Introduction to Parallel Performance Engineering with Score-P

Marc-André Hermanns
Jülich Supercomputing Centre

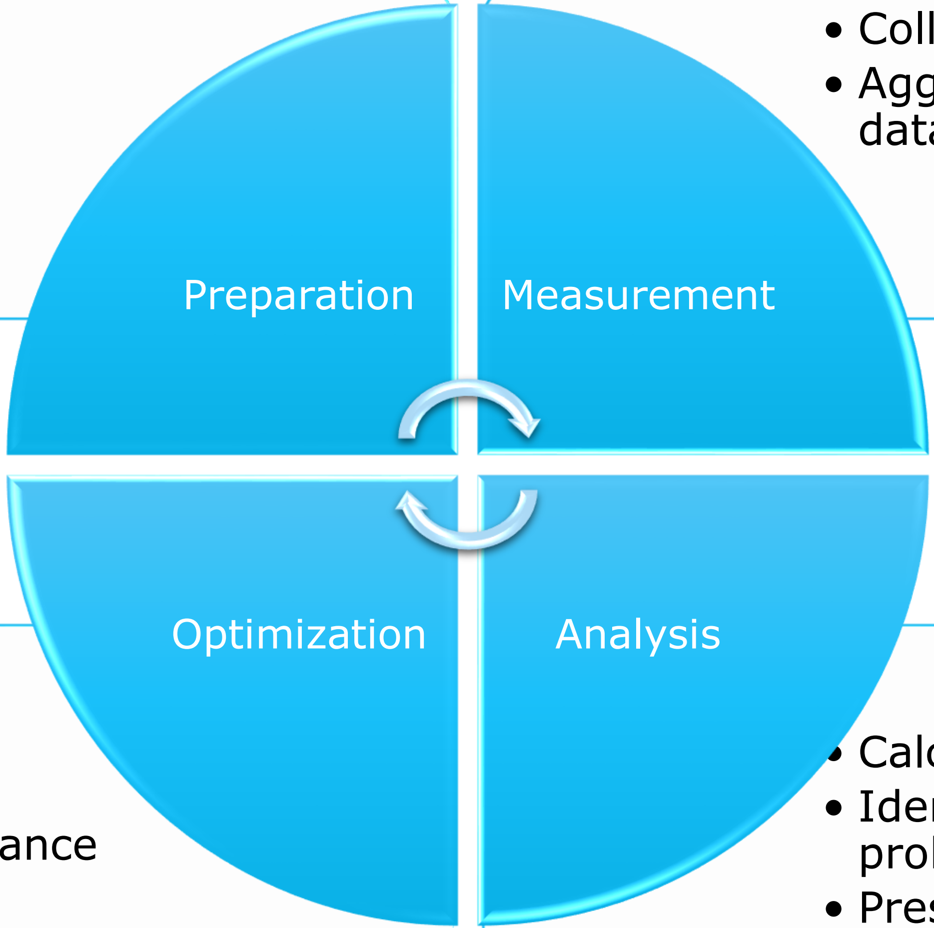
(with content used with permission from tutorials by Bernd Mohr/JSC and Luiz DeRose/Cray)

