ISC'22 Tutorial: Practical Hybrid Parallel Application Performance Engineering

Sameer Shende
University of Oregon

Markus Geimer
Jülich Supercomputing Centre

Bill Williams
Technische Universität Dresden

Half-day tutorial logistics

- Normally this is a full-day tutorial featuring *hands-on* exercises using an E4S container image and a remote HPC system
  - AWS instances are available with all of the presented tools installed
  - Exercise materials and slides are provided for you to do on your own
  - But insufficient time to include everything in a half-day tutorial for a class

- Instead we'll be giving **live demonstrations** of our tools
  - Showing how they are used and the key functionality that they offer
  - Using provided example measurements allow to follow along
  - Refer to the tutorial slides for additional details we don't have time to cover (marked by the book icon  in the upper right)

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

HELMHOLTZ
RESEARCH FOR GRAND CHALLENGES

- Activities
  - Development and integration of HPC application tools
    - Primarily correctness checking & performance analysis
  - Academic workshops: e.g. ProTools@SC21 (Sunday 14 November 2021)
  - Tools training via conference tutorials and multi-day "bring-your-own-code" Tuning Workshops
    - Face-to-face & side-by-side hands-on coaching now successfully migrated to virtual/on-line events

https://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.)

ARM Ltd.

- Allinea software

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.)

Friedrich-Alexander-Universität

- Erlangen Regional Computing Center (RRZE)

Technical University of Munich

- Chair for Computer Architecture

University of Oregon

- Performance Research Laboratory

University of Stuttgart

- HPC Centre

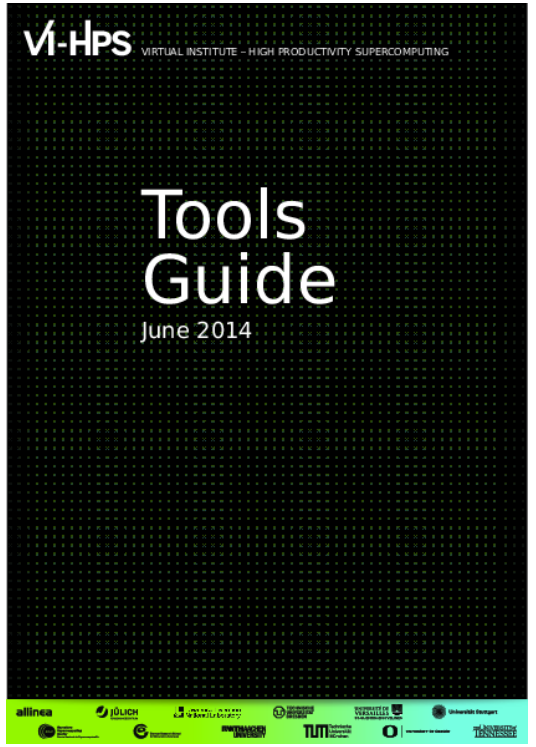
University of Versailles St-Quentin

- LRC ITACA

Productivity tools

- MUST / ARCHER
  - MPI & OpenMP usage correctness checking
- PAPI
  - Interfacing to hardware performance counters
- **CUBE**
  - Analysis report exploration & processing
- **Scalasca**
  - Large-scale parallel performance analysis
- **TAU**
  - Integrated parallel performance system
- **Vampir**
  - Interactive graphical trace visualization & analysis
- **Score-P**
  - Community-developed instrumentation & measurement infrastructure

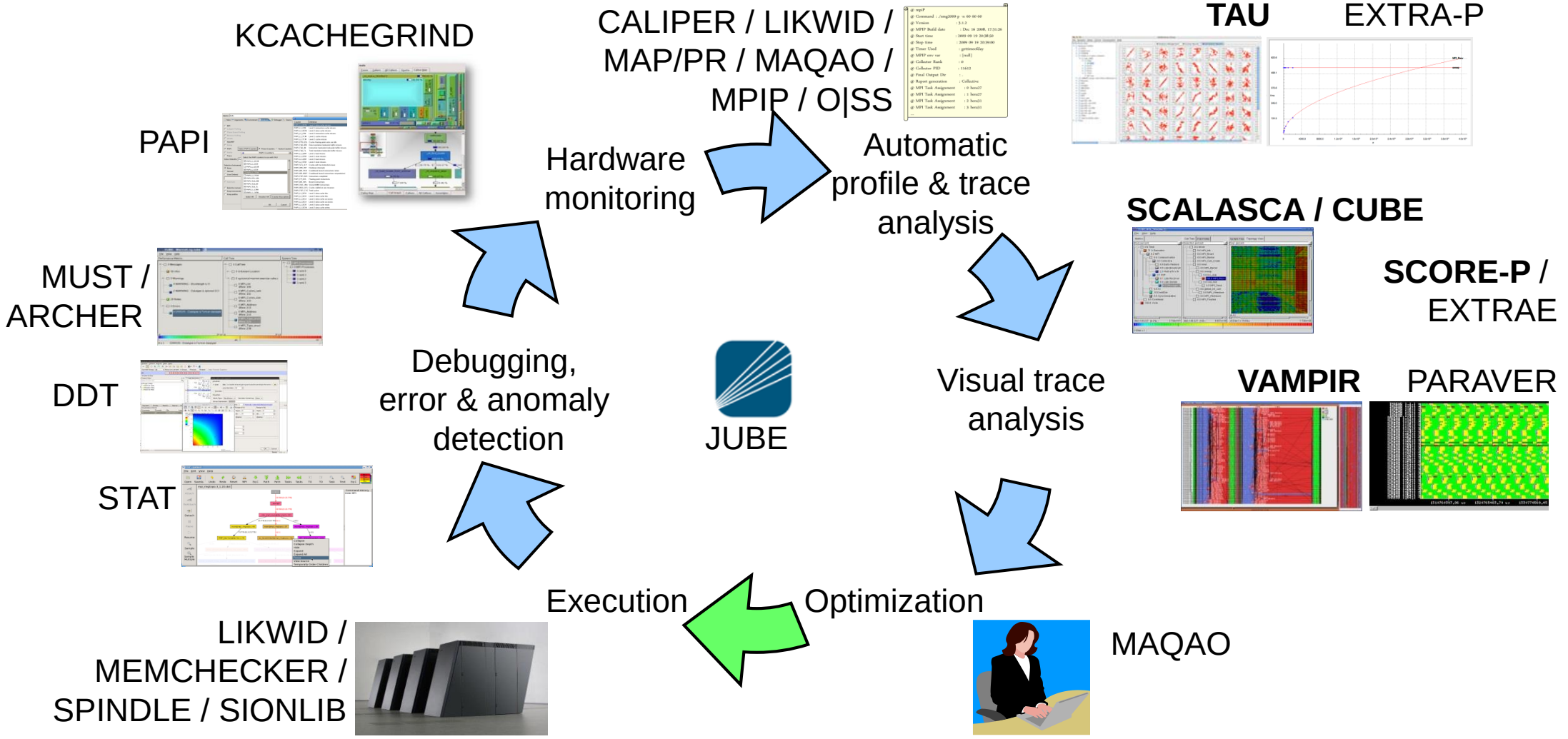
For a brief overview of tools consult the VI-HPS Tools Guide:



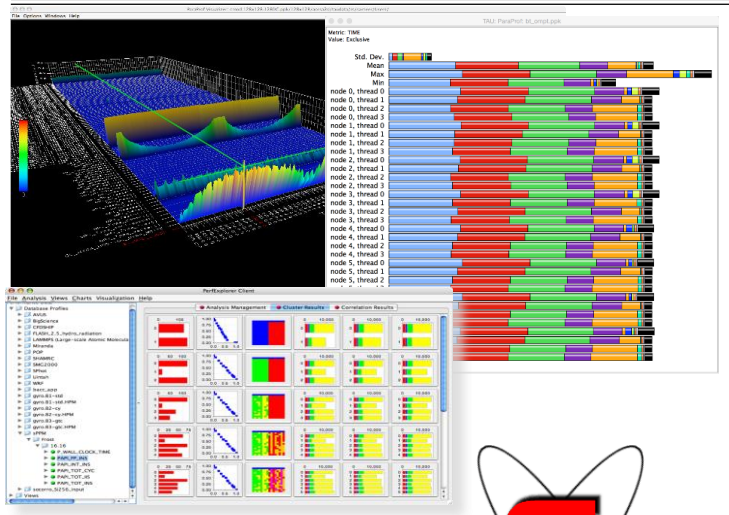
Productivity tools (cont.)

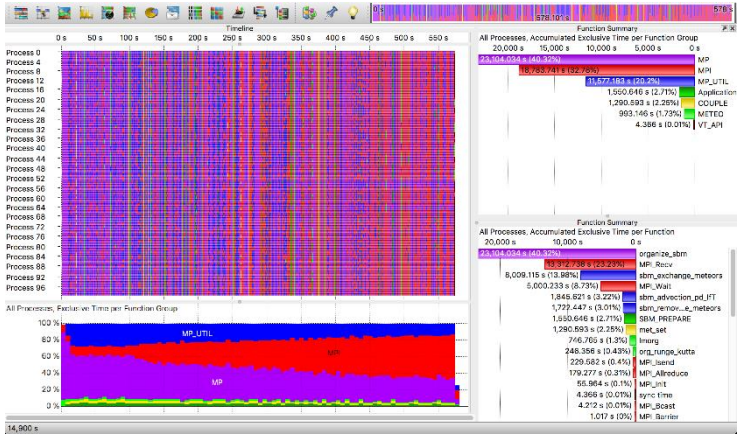
- Caliper: Library for application event annotation, logging and profiling
- Extra-P: Automated performance modelling
- FORGE DDT/MAP/PR: Parallel debugging, profiling & performance reports
- JUBE: Automatic workflow execution for benchmarking, testing & production
- Kcachegrind: Callgraph-based cache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

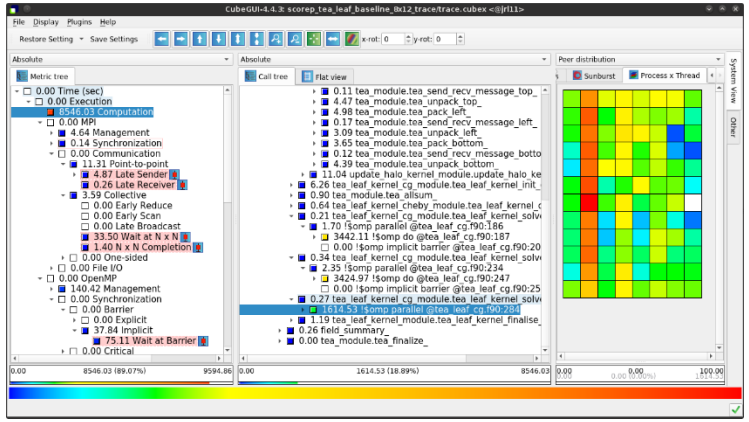
Technologies and their integration



Tools featured in this tutorial







Tuning and Analysis Utilities

VAMPIR

scalasca

Score-P
Scalable performance measurement infrastructure for parallel codes

Agenda

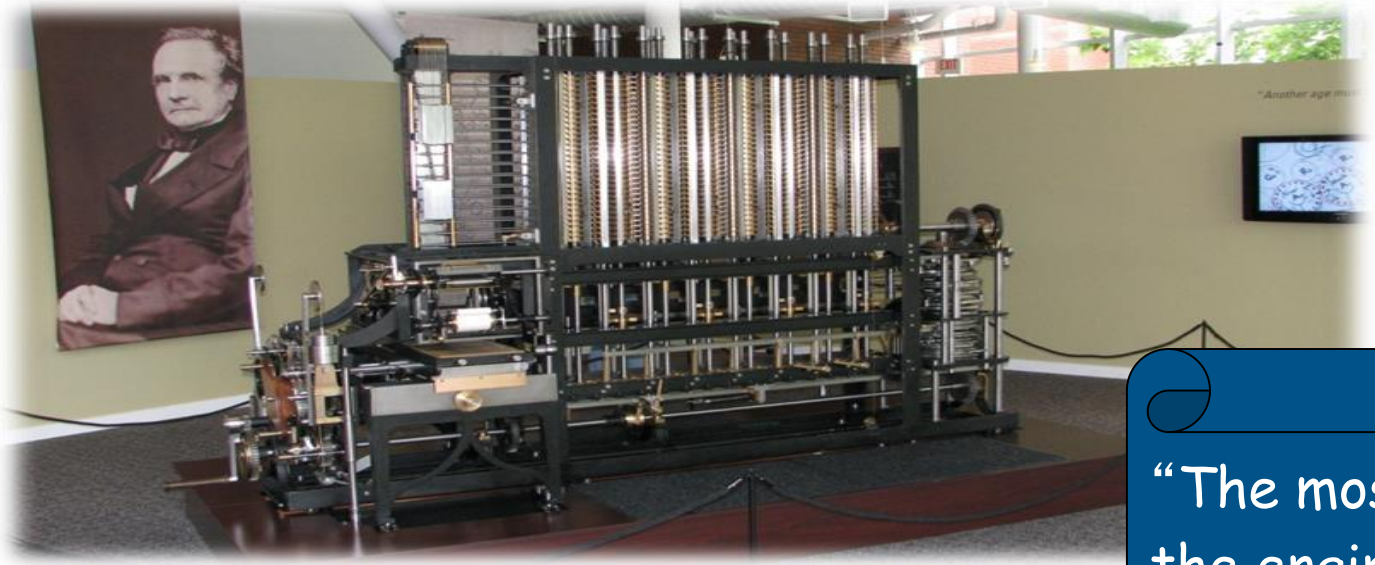
Time	Topic	Presenter
09:00	Welcome, Introduction to VI-HPS & Setup	Shende
09:15	Introduction to parallel performance engineering	Shende
09:45	Instrumentation & measurement with ***Score-P***	Williams
10:30	Exploration of execution call-path profiles with ***CUBE***	Geimer
11:30		
11:00	*Break*	
11:30	Interactive event trace examination & analysis with ***Vampir***	Williams
12:00	Event trace collection & automated analysis with ***Scalasca***	Geimer
12:30	Examination of profiles with ***TAU ParaProf & PerfExplorer***	Shende
13:00	*Adjourn*	

Introduction to Parallel Performance Engineering

Sameer Shende
University of Oregon

(with content used with permission from tutorials
by Bill Williams/TU Dresden,
Bernd Mohr/JSC and Luiz DeRose/Oracle)

Performance: an old problem



Difference Engine

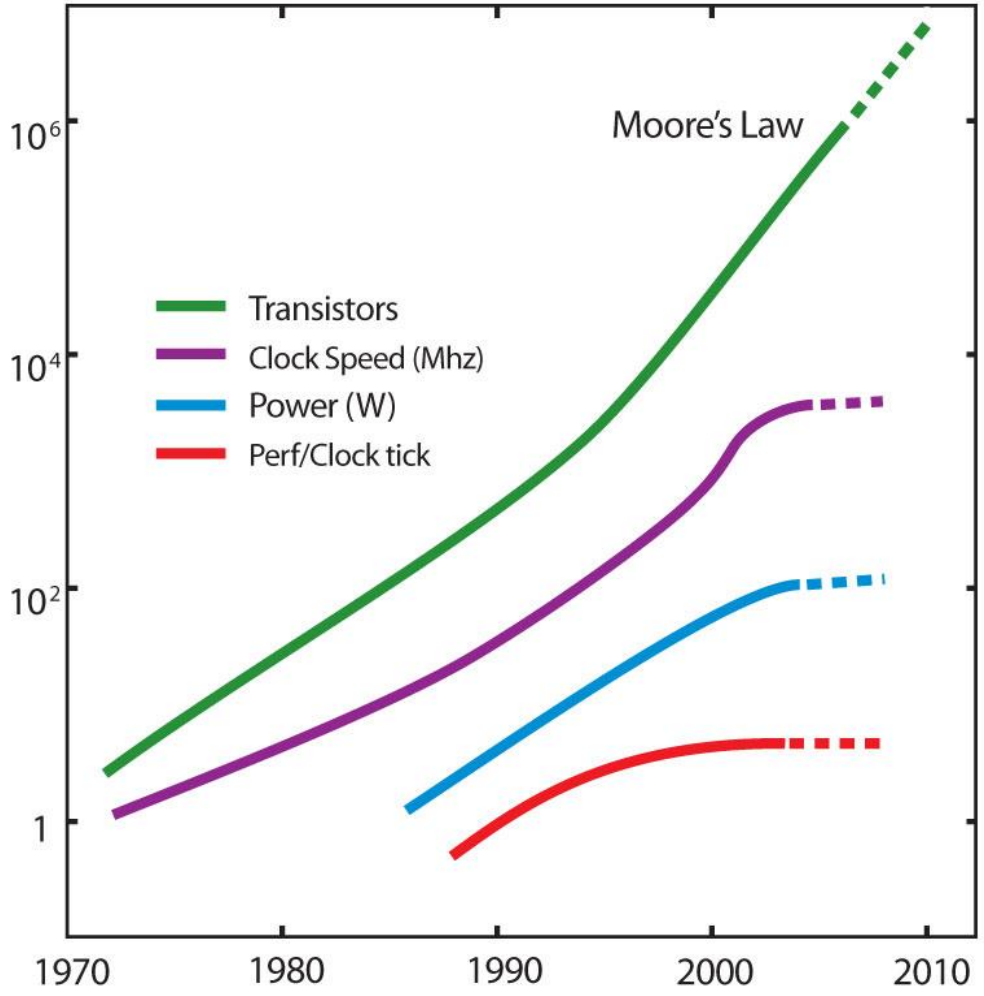
"The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible."

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!

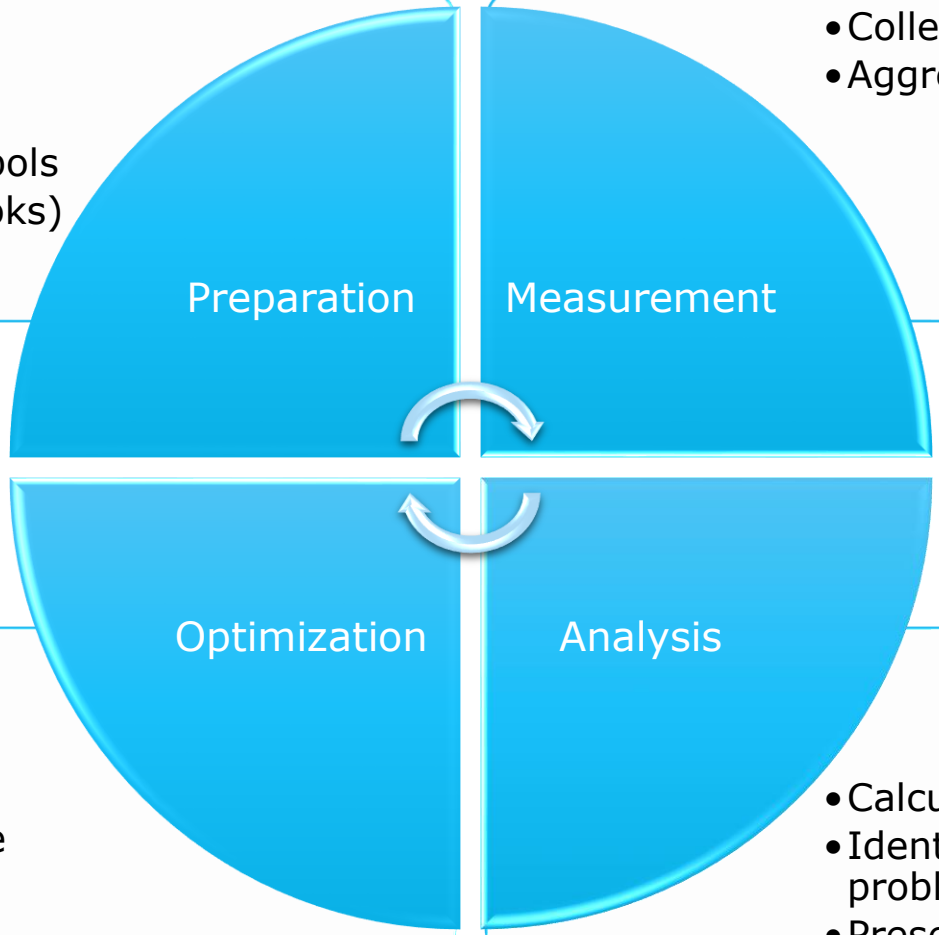


Performance factors of parallel applications

- "Sequential" performance factors
  - Computation
  - Cache and memory
  - Input / output
- "Parallel" performance factors
  - Partitioning / decomposition
  - Communication (i.e., message passing)
  - Multithreading
  - Synchronization / locking

Performance engineering workflow

- Build model of predicted performance
- Select data to measure
- 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

Parallel Performance Engineering in Practice

- Starting point: well-understood, well-optimized code at scale N
- Goal: scale to M >> N

- Predict behavior: what is the current bottleneck, what performance should we see?

- Measure possible bottlenecks
  - Idle resources
  - Changes in profile

- Minimize perturbation
  - May require multiple measurements!

Performance Modeling: Predicting Behavior

- Simplest models: scaling properties
  - What parts of the code are serial and parallel?
  - How much time is spent in each?
  - How efficient are they currently?

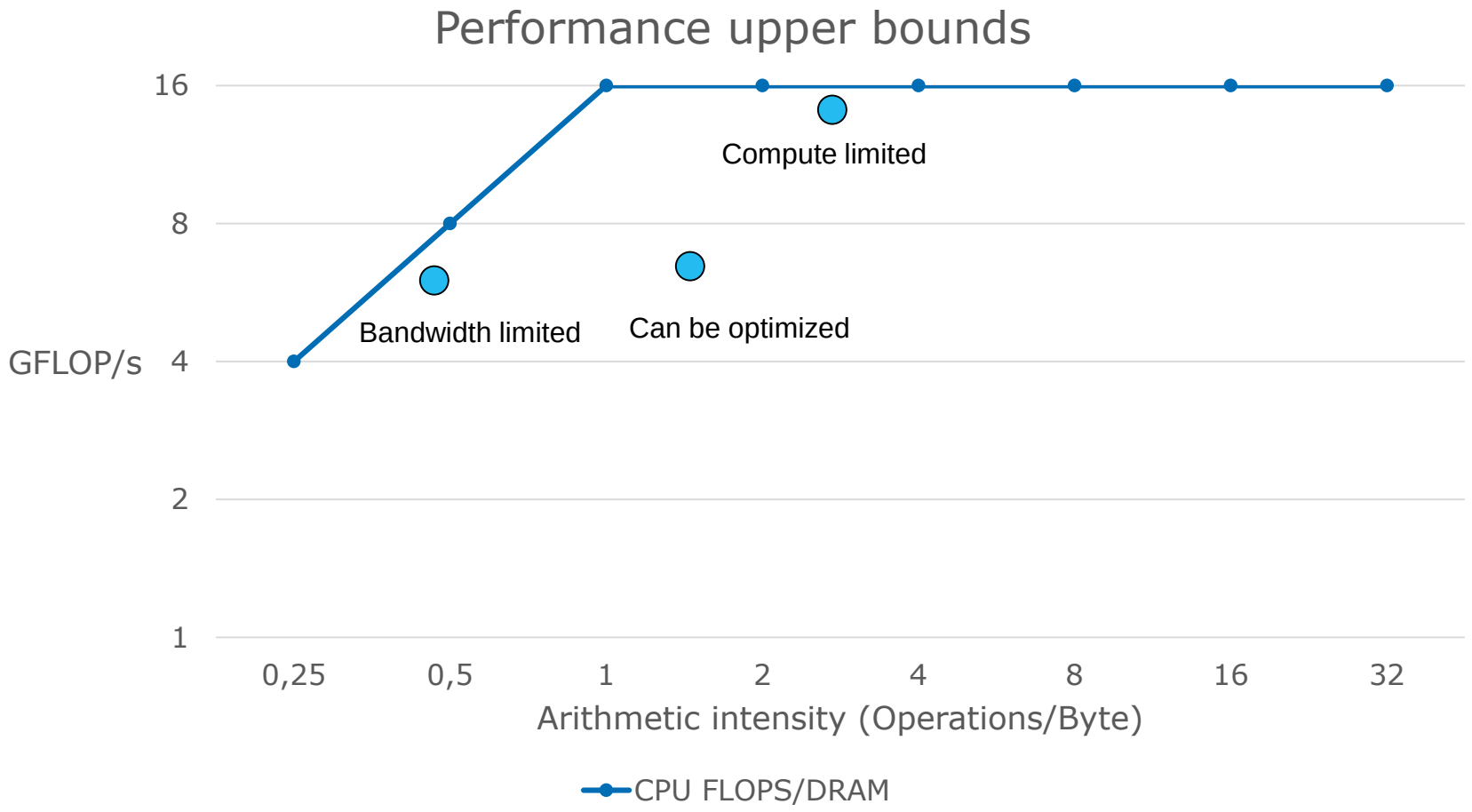
- More complex concepts
  - Roofline model (comparing throughput to theoretical maxima)
  - Load balancing: what code is responsible for idle resources?
  - Critical path analysis (e.g. Scalasca)

Strong and weak scaling

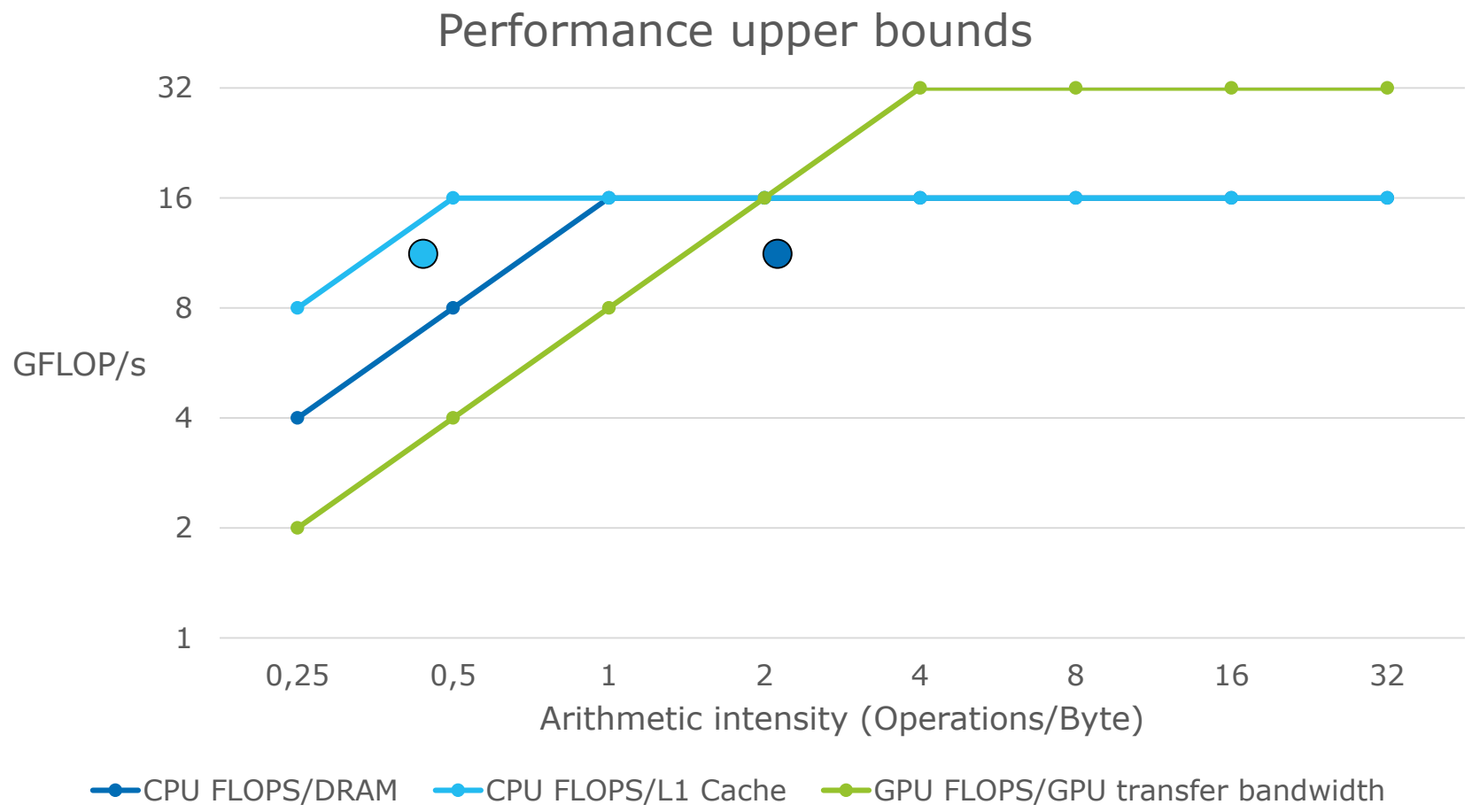
- Strong scaling: increasing compute power yields faster solutions on the same problem
  - Amdahl's law: Speedup = (serial + parallel) / (serial + parallel / N) = 1 / (serial + parallel / N)

- Weak scaling: increasing compute power yields larger problems solved in the same time
  - Gustafson's law: convert Amdahl's law to measure scaled speedup (as a factor of problem size)

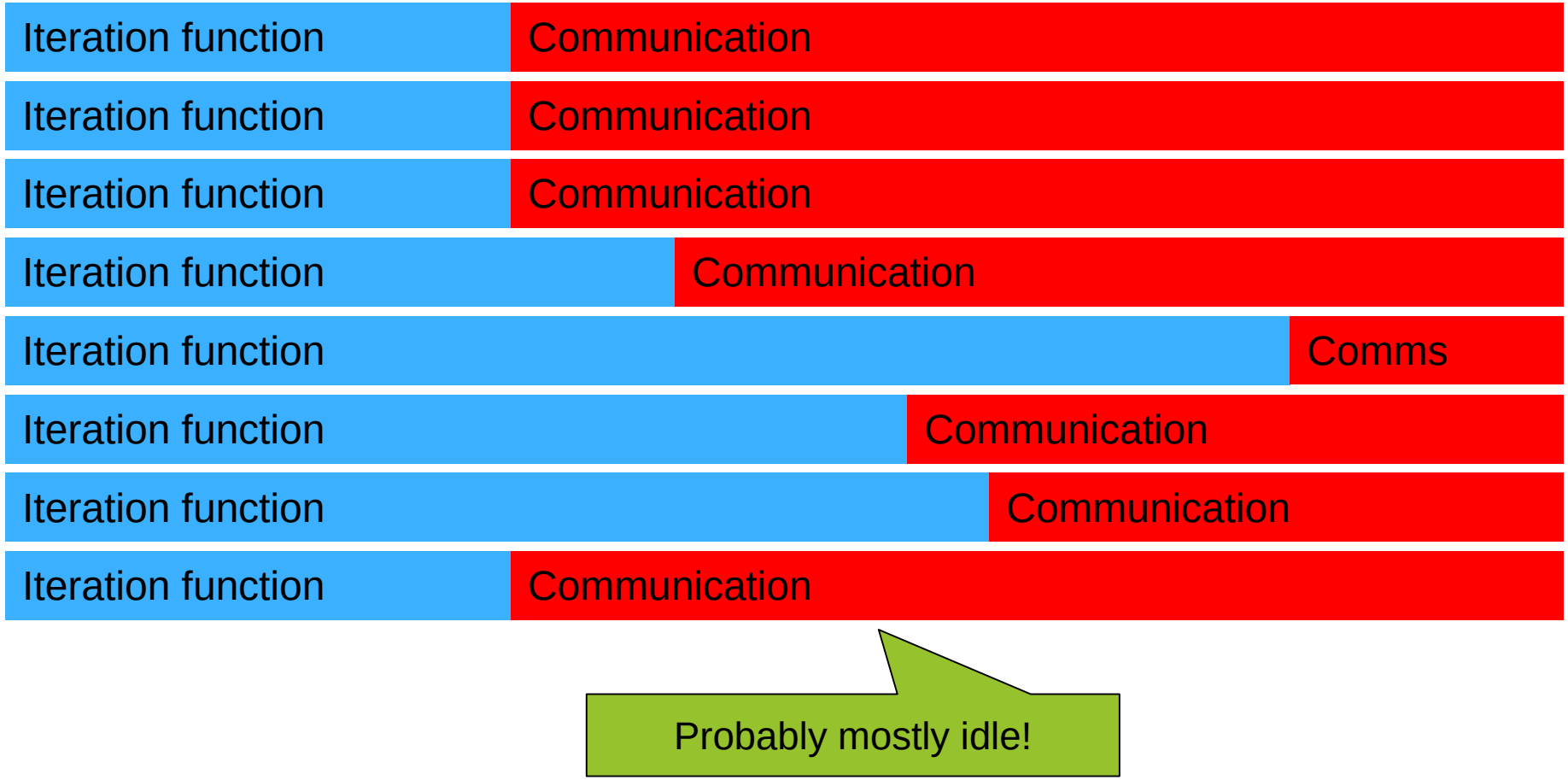
Roofline Model



Multiple Roofline Model



Load balancing



Critical Path Analysis

- Key concept: critical path is the sequence of tasks that govern execution time
  - At any given time, what is the job waiting for? May be computation or transfer or a combination!
- Optimizing tasks off the critical path can’t speed up execution
- Example: load imbalance due to a rare case being 2x slower than the common case
  - The slow rare case may only be 1% of aggregated execution time, but responsible for 50% of wall time in iterations
  - Optimize the critical path = make the rare case closer to the speed of the common case

What to Measure

- So you have some hypothesis about how your code will behave
- This requires certain data
  - Simple scaling models: execution time, possibly subdivided between serial and parallel parts
  - Roofline model: operations/second and bytes/second corresponding to one or more rooflines
  - Load balancing: distribution of time spent in computation and communication
  - Critical path: detailed measurement of execution time across all nodes and threads
- Allows you to ignore certain other data
  - Example: load balancing
  - Detection typically based on communication wait states
  - Don’t need to analyze computation details for that
- When possible, measure only what you need to test your hypothesis
  - All-in-one-run only when it’s unavoidable

Measurement Practices

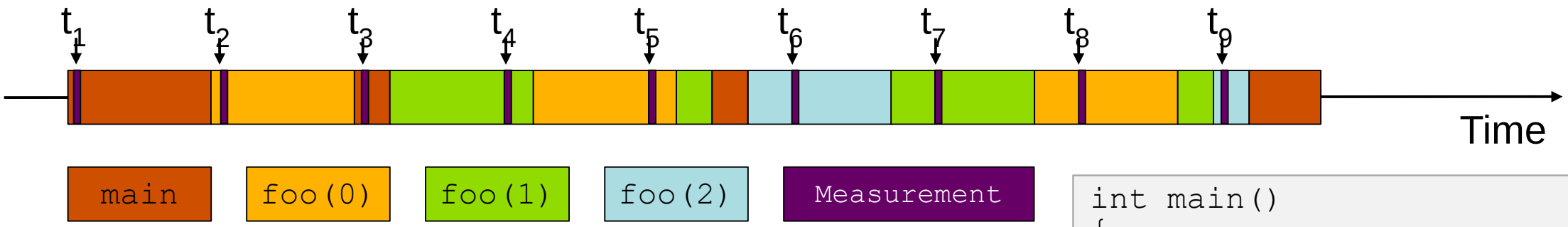
- Measurements on HPC systems are noisy
  - Shared resources: anything short of full-system DAT probably shares something (and maybe even then, if you use site-shared filesystems)
  - Nondeterminism: cache effects, which nodes were allocated, small race conditions
- Particularly relevant to wall time, but can affect other metrics
- As with all scientific measurements, repeat the experiment
  - Especially if the initial results look weird!

Measurement 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*

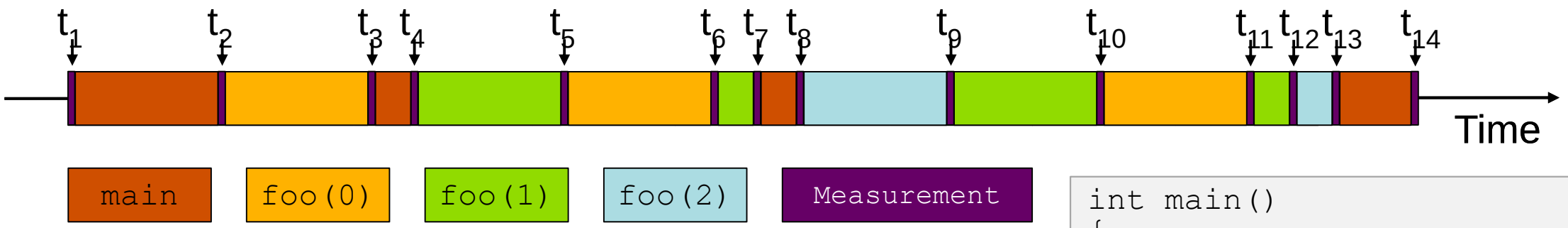
Sampling



- 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");
}
```

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*

Tracing

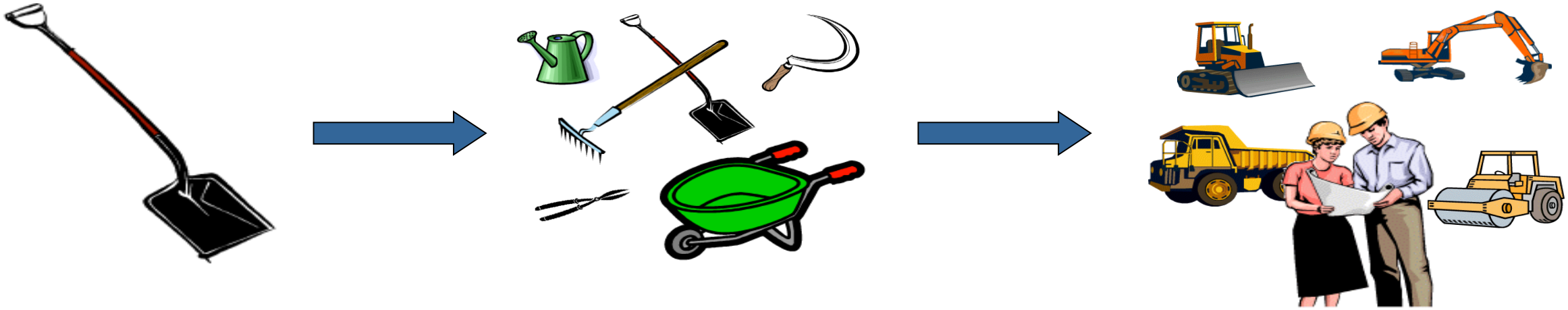
- Recording detailed information about significant points (events) during execution of the program
  - Enter / leave of a region (function, loop, …)
  - Send / receive a message, …
- Save information in event record
  - Timestamp, location, event type
  - Plus event-specific information (e.g., communicator,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

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

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

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

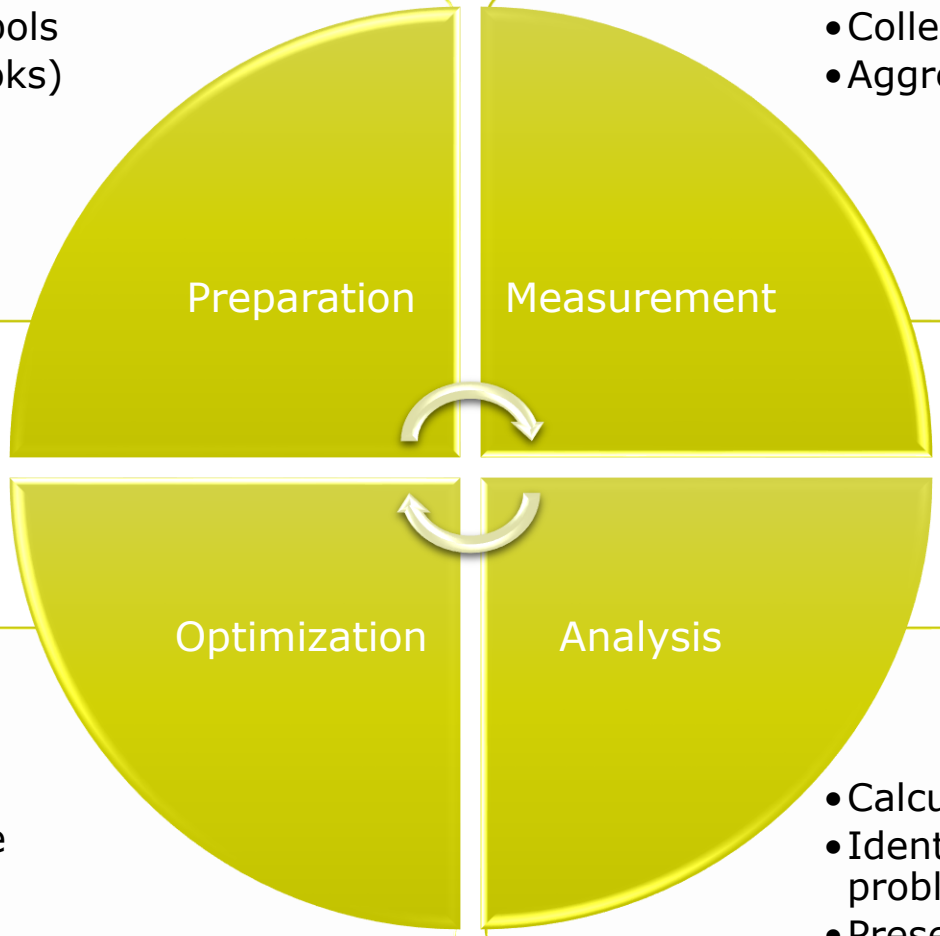
- Do I have a performance problem at all?
  - Time / speedup / scalability measurements
- What is the key bottleneck (computation / communication)?
  - MPI / OpenMP / flat profiling
- Where is the key bottleneck?
  - Call-path profiling, detailed basic block profiling
- Why is it there?
  - Hardware 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

Measurement with Score-P

Bill Williams, TU Dresden

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

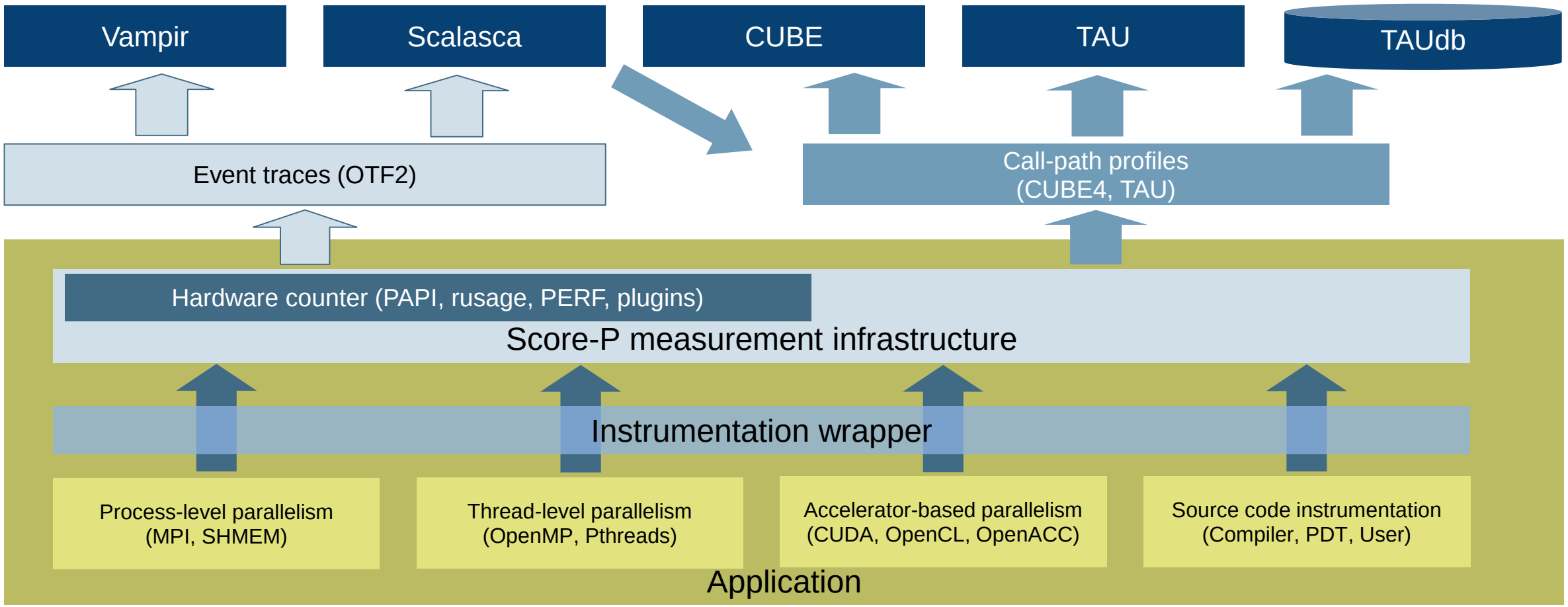
- Calculation of metrics
- Identification of performance problems
- Presentation of results

- Infrastructure for instrumentation and performance measurements
- Instrumented application can be used to produce several results:
  - Call-path profiling: CUBE4 data format used for data exchange
  - Event-based tracing: OTF2 data format used for data exchange
- Supported parallel paradigms:
  - Multi-process: MPI, SHMEM
  - Thread-parallel: OpenMP, Pthreads
  - Accelerator-based: CUDA, OpenCL

- Open Source; portable and scalable to all major HPC systems
- Initial project funded by BMBF
- Close collaboration with PRIMA project funded by DOE

GEFÖRDERT VOM
Bundesministerium für Bildung und Forschung

Score-P overview



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

Reference hands-on[1]: NPB-MZ-MPI / BT

Performance analysis steps

- Reference preparation for validation

- Program instrumentation
- Summary measurement collection

- Summary experiment scoring
- Trace measurement collection with filtering

NPB-MZ-MPI / BT suite

```
% cd ~/Tutorial
% ls
bin/
bin.scorep/
BT-MZ/
common/
config/
jobscript/
LU-MZ/
Makefile
README
README.install
README.tutorial
SP-MZ/
sys/
```

- The NAS Parallel Benchmark suite (MPI+OpenMP version)
  - http://www.nas.nasa.gov/Software/NPB
- Start in the *Tutorial* directory

NPB-MZ-MPI / BT build

```
% make bt-mz CLASS=W NPROCS=4
cd BT-MZ; make CLASS=W NPROCS=4 VERSION=
make: Entering directory 'BT-MZ'
cd ../sys; cc  -o setparams setparams.c -lm
../sys/setparams bt-mz 4 W
mpif77 -c  -O3 -fopenmp bt.f
 [...]
cd ../common;  mpif77 -c  -O3 -fopenmp timers.f
mpif77 -O3 -fopenmp -o ../bin/bt-mz_W.4 \
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/bt-mz_W.4
make: Leaving directory 'BT-MZ'
```

- Benchmark name:
  - **bt-mz**, lu-mz, sp-mz
- Number of MPI processes:
  - NPROCS=**4**
- Benchmark class:
  - S, **W**, A, B, C, D, E
  - CLASS=**W**

NPB-MZ-MPI / BT reference execution

```
% cd bin
% OMP_NUM_THREADS=4 mpiexec -np 4 ./bt-mz_W.4
 NAS Parallel Benchmarks (NPB3.3-MZ-MPI) - BT-MZ MPI+OpenMP \
>Benchmark

 Number of zones:   4 x   4
 Iterations: 200    dt:   0.000800
 Number of active processes:     4

 Use the default load factors with threads
 Total number of threads:     16   (  4.0 threads/process)

 Calculated speedup =     15.78

 Time step    1
 [... More application output ...]
 Time step  200
 [... More application output ...]
 BT-MZ Benchmark Completed.
 Time in seconds = 4.37
```

- Launch as a hybrid MPI+OpenMP application

Save the benchmark run time to be able to refer to it later. (Beware of potential over-subscription)

Performance analysis steps

- Reference preparation for validation

- Program instrumentation
- Summary measurement collection

- Summary experiment scoring
- Trace measurement collection with filtering

NPB-MZ-MPI / BT instrumentation

```
% cd ..
% make clean
```

- Start in the *Tutorial* directory again and clean-up the build

NPB-MZ-MPI / BT instrumentation

```
#            SITE- AND/OR PLATFORM-SPECIFIC DEFINITIONS
#-----------------------------------------------------------------
# Items in this file may need to be changed for each platform.
#-----------------------------------------------------------------
COMPFLAGS = -fopenmp
...
#-----------------------------------------------------------------
# The Fortran compiler used for MPI programs
#-----------------------------------------------------------------
#MPIF77 = mpif77

# Alternative variants to perform instrumentation
...
MPIF77 = scorep --user mpif77

# 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
  - Prefix compiler witch `scorep` command (or use compiler wrappers, see reference material)

NPB-MZ-MPI / BT instrumented build

```
% make bt-mz CLASS=W NPROCS=4
cd BT-MZ; make CLASS=W NPROCS=4 VERSION=
make: Entering directory 'BT-MZ'
cd ../sys; cc  -o setparams setparams.c -lm
../sys/setparams bt-mz 4 W
mpif77 -c  -O3 -fopenmp bt.f
 [...]
cd ../common;  scorep mpif77 -c  -O3 -fopenmp timers.f
scorep mpif77 -O3 -fopenmp -o ../bin.scorep/bt-mz_W.4 \
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_W.4
make: Leaving directory 'BT-MZ'
```

- Re-build executable prefixing the compiler with the `scorep` command

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 environment variables

NPB-MZ-MPI / BT summary measurement collection

```
% cd bin.scorep
% export SCOREP_TIMER=gettimeofday
% export SCOREP_EXPERIMENT_DIRECTORY=scorep_bt-mz_W_4x4_sum
% OMP_NUM_THREADS=4 mpiexec -np 4 ./bt-mz_W.4
 NAS Parallel Benchmarks (NPB3.3-MZ-MPI) - BT-MZ MPI+OpenMP \
>Benchmark

 Number of zones:   4 x   4
 Iterations: 200    dt:   0.000800
 Number of active processes:     4

 Use the default load factors with threads
 Total number of threads:     16  (  4.0 threads/process)

 Calculated speedup =     15.78

 Time step    1
 [... More application output ...]
 BT-MZ Benchmark Completed.
 Time in seconds = 11.09
```

- Change to the directory containing the new executable before running it with the desired configuration
- Run instrumented application

NPB-MZ-MPI / BT summary analysis report examination

```
% ls
bt-mz_W.4  scorep_bt-mz_W_4x4_sum
% ls scorep_bt-mz_W_4x4_sum/
MANIFEST.md  profile.cubex  scorep.cfg




% # optional
% cube scorep_bt-mz_W_4x4_sum/profile.cubex
% paraprof scorep_bt-mz_W_4x4_sum/profile.cubex

  [CUBE or TAU ParaProf GUI showing summary analysis report]
```

- Creates experiment directory including
  - Experiment directory overview (MANIFEST.md)
  - A record of the measurement configuration (scorep.cfg)
  - The analysis report that was collated after measurement (profile.cubex)

Congratulations!?

