scalasca

Scalasca 2.6 | User Guide

Scalable Automatic Performance Analysis

April 2021
The Scalasca Development Team
scalasca@fz-juelich.de

JÜLICH
Forschungszentrum

The entire code of Scalasca v2 is licensed under the BSD-style license agreement given below, except for the third-party code distributed in the 'vendor/' subdirectory. Please see the corresponding COPYING files in the subdirectories of 'vendor/' included in the distribution tarball for details.

Scalasca v2 License Agreement

Contents

1 Introduction

Supercomputing is a key technology of modern science and engineering, indispensable to solve critical problems of high complexity. However, since the number of cores on modern supercomputers is increasing from generation to generation, HPC applications are required to harness much higher degrees of parallelism to satisfy their growing demand for computing power. Therefore—as a prerequisite for the productive use of today's large-scale computing systems—the HPC community needs powerful and robust performance analysis tools that make the optimization of parallel applications both more effective and more efficient.

The Scalasca Trace Tools developed at the Jülich Supercomputing Centre are a collection of trace-based performance analysis tools that have been specifically designed for use on large-scale systems featuring hundreds of thousands of CPU cores, but also suitable for smaller HPC platforms. A distinctive feature of the Scalasca Trace Tools is its scalable automatic trace-analysis component which provides the ability to identify wait states that occur, for example, as a result of unevenly distributed workloads [6]. Especially when trying to scale communication intensive applications to large process counts, such wait states can present severe challenges to achieving good performance. Besides merely identifying wait states, the trace analyzer is also able to pinpoint their root causes (i.e., delays) [3], and to identify the activities on the critical path of the target application [2], highlighting those routines which determine the length of the program execution and therefore constitute the best candidates for optimization.

The current focus of the Scalasca Trace Tools analyses is on applications using MPI [11], OpenMP [15], POSIX threads [7], or hybrid MPI+OpenMP/Pthreads parallelization. While traces from applications using CUDA [13], OpenCL [8], or OpenACC [14] parallelization can be handled as long as they do not contain any device activities (i.e., only host-side events have been measured), no specific support for those paradigms has been implemented yet. Thus, analysis results from such traces need to be interpreted with care, but may nevertheless provide useful insights. We intend to add better support for those accelerator programming models in the future.

Unlike previous versions of the Scalasca toolset—which used a custom measurement system and trace data format—the Scalasca Trace Tools 2.x release series is based on the community-driven instrumentation and measurement infrastructure Score-P [10]. The Score-P software is jointly developed by a consortium of partners from Germany and the US, and supports a number of complementary performance analysis tools through the use of the common data formats CUBE4 for profiles and the Open Trace Format 2 (OTF2) [5] for event trace data. This significantly improves interoperability between Scalasca and other performance analysis tool suites such as Vampir [9] and TAU [18]. Nevertheless, backward compatibility to Scalasca 1.x is maintained where possible. For example, the Scalasca trace analyzer is still able to process trace measurements generated by the measurement system of the Scalasca 1.x release series.

This user guide is intended to address the needs of users which are new to Scalasca as well as those already familiar with previous versions of the Scalasca toolset. For both user groups, it is recommended to work through Chapter 2 to get familiar with the intended Scalasca analysis workflow in general, and to learn about the changes compared to the Scalasca 1.x release series which are highlighted when appropriate. Later chapters then provide more in-depth reference information for the individual Scalasca commands and tools, and can be consulted when necessary.

2 Getting started

This chapter provides an introduction to the use of the Scalasca Trace Tools on the basis of the analysis of an example application. The most prominent features are addressed, and at times a reference to later chapters with more in-depth information on the corresponding topic is given.

Use of the Scalasca Trace Tools involves three phases: instrumentation of the target application, execution measurement collection and analysis, and examination of the analysis report. For instrumentation and measurement, the Scalasca Trace Tools 2.x release series leverages the Score-P infrastructure, while the Cube graphical user interface is used for analysis report examination. The Scalasca Trace Tools complement the functionality provided by Score-P and Cube with scalable automatic trace-analysis components, as well as convenience commands for controlling execution measurement collection and analysis, and analysis report post-processing.

Most of Scalasca's functionality can be accessed through the `scalasca` command, which provides action options that in turn invoke the corresponding underlying commands `scorep`, `scan` and `square`. These actions are:

1. `scalasca -instrument`

   (or short `skin`) familiar to users of the Scalasca 1.x series is **deprecated** and only provided for backward compatibility. It tries to map the command-line options of the Scalasca 1.x instrumenter onto corresponding options of Score-P's instrumenter command `scorep`—as far as this is possible. However, to take full advantage of the functionality provided by Score-P, **users are strongly encouraged to use the `scorep` instrumenter command directly**. To assist in transitioning existing measurement configurations to Score-P, the Scalasca instrumentation wrapper prints the converted command that is actually executed to standard output.

2. `scalasca -analyze`

   (or short `scan`) is used to control the Score-P measurement environment during the execution of the target application—supporting both runtime summarization and/or event trace collection, optionally including hardware-counter information—and to automatically initiate Scalasca's trace analysis after measurement completion if tracing was requested.

3. `scalasca -examine`

   (or short `square`) is used to post-process the analysis report generated by a Score-P profiling measurement and/or Scalasca's automatic post-mortem trace analysis, and to start the Cube graphical user interface for analysis report examination.

To get a brief usage summary, call the `scalasca` command without arguments, or use `scalasca --quickref` to open the Scalasca Quick Reference (with a suitable PDF viewer). See also Section 3.1 for a complete reference of the `scalasca` command.

Note:

> Under the hood, the Scalasca convenience commands leverage a number of other commands provided by Score-P as well as the CubeLib and CubeGUI components. Therefore, it is generally advisable to include the executable directories of appropriate installations of all those components in the shell search path (`$PATH`).

The following three sections provide a quick overview of each of these actions and how to use them during the corresponding step of the performance analysis, before a tutorial-style full workflow example is presented in Section 2.4.

2.1 Instrumentation

To generate measurements which can be used as input for the Scalasca Trace Tools, user application programs first need to be instrumented. That is, special measurement calls have to be inserted into the program code which are then executed at specific important points (events) during the application run. Unlike previous versions of Scalasca which used a custom measurement system, this task is now accomplished by the community instrumentation and measurement infrastructure Score-P.

As already mentioned in the previous section, use of the `scalasca -instrument` and `skin` commands is discouraged, and therefore not discussed in this guide. Instead, all the necessary instrumentation of user routines, OpenMP constructs, MPI functions, etc. should be handled by the Score-P instrumenter, which is accessed through the `scorep` command. Therefore, the compile and link commands to build the application that is to be analyzed should be prefixed with `scorep` (e.g., in a Makefile).

For example, to instrument the MPI application executable `myapp` generated from the two Fortran source files `foo.f90` and `bar.f90`, the following compile and link commands

```
$ mpif90 -c foo.f90
$ mpif90 -c bar.f90
$ mpif90 -o myapp foo.o bar.o
```

have to be replaced by corresponding commands involving the Score-P instrumenter:

```
$ scorep  mpif90 -c foo.f90
$ scorep  mpif90 -c bar.f90
$ scorep  mpif90 -o myapp foo.o bar.o
```

This will automatically instrument every routine entry and exit seen by the compiler, intercept MPI function calls to gather message-passing information, and link the necessary Score-P measurement libraries to the application executable.

Alternatively, the Score-P compiler wrapper commands (e.g., `scorep-mpif90`) can be used as a *compiler replacement*. These commands allow to control instrumentation via an environment variable, which may be required during the configuration step with certain build systems such as CMake or GNU Autotools. Please refer to the Section "Score-P Compiler Wrapper Usage" in the Score-P User Manual [17] for details.

Attention:

> **The Score-P instrumenter commands (`scorep` prefix or compiler wrapper) must be used with the link command** to ensure that all required Score-P measurement libraries are linked with the executable. However, not all object files need to be instrumented, thereby avoiding measurements and data collection for routines and OpenMP constructs defined in those files. Nevertheless, for the Scalasca trace analysis to work correctly, **instrumenting files defining OpenMP parallel regions is essential**.

Although generally most convenient, automatic compiler-based function instrumentation as used by default may result in too many and/or too disruptive measurements, which can be addressed by selective instrumentation and measurement filtering. While the most basic steps will be briefly covered in Section 2.4.4, please also consult the Score-P manual [17] for details on the available instrumentation and filtering options.

2.2 Runtime measurement collection & analysis

While applications instrumented by Score-P can be executed directly with a measurement configuration defined via environment variables, the `scalasca -analyze` (or short `scan`) convenience command provided by Scalasca can be used to control certain aspects of the Score-P measurement environment during the execution of the target application. To produce a performance measurement using an instrumented executable, the target application execution command is prefixed with the `scalasca -analyze` (or short `scan`) command:

```
$ scalasca -analyze [options] \
           [<launch_cmd> [<launch_flags>]] <target> [<target_args>]
```

For pure MPI or hybrid MPI+OpenMP/Pthreads applications, *launch_cmd* is typically the MPI execution command such as `mpirun` or `mpiexec`, with *launch_flags* being the corresponding command-line arguments as used for uninstrumented runs, for example, to specify the number of compute nodes or MPI ranks. For non-MPI (i.e., serial and pure multi-threaded) applications, the launch command and flags can usually be omitted.

In case of the example MPI application executable `myapp` introduced in the previous section, a measurement command starting the application with four MPI ranks could therefore be:

```
$ scalasca -analyze  mpiexec -n 4 ./myapp
```

Attention:

> A unique directory is used for each measurement experiment, which (by default) must not already exist when measurement starts: otherwise measurement is aborted immediately.

A default name for the experiment directory is composed of the prefix "`scorep_`", the target application executable name, the run configuration (e.g., number of MPI ranks and/or OpenMP threads), and a few other parameters of the measurement configuration. For example, a measurement of the `myapp` application as outlined above will produce a measurement experiment directory named "`scorep_myapp_4_sum`".

Note:

A number of settings regarding the measurement configuration can be specified in different ways. See the scan command reference in Section 3.2 for details and available configuration options.

When measurement has completed, the measurement experiment directory contains various log files and one or more analysis reports. By default, runtime summarization is used to provide a summary report of the number of visits and time spent on each callpath by each process/thread, as well as hardware counter metrics (if configured). For MPI or hybrid MPI+OpenMP/Pthreads applications, MPI message statistics are also included.

Event trace data can also be collected as part of a measurement. This measurement mode can be enabled by passing the -t option to the scalasca -analyze command (or alternatively by setting the environment variable SCOREP_ENABLE_TRACING to either "1", "true", or "yes").

Note:

Enabling event trace collection does not automatically turn off summarization mode (i.e., both a summary profile and event traces are collected). It has to be explicitly disabled when this behavior is undesired.

When collecting a trace measurement, experiment trace analysis is automatically initiated after measurement is complete to quantify wait states that cannot be detected with runtime summarization, to determine their root causes, and to identify the critical path of the application. In addition to examining the trace-analysis report, the generated event traces can also be visualized with a third-party graphical trace browser such as Vampir [9].

Warning:

Traces can easily become extremely large and unwieldy, and uncoordinated intermediate trace buffer flushes may result in cascades of distortion, which renders such traces to be of little value. **It is therefore extremely important to set up an adequate measurement configuration *before* initiating trace collection and analysis!** Please see Section 2.4.4 as well as the Score-P User Manual [17] for more details on how to set up a filtering file and adjust Score-P's internal memory management.

2.3 Analysis report examination

The results of the runtime summarization and/or the automatic trace analysis are stored in one or more reports (i.e., CUBE4 files) in the measurement experiment directory. These reports can be post-processed and examined using the scalasca -examine (or short square) command, providing an experiment directory name as argument:

```
  $ scalasca -examine [options] <experiment_name>
```

Post-processing is performed leveraging commands provided by the CubeLib component the first time an experiment is examined, before launching the Cube analysis report browser (CubeGUI). If the scalasca -examine command is provided with an already processed experiment directory, or with a CUBE4 file specified as argument, the viewer is launched immediately.

Instead of interactively examining the measurement analysis results, a textual score report can also be obtained using the `-s` option (see Section 3.3 for further command-line options) without launching the viewer:

```
$ scalasca -examine -s <experiment_name>
```

This score report is generated by Score-P's `scorep-score` utility and provides a breakdown of the different types of regions included in the measurement and their estimated associated trace buffer capacity requirements, aggregate trace size, and largest process trace buffer size, which can be used to set up a filtering file and to determine an appropriate setting for `SCOREP_TOTAL_MEMORY` to be used for subsequent trace measurements. See Section 2.4.4 for more details.

