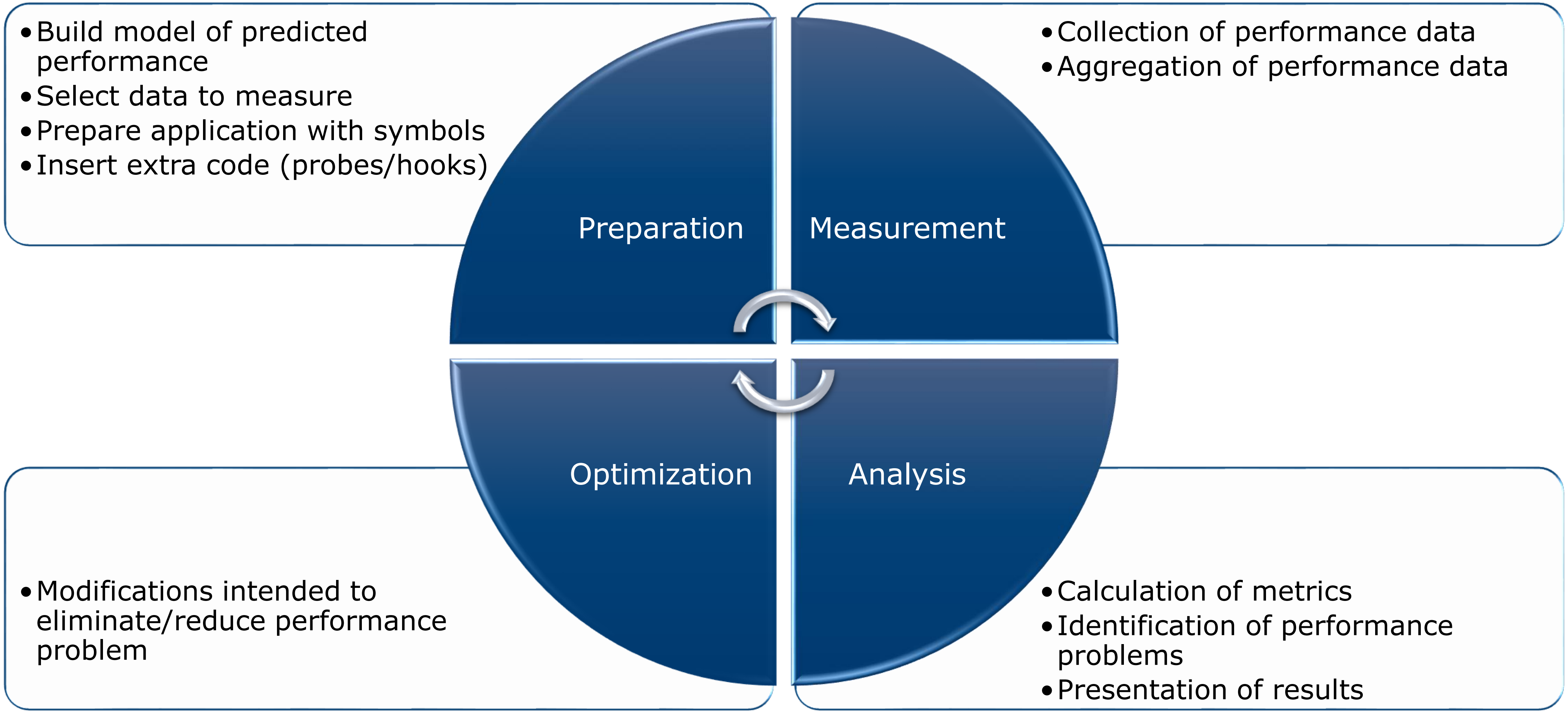
Score-P – Joint instrumentation & measurement infrastructure for Scalasca, TAU, and Vampir

VI-HPS Team

Score-P
Scalable performance measurement infrastructure for parallel codes

Barcelona Supercomputing Center – JÜLICH Forschungszentrum – Leibniz Supercomputing Centre – RWTH AACHEN UNIVERSITY – TECHNISCHE UNIVERSITÄT DRESDEN – UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES – Universität Stuttgart – FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG – Lawrence Livermore National Laboratory – Linaro – TECHNISCHE UNIVERSITÄT DARMSTADT – Technische Universität München – UNIVERSITY OF OREGON – THE UNIVERSITY of TENNESSEE

Performance engineering workflow



Preparation

- Build model of predicted performance
- Select data to measure
- 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

Score-P

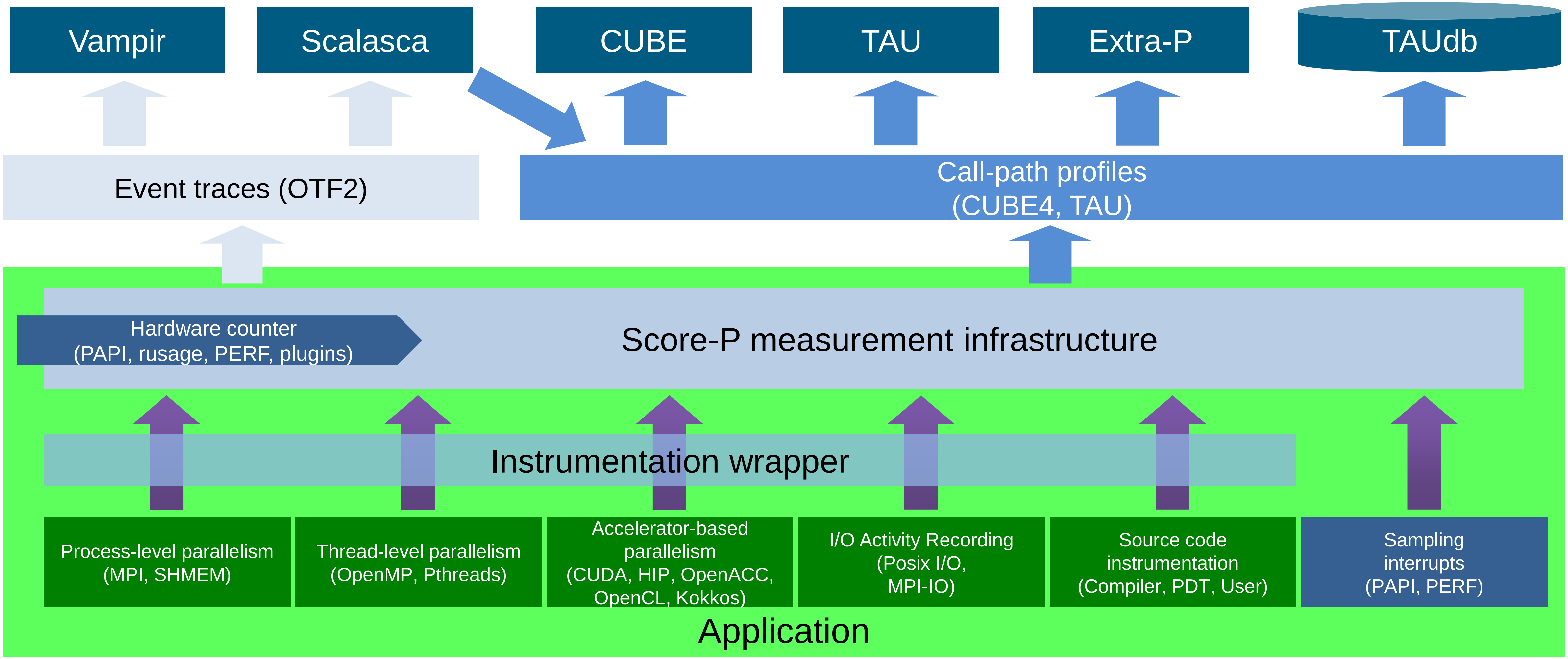
- Infrastructure for instrumentation and performance measurements
- Instrumented application can be used to produce several results:
  - Call-path profiling: CUBE4 data format used for data exchange
  - Event-based tracing: OTF2 data format used for data exchange

