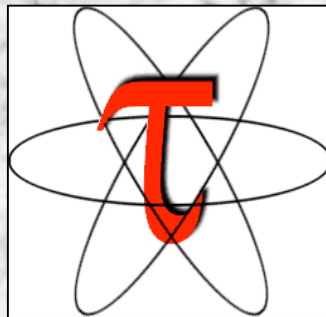


TAU Parallel Performance System

PDC Summer School in HPC



Part 2: TAU Components and Usage



Tutorial Outline – Part 2



TAU Components and Usage

- ❑ Configuration
- ❑ Instrumentation
 - Source, library, dynamic, multi-level
- ❑ Measurement
 - Profiling and tracing
- ❑ Analysis
 - ParaProf
 - Vampir
- ❑ Examples of use

Using TAU for Performance Experimentation



- ❑ Instrumentation
 - Application code and libraries
 - Selective instrumentation
- ❑ Install, compile, and link with TAU measurement library
 - Configure TAU system
 - Multiple configurations for different measurements options
 - Does not require change in instrumentation – just relink
 - Selective measurement control
- ❑ Execute “experiments” to produce performance data
 - Performance data generated at end or during execution
- ❑ Use analysis tools to look at performance results



Using TAU in Practice

- ❑ **Install TAU**
 - % configure ; make clean install
- ❑ **Instrument application**
 - TAU Profiling API
- ❑ **Typically modify application makefile**
 - Include TAU's stub makefile, modify variables
- ❑ **Set environment variables**
 - Directory where profiles/traces are to be stored
- ❑ **Execute application**
 - % mpirun -np <procs> a.out
- ❑ **Analyze performance data**
 - paraprof, vampir, pprof, paraver ...



Instrumentation Alternatives

- ❑ Manual instrumentation at the source
 - Use TAU API appropriate for source language
- ❑ Automatic source-level instrumentation
 - Source rewriting
 - Directive rewriting (e.g., for OpenMP)
- ❑ Library instrumentation
 - Typically done at source level
 - Wrapper interposition library (e.g., PMPI)
- ❑ Binary Instrumentation
 - Pre-execution or runtime binary rewriting
- ❑ Dynamic runtime instrumentation



TAU Measurement API for C/C++

- ❑ Initialization and runtime configuration
 - TAU_PROFILE_INIT(*argc*, *argv*);
TAU_PROFILE_SET_NODE(*myNode*);
TAU_PROFILE_SET_CONTEXT(*myContext*);
TAU_PROFILE_EXIT(*message*);
TAU_REGISTER_THREAD();
- ❑ Function and class methods
 - TAU_PROFILE(*name*, *type*, *group*);
- ❑ Template
 - TAU_TYPE_STRING(*variable*, *type*);
TAU_PROFILE(*name*, *type*, *group*);
CT(*variable*);
- ❑ User-defined timing
 - TAU_PROFILE_TIMER(*timer*, *name*, *type*, *group*);
TAU_PROFILE_START(*timer*);
TAU_PROFILE_STOP(*timer*);

TAU Measurement API for C/C++ (continued)



□ User-defined events

- TAU_REGISTER_EVENT(variable, event_name);
TAU_EVENT(variable, value);
TAU_PROFILE_STMT(statement);

□ Mapping

- TAU_MAPPING(statement, key);
TAU_MAPPING_OBJECT(funcIdVar);
TAU_MAPPING_LINK(funcIdVar, key);
- TAU_MAPPING_PROFILE (funcIdVar);
TAU_MAPPING_PROFILE_TIMER(timer, funcIdVar);
TAU_MAPPING_PROFILE_START(timer);
TAU_MAPPING_PROFILE_STOP(timer);

□ Reporting

- TAU_REPORT_STATISTICS();
TAU_REPORT_THREAD_STATISTICS();



Example: Manual Instrumentation (C++)

```
#include <TAU.h>
int main(int argc, char **argv)
{
    TAU_PROFILE("int main(int, char **)", " ", TAU_DEFAULT); /* name,type,group */
    TAU_PROFILE_INIT(argc, argv);
    TAU_PROFILE_SET_NODE(0); /* for sequential programs */
    foo();
    return 0;
}
int foo(void)
{
    TAU_PROFILE("int foo(void)", " ", TAU_DEFAULT); // measures entire foo()
    TAU_PROFILE_TIMER(t, "foo(): for loop", "[23:45 file.cpp]", TAU_USER);
    TAU_PROFILE_START(t);
    for(int i = 0; i < N ; i++){
        work(i);
    }
    TAU_PROFILE_STOP(t);
    // other statements in foo ...
}
```

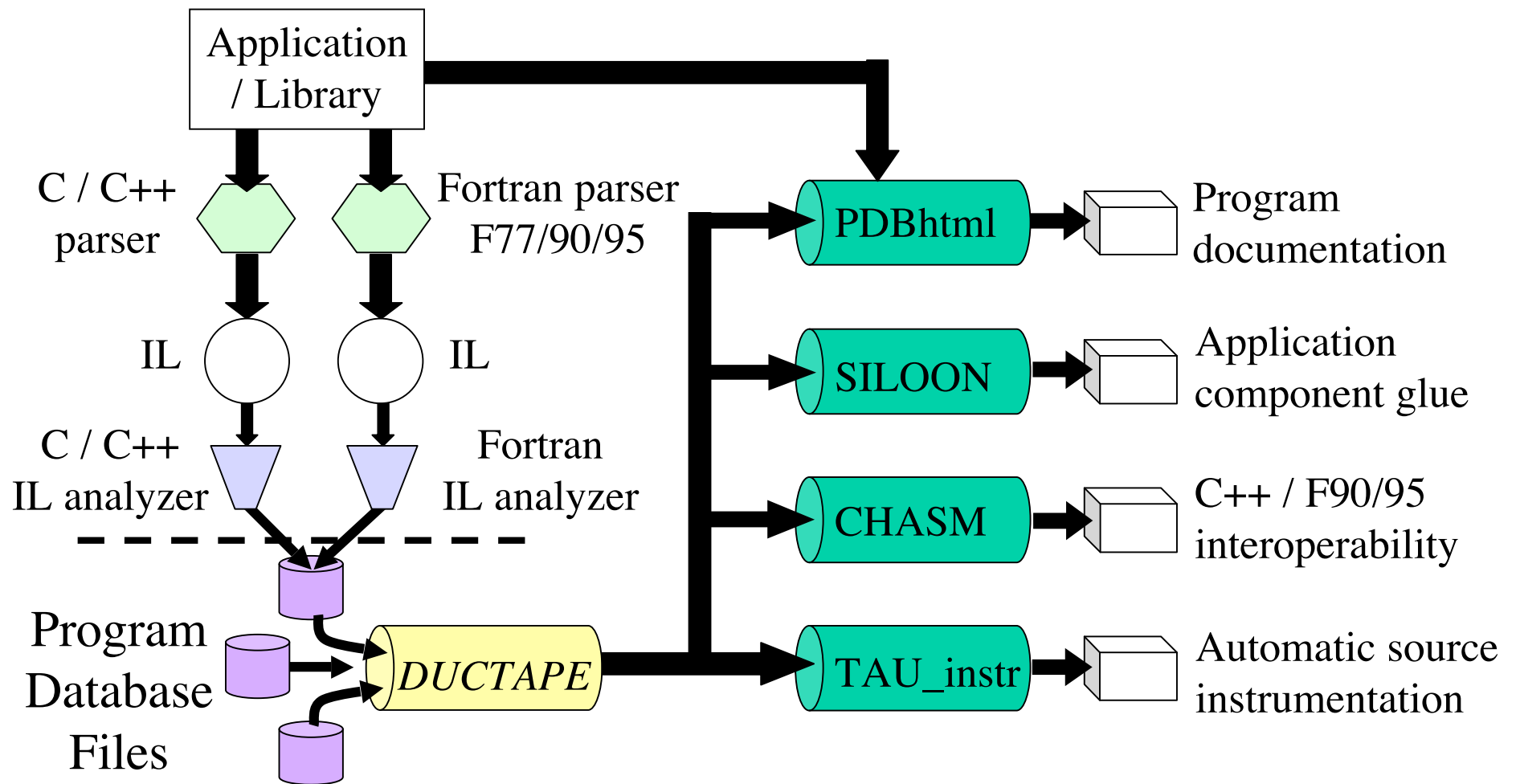



Program Database Toolkit (PDT)

- ❑ Program code analysis framework
 - develop source-based analysis tools
- ❑ *High-level interface* to source code information
- ❑ *Integrated toolkit* for source code parsing, database creation, and database query
 - Commercial grade front-end parsers
 - Portable IL analyzer, database format, and access API
 - Open software approach for tool development
- ❑ Multiple source languages
- ❑ Implement automatic performance instrumentation tools
 - *tau_instrumentor* for automatic source instrumentation



Program Database Toolkit (PDT)





PDT Components

□ Language front end

- Edison Design Group (EDG): C, C++, Java
- Mutek Solutions Ltd.: F77, F90
- Cleanscape FortranLint F95 parser/analyzer
- Creates an intermediate-language (IL) tree

□ IL Analyzer

- Processes the intermediate language (IL) tree
- Creates “program database” (PDB) formatted file

□ DUCTAPE (Bernd Mohr, ZAM, Germany)

- C++ program Database Utilities and Conversion Tools
Application Environment
- C++ library to process the PDB for PDT applications

□ Intel/KAI C++ headers for std. C++ library



Contents of PDB files

- ❑ Source file names
- ❑ Routines, Classes, Methods, Templates, Macros, Modules
- ❑ Parameters, signature
- ❑ Entry and exit point information (return)
- ❑ Location information for all of the above
- ❑ Static callgraph
- ❑ Header file inclusion tree
- ❑ Statement-level information
 - Loops, if-then-else, switch ...



