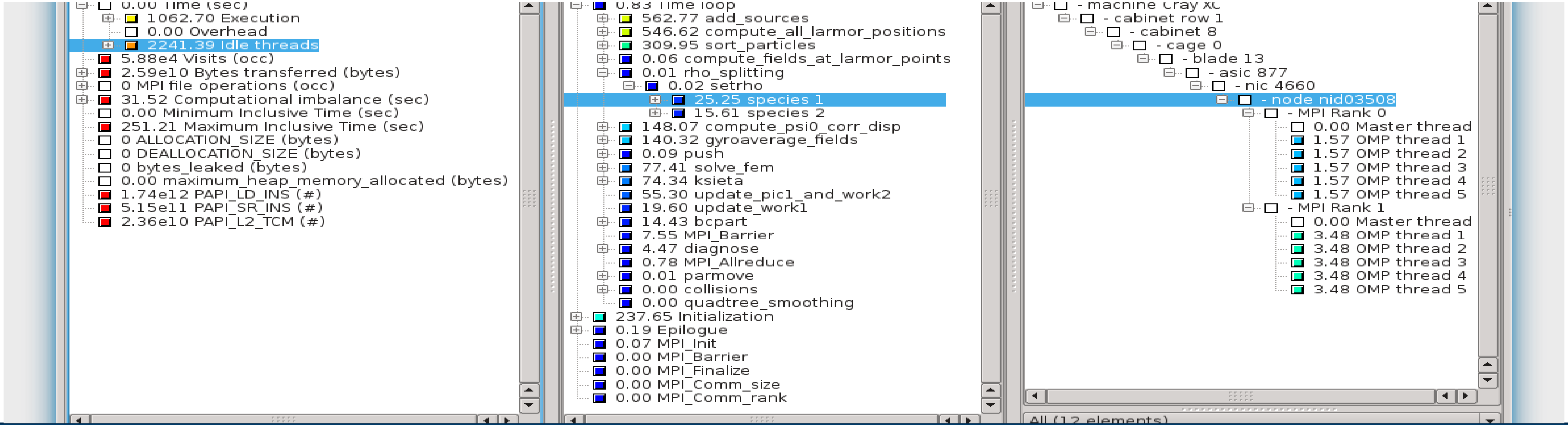
MPI/OPENMP COURSE – SCORE-P

AUGUST 16, 2024 I MICHAEL KNOBLOCH I M.KNOBLOCH@FZ-JUELICH.DE



PERFORMANCE ANALYSIS

USING THE SCORE-P ECOSYSTEM

MOTIVATION

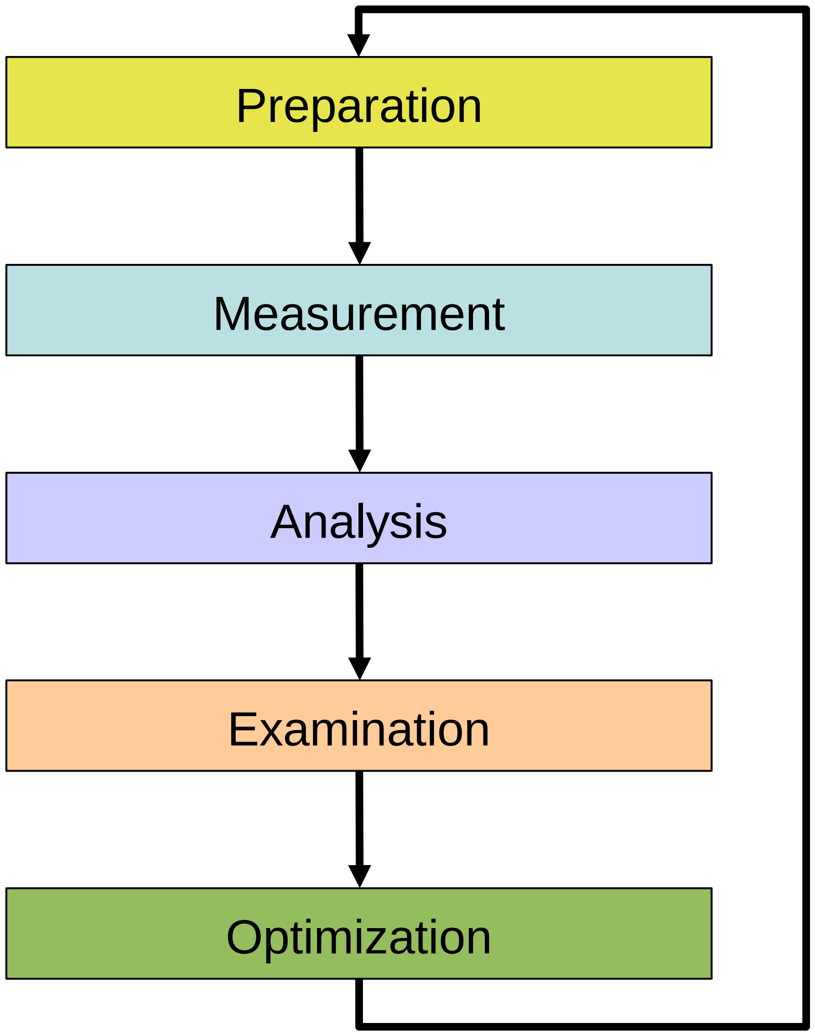
- Writing parallel code is hard
- Writing **fast/efficient** parallel code is even harder
- “Parallel” (multi core/node) performance factors
  - Partitioning / decomposition
    - ☞ Load balancing
  - Communication (i.e., message passing)
  - Multithreading
  - Core binding / NUMA
  - Synchronization / locking
  - I/O
    - ☞Parallel I/O matters

TUNING BASICS

- Carefully set various tuning parameters
  - The right (parallel) algorithms and libraries
  - Compiler flags and directives
  - Correct machine usage (mapping and bindings)
    - ☞Get the most performance before tuning!

- Measurement is better than guessing
  - To determine performance bottlenecks
  - To compare alternatives
  - To validate tuning decisions and optimizations
    - ☞After each step!

PERFORMANCE ENGINEERING WORKFLOW



- Prepare application (with symbols), insert extra code (probes/hooks)
- Collection of data relevant to execution performance analysis
- Calculation of metrics, identification of performance metrics
- Presentation of results in an intuitive/understandable form
- Modifications intended to eliminate/reduce performance problems

THE 80/20 RULE

- Programs typically spend 80% of their time in 20% of the code
  - ☞ *Know what matters!*
- Developers typically spend 20% of their effort to get 80% of the total speedup possible for the application
  - ☞ *Know when to stop!*
- Don't optimize what does not matter
  - ☞ *Make the common case fast!*

PERFORMANCE MEASUREMENT

Two dimensions

When performance measurement is triggered

- **External trigger** (asynchronous)
  - **Sampling**
    - Trigger: Timer interrupt OR Hardware counters overflow
- **Internal trigger** (synchronous)
  - Code **instrumentation** (automatic or manual)

How performance data is recorded

- **Profile**
  - Summation of events over time
- **Trace**
  - Sequence of events over time

Score-P

Scalable performance measurement infrastructure for parallel codes

- Community-developed open-source
- Replaced tool-specific instrumentation and measurement components of partners
- http://www.score-p.org

