EPICURE GPU Hackathon
Tuning Basics and Tools Overview: Score-P / Cube / Vampir / Scalasca

Ilya Zhukov
Jülich Supercomputing Centre

Performance factors of parallel applications

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

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

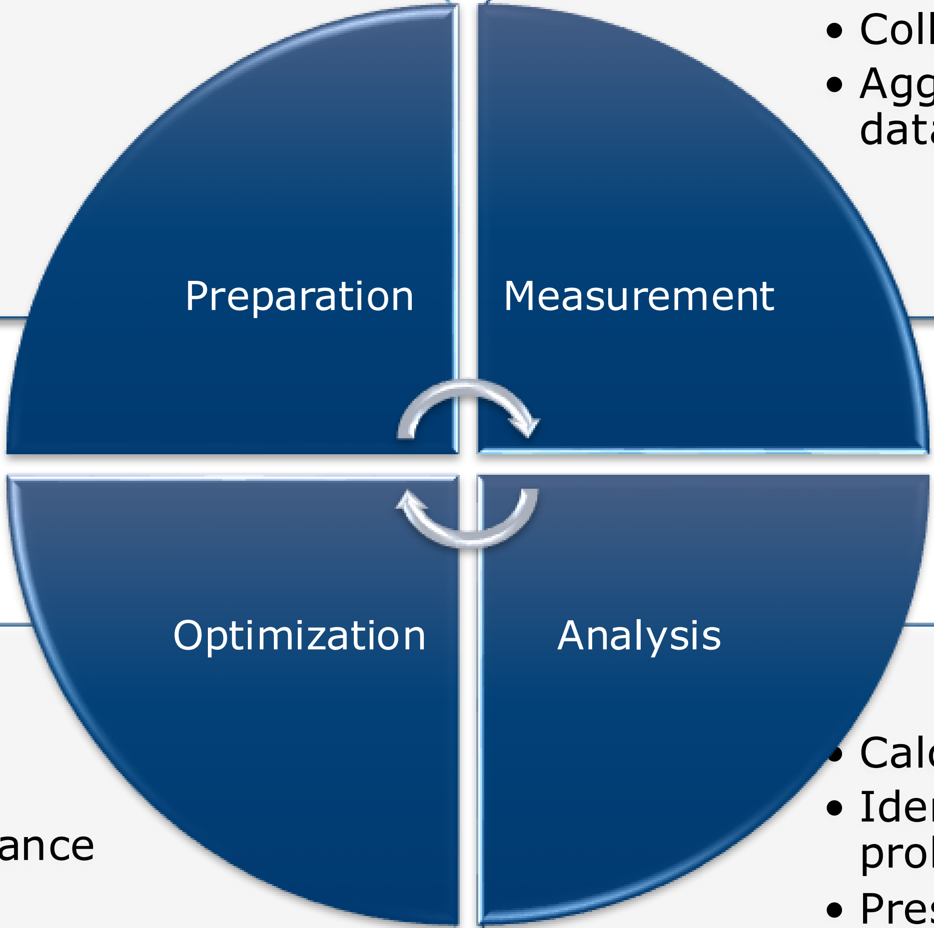
Tuning basics

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

Performance engineering workflow

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

- Collection of performance data
- Aggregation of performance data



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

- Modifications intended to eliminate/reduce performance problem

The 80/20 rule

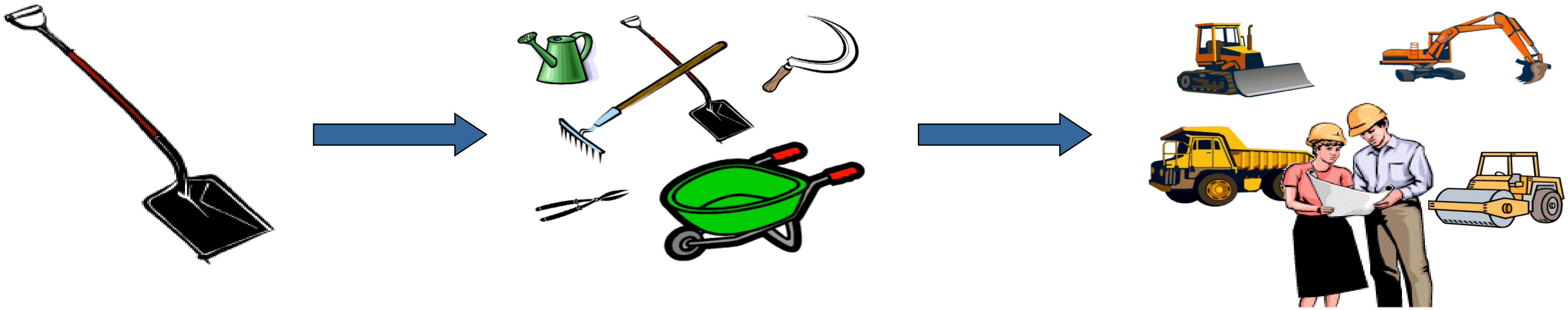
- Programs typically spend 80% of their time in 20% of the code

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

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

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

No single solution is sufficient!



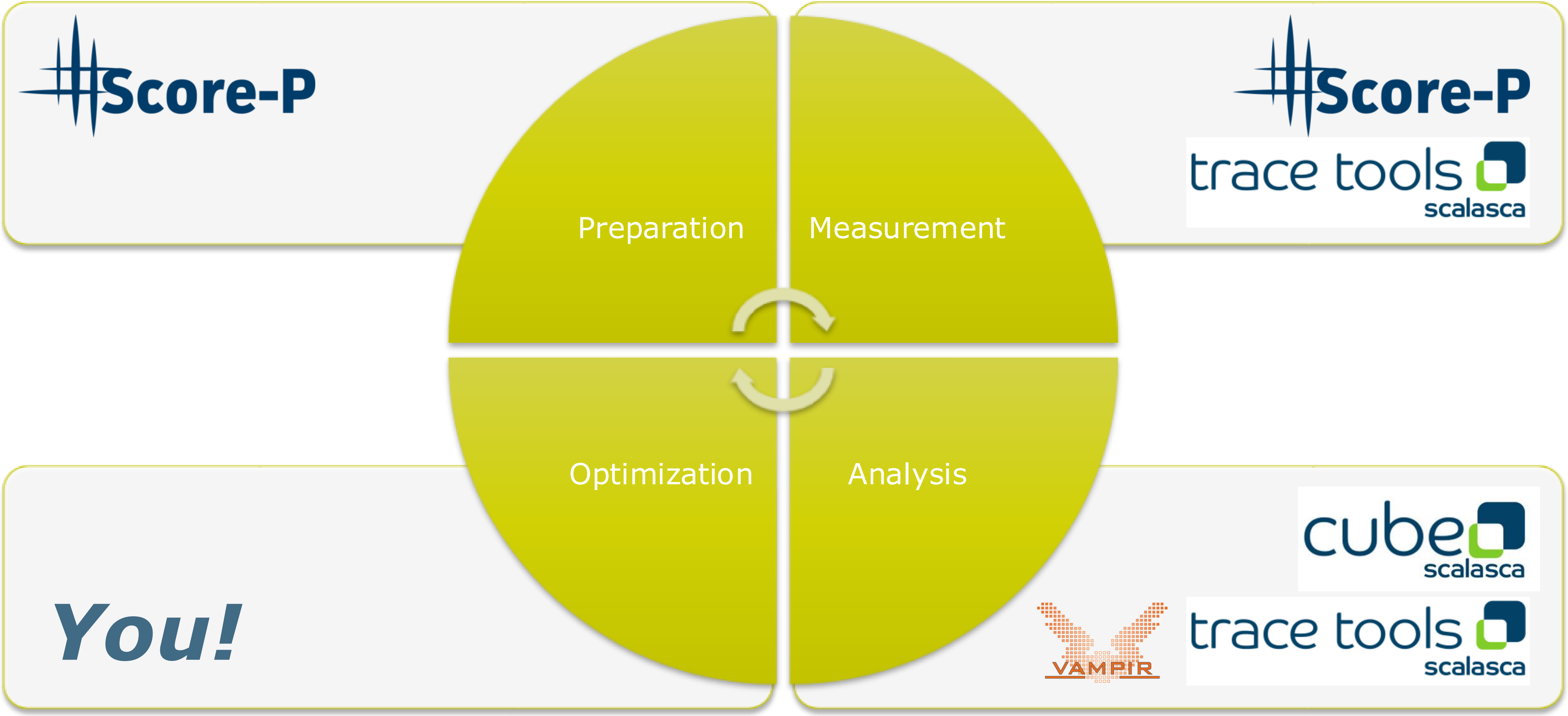
A combination of different methods, tools and techniques is typically needed!

