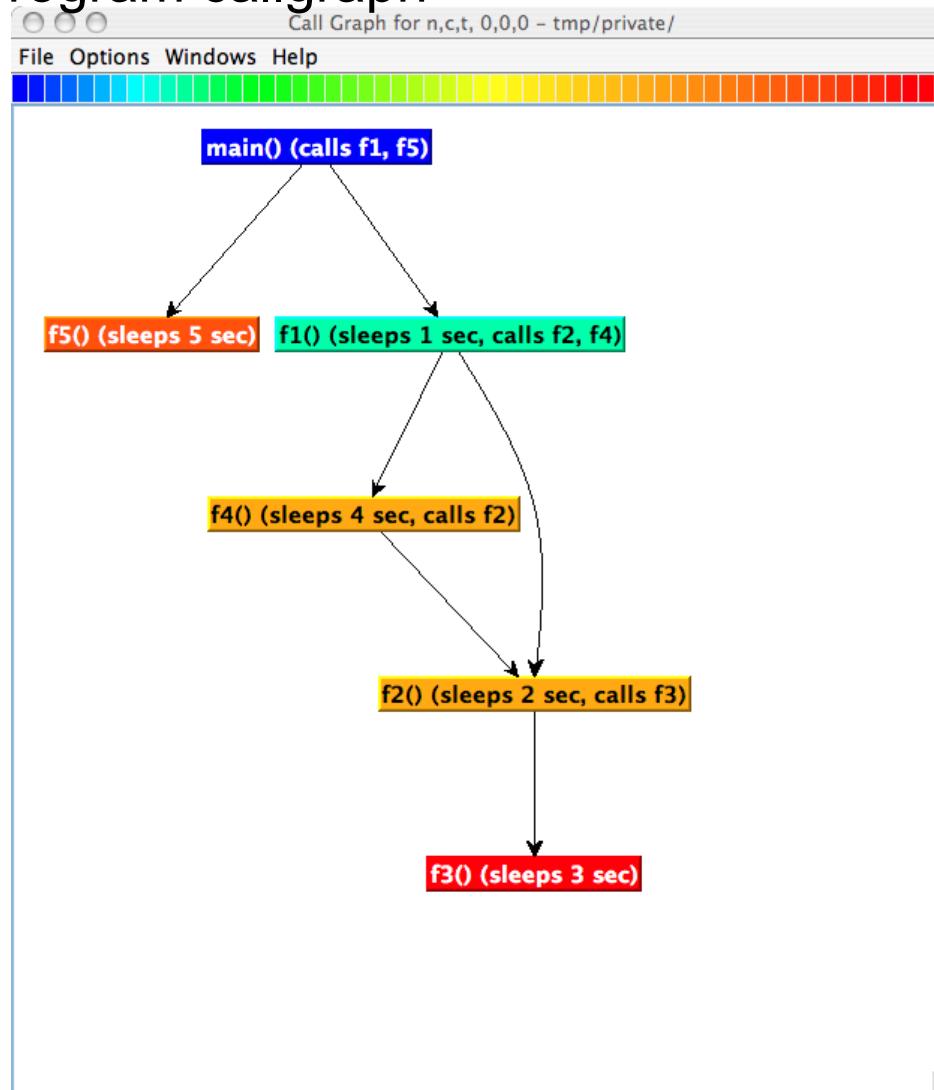
```
% make CC=taucc CXX=taucxx F90=tauf90  
% ccc_mprun -n 8 ./a.out  
% paraprof --pack app.ppk  
  Move the app.ppk file to your desktop.  
% paraprof app.ppk
```

# Generate a Callpath Profile

VI-HPS



- Generates program callgraph



## Generate a Callpath Profile

```
% export TAU_MAKEFILE=$TAU_LIB_DIR/Makefile.tau-icpc-mpi-pdt
% export PATH=<taudir>/<arch>/bin:$PATH
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)

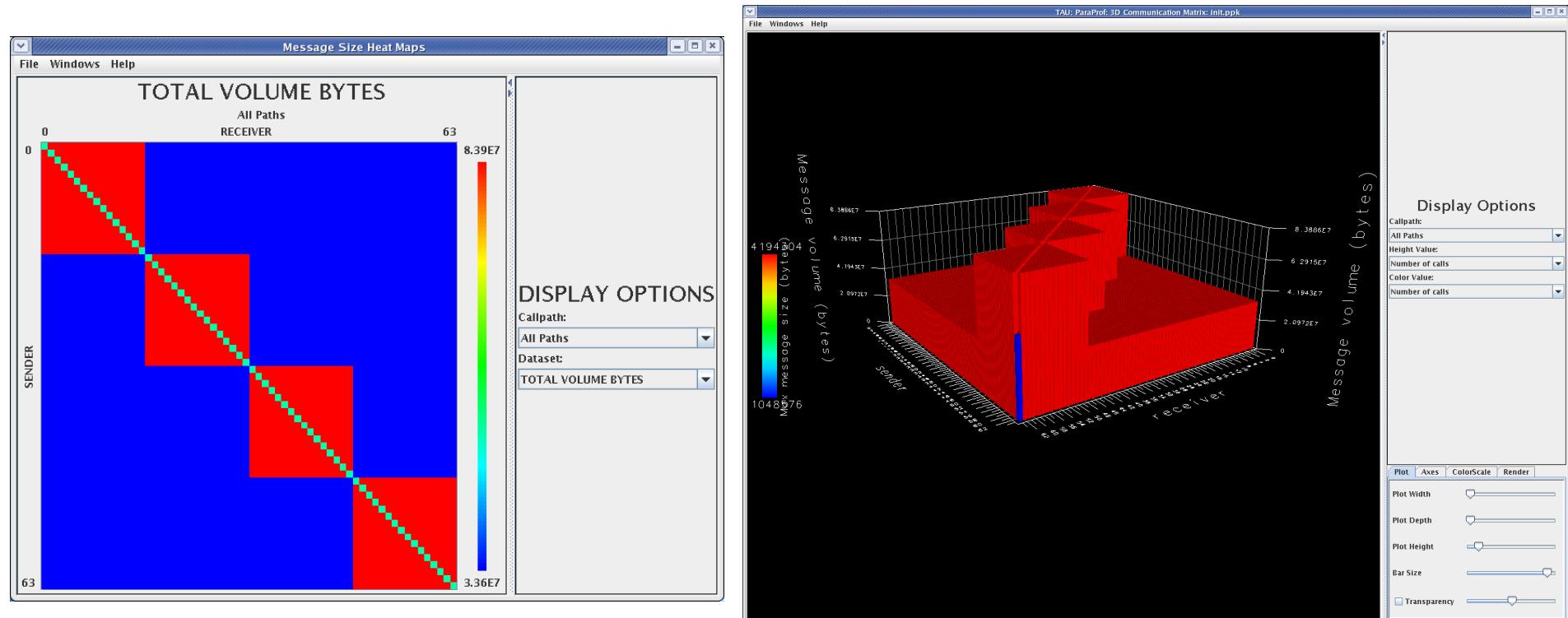
% export TAU_CALLPATH=1
% export TAU_CALLPATH_DEPTH=100
(truncates all calling paths to a specified depth)

% ccc_mprun -n 8 ./a.out
% paraprof --pack app.ppk
Move the app.ppk file to your desktop.
% paraprof app.ppk
(Windows -> Thread -> Call Graph)
```

# Communication Matrix Display

VI-HPS

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



## Evaluate Scalability using PerfExplorer Charts



```
% export TAU_MAKEFILE=$TAU_LIB_DIR/Makefile.tau-icpc-mpi-pdt
% export PATH=<taudir>/<arch>/bin:$PATH
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% export TAU_COMM_MATRIX=1

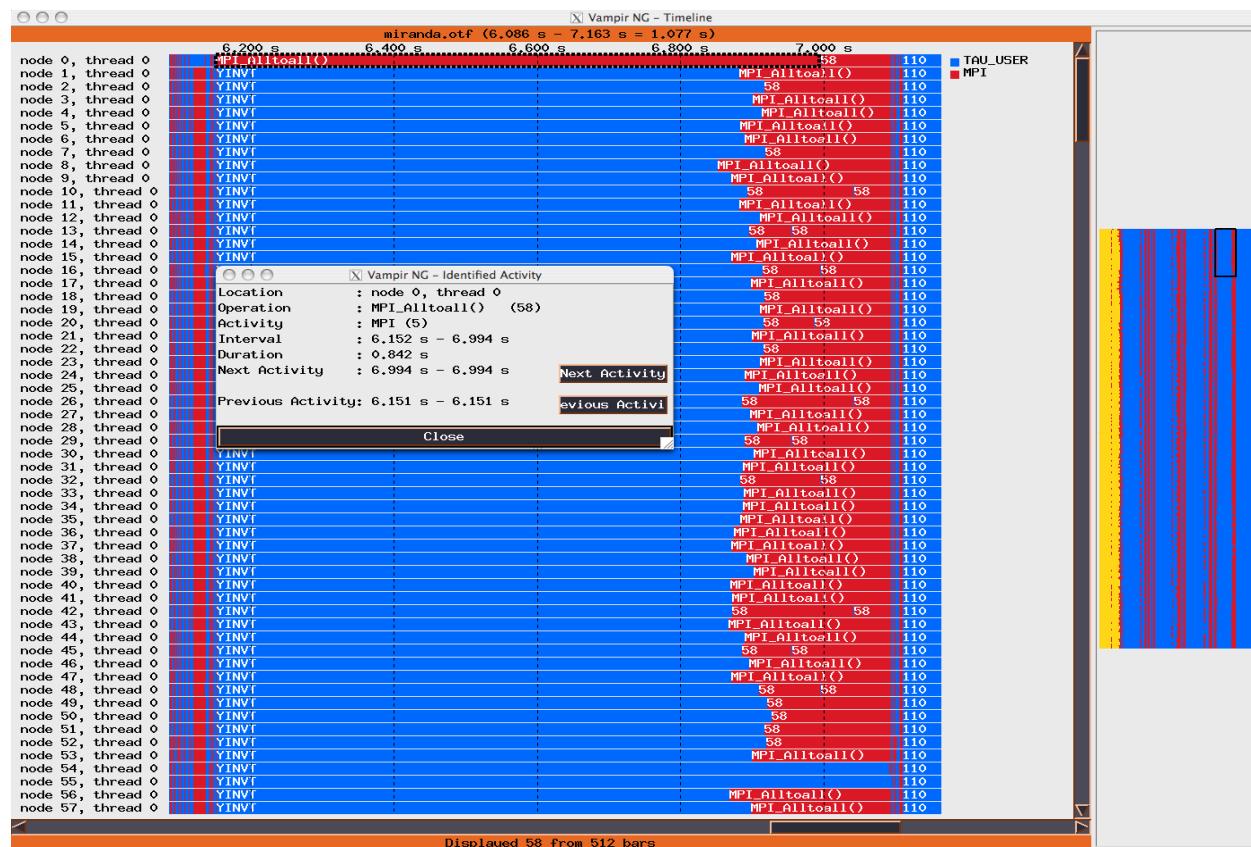
% ccc_mprun -n 8 ./a.out

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

# Generating a Trace File

VI-HPS

- Goal: Identify the temporal aspect of performance. What happens in my code at a given time? When?
  - Event trace visualized in Vampir/Jumpshot/Paraver



## Generate OTF traces using VampirTrace



```
% export TAU_MAKEFILE=$TAU_LIB_DIR/Makefile.tau-icpc-papi-mpi-pdt-
    vampirtrace-trace
% export PATH=<taudir>/<arch>/bin:$PATH
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)

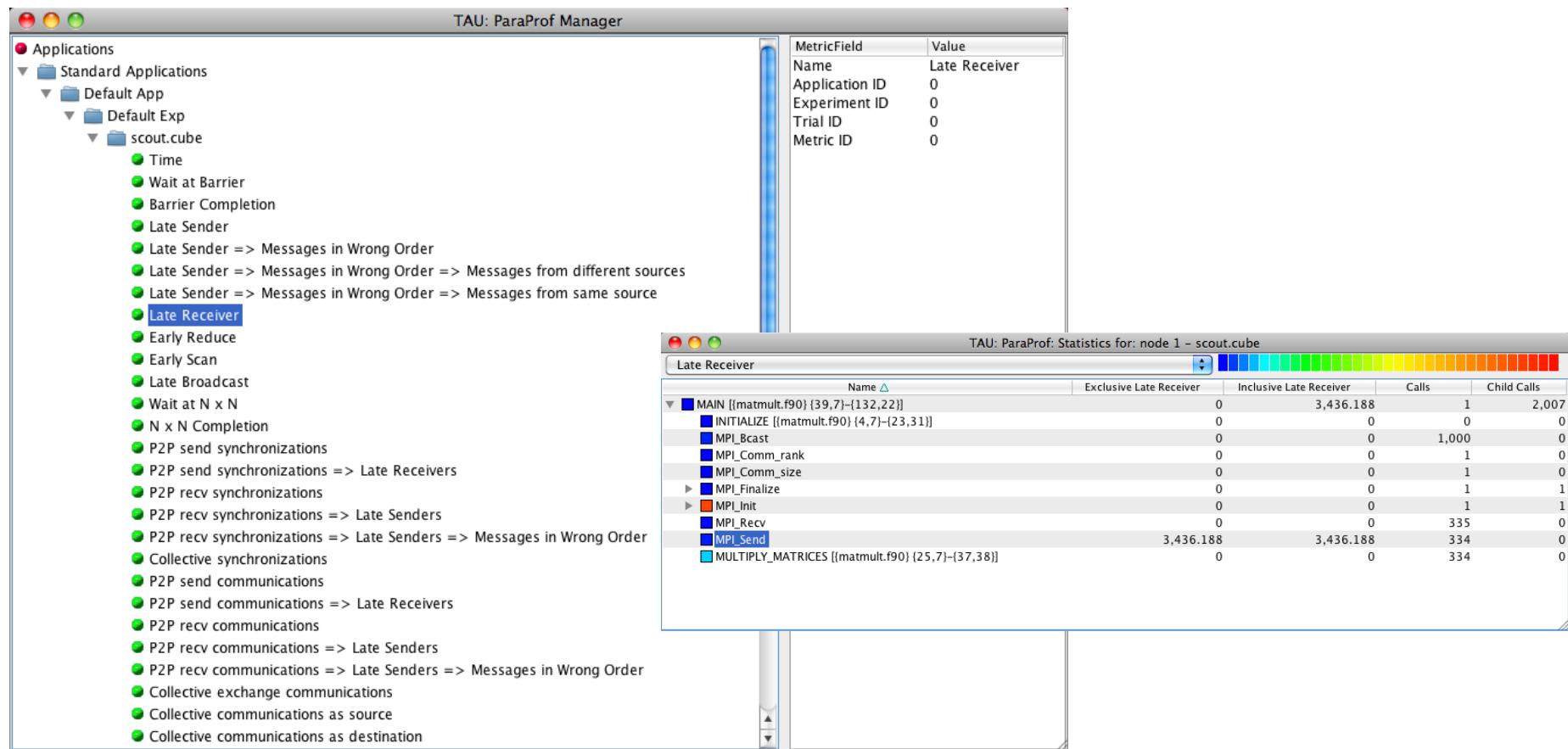
% ccc_mprun -n 8 ./a.out

% module load vampir
% vampir *.otf &
```

# Using TAU with Scalasca

VI-HPS

- Goal: Identify bottlenecks in the code using Scalasca



- Go to tutorial directory