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

Performance engineering workflow



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

Measurement
- Collection of performance data
- Aggregation of performance data

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

Optimization
- 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!*

Metrics of performance

- 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? …

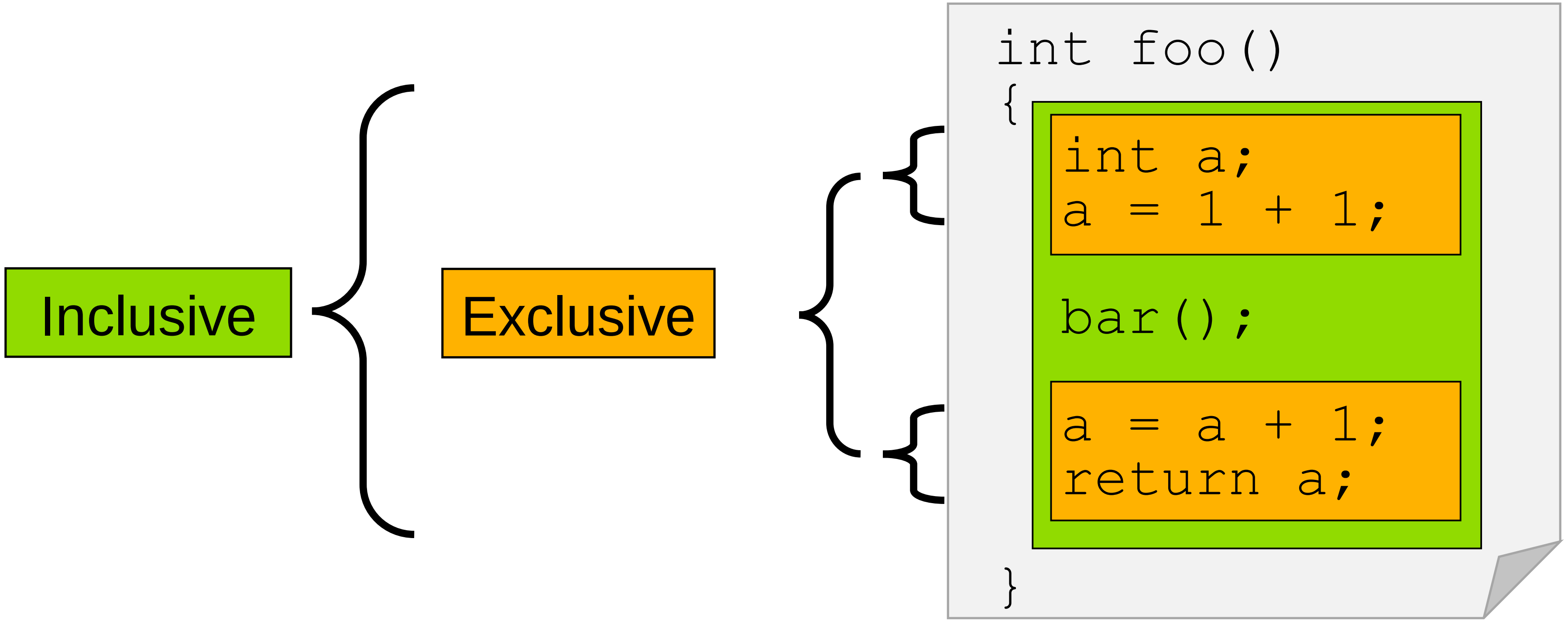
Execution time

- 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

Inclusive vs. Exclusive values

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

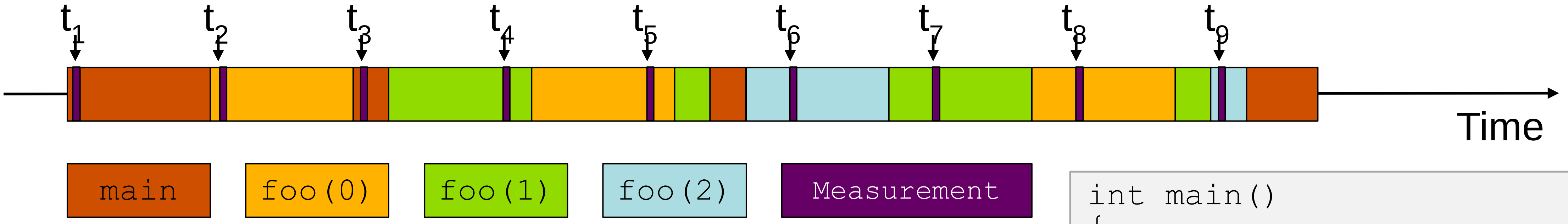


Classification of measurement techniques

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

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

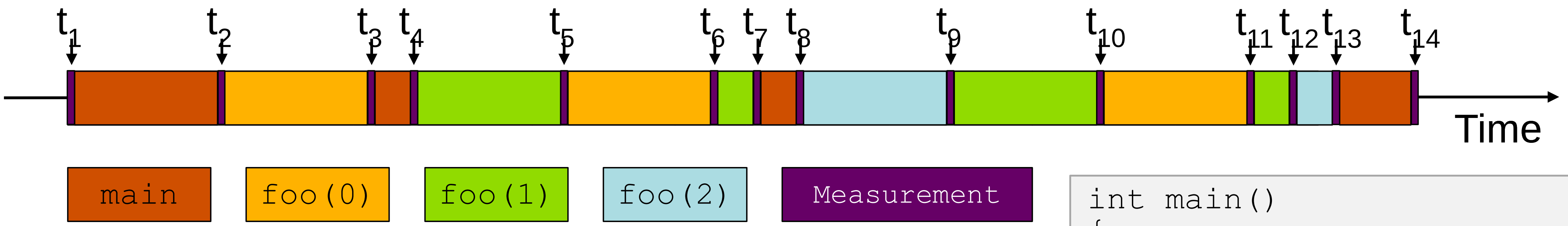
Sampling



- 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");
}
```

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**

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*

Tracing Pros & Cons

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

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

Score-P functionality

- Provide typical functionality for HPC performance tools
- Support all fundamental concepts of partner's tools

- Instrumentation (various methods)
- Flexible measurement without re-compilation:
  - Basic and advanced profile generation
  - Event trace recording
  - Online access to profiling data

- MPI/SHMEM, OpenMP/Pthreads, and hybrid parallelism (and serial)
- Enhanced functionality (CUDA, OpenCL, OpenACC, highly scalable I/O)

NPB-MZ-MPI / BT instrumentation

```
#---------------------------------------------------------------------
# The Fortran compiler used for MPI programs
#---------------------------------------------------------------------
MPIF77 = mpif77

# Alternative variants to perform instrumentation
...
MPIF77 = scorep --user  mpif77
...
#MPIF77 = $(PREP) mpif77

# This links MPI Fortran programs; usually the same as ${MPIF7…
FLINK   = $(MPIF77)
...
```

- Edit config/make.def to adjust build configuration
  - Modify specification of compiler/linker: MPIF77

Uncomment the compiler wrapper specification

NPB-MZ-MPI / BT instrumented build

```
% make clean

% make bt-mz CLASS=C NPROCS=8
cd BT-MZ; make CLASS=C NPROCS=8 VERSION=
make: Entering directory 'BT-MZ'
cd ../sys; cc -o setparams setparams.c -lm
../sys/setparams bt-mz 8 B
scorep --user mpif77 -c  -O3 -fopenmp bt.f
 [...]
cd ../common; scorep --user mpif77 -c  -O3 -qopenmp timers.f
scorep --user mpif77 -O3 -qopenmp -o ../bin.scorep/bt-mz_C.8 \
bt.o initialize.o exact_solution.o exact_rhs.o set_constants.o \
adi.o rhs.o zone_setup.o x_solve.o y_solve.o exch_qbc.o \
solve_subs.o z_solve.o add.o error.o verify.o mpi_setup.o \
../common/print_results.o ../common/timers.o
Built executable ../bin.scorep/bt-mz_C.8
make: Leaving directory 'BT-MZ'
```

- Return to root directory and clean-up
- Re-build executable using Score-P compiler wrapper

Summary measurement collection

```
% cd bin.scorep
% cp ../jobscript/claix/scorep.lsf .
% vi scorep.lsf

[...]
export SCOREP_EXPERIMENT_DIRECTORY=scorep_bt-mz_sum
[...]

