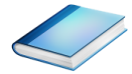
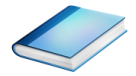
Score-P:
Specialized Measurements and Analyses



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*

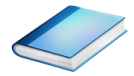
```
% SCOREP_WRAPPER=off \
>   cmake .. \
>   -DCMAKE_C_COMPILER=scorep-icc \
>   -DCMAKE_CXX_COMPILER=scorep-icpc
```

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 the *Makefiles*
- Run `scorep-wrapper --help` for a detailed description and the available wrapper scripts of the Score-P installation

User instrumentation



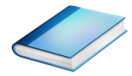
- No replacement for automatic compiler instrumentation

- Can be used to further subdivide functions
  - E.g., multiple loops inside a function
- Can be used to partition application into coarse grain phases
  - E.g., initialization, solver, & finalization

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

- Available for Fortran / C / C++

User instrumentation: Fortran



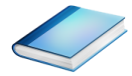
```
#include "scorep/SCOREP_User.inc"

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`

User instrumentation: C/C++

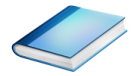


```
#include "scorep/SCOREP_User.h"

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… */
}
```

User instrumentation: C++

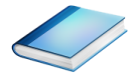


```
#include "scorep/SCOREP_User.h"

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
```

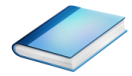
Fortran (requires C preprocessor)

```
#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
```

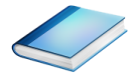
  - Use the `perf` tool to get available metrics and valid combinations:

```
% perf list
```

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

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

Mastering application memory usage

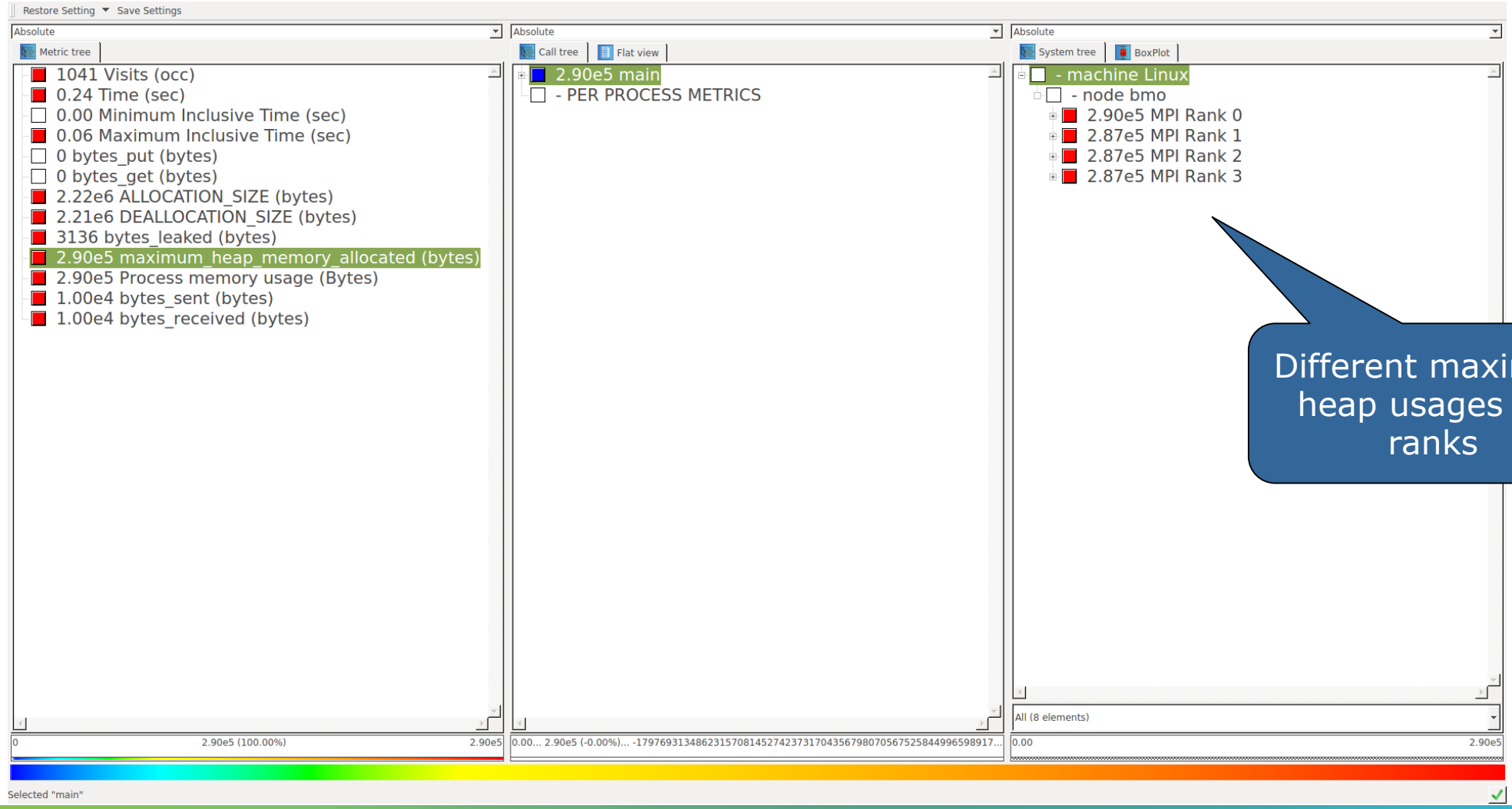


- Determine the maximum heap usage per process
- Find high frequent small allocation patterns
- Find memory leaks
- Support for:
  - C, C++, MPI, and SHMEM (Fortran only for GNU Compilers)
  - Profile and trace generation (profile recommended)
    - Memory leaks are recorded only in the profile
    -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

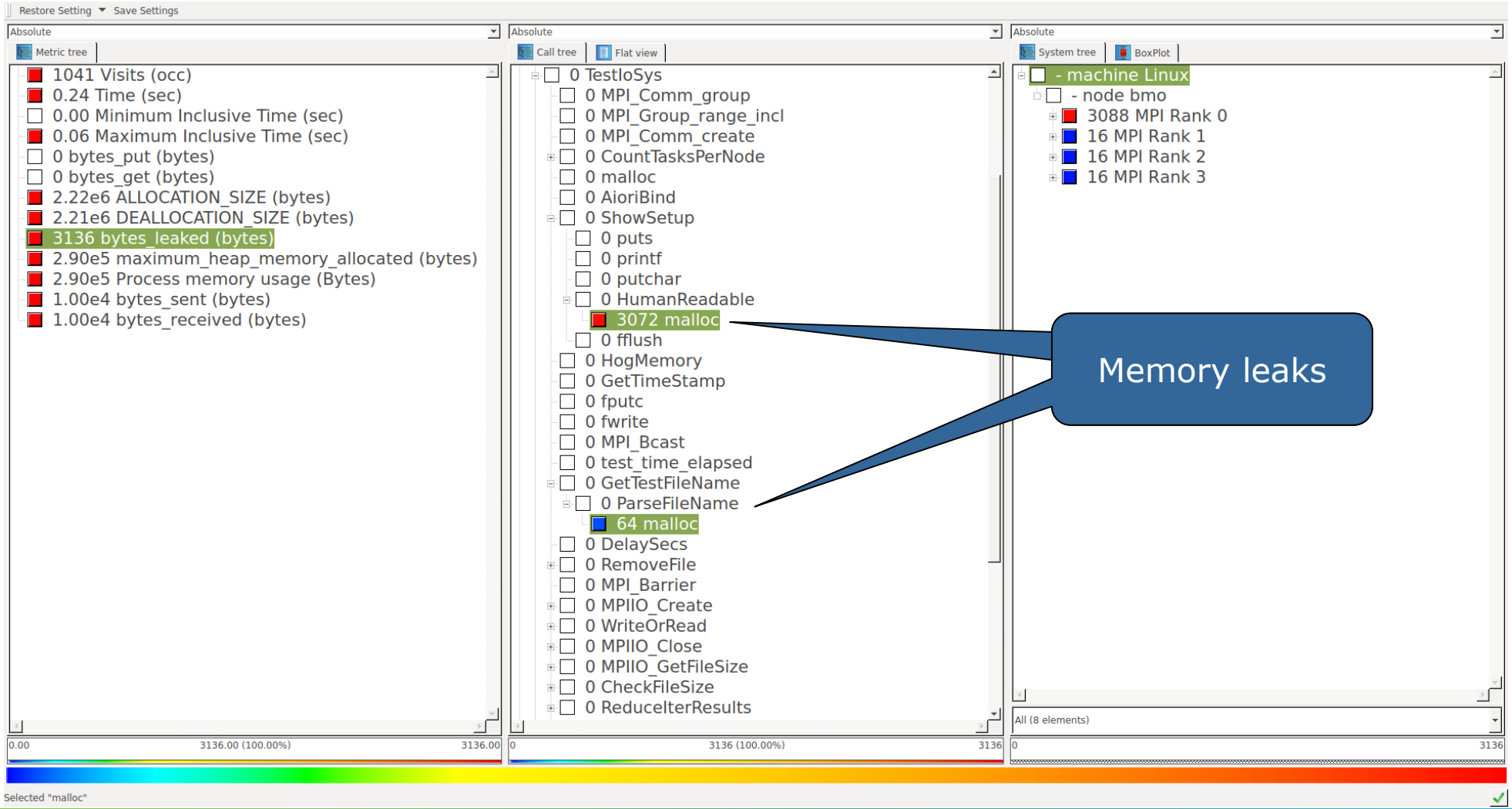
```
% export SCOREP_MEMORY_RECORDING=true
% export SCOREP_MPI_MEMORY_RECORDING=true
```

- Set new configuration variable to enable memory recording

Mastering application memory usage



Mastering application memory usage



Mastering accelerators

- Record CUDA applications and device activities

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

- Record OpenCL applications and device activities

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

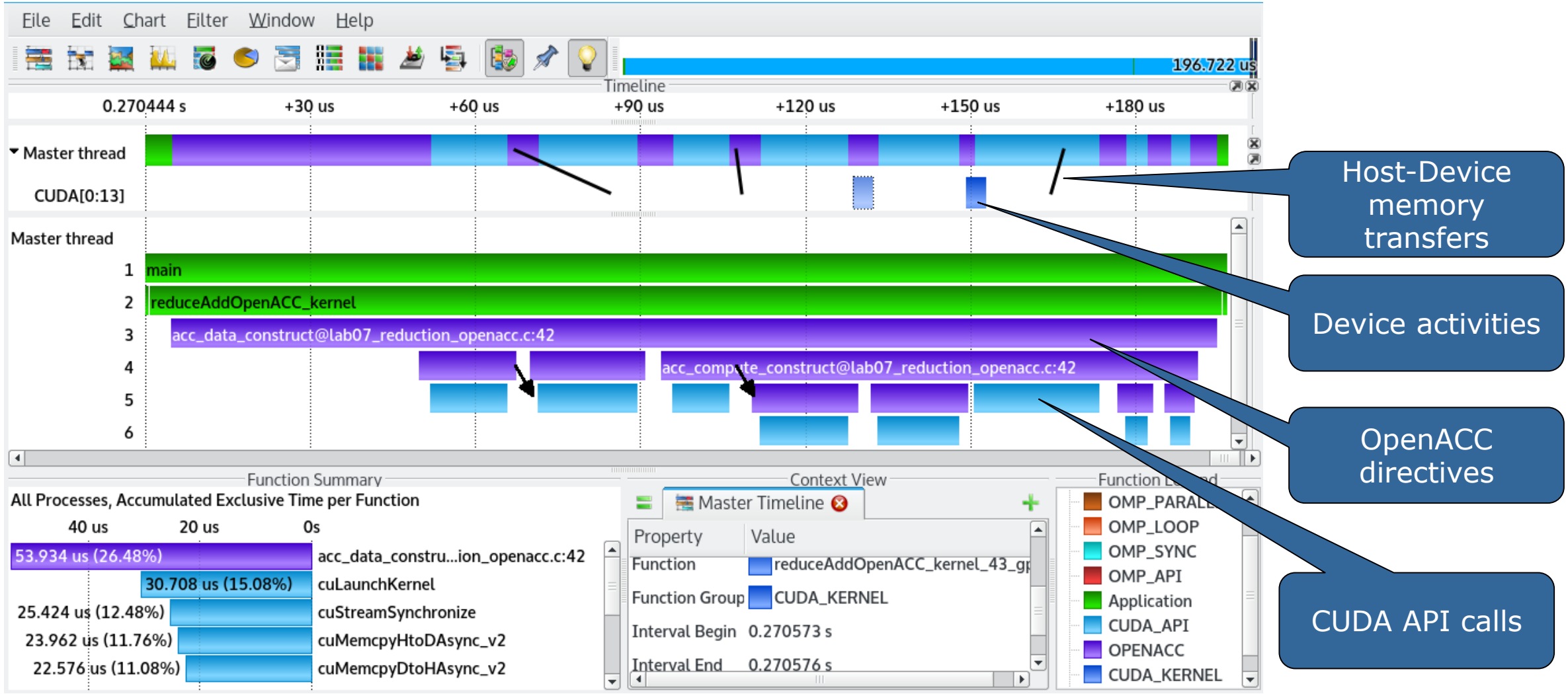
- Record OpenACC applications

