

Parallel Performance Evaluation Tools for HPC Systems

Allen D. Malony

Dept of Computer and
Information Sciences
University of Oregon
malony@cs.uoregon.edu

Shirley Moore

Innovative Computing
Laboratory
University of Tennessee
shirley@eecs.utk.edu

Rick Kuftrin

National Center for
Supercomputing Applications
University of Illinois
rkuftrin@illinois.edu

Markus Geimer

Jülich Supercomputing Centre
m.geimer@fz-juelich.de

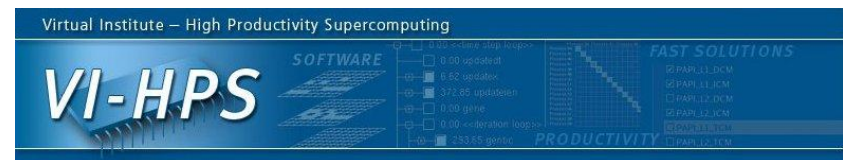
Andreas Knüpfer

Technical University Dresden
andreas.knuepfer@tu-dresden.de

International Conference on Computational Science (ICCS 2009)

Baton Rouge, Louisiana, USA

May 24, 2009



Tutorial Agenda

Time	Topic	Speaker
8:00 – 8:45	Introduction to Performance Engineering	Malony
8:45 – 9:00	POINT	Malony
9:00 – 9:30	PAPI – Performance API	Moore
9:30 – 10:00	PerfSuite	Kufrin
10:00 – 10:30	BREAK	
10:30 – 10:45	POINT/VI-HPS LiveDVD	Malony
10:45 – 12:00	TAU	Malony
12:00 – 1:30	LUNCH	
1:30 – 1:45	VI-HPS	Geimer
1:45 – 3:00	Scalasca	Geimer
3:00 – 3:30	BREAK	
3:30 – 4:45	Vampir/VampirTrace	Knüpfer
4:45 – 5:30	LiveDVD Hands-on	All
5:30	TUTORIAL ENDS	

POINT

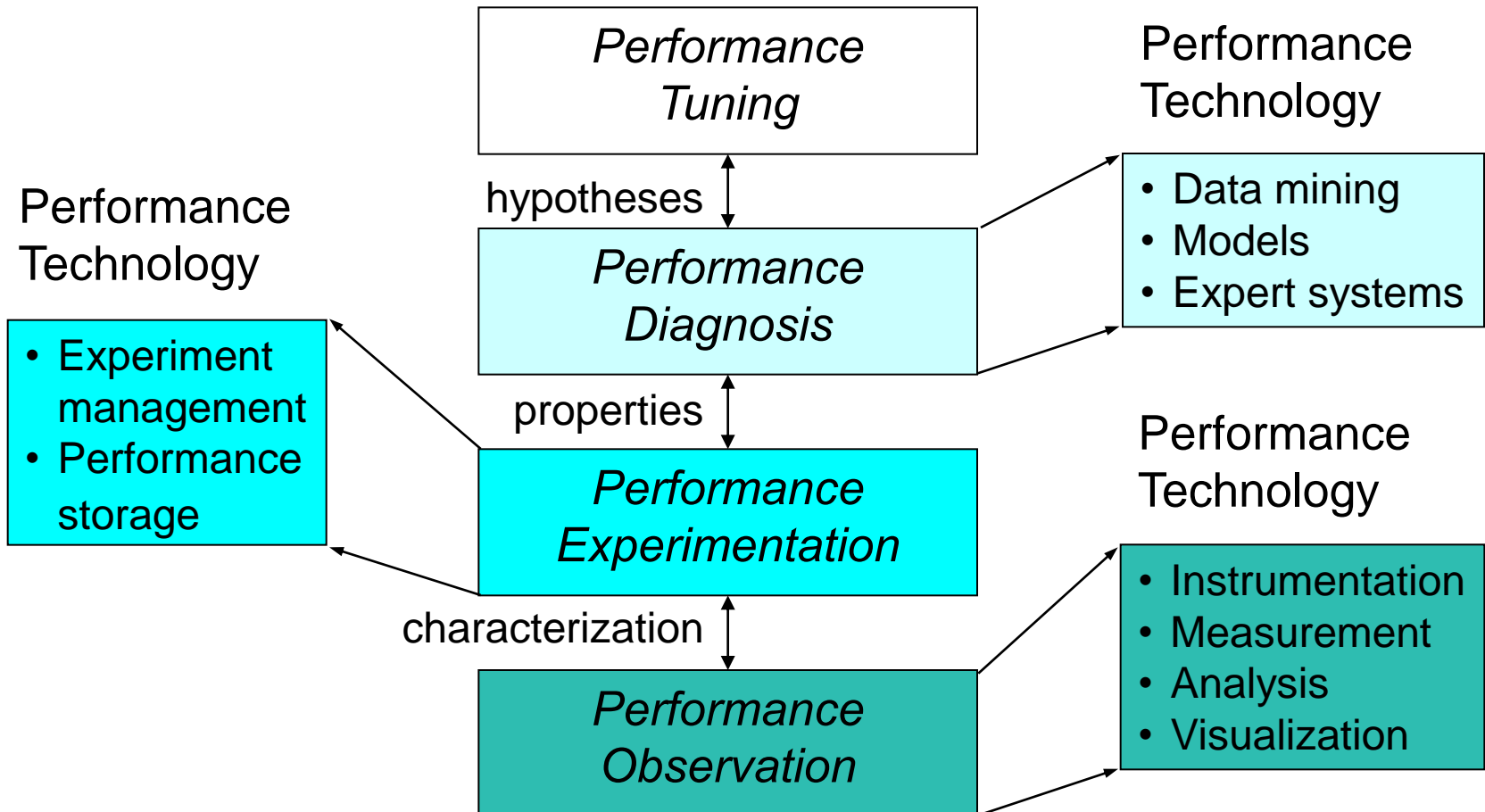


INTRODUCTION TO PERFORMANCE ENGINEERING

Allen D. Malony
Performance Research Laboratory
University of Oregon

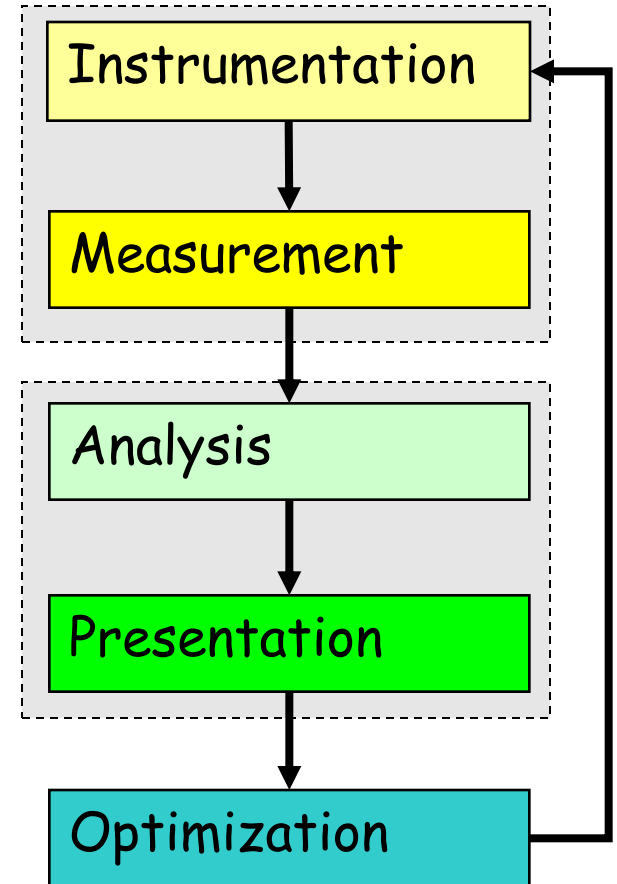
Performance Engineering

- Optimization process
- Effective use of performance technology



Performance Optimization Cycle

- Expose factors
- Collect performance data
- Calculate metrics
- Analyze results
- Visualize results
- Identify problems
- Tune performance



Parallel Performance Properties

- Parallel code performance is influenced by both sequential and parallel factors?
- Sequential factors
 - Computation and memory use
 - Input / output
- Parallel factors
 - Thread / process interactions
 - Communication and synchronization

Performance Observation

- Understanding performance requires observation of performance properties
- Performance tools and methodologies are primarily distinguished by what observations are made and how
 - What aspects of performance factors are seen
 - What performance data is obtained
- Tools and methods cover broad range

Metrics and Measurement

- Observability depends on measurement
- A metric represents a type of measured data
 - Count, time, hardware counters
- A measurement records performance data
 - Associates with program execution aspects
- Derived metrics are computed
 - Rates (e.g., flops)
- Metrics / measurements decided by need

Execution Time

- Wall-clock time
 - Based on realtime clock
- Virtual process time
 - Time when process is executing
 - ser time and system time
 - Does not include time when process is stalled
- Parallel execution time
 - Runs whenever any parallel part is executing
 - Global time basis

Direct Performance Observation

- Execution *actions* exposed as *events*
 - In general, actions reflect some execution state
 - presence at a code location or change in data
 - occurrence in parallelism context (thread of execution)
 - Events encode actions for observation
- Observation is *direct*
 - Direct instrumentation of program code (probes)
 - Instrumentation invokes performance measurement
 - Event measurement = performance data + context
- Performance experiment
 - Actual events + performance measurements

Indirect Performance Observation

- Program code instrumentation is not used
- Performance is observed indirectly
 - Execution is interrupted
 - can be triggered by different events
 - Execution state is queried (sampled)
 - different performance data measured
 - *Event-based sampling* (EBS)
- Performance attribution is inferred
 - Determined by execution context (state)
 - Observation resolution determined by interrupt period
 - Performance data associated with context for period

Direct Observation: Events

- Event types
 - Interval events (begin/end events)
 - measures performance between begin and end
 - metrics monotonically increase
 - Atomic events
 - used to capture performance data state
- Code events
 - Routines, classes, templates
 - Statement-level blocks, loops
- User-defined events
 - Specified by the user
- Abstract mapping events

Direct Observation: Instrumentation

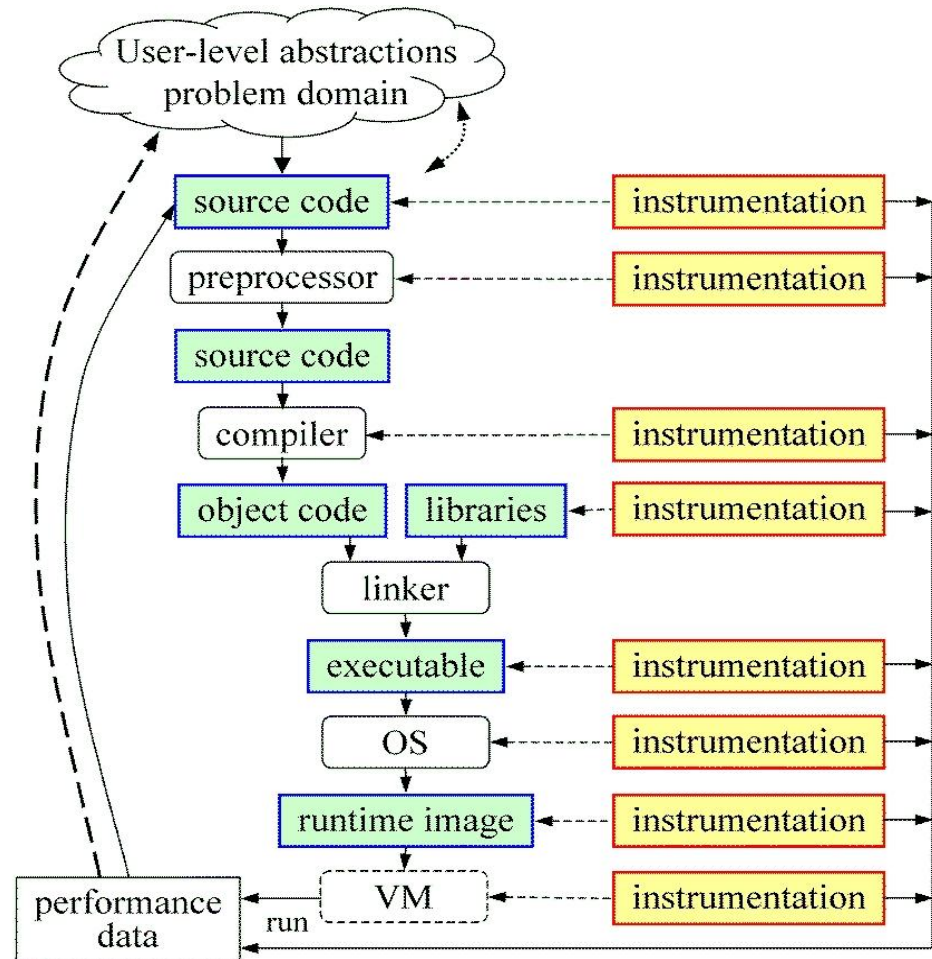
- Events defined by instrumentation access
- Instrumentation levels
 - Source code
 - Object code
 - Runtime system
 - Library code
 - Executable code
 - Operating system
- Different levels provide different information
- Different tools needed for each level
- Levels can have different granularity

Direct Observation: Techniques

- Static instrumentation
 - Program instrumented prior to execution
- Dynamic instrumentation
 - Program instrumented at runtime
- Manual and automatic mechanisms
- Tool required for automatic support
 - Source time: preprocessor, translator, compiler
 - Link time: wrapper library, preload
 - Execution time: binary rewrite, dynamic
- Advantages / disadvantages

Direct Observation: Mapping

- Associate performance data with high-level semantic abstractions
- Abstract events at user-level provide semantic context



Indirect Observation: Events/Triggers

- Events are actions external to program code
 - Timer countdown, HW counter overflow, ...
 - Consequence of program execution
 - Event frequency determined by:
 - Type, setup, number enabled (exposed)
- Triggers used to invoke measurement tool
 - Traps when events occur (interrupt)
 - Associated with events
 - May add differentiation to events

Indirect Observation: Context

- When events trigger, execution context determined at time of trap (interrupt)
 - Access to PC from interrupt frame
 - Access to information about process/thread
 - Possible access to call stack
 - requires call stack unwinder
- Assumption is that the context was the same during the preceding period
 - Between successive triggers
 - Statistical approximation valid for long running programs

Direct / Indirect Comparison

- Direct performance observation
 - ☺ Measures performance data exactly
 - ☺ Links performance data with application events
 - ☹ Requires instrumentation of code
 - ☹ Measurement overhead can cause execution intrusion and possibly performance perturbation
- Indirect performance observation
 - ☺ Argued to have less overhead and intrusion
 - ☺ Can observe finer granularity
 - ☺ No code modification required (may need symbols)
 - ☹ Inexact measurement and attribution

Measurement Techniques

- When is measurement triggered?
 - External agent (indirect, asynchronous)
 - interrupts, hardware counter overflow, ...
 - Internal agent (direct, synchronous)
 - through code modification
- How are measurements made?
 - Profiling
 - summarizes performance data during execution
 - per process / thread and organized with respect to context
 - Tracing
 - trace record with performance data and timestamp
 - per process / thread

Measured Performance

- Counts
- Durations
- Communication costs
- Synchronization costs
- Memory use
- Hardware counts
- System calls

Critical issues

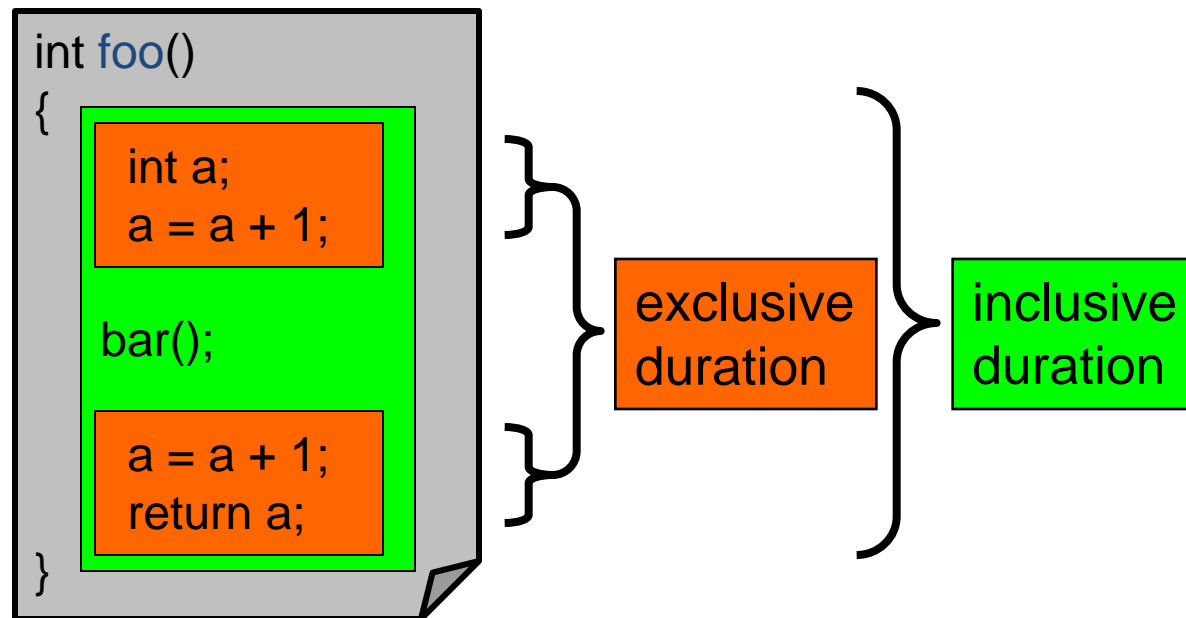
- Accuracy
 - Timing and counting accuracy depends on resolution
 - Any performance measurement generates overhead
 - Execution on performance measurement code
 - Measurement overhead can lead to intrusion
 - Intrusion can cause perturbation
 - alters program behavior
- Granularity
 - How many measurements are made
 - How much overhead per measurement
- Tradeoff (general wisdom)
 - Accuracy is inversely correlated with granularity

Profiling

- Recording of aggregated information
 - Counts, time, ...
- ... about program and system entities
 - Functions, loops, basic blocks, ...
 - Processes, threads
- Methods
 - Event-based sampling (indirect, statistical)
 - Direct measurement (deterministic)

Inclusive and Exclusive Profiles

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



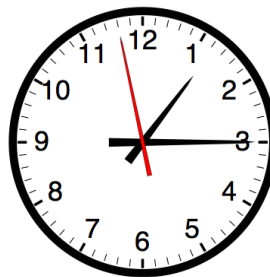
Flat and Callpath Profiles

- Static call graph
 - Shows all parent-child calling relationships in a program
- Dynamic call graph
 - Reflects actual execution time calling relationships
- Flat profile
 - Performance metrics for when event is active
 - Exclusive and inclusive
- Callpath profile
 - Performance metrics for calling path (event chain)
 - Differentiate performance with respect to program execution state
 - Exclusive and inclusive

Tracing Measurement

Process A:

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



1	master
2	worker
3	...

Process B:

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

MONITOR

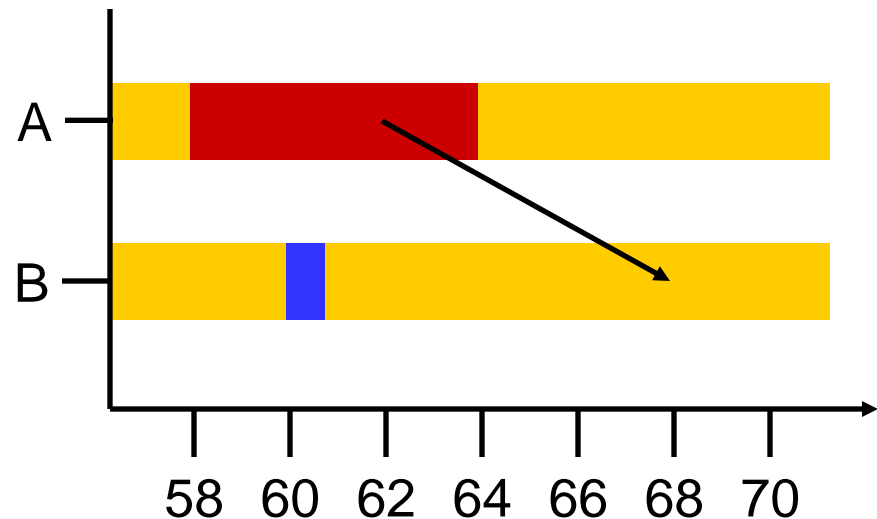
...			
58	A	ENTER	1
60	B	ENTER	2
62	A	SEND	B
64	A	EXIT	1
68	B	RECV	A
69	B	EXIT	2
...			

Tracing Analysis and Visualization

1	master
2	worker
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
...			



Trace Formats

- Different tools produce different formats
 - Differ by event types supported
 - Differ by ASCII and binary representations
 - Vampir Trace Format (VTF)
 - KOJAK (EPILOG)
 - Jumpshot (SLOG-2)
 - Paraver
- Open Trace Format (OTF)
 - Supports interoperation between tracing tools

Profiling / Tracing Comparison

- Profiling
 - ☺ Finite, bounded performance data size
 - ☺ Applicable to both direct and indirect methods
 - ☹ Loses time dimension (not entirely)
 - ☹ Lacks ability to fully describe process interaction
- Tracing
 - ☺ Temporal and spatial dimension to performance data
 - ☺ Capture parallel dynamics and process interaction
 - ☹ Some inconsistencies with indirect methods
 - ☹ Unbounded performance data size (large)
 - ☹ Complex event buffering and clock synchronization

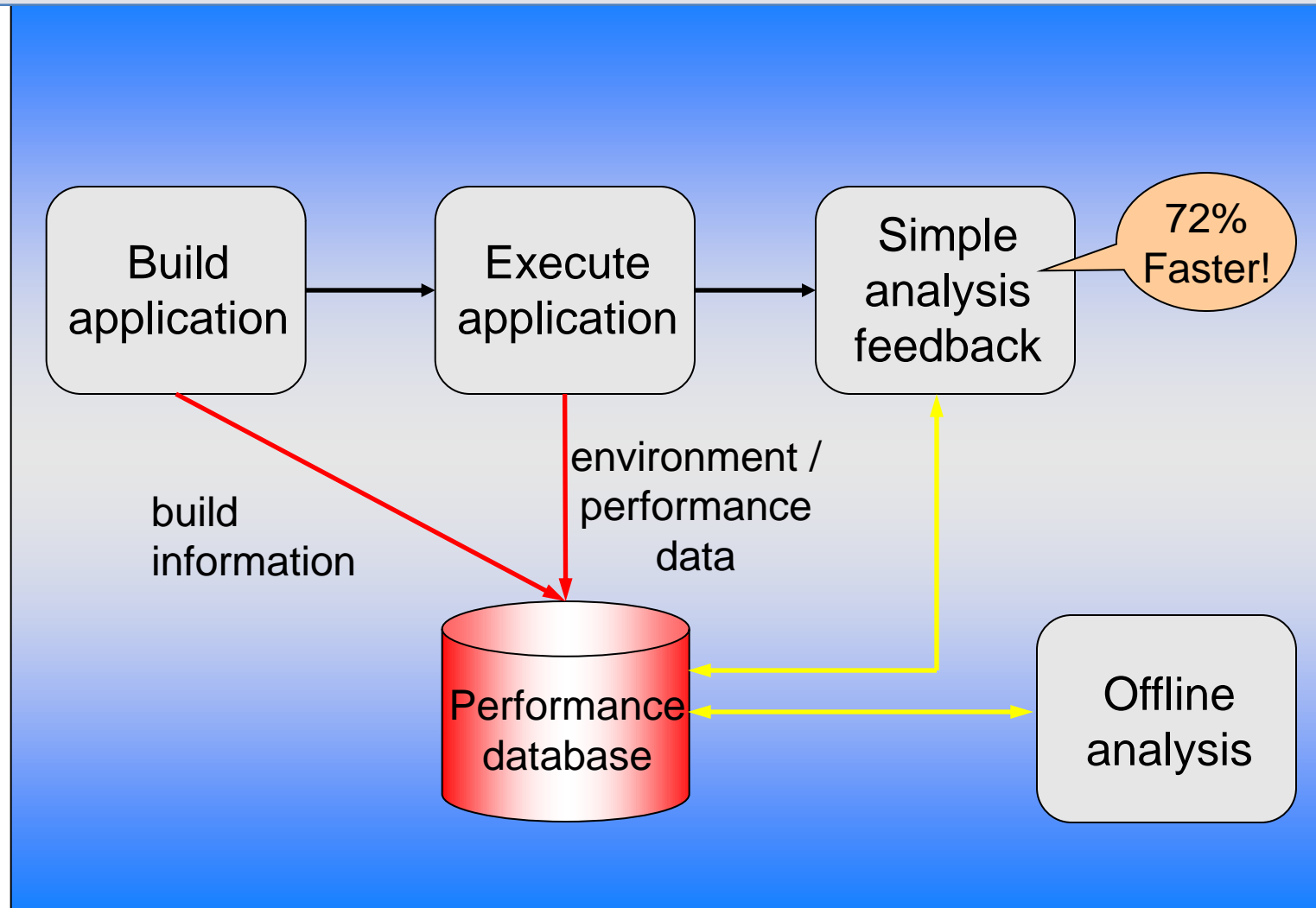
Performance Problem Solving Goals

- Answer questions at multiple levels of interest
 - High-level performance data spanning dimensions
 - machine, applications, code revisions, data sets
 - examine broad performance trends
 - Data from low-level measurements
 - use to predict application performance
- Discover general correlations
 - performance and features of external environment
 - Identify primary performance factors
- Benchmarking analysis for application prediction
- Workload analysis for machine assessment

Performance Analysis Questions

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

Automatic Performance Analysis



Performance Data Management

- Performance diagnosis and optimization involves multiple performance experiments
- Support for common performance data management tasks augments tool use
 - Performance experiment data and metadata storage
 - Performance database and query
- What type of performance data should be stored?
 - Parallel profiles or parallel traces
 - Storage size will dictate
 - Experiment metadata helps in meta analysis tasks
- Serves tool integration objectives

Metadata Collection

- Integration of metadata with each parallel profile
 - Separate information from performance data
- Three ways to incorporate metadata
 - Measured hardware/system information
 - CPU speed, memory in GB, MPI node IDs, ...
 - Application instrumentation (application-specific)
 - Application parameters, input data, domain decomposition
 - Capture arbitrary name/value pair and save with experiment
 - Data management tools can read additional metadata
 - Compiler flags, submission scripts, input files, ...
 - Before or after execution
- Enhances analysis capabilities

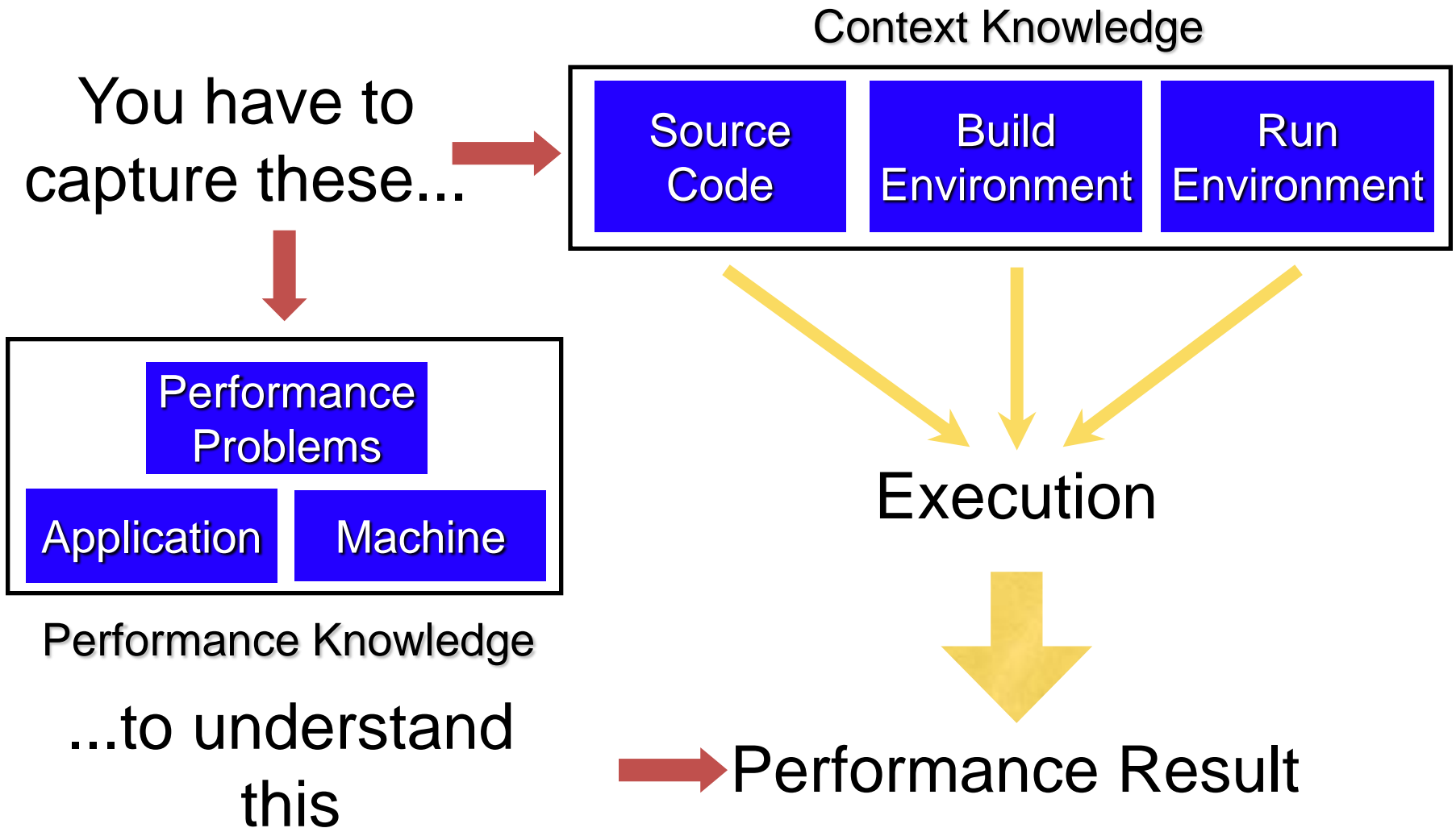
Performance Data Mining

- Conduct parallel performance analysis in a systematic, collaborative and reusable manner
 - Manage performance complexity and automate process
 - Discover performance relationship and properties
 - Multi-experiment performance analysis
- Data mining applied to parallel performance data
 - Comparative, clustering, correlation, characterization, ...
 - Large-scale performance data reduction
- Implement extensible analysis framework
 - Abstraction / automation of data mining operations
 - Interface to existing analysis and data mining tools

How to explain performance?

- Should not just redescribe performance results
- Should explain performance phenomena
 - What are the causes for performance observed?
 - What are the factors and how do they interrelate?
 - Performance analytics, forensics, and decision support
- Add *knowledge* to do more intelligent things
 - Automated analysis needs good informed feedback
 - Performance model generation requires interpretation
- Performance knowledge discovery framework
 - Integrating meta-information
 - Knowledge-based performance problem solving

Metadata and Knowledge Role



Performance Optimization Process

- Performance characterization
 - Identify major performance contributors
 - Identify sources of performance inefficiency
 - Utilize timing and hardware measures
- Performance diagnosis (Performance Debugging)
 - Look for conditions of performance problems
 - Determine if conditions are met and their severity
 - What and where are the performance bottlenecks
- Performance tuning
 - Focus on dominant performance contributors
 - Eliminate main performance bottlenecks

POINT Project

- “High-Productivity Performance Engineering (Tools, Methods, Training) for NSF HPC Applications”
 - NSF SDCl, Software Improvement and Support
 - University of Oregon, University of Tennessee, National Center for Supercomputing Applications, Pittsburgh Supercomputing Center
- POINT project
 - Petascale Productivity from Open, Integrated Tools
 - <http://www.nic.uoregon.edu/point>

Motivation

- Promise of HPC through scalable scientific and engineering applications
- Performance optimization through effective performance engineering methods
 - Performance analysis / tuning “best practices”
- Productive petascale HPC will require
 - Robust parallel performance tools
 - Training good performance problem solvers

Objectives

- Robust parallel performance environment
 - Uniformly available across NSF HPC platforms
- Promote performance engineering
 - Training in performance tools and methods
 - Leverage NSF TeraGrid EOT
- Work with petascale applications teams
- Community building

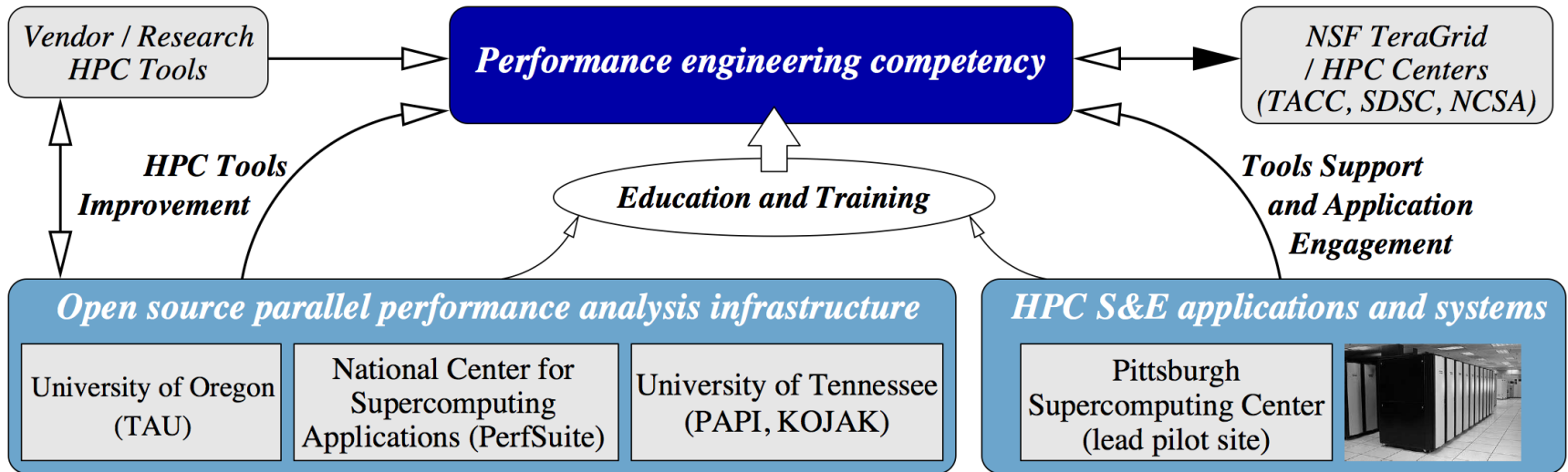
Challenges

- Consistent performance tool environment
 - Tool integration, interoperation, and scalability
 - Uniform deployment across NSF HPC platforms
- Useful evaluation metrics and process
 - Make part of code development routine
 - Recording performance engineering history
- Develop performance engineering culture
 - Proceed beyond “hand holding” engagements

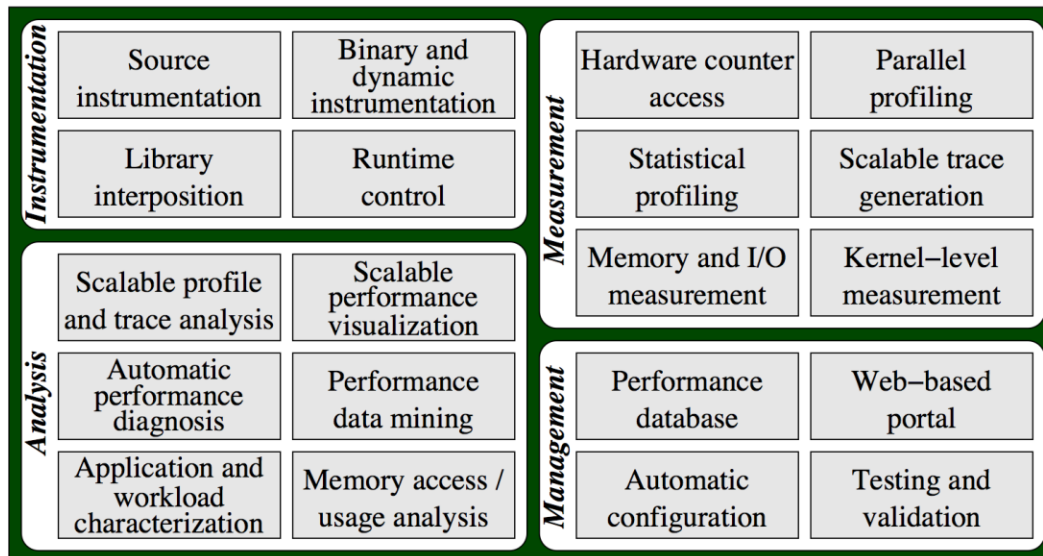
Performance Engineering Levels

- Target different performance tool users
 - Different levels of expertise
 - Different performance problem solving needs
- Level 0 (entry)
 - Simpler tool use, limited performance data
- Level 1 (intermediate)
 - More tool sophistication, increased information
- Level 2 (advanced)
 - Access to powerful performance techniques

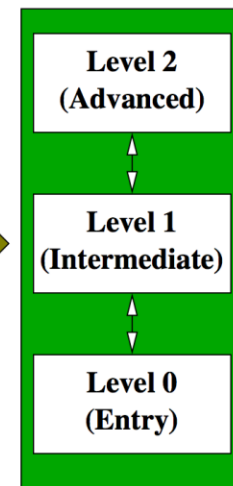
POINT Project Organization



Performance Technology Expertise



Performance Engineering Process



Testbed Apps
 ENZO
 NAMD
 NEMO3D

Parallel Performance Technology

- PAPI
 - University of Tennessee, Knoxville
- PerfSuite
 - National Center for Supercomputing Applications
- TAU Performance System
 - University of Oregon
- Kojak / Scalasca
 - Research Centre Juelich
- Vampir and VampirTrace
 - T.U. Dresden



Parallel Engineering Training

- User engagement
- User support in TeraGrid
- Training workshops
- Quantify tool impact
- POINT lead pilot site
 - Pittsburgh Supercomputing Center
 - NSF TeraGrid site



Testbed Applications

- ENZO
 - Adaptive mesh refinement (AMR), grid-based hybrid code (hydro+Nbody) designed to do simulations of cosmological structure formation
- NAMD
 - Mature community parallel molecular dynamics application deployed for research in large-scale biomolecular systems
- NEMO3D
 - Quantum mechanical based simulation tool created to provide quantitative predictions for nanometer-scale semiconductor devices

PAPI

Shirley Moore, Dan Terpstra
Innovative Computing Lab
University of Tennessee, Knoxville

Hardware Counters

Hardware performance counters available on most modern microprocessors can provide insight into:

1. Whole program timing
2. Cache behaviors
3. Branch behaviors
4. Memory and resource access patterns
5. Pipeline stalls
6. Floating point efficiency
7. Instructions per cycle

Hardware counter information can be obtained with:

1. Subroutine or basic block resolution
2. Process or thread attribution

What's PAPI?



- Middleware to provide a consistent programming interface for the performance counter hardware found in most major micro-processors.
- Countable events are defined in two ways:
 - Platform-neutral *preset* events
 - Platform-dependent native events
- Presets can be **derived** from multiple *native events*
- All events are referenced by name and collected in EventSets for sampling
- Events can be **multiplexed** if counters are limited
- Statistical sampling implemented by:
 - Hardware overflow if supported by the platform
 - Software overflow with timer driven sampling

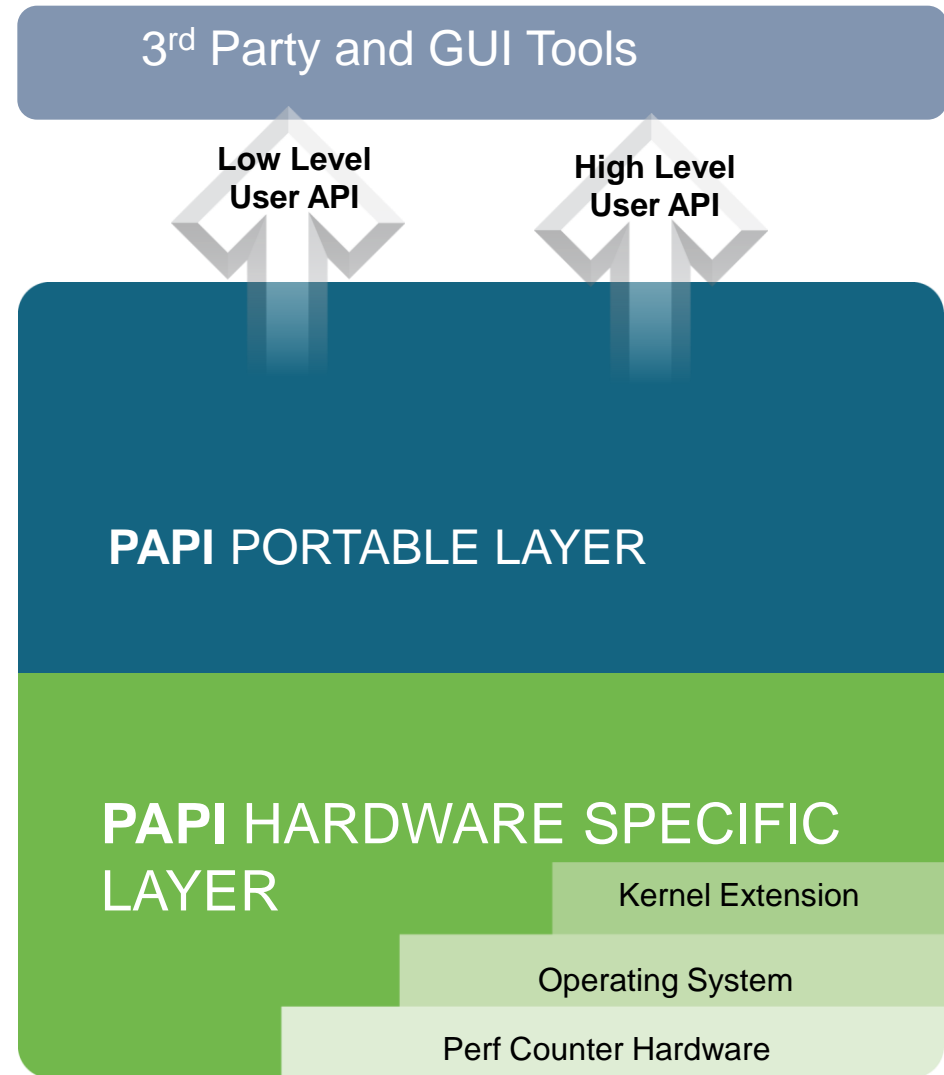
Where's PAPI

- PAPI runs on most modern processors and operating systems of interest to HPC:
 - IBM POWER series / AIX
 - POWER4,5,6 PowerPC / Linux
 - Blue Gene/L/P
 - Intel Pentium series, Core2, Core i7 / Linux
 - Intel Itanium 1, 2, Montecito, Montvale
 - AMD Athlon, Opteron multicore / Linux
 - Cray X1, X2, XT3/4/5 Catamount / CLE
 - Altix, Sparc, SiCortex ...

PAPI Counter Interfaces

PAPI provides 3 interfaces to the underlying counter hardware:

1. **A Low Level API manages hardware events in user defined groups called EventSets, and provides access to advanced features.**
2. **A High Level API provides the ability to start, stop and read the counters for a specified list of events.**
3. **Graphical and end-user tools provide facile data collection and visualization.**



PAPI Preset Events

◆ Preset Events

- Standard set of over 100 events for application performance tuning
- No standardization of the exact definition
- Mapped to either single or linear combinations of native events on each platform
- Use `papi_avail` utility to see what preset events are available on a given platform

PAPI_L2_DCH: Level 1 data cache hits
PAPI_L2_DCW: Level 1 data cache writes
PAPI_L2_DCM: Level 1 data cache misses

PAPI_L2_ICH: Level 1 instruction cache hits
PAPI_L2_ICM: Level 1 instruction cache misses

PAPI_L2_TCH: Level 1 total cache hits
PAPI_L2_TCA: Level 1 total cache accesses
PAPI_L2_TCR: Level 1 total cache reads
PAPI_L2_TCW: Level 1 total cache writes
PAPI_L2_TCM: Level 1 cache misses

PAPI_L2_LDM: Level 1 load misses
PAPI_L2_STM: Level 1 store misses

Level 3 Cache

PAPI_L3_DCH: Level 1 data cache hits
PAPI_L3_DCA: Level 1 data cache accesses
PAPI_L3_DCR: Level 1 data cache reads
PAPI_L3_DCW: Level 1 data cache writes
PAPI_L3_DCM: Level 1 data cache misses

PAPI_L3_ICH: Level 1 instruction cache hits
PAPI_L3_ICA: Level 1 instruction cache accesses
PAPI_L3_ICR: Level 1 instruction cache reads
PAPI_L3_ICW: Level 1 instruction cache writes
PAPI_L3_ICM: Level 1 instruction cache misses

PAPI_L3_TCH: Level 1 total cache hits
PAPI_L3_TCA: Level 1 total cache accesses
PAPI_L3_TCR: Level 1 total cache reads
PAPI_L3_TCW: Level 1 total cache writes
PAPI_L3_TCM: Level 1 cache misses

PAPI_L3_LDM: Level 1 load misses
PAPI_L3_STM: Level 1 store misses

PAPI_CA_SNP: Requests for a snoop
PAPI_CA_SHR: Requests for exclusive access to shared cache line

PAPI Native Events

- Native Events
 - Any event countable by the CPU
 - Same interface as for preset events
 - Use *papi_native_avail* utility to see all available native events
- Use *papi_event_chooser* utility to select a compatible set of events

PRESET,
PAPI_L2_DCA,
DERIVED_ADD,
L2_LD:SELF:ANY:MESI,
L2_ST:SELF:MESI

```
{ .pme_name = "L2_ST",  
  .pme_code = 0x2a,  
  .pme_flags = PFMLIB_CORE_CSPEC,  
  .pme_desc = "L2 store requests",  
  .pme_umasks = {  
    { .pme_uname = "MESI",  
      .pme_udesc = "Any cacheline access",  
      .pme_ucose = 0xf  
    },  
    { .pme_uname = "I_STATE",  
      .pme_udesc = "Invalid cacheline",  
      .pme_ucose = 0x1  
    },  
    { .pme_uname = "S_STATE",  
      .pme_udesc = "Shared cacheline",  
      .pme_ucose = 0x2  
    },  
    { .pme_uname = "E_STATE",  
      .pme_udesc = "Exclusive cacheline",  
      .pme_ucose = 0x4  
    },  
    { .pme_uname = "M_STATE",  
      .pme_udesc = "Modified cacheline",  
      .pme_ucose = 0x8  
    }  
  },  
  { .pme_uname = "SELF",  
    .pme_udesc = "This core",  
    .pme_ucose = 0x40  
  },  
  { .pme_uname = "BOTH_CORES",  
    .pme_udesc = "Both cores",  
    .pme_ucose = 0xc0  
  }  
},  
.pme_numasks = 7  
},
```

PAPI High-level Interface

- Meant for application programmers wanting coarse-grained measurements
- Calls the lower level API
- Allows only PAPI preset events
- Easier to use and less setup (less additional code) than low-level
- Supports 8 calls in C or Fortran:

`PAPI_start_counters`

`PAPI_stop_counters`

`PAPI_read_counters`

`PAPI_accum_counters`

`PAPI_num_counters`

`PAPI_flips`

`PAPI_ipc`

`PAPI_flops`

PAPI High-level Example

```
#include "papi.h"
#define NUM_EVENTS 2
long_long values[NUM_EVENTS];
unsigned int
    Events[NUM_EVENTS]={PAPI_TOT_INS,PAPI_TOT_CYC};

/* Start the counters */
PAPI_start_counters((int*)Events,NUM_EVENTS);

/* What we are monitoring... */
do_work();

/* Stop counters and store results in values */
retval = PAPI_stop_counters(values,NUM_EVENTS);
```

Low-level Interface

- Increased efficiency and functionality over the high level PAPI interface
- Obtain information about the executable, the hardware, and the memory environment
- Multiplexing
- Callbacks on counter overflow
- Profiling
- About 60 functions

PAPI Low-level Example

```
#include "papi.h"
#define NUM_EVENTS 2
int Events[NUM_EVENTS]={PAPI_FP_INS,PAPI_TOT_CYC};
int EventSet;
long_long values[NUM_EVENTS];
/* Initialize the Library */
retval = PAPI_library_init(PAPI_VER_CURRENT);
/* Allocate space for the new eventset and do setup */
retval = PAPI_create_eventset(&EventSet);
/* Add Flops and total cycles to the eventset */
retval = PAPI_add_events(EventSet,Events,NUM_EVENTS);
/* Start the counters */
retval = PAPI_start(EventSet);

do_work(); /* What we want to monitor*/

/*Stop counters and store results in values */
retval = PAPI_stop(EventSet,values);
```

PAPI & Multicore

- Multicore is the (near term) future of petascale computing
- Minimizing resource contention will be key
 - Memory bandwidth
 - Cache sharing
 - Bus and other resource contention

The Multicore Dilemma

- Multicore is the (near term) future of Petascale computing
- Minimizing Resource contention is key
 - Memory bandwidth
 - Cache sharing & collisions
 - Bus and other resource contention
- Current tools don't support first-person counting of shared events
- Current architectures don't encourage first-person counting of shared events

Multicore counter support

- Intel Core2 Duos:
 - SELF/ANY
 - L2 shared cache, bus, snoop
 - 39 events/~140 are core qualified
- AMD Barcelona
 - 4 L3 shared cache events:
 - READ_REQUEST_TO_L3_CACHE
 - L3_CACHE_MISSES
 - L3_FILLS_CAUSED_BY_L2_EVICTIONS
 - L3_EVICTIONS
 - First 3 are qualified per core:
 - CORE0, CORE1, CORE2, CORE3
 - Only 1 core can count these events at a time

Multicore counter support (cont.)

- Intel i7 (Nehalem)
 - Nehalem support available in development version of PAPI
 - The Nehalem has 7 counters per core; 3 fixed and 4 general purpose.
 - Another 8 shared counters are provided on-chip to support "Uncore" events. These counters are not currently supported by PAPI.
 - 117 native events are available to PAPI users, along with 28 PRESET events.

Current “State of the Art”

- Counter support for shared resources is broken
 - Every vendor has a different approach
 - Often 3rd person, not 1st person
 - Counts often polluted by other cores
 - No exclusive reservation of shared counter resources
 - No migration of events with tasks
- PAPI research is underway to address this

Extending PAPI beyond the CPU

- PAPI has historically targeted on on-processor performance counters
- Several categories of off-processor counters exist
 - network interfaces: Myrinet, Infiniband, GigE
 - memory interfaces: Cray X1, SeaStar, Gemini
 - thermal and power interfaces: ACPI, Im-sensors
 - accelerators?
- CHALLENGE:
 - Extend the PAPI interface to address multiple counter domains
 - Preserve the PAPI calling semantics, ease of use, and platform independence for existing applications

Motivation

- Performance counters also exist in off-CPU resources
- All information is valuable for performance optimization
- Increasing cpu counts & power demands place greater importance on:
 - Thermal health and management
 - Power consumption
- Multicore systems require careful resource balancing
- Higher processor & core counts make communications metrics more critical:
 - Bandwidth
 - Latency
 - Dropped packets
 - Bytes transferred

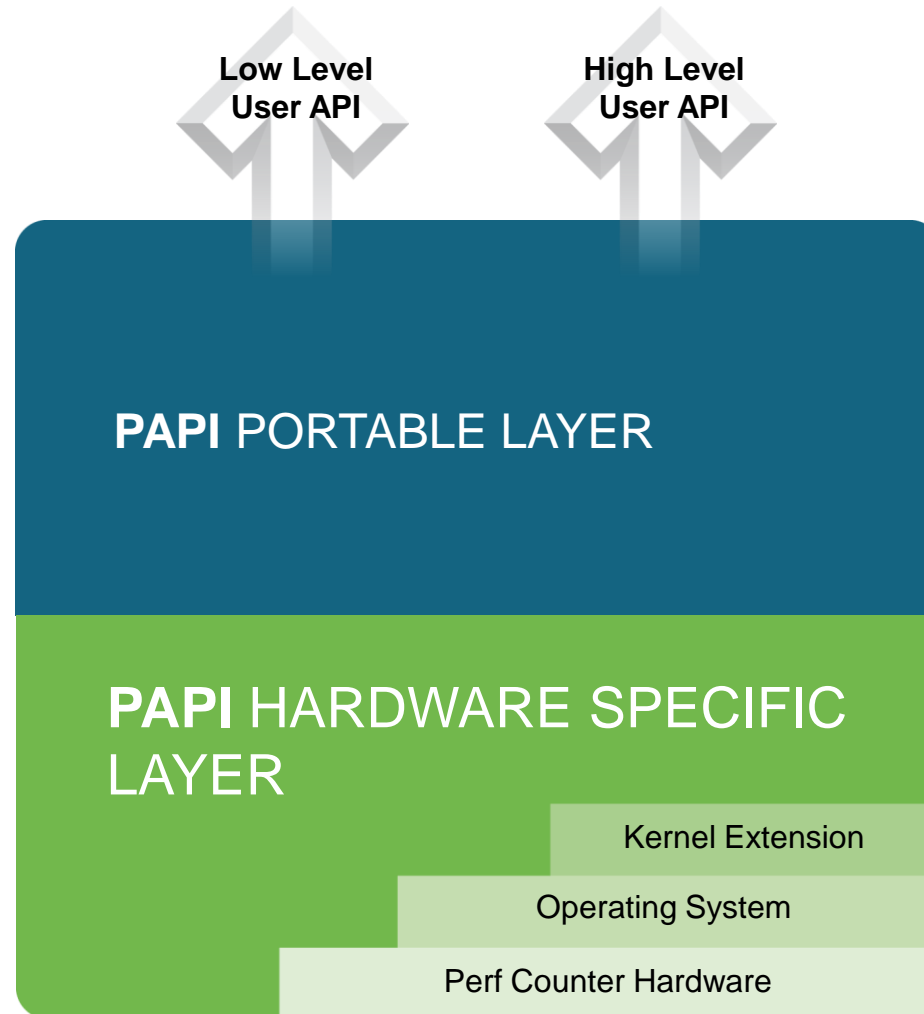
Limitations

- Interfaces are often obscure, unexposed or non-standard
- Performance data (accelerators) can be vastly different than cpus
- Measurements are usually system-wide and asynchronous
 - May not matter on dedicated single-task OS's like Cray Catamount or CLE and Blue Gene CNK
 - But matters more for Multicore
- Often very different time scales

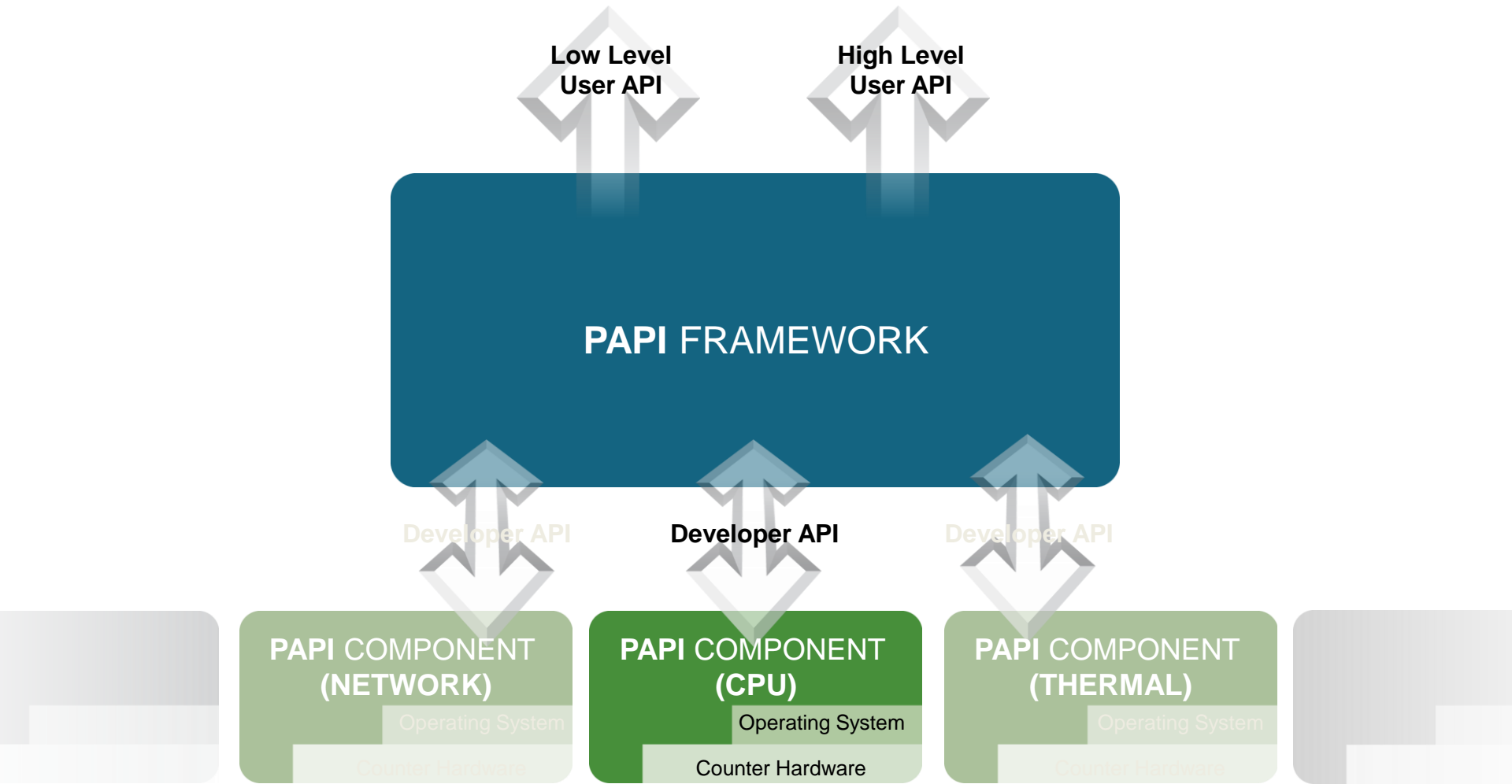
Component PAPI Goals

- Support simultaneous access to on- and off-processor counters
- Isolate hardware dependent code in separable 'component' modules
- Extend platform independent code to support multiple simultaneous components
- Add or modify API calls to support access to any of several components
- Modify build environment for easy selection and configuration of multiple available components

Monolithic 'PAPI Classic'

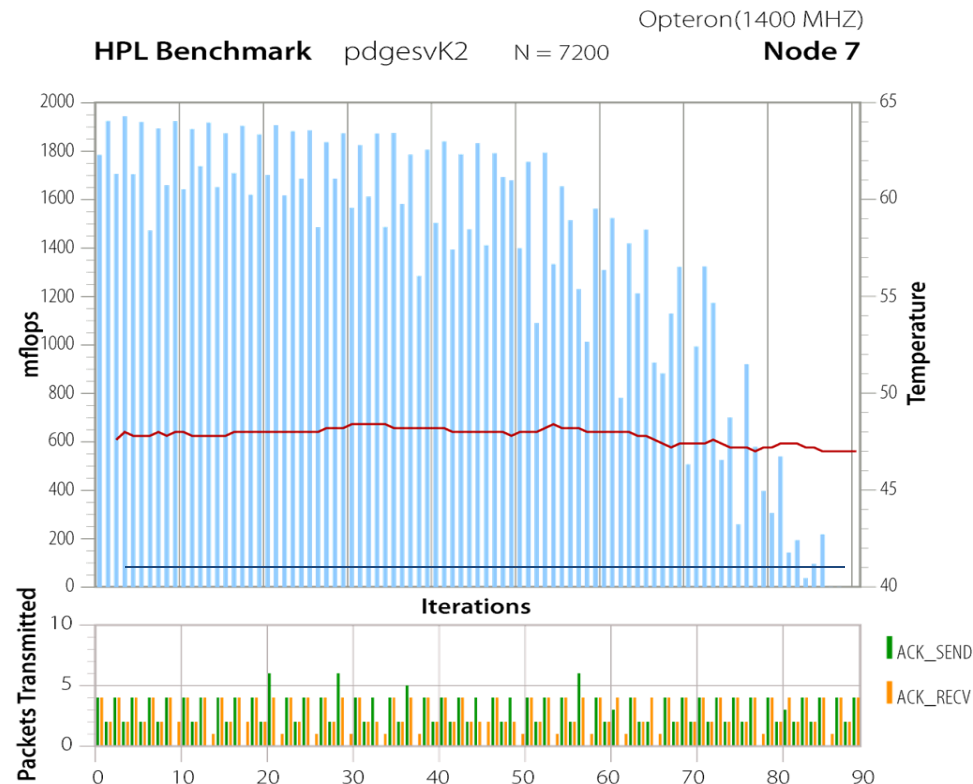


Component PAPI



Multi Component Measurements

- ◆ HPC HPL benchmark on Opteron with 3 performance metrics:
 - FLOPS; Temperature; Network Sends/Receives
 - Temperature is from an on-chip thermal diode



Myrinet MX Counters

LANAI_UPTIME	ACK_NACK_FRAMES_IN_PIPE	REPLY_SEND	ROUTE_DISPERSION
COUNTERS_UPTIME	NACK_BAD_ENDPT	REPLY_RECV	OUT_OF_SEND_HANDLES
BAD_CRC8	NACK_ENDPT_CLOSED	QUERY_UNKNOWN	OUT_OF_PULL_HANDLES
BAD_CRC32	NACK_BAD_SESSION	DATA_SEND_NULL	OUT_OF_PUSH_HANDLES
UNSTRIPPED_ROUTE	NACK_BAD_RDMAWIN	DATA_SEND_SMALL	MEDIUM_CONT_RACE
PKT_DESC_INVALID	NACK_EVENTQ_FULL	DATA_SEND_MEDIUM	CMD_TYPE_UNKNOWN
RCV_PKT_ERRORS	SEND_BAD_RDMAWIN	DATA_SEND_RNDV	UREQ_TYPE_UNKNOWN
PKT_MISROUTED	CONNECT_TIMEOUT	DATA_SEND_PULL	INTERRUPTS_OVERRUN
DATA_SRC_UNKNOWN	CONNECT_SRC_UNKNOWN	DATA_RECV_NULL	WAITING_FOR_INTERRUPT_DMA
DATA_BAD_ENDPT	QUERY_BAD_MAGIC	DATA_RECV_SMALL_INLINE	WAITING_FOR_INTERRUPT_ACK
DATA_ENDPT_CLOSED	QUERY_TIMED_OUT	DATA_RECV_SMALL_COPY	WAITING_FOR_INTERRUPT_TIMER
DATA_BAD_SESSION	QUERY_SRC_UNKNOWN	DATA_RECV_MEDIUM	
PUSH_BAD_WINDOW	RAW SENDS	DATA_RECV_RNDV	SLABS_RECYCLING
PUSH_DUPLICATE	RAW RECEIVES	DATA_RECV_PULL	SLABS_PRESSURE
PUSH_OBSOLETE	RAW_OVERSIZED_PACKETS	ETHER_SEND_UNICAST_CNT	SLABS_STARVATION
PUSH_RACE_DRIVER	RAW_RECV_OVERRUN	ETHER_SEND_MULTICAST_CNT	OUT_OF_RDMA_HANDLES
PUSH_BAD_SEND_HANDLE	RAW_DISABLED		EVENTQ_FULL
_MAGIC	CONNECT_SEND	ETHER_RECV_SMALL_CNT	BUFFER_DROP
PUSH_BAD_SRC_MAGIC	CONNECT_RECV	ETHER_RECV_BIG_CNT	MEMORY_DROP
PULL_OBSOLETE	ACK_SEND	ETHER_OVERRUN	HARDWARE_FLOW_CONTROL
PULL_NOTIFY_OBSOLETE	ACK_RECV	ETHER_OVERSIZED	SIMULATED_PACKETS_LOST
PULL_RACE_DRIVER	PUSH_SEND	DATA_RECV_NO_CREDITS	LOGGING_FRAMES_DUMPED
ACK_BAD_TYPE	PUSH_RECV	PACKETS_RESENT	WAKE_INTERRUPTS
ACK_BAD_MAGIC	QUERY_SEND	PACKETS_DROPPED	AVERTED_WAKEUP_RACE
ACK_RESEND_RACE	QUERY_RECV	MAPPER_ROUTES_UPDATE	DMA_METADATA_RACE
LATE_ACK			

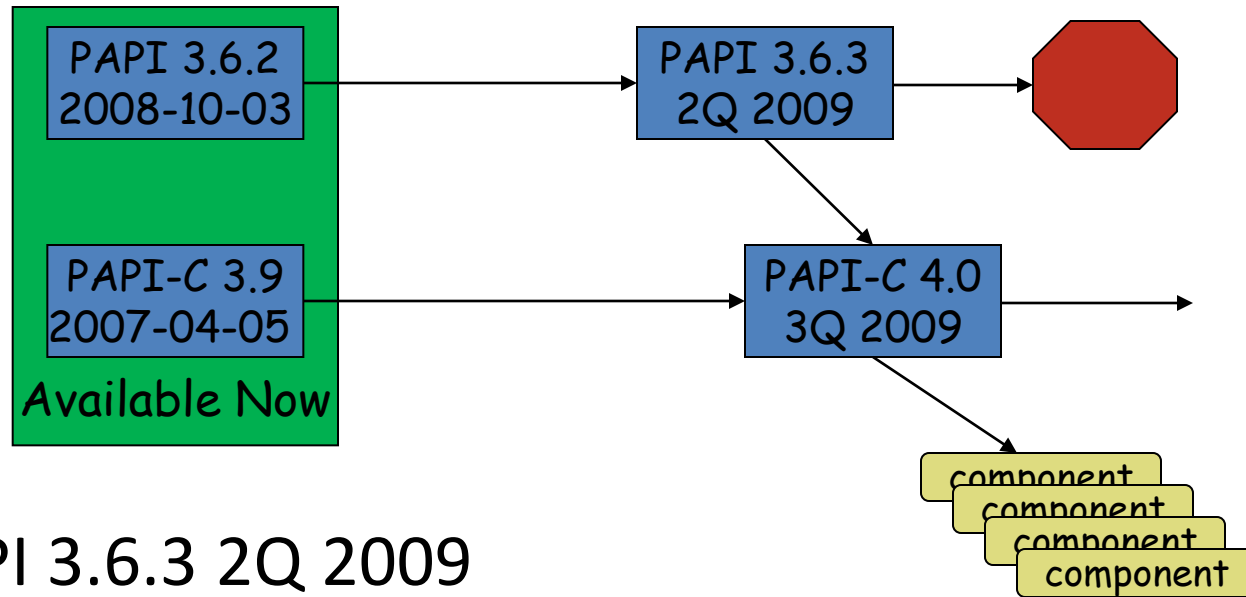
Myrinet MX Counters

LANAI_UPTIME	ACK_NACK_FRAMES_IN_PIPE	REPLY_SEND	ROUTE_DISPERSION
COUNTERS_UPTIME	NACK_BAD_ENDPT	REPLY_RECV	OUT_OF_SEND_HANDLES
BAD_CRC8	NACK_ENDPT_CLOSED	QUERY_UNKNOWN	OUT_OF_PULL_HANDLES
BAD_CRC32	NACK_BAD_SESSION	DATA_SEND_NULL	OUT_OF_PUSH_HANDLES
UNSTRIPPED_ROUTE	NACK_BAD_RDMAWIN	DATA_SEND_SMALL	MEDIUM_CONT_RACE
PKT_DESC_INVALID	NACK_EVENTQ_FULL	DATA_SEND_MEDIUM	CMD_TYPE_UNKNOWN
RCV_PKT_ERRORS	SEND_BAD_RDMAWIN	DATA_SEND_RNDV	UREQ_TYPE_UNKNOWN
PKT_MISROUTED	CONNECT_TIMEOUT	DATA_SEND_PULL	INTERRUPTS_OVERRUN
DATA_SRC_UNKNOWN	CONNECT_SRC_UNKNOWN	DATA_RECV_NULL	WAITING_FOR_INTERRUPT_DMA
DATA_BAD_ENDPT	QUERY_BAD_MAGIC	DATA_RECV_SMALL_INLINE	WAITING_FOR_INTERRUPT_ACK
DATA_ENDPT_CLOSED	QUERY_TIMED_OUT	DATA_RECV_SMALL_COPY	WAITING_FOR_INTERRUPT_TIMER
DATA_BAD_SESSION	QUERY_SRC_UNKNOWN	DATA_RECV_MEDIUM	
PUSH_BAD_WINDOW	RAW SENDS	DATA_RECV_RNDV	SLABS_RECYCLING
PUSH_DUPLICATE	RAW RECEIVES	DATA_RECV_PULL	SLABS_PRESSURE
PUSH_OBSOLETE	RAW_OVERSIZED_PACKETS	ETHER_SEND_UNICAST_CNT	SLABS_STARVATION
PUSH_RACE_DRIVER	RAW_RECV_OVERRUN	ETHER_SEND_MULTICAST_CNT	OUT_OF_RDMA_HANDLES
PUSH_BAD_SEND_HANDLE	RAW_DISABLED		EVENTQ_FULL
MAGIC	CONNECT_SEND	ETHER_RECV_SMALL_CNT	BUFFER_DROP
PUSH_BAD_SRC_MAGIC	CONNECT_RECV	ETHER_RECV_BIG_CNT	MEMORY_DROP
PULL_OBSOLETE	ACK_SEND	ETHER_OVERRUN	HARDWARE_FLOW_CONTROL
PULL_NOTIFY_OBSOLETE	ACK_RECV	ETHER_OVERSIZED	SIMULATED_PACKETS_LOST
PULL_RACE_DRIVER	PUSH_SEND	DATA_RECV_NO_CREDITS	LOGGING_FRAMES_DUMPED
ACK_BAD_TYPE	PUSH_RECV	PACKETS_RESENT	WAKE_INTERRUPTS
ACK_BAD_MAGIC	QUERY_SEND	PACKETS_DROPPED	AVERTED_WAKEUP_RACE
ACK_RESEND_RACE	QUERY_RECV	MAPPER_ROUTES_UPDATE	DMA_METADATA_RACE
LATE_ACK			

PAPI-C Status

- PAPI 3.9 technology preview available with documentation
- PAPI 3.9.x under active development
- Implemented Components:
 - Myrinet MX
 - ACPI temperature sensor component
 - ‘toy’ network component
- Tested on HPC Challenge benchmarks
- Tested platforms include Pentium III, Pentium 4, Core2, Itanium and Opteron
- Platforms in development include Nehalem, POWER, SiCortex, BG/P

PAPI Release Timeline



- PAPI 3.6.3 2Q 2009
 - Nehalem, Shanghai support
 - Terminal releases of ‘PAPI Classic’
- Component PAPI 4.0 by 3Q 2009
 - Components to follow
 - Ideas welcome

New in PAPI

- 3.6.0
 - AMD Barcelona (quad-core)
 - Cray XT3/4, X2 CLE
 - Itanium multi-core
 - FreeBSD support
 - POWER6 / Linux
- 3.6.1
 - SiCortex, Cell
- 3.6.2
 - POWER 5, 5+, 5++, 6 AIX
- 3.6.3
 - Nehalem, Shanghai

Future Directions

- Power PAPI
 - Measure power consumption
- Multi-core support
 - Memory bandwidth
 - Cache sharing
 - Bus and other resource contention
- User-defined events
 - `USER, L2 DTLB miss ratio, DERIVED POSTFIX, |N0|N1|N2|+|/T, DTLB_L1M_L2M, DTLB_L1M_L2H, DTLB_L1M_L2M`
- Multi-CPU PAPI
 - For heterogeneous systems like RoadRunner
- User-driven documentation
 - Wiki man pages
 - Wiki users guide
 - User submitted event configurations

For more information

- PAPI Website: <http://icl.cs.utk.edu/papi/>
 - Software
 - Release notes
 - Documentation
 - Links to tools that use PAPI
 - Mailing/discussion lists

PERFSUITE

Rick Kufrin

National Center for Supercomputing Applications

University of Illinois at Urbana-Champaign

PerfSuite Background

- Active development since Linux clusters were adopted at NCSA in 2001
 - No tools then available for CPU beyond gprof
- UI/NCSA Open Source license approved 2003
- Targeted to users of all levels of expertise
 - The intent is to provide an easy-to-use mechanism for measuring application performance, and to expose problem areas for further exploration
- Low measurement overhead also important
- Close collaboration/sharing with UTK from outset

PerfSuite and POINT

- NSF SDCI program enables maintenance, enhancement, interoperability, and integration
- PerfSuite fills the Level 0 (entry) role for performance measurement within POINT
 - Simple (in most cases, no code change/relink needed)
 - Low overhead (default case is nearly non-intrusive)
 - Limited information... but still very useful and in some cases sufficient
- PerfSuite has never attempted to supply sophisticated graphical/visualization or data management capabilities
 - By partnering with TAU, advanced graphical tools come as a natural by-product
 - PerfDMF infrastructure is mature, and well-suited for importing data collected by PerfSuite
- POINT's application and training thrust (PSC) will expose to wider user base

What Does PerfSuite Provide?

