# *Performance Technology for Complex Parallel Systems*

**Allen D. Malony, Sameer Shende**
{malony,sameer}@cs.uoregon.edu

Department of Computer and Information Science
University of Oregon

**Bernd Mohr**
b.mohr@fz-juelich.de

Forschungszentrum Jülich
John von Neumann - Institut für Computing
Zentralinstitut für Angewandte Mathematik


# *Tutorial Outline – Part 1*

## Overview and Introduction (Malony, 1 hour)

- Introduction
  - Performance technology
  - Complexity challenges and general problems
- Computation Model for Performance Technology
  - Framework for performance problem solving
- TAU Performance System
  - Model-oriented framework architecture
  - TAU performance system toolkit
  - TAU measurement API and library configuration
  - Performance mapping

# *Tutorial Outline – Part 2*

## Complexity Scenarios (Shende, 1 hour)

- Message passing computation
- Multi-threaded computation
- Mixed-mode parallel computation
  - OpenMP+MPI
  - Java+MPI
- Object-oriented programming and C++
- Hierarchical parallel software frameworks
  - Task-based parallelism
  - Module coupling
- Evolution of the TAU performance system

# *Tutorial Outline – Part 3*

## Alternative Tools and Frameworks (Mohr, 1 hour)

- Commercial solutions
  - Vampir
  - Guideview
  - VGV
- Smart event trace analysis
  - KOJAK/EXPERT
- OpenMP performance interface (OPARI)
- APART European Commission IST working group
- Parallel performance tool integration
  - Integration of TAU and EXPERT

# *Tutorial Goals*

- Develop an appreciation for performance problem solving in complex computational environments
- Learn about the TAU performance system: measurement API, configuration, and analysis tools
- Understand how TAU is applied in complex parallel computation scenarios
- Learn about other tools and frameworks for performance analysis in complex parallel systems
- Consider how TAU and other tools may be applied to performance problems of tutorial participants and provide opportunity for follow-on interaction

# *Performance Technology for Complex Parallel Systems*

## *Part 1 – Overview and TAU Introduction*

*Allen D. Malony*

cs.uoregon.edu
COMPUTER & INFORMATION SCIENCE • UNIVERSITY OF OREGON

# *Performance Needs Æ Performance Technology*

- Observe/analyze/understand performance behavior
  - Multiple levels of software and hardware
  - Different types and detail of performance data
  - Alternative performance problem solving methods
  - Multiple targets of software and system application
- Robust AND ubiquitous performance technology
  - Broad scope of performance observability
  - Flexible and configurable mechanisms
  - Technology integration and extension
  - Cross-platform portability
  - Open, layered, and modular framework architecture

# *Parallel Performance Technology*

- Performance instrumentation tools
  - Different program code levels
  - Different system levels
- Performance measurement tools
  - Profiling and tracing of SW/HW performance events
  - Different SW and HW levels
- Performance analysis tools
  - Performance data analysis and presentation
  - Online and offline tools
- Performance experimentation
- Performance modeling and prediction tools

# *Complex Parallel Systems*

- Complexity in computing system architecture
  - Diverse parallel system architectures
    - Shared / distributed memory, cluster, hybrid, NOW, …
  - Sophisticated processor and memory architectures
  - Advanced network interface and switching architecture
- Complexity in parallel software environment
  - Diverse parallel programming paradigms
    - Shared memory multi-threading, message passing, hybrid
  - Hierarchical, multi-level software architectures
  - Optimizing compilers and sophisticated runtime systems
  - Advanced numerical libraries and application frameworks

# *Complexity Challenges*

- Computing system environment complexity
  - Observation integration and optimization
  - Access, accuracy, and granularity constraints
  - Diverse/specialized observation capabilities/technology
  - Restricted modes limit performance problem solving
- Sophisticated software development environments
  - Programming paradigms and performance models
  - Performance data mapping to software abstractions
  - Uniformity of performance abstraction across platforms
  - Rich observation capabilities and flexible configuration
  - Common performance problem solving methods

# *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?***

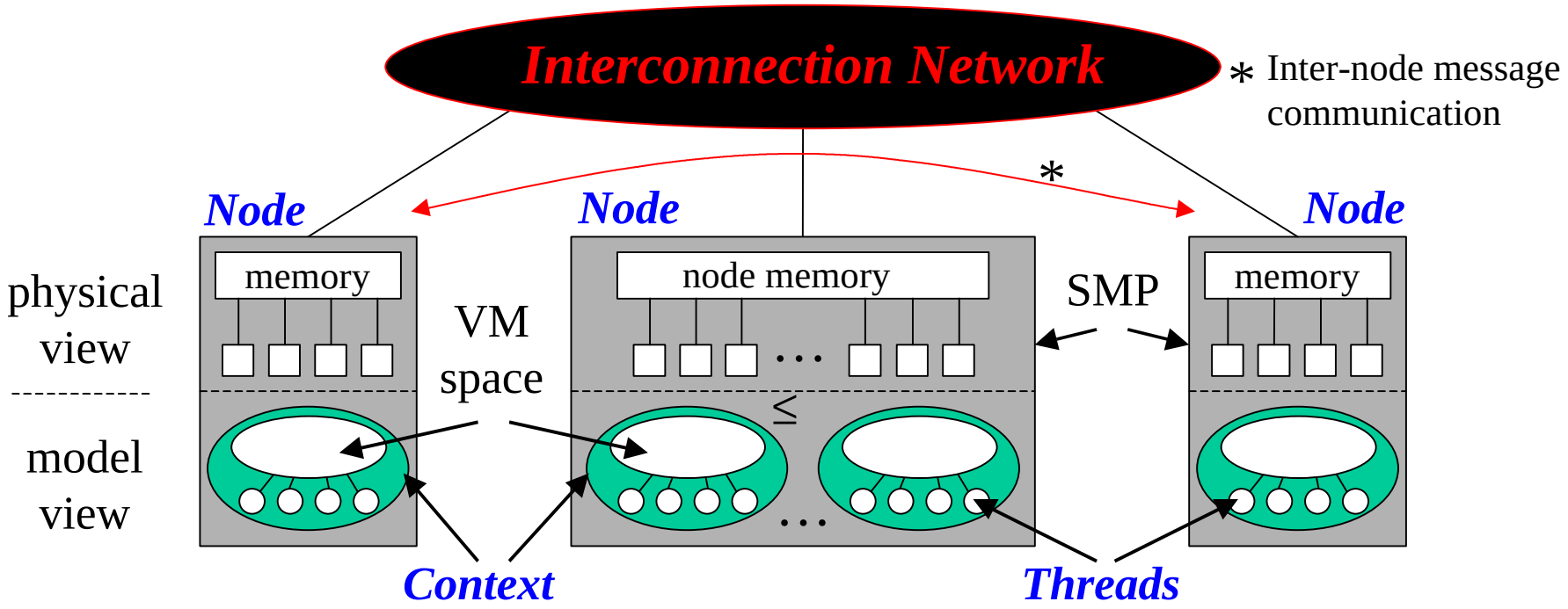
***How do we apply performance technology effectively for the variety and diversity of performance problems that arise in the context of complex parallel and distributed computer systems.***

# *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



# *Framework for Performance Problem Solving*

- Model-based composition
  - Instrumentation / measurement / execution models
    - performance observability constraints
    - performance data types and events
  - Analysis / presentation model
    - performance data processing
    - performance views and model mapping
  - Integration model
    - performance tool component configuration / integration
- ***Can performance problem solving framework be designed based on general complex system model?***

# *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)*

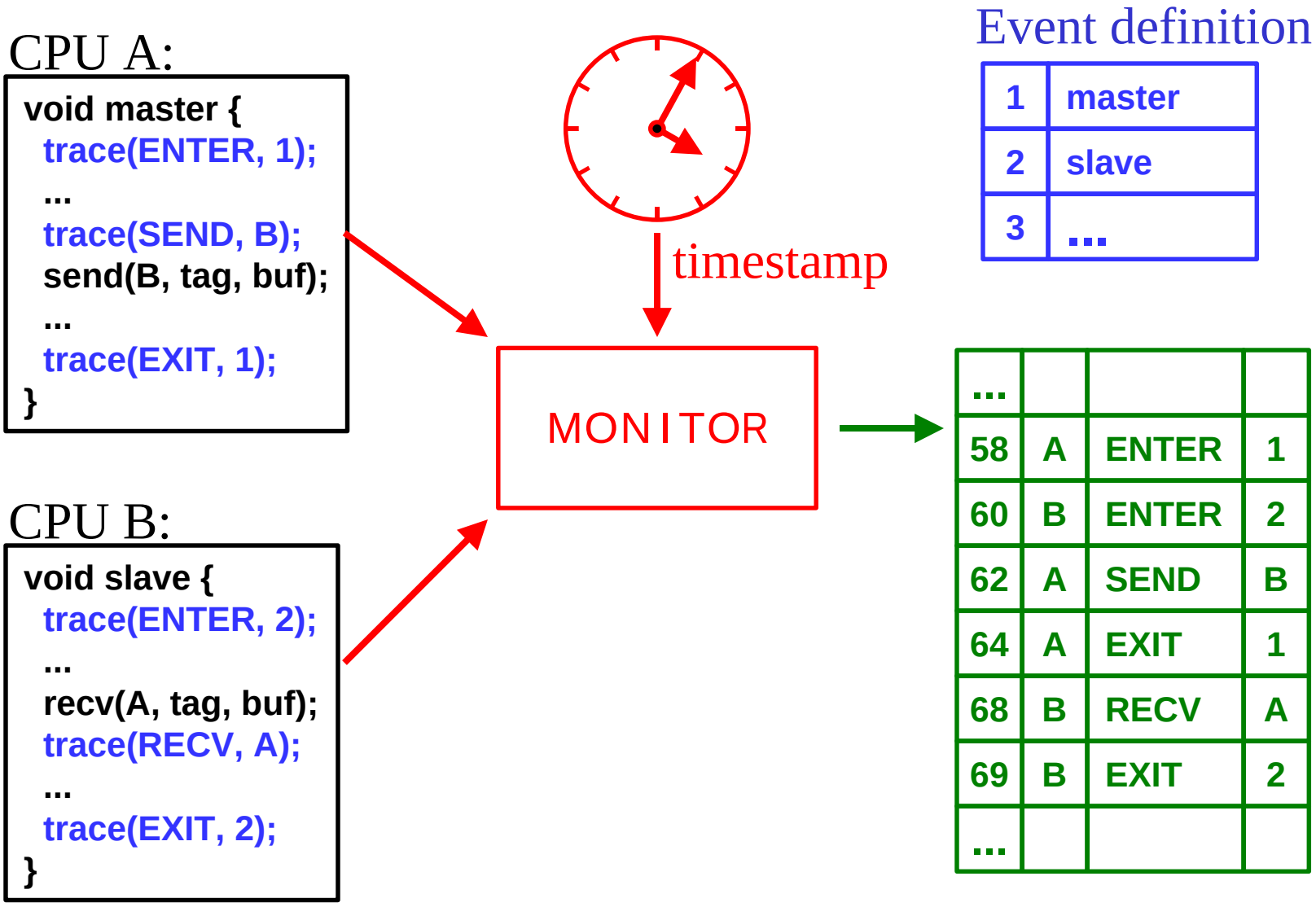
- Dynamic code instrumentation
  - a debugger-like instrumentation approach
  - executable code instrumentation on running program
  - DynInst and DPCL are examples

  +/– 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

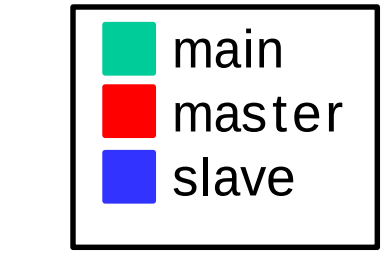
  – can only record information about library entities

# *Event Tracing: Instrumentation, Monitor, Trace*

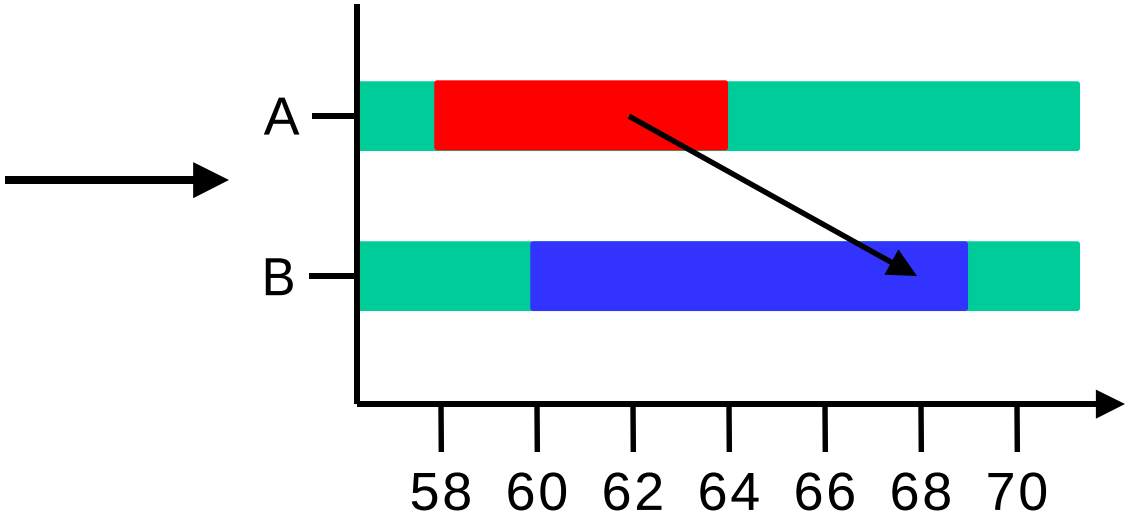


# *Event Tracing: "Timeline" Visualization*

| | |
|---|---|
| 1 | master |
| 2 | slave |
| 3 | ... |



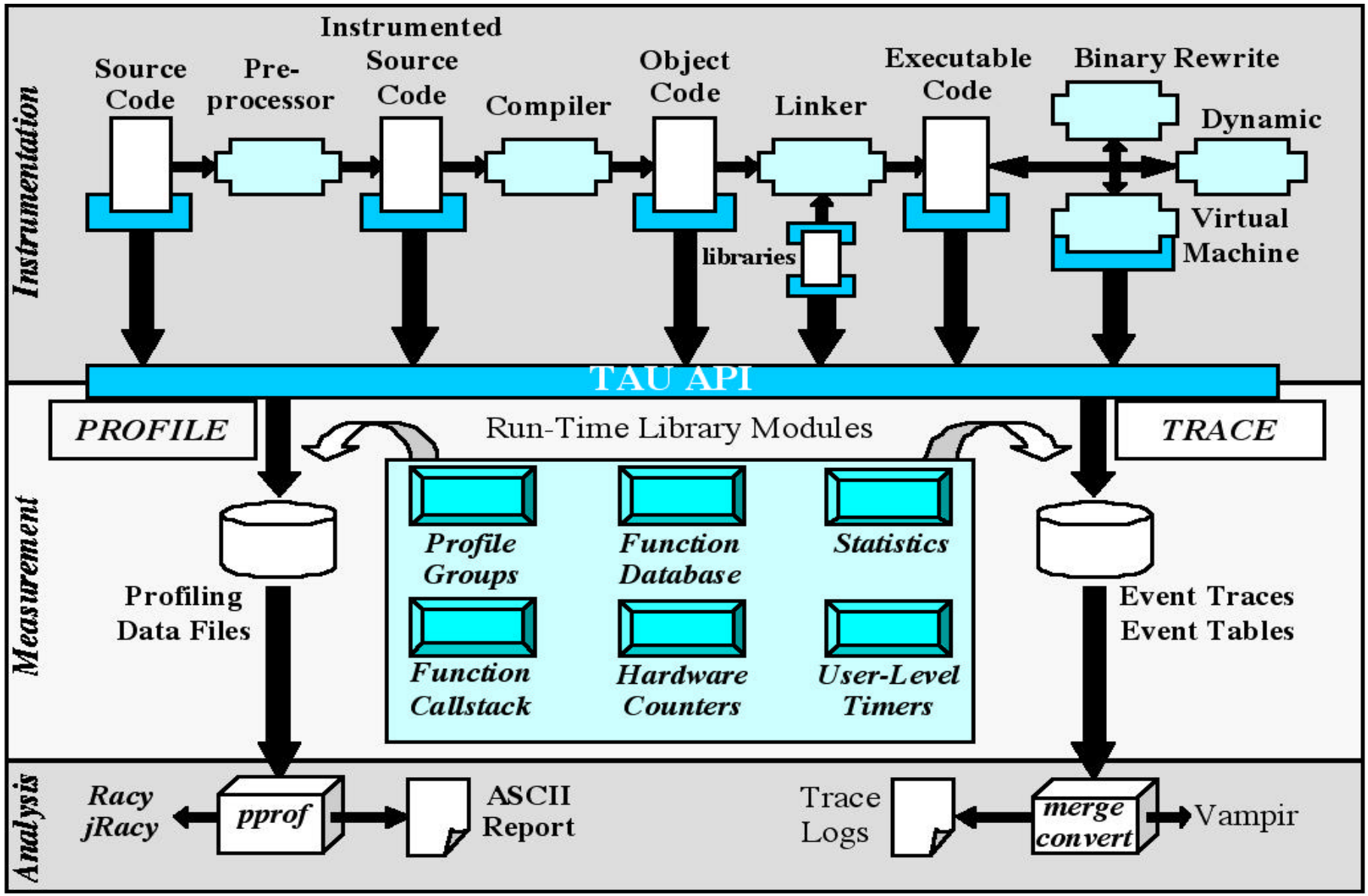

| | | | |
|---|---|---|---|
| ... | | | |
| 58 | A | ENTER | 1 |
| 60 | B | ENTER | 2 |
| 62 | A | SEND | B |
| 64 | A | EXIT | 1 |
| 68 | B | RECV | A |
| 69 | B | EXIT | 2 |
| ... | | | |



# *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)*
  - Object code
    - pre-instrumented libraries (e.g., MPI using PMPI)
    - statically linked
    - dynamically linked
    - fast breakpoints (compiler generated)
  - Executable code
    - dynamic instrumentation (pre-execution) using *DynInstAPI*

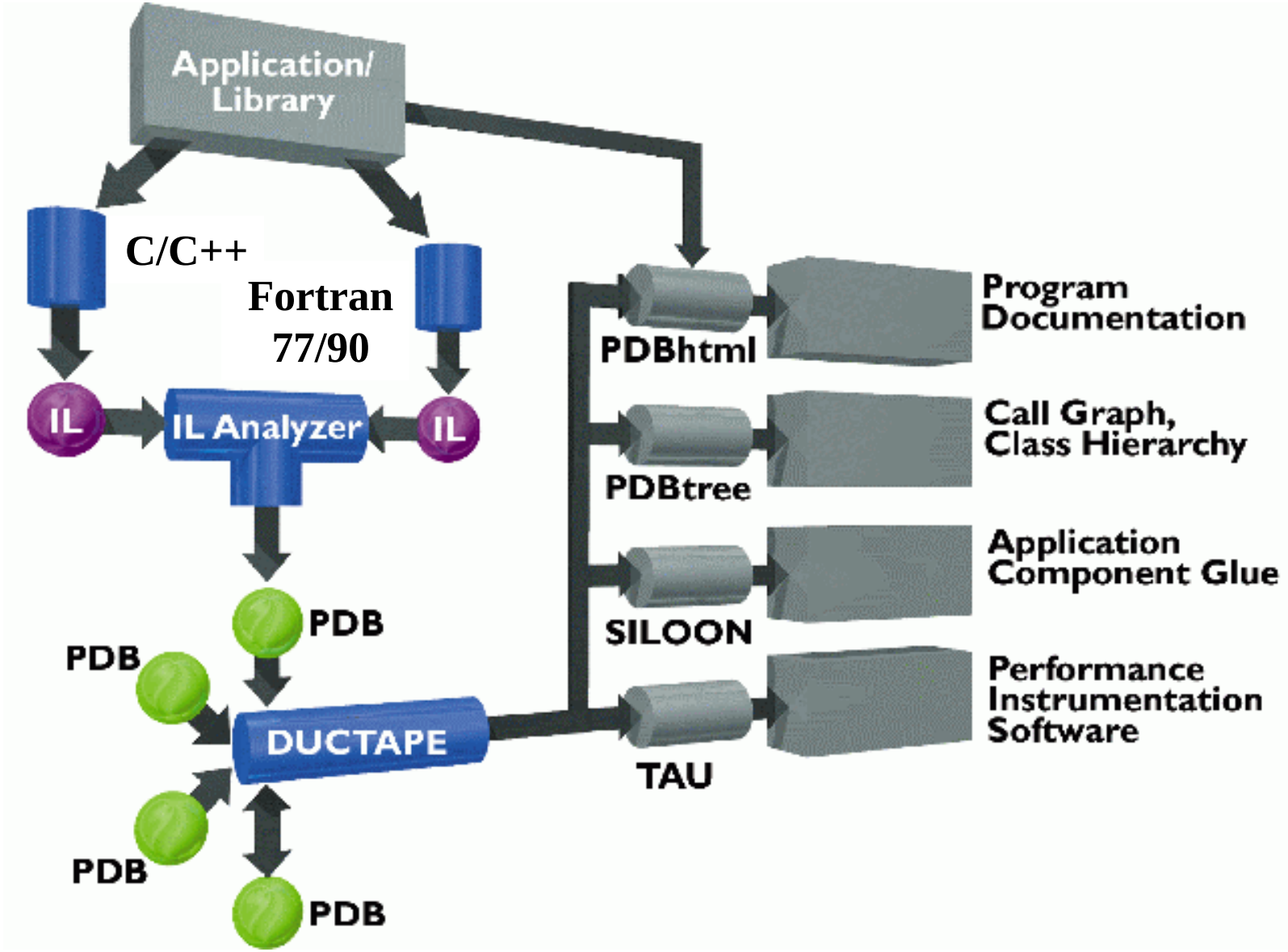
# *TAU Instrumentation (continued)*

- 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 TAU-OPTIONS]

| | |
|---|---|
| {-pthread, -sproc} | Use pthread or SGI sproc threads |
| -smarts | Use SMARTS API for threads |
| -openmp | Use OpenMP threads |
| -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 |
| -TRACE | Generate TAU event traces |
| -PROFILE | Generate TAU profiles |
| -PROFILECOUNTERS | Use hardware performance counters |
| -SGITIMERS | Use fast nsec timers on SGI systems |
| -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);
- 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();

# *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)*

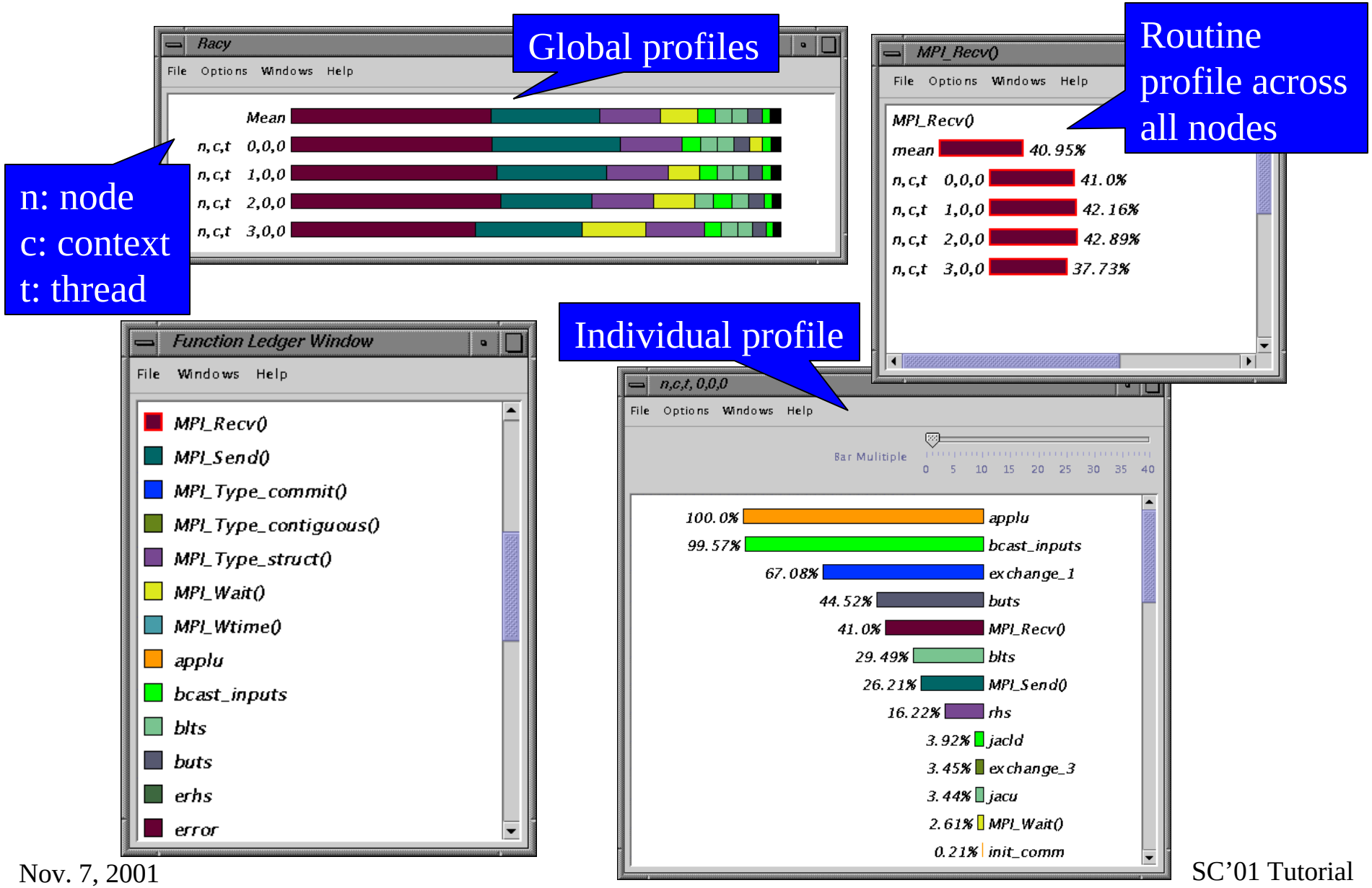
- Intel Quad PIII Xeon, RedHat, PGI F90
- F90 + MPICH
- Profile for:
  - Node
  - Context
  - Thread
- Application events and MPI events

emacs@neutron.cs.uoregon.edu

Buffers Files Tools Edit Search Mule Help

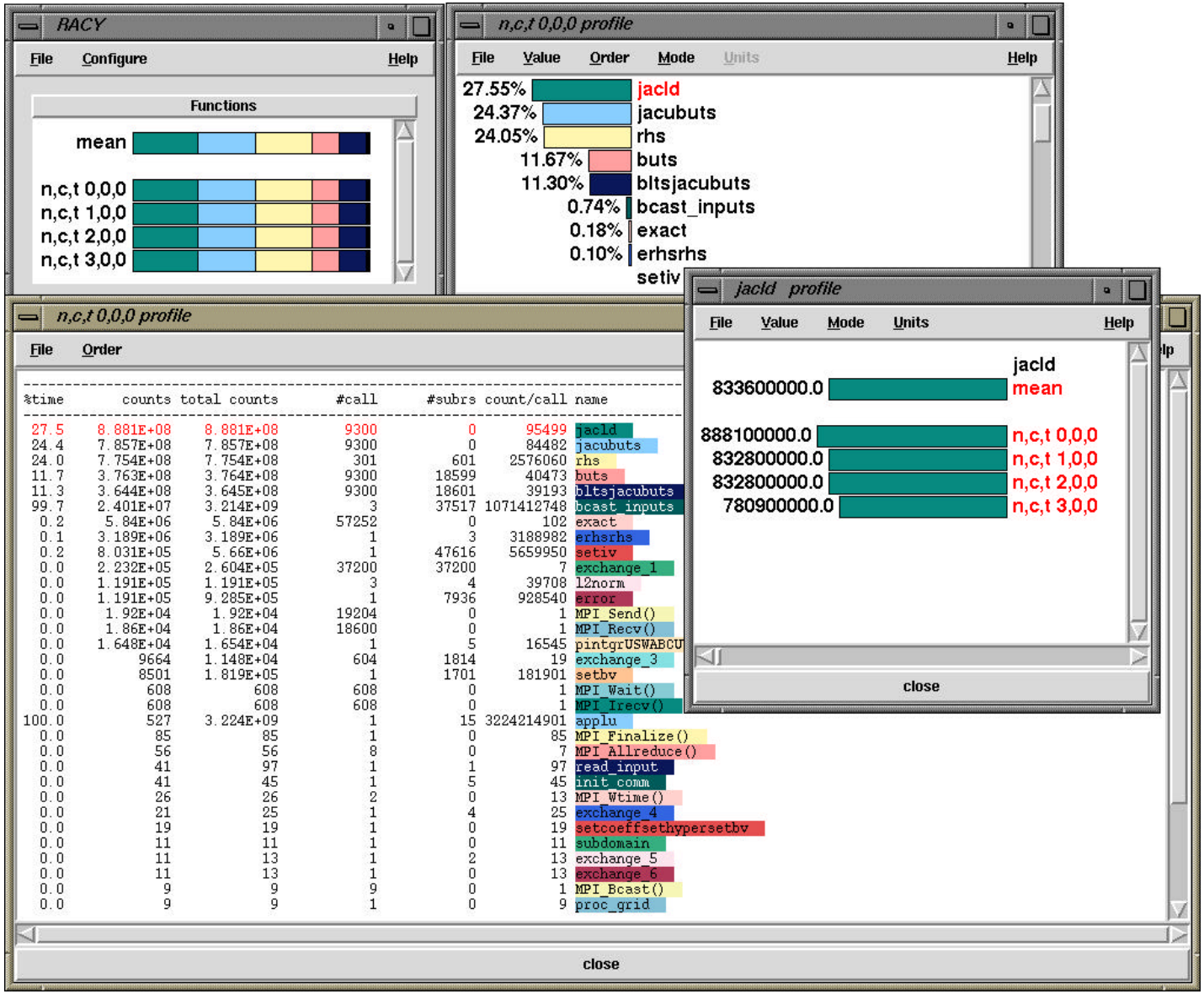
```
Reading Profile files in profile.*

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)*



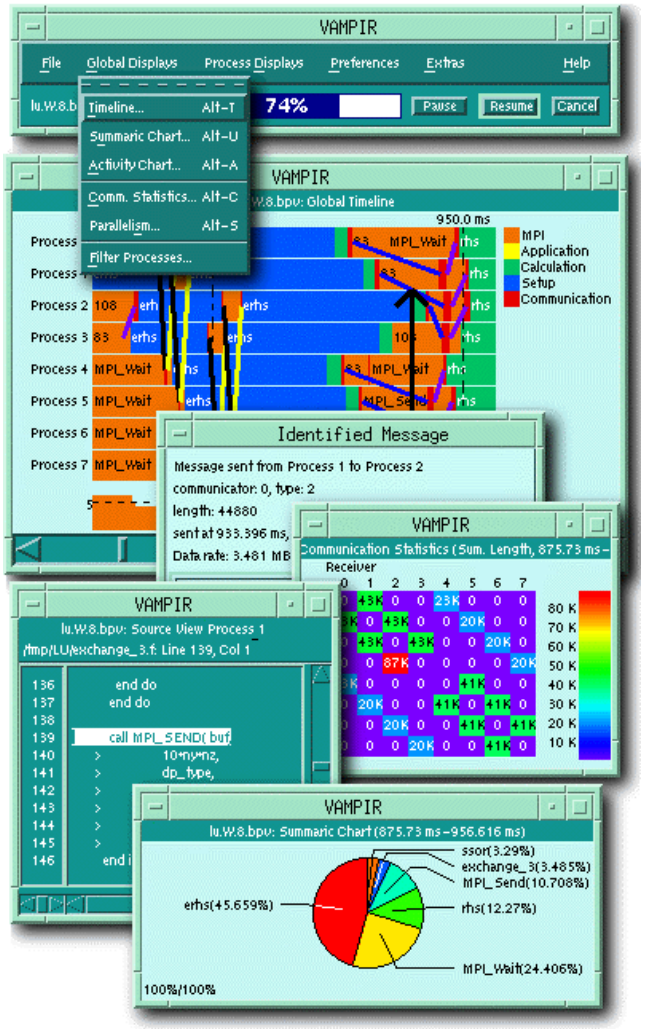
# TAU and PAPI (NAS Parallel Benchmark – LU )

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



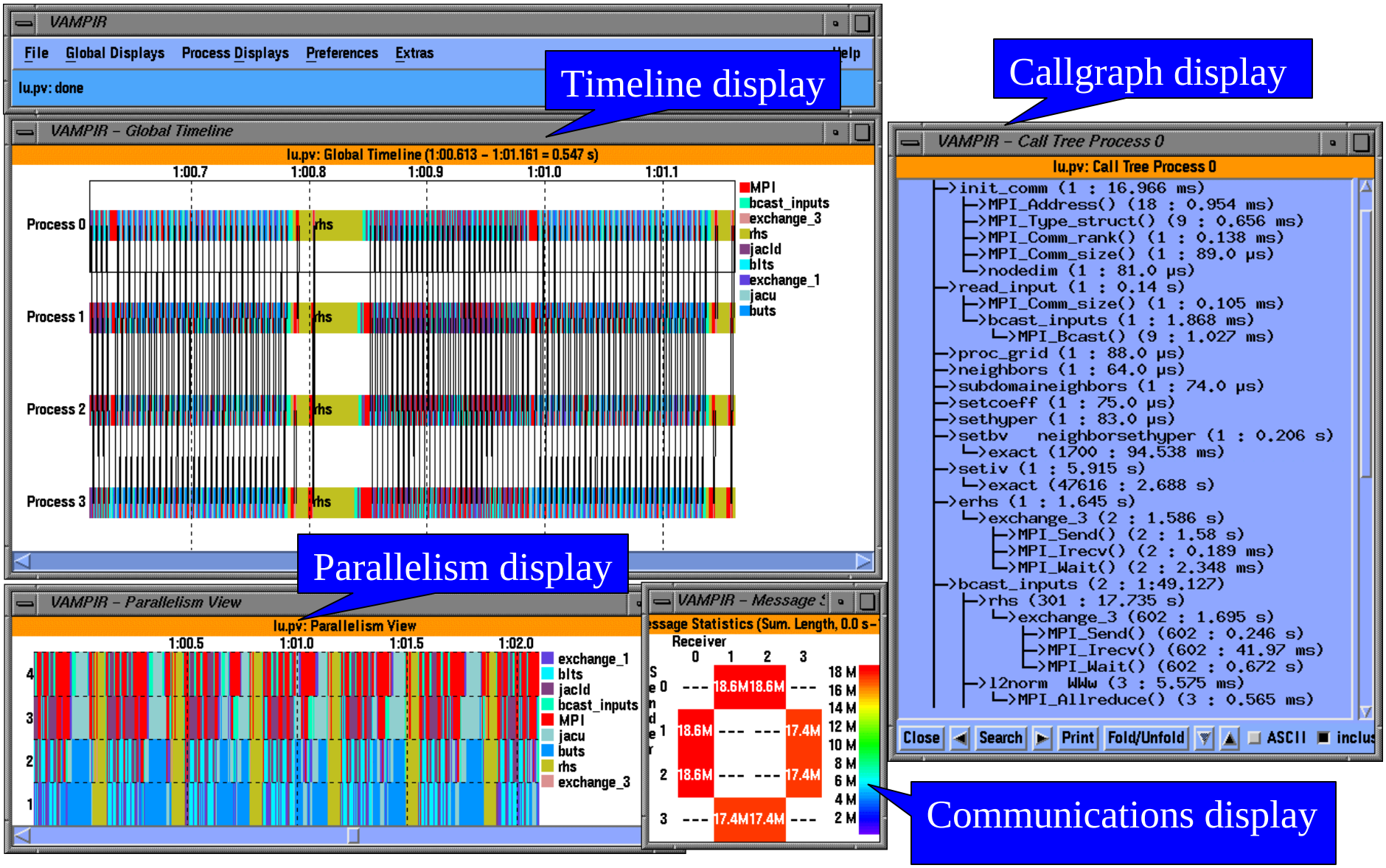
# *Vampir Trace Visualization Tool*

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



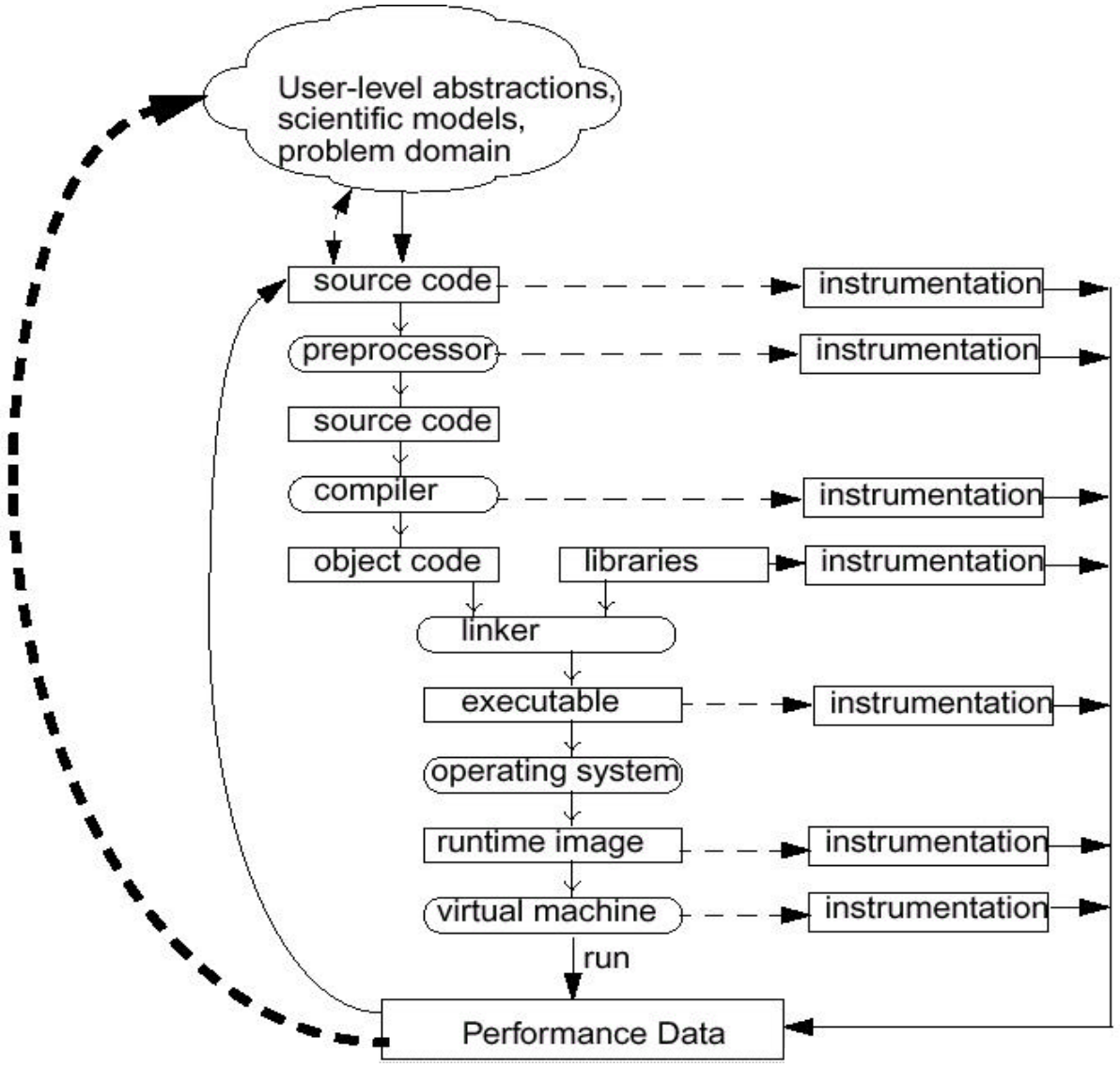

- **http://www.pallas.de/pages/vampir.htm**

# *Vampir (NAS Parallel Benchmark – LU)*



# *Semantic Performance Mapping*

- Associate performance measurements with high-level semantic abstractions
- Need mapping support in the performance measurement system to assign data correctly



# *Semantic Entities/Attributes/Associations (SEAA)*

- New dynamic mapping scheme (S. Shende, Ph.D. thesis)
  - Contrast with ParaMap (Miller and Irvin)
  - Entities defined at any level of abstraction
  - Attribute entity with semantic information
  - Entity-to-entity associations
- Two association types (implemented in TAU API)
  - Embedded – extends data structure of associated object to store performance measurement entity
  - External – creates an external look-up table using address of object as the key to locate performance measurement entity

# *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]);
}
```

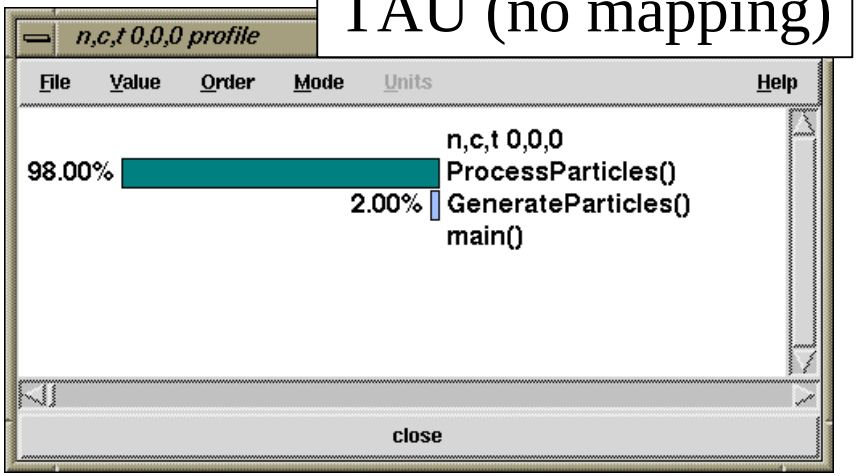
- How much time is spent processing face *i* particles?
- What is the distribution of performance among faces?

