ISC’18 Tutorial:
Hands-on Practical Hybrid Parallel Application Performance Engineering

Christian Feld
Jülich Supercomputing Centre

Markus Geimer
Jülich Supercomputing Centre

Sameer Shende
University of Oregon

Ronny Tschüter
TU Dresden

Agenda (morning)

Time	Topic	Presenter
09:00	Introduction to VI-HPS & parallel performance engineering	Geimer/Shende
09:45	Setup for hands-on exercises with Live-ISO/OVA & Stampede2	all
10:00	Instrumentation & measurement of applications with ***Score-P***	Feld/Tschüter
10:30	Exploration & visualization of call-path profiles with ***CUBE***	Geimer
11:00	*Coffee break*	
11:30	Configuration & customization of ***Score-P*** measurements	Feld/Tschüter
12:00	Examination & visualization of profiles with ***TAU***	Shende
12:45	Specialized ***Score-P*** measurements and analyses	Feld
13:00	*Lunch break*	

Agenda (afternoon)

Time	Topic	Presenter
14:00	Automated analysis of traces for inefficiencies with ***Scalasca***	Geimer
14:45	Interactive visualization and time-interval statistics with ***Vampir***	Tschüter
15:30	Specialized ***Score-P*** measurements and analyses	Feld
16:00	*Coffee break*	
16:30	Performance data management with *TAU* ***PerfExplorer***	Shende
16:45	Parallel application performance analysis case studies	all
17:45	Review & conclusion	Geimer
18:00	*Adjourn*	

Virtual Institute – High Productivity Supercomputing

- **Goal**: Improve the quality and accelerate the development process of complex simulation codes running on highly-parallel computer systems
- Start-up funding (2006–2011) by Helmholtz Association of German Research Centres
- Activities
  - Development and integration of HPC programming tools
    - Correctness checking & performance analysis
  - Academic workshops
  - Training workshops
  - Service
    - Support email lists
    - Application engagement

HELMHOLTZ
RESEARCH FOR GRAND CHALLENGES

http://www.vi-hps.org

VI-HPS partners (founders)

Forschungszentrum Jülich

- Jülich Supercomputing Centre

RWTH Aachen University

- Centre for Computing & Communication

Technische Universität Dresden

- Centre for Information Services & HPC

University of Tennessee (Knoxville)

- Innovative Computing Laboratory

VI-HPS partners (cont.)

Allinea Software Ltd.

- Now part of ARM

Barcelona Supercomputing Center

- Centro Nacional de Supercomputación

Lawrence Livermore National Lab.

- Center for Applied Scientific Computing

Leibniz Supercomputing Centre

Technical University of Darmstadt

- Laboratory for Parallel Programming

VI-HPS partners (cont.)

Technical University of Munich

- Chair for Computer Architecture

TECHNISCHE UNIVERSITÄT MÜNCHEN

University of Oregon

- Performance Research Laboratory

UNIVERSITY OF OREGON

University of Stuttgart

- HPC Centre

Universität Stuttgart

University of Versailles St-Quentin

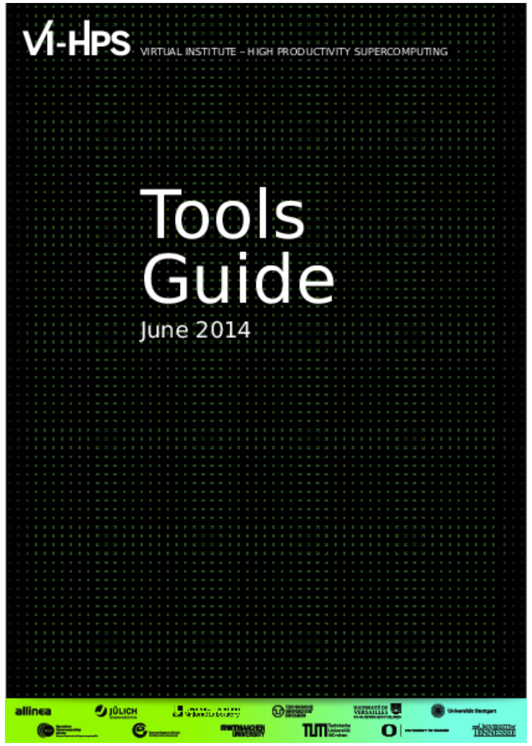
- LRC ITACA

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

Productivity tools

- MUST / ARCHER
  - MPI & OpenMP usage correctness checking
- PAPI
  - Interfacing to hardware performance counters
- Periscope Tuning Framework
  - Automatic analysis and tuning
- **Scalasca**
  - Large-scale parallel performance analysis
- **TAU**
  - Integrated parallel performance system
- **Vampir**
  - Interactive graphical trace visualization & analysis
- **Score-P**
  - Community-developed instrumentation & measurement infrastructure

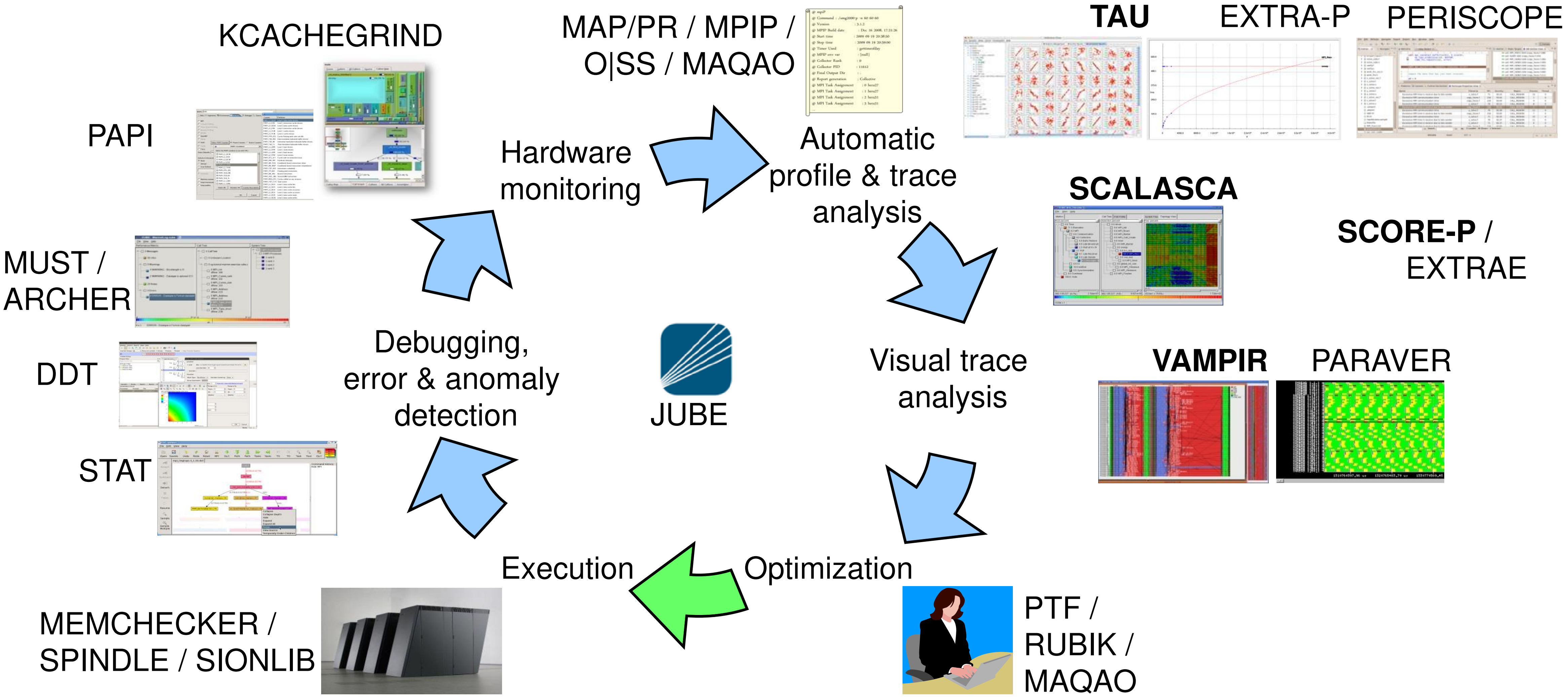
For a brief overview of tools consult the VI-HPS Tools Guide:



Productivity tools (cont.)

- DDT/MAP/PR: Parallel debugging, profiling & performance reports
- Extra-P: Automated performance modelling
- JUBE: Automatic workflow execution for benchmarking, testing & production
- Kcachegrind: Callgraph-based cache analysis [x86 only]
- MAQAO: Assembly instrumentation & optimization [x86-64 only]
- mpiP/mpiPview: MPI profiling tool and analysis viewer
- Open MPI: Integrated memory checking
- Open|SpeedShop: Integrated parallel performance analysis environment
- Paraver/Dimemas/Extrae: Event tracing and graphical trace visualization & analysis
- Rubik: Process mapping generation & optimization [BG only]
- SIONlib/Spindle: Optimized native parallel file I/O & shared library loading
- STAT: Stack trace analysis tools

Technologies and their integration

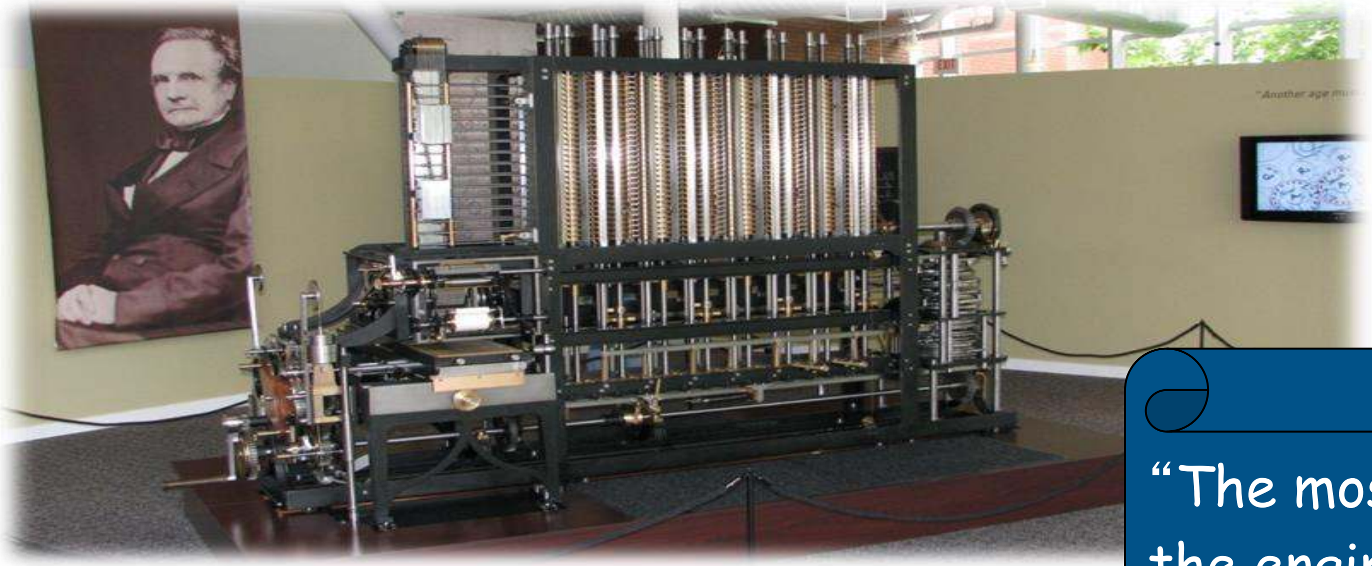


Introduction to Parallel Performance Engineering

Sameer Shende
University of Oregon

(with content used with permission from tutorials by Bernd Mohr/JSC and Luiz DeRose/Cray)

Performance: an old problem



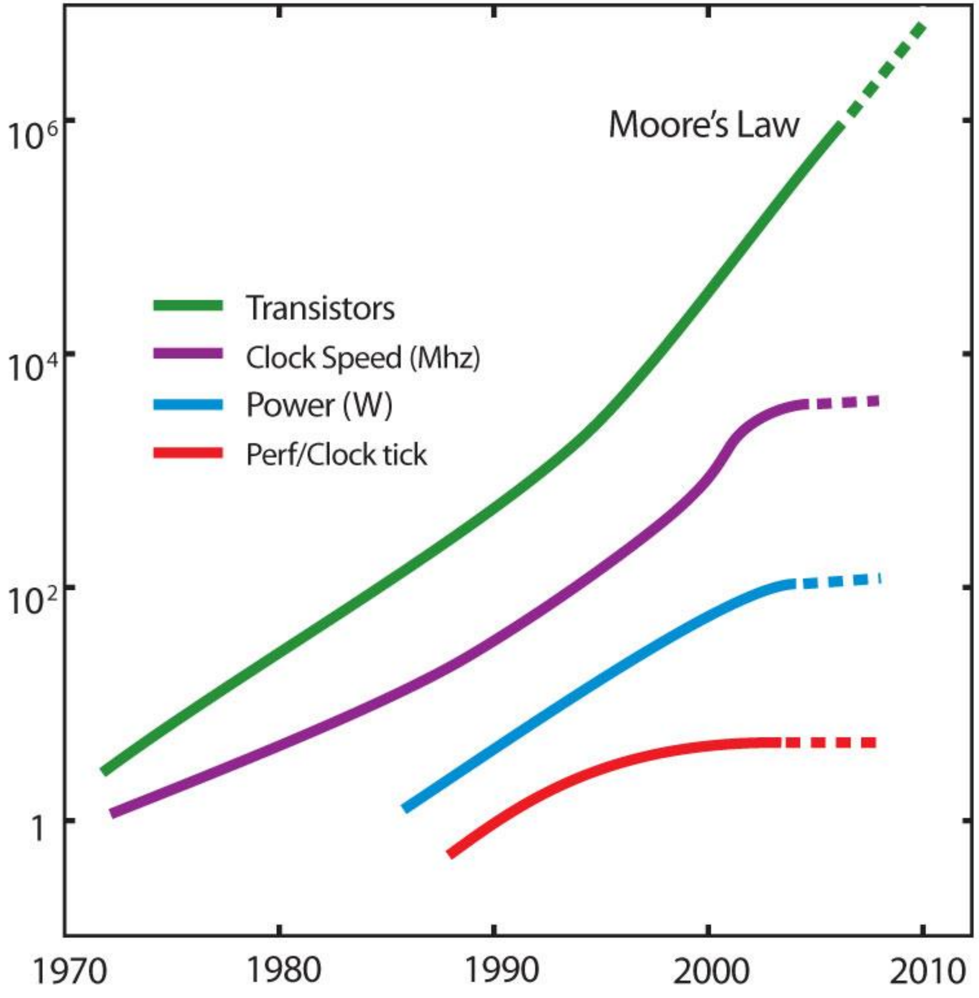
Difference Engine

> "The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible."
>
> Charles Babbage
> 1791 – 1871

Today: the "free lunch" is over

- Moore's law is still in charge, but
  - Clock rates no longer increase
  - Performance gains only through increased parallelism
- Optimizations of applications more difficult
  - Increasing application complexity
    - Multi-physics
    - Multi-scale
  - Increasing machine complexity
    - Hierarchical networks / memory
    - More CPUs / multi-core

☞ Every doubling of scale reveals a new bottleneck!



Performance factors of parallel applications

- "Sequential" performance factors
  - Computation
    - ☞ Choose right algorithm, use optimizing compiler
  - Cache and memory
    - ☞ Tough! Only limited tool support, hope compiler gets it right
  - Input / output
    - ☞ Often not given enough attention

- "Parallel" performance factors
  - Partitioning / decomposition
  - Communication (i.e., message passing)
  - Multithreading
  - Synchronization / locking
    - ☞ More or less understood, good tool support

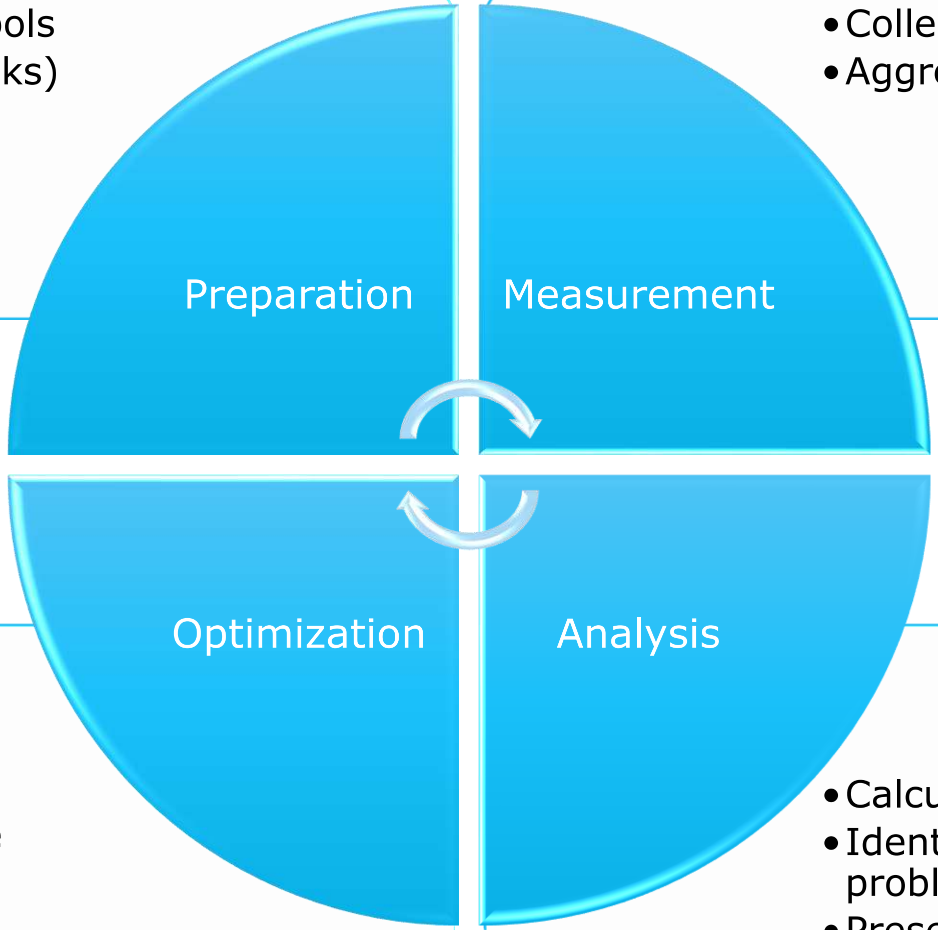
Tuning basics

- Successful engineering is a combination of
  - Careful setting of various tuning parameters
  - The right algorithms and libraries
  - Compiler flags and directives
  - …
  - Thinking !!!
- Measurement is better than guessing
  - To determine performance bottlenecks
  - To compare alternatives
  - To validate tuning decisions and optimizations
    - ☞ After each step!

Performance engineering workflow

- Prepare application with symbols
- Insert extra code (probes/hooks)

- Collection of performance data
- Aggregation of performance data



- Modifications intended to eliminate/reduce performance problem

- Calculation of metrics
- Identification of performance problems
- Presentation of results

The 80/20 rule

- Programs typically spend 80% of their time in 20% of the code

- Programmers typically spend 20% of their effort to get 80% of the total speedup possible for the application
  - ☞ *Know when to stop!*

- Don't optimize what does not matter
  - ☞ *Make the common case fast!*

> "If you optimize everything, you will always be unhappy."
>
> Donald E. Knuth

Metrics of performance

- What can be measured?
  - A **count** of how often an event occurs
    - E.g., the number of MPI point-to-point messages sent
  - The **duration** of some interval
    - E.g., the time spent these send calls
  - The **size** of some parameter
    - E.g., the number of bytes transmitted by these calls

- Derived metrics
  - E.g., rates / throughput
  - Needed for normalization

Example metrics

- Execution time
- Number of function calls
- CPI
  - CPU cycles per instruction
- FLOPS
  - Floating-point operations executed per second

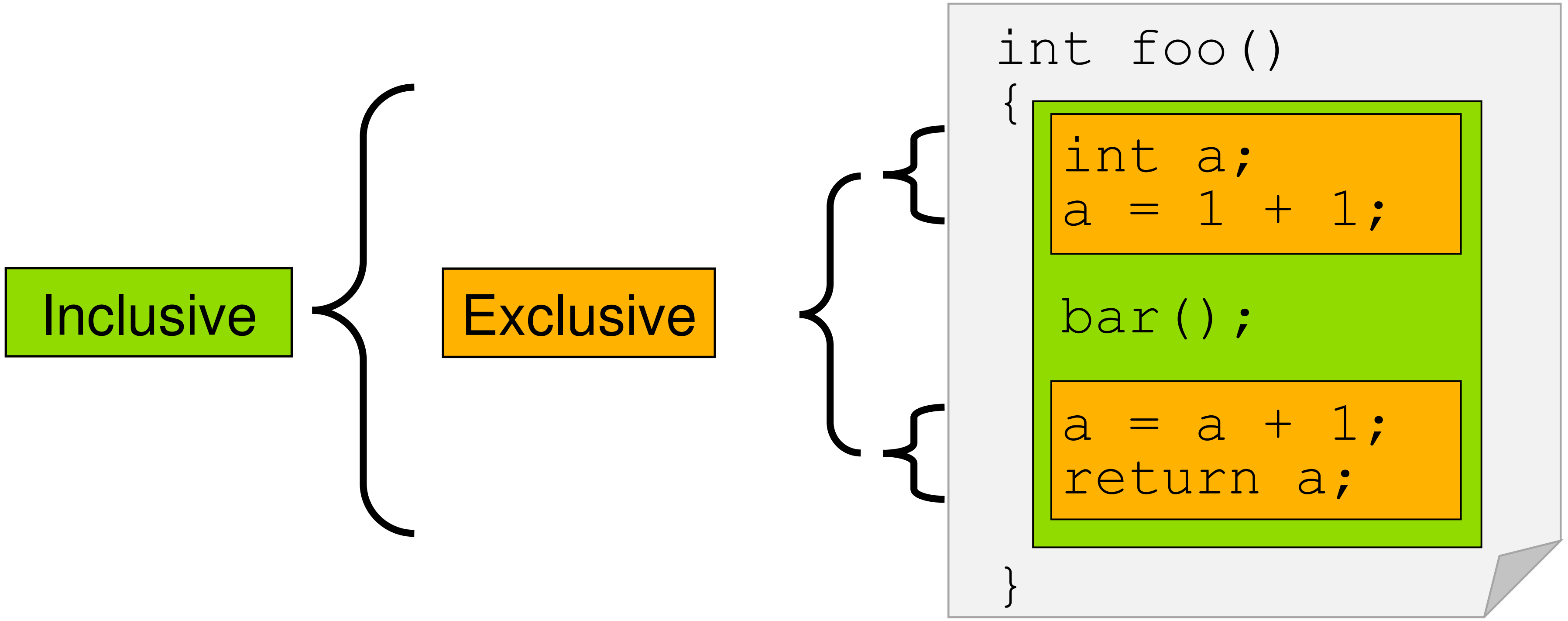
"math" Operations?
HW Operations?
HW Instructions?
32-/64-bit? …

Execution time

- Wall-clock time
  - Includes waiting time: I/O, memory, other system activities
  - In time-sharing environments also the time consumed by other applications
- CPU time
  - Time spent by the CPU to execute the application
  - Does not include time the program was context-switched out
    - Problem: Does not include inherent waiting time (e.g., I/O)
    - Problem: Portability? What is user, what is system time?
- Problem: Execution time is non-deterministic
  - Use mean or minimum of several runs

Inclusive vs. Exclusive values

- Inclusive
  - Information of all sub-elements aggregated into single value
- Exclusive
  - Information cannot be subdivided further



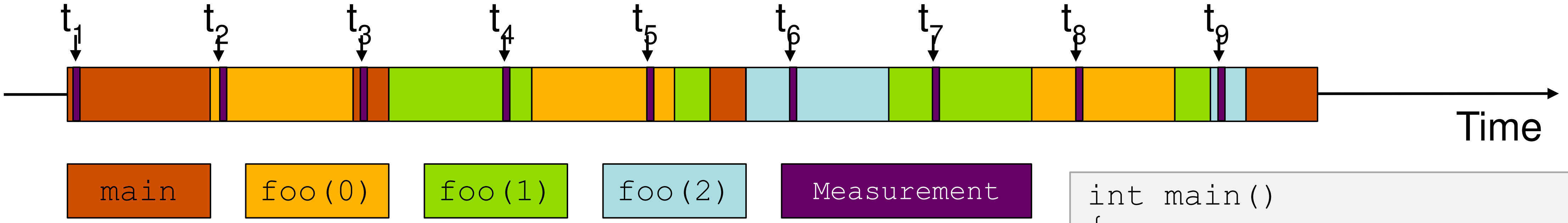
Classification of measurement techniques

- **How are performance measurements triggered?**
  - **Sampling**
  - **Code instrumentation**

- How is performance data recorded?
  - Profiling / Runtime summarization
  - Tracing

- How is performance data analyzed?
  - Online
  - Post mortem

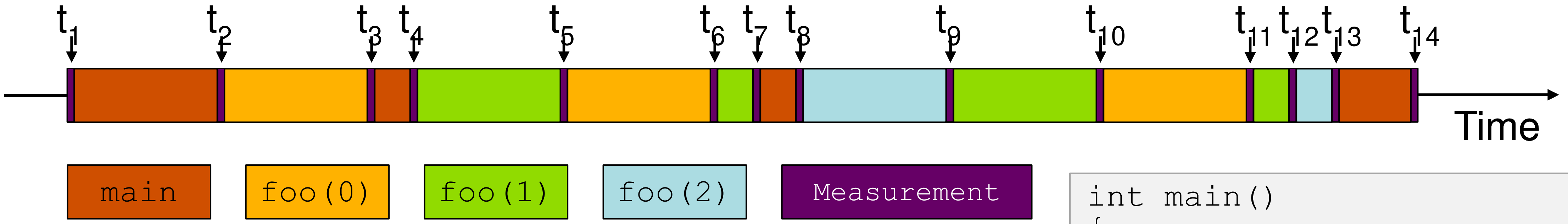
Sampling



- Running program is periodically interrupted to take measurement
  - Timer interrupt, OS signal, or HWC overflow
  - Service routine examines return-address stack
  - Addresses are mapped to routines using symbol table information
- Statistical inference of program behavior
  - Not very detailed information on highly volatile metrics
  - Requires long-running applications
- Works with unmodified executables

```
int main()
{
  int i;

  for (i=0; i < 3; i++)
    foo(i);

  return 0;
}

void foo(int i)
{

  if (i > 0)
    foo(i - 1);

}
```

Instrumentation



- Measurement code is inserted such that every event of interest is captured directly
  - Can be done in various ways
- Advantage:
  - Much more detailed information
- Disadvantage:
  - Processing of source-code / executable necessary
  - Large relative overheads for small functions

```
int main()
{
   int i;
   Enter("main");
   for (i=0; i < 3; i++)
      foo(i);
   Leave("main");
   return 0;
}

void foo(int i)
{
   Enter("foo");
   if (i > 0)
      foo(i - 1);
   Leave("foo");
}
```

Instrumentation techniques

- Static instrumentation
  - Program is instrumented prior to execution
- Dynamic instrumentation
  - Program is instrumented at runtime

- Code is inserted
  - Manually
  - Automatically
    - By a preprocessor / source-to-source translation tool
    - By a compiler
    - By linking against a pre-instrumented library / runtime system
    - By binary-rewrite / dynamic instrumentation tool

Critical issues

- Accuracy
  - Intrusion overhead
    - Measurement itself needs time and thus lowers performance
  - Perturbation
    - Measurement alters program behaviour
    - E.g., memory access pattern
  - Accuracy of timers & counters
- Granularity
  - How many measurements?
  - How much information / processing during each measurement?

☞ *Tradeoff: Accuracy vs. Expressiveness of data*

Classification of measurement techniques

- How are performance measurements triggered?
  - Sampling
  - Code instrumentation
- **How is performance data recorded?**
  - **Profiling / Runtime summarization**
  - **Tracing**
- How is performance data analyzed?
  - Online
  - Post mortem

Profiling / Runtime summarization

- Recording of aggregated information
  - Total, maximum, minimum, …
- For measurements
  - Time
  - Counts
    - Function calls
    - Bytes transferred
    - Hardware counters
- Over program and system entities
  - Functions, call sites, basic blocks, loops, …
  - Processes, threads

☞ *Profile = summarization of events over execution interval*

Types of profiles

- Flat profile
  - Shows distribution of metrics per routine / instrumented region
  - Calling context is not taken into account
- Call-path profile
  - Shows distribution of metrics per executed call path
  - Sometimes only distinguished by partial calling context (e.g., two levels)
- Special-purpose profiles
  - Focus on specific aspects, e.g., MPI calls or OpenMP constructs
  - Comparing processes/threads

Tracing

- Recording detailed information about significant points (events) during execution of the program
  - Enter / leave of a region (function, loop, …)
  - Send / receive a message, …
- Save information in event record
  - Timestamp, location, event type
  - Plus event-specific information (e.g., communicator, sender / receiver, …)
- Abstract execution model on level of defined events

☞ *Event trace = Chronologically ordered sequence of event records*

Event tracing

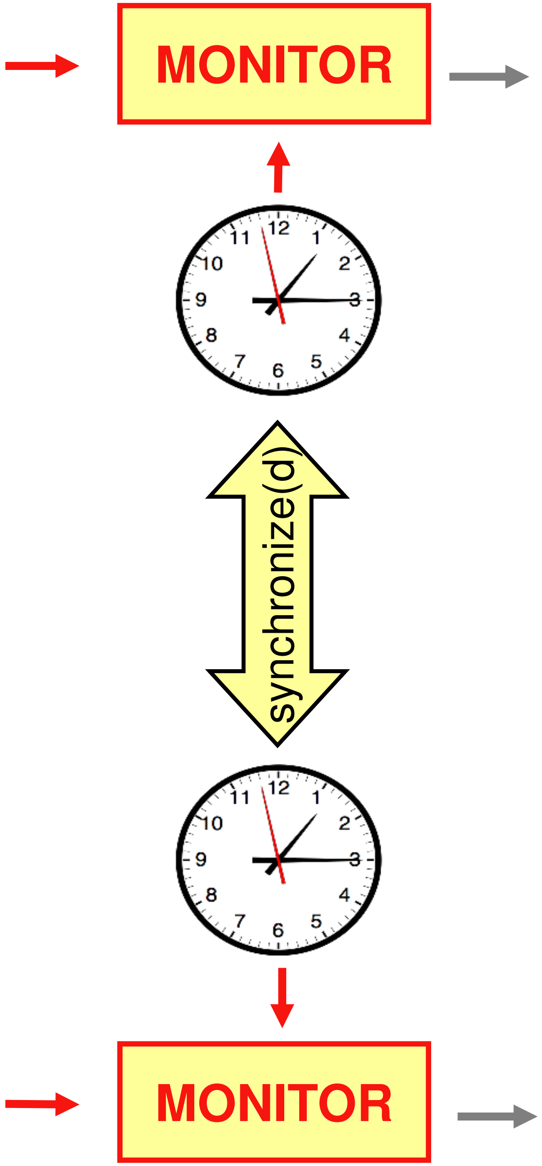
Process A

```
void foo() {
  trc_enter("foo");
  ...
  trc_send(B);
  send(B, tag, buf);
  ...
  trc_exit("foo");
}
```

instrument

Process B

```
void bar() {
  trc_enter("bar");
  ...
  recv(A, tag, buf);
  trc_recv(A);
  ...
  trc_exit("bar");
}
```



Local trace A

...	
58	ENTER foo
62	SEND to B
64	EXIT foo
...	

Local trace B

...	
60	ENTER bar
68	RECV from A
69	EXIT bar
...	

(Virtual merge)

Global trace view

...		
58	A	ENTER foo
60	B	ENTER bar
62	A	SEND to B
64	A	EXIT foo
68	B	RECV from A
69	B	EXIT bar
...		

Tracing Pros & Cons

- Tracing advantages
  - Event traces preserve the **temporal** and **spatial** relationships among individual events (☞ context)
  - Allows reconstruction of **dynamic** application behaviour on any required level of abstraction
  - Most general measurement technique
    - Profile data can be reconstructed from event traces
- Disadvantages
  - Traces can very quickly become extremely large
  - Writing events to file at runtime may causes perturbation

Classification of measurement techniques

- How are performance measurements triggered?
  - Sampling
  - Code instrumentation

- How is performance data recorded?
  - Profiling / Runtime summarization
  - Tracing

- **How is performance data analyzed?**
  - **Online**
  - **Post mortem**

Online analysis

- Performance data is processed during measurement run
  - Process-local profile aggregation
  - Requires formalized knowledge about performance bottlenecks
  - More sophisticated inter-process analysis using
    - "Piggyback" messages
    - Hierarchical network of analysis agents
- Online analysis often involves application steering to interrupt and re-configure the measurement

Post-mortem analysis

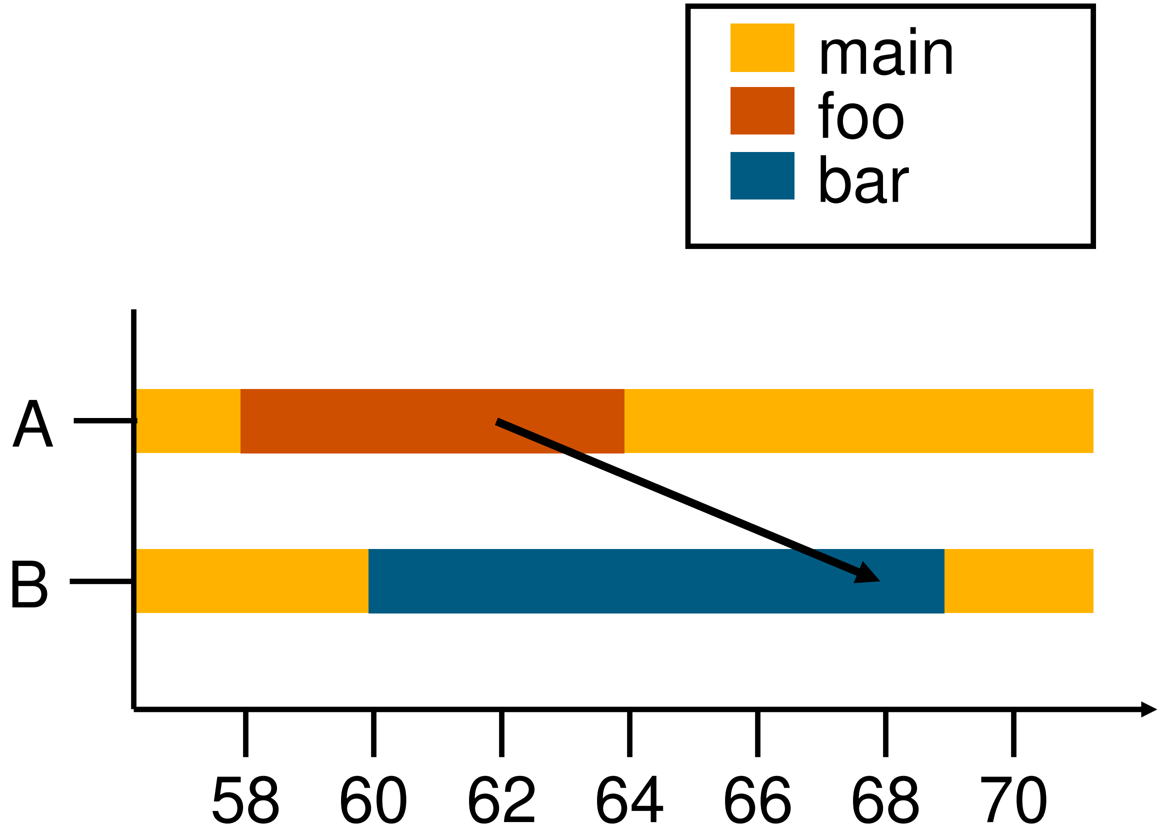
- Performance data is stored at end of measurement run
- Data analysis is performed afterwards
  - Automatic search for bottlenecks
  - Visual trace analysis
  - Calculation of statistics

Example: Time-line visualization

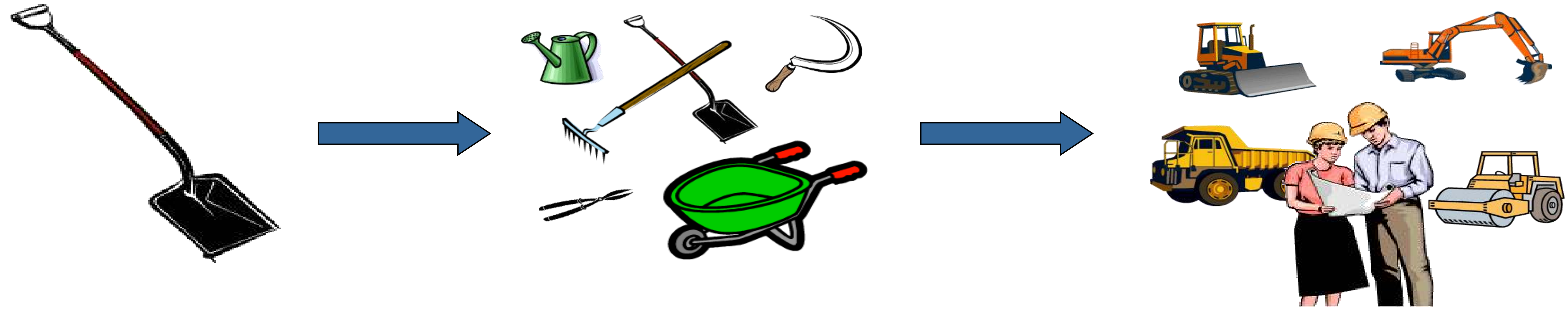
Global trace view

...		
58	A	ENTER foo
60	B	ENTER bar
62	A	SEND to B
64	A	EXIT foo
68	B	RECV from A
69	B	EXIT bar
...		

Post-Mortem
Analysis



No single solution is sufficient!



A combination of different methods, tools and techniques is typically needed!

- Analysis
  - Statistics, visualization, automatic analysis, data mining, ...
- Measurement
  - Sampling / instrumentation, profiling / tracing, ...
- Instrumentation
  - Source code / binary, manual / automatic, ...

Typical performance analysis procedure

- Do I have a performance problem at all?
  - Time / speedup / scalability measurements
- What is the key bottleneck (computation / communication)?
  - MPI / OpenMP / flat profiling
- Where is the key bottleneck?
  - Call-path profiling, detailed basic block profiling
- Why is it there?
  - Hardware counter analysis, trace selected parts to keep trace size manageable
- Does the code have scalability problems?
  - Load imbalance analysis, compare profiles at various sizes function-by-function

Hands-on:
NPB-MZ-MPI / BT

VI-HPS Team

Tutorial exercise objectives

- Familiarize with usage of VI-HPS tools
  - Complementary tools' capabilities & interoperability
- Prepare to apply tools productively to *your* application(s)
- Exercise is based on a small portable benchmark code
  - Unlikely to have significant optimization opportunities

- Optional (recommended) exercise extensions
  - Analyze performance of alternative configurations
  - Investigate effectiveness of system-specific compiler/MPI optimizations and/or placement/binding/affinity capabilities
  - Investigate scalability and analyze scalability limiters
  - Compare performance on different HPC platforms
  - …

Access to Stampede2

```
# Connect to a Stampede2 login node
% ssh -Y userid@stampede2.tacc.utexas.edu
```

- Logging in to Stampede2

```
$HOME
$WORK
$SCRATCH

/home1/03529/tg828282/Tutorial
(shortcut: ~tg828282/Tutorial)
```

Tutorial materials

- File systems & directories
  - Use $SCRATCH for the tutorial
    - Fast Lustre file system, ~30 PB
    - No backup
    - Files may be automatically purged 10 days after last modification

- More extensive documentation:
  - https://portal.tacc.utexas.edu/user-guides/stampede2

Compiling & job submission

- Development environment: Intel compiler with Intel MPI
  - Use Intel's MPI compiler wrappers
    - `mpiicc`
    - `mpiicpc`
    - `mpiifort`

- Stampede2 uses the SLURM batch system
  - Jobs submitted from tutorial accounts with provided job scripts will automatically be run in a reservation

```
% sbatch jobscript.sbatch
% squeue -u $USER
% scancel <jobid>
```

← Submit job
← View job queue
← Cancel job

Local installation

- VI-HPS tools not yet installed system-wide
  - Source provided shell code snippet to add local tool installations to $PATH
  - Required for each shell session

```
% source ~tg828282/Tutorial/vihps.sh
```

- Copy tutorial sources to your working directory, ideally on a parallel file system (recommended: $SCRATCH)

```
% cd $SCRATCH
% tar zxvf ~tg828282/Tutorial/NPB3.3-MZ-MPI.tar.gz
% cd NPB3.3-MZ-MPI
```

NPB-MZ-MPI suite

- The NAS Parallel Benchmark suite (MPI+OpenMP version)
  - Available from http://www.nas.nasa.gov/Software/NPB
  - 3 benchmarks in Fortran77
  - Configurable for various sizes & classes
- Move into the NPB3.3-MZ-MPI root directory

```
% ls
bin/     common/   jobscript/   Makefile   README.install    SP-MZ/
BT-MZ/   config/   LU-MZ/       README     README.tutorial   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 is ready to “make” one or more of the benchmarks and install them into a (tool-specific) “bin” subdirectory

Building an NPB-MZ-MPI benchmark

```
% make
 ===========================================
 =      NAS PARALLEL BENCHMARKS 3.3        =
 =      MPI+OpenMP Multi-Zone Versions     =
 =      F77                                =
 ===========================================

 To make a NAS multi-zone benchmark type

        make <benchmark-name> CLASS=<class> NPROCS=<nprocs>

 where <benchmark-name> is "bt-mz", "lu-mz", or "sp-mz"
       <class>          is "S", "W", "A" through "F"
       <nprocs>         is number of processes

 [...]

****************************************************************
* Custom build configuration is specified in config/make.def  *
* Suggested tutorial exercise configuration for Stampede2:    *
*       make bt-mz CLASS=C NPROCS=32                          *
****************************************************************
```

- Type "make" for instructions

Building an NPB-MZ-MPI benchmark

```
% make bt-mz CLASS=C NPROCS=32
make[1]: Entering directory `BT-MZ'
make[2]: Entering directory `sys'
icc  -o setparams setparams.c -lm
make[2]: Leaving directory `sys'
../sys/setparams bt-mz 32 C
make[2]: Entering directory `../BT-MZ'
mpiifort -c  -g -O3 -qopenmp       bt.f
                                      [...]
mpiifort -c  -g -O3 -qopenmp       mpi_setup.f
cd ../common;  mpiifort -c  -g -O3 -qopenmp        print_results.f
cd ../common;  mpiifort -c  -g -O3 -qopenmp        timers.f
mpiifort -g -O3 -qopenmp   -o ../bin/bt-mz_C.32 bt.o
 initialize.o exact_solution.o exact_rhs.o set_constants.o adi.o
 rhs.o zone_setup.o x_solve.o y_solve.o  exch_qbc.o solve_subs.o
 z_solve.o add.o error.o verify.o mpi_setup.o ../common/print_results.o
 ../common/timers.o
make[2]: Leaving directory `BT-MZ'
Built executable ../bin/bt-mz_C.32
make[1]: Leaving directory `BT-MZ'
```

- Specify the benchmark configuration
  - benchmark name: **bt-mz**, lu-mz, sp-mz
  - the number of MPI processes: NPROCS=**32**
  - the benchmark class (S, W, A, B, C, D, E): CLASS=**C**

Shortcut: % **make suite**

NPB-MZ-MPI / BT (Block Tridiagonal Solver)

- What does it do?
  - Solves a discretized version of the unsteady, compressible Navier-Stokes equations in three spatial dimensions
  - Performs 200 time-steps on a regular 3-dimensional grid
- Implemented in 20 or so Fortran77 source modules

- Uses MPI & OpenMP in combination
  - Proposed hands-on setup on Stampede2:
    - 2 compute nodes with 1 Intel Xeon Phi 7250 CPU (Knights Landing, KNL) each
    - 32 MPI processes with 4 OpenMP threads each
  - bt-mz_C.32 should run in less than 30 seconds

NPB-MZ-MPI / BT reference execution

```
% cd bin
% cp ../jobscript/stampede2/reference.sbatch .
% less reference.sbatch
% sbatch reference.sbatch
% less mzmpibt.o<job_id>
 NAS Parallel Benchmarks (NPB3.3-MZ-MPI) - BT-MZ MPI+OpenMP Benchmark
 Number of zones:  16 x  16
 Iterations:  200    dt:   0.000100
 Number of active processes:    32
 Total number of threads:      128   (  4.0 threads/process)

 Time step    1
 Time step   20
  [...]
 Time step  180
 Time step  200
 Verification Successful

 BT-MZ Benchmark Completed.
 Time in seconds = 22.34
```

- Copy jobscript and launch as a hybrid MPI+OpenMP application

Hint: save the benchmark output (or note the run time) to be able to refer to it later

Tutorial exercise steps

- Edit config/make.def to adjust build configuration
  - Modify specification of compiler/linker: MPIF77
  - See next slide for details
- Make clean and build new tool-specific executable

```
% make clean
% make bt-mz CLASS=C NPROCS=32
Built executable ../bin.$(TOOL)/bt-mz_C.32
```

- Change to the directory containing the new executable before running it with the desired tool configuration

```
% cd bin.$(TOOL)
% cp ../jobscript/stampede2/$(TOOL).sbatch .
% sbatch $(TOOL).sbatch
```

NPB-MZ-MPI / BT: config/make.def

```
#                  SITE- AND/OR PLATFORM-SPECIFIC DEFINITIONS.
#
#---------------------------------------------------------------------------

#---------------------------------------------------------------------------
# Configured for generic MPI with INTEL compiler
#---------------------------------------------------------------------------
#OPENMP  = -fopenmp      # GCC compiler
OPENMP = -qopenmp        # Intel compiler

...
#---------------------------------------------------------------------------
# The Fortran compiler used for MPI programs
#---------------------------------------------------------------------------
MPIF77 = mpiifort

# Alternative variant to perform instrumentation
#MPIF77 = scorep --user mpiifort

# PREP is a generic preposition macro for instrumentation preparation
#MPIF77 = $(PREP) mpiifort
...
```

Default (no instrumentation)

Hint: uncomment a compiler wrapper to do instrumentation

Score-P – A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir

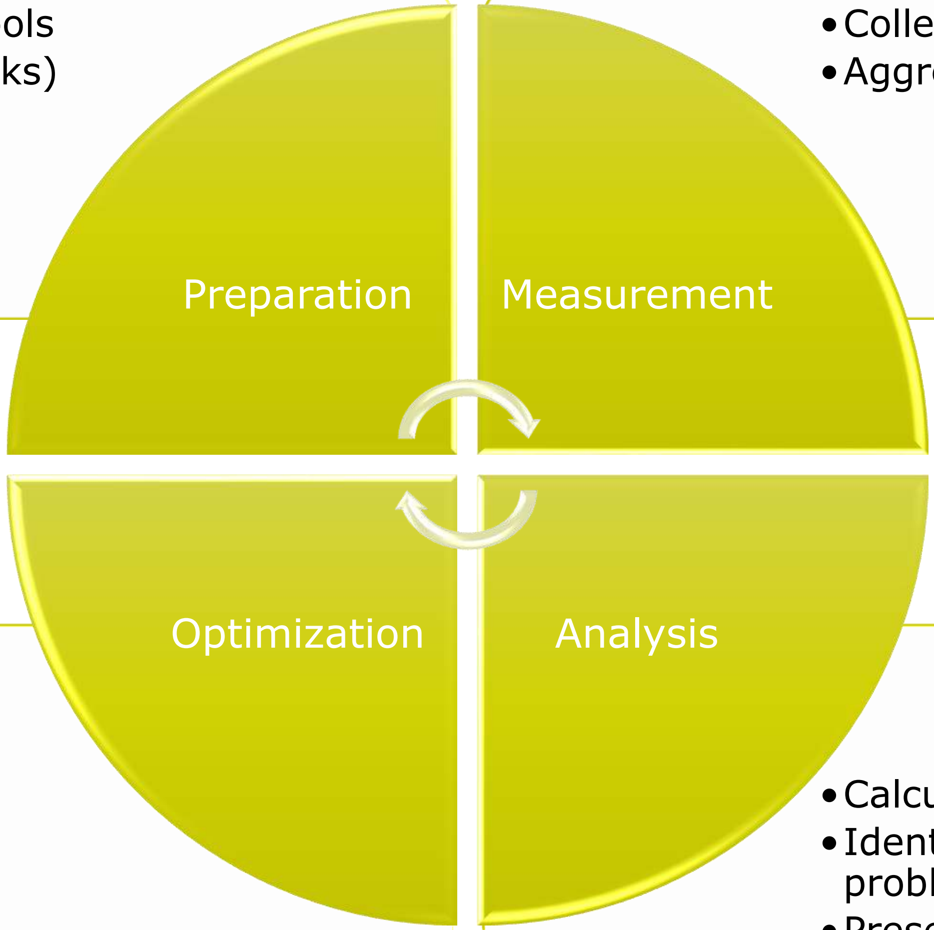
VI-HPS Team

Score-P
Scalable performance measurement infrastructure for parallel codes

Performance engineering workflow

- Prepare application with symbols
- Insert extra code (probes/hooks)

- Collection of performance data
- Aggregation of performance data



- Modifications intended to eliminate/reduce performance problem

- Calculation of metrics
- Identification of performance problems
- Presentation of results

Fragmentation of tools landscape

- Several performance tools co-exist
  - Separate measurement systems and output formats
- Complementary features and overlapping functionality
- Redundant effort for development and maintenance
  - Limited or expensive interoperability
- Complications for user experience, support, training

Vampir	Scalasca	TAU	Periscope
VampirTrace OTF	EPILOG / CUBE	TAU native formats	Online measurement

Score-P project idea

- Start a community effort for a common infrastructure
  - Score-P instrumentation and measurement system
  - Common data formats OTF2 and CUBE4
- Developer perspective:
  - Save manpower by sharing development resources
  - Invest in new analysis functionality and scalability
  - Save efforts for maintenance, testing, porting, support, training
- User perspective:
  - Single learning curve
  - Single installation, fewer version updates
  - Interoperability and data exchange
- Project funded by BMBF
- Close collaboration PRIMA project funded by DOE

GEFÖRDERT VOM
Bundesministerium für Bildung und Forschung

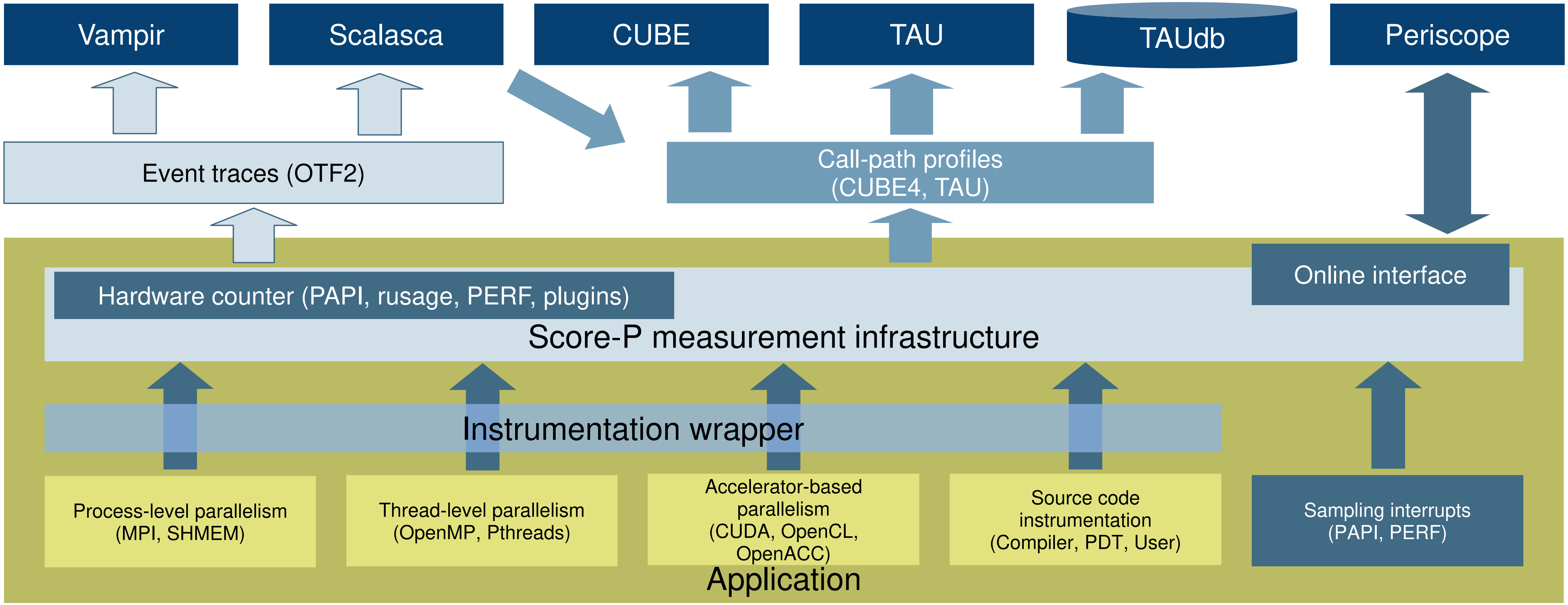
Partners

- Forschungszentrum Jülich, Germany
- Gesellschaft für numerische Simulation mbH Braunschweig, Germany
- RWTH Aachen, Germany
- Technische Universität Darmstadt, Germany
- Technische Universität Dresden, Germany
- Technische Universität München, Germany
- University of Oregon, Eugene, USA

Design goals

- Functional requirements
  - Generation of call-path profiles and event traces
  - Using direct instrumentation and sampling
  - Flexible measurement without re-compilation
  - Recording time, visits, communication data, hardware counters
  - Access and reconfiguration also at runtime
  - Support for MPI, SHMEM, OpenMP, Pthreads, CUDA, OpenCL, OpenACC and their valid combinations
  - Highly scalable I/O
- Non-functional requirements
  - Portability: all major HPC platforms
  - Scalability: petascale
  - Low measurement overhead
  - Robustness
  - Open Source: 3-clause BSD license

Score-P overview



Future features and management

- Scalability to maximum available CPU core count
- Support for binary instrumentation
- Support for new programming models, e.g., PGAS
- Support for new architectures

- Ensure a single official release version at all times which will always work with the tools
- Allow experimental versions for new features or research

- Commitment to joint long-term cooperation
  - Development based on meritocratic governance model
  - Open for contributions and new partners

Hands-on:
NPB-MZ-MPI / BT

Score-P

Performance analysis steps

- 0.0 Reference preparation for validation

- 1.0 Program instrumentation
- 1.1 Summary measurement collection
- 1.2 Summary analysis report examination

- 2.0 Summary experiment scoring
- 2.1 Summary measurement collection with filtering
- 2.2 Filtered summary analysis report examination

- 3.0 Event trace collection
- 3.1 Event trace examination & analysis

Recap: Local installation

- VI-HPS tools not yet installed system-wide
  - Source provided shell code snippet to add local tool installations to $PATH
  - Required for each shell session

```
% source ~tg828282/Tutorial/vihps.sh
```

- Copy tutorial sources to your working directory, ideally on a parallel file system (recommended: $SCRATCH)

```
% cd $SCRATCH
% tar zxvf ~tg828282/Tutorial/NPB3.3-MZ-MPI.tar.gz
% cd NPB3.3-MZ-MPI
```

NPB-MZ-MPI / BT instrumentation

```
#---------------------------------------------------------------------
# The Fortran compiler used for MPI programs
#---------------------------------------------------------------------
#MPIF77 = mpiifort

# Alternative variants to perform instrumentation
...
MPIF77 = scorep --user mpiifort

# This links MPI Fortran programs; usually the same as ${MPIF77}
FLINK   = $(MPIF77)
...
```

- Edit config/make.def to adjust build configuration
  - Modify specification of compiler/linker: MPIF77

Uncomment the Score-P compiler wrapper specification

NPB-MZ-MPI / BT instrumented build

```
% make clean

% make bt-mz CLASS=C NPROCS=32
cd BT-MZ; make CLASS=C NPROCS=32 VERSION=
make: Entering directory 'BT-MZ'
cd ../sys; icc  -o setparams setparams.c -lm
../sys/setparams bt-mz 32 C
scorep --user mpiifort -c  -g -O3 -qopenmp bt.f
 [...]
cd ../common;  scorep --user mpiifort -c  -g -O3 -qopenmp timers.f
 [...]
scorep --user mpiifort -g -O3 -qopenmp -o ../bin.scorep/bt-mz_C.32 \
bt.o initialize.o exact_solution.o exact_rhs.o set_constants.o \
adi.o rhs.o zone_setup.o x_solve.o y_solve.o exch_qbc.o \
solve_subs.o z_solve.o add.o error.o verify.o mpi_setup.o \
../common/print_results.o ../common/timers.o
Built executable ../bin.scorep/bt-mz_C.32
make: Leaving directory 'BT-MZ'
```

- Return to root directory and clean-up
- Re-build executable using Score-P compiler wrapper

Measurement configuration: scorep-info

```
% scorep-info config-vars --full
SCOREP_ENABLE_PROFILING
  Description: Enable profiling
 [...]
SCOREP_ENABLE_TRACING
  Description: Enable tracing
 [...]
SCOREP_TOTAL_MEMORY
  Description: Total memory in bytes for the measurement system
 [...]
SCOREP_EXPERIMENT_DIRECTORY
  Description: Name of the experiment directory
 [...]
SCOREP_FILTERING_FILE
  Description: A file name which contain the filter rules
 [...]
SCOREP_METRIC_PAPI
  Description: PAPI metric names to measure
 [...]
SCOREP_METRIC_RUSAGE
  Description: Resource usage metric names to measure
 [... More configuration variables ...]
```

- Score-P measurements are configured via environmental variables

Summary measurement collection

```
% cd bin.scorep
% cp ../jobscript/stampede2/scorep.sbatch .
% vim scorep.sbatch

# Score-P measurement configuration
export SCOREP_EXPERIMENT_DIRECTORY=scorep_bt-mz_sum
#export SCOREP_FILTERING_FILE=../config/scorep.filt
#export SCOREP_TOTAL_MEMORY=50M
#export SCOREP_METRIC_PAPI=PAPI_TOT_INS,PAPI_TOT_CYC
#export SCOREP_ENABLE_TRACING=true

# Run the application
ibrun ./bt-mz_${CLASS}.${PROCS}

% sbatch ./scorep.sbatch
```

- Change to the directory containing the new executable before running it with the desired configuration
- Check settings

Leave these lines commented out for the moment

- Submit job

Summary measurement collection

```
% less mzmpibt.o<job_id>

 NAS Parallel Benchmarks (NPB3.3-MZ-MPI) - BT-MZ MPI+OpenMP \
>Benchmark

 Number of zones:  16 x  16
 Iterations: 200    dt:   0.000100
 Number of active processes:    32

 Use the default load factors with threads
 Total number of threads:    128  (  4.0 threads/process)

 Calculated speedup =    125.90

 Time step    1

 [... More application output ...]
```

- Check the output of the application run

BT-MZ summary analysis report examination

```
% ls
bt-mz_C.32  mzmpibt.e<job_id>  mzmpibt.o<job_id>
scorep_bt-mz_sum
% ls scorep_bt-mz_sum
profile.cubex  scorep.cfg