- Overall hardware performance event counts for all or a portion of your application
- Profiling with statistical sampling using either time- or event-based triggers
 - Generalization of the approach used by gprof
- Flexible XML-based output along with various techniques for display, manipulation, combining, transformation
- Information about processor in use (type, cache/TLB specs, etc) – this “metadata” is stored along with measurement
- Functionality available through easy-to-use command line tool that can be used with most applications without need for modification
- Also available through several libraries for finer control

PerfSuite and XML

- In PerfSuite, nearly all data (input, output, configuration, etc) is represented as XML (eXtensible Markup Language) documents
- This provides the ability to manipulate & transform the data in many ways using standard software / skills
- Machine-independent (no binary files)
 - ... opens the data up to the user
- There are numerous high-quality XML-aware libraries available from either compiled or interpreted languages that can make it easy to transform the data for your needs
 - New in PS version 1.0.0: Java API for accessing data
- The structured, well-defined nature of XML makes it natural for import into DB-driven infrastructure such as PerfDMF

PerfSuite Counter-Related Software

- Four performance counter-related utilities:
 - psconfig - configure / select performance events
 - psinv - query events and machine information
 - psrun - generate raw counter or statistical profiling data from an unmodified binary
 - psprocess - pre- and post-process data
- Three libraries (shared and static, serial and threaded)
 - libperfsuite – the “core” library that can be used standalone and will be built regardless of the availability of other software
 - libpshwpc – HardWare Performance Counter library, also built regardless of other software. Without counter support, will only perform time-based profiling through profil() or interval timers.
 - libpshwpc_mpi – a convenience library based on the MPI standard PMPI interface.
- PerfSuite does not require kernel patches

psinv: Processor Inventory

- Lists information about the characteristics of the computer
- This same information is also stored in pstrun XML output and is useful for later generating derived metrics (or for remembering where you ran your program!)
- x86/x86-64 version also shows processor features and descriptions
- Lists available hardware performance events

```
titan:~3% psinv -v
System Information -
Processors:                2
Total Memory:              2007.16 MB
System Page Size:         16.00 KB

Processor Information -
Vendor:                    Intel
Processor family:         IPF
Model (Type):             Itanium
Revision:                 6
Clock Speed:              800.136 MHz

Cache and TLB Information -
Cache levels:             3
Caches/TLBs:             7

Cache Details -
Level 1:
    Type:                  Data
    Size:                  16 KB
    Line size:            32 bytes
    Associativity:        4-way set associative

    Type:                  Instruction
    Size:                  16 KB
    Line size:            32 bytes
    Associativity:        4-way set associative
```

psinv: PAPI Event Summary

```
% psinv -p
PAPI Standard Event Information -
Standard events:      43
Non-derived events:  26
Derived events:      17

PAPI Standard Event Details -
Non-derived:
    PAPI_BR_INS:      Branch instructions
    PAPI_BR_PRC:      Conditional branch instructions correctly predicted
    PAPI_L1_DCA:      Level 1 data cache accesses
    PAPI_L1_DCM:      Level 1 data cache misses
    PAPI_L1_ICM:      Level 1 instruction cache misses
    PAPI_L2_DCA:      Level 2 data cache accesses
    PAPI_L2_DCR:      Level 2 data cache reads
    PAPI_L2_DCW:      Level 2 data cache writes
    PAPI_L2_ICM:      Level 2 instruction cache misses
    PAPI_L2_STM:      Level 2 store misses
    PAPI_L2_TCM:      Level 2 cache misses

Derived:
    PAPI_BR_MSP:      Conditional branch instructions mispredicted
    PAPI_BR_NTK:      Conditional branch instructions not taken
    PAPI_BR_TKN:      Conditional branch instructions taken
    PAPI_FLOPS:       Floating point instructions per second
    PAPI_FP_INS:      Floating point instructions
    PAPI_L1_DCH:      Level 1 data cache hits
```

psrun: Performance Measurement

- Hardware performance counting and profiling with unmodified dynamically-linked executables
- Available for x86, x86-64, and ia64
- POSIX threads support
- Automatic multiplexing
- Can be used with MPI and OpenMP
- Optionally collects resource usage
- Supports all PAPI standard and CPU-native events
- Input/Output = XML documents (can request plain text)

psrun “Cookbook”

```
# First, be sure to set all paths properly (can do in .cshrc/.profile)

% set PSDIR=/opt/perfsuite
% source $PSDIR/bin/psenv.csh

# Use psrun on your program to generate the data,
# then use psprocess to produce an HTML file (default is plain text)

% psrun myprog
% psprocess --html myprog.12345.xml > myprog.html

# Take a look at the results

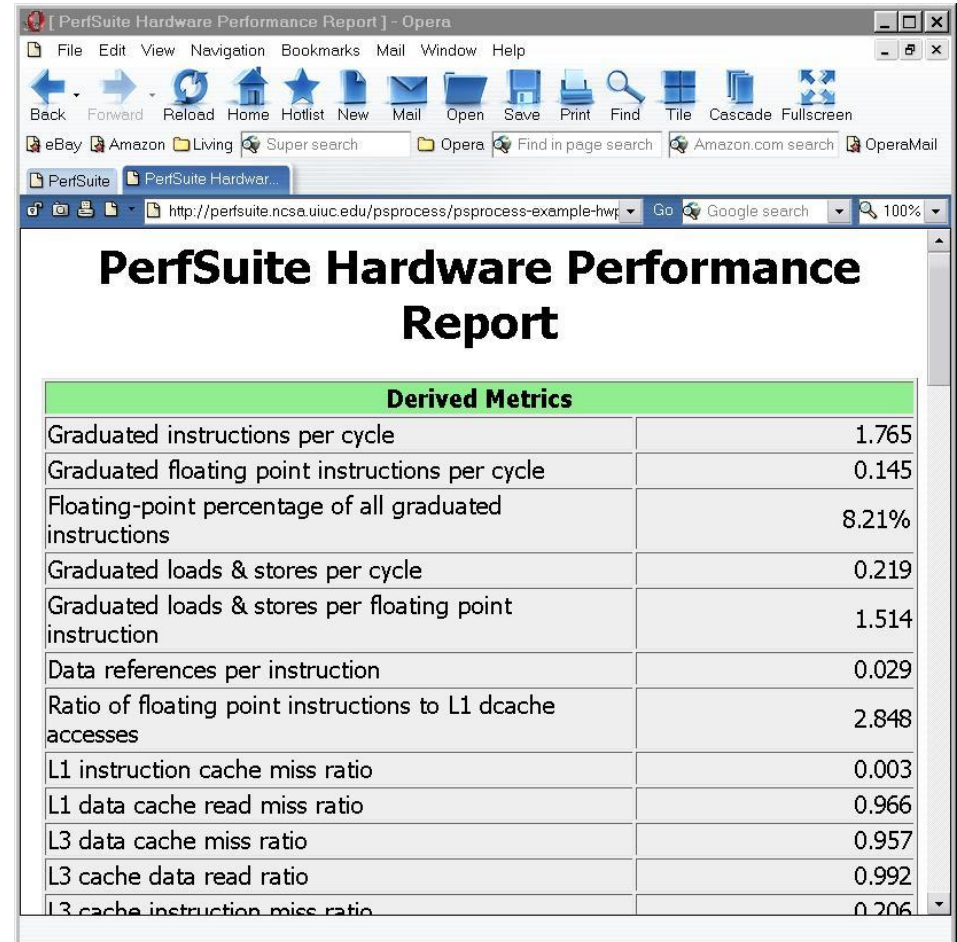
% your-web-browser myprog.html

# Second run, but this time profiling instead of counting

% psrun -C -c papi_profile_cycles.xml myprog
% psprocess -e myprog myprog.67890.xml
```

psprocess: Post-process Results

- This style of output is customizable by you.
- By default, the information it contains and its visual appearance are based on PerfSuite-provided defaults, but these can be easily replaced to suit your preference.
- This output is generated by psprocess using XML Transformations. The stylesheet is in the share/perfsuite/xml/pshwpc subdirectory, with a “xsl” file extension



Derived Metrics	
Graduated instructions per cycle	1.765
Graduated floating point instructions per cycle	0.145
Floating-point percentage of all graduated instructions	8.21%
Graduated loads & stores per cycle	0.219
Graduated loads & stores per floating point instruction	1.514
Data references per instruction	0.029
Ratio of floating point instructions to L1 dcache accesses	2.848
L1 instruction cache miss ratio	0.003
L1 data cache read miss ratio	0.966
L3 data cache miss ratio	0.957
L3 cache data read ratio	0.992
L3 cache instruction miss ratio	0.206

psprocess: Text Mode (default)

PerfSuite Hardware Performance Summary Report

Version : 1.0
Created : Mon Dec 30 11:31:53 AM Central Standard Time 2002
Generator : psprocess 0.5
XML Source : /u/ncsa/anyuser/performance/psrun-ia64.xml

Execution Information

=====
Date : Sun Dec 15 21:01:20 2002
Host : user01

Processor and System Information

=====
Node CPUs : 2
Vendor : Intel
Family : IPF
Model : Itanium
CPU Revision : 6
Clock (MHz) : 800.136
Memory (MB) : 2007.16
Pagesize (KB) : 16

psprocess: Text Mode, cont'd

Cache Information

```
=====  
Cache levels : 3  
-----
```

Level 1

```
Type           : data  
Size (KB)      : 16  
Linesize (B)   : 32  
Assoc          : 4  
Type           : instruction  
Size (KB)      : 16  
Linesize (B)   : 32  
Assoc          : 4  
-----
```

Level 2

```
Type           : unified  
Size (KB)      : 96  
Linesize (B)   : 64  
Assoc          : 6
```

The reports (text or HTML) generated by psprocess have several sections, covering:

- Report creation details
- Run details
- Machine information
- Raw counter listings
- Counter explanations and index
- Derived metrics
- Run annotation defined by you

Derived metrics are evaluated at run-time and can be extended (text mode only)

psprocess: Text Mode, cont'd

Index	Description	Counter Value
1	Conditional branch instructions mispredicted.....	4831072449
4	Floating point instructions.....	86124489172
5	Total cycles.....	594547754568
6	Instructions completed.....	1049339828741

Statistics

Graduated instructions per cycle.....	1.765
Graduated floating point instructions per cycle....	0.145
Level 3 cache miss ratio (data).....	0.957
Bandwidth used to level 3 cache (MB/s).....	385.087
% cycles with no instruction issue.....	10.410
% cycles stalled on memory access.....	43.139
MFLOPS (cycles).....	115.905
MFLOPS (wallclock).....	114.441

Configuring Your Measurement

- All PerfSuite runs are configured according to an XML document that specifies what is to be measured
 - if you don't specify a custom configuration, a default is used
- A custom configuration document (file) is supplied in one of two ways
 - psrun option “-c *filename*”
 - PS_HWPC_CONFIG environment variable, which can be set to *filename*
- Creating new configuration files is easy, and can be done with either a text editor or the tool “psconfig”

Example Configuration

```
<?xml version="1.0" encoding="UTF-8" ?>
•<ps_hwpc_eventlist class="PAPI">
  <ps_hwpc_event type="preset" name="PAPI_BR_MSP" />
  <ps_hwpc_event type="preset" name="PAPI_BR_PRC" />
  <ps_hwpc_event type="preset" name="PAPI_BR_TKN" />
  <ps_hwpc_event type="preset" name="PAPI_FP_INS" />
  <ps_hwpc_event type="preset" name="PAPI_TOT_CYC" />
  <ps_hwpc_event type="preset" name="PAPI_TOT_INS" />
  <ps_hwpc_event type="preset" name="PAPI_L1_DCA" />
  <ps_hwpc_event type="preset" name="PAPI_L1_DCM" />
  <ps_hwpc_event type="preset" name="PAPI_L1_TCM" />
  <ps_hwpc_event type="preset" name="PAPI_L2_DCA" />
  <ps_hwpc_event type="preset" name="PAPI_L2_DCM" />
</ps_hwpc_eventlist>
```

- You can edit this file like any text file
- The XML document root element “ps_hwpc_eventlist” indicates this configuration is to be used for aggregate counting (not profiling)

Using Processor “Native Events”

- It's easy to work with native events in addition to PAPI standard events by modifying the configuration file slightly.
- Instead of using the XML attributes `type="preset"` `name="PAPI_EVENTNAME"`, use the attribute `type="native"` and enclose the event name as the *content* of the element
- Can be used with profiling configurations

```
<ps_hwpc_event type="native">NOPS_RETIRED</ps_hwpc_event>  
<ps_hwpc_event type="native">BACK_END_BUBBLE_ALL</ps_hwpc_event>
```

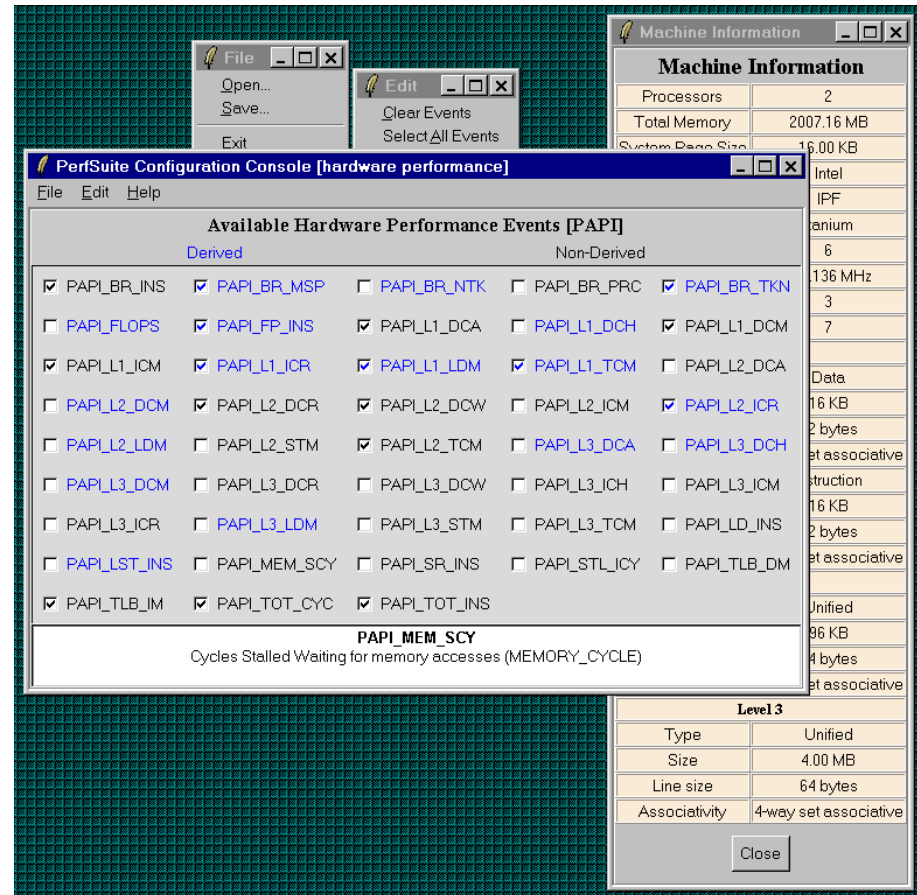
Configuring for Profiling

- Setting up for profiling is similar to counting - all you have to do is modify the XML configuration document:
- The XML document “root element” is now <ps_hwpc_profile>, not <ps_hwpc_eventlist>
- You can supply an optional “threshold”, or sampling rate
- Only one event is allowed in the document
- psconfig does not yet support profiling, need to edit by hand

```
<?xml version="1.0" encoding="UTF-8" ?>
<ps_hwpc_profile class="PAPI">
  <ps_hwpc_event type="preset" name="PAPI_BR_MSP"
    threshold="100000" />
</ps_hwpc_profile>
```

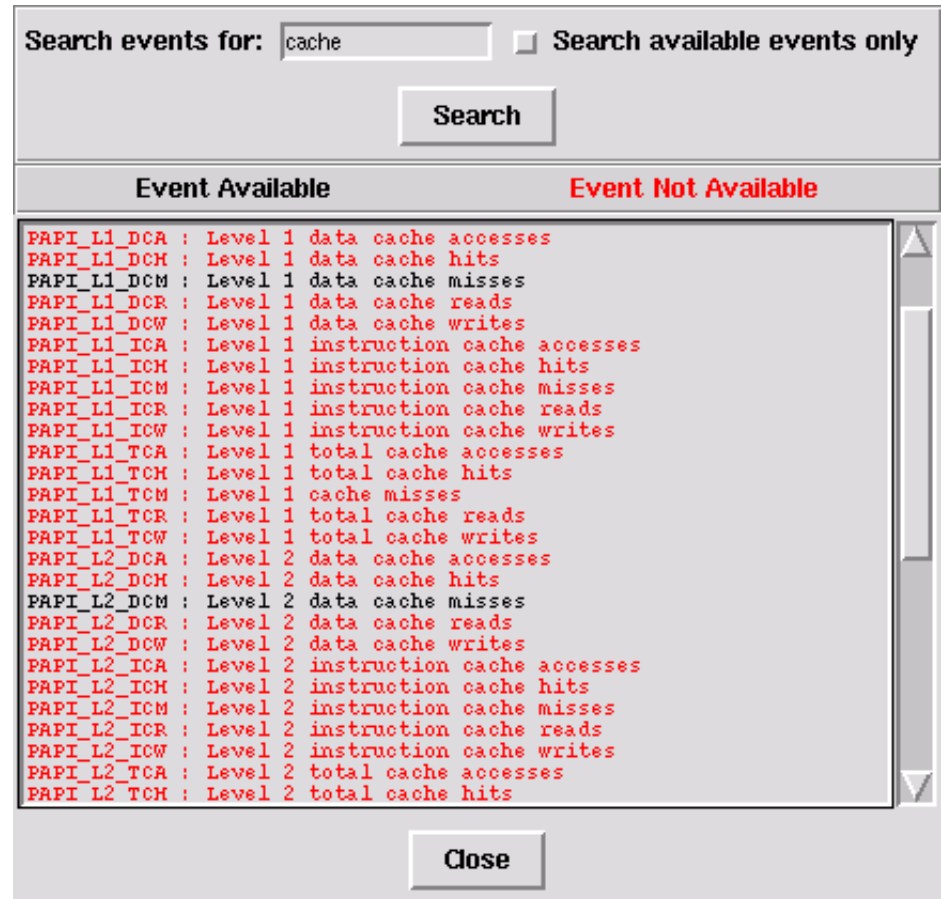
psconfig: Graphical Configuration

- Graphical user interface makes it easy to select events
- Can read in or write out valid XML documents to be used by psrun
- Provides text description of events with mouse click
- Searching capabilities
- Profiling not yet supported

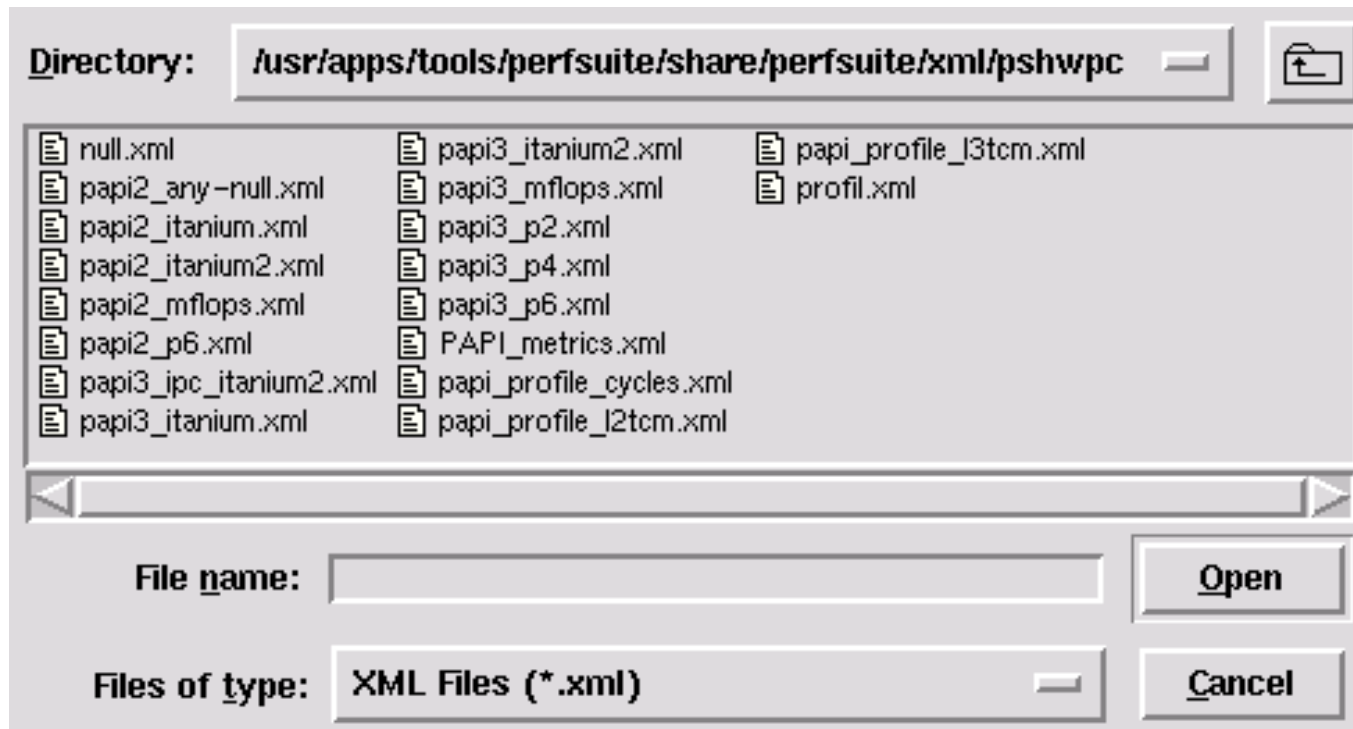


Searching Events with psconfig

- Selecting “Edit”, “Search Events...” brings up a window like this that allows you to search events for keywords
- Can restrict the search to only events available on your computer
- The search is based on the event’s description, not its standard event name (PAPI_TOT_CYC)



Browsing Predefined Event Configurations



- Selecting “File”, “Default Hardware Event Configurations...” brings up the directory with pre-selected configuration documents
- Opening one of them will show you which events will be used
- You can base custom configuration files using these as a start

psrun: Advanced Use

- psrun supports a few options that can be useful in working with shared or distributed memory programs:
- `-p / --pthreads`
 - uses a POSIX thread-aware variant of the library that captures thread creation and measures performance of each, depositing the results in an XML document with the thread ID embedded:
- `-f / --fork`
 - monitors child processes that are created. Not enabled by default.
- `-a / --annotate`
 - inserts an XML “element” with a user-supplied annotation (text)

psprocess: Advanced Use

- `psprocess` is meant to be a “generic” processor for different XML document types generated by PerfSuite. For hardware counting, the most common type is `<hwpcreport>`
- Individual documents can be combined into a “multi-document” with the option `-c / --combine`. With hardware counter data, `psprocess` summarizes the information contained in them with descriptive statistics (mean, max, min, sum, stddev)
- `-s LIST` is a very useful option to be used with profiling runs. `LIST` is a comma-separated list of modules, files, functions, lines used to limit the amount of output
- `-t THRESHOLD` is also helpful in limiting the output of profiling runs. `THRESHOLD` is a number that specifies the minimum % of samples required for a given entry to be displayed. Example: `“-t 2”` means “don’t show me anything that didn’t account for at least 2% of the samples collected”
- `psprocess help` output (`“-h”`) lists all available options and types

psprocess: User-defined Metrics

- `psprocess` allows the creation of user-defined metrics.
- User-defined metrics are stored in a file of your choice that contains expression templates (reminiscent of MathML)
- Select via `PS_HWPC_METRICS` env. variable or “`psprocess -m`”

```
<?xml version="1.0" encoding="UTF-8" ?>
<psmetrics class="hwpc">
  <metric namespace="PAPI" type="ratio">
    <name>PS_RATIO_GINS_CYC</name>
    <description lang="en_US">Graduated instructions per cycle</description>
    <definition>
      <apply>
        <divide>
          <ci>PAPI_TOT_INS</ci>
          <ci>PAPI_TOT_CYC</ci>
        </divide>
      </apply>
    </definition>
  </metric>
</psmetrics>
```

PerfSuite Environment Variables

- PS_HWPC: “off” or “on”, controls whether measurement takes place at all (for API)
- PS_HWPC_CONFIG: set to the name of the XML event file created with psconfig or “by hand”. A default is used if not set
- PS_HWPC_FILE: controls the prefix of the XML output document (default is the name of the command being measured)
- PS_HWPC_ANNOTATION - adds an arbitrary “note” to the XML output
- PS_HWPC_DOMAIN: controls whether counting at user or system level (or both)
- PS_HWPC_THRESHOLD: sets threshold for profiling
- PS_HWPC_FORMAT: “text” or “xml”, controls whether output is in an XML document or plain text (similar to a psprocess report)
- PSRUN_DOFORK: if set (to anything), monitors child processes also

“psrun -h” will show a complete listing of recognized variables

PerfSuite's XML Document Hierarchy

- The basic per-thread XML document that is created by PerfSuite is called an “hwpcreport”
 - These are in either “counting” or “profiling” mode
- Logical collections of the basic documents can be grouped together using the “-c” (“combine”) option to psprocess. The result is called a “multihwpcreport”.
 - This is where the notion of a parallel run of arbitrary scale enters and can be applied to shared- or distributed-memory runs
 - Subsequent processing with psprocess recognizes these “multi” documents and provides different statistics, more appropriate for parallel runs
- The basic concept is extensible to further logical collections of one or more runs, threads, tasks, etc

Example: Parallel CFD Kernel

- ASPCG (Additive Schwarz Preconditioned Conjugate Gradient)
- Courtesy Prof. Danesh Tafti (Virginia Tech)
- Fortran77 kernel with OpenMP directives closely approximates the per-node performance of a full-featured parallel computational fluid dynamics application (GenIDLEST)

Generating Overall Event Counts

- All that is necessary is to ensure that your application is linked dynamically
- **psrun**, when configured with PAPI support, collects aggregate counting data over the entire run by default
- The option “-p” requests per-thread collection (for POSIX thread programs)
- A successful run will result in N+1 XML documents, which are input to **psprocess**

Data Collection with 1 and 2 Threads

```
$ OMP_NUM_THREADS=1 time psrun -p ./aspcg
  L2 norm in CG after   25 Iterations is   2.7472E-02

flop is   13199785984.0000

  Time spent in matxvec is   0.2357E+02 with dtime
  Approximate performance based on dtime is =  0.56002E+03 MFLOPS/s
23.58user 0.44system 0:24.33elapsed 98%CPU
$ ls aspcg*.xml
aspcg.0.6461.twinpeaks.xml  aspcg.1.6461.twinpeaks.xml
$ OMP_NUM_THREADS=2 time psrun -p ./aspcg
27.82user 0.59system 0:15.09elapsed 188%CPU
$ ls aspcg*.xml
aspcg.0.6461.twinpeaks.xml  aspcg.1.6578.twinpeaks.xml
aspcg.0.6578.twinpeaks.xml  aspcg.2.6578.twinpeaks.xml
aspcg.1.6461.twinpeaks.xml
```

Note: All Threads are Tracked

- Some OpenMP compilers (e.g., Intel) may create an additional “monitor thread” that is probably not relevant to your analysis
- Which thread is usually apparent from the output of **psprocess**:

```
$ psprocess aspcg.1.6461.twinpeaks.xml
MFLOPS (wall clock)..... 0.000
MIPS (wall clock)..... 0.000
CPU time (seconds)..... 0.000
Wall clock time (seconds)..... 24.081
% CPU utilization..... 0.000
```

```
$ psprocess aspcg.0.6461.twinpeaks.xml
MFLOPS (wall clock)..... 526.521
MIPS (wall clock)..... 1463.966
CPU time (seconds)..... 22.803
Wall clock time (seconds)..... 24.101
% CPU utilization..... 94.614
```

Profiling with 2 Threads

```
$ psrun -p -C -c papi_profile_cycles.xml ./aspcg
$ ls aspcg*.xml
aspcg.0.6461.twinpeaks.xml  aspcg.1.6578.twinpeaks.xml
aspcg.0.6578.twinpeaks.xml  aspcg.1.6844.twinpeaks.xml
aspcg.0.6844.twinpeaks.xml  aspcg.2.6578.twinpeaks.xml
aspcg.1.6461.twinpeaks.xml  aspcg.2.6844.twinpeaks.xml
$ psprocess aspcg.0.6844.twinpeaks.xml
$ psprocess -x aspcg.0.6844.twinpeaks.xml
```

- Supplying `-C` requests that the PerfSuite standard configuration file directory is searched. “papi_profile_cycles.xml” is installed there by default, and directs profiling using the PAPI event `PAPI_TOT_CYC`
- To avoid proliferation of similarly-named output documents, consider using the `-o FILENAME` option to the `psrun` command line (not done above)
- By default, processing a profile will send a text-based profile report to standard output
- You can instead request an XML-based output document be created by using the `psprocess` option `-x`. These documents can be used by other tools such as TAU’s Paraprof visualizer:

```
$ paraprof prof-t0.xml prof-t1.xml prof-t2.xml
```

psprocess Text-Based Profiles

Profile Information

```
=====  
=====  
Class           : PAPI  
Version        : 3.6.2  
Event          : PAPI TOT_CYC (Total cycles)  
Period         : 100000  
Samples        : 200471  
Domain         : user  
Run Time       : 27.19 (seconds)  
Min Self %     : (all)
```

Module Summary

```
-----  
Samples  Self %  Total %  Module  
-----  
186068   92.82%   92.82%  /home/rkufrin/apps/aspcg/aspcg  
14182    7.07%   99.89%  /opt/intel/cc/9.0/lib/libguide.so  
187      0.09%   99.98%  /lib/ld-2.3.6.so  
18       0.01%   99.99%  /lib/tls/libc-2.3.6.so  
15       0.01%  100.00%  /lib/tls/libpthread-2.3.6.so  
1        0.00%  100.00%  /tmp/perfsuite/lib/libpsrun_r.so.0.0.1
```

File Summary

```
-----  
Samples  Self %  Total %  File  
-----  
154346   76.99%   76.99%  /home/rkufrin/apps/aspcg/pc_jac2d_blk3.f  
14506    7.24%   84.23%  /home/rkufrin/apps/aspcg/cg3_blk.f  
14505    7.24%   91.46%  ??  
10185    5.08%   96.54%  /home/rkufrin/apps/aspcg/matxvec2d_blk3.f  
3042     1.52%   98.06%  /home/rkufrin/apps/aspcg/dot_prod2d_blk3.f  
2366     1.18%   99.24%  /home/rkufrin/apps/aspcg/add_exchange2d_blk3.f  
834      0.42%   99.66%  /home/rkufrin/apps/aspcg/main3.f  
687      0.34%  100.00%  /home/rkufrin/apps/aspcg/cs_jac2d_blk3.f
```

Text-based profiles, cont'd

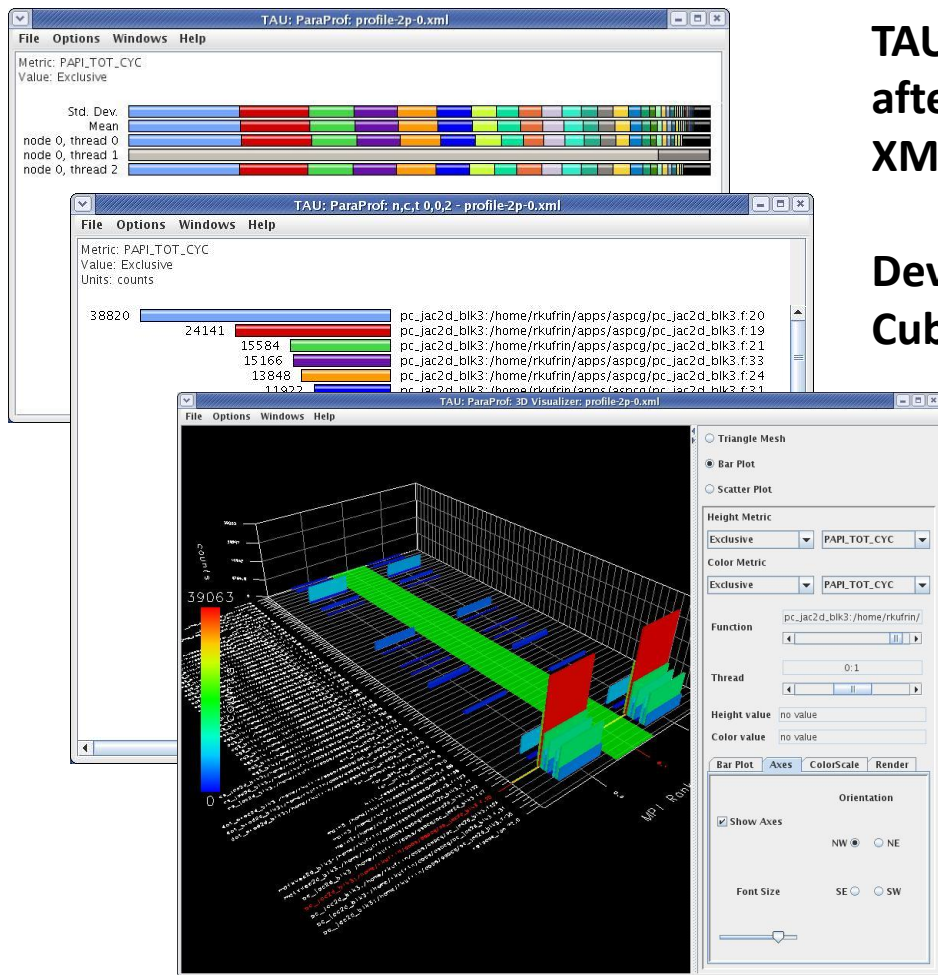
Function Summary

Samples	Self %	Total %	Function
154346	76.99%	76.99%	pc_jac2d_blk3
14506	7.24%	84.23%	cg3_blk
10185	5.08%	89.31%	matxvec2d_blk3
6937	3.46%	92.77%	__kmp_x86_pause
4711	2.35%	95.12%	__kmp_wait_sleep
3042	1.52%	96.64%	dot_prod2d_blk3
2366	1.18%	97.82%	add_exchange2d_blk3

Function:File:Line Summary

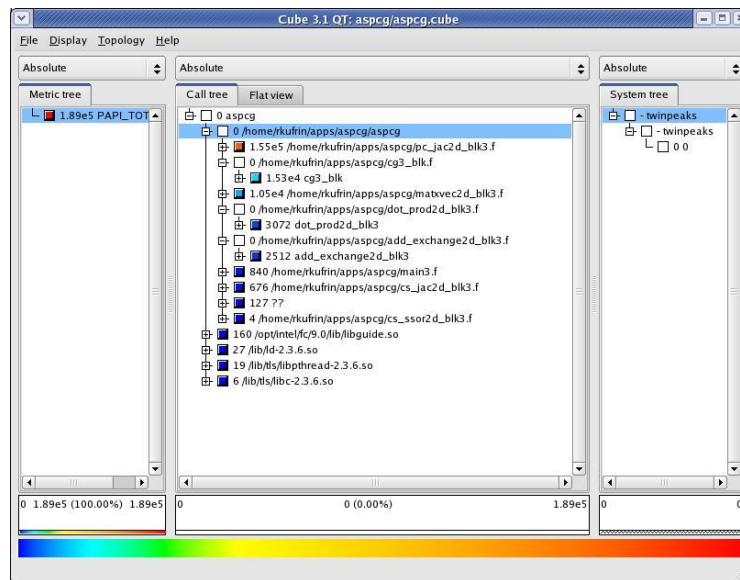
Samples	Self %	Total %	Function:File:Line
39063	19.49%	19.49%	pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:20
24134	12.04%	31.52%	pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:19
15626	7.79%	39.32%	pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:21
15028	7.50%	46.82%	pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:33
13878	6.92%	53.74%	pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:24
11880	5.93%	59.66%	pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:31
8896	4.44%	64.10%	pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:22
7863	3.92%	68.02%	matxvec2d_blk3:/home/rkufrin/apps/aspcg/matxvec2d_blk3.f:19
7145	3.56%	71.59%	pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:32

PerfSuite Profiles with ParaProf and Cube3



TAU's Paraprof can display PerfSuite profiles after being mapped to source and stored as XML (psprocess -x)

Development version of psprocess produces Cube XML files directly



PerfSuite Library Access (API)

- All of the functionality is also available from within your program (C/C++/Fortran) through a small API
- Same XML documents are read, same XML documents are written, small additional functionality
- Why would you want to use this?
 - Primarily to gain finer control over where measurements are taken in your program. For example, you might defer measurement until program initialization has completed
- For complex uses, you are probably better off using an “industrial-strength” performance library
- The intent of the API is to “abstract out” the process of performance measurement to a very high level

libperfsuite: Core Library

- This library is available regardless of the presence of hardware counter support
- Small number of useful routines callable from either C or FORTRAN (use “PSF_” instead of “ps_” with FORTRAN)

```
int ps_cpuspeed          (double *mhz);
int ps_cpuusage         (pid_t pid, ps_time_t *utime,
                        ps_time_t *stime);
int ps_dmemusage        (float *total_mb, float *used_mb,
                        float *free_mb);
int ps_memusage         (pid_t pid, float *vsize_mb,
                        float *rss_mb);
int ps_procstat         (pid_t pid,
                        ps_procstat_t *p);
int ps_rtc              (unsigned long long *rtcval);
int ps_rtcinit          (void);
const char *ps_strerror (int code);
```

- `#include <perfsuite.h>` (or “fperfsuite.h”)

libpshwpc: Performance Collection API

C / C++

```
ps_hwpc_init (void)
ps_hwpc_start (void)
ps_hwpc_read (long long *values)
ps_hwpc_suspend (void)
ps_hwpc_stop (char *prefix)
ps_hwpc_shutdown (void)
```

Fortran

```
call psf_hwpc_init (ierr)
call psf_hwpc_start (ierr)
call psf_hwpc_read (integer*8
    values,ierr)
call psf_hwpc_suspend (ierr)
call psf_hwpc_stop (prefix, ierr)
call psf_hwpc_shutdown (ierr)
```

- Call “init” once, call “start”, “read” and “suspend” as many times as you like. Call “stop” (supplying a file name prefix of your choice) to get the performance data XML document.
- Optionally, call “shutdown”.
- Example programs demonstrating use are installed in PerfSuite “examples” subdirectory.
- Additional routines ps_hwpc_numevents() and ps_hwpc_eventnames() allow querying current configuration

FORTRAN API Example

```
include 'fperfsuite.h'
call PSF_hwpc_init(ierr)
call PSF_hwpc_start(ierr)
do j = 1, n
  do i = 1, m
    do k = 1, 1
      c(i,j) = c(i,j) + a(i,k)*b(k,j)
    end do
  end do
end do
call PSF_hwpc_stop('perf', ierr)
call PSF_hwpc_shutdown(ierr)
```

```
% ifort -c matmult.f -I/opt/perfsuite/include
```

```
% ifort matmult.o -L /usr/apps/tools/perfsuite/lib/intel
-L/usr/apps/tools/papi/lib -lpshwpc -lperfsuite -lpapi
```

PerfSuite XML Java API

- Provides programmatic access to the information contained in PerfSuite reports through Java.
- Includes detailed Javadoc documentation that is installed in `$PREFIX/share/perfsuite/doc/javadoc`
- Currently supports HWPC report (“count” and “profile” mode), resource report and multi-HWPC reports; parses all elements in them and places the data in Java objects that can be accessed via “getter” methods.

```
$ JARFILE=$PREFIX/share/perfsuite/javalib/perfsuite.jar  
$ javac -classpath $JARFILE MyClass.java  
$ java -classpath $JARFILE:. MyClass <arguments>
```

Example Use of PS XML Java API

```
import java.util.*;
import org.perfsuite.xml.*;

// The "newInstance" method is used to parse any supported type of XML
// document that PerfSuite generates. It accepts the name of the
// file to parse and a flag to indicate whether XML validation is done.
PS_Report report0 = PS_Report.newInstance (filename, false);

// Use "instanceof" to determine the type of report that was parsed.
// This example shows how to handle a report with event totals.
if (report0 instanceof PS_HwpcCountingReport) {
    PS_HwpcCountingReport report = (PS_HwpcCountingReport) report0;
    Map<String, PS_HwpcEvent> eventMap = report.getEvents();
    for (Iterator it = eventMap.entrySet().iterator(); it.hasNext(); ) {
        Map.Entry entry = (Map.Entry) it.next();
        PS_HwpcEvent event = (PS_HwpcEvent) entry.getValue();
        System.out.println ("Event: " + event.getName() +
                             ", Count: " + event.getCount() +
                             ", Type: " + event.getType() +
                             ", Derived: " + event.getDerived());
    }
}
```

Issues at Higher Scales of Parallelism

- How well can PerfSuite be expected to scale to extreme levels of parallelism?
 - All monitoring is contained within the context of a single core/processor/thread. No communication or synchronization required between threads as measurement proceeds, so not impacted.
 - Currently, results/output are written to local disk files; PerfSuite enforces serialized output from multithreaded programs to minimize filesystem contention. Not an issue to date, but warrants rethinking.
 - PC-to-source code mapping (for profiling runs) is currently done through the psprocess command, and can consume significant times for large programs at high levels of parallelism.
- While PerfSuite has been used successfully on core counts of hundreds to thousands, further work needs to be done to improve existing barriers to scalability. These issues are a key piece of work ongoing under the POINT collaboration.

Recent and Upcoming in PerfSuite

- Current stable release is version 0.6.2
 - Provides nearly all of the features covered in this presentation
- Version 1.0 is now in alpha release state
 - Alpha releases are for incorporating new features, major modifications
 - Much new functionality and reengineering on the roadmap:
 - *In current alpha:*
 - New Java API for user access to PerfSuite XML documents (do what you like with the data PerfSuite collects).
 - New support for Cube3 output.
 - *For later release in alpha cycle:*
 - Extending the ability to collect performance data from traditional programming languages to Java
 - Enhanced profiling capabilities, including substantial reduction in memory requirements for profiling runs
 - New Java-based implementation of `psprocess`
 - Improved scalability of profiling output and post-processing for parallel runs
 - Current and potential users' feedback, bug reports, encouraged

For More Information and Downloads

- PerfSuite web sites:
 - <http://perfsuite.ncsa.uiuc.edu/>
 - <http://www.sf.net/projects/perfsuite/>



TAU PERFORMANCE SYSTEM

Sameer Shende

Alan Morris, Wyatt Spear, Scott Biersdorff

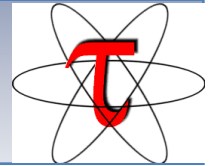
Performance Research Lab

Allen D. Malony, Kevin Huck, Aroon Nataraj

Department of Computer and Information Science

University of Oregon

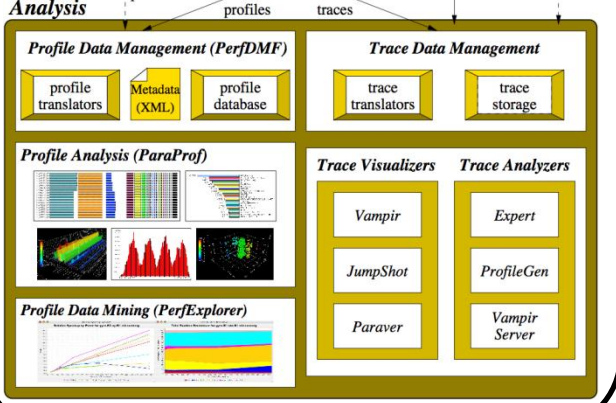
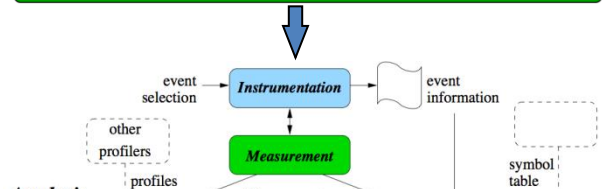
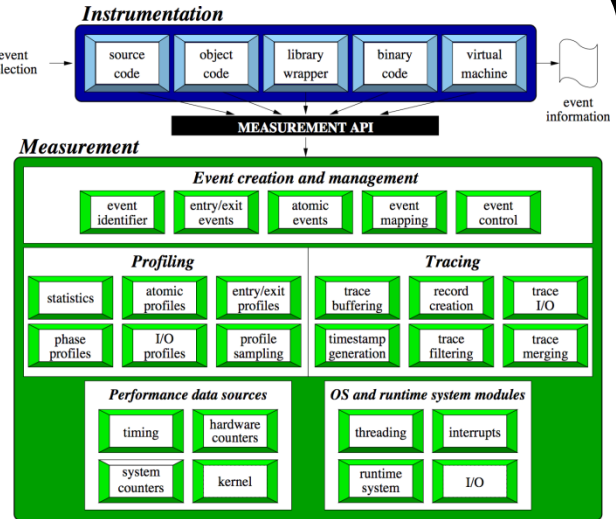
TAU Performance System[®]



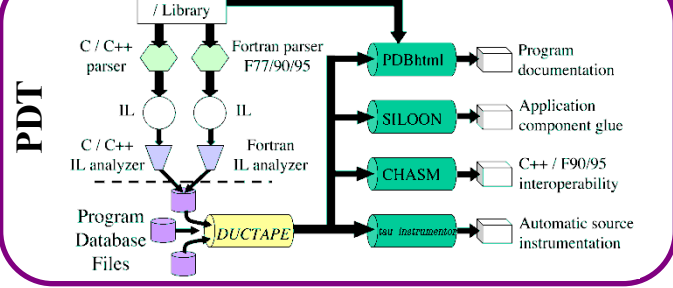
- Tuning and Analysis Utilities (16+ year project)
- Performance problem solving framework for HPC
 - Integrated, scalable, flexible, portable
 - Target all parallel programming / execution paradigms
- Integrated performance toolkit (open source)
 - Instrumentation, measurement, analysis, visualization
 - Widely-ported performance profiling / tracing system
 - Performance data management and data mining
- Broad application use (NSF, DOE, DOD, ...)

TAU Performance System Components

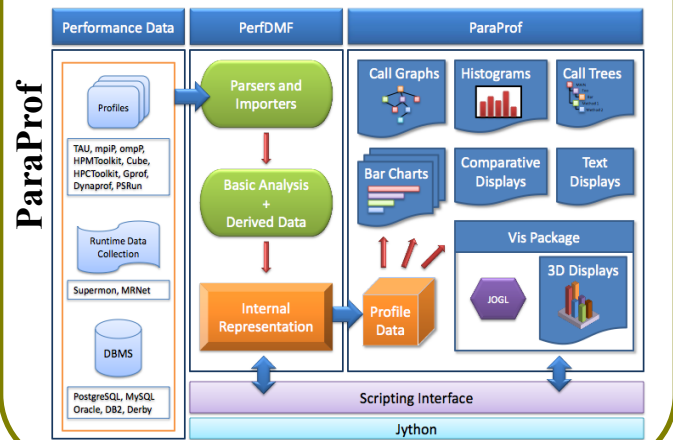
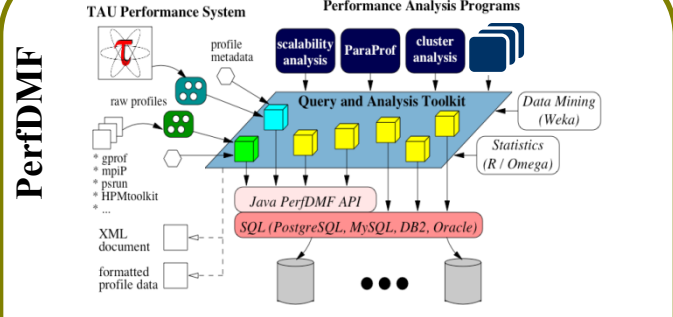
TAU Architecture



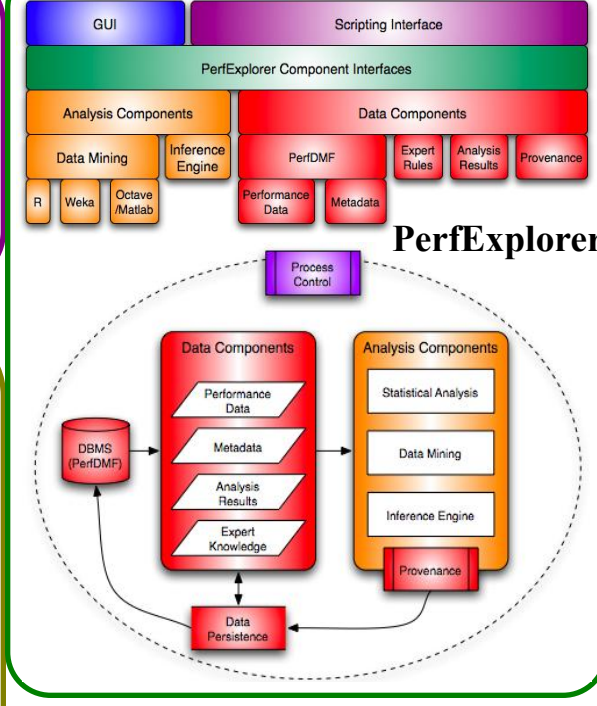
Program Analysis



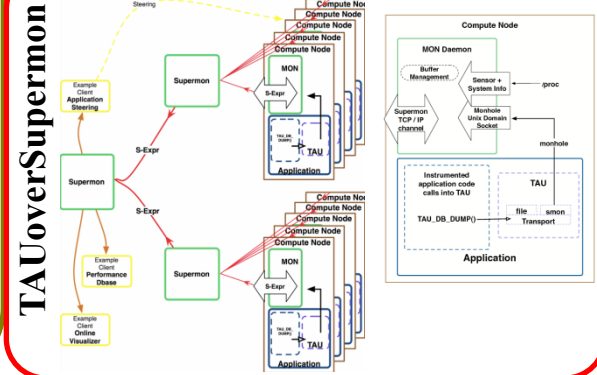
Parallel Profile Analysis



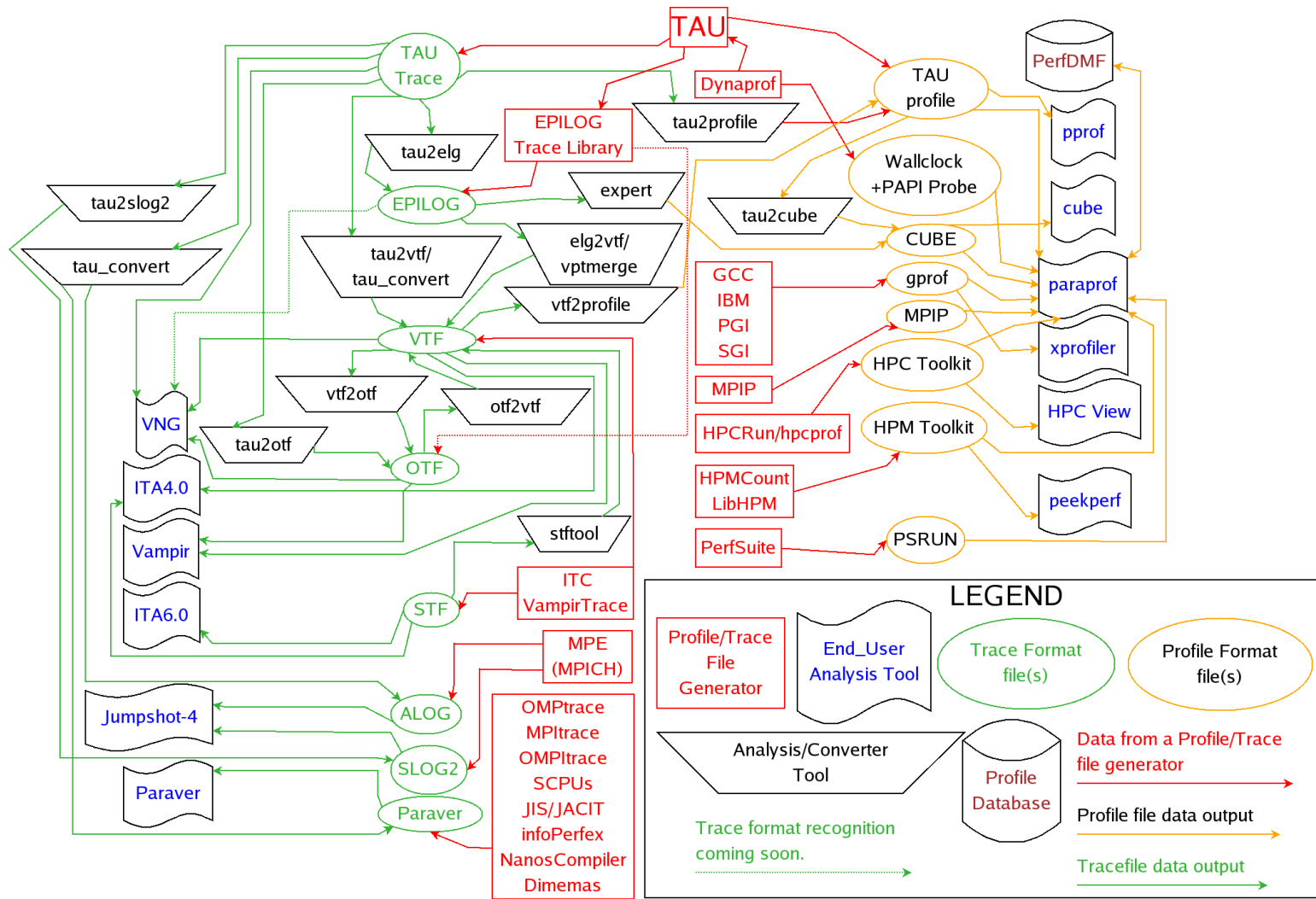
Performance Data Mining



Performance Monitoring

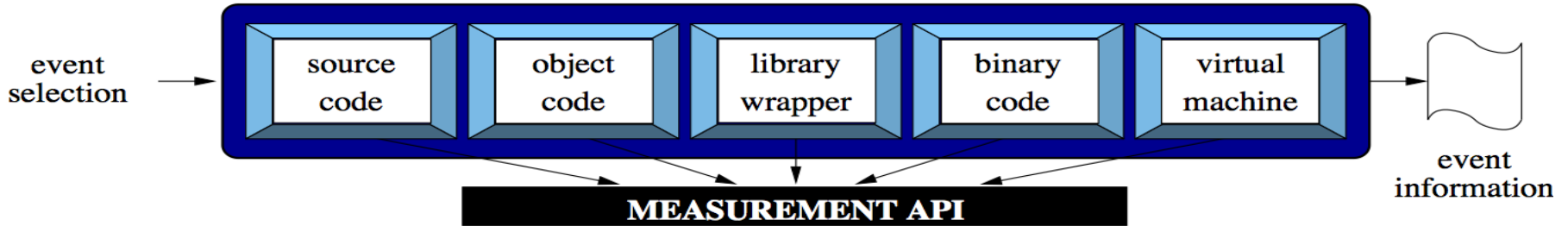


Building Bridges to Other Tools

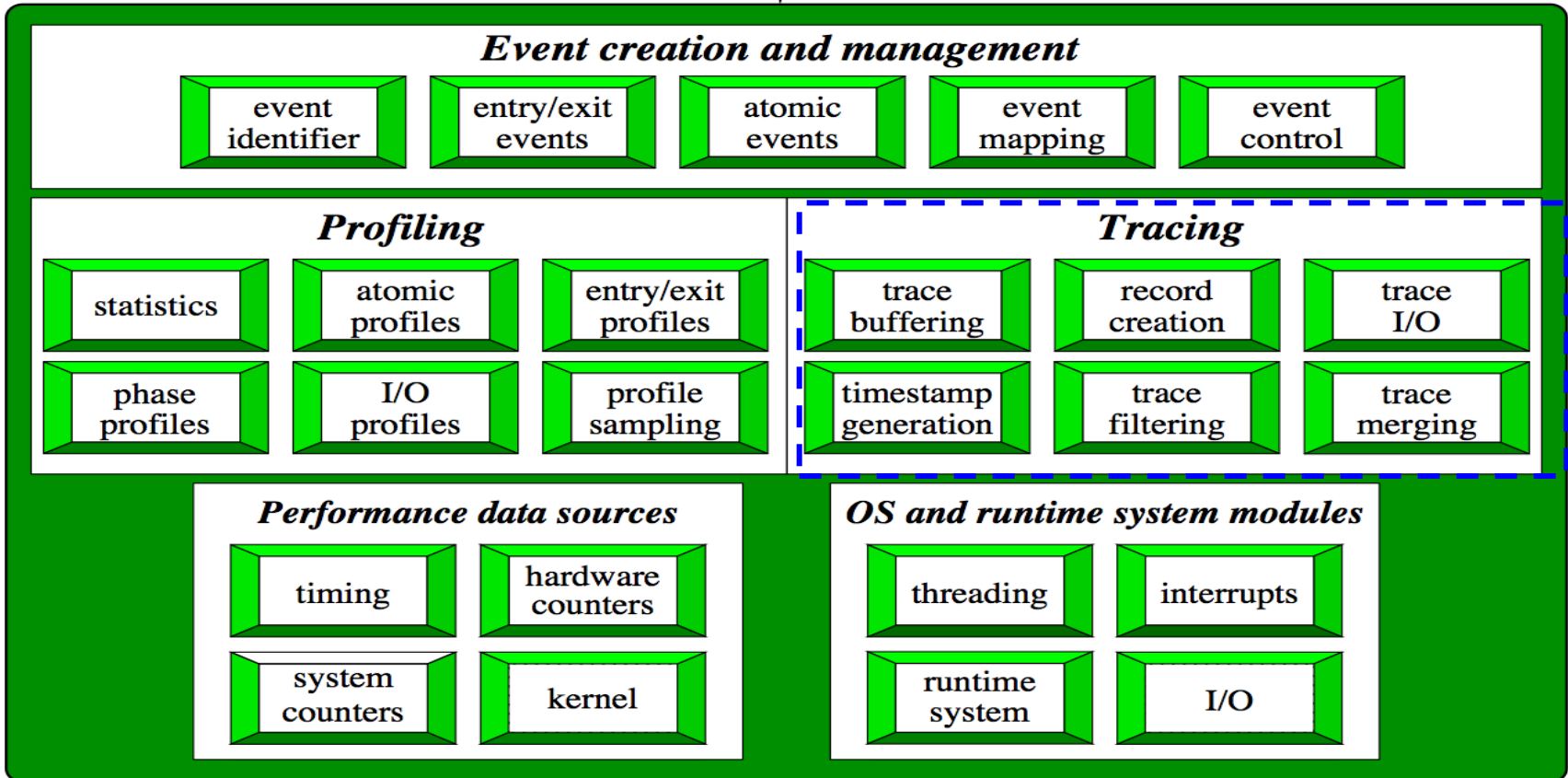


TAU Instrumentation / Measurement

Instrumentation



Measurement



Direct Performance Observation

- Execution actions of interest exposed as events
 - In general, actions reflect some execution state
 - presence at a code location or change in data
 - occurrence in parallelism context (thread of execution)
 - Events encode actions for performance system to observe
- Observation is direct
 - Direct instrumentation of program (system) code (probes)
 - Instrumentation invokes performance measurement
 - Event measurement: performance data, meta-data, context
- Performance experiment
 - Actual events + performance measurements
- Contrast with (indirect) event-based sampling

TAU Instrumentation Approach

- Support for standard program events
 - Routines, classes and templates
 - Statement-level blocks
 - Begin/End events (Interval events)
- Support for user-defined events
 - Begin/End events specified by user
 - Atomic events (e.g., size of memory allocated/freed)
 - Flexible selection of event statistics
- Provides static events and dynamic events
- Enables “semantic” mapping
- Specification of event groups (aggregation, selection)
- Instrumentation optimization

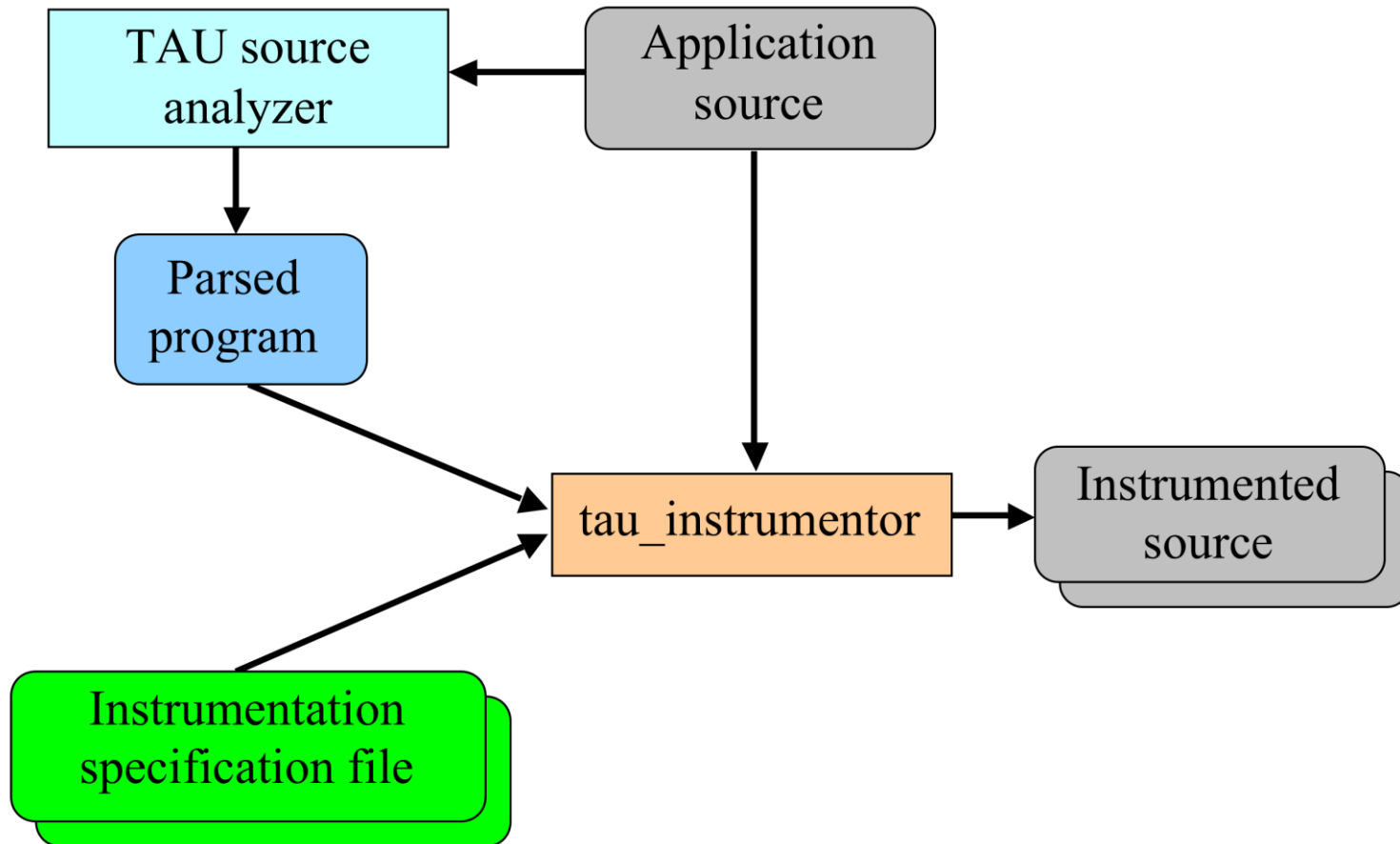
TAU Event Interface

- Events have a type, a group association, and a name
- TAU event names are character strings
 - Powerful way to encode event information
 - Inefficient way to communicate each event occurrence
- TAU maps a new event name to an event ID
 - Done when event is first encountered (get event handle)
 - Event ID is used for subsequent event occurrences
 - Assigning a uniform event ID a priori is problematic
- A new event is identified by a new event name in TAU
 - Can create new event names at runtime
 - Allows for dynamic events (TAU renames events)
 - Allows for context-based, parameter-based, phase events

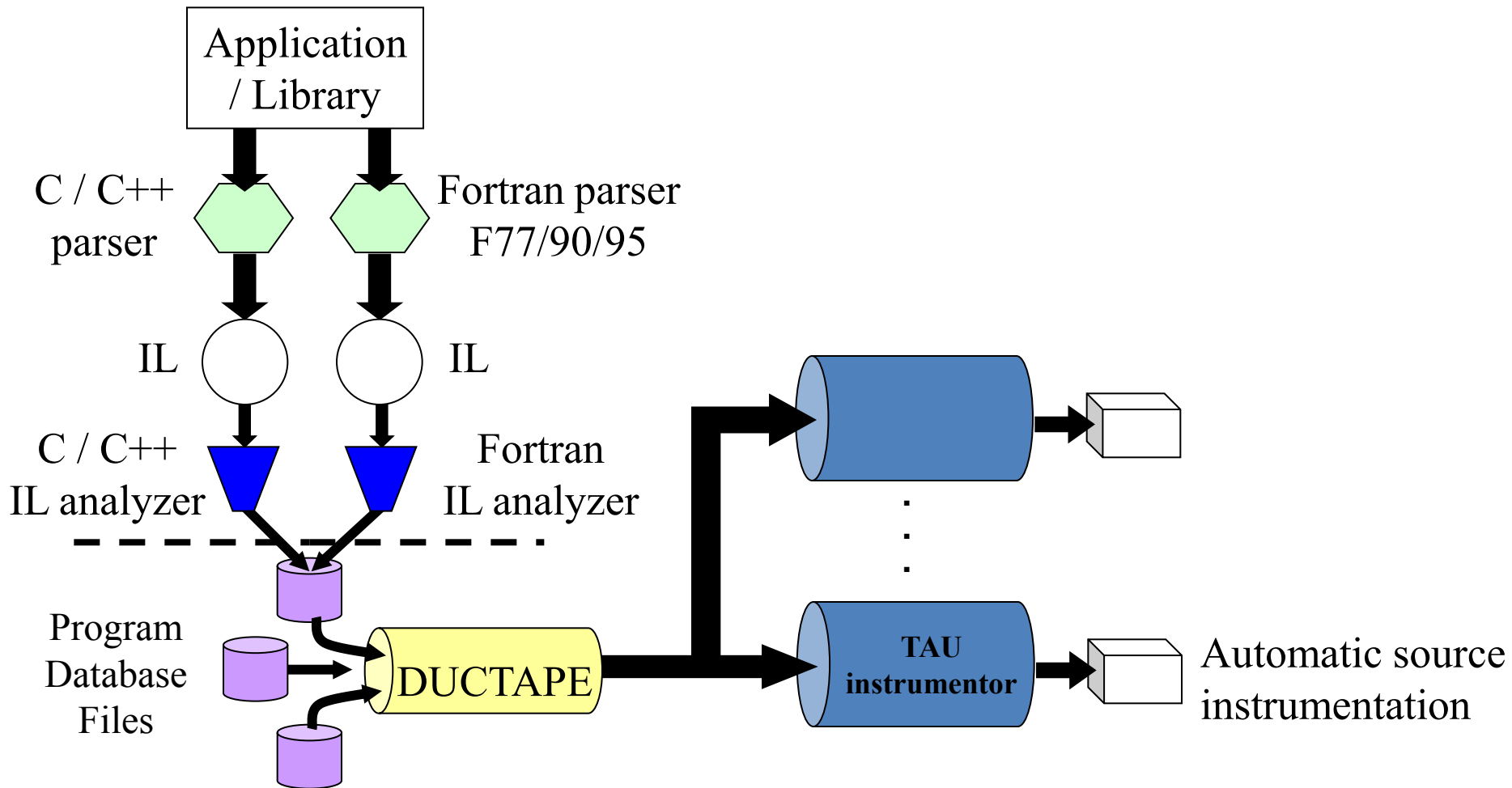
TAU Instrumentation Mechanisms

- Source code
 - Manual (TAU API, TAU component API)
 - Automatic (robust)
 - C, C++, F77/90/95 (Program Database Toolkit (PDT))
 - OpenMP (directive rewriting (Opari), POMP2 spec)
 - Library header wrapping
- Object code
 - Pre-instrumented libraries (e.g., MPI using PMPI)
 - Statically- and dynamically-linked (with LD_PRELOAD)
- Executable code
 - Binary and dynamic instrumentation (Dyninst)
 - Virtual machine instrumentation (e.g., Java using JVMPI)
- TAU_COMPILER to automate instrumentation process

