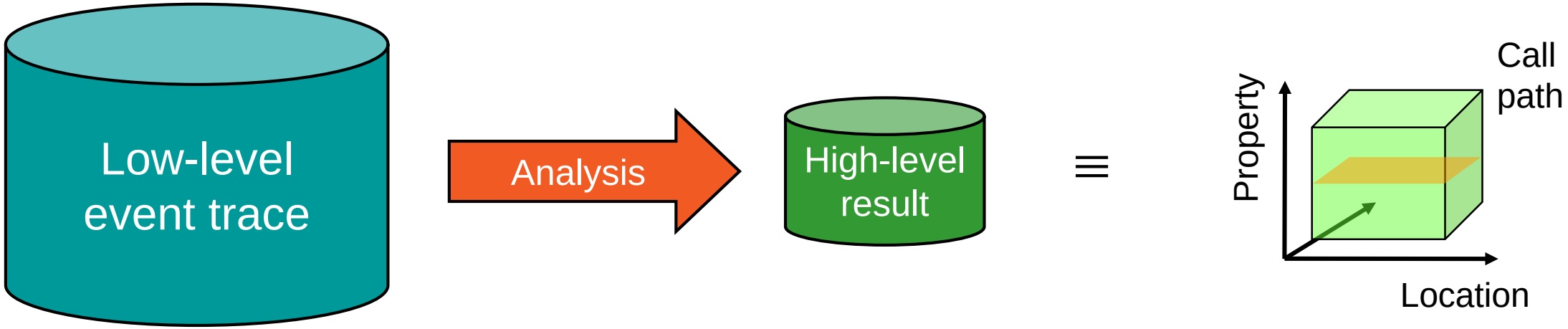
Automatic trace analysis with the Scalasca Trace Tools

The Scalasca Team
Jülich Supercomputing Centre

trace tools
scalasca

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

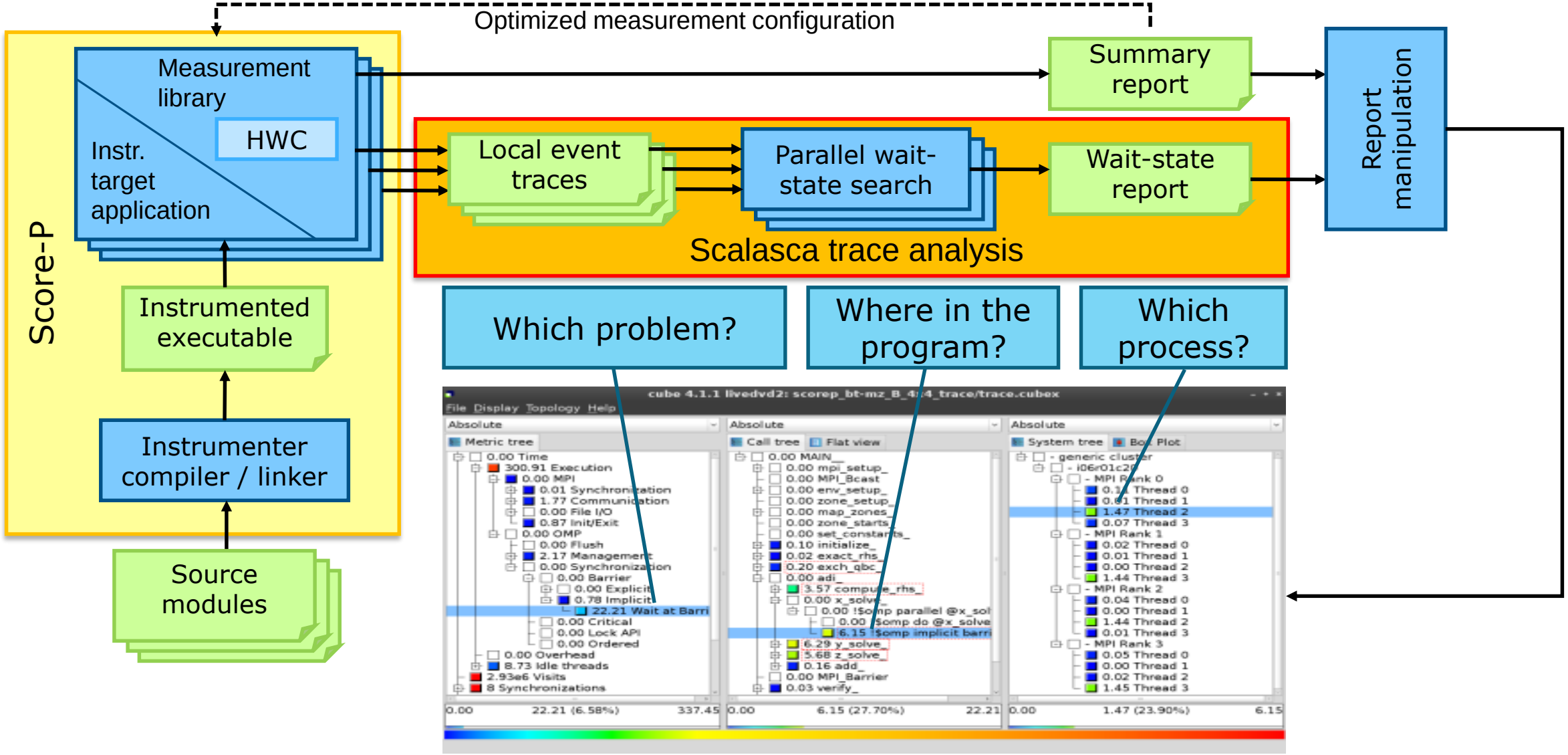
Scalasca Trace Tools: Objective

- 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), also works with later versions

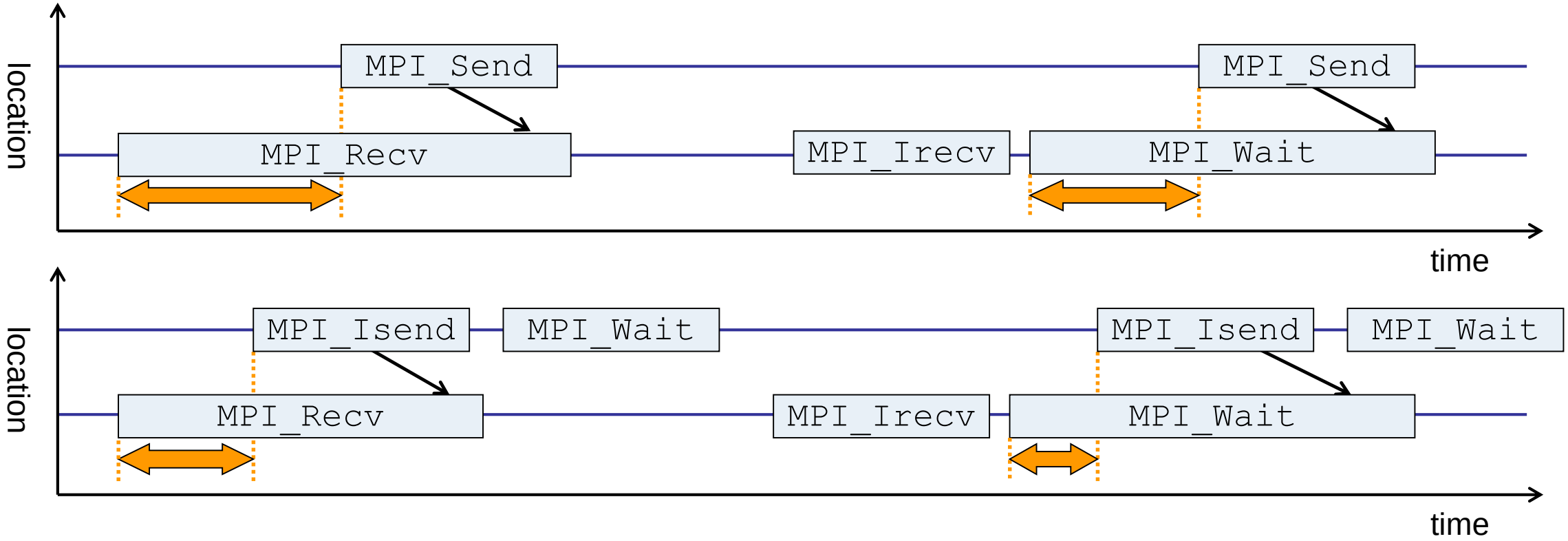
Scalasca Trace Tools: Features

- Open source, 3-clause BSD license
- Fairly portable
  - HPC/Cray XT/XE/XK/XC/EX, IBM Blue Gene, 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