% cube scorep_bt-mz_sum/profile.cubex

       [CUBE GUI showing summary analysis report]
```

- Creates experiment directory including
  - A record of the measurement configuration (scorep.cfg)
  - The analysis report that was collated after measurement (profile.cubex)
- Interactive exploration with Cube

Hint:
Copy ‘profile.cubex’ to Live-DVD environment using ‘scp’ to improve responsiveness of GUI

Further information

- Community instrumentation & measurement infrastructure
  - Instrumentation (various methods)
  - Basic and advanced profile generation
  - Event trace recording
  - Online access to profiling data
- Available under 3-clause BSD open-source license
- Documentation & Sources:
  - http://www.score-p.org
- User guide also part of installation:
  - `<prefix>/share/doc/scorep/{pdf,html}/`
- Support and feedback: support@score-p.org
- Subscribe to news@score-p.org, to be up to date

Analysis report examination with Cube

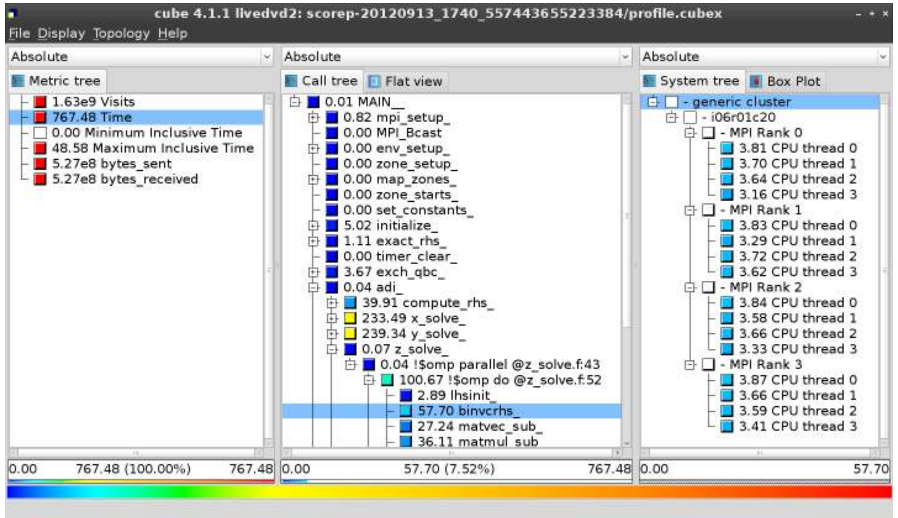
Markus Geimer
Jülich Supercomputing Centre

Cube

- Parallel program analysis report exploration tools
  - Libraries for XML+binary report reading & writing
  - Algebra utilities for report processing
  - GUI for interactive analysis exploration
    - Requires Qt4 ≥4.6 or Qt 5

- Originally developed as part of the Scalasca toolset

- Now available as a separate component
  - Can be installed independently of Score-P, e.g., on laptop or desktop
  - Latest release: Cube v4.4 (May 2018)

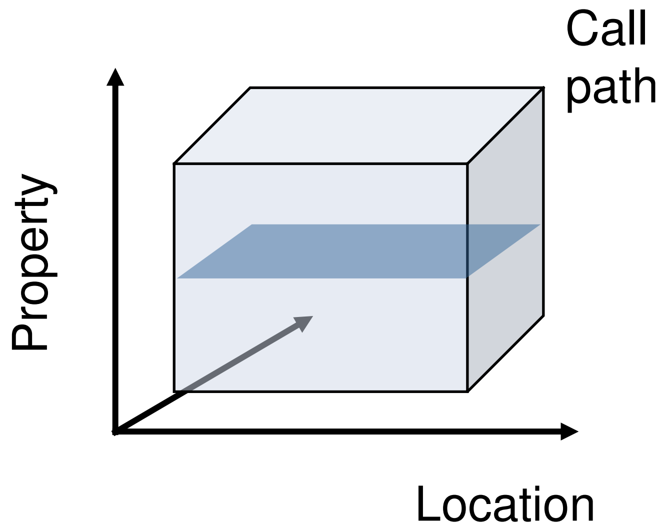


Analysis presentation and exploration

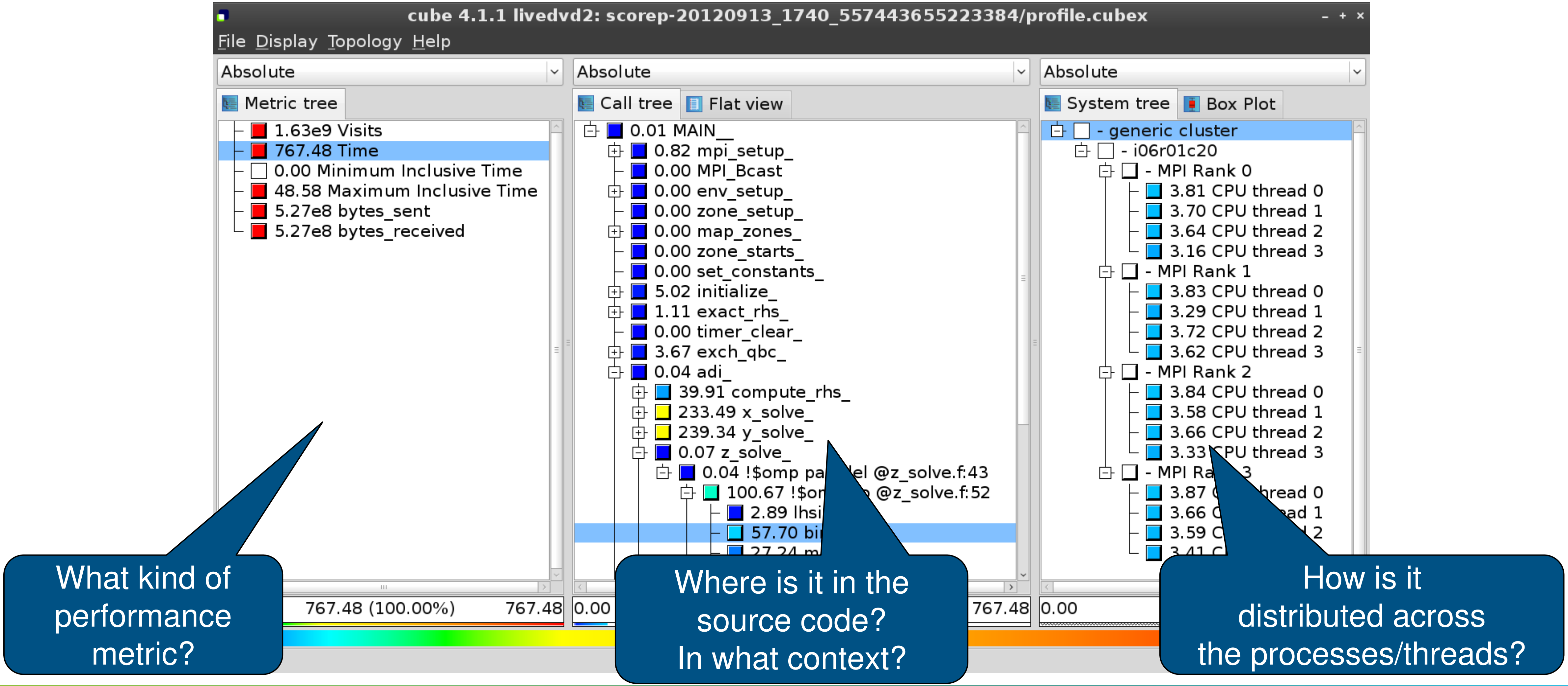
- Representation of values (severity matrix) on three hierarchical axes
  - Performance property (metric)
  - Call path (program location)
  - System location (process/thread)

- Three coupled tree browsers

- Cube displays severities
  - As value: for precise comparison
  - As color: for easy identification of hotspots
  - Inclusive value when closed & exclusive value when expanded
  - Customizable via display modes

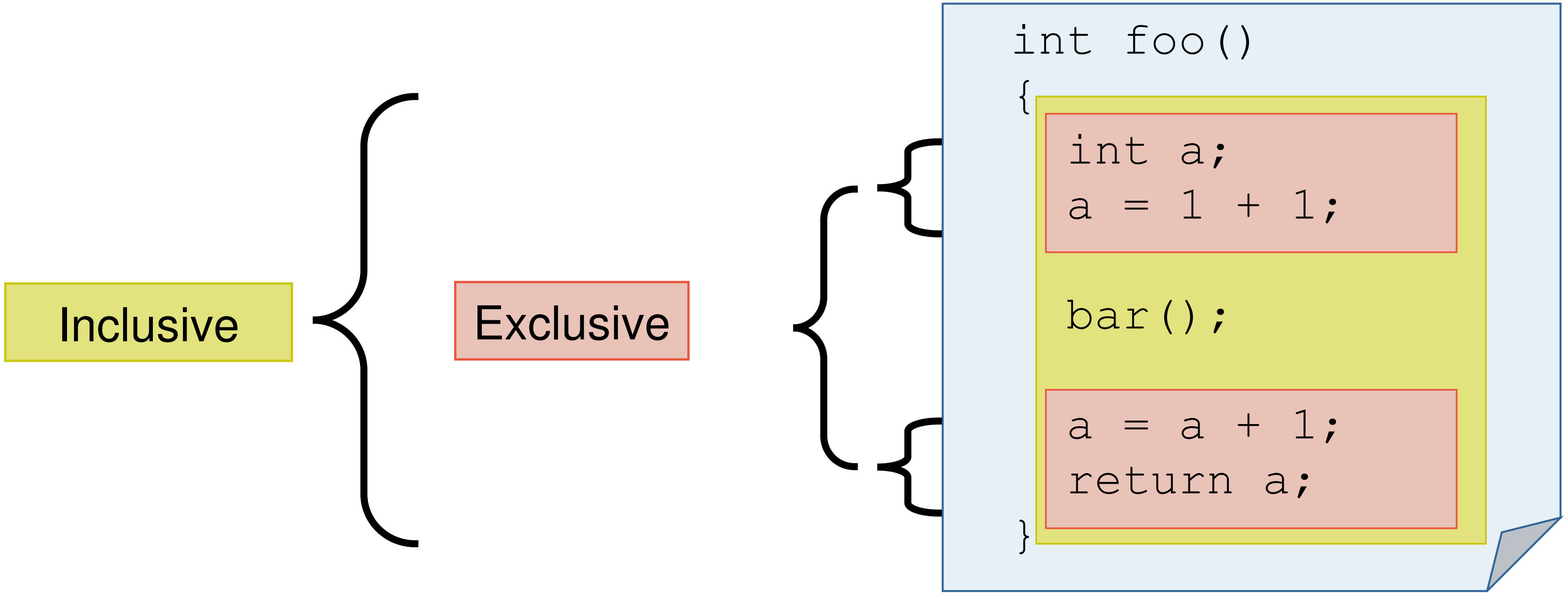


Analysis presentation

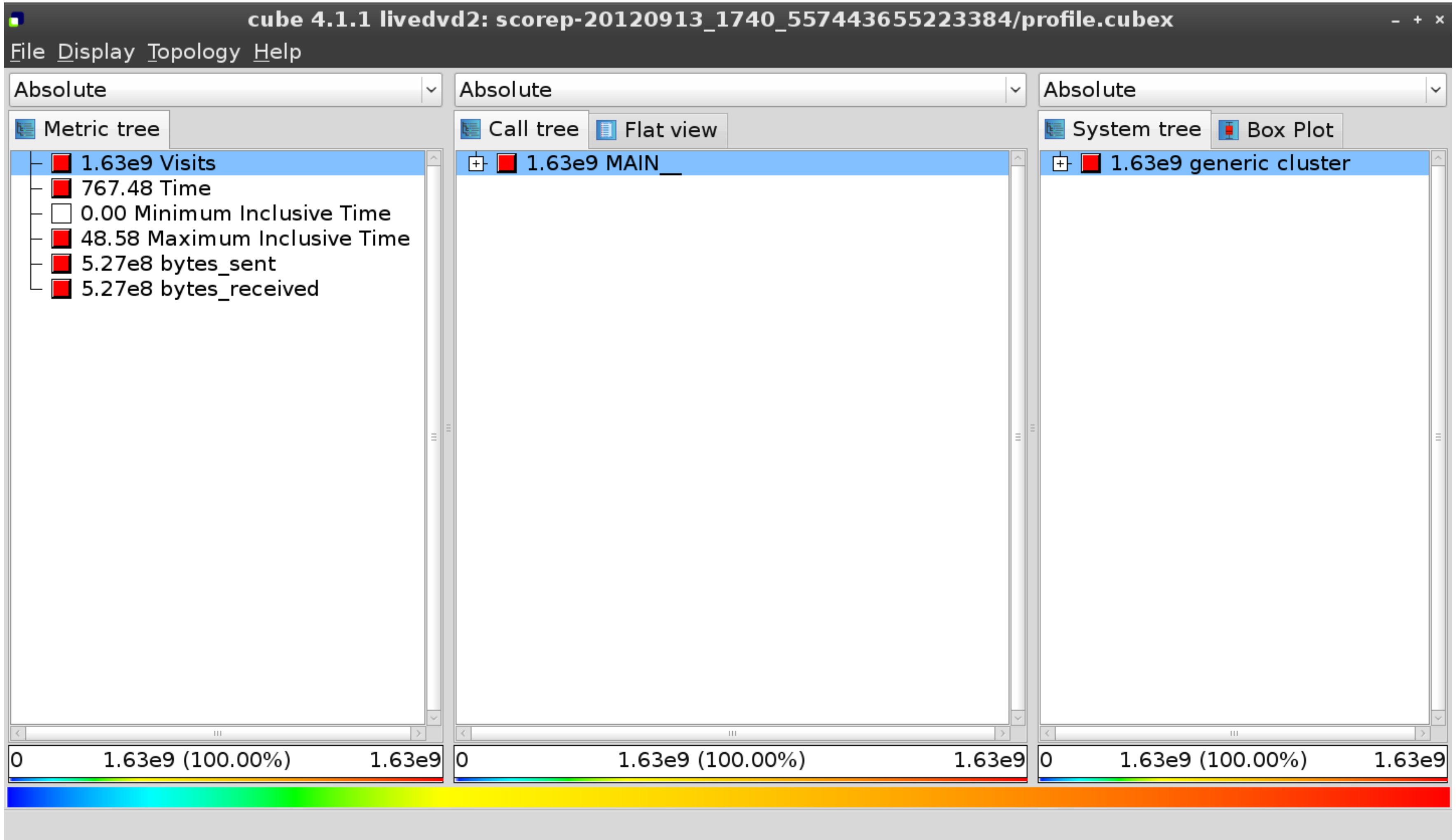


Inclusive vs. exclusive values

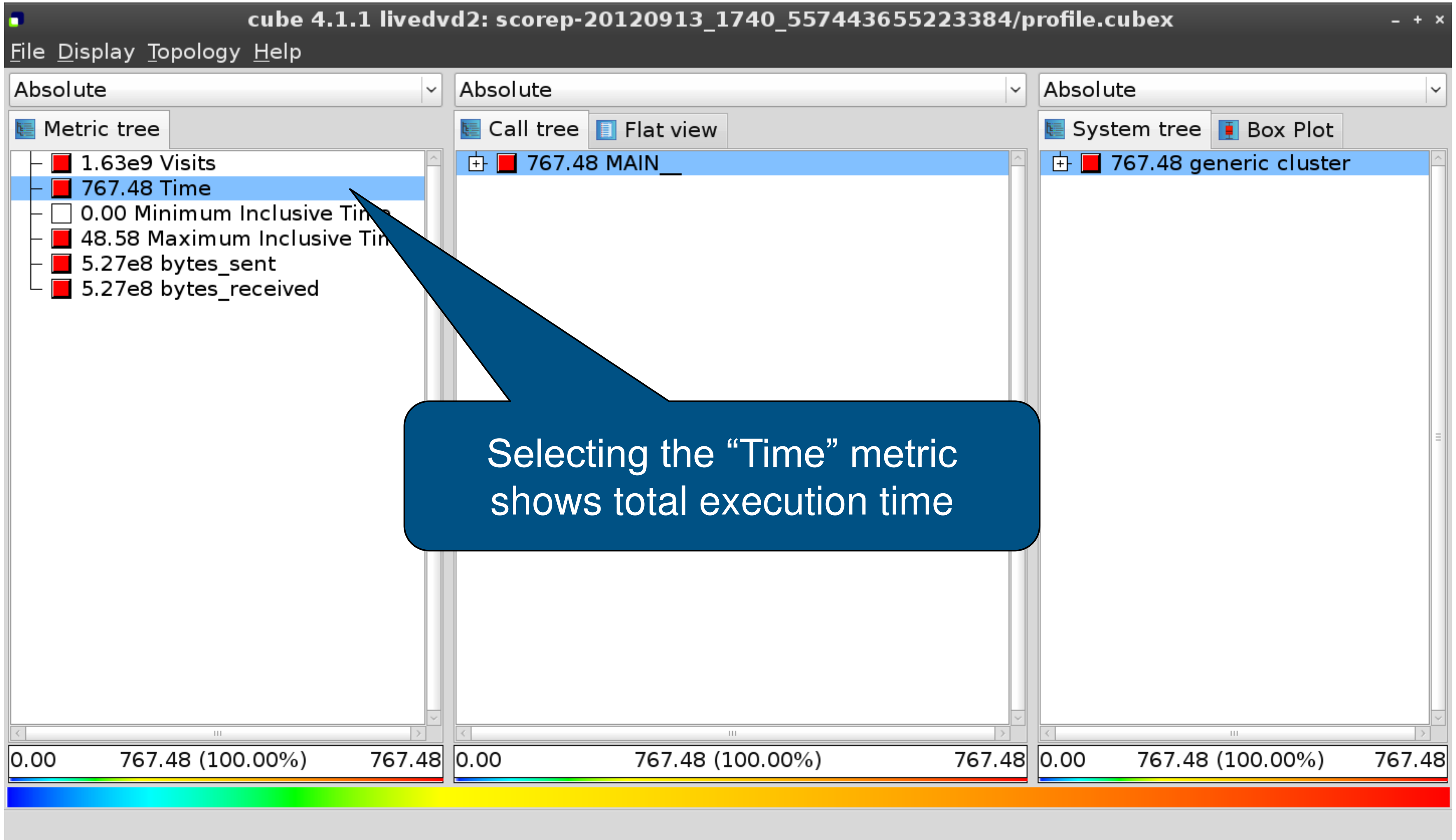
- Inclusive
  - Information of all sub-elements aggregated into single value
- Exclusive
  - Information cannot be subdivided further



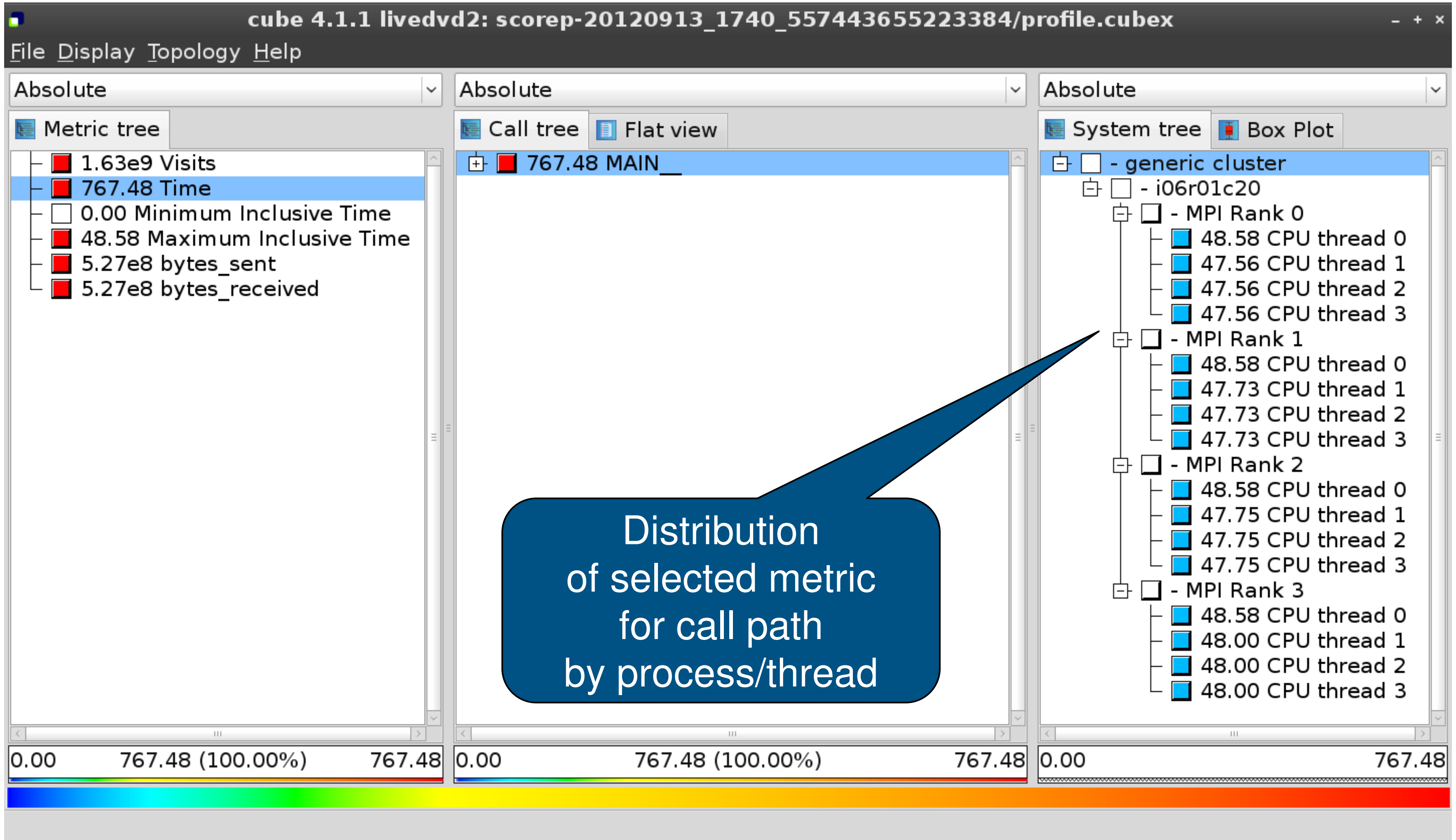
Score-P analysis report exploration (opening view)



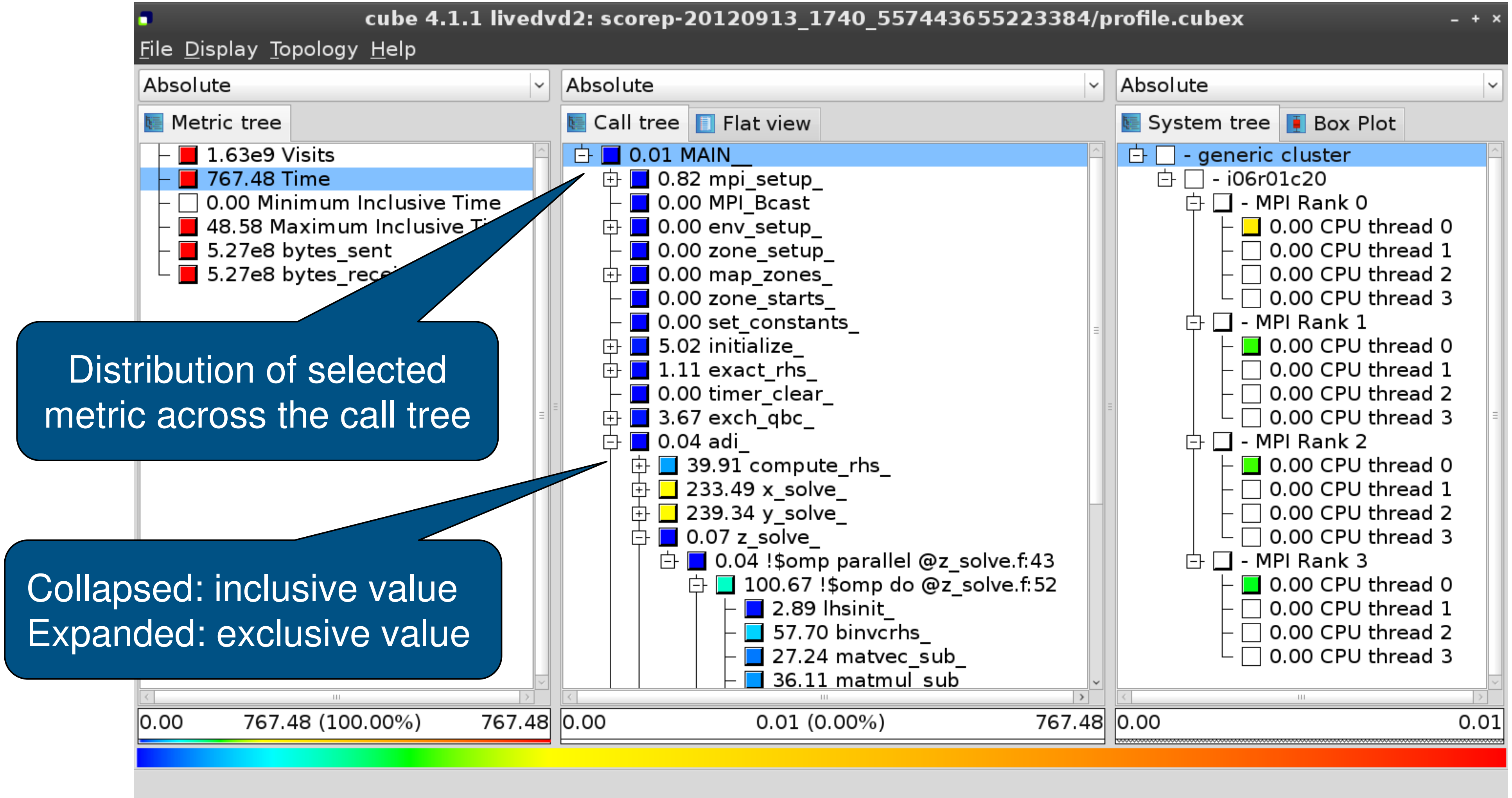
Metric selection



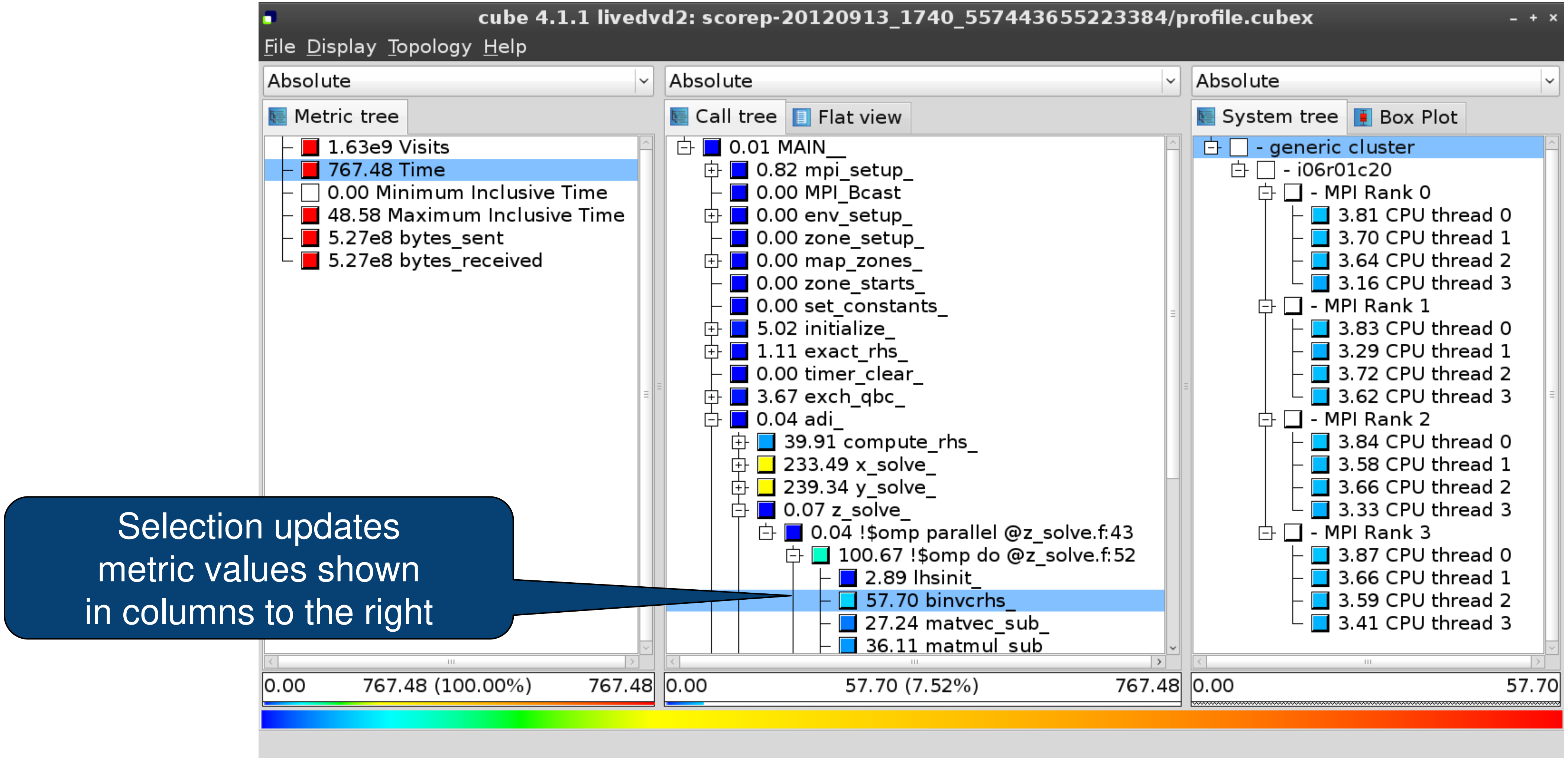
Expanding the system tree



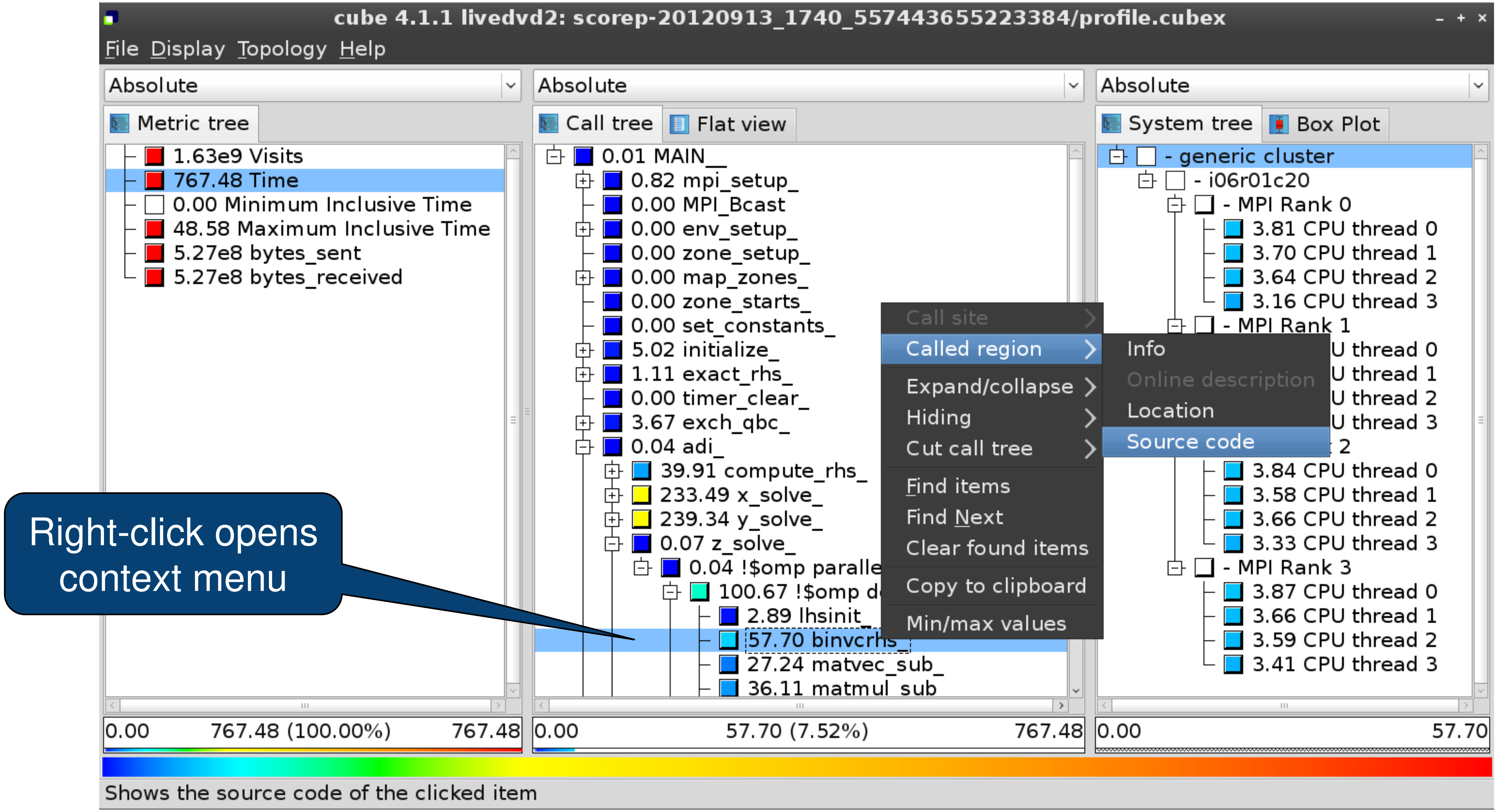
Expanding the call tree



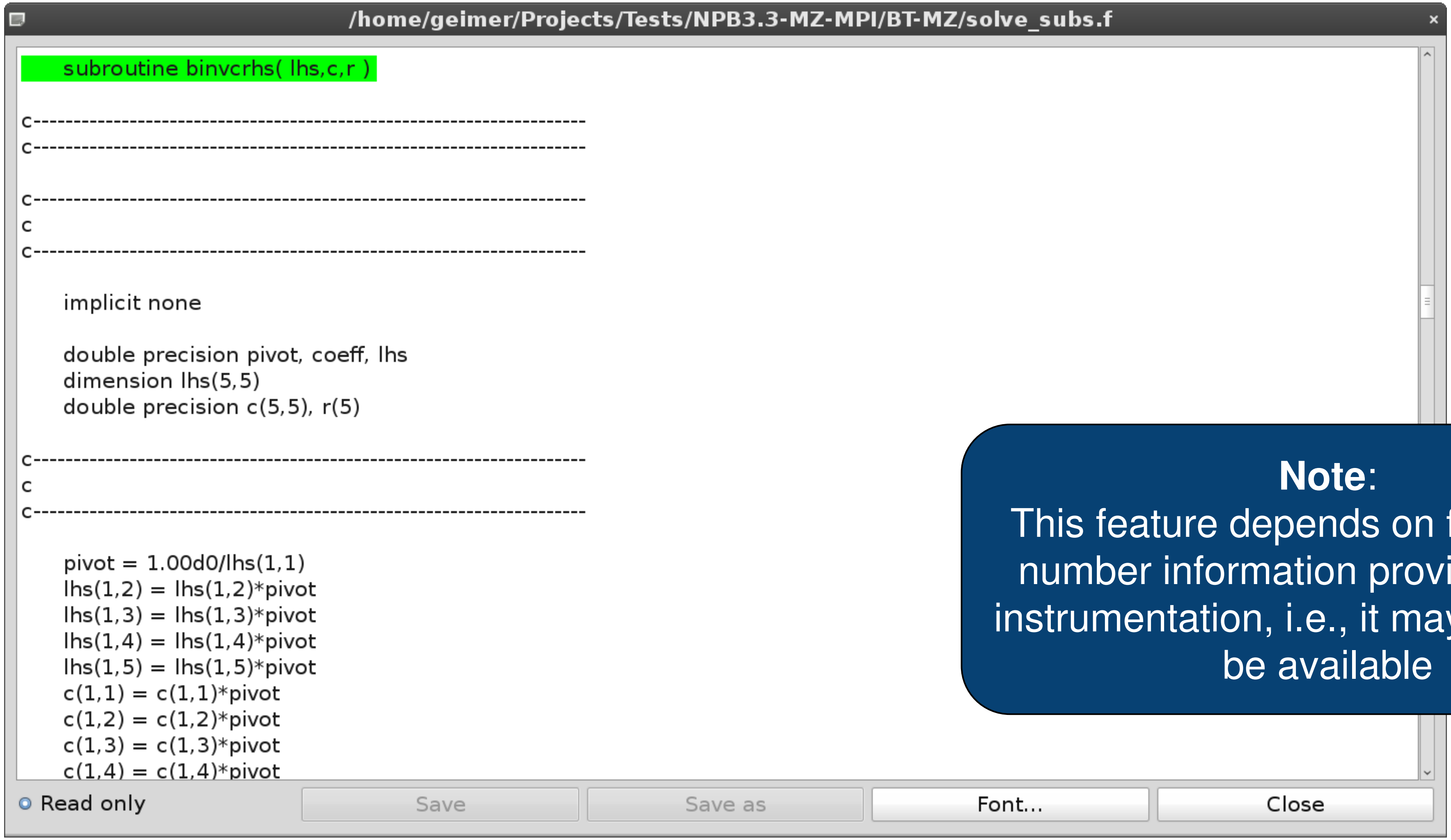
Selecting a call path



Source-code view via context menu

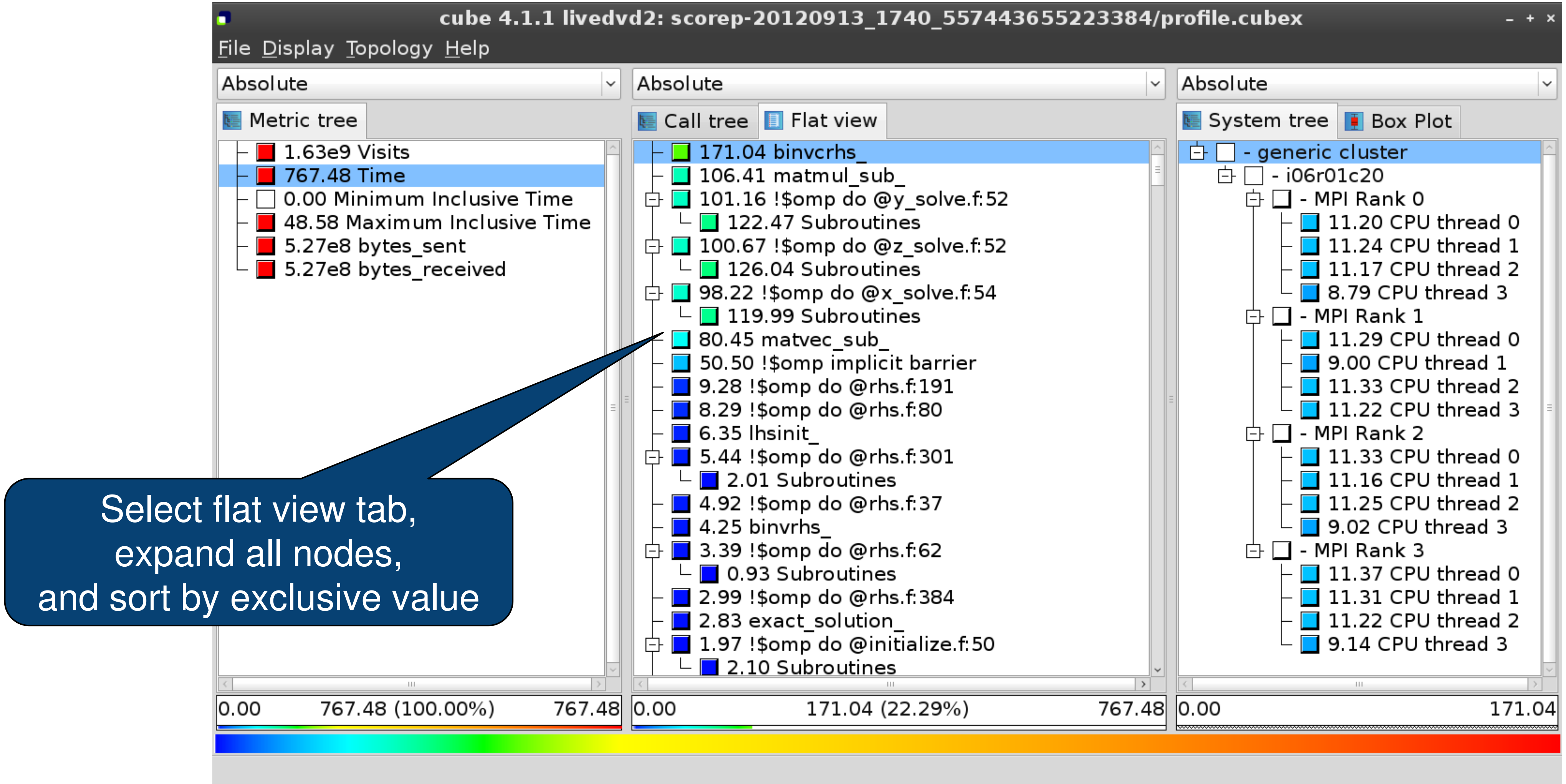


Source-code view

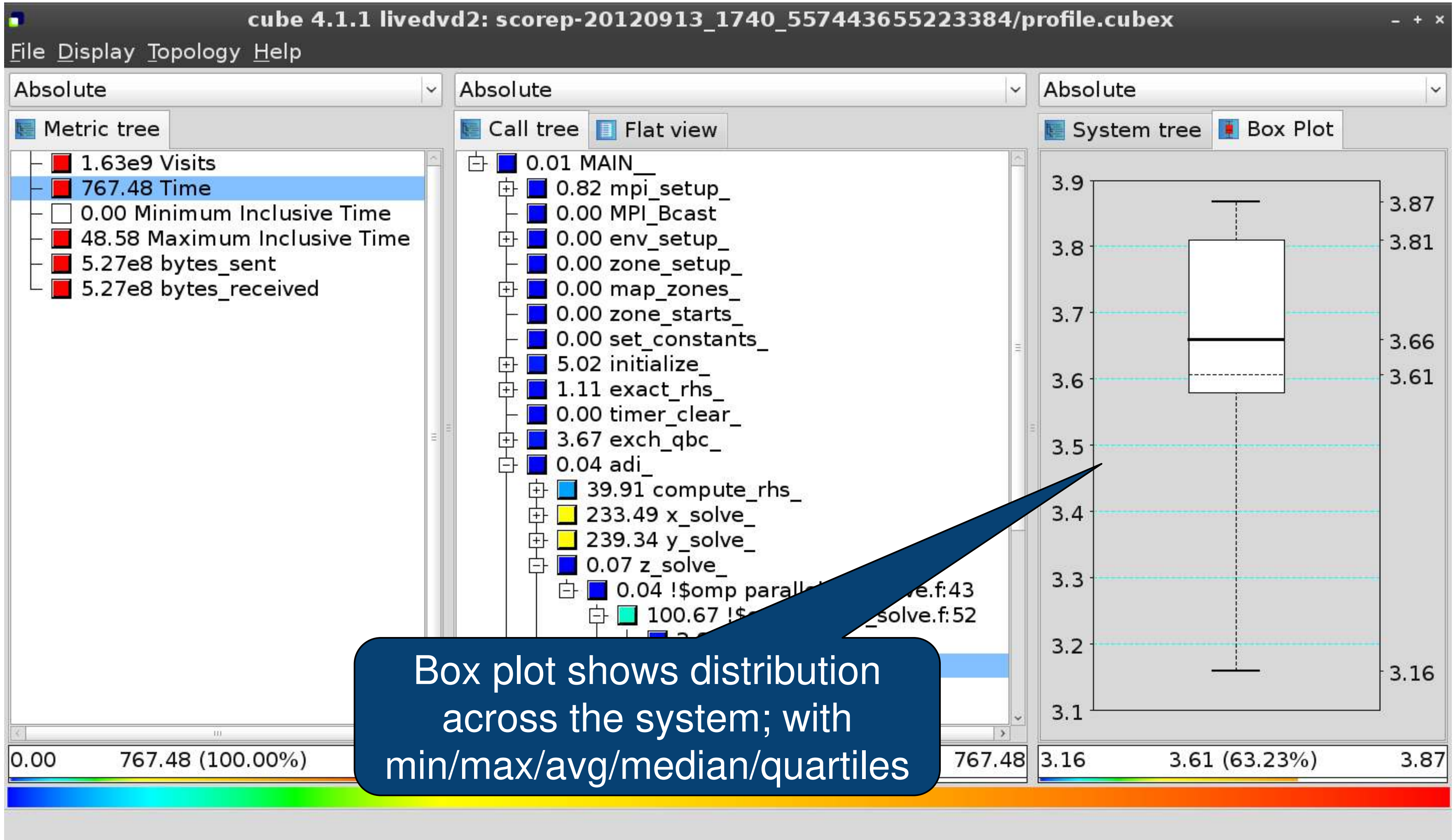


Note:
This feature depends on file and line number information provided by the instrumentation, i.e., it may not always be available

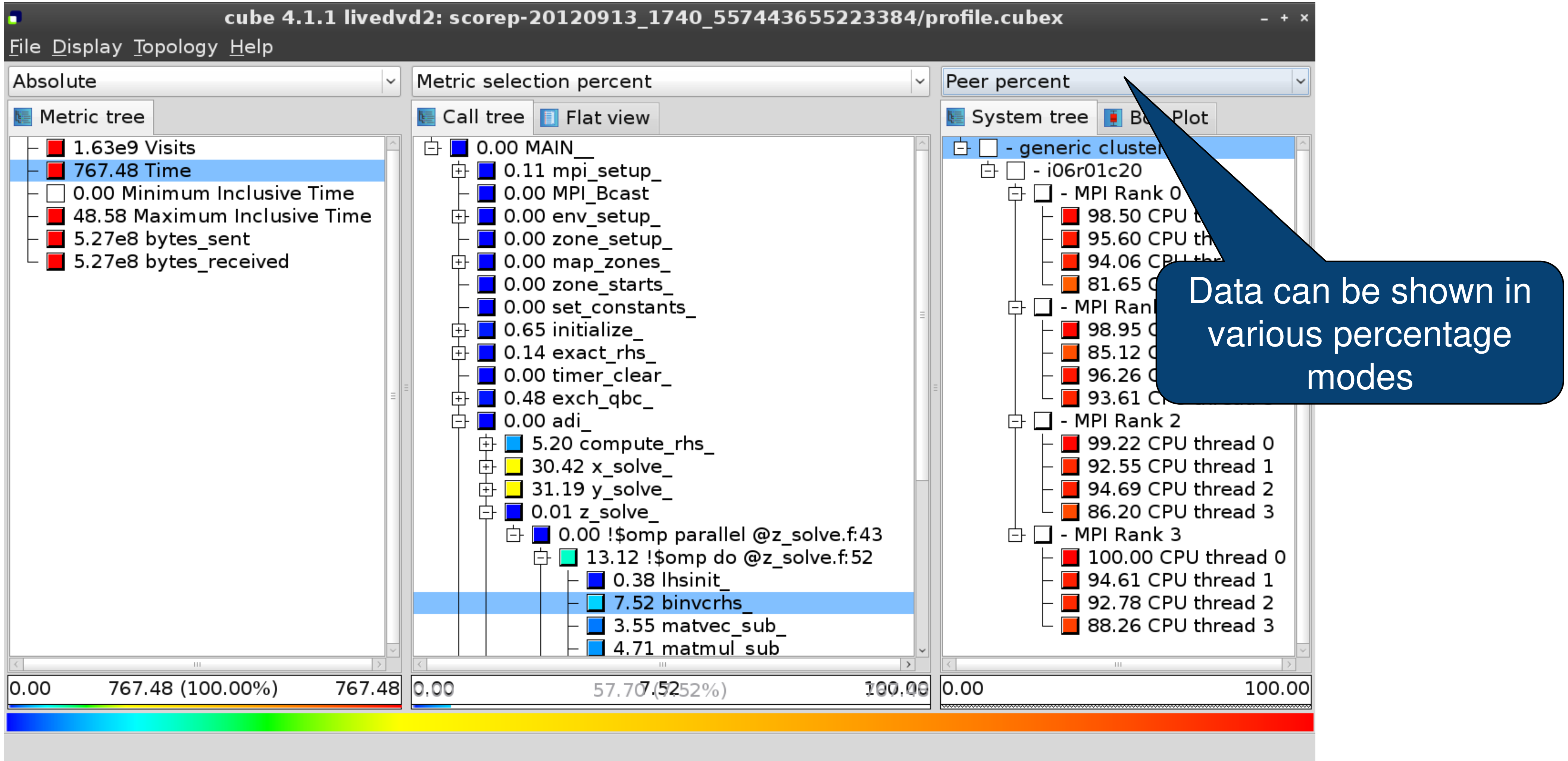
Flat profile view



Box plot view



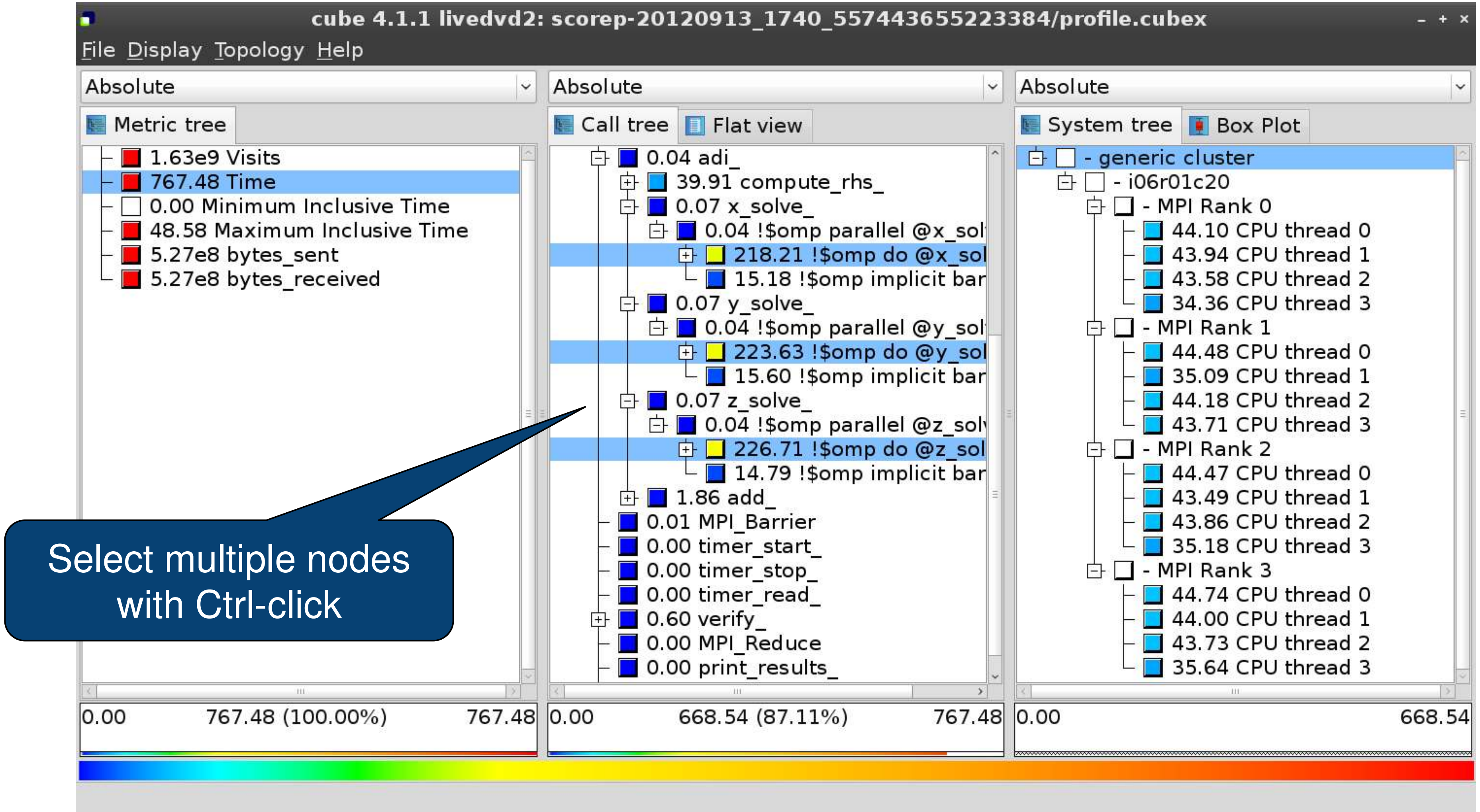
Alternative display modes



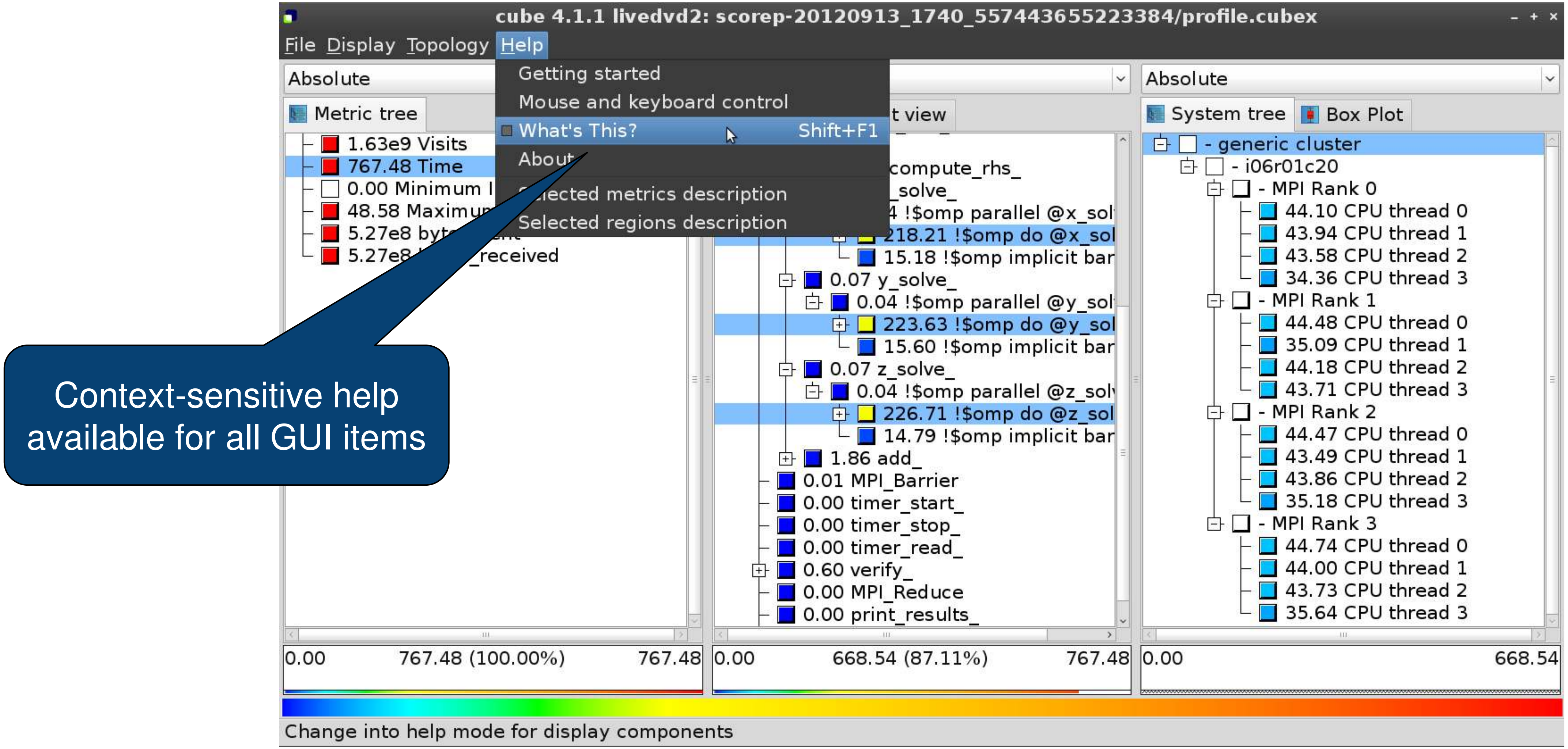
Important display modes

- Absolute
  - Absolute value shown in seconds/bytes/counts
- Selection percent
  - Value shown as percentage w.r.t. the selected node "on the left" (metric/call path)
- Peer percent (system tree only)
  - Value shown as percentage relative to the maximum peer value

Multiple selection



Context-sensitive help



Derived metrics

- Derived metrics are defined using CubePL expressions, e.g.:

  metric::time(i)/metric::visits(e)

- Values of derived metrics are not stored, but calculated on-the-fly
- Types of derived metrics:
  - Prederived: evaluation of the CubePL expression is performed before aggregation
  - Postderived: evaluation of the CubePL expression is performed after aggregation

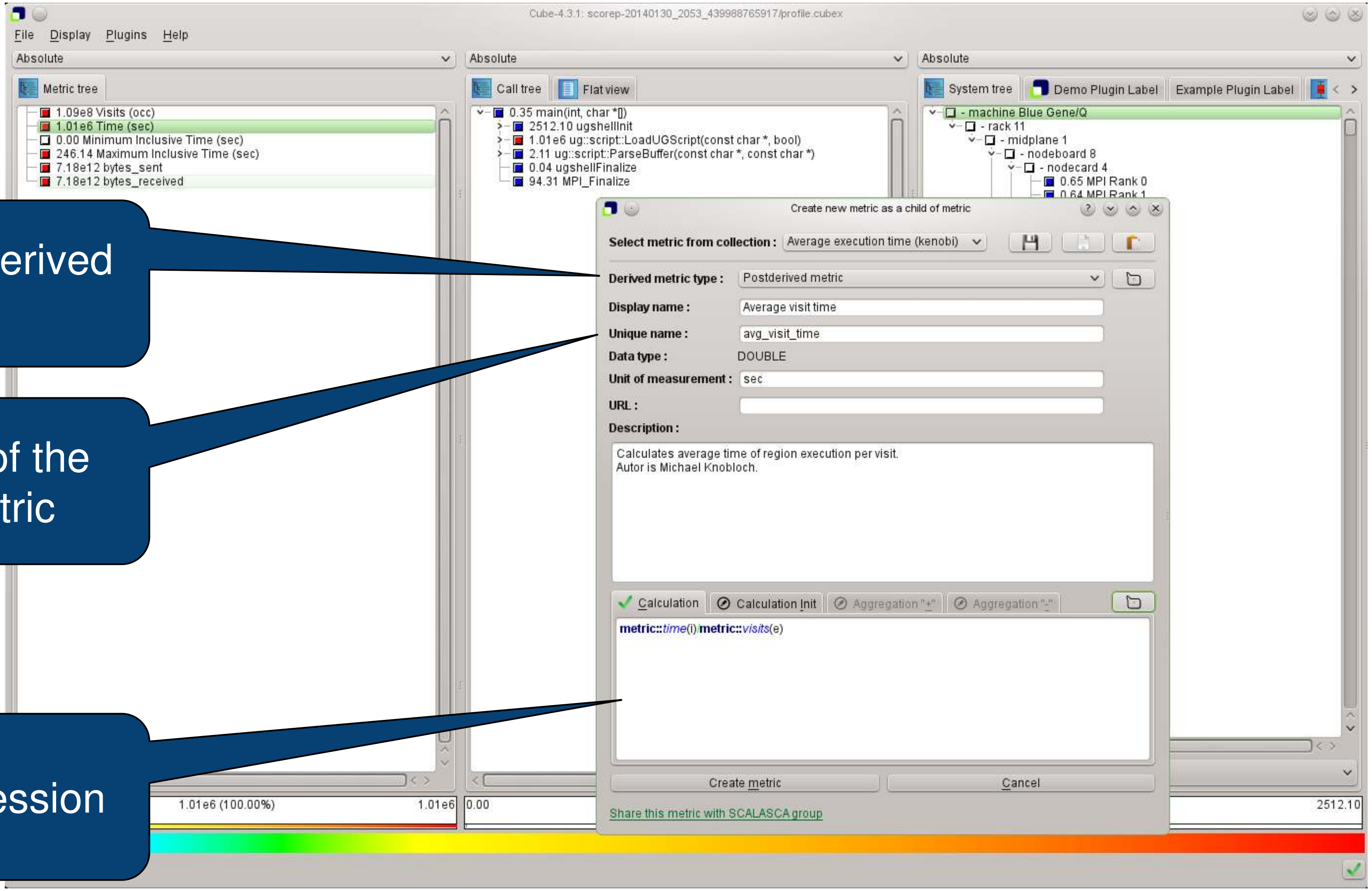
- Examples:
  - "Average execution time": Postderived metric with expression

    metric::time(i)/metric::visits(e)

  - "Number of FLOP per second": Postderived metric with expression

    metric::FLOP()/metric::time()

Derived metrics in Cube GUI

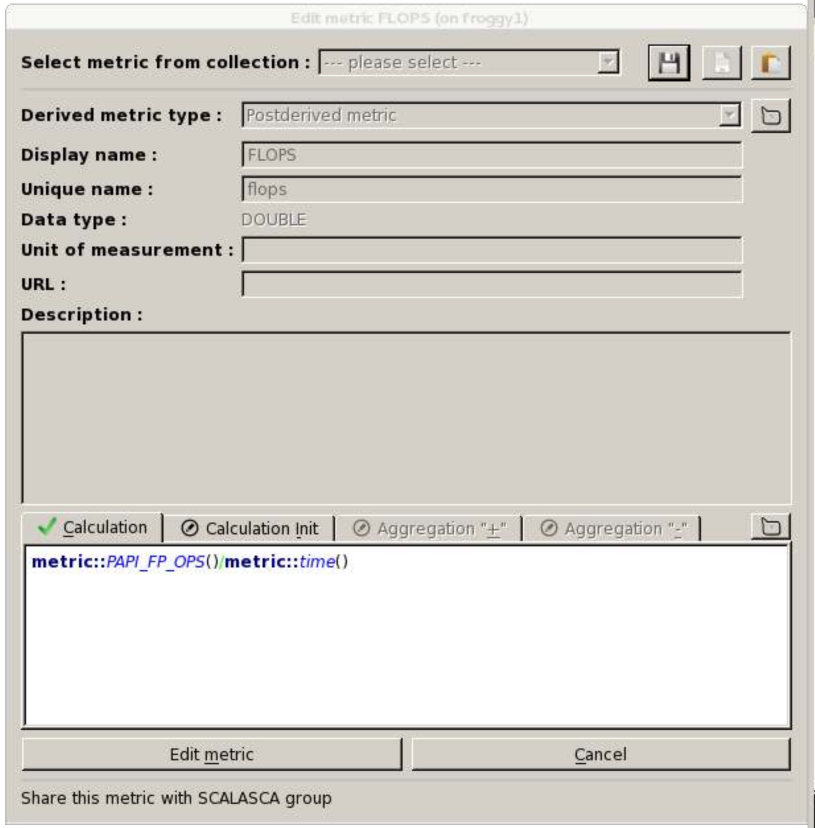


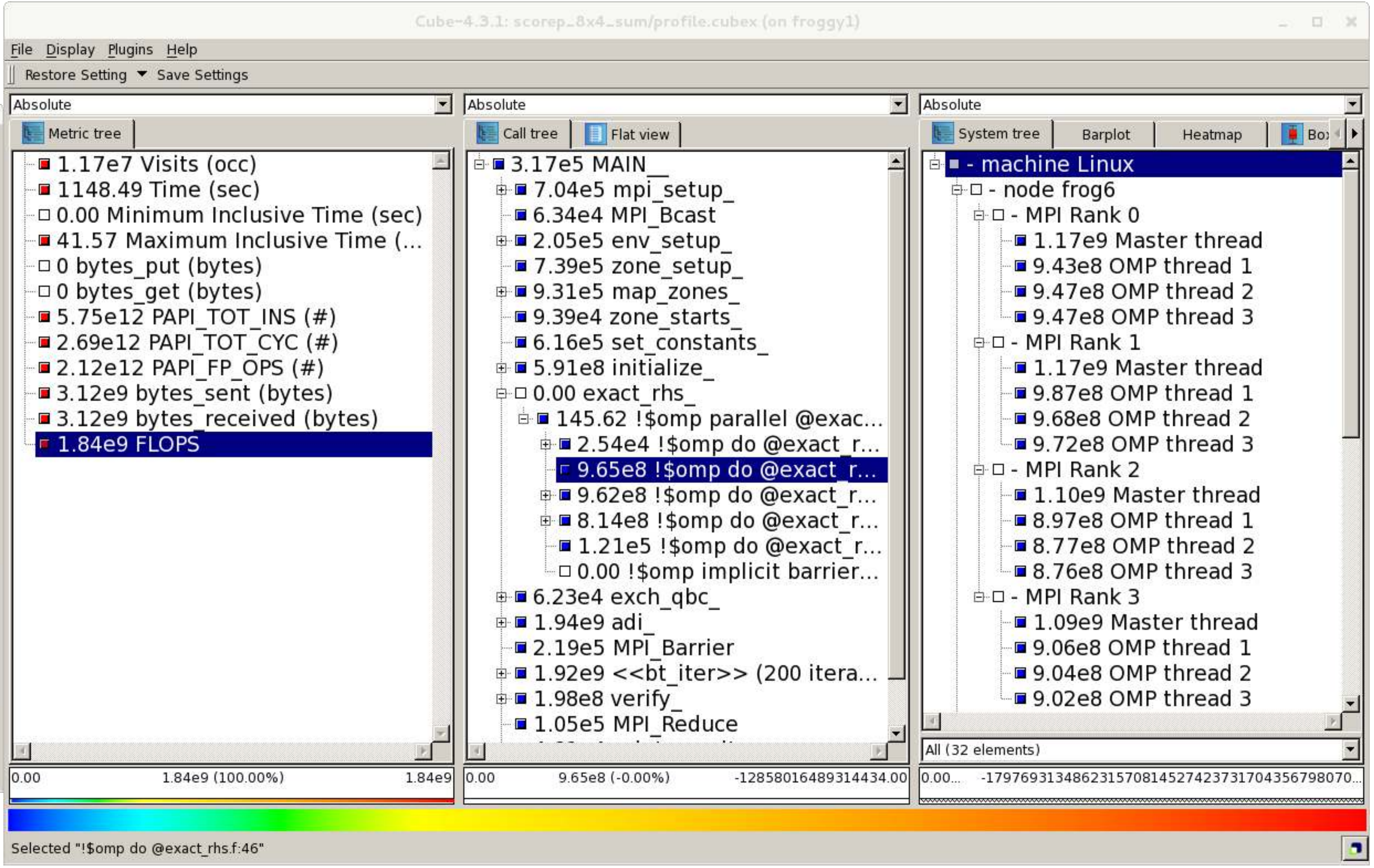
Collection of derived metrics

Parameters of the derived metric

CubePL expression

Example: FLOPS based on PAPI_FP_OPS and time





CUBE algebra utilities

- Extracting solver sub-tree from analysis report

```
% cube_cut  -r '<<ITERATION>>'  scorep_bt-mz_C_32x4_sum/profile.cubex
Writing cut.cubex... done.
```

- Calculating difference of two reports

```
% cube_diff  scorep_bt-mz_C_32x4_sum/profile.cubex  cut.cubex
Writing diff.cubex... done.
```

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

Iteration profiling

- Show time dependent behavior by “unrolling” iterations

- Preparations:
  - Mark loop body by using Score-P instrumentation API in your source code

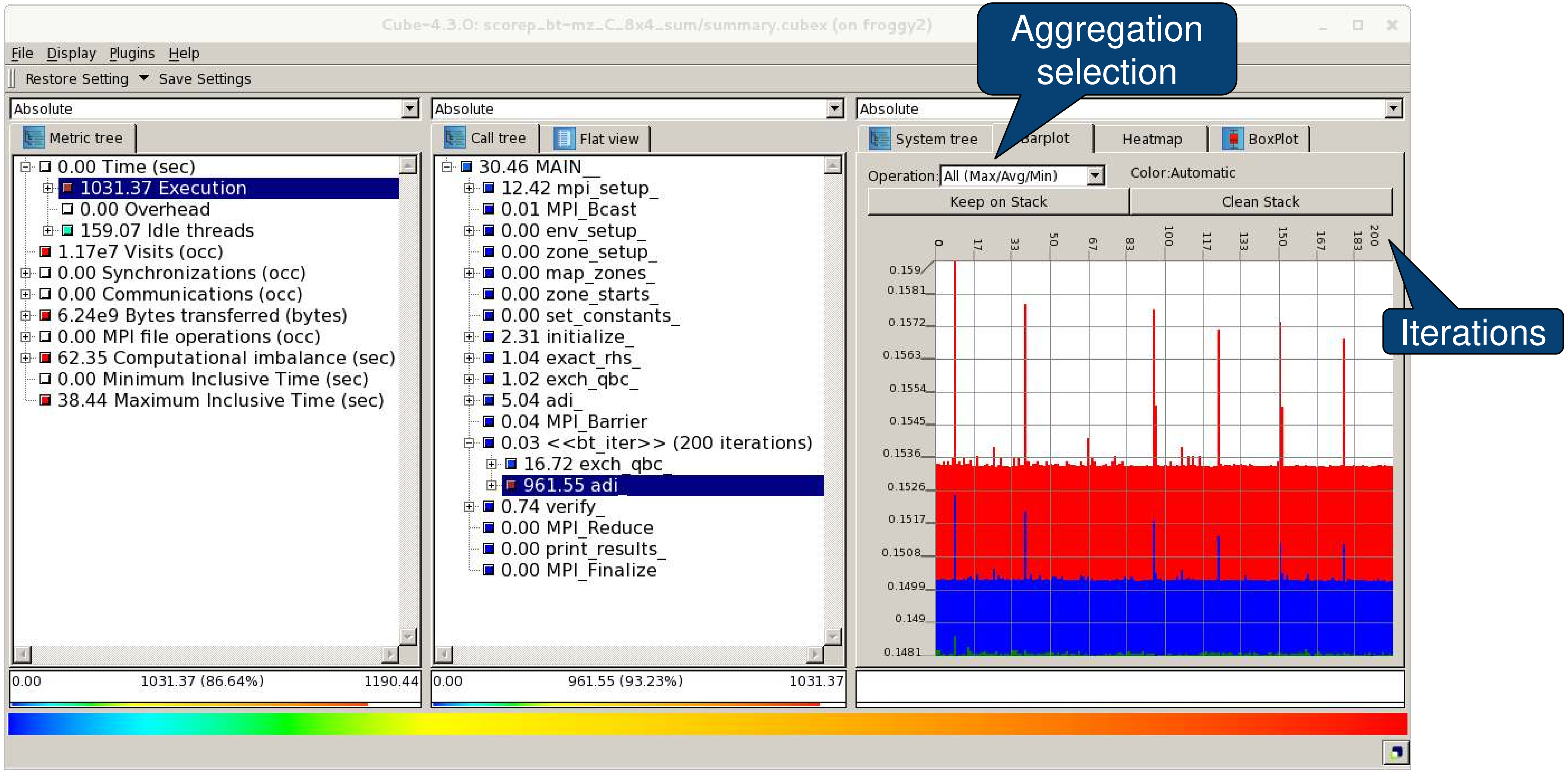
```
SCOREP_USER_REGION_DEFINE( scorep_bt_loop )
SCOREP_USER_REGION_BEGIN( scorep_bt_loop, "<<bt_iter>>", SCOREP_USER_REGION_TYPE_DYNAMIC )
SCOREP_USER_REGION_END( scorep_bt_loop )
```

- Result in the Cube profile:
  - Iterations shown as separate call trees
  - ➢ Useful for checking results for specific iterations

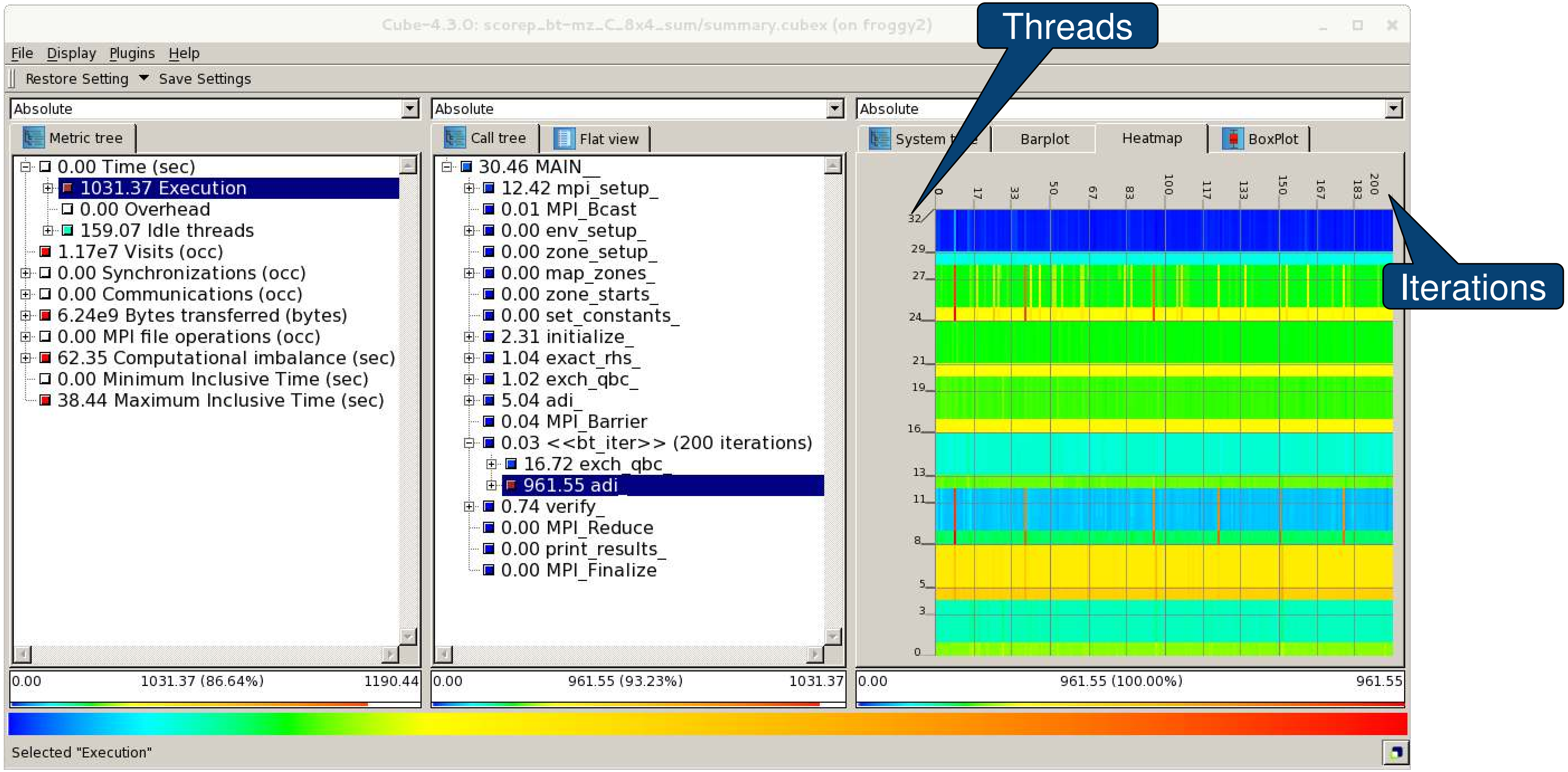
or

  - Select your user-instrumented region and mark it as loop
  - Choose “Hide iterations”
  - ➢ View the Barplot statistics or the (thread x iterations) Heatmap

Iteration profiling: Barplot



Iteration profiling: Heatmap



Cube: Further information

- Parallel program analysis report exploration tools
  - Libraries for Cube report reading & writing
  - Algebra utilities for report processing
  - GUI for interactive analysis exploration
- Available under 3-clause BSD open-source license
- Documentation & sources:
  - http://www.scalasca.org
- User guide also part of installation:
  - \`cube-config --cube-dir\`/share/doc/CubeGuide.pdf
- Contact:
  - mailto: scalasca@fz-juelich.de

Score-P – A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir

VI-HPS Team

Score-P
Scalable performance measurement infrastructure for parallel codes

Congratulations!?

- If you made it this far, you successfully used Score-P to
  - instrument the application
  - analyze its execution with a summary measurement, and
  - examine it with one the interactive analysis report explorer GUIs
- ... revealing the call-path profile annotated with
  - the “Time” metric
  - Visit counts
  - MPI message statistics (bytes sent/received)
- ... but how ***good*** was the measurement?
  - The measured execution produced the desired valid result
  - however, the execution took rather longer than expected!
    - even when ignoring measurement start-up/completion, therefore
    - it was probably dilated by instrumentation/measurement overhead

Performance analysis steps

- 0.0 Reference preparation for validation

- 1.0 Program instrumentation
- 1.1 Summary measurement collection
- 1.2 Summary analysis report examination

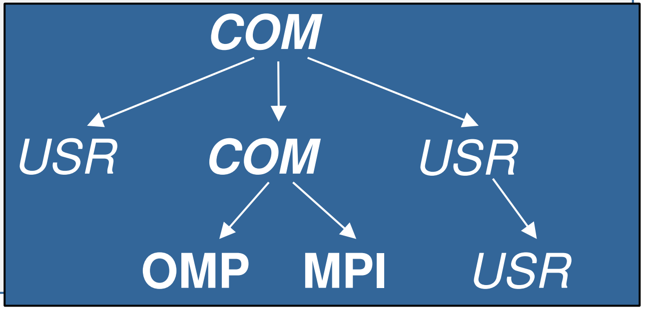
- 2.0 Summary experiment scoring
- 2.1 Summary measurement collection with filtering
- 2.2 Filtered summary analysis report examination

- 3.0 Event trace collection
- 3.1 Event trace examination & analysis

BT-MZ summary analysis result scoring

```
% scorep-score scorep_bt-mz_sum/profile.cubex

Estimated aggregate size of event trace:                   160 GB
Estimated requirements for largest trace buffer (max_buf):   6 GB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):         6 GB
(warning: The memory requirements cannot be satisfied by Score-P to avoid
 intermediate flushes when tracing. Set SCOREP_TOTAL_MEMORY=4G to get the
 maximum supported memory or reduce requirements using USR regions filters.)

flt type      max_buf[B]        visits  time[s] time[%] time/visit[us]  region
     ALL  5,421,104,056 6,586,922,497  8162.56   100.0           1.24  ALL
     USR  5,407,570,350 6,574,832,225  3960.99    48.5           0.60  USR
     OMP     15,783,372    10,975,232  4085.92    50.1         372.29  OMP
     MPI        944,200       386,560    92.05     1.1         238.13  MPI
     COM        665,210       728,480    23.60     0.3          32.40  COM
```

160 GB total memory
6 GB per rank!

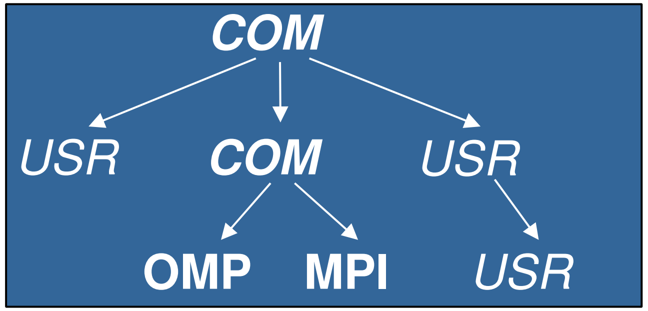


- Report scoring as textual output

- Region/callpath classification
  - **MPI** pure MPI functions
  - **OMP** pure OpenMP regions
  - **USR** user-level computation
  - **COM** “combined” USR+OpenMP/MPI
  - **ANY/ALL** aggregate of all region types

BT-MZ summary analysis report breakdown

```
% scorep-score -r scorep_bt-mz_sum/profile.cubex
  [...]
  [...]
flt type      max_buf[B]          visits  time[s] time[%] time/visit[us]  region
     ALL  5,421,104,056 6,586,922,497  8162.56   100.0             1.24  ALL
     USR  5,407,570,350 6,574,832,225  3960.99    48.5             0.60  USR
     OMP     15,783,372    10,975,232  4085.92    50.1           372.29  OMP
     MPI        944,200       386,560    92.05     1.1           238.13  MPI
     COM        665,210       728,480    23.60     0.3            32.40  COM

     USR  1,741,005,318 2,110,313,472  1204.11    14.8             0.57  matmul_sub_
     USR  1,741,005,318 2,110,313,472   851.97    10.4             0.40  matvec_sub_
     USR  1,741,005,318 2,110,313,472  1754.58    21.5             0.83  binvcrhs_
     USR     76,367,538    87,475,200    65.93     0.8             0.75  lhsinit_
     USR     76,367,538    87,475,200    59.43     0.7             0.68  binvrhs_
     USR     56,913,688    68,892,672    24.62     0.3             0.36  exact_solution_
```



More than
5 GB just for these
6 regions

BT-MZ summary analysis score

- Summary measurement analysis score reveals
  - Total size of event trace would be ~160 GB
  - Maximum trace buffer size would be ~6 GB per rank
    - smaller buffer would require flushes to disk during measurement resulting in substantial perturbation
  - 99.7% of the trace requirements are for USR regions
    - purely computational routines never found on COM call-paths common to communication routines or OpenMP parallel regions
  - These USR regions contribute around 49% 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-MZ summary analysis report filtering

```
% cat ../config/scorep.filt
SCOREP_REGION_NAMES_BEGIN
  EXCLUDE
    binvcrhs*
    matmul_sub*
    matvec_sub*
    exact_solution*
    binvrhs*
    lhs*init*
    timer_*
SCOREP_REGION_NAMES_END

% scorep-score -f ../config/scorep.filt -c 2 \
      scorep_bt-mz_sum/profile.cubex

Estimated aggregate size of event trace:                   1156 MB
Estimated requirements for largest trace buffer (max_buf):   41 MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):         49 MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=49MB to avoid \
>intermediate flushes
 or reduce requirements using USR regions filters.)
```

- Report scoring with prospective filter listing 6 USR regions

1,1 GB of memory in total, 49 MB per rank!

(Including 2 metric values)

BT-MZ summary analysis report filtering

```
% scorep-score -r -f ../config/scorep.filt \
      scorep_bt-mz_sum/profile.cubex
flt type     max_buf[B]          visits  time[s] time[%] time/       region
                                                         visit[us]
 -    ALL 5,421,104,056 6,586,922,497  8162.56   100.0       1.24  ALL
 -    USR 5,407,570,350 6,574,832,225  3960.99    48.5       0.60  USR
 -    OMP    15,783,372    10,975,232  4085.92    50.1     372.29  OMP
 -    MPI       944,200       386,560    92.05     1.1     238.13  MPI
 -    COM       665,210       728,480    23.60     0.3      32.40  COM

 *    ALL    17,390,726    12,138,209  4201.91    51.5     346.17  ALL-FLT
 +    FLT 5,407,531,376 6,574,784,288  3960.65    48.5       0.60  FLT
 -    OMP    15,783,372    10,975,232  4085.92    50.1     372.29  OMP-FLT
 -    MPI       944,200       386,560    92.05     1.1     238.13  MPI-FLT
 *    COM       665,210       728,480    23.60     0.3      32.40  COM-FLT
 *    USR        38,974        47,937     0.34     0.0       7.14  USR-FLT

 +    USR 1,741,005,318 2,110,313,472  1204.11    14.8       0.57  matmul_sub_
 +    USR 1,741,005,318 2,110,313,472   851.97    10.4       0.40  matvec_sub_
 +    USR 1,741,005,318 2,110,313,472  1754.58    21.5       0.83  binvcrhs_
 +    USR    76,367,538    87,475,200    65.93     0.8       0.75  lhsinit_
 +    USR    76,367,538    87,475,200    59.43     0.7       0.68  binvrhs_
 +    USR    56,913,688    68,892,672    24.62     0.3       0.36  exact_solution_
```

- Score report breakdown by region

Filtered routines marked with '+'

BT-MZ filtered summary measurement

```
% cd bin.scorep
% cp ../jobscript/stampede2/scorep.sbatch .
% vim scorep.sbatch

# Score-P measurement configuration
export SCOREP_EXPERIMENT_DIRECTORY=scorep_bt-mz_sum_filter
export SCOREP_FILTERING_FILE=../config/scorep.filt
#export SCOREP_TOTAL_MEMORY=50M
#export SCOREP_METRIC_PAPI=PAPI_TOT_INS,PAPI_TOT_CYC
#export SCOREP_ENABLE_TRACING=true

# Run the application
ibrun ./bt-mz_${CLASS}.${PROCS}

% sbatch ./scorep.sbatch
```

- Set new experiment directory and re-run measurement with new filter configuration

- Submit job

Score-P filtering

```
% cat ../config/scorep.filt
SCOREP_REGION_NAMES_BEGIN
  EXCLUDE
    binvcrhs*
    matmul_sub*
    matvec_sub*
    exact_solution*
    binvrhs*
    lhs*init*
    timer_*
SCOREP_REGION_NAMES_END

% export SCOREP_FILTERING_FILE=\
../config/scorep.filt
```

Region name filter block using wildcards

Apply filter

- Filtering by source file name
  - All regions in files that are excluded by the filter are ignored
- Filtering by region name
  - All regions that are excluded by the filter are ignored
  - Overruled by source file filter for excluded files
- Apply filter by
  - exporting **SCOREP_FILTERING_FILE** environment variable
- Apply filter at
  - Run-time
  - Compile-time (GCC-plugin only)
    - Add cmd-line option **--instrument-filter**
    - No overhead for filtered regions but recompilation

Source file name filter block

- Keywords
  - Case-sensitive
  - SCOREP_FILE_NAMES_BEGIN, SCOREP_FILE_NAMES_END
    - Define the source file name filter block
    - Block contains EXCLUDE, INCLUDE rules
  - EXCLUDE, INCLUDE rules
    - Followed by one or multiple white-space separated source file names
    - Names can contain bash-like wildcards *, ?, []
    - Unlike bash, * may match a string that contains slashes
- EXCLUDE, INCLUDE rules are applied in sequential order
- Regions in source files that are excluded after all rules are evaluated, get filtered

```
# This is a comment
SCOREP_FILE_NAMES_BEGIN
  # by default, everything is included
  EXCLUDE */foo/bar*
  INCLUDE */filter_test.c
SCOREP_FILE_NAMES_END
```

Region name filter block

- Keywords
  - Case-sensitive
  - `SCOREP_REGION_NAMES_BEGIN`, `SCOREP_REGION_NAMES_END`
    - Define the region name filter block
    - Block contains `EXCLUDE`, `INCLUDE` rules
  - `EXCLUDE`, `INCLUDE` rules
    - Followed by one or multiple white-space separated region names
    - Names can contain bash-like wildcards `*`, `?`, `[]`
- `EXCLUDE`, `INCLUDE` rules are applied in sequential order
- Regions that are excluded after all rules are evaluated, get filtered

```
# This is a comment
SCOREP_REGION_NAMES_BEGIN
  # by default, everything is included
  EXCLUDE *
  INCLUDE bar foo
          baz
          main
SCOREP_REGION_NAMES_END
```

Region name filter block, mangling

- Name mangling
  - Filtering based on names seen by the measurement system
    - Dependent on compiler
    - Actual name may be mangled
- `scorep-score` names as starting point (e.g. `matvec_sub_`)
  - Use `*` for Fortran trailing underscore(s) for portability
  - Use `?` and `*` as needed for full signatures or overloading

```
void bar(int* a) {
    *a++;
}
int main() {
    int i = 42;
    bar(&i);
    return 0;
}
```

```
# filter bar:
# for gcc-plugin, scorep-score
# displays 'void bar(int*)',
# other compilers may differ

SCOREP_REGION_NAMES_BEGIN
  EXCLUDE void?bar(int?)
SCOREP_REGION_NAMES_END
```

Further information

- Community instrumentation & measurement infrastructure
  - Instrumentation (various methods)
  - Basic and advanced profile generation
  - Event trace recording
  - Online access to profiling data
- Available under 3-clause BSD open-source license
- Documentation & Sources:
  - http://www.score-p.org
- User guide also part of installation:
  - `<prefix>/share/doc/scorep/{pdf,html}/`
- Support and feedback: support@score-p.org
- Subscribe to news@score-p.org, to be up to date

VI-HPS

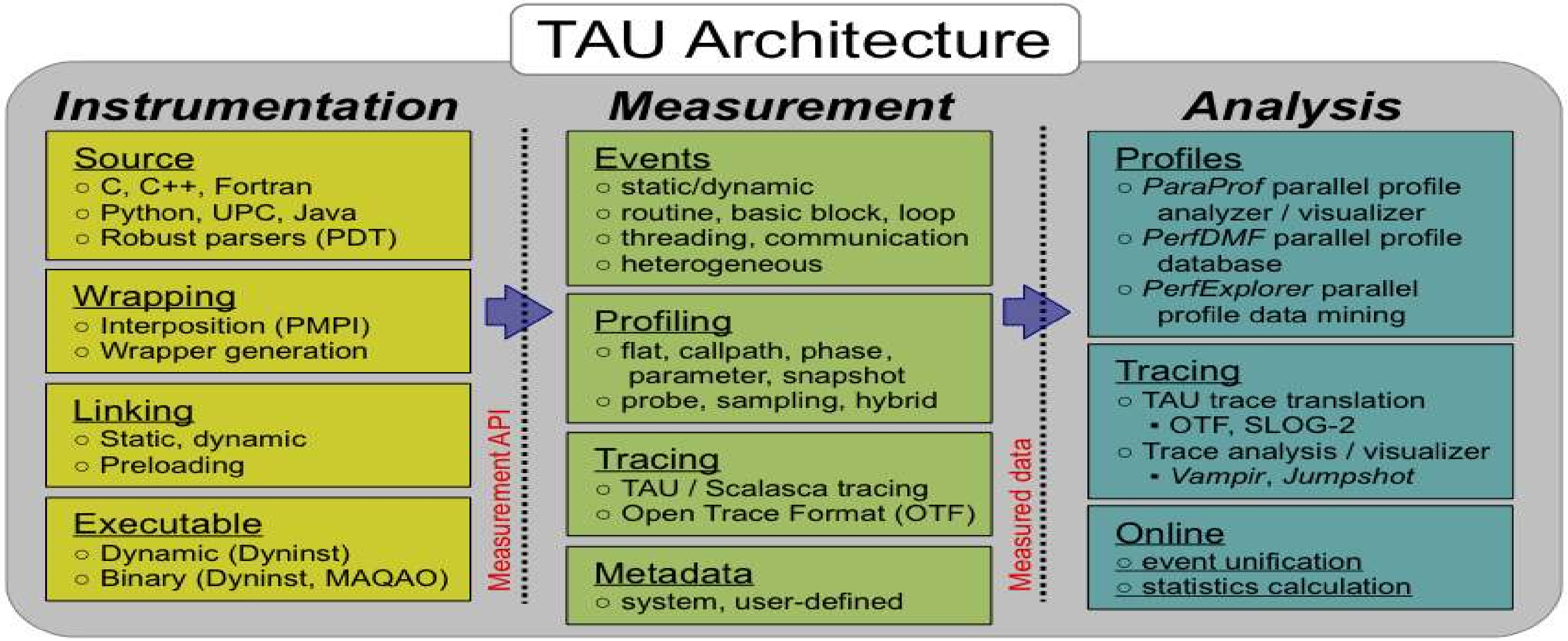
Examination and Visualization of profiles with TAU

Tuning and Analysis Utilities

Sameer Shende
sameer@cs.uoregon.edu
University of Oregon
http://tau.uoregon.edu

TAU Performance System®

- Parallel performance framework and toolkit
  - Supports all HPC platforms, compilers, runtime system
  - Provides portable instrumentation, measurement, analysis



TAU Performance System

- Instrumentation
  - Fortran, C++, C, UPC, Java, Python, Chapel
  - Automatic instrumentation
- Measurement and analysis support
  - MPI, OpenSHMEM, ARMCI, PGAS, DMAPP
  - pthreads, OpenMP, OMPT interface, hybrid, other thread models
  - GPU, CUDA, OpenCL, OpenACC
  - Parallel profiling and tracing
  - Use of Score-P for native OTF2 and CUBEX generation
  - Efficient callpath proflles and trace generation using Score-P
- Analysis
  - Parallel profile analysis (ParaProf), data mining (PerfExplorer)
  - Performance database technology (TAUdb)
  - 3D profile browser

TAU Performance System

- TAU supports both sampling and direct instrumentation
- Memory debugging as well as I/O performance evaluation
- Profiling as well as tracing
- Interfaces with Score-P for more efficient measurements
- TAU's instrumentation covers:
  - Runtime library interposition (tau_exec)
  - Compiler-based instrumentation
  - Native generation of OTF2 traces (TAU_TRACE=1, TAU_TRACE_FORMAT=otf2)
  - Callsite instrumentation with profiles and traces (TAU_CALLSITE=1)
  - PDT based Source level instrumentation: routine & loop
  - Event based sampling (TAU_SAMPLING=1 or tau_exec -ebs)
  - Callstack unwinding with sampling (TAU_EBS_UNWIND=1)
  - OpenMP Tools Interface (OMPT, tau_exec –T ompt)
  - CUDA CUPTI, OpenCL (tau_exec -T cupti -cupti)

Application Performance Engineering using TAU

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

Using TAU

- TAU supports several compilers, measurement, and thread options
  - Intel compilers, profiling with hardware counters using PAPI, MPI library, CUDA…
  - Each measurement configuration of TAU corresponds to a unique stub makefile (configuration file) and library that is generated when you configure it
- To instrument source code automatically using PDT
  - Choose an appropriate TAU stub makefile in <arch>/lib:
  - **% module load tau**
  - **% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-ompt-mpi-pdt-openmp**
    % export TAU_OPTIONS= '-optVerbose …' (see tau_compiler.sh)
  - Use tau_f90.sh, tau_cxx.sh, tau_upc.sh, or tau_cc.sh as F90, C++, UPC, or C compilers respectively:
  - **% mpif90 foo.f90 changes to**
  - **% tau_f90.sh foo.f90**
- Set runtime environment variables, execute application and analyze performance data:
  - **% pprof (for text based profile display)**
  - **% paraprof (for GUI)**

Installing and Configuring TAU

- Installing PDT:
  - wget http://tau.uoregon.edu/pdt_lite.tgz
  - ./configure –prefix=<dir>; make ; make install
- Installing TAU:
  - wget http://tau.uoregon.edu/tau.tgz
  - ./configure –scorep=download –arch=x86_64 -bfd=download -pdt=<dir> -papi=<dir> ...
  - For MIC (KNC):
  - ./configure -scorep=download –arch=mic_linux –pdt=<dir> -pdt_c++=g++ -papi=dir …
  - make install
- Using TAU:
  - export TAU_MAKEFILE=<taudir>/x86_64/lib/Makefile.tau-<TAGS>
  - make CC=tau_cc.sh CXX=tau_cxx.sh F90=tau_f90.sh

Different Makefiles for TAU Compiler and Runtime Options

```
%  . ~tg828282/Tutorial/vihps.sh
% ls $TAU/Makefile.*
Makefile.tau-icpc-papi-mpi-pdt
Makefile.tau-icpc-papi-mpi-pdt-openmp-opari
Makefile.tau-icpc-papi-ompt-mpi-pdt-openmp
Makefile.tau-icpc-papi-mpi-pdt-openmp-opari-scorep
Makefile.tau-icpc-papi-mpi-pdt-scorep
Makefile.tau-icpc-papi-ompt-mpi-pdt-openmp
```

- For an MPI+OpenMP+F90 application with Intel MPI, you may choose

  Makefile.tau-icpc-papi-mpi-pdt
  - Supports MPI instrumentation & PDT for automatic source instrumentation