- If you made it this far, you successfully used Score-P to
  - instrument the application
  - record its execution with a summary measurement, and
  - [optional] 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

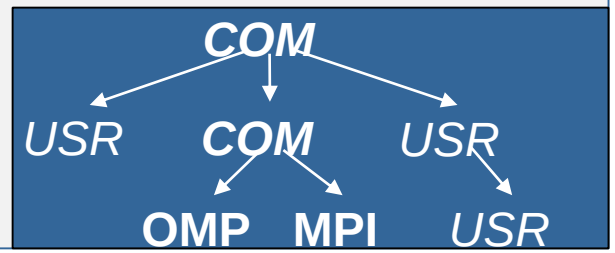
- Reference preparation for validation

- Program instrumentation
- Summary measurement collection

- Summary experiment scoring
- Trace measurement collection with filtering

NPB-MZ-MPI / BT summary analysis result scoring

```
% scorep-score scorep_bt-mz_W_4x4_sum/profile.cubex

Estimated aggregate size of event trace:                   1031MB
Estimated requirements for largest trace buffer (max_buf): 266MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       274MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=274MB to avoid intermediate fl
 or reduce requirements using USR regions filters.)

flt     type  max_buf[B]     visits time[s] time[%] time/visit[us]  region
         ALL 278,581,239 41,158,333 1284.51   100.0          31.21  ALL
         USR 274,792,492 40,418,321  286.86    22.3           7.10  USR
         OMP   8,768,540    685,952  862.00    67.1        1256.64  OMP
         COM     377,130     46,740  112.21     8.7        2442.29  COM
         MPI     124,120      7,316   23.44     1.8        3204.09  MPI
      SCOREP          41          4    0.00     0.0          60.75  SCOREP
```



1 GB total memory
265 MB per rank!

Region/callpath classification

- **MPI** pure MPI functions
- **OMP** pure OpenMP regions
- **USR** user-level computation
- **COM** "combined" USR+OpenMP/MPI
- **SCOREP** measurement internals
- **ANY/ALL** aggregate of all region types

NPB-MZ-MPI / BT summary analysis report breakdown

```
% scorep-score -r scorep_bt-mz_W_4x4_sum/profile.cubex
  [...]
  [...]
flt     type  max_buf[B]     visits time[s] time[%] time/visit[us]  region
         ALL 278,581,239 41,158,333 1284.51   100.0          31.21  ALL
         USR 274,792,492 40,418,321  286.86    22.3           7.10  USR
         OMP   8,768,540    685,952  862.00    67.1        1256.64  OMP
         COM     377,130     46,740  112.21     8.7        2442.29  COM
         MPI     124,120      7,316   23.44     1.8        3204.09  MPI
      SCOREP          41          4    0.00     0.0          60.75  SCOREP

         USR  85,774,338 12,516,672   88.69     6.9           7.09  matmul_sub
         USR  85,774,338 12,516,672   91.14     7.1           7.28  binvcrhs
         USR  85,774,338 12,516,672   86.03     6.7           6.87  matvec_sub
         USR   7,974,876  1,170,624    7.58     0.6           6.48  lhsinit
         USR   7,974,876  1,170,624    7.76     0.6           6.63  binvrhs
         USR   3,473,912    526,848    5.65     0.4          10.73  exact_solution
 [...]
```

More than 270 MB just for these 6 regions

NPB-MZ-MPI / BT summary analysis score

- Summary measurement analysis score reveals
  - Total size of event trace would be ~1025 MB
  - Maximum trace buffer size would be ~265 MB per rank
    - smaller buffer would require flushes to disk during measurement resulting in substantial perturbation
  - 99.8% 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 22% 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 (compile-time) filter file listing (USR) regions not to be measured
  - Alternatively, replace compiler instrumentation with sampling, see reference material

NPB-MZ-MPI / BT summary analysis report filtering

```
% scorep-score -g scorep-bt-mz_W_4x4_profile/profile.cubex
An initial filter file template has been generated:
'initial_scorep.filter'
To use this file for filtering at run-time, set the respective
Score-P variable:
    SCOREP_FILTERING_FILE=initial_scorep.filter
For compile-time filtering 'scorep' has to be provided with
the '--instrument-filter' option:
    $ scorep --instrument-filter=initial_scorep.filter
Compile-time filtering depends on support in the used Score-P
installation.
The filter file is annotated with comments, please check if
the selection is suitable for your purposes and add or remove
functions if needed.
```

- Report scoring with prospective filter listing 6 USR regions

NPB-MZ-MPI / BT 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 \
>scorep_bt-mz_W_4x4_sum/profile.cubex

Estimated aggregate size of event trace:                   29MB
Estimated requirements for largest trace buffer (max_buf): 9MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       17MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=17MB to avoid intermediate flushes
 or reduce requirements using USR regions filters.)
```

- Report scoring with prospective filter listing 6 USR regions

29 MB of memory in total, 9 MB per rank!

NPB-MZ-MPI / BT summary analysis report filtering

```
% scorep-score -r -f ../config/scorep.filt \
> scorep__bt-mz_W_4x4_sum/profile.cubex
flt     type  max_buf[B]     visits time[s] time[%] time/visit[us]  region
-        ALL 278,581,239 41,157,533 1284.51   100.0          31.21  ALL
-        USR 274,792,492 40,418,321  286.86    22.3           7.10  USR
-        OMP   8,768,540    685,952  862.00    67.1        1256.64  OMP
-        COM     377,130     46,740  112.21     8.7        2442.29  COM
-        MPI     124,120      7,316   23.44     1.8        3204.09  MPI
-     SCOREP          41          4    0.00     0.0          60.75  SCOREP


*        ALL   7,357,804    739,321 1284.51   100.0          31.21  ALL-FLT
+        FLT 274,791,764 40,418,212  286.86    22.3           7.10  FLT
-        OMP   6,882,860    685,952  862.00    67.1        1256.64  OMP-FLT
*        COM     377,130     46,740  112.21     8.7        2442.29  COM-FLT
-        MPI     124,120      7,316   23.44     1.8        3204.09  MPI-FLT
*        USR         728        109    0.00     0.0           1.22  USR-FLT
-     SCOREP          41          4    0.00     0.0          60.75  SCOREP-FLT

[...]
```

Filtered routines marked with '+'

- Score report breakdown by region

NPB-MZ-MPI / BT filtered trace measurement collection

```
% export SCOREP_FILTERING_FILE=../config/scorep.filt
% export SCOREP_TOTAL_MEMORY=17M
% export SCOREP_EXPERIMENT_DIRECTORY= \
scorep_bt-mz_W_4x4_trace

# 1. Score-P way: generates OTF2 traces to be used with
#    Vampir and Scalasca
% export SCOREP_ENABLE_TRACING=true
% OMP_NUM_THREADS=4 mpiexec -np 4 ./bt-mz_W.4



# 2. Scalasca way: generates OTF2 traces AND Scalasca
#    trace analysis on top
% ...
% OMP_NUM_THREADS=4 scan -t mpiexec -np 4 ./bt-mz_W.4
```

- Apply filter configuration and re-run measurement
- Two options to generate traces
- Filtered summary experiment also possible

NPB-MZ-MPI / BT filtered trace measurement collection

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

 Number of zones:   4 x   4
 Iterations: 200    dt:   0.000800
 Number of active processes:     4

 Use the default load factors with threads
 Total number of threads:     16  (  4.0 threads/process)

 Calculated speedup =     15.78

 Time step    1
 [... More application output ...]
 BT-MZ Benchmark Completed.
 Time in seconds = 4.81
```

- Output from filtered run

NPB-MZ-MPI / BT filtered trace examination

```
% ls
bt-mz_W.4  scorep_bt-mz_W_4x4_sum/  scorep_bt-mz_W_4x4_trace/

% ls scorep_bt-mz_W_4x4_trace/
MANIFEST.md    scorep.filter  scout.log   traces.otf2
profile.cubex  scorep.log     traces/     trace.stat
scorep.cfg     scout.cubex    traces.def


# Trace examination with Vampir
% vampir scorep_bt-mz_W_4x4_trace/traces.otf2

 [Vampir GUI showing trace]

# Trace-analysis examination with Scalasca/Cube:
# Creates additional trace.cubex file by post-processing
# scout.cubex
% square scorep_bt-mz_W_4x4_trace/

 [CUBE GUI showing trace analysis report]
```

- Trace collection generates `traces.otf2`
- Trace analysis generates `scout.cubex`
- Use *Vampir* to examine trace

Function Groups

- Frequently asked questions:
  - How do I structure my code to make it tools-comprehensible?
  - What does Score-P do automatically to make my measurement easier to read?
- Extend the USR/COM/MPI/… concept from scoring: *function groups*
- Predefined groupings:
  - Per paradigm
  - Within paradigms (MPI categories)
  - Namespace/class hierarchy

Score-P: Further information

- Scalable Performance Measurement Infrastructure for Parallel Codes
  - Instrumenter, libraries, and tools to generate profile and trace measurements
  - Bundled with OTF2 (tracing), OPARI2 (OpenMP instrumentation), CubeWriter, and CubeLib (profiling)
- 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/scorep.pdf
- Contact:
  - mailto: support@score-p.org

Score-P:
Specialized Measurements and Analyses

Bill Williams

Score-P filtering: Automatic Generation of Filter Files

- **Basic usage**: `scorep-score -g` default heuristic targets:
  - Buffer usage: relevancy
  - Time per visits: overhead
- Creates annotated filter file:
  - `initial_scorep.filter`
  - Repeated calls create backups
  - Usage with `-f <file>` results in inclusion
- **Objective**:
  - Starting point for filtering
  - Syntax introduction

```
-g [<list>]
Generation of an initial filter file with the name
'initial_scorep.filter'. A valid parameter list has the form
KEY=VALUE[,KEY=VALUE]*. By default, uses the following control
parameters:

    `bufferpercent=1,timepervisit=1`

A region is included in the filter file (i.e., excluded from
measurement) if it matches all of the given conditions, with the
following keys:
- `bufferpercent`       : estimated memory requirements exceed the
                          given threshold in percent of the total
                          estimated trace buffer requirements
- `bufferabsolute`      : estimated memory requirements exceed
                          the given absolute threshold in MB
- `visits`              : number of visits exceeds the given
                          threshold
- `timepervisit`        : time per visit value is below the given
                          threshold in microseconds
- `type`                : region type matches the given value
                          (allowed: 'usr', 'com', 'both')
```

Score-P filtering: Automatic Generation of Filter Files

```
. . .
#
# Generated with the following parameters:
#  - A region has to use at least 1% of the estimated trace buffer.
#  - A region has to have a time/visits value of less than 1 us.
#  - A region that is of type USR.
#
# The file contains comments for each region providing additional information
# regarding the respective region.
# The common prefix for the files is:
# '/data/work/sc21_demo/BT-MZ/'
#
# Please refer to the Score-P user guide for more options on filtering.
SCOREP_REGION_NAMES_BEGIN
  EXCLUDE
    # type=USR max_buf=165,340,188 visits=12,516,672, time=  10.96s ( 17.1%), time/visit=    0.88us
    # name='binvcrhs'
    # file='BT-MZ/solve_subs.f'
    MANGLED binvcrhs_

. . .

SCOREP_REGION_NAMES_END
```

Introductory preamble

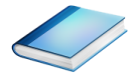
Used generation parameters (either default or explicit)

Common path prefix to simplify file paths in entries

Annotated entry with comments containing `scorep-score` stats

Additional entries

Score-P filtering



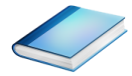
```
% cat ../config/scorep.filt
SCOREP_REGION_NAMES_BEGIN
  EXCLUDE
    binvcrhs*
    matmul_sub*
    matvec_sub*
    exact_solution*
    binvrhs*
    lhs*init*
    timer_*
SCOREP_REGION_NAMES_END

% 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, Intel compiler)
    - Add cmd-line option **`--instrument-filter`**
    - No overhead for filtered regions but recompilation

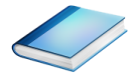
Source file name filter block



- 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
```

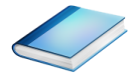
Region name filter block



- 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

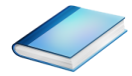


- Name mangling
  - Filtering based on names seen by the measurement system
    - Dependent on compiler
    - Actual name may be mangled
- `scorep-score` names as starting point (e.g. `matvec_sub_`)
  - Use `*` for Fortran trailing underscore(s) for portability
  - Use `?` and `*` as needed for full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

Mastering application memory usage



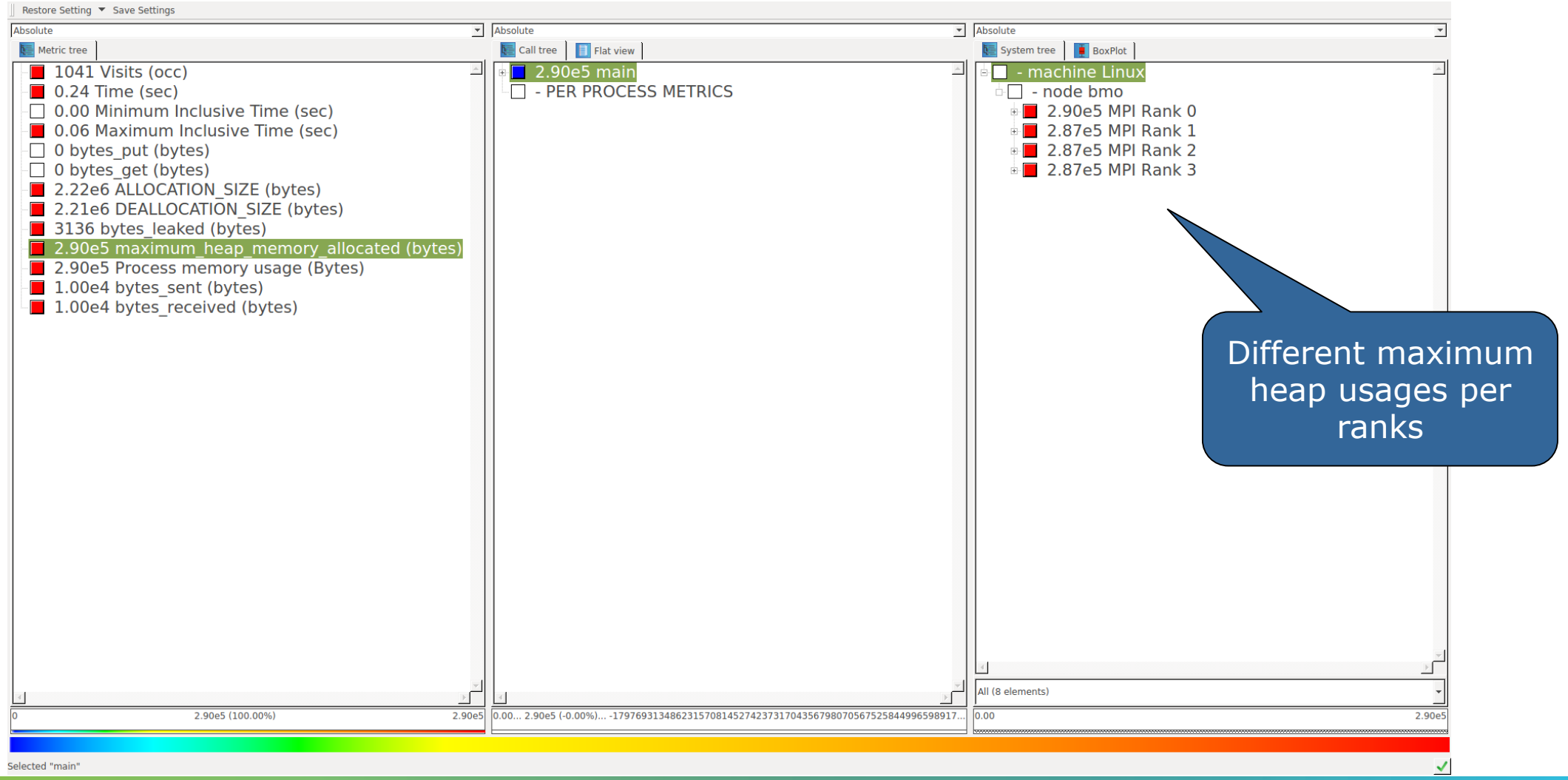
- Determine the maximum heap usage per process
- Find high frequent small allocation patterns
- Find memory leaks
- Support for:
  - C, C++, MPI, and SHMEM (Fortran only for GNU Compilers)
  - Profile and trace generation (profile recommended)
    - Memory leaks are recorded only in the profile
    - 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
```

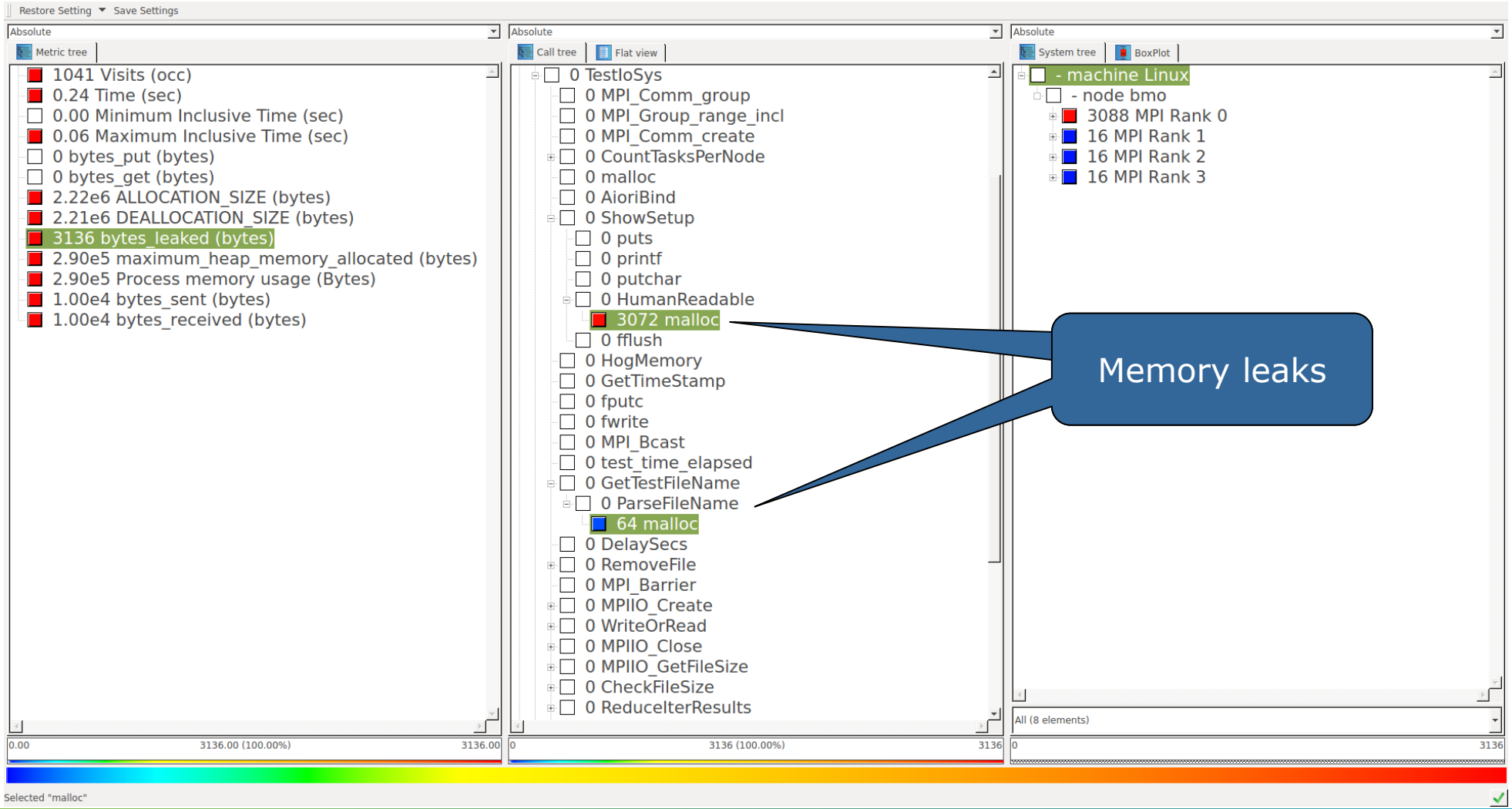
- Set new configuration variable to enable memory recording

- Available since Score-P 2.0

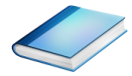
Mastering application memory usage



Mastering application memory usage

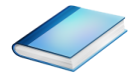


Score-P user instrumentation API



- Not a 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`

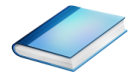
Score-P user instrumentation API (C/C++)

```
#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

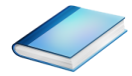
```
#include "scorep/SCOREP_User.inc"

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++

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
```

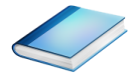
  - Use the `perf` tool to get available metrics and valid combinations:

```
% 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)

Mastering build systems



- Hooking up the Score-P instrumenter `scorep` into complex build environments like *Autotools* or *CMake* was always challenging
- Score-P provides 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*

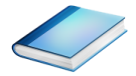
```
% SCOREP_WRAPPER=off \
> cmake .. \
> -DCMAKE_C_COMPILER=scorep-icc \
> -DCMAKE_CXX_COMPILER=scorep-icpc
```

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

Hybrid measurement with sampling



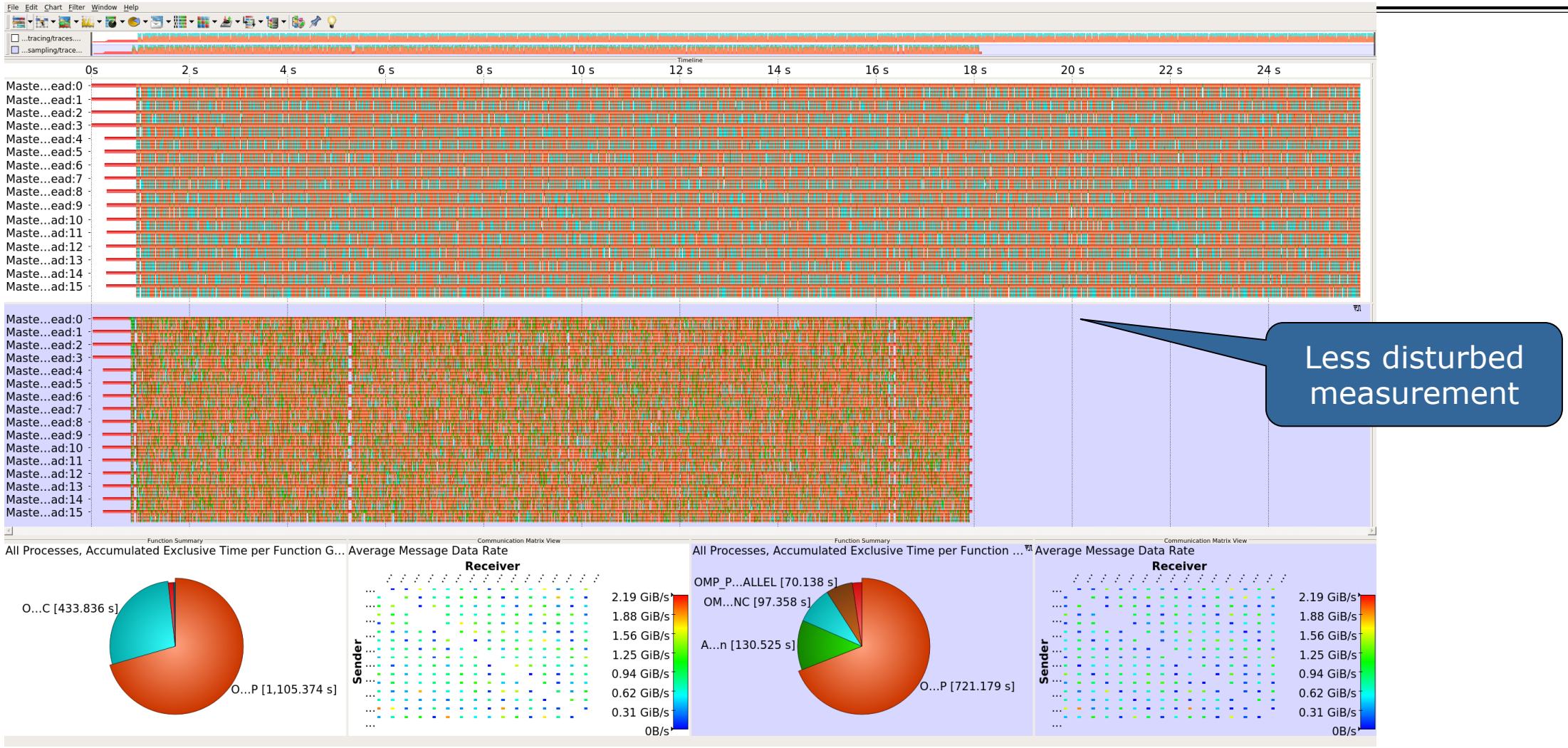
- Automatic compiler instrumentation greatly disturbs C++ applications because of frequent/short function calls => Use sampling instead
- Novel combination of sampling events and instrumentation of MPI, OpenMP, …
  - Sampling replaces compiler instrumentation (use `-nocompiler`)
  - Instrumentation is used for parallel activities
- 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

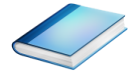
- Available since Score-P 2.0, only x86-64 supported currently

Mastering C++ applications



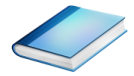
Advanced Score-P topics: Wrapping calls to 3rd party libraries

Wrapping calls to 3rd party libraries

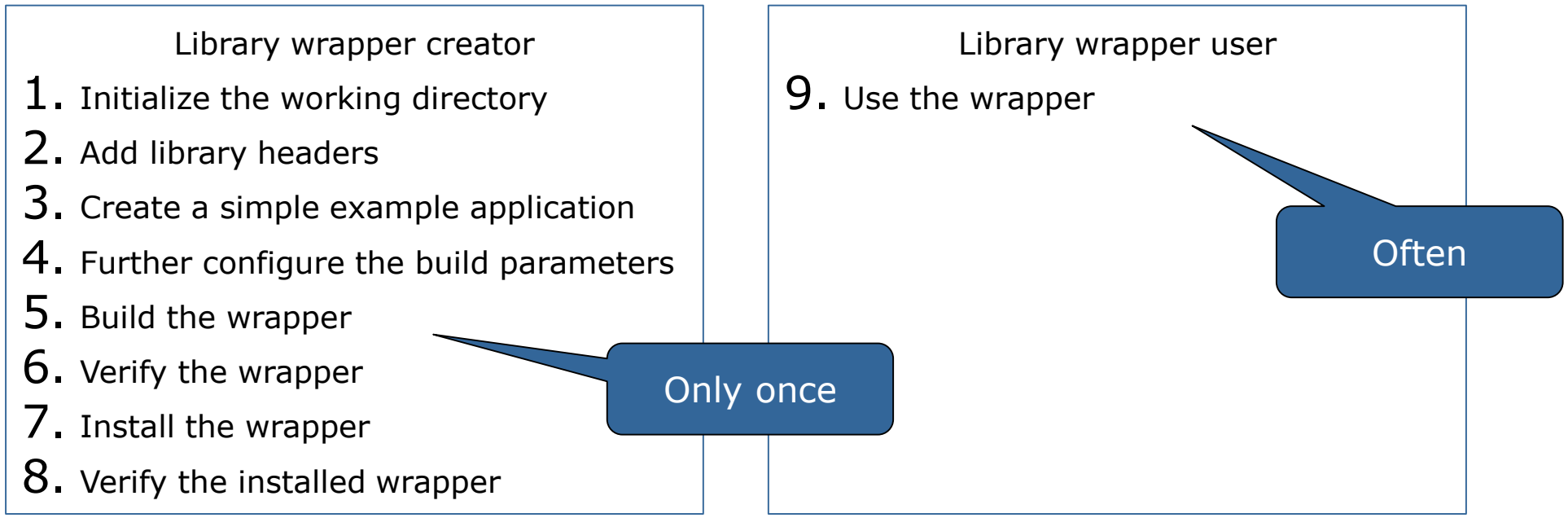


- Score-P does not record function calls to non-instrumented external libraries
- Increase insight into the behavior of the application
  - How does the application use the external library?
  - How does this compares to the usage of other libraries?
- Manual user instrumentation of the application using the library should be avoided
- Vendor provided libraries cannot be instrumented, but API provided in headers

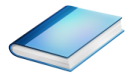
Wrapping calls to 3rd party libraries: Library wrapper generator



- Workflow to generate library wrappers for most C/C++ library
- Tailored towards user of the external library, not users of Score-P
- Results can be shared by multiple users
- Workflow driver `scorep-libwrap-init --help` provides instructions



Wrapping calls to 3rd party libraries: Workflow



- Start workflow 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
```

Omit to install into Score-P

Flags used to compile/link

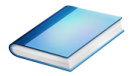
Working directory can be archived for later rebuild

- 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.
```

Tells you how to use the wrapper with Score-P

Wrapping calls to 3rd party libraries: Usage and result



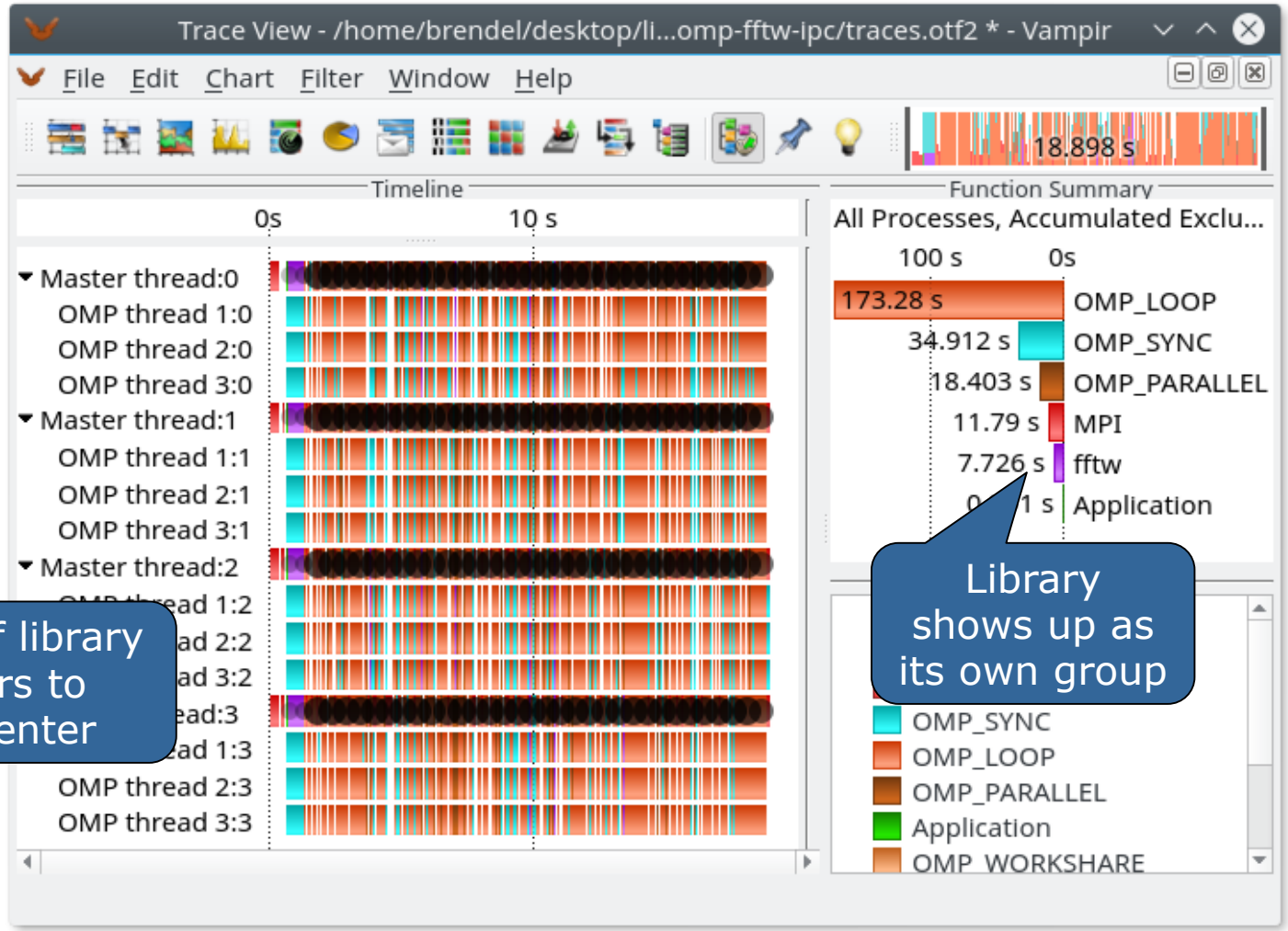
- List of available wrappers:

```
% scorep-info libwrap-summary
```

- Instrumentation:

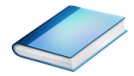
```
% cd <application>
% export \
SCOREP_WRAPPER_INSTRUMENTER_FLAGS=\
    --libwrap=fftw
% make clean
% make
# run application as usual
```

- MPI + OpenMP
- Calls to FFTW library



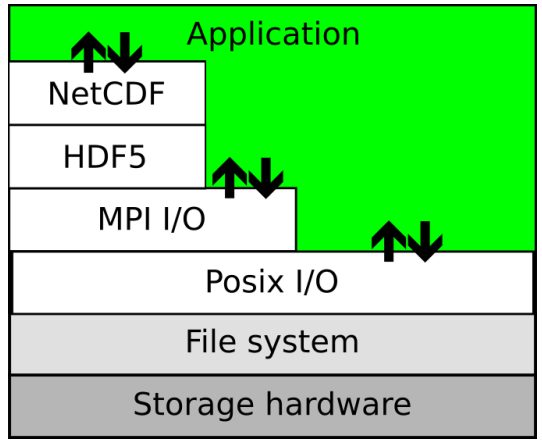
Specialized Score-P Measurements and Analyses: Multi-layered I/O recording

File I/O recording



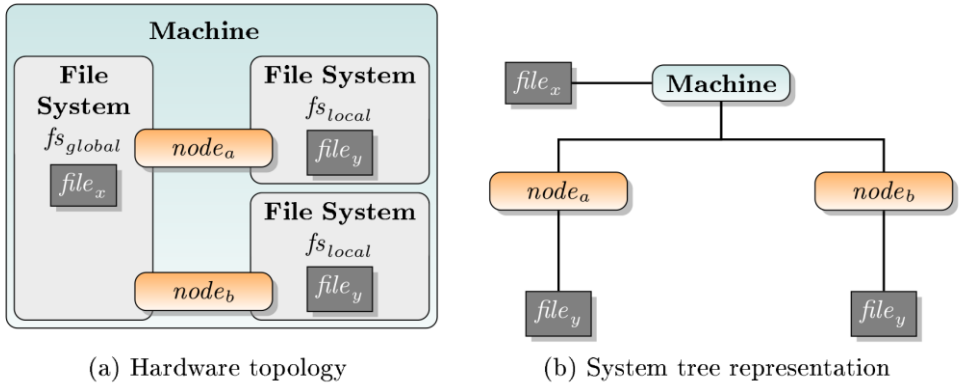
- Omnipresent in todays HPC applications

- Record interaction between multiple layers
  - MPI I/O (`MPI_File_open`)
  - ISO C I/O (`fopen`)
  - POSIX I/O (`open`, interface to OS)



- System tree information determine whether file resides in a shared filesystem

- High level of detail
  => Trace data might increase dramatically



(a) Hardware topology (b) System tree representation

NetCDF & HDF5 will be supported later

B_EFF I/O

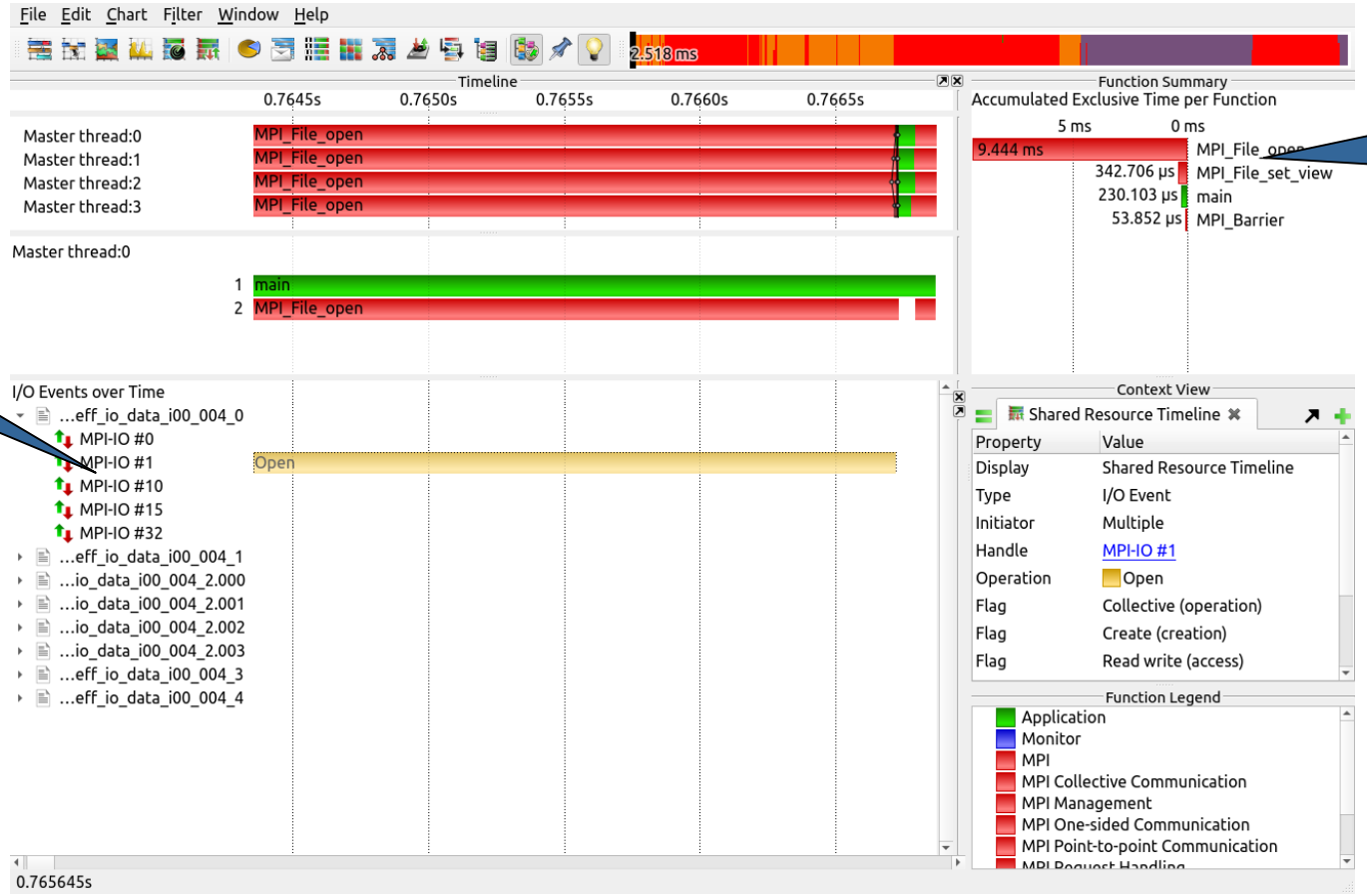
- MPI I/O benchmark
- fs.hlrs.de/projects/par/mpi/b_eff_io/

- MPI I/O is enabled by default

```
% scorep-mpicc -o b_eff_io b_eff_io.c
% export SCOREP_EXPERIMENT_DIRECTORY=scorep-b_eff_io-4-profile
% mpirun -n 4 -c 6 ./b_eff_io -MB 2048 -MT 98304 -rewrite -N 4 -T 60
% scorep-scorep –g scorep-b_eff_io-4-profile/profile.cubex
% export SCOREP_EXPERIMENT_DIRECTORY=scorep-b_eff_io-4-tracing
% export SCOREP_FILTERING_FILE=initial_scorep.filter
% export SCOREP_ENABLE_TRACING=true
% export SCOREP_TOTAL_MEMORY=31MB
% mpirun -n 4 -c 6 ./b_eff_io -MB 2048 -MT 98304 -rewrite -N 4 -T 60
```

Result visualization

Hierarchy of files and corresponding handles



MPI I/O functions recorded

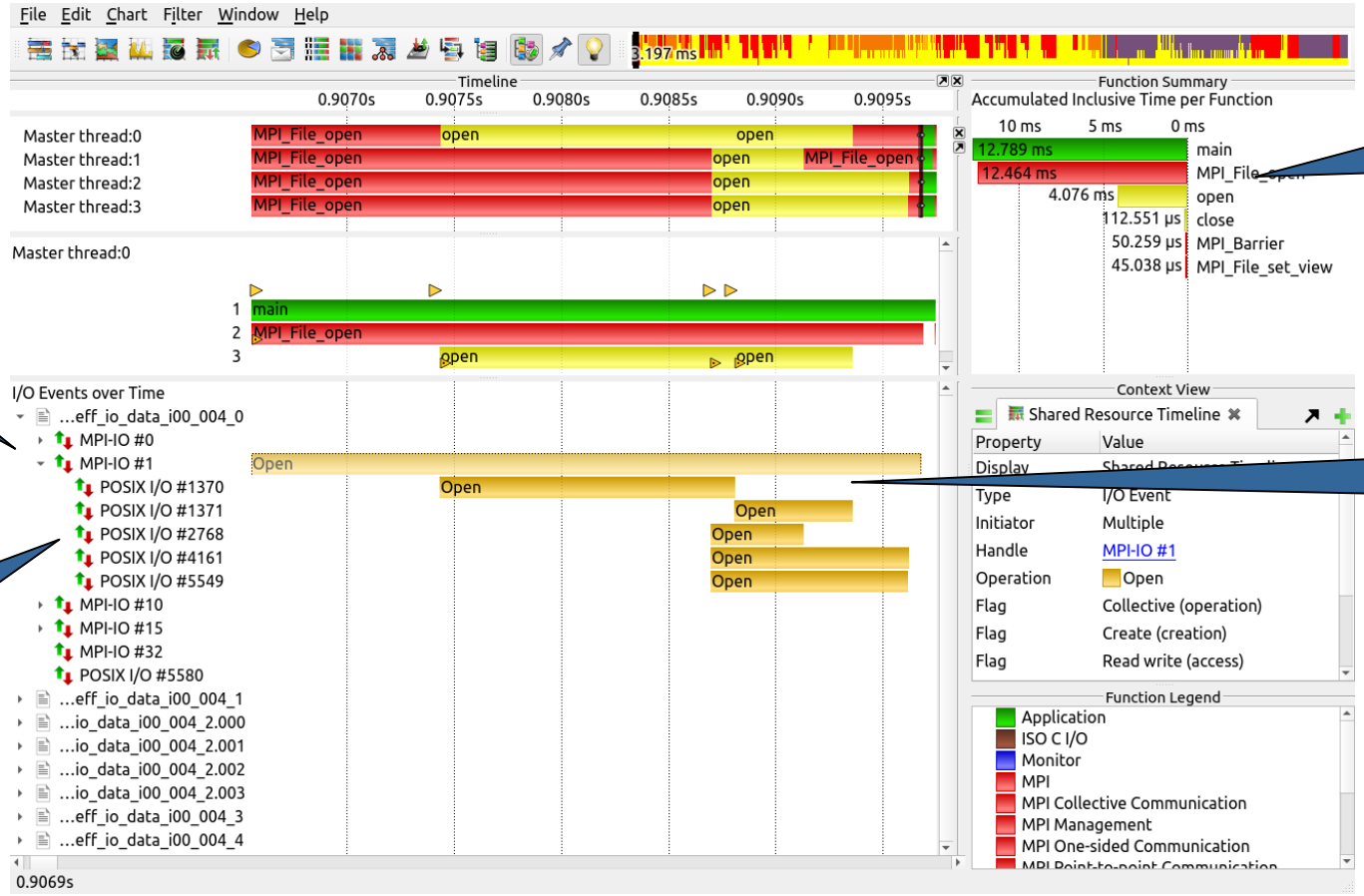
B_EFF I/O

- Enabling ISO C and POSIX I/O instrumentation

- Instrumentation might require threading support

```
% export SCOREP_WRAPPER_INSTRUMENTER_FLAGS=\
    '--io=runtime:posix --thread=pthread'
% scorep-mpicc -o b_eff_io b_eff_io.c
% export SCOREP_EXPERIMENT_DIRECTORY=scorep-b_eff_io-4-profile+posix
% export SCOREP_ENABLE_TRACING=false
% scorep-scorep scorep-b_eff_io-4-profile+posix/profile.cubex
% export SCOREP_EXPERIMENT_DIRECTORY=scorep-b_eff_io-4-tracing+posix
% export SCOREP_ENABLE_TRACING=true
% export SCOREP_TOTAL_MEMORY=61MB
% mpirun -n 4 -c 6 ./b_eff_io -MB 2048 -MT 98304 -rewrite -N 4 -T 60
```

Result visualization



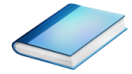
Increased hierarchy

Each rank operates on its own POSIX handle

MPI and POSIX I/O functions recorded

Rank 0 ensures file exists for all ranks

Mastering accelerators



- 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
```