```
% export SCOREP_OPENACC_ENABLE=yes
```

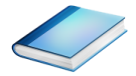
  - Can be combined with CUDA if it is a NVIDIA device

```
% export SCOREP_CUDA_ENABLE=kernel
```

Mastering accelerators



Hybrid measurement with sampling

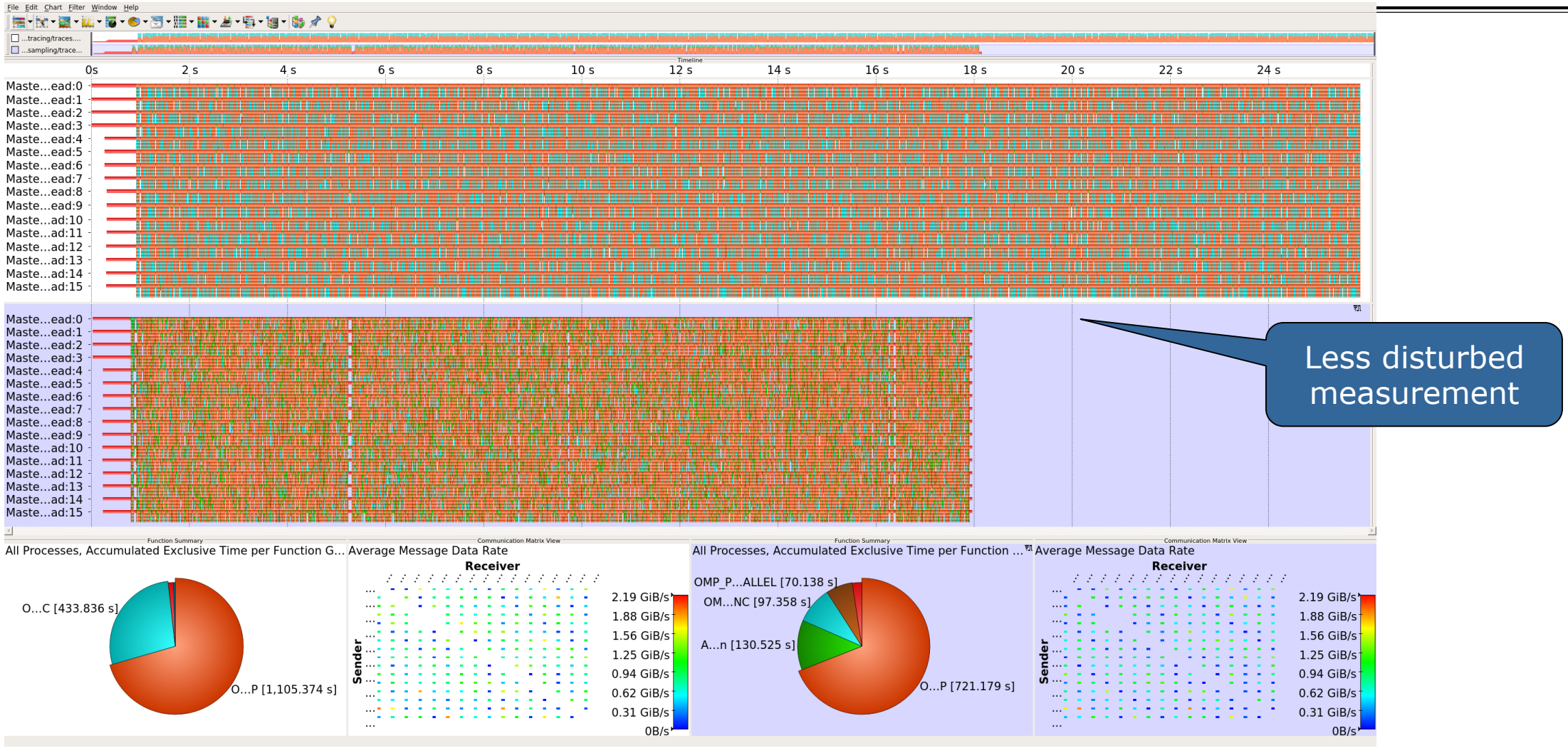


- Automatic compiler instrumentation greatly disturbs C++ applications because of frequent/short function calls ⇒ Use sampling instead
- Novel combination of sampling events and instrumentation of MPI, OpenMP, …
  - Sampling replaces only compiler instrumentation (use `--nocompiler`)
  - Instrumentation is still used for parallel activities (MPI, OpenMP, CUDA, I/O)
- Supports profile and trace generation

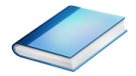
```
% export SCOREP_ENABLE_UNWINDING=true
% # use the default sampling frequency
% #export SCOREP_SAMPLING_EVENTS=perf_cycles@2000000
```

- Set new configuration variable to enable sampling

Mastering C++ applications



Instrumenting calls to 3rd party libraries



- Score-P does not record function calls to non-instrumented external libraries
- Increase insight into the behavior of the application
  - How does the application use the external library?
  - How does this compares to the usage of other libraries?
- Manual user instrumentation of the application using the library should be avoided
- Vendor provided libraries cannot be instrumented, but API is provided in headers

Instrumenting calls to 3rd party libraries: Library wrapper generator

- Workflow to generate library wrappers for most C/C++ libraries
- Tailored towards users of the external library, not users of Score-P
- Results can be shared by multiple users
- Workflow driver `scorep-libwrap-init --help` provides instructions

Only once

Library wrapper creator

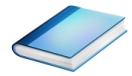
1. Initialize the working directory
2. Add library headers
3. Create a simple example application
4. Further configure the build parameters
5. Build the wrapper
6. Verify the wrapper
7. Install the wrapper
8. Verify the installed wrapper

Library wrapper user

9. Use the wrapper

Often

Instrumenting calls to 3rd party libraries: Workflow



- Start workflow 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
```