PDT 3.2 Functionality

- ❑ C++ statement-level information implementation
 - for, while loops, declarations, initialization, assignment...
 - PDB records defined for most constructs
- ❑ DUCTAPE
 - Processes PDB 1.x, 2.x, 3.x uniformly
- ❑ PDT applications
 - XMLgen
 - PDB to XML converter
 - Used for CHASM and CCA tools
 - PDBstmt
 - Statement callgraph display tool



PDT 3.2 Functionality (continued)

- ❑ **Cleanscape Flint parser fully integrated for F90/95**
 - Flint parser (f95parse) is very robust
 - Produces PDB records for TAU instrumentation (stage 1)
 - Linux (x86, IA-64, Opteron, Power4), HP Tru64, IBM AIX, Cray X1, T3E, Solaris, SGI, Apple, Windows, Power4 Linux (IBM Blue Gene/L compatible)
 - Full PDB 2.0 specification (stage 2) [SC'04]
 - Statement level support (stage 3) [SC'04]
- ❑ **PDT 3.2 released in June 3, 2004**
 - Important bug fixes
- ❑ **<http://www.cs.uoregon.edu/research/paracomp/pdtoolkit>**



tau_instrumentor: A PDT Instrumentation Tool

```
% 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
```



Instrumentation Control

- ❑ Selection of which performance events to observe
 - Could depend on scope, type, level of interest
 - Could depend on instrumentation overhead
- ❑ How is selection supported in instrumentation system?
 - No choice
 - Include / exclude routine and file lists (TAU)
 - Environment variables
 - Static vs. dynamic
- ❑ Problem: Controlling instrumentation of small routines
 - High relative measurement overhead
 - Significant intrusion and possible perturbation



Example: tau_reduce

- ❑ *tau_reduce* implements overhead reduction in TAU
- ❑ Consider *klargest* example
 - Find *k*th largest element in a *N* elements
 - Compare two methods: *quicksort*, *select_kth_largest*
- ❑ *i = 2324, N = 1000000 (uninstrumented)*
 - *quicksort*: (wall clock) = 0.188511 secs
 - *select_kth_largest*: (wall clock) = 0.149594 secs
 - Total: (P3/1.2GHz, *time*) = 0.340u 0.020s 0:00.37
- ❑ Execution with all routines instrumented
- ❑ Execution with rule-based selective instrumentation
 - `usec>1000 & numcalls>400000 & usecs/call<30 & percent>25`



Reducing Instrumentation on One Processor

Before selective instrumentation reduction

NODE 0;CONTEXT 0;THREAD 0:

%Time	Exclusive msec	Inclusive msec	#Call	#Subrs	Inclusive Name usec/call
100.0	13	4,982	1	4	4982030 int main
93.5	3,223	4,659	4.20241E+06	1.40268E+07	1 void quicksort
62.9	0.00481	3,134	5	5	626839 int kth_largest_qs
36.4	137	1,813	28	450057	64769 int select_kth_largest
33.6	150	1,675	449978	449978	4 void sort_5elements
28.8	1,435	1,435	1.02744E+07	0	0 void interchange
0.4	20	20	1	0	20668 void setup
0.0	0.0118	0.0118	49	0	0 int ceil

After selective instrumentation reduction

NODE 0;CONTEXT 0;THREAD 0:

%Time	Exclusive msec	Inclusive total msec	#Call	#Subrs	Inclusive Name usec/call
100.0	14	383	1	4	383333 int main
50.9	195	195	5	0	39017 int kth_largest_qs
40.0	153	153	28	79	5478 int select_kth_largest
5.4	20	20	1	0	20611 void setup
0.0	0.02	0.02	49	0	0 int ceil



TAU's MPI Wrapper Interposition Library

- ❑ Uses standard MPI Profiling Interface
 - Provides name shifted interface
 - *MPI_Send* \Leftrightarrow *PMPI_Send*
 - weak bindings
- ❑ Instrument MPI wrapper library
 - Use TAU measurement API
- ❑ Interpose TAU's MPI wrapper library
 - Replace **-lmpi** by **-lTauMpi -lpmpi -lmpi**
- ❑ No change to the source code!
 - Just **re-link** the application to generate performance data



MPI Library Instrumentation (MPI_Send)

```
int  MPI_Send(...) /* TAU redefines MPI_Send */
...
{
  int  returnVal, typesize;
  TAU_PROFILE_TIMER(tautimer, "MPI_Send()", " ", TAU_MESSAGE);
  TAU_PROFILE_START(tautimer);
  if (dest != MPI_PROC_NULL) {
    PMPI_Type_size(datatype, &typesize);
    TAU_TRACE_SENDMSG(tag, dest, typesize*count);
  }
  /* Wrapper calls PMPI_Send */
  returnVal = PMPI_Send(buf, count, datatype, dest, tag,
comm);
  TAU_PROFILE_STOP(tautimer);
  return returnVal;
}
```



Instrumentation of OpenMP Constructs



- ❑ **OpenMP Pragma And Region Instrumentor**
- ❑ Source-to-Source translator to insert *POMP* calls around OpenMP constructs and API functions
- ❑ **Supports**
 - Fortran77 and Fortran90, OpenMP 2.0
 - C and C++, OpenMP 1.0
 - *POMP* Extensions
 - Preserves source information (**#line line file**)
 - Measurement library implementations
 - EPILOG, TAU POMP, DPOMP (IBM)
- ❑ **Work in Progress**
 - Investigating standardization through OpenMP Forum



POMP OpenMP Performance Tool Interface

□ OpenMP Instrumentation

- OpenMP Directive Instrumentation
- OpenMP Runtime Library Routine Instrumentation

□ POMP Extensions

- Runtime Library Control (**init**, **finalize**, **on**, **off**)
- (Manual) User Code Instrumentation (**begin**, **end**)
- Conditional Compilation (**#ifdef _POMP, !\$P**)
- Conditional / Selective Transformations
(**[no]instrument**)



Example: !\$OMP PARALLEL DO

```
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)
```



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: Opari 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"
```



Example: TAU POMP Implementation

```
TAU_GLOBAL_TIMER(tfor, "for enter/exit",
                 "[OpenMP]", OpenMP);

void pomp_for_enter(OMPRegDescr* r) {
    #ifdef TAU_AGGREGATE_OPENMP_TIMINGS
        TAU_GLOBAL_TIMER_START(tfor)
    #endif
    #ifdef TAU_OPENMP_REGION_VIEW
        TauStartOpenMPRegionTimer(r);
    #endif
}

void pomp_for_exit(OMPRegDescr* r) {
    #ifdef TAU_AGGREGATE_OPENMP_TIMINGS
        TAU_GLOBAL_TIMER_STOP(tfor)
    #endif
    #ifdef TAU_OPENMP_REGION_VIEW
        TauStopOpenMPRegionTimer(r);
    #endif
}
```



OPARI: Basic Usage (F90)

- ❑ Reset OPARI state information
 - `rm -f opari.rc`
- ❑ Call OPARI for each input source file
 - `opari file1.f90`
 - ...
 - `opari fileN.f90`
- ❑ Generate OPARI runtime table, compile it with ANSI C
 - `opari -table opari.tab.c`
 - `cc -c opari.tab.c`
- ❑ Compile modified files `*.mod.f90` using OpenMP
- ❑ Link the resulting object files, the OPARI runtime table `opari.tab.o` and the TAU POMP RTL



Dynamic Instrumentation

- ❑ TAU uses DyninstAPI for runtime code patching
- ❑ *tau_run* (mutator) loads measurement library
- ❑ Instruments mutatee
 - Application binary
 - Uses TAU-developed instrumentation specification
- ❑ MPI issues
 - One mutator per executable image [TAU, DynaProf]
 - One mutator for several executables [Paradyn, DPCL]



Using DyninstAPI with TAU

Step I: Install DyninstAPI[Download from <http://www.dyninst.org>]

