

Scalasca 2.3 | User Guide

Scalable Automatic Performance Analysis

April 2016
The Scalasca Development Team
scalasca@fz-juelich.de

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

Copyright © 1998–2016	Forschungszentrum Jülich GmbH, Germany
Copyright © 2009–2015	German Research School for Simulation Sciences GmbH, Jülich/Aachen, Germany
Copyright © 2014–2015	RWTH Aachen University, Germany
Copyright © 2003–2008	University of Tennessee, Knoxville, USA
Copyright © 2006	Technische Universität Dresden, Germany

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the names of
 - the Forschungszentrum Jülich GmbH,
 - the German Research School for Simulation Sciences GmbH,
 - the RWTH Aachen University,
 - the University of Tennessee,
 - the Technische Universität Dresden,

nor the names of their contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Attention

The Scalasca User Guide is currently being rewritten and still incomplete. However, it should already contain enough information to get you started and avoid the most common pitfalls.

Contents

1	Introduction	1
2	Getting started	3
2.1	Instrumentation	4
2.2	Runtime measurement collection & analysis	5
2.3	Analysis report examination	6
2.4	A full workflow example	7
2.4.1	Preparing a reference execution	8
2.4.2	Instrumenting the application code	11
2.4.3	Initial summary measurement	12
2.4.4	Optimizing the measurement configuration	14
2.4.5	Summary measurement & examination	17
2.4.5.1	Using the Cube browser	19
2.4.6	Trace collection and analysis	21
	Bibliography	27

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.

Jointly developed at the Jülich Supercomputing Centre and the German Research School for Simulation Sciences (Aachen), the Scalasca Tracing Tools are a collection of trace-based performance analysis tools that have been specifically designed for use on large-scale systems such as IBM Blue Gene or Cray XT and successors, but also suitable for smaller HPC platforms. While the current focus is on applications using MPI [8], OpenMP [10] or hybrid MPI+OpenMP, support for other parallel programming paradigms may be added in the future. A distinctive feature of the Scalasca Tracing 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 [5]. Especially when trying to scale communication intensive applications to large process counts, such wait states can present severe challenges to achieving good performance. In addition, the trace analyzer is able 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.

In contrast to previous versions of the Scalasca toolset – which used a custom measurement system and trace data format – the Scalasca Tracing Tools 2.x release series is based on the community-driven instrumentation and measurement infrastructure Score-P [7]. 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) [4] for event trace data. This significantly improves interoperability between Scalasca and other performance analysis tool suites such as Vampir [6] and TAU [13]. 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 Tracing 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 Tracing 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 Tracing Tools 2.x release series leverages the Score-P infrastructure, while the Cube graphical user interface is used for analysis report examination. Scalasca complements the functionality provided by Cube and Score-P with scalable automatic trace-analysis components, as well as convenience commands for controlling execution measurement collection and analysis, and analysis report postprocessing.

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 improved functionality provided by Score-P, **users are strongly encouraged to use the `scorep` instrumenter command directly**. Please refer to the Score-P User Manual [12] for details. 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 postprocess the analysis report generated by a Score-P profiling measurement and/or Scalasca's automatic post-mortem trace analysis, and to start the analysis report examination browser Cube.

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).

Note:

Under the hood, the Scalasca convenience commands leverage a number of other commands provided by Score-P and Cube. Therefore, it is generally advisable to include the executable directories of appropriate installations of all three 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 Tracing 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 detail. Instead, all the necessary instrumentation of user routines, OpenMP constructs and MPI functions 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 using 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.

Attention:

The `scorep` instrumenter 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. **Instrumenting**

files defining OpenMP parallel regions is essential, as Score-P has to track the creation of new threads.

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 with selective instrumentation and measurement filtering. While the most basic steps will be briefly covered in Section 2.4.4, please refer to the Score-P manual [12] 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 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, e.g., to specify the number of compute nodes or MPI ranks. For non-MPI (i.e., serial and pure OpenMP) applications, the launch command and associated 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 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 `OMP_NUM_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". Alternatively, the experiment directory name can also be explicitly specified with the `-e <experiment_name>` option of `scalasca -analyze` or via the environment variable `SCOREP_EXPERIMENT_DIRECTORY`.