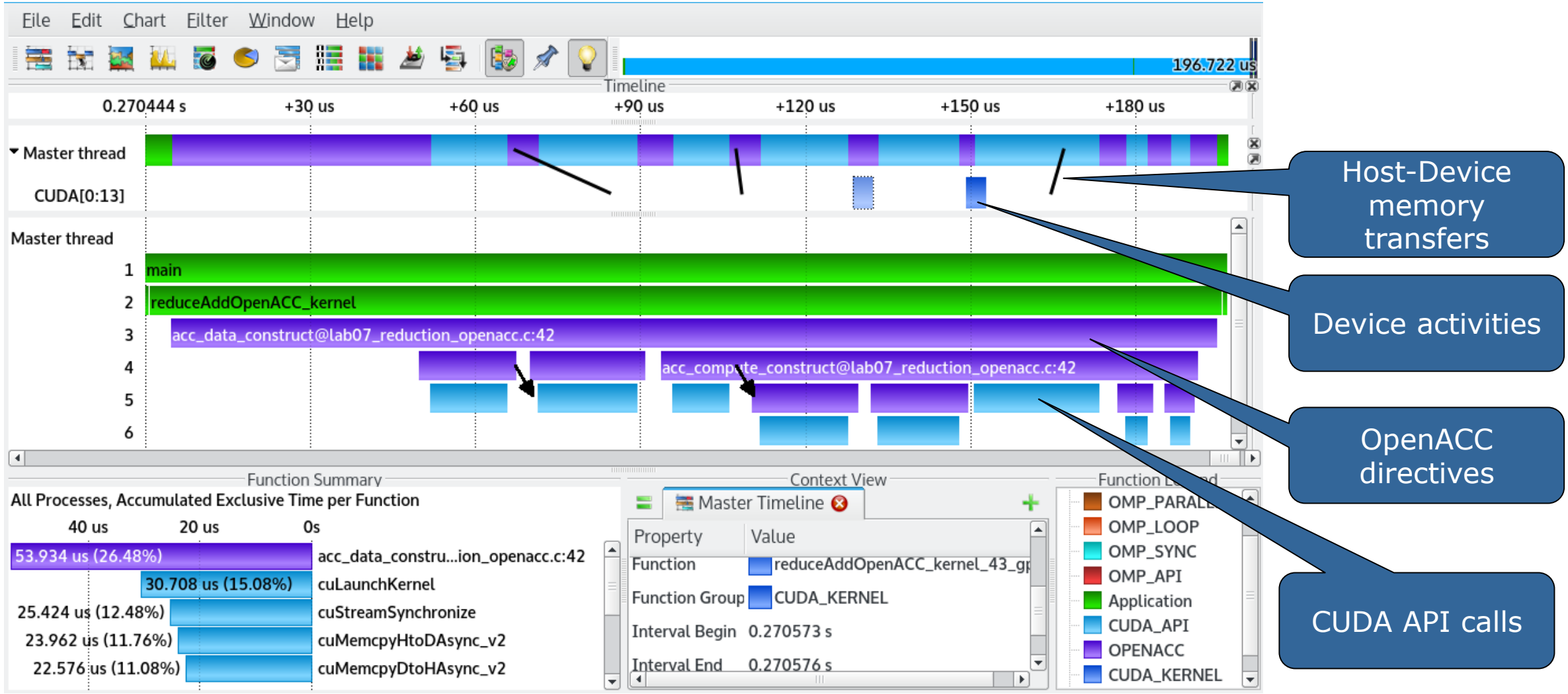
- Record OpenACC applications

```
% export SCOREP_OPENACC_ENABLE=yes
```

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

```
% export SCOREP_CUDA_ENABLE=kernel
```

Mastering accelerators



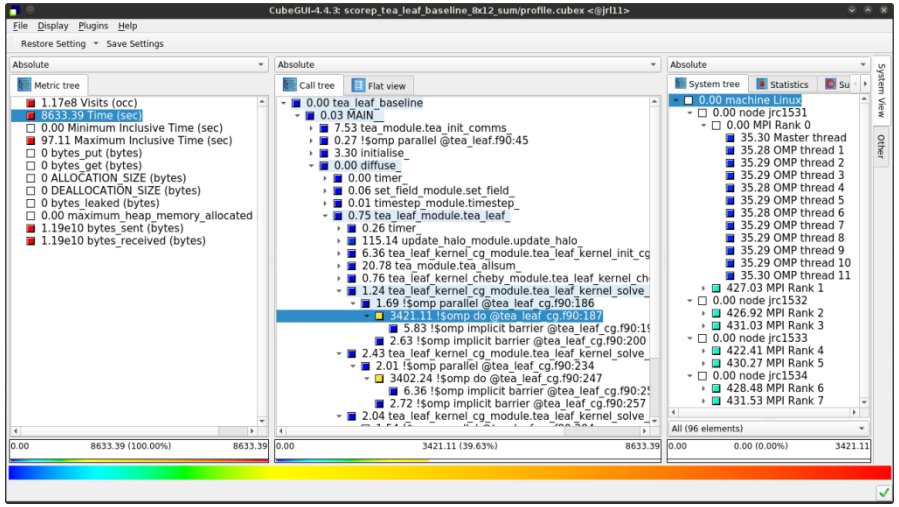
Analysis report examination with Cube

Markus Geimer
Jülich Supercomputing Centre

Cube

CubeLib DOI 10.5281/zenodo.6451943
CubeGUI DOI 10.5281/zenodo.6451949

- Parallel program analysis report exploration tools
  - Libraries for XML+binary report reading & writing
  - Algebra utilities for report processing
  - GUI for interactive analysis exploration
    - Requires Qt ≥4.8.3 (including Qt 5/6)



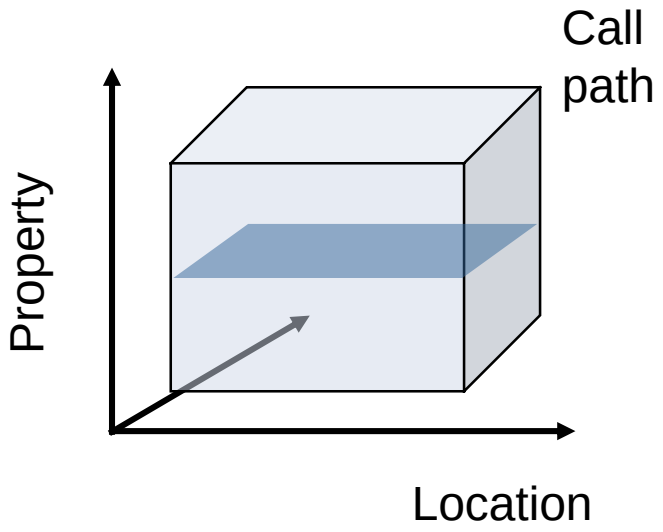
- Originally developed as part of the Scalasca toolset

- Now available as a separate components
  - Can be installed independently of Score-P, e.g., on laptop or desktop
  - Latest release: Cube v4.7 (April 2022)

Note: source distribution tarballs for Linux, as well as binary packages provided for Windows & MacOS, from **www.scalasca.org** website in software/Cube-4x

Analysis presentation and exploration

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

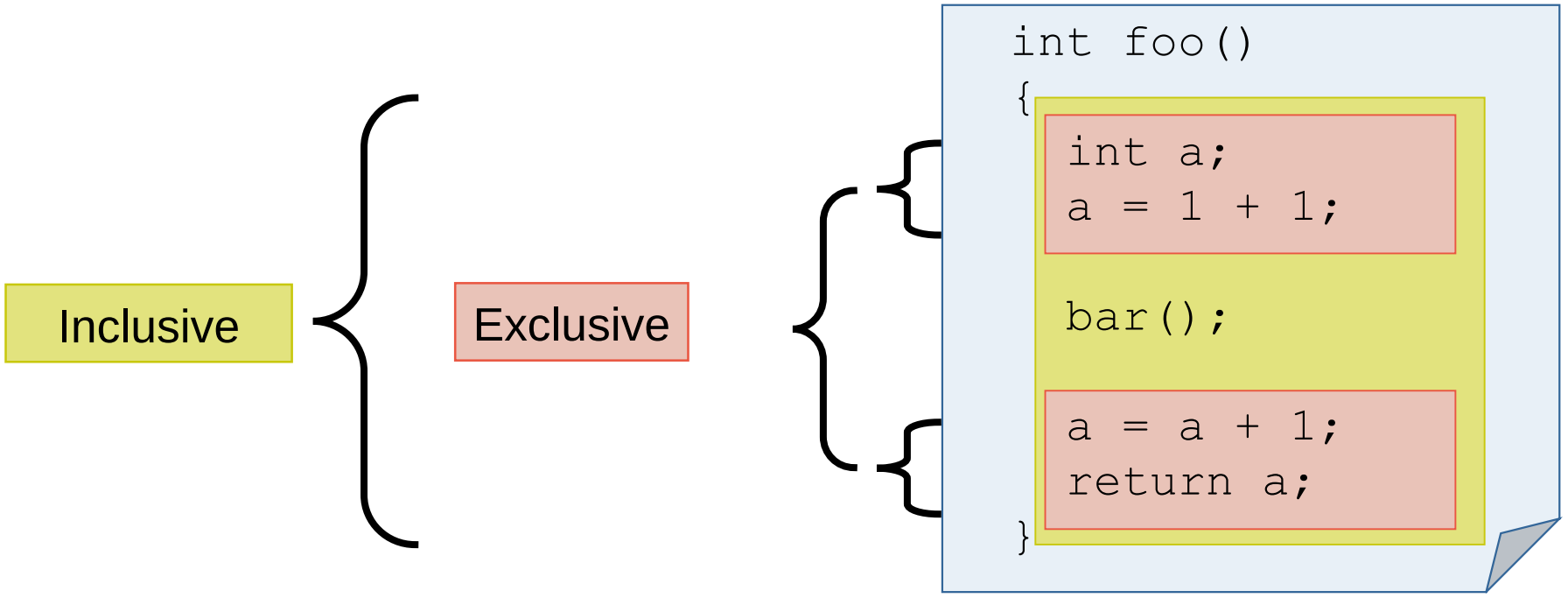


- Three coupled tree browsers

- 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*

Inclusive vs. exclusive values

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



Demo:
TeaLeaf case study

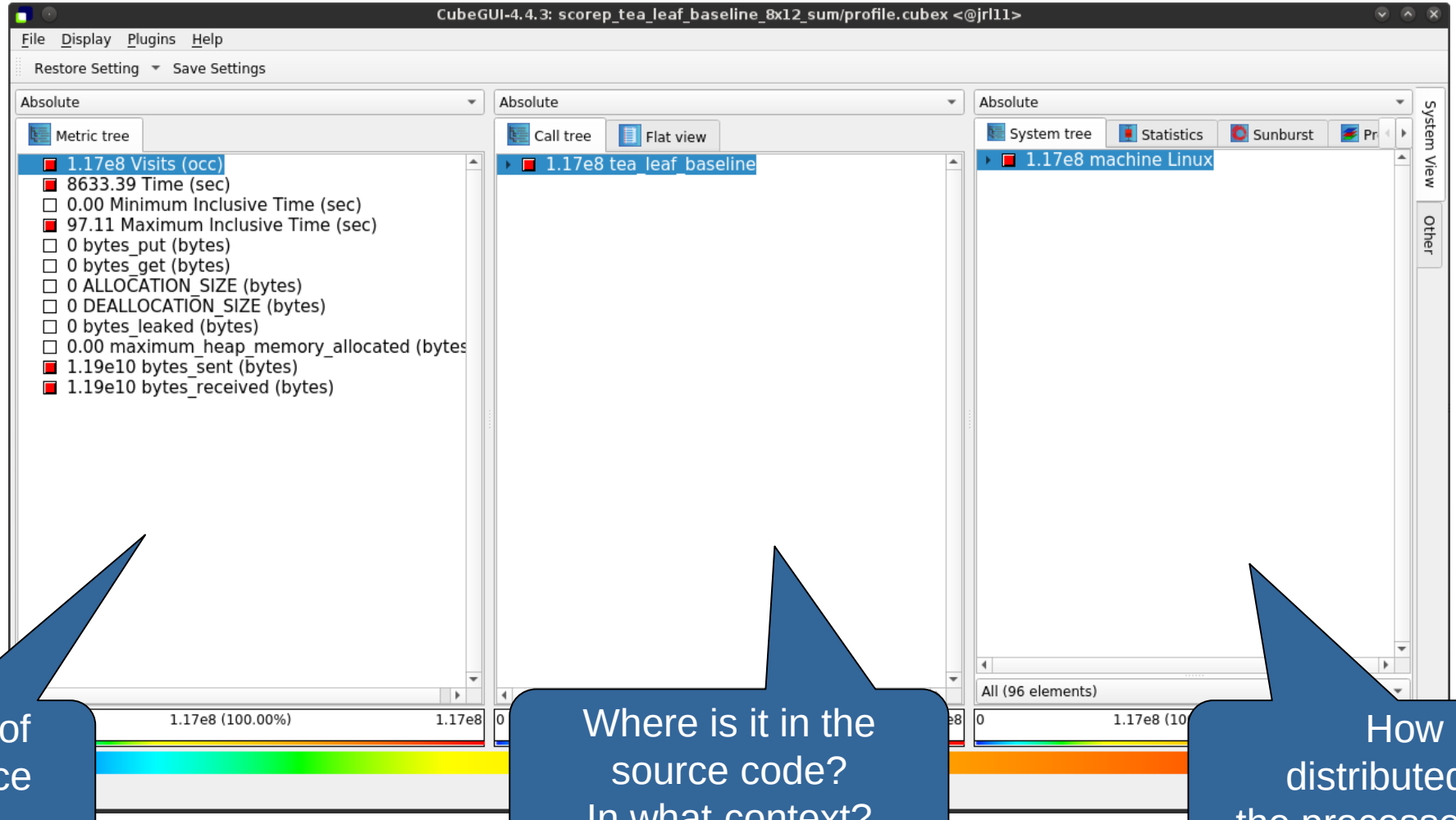
Case study: TeaLeaf

- HPC mini-app developed by the UK Mini-App Consortium
  - Solves the linear 2D heat conduction equation on a spatially decomposed regular grid using a 5 point stencil with implicit solvers
  - Part of the Mantevo 3.0 suite
  - Available on GitHub: http://uk-mac.github.io/TeaLeaf/

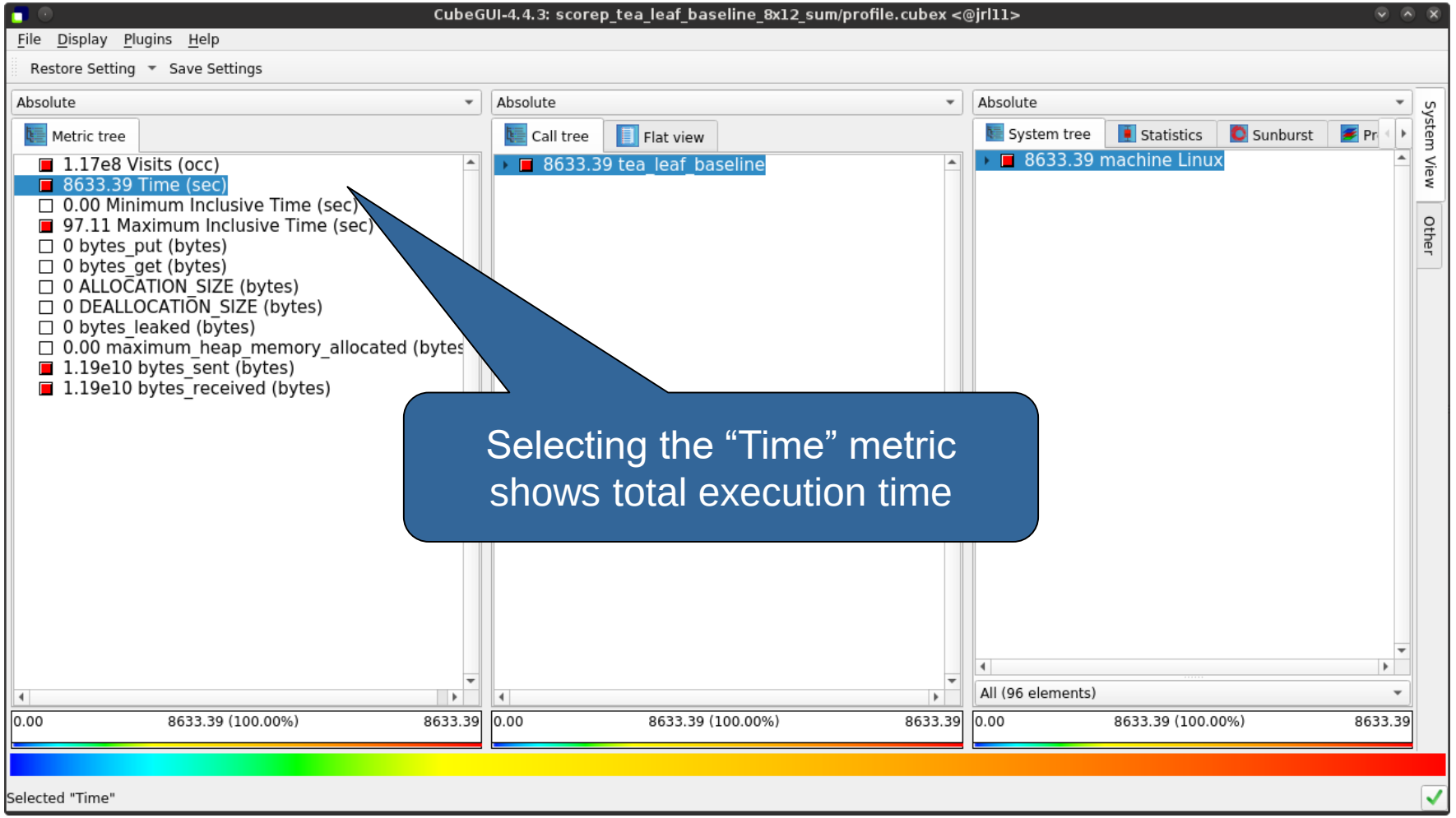
- Measurements of TeaLeaf reference v1.0 taken on Jureca cluster @ JSC
  - Using Intel 19.0.3 compilers, Intel MPI 2019.3, and Score-P 5.0
  - Run configuration
    - 8 MPI ranks with 12 OpenMP threads each
    - Distributed across 4 compute nodes (2 ranks per node)
    - Test problem "5": 4000 × 4000 cells, CG solver

```
% cd ~/workshop-vihps/Experiments
% cube scorep_tea_leaf_baseline_8x12_sum/profile.cubex
                                    [GUI showing summary analysis report]
```

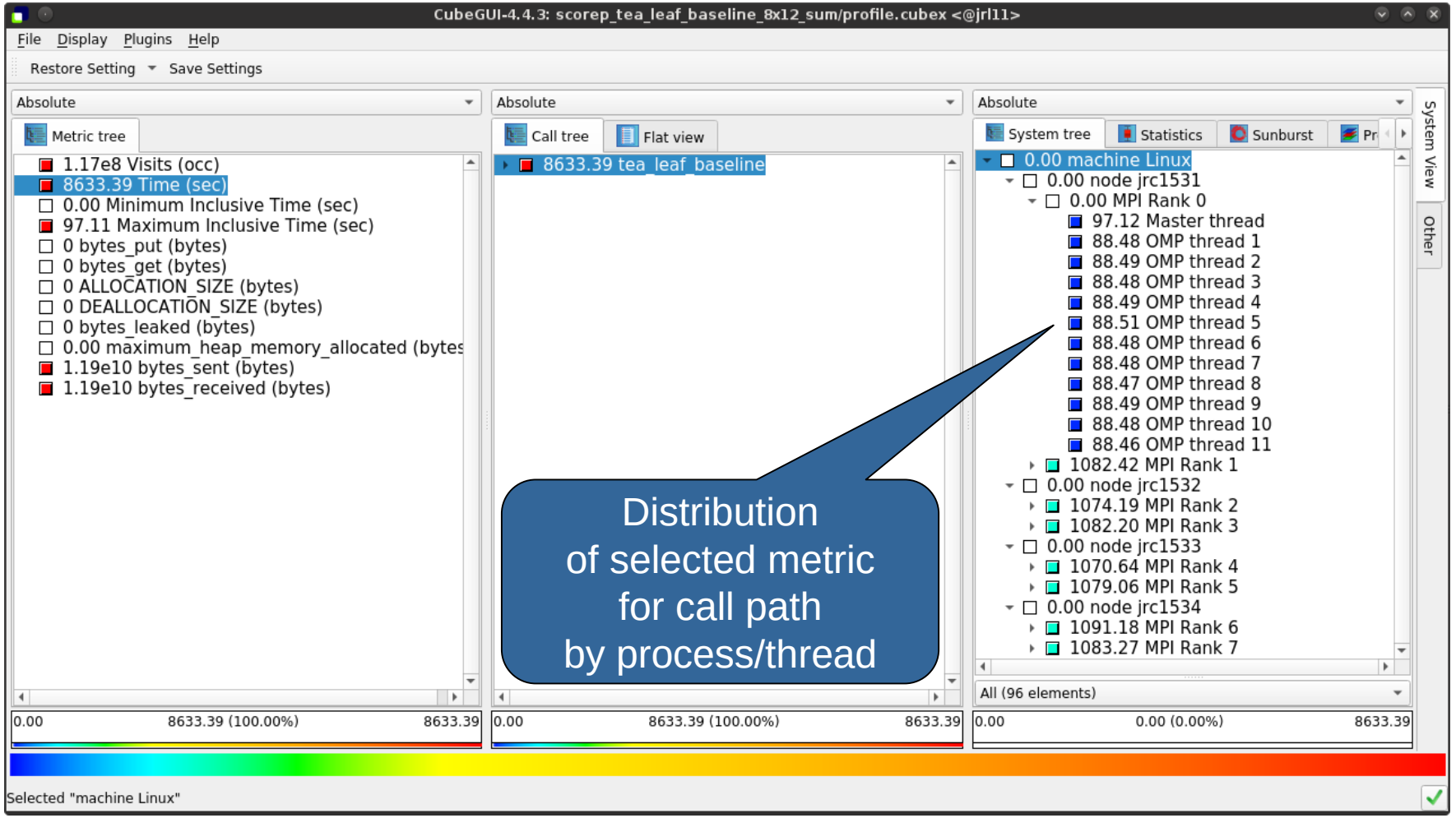
Score-P analysis report exploration (opening view)



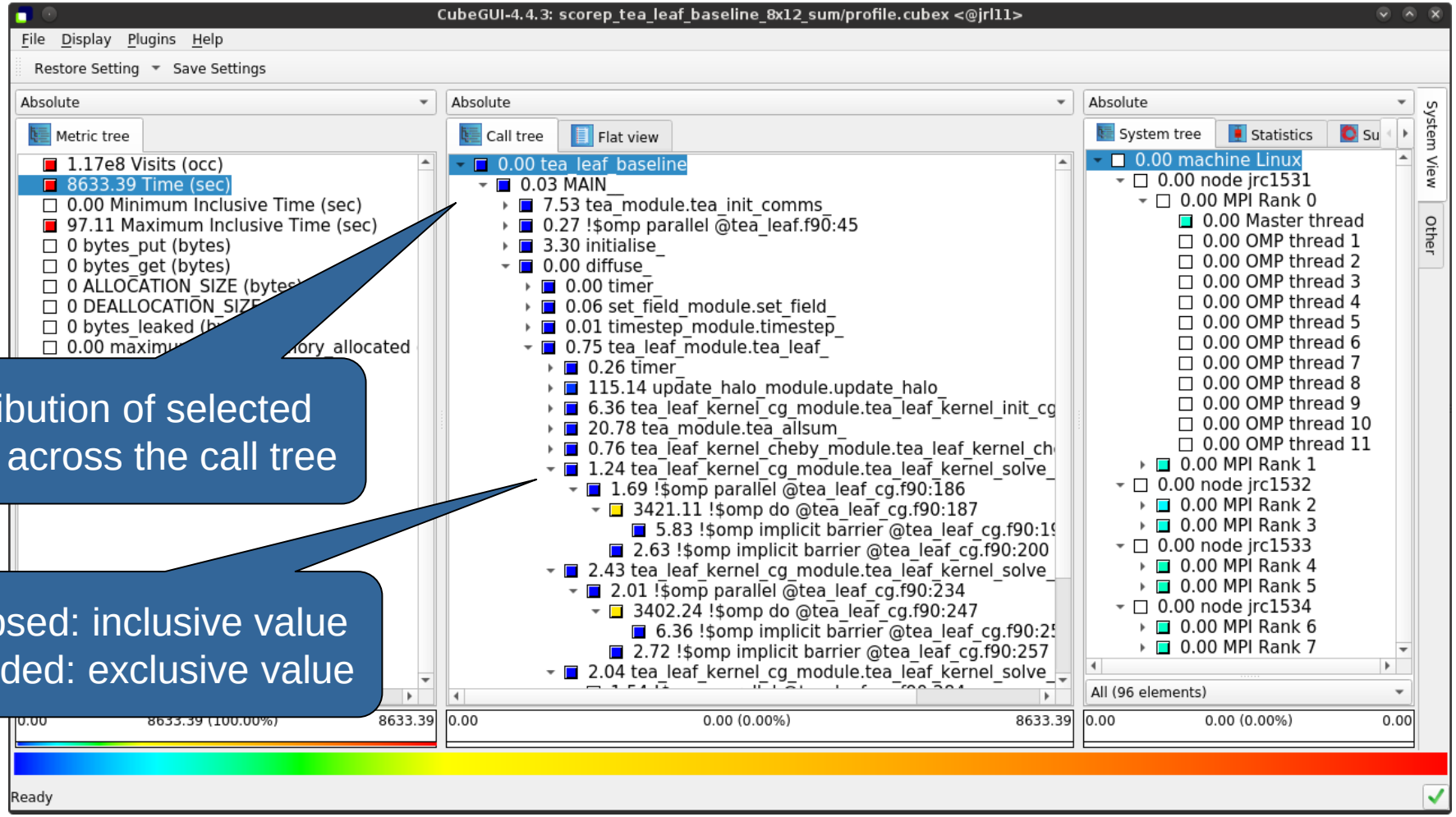
Metric selection



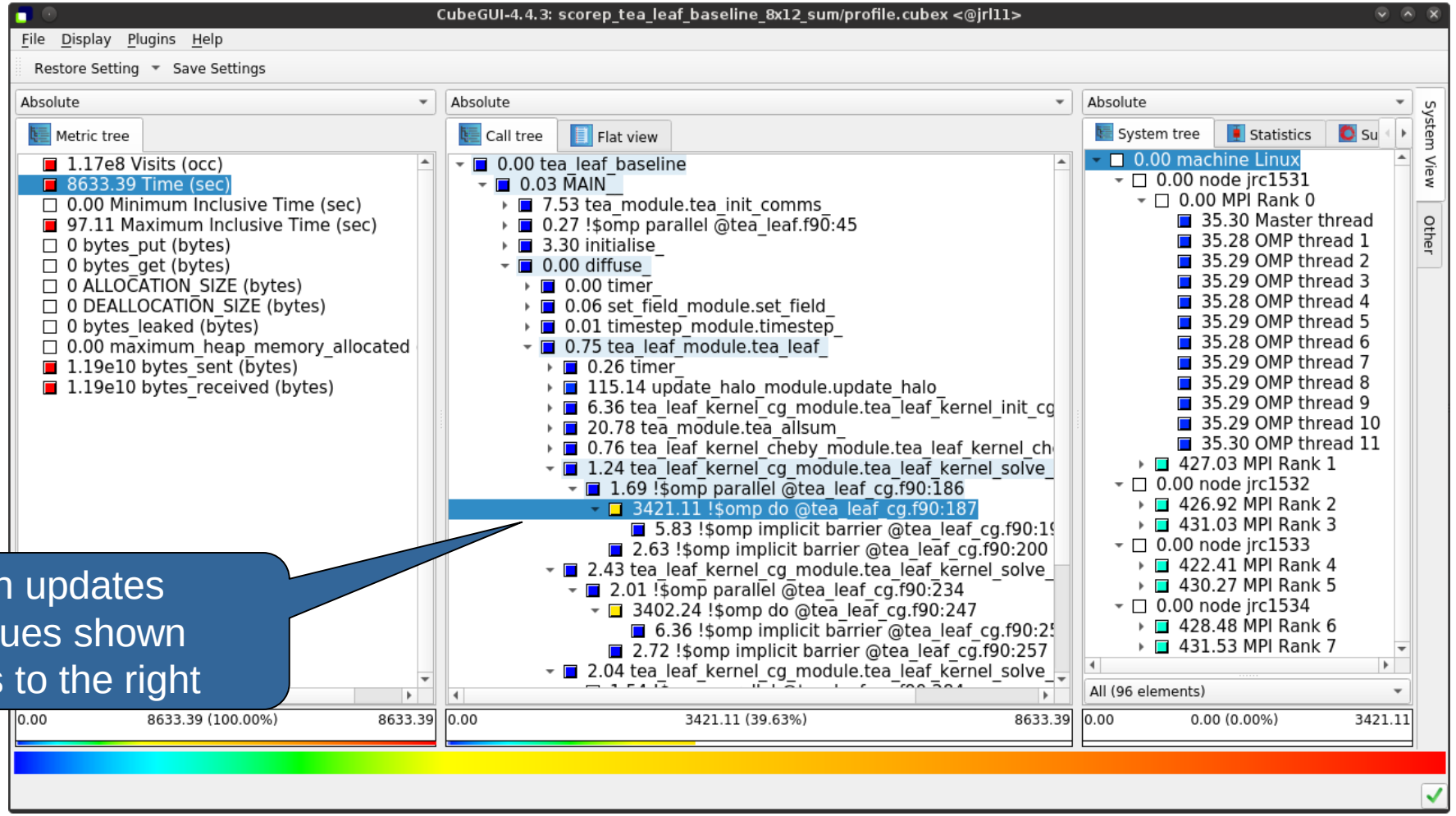
Expanding the system tree



Expanding the call tree

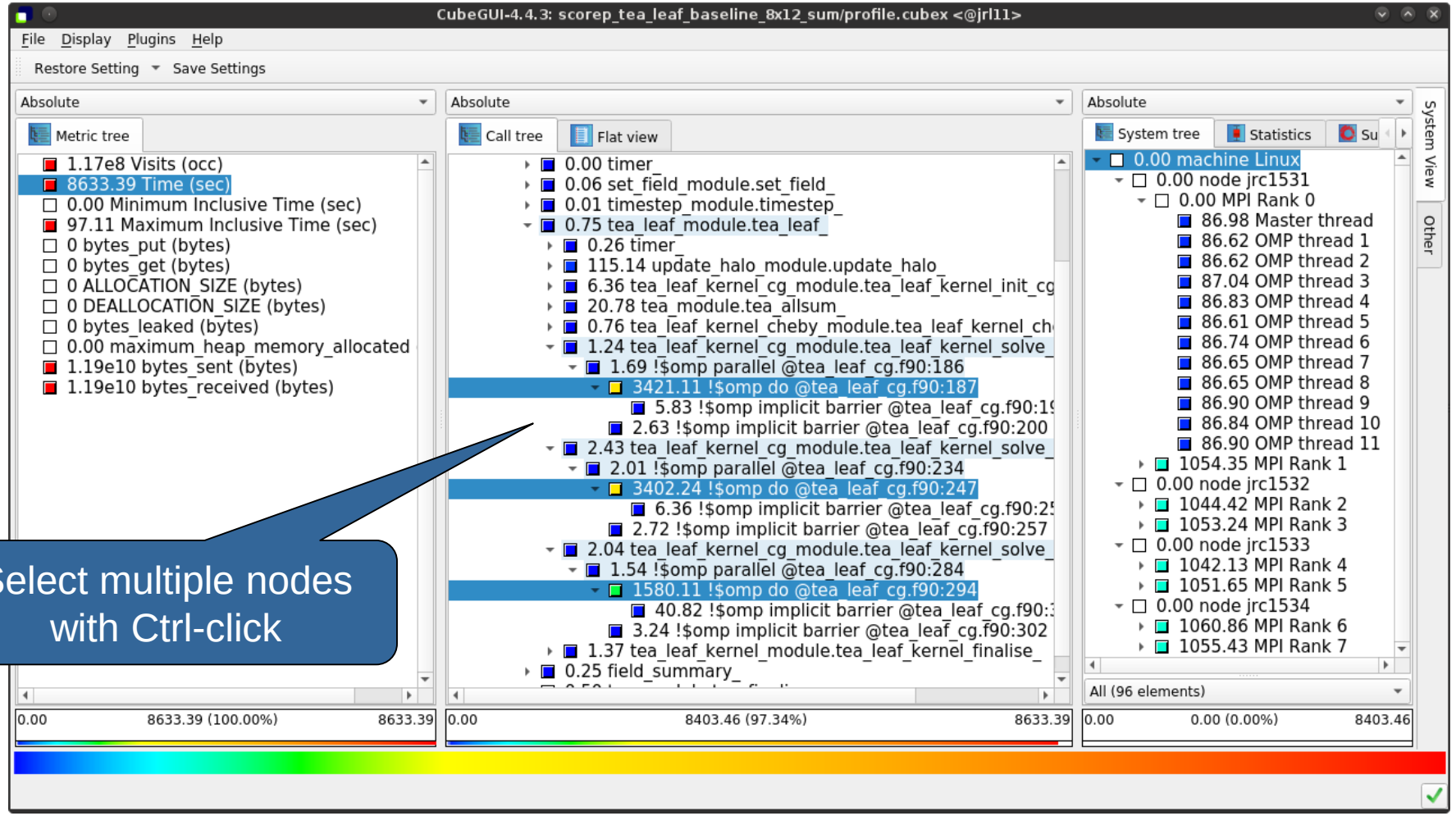


Selecting a call path

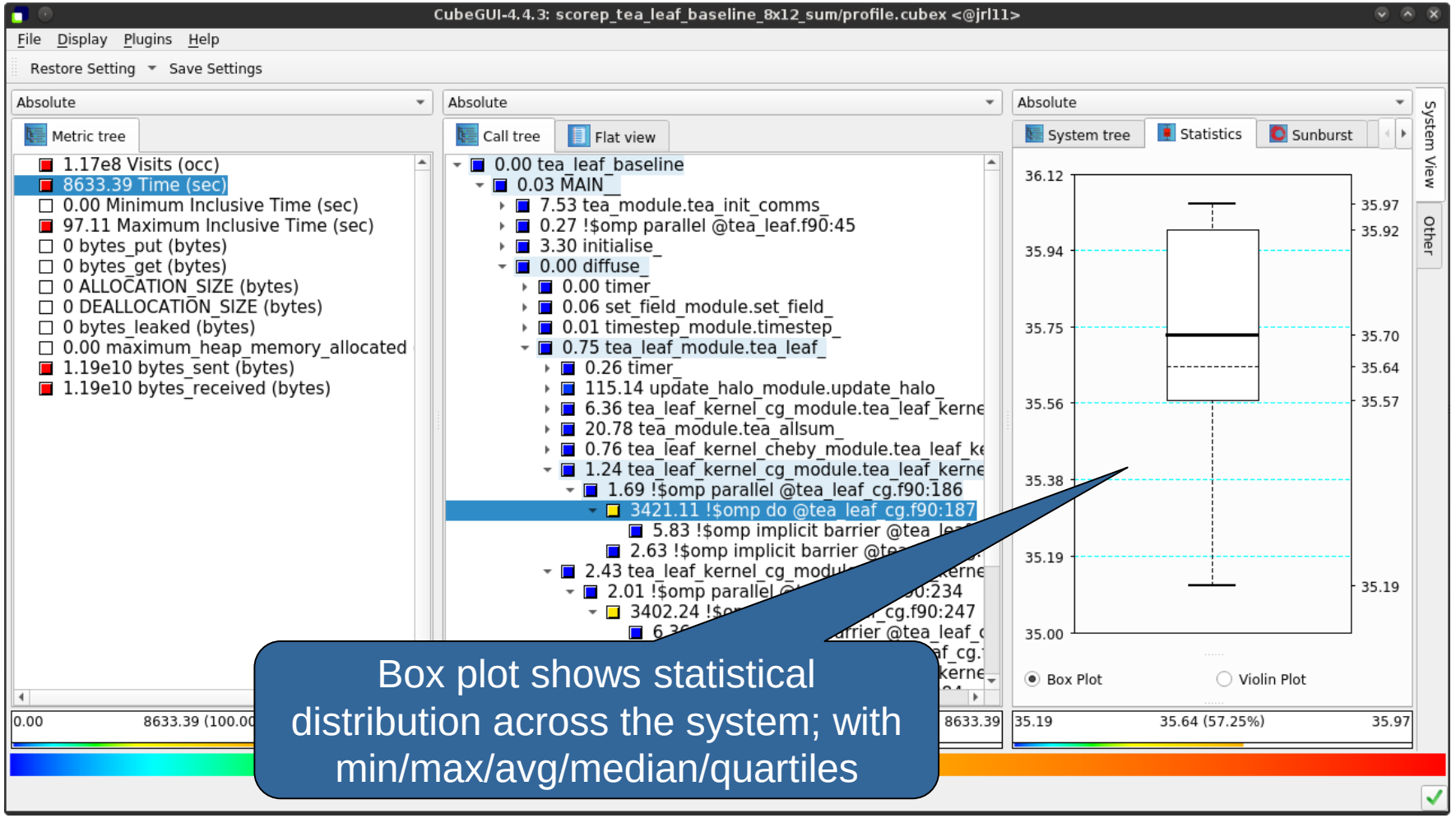


Selection updates metric values shown in columns to the right

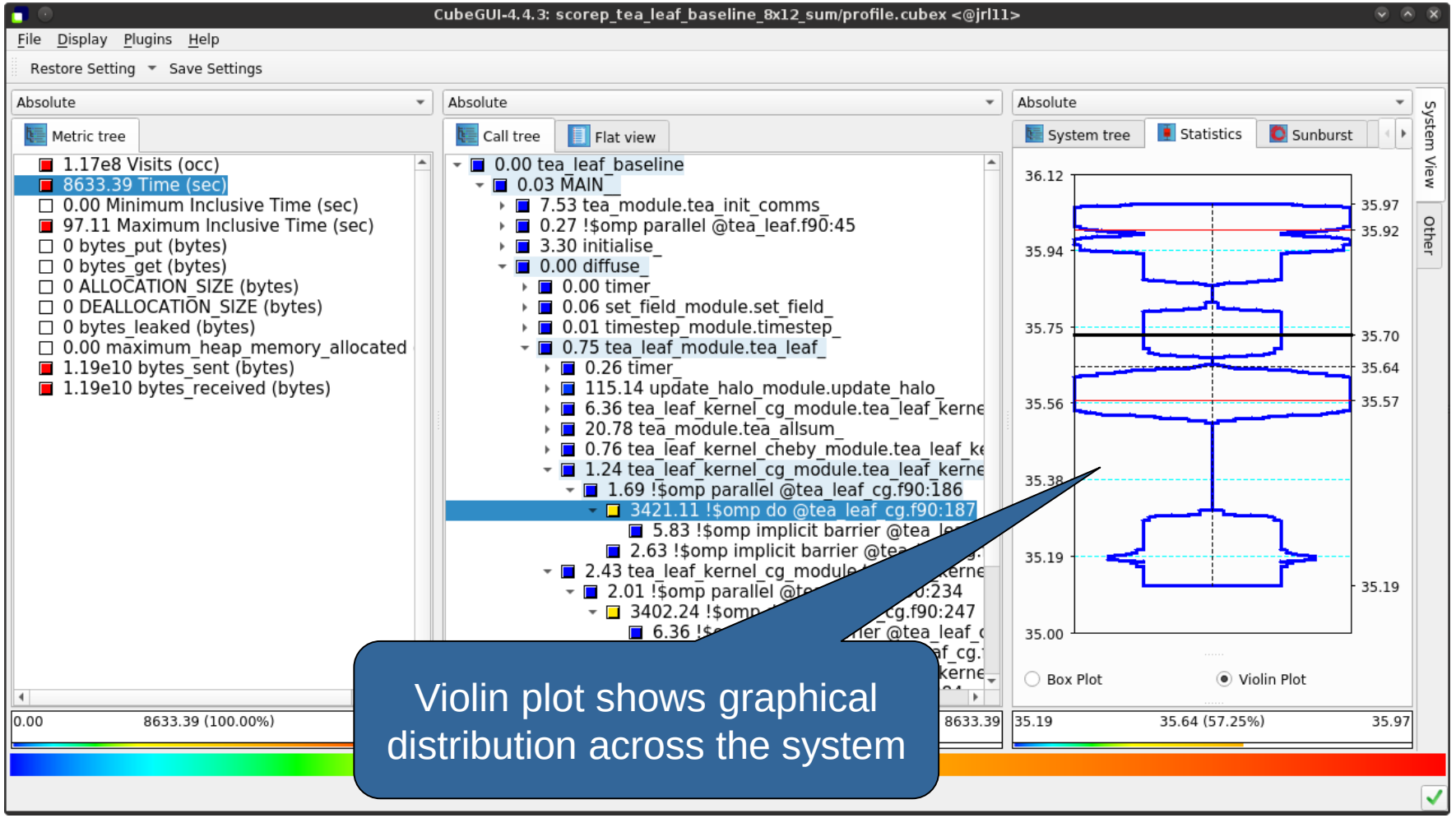
Multiple selection



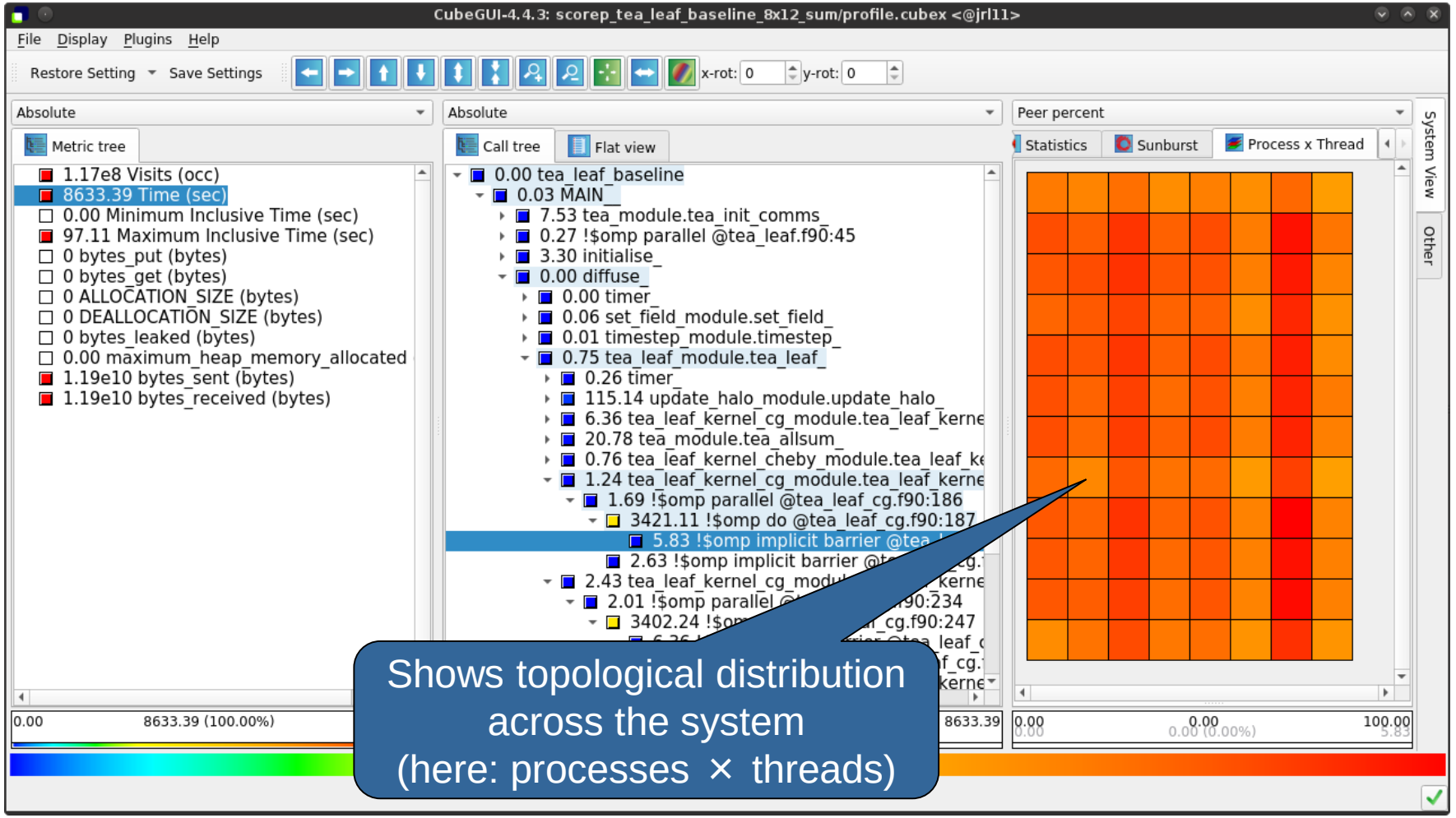
Box plot view



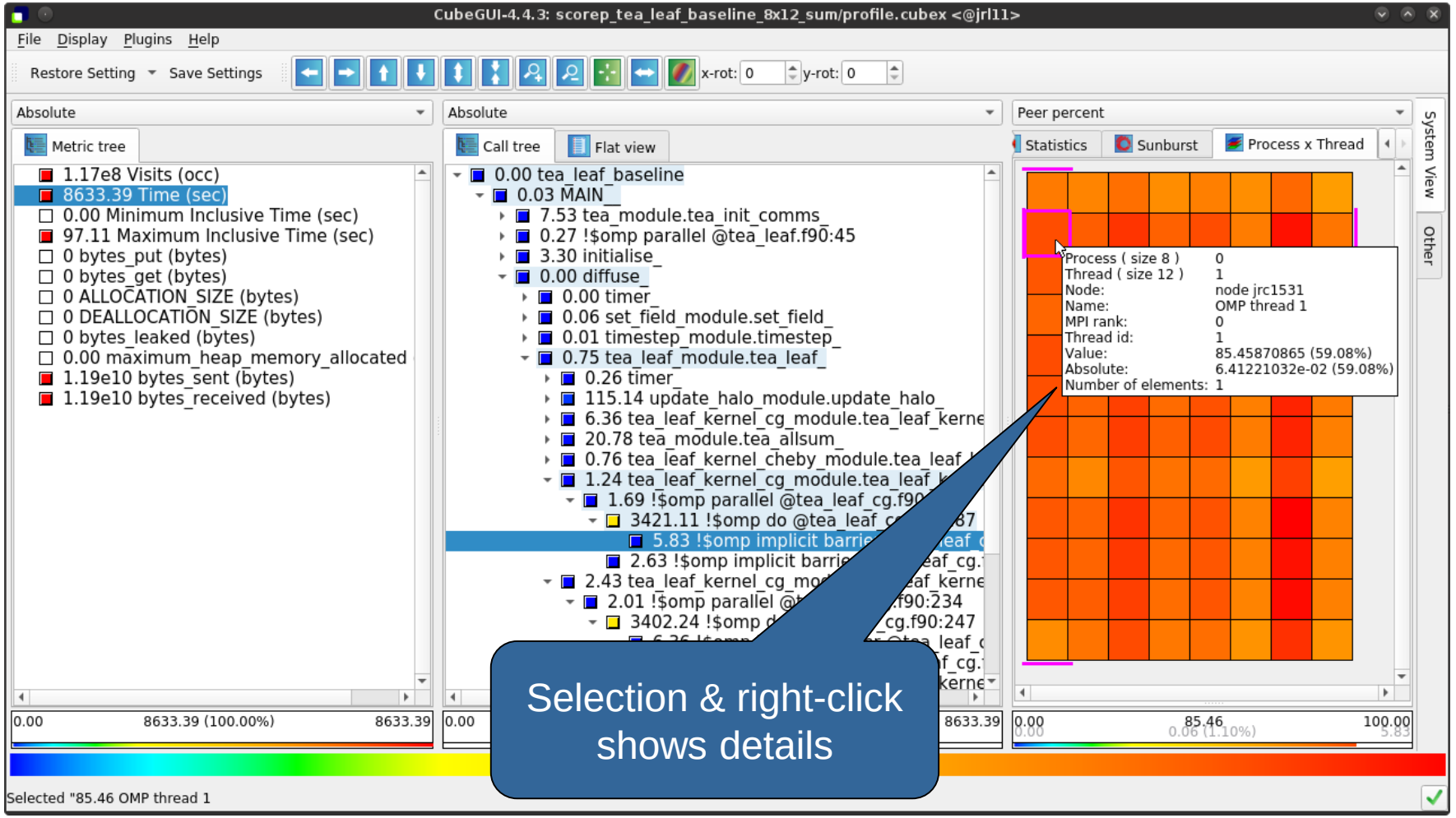
Violin plot view



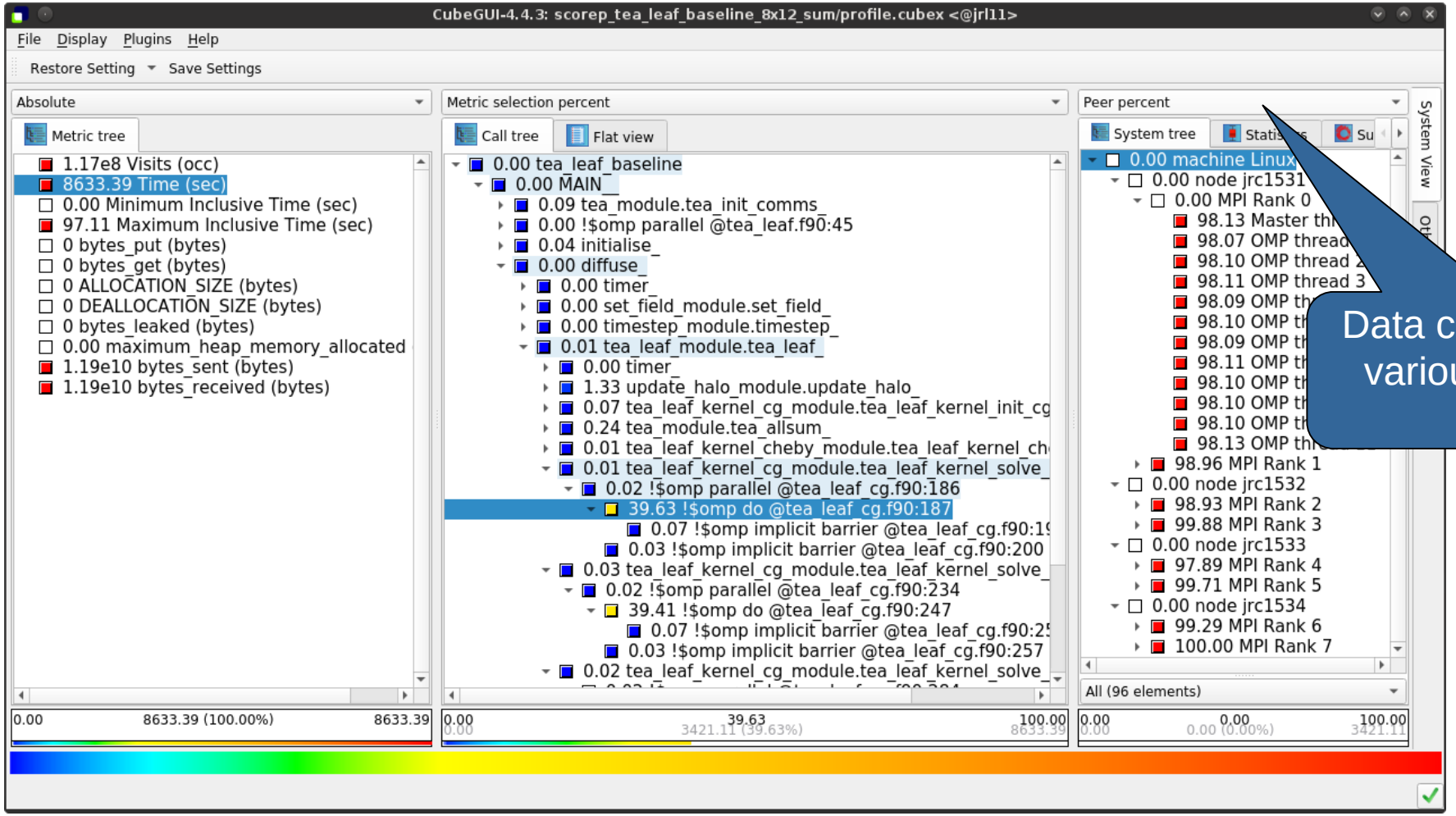
Topology view



Topology view (cont.)