```
% cd dyninstAPI-4.0.2/core; make
```

Set DyninstAPI environment variables (including LD_LIBRARY_PATH)

Step II: Configure TAU with Dyninst

```
% configure -dyninst=/usr/local/dyninstAPI-4.0.2
```

```
% make clean; make install
```

Builds <taudir>/<arch>/bin/tau_run

```
% tau_run [<-o outfile>] [-Xrun<libname>]  
  [-f <select_inst_file>] [-v] <infile>
```

```
% tau_run -o a.inst.out a.out
```

Rewrites a.out

```
% tau_run klargest
```

Instruments klargest with TAU calls and executes it

```
% tau_run -XrunTAUsh-papi a.out
```

Loads libTAUsh-papi.so instead of libTAU.so for measurements

NOTE: All compilers and platforms are not yet supported (work in progress)

TAU Measurement



□ Performance information

- High-resolution *timer library* (real-time / virtual clocks)
- General *software counter library* (user-defined events)
- Hardware performance counters
 - *PAPI* (Performance API) (UTK, Ptools Consortium)
 - consistent, portable API

□ Measurement types

- Parallel profiling
 - includes multiple counters, callpaths, performance mapping
- Parallel tracing

□ Support for online performance data access



Multi-Threading Performance Measurement

□ General issues

- Thread identity and per-thread data storage
- Performance measurement support and synchronization
- Fine-grained parallelism
 - different forms and levels of threading
 - greater need for efficient instrumentation

□ TAU general threading and measurement model

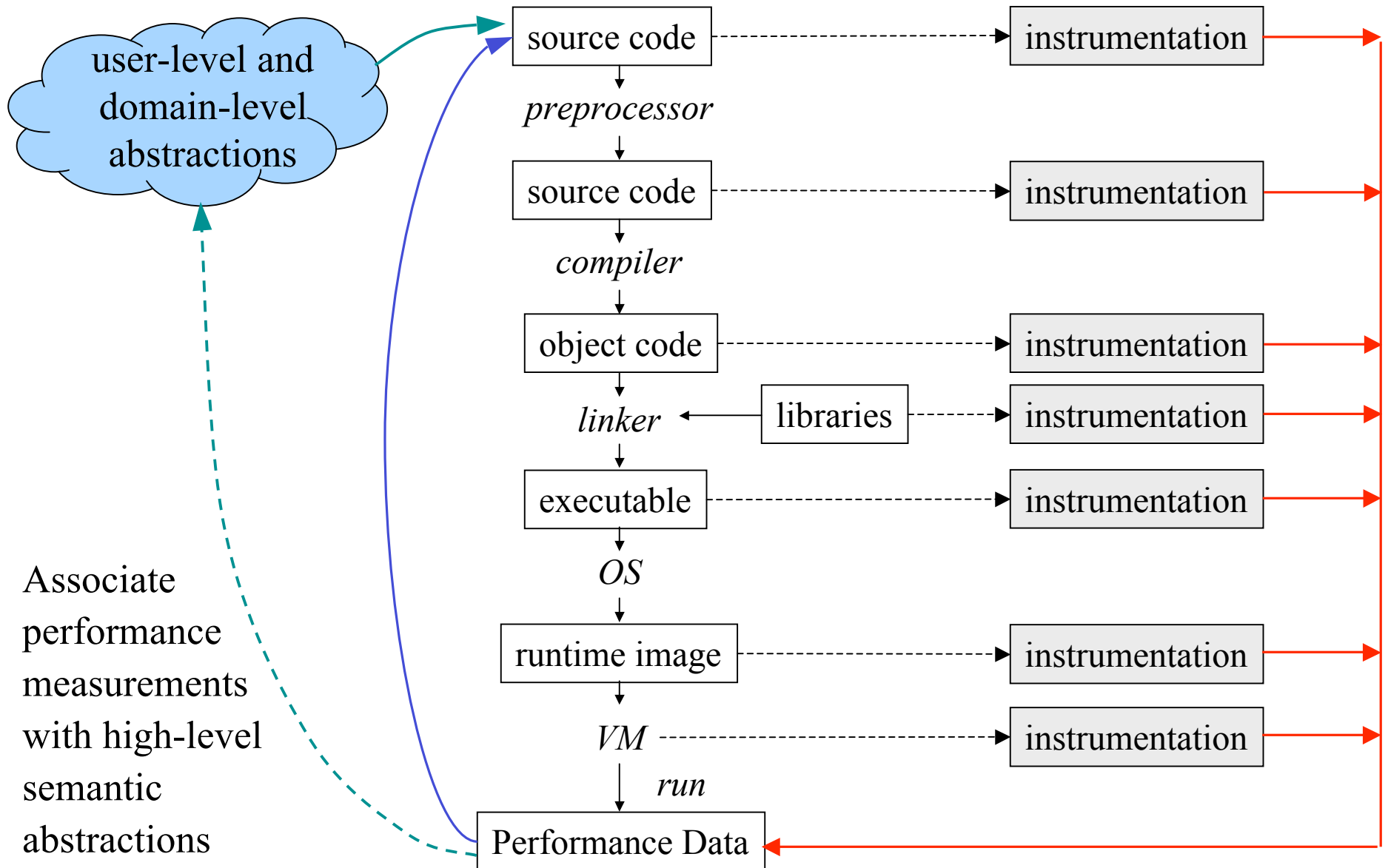
- Common thread layer and measurement support
- Interface to system specific libraries (reg, id, sync)

□ Target different thread systems with core functionality

- Pthreads, Windows, Java, OpenMP



Semantic Performance Mapping



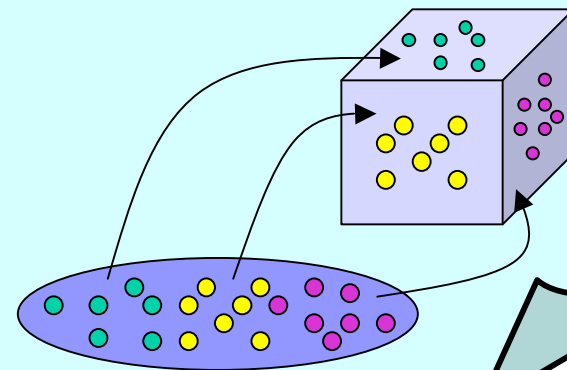
Associate performance measurements with high-level semantic abstractions



Hypothetical Mapping Example

- Particles distributed on surfaces of a cube

```
Particle* P[MAX]; /* Array of particles */
int GenerateParticles() {
    /* distribute particles over all faces of the cube */
    for (int face=0, last=0; face < 6; face++){
        /* particles on this face */
        int particles_on_this_face = num(face);
        for (int i=last; i < particles_on_this_face; i++) {
            /* particle properties are a function of face */
            P[i] = ... f(face);
            ...
        }
        last+= particles_on_this_face;
    }
}
```





Hypothetical Mapping Example (continued)