```
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-ompt-mpi-pdt-openmp

% tau_f90.sh matmult.f90 -o matmult
% mpirun -np 256 ./matmult
% paraprof
```

Compile-Time Options

Optional parameters for the TAU_OPTIONS environment variable:

% tau_compiler.sh

-optVerbose	Turn on verbose debugging messages
-optCompInst	Use compiler based instrumentation
-optNoCompInst	Do not revert to compiler instrumentation if source instrumentation fails.
-optTrackIO	Wrap POSIX I/O call and calculates vol/bw of I/O operations (configure TAU with –*iowrapper*)
-optTrackGOMP	Enable tracking GNU OpenMP runtime layer (used without –opari)
-optMemDbg	Enable runtime bounds checking (see TAU_MEMDBG_* env vars)
-optKeepFiles	Does not remove intermediate .pdb and .inst.* files
-optPreProcess	Preprocess sources (OpenMP, Fortran) before instrumentation
-optTauSelectFile=”<file>"	Specify selective instrumentation file for *tau_instrumentor*
-optTauWrapFile=”<file>"	Specify path to *link_options.tau* generated by *tau_gen_wrapper*
-optHeaderInst	Enable Instrumentation of headers
-optTrackUPCR	Track UPC runtime layer routines (used with tau_upc.sh)
-optLinking=""	Options passed to the linker. Typically $(TAU_MPI_FLIBS) $(TAU_LIBS) $(TAU_CXXLIBS)
-optCompile=""	Options passed to the compiler. Typically $(TAU_MPI_INCLUDE) $(TAU_INCLUDE) $(TAU_DEFS)
-optPdtF95Opts=""	Add options for Fortran parser in PDT (f95parse/gfparse) …

Compile-Time Options (contd.)

- Optional parameters for the TAU_OPTIONS environment variable:

% tau_compiler.sh

-optMICOffload	Links code for Intel MIC offloading, requires both host and MIC TAU libraries
-optShared	Use TAU's shared library (libTAU.so) instead of static library (default)
-optPdtCxxOpts=""	Options for C++ parser in PDT (cxxparse).
-optPdtF90Parser=""	Specify a different Fortran parser
-optPdtCleanscapeParser	Specify the Cleanscape Fortran parser instead of GNU gfparser
-optTau=""	Specify options to the tau_instrumentor
-optTrackDMAPP	Enable instrumentation of low-level DMAPP API calls on Cray
-optTrackPthread	Enable instrumentation of pthread calls

See tau_compiler.sh for a full list of TAU_OPTIONS.

…

Compiling Fortran Codes with TAU

- If your Fortran code uses free format in .f files (fixed is default for .f), you may use:
  % export TAU_OPTIONS= '-optPdtF95Opts="-R free" -optVerbose '

- To use the compiler based instrumentation instead of PDT (source-based):
  % export TAU_OPTIONS= '-optCompInst -optVerbose'

- If your Fortran code uses C preprocessor directives (#include, #ifdef, #endif):
  % export TAU_OPTIONS= '-optPreProcess -optVerbose -optDetectMemoryLeaks'

- To use an instrumentation specification file:
  % export TAU_OPTIONS= '-optTauSelectFile=select.tau -optVerbose -optPreProcess'
  % cat select.tau
  BEGIN_INSTRUMENT_SECTION
  loops routine="#"
  # this statement instruments all outer loops in all routines. # is wildcard as well as comment in first column.
  END_INSTRUMENT_SECTION

Runtime Environment Variables

Environment Variable	Default	Description
TAU_TRACE	0	Setting to 1 turns on tracing
TAU_TRACE_FORMAT	default	Setting to “otf2” generates OTF2 traces natively.
TAU_CALLPATH	0	Setting to 1 turns on callpath profiling
TAU_CALLSITE	0	Setting to 1 generates callsite information in events. May be used with tracing.
TAU_TRACK_MEMORY_FOOTPRINT	0	Setting to 1 turns on tracking memory usage by tracking the resident set size and high water mark of memory usage
TAU_TRACK_LOAD	0	Setting to 1 tracks system load periodically.
TAU_CALLPATH_DEPTH	2	Specifies depth of callpath. Setting to 0 generates no callpath or routine information, setting to 1 generates flat profile and context events have just parent information (e.g., Heap Entry: foo)
TAU_SAMPLING	1	Setting to 1 enables event-based sampling.
TAU_TRACK_SIGNALS	0	Setting to 1 generate debugging callstack info when a program crashes
TAU_COMM_MATRIX	0	Setting to 1 generates communication matrix display using context events
TAU_THROTTLE	1	Setting to 0 turns off throttling. Throttles instrumentation in lightweight routines that are called frequently
TAU_THROTTLE_NUMCALLS	100000	Specifies the number of calls before testing for throttling
TAU_THROTTLE_PERCALL	10	Specifies value in microseconds. A routine is throttled if it takes less than 10 microseconds per call (and called > 10000 times).
TAU_COMPENSATE	0	Setting to 1 enables runtime compensation of instrumentation overhead
TAU_PROFILE_FORMAT	Profile	Setting to “merged” generates a single file. “snapshot” generates xml format
TAU_METRICS	TIME	Setting to a comma separated list generates other metrics. (e.g., ENERGY,TIME,P_VIRTUAL_TIME,PAPI_FP_INS,PAPI_NATIVE_<event>:<subevent>)

Runtime Environment Variables (contd.)

Environment Variable	Default	Description
TAU_TRACK_MEMORY_LEAKS	0	Tracks allocates that were not de-allocated (needs –optMemDbg or tau_exec –memory)
TAU_EBS_SOURCE	TIME	Allows using PAPI hardware counters for periodic interrupts for EBS (e.g., TAU_EBS_SOURCE=PAPI_TOT_INS when TAU_SAMPLING=1)
TAU_EBS_PERIOD	100000	Specifies the overflow count for interrupts
TAU_MEMDBG_ALLOC_MIN/MAX	0	Byte size minimum and maximum subject to bounds checking (used with TAU_MEMDBG_PROTECT_*)
TAU_MEMDBG_OVERHEAD	0	Specifies the number of bytes for TAU's memory overhead for memory debugging.
TAU_MEMDBG_PROTECT_BELOW/ABOVE	0	Setting to 1 enables tracking runtime bounds checking below or above the array bounds (requires –optMemDbg while building or tau_exec –memory)
TAU_MEMDBG_ZERO_MALLOC	0	Setting to 1 enables tracking zero byte allocations as invalid memory allocations.
TAU_MEMDBG_PROTECT_FREE	0	Setting to 1 detects invalid accesses to deallocated memory that should not be referenced until it is reallocated (requires –optMemDbg or tau_exec –memory)
TAU_MEMDBG_ATTEMPT_CONTINUE	0	Setting to 1 allows TAU to record and continue execution when a memory error occurs at runtime.
TAU_MEMDBG_FILL_GAP	Undefined	Initial value for gap bytes
TAU_MEMDBG_ALINGMENT	Sizeof(int)	Byte alignment for memory allocations
TAU_EVENT_THRESHOLD	0.5	Define a threshold value (e.g., .25 is 25%) to trigger marker events for min/max

Simplifying TAU's usage (tau_exec)

- Uninstrumented execution
  - % mpirun -np 4 ./a.out
- Track MPI performance
  - % mpirun -np 4 tau_exec ./a.out
- Track POSIX I/O and MPI performance (MPI enabled by default)
  - % mpirun -np 4 tau_exec –T mpi,pdt –io ./a.out
- Track memory operations
  - % export TAU_TRACK_MEMORY_LEAKS=1
  - % mpirun –np 8 tau_exec –memory_debug ./a.out (bounds check)
- Use event based sampling (compile with –g)
  - % mpirun –np 8 tau_exec –ebs ./a.out
  - Also –ebs_source=<PAPI_COUNTER> -ebs_period=<overflow_count>
- Load wrapper interposition library
  - % mpirun –np 8 tau_exec –loadlib=<path/libwrapper.so> ./a.out
- Track GPGPU operations
  - % mpirun –np 8 tau_exec –cupti ./a.out
  - % mpirun –np 8 tau_exec –opencl ./a.out
  - % mpirun –np 8 tau_exec –openacc ./a.out

Binary Rewriting Instrumentation

- Support for both static and dynamic executables
- Specify a list of routines to instrument
- Specify the TAU measurement library to be injected
- MAQAO [UVSQ, Intel Exascale Labs]:

```
% tau_rewrite -T [tags] a.out -o a.inst
```

- DyninstAPI [U. Maryland and U. Wisconsin, Madison]:

```
% tau_run -T [tags] a.out -o a.inst
```

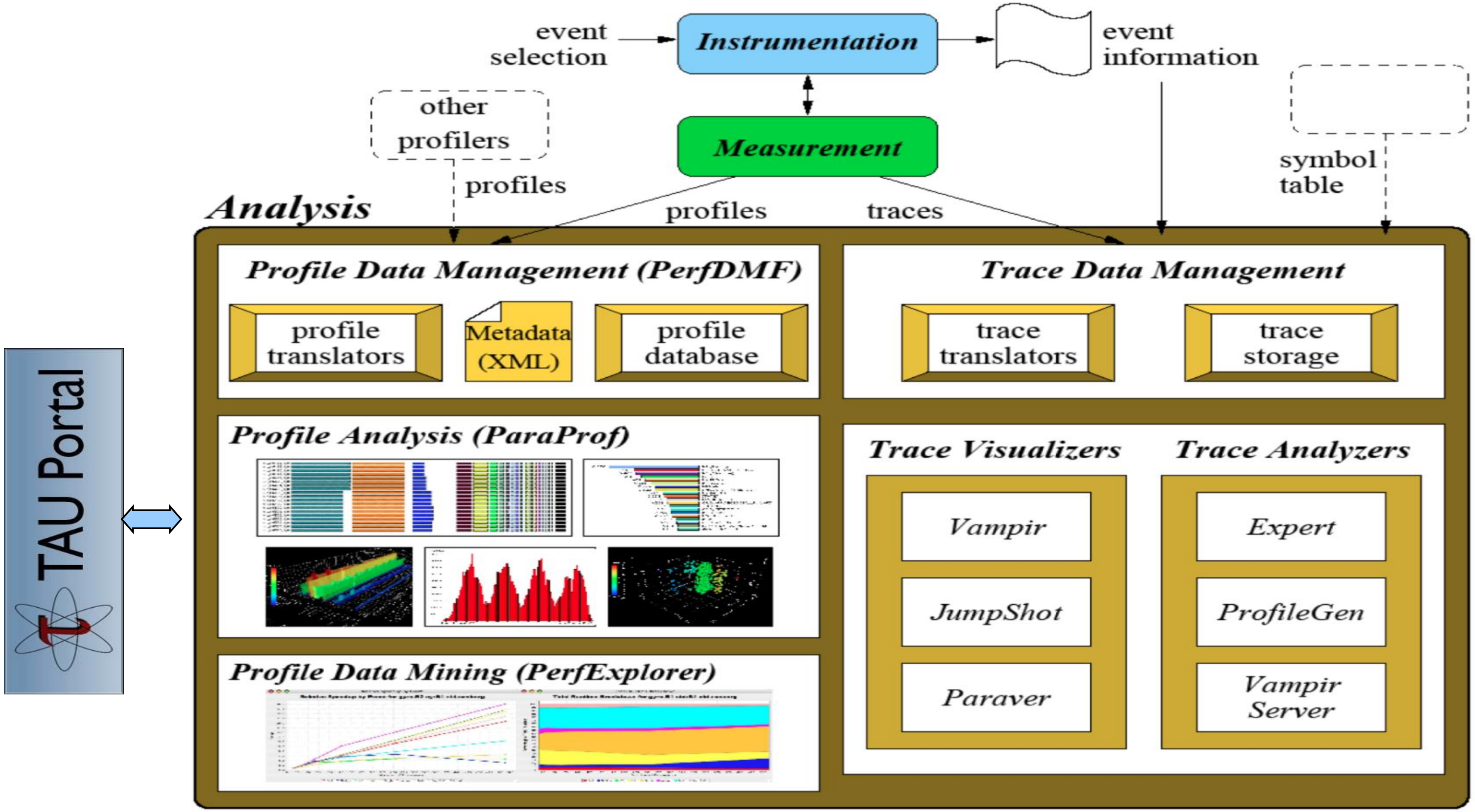
- Pebil [UC San Diego]:

```
% tau_pebil_rewrite -T [tags] a.out -o a.inst
```

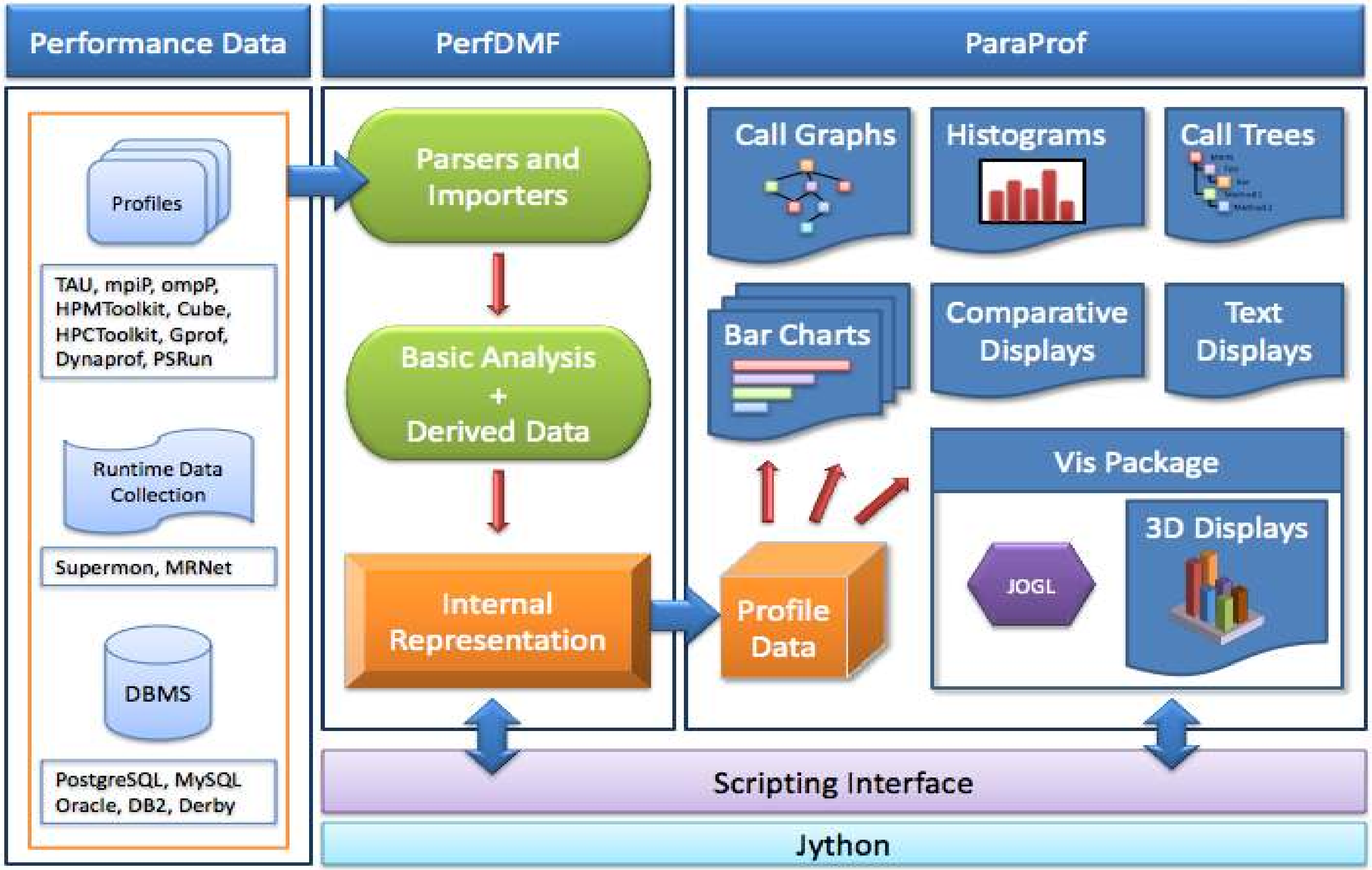
- Execute the application to get measurement data:

```
% mpirun -np 256 ./a.inst
```

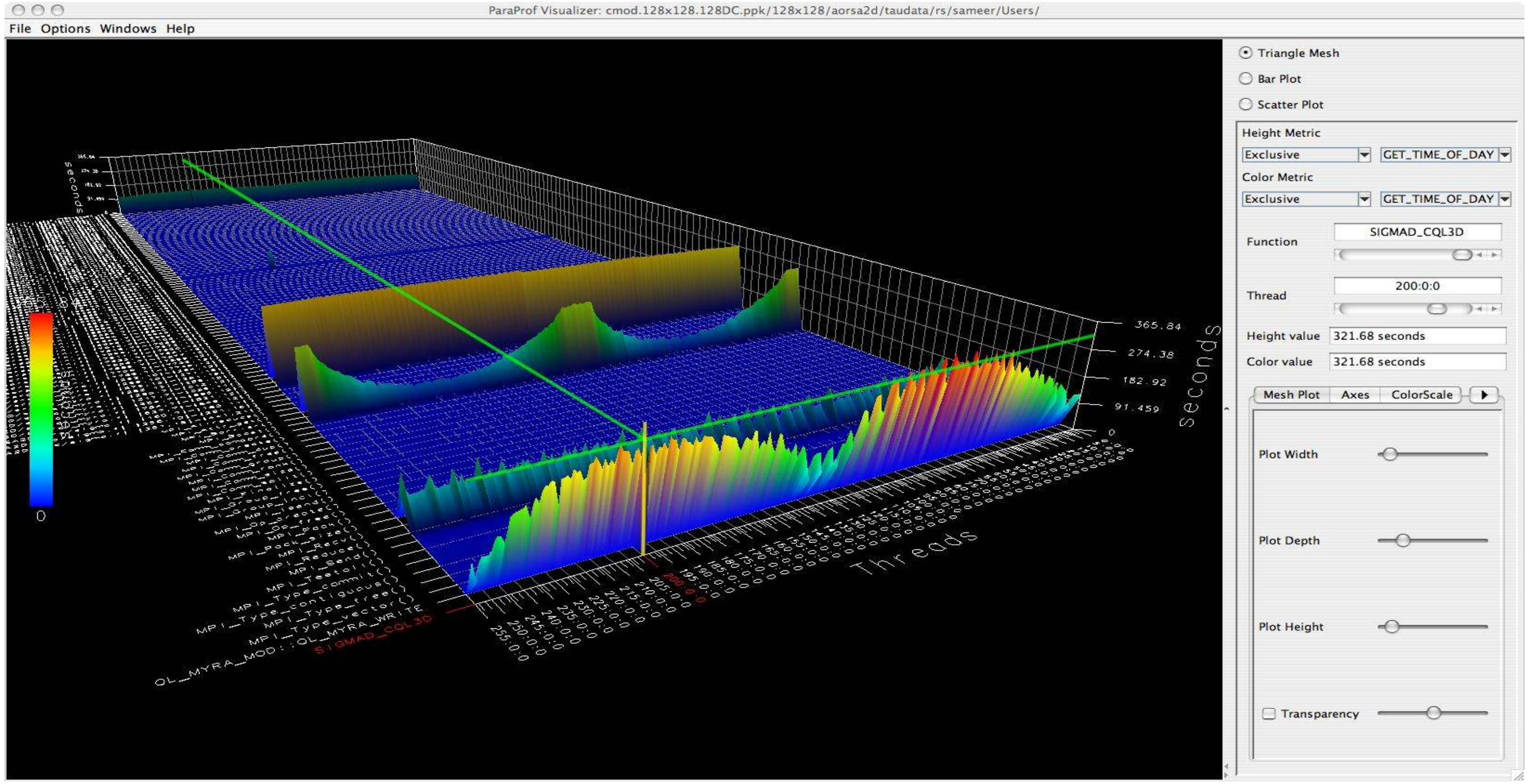
TAU Analysis



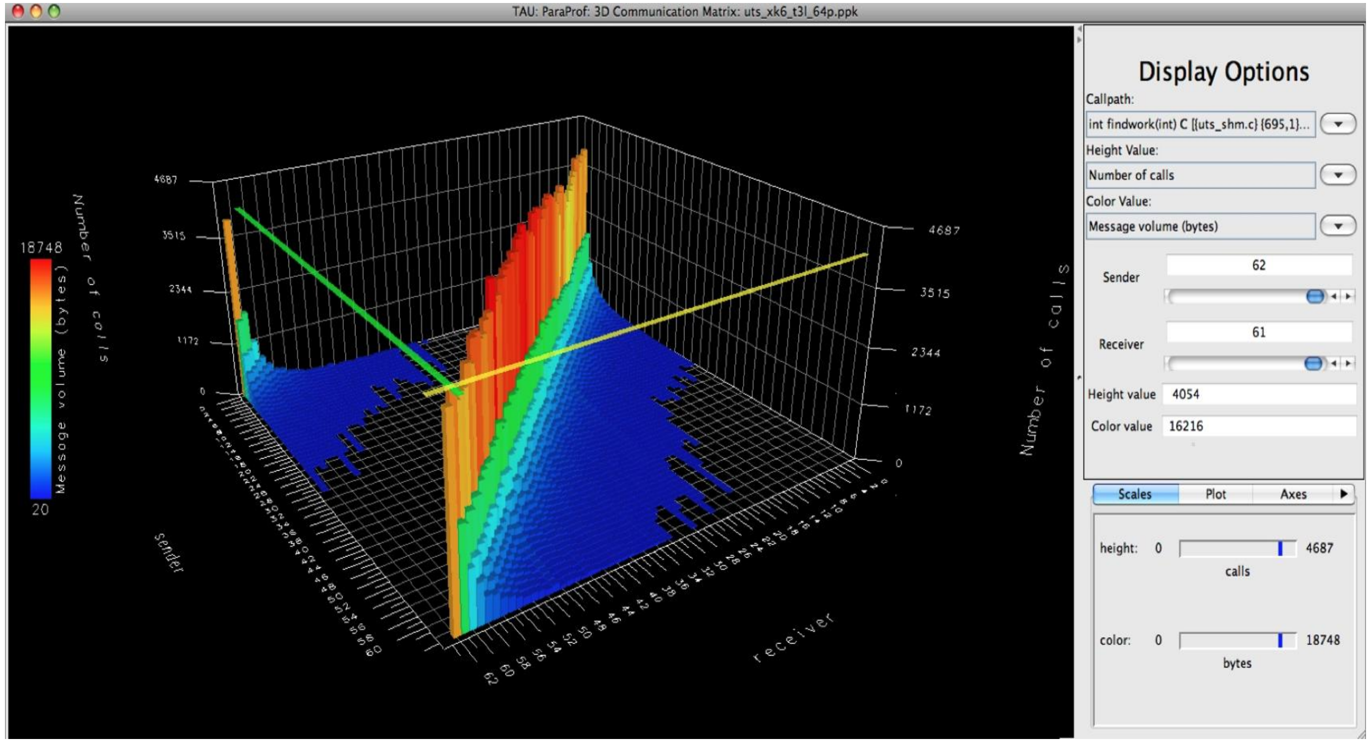
ParaProf Profile Analysis Framework



Parallel Profile Visualization: ParaProf



ParaProf 3D Communication Matrix



% export TAU_COMM_MATRIX=1

TAU tutorial exercise objectives

- Familiarise with usage of TAU tools
  - complementary tools' capabilities & interoperability
- Prepare to apply tools productively to *your* applications(s)
- Exercise is based on a small portable benchmark code
  - unlikely to have significant optimisation opportunities

- Optional (recommended) exercise extensions
  - analyse performance of alternative configurations
  - investigate effectiveness of system-specific compiler/MPI optimisations and/or placement/binding/affinity capabilities
  - investigate scalability and analyse scalability limiters
  - compare performance on different HPC platforms
  - …

Local Installation (*Stampede, TACC*)

- Setup preferred program environment compilers
  - Default set Intel Compilers with Intel MPI
  - Generate profile files using Score-P

```
% . /home1/03529/tg828282/Tutorial/vihps.sh
% paraprof profile.cubex &
```

For PerfExplorer:

```
% wget http://tau.uoregon.edu/data.tgz; tar zxf data.tgz; cd data
% cat README
And follow the steps
```

NPB-MZ-MPI Suite

- The NAS Parallel Benchmark suite (MPI+OpenMP version)
  - Available from:

    http://www.nas.nasa.gov/Software/NPB
  - 3 benchmarks in Fortran77
  - Configurable for various sizes & classes
- Move into the NPB3.3-MZ-MPI root directory

```
% ls
bin/     common/  jobscript/  Makefile  README.install   SP-MZ/
BT-MZ/   config/  LU-MZ/      README    README.tutorial  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 a (tool-specific) “bin” subdirectory

NPB-MZ-MPI / BT: config/make.def

```
#                    SITE- AND/OR PLATFORM-SPECIFIC DEFINITIONS.
#
#---------------------------------------------------------------------------

#---------------------------------------------------------------------------
# Configured for generic MPI with GCC compiler
#---------------------------------------------------------------------------
#OPENMP  = -fopenmp      # GCC compiler
OPENMP = -openmp         # Intel compiler

...
#---------------------------------------------------------------------
# The Fortran compiler used for MPI programs
#---------------------------------------------------------------------
# MPIF77 = mpiifort # Intel compiler

# Alternative variant to perform instrumentation
MPIF77 = tau_f90.sh -tau_makefile=<path>/Makefile.tau-[options]

# PREP is a generic preposition macro for instrumentation preparation
#MPIF77 = $(PREP) mpif77 -f77=ifort
#MPIF77 = scorep …
...
```

Default (no instrumentation)

Uncomment TAU's compiler wrapper to do source instrumentation with TAU
Comment out Score-P wrapper

Building an NPB-MZ-MPI Benchmark

```
% make
  ===========================================
  =      NAS PARALLEL BENCHMARKS 3.3        =
  =      MPI+OpenMP Multi-Zone Versions     =
  =      F77                                =
  ===========================================

  To make a NAS multi-zone benchmark type

         make <benchmark-name> CLASS=<class> NPROCS=<nprocs>

  where <benchmark-name> is "bt-mz", "lu-mz", or "sp-mz"
        <class>          is "S", "W", "A" through "F"
        <nprocs>         is number of processes

  [...]

 ****************************************************************
 * Custom build configuration is specified in config/make.def   *
 * Suggested tutorial exercise configuration for HPC systems:   *
 *       make bt-mz CLASS=C NPROCS=32                             *
 ****************************************************************
```

- Type "make" for instructions

Building an NPB-MZ-MPI Benchmark

```
% make suite
make[1]: Entering directory `BT-MZ'
make[2]: Entering directory `sys'
cc  -o setparams setparams.c -lm
make[2]: Leaving directory `sys'
../sys/setparams bt-mz 32 C
make[2]: Entering directory `../BT-MZ'
tau_f90.sh -c  -O3 -g -openmp       bt.f
                                      […]
tau_f90.sh -c  -O3 -g -openmp       mpi_setup.f
cd ../common;  mpiifort -c  -O3 -g -openmp        print_results.f
cd ../common;  mpiifort -c  -O3 -g -openmp        timers.f
tau_f90.sh -O3 -g -openmp   -o ../bin.tau/bt-mz_C.8 bt.o
 initialize.o exact_solution.o exact_rhs.o set_constants.o adi.o
 rhs.o zone_setup.o x_solve.o y_solve.o  exch_qbc.o solve_subs.o
 z_solve.o add.o error.o verify.o mpi_setup.o ../common/print_results.o
 ../common/timers.o
make[2]: Leaving directory `BT-MZ'
Built executable ../bin.tau/bt-mz_C.32
make[1]: Leaving directory `BT-MZ'
```

- Specify the benchmark configuration
  - benchmark name: **bt-mz**, lu-mz, sp-mz
  - the number of MPI processes: NPROCS=**3C**
  - the benchmark class (S, W, A, B, C, D, E): CLASS=**C**

Shortcut: `% make suite`

NPB-MZ-MPI / BT (Block Tridiagonal Solver)

- What does it do?
  - Solves a discretized version of the unsteady, compressible Navier-Stokes equations in three spatial dimensions
  - Performs 200 time-steps on a regular 3-dimensional grid
- Implemented in 20 or so Fortran77 source modules

- Uses MPI & OpenMP in combination
  - 2 compute nodes with 1 Intel Xeon Phi 7250 CPU (Knights Landing, KNL) each
  - 32 processes each with 4 OpenMP threads should be reasonable

  - bt-mz_C.32 should take around 30 seconds

TAU Source Instrumentation

- Edit config/make.def to adjust build configuration
  - Uncomment specification of compiler/linker: MPIF77 = tau_f77.sh
- Make clean and build new tool-specific executable

```
% make clean
% make bt-mz CLASS=C NPROCS=32
Built executable ../bin.tau/bt-mz_C.32
```

- Change to the directory containing the new executable before running it with the desired tool configuration

```
% cd bin.tau
% cp ../jobscript/stampede2/tau.sbatch .
% sbatch tau.sbatch
```

NPB-MZ-MPI / BT with TAU

```
% cd bin
% cp ../jobscript/stampede2/tau.sbatch .
% sbatch tau.sbatch
% cat mzmpibt.o<job_id>
 NAS Parallel Benchmarks (NPB3.3-MZ-MPI) - BT-MZ MPI+OpenMP Benchmark
 Number of zones:   16 x 16
 Iterations:  200    dt:   0.000300
 Number of active processes:     32
 Total number of threads:       128 (  4.0 threads/process)

 Time step    1
 Time step   20
  [...]
 Time step  180
 Time step  200
 Verification Successful

 BT-MZ Benchmark Completed.
 Time in seconds = 22.34
% paraprof &
% paraprof --pack bt.ppk
<Copy file over to desktop using scp>
% paraprof bt.ppk  &
```

- Copy jobscript and launch as a hybrid MPI+OpenMP application

Hint: save the benchmark output (or note the run time) to be able to refer to it later

tau_exec

```
$ tau_exec

Usage: tau_exec [options] [--] <exe> <exe options>

Options:
        -v             Verbose mode
        -s             Show what will be done but don't actually do anything (dryrun)
        -qsub          Use qsub mode (BG/P only, see below)
        -io            Track I/O
        -memory        Track memory allocation/deallocation
        -memory_debug  Enable memory debugger
        -cuda          Track GPU events via CUDA
        -cupti         Track GPU events via CUPTI (Also see env. variable TAU_CUPTI_API)
        -opencl        Track GPU events via OpenCL
        -openacc       Track GPU events via OpenACC (currently PGI only)
        -ompt          Track OpenMP events via OMPT interface
        -armci         Track ARMCI events via PARMCI
        -ebs           Enable event-based sampling
        -ebs_period=<count> Sampling period (default 1000)
        -ebs_source=<counter> Counter (default itimer)
        -um           Enable Unified Memory events via CUPTI
        -T <DISABLE,GNU,ICPC,MPI,OMPT,OPENMP,PAPI,PDT,PROFILE,PTHREAD,SCOREP,SERIAL> : Specify TAU tags
        -loadlib=<file.so>   : Specify additional load library
        -XrunTAUsh-<options> : Specify TAU library directly
        -gdb           Run program in the gdb debugger

Notes:
           Defaults if unspecified: -T MPI
           MPI is assumed unless SERIAL is specified
```

- Tau_exec preloads the TAU wrapper libraries and performs measurements.

No need to recompile the application!

tau_exec Example (continued)

```
Example:
    mpirun -np 2 tau_exec -T icpc,ompt,mpi  -ompt ./a.out
    mpirun -np 2 tau_exec -io ./a.out
Example - event-based sampling with samples taken every 1,000,000 FP instructions
    mpirun -np 8 tau_exec -ebs -ebs_period=1000000 -ebs_source=PAPI_FP_INS ./ring
Examples - GPU:
    tau_exec -T serial,cupti -cupti ./matmult (Preferred for CUDA 4.1 or later)
    tau_exec -openacc ./a.out
    tau_exec -T serial -opencl ./a.out (OPENCL)
    mpirun -np 2 tau_exec -T mpi,cupti,papi -cupti -um ./a.out (Unified Virtual Memory in CUDA 6.0+)

qsub mode (IBM BG/Q only):
    Original:
      qsub -n 1 --mode smp -t 10 ./a.out
    With TAU:
      tau_exec -qsub -io -memory -- qsub -n 1 … -t 10 ./a.out

Memory Debugging:
    -memory option:
      Tracks heap allocation/deallocation and memory leaks.
    -memory_debug option:
      Detects memory leaks, checks for invalid alignment, and checks for
      array overflow.  This is exactly like setting TAU_TRACK_MEMORY_LEAKS=1
      and TAU_MEMDBG_PROTECT_ABOVE=1 and running with -memory
```

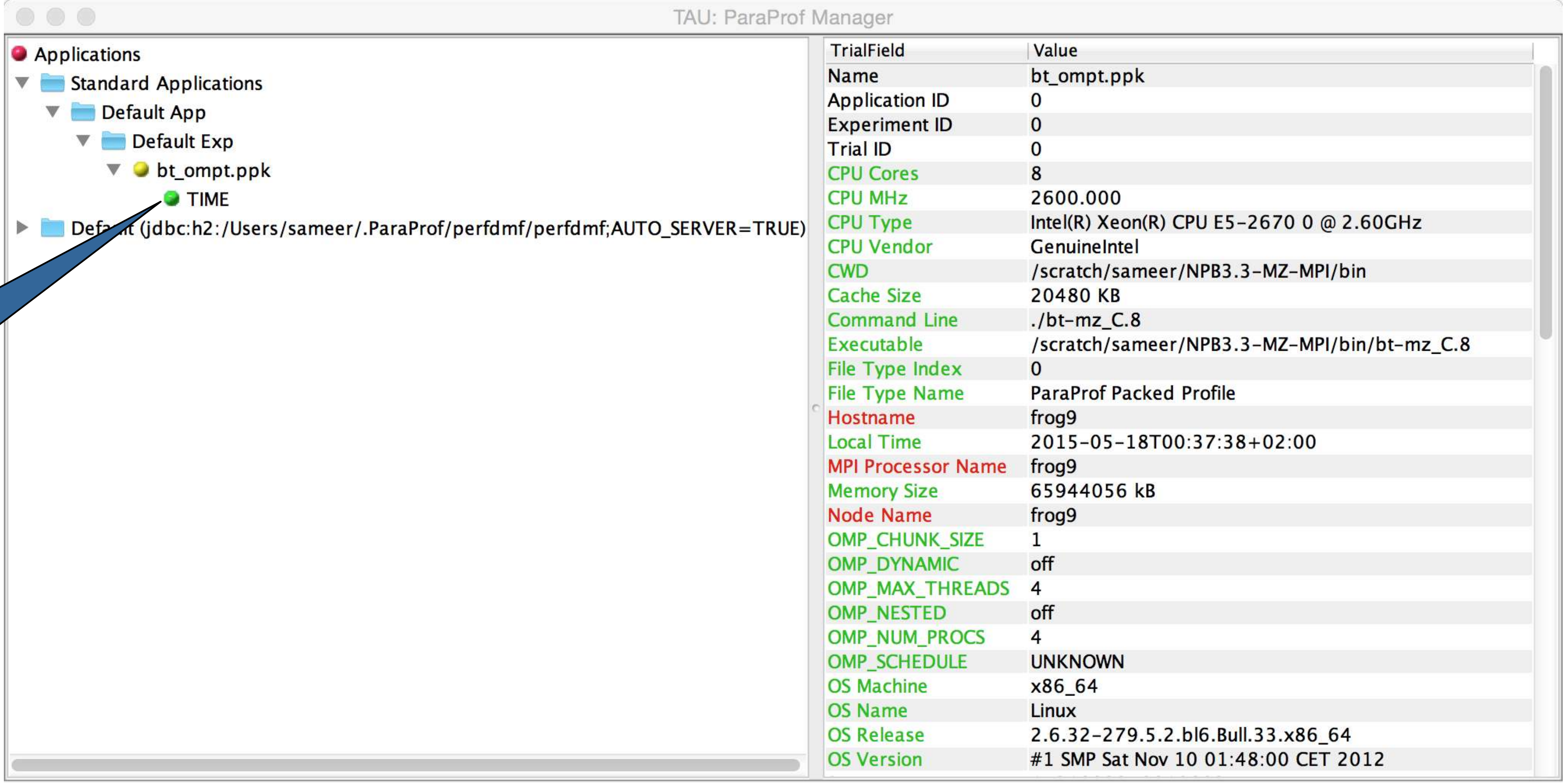
- tau_exec can enable event based sampling while launching the executable using the –ebs flag!
- On stampede, you need to put perl-mic/bin in your path
- ibrun.symm –m test.sh
- Within test.sh call tau_exec –T ompt

TAU Analysis Tools: paraprof

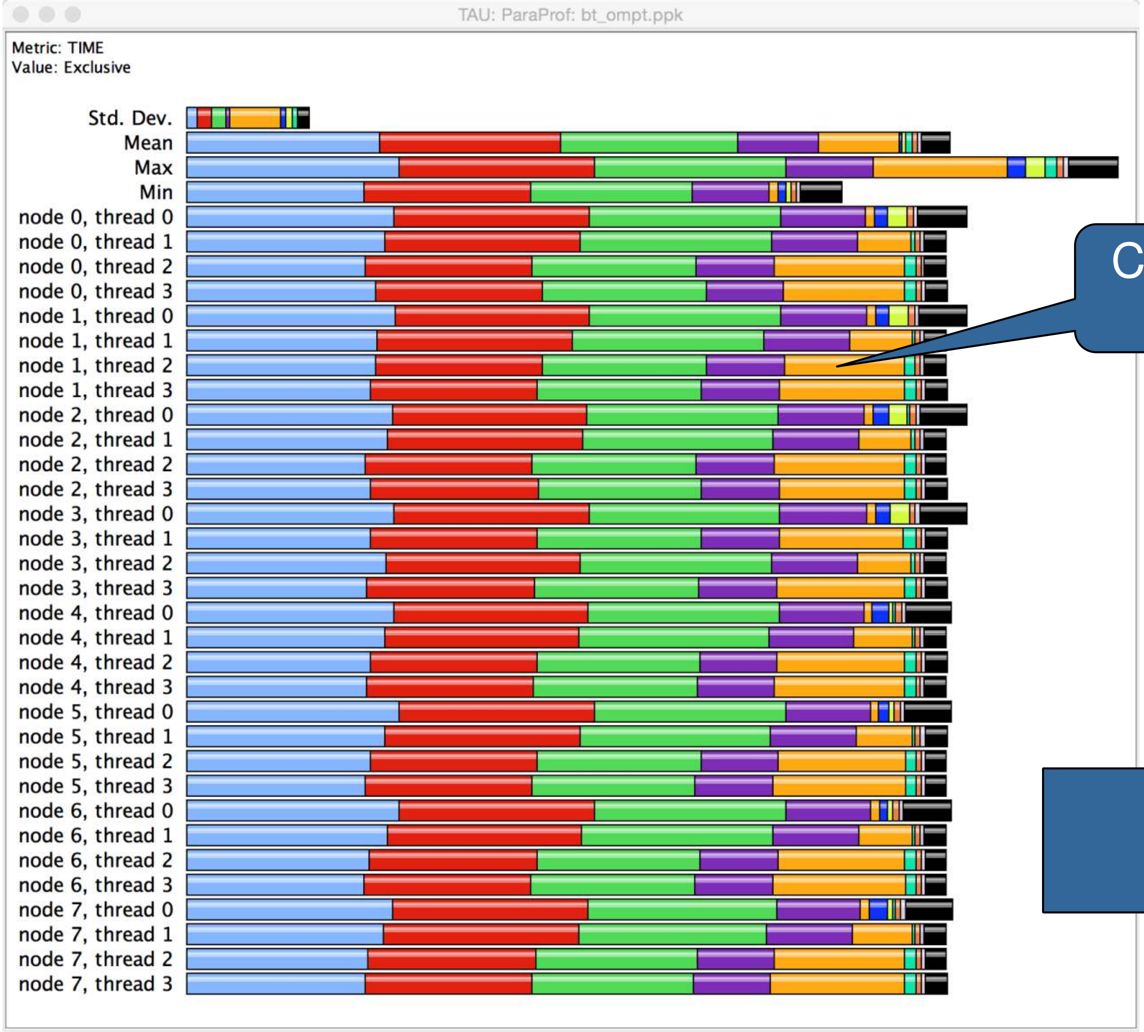
- Launch paraprof

```
% paraprof
```

Metric



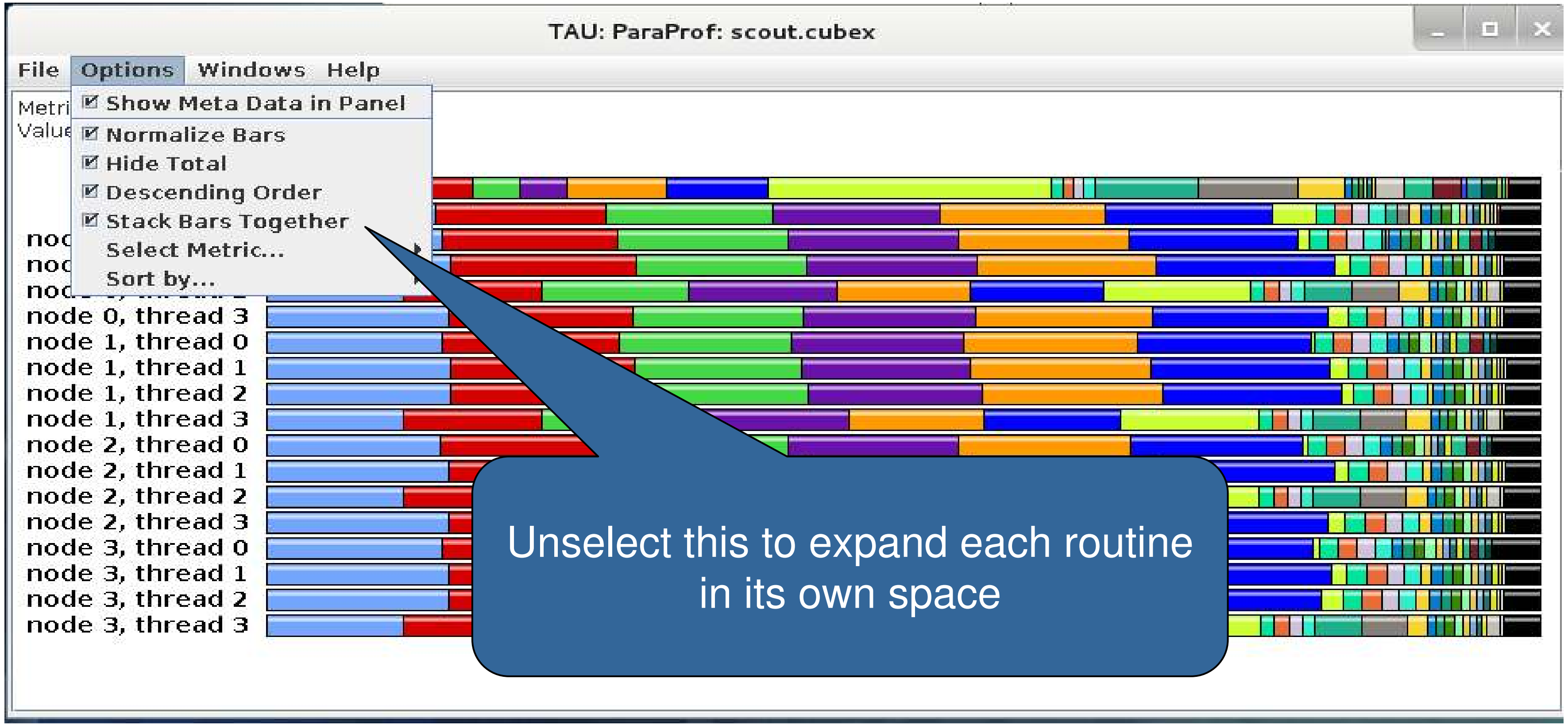
Paraprof main window



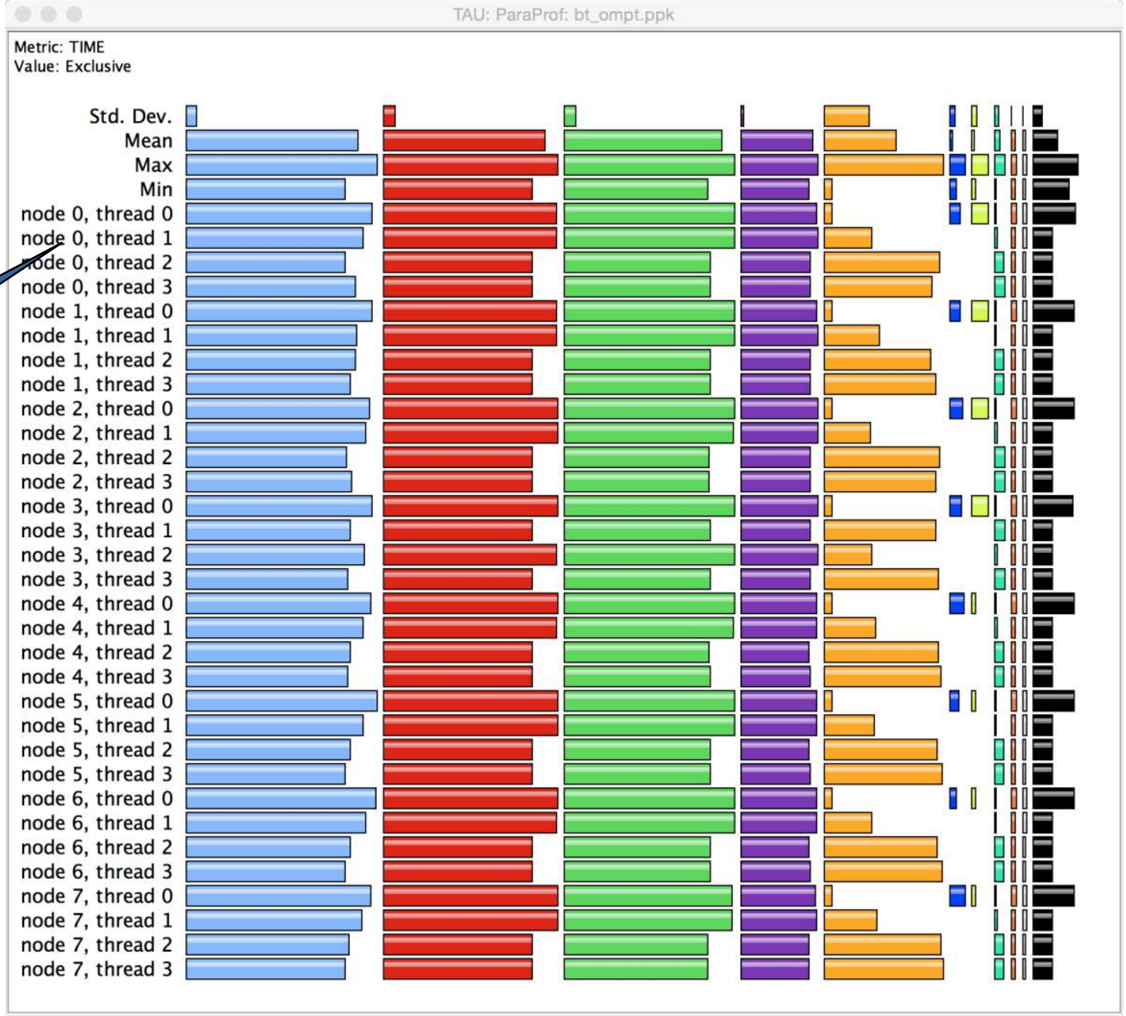
Colors represent code regions

Options -> uncheck Stack Bars Together

Paraprof main window



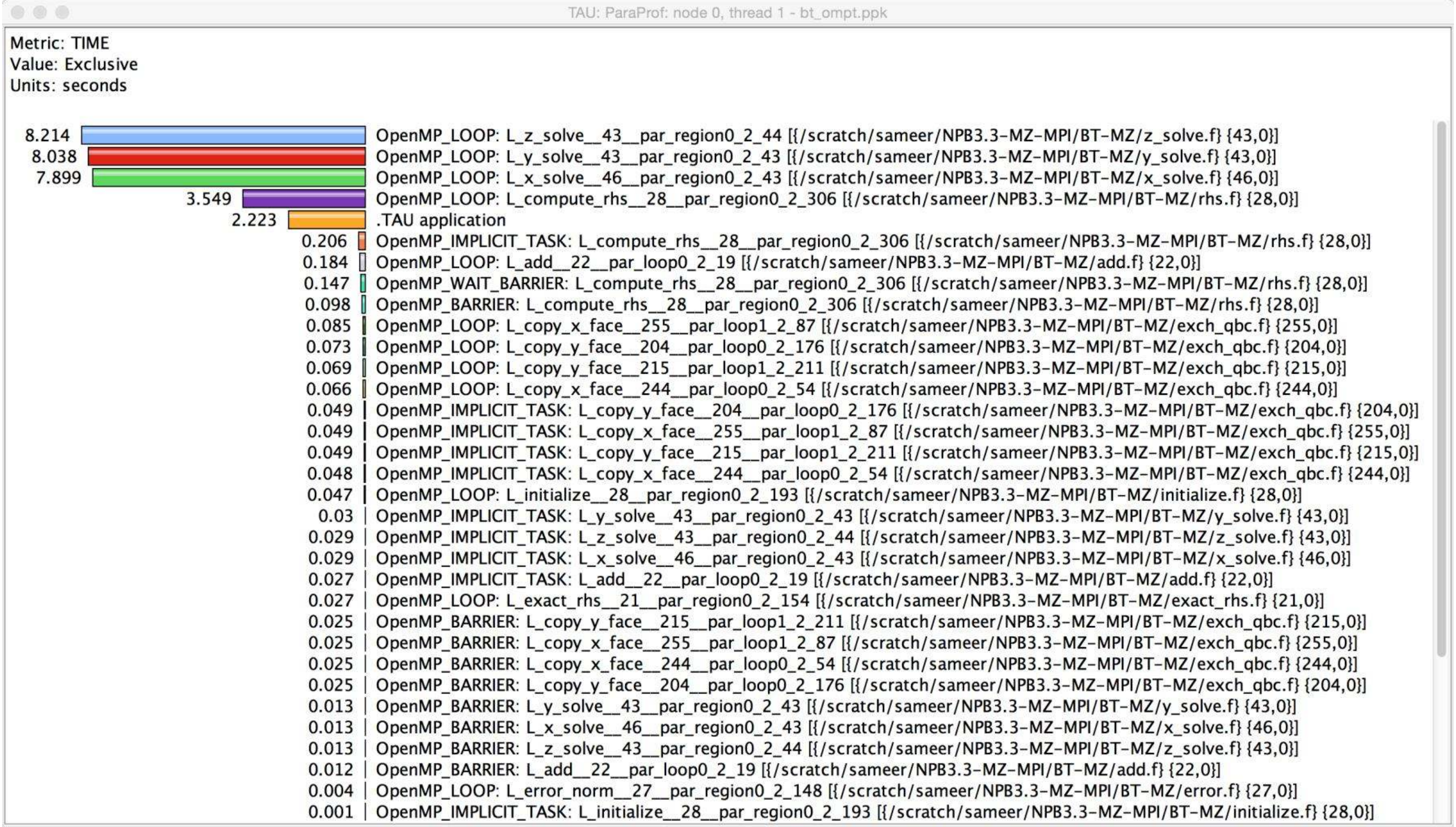
Paraprof main window



Left/right click here

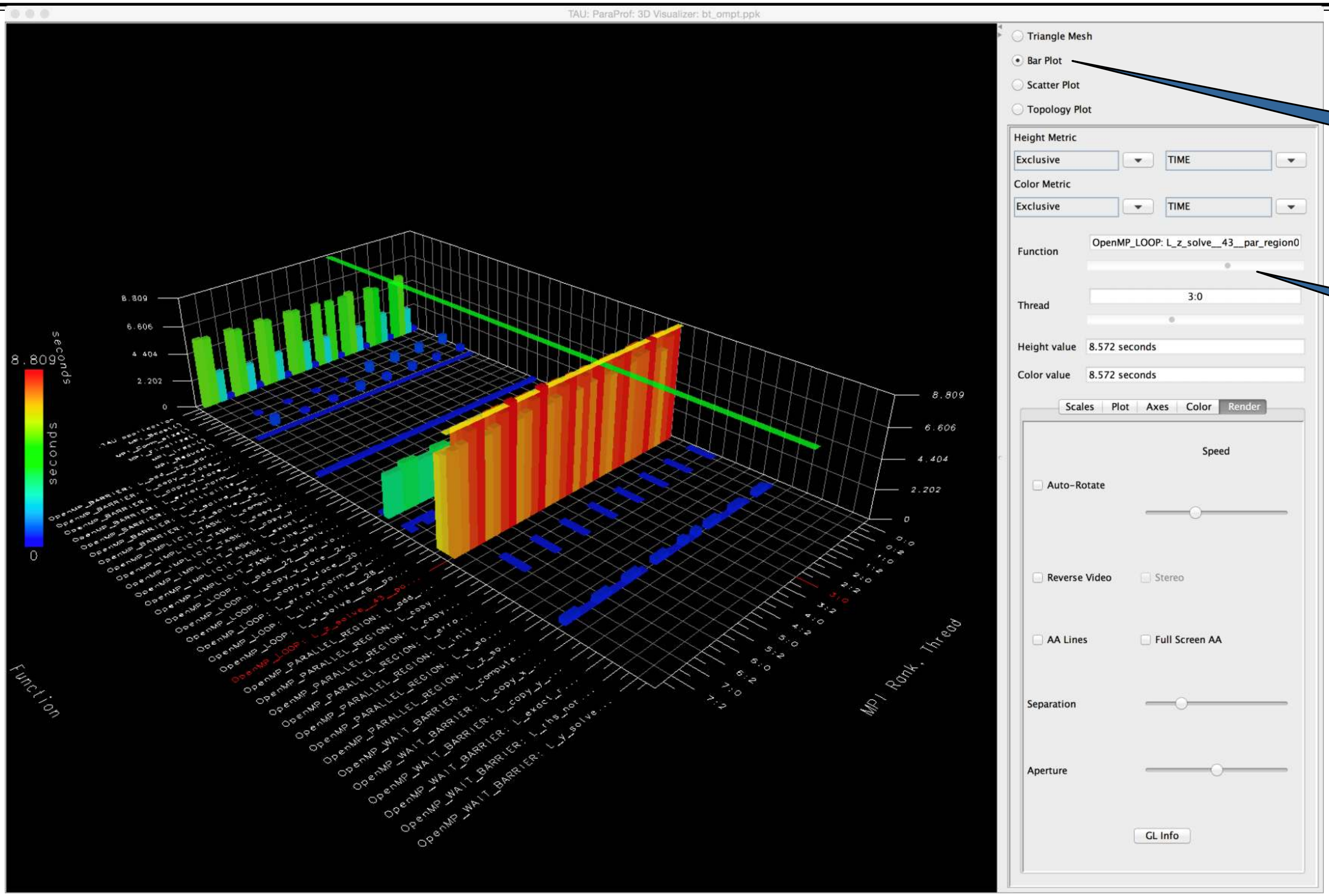
Each routine occupies its own space. Can see the extent of imbalance across all threads.

Paraprof node window (function barchart window)



Exclusive time spent in each code region (OpenMP loop) is shown here for MPI rank 0 thread 1

Paraprof 3D visualization window

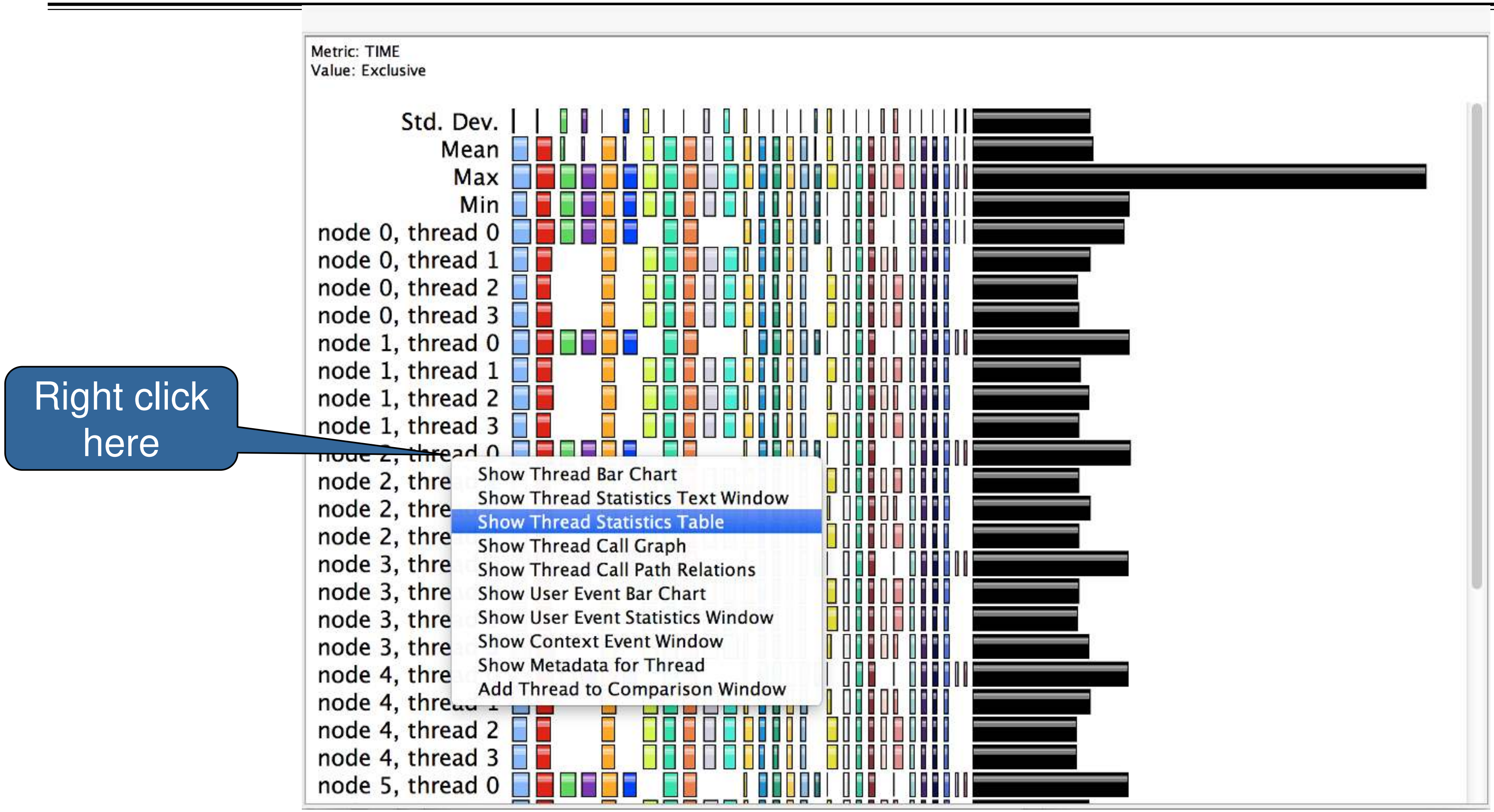


Click Bar Plot

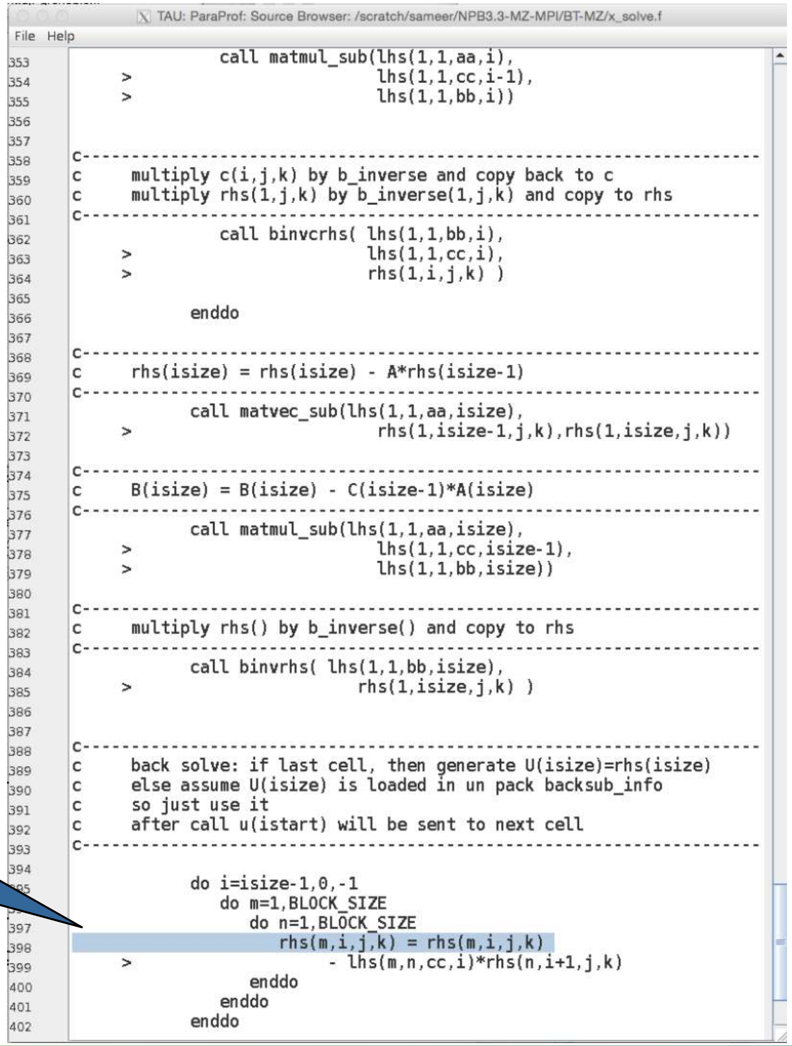
Move Function and Thread Sliders

Windows -> 3D visualization

Paraprof Thread Statistics Table with TAU_SAMPLING=1



Statement Level Profiling with TAU



Source location where samples are taken. Compute intensive region.

Paraprof Thread Statistics Table

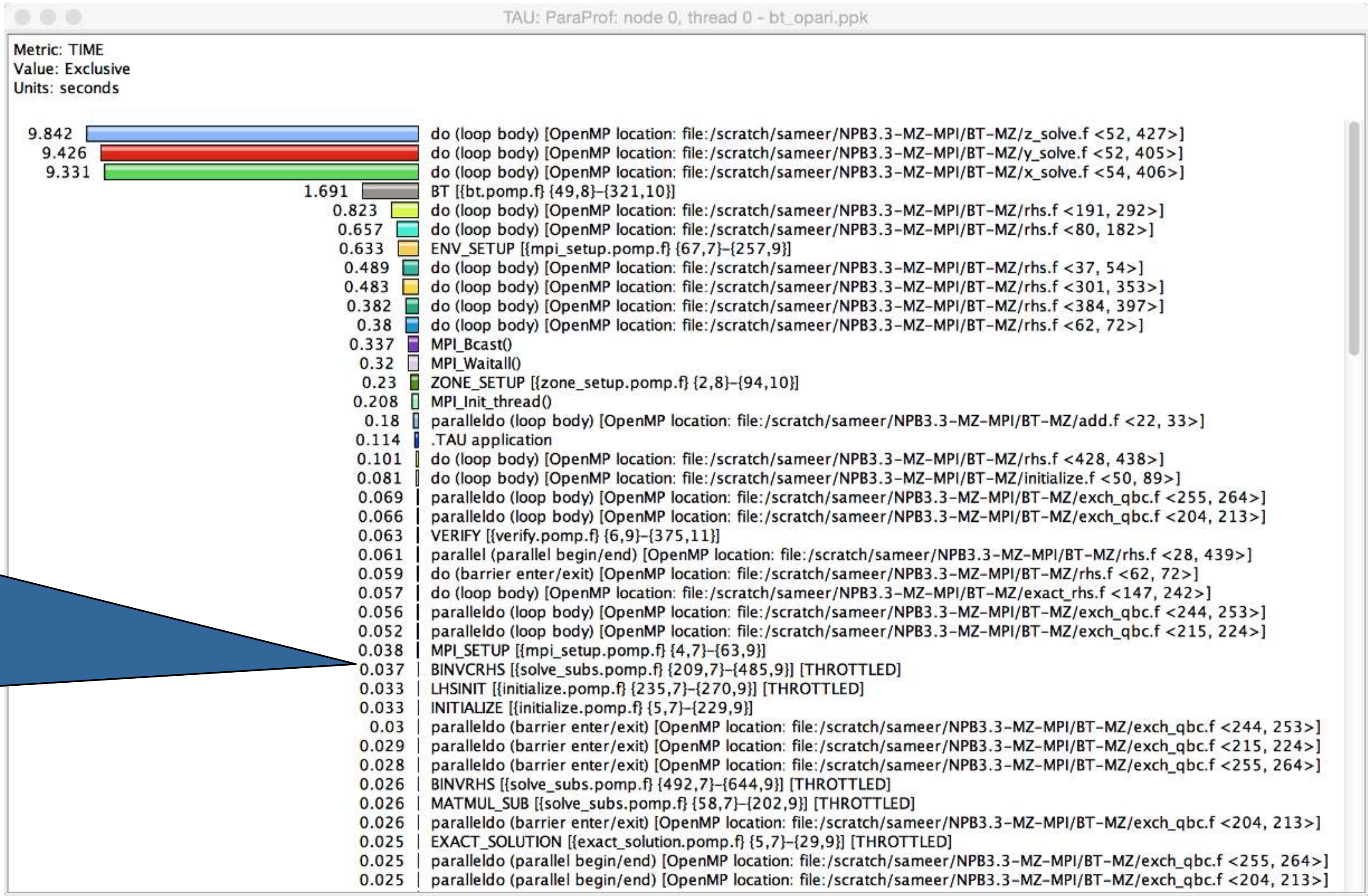
TAU: ParaProf: Statistics for: node 2, thread 0 - bt_ebs.ppk

Name	Exclusive TIME	Inclusive TIME	Calls	Child Calls
.TAU application	1.754	36.26	1	88,049
OpenMP_PARALLEL_REGION: L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f} {43,0}]	0.061	8.692	6,432	12,864
OpenMP_IMPLICIT_TASK: L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f} {43,0}]	0.04	8.568	6,432	6,432
OpenMP_LOOP: L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f} {43,0}]	8.528	8.528	6,432	0
[CONTEXT] OpenMP_LOOP: L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f} {43,0}]	0	9.23	847	0
[SUMMARY] L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f}]	3.67	3.67	340	0
[SAMPLE] L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f}]	3.67	3.67	340	0
[SAMPLE] L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f} {419}]	0.22	0.22	21	0
[SAMPLE] L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f} {58}]	0.17	0.17	16	0
[SAMPLE] L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f} {418}]	0.16	0.16	12	0
[SAMPLE] L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f} {123}]	0.11	0.11	11	0
[SAMPLE] L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f} {193}]	0.08	0.08	5	0
[SAMPLE] L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f} {126}]	0.07	0.07	7	0
[SAMPLE] L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f} {247}]	0.07	0.07	6	0
[SAMPLE] L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f} {158}]	0.06	0.06	5	0
[SAMPLE] L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f} {313}]	0.06	0.06	4	0
[SAMPLE] L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f} {230}]	0.06	0.06	4	0
[SAMPLE] L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f} {308}]	0.05	0.05	3	0
[SAMPLE] L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f} {191}]	0.05	0.05	3	0
[SAMPLE] L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f} {81}]	0.05	0.05	4	0
[SAMPLE] L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f} {301}]	0.05	0.05	5	0
[SAMPLE] L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f} {67}]	0.05	0.05	5	0
[SAMPLE] L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f} {175}]	0.04	0.04	4	0
[SAMPLE] L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f} {89}]	0.04	0.04	4	0
[SAMPLE] L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f} {55}]	0.04	0.04	4	0
[SAMPLE] L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f} {275}]	0.04	0.04	4	0
[SAMPLE] L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f} {129}]	0.04	0.04	4	0
[SAMPLE] L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f} {168}]	0.04	0.04	4	0
[SAMPLE] L_z_solve__43__par_region0_2_44 [{/scratch/sameer/NPB3.3-MZ-MPI/BT-MZ/z_solve.f} {238}]	0.04	0.04	4	0

Show Source Code
Show Function Bar Chart
Show Function Histogram
Assign Function Color
Reset to Default Color

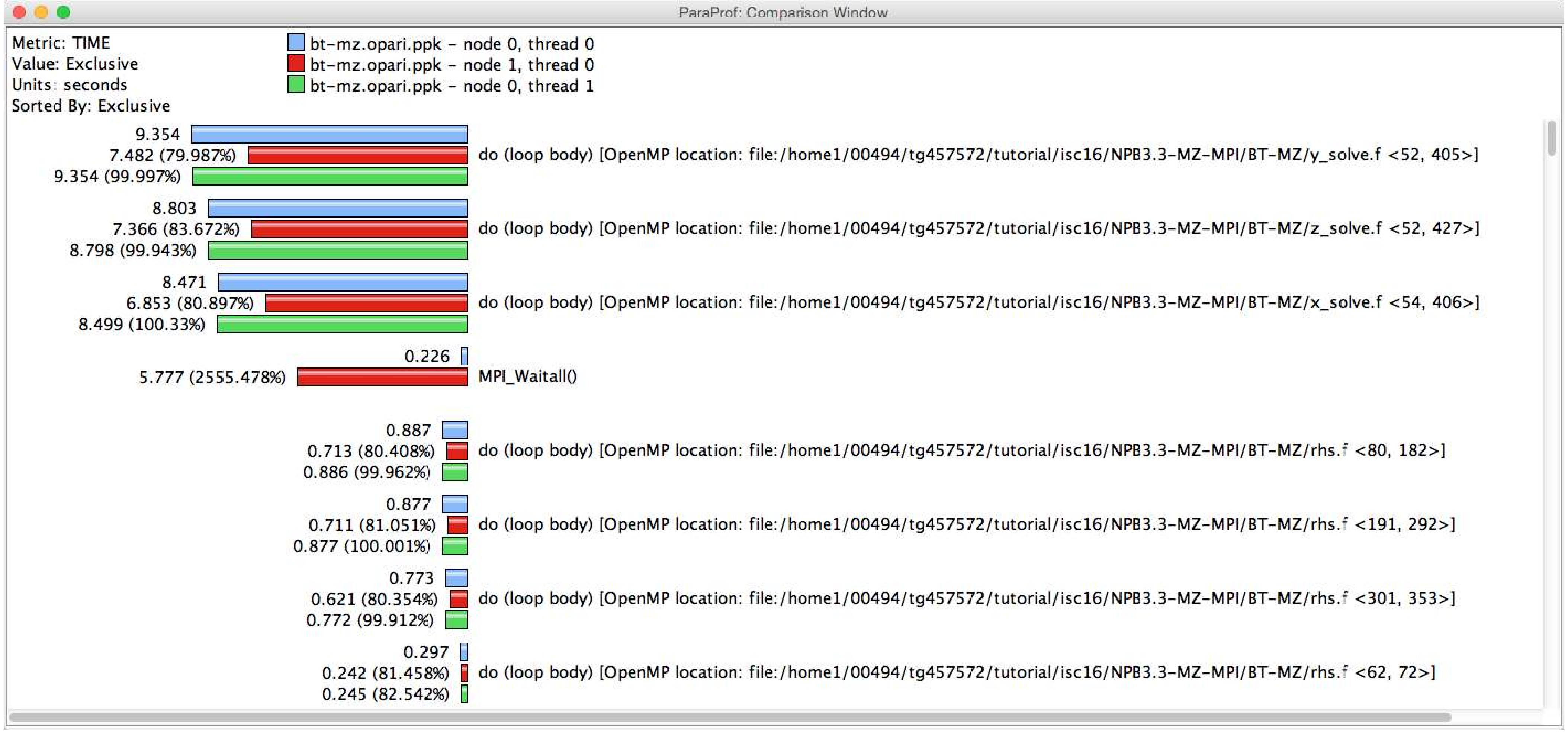
Right click here and choose "Show Source Code" for a sample

Instrumenting Source Code with PDT and Opari

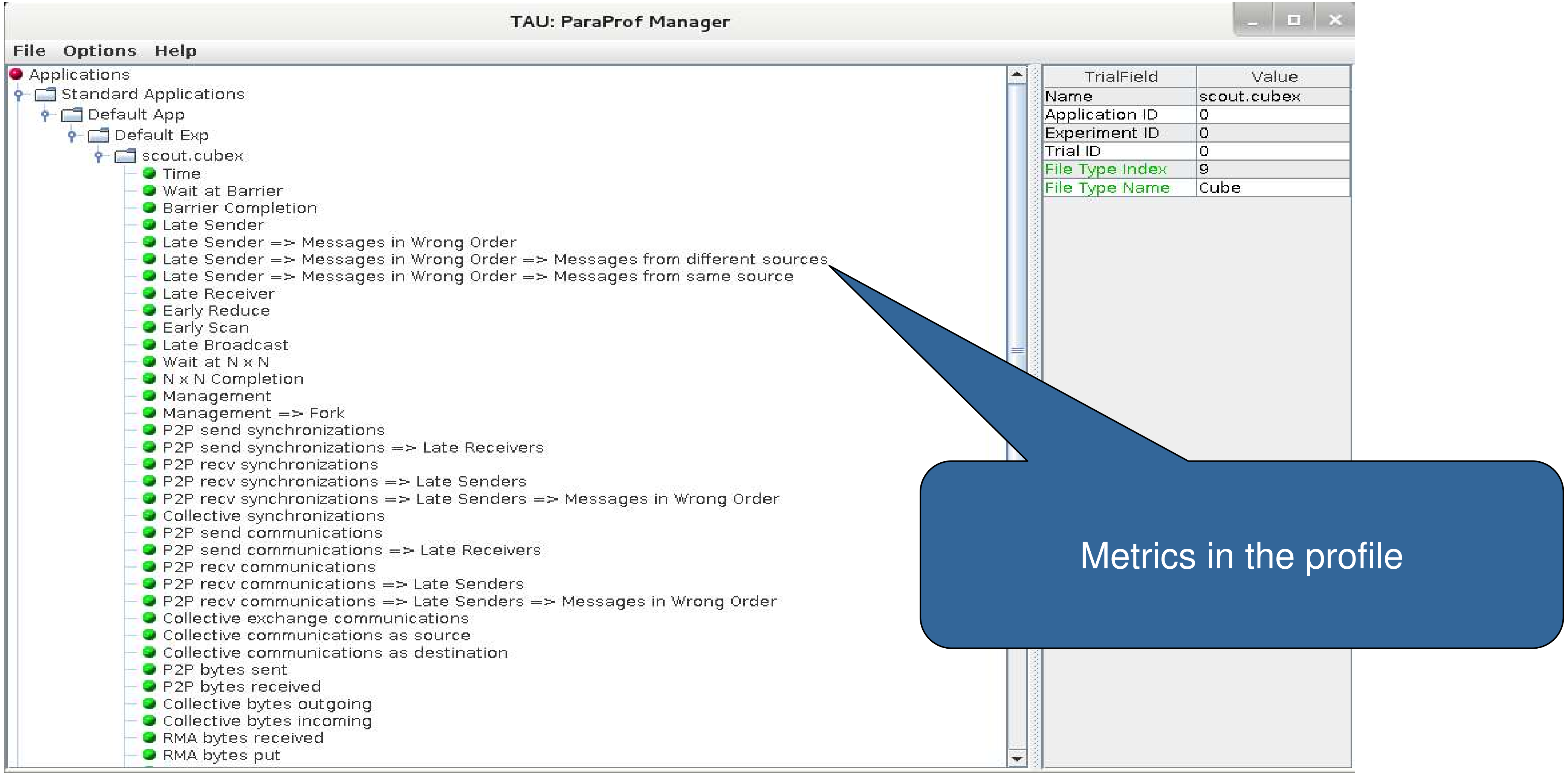


Frequently executing lightweight routines are automatically throttled at runtime. Reduces runtime dilation.