Omit to install into Score-P

Flags used to compile/link

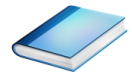
Working directory can be archived for later rebuild

- 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.
```

Tells you how to use the wrapper with Score-P

Instrumenting calls to 3rd party libraries: Usage and result



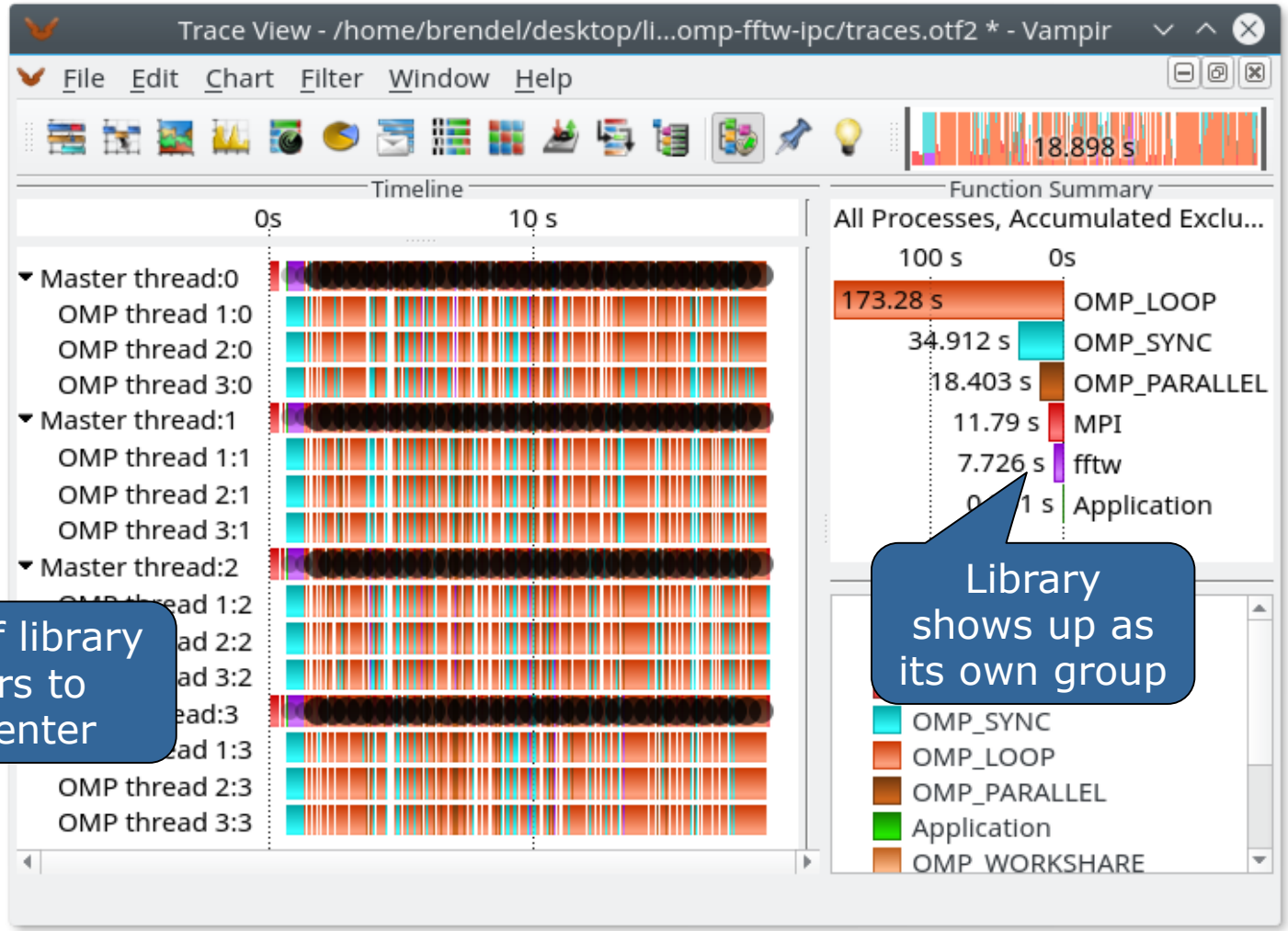
- List of available wrappers:

```
% scorep-info libwrap-summary
```

- Instrumentation:

```
% cd <application>
% export \
SCOREP_WRAPPER_INSTRUMENTER_FLAGS=\
    --libwrap=fftw
% make clean
% make
# run application as usual
```

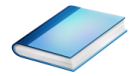
Pass list of library wrappers to instrumenter



Library shows up as its own group

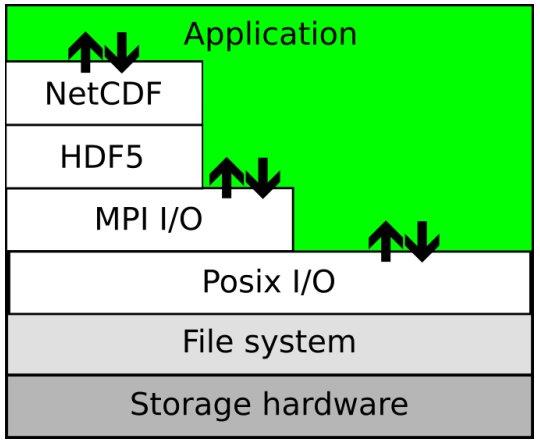
- MPI + OpenMP
- Calls to FFTW library

Mastering File I/O



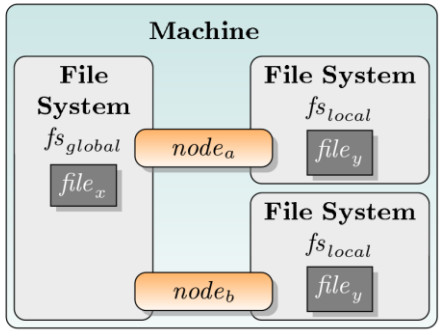
- Omnipresent in todays HPC applications

- Record interaction between multiple layers
  - MPI I/O (`MPI_File_open`)
  - ISO C I/O (`fopen`)
  - POSIX I/O (`open`, interface to OS)

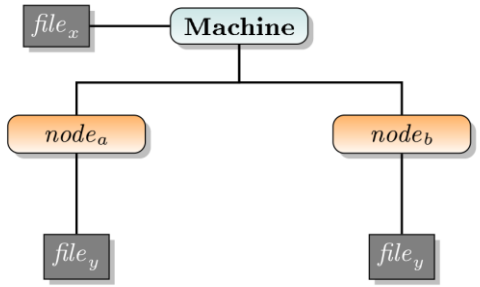


- System tree information determine whether file resides in a shared file system

- High level of detail
  => Trace data might increase dramatically



(a) Hardware topology



(b) System tree representation

NetCDF & HDF5 will be supported later

Mastering File I/O: B_EFF I/O hands-on

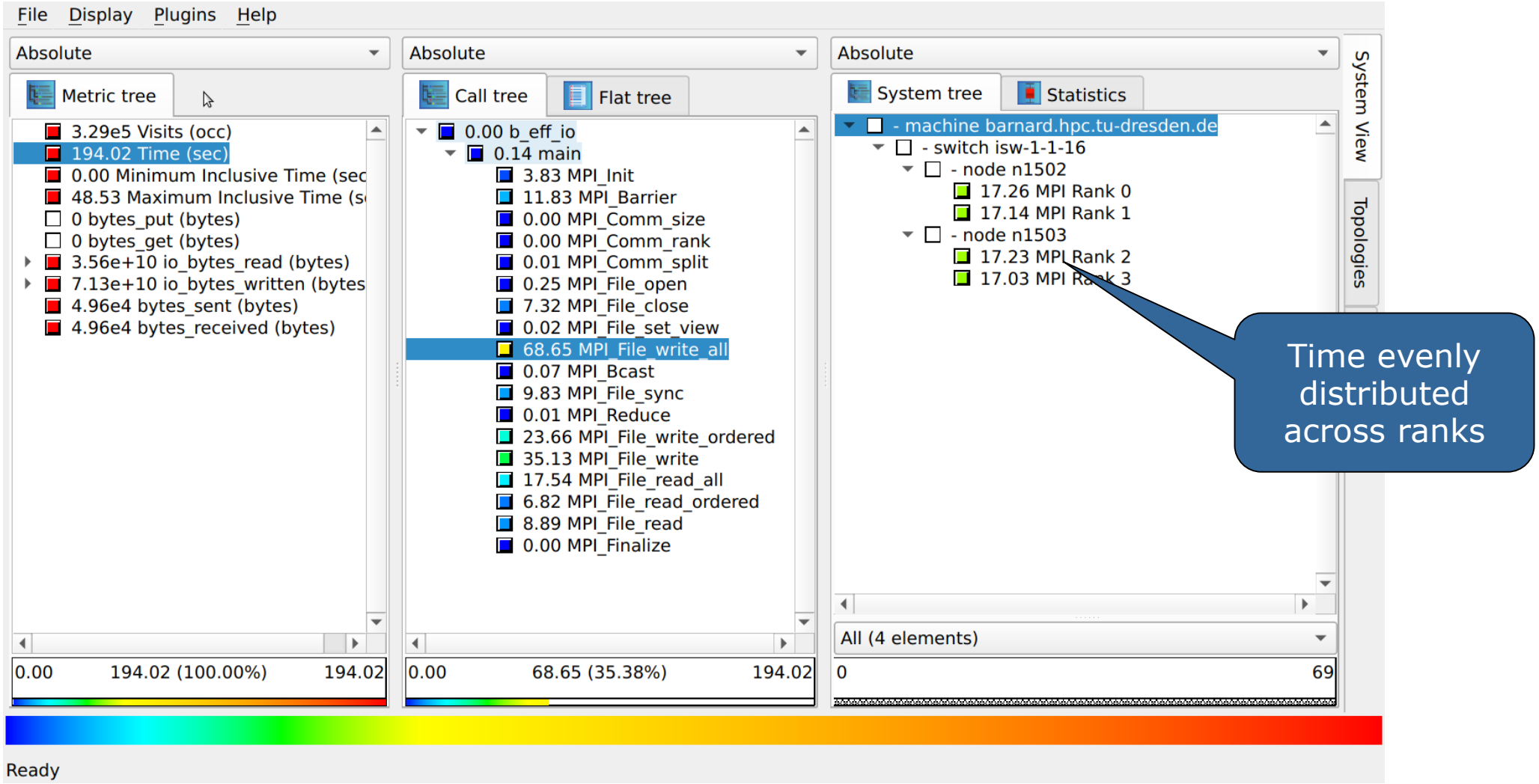
- Setup MPI I/O hands-on
  fs.hlrs.de/projects/par/mpi//b_eff_io/

```
% cd $VIHPS_WORKSPACE/hands-on
% tar xf $VIHPS_ROOT/hands-on/score-p-io.tar.gz
% cd score-p-io
```

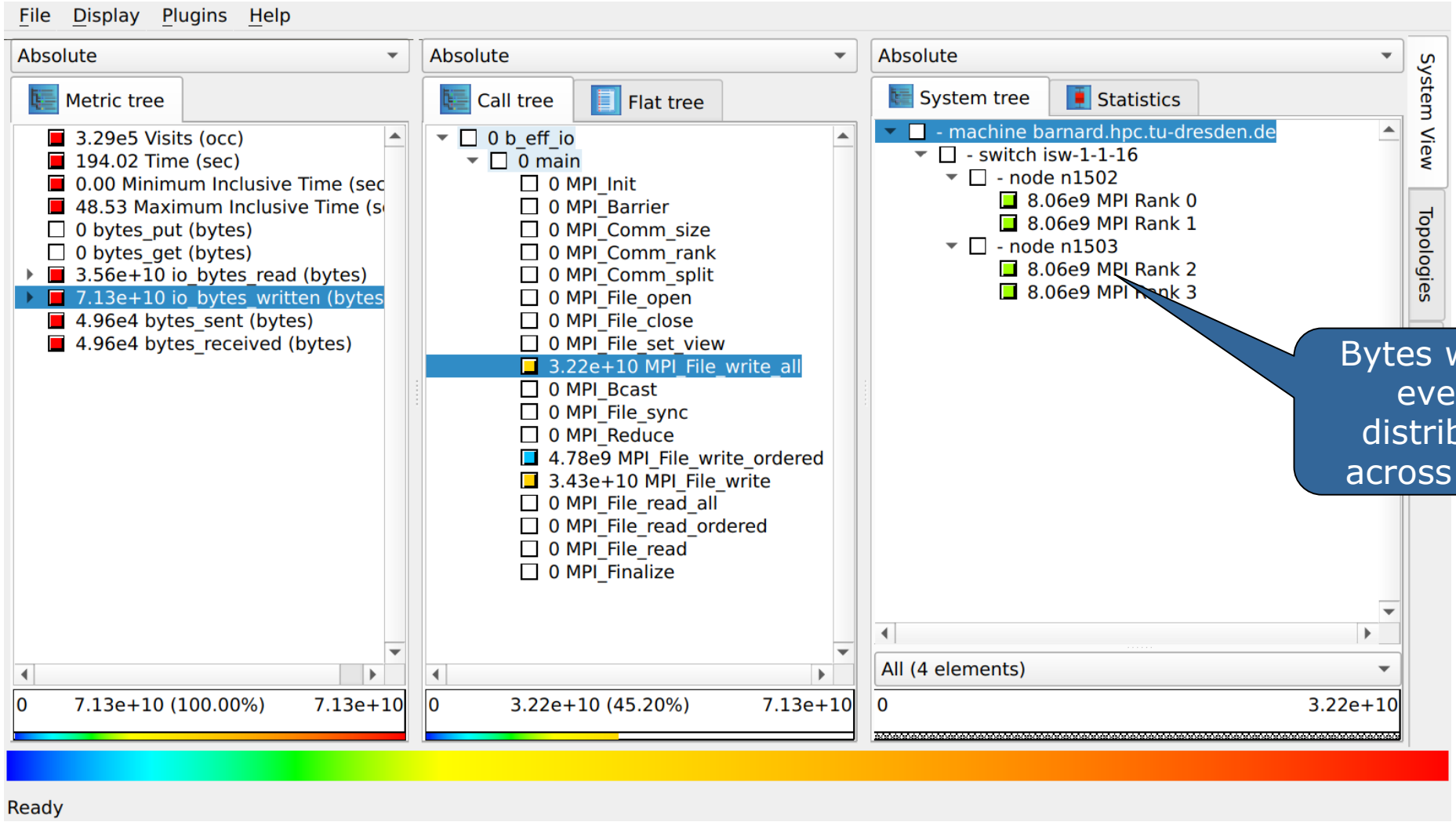
- MPI I/O is enabled by default