TECHNISCHE UNIVERSITÄT DRESDEN

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

German Research School for Simulation Sciences

RWTH AACHEN UNIVERSITY

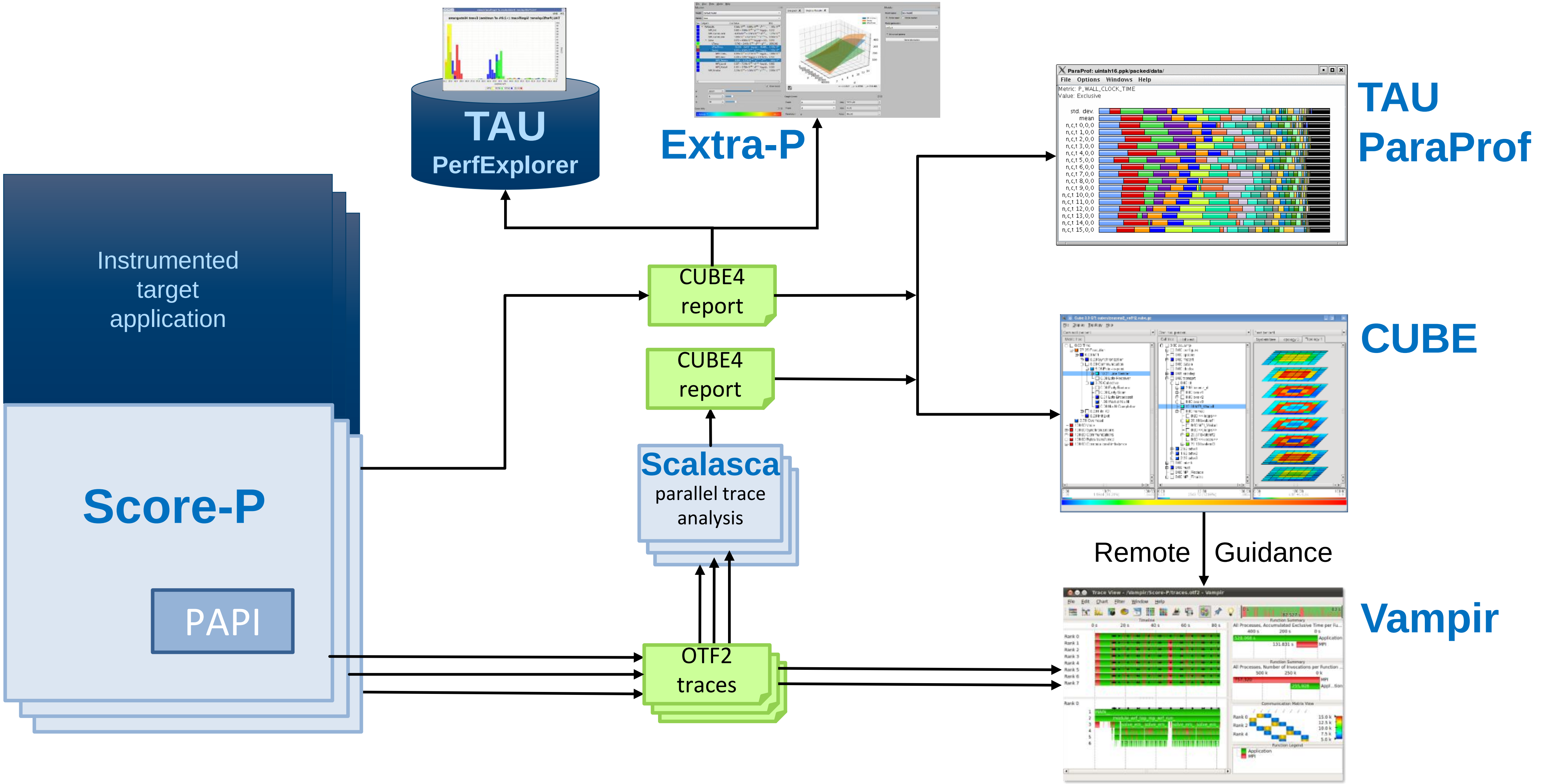
TUM Technische Universität München

UNIVERSITY OF OREGON

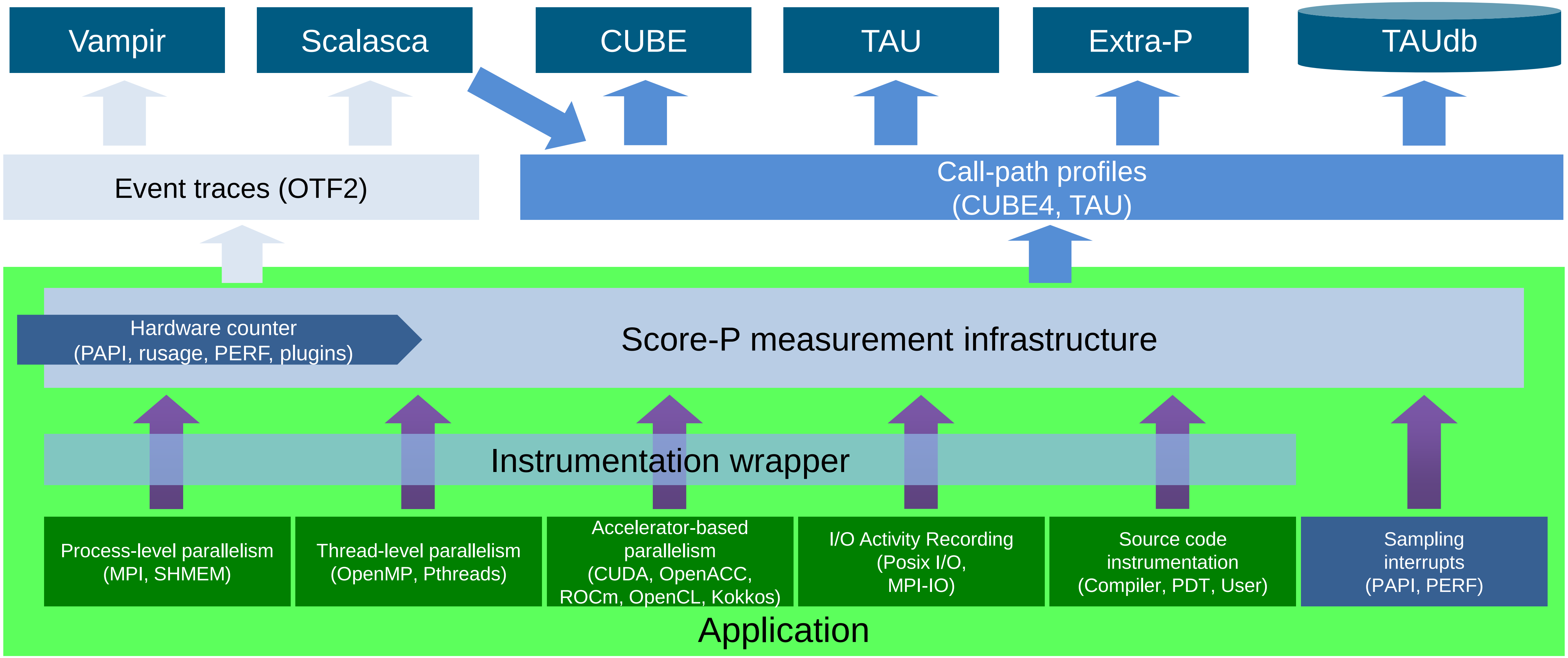
JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Score-P TOOL ECOSYSTEM

JÜLICH Forschungszentrum
JÜLICH SUPERCOMPUTING CENTRE



Score-P ARCHITECTURE

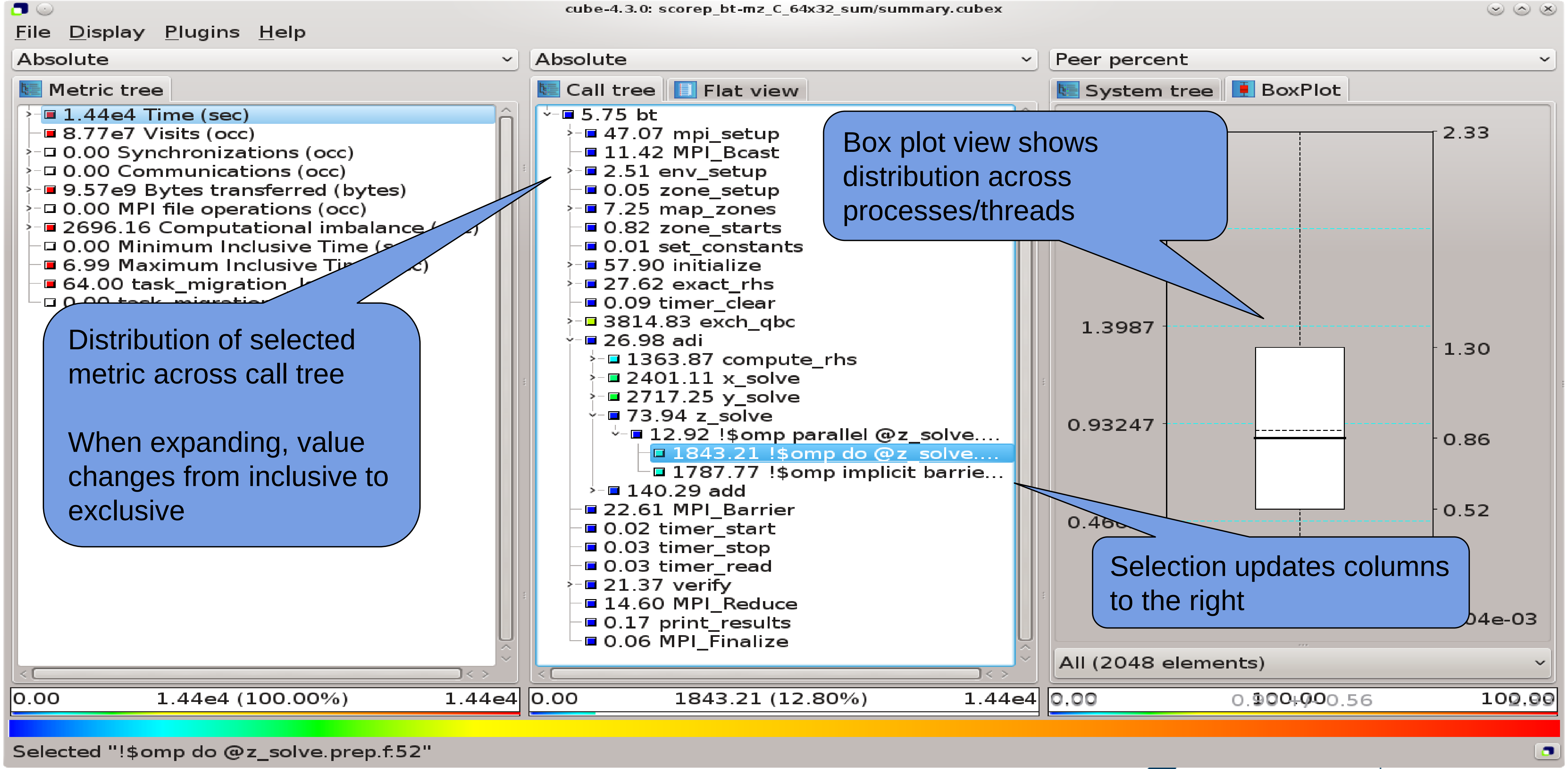


JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Score-P FUNCTIONALITY

- Provide typical functionality for HPC performance tools
- **Instrumentation** (various methods)
  - Multi-process paradigms (MPI, SHMEM)
  - Thread-parallel paradigms (OpenMP, POSIX threads)
  - Accelerator-based paradigms (OpenACC, CUDA, OpenCL. Kokkos)
  - **In any combination!**
- Flexible **measurement** without re-compilation:
  - Basic and advanced **profile** generation (⇨ CUBE4 format)
  - Event **trace** recording (⇨ OTF2 format)
- Highly scalable I/O functionality
- Support all fundamental concepts of partner's tools