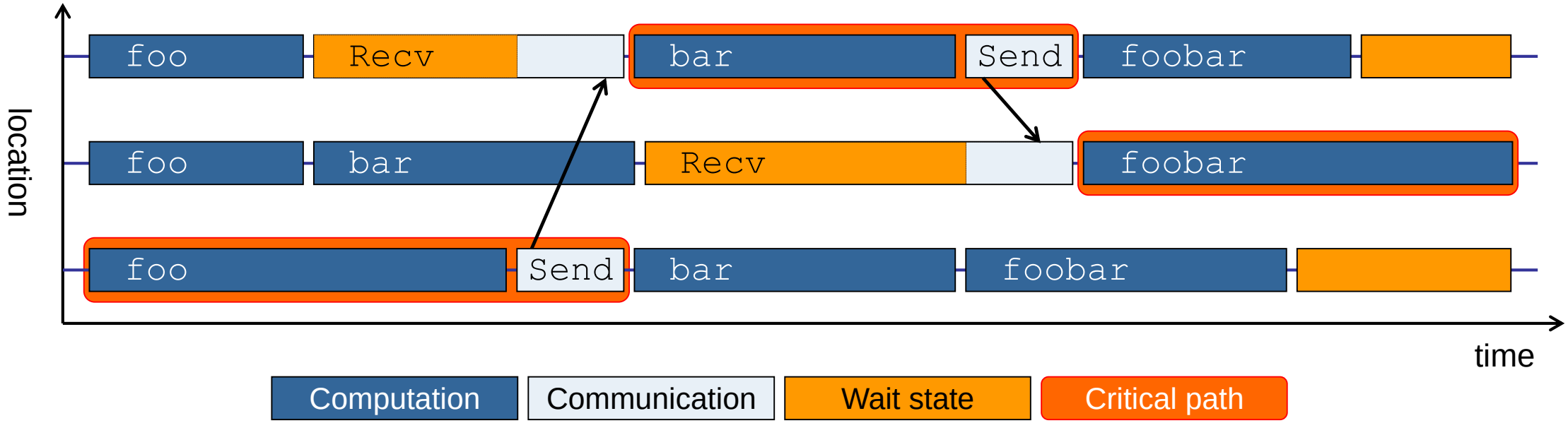


Example: "*Late Sender*" wait state



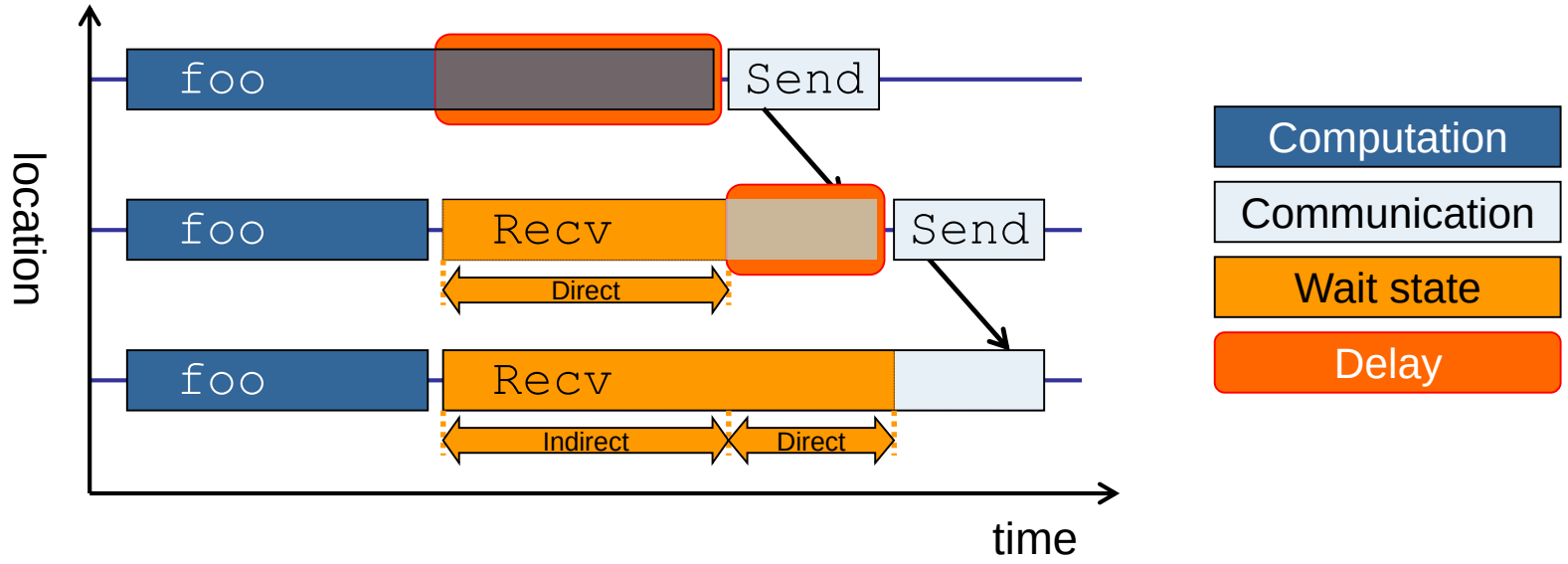
- Waiting time caused by a blocking receive operation posted earlier than the corresponding send
- Applies to blocking as well as non-blocking communication

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*

Hands-on:
NPB-MZ-MPI / BT

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 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.
```

- Scalasca measurement collection & analysis nexus

Automatic measurement configuration

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

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
   -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
```

- Scalasca analysis report explorer (Cube)

Recap: Local installation (Archer2)

- Setup access to Scalasca and associated tools
  - Source `module use /work/y23/shared/tutorial/modulefiles` to prepare the environment
  - Required for each shell session
  - Score-P and Scalasca installations are toolchain specific

```
% module restore PrgEnv-gnu
% module use /work/y23/shared/tutorial/modulefiles
% module load scalasca
```

  - Check `module avail scalasca` for alternate Score-P/Scalasca modules available

- Copy tutorial sources to your personal workspace

```
% cd $WORK
% tar zxvf /work/y23/shared/tutorial/NPB3.3-MZ-MPI.tar.gz
% cd NPB3.3-MZ-MPI
```

BT-MZ summary measurement collection...

```
% cd bin.scorep
% cp ../jobscript/archer2/scalasca.sbatch .
% cat scalasca.sbatch

# Scalasca nexus configuration for profiling
#NEXUS="scalasca -analyze"
# Scalasca nexus configuration for profiling
#NEXUS="scalasca -analyze -t"

# Score-P measurement configuration
export SCOREP_FILTERING_FILE=../config/scorep.filt
#export SCOREP_TOTAL_MEMORY=100M


# run the application
scalasca -analyze  srun ./bt-mz_C.8
```

```
% sbatch --reservation=tra031_192  scalasca.sbatch
```

- Change to directory with the Score-P instrumented executable and edit the job script

Hint:
scan = scalasca -analyze
-s = profile/summary (def)

- Submit the job

BT-MZ summary measurement

```
S=C=A=N: Scalasca 2.6 runtime summarization
S=C=A=N: ./scorep_bt-mz_C_8x6_sum experiment archive
S=C=A=N: Thu Jun  10 11:48:50 2021: Collect start
srun  ./bt-mz_C_8x6

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

 Number of zones:  8 x  8
 Iterations: 200    dt:   0.000100
 Number of active processes:     8

 [... More application output ...]

S=C=A=N: Thu Jun  10 11:49:02 2021: Collect done (status=0) 12s
S=C=A=N: ./scorep_bt-mz_C_8x6_sum complete.
```

- Run the application using the Scalasca measurement collection & analysis nexus prefixed to launch command
- Creates experiment directory:
  scorep_bt-mz_C_8x6_sum

BT-MZ summary analysis report examination

- Score summary analysis report

```
% square -s  scorep_bt-mz_C_8x6_sum
INFO: Post-processing runtime summarization report (profile.cubex)...
INFO: Score report written to ./scorep_bt-mz_C_8x6_sum/scorep.score
```

- Post-processing and interactive exploration with Cube

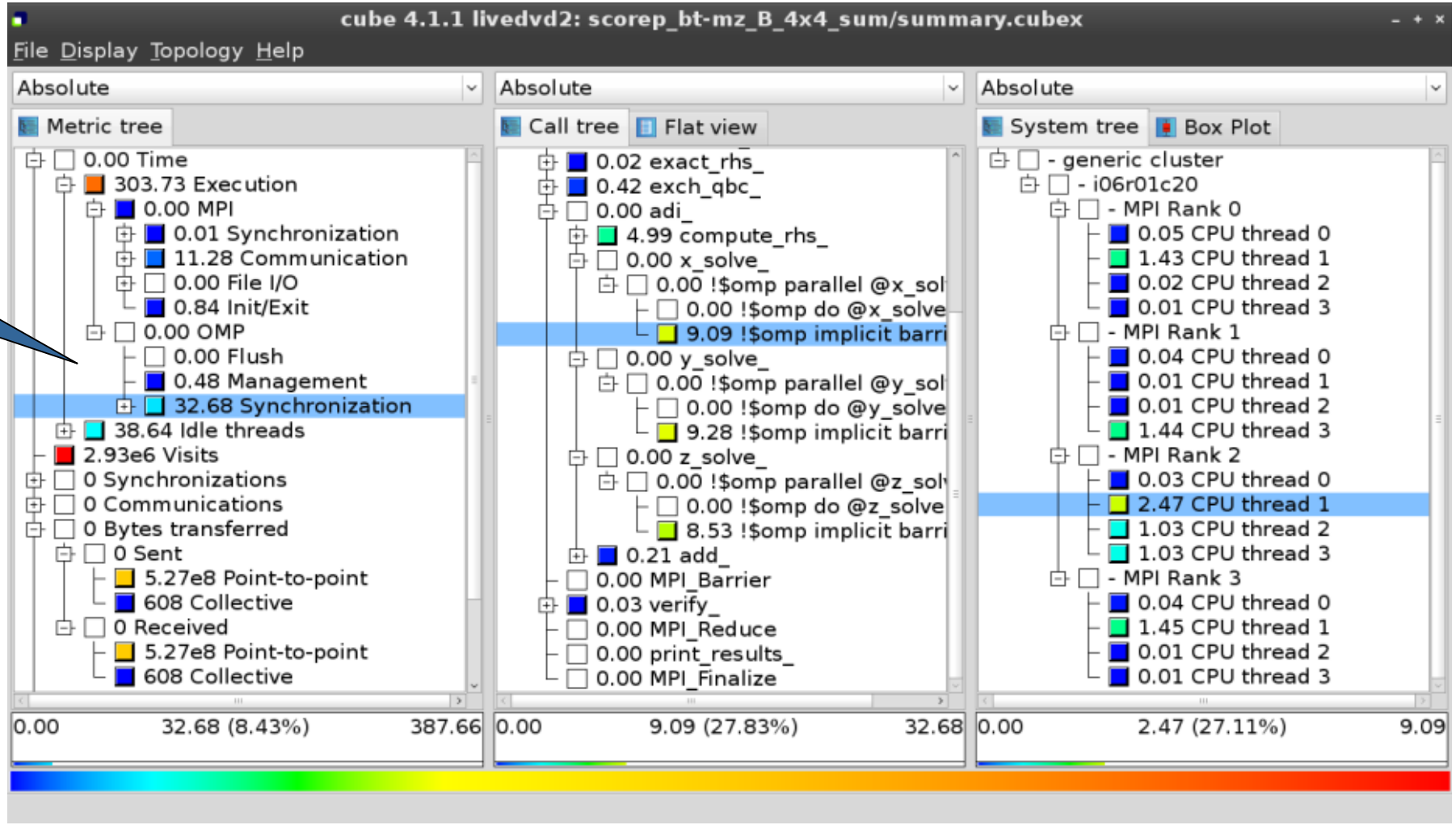
```
% square  scorep_bt-mz_C_8x6_sum
INFO: Displaying ./scorep_bt-mz_C_8x6_sum/summary.cubex...
                    [GUI showing summary analysis report]
```

Hint:
Copy ‘summary.cubex’ to local system (laptop) using ‘scp’ to improve responsiveness of GUI

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

Post-processed summary analysis report

Split base metrics into more specific metrics



Performance analysis steps

- 0.0 Reference preparation for validation

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

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

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

BT-MZ trace measurement collection...

```
% cd bin.scorep
% cp ../jobscript/archer2/scalasca.sbatch .
% vim scalasca.sbatch

# Scalasca nexus configuration for profiling
#NEXUS="scalasca -analyze"
# Scalasca nexus configuration for profiling
#NEXUS="scalasca -analyze -t"

# Score-P measurement configuration
export SCOREP_FILTERING_FILE=../config/scorep.filt
export SCOREP_TOTAL_MEMORY=100M

# run the application
scalasca -analyze -t  srun ./bt-mz_C.8
```

```
% sbatch -reservation=ta031_192 scalasca.sbatch
```

- Change to directory with the Score-P instrumented executable and edit the job script
- Add "-t" to the `scan` command
- Submit the job

BT-MZ trace measurement … collection

```
S=C=A=N: Scalasca 2.6 trace collection and analysis
S=C=A=N: ./scorep_bt-mz_C_8x6_trace experiment archive
S=C=A=N: Thu Jun  10 12:05:30 2021: Collect start
srun  ./bt-mz_C.8

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

 Number of zones:  16 x  16
 Iterations: 200    dt:   0.000100
 Number of active processes:     8

 [... More application output ...]

S=C=A=N: Thu Jun  10 12:05:44 2021: Collect done (status=0) 14s
```

- Using new experiment directory

  scorep_bt-mz_C_8x6_trace

- Starts measurement with collection of trace files …

BT-MZ trace measurement … analysis

```
…
S=C=A=N: Thu Jun  10 12:05:44 2021: Analyze start
 srun  scout.hyb --time-correct \
>    ./scorep_bt-mz_C_8x6_trace/traces.otf2

SCOUT   (Scalasca 2.6)

Analyzing experiment archive ./scorep_bt-mz_C_8x6_trace/traces.otf2

Opening experiment archive ... done (0.002s).
Reading definition data    ... done (0.004s).
Reading event trace data   ... done (0.113s).
Preprocessing              ... done (0.179s).
Timestamp correction       ... done (0.431s).
Analyzing trace data       ... done (5.174s).
Writing analysis report    ... done (0.175s).

Max. memory usage         : 422.312MB

        # passes         : 1
        # violated       : 0

Total processing time     : 6.140s
S=C=A=N: Thu Jun  10 12:05:51 2021: Analyze done (status=0) 7s
```