```
% cd tutorial/NPB3.3-MZ-MPI /
```

- Activate the TAU compiler wrappers

```
% vim config/make.def

          #F77 = mpif77
          F77 = tau_f90.sh
% module load tau
% export TAU_MAKEFILE=$TAU_LIB_DIR/
Makefile.tau-icpc-papi-mpi-pdt-epilog-scalasca-trace
% make bt_mz NPROCS=4 CLASS=W
Run using:
scan -t ccc_mprun -n 4 ./bt_mz...
% cd epik_*; paraprof scout.cube &
```

- Pre-processor based substitution by re-defining a call (e.g., `read`)
  - Tool defined header file with same name `<unistd.h>` takes precedence
  - Header redefines a routine as a different routine using macros
  - Substitution: `read()` substituted by preprocessor as `tau_read()` at callsite
- Preloading a library at runtime
  - Library preloaded (`LD_PRELOAD` env var in Linux) in the address space of executing application intercepts calls from a given library
  - Tool's wrapper library defines `read()`, gets address of global `read()` symbol (`dlsym`), internally calls timing calls around call to global `read`
- Linker based substitution
  - Wrapper library defines `__wrap_read` which calls `__real_read` and linker is passed `-Wl,-wrap,read` to substitute all references to `read` from application's object code with the `__wrap_read` defined by the tool

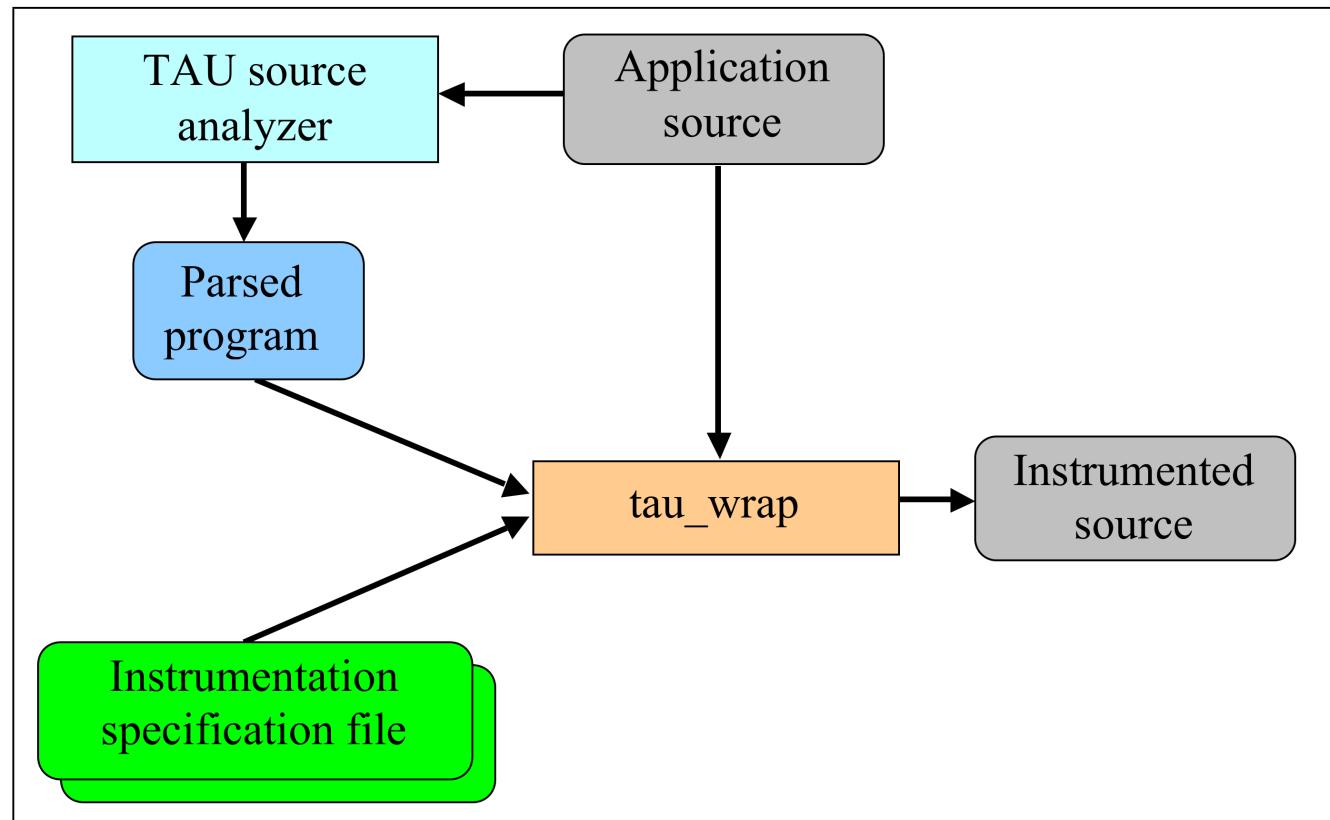
- Pre-processor based substitution by re-defining a call
  - Compiler replaces `read()` with `tau_read()` in the body of the source code
- Advantages:
  - Simple to instrument
    - Preprocessor based replacement
    - A header file redefines the calls
    - No special linker or runtime flags required
- Disadvantages
  - Only works for C & C++ for replacing calls in the body of the code.
  - Incomplete instrumentation: fails to capture calls in uninstrumented libraries (e.g., `lib hdf5.a`)

- Linker based substitution
  - Wrapper library defines `__wrap_read` which calls `__real_read` and linker is passed `-Wl,-wrap, read`
- Advantages
  - Tool can intercept all references to a given call
  - Works with static as well as dynamic executables
  - No need to recompile the application source code, just re-link the application objects and libraries with the tool wrapper library
- Disadvantages
  - Wrapping an entire library can lengthen the linker command line with multiple `-Wl,-wrap,<func>` arguments. It is better to store these arguments in a file and pass the file to the linker
  - Approach does not work with un-instrumented binaries

- Automates creation of wrapper libraries using TAU
- Input:
  - header file (foo.h)
  - library to be wrapped (/path/to/libfoo.a)
  - technique for wrapping
    - Preprocessor based redefinition (-d)
    - Runtime preloading (-r)
    - Linker based substitution (-w: default)
  - Optional selective instrumentation file (-f select)
    - Exclude list of routines, or
    - Include list of routines
- Output:
  - wrapper library
  - optional *link\_options.tau* file (-w), pass –optTauWrapFile=<file> in TAU\_OPTIONS environment variable

- *tau\_gen\_wrapper* shell script:
  - parses source of header file using static analysis tool Program Database Toolkit (PDT)
  - Invokes *tau\_wrap*, a tool that generates
    - instrumented wrapper code,
    - an optional *link\_options.tau* file (for linker-based substitution, -w)
    - Makefile for compiling the wrapper interposition library
  - Builds the wrapper library using make
- Use TAU\_OPTIONS environment variable to pass location of *link\_options.tau* file using

```
% export TAU_OPTIONS='--optTauWrapFile=<path/to/  
link_options.tau> -optVerbose'
```
- Use *tau\_exec --loadlib=<wrapperlib.so>* to pass location of wrapper library for preloading based substitution



# HDF5 Library Wrapping

VI-HPS

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

Usage : tau\_gen\_wrapper <header> <library> [-r|-d|-w (default)] [-g groupname] [-i headerfile] [-c|-c++|-fortran] [-f <instr\_spec\_file> ]

- instruments using runtime preloading (-r), or -Wl,-wrap linker (-w), redirection of header file to redefine the wrapped routine (-d)
- instrumentation specification file (select.tau)
- group (hdf5)
- tau\_exec loads libhdf5\_wrap.so shared library using –loadlib=<libwrap\_pkg.so>
- creates the wrapper/ directory

NODE 0;CONTEXT 0;THREAD 0:

%Time	Exclusive msec	Inclusive total msec	#Call	#Subrs	Inclusive Name usec/call
100.0	0.057	1	1	13	1236 .TAU Application
70.8	0.875	0.875	1	0	875 hid_t H5Fcreate()
9.7	0.12	0.12	1	0	120 herr_t H5Fcclose()
6.0	0.074	0.074	1	0	74 hid_t H5Dcreate()
3.1	0.038	0.038	1	0	38 herr_t H5Dwrite()
2.6	0.032	0.032	1	0	32 herr_t H5Dclose()
2.1	0.026	0.026	1	0	26 herr_t H5check_version()
0.6	0.008	0.008	1	0	8 hid_t H5Screate_simple()
0.2	0.002	0.002	1	0	2 herr_t H5Tset_order()
0.2	0.002	0.002	1	0	2 hid_t H5Tcopy()
0.1	0.001	0.001	1	0	1 herr_t H5Sclose()
0.1	0.001	0.001	2	0	0 herr_t H5open()
0.0	0	0	1	0	0 herr_t H5Tclose()

- Setting environment variable `TAU_OPTIONS=-optTrackIO` links in TAU's wrapper interposition library using linker-based substitution
- Instrumented application generates bandwidth, volume data
- Workflow:
  - `% export TAU_OPTIONS='-optTrackIO -optVerbose'`
  - `% export TAU_MAKEFILE=/path/to/tau/x86_64/lib/Makefile.tau-icpc-mpi-pdt`
  - `% make CC=tau_cc.sh CXX=tau_cxx.sh F90=tau_f90.sh`
  - `% ccc_mprun -n 8 ./a.out`
  - `% paraprof`
- Get additional data regarding individual arguments by setting environment variable `TAU_TRACK_IO_PARAMS=1` prior to running

- Preloading a library at runtime
  - Tool defines `read()`, gets address of global `read()` symbol (`dlsym`), internally calls timing calls around call to global `read`
  - *tau\_exec* tool uses this mechanism to intercept library calls
- Advantages
  - No need to re-compile or re-link the application source code
  - Drop-in replacement library implemented using `LD_PRELOAD` environment variable under Linux, Cray CNL, IBM BG/P CNK, Solaris...
- Disadvantages
  - Only works with dynamic executables. Default compilation mode under Cray XE6 and IBM BG/P is to use static executables
  - Not all operating systems support preloading of dynamic shared objects (DSOs)

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

# Preloading a Specific TAU Measurement Library

VI-HPS

```
% ./configure -pdt=<dir> -mpi -papi=<dir>; make install
```

Creates in \$TAU\_LIB\_DIR:

```
Makefile.tau-papi-mpi-pdt  
shared-papi-mpi-pdt/libTAU.so
```

```
% ./configure -pdt=<dir> -mpi; make install creates
```

```
Makefile.tau-icpc-mpi-pdt  
shared-mpi-pdt/libTAU.so
```

To explicitly choose preloading of shared-<options>/libTAU.so change:

```
% ccc_mprun -n 8 ./a.out      to
```

```
% ccc_mprun -n 8 tau_exec -T <comma_separated_options> ./a.out
```

```
% ccc_mprun -n 8 tau_exec -T papi,mpi,pdt ./a.out
```

Preloads <taudir>/<arch>/shared-papi-mpi-pdt/libTAU.so

```
% ccc_mprun -n 8 tau_exec -T papi ./a.out
```

Preloads <taudir>/<arch>/shared-papi-mpi-pdt/libTAU.so by matching.

```
% ccc_mprun -n 8 tau_exec -T papi,mpi,pdt -s ./a.out
```

Does not execute the program. Just displays the library that it will preload if executed without the **-s** option.

**NOTE:** -mpi configuration is selected by default. Use -T serial for

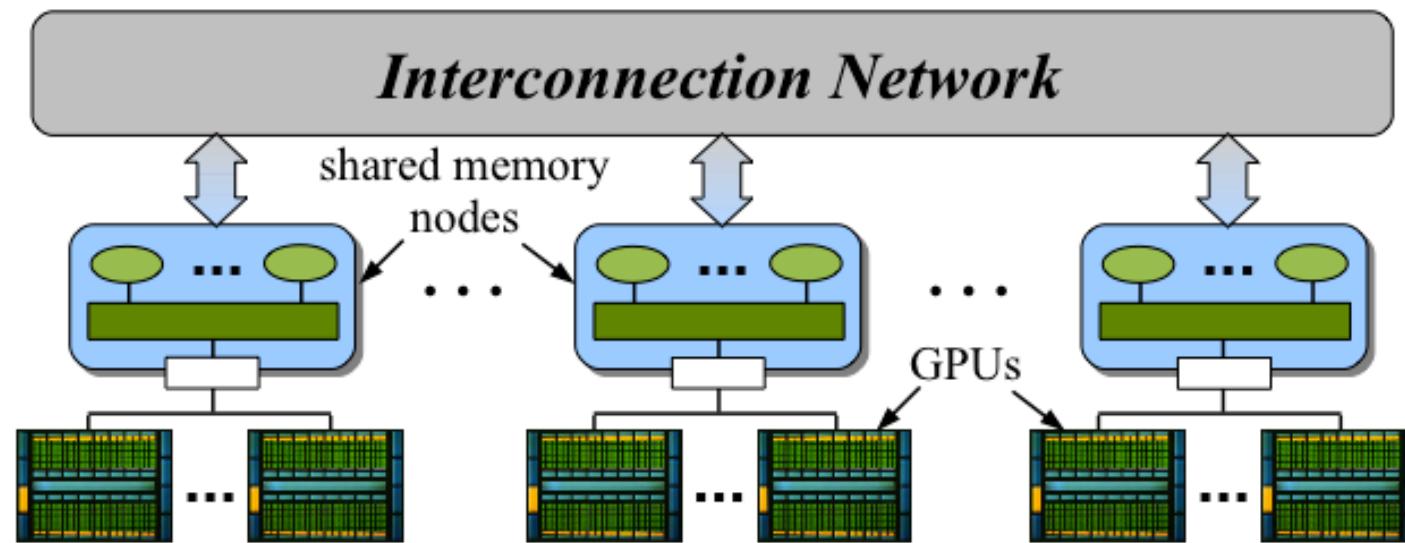
Sequential programs

U. Versailles

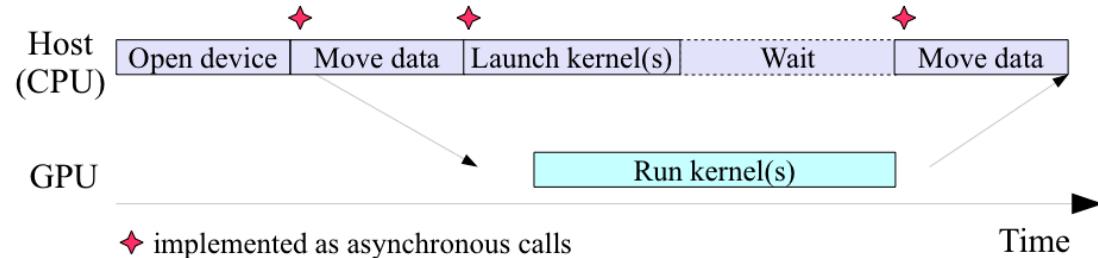
- Uninstrumented execution
  - % ccc\_mprun -n 8 ./a.out
- Track MPI performance
  - % ccc\_mprun -n 8 **tau\_exec** ./a.out
- Track POSIX I/O and MPI performance (MPI enabled by default)
  - % ccc\_mprun -n 8 **tau\_exec** –io ./a.out
- Track memory operations
  - % setenv TAU\_TRACK\_MEMORY\_LEAKS 1
  - % ccc\_mprun -n 8 **tau\_exec** –memory ./a.out
- Use event based sampling (compile with –g)
  - % ccc\_mprun -n 8 **tau\_exec** –ebs ./a.out
  - Also –ebs\_source=<PAPI\_COUNTER> –ebs\_period=<overflow\_count>
- Load wrapper interposition library
  - % ccc\_mprun -n 8 **tau\_exec** –loadlib=<path/libwrapper.so> ./a.out
- **Track GPGPU operations**
  - % ccc\_mprun -n 8 **tau\_exec** –cuda ./a.out
  - % ccc\_mprun -n 8 **tau\_exec** –opencl ./a.out

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

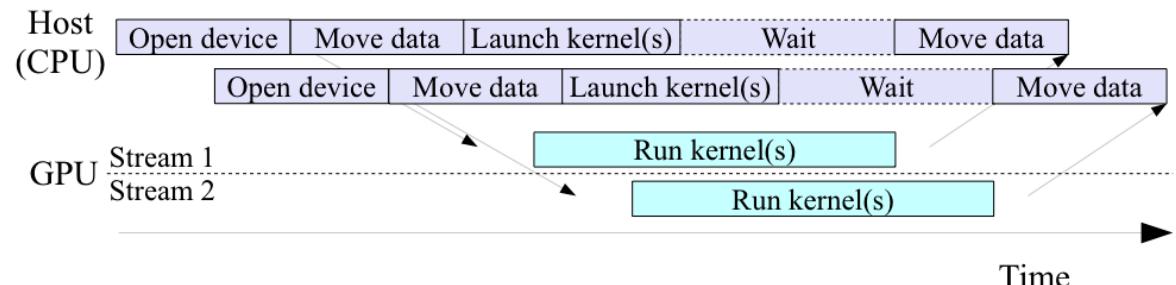
- Multi-CPU, multicore shared memory nodes
- GPU accelerators connected by high-BW I/O
- Cluster interconnection network



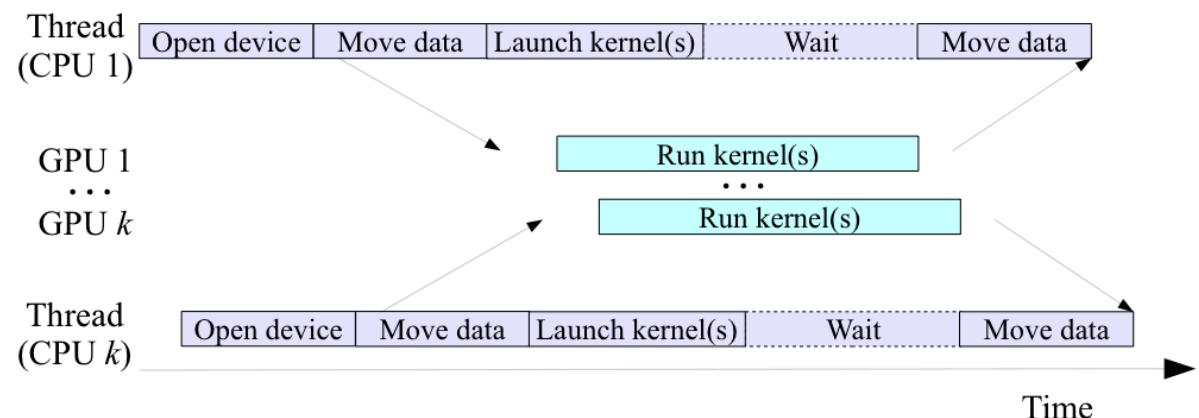
- Single GPU



- Multi-stream



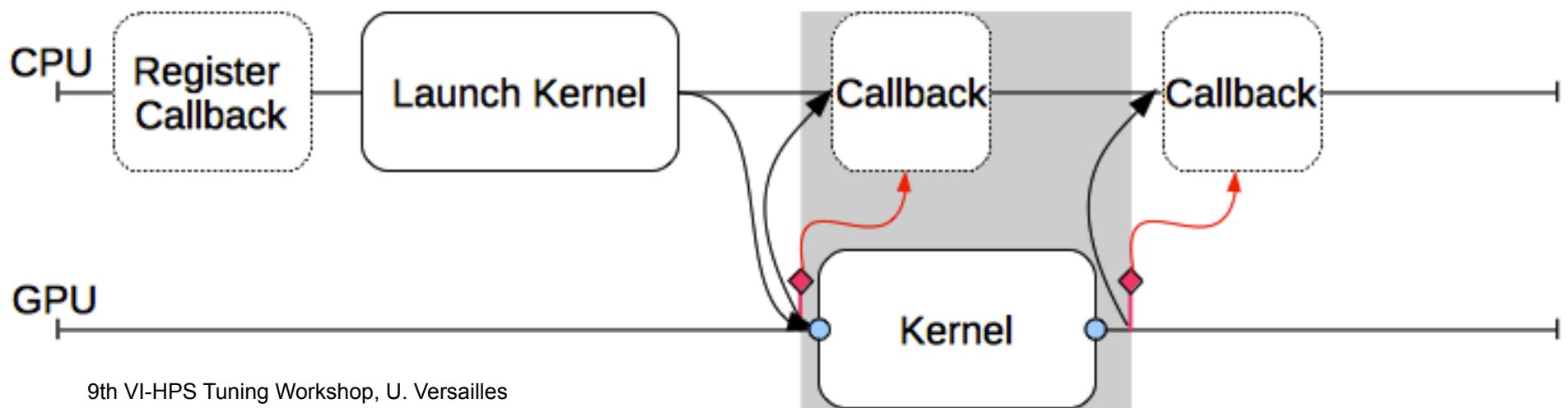
- Multi-CPU,  
Multi-GPU



# Host-GPU Measurement – Callback Method

VI-HPS

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

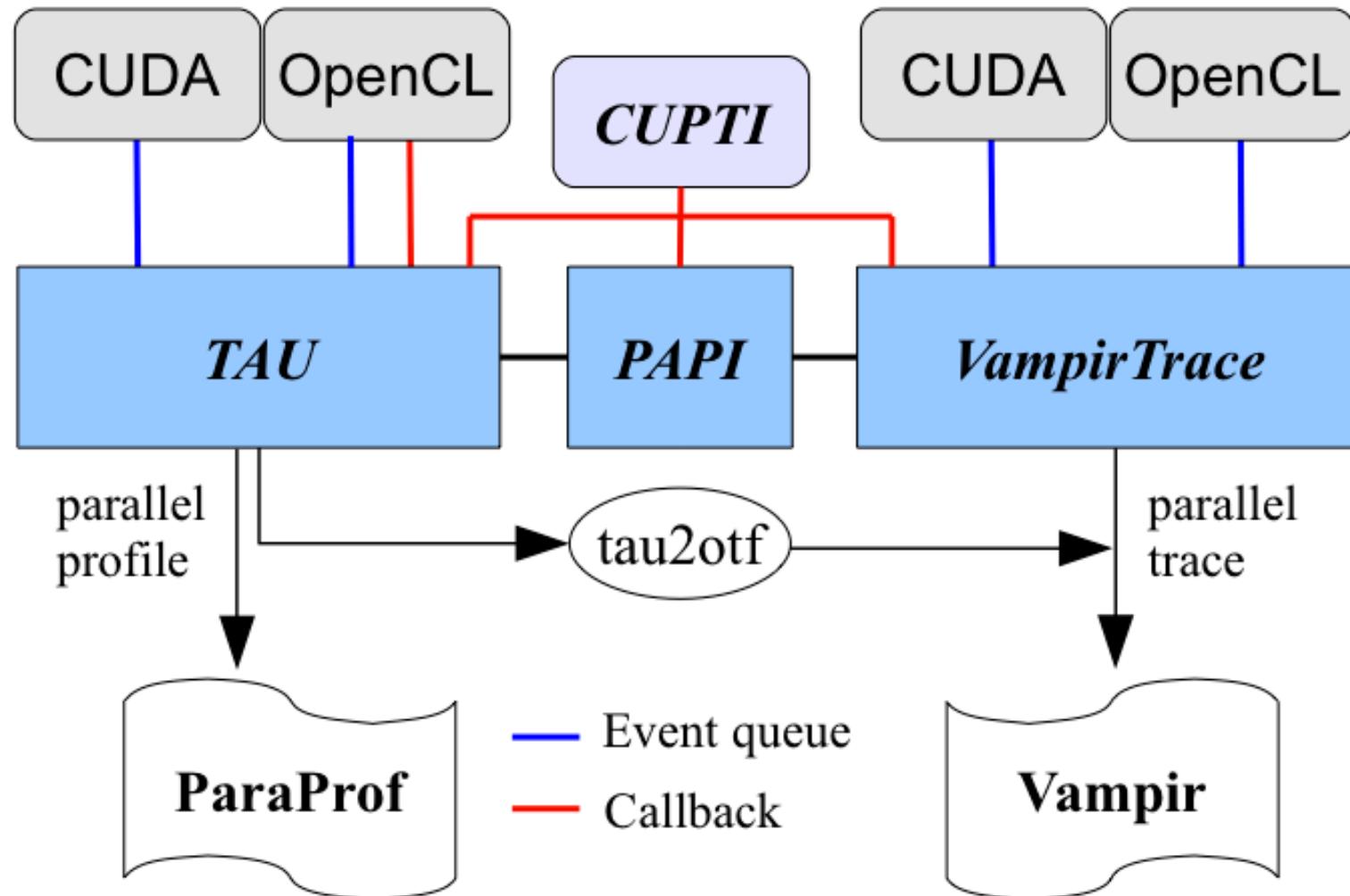


- Synchronous method
  - Place instrumentation appropriately around GPU calls (kernel launch, library routine, ...)
  - Wrap (synchronous) library with performance tool
- Event queue method
  - Utilize CUDA and OpenCL event support
  - Again, need instrumentation to create and insert events in the streams with kernel launch and process events
  - Can be implemented with driver library wrapping
- Callback method
  - Utilize language-level callback support in OpenCL
  - Utilize NVIDIA CUDA Performance Tool Interface (CUPTI)
  - Need to appropriately register callbacks

- Support the Host-GPU performance perspective
- Provide integration with existing measurement system to facilitate tool use
- Utilize support in GPU driver library and device
- Tools
  - TAU performance system
  - Vampir
  - PAPI
  - NVIDIA CUPTI

# GPU Performance Tool Interoperability

VI-HPS



- NVIDIA is developing CUPTI to enable the creation of profiling and tracing tools
- Callback API
  - Interject tool code at the entry and exit to each CUDA runtime and driver API call
- Counter API
  - Query, configure, start, stop, and read the counters on CUDA-enabled devices
- CUPTI is delivered as a dynamic library
- CUPTI is released with CUDA 4.1

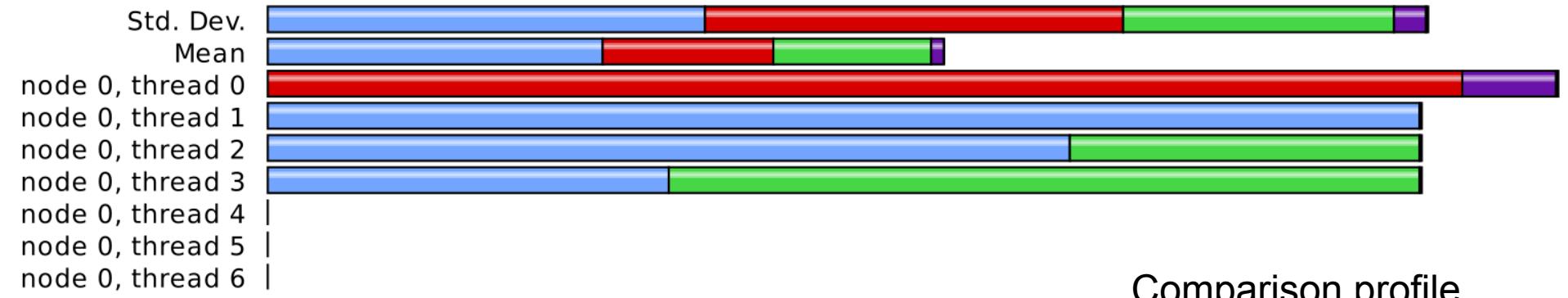
- Multiple performance perspectives
- Integrate Host-GPU support in TAU measurement framework
  - Enable use of each measurement approach
  - Include use of PAPI and CUPTI
  - Provide profiling and tracing support
- Tutorial
  - Use TAU library wrapping of libraries
  - Use `tau_exec` to work with binaries