- 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 / how near to limits
- 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 engineering workflow



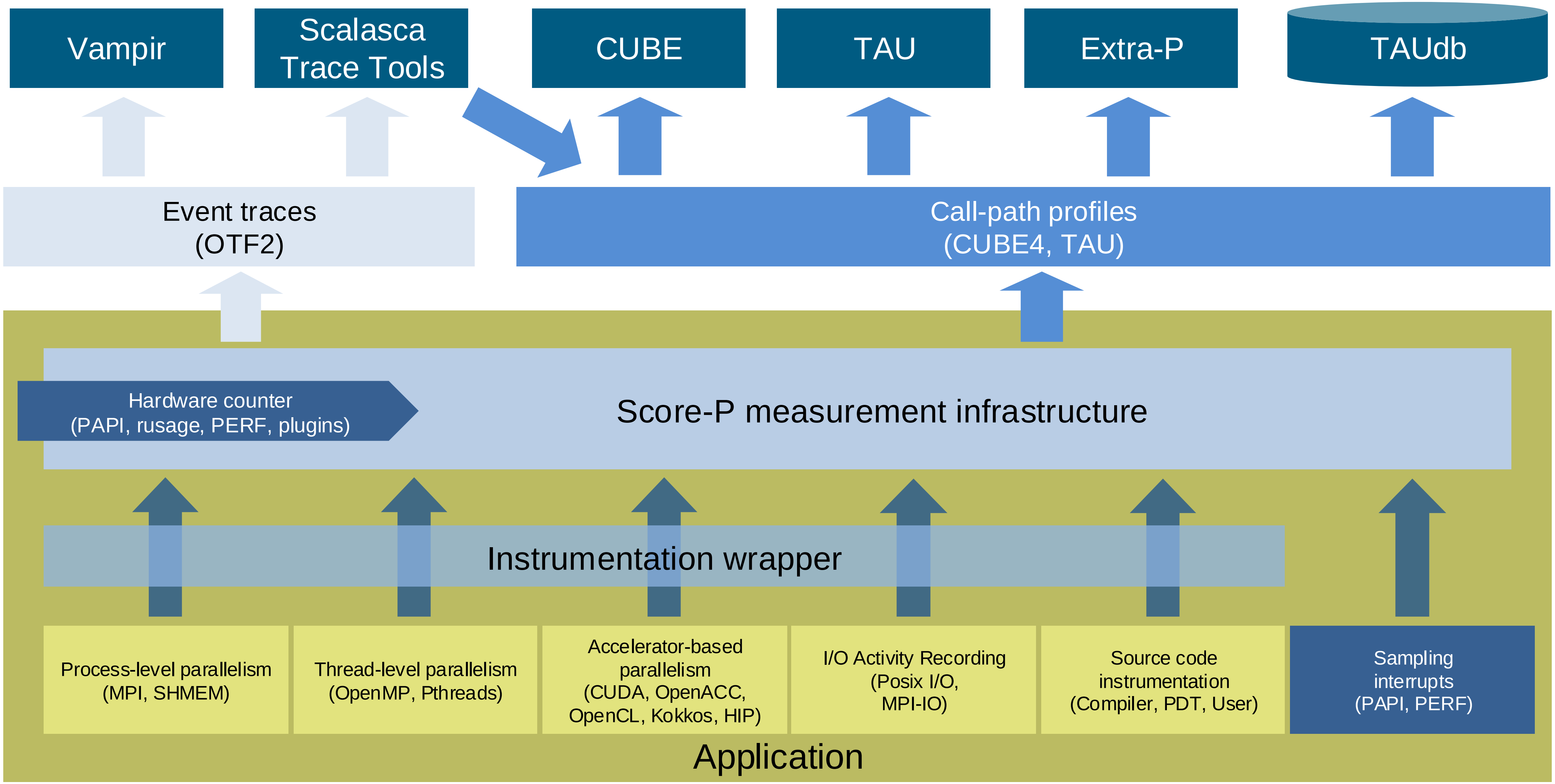
Score-P

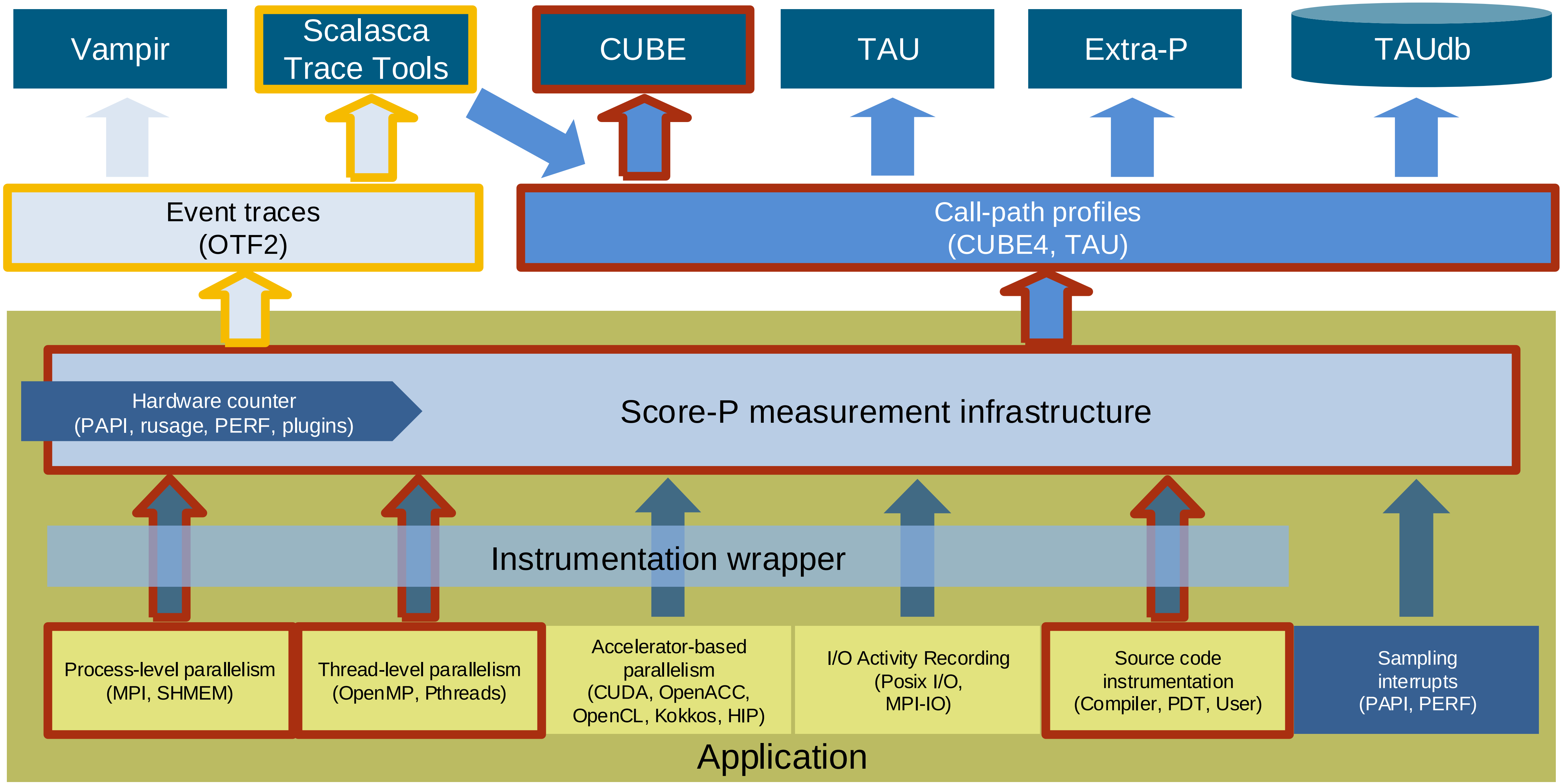
DOI 10.5281/zenodo.1240731

- 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, POSIX threads
  - Accelerator-based: CUDA, HIP, OpenCL, OpenACC

- Initial project funded by BMBF
- Close collaboration with PRIMA project funded by DOE
- Further developed in multiple 3rd-party funded projects

GEFÖRDERT VOM
Bundesministerium für Bildung und Forschung


Vampir
Scalasca Trace Tools
CUBE
TAU
Extra-P
TAUdb
Event traces (OTF2)
Call-path profiles (CUBE4, TAU)
Hardware counter (PAPI, rusage, PERF, plugins)
Score-P measurement infrastructure
Instrumentation wrapper
Process-level parallelism (MPI, SHMEM)
Thread-level parallelism (OpenMP, Pthreads)
Accelerator-based parallelism (CUDA, OpenACC, OpenCL, Kokkos, HIP)
I/O Activity Recording (Posix I/O, MPI-IO)
Source code instrumentation (Compiler, PDT, User)
Sampling interrupts (PAPI, PERF)
Application


Vampir
Scalasca Trace Tools
CUBE
TAU
Extra-P
TAUdb
Event traces (OTF2)
Call-path profiles (CUBE4, TAU)
Hardware counter (PAPI, rusage, PERF, plugins)
Score-P measurement infrastructure
Instrumentation wrapper
Process-level parallelism (MPI, SHMEM)
Thread-level parallelism (OpenMP, Pthreads)
Accelerator-based parallelism (CUDA, OpenACC, OpenCL, Kokkos, HIP)
I/O Activity Recording (Posix I/O, MPI-IO)
Source code instrumentation (Compiler, PDT, User)
Sampling interrupts (PAPI, PERF)
Application