- Continues with automatic (parallel) analysis of trace files

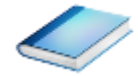
BT-MZ trace analysis report exploration

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

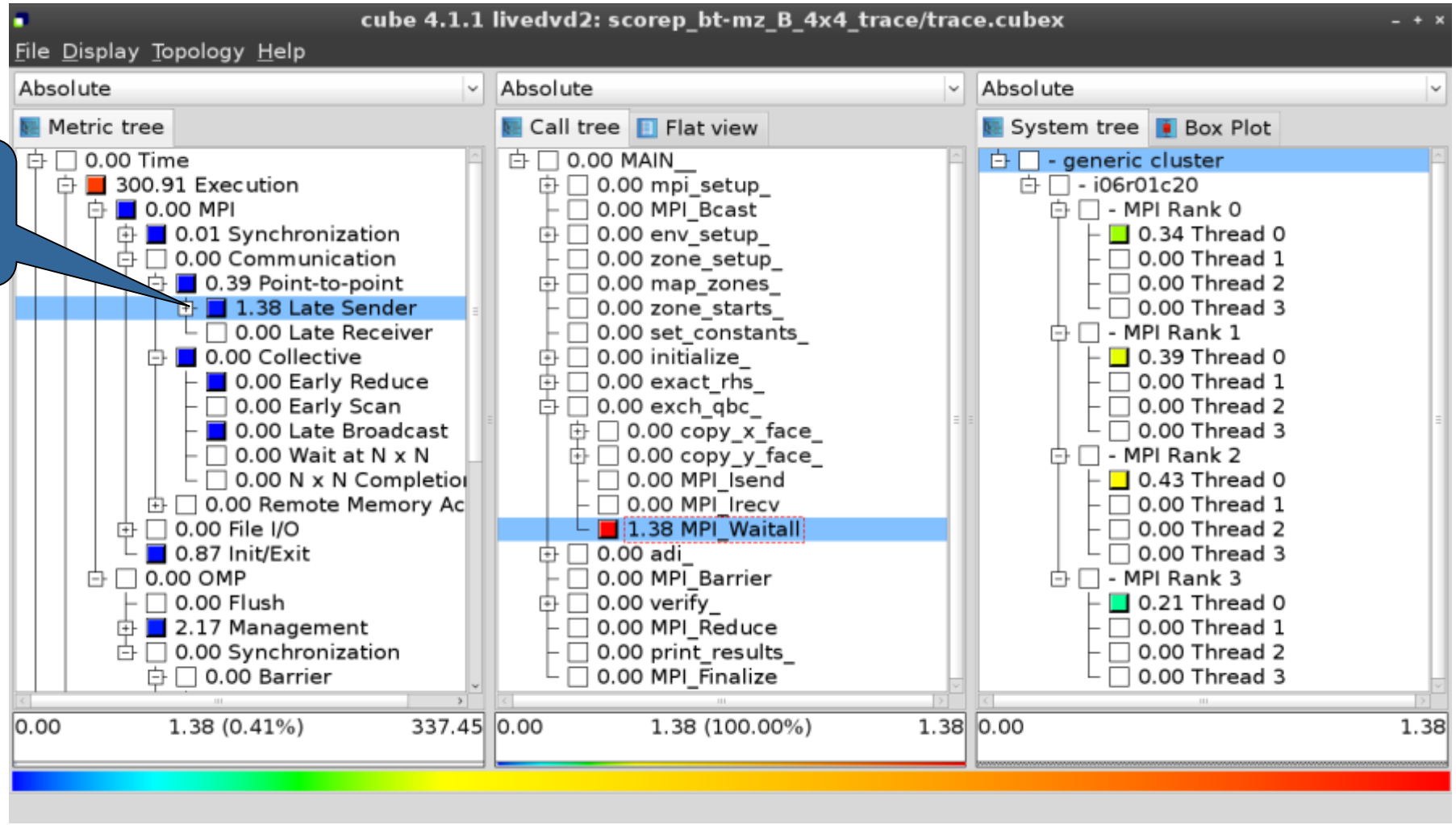
```
% square  scorep_bt-mz_C_8x6_trace
INFO: Post-processing runtime summarization report (profile.cubex)...
INFO: Post-processing trace analysis report (scout.cubex)...
INFO: Displaying ./scorep_bt-mz_C_8x6_trace/trace.cubex...

                [GUI showing trace analysis report]
```

Hint:
Run ‘square -s’ first and then copy ‘trace.cubex’ to local system (laptop) using ‘scp’ to improve responsiveness of GUI

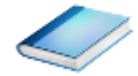
Post-processed trace analysis report



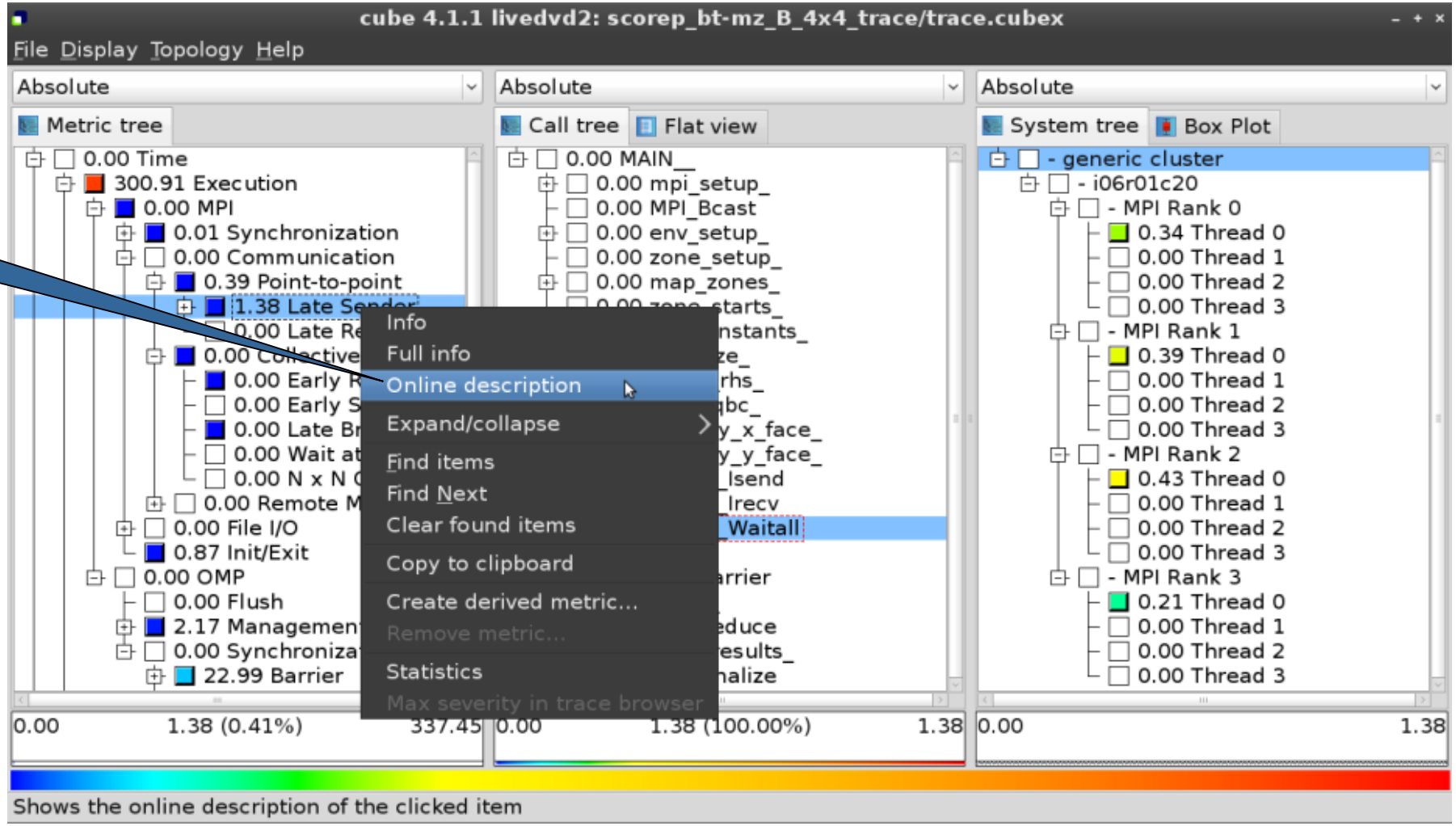
Additional trace-based metrics in metric hierarchy



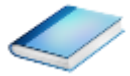
Online metric description



Access online metric description via context menu



Online metric description

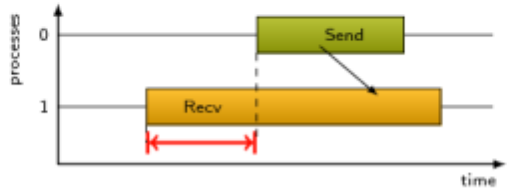


Performance properties

Late Sender Time

Description:

Refers to the time lost waiting caused by a blocking receive operation (e.g., `MPI_Recv` or `MPI_Wait`) that is posted earlier than the corresponding send operation.



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

Unit:

Seconds

Diagnosis:

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

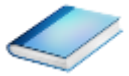
Parent:

MPI Point-to-point Communication Time

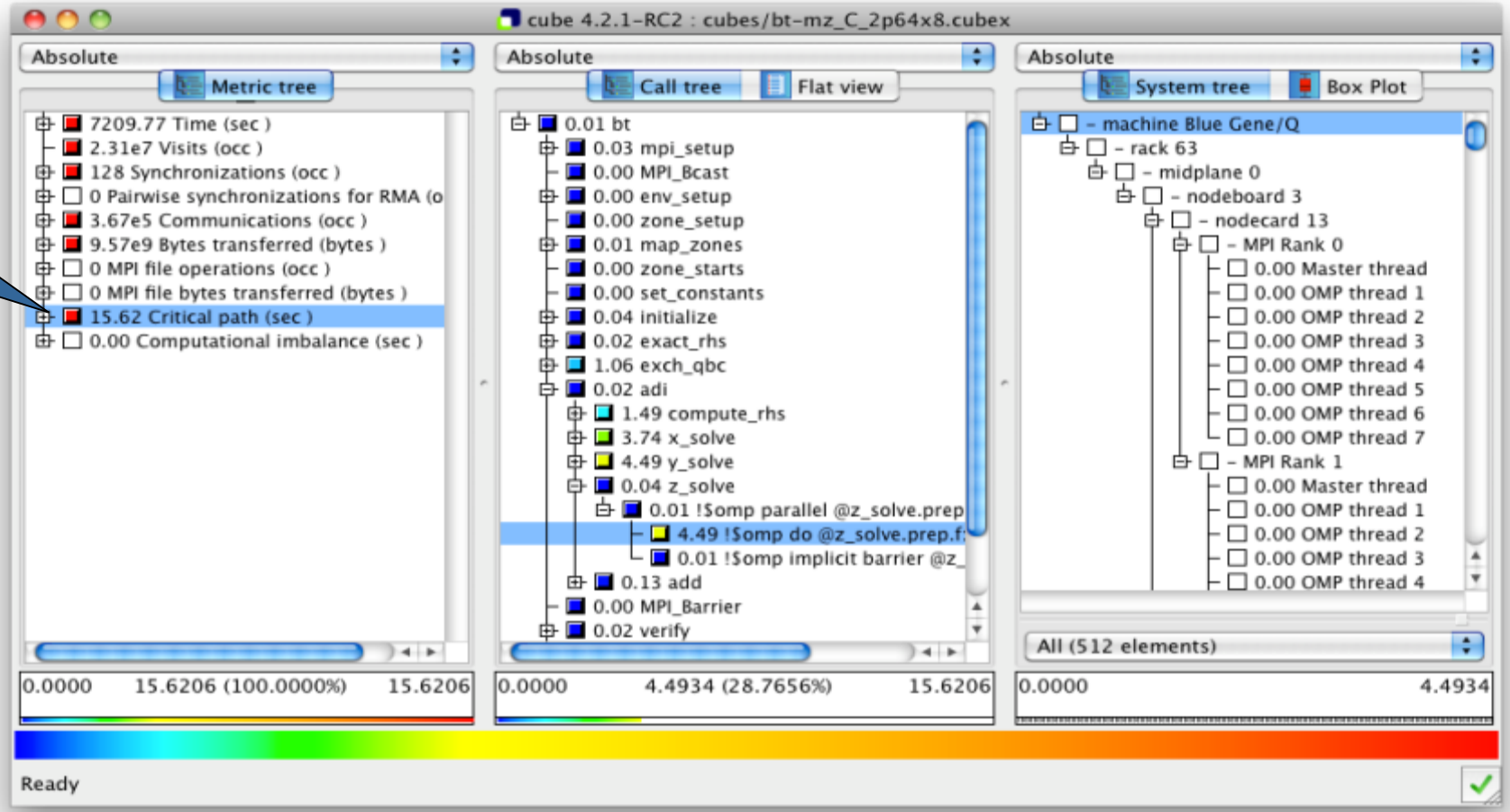
Children:

Close

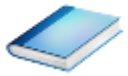
Critical-path analysis



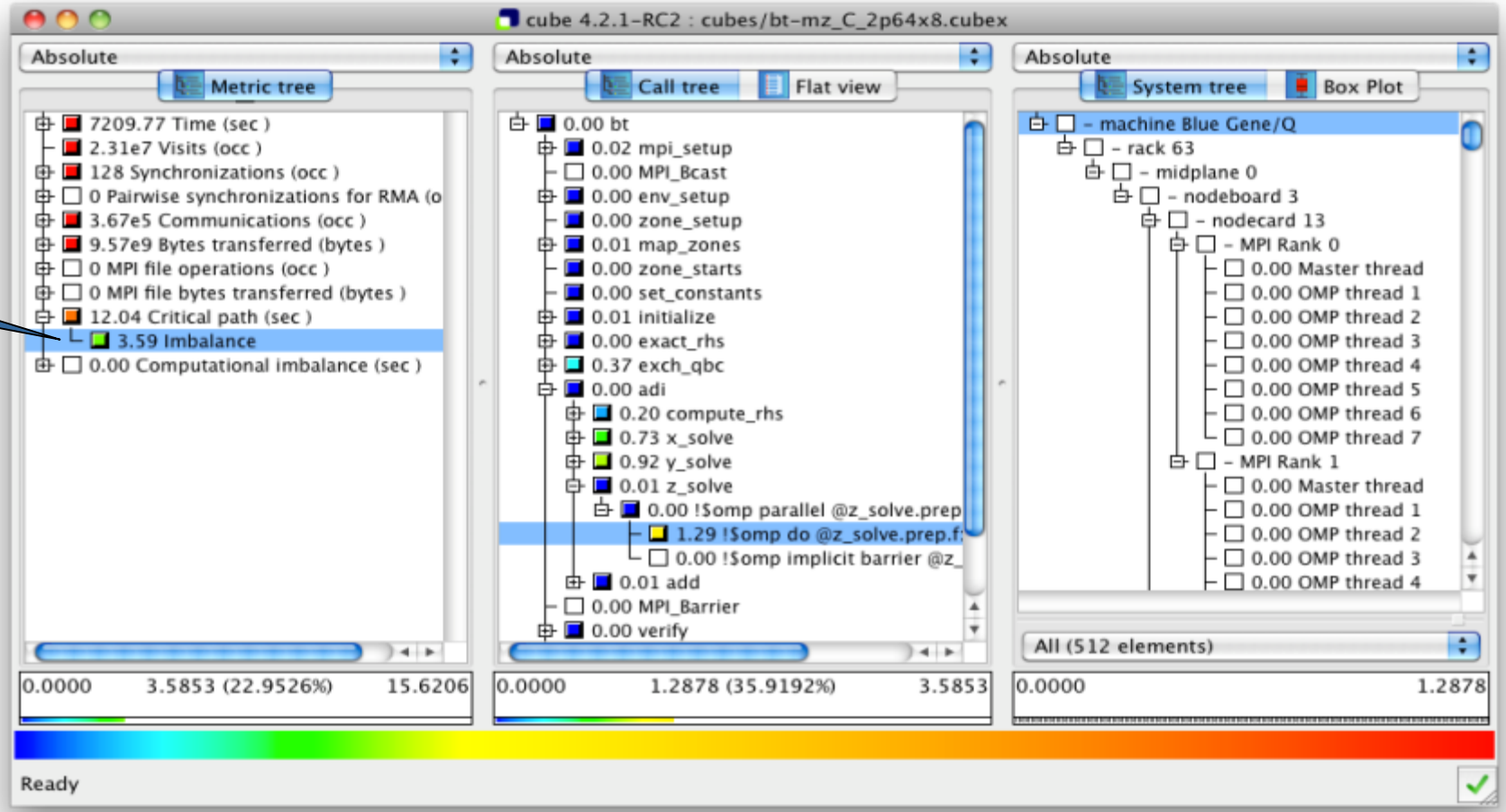
Critical-path profile shows wall-clock time impact



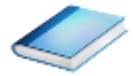
Critical-path analysis



Critical-path imbalance highlights inefficient parallelism

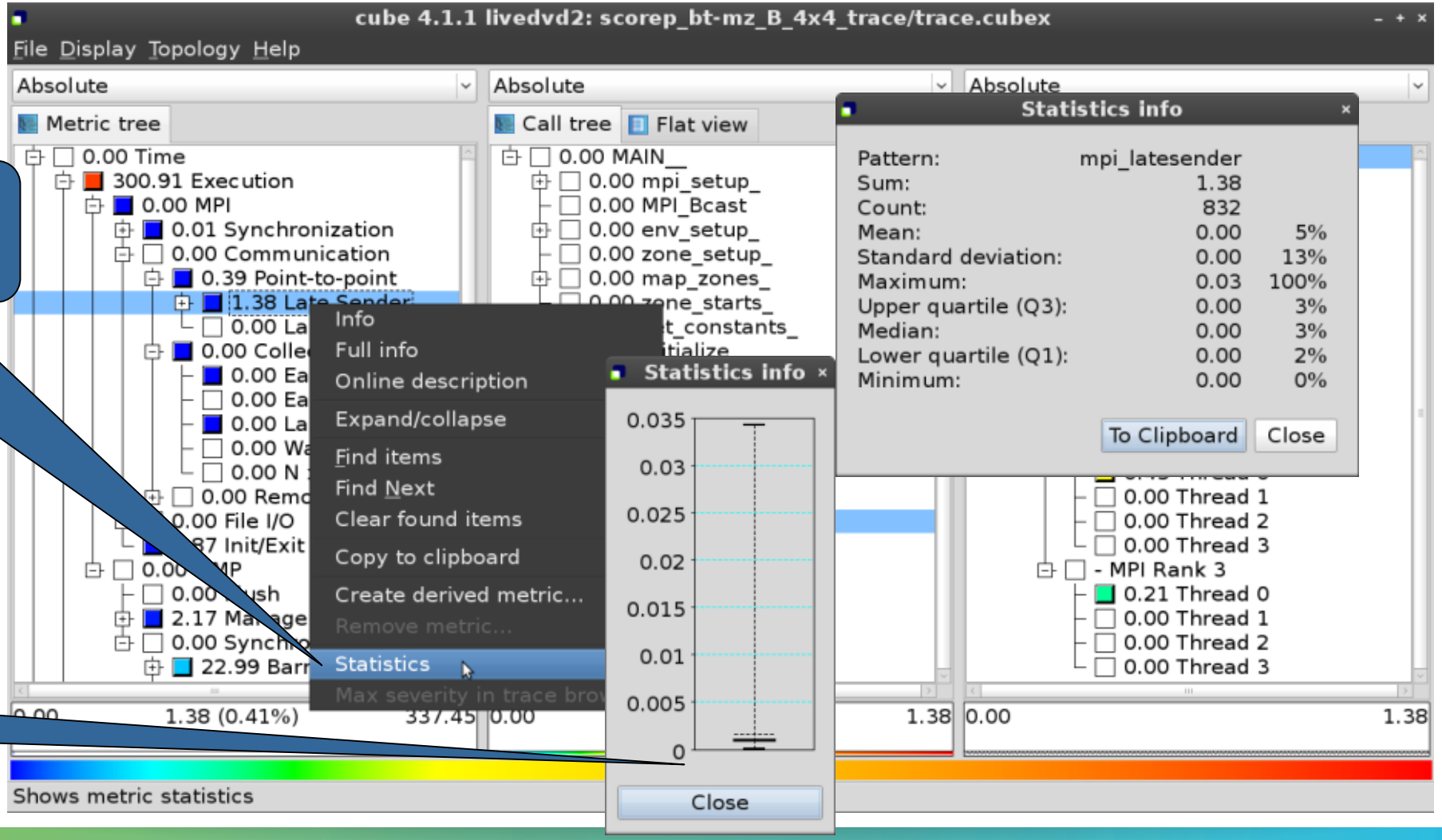


Pattern instance statistics



Access pattern instance statistics via context menu

Click to get statistics details



Demo:
TeaLeaf case study

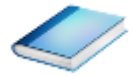
Case study: TeaLeaf

- 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: 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

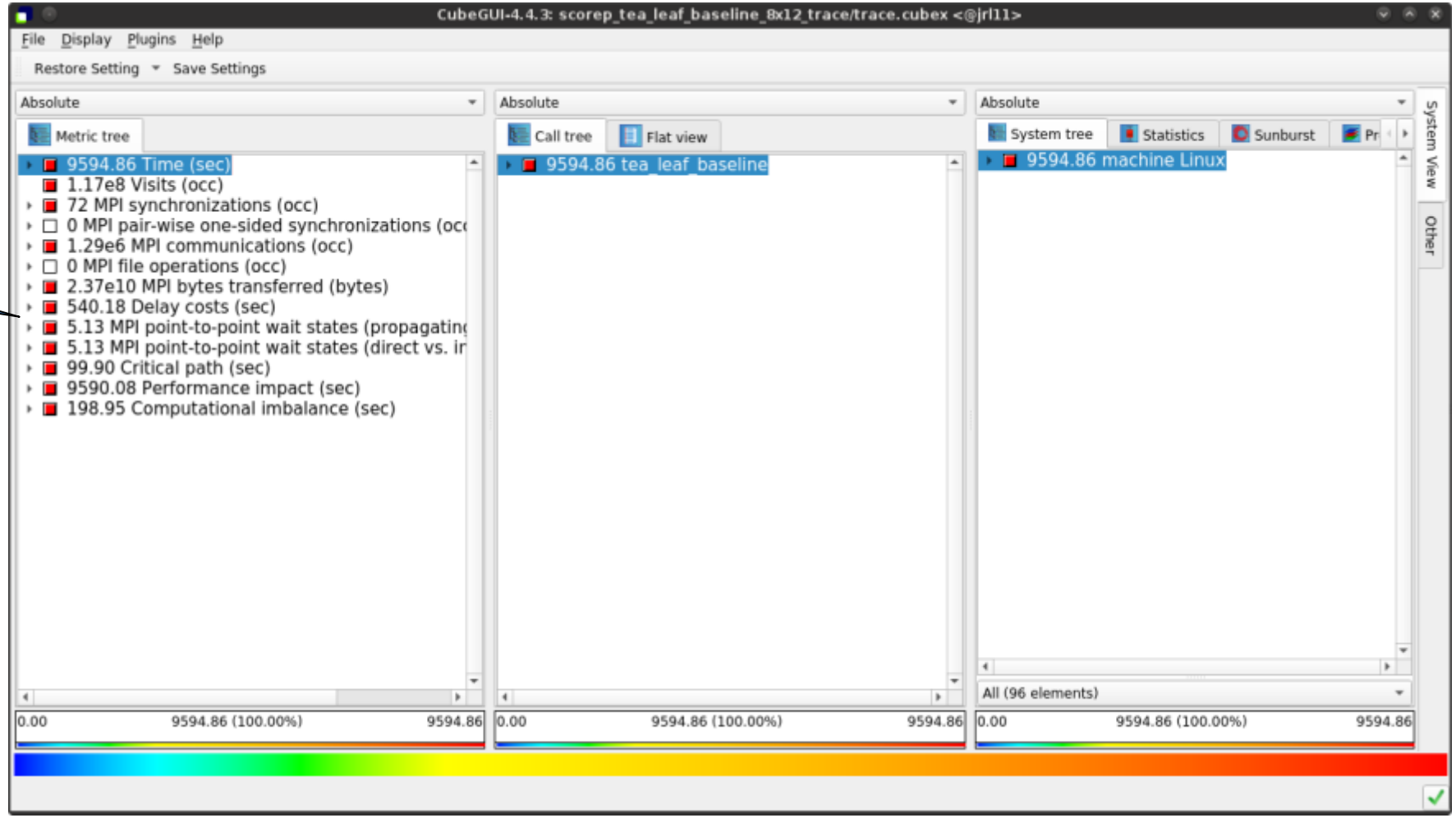
```
% cp /home/vihps/public/case_studies/↵
                              scorep_tea_leaf_baseline_8x12_trace .
% cube scorep_tea_leaf_baseline_8x12_trace/trace.cubex
                              [GUI showing post-processed trace analysis report]
```