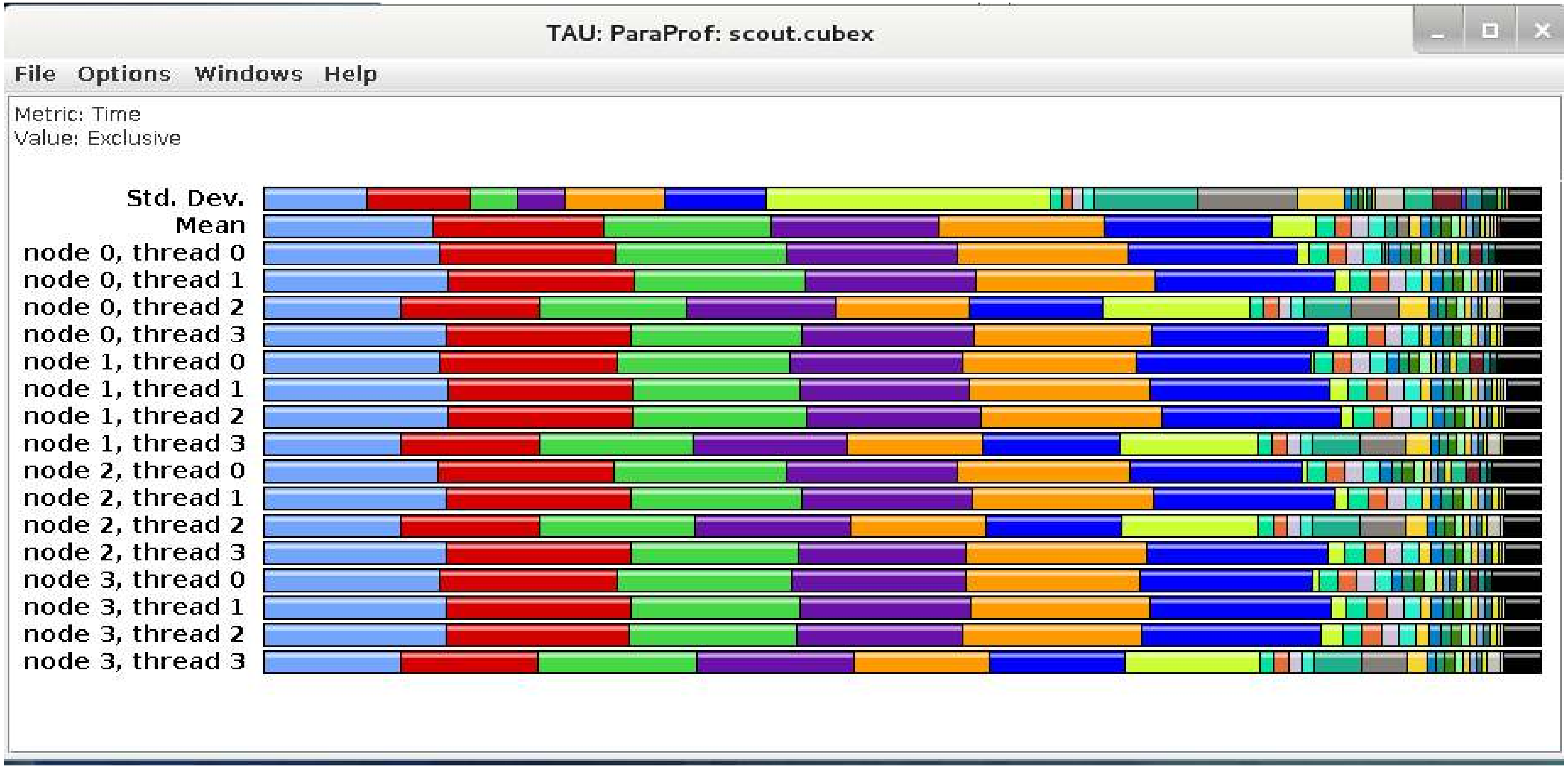
ParaProf Comparison Window



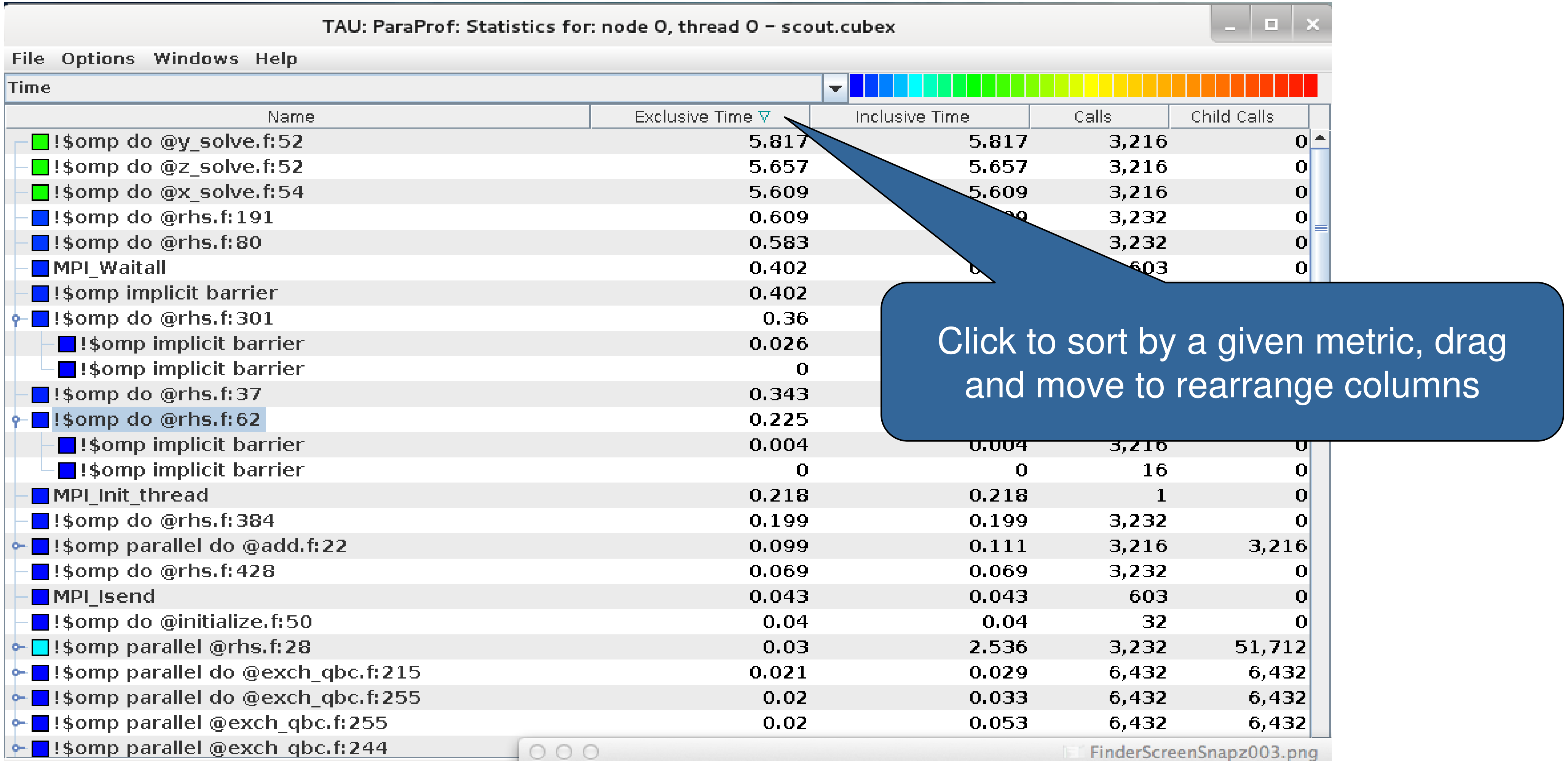
ParaProf Manager Widow: scout.cubex



ParaProf: Main Window



ParaProf: Thread Statistics Table



Name	Exclusive Time ▽	Inclusive Time	Calls	Child Calls
!$omp do @y_solve.f:52	5.817	5.817	3,216	0
!$omp do @z_solve.f:52	5.657	5.657	3,216	0
!$omp do @x_solve.f:54	5.609	5.609	3,216	0
!$omp do @rhs.f:191	0.609	[illegible]	3,232	0
!$omp do @rhs.f:80	0.583	[illegible]	3,232	0
MPI_Waitall	0.402	[illegible]	[illegible]	0
!$omp implicit barrier	0.402	[illegible]	[illegible]	[illegible]
!$omp do @rhs.f:301	0.36	[illegible]	[illegible]	[illegible]
!$omp implicit barrier	0.026	[illegible]	[illegible]	[illegible]
!$omp implicit barrier	0	[illegible]	[illegible]	[illegible]
!$omp do @rhs.f:37	0.343	[illegible]	[illegible]	[illegible]
!$omp do @rhs.f:62	0.225	[illegible]	[illegible]	[illegible]
!$omp implicit barrier	0.004	0.004	3,216	0
!$omp implicit barrier	0	0	16	0
MPI_Init_thread	0.218	0.218	1	0
!$omp do @rhs.f:384	0.199	0.199	3,232	0
!$omp parallel do @add.f:22	0.099	0.111	3,216	3,216
!$omp do @rhs.f:428	0.069	0.069	3,232	0
MPI_Isend	0.043	0.043	603	0
!$omp do @initialize.f:50	0.04	0.04	32	0
!$omp parallel @rhs.f:28	0.03	2.536	3,232	51,712
!$omp parallel do @exch_qbc.f:215	0.021	0.029	6,432	6,432
!$omp parallel do @exch_qbc.f:255	0.02	0.033	6,432	6,432
!$omp parallel @exch_qbc.f:255	0.02	0.053	6,432	6,432
!$omp parallel @exch_qbc.f:244				

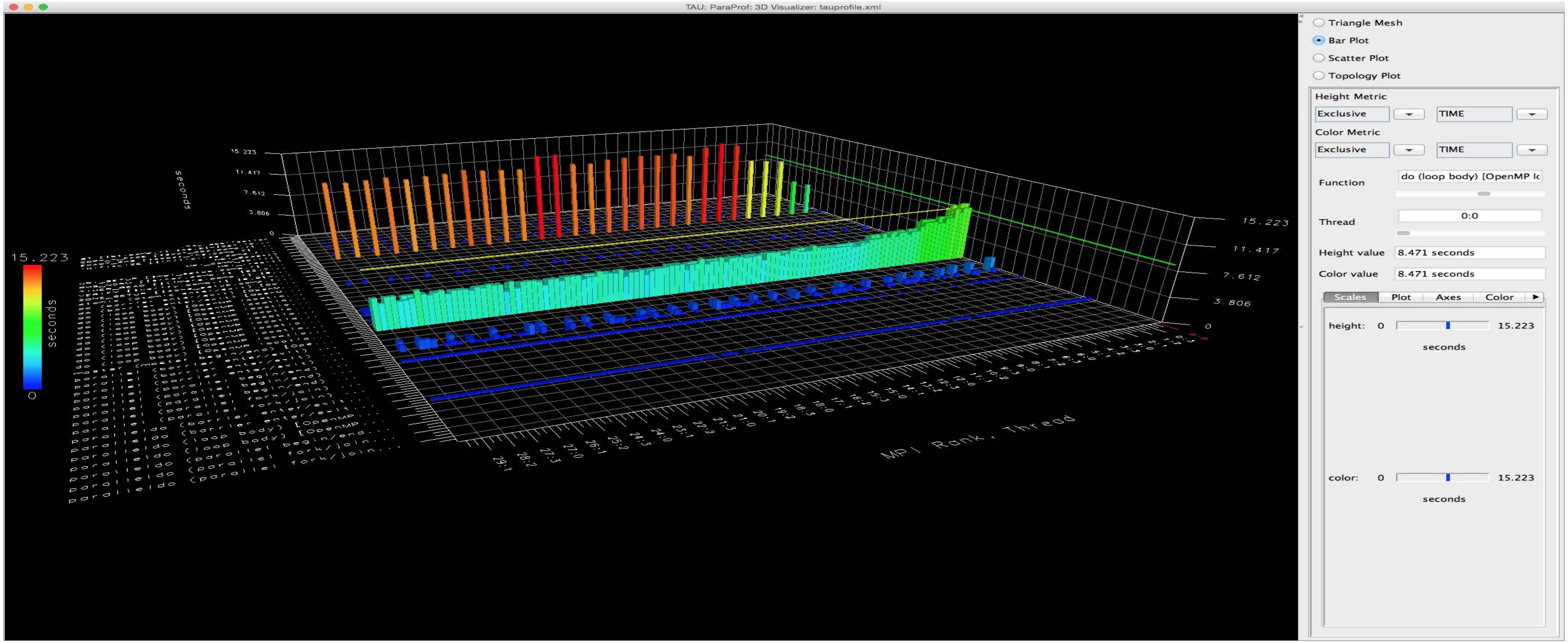
ParaProf: Callpath Thread Relations Window

TAU: ParaProf: Call Path Data n,c,t, 0,0,0 - scout.cubex

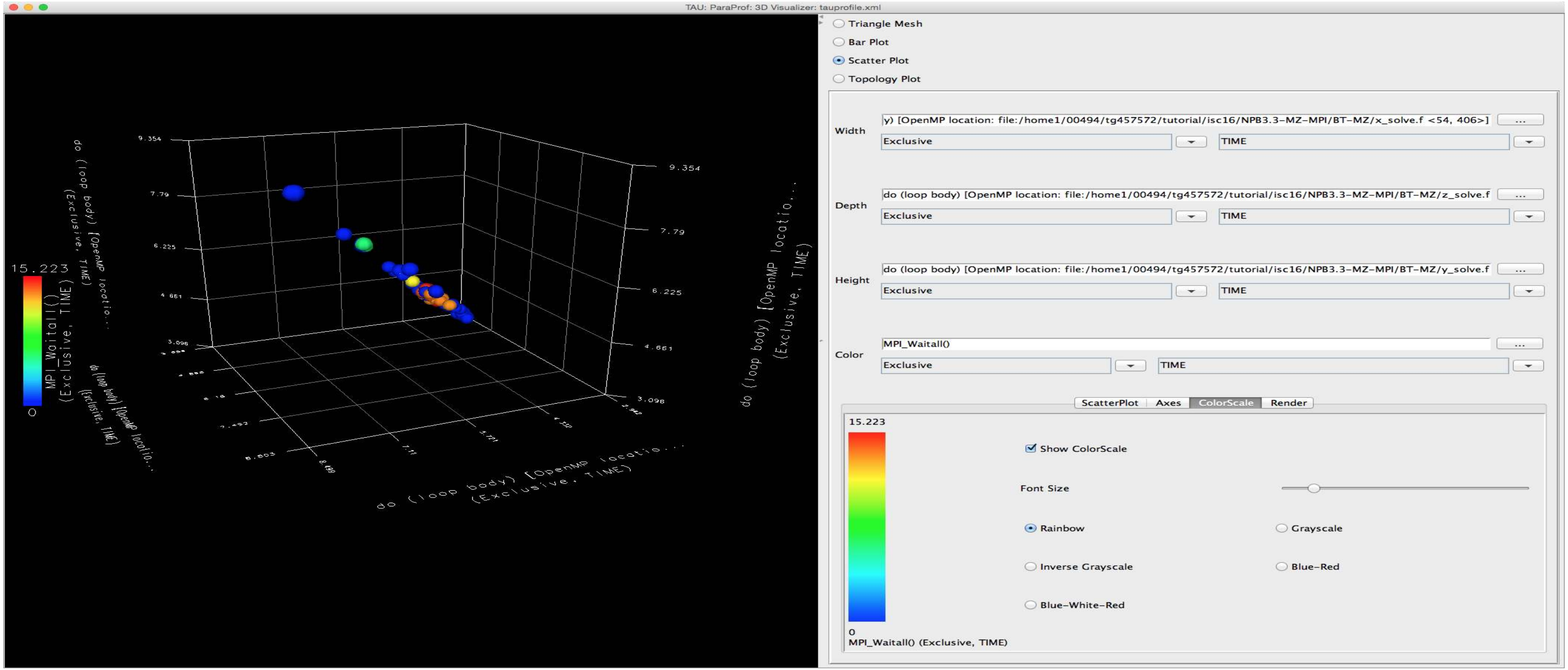
File Options Windows Help

```
Metric Name: Time
Sorted By: Exclusive
Units: seconds


          0.04            0.04     32/32             !$omp parallel @initialize.f:28
  -->     0.04            0.04     32                !$omp do @initialize.f:50


          0.03           2.536     3232/3232         compute_rhs_
  -->     0.03           2.536     3232              !$omp parallel @rhs.f:28
        9.8E-4          9.8E-4     3232/3232         !$omp master @rhs.f:424
         0.225           0.228     3232/3232         !$omp do @rhs.f:62
         0.002           0.002     3232/3232         !$omp master @rhs.f:74
         0.002           0.002     3232/3232         !$omp master @rhs.f:293
         0.199           0.199     3232/3232         !$omp do @rhs.f:384
         0.002           0.002     3232/3232         !$omp master @rhs.f:183
         0.343           0.343     3232/3232         !$omp do @rhs.f:37
         0.016           0.016     3232/3232         !$omp do @rhs.f:372
         0.014           0.027     3232/3232         !$omp do @rhs.f:413
         0.609           0.609     3232/3232         !$omp do @rhs.f:191
          0.36           0.386     3232/3232         !$omp do @rhs.f:301
         0.583           0.583     3232/3232         !$omp do @rhs.f:80
         0.019           0.019     3232/3232         !$omp do @rhs.f:400
         0.006           0.006     3232/51680        !$omp implicit barrier
         0.069           0.069     3232/3232         !$omp do @rhs.f:428
         0.015           0.015     3232/3232         !$omp do @rhs.f:359


         0.021           0.029     6432/6432         !$omp parallel @exch_qbc.f:215
  -->    0.021           0.029     6432              !$omp parallel do @exch_qbc.f:215
         0.007           0.007     6432/51680        !$omp implicit barrier


          0.02           0.033     6432/6432         !$omp parallel @exch_qbc.f:255
  -->     0.02           0.033     6432              !$omp parallel do @exch_qbc.f:255
         0.013           0.013     6432/51680        !$omp implicit barrier
```

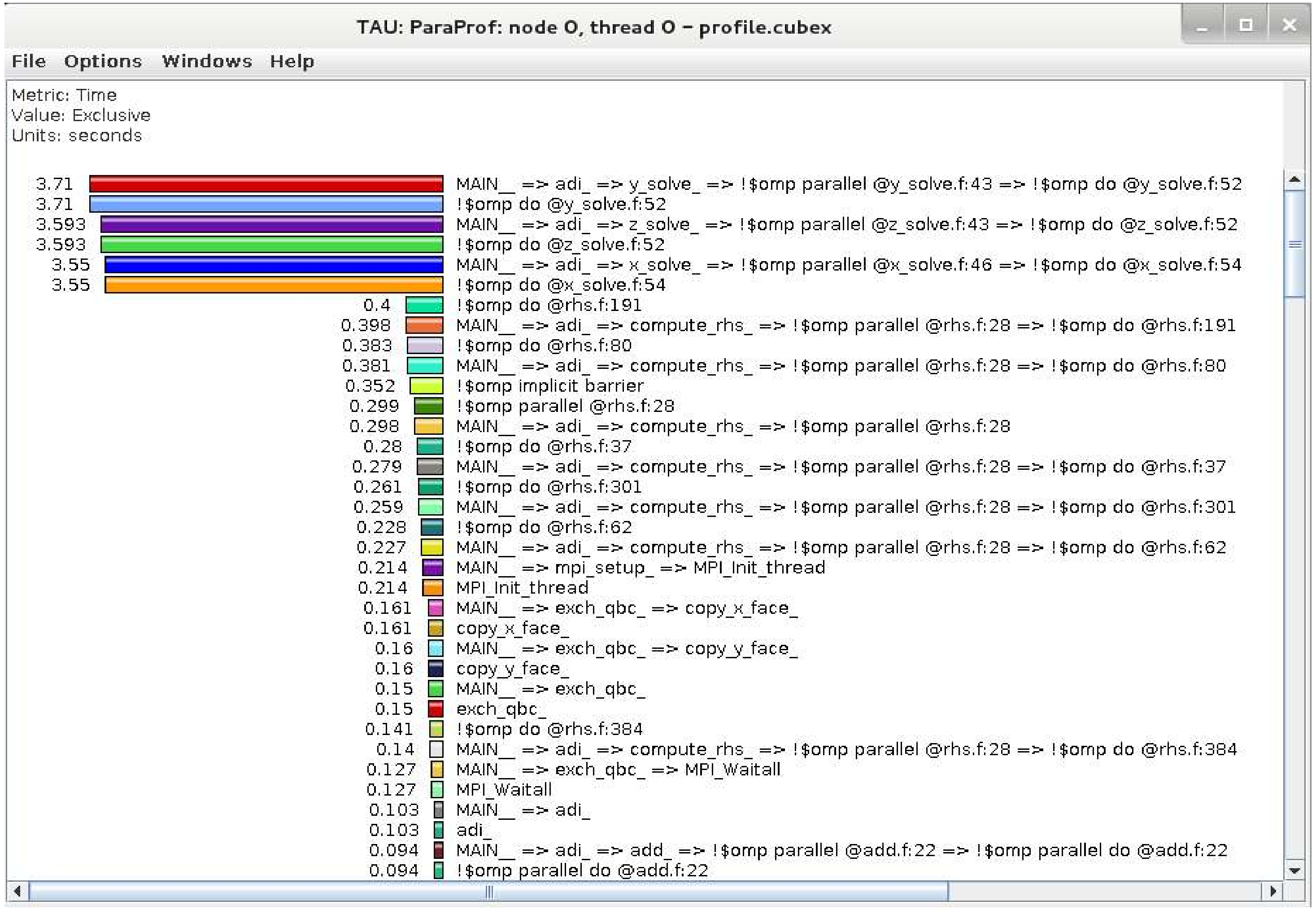
ParaProf: 3D Visualization Window Showing Entire Profile



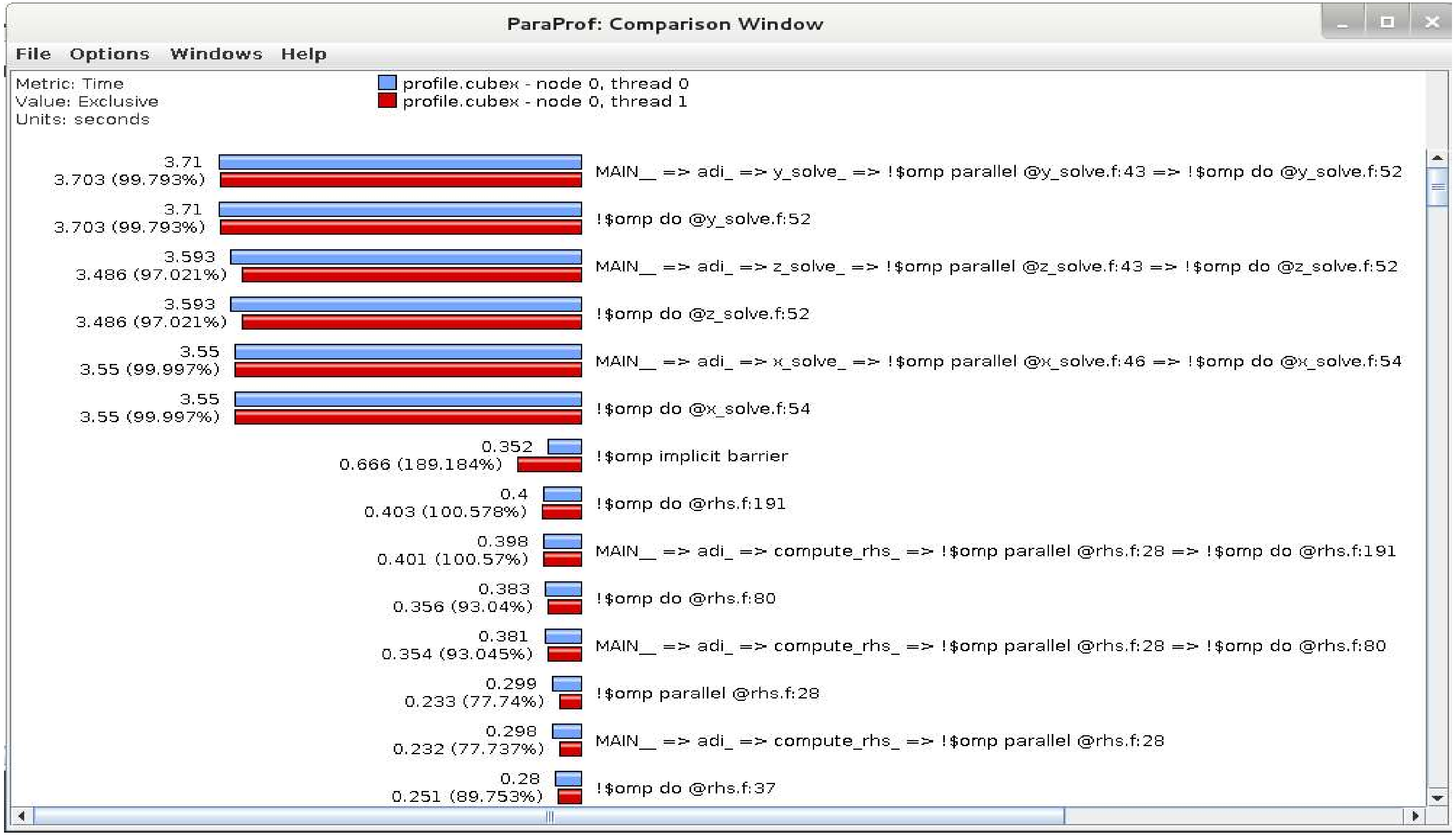
ParaProf: 3D Scatter Plot



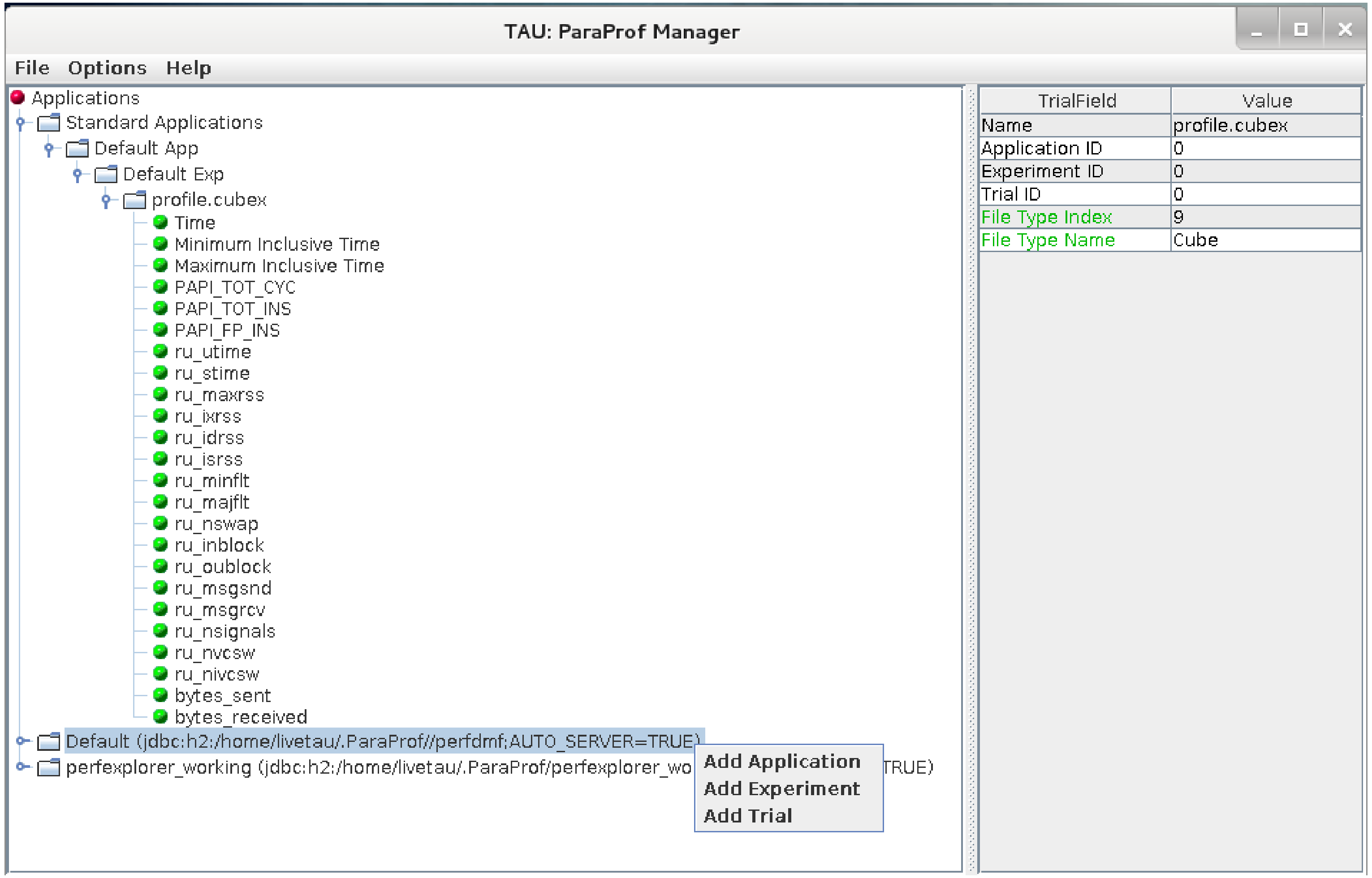
ParaProf: Node View



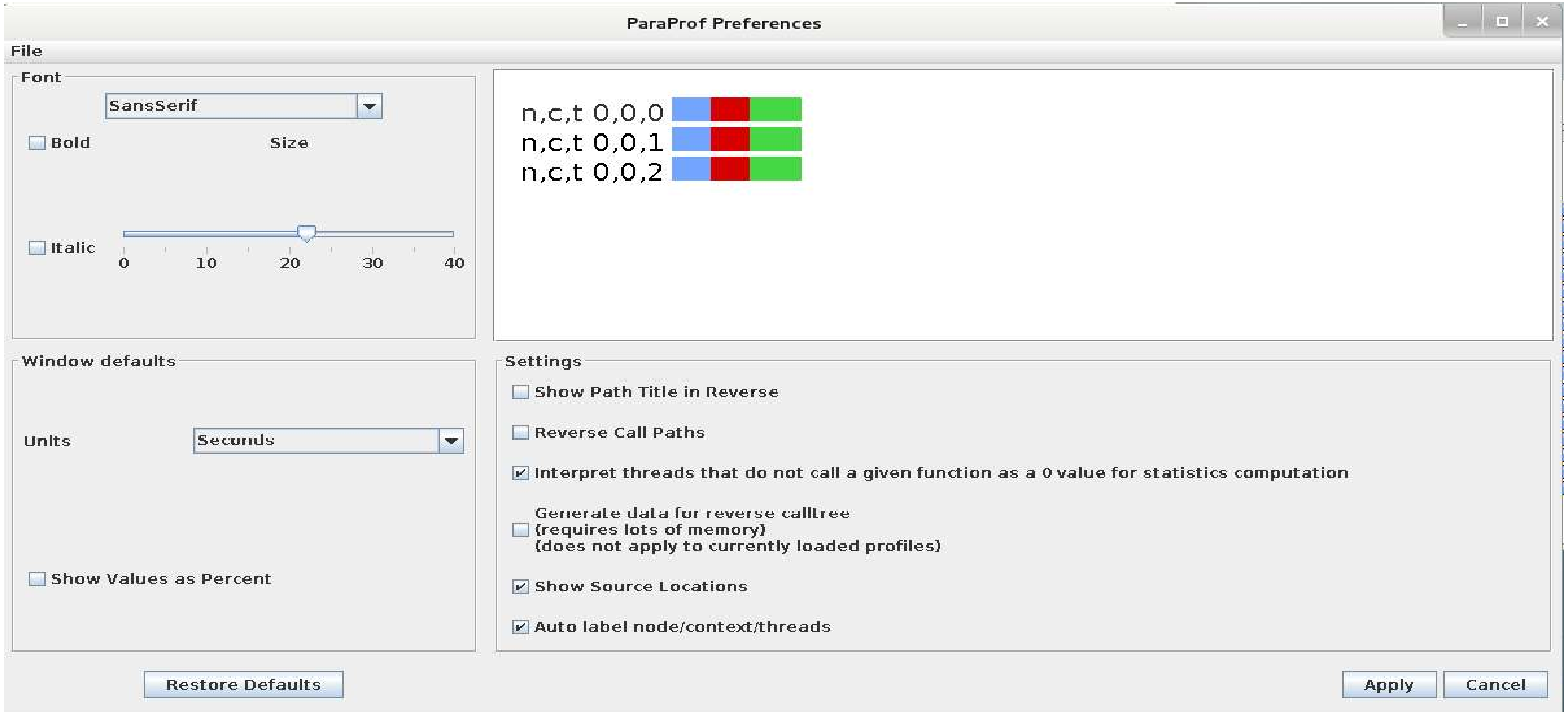
ParaProf: Add thread to comparison window



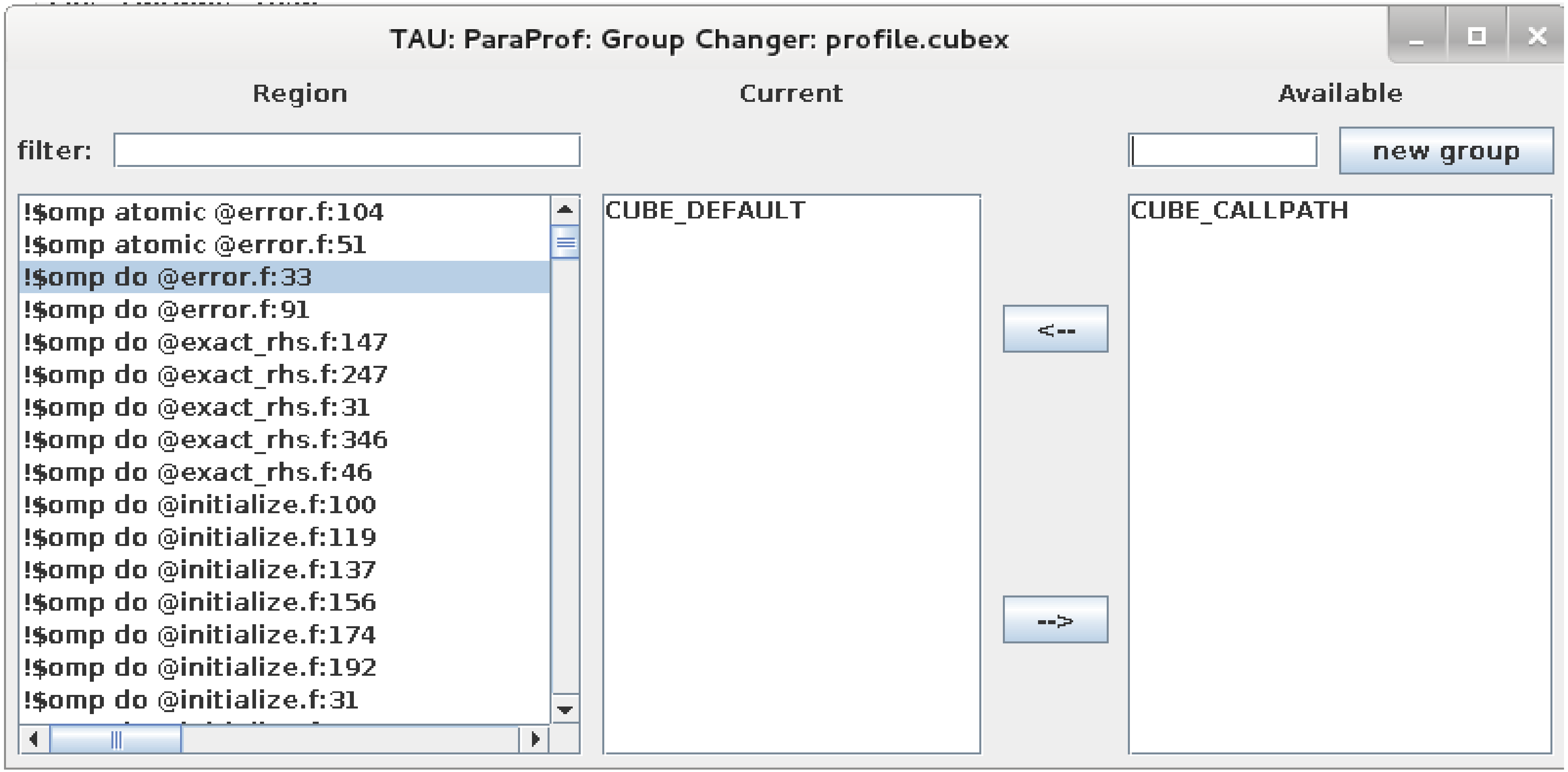
ParaProf: Score-P Profile Files, Database



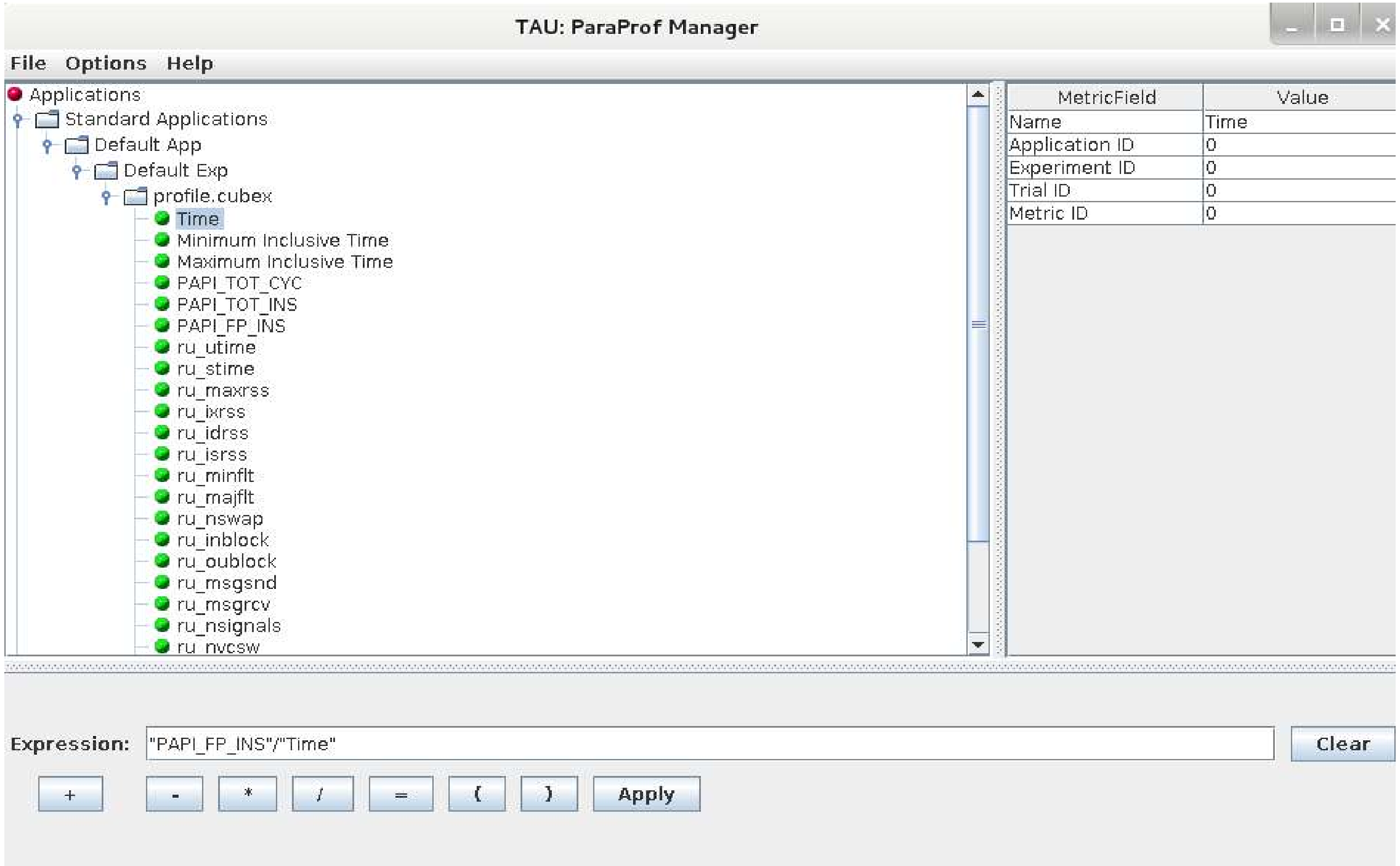
ParaProf: File Preferences Window



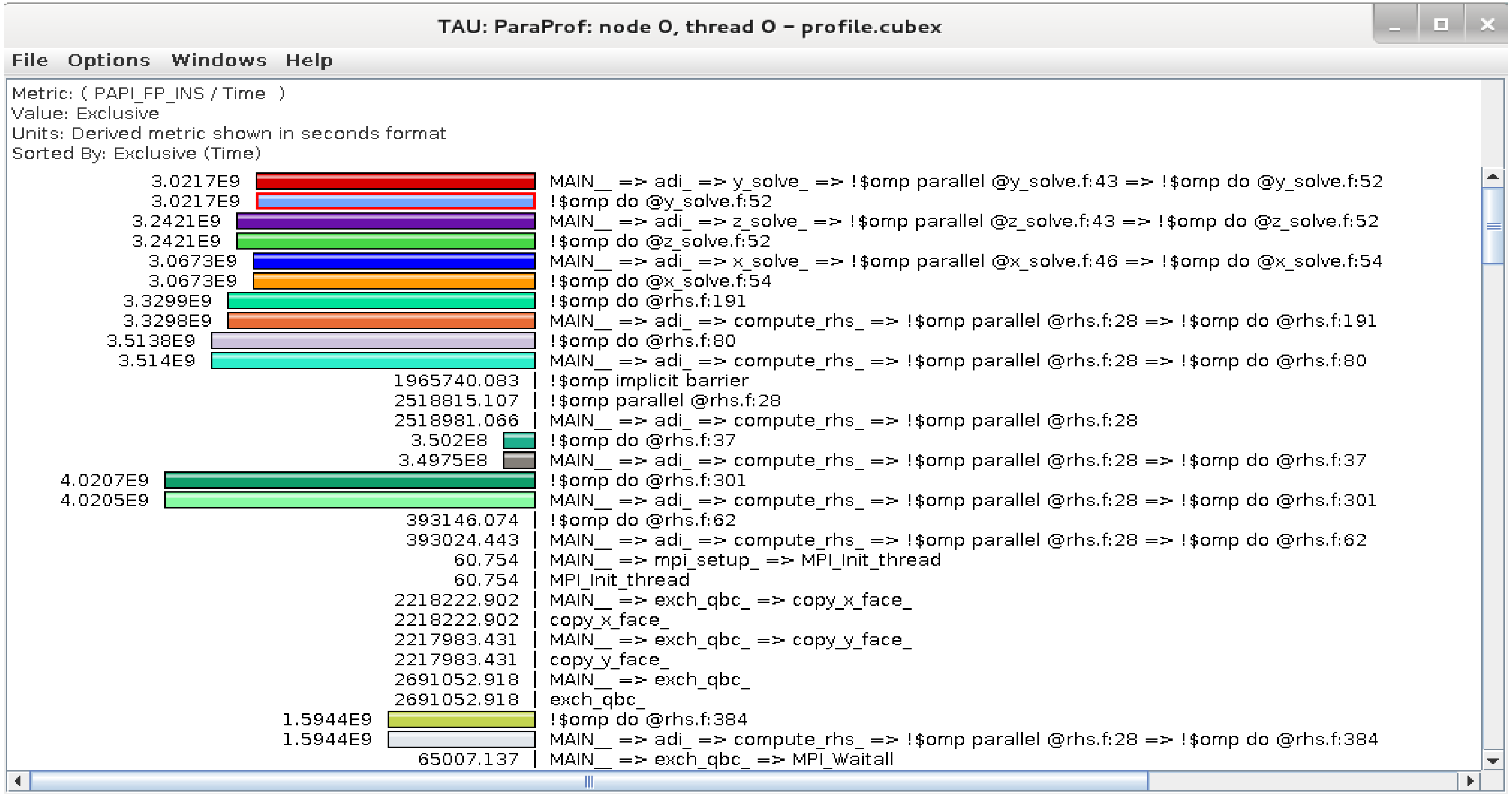
ParaProf: Group Changer Window



ParaProf: Derived Metric Panel in Manager Window



Sorting Derived FLOPS metric by Exclusive Time



Download TAU from U. Oregon

http://tau.uoregon.edu

http://www.hpclinux.com [LiveDVD, OVA]

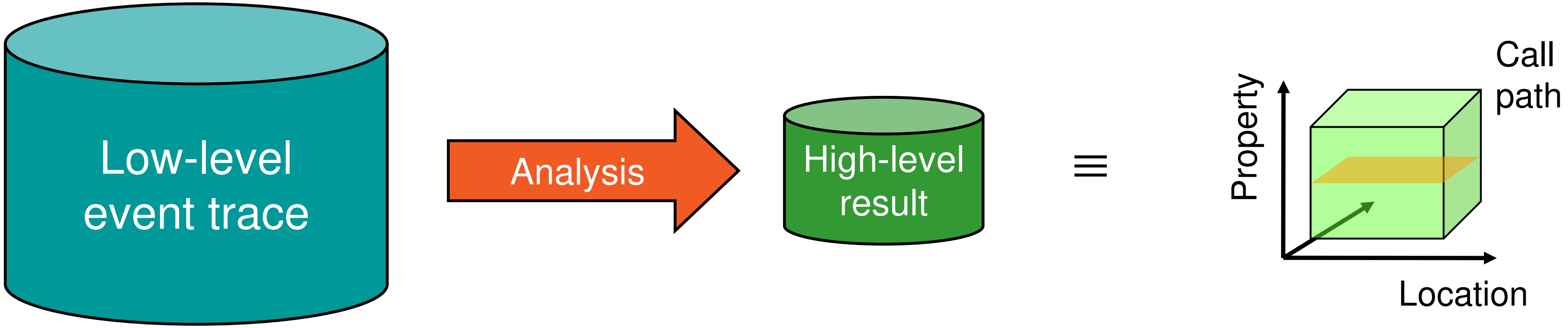
Free download, open source, BSD license

Automatic trace analysis with the Scalasca Trace Tools

Markus Geimer
Jülich Supercomputing Centre

Automatic trace analysis

- Idea
  - Automatic search for patterns of inefficient behavior
  - Classification of behavior & quantification of significance
  - Identification of delays as root causes of inefficiencies



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

Scalasca Trace Tools: Objective

- Development of a **scalable trace-based** performance analysis toolset for the most popular parallel programming paradigms
  - Current focus: MPI, OpenMP, and POSIX threads

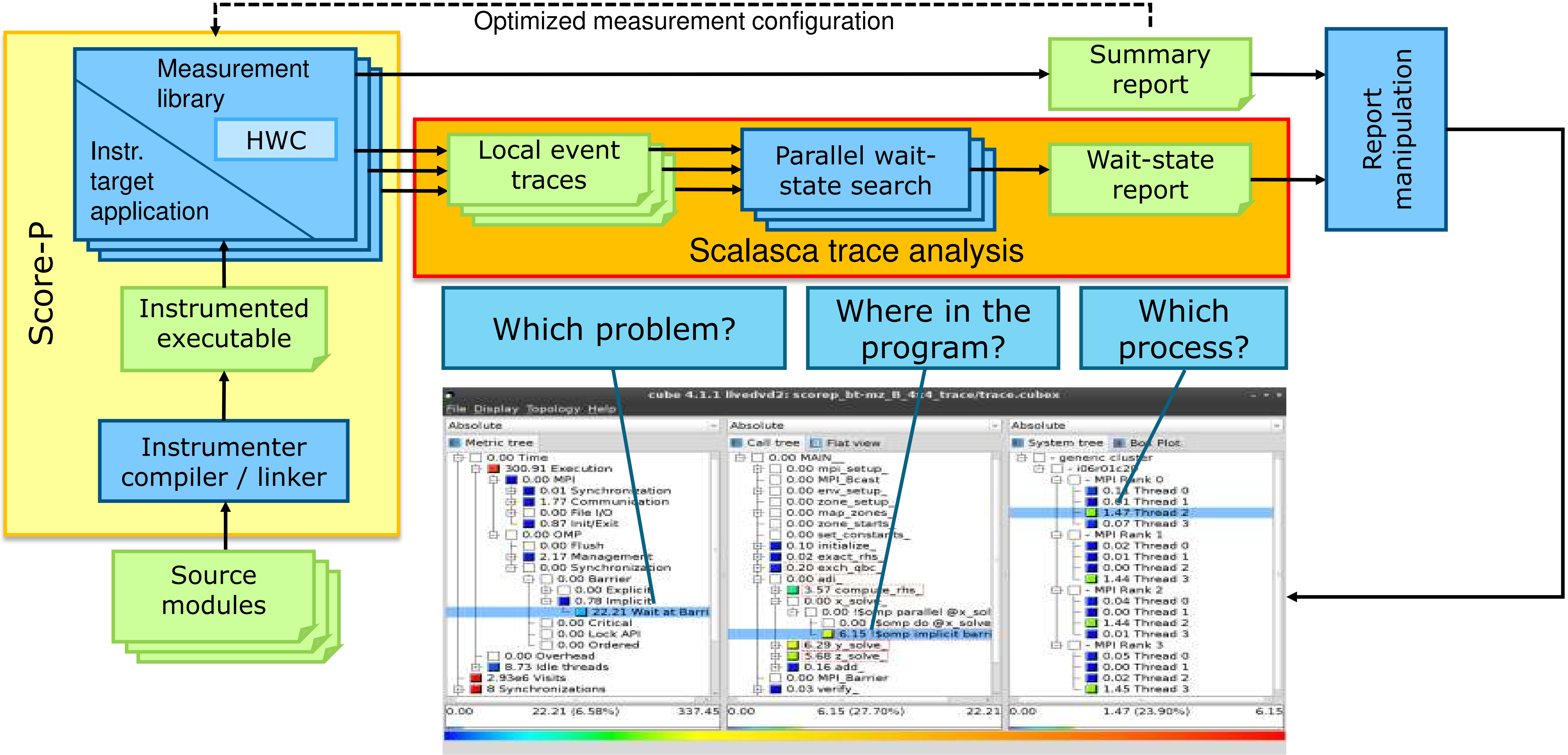
- Specifically targeting large-scale parallel applications
  - Such as those running on IBM Blue Gene or Cray systems with one million or more processes/threads

- Latest release:
  - Scalasca v2.4 coordinated with Score-P v4.0 (May 2018)

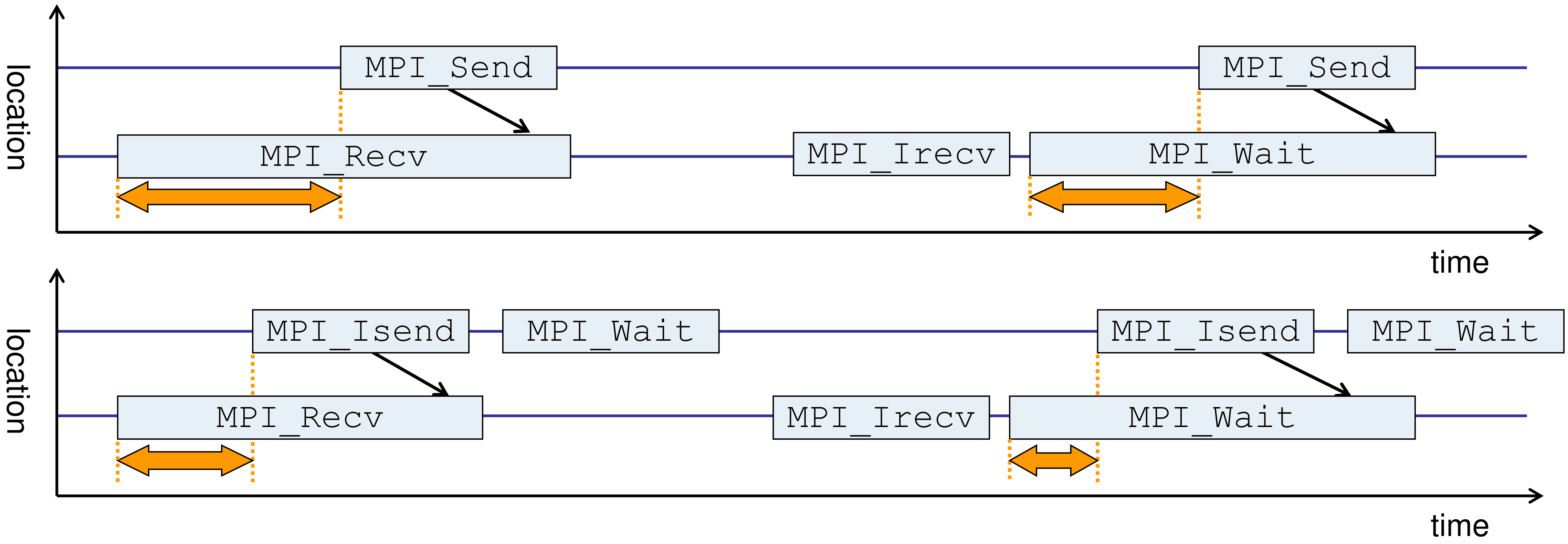
Scalasca Trace Tools features

- Open source, 3-clause BSD license
- Fairly portable
  - IBM Blue Gene, Cray XT/XE/XK/XC, SGI Altix, Fujitsu FX10/100 & K computer, Linux clusters (x86, Power, ARM), Intel Xeon Phi, ...
- Uses Score-P instrumenter & measurement libraries
  - Scalasca v2 core package focuses on trace-based analyses
  - Supports common data formats
    - Reads event traces in OTF2 format
    - Writes analysis reports in CUBE4 format
- Current limitations:
  - Unable to handle traces
    - With MPI thread level exceeding MPI_THREAD_FUNNELED
    - Containing CUDA or SHMEM events, or OpenMP nested parallelism
  - PAPI/rusage metrics for trace events are ignored

Scalasca workflow

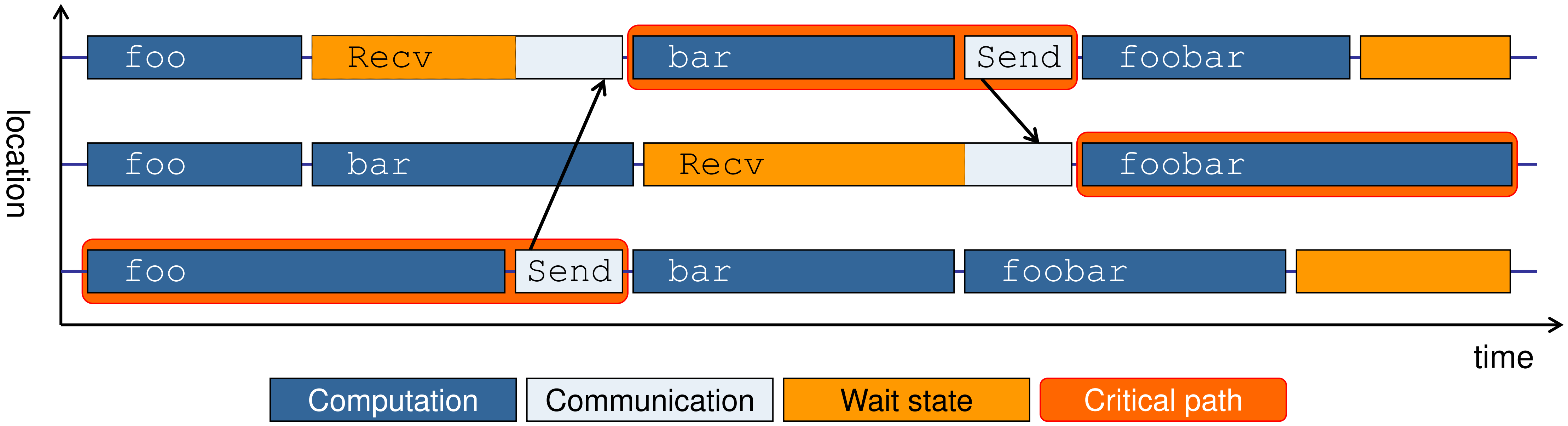


Example: "*Late Sender*" wait state



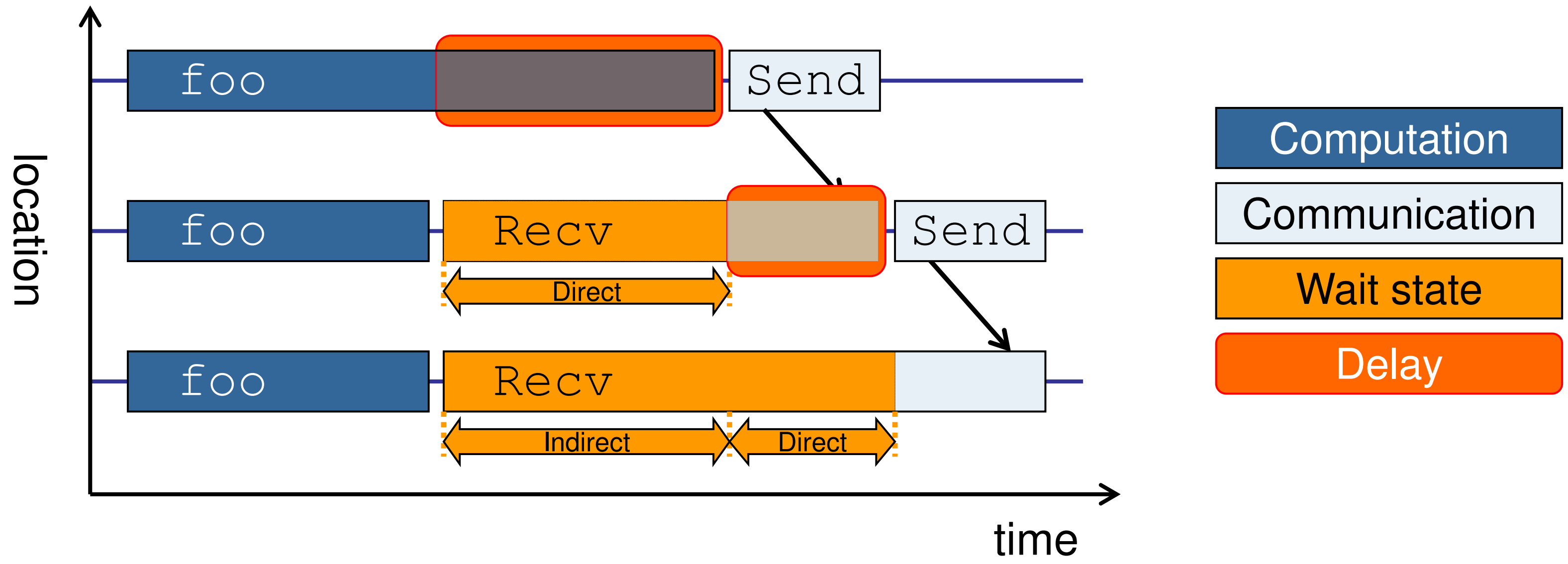
- Waiting time caused by a blocking receive operation posted earlier than the corresponding send
- Applies to blocking as well as non-blocking communication

Example: Critical path



- Shows call paths and processes/threads that are responsible for the program's wall-clock runtime
- Identifies good optimization candidates and parallelization bottlenecks

Example: Root-cause analysis



- Classifies wait states into direct and indirect (i.e., caused by other wait states)
- Identifies *delays* (excess computation/communication) as root causes of wait states
- Attributes wait states as *delay costs*

Hands-on:
NPB-MZ-MPI / BT

Performance analysis steps

- 0.0 Reference preparation for validation

- 1.0 Program instrumentation
- 1.1 Summary measurement collection
- 1.2 Summary analysis report examination

- 2.0 Summary experiment scoring
- 2.1 Summary measurement collection with filtering
- 2.2 Filtered summary analysis report examination

- 3.0 Event trace collection
- 3.1 Event trace examination & analysis

Scalasca command – One command for (almost) everything

```
% scalasca
Scalasca 2.4
Toolset for scalable performance analysis of large-scale parallel applications
usage: scalasca [OPTION]... ACTION <argument>...
    1. prepare application objects and executable for measurement:
       scalasca -instrument <compile-or-link-command> # skin (using scorep)
    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

Options:
   -c, --show-config     show configuration summary and exit
   -h, --help            show this help and exit
   -n, --dry-run         show actions without taking them
       --quickref        show quick reference guide and exit
       --remap-specfile  show path to remapper specification file and exit
   -v, --verbose         enable verbose commentary
   -V, --version         show version information and exit
```

- The '`scalasca -instrument`' command is deprecated and only provided for backwards compatibility with Scalasca 1.x., recommended: use Score-P instrumenter directly

Scalasca compatibility command: skin / scalasca -instrument

```
% skin
Scalasca 2.4: application instrumenter (using Score-P instrumenter)
usage: skin [-v] [-comp] [-pdt] [-pomp] [-user] [--*] <compile-or-link-command>
  -comp={all|none|...}: routines to be instrumented by compiler [default: all]
                  (... custom instrumentation specification depends on compiler)
  -pdt:  process source files with PDT/TAU instrumenter
  -pomp: process source files for POMP directives
  -user: enable EPIK user instrumentation API macros in source code
  -v:    enable verbose commentary when instrumenting

  --*:   options to pass to Score-P instrumenter
```

- Scalasca application instrumenter
  - Provides compatibility with Scalasca 1.x
  - **Deprecated! Use Score-P instrumenter directly.**

Scalasca convenience command: scan / scalasca -analyze

```
% scan
Scalasca 2.4: measurement collection & analysis nexus
usage: scan {options} [launchcmd [launchargs]] target [targetargs]
      where {options} may include:
  -h    Help: show this brief usage message and exit.
  -v    Verbose: increase verbosity.
  -n    Preview: show command(s) to be launched but don't execute.
  -q    Quiescent: execution with neither summarization nor tracing.
  -s    Summary: enable runtime summarization. [Default]
  -t    Tracing: enable trace collection and analysis.
  -a    Analyze: skip measurement to (re-)analyze an existing trace.
  -e exptdir   : Experiment archive to generate and/or analyze.
                 (overrides default experiment archive title)
  -f filtfile  : File specifying measurement filter.
  -l lockfile  : File that blocks start of measurement.
  -m metrics   : Metric specification for measurement.
```

- Scalasca measurement collection & analysis nexus

Scalasca advanced command:
scout - Scalasca automatic trace analyzer

```
% scout.hyb --help
SCOUT   Copyright (c) 1998-2018 Forschungszentrum Juelich GmbH
        Copyright (c) 2009-2014 German Research School for Simulation
                                Sciences GmbH

Usage: <launchcmd> scout.hyb [OPTION]... <ANCHORFILE | EPIK DIRECTORY>
Options:
  --statistics        Enables instance tracking and statistics [default]
  --no-statistics     Disables instance tracking and statistics
  --critical-path     Enables critical-path analysis [default]
  --no-critical-path  Disables critical-path analysis
  --rootcause         Enables root-cause analysis [default]
  --no-rootcause      Disables root-cause analysis
  --single-pass       Single-pass forward analysis only
  --time-correct      Enables enhanced timestamp correction
  --no-time-correct   Disables enhanced timestamp correction [default]
  --verbose, -v       Increase verbosity
  --help              Display this information and exit
```

- Provided in serial (.ser), OpenMP (.omp), MPI (.mpi) and MPI+OpenMP (.hyb) variants

Scalasca advanced command: clc_synchronize

- Scalasca trace event timestamp consistency correction

```
Usage: <launchcmd> clc_synchronize.hyb <ANCHORFILE | EPIK_DIRECTORY>
```

- Provided in MPI (.mpi) and MPI+OpenMP (.hyb) variants
- Takes as input a trace experiment archive where the events may have timestamp inconsistencies
  - E.g., multi-node measurements on systems without adequately synchronized clocks on each compute node
- Generates a new experiment archive (always called ./clc_sync) containing a trace with event timestamp inconsistencies resolved
  - E.g., suitable for detailed examination with a time-line visualizer

Scalasca convenience command: square / scalasca -examine

```
% square
Scalasca 2.4: analysis report explorer
usage: square [-v] [-s] [-f filtfile] [-F] <experiment archive | cube file>
   -c <none | quick | full> : Level of sanity checks for newly created reports
   -F                       : Force remapping of already existing reports
   -f filtfile              : Use specified filter file when doing scoring
   -s                       : Skip display and output textual score report
   -v                       : Enable verbose mode
   -n                       : Do not include idle thread metric
```

- Scalasca analysis report explorer (Cube)

Automatic measurement configuration

- scan configures Score-P measurement by automatically setting some environment variables and exporting them
  - E.g., experiment title, profiling/tracing mode, filter file, …
  - Precedence order:
    - Command-line arguments
    - Environment variables already set
    - Automatically determined values
- Also, scan includes consistency checks and prevents corrupting existing experiment directories
- For tracing experiments, after trace collection completes then automatic parallel trace analysis is initiated
  - Uses identical launch configuration to that used for measurement (i.e., the same allocated compute resources)

Setup environment

- Remember to source provided shell code snippet to add local tool installations to $PATH

```
% source ~tg828282/Tutorial/vihps.sh
```

- Change to directory containing NPB3.3-MZ-MPI sources
- Existing instrumented executable in bin.scorep/ directory can be reused

```
% cd $SCRATCH/NPB3.3-MZ-MPI
```

BT-MZ summary measurement collection...

```
% cd bin.scorep
% cp ../jobscript/stampede2/scalasca.sbatch .
% vi scalasca.sbatch

# Score-P measurement configuration
export SCOREP_FILTERING_FILE=../config/scorep.filt
#export SCOREP_TOTAL_MEMORY=50M
#export SCOREP_METRIC_PAPI=PAPI_TOT_INS,PAPI_TOT_CYC

# Scalasca configuration
export SCAN_ANALYZE_OPTS="--time-correct"

# Run the application using Scalasca nexus
scalasca -analyze ibrun ./bt-mz_${CLASS}.${PROCS}
```

- Change to directory with the executable and edit the job script

```
% sbatch scalasca.sbatch
```

- Submit the job

BT-MZ summary measurement

```
S=C=A=N: Scalasca 2.4 runtime summarization
S=C=A=N: ./scorep_bt-mz_C_32x4_sum experiment archive
S=C=A=N: Mon Aug 21 07:52:03 2017: Collect start
ibrun ./bt-mz_C.32

 NAS Parallel Benchmarks (NPB3.3-MZ-MPI) -
    BT-MZ MPI+OpenMP Benchmark

 Number of zones:  16 x  16
 Iterations: 200    dt:   0.000100
 Number of active processes:     32

 [... More application output ...]

S=C=A=N: Mon Aug 21 07:52:36 2017: Collect done (status=0) 33s
S=C=A=N: ./scorep_bt-mz_C_32x4_sum complete.
```

- Run the application using the Scalasca measurement collection & analysis nexus prefixed to launch command
- Creates experiment directory:
  scorep_bt-mz_C_32x4_sum

BT-MZ summary analysis report examination

- Score summary analysis report

```
% square -s  scorep_bt-mz_C_32x4_sum
INFO: Post-processing runtime summarization result...
INFO: Score report written to ./scorep_bt-mz_C_32x4_sum/scorep.score
```

- Post-processing and interactive exploration with Cube

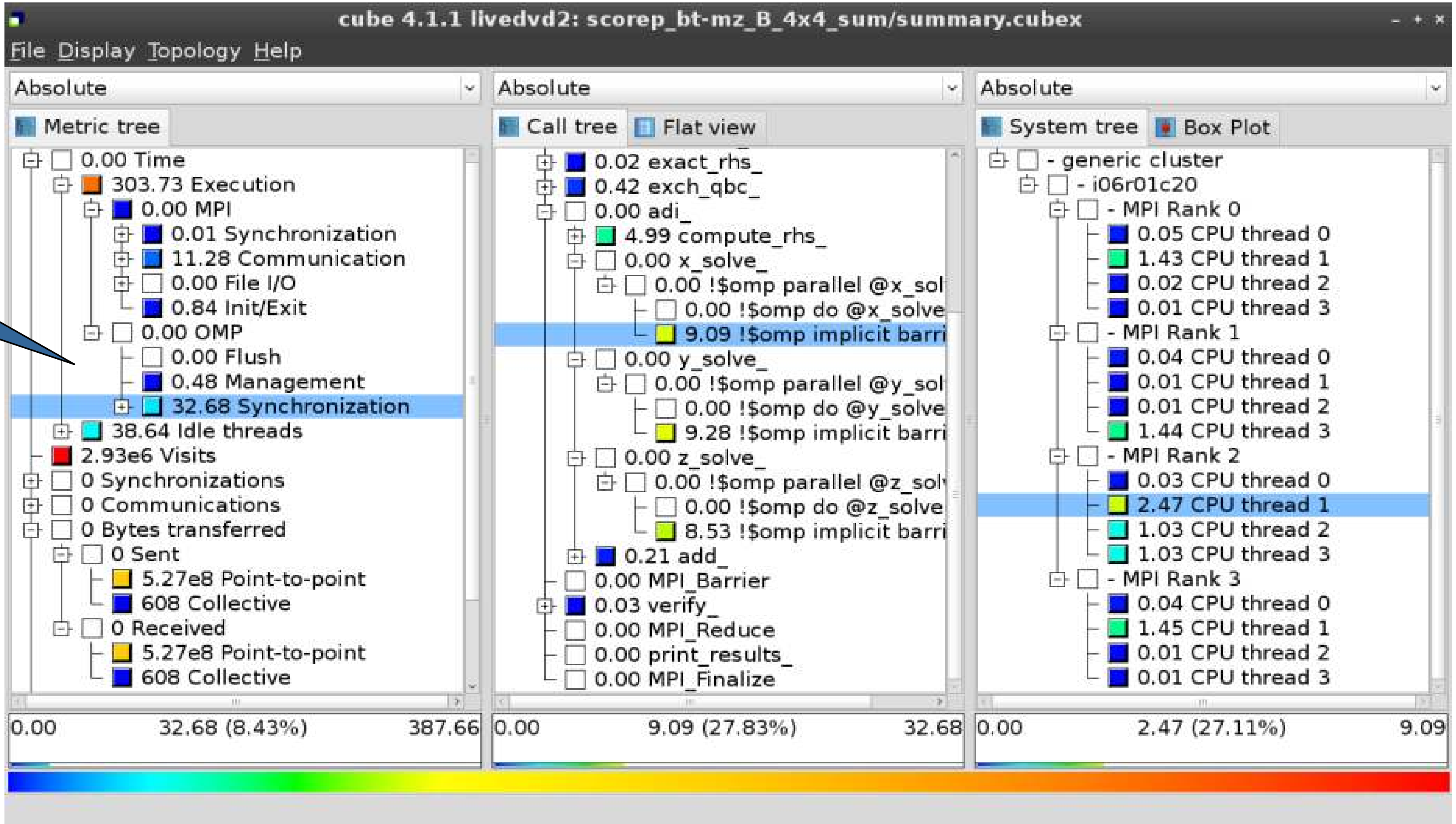
```
% square  scorep_bt-mz_C_32x4_sum
INFO: Displaying ./scorep_bt-mz_C_32x4_sum/summary.cubex...

                  [GUI showing summary analysis report]
```

Hint:
Copy 'summary.cubex' to Live-DVD environment using 'scp' to improve responsiveness of GUI

- The post-processing derives additional metrics and generates a structured metric hierarchy

Post-processed summary analysis report

Split base metrics into more specific metrics



Performance analysis steps

- 0.0 Reference preparation for validation

- 1.0 Program instrumentation
- 1.1 Summary measurement collection
- 1.2 Summary analysis report examination

- 2.0 Summary experiment scoring
- 2.1 Summary measurement collection with filtering
- 2.2 Filtered summary analysis report examination

- 3.0 Event trace collection
- 3.1 Event trace examination & analysis

BT-MZ trace measurement collection...

```
% cd bin.scorep
% cp ../jobscript/stampede2/scalasca.sbatch .
% vi scalasca.sbatch

# Score-P measurement configuration
export SCOREP_FILTERING_FILE=../config/scorep.filt
export SCOREP_TOTAL_MEMORY=50M
export SCOREP_METRIC_PAPI=PAPI_TOT_INS,PAPI_TOT_CYC

# Scalasca configuration
export SCAN_ANALYZE_OPTS="--time-correct"

# Run the application using Scalasca nexus
scalasca -analyze -t ibrun ./bt-mz_${CLASS}.${PROCS}
```

- Change to directory with the executable and edit the job script
- Add "-t" to the scalasca -analyze command

```
% sbatch scalasca.sbatch
```

- Submit the job

BT-MZ trace measurement ... collection

```
S=C=A=N: Scalasca 2.4 trace collection and analysis
S=C=A=N: Mon Aug 21 07:58:54 2017: Collect start
ibrun ./bt-mz_C.32

 NAS Parallel Benchmarks (NPB3.3-MZ-MPI) - BT-MZ MPI+OpenMP \
>Benchmark

 Number of zones:  16 x  16
 Iterations: 200    dt:   0.000100
 Number of active processes:    32

 [... More application output ...]

S=C=A=N: Mon Aug 21 07:59:29 2017: Collect done (status=0) 35s
```

- Starts measurement with collection of trace files ...

BT-MZ trace measurement ... analysis

```
S=C=A=N: Mon Aug 21 07:59:30 2017: Analyze start
ibrun scout.hyb ./scorep_bt-mz_C_32x4_trace/traces.otf2

Analyzing experiment archive ./scorep_bt-mz_C_32x4_trace/traces.otf2

Opening experiment archive ... done (0.040s).
Reading definition data    ... done (0.127s).
Reading event trace data   ... done (0.726s).
Preprocessing              ... done (0.311s).
Timestamp correction       ... done (0.556s).
Analyzing trace data       ... done (15.144s).
Writing analysis report    ... done (0.738s).

Total processing time     : 17.754s
S=C=A=N: Mon Aug 21 07:59:50 2017: Analyze done (status=0) 20s
```

- Continues with automatic (parallel) analysis of trace files

BT-MZ trace analysis report exploration

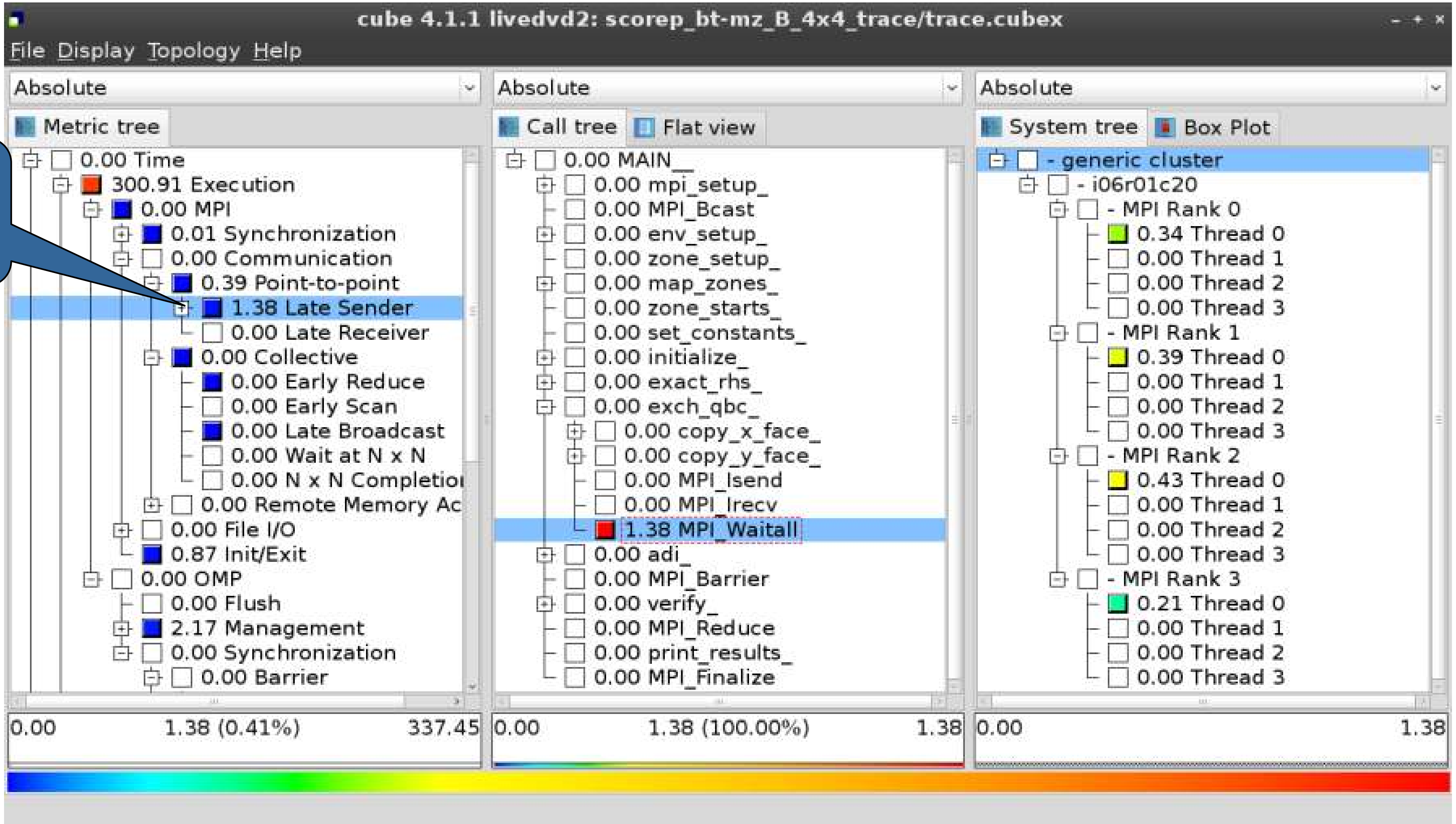
- Produces trace analysis report in the experiment directory containing trace-based wait-state metrics

```
% square  scorep_bt-mz_C_32x4_trace
INFO: Post-processing runtime summarization result...
INFO: Post-processing trace analysis report...
INFO: Displaying ./scorep_bt-mz_C_32x4_trace/trace.cubex...

                [GUI showing trace analysis report]
```

Hint:
Run ‘square -s’ first and then copy ‘trace.cubex’ to Live-DVD environment using ‘scp’ to improve responsiveness of GUI

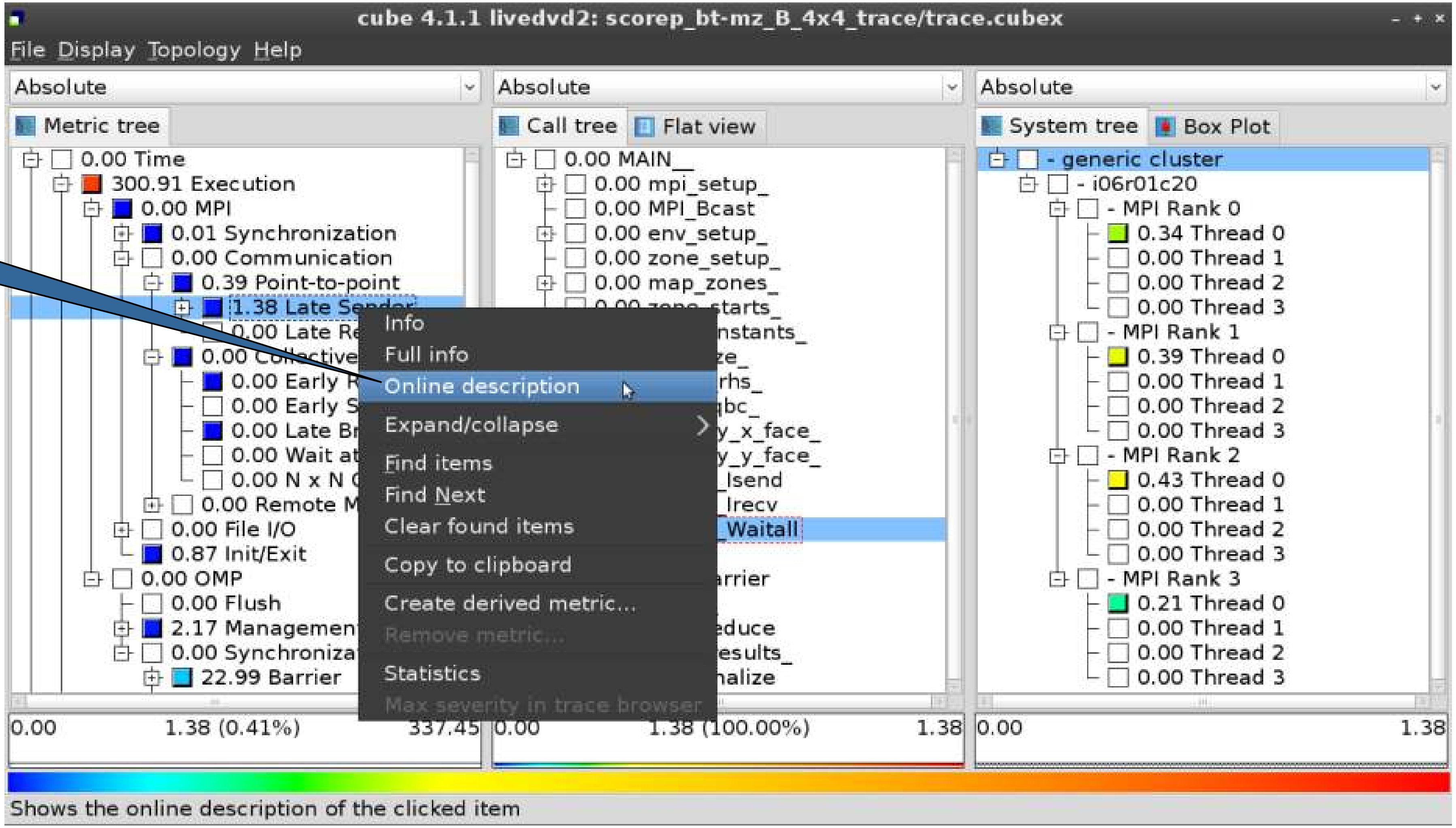
Post-processed trace analysis report

Additional trace-based metrics in metric hierarchy



Online metric description

Access online metric description via context menu



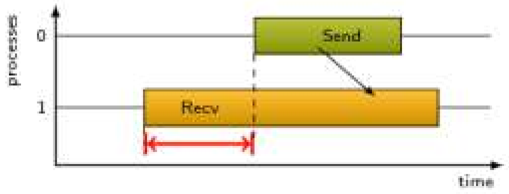
Online metric description

Performance properties

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.



