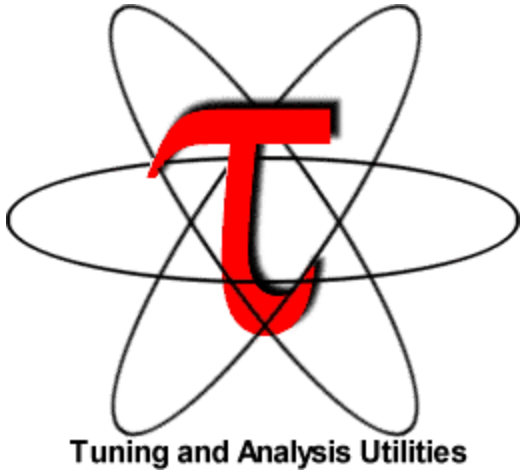
Tuning and Analysis Utilities

Sameer Shende
University of Oregon



General Problems

How do we create robust and ubiquitous performance technology for the analysis and tuning of parallel and distributed software and systems in the presence of (evolving) complexity challenges?

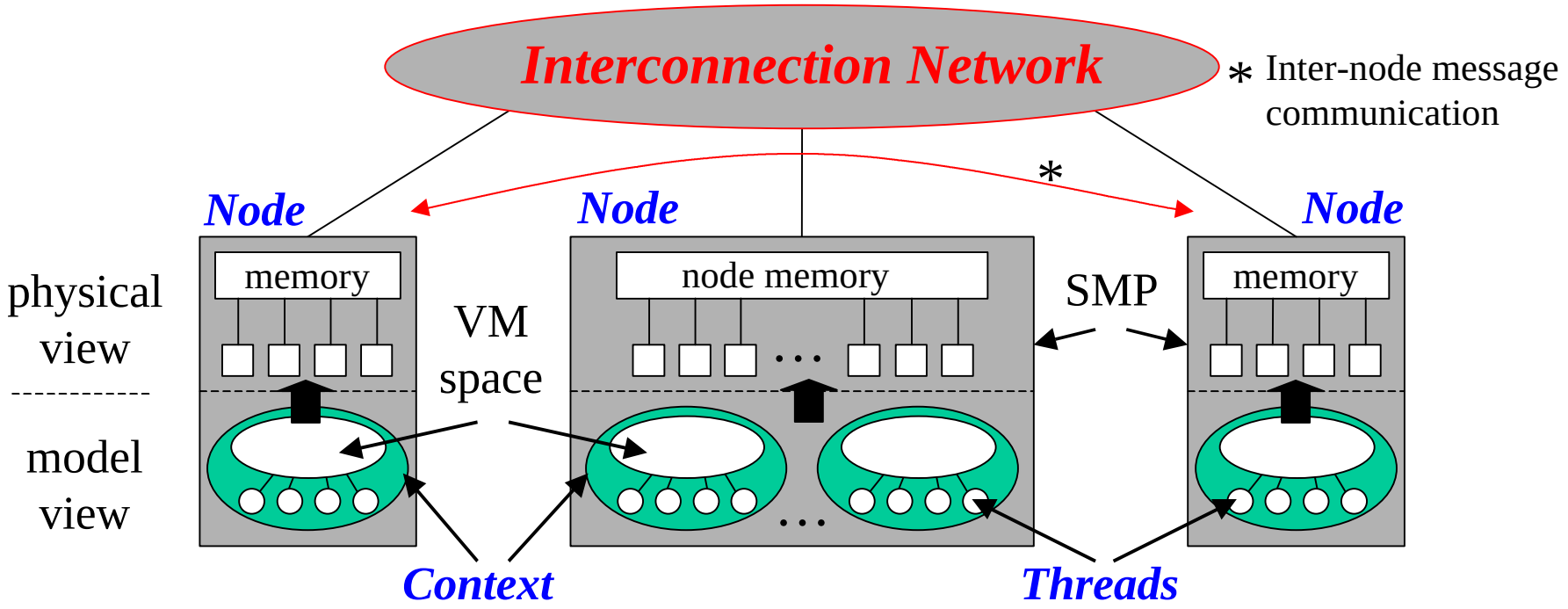
How do we apply performance technology effectively for the variety and diversity of performance problems that arise in the context of complex parallel and distributed computer systems.

Computation Model for Performance Technology

- How to address dual performance technology goals?
  - Robust capabilities + widely available methodologies
  - Contend with problems of system diversity
  - Flexible tool composition/configuration/integration
- Approaches
  - Restrict computation types / performance problems
    - limited performance technology coverage
  - Base technology on abstract computation model
    - general architecture and software execution features
    - map features/methods to existing complex system types
    - develop capabilities that can adapt and be optimized

General Complex System Computation Model

- ***Node***: physically distinct shared memory machine
  - Message passing *node interconnection network*
- ***Context***: distinct virtual memory space within node
- ***Thread***: execution threads (user/system) in context



Definitions – Profiling

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

Definitions – Tracing

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

Definitions – Instrumentation

- Instrumentation
  - Insertion of extra code (hooks) into program
  - Source instrumentation
    - Done by compiler, source-to-source translator, or manually
    - \+ portable
    - \+ links back to program code
    - – re-compile is necessary for (change in) instrumentation
    - – requires source to be available
    - – hard to use in standard way for mix-language programs
    - – source-to-source translators hard to develop for C++, F90
  - Object code instrumentation
    - "re-writing" the executable to insert hooks

Definitions – Instrumentation (continued)