Automatic Source-level Instrumentation



Program Database Toolkit (PDT)



MPI Wrapper Interposition Library

- Uses standard MPI Profiling Interface
 - Provides name shifted interface
 - MPI_Send = PMPI_Send
 - Weak bindings
- Create TAU instrumented MPI library
 - Interpose between MPI and TAU
 - Done during program link
 - -lmpi replaced by -lTauMpi -lpmpi -lmpi
 - No change to the source code!
 - Just re-link application to generate performance data

MPI Shared Library Instrumentation

- Interpose the MPI wrapper library for applications that have already been compiled
 - Avoid re-compilation or re-linking
- Requires shared library MPI
 - Uses LD_PRELOAD for Linux
 - On AIX use MPI_EUILIB / MPI_EUILIBPATH
 - Does not work on XT3
- Approach will work with other shared libraries
- Use TAU tauex
 - % mpirun -np 4 tauex a.out

Selective Instrumentation File

- Specify a list of events to exclude or include
- # is a wildcard in a routine name

```
BEGIN_EXCLUDE_LIST
```

```
Foo
```

```
Bar
```

```
D#EMM
```

```
END_EXCLUDE_LIST
```

```
BEGIN_INCLUDE_LIST
```

```
int main(int, char **)
```

```
F1
```

```
F3
```

```
END_INCLUDE_LIST
```

Selective Instrumentation File

- Optionally specify a list of files
- * and ? may be used as wildcard characters

```
BEGIN_FILE_EXCLUDE_LIST
```

```
f*.f90
```

```
Foo?.cpp
```

```
END_FILE_EXCLUDE_LIST
```

```
BEGIN_FILE_INCLUDE_LIST
```

```
main.cpp
```

```
foo.f90
```

```
END_FILE_INCLUDE_LIST
```

Selective Instrumentation File

- User instrumentation commands
 - Placed in INSTRUMENT section
 - Routine entry/exit
 - Arbitrary code insertion
 - Outer-loop level instrumentation

```
BEGIN_INSTRUMENT_SECTION
loops file="foo.f90" routine="matrix#"
memory file="foo.f90" routine="#"
io routine="matrix#"
[static/dynamic] phase routine="MULTIPLY"
dynamic [phase/timer] name="foo" file="foo.cpp" line=22 to line=35
file="foo.f90" line = 123 code = " print *, \" Inside foo\""
exit routine = "int foo()" code = "cout <<\"exiting foo\"<<endl;"
END_INSTRUMENT_SECTION
```

TAU Measurement Approach

- Portable and scalable parallel profiling solution
 - Multiple profiling types and options
 - Event selection and control (enabling/disabling, throttling)
 - Online profile access and sampling
 - Online performance profile overhead compensation
- Portable and scalable parallel tracing solution
 - Trace translation to OTF, EPILOG, Paraver, and SLOG2
 - Trace streams (OTF) and hierarchical trace merging
- Robust timing and hardware performance support
- Multiple counters (hardware, user-defined, system)
- Performance measurement of I/O and Linux kernel

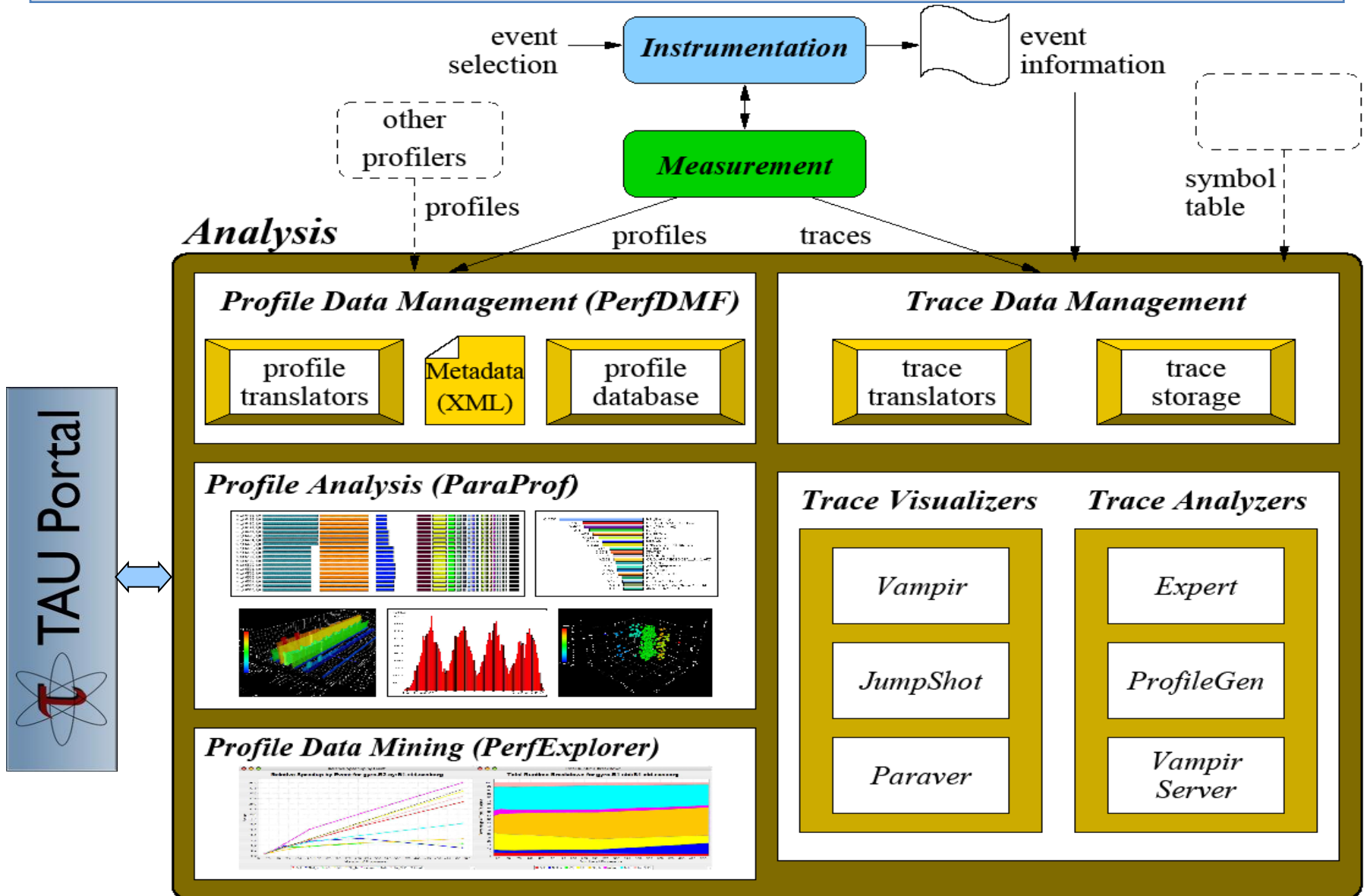
TAU Measurement Mechanisms

- Parallel profiling
 - Function-level, block-level, statement-level
 - Supports user-defined events and mapping events
 - Support for flat, callgraph/callpath, phase profiling
 - Support for parameter and context profiling
 - Support for tracking I/O and memory (library wrappers)
 - Parallel profile stored (dumped, shapshot) during execution
- Tracing
 - All profile-level events
 - Inter-process communication events
 - Inclusion of multiple counter data in traced events

Types of Parallel Performance Profiling

- Flat profiles
 - Metric (e.g., time) spent in an event (callgraph nodes)
 - Exclusive/inclusive, # of calls, child calls
- Callpath profiles (Calldepth profiles)
 - Time spent along a calling path (edges in callgraph)
 - “main=> f1 => f2 => MPI_Send” (event name)
 - TAU_CALLPATH_DEPTH environment variable
- Phase profiles
 - Flat profiles under a phase (nested phases are allowed)
 - Default “main” phase
 - Supports static or dynamic (per-iteration) phases

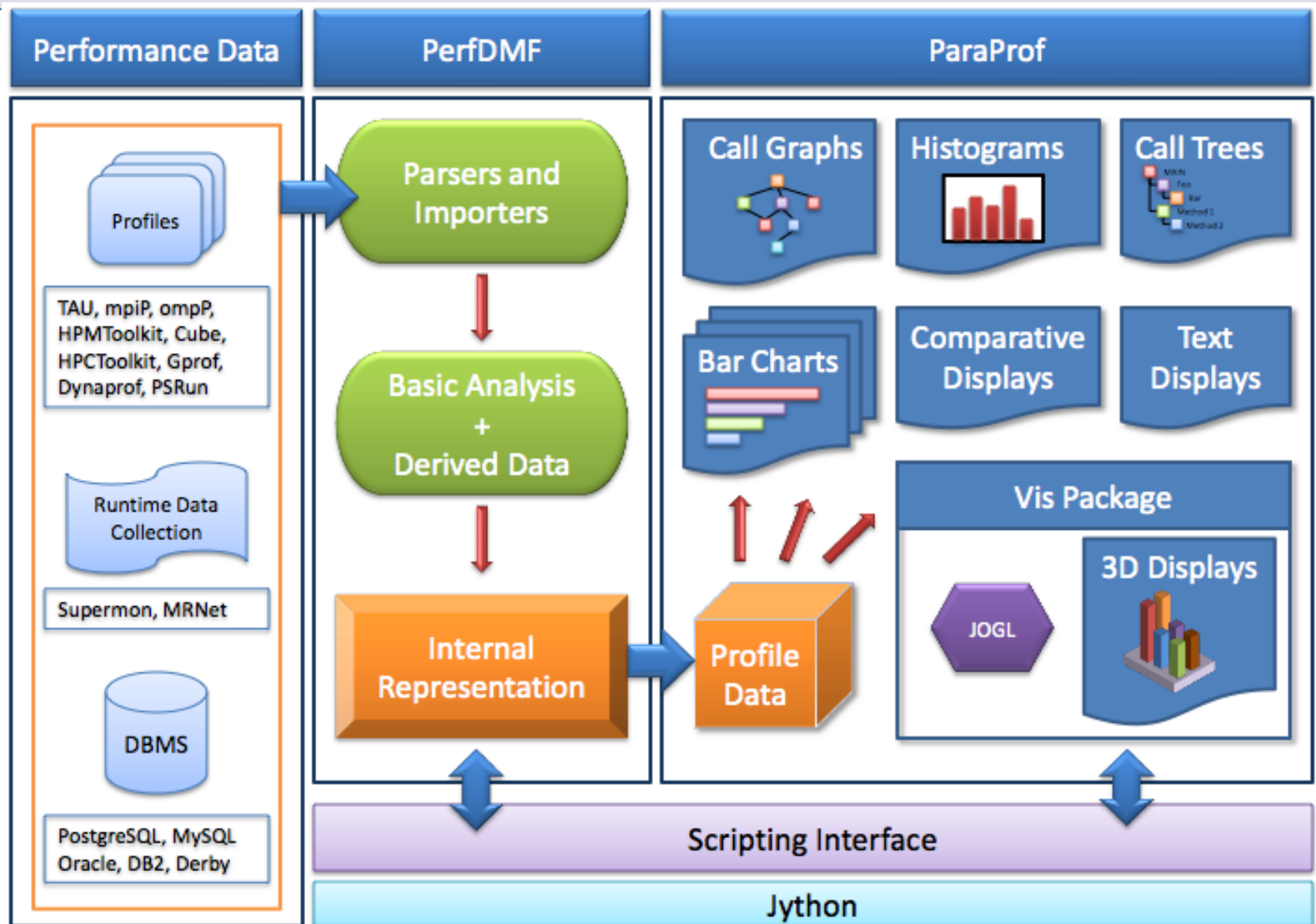
TAU Analysis



Performance Analysis

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

ParaProf Profile Analysis Framework



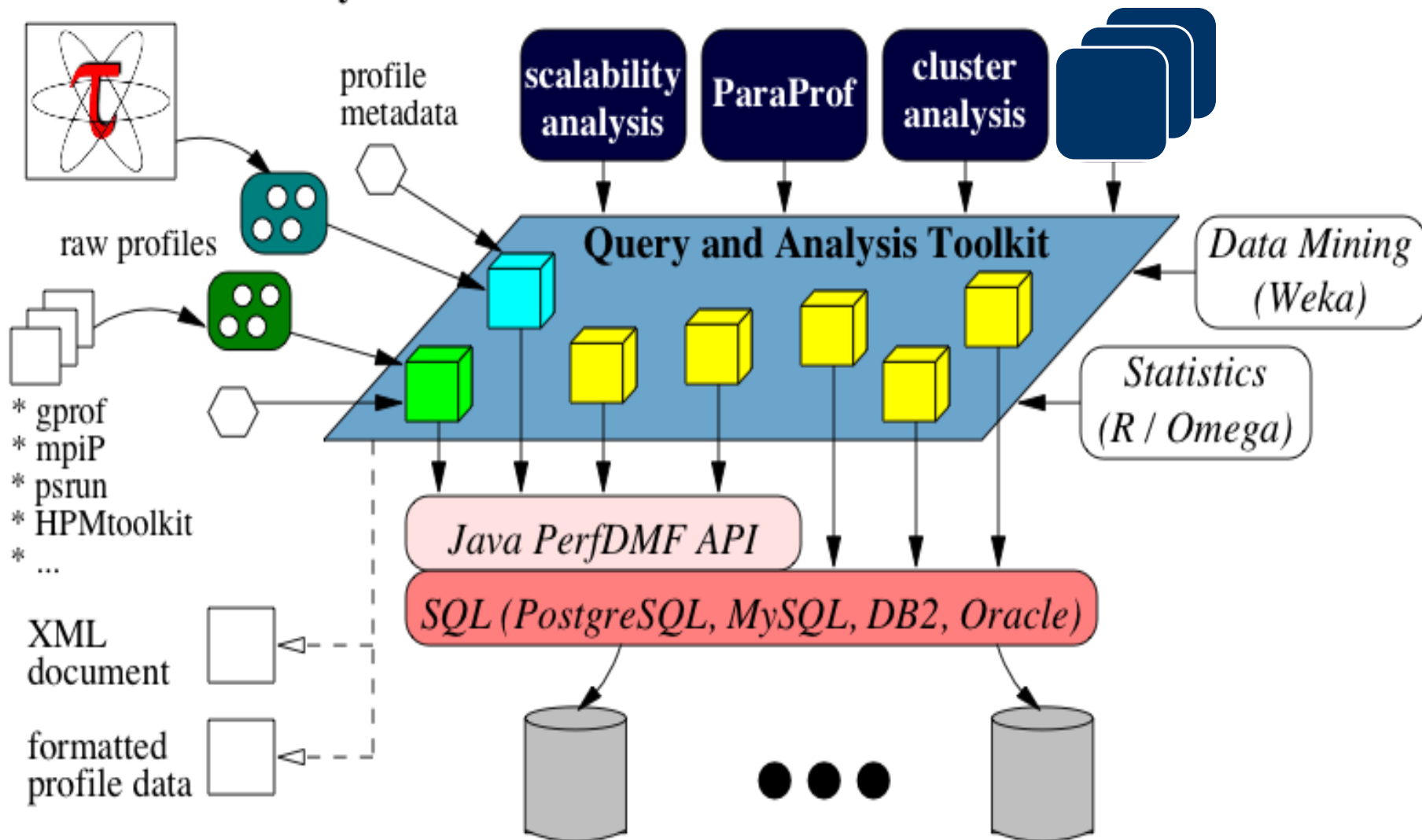
Performance Data Management

- Provide an open, flexible framework to support common data management tasks
 - Foster multi-experiment performance evaluation
- Extensible toolkit to promote integration and reuse across available performance tools (PerfDMF)
 - Originally designed to address critical TAU requirements
 - Supported profile formats:
TAU, CUBE (Scalasca), HPC Toolkit (Rice), HPM Toolkit (IBM), gprof, mpiP, psrun (PerfSuite), Open|SpeedShop, ...
 - Supported DBMS:
PostgreSQL, MySQL, Oracle, DB2, Derby/Cloudscape
 - Profile query and analysis API
- Reference implementation for PERI-DB project

PerfDMF Architecture

TAU Performance System

Performance Analysis Programs



Metadata Collection

- Integration of XML metadata for each parallel profile
- Three ways to incorporate metadata
 - Measured hardware/system information (TAU, PERI-DB)
 - CPU speed, memory in GB, MPI node IDs, ...
 - Application instrumentation (application-specific)
 - TAU_METADATA() used to insert any name/value pair
 - Application parameters, input data, domain decomposition
 - PerfDMF data management tools can incorporate an XML file of additional metadata
 - Compiler flags, submission scripts, input files, ...
- Metadata can be imported from / exported to PERI-DB

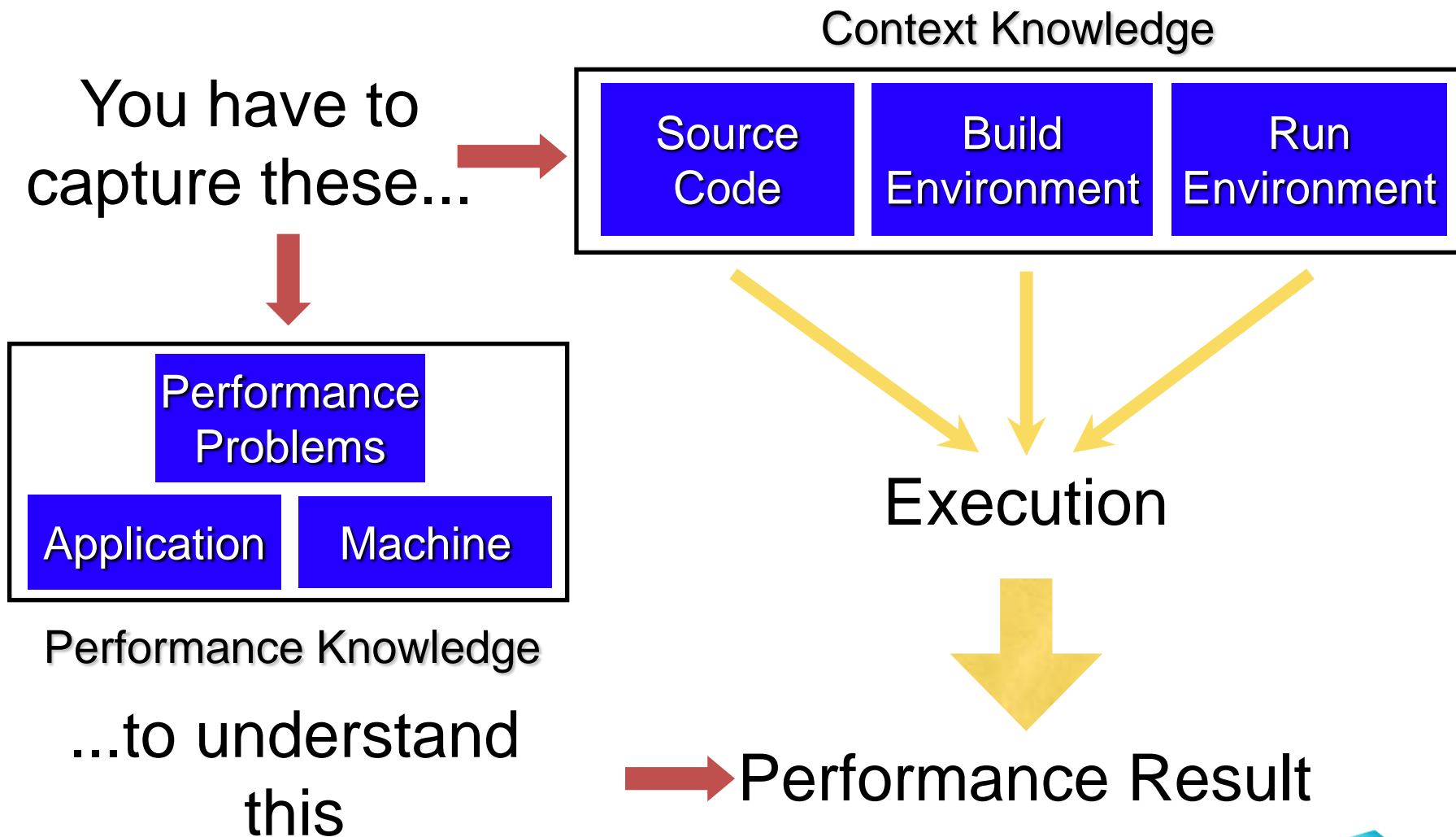
Performance Data Mining / Analytics

- Conduct systematic and scalable analysis process
 - Multi-experiment performance analysis
 - Support automation, collaboration, and reuse
- Performance knowledge discovery framework
 - Data mining analysis applied to parallel performance data
 - comparative, clustering, correlation, dimension reduction, ...
 - Use the existing TAU infrastructure
- PerfExplorer v1 performance data mining framework
 - Multiple experiments and parametric studies
 - Integrate available statistics and data mining packages
 - Weka, R, Matlab / Octave
 - Apply data mining operations in interactive environment

How to explain performance?

- Should not just redescribe the performance results
- Should explain performance phenomena
 - What are the causes for performance observed?
 - What are the factors and how do they interrelate?
 - Performance analytics, forensics, and decision support
- Need to add knowledge to do more intelligent things
 - Automated analysis needs good informed feedback
 - iterative tuning, performance regression testing
 - Performance model generation requires interpretation
- We need better methods and tools for
 - Integrating meta-information
 - Knowledge-based performance problem solving

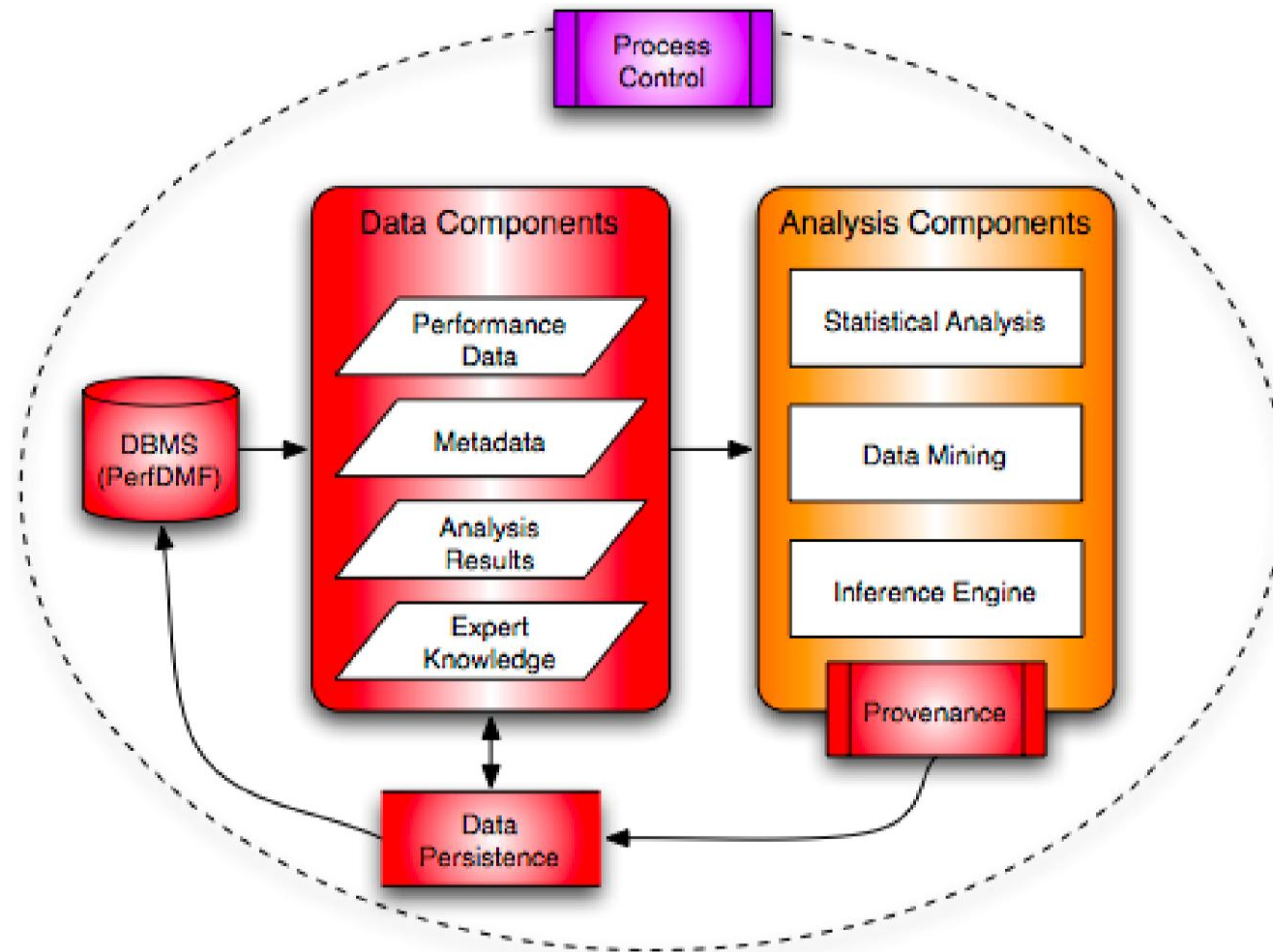
Role of Metadata and Knowledge Role



PerfExplorer v2 – Requirements

- Component-based analysis process
 - Analysis operations implemented as modules
 - Linked together in analysis process and workflow
- Scripting
 - Provides process/workflow development and automation
- Metadata input, management, and access
- Inference engine
 - Reasoning about causes of performance phenomena
 - Analysis knowledge captured in expert rules
- Persistence of intermediate analysis results
- Provenance
 - Provides historical record of analysis results

PerfExplorer v2 Architecture



Parallel Profile Analysis – pprof

```

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
-----
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       jaclcd
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-----

```


Metadata for Each Experiment

TAU: ParaProf Manager

File Options Help

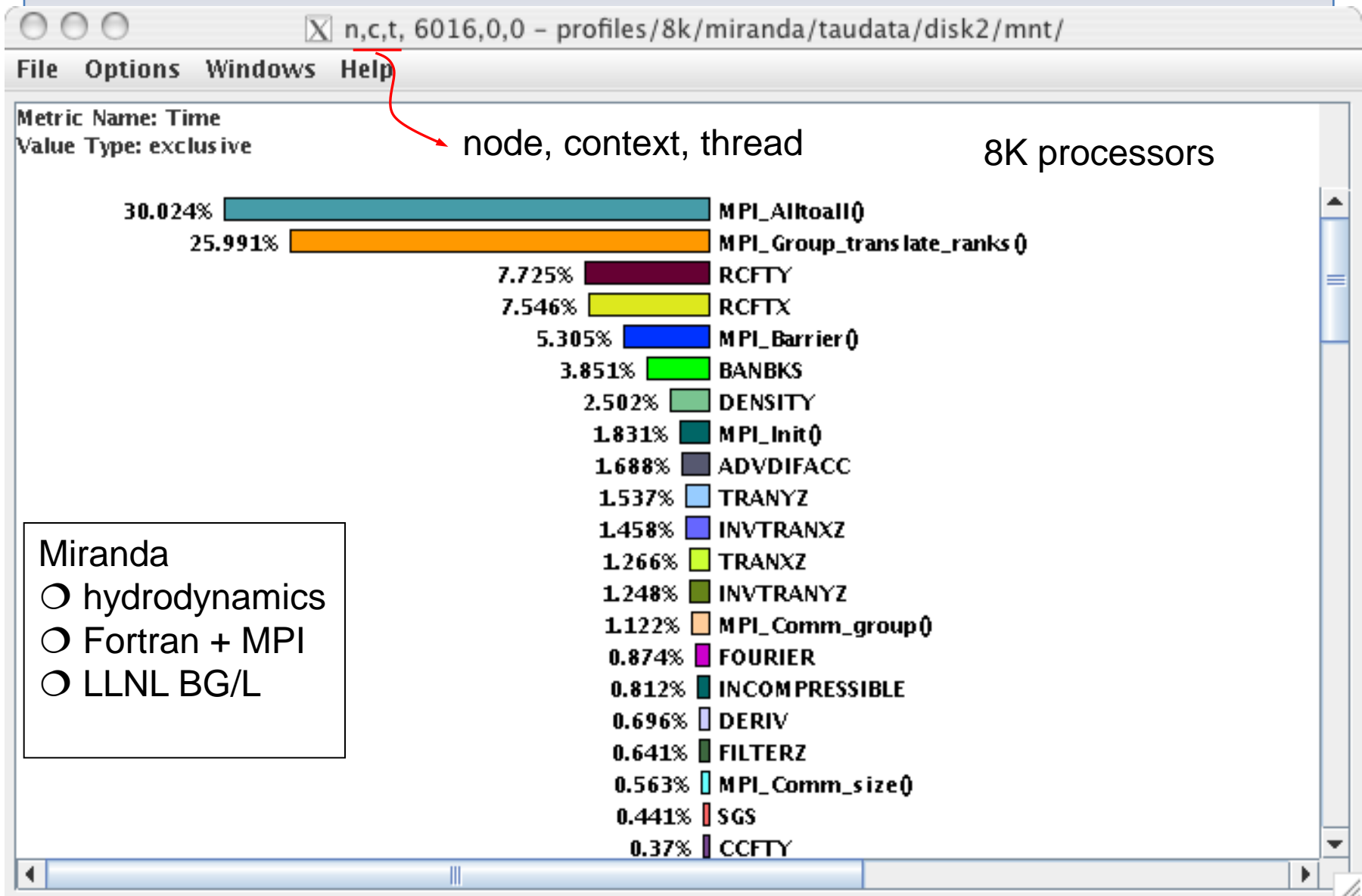
Applications

- Standard Applications
 - Default App
 - Default Exp
 - f90/pdt_mpi/examples/tau2/amorris/home/
 - PAPI_FP_OPS
 - GET_TIME_OF_DAY
- Default (jdbc:postgresql://spaceghost.cs.uoregon.edu:5432)
- utionium (jdbc:postgresql://utionium.cs.uoregon.edu:5432)
- spaceghost2 (jdbc:postgresql://spaceghost.cs.uoregon.edu:5432)
- proton_mysql (jdbc:mysql://192.168.1.1:3306/perfdm)
- spaceghost_peri_milc (jdbc:postgresql://spaceghost.cs.uoregon.edu:5432)
- proton_postgresql (jdbc:postgresql://192.168.1.1:5432)
- utionium_oracle (jdbc:oracle:thin:@//utionium.cs.uoregon.edu:1521)
- perijtc (jdbc:postgresql://spaceghost.cs.uoregon.edu:5432)

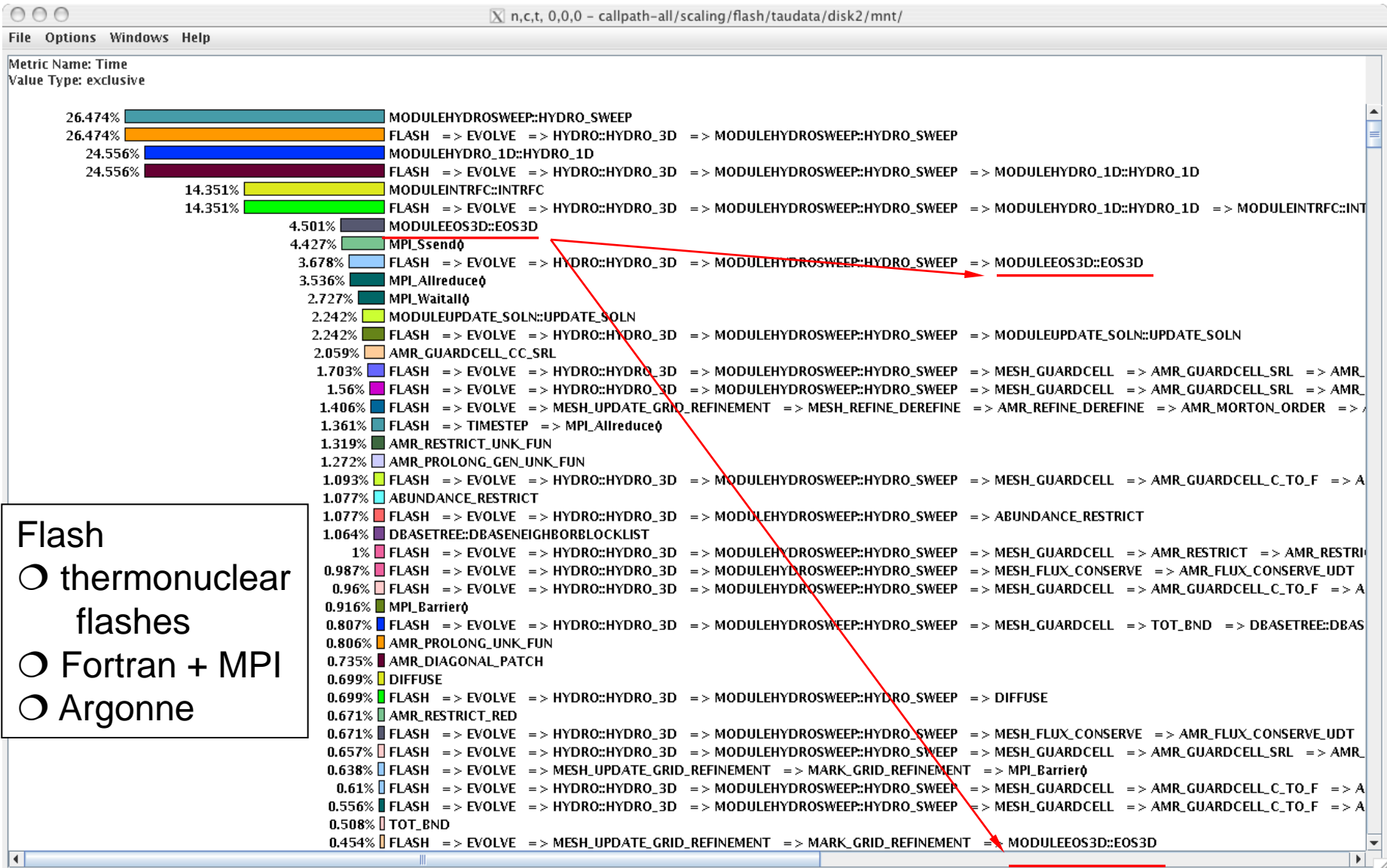
Multiple PerfDMF DBs

TrialField	Value
Name	f90/pdt_mpi/examples/tau2/amorris/home/
Application ID	0
Experiment ID	0
Trial ID	0
CPU Cores	2
CPU MHz	2992.505
CPU Type	Intel(R) Xeon(R) CPU 5160 @ 3.00GHz
CPU Vendor	GenuineIntel
CWD	/home/amorris/tau2/examples/pdt_mpi/f90
Cache Size	4096 KB
Executable	/home/amorris/tau2/examples/pdt_mpi/f...
Hostname	demon.nic.uoregon.edu
Local Time	2007-07-04T04:21:14-07:00
MPI Processor Name	demon.nic.uoregon.edu
Memory Size	8161240 KB
Node Name	demon.nic.uoregon.edu
OS Machine	x86_64
OS Name	Linux
OS Release	2.6.9-42.0.3.EL.perfctrsm
OS Version	#1 SMP Fri Nov 3 07:34:13 PST 2006
Starting Timestamp	1183548072220996
TAU Architecture	x86_64
TAU Config	-papi=/usr/local/packages/papi-3.5.0 -M...
Timestamp	1183548074317538
UTC Time	2007-07-04T11:21:14Z
pid	11395
username	amorris

ParaProf – Flat Profile



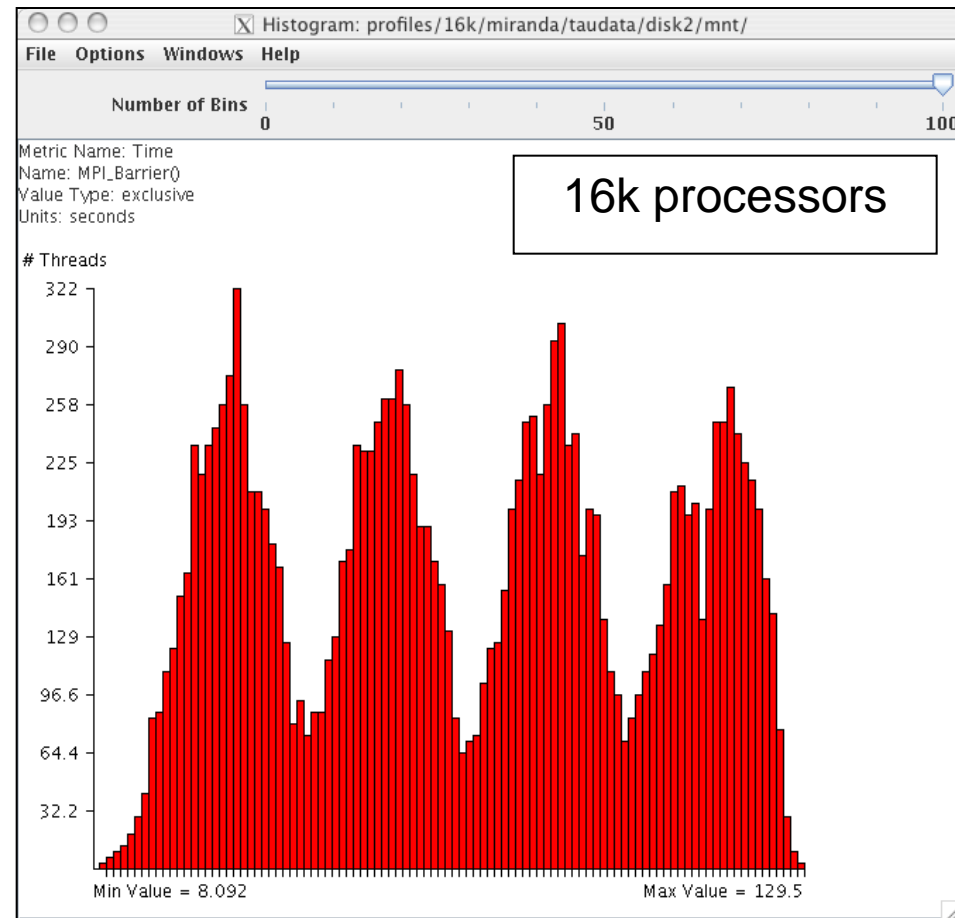
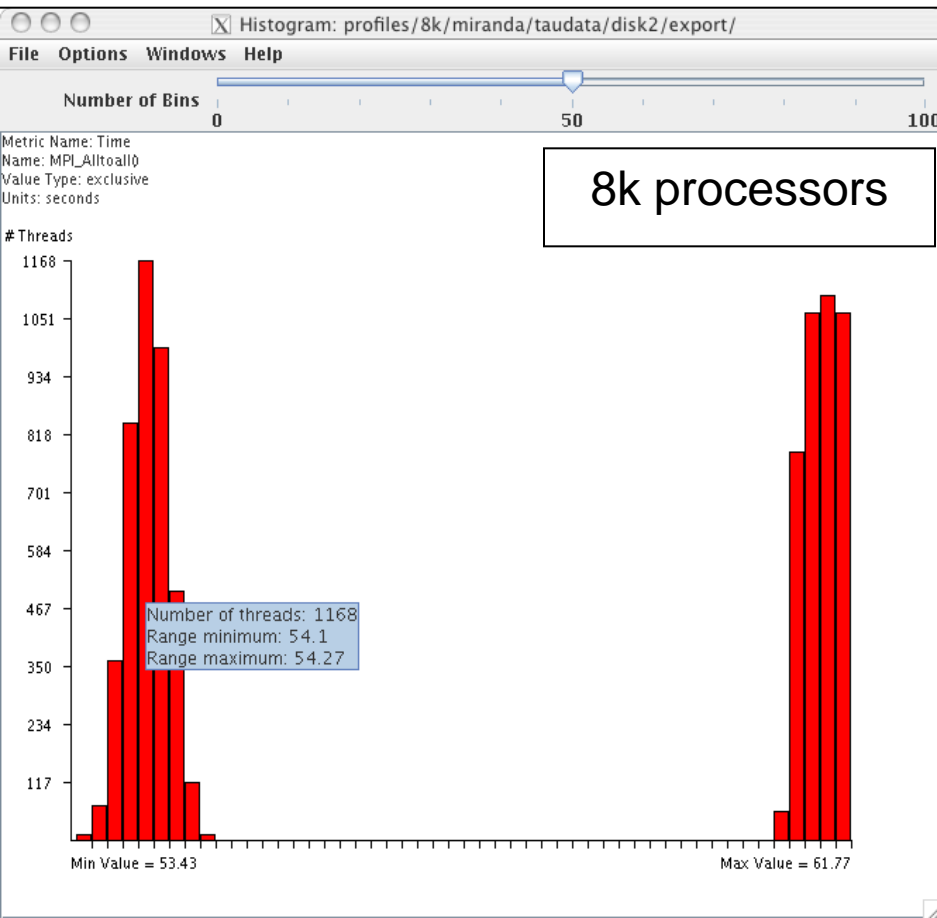
ParaProf – Callpath Profile



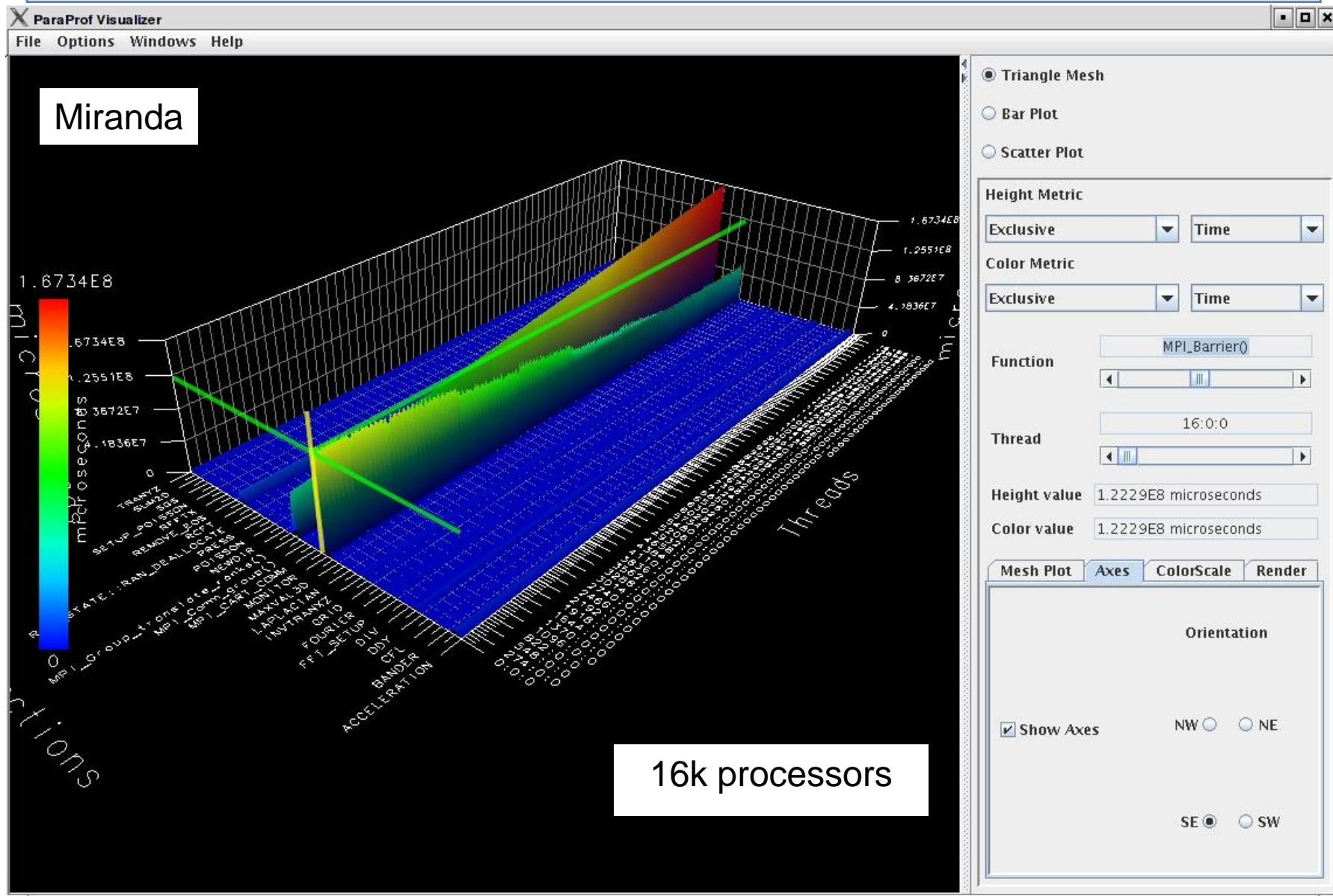
Flash

- thermonuclear flashes
- Fortran + MPI
- Argonne

ParaProf – Scalable Histogram

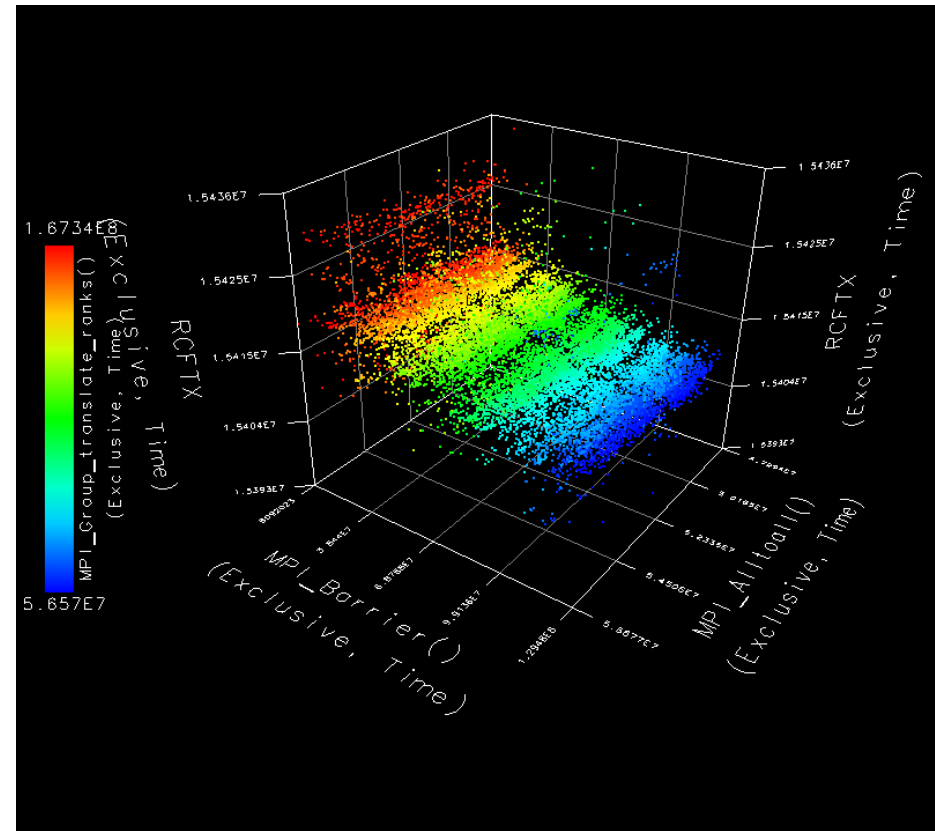


ParaProf – 3D View (Full Profile)



ParaProf – 3D Scatterplot

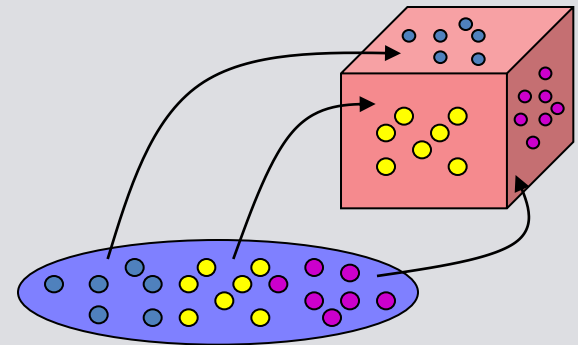
- Each point is a “thread” of execution
- A total of four metrics shown in relation
- ParaProf’s visualization library
 - JOGL
- Miranda



Performance Mapping

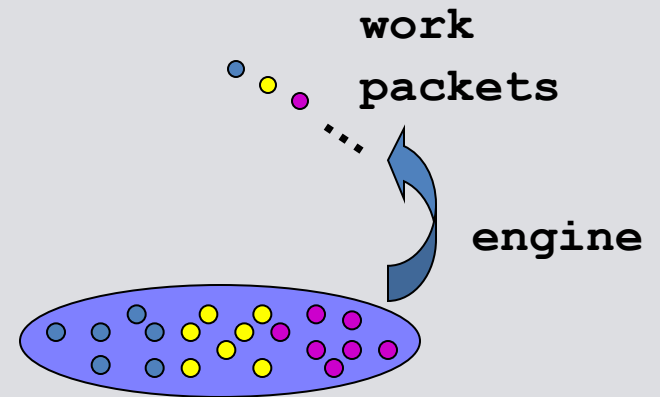
- Example: Particles distributed on cube surface

```
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;
    }
}
```



Performance Mapping

```
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]);  
}
```

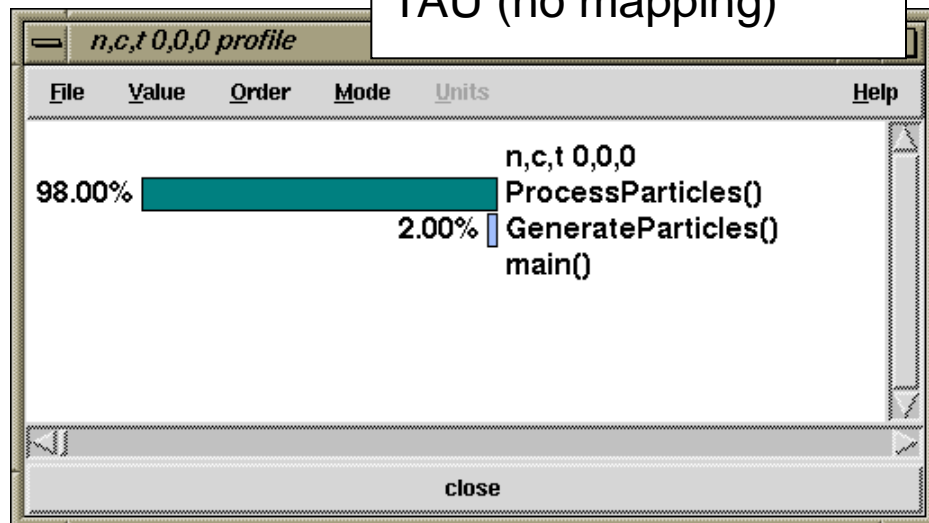


- How much time (flops) spent processing face i particles?
- What is the distribution of performance among faces?

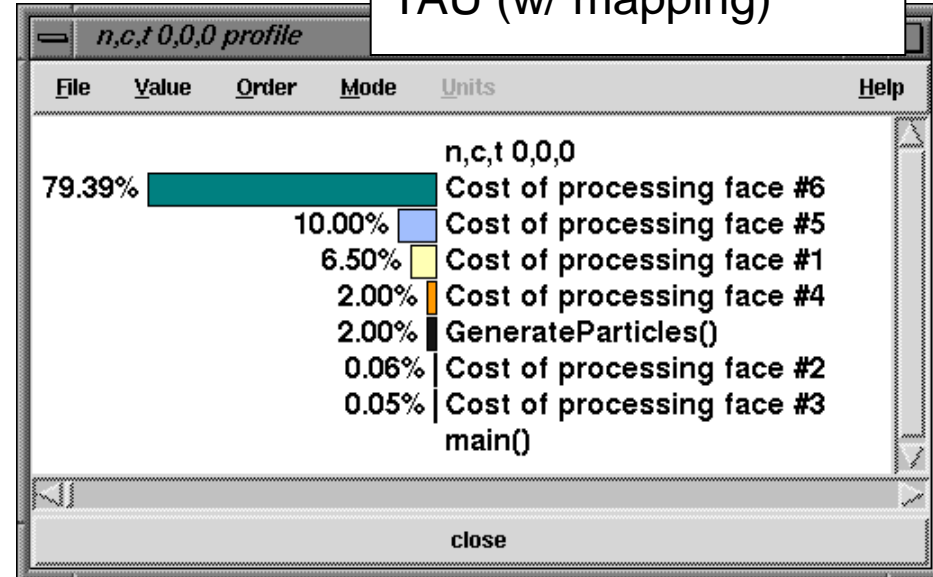
No Mapping versus Mapping

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

TAU (no mapping)



TAU (w/ mapping)



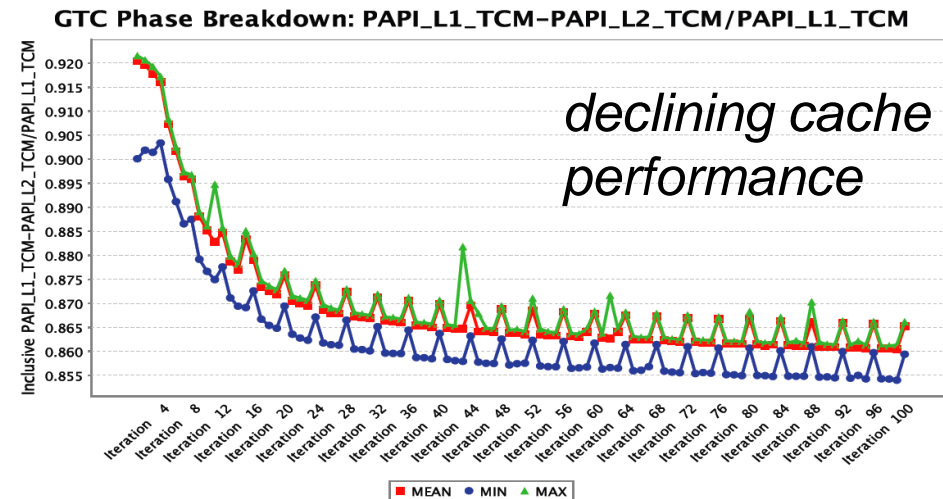
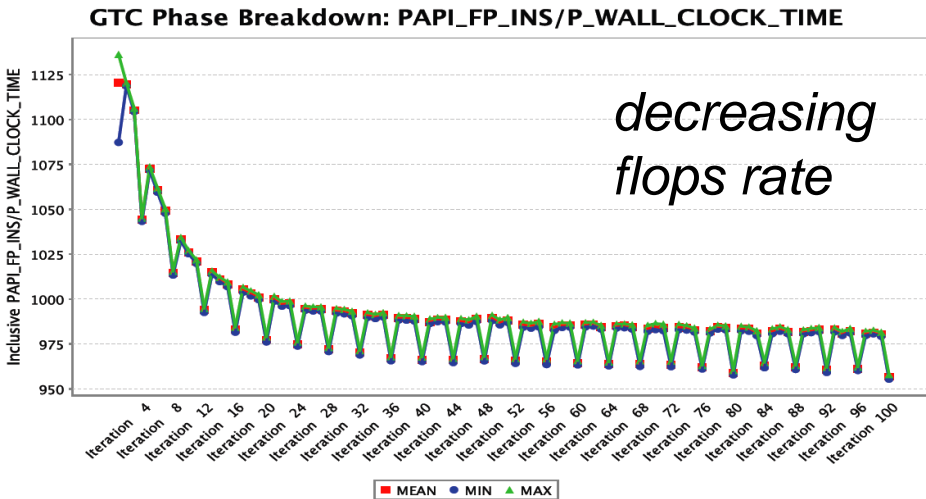
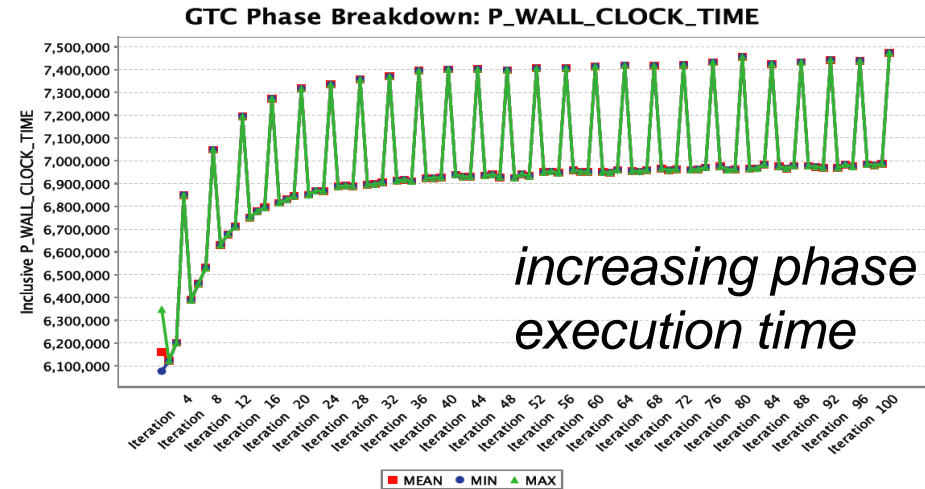
NAS BT – Phase Profile

Main phase shows nested phases and immediate events



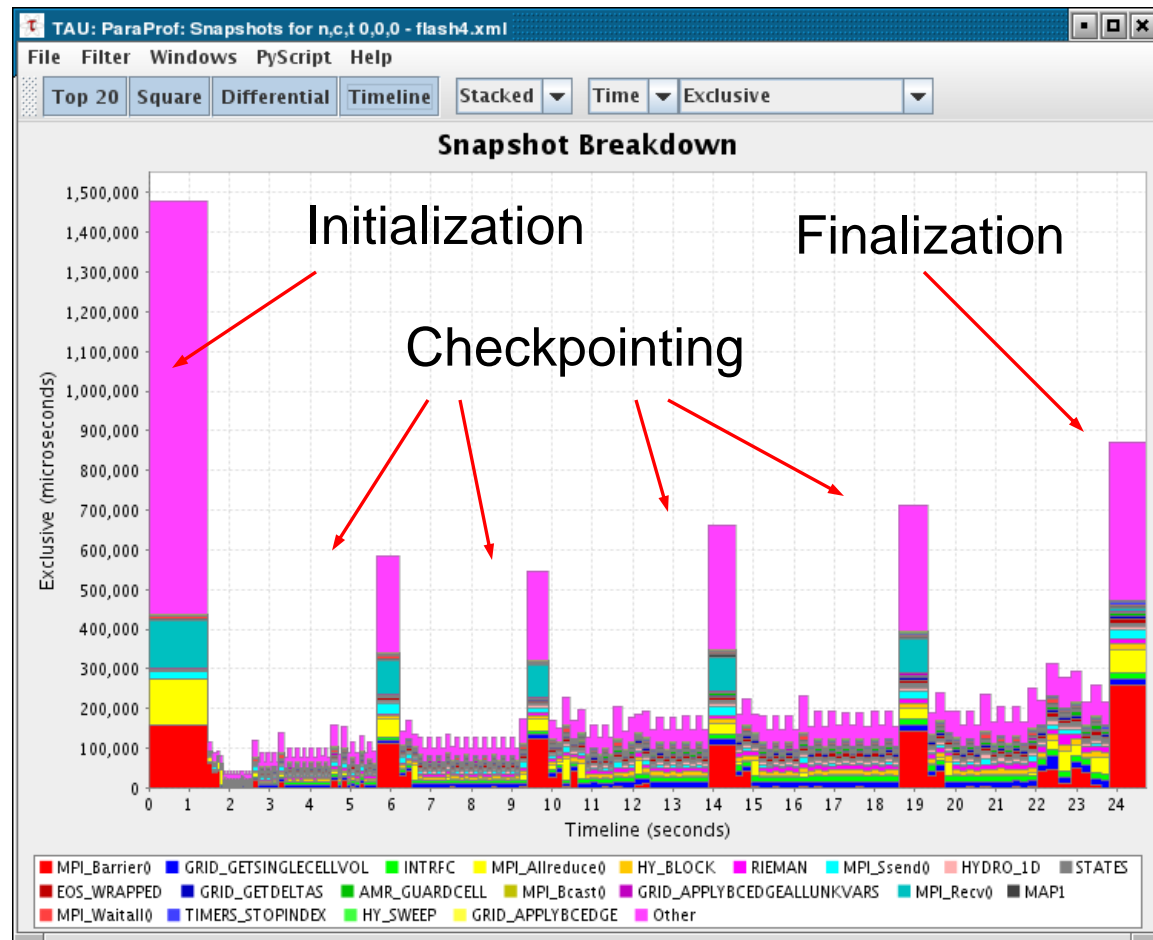
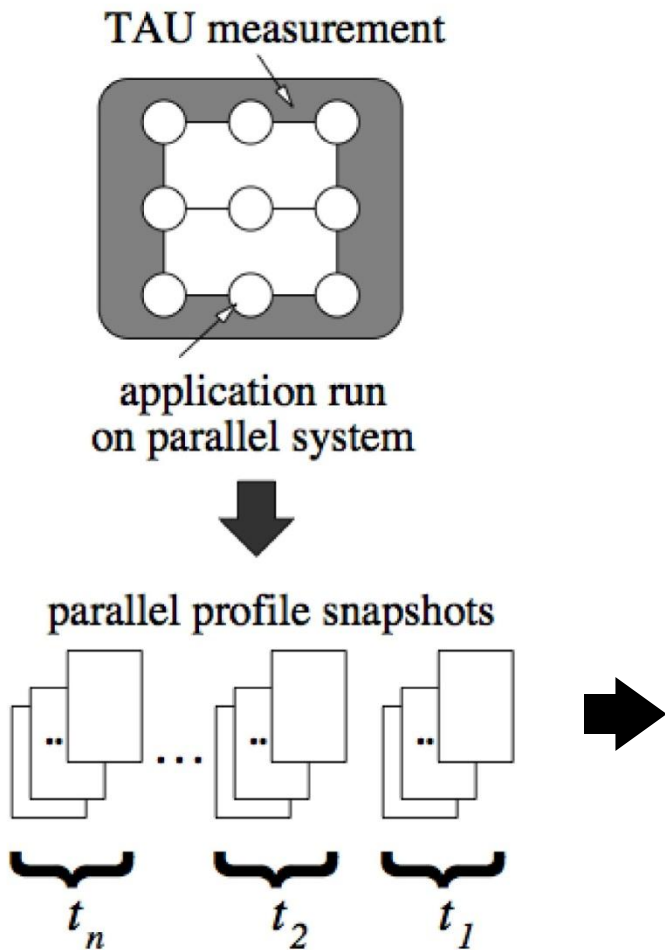
Phase Profiling of HW Counters

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



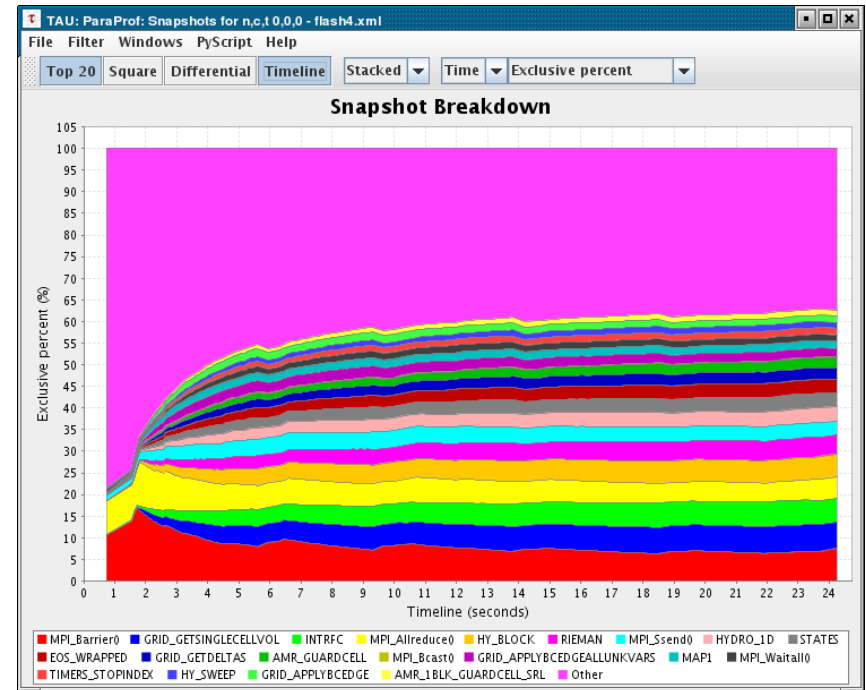
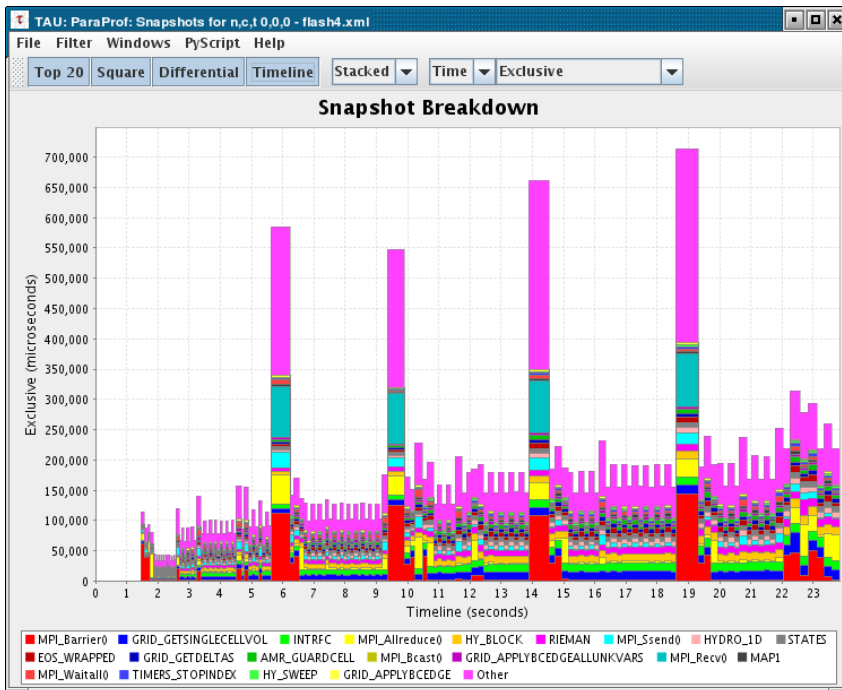
Profile Snapshots in ParaProf

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

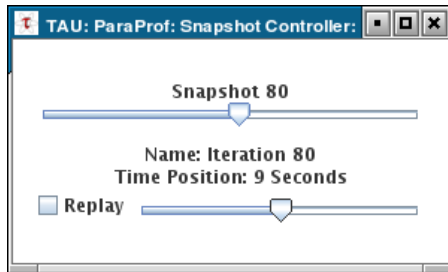


Profile Snapshot Views

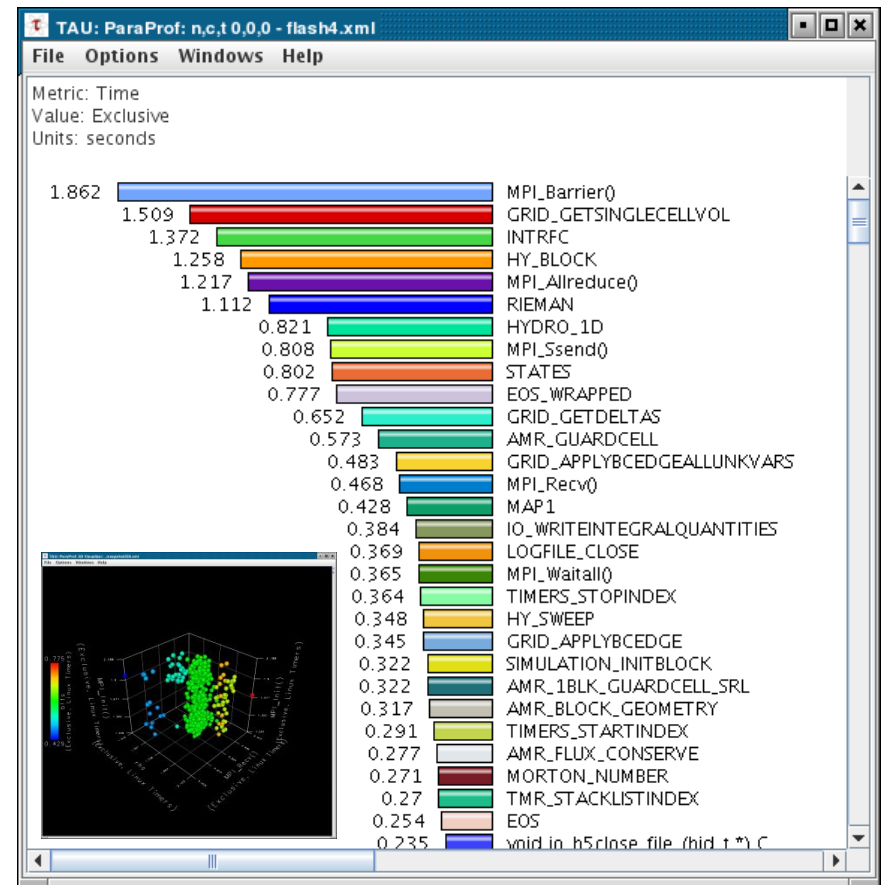
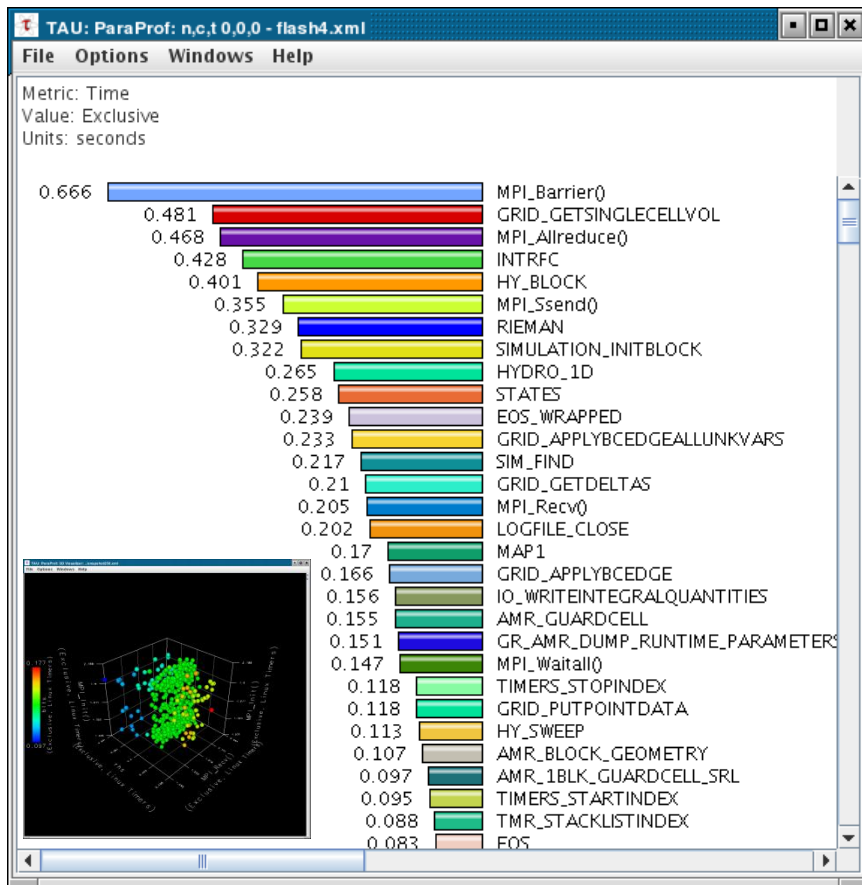
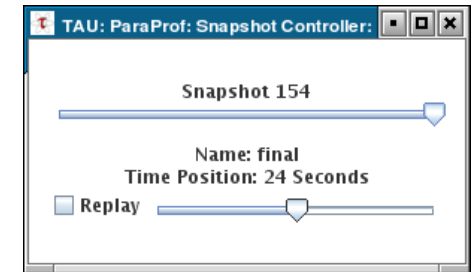
- Only show main loop
- Percentage breakdown