If the receiving process is waiting for multiple messages to arrive (e.g., in an call to `MPI_Waitall`), the maximum waiting time is accounted, i.e., the waiting time due to the latest sender.

Unit:

Seconds

Diagnosis:

Try to replace `MPI_Recv` with a non-blocking receive `MPI_Irecv` that can be posted earlier, proceed concurrently with computation, and complete with a wait operation after the message is expected to have been sent. Try to post sends earlier, such that they are available when receivers need them. Note that outstanding messages (i.e., sent before the receiver is ready) will occupy internal message buffers, and that large numbers of posted receive buffers will also introduce message management overhead, therefore moderation is advisable.

Parent:

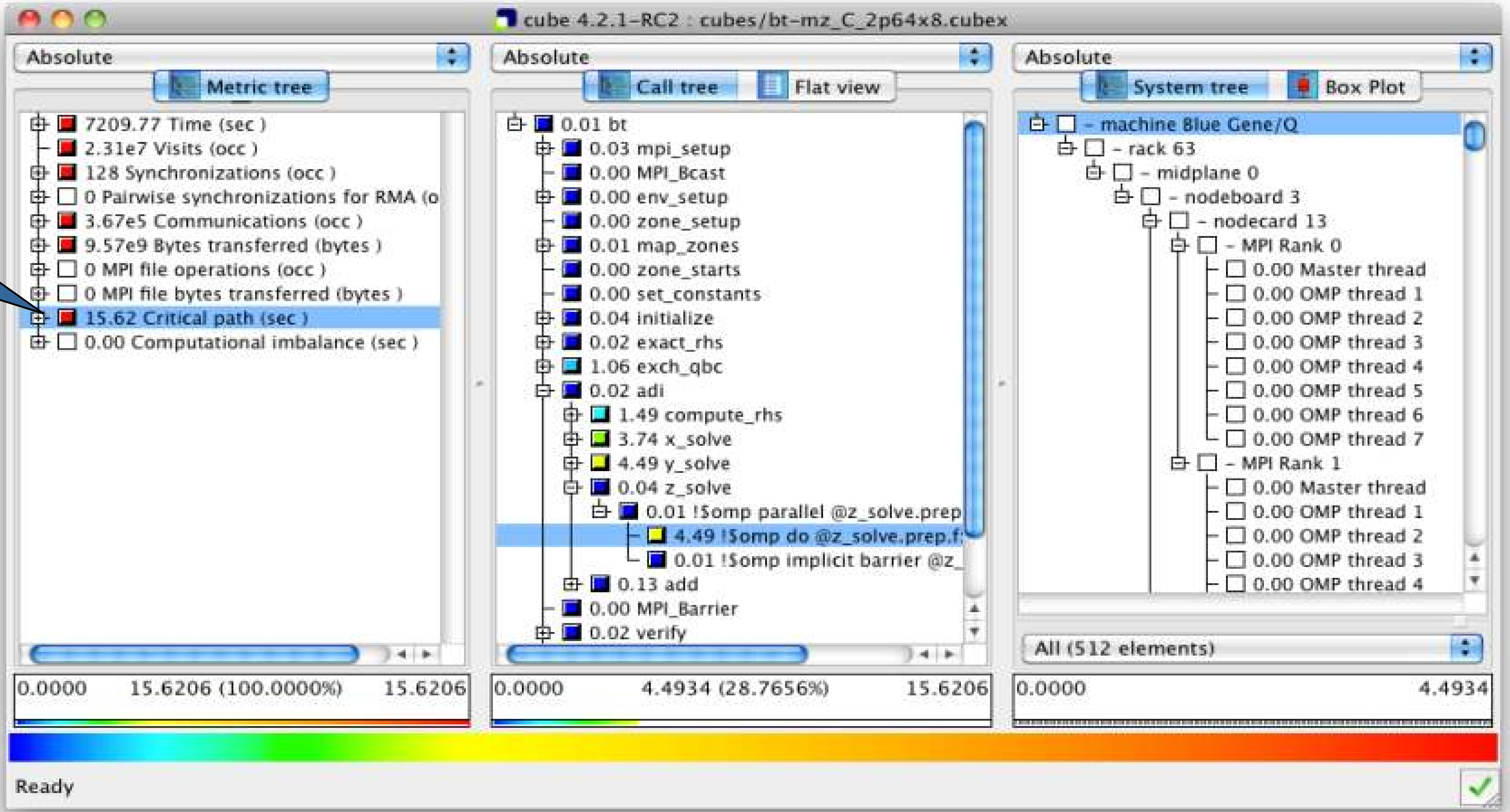
MPI Point-to-point Communication Time

Children:

Close

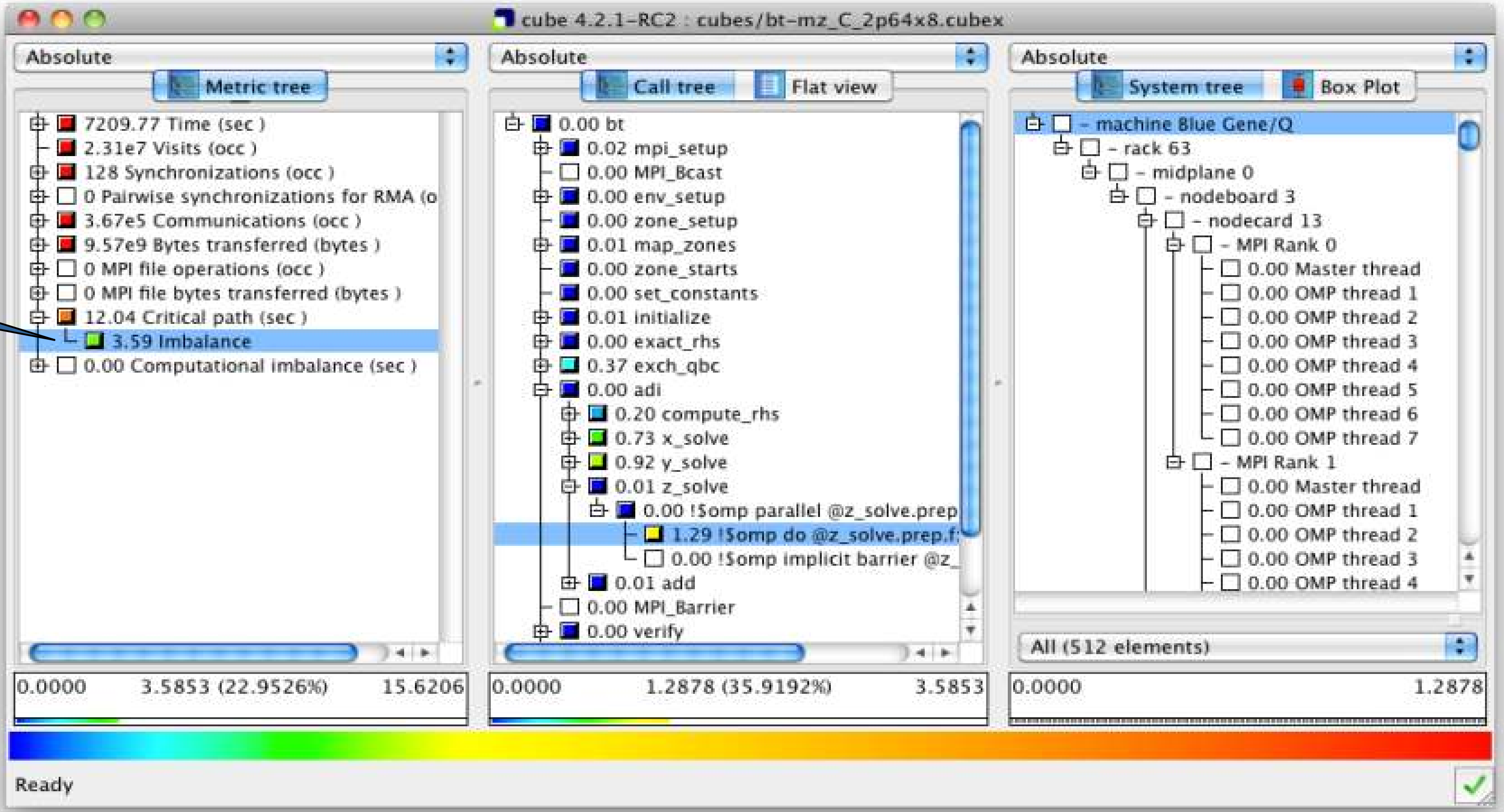
Critical-path analysis

Critical-path profile shows wall-clock time impact



Critical-path analysis

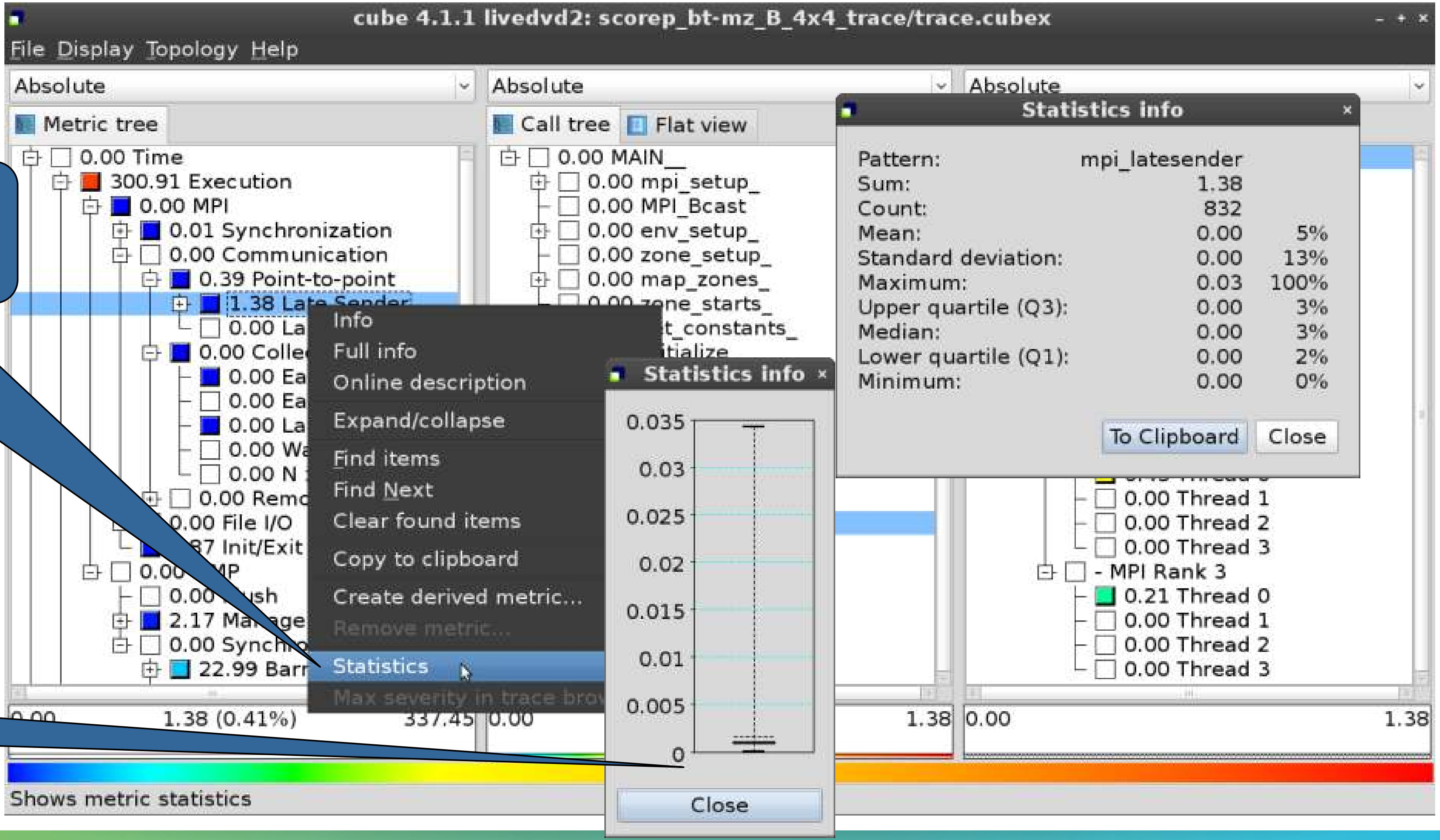
Critical-path imbalance highlights inefficient parallelism



Pattern instance statistics

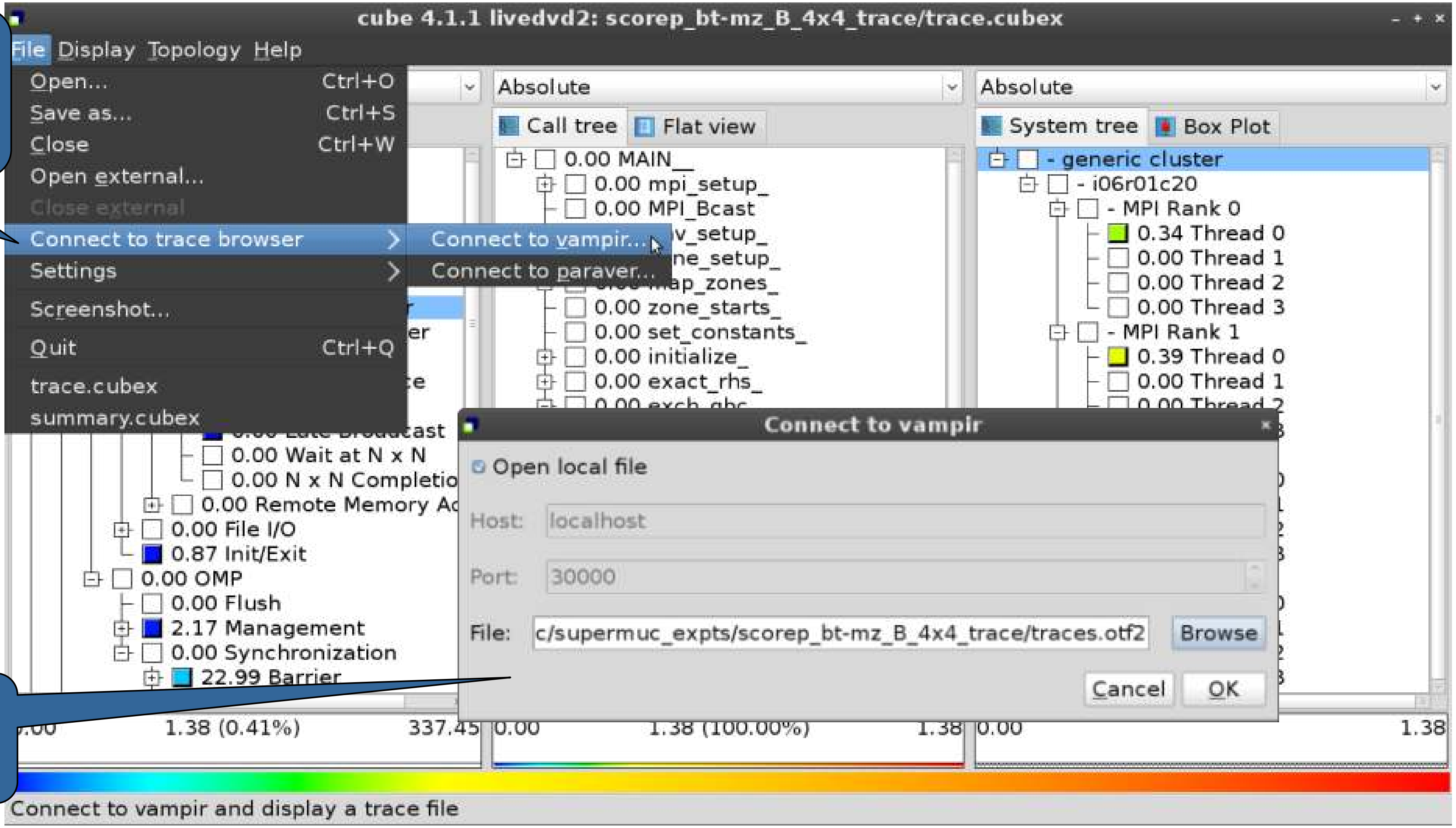
Access pattern instance statistics via context menu

Click to get statistics details



Connect to Vampir trace browser

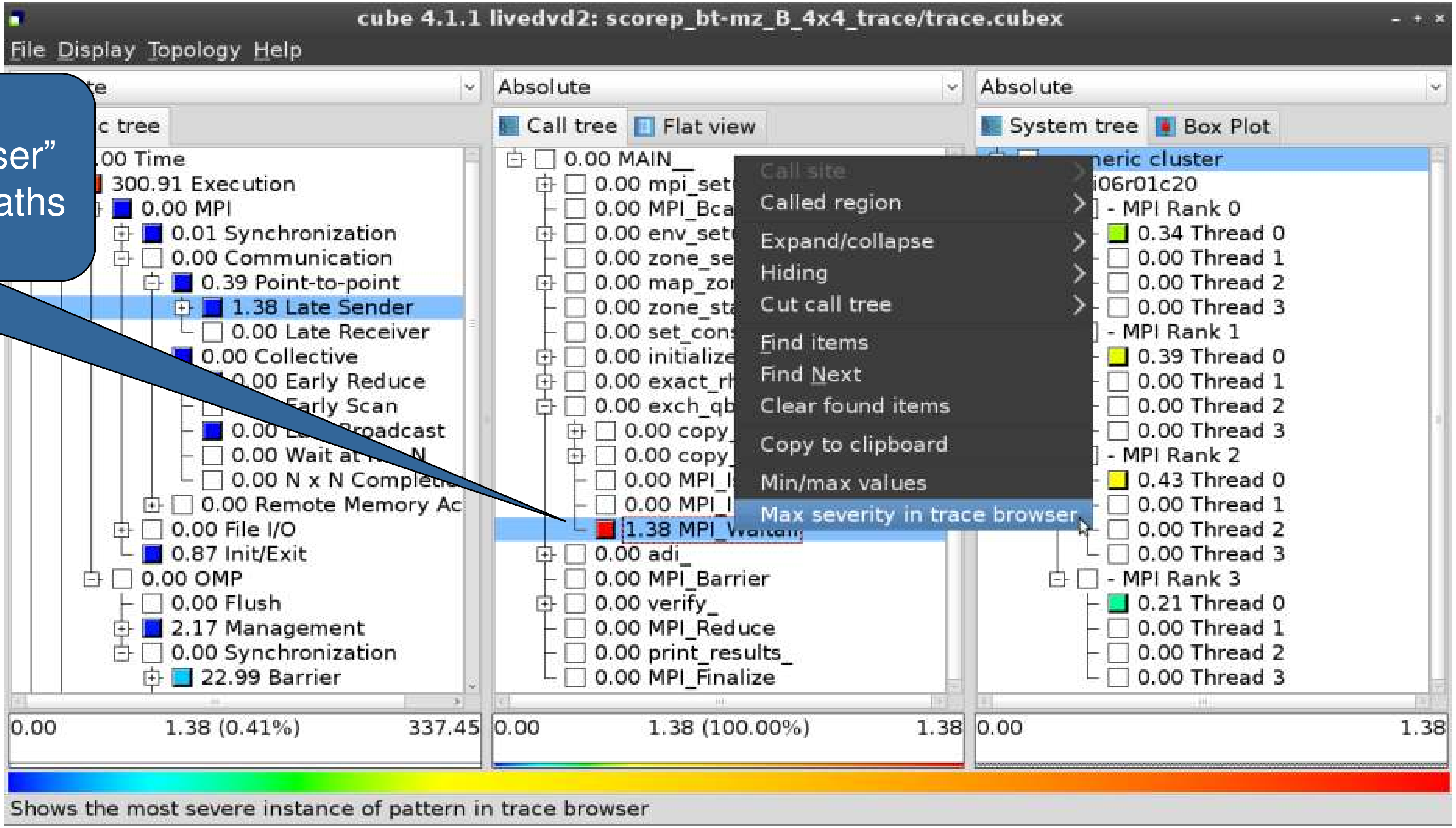
To investigate most severe pattern instances, connect to a trace browser…



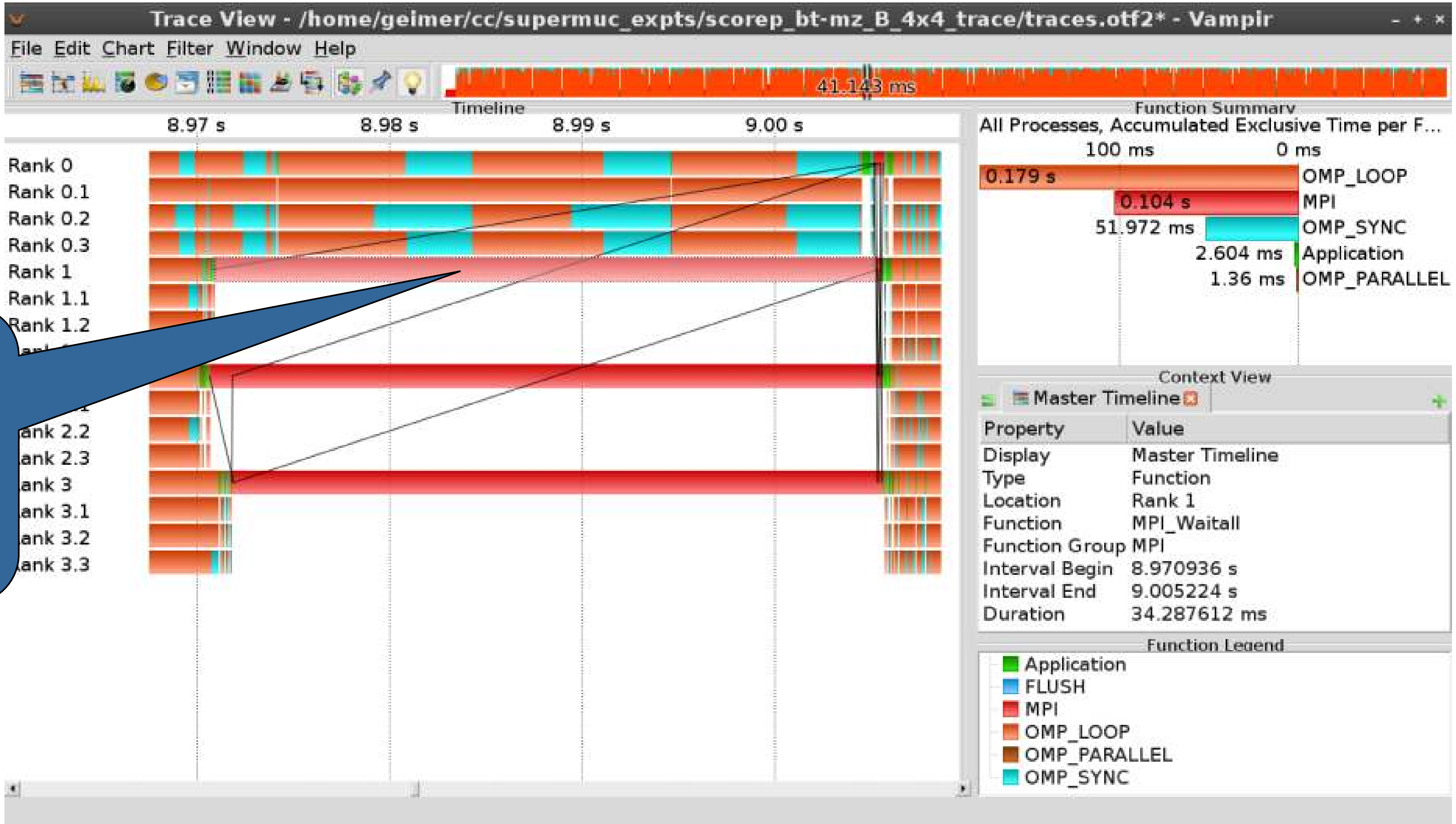
…and select trace file from the experiment directory

Show most severe pattern instances

Select
“Max severity in trace browser”
from context menu of call paths
marked with a red frame



Investigate most severe instance in Vampir



Vampir will automatically zoom to the worst instance in multiple steps (i.e., undo zoom provides more context)

Scalasca Trace Tools: Further information

- Collection of trace-based performance tools
  - Specifically designed for large-scale systems
  - Features an automatic trace analyzer providing wait-state, critical-path, and delay analysis
  - Supports MPI, OpenMP, POSIX threads, and hybrid MPI+OpenMP/Pthreads
- Available under 3-clause BSD open-source license
- Documentation & sources:
  - http://www.scalasca.org
- Contact:
  - mailto: scalasca@fz-juelich.de

Performance Analysis with Vampir

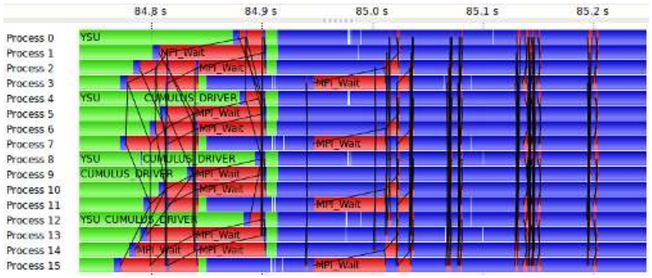
Ronny Tschüter, Bert Wesarg, Matthias Weber
Technische Universität Dresden

Outline

- **Part I: Welcome to the Vampir Tool Suite**
  - Mission
  - Event Trace Visualization
  - Vampir & VampirServer
  - The Vampir Displays
- **Part II: Vampir Hands-On**
  - Visualizing and analyzing NPB-MZ-MPI / BT

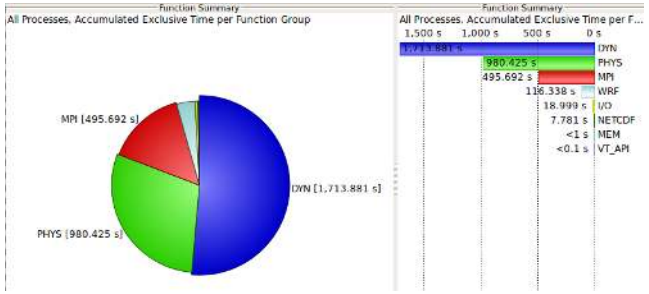
Event Trace Visualization with Vampir

- Alternative and supplement to automatic analysis
- Show dynamic run-time behavior graphically at any level of detail
- Provide statistics and performance metrics



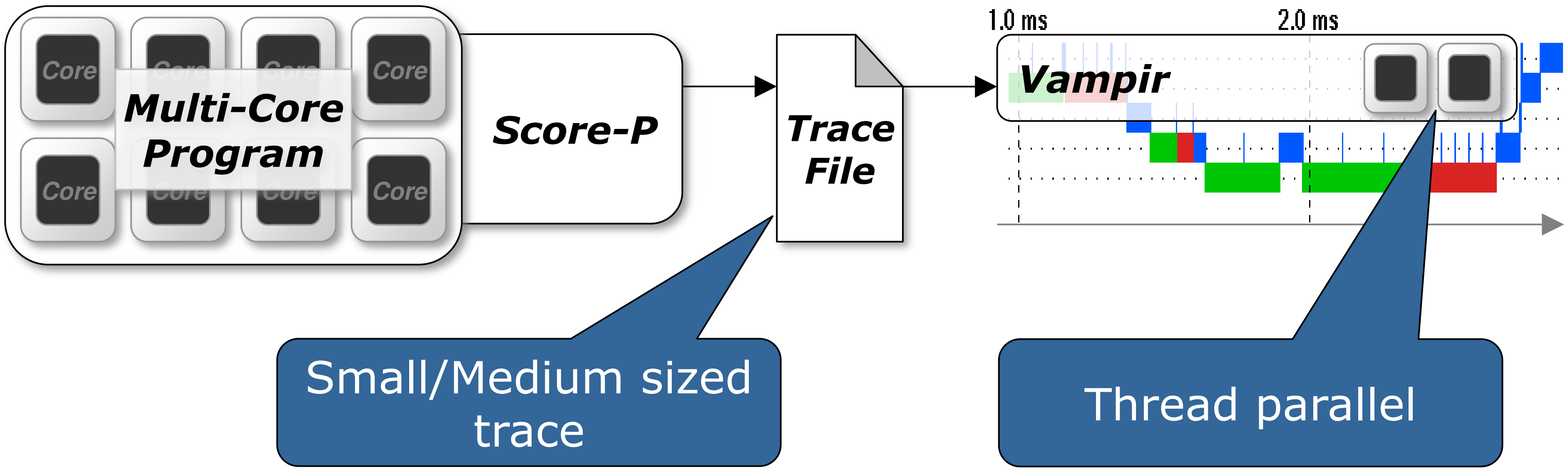
- **Timeline charts**
  - Show application activities and communication along a time axis

- **Summary charts**
  - Provide quantitative results for the currently selected time interval



Visualization Modes (1)
Directly on front end or local machine

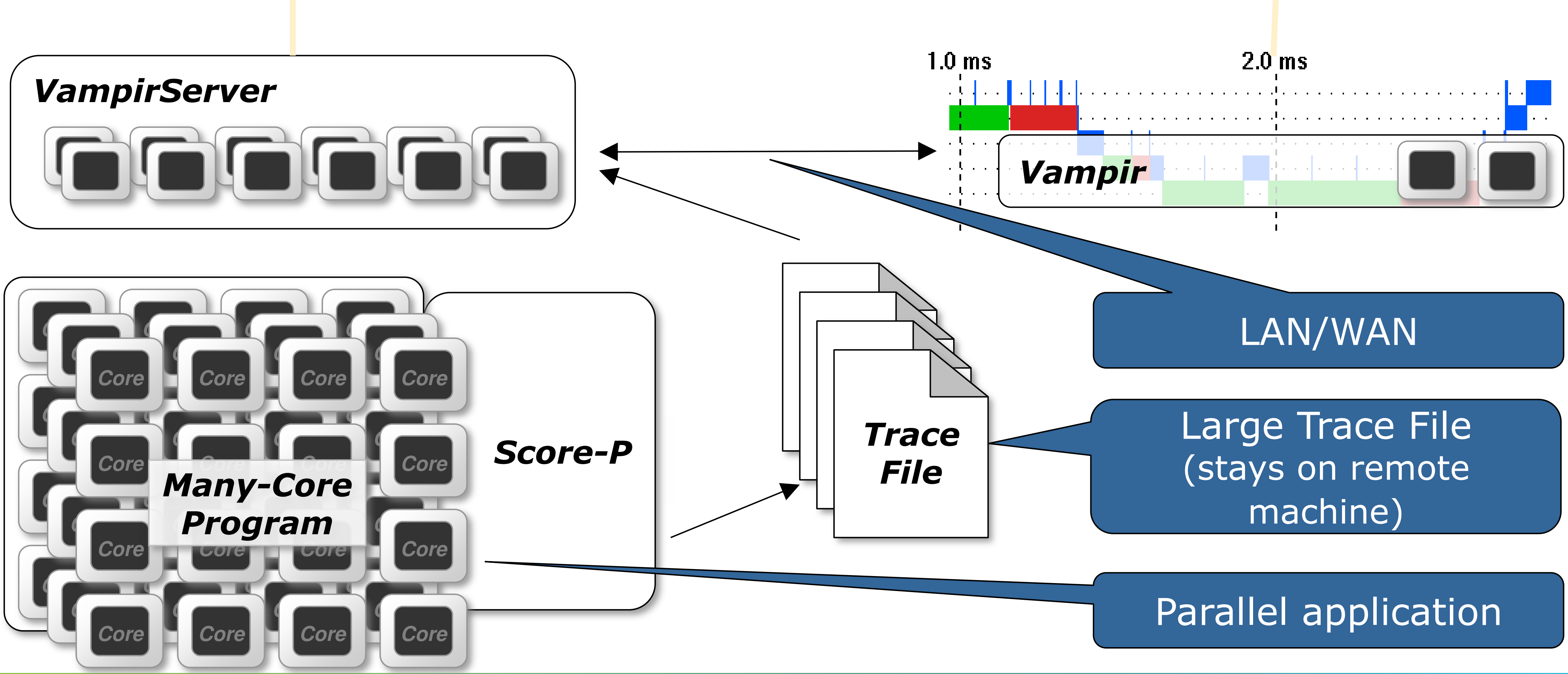
```
% vampir
```



Visualization Modes (2)
On local machine with remote VampirServer

```
% vampirserver start
```

```
% vampir
```



The main displays of Vampir

- Timeline Charts:
  - Master Timeline
  - Process Timeline
  - Counter Data Timeline
  - Performance Radar
- Summary Charts:
  - Function Summary
  - Message Summary
  - Process Summary
  - Communication Matrix View

Hands-on:
Visualizing and analyzing NPB-MZ-MPI / BT

Help! Where is my trace file?

```
% ls $SCRATCH/NPB3.3-MZ-MPI/bin.scorep/\
> scorep_bt-mz_C_32x4_trace
profile.cubex  scorep.cfg   traces/   traces.def  traces.otf2
```

```
% ls ~tg828282/Tutorial/Experiments/scorep_bt-mz_C_32x4_trace
profile.cubex  scorep.cfg   traces/   traces.def  traces.otf2
```

- If you followed the Score-P hands-on up to the trace experiment
- If you did not follow to that point, take a prepared trace

Starting VampirServer on Stampede

```
% vampirserver start
Launching VampirServer...
Submitting batch job (this might take a while)...
```

- Start VampirServer on Stampede2

Starting VampirServer on Stampede

```
% vampirserver start
Launching VampirServer...
Submitting batch job (this might take a while)...

VampirServer 9.2.0  (r10676)
Licensed to ZIH, TU Dresden (@ISC 2017)
Running 4 analysis processes... (abort with \
  vampirserver stop 28974)
VampirServer <28974> listens on: \
  c401-602.stampede2.tacc.utexas.edu:30019
```

- Start VampirServer on Stampede2

Copy host:port

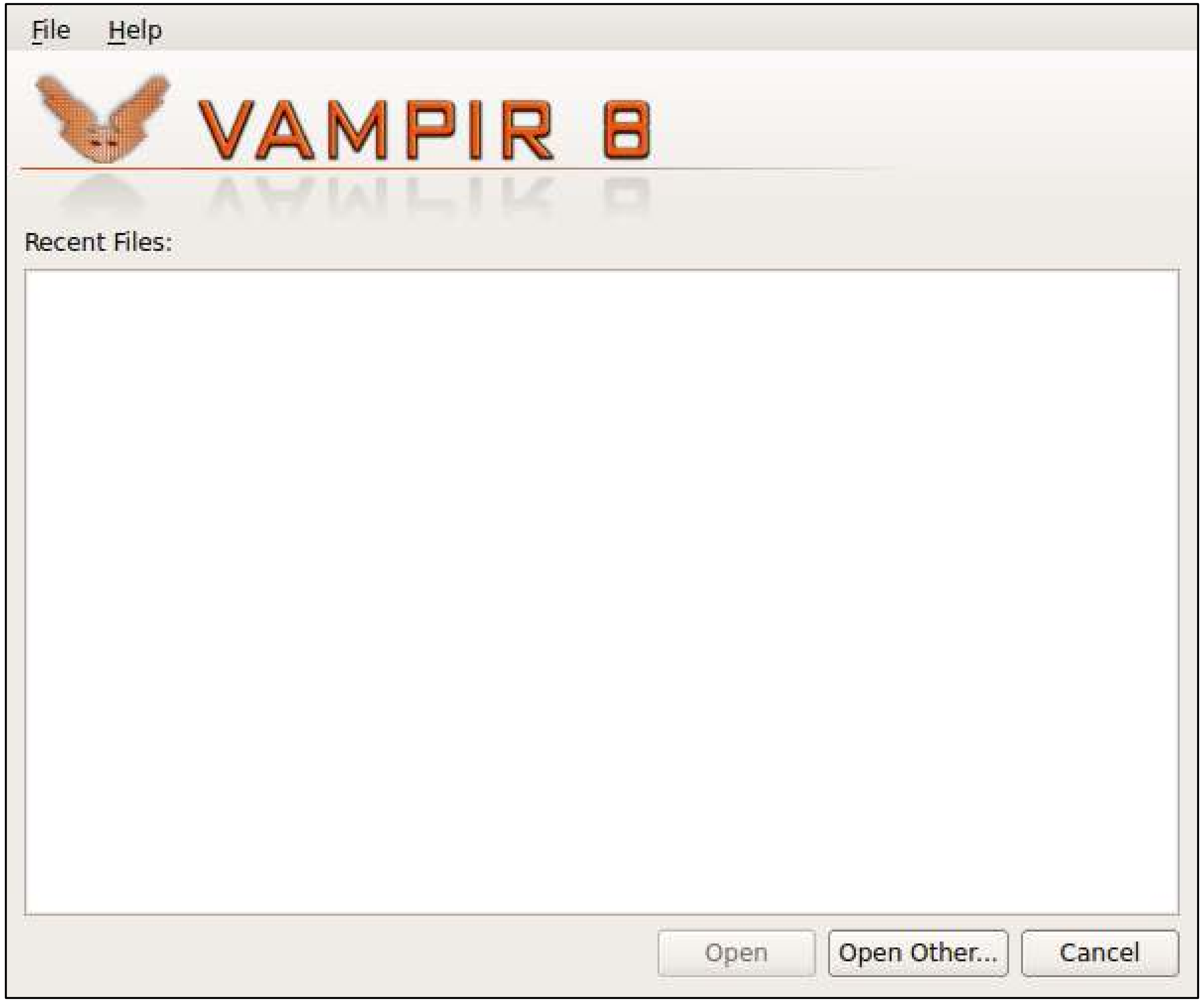
Start Vampir

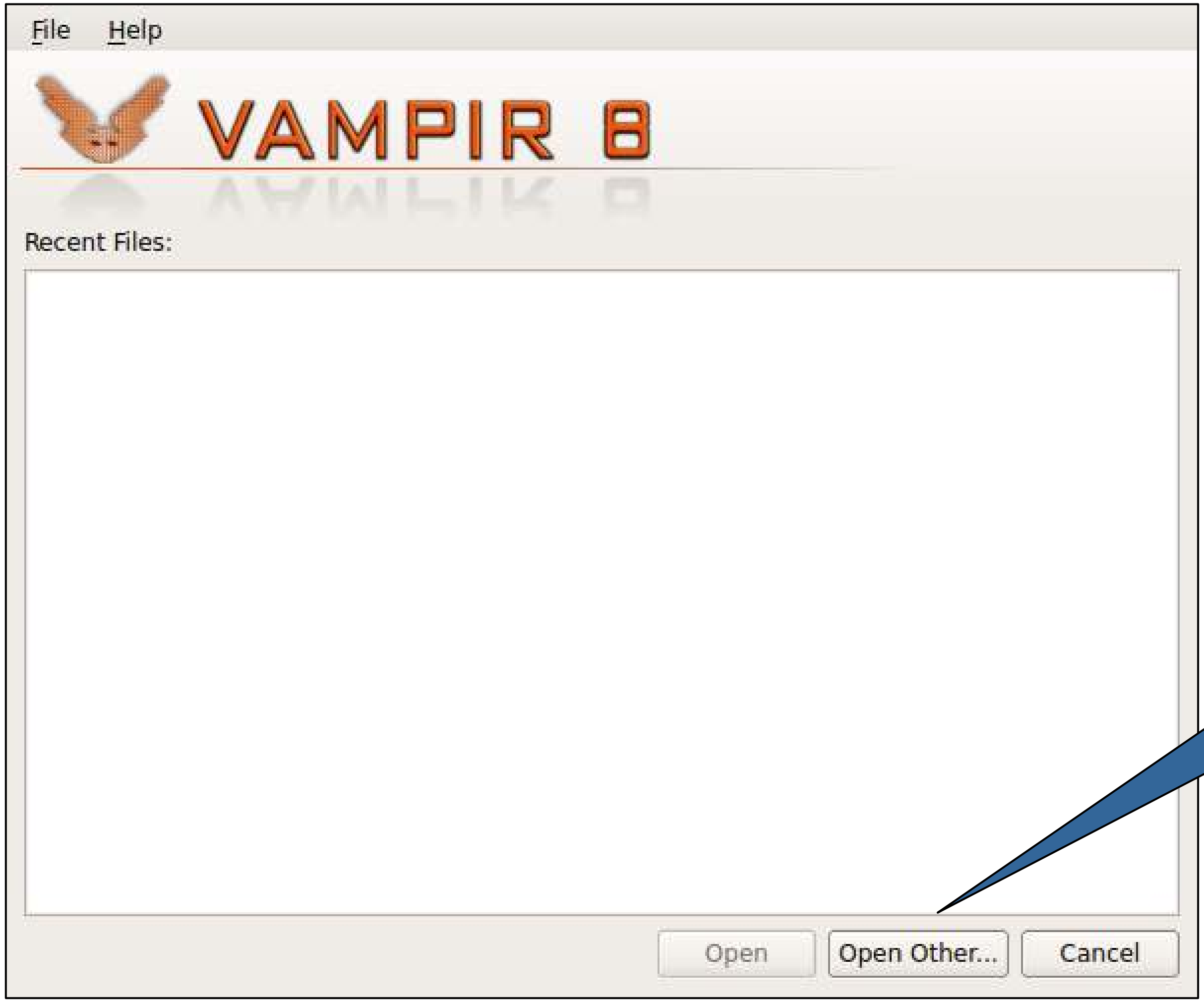
```
% ssh -N -L 30000:c401-602.stampede2.tacc.utexas.edu:30019 \
    stampede.tacc.utexas.edu
```

host:port from VampirServer output

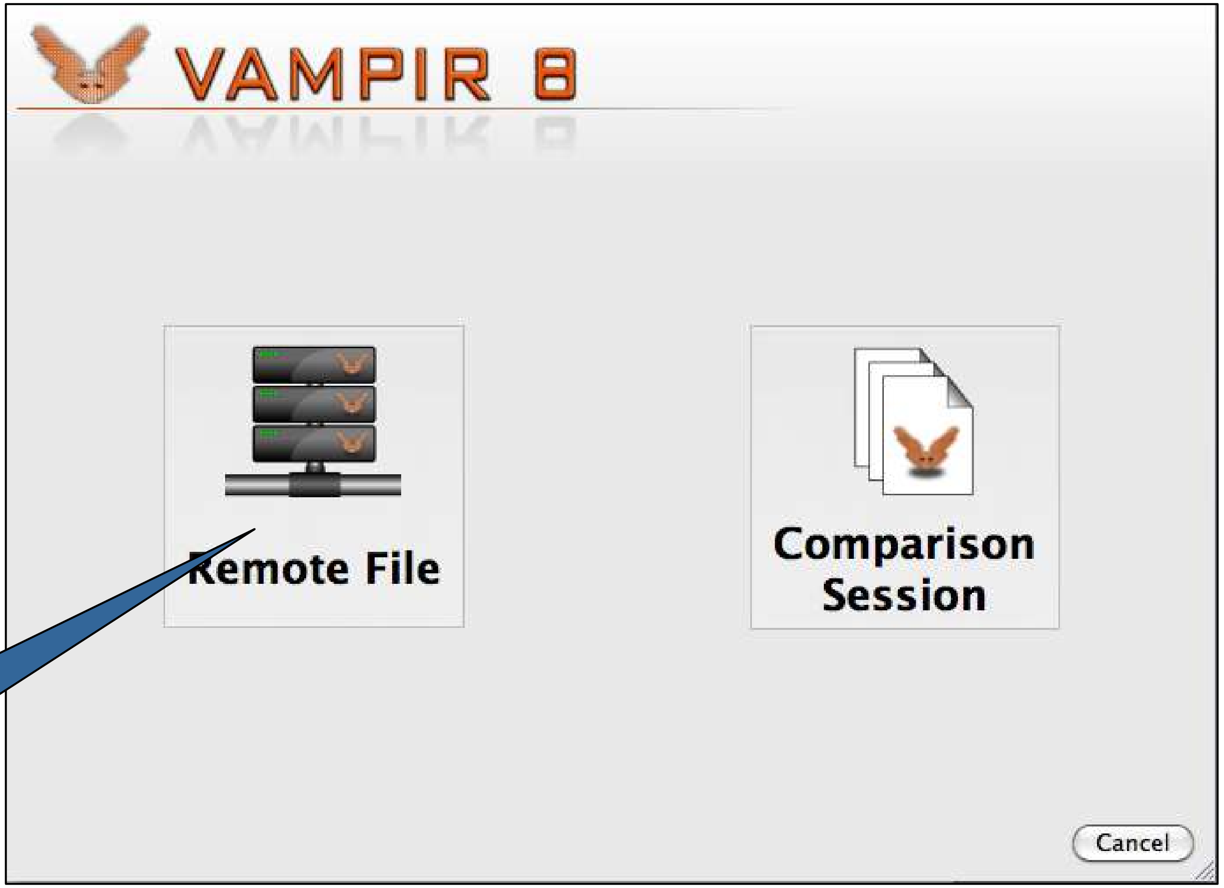
- Open a port forwarding to Stampede2 to be able to access the VampirServer

Start Vampir on local computer

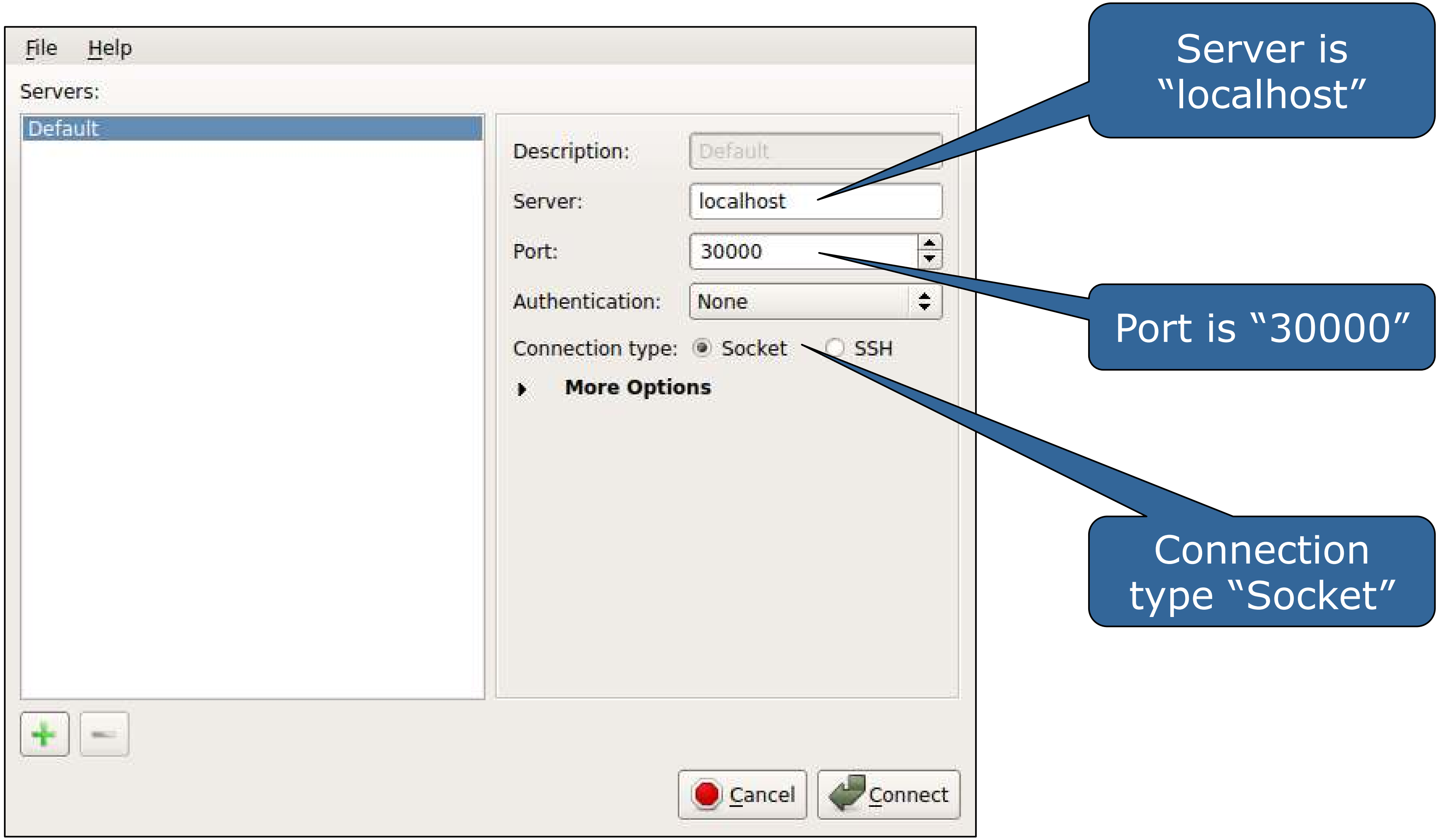


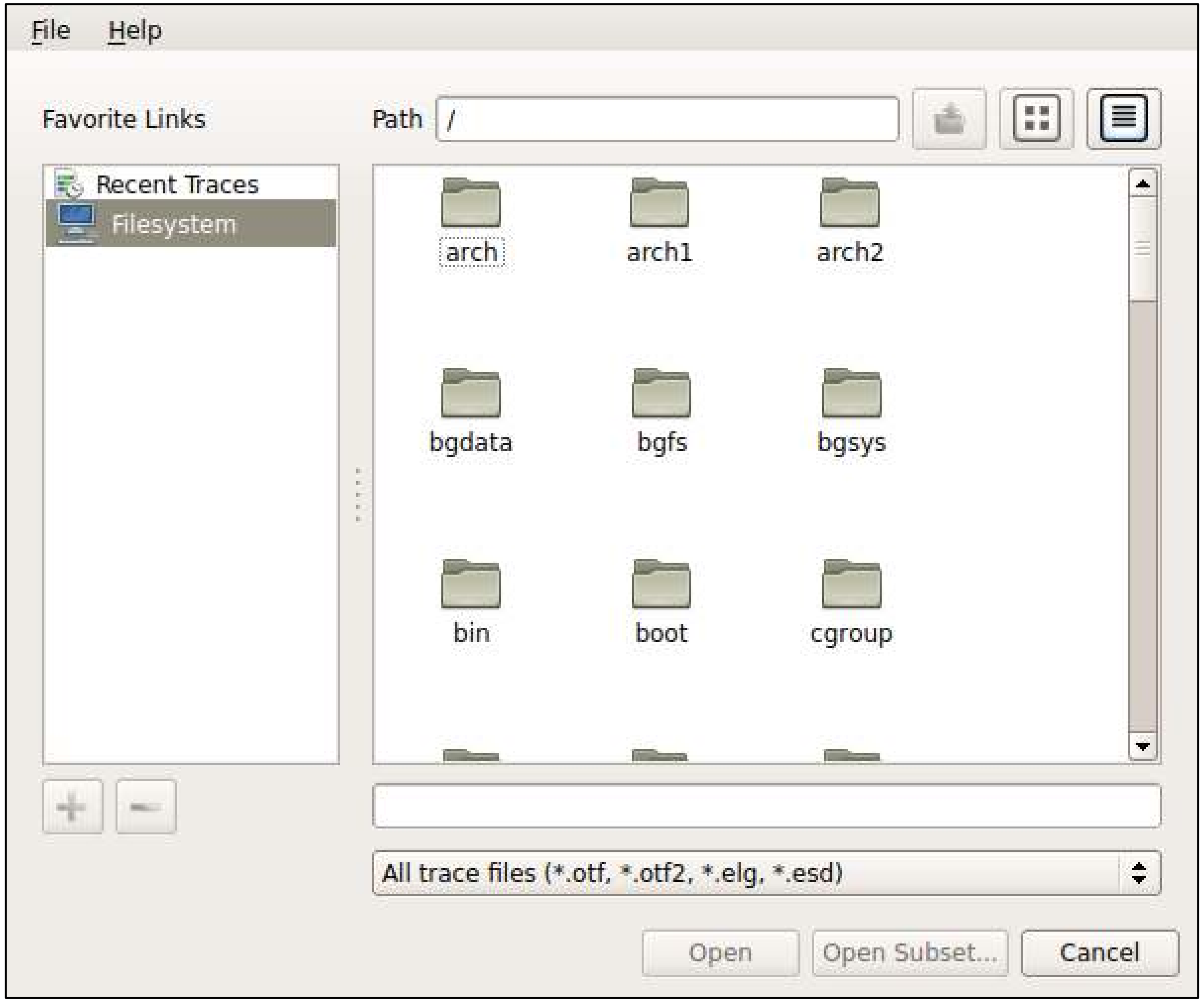


Use the "Open Other" option

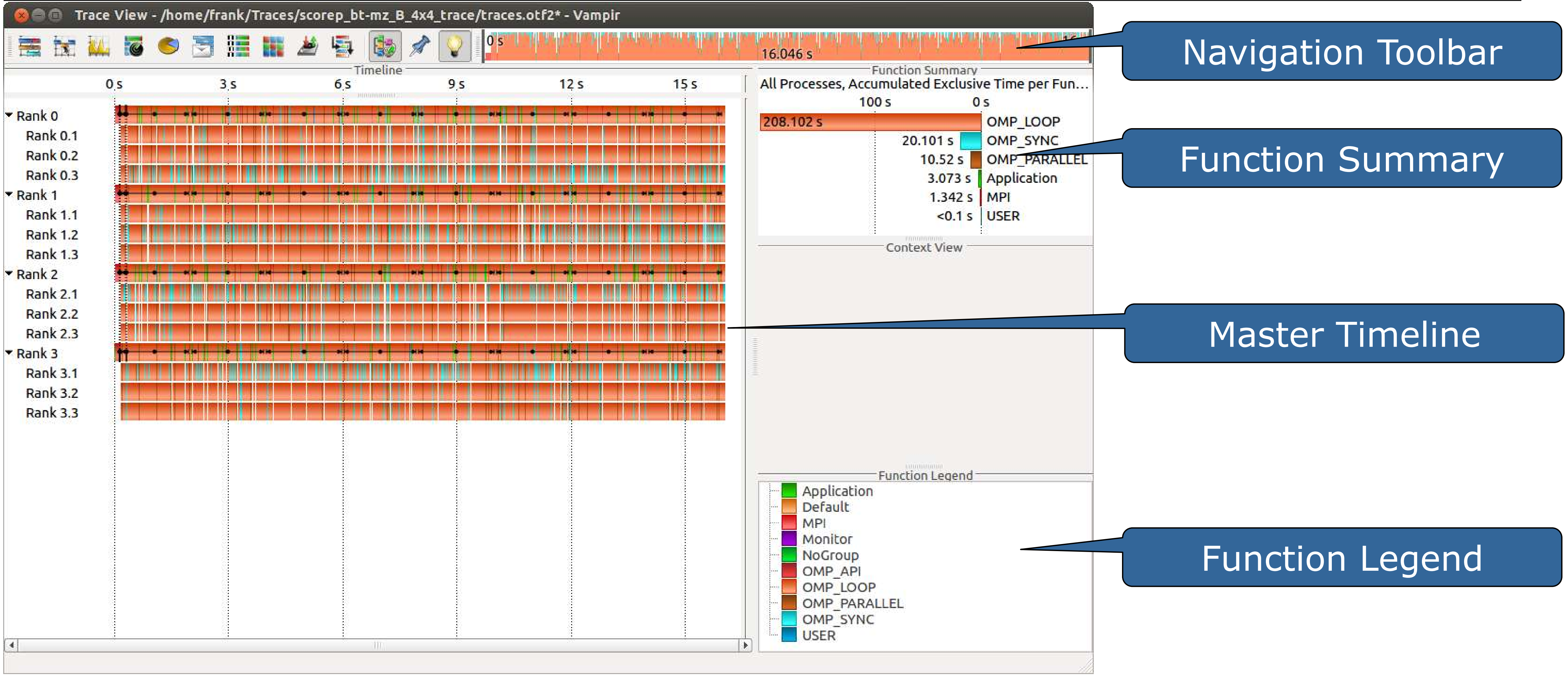


Select "Remote File"

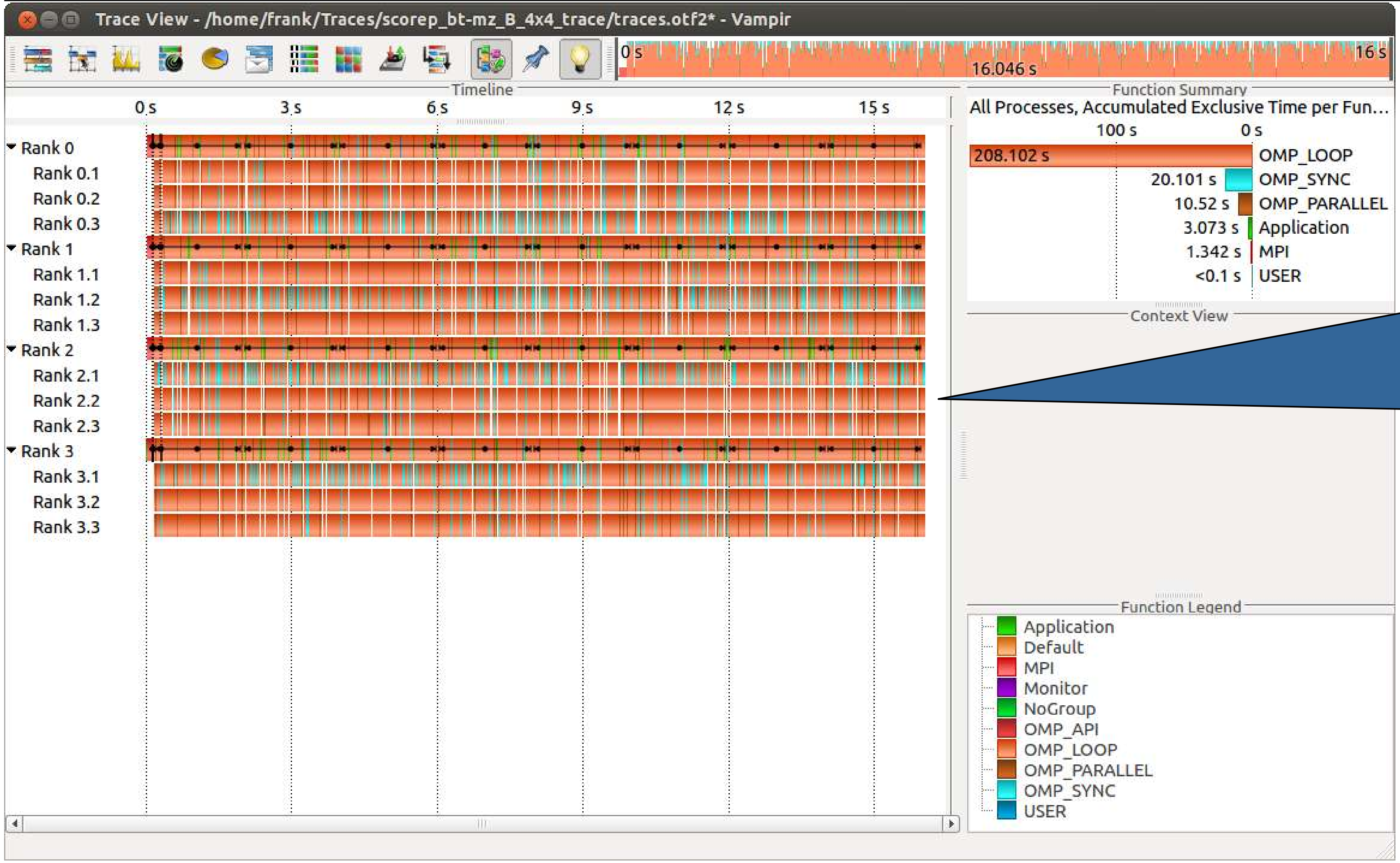

File
Help
Servers:
Default
Description:
Default
Server:
localhost
Port:
30000
Authentication:
None
Connection type:
Socket
SSH
More Options
Cancel
Connect
Server is "localhost"
Port is "30000"
Connection type "Socket"


File
Help
Favorite Links
Path /
Recent Traces
Filesystem
arch
arch1
arch2
bgdata
bgfs
bgsys
bin
boot
cgroup
All trace files (*.otf, *.otf2, *.elg, *.esd)
Open
Open Subset...
Cancel

Visualization of the NPB-MZ-MPI / BT trace

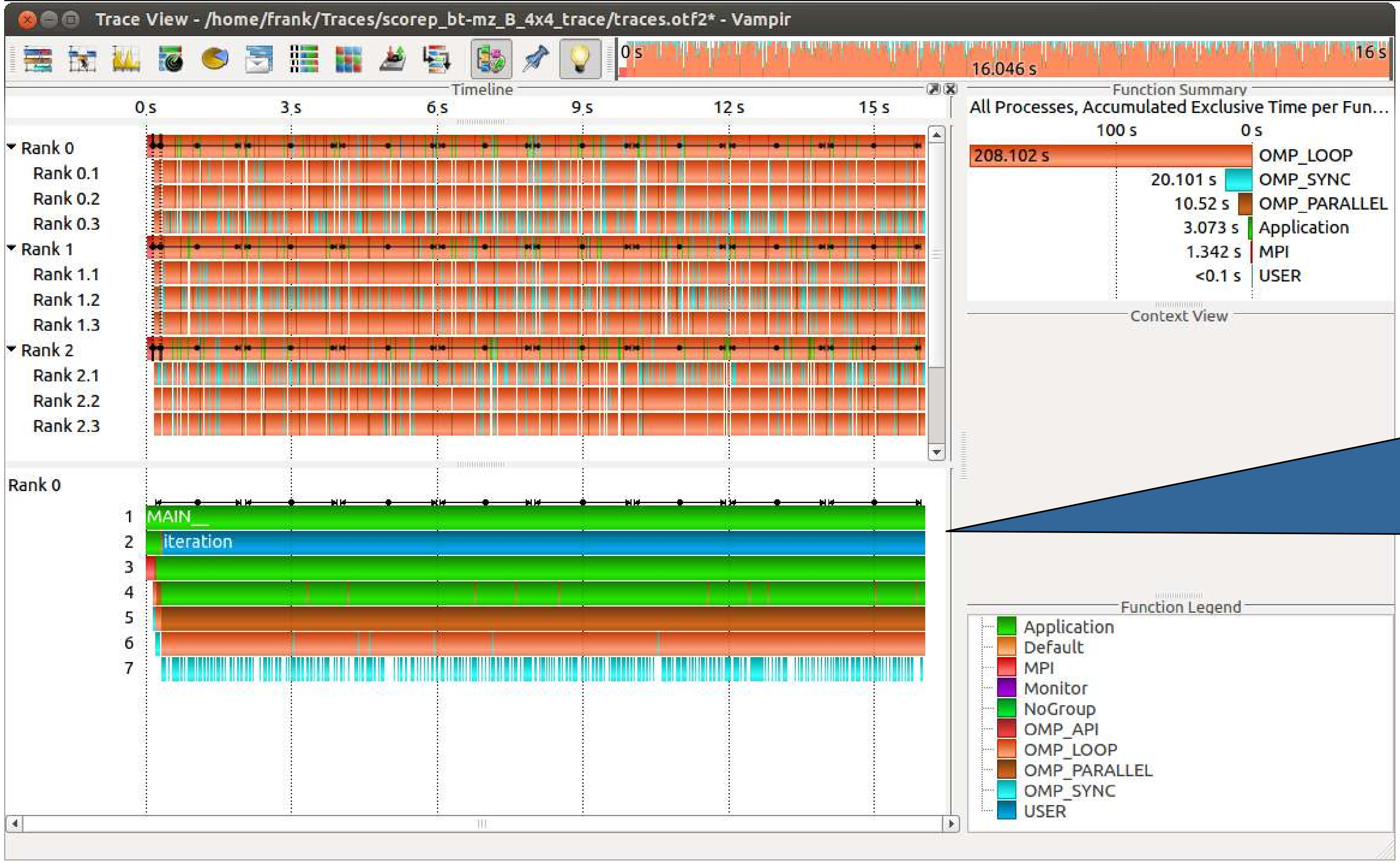


Visualization of the NPB-MZ-MPI / BT trace
Master Timeline



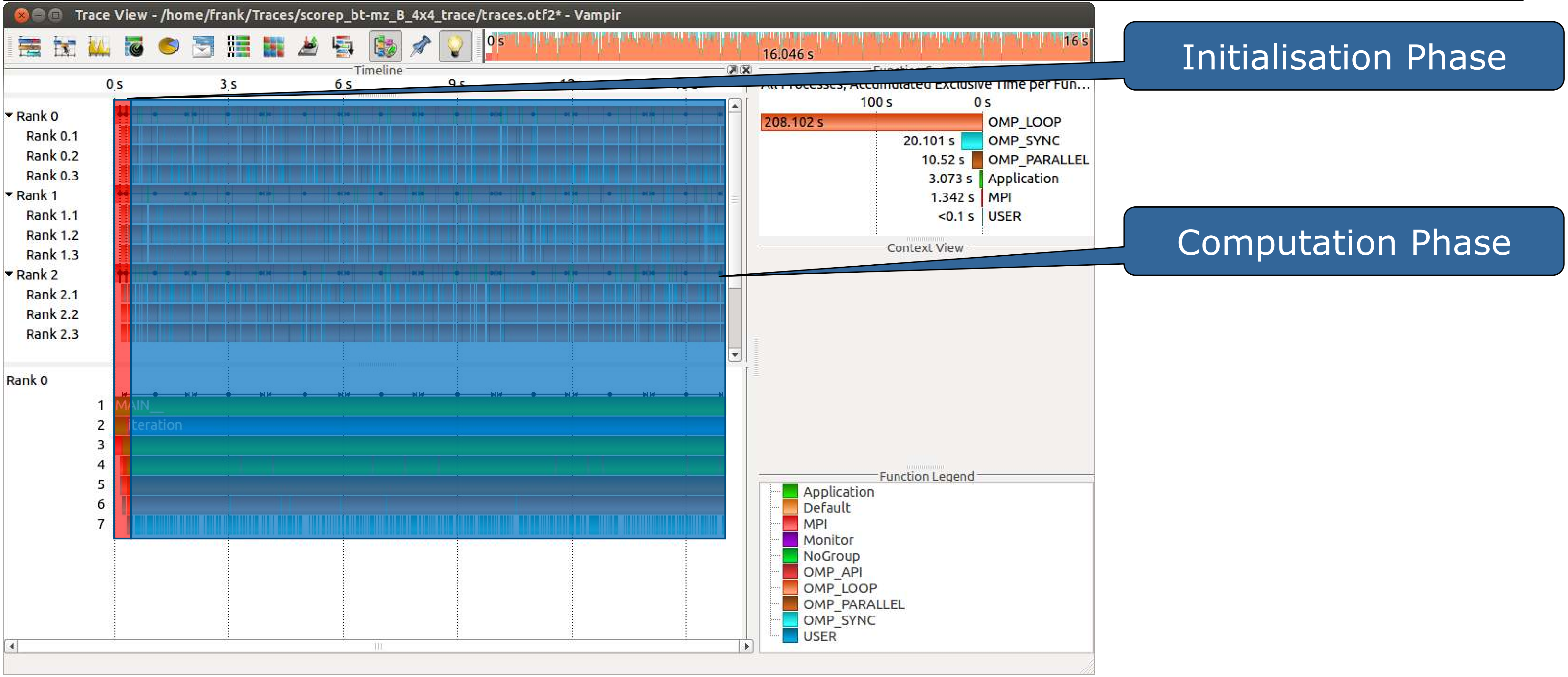
Detailed information about functions, communication and synchronization events for collection of processes.

Visualization of the NPB-MZ-MPI / BT trace
Process Timeline

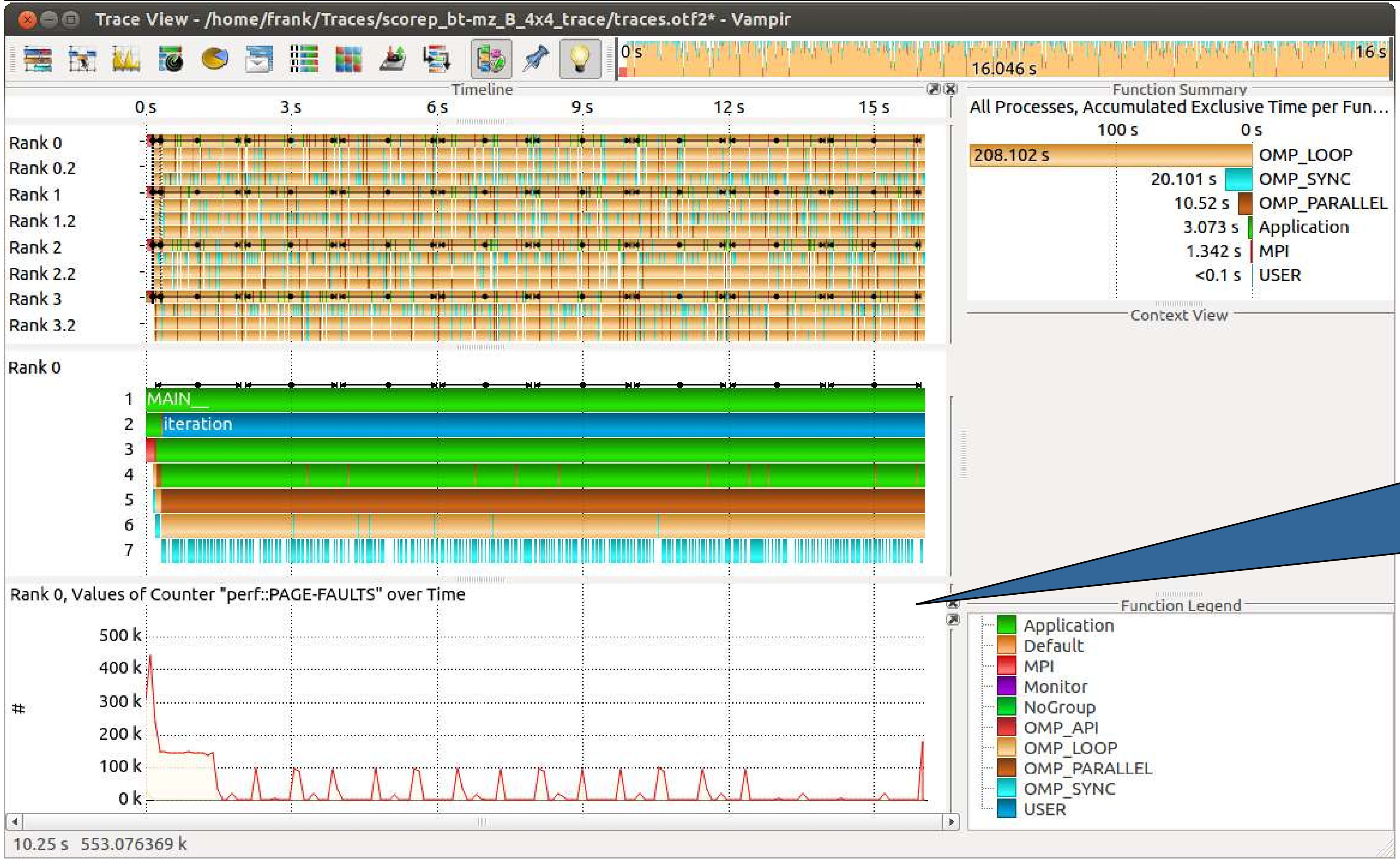


Detailed information about different levels of function calls in a stacked bar chart for an individual process.