```
% scorep-mpiicc -g -O3 -o b_eff_io b_eff_io.c
% export SCOREP_EXPERIMENT_DIRECTORY=scorep-b_eff_io-4-profile
% srun -A $SBATCH_ACCOUNT --reservation $SBATCH_RESERVATION \
      -n 4 --ntasks-per-socket=1 -c 6 ./run.sh b_eff_io
% cube scorep-b_eff_io-4-profile/profile.cubex
```

Mastering File I/O: Result visualization



Mastering File I/O: Result visualization



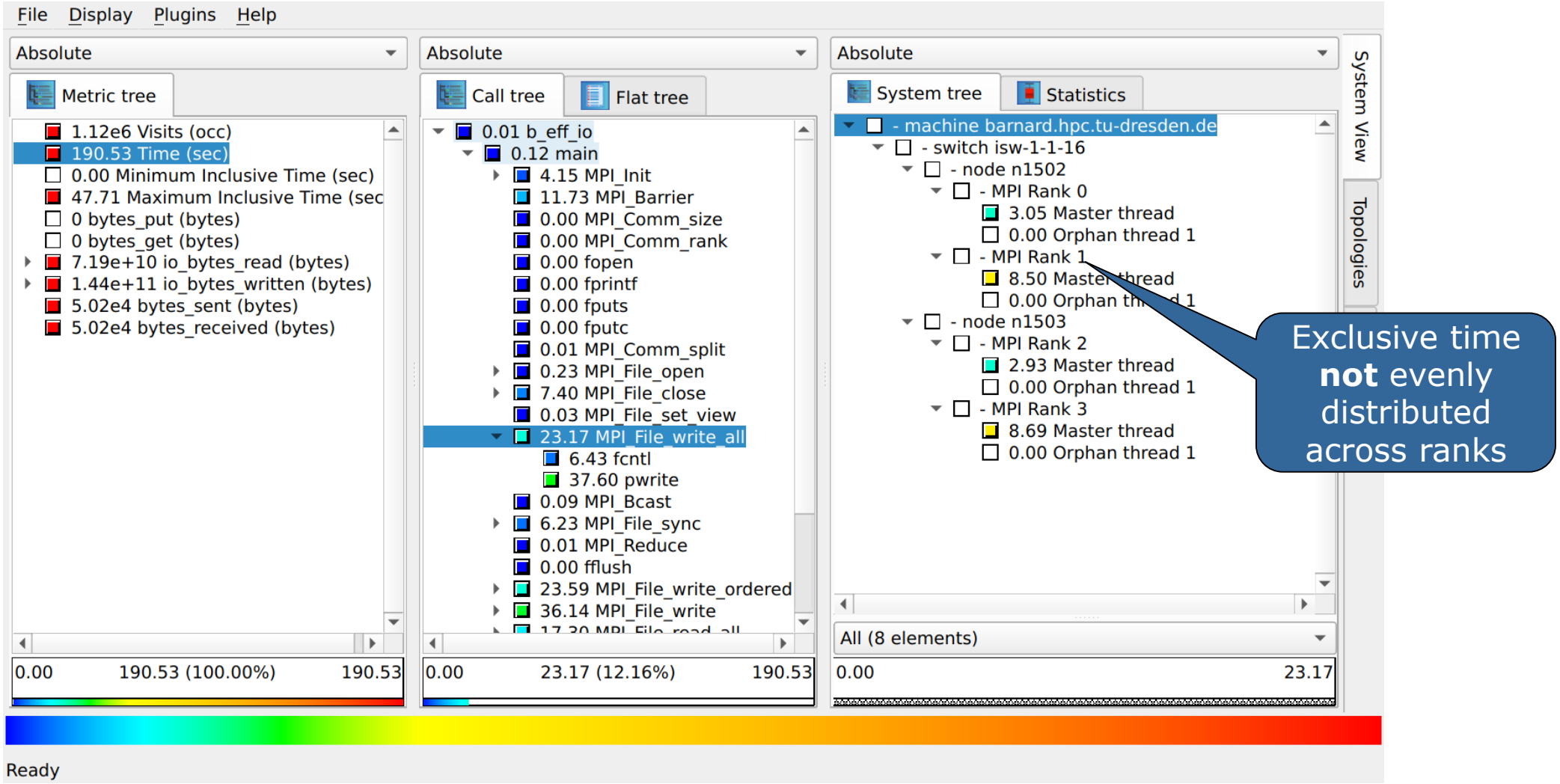
Mastering File I/O: B_EFF I/O hands-on

- Enabling ISO C and POSIX I/O instrumentation

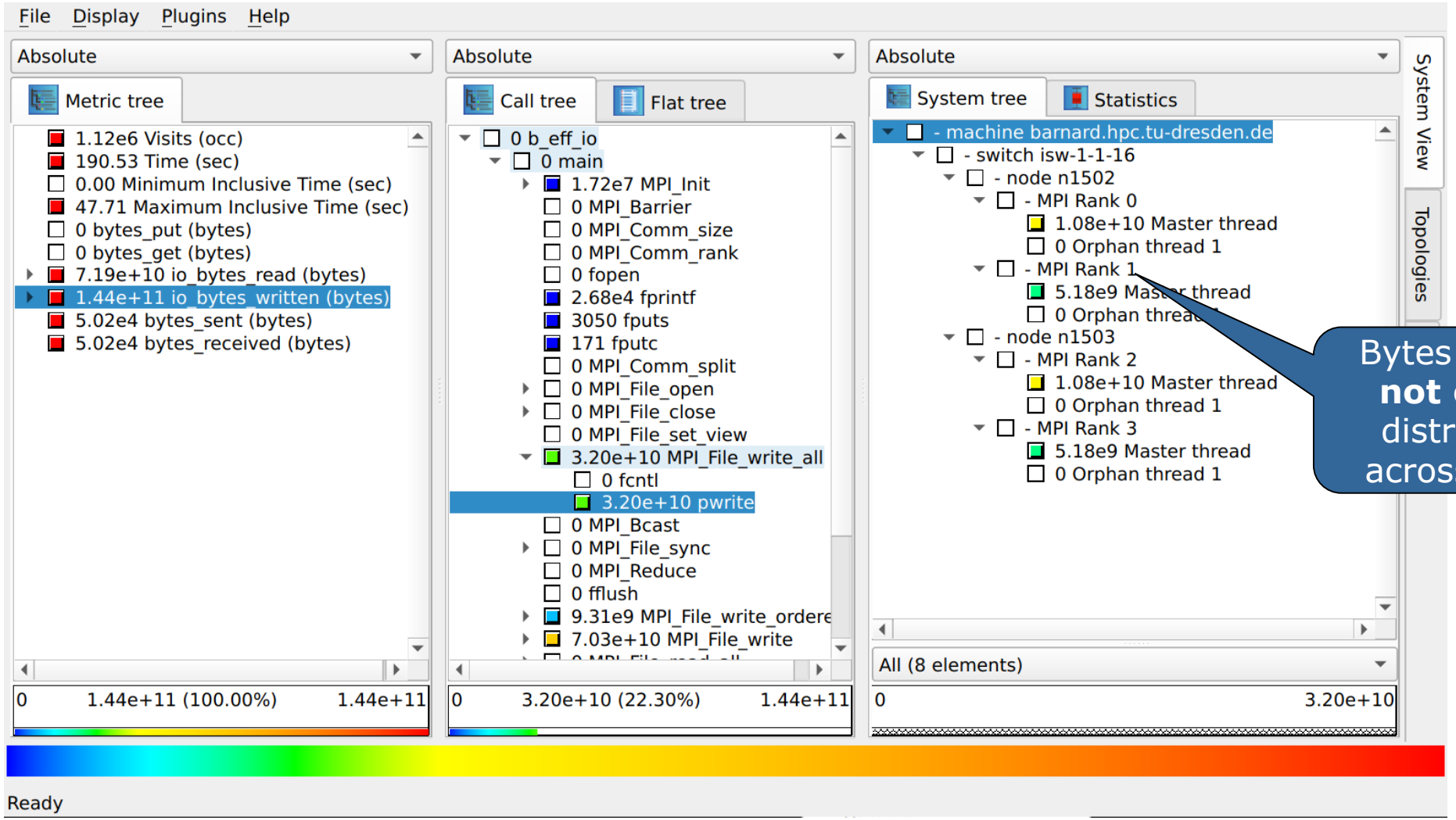
- Instrumentation does require threading support