# *No Performance Mapping versus Mapping*

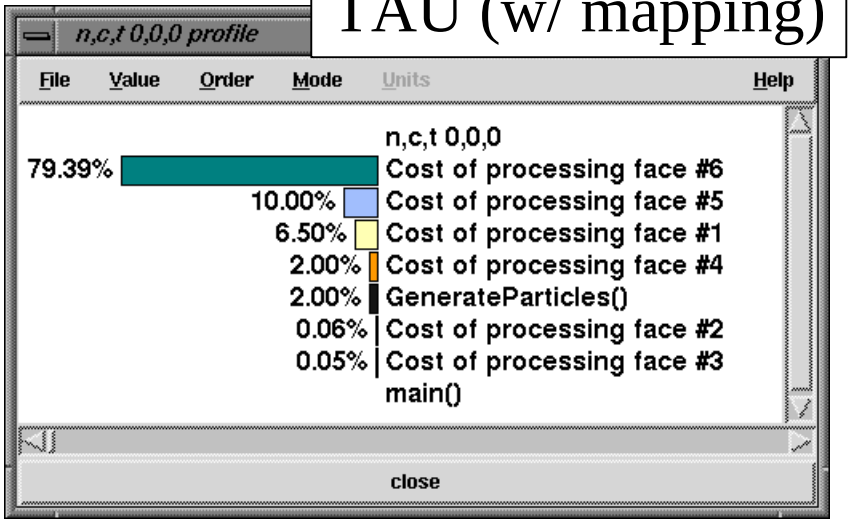
- Typical performance tools report performance with respect to routines
- Does not provide support for mapping

- Performance tools with SEAA mapping can observe performance with respect to scientist's programming and problem abstractions

TAU (no mapping)




TAU (w/ mapping)



# *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

# *Performance Technology for Complex Parallel Systems*

## *Part 2 – Complexity Scenarios*

*Sameer Shende*

cs.uoregon.edu
COMPUTER & INFORMATION SCIENCE • UNIVERSITY OF OREGON

# *Goals*

- Explore performance analysis issues in different parallel computing and programming contexts
- Demonstrate TAU's usage in different complex parallel system contexts and application case studies
- Explore how to bridge the semantic gap between performance data that tools capture, and user and system programming and execution abstractions
- Highlight TAU performance mapping API
  - C++ template instrumentation
  - Hierarchical software systems
- Discuss TAU performance system evolution

# *Complexity Scenarios*

- Message passing computation
  - Observe message communication events
  - Associate messaging events with program events
  - SPMD applications with multiple processes
  - SIMPLE hydrodynamics application (C, MPI)
- Multi-threaded computation
  - (Abstract) thread-based performance measurement
  - Multi-threaded parallel execution
  - Asynchronous runtime system scheduling
  - Multi-threading performance analysis in Java

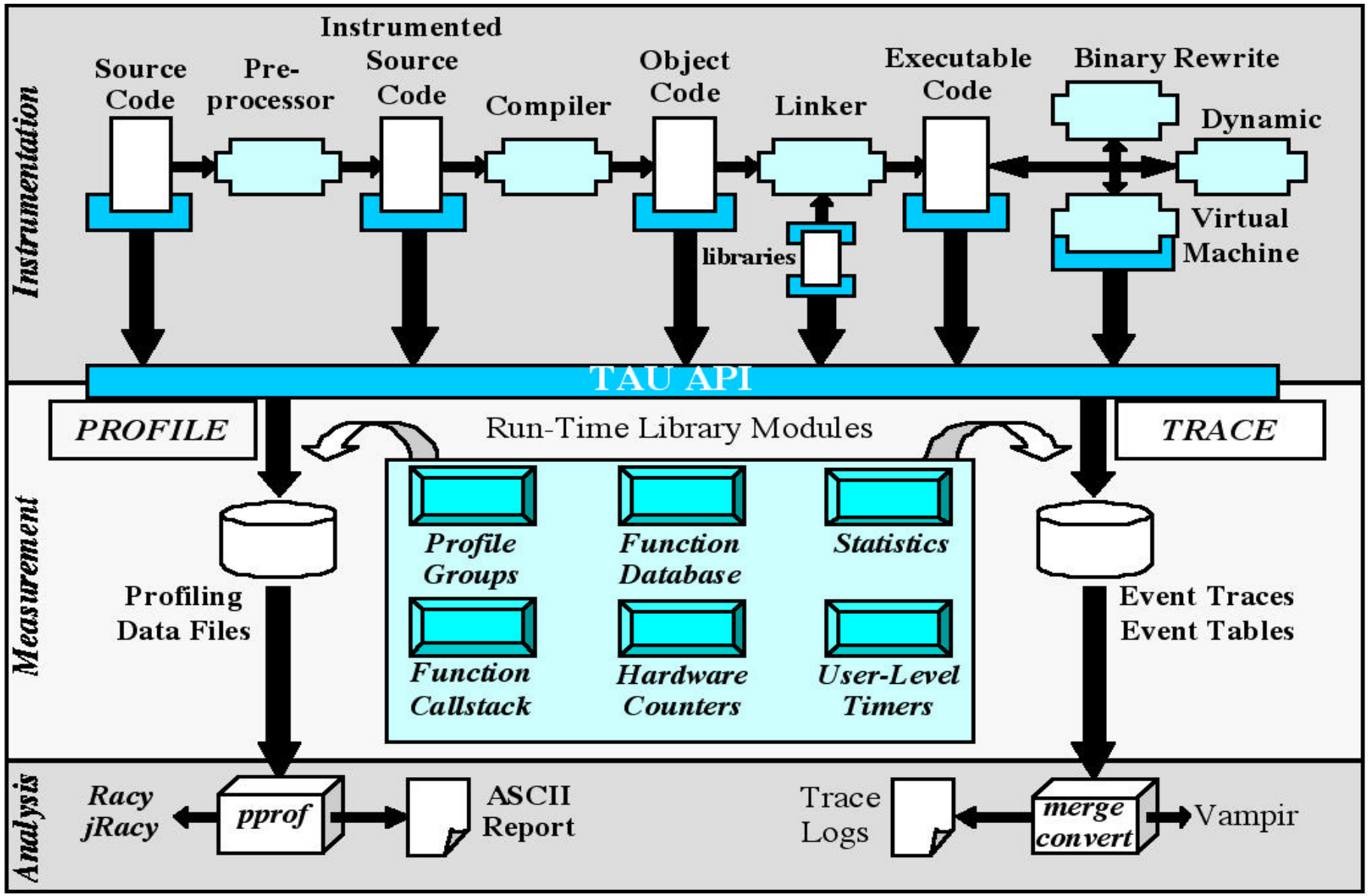
# *Complexity Scenarios (continued)*

- Mixed-mode parallel computation
  - Portable shared memory and message passing APIs
  - Integrate messaging events with multi-threading events
  - OpenMP + MPI and Java + MPI case studies
- Object-oriented programming and C++
  - Object-based performance analysis
  - Performance measurement of template-derived code
  - Array classes and expression transformation
- Hierarchical parallel software frameworks
  - Multi-level software framework and work scheduling
  - Module-specific performance mapping

# *Strategies for Empirical Performance Evaluation*

- Empirical performance evaluation as a series of performance experiments
  - Experiment trials describing instrumentation and measurement requirements
  - Where/When/How axes of empirical performance space
    - where are performance measurements made in program
    - when is performance instrumentation done
    - how are performance measurement/instrumentation chosen
- Strategies for achieving flexibility and portability goals
  - Limited performance methods restrict evaluation scope
  - Non-portable methods force use of different techniques
  - Integration and combination of strategies

# Multi-Level Instrumentation in TAU



# *Multi-Level Instrumentation*

- Uses multiple instrumentation interfaces
- Shares information: cooperation between interfaces
- Taps information at multiple levels
- Provides selective instrumentation at each level
- Targets a common performance model
- Presents a unified view of execution

# *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, high-res physical, CPU, process virtual

# *SIMPLE Source Instrumentation (Preprocessed)*

```
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)

```
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)

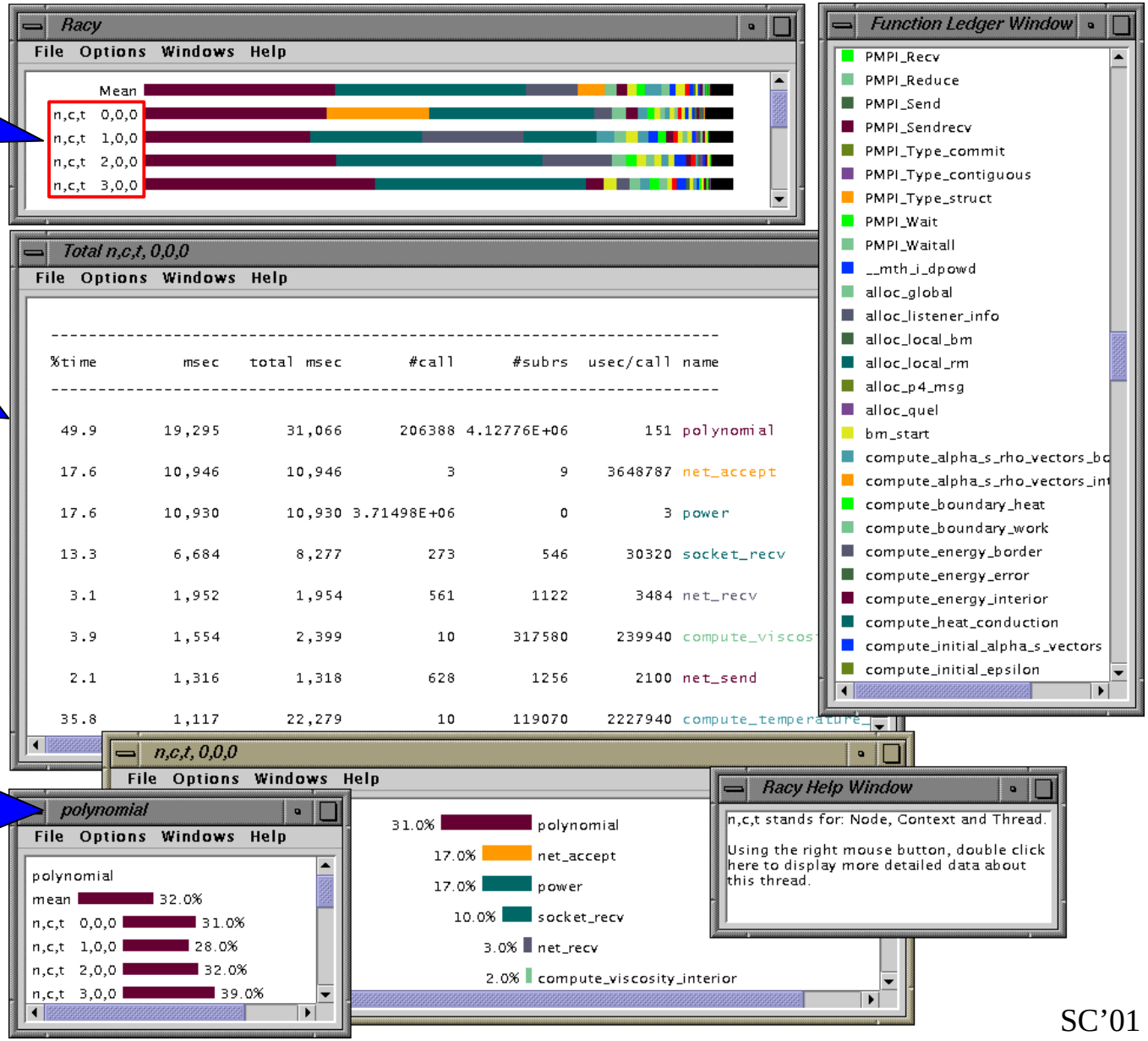
```
int  MPI_Recv(…)
...
{
  int  returnVal, size;
  TAU_PROFILE_TIMER(tautimer, "MPI_Recv()",  " ", TAU_MESSAGE);
  TAU_PROFILE_START(tautimer);
  returnVal = PMPI_Recv(buf, count, datatype, src, tag, comm,
    status);
  if (src != MPI_PROC_NULL && returnVal == MPI_SUCCESS) {
    PMPI_Get_count( status, 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

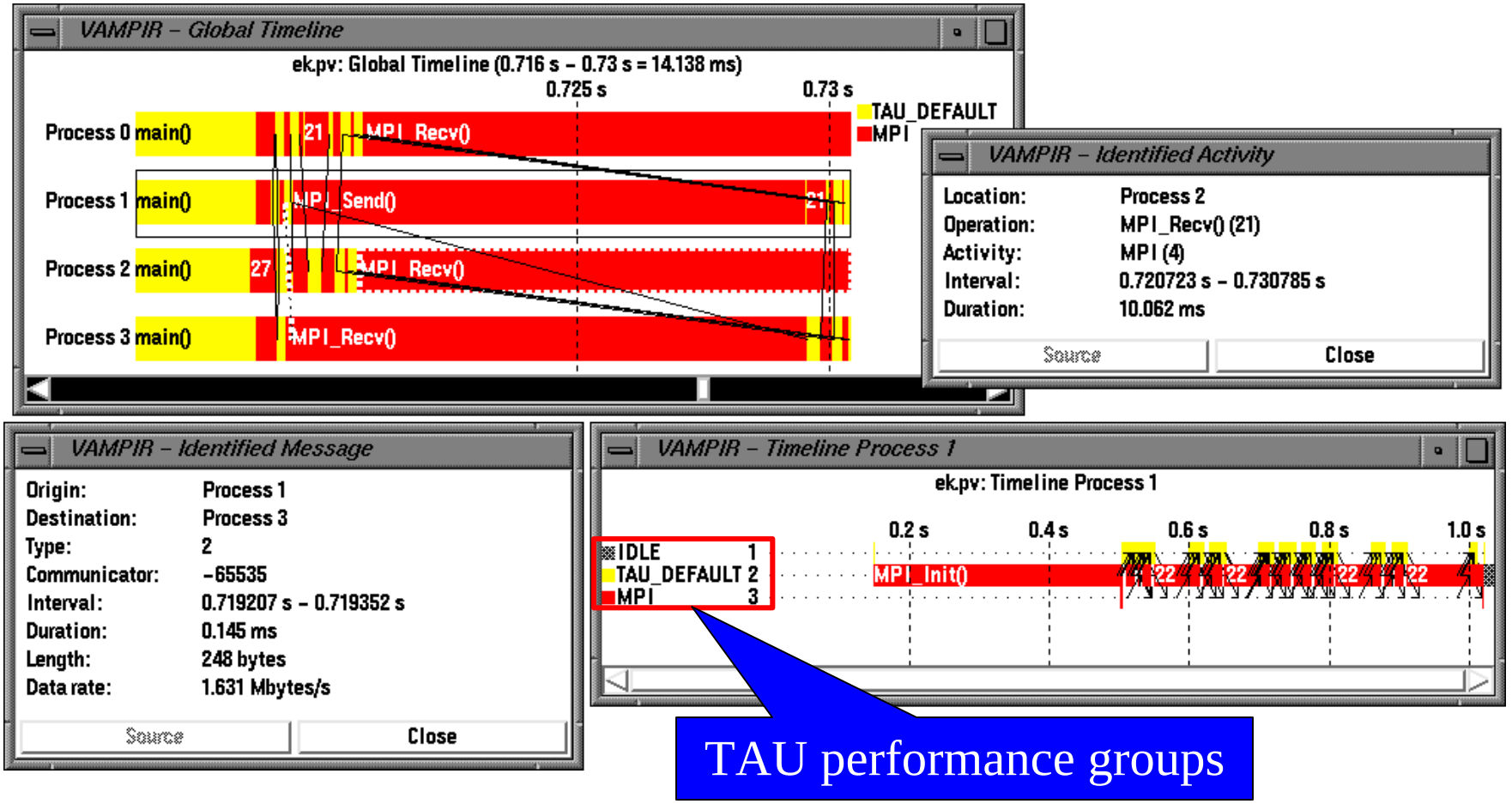
profile per process

global routine profile



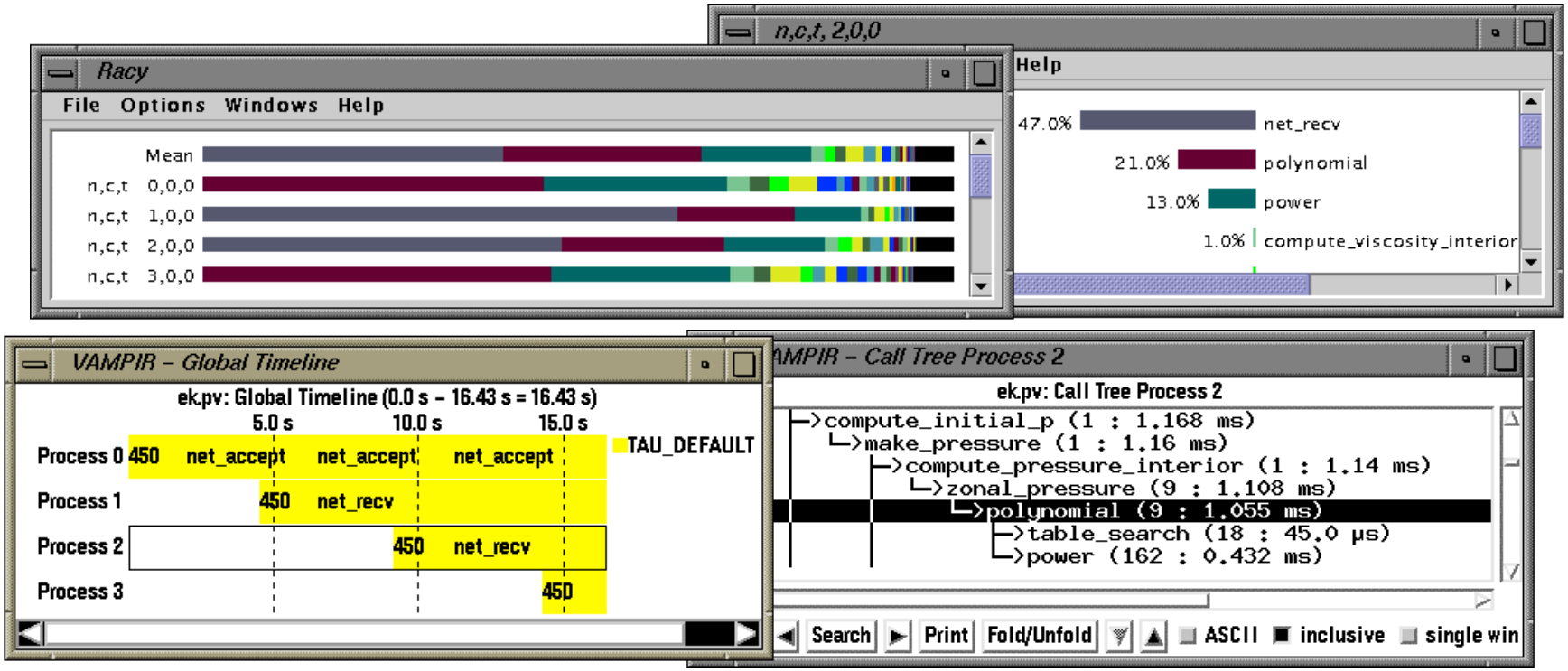
# *Multi-Level Instrumentation (Tracing)*

- No modification of source instrumentation!



# *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



# *Multi-Threading Performance Measurement*

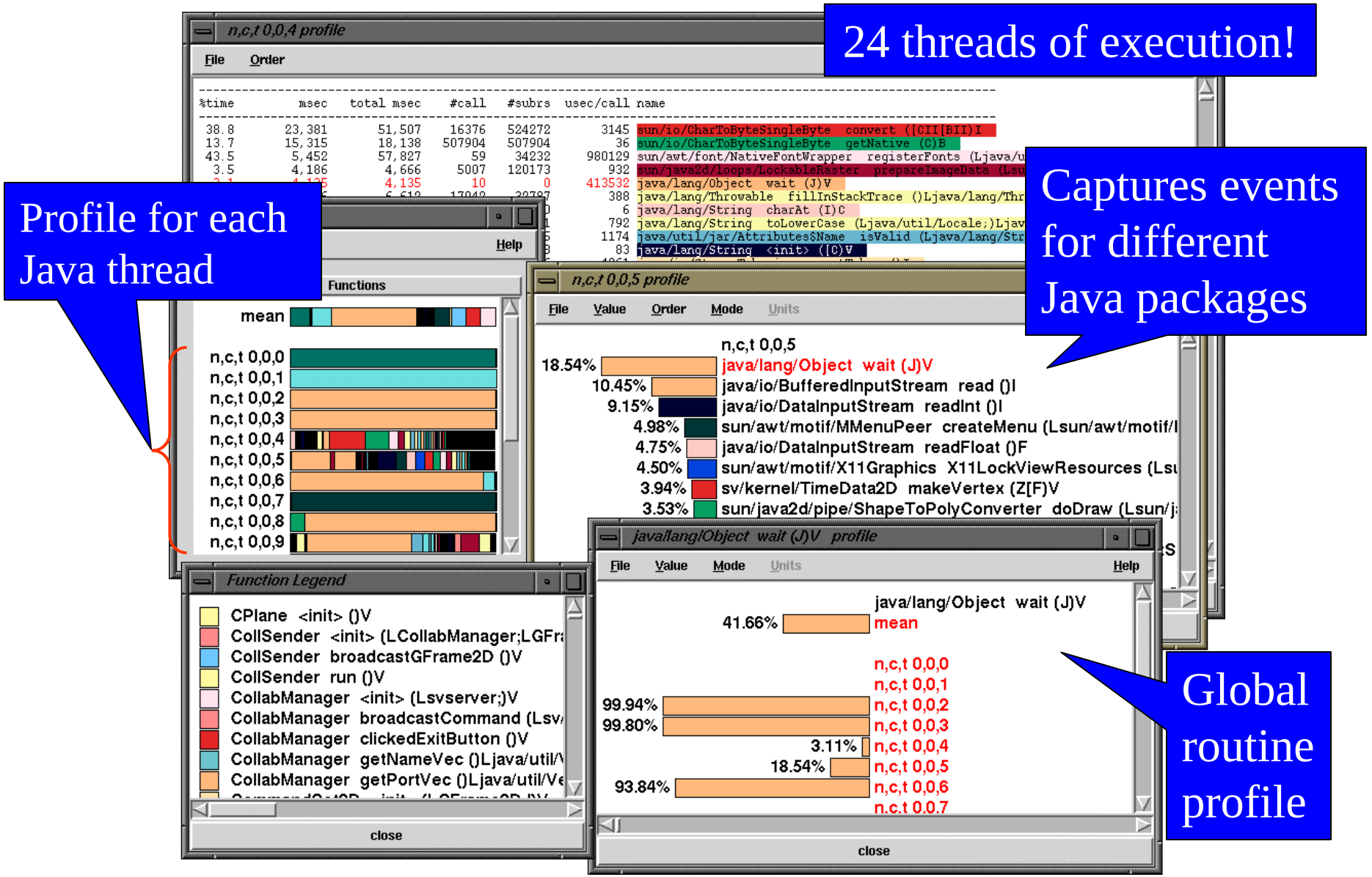
- General issues
  - Thread identity and per-thread data storage
  - Performance measurement support and synchronization
  - Fine-grained parallelism
    - different forms and levels of threading
    - greater need for efficient instrumentation
- TAU general threading and measurement model
  - Common thread layer and measurement support
  - Interface to system specific libraries (reg, id, sync)
- Target different thread systems with core functionality
  - Pthreads, Windows, Java, SMARTS, Tulip, OpenMP

# *Java Multi-Threading Performance (Test Case)*

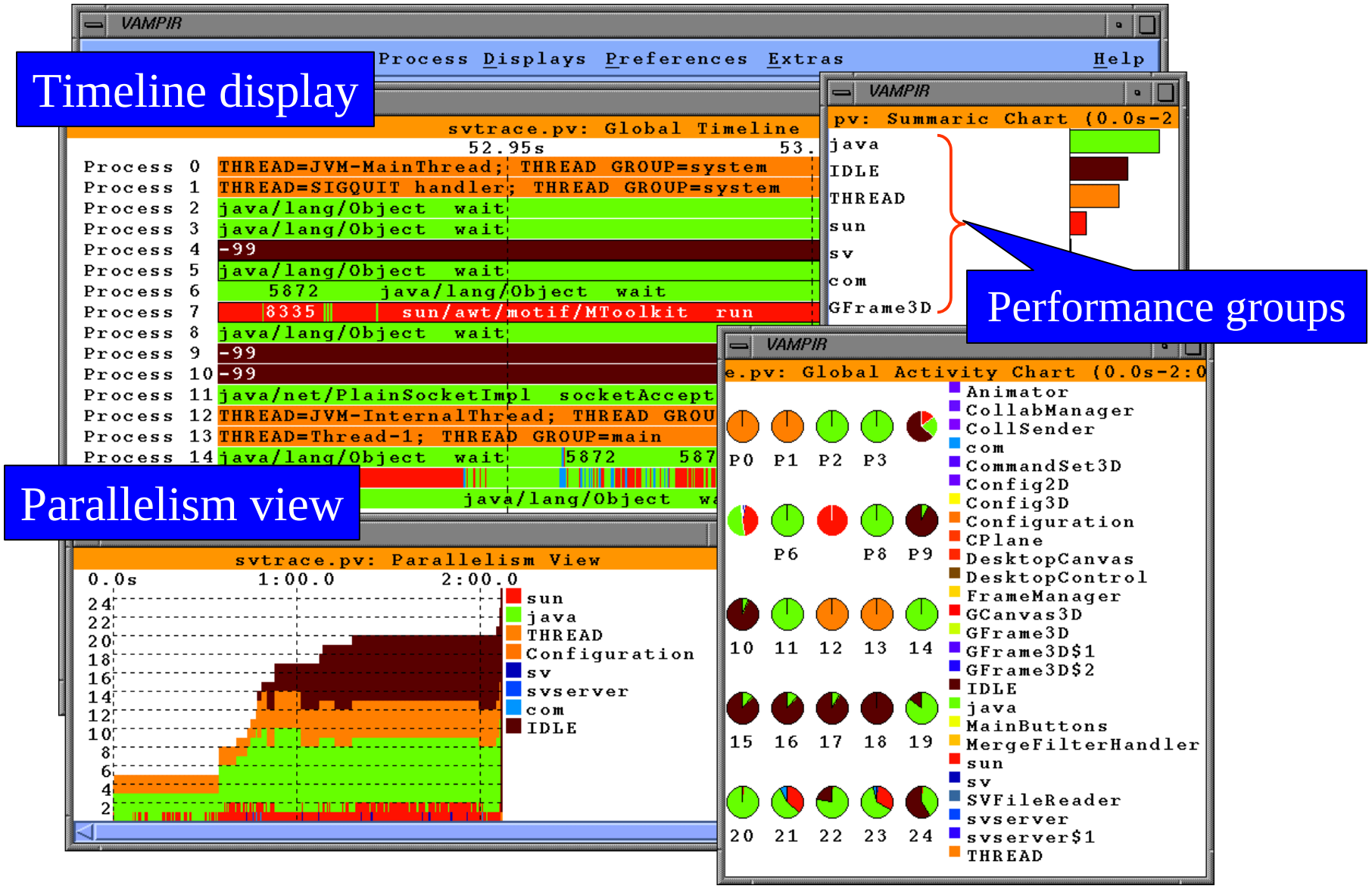
- Profile and trace Java (JDK 1.2+) applications
- Observe user-level and system-level threads
- Observe events for different Java packages
  - /lang, /io, /awt, …
- Test application
  - SciVis, NPAC, Syracuse University

