SCALASCA v2.6 Quick Reference

General

- SCALASCA is an open-source toolset for scalable performance analysis of large-scale parallel applications.
- SCALASCA uses the community measurement system SCORE-P to generate profiles and tracing results.
- Use the **`scorep`** and **`scalasca`** commands with appropriate action flags to *instrument* application object files and executables, *analyze* execution measurements, and interactively *examine* measurement/analysis experiment archives.
- For short usage explanations, use SCALASCA commands without arguments, or add '**`-v`**' for verbose commentary.
- View possible parameters for the SCORE-P instrumenter by calling **`scorep --help`**

SCORE-P Instrumentation

- Prepend **`scorep`** and any instrumentation flags to your compile/link commands. Alternatively, use the SCORE-P compiler wrappers.
- Additional options to the instrumenter must be specified *before* the compiler/linker command.
- By default, MPI and OpenMP operations are automatically instrumented, and many compilers are also able to instrument all routines found in source files (unless explicitly disabled with **`--nocompiler`**).
- To enable manual instrumentation (described on page 4) using the user instrumentation API, add **`--user`**, and/or using POMP directives, add **`--pomp`**.
- To enable automatic source-to-source function instrumentation with PDToolkit, use **`--pdt`**.
- To enable CUDA instrumentation use **`--cuda`**.
- Examples:

Original command:	SCORE-P instrumentation command:
`mpicc -c foo.c`	**`scorep`** `mpicc -c foo.c`
`mpicxx -o foo foo.cpp`	**`scorep --user`** `mpicxx -o foo foo.cpp`
`mpif90 -openmp -o bar bar.f90`	**`scorep`** `mpif90 -openmp -o bar bar.f90`

- Often it is preferable to prefix Makefile compile/link commands with `$(PREP)` and set `PREP="scorep"` for instrumented builds (leaving `PREP` unset for uninstrumented builds).

Measurement & Analysis

- Prepend **`scalasca -analyze`** (or ***`scan`***) to the usual execution command line to perform a measurement with SCALASCA runtime summarization and associated automatic trace analysis (if applicable).
- SCORE-P instrumented applications can be run without prefix, using only environment variables for control. In this case the trace analysis won't be started automatically.
- To reuse an existing measurement for analysis, add the flag **`-a`**.
- Each measurement is stored in a new experiment archive which is not overwritten by a subsequent measurement.
- By default, only a runtime summary (profile) is collected (equivalent to specifying **`-s`**).
- To enable trace collection & analysis, add the flag **`-t`**.
- An archive directory name can be explicitly specified with **`scan -e`** ***`title`***.
- To analyze MPI and hybrid OpenMP+MPI applications, use the usual MPI launcher command and arguments.
- To analyze serial and (pure) OpenMP applications, omit the MPI launcher command.
- Examples:

Original command:	SCALASCA measurement & analysis command:	Experiment archive:
`mpiexec -np 4 foo args`	**`scalasca -analyze`** `mpiexec -np 4 foo args`	`# scorep_foo_4_sum`
`OMP_NUM_THREADS=3 bar`	`OMP_NUM_THREADS=3` **`scan -t`** `bar`	`# scorep_bar_Ox3_trace`
`mpiexec -np 4 foobar`	**`scan -s`** `mpiexec -np 4 foobar`	`# scorep_foobar_4x3_sum`
		`# (w/ 3 OpenMP threads)`

Measurement configuration

Measurement is controlled by a number of variables which can be set through corresponding environment variables: the configuration is stored in the experiment archive as `scorep.cfg`. The most important variables are:

Variable	Purpose	SCORE-P default
`SCOREP_EXPERIMENT_DIRECTORY`	Experiment archive title, explicitly specified by `-e` or automatically given a reasonable name if not specified.	`scorep-`*timestamp*
`SCOREP_ENABLE_PROFILING`	Enabling or disabling profile generation.	true
`SCOREP_ENABLE_TRACING`	Enabling or disabling trace generation.	false
`SCOREP_FILTERING_FILE`	Name of file containing a specification of functions which should be ignored during measurement.	—
`SCOREP_VERBOSE`	Controls generation of additional (debugging) output by measurement system.	false
`SCOREP_TOTAL_MEMORY`	Size of per-process memory reserved for SCORE-P in bytes.	16 384 000

The full list of supported configuration variables and their values can be retrieved using `scorep-info config-vars --full`.