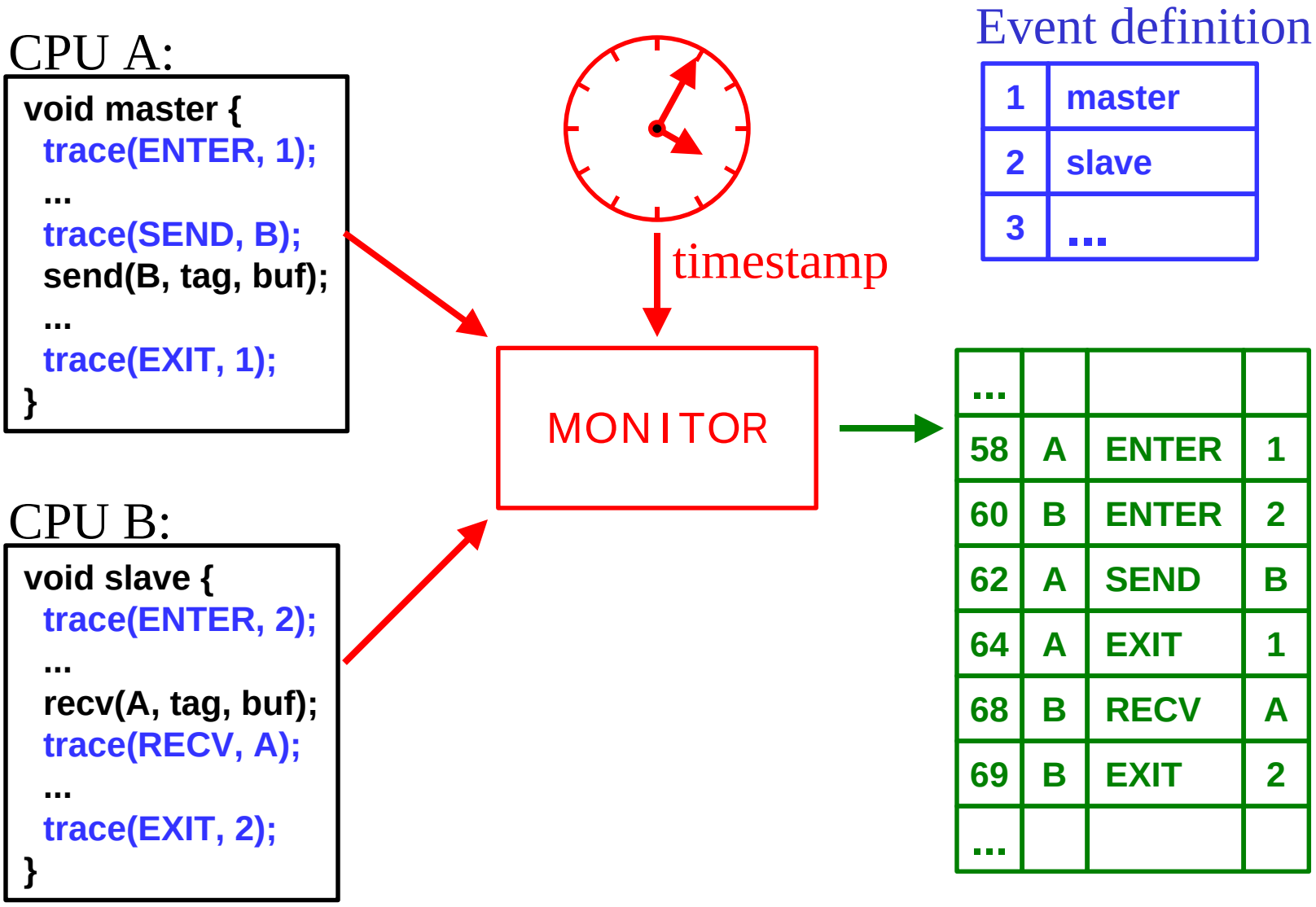
- Dynamic code instrumentation
  - a debugger-like instrumentation approach
  - executable code instrumentation on running program
  - DynInst and DPCL are examples

  +/– switch around compared to source instrumentation
- Pre-instrumented library
  - typically used for MPI and PVM program analysis
  - supported by link-time library interposition

  \+ easy to use since only re-linking is necessary

  – can only record information about library entities

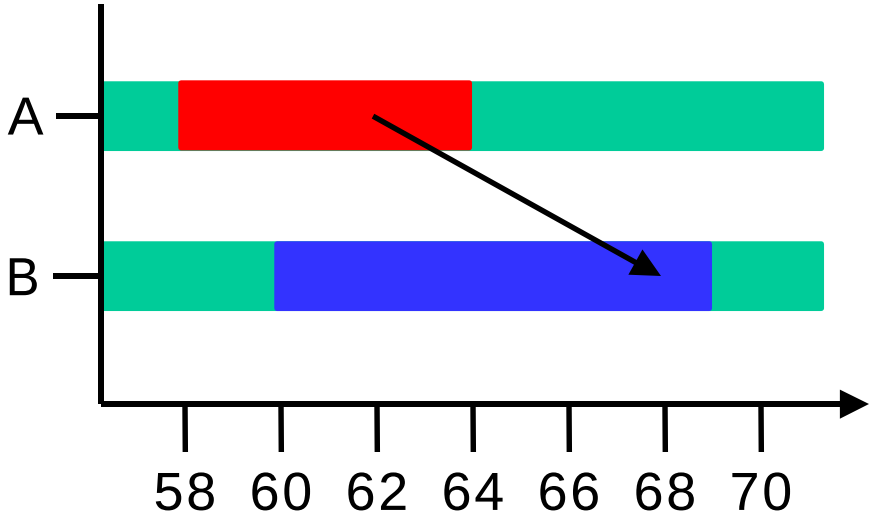
Event Tracing: Instrumentation, Monitor, Trace



Event Tracing: "Timeline" Visualization

1	master
2	slave
3	...

