Score-P – A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir

VI-HPS Team

Score-P
Scalable performance measurement infrastructure for parallel codes

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

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

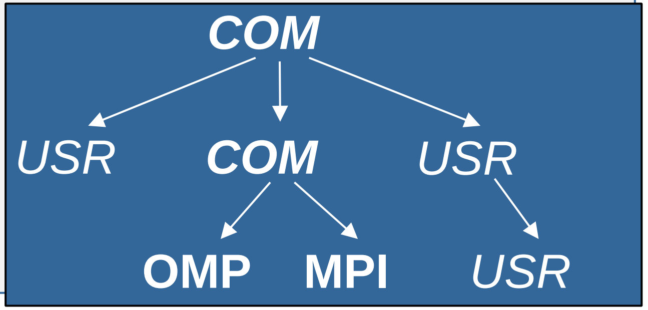
BT-MZ summary analysis result scoring

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

Estimated aggregate size of event trace:                   160 GB
Estimated requirements for largest trace buffer (max_buf):   6 GB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):         6 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 5,421,104,056 6,586,922,497 8268.20   100.0           1.26  ALL
       USR 5,407,570,350 6,574,832,225 3350.21    40.5           0.51  USR
       OMP    15,783,372    10,975,232 4094.63    49.5         373.08  OMP
       MPI       944,200       386,560  803.98     9.7        2079.83  MPI
       COM       665,210       728,480   19.38     0.2          26.60  COM
```

160 GB total memory
6 GB per rank!



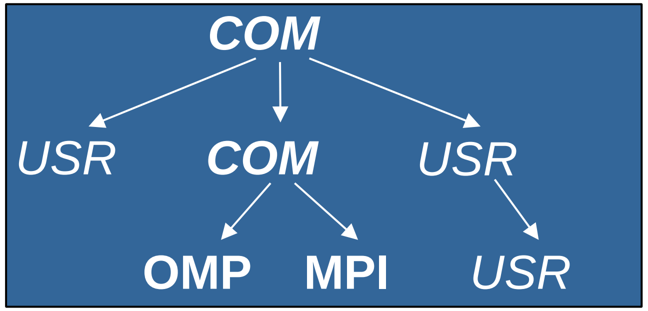
- Report scoring as textual output

- 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 5,421,104,056 6,586,922,497 8268.20   100.0           1.26  ALL
       USR 5,407,570,350 6,574,832,225 3350.21    40.5           0.51  USR
       OMP    15,783,372    10,975,232 4094.63    49.5         373.08  OMP
       MPI       944,200       386,560  803.98     9.7        2079.83  MPI
       COM       665,210       728,480   19.38     0.2          26.60  COM

       USR 1,741,005,318 2,110,313,472  850.28    10.3           0.40  matvec_sub_
       USR 1,741,005,318 2,110,313,472 1309.20    15.8           0.62  binvcrhs_
       USR 1,741,005,318 2,110,313,472 1064.72    12.9           0.50  matmul_sub_
       USR    76,367,538    87,475,200   53.39     0.6           0.61  lhsinit_
       USR    76,367,538    87,475,200   47.67     0.6           0.54  binvrhs_
       USR    56,913,688    68,892,672   24.90     0.3           0.36  exact_solution_
```



More than
5.4 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 ~6 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 -c 2 \
      scorep_bt-mz_sum/profile.cubex