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


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

Partners

- Forschungszentrum Jülich, Germany
- Gesellschaft für numerische Simulation mbH Braunschweig, Germany
- RWTH Aachen, Germany
- Technische Universität Darmstadt, Germany
- Technische Universität Dresden, Germany
- Technische Universität München, Germany
- University of Oregon, Eugene, USA

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

Local installation (Karolina)

- Set account and environment (e.g. NVHPC + OpenMPI) via modules:

```
% module load nvompi
```

- Load the *corresponding* modules for the tool environment:

```
% module load Scalasca/2.6.1-NVHPC-24.3-CUDA-12.3.0
```

Scalasca module loads Score-P & CUBE module dependencies

- Copy example sources to your WORK directory (or your personal workspace)
  - Only required if not done already (for opening exercise)

```
% cd $WORK
% cp -r /mnt/proj2/dd-24-88/jsc/examples/CloverLeaf_OpenACC  .
% cd CloverLeaf_OpenACC
```

Score-P instrumenter

- `scorep` instrumenter is used as a preposition to compile & link commands

```
% scorep  ftn -fopenmp -c solve.f90
% scorep  cc -c timer.c
% scorep  mpif90 -o a.out solve.o timer.o -fopenmp -lfft -lcuda
```

- Instrumenter uses heuristics to determine when MPI & OpenMP are employed to perform source processing, direct compilers' function instrumentation and link measurement libraries
  - no heuristics yet for CUDA, Kokkos, OpenACC, …

- Instrumenter is highly configurable via flags: see `scorep --help`
  - should be used when heuristics fail or for custom instrumentation options

```
% scorep --cuda --nomemory  mpif90 -o a.out solve.o timer.o -fopenmp -lfft -lcuda
```

CloverLeaf_OpenACC: Makefile

```
#Crown Copyright 2012 AWE
#
# This file is part of CloverLeaf.
#
# CloverLeaf is free software...
#
# Agnostic, platform independent Makefile for the CloverLeaf benchmark code.
# It is not meant to be clever in any way, just a simple build script.
#
# this works as well:-
#
# make COMPILER=PGI [OPENMP=1]
#

...

#PREP=scorep --openacc --cuda --user

MPI_COMPILER=$(PREP) mpif90

# No preposition for C/CXX_MPI_COMPILER!
C_MPI_COMPILER=mpicc
CXX_MPI_COMPILER=mpic++

...
```

Specify the suite of compilers (and optionally OpenMP)

No instrumentation by default

Set/uncomment PREP macro for instrumenter preposition

Instrumenting clover_leaf

Score-P instrumenter options:
--compiler: source code routines (default)
--mpp=mpi: MPI determined by heuristics
--openacc: enable OpenACC
--cuda: enable CUDA
--user: enable Score-P user API (optional)

```
% make clean
% make PREP="scorep --openacc --cuda --user"

mpicc -c timer_c.c
scorep --openacc --cuda --user  mpif90 -O3 -acc=gpu -ta=nvidia \
    data.f90 definitions.f90 pack_kernel.f90 clover.F90 report.f90 timer.f90 \
    parse.f90 read_input.f90 initialise_chunk_kernel.f90 initialise_chunk.f90 build_field.f90 \
    update_tile_halo_kernel.f90 update_tile_halo.f90 update_halo_kernel.f90 update_halo.f90 \
    ideal_gas_kernel.f90 ideal_gas.f90 start.f90 generate_chunk_kernel.f90 generate_chunk.f90 \
    initialise.f90 field_summary_kernel.f90 field_summary.f90 viscosity_kernel.f90 viscosity.f90 \
    calc_dt_kernel.f90 calc_dt.f90 timestep.f90 accelerate_kernel.f90 accelerate.f90 \
    revert_kernel.f90 revert.f90 PdV_kernel.f90 PdV.f90 flux_calc_kernel.f90 flux_calc.f90 \
    advec_cell_kernel.f90 advec_cell_driver.f90 advec_mom_kernel.f90 advec_mom_driver.f90 \
    reset_field_kernel.f90 reset_field.f90 hydro.F90 clover_leaf.F90 visit.f90 \
    timer_c.o \
    -o bin.scorep/clover_leaf
```

Mastering build systems

- Hooking up the Score-P instrumenter `scorep` into complex build environments like *Autotools* or *CMake* was always challenging
- Score-P provides convenience wrapper scripts to simplify this
- *Autotools* and *CMake* need the used compiler already in the *configure step,* but instrumentation should not happen in this step only in the *build step*

```
% SCOREP_WRAPPER=off \
> cmake .. \
> -DCMAKE_C_COMPILER=scorep-icc \
> -DCMAKE_CXX_COMPILER=scorep-icpc \
> -DCMAKE_Fortran_COMPILER=scorep-ifort
```