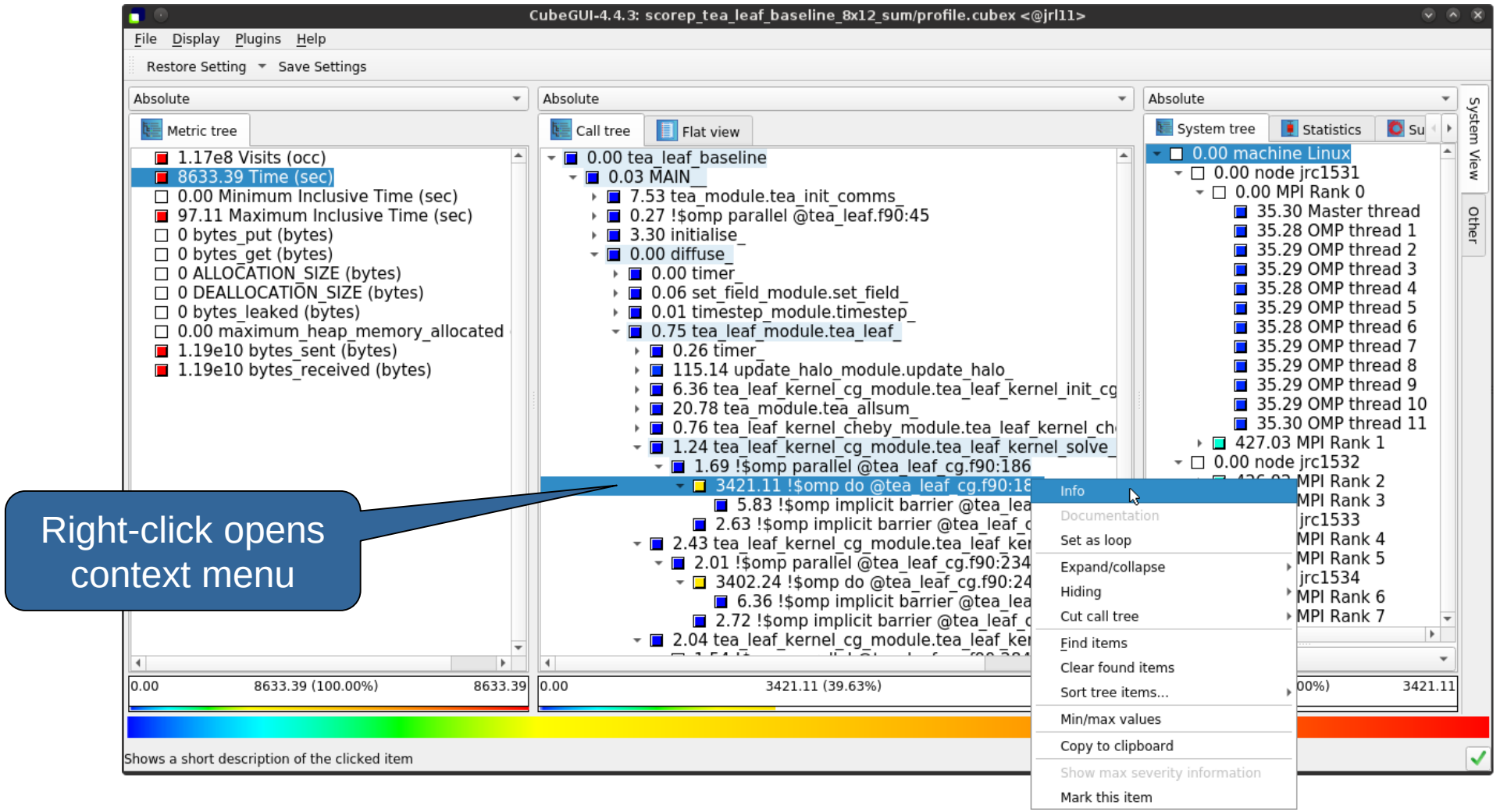
Alternative display modes



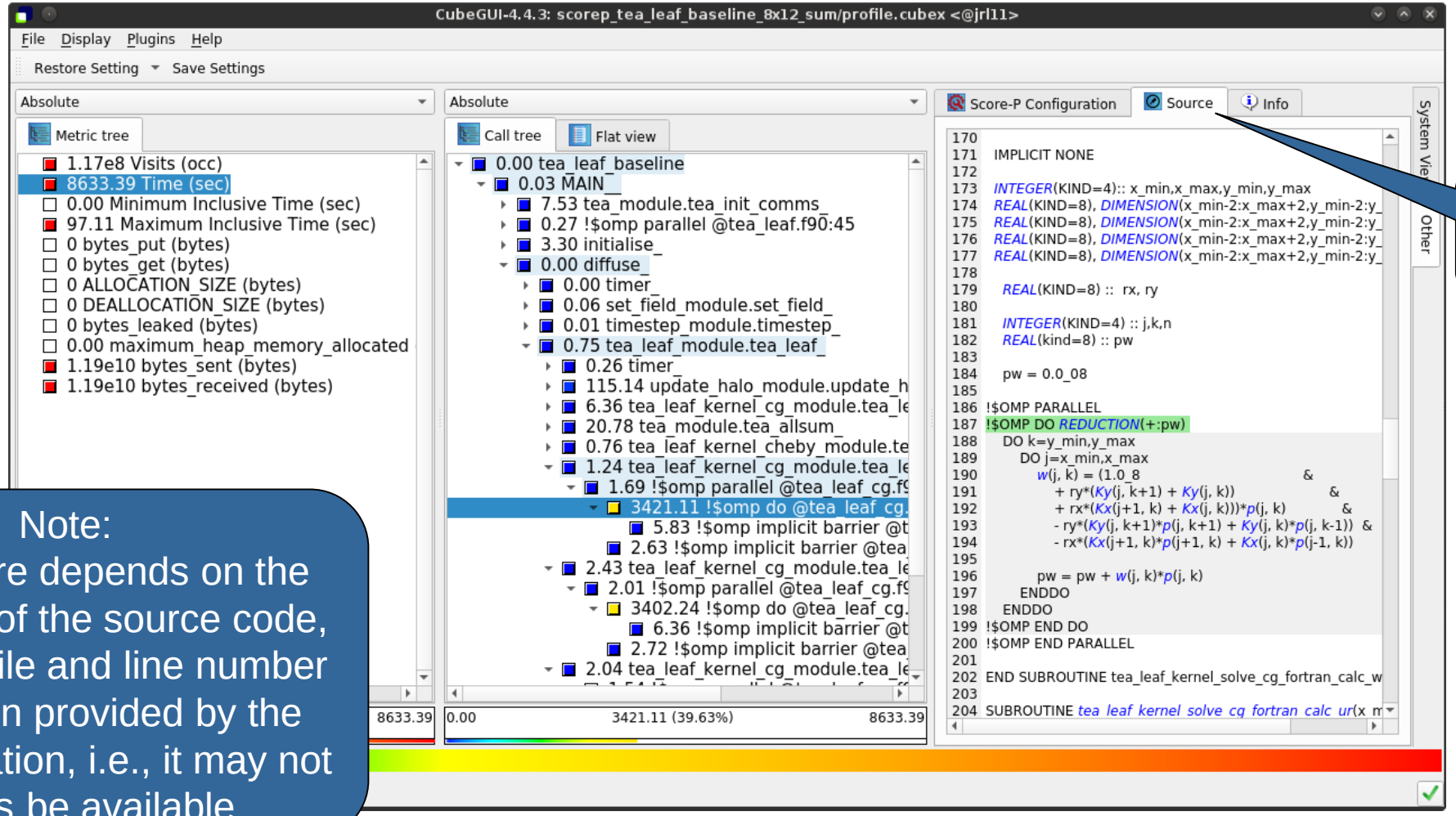
Important display modes

- 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

Source-code view via context menu



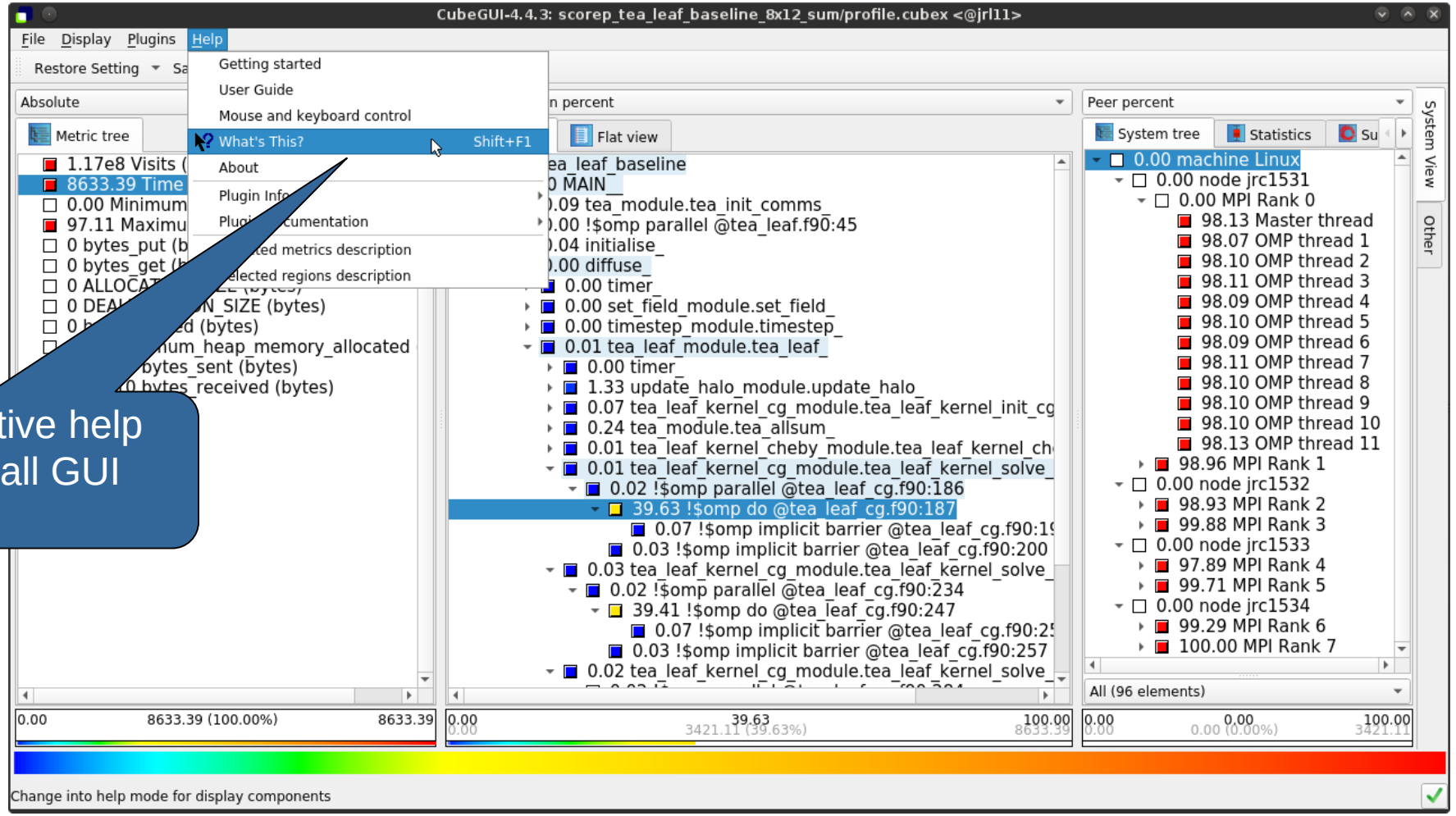
Source-code view



Select “Source” tab

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

Context-sensitive help



Context-sensitive help available for all GUI items

Scalasca report post-processing

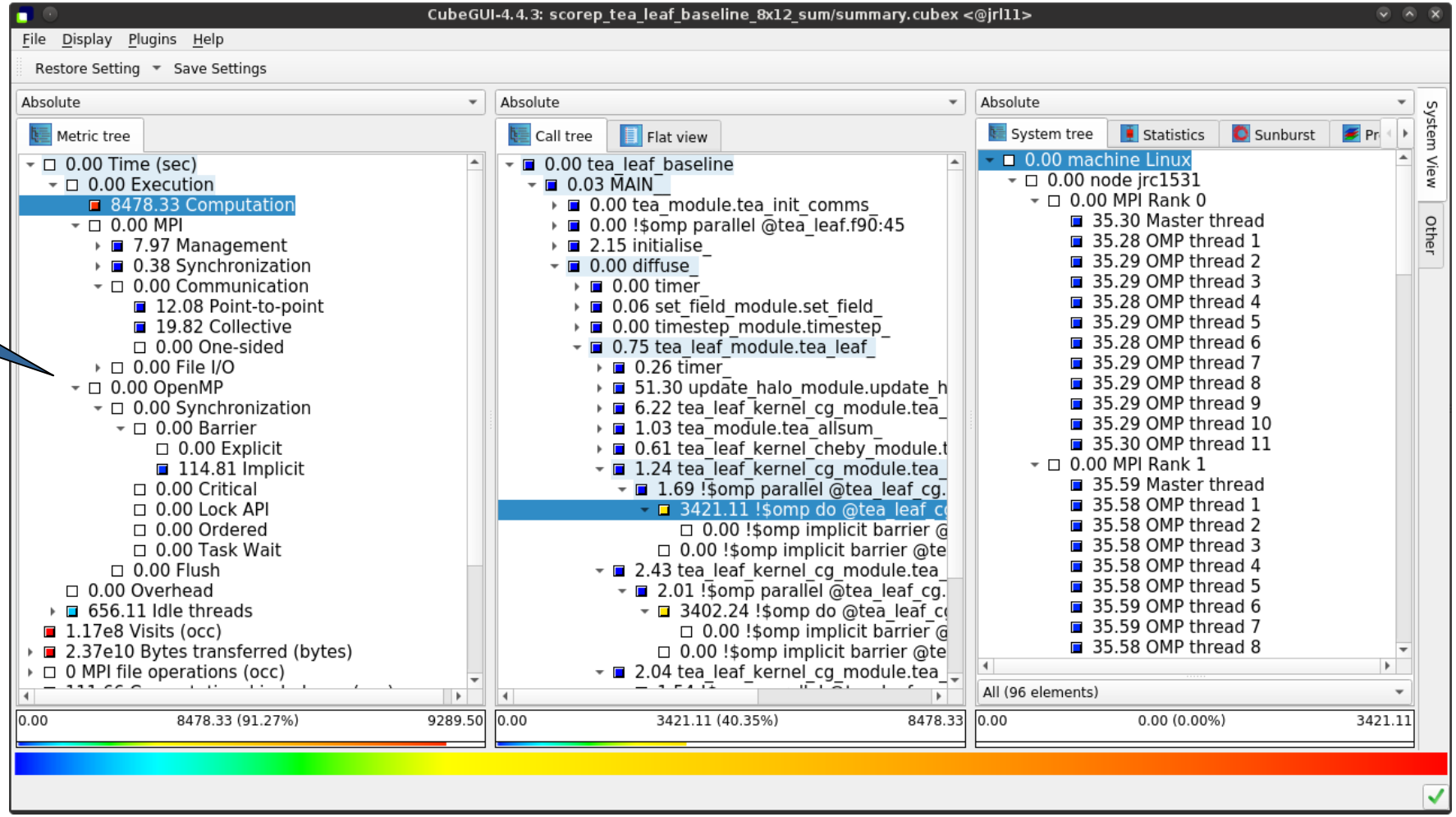
- Scalasca's report post-processing derives additional metrics and generates a structured metric hierarchy

- Automatically run (if needed) when using the **square** convenience command:

```
% square scorep_tea_leaf_baseline_8x12_sum
INFO: Post-processing runtime summarization report (profile.cubex)...
INFO: Displaying ./scorep_tea_leaf_baseline_8x12_sum/summary.cubex...

                          [GUI showing post-processed summary analysis report]
```

Post-processed summary analysis report

Split base metrics into more specific metrics

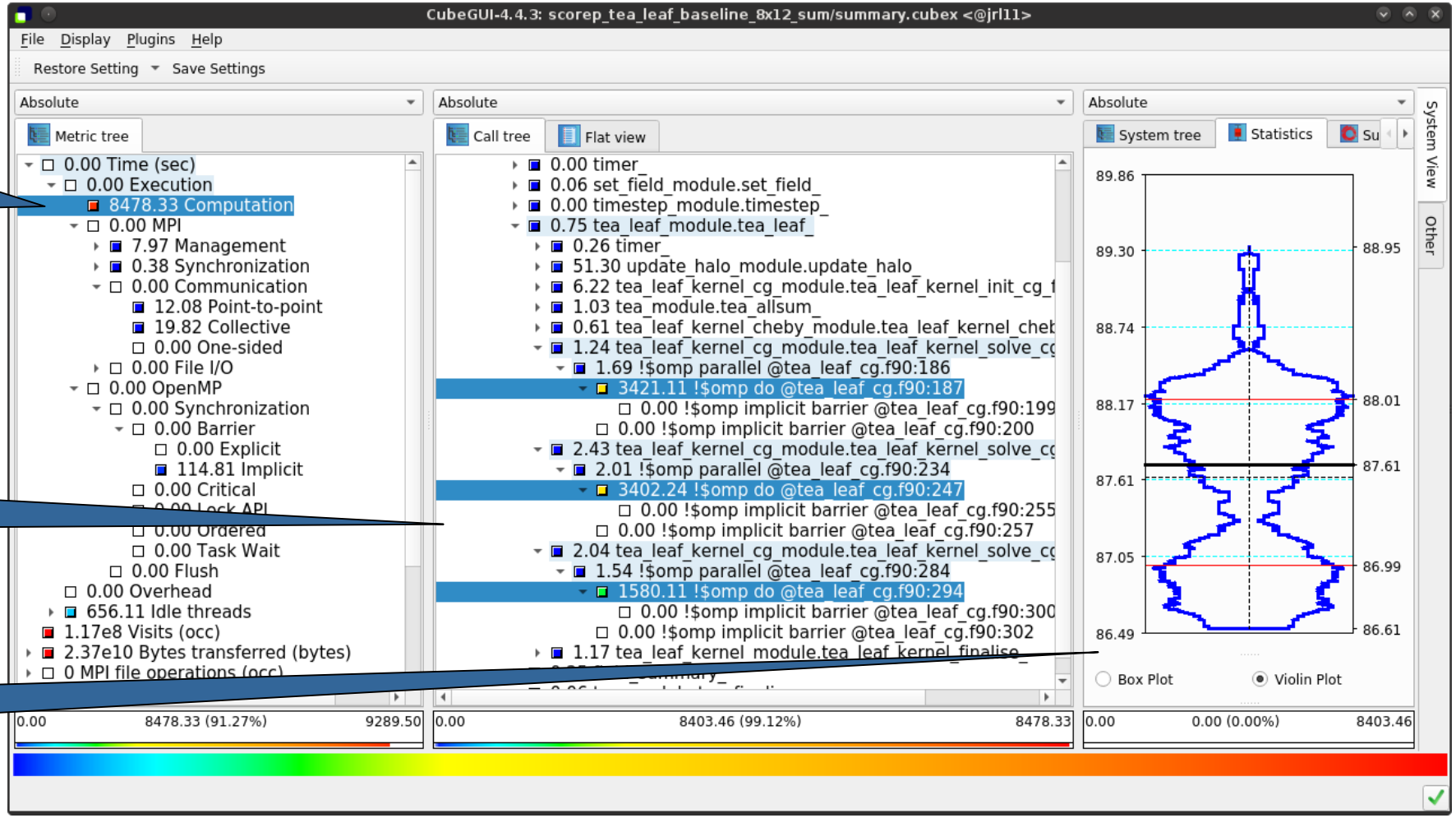


TeaLeaf summary report analysis (I)

91% of the execution time is computation…

…almost entirely spent in 3 OpenMP `do` loops…

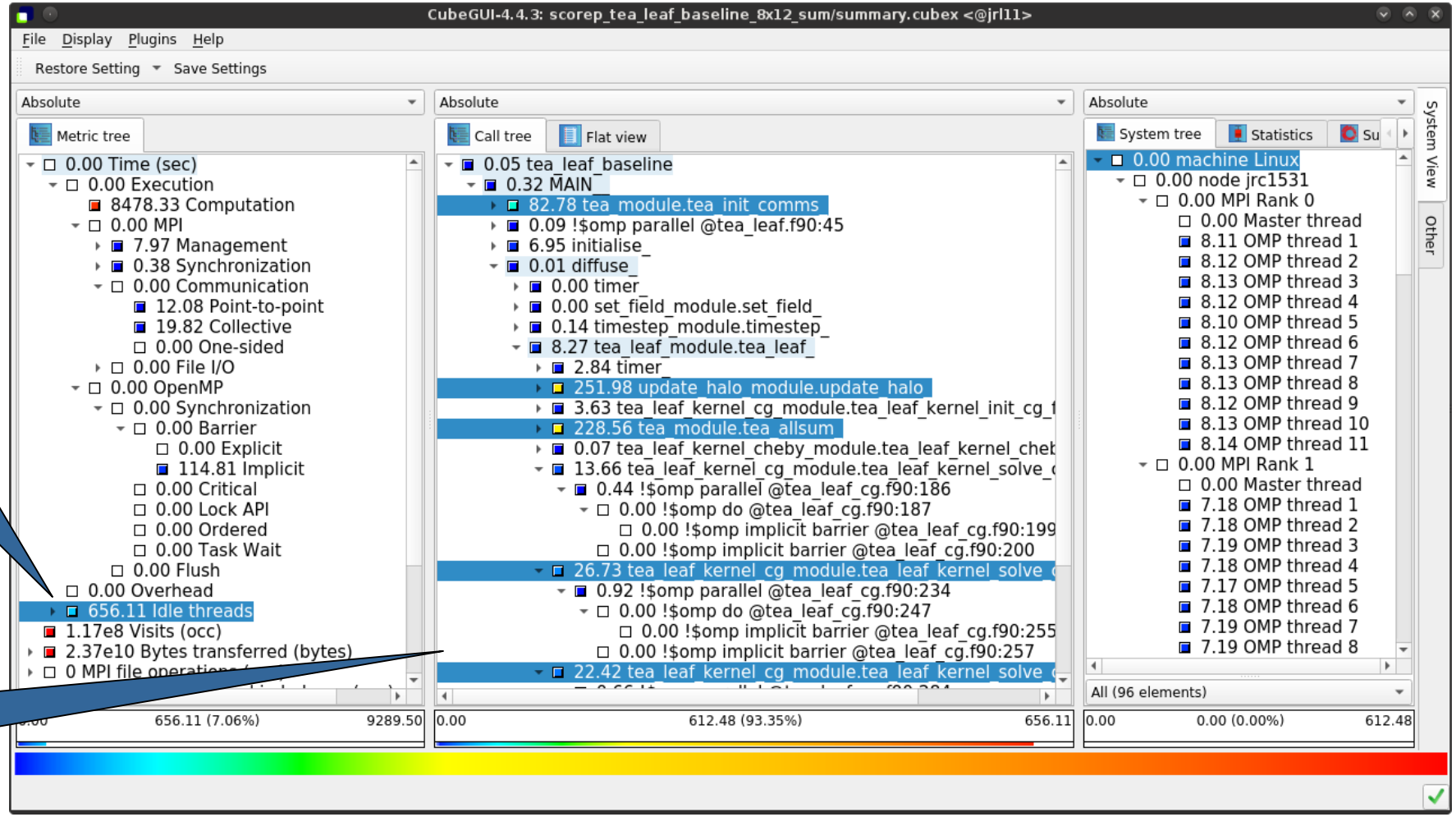
…with a slight imbalance across ranks & threads



TeaLeaf summary report analysis (II)

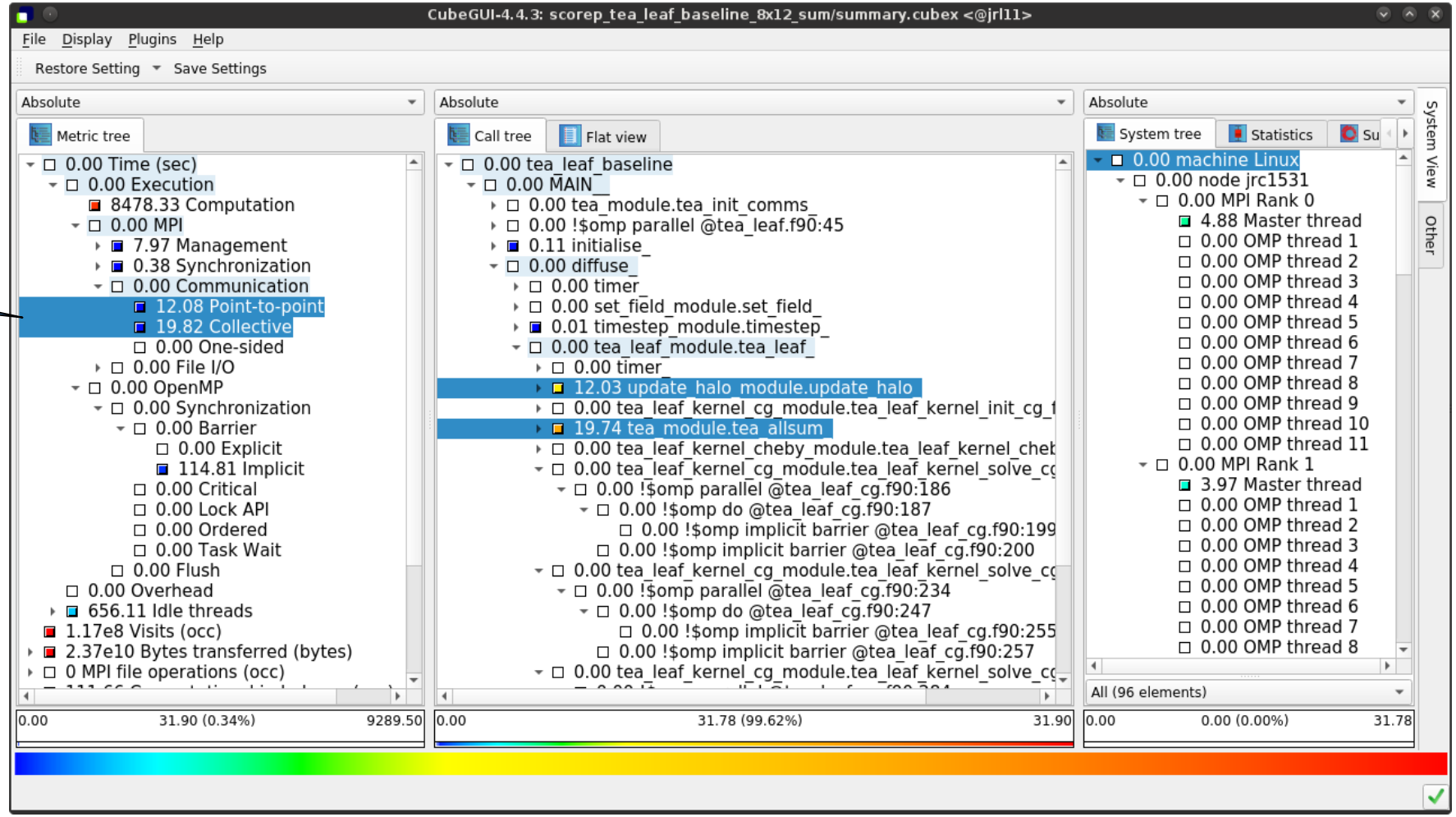
7% of the execution time are lost due to idle threads…

…in non-OpenMP parallelized code regions



TeaLeaf summary report analysis (III)

MPI communication time is negligible (0.34%); communication is only on the master threads (MPI_THREAD_FUNNELED)



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:
  - https://www.scalasca.org
- User guide also part of installation:
  - <prefix>/share/doc/cubegui/CubeUserGuide.pdf
- Contact:
  - mailto: scalasca@fz-juelich.de

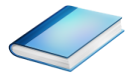
Acknowledgement

This tutorial is sponsored by the DEEP-SEA project.

www.deep-projects.eu
@DEEPprojects
@deep-projects

The DEEP Projects have received funding from the European Commission's FP7, H2020, and EuroHPC Programmes, under Grant Agreements n° 287530, 610476, 754304, and 955606.

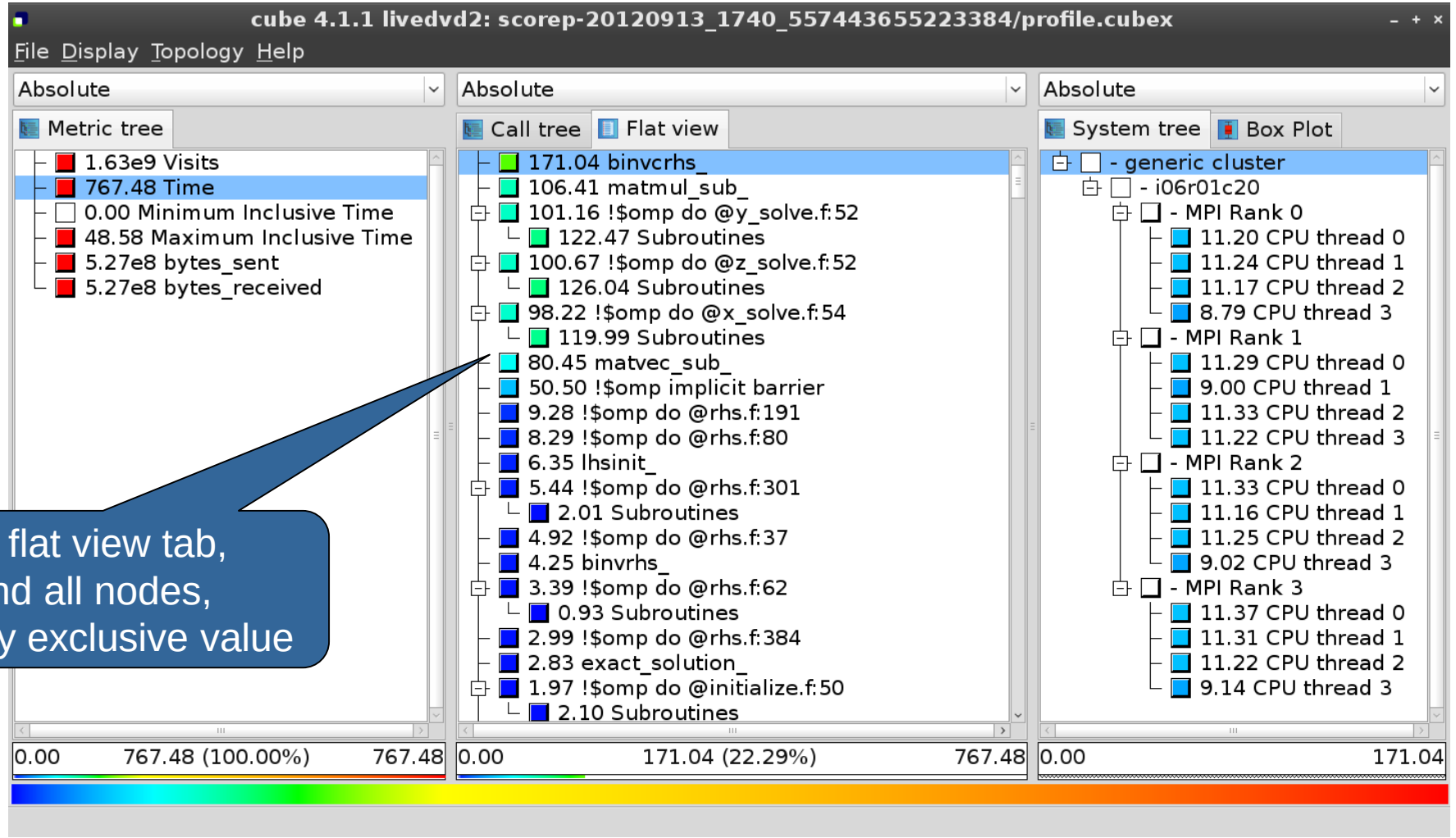
The EuroHPC Joint Undertaking (JU) receives support from the European Union's Horizon 2020 research and innovation programme and Germany, France, Spain, Greece, Belgium, Sweden, United Kingdom, Switzerland.

Reference material 

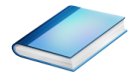
---

cube
scalasca

Flat profile view



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

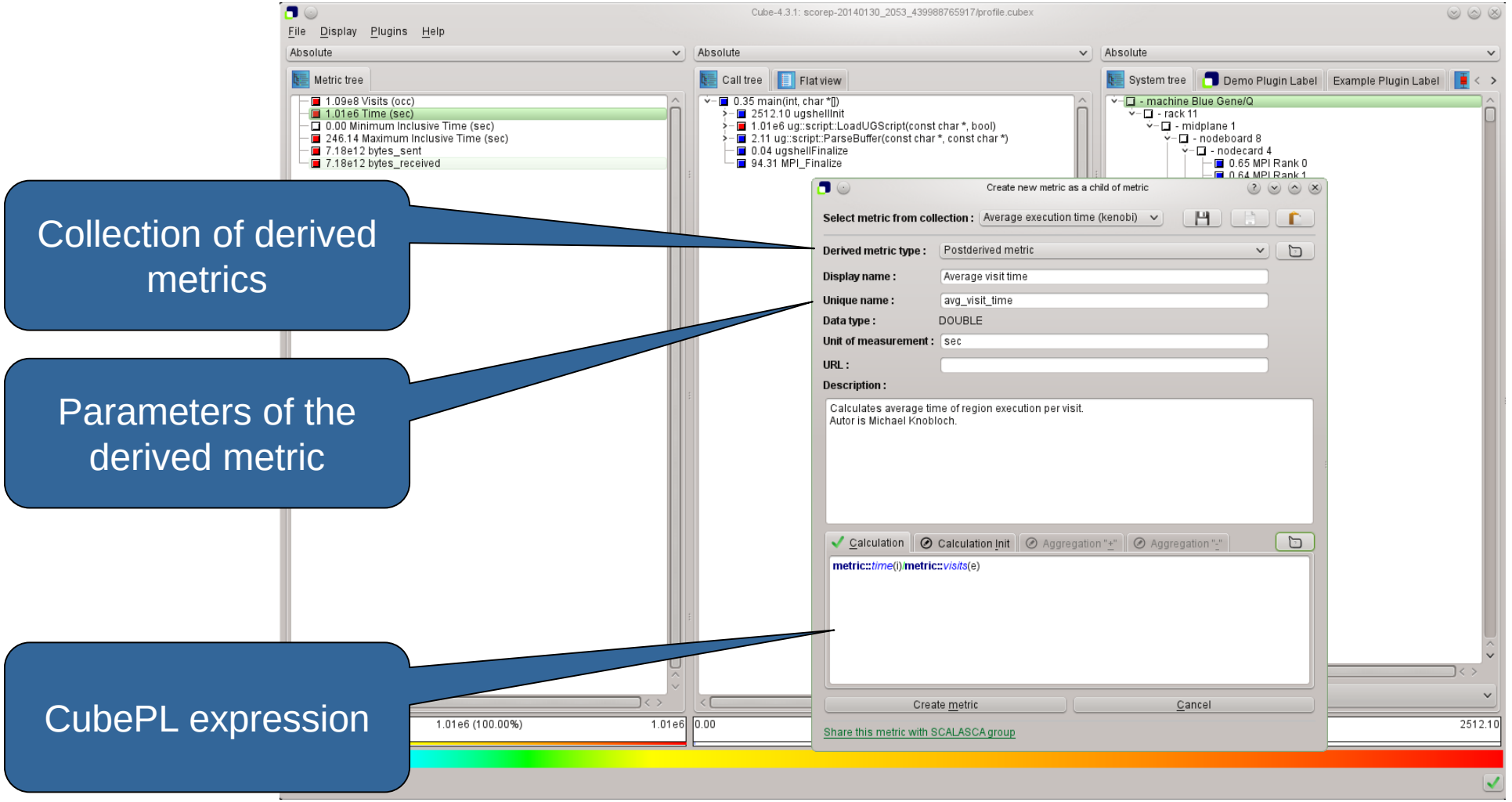
- Examples:
  - "Average execution time": Postderived metric with expression

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

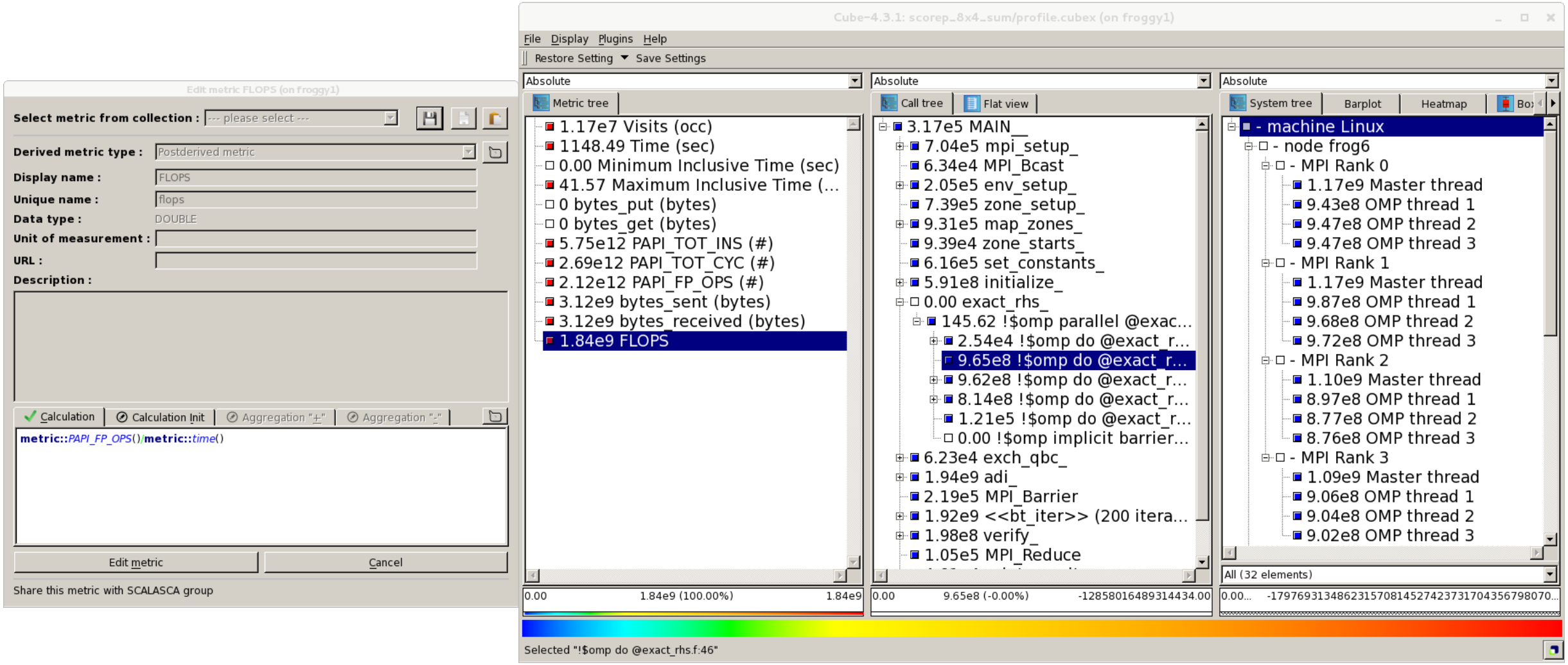
  - "Number of FLOP per second": Postderived metric with expression

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

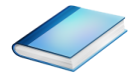
Derived metrics in Cube GUI



Example: FLOPS based on PAPI_FP_OPS and time



CUBE algebra utilities



- Extracting solver sub-tree from analysis report

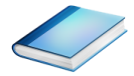
```
% cube_cut  -r '<<ITERATION>>'  scorep_bt-mz_C_32x4_sum/profile.cubex
Writing cut.cubex... done.
```

- 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

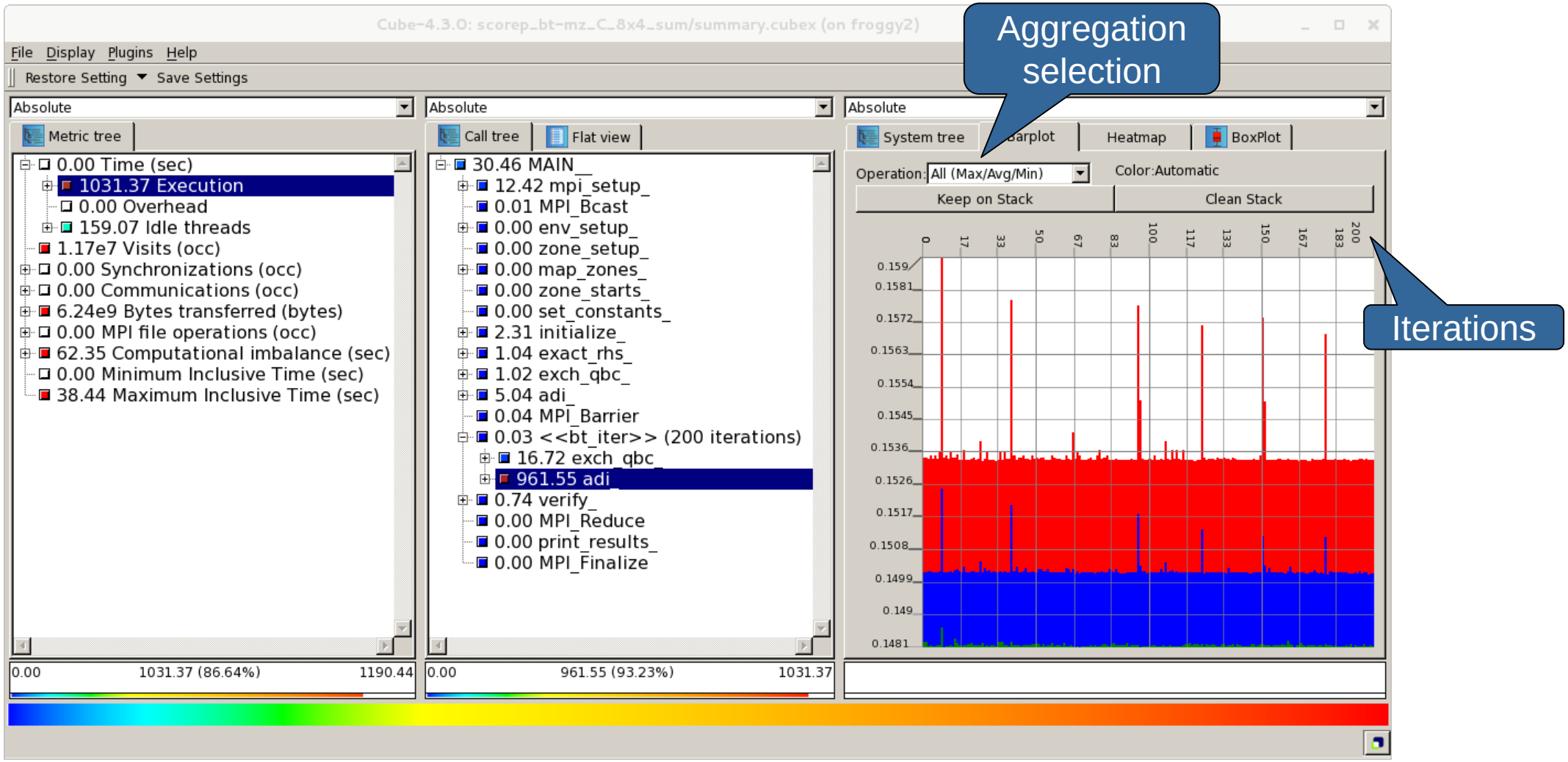
➢ Useful for checking results for specific iterations

or

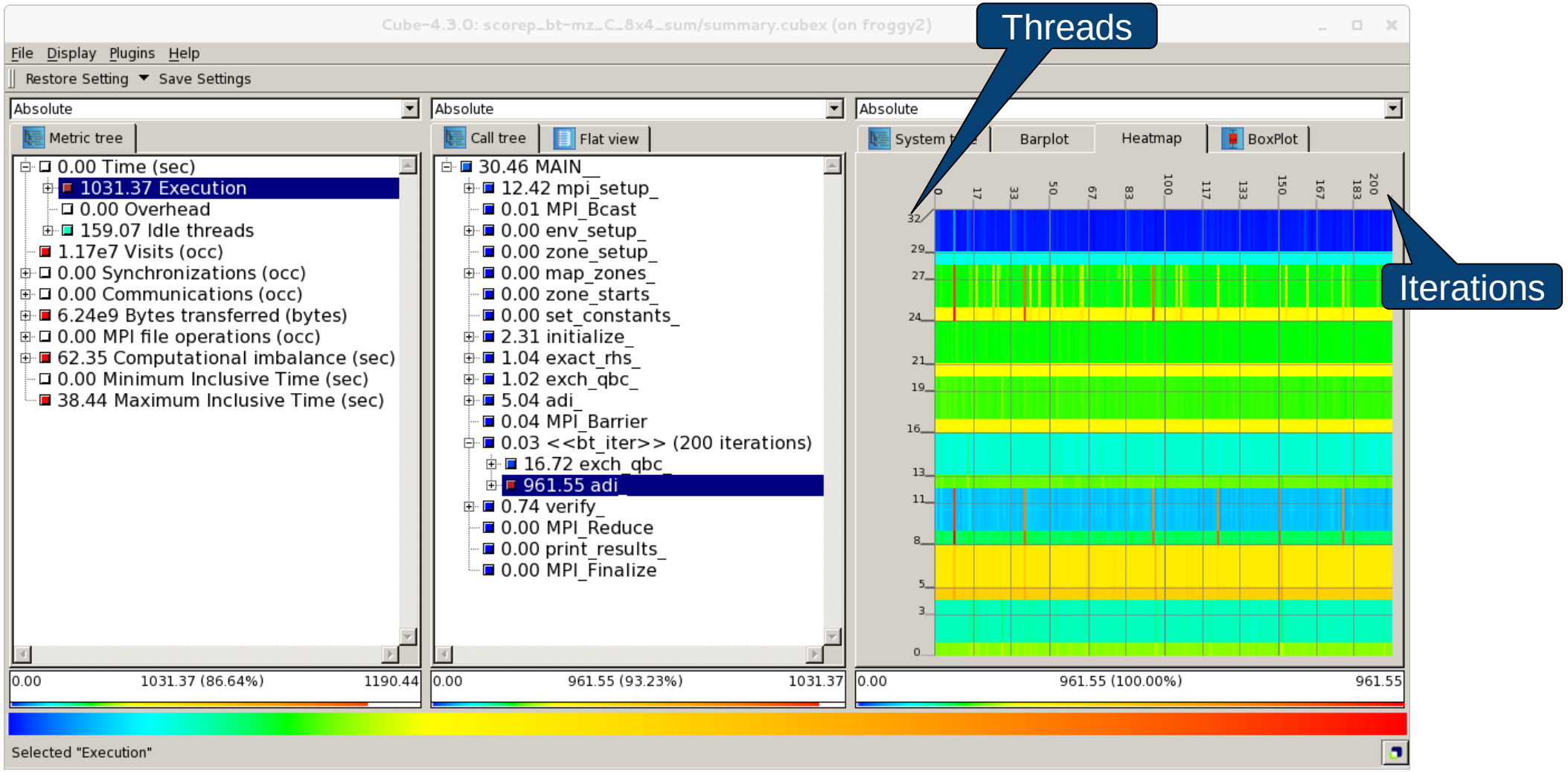
  - Select your user-instrumented region and mark it as loop
  - Choose “Hide iterations”

➢ View the Barplot statistics or the (thread x iterations) Heatmap

Iteration profiling: Barplot

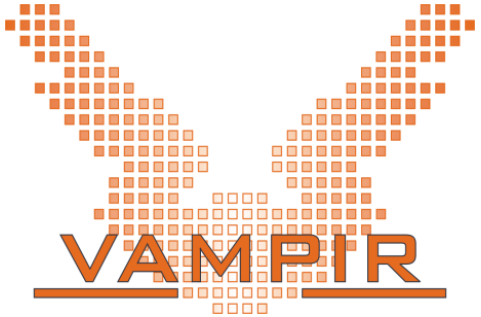


Iteration profiling: Heatmap



Performance Analysis with Vampir

Bill Williams
TU Dresden

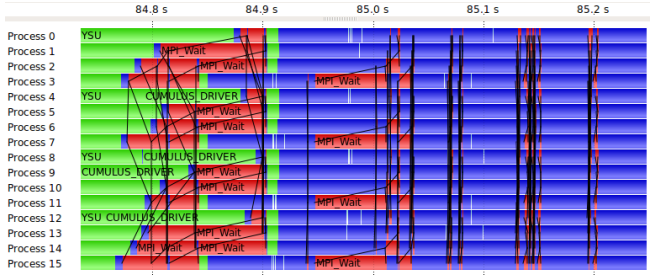


Event Trace Visualization with Vampir

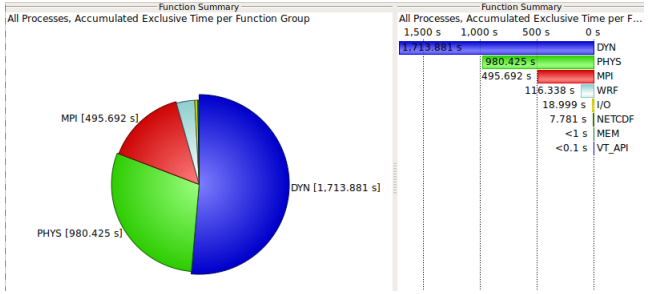
- Visualization of dynamic runtime behaviour at any level of detail along with statistics and performance metrics
- Alternative and supplement to automatic analysis

- **Typical questions that Vampir helps to answer**
  - What happens in my application execution during a given time in a given process or thread?
  - How do the communication patterns of my application execute on a real system?
  - Are there any imbalances in computation, I/O or memory usage and how do they affect the parallel execution of my application?