The Cube viewer can also be directly used on an experiment archive—opening a dialog window to choose one of the contained CUBE4 files—or an individual CUBE4 file as shown below:

```
$ cube <experiment_name>
$ cube <file>.cubex
```

However, keep in mind that no post-processing is performed in this case, so that only a subset of Scalasca's analyses and metrics may be shown.

2.4 A full workflow example

While the previous sections introduced the three basic actions supported by the Scalasca Trace Tools based on an abstract example, this section will now guide through the typical analysis workflow using a moderately complex, MPI-based benchmark code: BT from the NAS Parallel Benchmarks (NPB-MPI 3.3.1) [12]. The BT benchmark implements a simulated computational fluid dynamics (CFD) application using a block-tridiagonal solver for a synthetic system of nonlinear partial differential equations and consists of about 20 Fortran 77 source code files. Although BT does not exhibit significant performance bottlenecks—after all, it is a highly optimized benchmark—it serves as a good example to demonstrate the overall workflow, including typical configuration steps and how to avoid common pitfalls.

The example measurements shown in this section were carried out using the Scalasca Trace Tools v2.5 in conjunction with Score-P v5.0, CubeLib v4.4.3, and CubeGUI v4.4.3 on the JURECA cluster at Jülich Supercomputing Centre. JURECA's compute nodes are equipped with two Intel Xeon E5-2680 v3 (Haswell) 12-core CPUs running at 2.5 GHz, and connected via an EDR InfiniBand fat-tree network. The BT benchmark code was compiled using Intel compilers and linked against ParTec ParaStation MPI (which is based on MPICH). The example commands shown below should therefore be representative for using the Scalasca Trace Tools in a typical HPC cluster environment. For convenience, the resulting post-processed Cube files are also available for download on the Scalasca documentation web page [16].

Note:

> In the following, it is assumed that all Scalasca commands are available in the shell's search path ($PATH), for example, after loading site-specific environment modules. Also, remember that the Scalasca convenience commands use other executables provided by Score-P, CubeLib, and CubeGUI, which therefore need to be available in the search path as well.

2.4.1 Preparing a reference execution

As a first step of every performance analysis, a reference execution using an uninstrumented executable should be performed. First, this step verifies that the code executes cleanly and produces correct results. Second, it later allows to assess the run-time overhead introduced by instrumentation and measurement. And finally, it provides a baseline to compare with after applying some code optimizations. At this stage an appropriate test configuration should be chosen, such that it is both repeatable and long enough to be representative. (Note that excessively long execution durations can make measurement analysis inconvenient or even prohibitive, and therefore should be avoided.)

After unpacking the NPB-MPI source archive, the build system has to be adjusted to the respective environment. For the NAS benchmarks, this is accomplished by a Makefile snippet defining a number of variables used by a generic Makefile. This snippet is called make.def and has to reside in the config/ subdirectory, which already contains a template file that can be copied and adjusted appropriately. In particular, the MPI Fortran compiler wrapper and flags need to be specified, for example:

```
MPIF77     = mpifort
FFLAGS     = -O2
FLINKFLAGS = -O2
```

Note that the MPI C compiler wrapper and flags are not used for building BT, but may also be set in the config/make.def file accordingly to experiment with other NPB benchmarks.

Next, the benchmark can be built from the top-level directory by running make, specifying the number of MPI ranks to use via the NPROCS variable—for BT, this is required to be a square number—as well as the problem size via the CLASS variable on the command line. Valid problem classes (of increasing size) are W, S, A, B, C, D, and E, and can be used to adjust the benchmark runtime to the execution environment. For example, class W or S is appropriate for execution on a laptop with 4 MPI ranks, while the other problem sizes are more suitable for "real" configurations. For the example run on JURECA, 144 MPI ranks and problem class D have been chosen:

```
$ make bt NPROCS=144 CLASS=D
   =========================================
   =      NAS Parallel Benchmarks 3.3      =
   =      MPI/F77/C                        =
   =========================================

cd BT; make NPROCS=144 CLASS=D SUBTYPE= VERSION=
make[1]: Entering directory ‘/tmp/NPB3.3-MPI/BT’
```

```
make[2]: Entering directory ‘/tmp/NPB3.3-MPI/sys’
cc -g  -o setparams setparams.c
make[2]: Leaving directory ‘/tmp/NPB3.3-MPI/sys’
../sys/setparams bt 144 D
make[2]: Entering directory ‘/tmp/NPB3.3-MPI/BT’
mpifort -c  -O2 bt.f
mpifort -c  -O2 make_set.f
mpifort -c  -O2 initialize.f
mpifort -c  -O2 exact_solution.f
mpifort -c  -O2 exact_rhs.f
mpifort -c  -O2 set_constants.f
mpifort -c  -O2 adi.f
mpifort -c  -O2 define.f
mpifort -c  -O2 copy_faces.f
mpifort -c  -O2 rhs.f
mpifort -c  -O2 solve_subs.f
mpifort -c  -O2 x_solve.f
mpifort -c  -O2 y_solve.f
mpifort -c  -O2 z_solve.f
mpifort -c  -O2 add.f
mpifort -c  -O2 error.f
mpifort -c  -O2 verify.f
mpifort -c  -O2 setup_mpi.f
cd ../common; mpifort -c  -O2 print_results.f
cd ../common; mpifort -c  -O2 timers.f
make[3]: Entering directory ‘/tmp/NPB3.3-MPI/BT’
mpifort -c  -O2 btio.f
mpifort -O2 -o ../bin/bt.D.144 bt.o make_set.o initialize.o exact_solution.o \
    exact_rhs.o set_constants.o adi.o define.o copy_faces.o rhs.o solve_subs.o \
    x_solve.o y_solve.o z_solve.o add.o error.o verify.o setup_mpi.o \
    ../common/print_results.o ../common/timers.o btio.o
make[3]: Leaving directory ‘/tmp/NPB3.3-MPI/BT’
make[2]: Leaving directory ‘/tmp/NPB3.3-MPI/BT’
make[1]: Leaving directory ‘/tmp/NPB3.3-MPI/BT’
```

The resulting executable encodes the benchmark configuration in its name and is placed into the `bin/` subdirectory. For the example `make` command above, it is named `bt.D.144`. This binary can now be executed, either via submitting an appropriate batch job (which is beyond the scope of this user guide) or directly in an interactive session.

```
$ cd bin
$ mpiexec -n 144 ./bt.D.144


 NAS Parallel Benchmarks 3.3 -- BT Benchmark

 No input file inputbt.data. Using compiled defaults
 Size:  408x 408x 408
 Iterations:  250    dt:   0.0000200
 Number of active processes:   144

 Time step    1
 Time step   20
 Time step   40
 Time step   60
 Time step   80
 Time step  100
 Time step  120
 Time step  140
 Time step  160
 Time step  180
 Time step  200
 Time step  220
 Time step  240
 Time step  250
 Verification being performed for class D
 accuracy setting for epsilon =  0.1000000000000E-07
 Comparison of RMS-norms of residual
           1 0.2533188551738E+05 0.2533188551738E+05 0.1497879774166E-12
           2 0.2346393716980E+04 0.2346393716980E+04 0.8488743310506E-13
           3 0.6294554366904E+04 0.6294554366904E+04 0.3034271788588E-14
           4 0.5352565376030E+04 0.5352565376030E+04 0.8308967344119E-13
           5 0.3905864038618E+05 0.3905864038618E+05 0.6650300273080E-13
 Comparison of RMS-norms of solution error
           1 0.3100009377557E+03 0.3100009377557E+03 0.1373406191445E-12
           2 0.2424086324913E+02 0.2424086324913E+02 0.1600422929406E-12
           3 0.7782212022645E+02 0.7782212022645E+02 0.4090394153928E-13
           4 0.6835623860116E+02 0.6835623860116E+02 0.3617356324816E-13
           5 0.6065737200368E+03 0.6065737200368E+03 0.2605201960010E-13
 Verification Successful


 BT Benchmark Completed.
 Class           =                        D
 Size            =            408x 408x 408
 Iterations      =                      250
 Time in seconds =                   216.00
 Total processes =                      144
 Compiled procs  =                      144
 Mop/s total     =                270070.08
 Mop/s/process   =                  1875.49
 Operation type  =           floating point
 Verification    =               SUCCESSFUL
 Version         =                    3.3.1
 Compile date    =              18 Mar 2019

 Compile options:
    MPIF77       = mpifort
    FLINK        = $(MPIF77)
    FMPI_LIB     = (none)
    FMPI_INC     = (none)
    FFLAGS       = -O2
    FLINKFLAGS   = -O2
    RAND         = (none)
```

```
 Please send feedbacks and/or the results of this run to:

 NPB Development Team
 Internet: npb@nas.nasa.gov
```

In the selected configuration, the BT benchmark executes 250 iterations of the time step loop, and then verifies that the result matches the expected outcome. Before exiting, the benchmark also reports some configuration details, as well as the wall-clock execution time (216.00 seconds) for the core computation.

2.4.2 Instrumenting the application code

Now that the reference execution was successful, it is time to prepare an instrumented executable using Score-P to perform an initial measurement. By default, Score-P leverages the compiler to automatically instrument every function entry and exit. This is usually the best first approach if one does not have detailed knowledge about the application and needs to identify the hotspots in the code. For BT, using Score-P for instrumentation is simply accomplished by prefixing the compile and link commands specified in the `config/make.def` Makefile snippet by the Score-P instrumenter command `scorep`:

```
MPIF77     = scorep mpifort
```

Note that the linker specification variable `FLINK` in `config/make.def` defaults to the value of `MPIF77`, that is, no further modifications are necessary in this case.

Recompilation of the BT source code in the top-level directory now creates an instrumented executable, overwriting the uninstrumented binary (for archiving purposes, one may consider renaming it before recompiling):

```
$ cd ..
$ make clean
rm -f core
rm -f *~ */core */*~ */*.o */npbparams.h */*.obj */*.exe
rm -f MPI_dummy/test MPI_dummy/libmpi.a
rm -f sys/setparams sys/makesuite sys/setparams.h
rm -f btio.*.out*

$ make bt NPROCS=144 CLASS=D
   =========================================
   =      NAS Parallel Benchmarks 3.3      =
   =      MPI/F77/C                        =
   =========================================

cd BT; make NPROCS=144 CLASS=D SUBTYPE= VERSION=
make[1]: Entering directory ‘/tmp/NPB3.3-MPI/BT’
make[2]: Entering directory ‘/tmp/NPB3.3-MPI/sys’
```

```
cc -g  -o setparams setparams.c
make[2]: Leaving directory ‘/tmp/NPB3.3-MPI/sys’
../sys/setparams bt 144 D
make[2]: Entering directory ‘/tmp/NPB3.3-MPI/BT’
scorep mpifort -c  -O2 bt.f
scorep mpifort -c  -O2 make_set.f
scorep mpifort -c  -O2 initialize.f
scorep mpifort -c  -O2 exact_solution.f
scorep mpifort -c  -O2 exact_rhs.f
scorep mpifort -c  -O2 set_constants.f
scorep mpifort -c  -O2 adi.f
scorep mpifort -c  -O2 define.f
scorep mpifort -c  -O2 copy_faces.f
scorep mpifort -c  -O2 rhs.f
scorep mpifort -c  -O2 solve_subs.f
scorep mpifort -c  -O2 x_solve.f
scorep mpifort -c  -O2 y_solve.f
scorep mpifort -c  -O2 z_solve.f
scorep mpifort -c  -O2 add.f
scorep mpifort -c  -O2 error.f
scorep mpifort -c  -O2 verify.f
scorep mpifort -c  -O2 setup_mpi.f
cd ../common; scorep mpifort -c  -O2 print_results.f
cd ../common; scorep mpifort -c  -O2 timers.f
make[3]: Entering directory ‘/tmp/NPB3.3-MPI/BT’
scorep mpifort -c  -O2 btio.f
scorep mpifort -O2 -o ../bin/bt.D.144 bt.o make_set.o initialize.o \
    exact_solution.o exact_rhs.o set_constants.o adi.o define.o copy_faces.o \
    rhs.o solve_subs.o x_solve.o y_solve.o z_solve.o add.o error.o verify.o \
    setup_mpi.o ../common/print_results.o ../common/timers.o btio.o
make[3]: Leaving directory ‘/tmp/NPB3.3-MPI/BT’
make[2]: Leaving directory ‘/tmp/NPB3.3-MPI/BT’
make[1]: Leaving directory ‘/tmp/NPB3.3-MPI/BT’
```