```
% ./configure -jdk=<dir_where_jdk_is_installed>
% setenv LD_LIBRARY_PATH
  $LD_LIBRARY_PATH\:<taudir>/<arch>/lib
% java -XrunTAU svserver
```

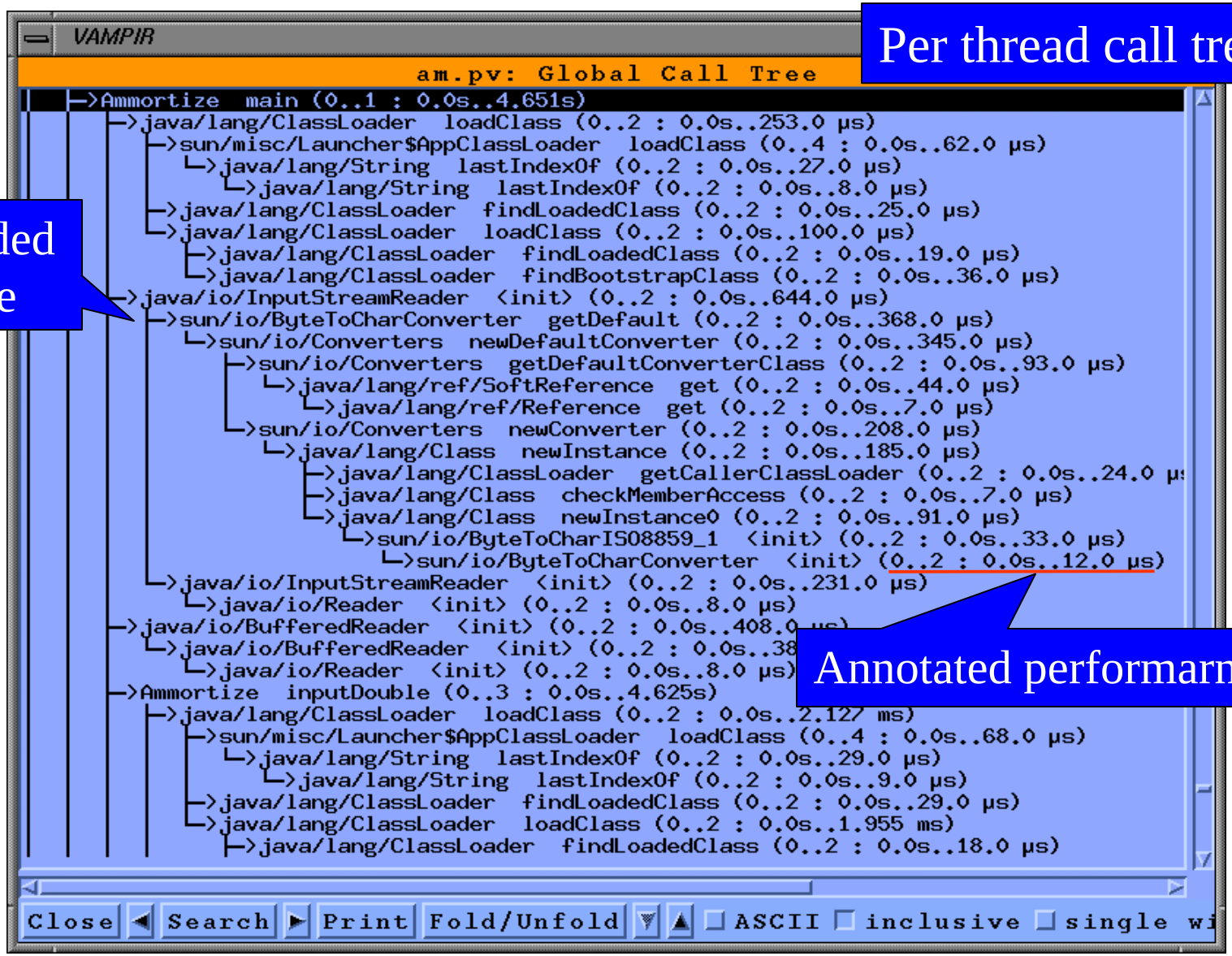
# TAU Profiling of Java Application (SciVis)



# TAU Tracing of Java Application (SciVis)



# *Vampir Dynamic Call Tree View (SciVis)*



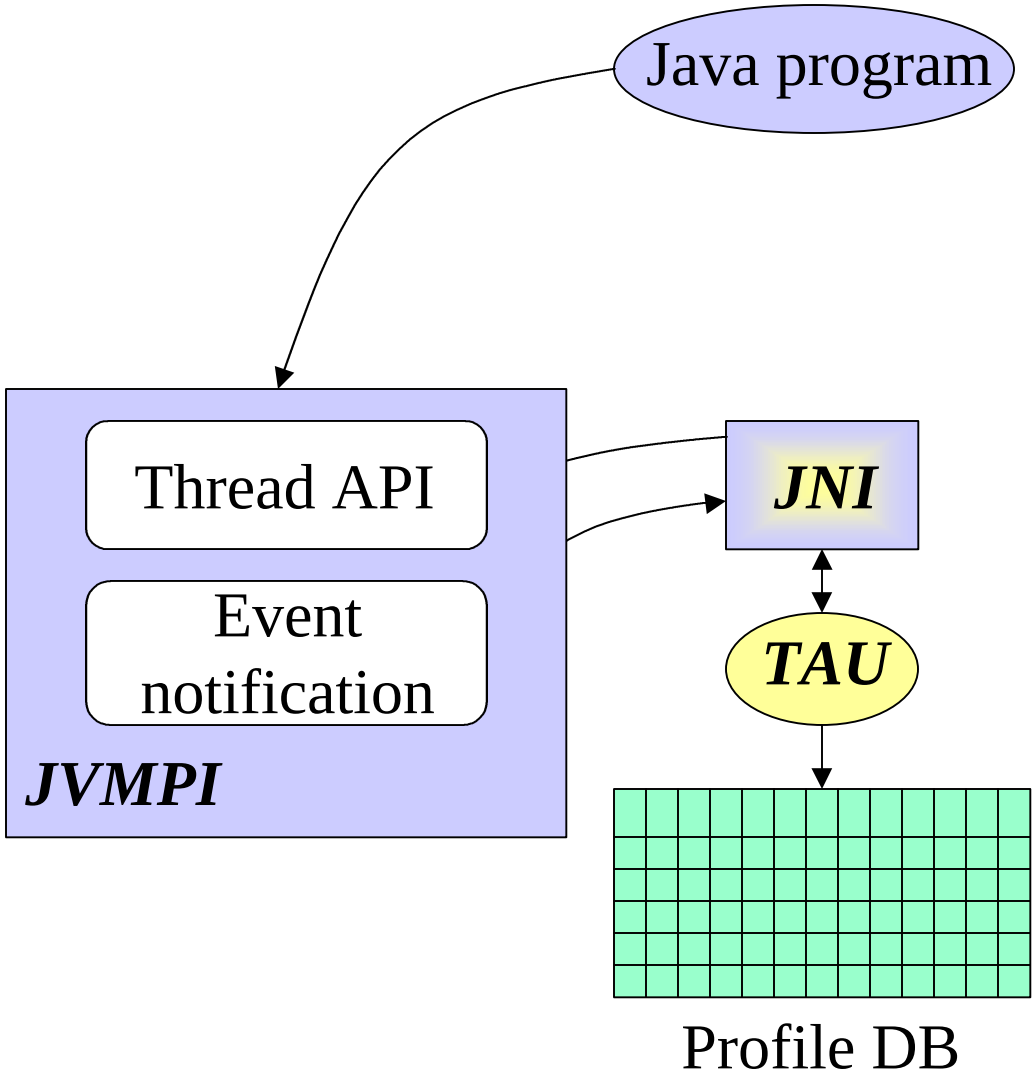
# *Virtual Machine Performance Instrumentation*

- Integrate performance system with VM
  - Captures robust performance data (e.g., thread events)
  - Maintain features of environment
    - portability, concurrency, extensibility, interoperation
  - Allow use in optimization methods
- JVM Profiling Interface (JVMPI)
  - Generation of JVM events and hooks into JVM
  - Profiler agent (TAU) loaded as shared object
    - registers events of interest and address of callback routine
  - Access to information on dynamically loaded classes
  - No need to modify Java source, bytecode, or JVM

# *JVMPI Events*

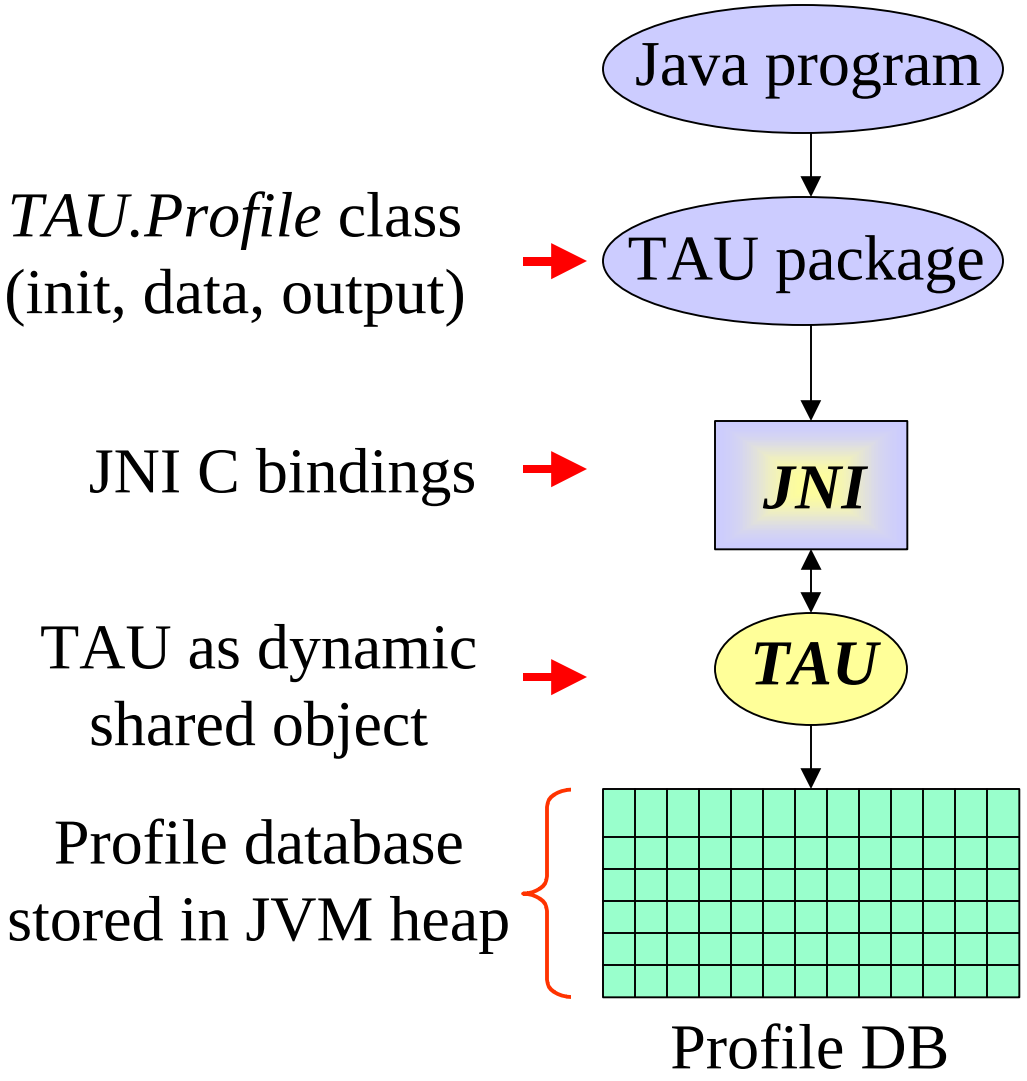
- Method transition events
- Memory events
- Heap arena events
- Garbage collection events
- Class events
- Global reference events
- Monitor events
- Monitor wait events
- Thread events
- Dump events
- Virtual machine events

# TAU Java JVM Instrumentation Architecture




- Robust set of events
- Portability
- Access to thread info
- Measurement options
- Limitations
  - Overhead
  - Many events
  - Event control
  - No user-defined events

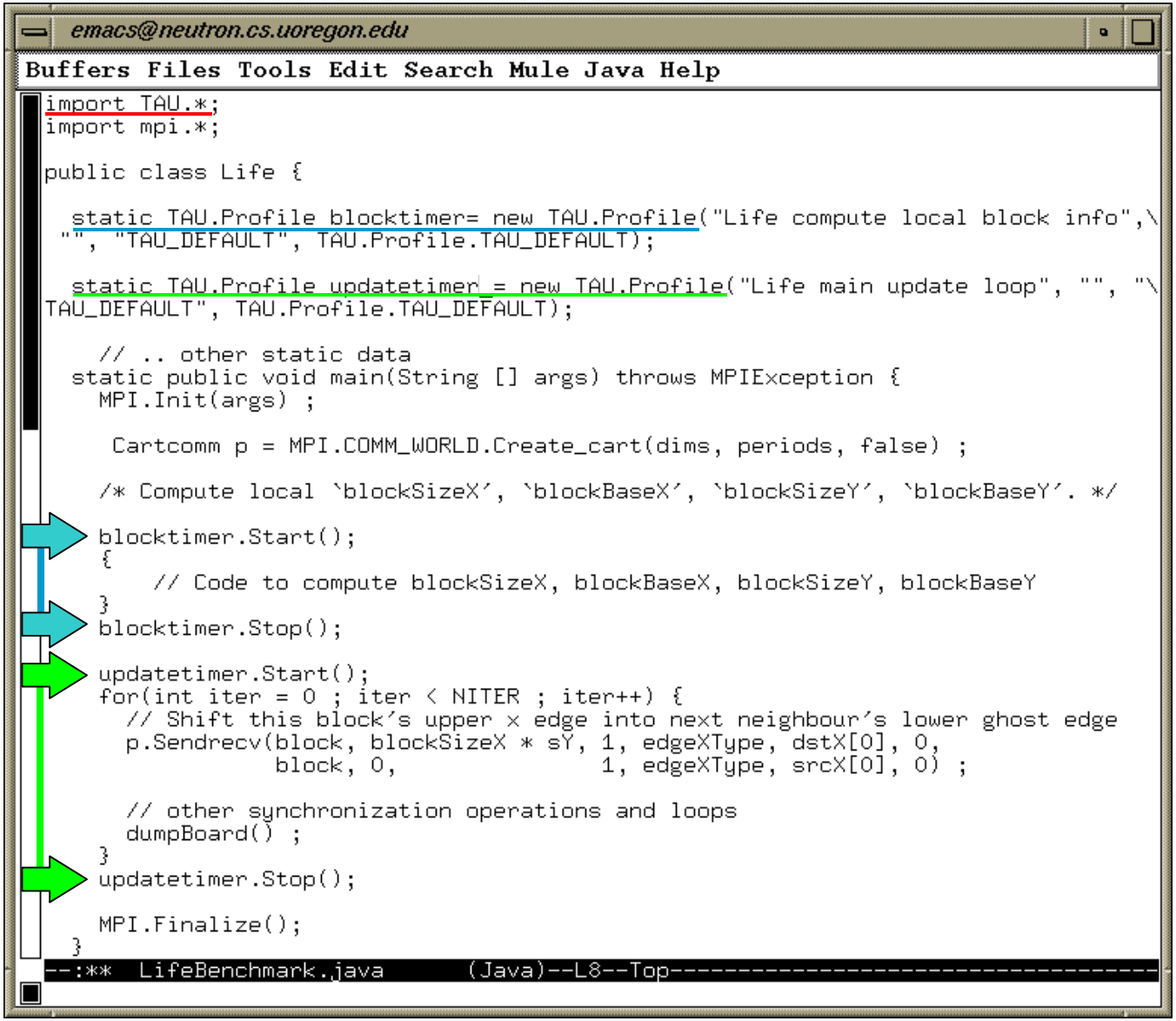
# *TAU Java Source Instrumentation Architecture*




- Any code section can be measured
- Portability
- Measurement options
  - Profiling, tracing
- Limitations
  - Source access only
  - Lack of thread information
  - Lack of node information

# Java Source-Level Instrumentation

- TAU Java package
- User-defined events
- *TAU.Profile* class for new "timers"
  - Start/Stop
- Performance data output at end



# *Mixed-mode Parallel Programs (OpenMPI + MPI)*

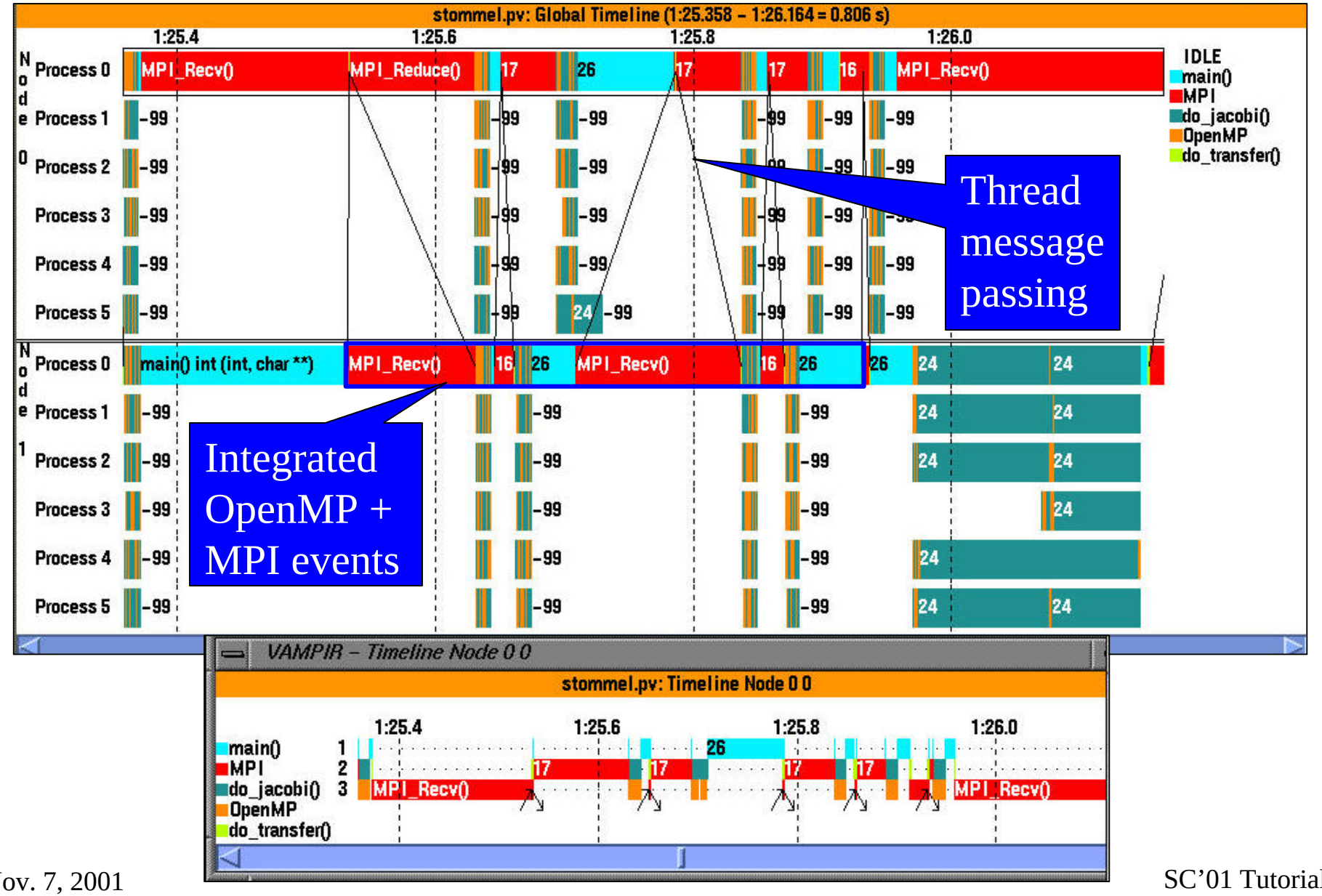
- Portable mixed-mode parallel programming
  - Multi-threaded shared memory programming
  - Inter-node message passing
- Performance measurement
  - Access to runtime system and communication events
  - Associate communication and application events
- 2-Dimensional Stommel model of ocean circulation
  - OpenMP for shared memory parallel programming
  - MPI for cross-box message-based parallelism
  - Jacobi iteration, 5-point stencil
  - Timothy Kaiser (San Diego Supercomputing Center)

# *Stommel Instrumentation*

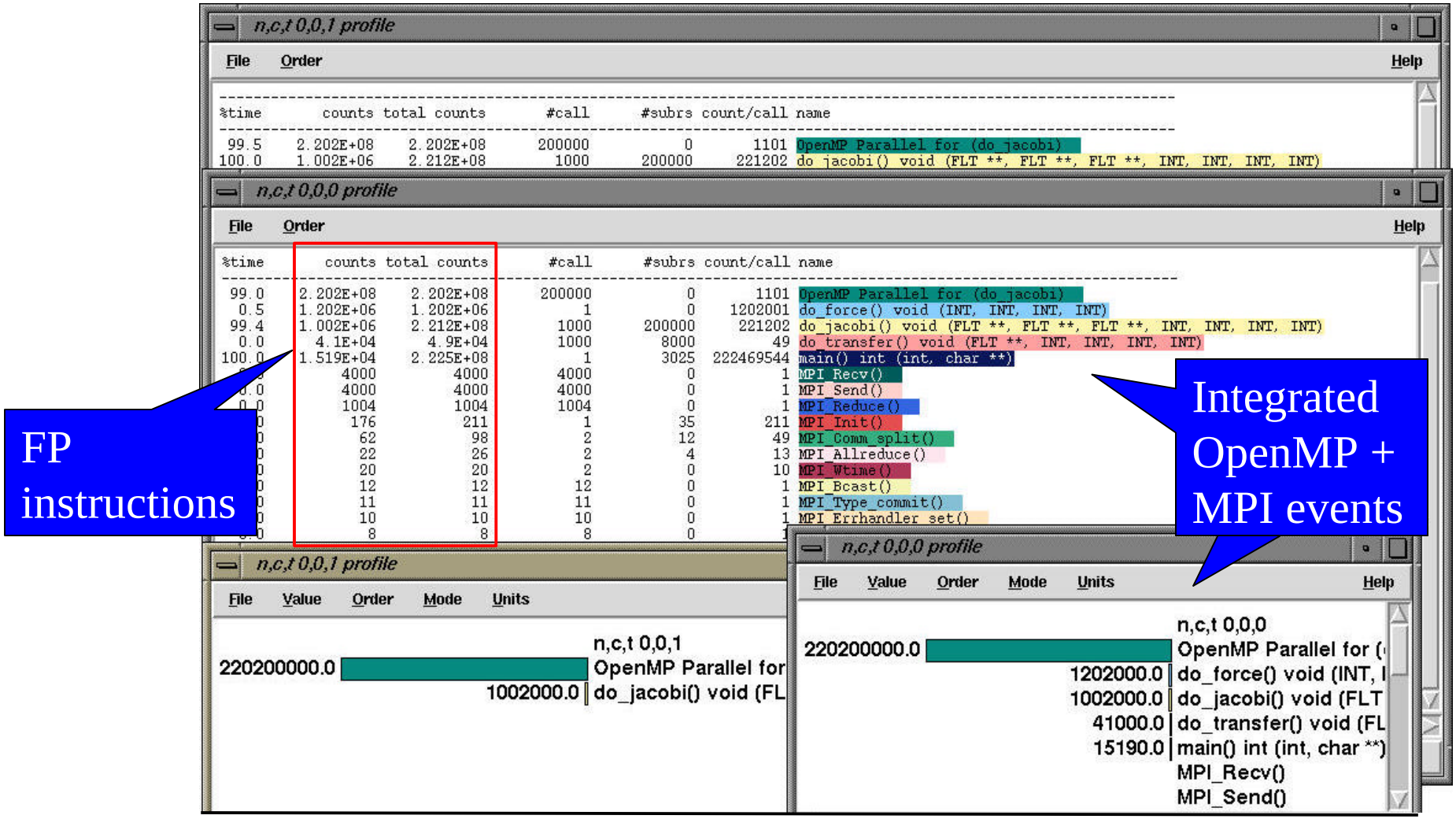
- OpenMP directive instrumentation (see OPARI in Part 3)

```
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"
```

# OpenMP + MPI Ocean Modeling (Trace)



# OpenMP + MPI Ocean Modeling (HW Profile)



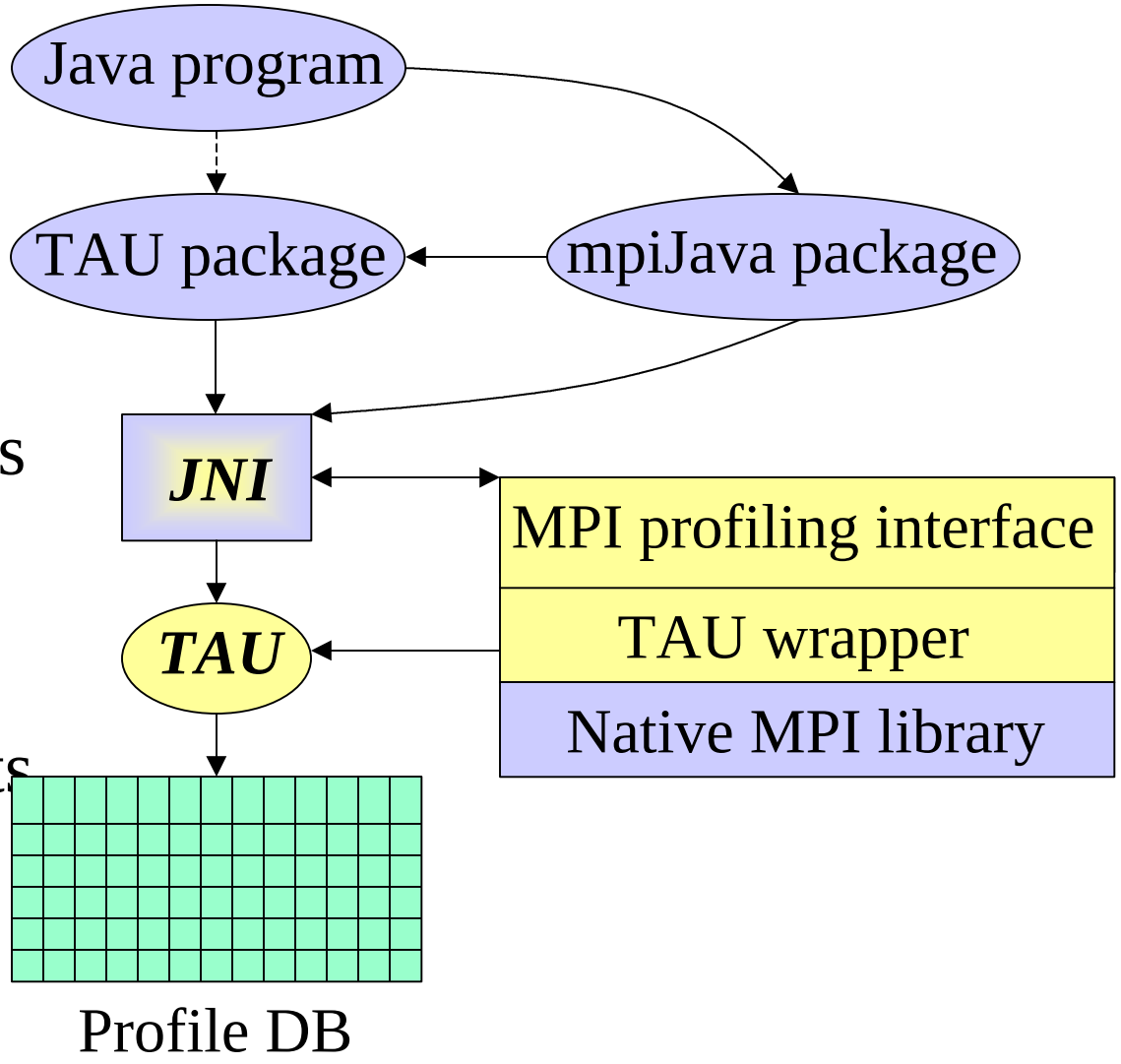

% configure -papi=../packages/papi -openmp -c++=pgCC -cc=pgcc -mpiinc=../packages/mpich/include -mpilib=../packages/mpich/lib

# *Mixed-mode Parallel Programs (Java + MPI)*

- Explicit message communication libraries for Java
- MPI performance measurement
  - MPI profiling interface - link-time interposition library
  - TAU wrappers in native profiling interface library
  - Send/Receive events and communication statistics
- mpiJava (Syracuse, JavaGrande, 1999)
  - Java wrapper package
  - JNI C bindings to MPI communication library
  - Dynamic shared object (*libmpijava.so*) loaded in JVM
  - *prunjava* calls *mpirun* to distribute program to nodes
  - Contrast to Java RMI-based schemes (MPJ, CCJ)

# *TAU mpiJava Instrumentation Architecture*

- No source instrumentation required
- Portability
- Measurement options
- Limitations
  - MPI events only
  - No mpiJava events
  - Node info only
  - No thread info



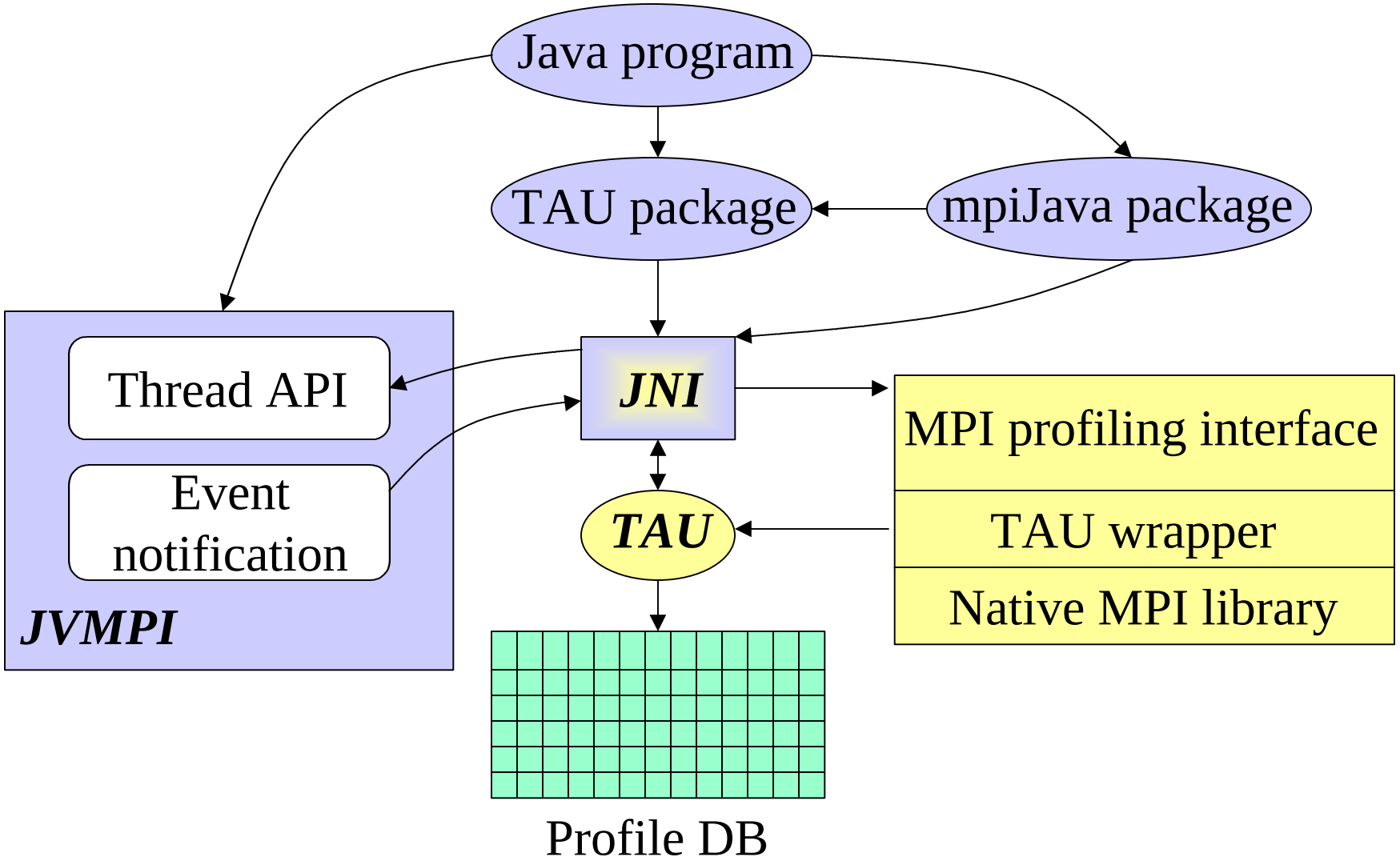
# *Java Multi-threading and Message Passing*

- Java threads and MPI communications
  - Shared-memory multi-threading events
  - Message communications events
- Unified performance measurement and views
  - Integration of performance mechanisms
  - Integrated association of performance events
    - thread event and communication events
    - user-defined (source-level) performance events
    - JVM events
- Requires instrumentation and measurement cooperation

# *Instrumentation and Measurement Cooperation*

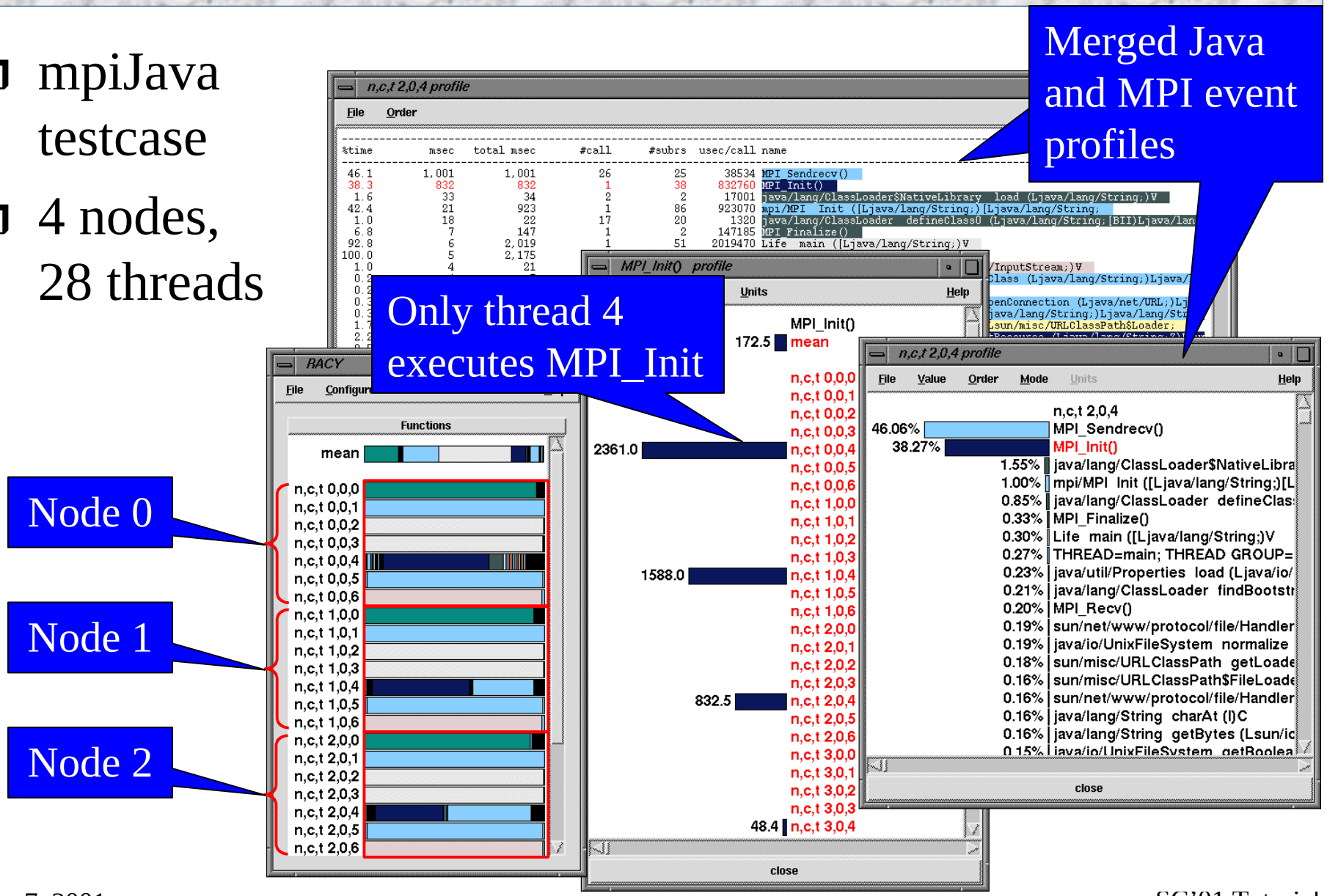
- Problem
  - JVMPI doesn't see MPI events (e.g., rank (node))
  - MPI profiling interfaces doesn't see threads
  - Source instrumentation doesn't see either!
- Need cooperation between interfaces
  - MPI exposes rank and gets thread information
  - JVMPI exposes thread information and gets rank
  - Source instrumentation gets both
  - Post-mortem matching of sends and receives
- Selective instrumentation
  - java -XrunTAU:exclude=java/io,sun

# *TAU Java Instrumentation Architecture*



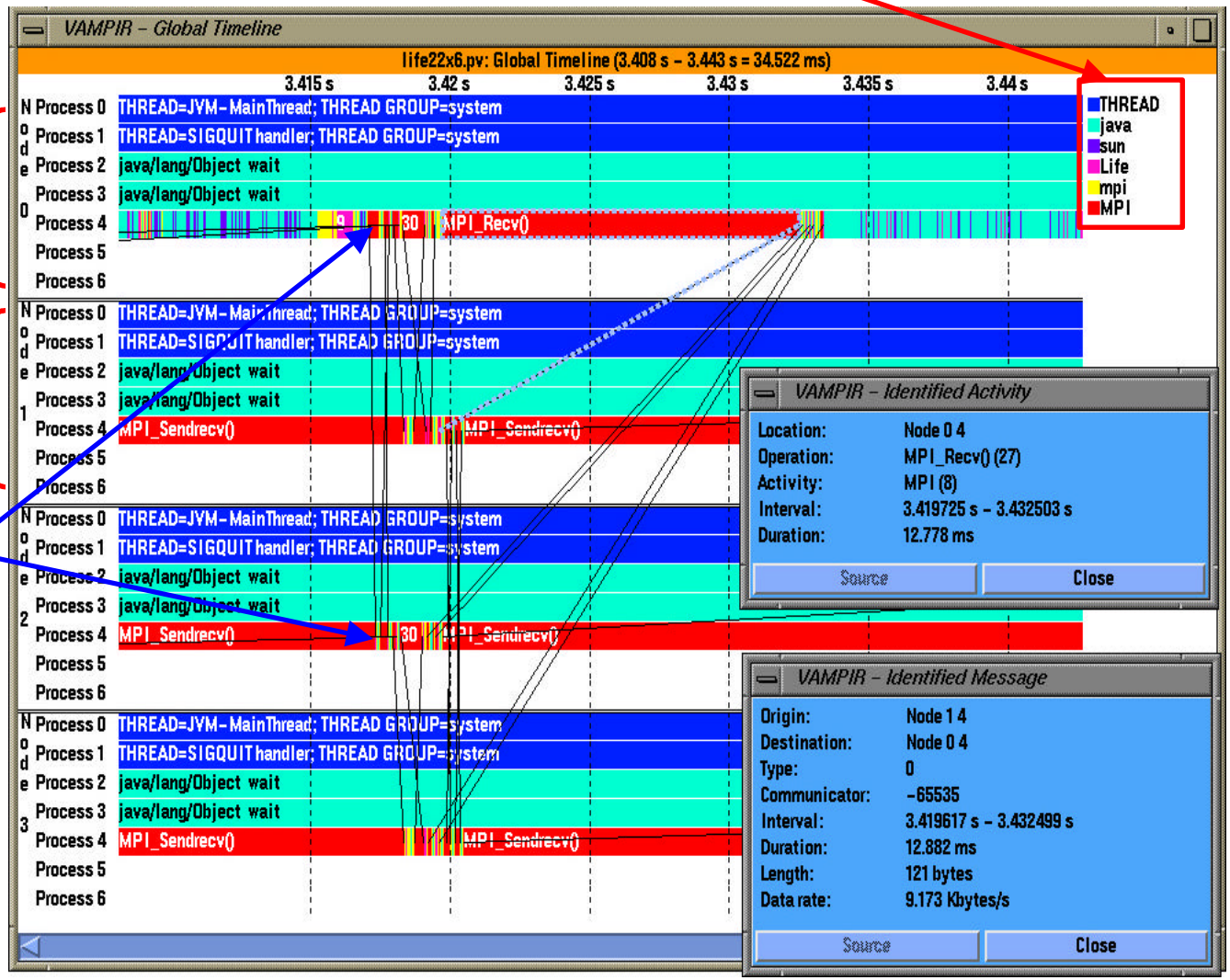
# *Parallel Java Game of Life (Profile)*

- mpiJava testcase
- 4 nodes, 28 threads



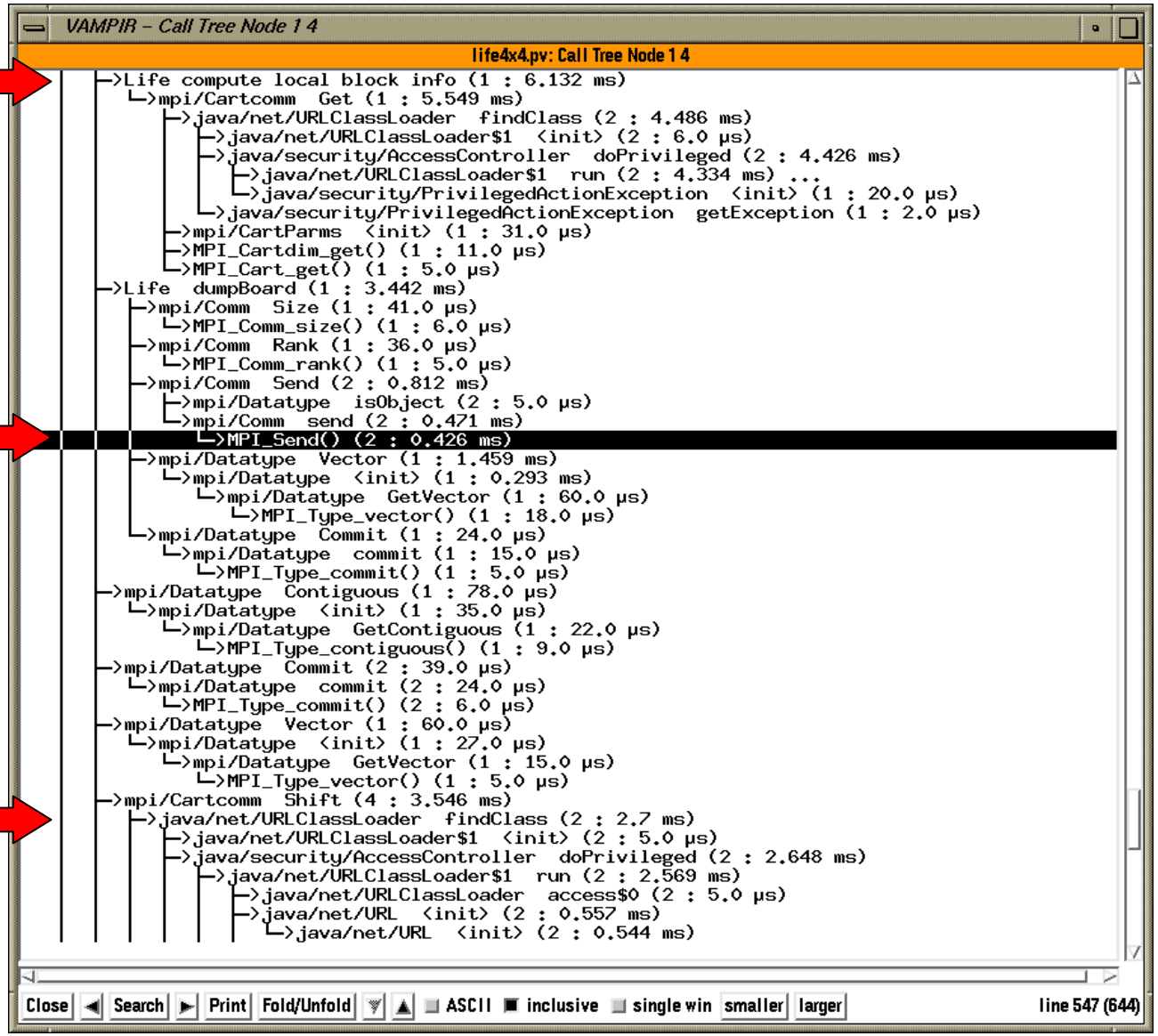
# *Parallel Java Game of Life (Trace)*

- Integrated event tracing
- Merged trace viz
- Node process grouping
- Thread message pairing
- Vampir display
- Multi-level event grouping



# *Integrated Performance View (Callgraph)*

- Source level
- MPI level
- Java packages level

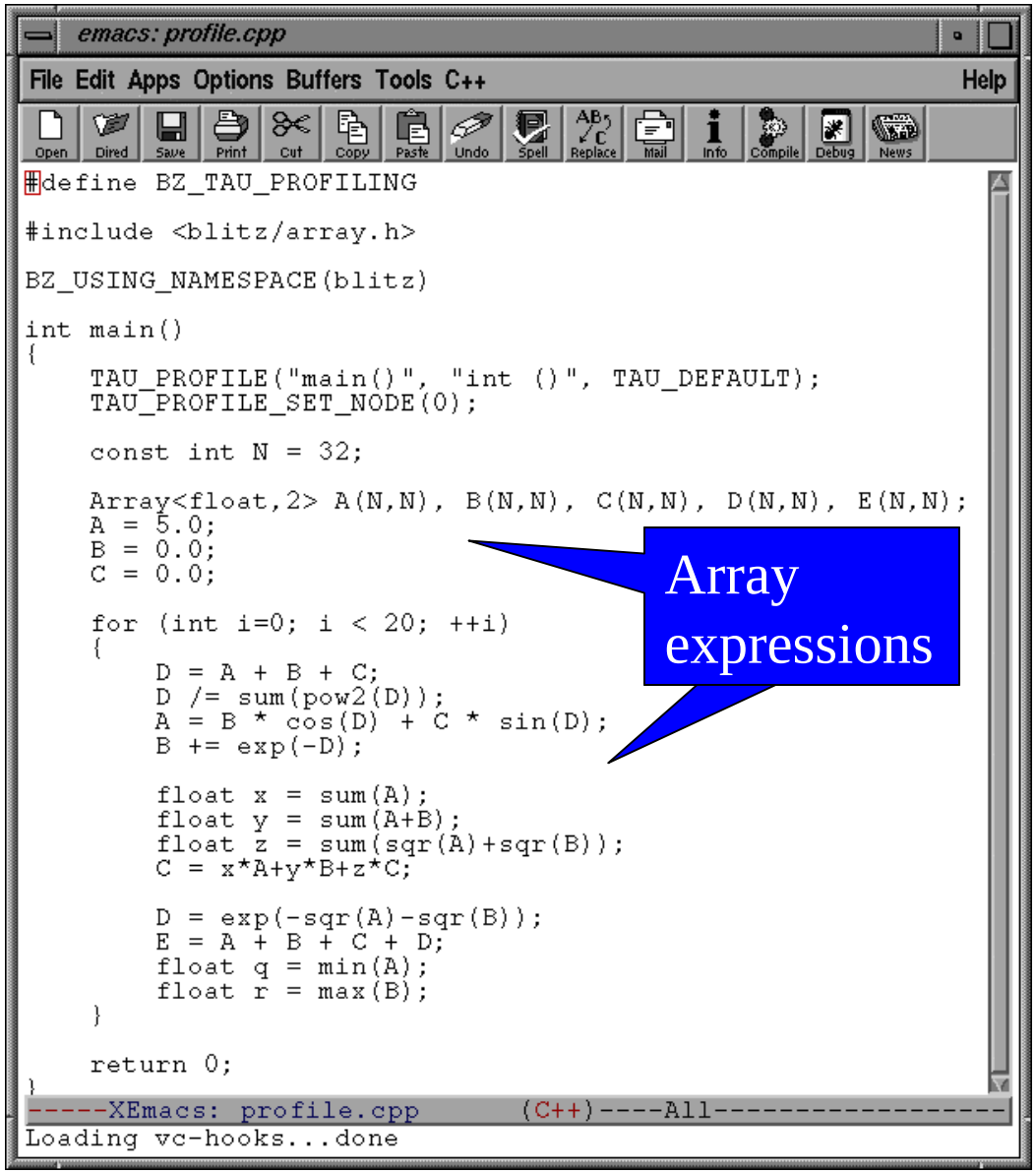


# *Object-Oriented Programming and C++*

- Object-oriented programming is based on concepts of abstract data types, encapsulation, inheritance, …
- Languages (such as C++) provide support implementing domain-specific abstractions in the form of class libraries
- Furthermore, generic programming mechanisms allow for efficient coding abstractions and compile-time transformations
- Creates a semantic gap between the transformed code and what the user expects (and describes in source code)
- Need a mechanism to expose the nature of high-level abstract computation to the performance tools
- Map low-level performance data to high-level semantics

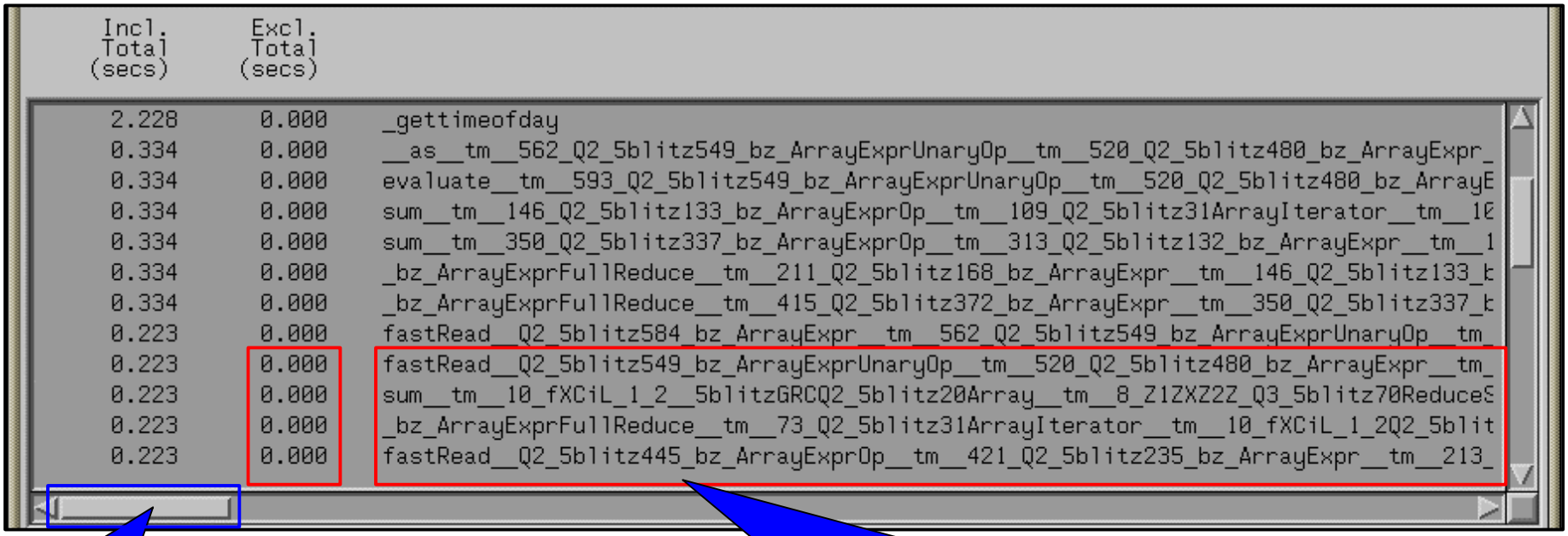
# C++ Template Instrumentation (Blitz++, PETE)

- High-level objects
  - Array classes
  - Templates (Blitz++)
- Optimizations
  - Array processing
  - Expressions (PETE)
- Relate performance data to high-level statement
- Complexity of template evaluation



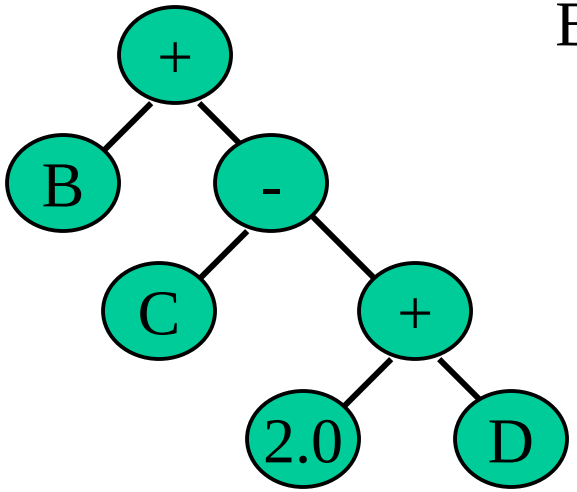
# *Standard Template Instrumentation Difficulties*

- Instantiated templates result in mangled identifiers
- Standard profiling techniques / tools are deficient
  - Integrated with proprietary compilers
  - Specific systems platforms and programming models



# *Blitz++ Library Instrumentation*

- Expression templates embed the form of the expression in a template name




BinOp<Add,
B, <BinOp<Subtract,
C, <BinOp<Multiply,
Scalar<2.0>, D>>>

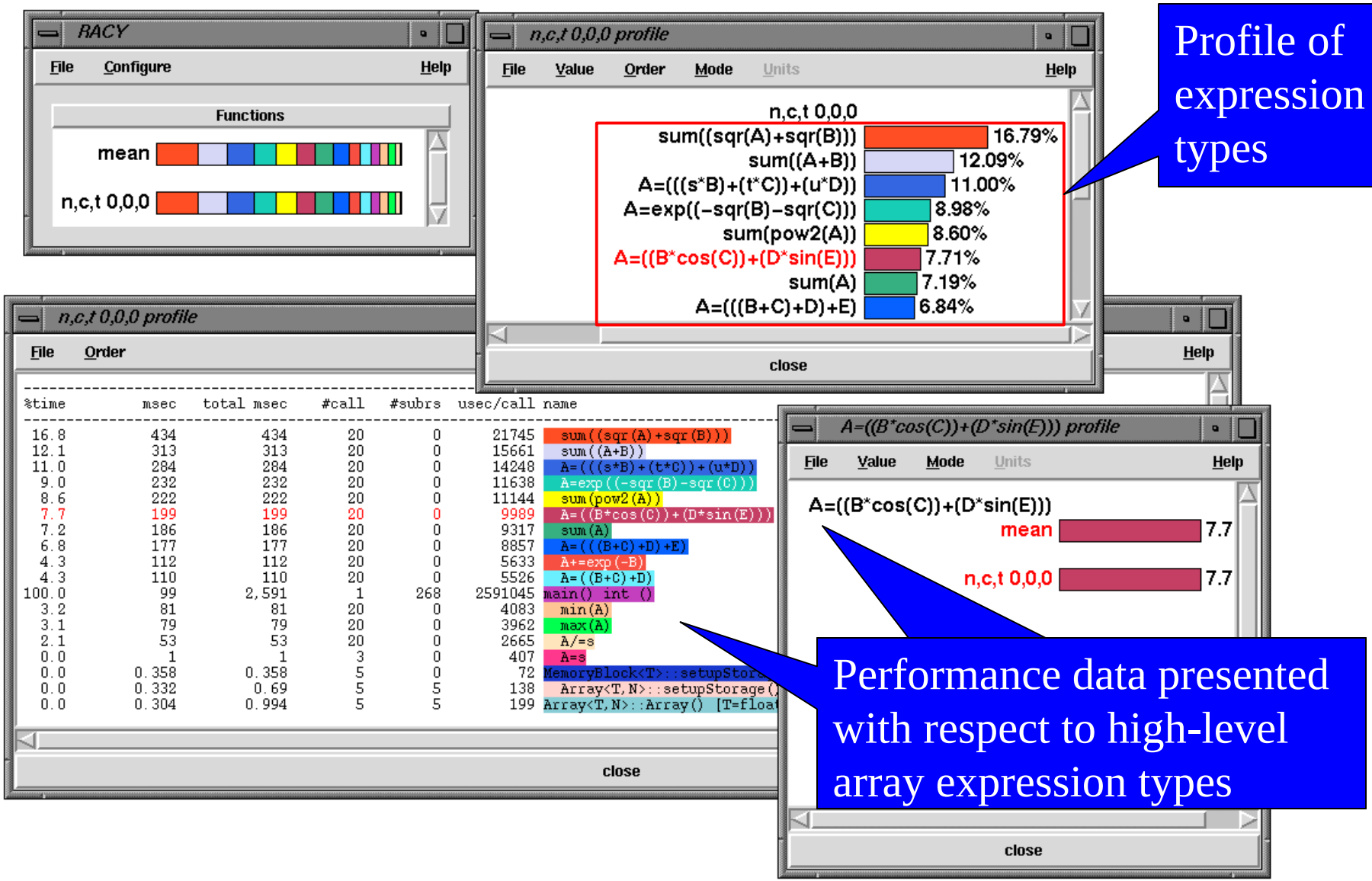
- In Blitz++, the library describes the structure of the expression template to the profiling toolkit
- Allows for pretty printing the expression templates

  Expression: B + C - 2.0 * D

# *Blitz++ Library Instrumentation (example)*