% bsub < scorep.lsf
```

- Change to the directory containing the new executable before running it with the desired configuration
- Check settings

Leave other lines commented out for the moment

- Submit job

Summary measurement collection

```
% less mzmpibt_scorep.<job_id>

 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

 Use the default load factors with threads
 Total number of threads:    48  (  6.0 threads/process)

 Calculated speedup =   47.97

 Time step    1

 [... More application output ...]
```

- Check the output of the application run

BT-MZ summary analysis report examination

```
% ls
bt-mz_C.8  mzmpibt-<job_id>.out   scorep_bt-mz_sum
% ls scorep_bt-mz_sum
profile.cubex  scorep.cfg




% module load cube/4.3.4-gnu
% cube scorep_bt-mz_sum/profile.cubex

       [CUBE GUI showing summary analysis report]
```

- Creates experiment directory including
  - A record of the measurement configuration (scorep.cfg)
  - The analysis report that was collated after measurement (profile.cubex)
- Interactive exploration with Cube

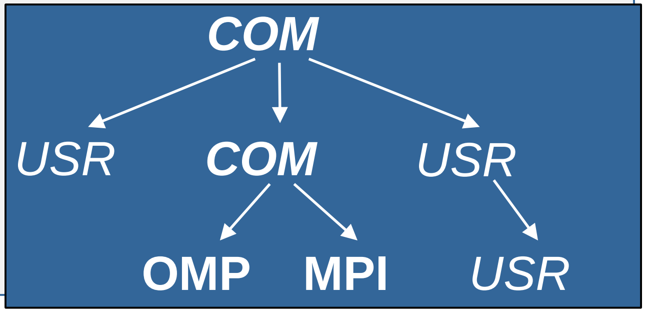
Congratulations!?

- If you made it this far, you successfully used Score-P to
  - instrument the application
  - analyze its execution with a summary measurement, and
  - examine it with one the interactive analysis report explorer GUIs
- ... revealing the call-path profile annotated with
  - the “Time” metric
  - Visit counts
  - MPI message statistics (bytes sent/received)
- ... but how ***good*** was the measurement?
  - The measured execution produced the desired valid result
  - however, the execution took rather longer than expected!
    - even when ignoring measurement start-up/completion, therefore
    - it was probably dilated by instrumentation/measurement overhead

BT-MZ summary analysis result scoring

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

Estimated aggregate size of event trace:                   160 GB
Estimated requirements for largest trace buffer (max_buf):  21 GB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):        21 GB
(warning: The memory requirements cannot be satisfied by Score-P to avoid
 intermediate flushes when tracing. Set SCOREP_TOTAL_MEMORY=4G to get the
 maximum supported memory or reduce requirements using USR regions filters.)

flt  type      max_buf[B]        visits time[s] time[%] time/visit[us]  region
      ALL 21,476,892,880 6,583,834,153 2537.82   100.0           0.39  ALL
      USR 21,431,996,118 6,574,793,529 1222.38    48.2           0.19  USR
      OMP     42,257,056     8,283,136 1270.45    50.1         153.38  OMP
      COM      2,351,570       723,560    2.99     0.1           4.13  COM
      MPI        288,136        33,928   42.00     1.7        1237.91  MPI
```

160 GB total memory
21 GB per rank!

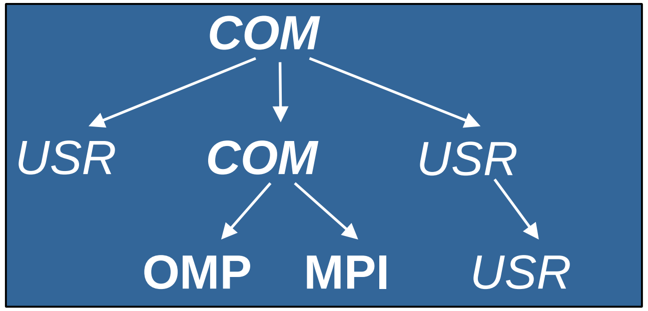


- Report scoring as textual output

- Region/callpath classification
  - **MPI** pure MPI functions
  - **OMP** pure OpenMP regions
  - **USR** user-level computation
  - **COM** “combined” USR+OpenMP/MPI
  - **ANY/ALL** aggregate of all region types

BT-MZ summary analysis report breakdown

```
% scorep-score -r scorep_bt-mz_sum/profile.cubex
  [...]
  [...]
flt  type     max_buf[B]         visits time[s] time[%] time/visit[us]  region
       ALL 21,476,892,880 6,583,834,153 2537.82   100.0           0.39  ALL
       USR 21,431,996,118 6,574,793,529 1222.38    48.2           0.19  USR
       OMP     42,257,056     8,283,136 1270.45    50.1         153.38  OMP
       COM      2,351,570       723,560    2.99     0.1           4.13  COM
       MPI        288,136        33,928   42.00     1.7        1237.91  MPI

       USR  6,883,222,086 2,110,313,472  513.22    20.2           0.24  binvcrhs_
       USR  6,883,222,086 2,110,313,472  381.01    15.0           0.18  matmul_sub_
       USR  6,883,222,086 2,110,313,472  284.34    11.2           0.13  matvec_sub_
       USR    293,617,584    87,475,200   20.34     0.8           0.23  lhsinit_
       USR    293,617,584    87,475,200   14.82     0.6           0.17  binvrhs_
       USR    224,028,792    68,892,672    8.64     0.3           0.13  exact_solution_
```



More than 20 GB just for these 6 regions

BT-MZ summary analysis score

- Summary measurement analysis score reveals
  - Total size of event trace would be ~160 GB
  - Maximum trace buffer size would be ~21 GB per rank
    - smaller buffer would require flushes to disk during measurement resulting in substantial perturbation
  - 99.9% of the trace requirements are for USR regions
    - purely computational routines never found on COM call-paths common to communication routines or OpenMP parallel regions
  - These USR regions contribute around 39% of total time
    - however, much of that is very likely to be measurement overhead for frequently-executed small routines
- Advisable to tune measurement configuration
  - Specify an adequate trace buffer size
  - Specify a filter file listing (USR) regions not to be measured

BT-MZ summary analysis report filtering

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

% scorep-score -f ../config/scorep.filt \
      scorep_bt-mz_sum/profile.cubex

Estimated aggregate size of event trace:                   343MB
Estimated requirements for largest trace buffer (max_buf): 43MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       49MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=215MB to avoid intermediate flushes
 or reduce requirements using USR regions filters.)
```

- Report scoring with prospective filter listing 6 USR regions

343 MB of memory in total,
43 MB per rank!

BT-MZ summary analysis report filtering

```
% scorep-score -r -f ../config/scorep.filt \
      scorep_bt-mz_sum/profile.cubex
flt     type     max_buf[B]        visits time[s] time[%] time/visit[us]  region
 -       ALL 21,476,892,880 6,583,834,153 2537.82   100.0           0.39  ALL
 -       USR 21,431,996,118 6,574,793,529 1222.38    48.2           0.19  USR
 -       OMP     42,257,056     8,283,136 1270.45    50.1         153.38  OMP
 -       COM      2,351,570       723,560    2.99     0.1           4.13  COM
 -       MPI        288,136        33,928   42.00     1.7        1237.91  MPI