```
% export SCOREP_WRAPPER_INSTRUMENTER_FLAGS=\
    '--io=runtime:posix --thread=pthread'
% scorep mpiicc -g -O3 -o b_eff_io b_eff_io.c
% export SCOREP_EXPERIMENT_DIRECTORY=scorep-b_eff_io-4-profile+posix
% srun -A $SBATCH_ACCOUNT --reservation $SBATCH_RESERVATION \
      -n 4 --ntasks-per-socket=1 -c 6 ./run.sh b_eff_io
% cube scorep-b_eff_io-4-profile+posix/profile.cubex
```

Mastering File I/O: Result visualization



Mastering File I/O: Result visualization



Mastering Python

- Install Score-P Python bindings via

```
pip install --user scorep
```

- Execute Score-P as a module via

```
python -m scorep …
```

- Score-P environment variables are still valid and needed

```
#!/bin/env bash

export SCOREP_EXPERIMENT_DIRECTORY=…

export SCOREP_FILTERING_FILE=…

python3 –m scorep <path/to/my_script.py>
```

Fig.: Vampir – module import

Mastering Python: Score-P options

- Accepts additional switches `-m scorep <options>`
- These include
  - `--thread=pthread`
  - `--mpp=mpi`
  - `--io=runtime:posix`
  - `--cuda` / `--hip`