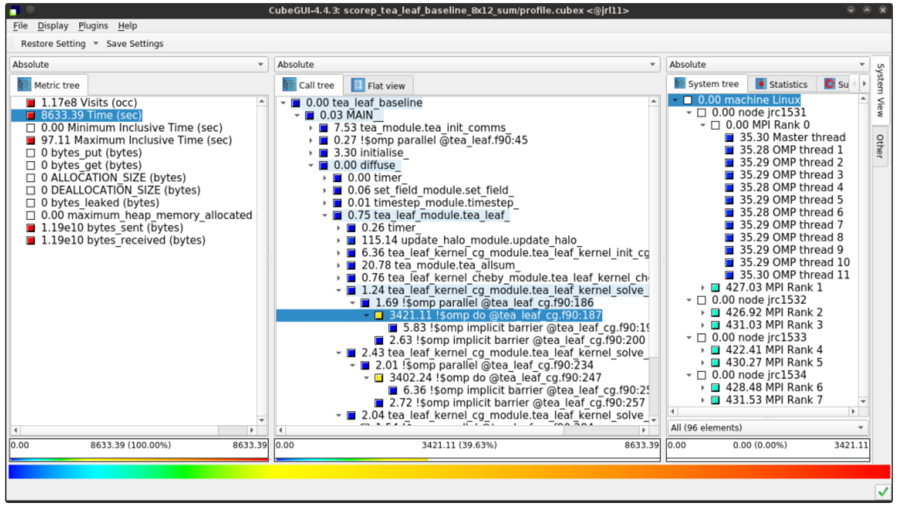
Score-P features

- Open source: 3-clause BSD license
  - Commitment to joint long-term cooperation
  - Development based on meritocratic governance model
  - Open for contributions and new partners
- Portability: supports all major HPC platforms
- Scalability: successful measurements with >1M threads
- Functionality:
  - Generation of call-path profiles and event traces (supporting highly scalable I/O)
  - Using direct instrumentation and sampling
  - Flexible measurement configuration without re-compilation
  - Recording of time, visits, communication data, hardware counters
  - Support for MPI, SHMEM, OpenMP, Pthreads, CUDA, HIP, OpenCL, OpenACC and valid combinations
- Latest release: Score-P 8.4 (Mar 2024)

cube scalasca

CubeLib DOI 10.5281/zenodo.1248078
CubeGUI DOI 10.5281/zenodo.1248087

- 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 ≥ 5



- Originally developed as part of the Scalasca toolset

- Now available as separate components
  - Can be installed independently of Score-P and Scalasca, e.g., on laptop/desktop
  - Latest releases: Cube v4.8.2 (Sep 2023)

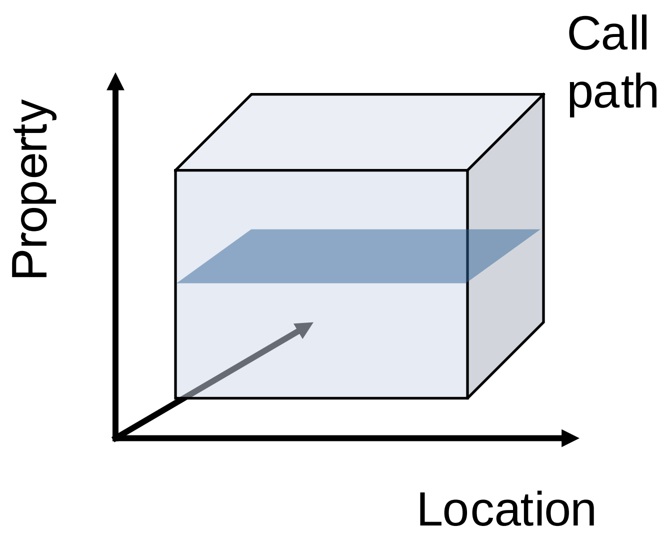
Note: source distribution tarballs for Linux, as well as binary packages provided for Linux, Windows & MacOS, from **www.scalasca.org** website in Software/Cube 4.x

Analysis presentation and exploration

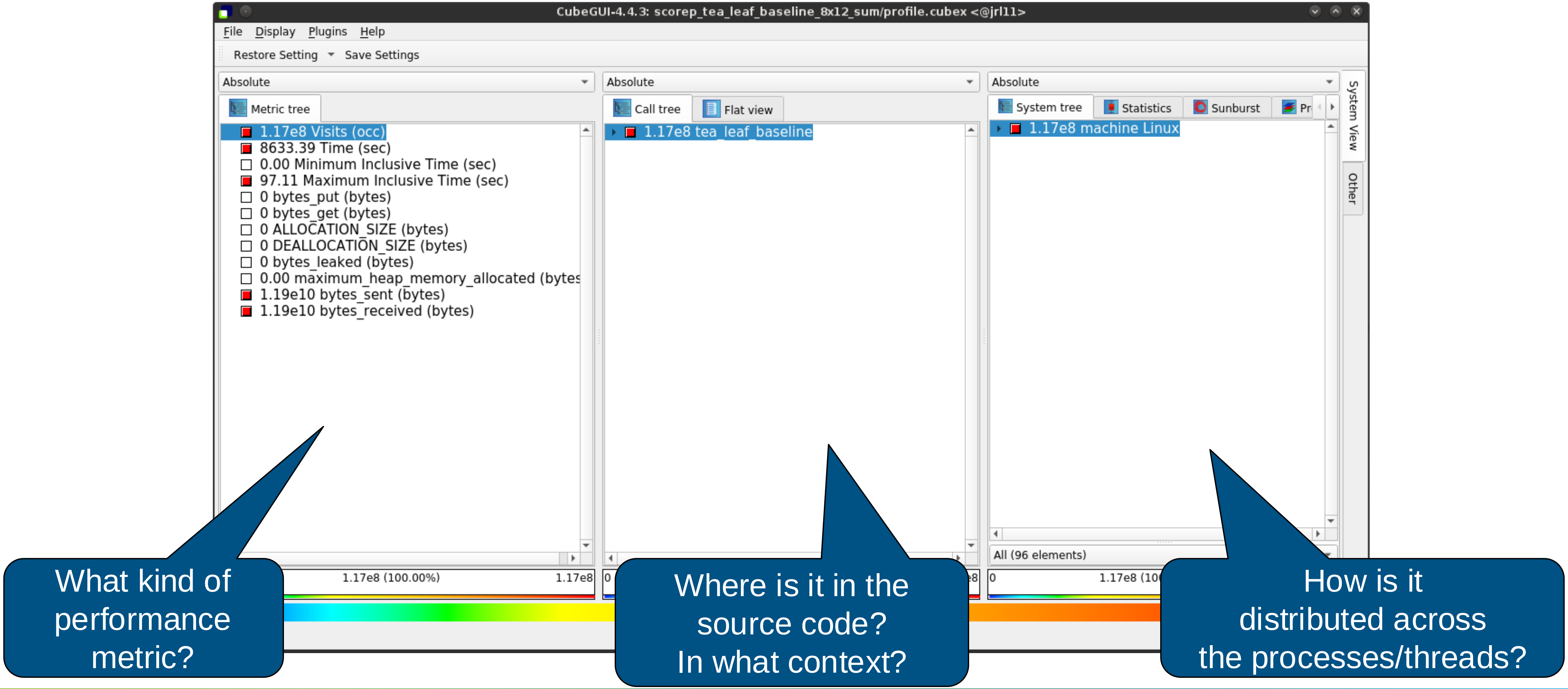
- Representation of values (severity matrix) on three hierarchical axes
  - Performance property (metric)
  - Call path (program location)
  - System location (process/thread)

- Three coupled tree browsers

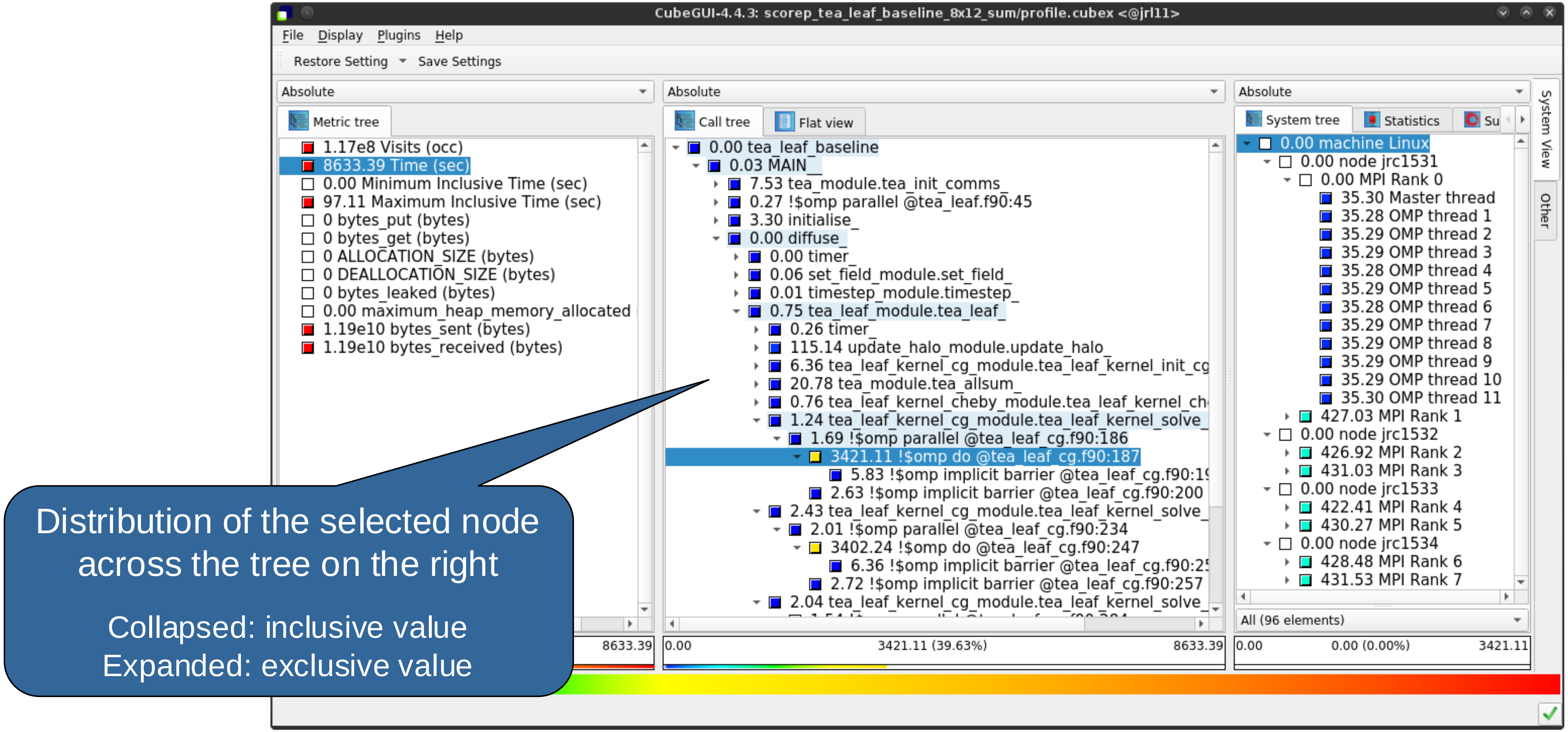
- Cube displays severities
  - As value: for precise comparison
  - As color: for easy identification of hotspots
  - Inclusive value when closed & exclusive value when expanded
  - Customizable via display modes