 *       ALL     44,928,768     9,050,465 1315.44    51.8         145.35  ALL-FLT
 +       FLT 21,431,964,112 6,574,783,688 1222.37    48.2           0.19  FLT
 -       OMP     42,257,056     8,283,136 1270.45    50.1         153.38  OMP-FLT
 *       COM      2,351,570       723,560    2.99     0.1           4.13  COM-FLT
 -       MPI        288,136        33,928   42.00     1.7        1237.91  MPI-FLT
 *       USR         32,006         9,841    0.00     0.0           0.31  USR-FLT

 +       USR  6,883,222,086 2,110,313,472  513.22    20.2           0.24  binvcrhs_
 +       USR  6,883,222,086 2,110,313,472  381.01    15.0           0.18  matmul_sub_
 +       USR  6,883,222,086 2,110,313,472  284.34    11.2           0.13  matvec_sub_
 +       USR    293,617,584    87,475,200   20.34     0.8           0.23  lhsinit_
 +       USR    293,617,584    87,475,200   14.82     0.6           0.17  binvrhs_
 +       USR    224,028,792    68,892,672    8.64     0.3           0.13  exact_solution_
 -       OMP      3,357,504       308,736    0.09     0.0           0.31  !$omp parallel @exch_qbc.f:244
```

BT-MZ filtered summary measurement

```
% vi scorep.lsf
[…]
export SCOREP_EXPERIMENT_DIRECTORY=scorep_bt-mz_sum_filter
export SCOREP_FILTERING_FILE=../config/scorep.filt
[…]