```
% ./a.out  (uninstrumented)
% tau_exec -T serial -cuda ./a.out
% paraprof
```

- Demonstration of multiple GPU device use
- *main* → *solverThread* → *reduceKernel*
- One Keeneland node with three GPUs
- Performance profile for:
  - One *main* thread
  - Three *solverThread* threads
  - Three *reduceKernel* “threads”

Metric: TIME

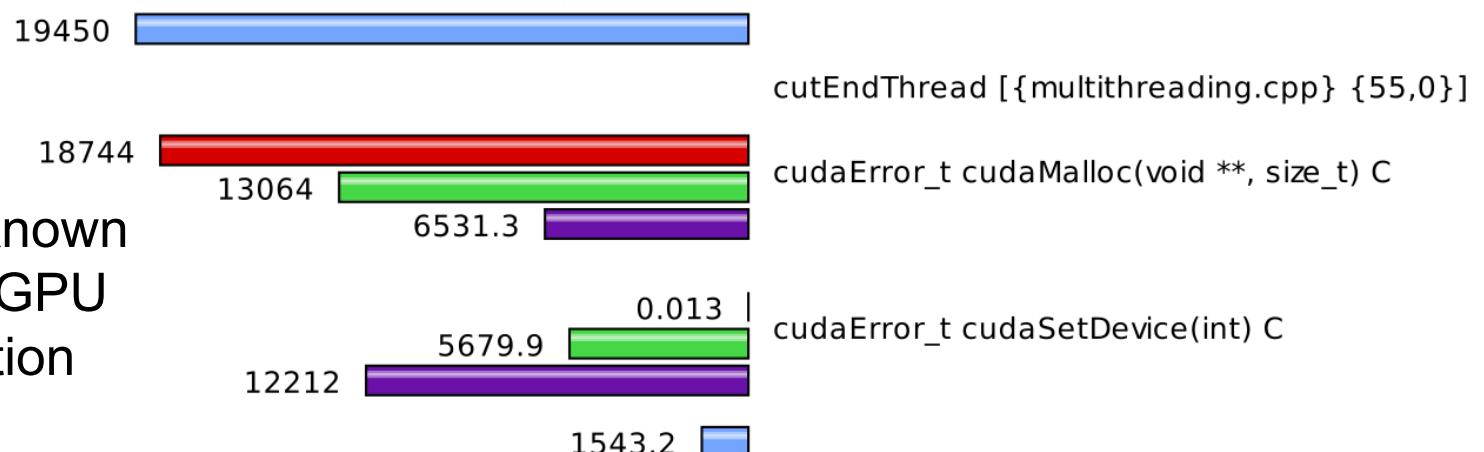
Value: Exclusive



Comparison profile

Metric: TIME  
Value: Exclusive  
Units: milliseconds

node 0, thread 0  
node 0, thread 1  
node 0, thread 2  
node 0, thread 3



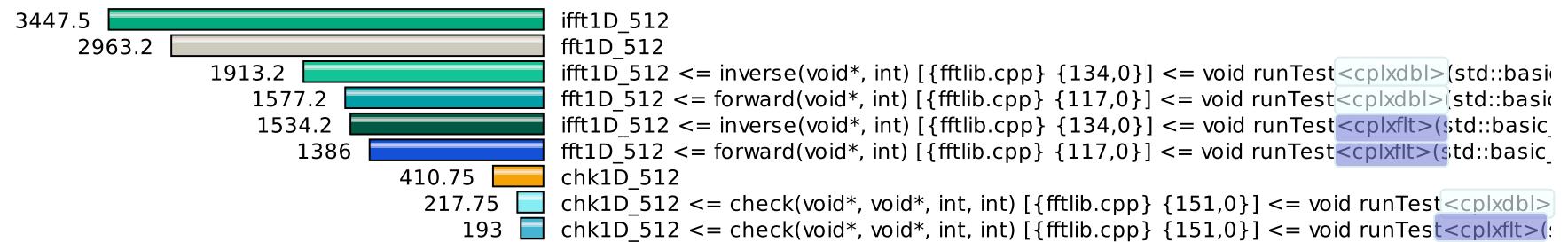
Identified a known overhead in GPU context creation

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

Metric: TAUGPU\_TIME

Value: Exclusive

Units: microseconds

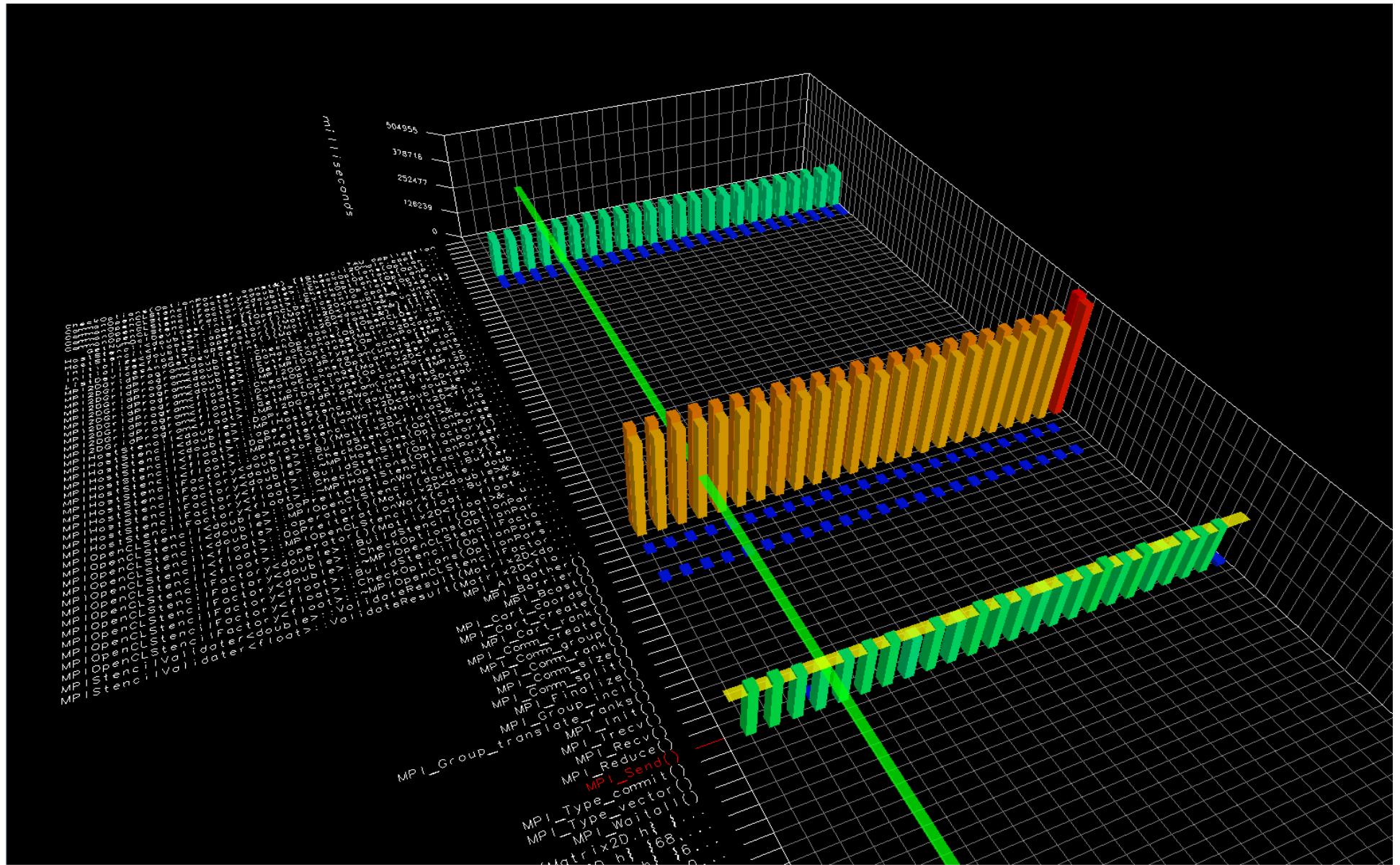


Each kernel (ifft1D\_512, fft1D\_512 and chk1D\_512) is broken down by callsite, either during the single precession or double precession step.

- Compute 2D, 9-point stencil
  - Multiple GPUs using MPI
  - CUDA and OpenCL versions
- One Keeneland node with 3 GPUs
- Eight Keeneland nodes with 24 GPUs
- Performance profile and trace
  - Application events
  - Communication events
  - Kernel execution

# Stencil2D Parallel Profile

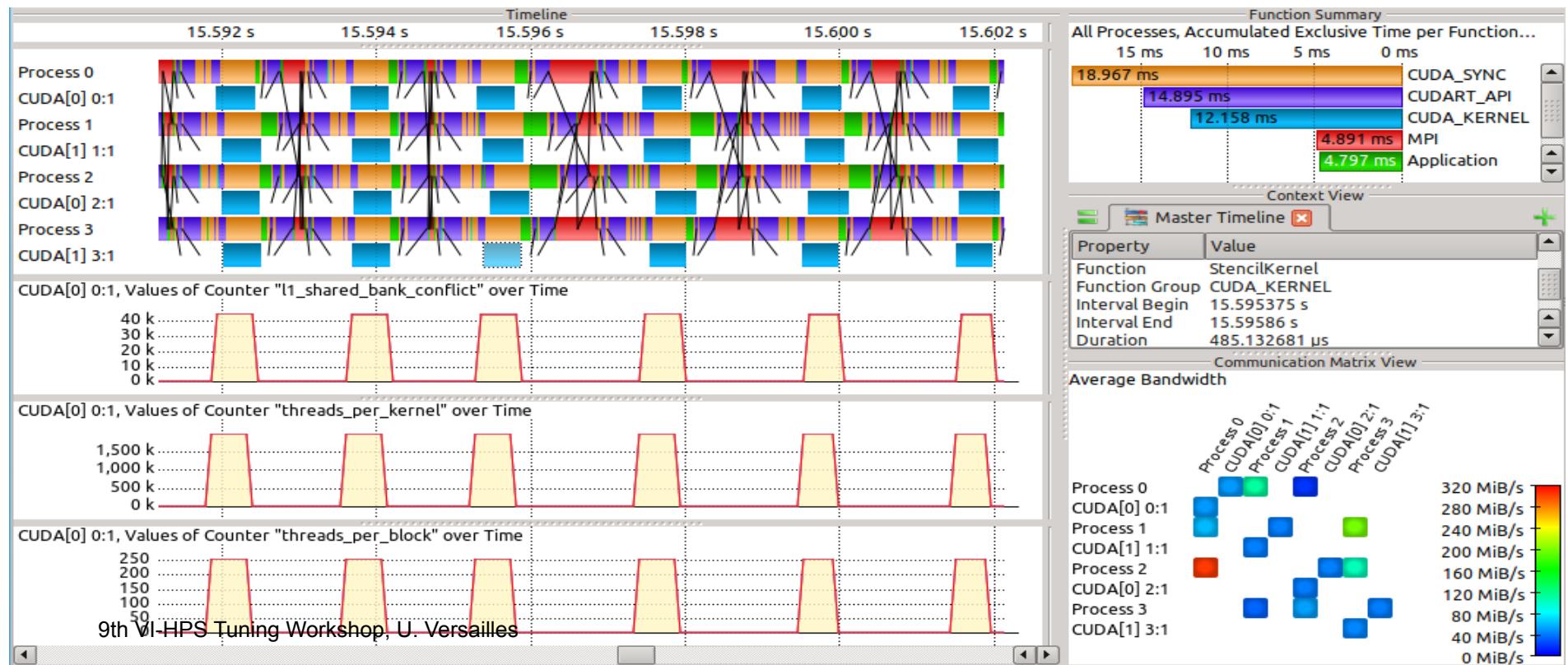
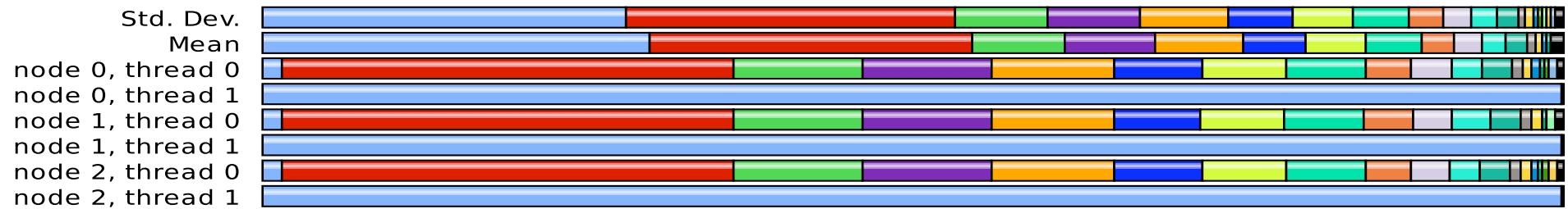