Estimated aggregate size of event trace:                   473MB
Estimated requirements for largest trace buffer (max_buf): 17MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       25MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=25MB to avoid
intermediate flushes or reduce requirements using USR regions
filters.)
```

- Report scoring with prospective filter listing 6 USR regions

0.5 GB of memory in total, 25 MB per rank!

(Including 2 metric values)

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 5,421,104,056 6,586,922,497 8268.20   100.0           1.26  ALL
 -     USR 5,407,570,350 6,574,832,225 3350.21    40.5           0.51  USR
 -     OMP    15,783,372    10,975,232 4094.63    49.5         373.08  OMP
 -     MPI       944,200       386,560  803.98     9.7        2079.83  MPI
 -     COM       665,210       728,480   19.38     0.2          26.60  COM

 *     ALL    17,390,726    12,138,209 4918.03    59.5         405.17  ALL-FLT
 +     FLT 5,407,531,376 6,574,784,288 3350.17    40.5           0.51  FLT
 -     OMP    15,783,372    10,975,232 4094.63    49.5         373.08  OMP-FLT
 -     MPI       944,200       386,560  803.98     9.7        2079.83  MPI-FLT
 *     COM       665,210       728,480   19.38     0.2          26.60  COM-FLT
 *     USR        38,974        47,937    0.04     0.0           0.93  USR-FLT

 +     USR 1,741,005,318 2,110,313,472  850.28    10.3           0.40  matvec_s
 +     USR 1,741,005,318 2,110,313,472 1309.20    15.8           0.62  binvcrhs
 +     USR 1,741,005,318 2,110,313,472 1064.72    12.9           0.50  matmul_s
 +     USR    76,367,538    87,475,200   53.39     0.6           0.61  lhsinit_
 +     USR    76,367,538    87,475,200   47.67     0.6           0.54  binvrhs_
 +     USR    56,913,688    68,892,672   24.90     0.3           0.36  exact_so
 -     OMP     1,259,064       411,648    0.57     0.0           1.38  !$omp pa
```

Filtered routines marked with '+'

- Score report breakdown by region

BT-MZ filtered summary measurement

```
% cd bin.scorep
% vi scorep.sbatch

…

export SCOREP_EXPERIMENT_DIRECTORY=scorep_bt-mz_sum_filter
export SCOREP_TIMER='gettimeofday'
export SCOREP_FILTERING_FILE=../config/scorep.filt
#export SCOREP_TOTAL_MEMORY=25M
#export SCOREP_METRIC_PAPI=PAPI_TOT_INS,PAPI_TOT_CYC
#export SCOREP_ENABLE_TRACING=true

# run the application
mpiexec $EXE

% sbatch scorep.sbatch
```

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

- Submit job

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 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

```
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
```

Further information

- Community instrumentation & measurement infrastructure
  - Instrumentation (various methods)
  - Basic and advanced profile generation
  - Event trace recording
  - Online access to profiling data
- Available under New 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

Score-P:
Specialized Measurements and Analyses

Score-P

Mastering build systems

- Hooking up the Score-P instrumenter `scorep` into complex build environments like *Autotools* or *CMake* was always challenging
- Score-P provides new convenience wrapper scripts to simplify this (since Score-P 2.0)
- *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*

- Example: using Score-P wrapper in the configure step of a *CMake* project

```
% 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

Mastering build systems

- Using the wrapper in the build step

```
% scorep-icc COMPILER_FLAGS ...
```

will expand to the following call

```
% scorep $SCOREP_WRAPPER_INSTRUMENTER_FLAGS \
    gcc $SCOREP_WRAPPER_COMPILER_FLAGS \
    COMPILER_FLAGS ...
```

- Example

```
% make SCOREP_WRAPPER_INSTRUMENTER_FLAGS=--verbose
```

will result in the execution of