Snapshot Replay in ParaProf

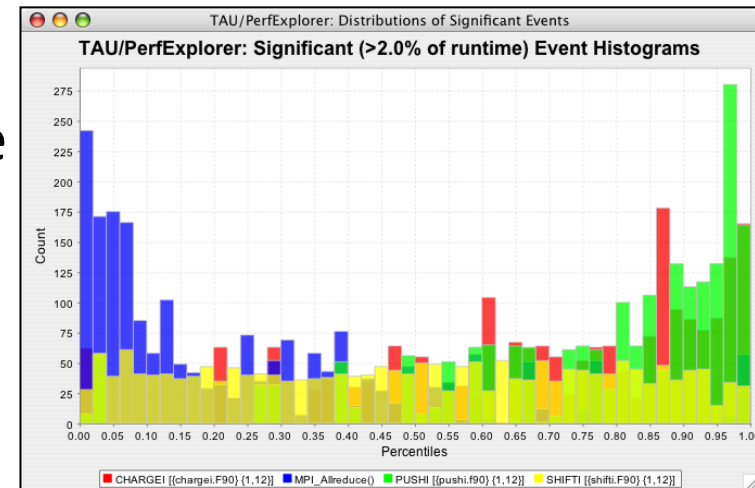
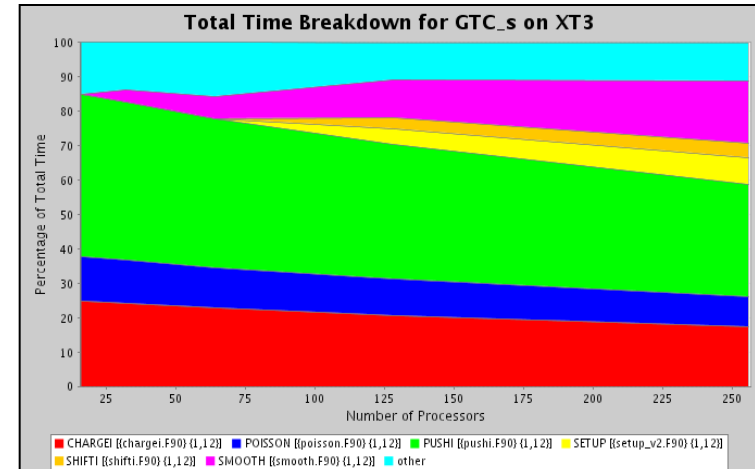


All windows dynamically update

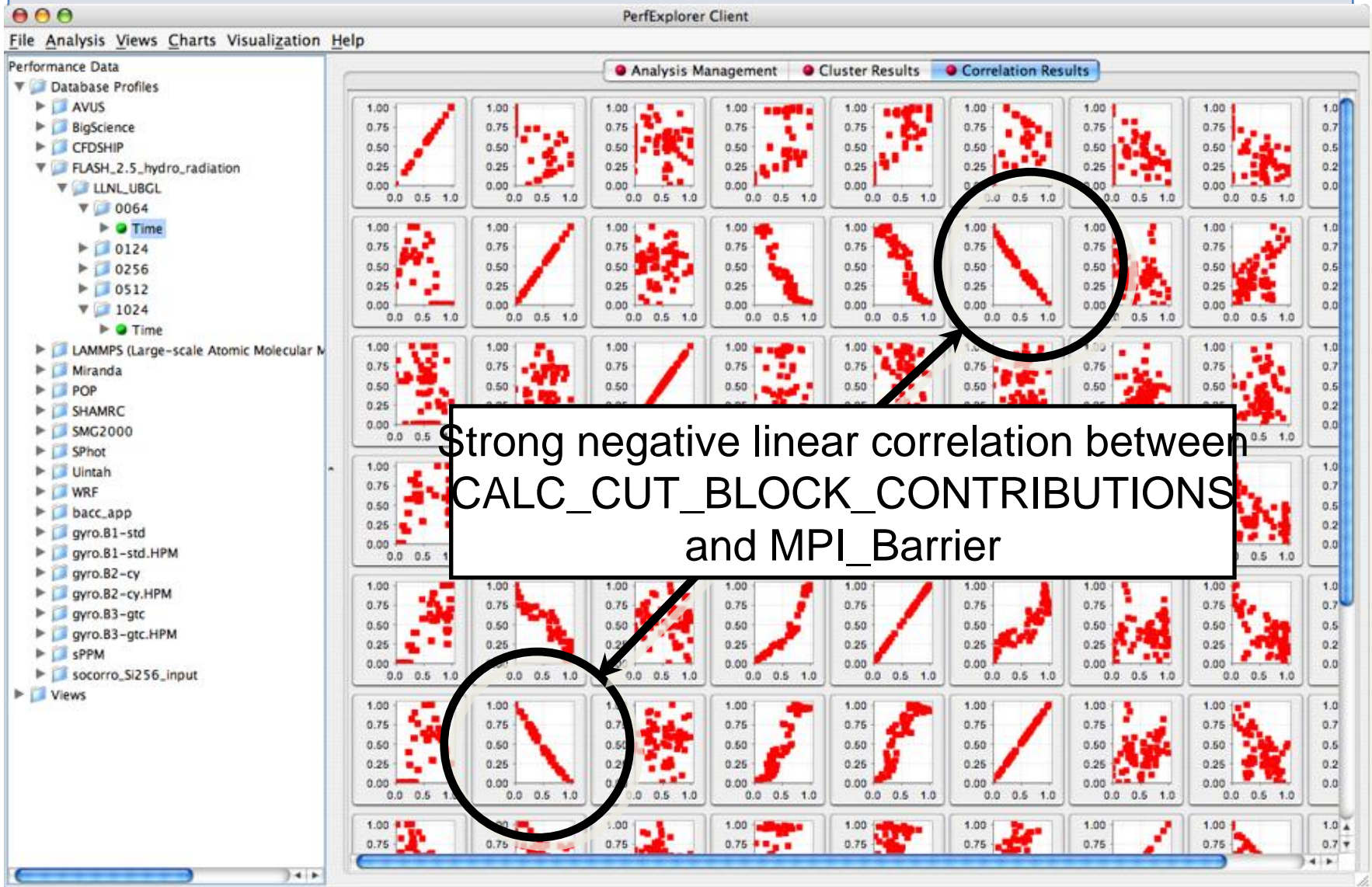


PerfExplorer – Relative Comparisons

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

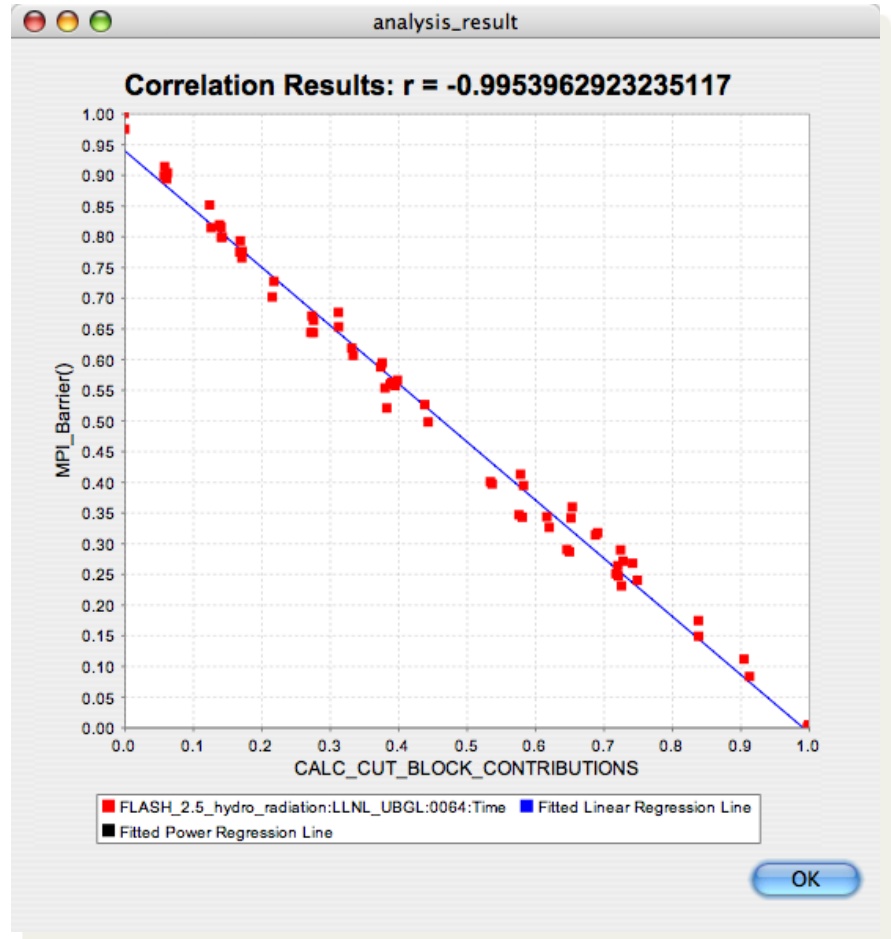


PerfExplorer – Correlation Analysis

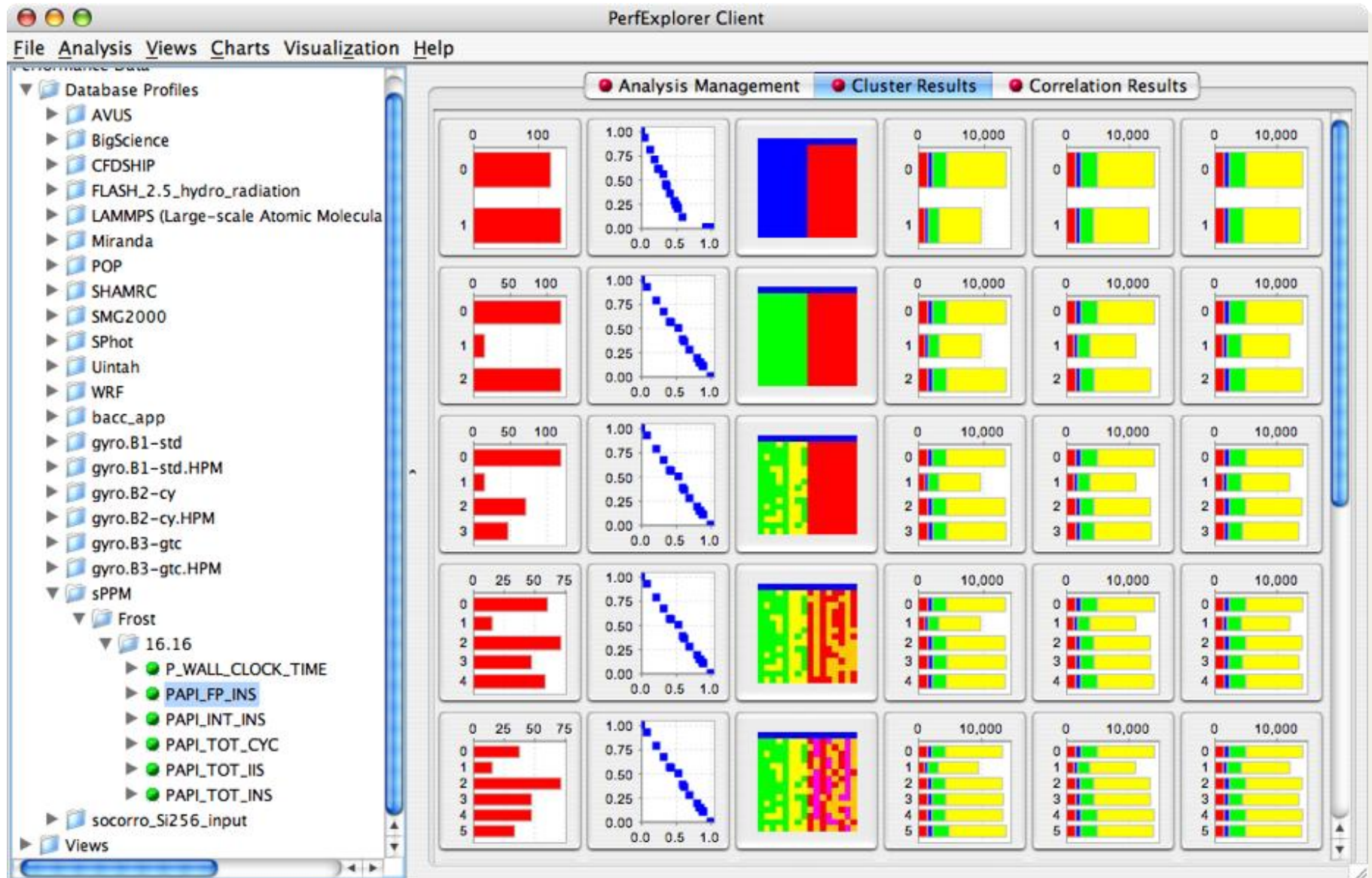


PerfExplorer – Correlation Analysis

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

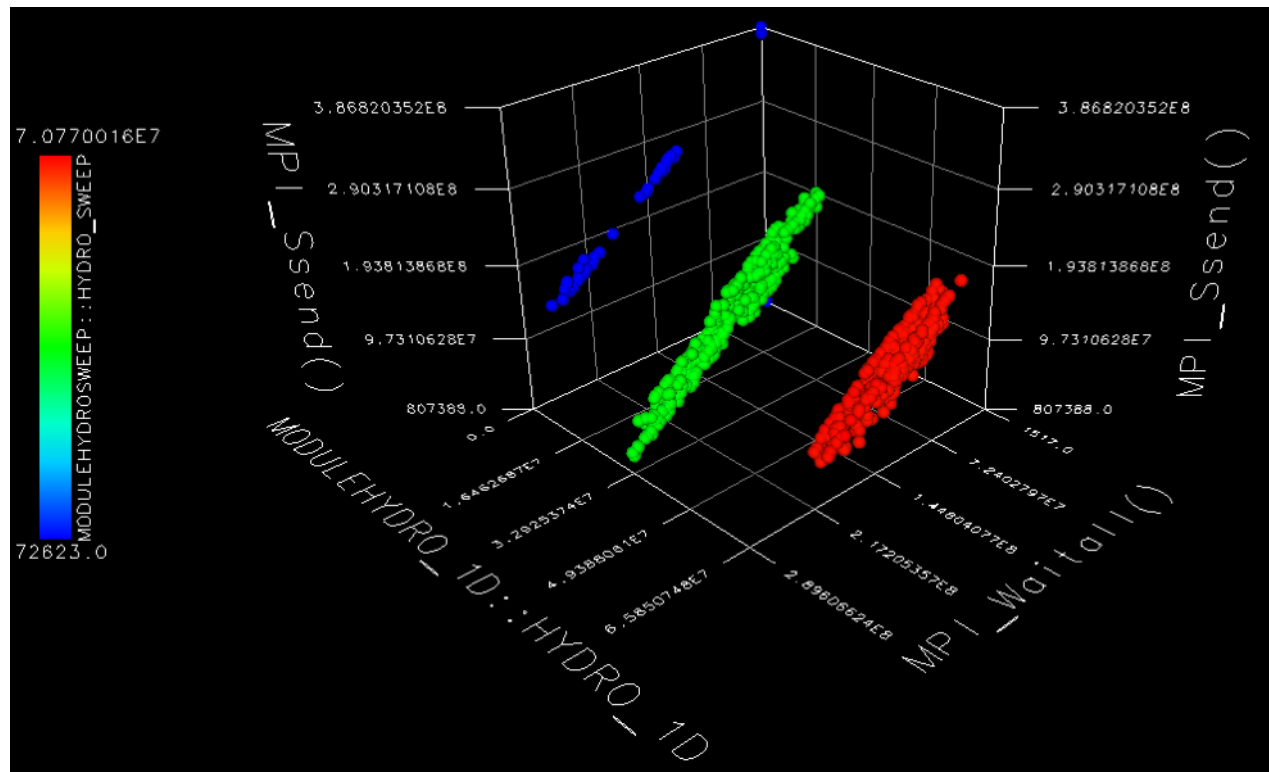


PerfExplorer – Cluster Analysis

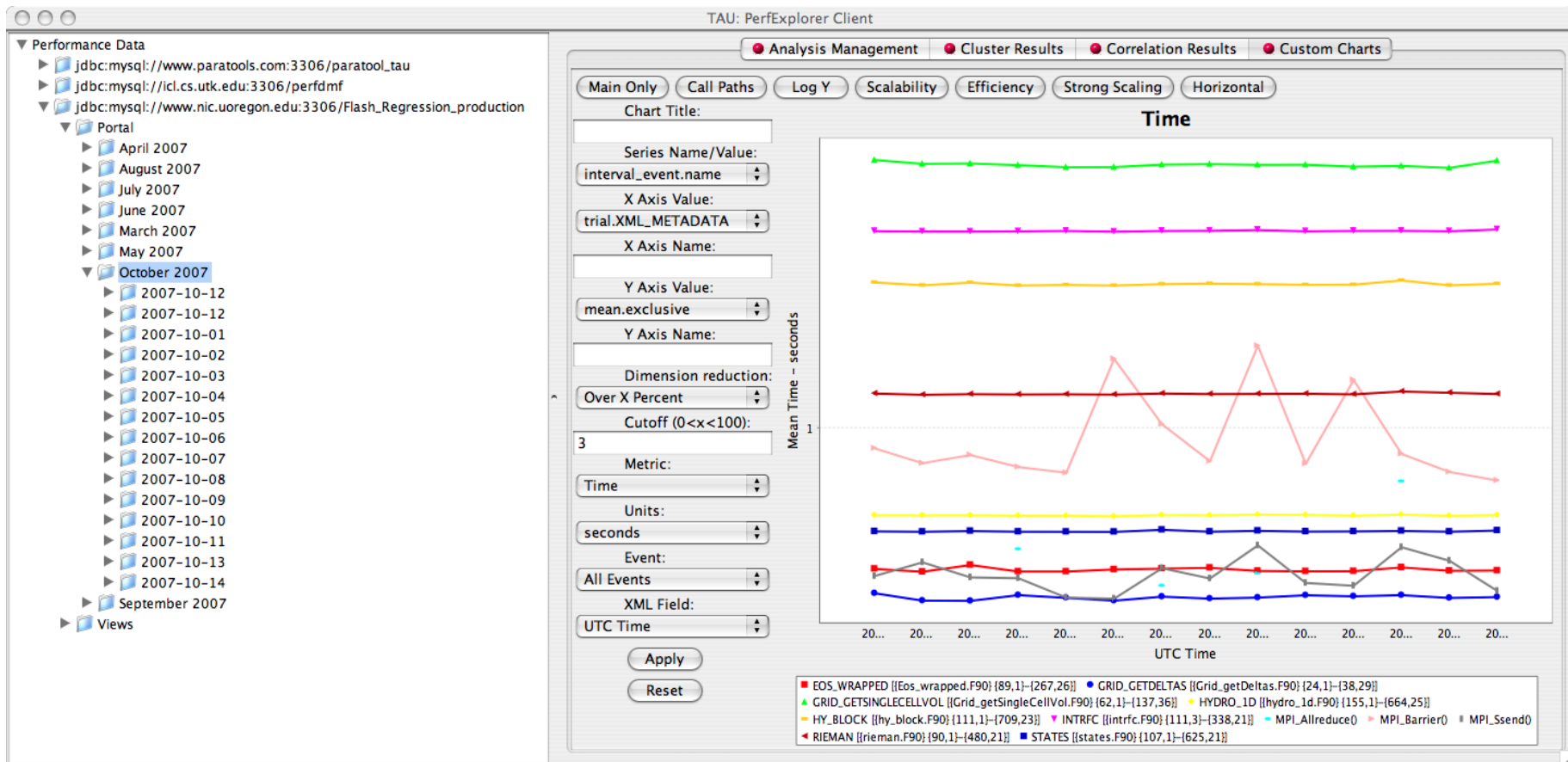


PerfExplorer – Cluster Analysis

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



PerfExplorer – Performance Regression



Other Projects in TAU

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

Using TAU

- Install TAU
 - % configure [options]; make clean install
- Modify application makefile and choose TAU configuration
 - Select TAU's stub makefile
 - Change name of compiler in makefile
- Set environment variables
 - Directory where profiles/traces are to be stored/counter selection
 - TAU options
- Execute application
 - % mpirun -np <procs> a.out;
- Analyze performance data
 - paraprof, vampir, pprof, paraver ...

Application Build Environment

- Minimize impact on user's application build procedures
- Handle parsing, instrumentation, compilation, linking
- Dealing with Makefiles
 - Minimal change to application Makefile
 - Avoid changing compilation rules in application Makefile
 - No explicit inclusion of rules for process stages
- Some applications do not use Makefiles
 - Facilitate integration in whatever procedures used
- Two techniques:
 - TAU shell scripts (tau_<compiler>.sh)
 - Invokes all PDT parser, TAU instrumenter, and compiler
 - TAU_COMPILER

Configuring TAU

- TAU can measure several metrics with profiling and tracing approaches
- Different tools can also be invoked to instrument programs for TAU measurement
- Each configuration of TAU produces a measurement library for an architecture
- Each measurement configuration of TAU also creates a corresponding stub makefile that can be used to compile programs
- Typically configure multiple measurement libraries

TAU Measurement System Configuration

- **configure [OPTIONS]**
 - {-c++=<CC>, -cc=<cc>} Specify C++ and C compilers
 - -pdt=<dir> Specify location of PDT
 - -opari=<dir> Specify location of Opari OpenMP tool
 - -papi=<dir> Specify location of PAPI
 - -vampirtrace=<dir> Specify location of VampirTrace
 - -mpi[inc/lib]=<dir> Specify MPI library instrumentation
 - -dyninst=<dir> Specify location of DynInst Package
 - -shmem[inc/lib]=<dir> Specify PSHMEM library instrumentation
 - -python[inc/lib]=<dir> Specify Python instrumentation
 - -tag=<name> Specify a unique configuration name
 - -epilog=<dir> Specify location of EPILOG
 - -slog2 Build SLOG2/Jumpshot tracing package
 - -otf=<dir> Specify location of OTF trace package
 - -arch=<architecture> Specify architecture explicitly
(bgl, xt3, ibm64, ibm64linux...)
 - {-pthread, -sproc} Use pthread or SGI sproc threads
 - -openmp Use OpenMP threads
 - -jdk=<dir> Specify Java instrumentation (JDK)
 - -fortran=[vendor] Specify Fortran compiler

TAU Measurement System Configuration

- configure [OPTIONS]
 - -TRACE Generate binary TAU traces
 - -PROFILE (default) Generate profiles (summary)
 - -PROFILECALLPATH Generate call path profiles
 - -PROFILEPHASE Generate phase based profiles
 - -PROFILEMEMORY Track heap memory for each routine
 - -PROFILEHEADROOM Track memory headroom to grow
 - -MULTIPLECOUNTERS Use hardware counters + time
 - -COMPENSATE Compensate timer overhead
 - -CPUTIME Use usertime+system time
 - -PAPIWALLCLOCK Use PAPI's wallclock time
 - -PAPIVIRTUAL Use PAPI's process virtual time
 - -SGITIMERS Use fast IRIX timers
 - -LINUXTIMERS Use fast x86 Linux timers

TAU Configuration – Examples

- Configure using PDT and MPI for x86_64 Linux

```
./configure --pdt=/usr/pkg/pkg/pdtoolkit-3.14
  -mpiinc=/usr/pkg/mpich/include -mpilib=/usr/pkg/mpich/lib
  -mpilibrary='-lmpich -L/usr/gm/lib64 -lgm -lpthread -ldl'
```
- Use PAPI counters (one or more) with C/C++/F90 automatic instrumentation for Cray CNL. Also instrument the MPI library. Use PGI compilers.

```
./configure -arch=craycnl -cc=cc -c++=CC -fortran=pgi -papi=/opt/xt-
  tools/papi/3.2.1 -mpi -MULTIPLECOUNTERS; make clean install
```
- Stub makefiles

```
/usr/pkg/tau/x86_64/lib/Makefile.tau-mpi-pdt-pgi
/usr/pkg/tau/x86_64/lib/Makefile.tau-multiplecounters-
  mpi-papi-pdt-pgi
```

Stub Makefiles Configuration Parameters

- TAU scripts use stub makefiles to select performance measurements
- Variables:
 - TAU_CXX Specify the C++ compiler used by TAU
 - TAU_CC, TAU_F90 Specify the C, F90 compilers
 - TAU_DEFS Defines used by TAU (add to CFLAGS)
 - TAU_LDFLAGS Linker options (add to LDFLAGS)
 - TAU_INCLUDE Header files include path (add to CFLAGS)
 - TAU_LIBS Statically linked TAU library (add to LIBS)
 - TAU_SHLIBS Dynamically linked TAU library
 - TAU_MPI_LIBS TAU's MPI wrapper library for C/C++
 - TAU_MPI_FLIBS TAU's MPI wrapper library for F90
 - TAU_FORTRANLIBS Must be linked in with C++ linker for F90
 - TAU_CXXLIBS Must be linked in with F90 linker
 - TAU_INCLUDE_MEMORY Use TAU's malloc/free wrapper lib
 - TAU_DISABLE TAU's dummy F90 stub library
 - TAU_COMPILER Instrument using tau_compiler.sh script

TAU Measurement Configuration

- `% cd /opt/tau-2.18/x86_64/lib; ls Makefile.*`
 - `Makefile.tau-pdt`
 - `Makefile.tau-mpi-pdt`
 - `Makefile.tau-callpath-mpi-pdt`
 - `Makefile.tau-mpi-pdt-trace`
 - `Makefile.tau-mpi-compensate-pdt`
 - `Makefile.tau-multiplecounters-mpi-papi-pdt`
 - `Makefile.tau-multiplecounters-mpi-papi-pdt-trace`
 - `Makefile.tau-pthread-pdt...`
- For an MPI+F90 application, you may want to start with:
 - `Makefile.tau-mpi-pdt`
 - Supports MPI instrumentation & PDT for automatic source instrumentation
- `% setenv TAU_MAKEFILE
/opt/tau-2.18/x86_64/lib/Makefile.tau-mpi-pdt`

-PROFILE Option

- Generates flat profiles
 - One for each MPI process
 - It is the default option.
- Uses wallclock time
 - `gettimeofday()` sys call
- Calculates exclusive, inclusive time spent in each timer and number of calls

Generating a Flat Profile with MPI

```
% setenv TAU_MAKEFILE /opt/tau-2.18/x86_64
                        /lib/Makefile.tau-mpi-pdt
% set path=(/opt/tau-2.18/x86_64/bin $path)
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)

% qsub run.job
% paraprof --pack app.ppk
Move the app.ppk file to your desktop.

% paraprof app.ppk
```

Generating a Loop-level Profile

```
% setenv TAU_MAKEFILE /opt/tau-2.18/x86_64
                               /lib/Makefile.tau-mpi-pdt
% setenv TAU_OPTIONS '-optTauSelectFile=select.tau -optVerbose'
% cat select.tau
BEGIN_INSTRUMENT_SECTION
loops routine="#"
END_INSTRUMENT_SECTION

% set path=(/opt/tau-2.18/x86_64/bin $path)
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% qsub run.job
% paraprof --pack app.ppk
Move the app.ppk file to your desktop.

% paraprof app.ppk
```

Compiler-based Instrumentation

```
% setenv TAU_MAKEFILE /opt/tau-2.18/x86_64
                        /lib/Makefile.tau-mpi
% setenv TAU_OPTIONS '-optCompInst -optVerbose'
% % set path=(/opt/tau-2.18/x86_64/bin $path)
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)

% qsub run.job
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.
% paraprof app.ppk
```

-MULTIPLECOUNTERS Option

- Instead of one metric, profile or trace with more than one metric
 - Set environment variables COUNTER[1-25] to specify the metric
 - % setenv COUNTER1 GET_TIME_OF_DAY
 - % setenv COUNTER2 PAPI_L2_DCM
 - % setenv COUNTER3 PAPI_FP_OPS
 - % setenv COUNTER4 PAPI_NATIVE_<native_event>
 - % setenv COUNTER5 P_WALL_CLOCK_TIME ...
- When used with –TRACE option, the first counter must be GET_TIME_OF_DAY
 - % setenv COUNTER1 GET_TIME_OF_DAY
 - Provides a globally synchronized real time clock for tracing
- -multiplecounters appears in the name of the stub Makefile
- Often used with –papi=<dir> to measure hardware performance counters and time
- papi_native_avail and papi_avail are two useful tools

Generate a PAPI profile

```
% setenv TAU_MAKEFILE /opt/tau-2.18/x86_64
                        /lib/Makefile.tau-multiplecounters-papi-mpi-pdt
% setenv TAU_OPTIONS '-optTauSelectFile=select.tau -optVerbose'
% cat select.tau
BEGIN_INSTRUMENT_SECTION
loops routine="#"
END_INSTRUMENT_SECTION

% set path=(/opt/tau-2.18/x86_64/bin $path)
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% setenv COUNTER1 GET_TIME_OF_DAY
% setenv COUNTER2 PAPI_FP_INS
% qsub run.job
% paraprof --pack app.ppk
Move the app.ppk file to your desktop.
% paraprof app.ppk
Choose Options -> Show Derived Panel -> Arg 1 = PAPI_FP_INS,
Arg 2 = GET_TIME_OF_DAY, Operation = Divide -> Apply, choose.
```

-PROFILECALLPATH Option

- Generates profiles that show the calling order (edges and nodes in callgraph)
 - A=>B=>C shows the time spent in C when it was called by B and B was called by A
 - Control the depth of callpath using TAU_CALLPATH_DEPTH environment variable
 - -callpath in the name of the stub Makefile name or setting TAU_CALLPATH= 1 at runtime (TAU v2.18.1+)

-DEPTHLIMIT Option

- Allows users to enable instrumentation at runtime based on the depth of a calling routine on a callstack
 - Disables instrumentation in all routines a certain depth away from the root in a callgraph
- TAU_DEPTH_LIMIT environment variable specifies depth
 - % setenv TAU_DEPTH_LIMIT 1
 - enables instrumentation in only “main”
 - % setenv TAU_DEPTH_LIMIT 2
 - enables instrumentation in main and routines that are directly called by main
- Stub makefile has -depthlimit in its name:
 - setenv TAU_MAKEFILE <taudir>/<arch>/lib/Makefile.tau-icpc-mpi-depthlimit-pdt

Generate a Callpath Profile

```
% setenv TAU_MAKEFILE /opt/tau-2.18/x86_64
                        /lib/Makefile.tau-callpath-mpi-pdt
% set path=(/opt/tau-2.18/x86_64/bin $path)
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% setenv TAU_CALLPATH_DEPTH 100
```

NOTE: In TAU v2.18.1+ you may simply use:

```
% setenv TAU_CALLPATH 1
to generate the callpath profiles without any recompilation.
% qsub run.job
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.
% paraprof app.ppk
(Windows -> Thread -> Call Graph)
```

-TRACE Configuration Option

- Generates event-trace logs, rather than summary profiles
- Traces show when and where an event occurred in terms of location and the process that executed it
- Traces from multiple processes are merged:
 - % tau_treemerge.pl
 - generates tau.trc and tau.edf as merged trace and event definition file
- TAU traces can be converted to Vampir's OTF/VTF3, Jumpshot SLOG2, Paraver trace formats:
 - % tau2otf tau.trc tau.edf app.otf
 - % tau2vtf tau.trc tau.edf app.vpt.gz
 - % tau2slog2 tau.trc tau.edf -o app.slog2
 - % tau_convert -paraver tau.trc tau.edf app.prv
- Stub Makefile has -trace in its name
 - % setenv TAU_MAKEFILE <taudir>/<arch>/lib/Makefile.tau-icpc-mpi-pdt-trace

Generate a Trace File

```
% setenv TAU_MAKEFILE /opt/tau-2.18/x86_64
                        /lib/Makefile.tau-mpi-pdt-trace
% set path=(/opt/tau-2.18/x86_64/bin $path)
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% qsub run.job
% tau_treemerge.pl
(merges binary traces to create tau.trc and tau.edf files)
JUMPSHOT:
% tau2slog2 tau.trc tau.edf -o app.slog2
% jumpshot app.slog2
    OR
VAMPIR:
% tau2otf tau.trc tau.edf app.otf -n 4 -z
(4 streams, compressed output trace)
% vampir app.otf
(or vng client with vngd server)
```

Instrumentation Specification

```
% tau_instrumentor
```

```
Usage : tau_instrumentor <pdbfile> <sourcefile> [-o <outputfile>] [-noinline]  
[-g groupname] [-i headerfile] [-c|-c++|-fortran] [-f <instr_req_file> ]
```

```
For selective instrumentation, use -f option
```

```
% tau_instrumentor foo.pdb foo.cpp -o foo.inst.cpp -f selective.dat
```

```
% cat selective.dat
```

```
# Selective instrumentation: Specify an exclude/include list of routines/files.
```

```
BEGIN_EXCLUDE_LIST
```

```
void quicksort(int *, int, int)
```

```
void sort_5elements(int *)
```

```
void interchange(int *, int *)
```

```
END_EXCLUDE_LIST
```

```
BEGIN_FILE_INCLUDE_LIST
```

```
Main.cpp
```

```
Foo?.c
```

```
*.C
```

```
END_FILE_INCLUDE_LIST
```

```
# Instruments routines in Main.cpp, Foo?.c and *.C files only
```

```
# Use BEGIN_[FILE]_INCLUDE_LIST with END_[FILE]_INCLUDE_LIST
```

Outer Loop Level Instrumentation

```
BEGIN_INSTRUMENT_SECTION
loops file="loop_test.cpp" routine="multiply"
# it also understands # as the wildcard in routine name
# and * and ? wildcards in file name.
# You can also specify the full
# name of the routine as is found in profile files.
#loops file="loop_test.cpp" routine="double multiply#"
END_INSTRUMENT_SECTION
```

```
% pprof
```

```
NODE 0;CONTEXT 0;THREAD 0:
```

```
-----
%Time      Exclusive      Inclusive      #Call      #Subrs      Inclusive Name
          msec          total msec
-----
100.0         0.12          25,162         1           1      25162827 int main(int, char **)
100.0         0.175         25,162         1           4      25162707 double multiply()
 90.5        22,778        22,778         1           0      22778959 Loop: double multiply() [
file = <loop_test.cpp> line,col = <23,3> to <30,3> ]
 9.3          2,345         2,345         1           0      2345823 Loop: double multiply() [
file = <loop_test.cpp> line,col = <38,3> to <46,7> ]
 0.1           33           33           1           0      33964 Loop: double
multiply() [ file = <loop_test.cpp> line,col = <16,10> to <21,12> ]
-----
```

Support Acknowledgements

- Department of Energy (DOE)
 - Office of Science
 - MICS, Argonne National Lab
 - ASC/NNSA
 - University of Utah ASC/NNSA Level 1
 - ASC/NNSA, Lawrence Livermore National Lab



- Department of Defense (DoD)
 - HPC Modernization Office (HPCMO)



- NSF Software Development for Cyberinfrastructure (SDCI)
- Research Centre Juelich



- Los Alamos National Laboratory
- TU Dresden
- ParaTools, Inc.



ParaTools

POINT

Parallel Performance Evaluation Tools for HPC Systems: ICCS '09



For more information

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

VIRTUAL INSTITUTE – HIGH PRODUCTIVITY SUPERCOMPUTING

Markus Geimer
Jülich Supercomputing Centre

POINT

Parallel Performance Evaluation Tools for HPC Systems: ICCS '09



Virtual Institute – High Productivity Supercomputing

Goal: Improve the quality and accelerate the development process of complex simulation codes running on highly-parallel computer systems

- Funded by Helmholtz Association of German Research Centres
- Activities
 - Development and integration of HPC programming tools
 - Correctness checking & performance analysis
 - Training workshops
 - Service
 - Support email lists
 - Application engagement
 - Academic workshops



www.vi-hps.org

Partners



Forschungszentrum Jülich

- Jülich Supercomputing Centre



RWTH Aachen University

- Centre for Computing and Communication



Technical University of Dresden

- Centre for Information Services and HPC



University of Tennessee (Knoxville)

- Innovative Computing Laboratory



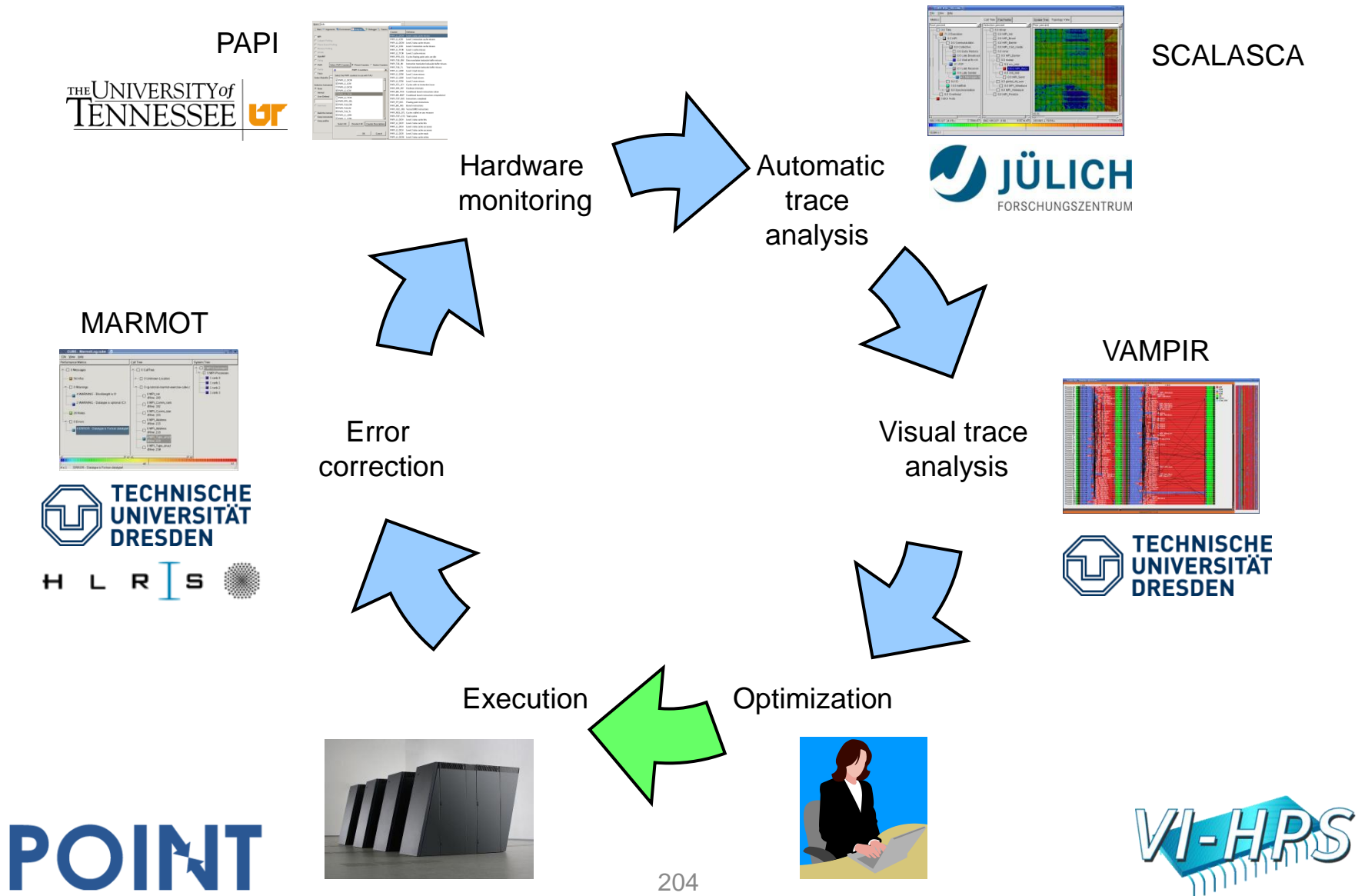
University of Stuttgart

- High Performance Computing Centre

Productivity tools

- Marmot
 - Free MPI correctness checking tool
- PAPI
 - Free library interfacing to hardware performance counters
- Scalasca
 - Open-source toolset for analysing the performance behaviour of parallel applications to automatically identify inefficiencies
- Vampir
 - Commercial framework and graphical analysis tool to display and analyse event traces
- VampirTrace
 - Open-source tool generating event traces for analysis and visualization by Vampir
- [Tuning Workshop Live-DVD contains latest tools releases]

Technologies and their integration

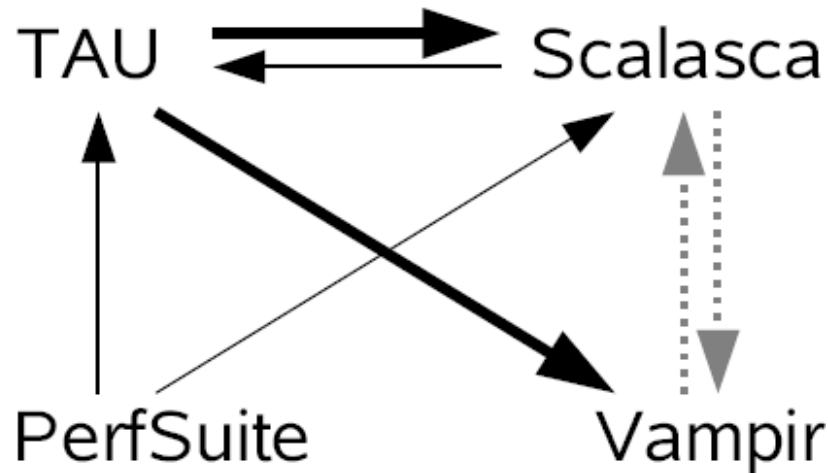


VI-HPS component technologies

- Key tool components also provided as open-source
 - Program/library instrumentation
 - OPARI, POMP
 - Scalable I/O
 - SIONlib
 - Libraries & tools for handling (and converting) traces
 - PEARL, EARL, EPILOG, OTF, CE
 - Analysis algebra & hierarchical/topological presentation
 - CUBE

POINT/VI-HPS tool interoperability

- PerfSuite can generate reports in CUBE format
- TAU can use Scalasca & VampirTrace measurement libs and can present reports in PerfSuite & CUBE formats
- TAU & Vampir use OPARI to instrument OpenMP sources, and Scalasca can use TAU source instrumenter
- Scalasca & Vampir traces can be inter-converted



VI-HPS Training & Tuning Workshops

- Goals
 - Give an overview of the programming tools suite
 - Explain the functionality of individual tools
 - Teach how to use the tools effectively
 - Offer hands-on experience and expert assistance using tools
 - Receive feedback from users to guide future development
- For best results, bring & analyse/tune your own code(s)!
- VI-HPS Tuning Workshop series
 - Aachen (Mar'08), Dresden (Oct'08), Jülich (Feb'09), ...
- Joint POINT/VI-HPS Tutorial series
 - Austin/SC (Nov'08), Baton Rouge/ICCS (May'09), ...
- Training with individual tools & platforms (e.g., BlueGene)

SCALABLE PERFORMANCE ANALYSIS OF LARGE-SCALE PARALLEL APPLICATIONS

Markus Geimer

Brian J. N. Wylie

Jülich Supercomputing Centre



POINT

Parallel Performance Evaluation Tools for HPC Systems: ICCS '09

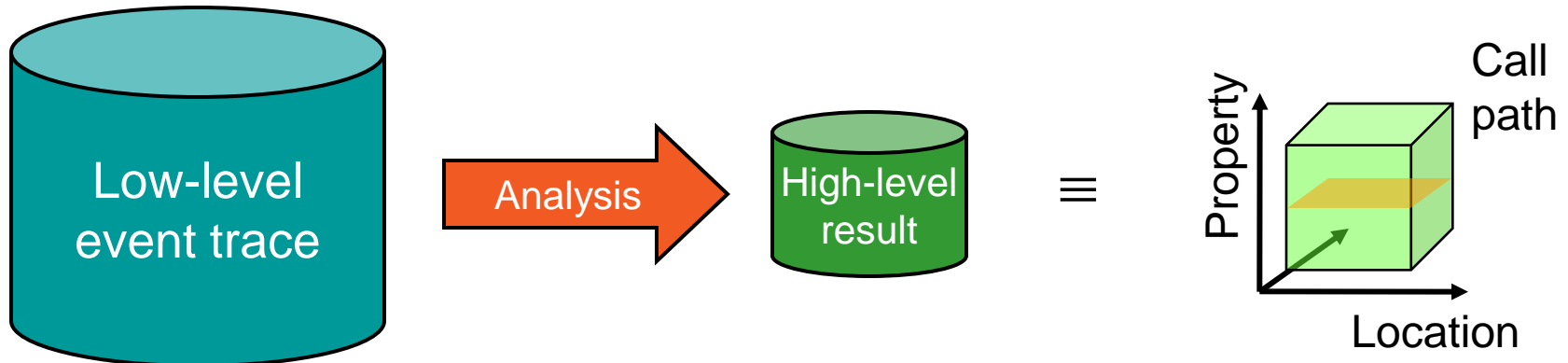


Performance analysis, tools & techniques

- Profile analysis
 - Summary of aggregated metrics
 - per function/call-path and/or per process/thread
 - Most tools (can) generate and/or present such profiles
 - but they do so in *very* different ways, often from event traces!
 - e.g., mpiP, ompP, Tau, **Scalasca**, Sun Studio, Vampir, ...
- Time-line analysis
 - Visual representation of the space/time sequence of events
 - Requires an execution trace
 - e.g., Vampir, Paraver, Sun Studio Performance Analyzer, ...
- Pattern analysis
 - Search for characteristic event sequences in event traces
 - Can be done manually, e.g., via visual time-line analysis
 - Can be done automatically, e.g., KOJAK, **Scalasca**

Automatic trace analysis

- Idea
 - Automatic search for patterns of inefficient behaviour
 - Classification of behaviour & quantification of significance



- Guaranteed to cover the entire event trace
- Quicker than manual/visual trace analysis
- Parallel replay analysis exploits memory & processors to deliver scalability

The Scalasca project

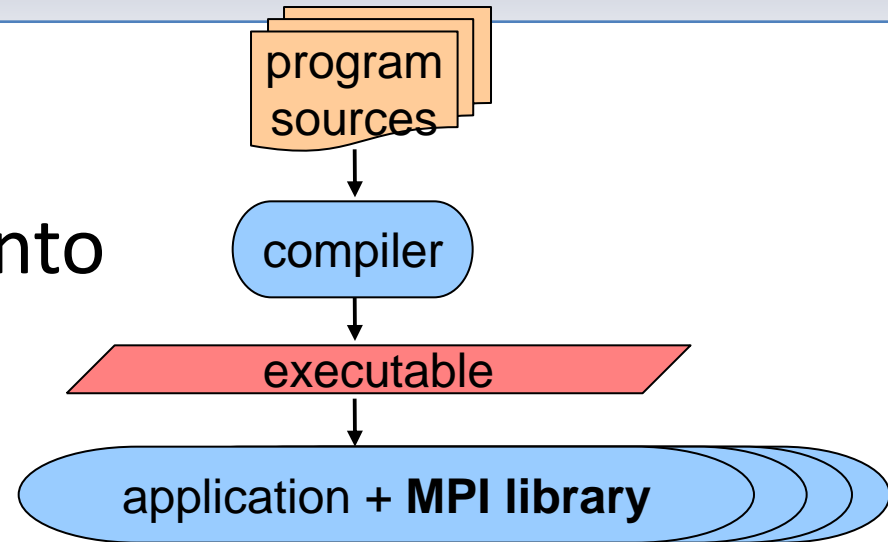
- Overview
 - Helmholtz Initiative & Networking Fund project started in 2006
 - Headed by Prof. Felix Wolf (RWTH Aachen University & FZJ)
 - Follow-up to pioneering KOJAK project (started 1998)
 - Automatic pattern-based trace analysis
- Objective
 - Development of a **scalable** performance analysis toolset
 - Specifically targeting **large-scale** parallel applications
 - such as those running on BlueGene/P or Cray XT with 10,000s to 100,000s of processes
 - Latest release in November 2008: Scalasca v1.1
 - Available on VI-HPS Tuning Workshop Live-DVD
 - Scalasca 1.2 β currently in testing

Scalasca features

- Open source, New BSD license
- Portable
 - BG/P, BG/L, IBM SP & blade clusters, Cray XT, SGI Altix, SiCortex, Solaris & Linux clusters, ...
- Supports parallel programming paradigms & languages
 - MPI, OpenMP & hybrid OpenMP/MPI
 - Fortran, C, C++
- Integrated measurement & analysis toolset
 - Runtime summarization (aka profiling)
 - Automatic event trace analysis

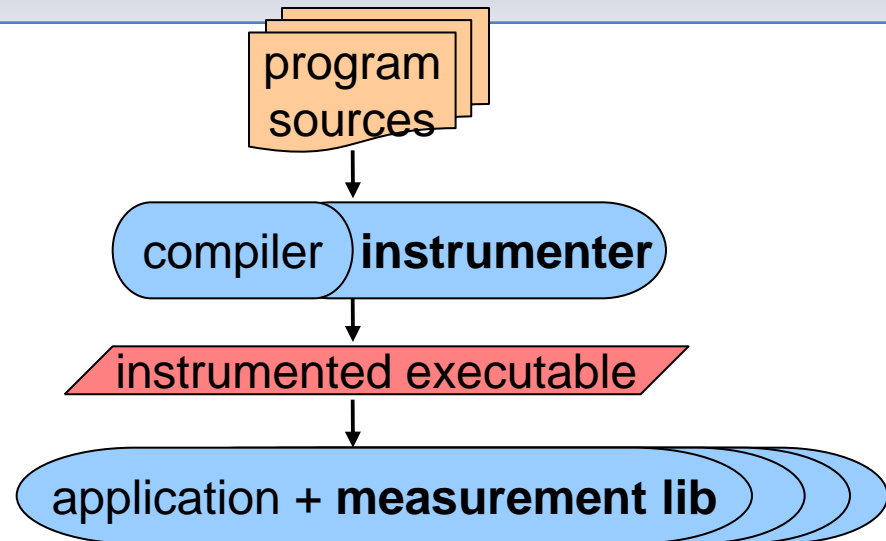
Generic MPI application build & run

- Application code compiled & linked into executable using MPICC/CXX/FC
- Launched with MPIEXEC
- Application processes interact via MPI library



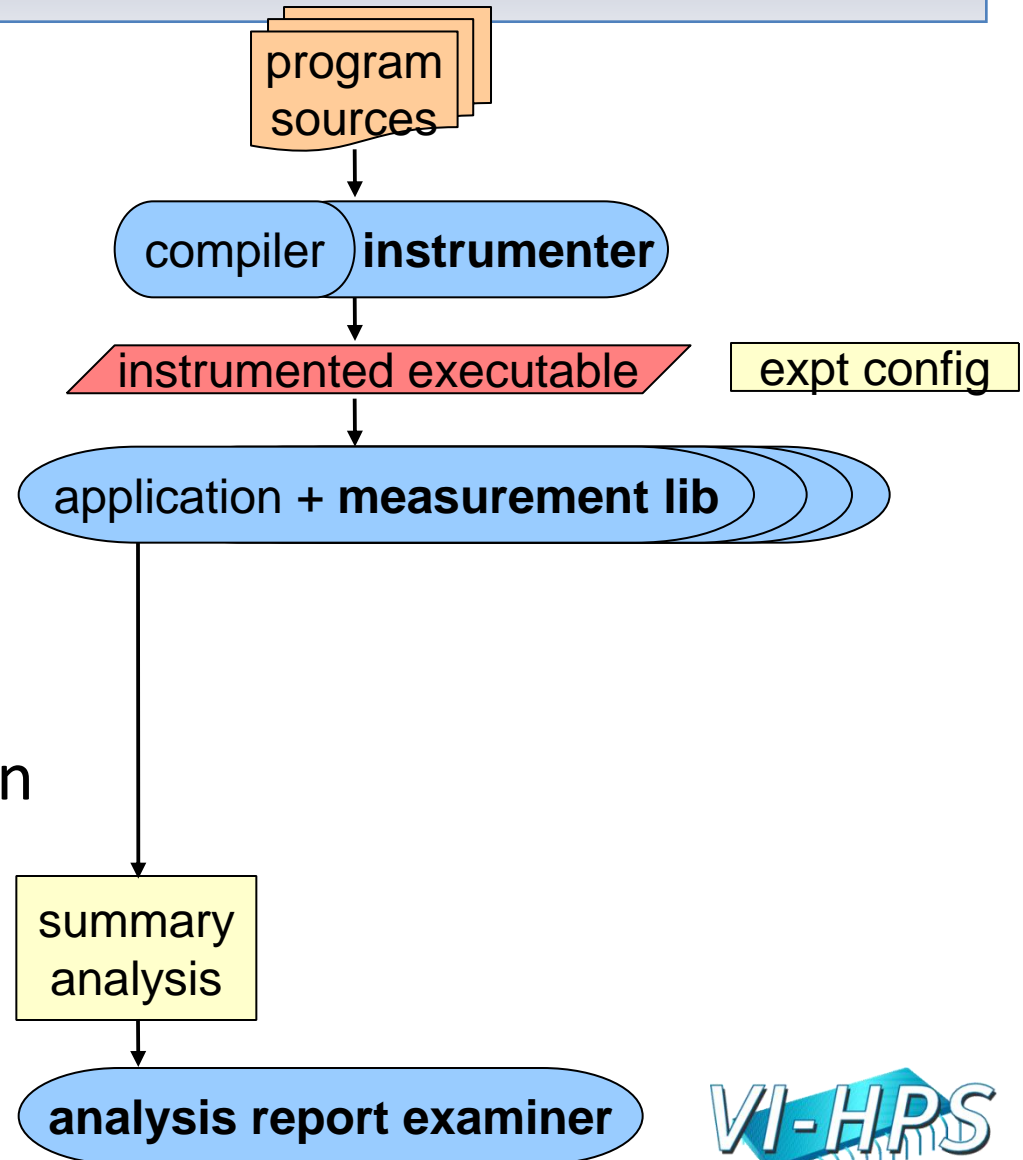
Application instrumentation

- Automatic/manual code instrumenter
- Program sources processed to add instrumentation and measurement library into application executable
- Exploits MPI standard profiling interface (PMPI) to acquire MPI events



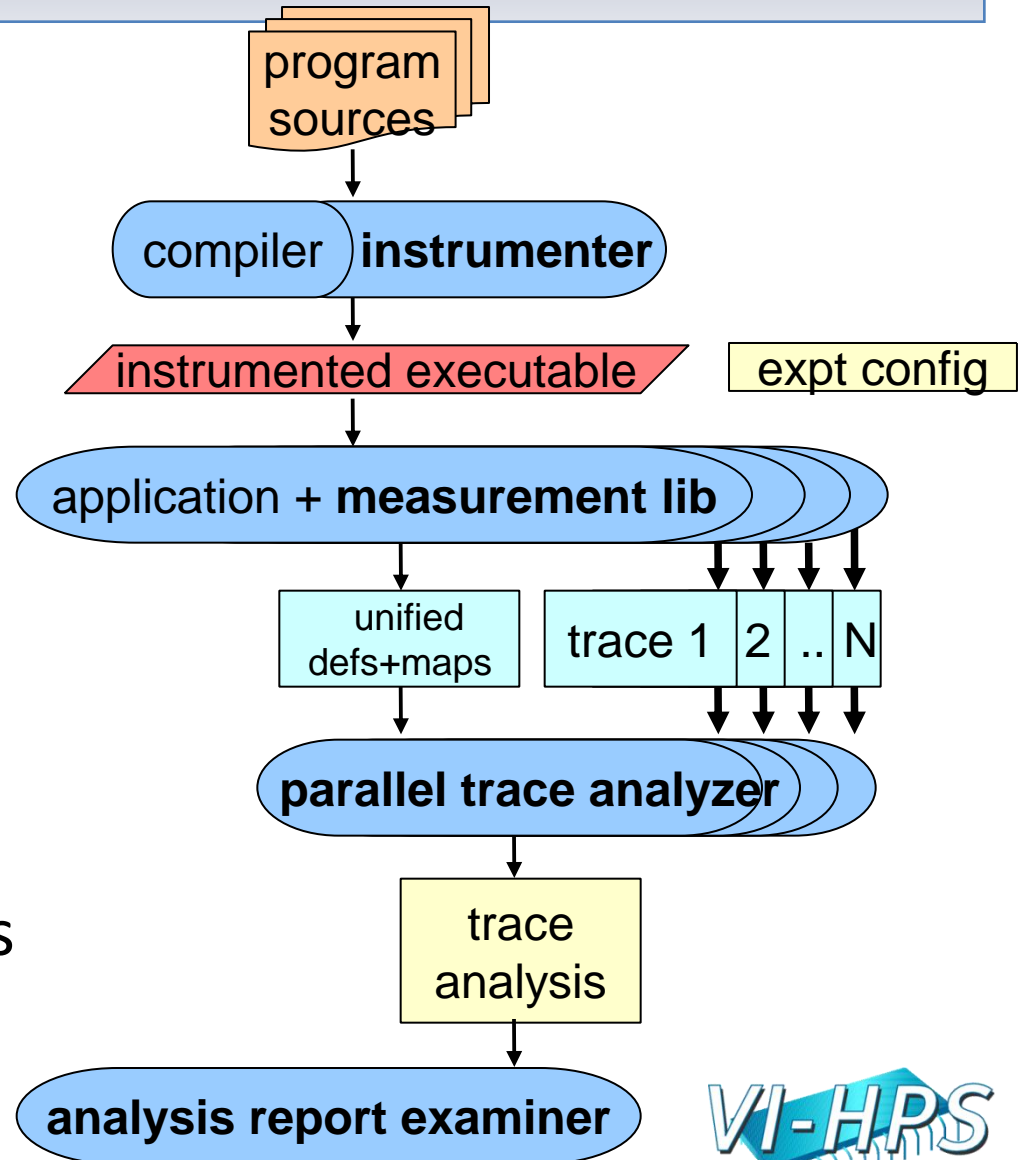
Measurement runtime summarization

- Measurement library manages threads & events produced by instrumentation
- Measurements summarized by thread & call-path during execution
- Analysis report unified & collated at finalization
- Presentation of summary analysis



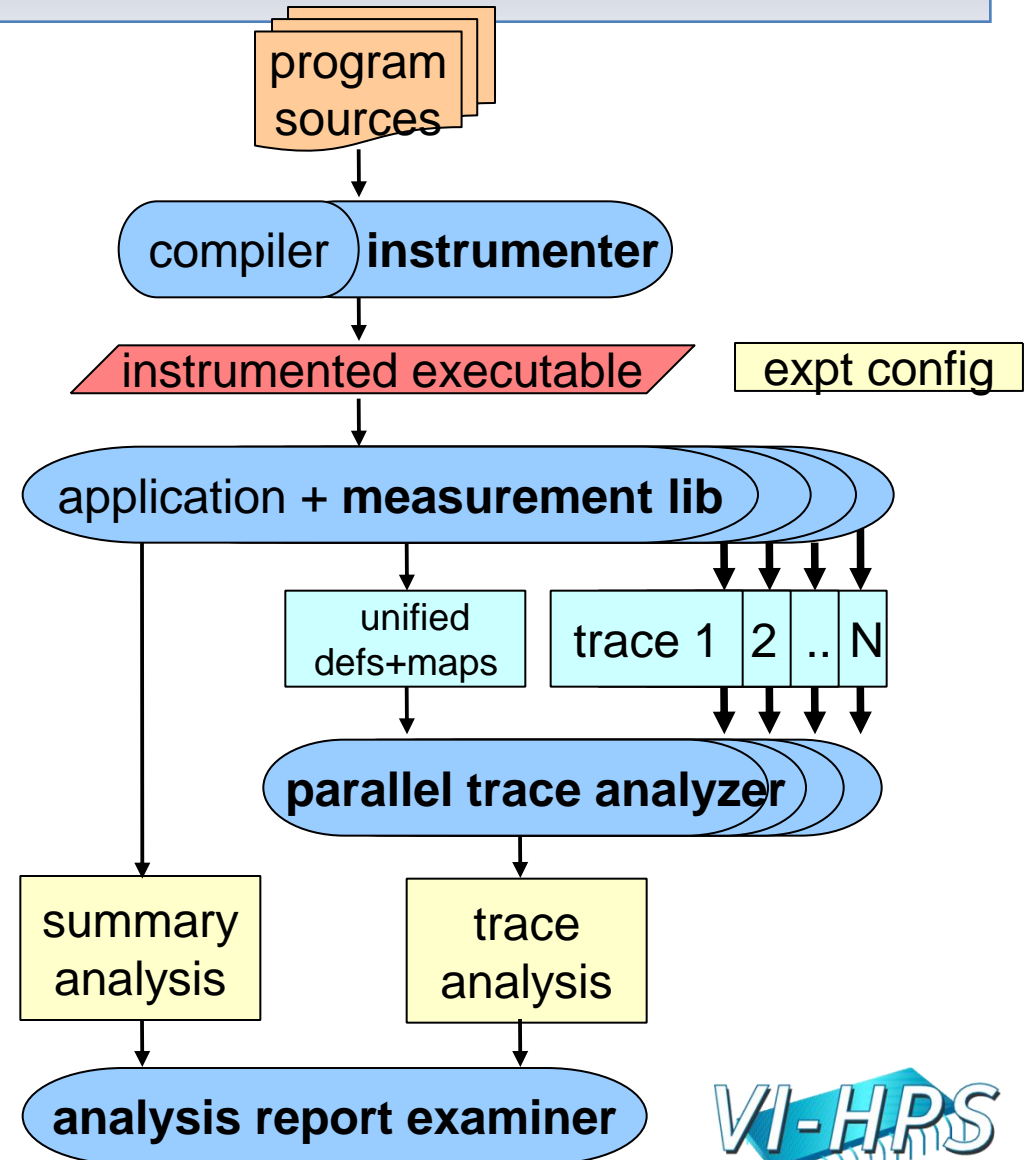
Measurement event tracing & analysis

- During measurement time-stamped events buffered for each thread
- Flushed to files along with unified definitions & maps at finalization
- Follow-up analysis replays events and produces extended analysis report
- Presentation of analysis report



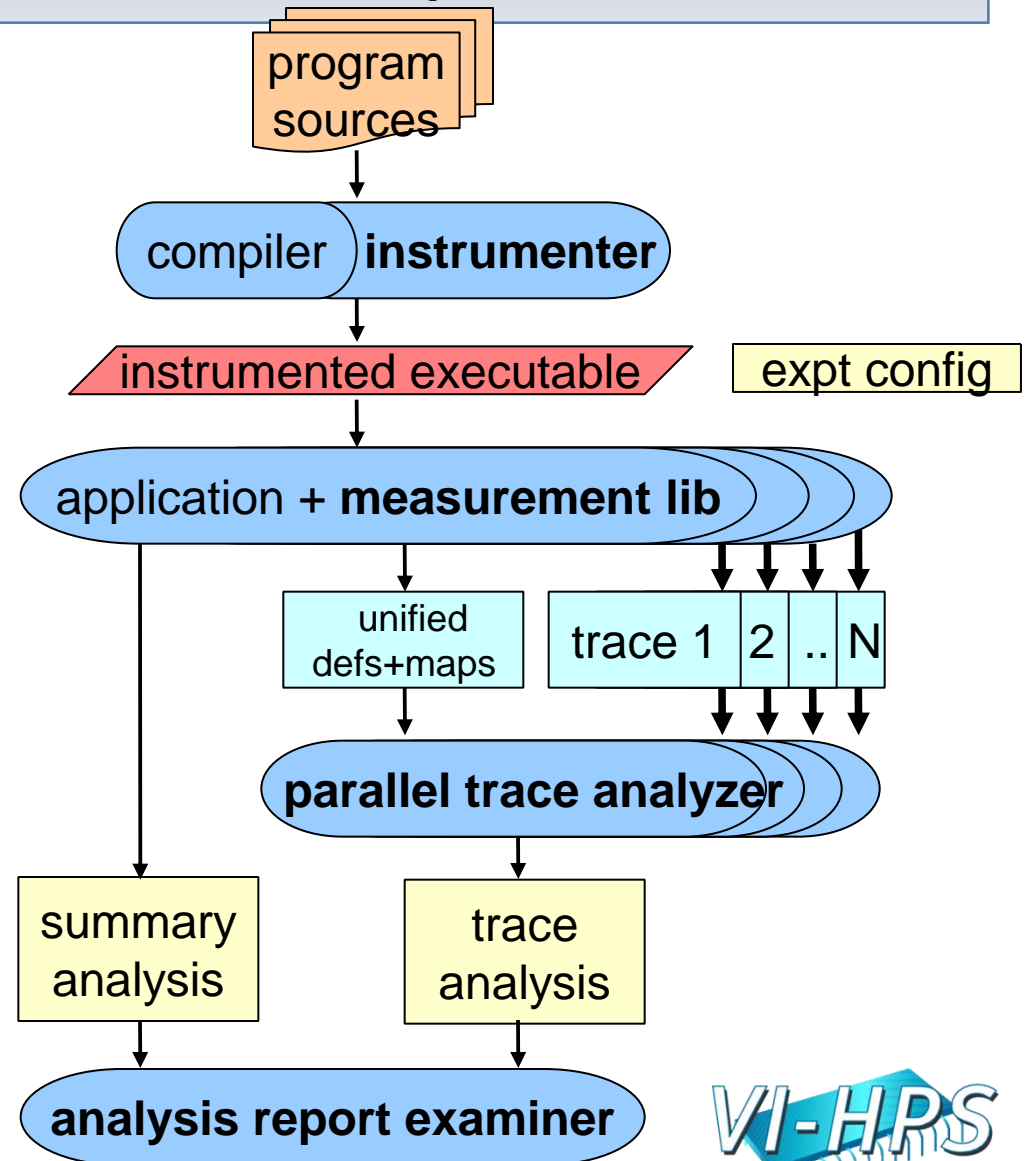
Generic parallel tools architecture

- Automatic/manual code instrumenter
- Measurement library for runtime summary & event tracing
- Parallel (and/or serial) event trace analysis when desired
- Analysis report examiner for interactive exploration of measured execution performance properties



Scalasca toolset components

- Scalasca instrumenter = SKIN
- Scalasca measurement collector & analyzer = SCAN
- Scalasca analysis report examiner = SQUARE



EPIK

- Measurement & analysis runtime system
 - Manages runtime configuration and parallel execution
 - Configuration specified via EPIK.CONF file or environment
 - epik_conf reports current measurement configuration
 - Creates experiment archive (directory): **epik_<title>**
 - Optional runtime summarization report
 - Optional event trace generation (for later analysis)
 - Optional filtering of (compiler instrumentation) events
 - Optional incorporation of HWC measurements with events
 - via PAPI library, using PAPI preset or native counter names
- Experiment archive directory
 - Contains (single) measurement & associated files (e.g., logs)
 - Contains (subsequent) analysis reports

scalasca

- One command for everything

% **scalasca**

Scalasca 1.1

Toolset for scalable performance analysis of large-scale apps

usage: scalasca [-v][-n] {action}

1. prepare application objects and executable for measurement:

scalasca *-instrument* <compile-or-link-command> # **skin**

2. run application under control of measurement system:

scalasca *-analyze* <application-launch-command> # **scan**

3. interactively explore measurement analysis report:

scalasca *-examine* <experiment-archive|report> # **square**

[-h] show quick reference guide (only)

scalasca actions

- One command for everything

% **scalasca** -usage

% scalasca **-instrument** [options] <compile-or-link-command>

% scalasca **-analyze** [options] <application-launch-command>

% scalasca **-examine** [options] <experiment-archive | report>

... that does nothing!