```
int ProcessParticle(Particle *p) {
    /* perform some computation on p */
}
int main() {
    GenerateParticles();
    /* create a list of particles */
    for (int i = 0; i < N; i++)
        /* iterates over the list */
        ProcessParticle(P[i]);
}
```

- ❑ How much time is spent processing *face i* particles?
- ❑ What is the distribution of performance among faces?
- ❑ How is this determined if execution is parallel?

Semantic Entities/Attributes/Associations (SEAA)



- New dynamic mapping scheme
 - Entities defined at any level of abstraction
 - Attribute entity with semantic information
 - Entity-to-entity associations
- Two association types (implemented in TAU API)
 - Embedded
 - External
- “The Role of Performance Mapping”
 - Dr. Sameer Shende
 - Ph.D. thesis

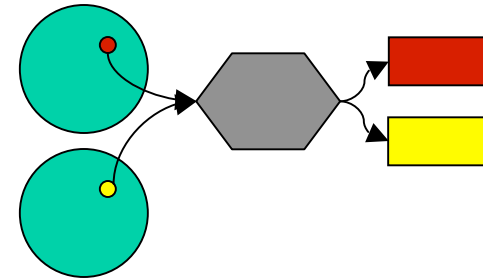


Mapping Associations

□ Embedded association

○ Embedded

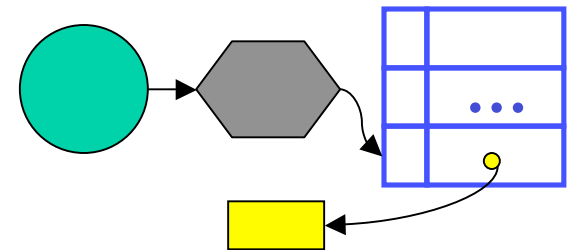
extends associated object to store performance measurement entity



□ External association

○ External

creates an external look-up table using address of object as key to locate performance measurement entity

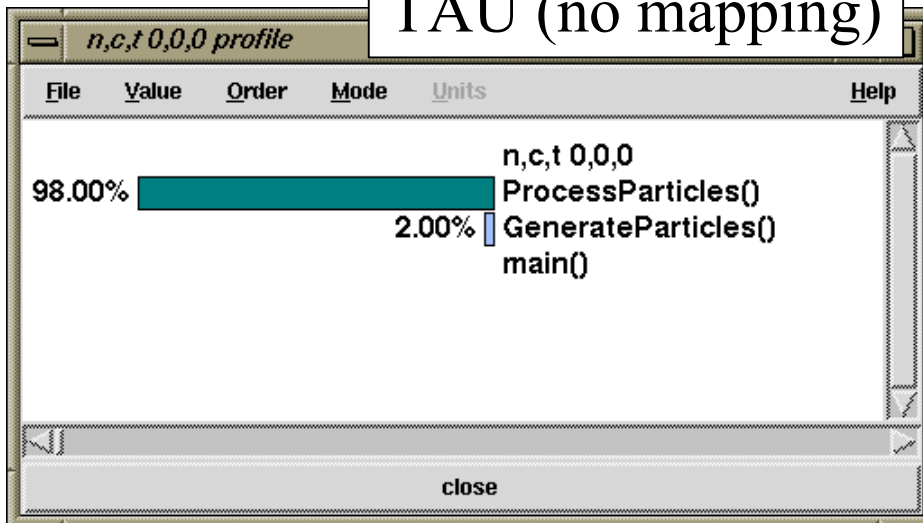




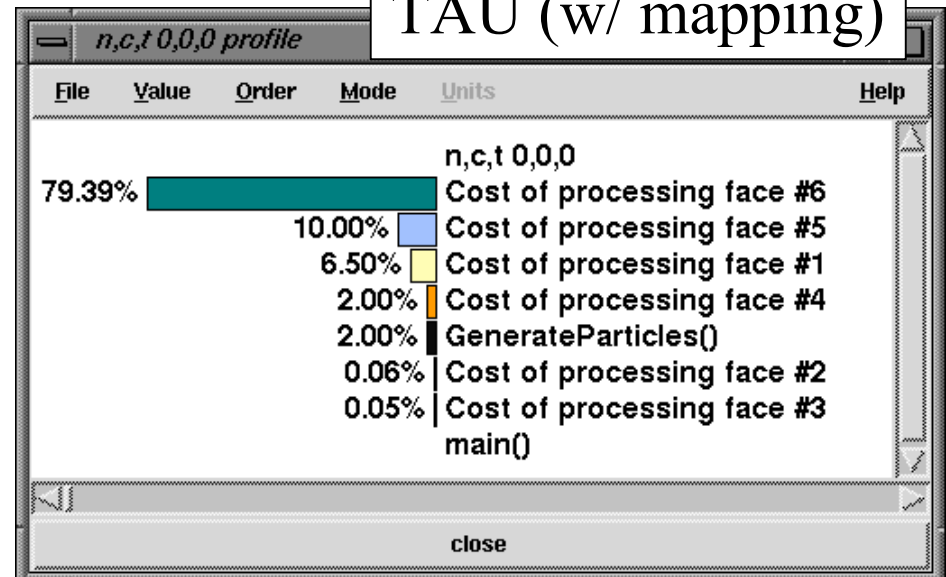
No Performance Mapping versus Mapping

- ❑ Typical performance tools report performance with respect to routines
- ❑ Does not provide support for mapping
- ❑ Performance tools with SEAA mapping can observe performance with respect to scientist's programming and problem abstractions

TAU (no mapping)



TAU (w/ mapping)





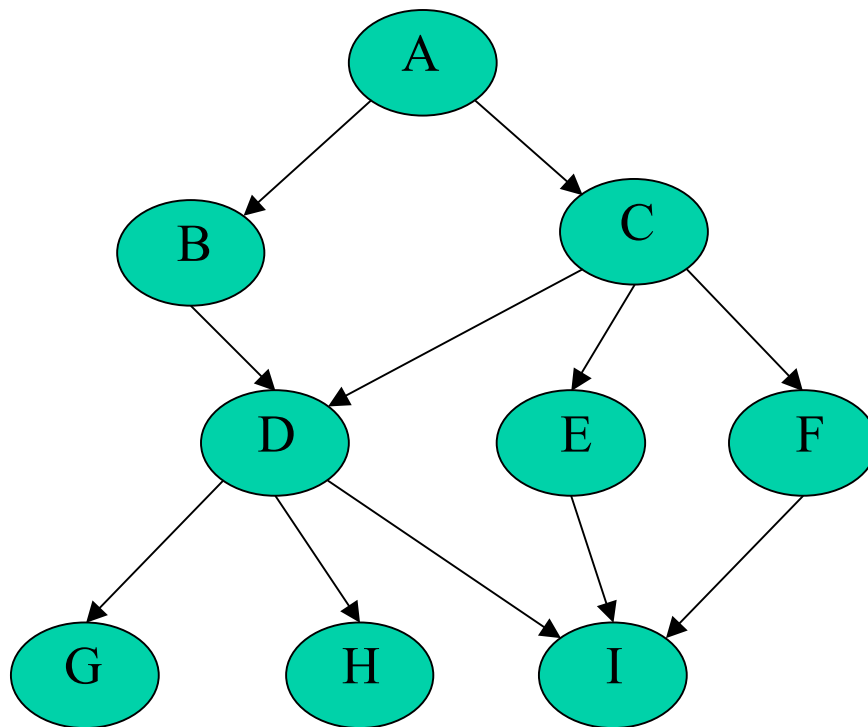
Performance Mapping and Callpath Profiling

- ❑ Associate performance with “significant” entities (events)
- ❑ Source code points are important
 - Functions, regions, control flow events, user events
- ❑ Execution process and thread entities are important
- ❑ Some entities are more abstract, harder to measure
- ❑ Consider callgraph (callpath) profiling
 - Measure time (metric) along an edge (path) of callgraph
 - Incident edge gives parent / child view
 - Edge sequence (path) gives parent / descendant view
- ❑ Problem: Callpath profiling when callgraph is unknown
 - Determine callgraph dynamically at runtime
 - Map performance measurement to dynamic call path state



Callgraph (Callpath) Profiling

- Measure time (metric) along an edge (path) of callgraph
 - Incident edge gives parent / child view
 - Edge sequence (path) gives parent / descendant view



- 1-level callpath
 - Immediate descendant
 - $A \rightarrow B$, $E \rightarrow I$, $D \rightarrow H$
 - $C \rightarrow H$?
- k -level callpath
 - k call descendant
 - 2-level: $A \rightarrow D$, $C \rightarrow I$
 - 2-level: $A \rightarrow I$?
 - 3-level: $A \rightarrow H$



k-Level Callpath Implementation in TAU

- ❑ TAU maintains a performance event (routine) callstack
- ❑ Profiled routine (child) looks in callstack for parent
 - Previous profiled performance event is the parent
 - A *callpath profile structure* created first time parent calls
 - TAU records parent in a *callgraph map* for child
 - String representing k-level callpath used as its key
 - “a()=>b()=>c()” : name for time spent in “c” when called by “b” when “b” is called by “a”
- ❑ Map returns pointer to callpath profile structure
 - k-level callpath is profiled using this profiling data
 - Set environment variable **TAU_CALLPATH_DEPTH** to depth
- ❑ Build upon TAU’s performance mapping technology
- ❑ Measurement is independent of instrumentation
- ❑ Use `-PROFILECALLPATH` to configure TAU



Running Applications

```
% set path=($path <taudir>/<arch>/bin)
% set path=($path $PET_HOME/PTOOLS/tau-2.13.5/src/rs6000/bin)
% setenv LD_LIBRARY_PATH
$LD_LIBRARY_PATH\:<taudir>/<arch>/lib
```

For PAPI (1 counter, if multiplecounters is not used):

```
% setenv PAPI_EVENT PAPI_L1_DCM (Level 1 Data cache misses)
```

For PAPI (multiplecounters):

```
% setenv COUNTER1 PAPI_FP_INS (Floating point
instructions)
```

```
% setenv COUNTER2 PAPI_TOT_CYC (Total cycles)
```

```
% setenv COUNTER3 P_VIRTUAL_TIME (Virtual time)
```

```
% setenv COUNTER4 LINUX_TIMERS (Wallclock time)
```

(NOTE: PAPI_FP_INS and PAPI_L1_DCM cannot be used together on Power4. Other restrictions may apply to no. of counters used.)