CUBE EXAMPLE



SCORE-P: ADVANCED FEATURES

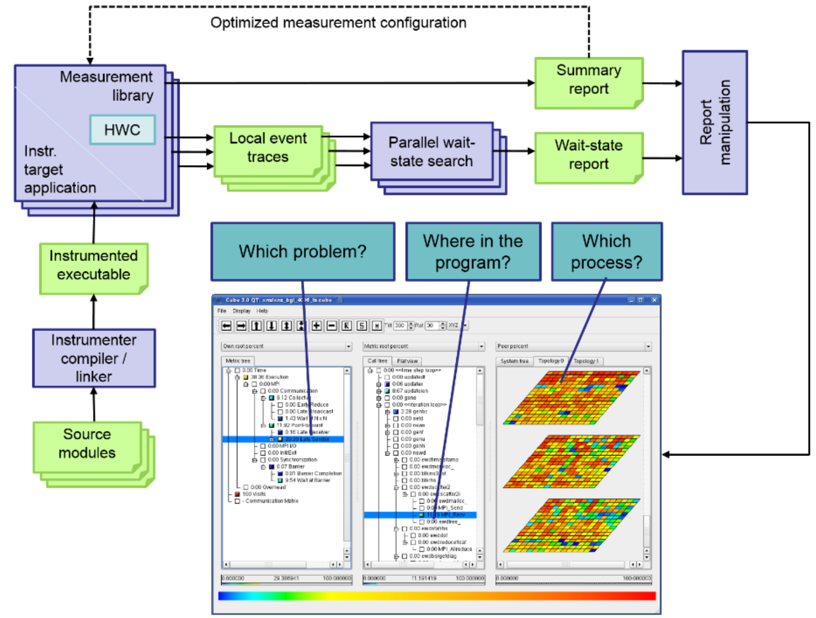
- Measurement can be extensively configured via environment variables
- Allows for targeted measurements:
  - Selective recording
  - Phase profiling
  - Parameter-based profiling
  - …
- GPU support: CUDA, OpenACC, OpenCL, HIP, Kokkos, …
- Please ask us or see the user manual for details

SCALASCA

http://www.scalasca.org/

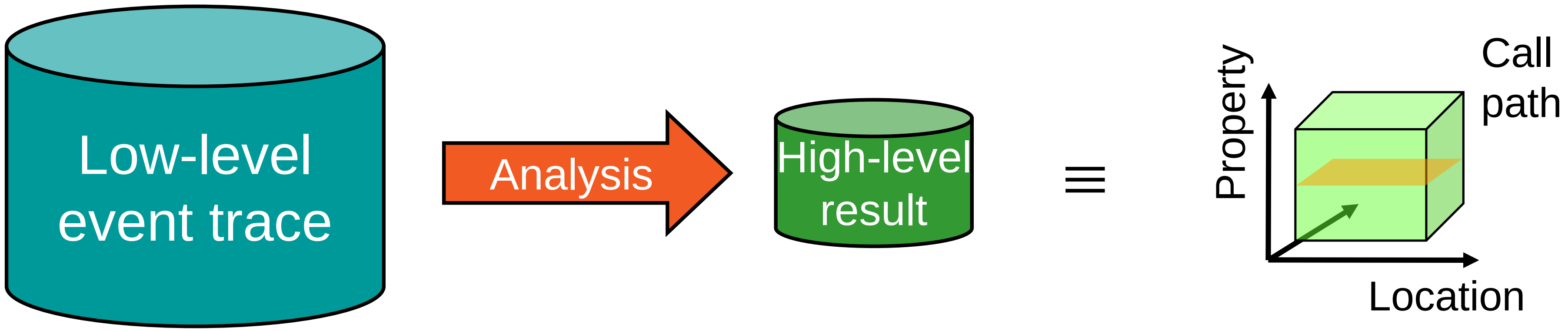
- Scalable Analysis of Large Scale Applications

- Approach
  - **Instrument** C, C++, and Fortran parallel applications (**with Score-P**)
  - Option 1: **scalable call-path profiling**
  - Option 2: **scalable event trace analysis**
    - **Collect** event traces
    - **Process trace in parallel**
      - Wait-state analysis
      - Delay and root-cause analysis
      - Critical path analysis
    - **Categorize and rank** results



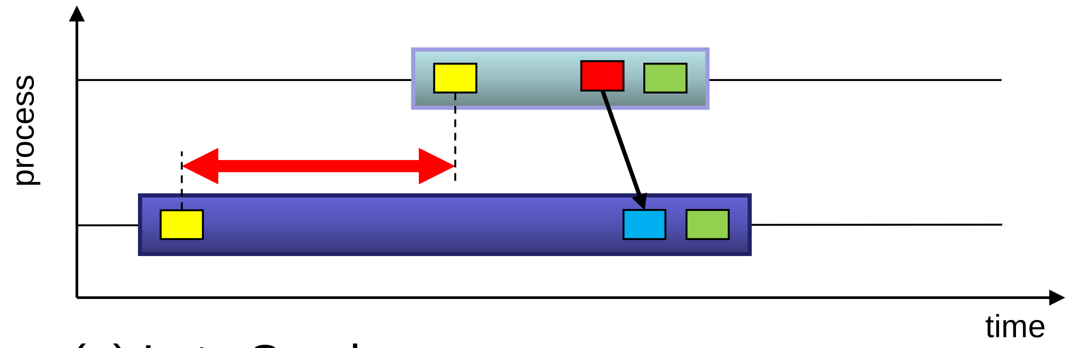
AUTOMATIC TRACE ANALYSIS

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

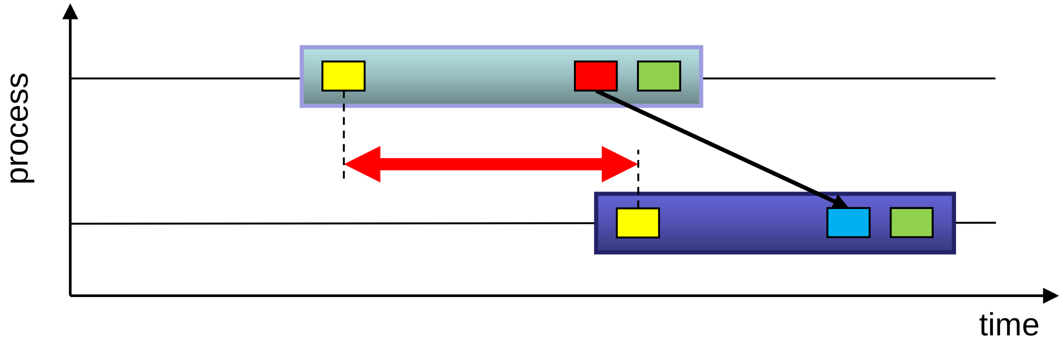


- Guaranteed to cover the entire event trace
- Quicker than manual/visual trace analysis
- Parallel replay analysis exploits available memory & processors to deliver scalability