– simply a shell script wrapper for action commands:

% **skin** [options] <compile-or-link-command>

- *prepare application objects and executable for measurement*

% **scan** [options] <application-launch-command>

- *run application under control of measurement system*

% **square** [options] <experiment-archive | report>

- *interactively explore measurement analysis report*

OPARI

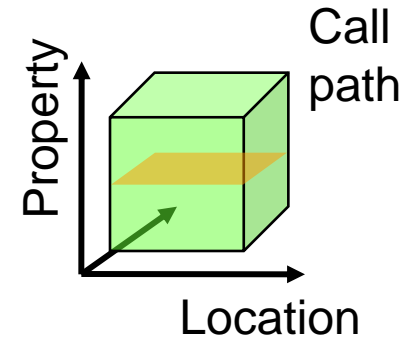
- Automatic instrumentation of OpenMP & POMP directives via source pre-processor
 - Parallel regions, worksharing, synchronization
 - Currently limited to OpenMP 2.5
 - No special handling of guards, dynamic or nested thread teams
 - Configurable to disable instrumentation of locks, etc.
 - Typically invoked internally by instrumentation tools
- Used by Scalasca/Kojak, ompP, TAU, VampirTrace, etc.
 - Provided with Scalasca, but also available separately

CUBE3

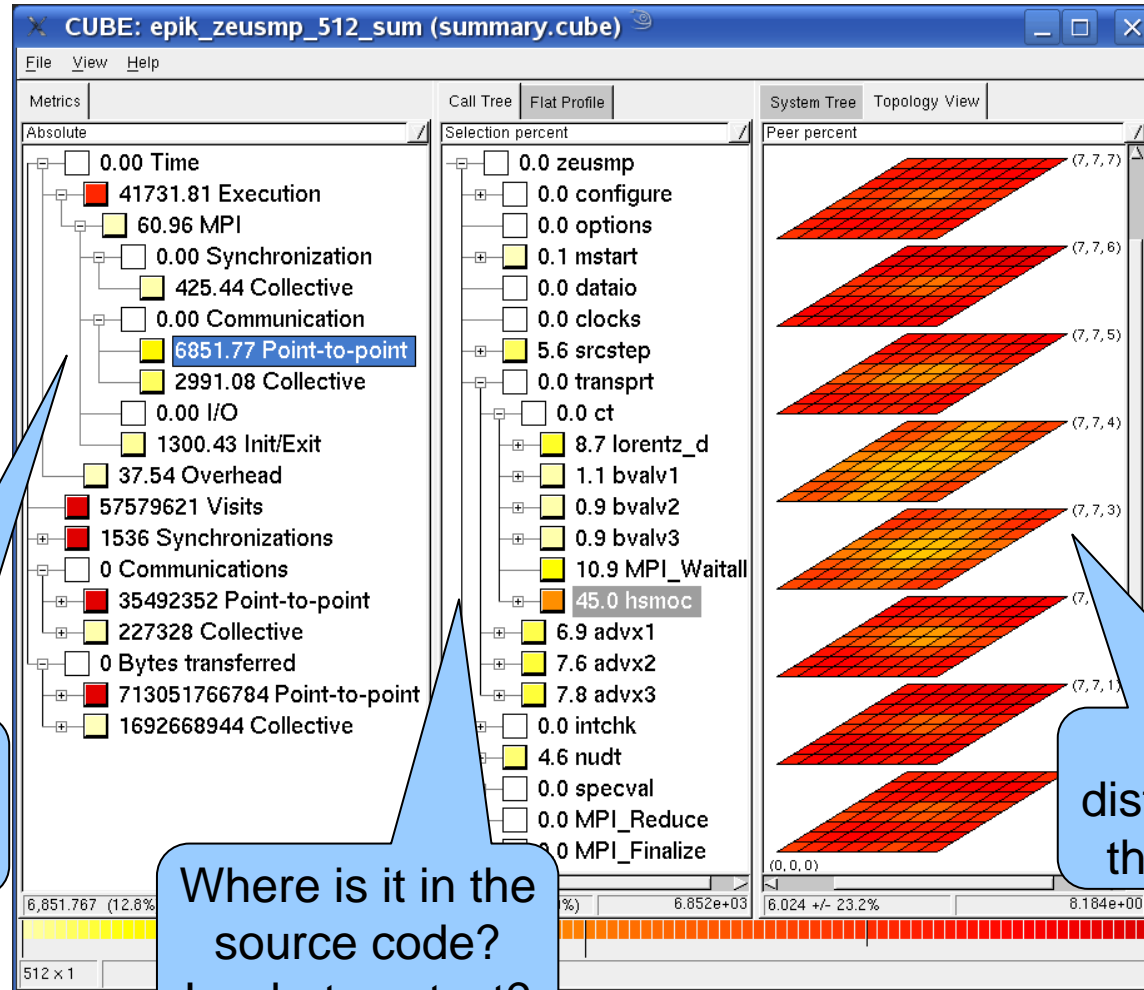
- Parallel program analysis report exploration tools
 - Libraries for XML report reading & writing
 - Algebra utilities for report processing
 - GUI for interactive analysis exploration
 - requires Qt4 or wxGTK widgets library
 - can be installed independently of Scalasca instrumenter and measurement collector/analyzer, e.g., on laptop or desktop
- Used by Scalasca/Kojak, Marmot, ompP, PerfSuite, etc.
 - Provided with Scalasca, but also available separately

Analysis presentation and exploration

- Representation of values (severity matrix) on three hierarchical axes
 - Performance property (metric)
 - Call-tree path (program location)
 - System location (process/thread)
- Three coupled tree browsers
- CUBE3 displays severities
 - As value: for precise comparison
 - As colour: for easy identification of hotspots
 - Inclusive value when closed & exclusive value when expanded
 - Customizable via display mode



Scalasca analysis report explorer (summary)



What kind of performance problem?

Where is it in the source code?
In what context?

How is it distributed across the processes?

Scalasca analysis report explorer (trace)

The screenshot shows the Scalasca analysis report explorer interface. The main window displays a tree view of metrics for the trace file 'epik_zeusmp_512_trace (trace.cube)'. The 'Point-to-point' metric is highlighted with a blue callout box. A secondary window titled 'CUBE metric description <@j36>' provides a detailed description of the 'Late Sender Time' metric, including a diagram illustrating the timing of a blocking receive operation relative to a send operation.

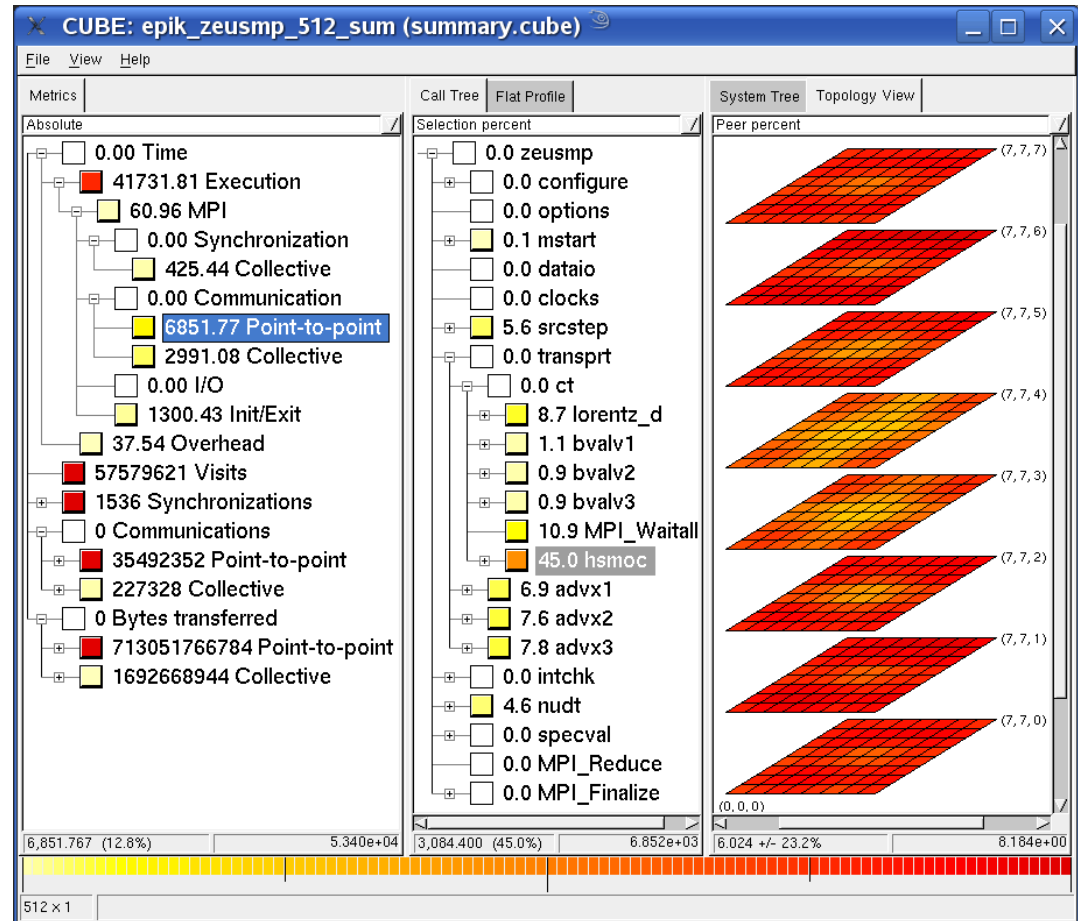
Additional metrics determined from trace

CUBE metric description <@j36>
Late Sender Time
 Description: Refers to the time lost waiting caused by a blocking receive operation (e.g., MPI_Recv() or MPI_Wait()) that is posted earlier than the corresponding send operation.

The diagram shows a timeline with 'location' on the vertical axis and 'time' on the horizontal axis. A 'Send' operation is shown as a red box, and a 'Recv' operation is shown as a yellow box. A red double-headed arrow indicates the time interval between the start of the 'Recv' operation and the start of the 'Send' operation, representing the 'Late Sender Time'.

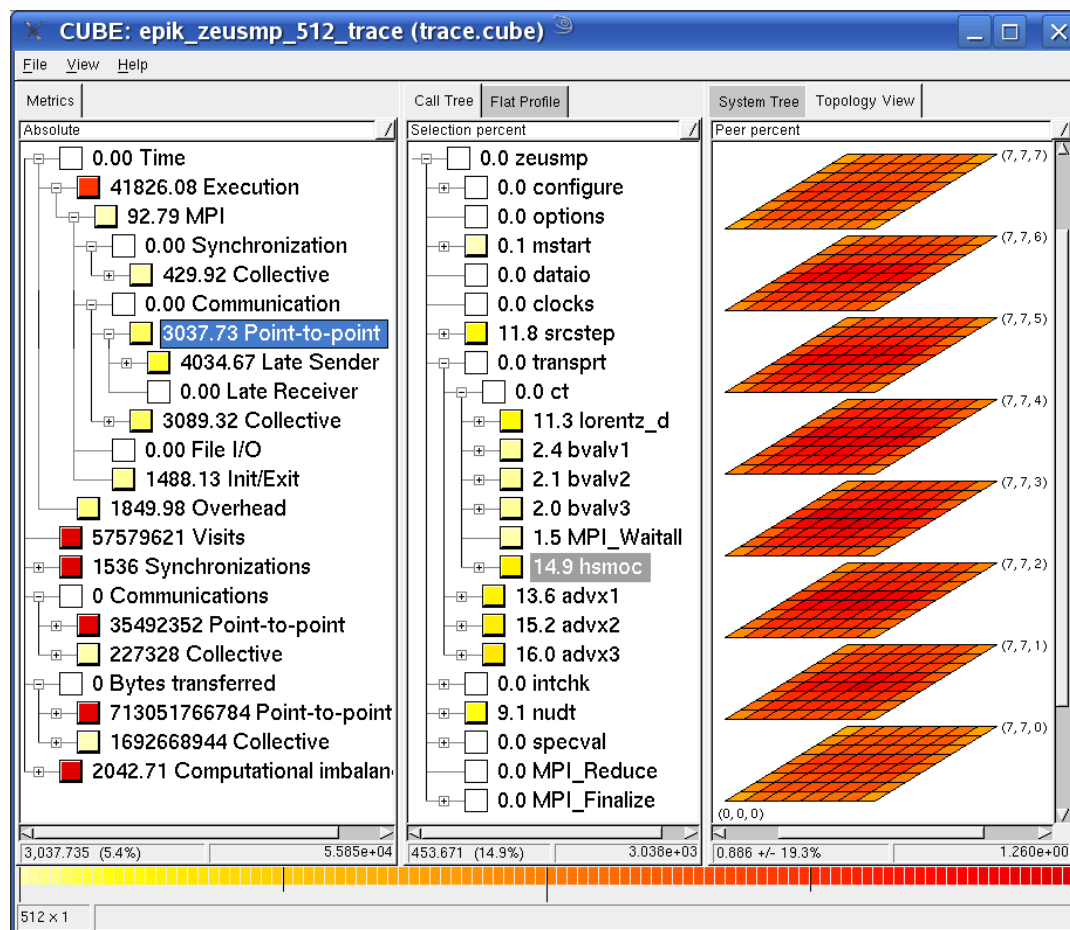
Scalasca summary analysis: zeusmp2 on jump

- 12.8% of time spent in MPI point-to-point communication
- 45.0% of which is on program callpath `transprt/ct/hsmoc`
- With 23.2% std dev over 512 processes
- Lowest values in 3rd and 4th planes of the Cartesian grid



Scalasca trace analysis: zeusmp2 on jump

- MPI point-to-point communication time separated into transport and Late Sender fractions
- Late Sender situations dominate (57%)
- Distribution of transport time (43%) indicates congestion in interior of grid



Scalasca 1.1 functionality

- MPI measurement & analyses
 - scalable runtime summarization & event tracing
 - only requires application executable re-linking
 - limited analyses of non-blocking point-to-point, RMA, ...
- OpenMP measurement & analysis
 - serial event trace analysis (of merged traces)
 - runtime summarization limited to master thread only
 - requires (automatic) application source instrumentation
 - restricted to fixed OpenMP thread teams
- Hybrid OpenMP/MPI measurement & analysis
 - combined requirements/capabilities
 - automatic trace analysis is scalable but incomplete
 - can repeat analysis with serial analyzer when desired

Scalasca 1.2 β additional functionality

- OpenMP measurement & analysis
 - run-time summaries include OpenMP metrics (for all threads)
 - not all threads need to participate in parallel regions
 - trace collection & analysis unchanged
- Hybrid OpenMP/MPI measurement & analysis
 - OpenMP metrics (for all threads) included in run-time summaries and trace analysis
- MPI File I/O analysis
 - collective read/write time
 - file operations (reads/writes)

Tutorial Exercise

NPB-MPI BT



Performance analysis steps

1. Reference preparation for validation
 2. Program instrumentation: `skin`
 3. Summary measurement collection & analysis: `scan [-s]`
 4. Summary analysis report examination: `square`
 5. Summary experiment scoring: `square -s`
 6. Event trace collection & analysis: `scan -t`
 7. Event trace analysis report examination: `square`
- Configuration & customization
 - Instrumentation, Measurement, Analysis, Presentation

Live-DVD tutorial sources

- Tutorial example sources provided for several programs (implemented in various languages & parallelizations)
 - Scalasca
 - jacobi # MPI/OpenMP/hybrid x C/C++/Fortran
 - sweep3d # MPI/Fortran
 - smg2000 # MPI/C
 - NPB3.3-MPI # MPI/Fortran & C
 - NPB3.3-OMP # OpenMP/Fortran & C
 - NPB3.3-MZ-MPI # hybrid OpenMP+MPI/Fortran
- This tutorial concentrates on NPB3.3-MPI-BT
 - but can be repeated substituting other examples as desired

NPB-BT

- Intermediate-level tutorial example
- Available in MPI, OpenMP, hybrid OpenMP/MPI variants
 - also MPI File I/O variants (collective & individual)
- Summary measurement collection & analysis
 - Automatic instrumentation
 - OpenMP, MPI & application functions
 - Summary analysis report examination
- Trace measurement collection & analysis
 - Filter determination, specification & configuration
 - Automatic trace analysis report patterns
- (Analysis report algebra)

NPB-MPI suite

- The NAS Parallel Benchmark suite (sample MPI version)
 - Available from <http://www.nas.nasa.gov/Software/NPB>
 - 9 benchmarks (7 in Fortran77, 2 in C)
 - Configurable for various sizes & classes
- Move into the NPB3.3-MPI root directory

```
% cd NPB3.3-MPI; ls
BT/      CG/      DT/      EP/      FT/      IS/      LU/      MG/      SP/
bin/     common/ config/  Makefile  README  README.install  sys/
```

- Subdirectories contain source code for each benchmark
 - plus additional configuration and common code
- The provided distribution has already been configured for the tutorial, such that it's ready to “make” one or more of the benchmarks and install them into the “bin” subdirectory

Building an NPB benchmark

- Specify the benchmark configuration
 - benchmark name: **bt**, cg, dt, ep, ft, is, lu, mg, sp
 - the number of MPI processes: **NPROC=16**
 - the benchmark class (S, W, A, B, C, D, E, F): **CLASS=W**

```
% make bt NPROCS=16 CLASS=W
cd BT; make NPROCS=16 CLASS=W SUBTYPE= VERSION=
gmake: Entering directory 'BT'
cd ../sys; cc -o setparams setparams.c
../sys/setparams bt 16 W
mpif77 -c -O bt.f
...
cd ../common; mpif77 -c -O timers.f
mpif77 -c -O btio.f
mpif77 -O -o ../bin/bt.W.16 \
bt.o make_set.o initialize.o exact_solution.o exact_rhs.o \
set_constants.o adi.o define.o copy_faces.o rhs.o solve_subs.o \
x_solve.o y_solve.o z_solve.o add.o error.o verify.o setup_mpi.o \
../common/print_results.o ../common/timers.o btio.o
gmake: Leaving directory 'BT'
```

NPB-MPI BT (Block Tridiagonal solver)

- What does it do?
 - Solves a discretized version of unsteady, compressible Navier-Stokes equations in three spatial dimensions
 - Performs 200 time-steps on a regular 3-dimensional grid
- Can be configured to include various forms of parallel I/O
 - e.g., MPI collective file I/O: SUBTYPE=full
- Implemented in 20 or so Fortran77 source modules

- Needs a square number of processes
 - bt.W.4 should run in around 5 seconds with 4 processes
 - bt.A.4 should take around 16-20x longer (90 seconds)
 - bt.W.16 may also run in around 5 seconds with 16 processes

BT-MPI reference execution

- Launch as an MPI application

```
% mpiexec -np 16 bin/bt.W.16
NAS Parallel Benchmarks 3.3 -- BT Benchmark
Size: 24x 24x 24
Iterations: 200 dt: 0.0008000
Number of active processes: 16

Time step 1
Time step 20
Time step 40
Time step 60
...
Time step 160
Time step 180
Time step 200
Verification Successful

BT Benchmark Completed.
Time in seconds = 4.70
```

Load the Scalasca module

- Load the module

```
% module load UNITE
UNITE loaded
% module load scalasca
scalasca/1.1 loaded
```

- ... and run **scalasca** for brief usage information

```
% scalasca
Scalasca 1.1
Toolset for scalable performance analysis of large-scale applications
usage: scalasca [-v][-n] {action}
  1. prepare application objects and executable for measurement:
     scalasca -instrument <compile-or-link-command> # skin
  2. run application under control of measurement system:
     scalasca -analyze <application-launch-command> # scan
  3. interactively explore measurement analysis report:
     scalasca -examine <experiment-archive|report> # square

-v: enable verbose commentary
-n: show actions without taking them
-h: show quick reference guide (only)
```

NPB-MPI-BT instrumented build

- Return to root directory and clean-up

```
% make clean
```

- Re-build specifying Scalasca instrumenter as PREP

```
% make bt NPROCS=16 CLASS=W PREP="scalasca -instrument"
cd BT; make NPROCS=16 CLASS=W SUBTYPE= VERSION=
gmake: Entering directory 'BT'
cd ../sys; cc -o setparams setparams.c
../sys/setparams bt 16 W
scalasca -instrument mpif77 -c -O bt.f
...
cd ../common; scalasca -instrument mpif77 -c -O timers.f
scalasca -instrument mpif77 -c -O btio.f
scalasca -instrument mpif77 -O -o ../bin/bt.W.16 \
bt.o make_set.o initialize.o exact_solution.o exact_rhs.o \
set_constants.o adi.o define.o copy_faces.o rhs.o solve_subs.o \
x_solve.o y_solve.o z_solve.o add.o error.o verify.o setup_mpi.o \
../common/print_results.o ../common/timers.o btio.o
gmake: Leaving directory 'BT'
```


NPB instrumentation

- PREP macro in Makefile definitions (config/make.def) used as a preparator prefix for compile/link commands

```
MPIF77 = $(PREP) mpif77
FLINK = $(MPIF77)
FFLAGS = -O

mpi-bt: $(OBJECTS)
    $(FLINK) $(FFLAGS) -o mpi-bt $(OBJECTS)
.f.o:
    $(MPIF77) $(FFLAGS) -c $<
```

- By default, PREP macro is not set and no instrumentation is performed for a regular “production” build
- Specifying a PREP value in the Makefile or on the make command line uses it as a prefix, e.g., for instrumentation
 - make PREP=“scalasca -instrument”

BT-MPI summary measurement

- Run the application using the Scalasca measurement collection & analysis nexus prefixed to launch command

```
% scalasca -analyze mpiexec -np 16 ./bt.W.mpi
S=C=A=N: Scalasca 1.1 runtime summarization
S=C=A=N: ./epik_bt_16_sum experiment archive
S=C=A=N: Sun Mar 29 16:36:31 2009: Collect start
mpiexec -np 16 ./bt.A.mpi
[00000]EPIK: Created new measurement archive ./epik_bt_16_sum
[00000]EPIK: Activated ./epik_bt_16_sum [NO TRACE] (0.006s)

[... Application output ...]

[00000]EPIK: Closing experiment ./epik_bt_16_sum
[00000]EPIK: 102 unique paths (102 max paths, 4 max frames, 0 unknown)
[00000]EPIK: Unifying... done (0.023s)
[00000]EPIK: Collating... done (0.049s)
[00000]EPIK: Closed experiment ./epik_bt_16_sum (0.073s)
S=C=A=N: Sun Mar 29 16:36:45 2009: Collect done (status=0) 14s
S=C=A=N: ./epik_bt_16_sum complete.
```

- Produces experiment directory ./epik_bt_16_sum

BT-MPI summary analysis report examination

- Interactive exploration with Scalasca GUI

```
% scalasca -examine epik_bt_16_sum
INFO: Post-processing runtime summarization result...
INFO: Displaying ./epik_bt_16_sum/summary.cube...

[GUI showing summary analysis report]
```

- Report scoring as textual output

```
% scalasca -examine -s epik_bt_16_sum
cube3_score ./epik_bt_16_sum/summary.cube
Reading ./epik_bt_16_sum/summary.cube... done.
Estimated aggregate size of event trace (total_tbc): 513,823,960 bytes
Estimated size of largest process trace (max_tbc): 32,528,680 bytes
(When tracing set ELG_BUFFER_SIZE to avoid intermediate flushes or
reduce requirements using filter file listing names of USR regions.)
```

flt	type	max_tbc	time	%	region
	ANY	32528680	220.22	100.00	(summary) ALL
	MPI	642712	194.57	88.35	(summary) MPI
	USR	31688040	24.62	11.18	(summary) USR
	COM	197928	1.03	0.47	(summary) COM

Analysis report exploration (opening view)

Cube 3.0 QT: epik_bt_16_sum/summary.cube.gz

File Display Topology Help

Metric tree

- 220.22 Time
 - 2.13e7 Visits
- 32 Synchronizations
- 1.55e5 Communications
- 1.13e9 Bytes transferred
- 0.62 Computational imbalance

Call tree Flat view


- 220.22 MAIN

System tree Topology 0

- 220.22 Linux Cluster

About Cube 3.0 QT

This is Cube 3.0 QT.

 (c) 2009 Juelich Supercomputing Centre,
Forschungszentrum Juelich GmbH

Home page: www.scalasca.org

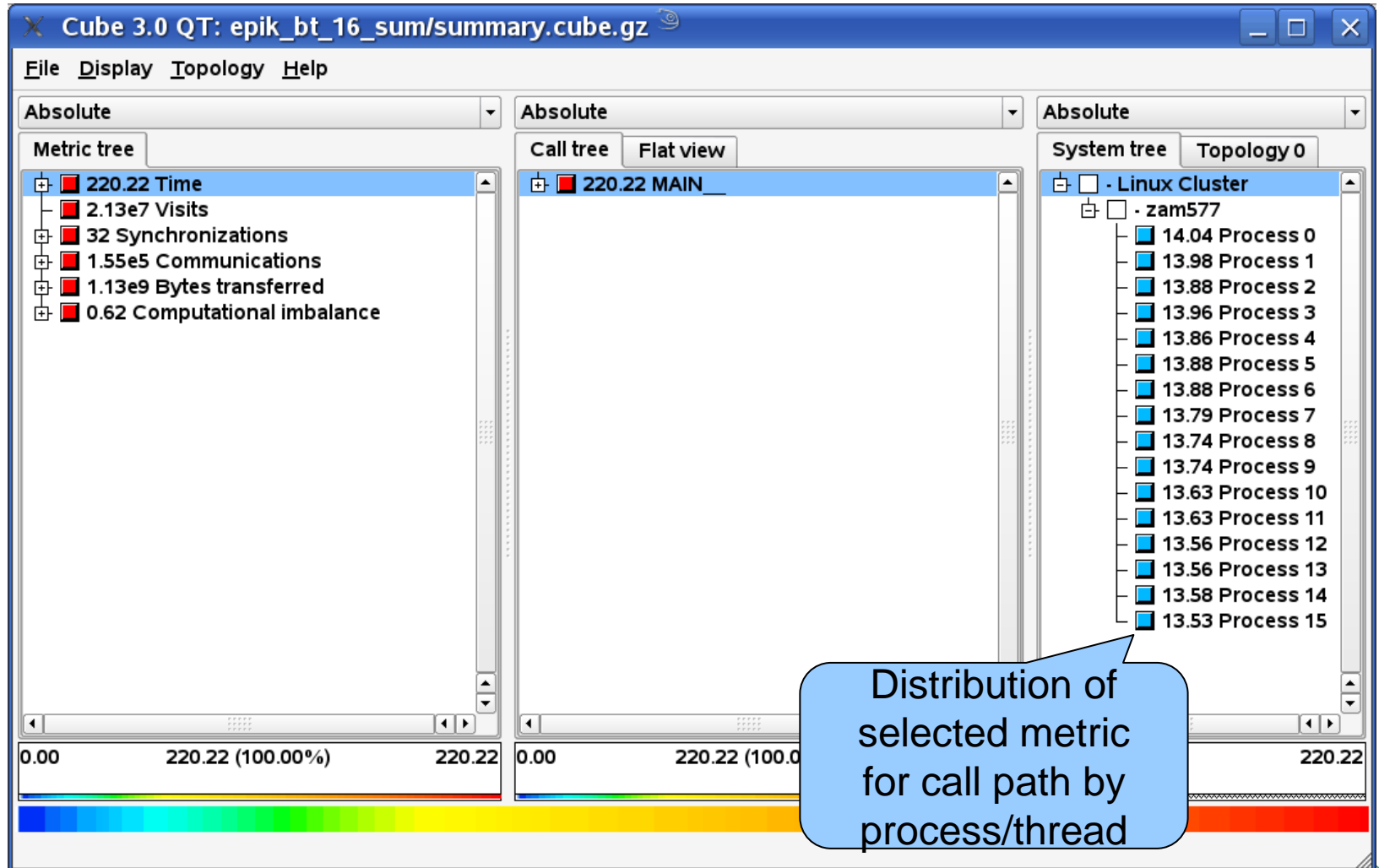
Technical support: scalasca@fz-juelich.de

0.00 220.22 (100.00%) 220.22

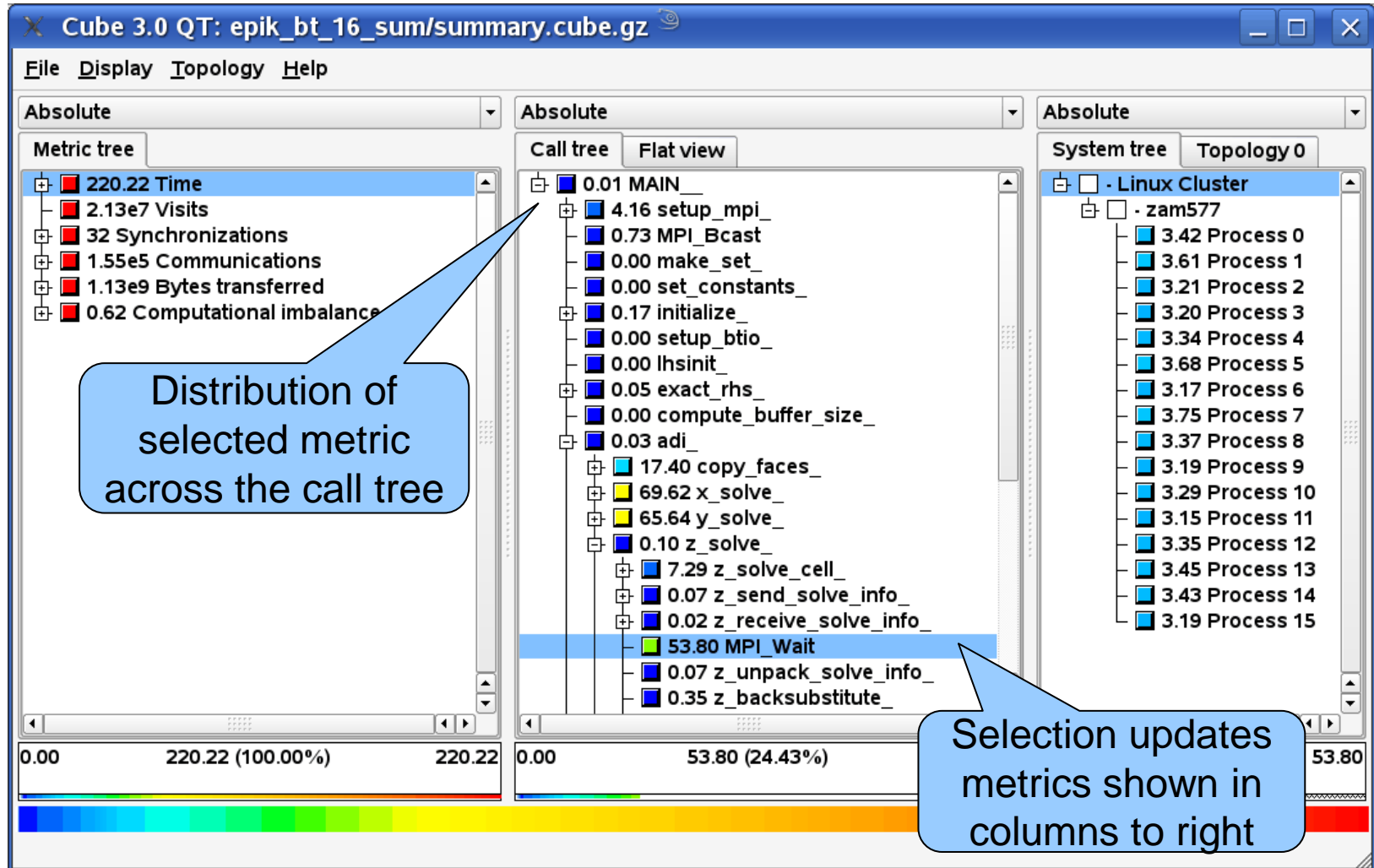
0.00 220.22 (100.00%) 220.22

0.00 220.22 (100.00%) 220.22

Analysis report exploration (system tree)



Analysis report exploration (call tree)



Analysis report exploration (source browser)

The source browser window displays the following Fortran code:

```
-----  
c in our terminology stage is the number of the cell in the y-dir-  
c i.e. stage = 1 means the start of the line stage=ncells means e  
c -----  
c do stage = 1,ncells  
c   c = slice(3,stage)  
c   isize = cell_size(1,c) - 1  
c   jsize = cell_size(2,c) - 1  
c   ksize = cell_size(3,c) - 1  
c -----  
c set last-cell flag  
c -----  
c if (stage .eq. ncells) then  
c   last = 1  
c else  
c   last = 0  
c endif  
c  
c if (stage .eq. 1) then  
c -----  
c This is the first cell, so solve without receiving data  
c -----  
c   first = 1  
c   call lhsz(c)  
c   call z_solve_cell(first,last,c)  
c else  
c -----  
c Not the first cell of this line, so receive info from  
c processor working on preceeding cell  
c -----  
c   first = 0  
c   call z_receive_solve_info(recv_id,c)  
c -----  
c overlap computations and communications  
c -----  
c   call lhsz(c)  
c -----  
c wait for completion  
c -----  
c call mpi_wait(send_id,r_status,error)  
c call mpi_wait(recv_id,r_status,error)  
c -----  
c install C'(kstart+1) and rhs'(kstart+1) to be used in this cell  
c -----  
c   call z_unpack_solve_info(c)  
c   call z_solve_cell(first,last,c)  
c endif  
c  
c if (last .eq. 0) call z_send_solve_info(send_id,c)  
c enddo
```

The analysis report explorer window shows a hierarchical tree of tasks. The 'Flat view' is selected, showing the following tasks and their execution times:

- 0.00 MAIN_
- 0.00 setup_mpi_
- 0.00 MPI_Bcast
- 0.00 make_set_
- 0.00 set_constants_
- 0.00 initialize_
- 0.00 setup_btio_
- 0.00 lhsinit_
- 0.00 exact_rhs_
- 0.00 compute_buffer_size_
- 0.00 adi_
- 16.05 copy_faces_
- 61.57 x_solve_
- 57.78 y_solve_
- 0.00 z_solve_
- 0.00 z_solve_cell_
- 0.02 z_send_solve_info_
- 0.01 z_receive_solve_info_
- 53.80 MPI_Wait
- 0.00 z_unpack_solve_info_
- 0.00 z_backsubstitute_

The system tree on the right shows a Linux Cluster with 15 processes:

- Linux Cluster
 - zam577
 - 3.42 Process 0
 - 3.61 Process 1
 - 3.21 Process 2
 - 3.20 Process 3
 - 3.34 Process 4
 - 3.68 Process 5
 - 3.17 Process 6
 - 3.75 Process 7
 - 3.37 Process 8
 - 3.19 Process 9
 - 3.29 Process 10
 - 3.15 Process 11
 - 3.35 Process 12
 - 3.45 Process 13
 - 3.43 Process 14
 - 3.19 Process 15

The bottom status bar shows a total execution time of 53.80 (28.42%) and 189.31. The system tree shows a total execution time of 0.00 and 53.80.

BT-MPI summary analysis score

- Summary measurement analysis score reveals
 - Total size of event trace would be over 500MB
 - Maximum trace buffer size would be over 30MB per process
 - smaller buffer would require flushes to disk during measurement resulting in substantial perturbation
 - 97% of the trace requirements are for USR regions
 - purely computational routines never found on COM call-paths common to communication routines
 - These USR regions contribute around 10% of total time
 - however, much of that is very likely to be measurement overhead for frequently-executed small routines
- Advisable to tune measurement configuration
 - Specify an adequate trace buffer size
 - Specify a filter file listing (USR) regions not to be measured

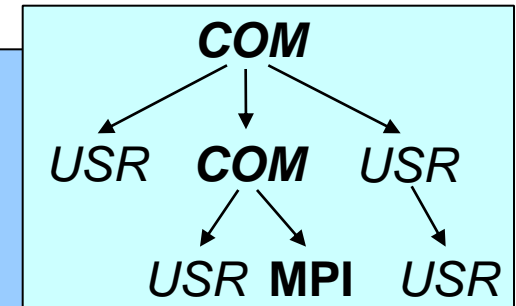
BT-MPI summary analysis report breakdown

- Report scoring with region breakdown

```
% cube3_score -r ./epik_bt_16_sum/summary.cube
```

flt	type	max_tbc	time	% region	
	ANY	32528680	220.22	100.00	(summary) ALL
	MPI	642712	194.57	88.35	(summary) MPI
	USR	31688040	24.62	11.18	(summary) USR
	COM	197928	1.03	0.47	(summary) COM
	USR	10231704	4.44	2.02	binvcrhs_
	USR	10231704	3.06	1.39	matvec_sub_
	USR	10231704	3.53	1.60	matmul_sub_
	USR	492048	0.16	0.07	binvrhs_
	USR	360576	0.12	0.05	exact_solution_
	MPI	241500	0.27	0.12	MPI_Isend
	MPI	222180	0.12	0.06	MPI_Irecv
	MPI	173664	173.02	78.57	MPI_Wait
	USR	57888	0.06	0.03	lhsabinit_
	USR	19296	3.53	1.60	y_solve_cell_

...



BT-MPI summary analysis report filtering

- Report scoring with filter listing 6 USR regions

```
% cube3_score -r -f npb.filt ./epik_bt_16_sum/summary.cube
Applying filter "./npb.filt":
Estimated aggregate size of event trace (total_tbc): 16,852,888 bytes
Estimated size of largest process trace (max_tbc): 1,053,304 bytes
```

flt	type	max_tbc	time	% region
+	FLT	31475376	11.37	5.16 (summary) FLT
*	ANY	1053328	208.85	94.84 (summary) ALL-FLT
-	MPI	642712	194.57	88.35 (summary) MPI-FLT
*	USR	212688	13.25	6.02 (summary) USR-FLT
*	COM	197928	1.03	0.47 (summary) COM-FLT
+	USR	10231704	4.44	2.02 binvrhs_
+	USR	10231704	3.06	1.39 matvec_sub_
+	USR	10231704	3.53	1.60 matmul_sub_
+	USR	492048	0.16	0.07 binvrhs_
+	USR	360576	0.12	0.05 exact_solution_
-	MPI	241500	0.27	0.12 MPI_Isend
-	MPI	222180	0.12	0.06 MPI_Irecv
-	MPI	173664	173.02	78.57 MPI_Wait
+	USR	57888	0.06	0.03 lhsabinit_

```
% cat npb.filt
# filter for bt
binvrhs_
matvec_sub_
matmul_sub_
binvrhs_
exact_solution_
lhsabinit_
```

BT-MPI filtered summary measurement

- Rename former measurement archive directory, set new filter configuration and re-run the measurement

```
% mv epik_bt_16_sum epik_bt_16_sum.nofilt
% export EPK_FILTER=npb.filt
% scalasca -analyze mpiexec -np 16 ./bt.W.16
S=C=A=N: Scalasca 1.1 runtime summarization
S=C=A=N: ./epik_bt_16_sum experiment archive
S=C=A=N: Sun Mar 29 16:58:34 2009: Collect start
mpiexec -np 16 ./bt.W.16
[00000]EPIK: Created new measurement archive ./epik_bt_16_sum
[00000]EPIK: EPK_FILTER "npb.filt" filtered 6 of 96 functions
[00000]EPIK: Activated ./epik_bt_16_sum [NO TRACE] (0.071s)

[... Application output ...]

[00000]EPIK: Closing experiment ./epik_bt_16_sum
[00000]EPIK: 84 unique paths (84 max paths, 4 max frames, 0 unknowns)
[00000]EPIK: Unifying... done (0.014s)
[00000]EPIK: Collating... done (0.059s)
[00000]EPIK: Closed experiment ./epik_bt_16_sum (0.075s)
S=C=A=N: Sun Mar 29 16:58:41 2009: Collect done (status=0) 7s
S=C=A=N: ./epik_bt_16_sum complete.
```

BT-MPI tuned summary analysis report score

- Scoring of new analysis report as textual output

```
% scalasca -examine -s epik_bt_16_sum
INFO: Post-processing runtime summarization result...
cube3_score ./epik_bt_16_sum/summary.cube
Estimated aggregate size of event trace (total_tbc): 16,852,888 bytes
Estimated size of largest process trace (max_tbc): 1,053,328 bytes
(When tracing set ELG_BUFFER_SIZE to avoid intermediate flushes or
 reduce requirements using filter file listing names of USR regions.)

flt  type          max_tbc          time           % region
     ANY          1053328         98.39    100.00 (summary) ALL
     MPI           642712         86.83     88.25 (summary) MPI
     USR           212688          9.88     10.04 (summary) USR
     COM           197928          1.68      1.71 (summary) COM
```

- Significant reduction in runtime (measurement overhead)
 - Not only reduced time for USR regions, but MPI reduced too!
- Further measurement tuning (filtering) may be appropriate
 - e.g., use “timer_*” to filter timer_start_, timer_read_, etc.

BT-MPI trace measurement collection...

- Re-run the application using Scalasca nexus with “-t” flag

```
% scalasca -analyze -t mpiexec -np 16 ./bt.W.16
S=C=A=N: Scalasca trace collection and analysis
S=C=A=N: ./epik_bt_16_trace experiment archive
S=C=A=N: Sun Apr  5 18:50:57 2009: Collect start
mpiexec -np 16 ./bt.W.16
[00000]EPIK: Created new measurement archive ./epik_bt_16_trace
[00000]EPIK: EPK_FILTER "npb.filt" filtered 6 of 96 functions
[00000]EPIK: Activated ./epik_bt_16_trace [10000000 bytes] (0.051s)

[... Application output ...]

[00000]EPIK: Closing experiment ./epik_bt_16_trace [0.016GB] (max 1053310)
[00000]EPIK: Flushed 1053330 bytes to file ./epik_bt_16_trace/ELG/00000
[00000]EPIK: 84 unique paths (84 max paths, 4 max frames, 0 unknowns)
[00000]EPIK: Unifying... done (0.021s)
[00013]EPIK: Flushed 1053306 bytes to file ./epik_bt_16_trace/ELG/00013
...
[00001]EPIK: Flushed 1053306 bytes to file ./epik_bt_16_trace/ELG/00001
[00000]EPIK: 1flush=0.001GB@2.582MB/s, Pflush=0.015GB@35.458MB/s
[00000]EPIK: Closed experiment ./epik_bt_16_trace (0.178s)
S=C=A=N: Sun Apr  5 18:51:05 2009: Collect done (status=0) 8s
[... continued ...]
```

- Separate trace file per MPI rank written straight into new experiment directory ./epik_bt_16_trace

BT-MPI trace measurement ... analysis

- Continues with automatic (parallel) analysis of trace files

```
S=C=A=N: Sun Apr  5 18:51:05 2009: Analyze start
mpiexec -np 16 scout ./epik_bt_16_trace
SCOUT Copyright (c) 1998-2009 Forschungszentrum Juelich GmbH

Analyzing experiment archive ./epik_bt_16_trace

Reading definitions file ... done (0.563s).
Reading event trace files ... done (0.495s).
Preprocessing ... done (0.134s).
Analyzing event traces ... done (2.186s).
Writing CUBE report ... done (0.160s).

Total processing time      : 3.737s
Max. memory usage         : 7.000MB

S=C=A=N: Sun Apr  5 18:51:09 2009: Analyze done (status=0) 4s
S=C=A=N: ./epik_bt_16_trace complete.
```

- Produces trace analysis report in experiment directory

```
% scalasca -examine epik_bt_16_trace
INFO: Post-processing runtime summarization result...
INFO: Post-processing trace analysis report ...
INFO: Displaying ./epik_bt_16_sum/trace.cube...
```


Trace analysis report exploration

The screenshot displays the Cube 3.0 QT interface for trace analysis. The main window shows a metric tree on the left and a call tree on the right. The metric tree is expanded to show 'Late Sender' metrics, with 'Late Sender' highlighted in blue. A call tree entry '4.14 MPI_Wait' is also highlighted. An 'Online description' window is open, showing the definition of 'Late Sender Time' and a diagram illustrating a blocking receive operation. A blue callout box points to the 'Late Sender' metric in the tree, with the text 'Additional trace-based metrics in metric hierarchy'.

Online description

Late Sender Time

Description:
Refers to the time lost waiting caused by a blocking receive operation (e.g., MPI_Recv() or MPI_Wait()) that is posted earlier than the corresponding send operation.

The diagram shows a timeline with 'location' on the vertical axis and 'time' on the horizontal axis. An orange box labeled 'Send' is positioned above a yellow box labeled 'Recv'. A red double-headed arrow below the 'Recv' box indicates the duration of the blocking receive operation, which occurs before the 'Send' operation is completed.

Metric tree (Left):

- 0.00 Time
 - 9.60 Execution
 - 0.06 MPI
 - 0.00 Synchronization
 - 0.00 Collective
 - 0.07 Wait at Barrier
 - 0.02 Barrier Completion
- 64.98 Point-to-point
 - 18.50 Late Sender
 - 0.00 Late Receiver
 - 0.16 Collective
 - 0.00 Early Reduce
 - 0.00 Early Scan
 - 0.00 Late Broadcast
 - 0.05 Wait at N x N
 - 0.01 N x N Completion
- 0.00 File I/O
- 3.89 Init/Exit
- 0.83 Overhead
- 5.48e5 Visits

Call tree (Right):

- 0.00 exact_ms_
- 0.00 compute_buffer_size_
- 0.00 adi_
- 7.01 copy_faces_
- 4.18 x_solve_
- 3.14 y_solve_
- 0.00 z_solve_
- 0.00 z_solve_cell_
- 0.00 z_send_solve_info_
- 0.00 z_receive_solve_info_
- 4.14 MPI_Wait
- 0.00 z_unpack_solve_info_
- 0.00 z_backsubstitute_

Process list (Bottom Right):

- 0.10 Process 6
- 0.21 Process 7
- 0.23 Process 8
- 0.48 Process 9
- 0.22 Process 10
- 0.14 Process 11
- 0.59 Process 12
- 0.24 Process 13
- 0.31 Process 14
- 0.26 Process 15

Summary (Bottom):

0.00	18.50 (18.84%)	98	4.14 (22.38%)	18.50	0.00	4.14
------	----------------	----	---------------	-------	------	------

Further information

- Consult quick reference guide for further information

```
% scalasca -h  
Scalasca 1.1 - quick reference guide  
pdfview /UNITE/packages/scalasca/1.1/doc/manuals/quickref.pdf
```

[PDF viewer showing quick reference guide]

- CUBE GUI provides context sensitive help and on-line metric descriptions
- EPIK archive directories contain analysis report(s), measurement collection & analysis logs, etc.
- Instrumentation, measurement, analysis & presentation can all be extensively customized
- Visit www.scalasca.org or mail scalasca@fz-juelich.de

EPIK user instrumentation API

- EPIK user instrumentation API
 - #include “epik_user.h”
 - EPIK_USER_REG(epik_solve, “<<Solve>>”)
 - EPIK_USER_START(epik_solve)
 - EPIK_USER_END(epik_solve)
- Can be used to mark initialization, solver & other phases
 - Annotation macros ignored by default
 - Instrumentation enabled with “-user” flag
 - Also available for Fortran
 - #include “epik_user.inc” and use C preprocessor
- Appear as additional regions in analyses
 - Distinguishes performance of important phase from rest

EPIK measurement configuration

- Via ./EPIK.CONF file

```
EPK_FILTER=smg2000.filt  
ELG_BUFFER_SIZE=40000000
```

- Via environment variables

```
% export EPK_FILTER=smg2000.filt  
% export ELG_BUFFER_SIZE=40000000
```

- Via command-line flags (partially)

```
% scalasca -analyze -f smg2000.filt ...
```

- To show current/default configuration

```
% epik_conf
```

- Actual Scalasca measurement configuration saved in experiment archive as `epik.conf`

CUBE algebra utilities

- Extracting solver sub-tree from analysis report

```
% cube3_cut -r '<<SMG.Solve>>' epik_smg2000_12_trace/trace.cube  
Writing cut.cube... done.
```

- Calculating difference of two reports

```
% cube3_diff epik_smg2000_12_trace/trace.cube cut.cube  
Writing diff.cube... done.
```

- Additional utilities for merging, calculating mean, etc.
 - Default output of `cube3_utility` is a new report `utility.cube`
- Further utilities for report scoring & statistics
- Run utility with “-h” (or no arguments) for brief usage info

Scalasca usage recap

1. Reference preparation for validation
2. Program instrumentation: `skin`
3. Summary measurement collection & analysis: `scan [-s]`
4. Summary analysis report examination: `square`
5. Summary experiment scoring: `square -s`
6. Event trace collection & analysis: `scan -t`
7. Event trace analysis report examination: `square`
 - General usage/help: `scalasca [-h]`
 - Instrumentation, measurement, analysis & presentation can all be extensively customized
 - Visit www.scalasca.org or mail scalasca@fz-juelich.de

skin – Scalasca application instrumenter

- Prepares application objects & executables for measurement
 - skin = scalasca -instrument
 - skin [options] *<compile-or-link-command>*
 - defaults to automatic function instrumentation by compiler
 - available for most compilers, but not all
 - for OpenMP, includes source-level pre-processing of directives to insert POMP instrumentation
 - [-pomp]
 - source-level pre-processing of OpenMP & POMP directives **instead** of automatic compiler instrumentation
 - [-user]
 - additionally enable EPIK user instrumentation API
 - offers complementary program structure information for analyses via user-provided annotations (e.g., phases, loops, ...)

scan – Scalasca measurement collection/analysis

- Runs application under control of measurement system to collect and analyze an execution experiment
 - scan = scalasca -analyze
 - scan [options] <application-launch-command>
 - e.g., scan [options] [\$MPIEXEC [mpiexec-options]] [target [args]]
 - [-s] collect summarization experiment [default]
 - [-t] collect event traces and then analyze them automatically
 - Additional options
 - [-e] experiment archive (directory) name
 - [-f filter] specify file listing routines to ignore during measurement
 - [-m metric1:metric2:...] include hardware counter metrics
 - [-n] preview scan and perform checks but don't execute
 - [-q] quiesce (disable most) measurement collection
 - [-a] (re-)analyze a previously collected experiment

square – Scalasca analysis report examiner

- Prepares and presents measurement analysis report(s) for interactive exploration
 - square = scalasca -examine
 - square [options] <experiment-archive/report>
 - e.g., square epik_title
 - Post-processes intermediate measurement analysis reports
 - Launches GUI and presents default analysis report (if multiple reports available)
 - trace analysis given precedence over summary analysis
 - select other reports via File/Open menu
 - [-s] skip display and output textual score report
 - estimate total trace size and maximum rank trace size
 - breakdown of USR vs. MPI/OMP vs. COM region requirements

Performance analysis & tuning case studies



Additional Live-DVD example experiments

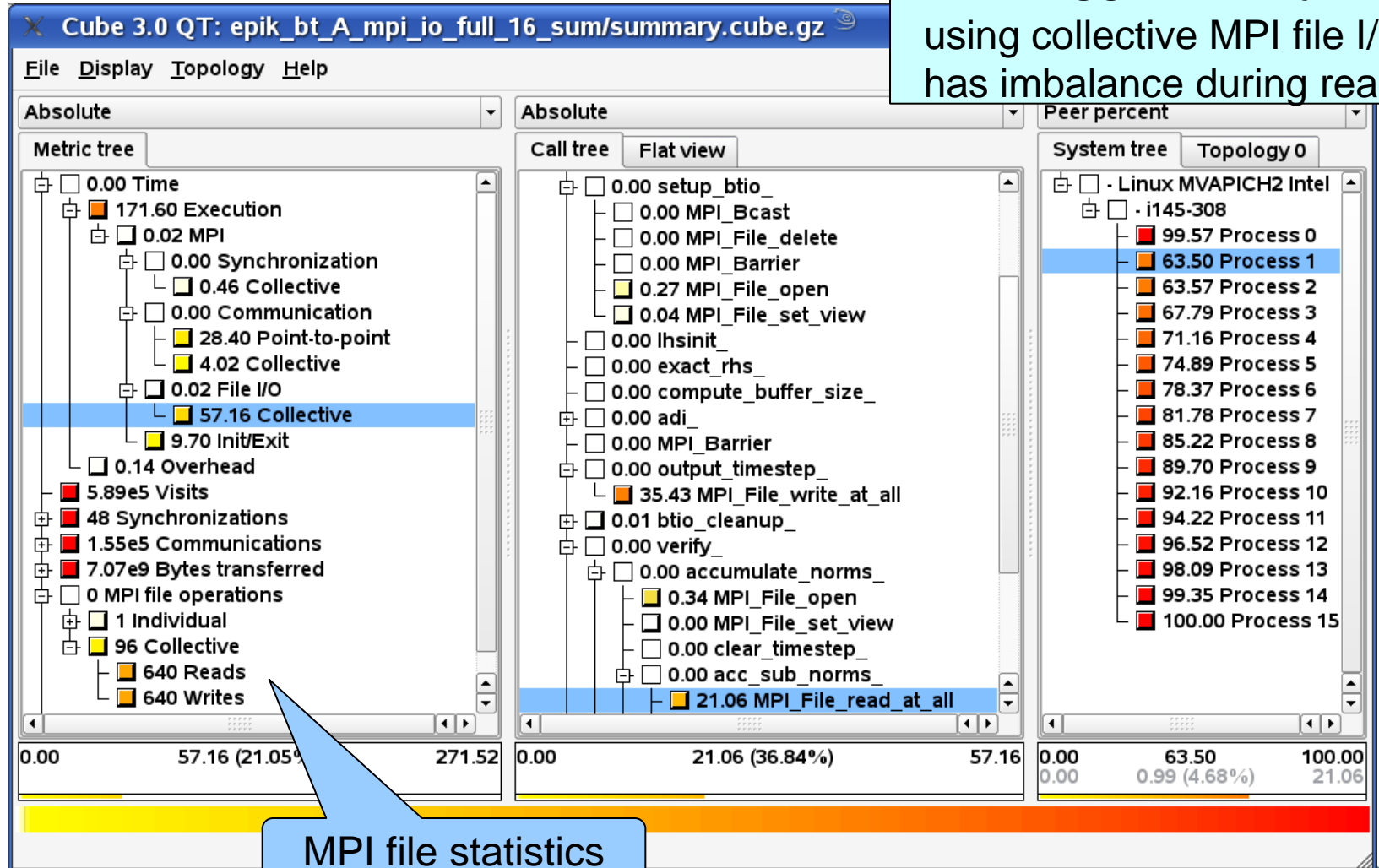
- Example experiment archives provided for examination:
 - jugene_sweep3d
 - 65,536 MPI processes on BG/P (trace)
 - jump_zeusmp2
 - 512 MPI processes on p690 cluster (summary & trace)
 - marenostrium_wrf-nmm
 - 1600 MPI processes on JS21 blade cluster, solver extract
 - summary analysis with 8 PowerPC hardware counters
 - trace analysis showing NxN completion problem on some blades
 - neptun_jacobi
 - 12 MPI processes, or 12 OpenMP threads, or 4x3 hybrid parallelizations implemented in C, C++ & Fortran on SGI Altix
 - ranger_smg2000
 - 12,288 MPI processes on Sun Constellation cluster, solve extract

Scalasca NPB-BT experiments

- Comparison of NPB-BT class A in various configurations run on a single dedicated 16-core cluster compute node
 - 16 MPI processes
 - optionally built using MPI File I/O (e.g., SUBTYPE=full)
 - optionally including PAPI counter metrics in measurement (e.g., EPK_METRICS=PAPI_FP_OPS:DISPATCH_STALLS)
 - 16 OpenMP threads
 - 4 MPI processes each with 4 OpenMP threads (MZ-MPI)
- NPB-BT-MZ class B on Cray XT5 (8-core compute nodes)
 - 32 MPI processes with OMP_NUM_THREADS=8
 - More threads created on some processes (and fewer on others) as application attempts to balance work distribution

16-process summary analysis: MPI File I/O time

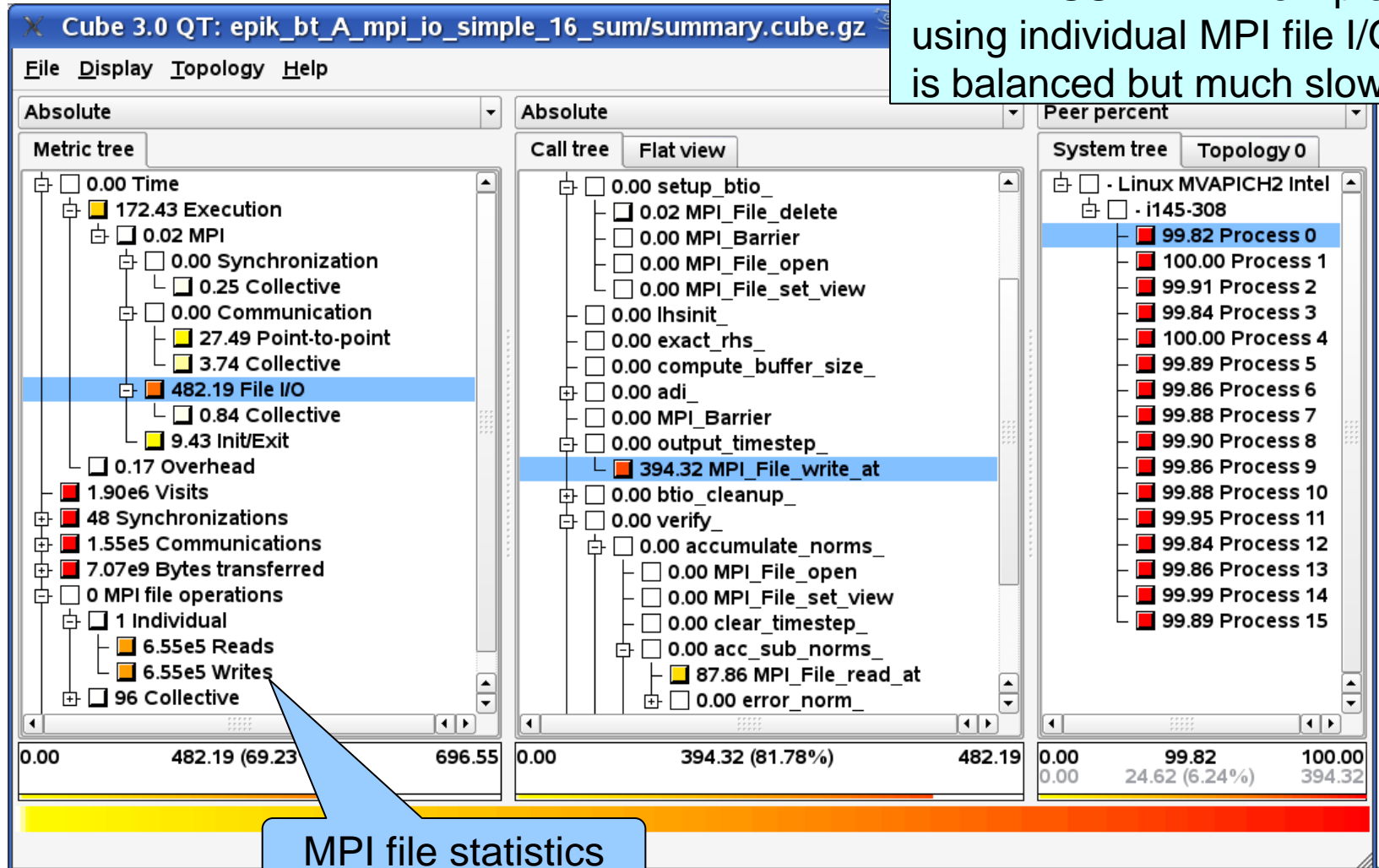
BT-MPI SUBTYPE=full
using collective MPI file I/O
has imbalance during read



MPI file statistics

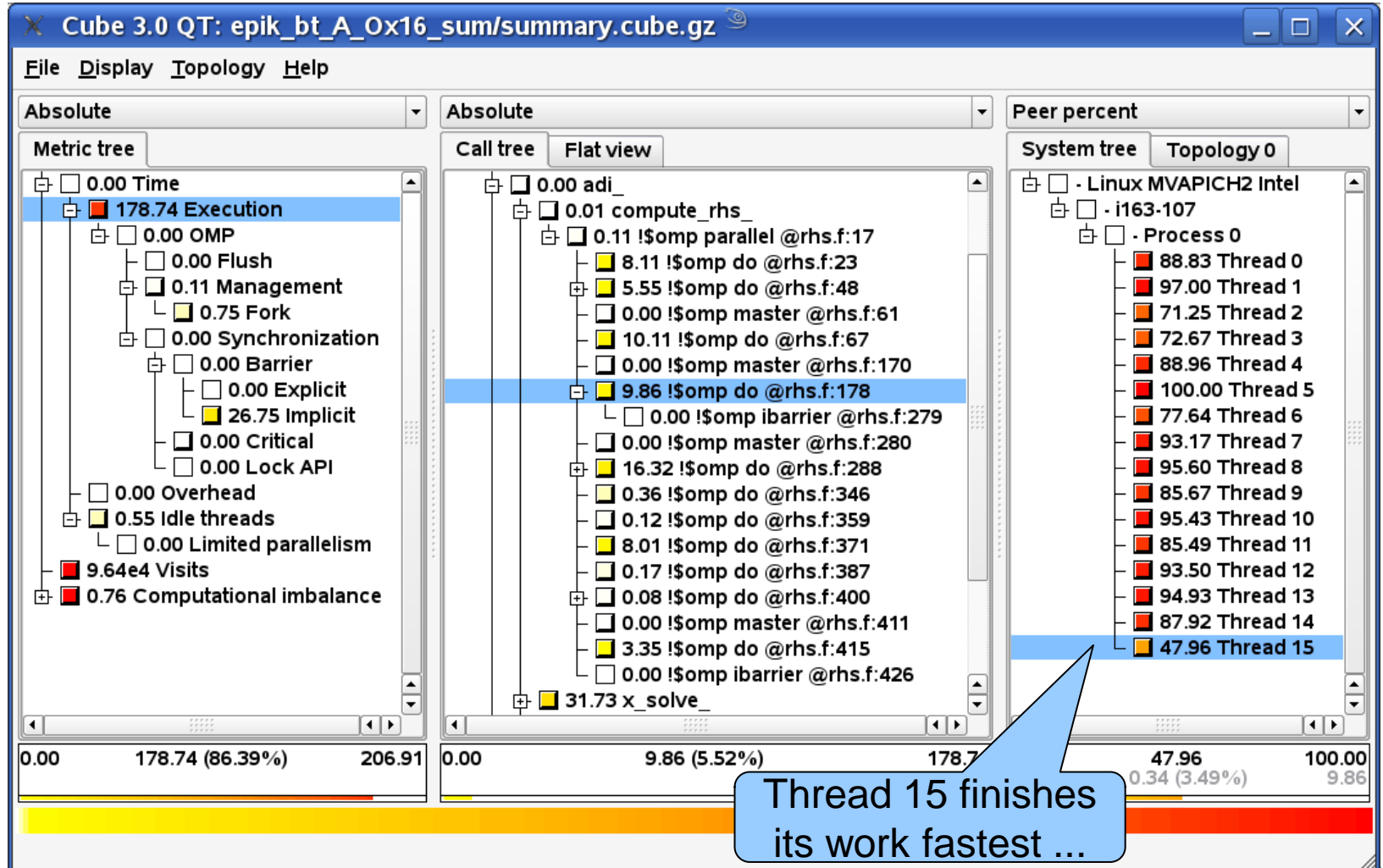
16-process summary analysis: MPI File I/O time

BT-MPI SUBTYPE=simple using individual MPI file I/O is balanced but much slower

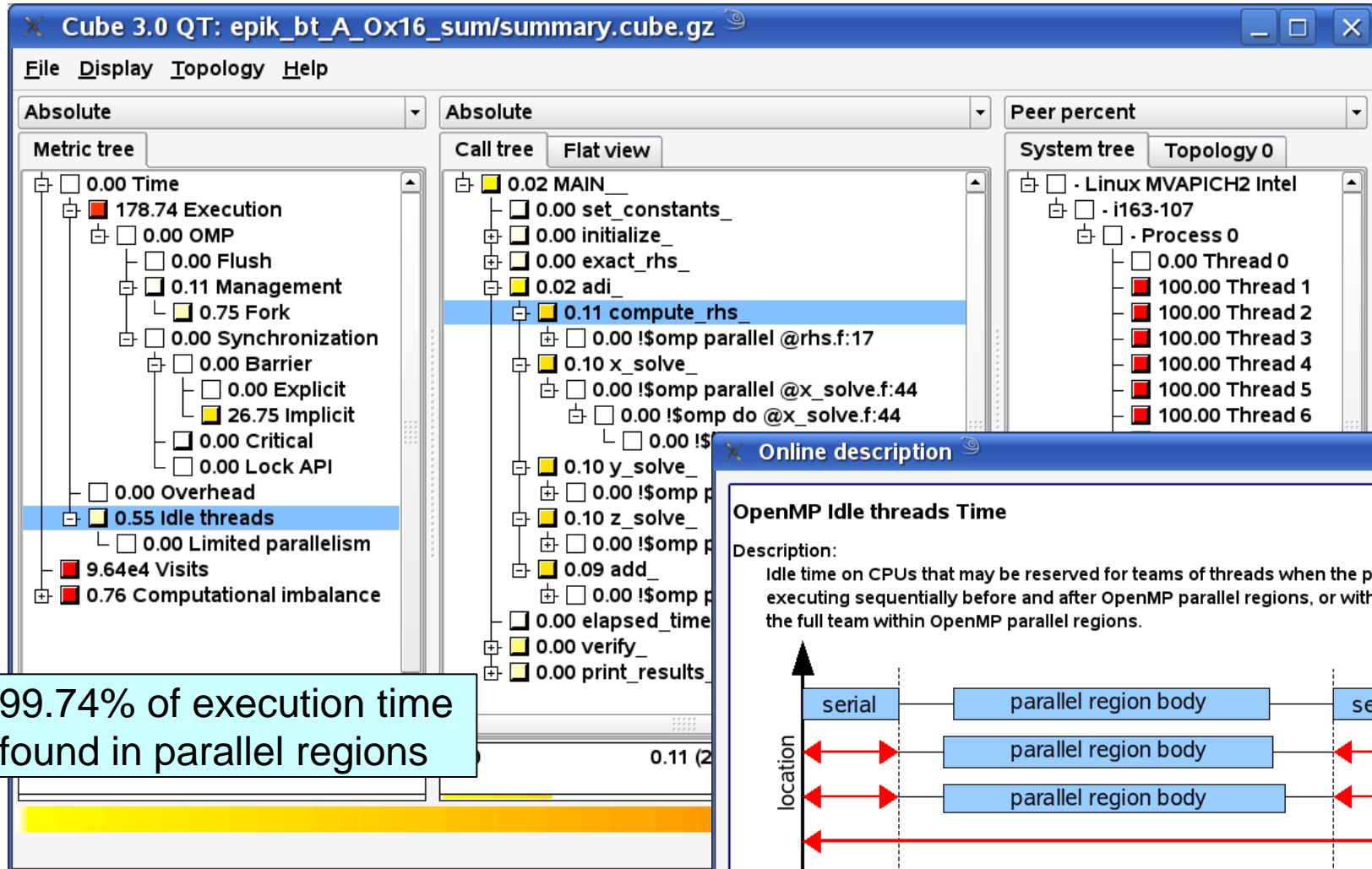


MPI file statistics

16-thread summary analysis: Execution time

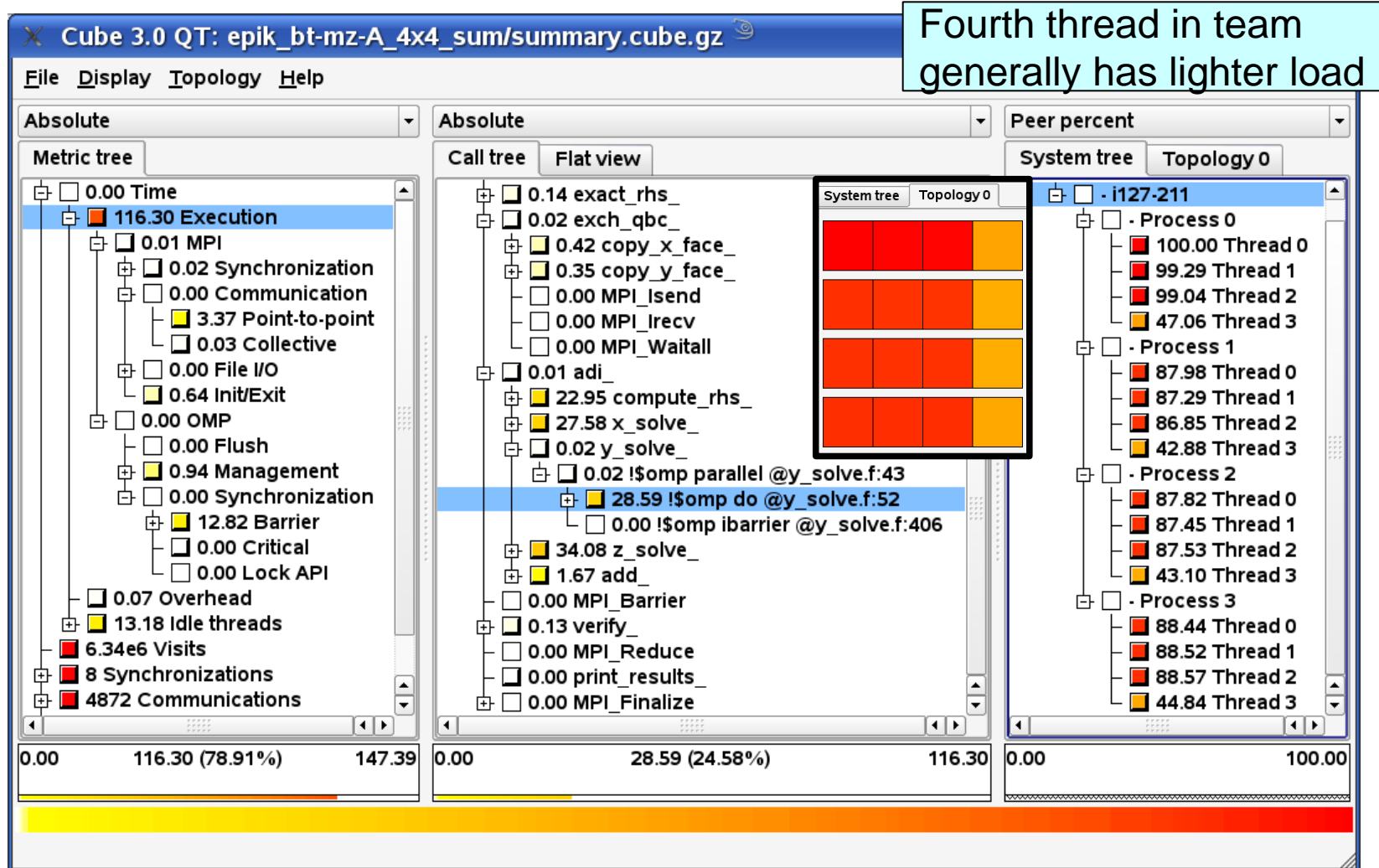


16-thread summary analysis: Idle threads time



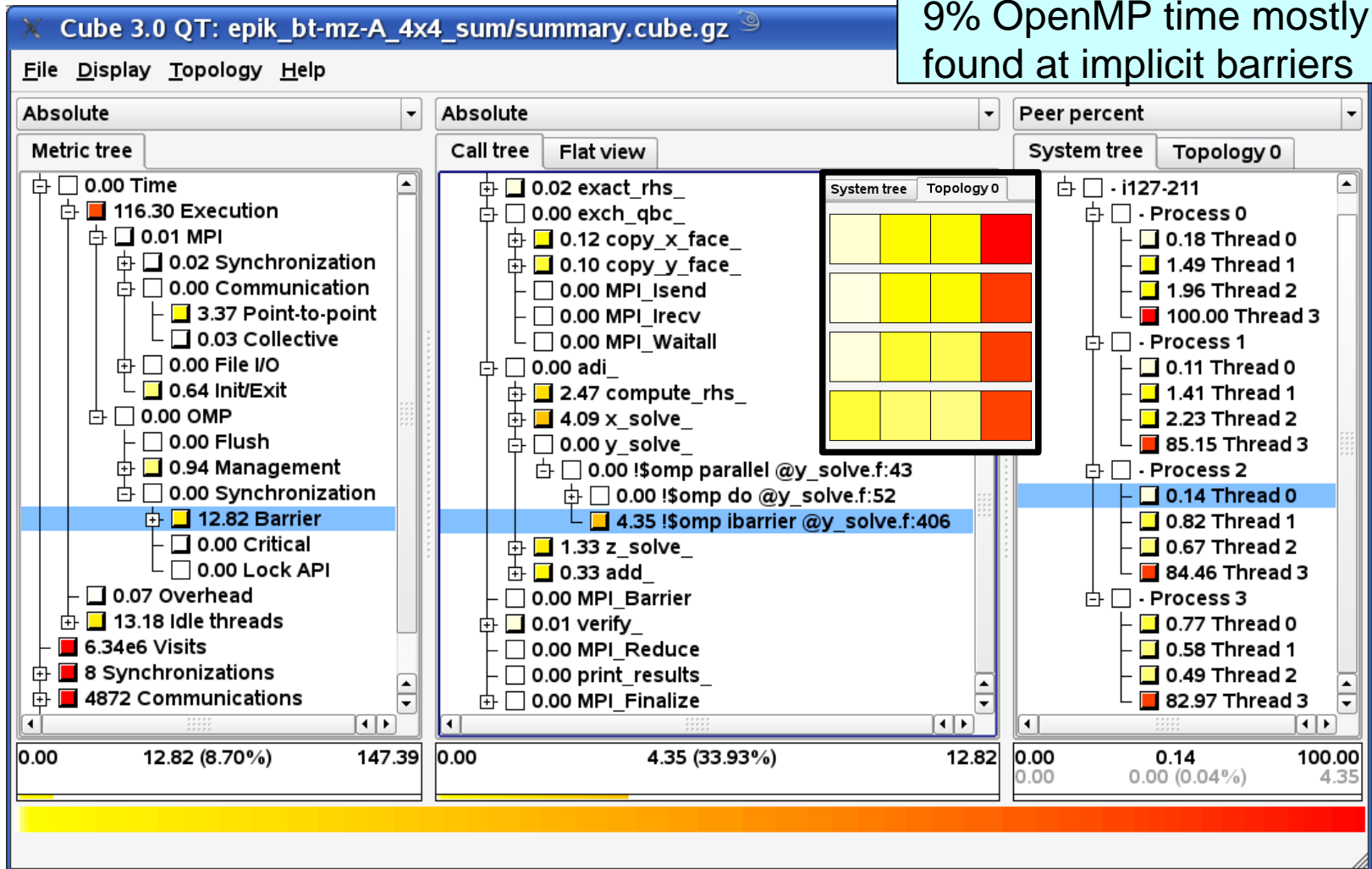
99.74% of execution time found in parallel regions