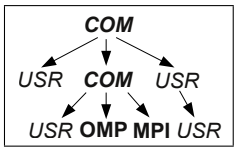
SCORE-P experiment archives – typical directory content

File	Description
`profile.cubex`	Analysis report of runtime summarization.
`scorep.cfg`	Measurement configuration when the experiment was collected.
`scorep.log`	Output of the instrumented program and measurement system.
`scout.cubex`	Intermediate analysis report of the parallel trace analyzer.
`scout.log`	Output of the parallel trace analyzer.
`summary.cubex`	Post-processed analysis report of runtime summarization. (May include HWC metrics.)
`trace.cubex`	Post-processed analysis report of the trace analyzer. (Does not include HWC metrics.)
`trace.stat`	Most-severe pattern instances and pattern statistics.
`traces/`	Sub-directory containing event traces for each process/thread.
`traces.def/.otf2`	Definitions and anchor files for the event traces.

Determining trace buffer capacity requirements

Based on an analysis report, the required trace buffer capacity can be estimated using

`scorep-score [-r] [-f` *filter_file*`]` *experiment_archive*`/profile.cubex`



- To get detailed information per region (i.e., function or subroutine), use `-r`
- To take a proposed filter file into account, use `-f` *filter_file*
- The report specifies the maximum estimated required memory per process, which can be used to set `SCOREP_TOTAL_MEMORY` appropriately to avoid intermediate flushes in subsequent tracing experiments.

The SCORE-P filter format allows to include and exclude functions from measurement using their names with possible use of wildcards. The respective commands are processed in sequence, allowing for hierarchical inclusion-exclusion schemes. To the right is an example for a standard filter file.

```
SCOREP_REGION_NAMES_BEGIN
  EXCLUDE
    binvcrhs*
    matmul_sub*
SCOREP_REGION_NAMES_END
```