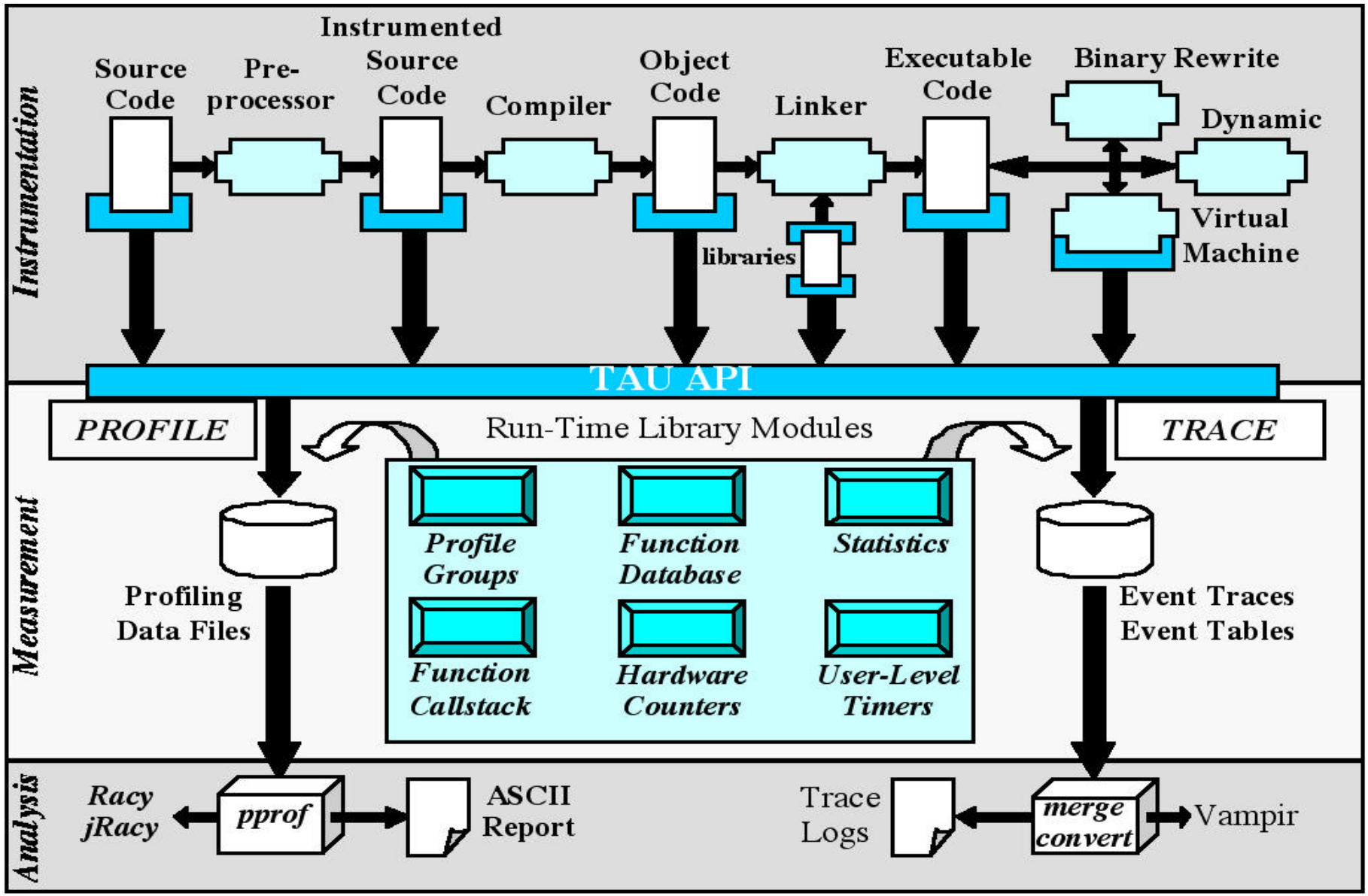
...			
58	A	ENTER	1
60	B	ENTER	2
62	A	SEND	B
64	A	EXIT	1
68	B	RECV	A
69	B	EXIT	2
...			



TAU Performance System Framework

- Tuning and Analysis Utilities
- Performance system framework for scalable parallel and distributed high-performance computing
- Targets a general complex system computation model
  - nodes / contexts / threads
  - Multi-level: system / software / parallelism
  - Measurement and analysis abstraction
- Integrated toolkit for performance instrumentation, measurement, analysis, and visualization
  - Portable performance profiling/tracing facility
  - Open software approach

TAU Performance System Architecture



TAU Instrumentation

- Flexible instrumentation mechanisms at multiple levels
  - Source code
    - manual
    - automatic using *Program Database Toolkit (PDT), OPARI*
  - Object code
    - pre-instrumented libraries (e.g., MPI using PMPI)
    - statically linked
    - dynamically linked (e.g., Virtual machine instrumentation)
    - fast breakpoints (compiler generated)
  - Executable code
    - dynamic instrumentation (pre-execution) using *DynInstAPI*

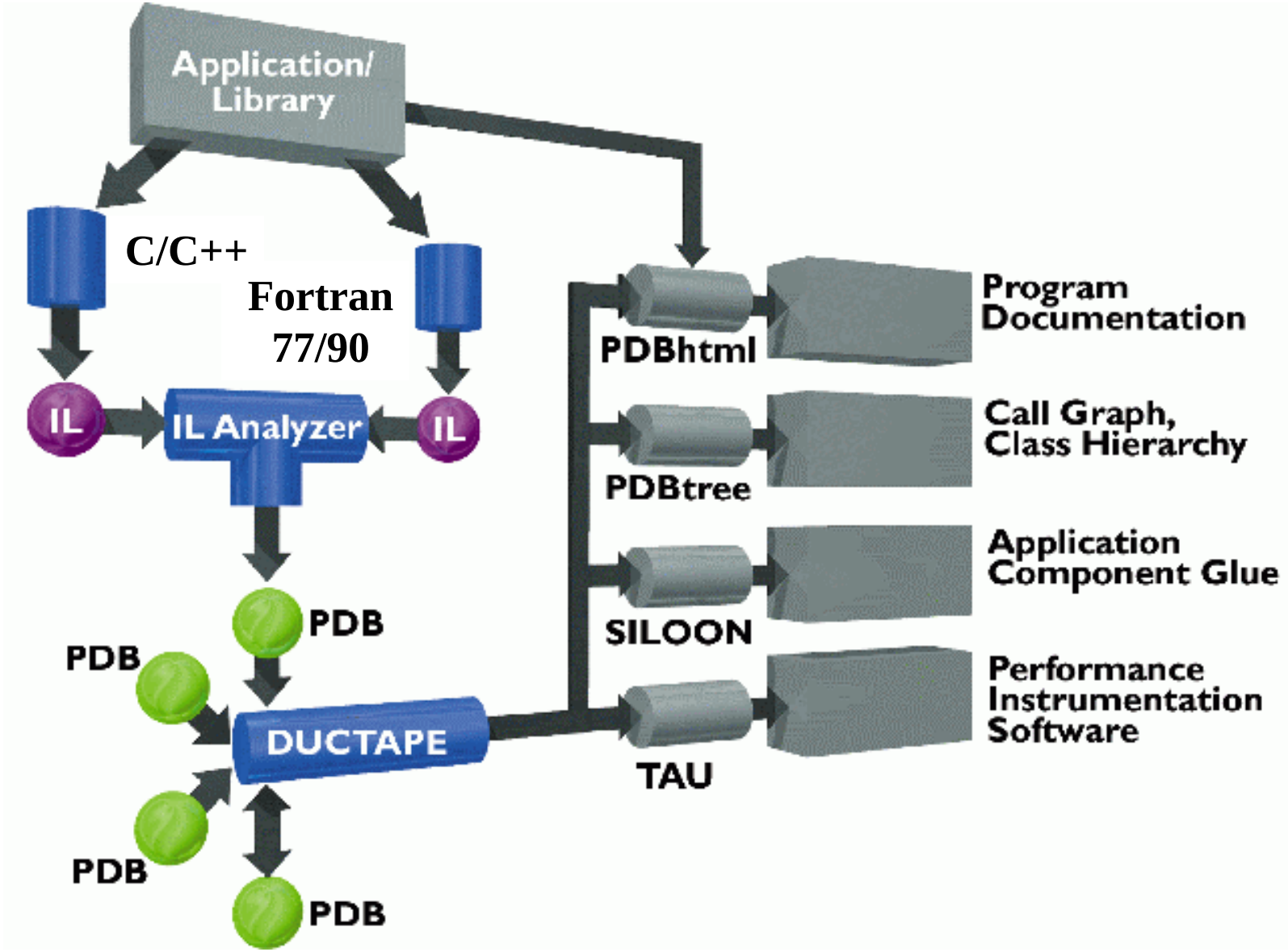
TAU Instrumentation (continued)