```
#ifdef BZ_TAU_PROFILING
static string exprDescription;
if (!exprDescription.length()) {
  exprDescription = "A";
  prettyPrintFormat format(_bz_true); // Terse mode on
  format.nextArrayOperandSymbol();
  T_update::prettyPrint(exprDescription);
  expr.prettyPrint(exprDescription, format);
}
TAU_PROFILE(" ", exprDescription, TAU_BLITZ);
#endif
```

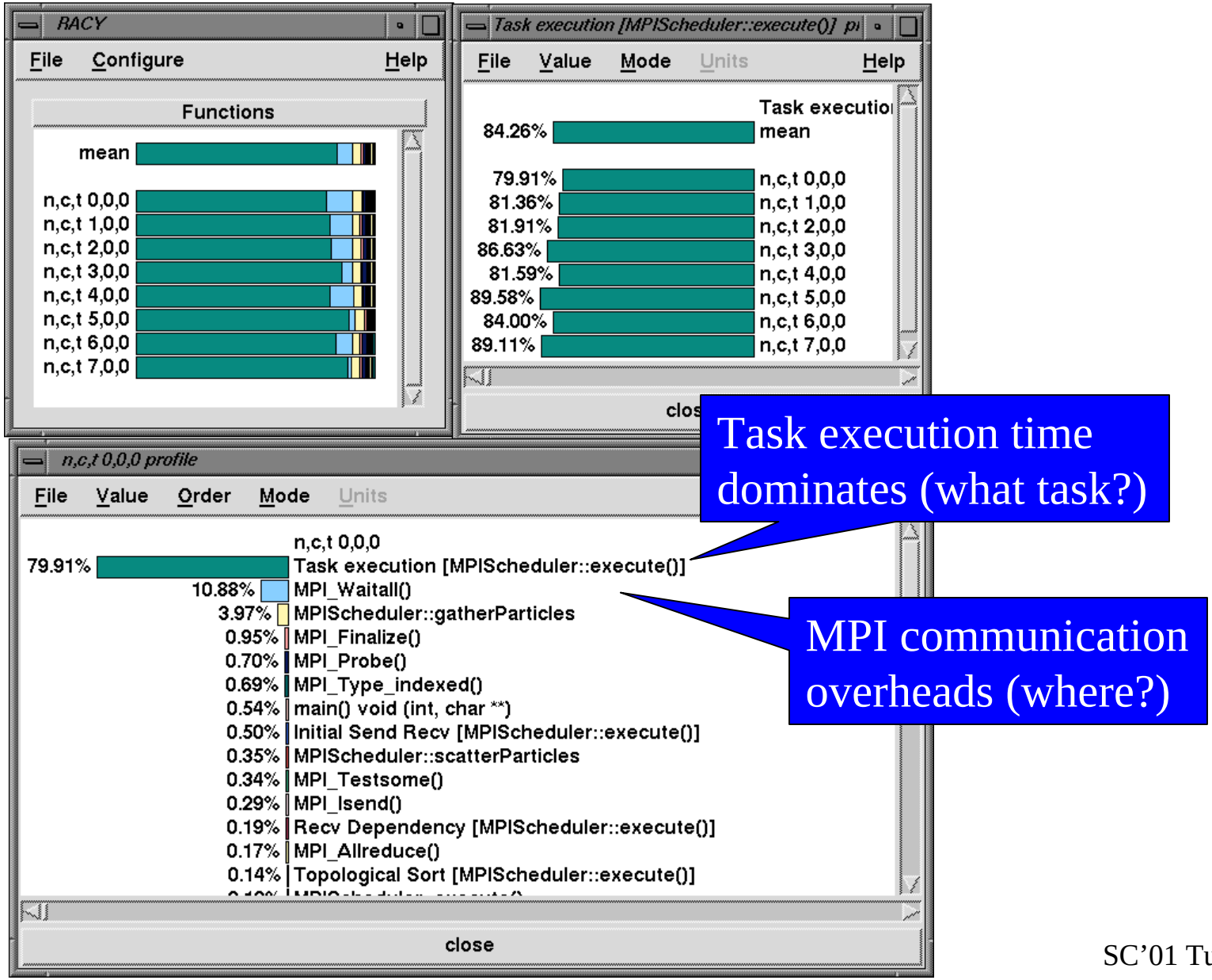
# TAU Instrumentation and Profiling for C++



# *Hierarchical Parallel Software (C-SAFE/Uintah)*

- Center for Simulation of Accidental Fires & Explosions
  - ASCI Level 1 center
  - PSE for multi-model simulation high-energy explosion
- Uintah parallel programming framework
  - Component-based and object-parallel
  - Multi-model task-graph scheduling and execution
  - Shared-memory (thread), distributed-memory (MPI), and mixed-model parallelization
  - Integrated with SCIRun framework
- TAU integration in Uintah
  - Mapping: task object 1 grid object 1 patch object

# Task Execution in Uintah Parallel Scheduler



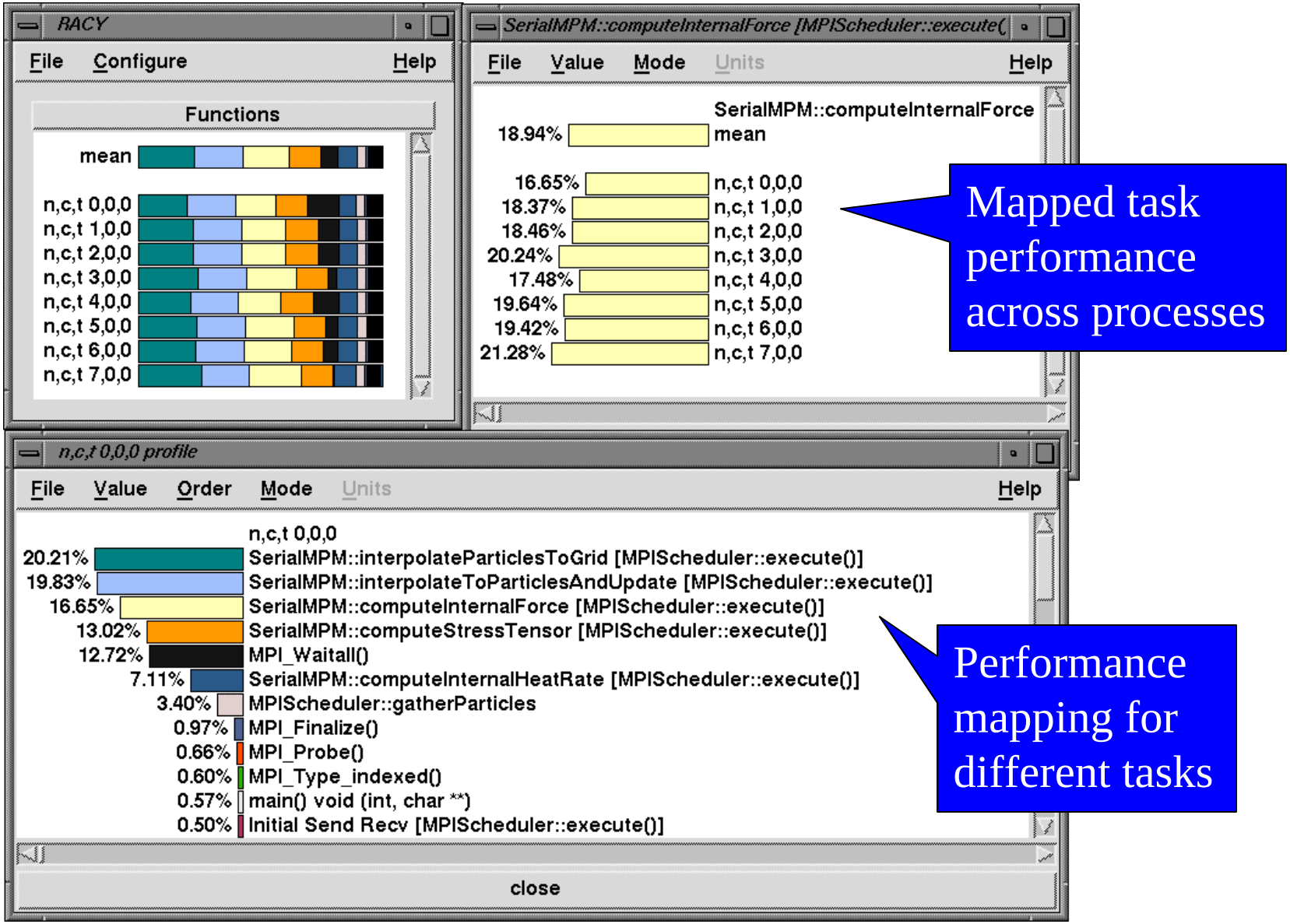
# *Task Computation and Mapping*

- Task computations on individual particles generate work packets that are scheduled and executed
  - Work packets that "interpolate particles to grid"
- Assign semantic name to a task abstraction
  - SerialMPM::interpolateParticleToGrid
- Partition execution time among different tasks
  - Need to relate the performance of each particle computation (work packet) to the associated task
  - Map TAU timer object to task (abstract) computation
- Further partition the performance data along different domain-specific axes (task, patches, …)
- Helps bridge the semantic-gap!

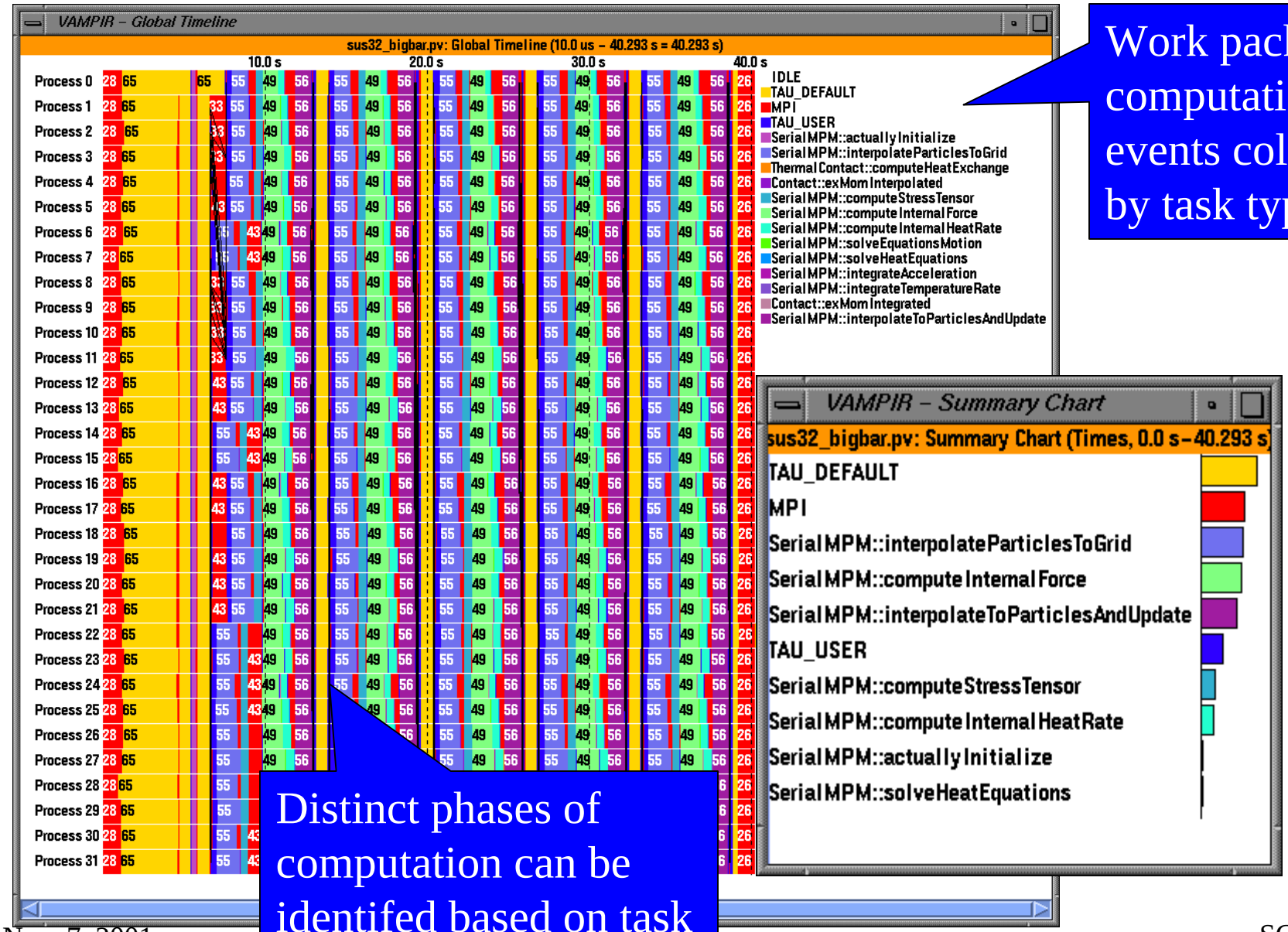
# *Mapping Instrumentation (example)*