Note:

A number of settings regarding the measurement configuration can be specified in different ways. Command-line options provided to `scalasca -analyze` always have highest precedence, followed by Score-P environment variables, and finally automatically determined defaults.

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 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 1).

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 determined with runtime summarization. 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 [6].

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 [12] 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 postprocessed and examined using the `scalasca -examine` (or short square) command, providing an experiment directory name as argument:

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

Postprocessing is done the first time an experiment is examined, before launching the Cube analysis report browser. 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 without launching the viewer:

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

This score report comes from the `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 (`max_buf`), which can be used to set up a filtering file and to determine an appropriate `SCOREP_TOTAL_MEMORY` setting for a subsequent trace measurement. 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 postprocessing 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 general usage workflow of Scalasca based on an abstract example, we will now guide through an example analysis of a moderately complex benchmark code using MPI: BT from the NAS Parallel Benchmarks (NPB-MPI 3.3) [9]. The BT benchmark implements a simulated 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 (available for download on the Scalasca documentation web page [11]) were carried out using Scalasca in combination with Score-P 1.4 and Cube 4.3 on the JUROPA cluster at Jülich Supercomputing Centre. JUROPA's compute nodes are equipped with two Intel Xeon X5570 (Nehalem-EP) quad-core CPUs running at 2.93 GHz, and connected via a QDR Infiniband fat-tree network. The code was compiled using Intel compilers and linked against ParTec ParaStation MPI (which is based on MPICH2). The example commands shown below – which are assumed to be available in `PATH`, e.g., after loading site-specific environment modules – should therefore be representative for using Scalasca in a typical HPC cluster environment.

Note:

Remember that the Scalasca commands use other commands provided by Score-P and Cube. It is assumed that the executable directories of appropriate installations of all three components are available in the shell search path.

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. On the one hand, this step verifies that the code executes cleanly and produces correct results, and on the other hand later allows to assess the overhead introduced by instrumentation and measurement. 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      = mpif77
FFLAGS      = -O2
FLINKFLAGS  = -O2
```

Note that the MPI C compiler wrapper and flags are not used for building BT, but may also be set 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 (for BT, this is required to be a square number) as well as the problem size on the command line:

```
% make bt NPROCS=64 CLASS=D
=====
=      NAS Parallel Benchmarks 3.3      =
=      MPI/F77/C                        =
=====

cd BT; make NPROCS=64 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 64 D
make[2]: Entering directory '/tmp/NPB3.3-MPI/BT'
mpif77 -c -O2 bt.f
mpif77 -c -O2 make_set.f
mpif77 -c -O2 initialize.f
mpif77 -c -O2 exact_solution.f
mpif77 -c -O2 exact_rhs.f
mpif77 -c -O2 set_constants.f
mpif77 -c -O2 adi.f
mpif77 -c -O2 define.f
mpif77 -c -O2 copy_faces.f
mpif77 -c -O2 rhs.f
```

```
mpif77 -c -O2 solve_subs.f
mpif77 -c -O2 x_solve.f
mpif77 -c -O2 y_solve.f
mpif77 -c -O2 z_solve.f
mpif77 -c -O2 add.f
mpif77 -c -O2 error.f
mpif77 -c -O2 verify.f
mpif77 -c -O2 setup_mpi.f
cd ../common; mpif77 -c -O2 print_results.f
cd ../common; mpif77 -c -O2 timers.f
make[3]: Entering directory '/tmp/NPB3.3-MPI/BT'
mpif77 -c -O2 btio.f
mpif77 -O2 -o ../bin/bt.D.64 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'
```

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 single-core laptop with 4 MPI ranks, while the other problem sizes are more suitable for "real" configurations.

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.64`. 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 64 ./bt.D.64
```

