PERFORMANCE ANALYSIS IN A NUTSHELL

AN INTRO WITH SCORE-P, SCALASCA, AND VAMPIR

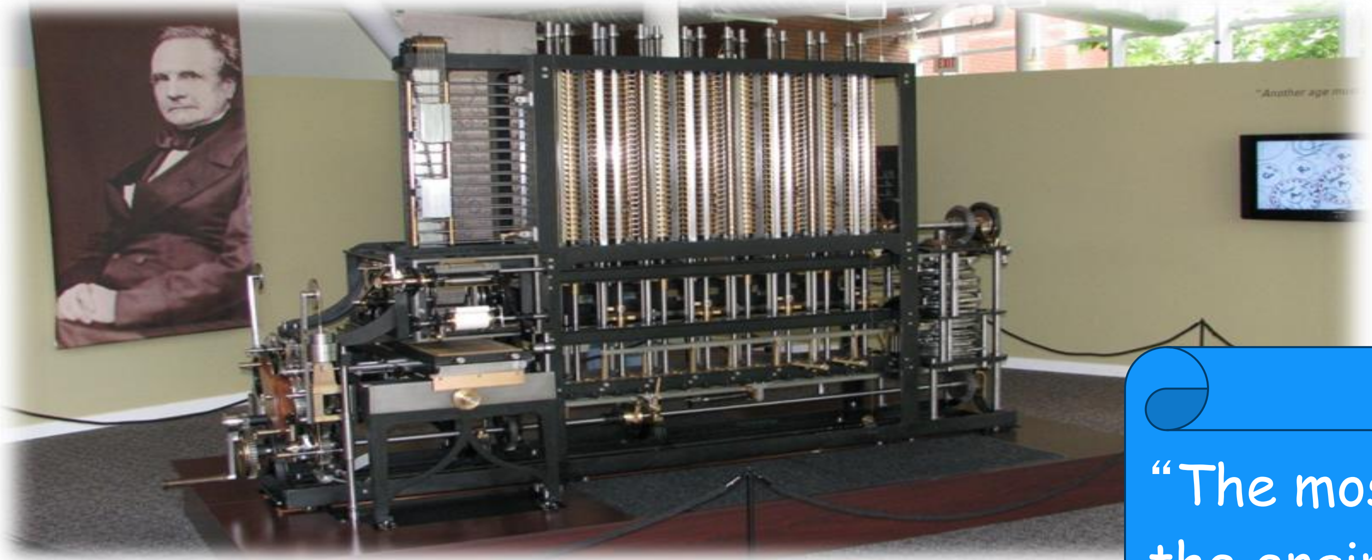
OCTOBER 13, 2022 I MICHAEL KNOBLOCH

DISCLAIMER

Tools will ***not*** automatically make you, your applications or computer systems more productive.

However, they can help you understand ***how*** your parallel code executes and ***when / where*** it's necessary to work on correctness and performance issues.

PERFORMANCE: AN OLD PROBLEM



Difference Engine

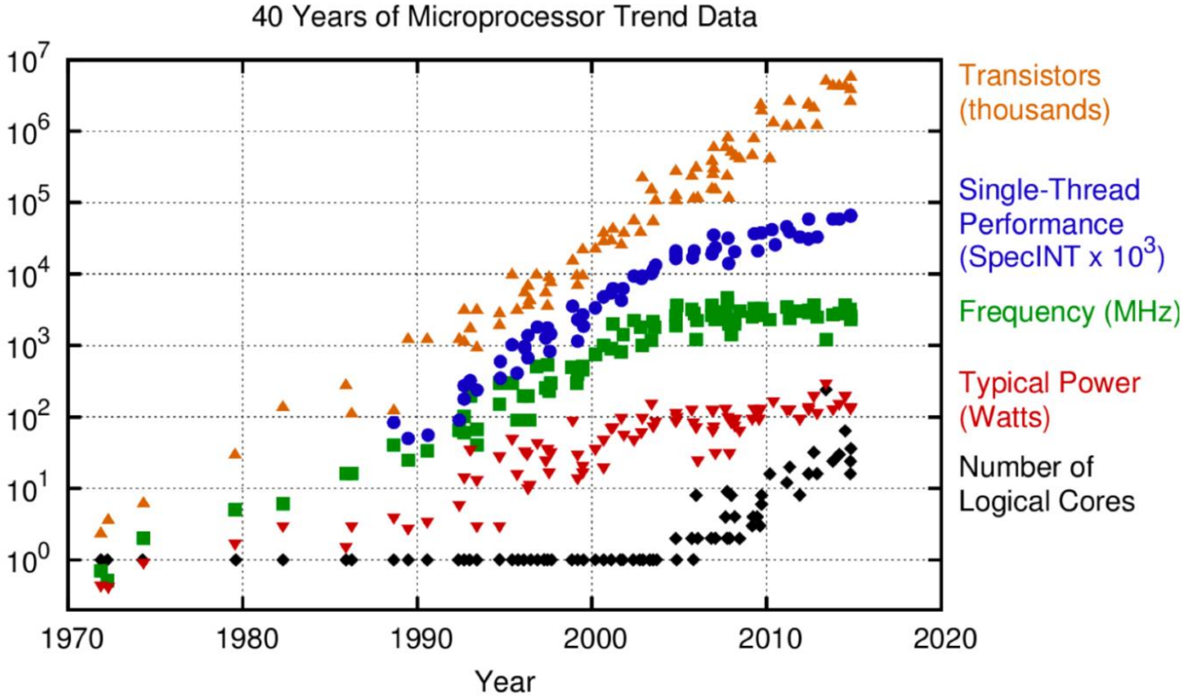
"The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible."

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
    - Accelerators
    - Modular supercomputer architecture

☞ Every doubling of scale reveals a new bottleneck!



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

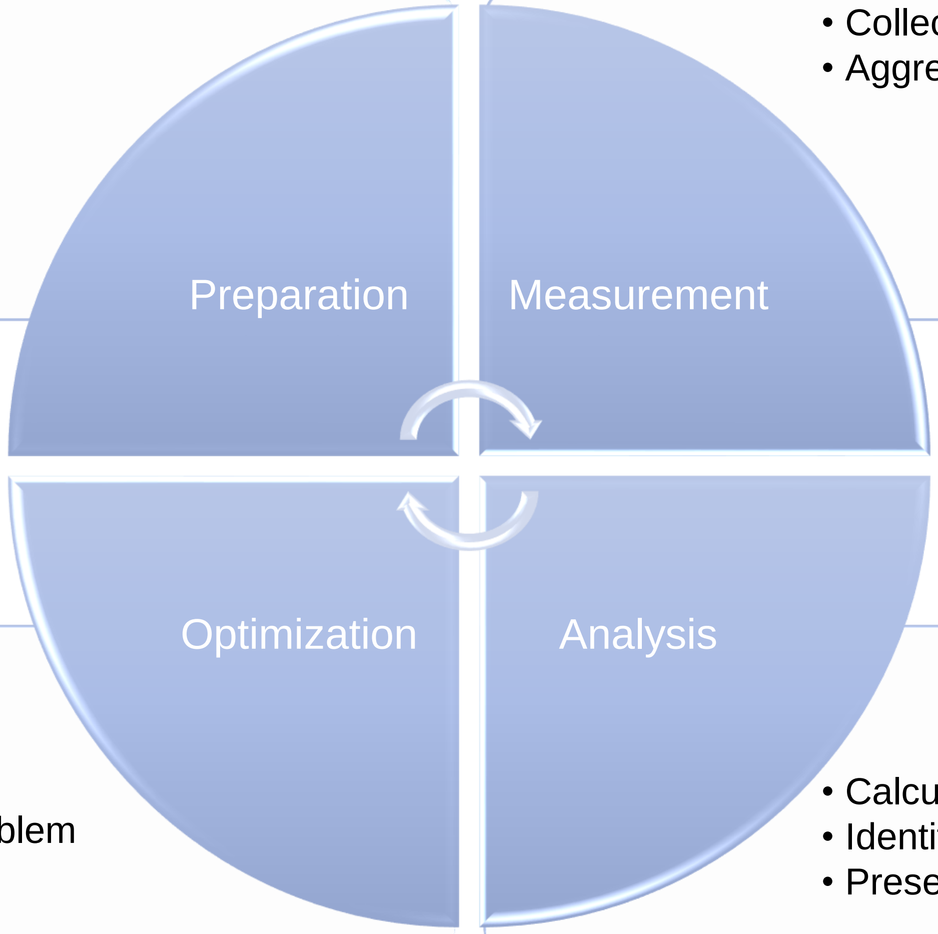
TUNING BASICS

- Successful performance engineering is a combination of
  - Careful setting of various tuning parameters
  - The right algorithms and libraries
  - Compiler flags and directives
  - Correct machine usage (mapping and bindings)
  - …
  - Thinking !!!
- Measurement is better than guessing
  - To determine performance bottlenecks
  - To compare alternatives
  - To validate tuning decisions and optimizations
    - After each step!
- Modeling is extremely useful but very difficult and rarely available
  - Allows to evaluate performance impact of optimization without implementing it
  - Simplifies search in large parameter space

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

PERFORMANCE METRICS

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

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

EXAMPLE METRICS

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

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

EXECUTION TIME

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

- Problem: Execution time is non-deterministic
  - Use mean or minimum of several runs

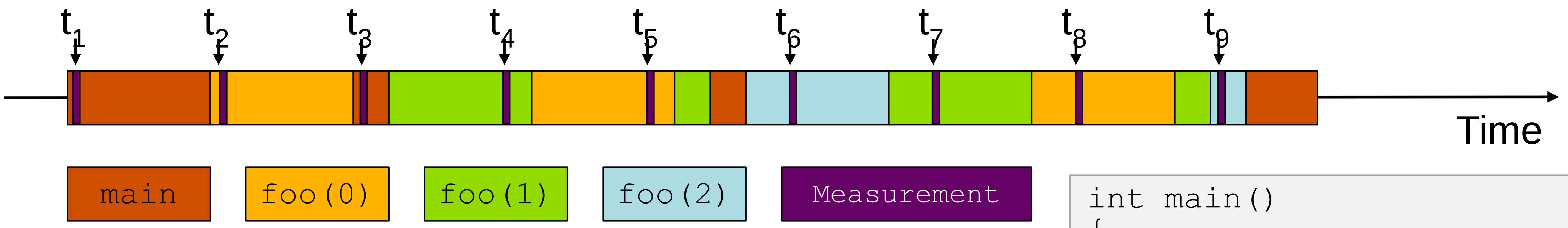
CLASSIFICATION OF MEASUREMENT TECHNIQUES

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

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

- How is performance data analyzed?
  - Online – No suitable tools anymore
  - Post mortem

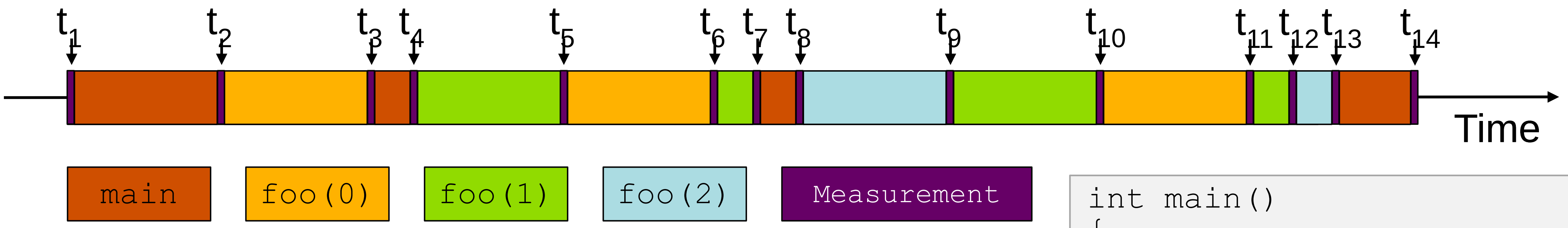
SAMPLING



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

```
int main()
{
  int i;

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

  return 0;
}

void foo(int i)
{

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

}
```

INSTRUMENTATION



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

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

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

INSTRUMENTATION TECHNIQUES

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

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

CRITICAL ISSUES

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

- *Tradeoff: Accuracy vs. Expressiveness of data*

CLASSIFICATION OF MEASUREMENT TECHNIQUES

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

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

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

PROFILING / RUNTIME SUMMARIZATION

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

- *Profile = summarization of events over execution interval*

TYPES OF PROFILES

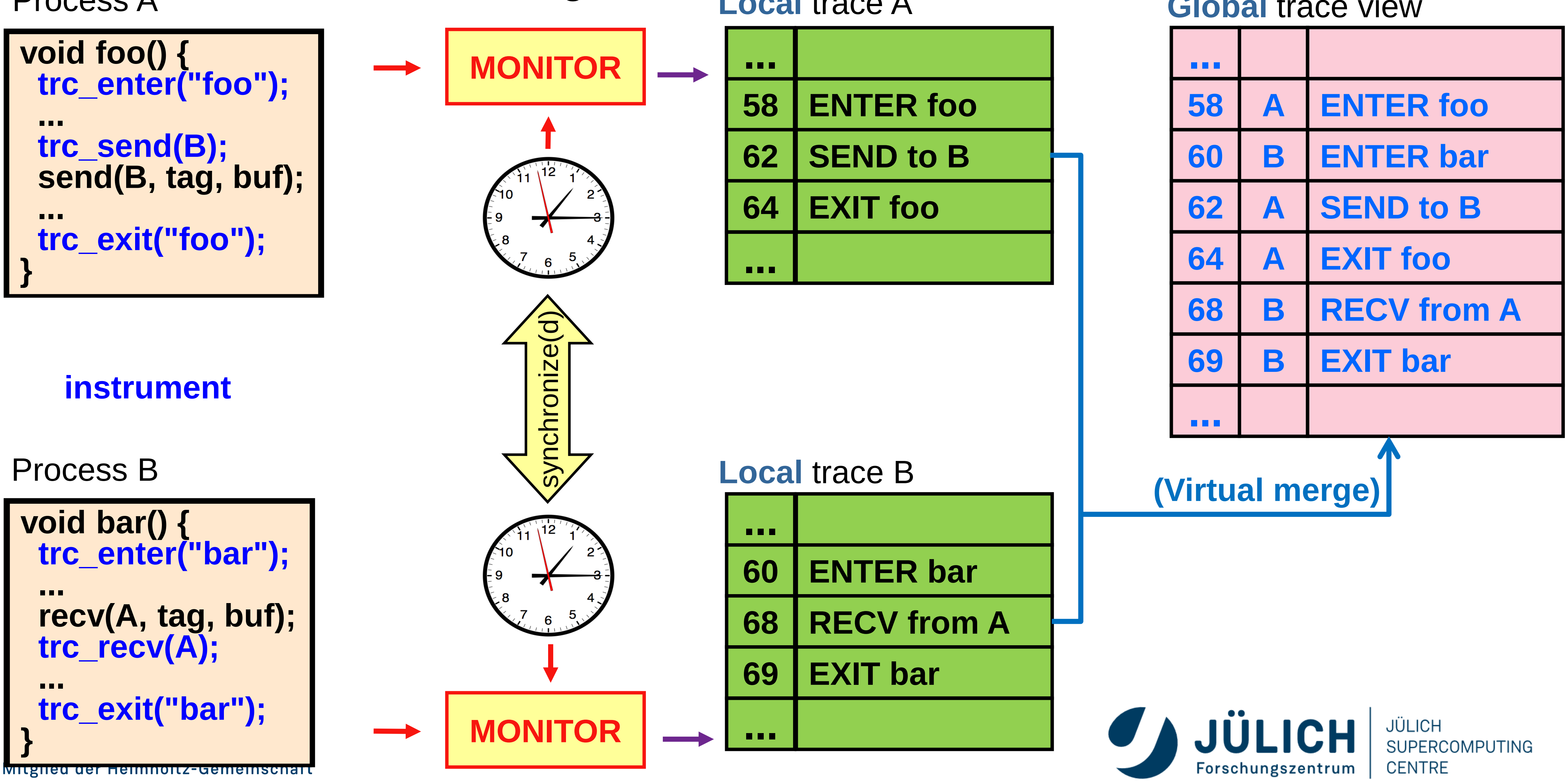
- Flat profile
  - Shows distribution of metrics per routine / instrumented region
  - Calling context is not taken into account
- Call-path profile
  - Shows distribution of metrics per executed call path
  - Sometimes only distinguished by partial calling context (e.g., two levels)
- Special-purpose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