```
% mpirun -np <n> <application>
```

```
% llsubmit job.sh
```



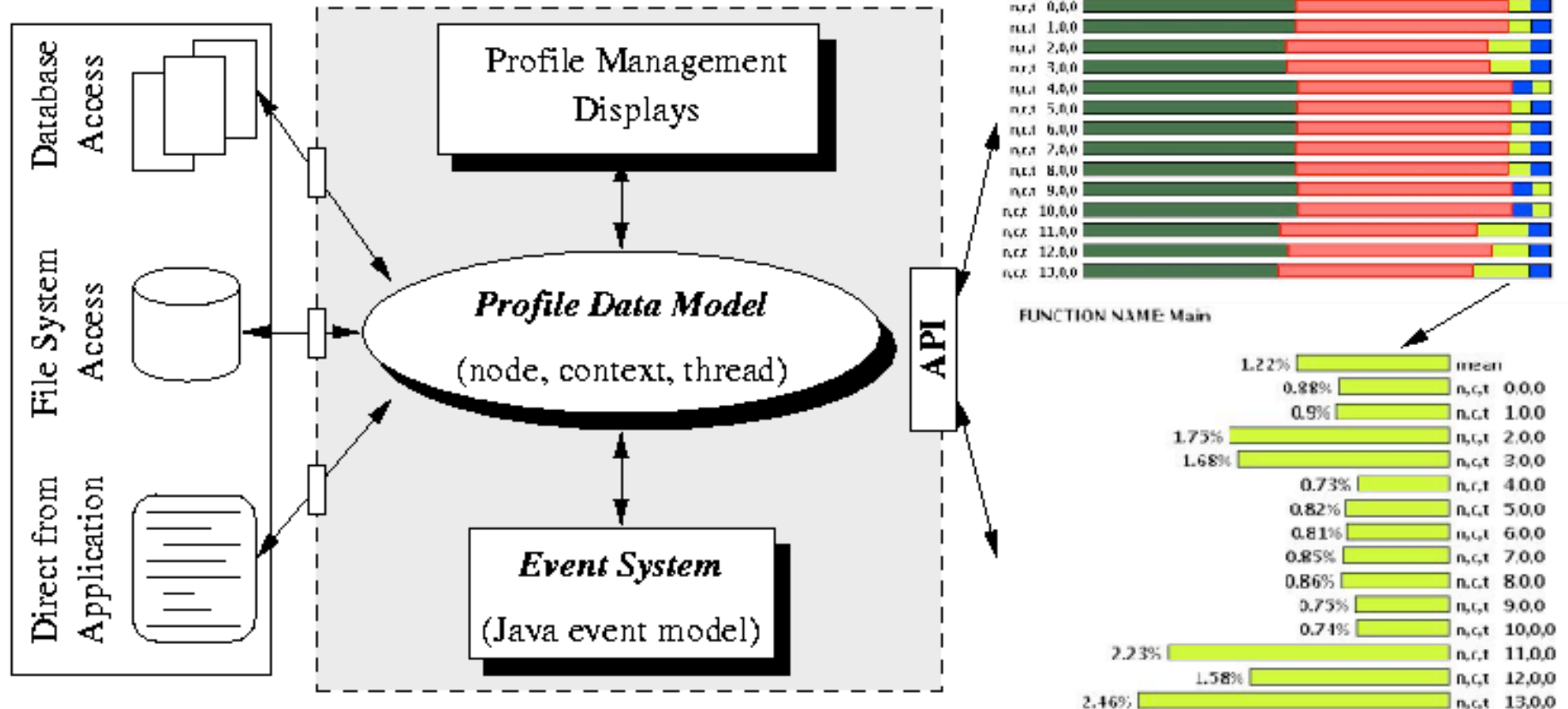
TAU Performance Analysis

- ❑ Analysis of parallel profile and trace measurement
- ❑ Parallel profile analysis
 - *Pprof*: parallel profiler with text-based display
 - *ParaProf*: graphical, scalable parallel profile analysis
- ❑ Parallel trace analysis
 - Format conversion (ALOG, VTF 3.0, Paraver, EPILOG)
 - Trace visualization using *Vampir* (Pallas/Intel)
 - Parallel profile generation from trace data



ParaProf Framework Architecture

- Portable, extensible, and scalable tool for profile analysis
- Try to offer “best of breed” capabilities to analysts
- Build as profile analysis framework for extensibility





ParaProf Manager

The screenshot shows the ParaProf Manager application window. The title bar reads "ParaProf Manager". The menu bar contains "File", "Options", and "Help". The left pane shows a tree view under "Applications" with a context menu open over "Default Exp". The right pane shows a table of fields and their values.

Field	Value
Name	Default Exp
Application ID	0
Experiment ID	0
User Data	
System Name	
System Machine Type	
System Arch.	
System OS	
System Memory Size	
System Processor Amount	
System L1 Cache Size	
System L2 Cache Size	
System User Data	
Configuration Prefix	
Configuration Architecture	
Configuration CPP	
Configuration CC	
Configuration JDK	
Configuration Profile	
Configuration User Data	
Compiler CPP Name	
Compiler CPP Version	
Compiler CC Name	
Compiler CC Version	
Compiler Java Dir. Path	
Compiler Java Version	
Compiler User Data	

- ❑ Powerful manager for control of data sources
 - Directly from files
 - Profile database
 - Runtime (online)
- ❑ Conveniences to facilitate working with data



ParaProf Manager (continued)

- Data management windows
 - Loading flat files from disk
 - Generating new derived metrics
 - Database interface

The screenshot displays the ParaProf Manager application window. The main interface is divided into several sections:

- Applications Panel:** A tree view on the left showing a hierarchy: Applications > Star > Add Trial > Default Exp > Default Trial. Under Default Trial, there are four items with green status indicators: CPU_TIME, GET_TIME_OF_DAY, packed_DP_uop_all, and packed_DP_uop_all / GET_TIME_OF_DAY. The last item is selected.
- Table:** A table on the right with columns 'Field' and 'Value'. It contains the following data:

Field	Value
Name	packed_DP_uop_all / GET_T...
Application ID	0
Experiment ID	0
Trial ID	0
Metric ID	3
- Arguments:** Two input fields for 'Argument 1' (0:0:0:3) and 'Argument 2' (0:0:0:2), with a 'Divide' dropdown menu and an 'Apply operation' button.
- Database Configuration Window:** A separate window with a 'Password' field (masked with asterisks), a 'Config File' field (./bertie/Robert/Code/Data/multiplecount), and 'Cancel' and 'Ok' buttons.
- Load Trial Window:** A separate window with a 'Trial Type' dropdown (pprof.dat), a 'Dir. Location' field (bertie/Robert/Code/Data/multiplecount), and 'Cancel' and 'Ok' buttons.

Red arrows indicate the flow of information: from the selected application in the tree to the 'Load Trial' window, from the 'Database Configuration' window to the 'DB Applications' folder, and from the 'Trial information' table to the 'Load Trial' window.



ParaProf Derived Metrics

The screenshot shows the ParaProf Manager application window. The title bar reads "ParaProf Manager". The menu bar contains "File" and "Help".

The left pane is a tree view with the following structure:

- Standard Applications
 - Default App
 - Experiments
 - Default Exp
 - Trials
 - Default Trial : 512proc/samrai/taudata/neutron
 - 0000 - P_WALL_CLOCK_TIME
 - 0001 - PAPI_FP_INS
 - 0002 - PAPI_FP_INS / P_WALL_CLOCK_TIME

Below the tree view are sections for "Runtime Applications" and "DB Applications".

The right pane contains the following text:

ParaProf Manager

Clicking on different values causes ParaProf to display the clicked on metric.

The sub-window below allow you to generate new metrics based on those that were gathered during the run. The operand number options for Operand A and B correspond the numbers prefixing the values.

The bottom section is titled "Apply operations here!" and contains the following fields:

- Op A: 0001 - PAPI_FP_INS
- Op B: 0000 - P_WALL_CLOCK_TIME
- Operation: Divide
- Apply Operation button



ParaProf Profile Analysis Displays

textual profile

%Time	Time	total Time	#calls	#subrs	total Time
100.0	8.757522992004E8	1.1717499847999E9	1.0	253.0	1.17174998
21.8	2.553087943998E8	2.553087943998E8			
0.9	1.0481863198E7	1.0481863198E7			
2.7	5075165.6002	3.16434583997E7			
0.4	5025481.6	5025481.6			
0.3	4027989.5978	4027989.5978			
0.2	2935568.0	2935568.0			