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: 64

```
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 = 0.10000000000000E-07
Comparison of RMS-norms of residual
    1 0.2533188551738E+05 0.2533188551738E+05 0.1479210131727E-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 0.8597827149538E-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 0.1582835864248E-12
    3 0.7782212022645E+02 0.7782212022645E+02 0.4053872777553E-13
    4 0.6835623860116E+02 0.6835623860116E+02 0.3762882153975E-13
    5 0.6065737200368E+03 0.6065737200368E+03 0.2474004739002E-13
Verification Successful
```

```
BT Benchmark Completed.
Class          = D
Size           = 408x 408x 408
Iterations     = 250
Time in seconds = 462.95
Total processes = 64
Compiled procs = 64
Mop/s total   = 126009.74
Mop/s/process = 1968.90
Operation type = floating point
Verification   = SUCCESSFUL
Version        = 3.3
Compile date   = 29 Jan 2015
```

```
Compile options:
MPIF77        = mpif77
FLINK         = $(MPIF77)
FMPI_LIB      = (none)
FMPI_INC      = (none)
FFLAGS        = -O2
FLINKFLAGS    = -O2
RAND          = (none)
```

Please send the results of this run to:

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

If email is not available, send this to:

MS T27A-1
NASA Ames Research Center

Moffett Field, CA 94035-1000

Fax: 650-604-3957

Note that this application verified its successful calculation and reported the associated wall-clock execution time 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, when you don't have detailed knowledge about the application and need to identify the hotspots in your 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 mpif77
```

Note that the linker specification variable `FLINK` in `config/make.def` defaults to the value of `MPIF77`, i.e., 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=64 CLASS=D
=====
=      NAS Parallel Benchmarks 3.3      =
=      MPI/F77/C                        =
=====

cd BT; make NPROCS=64 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 64 D
make[2]: Entering directory '/tmp/NPB3.3-MPI/BT'
scorep mpif77 -c -O2 bt.f
```

```
scorep mpif77 -c -O2 make_set.f
scorep mpif77 -c -O2 initialize.f
scorep mpif77 -c -O2 exact_solution.f
scorep mpif77 -c -O2 exact_rhs.f
scorep mpif77 -c -O2 set_constants.f
scorep mpif77 -c -O2 adi.f
scorep mpif77 -c -O2 define.f
scorep mpif77 -c -O2 copy_faces.f
scorep mpif77 -c -O2 rhs.f
scorep mpif77 -c -O2 solve_subs.f
scorep mpif77 -c -O2 x_solve.f
scorep mpif77 -c -O2 y_solve.f
scorep mpif77 -c -O2 z_solve.f
scorep mpif77 -c -O2 add.f
scorep mpif77 -c -O2 error.f
scorep mpif77 -c -O2 verify.f
scorep mpif77 -c -O2 setup_mpi.f
cd ../common; scorep mpif77 -c -O2 print_results.f
cd ../common; scorep mpif77 -c -O2 timers.f
make[3]: Entering directory '/tmp/NPB3.3-MPI/BT'
scorep mpif77 -c -O2 btio.f
scorep mpif77 -O2 -o ../bin/bt.D.64 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 64 ./bt.D.64
S=C=A=N: Scalasca 2.2 runtime summarization
S=C=A=N: ./scorep_bt_64.sum experiment archive
S=C=A=N: Thu Jan 29 22:05:09 2015: Collect start
mpiexec -n 64 [...] ./bt.D.64
```

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: 64

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 = 0.10000000000000E-07
Comparison of RMS-norms of residual
  1 0.2533188551738E+05 0.2533188551738E+05 0.1479210131727E-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 0.8597827149538E-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 0.1582835864248E-12
  3 0.7782212022645E+02 0.7782212022645E+02 0.4053872777553E-13
  4 0.6835623860116E+02 0.6835623860116E+02 0.3762882153975E-13
  5 0.6065737200368E+03 0.6065737200368E+03 0.2474004739002E-13
Verification Successful

```

BT Benchmark Completed.