% bsub < scorep.lsf
```

- Set new experiment directory and re-run measurement with new filter configuration
- Submit job

Analysis report examination with Cube

Marc-André Hermanns
Jülich Supercomputing Centre

Cube (Performance Report Browser & Tools)

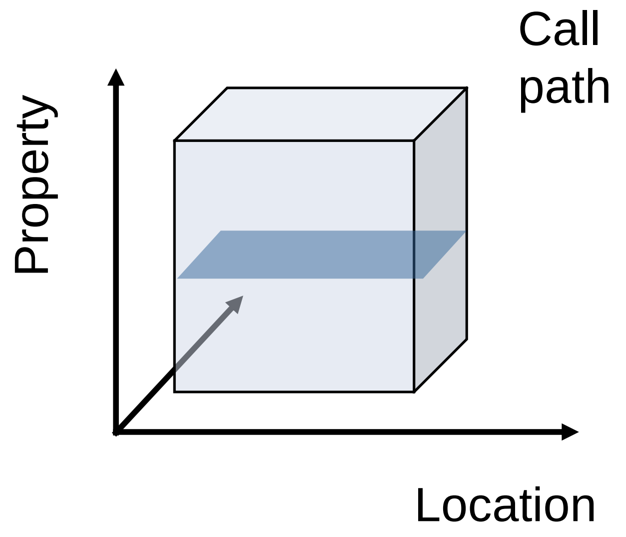
- Parallel program analysis report exploration tools
  - Libraries for XML+binary report reading & writing
  - Algebra utilities for report processing
  - GUI for interactive analysis exploration
    - Requires Qt4 ≥4.6 or Qt 5

- 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 4.3.4 (April 2016)

Analysis presentation and exploration

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

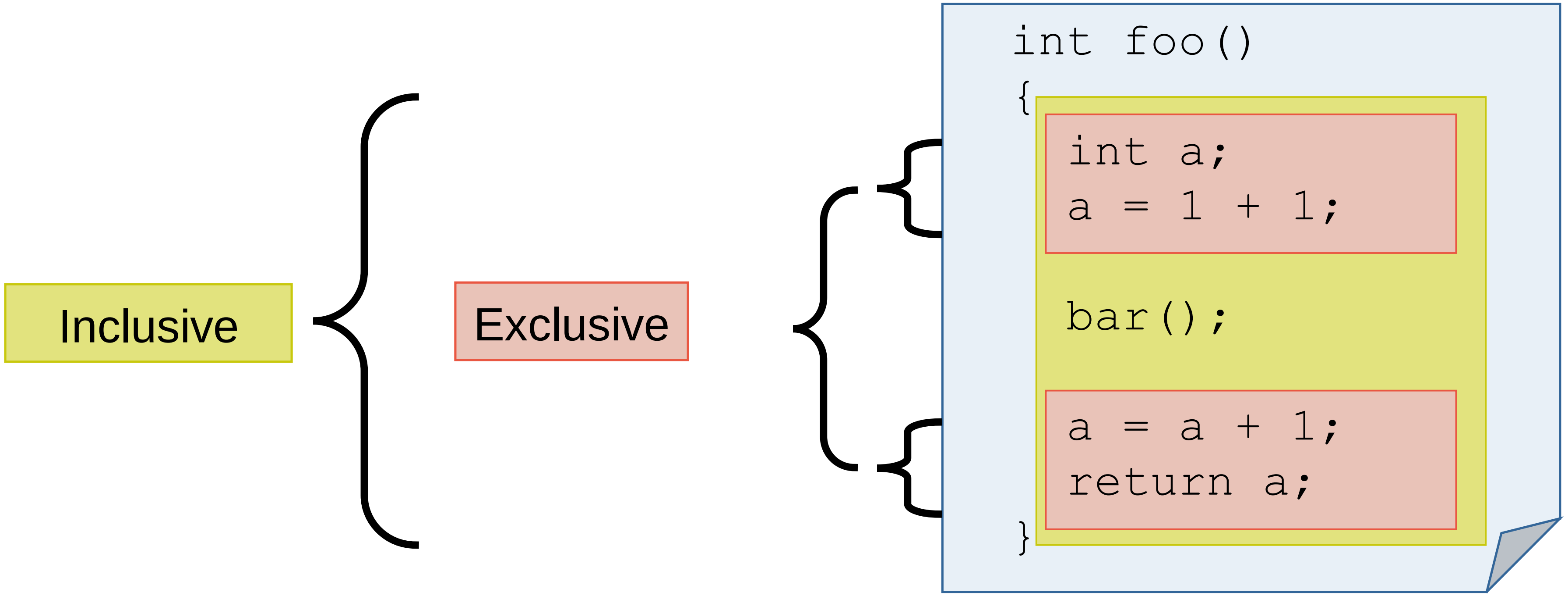


- Three coupled tree browsers

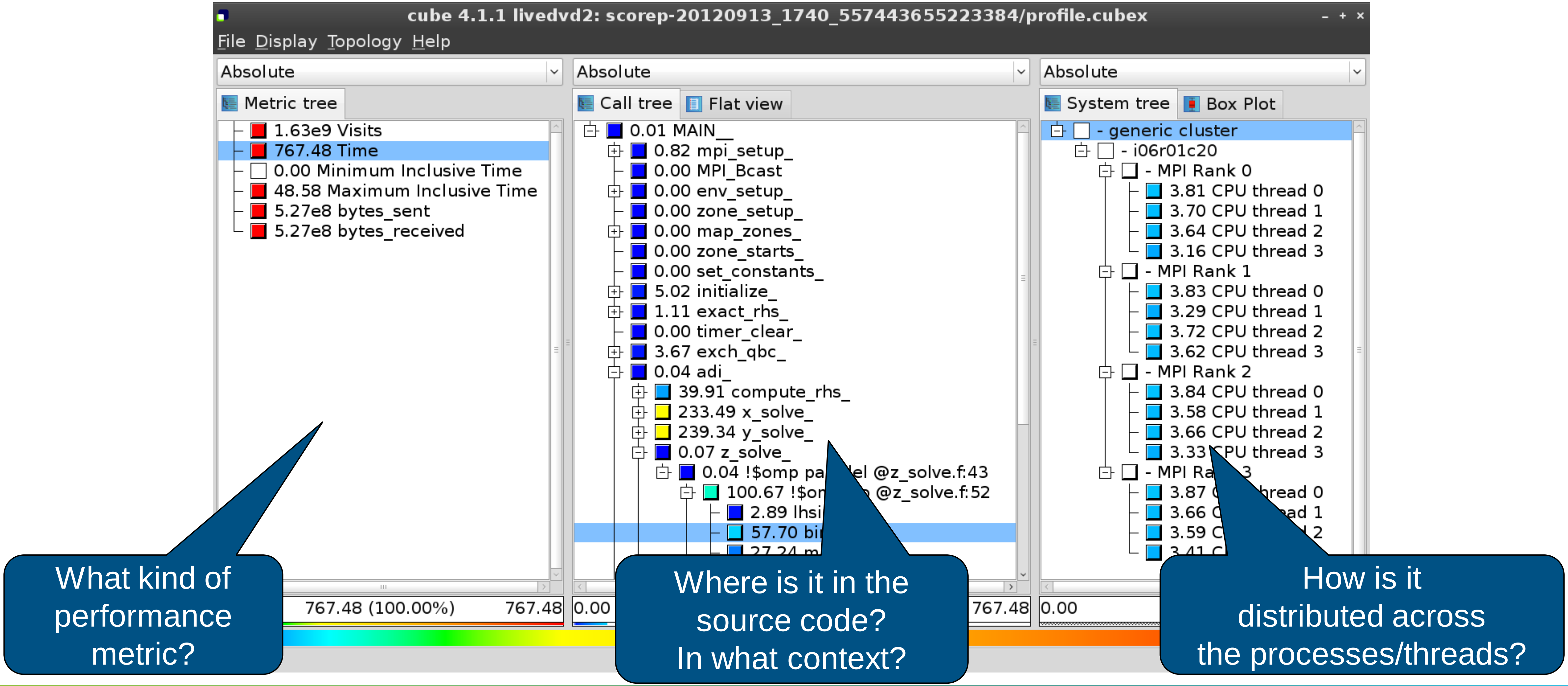
- Cube displays severities
  - As value: for precise comparison
  - As color: for easy identification of hotspots