TRACING

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

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

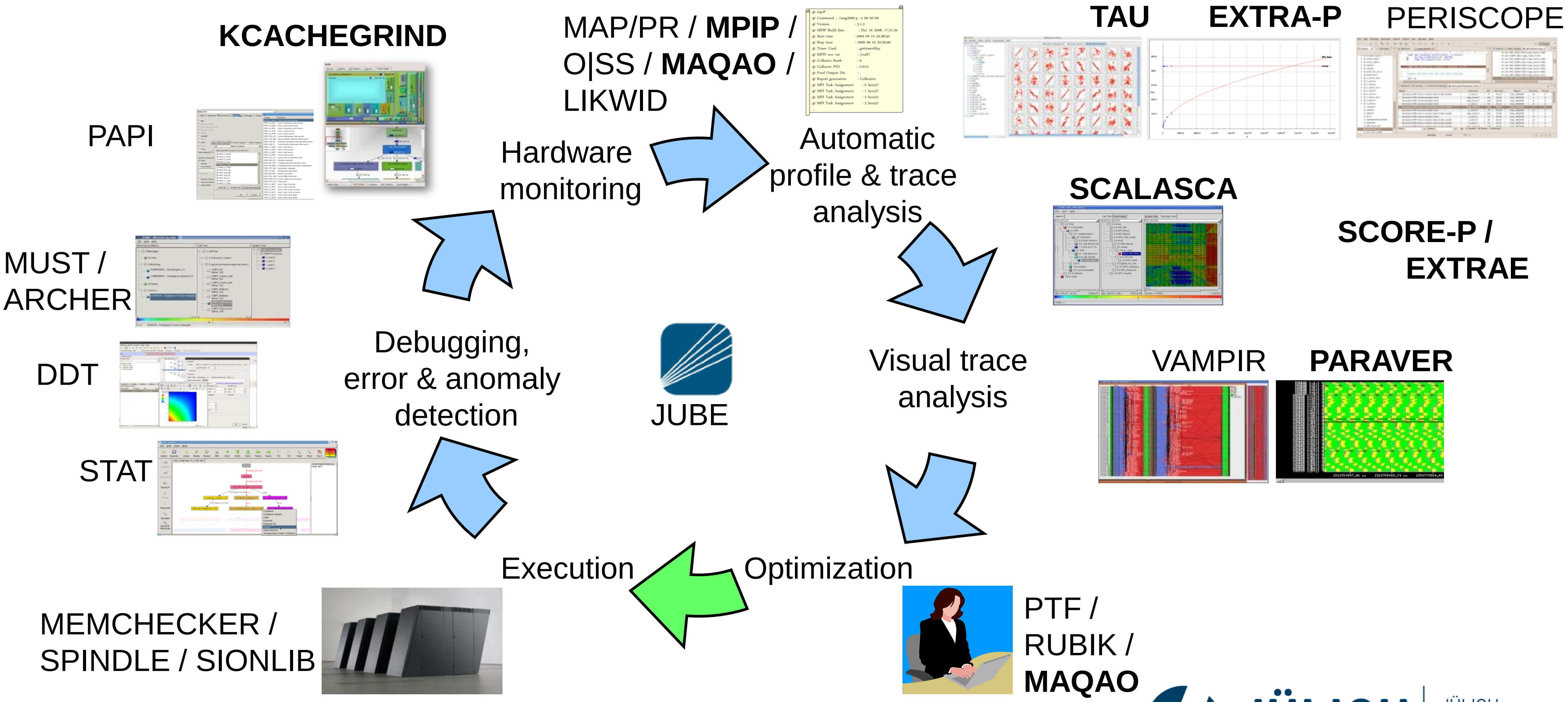
Event tracing



TRACING PROS & CONS

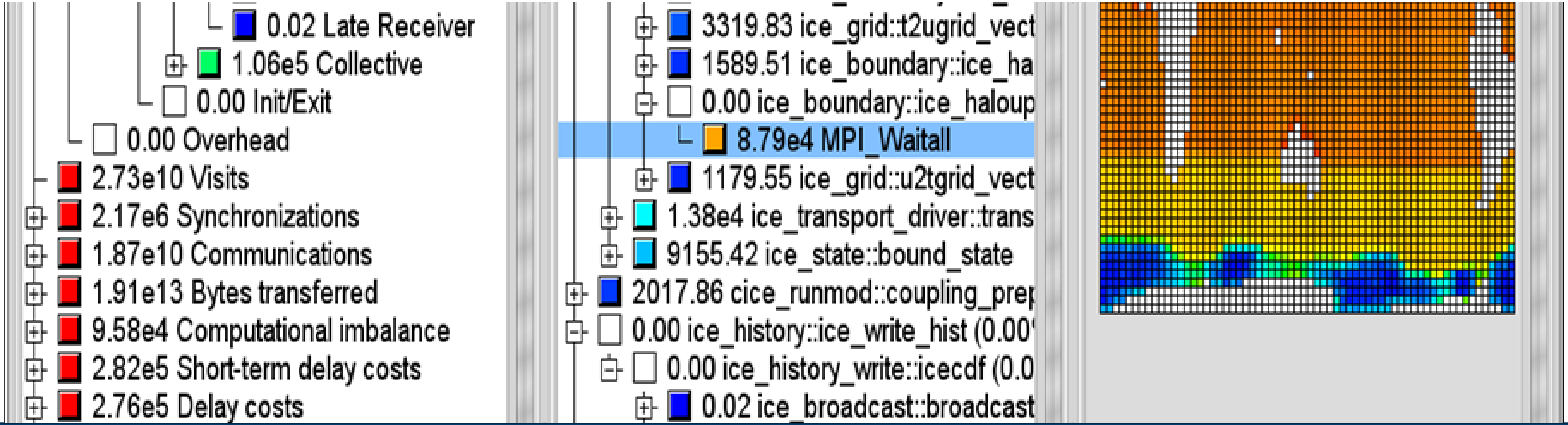
- Tracing advantages
  - Event traces preserve the **temporal** and **spatial** relationships among individual events (☞ context)
  - Allows reconstruction of **dynamic** application behavior on any required level of abstraction
  - Most general measurement technique
    - Profile data can be reconstructed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

TECHNOLOGIES AND THEIR INTEGRATION



REMARK: NO SINGLE SOLUTION IS SUFFICIENT!

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



SCORE-P AND SCALASCA

SCORE-P

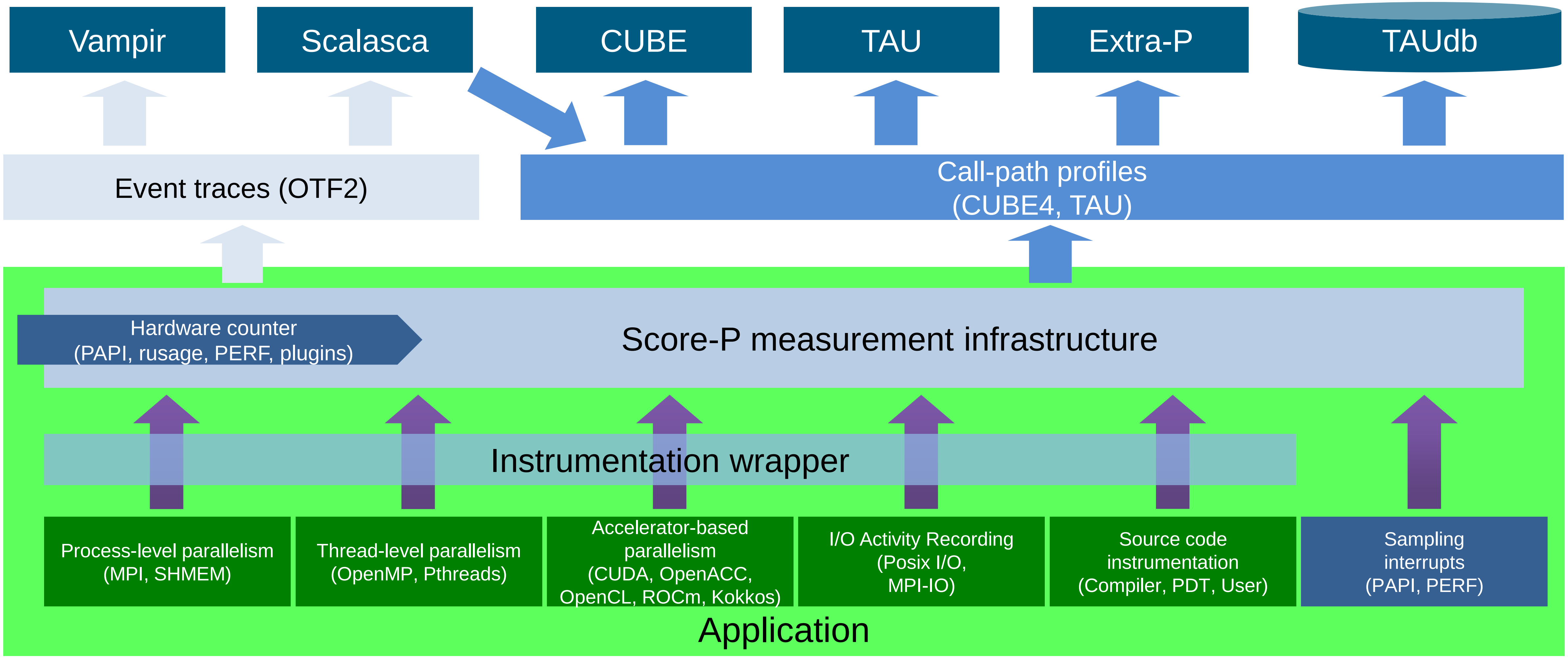
- Infrastructure for instrumentation and performance measurements
- Instrumented application can be used to produce several results:
  - Call-path profiling: CUBE4 data format used for data exchange
  - Event-based tracing: OTF2 data format used for data exchange

- Supported parallel paradigms:
  - Multi-process: MPI, SHMEM
  - Thread-parallel: OpenMP, Pthreads
  - Accelerator-based: CUDA, OpenCL, OpenACC, ROCm, Kokkos

- Open Source; portable and scalable to all major HPC systems
- Initial project funded by BMBF
- Further developed in multiple 3rd-party funded projects

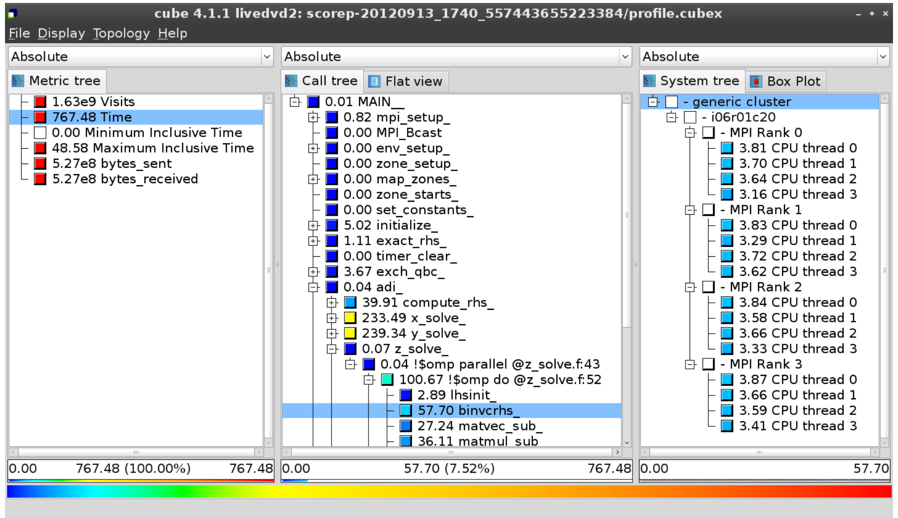
GEFÖRDERT VOM
Bundesministerium für Bildung und Forschung

SCORE-P OVERVIEW



CUBE

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



- Originally developed as part of the Scalasca toolset

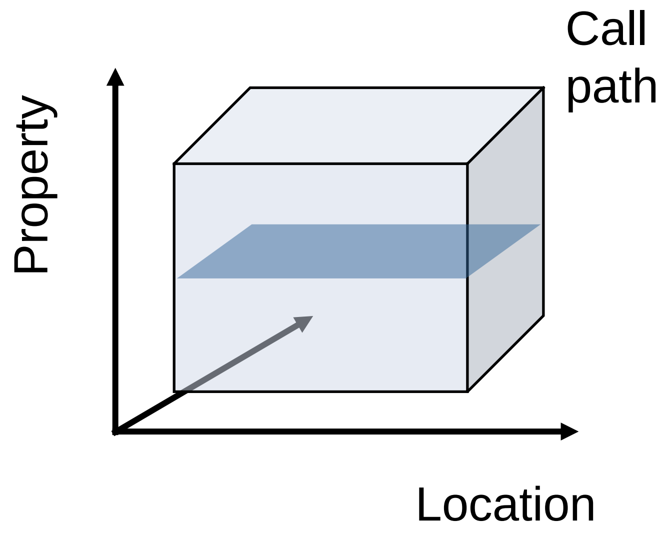
- Now available as a separate component
  - Can be installed independently of Score-P, e.g., on laptop or desktop
  - Latest release: Cube v4.6 (April 2021)

ANALYSIS PRESENTATION AND EXPLORATION - CUBE

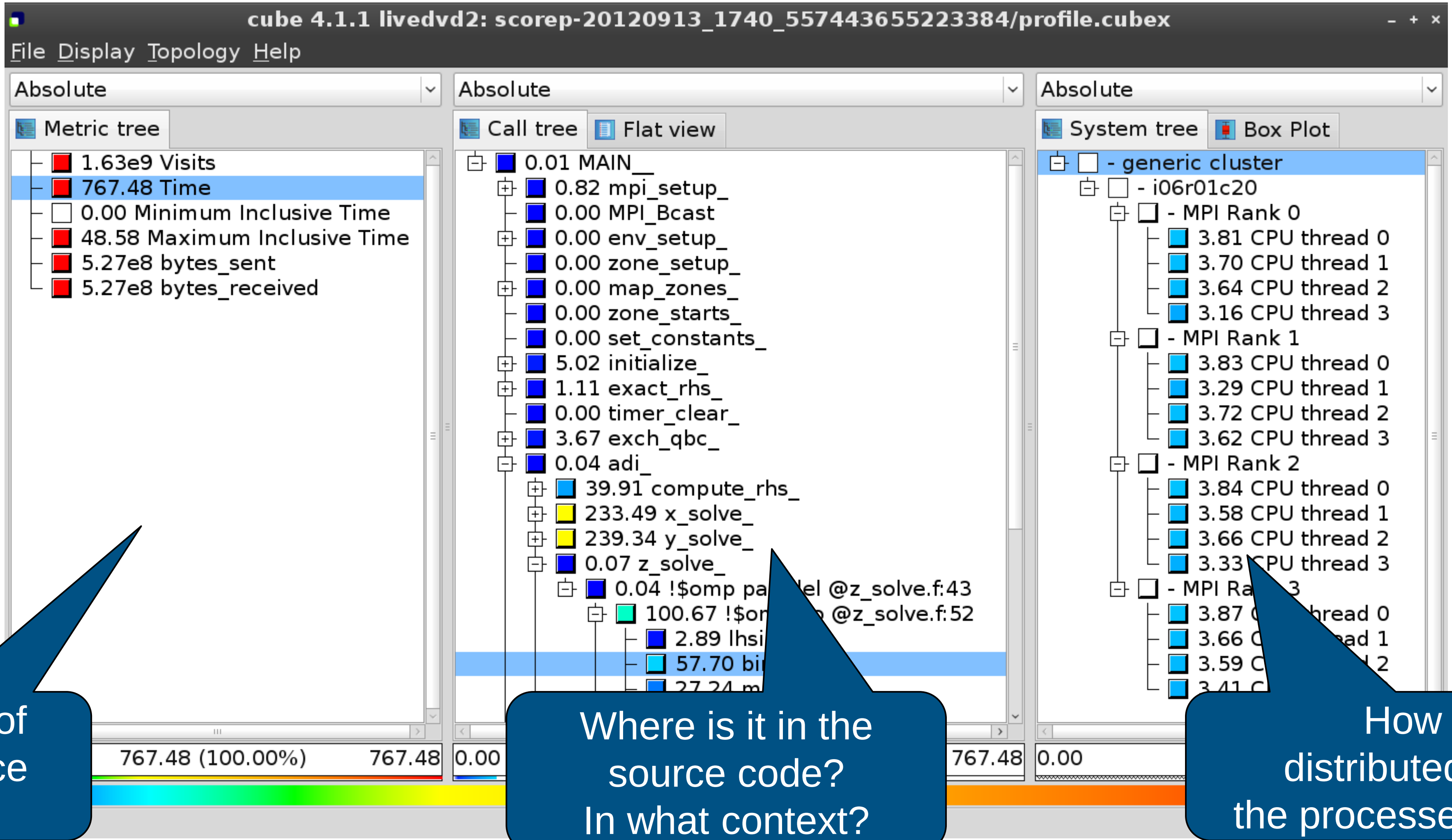
- Representation of values (severity matrix) on three hierarchical axes
  - Performance property (metric)
  - Call path (program location)
  - System location (process/thread)

- Three coupled tree browsers

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



ANALYSIS PRESENTATION



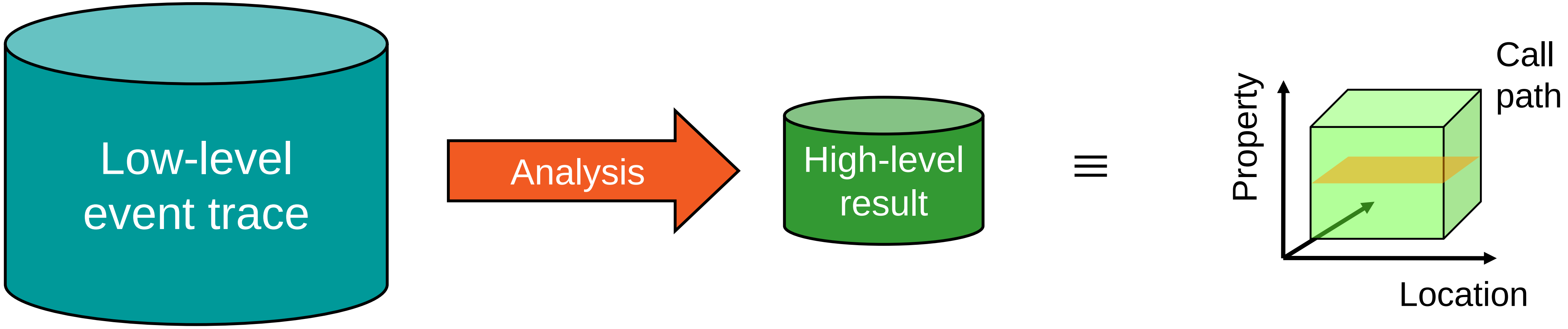
What kind of performance metric?

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

How is it distributed across the processes/threads?

AUTOMATIC TRACE ANALYSIS

- Idea
  - 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

SCALASCA TRACE TOOLS: OBJECTIVE

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

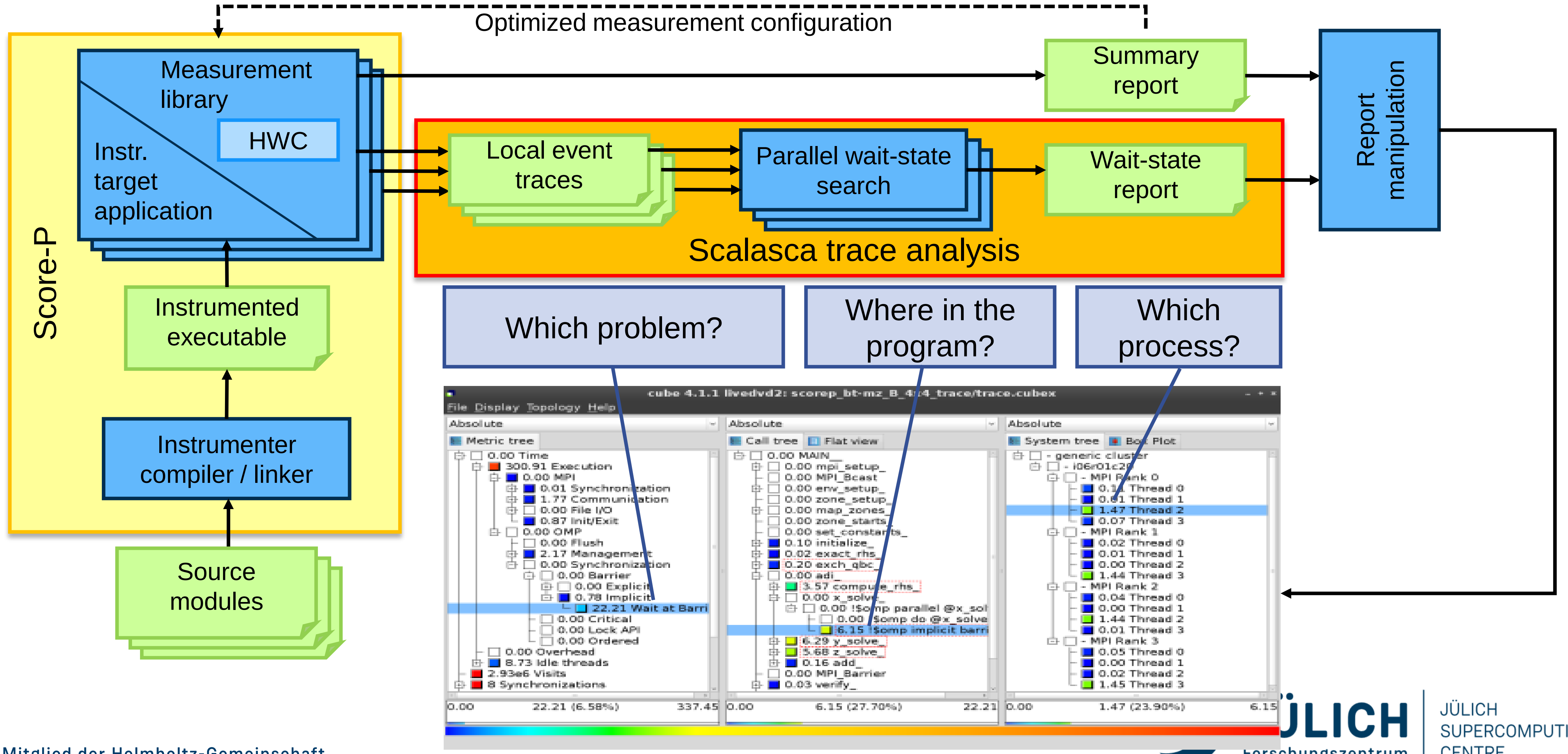
- Specifically targeting large-scale parallel applications
  - Demonstrated scalability up to 1.8 million parallel threads
  - Of course also works at small/medium scale