Hint:
Copy 'trace.cubex' to local system (laptop) using 'scp' to improve responsiveness of GUI

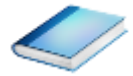
Scalasca analysis report exploration (opening view)



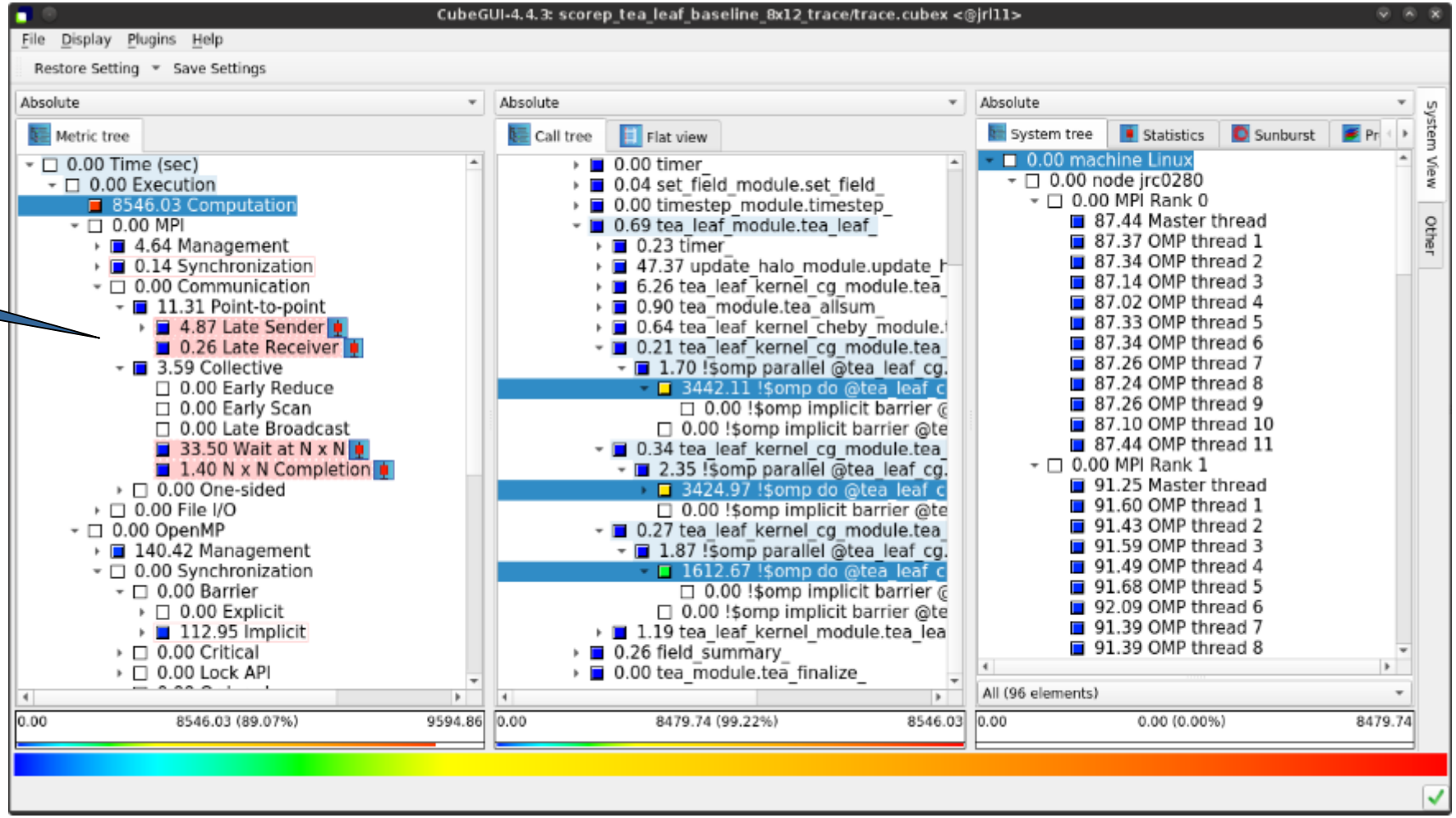
Additional top-level metrics produced by the trace analysis…



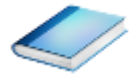
Scalasca wait-state metrics



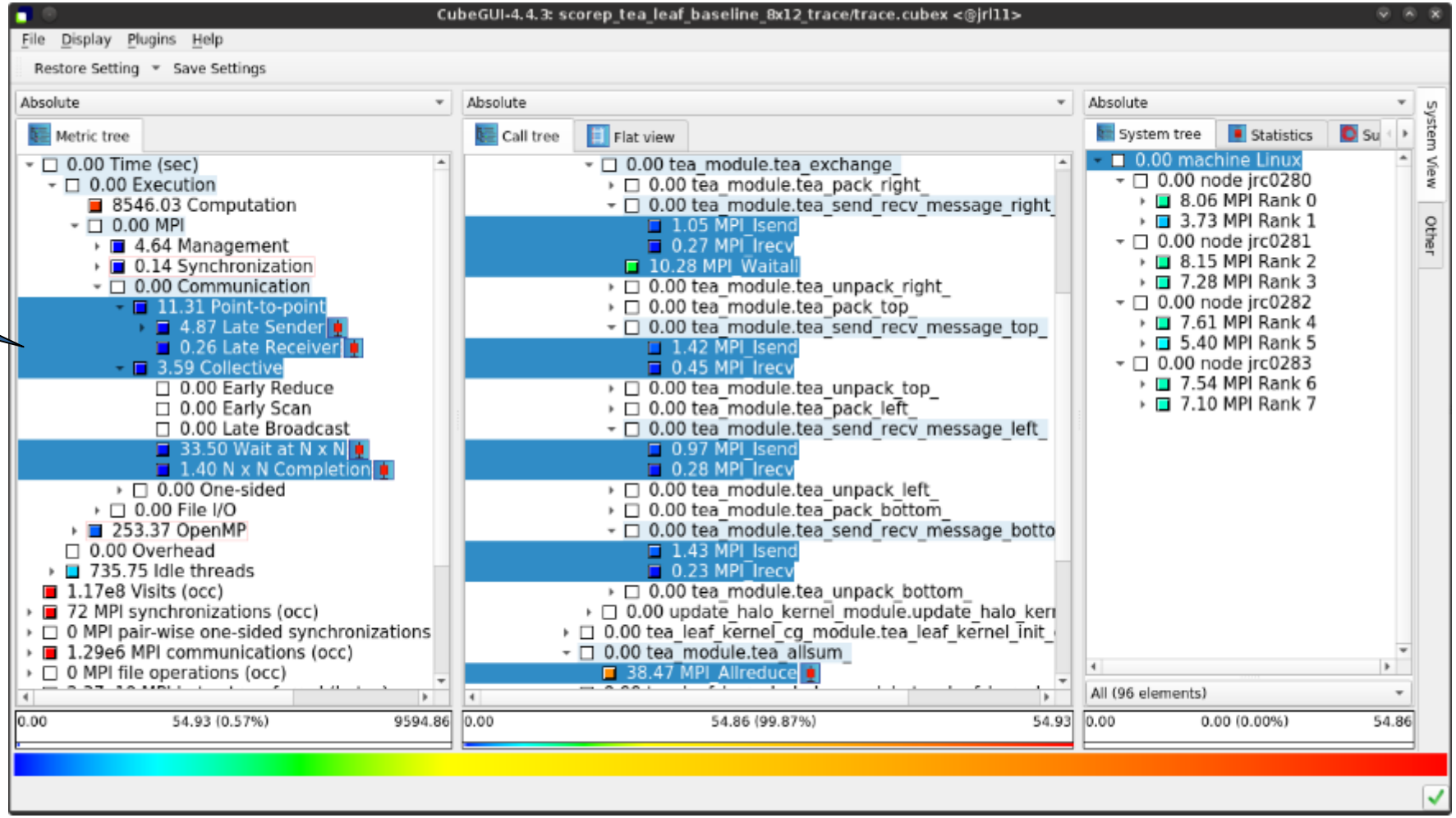
…plus additional wait-state metrics as part of the “Time” hierarchy



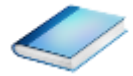
TeaLeaf Scalasca report analysis (I)



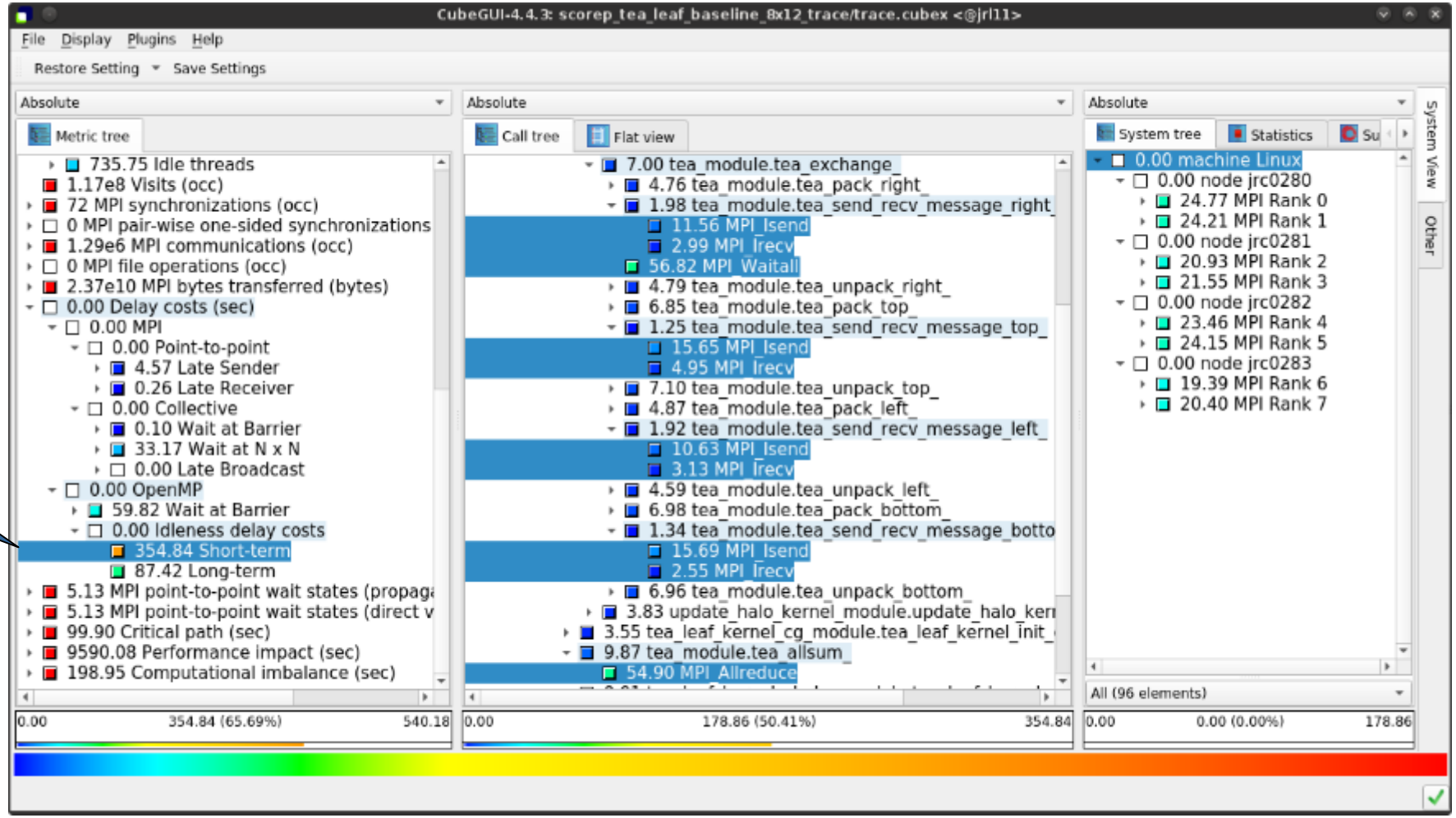
While MPI communication time and wait states are small (~0.6% of the total execution time)…