  - Inclusive value when closed & exclusive value when expanded
  - Customizable via display modes

Inclusive vs. exclusive values

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



Analysis presentation

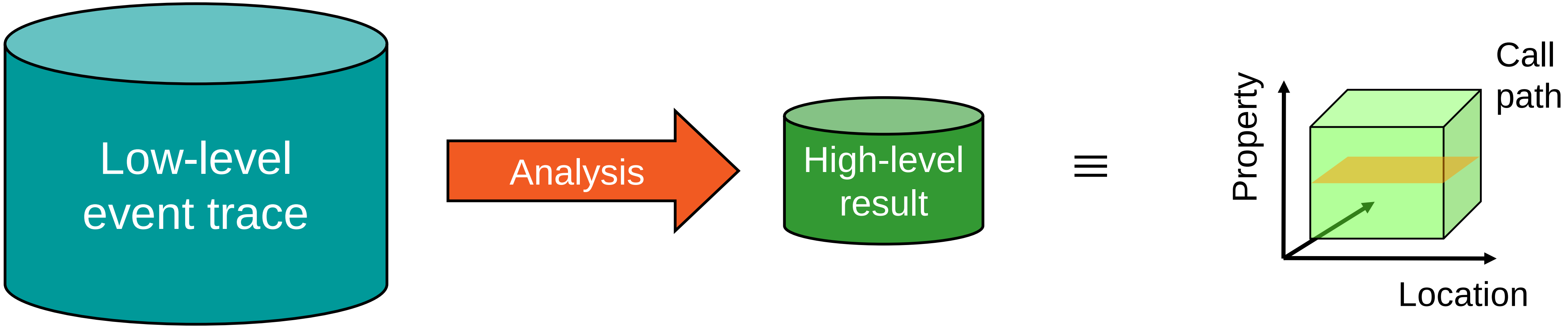


Automatic trace analysis with the Scalasca Trace Tools

Brian Wylie
Jülich Supercomputing Centre

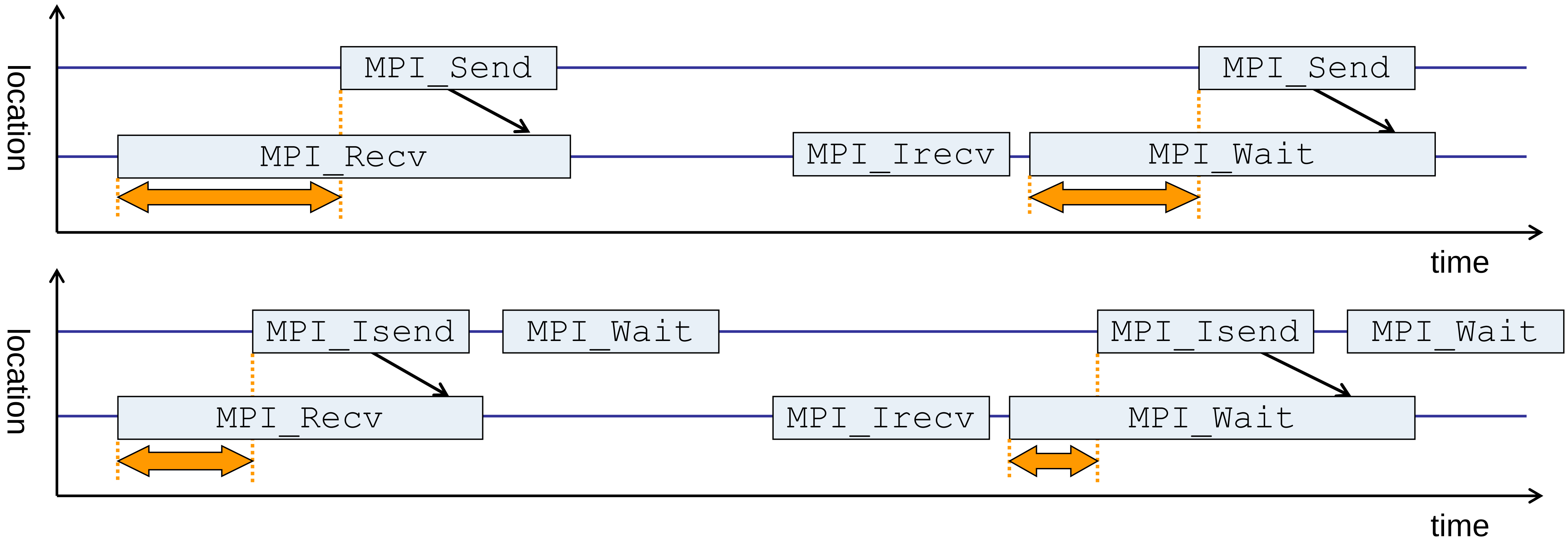
Automatic trace analysis using Scalasca

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



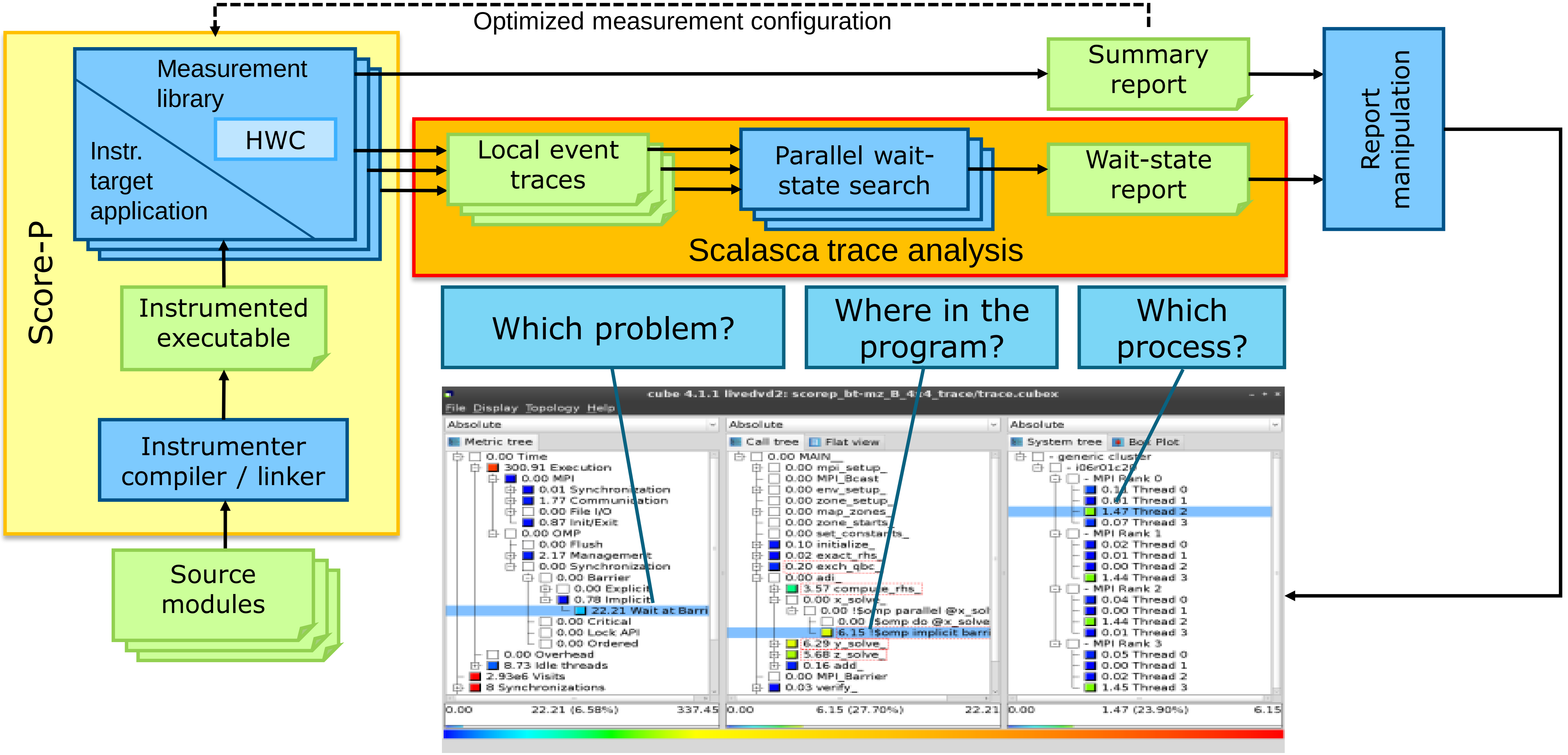
  - Guaranteed to cover the entire event trace
  - Quicker than manual/visual trace analysis
  - Parallel replay analysis exploits available memory & processors to deliver scalability

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

Scalasca workflow



Local setup (CLAIX)

- Load environment modules
  - Required for each shell session