```
% scorep --verbose icc ...
```

Enable verbose output of instrumentation in the *build step*

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

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

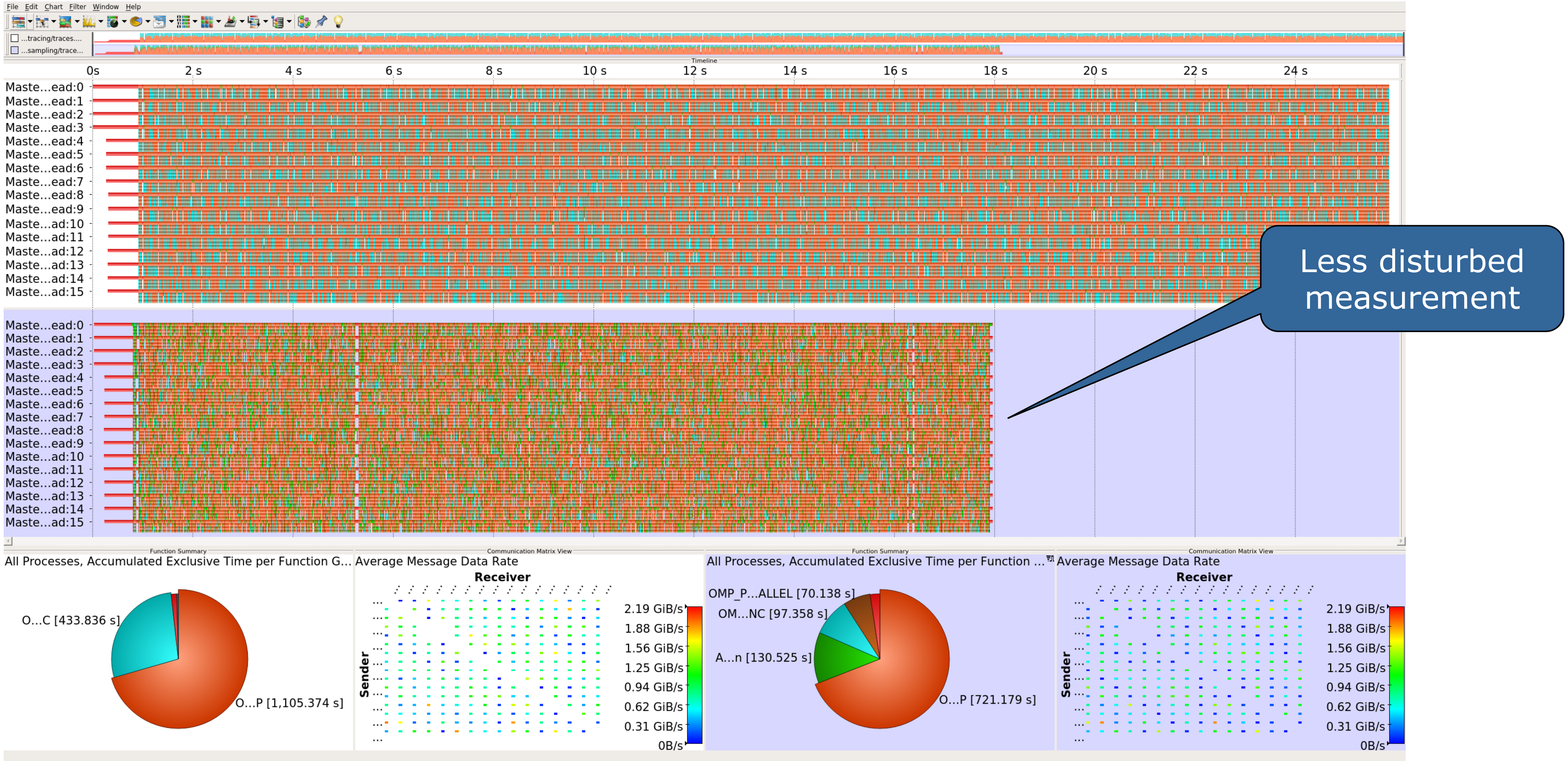
```
% export SCOREP_ENABLE_UNWINDING=true
% # use the default sampling frequency
% #export SCOREP_SAMPLING_EVENTS=perf_cycles@2000000