0.2	2130867.201	2130867.201			
0.1	1642712.0005	1642712.0005			
0.1	1632580.8013	1632580.8013			

legend

- main() void (int, char **)
- MPI_Reduce()
- MPI_Waitsome()
- MPI_Scheduler::execute()
- MPI_Init_thread()
- MPI_Allreduce()
- MPI_Barrier()
- MPI_Type_indexed()

full profile with display adjustment

Bar Multiple: 0 5 10 15 20 25 30 35 40

thread display

Metric Name	Value
main	74.74%
MPI_Reduce	21.79%
MPI_Init_thread	0.89%
MPI_Waitsome	0.43%
MPI_Allreduce	0.43%
MPI_Barrier	0.34%
MPI_Type_indexed	0.25%
MPI_Type_indexed	0.18%
SerialMPM::comp	0.14%
MPI_Probe	0.14%

event display

Metric Name	Value
mean	72.66%
n,c,t 223,0,0	78.19%
n,c,t 240,0,0	77.62%
n,c,t 264,0,0	77.46%



ParaProf User Event Display (MPI message size)

Message size sent to nodes legend:

- Message size sent to all nodes
- Message size sent to node 1
- Message size sent to node 10
- Message size sent to node 11
- Message size sent to node 12
- Message size sent to node 13
- Message size sent to node 14
- Message size sent to node 15
- Message size sent to node 16
- Message size sent to node 17
- Message size sent to node 18
- Message size sent to node 19
- Message size sent to node 2
- Message size sent to node 20
- Message size sent to node 21
- Message size sent to node 22

Menu Options:

- Show Function Ledger
- Show Group Ledger
- Show User Event Ledger
- Show Call Path Relations
- Close All Sub-Windows

Bar Chart Data (Metric Name: GET_TIME_OF_DAY):

Node	Message Size
n,c,t 0,0,0	mean
n,c,t 1,0,0	Message size sent to all nodes
n,c,t 2,0,0	Message size sent to node 1
n,c,t 3,0,0	Message size sent to node 10
n,c,t 4,0,0	Message size sent to node 11
n,c,t 5,0,0	Message size sent to node 12
n,c,t 6,0,0	Message size sent to node 13
n,c,t 7,0,0	Message size sent to node 14
n,c,t 8,0,0	Message size sent to node 15
n,c,t 9,0,0	Message size sent to node 16
n,c,t 10,0,0	Message size sent to node 17
n,c,t 11,0,0	Message size sent to node 18
n,c,t 12,0,0	Message size sent to node 19
n,c,t 13,0,0	Message size sent to node 2
n,c,t 14,0,0	Message size sent to node 20
n,c,t 15,0,0	Message size sent to node 21
n,c,t 16,0,0	Message size sent to node 22

Table 1: GET_TIME_OF_DAY (Sorted By: exclusive, Units: microseconds)

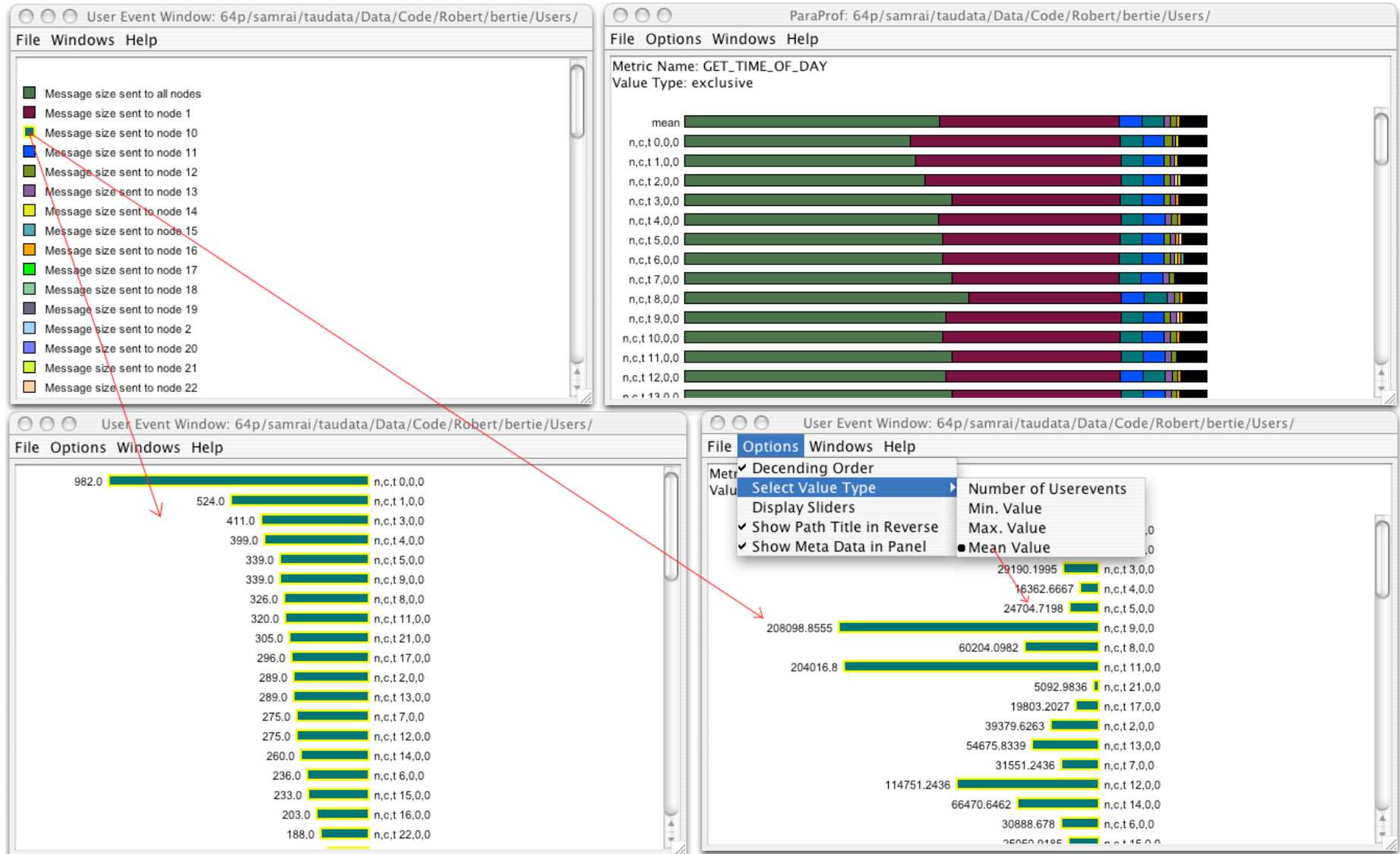
NumSamples	MaxValue	MinValue	MeanValue	name
11233	2473872.0	4.0	36610.5785	Message size se
486	2473872.0	160.0	154443.144	Message size se
474	2415360.0	40.0	149059.7468	Message size se
490	2373456.0	32.0	56988.6367	Message size se
530	727056.0	120.0	45053.7962	Message size se
169	675840.0	64.0	34364.355	Message size se
100	636480.0	4.0	27129.52	Message size se
785	568320.0	4.0	14094.6955	Message size se
116	544320.0	48.0	44834.2069	Message size se
848	519680.0	24.0	20789.4245	Message size se
443	517440.0	240.0	69900.1716	Message size se
94	335360.0	2560.0	79705.1915	Message size se

Table 2: GET_TIME_OF_DAY (Sorted By: exclusive, Units: microseconds)

	total Time	#calls	#subrs	total Time/call	name
912E8	8.63587912E8	193.0	0.0	4474549.0	algs::Hyp
726E8	5.96311726E8	39239.0	0.0	15197.0	MPI_Allre
75E7	1.99979608E8	33.0	245252.0	6059988.0	mesh::Gri
32E7	1.06216242E8	257.0	582252.0	413293.0	algs::Hyp
55E7	9.84845747E8	193.0	1158.0	5102828.0	algs::Hyp
31E7	1.8790031E7	512731.0	0.0	37.0	MPI_Test(
48E7	1.5461689E7	32.0	107258.0	483178.0	algs::Hyp
45E7	1.0635845E7	141.0	0.0	75432.0	MPI_Bcast
85E7	1.071754E7	10840.0	10840.0	989.0	MPI_Isend
36E7	1.730215E7	33.0	39349.0	524308.0	algs::Hyp
.0	8077708.0	405288.0	0.0	20.0	MPI_Comm
.0	9074516.0	32.0	32948.0	283579.0	algs::Hyp



ParaProf User Event Details (MPI message size)





ParaProf Profile Analysis Features

- ❑ Inter-window event management
 - Full event propagation
 - Hyperlinked displays
- ❑ Window configuration and help management
 - Popup menus
 - Full preference control
 - Data view control
- ❑ Maturation of profile performance data views
- ❑ Java-based implementation
 - Extensible
- ❑ Performance database connectivity



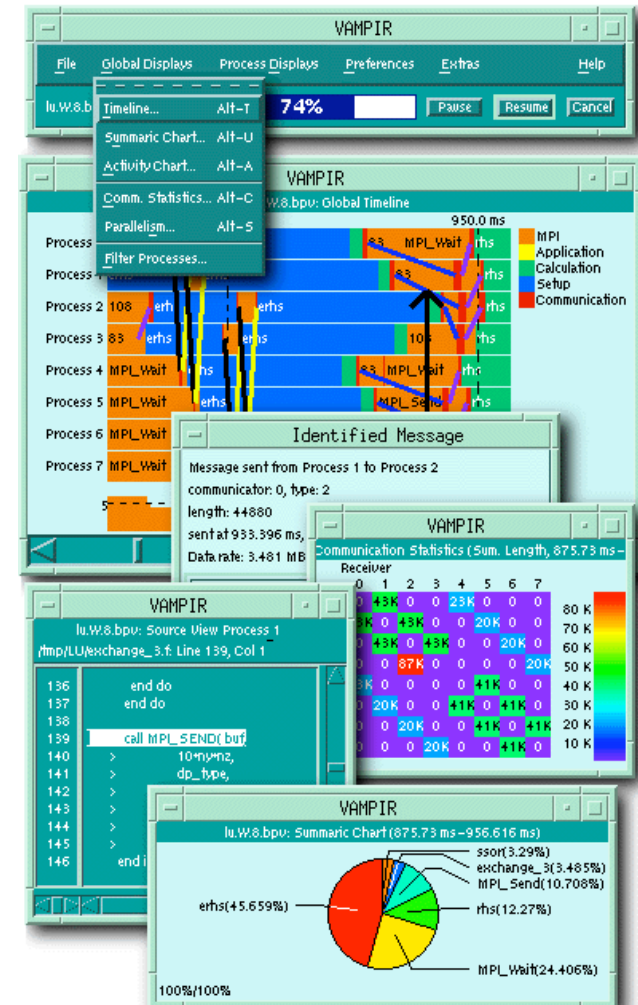
ParaProf Enhancements

- ❑ Readers completely separated from the GUI
- ❑ Access to performance profile database
- ❑ Profile translators
 - *mpiP, papiprof, dynaprof*
- ❑ Callgraph display
 - *prof / gprof* style with hyperlinks
- ❑ Integration of 3D performance plotting library
- ❑ Scalable profile analysis
 - Statistical histograms, cluster analysis, ...
- ❑ Generalized programmable analysis engine
- ❑ Cross-experiment analysis



Vampir Trace Visualization

- Visualization and Analysis of MPI Programs
- Originally developed by Forschungszentrum Jülich
- Current development by Technical University Dresden
- Distributed by PALLAS, Germany



□ <http://www.pallas.de/pages/vampir.htm>



Using TAU with Vampir

- ❑ Configure TAU with `-TRACE` option