- **Timeline charts**
  - Application activities and communication along a time axis

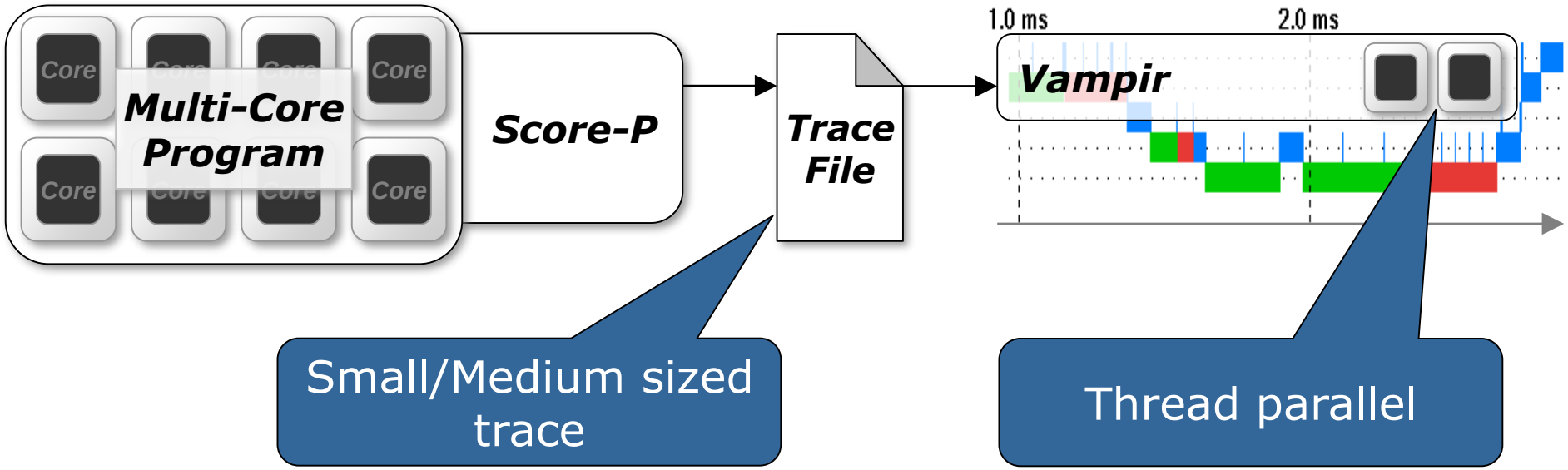


- **Summary charts**
  - Quantitative results for the currently selected time interval



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

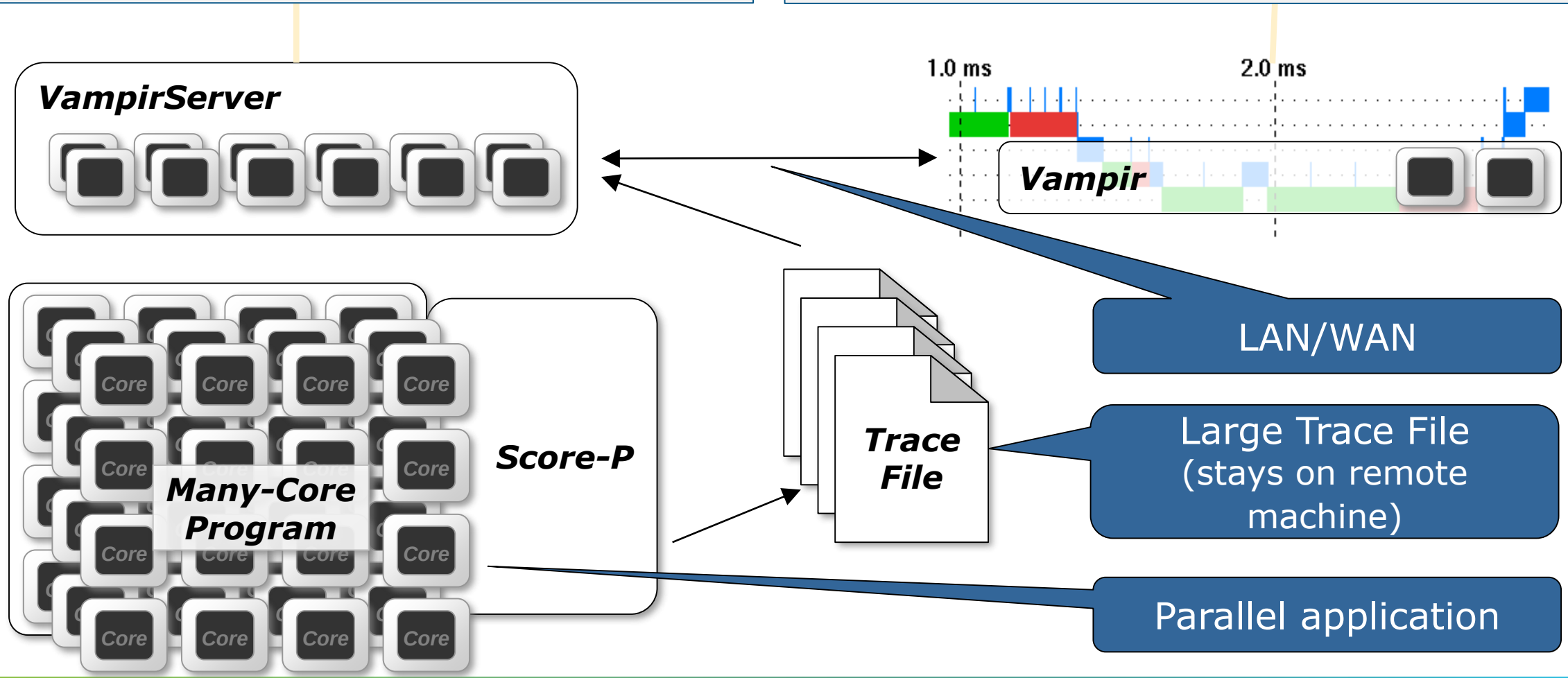
```
% vampir
```



Visualization Modes (2)
On local machine with remote VampirServer

```
% vampirserver start
```

```
% vampir
```



Main Performance Charts of Vampir

Timeline Charts


Master Timeline ➡ *all threads' activities*


Process Timeline ➡ *single thread's activities*


Summary Timeline ➡ *all threads' function call statistics*


Performance Radar ➡ *all threads' performance metrics*


Counter Data Timeline ➡ *single threads' performance metrics*


I/O Timeline ➡ *all threads' I/O activities*

Summary Charts


Function Summary


Message Summary

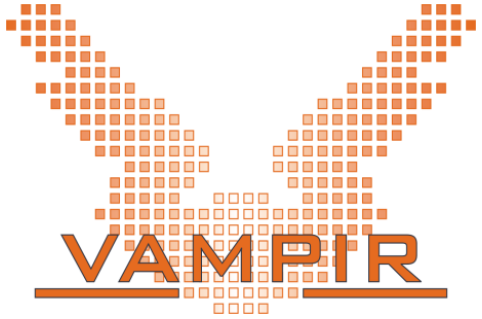

I/O Summary


Process Summary


Communication Matrix View


Call Tree

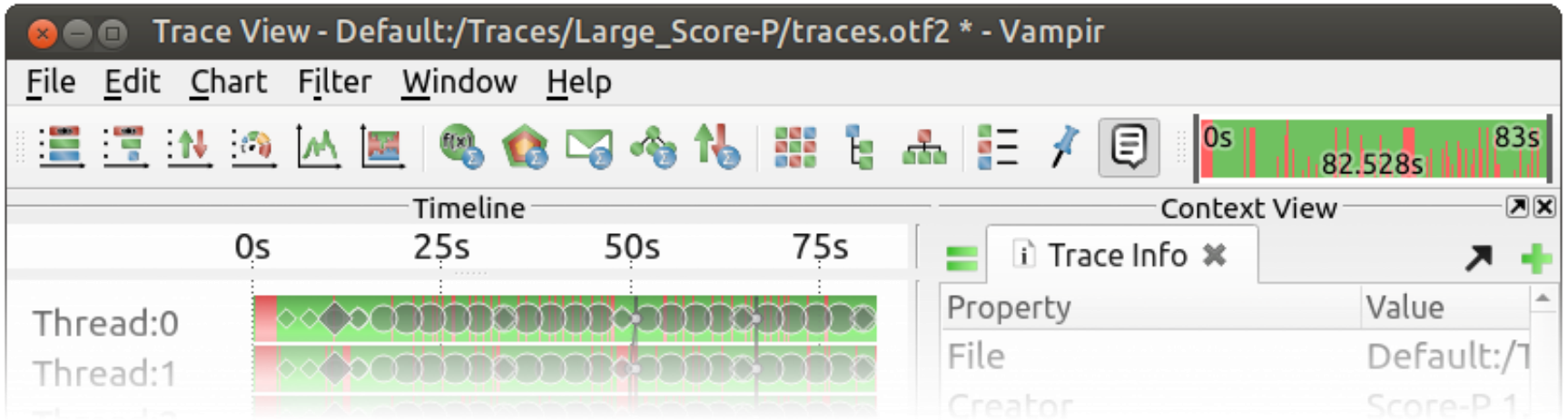
New Features in Vampir 10



What's New in Vampir 10
(released in Nov '21)

What's New Dialog when starting new version the first time

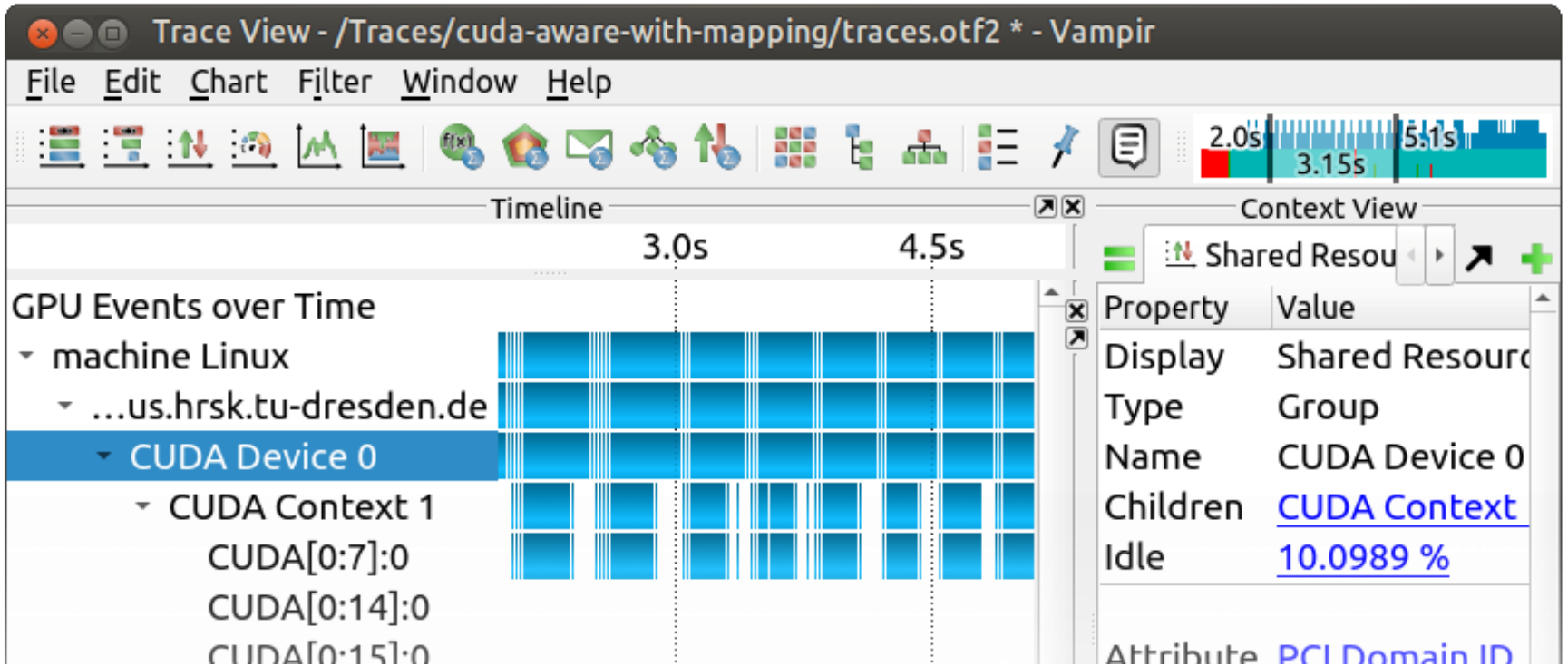
New Chart Icons



What's New in Vampir 10
(released in Nov '21)

Support new OTF2 3.0 features
- Accelerator devices and contexts
- Non-blocking collective operations

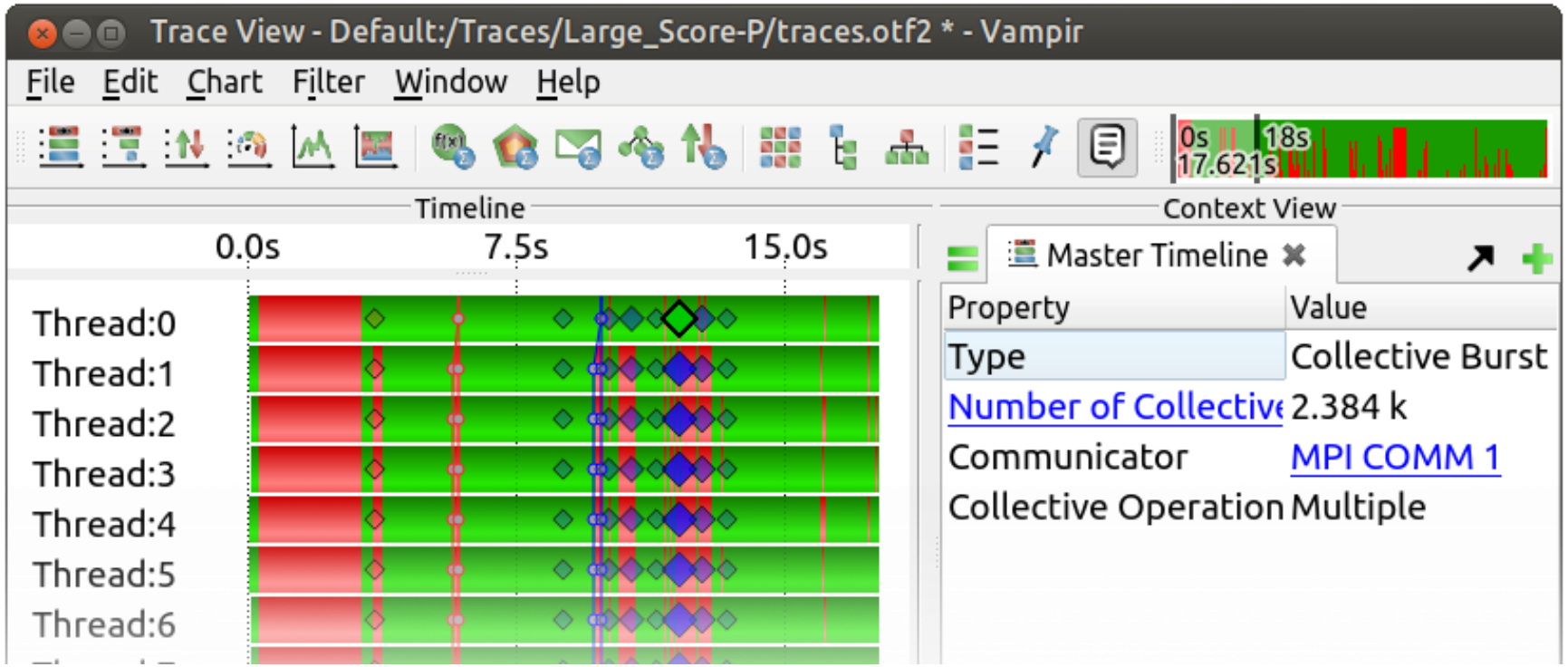
No Score-P release yet



What's New in Vampir 10
(released in Nov '21)

Collective Operation Bursts

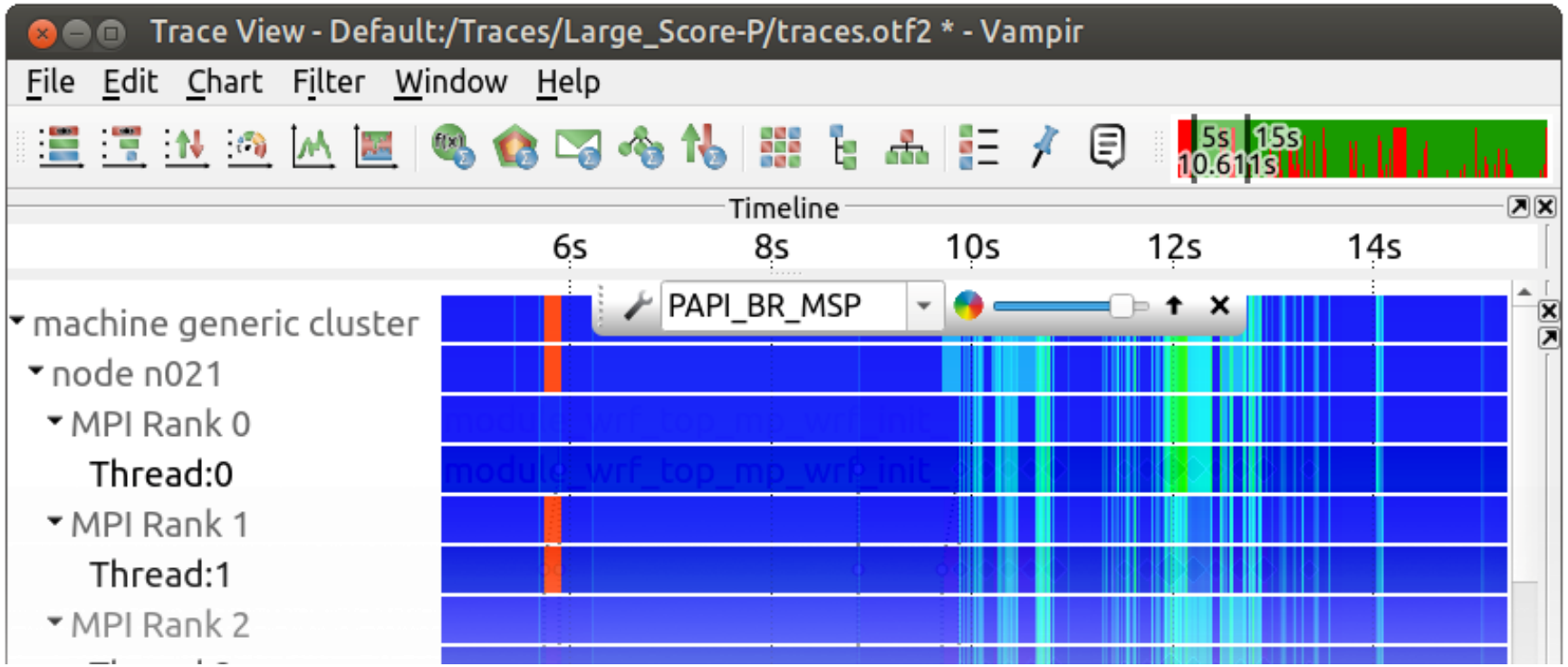
- Collated into bursts, when too many of them occur in a specific interval
- Collectives also visible if beginning is not in the zoom interval



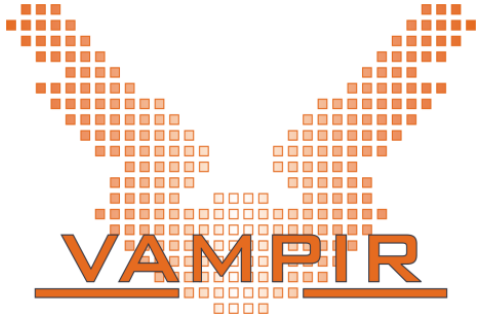
What's New in Vampir 10
(released in Nov '21)

Summarizations in the Performance Overlay

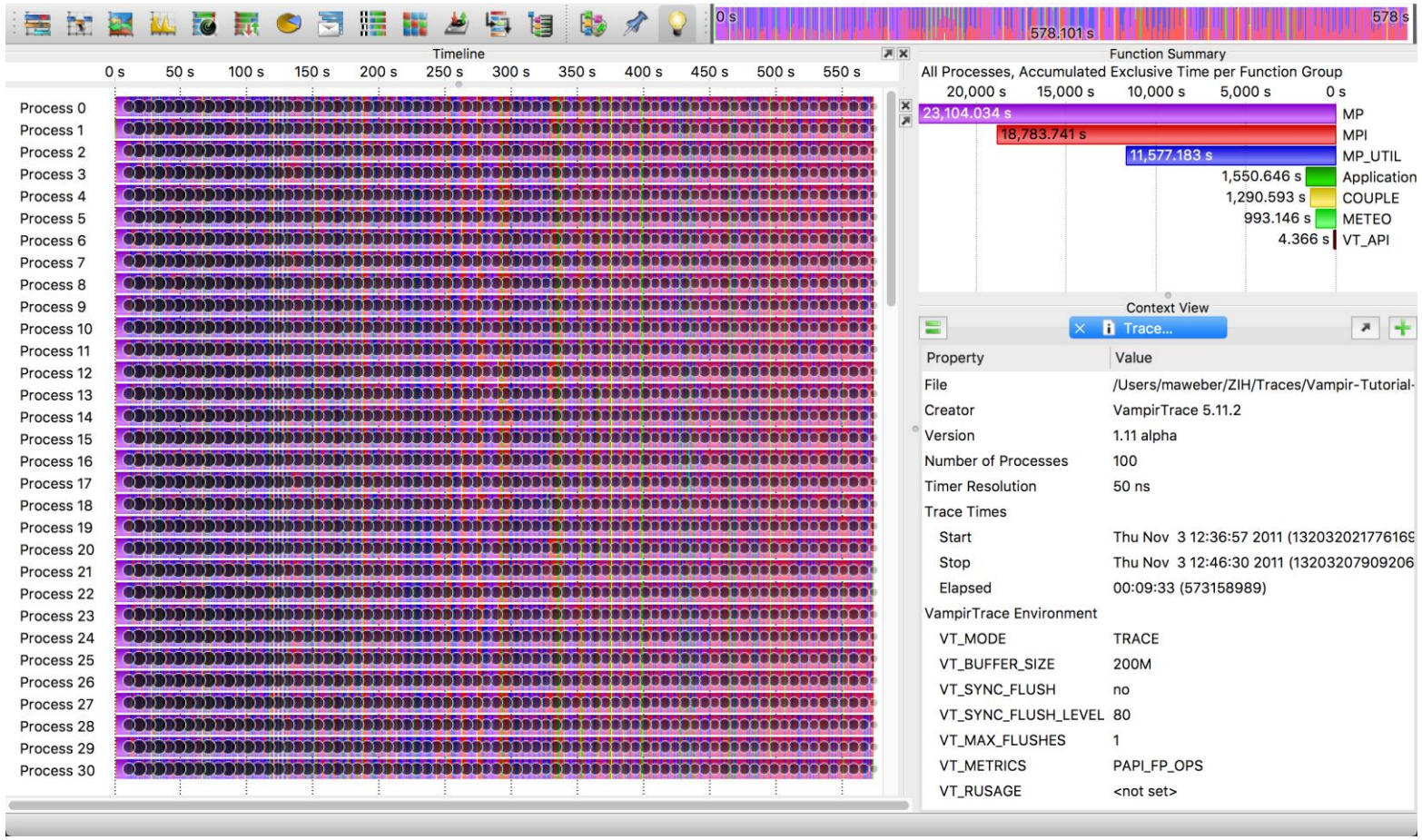
- Values are summarized into their parent groups



Vampir Case Study: Analyzing Load Imbalance in COSMO-SPECS

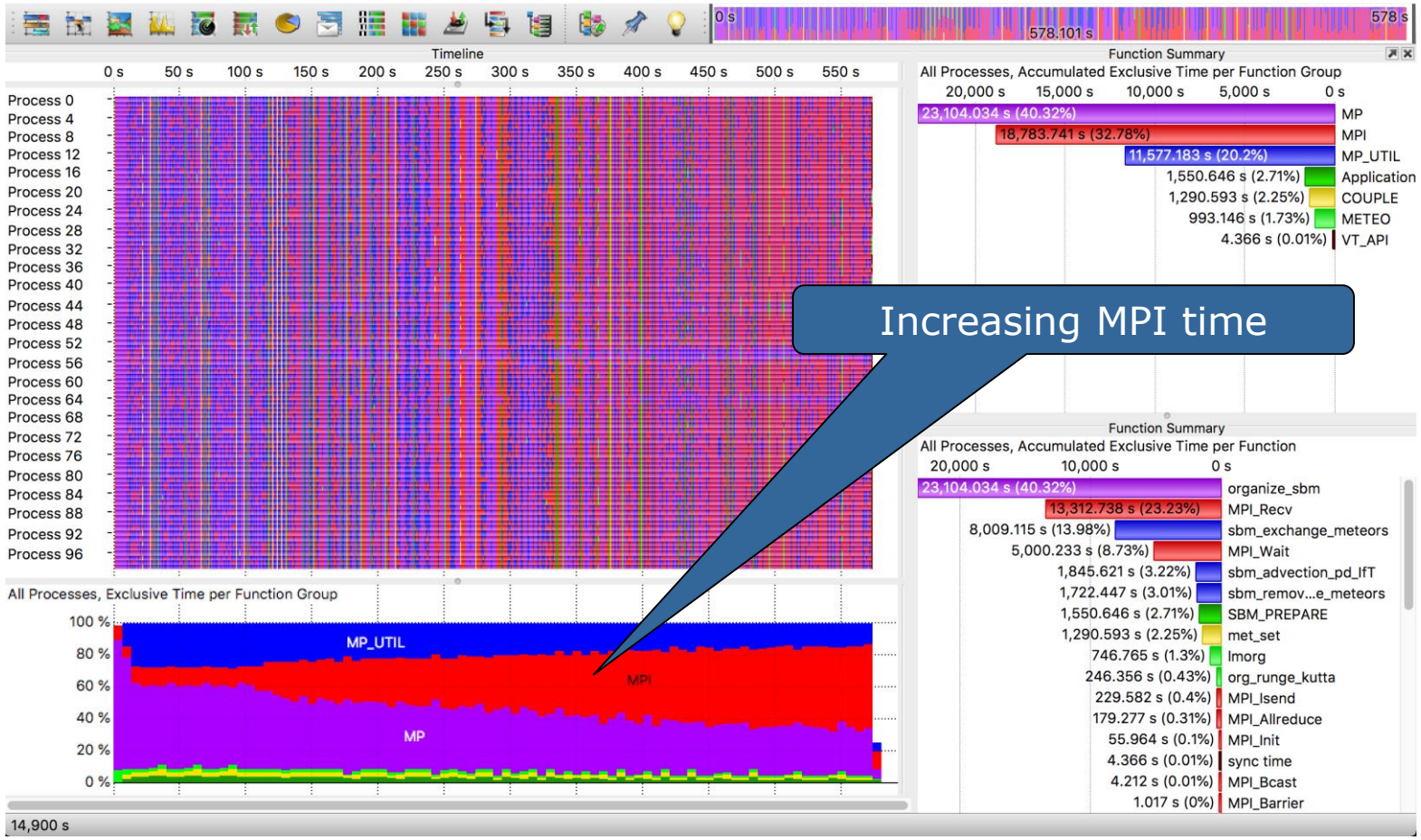


COSMO-SPECS



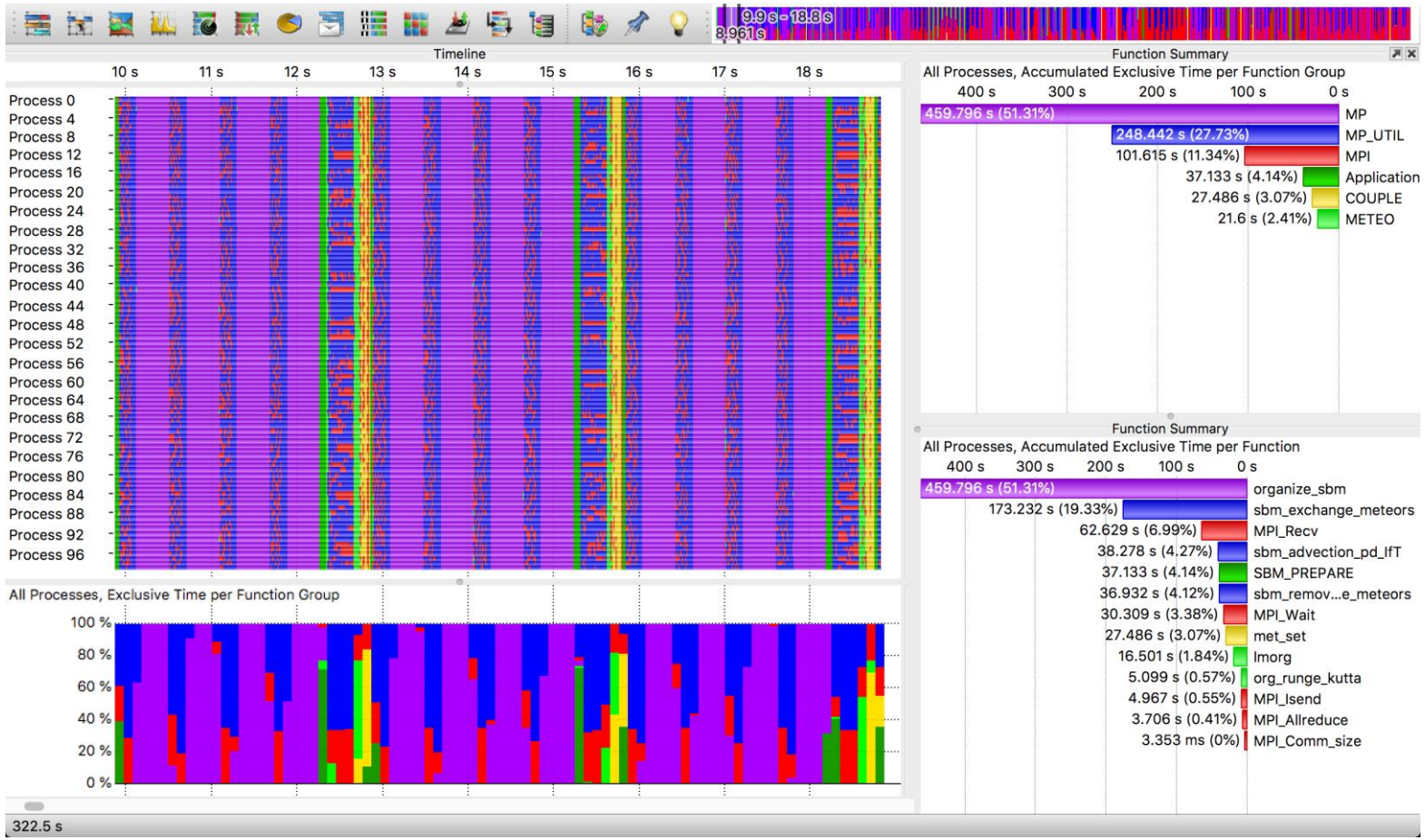
- Weather forecast code COSMO-SPECS
- Run with 100 processes
- COSMO: weather model (METEO group)
- SPECS: microphysics for accurate cloud calculation (MP and MP_UTIL group)
- Coupling of both models done in COUPLE group

COSMO-SPECS



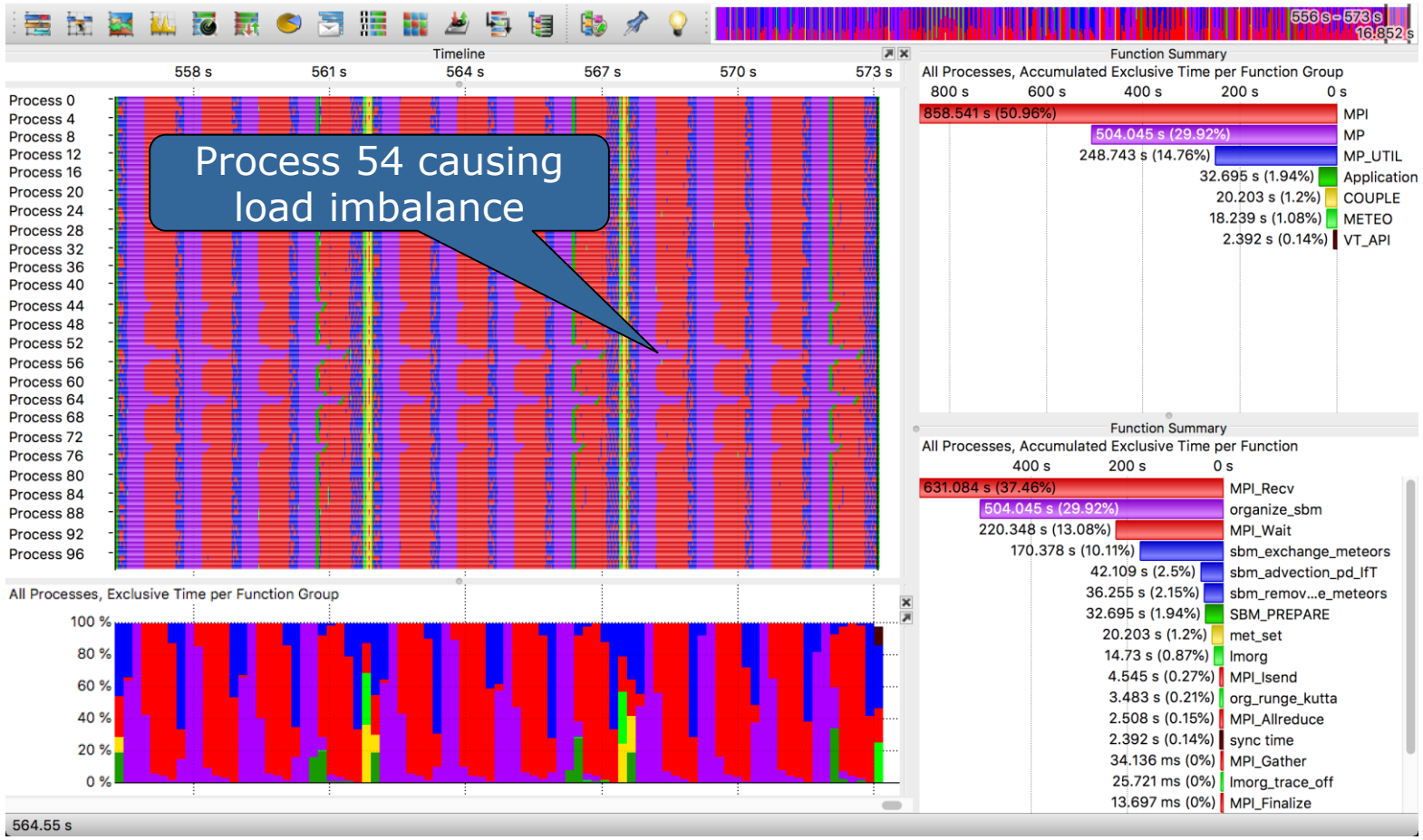
- Compared to METEO, MP and MP_UTIL are very compute intensive, however this is due to more complex calculations and no performance issue
- Problem: >32% of time spent in MPI
- MPI runtime share increases throughout the application run

COSMO-SPECS



- Zoom into the first three iterations
- MP/MP_UTIL perform four sub-steps in one iteration
- Low MPI time share
- Everything is balanced and looks okay

COSMO-SPECS

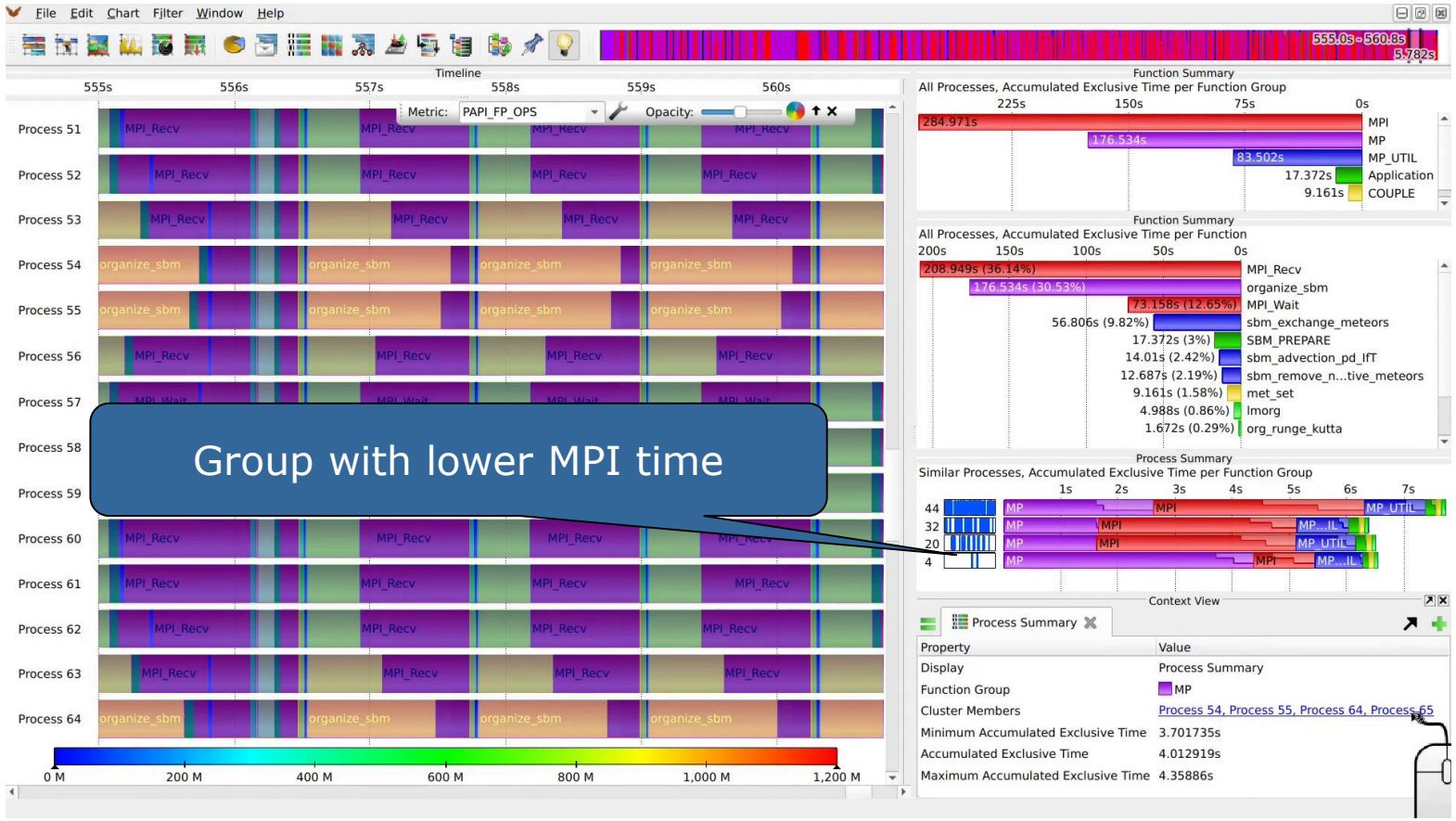


- Zoom into the last three iterations
- Very high MPI time share (>50%)
- Large load imbalance caused by MP functions around **`Process 54`** and **`Process 64`**

COSMO-SPECS

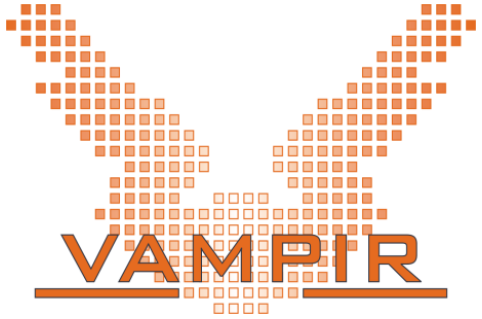
- **`PAPI_FP_OPS`** counter showing higher FLOPs rates on processes causing the imbalance
- Reason for imbalance: Static grid used for distribution of processes. Depending on the weather, expensive cloud computations (MP group) may be only necessary on some processes

COSMO-SPECS

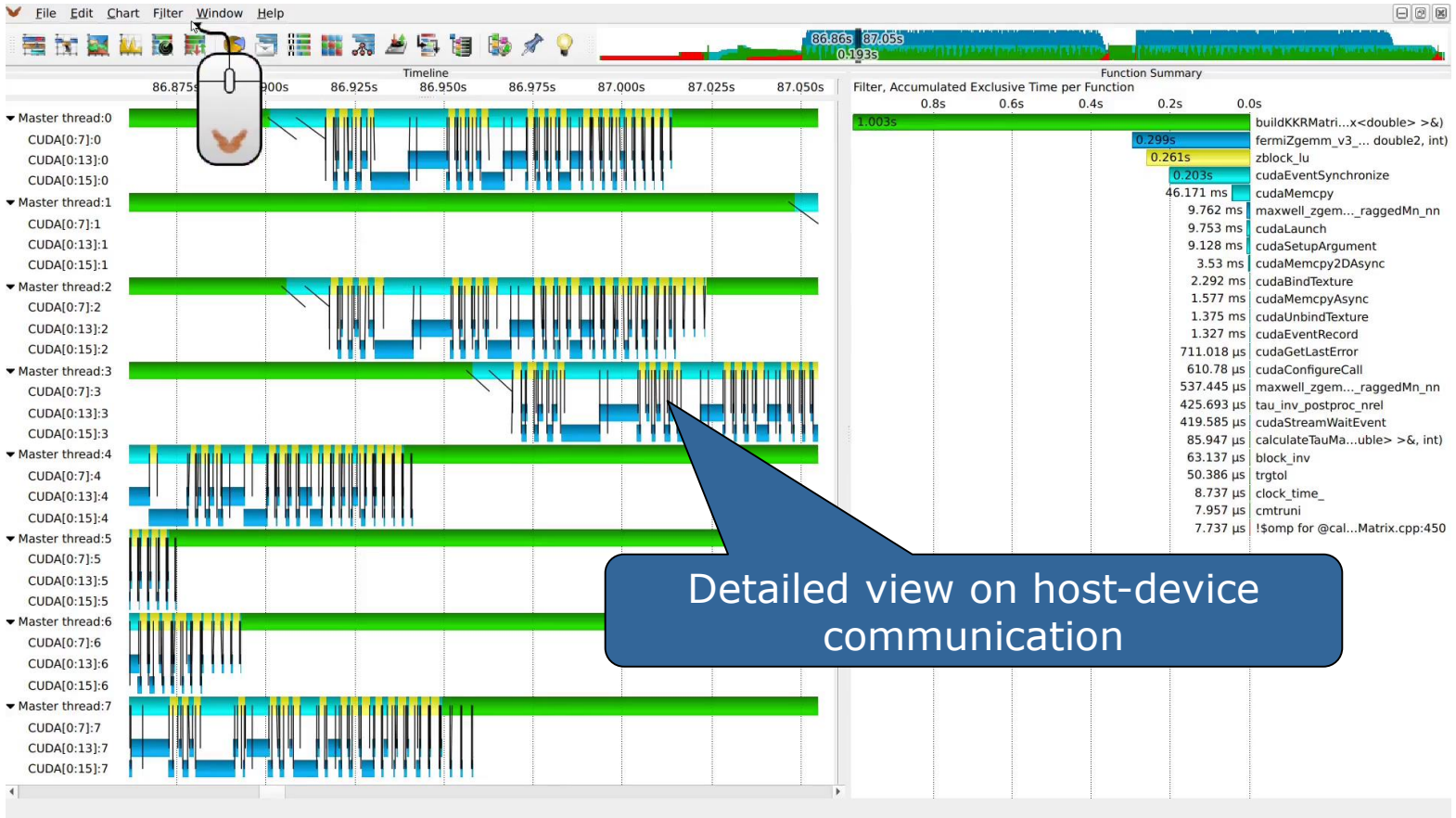


- Process Summary helps finding outliers
- Groups processes by their behavior (similar call/duration profile)
- Number of expected groups is variable
- In this case 4 yields the best results