% 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



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

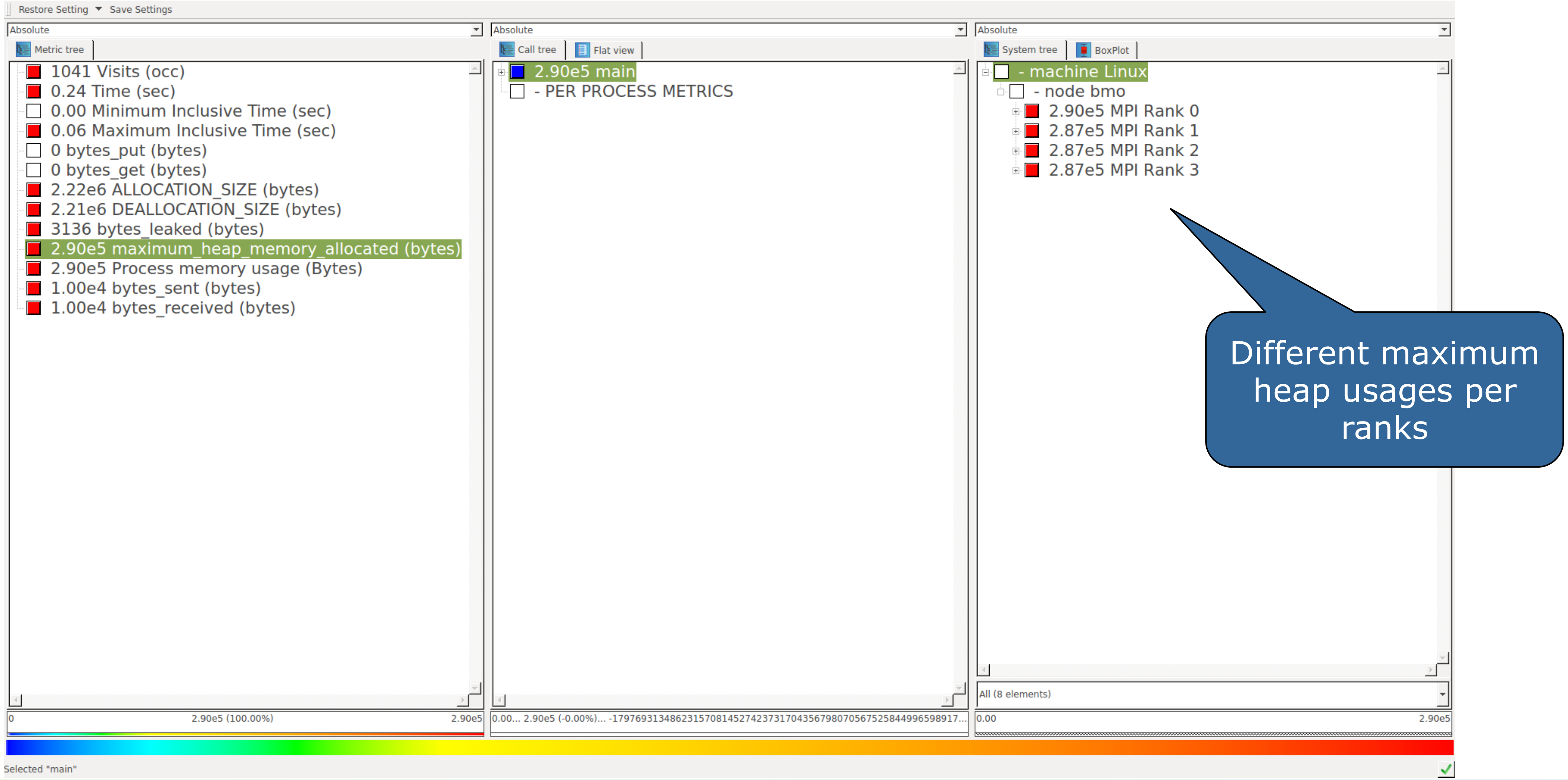
```
% export SCOREP_MEMORY_RECORDING=true
% export SCOREP_MPI_MEMORY_RECORDING=true

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