4x4 summary analysis: Execution time



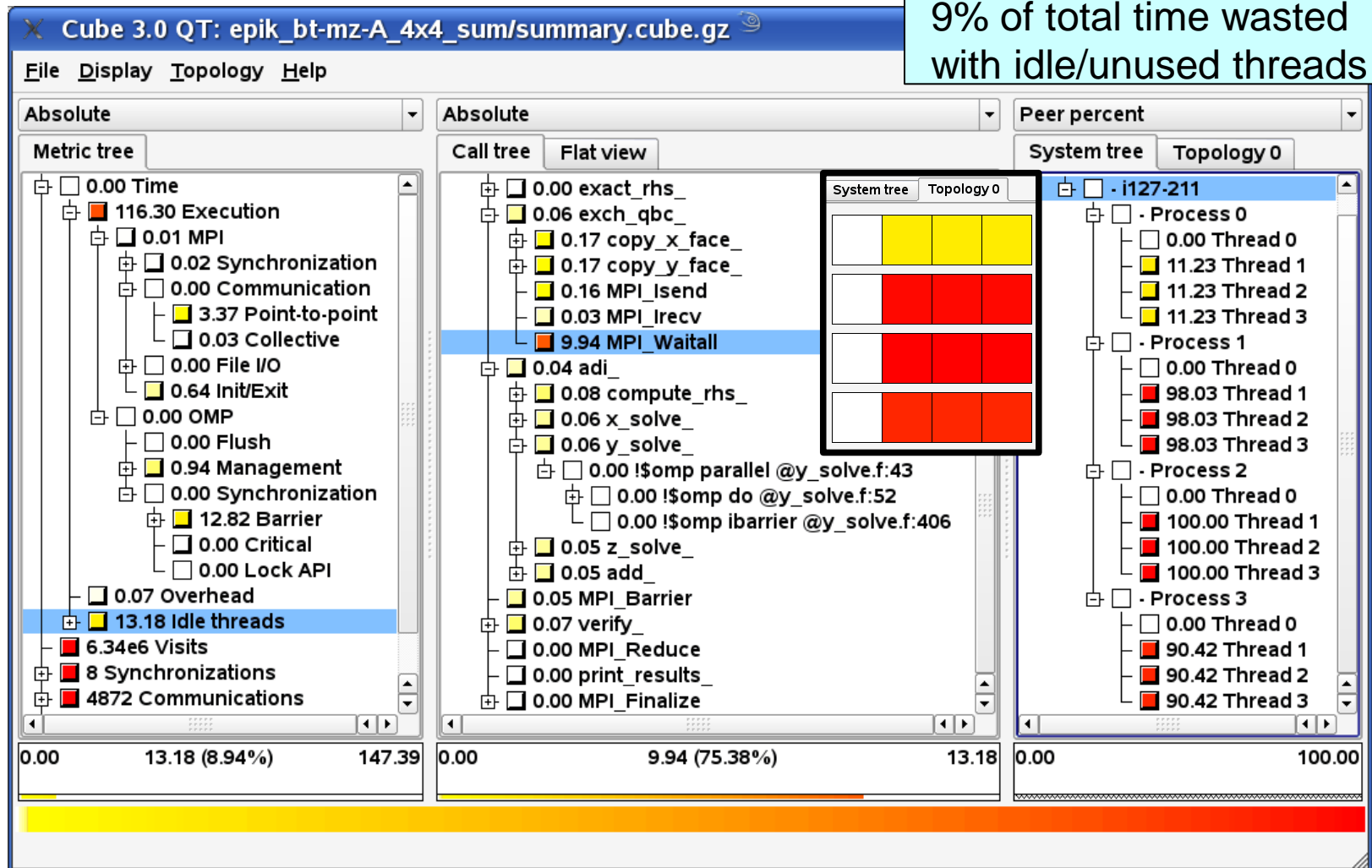
4x4 summary analysis: OpenMP time

9% OpenMP time mostly found at implicit barriers

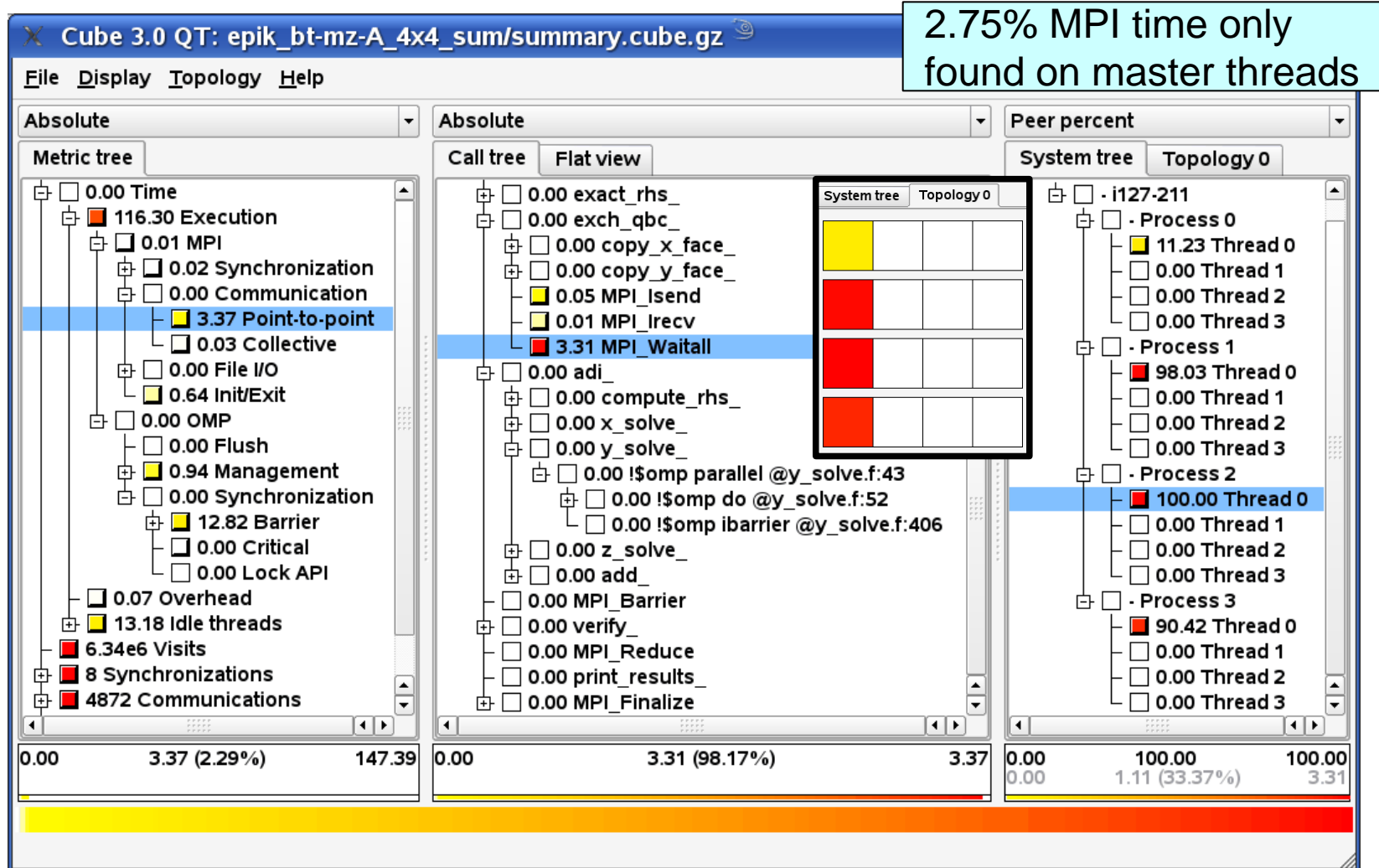


4x4 summary analysis: Idle threads time

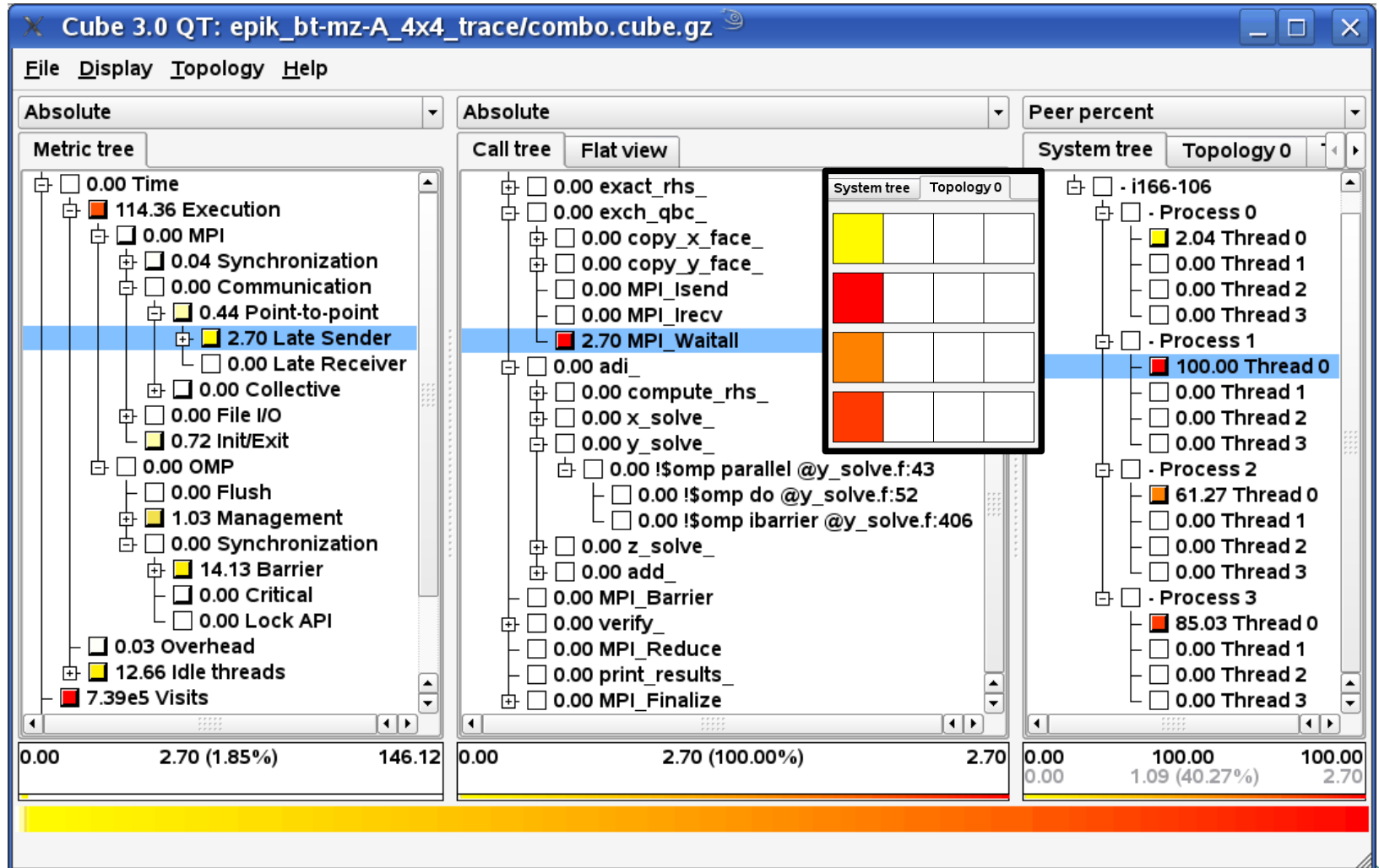
9% of total time wasted with idle/unused threads



4x4 summary analysis: MPI time

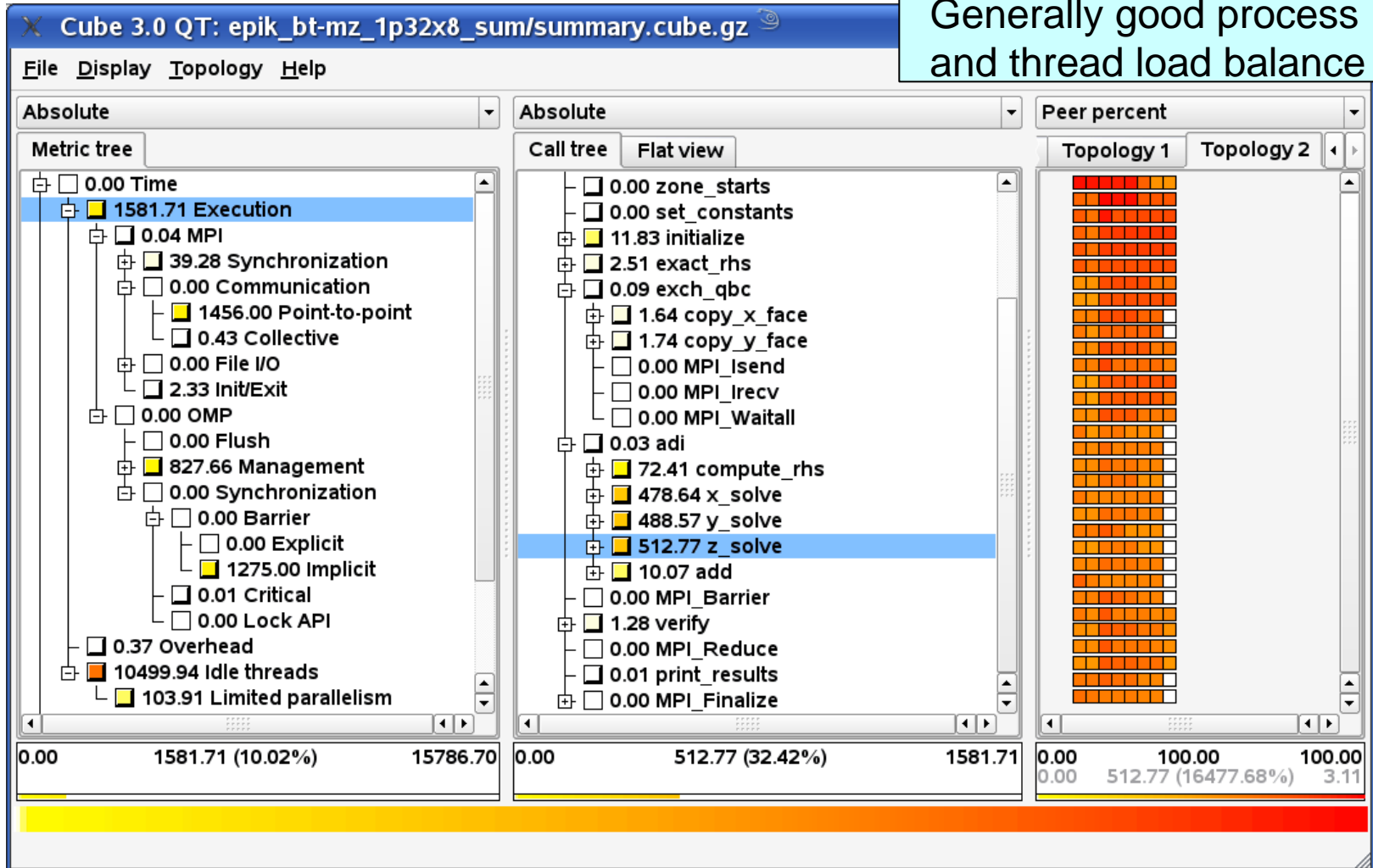


4x4 combined summary & trace analysis

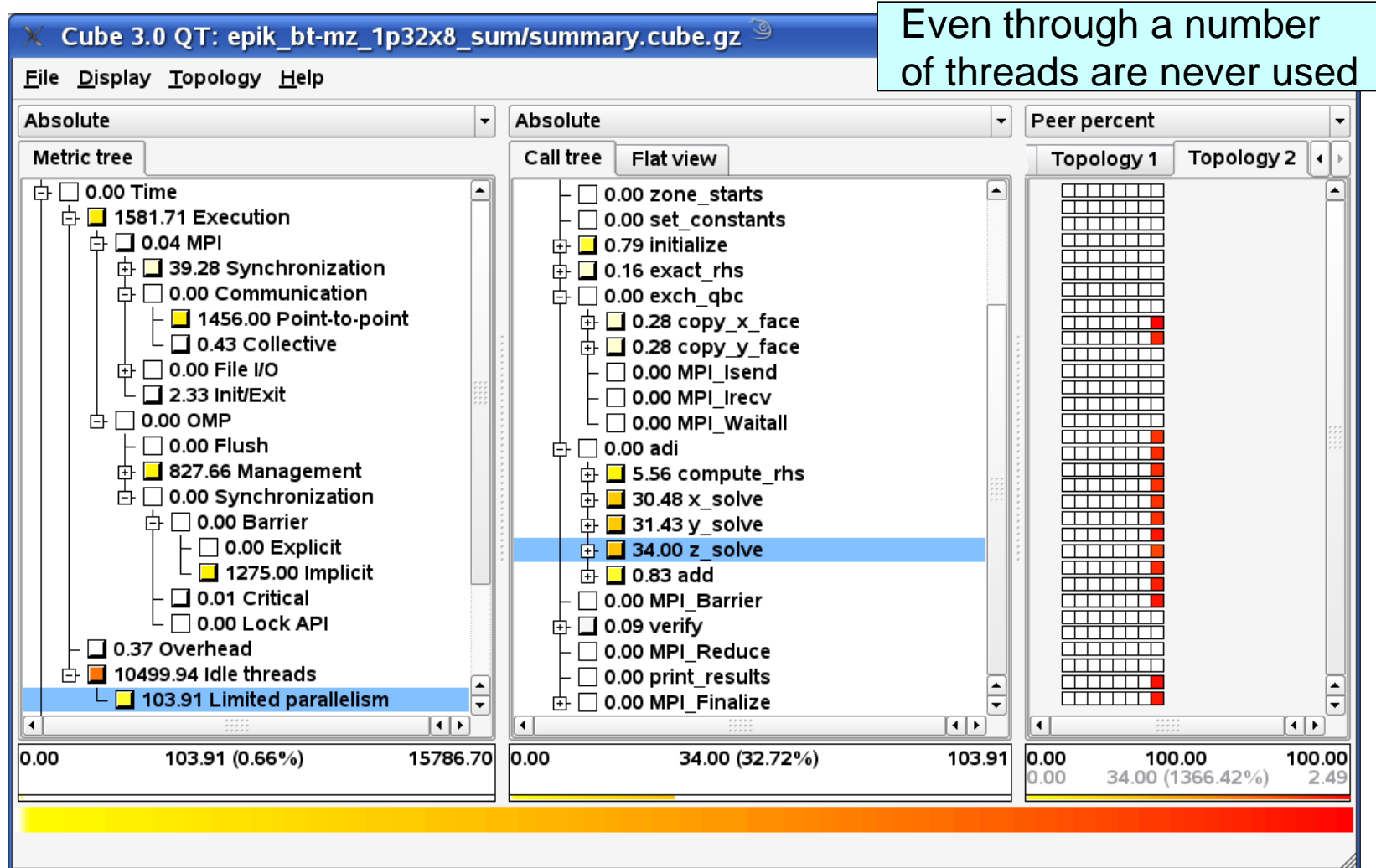


32x8 summary analysis: Excl. execution time

Generally good process and thread load balance

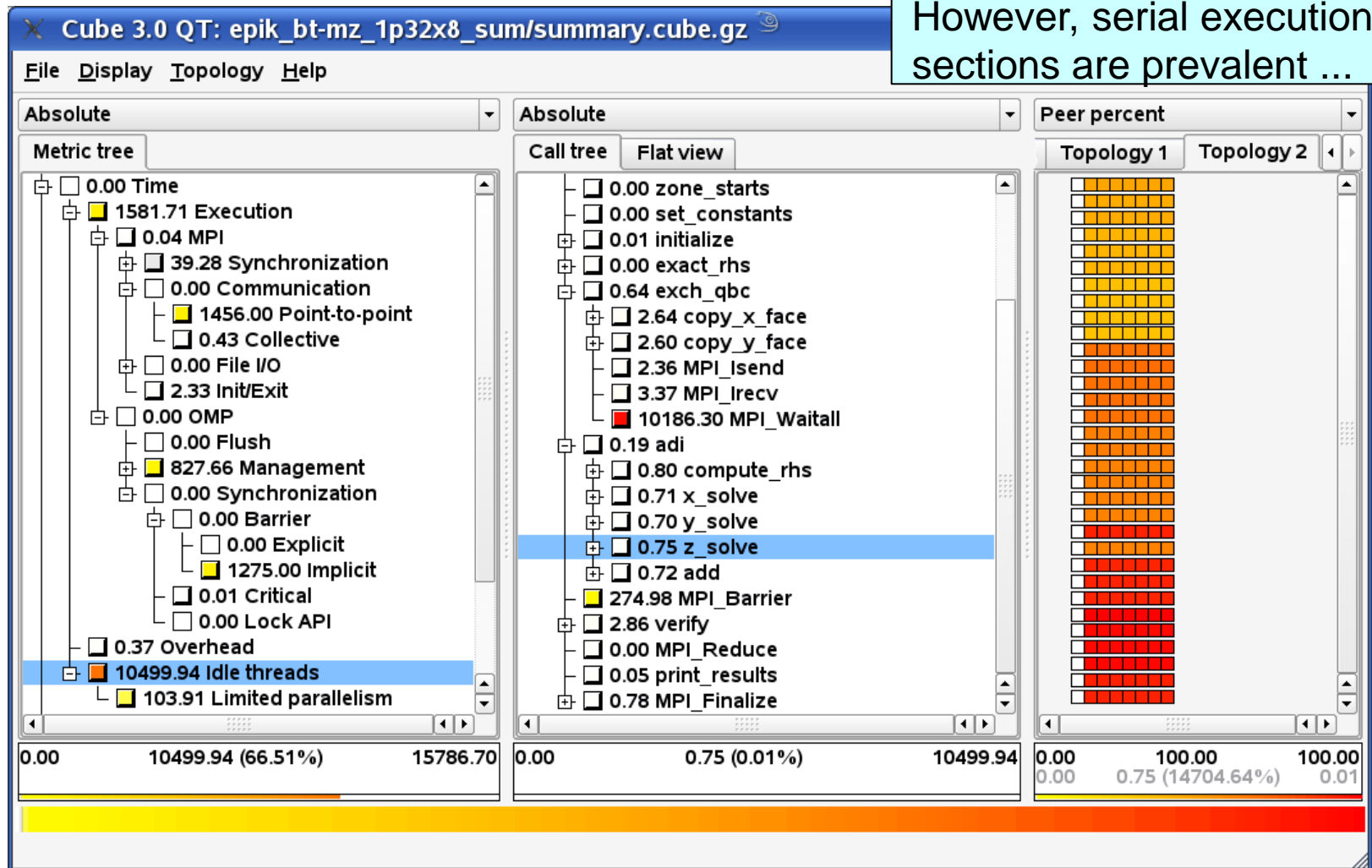


32x8 summary analysis: Limited parallelism

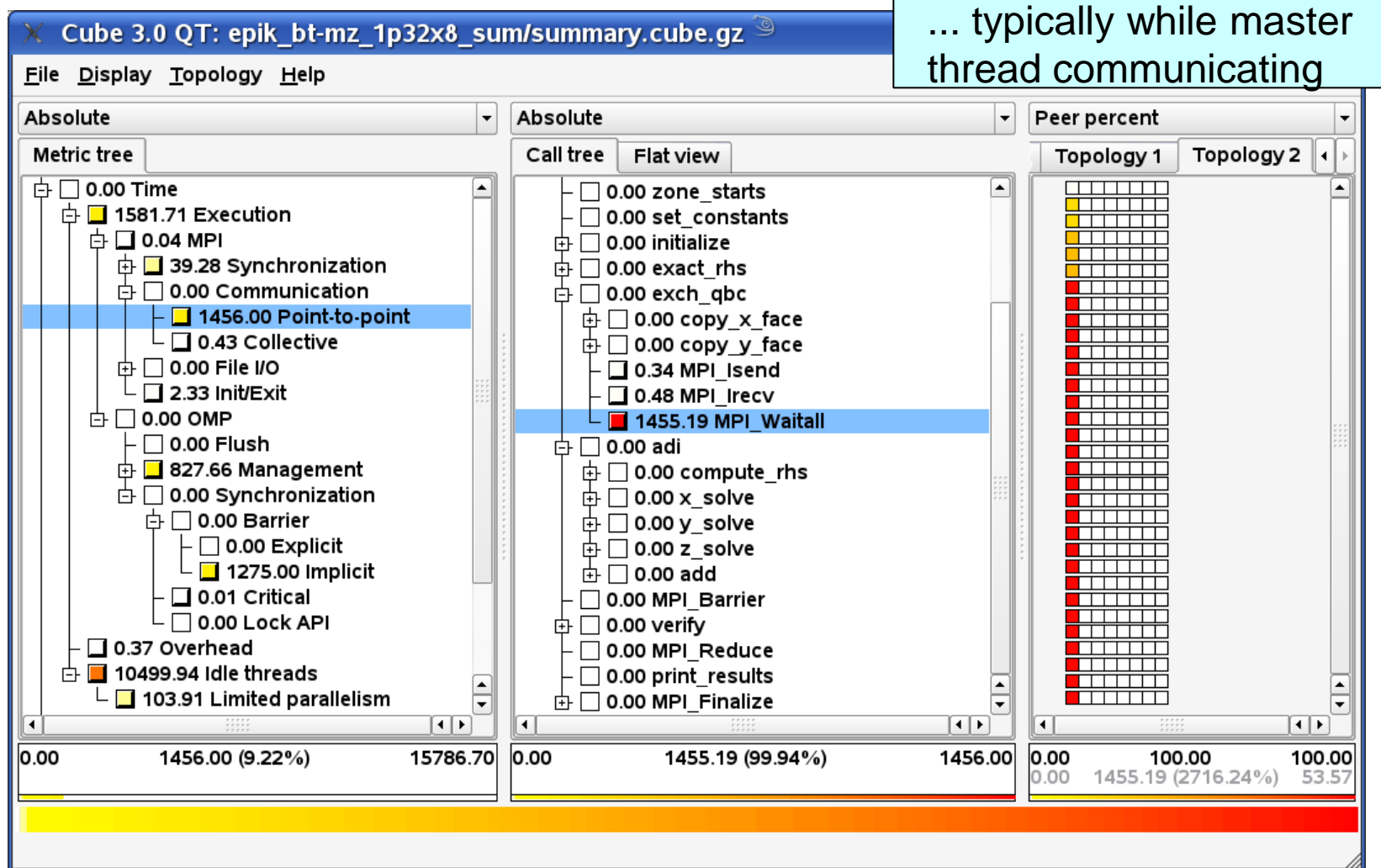


32x8 summary analysis: Idle threads time

However, serial execution sections are prevalent ...

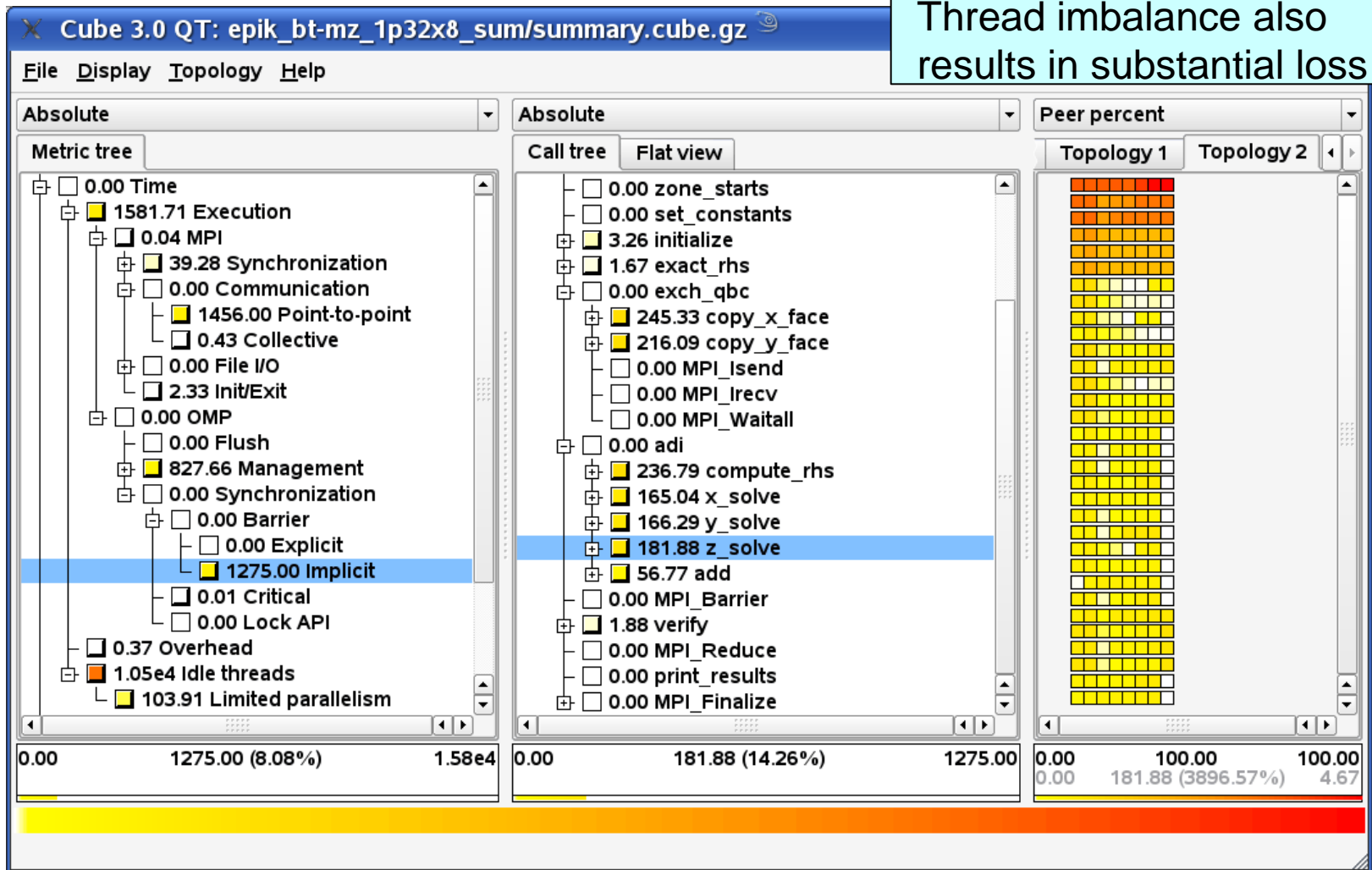


32x8 summary analysis: MPI communication time

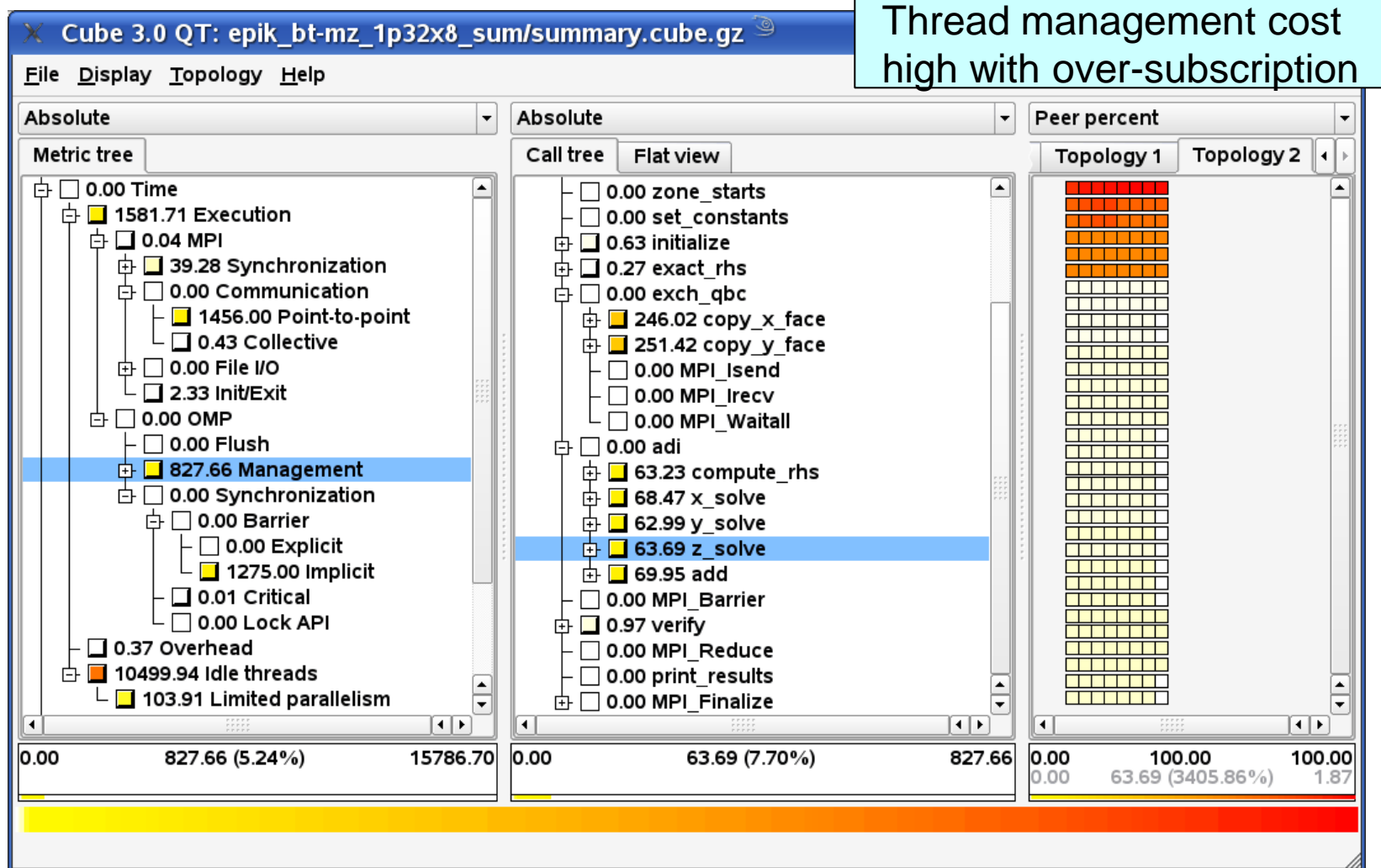


32x8 summary analysis: Implicit barrier time

Thread imbalance also results in substantial loss

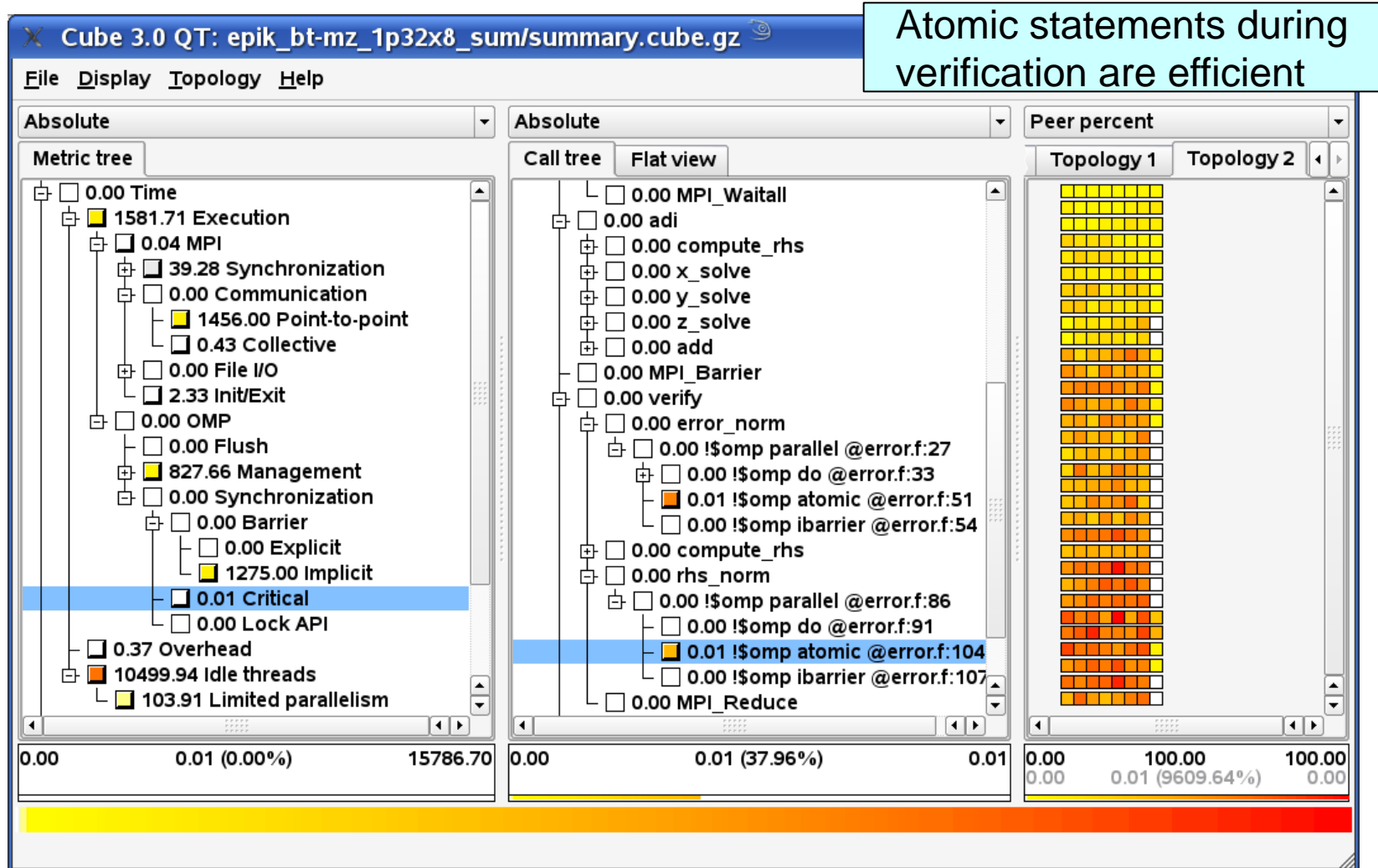


32x8 summary analysis: Thread management



Thread management cost high with over-subscription

32x8 summary analysis: Critical section time



Atomic statements during verification are efficient

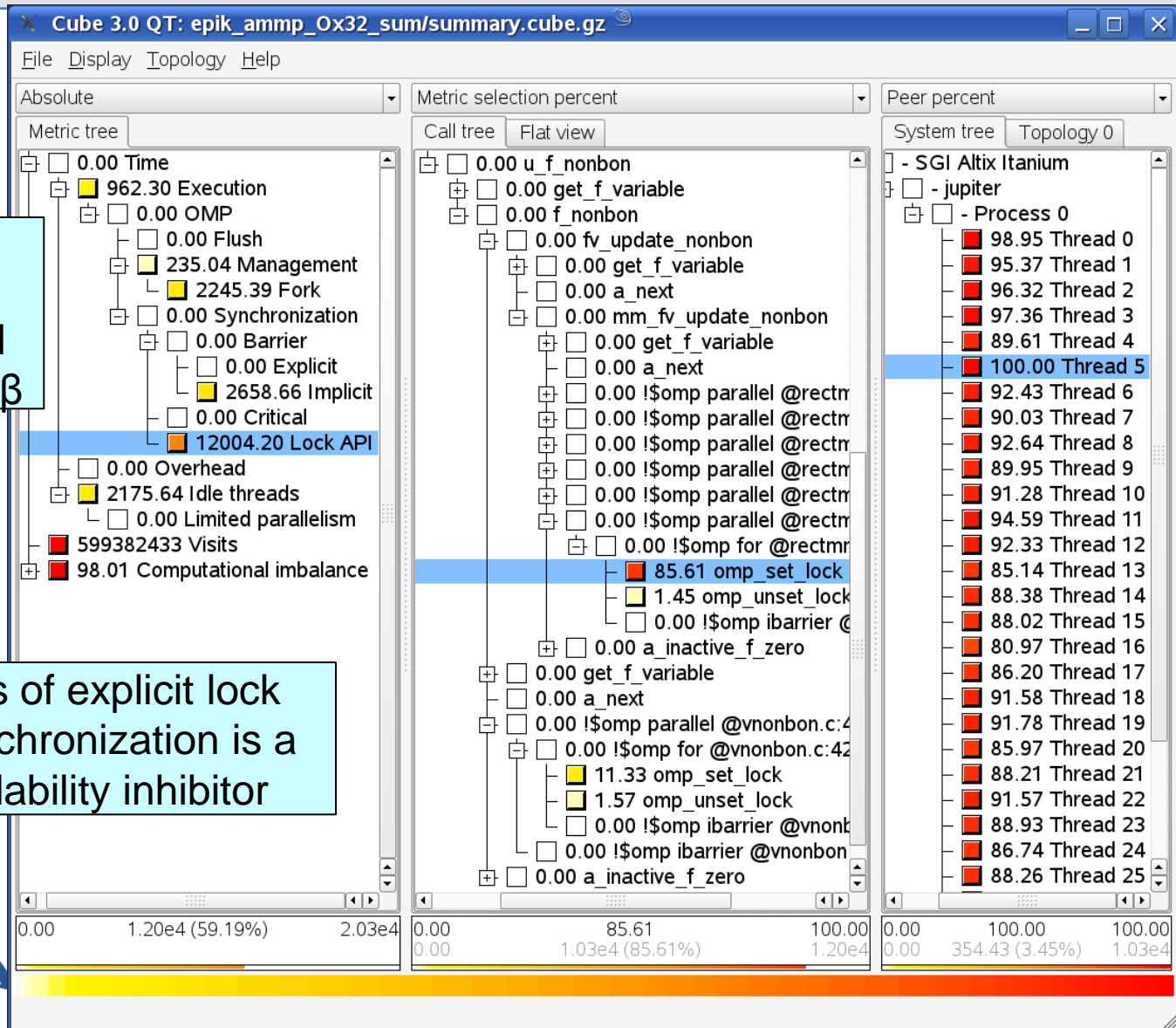
AMMP on Altix case study

- Molecular mechanics simulation
 - original version developed by Robert W. Harrison
- SPEC OMP benchmark parallel version
 - ~14,000 lines (in 28 source modules): 100% C
- Run with 32 threads on SGI Altix 4700 at TUD-ZIH
 - Built with Intel compilers
 - 333 simulation timesteps for 9582 atoms
- Scalasca summary measurement
 - Minimal measurement dilation
 - 60% of total time lost in synchronization with lock API
 - 12% thread management overhead

ammp on jupiter@32 OpenMP lock analysis

OpenMP metrics reworked with v1.2β

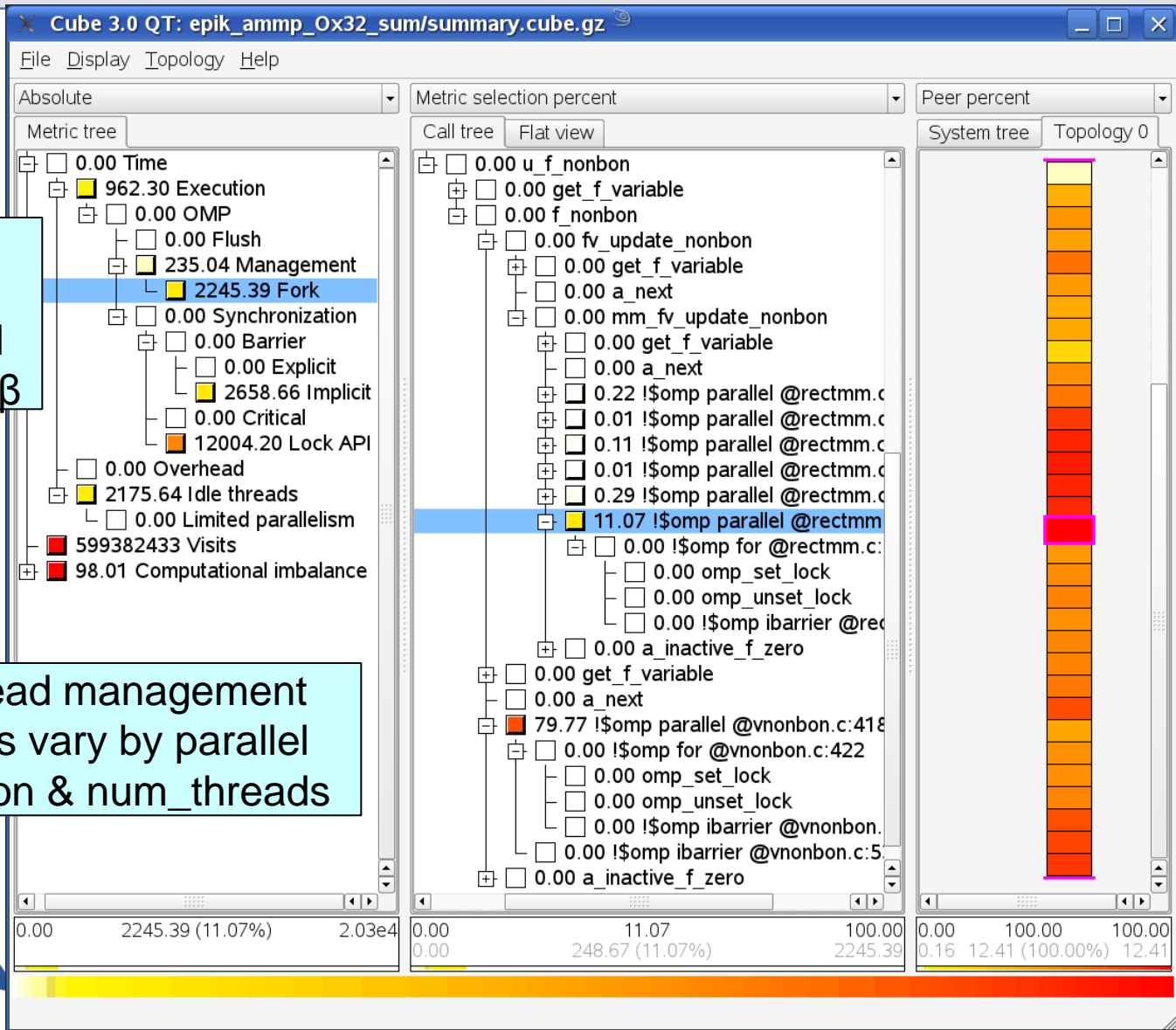
Lots of explicit lock synchronization is a scalability inhibitor



ammp on jupiter@32 OpenMP fork analysis

OpenMP metrics reworked with v1.2β

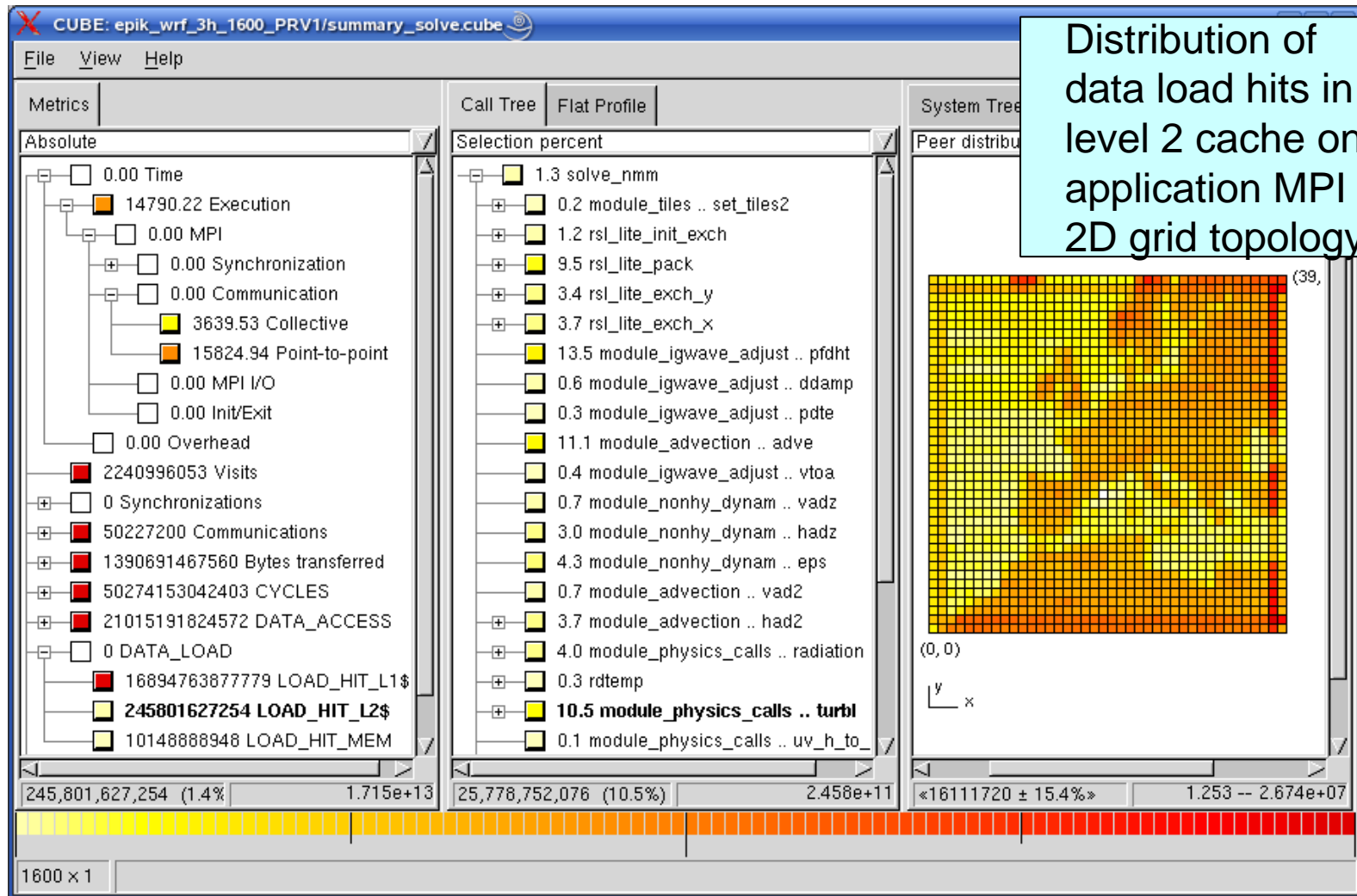
Thread management costs vary by parallel region & num_threads



WRF/MareNostrum case study

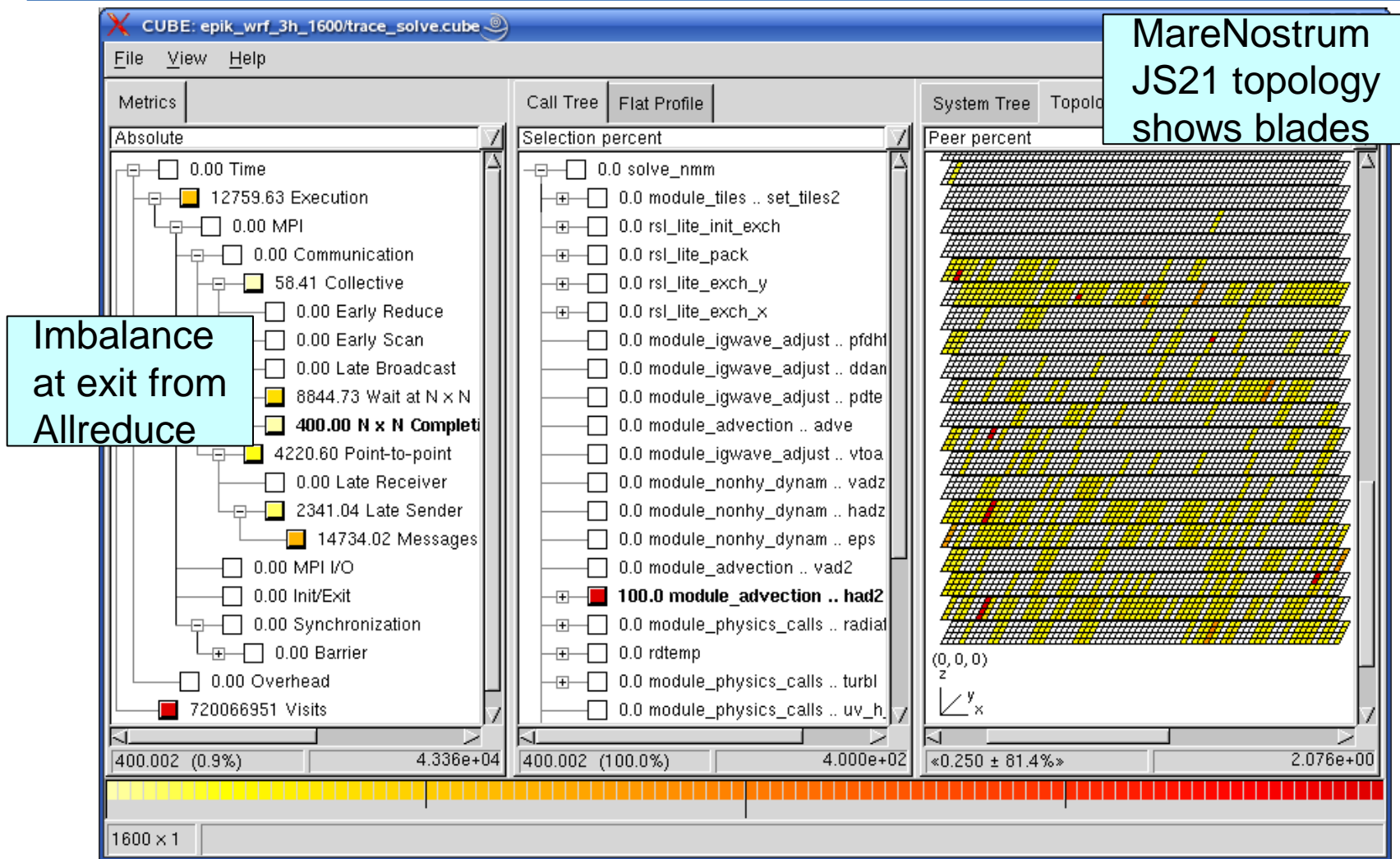
- Numerical weather prediction
 - public domain code developed by US NOAA
 - flexible, state-of-the-art atmospheric simulation
 - Non-hydrostatic Mesoscale Model (NMM)
- MPI parallel version 2.1.2 (Jan-2006)
 - >315,000 lines (in 480 source modules): 75% Fortran, 25% C
- Eur-12km dataset configuration
 - 3-hour forecast (360 timesteps) with checkpointing disabled
- Run with 1600 processes on MareNostrum
 - IBM BladeCenter cluster at BSC
- Scalasca summary and trace measurements
 - 15% measurement dilation with 8 hardware counters
 - 23GB trace analysis in 5 mins

WRF on MareNostrum@1600 with HWC metrics

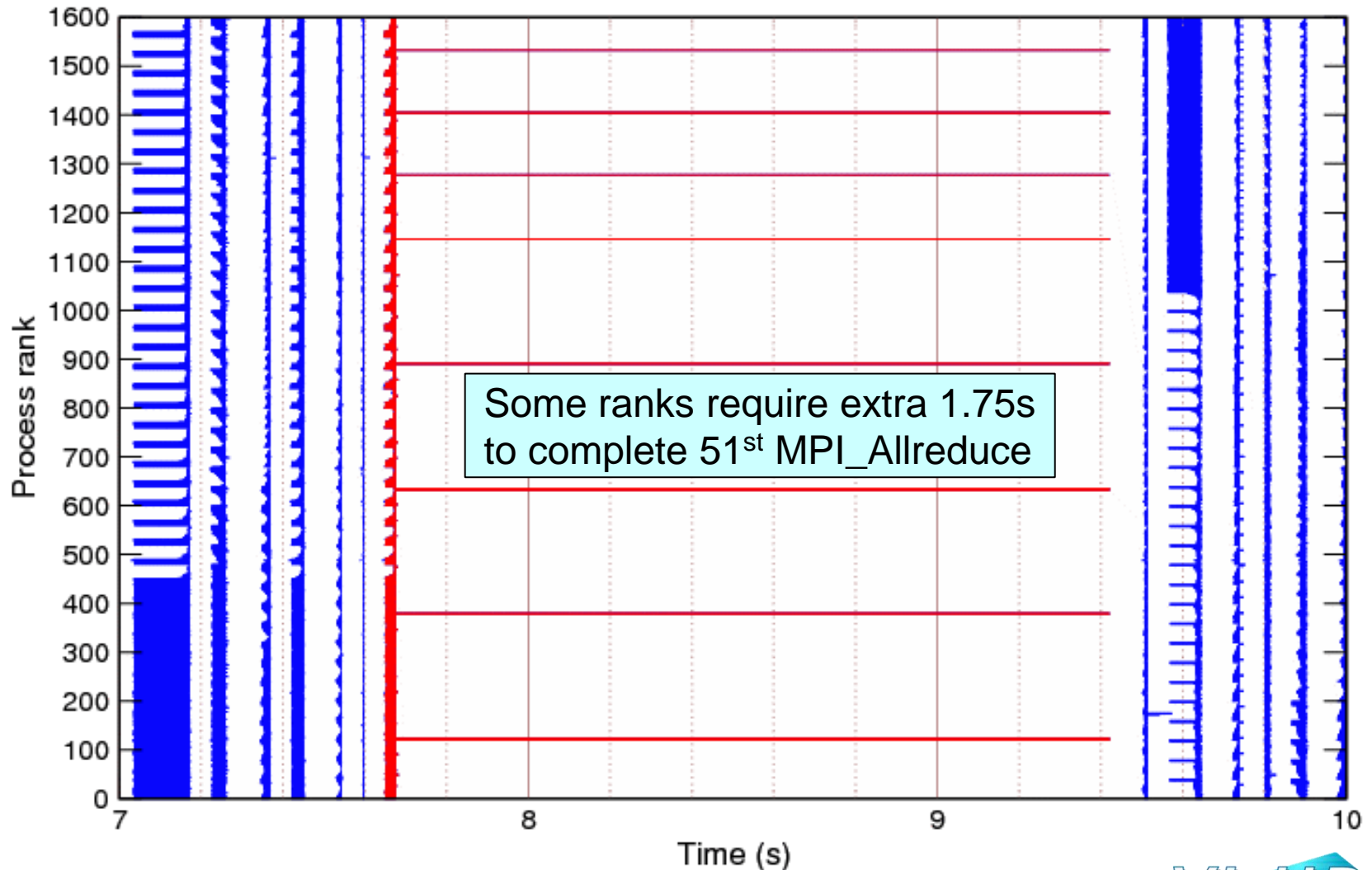


Distribution of data load hits in level 2 cache on application MPI 2D grid topology

WRF on MareNostrum@1600 trace analysis



WRF on MareNostrum@1600 time-line extract

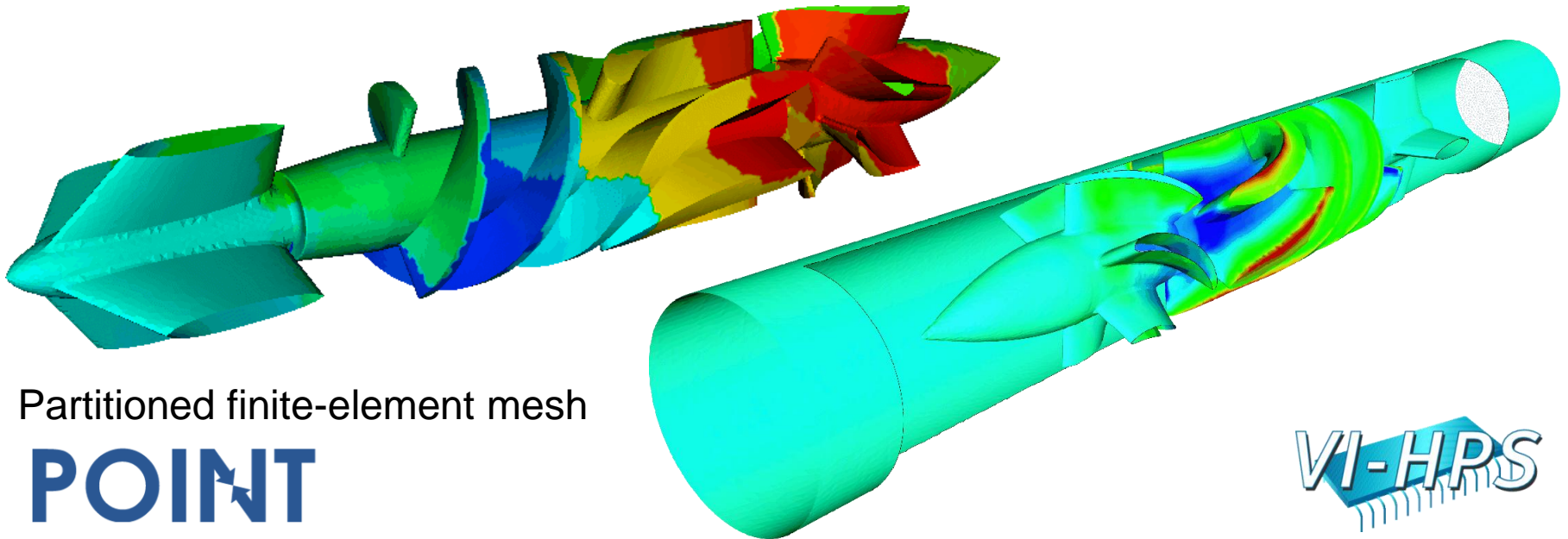


WRF/MareNostrum experience

- Limited system I/O requires careful management
 - Selective instrumentation and measurement filtering
- PowerPC hardware counter metrics included in summary
- Automated trace analysis quantified impact of imbalanced exit from MPI_Allreduce in “NxN completion time” metric
 - Intermittent but serious MPI library/system problem, that restricts application scalability
 - Only a few processes directly impacted, however, communication partners also quickly blocked
- Presentation using logical and physical topologies
 - MPI Cartesian topology provides application insight
 - Hardware topology helps localize system problems

XNS on BlueGene/L case study

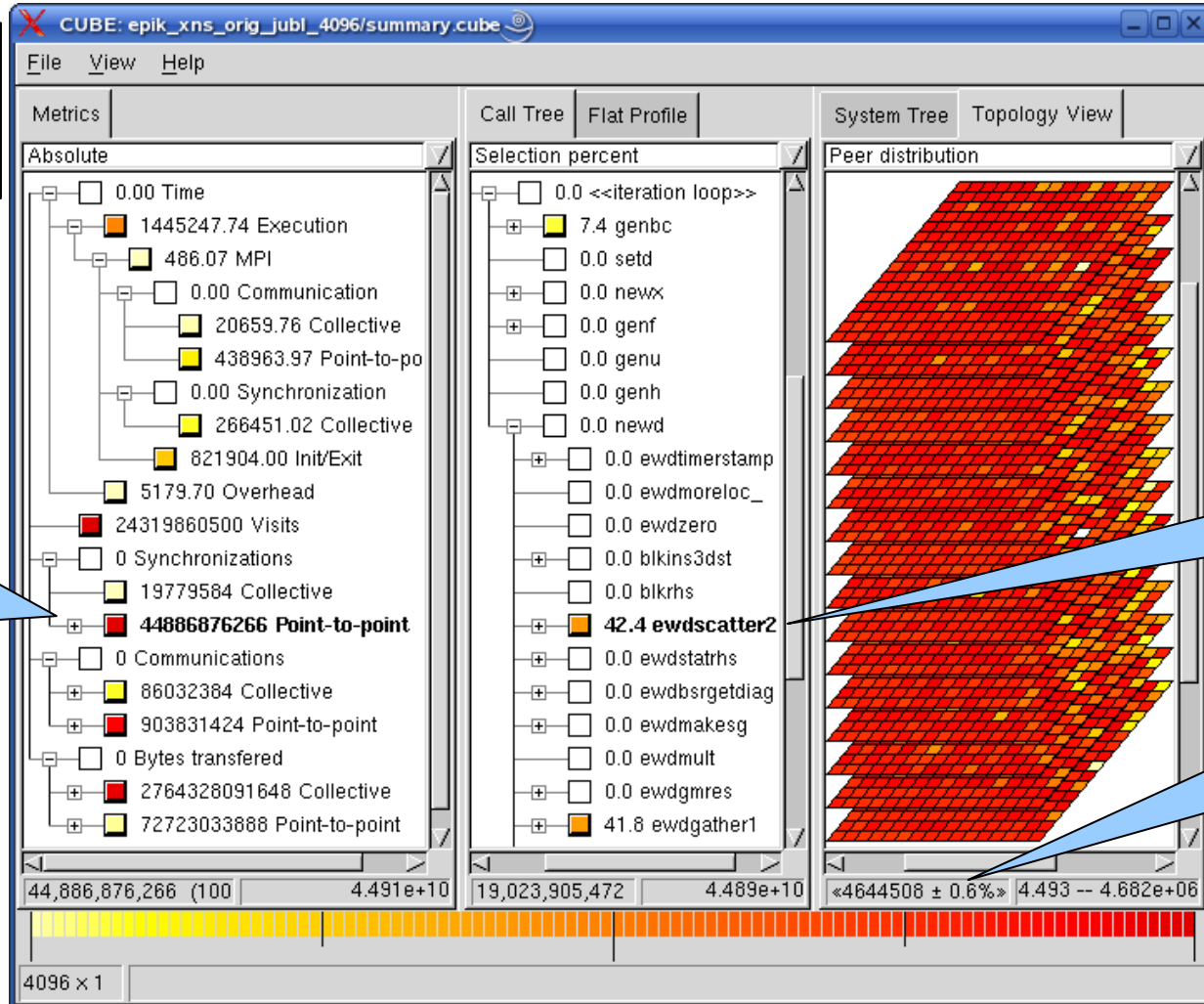
- CFD simulation of unsteady flows
 - developed by RWTH CATS group of Marek Behr
 - exploits finite-element techniques, unstructured 3D meshes, iterative solution strategies
- MPI parallel version (Dec-2006)
 - >40,000 lines of Fortran & C
 - DeBaKey blood-pump dataset (3,714,611 elements)