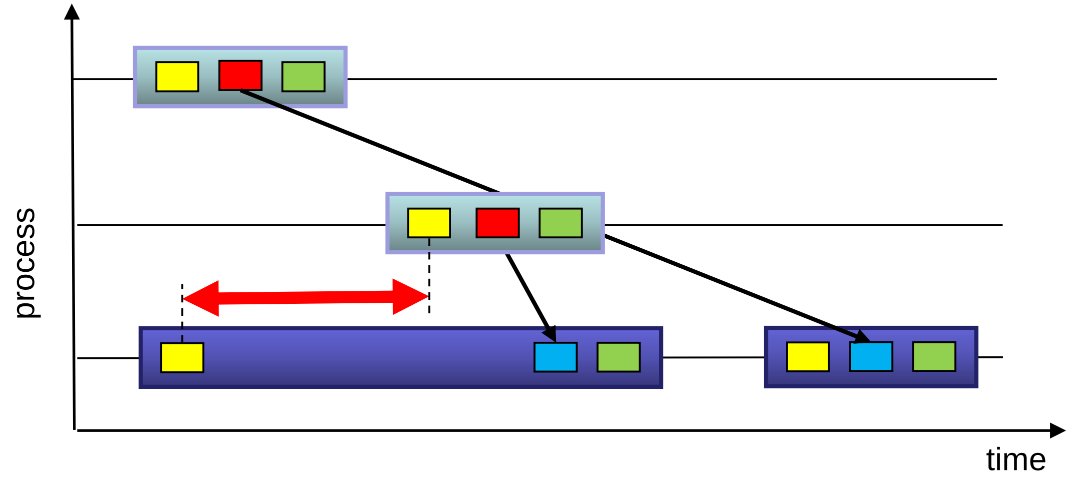
EXAMPLE MPI WAIT STATES



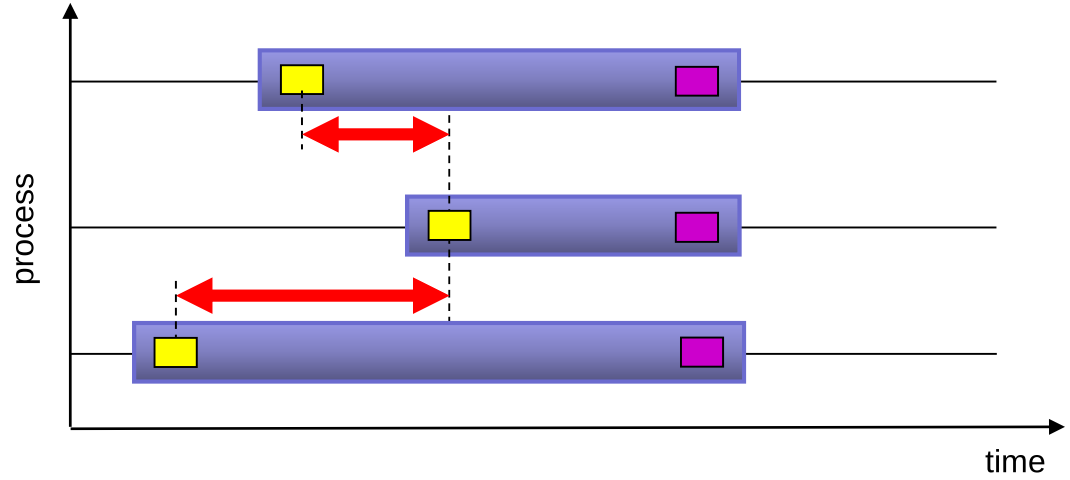
(a) Late Sender



(b) Late Receiver



(c) Late Sender / Wrong Order



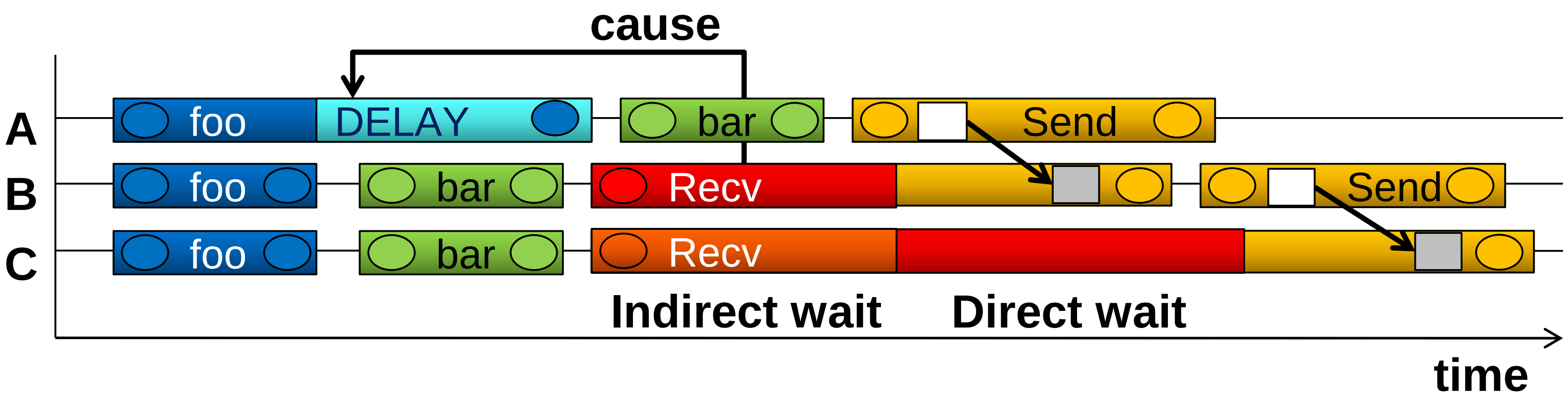
(d) Wait at N x N

ENTER EXIT SEND RECV COLLEXIT

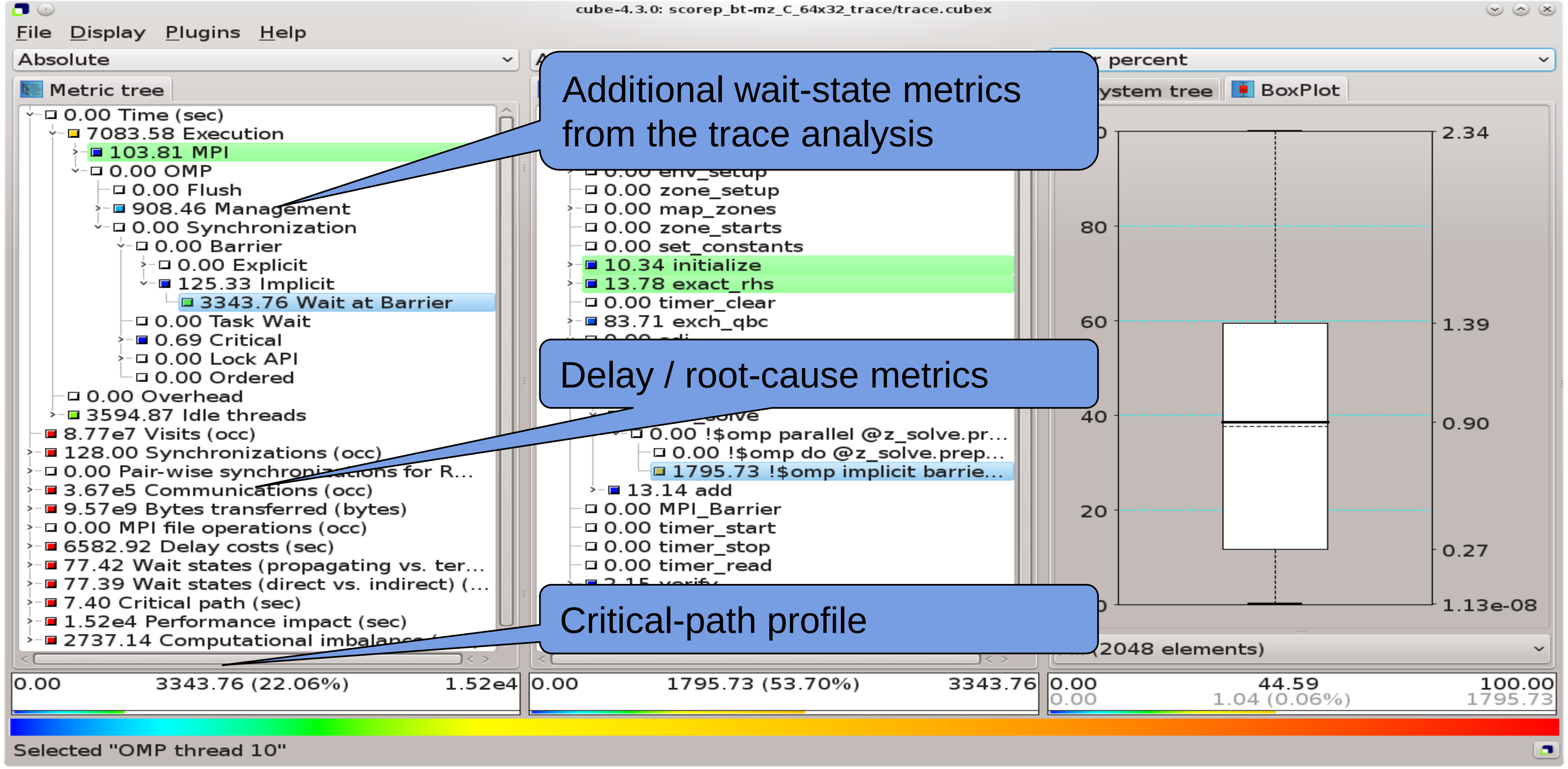
SCALASCA ROOT CAUSE ANALYSIS

- **Root-cause analysis**
  - Wait states typically caused by load or communication imbalances earlier in the program
  - Waiting time can also propagate (e.g., indirect waiting time)
  - Enhanced performance analysis to find the root cause of wait states

- **Approach**
  - Distinguish between direct and indirect waiting time
  - Identify call path/process combinations delaying other processes and causing first order waiting time
  - Identify original **delay**



SCALASCA TRACE ANALYSIS EXAMPLE



VAMPIR EVENT TRACE VISUALIZER

- Offline trace visualization for Score-Ps OTF2 trace files
- Visualization of MPI, OpenMP and application events:
  - All diagrams highly customizable (through context menus)
  - Large variety of displays for ANY part of the trace
- http://www.vampir.eu