```

Class          = D
Size           = 408x 408x 408
Iterations     = 250
Time in seconds = 940.81
Total processes = 64
Compiled procs = 64
Mop/s total    = 62006.00
Mop/s/process  = 968.84
Operation type  = floating point
Verification    = SUCCESSFUL
Version         = 3.3
Compile date    = 29 Jan 2015

```

Compile options:

```

MPIF77         = scorep mpif77
FLINK           = $(MPIF77)
FMPI_LIB       = (none)
FMPI_INC       = (none)
FFLAGS         = -O2
FLINKFLAGS     = -O2
RAND           = (none)

```

Please send the results of this run to:

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

If email is not available, send this to:

MS T27A-1
NASA Ames Research Center
Moffett Field, CA 94035-1000

Fax: 650-604-3957

```
S=C=A=N: Thu Jan 29 22:20:59 2015: Collect done (status=0) 950s  
S=C=A=N: ./scorep_bt_64_sum complete.
```

```
% ls scorep_bt_64_sum  
profile.cubex  scorep.cfg  scorep.log
```

As can be seen, the measurement run successfully produced an experiment directory `scorep_bt_64_sum` containing

- 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 about twice as long as the reference run (940.81 vs. 462.95 seconds). That is, instrumentation and associated measurements introduced a non-negligible amount of 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.

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 additional experiments. This can be achieved in various ways, e.g., using runtime filtering, selective recording, or manual instrumentation controlling measurement. Please refer to the Score-P Manual [12] for details on the available options. However, in many cases it is already sufficient to filter a small number of frequently executed but otherwise unimportant user functions to reduce the measurement overhead to an acceptable level. The selection of those routines 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 help 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_64_sum
INFO: Post-processing runtime summarization report...
scorep-score -r ./scorep_bt_64_sum/profile.cubex > ./scorep_bt_64_sum/scorep.score
INFO: Score report written to ./scorep_bt_64_sum/scorep.score
```

```
% head -n 20 scorep_bt_64_sum/scorep.score
```

```
Estimated aggregate size of event trace:          3700GB
Estimated requirements for largest trace buffer (max_buf): 58GB
Estimated memory requirements (SCOREP_TOTAL_MEMORY): 58GB
(hint: When tracing set SCOREP_TOTAL_MEMORY=58GB to avoid intermediate flushes
or reduce requirements using USR regions filters.)
```

flt type	max_buf[B]	visits	time[s]	time[%]	time/ visit[us]	region
ALL	62,076,748,138	152,783,214,921	60774.81	100.0	0.40	ALL
USR	62,073,899,966	152,778,875,273	58840.43	96.8	0.39	USR
MPI	2,267,202	2,909,568	1633.69	2.7	561.49	MPI
COM	580,970	1,430,080	300.69	0.5	210.26	COM
USR	20,525,692,668	50,517,453,756	12552.16	20.7	0.25	binvrhs_
USR	20,525,692,668	50,517,453,756	8069.43	13.3	0.16	matmul_sub_
USR	20,525,692,668	50,517,453,756	6308.60	10.4	0.12	matvec_sub_
USR	447,119,556	1,100,528,112	130.68	0.2	0.12	exact_solution_
USR	50,922,378	124,121,508	19.78	0.0	0.16	binvrhs_
MPI	855,834	771,456	11.17	0.0	14.48	MPI_Isend
MPI	855,834	771,456	5.16	0.0	6.69	MPI_Irecv

As can be seen from the top of the score output, 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 62 GB (~58 GiB). Considering the 24 GiB of main memory available on the JUROPA compute nodes and the 8 MPI ranks per node, 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. Currently, the following groups are distinguished:

- **MPI**: MPI API functions.
- **OMP**: OpenMP constructs and API functions.
- **COM**: User functions/regions that appear on a call path to an OpenMP construct, or an OpenMP or MPI API function. Useful to provide the context of MPI/OpenMP usage.
- **USR**: User functions/regions that do not appear on a call path to an OpenMP construct, or an OpenMP or MPI API function.

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`, `matmul_sub` and `binvrhs`, 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 250 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 447 MB 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 [12] 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.