Visualization of the NPB-MZ-MPI / BT trace
Typical program phases

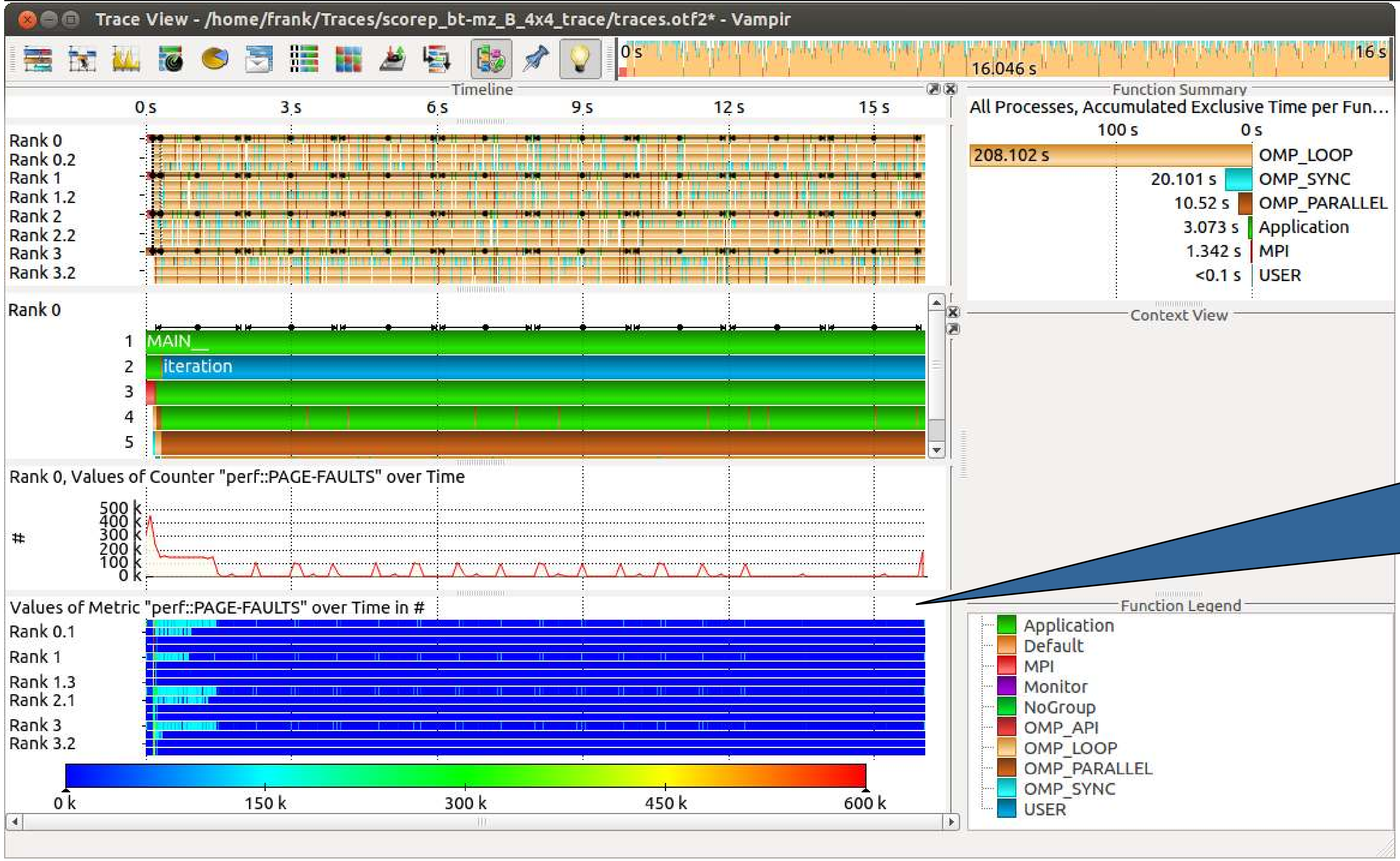


Visualization of the NPB-MZ-MPI / BT trace
Counter Data Timeline



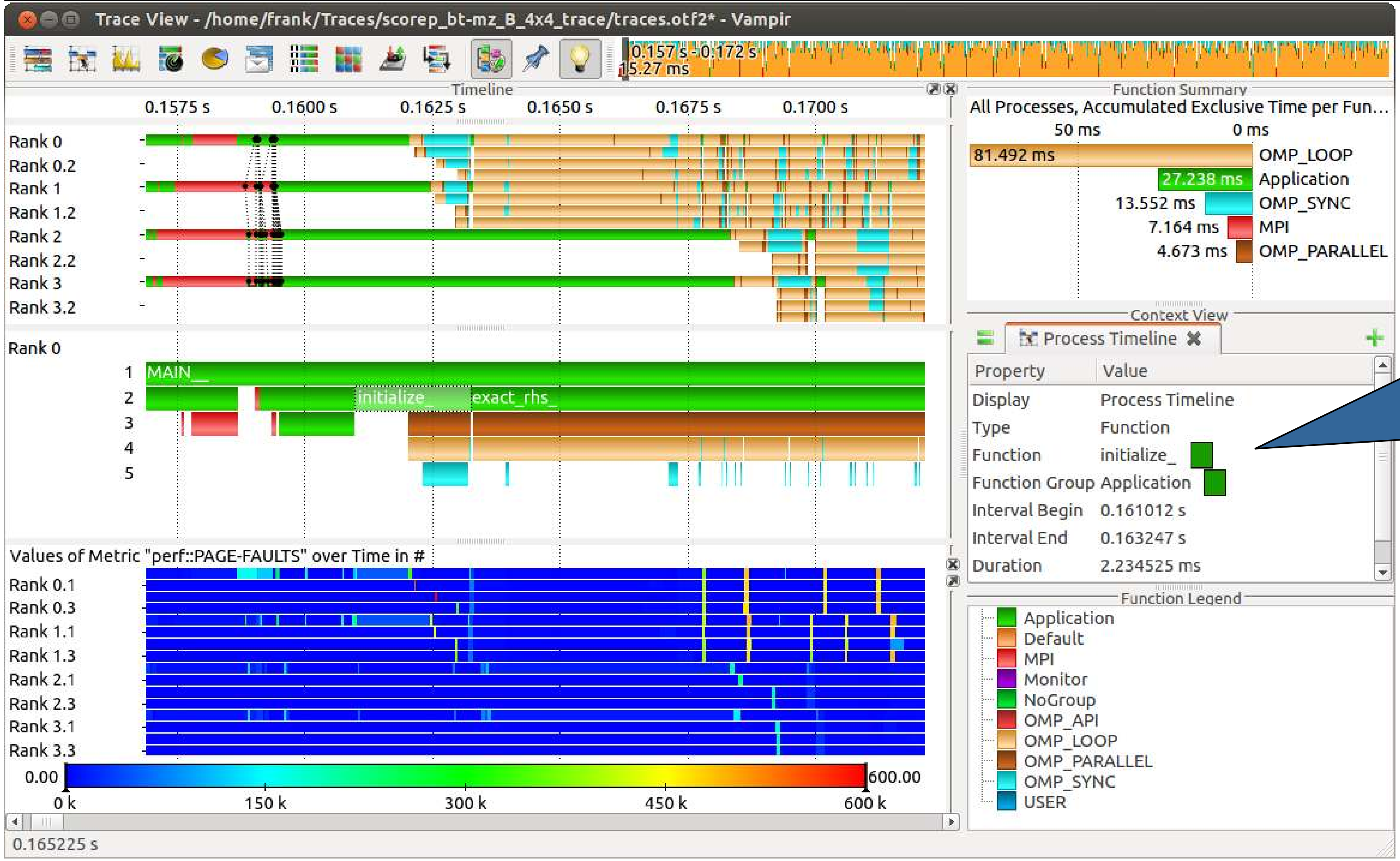
Detailed counter information over time for an individual process.

Visualization of the NPB-MZ-MPI / BT trace
Performance Radar



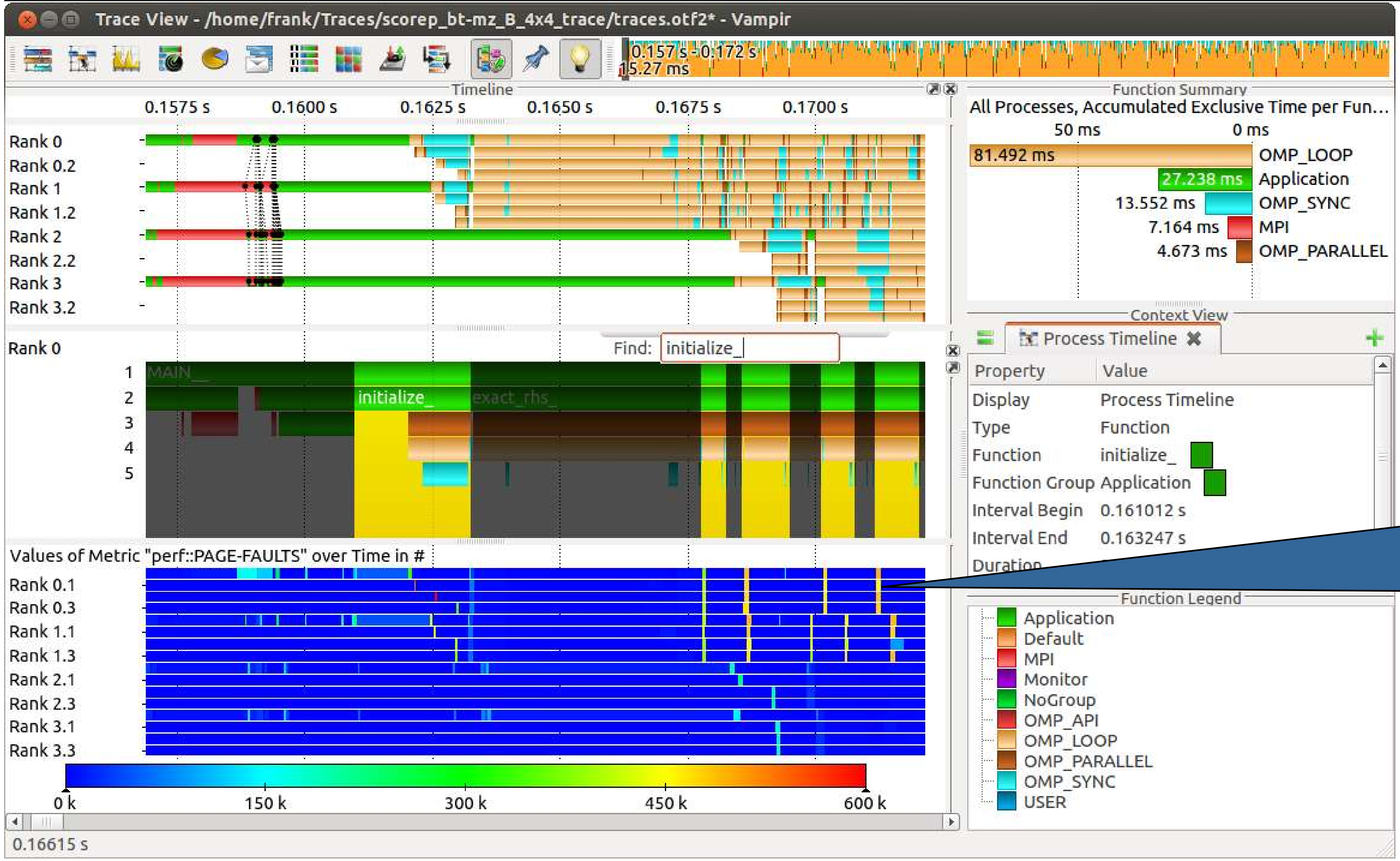
Detailed counter information over time for a collection of processes.

Visualization of the NPB-MZ-MPI / BT trace
Zoom in: Inititialisation Phase



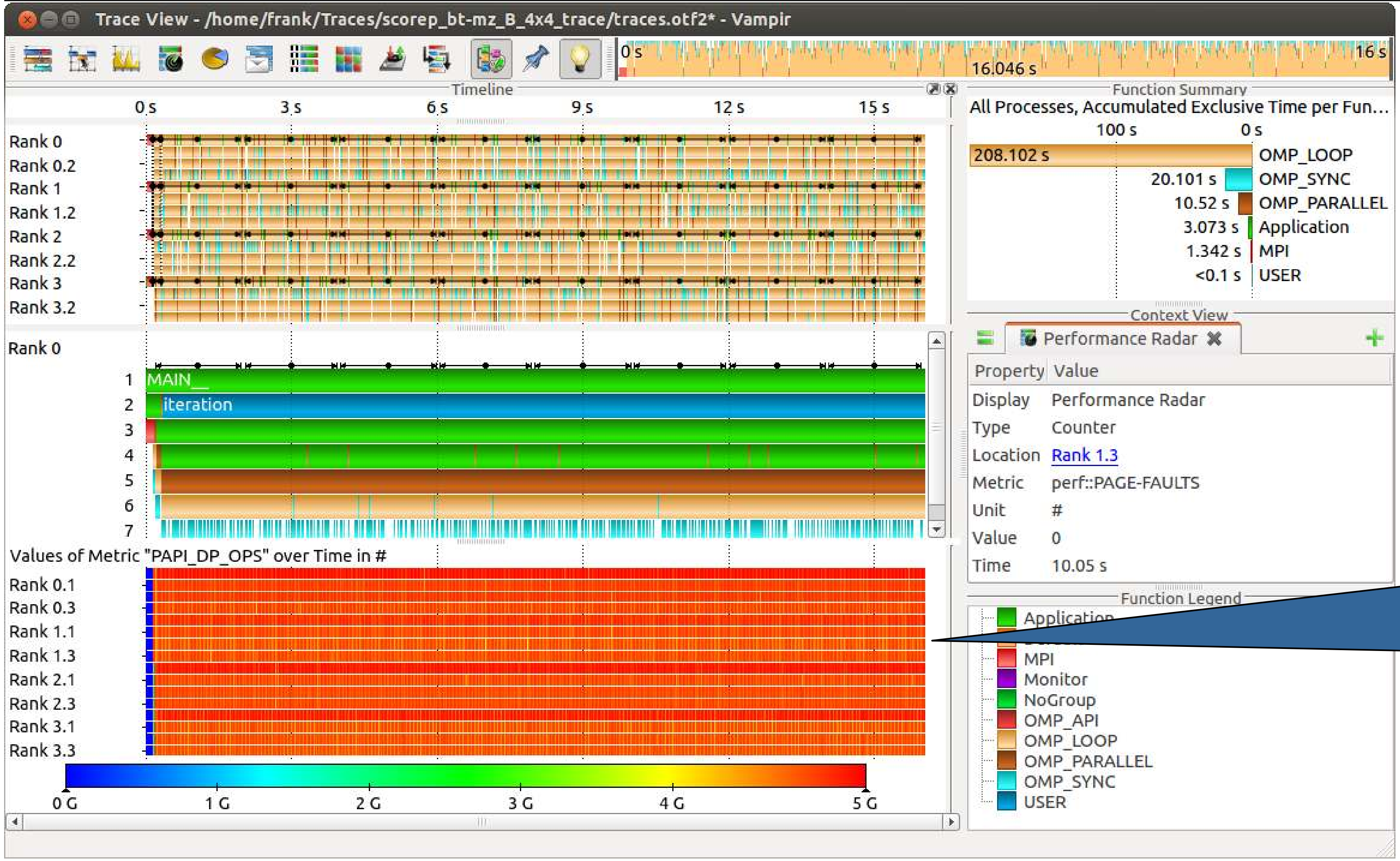
Context View: Detailed information about function "initialize_".

Visualization of the NPB-MZ-MPI / BT trace
Find Function



Execution of function "initialize_" results in higher page fault rates.

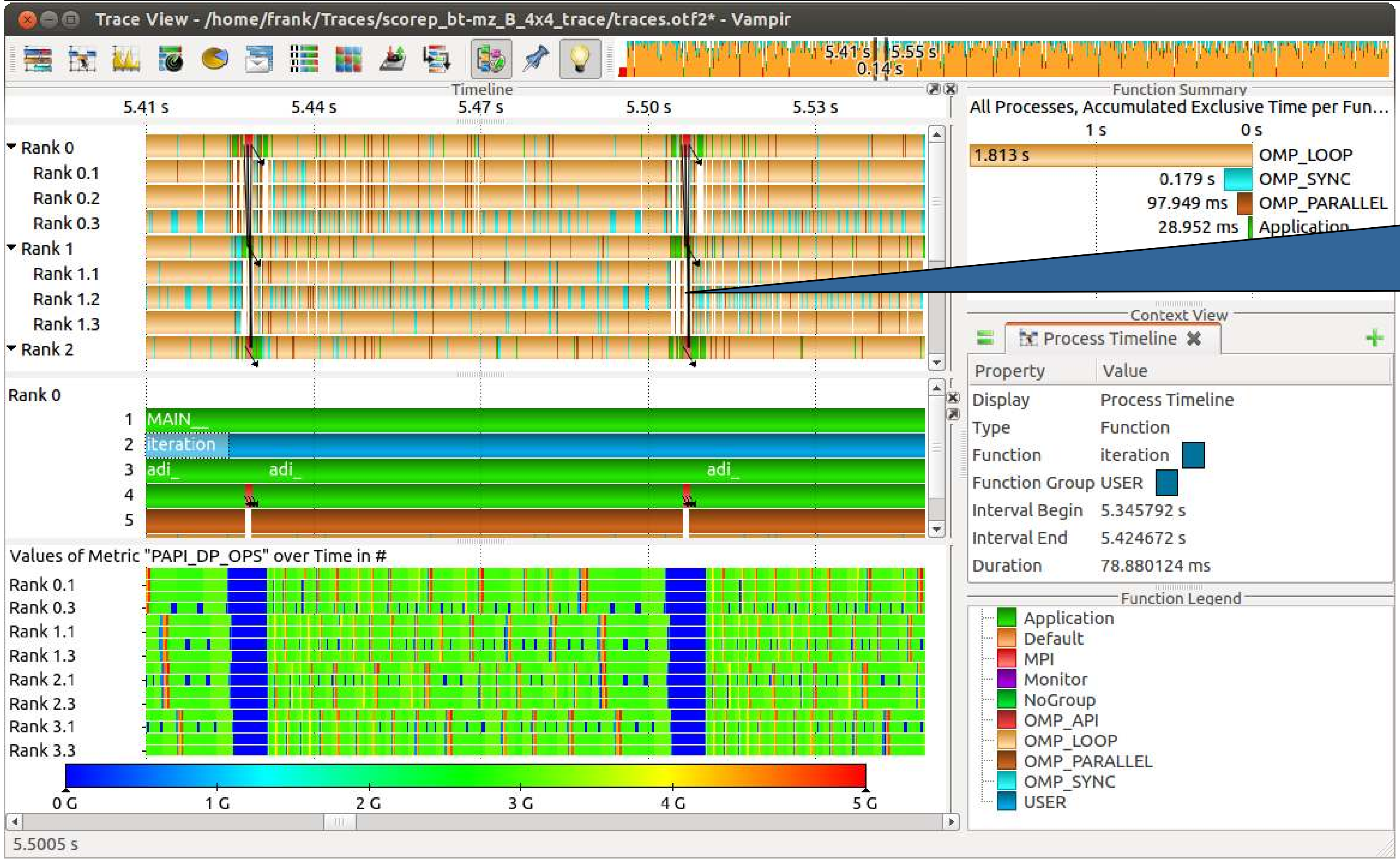
Visualization of the NPB-MZ-MPI / BT trace
Computation Phase



Computation phase results in higher floating point operations.

Visualization of the NPB-MZ-MPI / BT trace

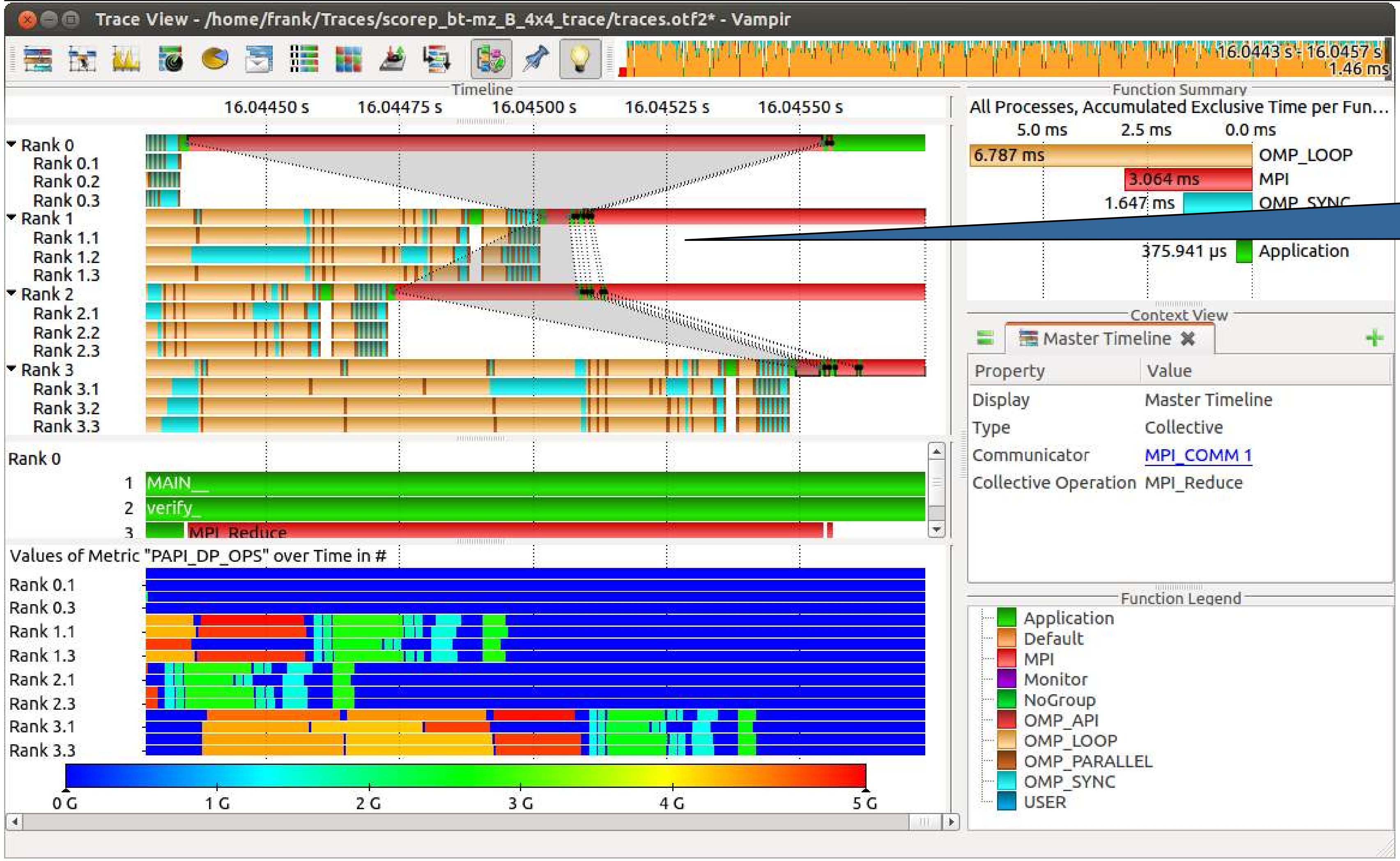
Zoom in: Computation Phase



MPI communication results in lower floating point operations.

Visualization of the NPB-MZ-MPI / BT trace

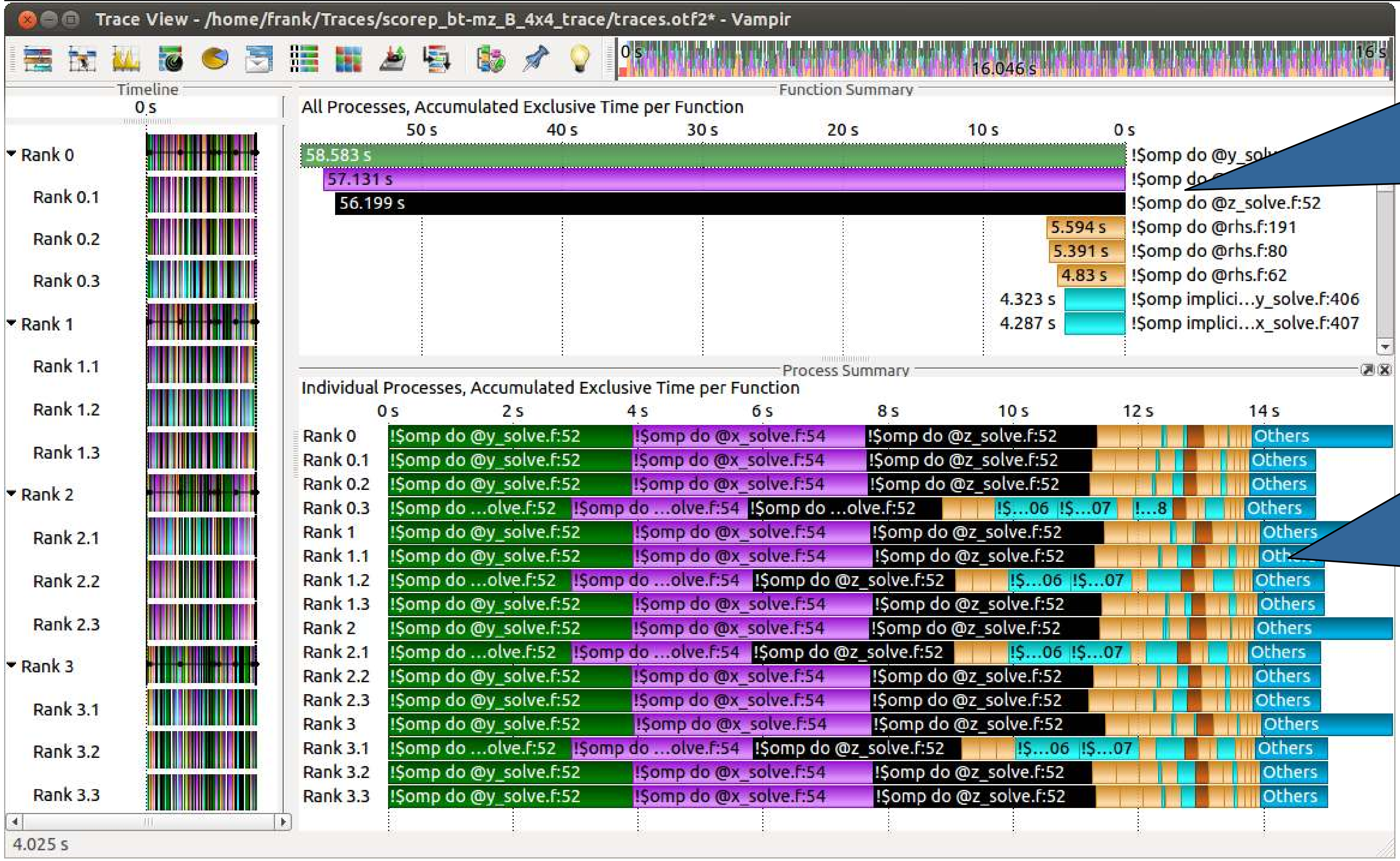
Zoom in: Finalisation Phase



"Early reduce" bottleneck.

Visualization of the NPB-MZ-MPI / BT trace

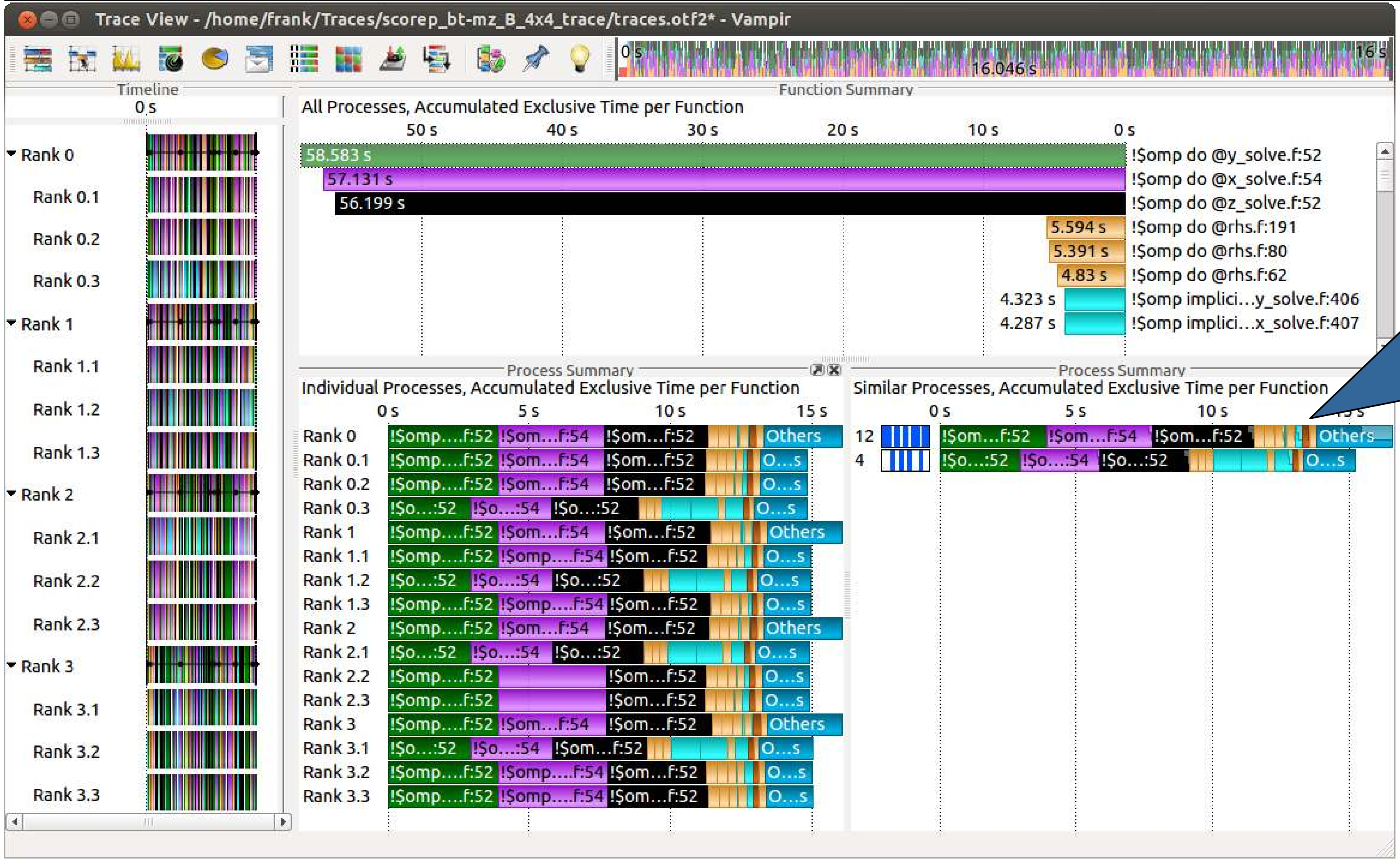
Process Summary



Function Summary: Overview of the accumulated information across all functions and for a collection of processes.

Process Summary: Overview of the accumulated information across all functions and for every process independently.

Visualization of the NPB-MZ-MPI / BT trace
Process Summary



Find groups of similar processes and threads by using summarized function information.

Summary and Conclusion

Summary

- Vampir & VampirServer
  - Interactive trace visualization and analysis
  - Intuitive browsing and zooming
  - Scalable to large trace data sizes (20 TiByte)
  - Scalable to high parallelism (200,000 processes)
- Vampir for Linux, Windows, and Mac OS X

INDIANA UNIVERSITY

SPONSORED BY THE
Federal Ministry of Education and Research

SILC

OAK RIDGE National Laboratory

LMAC

READEX
Runtime Exploitation of Application Dynamism for Energy-efficient eXascale computing

eeClust
Energy-Efficient Cluster Computing

nextgenio

Score-E

VAMPIR

European Commission
Horizon 2020 European Union funding for Research & Innovation

Microsoft

ITEA 2
INFORMATION TECHNOLOGY FOR EUROPEAN ADVANCEMENT

SEVENTH FRAMEWORK PROGRAMME

H4H

ParMA

http://www.vampir.eu

vampirsupport@zih.tu-dresden.de

Score-P – A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir

VI-HPS Team

Score-P
Scalable performance measurement infrastructure for parallel codes

Score-P: Specialized Measurements and Analyses

Score-P

Mastering build systems

- Hooking up the Score-P instrumenter `scorep` into complex build environments like *Autotools* or *CMake* was always challenging
- Score-P provides new convenience wrapper scripts to simplify this (since Score-P 2.0)
- *Autotools* and *CMake* need the used compiler already in the *configure step,* but instrumentation should not happen in this step, only in the *build step*

```
% SCOREP_WRAPPER=off \
> cmake .. \
> -DCMAKE_C_COMPILER=scorep-icc \
> -DCMAKE_CXX_COMPILER=scorep-icpc
```

Disable instrumentation in the *configure step*

Specify the wrapper scripts as the compiler to use

- Allows to pass addition options to the Score-P instrumenter and the compiler via environment variables without modifying the *Makefiles*
- Run `scorep-wrapper --help` for a detailed description and the available wrapper scripts of the Score-P installation

Mastering C++ applications

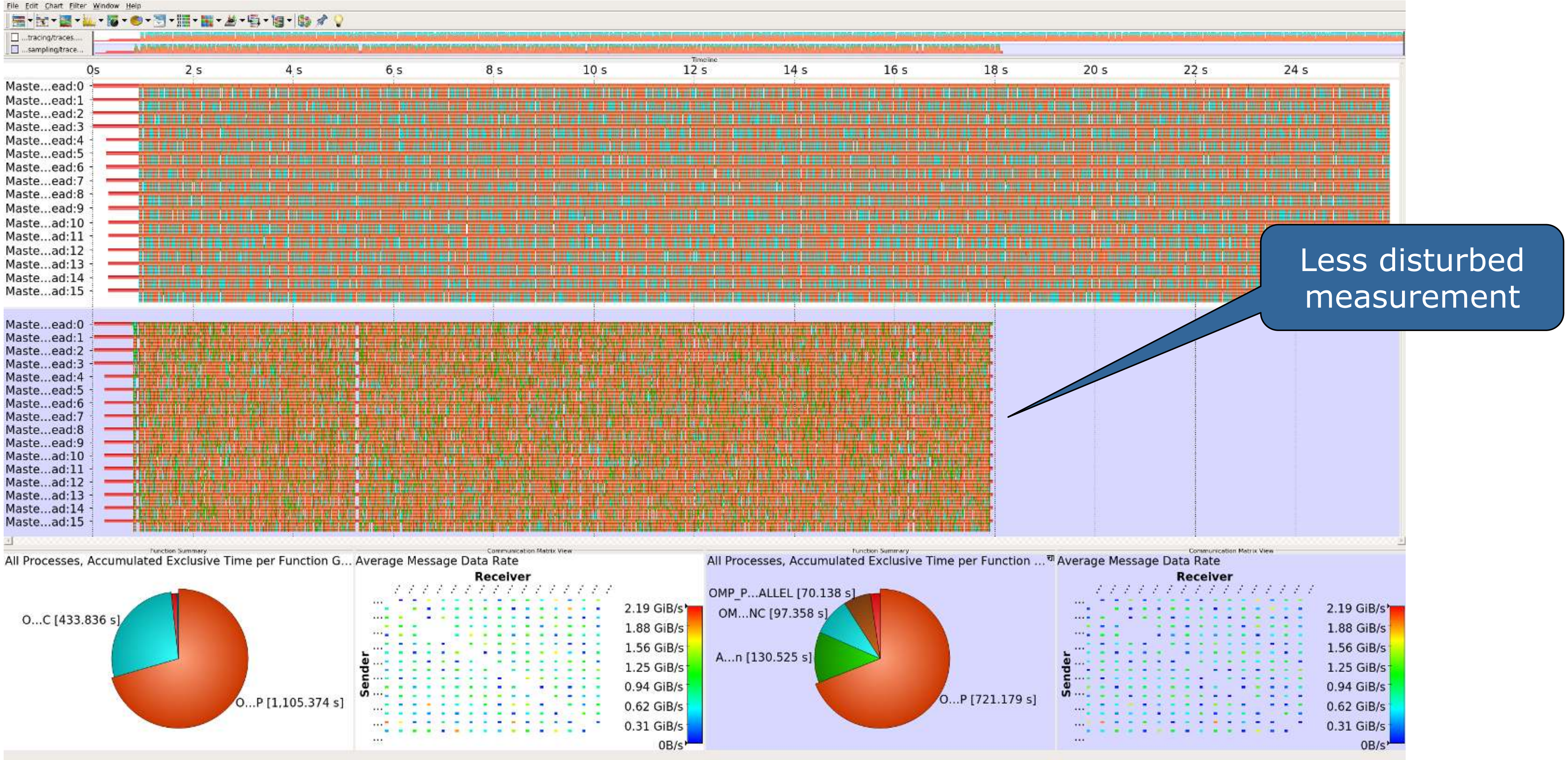
- Automatic compiler instrumentation greatly disturbs C++ applications because of frequent/short function calls => Use sampling instead
- Novel combination of sampling events and instrumentation of MPI, OpenMP, …
  - Sampling replaces compiler instrumentation (instrument with `--nocompiler` to further reduce overhead) => Filtering not needed anymore
  - Instrumentation is used to get accurate times for parallel activities to still be able to identifies patterns of inefficiencies
- Supports profile and trace generation

```
% export SCOREP_ENABLE_UNWINDING=true
% # use the default sampling frequency
% #export SCOREP_SAMPLING_EVENTS=perf_cycles@2000000

% OMP_NUM_THREADS=4 mpiexec -np 4 ./bt-mz_W.4
```

- Set new configuration variable to enable sampling

- Available since Score-P 2.0, only x86-64 supported currently

Mastering C++ applications



Wrapping calls to 3rd party libraries

- Enables users to install library wrappers for any C/C++ library
- Intercept calls to a library API
  - no need to either build the library with Score-P or add manual instrumentation to the application using the library
  - no need to access the source code of the library, header and library files suffice
- Score-P needs to be executed with `--libwrap=…`
- Execute `scorep-libwrap-init` for directions:

Step 1: Initialize the working directory
Step 2: Add library headers
Step 3: Create a simple example application
Step 4: Further configure the build parameters
Step 5: Build the wrapper
Step 6: Verify the wrapper
Step 7: Install the wrapper
Step 8: Verify the installed wrapper

Only once

Often

Step 9: Use the wrapper

Wrapping calls to 3rd party libraries

- Generate your own library wrappers by telling `scorep-libwrap-init` how you would compile and link an application, e.g. using FFTW

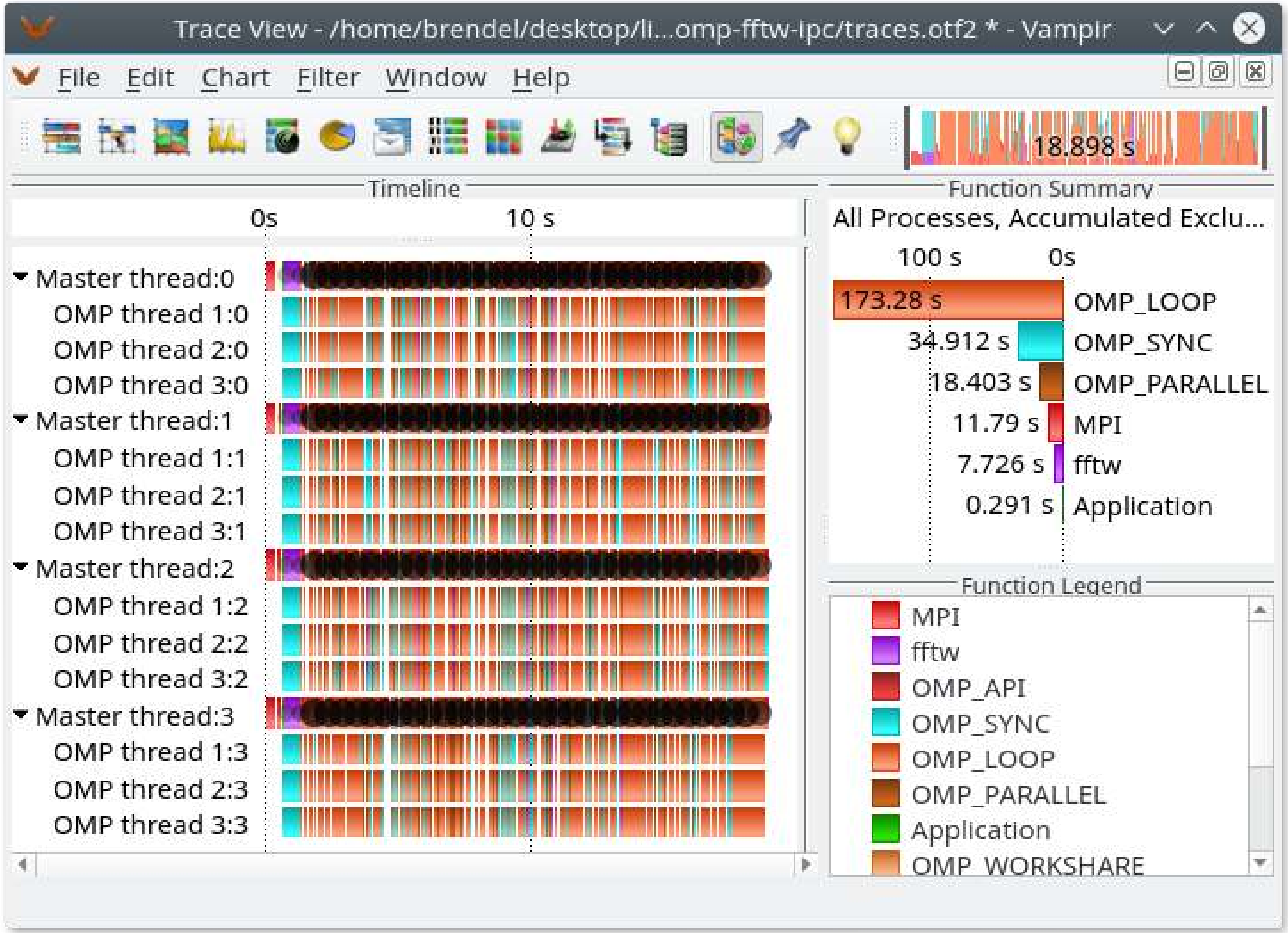
```
% scorep-libwrap-init           \
>   --name=fftw                 \
>   --prefix=$PREFIX            \
>   -x c                        \
>   --cppflags="-O3 -DNDEBUG -openmp -I$FFTW_INC" \
>   --ldflags="-L$FFTW_LIB"     \
>   --libs="-lfftw3f -lfftw3" \
>   working_directory
```

- Generate and build wrapper

```
% cd working_directory
% ls                    # (Check README.md for instructions)
% make                  # Generate and build wrapper
% make check            # See if header analysis matches symbols
% make install          #
% make installcheck     # More checks: Linking etc.
```

Wrapping calls to 3rd party libraries

- MPI + OpenMP
- Calls to FFTW library



Mastering application memory usage

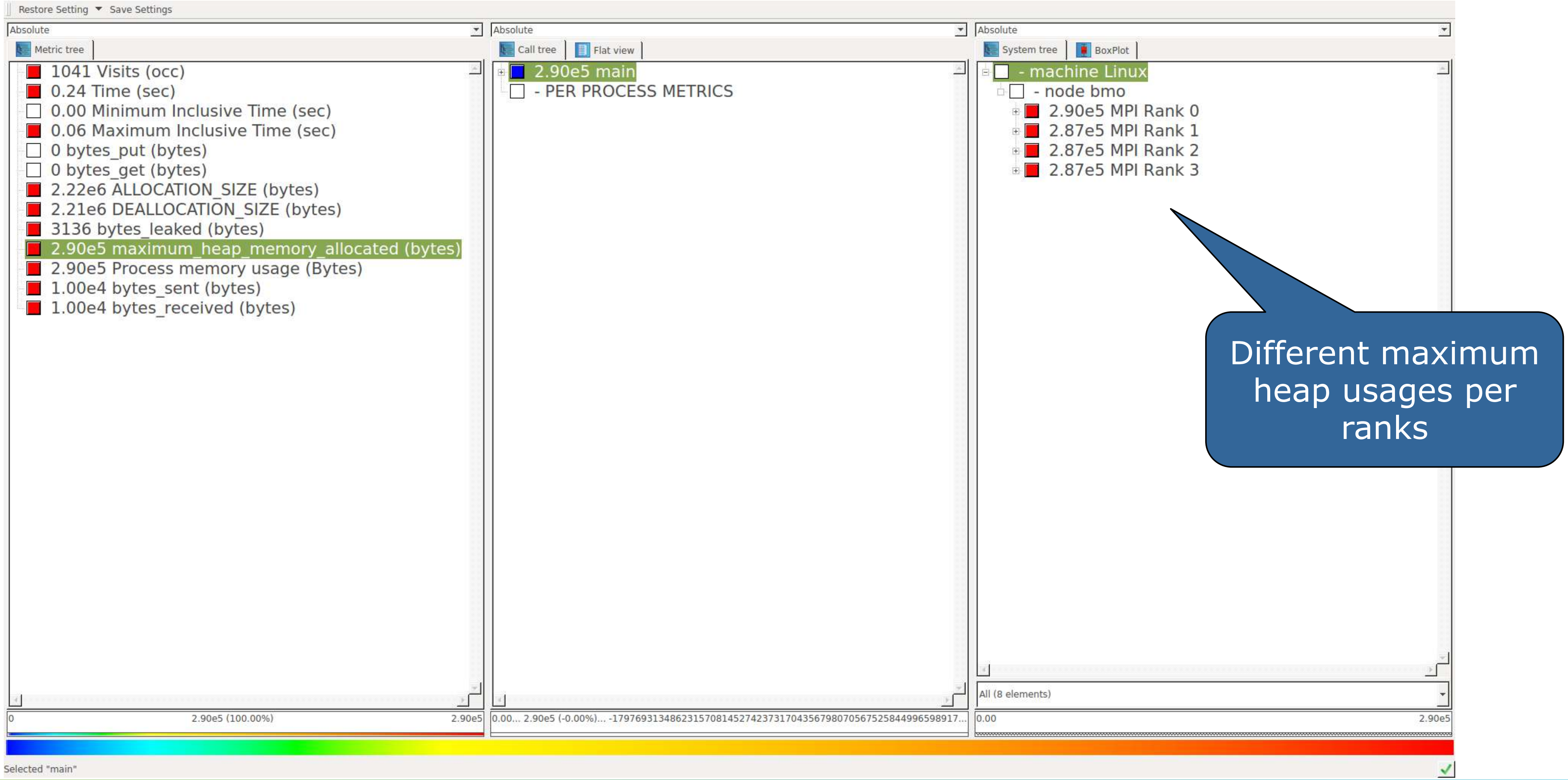
- Determine the maximum heap usage per process
- Find high frequent small allocation patterns
- Find memory leaks
- Support for:
  - C, C++, MPI, and SHMEM (Fortran only for GNU Compilers)
  - Profile and trace generation (profile recommended)
    - Memory leaks are recorded only in the profile
    - Resulting traces are not supported by Scalasca yet

```
% export SCOREP_MEMORY_RECORDING=true
% export SCOREP_MPI_MEMORY_RECORDING=true

% OMP_NUM_THREADS=4 mpiexec -np 4 ./bt-mz_W.4
```

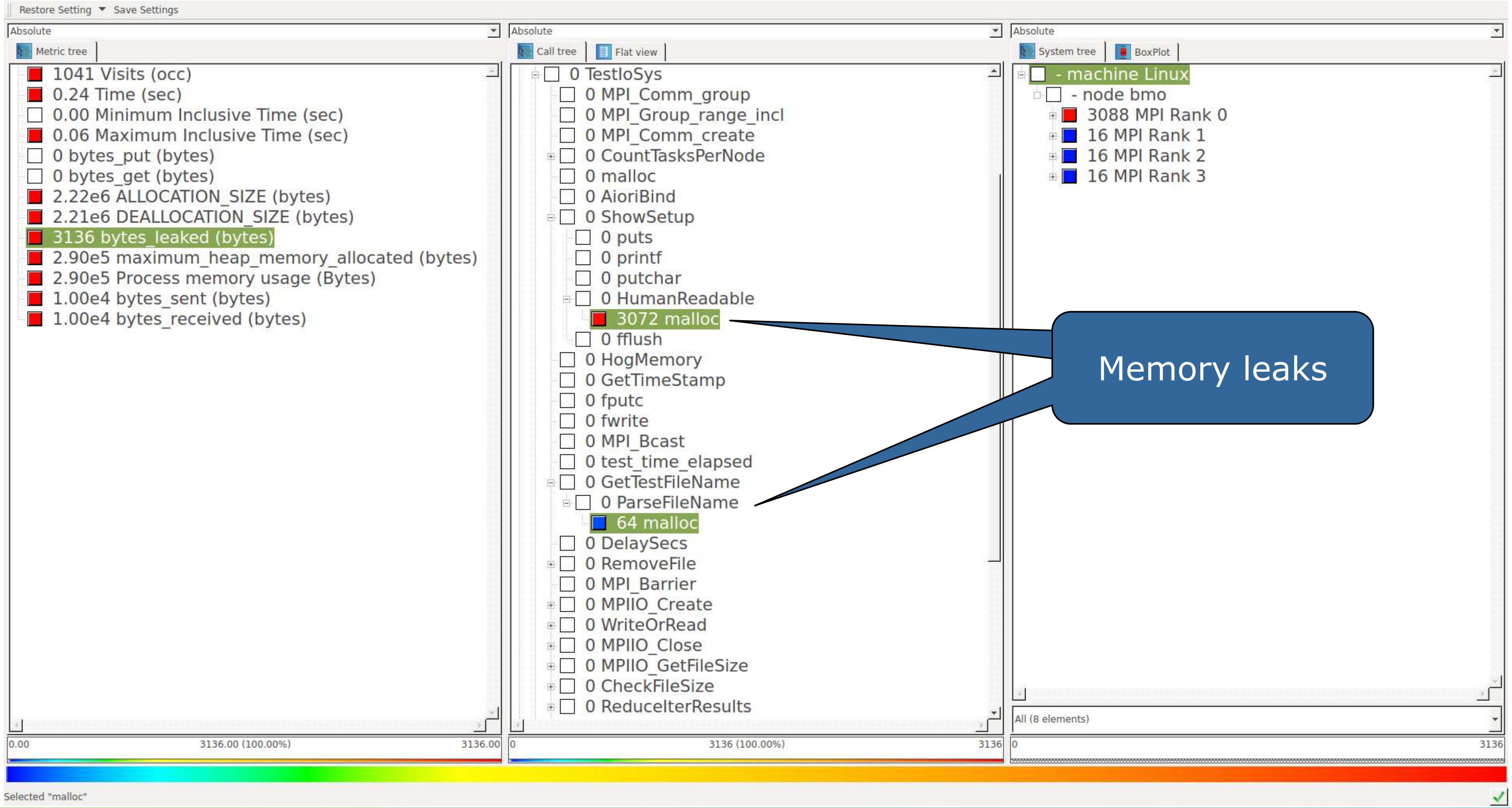
- Set new configuration variable to enable memory recording

- Available since Score-P 2.0

Mastering application memory usage



Mastering application memory usage



Mastering heterogeneous applications

- Record CUDA applications and device activities

```
% export SCOREP_CUDA_ENABLE=gpu,kernel,idle
```

- Record OpenCL applications and device activities

```
% export SCOREP_OPENCL_ENABLE=api,kernel
```

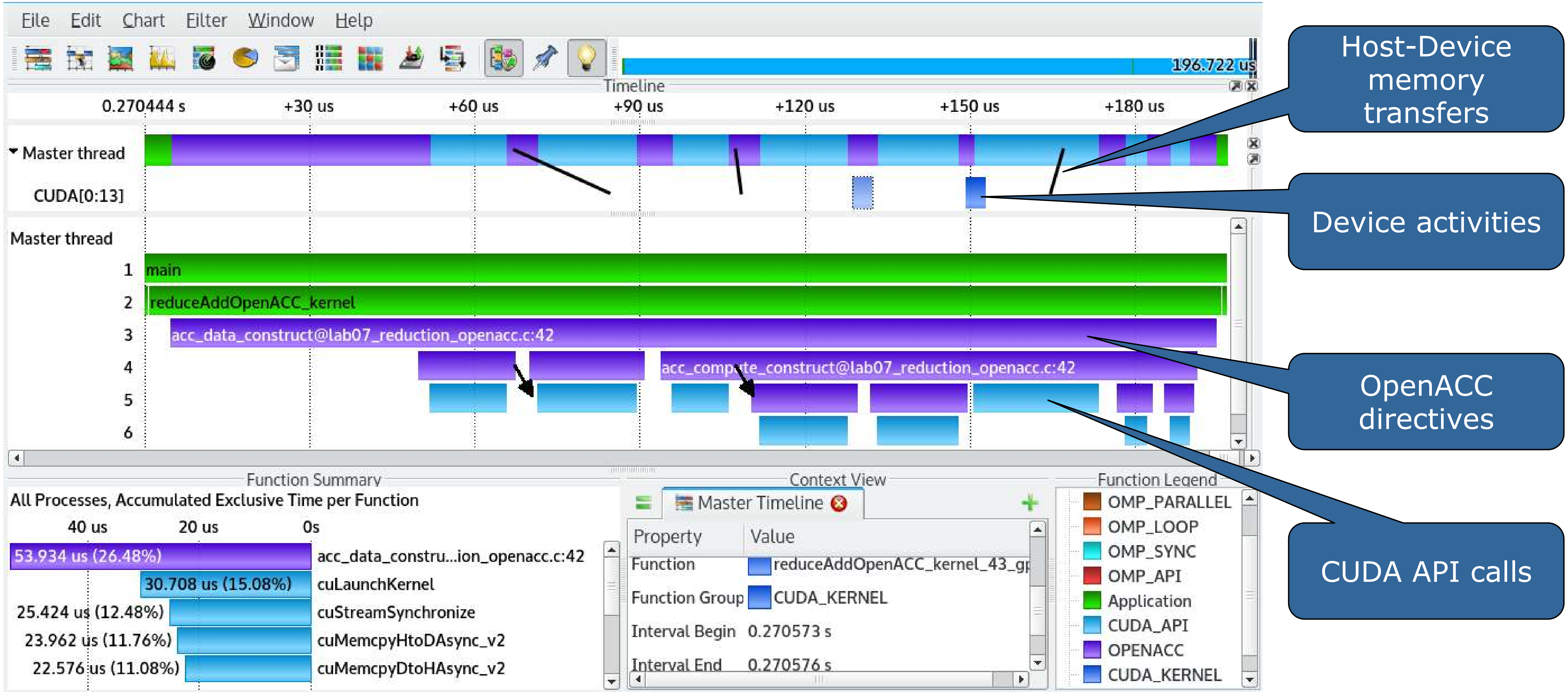
- Record OpenACC applications

```
% export SCOREP_OPENACC_ENABLE=yes
```

  - Can be combined with CUDA if it is a NVIDIA device

```
% export SCOREP_CUDA_ENABLE=kernel
```

Mastering heterogeneous applications



Enriching measurements with performance counters

- Record metrics from PAPI:

```
% export SCOREP_METRIC_PAPI=PAPI_TOT_CYC
% export SCOREP_METRIC_PAPI_PER_PROCESS=PAPI_L3_TCM
```

  - Use PAPI tools to get available metrics and valid combinations:

```
% papi_avail
% papi_native_avail
```

- Record metrics from Linux perf:

```
% export SCOREP_METRIC_PERF=cpu-cycles
% export SCOREP_METRIC_PERF_PER_PROCESS=LLC-load-misses
```

  - Use the `perf` tool to get available metrics and valid combinations:

```
% perf list
```

- Write your own metric plugin
  - Repository of available plugins: https://github.com/score-p

Only the master thread records the metric (assuming all threads of the process access the same L3 cache)

Score-P user instrumentation API

- No replacement for automatic compiler instrumentation
- Can be used to further subdivide functions
  - E.g., multiple loops inside a function
- Can be used to partition application into coarse grain phases
  - E.g., initialization, solver, & finalization
- Enabled with `--user` flag to Score-P instrumenter
- Available for Fortran / C / C++

Score-P user instrumentation API (Fortran)

```
#include "scorep/SCOREP_User.inc"

subroutine foo(…)
  ! Declarations
  SCOREP_USER_REGION_DEFINE( solve )

  ! Some code…
  SCOREP_USER_REGION_BEGIN( solve, "<solver>", \
                            SCOREP_USER_REGION_TYPE_LOOP )
  do i=1,100
    [...]
  end do
  SCOREP_USER_REGION_END( solve )
  ! Some more code…
end subroutine
```

- Requires processing by the C preprocessor
  - For most compilers, this can be automatically achieved by having an uppercase file extension, e.g., `main.F` or `main.F90`

Score-P user instrumentation API (C/C++)

```
#include "scorep/SCOREP_User.h"

void foo()
{
  /* Declarations */
  SCOREP_USER_REGION_DEFINE( solve )

  /* Some code… */
  SCOREP_USER_REGION_BEGIN( solve, “<solver>",
                            SCOREP_USER_REGION_TYPE_LOOP )
  for (i = 0; i < 100; i++)
  {
    [...]
  }
  SCOREP_USER_REGION_END( solve )
  /* Some more code… */
}
```

Score-P user instrumentation API (C++)

```
#include "scorep/SCOREP_User.h"

void foo()
{
  // Declarations

  // Some code…
  {
    SCOREP_USER_REGION( "<solver>",
                        SCOREP_USER_REGION_TYPE_LOOP )
    for (i = 0; i < 100; i++)
    {
      [...]
    }
  }
  // Some more code…
}
```

Score-P measurement control API

- Can be used to temporarily disable measurement for certain intervals
  - Annotation macros ignored by default
  - Enabled with `--user` flag

```
#include “scorep/SCOREP_User.inc”

subroutine foo(…)
  ! Some code…
  SCOREP_RECORDING_OFF()
  ! Loop will not be measured
  do i=1,100
    [...]
  end do
  SCOREP_RECORDING_ON()
  ! Some more code…
end subroutine
```

Fortran (requires C preprocessor)

```
#include “scorep/SCOREP_User.h”

void foo(…) {
  /* Some code… */
  SCOREP_RECORDING_OFF()
  /* Loop will not be measured */
  for (i = 0; i < 100; i++) {
    [...]
  }
  SCOREP_RECORDING_ON()
  /* Some more code… */
}
```

C / C++

Score-P:
Conclusion and Outlook

Project management

- Ensure a single official release version at all times which will always work with the tools
- Allow experimental versions for new features or research
- Commitment to joint long-term cooperation
  - Development based on meritocratic governance model
  - Open for contributions and new partners

Future features

- Scalability to maximum available CPU core count

- Support for emerging architectures and new programming models

- Features currently worked on:
  - Hardware and MPI topologies
  - MPI-3 RMA support
  - OpenMP tool support (OMPT)
  - I/O recording
  - Basic support of measurements without re-compiling/-linking
  - Java recording
  - Persistent memory recording (e.g., PMEM, NVRAM, …)

Further information

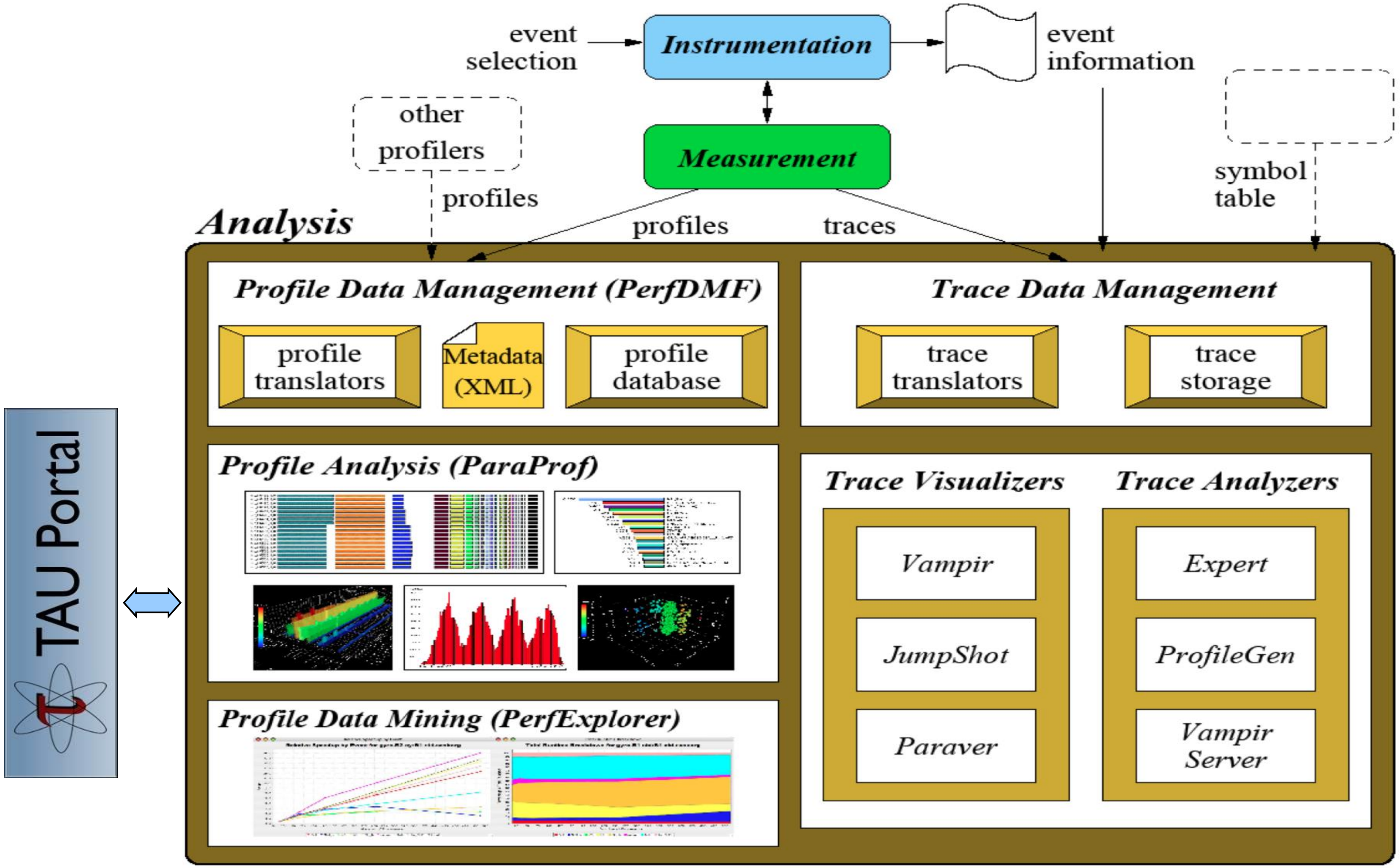
- Community instrumentation & measurement infrastructure
  - Instrumentation (various methods) and sampling
  - Basic and advanced profile generation
  - Event trace recording
  - Online access to profiling data
- Available under 3-clause BSD open-source license
- Documentation & Sources:
  - http://www.score-p.org
- User guide also part of installation:
  - `<prefix>/share/doc/scorep/{pdf,html}/`
- Support and feedback: support@score-p.org
- Subscribe to news@score-p.org, to be up to date

Performance data management with TAU PerfExplorer

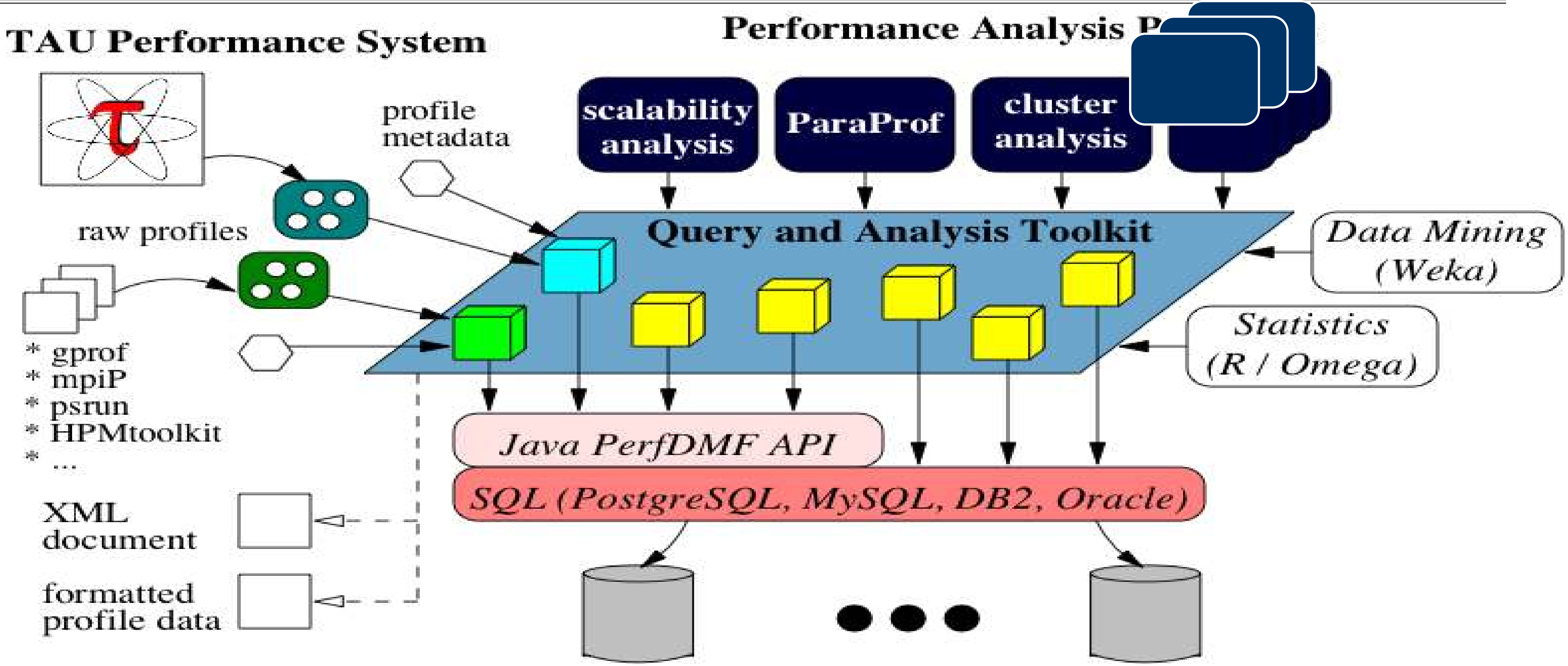
Tuning and Analysis Utilities

Sameer Shende
sameer@cs.uoregon.edu
University of Oregon
http://tau.uoregon.edu

TAU Analysis



TAUdb: Performance Data Management Framework



Using TAUdb

- Configure TAUdb (Done by each user)

  % taudb_configure --create-default

  - Choose derby, PostgreSQL, MySQL, Oracle or DB2
  - Hostname
  - Username
  - Password
  - Say yes to downloading required drivers (we are not allowed to distribute these)
  - Stores parameters in your ~/.ParaProf/taudb.cfg file
- Configure PerfExplorer (Done by each user)

  % perfexplorer_configure
- Execute PerfExplorer

  % perfexplorer

Using PerfExplorer

```
% wget http://tau.uoregon.edu/data.tgz  (Contains CUBE profiles from Score-P)
% tar zxf data.tgz; cd data; cat README; cd tau; ./upload.sh; perfexplorer
Or manually:
% taudb_configure --create-default
(Chooses derby, blank user/passwd, yes to save passwd, defaults)
% perfexplorer_configure
(Yes to load schema, defaults)
% paraprof
(load each trial: DB -> Add Trial -> Type (Paraprof Packed Profile) -> OK) OR use taudb_loadtrial –
  a "app" –x "experiment" –n "name" file.ppk
Then,
% tar zxf $TAU/data.tgz; cd data/tau;
% taudb_loadtrial –a BT_MZ –x "Class_B" bt-mz_B.*.ppk
% perfexplorer
(Select experiment, Menu: Charts -> Speedup)
```

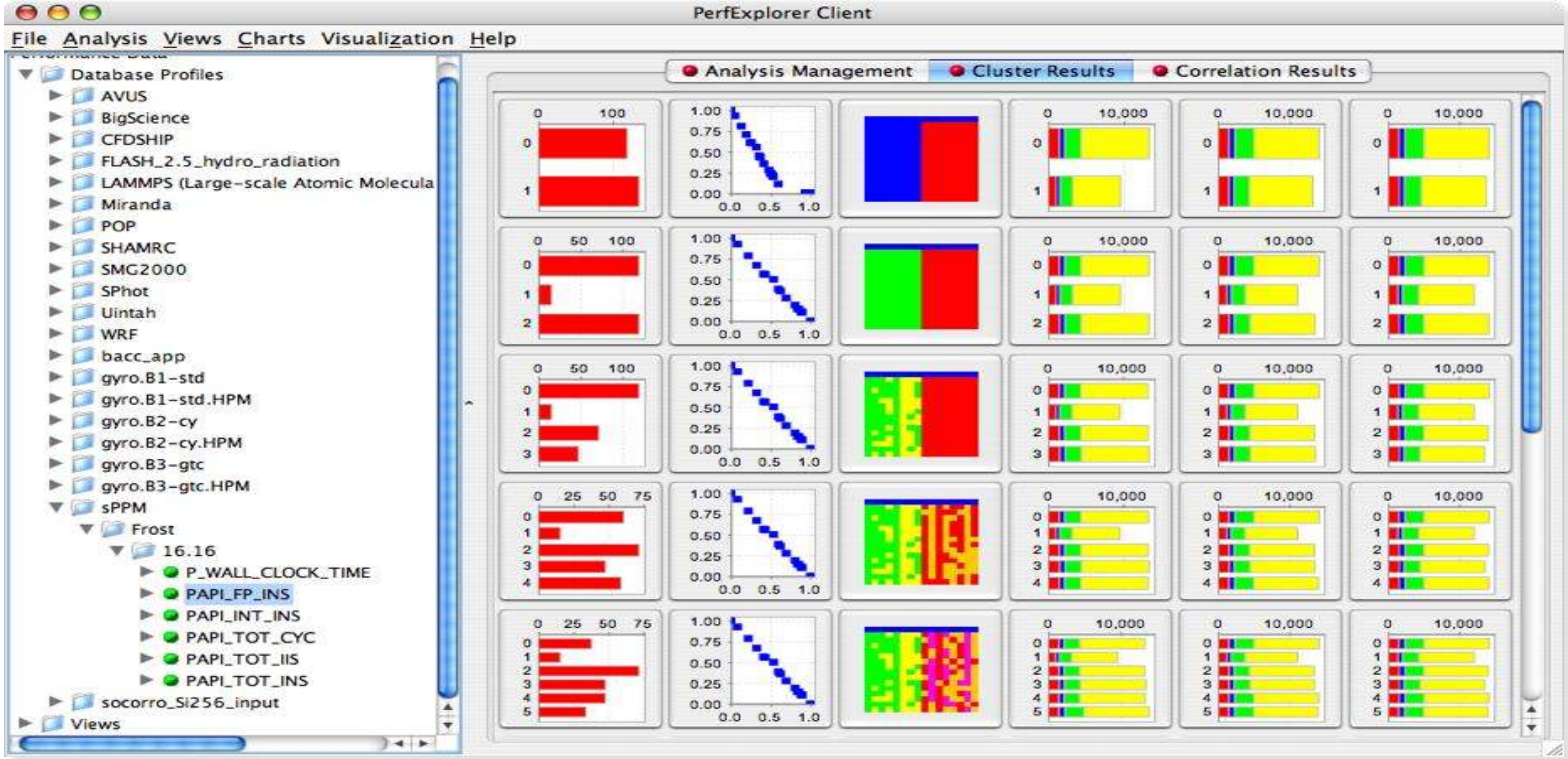
Performance Data Mining (PerfExplorer)

- Performance knowledge discovery framework
  - Data mining analysis applied to parallel performance data
    - comparative, clustering, correlation, dimension reduction, …
  - Use the existing TAU infrastructure
    - TAU performance profiles, taudb
  - Client-server based system architecture
- Technology integration
  - Java API and toolkit for portability
  - taudb
  - R-project/Omegahat, Octave/Matlab statistical analysis
  - WEKA data mining package
  - JFreeChart for visualization, vector output (EPS, SVG)

PerfExplorer: Using Cluster Analysis

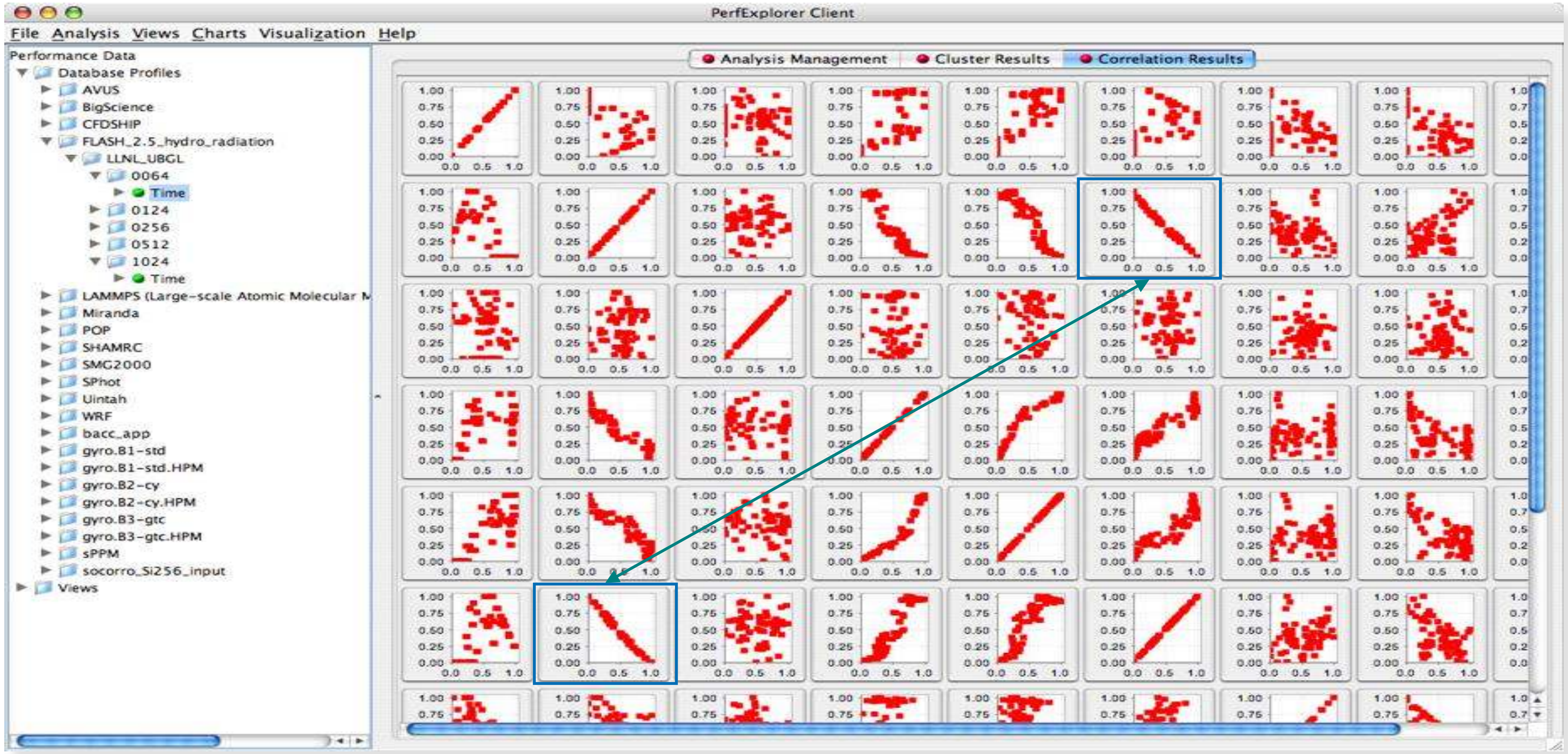
- Performance data represented as vectors - each dimension is the cumulative time for an event
- *k*-means: *k* random centers are selected and instances are grouped with the "closest" (Euclidean) center
- New centers are calculated and the process repeated until stabilization or max iterations
- Dimension reduction necessary for meaningful results
- Virtual topology, summaries constructed

PerfExplorer - Cluster Analysis (sPPM)



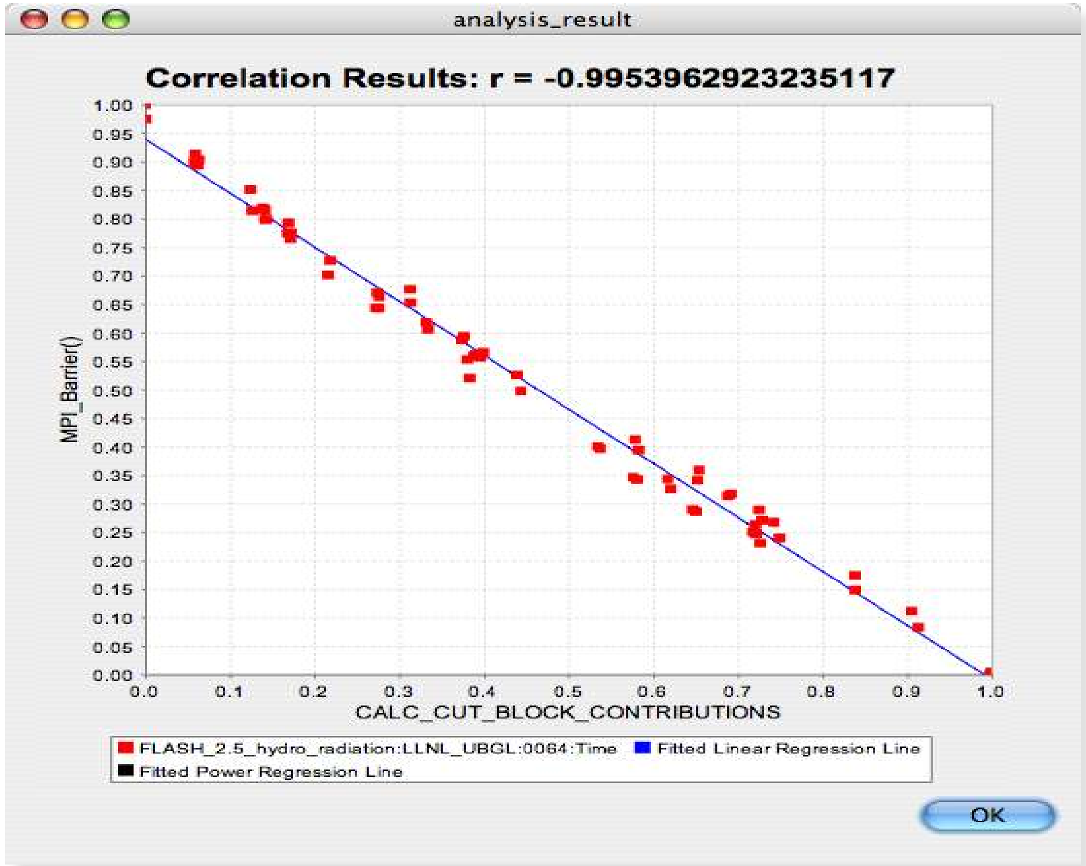
PerfExplorer - Correlation Analysis (Flash)

- Describes strength and direction of a linear relationship between two variables (events) in the data



PerfExplorer - Correlation Analysis (Flash)