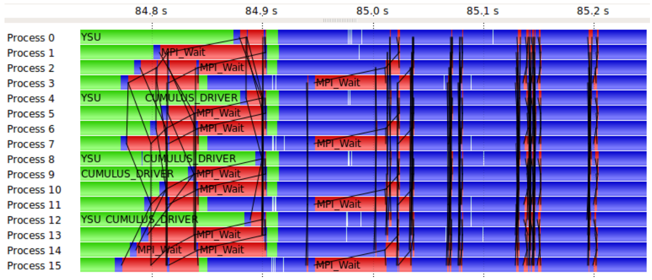
- Advantage:
  - Detailed view of dynamic application behavior
- Disadvantage:
  - Completely manual analysis
  - Too many details can hide the relevant parts

EVENT TRACE VISUALIZATION WITH VAMPIR

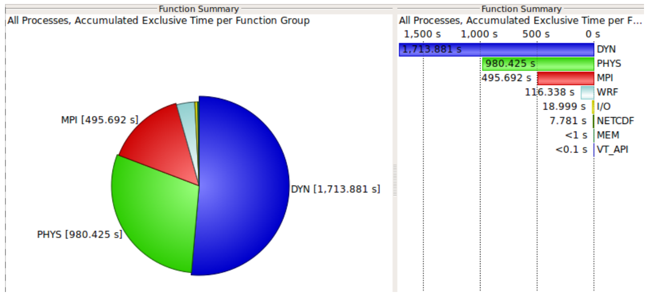
- Visualization of dynamic runtime behaviour at any level of detail along with statistics and performance metrics
- Alternative and supplement to automatic analysis

- **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



VAMPIR PERFORMANCE CHARTS

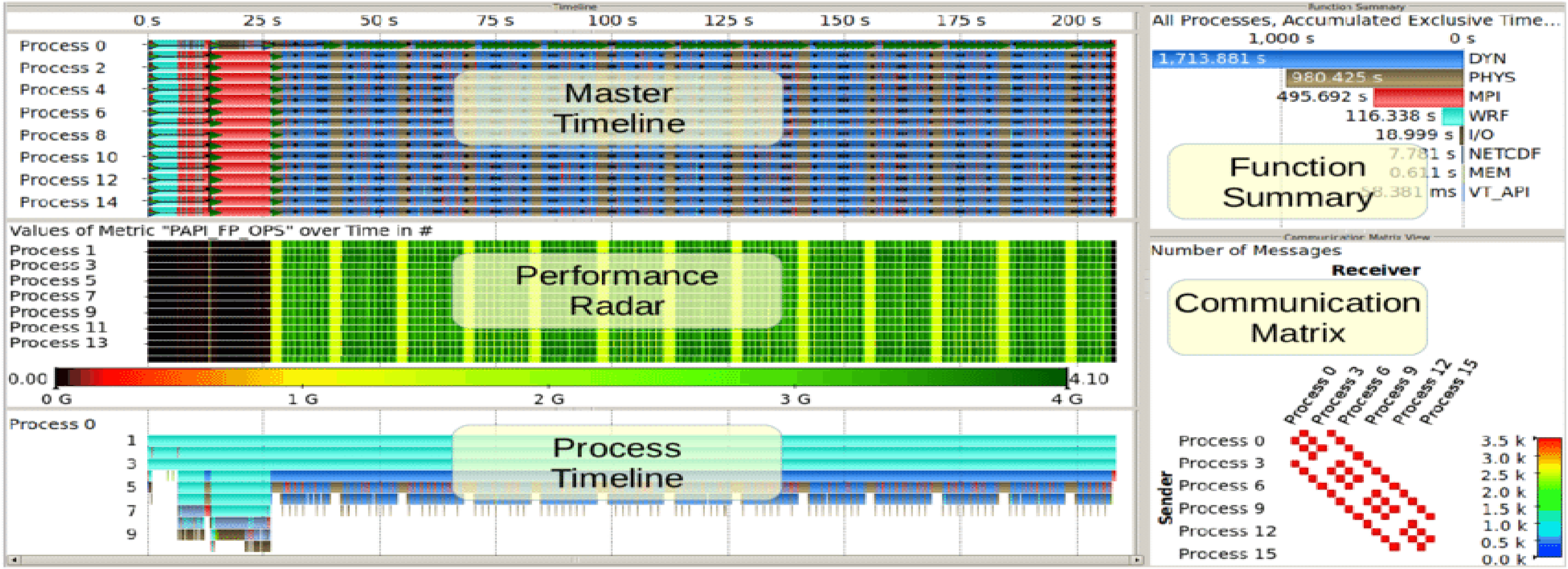
Timeline Charts

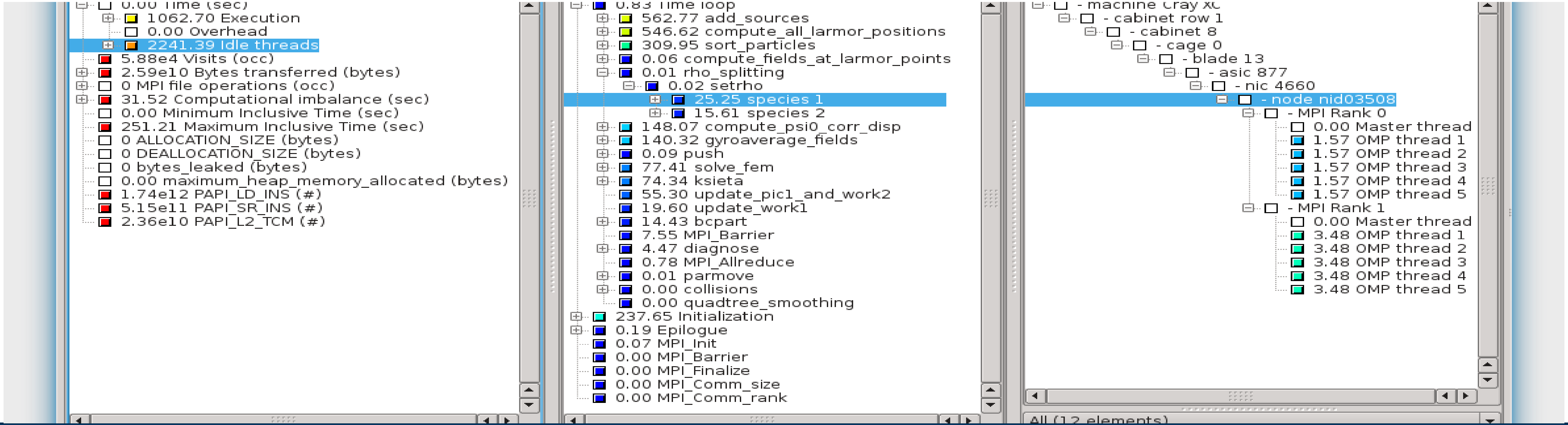
Master Timeline	➡	*all threads' activities*
Process Timeline	➡	*single thread's activities*
Summary Timeline	➡	*all threads' function call statistics*
Performance Radar	➡	*all threads' performance metrics*
Counter Data Timeline	➡	*single threads' performance metrics*
I/O Timeline	➡	*all threads' I/O activities*

Summary Charts

- Function Summary
- Message Summary
- I/O Summary
- Process Summary
- Communication Matrix View
- Call Tree

VAMPIR DISPLAYS





TOOLS DEMO: BT-MZ WITH SCORE-P

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, performance modeling

WHAT IS THE KEY BOTTLENECK?

- Generate flat MPI profile using Score-P/Scalasca
  - Only requires re-linking
  - Low runtime overhead

- Provides detailed information on MPI usage
  - How much time is spent in which operation?
  - How often is each operation called?
  - How much data was transferred?

- Limitations:
  - Computation on non-master threads and outside of MPI_Init/MPI_Finalize scope ignored