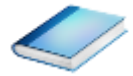
TeaLeaf Scalasca report analysis (II)



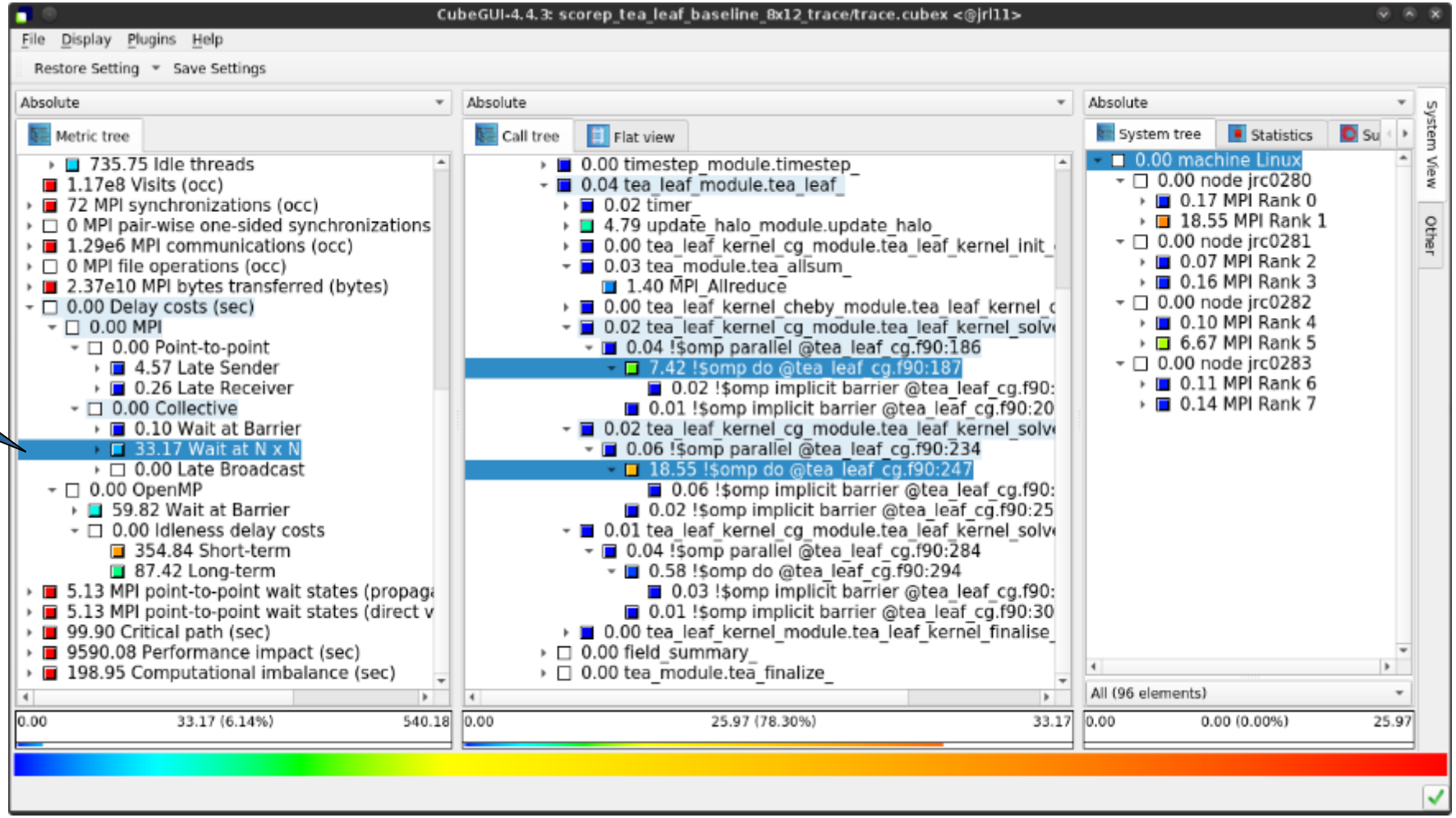
…they directly cause a significant amount of the OpenMP thread idleness



TeaLeaf Scalasca report analysis (III)

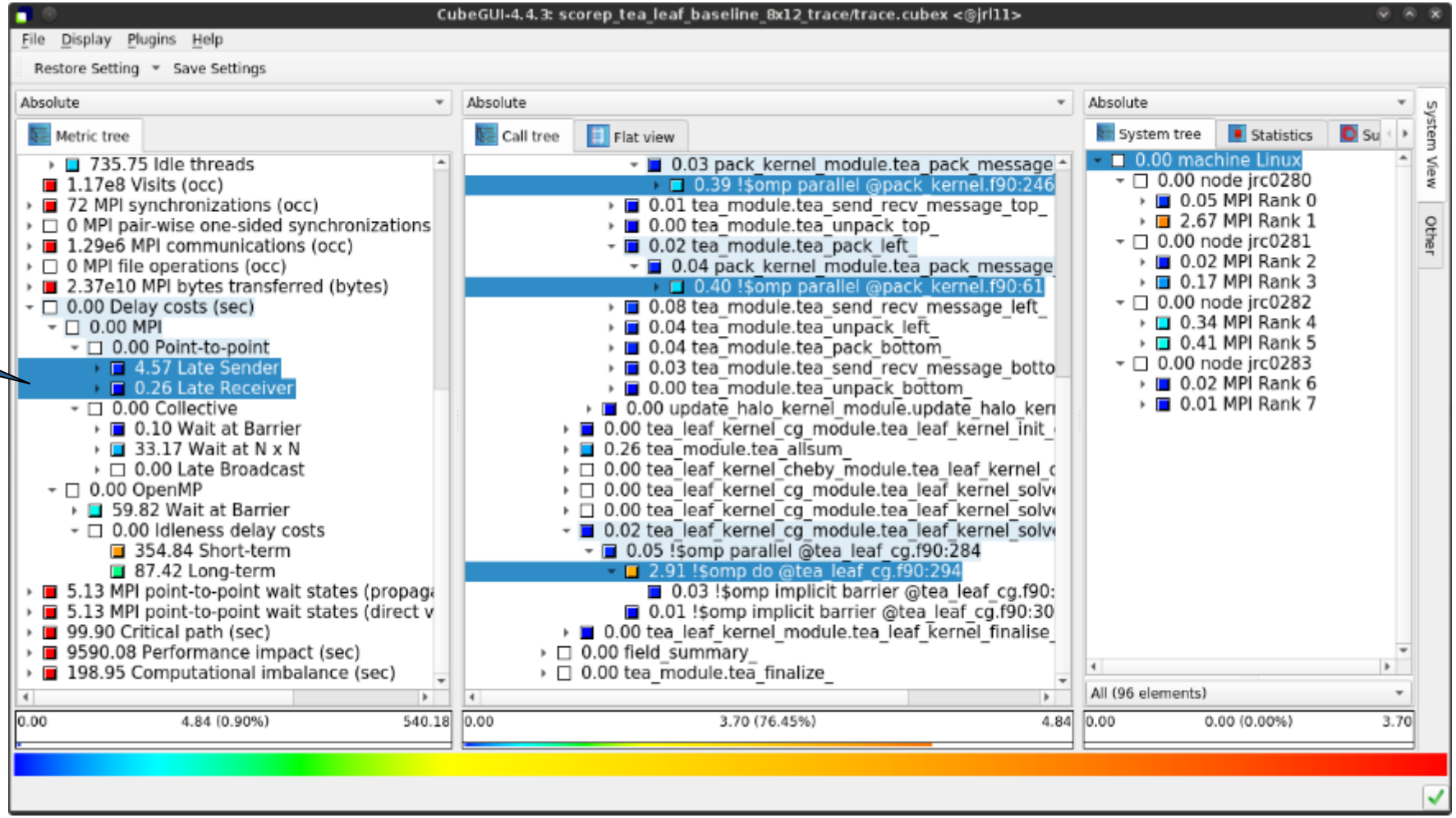


The “Wait at NxN” collective wait states are mostly caused by the first 2 OpenMP `do` loops of the solver (on ranks 5 & 1, resp.)…

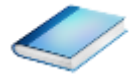


TeaLeaf Scalasca report analysis (IV)