The effectiveness of this filter can be examined by scoring the initial summary report again, this time 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_64_sum
scorep-score -f npb-bt.filt -r ./scorep_bt_64_sum/profile.cubex \
    > ./scorep_bt_64_sum/scorep.score_npb-bt.filt
INFO: Score report written to ./scorep_bt_64_sum/scorep.score_npb-bt.filt

% head -n 25 scorep_bt_64_sum/scorep.score_npb-bt.filt
```

```
Estimated aggregate size of event trace:          3298MB
Estimated requirements for largest trace buffer (max_buf): 53MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY): 55MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=55MB to avoid intermediate flushes
or reduce requirements using USR regions filters.)
```

flt type	max_buf[B]	visits	time[s]	time[%]	time/ visit[us]	region
- ALL	62,076,748,138	152,783,214,921	60774.81	100.0	0.40	ALL
- USR	62,073,899,966	152,778,875,273	58840.43	96.8	0.39	USR
- MPI	2,267,202	2,909,568	1633.69	2.7	561.49	MPI
- COM	580,970	1,430,080	300.69	0.5	210.26	COM
* ALL	54,527,956	130,325,541	33713.95	55.5	258.69	ALL-FLT
+ FLT	62,024,197,560	152,652,889,380	27060.86	44.5	0.18	FLT
* USR	51,679,784	125,985,893	31779.57	52.3	252.25	USR-FLT
- MPI	2,267,202	2,909,568	1633.69	2.7	561.49	MPI-FLT
* COM	580,970	1,430,080	300.69	0.5	210.26	COM-FLT

+	USR	20,525,692,668	50,517,453,756	12552.16	20.7	0.25	binvrhs_
+	USR	20,525,692,668	50,517,453,756	8069.43	13.3	0.16	matmul_sub_
+	USR	20,525,692,668	50,517,453,756	6308.60	10.4	0.12	matvec_sub_
+	USR	447,119,556	1,100,528,112	130.68	0.2	0.12	exact_solution_
-	USR	50,922,378	124,121,508	19.78	0.0	0.16	binvrhs_
-	MPI	855,834	771,456	11.17	0.0	14.48	MPI_Isend

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 3.3 GiB, and the process-local maximum across all ranks is reduced to 53 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 55 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 not overwrite the existing experiment directory and abort immediately.

```
% mv scorep_bt_64_sum scorep_bt_64_sum.nofilt
% scalasca -analyze -f npb-bt.filt mpiexec -n 64 ./bt.D.64
S=C=A=N: Scalasca 2.2 runtime summarization
S=C=A=N: ./scorep_bt_64_sum experiment archive
S=C=A=N: Thu Jan 29 22:20:59 2015: Collect start
mpiexec -n 64 [...] ./bt.D.64
```

```
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: 64
```

```
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 = 0.10000000000000E-07
Comparison of RMS-norms of residual
  1 0.2533188551738E+05 0.2533188551738E+05 0.1479210131727E-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 0.8597827149538E-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 0.1582835864248E-12
  3 0.7782212022645E+02 0.7782212022645E+02 0.4053872777553E-13
  4 0.6835623860116E+02 0.6835623860116E+02 0.3762882153975E-13
  5 0.6065737200368E+03 0.6065737200368E+03 0.2474004739002E-13
Verification Successful
```

BT Benchmark Completed.

Class	=	D
Size	=	408x 408x 408
Iterations	=	250
Time in seconds	=	477.66
Total processes	=	64
Compiled procs	=	64
Mop/s total	=	122129.54
Mop/s/process	=	1908.27
Operation type	=	floating point
Verification	=	SUCCESSFUL
Version	=	3.3
Compile date	=	29 Jan 2015

Compile options:

MPIF77	=	scorep mpif77
FLINK	=	\$(MPIF77)
FMPI_LIB	=	(none)
FMPI_INC	=	(none)
FFLAGS	=	-O2
FLINKFLAGS	=	-O2
RAND	=	(none)

Please send the results of this run to:

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

If email is not available, send this to:

MS T27A-1
NASA Ames Research Center
Moffett Field, CA 94035-1000

Fax: 650-604-3957

```
S=C=A=N: Thu Jan 29 22:29:02 2015: Collect done (status=0) 483s  
S=C=A=N: ./scorep_bt_64_sum complete.
```

Notice that applying the runtime filtering reduced the measurement overhead significantly, down to now only 3% (477.66 seconds vs. 462.95 seconds for the reference run). This new measurement with the optimized configuration should therefore accurately represent the real runtime behavior of the BT application, and can now be postprocessed and interactively explored using the Cube result browser. These two steps can be conveniently initiated using the `scalasca -examine` command:

```
% scalasca -examine scorep_bt_64_sum  
INFO: Post-processing runtime summarization report...  
INFO: Displaying ./scorep_bt_64_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 97% of the overall CPU allocation time is spent executing computational user functions, while 2.6% 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`. Execution 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.