Vampir Showcase: Analyzing CUDA Applications



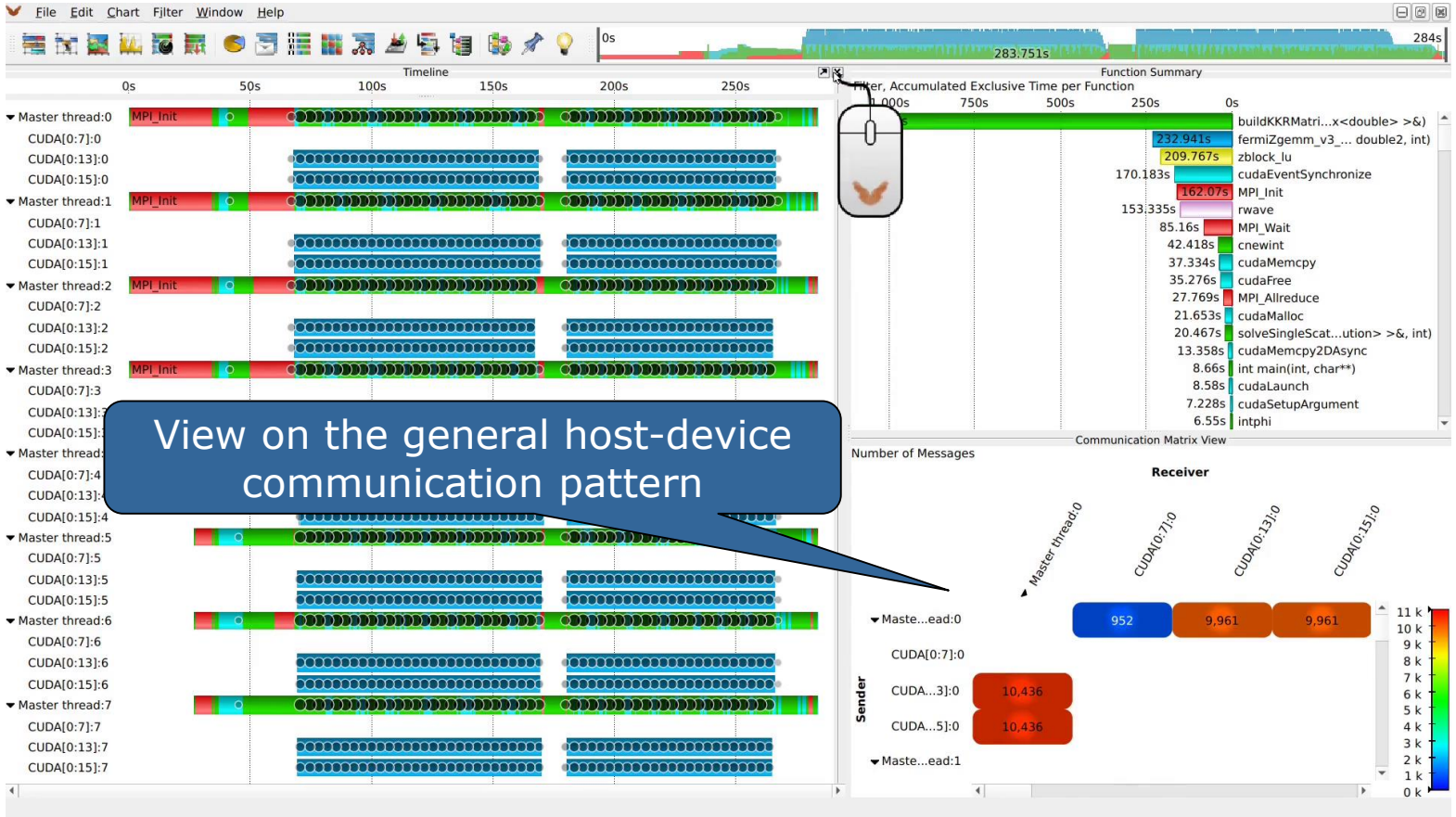
CUDA Application



- Material science code LSMS
- CUDA is utilized for heavy computations

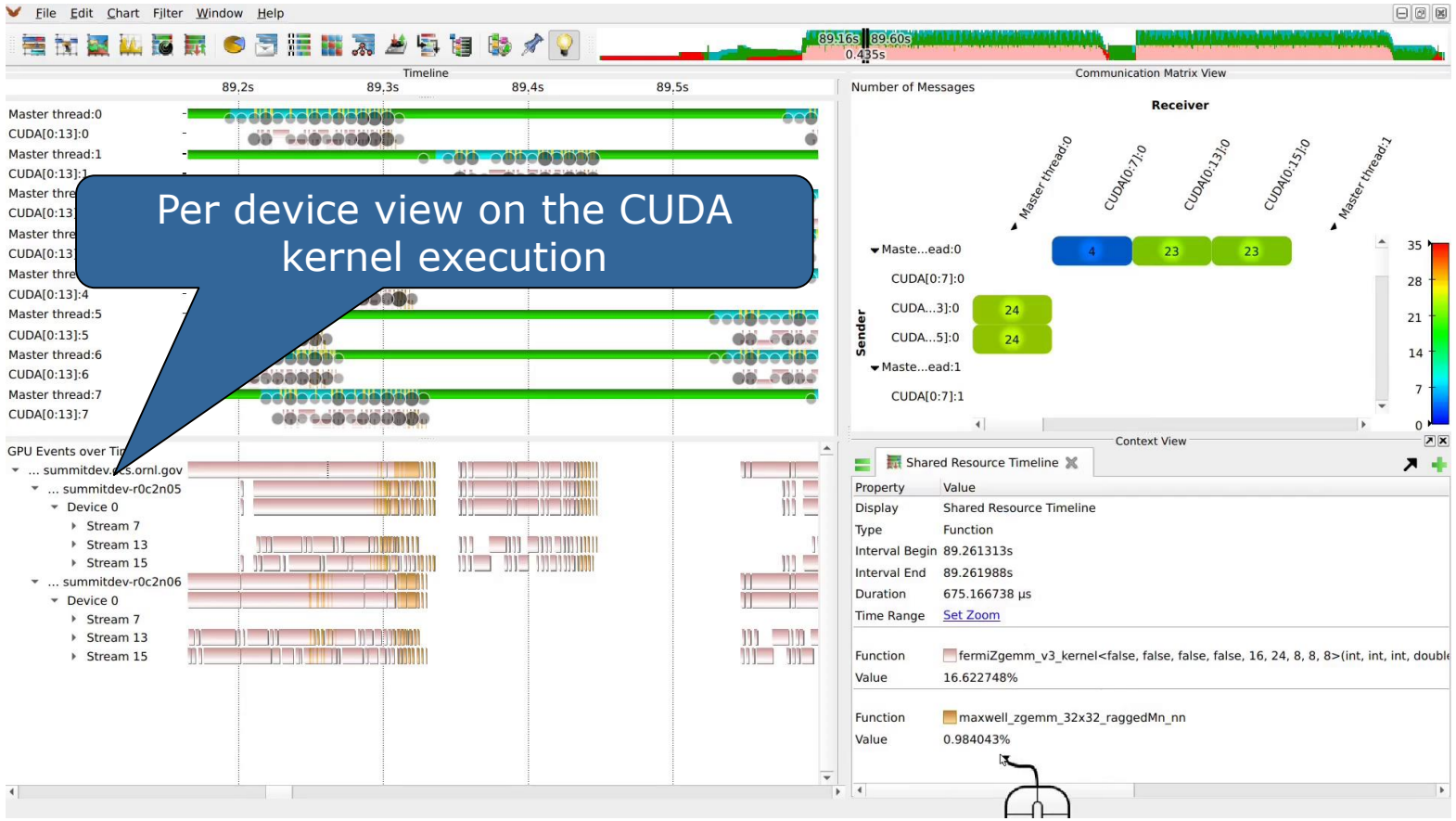
- CUDA streams are child's of the owning Process
- Allows an in-depth analysis of host-device communication

CUDA Application



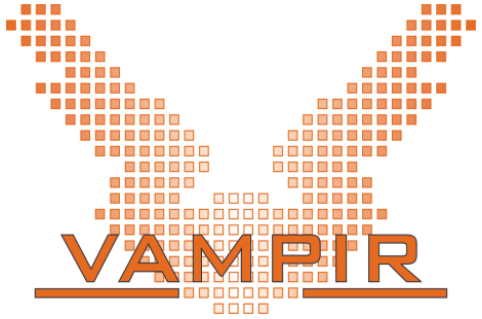
- Communication Matrix best for analyzing the general communication pattern
- Expectation: balanced communication, represented by a symmetric matrix
- Problem: communication with stream 7 is different

CUDA Application

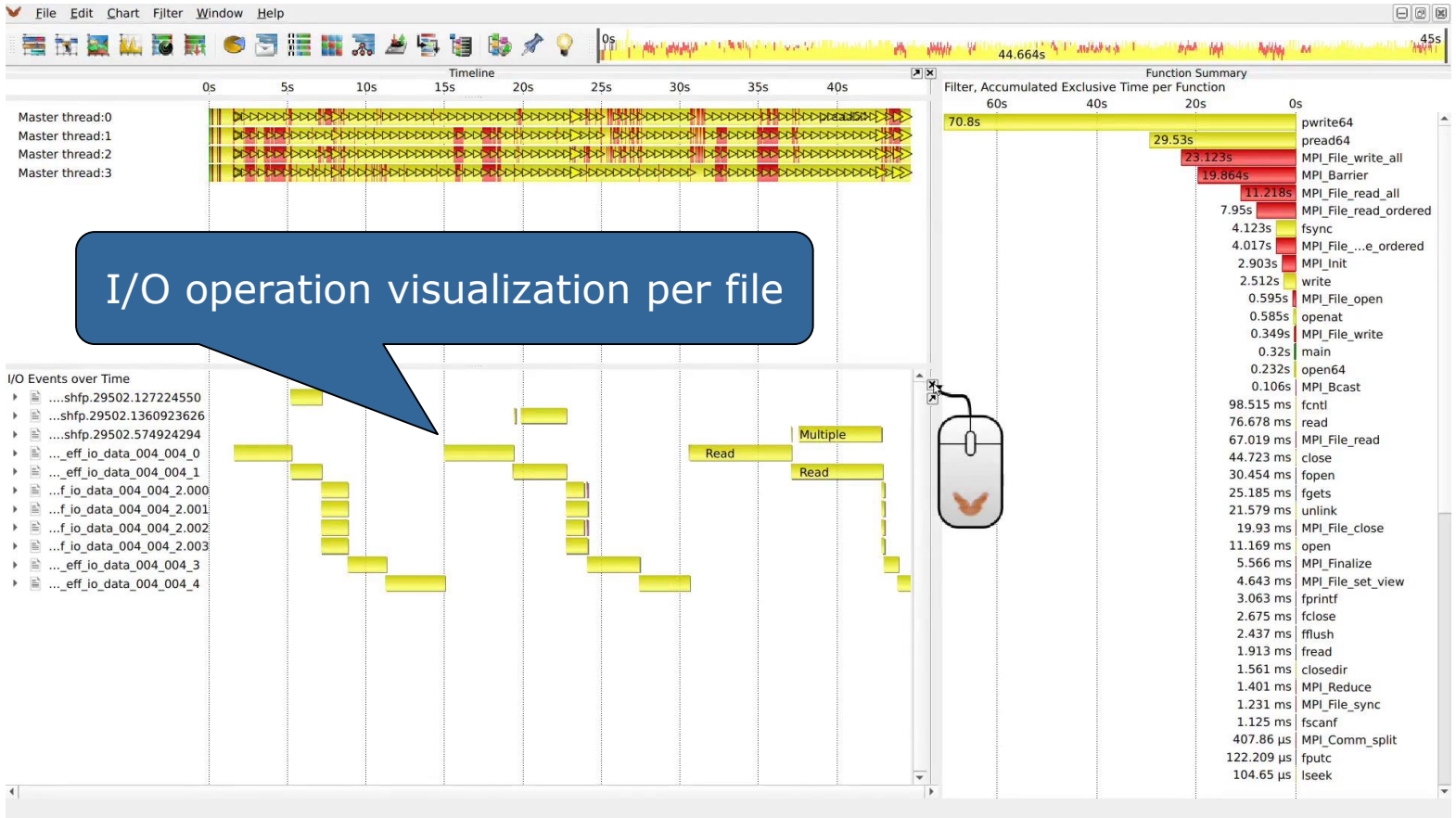


- Shared Resource Timeline offers a per device view on kernel executions
- Best suited for analyzing multi GPU per node scenarios
- Allows a dedicated analysis of kernel execution patterns
- Yields insights of the actual hardware usage

Vampir Showcase:
Analyzing Multilayer File I/O Applications

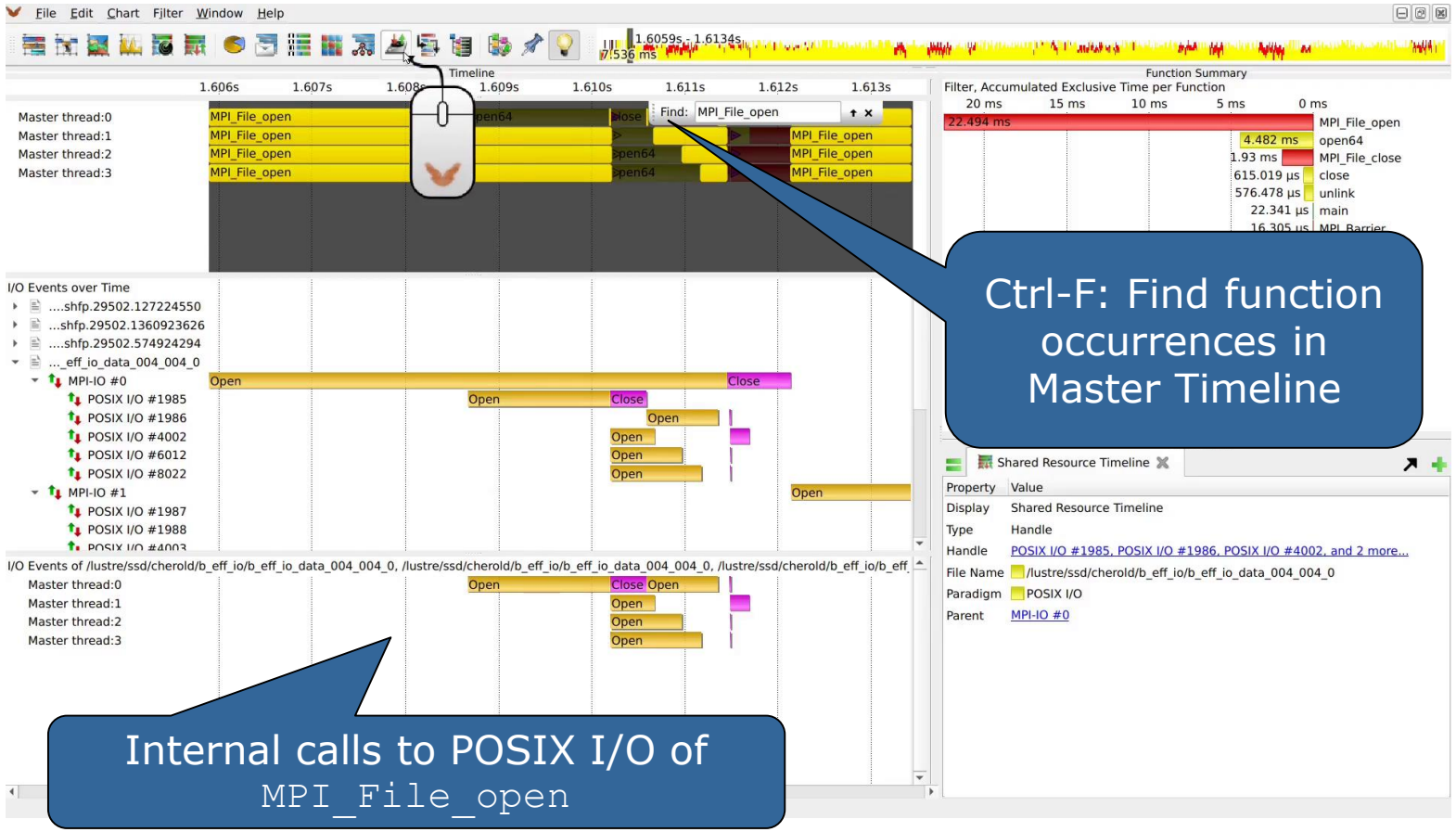


Multilayer File I/O Application



- IO bandwidth benchmark b_eff_io
- Measures achievable I/O bandwidth of parallel MPI-I/O applications
- Shared Resource Timeline offers a per file and per thread view on File I/O operations

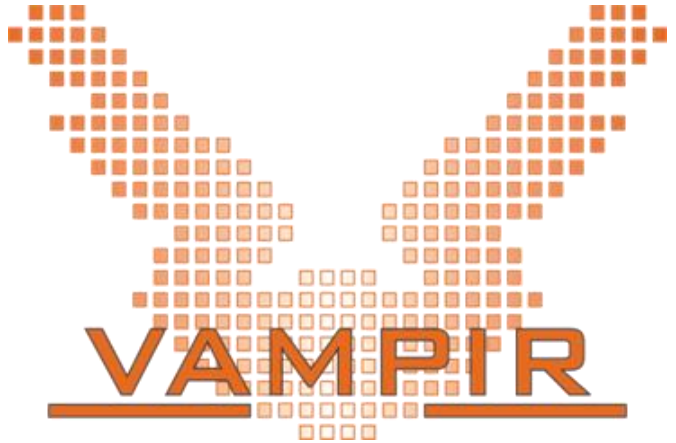
Multilayer File I/O Application



- Visualization of I/O on multiple layers (MPI & POSIX)
- Example: behavior of **`MPI_File_open`**
- Internally uses POSIX **`open`** for opening the actual file on disk
- Multiple consecutive calls to **`open`** and **`close`** on master rank

Summary

- Vampir
  - Interactive trace visualization and analysis of:
    - MPI, OpenMP, CUDA applications
    - File I/O
    - Hardware performance counters
    - (Collective) communication
  - Intuitive browsing and zooming
  - Available for Linux, Windows, and macOS
- VampirServer
  - Scalable to large trace data sizes (20 TiByte)
  - Scalable to high parallelism (200,000 processes)



Automatic trace analysis with the Scalasca Trace Tools

Markus Geimer
Jülich Supercomputing Centre

trace tools
scalasca

Scalasca Trace Tools

DOI 10.5281/zenodo.4700519

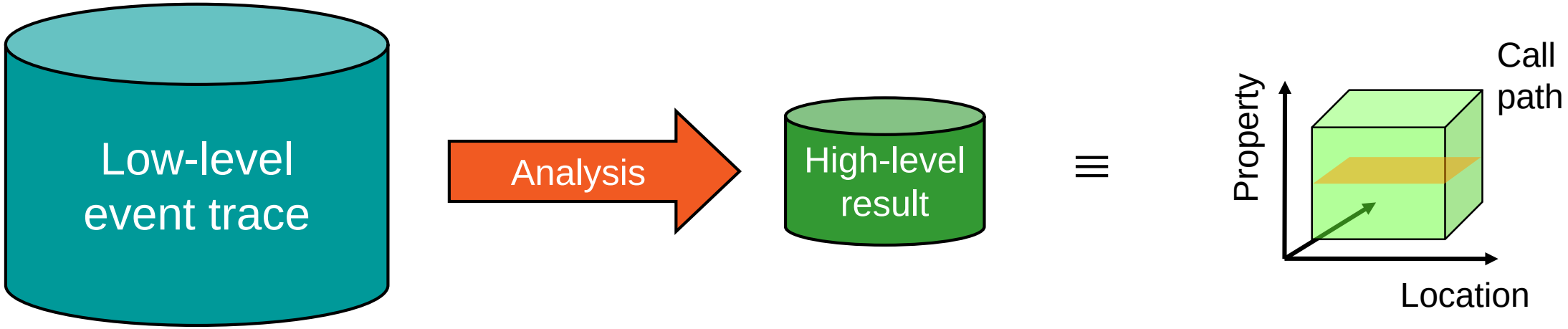
- **Scalable trace-based** performance analysis toolset for the most popular parallel programming paradigms
  - Current focus: MPI, OpenMP, and (to a limited extend) POSIX threads
  - Analysis of traces including only host-side events from applications using CUDA, OpenCL, or OpenACC (also in combination with MPI and/or OpenMP) is possible, but results need to be interpreted with some care

- Specifically targeting large-scale parallel applications
  - Demonstrated scalability up to 1.8 million parallel threads
  - Of course also works at small/medium scale

- Latest release:
  - Scalasca Trace Tools v2.6 (April 2021), coordinated with Score-P v7.0

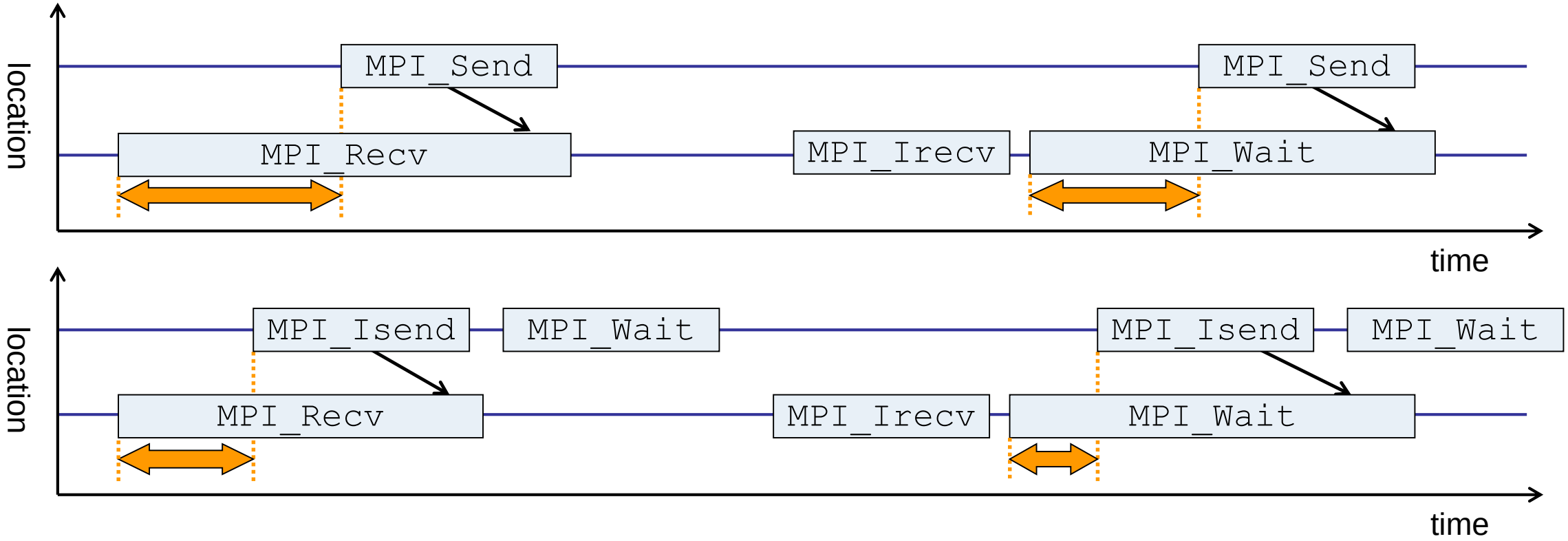
Automatic trace analysis

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



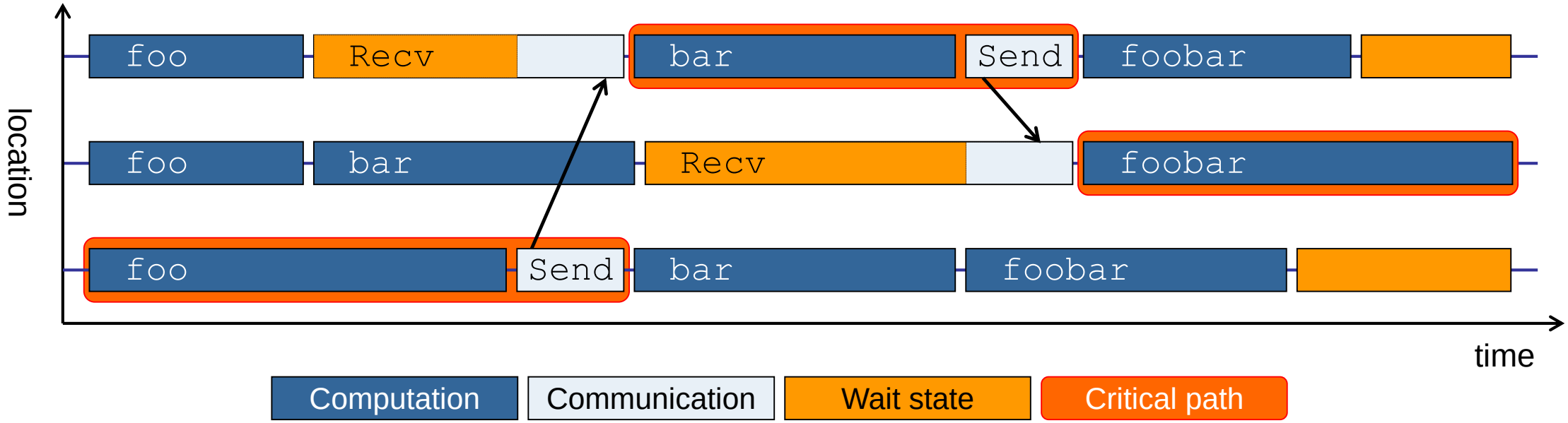
  - Guaranteed to cover the entire event trace
  - Quicker than manual/visual trace analysis
  - Parallel replay analysis exploits available memory & processors to deliver scalability

Example: "*Late Sender*" wait state



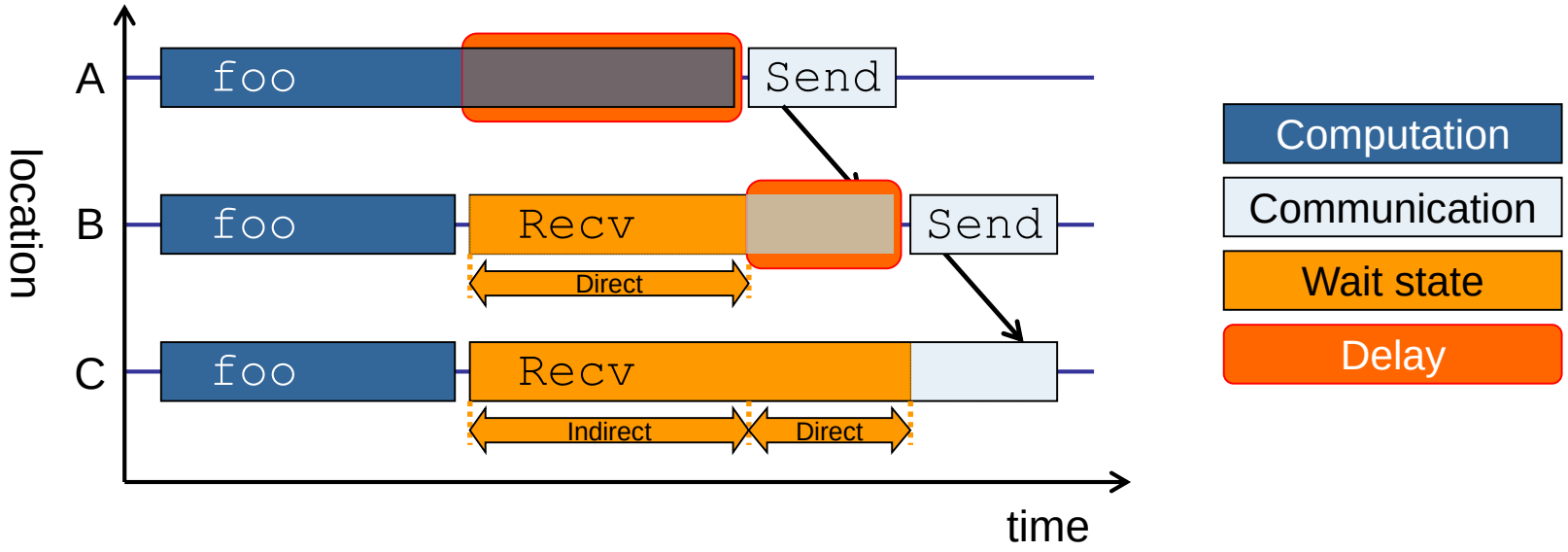
- Waiting time caused by a blocking receive operation posted earlier than the corresponding send
- Applies to blocking as well as non-blocking communication

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*

Demo:
TeaLeaf case study

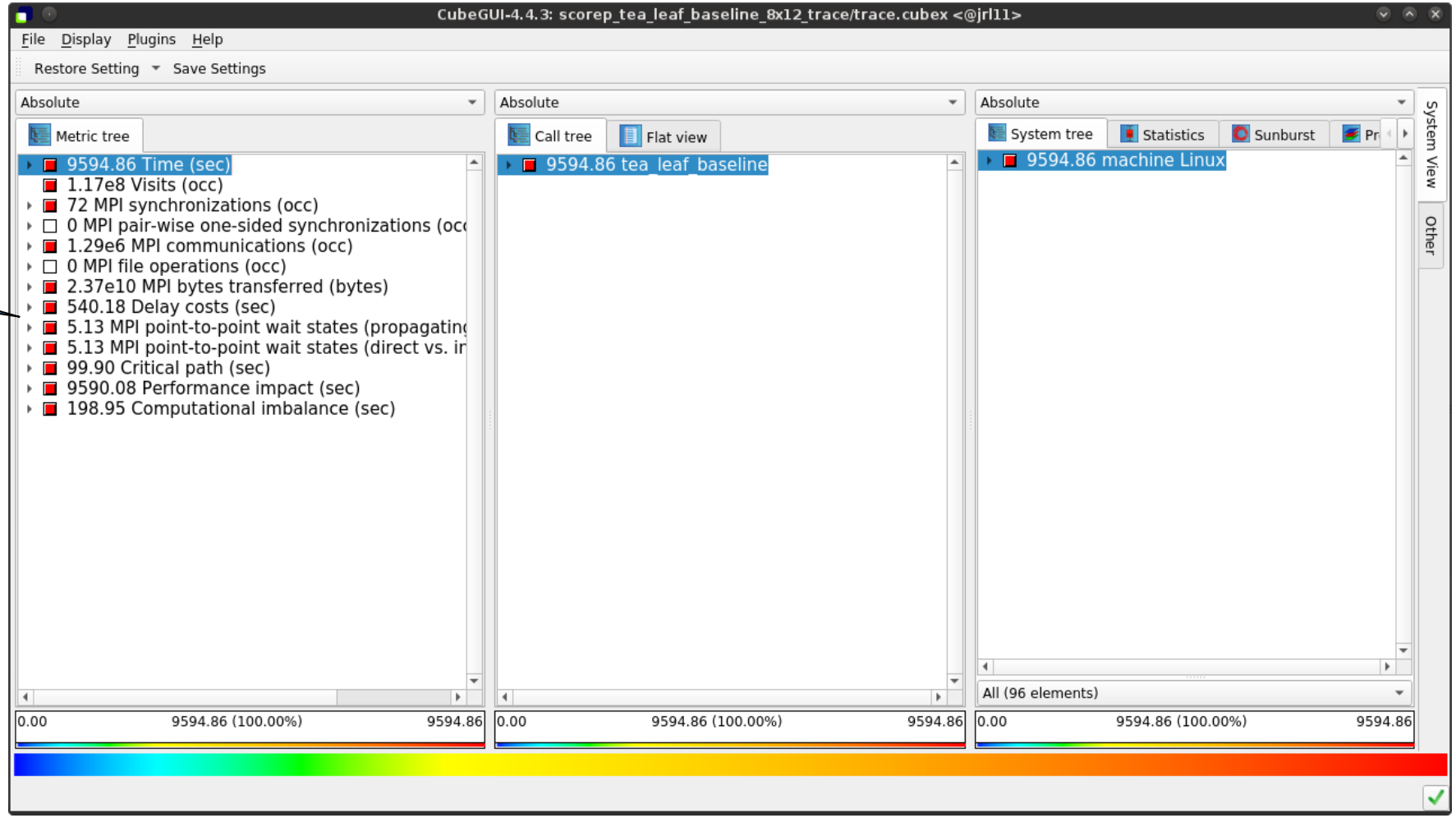
TeaLeaf case study (recap)

- HPC mini-app developed by the UK Mini-App Consortium
  - Solves the linear 2D heat conduction equation on a spatially decomposed regular grid using a 5 point stencil with implicit solvers
  - Part of the Mantevo 3.0 suite
  - Available on GitHub: https://uk-mac.github.io/TeaLeaf/
- Measurements of TeaLeaf reference v1.0 taken on Jureca cluster @ JSC
  - Using Intel 19.0.3 compilers, Intel MPI 2019.3, Score-P 5.0, and Scalasca 2.5
  - Run configuration
    - 8 MPI ranks with 12 OpenMP threads each
    - Distributed across 4 compute nodes (2 ranks per node)
    - Test problem "5": 4000 × 4000 cells, CG solver

```
% square scorep_tea_leaf_baseline_8x12_trace
INFO: Post-processing trace analysis report (scout.cubex)...
INFO: Displaying ./scorep_tea_leaf_baseline_8x12_trace/trace.cubex...
                          [GUI showing post-processed trace analysis report]
```

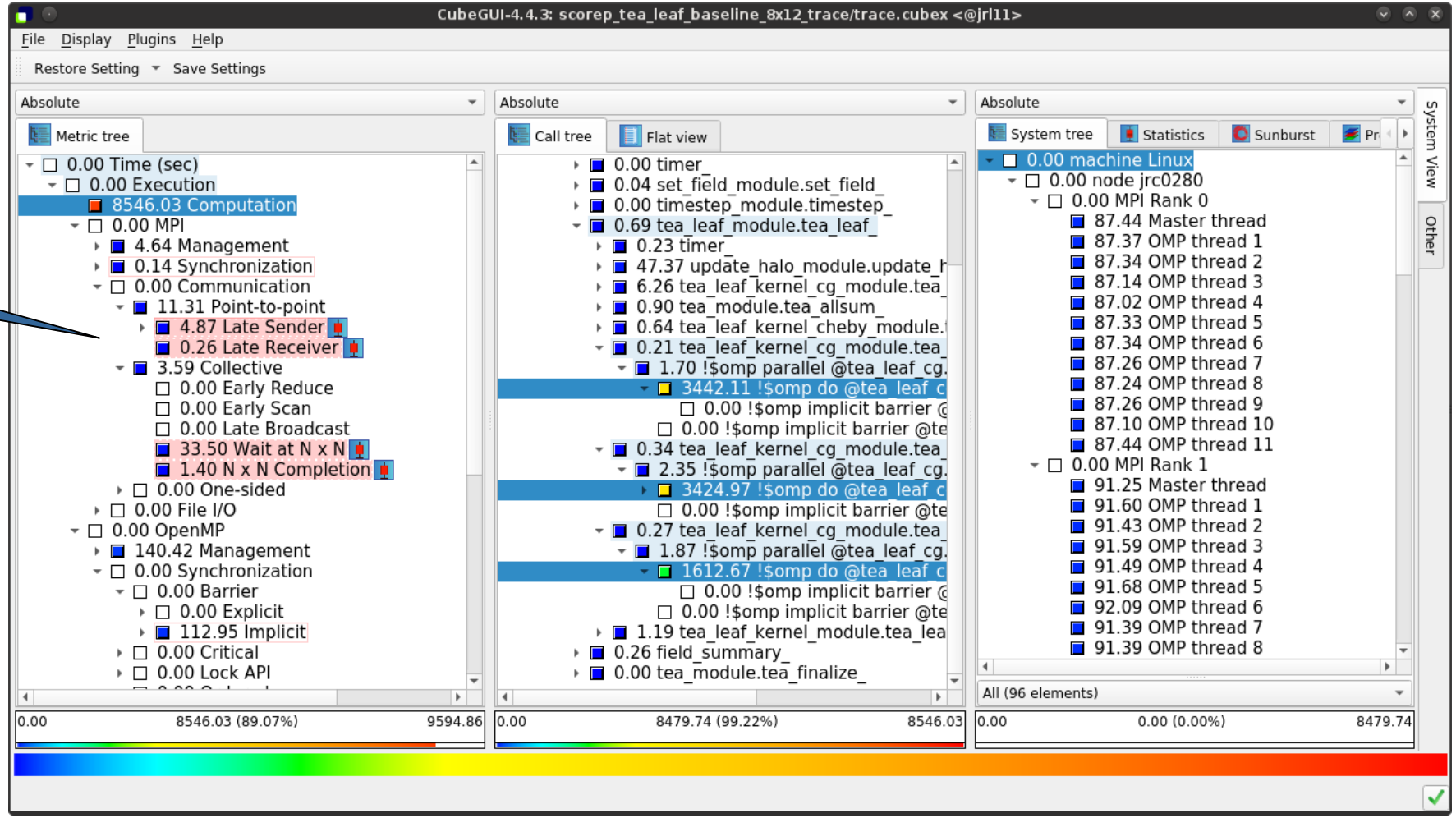
Scalasca analysis report exploration (opening view)

Additional top-level metrics produced by the trace analysis…



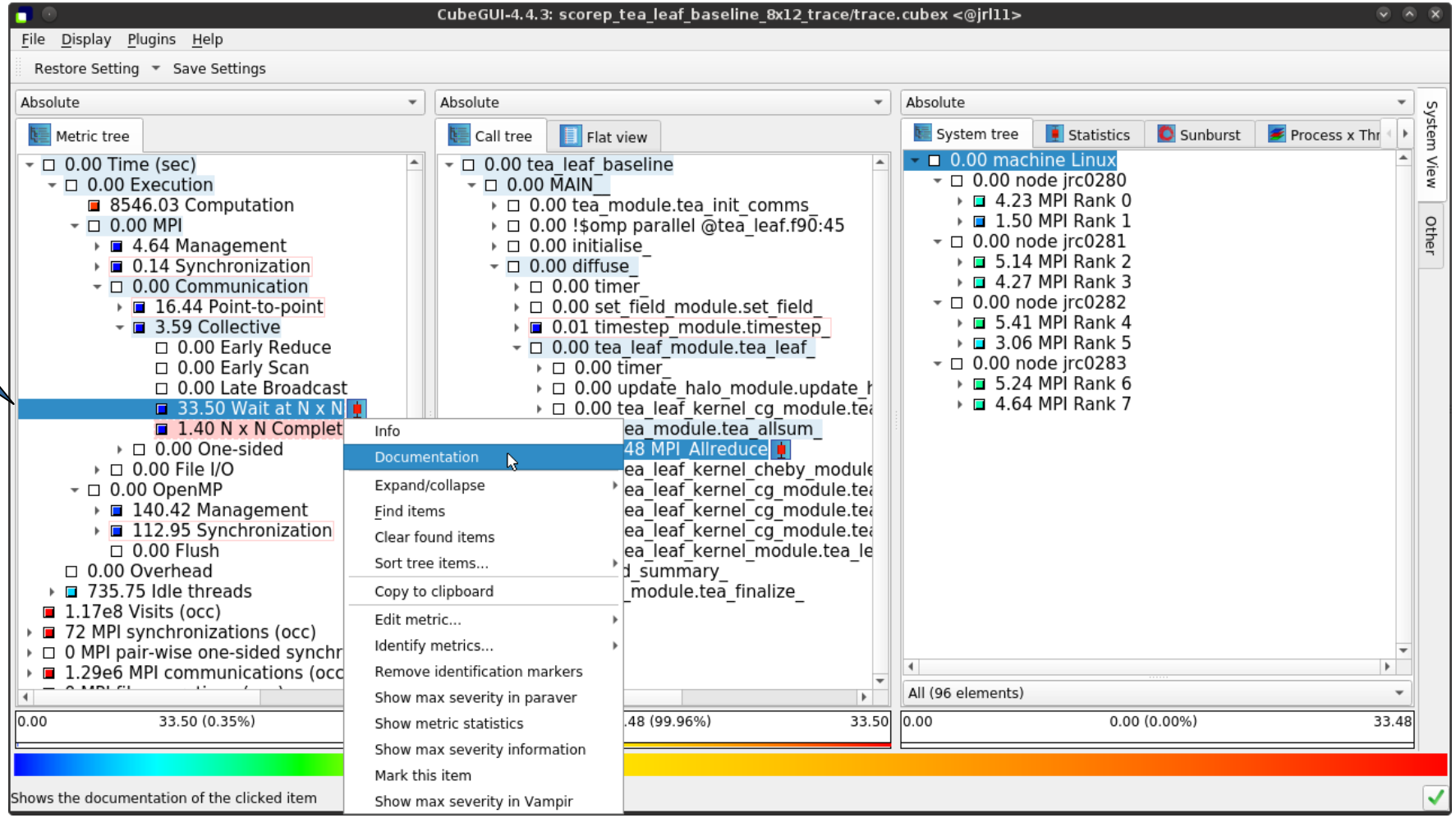
Scalasca wait-state metrics

…plus additional wait-state metrics as part of the “Time” hierarchy



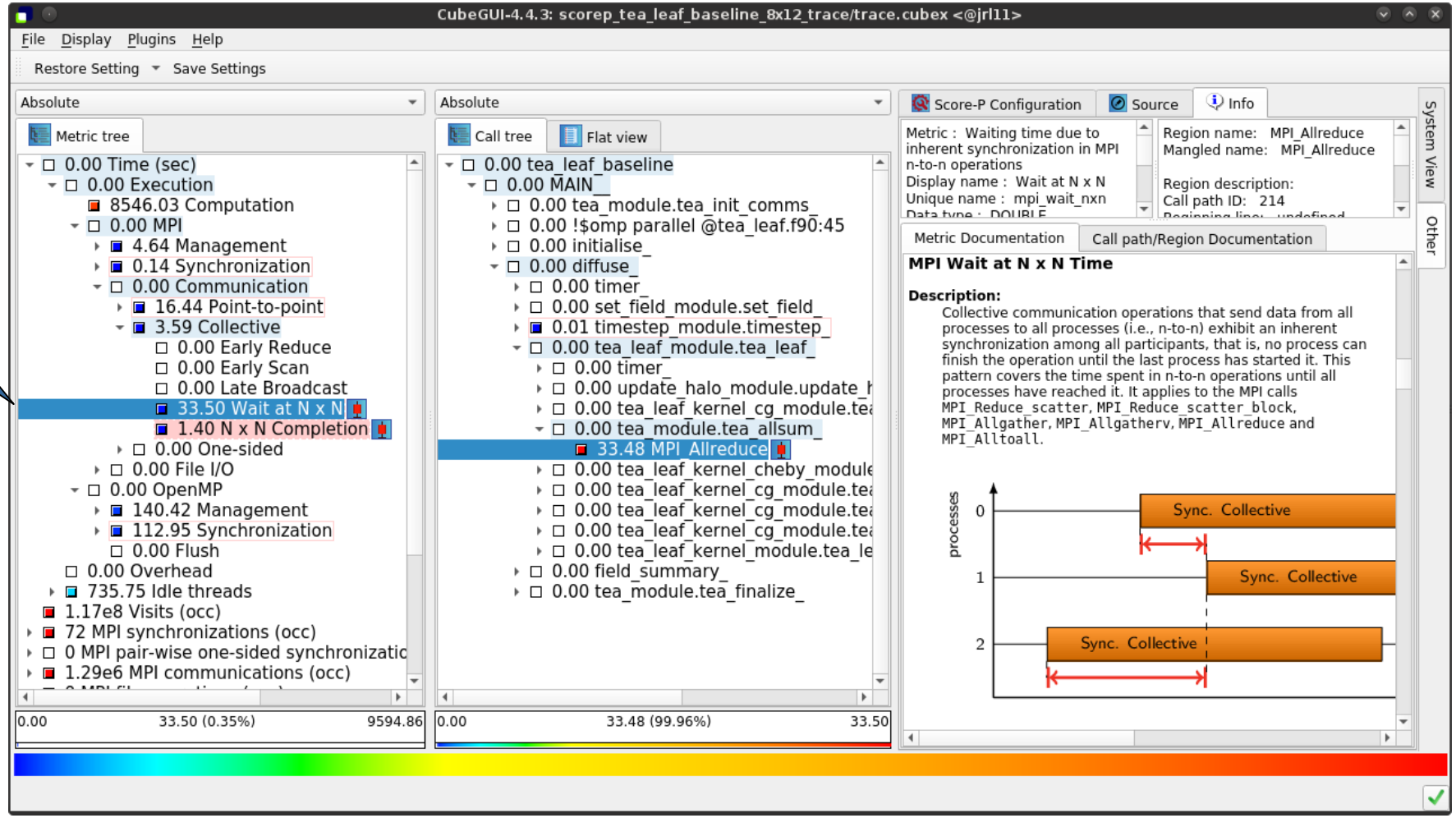
Online metric description

Access online metric description via context menu (right-click)



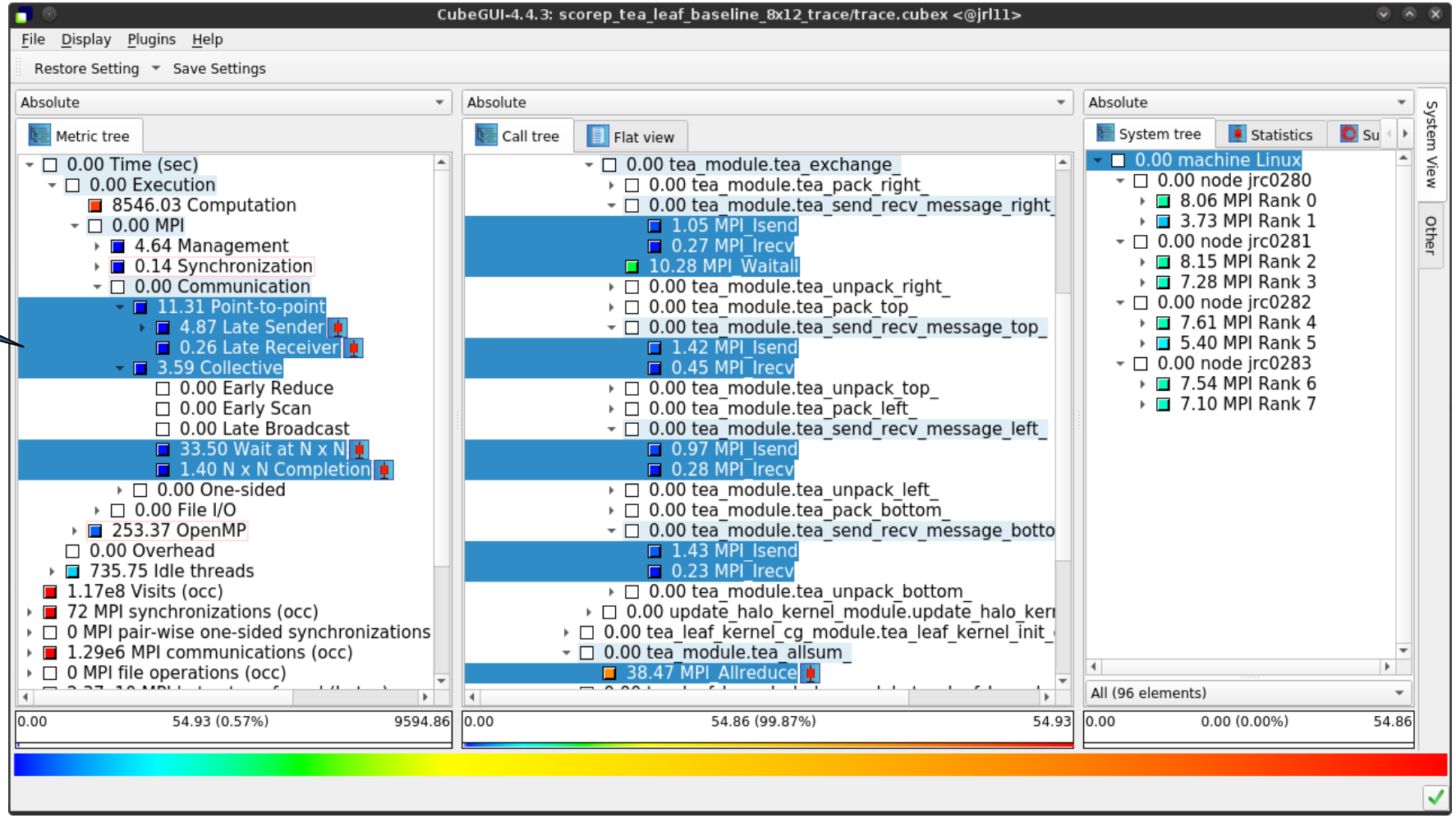
Online metric description (cont.)