- Targets common measurement interface (*TAU API*)
- Object-based design and implementation
  - Macro-based, using constructor/destructor techniques
  - Program units: function, classes, templates, blocks
  - Uniquely identify functions and templates
    - name and type signature (name registration)
    - static object creates performance entry
    - dynamic object receives static object pointer
    - runtime type identification for template instantiations
  - C and Fortran instrumentation variants
- Instrumentation and measurement optimization

Program Database Toolkit (PDT)

- Program code analysis framework for developing source-based 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
- Target and integrate multiple source languages
- Use in TAU to build automated performance instrumentation tools

PDT Architecture and Tools



PDT Components

- Language front end
  - Edison Design Group (EDG): C, C++, Java
  - Mutek Solutions Ltd.: F77, F90
  - 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
  - processes and merges PDB files
  - C++ library to access the PDB for PDT applications

TAU Measurement

- Performance information
  - High-resolution timer library (real-time / virtual clocks)
  - General software counter library (user-defined events)
  - Hardware performance counters
    - *PCL* (Performance Counter Library) (ZAM, Germany)
    - *PAPI* (Performance API) (UTK, Ptools Consortium)
    - consistent, portable API
- Organization
  - Node, context, thread levels
  - Profile groups for collective events (runtime selective)
  - Performance data mapping between software levels

TAU Measurement (continued)

- Parallel profiling
  - Function-level, block-level, statement-level
  - Supports user-defined events
  - TAU parallel profile database
  - Function callstack
  - Hardware counts values (in replace of time)
- Tracing
  - All profile-level events
  - Interprocess communication events
  - Timestamp synchronization
- User-configurable measurement library (user controlled)

TAU Measurement System Configuration

- configure [OPTIONS]
  - {-c++=<CC>, -cc=<cc>} Specify C++ and C compilers
  - {-pthread, -sproc} Use pthread or SGI sproc threads
  - -openmp Use OpenMP threads
  - -jdk=<dir> Specify location of Java Dev. Kit
  - -opari=<dir> Specify location of Opari OpenMP tool
  - {-pcl, -papi}=<dir> Specify location of PCL or PAPI
  - -pdt=<dir> Specify location of PDT
  - -dyninst=<dir> Specify location of DynInst Package
  - {-mpiinc=<d>, mpilib=<d>} Specify MPI library instrumentation
  - -TRACE Generate TAU event traces
  - -PROFILE Generate TAU profiles
  - -CPUTIME Use usertime+system time
  - -PAPIWALLCLOCK Use PAPI to access wallclock time
  - -PAPIVIRTUAL Use PAPI for virtual (user) time

TAU Measurement Configuration – Examples

- ./configure -c++=KCC –SGITIMERS
  - Use TAU with KCC and fast nanosecond timers on SGI
  - Enable TAU profiling (default)
- ./configure -TRACE –PROFILE
  - Enable both TAU profiling and tracing
- ./configure -c++=guidec++ -cc=guidec
  -papi=/usr/local/packages/papi –openmp
  -mpiinc=/usr/packages/mpich/include
  -mpilib=/usr/packages/mpich/lib
  - Use OpenMP+MPI using KAI's Guide compiler suite and use PAPI for accessing hardware performance counters for measurements
- Typically configure multiple measurement libraries

TAU Measurement API

- 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 (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();

Compiling: TAU Makefiles

- Include TAU Makefile in the user's Makefile.
- Variables:
  - TAU_CXX — Specify the C++ compiler
  - TAU_CC — Specify the C compiler used by TAU
  - TAU_DEFS — Defines used by TAU. Add to CFLAGS
  - TAU_INCLUDE — Header files include path. Add to CFLAGS
  - TAU_LIBS — Statically linked TAU library. Add to LIBS
  - TAU_SHLIBS — Dynamically linked TAU library
  - TAU_MPI_LIBS — TAU's MPI wrapper library for C/C++
  - TAU_MPI_FLIBS — TAU's MPI wrapper library for F90
  - TAU_FORTRANLIBS — Must be linked in with C++ linker for F90.
- Note: Not including TAU_DEFS in CFLAGS disables instrumentation in C/C++ programs.