Disable instrumentation in the *configure step*

Specify the wrapper scripts as the compiler to use

- Allows to pass addition options to the Score-P instrumenter and the compiler via environment variables without modifying *Makefiles* (**`SCOREP_WRAPPER_INSTRUMENTER_FLAGS`** & **`SCOREP_WRAPPER_COMPILER_FLAGS`**)
- Run `scorep-wrapper --help` for a detailed description and the available wrapper scripts of each Score-P installation (depends on configured compilers)

Measurement configuration: scorep-info

```
% scorep-info config-vars --full
SCOREP_ENABLE_PROFILING
  Description: Enable profiling
 [...]
SCOREP_ENABLE_TRACING
  Description: Enable tracing
 [...]
SCOREP_TOTAL_MEMORY
  Description: Total memory in bytes for the measurement system
 [...]
SCOREP_EXPERIMENT_DIRECTORY
  Description: Name of the experiment directory
 [...]
SCOREP_FILTERING_FILE
  Description: A file name which contain the filter rules
 [...]
SCOREP_METRIC_PAPI
  Description: PAPI metric names to measure
 [...]
SCOREP_METRIC_RUSAGE
  Description: Resource usage metric names to measure
 [...]
SCOREP_OPENACC_ENABLE
  Description: OpenACC measurement features
 [... More configuration variables ...]
```

- Score-P measurements are configured via environmental variables

Required for OpenACC measurements.
[yes|default] recommended to start.
Additional CUDA measurement optional.

Score-P 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

% export SCOREP_FILTERING_FILE=\
../config/scorep.filt
```

Region name filter block using wildcards

Apply filter

- Filtering by source file name
  - All regions in files that are excluded by the filter are ignored
- Filtering by region name
  - All regions that are excluded by the filter are ignored
  - Overruled by source file filter for excluded files
- Apply filter by
  - exporting **`SCOREP_FILTERING_FILE`** environment variable
- Apply filter at
  - Run-time
  - Compile-time (GCC-plugin and Intel only)
    - Add cmd-line option **`--instrument-filter`**
    - No overhead for filtered regions but recompilation

Source file name filter block

- Keywords
  - Case-sensitive
  - SCOREP_FILE_NAMES_BEGIN, SCOREP_FILE_NAMES_END
    - Define the source file name filter block
    - Block contains EXCLUDE, INCLUDE rules
  - EXCLUDE, INCLUDE rules
    - Followed by one or multiple white-space separated source file names
    - Names can contain bash-like wildcards *, ?, []
    - Unlike bash, * may match a string that contains slashes
- EXCLUDE, INCLUDE rules are applied in sequential order
- Regions in source files that are excluded after all rules are evaluated, get filtered

```
# This is a comment
SCOREP_FILE_NAMES_BEGIN
  # by default, everything is included
  EXCLUDE */foo/bar*
  INCLUDE */filter_test.c
SCOREP_FILE_NAMES_END
```

Region name filter block

- Keywords
  - Case-sensitive
  - SCOREP_REGION_NAMES_BEGIN, SCOREP_REGION_NAMES_END
    - Define the region name filter block
    - Block contains EXCLUDE, INCLUDE rules
  - EXCLUDE, INCLUDE rules
    - Followed by one or multiple white-space separated region names
    - Names can contain bash-like wildcards *, ?, []
- EXCLUDE, INCLUDE rules are applied in sequential order
- Regions that are excluded after all rules are evaluated, get filtered

```
# This is a comment
SCOREP_REGION_NAMES_BEGIN
  # by default, everything is included
  EXCLUDE *
  INCLUDE bar foo
          baz
          main
SCOREP_REGION_NAMES_END
```

Region name filter block, mangling

- Name mangling
  - Filtering based on names seen by the measurement system
    - Dependent on compiler
    - Actual name may be mangled
- `scorep-score` names as starting point (e.g. `matvec_sub_`)
    - Use `*` for Fortran trailing underscore(s) for portability
    - Use `?` and `*` as needed for full signatures or overloading
    - Use `\` to escape special characters

```
void bar(int* a) {
    *a++;
}
int main() {
    int i = 42;
    bar(&i);
    return 0;
}
```

```
# filter bar:
# for gcc-plugin, scorep-score
# displays 'void bar(int*)',
# other compilers may differ

SCOREP_REGION_NAMES_BEGIN
  EXCLUDE void?bar(int?)
SCOREP_REGION_NAMES_END
```

New: generate initial filter file

```
% score-scorep --help
[…]
 -g [<list>] Generation of an initial filter file with the name
             'initial_scorep.filter'. A valid parameter list has the form
             KEY=VALUE[,KEY=VALUE]*. By default, uses the following control
             parameters:

                 `bufferpercent=1,timepervisit=1`

             A region is included in the filter file (i.e., excluded from
             measurement) if it matches all of the given conditions, with the
             following keys:
             - `bufferpercent`        : estimated memory requirements exceed the
                                        given threshold in percent of the total
                                        estimated trace buffer requirements
             - `bufferabsolute`       : estimated memory requirements exceed
                                        the given absolute threshold in MB
             - `visits`               : number of visits exceeds the given
                                        threshold
             […]
```

Score-P:
Specialized Measurements and Analyses

Score-P user instrumentation API

- Can be used to partition application into coarse grain phases
  - E.g., initialization, solver, & finalization

- Can be used to further subdivide functions
  - E.g., multiple loops inside a function

- Enabled with `--user` flag to Score-P instrumenter

- Available for Fortran / C / C++

Score-P user instrumentation API (Fortran)

```
#include "scorep/SCOREP_User.inc"

subroutine foo(…)
  ! Declarations
  SCOREP_USER_REGION_DEFINE( solve )

  ! Some code…
  SCOREP_USER_REGION_BEGIN( solve, “<solver>", \
                            SCOREP_USER_REGION_TYPE_LOOP )
  do i=1,100
    [...]
  end do
  SCOREP_USER_REGION_END( solve )
  ! Some more code…
end subroutine
```

- Requires processing by the C preprocessor
  - For most compilers, this can be automatically achieved by having an uppercase file extension, e.g., `main.F` or `main.F90`

Score-P user instrumentation API (C/C++)

```
#include "scorep/SCOREP_User.h"

void foo()
{
  /* Declarations */
  SCOREP_USER_REGION_DEFINE( solve )