2.4.3 Initial summary measurement

The instrumented executable prepared in the previous step can now be executed under the control of the `scalasca -analyze` (or short `scan`) convenience command to perform an initial summary measurement:

```
$ cd bin
$ scalasca -analyze mpiexec -n 144 ./bt.D.144
S=C=A=N: Scalasca 2.5 runtime summarization
S=C=A=N: ./scorep_bt_144_sum experiment archive
S=C=A=N: Mon Mar 18 13:44:46 2019: Collect start
mpiexec -n 144 ./bt.D.144


 NAS Parallel Benchmarks 3.3 -- BT Benchmark

 No input file inputbt.data. Using compiled defaults
```

```
Size:  408x 408x 408
Iterations:  250    dt:   0.0000200
Number of active processes:   144

Time step    1
Time step   20
Time step   40
Time step   60
Time step   80
Time step  100
Time step  120
Time step  140
Time step  160
Time step  180
Time step  200
Time step  220
Time step  240
Time step  250
Verification being performed for class D
accuracy setting for epsilon =  0.1000000000000E-07
Comparison of RMS-norms of residual
          1 0.2533188551738E+05 0.2533188551738E+05 0.1499315900507E-12
          2 0.2346393716980E+04 0.2346393716980E+04 0.8546885387975E-13
          3 0.6294554366904E+04 0.6294554366904E+04 0.2745293523008E-14
          4 0.5352565376030E+04 0.5352565376030E+04 0.8376934357159E-13
          5 0.3905864038618E+05 0.3905864038618E+05 0.6650300273080E-13
Comparison of RMS-norms of solution error
          1 0.3100009377557E+03 0.3100009377557E+03 0.1373406191445E-12
          2 0.2424086324913E+02 0.2424086324913E+02 0.1600422929406E-12
          3 0.7782212022645E+02 0.7782212022645E+02 0.4090394153928E-13
          4 0.6835623860116E+02 0.6835623860116E+02 0.3596566920650E-13
          5 0.6065737200368E+03 0.6065737200368E+03 0.2605201960010E-13
Verification Successful


BT Benchmark Completed.
Class           =                        D
Size            =            408x 408x 408
Iterations      =                      250
Time in seconds =                   413.25
Total processes =                      144
Compiled procs  =                      144
Mop/s total     =                141162.75
Mop/s/process   =                   980.30
Operation type  =           floating point
Verification    =               SUCCESSFUL
Version         =                    3.3.1
Compile date    =              18 Mar 2019

Compile options:
   MPIF77       = scorep mpifort
   FLINK        = $(MPIF77)
   FMPI_LIB     = (none)
```

```
    FMPI_INC     = (none)
    FFLAGS       = -O2
    FLINKFLAGS   = -O2
    RAND         = (none)


 Please send feedbacks and/or the results of this run to:

 NPB Development Team
 Internet: npb@nas.nasa.gov


S=C=A=N: Mon Mar 18 13:51:47 2019: Collect done (status=0) 421s
S=C=A=N: ./scorep_bt_144_sum complete.

$ ls scorep_bt_144_sum
MANIFEST.md  profile.cubex  scorep.cfg  scorep.log
```

As can be seen, the measurement run successfully produced an experiment directory named `scorep_bt_144_sum` containing

- a text file `MANIFEST.md` briefly describing the directory contents produced by the Score-P measurement system,
- the runtime summary result file `profile.cubex`,
- a copy of the measurement configuration in `scorep.cfg`, and
- the measurement log file `scorep.log`.

However, application execution took almost twice as long as the reference run (413.25 vs. 216.00 seconds). That is, instrumentation and associated measurements introduced a non-negligible amount of run-time overhead. While it is possible to interactively examine the generated summary result file using the Cube report browser, this should only be done with great caution since the substantial overhead negatively impacts the accuracy of the measurement. Therefore, such measurements can easily be misleading.

2.4.4 Optimizing the measurement configuration

To avoid drawing wrong conclusions based on skewed performance data due to excessive measurement overhead, it is often necessary to optimize the measurement configuration before conducting any additional experiments. This can be achieved in various ways, for example, using runtime filtering, selective recording, or manual instrumentation controlling measurement. Please consult the Score-P Manual [17] for details on the available options. However, in many cases it is already sufficient to filter a small number of frequently executed but computationally inexpensive user functions to reduce the measurement overhead to an acceptable level. In this context, filtering means that those functions are still executed, but no measurements are taken and recorded for them. Therefore, filtered functions no longer show up in the measurement report, and the associated execution time is attributed to the parent function from which they are called (similar to inlining performed by the compiler). The selection of the routines to be filtered has to be done with care, though, as it affects the granularity of the measurement and too aggressive filtering might "blur" the location of important hotspots.

To assist in identifying candidate functions for runtime filtering, the initial summary report can be scored using the -s option of the scalasca -examine command:

```
$ scalasca -examine -s scorep_bt_144_sum
INFO: Post-processing runtime summarization report (profile.cubex)...
scorep-score -r ./scorep_bt_144_sum/profile.cubex \
    > ./scorep_bt_144_sum/scorep.score
INFO: Score report written to ./scorep_bt_144_sum/scorep.score

$ head -n 25 scorep_bt_144_sum/scorep.score

Estimated aggregate size of event trace:                   3701GB
Estimated requirements for largest trace buffer (max_buf): 26GB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       26GB
(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/     region
                                                           visit[us]
       ALL 27,597,828,625 152,806,258,921 60341.59   100.0      0.39 ALL
       USR 27,592,847,542 152,791,289,689 50827.11    84.2      0.33 USR
       MPI      4,086,824      10,016,496  9177.48    15.2    916.24 MPI
       COM        894,218       4,952,592   336.80     0.6     68.01 COM
    SCOREP             41             144     0.19     0.0   1305.21 SCOREP

       USR  9,123,406,734  50,517,453,756  4716.33     7.8      0.09 matvec_sub_
       USR  9,123,406,734  50,517,453,756  8774.13    14.5      0.17 binvcrhs_
       USR  9,123,406,734  50,517,453,756  6520.04    10.8      0.13 matmul_sub_
       USR    200,157,360   1,108,440,240    89.94     0.1      0.08 exact_solution_
       USR     22,632,168     124,121,508    13.43     0.0      0.11 binvrhs_
       MPI      1,608,942       2,603,232     9.57     0.0      3.68 MPI_Irecv
       MPI      1,608,942       2,603,232    14.83     0.0      5.70 MPI_Isend
       MPI        861,432       4,771,008  7936.43    13.2   1663.47 MPI_Wait
       USR        234,936       1,301,184     3.24     0.0      2.49 lhsabinit_
       USR         78,312         433,728  6213.68    10.3  14326.21 x_solve_cell_
```

As can be seen from the top of the score report, the estimated size for an event trace measurement without filtering applied is approximately 3.7 TiB, with the process-local maximum across all ranks being roughly 26 GiB. Considering the 128 GiB of main memory available on JURECA's compute nodes, the 24 MPI ranks per node, and the fact that Score-P's internal memory buffer is limited to 4 GiB per process, a tracing experiment with this configuration is clearly prohibitive if disruptive intermediate trace buffer flushes are to be avoided.

The next section of the score output provides a table which shows how the trace memory requirements of a single process (column max_buf) as well as the overall number of visits and CPU allocation time are distributed among certain function groups. For traces that can be handled by the Scalasca Trace Tools, the most relevant groups are:

- **MPI**: MPI API functions.
- **OMP**: OpenMP constructs and API functions.

- **PTHREAD**: POSIX threads API functions.
- **COM**: User functions/regions that appear on a call path to a parallelization API call or construct (MPI/OpenMP/POSIX threads). These functions provide the context of parallelization API usage and should therefore only be filtered with care.
- **USR**: User functions/regions that do not appear on a call path to a parallelization API call or construct (MPI/OpenMP/POSIX threads).
- **SCOREP**: Artificial regions generated by the Score-P measurement system.

The detailed breakdown by region below the summary provides a classification according to these function groups (column `type`) for each region found in the summary report. Investigation of this part of the score report reveals that most of the trace data would be generated by about 50 billion calls to each of the three routines `matvec_sub`, `binvcrhs` and `matmul_sub`, which are classified as `USR`. And although the percentage of time spent in these routines at first glance suggest that they are important, the average time per visit is below 170 nanoseconds (column `time/visit`). That is, the relative measurement overhead for these functions is substantial, and thus a significant amount of the reported time is very likely spent in the Score-P measurement system rather than in the application itself. Therefore, these routines constitute good candidates for being filtered (like they are good candidates for being inlined by the compiler). Additionally selecting the `exact_solution` routine, which generates about 200 MiB of event data on a single rank with very little runtime impact, a reasonable Score-P filtering file would therefore look like this:

```
SCOREP_REGION_NAMES_BEGIN
    EXCLUDE
        binvcrhs_
        matvec_sub_
        matmul_sub_
        exact_solution_
SCOREP_REGION_NAMES_END
```

Please refer to the Score-P User Manual [17] for a detailed description of the filter file format, how to filter based on file names, define (and combine) blacklists and whitelists, and how to use wildcards for convenience. Also note that the run-time filtering approach used in this example only affects routines in the `USR` and `COM` groups. Measurements for other groups can—to certain degrees—be controlled by other means, as the generated events have to meet various consistency requirements.

The effectiveness of this filter—in terms of generated trace data—can be examined by scoring the initial summary report again, this time also specifying the filter file using the `-f` option of the `scalasca -examine` command. This way a filter file can be incrementally developed, avoiding the need to conduct many measurements to step-by-step investigate the effect of filtering individual functions.

```
$ scalasca -examine -s -f npb-bt.filt scorep_bt_144_sum
scorep-score -f npb-bt.filt -r ./scorep_bt_144_sum/profile.cubex \
    > ./scorep_bt_144_sum/scorep.score_npb-bt.filt
INFO: Score report written to ./scorep_bt_144_sum/scorep.score_npb-bt.filt

$ head -n 25 scorep_bt_144_sum/scorep.score_npb-bt.filt
```

```
Estimated aggregate size of event trace:                   3920MB
Estimated requirements for largest trace buffer (max_buf): 28MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       30MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=30MB to avoid intermediate flushes
 or reduce requirements using USR regions filters.)

flt   type     max_buf[B]          visits  time[s] time[%] time/    region
                                                           visit[us]
 -     ALL 27,597,828,625 152,806,258,921 60341.59   100.0      0.39 ALL
 -     USR 27,592,847,542 152,791,289,689 50827.11    84.2      0.33 USR
 -     MPI      4,086,824      10,016,496  9177.48    15.2    916.24 MPI
 -     COM        894,218       4,952,592   336.80     0.6     68.01 COM
 -  SCOREP             41             144     0.19     0.0   1305.21 SCOREP

 *     ALL     28,762,789     145,457,413 40241.14    66.7    276.65 ALL-FLT
 +     FLT 27,570,377,562 152,660,801,508 20100.44    33.3      0.13 FLT
 *     USR     23,781,706     130,488,181 30726.67    50.9    235.47 USR-FLT
 -     MPI      4,086,824      10,016,496  9177.48    15.2    916.24 MPI-FLT
 *     COM        894,218       4,952,592   336.80     0.6     68.01 COM-FLT
 -  SCOREP             41             144     0.19     0.0   1305.21 SCOREP-FLT

 +     USR  9,123,406,734  50,517,453,756  4716.33     7.8      0.09 matvec_sub_
 +     USR  9,123,406,734  50,517,453,756  8774.13    14.5      0.17 binvcrhs_
 +     USR  9,123,406,734  50,517,453,756  6520.04    10.8      0.13 matmul_sub_
 +     USR    200,157,360   1,108,440,240    89.94     0.1      0.08 exact_solution_
 -     USR     22,632,168     124,121,508    13.43     0.0      0.11 binvrhs_
 -     MPI      1,608,942       2,603,232     9.57     0.0      3.68 MPI_Irecv
 -     MPI      1,608,942       2,603,232    14.83     0.0      5.70 MPI_Isend
 -     MPI        861,432       4,771,008  7936.43    13.2   1663.47 MPI_Wait
 -     USR        234,936       1,301,184     3.24     0.0      2.49 lhsabinit_
```

Below the (original) function group summary, the score report now also includes a second summary with the filter applied. Here, an additional group FLT is added, which subsumes all filtered regions. Moreover, the column flt indicates whether a region/function group is filtered ("+"), not filtered ("-"), or possibly partially filtered ("*", only used for function groups).