Plain summary analysis report (opening view)

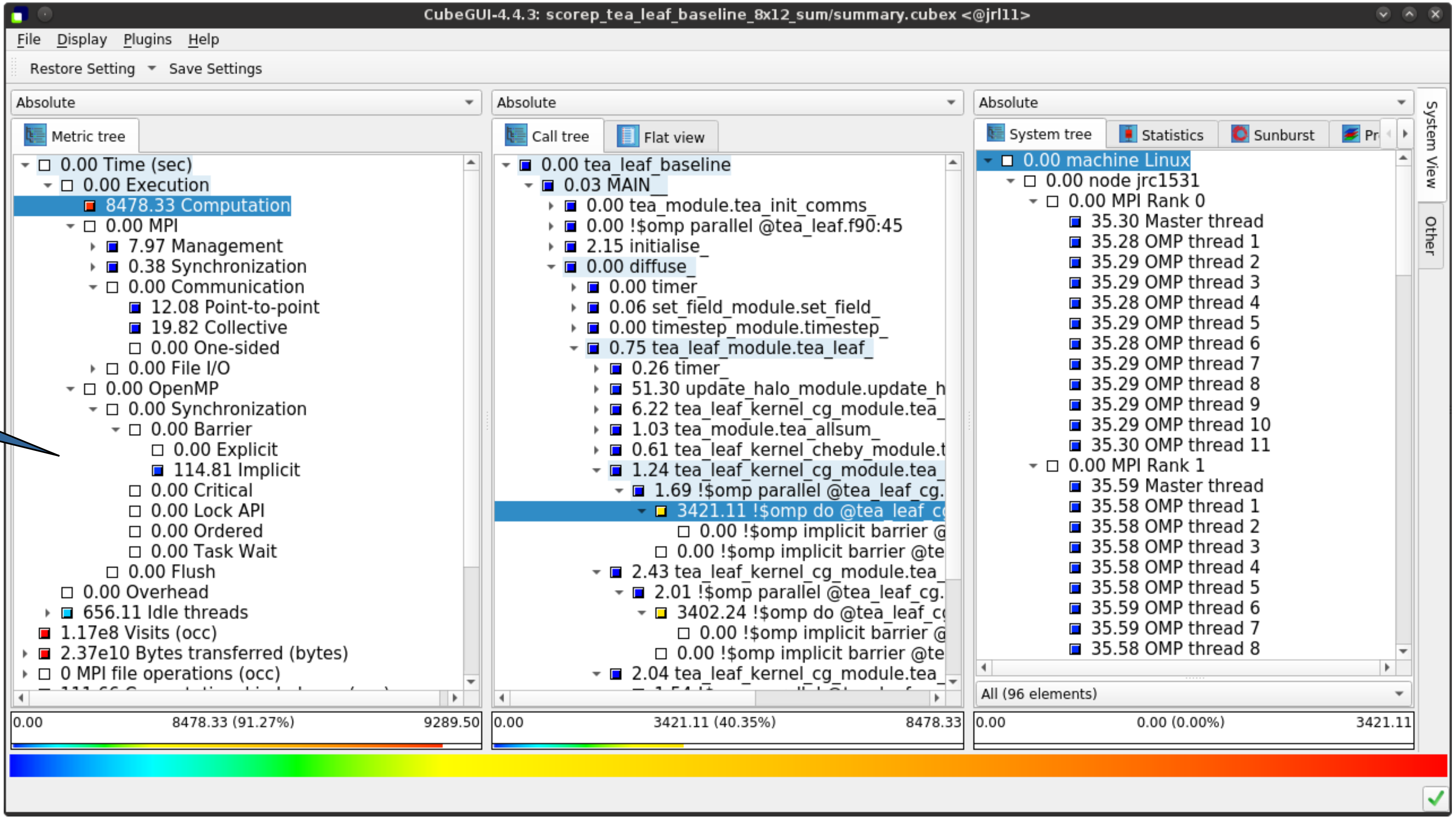


Plain summary analysis report (expanded call tree/system tree)



Post-processed summary analysis report (Scalasca)

Split base metrics from plain report into hierarchy of more specific metrics

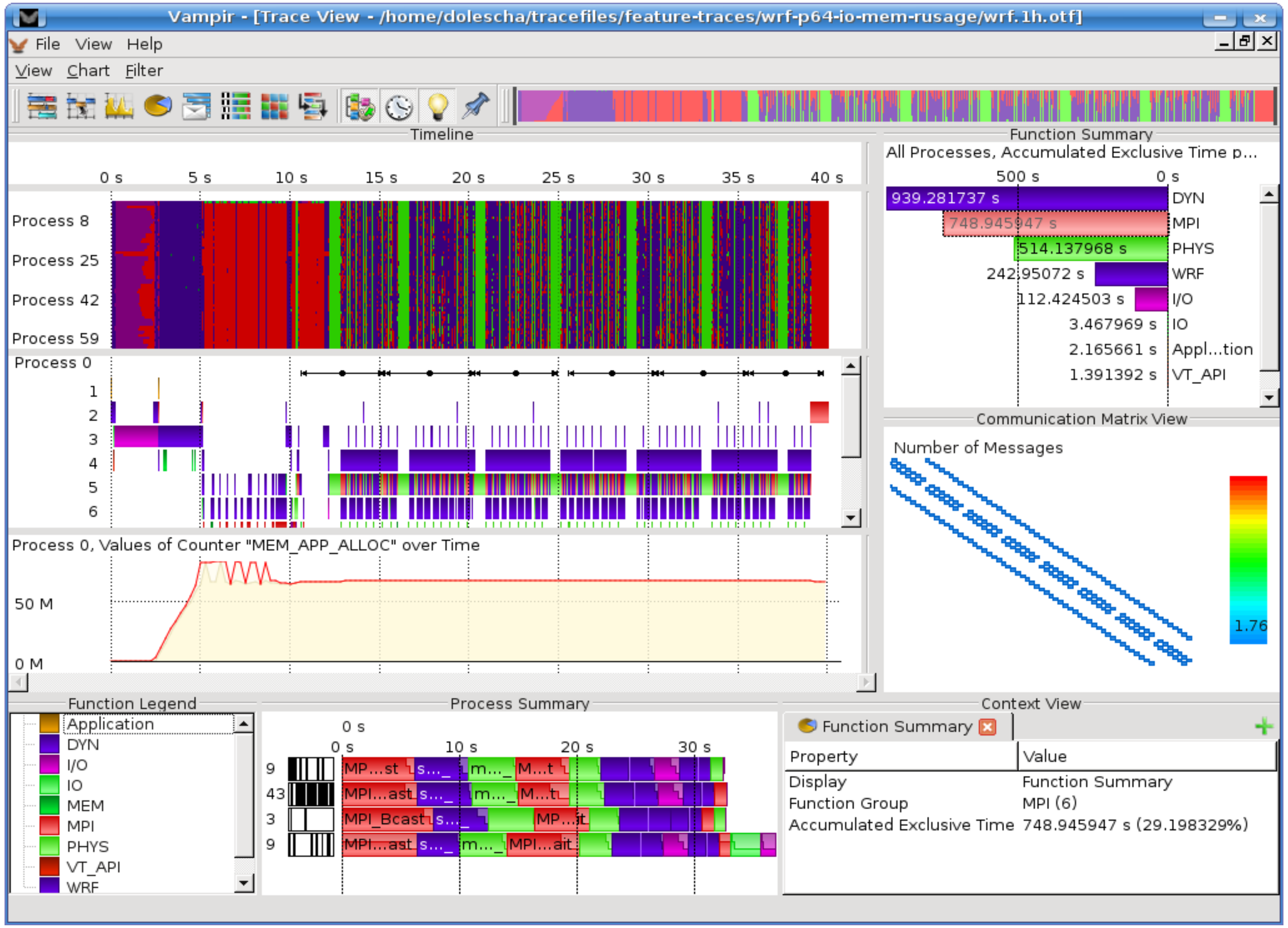


Vampir Event Trace Visualizer

- Offline trace visualization for Score-P's 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