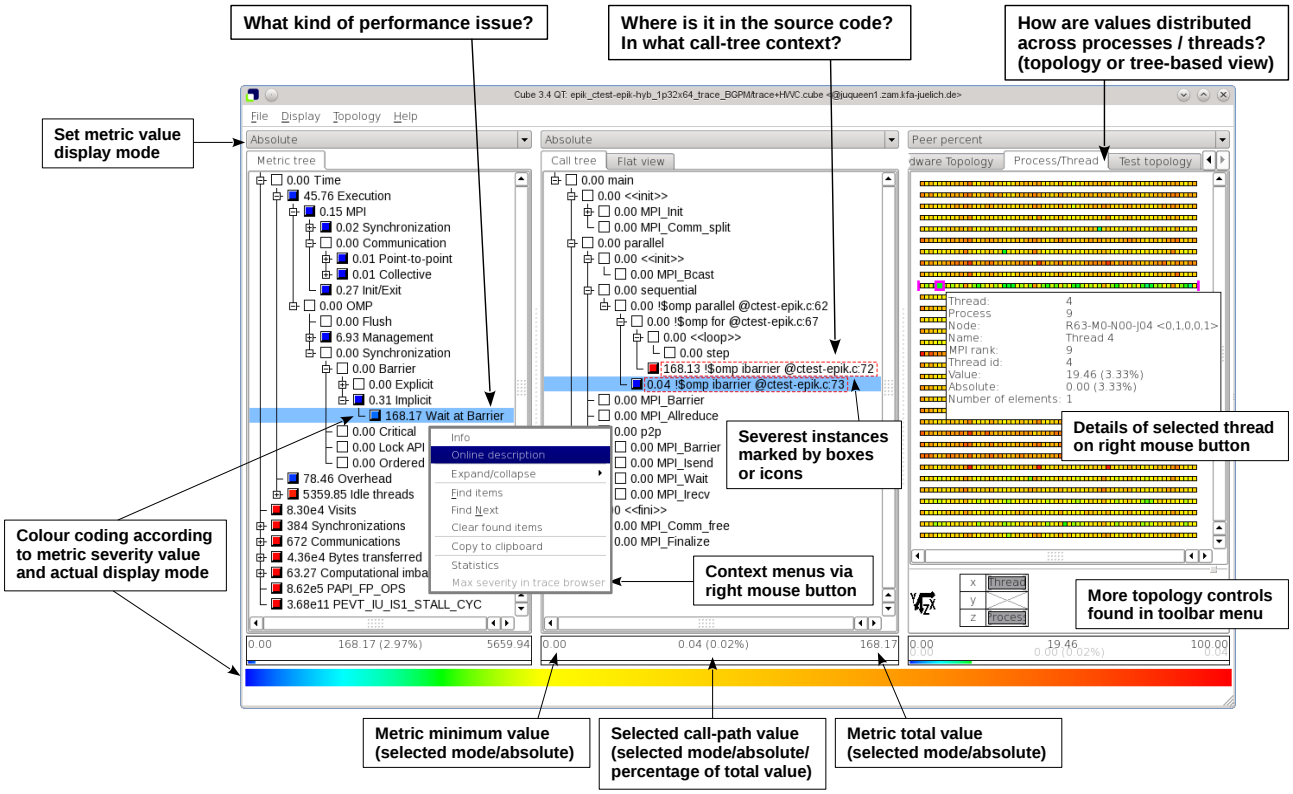
CUBE4 algebra and utilities

Uniform behavioural encoding, processing and interactive examination of parallel application execution analysis reports.

- CUBE4 provides a variety of utilities for differencing, combining and other operations on analysis reports, e.g., `cube_diff`, `cube_mean`, `cube_merge`.
- `cube_cut` can be used to prune uninteresting call-trees and/or re-root with a specified call-tree node.
- `cube_stat` can be used to produce custom statistical reports in CSV or plain text format.
- `cube_topoassist` can be used to add or modify topology specifications.

Analysis Report Examination

- To interactively examine the contents of a SCALASCA experiment, after final processing of runtime summary and trace analysis, use **`scalasca -examine`** (or ***`square`***) with the experiment archive directory name as argument.
- To skip the graphical user interface and get a textual score output (using the `scorep-score` utility), add the **`-s`** flag.
- If multiple analysis reports are available, a trace analysis report is shown in preference to a runtime summary report: other reports can be specified directly or selected from the File/Open menu.
- Results are displayed using three coupled tree browsers showing
  - Metrics (i.e., performance properties/problems)
  - Call-tree or flat region profile
  - System location (alternative: graphical displays, such as box/violin plot, Sunburst view, or physical/virtual Cartesian topologies. Topologies with more than 3 dimensions will be folded).



- Analyses are presented in trees, where collapsed nodes represent *inclusive* values (consisting of the value of the node itself and all of its child nodes), which can be selectively expanded to reveal *exclusive* values (i.e., the node 'self' value) and child nodes.
- When a node is selected from any tree, its *severity* value (and percentage) are shown in the panel below it, and that value distributed across the tree(s) to the right of it.
- Selective expansion of critical nodes, guided by the color scale, can be used to hone in on performance problems.
- Each tree browser provides additional information via a context menu (on the right mouse button), such as the description of the selected metric or source code for the selected region (where available).
- Metric severity values can be displayed in various modes:

Mode	Description
Absolute	Absolute value in the corresponding unit of measurement.
Root percent	Percentage relative to the inclusive value of the root node of the corresponding hierarchy.
Selection percent	Percentage relative to the value of selected node in corresponding tree browser to the left.
Peer percent	Percentage relative to the maximum of all peer values (all values of the current leaf level).
Peer distribution	Percentage relative to the maximum and non-zero minimum of all peer values.
External percent	Similar to "Root percent," but reference values are taken from another experiment.

Manual source-code instrumentation

- Region or phase annotations manually inserted in source files can augment or substitute automatic instrumentation, and can improve the structure of analysis reports to make them more readily comprehensible.
- These annotations can be used to mark any sequence or block of statements, such as functions, phases, loop nests, etc., and can be nested, provided that *every enter has a matching exit*.
- If automatic compiler instrumentation is not used (or not available), it is typically desirable to manually instrument at least the `main` function/program and perhaps its major phases (e.g., initialization, core/body, finalization).