As expected, the estimate for the aggregate event trace size drops down to 3.9 GiB, and the process-local maximum across all ranks is reduced to 28 MiB. Since the Score-P measurement system also creates a number of internal data structures (e.g., to track MPI requests and communicators), the suggested setting for the SCOREP_TOTAL_MEMORY environment variable to adjust the maximum amount of memory used by the Score-P memory management is 30 MiB when tracing is configured (see Section 2.4.6).

2.4.5 Summary measurement & examination

The filtering file prepared in Section 2.4.4 can now be applied to produce a new summary measurement, ideally with reduced measurement overhead to improve accuracy. This can be accomplished by providing the filter file name to scalasca -analyze via the -f option.

Attention:

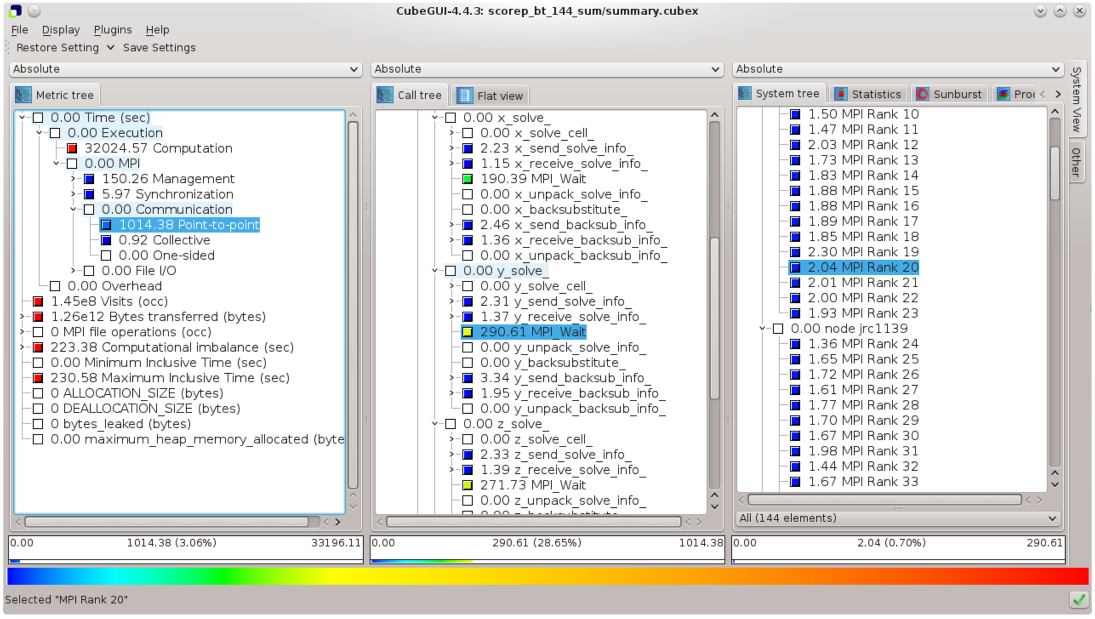
> Before re-analyzing the application, the unfiltered summary experiment should be renamed (or removed), since `scalasca -analyze` will by default not overwrite the existing experiment directory and abort immediately.

```
$ mv scorep_bt_144_sum scorep_bt_144_sum.nofilt
$ scalasca -analyze -f npb-bt.filt mpiexec -n 144 ./bt.D.144
S=C=A=N: Scalasca 2.5 runtime summarization
S=C=A=N: ./scorep_bt_144_sum experiment archive
S=C=A=N: Mon Mar 18 13:52:32 2019: Collect start
mpiexec -n 144 ./bt.D.144


 NAS Parallel Benchmarks 3.3 -- BT Benchmark

 No input file inputbt.data. Using compiled defaults
 Size:  408x 408x 408
 Iterations:  250    dt:   0.0000200
 Number of active processes:   144

 Time step    1
 Time step   20
 Time step   40
 Time step   60
 Time step   80
 Time step  100
 Time step  120
 Time step  140
 Time step  160
 Time step  180
 Time step  200
 Time step  220
 Time step  240
 Time step  250
 Verification being performed for class D
 accuracy setting for epsilon =  0.1000000000000E-07
 Comparison of RMS-norms of residual
           1 0.2533188551738E+05 0.2533188551738E+05 0.1499315900507E-12
           2 0.2346393716980E+04 0.2346393716980E+04 0.8546885387975E-13
           3 0.6294554366904E+04 0.6294554366904E+04 0.2745293523008E-14
           4 0.5352565376030E+04 0.5352565376030E+04 0.8376934357159E-13
           5 0.3905864038618E+05 0.3905864038618E+05 0.6650300273080E-13
 Comparison of RMS-norms of solution error
           1 0.3100009377557E+03 0.3100009377557E+03 0.1373406191445E-12
           2 0.2424086324913E+02 0.2424086324913E+02 0.1600422929406E-12
           3 0.7782212022645E+02 0.7782212022645E+02 0.4090394153928E-13
           4 0.6835623860116E+02 0.6835623860116E+02 0.3596566920650E-13
           5 0.6065737200368E+03 0.6065737200368E+03 0.2605201960010E-13
 Verification Successful


 BT Benchmark Completed.
 Class           =                        D
 Size            =            408x 408x 408
 Iterations      =                      250
 Time in seconds =                   228.02
 Total processes =                      144
 Compiled procs  =                      144
 Mop/s total     =                255839.13
 Mop/s/process   =                  1776.66
 Operation type  =           floating point
 Verification    =               SUCCESSFUL
 Version         =                    3.3.1
 Compile date    =              18 Mar 2019

 Compile options:
    MPIF77       = scorep mpifort
    FLINK        = $(MPIF77)
    FMPI_LIB     = (none)
    FMPI_INC     = (none)
    FFLAGS       = -O2
    FLINKFLAGS   = -O2
    RAND         = (none)


 Please send feedbacks and/or the results of this run to:

 NPB Development Team
 Internet: npb@nas.nasa.gov


S=C=A=N: Mon Mar 18 13:56:24 2019: Collect done (status=0) 232s
S=C=A=N: ./scorep_bt_144_sum complete.

$ ls scorep_bt_144_sum
MANIFEST.md    scorep.cfg     scorep.log
profile.cubex  scorep.filter  summary.cubex
```

This new measurement produced an experiment directory containing one additional file compared to the initial run: a copy of the measurement filter in `scorep.filter`. Notice that applying the runtime filtering reduced the measurement overhead significantly, down to now only 5.5% (228.02 seconds vs. 216.00 seconds for the reference run). This new measurement with the optimized configuration should therefore quite accurately represent the real runtime behavior of the BT application, and can now be post-processed and interactively explored using the Cube result browser. These two steps can be conveniently initiated using the `scalasca -examine` command:

```
$ scalasca -examine scorep_bt_144_sum
INFO: Post-processing runtime summarization report (profile.cubex)...
INFO: Displaying ./scorep_bt_144_sum/summary.cubex...
```

Examination of the summary result (see Figure 2.1 for a screenshot and Section 2.4.5.1 for a brief summary of how to use the Cube browser) shows that 96.5% of the overall CPU allocation time is spent in computations, while 3% of the time is spent in MPI point-to-point communication functions and the remainder scattered across other activities. The point-to-point time is almost entirely spent in `MPI_Wait` calls inside the three solver functions `x_solve`, `y_solve` and `z_solve`, as well as an `MPI_Waitall` in the boundary exchange routine `copy_faces`. Computation time is also mostly spent in the solver routines and the boundary exchange, however, inside the different `solve_cell`, `backsubstitute` and `compute_rhs` functions. While the aggregated time spent in the computational routines seems to be relatively balanced across the different MPI ranks (determined using the box plot view in the right pane), there is quite some variation for the `MPI_Wait` / `MPI_Waitall` calls.



```
□ 0.00 Time (sec)
  □ 0.00 Execution
    ■ 32024.57 Computation
    □ 0.00 MPI
      ■ 150.26 Management
      ■ 5.97 Synchronization
      □ 0.00 Communication
        ■ 1014.38 Point-to-point
        ■ 0.92 Collective
        □ 0.00 One-sided
      □ 0.00 File I/O
  □ 0.00 Overhead
■ 1.45e8 Visits (occ)
■ 1.26e12 Bytes transferred (bytes)
□ 0 MPI file operations (occ)
■ 223.38 Computational imbalance (sec)
□ 0.00 Minimum Inclusive Time (sec)
■ 230.58 Maximum Inclusive Time (sec)
□ 0 ALLOCATION_SIZE (bytes)
□ 0 DEALLOCATION_SIZE (bytes)
□ 0 bytes_leaked (bytes)
□ 0.00 maximum_heap_memory_allocated (byte
```

```
□ 0.00 x_solve_
  □ 0.00 x_solve_cell_
  ■ 2.23 x_send_solve_info_
  ■ 1.15 x_receive_solve_info_
  ■ 190.39 MPI_Wait
  □ 0.00 x_unpack_solve_info_
  □ 0.00 x_backsubstitute_
  ■ 2.46 x_send_backsub_info_
  ■ 1.36 x_receive_backsub_info_
  □ 0.00 x_unpack_backsub_info_
□ 0.00 y_solve_
  □ 0.00 y_solve_cell_
  ■ 2.31 y_send_solve_info_
  ■ 1.37 y_receive_solve_info_
  ■ 290.61 MPI_Wait
  □ 0.00 y_unpack_solve_info_
  □ 0.00 y_backsubstitute_
  ■ 3.34 y_send_backsub_info_
  ■ 1.95 y_receive_backsub_info_
  □ 0.00 y_unpack_backsub_info_
□ 0.00 z_solve_
  □ 0.00 z_solve_cell_
  ■ 2.33 z_send_solve_info_
  ■ 1.39 z_receive_solve_info_
  ■ 271.73 MPI_Wait
  □ 0.00 z_unpack_solve_info_
```

```
  ■ 1.50 MPI Rank 10
  ■ 1.47 MPI Rank 11
  ■ 2.03 MPI Rank 12
  ■ 1.73 MPI Rank 13
  ■ 1.83 MPI Rank 14
  ■ 1.88 MPI Rank 15
  ■ 1.88 MPI Rank 16
  ■ 1.89 MPI Rank 17
  ■ 1.85 MPI Rank 18
  ■ 2.30 MPI Rank 19
  ■ 2.04 MPI Rank 20
  ■ 2.01 MPI Rank 21
  ■ 2.00 MPI Rank 22
  ■ 1.93 MPI Rank 23
□ 0.00 node jrc1139
  ■ 1.36 MPI Rank 24
  ■ 1.65 MPI Rank 25
  ■ 1.72 MPI Rank 26
  ■ 1.61 MPI Rank 27
  ■ 1.70 MPI Rank 29
  ■ 1.77 MPI Rank 28
  ■ 1.67 MPI Rank 30
  ■ 1.98 MPI Rank 31
  ■ 1.44 MPI Rank 32
  ■ 1.67 MPI Rank 33
```

Figure 2.1: Screenshot of a summary experiment result in the Cube report browser.

2.4.5.1 Using the Cube browser

The following paragraphs provide a very brief introduction to the usage of the Cube analysis report browser. To make effective use of the GUI, however, please also consult the CubeGUI User Guide [4].

Cube is a generic user interface for presenting and browsing performance and debugging information from parallel applications. The underlying data model is independent from particular performance properties to be displayed. The Cube main window (see Figure 2.1) consists of three panels containing tree displays or alternate graphical views of analysis reports. The left panel shows performance properties of the execution, such as time or the number of visits. The middle pane shows the call tree or a flat profile of the application. The right pane either shows the system hierarchy consisting of, for example, machines, compute nodes, processes, and threads, or various graphical representations such as a topological view of the application's processes and threads (if available), or a box/violin plot view showing the statistical distribution of values across the system. All tree nodes are labeled with a metric value and a color-coded box which can help in identifying hotspots.

The metric value color is determined from the proportion of the total (root) value or some other specified reference value, using the color scale at the bottom of the window.

A click on a performance property or a call path selects the corresponding node. This has the effect that the metric value held by this node (such as execution time) will be further broken down into its constituents in the panels right of the selected node. For example, after selecting a performance property, the middle panel shows its distribution across the call tree. After selecting a call path (i.e., a node in the call tree), the system tree shows the distribution of the performance property in that call path across the system locations. A click on the icon to the left of a node in each tree expands or collapses that node. By expanding or collapsing nodes in each of the three trees, the analysis results can be viewed on different levels of granularity (inclusive vs. exclusive values).

All tree displays support a context menu, which is accessible using the right mouse button and provides further options. For example, to obtain the exact definition of a performance property, select "Documentation" in the context menu associated with each performance property. A brief description can also be obtained from the menu option "Info".