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

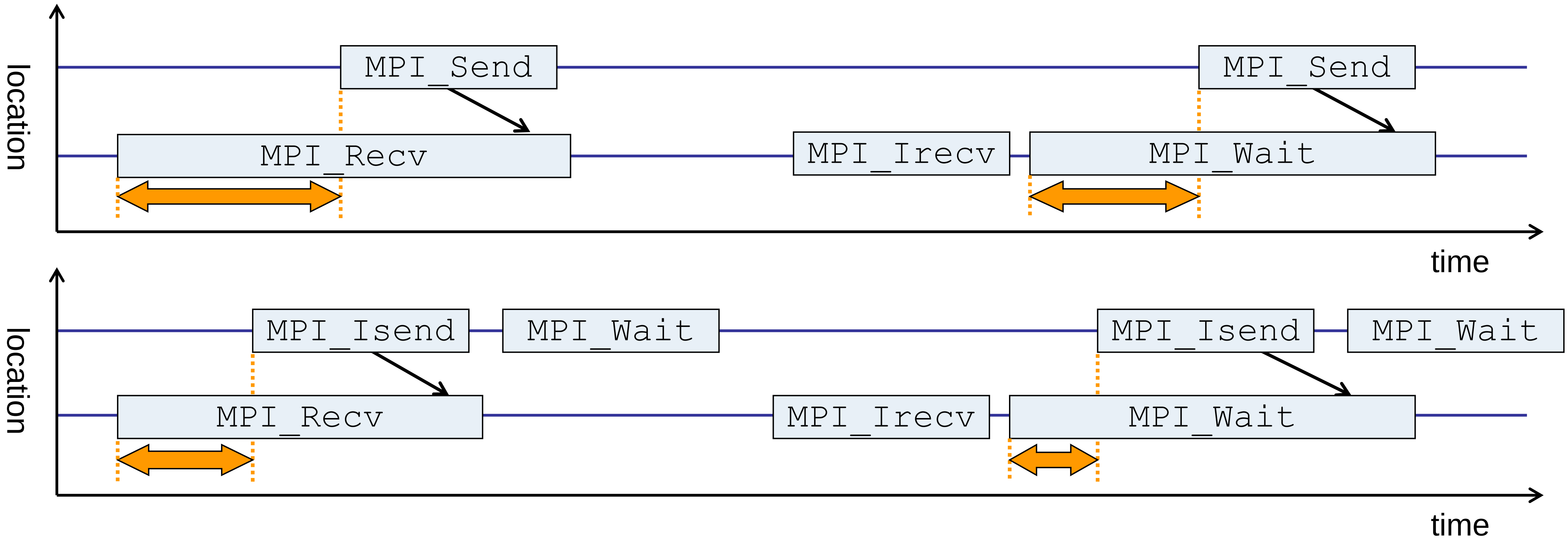
SCALASCA TRACE TOOLS: FEATURES

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

SCALASCA WORKFLOW

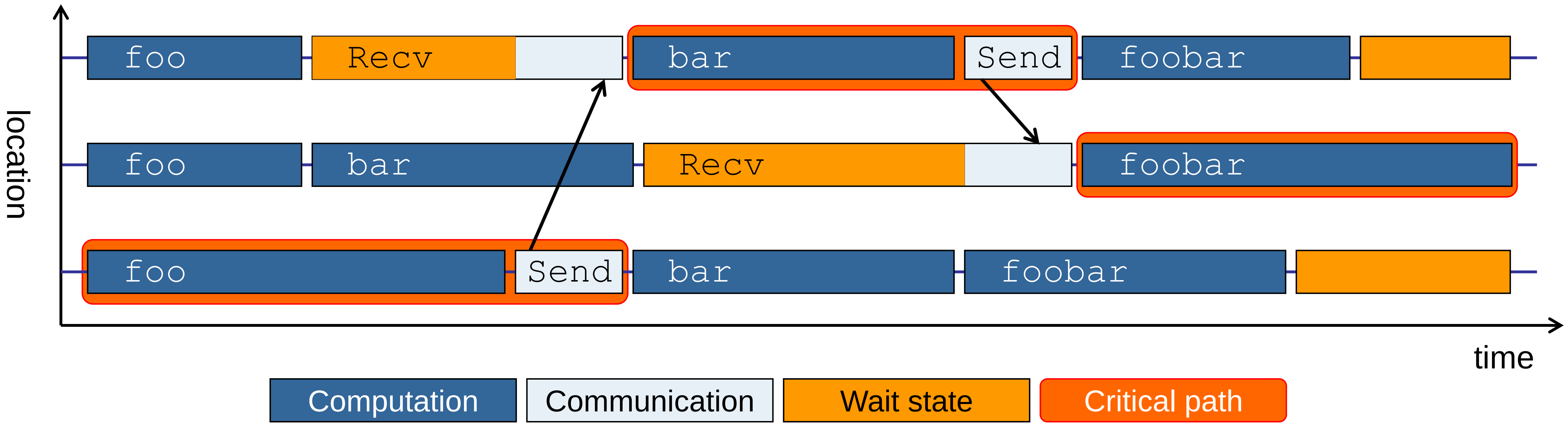


EXAMPLE: "*LATE SENDER*" WAIT STATE



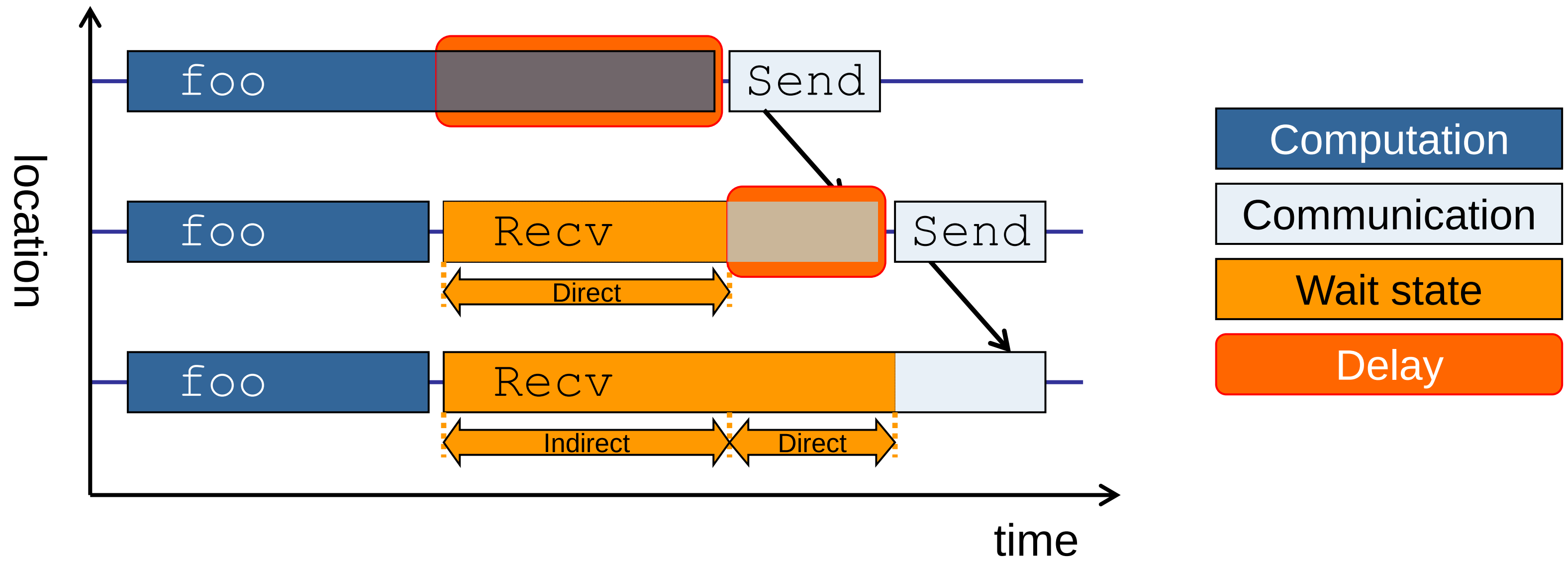
- Waiting time caused by a blocking receive operation posted earlier than the corresponding send
- Applies to blocking as well as non-blocking communication

EXAMPLE: CRITICAL PATH



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

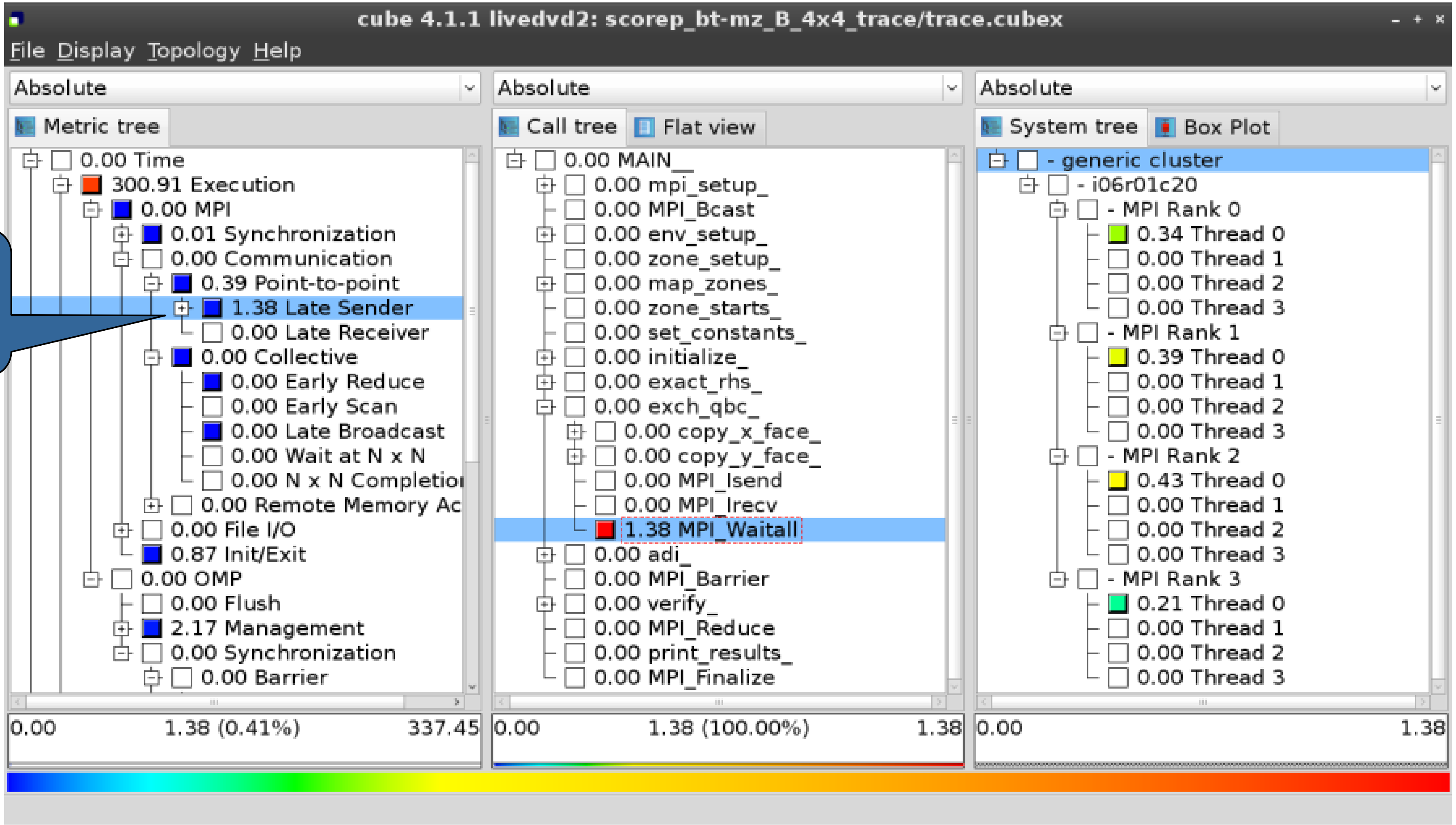
EXAMPLE: ROOT-CAUSE ANALYSIS



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

TRACE ANALYSIS REPORT

Additional trace-based metrics in metric hierarchy

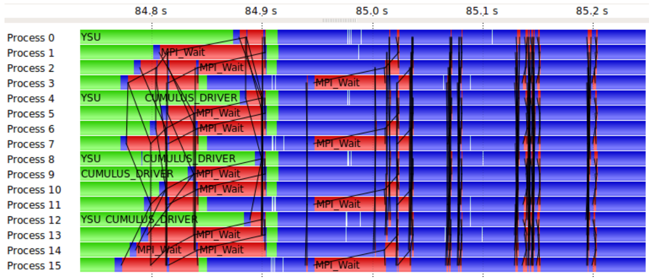


EVENT TRACE VISUALIZATION WITH VAMPIR

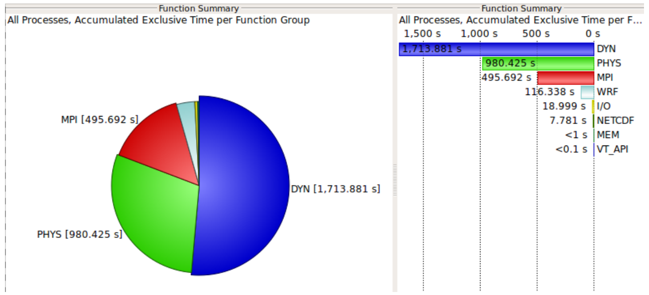
- Visualization of dynamic runtime behaviour at any level of detail along with statistics and performance metrics
- Alternative and supplement to automatic analysis

- **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

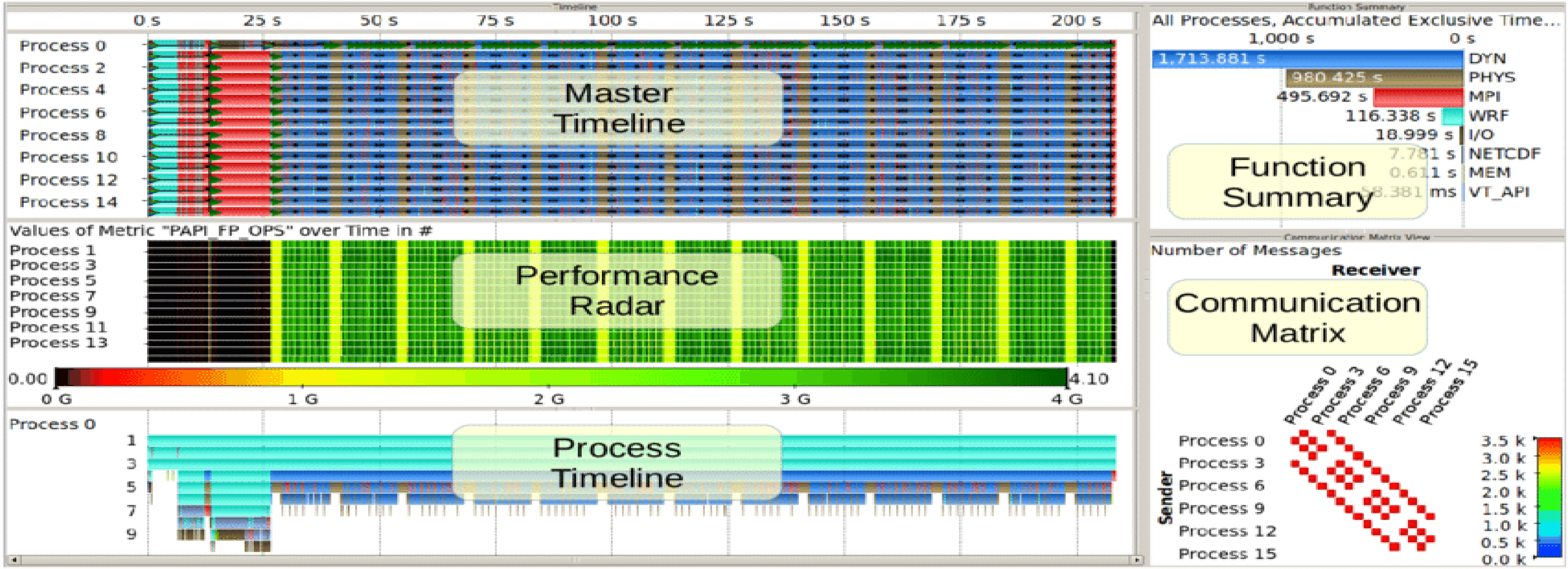


- **Summary charts**
  - Quantitative results for the currently selected time interval



JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

VAMPIR DISPLAYS



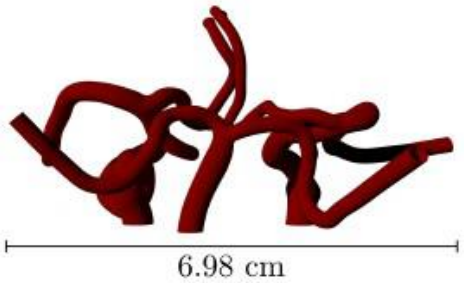
SCORE-P/CUBE CASE STUDY - HEMELB

HEMELB (SUPERMUC-NG: NO GPUS)