```
void MPIScheduler::execute(const ProcessorGroup * pc,
                           DataWarehouseP   & old_dw,
                           DataWarehouseP   & dw ) {
  ...
  TAU_MAPPING_CREATE(
    task->getName(), "[MPIScheduler::execute()]",
    (TauGroup_t)(void*)task->getName(), task->getName(), 0);
  ...
  TAU_MAPPING_OBJECT(tautimer)
  TAU_MAPPING_LINK(tautimer,TauGroup_t)(void*)task->getName());
  // EXTERNAL ASSOCIATION
  ...
  TAU_MAPPING_PROFILE_TIMER(doitprofiler, tautimer, 0)
  TAU_MAPPING_PROFILE_START(doitprofiler,0);
  task->doit(pc);
  TAU_MAPPING_PROFILE_STOP(0);
  ...
}
```

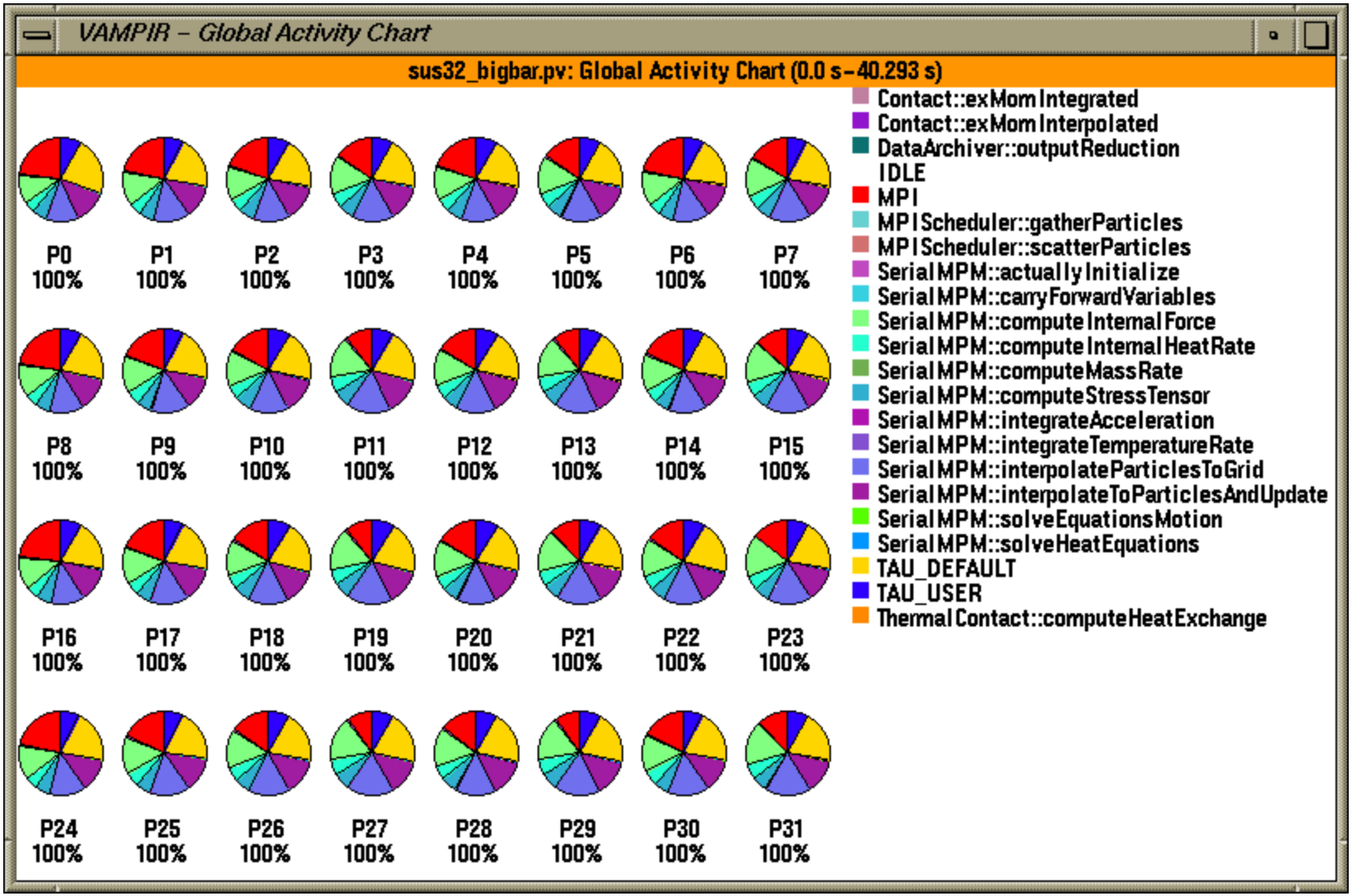
# *Work Packet – to – Task Mapping (Profile)*



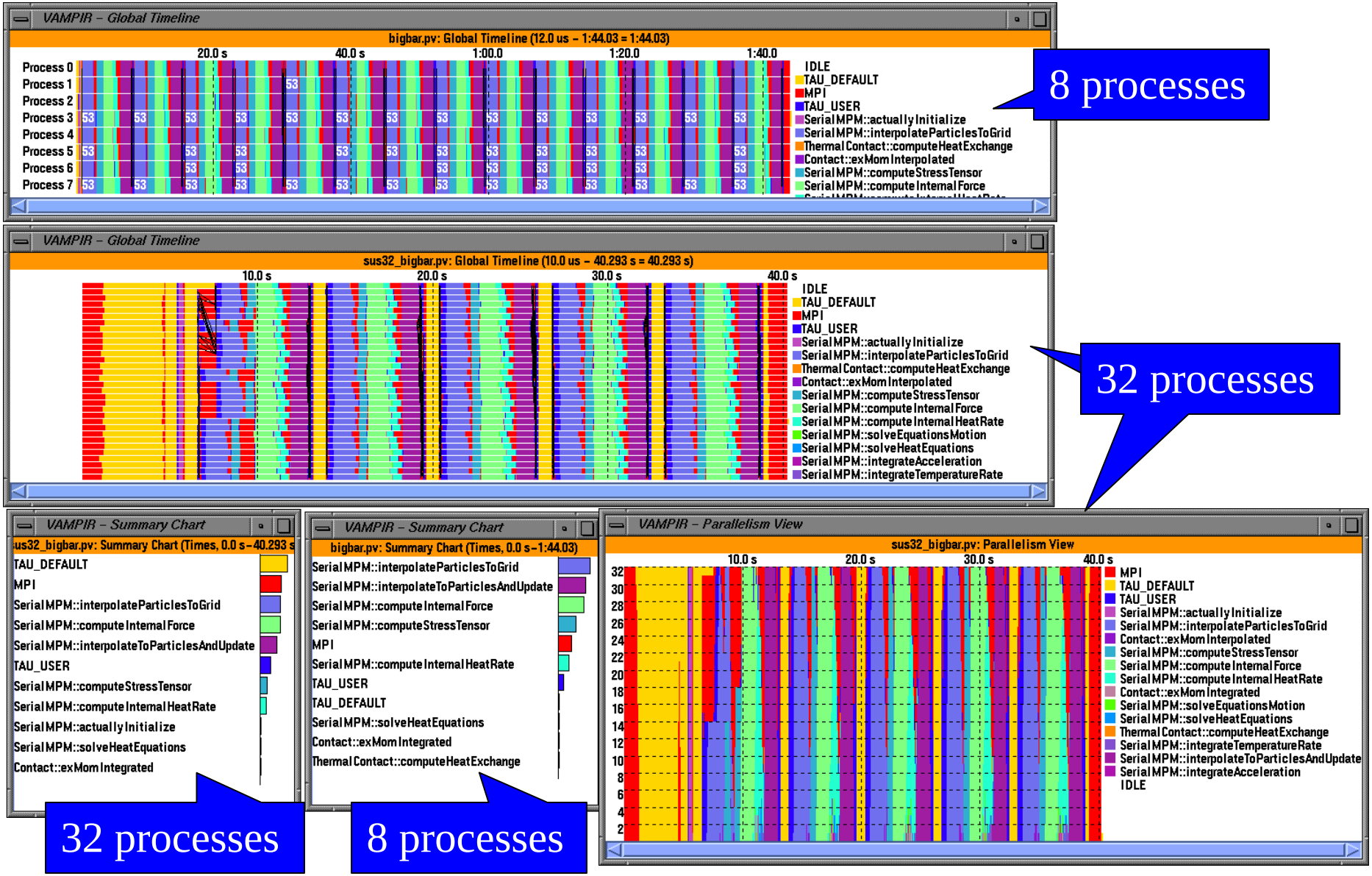
# *Work Packet – to – Task Mapping (Trace)*



# *Statistics for Relative Task Contributions*



# Comparing Uintah Traces for Scalability Analysis



# *Evolution of the TAU Performance System*

- Future parallel computing environments need to be more adaptive to achieve and sustain high performance levels
- TAU's existing strength lies in its robust support for performance instrumentation and measurement
- TAU will evolve to support new performance capabilities
  - Online performance data access via application-level API
  - Whole-system, integrative performance monitoring
  - Dynamic performance measurement control
  - Generalize performance mapping
  - Runtime performance analysis and visualization
- Three-year DOE MICS research and development grant

# *Performance Technology for Complex Parallel Systems*

## *Part 3 – Alternative Tools and Frameworks*

*Bernd Mohr*

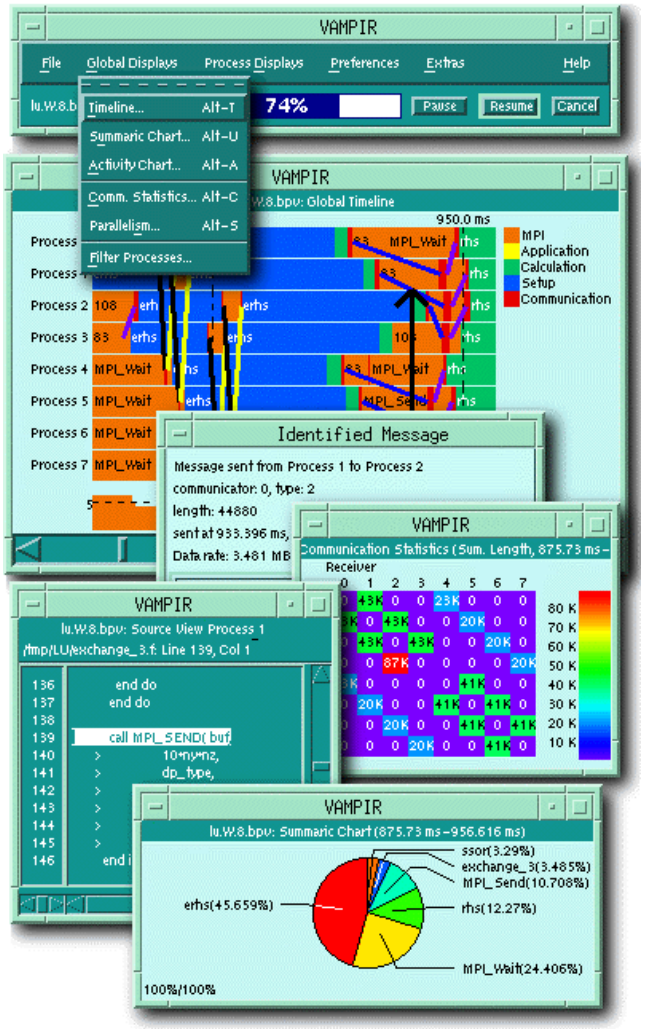
John von Neumann - Institut für Computing
Zentralinstitut für Angewandte Mathematik

# *Goals*

- Learn about commercial performance analysis products for complex parallel systems
  - Vampir event trace visualization and analysis tool
  - Vampirtrace event trace recording library
  - GuideView OpenMP performance analysis tool
  - VGV (integrated Vampir / GuideView environment)
- Learn about future advanced components for automatic performance analysis and guidance
  - EXPERT automatic event trace analyzer
- Discuss plans for performance tool integration

# *Vampir*

- **V**isualization and **A**nalysis of **MPI** P**R**ograms
- Originally developed by Forschungszentrum Jülich
- Current development by Technical University Dresden
- Distributed by PALLAS, Germany

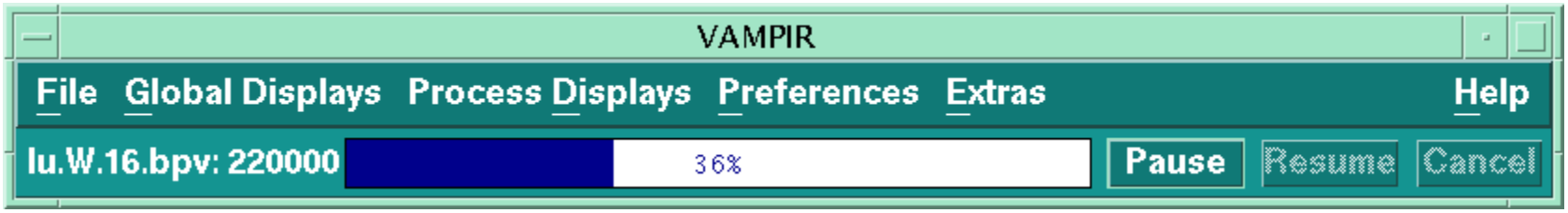



- **http://www.pallas.de/pages/vampir.htm**

# *Vampir: General Description*

- Offline trace analysis for message passing trace files
- Convenient user–interface / easy customization
- Scalability in time and processor–space
- Excellent zooming and filtering
- Display and analysis of MPI and application events:
  - User subroutines
  - Point–to–point communication
  - Collective communication
  - MPI–2 I/O operations
- Large variety of customizable (via context menus) displays for ANY part of the trace

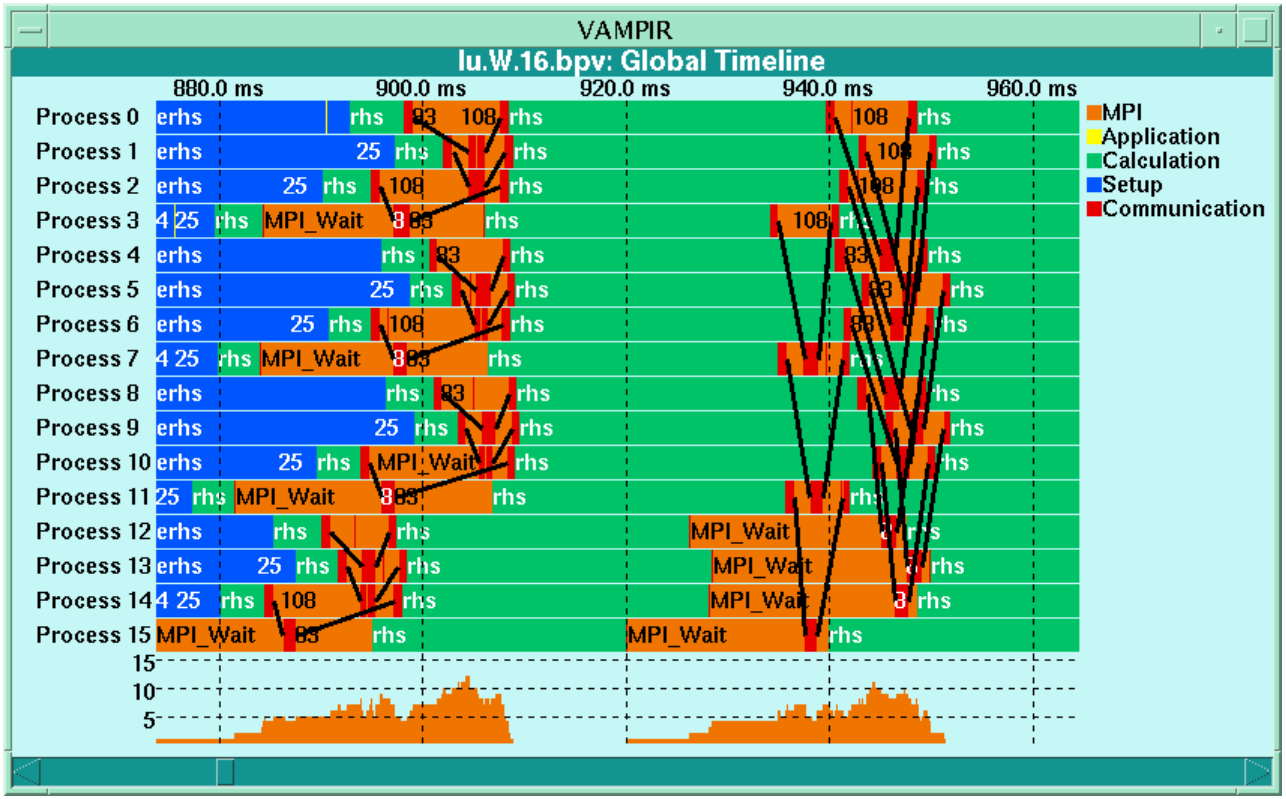
# *Vampir: Main Window*




- Trace file loading can be
  - Interrupted at any time
  - Resumed
  - Started at a specified time offset
- Provides main menu
  - Access to global and process local displays
  - Preferences
  - Help
- Trace file can be re–written (re–grouped symbols)

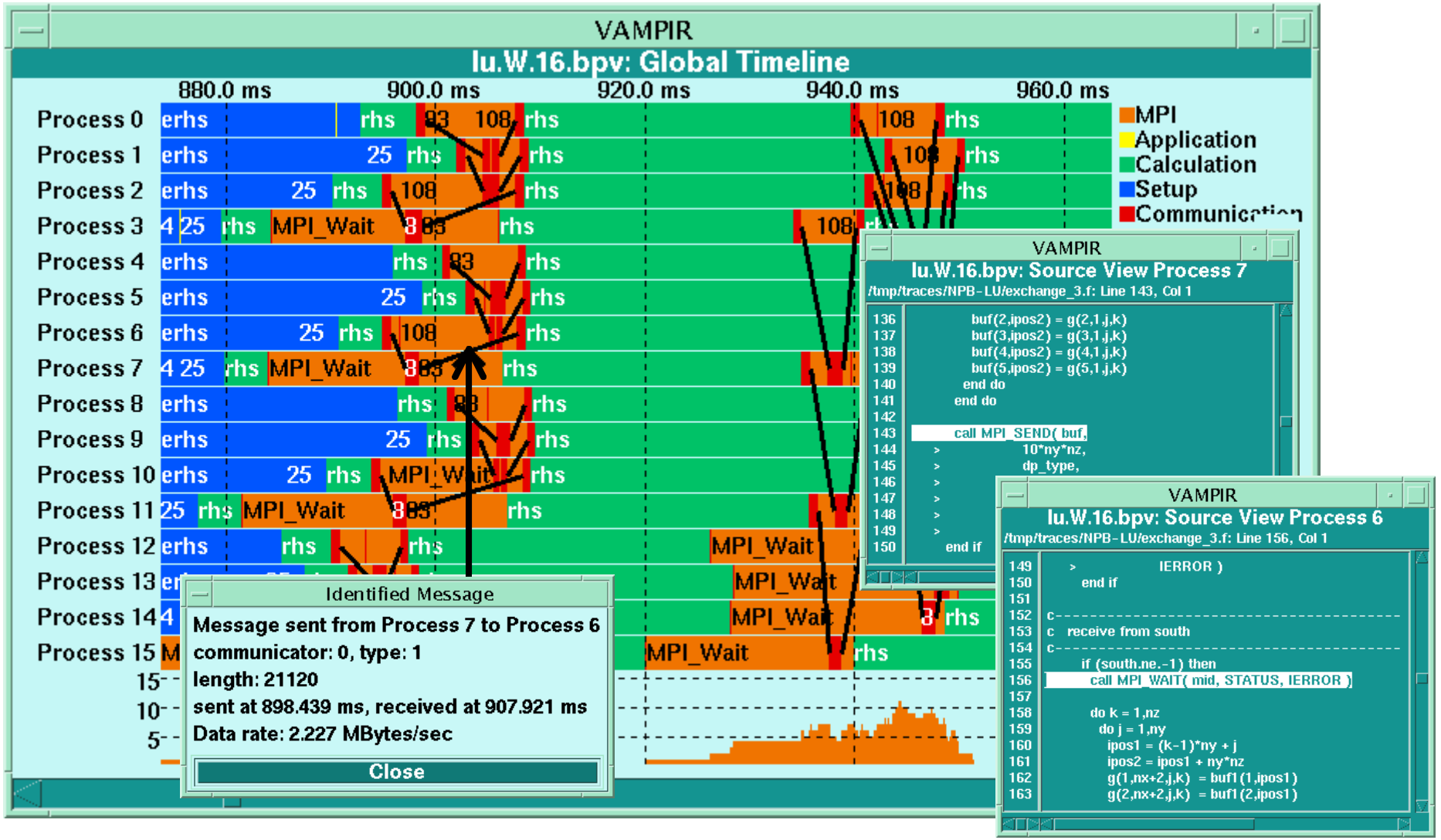
# *Vampir: Timeline Diagram*

- Functions organized into groups
- Coloring by group
- Message lines can be colored by tag or size




- Information about states, messages, collective, and I/O operations available by clicking on the representation

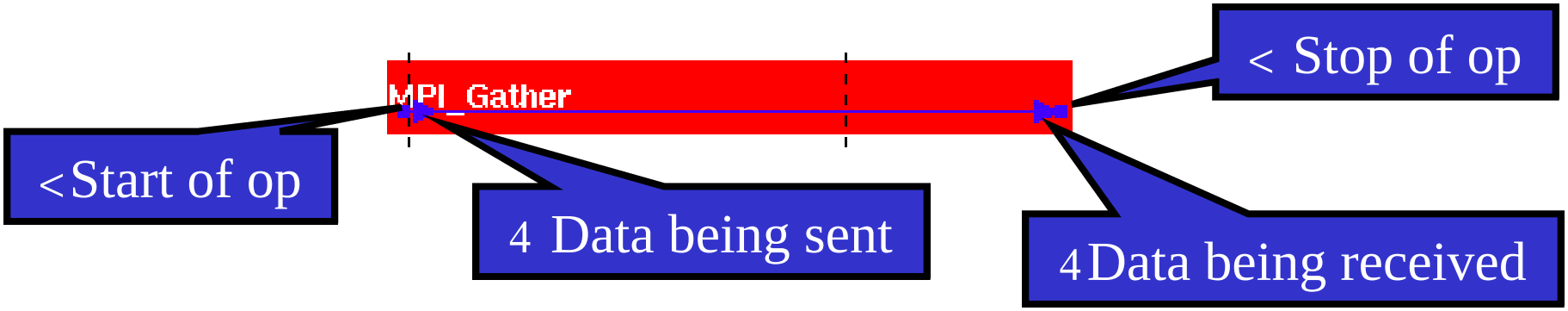
# *Vampir: Timeline Diagram (Message Info)*



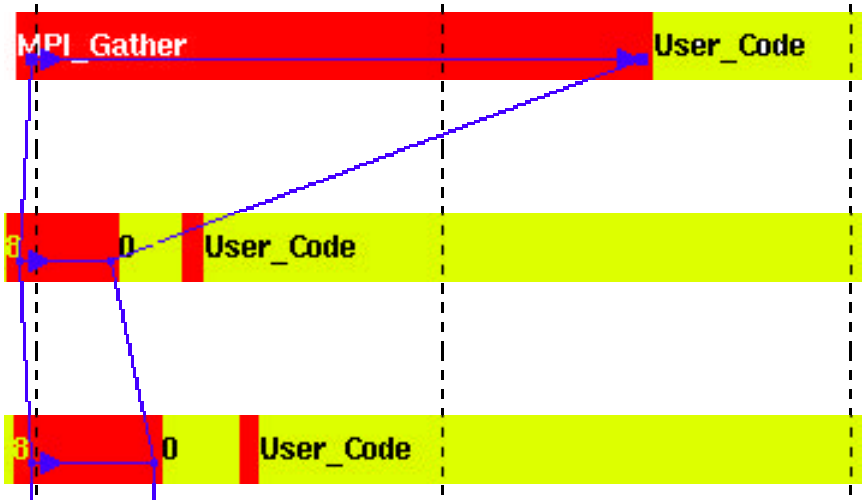

- Source–code references are displayed if recorded in trace

# *Vampir: Support for Collective Communication*

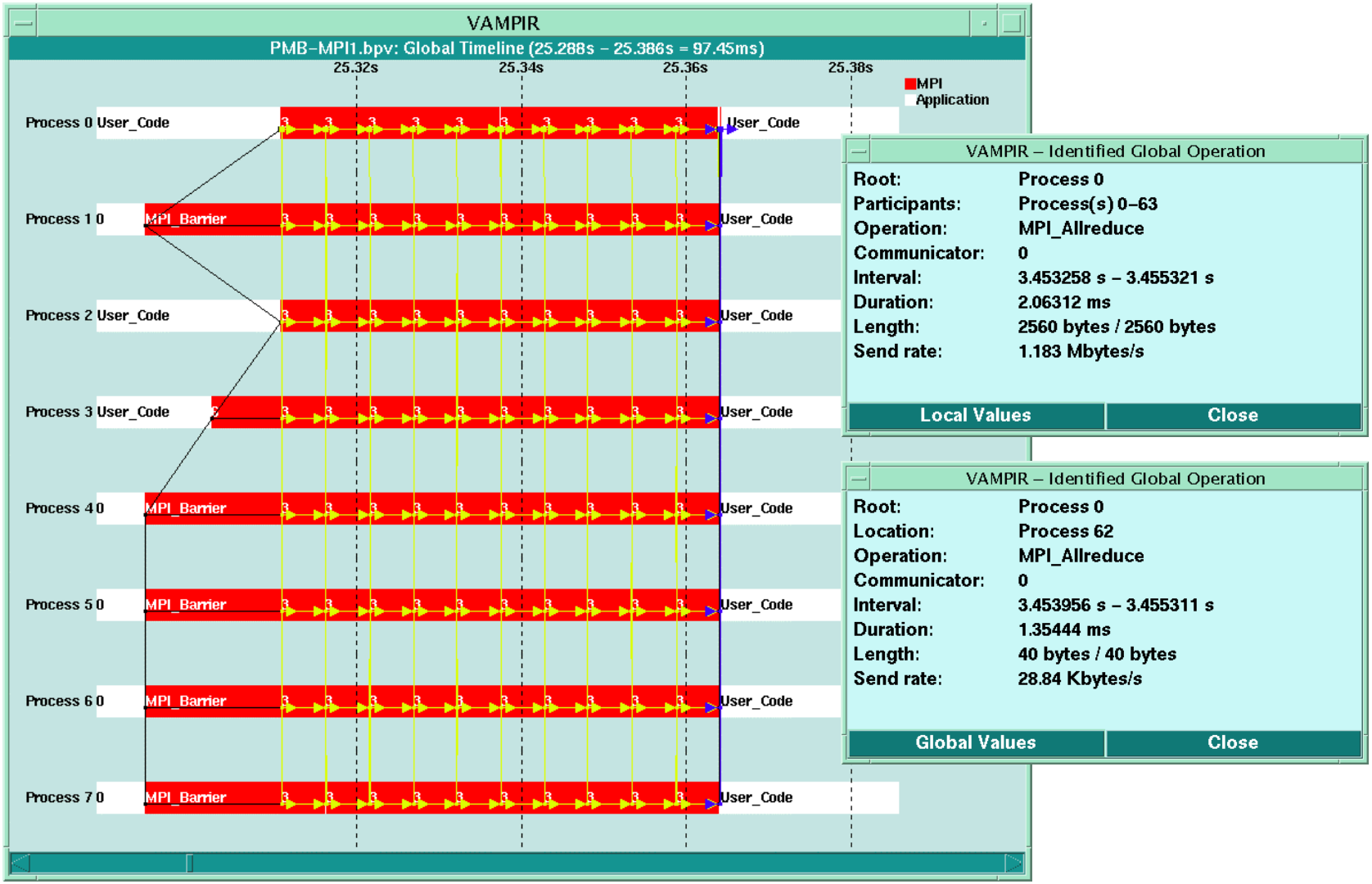
- For each process: locally mark operation



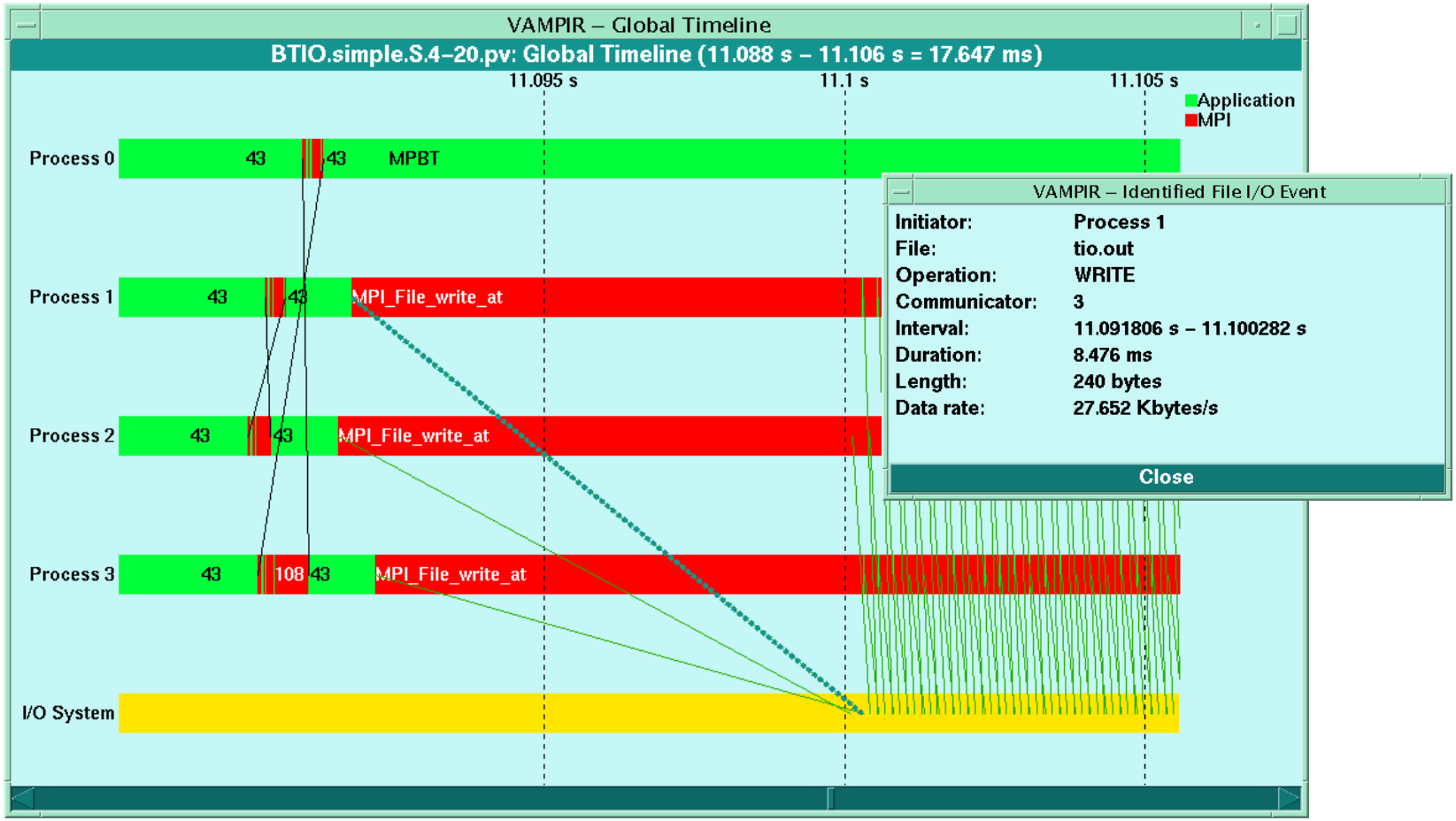

- Connect start/stop points by lines



# *Vampir: Collective Communication Display*

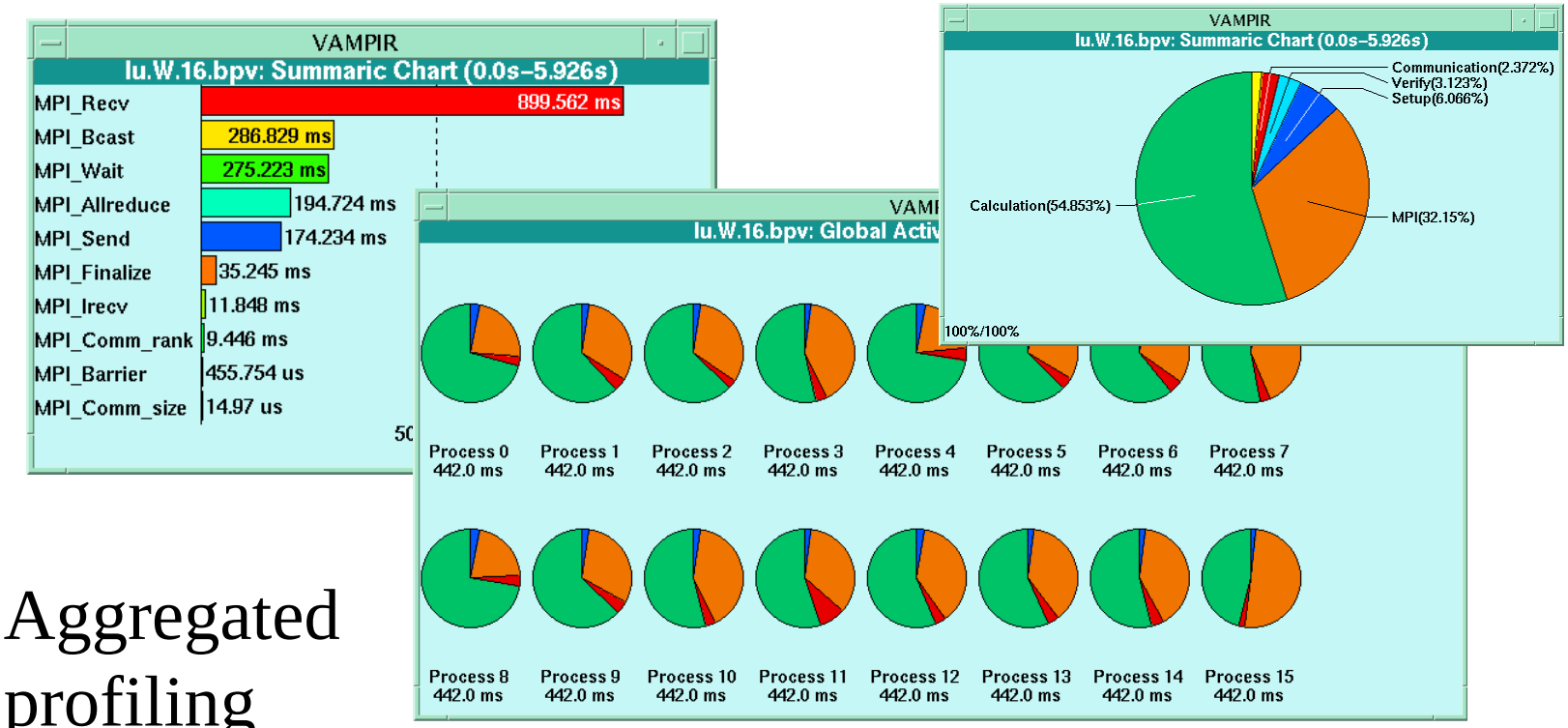


# *Vampir: MPI-I/O Support*




- MPI I/O operations shown as message lines to separate I/O system time line

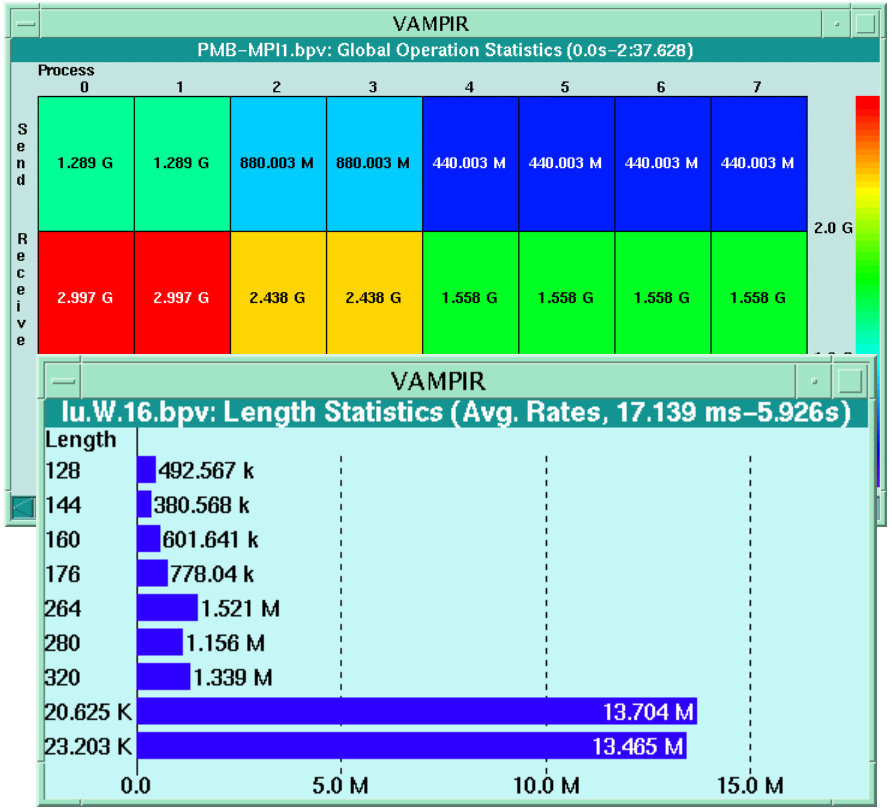
# *Vampir: Execution Statistics Displays*




- Aggregated profiling information: execution time, # calls, inclusive/exclusive
- Available for all/any group (activity)
- Available for all routines (symbols)
- Available for any trace part (select in timeline diagram)

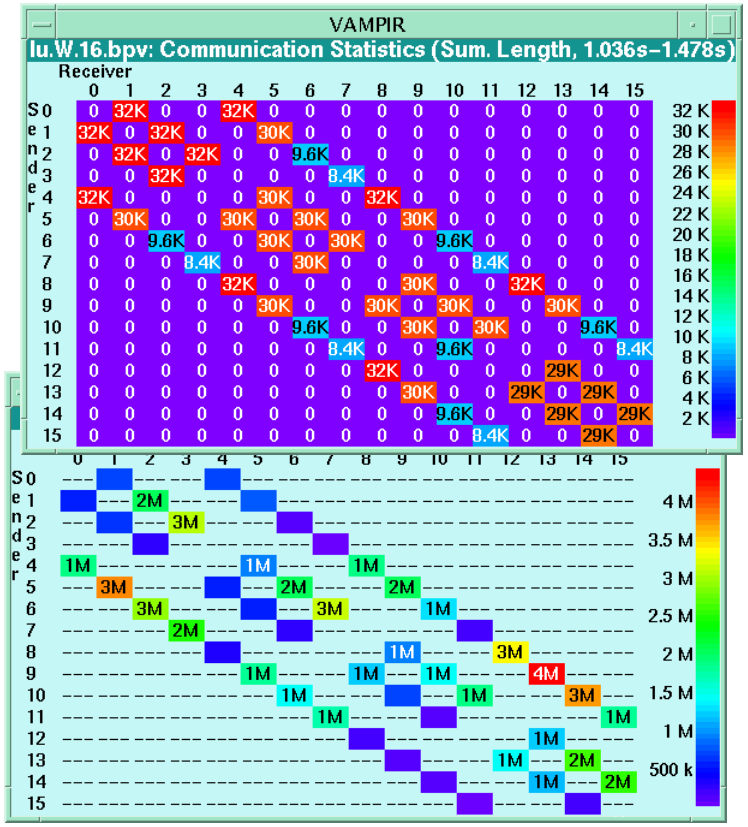
# *Vampir: Communication Statistics Displays*

- Bytes sent/received for collective operations

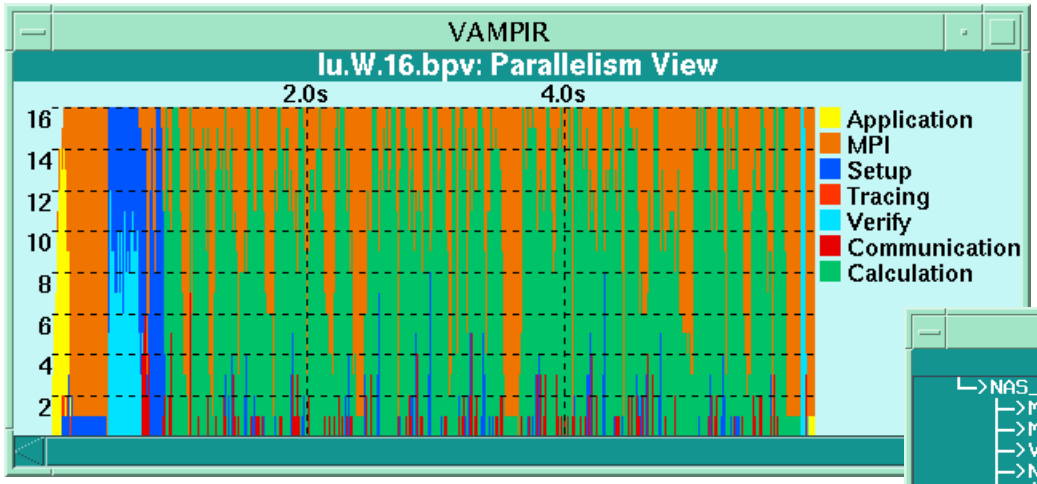



- Message length statistics
- Available for any trace part

- Byte and message count, min/max/avg message length and min/max/avg bandwidth for each process pair



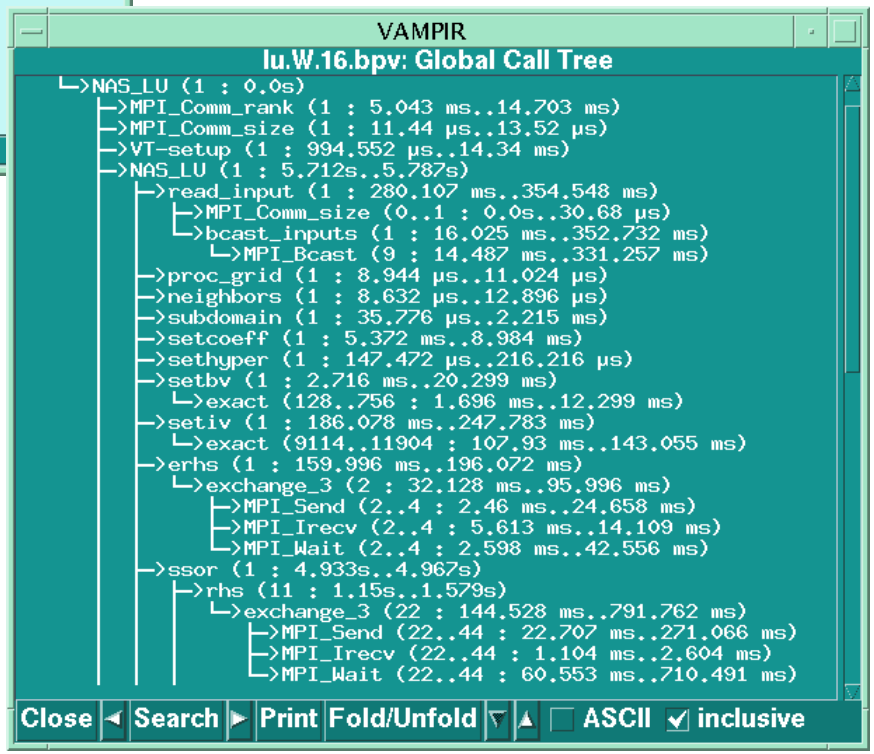
# *Vampir: Other Features*



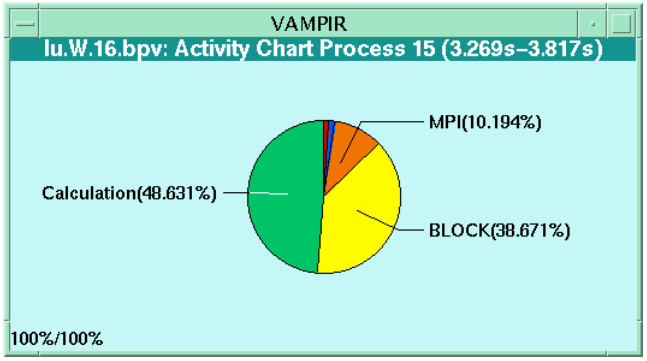

- Parallelism display

- Powerful filtering and trace comparison features
- All diagrams highly customizable (through context menus)

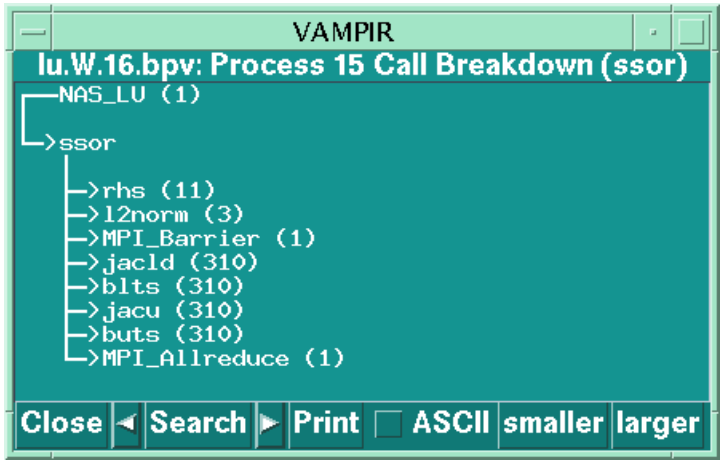
- Dynamic global call graph tree



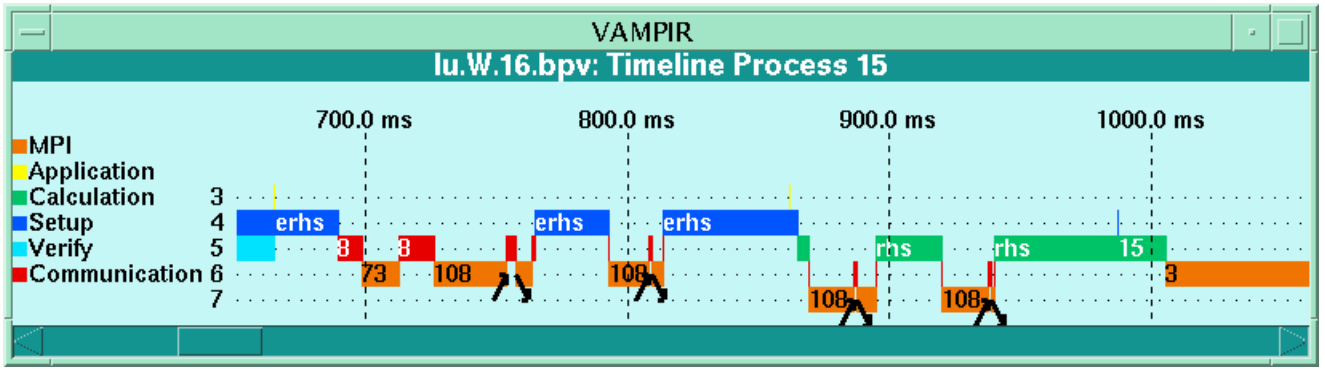
# *Vampir: Process Displays*




- Activity chart




- Call tree




- Timeline

- For all selected processes in the global displays

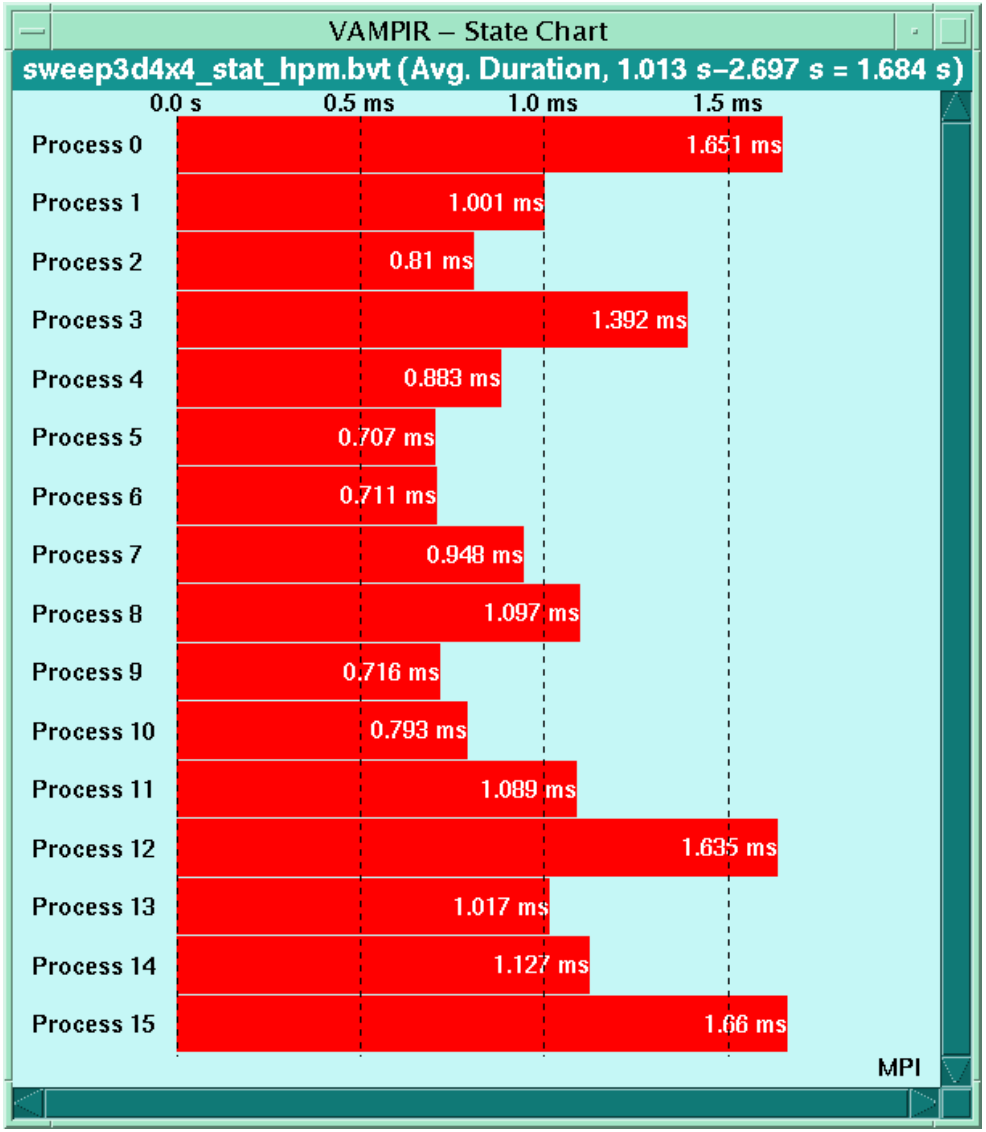
# *Vampir: New Features*

- New Vampir versions (3 and 4)
  - New core (dramatic timeline speedup, significantly reduced memory footprint)
  - Load–balance analysis display
  - Hardware counter value displays
  - Thread analysis
  - show hardware and grouping structure
  - Improved statistics displays

  - Raised scalability limits: can now analyse 100s of processes/threads

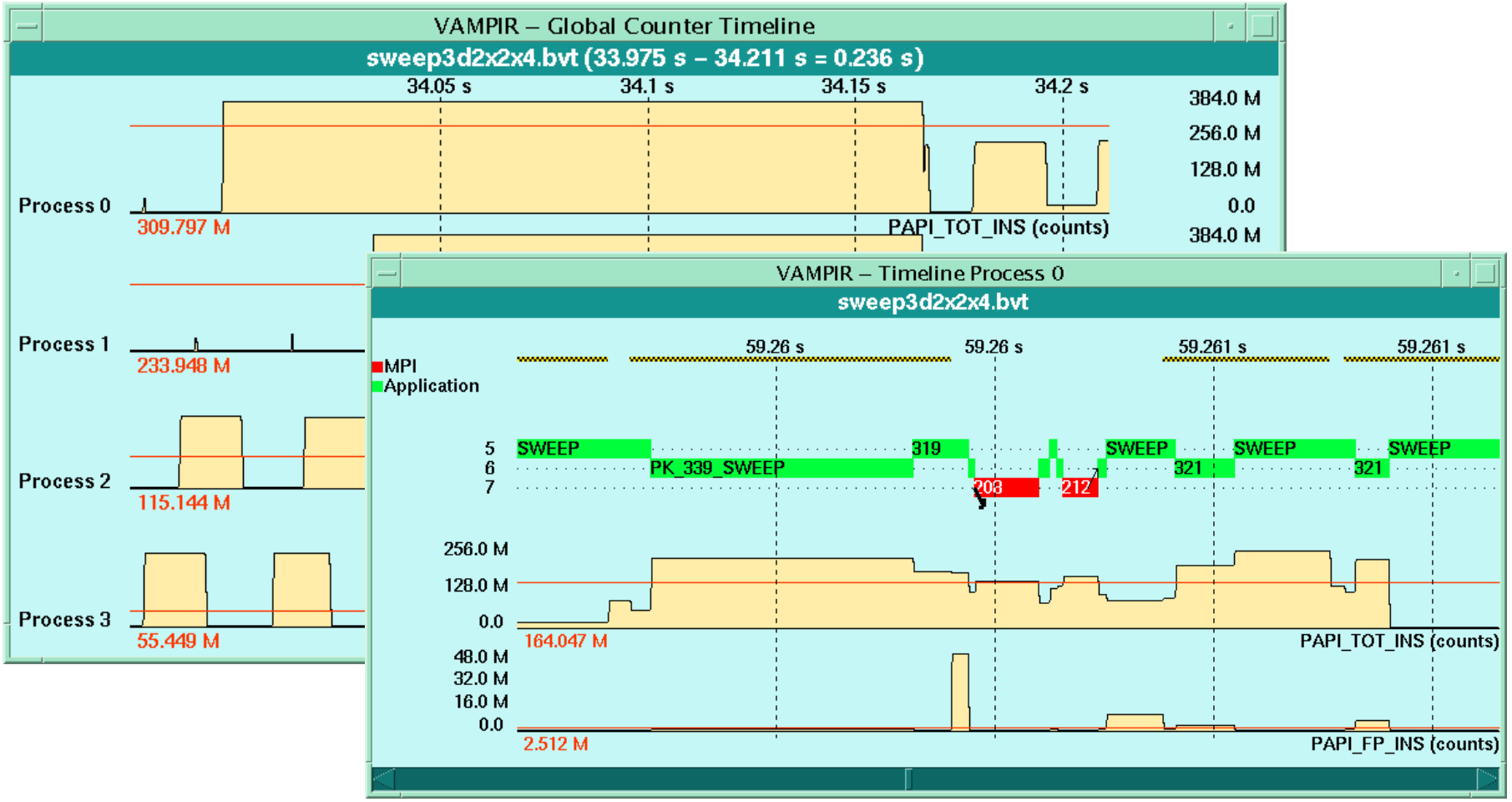
# *Vampir: Load Balance Analysis*

- *State Chart* display
- Aggregated profiling information:
  execution time, # calls,
  inclusive/exclusive
- For all/any group (activity)
- For all routines (symbols)
- For any trace part



# *Vampir: HPM Counter*

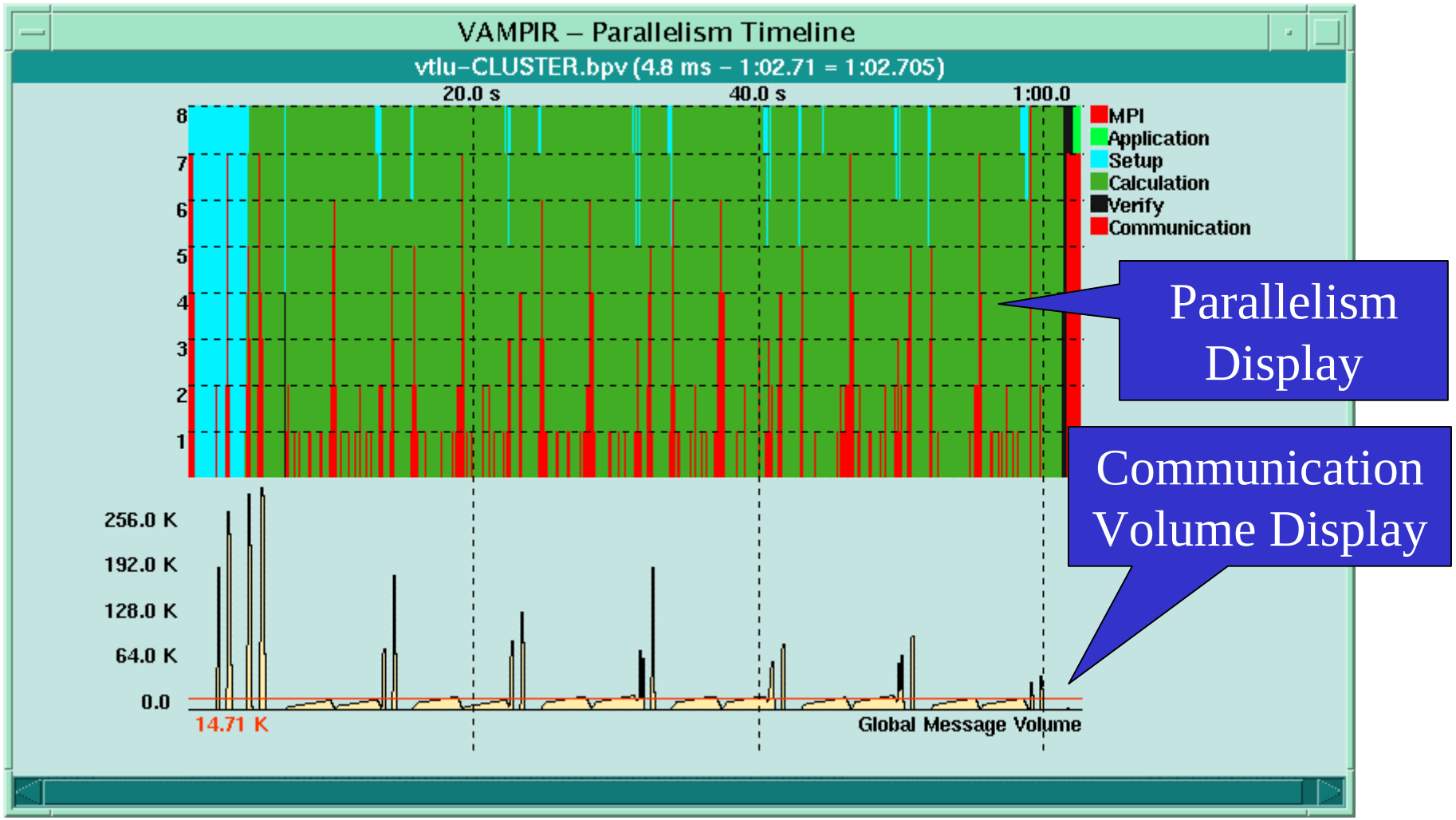
- *Counter Timeline* Display




- *Process Timeline* Display

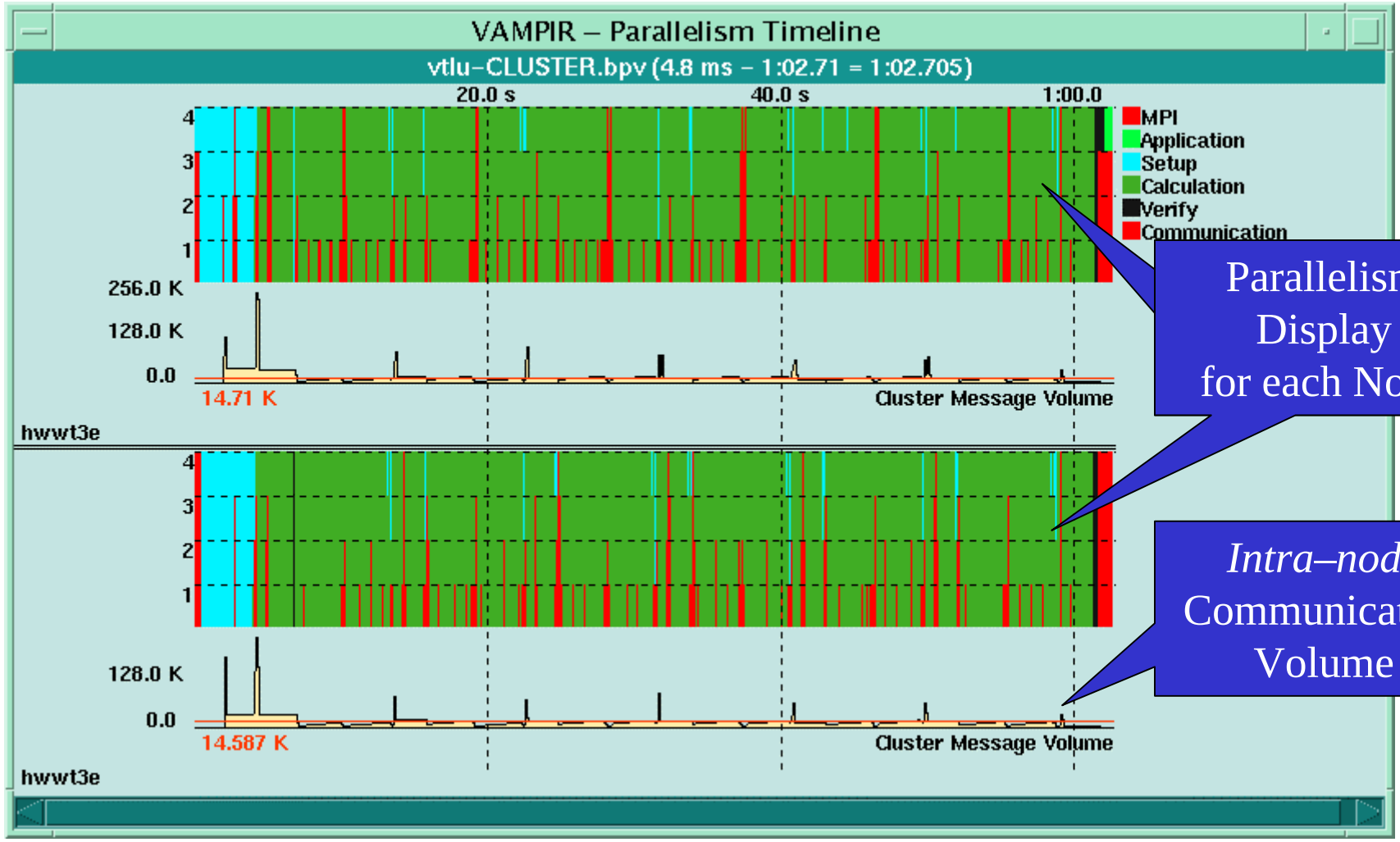
# *Vampir: Cluster Timeline*

- Display of whole system



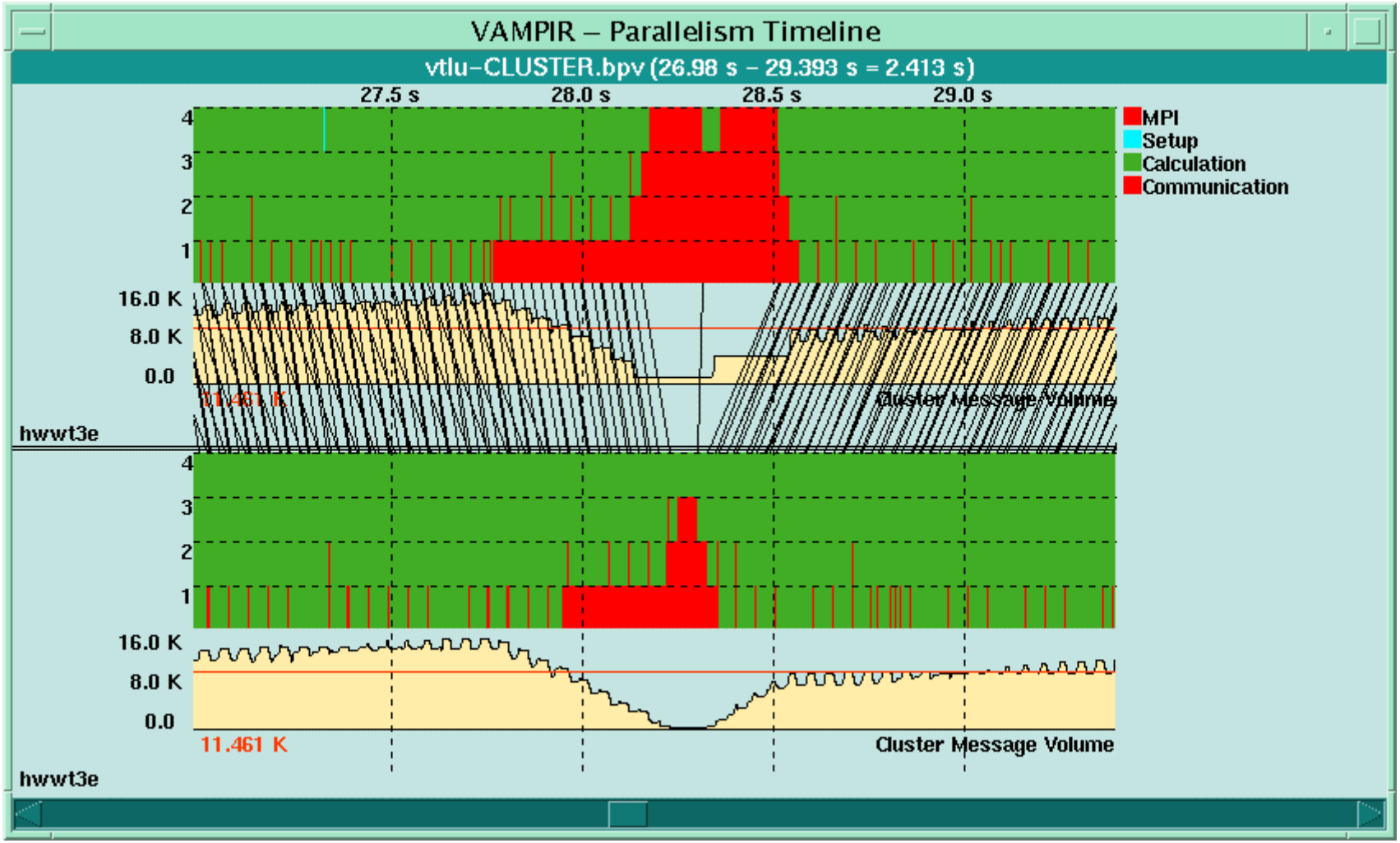
# *Vampir: Cluster Timeline*

- SMP or Grid Nodes Display



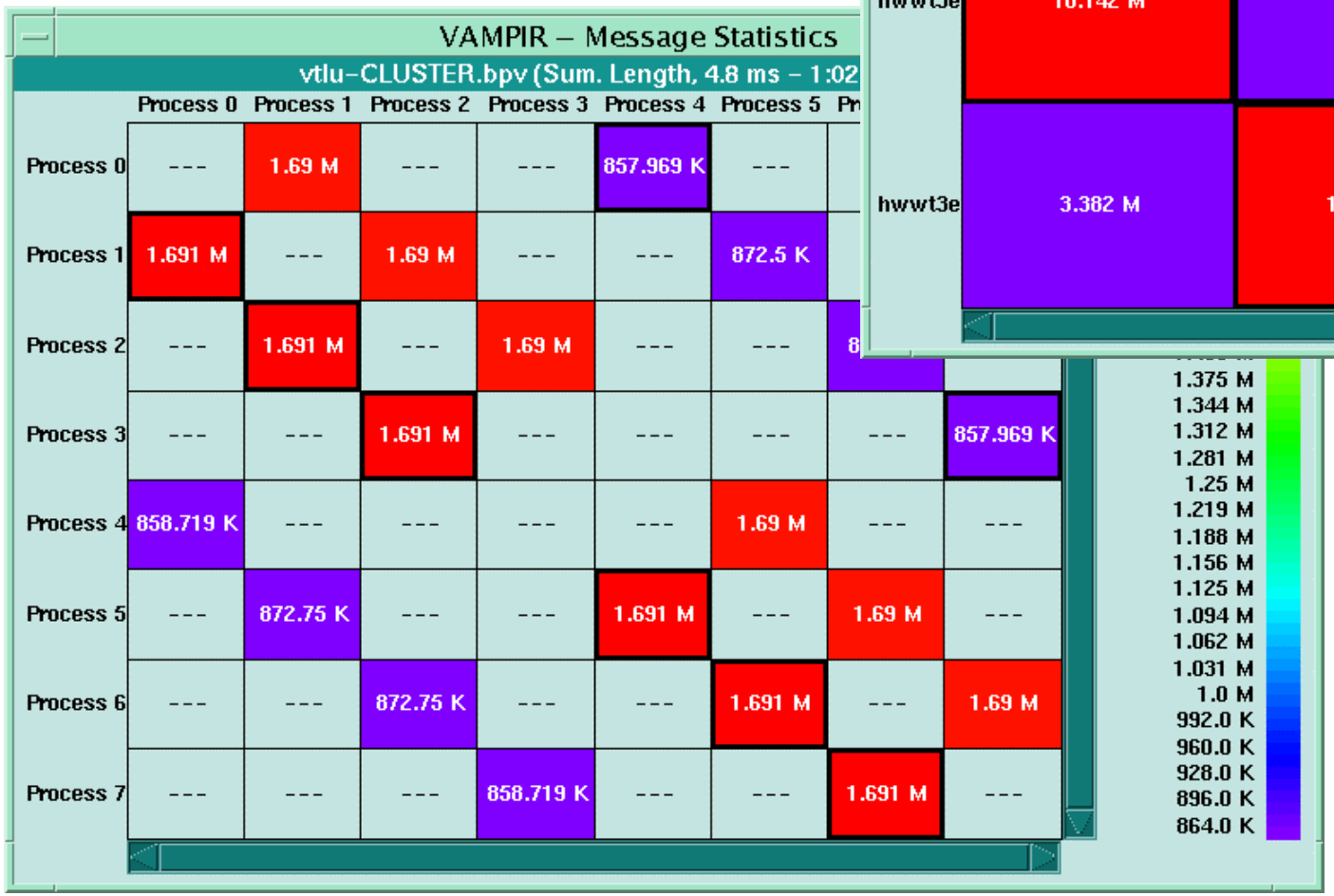
# *Vampir: Cluster Timeline(2)*

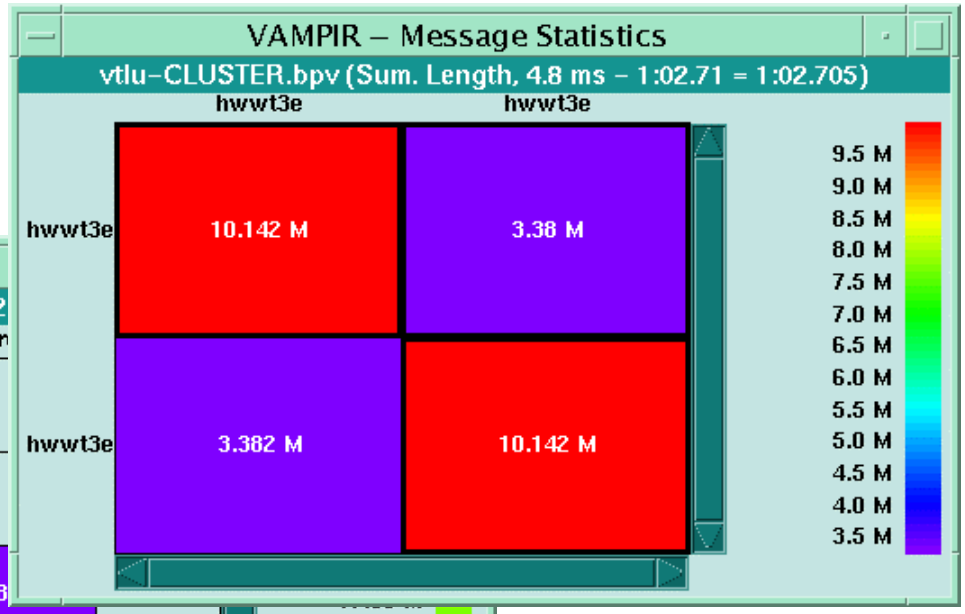
- Display of messages between nodes enabled



# *Vampir: Improved Message Statistics Display*

- Process View




- NodeView

# *Release Schedule*

- Vampir/SX and Vampirtrace/SX
  - Version 1 available via NEC Japan
  - Version 2 is ready for release
- Vampir/SC and Vampirtrace/SC
  - Version 3 is available from Pallas
  - Version 4 scheduled for Q4/2001
- Vampir and Vampirtrace
  - Version 3 is scheduled for Q4/2001
  - Version 4 will follow in 2002

# *Vampir Feature Matrix*

| | Vampir | | Vampir/SC | | Vampir/SX | |
|---|---|---|---|---|---|---|
| | 3 | 4 | 3 | 4 | 1 | 2 |
| New core | yes | yes | yes | yes | yes | yes |
| Load–balance displays | no | yes | no | yes | no | yes |
| Counter analysis | no | yes | yes | yes | yes | yes |
| Thread analysis | no | yes | no | yes | no | no |
| Grouping support | no | partial | partial | yes | partial | partial |
| Improved statistics displays | yes | yes | yes | yes | yes | yes |
| Scalability (processes) | 200 | 500 | 500 | 1000 | 500 | 1000 |

# *Vampirtrace*


- Commercial product of Pallas, Germany
- Library for Tracing of MPI and Application Events
  - Records MPI point-to-point communication
  - Records MPI collective communication
  - Records MPI–2 I/O operations
  - Records user subroutines (on request)
  - Records source–code information (some platforms)
  - Support for shmem (Cray T3E)
- Uses the PMPI profiling interface
- **`http://www.pallas.de/pages/vampirt.htm`**

# *Vampirtrace: Usage*

- Record MPI–related information
  - Re–link a compiled MPI application (no re-compilation)
    - **{f90,cc,CC} *.o -o myprog**
      **-L$(VTHOME)/lib -lVT -lpmpi -lmpi**
  - Re-link with **-vt** option to MPICH compiler scripts
    - **{mpif90,mpicc,mpiCC} -vt *.o -o myprog**
  - Execute MPI binary as usual
- Record user subroutines
  - Insert calls to Vampirtrace API (portable, but inconvenient)
  - Use automatic instrumentation (NEC SX, Fujitsu VPP, Hitachi SR)
  - Use instrumentation tool (Cray PAT, dyninst, ...)

# *Vampirtrace Instrumentation API (C / C++)*

- Calls for recording user subroutines