SCORE-P user instrumentation API

C/C++:

```
#include <scorep/SCOREP_User.h>
...
void foo() {
  ...  // local declarations
  SCOREP_USER_FUNC_BEGIN();
  ...  // executable statements
  if (...)  {
   SCOREP_USER_FUNC_END();
   return;
  } else {
   SCOREP_USER_REGION_DEFINE(r_name);
   SCOREP_USER_REGION_BEGIN(r_name, "bar",
          SCOREP_USER_REGION_TYPE_COMMON);
   ...
   SCOREP_USER_REGION_END(r_name);
  }
  ...  // executable statements
  SCOREP_USER_FUNC_END();
}
```

Fortran:

```
#include <scorep/SCOREP_User.inc>
...
subroutine foo
  ...  !  local declarations
  SCOREP_USER_FUNC_DEFINE()
  SCOREP_USER_REGION_DEFINE(r_name)
  SCOREP_USER_FUNC_BEGIN("foo")
  ...  !  executable statements
  if (...)  then
   SCOREP_USER_FUNC_END()
   return
  else
  SCOREP_USER_REGION_BEGIN(r_name, "bar",
         SCOREP_USER_REGION_TYPE_COMMON)
  ...
   SCOREP_USER_REGION_END(r_name)
  end if
  SCOREP_USER_FUNC_END()
end subroutine foo
```

- `SCOREP_USER_FUNC_BEGIN` and `SCOREP_USER_FUNC_END` are provided explicitly to mark the entry and exit(s) of functions/subroutines.
- Function names are automatically provided by C/C++, however, in annotated Fortran functions/subroutines an appropriate name should be registered with `SCOREP_USER_FUNC_BEGIN("func_name")`.
- Region identifiers (e.g., `r_name`) should be registered with `SCOREP_USER_REGION_DEFINE` in each annotated prologue before use with `SCOREP_USER_REGION_BEGIN` and `SCOREP_USER_REGION_END` in the associated body.
- Every exit/break/continue/return/etc. out of each annotated region must have corresponding `_END()` annotation(s).
- Source files annotated in this way need to be compiled with the `--user` flag given to the SCORE-P instrumenter, otherwise the annotations are ignored. Fortran source files need to be preprocessed (e.g., by FPP or CPP).

POMP user instrumentation API

POMP annotations provide a mechanism for preprocessors (such as OPARI2) to conditionally insert user instrumentation.

C/C++:

```
#pragma pomp inst init // once only, in main
...
#pragma pomp inst begin(name)
   ...
   [ #pragma pomp inst altend(name) ]
   ...
#pragma pomp inst end(name)
```

Fortran:

```
!POMP$ INST INIT ! once only, in main program
...
!POMP$ INST BEGIN(name)
   ...
   [ !POMP$ INST ALTEND(name) ]
   ...
!POMP$ INST END(name)
```

- Every intermediate exit/break/return/etc. from each annotated region must have an `altend` or `ALTEND` annotation.
- Source files annotated in this way need to be processed with the `--pomp` flag given to the SCORE-P instrumenter, otherwise the annotations are ignored.