2.4.6 Trace collection and analysis

While summary profiles only provide process- or thread-local data aggregated over time, event traces contain detailed time-stamped event data which also allows to reconstruct the dynamic behavior of an application. This enables tools such as the Scalasca trace analyzer to provide even more insights into the performance behavior of an application, for example, whether the time spent in MPI communication is real message processing time or incurs significant wait states (i.e., intervals where a process sits idle without doing useful work waiting for data from other processes to arrive).

Trace collection and subsequent automatic analysis by the Scalasca trace analyzer can be enabled using the `-t` option of `scalasca -analyze`. Since this option enables trace collection *in addition* to collecting a summary measurement, it is often used in conjunction with the `-q` option which turns off measurement entirely. (Note that the order in which these two options are specified matters: First turn off measurement using `-q`, then enable tracing with `-t`.)

Attention:

> **Do not forget to specify an appropriate measurement configuration** (i.e., a filtering file and `SCOREP_TOTAL_MEMORY` setting)! Otherwise, you may easily fill up your disks and suffer from uncoordinated intermediate trace buffer flushes, which typically render such traces to be of little (or no) value!

For our example measurement, scoring of the initial summary report in Section 2.4.4 with the filter applied estimated a total memory requirement of 30 MiB per process (which could be verified by re-scoring the filtered summary measurement). As this exceeds the default `SCOREP_TOTAL_MEMORY` setting of 16 MiB, use of the prepared filtering file alone is not yet sufficient to avoid intermediate trace buffer flushes. In addition, the `SCOREP_TOTAL_MEMORY` setting has to be adjusted accordingly before starting the trace collection and analysis. For the example measurement shown below, a slightly larger memory buffer of 32 MiB is used, although this is not strictly necessary. As an alternative, the filtering file could be extended to also exclude additional routines from measurement (e.g., `binvrhs`) to further reduce the trace buffer requirements at the expense of loosing details.

With this setting in place, a trace measurement can now be collected and subsequently analyzed by the Scalasca trace analyzer. Note that renaming or removing the summary experiment directory is not necessary, as trace experiments are created with suffix "`trace`".

```
$ export SCOREP_TOTAL_MEMORY=32M
$ scalasca -analyze -q -t -f npb-bt.filt mpiexec -n 144 ./bt.D.144
S=C=A=N: Scalasca 2.5 trace collection and analysis
S=C=A=N: ./scorep_bt_144_trace experiment archive
S=C=A=N: Mon Mar 18 13:57:08 2019: Collect start
mpiexec -n 144 ./bt.D.144


 NAS Parallel Benchmarks 3.3 -- BT Benchmark

 No input file inputbt.data. Using compiled defaults
 Size:  408x 408x 408
 Iterations:  250    dt:   0.0000200
 Number of active processes:   144

 Time step    1
 Time step   20
 Time step   40
 Time step   60
 Time step   80
 Time step  100
 Time step  120
 Time step  140
 Time step  160
 Time step  180
 Time step  200
 Time step  220
 Time step  240
 Time step  250
 Verification being performed for class D
 accuracy setting for epsilon =  0.1000000000000E-07
 Comparison of RMS-norms of residual
           1 0.2533188551738E+05 0.2533188551738E+05 0.1499315900507E-12
           2 0.2346393716980E+04 0.2346393716980E+04 0.8546885387975E-13
           3 0.6294554366904E+04 0.6294554366904E+04 0.2745293523008E-14
           4 0.5352565376030E+04 0.5352565376030E+04 0.8376934357159E-13
           5 0.3905864038618E+05 0.3905864038618E+05 0.6650300273080E-13
 Comparison of RMS-norms of solution error
           1 0.3100009377557E+03 0.3100009377557E+03 0.1373406191445E-12
           2 0.2424086324913E+02 0.2424086324913E+02 0.1600422929406E-12
           3 0.7782212022645E+02 0.7782212022645E+02 0.4090394153928E-13
           4 0.6835623860116E+02 0.6835623860116E+02 0.3596566920650E-13
           5 0.6065737200368E+03 0.6065737200368E+03 0.2605201960010E-13
 Verification Successful


 BT Benchmark Completed.
 Class           =                        D
 Size            =            408x 408x 408
 Iterations      =                     250
 Time in seconds =                  227.78
 Total processes =                     144
 Compiled procs  =                     144
 Mop/s total     =               256110.17
 Mop/s/process   =                 1778.54
 Operation type  =          floating point
 Verification    =              SUCCESSFUL
 Version         =                   3.3.1
 Compile date    =             18 Mar 2019

 Compile options:
    MPIF77       = scorep mpifort
    FLINK        = $(MPIF77)
    FMPI_LIB     = (none)
    FMPI_INC     = (none)
    FFLAGS       = -O2
    FLINKFLAGS   = -O2
    RAND         = (none)


 Please send feedbacks and/or the results of this run to:

 NPB Development Team
 Internet: npb@nas.nasa.gov


S=C=A=N: Mon Mar 18 14:01:04 2019: Collect done (status=0) 236s
S=C=A=N: Mon Mar 18 14:01:04 2019: Analyze start
mpiexec -n 144 scout.mpi ./scorep_bt_144_trace/traces.otf2
SCOUT   (Scalasca 2.5)
Copyright (c) 1998-2019 Forschungszentrum Juelich GmbH
Copyright (c) 2009-2014 German Research School for Simulation Sciences GmbH

Analyzing experiment archive ./scorep_bt_144_trace/traces.otf2

Opening experiment archive ... done (0.008s).
Reading definition data    ... done (0.017s).
Reading event trace data   ... done (0.280s).
Preprocessing              ... done (0.366s).
Analyzing trace data       ... done (2.511s).
Writing analysis report    ... done (0.104s).

Max. memory usage         : 170.812MB
*** WARNING ***
40472 clock condition violations detected:
   Point-to-point:    40472
   Collective    :        0
This usually leads to inconsistent analysis results.

Try running the analyzer using the '--time-correct' command-line
option to apply a correction algorithm.
```

```
Total processing time     : 3.383s
S=C=A=N: Mon Mar 18 14:01:09 2019: Analyze done (status=0) 5s
Warning: 3.686GB of analyzed trace data retained in ./scorep_bt_144_trace/traces!
S=C=A=N: ./scorep_bt_144_trace complete.

$ ls scorep_bt_144_trace
MANIFEST.md  scorep.filter  scout.cubex  trace.cubex  traces.def   trace.stat
scorep.cfg   scorep.log     scout.log    traces       traces.otf2
```

As can be seen, the `scalasca -analyze` convenience command automatically initiates the trace analysis after successful trace collection. Besides the already known files `MANIFEST.md`, `scorep.cfg`, `scorep.filter`, and `scorep.log`, the generated experiment directory `scorep_bt_144_trace` now contains artifacts related to trace measurement and analysis:

- an OTF2 trace archive consisting of the anchor file `traces.otf2`, the global definitions file `traces.def`, and the per-process data in the `traces/` directory,
- the trace analysis log file `scout.log`, and finally
- the trace analysis reports `scout.cubex` and `trace.stat`.

The Scalasca trace analyzer also warned about a number of point-to-point clock condition violations it detected. A clock condition violation is a violation of the logical event order that can occur when the local clocks of the individual compute nodes are insufficiently synchronized. For example, based on the measured timestamps, a receive operation may appear to have finished before the corresponding send operation started—something that is obviously not possible. The Scalasca trace analyzer includes a correction algorithm [1] that can be applied in such cases to restore the logical event order, while trying to preserve the length of intervals between local events in the vicinity of the violation.

To enable this correction algorithm, the `--time-correct` command-line option has to be passed to the Scalasca trace analyzer. However, since the analyzer is implicitly started through the `scalasca -analyze` command, this option has to be set using the `SCAN_ANALYZE_OPTS` environment variable, which holds command-line options that `scalasca -analyze` should forward to the trace analyzer. Instead of collecting and analyzing a new experiment, an existing trace measurement can also be re-analyzed using the `-a` option of the `scalasca -analyze` command:

```
$ export SCAN_ANALYZE_OPTS="--time-correct"
$ scalasca -analyze -a mpiexec -n 144 ./bt.D.144
S=C=A=N: Scalasca 2.5 trace analysis
S=C=A=N: ./scorep_bt_144_trace experiment archive
S=C=A=N: Mon Mar 18 14:02:57 2019: Analyze start
mpiexec -n 144 scout.mpi --time-correct ./scorep_bt_144_trace/traces.otf2
SCOUT   (Scalasca 2.5)
Copyright (c) 1998-2019 Forschungszentrum Juelich GmbH
Copyright (c) 2009-2014 German Research School for Simulation Sciences GmbH

Analyzing experiment archive ./scorep_bt_144_trace/traces.otf2

Opening experiment archive ... done (0.013s).
Reading definition data    ... done (0.023s).
Reading event trace data   ... done (0.615s).
Preprocessing              ... done (0.478s).
Timestamp correction       ... done (0.463s).
Analyzing trace data       ... done (2.653s).
Writing analysis report    ... done (0.106s).

Max. memory usage         : 174.000MB

 # passes        : 1
 # violated      : 9450
 # corrected     : 23578735
 # reversed-p2p  : 9450
 # reversed-coll : 0
 # reversed-omp  : 0
 # events        : 301330922
 max. error      : 0.000048 [s]
 error at final. : 0.000000 [%]
 Max slope       : 0.394363334

Total processing time     : 4.448s
S=C=A=N: Mon Mar 18 14:03:07 2019: Analyze done (status=0) 10s
Warning: 3.686GB of analyzed trace data retained in ./scorep_bt_144_trace/traces!
S=C=A=N: ./scorep_bt_144_trace complete.
```

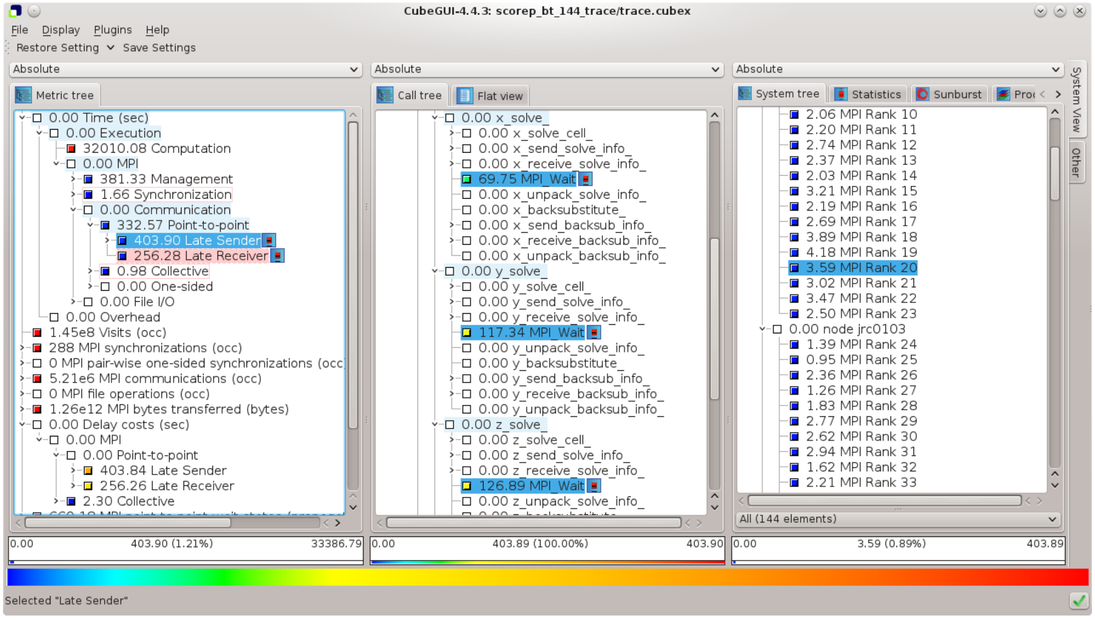
Note:

> The additional time required to execute the timestamp correction algorithm is typically small compared to the trace data I/O time and waiting times in the batch queue for starting a second analysis job. On platforms where clock condition violations are likely to occur (i.e., clusters), it is therefore often convenient to enable the timestamp correction algorithm by default.