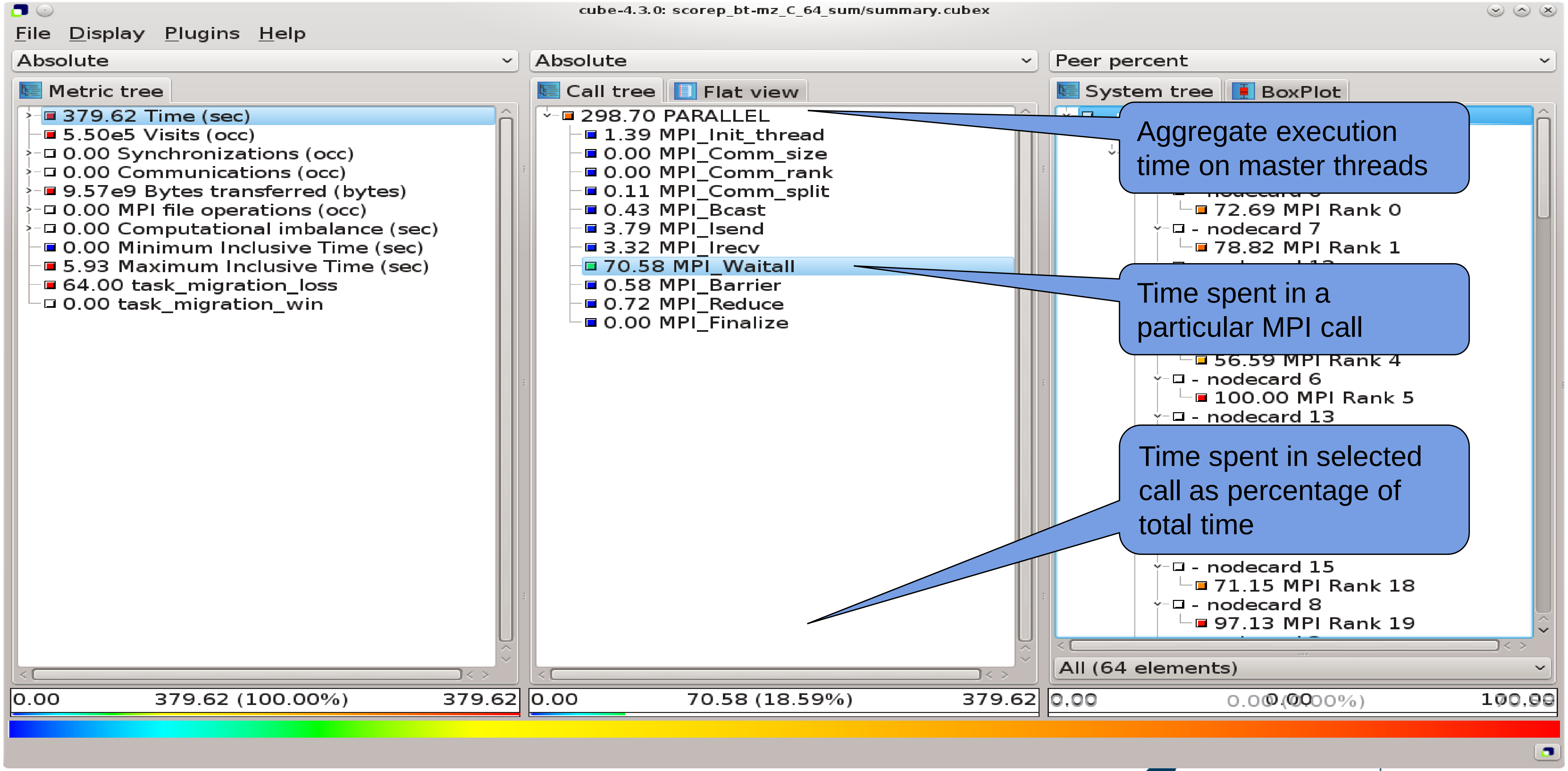
FLAT MPI PROFILE: RECIPE

1. Prefix your ***link command*** with
   “scorep --nocompiler”

2. Prefix your MPI ***launch command*** with
   “scalasca -analyze”

3. After execution, examine analysis results using
   “scalasca -examine scorep_<*title*>”

FLAT MPI PROFILE: EXAMPLE (CONT.)



WHERE IS THE KEY BOTTLENECK?

- Generate call-path profile using Score-P/Scalasca
  - Requires re-compilation
  - Runtime overhead depends on application characteristics
  - Typically needs some care setting up a good measurement configuration
    - Filtering
    - Selective instrumentation

- Option 1 (recommended for beginners):
  Automatic compiler-based instrumentation

- Option 2 (for in-depth analysis):
  Manual instrumentation of interesting phases, routines, loops

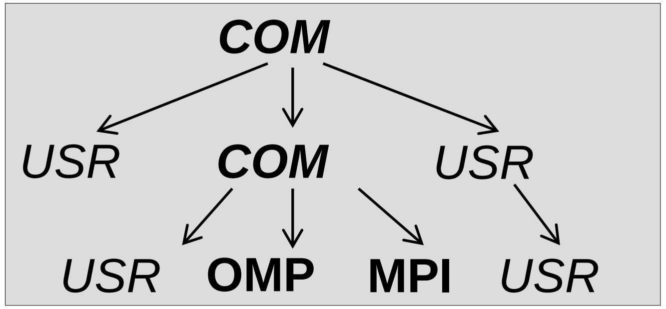
CALL-PATH PROFILE: RECIPE

1. Prefix your ***compile & link commands*** with
   “`scorep`”
2. Prefix your MPI ***launch command*** with
   “`scalasca -analyze`”
3. After execution, compare overall runtime with uninstrumented run to determine overhead
4. If overhead is too high
   1. Score measurement using
      “`scalasca -examine -s scorep_<title>`”
   2. Prepare filter file
   3. Re-run measurement with filter applied using prefix
      “`scalasca –analyze –f <filter_file>`”
5. After execution, examine analysis results using
   “`scalasca -examine scorep_<title>`”

CALL-PATH PROFILE: EXAMPLE (CONT.)

```
% scalasca -examine -s scorep_myprog_Ppnxt_sum
scorep-score -r ./scorep_myprog_Ppnxt_sum/profile.cubex
INFO: Score report written to ./scorep_myprog_Ppnxt_sum/scorep.score
```

- Estimates trace buffer requirements
- Allows to identify candidate functions for filtering
  ☞ Computational routines with high visit count and low time-per-visit ratio
- Region/call-path classification
  - MPI (pure MPI library functions)
  - OMP (pure OpenMP functions/regions)
  - USR (user-level source local computation
  - COM ("combined" USR + OpeMP/MPI)
  - ANY/ALL (aggregate of all region types)



CALL-PATH PROFILE: EXAMPLE (CONT.)

```
% less scorep_myprog_Ppnxt_sum/scorep.score
Estimated aggregate size of event trace:                   162GB
Estimated requirements for largest trace buffer (max_buf): 2758MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       2822MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=2822MB to avoid
 intermediate flushes or reduce requirements using USR regions
 filters.)

flt type    max_buf[B]         visits  time[s] time[%] time/      region
                                                       visit[us]
     ALL 2,891,417,902 6,662,521,083 36581.51   100.0       5.49  ALL
     USR 2,858,189,854 6,574,882,113 13618.14    37.2       2.07  USR
     OMP    54,327,600    86,353,920 22719.78    62.1     263.10  OMP
     MPI       676,342       550,010   208.98     0.6     379.96  MPI
     COM       371,930       735,040    34.61     0.1      47.09  COM

     USR   921,918,660 2,110,313,472  3290.11     9.0       1.56  matmul_sub
     USR   921,918,660 2,110,313,472  5914.98    16.2       2.80  binvcrhs
     USR   921,918,660 2,110,313,472  3822.64    10.4       1.81  matvec_sub
     USR    41,071,134    87,475,200   358.56     1.0       4.10  lhsinit
     USR    41,071,134    87,475,200   145.42     0.4       1.66  binvrhs
     USR    29,194,256    68,892,672    86.15     0.2       1.25  exact_solution
     OMP     3,280,320     3,293,184    15.81     0.0       4.80  !$omp parallel
     [...]
```

CALL-PATH PROFILE: FILTERING

- In this example, the 6 most fequently called routines are of type USR
- These routines contribute around 35% of total time
  - However, much of that is most likely measurement overhead
    - Frequently executed
    - Time-per-visit ratio in the order of a few microseconds