XNS-DeBaKey on jubl@4096 summary analysis

Point-to-point msgs
w/o data

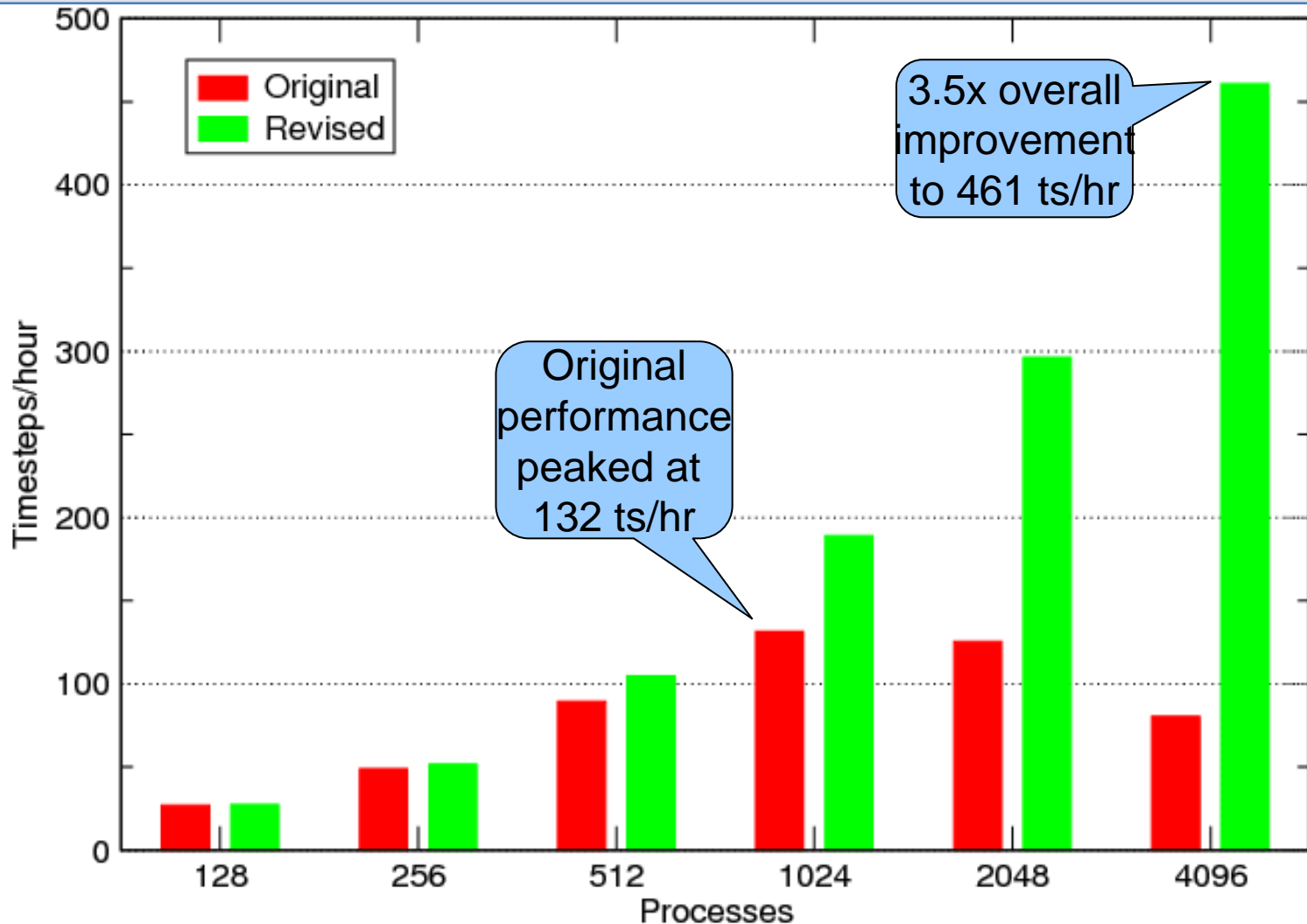
Masses of P2P synch operations



Primarily in scatter & gather

Processes all equally responsible

XNS-DeBakey scalability on BG/L



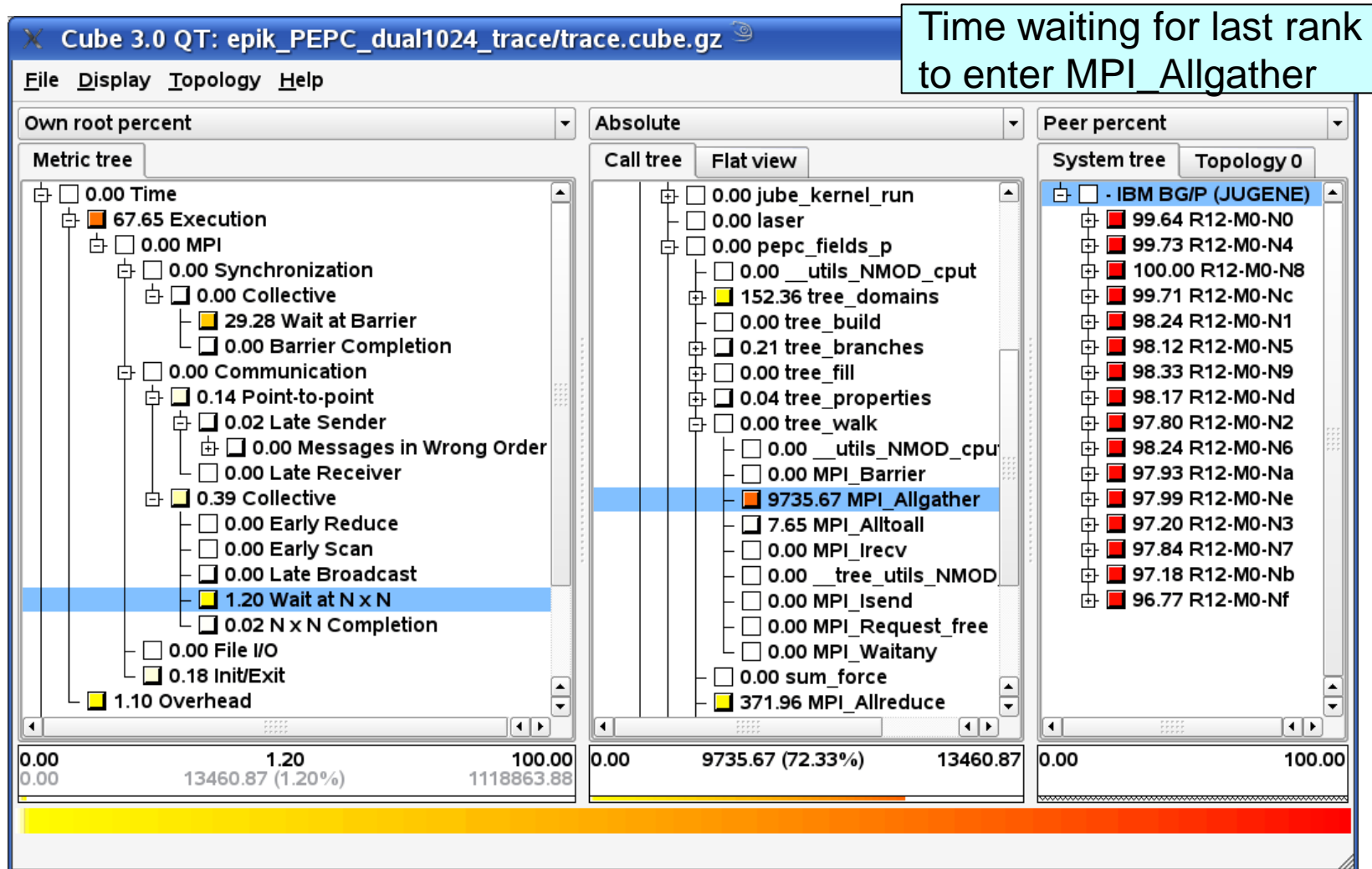
XNS on BlueGene/L experience

- Globally synchronized high-resolution clock facilitates efficient measurement & analysis
- Restricted compute node memory limits trace buffer size and analyzable trace size
- Summarization identified bottleneck due to unintended P2P synchronizations (messages with zero-sized payload)
- 4x solver speedup after replacing MPI_Sendrecv operations with size-dependant separate MPI_Send and MPI_Recv
- Significant communication imbalance remains due to mesh partitioning and mapping onto processors
- MPI_Scan implementation found to contain implicit barrier
 - responsible for 6% of total time with 4096 processes
 - decimated when substituted with simultaneous binomial tree

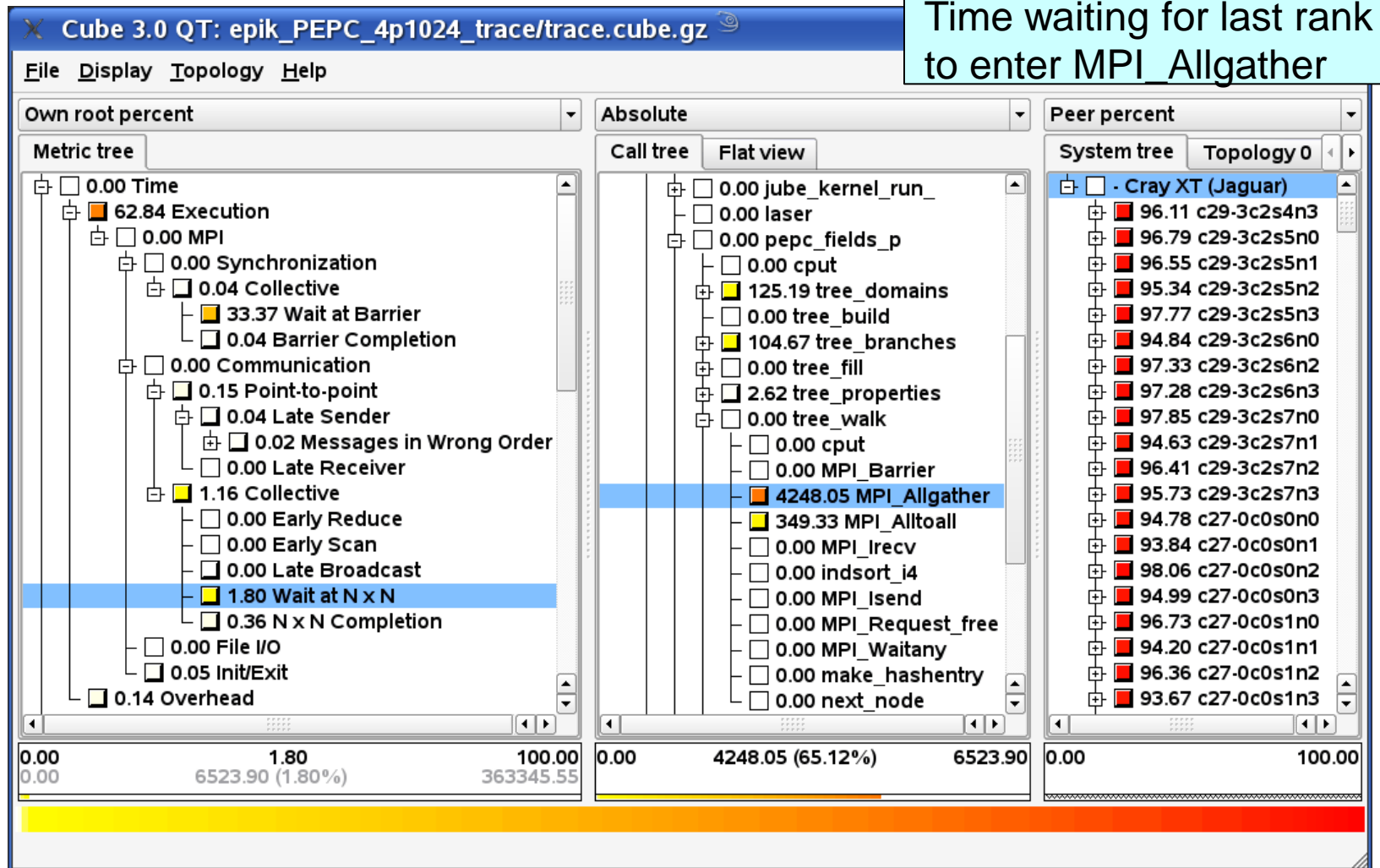
PEPC-B on BG/P & Cray XT case study

- Coulomb solver used for laser-plasma simulations
 - Developed by Paul Gibbon (JSC)
 - Tree-based particle storage with dynamic load-balancing
- MPI version
 - PRACE benchmark configuration, including file I/O
- Run on BlueGene/P in dual mode with 1024 processes
 - 2 processes per quad-core PowerPC node, 1100 seconds
- Run on Cray XT in VN (4p) mode with 1024 processes
 - 4 processes per quad-core Opteron node, 360 seconds

PEPC@1024 on BlueGene/P: Wait at NxN time



PEPC@1024 on Cray XT4: Wait at NxN time



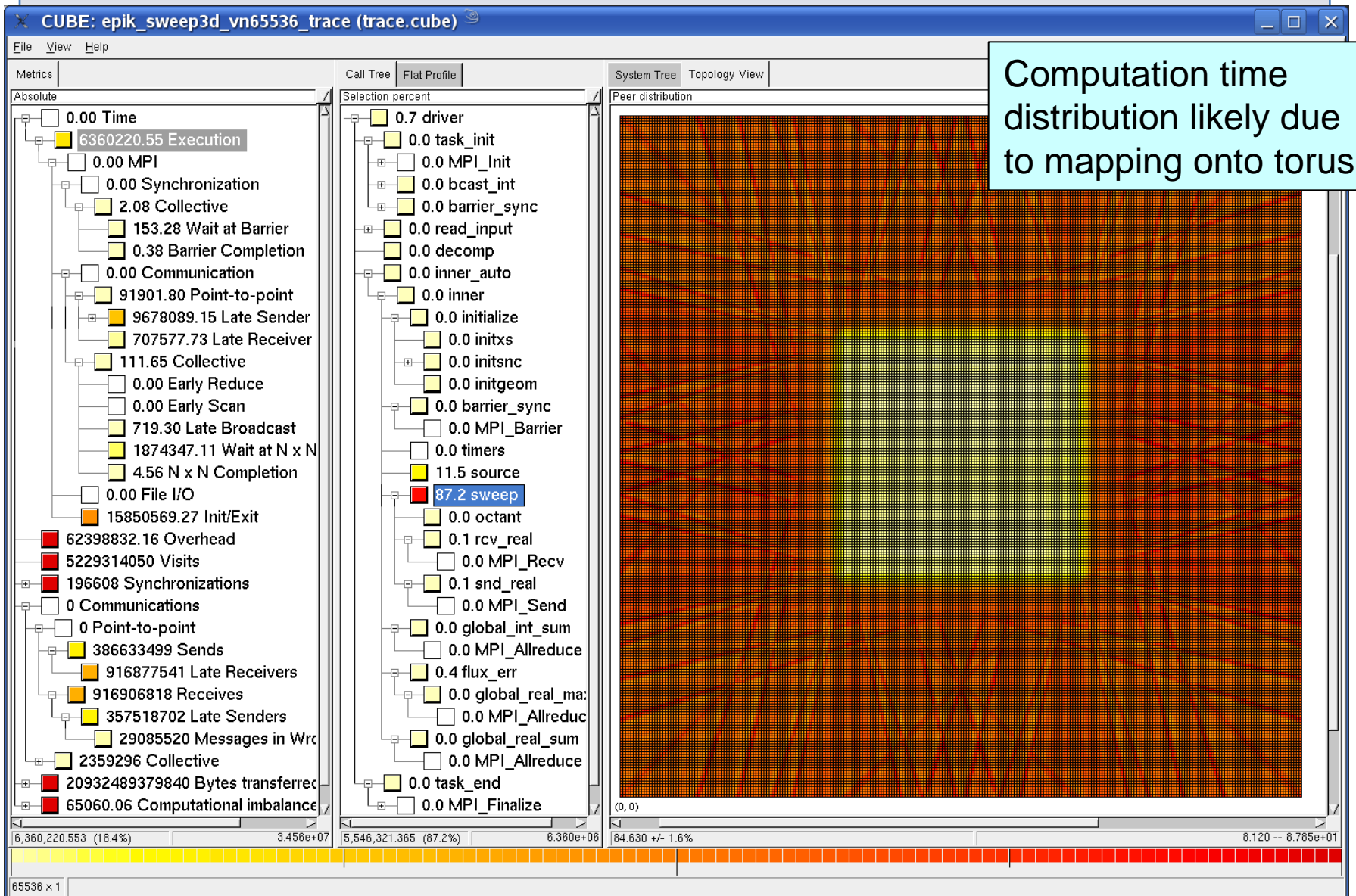
PEPC-B on BG/P & Cray XT experience

- Despite very different processor and network performance, measurements and analyses can be easily compared
 - different compilers affect function naming & in-lining
- Both spend roughly two-thirds of time in computation
 - tree_walk has expensive computation & communication
- Both waste 30% of time waiting to enter MPI_Barrier
 - not localized to particular processes, since particles are regularly redistributed
- Most of collective communication time is also time waiting for last ranks to enter MPI_Allgather & MPI_Alltoall
 - imbalance for MPI_Allgather twice as severe on BlueGene/P, however, almost 50x less for MPI_Alltoall
 - collective completion times also notably longer on Cray XT

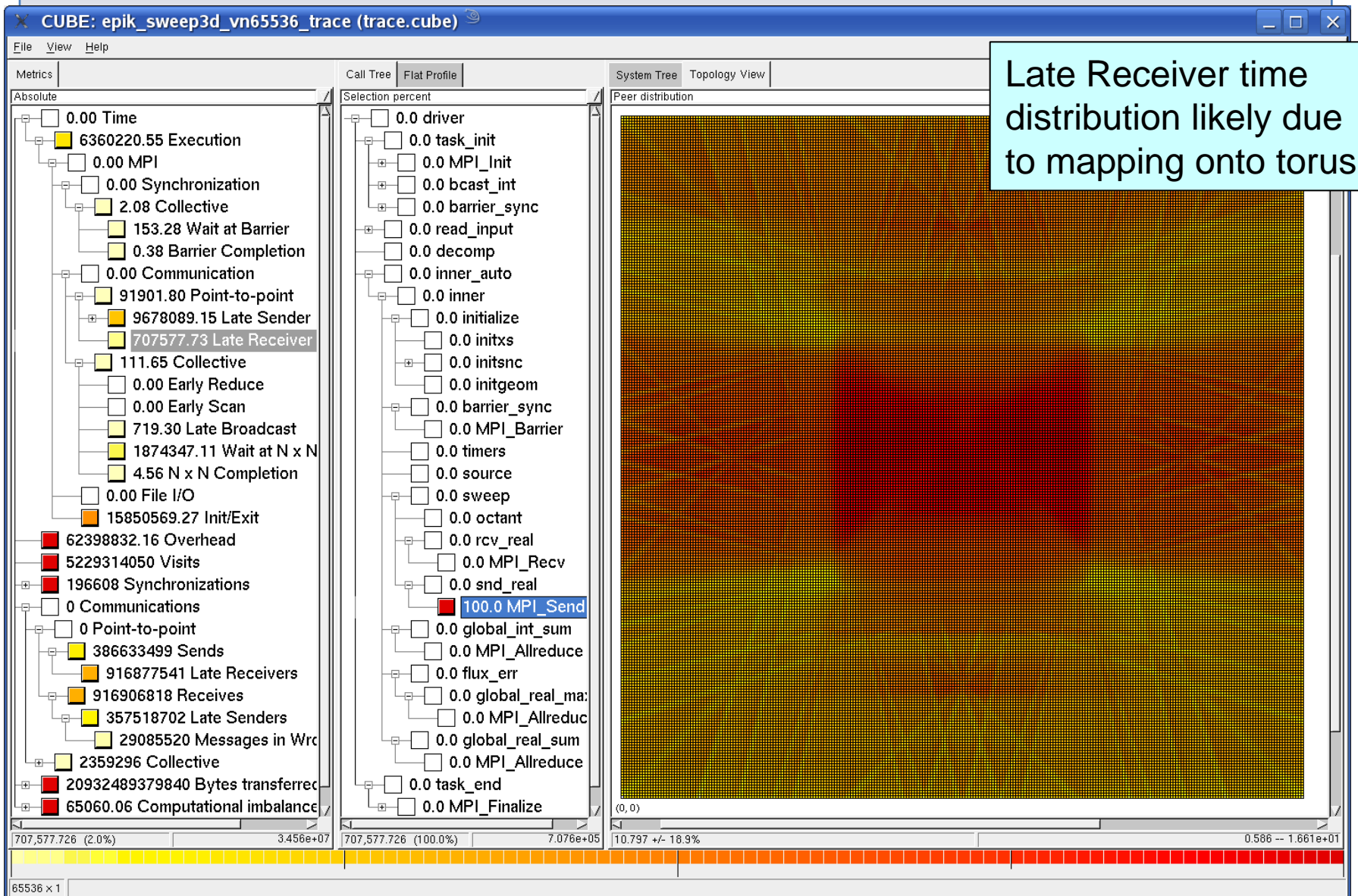
Sweep3d on BlueGene/P case study

- 3D neutron transport simulation
 - ASC benchmark
 - direct order solve uses diagonal sweeps through grid cells
- MPI parallel version 2.2b using 2D domain decomposition
 - ~2,000 lines (12 source modules), mostly Fortran77
- Run on IBM BlueGene/P in VN mode with 64k processes
 - 175GB trace written in 17 minutes, analyzed in 160 seconds
 - 16 minutes just to create 64k files (one per MPI rank)
 - SIONlib reduces this to a couple of seconds
 - Mapping of 256x256 grid of processes onto 3D physical torus results in regular performance artifacts

sweep3d on jugene@64k trace analysis



sweep3d on jugene@64k trace (wait) analysis



Acknowledgements

- The application and benchmark developers who provided their codes and/or measurement archives
- The facilities who made their HPC resources available
 - BSC, CSC, CSCS, EPCC, JSC, HLRS, LRZ, NCCS/ORNL, RWTH, RZG, TeraGrid/TACC, TUD/ZIH, ALCF, UTK/ICL

Further information

Scalable performance analysis of large-scale parallel applications

- toolset for scalable performance measurement & analysis of MPI, OpenMP & hybrid parallel applications
- supporting most popular HPC computer systems
- available under New BSD open-source license
- sources, documentation & publications:
 - <http://www.scalasca.org>
 - mailto: scalasca@fz-juelich.de

scalasca 

VAMPIRTRACE & VAMPIR INTRODUCTION AND OVERVIEW

Andreas Knüpfer

Technical University Dresden

Overview

- Introduction
- Event Trace Visualization
- Vampir & VampirServer
- The Vampir Displays
 - Timeline
 - Process Timeline with Performance Counters
 - Summary Display
 - Message Statistics
- VampirTrace
 - Instrumentation & Run-Time Measurement
- Conclusions

Introduction

Why bother with performance analysis?

- well, why are you here after all?
- efficient usage of expensive and limited resources
- scalability to achieve next bigger simulation

Profiling and Tracing

- have an optimization phase
 - just like testing and debugging phase
- use tools!
- avoid *do-it-yourself-with-printf* solutions, really!

Introduction: Profiling & Tracing

Program Instrumentation

- detect run-time events (points of interest)
- pass information to run-time measurement library

Profile Recording

- collect aggregated information (Time, Counts, ...)
- about program and system entities
 - functions, loops, basic blocks
 - application, processes, threads, ...

Trace Recording

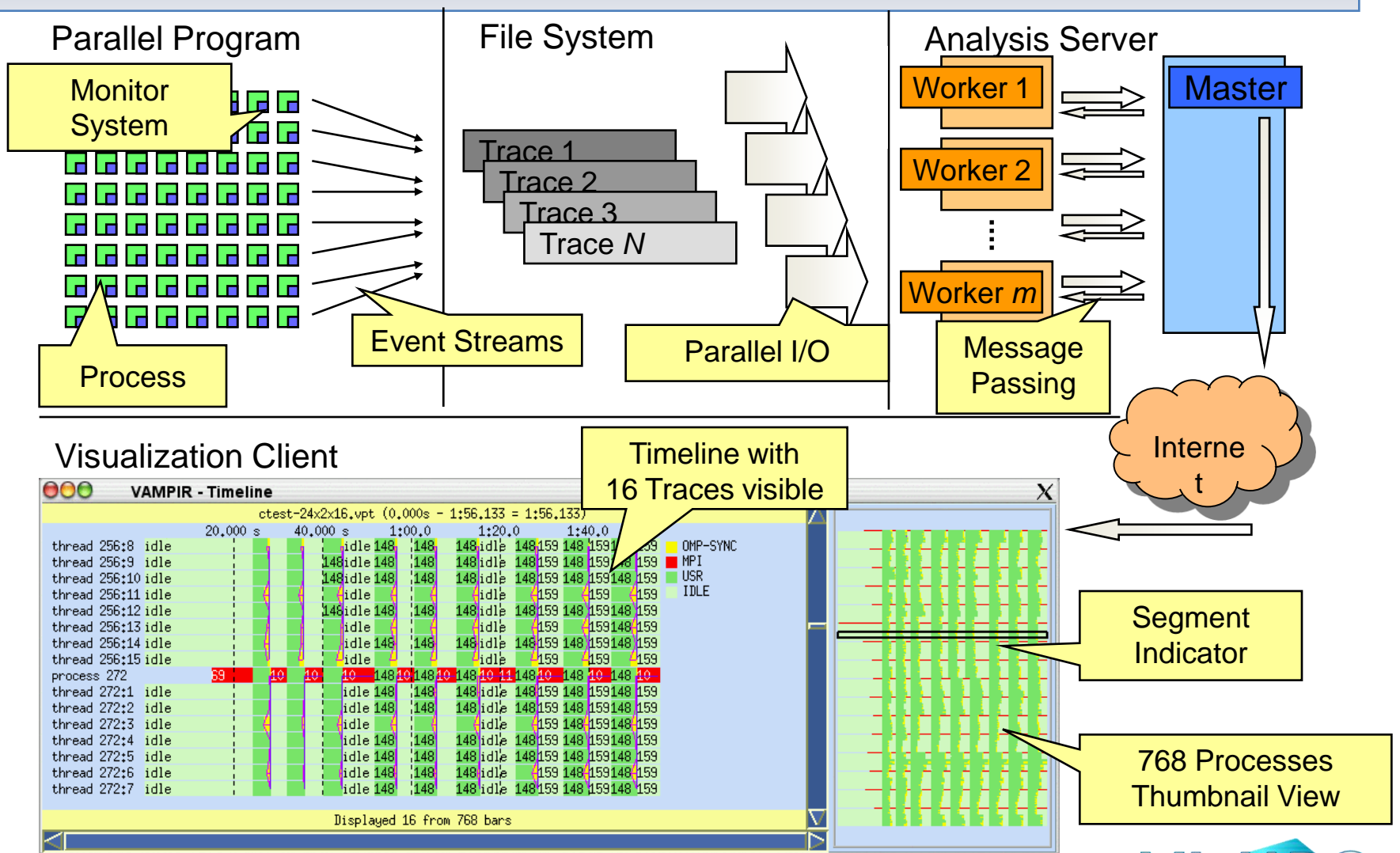
- save individual event records together with precise timestamp and process or thread ID
- plus event specific information

Event Trace Visualization

Trace Visualization

- alternative and supplement to automatic analysis
- show dynamic run-time behavior graphically
- provide statistics and performance metrics
 - Global timeline for parallel processes/threads
 - Process timeline plus performance counters
 - Statistic summary display
 - Message statistics
 - more
- interactive browsing, zooming, selecting
 - adapt statistics to zoom level (time interval)
 - also for very large and highly parallel traces

VampirServer Architecture

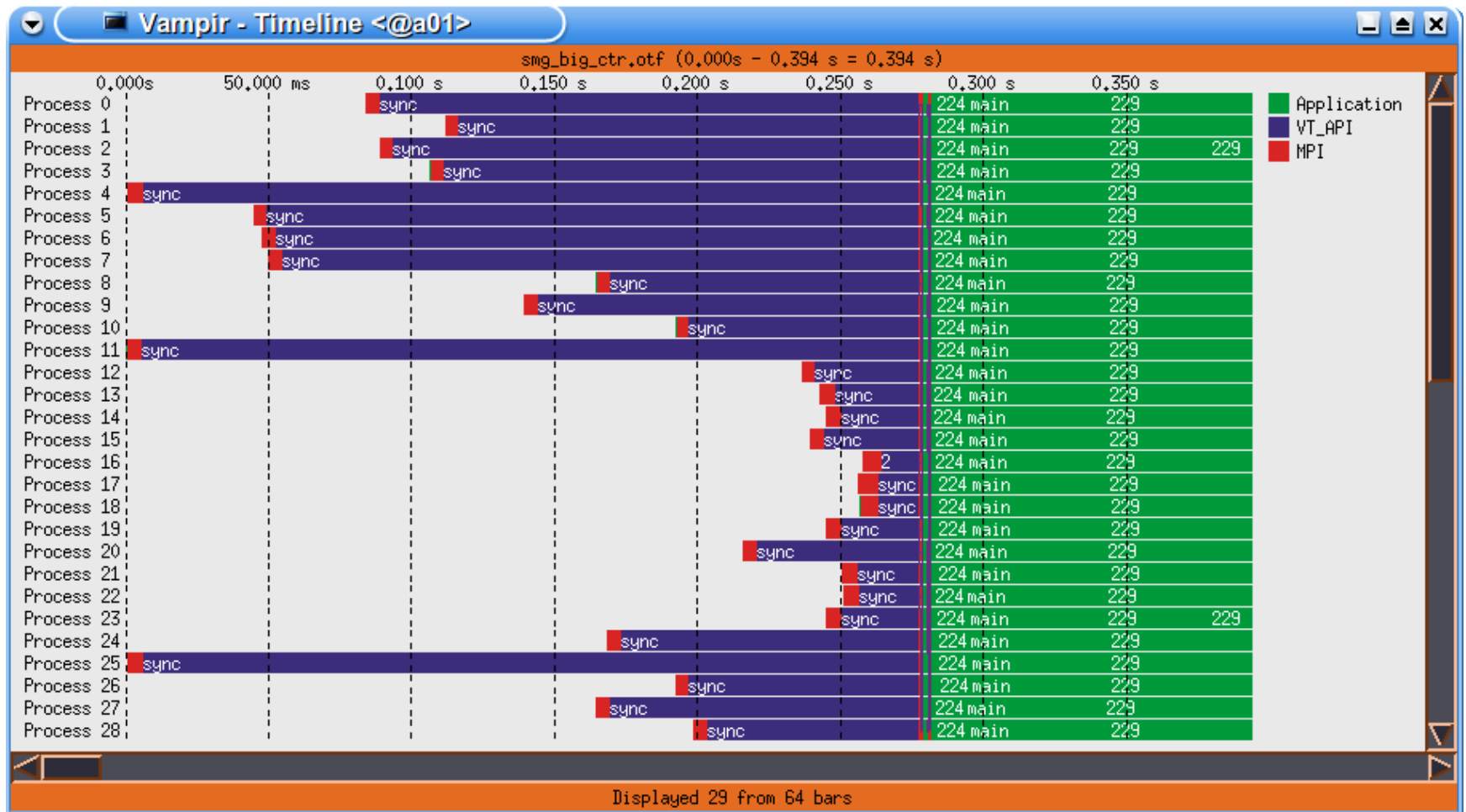


Vampir Displays

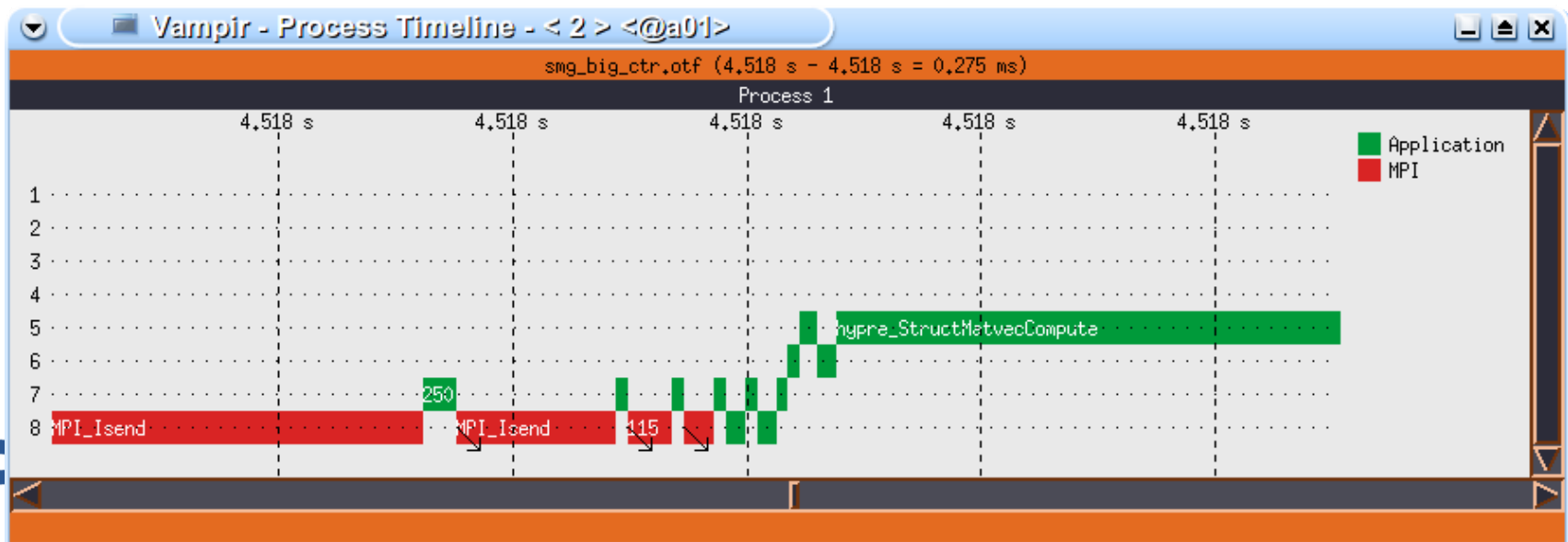
The main displays of Vampir:

- Global Timeline
- Process Timeline w/o Counters
- Statistic Summary
- Summary Timeline
- Message Statistics
- Collective Operation Statistics
- Counter Timeline
- Call Tree

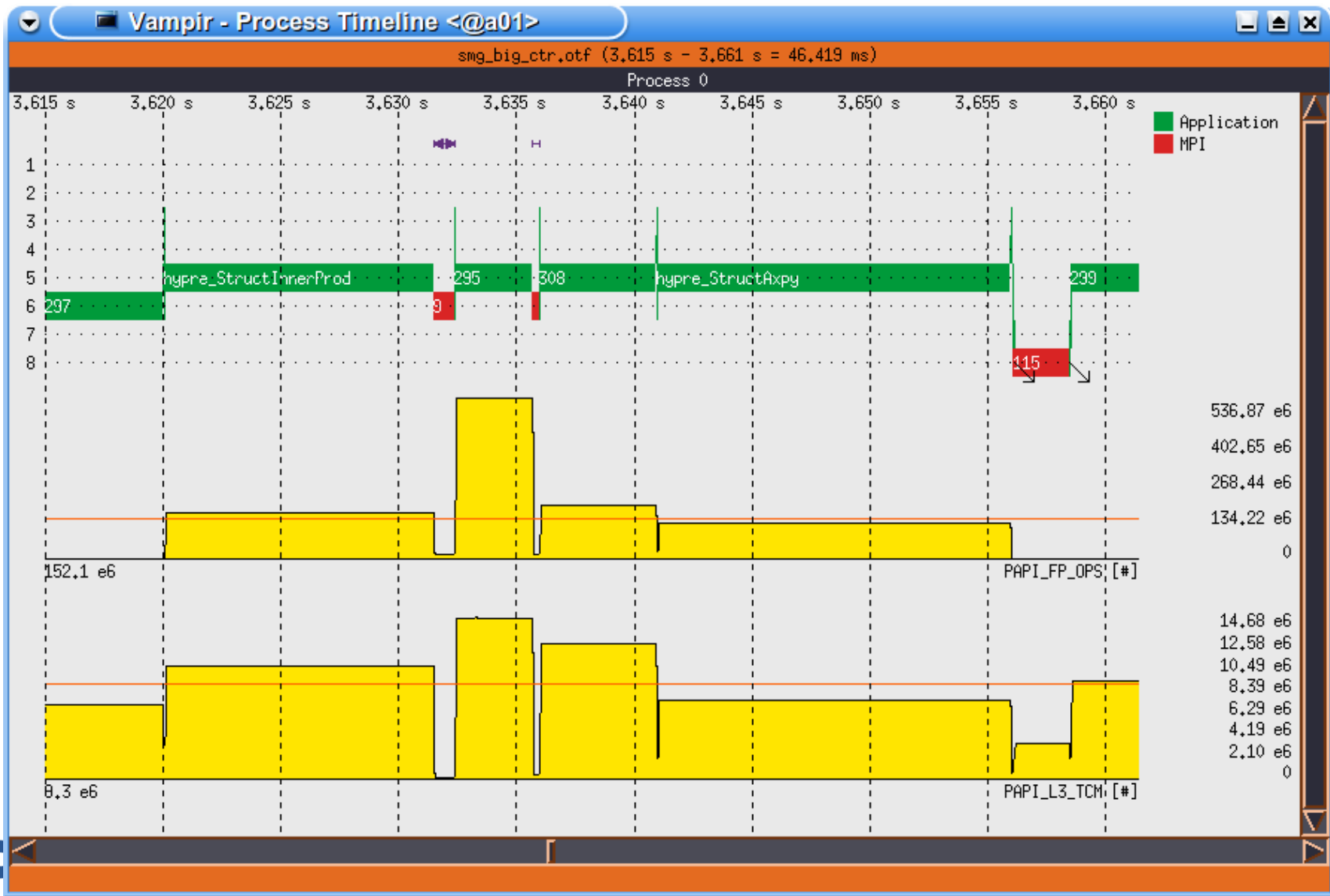
Vampir Global Timeline Display



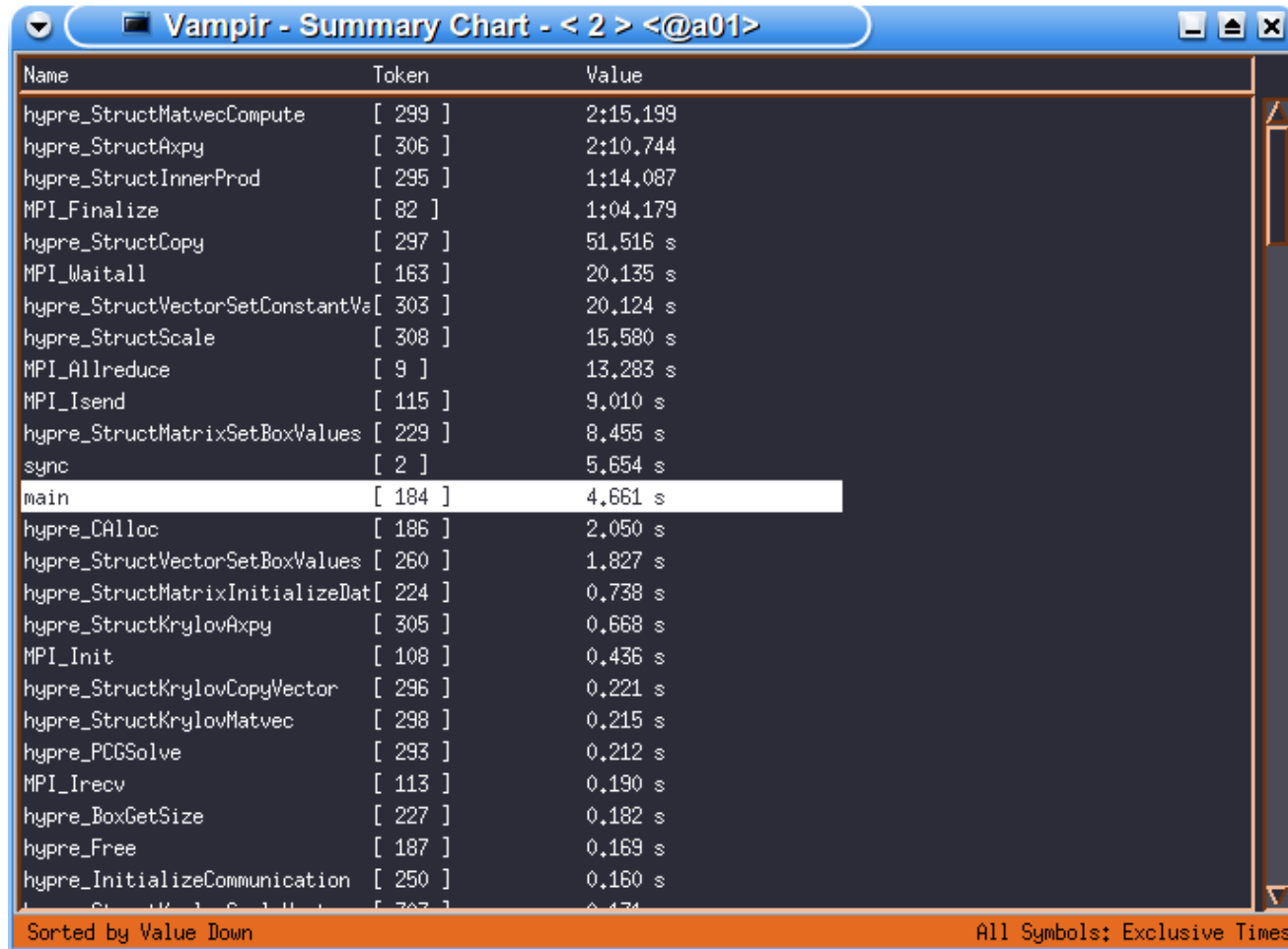
Process Timeline Display



Process Timeline with Counters



Statistic Summary Display

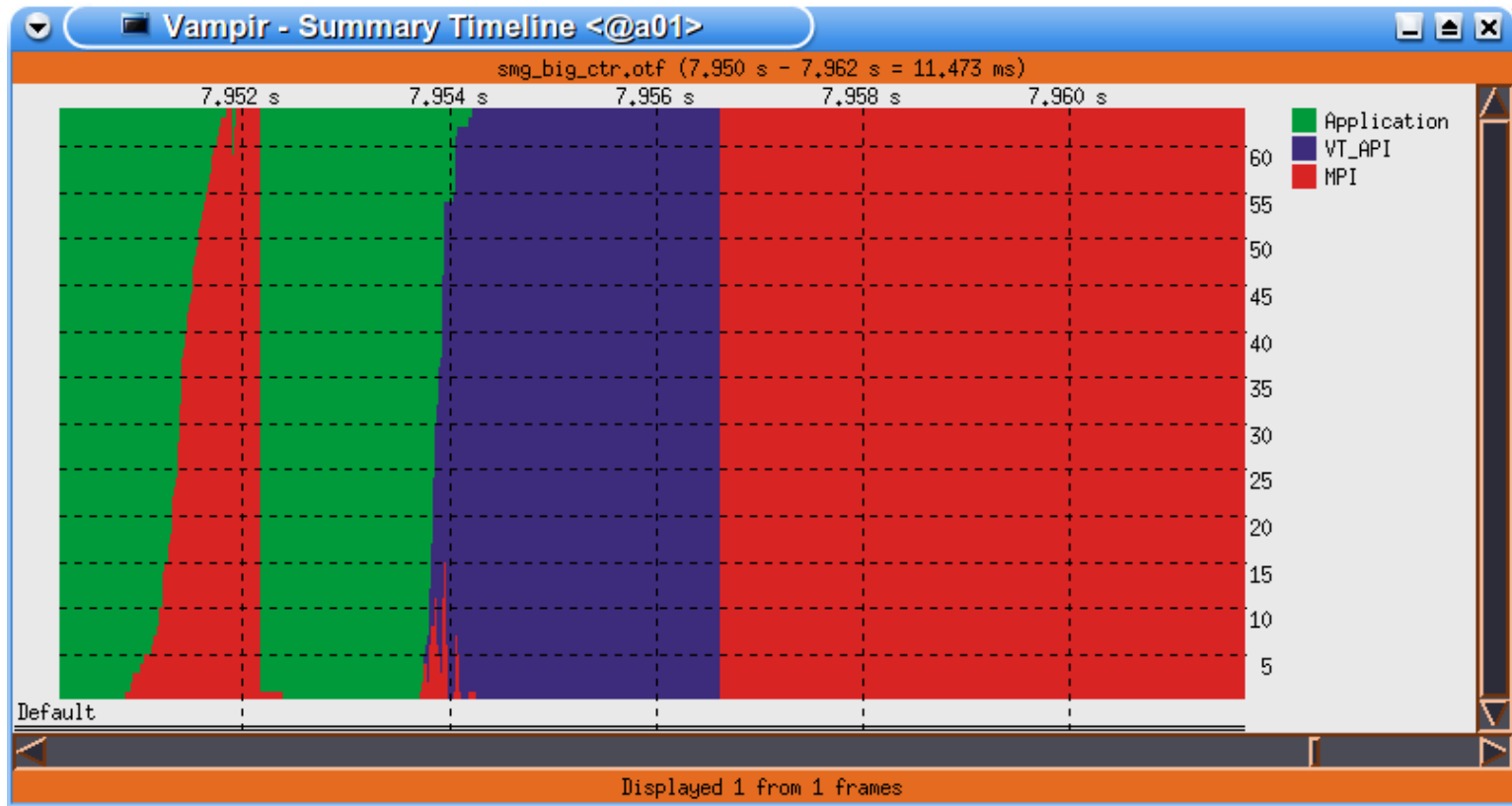


Vampir - Summary Chart - < 2 > <@a01>

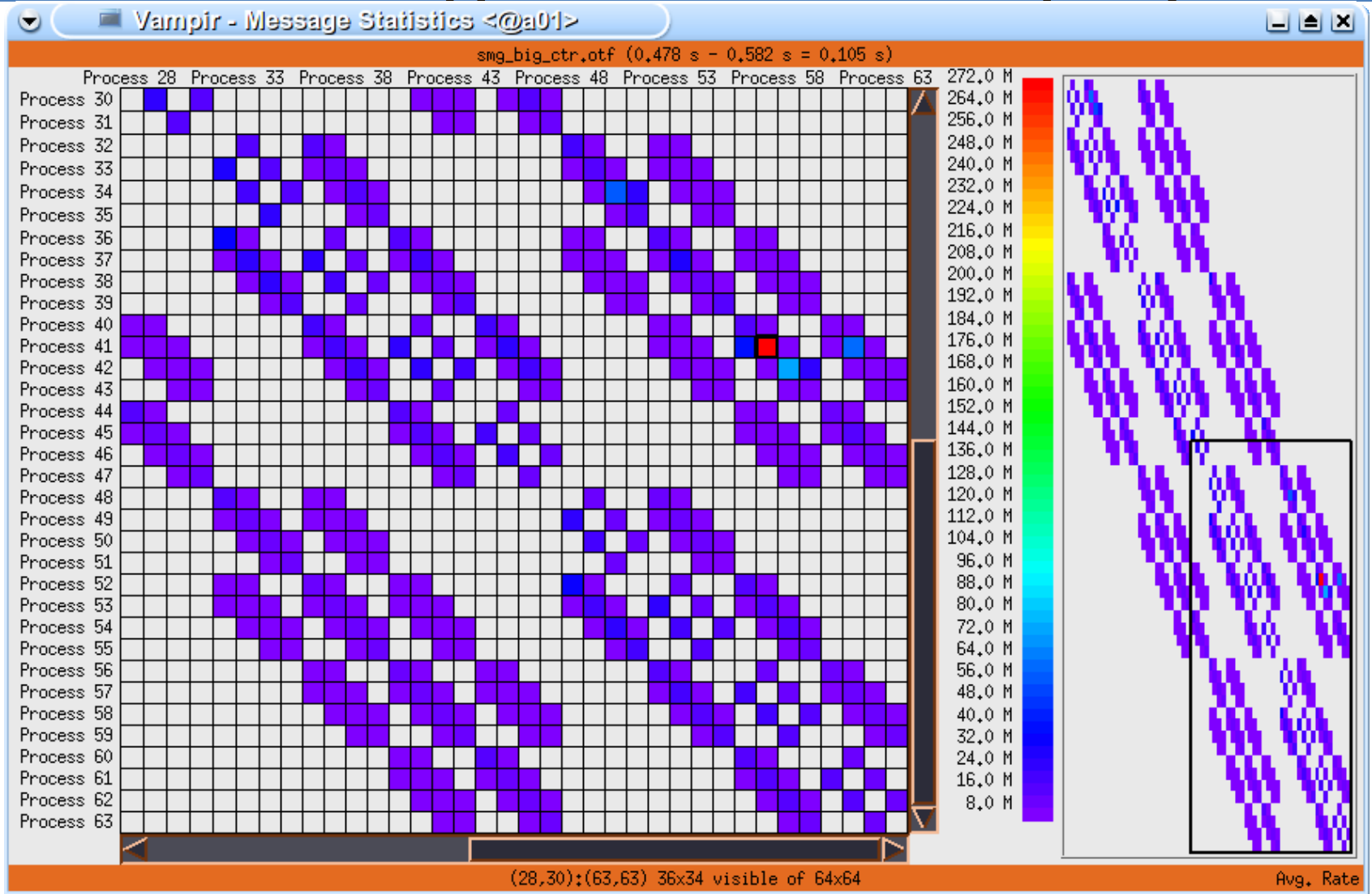
Name	Token	Value
hypre_StructMatvecCompute	[299]	2:15,199
hypre_StructAxy	[306]	2:10,744
hypre_StructInnerProd	[295]	1:14,087
MPI_Finalize	[82]	1:04,179
hypre_StructCopy	[297]	51,516 s
MPI_Waitall	[163]	20,135 s
hypre_StructVectorSetConstantVa	[303]	20,124 s
hypre_StructScale	[308]	15,580 s
MPI_Allreduce	[9]	13,283 s
MPI_Isend	[115]	9,010 s
hypre_StructMatrixSetBoxValues	[229]	8,455 s
sync	[2]	5,654 s
main	[184]	4,661 s
hypre_CAlloc	[186]	2,050 s
hypre_StructVectorSetBoxValues	[260]	1,827 s
hypre_StructMatrixInitializeDat	[224]	0,738 s
hypre_StructKrylovAxy	[305]	0,668 s
MPI_Init	[108]	0,436 s
hypre_StructKrylovCopyVector	[296]	0,221 s
hypre_StructKrylovMatvec	[298]	0,215 s
hypre_PCGSolve	[293]	0,212 s
MPI_Irecv	[113]	0,190 s
hypre_BoxGetSize	[227]	0,182 s
hypre_Free	[187]	0,169 s
hypre_InitializeCommunication	[250]	0,160 s
hypre_StructMatrixSetBoxValues	[287]	0,134 s

Sorted by Value Down All Symbols: Exclusive Times

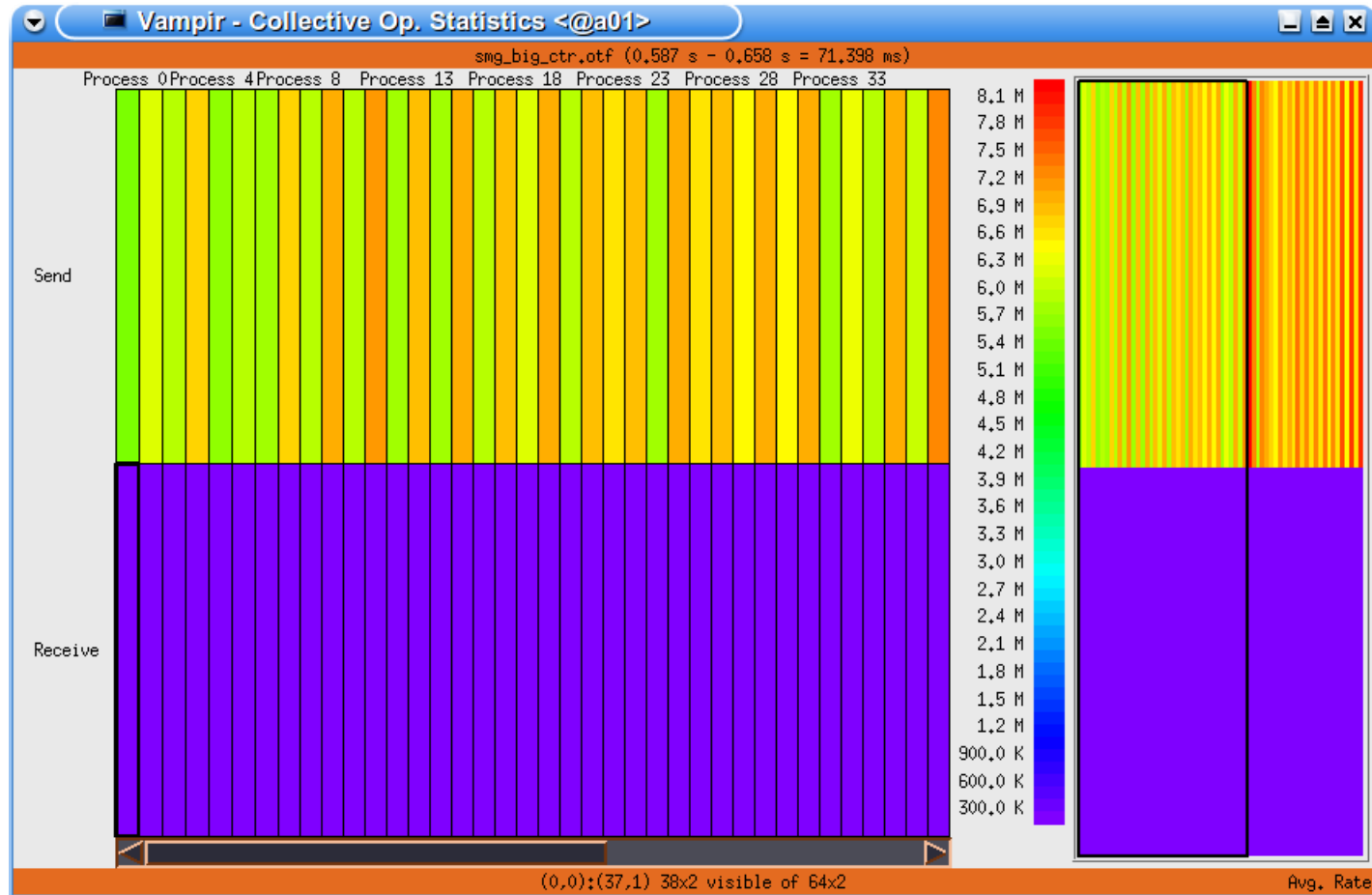
Summary Timeline Display



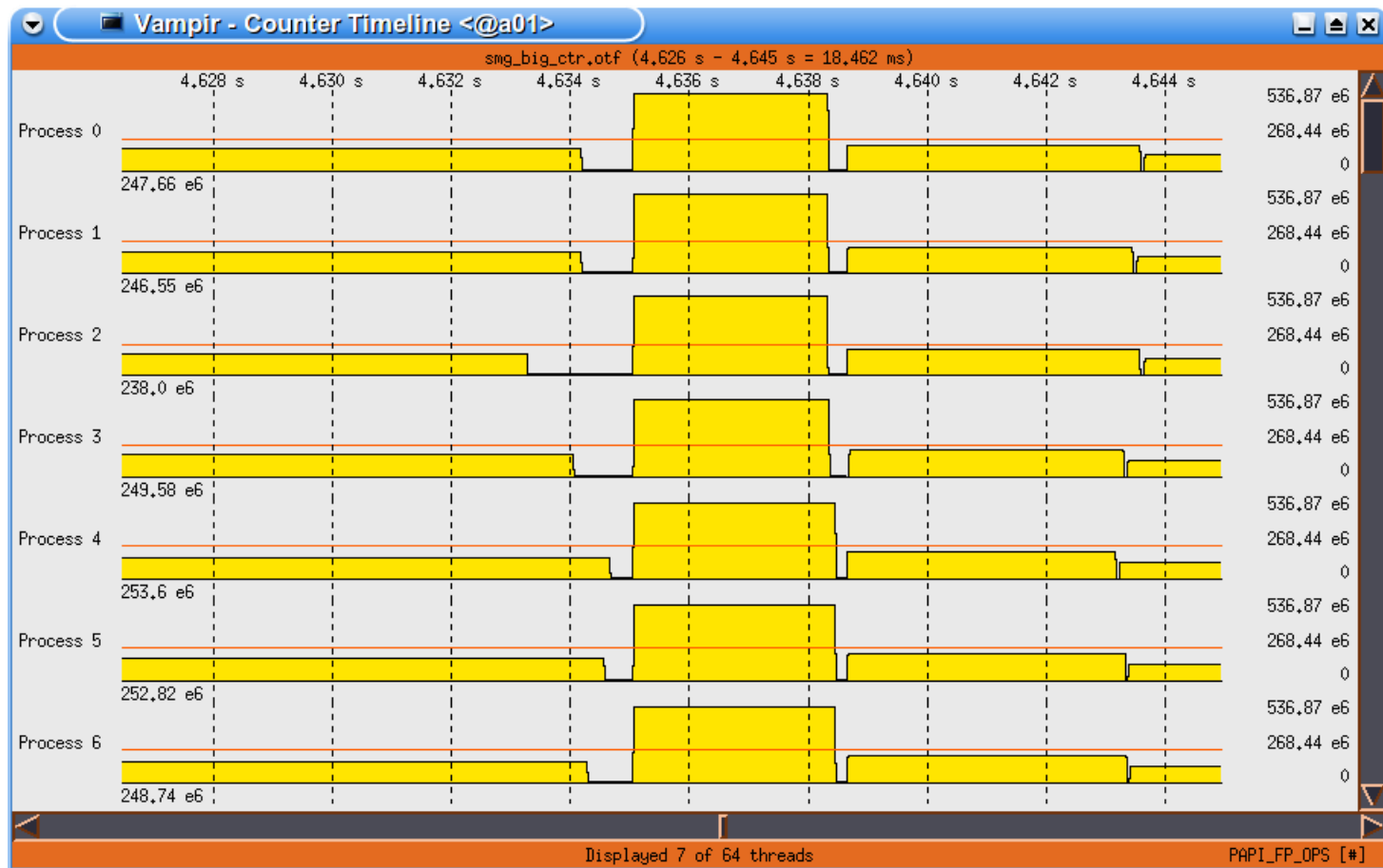
Message Statistics Display



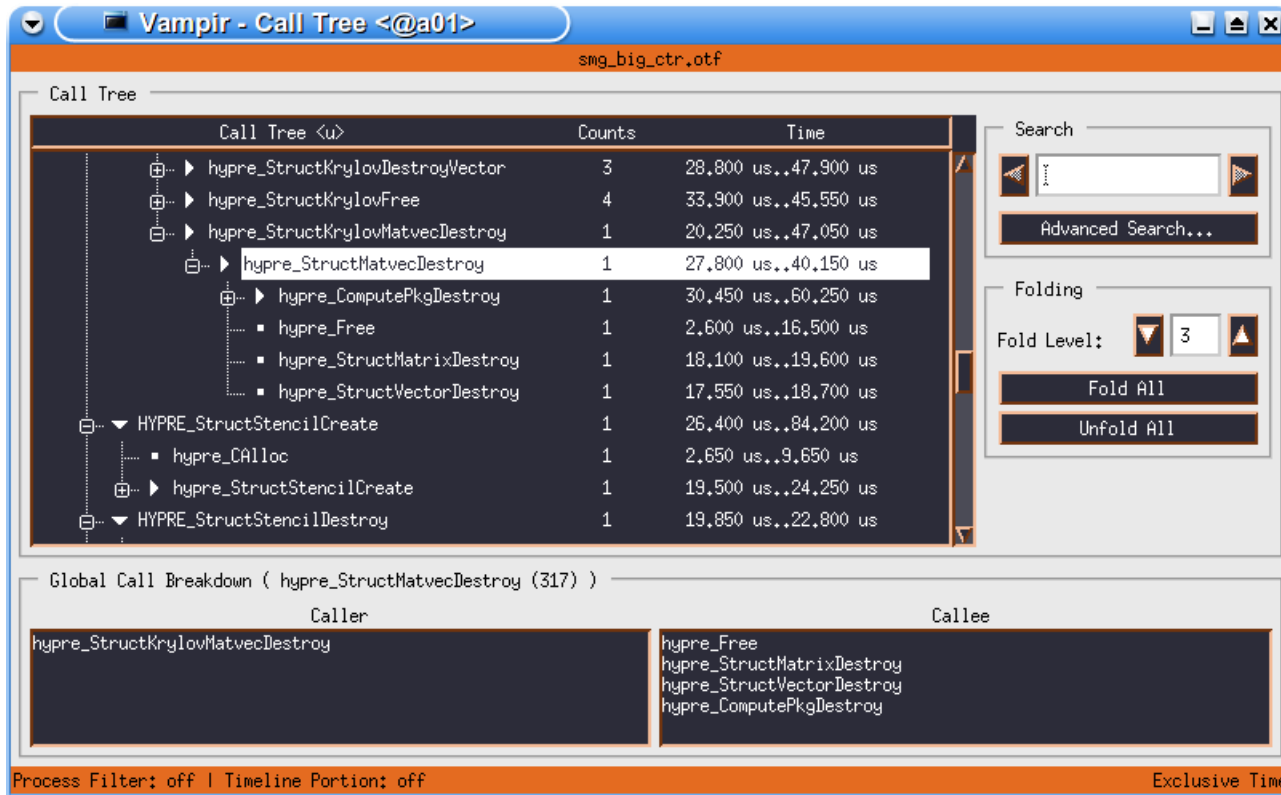
Collective Operation Statistics



Counter Timeline Display



Call Tree Display



Instrumentation & Measurement

- What do you need to do for it?
- Instrumentation (automatic with compiler wrappers)

```
CC=icc
```

```
CXX=icpc
```

```
F90=ifc
```

```
MPICC=mpicc
```

```
CC=vtcc
```

```
CXX=vtcxx
```

```
F90=vtf90
```

```
MPICC=vtcc
```

- Re-compile & re-link
- Trace Run (run with appropriate test data set)
- more details later

Instrumentation & Measurement

What does VampirTrace do in the background?

- during Instrumentation:
 - via compiler wrappers
 - by underlying compiler with specific options
 - MPI instrumentation with replacement lib
 - OpenMP instrumentation with Opari
 - also binary instrumentation with Dyninst
 - partial manual instrumentation

Instrumentation & Measurement

What does VampirTrace do in the background?

- during Trace Run:
 - event data collection
 - precise time measurement
 - parallel timer synchronization
 - collecting parallel process/thread traces
 - collecting performance counters (from PAPI, memory usage, POSIX I/O calls and fork/system/exec calls, and more ...)
 - filtering and grouping of function calls

Summary

- VampirTrace
 - convenient instrumentation and measurement
 - hides away complicated details
 - provides many options and switches for experts
- VampirTrace is part of Open MPI 1.3
- Vampir & VampirServer
 - interactive trace visualization and analysis
 - intuitive browsing and zooming
 - scalable to large trace data sizes (100GB)
 - scalable to high parallelism (2000 processes)
- Vampir for Windows in progress, beta available

VAMPIRTRACE & VAMPIR: DETAILS AND HANDS-ON

Overview

- Event tracing in general
- Instrumentation
- Run-time measurement
- Visualization and analysis

Profiling and Tracing

- Tracing Advantages
 - preserve temporal and spatial relationships
 - allow reconstruction of dynamic behavior on any required abstraction level
 - profiles can be calculated from trace
- Tracing Disadvantages
 - traces can become very large
 - may cause perturbation
 - instrumentation and tracing is complicated
 - event buffering, clock synchronization, ...

Common Event Types

- enter/leave of function/routine/region
 - time stamp, process/thread, function ID
- send/receive of P2P message (MPI)
 - time stamp, sender, receiver, length, tag, comm.
- collective communication (MPI)
 - time stamp, process, root, communicator, # bytes
- hardware performance counter values
 - time stamp, process, counter ID, value
- etc.