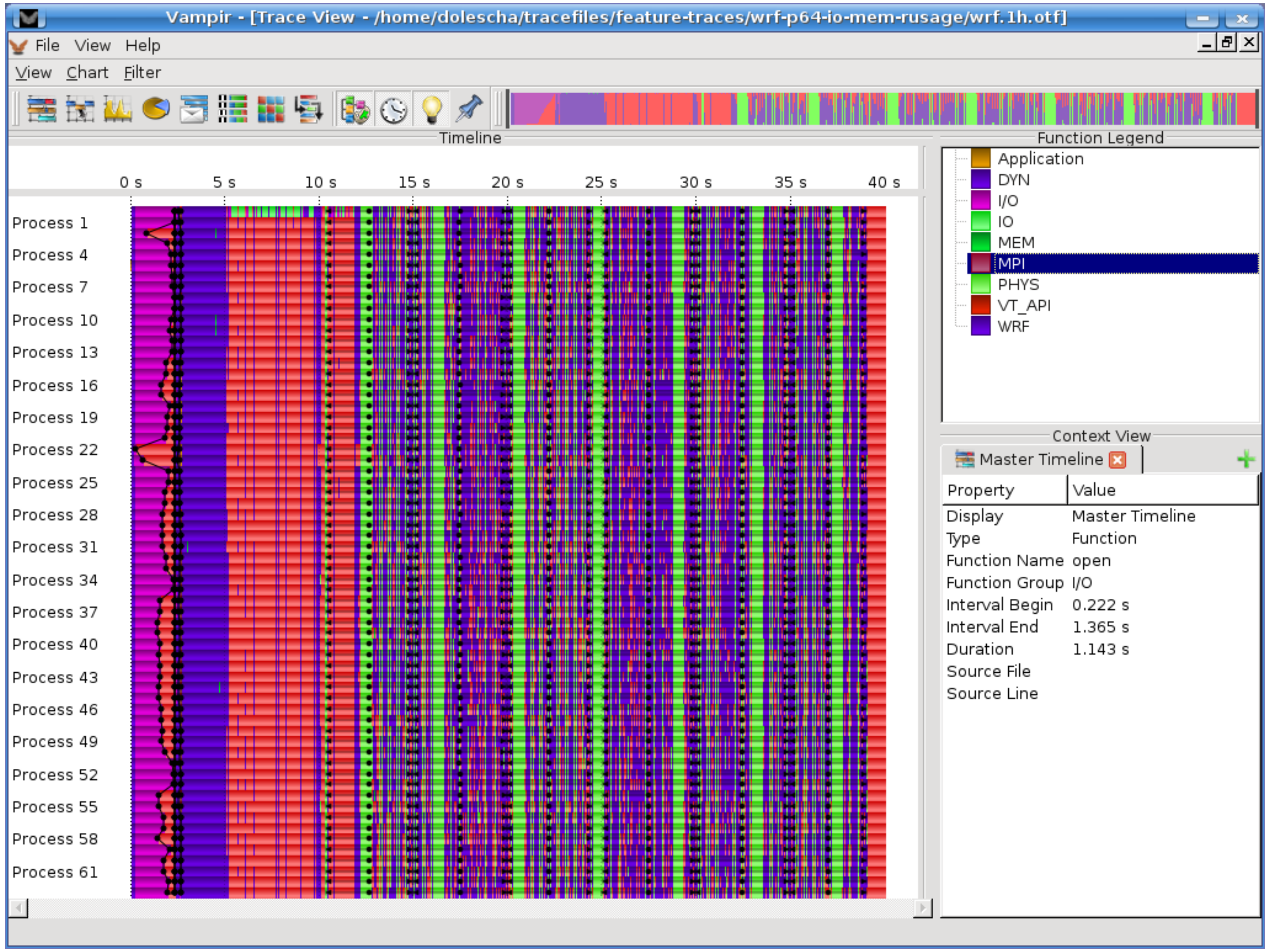
- Advantage:
  - Detailed view of dynamic application behavior
- Disadvantage:
  - Requires event traces (huge amount of data)
  - Completely manual analysis

Vampir Displays



Vampir: Timeline Diagram

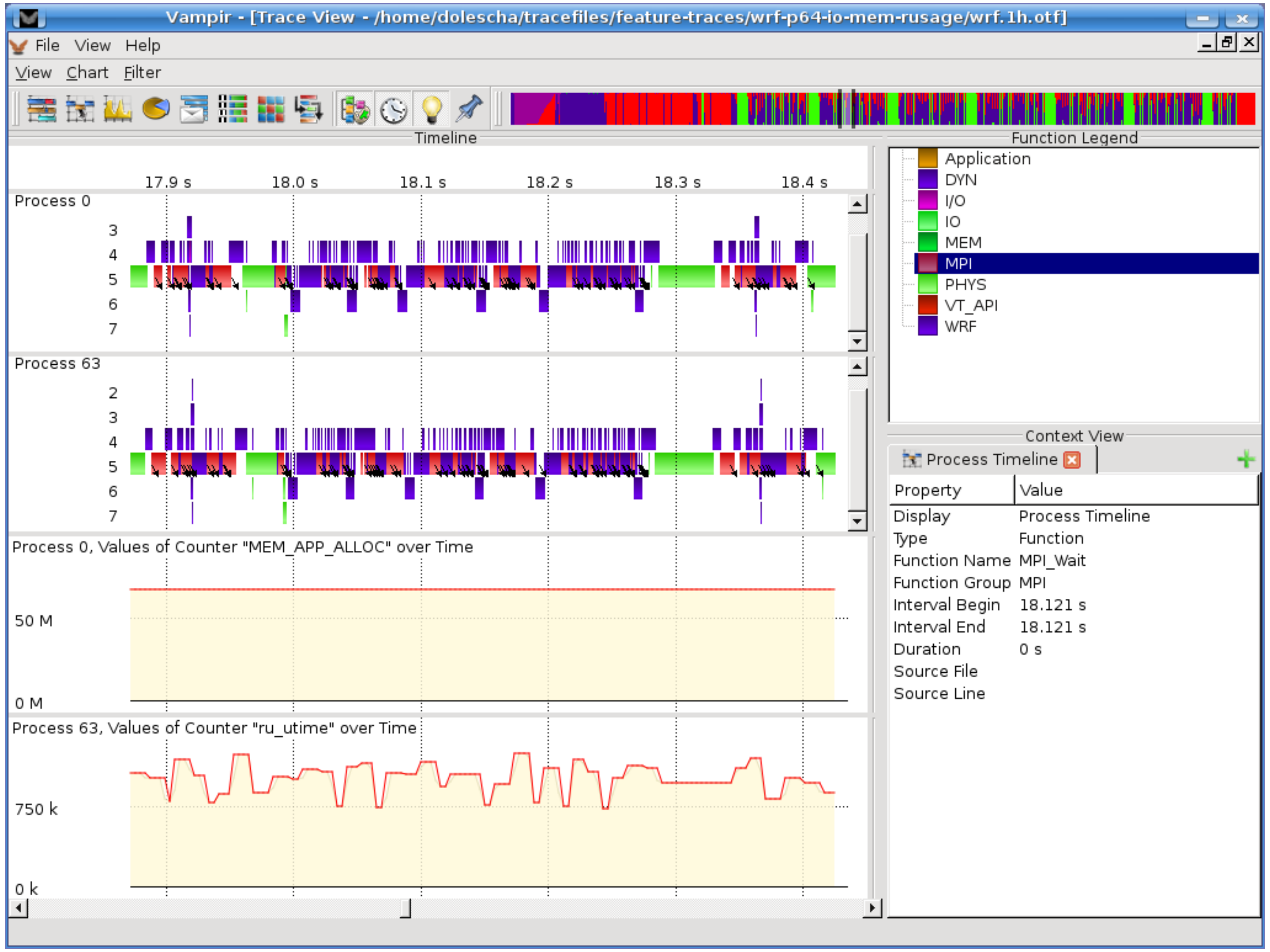
- Functions organized into groups
- Coloring by group
- Message lines can be colored by tag or size
- Information about states, messages, collective and I/O operations available through clicking on the representation



Vampir: Process and Counter Timelines

- Process timeline
  show call stack nesting

- Counter timelines
  for hardware and software counters

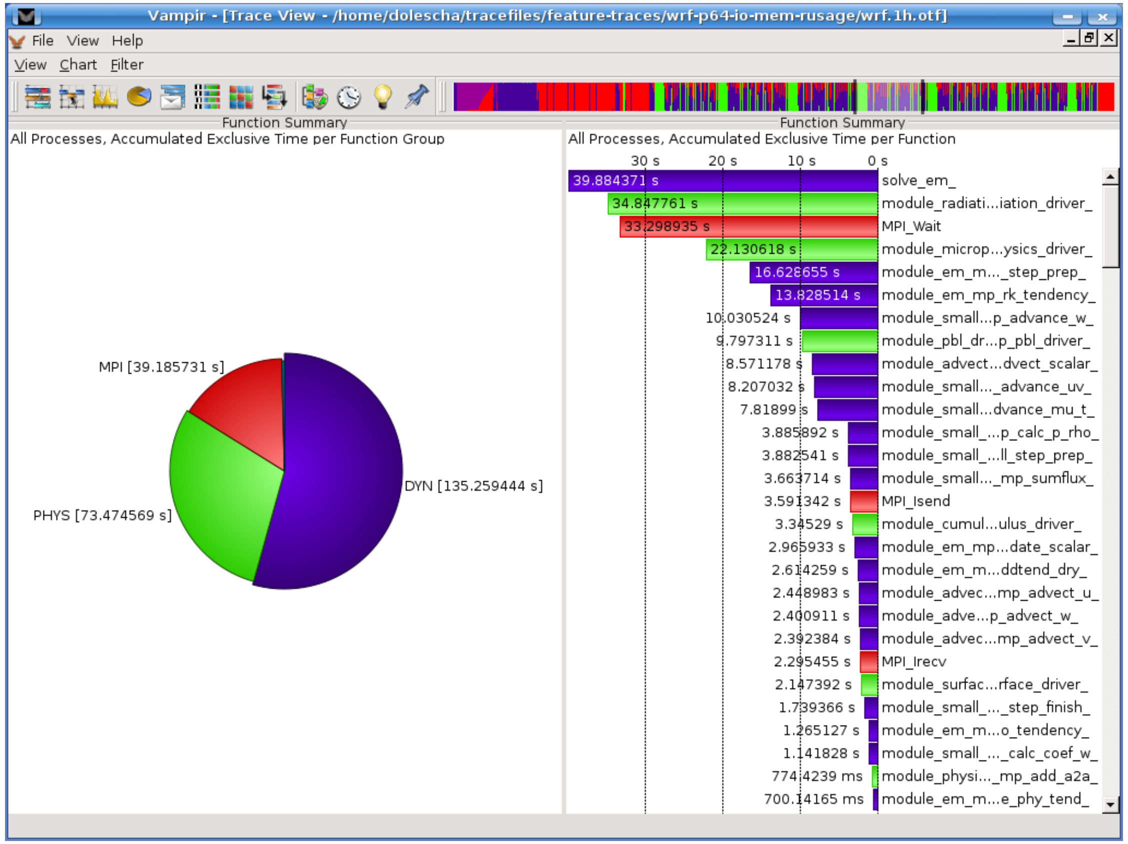


Vampir: Execution Statistics

- Aggregated profiling information: execution time, number of calls, inclusive/exclusive