  /* Some code… */
  SCOREP_USER_REGION_BEGIN( solve, “<solver>",
                            SCOREP_USER_REGION_TYPE_LOOP )
  for (i = 0; i < 100; i++)
  {
    [...]
  }
  SCOREP_USER_REGION_END( solve )
  /* Some more code… */
}
```

Score-P user instrumentation API (C++)

```
#include "scorep/SCOREP_User.h"

void foo()
{
  // Declarations

  // Some code…
  {
    SCOREP_USER_REGION( “<solver>",
                        SCOREP_USER_REGION_TYPE_LOOP )
    for (i = 0; i < 100; i++)
    {
      [...]
    }
  }
  // Some more code…
}
```

Score-P measurement control API

- Can be used to temporarily disable measurement for certain intervals
  - Annotation macros ignored by default
  - Enabled with `--user` flag

```
#include “scorep/SCOREP_User.inc”

subroutine foo(…)
  ! Some code…
  SCOREP_RECORDING_OFF()
  ! Loop will not be measured
  do i=1,100
    [...]
  end do
  SCOREP_RECORDING_ON()
  ! Some more code…
end subroutine
```

Fortran (requires C preprocessor)

```
#include “scorep/SCOREP_User.h”

void foo(…) {
  /* Some code… */
  SCOREP_RECORDING_OFF()
  /* Loop will not be measured */
  for (i = 0; i < 100; i++) {
    [...]
  }
  SCOREP_RECORDING_ON()
  /* Some more code… */
}
```

C / C++

Enriching measurements with performance counters

- Record metrics from PAPI:

```
% export SCOREP_METRIC_PAPI=PAPI_TOT_CYC
% export SCOREP_METRIC_PAPI_PER_PROCESS=PAPI_L3_TCM
```

  - Use PAPI tools to get available metrics and valid combinations:

```
% papi_avail
% papi_native_avail
```

- Record metrics from Linux perf:

```
% export SCOREP_METRIC_PERF=cpu-cycles
% export SCOREP_METRIC_PERF_PER_PROCESS=LLC-load-misses
```

Only the master thread records the metric (assuming all threads of the process access the same L3 cache)

  - Use the `perf` tool to get available metrics and valid combinations:

```
% perf list
```

- Write your own metric plugin
  - Repository of available plugins: https://github.com/score-p

Mastering heterogeneous applications

- Record CUDA applications and device activities

```
% export SCOREP_CUDA_ENABLE=runtime,kernel,idle
```

Idle is an artificial region defined as outside of kernel time

- Record OpenCL applications and device activities

```
% export SCOREP_OPENCL_ENABLE=api,kernel
```

- Record OpenACC applications

```
% export SCOREP_OPENACC_ENABLE=yes
```

  - Can be combined with CUDA if it is a NVIDIA device