**VI-HPS**



# Stencil2D Parallel Profile / Trace in Vampir

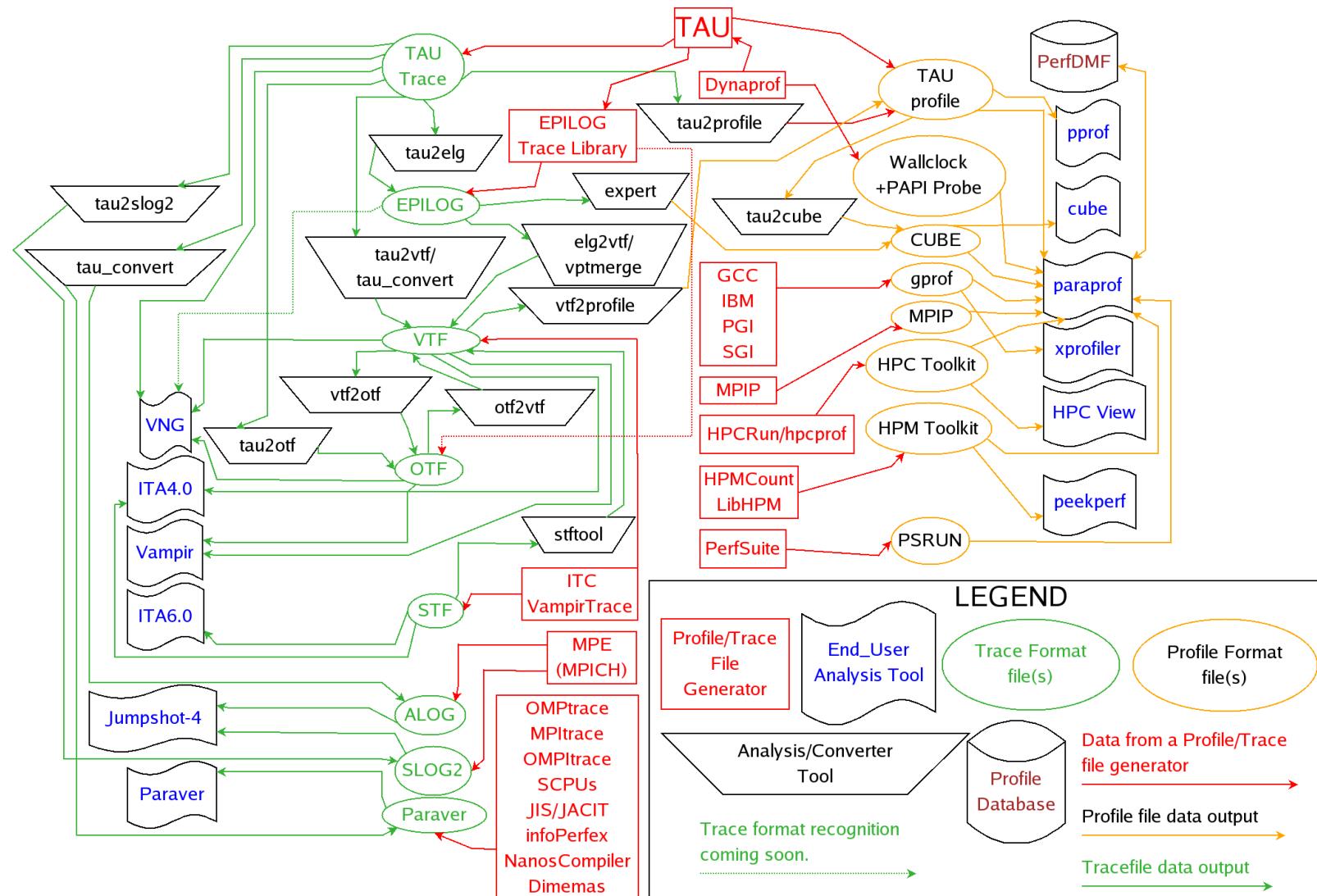
VI-HPS

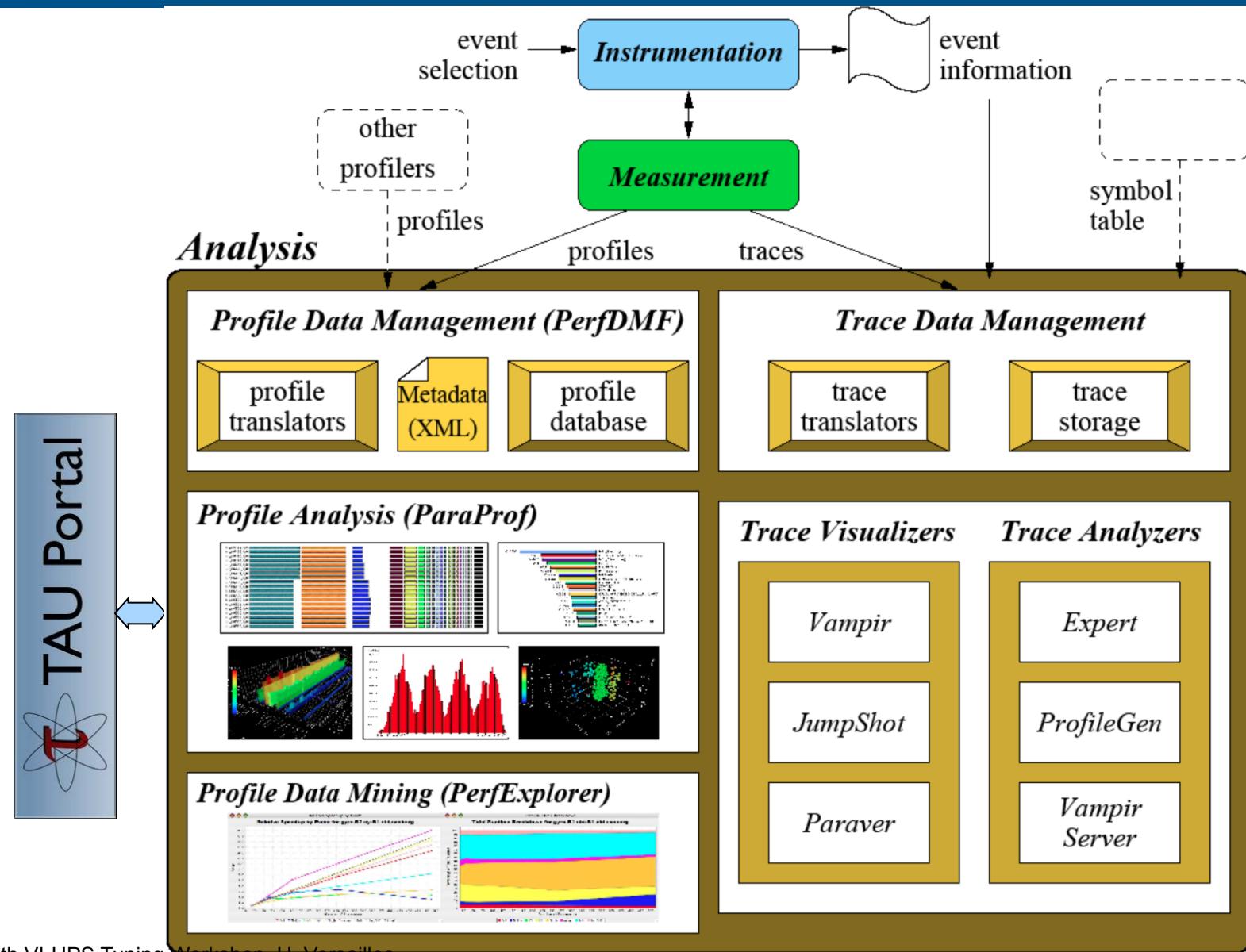
Metric: TAUGPU\_TIME  
Value: Exclusive



# Building Bridges to Other Tools

VI-HPS





## Example: NAMD with CUPTI

VI-HPS



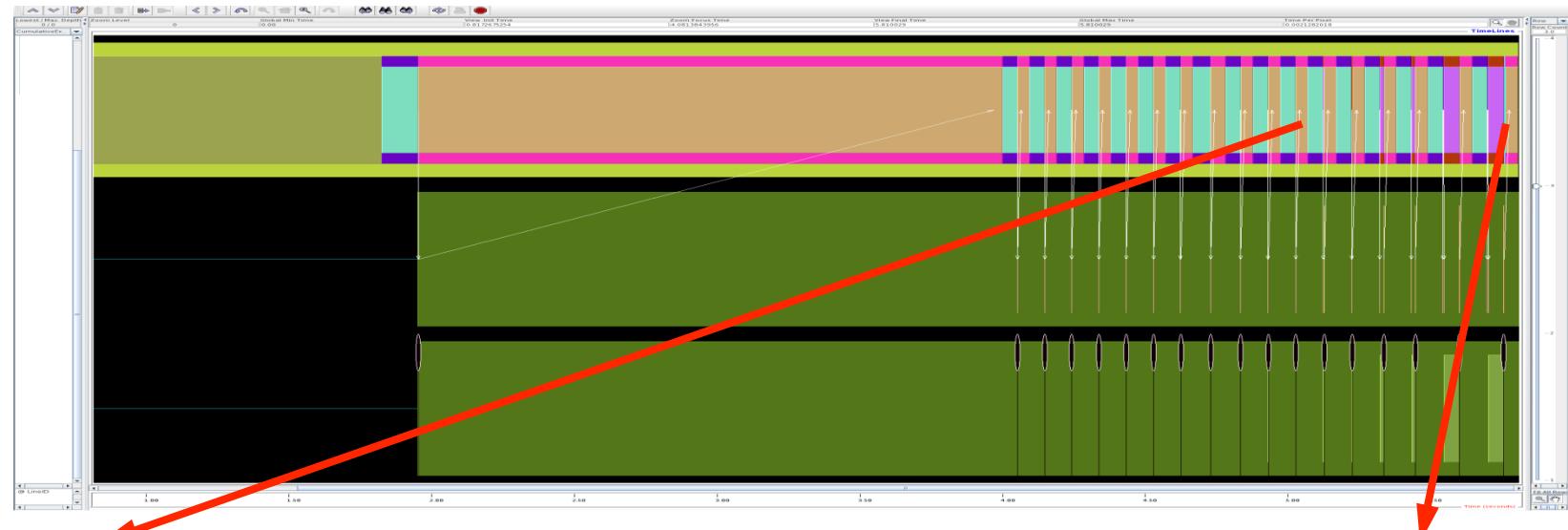
# HMPP SGEMM (CAPS Enterprise)

VI-HPS

Host  
Process

Transfer  
Kernel

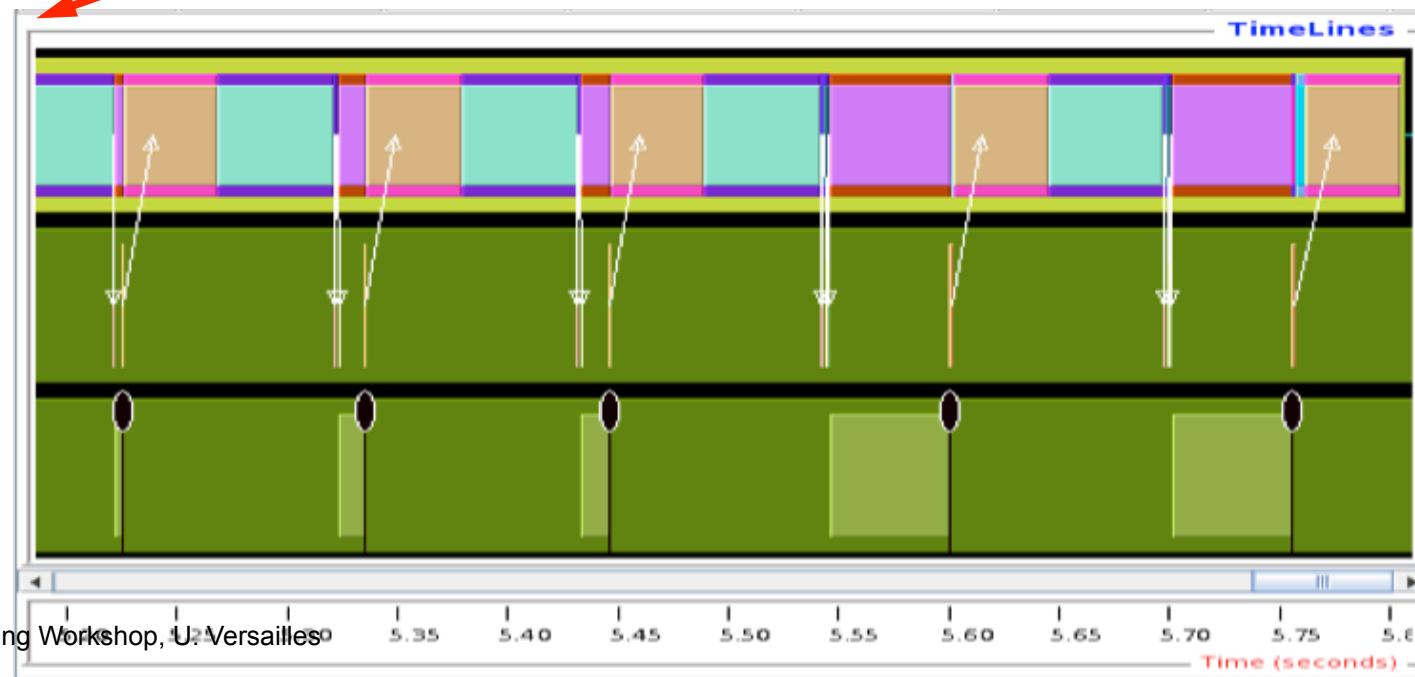
Compute  
Kernel



Host  
Process

Transfer  
Kernel

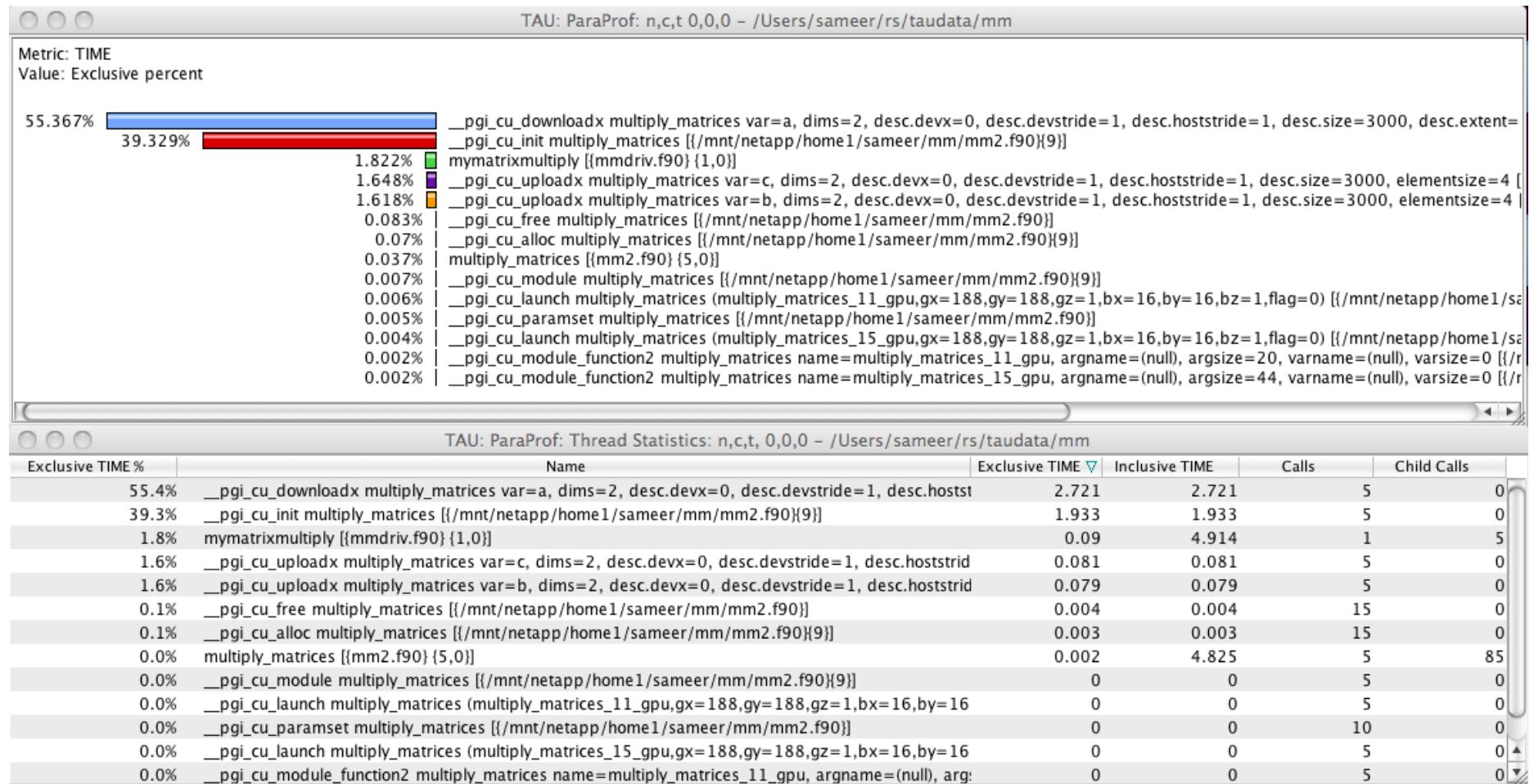
Compute  
Kernel



- PGI compiler allows users to annotate source code to identify loops that should be accelerated
- When a program is compiled with TAU, its measurement library intercepts the PGI runtime library layer to measure time spent in the runtime library routines and data transfers
- TAU also captures the arguments:
  - array data dimensions and sizes, strides, upload and download times, variable names, source file names, row and column information, and routines

# Example: PGI GPU-accelerated MM

VI-HPS



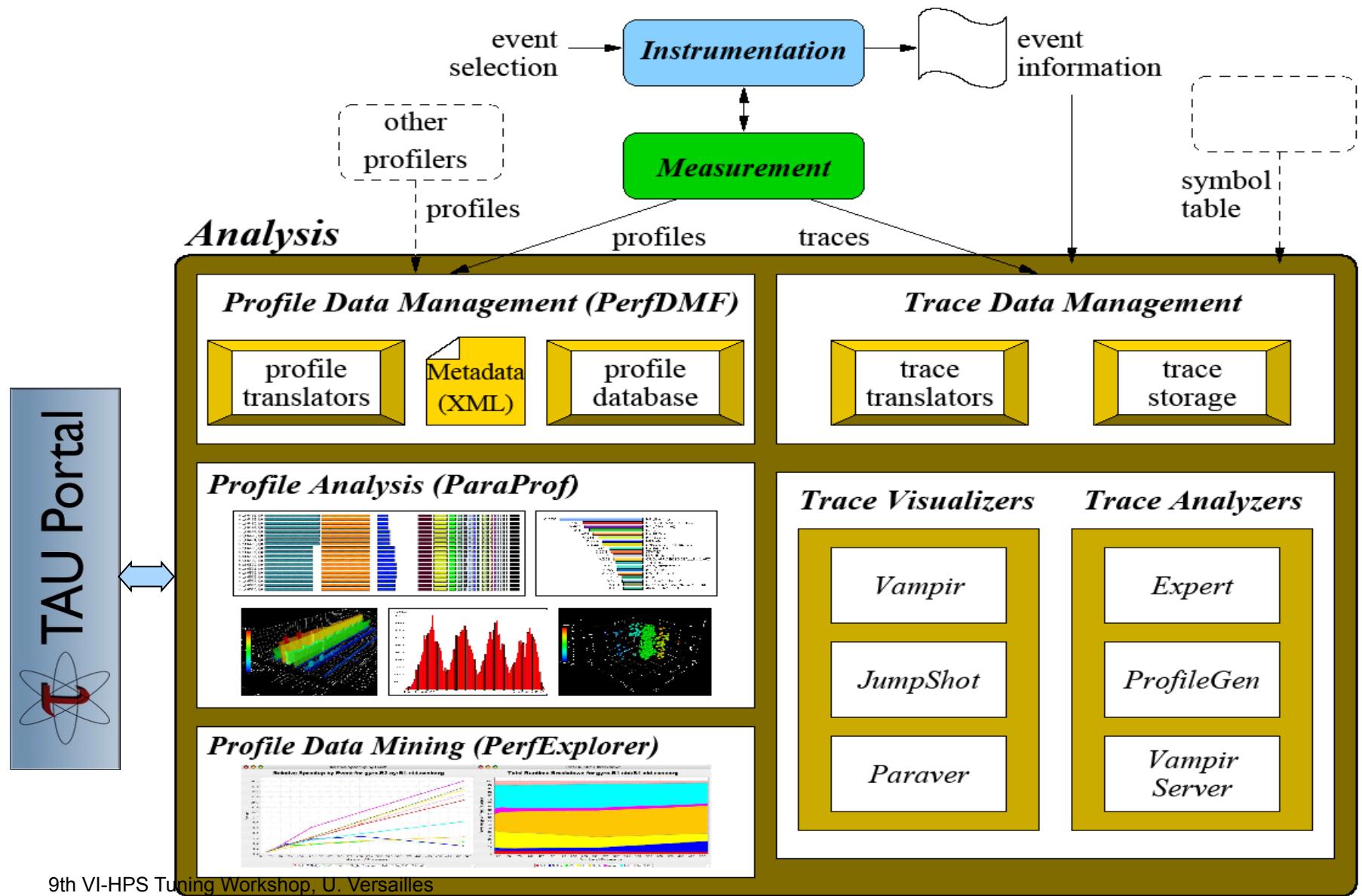
The screenshot shows the TAU ParaProf interface with two main windows. The bottom-left window is titled "TAU: ParaProf: Source Browser: ./mm2.f90" and displays the following Fortran code:

```
1 ! Simple matmul example
2
3     module mymm
4         contains
5             subroutine multiply_matrices( a, b, c, m )
6                 real, dimension(:,:) :: a,b,c
7                 i = 0
8
9                 !$acc region
10                 do j = 1,m
11                     do i = 1,m
12                         a(i,j) = 0.0
13                         enddo
14                         do k = 1,m
15                             do i = 1,m
16                                 a(i,j) = a(i,j) + b(i,k) * c(k,j)
17                                 enddo
18                                 enddo
19                         enddo
20                         !$acc end region
21                     end subroutine
22                 end module
23
```

The top-right window is titled "TAU: ParaProf: Function Data Window: /Users/sameer/rs/taudata/pgiacc/mm/PGI11.2" and displays performance metrics for the subroutine:

Name: \_\_pgi\_cu\_downloadx multiply\_matrices var=a, dims=2, desc.devx=0, desc.devstride=1, desc.hoststride=1, desc.size=3000, desc.extent=3000, elementsize=4  
[[/mnt/netapp/home1/sameer/mm/mm2.f90]{20}]]  
Metric Name: TIME  
Value: Exclusive  
Units: seconds

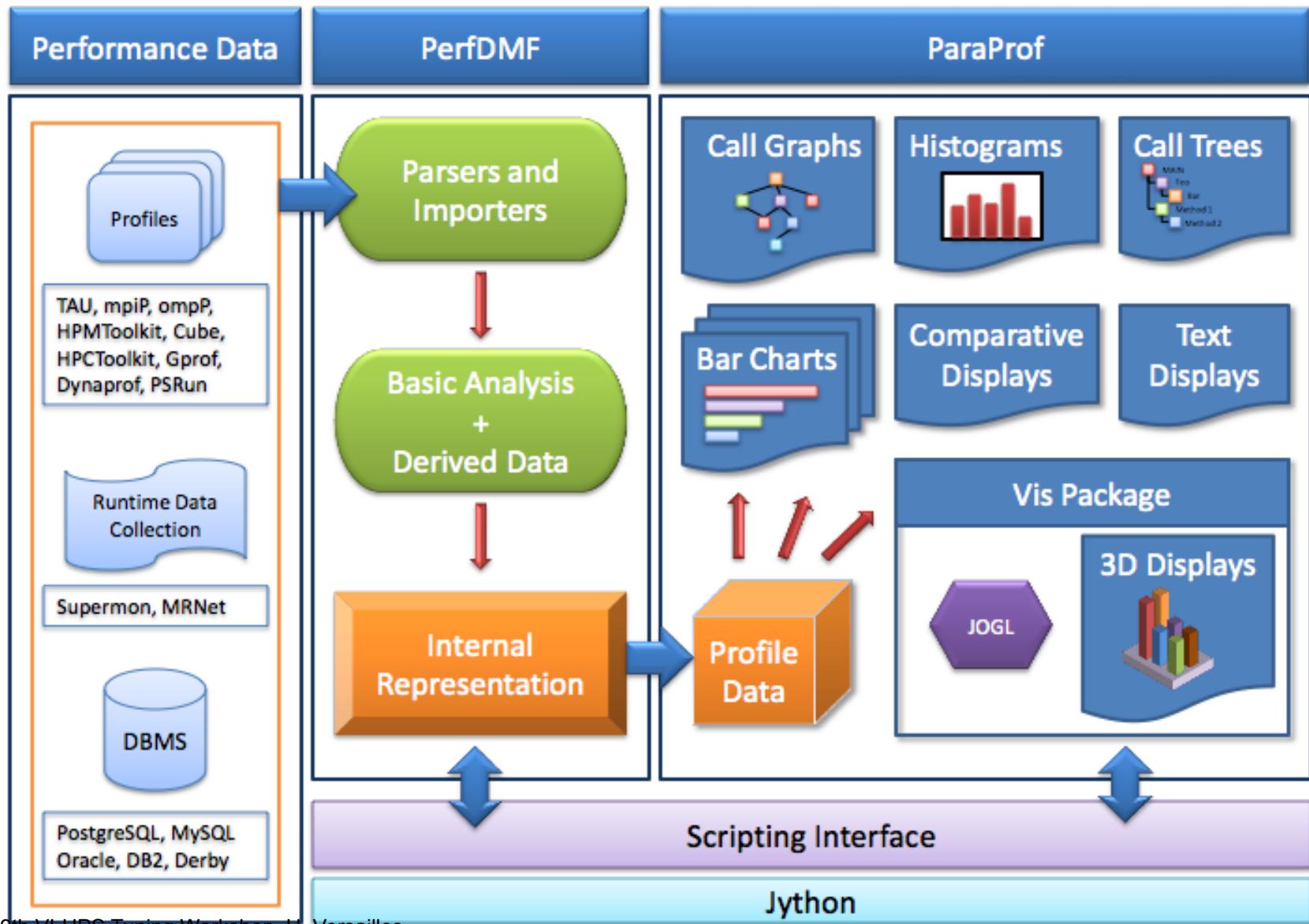
A progress bar indicates a value of 2.722 for "node 0". A context menu is open over the progress bar, listing options: Show Source Code (highlighted in blue), Show Function Bar Chart, Show Function Histogram, Assign Function Color, Reset to Default Color, and Rename.