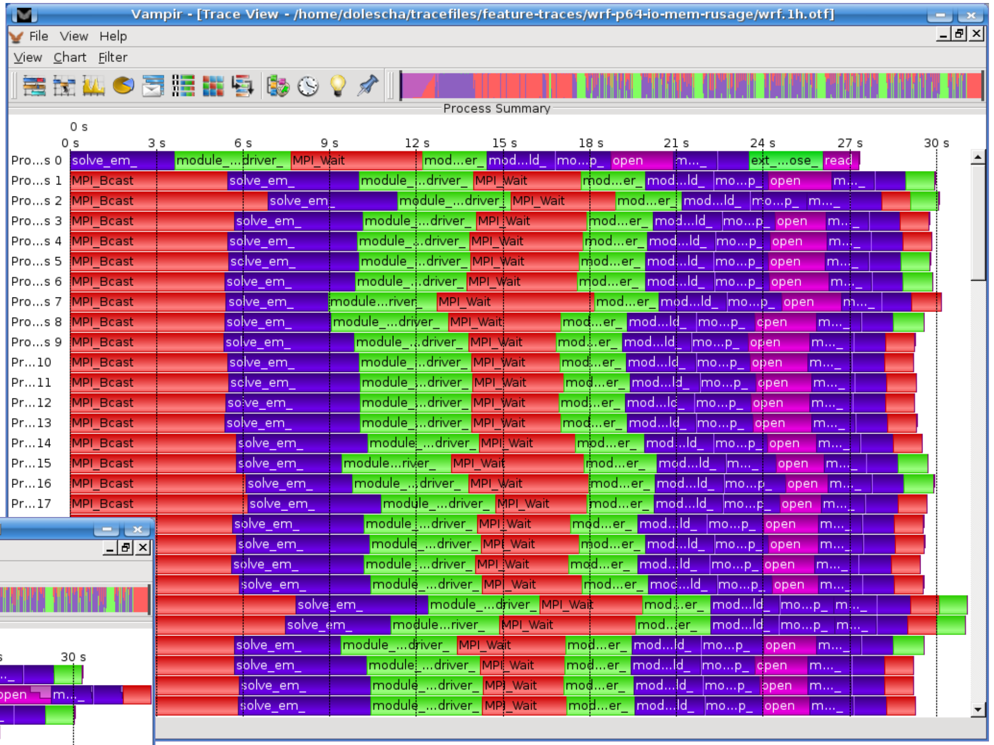
- Available for all / any group (activity) or all routines (symbols)

- Available for any part of the trace
  ⇨ selectable through time line diagram



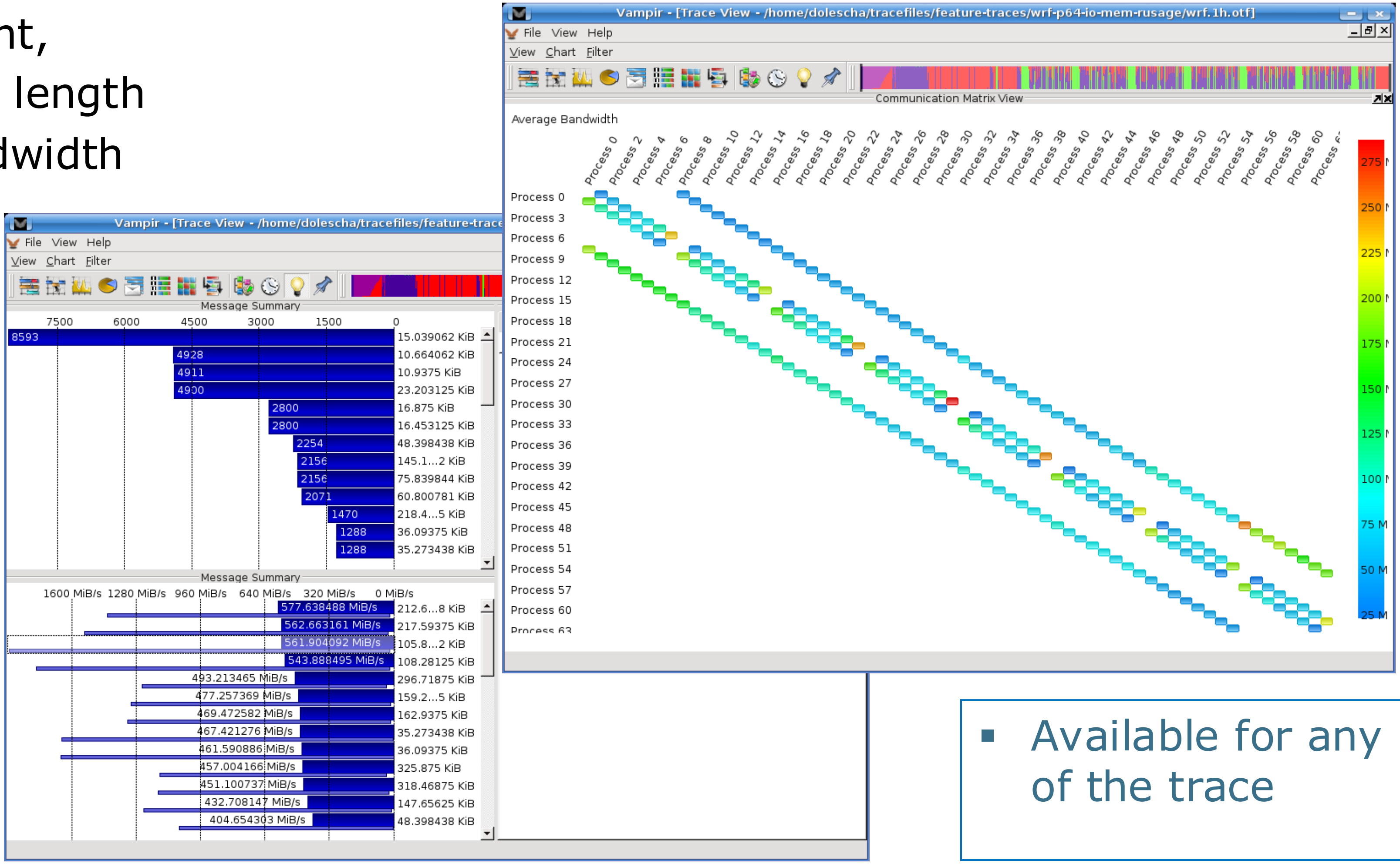
Vampir: Process Summary

- Execution statistics over all processes for comparison
- Clustering mode available for large process counts



Vampir: Communication Statistics

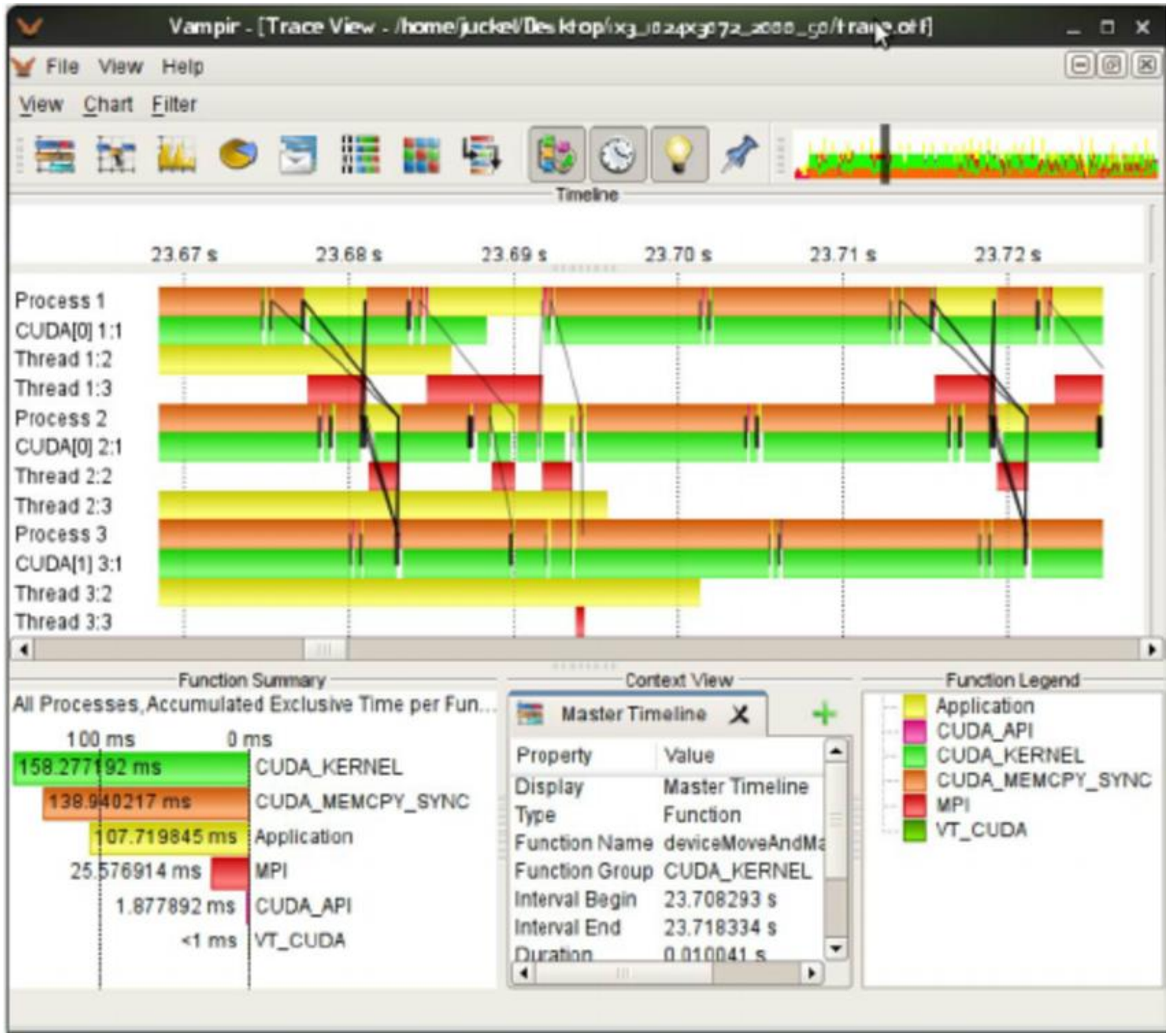
- Byte and message count, min/max/avg message length and min/max/avg bandwidth for each process pair
- Message length statistics