```
#!/bin/env bash

python3 –m scorep --io=runtime:posix <path/to/my_script.py>
```

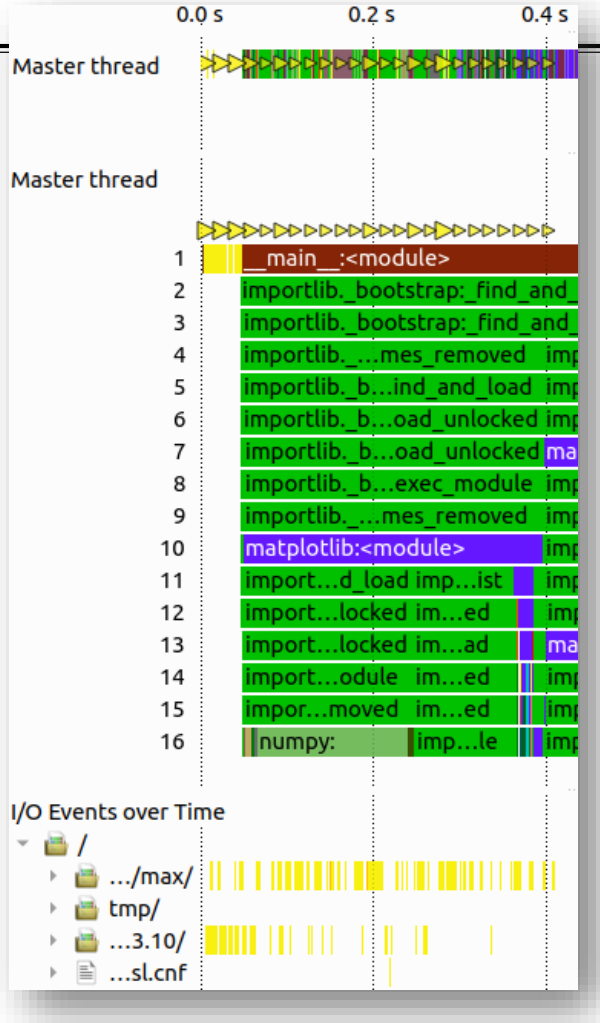


Fig.: Trace with POSIX-I/O

Mastering Python: Serial hands-on

- Setup hands-on

```
% cd $VIHPS_WORKSPACE/hands-on
% tar xf $VIHPS_ROOT/hands-on/score-p-python.tar.gz
% cd score-p-python
```

- Load Python and install modules

```
% ml Python
% pip install --user mpi4py numpy scorep
…
% python3 -c 'import mpi4py, numpy, scorep'
```

- Run simple serial code

```
% export SCOREP_EXPERIMENT_DIRECTORY=scorep-python-serial
% srun -A $SBATCH_ACCOUNT --reservation $SBATCH_RESERVATION \
      -n 1 python3 -m scorep ./01_serial.py
% cube scorep-python-serial/profile.cubex
```

Mastering Python: Selective instrumentation

- Enable/Disable instrumentation for certain functions
- Via decorator or context

```
import scorep

@scorep.instrumenter.enable()/disable()

with scorep.instrumenter.enable()/disable():
```

- [No] data is collected for these sections
- Disabling is useful for
  - Import of modules
  - Known uninteresting functions
  - Very 'hot' functions to reduce overhead
- Hint: name 'disabled' blocks for attribution in output

```
%%file 02_matmul.py
import scorep
with scorep.instrumenter.disable():
    import numpy as np
…

@scorep.instrumenter.disable()
def add(a,b)
    return a + b

def compute_diagonal_sum(A):
    sum_diag = 0
    for i in range(min(A.shape[0], A.shape[1])):
         sum_diag = add(sum_diag,A[i][i])
    return sum_diag

def main():
    A = generate_random_matrix(64, 64)
    diag_sum = compute_diagonal_sum(A)
…
```

Mastering Python: Selective instrumentation

- Selective enabling of functions
- If only certain region is of interest
- Make sure to execute Score-P with

`--noinstrumenter`

```
#!/bin/env bash
…
python3 –m scorep --noinstrumenter \
     02_matmul.py
```

```
%%file 02_matmul.py
import scorep
import numpy as np
…
def add(a,b)
    return a + b
def compute_diagonal_sum(A):
    sum_diag = 0
    for i in range(min(A.shape[0], A.shape[1])):
         sum_diag = add(sum_diag,A[i][i])
    return sum_diag

def main():
    A = generate_random_matrix(64, 64)
    with scorep.instrumenter.enable():
        diag_sum = compute_diagonal_sum(A)
…
```

Mastering Python: Selective instrumentation hands-on

- Run application with

```
% export SCOREP_EXPERIMENT_DIRECTORY=scorep-python-matmul
% srun -A $SBATCH_ACCOUNT --reservation $SBATCH_RESERVATION \
      -n 1 python3 -m scorep ./02_matmul.py
% cube scorep-python-matmul/profile.cubex
```

- Use selective instrumentation

```
% <editor> 02_matmul.py
# comment-in "scorep" statements and "fix" indentation
% export SCOREP_EXPERIMENT_DIRECTORY=scorep-python-matmul+selective
% srun -A $SBATCH_ACCOUNT --reservation $SBATCH_RESERVATION \
      -n 1 python3 -m scorep ./02_matmul.py
% cube scorep-python-matmul+selective/profile.cubex
```

Mastering Python: MPI

- MPI must be enabled explicitly via `--mpp=mpi`

```
srun –np 4 python3 –m scorep --mpp=mpi <path/to/my_script.py>
```

- Caveats
  - No support for `MPI_Mprobe/Mrecv`, default for `mpi4py` -> disable usage
  - No support for `MPI_THREAD_MULTIPLE`, default for `mpi4py` -> disable usage

```
#!/bin/env bash

# disables the use of MPI_Mprobe & MPI_Mrecv
export MPI4PY_RC_RECV_MPROBE=False

# Promise the MPI library (and Score-P) that the funneled mode is
# used at most
export MPI4PY_RC_THREAD_LEVEL=funneled

srun –np 4 python3 –m scorep --mpp=mpi <path/to/my_script.py>
```

Mastering Python: MPI hands-on

- Run application with

```
% export SCOREP_EXPERIMENT_DIRECTORY=scorep-python-mpi
% export MPI4PY_RC_RECV_MPROBE=False
% export MPI4PY_RC_THREAD_LEVEL=funneled
% srun -A $SBATCH_ACCOUNT --reservation $SBATCH_RESERVATION \
      -n 4 python3 -m scorep --mpp=mpi ./03_mpi.py
% cube scorep-python-mpi/profile.cubex
```

Mastering Python: Accelerators

- Score-P can be used to instrument accelerators (GPUs)
- `SCOREP_[CUDA|HIP]_ENABLE=yes` use default set of features

```
...
export SCOREP_CUDA_ENABLE=yes
export SCOREP_CUDA_BUFFER=1M # Increase if necessary

python3 -m scorep --cuda <path/to/my_script.py>
```

```
...
export SCOREP_HIP_ENABLE=yes
export SCOREP_HIP_ACTIVITY_BUFFER_SIZE=1M # Increase if necessary

python3 -m scorep --hip <path/to/my_script.py>
```

Further information

- Community instrumentation & measurement infrastructure
  - Instrumentation (various methods) and sampling
  - Basic and advanced profile generation
  - Event trace recording
  - Online access to profiling data
- Available under 3-clause BSD open-source license
- Documentation & Sources:
  - `https://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