```
#include "VT.h"
/* Symbols defined with/without source information */
VT_symdefl(123, "foo", "USER", "foo.c:6");
VT_symdef (123, "foo", "USER");

void foo {
    VT_begin(123);        /* 1st executable line */
    ...
    VT_end(123);          /* at EVERY exit point! */
}
```

- VT calls can only be used between **`MPI_Init`** and **`MPI_Finalize`**!
- Event numbers used must be globally unique
- Selective tracing: **`VT_traceoff(),VT_traceon()`**

# *VT++.h – C++ Class Wrapper for Vampirtrace*

```
#ifndef __VT_PLUSPLUS_
#define __VT_PLUSPLUS_
#include "VT.h"
class VT_Trace {
  public:  VT_Trace(int code) {VT_begin(code_ = code);}
          ~VT_Trace()         {VT_end(code_);}
  private: int code_;
};
#endif /* __VT_PLUSPLUS_ */
```

- Same tricks can be used to wrap other C++ tracing APIs
- Usage:

```
VT_symdef(123, "foo", "USER"); // symbol definition as before
void foo(void) {               // user subroutine to monitor
  VT_Trace vt(123);            // declare VT_Trace object in 1st line
  ...                          // => automatic tracing by ctor/dtor
}
```

# *Vampirtrace Instrumentation API (Fortran)*

- Calls for recording user subroutines

```
      include 'VT.inc'
      integer ierr
      call VTSYMDEF(123, "foo", "USER", ierr)    !or
      call VTSYMDEFL(123, "foo", "USER", "foo.f:8", ierr)
C
      SUBROUTINE foo(...)
      include 'VT.inc'
      integer ierr
      call VTBEGIN(123, ierr)
C     ...
      call VTEND(123, ierr);
      END
```

- Selective tracing: **`VTTRACEOFF(),VTTRACEON()`**

# *Vampirtrace: Runtime Configuration*

- Trace file collection and generation can be controlled by using a configuration file
  - Trace file name, location, size, flush behavior
  - Activation/deactivation of trace recording for specific processes, activities (groups of symbols), and symbols
- Activate a configuration file with environment variables

| | |
|---|---|
| `VT_CONFIG` | name of configuration file (use absolute pathname if possible) |
| `VT_CONFIG_RANK` | MPI rank of process which should read and process configuration file |

- Reduce trace file sizes
  - Restrict event collection in a configuration file
  - Use selective tracing functions

# *Vampirtrace: Configuration File Example*

```
# collect traces only for MPI ranks 1 to 5
TRACERANKS 1:5:1
# record at most 20000 records per rank
MAX-RECORDS 20000

# do not collect administrative MPI calls
SYMBOL MPI_comm*  off
SYMBOL MPI_cart*  off
SYMBOL MPI_group* off
SYMBOL MPI_type*  off