☞Avoid measurements to reduce the overhead

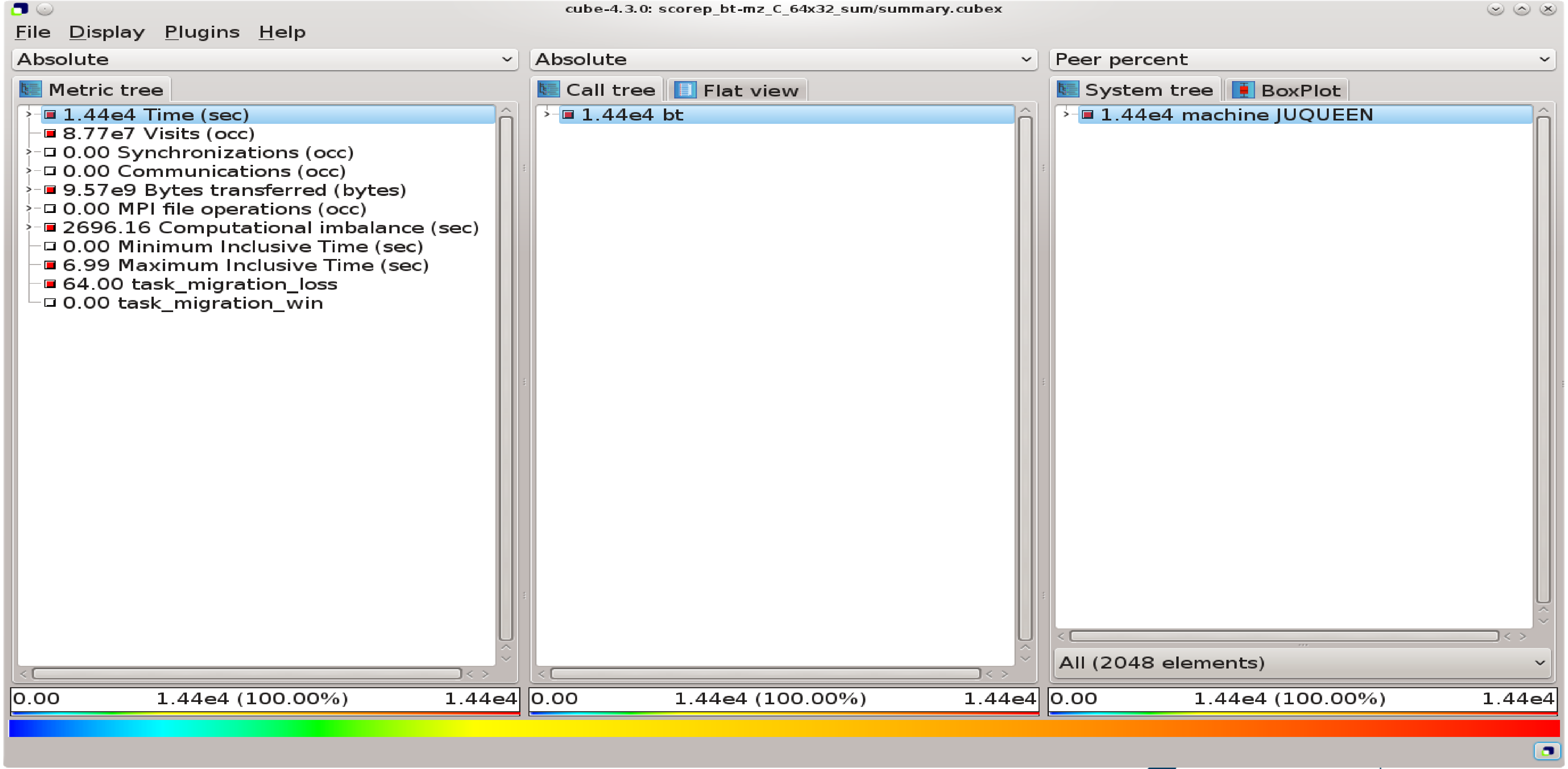
☞List routines to be filtered in simple text file

FILTERING: EXAMPLE

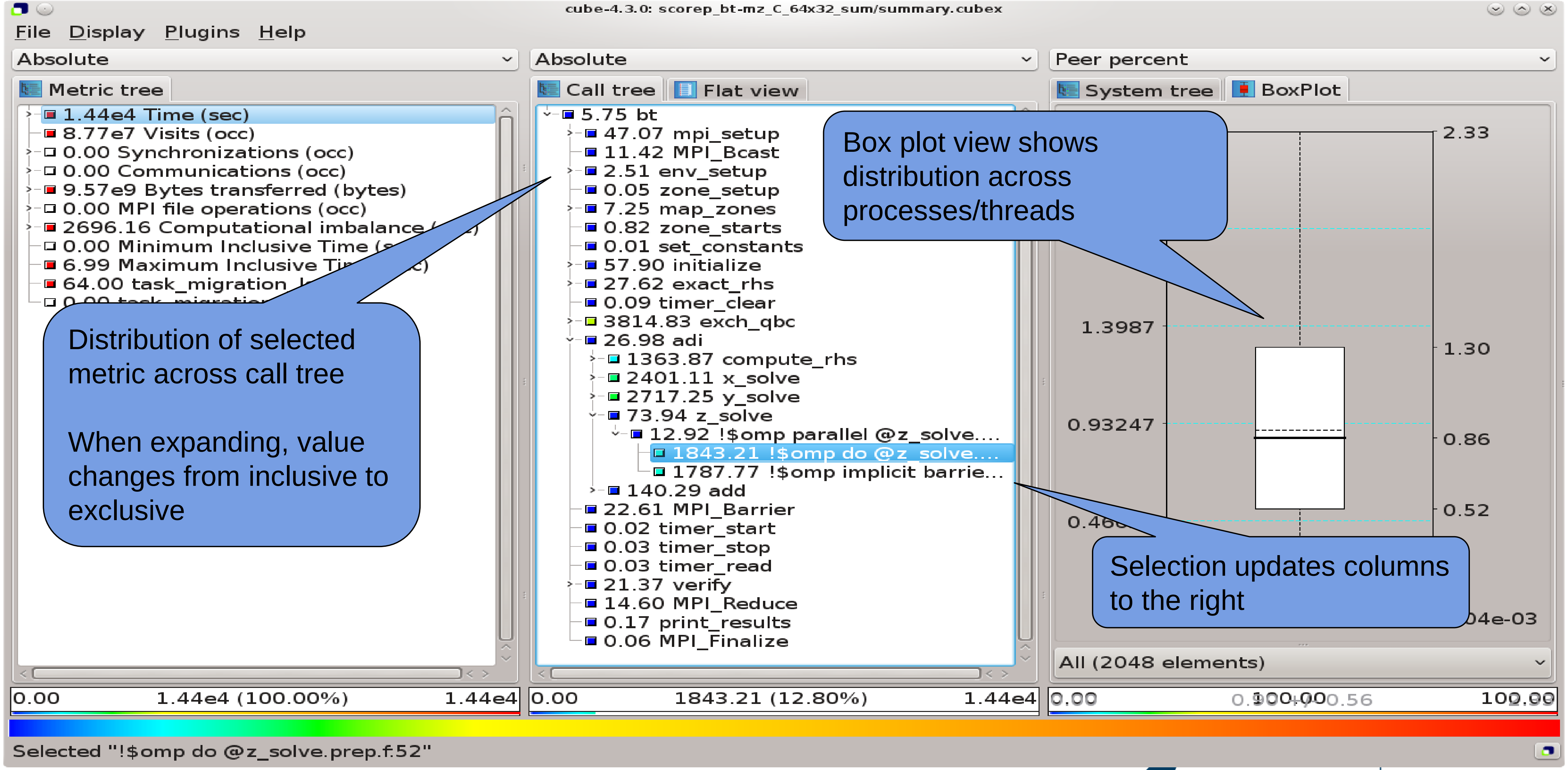
```
% cat filter.txt
SCOREP_REGION_NAMES_BEGIN
    EXCLUDE
        binvcrhs
        matmul_sub
        matvec_sub
        binvrhs
        lhsinit
        exact_solution
SCOREP_REGION_NAMES_END
```

- Score-P filtering files support
  - Wildcards (shell globs)
  - Blacklisting
  - Whitelisting
  - Filtering based on filenames

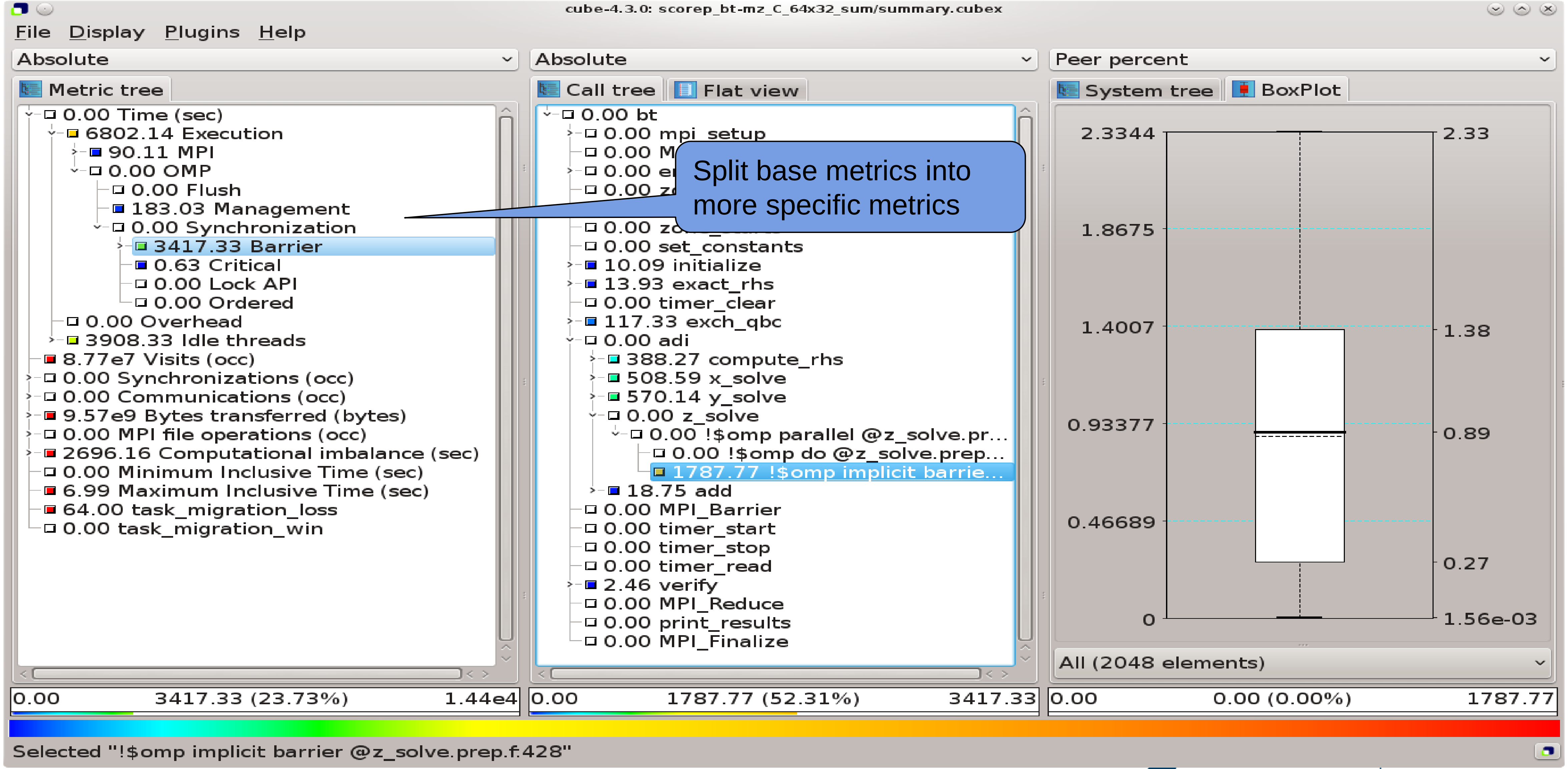
CALL-PATH PROFILE: EXAMPLE (CONT.)