```
% module load UNITE
% module load cube/4.3.4-gnu
% module load scorep/3.1-openmpi-intel-papi
% module load scalasca/2.3.1-openmpi-intel-papi
```

- Important:
  - Some Scalasca commands have a run-time dependency on Score-P & CUBE
  - Therefore also have those modules loaded when using Scalasca
  - Scalasca may need the same MPI and compiler modules as applications instrumented with Score-P

scalasca command – One command for (almost) everything

```
% scalasca
Scalasca 2.3.1
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 2.3.1: 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.
  -m metrics   : Metric specification for measurement.
```

- Scalasca measurement collection & analysis nexus

Scalasca convenience command: square / scalasca -examine

```
% square
Scalasca 2.3.1: analysis report explorer
usage: square [-v] [-s] [-f filtfile] [-F] <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                       : Skip display and output textual score report
   -v                       : Enable verbose mode
   -n                       : Do not include idle thread metric
```

- Scalasca analysis report explorer (Cube)

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)

BT-MZ summary measurement collection...

```
% cd bin.scorep
% cp ../jobscript/claix/scalasca.lsf .
% vim scalasca.lsf

 [...]

export SCOREP_FILTERING_FILE=scorep.filt
#export SCOREP_TOTAL_MEMORY=100M
#export SCOREP_METRIC_PAPI=PAPI_TOT_INS,PAPI_TOT_CYC

# Scalasca configuration
export SCAN_ANALYZE_OPTS="--time-correct"

scalasca -analyze  $MPIEXEC […] ./bt-mz_C.8
```

- Change to directory with the executable and edit the job script

```
% bsub < scalasca.lsf
```

- Submit the job

BT-MZ summary analysis report examination

- Score summary analysis report

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

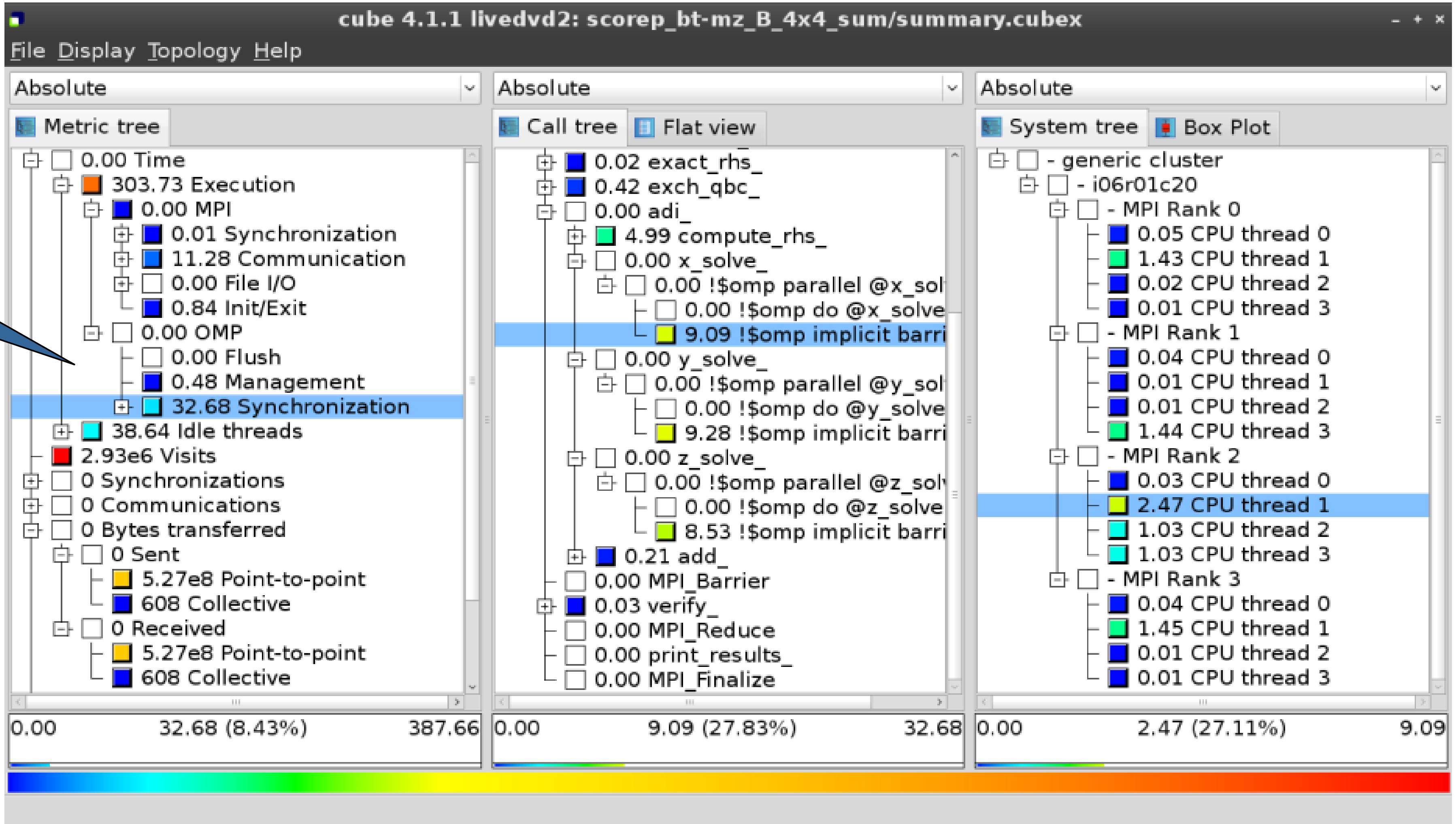
- Post-processing and interactive exploration with Cube

```
% square  scorep_bt-mz_C_8x6_sum
INFO: Displaying ./scorep_bt-mz_C_8x6_sum/summary.cubex...