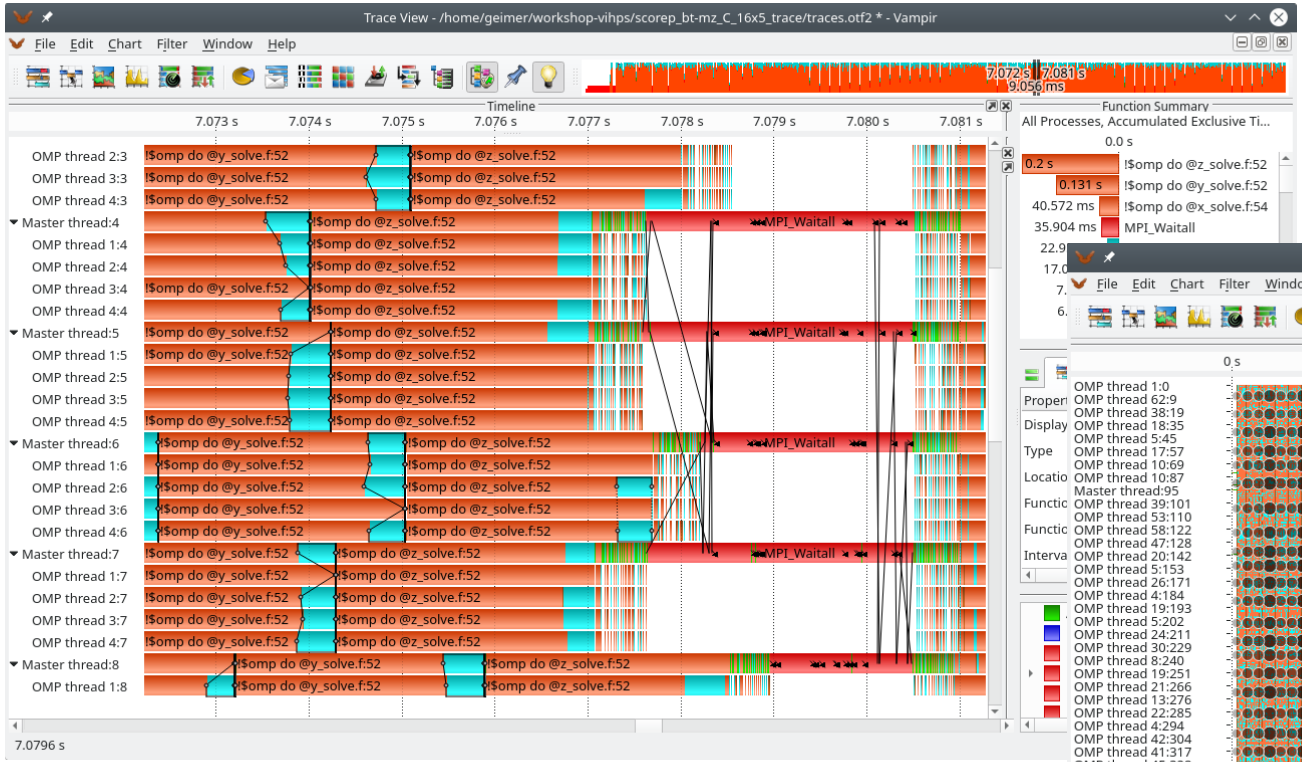
- Available for any part of the trace

Vampir: CUDA Example

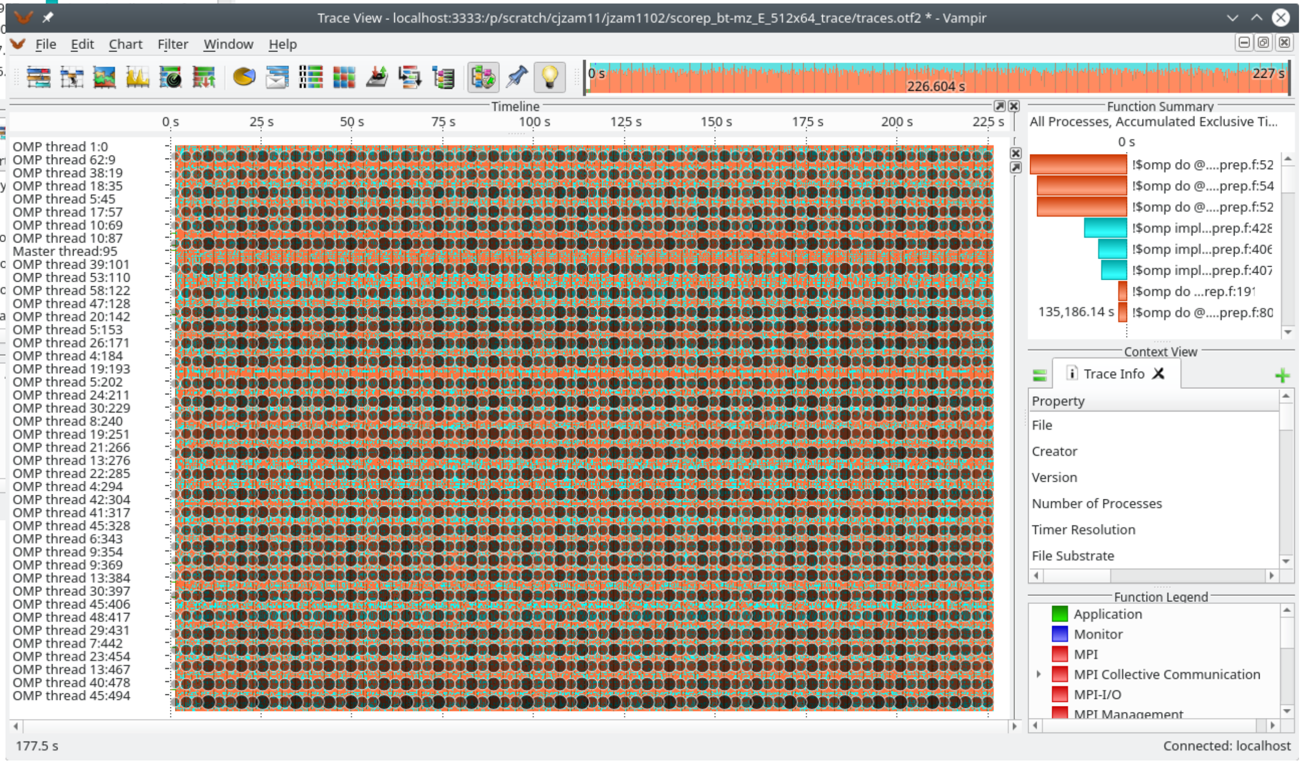
- Detailed information on kernel execution and memory transfers
- All statistics and displays also available for CUDA events



Motivation



While interactive trace visualization is often intuitive…



…finding bottlenecks may become the search for needles in a haystack…

trace tools scalasca

DOI 10.5281/zenodo.4103922

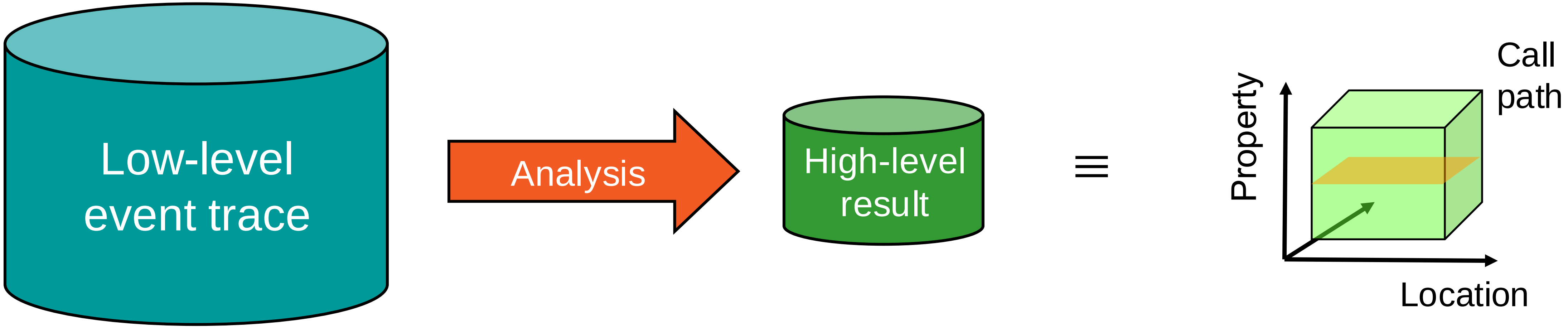
- **Scalable trace-based** performance analysis toolset for the most popular parallel programming paradigms
  - Current focus: MPI, OpenMP, and (to a limited extend) POSIX threads
  - Analysis of traces including only host-side events from applications using CUDA, OpenCL, or OpenACC (also in combination 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 v2.6.1 (Dec 2022)