Including TAU Makefile - Example

```
include /usr/tau/sgi64/lib/Makefile.tau-pthread-kcc
CXX = $(TAU_CXX)
CC  = $(TAU_CC)
CFLAGS = $(TAU_DEFS)
LIBS = $(TAU_LIBS)
OBJS = ...
TARGET= a.out
TARGET: $(OBJS)
     $(CXX) $(LDFLAGS) $(OBJS) -o $@ $(LIBS)
.cpp.o:
     $(CC) $(CFLAGS) -c $< -o $@
```

TAU Makefile for PDT

```
include /usr/tau/include/Makefile
CXX = $(TAU_CXX)
CC  = $(TAU_CC)
PDTPARSE = $(PDTDIR)/$(CONFIG_ARCH)/bin/cxxparse
TAUINSTR = $(TAUROOT)/$(CONFIG_ARCH)/bin/tau_instrumentor
CFLAGS = $(TAU_DEFS)
LIBS = $(TAU_LIBS)
OBJS = ...
TARGET= a.out
TARGET: $(OBJS)
      $(CXX) $(LDFLAGS) $(OBJS) -o $@ $(LIBS)
.cpp.o:
      $(PDTPARSE) $<
      $(TAUINSTR) $*.pdb $< -o $*.inst.cpp
      $(CC) $(CFLAGS) -c $*.inst.cpp -o $@
```

Setup: Running Applications

```
% setenv PROFILEDIR /home/data/experiments/profile/01
% setenv TRACEDIR   /home/data/experiments/trace/01
% set path=($path <taudir>/<arch>/bin)
% setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH\:<taudir>/<arch>/lib
For PAPI/PCL:
% setenv PAPI_EVENT PAPI_FP_INS
% setenv PCL_EVENT PCL_FP_INSTR
For Java (without instrumentation):
% java application
With instrumentation:
% java -XrunTAU application
% java -XrunTAU:exclude=sun/io,java application
For DyninstAPI:
% a.out
% tau_run a.out
% tau_run -XrunTAUsh-papi a.out
```

TAU Analysis

- Profile analysis
  - Pprof
    - parallel profiler with text-based display
  - Racy
    - graphical interface to pprof (Tcl/Tk)
  - jRacy
    - Java implementation of Racy
- Trace analysis and visualization
  - Trace merging and clock adjustment (if necessary)
  - Trace format conversion (ALOG, SDDF, Vampir)
  - Vampir (Pallas) trace visualization

Pprof Command

- pprof [-c|-b|-m|-t|-e|-i] [-r] [-s] [-n num] [-f file] [-l] [nodes]
  - -c Sort according to number of calls
  - -b Sort according to number of subroutines called
  - -m Sort according to msecs (exclusive time total)
  - -t Sort according to total msecs (inclusive time total)
  - -e Sort according to exclusive time per call
  - -i Sort according to inclusive time per call
  - -v Sort according to standard deviation (exclusive usec)
  - -r Reverse sorting order
  - -s Print only summary profile information
  - -n num Print only first number of functions
  - -f file Specify full path and filename without node ids
  - -l nodes List all functions and exit (prints only info about all contexts/threads of given node numbers)

Pprof Output (NAS Parallel Benchmark – LU)

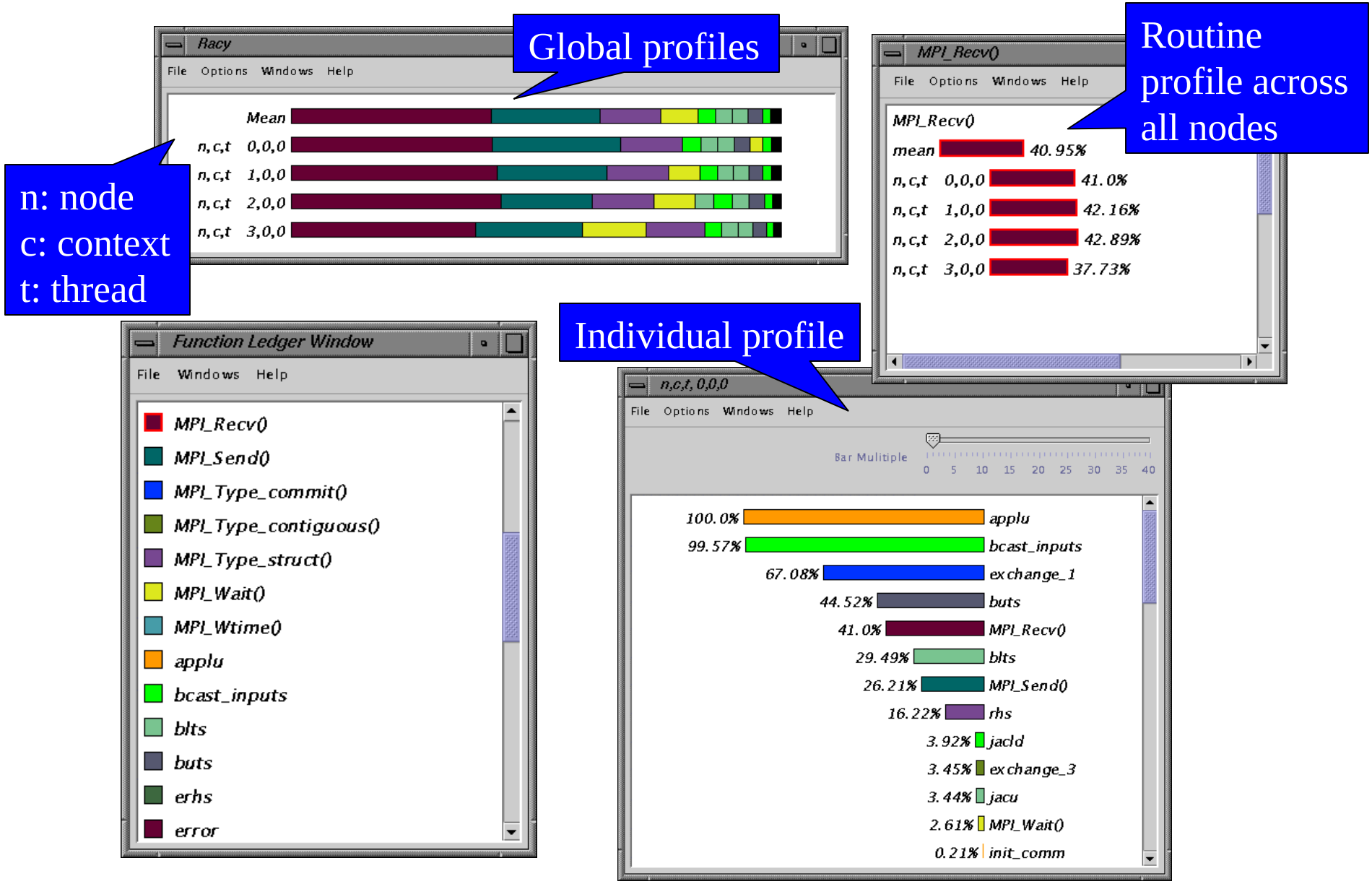
- Intel Quad PIII Xeon, RedHat, PGI F90
- F90 + MPICH
- Profile for:
  - Node
  - Context
  - Thread
- Application events and MPI events