```
% export SCOREP_CUDA_ENABLE=kernel
```

  - Check **`scorep-info config-vars –full`** for a wide range of further options and default values

Adding options will increase overhead to a varying degree

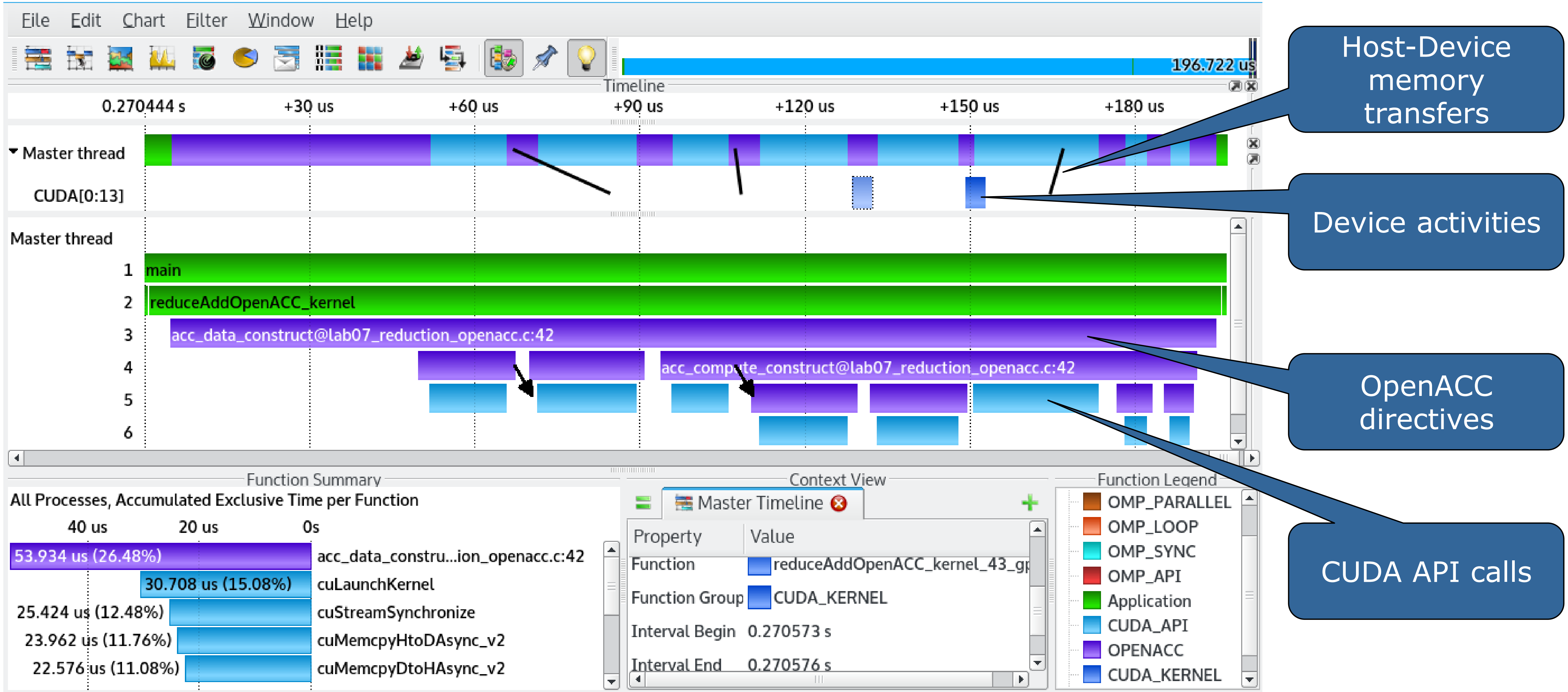
HIP/ROCm instrumentation

- Instrument with "scorep **--hip**" to ensure ROCm adapter is included
  - alternatively SCOREP_WRAPPER_INSTRUMENTER_FLAGS="--hip …"

- For measurement execution set SCOREP_HIP_ENABLE
  - api: all HIP API calls
  - kernel: HIP kernels
    - kernel_callsite: additional tracking of kernel callsites between launch and execution
  - malloc: HIP-managed host and device allocations
  - memcpy: H2D, D2H, H2H copies through HIP memcpy functions (not yet for D2D)
  - sync: device/stream synchronization calls
  - user: ROCTx support

  - default/yes/1/true: all of the above
  - none/no: disable feature

Mastering heterogeneous applications



All Processes, Accumulated Exclusive Time per Function

Time	Function
53.934 us (26.48%)	acc_data_constru...ion_openacc.c:42
30.708 us (15.08%)	cuLaunchKernel
25.424 us (12.48%)	cuStreamSynchronize
23.962 us (11.76%)	cuMemcpyHtoDAsync_v2
22.576 us (11.08%)	cuMemcpyDtoHAsync_v2

Master Timeline

Property	Value
Function	reduceAddOpenACC_kernel_43_gp
Function Group	CUDA_KERNEL
Interval Begin	0.270573 s
Interval End	0.270576 s

Function Legend:
- OMP_PARALLEL
- OMP_LOOP
- OMP_SYNC
- OMP_API
- Application
- CUDA_API
- OPENACC
- CUDA_KERNEL

Host-Device memory transfers

Device activities

OpenACC directives

CUDA API calls

Mastering application memory usage

- Determine the maximum heap usage per process
- Find high frequent small allocation patterns
- Find memory leaks
- Support for:
  - C, C++, MPI, and SHMEM (Fortran only for GNU Compilers)
  - Profile and trace generation (profile recommended)
    - Memory leaks are recorded only in the profile
    - Resulting traces are not supported by Scalasca yet

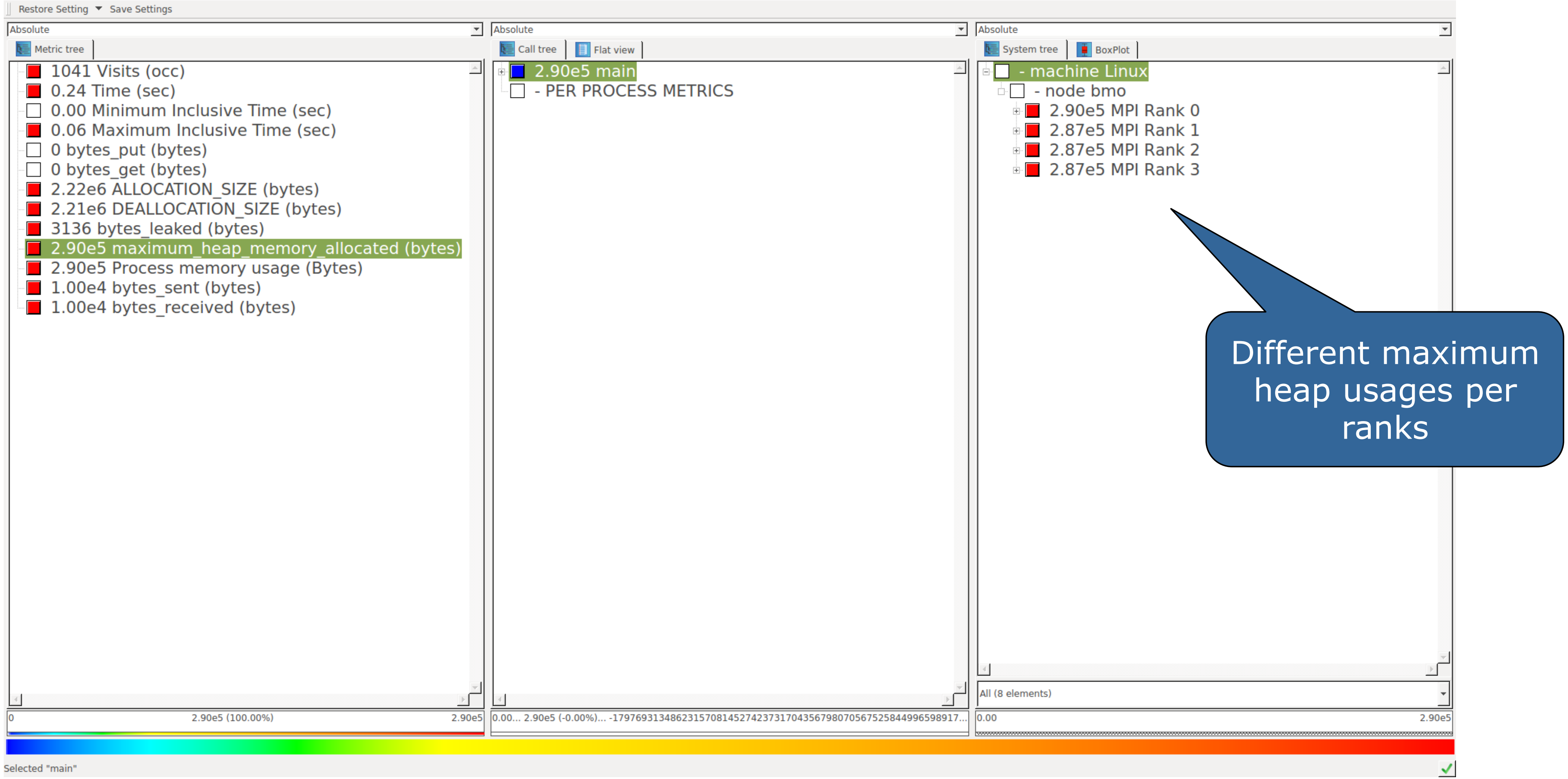
```
% export SCOREP_MEMORY_RECORDING=true
% export SCOREP_MPI_MEMORY_RECORDING=true