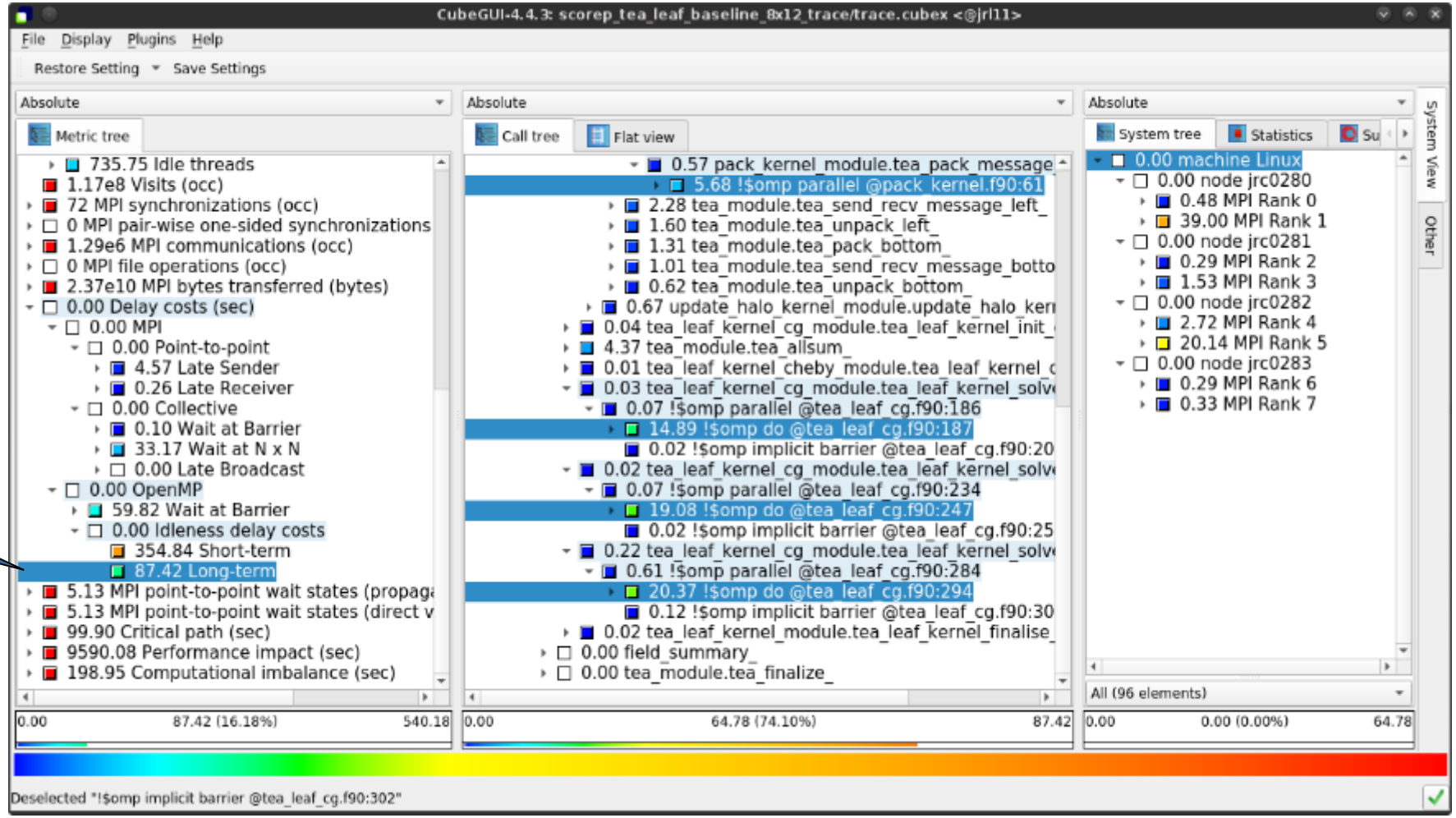
…while the MPI point-to-point wait states are caused by the 3rd solver `do` loop (on rank 1) and two loops in the halo exchange



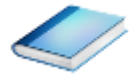
TeaLeaf Scalasca report analysis (V)



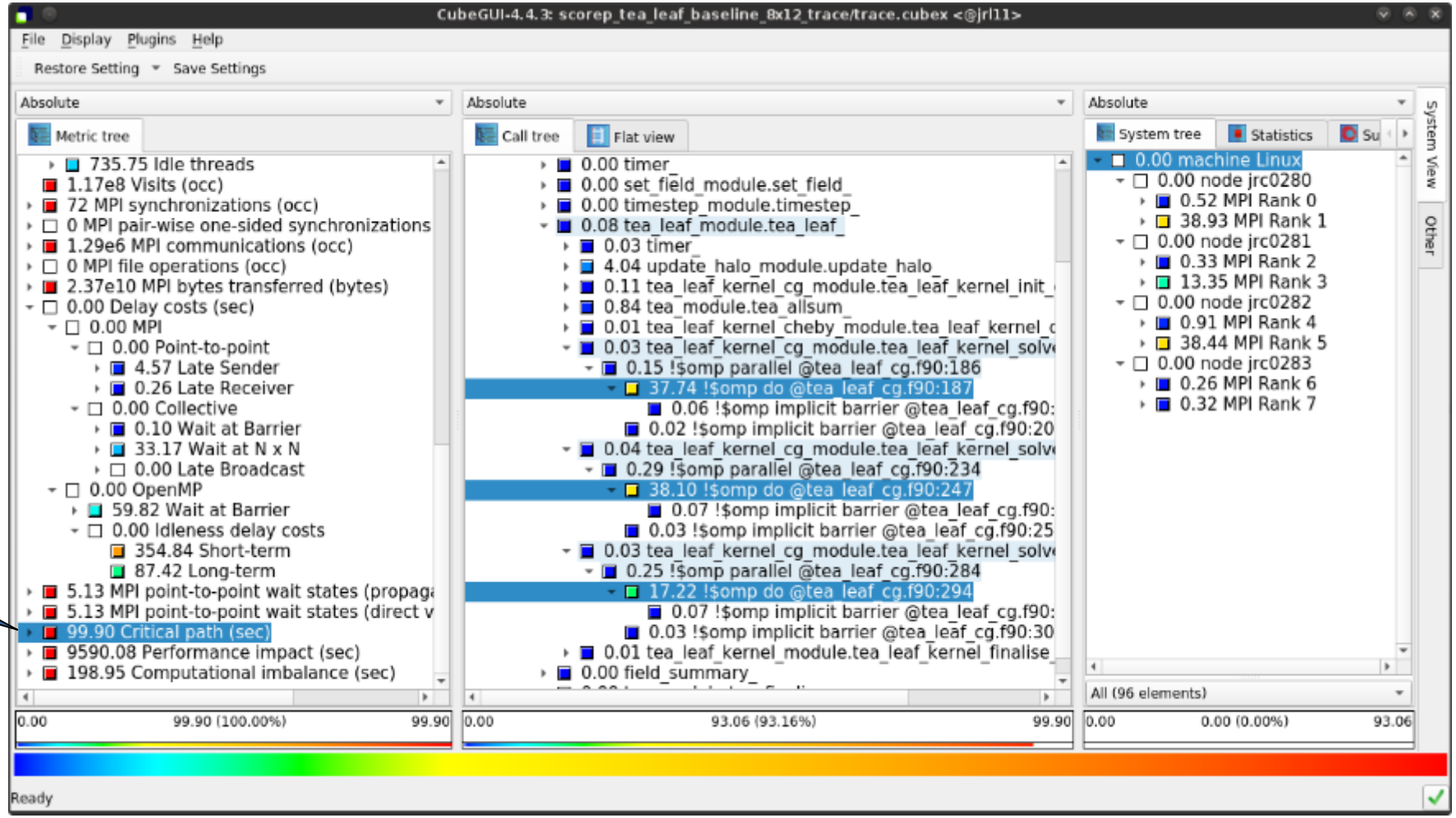
Various OpenMP `do` loops (incl. the solver loops) also cause OpenMP thread idleness on other ranks via propagation



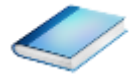
TeaLeaf Scalasca report analysis (VI)



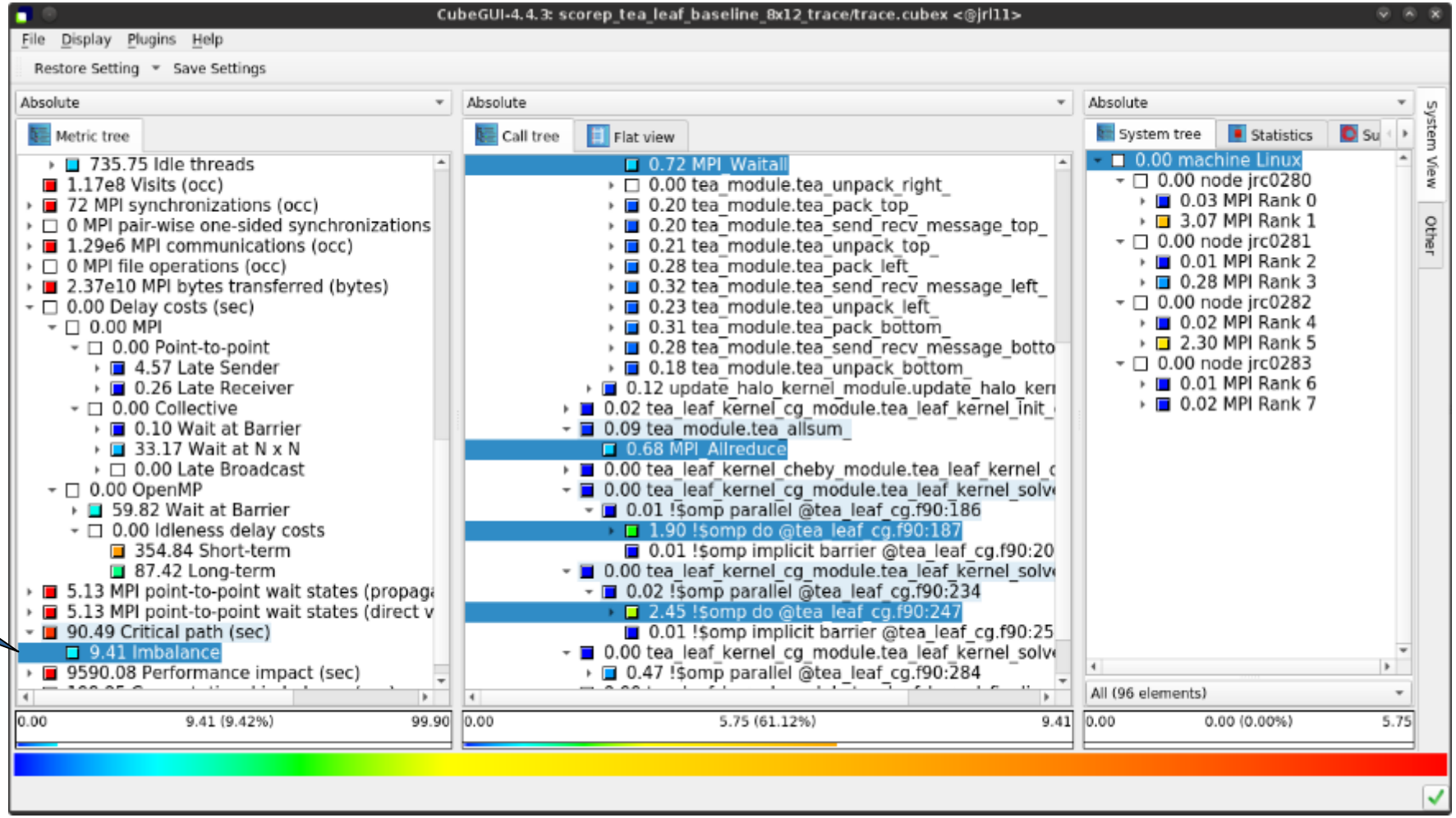
The Critical Path also highlights the three solver loops…



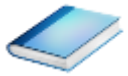
TeaLeaf Scalasca report analysis (VII)