Similar to the summary report, the trace analysis report can finally be post-processed and interactively explored using the Cube report browser, again using the `scalasca -examine` convenience command:

```
$ scalasca -examine scorep_bt_144_trace
INFO: Post-processing trace analysis report (scout.cubex)...
INFO: Displaying ./scorep_bt_144_trace/trace.cubex...
```

The report generated by the Scalasca trace analyzer is again a profile in CUBE4 format, however, enriched with additional performance properties. Examination shows that about two-thirds of the time spent in MPI point-to-point communication is waiting time, split into roughly 60% in *Late Sender* and 40% in *Late Receiver* wait states (see Figure 2.2). While the execution time in the `solve_cell` routines looked relatively balanced in the summary profile, examination of the *Critical path imbalance* metric shows that these routines in fact exhibit a small amount of imbalance, which is likely to cause the wait states at the next synchronization point. This can be verified using the *Late Sender delay costs* metric, which confirms that the `solve_cells` as well as the `y_backsubstitute` and `z_backsubstitute` routines are responsible for almost all of the *Late Sender* wait states. Likewise, the *Late Receiver delay costs* metric shows that the majority of the *Late Receiver* wait states are caused by the `solve_cells` routines as well as the `MPI_Wait` calls in the solver routines, where the latter indicates a communication imbalance or an inefficient communication pattern.



```
0.00 Time (sec)
  0.00 Execution
    32010.08 Computation
    0.00 MPI
      381.33 Management
      1.66 Synchronization
      0.00 Communication
        332.57 Point-to-point
          403.90 Late Sender
          256.28 Late Receiver
        0.98 Collective
        0.00 One-sided
      0.00 File I/O
  0.00 Overhead
1.45e8 Visits (occ)
288 MPI synchronizations (occ)
0 MPI pair-wise one-sided synchronizations (occ
5.21e6 MPI communications (occ)
0 MPI file operations (occ)
1.26e12 MPI bytes transferred (bytes)
0.00 Delay costs (sec)
  0.00 MPI
    0.00 Point-to-point
      403.84 Late Sender
      256.26 Late Receiver
    2.30 Collective
```

```
0.00 x_solve_
  0.00 x_solve_cell_
  0.00 x_send_solve_info_
  0.00 x_receive_solve_info_
    69.75 MPI_Wait
  0.00 x_unpack_solve_info_
  0.00 x_backsubstitute_
  0.00 x_send_backsub_info_
  0.00 x_receive_backsub_info_
  0.00 x_unpack_backsub_info_
0.00 y_solve_
  0.00 y_solve_cell_
  0.00 y_send_solve_info_
  0.00 y_receive_solve_info_
    117.34 MPI_Wait
  0.00 y_unpack_solve_info_
  0.00 y_backsubstitute_
  0.00 y_send_backsub_info_
  0.00 y_receive_backsub_info_
  0.00 y_unpack_backsub_info_
0.00 z_solve_
  0.00 z_solve_cell_
  0.00 z_send_solve_info_
  0.00 z_receive_solve_info_
    126.89 MPI_Wait
  0.00 z_unpack_solve_info_
```

```
  2.06 MPI Rank 10
  2.20 MPI Rank 11
  2.74 MPI Rank 12
  2.37 MPI Rank 13
  2.03 MPI Rank 14
  3.21 MPI Rank 15
  2.19 MPI Rank 16
  2.69 MPI Rank 17
  3.89 MPI Rank 18
  4.18 MPI Rank 19
  3.59 MPI Rank 20
  3.02 MPI Rank 21
  3.47 MPI Rank 22
  2.50 MPI Rank 23
0.00 node jrc0103
  1.39 MPI Rank 24
  0.95 MPI Rank 25
  2.36 MPI Rank 26
  1.26 MPI Rank 27
  1.83 MPI Rank 28
  2.77 MPI Rank 29
  2.62 MPI Rank 30
  2.94 MPI Rank 31
  1.62 MPI Rank 32
  2.21 MPI Rank 33
```

```
0.00    403.90 (1.21%)    33386.79
0.00    403.89 (100.00%)  403.90
0.00    3.59 (0.89%)      403.89
```

Selected "Late Sender"

Figure 2.2: Screenshot of a trace analysis result in the Cube report browser.

3 Command reference

This chapter provides a detailed reference for the most commonly used commands provided by the Scalasca Trace Tools package. For each command, it explains its purpose, usage, options, and configuration possibilities following a man-page style. Corresponding man pages are also installed under `<prefix>/share/man` (unless configured differently) which can be made available to the `man` command by adding the path to the `$MANPATH` environment variable.

3.1 scalasca – Scalasca information and proxy command

SYNOPSIS

scalasca [*OPTIONS*]
scalasca [*OPTIONS*] *ACTION ACTION_ARGS*

DESCRIPTION

The **scalasca** command allows to query information about a Scalasca Trace Tools installation, but can also be used as a proxy command for measurement, analysis, post-processing, and report examination tasks.

When used without an *ACTION* argument, the **scalasca** command provides various *OPTIONS* to query and display information about the Scalasca Trace Tools installation, such as the version number, the configuration summary report, or installation-dependent path information. It is also possible to display a quick reference guide (if a suitable PDF viewer is found on the system).

By contrast, if an *ACTION* argument is given, the **scalasca** command acts as a proxy which simply delegates the requested operation to the underlying specific command. It can therefore be considered a single-entry-point convenience command that is easy to remember.

ACTIONS

Since *ACTION* arguments enable the delegation of operations, please refer to the documentation of the respective underlying command for possible *ACTION_ARGS*. Also, note that *ACTION_ARGS* have to be provided **after** the *ACTION* argument.

-a, --analyze, --analyse
Run application under the control of the Scalasca measurement collection and analysis nexus **scan**(1).

-e, --examine
Examine measurement analysis report using the Scalasca analysis report explorer **square**(1).

Deprecated actions

-i, -inst, --instrument
Prepare application objects and executable for measurement using the Score-P instrumenter. This action only exists for backward compatibility with the Scalasca 1.x release series.
Users are strongly encouraged to use the 'scorep' instrumenter command directly to take full advantage of its improved functionality.

OPTIONS

-c, --show-config
Print configuration summary, then exit.

-h, --help
Print a brief usage summary, then exit.

-n, --dry-run
Print the command(s) to be launched, but do not execute them.

--quickref
Display quick reference guide, then exit.

--remap-specfile
Show path to remapper specification file used for trace-analysis report post-processing, then exit.

-v, --verbose
Enable verbose mode.

-V, --version
Print version information, then exit.

EXIT STATUS

scalasca exits with status 0 on success, and greater than 0 if errors occur.

EXAMPLES

scalasca --version
Displays the Scalasca Trace Tools version number.

scalasca --analyze mpiexec -n 4 foo args
Execute the instrumented MPI program *foo* with command-line arguments *args*, collecting a runtime summary (default). Results in an experiment directory *scorep_foo_4_sum*.

scalasca --examine -s -f filter scorep_foo_4_sum
Post-process measurement reports in *scorep_foo_4_sum* and generate a score report with the run-time measurement filter rules from the file *filter* applied.

SEE ALSO

scan(1), **square**(1)

3.2 scan – Scalasca measurement collection and analysis nexus

SYNOPSIS

scan [*OPTIONS*] [*LAUNCHER* [*LAUNCHER_ARGS*]] *TARGET* [*TARGET_ARGS*]

DESCRIPTION

scan, the Scalasca measurement collection and analysis nexus, manages the configuration and processing of performance experiments with an executable *TARGET*. *TARGET* needs to be instrumented beforehand using the Score-P instrumentation and measurement system. In particular, **scan** integrates the following steps:

- Measurement configuration
- Application execution using any given arguments *TARGET_ARGS*
- Collection of measured data
- Automatic post-mortem trace analysis (if configured)

Many different experiments can typically be performed with a single instrumented executable without needing to re-instrument, by using different measurement and analysis configurations. The default runtime summarization mode directly produces an analysis report for examination, whereas event trace collection and analysis are automatically done in two steps to produce a profile augmented with additional metrics.

Serial and multi-threaded programs are typically launched directly, whereas MPI and hybrid MPI+X programs usually require a special *LAUNCHER* command such as **mpiexec**, which may need additional arguments *LAUNCHER_ARGS* (e.g., to specify the number of processes to be created). **scan** automatically recognizes many MPI launchers, but if not, the MPI launcher name can be specified using the environment variable **SCAN_MPI_LAUNCHER** (see **ENVIRONMENT**).

scan examines the executable *TARGET* to determine whether Score-P instrumentation is present; otherwise the measurement is aborted. The number of MPI processes and OpenMP threads are determined from *LAUNCHER_ARGS* and/or the environment. If the target executable is not specified as one of the launcher arguments, it is expected to be the immediately following part of the command line. It may be necessary to use a double-dash specification (--) to explicitly separate the target from the preceding launcher specification. If there is an imposter executable or script, e.g., often used to specify placement/thread binding, that precedes the instrumented *TARGET*, it may be necessary to explicitly identify the target with the environment variable **SCAN_TARGET** (see **ENVIRONMENT**).

A unique directory is created for each measurement experiment, which must not already exist when measurement starts unless **SCAN_OVERWRITE** is enabled (see **ENVIRONMENT**); otherwise measurement is aborted. A default name for each measurement archive directory is created from a `'scorep_'` prefix, the name of the executable *TARGET*, the run configuration (e.g., number of processes specified), and the measurement configuration. This default name can be overwritten using the **SCOREP_EXPERIMENT_DIRECTORY** environment variable (see **ENVIRONMENT**) or the **-e** command-line option.

When measurement has completed, the measurement archive directory contains all artifacts produced by the measurement and subsequent trace analysis (if configured). In particular, the following files are produced independent from the selected measurement mode:

- `MANIFEST.md`: a text file briefly describing the directory contents produced by the Score-P measurement system
- `scorep.cfg`: a copy of the measurement configuration
- `scorep.log`: the measurement log file
- `scorep.filter`: a copy of the measurement filter (if provided)

In runtime summarization mode, the archive directory additionally contains:

- `profile.cubex`: the runtime summary result

In trace collection and analysis mode, the following additional files are generated:

- an OTF2 trace archive consisting of
  - the anchor file `traces.otf2`,
  - the global definitions file `traces.def`, and
  - the per-process data files in the `traces/` directory
- `scout.log`: the trace analysis log file
- `scout.cubex`: the trace analysis result
- `trace.stat`: trace analysis pattern statistics

In multi-run mode, the results of the individual runs are stored in subdirectories inside the top-level measurement archive directory. In addition, the following file will be archived:

- `scalasca_run.cfg`: a (possibly auto-generated) copy of the multi-run configuration specification file

OPTIONS

The **scan** command accepts the following command-line options. Note that configuration settings specified on the command line take precedence over those specified via environment variables (see **ENVIRONMENT**). Also, see **MULTI-RUN EXPERIMENTS** below for details on interactions with configuration file settings.

-h

Print a brief usage summary, then exit.

-v

Increase verbosity.

-n

Print the command(s) to be launched, but do not execute them.

-q

Quiescent execution with neither summarization nor tracing enabled. Sets both **SCOREP_ENABLE_PROFILING** and **SCOREP_ENABLE_TRACING** to 'false'.

-s

Enable runtime summarization mode. Sets **SCOREP_ENABLE_PROFILING** to 'true'. This is the default.

-t
Enable trace collection and analysis. Sets **SCOREP_ENABLE_TRACING** to 'true'.

-a
Skip measurement step to (re-)analyze an existing trace.

-e *experiment_dir*
Override default experiment archive name to generate and/or analyze *experiment_dir*. Sets **SCOREP_EXPERIMENT_DIRECTORY**.

-f *filter_file*
Use the measurement filter rules from *filter_file*. Sets **SCOREP_FILTERING_FILE**.

-l *lock_file*
Block start of measurement while *lock_file* exists.

-R *num_runs*
Specifies the number measurement runs per configuration (default=1).

-M *config_file*
Allows to specify a configuration file describing multi-run experiment settings. See **MULTI-RUN EXPERIMENTS** below for details.

-P *preset*
Specify a preset for a multi-run measurement, e.g., 'pop'. See **MULTI-RUN EXPERIMENTS** below for details.

-L
List all available multi-run presets.

-D *config_file*
Checks a multi-run config file for validity, if successful dumps the processed configuration for comparison, and exits.

ENVIRONMENT

Environment variables with the 'SCAN_' prefix may be used to configure the **scan** nexus itself (which is a serial workflow manager process), rather than the instrumented application process(es) which will be measured, which can also be configured via environment variables. Configuration specified on the nexus command-line takes precedence over that specified via environment variables. See **MULTI-RUN EXPERIMENTS** below for details on interactions with configuration file settings.

Environment variables controlling scan

SCAN_ANALYZE_OPTS
Specifies trace analyzer options (default: none). For details on the supported options, see **scout**(1).