emacs@neutron.cs.uoregon.edu

Buffers Files Tools Edit Search Mule Help

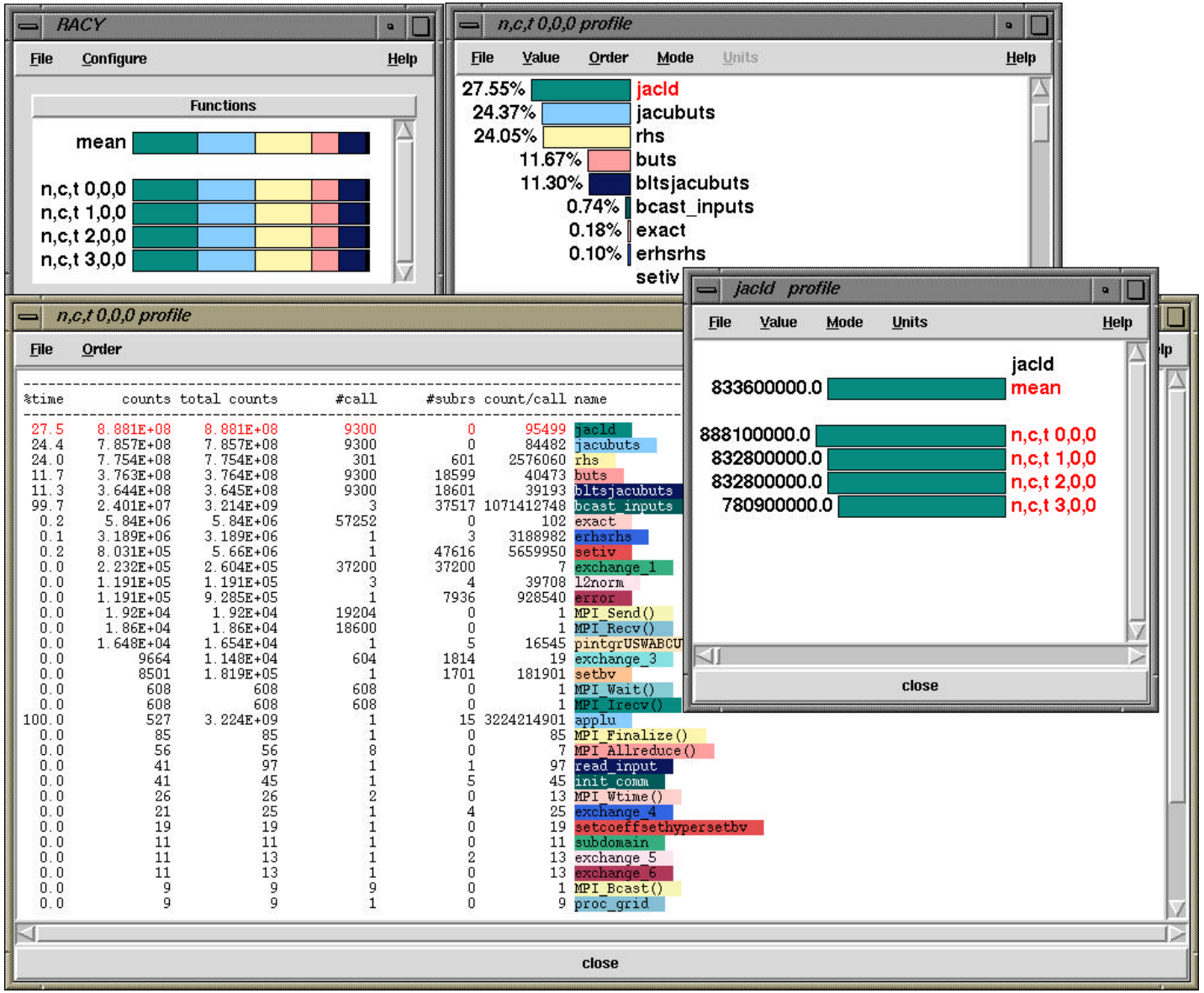
```
Reading Profile files in profile.*

NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0           1     3:11.293           1          15  191293269 applu
 99.6       3,667     3:10.463           3       37517   63487925 bcast_inputs
 67.1         491     2:08.326       37200       37200       3450 exchange_1
 44.5       6,461     1:25.159        9300       18600       9157 buts
 41.0    1:18.436     1:18.436       18600           0       4217 MPI_Recv()
 29.5       6,778       56,407        9300       18600       6065 blts
 26.2      50,142       50,142       19204           0       2611 MPI_Send()
 16.2      24,451       31,031         301         602     103096 rhs
  3.9       7,501        7,501        9300           0        807 jacld
  3.4         838        6,594         604        1812      10918 exchange_3
  3.4       6,590        6,590        9300           0        709 jacu
  2.6       4,989        4,989         608           0       8206 MPI_Wait()
  0.2        0.44          400           1           4     400081 init_comm
  0.2         398          399           1          39     399634 MPI_Init()
  0.1         140          247           1       47616     247086 setiv
  0.1         131          131       57252           0          2 exact
  0.1          89          103           1           2     103168 erhs
  0.1       0.966           96           1           2      96458 read_input
  0.0          95           95           9           0      10603 MPI_Bcast()
  0.0          26           44           1        7937      44878 error
  0.0          24           24         608           0         40 MPI_Irecv()
  0.0          15           15           1           5      15630 MPI_Finalize()
  0.0           4           12           1        1700      12335 setbv
  0.0           7            8           3           3       2893 l2norm
  0.0           3            3           8           0        491 MPI_Allreduce()
  0.0           1            3           1           6       3874 pintgr
  0.0           1            1           1           0       1007 MPI_Barrier()
  0.0       0.116        0.837           1           4        837 exchange_4
  0.0       0.512        0.512           1           0        512 MPI_Keyval_create()
  0.0       0.121        0.353           1           2        353 exchange_5
  0.0       0.024        0.191           1           2        191 exchange_6
  0.0       0.103        0.103           6           0         17 MPI_Type_contiguous()
--:--  NPB_LU.out        (Fundamental)--L8--Top----------------------------------------
```