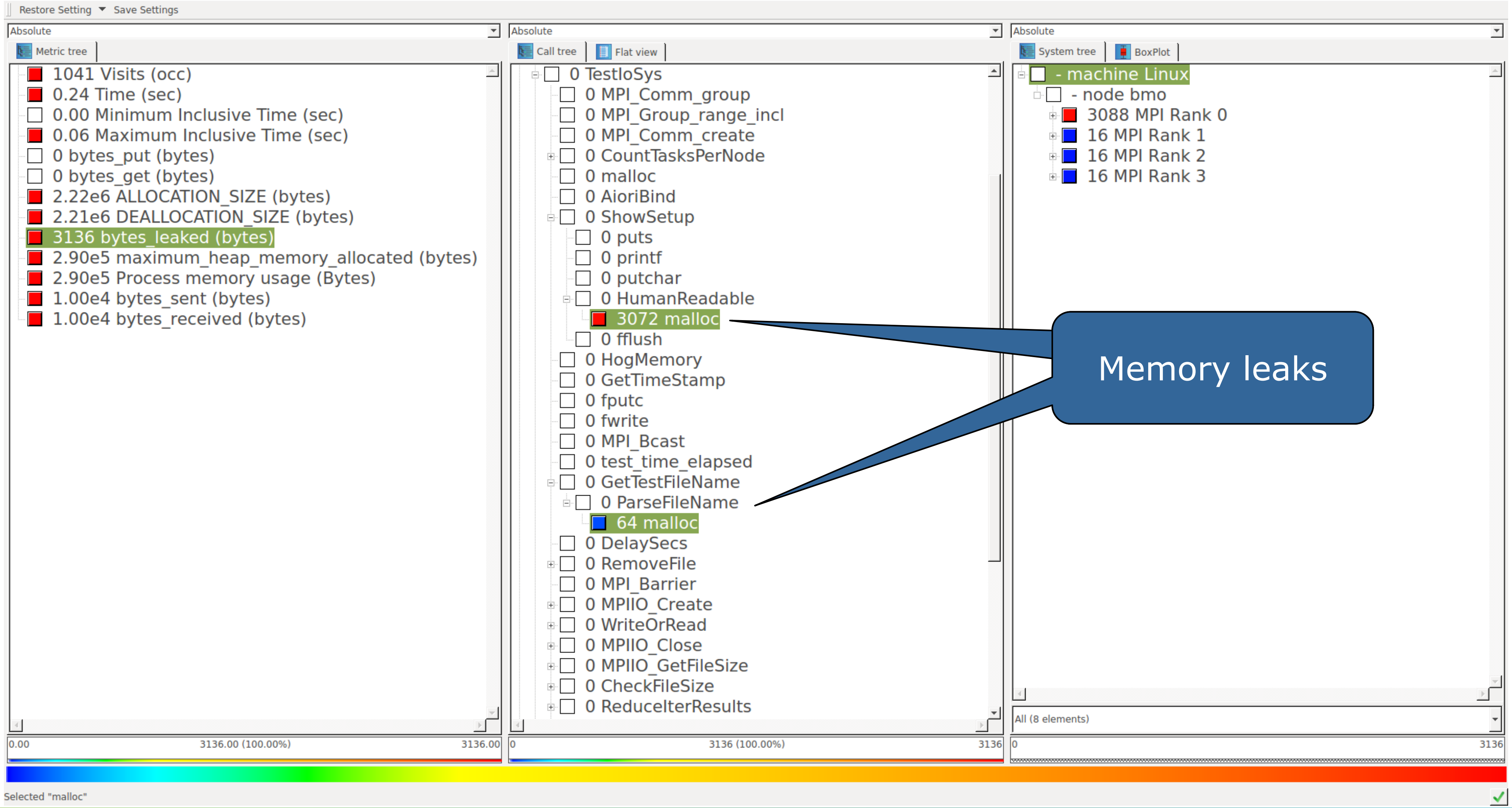
- Set new configuration variable to enable memory recording

- Available since Score-P 2.0

Mastering application memory usage



Mastering application memory usage



Mastering heterogeneous applications

- 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
```