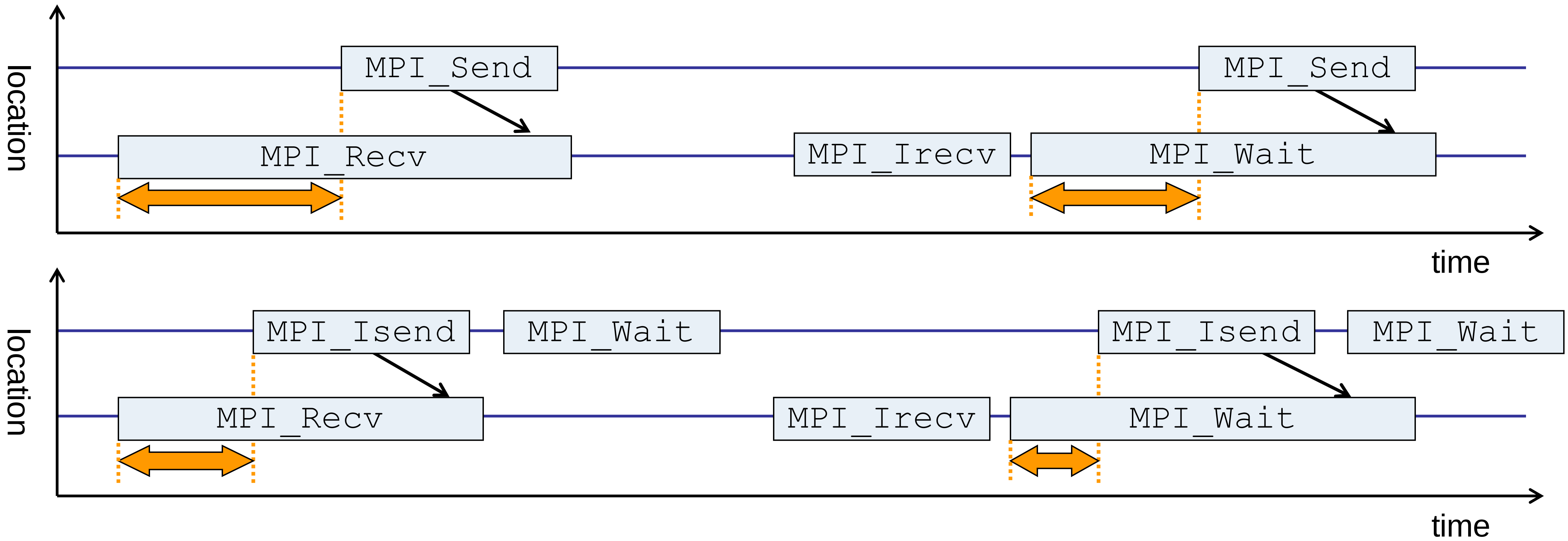
Automatic trace analysis

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



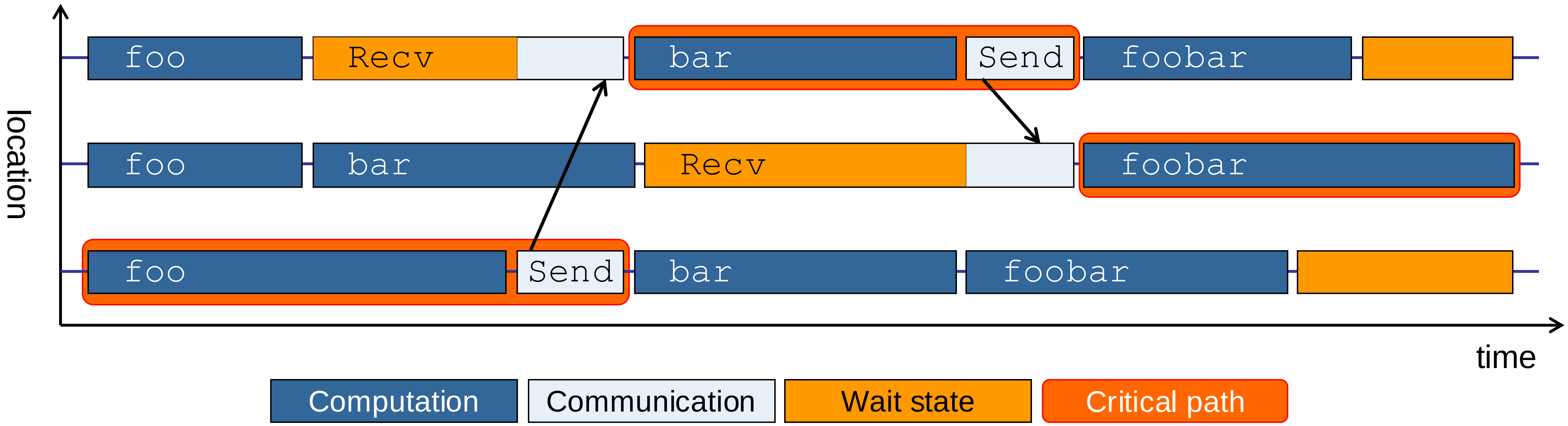
  - Guaranteed to cover the entire event trace
  - Quicker than manual/visual trace analysis
  - Parallel replay analysis exploits available memory & processors to deliver scalability

Example: "*Late Sender*" wait state



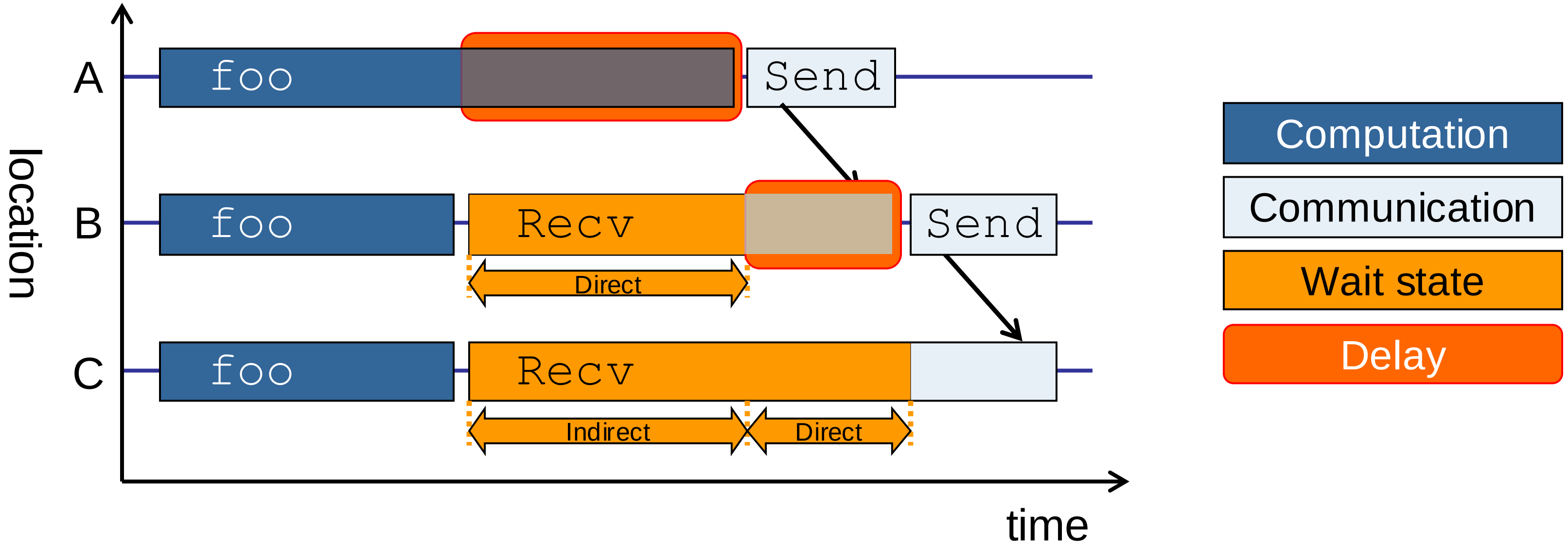
- Waiting time caused by a blocking receive operation posted earlier than the corresponding send
- Applies to blocking as well as non-blocking communication

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*

Scalasca Trace Tools features

- Open source: 3-clause BSD license
- Portability: supports all major HPC platforms
- Scalability: successful analyses with >1M threads

- Uses Score-P instrumenter & measurement libraries
  - Scalasca v2 core package focuses on trace-based analyses
  - Provides convenience commands for measurement, analysis, and postprocessing
  - Supports common data formats
    - Reads event traces in OTF2 format
    - Writes analysis reports in CUBE4 format

- Current limitations:
  - Unable to handle traces …
    - with MPI thread level exceeding MPI_THREAD_FUNNELED
    - containing memory events, CUDA/HIP/OpenCL device events (kernel, memcpy), SHMEM, or OpenMP nested parallelism
  - PAPI/rusage metrics for trace events are ignored

Putting it all together