jRacy (NAS Parallel Benchmark – LU)



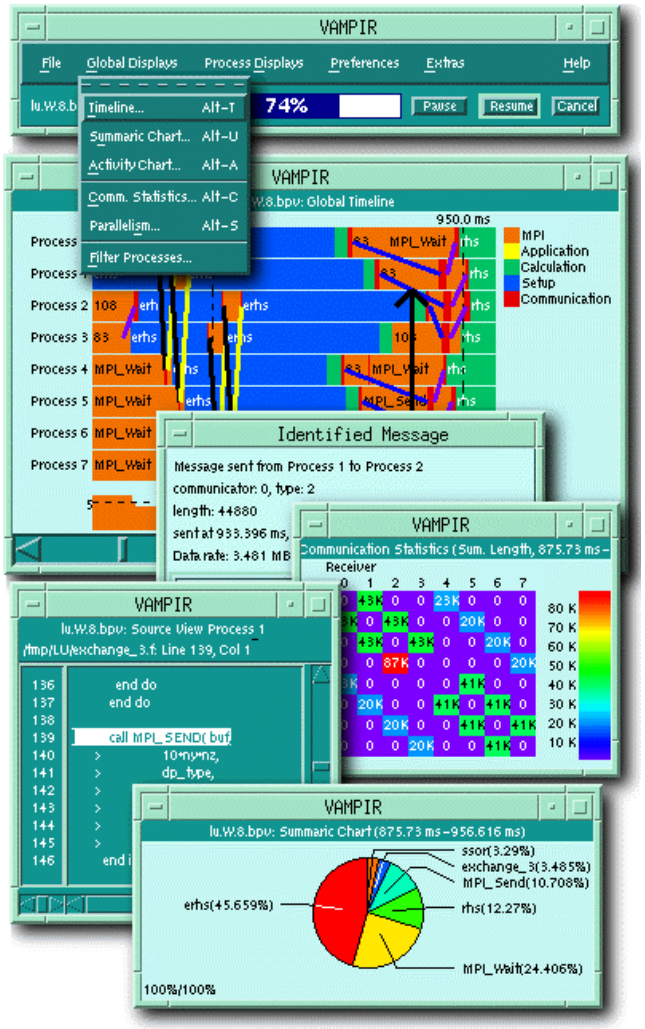
TAU and PAPI (NAS Parallel Benchmark – LU)

- Floating point operations
- Replaces execution time
- Only requires relinking to different measurement library