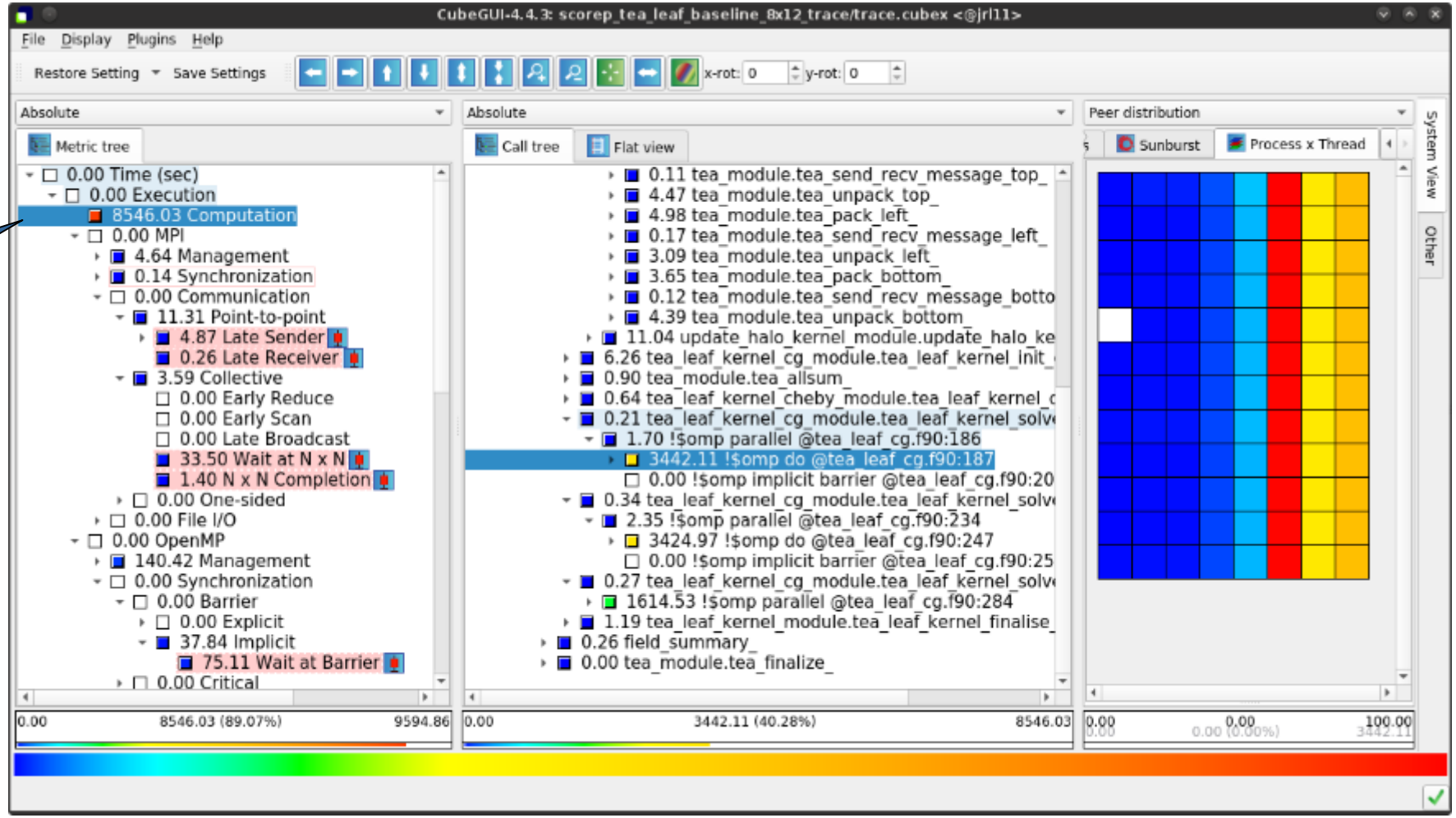
…with imbalance (time on critical path above average) mostly in the first two loops and MPI communication



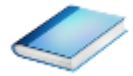
TeaLeaf Scalasca report analysis (VIII)



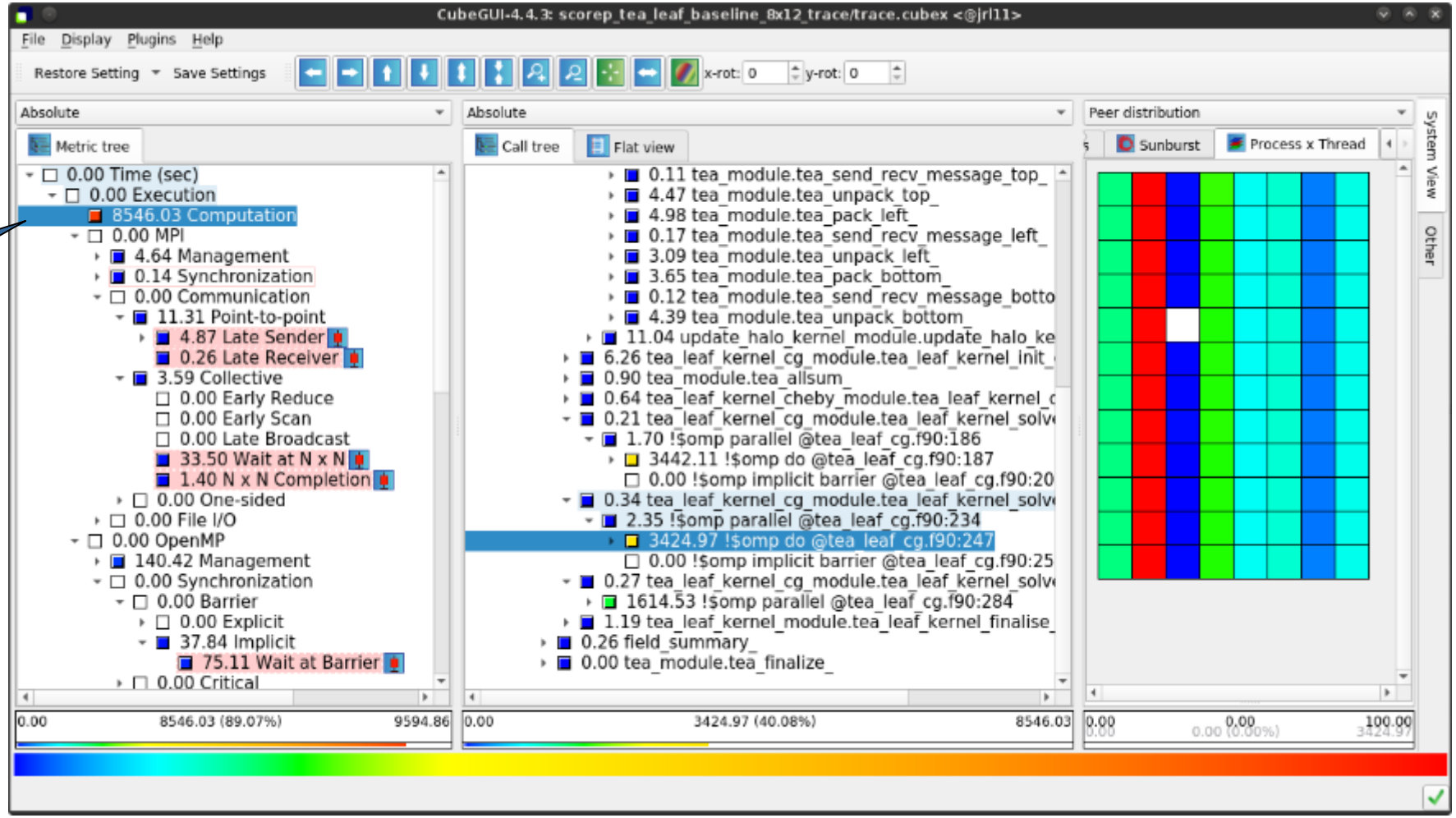
Computation time of 1st…



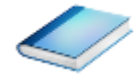
TeaLeaf Scalasca report analysis (IX)



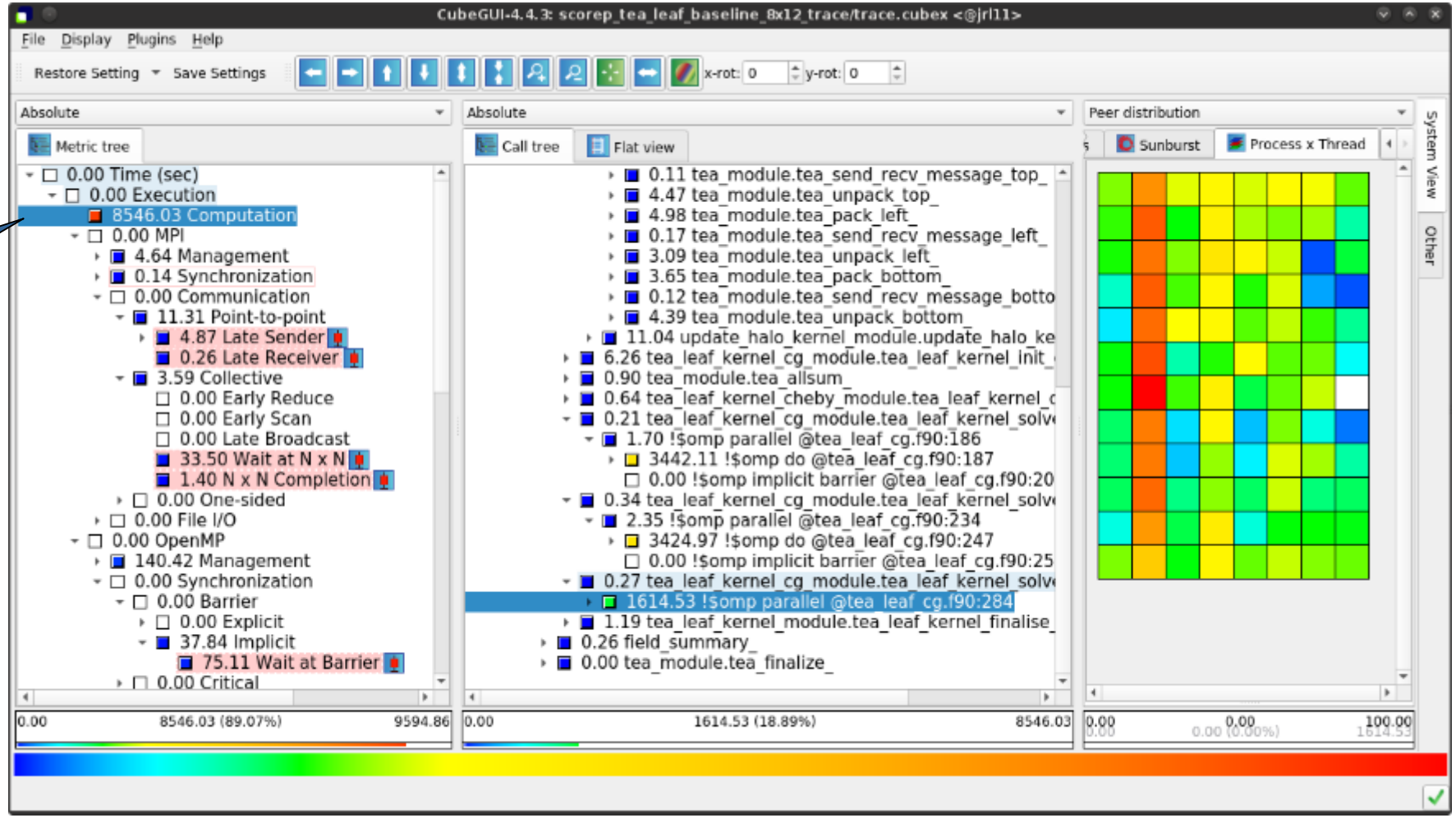
…and 2nd `do` loop mostly balanced within each rank, but vary considerably across ranks…



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%**

Scalasca Trace Tools: Further information

- Collection of trace-based performance tools
  - Specifically designed for large-scale systems
  - Features an automatic trace analyzer providing wait-state, critical-path, and delay analysis
  - Supports MPI, OpenMP, POSIX threads, and hybrid MPI+OpenMP/Pthreads
- Available under 3-clause BSD open-source license

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

trace tools scalasca

Exercises
(if you don't have your own code)

Warm-up

- Build the BT-MZ example code for class (i.e., problem size) "D"
  - Perform a baseline measurement w/o instrumentation (should run in ~190s)
  - Re-build the executable with Score-P instrumentation
- Repeat the hands-on exercise with the new executable
  - Perform a summary measurement
  - Score the summary measurement result
  - Adjust the measurement configuration appropriately
  - Perform a trace measurement and analysis

Trace analysis report examination

- What is the poportion of computation time vs. parallelization overheads?
- Which code regions are mostly responsible for the overall execution time?
- Are there any load balancing issues?
- If so, in which routines?
- What are the most significant wait states/parallelization overheads?
- What are their root causes?

Optimization

- What are possible optimizations?
  - Hint: Take a look at the TeaLeaf case study
- Modify the source code to apply those and re-do the measurement
  - Don’t worry – it’s straightforward even if you don’t know the code ;-)
  - Remember: One step at a time!
- How did the performance change?
  - The `cube_diff` tool or the “Cube Diff” plugin of the GUI (see the File → Context-free plugin menu) may come in handy here