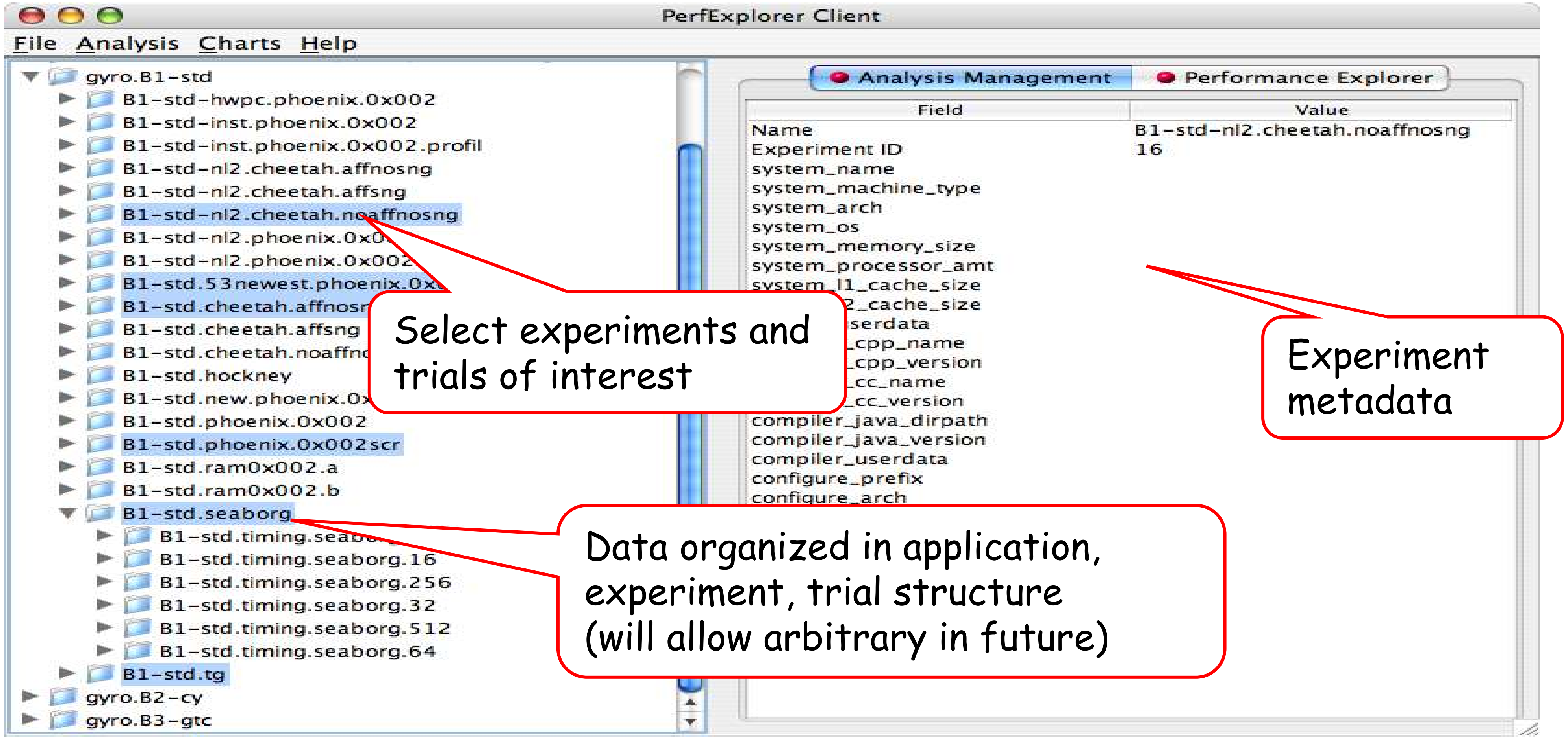
- -0.995 indicates strong, negative relationship
- As CALC_CUT_ BLOCK_CONTRIBUTIONS() increases in execution time, MPI_Barrier() decreases



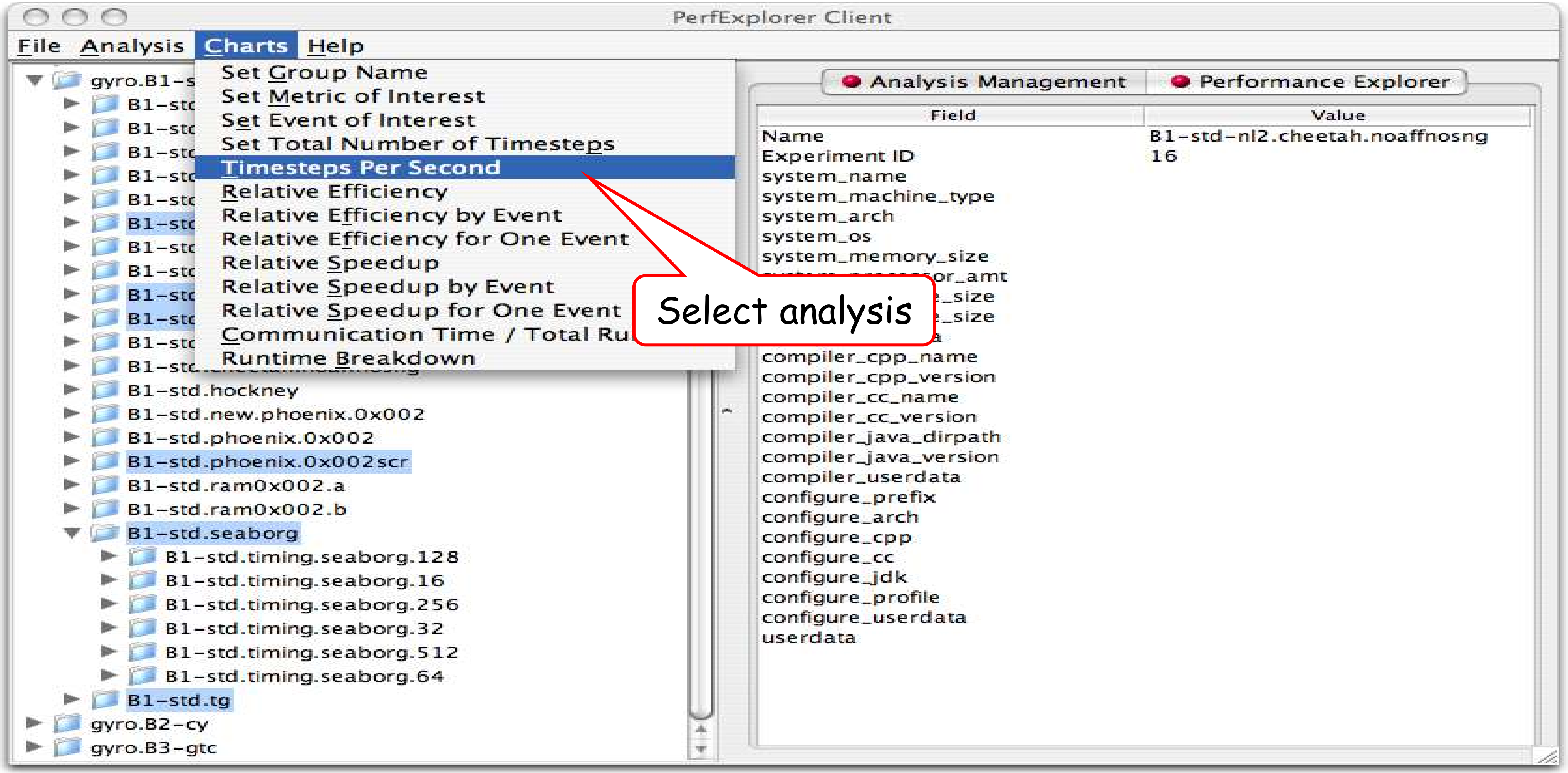
PerfExplorer - Comparative Analysis

- Relative speedup, efficiency
  - total runtime, by event, one event, by phase
- Breakdown of total runtime
- Group fraction of total runtime
- Correlating events to total runtime
- Timesteps per second

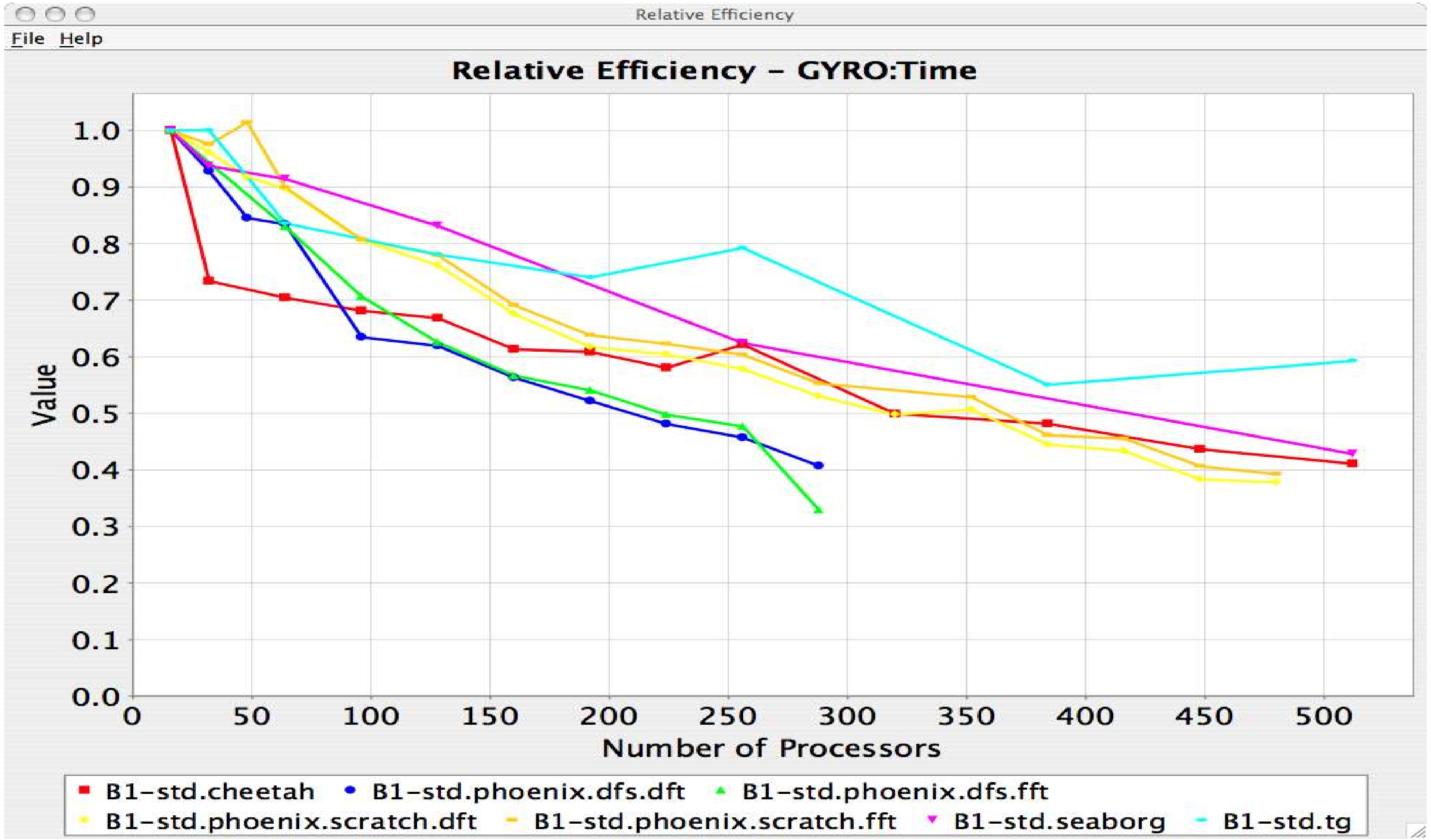
PerfExplorer - Interface



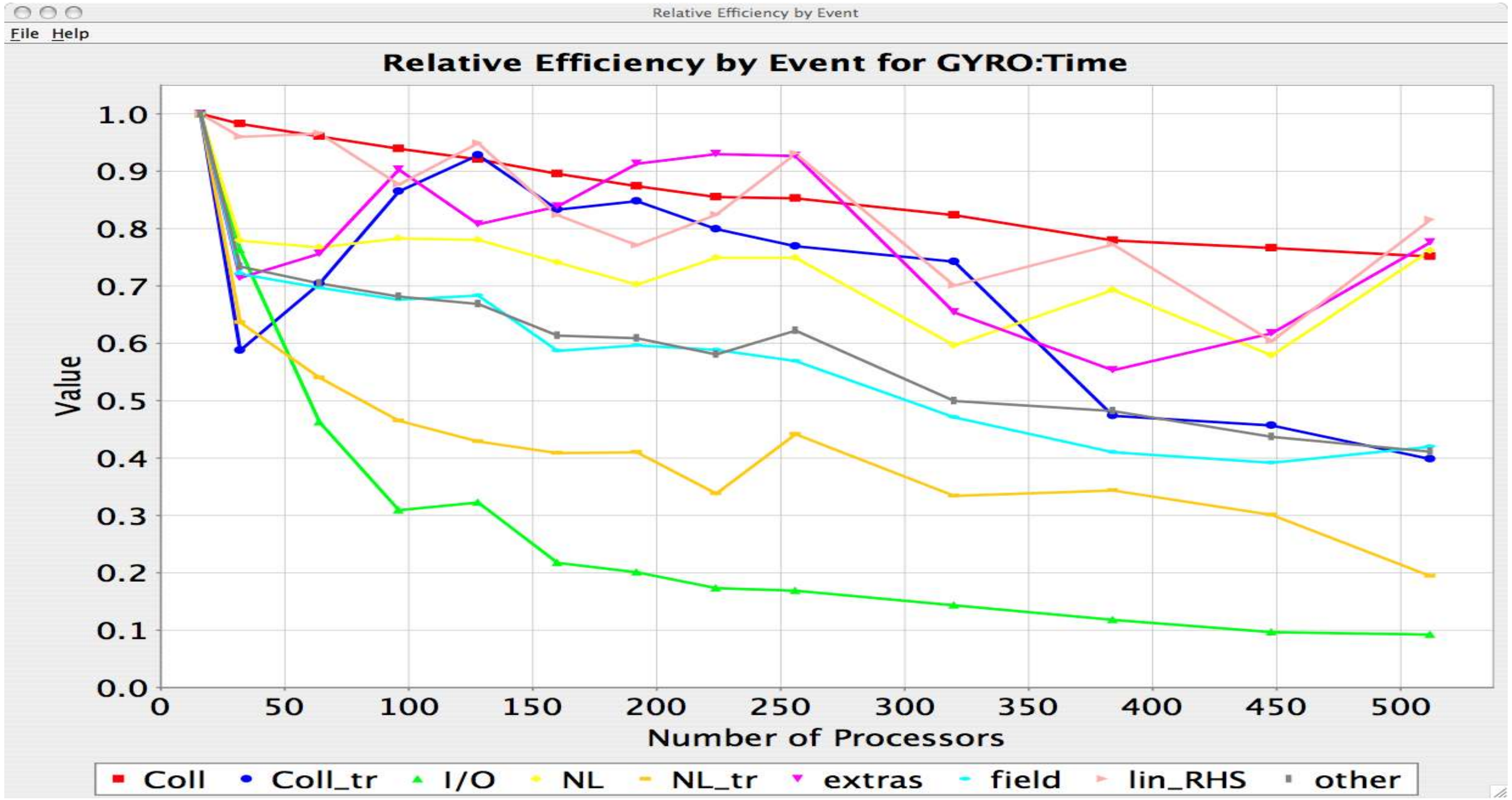
PerfExplorer - Interface



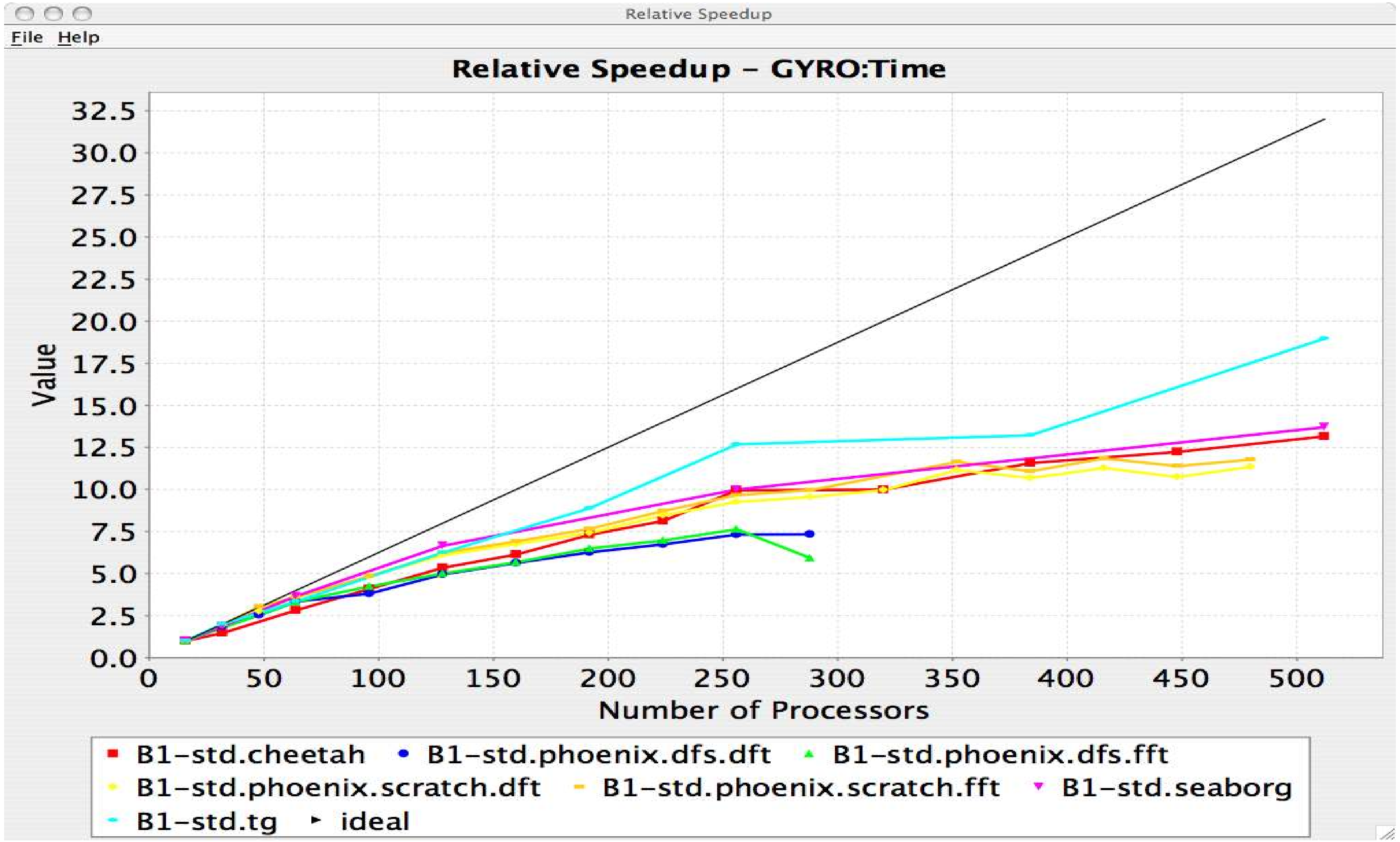
PerfExplorer - Relative Efficiency Plots



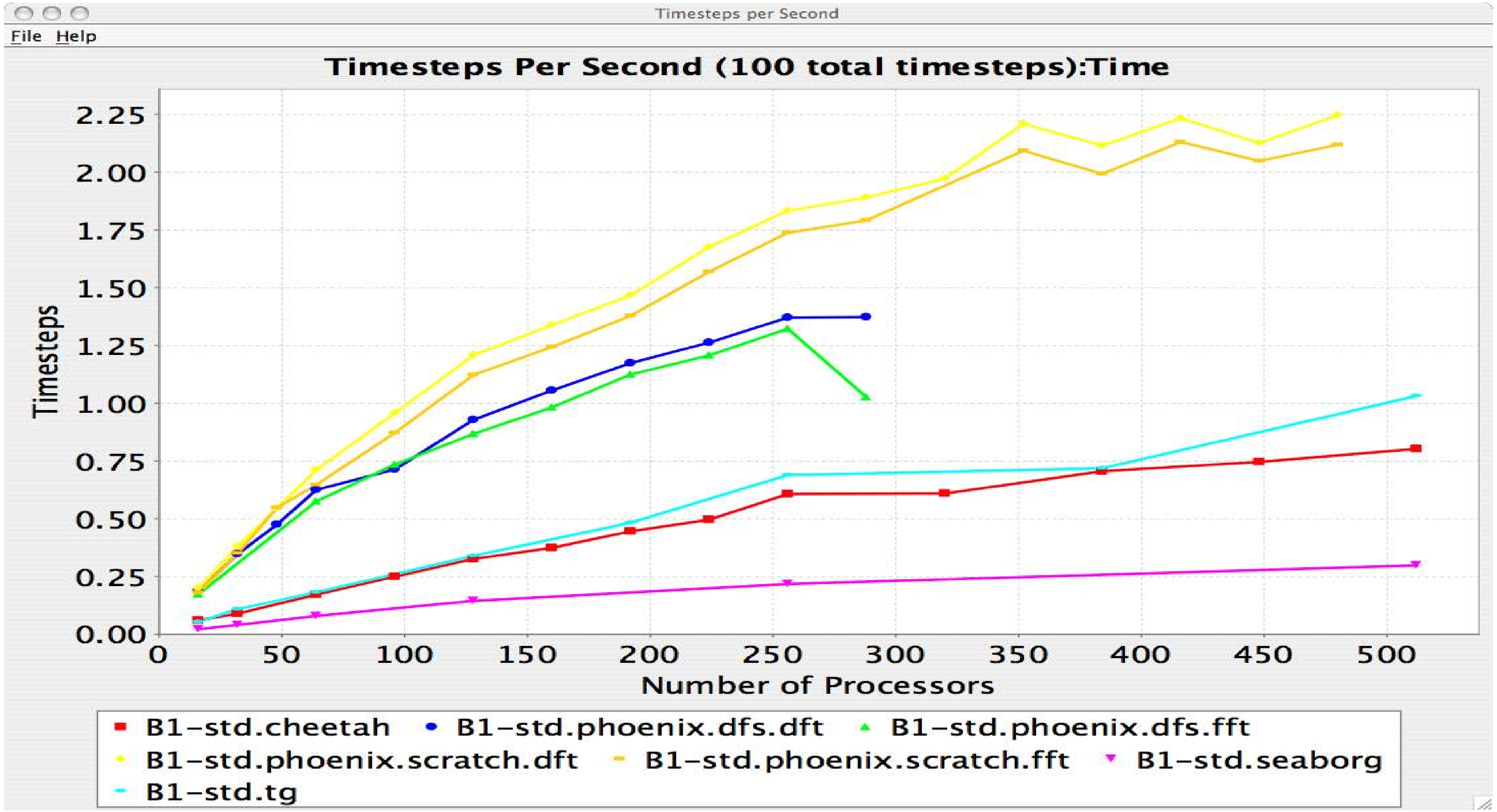
PerfExplorer - Relative Efficiency by Routine



PerfExplorer - Relative Speedup

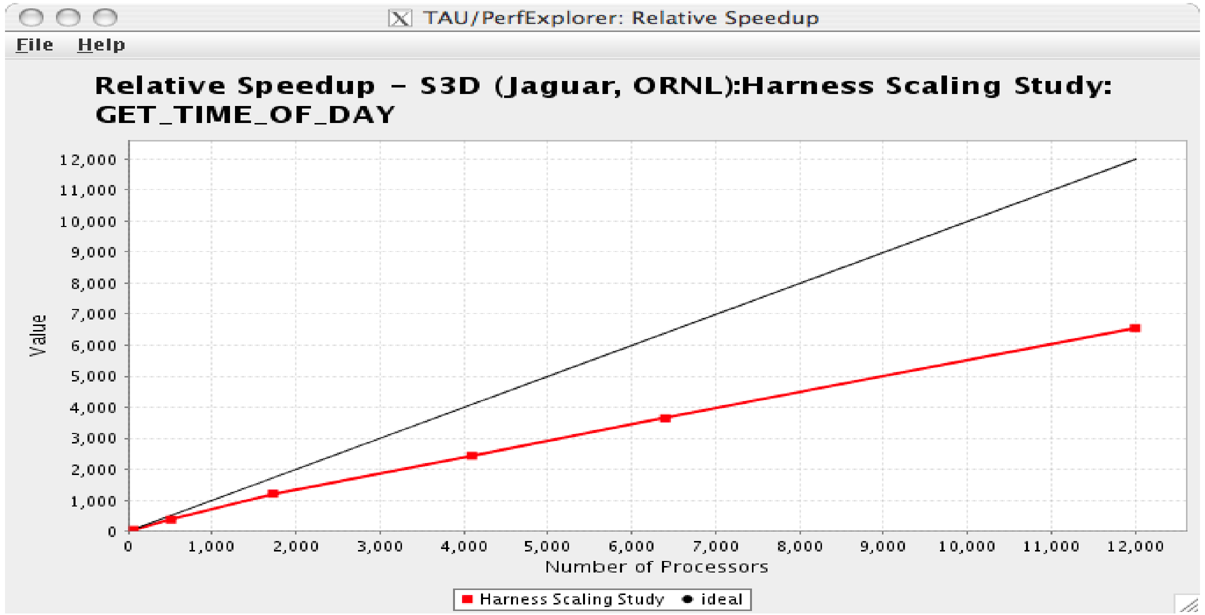


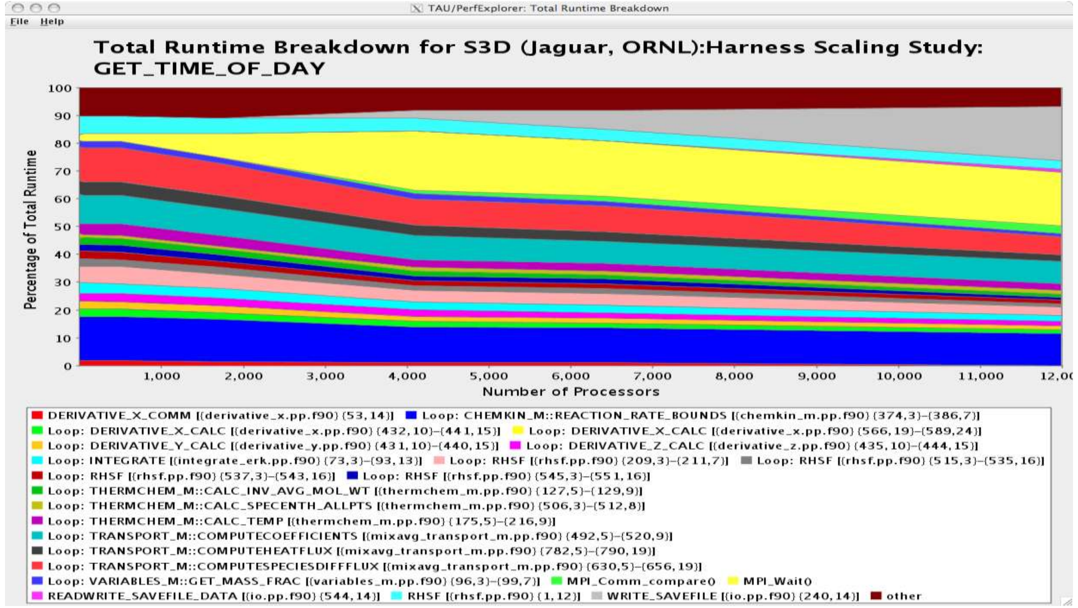
PerfExplorer - Timesteps Per Second



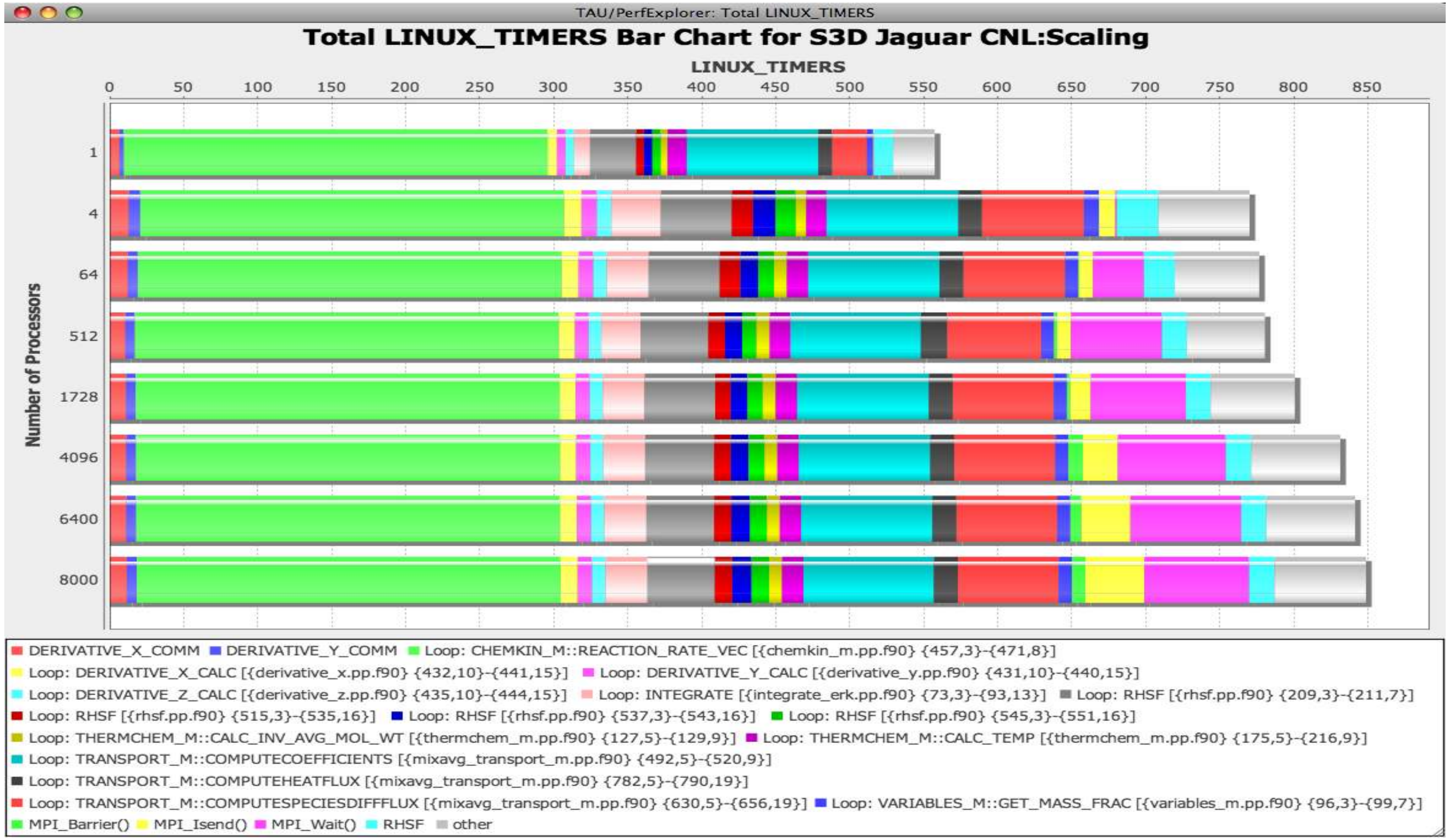
Evaluate Scalability

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

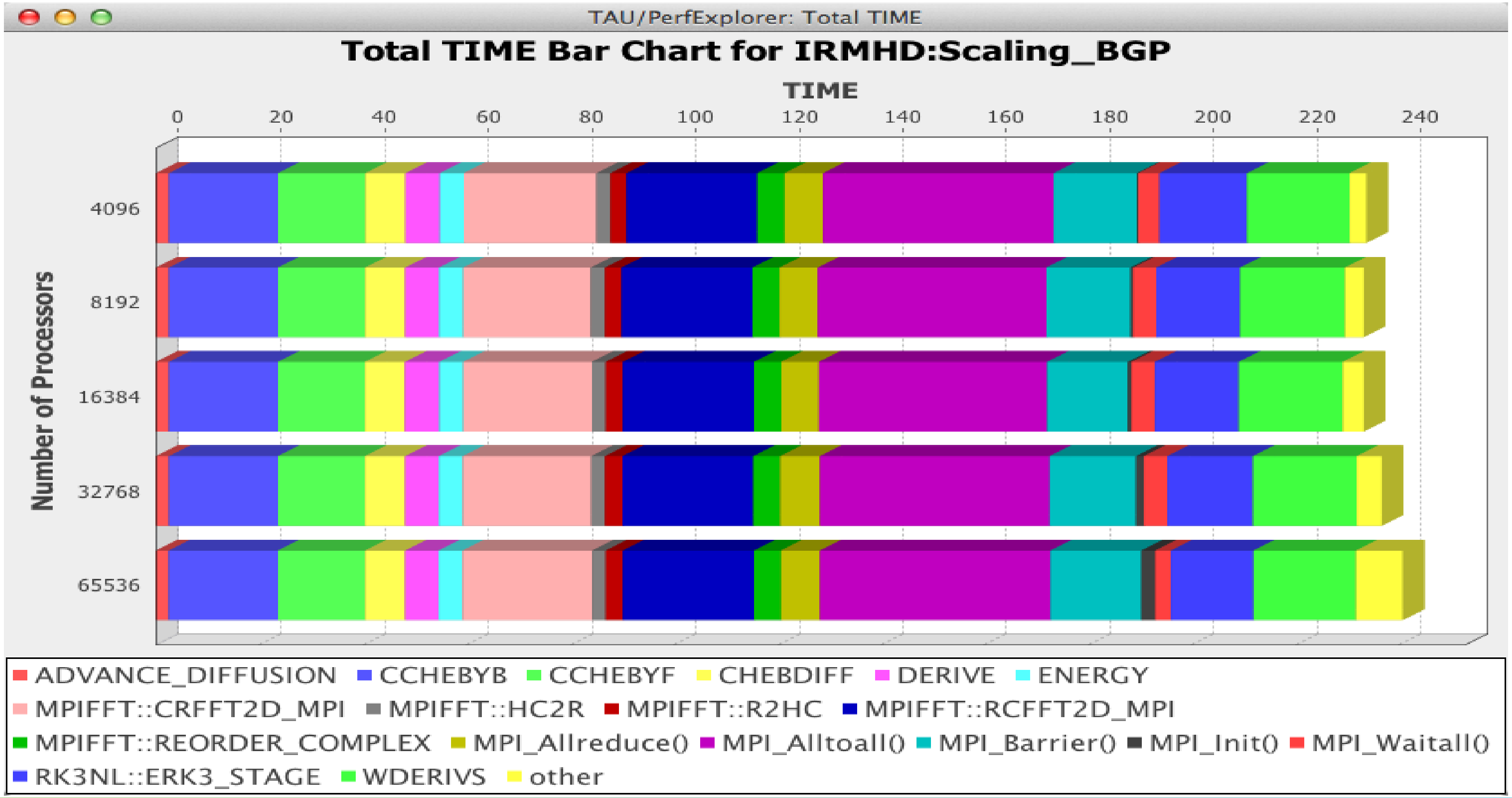




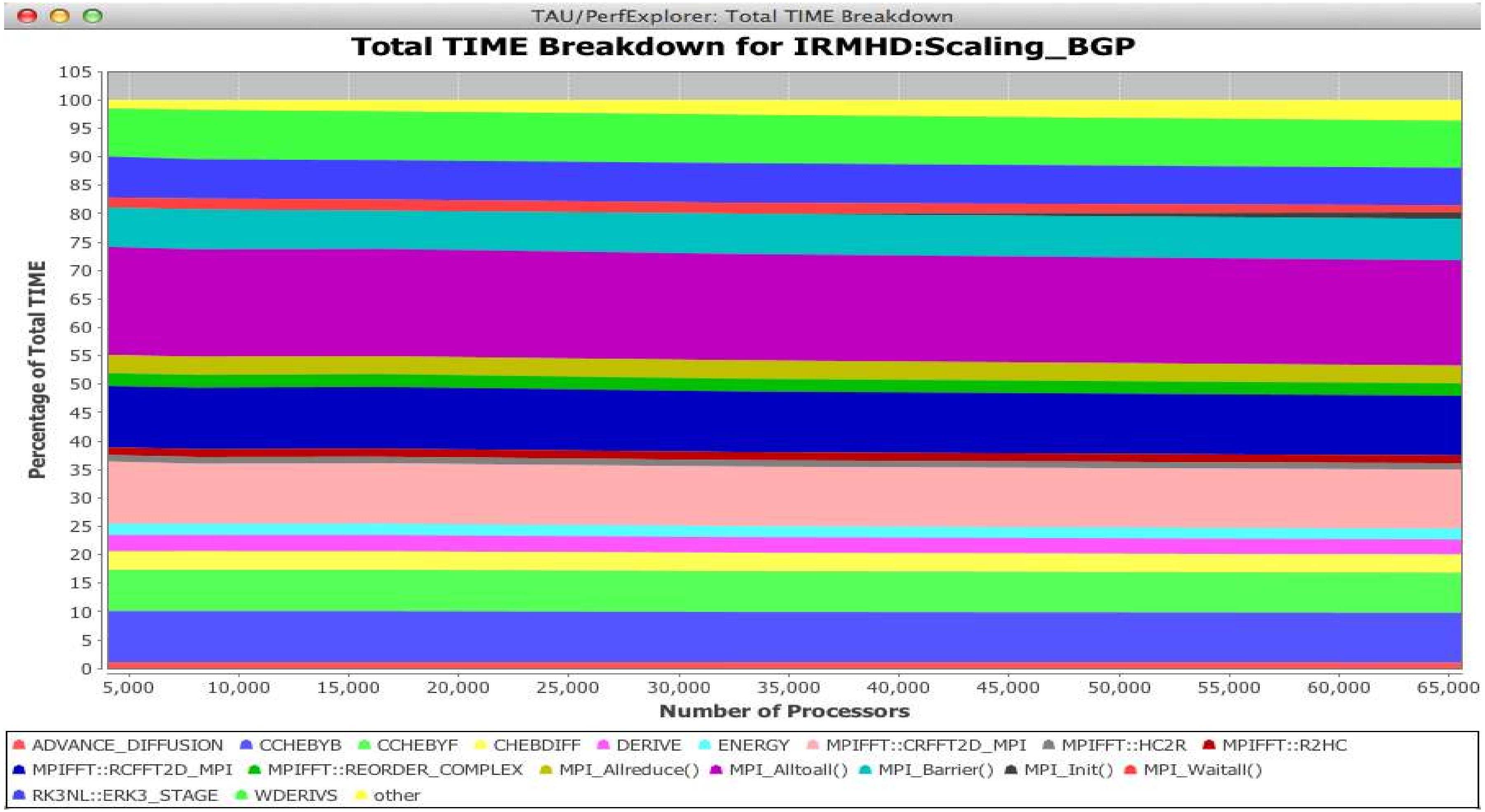
Evaluate Scalability



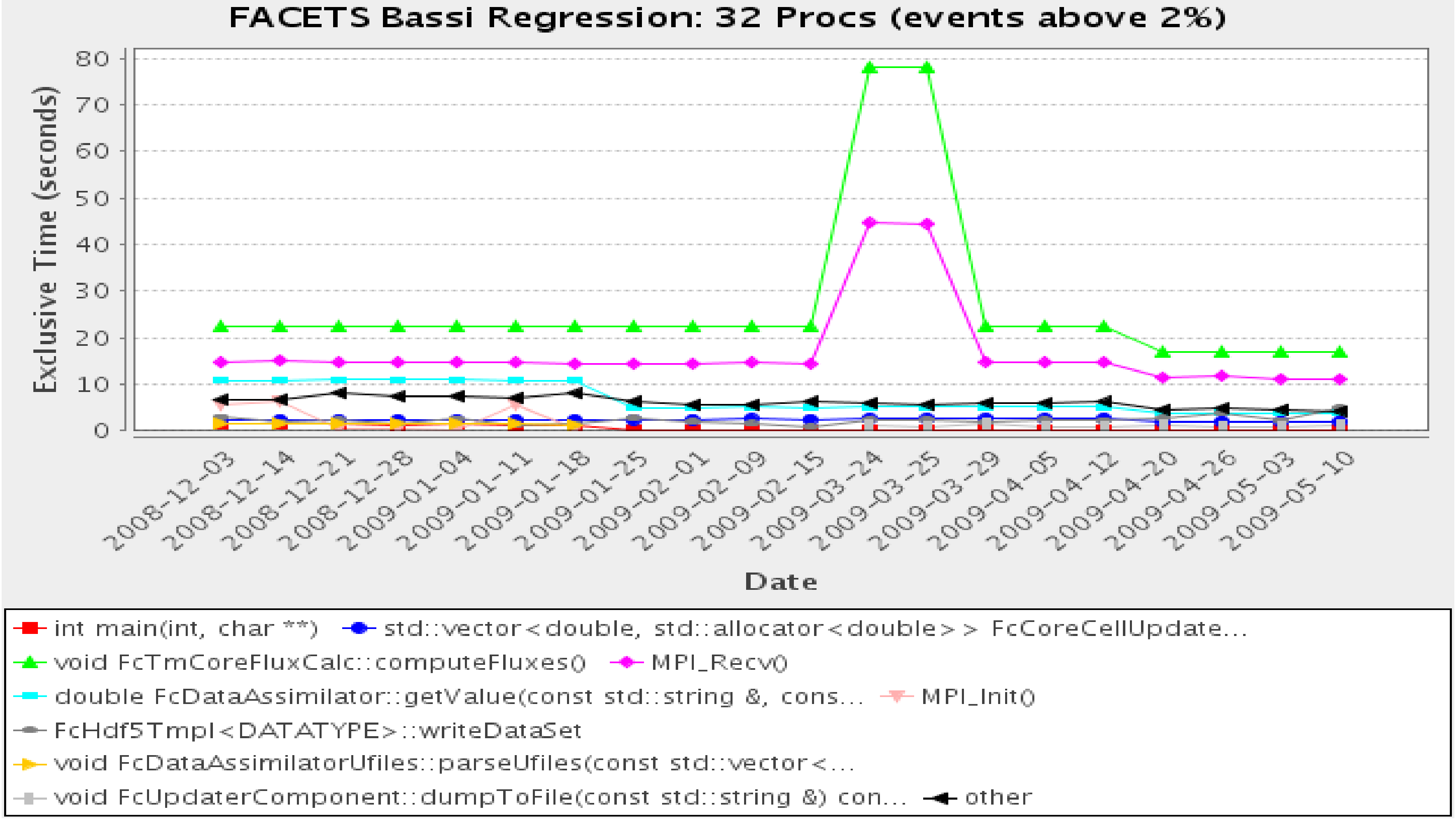
PerfExplorer



PerfExplorer



Performance Regression Testing



Download TAU from U. Oregon

http://tau.uoregon.edu

http://www.hpclinux.com [LiveDVD, OVA]

Free download, open source, BSD license

Parallel application performance analysis case studies

The VI-HPS Team

Outline

- Case I:
  - NPB3.3_MZ_MPI/**BT-MZ** (MPI+OpenMP) on *MARCONI-KNL*: load balancing
- Case II:
  - **k-Wave** (MPI+OpenMP) on *Salomon*: load-balancing in FFTW OpenMP parallel regions
- Case III:
  - **ICON** (MPI) on *Mistral*: automatic trace analysis of critical path of execution
- Case IV:
  - **PIConGPU** (MPI+CUDA): computation offload to multiple attached accelerator devices
- Case V:
  - **TensorFlow** (Python+CUDA): interpreted & compiled heterogeneous execution measurement

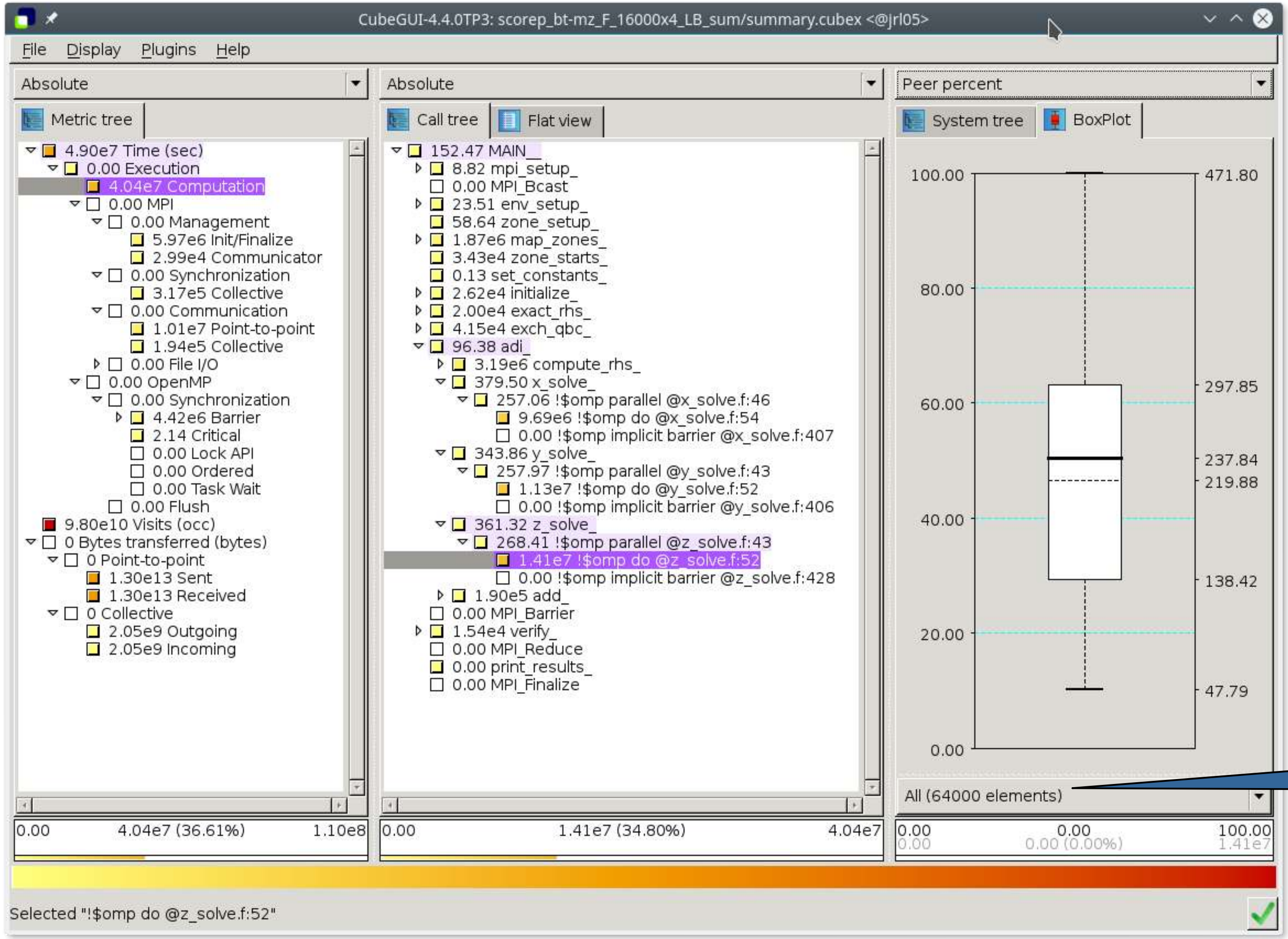
Case I: NPB3.3-MZ-MPI/BT-MZ: balancing OMP threads per process

- Same F77 benchmark code as used in tutorial exercise, CLASS=F (128x128 zones)
- Using Intel compilers and Intel MPI on *MARCONI-KNL* (68C), **`-xMIC-AVX512`**
- 4,000 MPI processes (4 ranks/KNL), OMP_NUM_THREADS=64
  - Default execution configuration "balances" number of OpenMP threads per MPI process
  - Threads reassigned from processes with simpler zones to those with more complex zones
- Intel compiler configuration file used when instrumenting with Score-P
  - Avoids instrumenting small/frequently-executed routines

- Since "balancing" scheme doesn't take account of threads (cores) per compute node, some KNL processors end up more over-subscribed
  - whereas 39 KNL nodes received 4 MPI processes each with 67 OpenMP threads (268 threads), two KNL nodes had 4 MPI processes each with only 62 OpenMP threads (248 threads)

⇒12% better performance exploiting hyper-threading and thread "balancing"

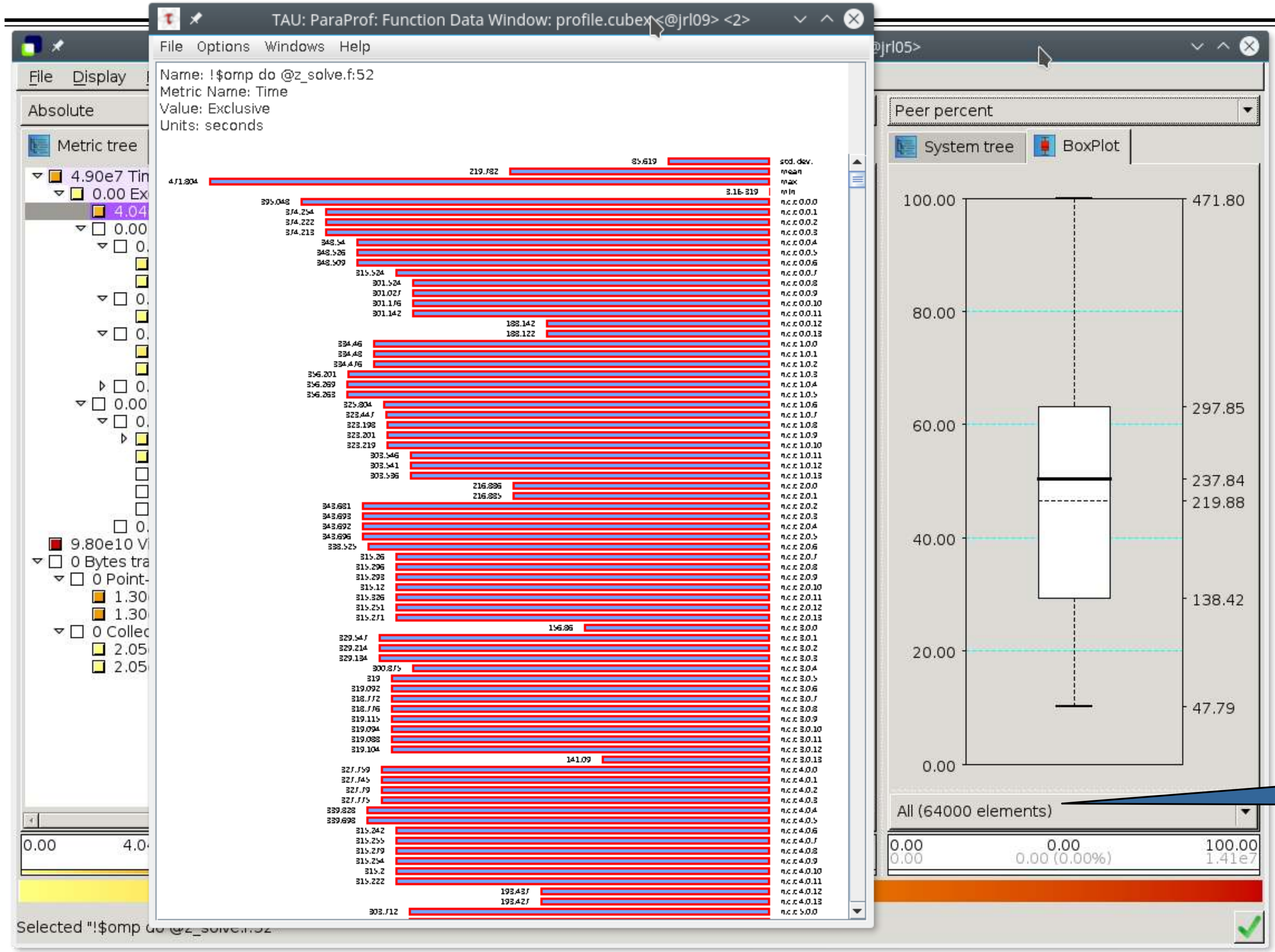
BT-MZ.F Score-P summary profile: 16p16000x4 “balancing” active



- BT-MZ class F (12032x8960x250)
- Wall clock time 1221 seconds (18.3 Mop/s total)
  - 15.0% MPI, 4.0% OMP
- Large computational imbalance for threads in each ADI solver direction, despite attempt at “balancing”

64,000 OpenMP threads in total, running on 1000 KNL nodes

BT-MZ.F Score-P summary profile: alternative presentations

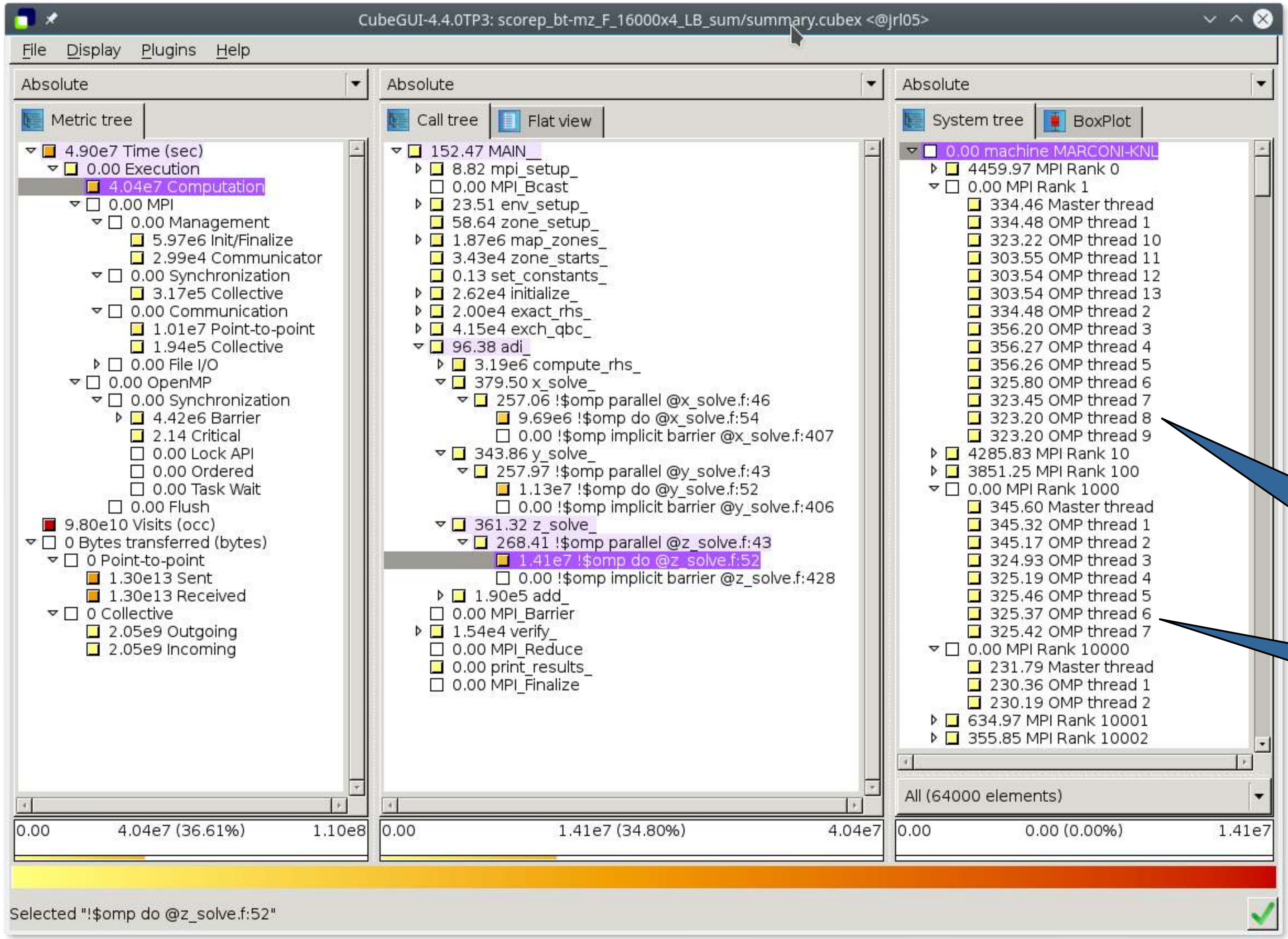


!$omp do @z_solve.f52 loop computation time by OpenMP thread

- CUBE system tree scrollable list limited to showing values of around 30 processes/threads at a time
- CUBE boxplot summarizes metric value distribution
  - min 47.8 to max 471.8 seconds
- ParaProf charts the same metric values graphically

64,000 OpenMP threads in total, running on 1000 KNL nodes

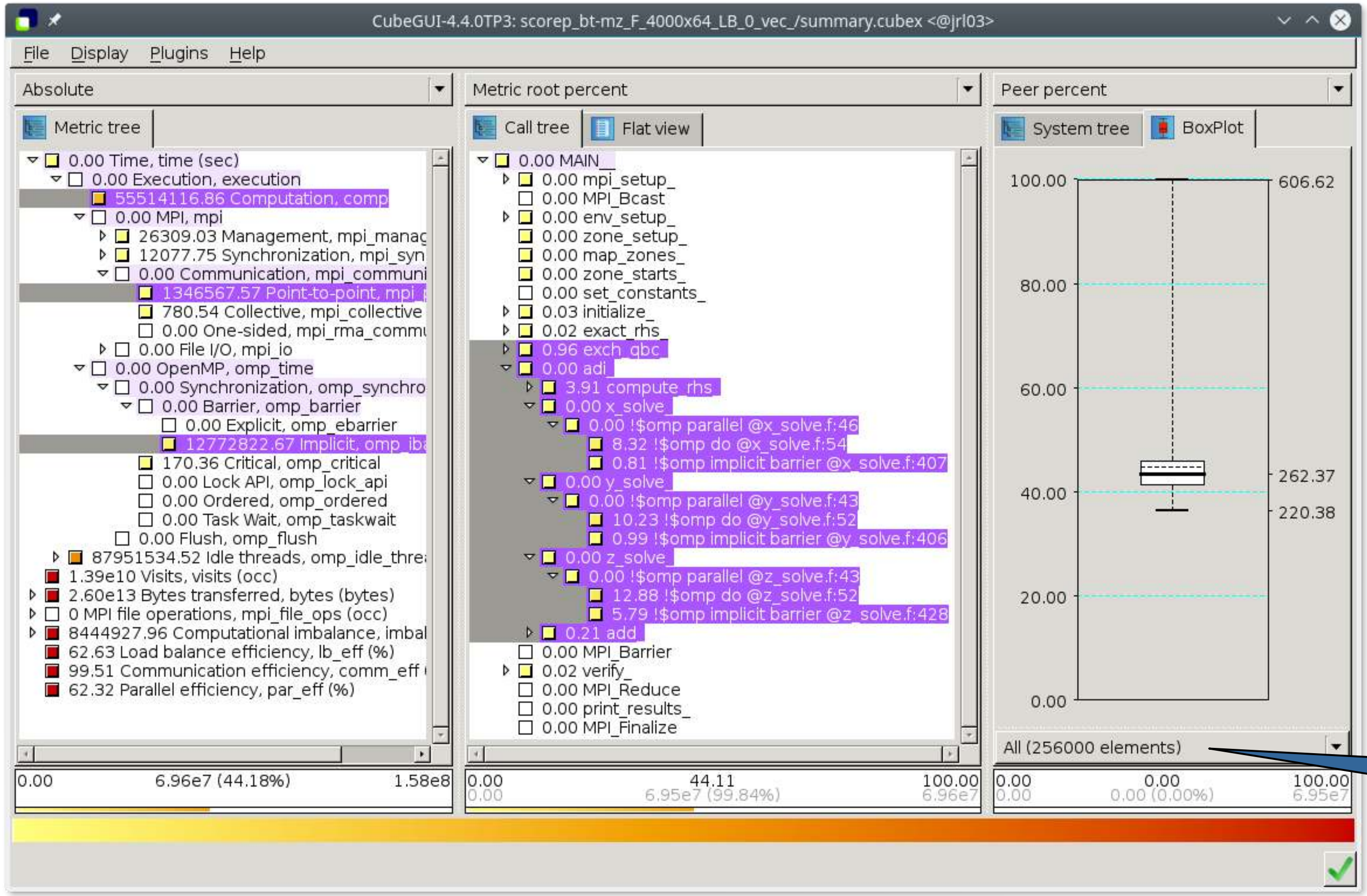
BT-MZ.F Score-P summary profile: 16p16000x4 “balancing” active



- BT-MZ class F (12032x8960x250)
- Wall clock time 1221 seconds (18.3 Mop/s total)
- 15% MPI, 4% OMP
- Computational imbalance for threads, despite attempt at “balancing”

Rank 1 has been assigned 14 threads, whereas rank 10000 only has 3 threads

BT-MZ.F Score-P summary profile: 4p4000x64 no “balancing”

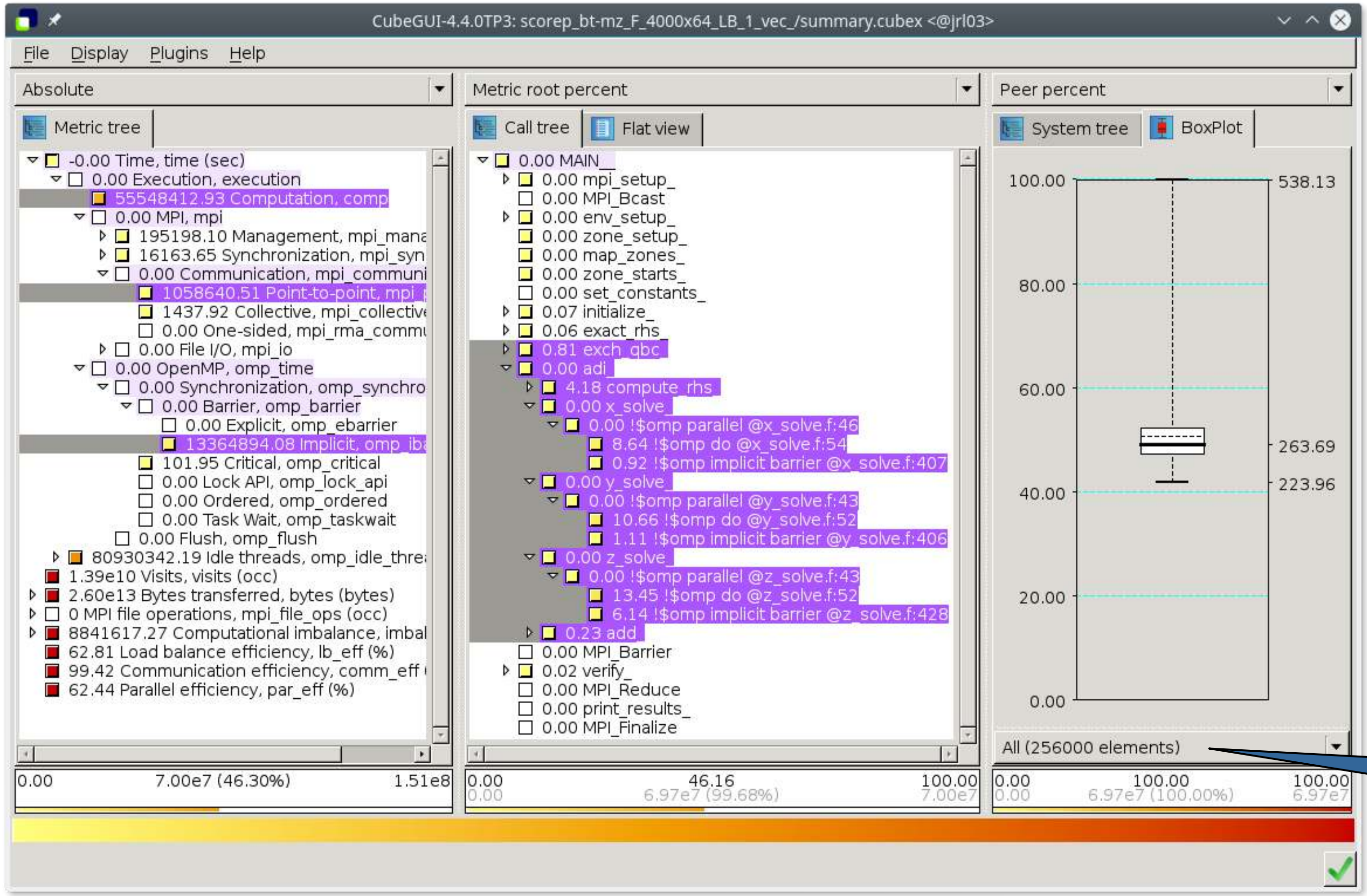


- BT-MZ class F (12032x8960x250):
  4 MPI ranks/node,
  64 OpenMP threads/rank
  =256 threads/node
- No load “balancing”
- 4000 KNL nodes using HW threading

- Wall clock time 603 seconds (37.0 Mop/s total)
  - time ranging from 220 to 606s

256,000 OpenMP threads in total, running on 4,000 KNL nodes

BT-MZ.F Score-P summary profile: 4p4000x64 “balancing” active

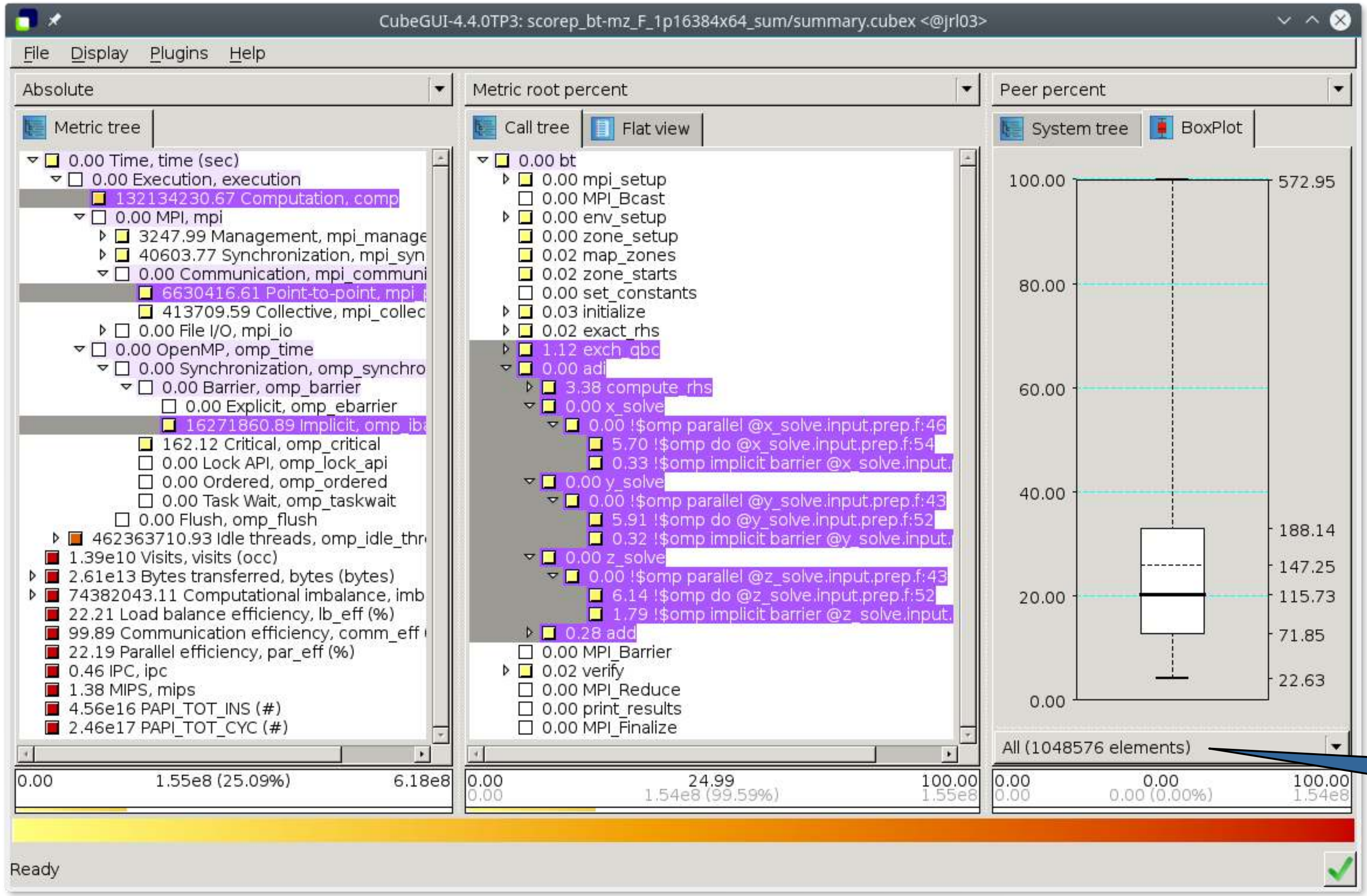


- BT-MZ class F (12032x8960x250): 4 MPI ranks/node, **62-67** OpenMP threads/rank ~256 threads/node
- Static “balancing”
- 4000 KNL nodes using HW threading

- Wall clock time 531 seconds (42.0 Mop/s total)
  - time ranging from 224 to 538s
- **12% gain** from static balancing of OpenMP threads per MPI process

256,000 OpenMP threads in total, running on 4,000 KNL nodes

BT-MZ.F Score-P summary profile: 1p16364x64 on *JUQUEEN* BG/Q

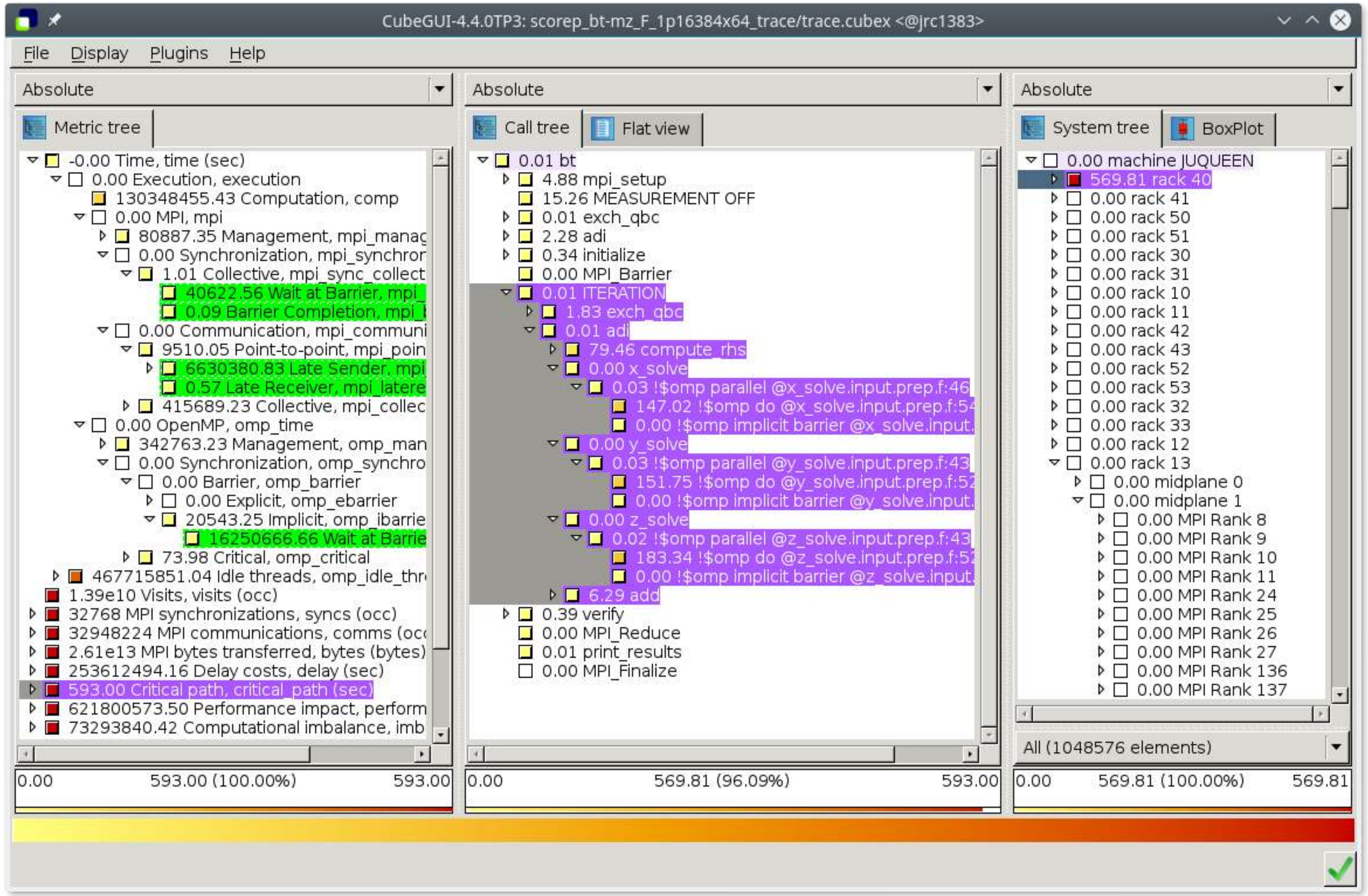


- BT-MZ class F (12032x8960x250): 1 MPI rank/node, 64 OpenMP threads/rank
- No load “balancing”
- 16,384 PowerPC A2 compute nodes (16 racks) of IBM Blue Gene/Q

- IBM XL compiler instrumentation with measurement filter
- Wall clock time 573 seconds (39.1 Mop/s total)
- 7% measurement dilation (including 2 hardware counters)

1M OpenMP threads in total, running on 16,384 nodes

BT-MZ.F Scalasca trace analysis: 1p16364x64 on *JUQUEEN* BG/Q



- BT-MZ class F (12032x8960x250): 1 MPI rank/node, 64 OpenMP threads/rank
- No load "balancing"
- 16,384 PowerPC A2 compute nodes (16 racks) of IBM Blue Gene/Q

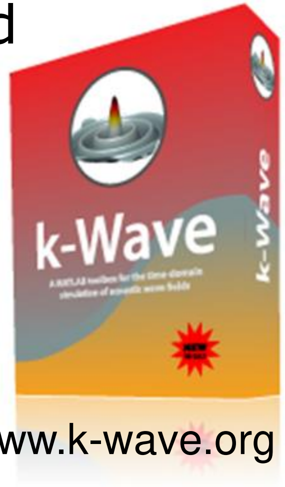
- 0.5 TiB event data written in 3.3s (using one SIONlib file per IONode)
- Scalasca automatic trace analysis
  - distinguishes waiting times for comm&synch operations as 3.7% of total CPU time, resulting in 75% idle threads
  - quantifies callpath contributions to critical path of execution (all on MPI rank 0)

Outline

- Case I:
  - NPB3.3_MZ_MPI/**BT-MZ** (MPI+OpenMP) on *MARCONI-KNL*: load balancing
- Case II:
  - **k-Wave** (MPI+OpenMP) on *Salomon*: load-balancing in FFTW OpenMP parallel regions
- Case III:
  - **ICON** (MPI) on *Mistral*: automatic trace analysis of critical path of execution
- Case IV:
  - **PIConGPU** (MPI+CUDA): computation offload to multiple attached accelerator devices
- Case V:
  - **TensorFlow** (Python+CUDA): interpreted & compiled heterogeneous execution measurement

Case II: k-Wave: load balancing in FFTW OpenMP parallel regions

- Toolbox for time-domain acoustic and ultrasound simulations in complex and tissue-realistic media, developed by Brno University of Technology (CZ)
- C++ code parallelized with MPI and OpenMP [+ CUDA unused]
  - FFTW library using OpenMP parallelization; parallel HDF5 file I/O
  - GCC compiler and OpenMPI library
- Executed on *Salomon* Intel Xeon compute nodes (IT4Innovations/CZ)
  - 64 MPI processes (2 per compute node), 12 OpenMP threads per process
  - Score-P runtime measurement filter used to eliminate FFTW computation routines

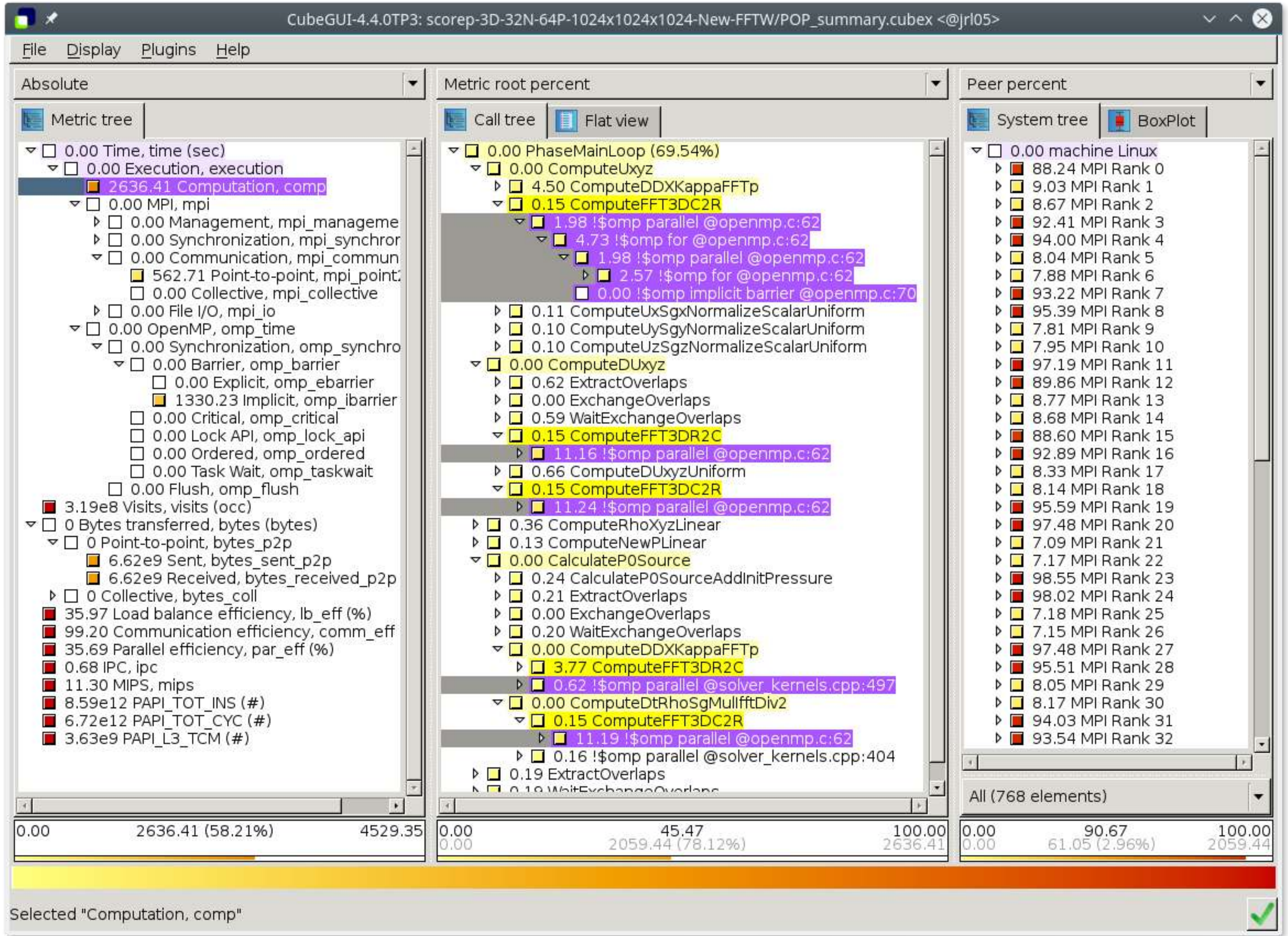


www.k-wave.org

- 3D domain decomposition (1024^3 on 4x4x4 processes) suffered major load imbalance
  - exterior MPI processes with fewer grid cells took 4x longer than interior
  - OpenMP-parallelized FFTs were much less efficient for (smaller) grid dimensions of exterior, requiring many more small and poorly-balanced nested parallel loops
- Revised to use a periodic domain with identical (padded) halo zones for each MPI rank