```
% configure -TRACE -SGITIMERS ...
```

- ❑ Execute application

```
% mpirun -np 4 a.out
```

- ❑ This generates TAU traces and event descriptors

- ❑ Merge all traces using `tau_merge`

```
% tau_merge *.trc app.trc
```

- ❑ Convert traces to Vampir Trace format using `tau_convert`

```
% tau_convert -pv app.trc tau.edf app.pv
```

- ❑ Load generated trace file in Vampir

```
% vampir app.pv
```



Case Study: SIMPLE Performance Analysis

- SIMPLE hydrodynamics benchmark
 - C code with MPI message communication
 - Multiple instrumentation methods
 - source-to-source translation (PDT)
 - MPI wrapper library level instrumentation (PMPI)
 - pre-execution binary instrumentation (DyninstAPI)
 - Alternative measurement strategies
 - statistical profiles of software actions
 - statistical profiles of hardware actions (PCL, PAPI)
 - program event tracing
 - choice of time source
 - gettimeofday, physical clock, CPU, process virtual



SIMPLE Source Instrumentation (Preprocessed)

- PDT automatically generates instrumentation code
 - Names events with full function signatures

```
int compute_heat_conduction(  
    double theta_hat[X][Y], double deltat, double new_r[X][Y],  
    double new_z[X][Y], double new_alpha[X][Y],  
    double new_rho[X][Y], double theta_l[X][Y],  
    double Gamma_k[X][Y], double Gamma_l[X][Y])  
{  
    TAU_PROFILE("int compute_heat_conduction(  
        double (*)[259], double, double (*)[259],  
        double (*)[259], double (*)[259], double (*)[259],  
        double (*)[259], double (*)[259], double (*)[259])",  
        " ", TAU_USER);  
    ...  
}
```

- Similarly for all other routines in SIMPLE program



MPI Library Instrumentation (MPI_Send)

- Uses MPI profiling interposition library (PMPI)

```
int MPI_Send(...)
...
{
    int returnVal, typesize;
    TAU_PROFILE_TIMER(tautimer, "MPI_Send()", " ", TAU_MESSAGE);
    TAU_PROFILE_START(tautimer);
    if (dest != MPI_PROC_NULL) {
        PMPI_Type_size(datatype, &typesize);
        TAU_TRACE_SENDMSG(tag, dest, typesize*count);
    }
    returnVal = PMPI_Send(buf, count, datatype, dest, tag,
comm);
    TAU_PROFILE_STOP(tautimer);
    return returnVal;
}
```




MPI Library Instrumentation (MPI_Recv)