CALL-PATH PROFILE: EXAMPLE (CONT.)



CALL-PATH PROFILE: EXAMPLE (CONT.)



WHY IS THE BOTTLENECK THERE?

- This is highly application dependent!
- Might require additional measurements
  - Hardware-counter analysis
    - CPU utilization
    - Cache behavior
  - Selective instrumentation
  - Automatic/manual event trace analysis

HARDWARE COUNTERS

- Counters: set of registers that count processor events, e.g. floating point operations or cycles
- Number of registers, counters and simultaneously measurable events vary between platforms
- Can be measured by:
  - perf:
    - Integrated in Linux since Kernel 2.6.31
    - Library and CLI
  - LIKWID:
    - Direct access to MSRs (requires Kernel module)
    - Consists of multiple tools and an API
  - PAPI (Performance API)

PAPI

- Portable API: Uses the same routines to access counters across all supported architectures
- Used by most performance analysis tools

- High-level interface:
  - Predefined standard events, e.g. PAPI_FP_OPS
  - Availability and definition of events varies between platforms
  - List of available counters: `papi_avail (-d)`
- Low-level interface:
  - Provides access to all machine specific counters
  - Non-portable
  - More flexible
  - List of available counters: `papi_native_avail`

TRACE GENERATION & ANALYSIS W/ SCALASCA

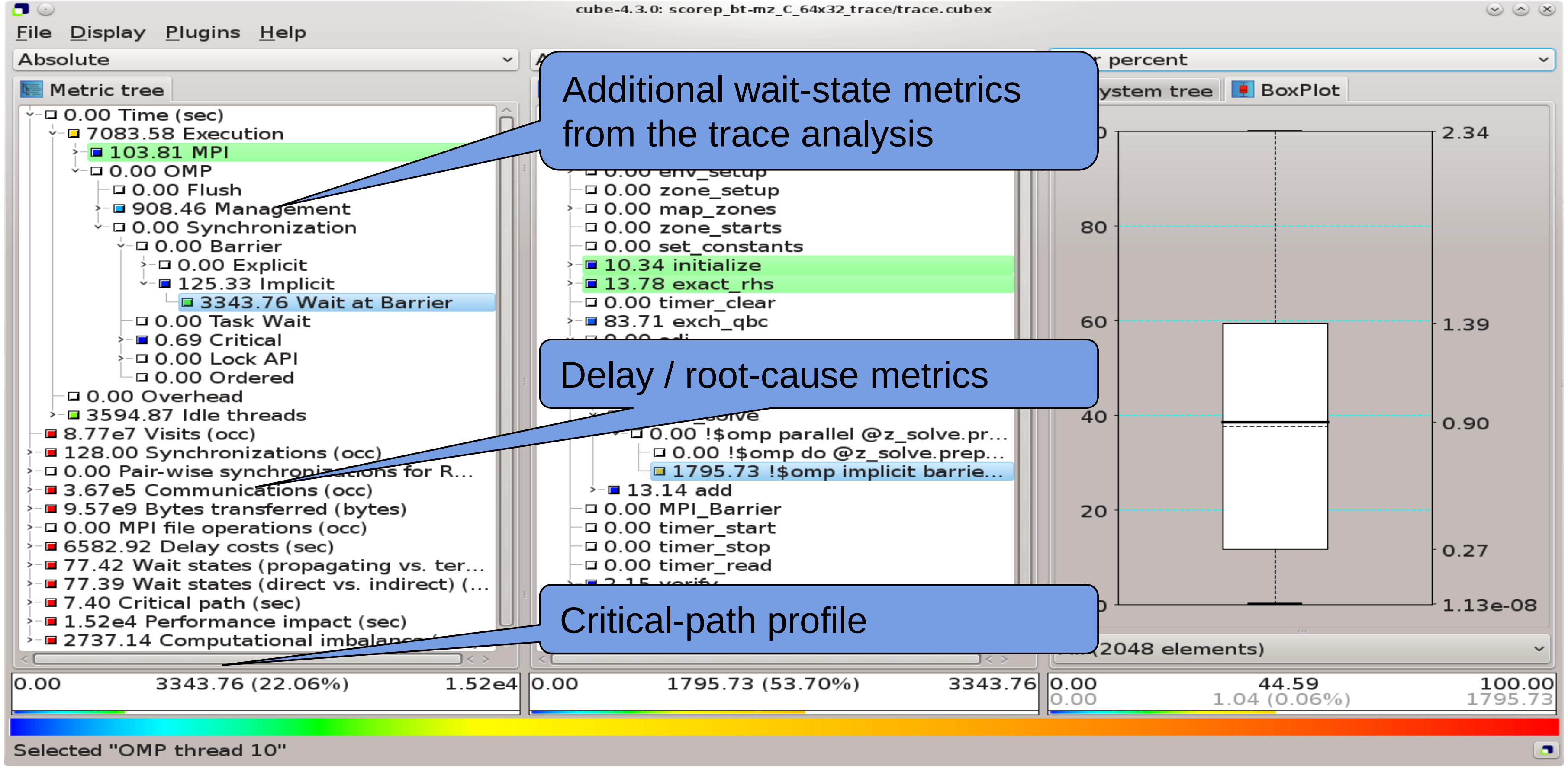
- Enable trace collection & analysis using "`-t`" option of "`scalasca -analyze`":

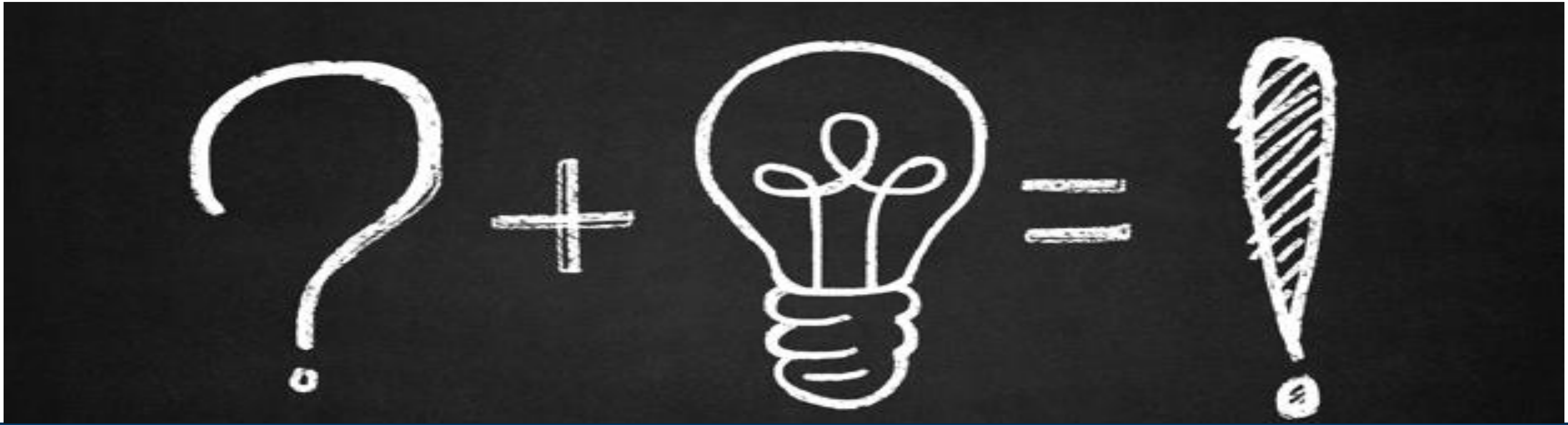
```
##########################
##  In the job script:  ##
##########################

module load ENV Score-P Scalasca
export SCOREP_TOTAL_MEMORY=120MB   # Consult score report
scalasca -analyze -f filter.txt -t \
    srun -n n [...] ./myprog
```

- **ATTENTION:**
  - Traces can quickly become extremely large!
  - Remember to use proper filtering, selective instrumentation, and Score-P memory specification
  - Before flooding the file system, **ask us for assistance!**

SCALASCA TRACE ANALYSIS EXAMPLE





QUESTIONS