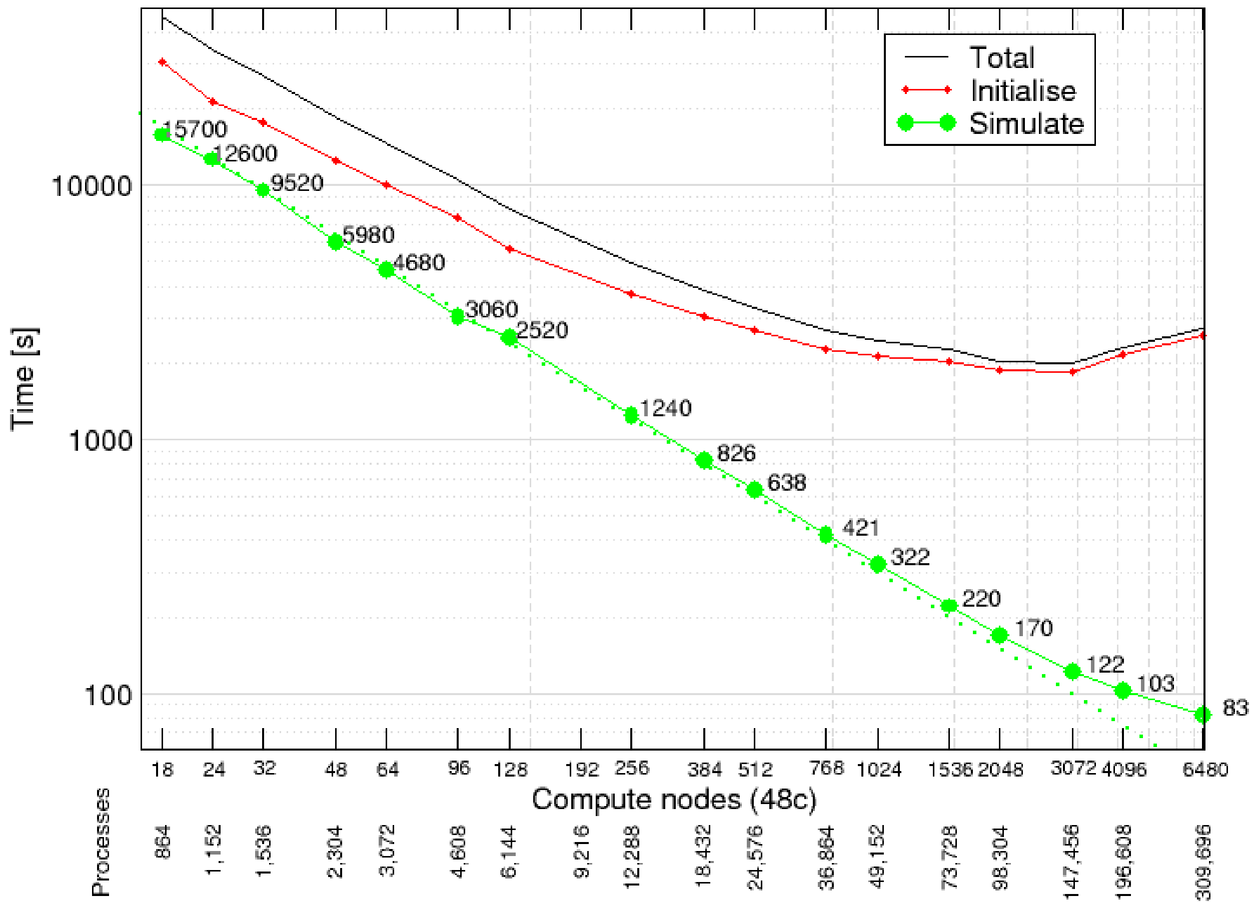
DOI 10.5281/zenodo.4105743

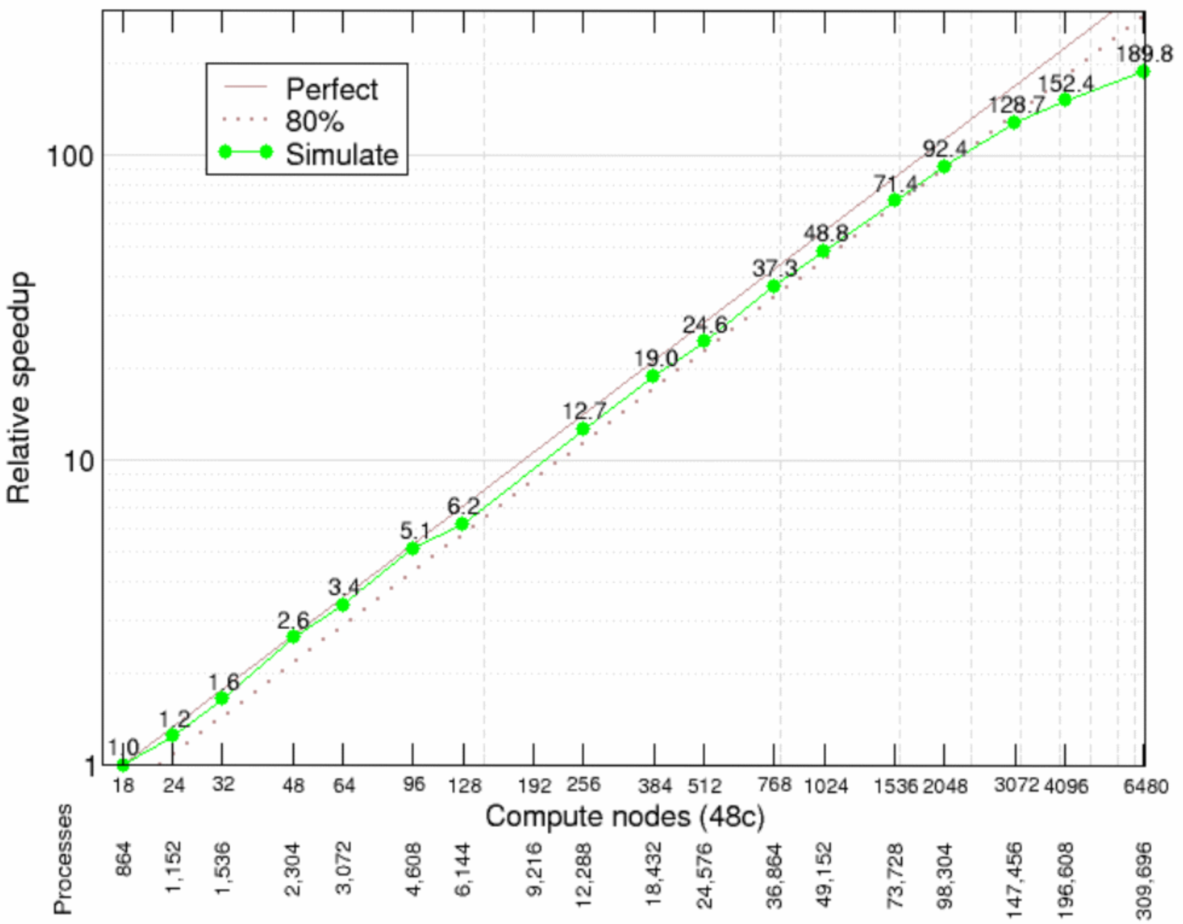
- 3D macroscopic blood flow in human arterial system developed by UC London (UK)
  - lattice-Boltzmann method tracking fluid particles on a lattice grid with complex boundary conditions
  - exascale flagship application of EU H2020 HPC Centre of Excellence for Computational Biomedicine
- HemeLB open-source code and test case: www.hemelb.org
  - C++ parallelized with MPI [+ CUDA unused]
    - Intel Studio 2019u4 compiler and MPI library (v19.0.4.243)
    - configured with 2 ‘reader’ processes (intermediate MPI file writing disabled)
    - MPI-3 shared-memory model employed within compute nodes
      to reduce memory requirements when distributing lattice blocks from reader processes
  - Focus of analysis 5,000 time-step (500µs) simulation of cerebrovascular “circle of Willis” geometry
    - 6.4µm lattice resolution (21.15 GiB): 10,154,448,502 lattice sites
- Executed on *SuperMUC-NG* Lenovo ThinkSystem SD650 (LRZ):
  - 2x 24-core Intel Xeon Platinum 8174 (‘Skylake’) @ 3.1GHz
  - 48 MPI processes/node, 6452 (of 6480) compute nodes: 309,696 MPI processes
  - 190x speed-up from 864 cores: 80% scaling efficiency to over 100,000 cores

⇒ ***Identification & quantification of impact of load balance and its variation***



HEMELB@SNG STRONG SCALING OF FOA *RUNSIMULATION*





[Execution of 9,216 processes on 192 compute nodes not possible due to insufficient compute nodes with adequate memory in 'fat' partition (768 GiB vs. regular 96 GiB node memory)]

HEMELB@SNG STRONG SCALING EFFICIENCY OF FOA *RUNSIMULATION*

Compute nodes	24	32	48	64	96	128	192	256	384	512	768	1024	1536	2048	3072	4096	6452
Processes	**1152**	**1536**	**2304**	**3072**	**4608**	**6144**	**9216**	**12288**	**18432**	**24576**	**36864**	**49152**	**73728**	**98304**	**147456**	**196608**	**309696**
Global scaling efficiency	**0.79**	**0.79**	**0.84**	**0.80**	**0.82**	**0.75**		**0.73**	**0.72**	**0.73**	**0.74**	**0.68**	**0.68**	**0.65**	**0.62**	**0.57**	**0.45**
- Parallel efficiency	0.79	0.80	0.87	0.83	0.86	0.80		0.75	0.74	0.74	0.77	0.71	0.72	0.70	0.72	0.70	0.73
- - Load balance efficiency	0.79	0.80	0.88	0.84	0.86	0.80		0.75	0.74	0.75	0.78	0.72	0.74	0.72	0.74	0.73	0.80
- - Communication efficiency	1.00	1.00	1.00	1.00	1.00	1.00		1.00	1.00	0.99	0.99	0.99	0.98	0.98	0.97	0.96	0.92
- Computation scaling	*1.00*	0.99	0.96	0.96	0.95	0.93		0.98	0.98	0.98	0.96	0.96	0.94	0.93	0.87	0.81	0.61
- - Instructions scaling	*1.00*	1.00	1.00	1.00	1.00	1.00		1.00	1.00	1.00	0.99	0.97	0.94	0.89	0.79	0.67	0.45
- - IPC scaling	*1.00*	0.99	0.96	0.96	0.95	0.93		0.98	0.98	0.99	0.98	0.99	1.00	1.04	1.11	1.21	1.36
IPC	1.411	1.395	1.353	1.355	1.342	1.316		1.377	1.387	1.396	1.383	1.390	1.417	1.473	1.566	1.704	1.919
										Key:	<0.65	<0.75	<0.85	<0.95	<1.00	>1.00

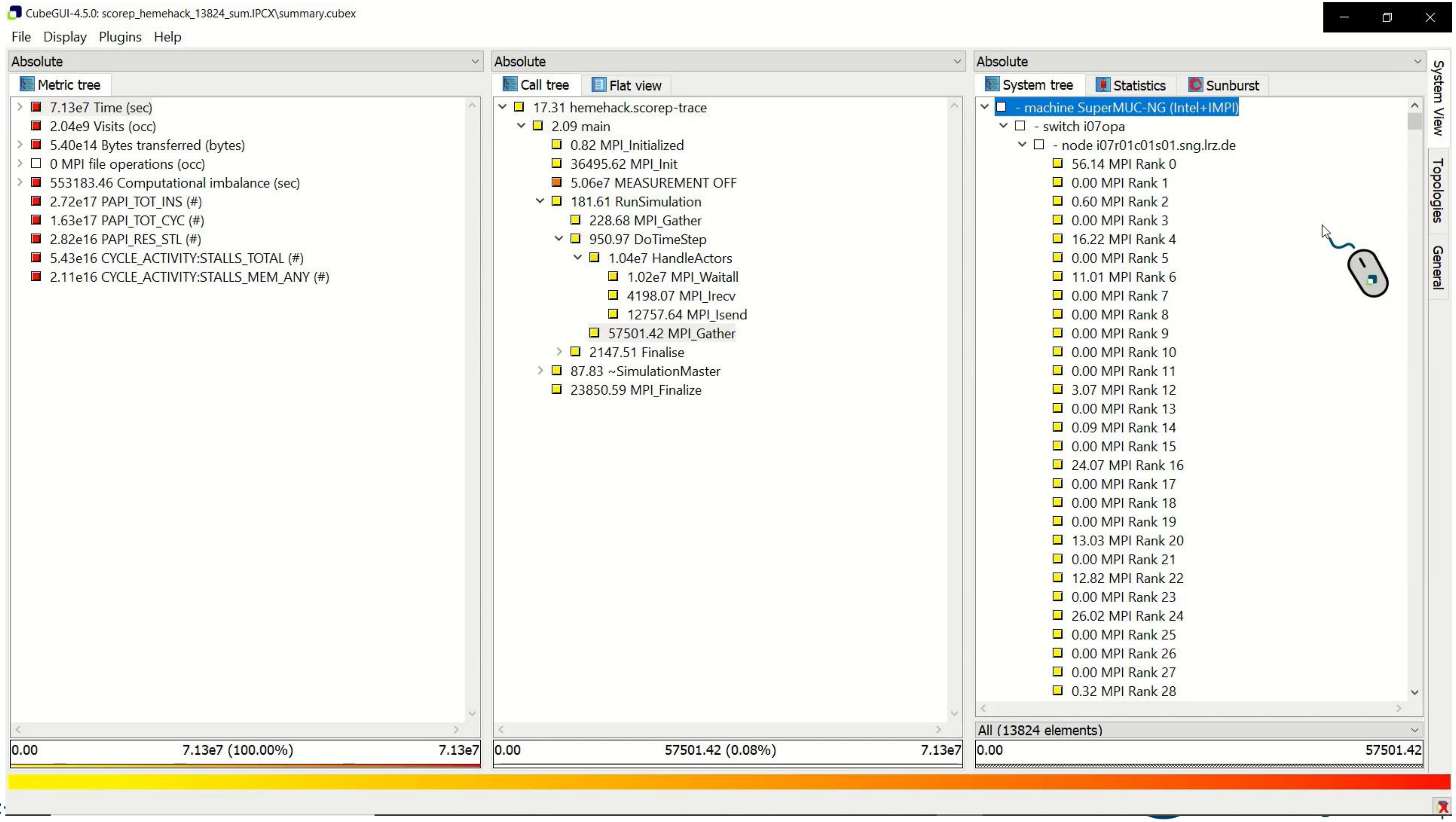
Global scaling efficiency fairly good around 80%, before degrading at larger scales

- Parallel efficiency deteriorating following Load balance efficiency
  - Communication efficiency excellent throughout
- Computation scaling (relative to 1152 processes) very good except at largest scale
  - Degradation of Instructions scaling partially compensated by improving IPC scaling

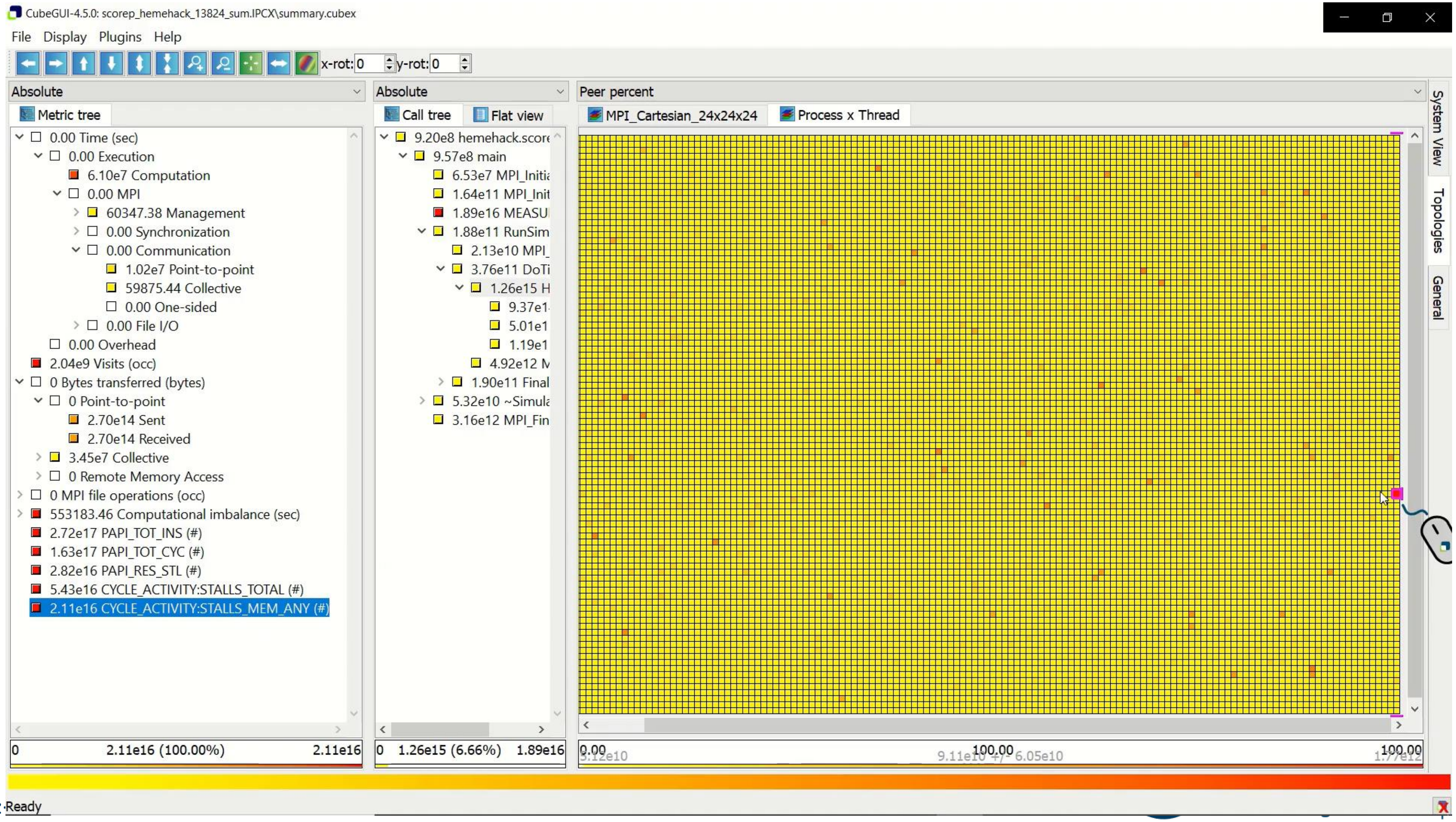
[POP CoE scaling efficiency model: www.pop-coe.eu]

INITIAL TREE PRESENTATION: TIME OF MPI_GATHER PER MPI PROCESS

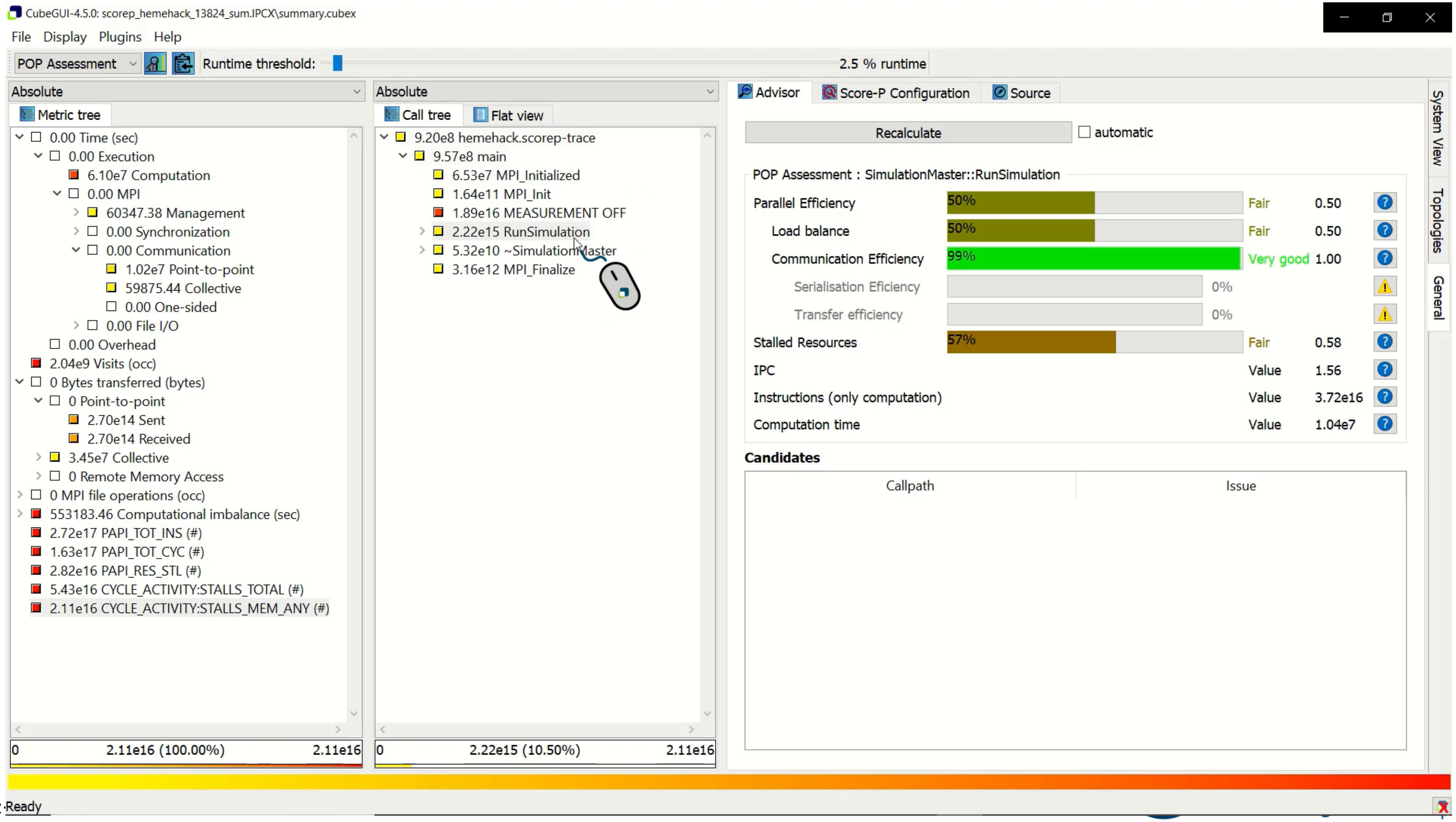
DOI 10.5281/zenodo.4080701



TOPOLOGICAL PRESENTATION:
STALLS_MEM_ANY FOR HANDLEACTORS



ADVISOR: POP EFFICIENCY ASSESSMENT FOR RUNSIMULATION



HEMELB (JUWELS-VOLTA)