% OMP_NUM_THREADS=4 mpiexec -np 4 ./bt-mz_W.4
```

- Set new configuration variable to enable memory recording

- Available since Score-P 2.0

Mastering application memory usage



- 🟥 1041 Visits (occ)
- 🟥 0.24 Time (sec)
- □ 0.00 Minimum Inclusive Time (sec)
- 🟥 0.06 Maximum Inclusive Time (sec)
- □ 0 bytes_put (bytes)
- □ 0 bytes_get (bytes)
- 🟥 2.22e6 ALLOCATION_SIZE (bytes)
- 🟥 2.21e6 DEALLOCATION_SIZE (bytes)
- 🟥 3136 bytes_leaked (bytes)
- 🟥 2.90e5 maximum_heap_memory_allocated (bytes)
- 🟥 2.90e5 Process memory usage (Bytes)
- 🟥 1.00e4 bytes_sent (bytes)
- 🟥 1.00e4 bytes_received (bytes)

0 2.90e5 (100.00%) 2.90e5

- 🟦 2.90e5 main
- □ - PER PROCESS METRICS

0.00... 2.90e5 (-0.00%)... -179769313486231570814527423731704356798070567525844996598917...

- □ - machine Linux
  - □ - node bmo
    - 🟥 2.90e5 MPI Rank 0
    - 🟥 2.87e5 MPI Rank 1
    - 🟥 2.87e5 MPI Rank 2
    - 🟥 2.87e5 MPI Rank 3

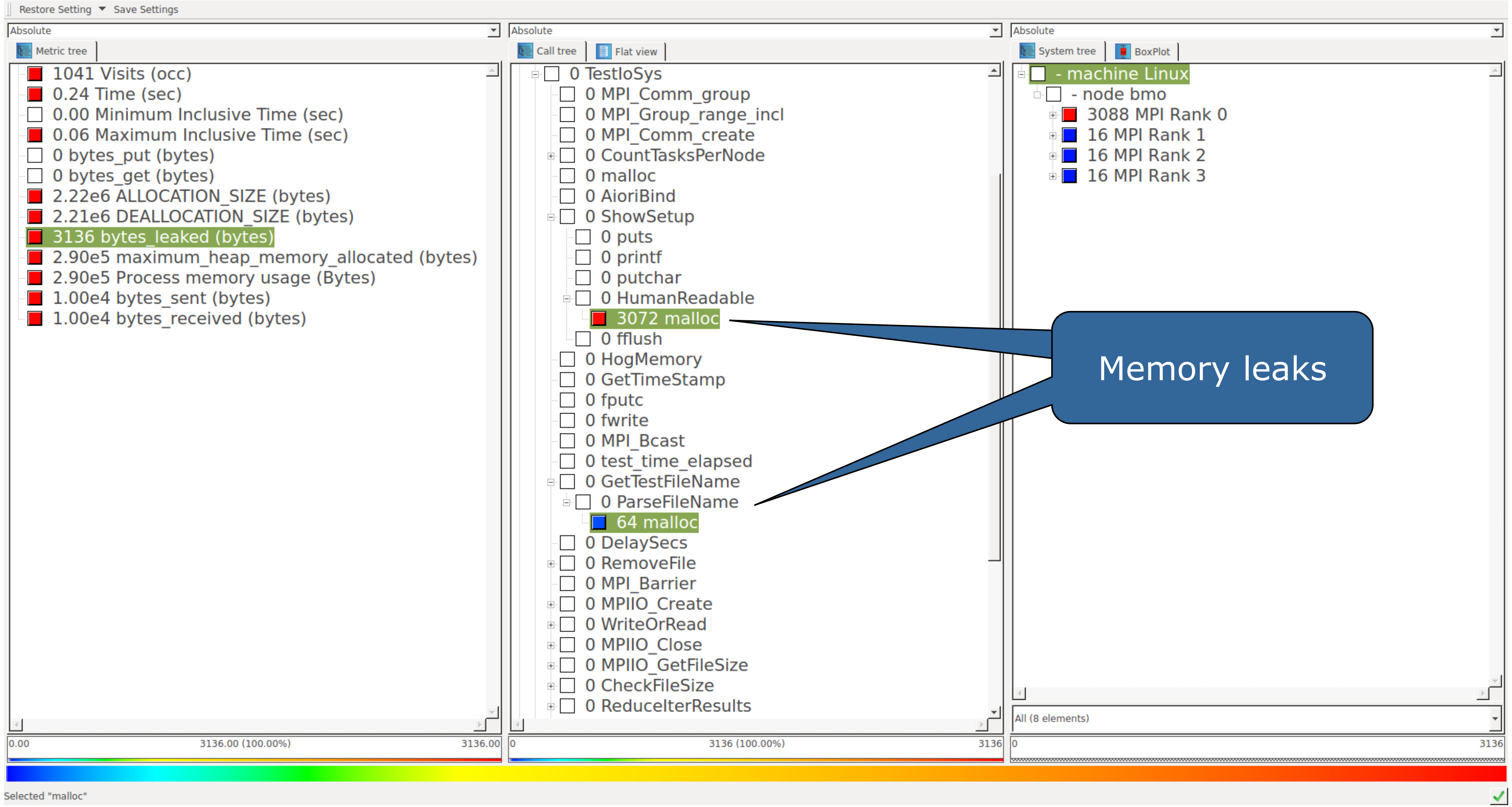
All (8 elements)

0.00 2.90e5

Selected "main"

Different maximum heap usages per ranks

Mastering application memory usage



```
🟥 1041 Visits (occ)
🟥 0.24 Time (sec)
□ 0.00 Minimum Inclusive Time (sec)
🟥 0.06 Maximum Inclusive Time (sec)
□ 0 bytes_put (bytes)
□ 0 bytes_get (bytes)
🟥 2.22e6 ALLOCATION_SIZE (bytes)
🟥 2.21e6 DEALLOCATION_SIZE (bytes)
🟥 3136 bytes_leaked (bytes)
🟥 2.90e5 maximum_heap_memory_allocated (bytes)
🟥 2.90e5 Process memory usage (Bytes)
🟥 1.00e4 bytes_sent (bytes)
🟥 1.00e4 bytes_received (bytes)
```

```
□ 0 TestIoSys
  □ 0 MPI_Comm_group
  □ 0 MPI_Group_range_incl
  □ 0 MPI_Comm_create
  □ 0 CountTasksPerNode
  □ 0 malloc
  □ 0 AioriBind
  □ 0 ShowSetup
    □ 0 puts
    □ 0 printf
    □ 0 putchar
    □ 0 HumanReadable
      🟥 3072 malloc
    □ 0 fflush
  □ 0 HogMemory
  □ 0 GetTimeStamp
  □ 0 fputc
  □ 0 fwrite
  □ 0 MPI_Bcast
  □ 0 test_time_elapsed
  □ 0 GetTestFileName
    □ 0 ParseFileName
      🟦 64 malloc
  □ 0 DelaySecs
  □ 0 RemoveFile
  □ 0 MPI_Barrier
  □ 0 MPIIO_Create
  □ 0 WriteOrRead
  □ 0 MPIIO_Close
  □ 0 MPIIO_GetFileSize
  □ 0 CheckFileSize
  □ 0 ReduceIterResults
```