```
int MPI_Recv(...)
...
{
    int returnVal, size;
    TAU_PROFILE_TIMER(tautimer, "MPI_Recv()", " ", TAU_MESSAGE);
    TAU_PROFILE_START(tautimer);
    returnVal = PMPI_Recv(buf, count, datatype, src, tag, comm,
        status);
    if (src != MPI_PROC_NULL && returnVal == MPI_SUCCESS) {
        PMPI_Get_count(status, MPI_BYTE, &size);
        TAU_TRACE_RECVMSG(status->MPI_TAG, status->MPI_SOURCE,
            size);
    }
    TAU_PROFILE_STOP(tautimer);
    return returnVal;
}
```



Multi-Level Instrumentation (Profiling)

four processes

event legend

Profile per process

%time	msec	total msec	#call	#subrs	usec/call	name
49.9	19,295	31,066	206388	4.12776E+06	151	polynomial
17.6	10,946	10,946	3	9	3648787	net_accept
17.6	10,930	10,930	3.71498E+06	0	3	power
13.3	6,684	8,277	273	546	30320	socket_recv
3.1	1,952	1,954	561	1122	3484	net_recv
3.9	1,554	2,399	10	317580	239940	compute_viscos
2.1	1,316	1,318	628	1256	2100	net_send
35.8	1,117	22,279	10	119070	2227940	compute_temperature

global profile

Function	Percentage
polynomial	31.0%
net_accept	17.0%
power	17.0%
socket_recv	10.0%
net_recv	3.0%
compute_viscosity_interior	2.0%

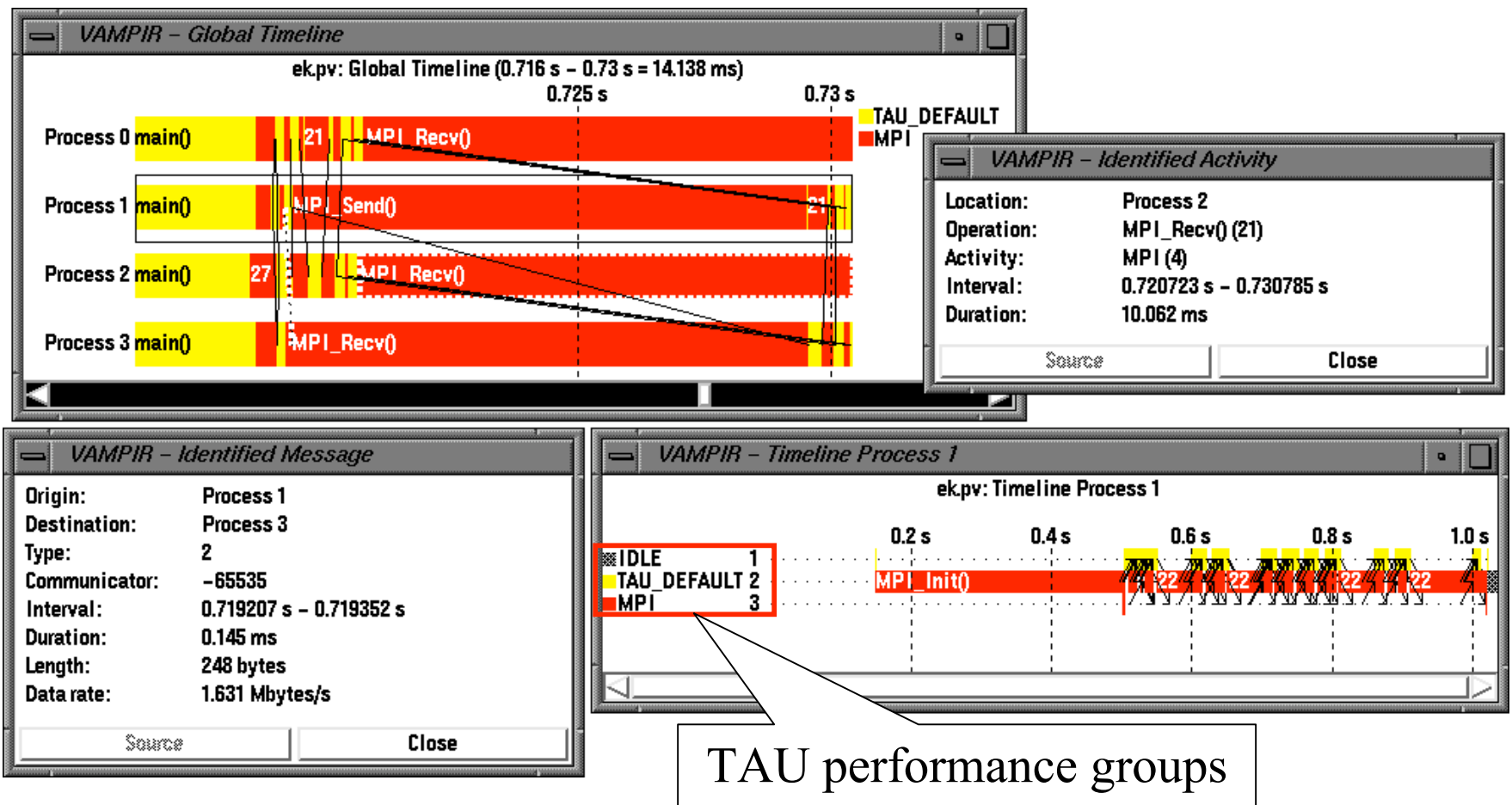
Racy Help Window

n,c,t stands for: Node, Context and Thread.
Using the right mouse button, double click here to display more detailed data about this thread.



Multi-Level Instrumentation (Tracing)

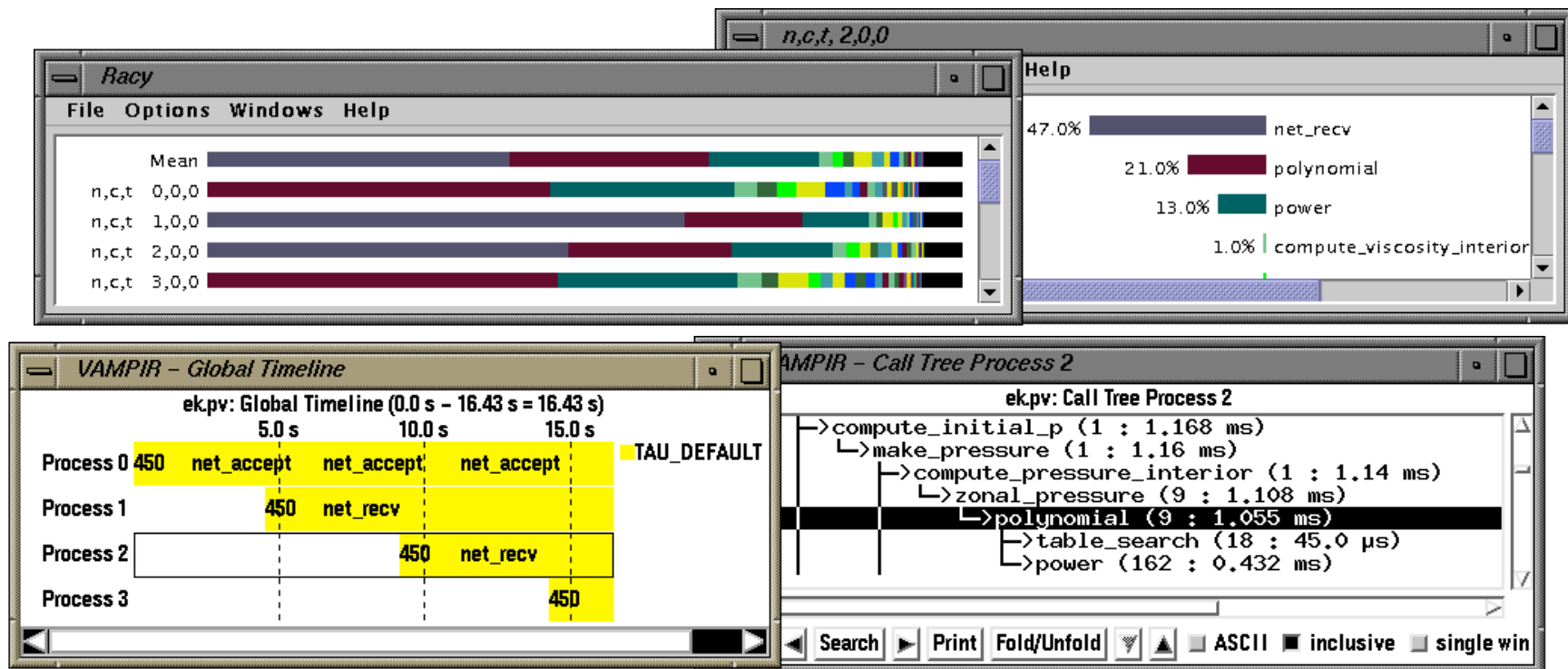
- Relink with TAU library configured for tracing
 - No modification of source instrumentation required!





Dynamic Instrumentation of SIMPLE

- ❑ Uses DynInstAPI for runtime code patching
- ❑ Mutator loads measurement library, instruments mutatee
 - One mutator (*tau_run*) per executable image
 - `mpirun -np <n> tau.shell`





Event Tracing using DyninstAPI

VAMPIR - Call Tree Process 0

ek.pv: Call Tree Process 0

- make_temperature (1 : 4.105 ms)
 - compute_temperature_interior (1 : 4.088 ms)
 - revised_temperature (18 : 2.633 ms)
 - ↳newton_raphson (18 : 2.524 ms)
 - ↳energy_equation (18 : 2.43 ms)
 - ↳zonal_energy (18 : 2.339 ms)
 - ↳polynomial (18 : 2.194 ms)
 - ↳table_search (36 : 100.0 µs)
 - ↳power (324 : 0.859 ms)
 - ↳zonal_pressure (9 : 1.176 ms)
 - ↳polynomial (9 : 1.134 ms)
 - ↳table_search (18 : 42.0 µs)
 - ↳power (162 : 0.384 ms)
 - compute_temperature_boundaries (1 : 5.0 µs)
 - make_sigma (1 : 37.0 µs)
 - ↳_mth_i_dpowd (9 : 31.0 µs)
 - make_cc_boundaries (1 : 17.0 µs)
 - make_gamma_interior (2 : 2.069 ms)
 - ↳PMPI_Irecv (2 : 0.16 ms)
 - ↳MPIR_ToPointer (4 : 10.0 µs)
 - ↳MPID_SBalloc (2 : 7.0 µs)
 - ↳MPID_IrecvDatatype (2 : 90.0 µs)
 - ↳MPID_Msg_rep (2 : 6.0 µs)
 - ↳MPID_IrecvContig (2 : 49.0 µs)
 - ↳MPID_Search_unexpected_queue_and_post (2 : 39.0 µs)
 - ↳MPID_Search_unexpected_queue (2 : 5.0 µs)
 - ↳MPID_Enqueue (2 : 15.0 µs)
 - ↳MPID_SBalloc (2 : 4.0 µs)
 - ↳PMPI_Isend (2 : 0.683 ms)
 - ↳MPIR_ToPointer (4 : 10.0 µs)
 - ↳MPID_SBalloc (2 : 5.0 µs)
 - ↳MPID_IsendDatatype (2 : 0.607 ms)
 - ↳MPID_Msg_rep (2 : 4.0 µs)
 - ↳MPID_IsendContig (2 : 0.586 ms)
 - ↳MPID_CH_Eagerb_isend_short (2 : 0.574 ms)
 - ↳MPID_CH_Pkt_pack (2 : 6.0 µs)
 - ↳p4_get_my_id (2 : 4.0 µs)
 - ↳send_message (2 : 0.508 ms)
 - ↳print_conn_type (2 : 6.0 µs)
 - ↳p4_dprintf1 (2 : 5.0 µs)
 - ↳socket_send (2 : 0.469 ms)
 - ↳p4_dprintf1 (8 : 97.0 µs)
 - ↳p4_num_total_ids (4 : 9.0 µs)
 - ↳net_send (4 : 0.31 ms)

VAMPIR - Process 0 Call Breakdown

ek.pv: Process 0 Call Breakdown (polynomial)

- zonal_pressure (27)
 - zonal_energy (36)
 - polynomial
 - table_search (126)
 - power (1134)

Close Search Print ASCII smaller larger

Close Search Print Fold/Unfold ASCII inclusive single win smaller larger line 246 (1078)