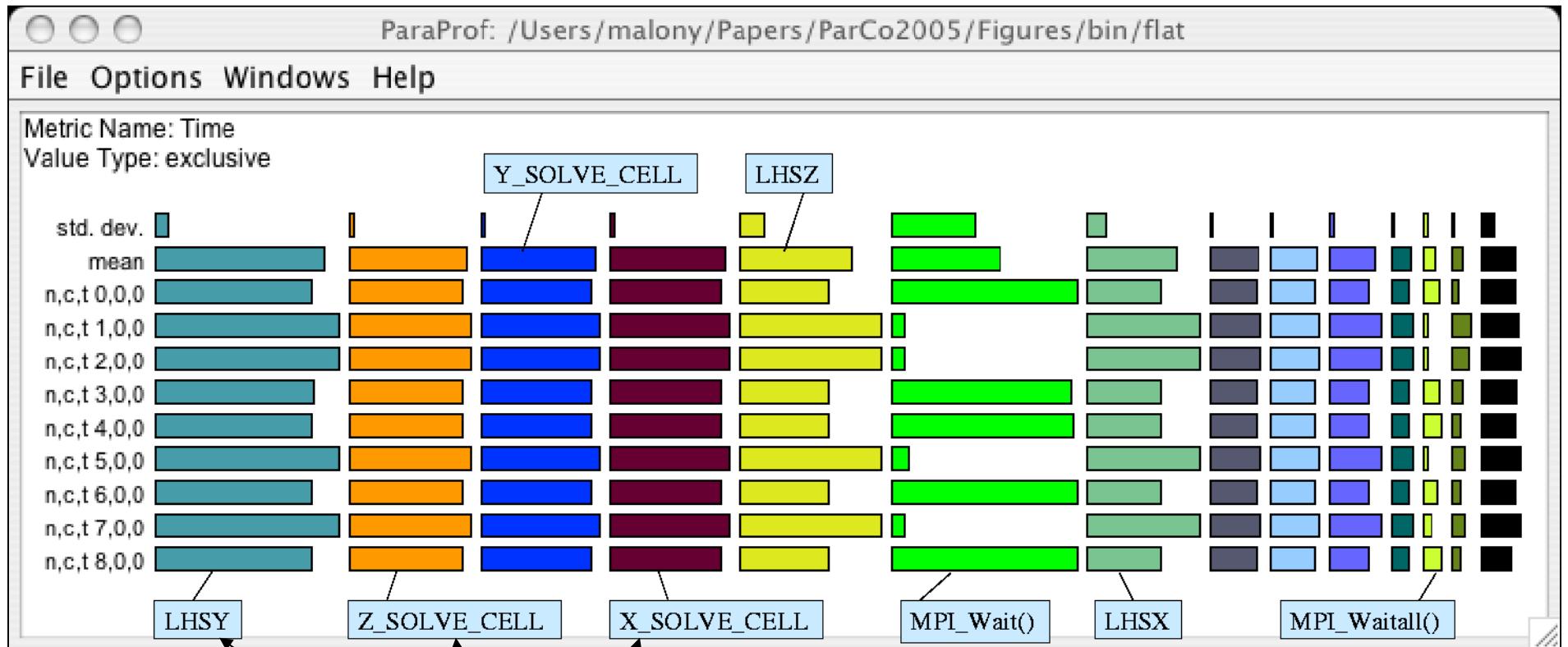


- Analysis of parallel profile and trace measurement
- Parallel profile analysis (ParaProf)
  - Java-based analysis and visualization tool
  - Support for large-scale parallel profiles
- Performance data management framework (PerfDMF)
- Parallel trace analysis
  - Translation to VTF (V3.0), EPILOG, OTF formats
  - Integration with Vampir / Vampir Server (TU Dresden)
  - Profile generation from trace data
- Online parallel analysis and visualization
- Integration with CUBE browser (Scalasca, UTK / FZJ)

# ParaProf Profile Analysis Framework

VI-HPS



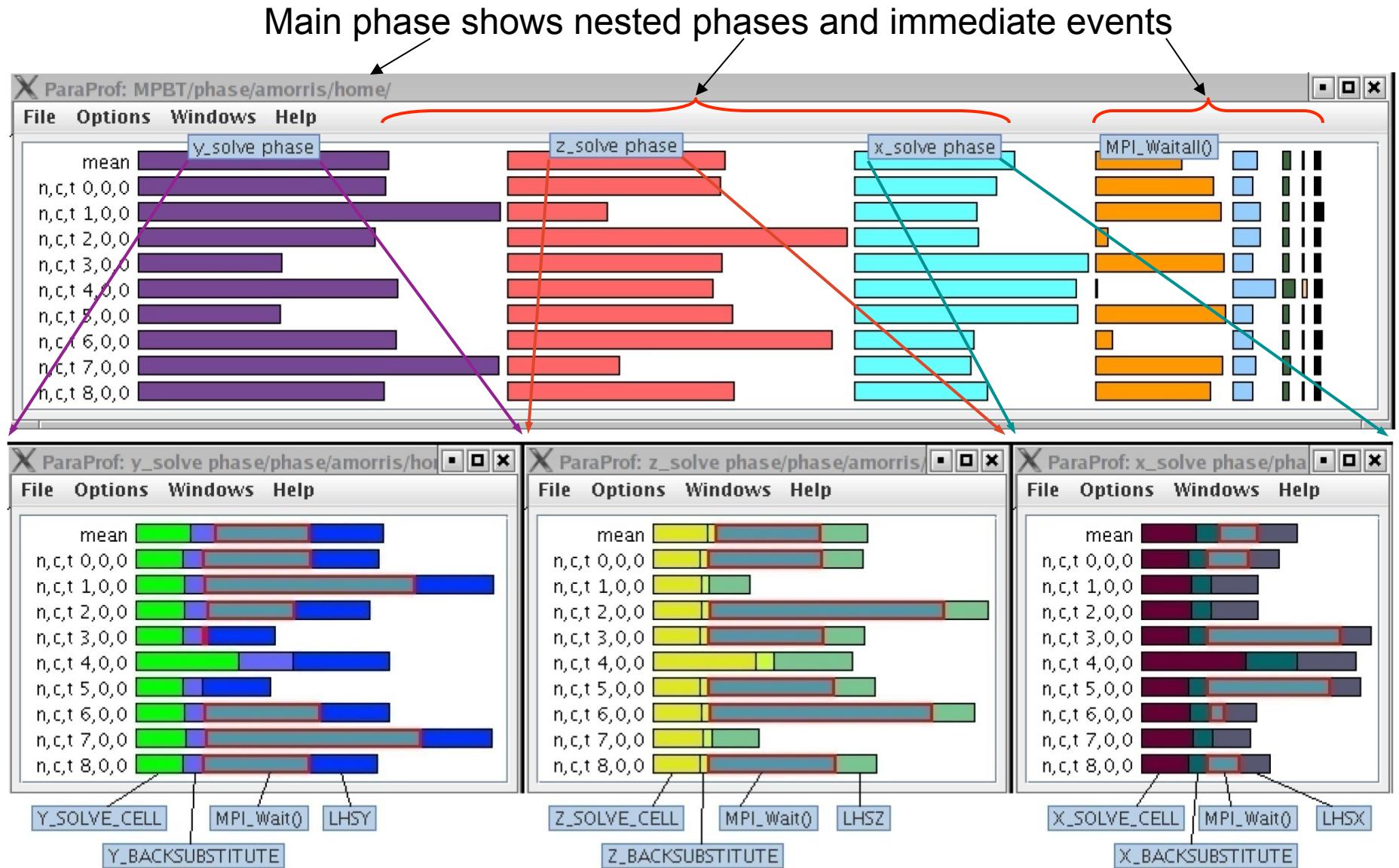


Application routine names reflect phase semantics

How is MPI\_Wait()  
distributed relative to  
solver direction?

# NAS BT – Phase Profile

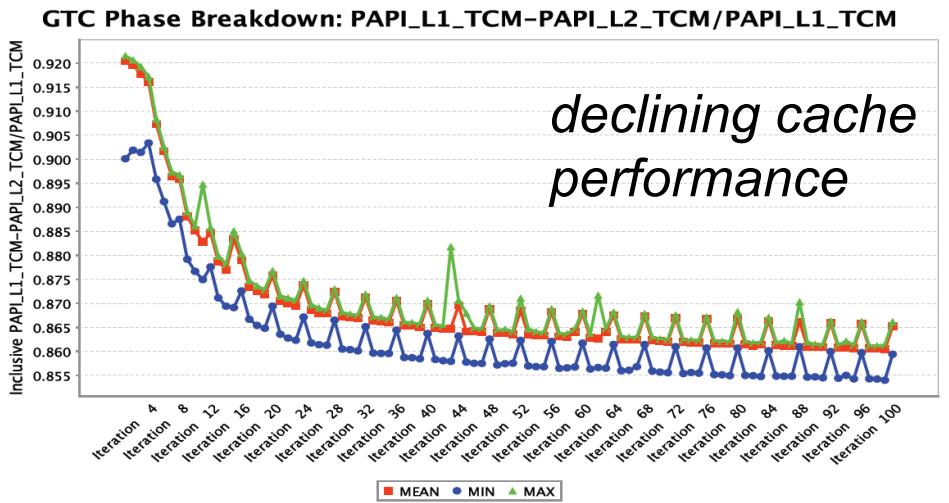
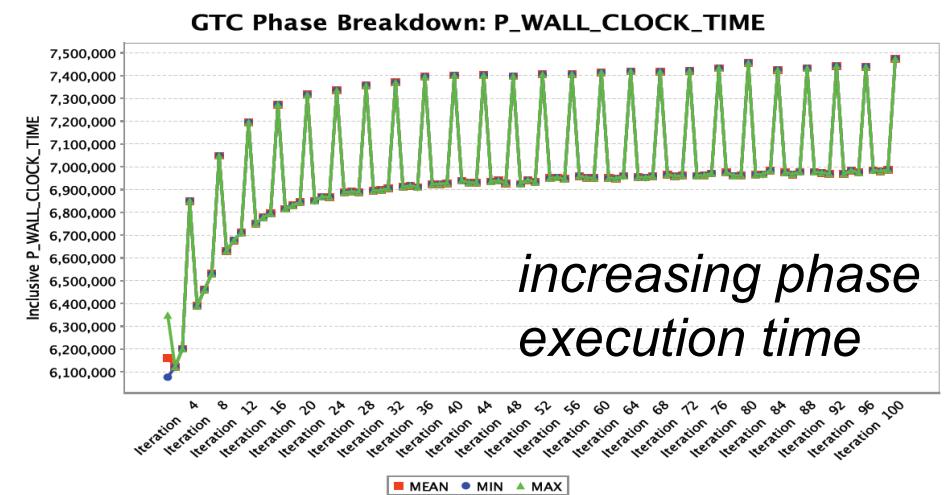
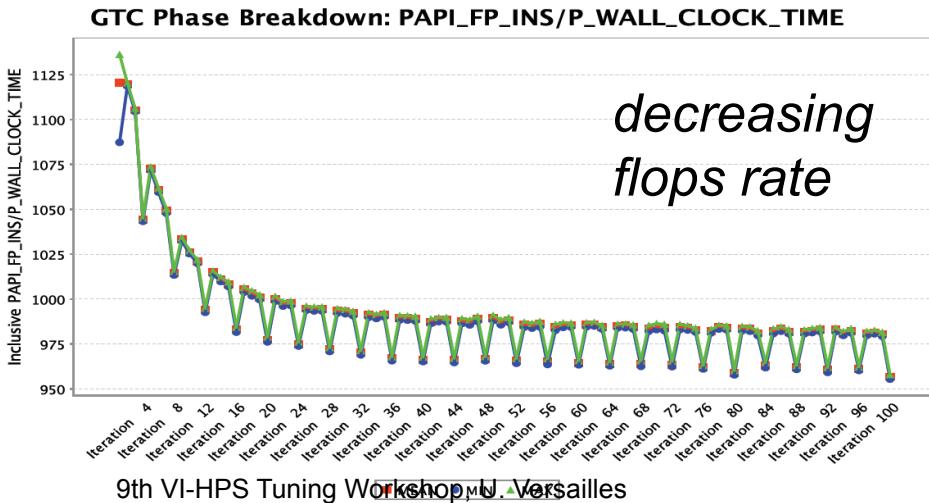
VI-HPS



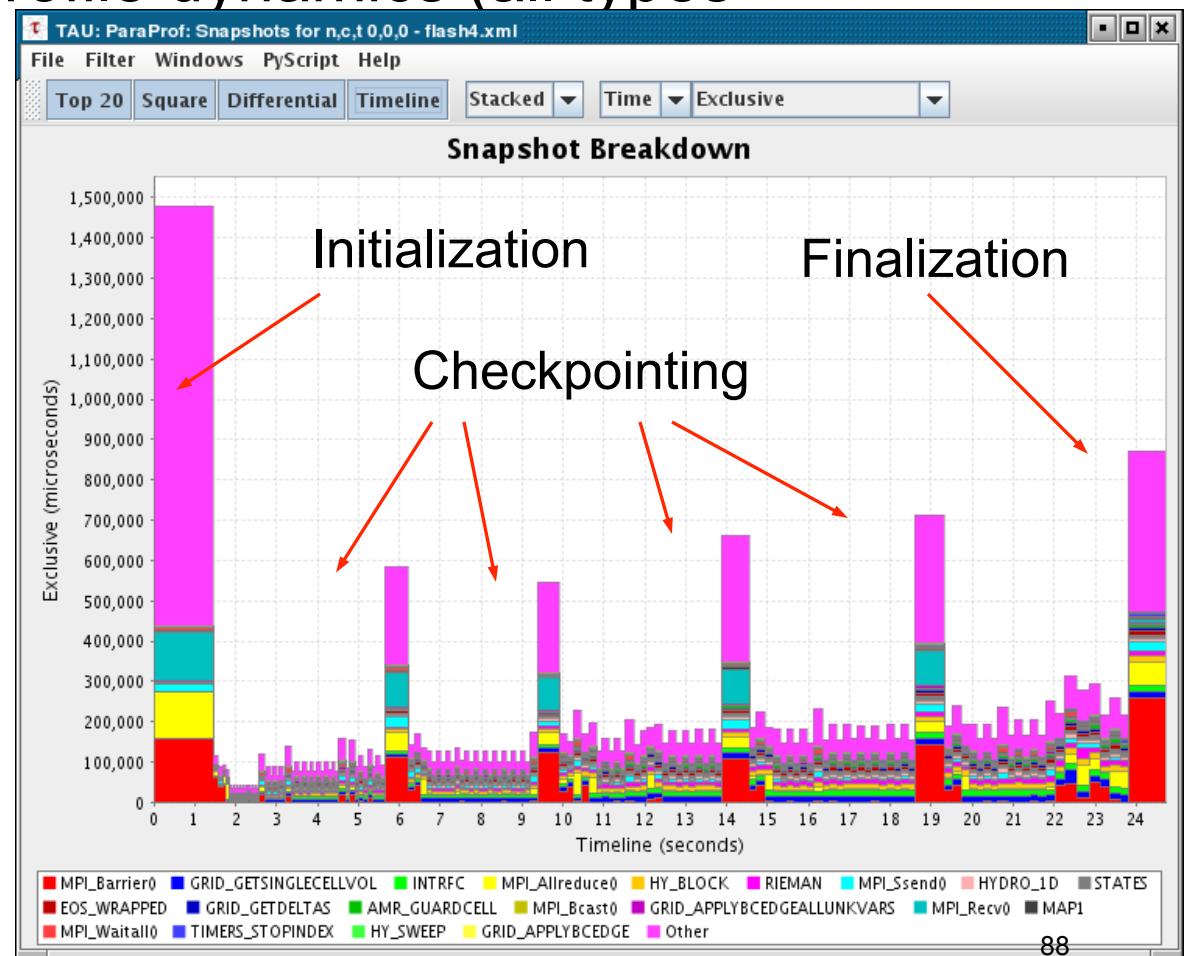
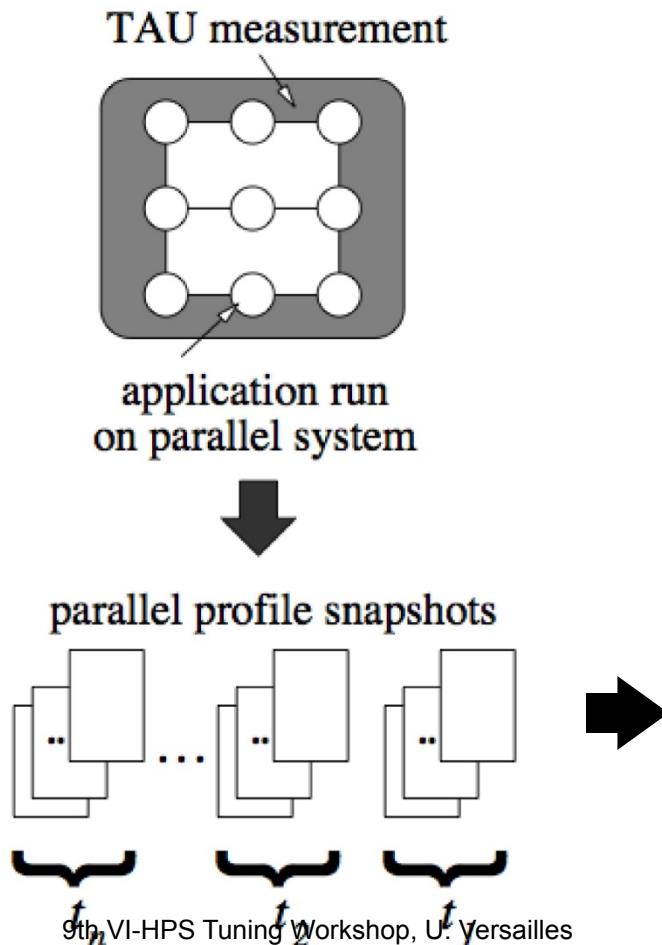
# Phase Profiling of HW Counters

VI-HPS

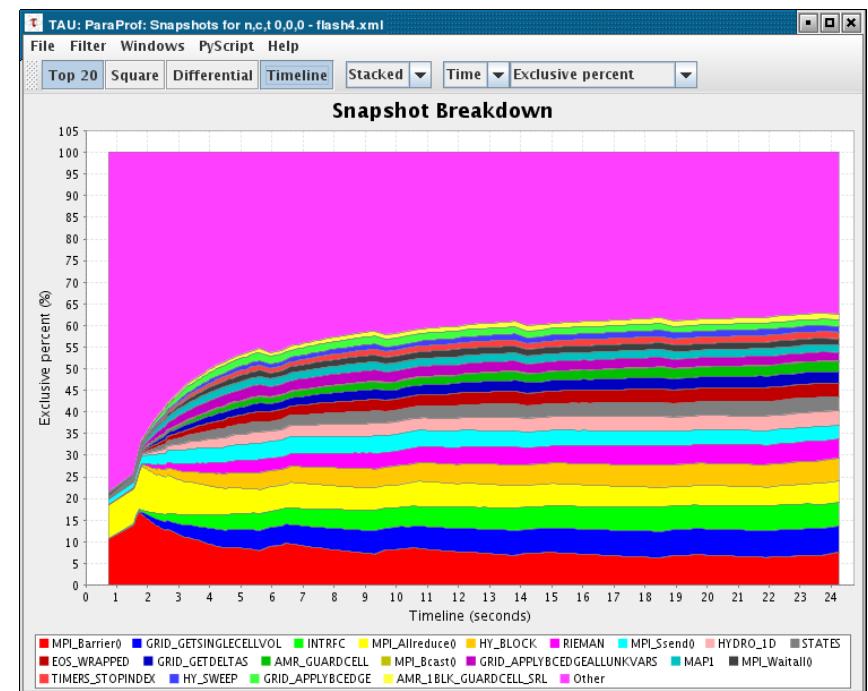
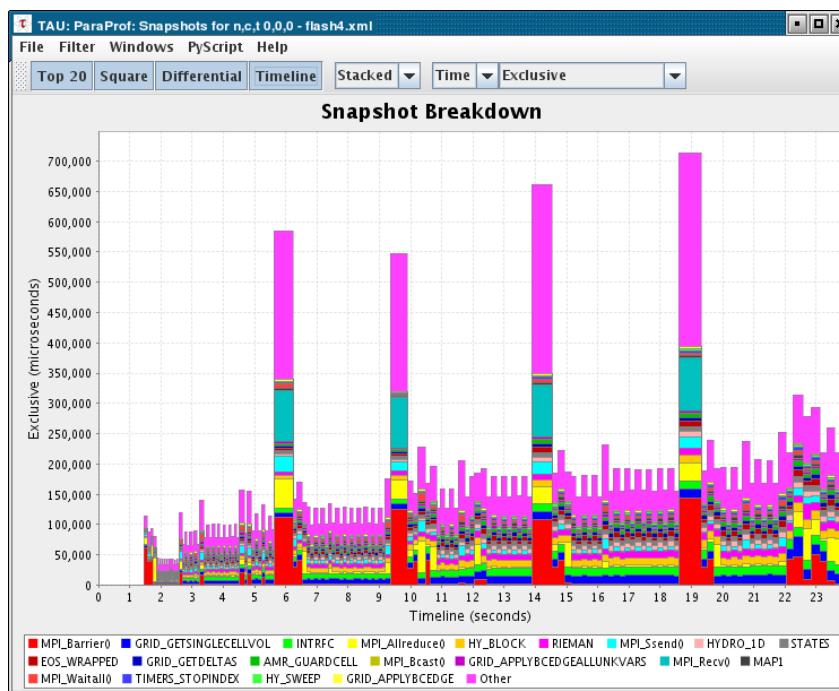
- GTC particle-in-cell simulation of fusion turbulence
- Phases assigned to iterations
- Poor temporal locality for one important data
- Automatically generated by PE2 python script



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