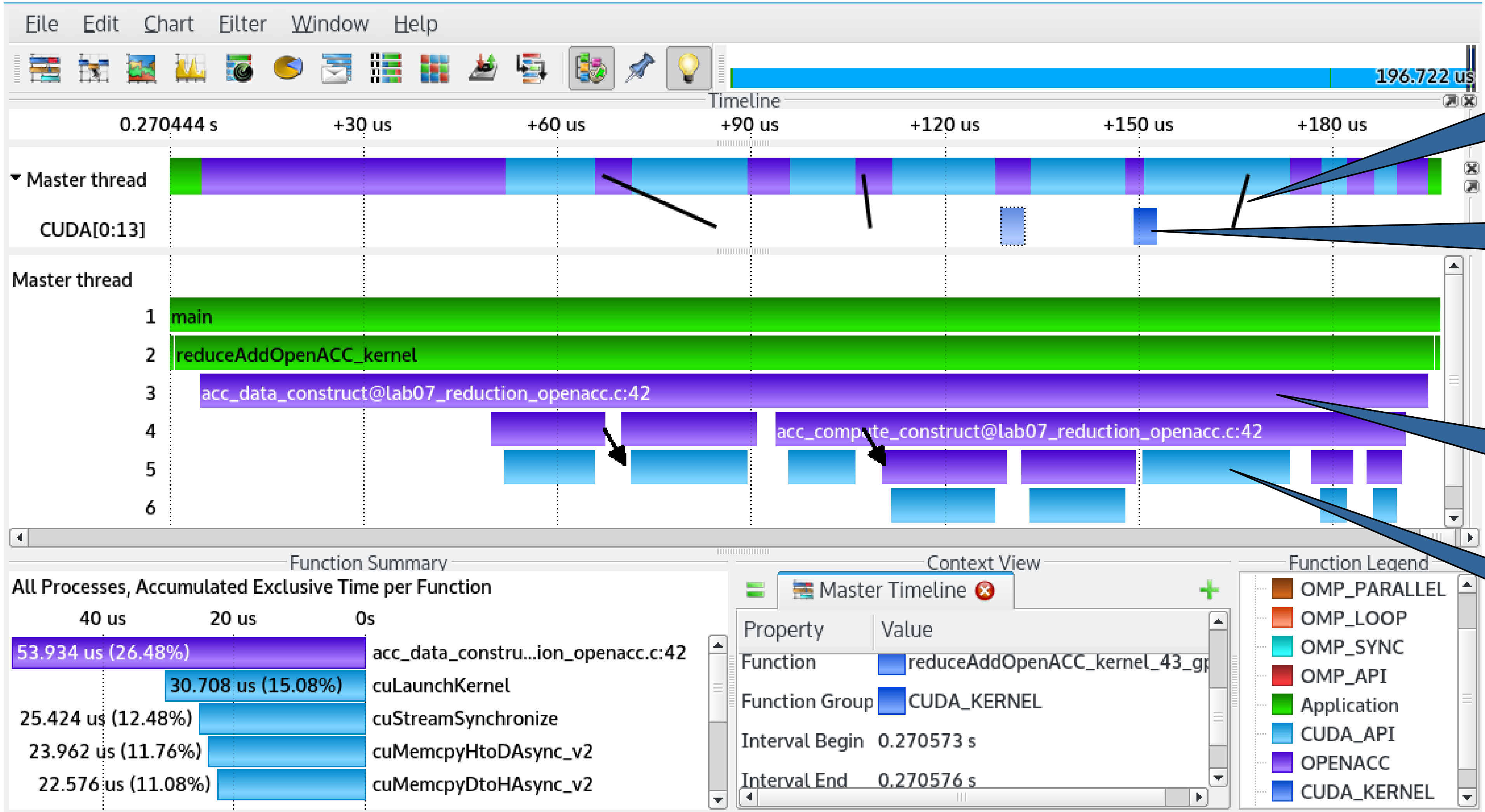
- Record OpenACC applications

```
% export SCOREP_OPENACC_ENABLE=yes
```

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

```
% export SCOREP_CUDA_ENABLE=kernel
```

Mastering heterogeneous applications



Host-Device memory transfers

Device activities

OpenACC directives

CUDA API calls

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)

Score-P user instrumentation API

- 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++

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++

Score-P:
Conclusion and Outlook

Project management

- Ensure a single official release version at all times which will always work with the tools
- Allow experimental versions for new features or research
- Commitment to joint long-term cooperation
  - Development based on meritocratic governance model
  - Open for contributions and new partners

Future features

- Scalability to maximum available CPU core count

- Support for emerging architectures and new programming models

- Features currently worked on:
  - User provided wrappers to 3rd party libraries
  - Hardware and MPI topologies
  - Basic support of measurements without re-compiling/-linking
  - I/O recording
  - Java recording
  - Persistent memory recording (e.g., PMEM, NVRAM, …)