```
□ - machine Linux
  □ - node bmo
    🟥 3088 MPI Rank 0
    🟦 16 MPI Rank 1
    🟦 16 MPI Rank 2
    🟦 16 MPI Rank 3
```

All (8 elements)

```
0.00        3136.00 (100.00%)        3136.00
0           3136 (100.00%)           3136
0                                    3136
```

Selected "malloc"

Memory leaks

Mastering C++ applications

- Automatic compiler instrumentation greatly disturbs C++ applications because of frequent/short function calls => Use sampling instead
- Novel combination of sampling events and instrumentation of MPI, OpenMP, …
  - Sampling replaces compiler instrumentation (instrument with `--nocompiler` to further reduce overhead) => Filtering not needed anymore
  - Instrumentation is used to get accurate times for parallel activities to still be able to identifies patterns of inefficiencies
- Supports profile and trace generation

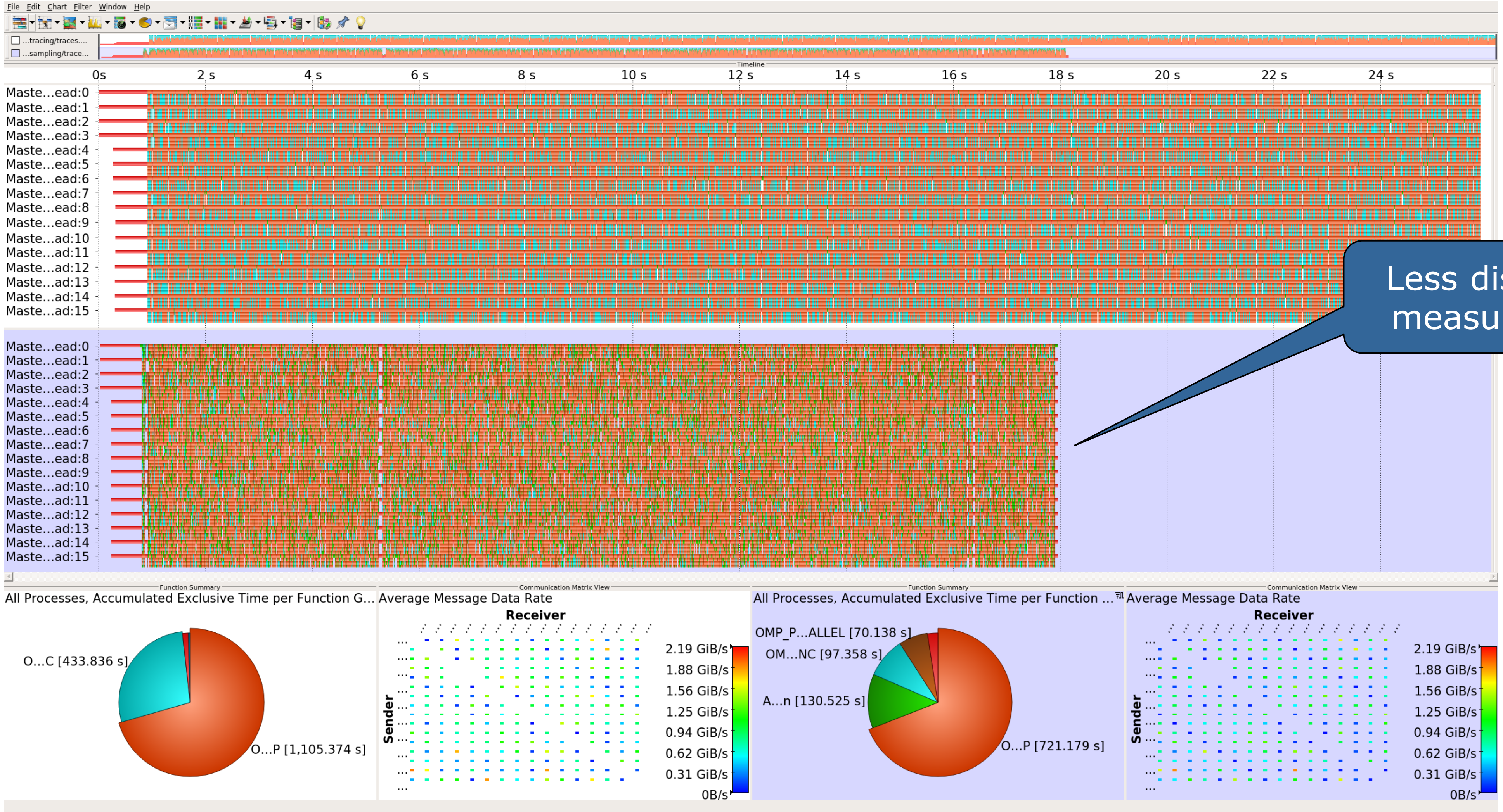
```
% export SCOREP_ENABLE_UNWINDING=true
% # use the default sampling frequency
% #export SCOREP_SAMPLING_EVENTS=perf_cycles@2000000