SCAN_CLEAN
If enabled, removes event trace data after successful trace analysis (default: 'false').

SCAN_MPI_LAUNCHER
Specifies a non-standard MPI launcher name.

SCAN_MPI_RANKS

Specifies the number of MPI processes, for example in an MPMD use case or if the number of ranks is not automatically identified correctly. The specified number will also be used in the automatically generated experiment title. While an experiment title with an incorrect number of processes is harmless (though generally confusing), the correct number is required for automatic parallel trace analysis.

SCAN_MULTIRUN_DEFAULT_CFG

Path to a multi-run configuration file that will be loaded per default in any measurement based on configuration files or presets. As a default settings file, only global settings are used to avoid interference with explicitly specified configs or presets.

SCAN_MULTIRUN_PRESET_PATH

Colon-separated list of paths containing preset files.

SCAN_OVERWRITE

If enabled, removes an existing experiment archive directory before measurement (default: 'false').

SCAN_SETENV

If environment variables are not automatically forwarded to MPI processes by the launcher, one can specify the syntax that the launcher requires for this as **SCAN_SETENV**. For example, "-foo" results in passing "-foo key val" to the launcher, while "–foo=" results in "–foo key=val".

SCAN_TARGET

If there is an imposter executable or script, for example, used to specify placement/thread binding, that precedes the instrumented target, it may be necessary to explicitly identify the target executable by setting **SCAN_TARGET** to the executable name.

SCAN_TRACE_ANALYZER

Specifies an alternative trace analyzer to be used (e.g., `scout.mpi` or `scout.hyb`). If '`none`' is specified, automatic trace analysis is skipped after measurement.

SCAN_TRACE_FILESYS

Specifies an optional list of colon separated paths identifying suitable file systems for tracing. If set, the file system of trace measurements has to match at least one of the specified file systems.

SCAN_WAIT

Time in seconds to wait for synchronization of a distributed filesystem after measurement completion.

Common Score-P environment variables controlling the measurement

SCOREP_EXPERIMENT_DIRECTORY

Explicit experiment archive title.

SCOREP_ENABLE_PROFILING

Enable or disable runtime summarization.

SCOREP_ENABLE_TRACING

Enable or disable event trace generation.

SCOREP_FILTERING_FILE

Name of run-time measurement filter file.

SCOREP_VERBOSE

Controls the generation of additional (debugging) output from the Score-P measurement system.

SCOREP_TOTAL_MEMORY

Size of per-process memory in bytes reserved for Score-P.

For further details, please refer to the Score-P documentation and/or the output of '`scorep-info config-vars`'.

MULTI-RUN EXPERIMENTS

scan also provides means to automate the generation of multiple measurements with varying configuration settings. This workflow can be employed for various analysis objectives, as long as the variations are based on environment variables. Likely candidates are:

1. Increasing the statistical significance through multiple repetitions of measurements with identical settings.
2. Spreading multiple hardware-counter measurements over different runs to limit the measurement overhead and/or to overcome hardware limitations (e.g., number of hardware performance counters that can be measured simultaneously).
3. Performing a series of measurements with varying application settings, like problem size or input data.

Results of such multi-run experiments can be used individually, aggregated manually using various Cube tools, or be passed to the **square**(1) command for automated report aggregation.

Attention:

The degree of non-determinism in an application's runtime behavior will influence the informative value of any aggregated result. Only with sufficient similarity between application runs will the combination of results be useful.

Multi-run experiments are set up using a plain-text configuration file, which is passed to the **scan** command via the **-M** command-line option. In this file, the begin of each measurement run configuration is marked by a line starting with a single dash (`-`) character; the remainder of the line will be ignored. Subsequent lines up to either the next run separator or the end of the file may contain at most one variable setting of the form '`VARIABLE=VALUE`'. Optionally, a section with global settings can be specified at the beginning of the config file, introduced by a line starting with two dashes (`--`); the remainder of this line will again be ignored. A variable defined in the global section will be applied in all subsequent run configurations unless it is overwritten by a run-specific setting. The configuration file format also allows for single-line comments starting with a hash character (#) and blank lines, both of which are ignored.

For example, the following multi-run configuration file defines a series of four subsequent measurements with different settings:

```
# example run configuration file
# global section
-- this can also hold comments
SCOREP_ENABLE_TRACING=true

-
# first run with two PAPI metrics
```

```
SCOREP_METRIC_PAPI=PAPI_TOT_CYC,PAPI_TOT_INS

-
# second run with different PAPI metric and increased Score-P memory
SCOREP_METRIC_PAPI=PAPI_LD_INS
SCOREP_TOTAL_MEMORY=42M

- third run with different PAPI metric
SCOREP_METRIC_PAPI=PAPI_VEC_DP

-
# fourth run using only global settings
```

Note that measurement configuration settings are not limited to **scan** or Score-P environment variables, but also allow for setting arbitrary variables in the measurement execution environment. Also, the order in which measurements are specified may have an impact on the aggregated result, see **square**(1) for details.

To ensure consistency and reproducibility, the environment must not contain Score-P or Scalasca variables when using a multi-run configuration file. Otherwise, **scan** will abort with an error providing a list of the offending variables. That is, all Score-P/Scalasca settings to be applied have to be placed in either the global or run-specific sections of the configuration. Moreover, all variables used anywhere in the configuration file will be unset before each measurement run, and then set to either the global or run-specific value if applicable, thus avoiding side effects from variable settings not specified in the configuration file. The Score-P variable SCOREP_EXPERIMENT_DIRECTORY will not have any effect inside the configuration file, as an automatic naming scheme—an extension to the default Scalasca scheme—is enforced to keep the multi-run measurement directories consistent. To set the experiment directory a priori, the **scan** command-line option **-e** can be used. Other **scan** options that control the measurement (**-q**, **-t**, and **-s**) will be ignored when used with a config file and should be set through the respective environment variables in the configuration file for consistency.

A variation of the configuration file mode described above is the preset mode. The preset mode combines predefined configurations for typical scenarios with global environment variables for the specific use case. In this case, only variables used in the preset configuration are blocked in the global environment to avoid interference with the functionality provided by the selected preset. Available presets can be listed by using the **-L** option. By default, only presets provided by the Scalasca installation are available. Additional presets can be provided by adding paths to the SCAN_MULTIRUN_PRESET_PATH variable using a colon as separator. Preset files have the extension .preset and follow the same syntax as multi-run configuration files. To ensure the functionality of a preset in the presence of additional user-defined Score-P/Scalasca variables, the preset configuration file has to contain all variables that may interfere with the functionality of the preset by using default settings.

In addition to multi-run experiments with different configuration settings, **scan** supports repeating a single or a set of measurements multiple times via the **-R** command-line option, for example, to provide increased statistical significance. For measurements without a configuration file, the measurement will be repeated the requested number of times with the current environment. In case of multi-run configurations, each individual run will be repeated the given number of times with the specified configuration.

For multi-run experiments, **scan** creates a common directory which contains the result of each individual measurement run stored in a subdirectory. The name of the base directory and the experiment directories contains the number of configurations as well as the number of repetitions. To support reproducibility, the configuration used is stored in the file `scalasca_run.cfg` in the common base directory. To test the validity of a configuration file before running a measurement, **scan** provides the **-D** option. In this mode, the provided configuration file is parsed and, on success, the processed data is dumped for comparison.

To store commonly used system- or user-specific variables, the user can specify a default configuration file via `SCAN_MULTIRUN_DEFAULT_CFG`. Its global settings will be used in any configuration- or preset-based multi-run measurement. Note that only the global settings are used to avoid interference with explicitly specified files by adding additional runs to the measurement.

EXIT STATUS

scan exits with status 0 if measurement and automatic trace analysis (if configured) were successful, and greater than 0 if errors occur.

NOTES

While parsing the arguments, unrecognized flags might be reported as ignored, and unrecognized options with required arguments might need to be quoted.

Instrumented applications can still be run without using **scan** to generate measurements, however, measurement configuration is then exclusively via Score-P environment variables (which must be explicitly exported to MPI processes) and trace analysis is not automatically started after event trace collection.

EXAMPLES

scan mpiexec -n 4 foo args
Execute the instrumented MPI program *foo* with command-line arguments *args*, collecting a runtime summary (default). Results in an experiment directory *scorep_foo_4_sum*.

OMP_NUM_THREADS=3 scan -s mpiexec -n 4 foobar
Execute the instrumented hybrid MPI+OpenMP program *foobar*, collecting a runtime summary (default, but explicitly requested). Results in an experiment directory *scorep_foobar_4x3_sum*.

OMP_NUM_THREADS=3 scan -q -t -f filter bar
Execute the instrumented OpenMP program *bar*, collecting only an event trace with the run-time measurement filter *filter* applied. Trace collection is immediately followed by Scalasca's automatic trace analysis. Results in an experiment directory *scorep_bar_Ox3_trace*.

SEE ALSO

scalasca(1), **square**(1), **scout**(1)

3.3 square – Scalasca analysis report explorer

SYNOPSIS

square [*OPTIONS*] (*EXPERIMENT_DIR* | *CUBE_FILE*)

DESCRIPTION

square, the Scalasca analysis report explorer, facilitates post-processing, scoring, and interactive examination of analysis reports from both runtime summarization and tracing experiments.

When provided with a Score-P experiment directory *EXPERIMENT_DIR*, **square** post-processes intermediate analysis reports produced by a measurement and/or an automatic trace analysis to derive additional metrics and construct a hierarchy of measured and derived metrics, and then presents this final report using the Cube GUI (unless the **-s** option is used). If intermediate reports were already processed, the final report is shown immediately. If there is more than one analysis report in a Score-P experiment directory, the most comprehensive report is shown by default.

When provided with the name of a specific analysis report *CUBE_FILE*, post-processing is skipped and the corresponding report is shown immediately.

Analysis report examination can only be done after measurement and analysis are completed. Parallel resources are not required, and it is often more convenient to examine analysis reports on a different system, such as a desktop computer where interactivity is superior.

Depending on the measurement configuration and the provided options, **square** places additional files into the experiment archive directory. For single-run experiments, the following files are created if the corresponding input files are available:

- `summary.cubex`: post-processed runtime summary result
- `trace.cubex`: post-processed trace analysis result

In scoring mode (**-s** option), **square** generates:

- `scorep.score`: detailed measurement score report, optionally suffixed with the name of a provided filter file (**-f** option)

In multi-run mode, aggregated reports are created if the corresponding input files are available:

- `profile_aggr.cubex`: aggregated runtime summary result
- `scout_aggr.cubex`: aggregated trace analysis result
- `scout+profile.cubex`: merged runtime summary and trace analysis result
- `summary_aggr.cubex`: post-processed aggregated runtime summary result
- `trace_aggr.cubex`: post-processed aggregated trace analysis result
- `trace+summary.cubex`: post-processed merged runtime summary and trace result

OPTIONS

Common options

-C *LEVEL*

Level of sanity checks for newly created reports (default: '**none**'). '**quick**' performs various sanity checks on the experiment meta data, while '**full**' also executes a more time-consuming check for negative metric values (which usually indicate a serious error).

-c *num_counter*

Specifies the number of hard- and software counters that shall be considered when generating a score report (option -s). By default, this value is 0, which means that only a timestamp is measured on each event. If you plan to record extra counters specify the number of counters. Otherwise, scoring may underestimate the required space.

-F

Force post-processing of analysis reports, even if a post-processed report already exists.

-f *filter_file*

Apply the specified filter file when generating a score report (option -s).

-s

Output a textual score report. Skips launching the Cube GUI.

-v

Enable verbose mode.

-n

Suppress the calculation of 'Idle Threads' metric.

-x <*scorep-score opt*>

Pass option(s) directly to **scorep-score**. Any composite options have to be quoted as needed.

Options for multi-run experiments

-S *MODE*

Set aggregation mode for runtime summarization results of each configuration. Currently supported modes are '**mean**' and '**merge**' (default).

-T *MODE*

Set aggregation mode for trace analysis results of each configuration. Currently supported modes are '**mean**' and '**merge**' (default).

-A

Force post-processing of every individual step report of a multi-run experiment.
WARNING: Depending on the number and size of the individual measurement reports, the time required to post-process all reports can be significant!

-I

Ignore structural sanity checks and force aggregation of measurements in a multi-run experiment.

MULTI-RUN EXPERIMENTS

For multi-run experiments, **square** provides additional options to aggregate the set of measurement results into a single Cube file. The user can choose between the two aggregation methods '`merge`' and '`mean`' to combine results from different configurations, which underneath use the corresponding CubeLib command-line tools. The default aggregation mode is to '`merge`' results.