CompBioMed

DOI 10.5281/zenodo.4117942

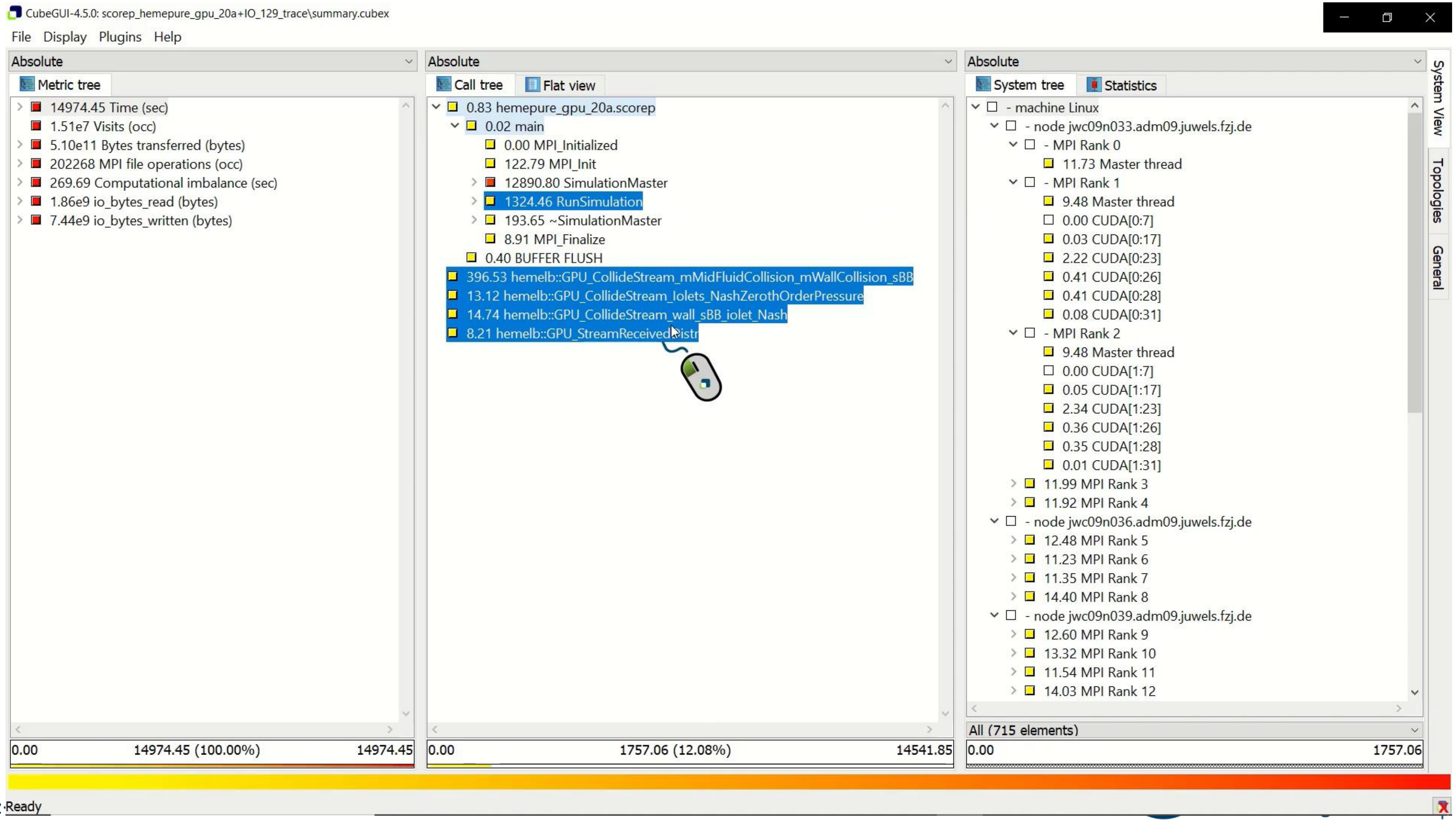
- 3D macroscopic blood flow in human arterial system developed by UC London (UK)
  - lattice-Boltzmann method tracking fluid particles on a lattice grid with complex boundary conditions
  - exascale flagship application of EU H2020 HPC Centre of Excellence for Computational Biomedicine
- HemeLB open-source code and test case: www.hemelb.org
  - C++ parallelized with MPI + CUDA (in development)
    - GCC/8.3.0 compiler, CUDA/10.1.105 and ParaStationMPI/5.4 library
    - configured with 2 ‘reader’ processes and intermediate MPI file writing
    - rank 0 ‘monitor’ process doesn’t participate in simulation

  - Focus of analysis 2,000 time-step (each 100µs) simulation of CBM2019_Arteries_patched geometry
    - 1.78 GiB: 66,401,494 lattice sites, 1+38 iolets
- Executed on *JUWELS-Volta* (@JSC):
  - 2x 20-core Intel Xeon Platinum 8168 (‘Skylake’) CPUs + 4 Nvidia V100 ‘Volta’ GPUs
  - 4* MPI processes/node (one per GPU), 32 (of 56) compute nodes: 129 MPI processes

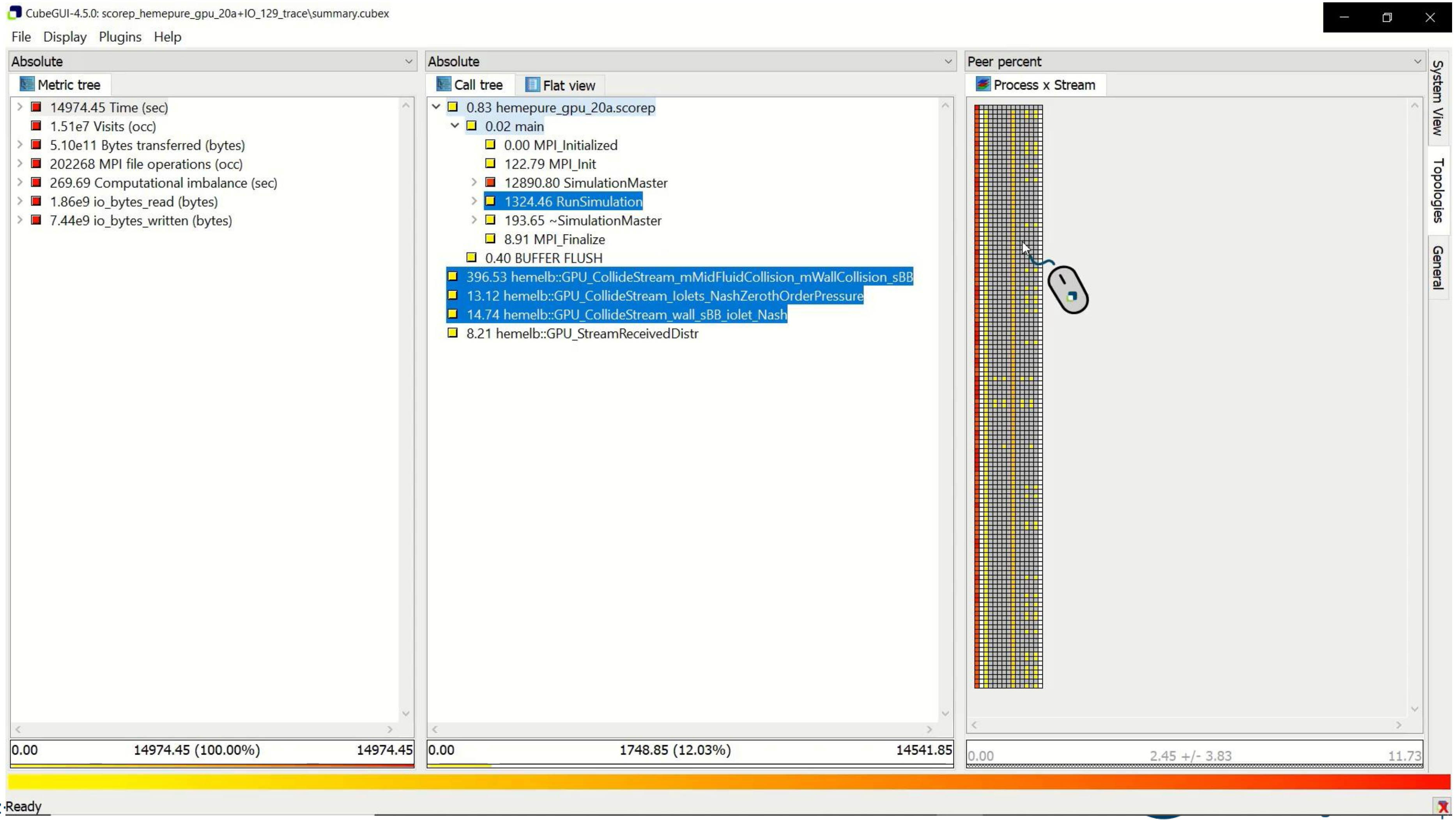
⇒ Identification & quantification of impact of load balance and its variation