% OMP_NUM_THREADS=4 mpiexec -np 4 ./bt-mz_W.4
```

- Set new configuration variable to enable sampling

- Available since Score-P 2.0, only x86-64 supported currently

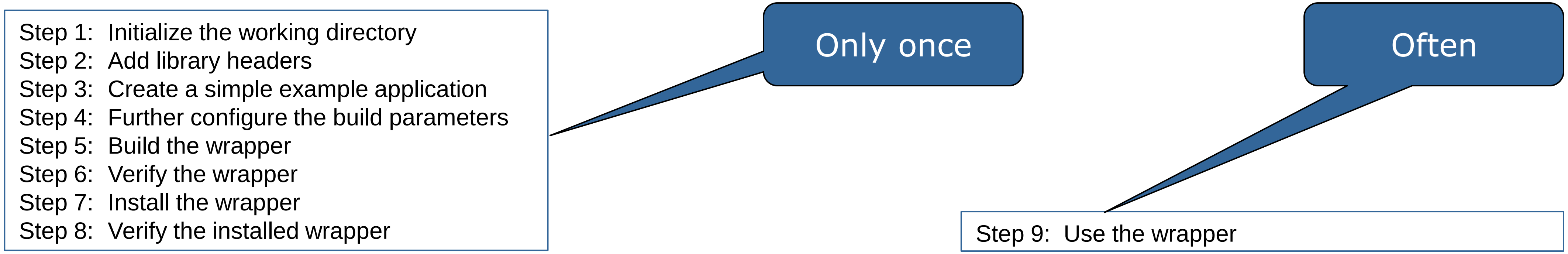
Mastering C++ applications



Less disturbed measurement

Wrapping calls to 3rd party libraries

- Enables users to install library wrappers for any C/C++ library
- Intercept calls to a library API
  - no need to either build the library with Score-P or add manual instrumentation to the application using the library
  - no need to access the source code of the library, header and library files suffice
- Score-P needs to be executed with `--libwrap=…`
- Execute `scorep-libwrap-init` for directions:



Step 1: Initialize the working directory
Step 2: Add library headers
Step 3: Create a simple example application
Step 4: Further configure the build parameters
Step 5: Build the wrapper
Step 6: Verify the wrapper
Step 7: Install the wrapper
Step 8: Verify the installed wrapper

Only once

Often

Step 9: Use the wrapper

Wrapping calls to 3rd party libraries

- Generate your own library wrappers by telling `scorep-libwrap-init` how you would compile and link an application, e.g. using FFTW

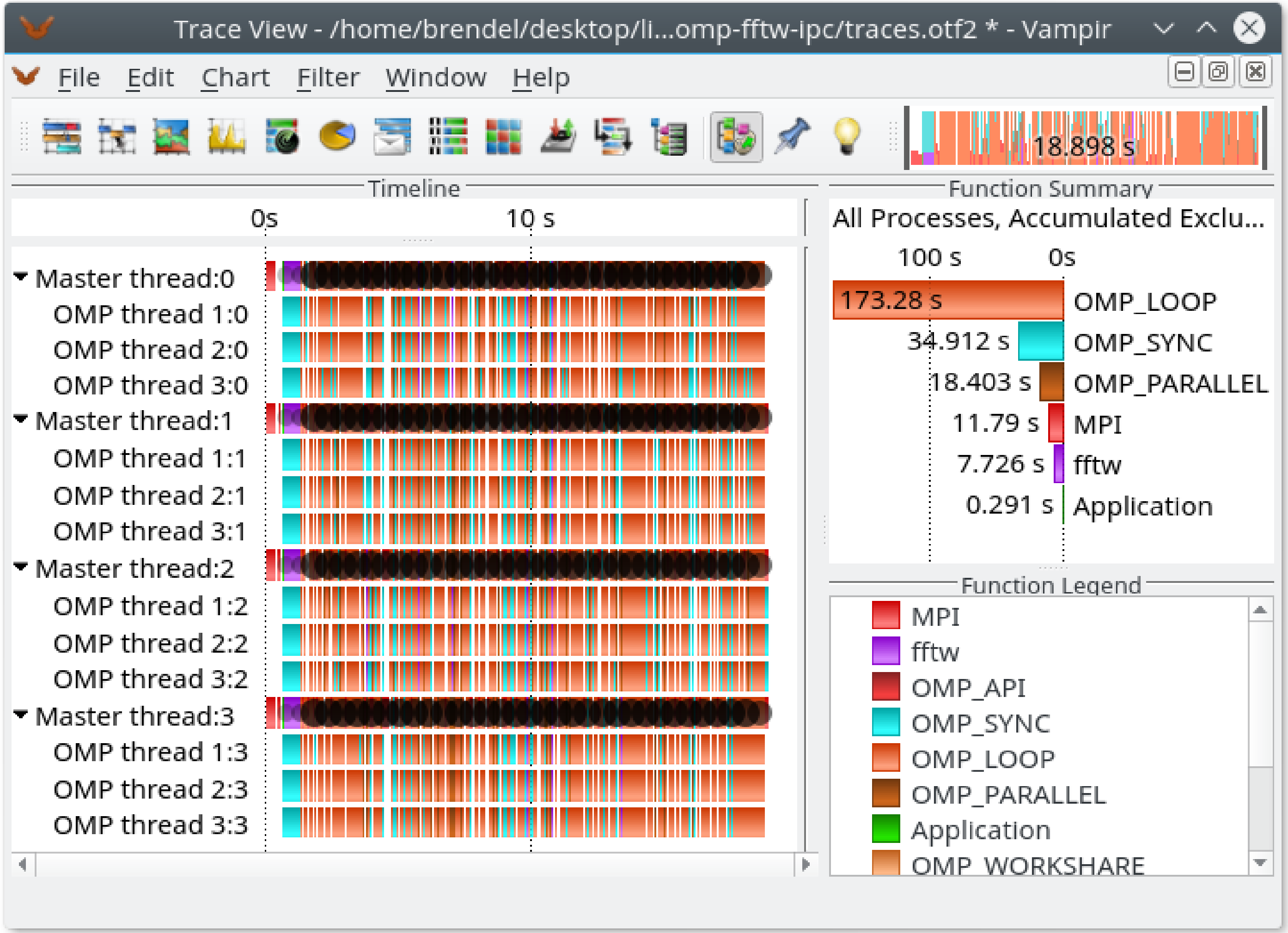
```
% scorep-libwrap-init         \
>   --name=fftw               \
>   --prefix=$PREFIX          \
>   -x c                      \
>   --cppflags=“-O3 -DNDEBUG -openmp -I$FFTW_INC“ \
>   --ldflags=“-L$FFTW_LIB“   \
>   --libs=“-lfftw3f -lfftw3“ \
>   working_directory
```

- Generate and build wrapper

```
% cd working_directory
% ls                  #  (Check README.md for instructions)
% make                # Generate and build wrapper
% make check          # See if header analysis matches symbols
% make install        #
% make installcheck   # More checks: Linking etc.
```

Wrapping calls to 3rd party libraries

- MPI + OpenMP
- Calls to FFTW library



Further information

- Community instrumentation & measurement infrastructure
  - Instrumentation (various methods) and sampling
  - Basic and advanced profile generation
  - Event trace recording
- Available under 3-clause BSD open-source license
- Documentation & Sources:
  - http://www.score-p.org
- User guide also part of installation:
  - `<prefix>/share/doc/scorep/{pdf,html}/`
- Support and feedback: support@score-p.org
- Subscribe to news@score-p.org, to be up to date