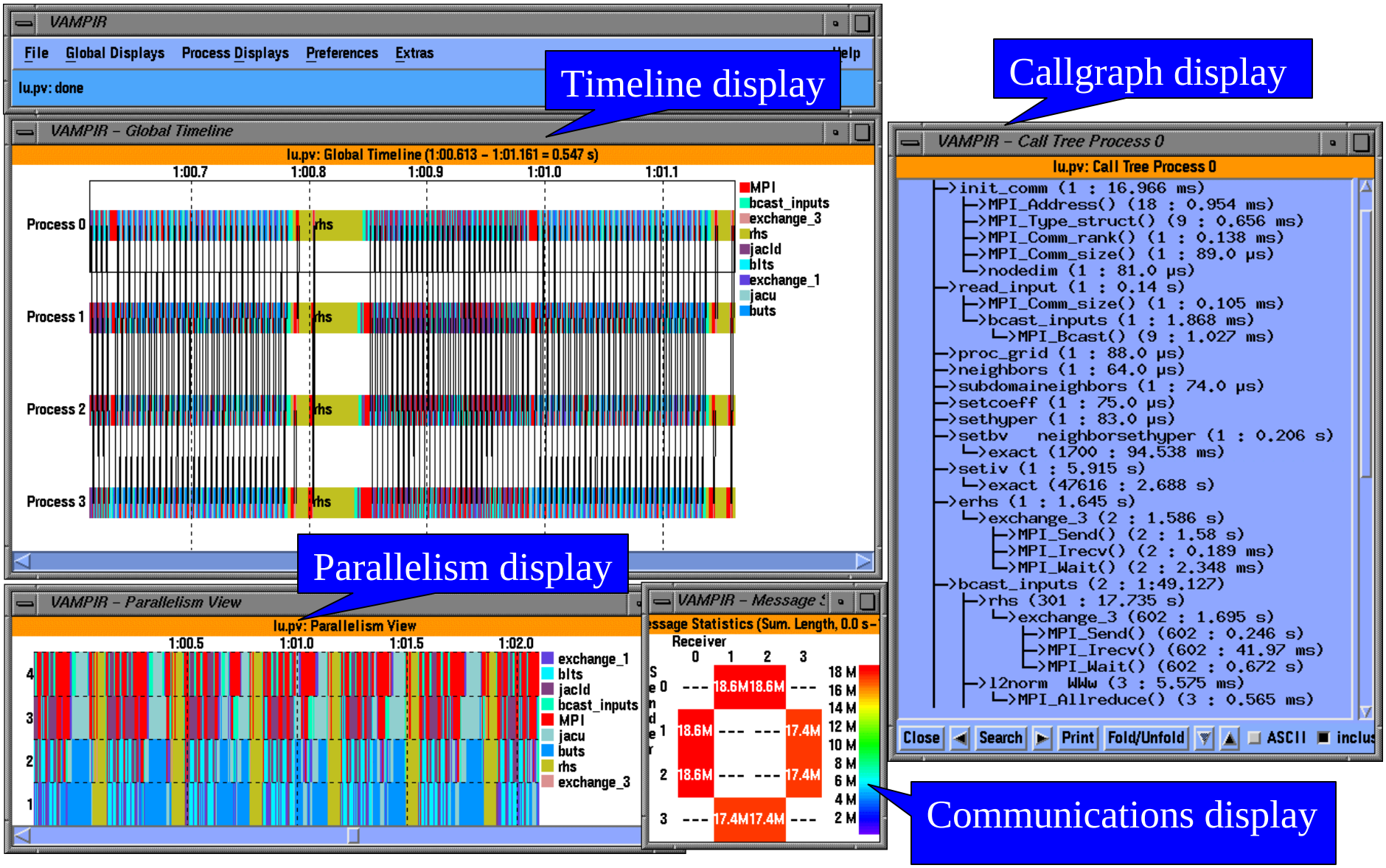
Vampir Trace Visualization Tool

- 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**

Vampir (NAS Parallel Benchmark – LU)



TAU Performance System Status

- Computing platforms
  - IBM SP, SGI Origin 2K/3K, Intel Teraflop, Cray T3E, Compaq SC, HP, Sun, Windows, IA-32, IA-64, Linux, …
- Programming languages
  - C, C++, Fortran 77/90, HPF, Java, OpenMP
- Communication libraries
  - MPI, PVM, Nexus, Tulip, ACLMPL, MPIJava
- Thread libraries
  - pthreads, Java,Windows, Tulip, SMARTS, OpenMP
- Compilers
  - KAI, PGI, GNU, Fujitsu, Sun, Microsoft, SGI, Cray, IBM, Compaq

TAU Performance System Status (continued)

- Application libraries
  - Blitz++, A++/P++, ACLVIS, PAWS, SAMRAI, Overture
- Application frameworks
  - POOMA, POOMA-2, MC++, Conejo, Uintah, UPS
- Projects
  - Aurora / SCALEA: ACPC, University of Vienna
- TAU full distribution (Version 2.10, web download)
  - Measurement library and profile analysis tools
  - Automatic software installation
  - Performance analysis examples
  - Extensive TAU User's Guide

PDT Status

- Program Database Toolkit (Version 2.0, web download)
  - EDG C++ front end (Version 2.45.2)
  - Mutek Fortran 90 front end (Version 2.4.1)
  - C++ and Fortran 90 IL Analyzer
  - DUCTAPE library
  - Standard C++ system header files (KCC Version 4.0f)
- PDT-constructed tools
  - Automatic TAU performance instrumentation
    - C, C++, Fortran 77, and Fortran 90
  - Program analysis support for SILOON and CHASM

Information

- TAU (http://www.acl.lanl.gov/tau)
- PDT (http://www.acl.lanl.gov/pdtoolkit)
- Tutorial at SC'01: M11
  B. Mohr, A. Malony, S. Shende, "*Performance Technology for Complex Parallel Systems*" Nov. 7, 2001, Denver, CO.
- LANL, NIC Booth, SC'01.

Support Acknowledgement

- TAU and PDT support:
  - Department of Engergy (DOE)
    - DOE 2000 ACTS contract
    - DOE MICS contract
    - DOE ASCI Level 3 (LANL, LLNL)
  - DARPA
  - NSF National Young Investigator (NYI) award

Hands-on session

- On mcurie.nersc.gov, copy files from
  /usr/local/pkg/acts/tau/tau2/tau-2.9/training
- See README file
- Set correct path e.g.,
  % set path=($path /usr/local/pkg/acts/tau/tau2/tau2.9/t3e/bin)
- Examine the Makefile.
- Type “make” in each directory; then execute the program
- Type “racy” or “vampir”
- Type a project name e.g., “matrix.pmf” and click OK to see the performance data.

Examples

The training directory contains example programs that illustrate the use of TAU instrumentation and measuremen options.

instrument	-	This contains a simple C++ example that shows how TAU's API can be used for manually instrumenting a C++ program. It highlights instrumentation for templates and user defined events.
threads	-	A simple multi-threaded program that shows how the main function of a thread is instrumented. Performance data is generated for each thread of execution. Configure with -pthread.
cthreads	-	Same as threads above, but for a C program. An instrumented C program may be compiled with a C compiler, but needs to be linked with a C++ linker. Configure with -pthread.
pi	-	An MPI program that calculates the value of pi and e. It highlights the use of TAU's MPI wrapper library. TAU needs to be configured with -mpiinc=<dir> and -mpilib=<dir>. Run using mpirun -np <procs> cpi <iterations>.
papi	-	A matrix multiply example that shows how to use TAU statement level timers for comparing the performance of two algorithms for matrix multiplication. When used with PAPI or PCL, this can highlight the cache behaviors of these algorithms. TAU should be configured with -papi=<dir> or -pcl=<dir> and the user should set PAPI_EVENT or PCL_EVENT respective environment variables, to use this.

Examples - (cont.)

papithreads - Same as papi, but uses threads to highlight how hardware performance counters may be used in a multi-threaded application. When it is used with PAPI, TAU should be configured with -papi=<dir> -pthread

autoinstrument - Shows the use of Program Database Toolkit (PDT) for automating the insertion of TAU macros in the source code. It requires configuring TAU with the -pdt=<dir> option. The Makefile is modified to illustrate the use of a source to source translator (tau_instrumentor).

NPB2.3 - The NAS Parallel Benchmark 2.3 [from NASA Ames]. It shows how to use TAU's MPI wrapper with a manually instrumented Fortran program. LU and SP are the two benchmarks. LU is instrumented completely, while only parts of the SP program are instrumented to contrast the coverage of routines. In both cases MPI level instrumentation is complete. TAU needs to be configured with -mpiinc=<dir> and -mpilib=<dir> to use this.