Open Trace Format (OTF)

- Open source trace file format
- Available at <http://www.tu-dresden.de/zih/otf/>
- Includes powerful libotf for reading/parsing/writing in custom applications
- multi-level API:
 - High level interface for analysis tools
 - Low level interface for trace libraries
- Actively developed in cooperation with the University of Oregon and the Lawrence Livermore National Laboratory

Instrumentation

- Instrumentation: Process of modifying programs to detect and report events
 - call instrumentation functions
 - provided by trace library
 - call for every run-time event of interest
 - there are various ways of instrumentation

Source Code Instrumentation

```
int foo(void* arg) {  
  
    if (cond) {  
  
        return 1;  
    }  
  
    return 0;  
}
```

```
int foo(void* arg) {  
    enter(7);  
    if (cond) {  
        leave(7);  
        return 1;  
    }  
    leave(7);  
    return 0;  
}
```

manually or automatically

Source Code Instrumentation

- manually
 - large effort, error prone
 - difficult to manage
 - see documentation of VampirTrace API
- automatically
 - via source to source translation
 - Program Database Toolkit (PDT)
 - OpenMP Pragma And Region Instrumentor (Opari)

Wrapper Functions

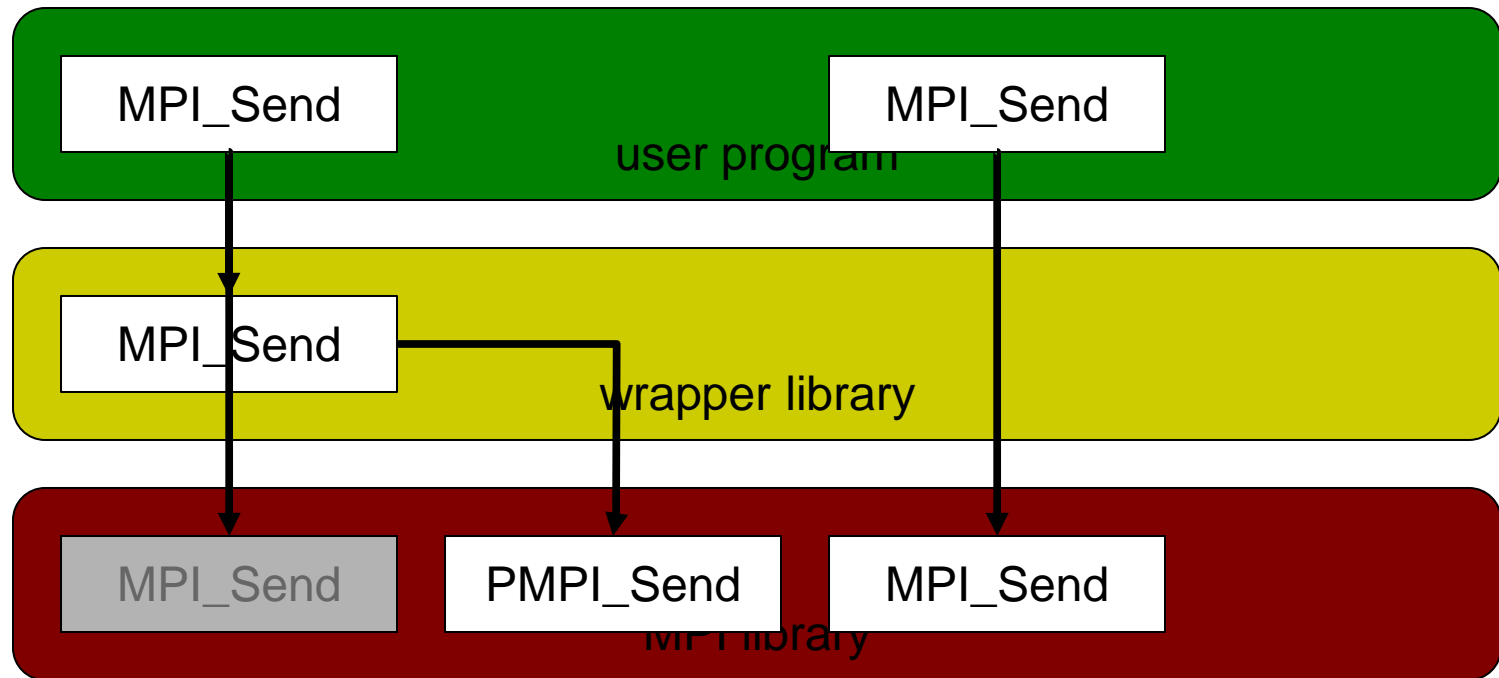
- provide wrapper functions
 - call instrumentation function for notification
 - call original target for functionality
 - via preprocessor directives:

```
#define MPI_Init WRAPPER_MPI_Init  
#define MPI_Send WRAPPER_MPI_Send
```

- via library preload:
 - preload instrumented dynamic library
- suitable for standard libraries (e.g. MPI, glibc)

The MPI Profiling Interface

- Each MPI function has two names:
 - MPI_xxx and PMPI_xxx
- Replacement of MPI routines at link time



Compiler Instrumentation

```
gcc -finstrument-functions -c foo.c
```

```
void __cyg_profile_func_enter( <args> );  
void __cyg_profile_func_exit( <args> );
```

- many compilers support this: GCC, Intel, IBM, PGI, NEC, Hitachi, Sun Fortran, ...
- no source modification necessary

Dynamic Instrumentation

- modify executable in file or binary in memory
- insert instrumentation calls
- very platform/machine dependent, expensive
- DynInst project (<http://www.dyninst.org>)
 - common interface
 - supported platforms: Alpha/Tru64, MIPS/IRIX, PowerPC/AIX, Sparc/Solaris, x86/Linux
x86/Windows, ia64/Linux

Practical Instrumentation

- Instrumentation with VampirTrace
 - hide instrumentation in compiler wrapper
 - use underlying compiler, add appropriate options

```
CC=mpicc
```

```
CC=vtcc
```

- Test Run
 - use representative test input
 - set parameters, environment variables, etc.
 - perform trace run
- Get Trace

Hands-on: Hello World

Hands-on: Hello World

- get example

```
%> tar xzf Desktop/Workshop\ Examples/01_hello_world.tgz
```

- default build and normal run

```
%> cd 01_hello_world  
%> make  
%> mpirun -np 4 ./hello
```


Hands-on: Hello World

- enable automatic instrumentation in Makefile:

```
# CC=mpicc  
CC=vtcc -vt:cc mpicc -vt:verbose
```

- re-build with tracing:

```
%> make clean  
%> make
```

Hands-on: Hello World

– run instrumented version:

```
%> export VT_BUFFER_SIZE="10M"
%> export VT_FILE_PREFIX="trace_hello1"

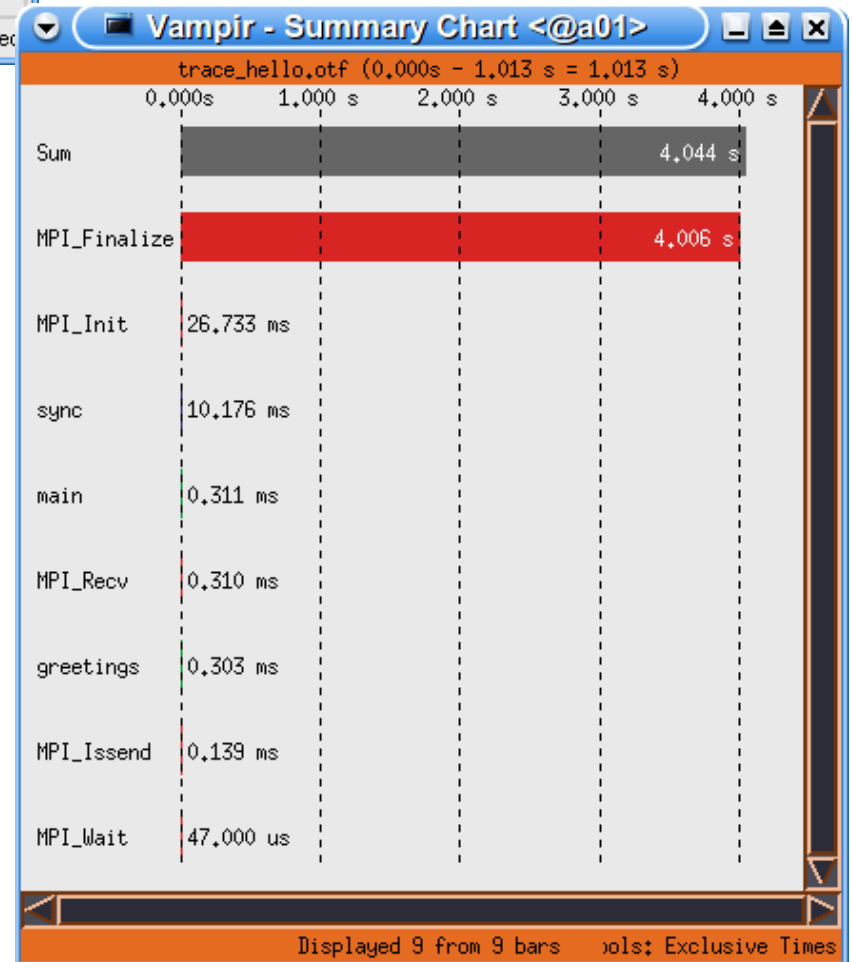
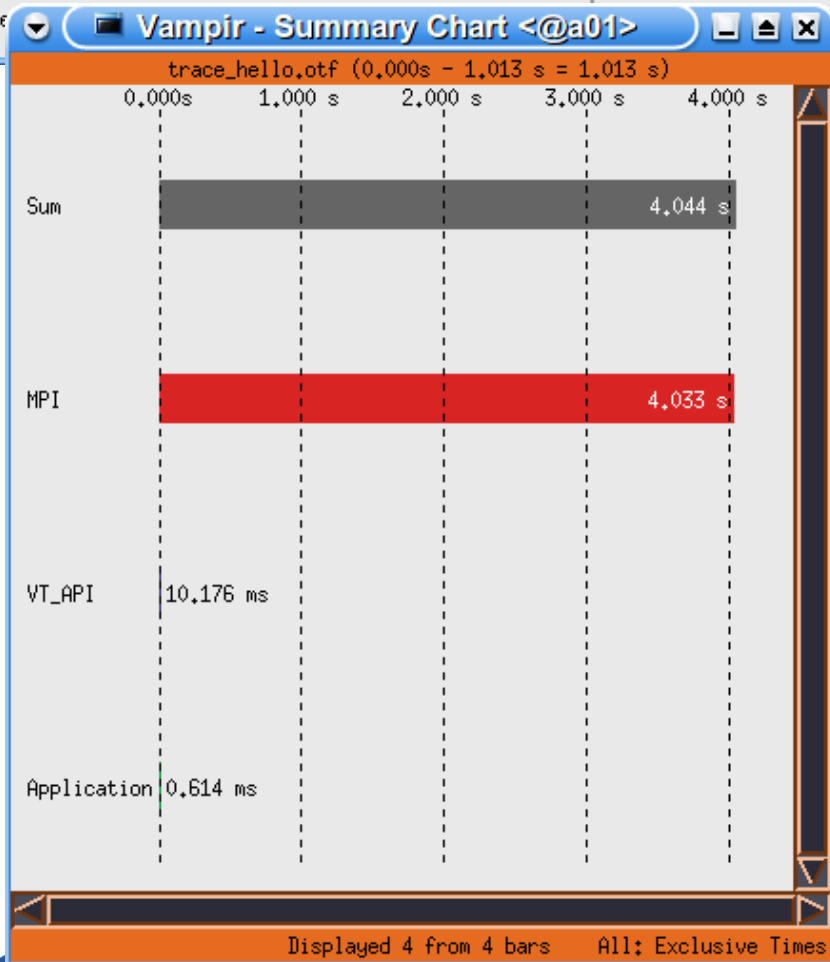
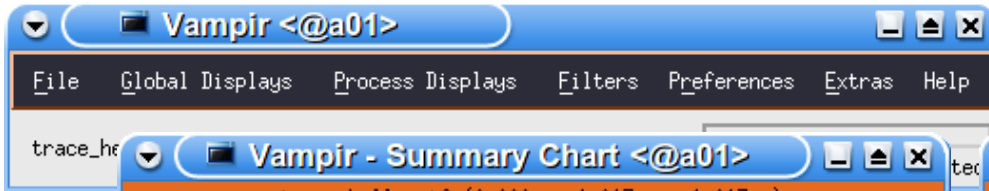
%> mpirun -np 4 ./hello

%> ls -alh
1.7K trace_hello1.0.def.z
214 trace_hello1.1.events.z
213 trace_hello1.2.events.z
212 trace_hello1.3.events.z
208 trace_hello1.4.events.z
16 trace_hello1.otf
```

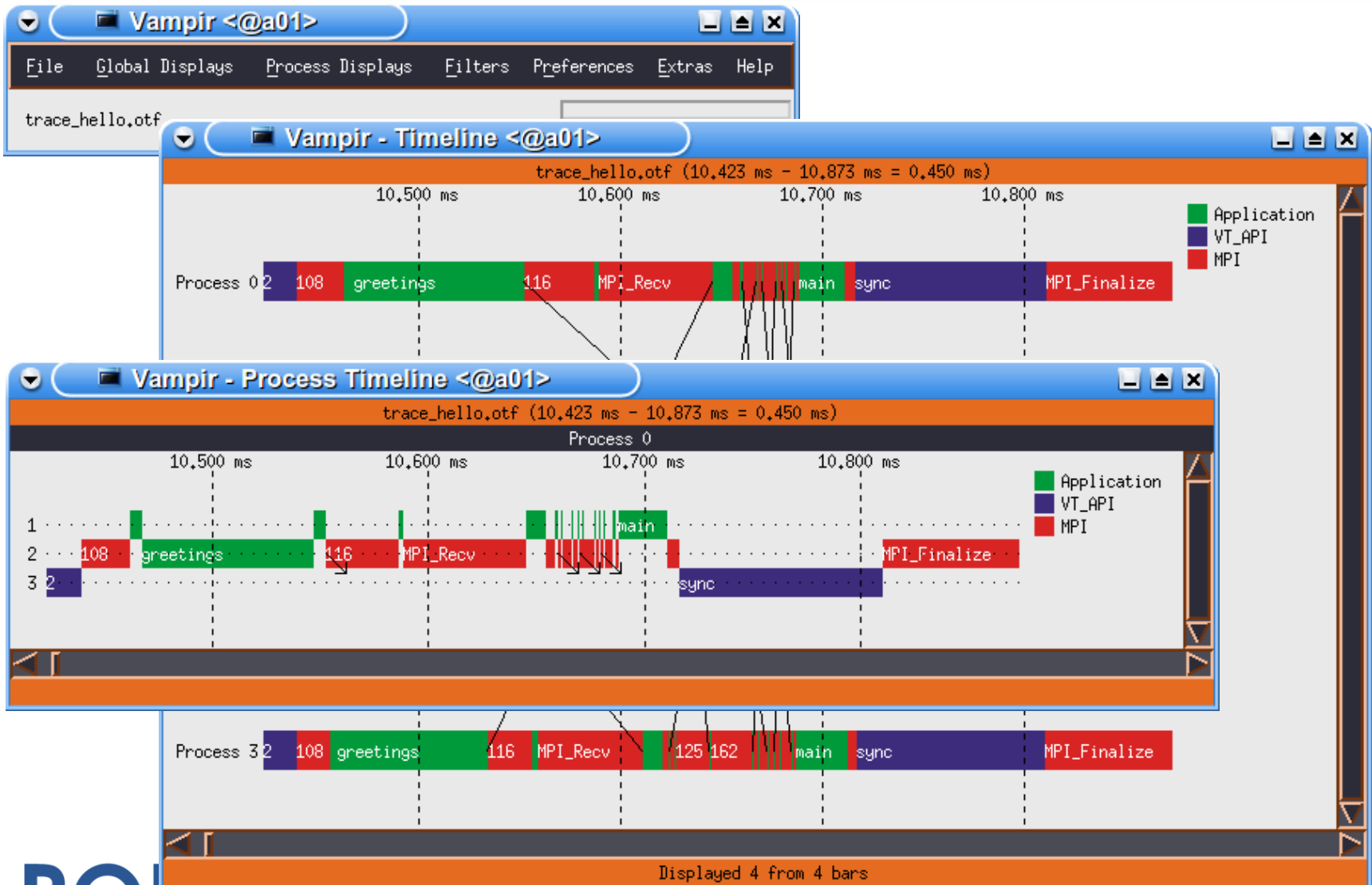
```
%> mpirun -np <X> vngd
```

```
%> vng &
```

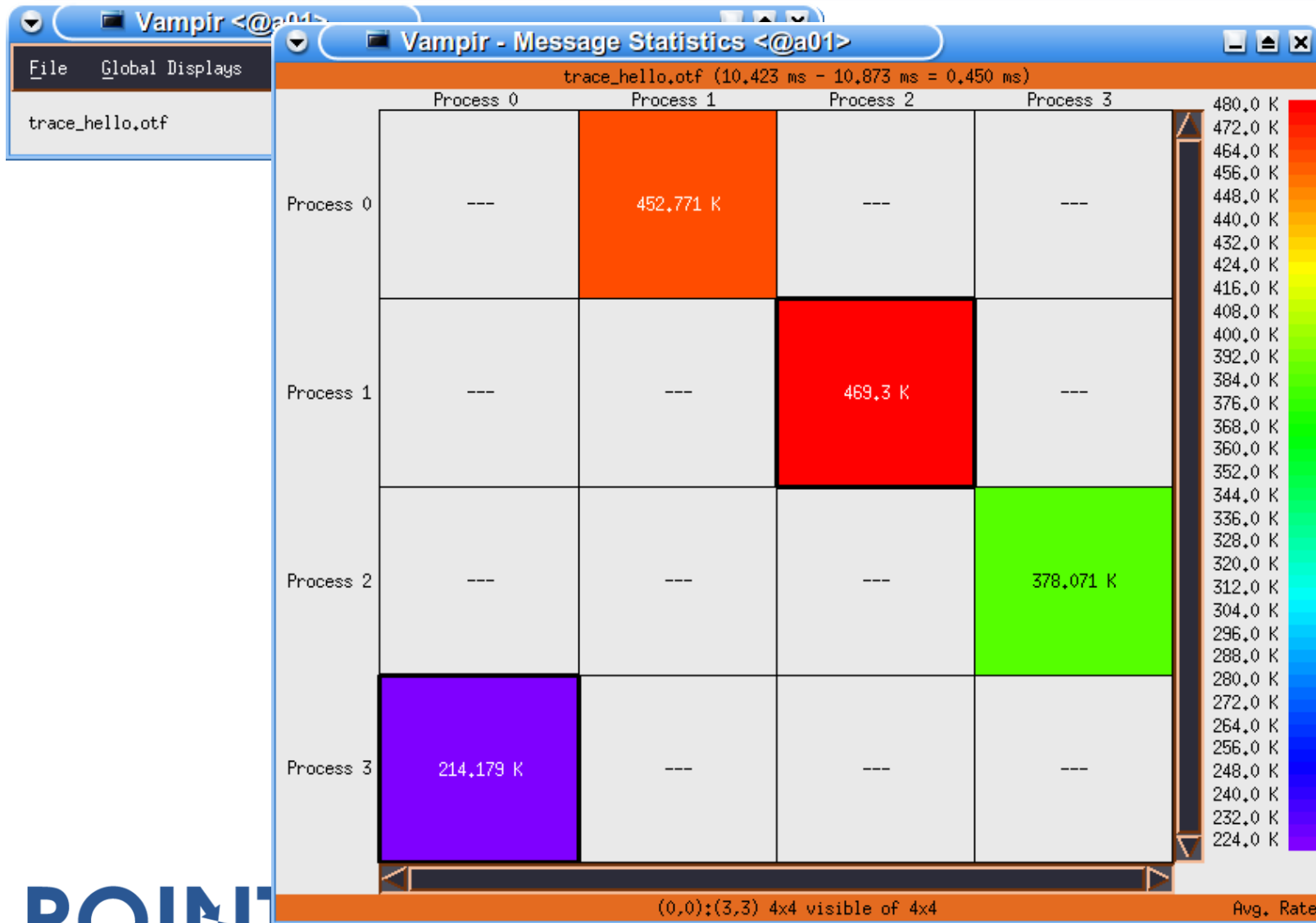
Hands-on: VampirServer



Hands-on: More Displays



Hands-on: More Displays



VAMPIRTRACE: Run-Time Parameters

VampirTrace Run-Time Options

- control options by environment variables:

VT_PFORM_GDIR	Directory for final trace files
VT_PFORM_LDIR	Directory for intermediate files
VT_FILE_PREFIX	Trace file name
VT_BUFFER_SIZE	Internal trace buffer size
VT_MAX_FLUSHES	Max number of buffer flushes
VT_MEMTRACE	Enable memory allocation tracing
VT_IOTRACE	Enable I/O tracing
VT_MPITRACE	Enable MPI tracing
VT_FILTER_SPEC	Name of filter definition file
VT_GROUPS_SPEC	Name of grouping definition file
VT_METRICS	PAPI counter selection

PAPI

- PAPI counters can be included in traces
 - If VampirTrace was build with PAPI support
 - If PAPI is available on the platform
- VT_METRICS specifies a list of PAPI counters

```
export VT_METRICS=PAPI_FP_OPS:PAPI_L2_TCM
```

- see also the PAPI commandes `papi_avail` and `papi_command_line`

Memory Allocation Counters

- Memory allocation counters can be recorded:
 - If build with mem. allocation tracing support
 - If GNU glibc is used on the platform
- intercept glibc functions like “malloc” and “free”
- Environment variable VT_MEMTRACE

```
export VT_MEMTRACE=yes
```

Environment Variables

- I/O counters can be included in traces
 - If VampirTrace was build with I/O tracing support
- Standard I/O calls like “open” and “read” are recorded
- Environment variable VT_IOTRACE

```
export VT_IOTRACE=yes
```

Function Filtering

- Filtering is one of the ways to reduce trace size
- Environment variable VT_FILTER_SPEC

```
export VT_FILTER_SPEC=/home/user/filter.spec
```

- Filter definition file contains a list of filters

```
my_*;test_* -- 1000  
debug_* -- 0  
calculate -- -1  
* -- 1000000
```

- See also the vtfiler tool
 - can generate a customized filter file
 - can reduce the size of existing trace files

Function Grouping

- Groups can be defined for related functions
 - Groups can be assigned different colors, highlighting different activities
- Environment variable VT_GROUPS_SPEC

```
export VT_GROUPS_SPEC=/home/<user>/groups.spec
```

- Groups file contains lists of associated entries

```
CALC=calculate  
MISC=my*;test  
UNKNOWN=*
```

Hands-on: SMG 2000 (homework)

Hands-on: SMG 2000

- prepare:

```
%> tar xvzf 02_smg2000.tar.gz  
%> cd smg2000
```

- check compiler/linker flags in Makefile.include

```
CC=mpicc
```

- compile and run

```
%> make  
%> mpirun -np 8 ./test/smg2000 -P 2 2 2 \  
-n 100 100 100 -c 2.0 3.0 40 -d 3 -solver 3
```

Hands-on: SMG 2000

- instrument by replacing compiler command in Makefile.include:

```
CC=vtcc -vt:cc mpicc -vt:verbose
```

- re-compile and run

```
%> make clean; make  
%> export VT_BUFFER_SIZE="10M"  
%> export VT_FILE_PREFIX="smg1"  
%> mpirun -np 8 ./test/smg2000 -P 2 2 2 \  
-n 100 100 100 -c 2.0 3.0 40 -d 3 -solver 3
```

- open with VampirServer

```
%> mpirun -np <X> vngd
```

```
%> vng &
```

Hands-on: SMG 2000

- change file prefix

```
%> export VT_FILE_PREFIX="smg2"
```

- create files 'groups' and 'filter'

```
hypr_struct=HYPRE_Struct*;hypr_Struct*  
hypr_compute=hypr_Comput*;HYPRE_Comput*  
hypr=hypr_*;HYPRE_*
```

```
hypr_Free -- 100  
hypr_CAlloc;hypr_MAlloc -- 0
```

- no re-compilation, only re-run

```
%> export VT_GROUPS_SPEC=groups  
%> export VT_FILTER_SPEC=filter  
%> mpirun -np 8 ./test/smg2000 -P 2 2 2 \  
-n 100 100 100 -c 2.0 3.0 40 -d 3 -solver 3
```


Hands-on: SMG 2000

- change file prefix

```
%> export VT_FILE_PREFIX="smg3"  
%> unset VT_GROUPS_SPEC  
%> unset VT_FILTER_SPEC
```

- specify counters

```
%> export VT_METRICS=PAPI_FP_OPS:PAPI_L2_TCM
```

- see PAPI commands for available counters:

- papi_avail and papi_native_avail

- no re-compilation, only re-run

```
%> mpirun -np 8 ./test/smg2000 -P 2 2 2 \  
-n 100 100 100 -c 2.0 3.0 40 -d 3 -solver 3
```

Hands-on: SMG 2000+OpenMP

- change Makefile.include for OpenMP support

```
CC = vtcc -openmp -vt:hyb -vt:verbose \  
-vt:opari "-table <pwd>/test/opari.tab.c \  
-rcfile <pwd>/test/opari.rc"  
INCLUDE_CFLAGS= -O -DTIMER_USE_MPI \  
-DHYPRE_USING_OPENMP ## activate OpenMP in SMG
```

- Opari source-to-source instrumentation
- requires special options for multi-dir build
- run with OpenMP

```
%> export VT_FILE_PREFIX="smg4"  
%> export VT_METRICS=PAPI_FP_OPS:PAPI_L2_TCM  
%> export OMP_NUM_THREADS=2  
%> mpirun -np 8 ./test/smg2000 -P 2 2 2 \  
-n 100 100 100 -c 2.0 3.0 40 -d 3 -solver 3
```

Hands-on: SMG 2000

- open SMG2000 trace from previous hands-on homework section or larger pre-prepared trace
- browse and analyze:
 - which function is called most often?
 - which function consumes most run-time?
 - identify a single iteration in the main solver
 - which function consumes most run-time in a single iteration?
 - which function achieves highest FLOP/s rate?
 - what is the average MPI communication speed?

Hands-on: Mandelbrot Example with Performance Counters

Hands-on: Performance Counters

```
%> tar xvzf 03_mandelbrot.tgz; cd 03_mandelbrot
```

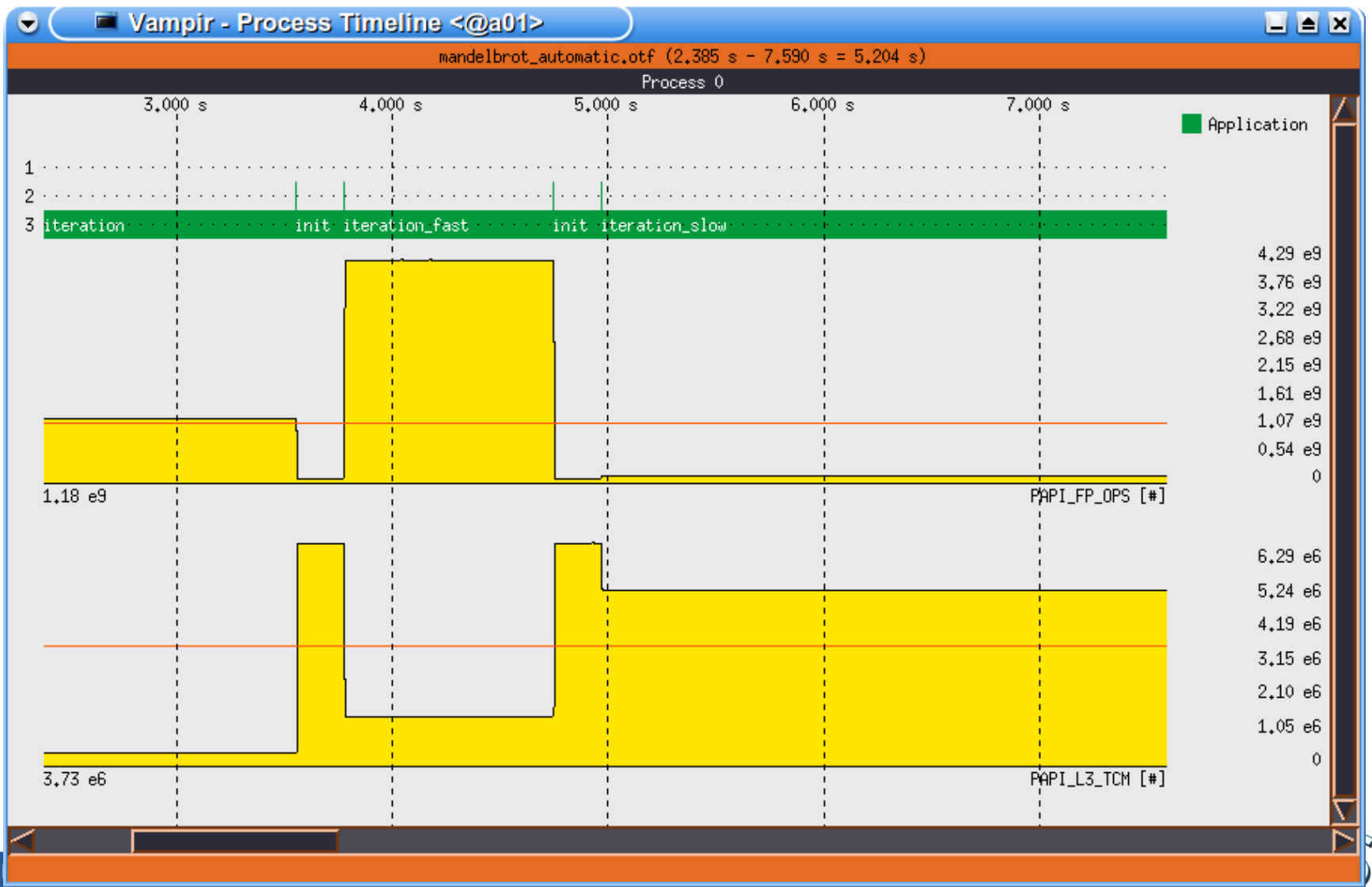
– tasks:

- modify Makefile for compiler instrumentation
- specify counters for Flops and cache misses
- set the file name prefix to “mdlb1” and the buffer size to 10M
- run instrumented program:

```
%> ./mandelbrot -2.0 -1.0 1.0 1.0 0.001 50
```

- open trace with VampirServer
- investigate three phases of the program
- how does performance differ? and why?

Hands-on: Performance Counters



Extra: Manual Instrumentation

- change the file name prefix to “mdl2”
- insert additional manual instrumentation (see also VampirTrace documentation)

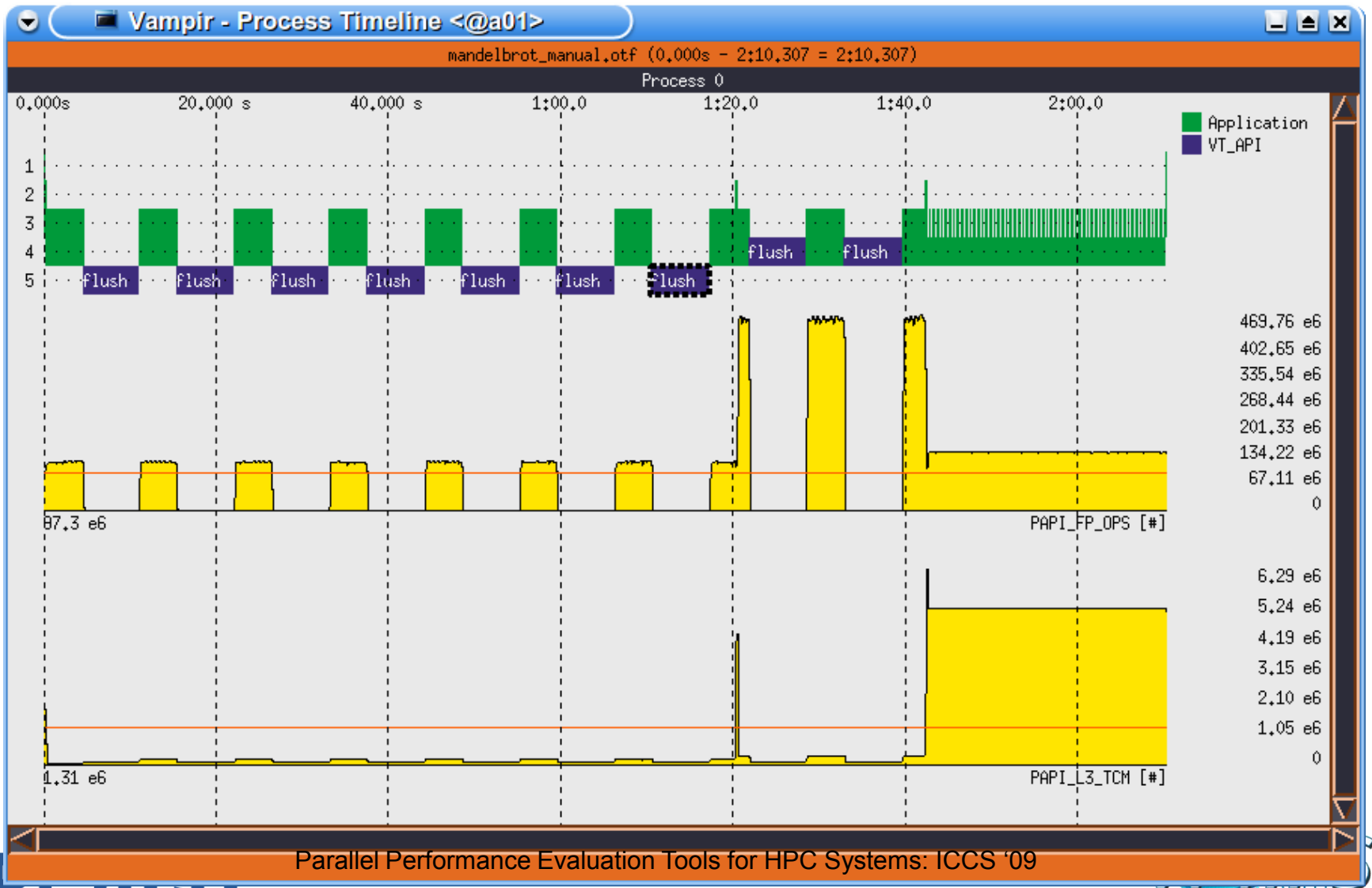
```
#include "vt_user.h"  
  
VT_USER_START("name");  
...  
VT_USER_END("name");
```

- catch all iterations of the outer loops
- re-compile
- re-run

Extra: Manual Instrumentation

- open new trace in Vampir
- what's wrong with the resulting trace?
 - fix it with `VT_MAX_FLUSHES` and `VT_BUFFER_SIZE`
 - is it really better now?
 - can one see more with more data?
- advanced: add a user defined counter that records the loop counter variable

Extra: Manual Instrumentation



Finding Performance Bottlenecks

Finding Bottlenecks

- Trace Visualization
 - Vampir provides a number of display types
 - each allows many different options
- Advice
 - identify essential parts of an application (initialization, main iteration, I/O, finalization)
 - identify important components of the code (serial computation, MPI P2P, collective MPI, OpenMP)
 - make a hypothesis about performance problems
 - consider application's internal workings if known
 - select the appropriate displays
 - use statistic displays in conjunction with timelines

Finding Bottlenecks

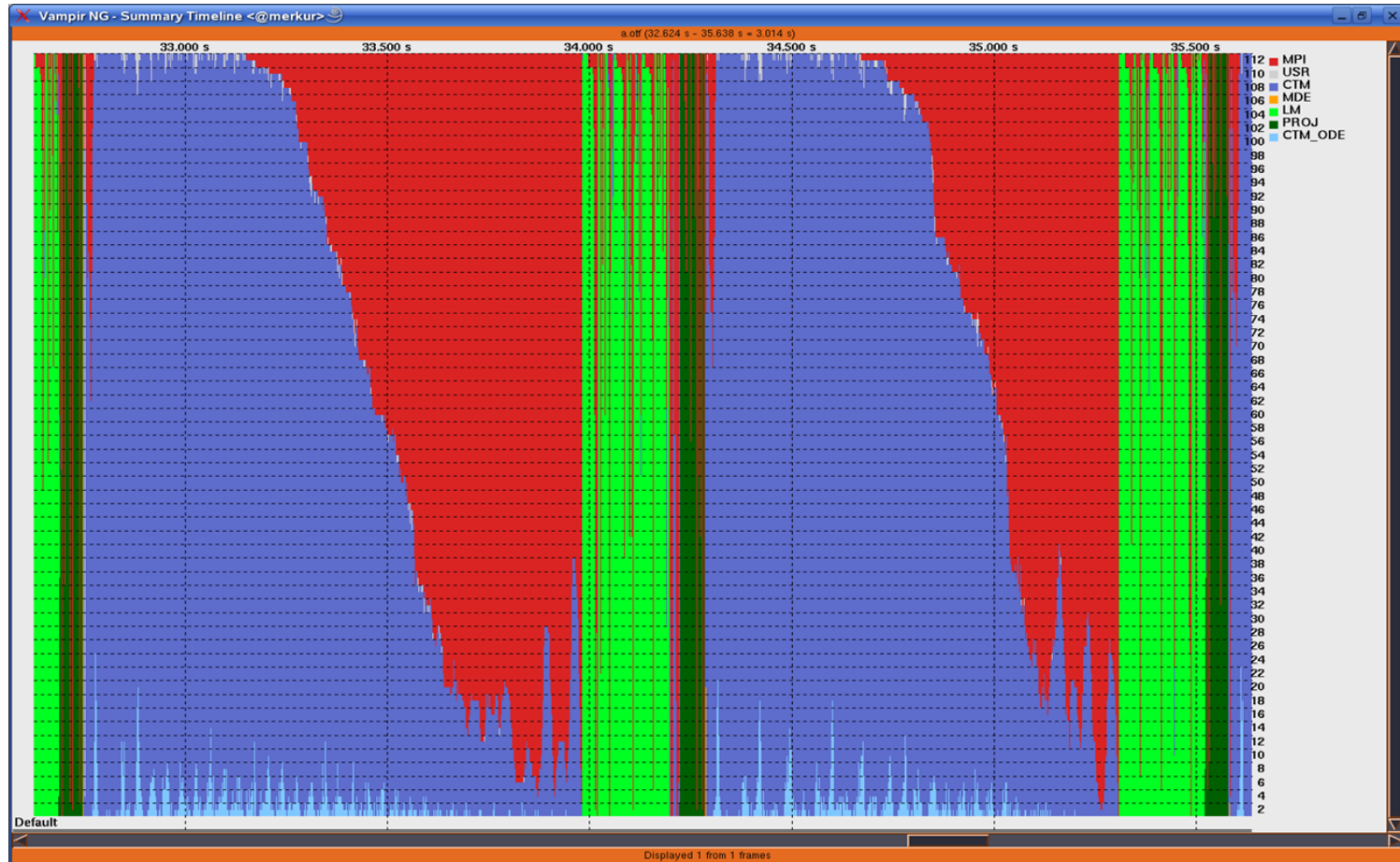
- Communication
- Computation
- Memory, I/O, etc
- Tracing itself

Bottlenecks in Communication

- communication as such (dominating over computation)
- late sender, late receiver
- point-to-point messages instead of collective communication
- unmatched messages
- overcharge of MPI's buffers
- bursts of large messages (bandwidth)
- frequent short messages (latency)
- unnecessary synchronization (barrier)

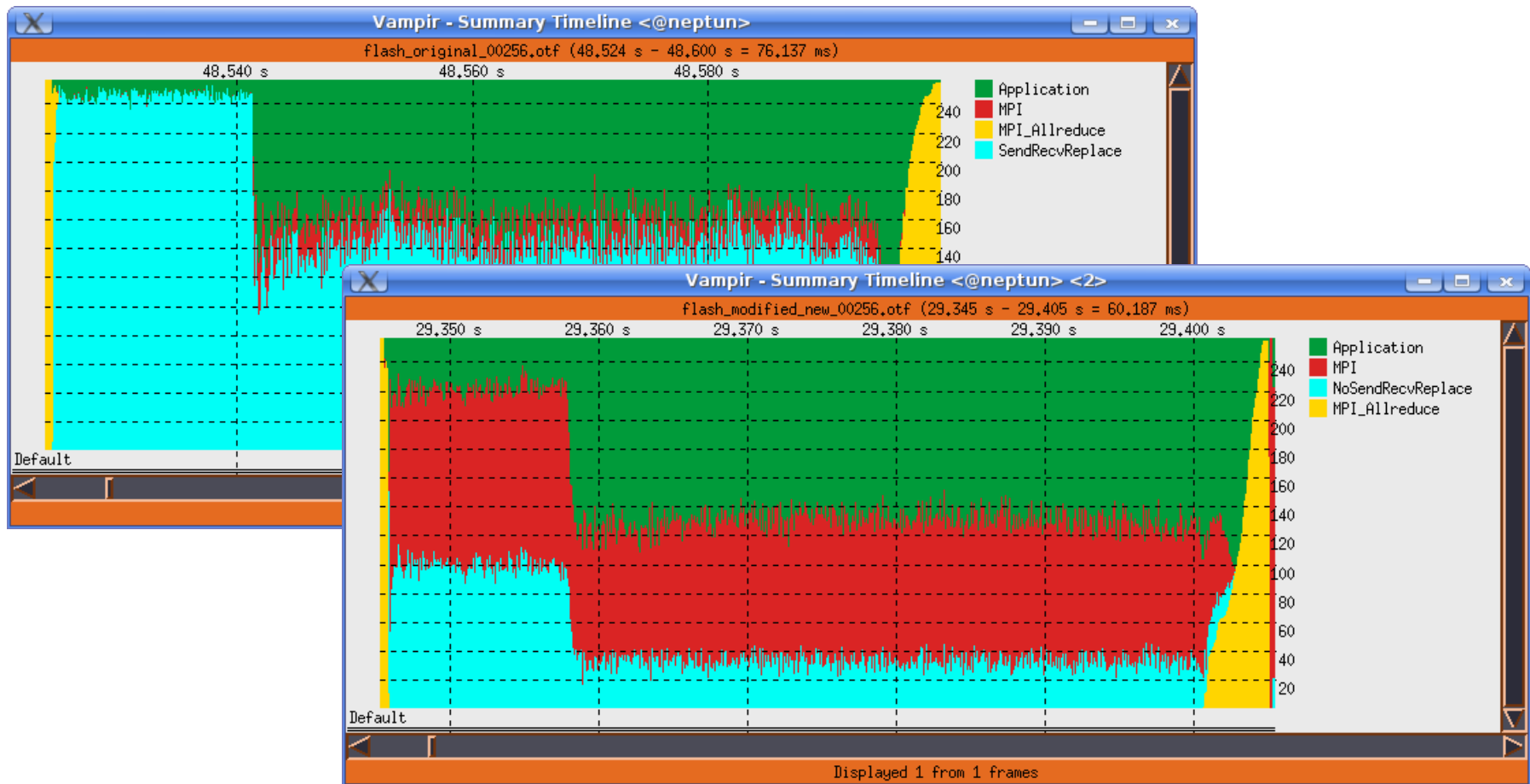
all of the above usually result in high MPI time share

Bottlenecks in Communication



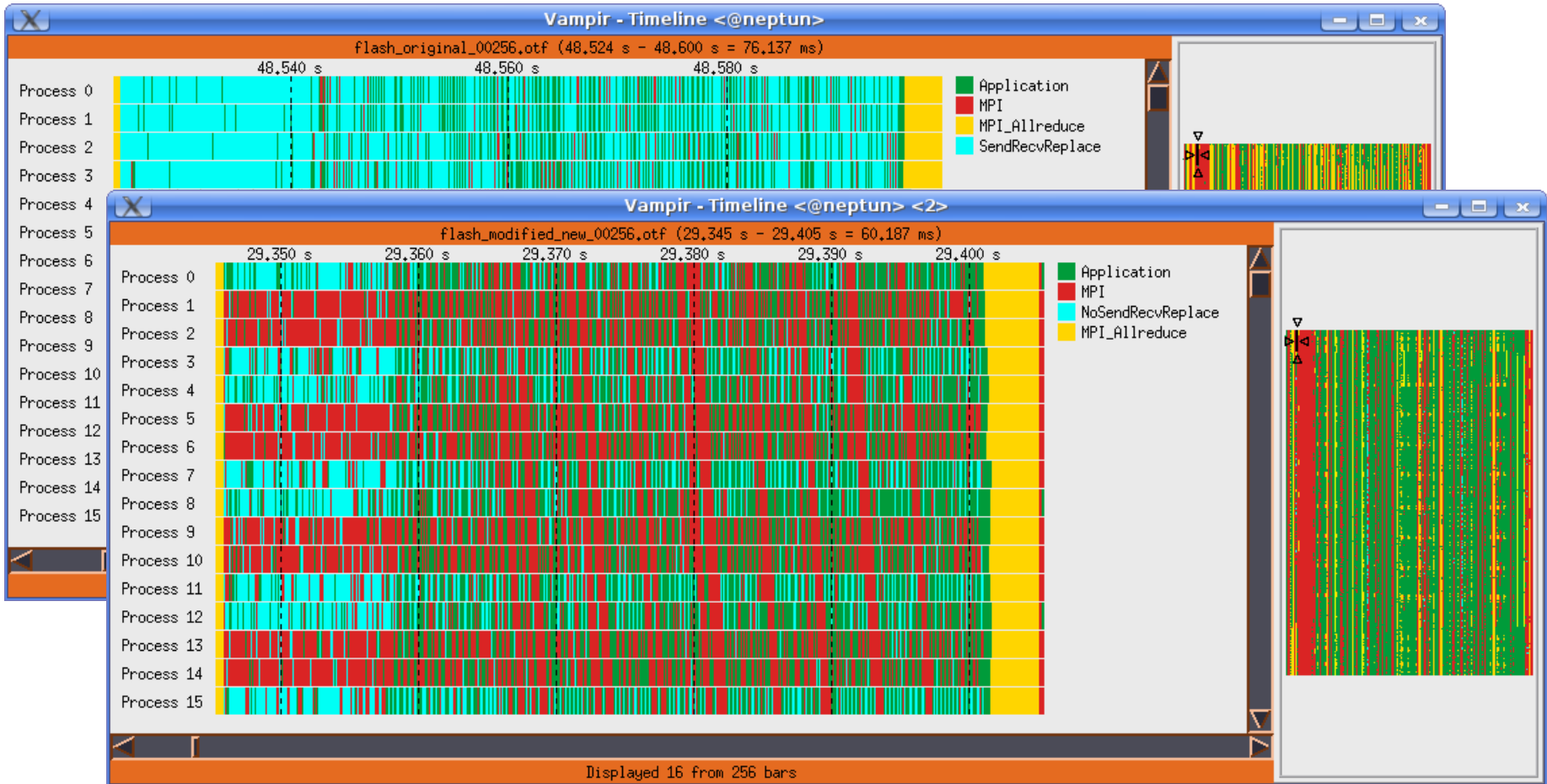
Example: prevalent communication

Bottlenecks in Communication



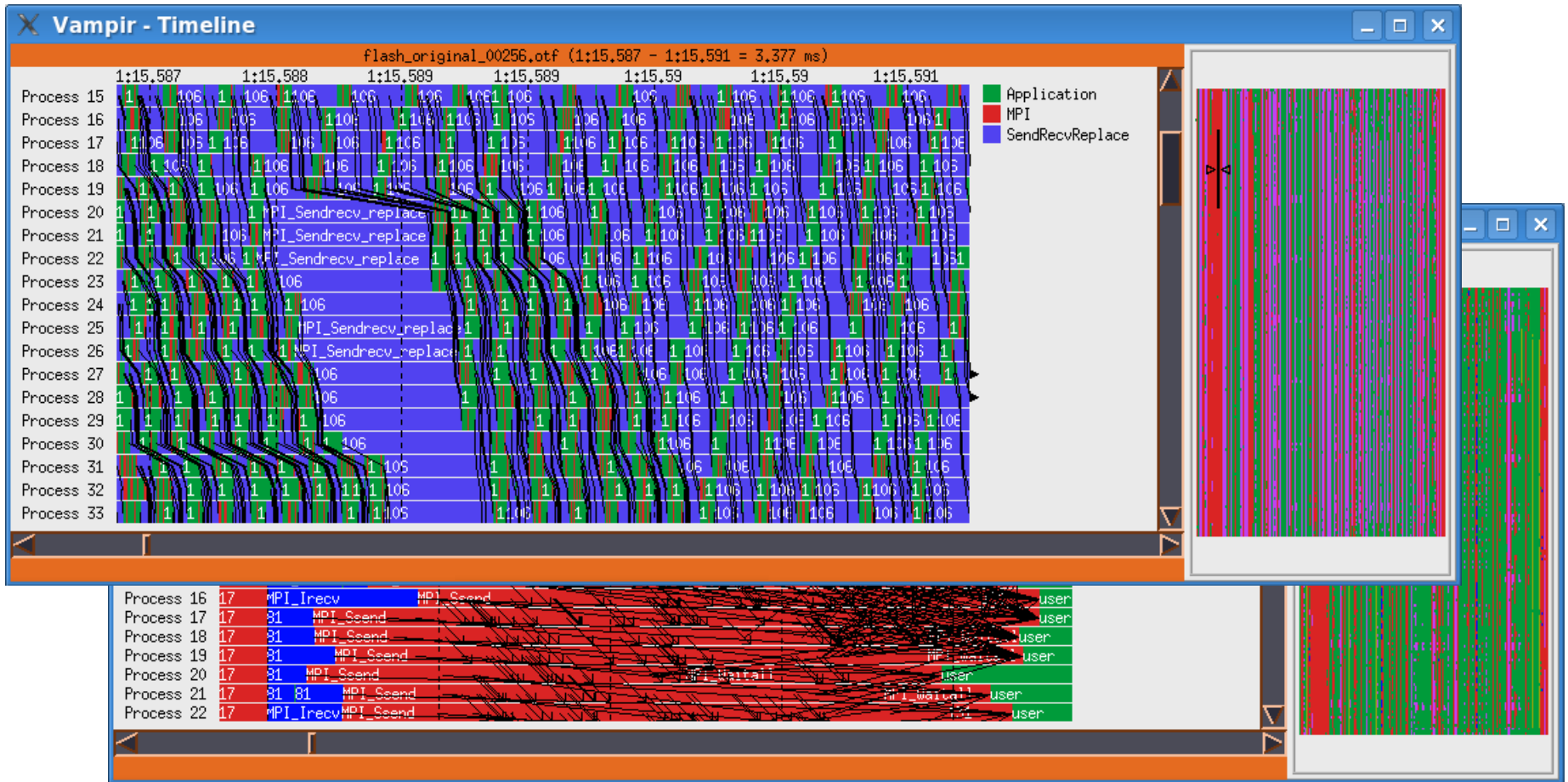
prevalent communication: MPI_Allreduce

Bottlenecks in Communication



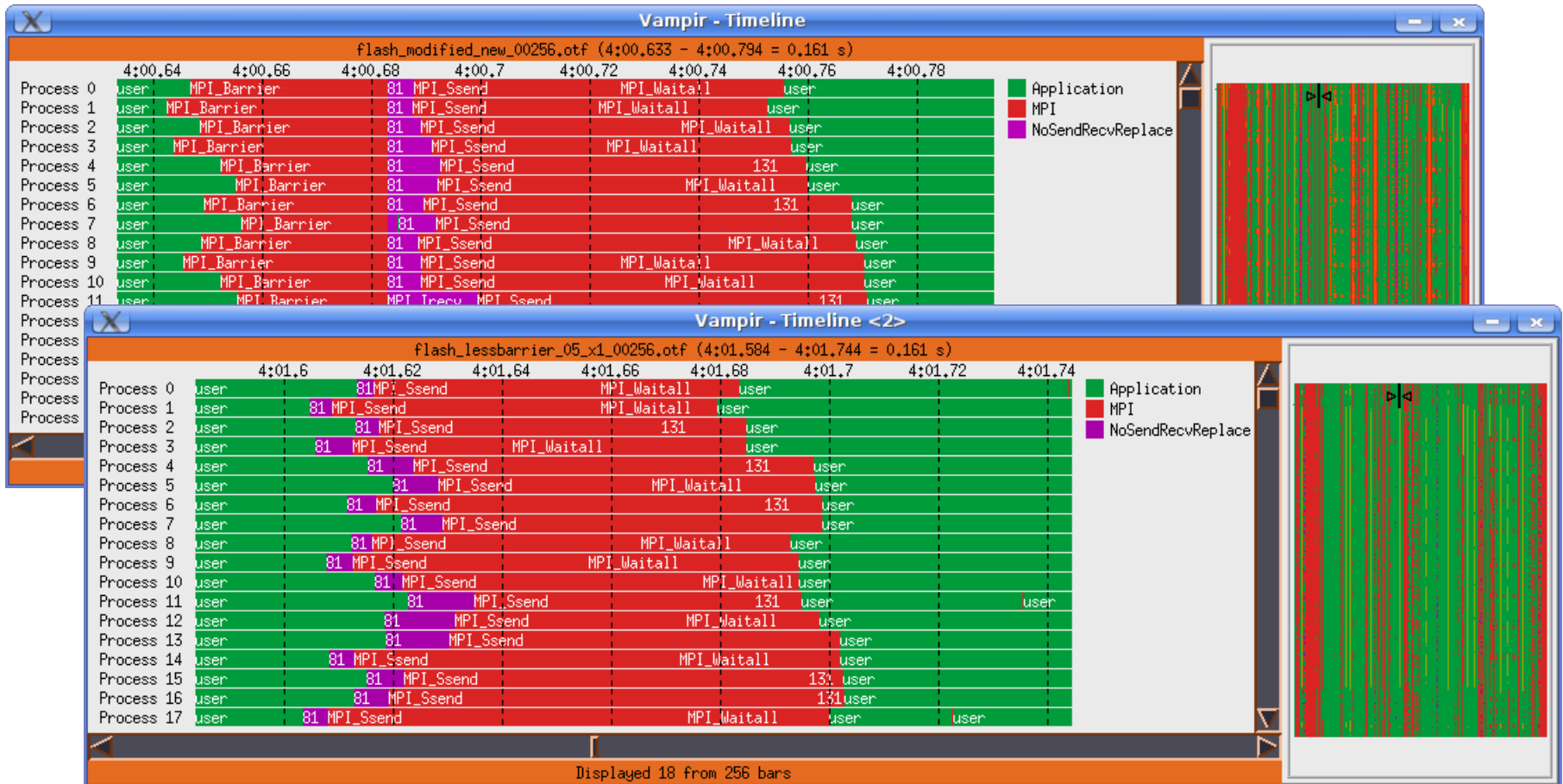
prevalent communication: timeline view

Bottlenecks in Communication



Propagated Delays in MPI_SendReceiveReplace

Bottlenecks in Communication



unnecessary MPI_Barriers

Bottlenecks in Communication

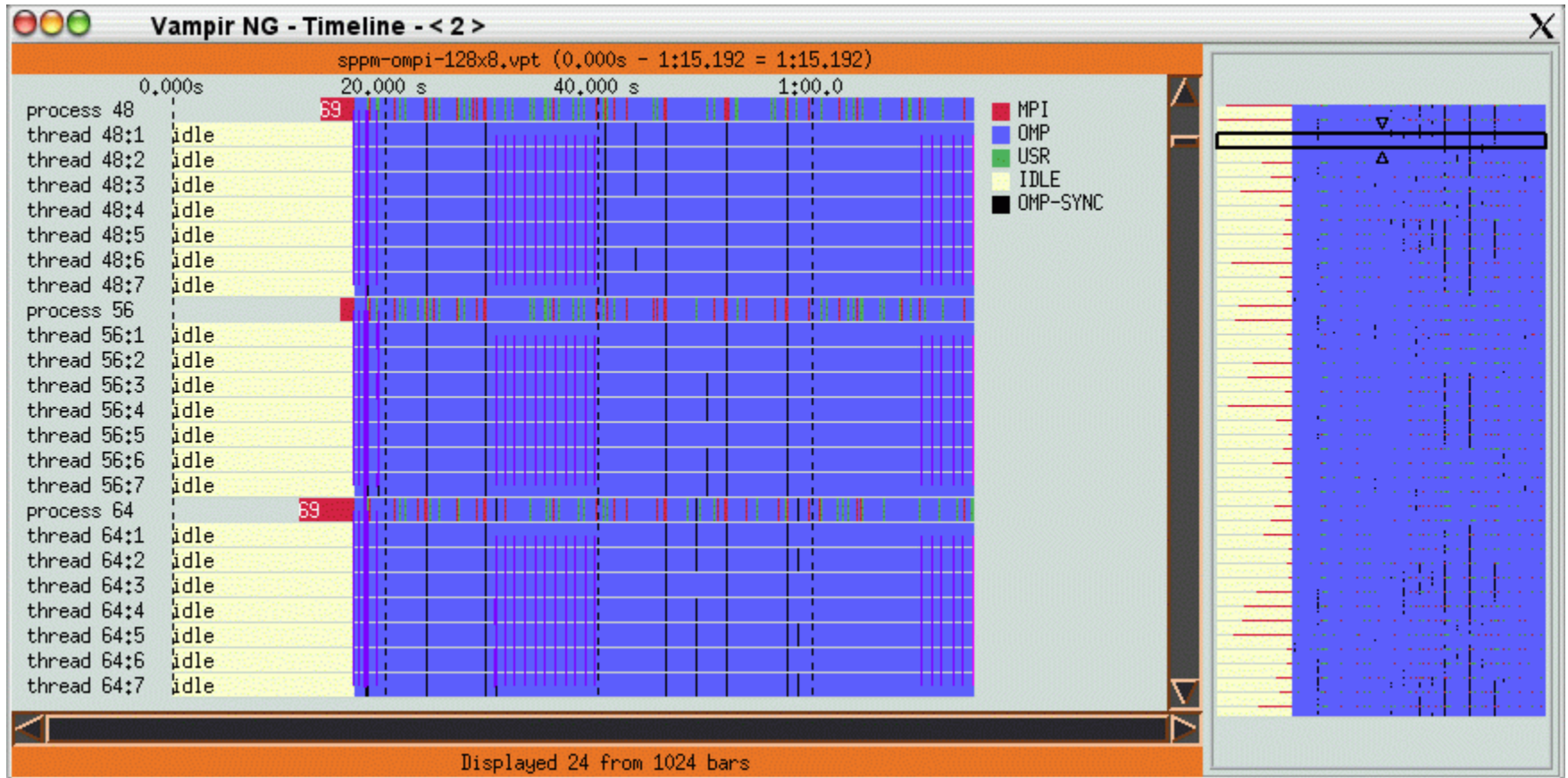


Patterns of Successive MPI_Allreduce Calls

Further Bottlenecks

- unbalanced computation
 - single late comer
- strictly serial parts of program
 - idle processes/threads
- very frequent tiny function calls
- sparse loops

Further Bottlenecks

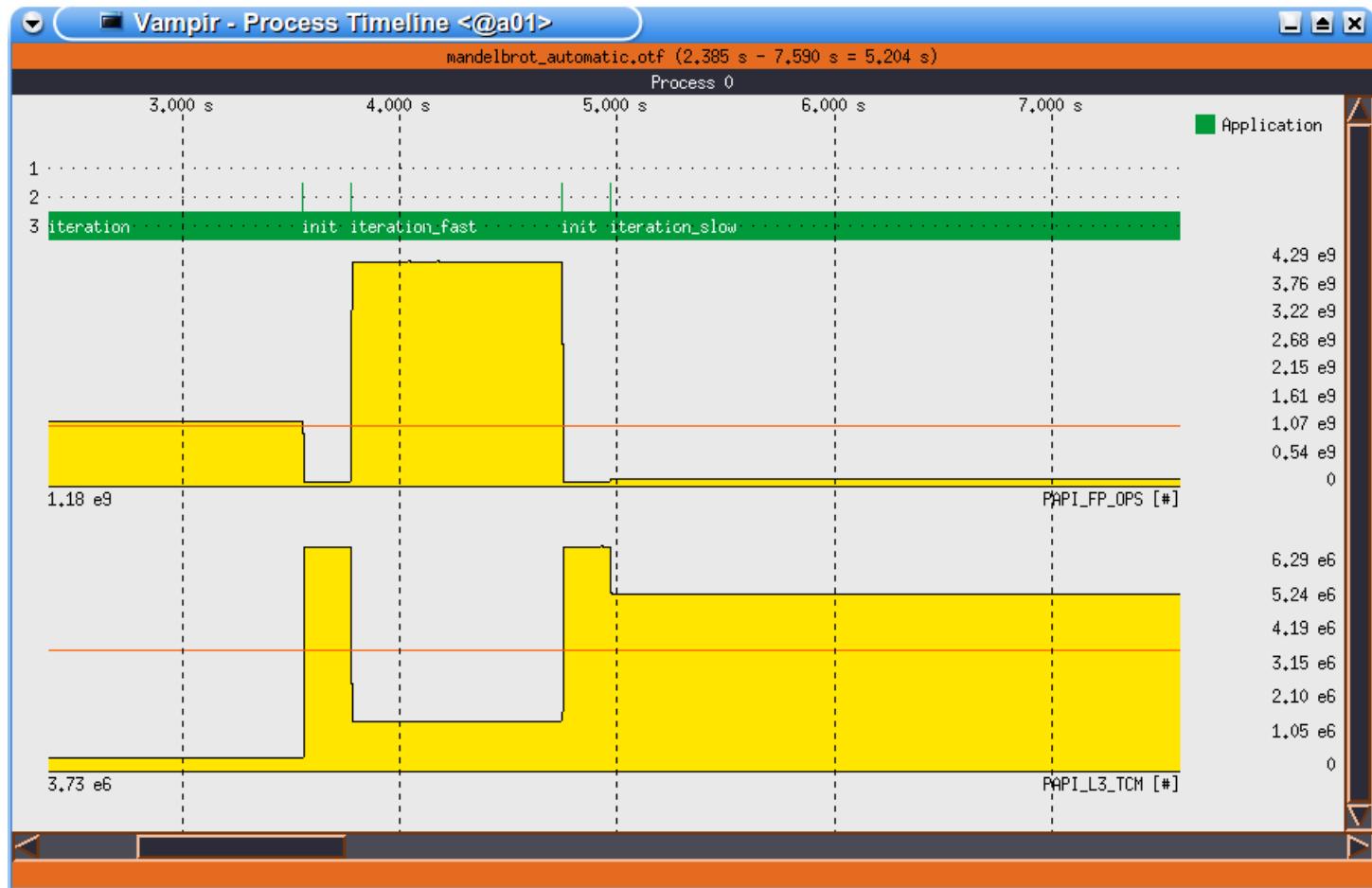


Example: Idle OpenMP threads

Bottlenecks in Computation

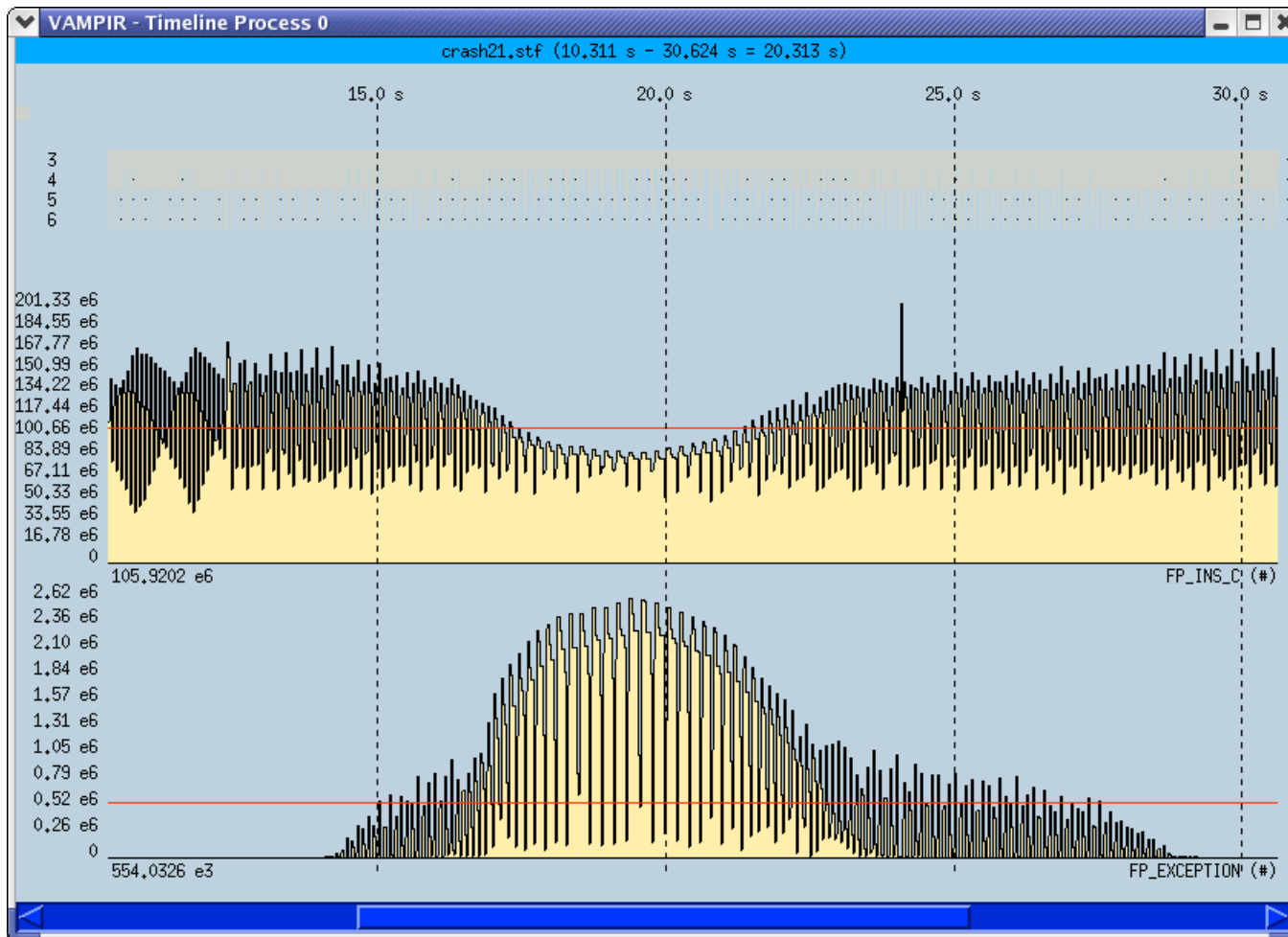
- memory bound computation
 - inefficient L1/L2/L3 cache usage
 - TLB misses
 - detectable via HW performance counters
- I/O bound computation
 - slow input/output
 - sequential I/O on single process
 - I/O load imbalance
- exception handling

Bottlenecks in Computation



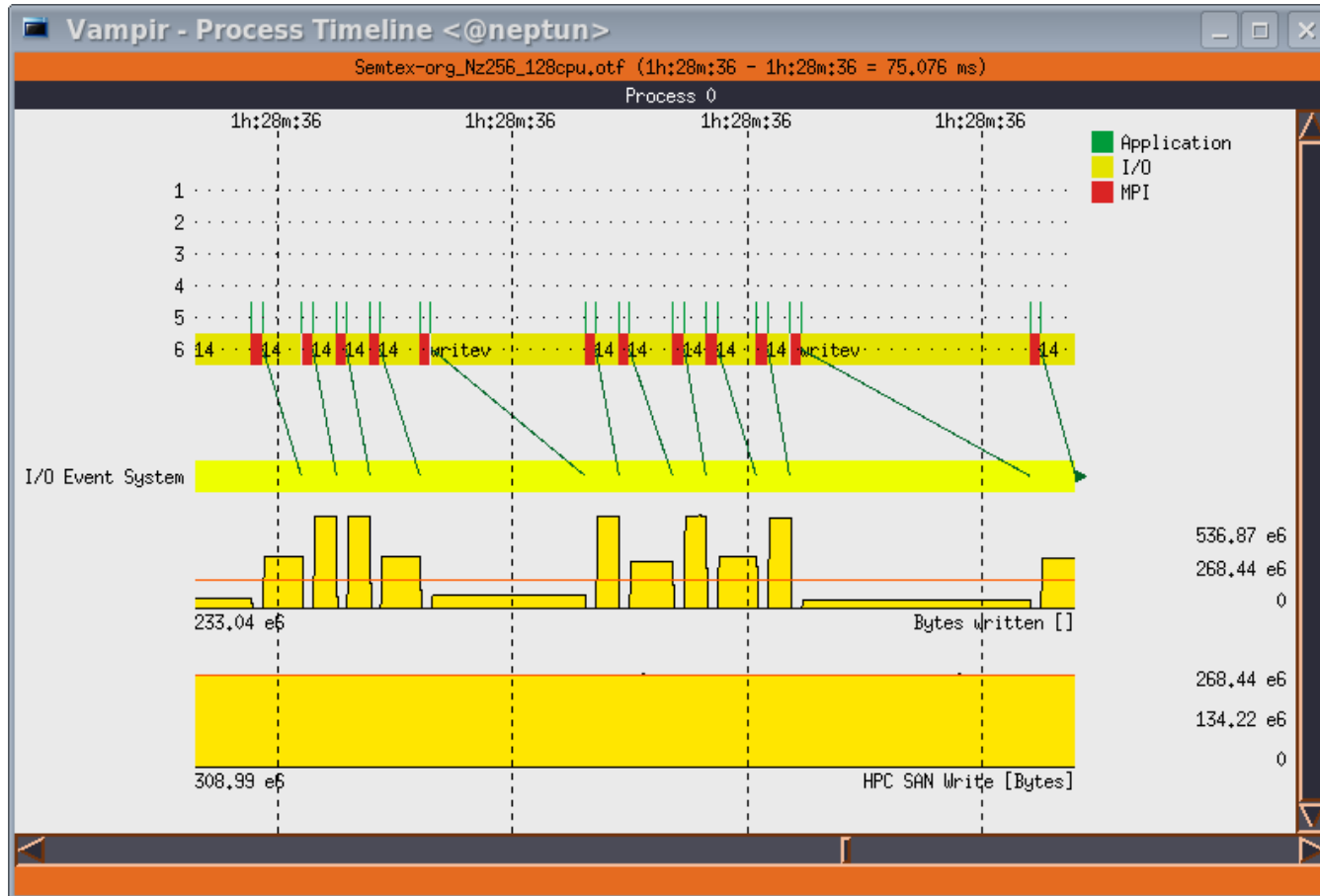
low FP rate due to heavy cache misses

Bottlenecks in Computation



low FP rate due to heavy FP exceptions

Bottlenecks in Computation

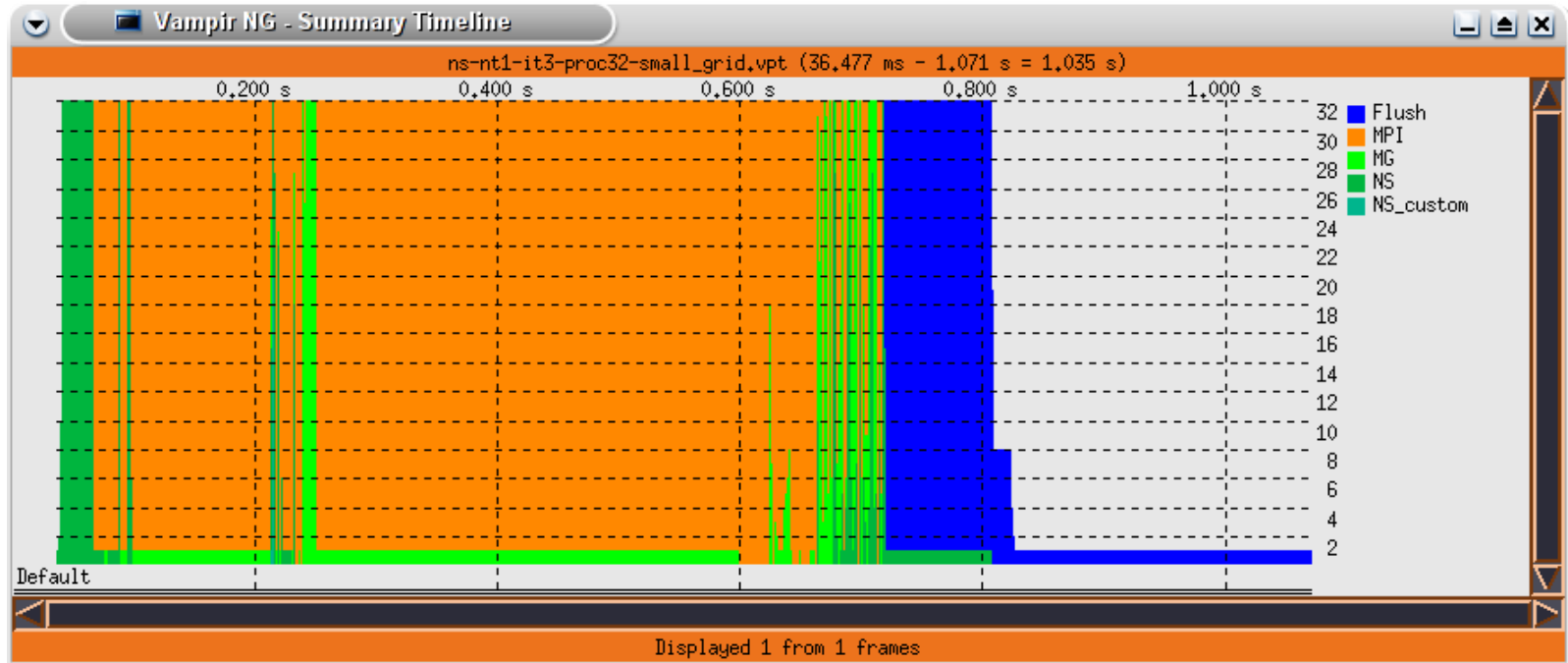


irregular slow I/O operations

Effects due to Tracing Itself

- measurement overhead
 - esp. grave for tiny function calls
 - solve with selective instrumentation
- long/frequent/asynchronous trace buffer flushes
- too many concurrent counters
- heisenbugs

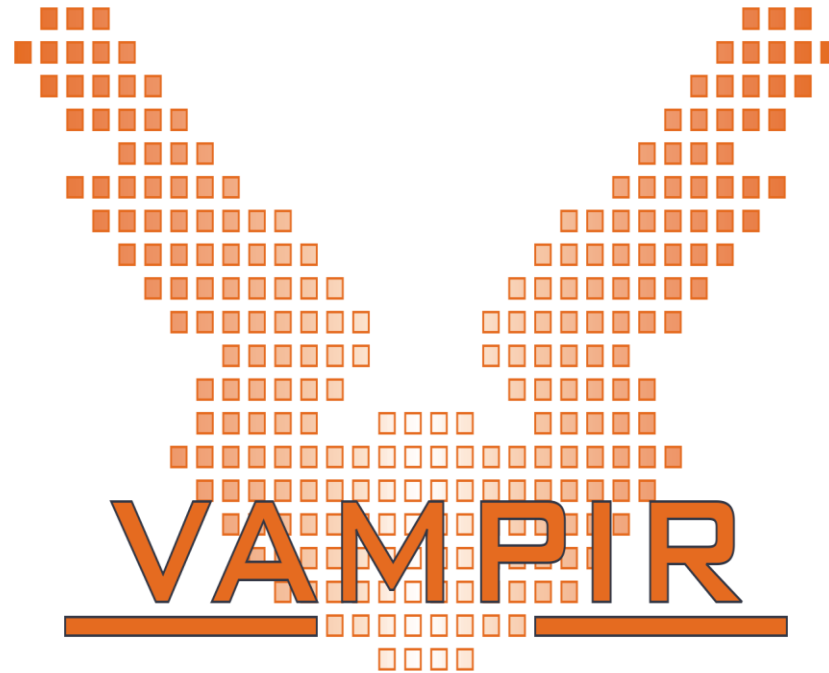
Effects due to Tracing Itself



Trace buffer flushes are explicitly marked in the trace. It is rather harmless at the end of a trace as shown here.

Conclusion and Outlook

- performance analysis very important in HPC
- use performance analysis tools for profiling and tracing
- do not spend effort in DIY solutions, e.g. like printf-debugging
- use tracing tools with some precautions
 - overhead
 - data volume
- let us know about problems and about feature wishes
- vampirsupport@zih.tu-dresden.de



Vampir and VampirTraces are available at <http://www.vampir.eu> and <http://www.tu-dresden.de/zih/vampirtrace/> , get support via vampirsupport@zih.tu-dresden.de