Tips for effective use of the SCALASCA toolset

1. Determine one or more repeatable execution configurations (input data, number of processes/threads) and time their overall execution to have a baseline for reference. (If possible, also identify maximum memory requirements.)
   - Ensure that the execution terminates cleanly, e.g., with `MPI_Finalize` and not calling `STOP` or `exit(`*val*`)`.
   - Excessively long execution durations can make measurement and analysis inconvenient, therefore the test configuration shouldn't be longer than sufficient to be representative.
2. Modify the application build procedure (e.g., Makefile) to prepend the SCORE-P instrumenter to compile and link commands, and produce an instrumented executable.
   - MPI library calls and OpenMP parallel regions will be instrumented by default, along with user functions if supported by the compiler.
   - Serial libraries and source modules using neither MPI nor OpenMP are generally not worth instrumenting with SCORE-P, and indeed may result in undesirable measurement overheads.
3. Prefix the usual launch/run command with the SCALASCA analyzer to run the instrumented executable under control of the SCALASCA measurement collection and analysis nexus to produce an experiment archive directory.
   - By default the experiment archive is produced in the current working directory, and its name will start with '`scorep_`' followed by some configuration descriptors if created automatically. The SCALASCA measurement & analysis nexus automatically generates a default experiment title from the target executable, compute node mode (if appropriate), number of MPI processes (or `0` if omitted), number of OpenMP threads (if `OMP_NUM_THREADS` is set), summarization or tracing mode, and optional metric specification.
   - If a similarly configured experiment has already been run and its archive directory blocks new measurement experiments.
   - If no path is given, e.g., in a run without SCALASCA, the name will start with '`scorep-`' followed by an unique identifier (timestamp). In this mode an existing archive is renamed with an unique suffix unless specified otherwise.
   - A call-path profile summary report containing Time and Visits metrics (and when appropriate also MPI file I/O and message statistics and hardware counters) for each process/thread is produced by default.
   - If the (default) measurement configuration is inadequate for a complete measurement to be collected, warnings will indicate that one or more configuration variables should be adjusted (e.g., `SCOREP_TOTAL_MEMORY`).
   - Compare the runtime to the (uninstrumented) reference to estimate implicit instrumentation dilation overhead.
4. Use the SCALASCA examiner to explore the analysis report in the experiment archive.
   - Uninstrumented or filtered routines will not appear in the analysis report, and their associated metric severities will be attributed to the last measured routine from which they are called (as if they were 'inlined').
   - Additional structure can be included in the analysis report by using the Score-P user instrumentation API to specify (nested) regions or phases as annotations in the source code.
5. Score the quality of the summary analysis report (particularly if dilation is significant), adjust measurement configuration using a filter file, adjust OpenMP instrumentation, or selectively instrument source modules (or routines).
   - Investigate use of a filter file specifying instrumented routines to be ignored during measurement collection.
   - Routines with very high visit counts and relatively low total times (which are not MPI functions and OpenMP parallel regions) are appropriate candidates for filtering, and can be identified from the flat profile view in the GUI, or score reports generated with '`scalasca -examine -s`' or using '`scorep_score -r`'.
   - Highly-recursive functions are typically also worth removing: recursion is often indicated by a large maximum call path depth.
   - A prospective filter file can be specified to scoring with '`-f`' for evaluation prior to being used to re-do measurement and re-check dilation.
   - Some routines might still present excessive overhead even when filtered, and these should not be instrumented. The build procedure may need to be adjusted not to prefix the SCORE-P instrumenter when compiling the associated source modules. When SCORE-P is configured with PDToolkit, it can be used to selectively instrument entire source modules or individual routines (see PDToolkit documentation for details).

6. Use scoring on the (revised/filtered) summary analysis report to determine an appropriate size for the SCORE-P memory settings.

   - For the estimated memory requirements per process, the `SCOREP_TOTAL_MEMORY` environment variable can be adjusted to avoid intermediate buffer flushes.
   - `SCOREP_TOTAL_MEMORY` should be set after consideration of the memory available when measuring the instrumented application execution and the system's I/O and filesystem performance and capacity. (These vary enormously from system to system and can quickly be overwhelmed by large traces!)
   - Additional user routines can be included in a filter file to reduce trace buffer requirements.

7. Repeat measurement specifying the '`-t`' flag to the SCALASCA analyzer (along with other configuration settings if necessary) to collect and automatically analyze execution traces.

   - Traces are generally written directly into the experiment archive to avoid copying at completion.
   - A filesystem capabable of efficient parallel file I/O should be used when available.
   - If there are SCORE-P messages reporting trace flushing to disk prior to closing the experiment, these intermediate flushes are often highly disruptive. Enlarging trace buffer sizes and/or adjusting instrumentation or the measurement filter and/or configuring a shorter execution (perhaps with fewer iterations or timesteps) may be appropriate.
   - Parallel trace analysis requires several times as much memory as the size of the respective (uncompressed process) traces, and it is currently not possible to analyze incomplete traces. When memory is restricted, trace sizes should be reduced accordingly.
   - If clock condition violations are reported during trace analysis, set the `SCAN_ANALYZE_OPTS` environment variable to `--time-correct` to incorporate a logical clock correction step during analysis.
   - Traces from hybrid OpenMP/MPI application executions are analyzed in parallel by default. If an OpenMP-aware trace analyzer is not available, metrics are only calculated for the master thread of OpenMP teams.
   - After the analysis report has been examined and verified to be complete, it is generally unnecessary to keep the often extremely large trace files used to generate it (unless further analysis or conversion is planned): these are in the `traces` subdirectory of a trace experiment archive, which can be deleted.

8. In addition to interactive exploration of analysis reports with the SCALASCA examiner, they can be processed with a variety of CUBE algebra tools and utilities.

9. If you encounter difficulties using SCORE-P or SCALASCA to instrument applications, configuring measurement collection and analysis, or interpreting analysis reports, contact `scalasca@fz-juelich.de` for assistance.