  ⇒ ***improved kernel by a factor of 6 and overall execution time by a factor of 2***

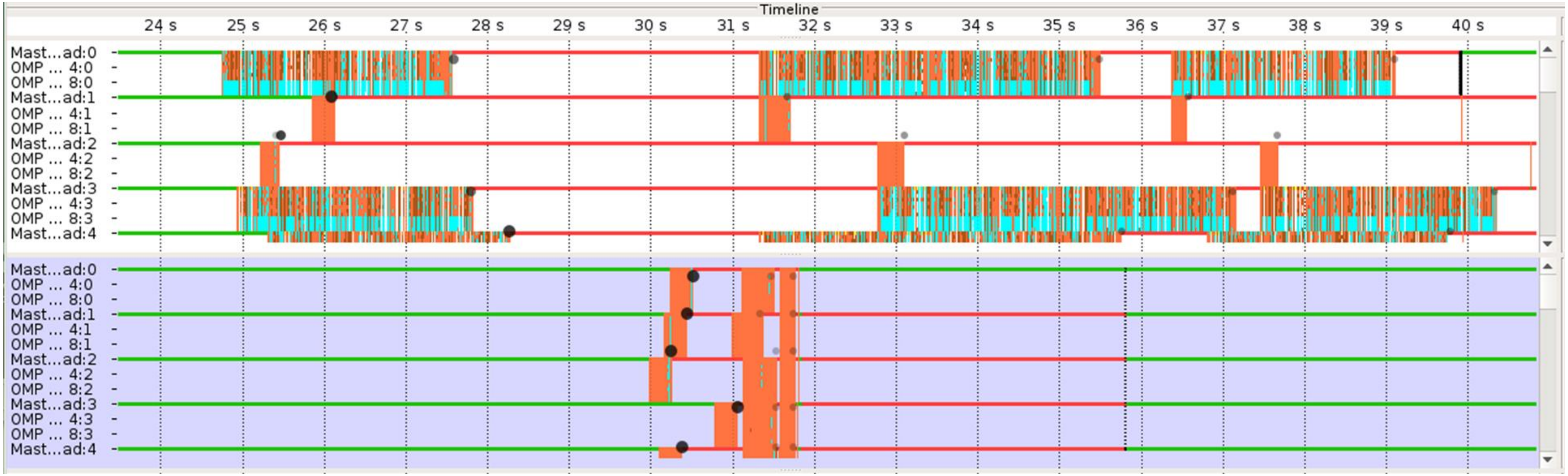
k-Wave summary profile (initial version): parallel region imbalance



PhaseMainLoop routine extract

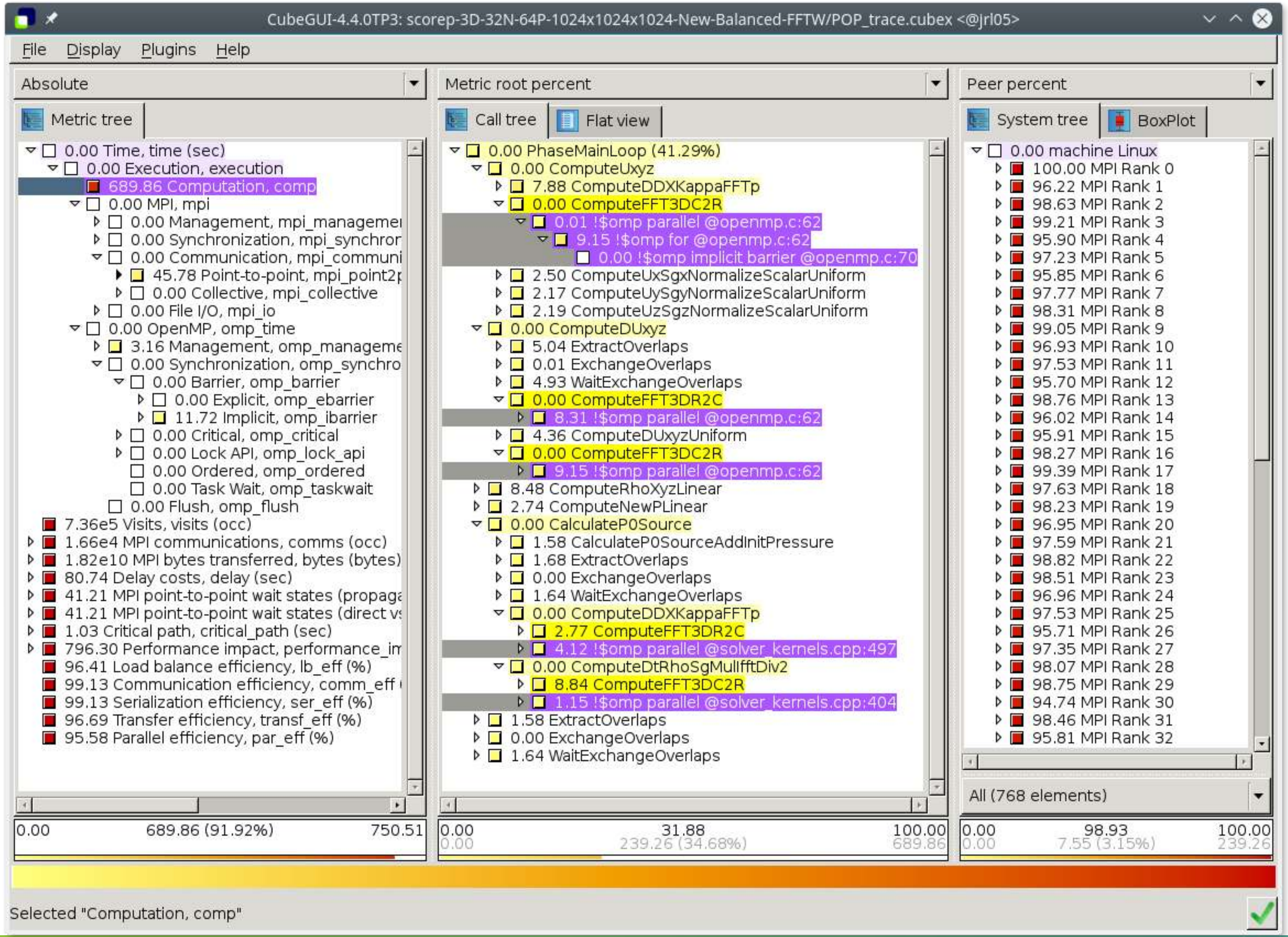
- 58% Computation time
  - 29% OMP + 12% MPI overheads
- 50% of Computation time in five ComputeFFT3D routines each with OpenMP parallel regions
  - with nested parallel regions inside
- Huge computation imbalance
  - half of the MPI ranks (1,2,5,6,9,…) ten times faster than the others
  - results in huge amounts of OpenMP implicit barrier synchronization time at end of parallel regions
- Only 35% parallel efficiency

k-Wave Vampir trace time-line comparison (original & revised)



- executions before [upper] and after [lower] balancing grid-points per MPI process
- showing processes for corner ranks (0&3) and edge ranks (1&2) of 4x4x4 geometry
- MPI synchronization in red, OpenMP synchronization in cyan

k-Wave summary profile (revised version): balanced parallel regions



- **`PhaseMainLoop`** routine
  Execution time reduced 6-fold from 4530 to 750 seconds
- Now 92% Computation time
  - 2% OMP + 6% MPI overhead
- 32% of Computation time now in ComputeFFT3D routines
  - simplified OpenMP parallel regions no longer nested
- Greatly improved load balance
  - 1.4% standard deviation
- Over 95% parallel efficiency

Outline

- Case I:
  - NPB3.3_MZ_MPI/**BT-MZ** (MPI+OpenMP) on *MARCONI-KNL*: load balancing
- Case II:
  - **k-Wave** (MPI+OpenMP) on *Salomon*: load-balancing in FFTW OpenMP parallel regions
- Case III:
  - **ICON** (MPI) on *Mistral*: automatic trace analysis of critical path of execution
- Case IV:
  - **PIConGPU** (MPI+CUDA): computation offload to multiple attached accelerator devices
- Case V:
  - **TensorFlow** (Python+CUDA): interpreted & compiled heterogeneous execution measurement

Case III: ICON

- Icosahedral non-hydrostatic unified weather forecasting and climate model jointly developed by
  - Max Planck Institute for Meteorology (MPI-M)
  - Germany's National Meteorological Service (DWD)
- ICON source code and test case provided by H. Bockelmann (DKRZ)
  - Mostly Fortran 90, some C; parallelized with MPI [+ OpenMP unused]
  - Intel compiler and bullx MPI library
  - 24-hour physical simulation in 10 min increments
- Executed on *Mistral* Intel Xeon compute nodes (DKRZ):
  - 2x 12-core Intel Xeon E5-2680 v3 (Haswell) @ 2.5GHz
  - 24 MPI processes/node, experiments with 4/8/16/32 compute nodes

⇒ ***Identification & quantification of impact of periodic additional computations***

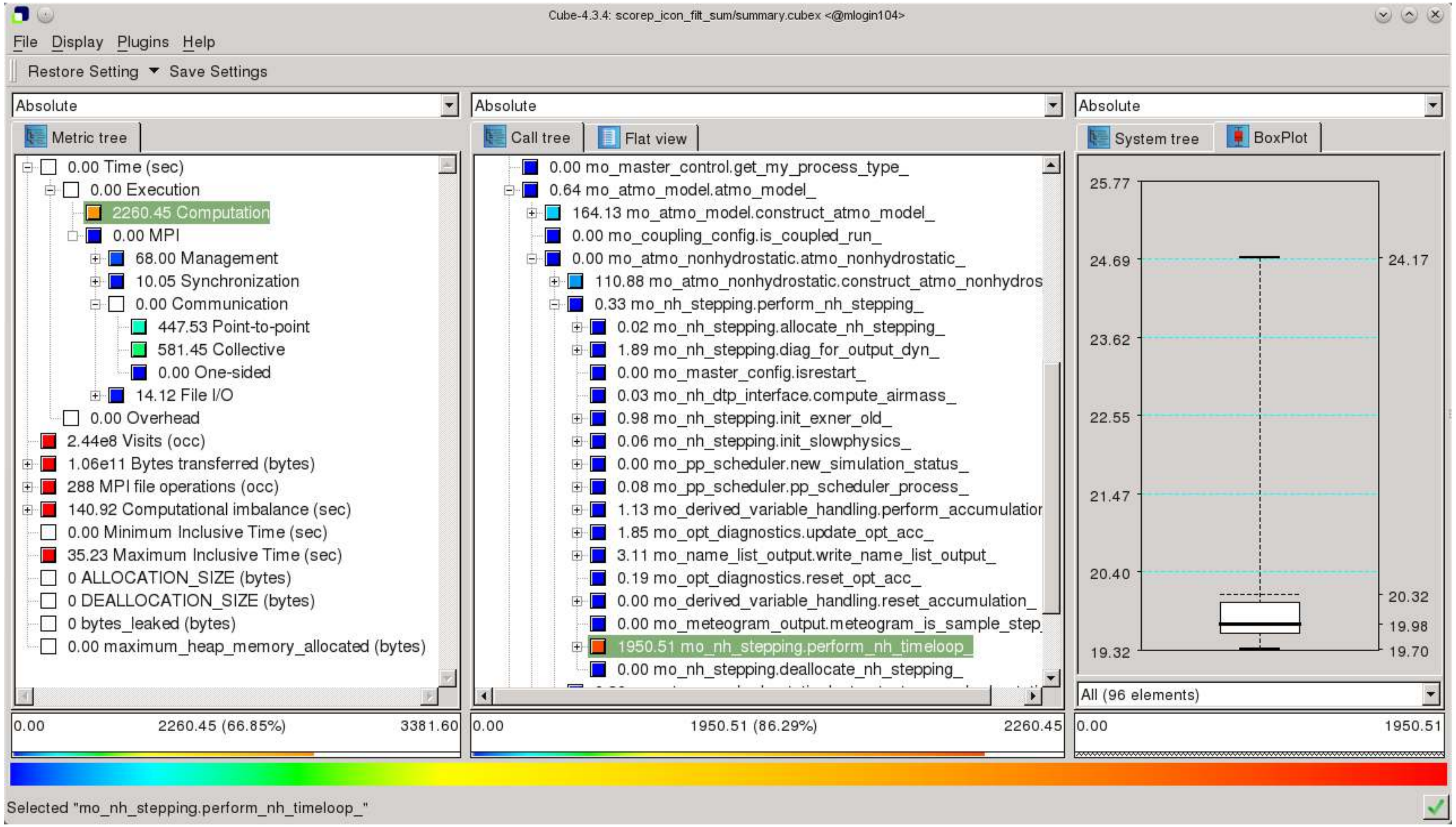
ICON instrumentation

- After configuration, adjusted compiler variables in top-level Makefile
  - Code parts written in C **not** instrumented

```
CC  = icc
FC  = scorep --user --mpp=mpi --thread=none ifort
F77 = scorep --user --mpp=mpi --thread=none ifort
```

- Initial instrumented run incurred ~120% overhead
- Preparing a good filter required several iterations
  - Filter out enough routines to achieve reasonable overhead…
  - …but not loose important information
- Overhead of filtered run <20%
  - Not perfect, but OK

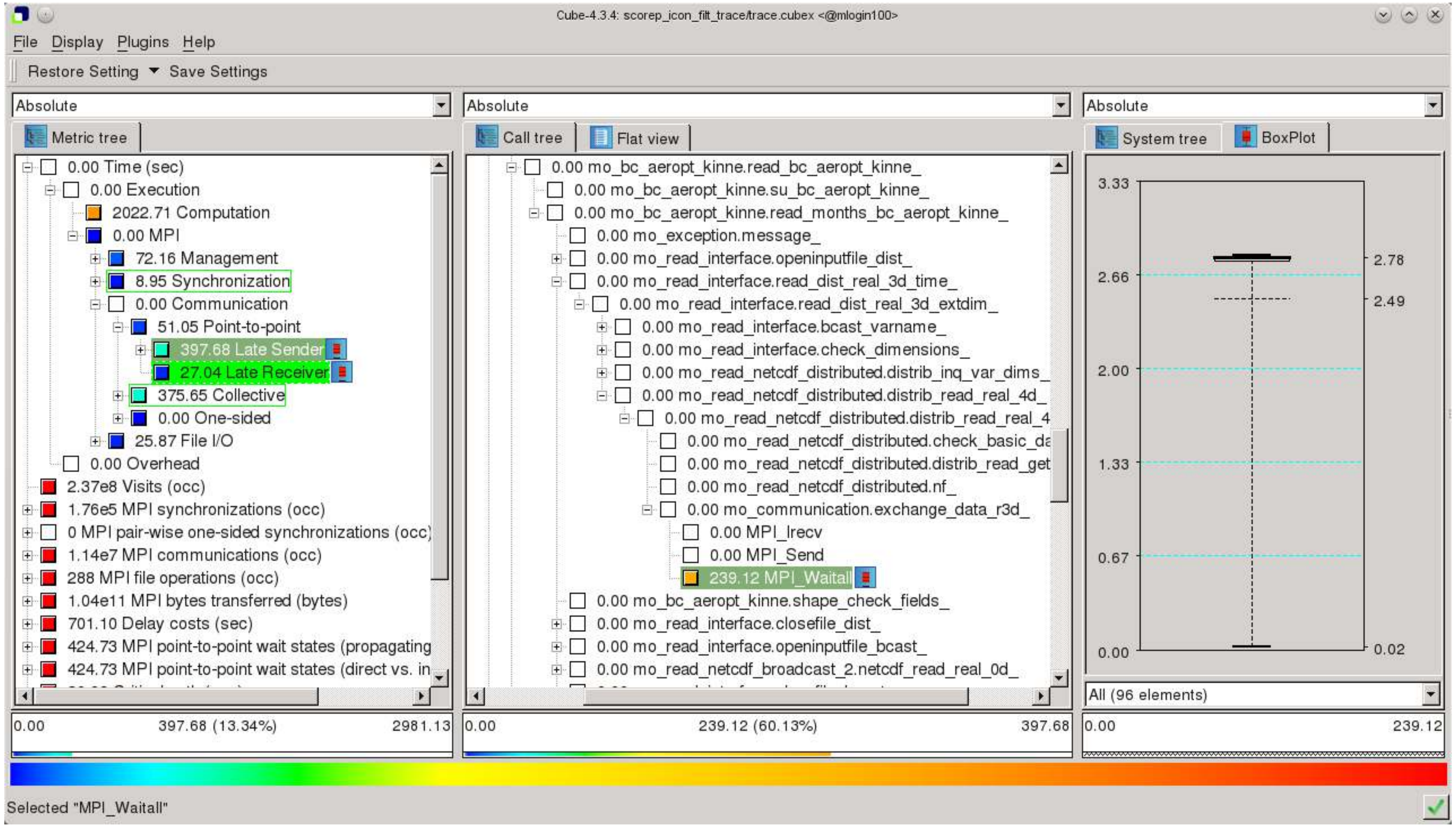
ICON summary profile analysis (4 compute nodes)



- 67% of time is computation
- >30% is MPI communication

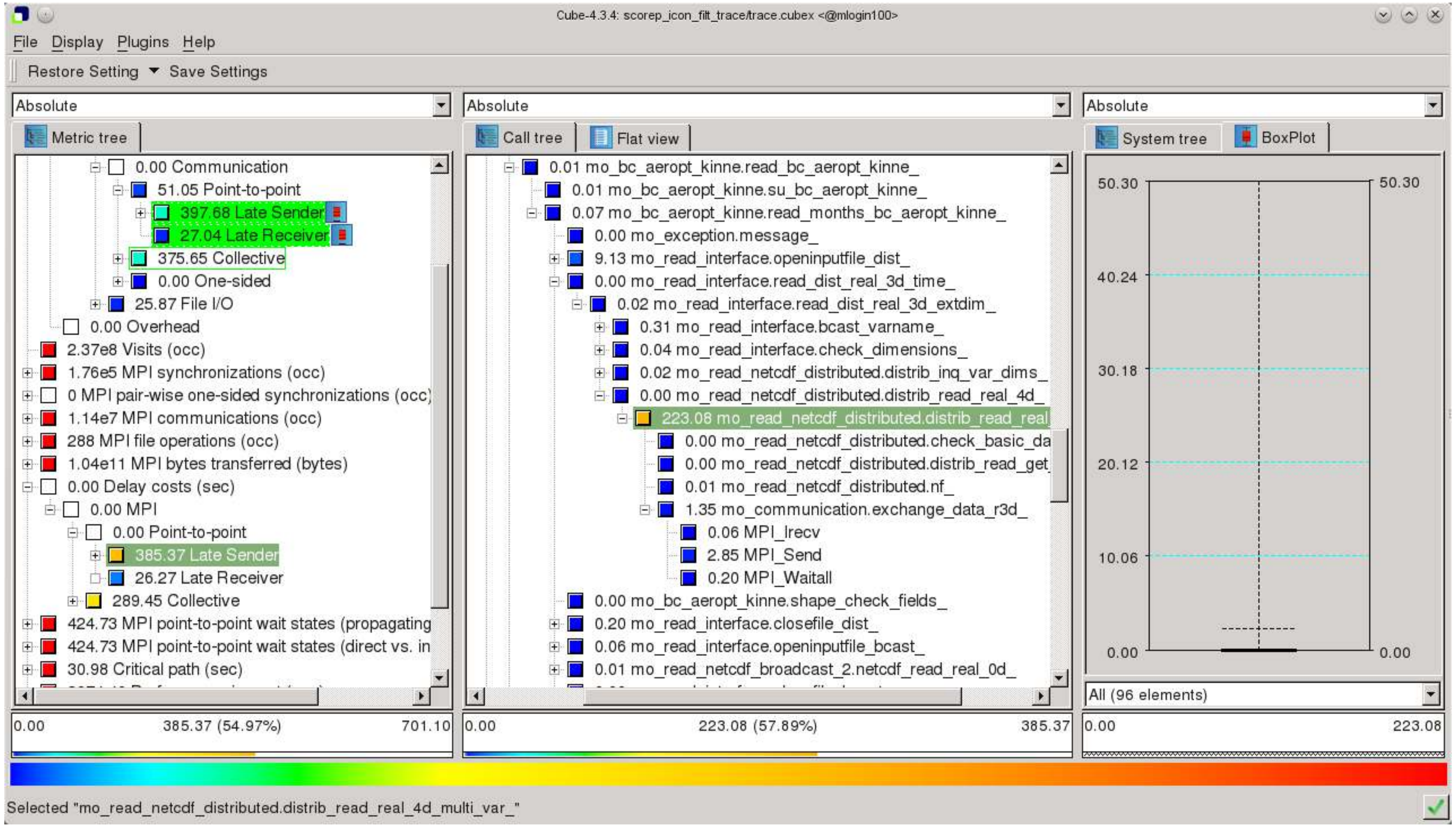
- 87% of computation is spent in timestep loop
  - …with quite some variation across ranks

ICON Scalasca trace analysis (4 compute nodes)



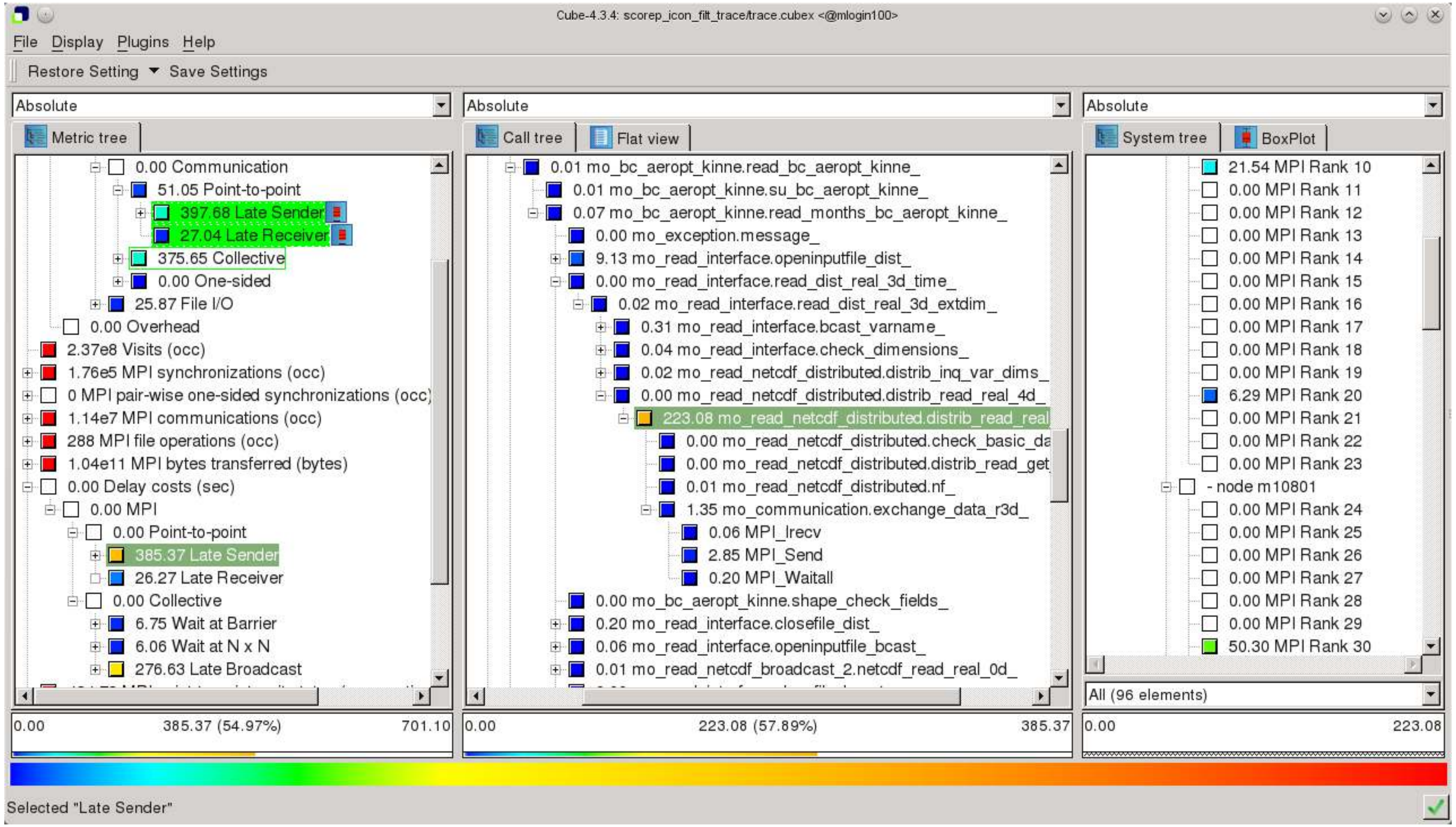
- Significant amount of *Late Sender* wait states
  - 13% of overall time
- 8% in **`MPI_Waitall`** called from **`exchange_data_r3d`**

ICON Scalasca trace analysis (cont.)



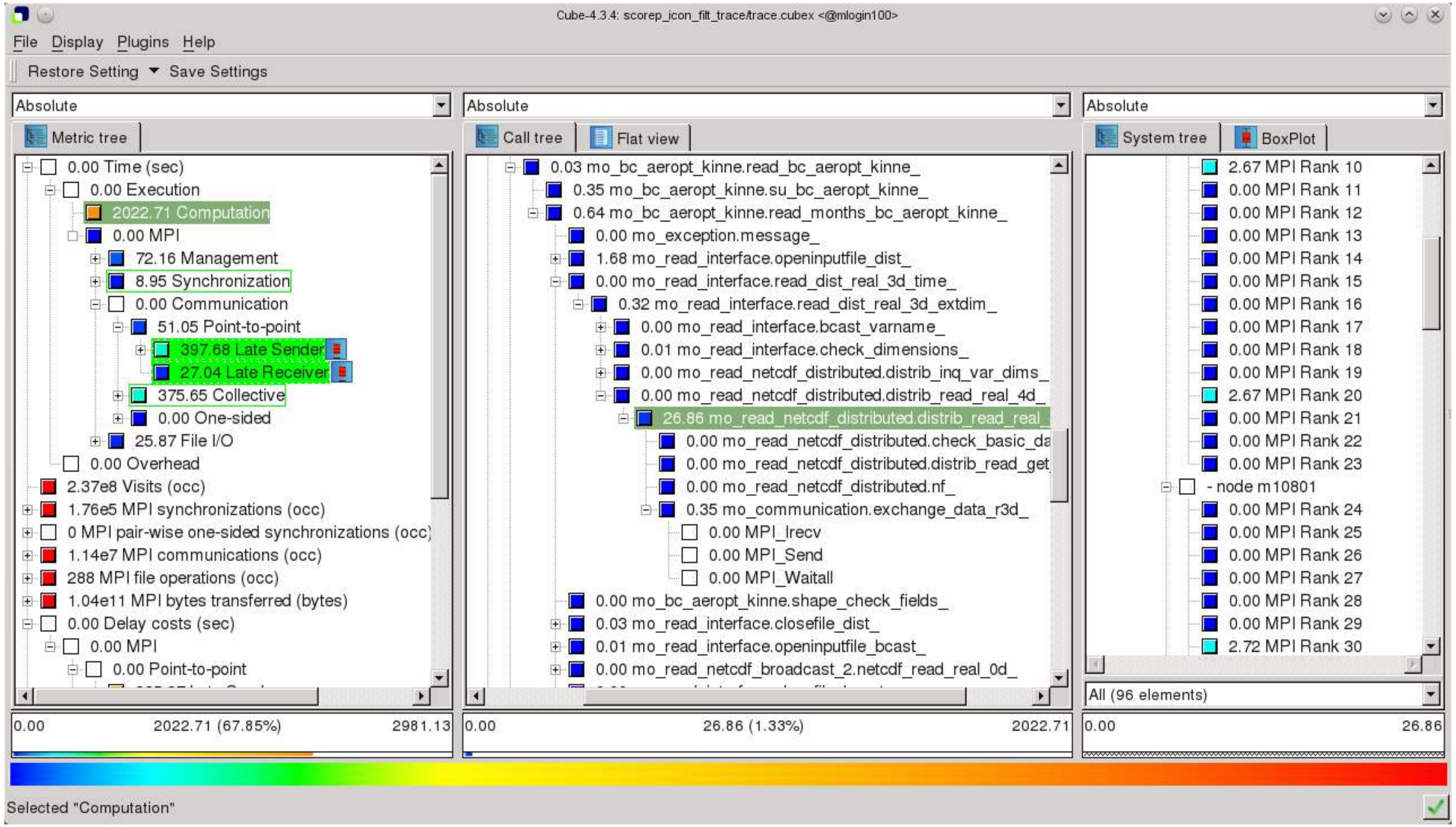
- …mostly due to imbalance in netCDF `distrib_read_real_4d_multi_var` …

ICON Scalasca trace analysis (cont.)



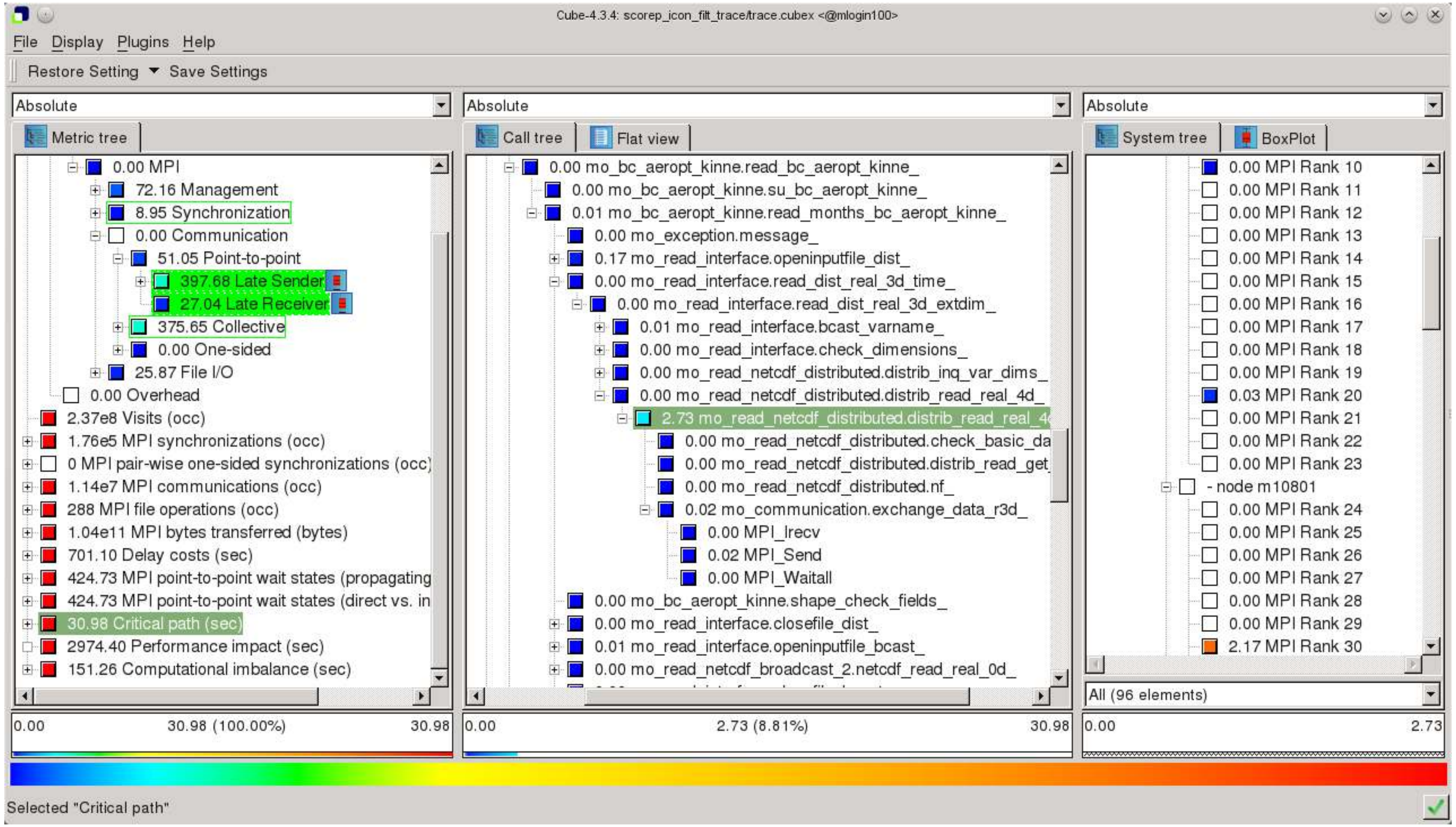
- …where every 10th rank causes wait states…

ICON Scalasca trace analysis (cont.)



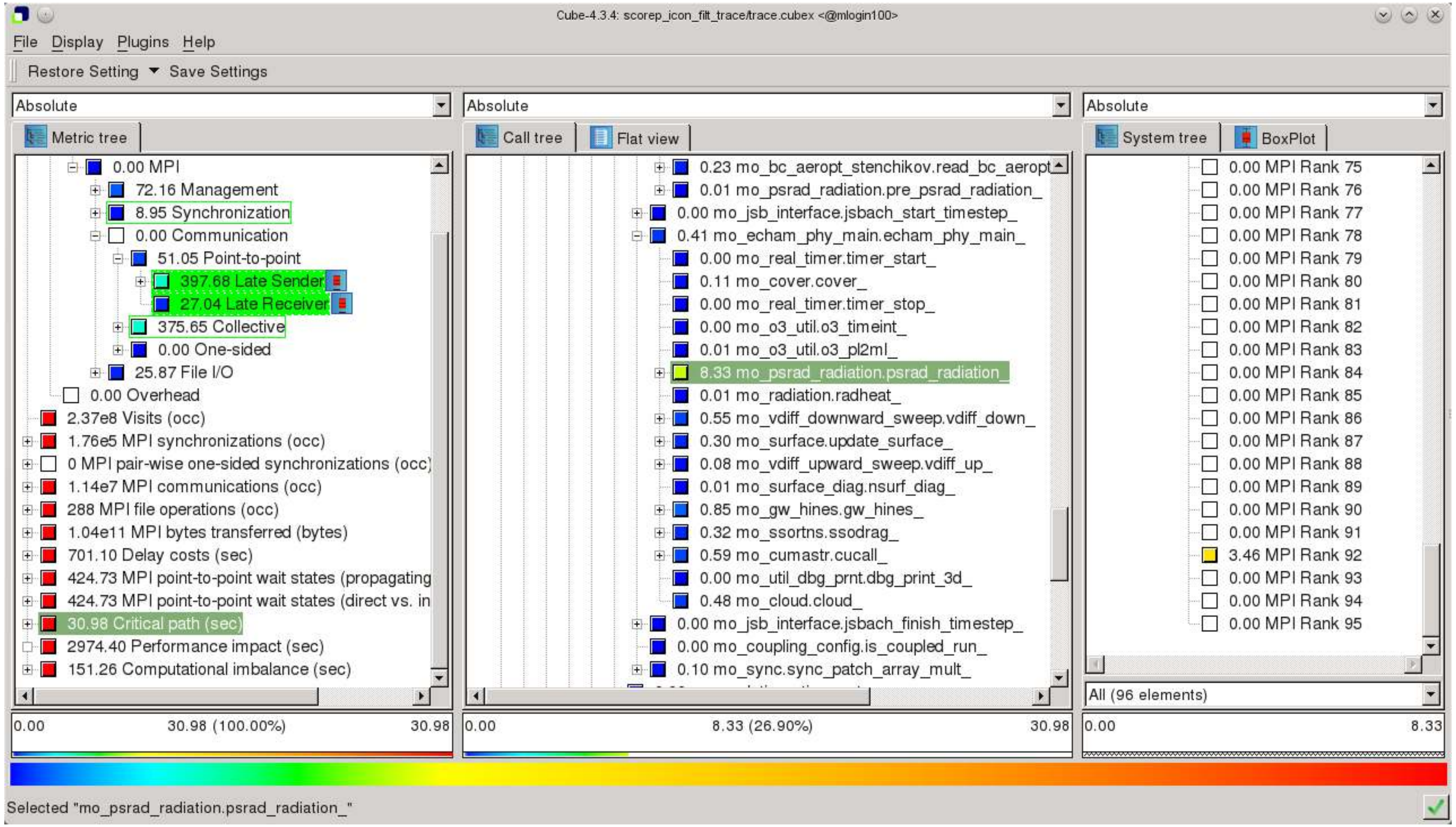
- …much larger than the imbalance itself!

ICON Scalasca trace analysis (cont.)



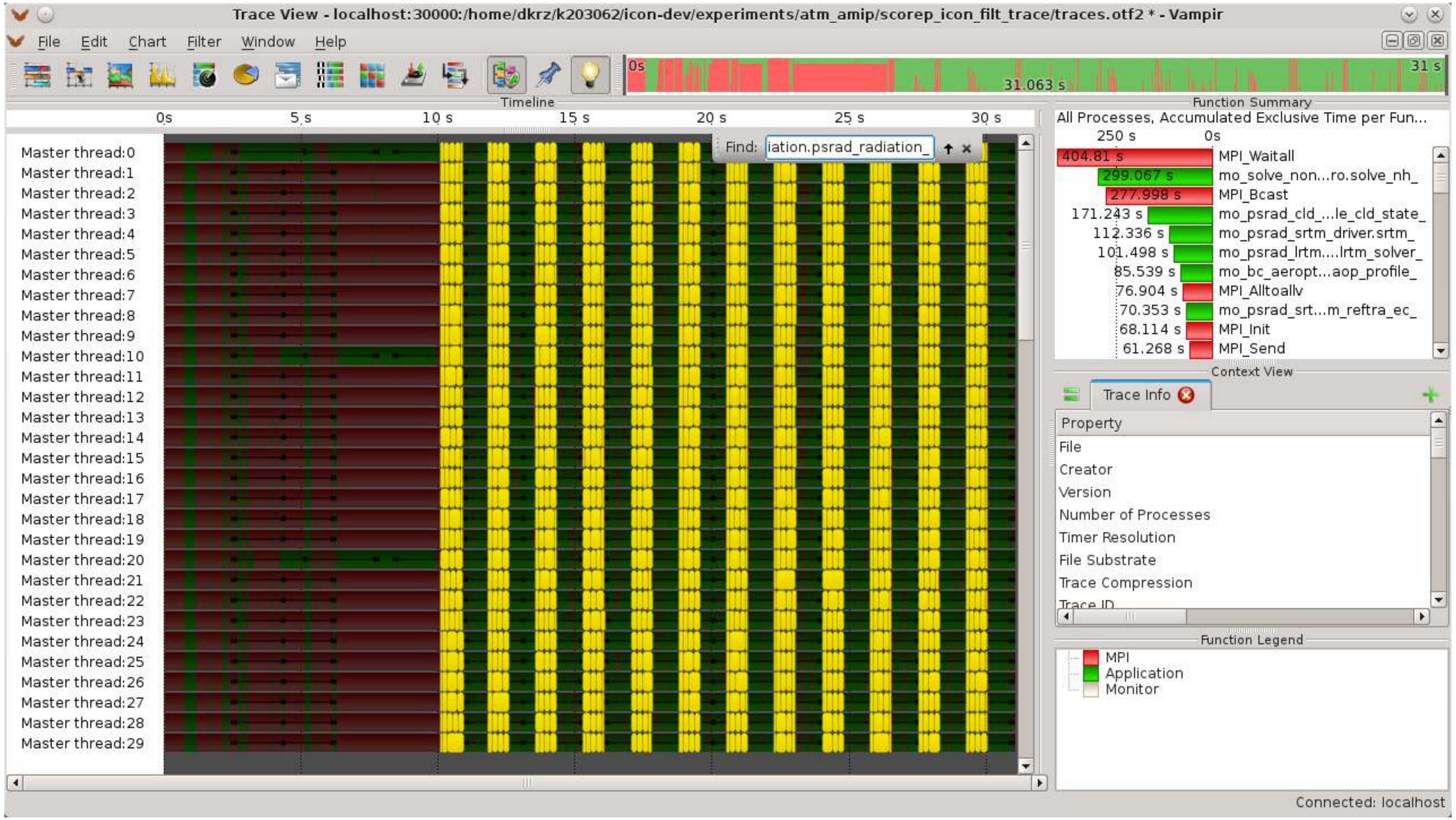
- This imbalance is also highlighted by the critical path…

ICON Scalasca trace analysis (cont.)



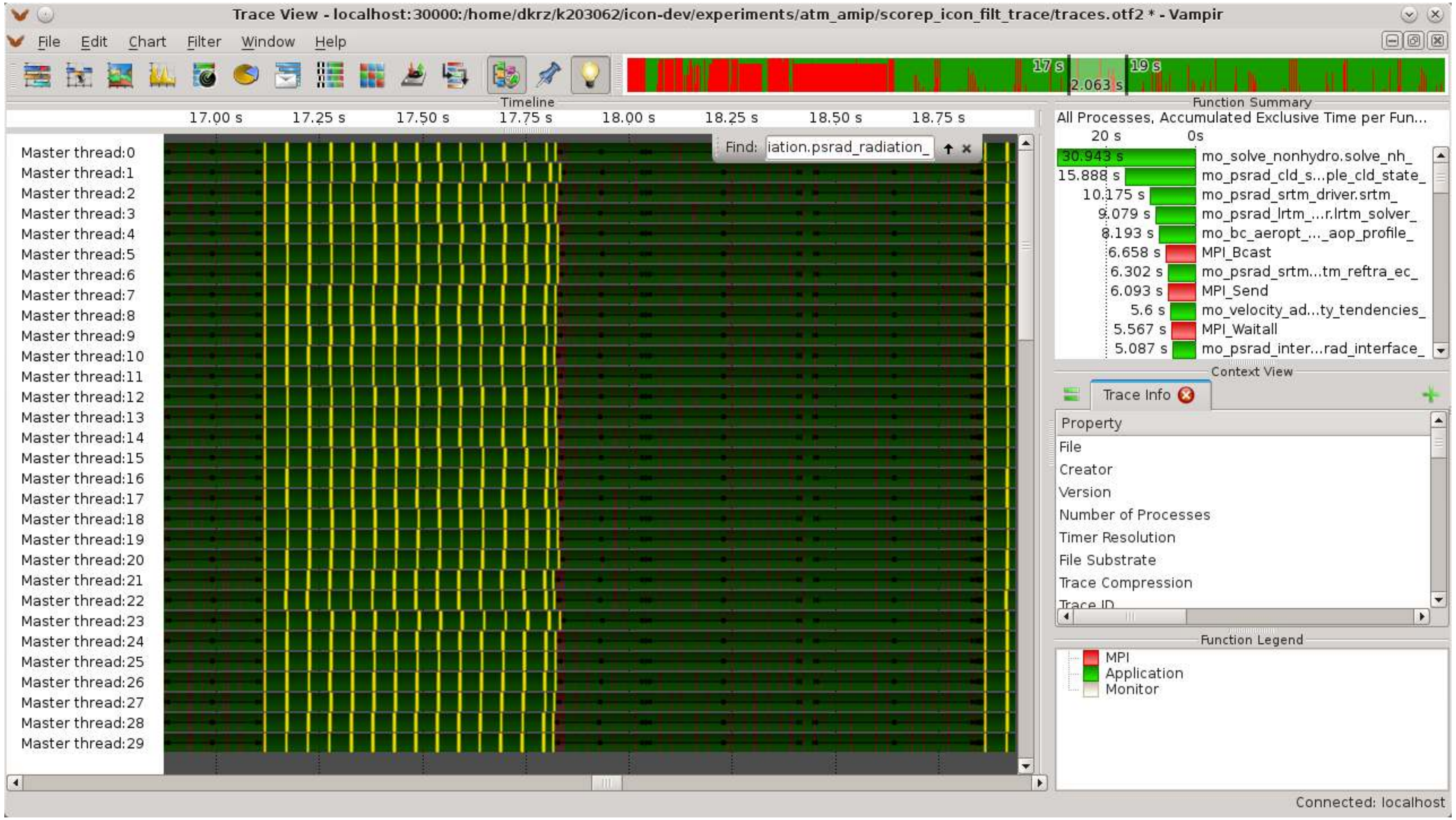
- …as well as the **`psrad_radiation`** routine which consumes >10% of the critical path time

ICON Vampir trace analysis



- **`psrad_radiation`** routine highlighted in trace

ICON Vampir trace analysis (cont.)

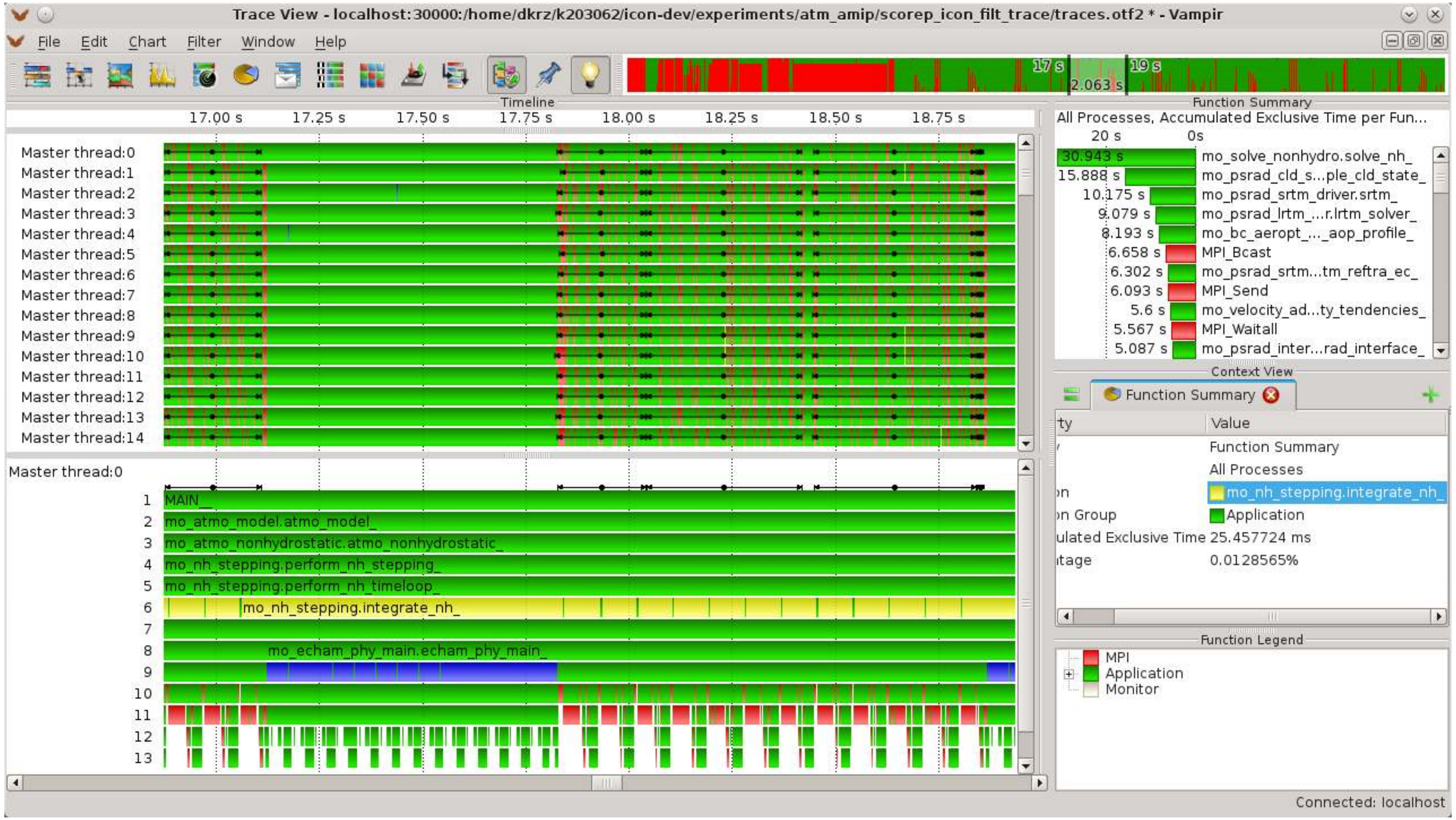


- …zoomed on one iteration block

Outline

- Case I:
  - NPB3.3_MZ_MPI/**BT-MZ** (MPI+OpenMP) on *MARCONI-KNL*: load balancing
- Case II:
  - **k-Wave** (MPI+OpenMP) on *Salomon*: load-balancing in FFTW OpenMP parallel regions
- Case III:
  - **ICON** (MPI) on *Mistral*: automatic trace analysis of critical path of execution
- Case IV:
  - **PIConGPU** (MPI+CUDA): computation offload to multiple attached accelerator devices
- Case V:
  - **TensorFlow** (Python+CUDA): interpreted & compiled heterogeneous execution measurement

ICON Vampir trace analysis (cont.)

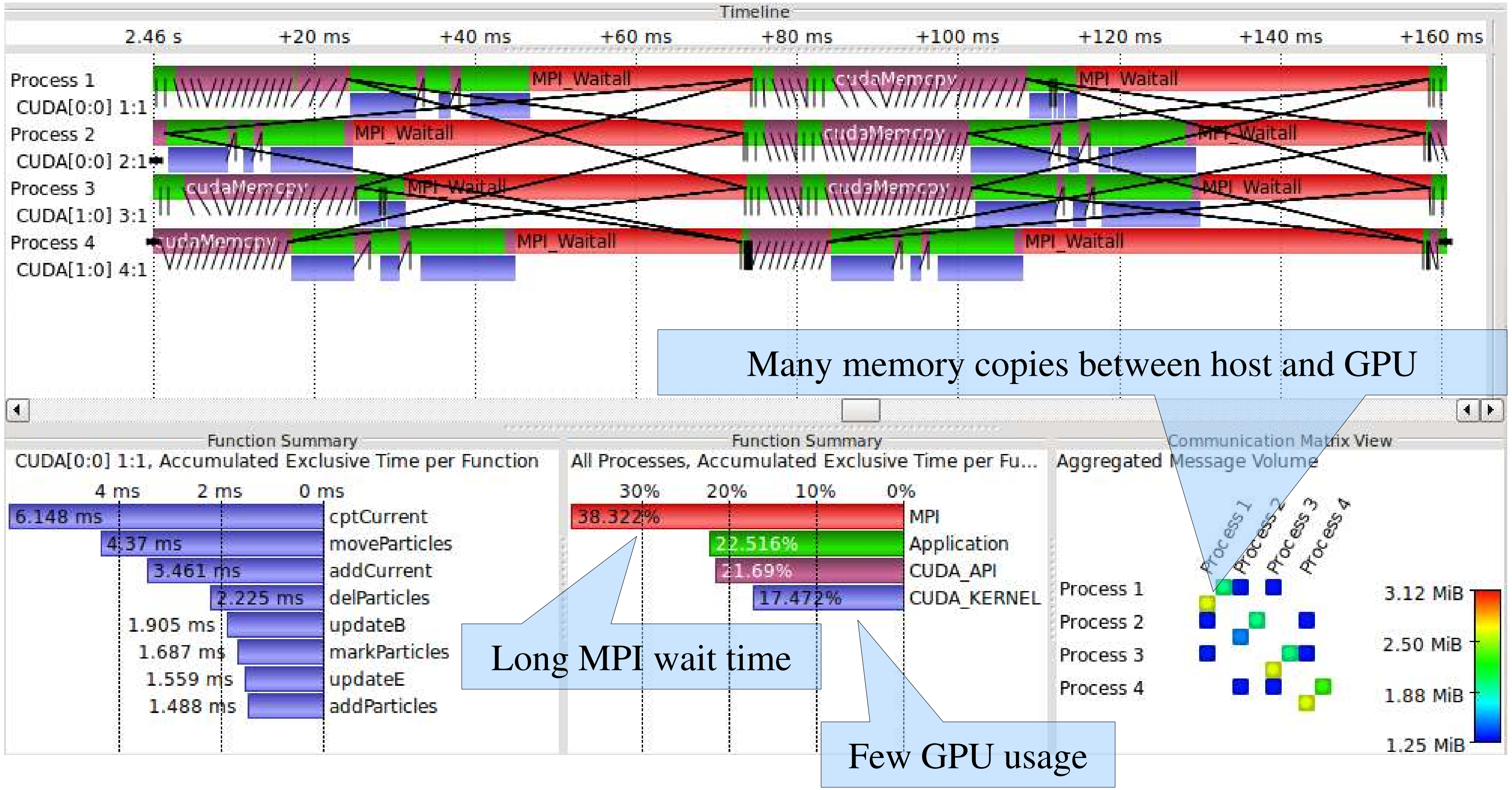


- ...with process timeline showing that **`psrad_radiation`** (blue) is called every 12th **`integrate_nh`** iteration (yellow)

Case IV: PIConGPU

- A fully-relativistic 3D3V plasma physics particle-in-cell code for many GPGPUs, developed by HZDR in collaboration with ZIH, TU Dresden
  https://github.com/ComputationalRadiationPhysics/picongpu
- Incremental software evolution
  - C++ & CUDA with MPI
- Continuous performance analysis and optimization
  - 2013 Gordon Bell Prize finalist for outstanding performance and scalability to over 18,000 GPGPUs

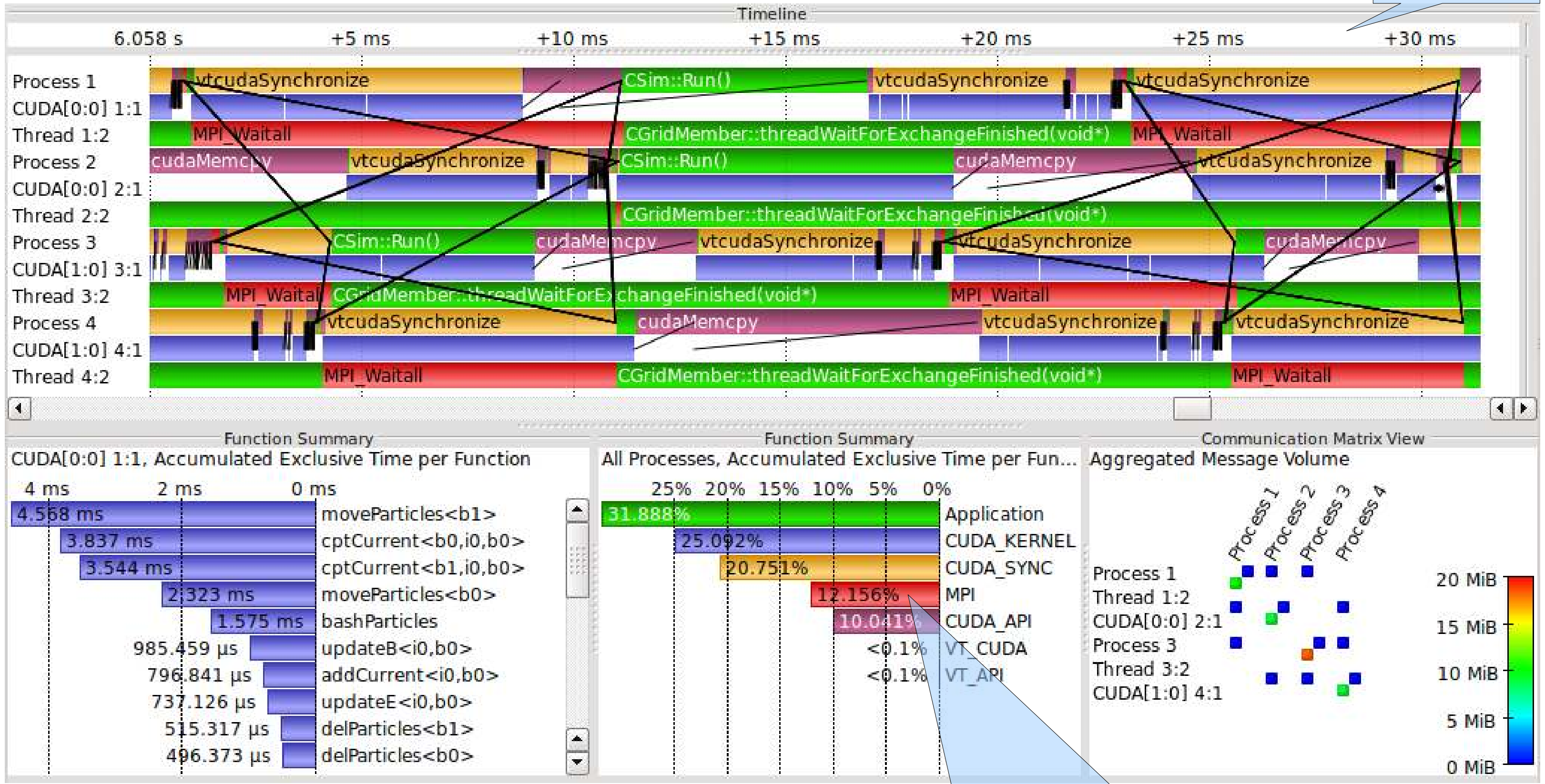
First parallel PIConGPU implementation (1 run step)



PIConGPU I (1 run step)

- General software design improvements

Sp ≈ 4

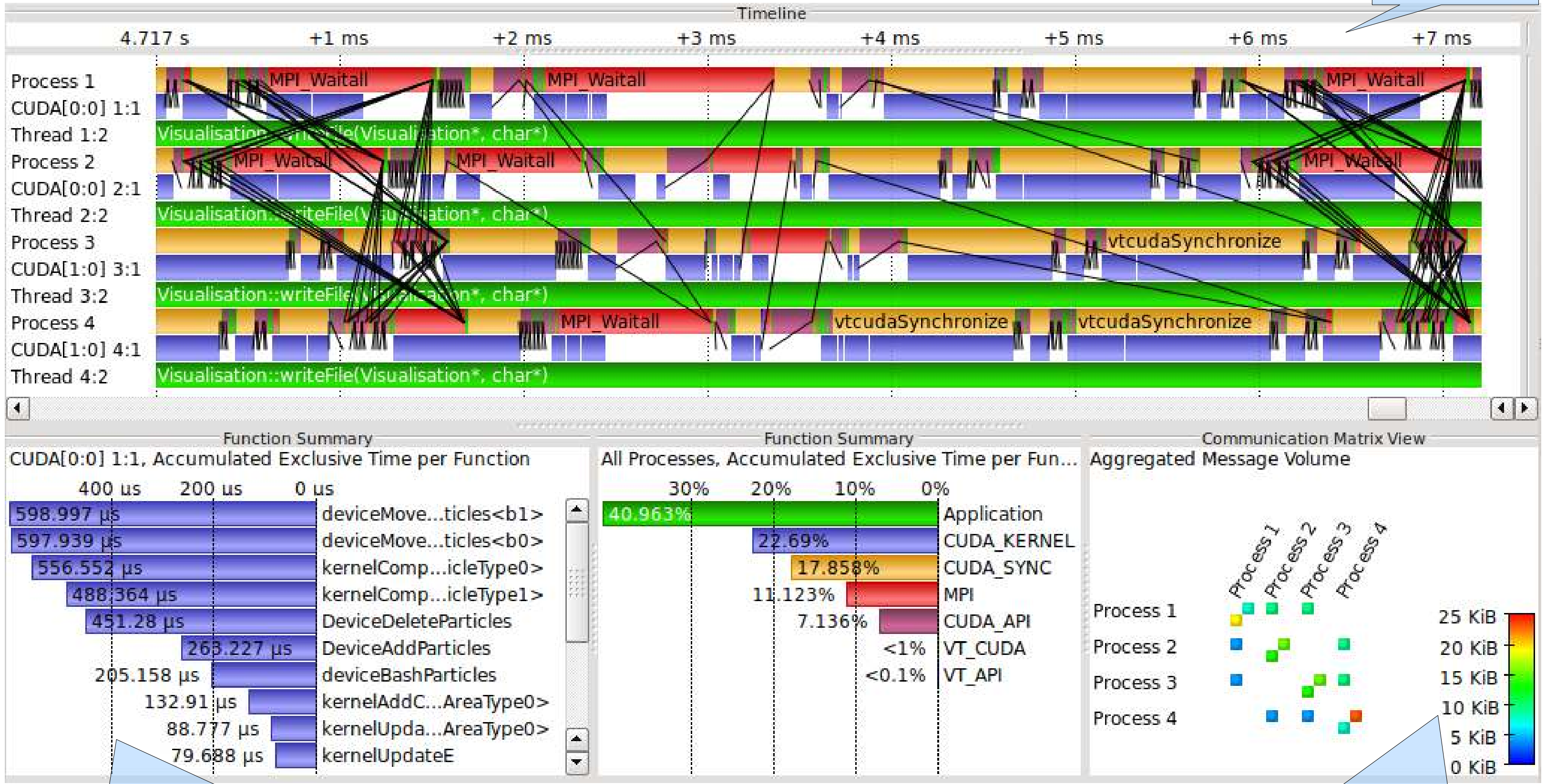


Dramatically reduced MPI wait time

PIConGPU II (1 run step)

- Software re-design

Sp ≈ 4



shorter kernel execution times

smaller messages (1 order)

PIConGPU – Today

Usage of CUDA streams

Small GPU compute idle

No MPI wait, just short MPI_Test

Outline

- Case I:
  - NPB3.3_MZ_MPI/**BT-MZ** (MPI+OpenMP) on *MARCONI-KNL*: load balancing
- Case II:
  - **k-Wave** (MPI+OpenMP) on *Salomon*: load-balancing in FFTW OpenMP parallel regions
- Case III:
  - **ICON** (MPI) on *Mistral*: automatic trace analysis of critical path of execution
- Case IV:
  - **PIConGPU** (MPI+CUDA): computation offload to multiple attached accelerator devices
- Case V:
  - **TensorFlow** (Python+CUDA): interpreted & compiled heterogeneous execution measurement

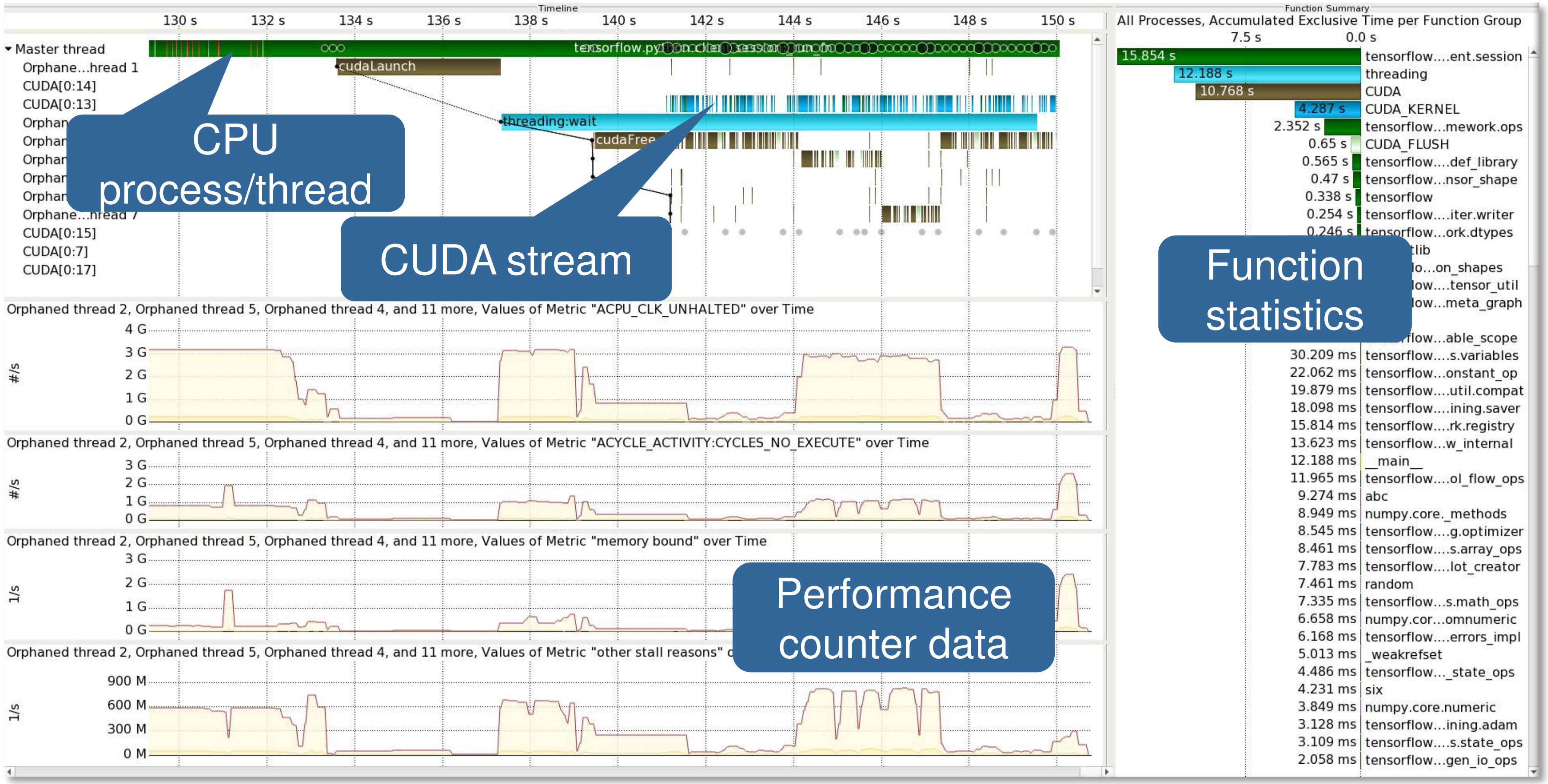
Case V: TensorFlow

- TensorFlow is one of the most popular Deep Learning frameworks
  - also the foundation of other tools, e.g., Keras
- Use additional Python bindings for Score-P to obtain execution performance data
  - available at https://github.com/score-p/scorep_binding_python
  - CUDA activities are recorded using CUPTI
- Execution of a single TensorFlow process on a workstation with a single GPU device, forking multiple threads

⇒ ***Optimized execution using NumPy array of doubles vs. native array***

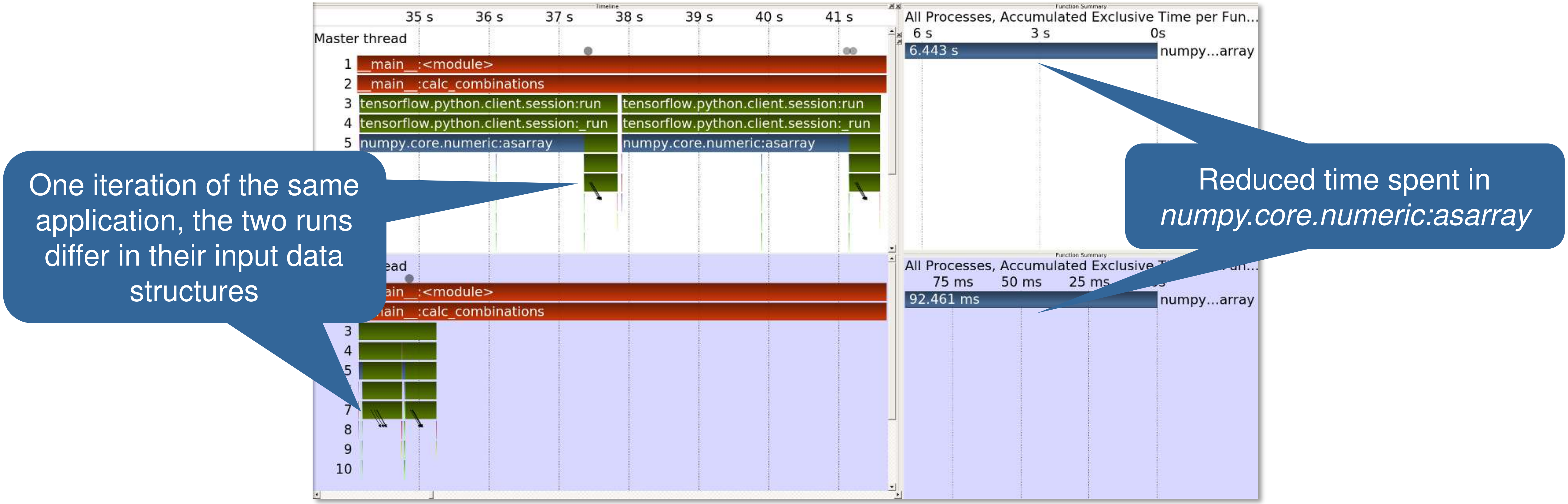
TensorFlow

- Application process, its threads, and CUDA streams with corresponding performance counter data



TensorFlow

- Comparison view of the original (top) and optimized (bottom) application run.



Summary

- Score-P instrumentation & measurement infrastructure is proven to be extremely scalable, portable and flexible
- Basis for diverse execution performance analyses with Scalasca, TAU & Vampir
- Successfully used with a wide variety of parallel applications

- Small representative selection:
  - NPB3.3_MZ_MPI/**BT-MZ** (MPI+OpenMP) on *MARCONI-KNL*: load balancing
  - **k-Wave** (MPI+OpenMP) on *Salomon*: load-balancing in FFTW OpenMP parallel regions
  - **ICON** (MPI) on *Mistral*: automatic trace analysis of critical path of execution
  - **PIConGPU** (MPI+CUDA): computation offload to multiple attached accelerator devices
  - **TensorFlow** (Python+CUDA): interpreted & compiled heterogeneous execution measurement

Review

Markus Geimer
Jülich Supercomputing Centre

Summary

You've been introduced to a variety of tools
- with hints to apply and use the tools effectively

Tools provide complementary capabilities
- computational kernel & processor analyses
- communication/synchronization analyses
- load-balance, scheduling, scaling, …

Tools are designed with various trade-offs
- general-purpose versus specialized
- platform-specific versus agnostic
- simple/basic versus complex/powerful

Tool selection

Which tools you use and when you use them likely to depend on the situation

- which are available on (or for) your computer system
- which support your programming paradigms and languages
- which you are familiar (comfortable) with using
- which type of issue you suspect
- which question you want to have answered

Being aware of (potentially) available tools and their capabilities can help finding the most appropriate tools

Workflow (getting started)

First ensure that the parallel application runs correctly

- no-one will care how quickly you can get invalid answers or produce a set of corefiles
- parallel debuggers help isolate known problems
- correctness checking tools can identify other issues
- (that might not cause problems right now, but will eventually)
  - e.g., race conditions, invalid/non-compliant usage

Best to start with an overview of execution performance

- fraction of time spent in computation vs comm/synch vs I/O
- which sections of the application/library code are most costly
- Example profilers: **Score-P + Cube/ParaProf, TAU**

and how it changes with scale or different configurations

- processes vs threads, mappings, bindings

Workflow (communication/synchronization)

Communication issues generally apply to every computer system (to different extents) and typically grow with the number of processes/threads

- Weak scaling: fixed computation per thread, and perhaps fixed localities, but increasingly distributed
- Strong scaling: constant total computation, increasingly divided amongst threads, while communication grows
- Collective communication (particularly of type “all-to-all”) result in increasing data movement
- Synchronizations of larger groups are increasingly costly
- Load-balancing becomes increasingly challenging, and imbalances more expensive
  - generally manifests as waiting time at following collective ops

Workflow (wasted waiting time)

Waiting times are difficult to determine in basic profiles

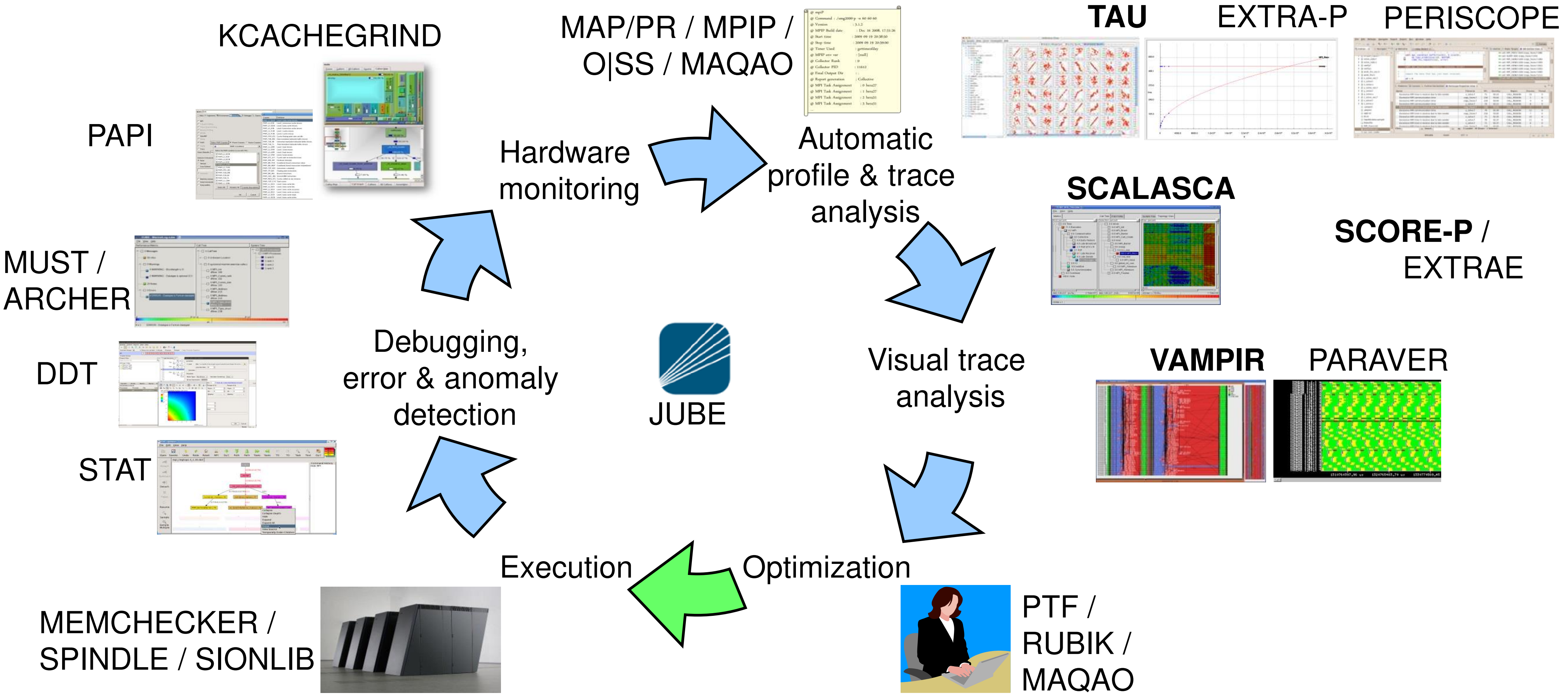
- Part of the time each process/thread spends in communication & synchronization operations may be wasted waiting time
- Need to correlate event times between processes/threads
  - Periscope uses augmented messages to transfer timestamps plus on-line analysis processes
  - Post-mortem event trace analysis avoids interference and provides a complete history
  - **Scalasca** automates trace analysis and ensures waiting times are completely quantified
  - **Vampir** allows interactive exploration and detailed examination of reasons for inefficiencies

Workflow (core computation)

Effective computation within processors/cores is also vital

- Optimized libraries may already be available
- Optimizing compilers can also do a lot
  - provided the code is clearly written and not too complex
  - appropriate directives and other hints can also help
- Processor hardware counters can also provide insight
  - although hardware-specific interpretation required
- Tools available from processor and system vendors help navigate and interpret processor-specific performance issues

Technologies and their integration



Further information

Website

- Introductory information about the VI-HPS portfolio of tools for high-productivity parallel application development
  - VI-HPS Tools Guide
  - links to individual tools sites for details and download
- Training material
  - tutorial slides
  - latest ISO image of VI-HPS Linux DVD with productivity tools
  - user guides and reference manuals for tools
- News of upcoming events
  - tutorials and workshops
  - mailing-list sign-up for announcements

http://www.vi-hps.org

Acknowledgement

Many thanks to the

Texas Advanced Computing Center (TACC)

for sponsoring this tutorial by providing
training accounts and compute time!