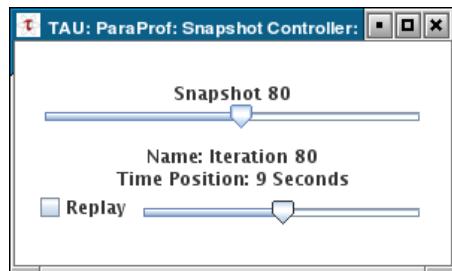


- Percentage breakdown
- Only show main loop

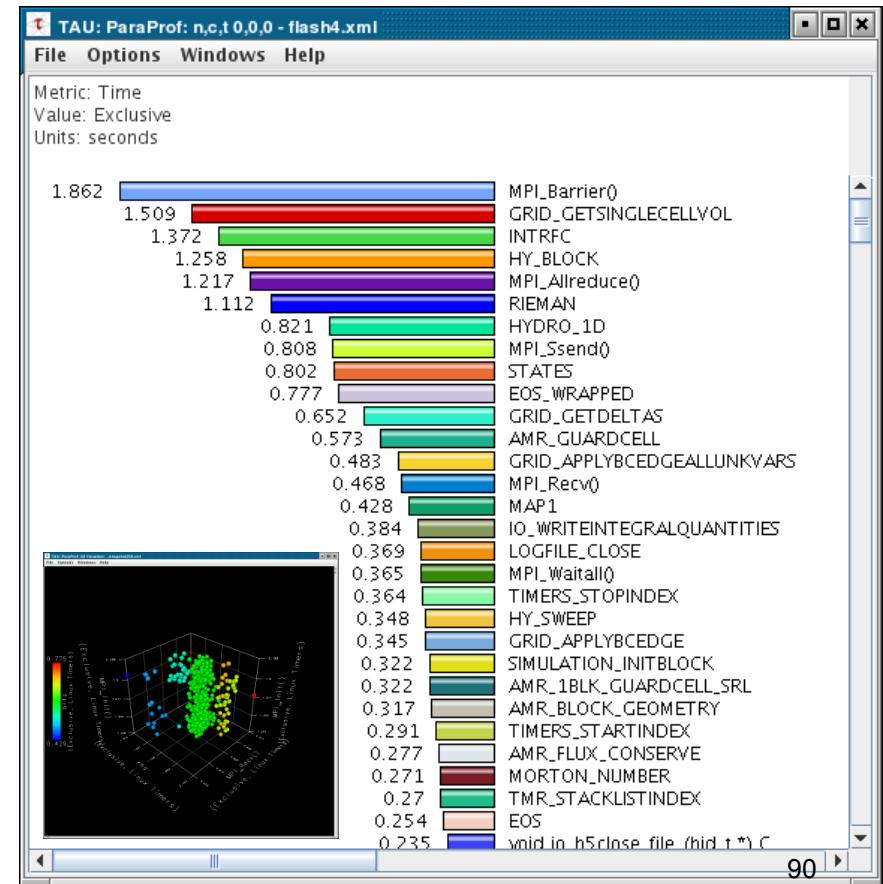
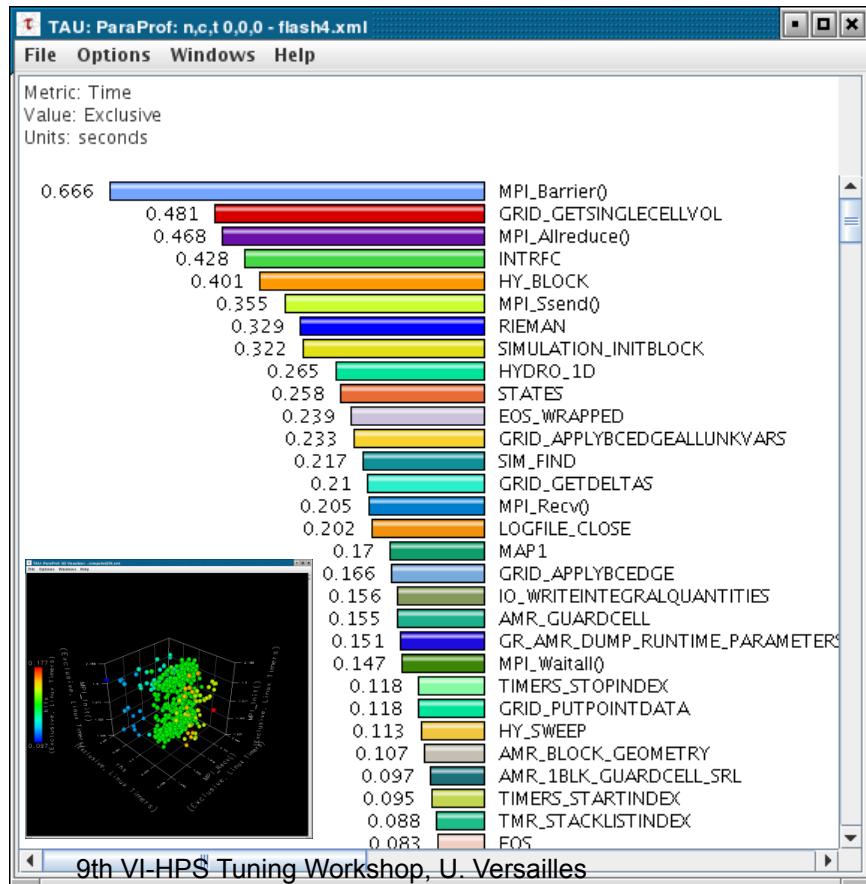
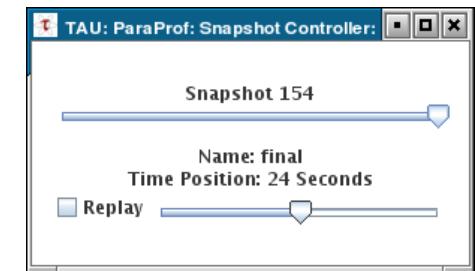


# Snapshot Replay in ParaProf

VI-HPS

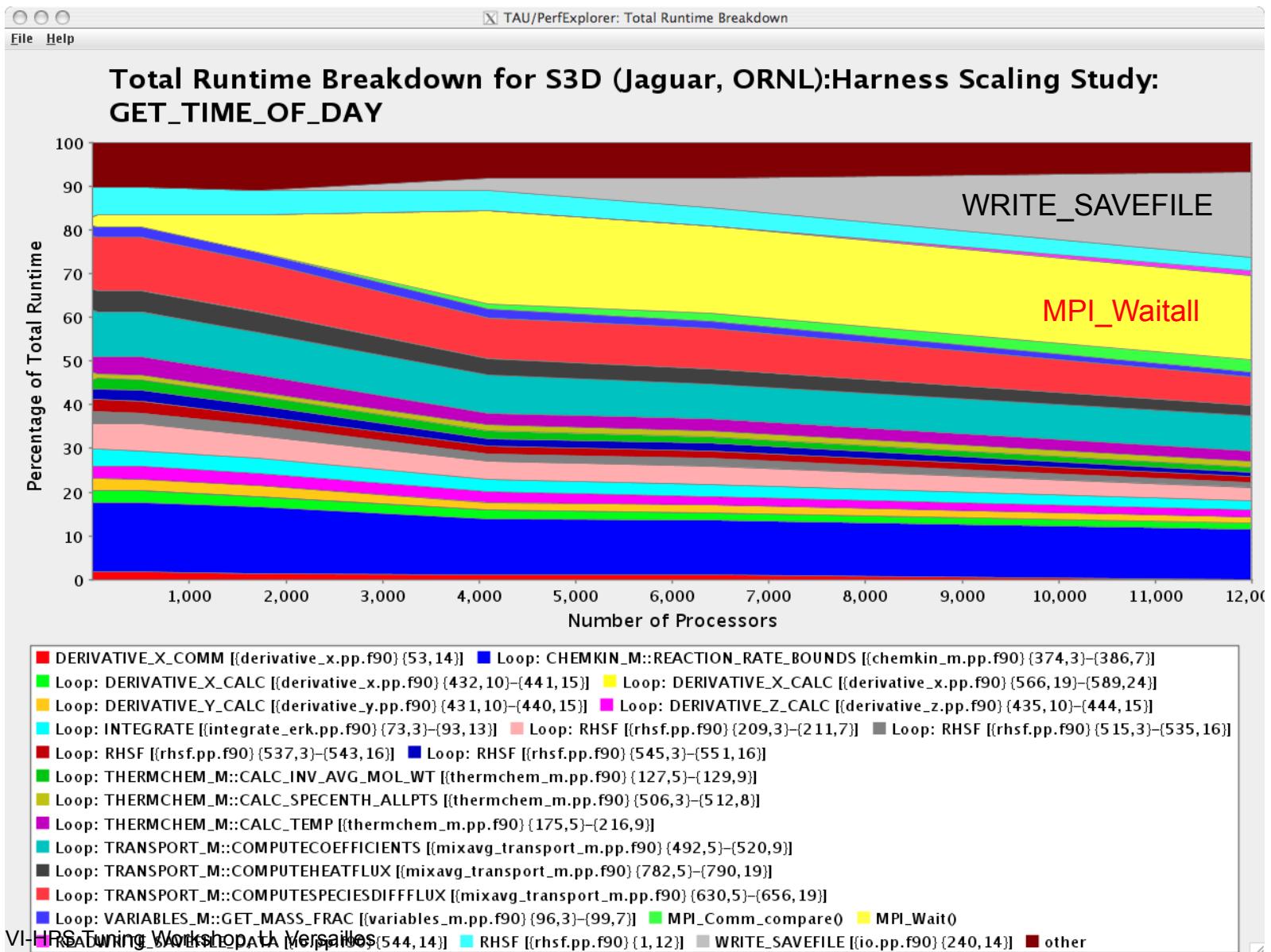


All windows dynamically update

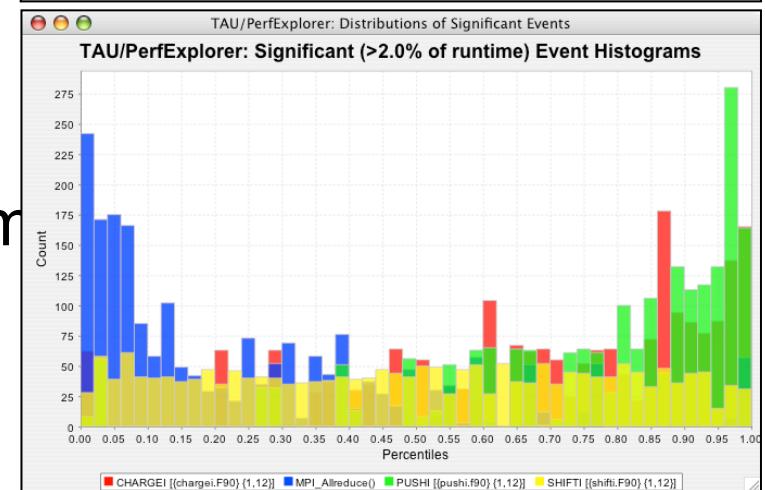
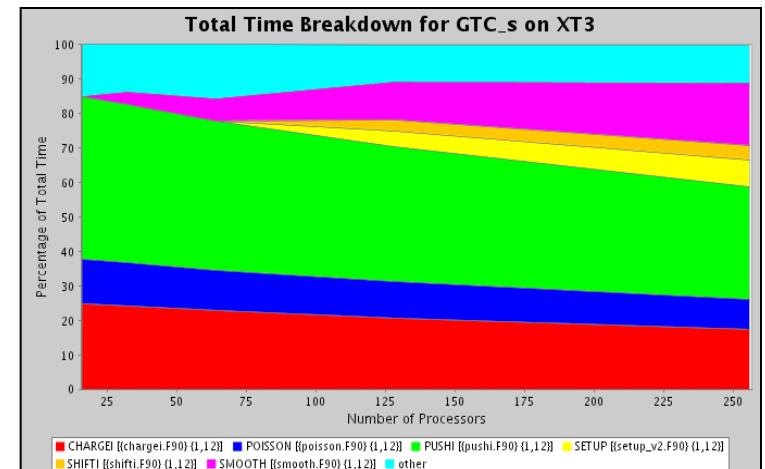


# PerfExplorer – Runtime Breakdown

VI-HPS



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

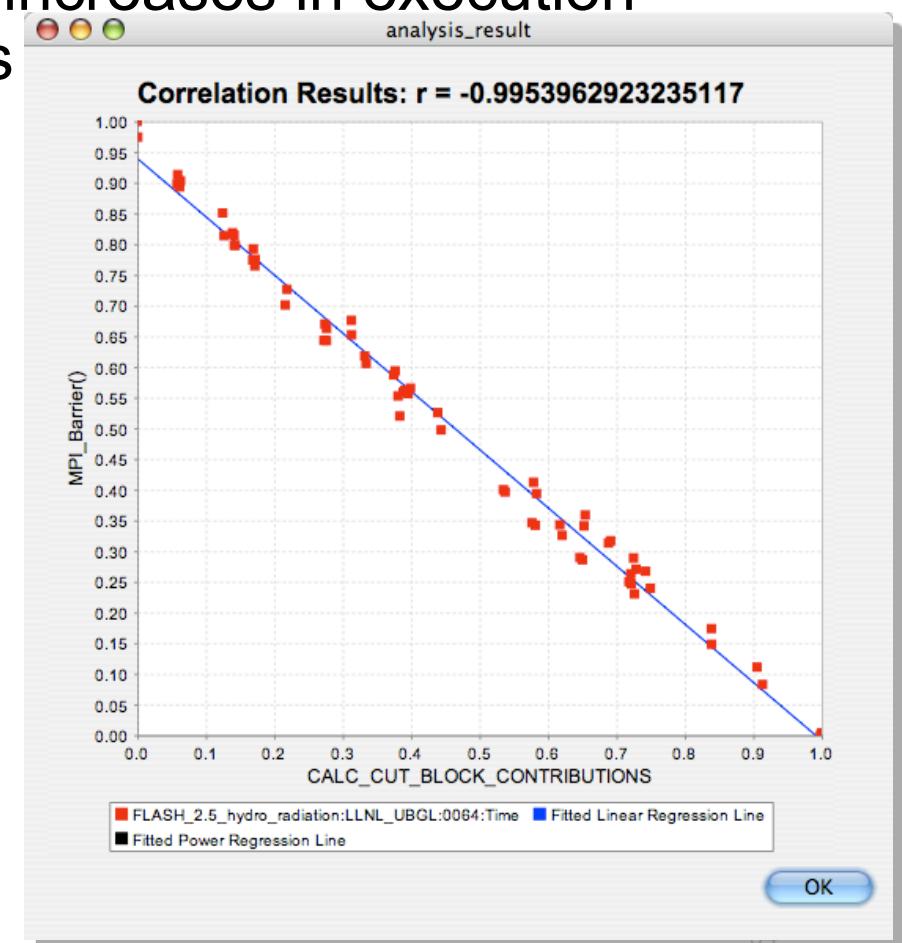


# PerfExplorer – Correlation Analysis

VI-HPS

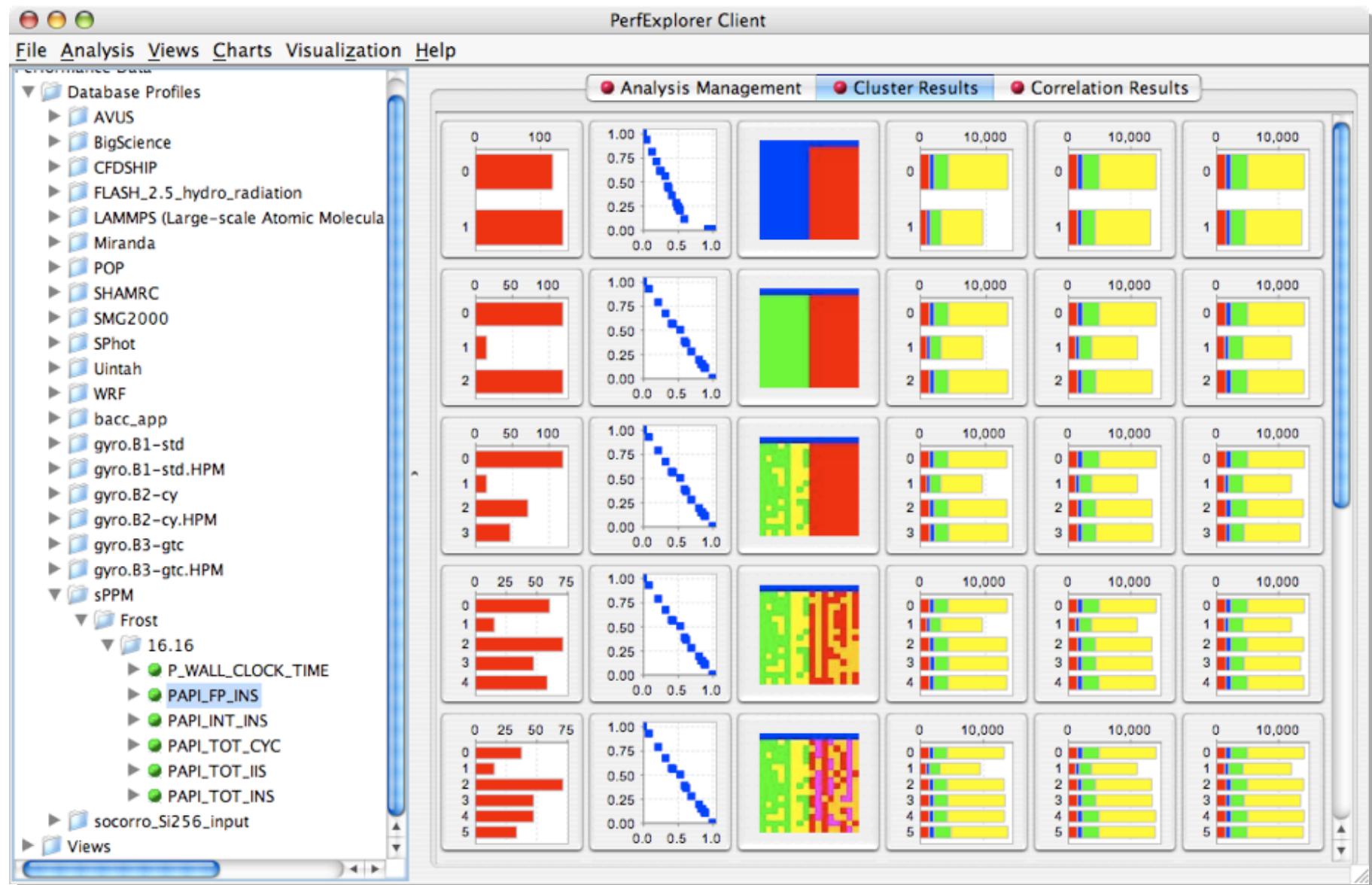


- -0.995 indicates strong, negative relationship
- As `CALC_CUT_BLOCK_CONTRIBUTIONS()` increases in execution time, `MPI_Barrier()` decreases

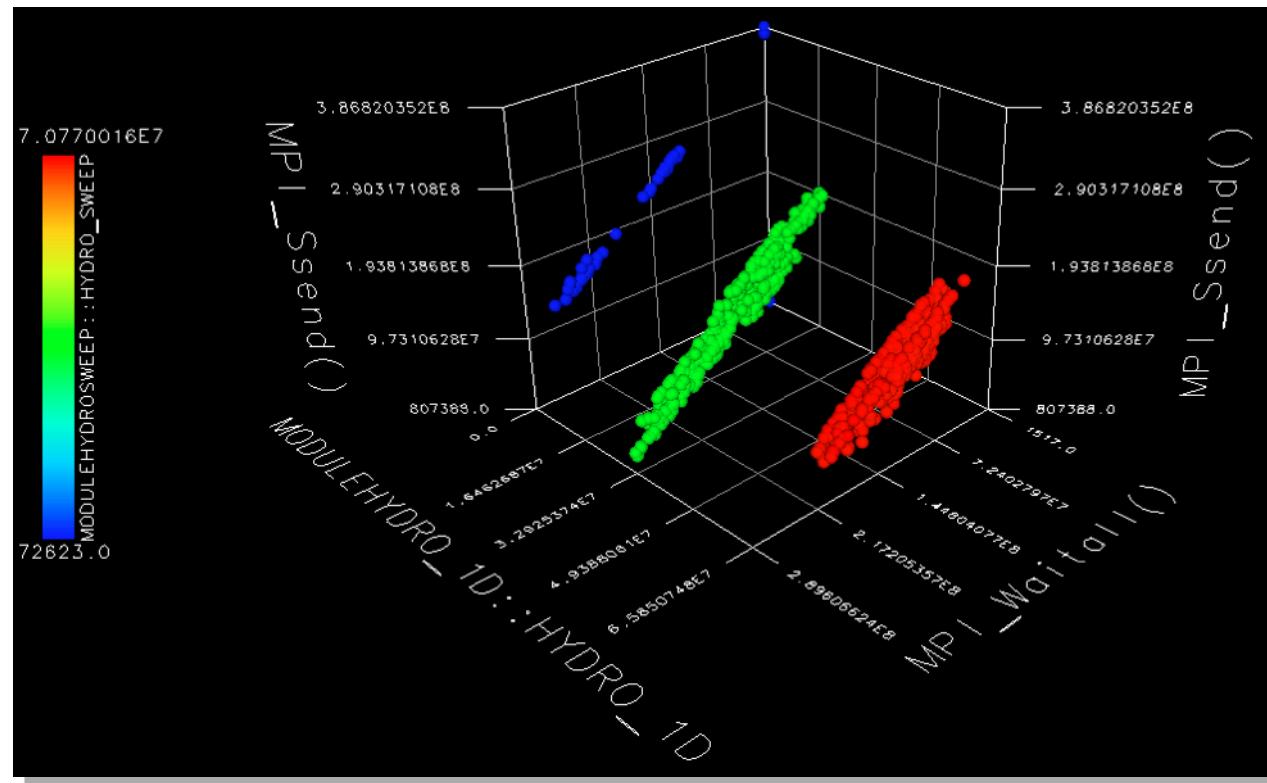


# PerfExplorer – Cluster Analysis

VI-HPS

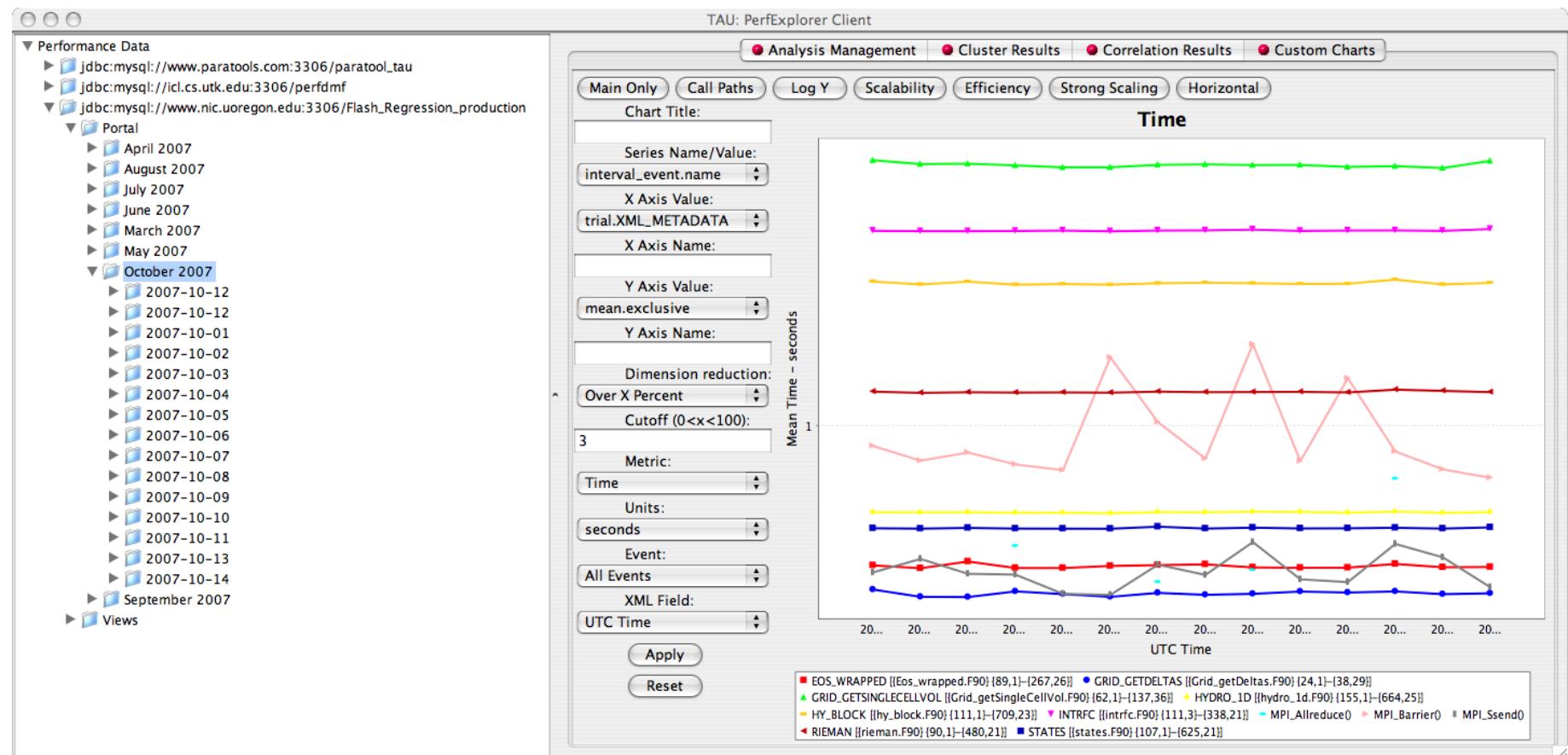


- Four significant events automatically selected
- Clusters and correlations are visible



# PerfExplorer – Performance Regression

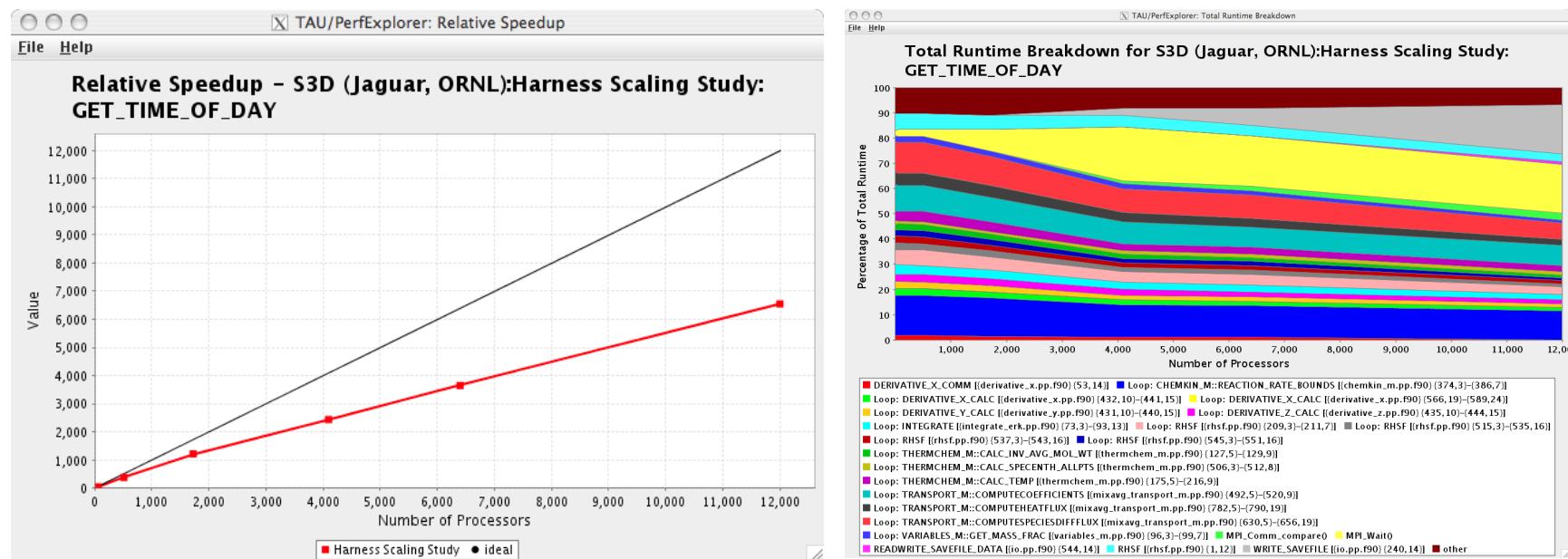
VI-HPS



# Usage Scenarios: Evaluate Scalability

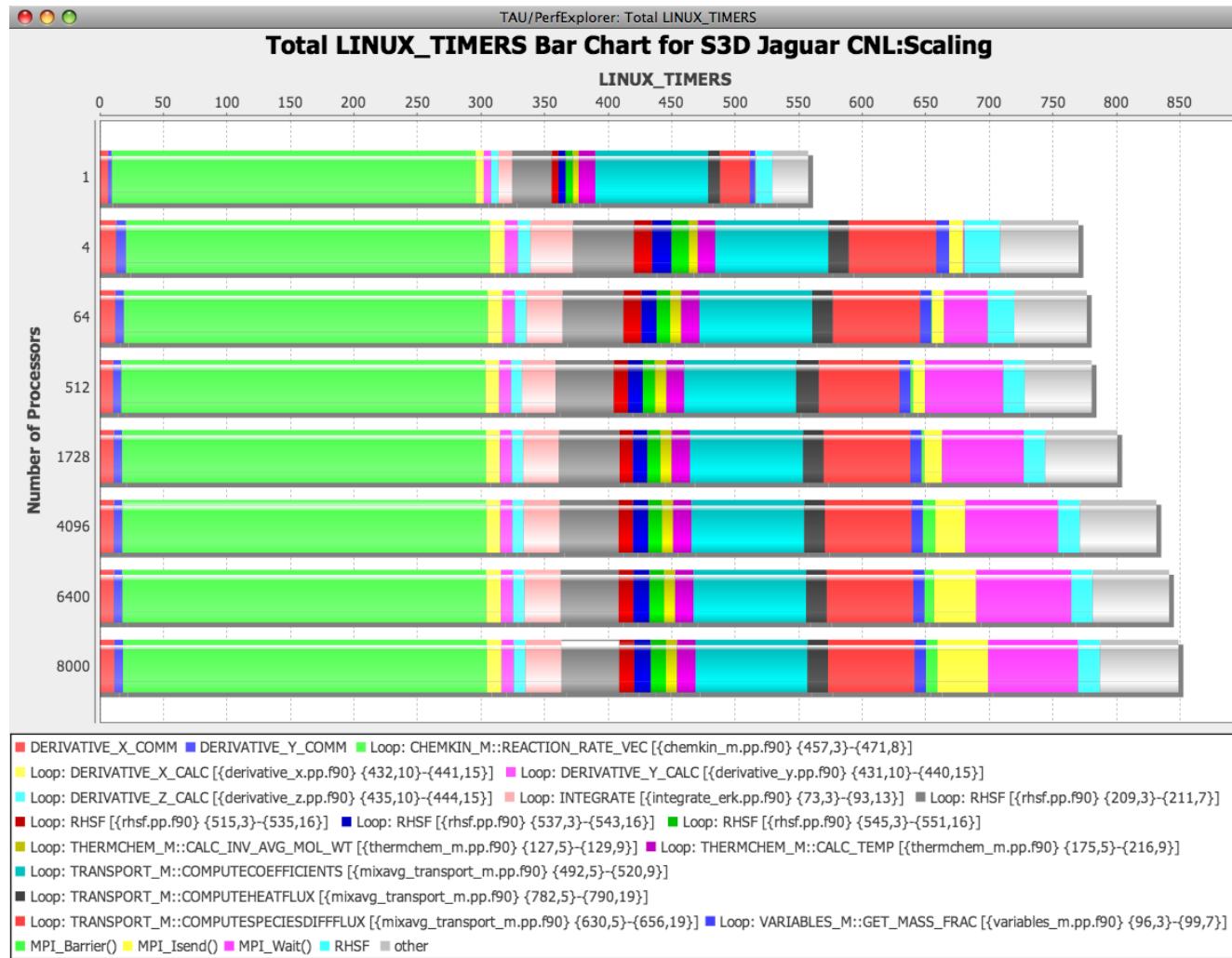
VI-HPS

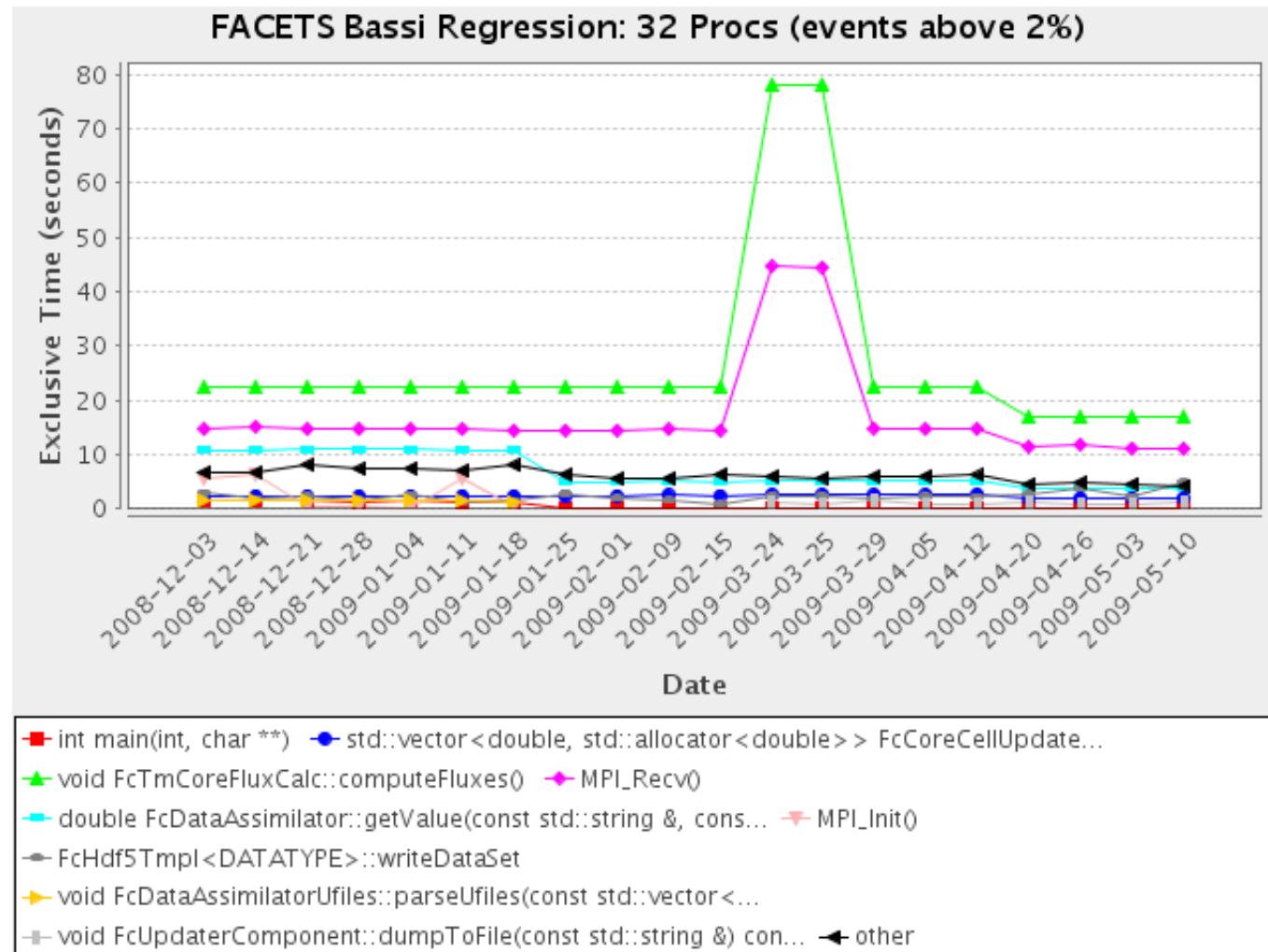
- Goal: How does my application scale? What bottlenecks at what CPU counts?
- Load profiles in PerfDMF database and examine with PerfExplorer



# Usage Scenarios: Evaluate Scalability

VI-HPS