TREE: TIME FOR ASYNCH. CUDA KERNELS ON SEPARATE CUDA STREAMS

DOI 10.5281/zenodo.4081080

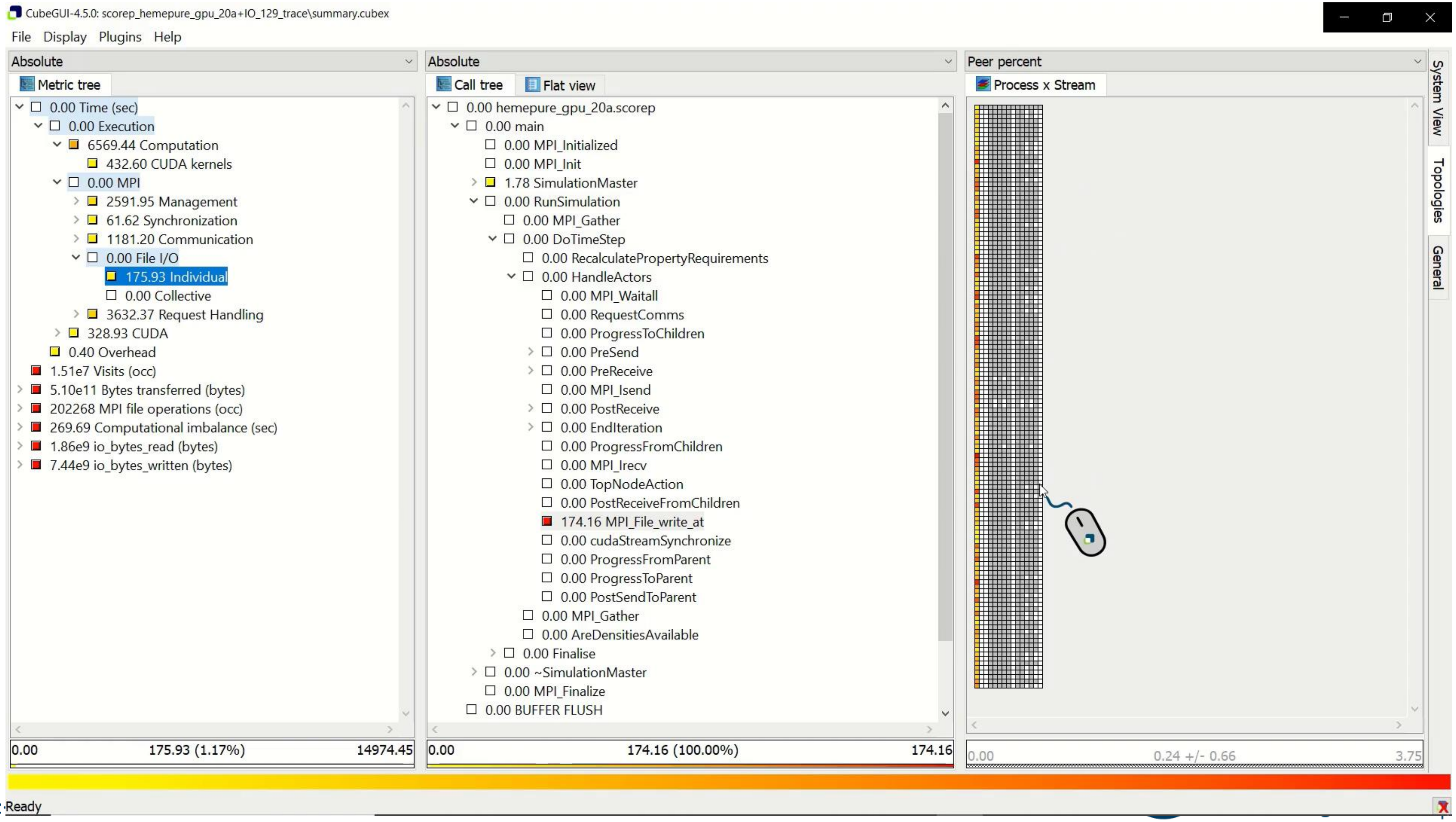


TOPO: TIME FOR ASYNCH. CUDA KERNELS ON SEPARATE CUDA STREAMS



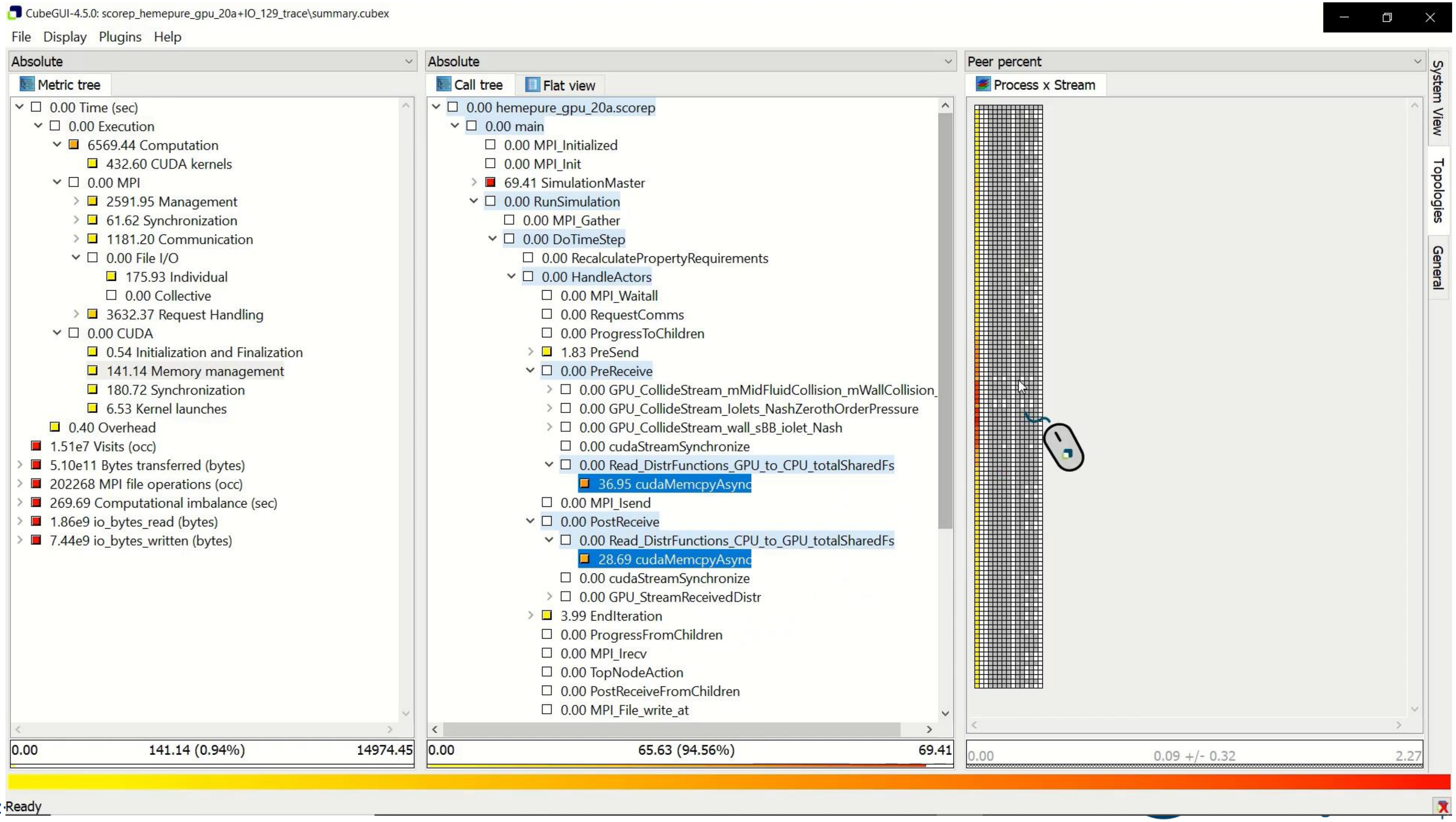
JÜLICH SUPERCOMPUTING CENTRE

TOPO: TIME FOR MPI FILE WRITING ON CPU VARIES PER MPI PROCESS

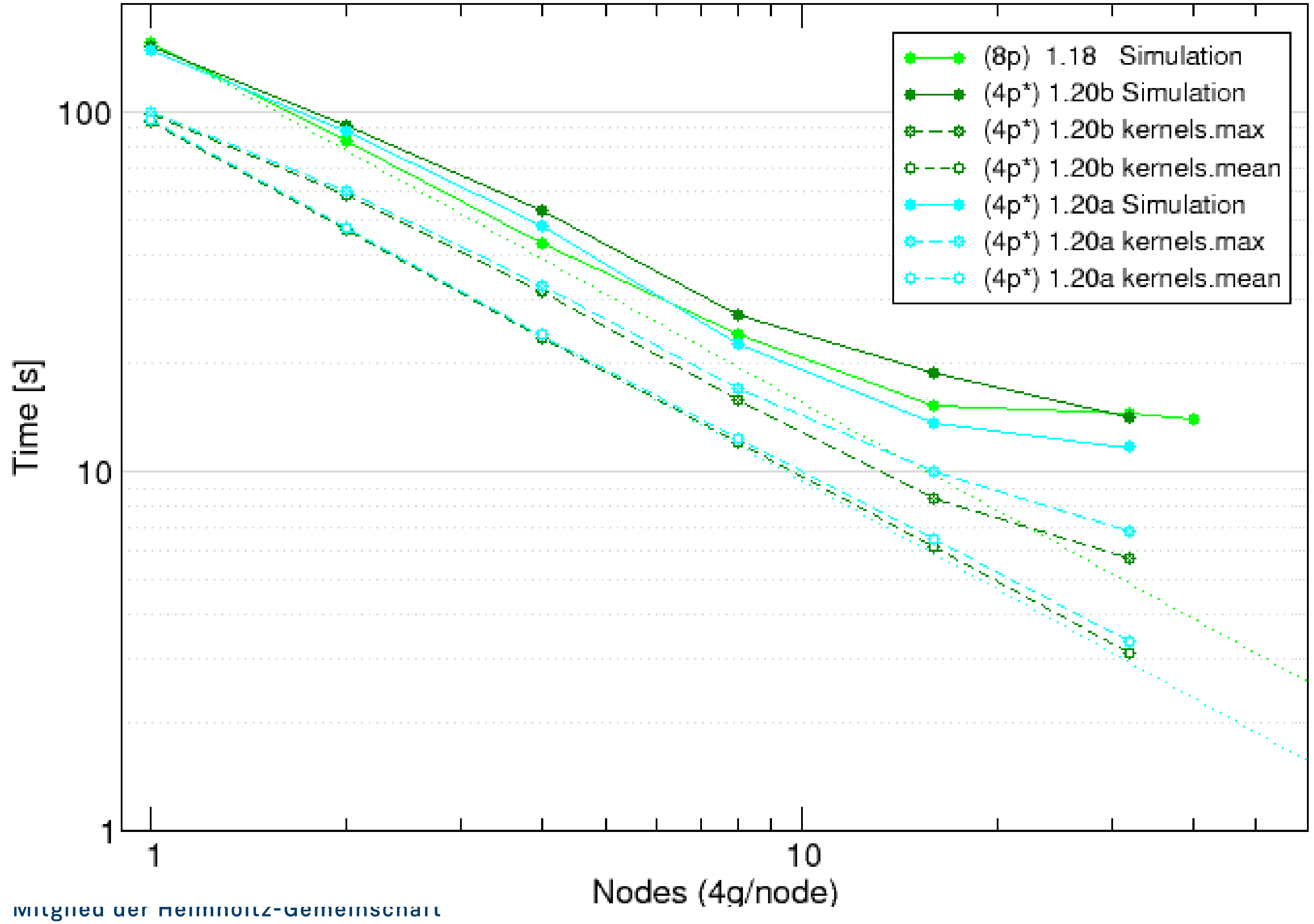


JÜLICH SUPERCOMPUTING CENTRE

TOPO: TIME FOR CUDA ASYNCHRONOUS MEMORY COPIES IS IMBALANCED



HEMELB@JUWELS-VOLTA STRONG SCALING OF FOA *RUNSIMULATION*



- Reference execution with 8ppn
  - multiple processes offloading GPU kernels generally unproductive
- Comparison of versions (4ppn)
  - v1.20a generally better
- Synchronous MPI file writing is the primary bottleneck
- CUDA kernels on GPUs
  - less than half of Simulation time (therefore GPUs mostly idle)
  - total kernel time scales very well (0.93 scaling efficiency)
  - load balance deteriorates (0.95 for single node, 0.50 for 32 nodes)

HEMELB@JUWELS/VOLTA STRONG SCALING EFFICIENCY OF *RUNSIMULATION*

	1n 5p	2n 9p	4n 17p	8n 33p	16n 65p	32n 129p
Simulation time [s]	147.87	88.38	48.13	22.66	13.68	11.67
Global scaling efficiency	0.64	0.53	0.49	0.52	0.43	0.25
– Parallel efficiency	0.64	0.53	0.50	0.54	0.47	0.29
– – Load balance efficiency (GPU)	0.95	0.78	0.73	0.73	0.65	0.50
– – Communication efficiency (GPU)	0.67	0.68	0.68	0.75	0.73	0.58
– Computation scaling (GPU)	1.00	1.00	0.99	0.96	0.92	0.87

Key:

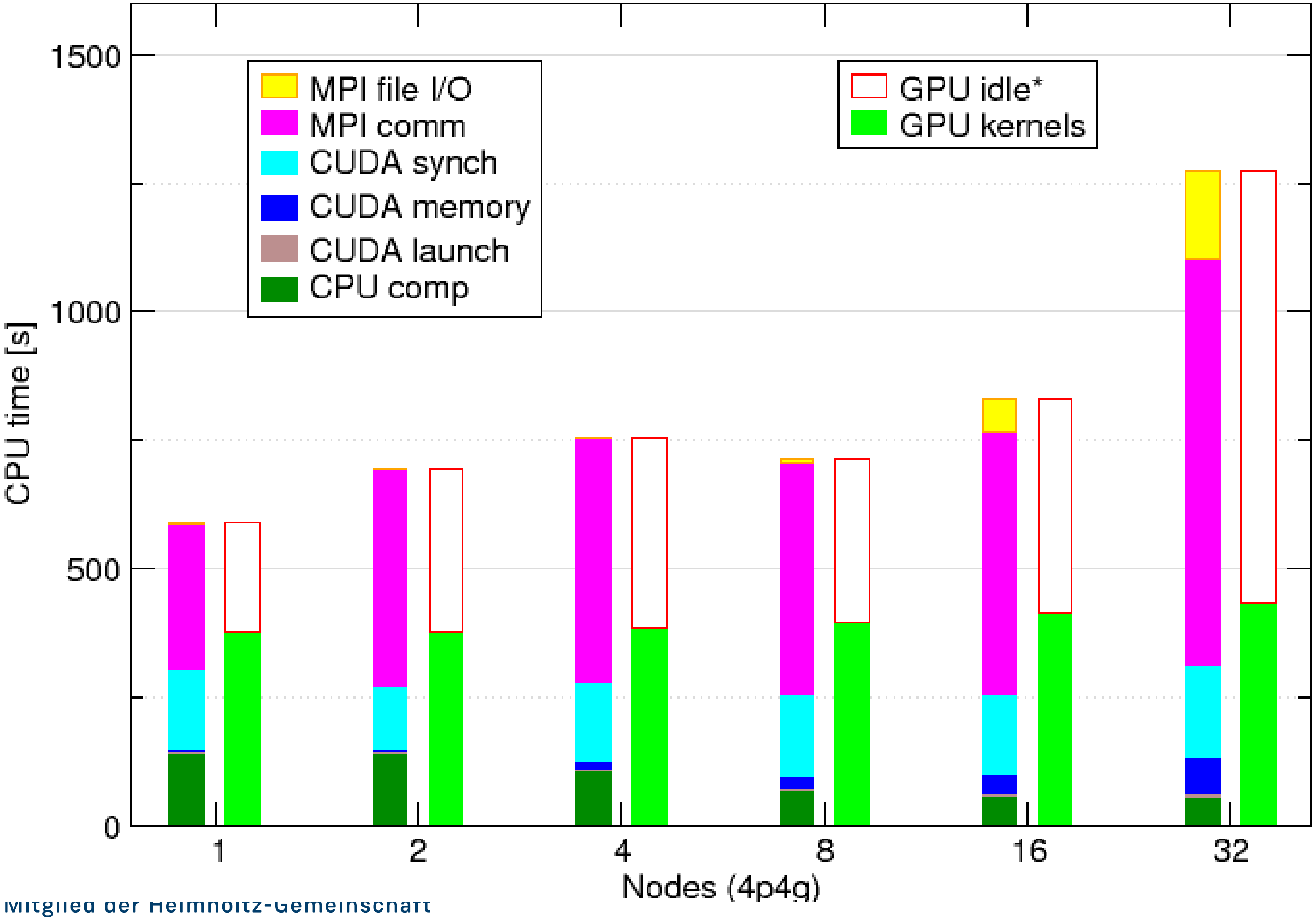
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0

Only considering GPUs (ignoring all CPU cores, 90% of which are completely unused)

- Single (quad-GPU) node already suffers significant communication inefficiency
  - includes MPI file writing, but doesn’t degrade much as additional nodes are included
- Load balance of GPUs deteriorates progressively
- GPU computation scaling remains reasonably good

[POP CoE scaling efficiency model: www.pop-coe.eu]

HEMELB@JUWELS-VOLTA STRONG SCALING OF FOA *RUNSIMULATION*



- CPU+GPU time breakdown
- CUDA kernels on GPUs
  - less than half of Simulation time (therefore GPUs mostly idle)
  - total kernel time scales very well (0.87 scaling efficiency)
- MPI processes on CPUs
  - computation time decreases
  - CUDA synchronization time fairly constant, but time for memory management increases somewhat
  - MPI communication time dominates, with much more time for file writing with 16+ nodes

SCALASCA CASE STUDY – TEA LEAF

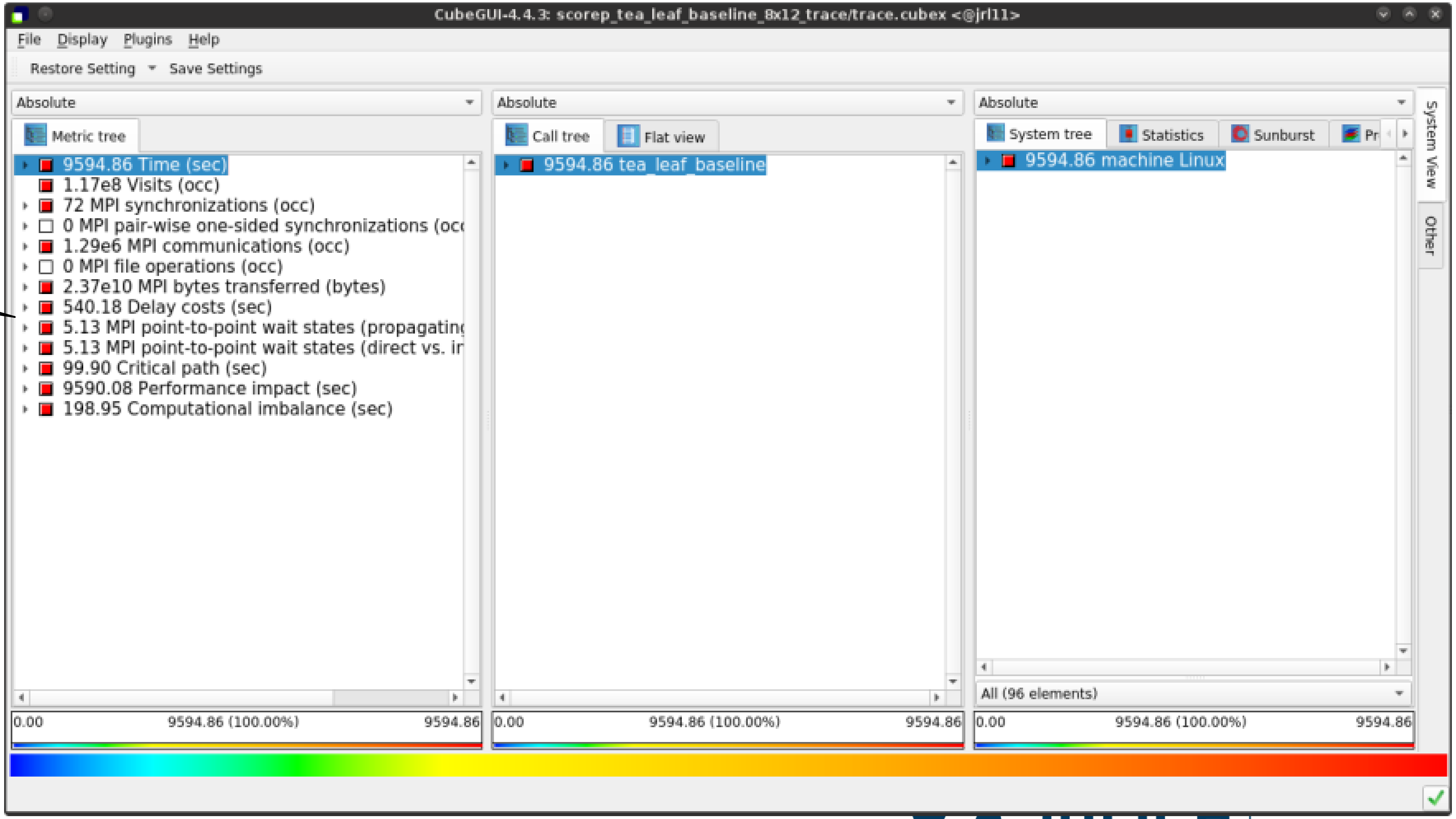
CASE STUDY: TEALEAF

- 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: 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

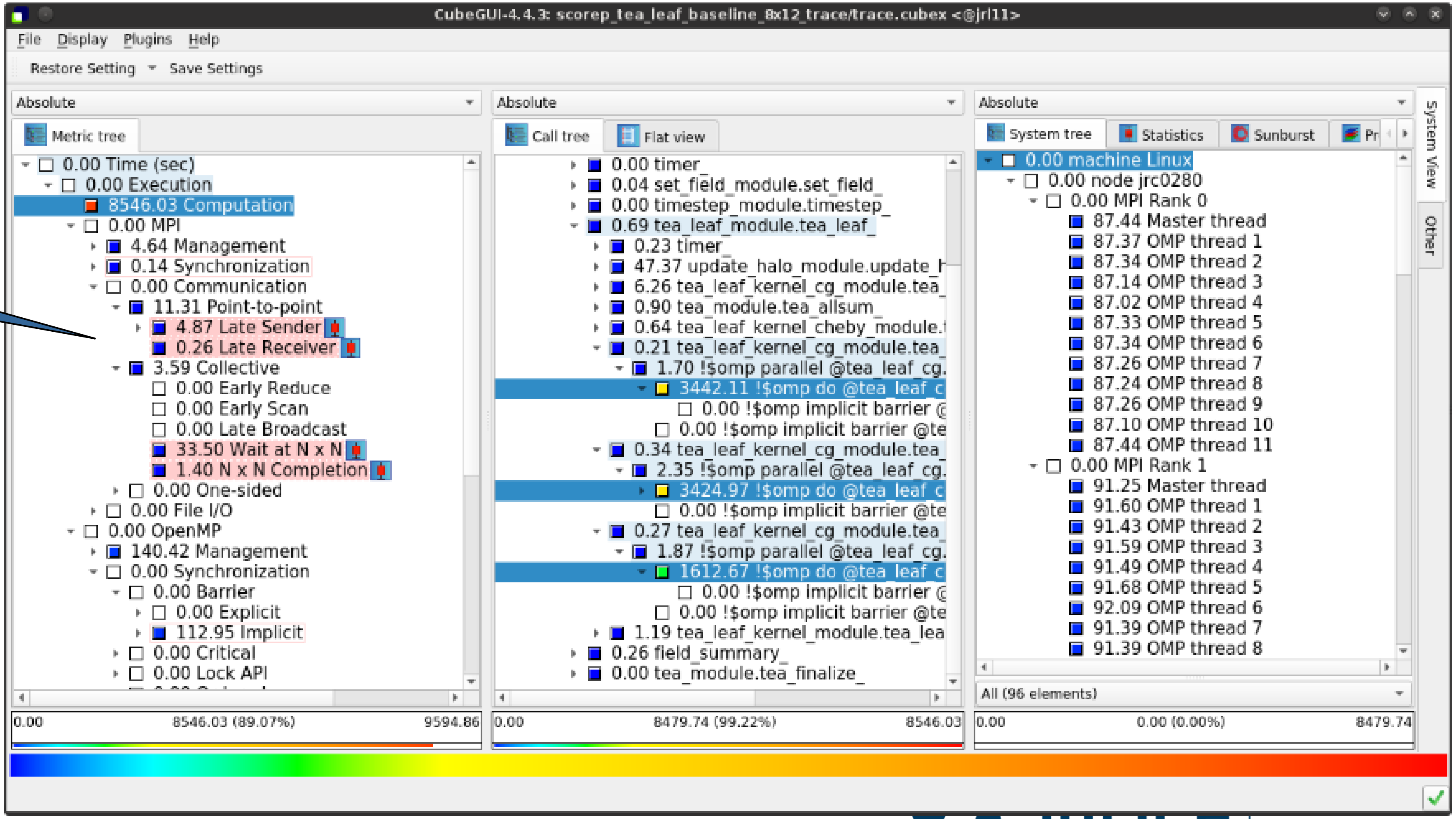
SCALASCA ANALYSIS REPORT EXPLORATION (OPENING VIEW)