Selection of different metric automatically updates description



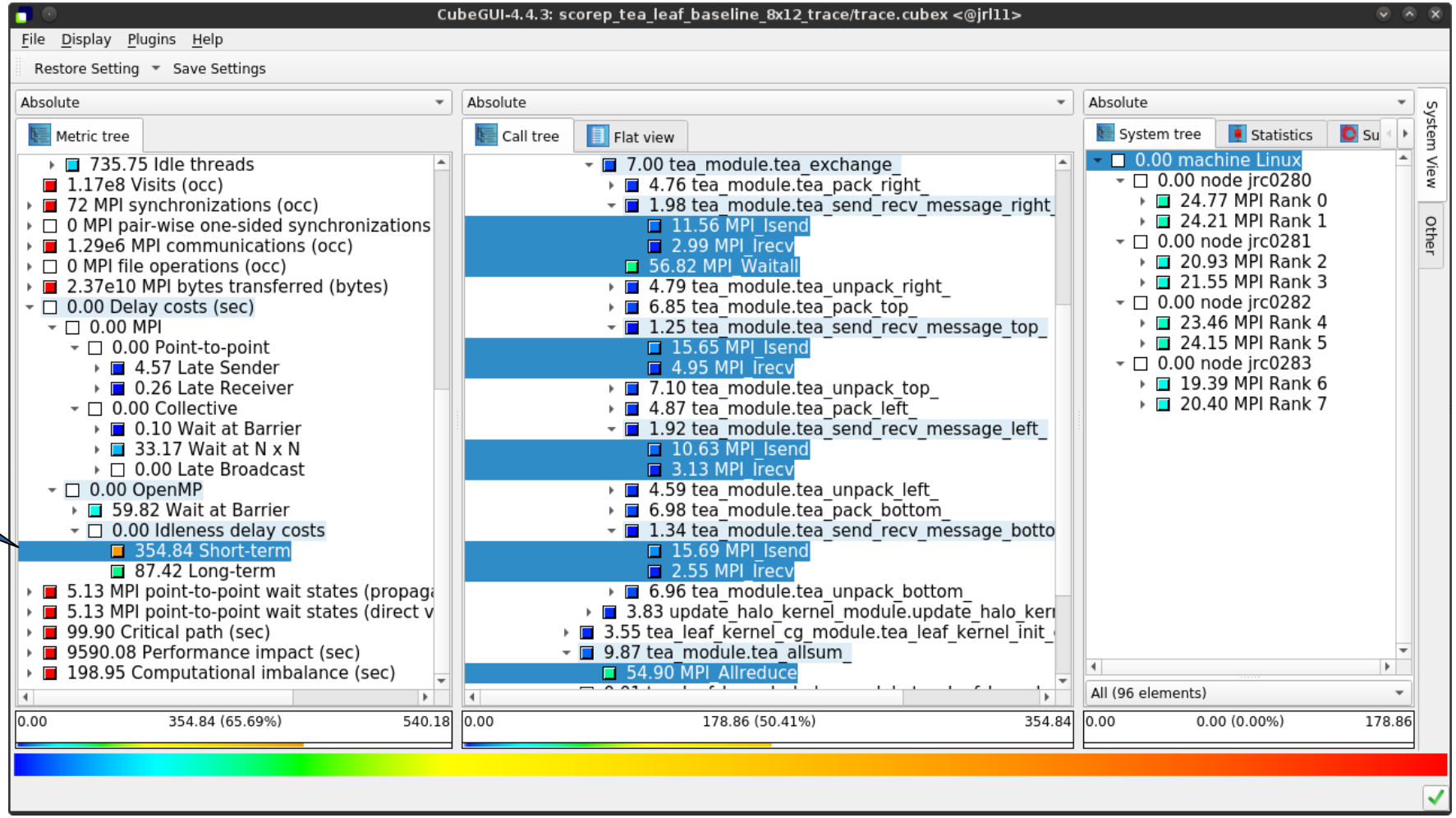
TeaLeaf Scalasca report analysis (I)

While MPI communication time and wait states are small (~0.6% of the total execution time)…



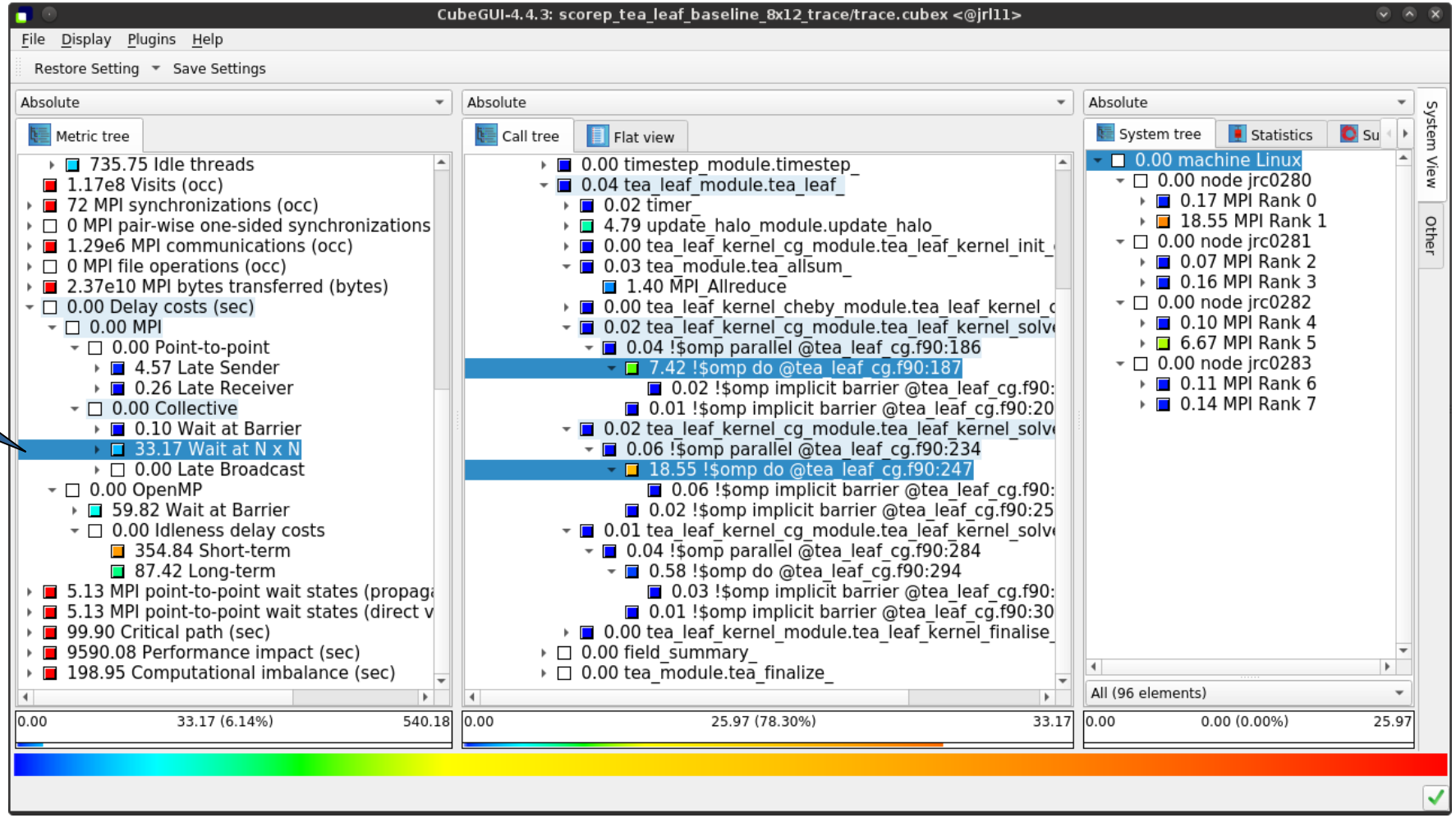
TeaLeaf Scalasca report analysis (II)

…they directly cause a significant amount of the OpenMP thread idleness



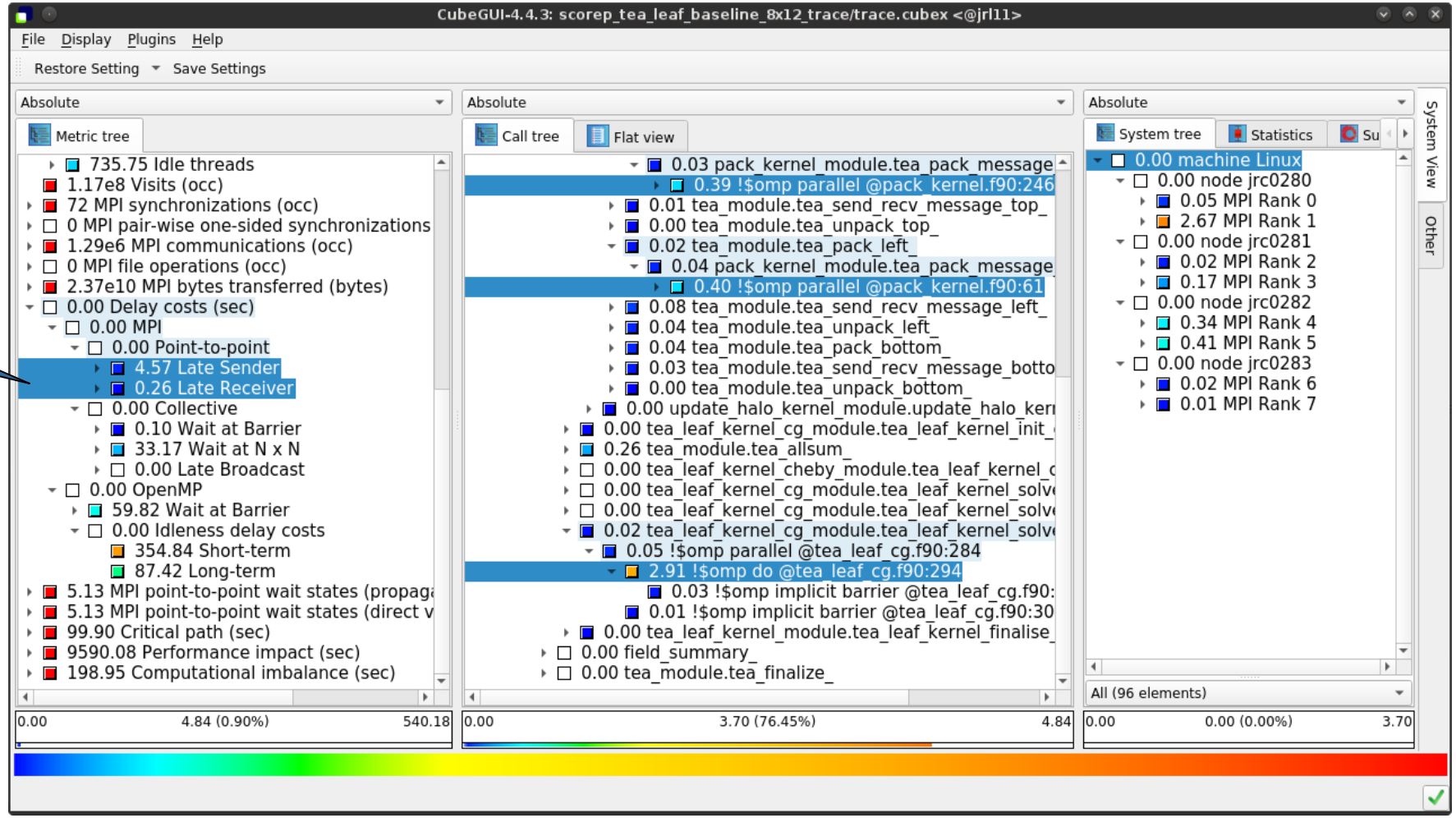
TeaLeaf Scalasca report analysis (III)

The “Wait at NxN” collective wait states are mostly caused by the first 2 OpenMP `do` loops of the solver (on ranks 5 & 1, resp.)…



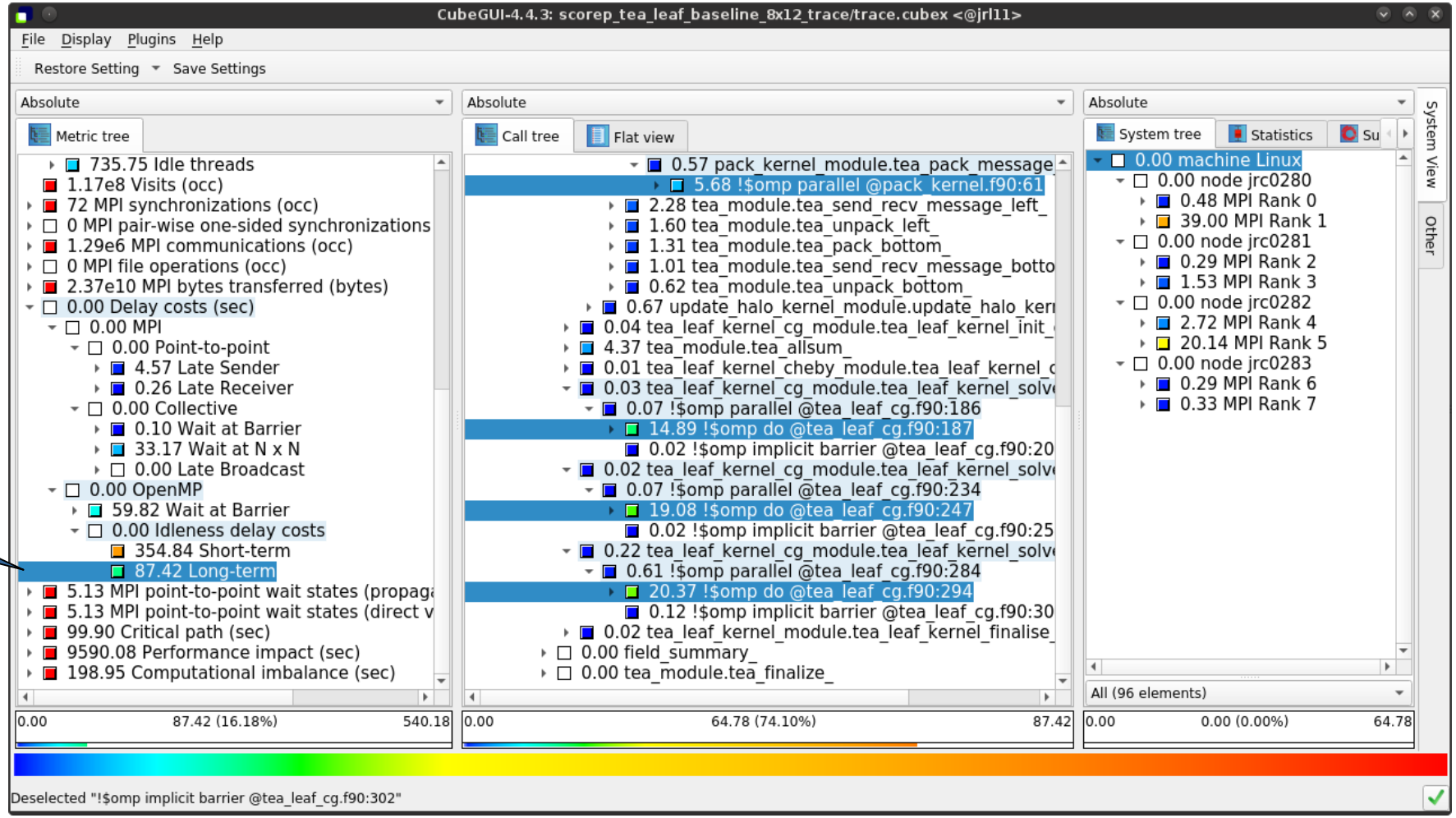
TeaLeaf Scalasca report analysis (IV)

…while the MPI point-to-point wait states are caused by the 3rd solver `do` loop (on rank 1) and two loops in the halo exchange



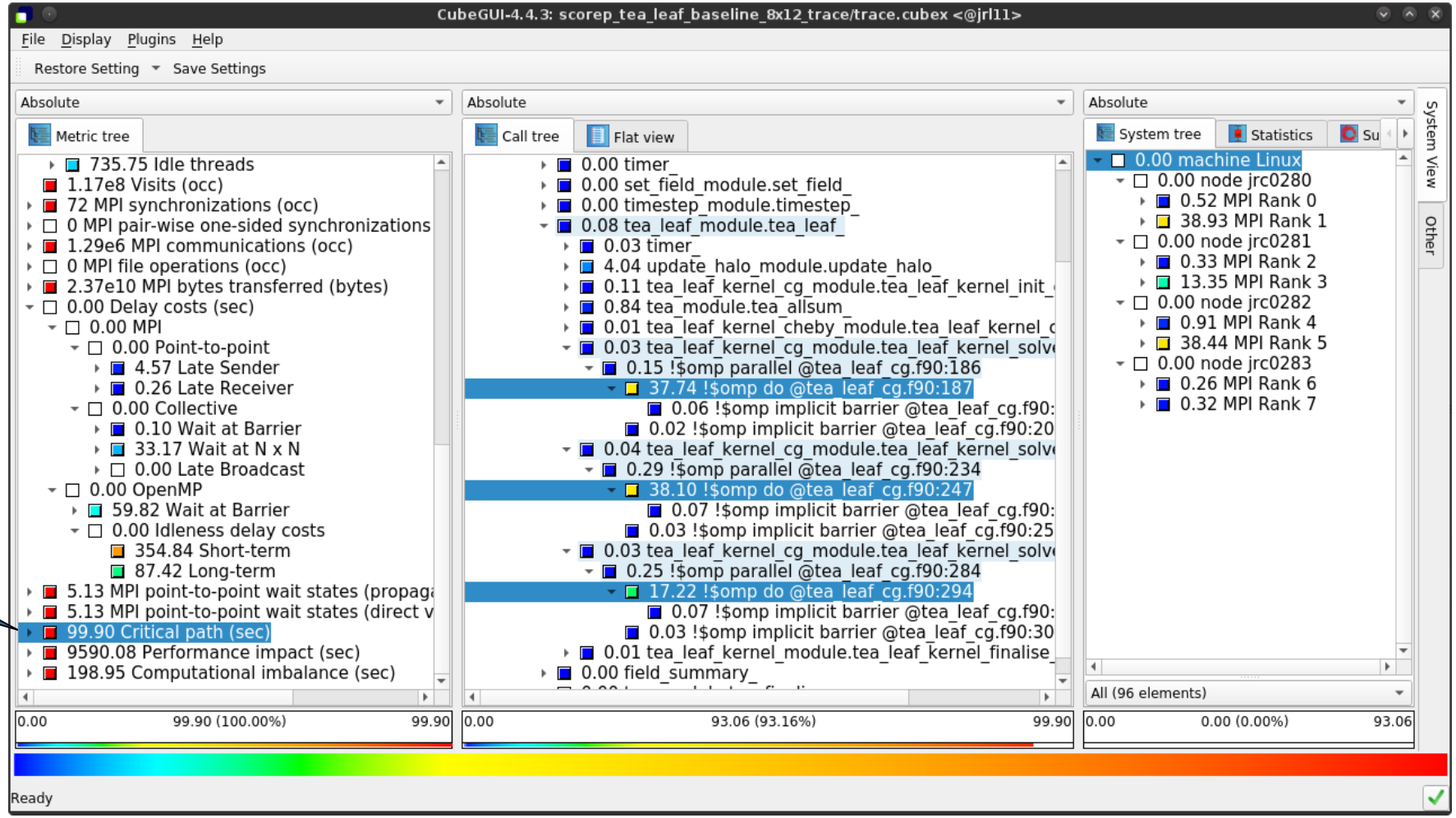
TeaLeaf Scalasca report analysis (V)

Various OpenMP `do` loops (incl. the solver loops) also cause OpenMP thread idleness on other ranks via propagation



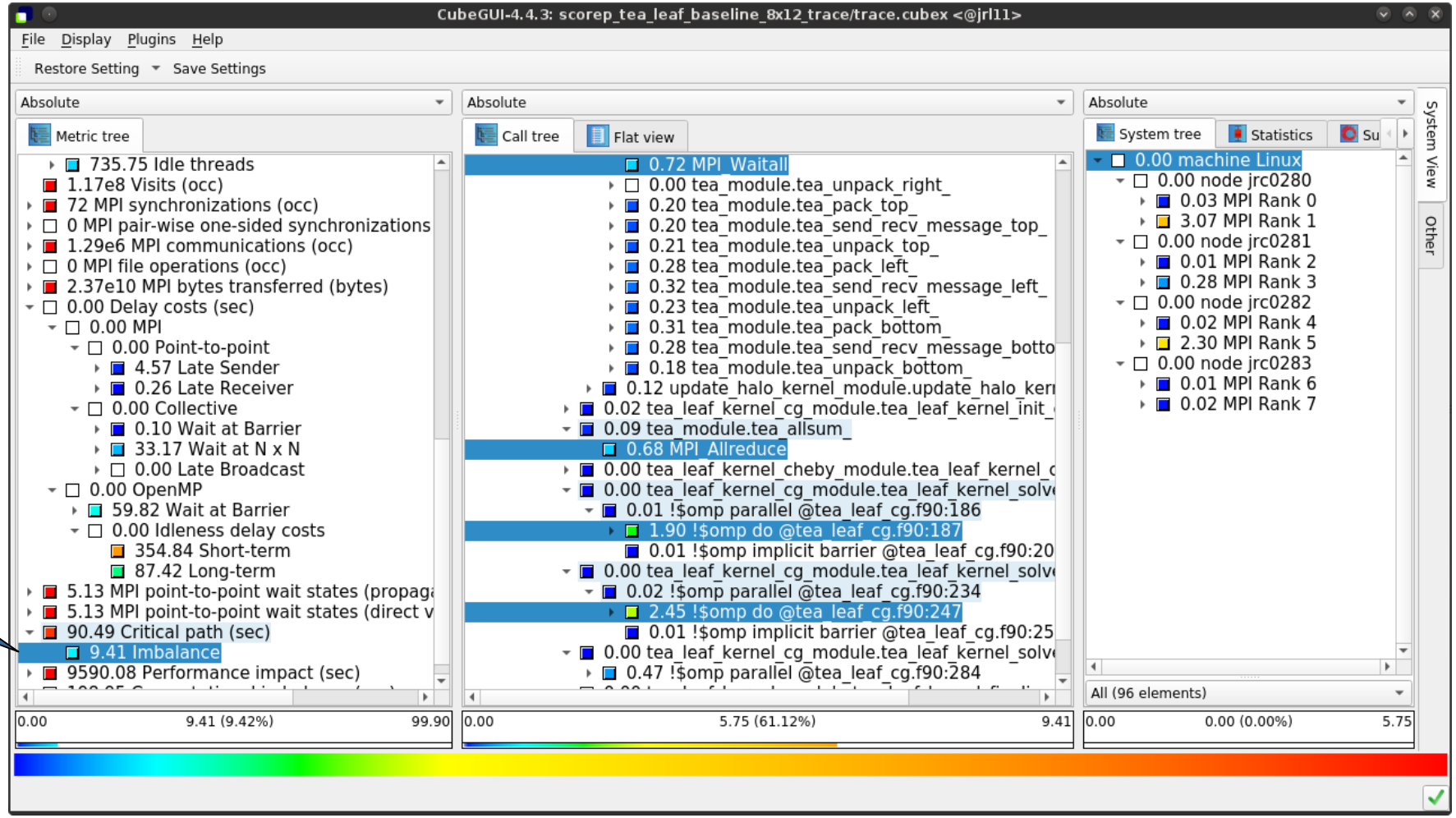
TeaLeaf Scalasca report analysis (VI)

The Critical Path also highlights the three solver loops…



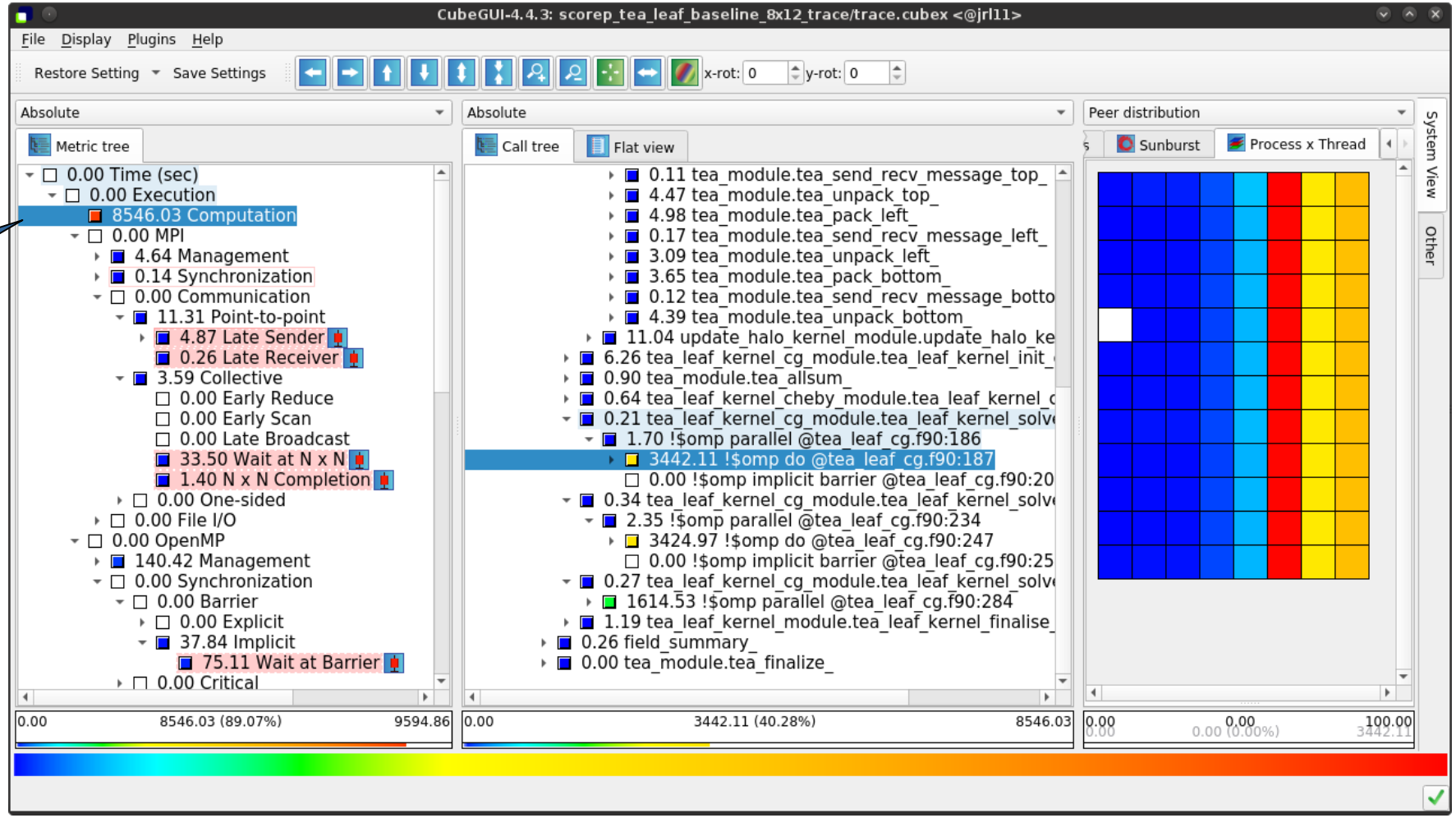
TeaLeaf Scalasca report analysis (VII)

…with imbalance (time on critical path above average) mostly in the first two loops and MPI communication



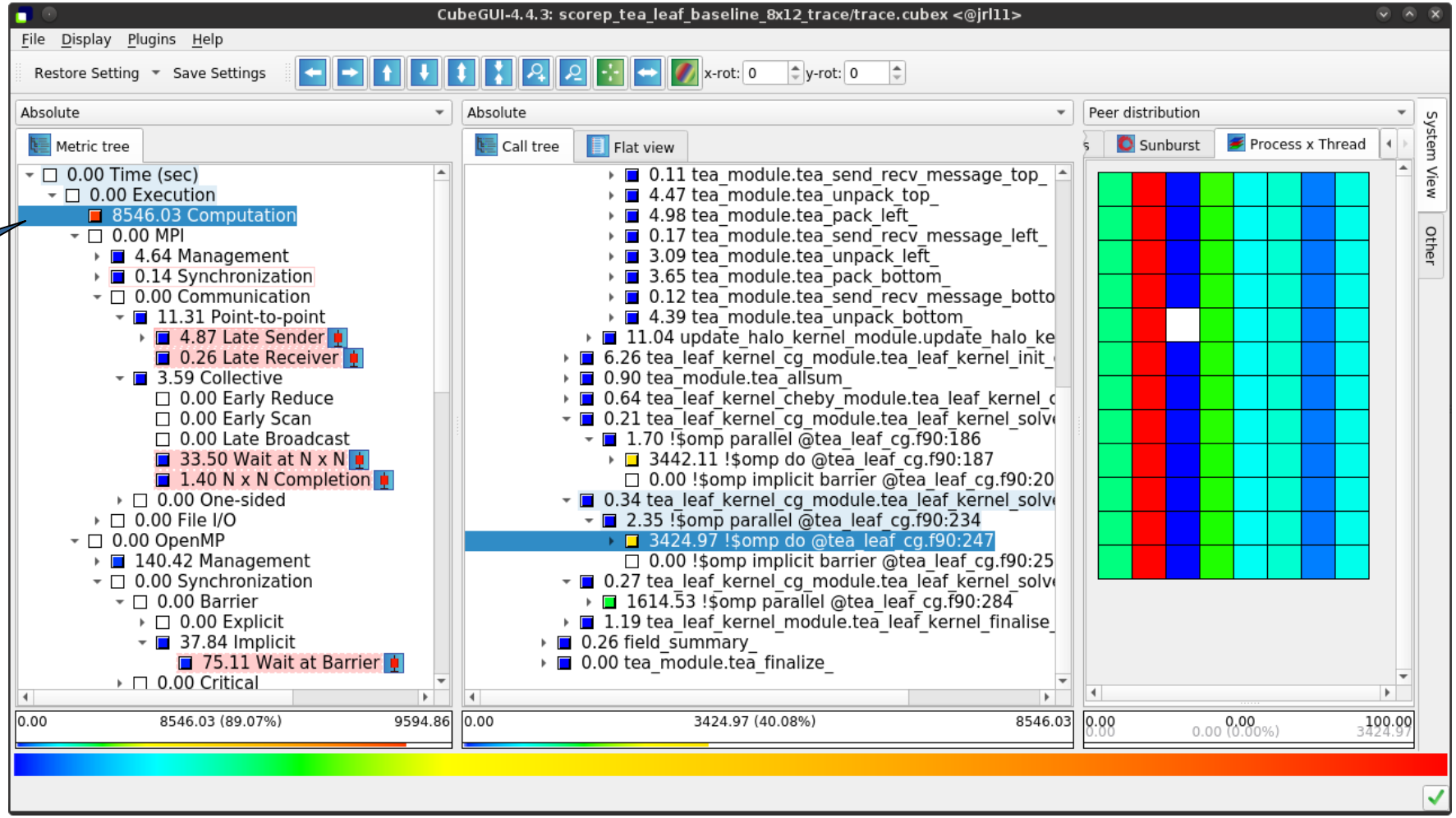
TeaLeaf Scalasca report analysis (VIII)

Computation time of 1st…



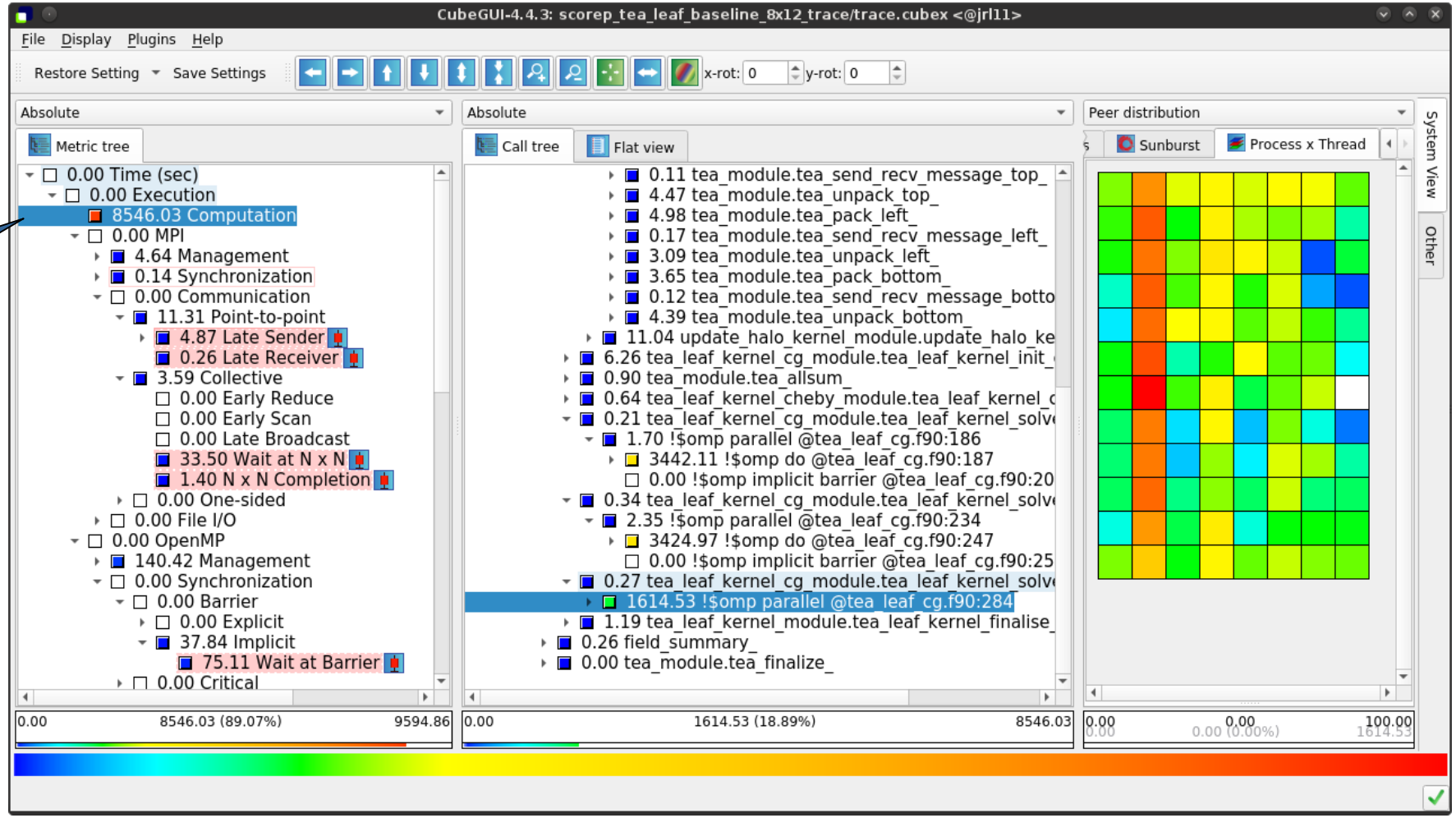
TeaLeaf Scalasca report analysis (IX)

…and 2nd `do` loop mostly balanced within each rank, but vary considerably across ranks…



TeaLeaf Scalasca report analysis (X)

…while the 3rd do loop also shows imbalance within each rank



TeaLeaf analysis summary

- The first two OpenMP do loops of the solver are well balanced within a rank, but are imbalanced across ranks
  - ➔ Requires a global load balancing strategy
- The third OpenMP do loop, however, is imbalanced within ranks,
  - causing direct "Wait at OpenMP Barrier" wait states,
  - which cause indirect MPI point-to-point wait states,
  - which in turn cause OpenMP thread idleness
  - ➔ Low-hanging fruit

- Adding a `SCHEDULE(guided)` clause reduced
  - the MPI point-to-point wait states by ~66%
  - the MPI collective wait states by ~50%
  - the OpenMP "Wait at Barrier" wait states by ~55%
  - the OpenMP thread idleness by ~11%
  - ➔ **Overall runtime (wall-clock) reduction by ~5%**

Using the Scalasca Trace Tools

A demo using NPB-MZ-MPI / BT

scan: Scalasca measurement and analysis nexus

- Used as prefix to the application launch command (e.g., `mpiexec`)
  - Can be used for both profile and trace measurements
- Configures Score-P measurement by automatically setting some environment variables and exporting them
  - For example, experiment title, profiling/tracing mode, filter file, …
  - Precedence order:
    - Command-line arguments
    - Environment variables already set
    - Automatically determined values
- Includes consistency checks and prevents corrupting existing experiment directories
- Initiates automatic parallel trace analysis after trace collection completes (if configured)
  - Selects the best-matching analyzer variant
  - Uses identical launch configuration to that used for measurement (i.e., the same allocated compute resources)

Example: NPB BT-MZ summary measurement

```
% export OMP_NUM_THREADS=2
% scan -f scorep.filt mpiexec -n 4 ./bt-mz_A.4

S=C=A=N: Scalasca 2.6 runtime summarization
S=C=A=N: Thu Aug 26 10:10:27 2021: Collect start
mpiexec -n 4 ./bt-mz_A.4

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

 Number of zones:   4 x   4
 Iterations: 200    dt:   0.000800
 Number of active processes:     4

 [... More application output ...]

S=C=A=N: Thu Aug 26 10:10:51 2021: Collect done (status=0) 24s
S=C=A=N: ./scorep_bt-mz_A_4x2_sum complete.
```

- Run the application using the Scalasca measurement collection & analysis nexus prefixed to launch command
- Creates experiment directory: scorep_bt-mz_A_4x2_sum

NPB BT-MZ summary analysis report examination

- Score summary analysis report

```
% square -s  scorep_bt-mz_A_4x2_sum
INFO: Post-processing runtime summarization report (profile.cubex)...
INFO: Score report written to ./scorep_bt-mz_A_4x2_sum/scorep.score
```

- Post-processing and interactive exploration with Cube

```
% square  scorep_bt-mz_A_4x2_sum
INFO: Displaying ./scorep_bt-mz_A_4x2_sum/summary.cubex...
                    [GUI showing summary analysis report]
```

Hint:
When working on a remote HPC system, run ‘square -s’ there first and then copy ‘summary.cubex’ to your desktop/laptop and use a local CubeGUI installation to improve responsiveness.

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

NPB BT-MZ trace measurement … collection

```
% export SCOREP_TOTAL_MEMORY=9MB
% scan -f scorep.filt -t mpiexec -n 4 ./bt-mz_A.4

S=C=A=N: Scalasca 2.6 trace collection and analysis
S=C=A=N: Thu Aug 26 11:57:15 2021: Collect start
mpiexec -n 4 ./bt-mz_A.4

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

 Number of zones:   4 x   4
 Iterations: 200    dt:   0.000800
 Number of active processes:     4

 [... More application output ...]

S=C=A=N: Thu Aug 26 11:57:38 2021: Collect done (status=0) 23s
```

- Starts measurement with collection of trace files ("`-t`" option)
- Using runtime measurement filter ("`-f scorep.filt`")

- Creates experiment directory: scorep_bt-mz_A_4x2_trace

NPB BT-MZ trace measurement … analysis

```
…
S=C=A=N: Thu Aug 26 11:57:38 2021: Analyze start
mpiexec -n 4 scout.hyb ./scorep_bt-mz_A_4x2_trace/traces.otf2

SCOUT   (Scalasca 2.6)

Analyzing experiment archive ./scorep_bt-mz_A_4x4_trace/traces.otf

Opening experiment archive ... done (0.000s).
Reading definition data    ... done (0.002s).
Reading event trace data   ... done (0.104s).
Preprocessing              ... done (0.094s).
Analyzing trace data       ... done (4.613s).
Writing analysis report    ... done (0.485s).

Max. memory usage         : 71.191MB

Total processing time     : 5.325s
S=C=A=N: Thu Aug 26 11:57:43 2021: Analyze done (status=0) 5s
S=C=A=N: ./scorep_bt-mz_A_4x2_trace complete.
```

- … and continues with automatic (parallel) analysis of trace files

NPB BT-MZ trace analysis report exploration

- Produces trace analysis report in the experiment directory containing trace-based wait-state, critical-path, and delay metrics

```
% square  scorep_bt-mz_A_4x2_trace
INFO: Post-processing runtime summarization result (profile.cubex)...
INFO: Post-processing trace analysis report (scout.cubex)...
INFO: Displaying ./scorep_bt-mz_A_4x2_trace/trace.cubex...

                          [GUI showing trace analysis report]
```

Hint:
When working on a remote HPC system, run ‘square -s’ there first and then copy ‘trace.cubex’ to your desktop/laptop and use a local CubeGUI installation to improve responsiveness.

Further information

trace tools scalasca

- 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:
  - https://www.scalasca.org
- Contact:
  - mailto: scalasca@fz-juelich.de

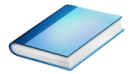
Acknowledgement

This tutorial is sponsored by the DEEP-SEA project.

www.deep-projects.eu
@DEEPprojects
@deep-projects

The DEEP Projects have received funding from the European Commission's FP7, H2020, and EuroHPC Programmes, under Grant Agreements n° 287530, 610476, 754304, and 955606.

The EuroHPC Joint Undertaking (JU) receives support from the European Union's Horizon 2020 research and innovation programme and Germany, France, Spain, Greece, Belgium, Sweden, United Kingdom, Switzerland.

Reference material 

---

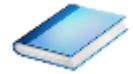
trace tools
scalasca

Scalasca command – One command for (almost) everything

```
% scalasca
Scalasca 2.6
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 2.6: 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
```

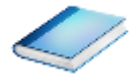
- Scalasca application instrumenter
  - Provides compatibility with Scalasca 1.x
  - **Deprecated! Use Score-P instrumenter directly.**

Scalasca convenience command: scan / scalasca -analyze

```
% scan
Scalasca 2.6: 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.
  -R #runs        : Specify the number of measurement runs per config.
  -M cfgfile      : Specify a config file for a multi-run measurement.
  -P preset       : Specify a preset for a multi-run measurement, e.g., 'pop'.
  -L              : List available multi-run presets.
  -D cfgfile      : Check a multi-run config file for validity and dump
                    the processed configuration for comparison.
```

- Scalasca measurement collection & analysis nexus

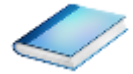
Scalasca advanced command: scout - Scalasca automatic trace analyzer



```
% scout.hyb --help
SCOUT   (Scalasca 2.6)
Copyright (c) 1998-2021 Forschungszentrum Juelich GmbH
Copyright (c) 2014-2021 RWTH Aachen University
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>
```

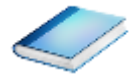
  - Provided in MPI (.mpi) and MPI+OpenMP (.hyb) variants
  - Takes as input a trace experiment archive where the events may have timestamp inconsistencies
    - E.g., multi-node measurements on systems without adequately synchronized clocks on each compute node
  - Generates a new experiment archive (always 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

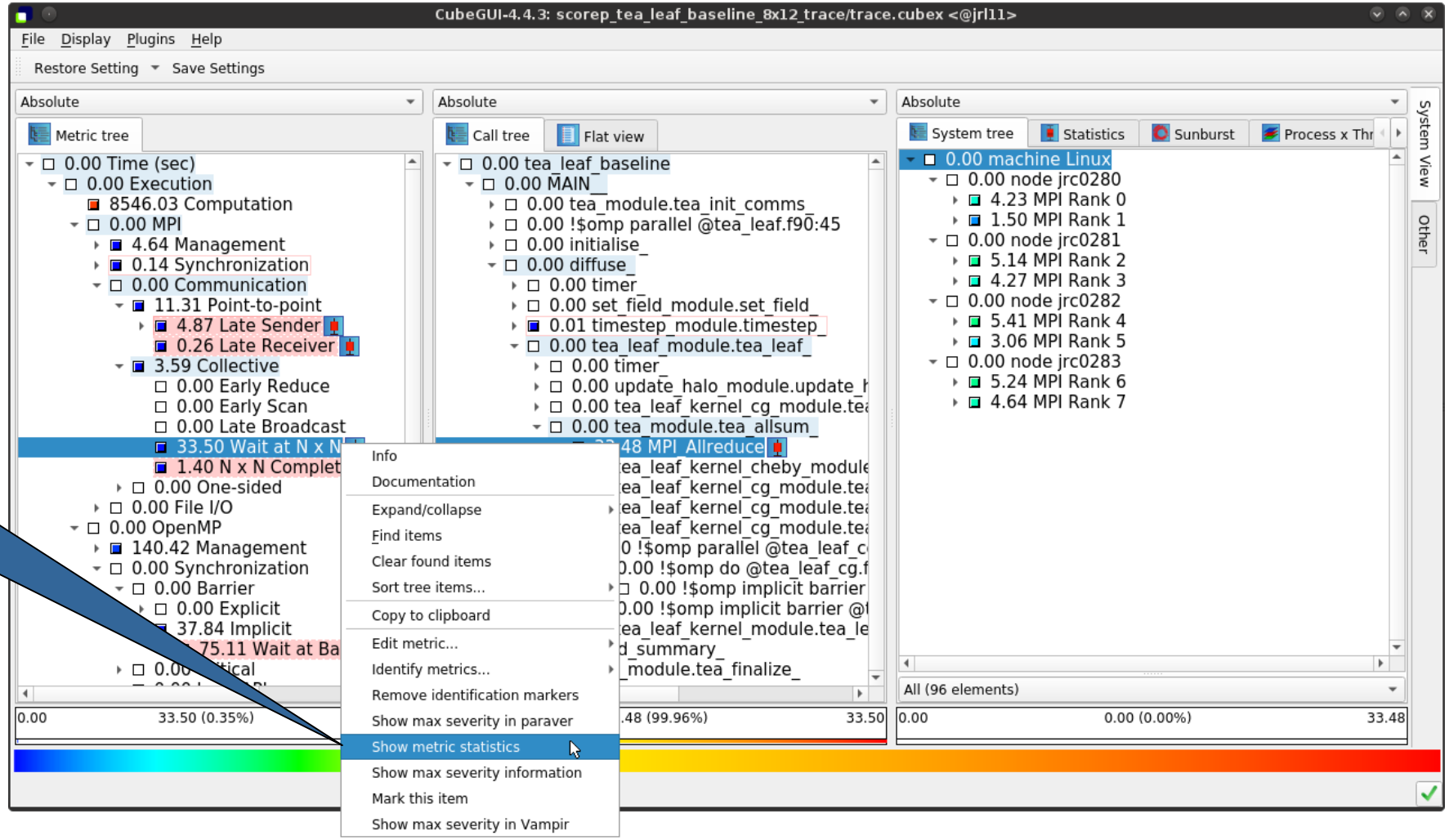
```
% square
Scalasca 2.6: analysis report explorer
usage: square [OPTIONS] <experiment archive | cube file>
   -C <none | quick | full> : Level of sanity checks for newly created reports
   -c <number>              : Consider number of counters when doing scoring (-s)
   -F                       : Force remapping of already existing reports
   -f filtfile              : Use specified filter file when doing scoring (-s)
   -s                       : Skip display and output textual score report
   -v                       : Enable verbose mode
   -n                       : Do not include idle thread metric
   -S <mean | merge>        : Aggregation method for summarization results of
                              each configuration (default: merge)
   -T <mean | merge>        : Aggregation method for trace analysis results of
                              each configuration (default: merge)
   -A                       : Post-process every step of a multi-run experiment
   -I                       : Ignore structural sanity checks and force aggregation
                              of measurements in a multi-run experiment
   -x <scorep-score opt>    : Pass option(s) to scorep-score
```

- Scalasca analysis report explorer (Cube)

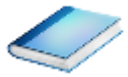
Metric statistics



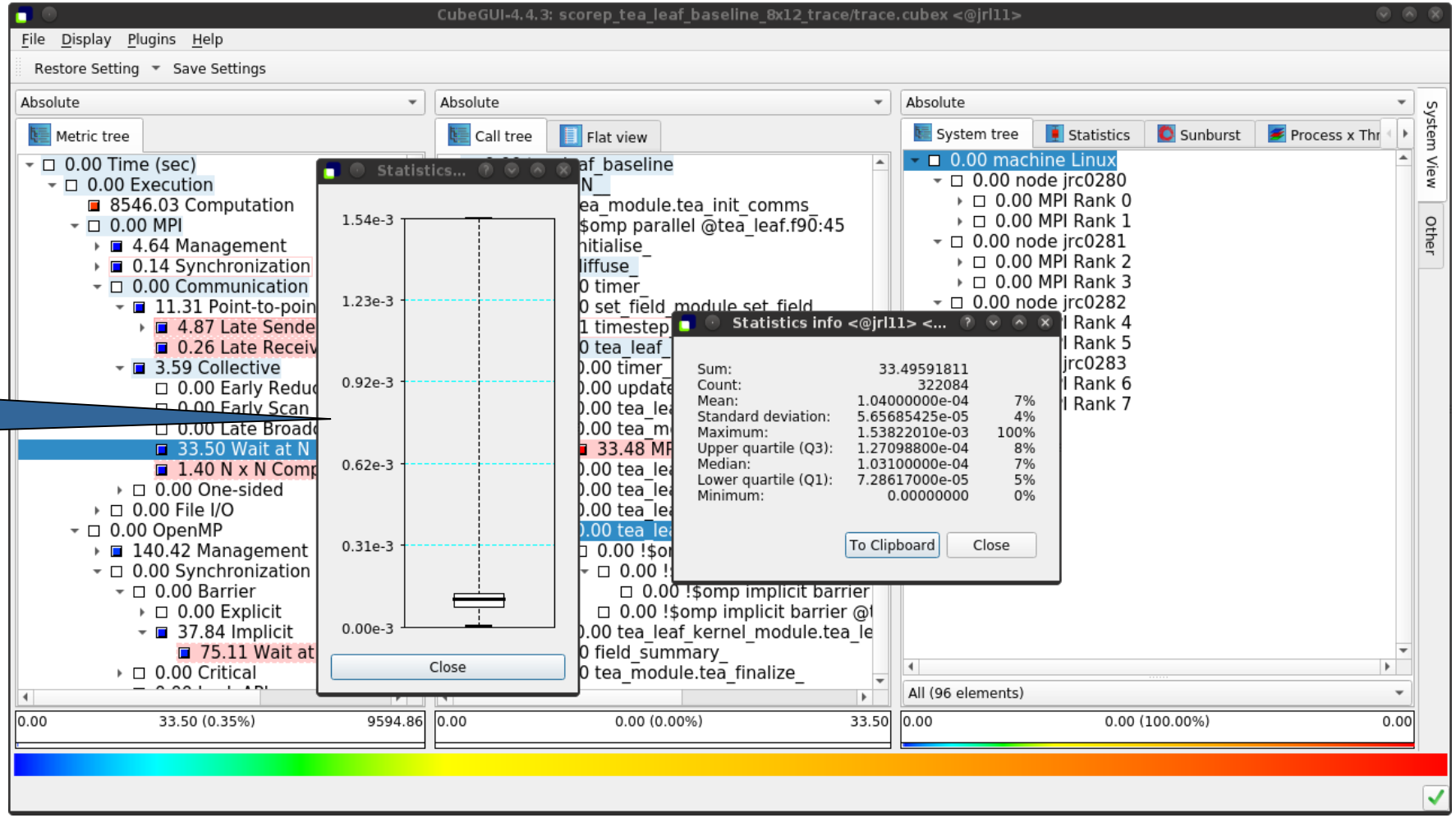
Access metric statistics for metrics marked with box plot icon from context menu



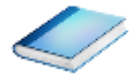
Metric statistics (cont.)