# do not collect USER events
ACTIVITY USER off
# except routine foo
SYMBOL foo on
```

- Be careful to record complete message transfers!
- See Vampirtrace User's Guide for complete description

# *New Features – Tracing*

- New Vampirtrace versions (3 and 4)
  - New core (significantly reduce memory and runtime overhead)
  - Better control of trace buffering and flush files
  - New filtering options
  - Event recording by thread
  - Support of MPI–I/O
  - Hardware counter data recording (PAPI)
  - Support of process/thread groups

# *Vampirtrace Feature Matrix*

| | Vampirtrace | | Vampirtrace/SC | | Vampirtrace/SX | |
|---|---|---|---|---|---|---|
| | 3 | 4 | 3 | 4 | 1 | 2 |
| New core | yes | yes | yes | yes | yes | yes |
| Buffer control | yes | yes | yes | yes | yes | yes |
| Recover trace | no | yes | no | yes | no | yes |
| New filter options | partial | yes | partial | yes | partial | yes |
| Thread events | no | yes | no | yes | no | no |
| MPI–I/O | yes | yes | no | yes | no | yes |
| Counter data | no | yes | yes | yes | yes | yes |
| Thread/process grouping | no | yes | partial | yes | partial | yes |
| Scalability (processes) | 200 | 500 | 500 | 1000 | 500 | 1000 |

# GuideView

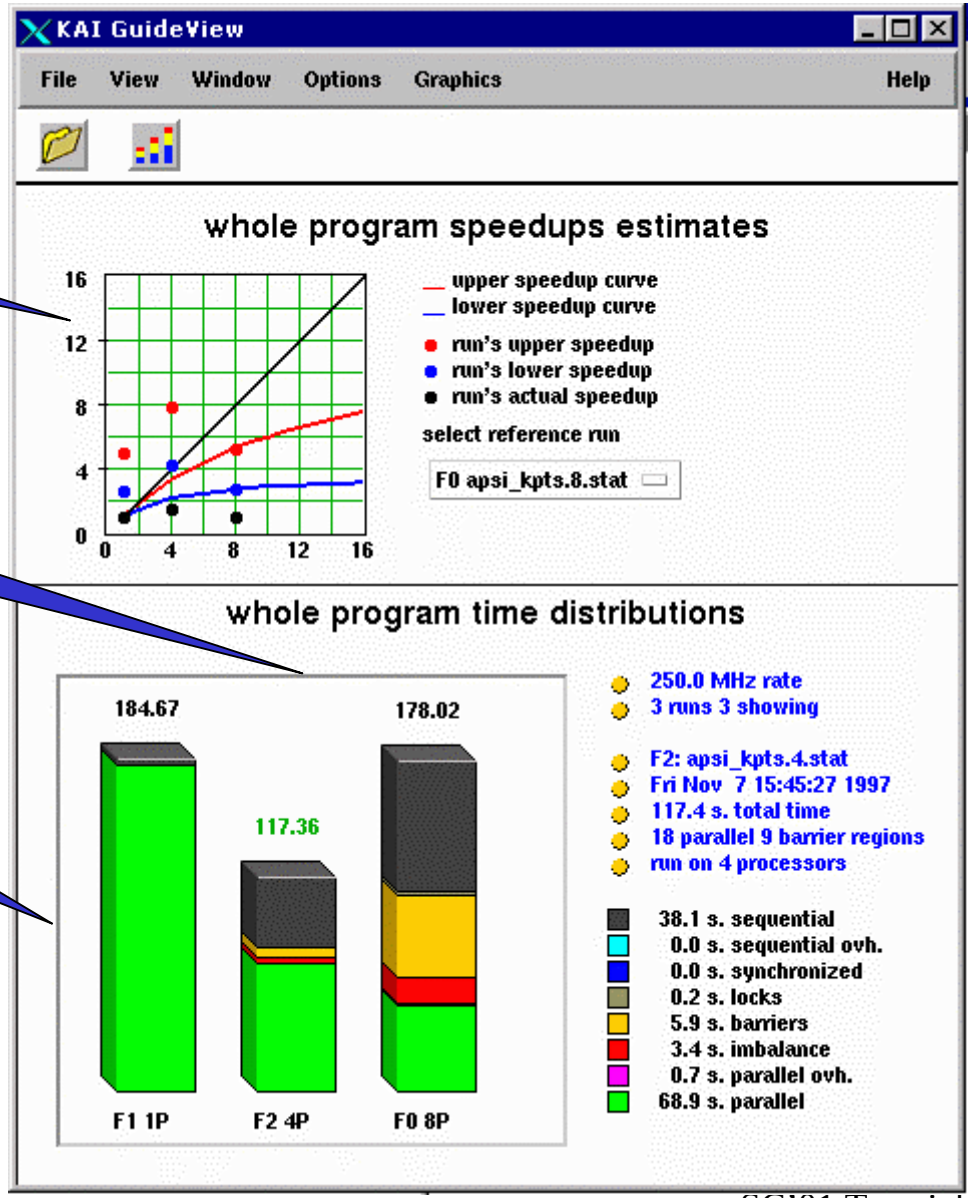

- Commercial product of KAI
- OpenMP Performance Analysis Tool
- Part of KAP/Pro Toolset for OpenMP
- Looks for OpenMP performance problems
  - Load imbalance, synchronization, false sharing
- Works from execution trace(s)
- Compile with Guide, link with instrumented library
  - **`guidec++ -WGstats myprog.cpp -o myprog`**
  - **`guidef90 -WGstats myprog.f90 -o myprog`**
  - Run with real input data sets
  - View traces with **`guideview`**
- **`http://www.kai.com/parallel/kappro/`**

# *GuideView: Whole Application View*

Compare actual vs. ideal performance

Identify bottlenecks (barriers, locks, seq. time)

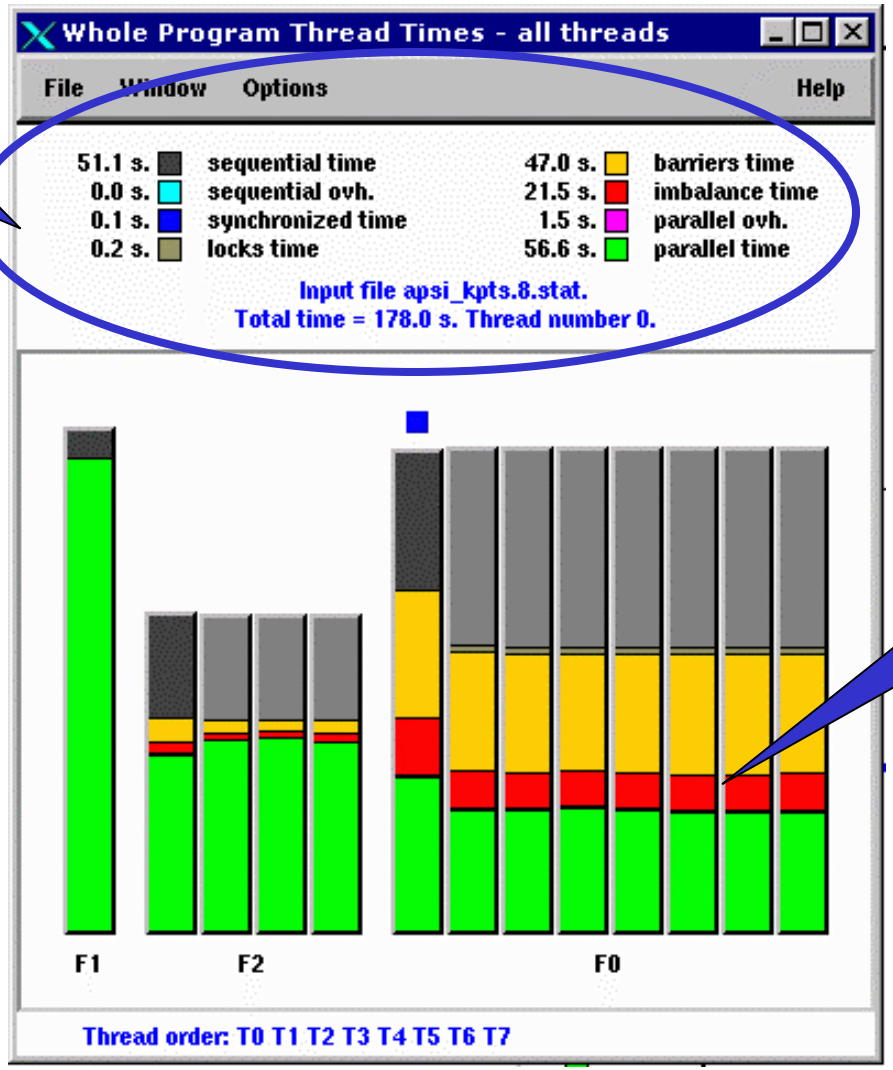
Compare multiple runs




- Different
  - Number of processors
  - Datasets
  - Platforms

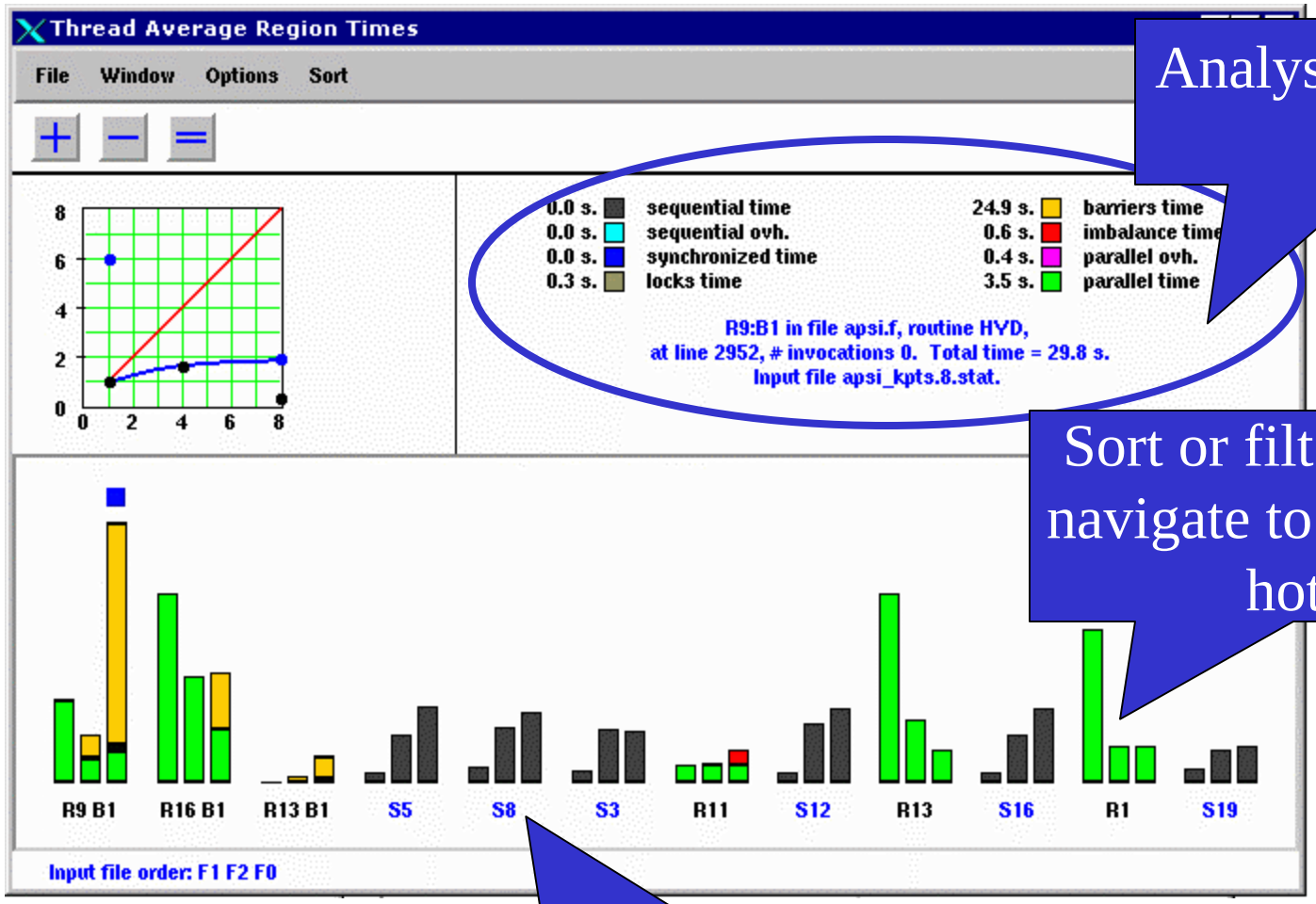
# *GuideView: Per Thread View*

Analyse each thread's performance




Show scalability problems

# *GuideView: Per Section View*




Analyse each parallel region

Sort or filter regions to navigate to performance hotspots

Identify serial regions that hurt scalability

# *GuideView: Analysis of hybrid Applications*

- Generate different Guide execution traces for each node
  - Run with node-local file system as current directory
  - Set trace file name with environment variable **`KMP_STATSFILE`**
    - point to file in node-local file system
      **`KMP_STATSFILE=/node-local/guide.gvs`**
    - use special meta-character sequences
      (**`%H`**: hostname, **`%I`**: pid, **`%P`**: number of threads used)
      **`KMP_STATSFILE=guide-%H.gvs`**
- Use "compare-multiple-run" feature to display together

- Just a hack, better: use VGV!

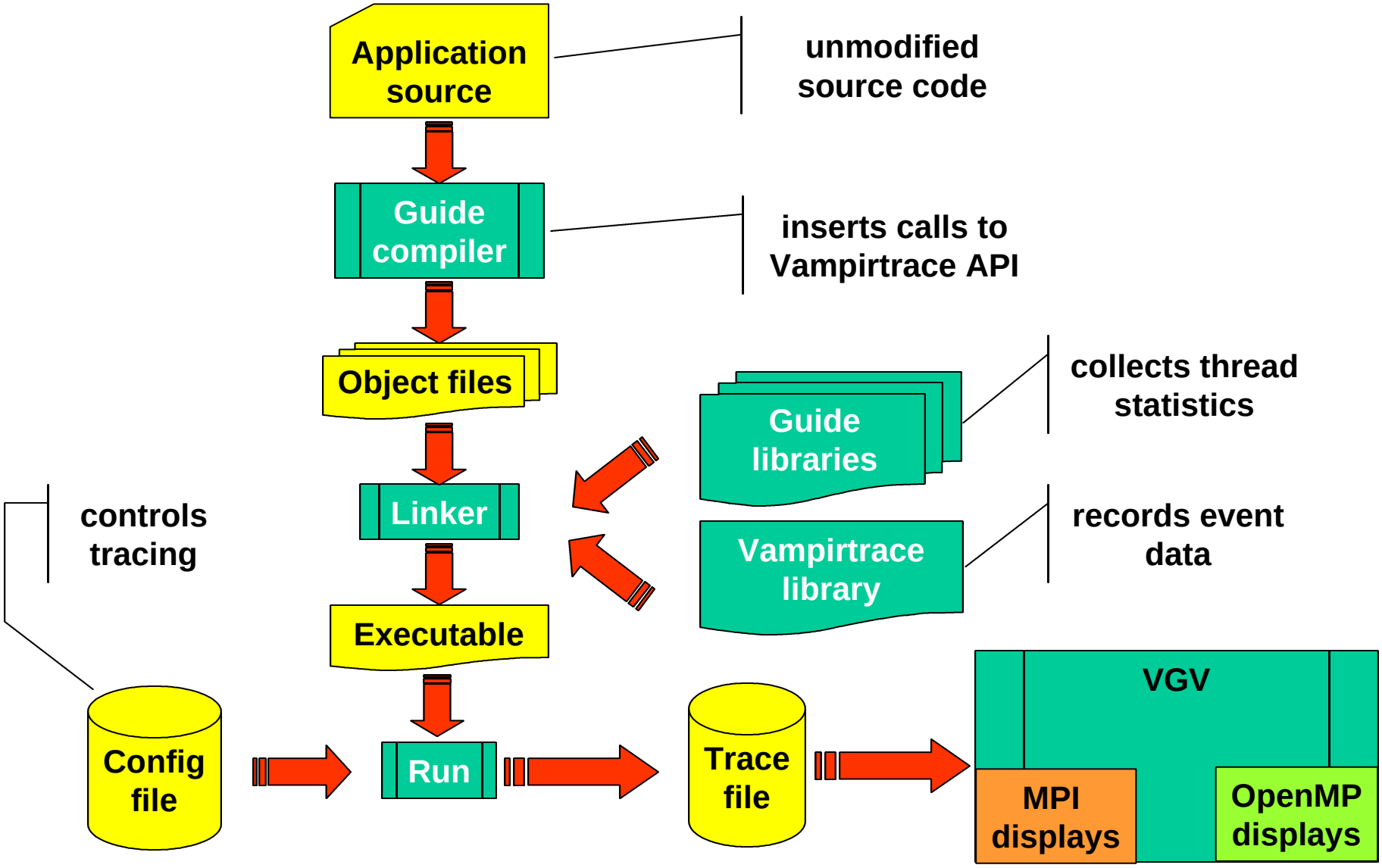
# *VGV – Architecture*

- Combine well–established tools
  - Guide and GuideView from KAI/Intel
  - Vampir/Vampirtrace from Pallas



- Guide compiler inserts instrumentation
- Guide runtime system collects thread–statistics
- PAPI is used to collect HPM data
- Vampirtrace handles event–based performance data acquisition and storage
- Vampir is extended by GuideView–style displays

# VGV – Architecture



# *VGV – Usage*

- Use Guide compilers by KAI
  - **`guidef77`**, **`guidef90`**
  - **`guidec`**, **`guidec++`**
- Include instrumentation flags (links with Guide RTS and Vampirtrace)
- Instrumentation can record
  - Parallel regions
  - MPI activity
  - Application routine calls
  - HPM data
- Trace file collection and generation controlled by configuration file

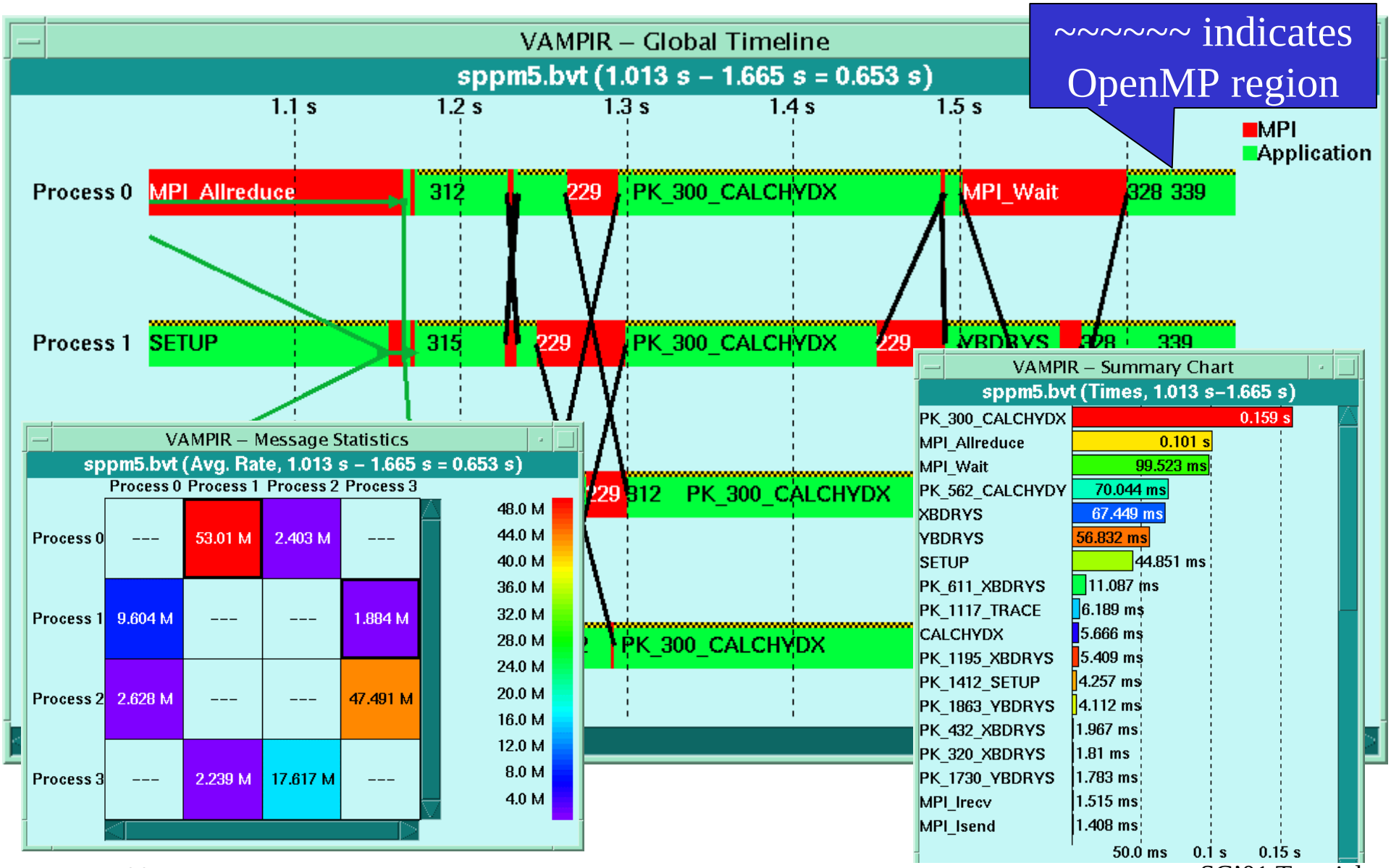
**Instrumentation flags for Guide**

**`-WGtrace`**
compile and link with Vampirtrace

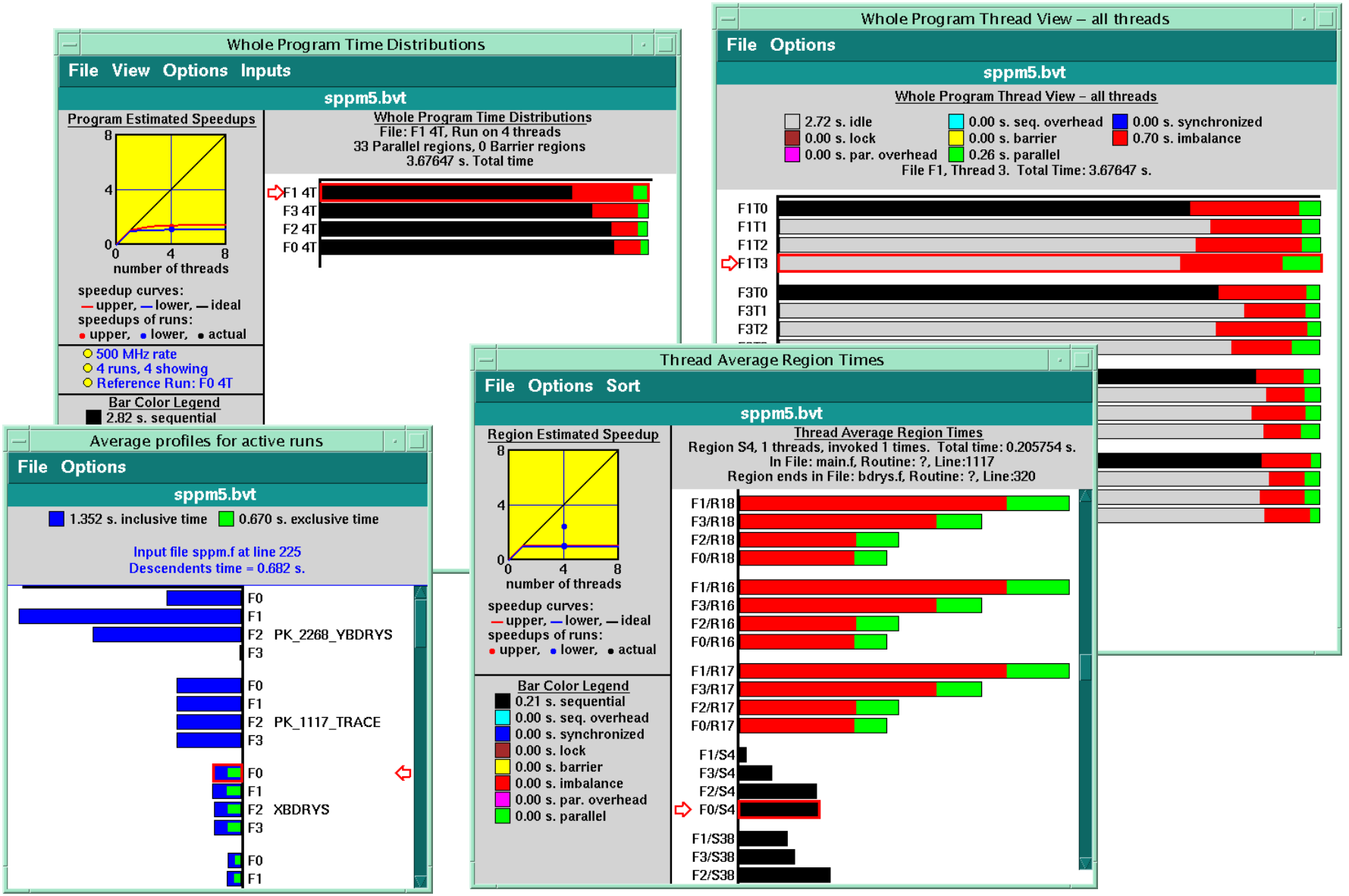
**`-WGprof`**
include routine entry/exit profiling

**`- WGprof_leafprune=`*`N`***
minimum size of procedures to retain in profile

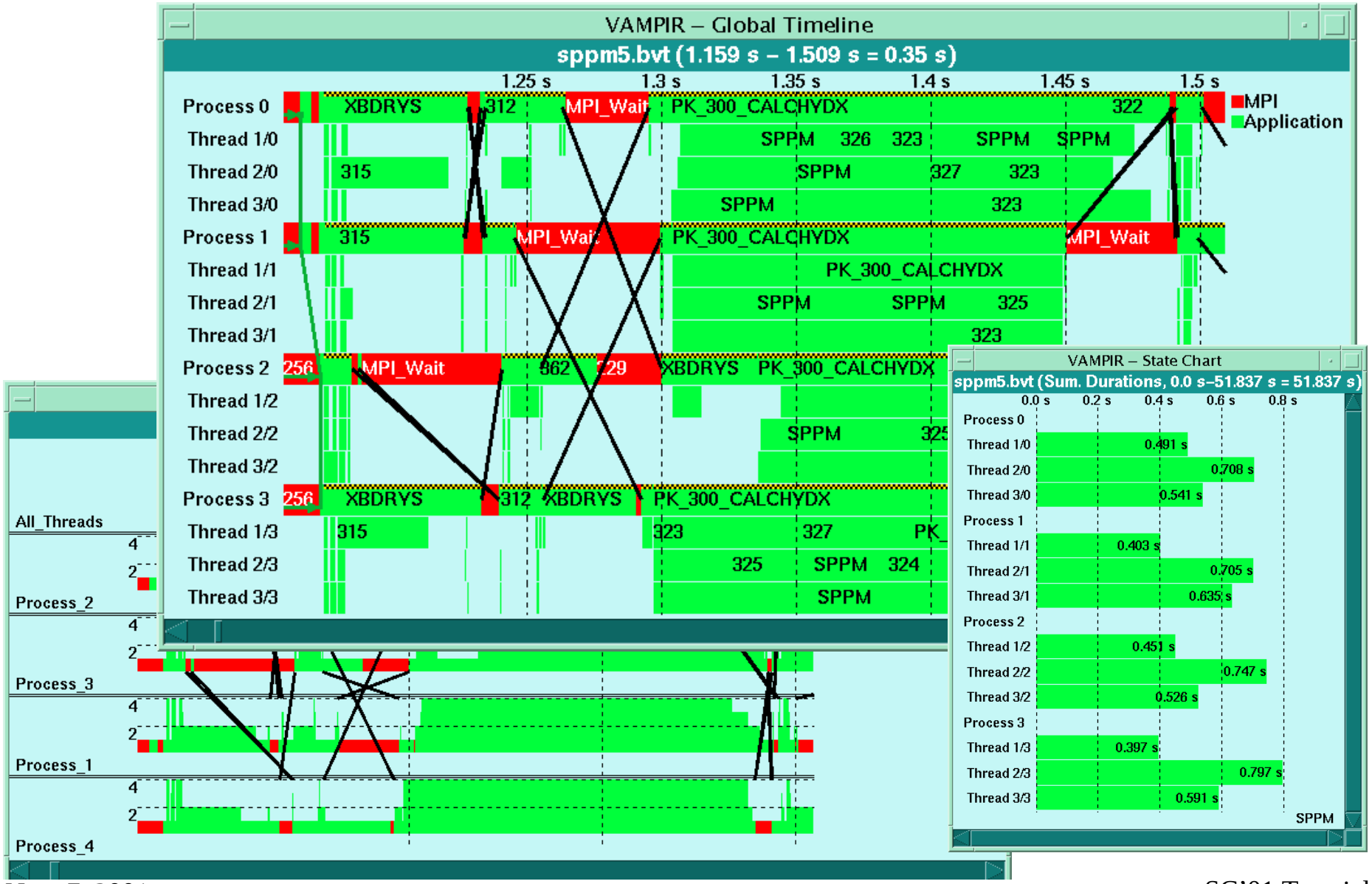
# *Vampir: MPI Performance Analysis*



# GuideView: OpenMP Performance Analysis



# *Vampir: Detailed Thread Analysis*



# *Availability and Roadmap*

- β–version available (register with Pallas or KAI/Intel)
  - IBM SP running AIX
  - IA 32 running Linux
  - Compaq Alpha running Tru64
- General release scheduled for Q1/2002
- Improvements in the pipeline
  - Scalability enhancements
  - Ports to other platforms

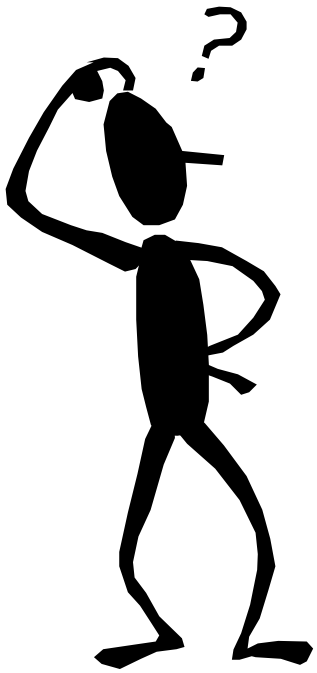
# *KOJAK Overview*

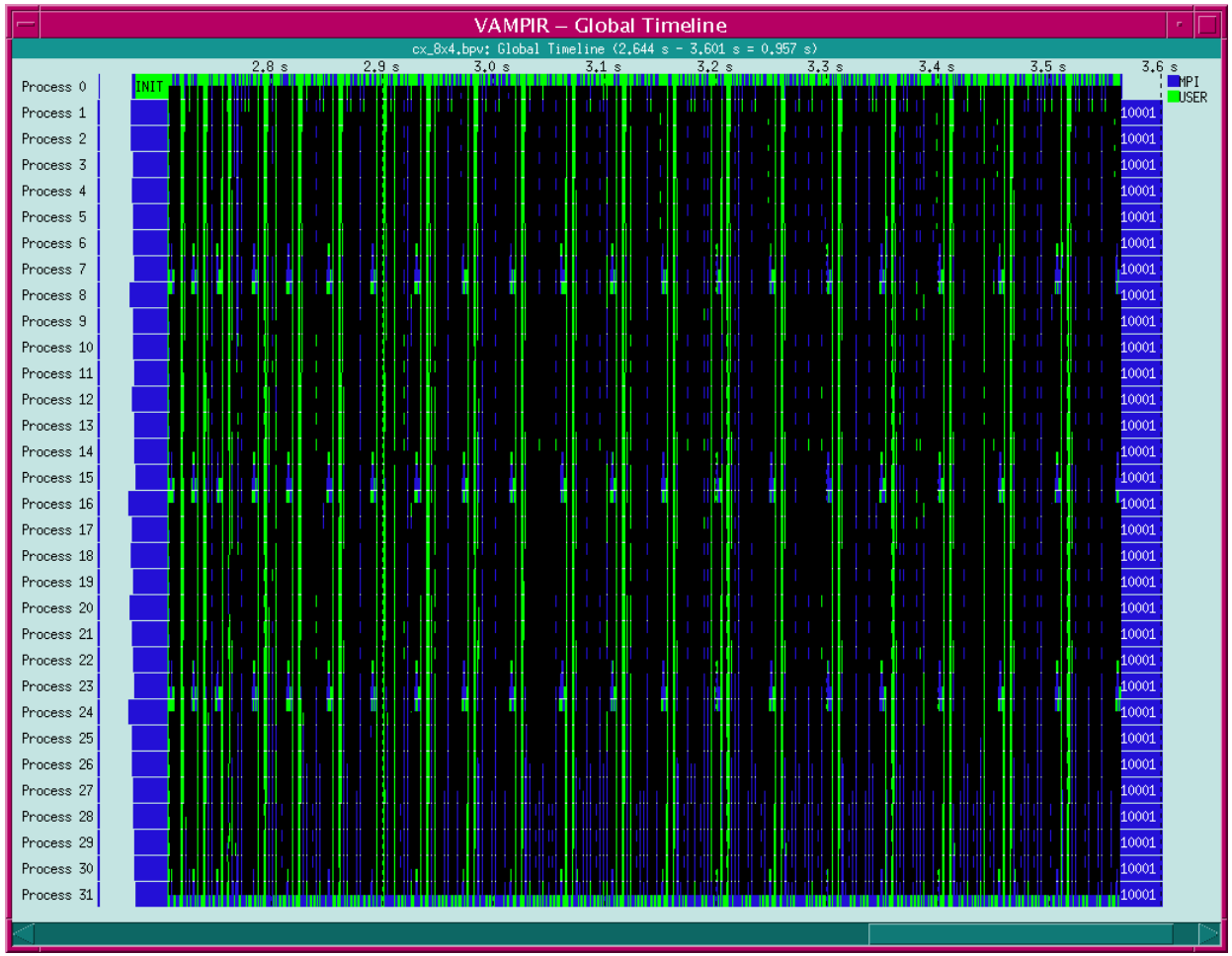
- **K**it for **O**bjective **J**udgement and **A**utomatic **K**nowledge-based detection of bottlenecks
- Long-term goal:
  Design and Implementation of a
  - Portable, Generic, Automatic

  Performance Analysis Environment
- Current Focus
  - Event Tracing
  - Clusters of SMP
  - MPI, OpenMP, and Hybrid Programming Model
- **`http://www.fz-juelich.de/zam/kojak/`**

# *Motivation Automatic Performance Analysis*

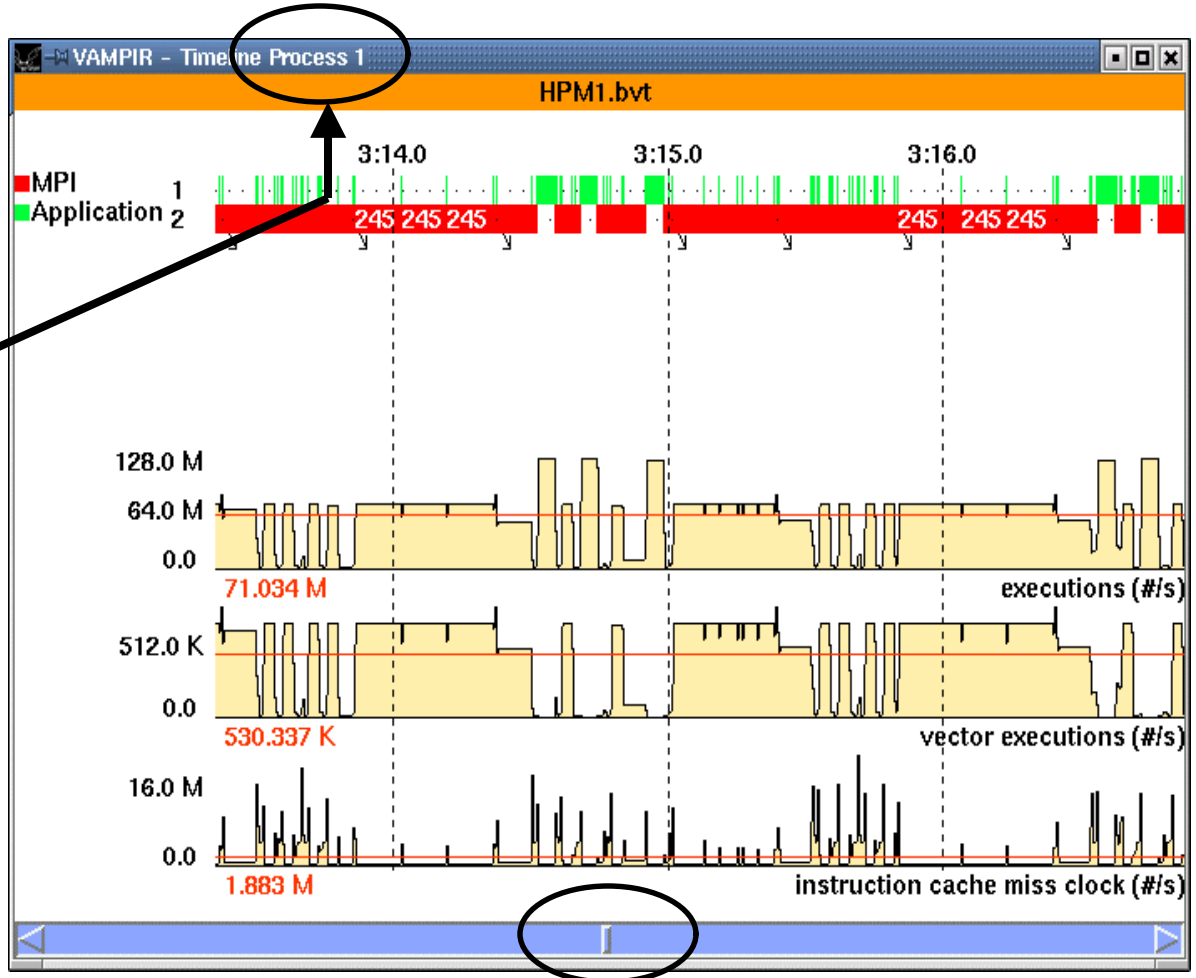
Traditional Tools:



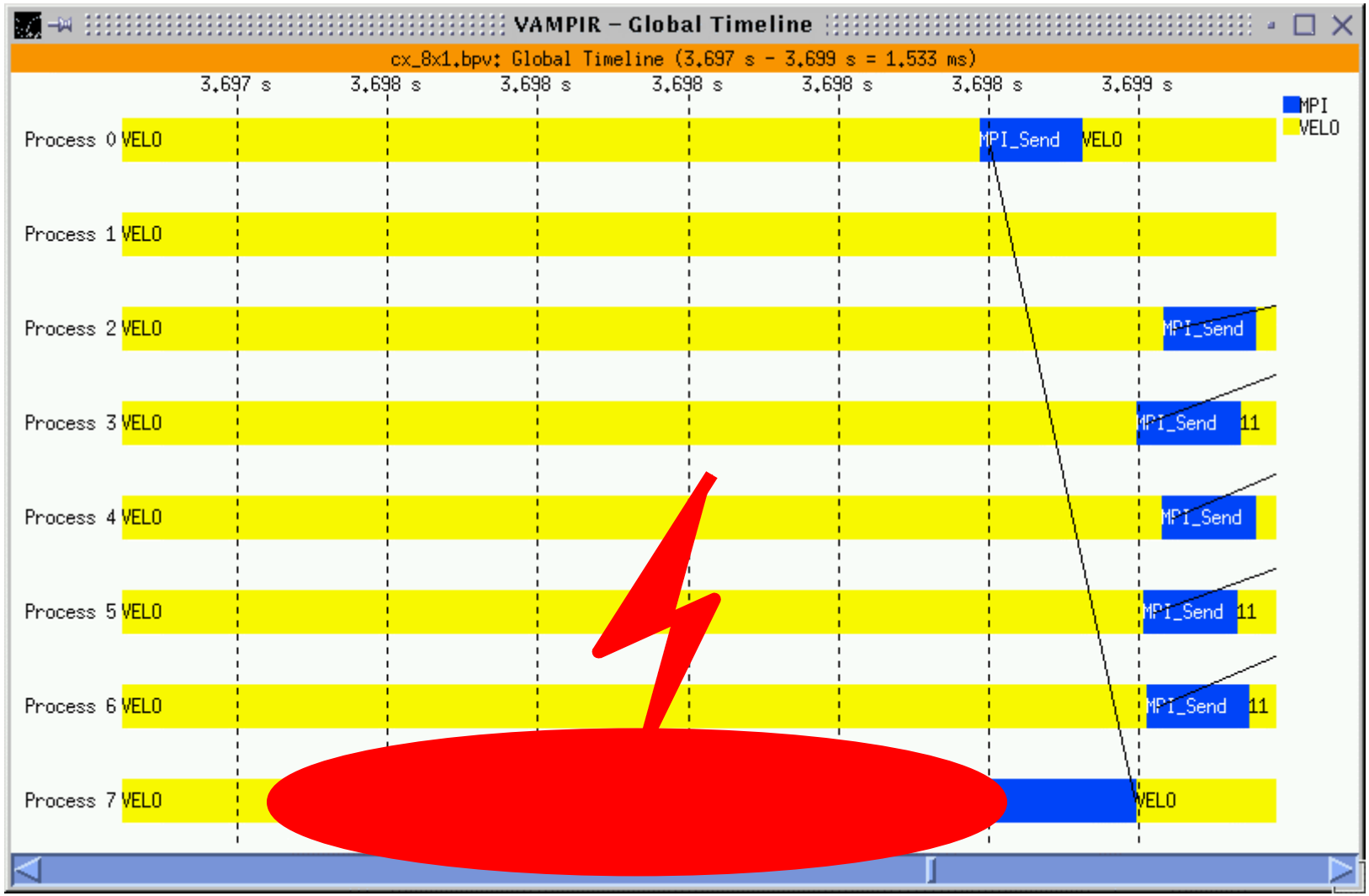


# *Motivation Automatic Performance Analysis (2)*

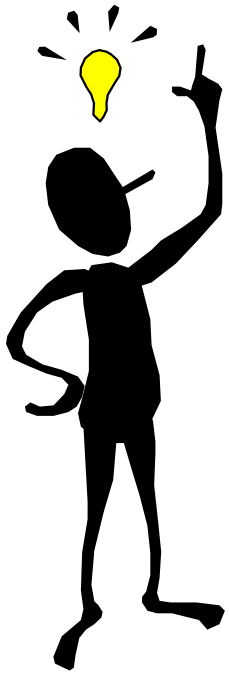
After lots of zooming and selecting:

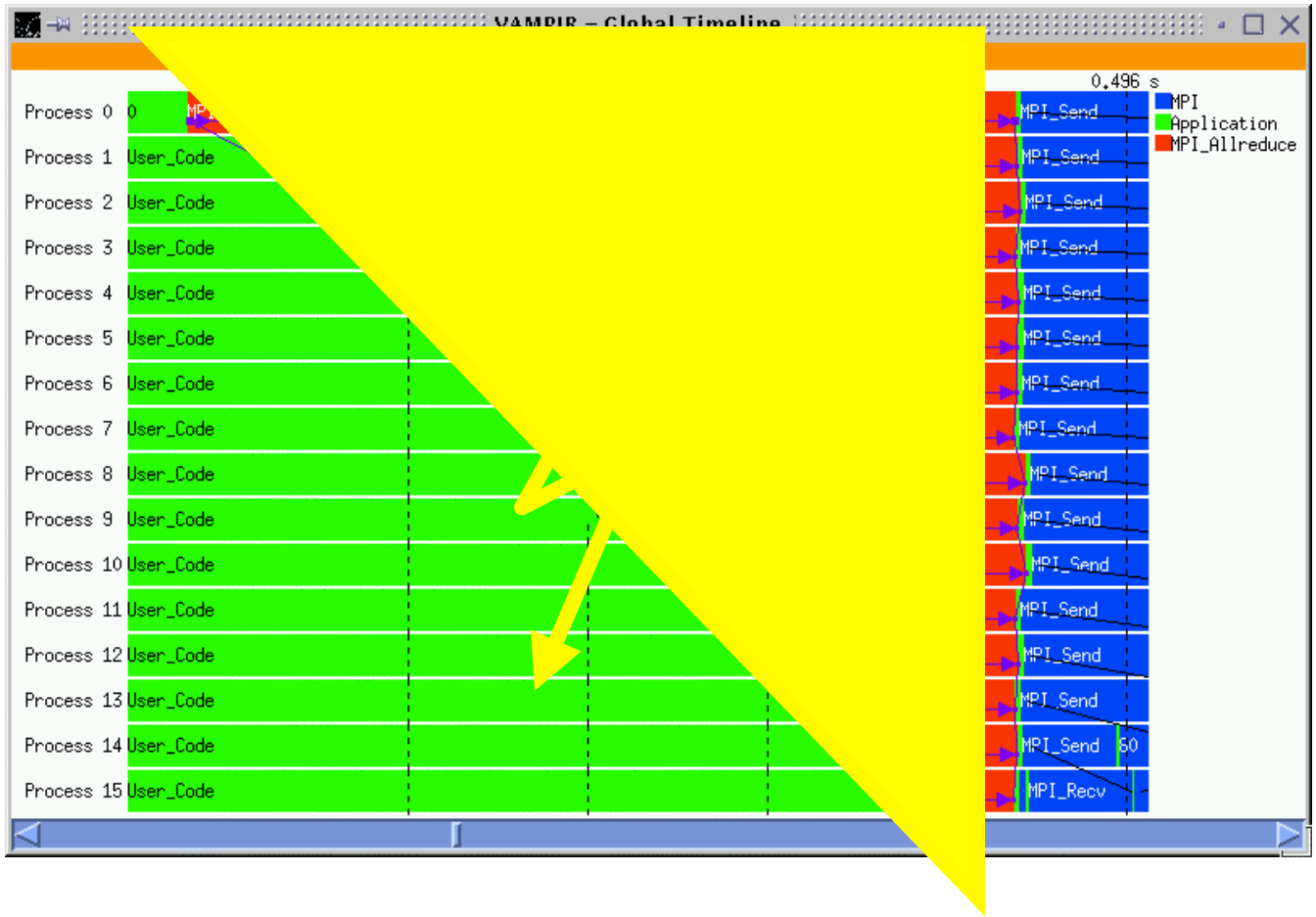


# *Automatic Analysis Example: Late Sender*

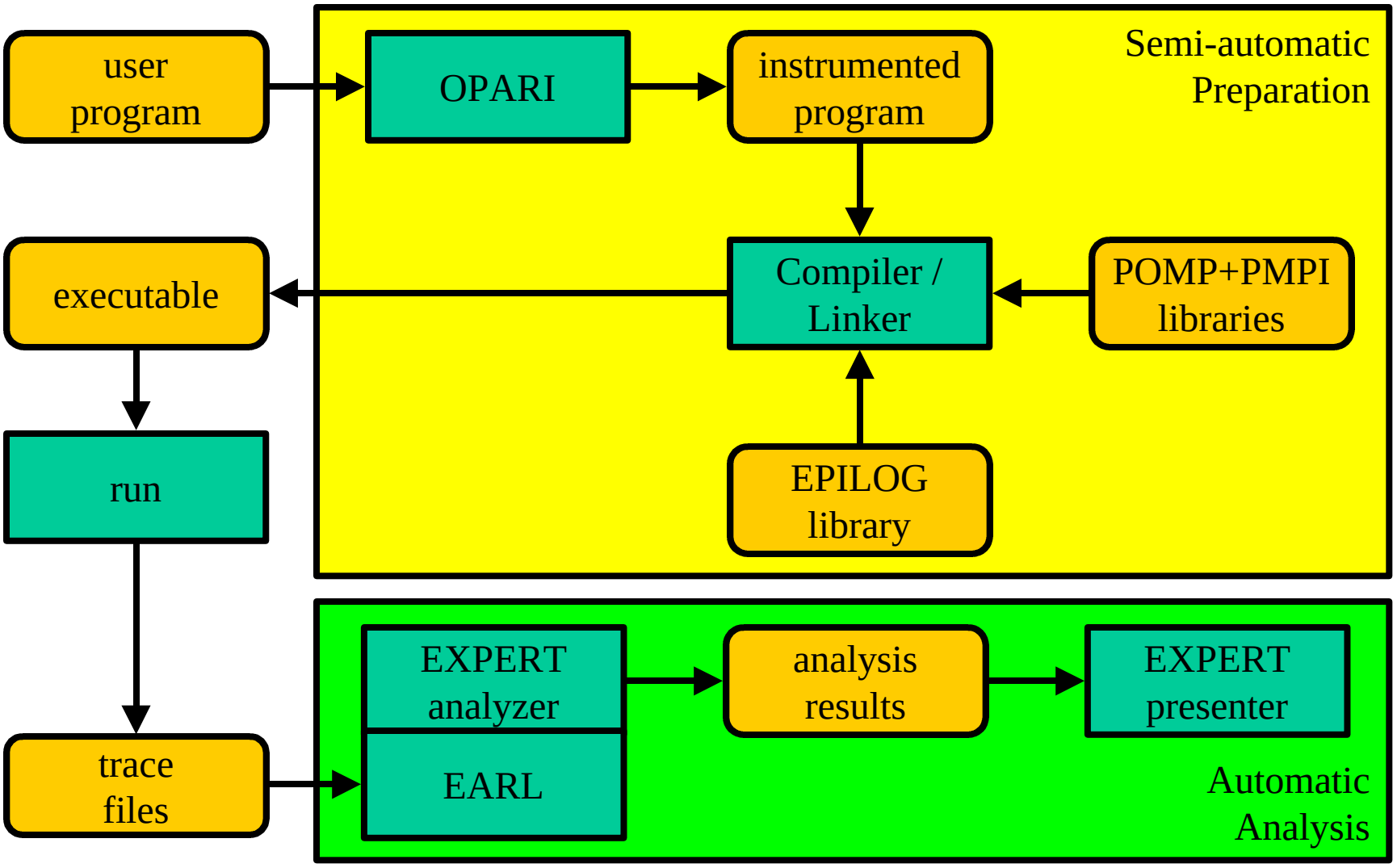


# *Automatic Analysis Example (2): Wait at NxN*





# EXPERT: Current Architecture



# *Event Tracing*

- **E**vent **P**rocessing, **I**nvestigation, and **LOG**ging (EPILOG)
- Open (public) event trace format and API for reading/writing trace records
- Event Types: region enter and exit, collective region enter and exit, message send and receive, parallel region fork and join, and lock aquire and release
- Supports
  - Hierarchical cluster hardware
  - Source code information
  - Performance counter values
- Thread-safe implementation

# *Instrumentation*

- Instrument user application with EPILOG calls
- Done: basic instrumentation
  - User functions and regions:
    - undocumented PGI compiler (and manual) instrumentation
  - MPI calls:
    - wrapper library utilizing PMPI
  - OpenMP:
    - source-to-source instrumentation
- Future work:
  - Tools for Fortran, C, C++ user function instrumentation
  - Object code and dynamic instrumentation

# *Instrumentation of OpenMP Constructs*


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

# *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: 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

# OPARI: Makefile Template (C/C++)

```
OMPCC  = ...          # insert C OpenMP compiler here
OMPCXX = ...          # insert C++ OpenMP compiler here

.c.o:
       opari $<
       $(OMPCC) $(CFLAGS) -c $*.mod.c
.cc.o:
       opari $<
       $(OMPCXX) $(CXXFLAGS) -c $*.mod.cc

opari.init:
       rm -rf opari.rc

opari.tab.o:
       opari -table opari.tab.c
       $(CC) -c opari.tab.c

myprog: opari.init myfile*.o ... opari.tab.o
       $(OMPCC) -o myprog myfile*.o opari.tab.o -lpomp

myfile1.o: myfile1.c myheader.h
myfile2.o: ...
```

# OPARI: Makefile Template (Fortran)

```
OMPF77 = ...          # insert f77 OpenMP compiler here
OMPF90 = ...          # insert f90 OpenMP compiler here

.f.o:
       opari $<
       $(OMPF77) $(CFLAGS) -c $*.mod.F
.f90.o:
       opari $<
       $(OMPF90) $(CXXFLAGS) -c $*.mod.F90

opari.init:
       rm -rf opari.rc

opari.tab.o:
       opari -table opari.tab.c
       $(CC) -c opari.tab.c

myprog: opari.init myfile*.o ... opari.tab.o
       $(OMPF90) -o myprog myfile*.o opari.tab.o -lpomp

myfile1.o: myfile1.f90
myfile2.o: ...
```

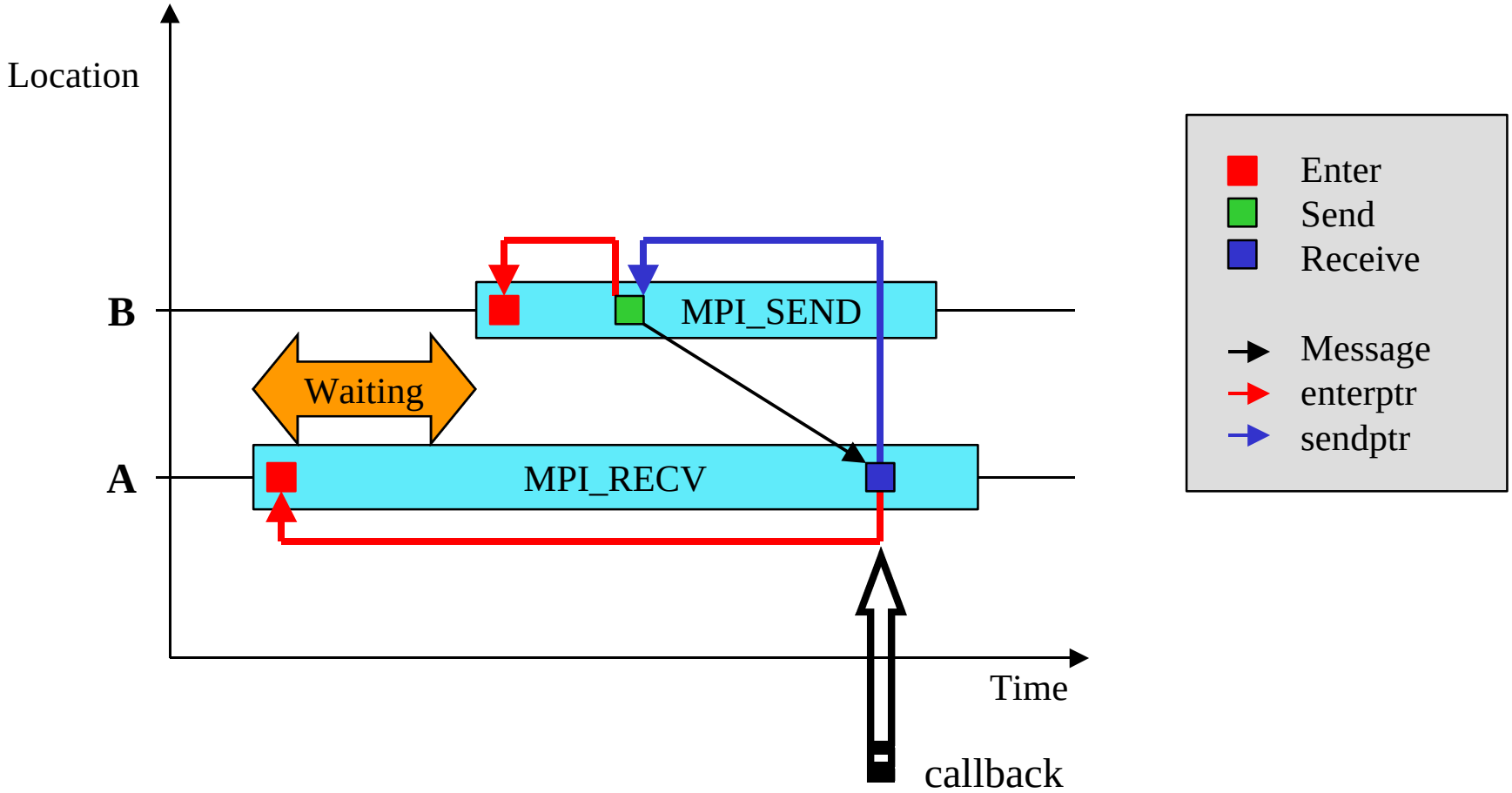
# *Automatic Analysis*

- **EX**tensible **PER**formance **T**ool (EXPERT)
- Programmable, extensible, flexible performance property specification
- Based on event patterns

- Analyzes along three hierarchical dimensions
  - Performance properties (general → specific)
  - Dynamic call tree position
  - Location (machine → node → process → thread)

- Done: fully functional demonstration prototype

# *Example: Late Sender (blocked receiver)*



# *Example: Late Sender (2)*