Additional top-level metrics produced by the trace analysis…



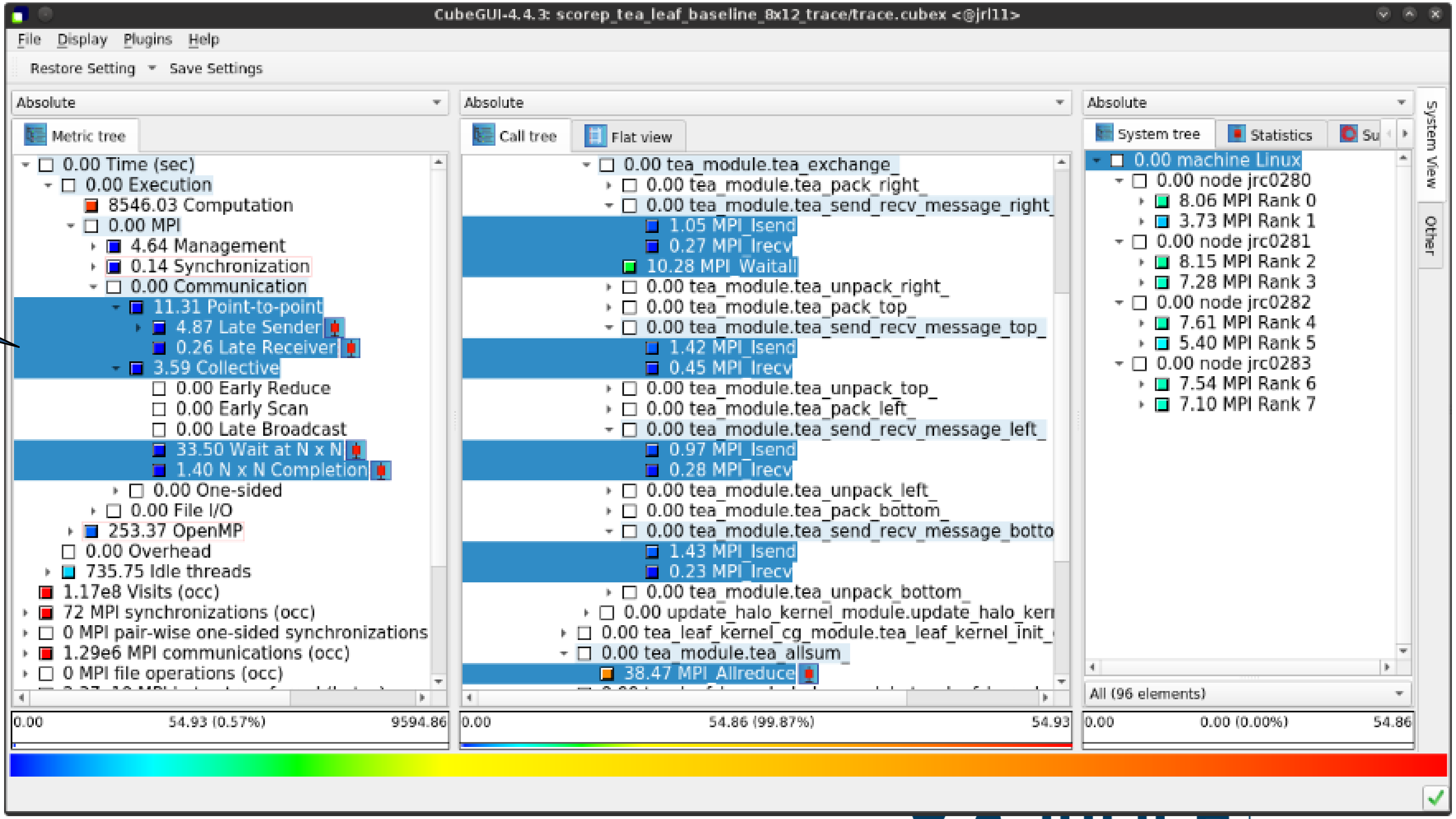
SCALASCA WAIT-STATE METRICS

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



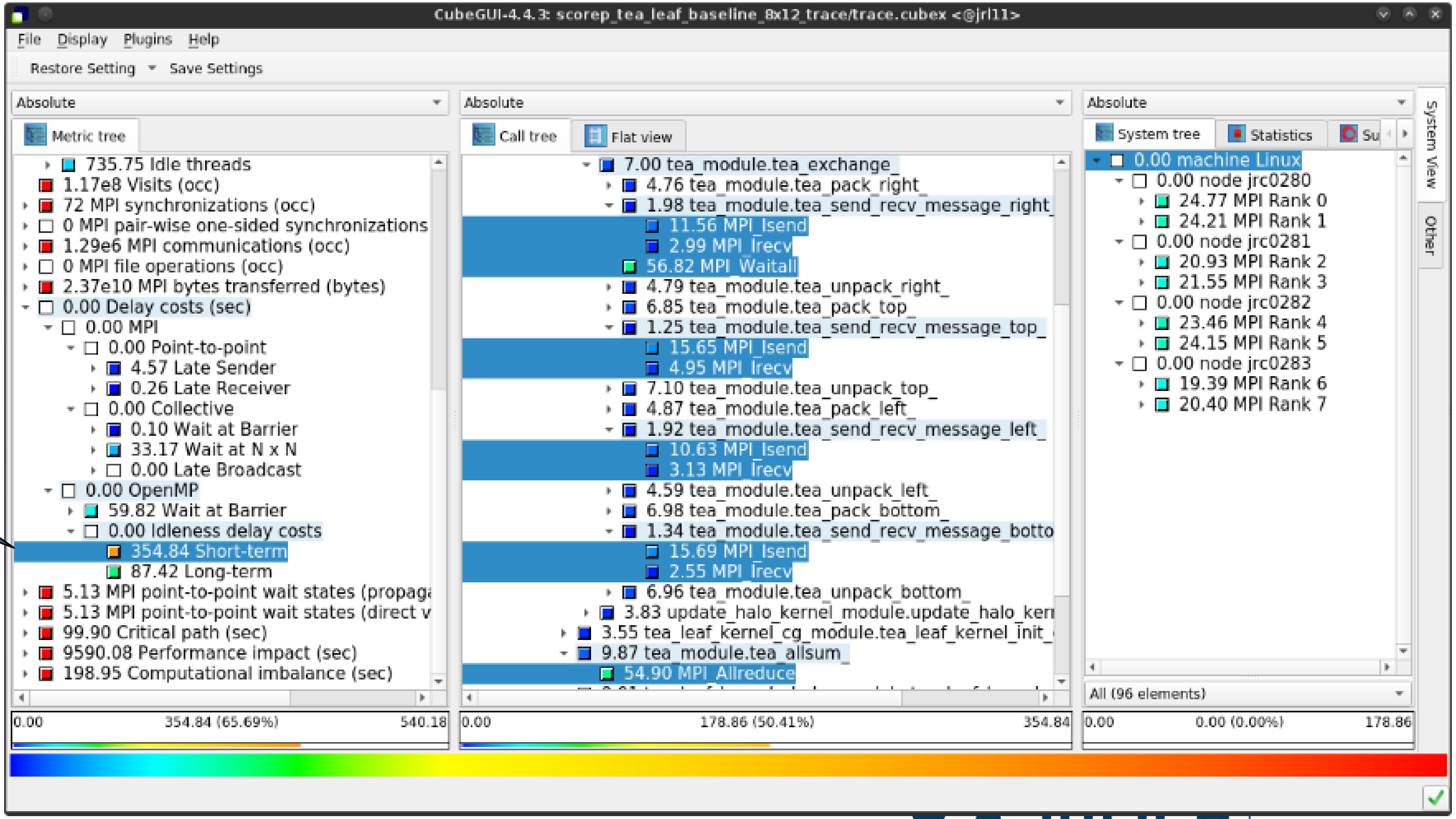
TEALEAF SCALASCA REPORT ANALYSIS (I)

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



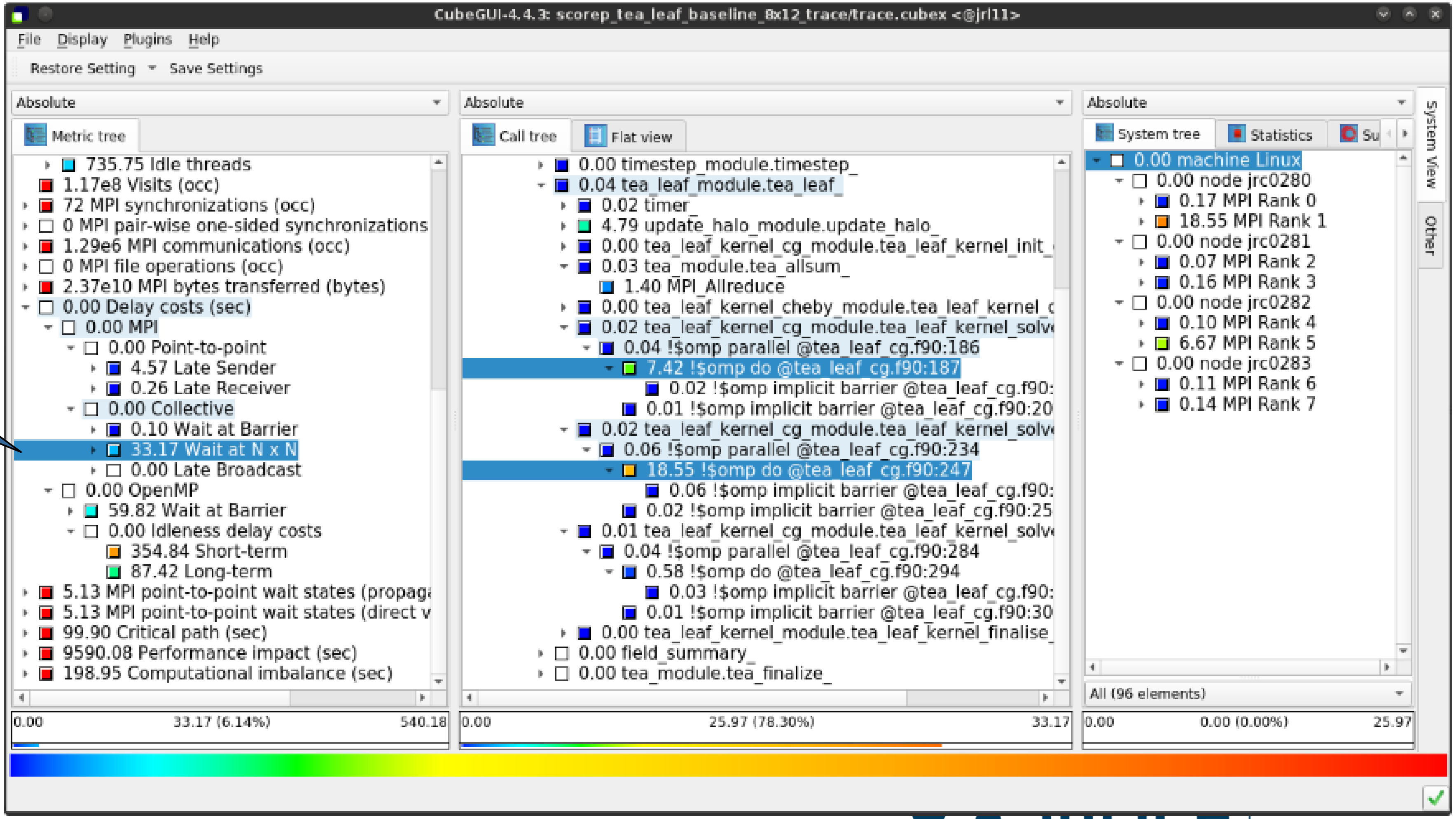
TEALEAF SCALASCA REPORT ANALYSIS (II)

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



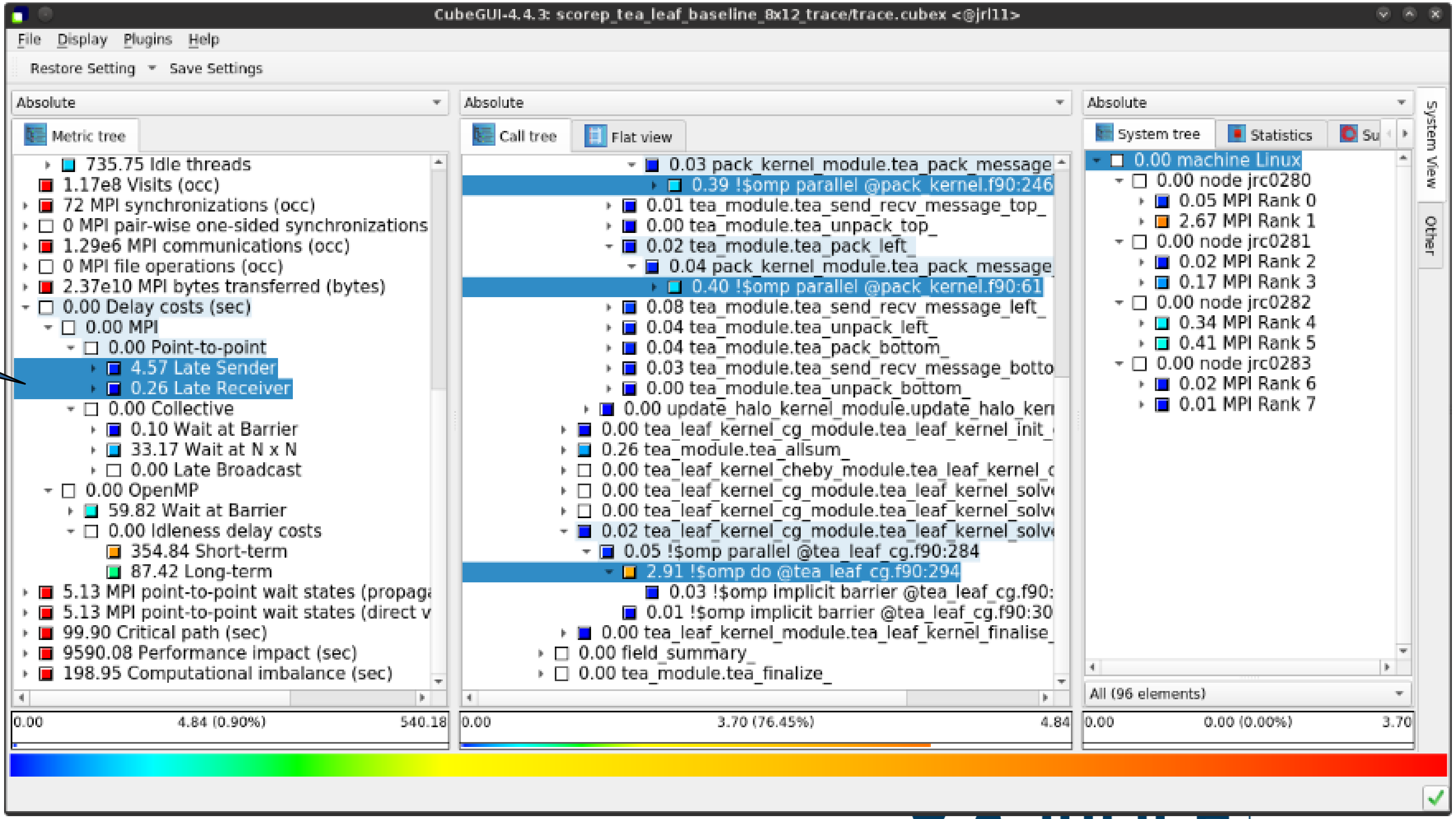
TEALEAF SCALASCA REPORT ANALYSIS (III)

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.)…



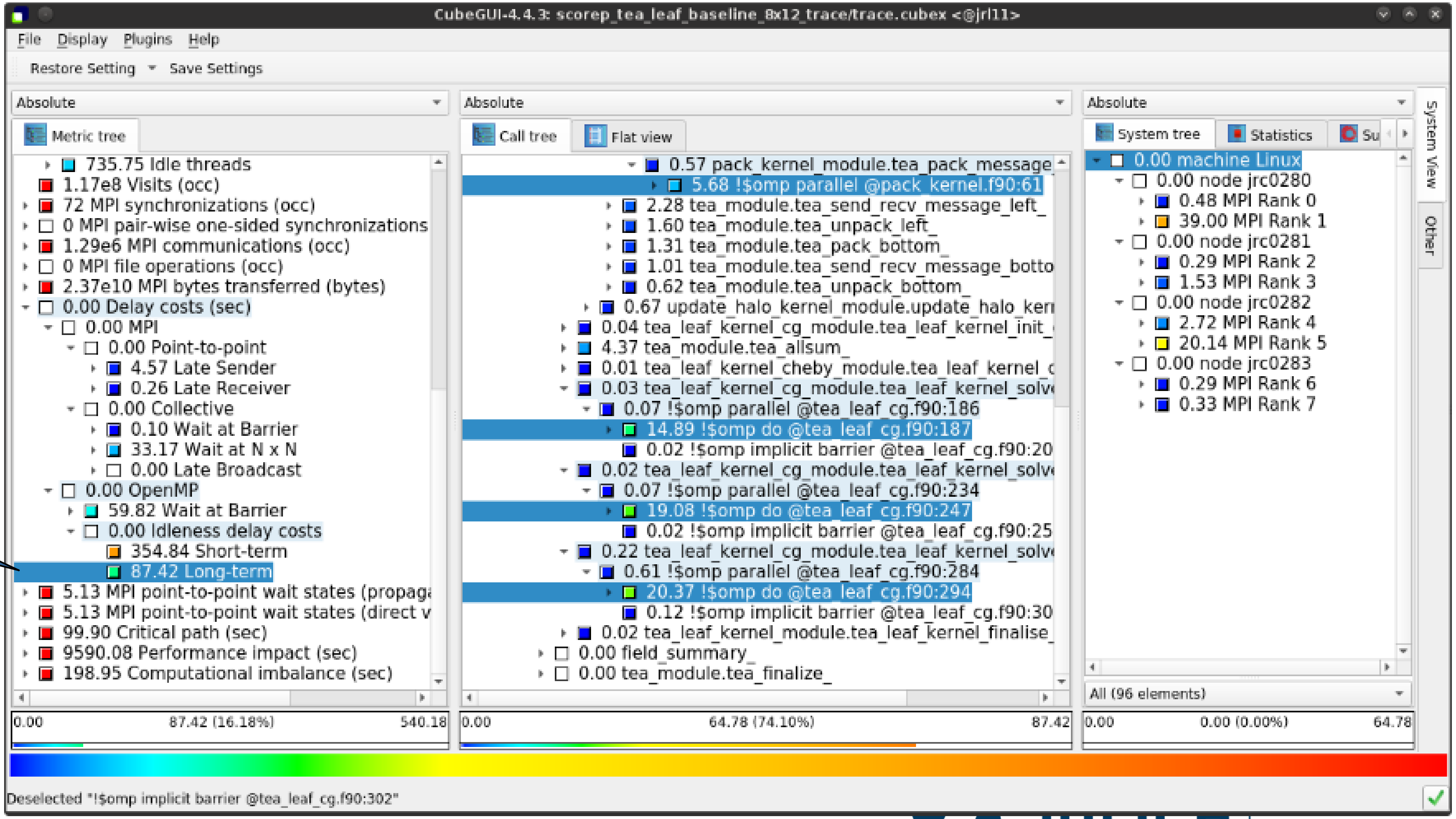
TEALEAF SCALASCA REPORT ANALYSIS (IV)

…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



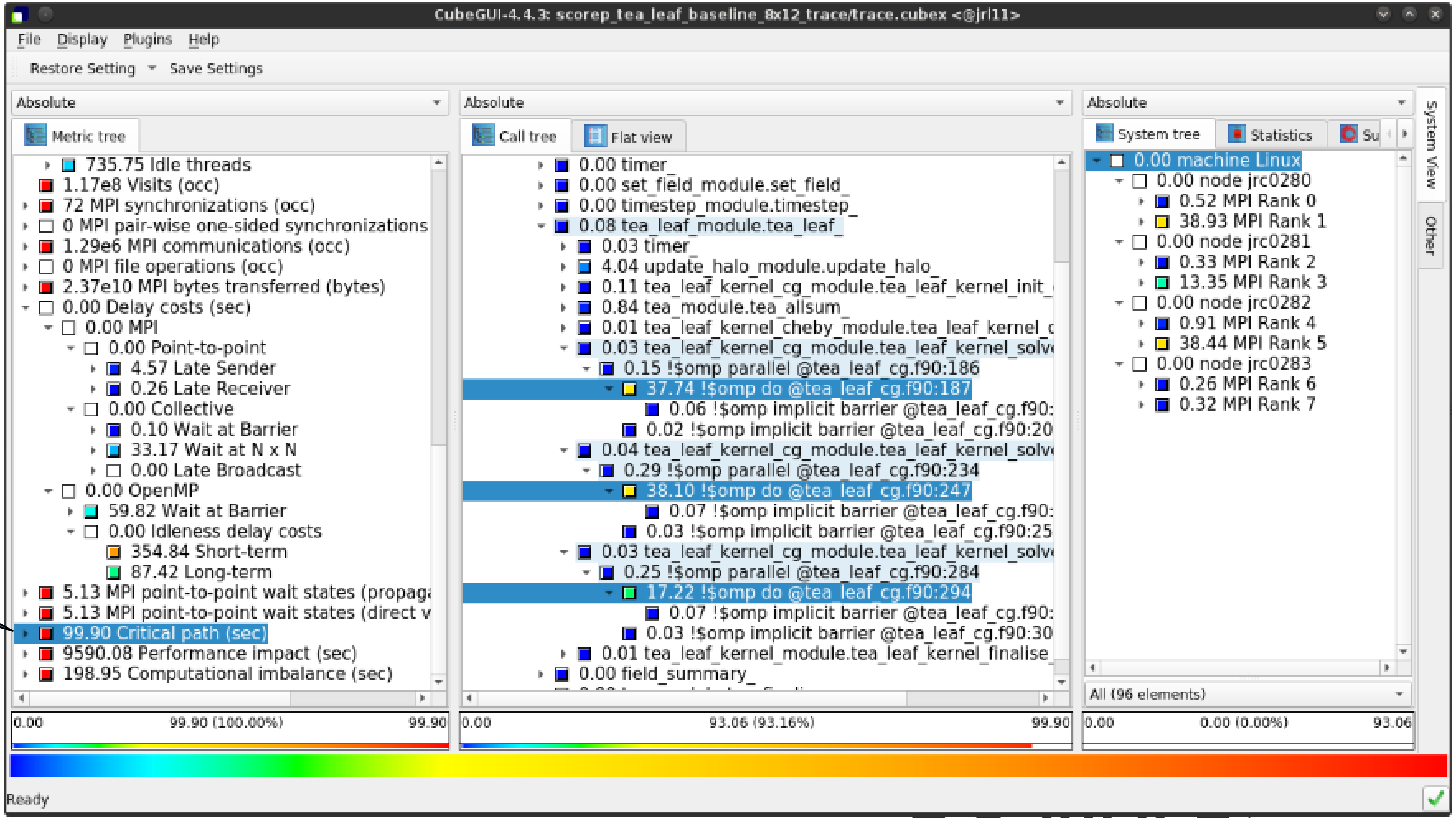
TEALEAF SCALASCA REPORT ANALYSIS (V)