```
% export TAU_MAKEFILE=$TAU_LIB_DIR/Makefile.tau-icpc-mpi-pdt
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% qsub run1p.job
% paraprof --pack 1p.ppk
% qsub run2p.job ...
% paraprof --pack 2p.ppk ... and so on.

On your client:
% perfdfmf_configure
(Choose derby, blank user/passwd, yes to save passwd, defaults)
% perfexplorer_configure
(Yes to load schema, defaults)
% paraprof
(load each trial: DB -> Add Trial -> Type (Paraprof Packed
Profile) -> OK, OR use perfdfmf_loadtrial on the commandline)
% perfexplorer
(Charts -> Speedup)
```

- TAU Portal
  - Support collaborative performance study
- Kernel-level system measurements (KTAU)
  - Application to OS noise analysis and I/O system analysis
- TAU performance monitoring
  - TAUoverSupermon and TAUoverMRNet
- PerfExplorer integration and expert-based analysis
  - OpenUH compiler optimizations
  - Computational quality of service in CCA
- Eclipse CDT and PTP integration
- Performance tools integration (NSF POINT project)

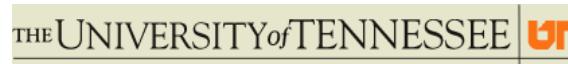
## Support Acknowledgements

- US Department of Energy (DOE)
  - Office of Science contracts
  - SciDAC, LBL contracts
  - LLNL-LANL-SNL ASC/NNSA contract
  - Battelle, PNNL contract
  - ANL, ORNL contract
- Department of Defense (DoD)
  - PETTT, HPTi
- National Science Foundation (NSF)
  - SDCI, SI-2
- University of Oregon
- ParaTools, Inc.
- University of Tennessee, Knoxville
  - Dr. Shirley Moore
- T.U. Dresden, GWT
  - Dr. Wolfgang Nagel and Dr. Andreas Knupfer
- Juelich Supercomputing Center
  - Dr. Bernd Mohr, Dr. Felix Wolf
  - Dr. Markus Geimer, Dr. Brian Wylie

9th VI-HPS Tuning Workshop, U. Versailles



Pacific Northwest  
NATIONAL LABORATORY



**ParaTools**

- TAU Website:  
<http://tau.uoregon.edu>
  - Software
  - Release notes
  - Documentation