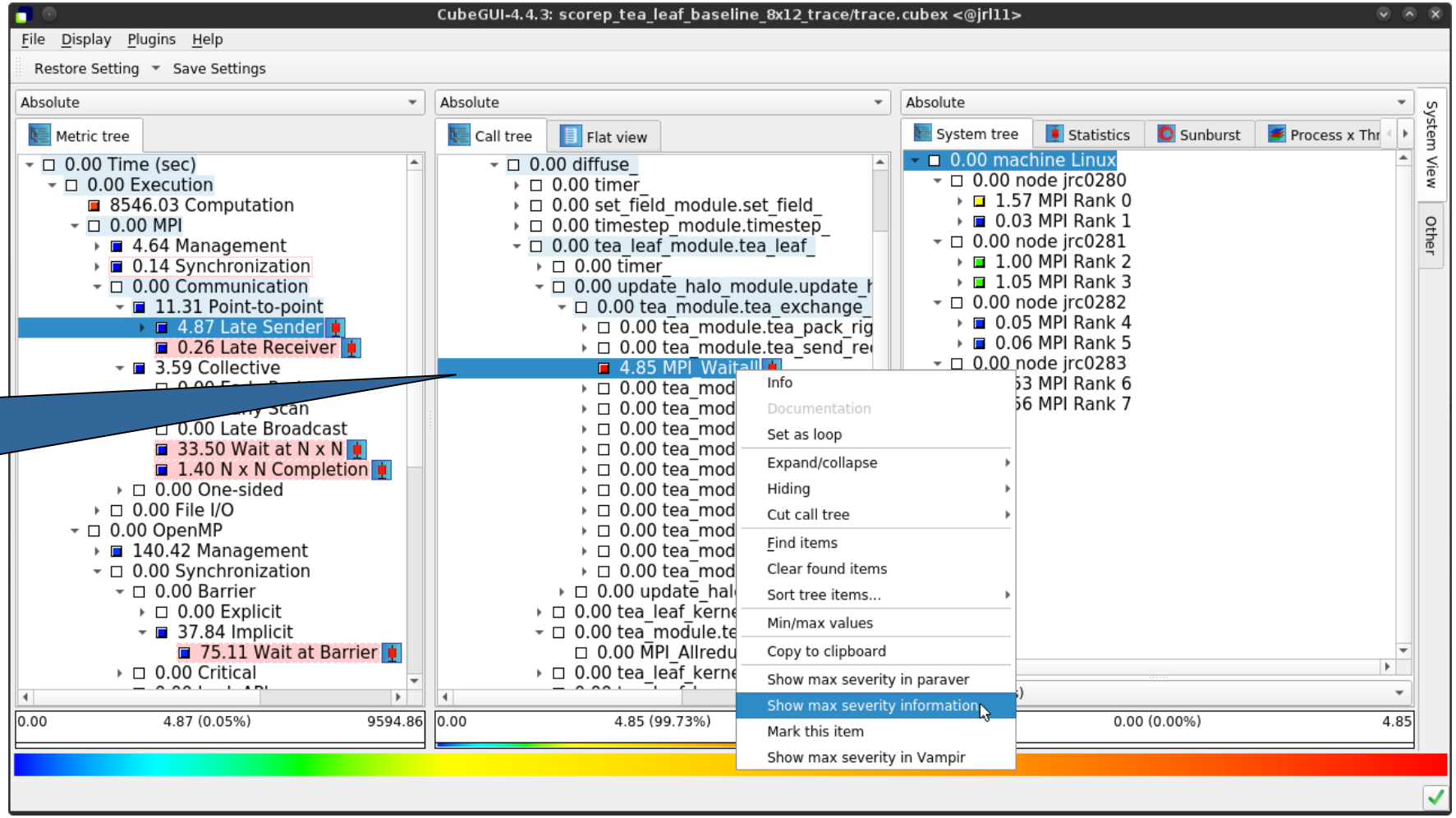
Shows instance statistics box plot, click to get details



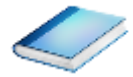
Metric instance statistics



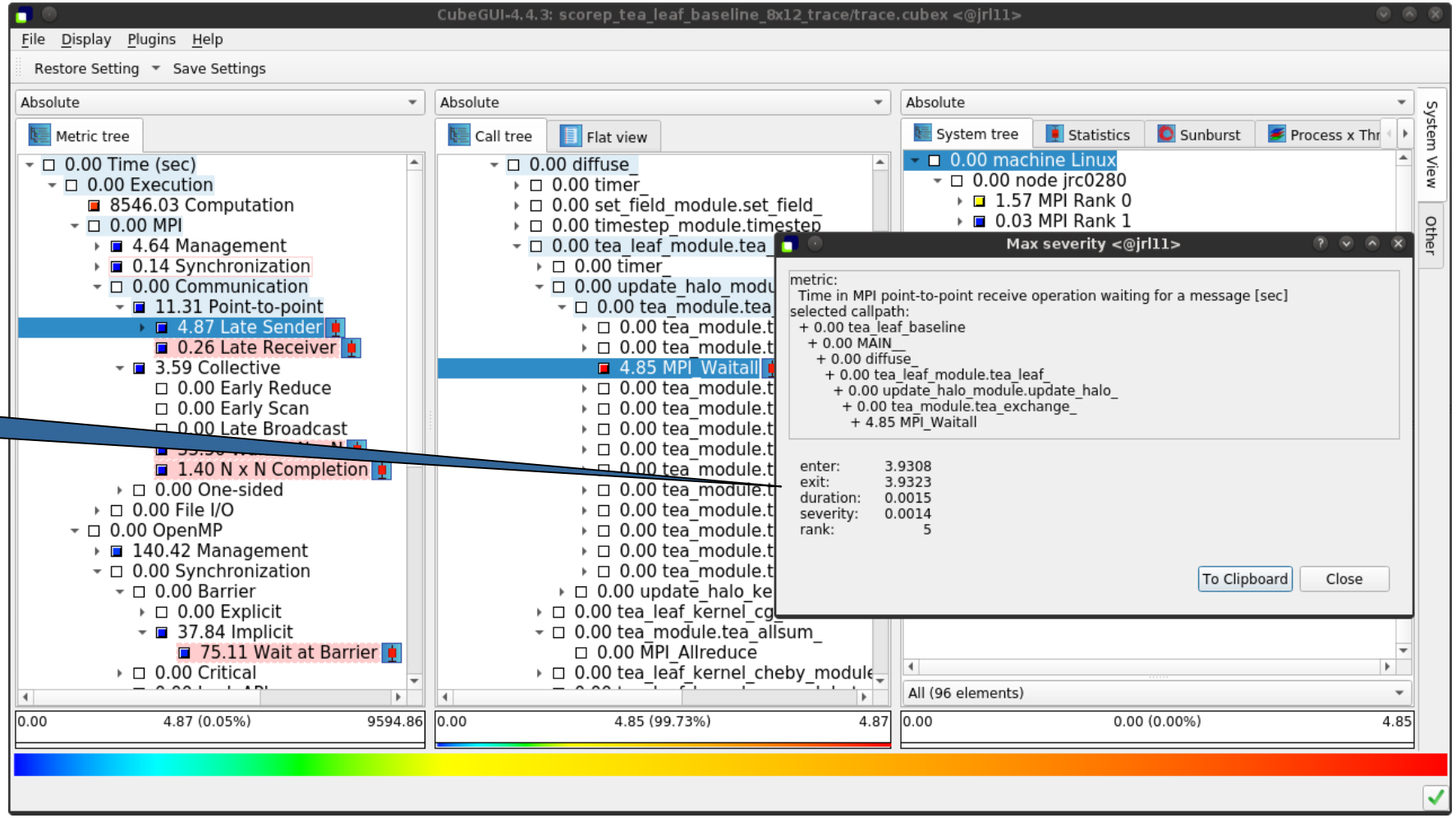
Access most-severe instance information for call paths marked with box plot icon via context menu



Metric instance statistics (cont.)

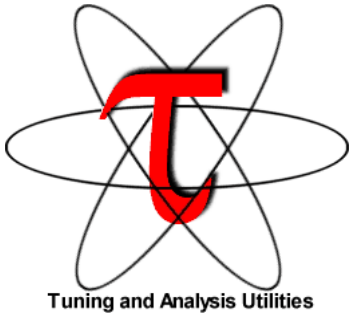


Shows instance details



VI-HPS

TAU Performance System®



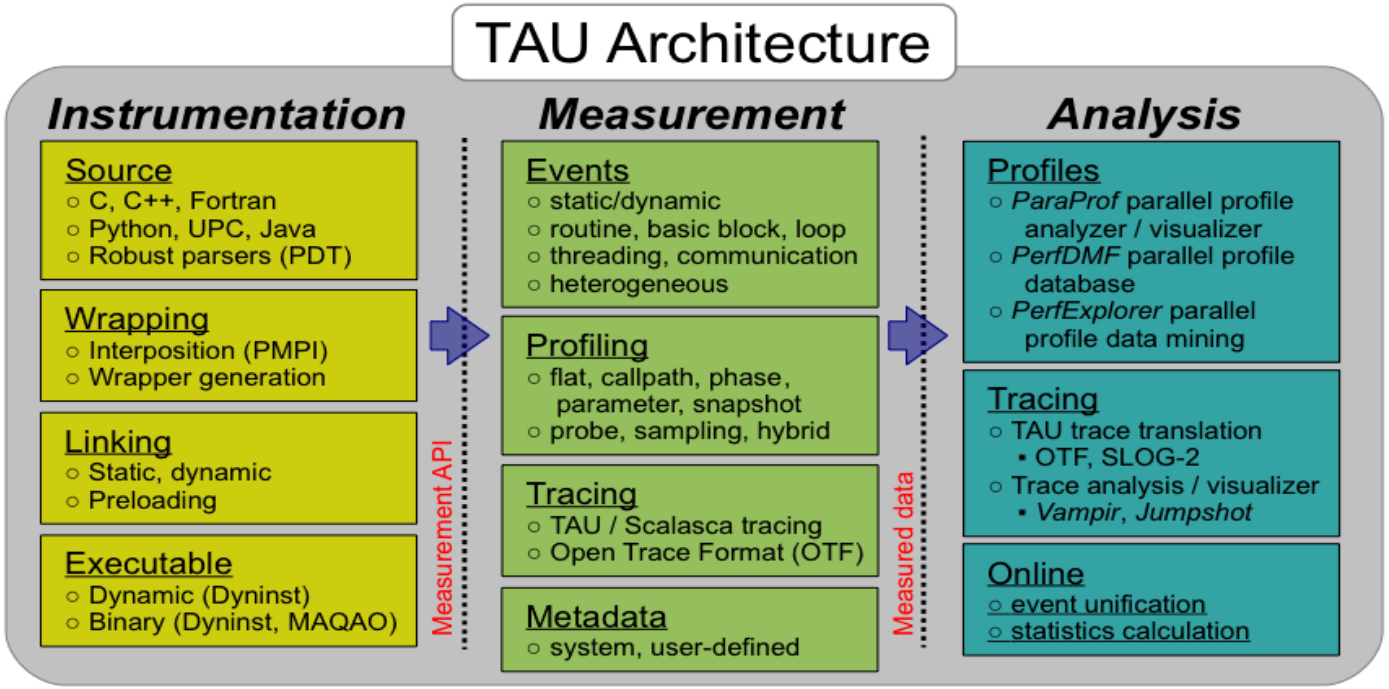
Sameer Shende
sameer@cs.uoregon.edu
University of Oregon
http://tau.uoregon.edu/TAU_ISC22.pdf

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? Using Likwid or PAPI, TAU measures floating point, Level 1 and 2 *data cache misses*, hits, branches taken.
- What is the time taken in OS routines for thread scheduling? How much time is wasted?
- 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?
- 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?

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, ROCm, Level Zero (Intel)
  - 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's Support for Runtime Systems

- *MPI*
  - PMPI profiling interface
  - MPI_T tools interface using performance and control variables
- *Pthread*
  - Captures time spent in routines per thread of execution
- *OpenMP*
  - OMPT tools interface to track salient OpenMP runtime events
  - Opari source rewriter
  - Preloading wrapper OpenMP runtime library when OMPT is not supported
- *OpenACC*
  - OpenACC instrumentation API
  - Track data transfers between host and device (per-variable)
  - Track time spent in kernels
- *Level Zero (DPC++/SYCL Intel oneAPI)*
  - Track execution of kernels on GPU
  - Track time spent in level zero runtime system calls

TAU's Support for Runtime Systems (contd.)

- *OpenCL*
  - OpenCL profiling interface
  - Track timings of kernels
- *CUDA*
  - Cuda Profiling Tools Interface (CUPTI)
  - Track data transfers between host and GPU
  - Track access to uniform shared memory between host and GPU
- *ROCm*
  - Rocprofiler and Roctracer instrumentation interfaces
  - Track data transfers and kernel execution between host and GPU
- *Kokkos*
  - Kokkos profiling API
  - Push/pop interface for region, kernel execution interface
- *Python*
  - Python interpreter instrumentation API
  - Tracks Python routine transitions as well as Python to C transitions

Examples of Multi-Level Instrumentation

- *MPI + OpenMP*
  - MPI_T + PMPI + OMPT may be used to track MPI and OpenMP
- *MPI + CUDA*
  - PMPI + CUPTI interfaces
- *Kokkos + OpenMP*
  - Kokkos profiling API + OMPT to transparently track events
- *Kokkos + pthread + MPI*
  - Kokkos + pthread wrapper interposition library + PMPI layer
- *Python + CUDA + MPI*
  - Python + CUPTI + pthread profiling interfaces (e.g., Tensorflow, PyTorch) + MPI
- *MPI + OpenCL*
  - PMPI + OpenCL profiling interfaces

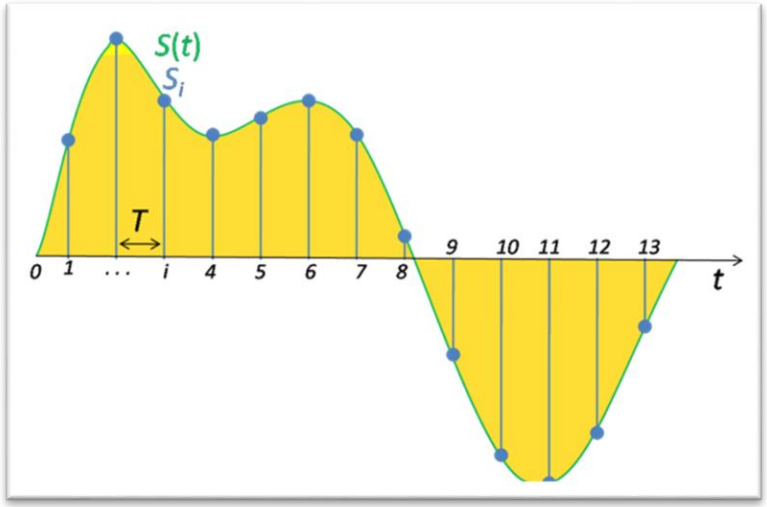
Performance Data Measurement

Direct via Probes

```
Call START('potential')
// code
Call STOP('potential')
```

- Exact measurement
- Fine-grain control
- Calls inserted into code

Indirect via Sampling



- No code modification
- Minimal effort
- Relies on debug symbols (**-g**)

Types of Performance Profiles

- *Flat* profiles
  - Metric (e.g., time) spent in an event
  - Exclusive/inclusive, # of calls, child calls, …
- *Callpath* profiles
  - Time spent along a calling path (edges in callgraph)
  - "*main=> f1 => f2 => MPI_Send*"
  - Set the TAU_CALLPATH and TAU_CALLPATH_DEPTH environment variables
- *Callsite* profiles
  - Time spent along in an event at a given source location
  - Set the TAU_CALLSITE environment variable
- *Phase* profiles
  - Flat profiles under a phase (nested phases allowed)
  - Default "main" phase
  - Supports static or dynamic (e.g. per-iteration) phases

Using TAU's Runtime Preloading Tool: tau_exec

- Preload a wrapper that intercepts the runtime system call and substitutes with another
  - **MPI**
  - **OpenMP**
  - **POSIX I/O**
  - **Memory allocation/deallocation routines**
  - **Wrapper library for an external package**
- No modification to the binary executable!
- Enable other TAU options (communication matrix, OTF2, event-based sampling)
- Add tau_exec before the name of the binary
  - mpirun –np 64 tau_exec ./a.out
  - mpirun tau_exec –T omp,mpi,papi -ompt ./a.out

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!

Simplifying TAU's usage (tau_exec)

- Uninstrumented execution linked with –dynamic (dynamic executables only!)

  % mpirun -np 16 ./a.out
- Track MPI performance

  % mpirun -np 16 tau_exec ./a.out
- Track OpenMP, and MPI performance (MPI enabled by default; OMPT in Clang 9+, Intel 19+)

  % export TAU_OMPT_SUPPORT_LEVEL=full;

  % mpirun -np 16 tau_exec –T mpi,pdt,ompt,v5,papi –ompt ./a.out
- Track memory operations

  % export TAU_TRACK_MEMORY_LEAKS=1

  % mpirun -np 16 tau_exec –memory_debug ./a.out (bounds check)
- Use event based sampling (compile with –g)

  % mpirun -np 16 tau_exec –ebs ./a.out

  Also –ebs_source=<PAPI_COUNTER> -ebs_period=<overflow_count> -ebs_resolution=<file|function|line>
- Load wrapper interposition library

  % mpirun -np 16 tau_exec –loadlib=<path/libwrapper.so> ./a.out
- Track GPGPU operations (-rocm, -l0, -opencl, -cupti, -cupti –um, -openacc):

  % mpirun -np 16 tau_exec –cupti ./a.out

Configuration tags for tau_exec

% ./configure –pdt=<dir> -mpi –papi=<dir>; make install
Creates in $TAU:
Makefile.tau-papi-mpi-pdt(Configuration parameters in stub makefile)
shared-papi-mpi-pdt/libTAU.so

% ./configure –pdt=<dir> -mpi; make install creates
Makefile.tau-mpi-pdt
shared-mpi-pdt/libTAU.so

To explicitly choose preloading of shared-<options>/libTAU.so change:
% mpirun -np 256 ./a.out to
% mpirun -np 256 tau_exec –T <comma_separated_options> ./a.out

% mpirun -np 256 tau_exec –T papi,mpi,pdt ./a.out
Preloads $TAU/shared-papi-mpi-pdt/libTAU.so
% mpirun -np 256 tau_exec –T papi ./a.out
Preloads $TAU/shared-papi-mpi-pdt/libTAU.so by matching.
% aprun –n 256 tau_exec –T papi,mpi,pdt –s ./a.out
Does not execute the program. Just displays the library that it will preload if executed without the –s option.
NOTE: -mpi configuration is selected by default. Use –T serial for
Sequential programs.

TAU Execution Command (tau_exec)

- Uninstrumented execution
  - % mpirun -np 256 ./a.out
- Track GPU operations
  - % mpirun –np 256 tau_exec –rocm ./a.out
  - % mpirun –np 256 tau_exec –cupti ./a.out
  - % mpirun –np 256 tau_exec –opencl ./a.out
  - % mpirun –np 256 tau_exec –openacc ./a.out
  - % mpirun –np 256 tau_exec –l0 ./a.out
- Track MPI performance
  - % mpirun -np 256 tau_exec ./a.out
- Track I/O, and MPI performance (MPI enabled by default)
  - % mpirun -np 256 tau_exec -io ./a.out
- Track OpenMP and MPI execution (using OMPT for Intel v19+ or Clang 8+)
  - % export TAU_OMPT_SUPPORT_LEVEL=full;
  - % mpirun –np 256 tau_exec –T ompt,v5,mpi -ompt ./a.out
- Track memory operations
  - % export TAU_TRACK_MEMORY_LEAKS=1
  - % mpirun –np 256 tau_exec –memory_debug ./a.out (bounds check)
- Use event based sampling (compile with –g)
  - % mpirun –np 256 tau_exec –ebs ./a.out
  - Also -ebs_source=<PAPI_COUNTER> -ebs_period=<overflow_count> -ebs_resolution=<file | function | line>

TAU's Runtime Environment Variables

Environment Variable	Default	Description
TAU_TRACE	0	Setting to 1 turns on tracing
TAU_CALLPATH	0	Setting to 1 turns on callpath profiling
TAU_TRACK_MEMORY_FOOTPRINT	0	Setting to 1 turns on tracking memory usage by sampling periodically the resident set size and high water mark of memory usage
TAU_TRACK_POWER	0	Tracks power usage by sampling 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. Throttle a routine if it is called over 100000 times and takes less than 10 usec of inclusive time per call
TAU_CALLSITE	0	Setting to 1 enables callsite profiling that shows where an instrumented function was called. Also compatible with tracing.
TAU_PROFILE_FORMAT	Profile	Setting to "merged" generates a single file. "snapshot" generates xml format

Runtime Environment Variables

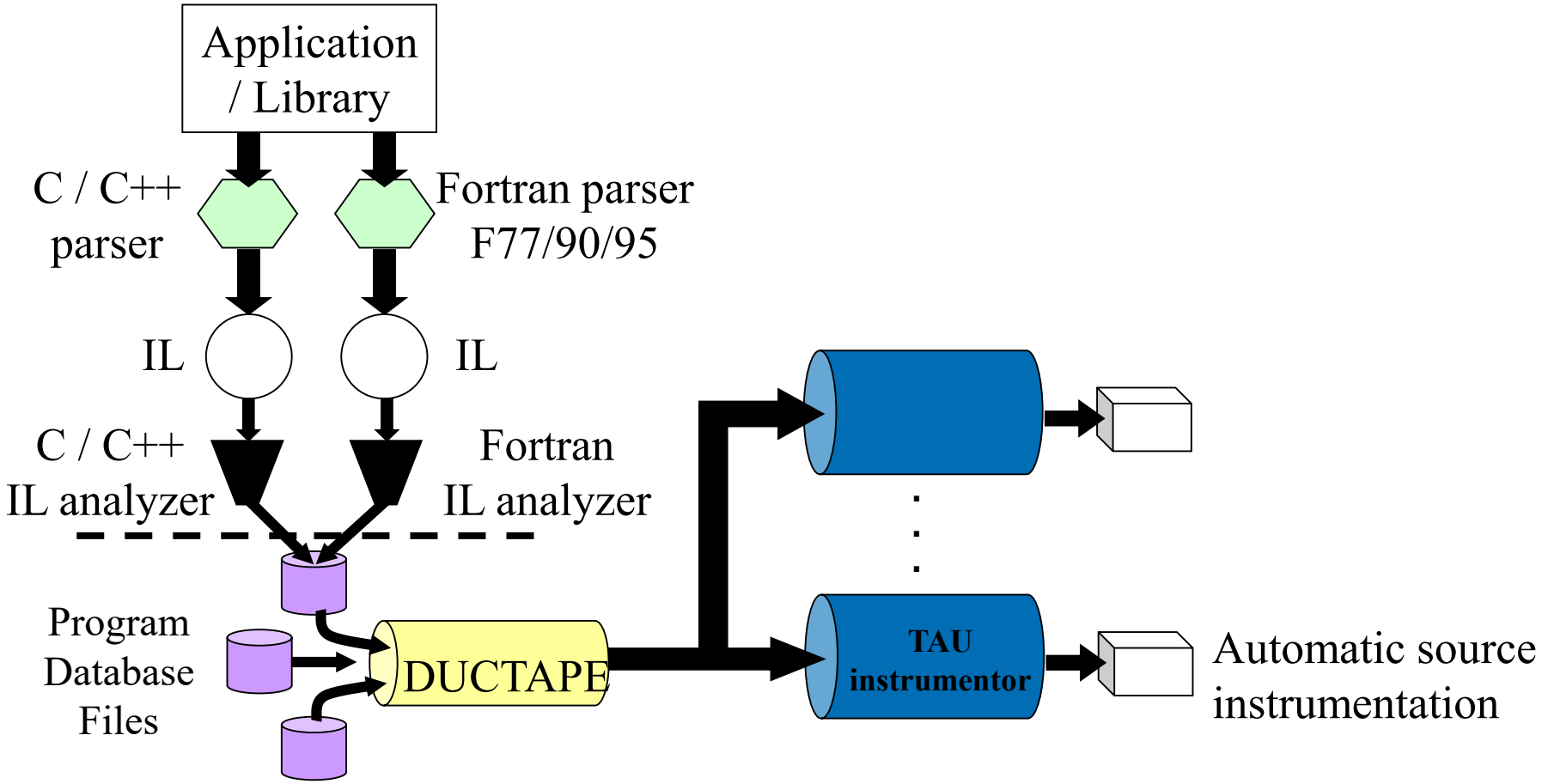
Environment Variable	Default	Description
TAU_METRICS	TIME	Setting to a comma separated list generates other metrics. (e.g., ENERGY,TIME,P_VIRTUAL_TIME,PAPI_FP_INS,PAPI_NATIVE_<event>:<subevent>)
TAU_TRACE	0	Setting to 1 turns on tracing
TAU_TRACE_FORMAT	Default	Setting to “otf2” turns on TAU’s native OTF2 trace generation (configure with –otf=download)
TAU_EBS_UNWIND	0	Setting to 1 turns on unwinding the callstack during sampling (use with tau_exec –ebs or TAU_SAMPLING=1)
TAU_EBS_RESOLUTION	line	Setting to “function” or “file” changes the sampling resolution to function or file level respectively.
TAU_TRACK_LOAD	0	Setting to 1 tracks system load on the node
TAU_SELECT_FILE	Default	Setting to a file name, enables selective instrumentation based on exclude/include lists specified in the file.
TAU_OMPT_SUPPORT_LEVEL	basic	Setting to “full” improves resolution of OMPT TR6 regions on threads 1.. N-1. Also, “lowoverhead” option is available.
TAU_OMPT_RESOLVE_ADDRESS_EAGERLY	1	Setting to 1 is necessary for event based sampling to resolve addresses with OMPT. Setting to 0 allows the user to do offline address translation.
TAU_EVENT_THRESHOLD	0.5	Define a threshold value (e.g., .25 is 25%) to trigger marker events for min/max

Runtime Environment Variables

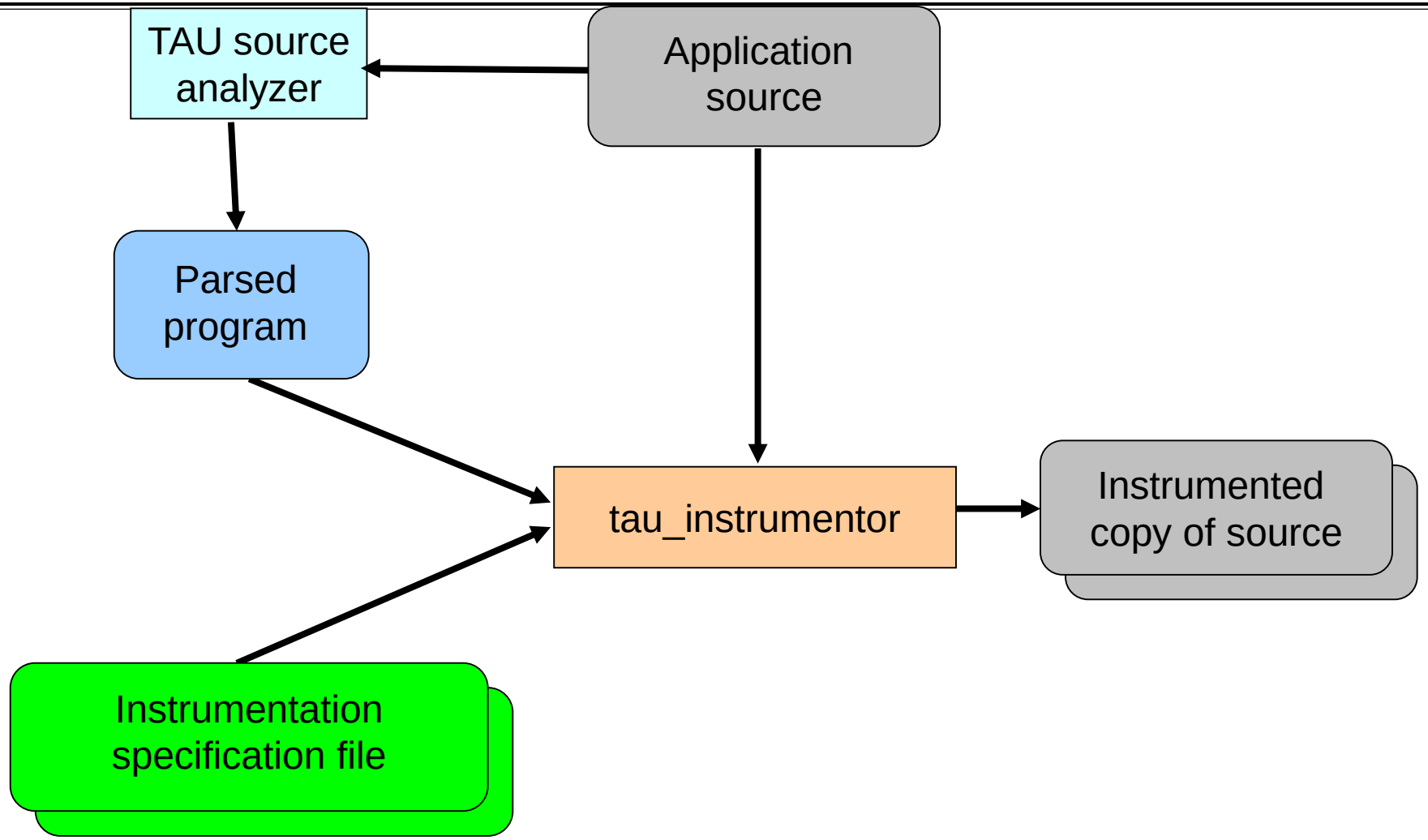
Environment Variable	Default	Description
TAU_TRACK_MEMORY_LEAKS	0	Tracks allocates that were not de-allocated (needs –optMemDbg or tau_exec –memory)
TAU_EBS_SOURCE	TIME	Allows using PAPI hardware counters for periodic interrupts for EBS (e.g., TAU_EBS_SOURCE=PAPI_TOT_INS when TAU_SAMPLING=1)
TAU_EBS_PERIOD	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's Source Instrumentation

TAU's Static Analysis System: Program Database Toolkit (PDT)



Automatic Source Instrumentation using PDT



Installing and Configuring TAU

- Installing PDT:
  - wget http://tau.uoregon.edu/pdt.tgz
  - ./configure; make ; make install
- Installing TAU :
  - wget http://tau.uoregon.edu/tau.tgz
  - ./configure –ompt –c++=mpiicpc –cc=mpiicc –fortran=mpiifort -mpi -bfd=download -pdt=<dir> -papi=<dir> ...
  - make install; export PATH=<taudir>/x86_64/bin:$PATH
  - All configurations are stored in <taudir>/.all_configs if you wish to see how TAU was configured!
- Using TAU for source instrumentation:
  - export TAU_MAKEFILE=<taudir>/x86_64/lib/Makefile.tau-<TAGS>
  - make CC=tau_cc.sh CXX=tau_cxx.sh F90=tau_f90.sh
  - Use tau_exec with uninstrumented binaries instead of recompiling the source code.

Installing TAU on your laptop for paraprof (GUI)

- Microsoft Windows
  - Install Java from Oracle.com
  - http://tau.uoregon.edu/tau.exe
  - Install, click on a ppk file to launch paraprof
- macOS
  - Install Java 11.0.3:
    - Download http://tau.uoregon.edu/java.dmg
    - If you have multiple Java installations, add to your ~/.zshrc (or ~/.bashrc as appropriate):
    - export PATH=/Library/Java/JavaVirtualMachines/jdk-11.0.3.jdk/Contents/Home/bin:$PATH
    - java -version
  - Download and install TAU (copy to /Applications from dmg):
    - http://tau.uoregon.edu/tau.dmg
    - export PATH=/Applications/TAU/tau/apple/bin:$PATH
    - paraprof app.ppk &
- Linux (http://tau.uoregon.edu/tau.tgz)
  - ./configure; make install; export PATH=<taudir>/x86_64/bin:$PATH
  - paraprof app.ppk &

Using TAU's Source Code Instrumentation

- 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-mpi-pdt-openmp-opari`**
  `% export TAU_OPTIONS='-optVerbose …' (see tau_compiler.sh )`

  Use tau_f90.sh, tau_f77.sh , tau_cxx.sh, tau_upc.sh, or tau_cc.sh as F90, F77, 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)

Makefiles for TAU Compiler and Runtime Options (AWS E4S.io)

```
% singularity run ~/ecp.simg
Singularity> ls /opt/tau/tau_latest/x86_64/lib/Makefile*
/opt/tau/tau_latest/x86_64/lib/Makefile.tau
/opt/tau/tau_latest/x86_64/lib/Makefile.tau-gnu-mpi-cupti-pdt
/opt/tau/tau_latest/x86_64/lib/Makefile.tau-level_zero-intel-icpc-mpi-pthread
/opt/tau/tau_latest/x86_64/lib/Makefile.tau-level_zero-pthread
/opt/tau/tau_latest/x86_64/lib/Makefile.tau-mpi-cupti-pdt
/opt/tau/tau_latest/x86_64/lib/Makefile.tau-mpi-pdt
/opt/tau/tau_latest/x86_64/lib/Makefile.tau-papi-mpi-pthread-python-pdt
/opt/tau/tau_latest/x86_64/lib/Makefile.tau-rocprofiler-rocm-clang-pthread-python-pdt
Singularity>
```

For an MPI+OpenMP+F90 application with GNU compilers and MPI, you may choose

/opt/tau/tau_latest/x86_64/lib/Makefile.tau-mpi-pdt

Supports MPI instrumentation & Opari and PDT for automatic source instrumentation

```
% export TAU_MAKEFILE=/opt/tau/tau_latest/x86_64/lib/Makefile.tau-mpi-pdt
% make F90=tau_f90,sh; mpirun –np 16 ./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

-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.

…

Selective Instrumentation File With Program Database Toolkit (PDT)

- To use an instrumentation specification file for source instrumentation:

  % export TAU_OPTIONS= '-optTauSelectFile=/path/to/select.tau –optVerbose '
  % cat select.tau

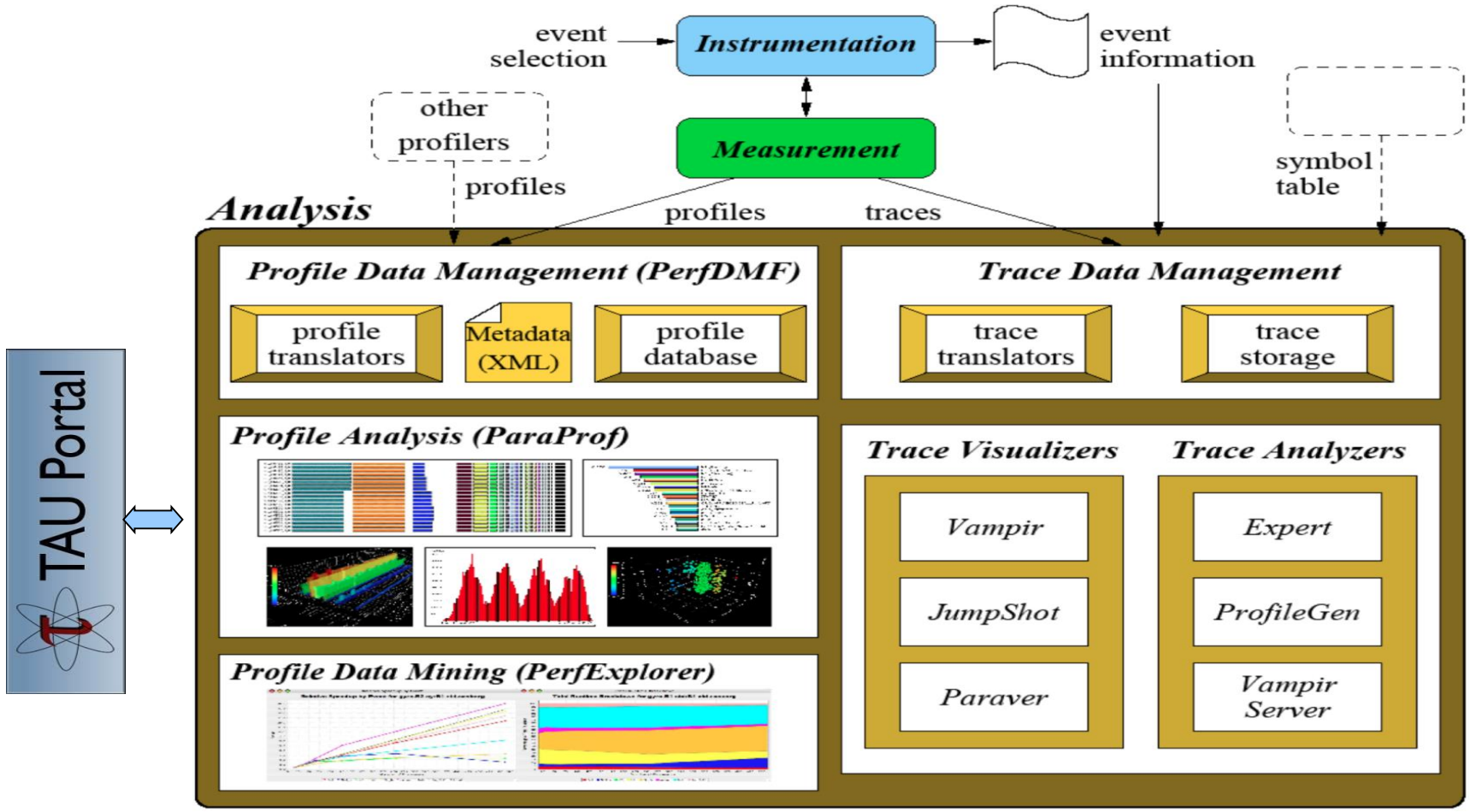
```
BEGIN_EXCLUDE_LIST
BINVCRHS
MATMUL_SUB
MATVEC_SUB
EXACT_SOLUTION
BINVRHS
LHS#INIT
TIMER_#
END_EXCLUDE_LIST
```

  - NOTE: paraprof can create this file from an earlier execution for you based on throttling.
  - File -> Create Selective Instrumentation File -> save
  - Selective instrumentation at runtime:

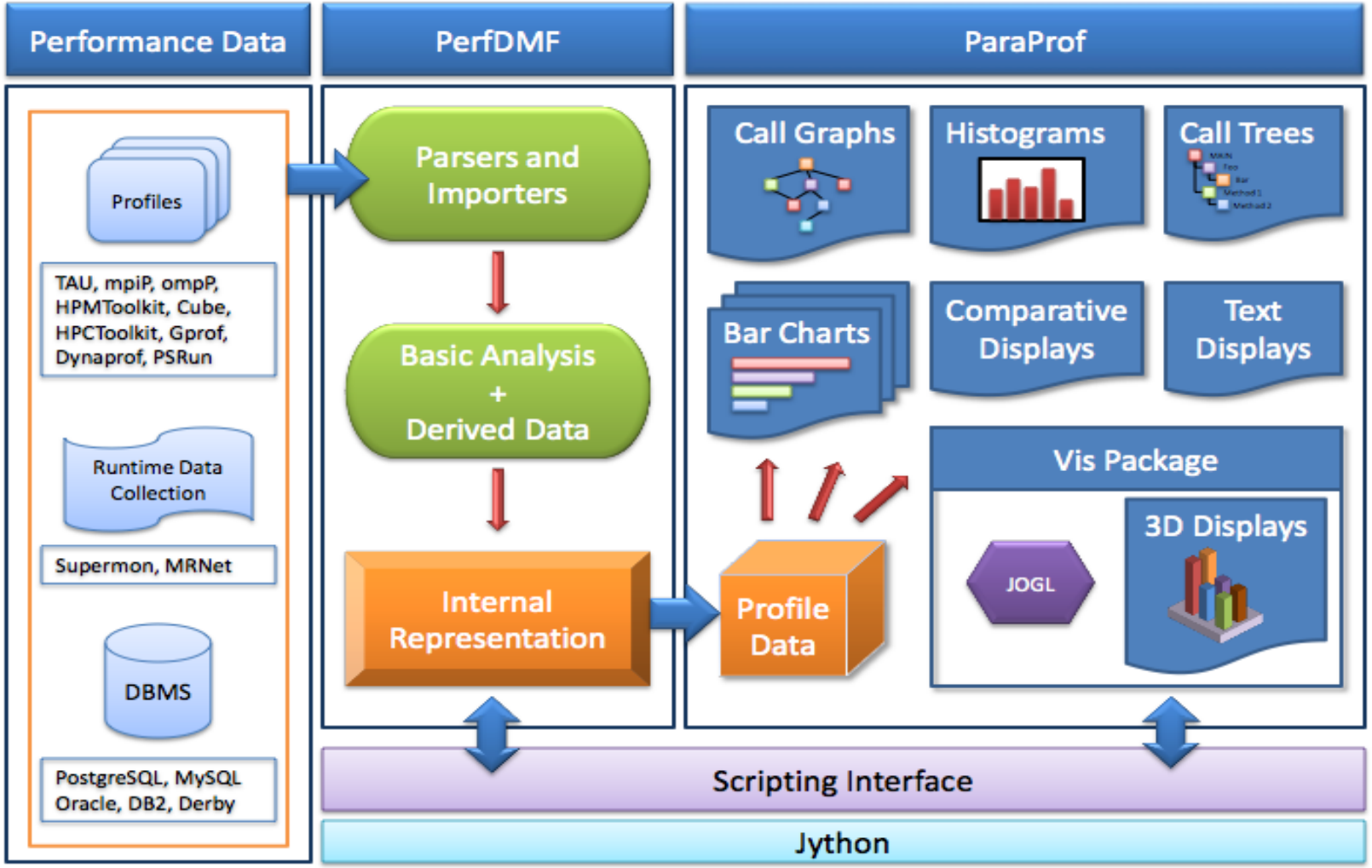
```
% export TAU_SELECT_FILE=select.tau
```

TAU's Analysis Tools: ParaProf

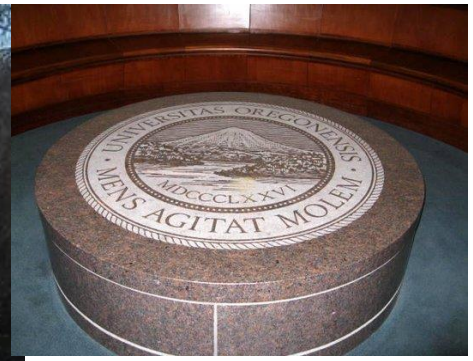
TAU Analysis



ParaProf Profile Analysis Framework



Performance Research Lab, University of Oregon, Eugene, USA



Support Acknowledgments

- US Department of Energy (DOE)
  - Office of Science contracts, ECP
  - SciDAC, LBL contracts
  - LLNL-LANL-SNL ASC/NNSA contract
  - Battelle, PNNL contract
  - ANL, ORNL contract
- Department of Defense (DoD)
  - PETTT, HPCMP
- National Science Foundation (NSF)
  - Glassbox, SI-2
- NASA
- CEA, France
- Partners:
  - University of Oregon
  - ParaTools, Inc., ParaTools, SAS
  - The Ohio State University
  - University of Tennessee, Knoxville
  - T.U. Dresden, GWT
  - Juelich Supercomputing Center















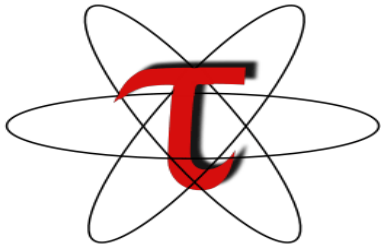




Acknowledgement

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative.

Download TAU from U. Oregon



http://tau.uoregon.edu

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

https://e4s.io [Containers for Extreme-Scale Scientific Software Stack]

Free download, open source, BSD license