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



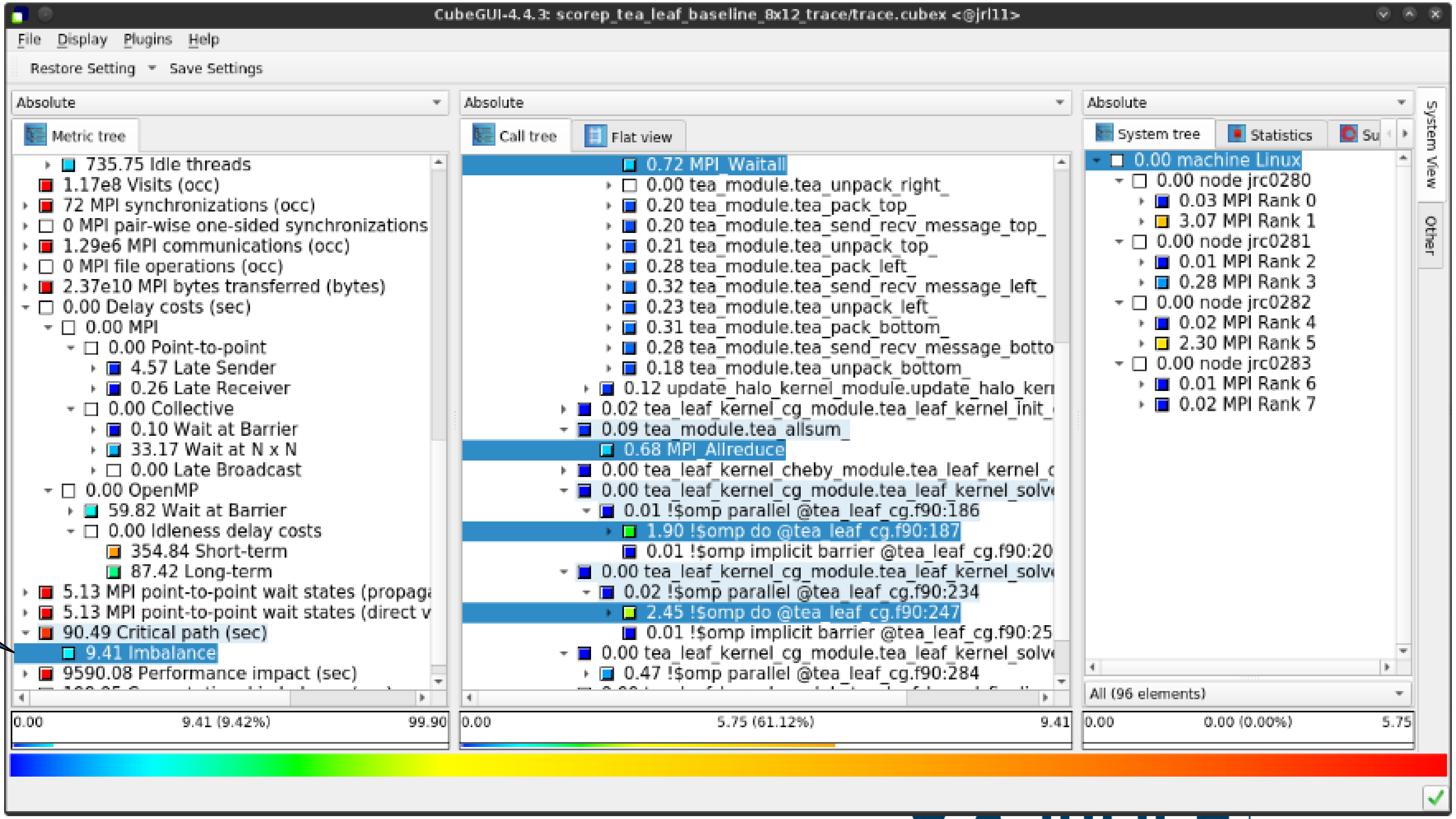
TEALEAF SCALASCA REPORT ANALYSIS (VI)

The Critical Path also highlights the three solver loops…

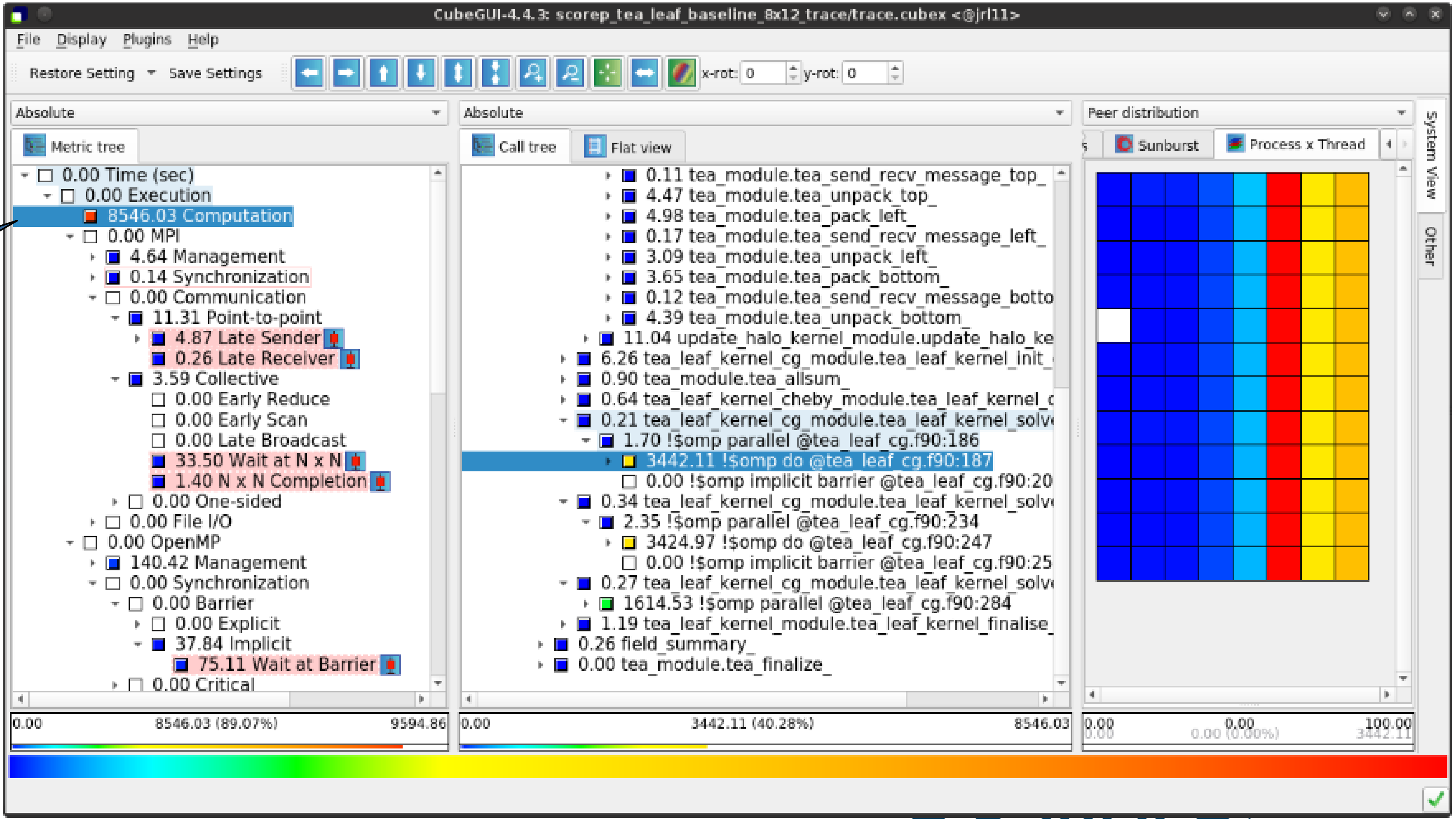


TEALEAF SCALASCA REPORT ANALYSIS (VII)

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

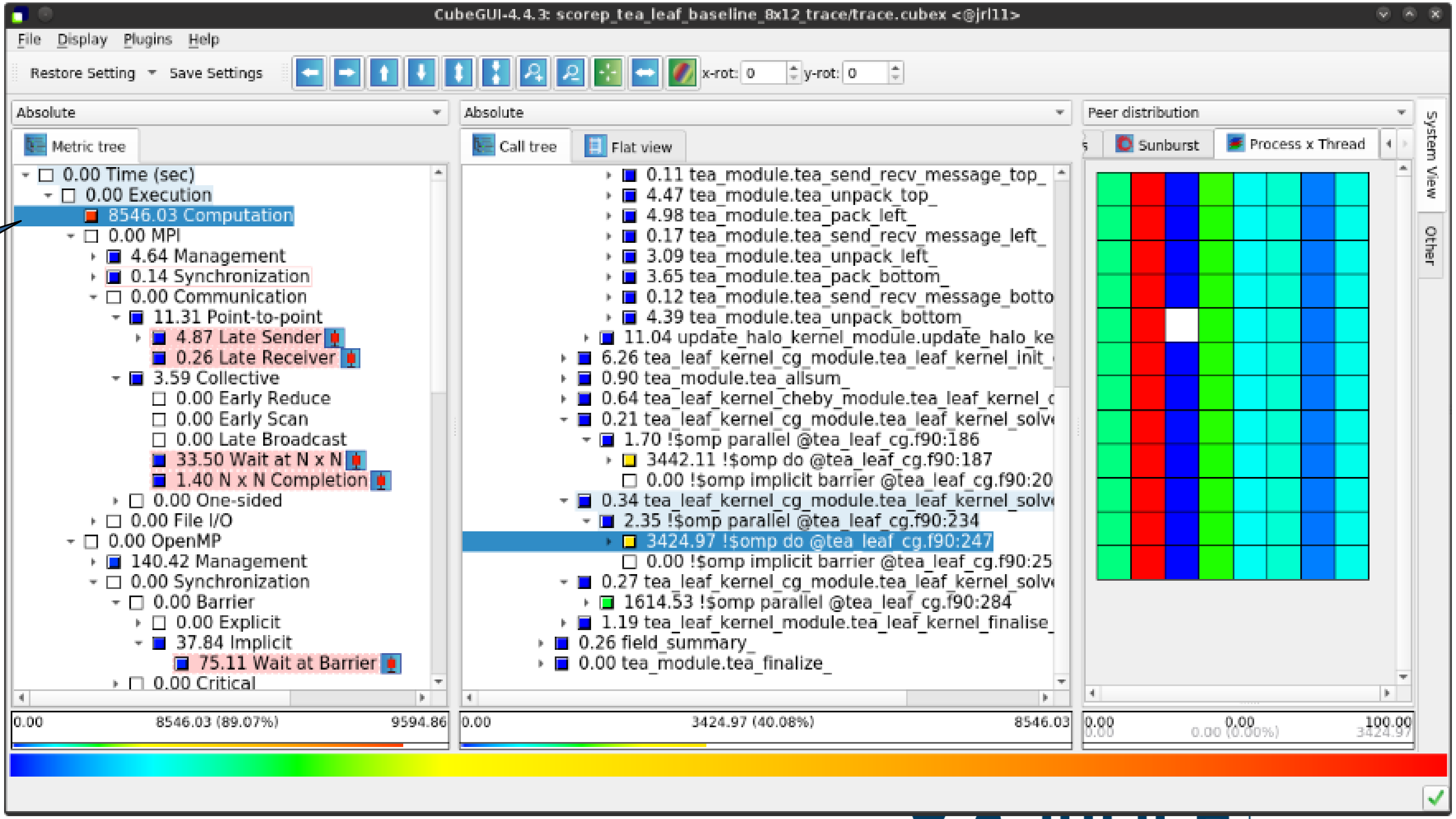


TEALEAF SCALASCA REPORT ANALYSIS (VIII)



Computation time of 1st…

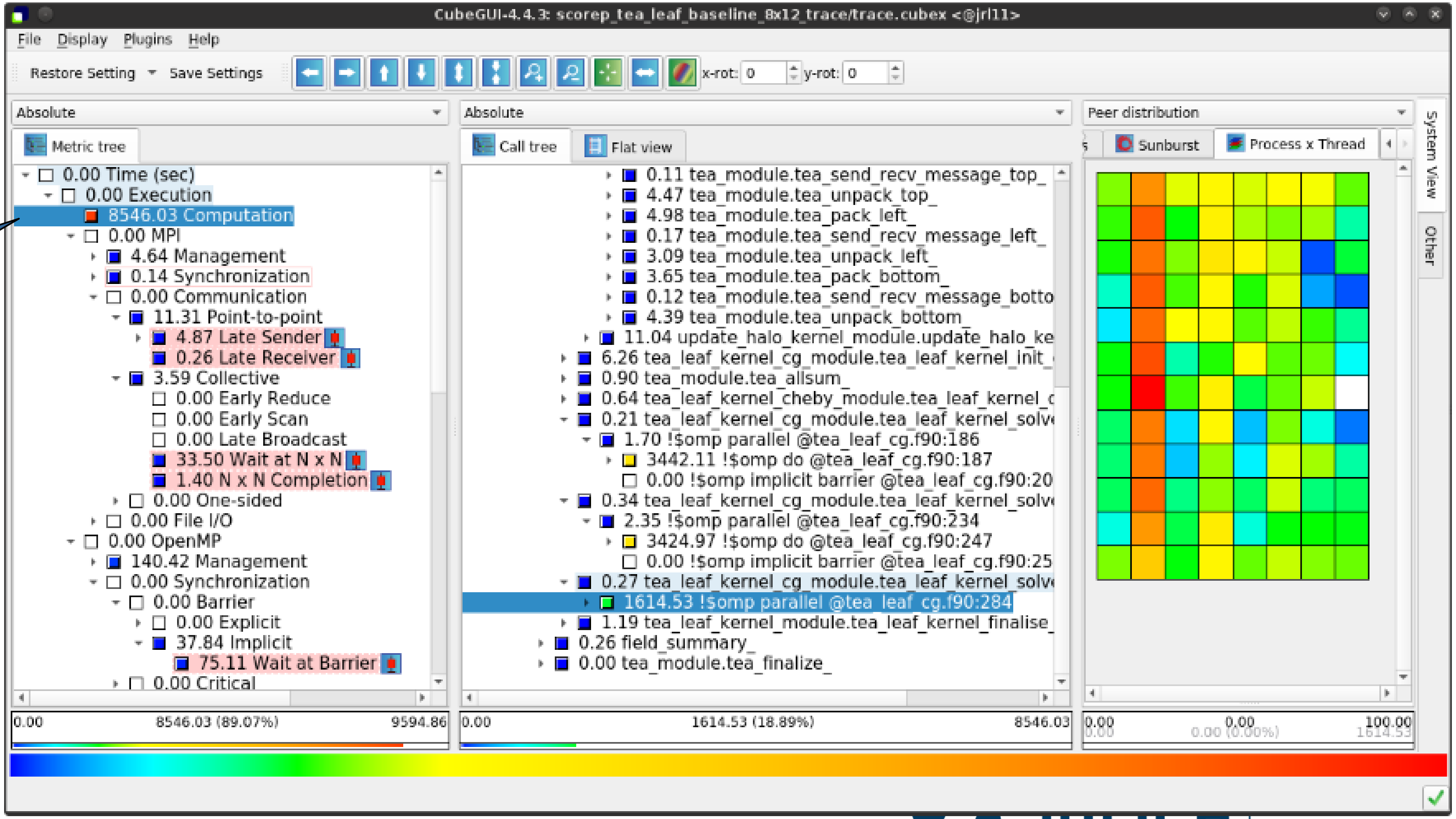
TEALEAF SCALASCA REPORT ANALYSIS (IX)



…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



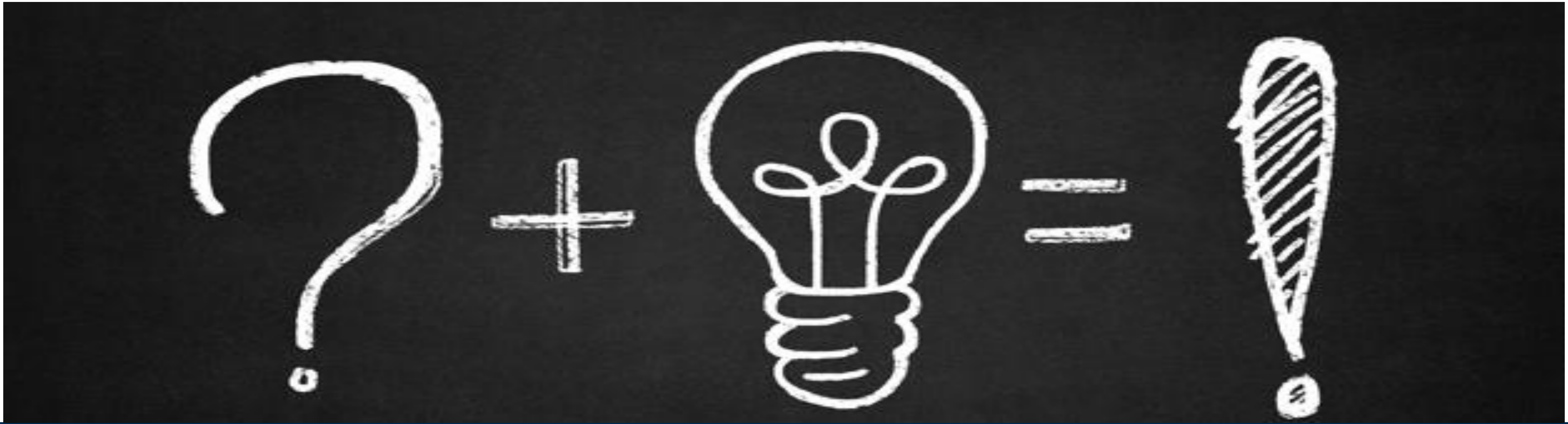
TEALEAF ANALYSIS SUMMARY

- 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%**



SUMMARY

TAKE AWAY MESSAGES

- Many performance analysis tools exist - for a reason
  - Different measurment and analysis techniques
    - Instrumentation vs. Sampling
    - Profiling vs. Tracing
  - Different hardware support
    - Vendor specific tools, e.g. NVIDIA NSIGHT COMPUTE, Intel VTune
    - Verndor agnostic tools, e.g. Score-P ecosystem, TAU, HPCToolkit

- Tools don‘t automagically increase performance
  - Performance analysis is a daunting task, requires experience
  - Performance tuning requires domain and architecture knowledge

    ☞Successful performance engineering often is a collaborative effort