```
class LateSender(Pattern):  # derived from class Pattern
def parent(self):           # "logical" parent at property
   level
   return "P2P"
def recv(self, recv):       # callback for recv events
   recv_start = self._trace.event(recv['enterptr'])
      if (self._trace.region(recv_start['regid'])['name']
          == "MPI_Recv"):
        send       = self._trace.event(recv['sendptr'])
        send_start = self._trace.event(send['enterptr'])
        if (self._trace.region(send_start['regid'])['name']
            == "MPI_Send"):
          idle_time =  send_start['time'] - recv_start['time']
          if idle_time > 0 :
            locid  = recv_start['locid']
            cnode  = recv_start['cnodeptr']
            self._severity.add(cnode, locid, idle_time)
```

# *Performance Properties (1)*

[100% = ($time_{last\ event}$ - $time_{1st\ event}$) * number of locations]

- Total # Execution + Idle Threads time
  - Execution # Sum of exclusive time spent in each region
  - Idle Threads # Time wasted in idle threads while executing “sequential” code
- Execution
  - MPI # Time spent in MPI functions
  - OpenMP # Time spent in OpenMP regions and API functions
  - I/O # Time spent in (sequential) I/O

# *Performance Properties (2)*

- MPI
  - Communication — # Sum of Collective, P2P, 1-sided
    - Collective — # Time spent in MPI collective communication operations
    - P2P — # Time spent in MPI point-to-point communication operations
    - 1-sided — # Time spent in MPI one-sided communication operations
  - I/O — # Time spent in MPI parallel I/O functions (**`MPI_File*`**)
  - Synchronization — # Time spent in **`MPI_Barrier`**

# *Performance Properties (3)*

- Collective
  - Early Reduce # Time wasted in root of N-to-1 operation by waiting for 1st sender (**`MPI_Gather`**, **`MPI_Gatherv`**, **`MPI_Reduce`**)
  - Late Broadcast # Time wasted by waiting for root sender in 1-to-N operation (**`MPI_Scatter`**, **`MPI_Scatterv`**, **`MPI_Bcast`**)
  - Wait at N x N # Time spent waiting for last participant at NxN operation (**`MPI_All*`**, **`MPI_Scan`**, **`MPI_Reduce_scatter`**)

# *Performance Properties (4)*

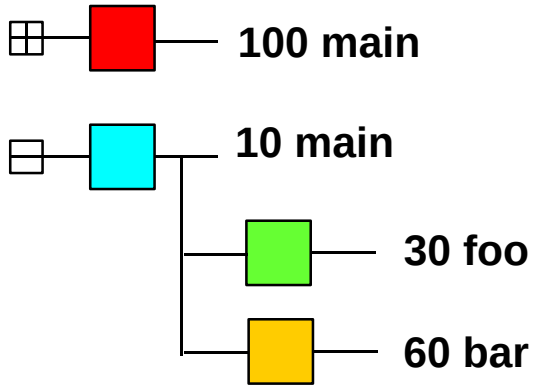
- P2P
  - Late Receiver — # Blocked sender
    - Messages in Wrong Order — # Receiver too late because waiting for another message from same sender
  - Late Sender — # Blocked receiver
    - Messages in Wrong Order — # Receiver blocked because waiting for another message from same sender
  - Patterns related to non-blocking communication
  - Too many small messages

# *Performance Properties (5)*

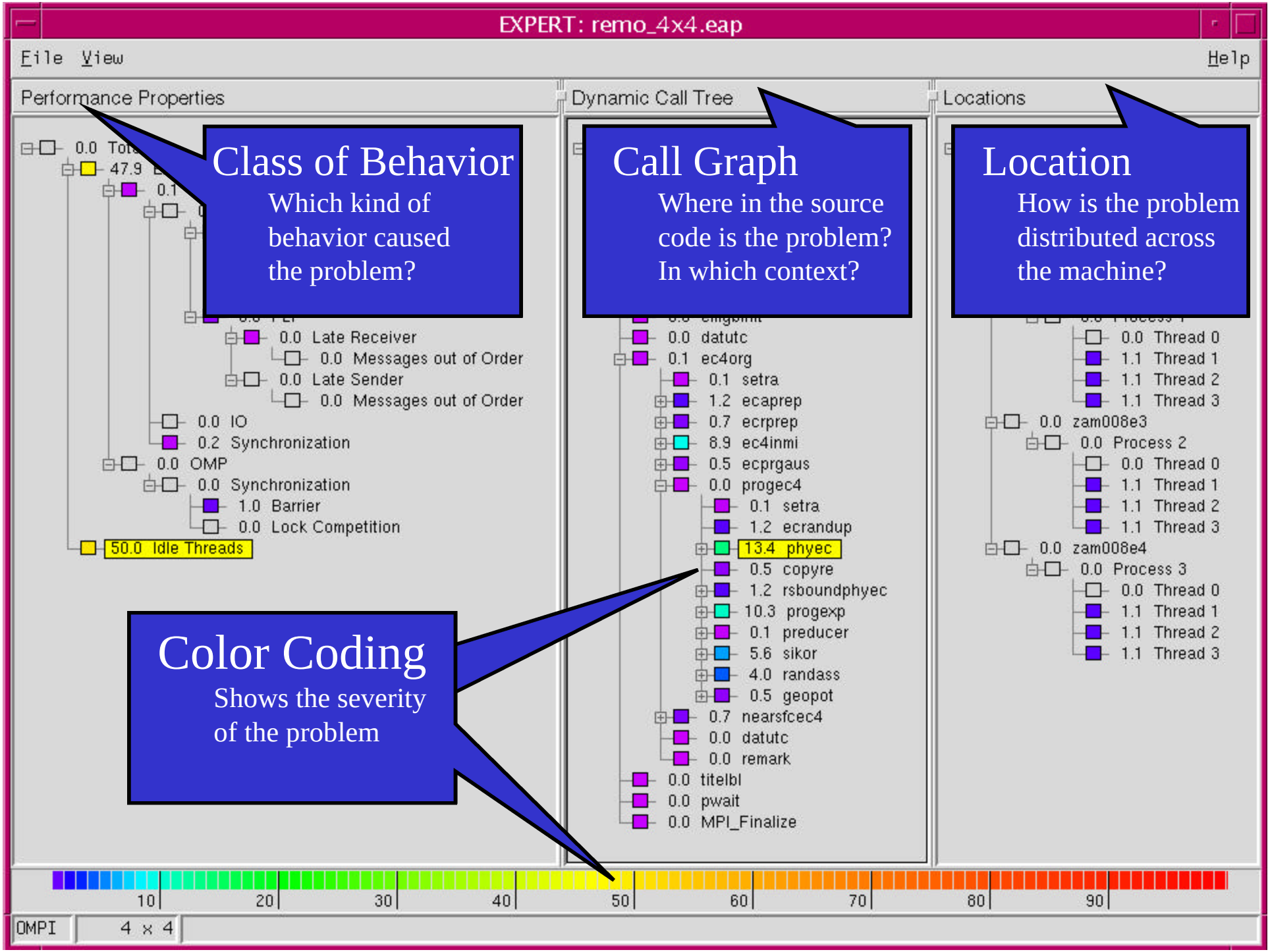
- OpenMP
  - Synchronization # Time spent in OpenMP barrier and lock operations
    - Barrier # Time spent in OpenMP barrier operations
      - Implicit
        - » Load Imbalance at Parallel Do, Single, Workshare
        - » Not Enough Sections
      - Explicit
    - Lock Competition # Time wasted in **`omp_set_lock`** by waiting for lock release
  - Flush

# *Expert Result Presentation*

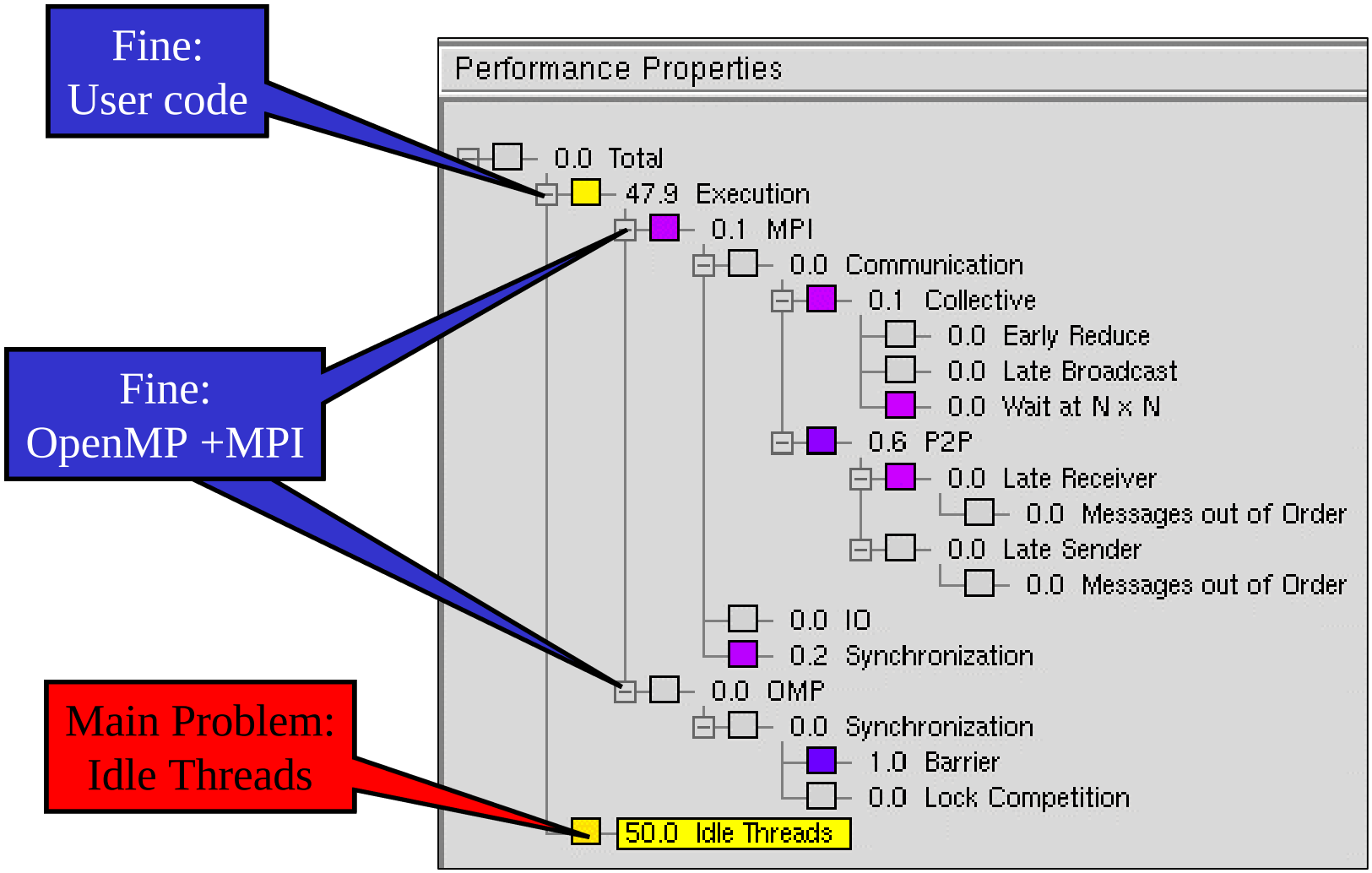
- Interconnected weighted tree browser
- Scalable still accurate
- Each node has weight
  - Percentage of CPU allocation time
  - I.e. time spent in subtree of call tree




- Displayed weight depends on state of node
  - Collapsed (including weight of descendants)
  - Expanded (without weight of descendants)
- Displayed using
  - Color:  allows to easily identify hot spots (bottlenecks)
  - Numerical value:  Detailed comparison

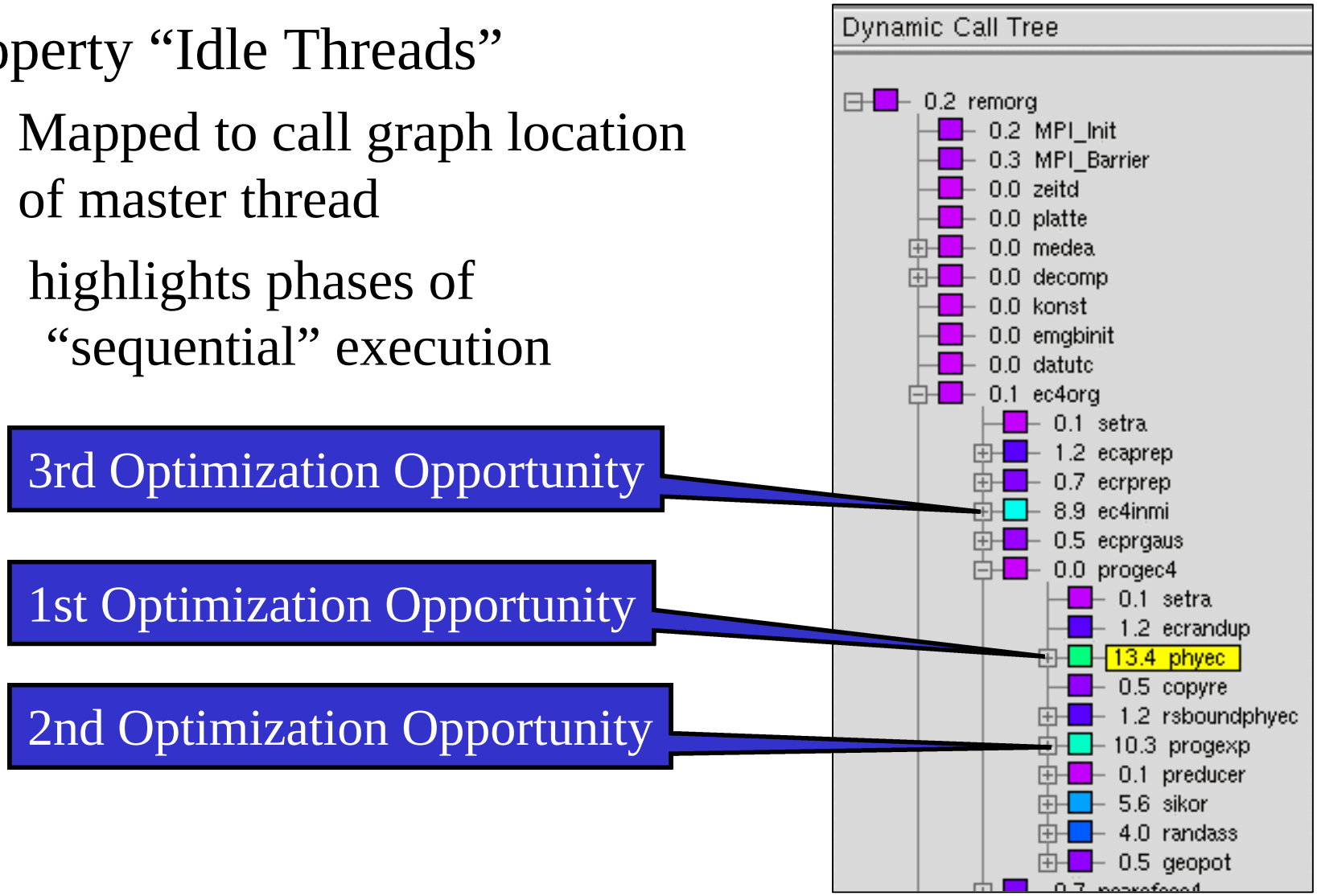

EXPERT: remo_4x4.eap
File View
Help
Performance Properties
Dynamic Call Tree
Locations
Class of Behavior
Which kind of behavior caused the problem?
Call Graph
Where in the source code is the problem? In which context?
Location
How is the problem distributed across the machine?
Color Coding
Shows the severity of the problem
0.0 Late Receiver
0.0 Messages out of Order
0.0 Late Sender
0.0 Messages out of Order
0.0 IO
0.2 Synchronization
0.0 OMP
0.0 Synchronization
1.0 Barrier
0.0 Lock Competition
50.0 Idle Threads
0.0 datutc
0.1 ec4org
0.1 setra
1.2 ecaprep
0.7 ecrprep
8.9 ec4inmi
0.5 ecprgaus
0.0 progec4
0.1 setra
1.2 ecrandup
13.4 phyec
0.5 copyre
1.2 rsboundphyec
10.3 progexp
0.1 preducer
5.6 sikor
4.0 randass
0.5 geopot
0.7 nearsfcec4
0.0 datutc
0.0 remark
0.0 titelbl
0.0 pwait
0.0 MPI_Finalize
0.0 Thread 0
1.1 Thread 1
1.1 Thread 2
1.1 Thread 3
0.0 zam008e3
0.0 Process 2
0.0 Thread 0
1.1 Thread 1
1.1 Thread 2
1.1 Thread 3
0.0 zam008e4
0.0 Process 3
0.0 Thread 0
1.1 Thread 1
1.1 Thread 2
1.1 Thread 3
10
20
30
40
50
60
70
80
90
OMPI
4 x 4

# *Performance Properties View*



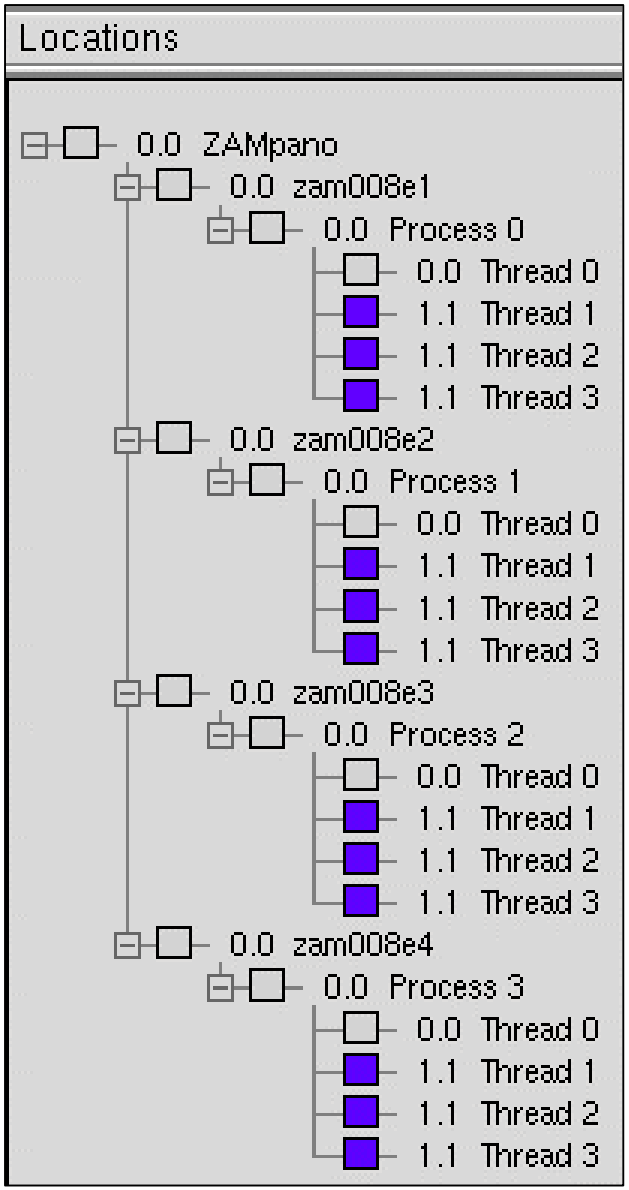
# *Dynamic Call Tree View*

- Property “Idle Threads”
  - Mapped to call graph location of master thread
  - ⇒ highlights phases of “sequential” execution



# *Locations View*

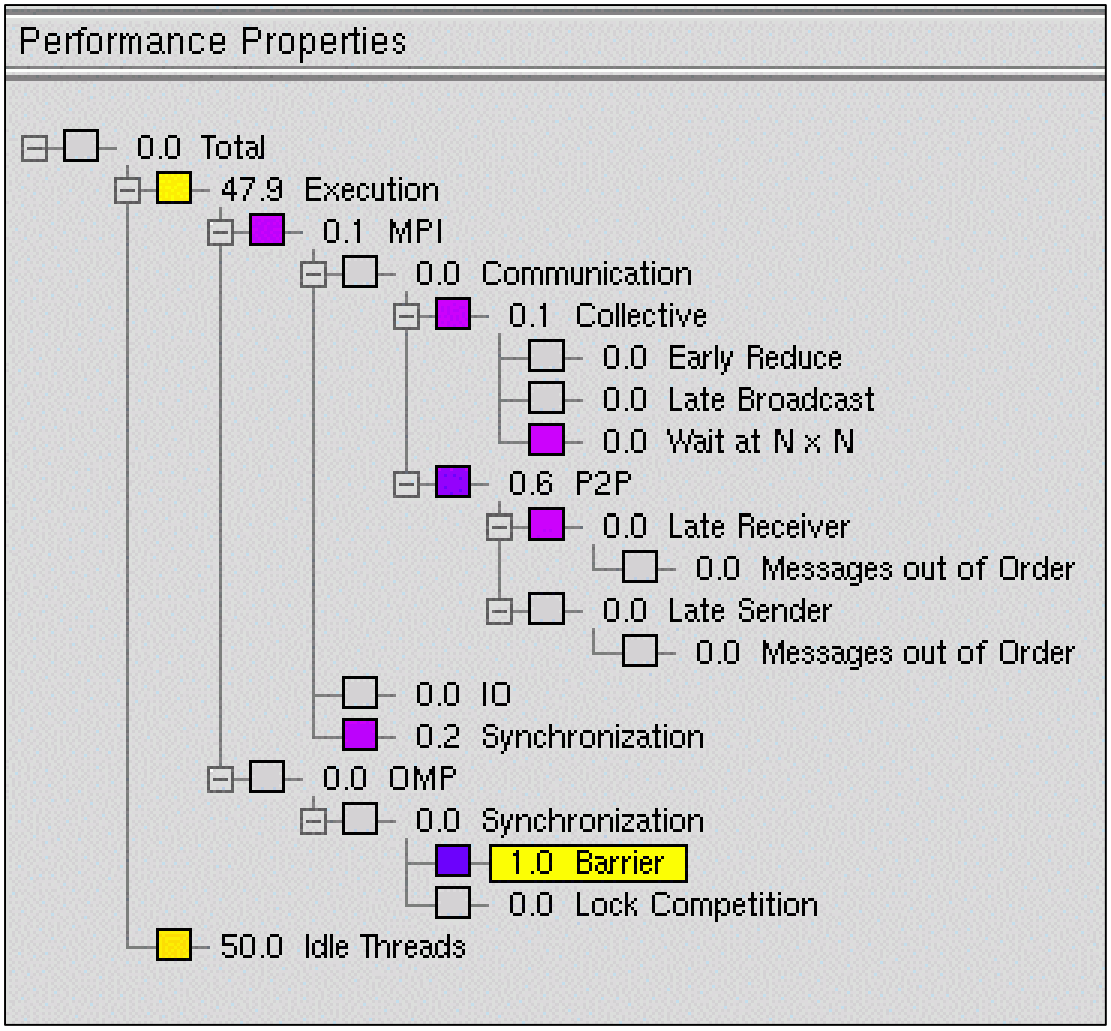
- Supports locations up to Grid scale
- Easily allows exploration of load balance problems on different levels
- [ Of course, Idle Thread Problem only applies to slave threads ]



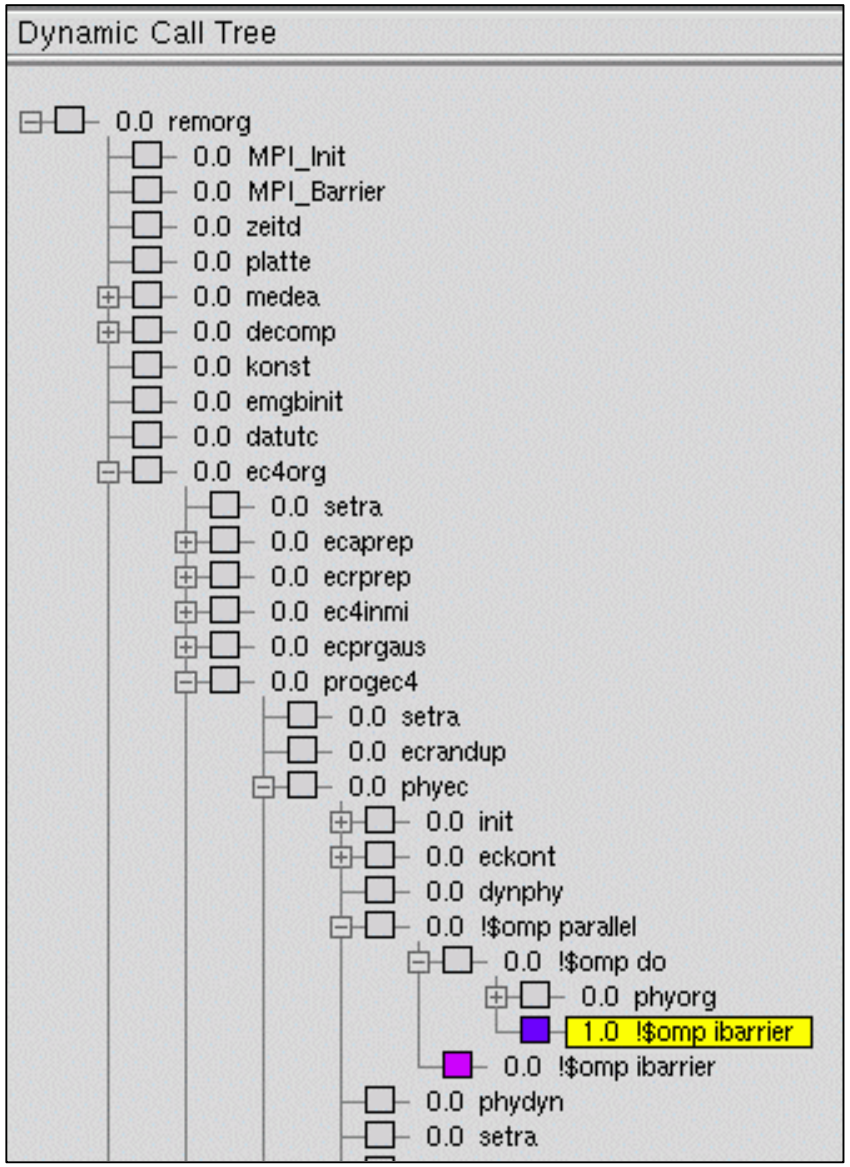
# *Performance Properties View (2)*

- Interconnected weighted trees:

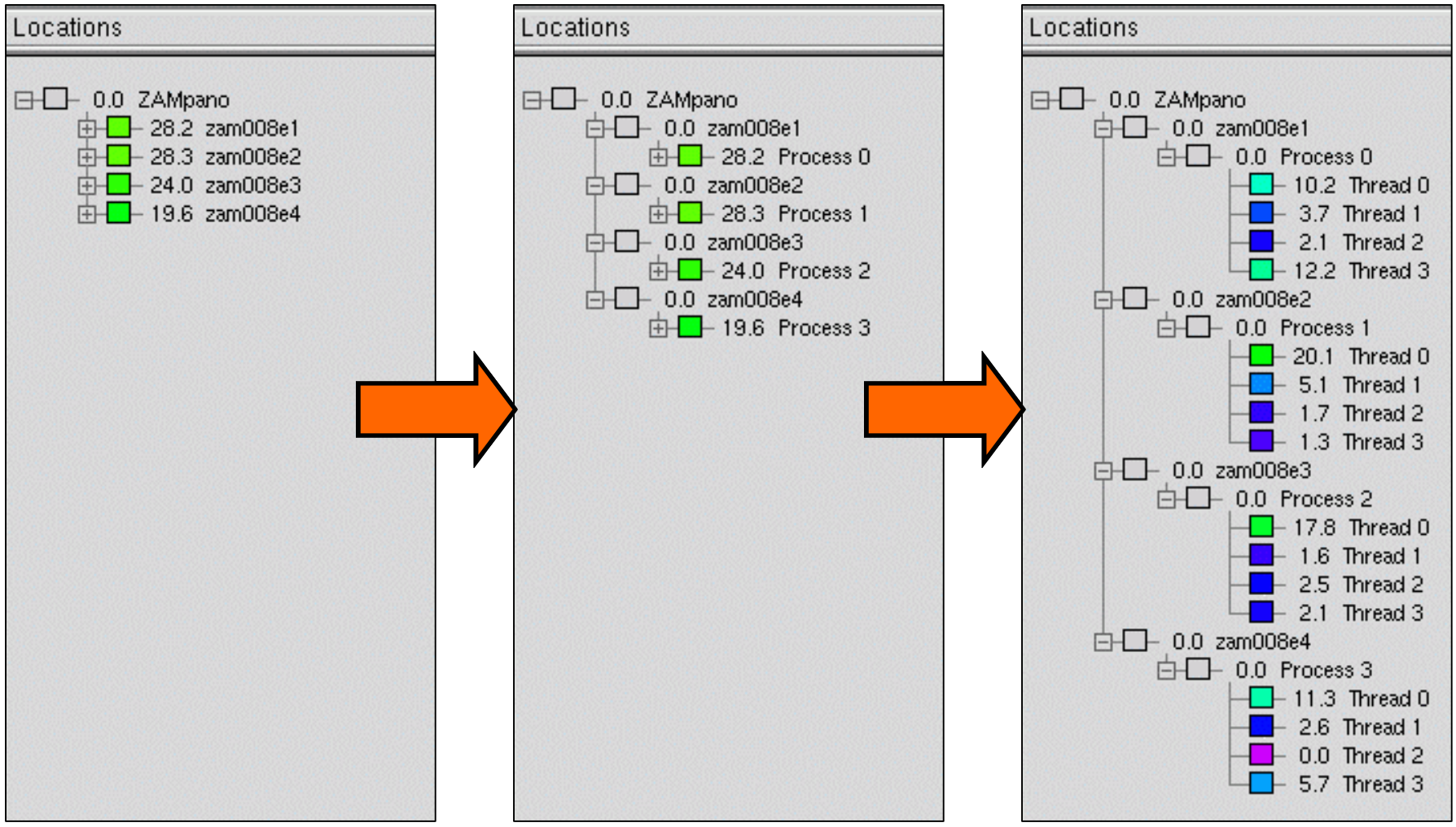
  ⇒ Selecting another node in one tree effects tree display right of it



# *Dynamic Call Tree View*



# *Locations View (2): Relative View*



# *Automatic Performance Analysis*


**A**utomatic **P**erformance **A**nalysis: **R**esources and **T**ools

**`http://www.fz-juelich.de/apart/`**

- ESPRIT Working Group 1999 - 2000
- IST Working Group 2001 - 2004
- 16 members worldwide
- Prototype Tools (Paradyn, Kappa-PI, Aurora, Peridot, KOJAK/EXPERT, TAU)

# *Performance Analysis Tool Integration*

- Complex systems pose challenging performance analysis problems that require robust methodologies and tools
- New performance problems will arise
  - Instrumentation and measurement
  - Data analysis and presentation
  - Diagnosis and tuning
- No one performance tool can address all concerns
- Look towards an integration of performance technologies
  - Evolution of technology frameworks to address problems
  - Integration support to link technologies to create performance problem solving environments

# *Performance Technology for Complex Parallel Systems*

## *REFERENCES*

cs.uoregon.edu
COMPUTER & INFORMATION SCIENCE • UNIVERSITY OF OREGON

John von Neumann - Institut für Computing
Zentralinstitut für Angewandte Mathematik

# *Papers and Reports*

- B. Mohr, A. Malony, S. Shende, and F. Wolf, "Towards a Performance Tool Interface for OpenMP: An Approach Based on Directive Rewriting," European Workshop on OpenMP (EWOMP 2001), Barcelona, September 2001.
- F. Wolf and B. Mohr, "Automatic Performance Analysis of SMP Cluster Applications," Technical Report IB-2001-05, John von Neumann - Institut für Computing, Forschungszentrum Jülich, ZAM, 2001.
- S. Shende and A. Malony, "Integration and Application of the TAU Performance System in Parallel Java Environments," Proceedings of the Joint ACM Java Grande - ISCOPE 2001 Conference, June 2001.
- S. Shende, A. D. Malony, and R. Ansell-Bell, "Instrumentation and Measurement Strategies for Flexible and Portable Empirical Performance Evaluation," Proceedings Tools and Techniques for Performance Evaluation Workshop, PDPTA'01, C.S.R.E.A., June 2001.

# *Papers and Reports*


- S. Shende, "The Role of Instrumentation and Mapping in Performance Measurement," Ph.D. Dissertation, University of Oregon, August 2001.
- T. Fahringer, M. Gerndt, B. Mohr, F. Wolf, G. Riley, and J. Träff, "Knowledge Specification for Automatic Performance Analysis – APART Technical Report, Revised Version," Technical Report IB-2001-08, John von Neumann - Institut für Computing, Forschungszentrum Jülich, ZAM, 2001.
- B. Mohr, A. Malony, S. Shende, and F. Wolf, "Design and Prototype of a Performance Tool Interface for OpenMP," 2nd Annual Los Alamos Computer Science Institute Symposium (LACSI 2001), October 2000.
- F. Wolf and B. Mohr, "Automatic Performance Analysis of MPI Applications Based on Event Traces," European Conference on Parallel Computing (Euro-Par 2000), München, August 2000.
- S. Shende and A. Malony, "Performance Tools for Parallel Java Environments," Proc. Second Workshop on Java for High Performance Computing, ICS 2000, May 2000.

# *Papers and Reports*


- A. Malony and S. Shende, "Performance Technology for Complex Parallel and Distributed Systems," Proc. Third Austrian-Hungarian Workshop on Distributed and Parallel Systems, DAPSYS 2000, "Distributed and Parallel Systems: From Concepts to Applications," (Eds. G. Kotsis and P. Kacsuk)Kluwer, Norwell, MA, August 2000, pp. 37-46.
- K. Lindlan, J. Cuny, A. Malony, S. Shende, B. Mohr, R. Rivenburgh, and C. Rasmussen. "A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates," Proceedings of SC2000: High Performance Networking and Computing Conference, Dallas, November 2000.
- A. Malony, "Tools for Parallel Computing: A Performance Evaluation Perspective," in J. Blazewicz et. al. (Editors), *Handbook on Parallel and Distributed Processing*, Springer Verlag, pp. 342-363, 2000.
- F. Wolf and B. Mohr, "EARL – Language Reference," Technical Report IB-2000-01, John von Neumann - Institut für Computing, Forschungszentrum Jülich, ZAM, 2000.

# *Papers and Reports*


- F. Wolf and B. Mohr, "Specifying Performance Properties Using Compound Runtime Events – APART Technical Report," Technical Report IB-2000-10, John von Neumann - Institut für Computing, Forschungszentrum Jülich, ZAM, 2000.
- S. Vajracharya, S. Karmesin, P. Beckman, J. Crotinger, A. Malony, S. Shende, R. Oldehoeft, and S. Smith, "SMARTS: Exploiting Temporal Locality and Parallelism through Vertical Execution," Proceedings of ACM International Conference on Supercomputing (ICS '99), June 1999.
- S. Shende, "Profiling and Tracing in Linux," Proceedings of the Extreme Linux Workshop #2, USENIX, Monterey CA, June 1999.
- S. Shende, A. D. Malony, J. Cuny, K. Lindlan, P. Beckman, and S. Karmesin, "Portable Profiling and Tracing for Parallel Scientific Applications using C++," Proceedings of ACM SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT '98), August 1998, pp. 134-145.

# *Papers and Reports*

- B. Mohr, A. Malony, and J. Cuny, "TAU," in G. Wilson (Ed.), *Parallel Programming Using C++*, MIT Press, 1996, p. 507-545.
- K. Shanmugam, A. Malony, and B. Mohr, "Speedy: An Integrated Performance Extrapolation Tool for pC++ Programs," Proceedings of the Joint Conference PERFORMANCE TOOLS '95 and MMB '95, Heidelberg, September 1995.
- A. Malony, B. Mohr, P. Beckman, and D. Gannon, "Program Analysis and Tuning Tools for a Parallel Object Oriented Language: An Experiment with the TAU System," Proceedings of the Workshop on Parallel Scientific Computing, Cape Cod, MA, October 1994.
- A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, and F. Bodin, "Performance Analysis of pC++: A Portable Data-Parallel Programming System for Scalable Parallel Computers," Proceedings of the 8th International Parallel Processing Symbosium (IPPS), Cancún, Mexico, April 1994, pp. 75-85.

# *Web References*

- TAU:
  *www.cs.uoregon.edu/research/paracomp/tau*
- PDT:
  *www.cs.uoregon.edu/research/paracomp/pdtoolkit*
- KOJAK (EXPERT, EARL, OPARI):
  *http://www.fz-juelich.de/zam/kojak/*
- APART:
  *http://www.fz-juelich.de/apart/*

# *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