Note:

The '`merge`' operation always copies metric data from the last measurement configuration in a given set in which data for a particular metric is available. This should be taken into account when setting up a multi-run experiment that is supposed to be aggregated using the **square** command later on. In particular, it is recommended to include a low-overhead measurement without hardware performance counters at the end of a measurement configuration set including hardware counter measurements in order to provide more accurate time information.

The aggregation of multi-run measurement results happens in the following order:

1. Aggregate results from multiple runs for each measurement configuration. At this point, the only supported mode for this aggregation is '`mean`', which is therefore hard-coded.
2. Aggregate averaged runtime summarization results from all configurations in ascending order using the selected mode (**-S** option).
3. Aggregate averaged trace analysis analysis results from all configurations in ascending order using the selected mode (**-T** option).
4. Merge the aggregated runtime summarization and trace analysis results into a combined report.
5. Post-process the combined report (step 4) if available, otherwise post-process the aggregated report(s) generated in either step 2 or step 3.

Depending on the measurement settings, those steps will be applied if the respective intermediate results are found. Before merging intermediate results, **square** performs sanity checks to compare the call-tree structure to ensure merging will result in a valid Cube file. In rare cases, where the user is aware of potential call-tree differences, it may be necessary to skip these checks, which can be accomplished by passing the **-I** option. However, note that this may produce erroneous or at least misleading results. The reports of the individual runs will only be post-processed when explicitly requested (**-A** option).

EXIT STATUS

square exits with status 0 on success, and greater than 0 if errors occur.

NOTES

To examine an analysis report on a different system, for example, a desktop or laptop computer, it is often best to post-process the report using **square**'s scoring functionality (**-s** option) on the system where the measurement has been taken, and then copy over the resulting post-processed Cube file. This is because **square** requires various command-line tools and support files from the Score-P, CubeLib, and Scalasca Trace Tools packages, which may not be available on the target computer.

EXAMPLES

square scorep_foo_4_trace
Post-process measurement reports in *scorep_foo_4_trace* and display the most comprehensive report using the Cube GUI.

square -s -f filter scorep_foo_4_sum
Post-process measurement reports in *scorep_foo_4_sum* and generate a score report with the run-time measurement filter rules from the file *filter* applied.

square -S mean scorep_foo_4_multi-run_c2_r4
Aggregate and post-process the measurement results of the multi-run experiment with two configurations and four runs per configuration stored in *scorep_foo_4_multi-run_c2_r4*. Then, show the most comprehensive report using the Cube GUI.

SEE ALSO

scalasca(1), **scan**(1)

3.4 scout – Scalasca parallel trace analyzer

SYNOPSIS

scout.ser [*OPTION*] (*ANCHOR_FILE* | *EXPERIMENT_DIR*)
scout.omp [*OPTION*] (*ANCHOR_FILE* | *EXPERIMENT_DIR*)
scout.mpi [*OPTION*] (*ANCHOR_FILE* | *EXPERIMENT_DIR*)
scout.hyb [*OPTION*] (*ANCHOR_FILE* | *EXPERIMENT_DIR*)

DESCRIPTION

scout is the scalable automatic trace-analysis component of the Scalasca Trace Tools. In particular, it provides the ability to

- identify wait states in communication and synchronization operations that occur, for example, as a result of unevenly distributed workloads,
- pinpoint the root causes of those wait states (i.e., delays), and
- identify the activities on the critical path of the target application, highlighting those routines which determine the length of the program execution and therefore constitute the best candidates for optimization.

Usually, **scout** is launched automatically by the Scalasca measurement collection and analysis nexus **scan**(1) after a successful measurement if event tracing is configured. However, it can also be run manually on an existing event trace measurement.

scout currently supports trace experiments in two different event trace formats: OTF2 traces generated by the Score-P instrumentation and measurement system, and legacy traces in EPILOG format generated by the measurement system of the Scalasca 1.x release series. For OTF2 event traces, **scout** has to be provided with the corresponding *ANCHOR_FILE* (e.g., '`traces.otf2`'), for EPILOG traces with the experiment directory name *EXPERIMENT_DIR*.

Depending on the build configuration and the capabilities of the target platform, the **scout** analyzer may be available in up to four forms:

scout.ser
is always built. It is used to analyze event traces generated by serial applications. It can also be used to analyze event traces from multi-threaded applications, however, it will then only provide information about the master thread.

scout.omp
is built whenever the Scalasca Trace Tools are configured with OpenMP support. It is used to analyze event traces generated by pure multi-threaded applications (e.g., using OpenMP or POSIX threads). It can also be used to analyze event traces from serial applications, though analysis incurs a higher overhead than using **scout.ser**.

scout.mpi
is built whenever the Scalasca Trace Tools are configured with MPI support. It is used to analyze event traces generated by pure MPI applications. It can also be used on traces from multi-threaded MPI applications, however, it will then only provide information about the master thread of each process and its MPI activities.

scout.hyb
is built if the Scalasca Trace Tools are configured with both MPI and OpenMP support. It is used to analyze event traces generated by multi-threaded MPI applications (e.g., MPI+OpenMP or MPI+Pthreads), providing information about all OpenMP/POSIX threads of each MPI process. It can also be used on traces from pure MPI applications, though analysis incurs a slightly higher overhead than using **scout.mpi**.

Note that **scout.mpi** and **scout.hyb** are implemented as MPI programs, and therefore have to be executed using appropriate MPI launch commands and flags. Also, the number of MPI processes for **scout** must be identical to the number of MPI processes used for the target application execution.

If successful, **scout** produces the following output files in the measurement archive directory:

- `scout.cubex`: the trace analysis result
- `trace.stat`: trace analysis pattern statistics

OPTIONS

scout accepts a number of command-line options to enable/disable particular analysis features. When **scout** is launched automatically from the Scalasca measurement collection and analysis nexus **scan**(1), these options can be passed to the analyzer via the **SCAN_ANALYZE_OPTS** environment variable.

Common options

--statistics
Enables most-severe instance tracking and wait-state statistics. This is the default.

--no-statistics
Disables most-severe instance tracking and wait-state statistics.

--critical-path
Enables critical-path analysis. This is the default.

--no-critical-path
Disables critical-path analysis.

--rootcause
Enables root-cause analysis. This is the default.

--no-rootcause
Disables root-cause analysis.

--single-pass
Use single-pass forward analysis only. Disables both critical-path and root-cause analysis, as well as the detection of *Late Receiver* wait states.

-v, --verbose
Increase verbosity.

--help
Print a brief usage summary, then exit.

MPI options (scout.mpi/scout.hyb only)

--time-correct

Enables enhanced timestamp correction. Event traces collected on clusters without a synchronized clock may contain logical clock condition violations (such as a receive completing before the corresponding send is initiated). When **scout** detects such situations, it issues a warning that the analysis may be inconsistent and recommends (re-)running trace analysis with its integrated timestamp correction algorithm activated.

--no-time-correct

Disables enhanced timestamp correction. This is the default.

EXIT STATUS

scout exits with status 0 if automatic trace analysis was successful, and greater than 0 if errors occur.

NOTES

scout poses a number of requirements on the input event trace data, which are documented in the *OPEN_ISSUES* file installed as part of the Scalasca Trace Tools documentation. It is also available online at [16]. If those requirements are not met, **scout** may abort, deadlock, or crash.

If **scout** crashes or deadlocks even though the documented requirements are met (which usually indicates a bug), restricting the scope of the analysis by disabling certain features (e.g., critical-path and/or root-cause analysis) may help as a workaround. In any case, please report such issues for further investigation (see Chapter 4).

EXAMPLES

scout.omp scorep_foo_Ox4_trace/traces.otf2
Perform the Scalasca OpenMP event trace analysis on the OTF2 event trace with anchor file *scorep_foo_Ox4_trace/traces.otf2*.

mpiexec -n 16 scout.mpi --time-correct scorep_bar_16_trace/traces.otf2
Apply the enhanced timestamp correction and perform the Scalasca MPI event trace analysis on the OTF2 event trace with anchor file *scorep_bar_16_trace/traces.otf*.

mpiexec -n 4 scout.hyb epik_foobar_4x4_trace
Perform the hybrid Scalasca MPI+OpenMP event trace analysis on the EPILOG event trace in the experiment archive *epik_foobar_4x4_trace* generated by the Scalasca 1.x release series.

SEE ALSO

scalasca(1), **scan**(1), **square**(1)

4 Reporting bugs

Like every other software, the Scalasca Trace Tools may contain bugs. If you encounter obviously broken, weird, or otherwise unexplainable behavior, please report it to `scalasca@fz-juelich.de`.

Before doing so, however, please check the *OPEN_ISSUES* file installed as part of the Scalasca Trace Tools documentation (also available online at [16]) whether the issue is already known. If not, make sure to include at least the following information in your bug report:

- The Scalasca Trace Tools version reported by '`scalasca --version`'.
- The Scalasca Trace Tools configuration reported by '`scalasca --show-config`'.
- The Score-P version reported by '`scorep --version`'.
- The Score-P configuration reported by '`scorep-info config-summary`'.
- The exact command line of the failing command.
- The exact failure/error message.

Also, if the trace analysis fails, please archive a copy of the entire experiment archive directory including the event trace data, as this may be required to aid in debugging. However, **ONLY PROVIDE TRACE DATA IF EXPLICITLY REQUESTED**, as the data volume may be excessive.

Bibliography

[1] D. Becker, R. Rabenseifner, F. Wolf, and J. Linford. Scalable timestamp synchronization for event traces of message-passing applications. *Parallel Computing*, 35(12):595–607, Dec. 2009. 24

[2] D. Böhme, B. R. de Supinski, M. Geimer, M. Schulz, and F. Wolf. Scalable critical-path based performance analysis. In *Proc. of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS), Shanghai, China*, pages 1330–1340. IEEE Computer Society, May 2012. 1

[3] D. Böhme, M. Geimer, F. Wolf, and L. Arnold. Identifying the root causes of wait states in large-scale parallel applications. In *Proc. of the 39th International Conference on Parallel Processing (ICPP), San Diego, CA, USA*, pages 90–100. IEEE Computer Society, Sept. 2010. 1

[4] CubeGUI User Guide. Available as part of the Cube installation or online at `https://www.scalasca.org/software/cube-4.x/documentation.html`. 20

[5] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, and F. Wolf. Open Trace Format 2 - The next generation of scalable trace formats and support libraries. In *Applications, Tools and Techniques on the Road to Exascale Computing (Proc. of Intl. Conference on Parallel Computing, ParCo, Aug./Sept. 2011, Ghent, Belgium)*, volume 22 of *Advances in Parallel Computing*, pages 481–490. IOS Press, May 2012. 1

[6] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr. A scalable tool architecture for diagnosing wait states in massively parallel applications. *Parallel Computing*, 35(7):375–388, July 2009. 1

[7] IEEE. *IEEE 1003.1-2017: IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7*. IEEE Computer Society Press, 2017. 1

[8] Khronos Group. The OpenCL specification, Version 3.0. `https://www.khronos.org/registry/OpenCL/`, Dec. 2020. 1

[9] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The Vampir performance analysis toolset. In *Tools for High Performance Computing (Proc. of the 2nd Parallel Tools Workshop, July 2008, Stuttgart, Germany)*, pages 139–155. Springer, July 2008. 1, 6

[10] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf. Score-P – A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Tools for High Performance Computing 2011 (Proc. of 5th Parallel Tools Workshop, Sept. 2011, Dresden, Germany)*, pages 79–91. Springer, Sept. 2012. 1

[11] Message Passing Interface Forum. MPI: A message-passing interface standard, Version 3.1. https://www.mpi-forum.org, June 2015. 1

[12] NASA Advanced Supercomputing Division. NAS Parallel Benchmarks website. https://www.nas.nasa.gov/publications/npb.html. 7

[13] NVIDIA Corporation. CUDA toolkit documentation. https://docs.nvidia.com/cuda/index.html. 1

[14] OpenACC-Standard.org. The OpenACC application programming interface, Version 3.1. https://www.openacc.org, Nov. 2020. 1

[15] OpenMP Architecture Review Board. OpenMP application program interface, Version 5.1. https://www.openmp.org, Nov. 2020. 1

[16] Scalasca 2.x series documentation web page. https://www.scalasca.org/software/scalasca-2.x/documentation.html. 7, 45, 47

[17] Score-P User Manual. Available as part of the Score-P installation or online at https://www.score-p.org. 4, 5, 6, 14, 16

[18] S. S. Shende and A. D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, May 2006. 1

scalasca

JÜLICH
Forschungszentrum

www.scalasca.org