                    [GUI showing summary analysis report]
```

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

Post-processed summary analysis report

Split base metrics into more specific metrics



BT-MZ trace measurement collection...

```
% cd bin.scorep
% cp ../jobscript/claix/scalasca.lsf .
% vim scalasca.lsf

 [...]

export SCOREP_FILTERING_FILE=scorep.filt
export SCOREP_TOTAL_MEMORY=100M
#export SCOREP_METRIC_PAPI=PAPI_TOT_INS,PAPI_TOT_CYC

# Scalasca configuration
export SCAN_ANALYZE_OPTS="--time-correct"

scalasca -analyze -t  mpiexec -np 8 ./bt-mz_C.8
```

```
% bsub < scalasca.lsf
```

- Change to directory with the executable and edit the job script
- Add "**-t**" to the scalasca -analyze command
- Set/uncomment **SCOREP_TOTAL_MEMORY** when more than 16MB per process
- Submit the job

BT-MZ trace measurement ... analysis

```
S=C=A=N: Fri Oct 21 10:06:16 2016: Analyze start
mpiexec -np 8 scout.hyb --time-correct \
        ./scorep_bt-mz_C_8x6_trace/traces.otf2

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

Opening experiment archive ... done (0.065s).
Reading definition data    ... done (0.191s).
Reading event trace data   ... done (0.839s).
Preprocessing              ... done (0.287s).
Timestamp correction       ... done (0.686s).
Analyzing trace data       ... done (8.884s).
Writing analysis report    ... done (0.219s).

Max. memory usage          : 795.684MB

Total processing time: 11.444s
S=C=A=N: Fri Oct 21 10:06:28 2016: Analyze done (status=0) 12s
```

- Continues with automatic (parallel) analysis of trace files

Timestamp correction is appropriate when clocks on compute nodes are not kept synchronized

Memory required for trace analysis typically several times size of trace files

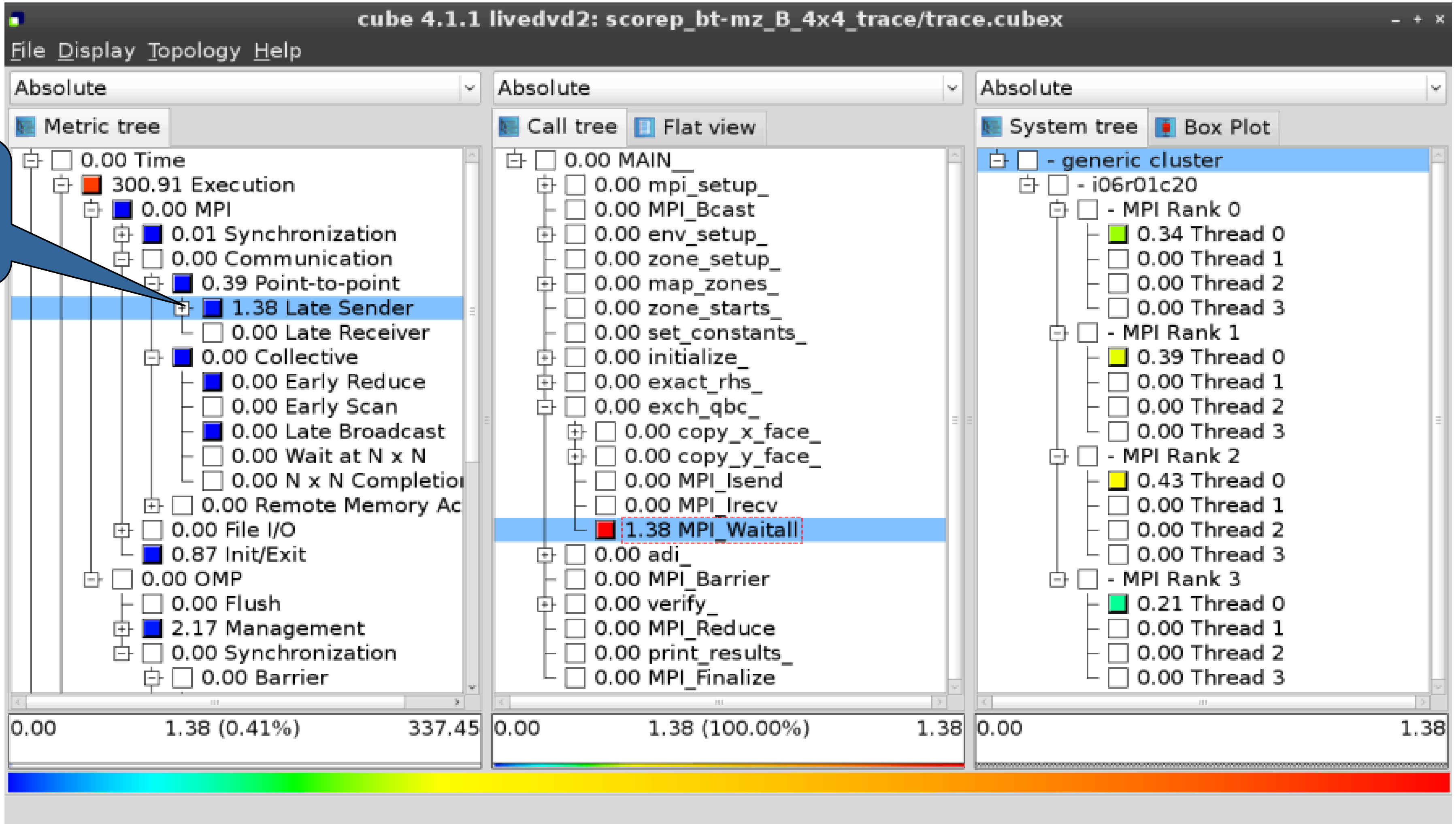
BT-MZ trace analysis report exploration

- 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 result...
INFO: Post-processing trace analysis report...
INFO: Displaying ./scorep_bt-mz_C_8x6_trace/trace.cubex...

                    [GUI showing trace analysis report]
```

Post-processed trace analysis report

Additional trace-based metrics in metric hierarchy



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