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 Cube User Guide [3].

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

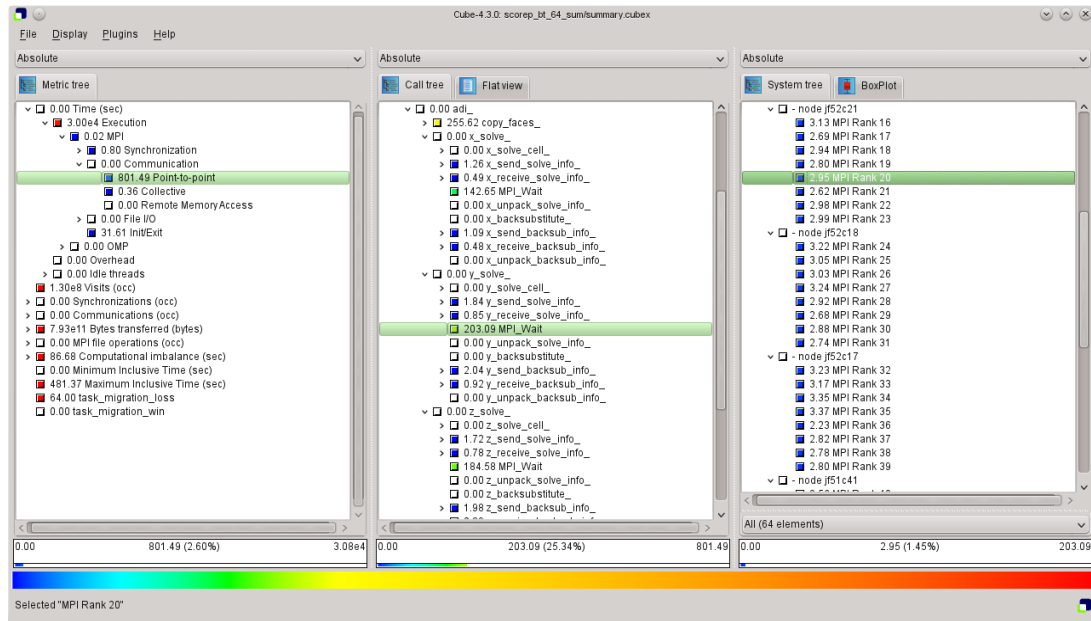


Figure 2.1: Screenshot of a summary experiment result in the Cube report browser.

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, e.g., machines, compute nodes, processes, and threads, a topological view of the application's processes and threads (if available), or a box 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 "Online Description" 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.)

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 55 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. (Alternatively, the filtering file could be extended to also exclude the `binvrhs` routine from measurement.) Note that renaming or removing the summary experiment directory is not necessary, as trace experiments are created with suffix "trace".

```
% export SCOREP_TOTAL_MEMORY=55M
% scalasca -analyze -q -t -f npb-bt.filt mpiexec -n 64 ./bt.D.64
S=C=A=N: Scalasca 2.2 trace collection and analysis
S=C=A=N: ./scorep_bt_64_trace experiment archive
S=C=A=N: Thu Jan 29 22:29:02 2015: Collect start
mpiexec -n 64 [...] ./bt.D.64
```

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: 64
```

```
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 = 0.10000000000000E-07
Comparison of RMS-norms of residual
  1 0.2533188551738E+05 0.2533188551738E+05 0.1479210131727E-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 0.8597827149538E-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 0.1582835864248E-12
  3 0.7782212022645E+02 0.7782212022645E+02 0.4053872777553E-13
  4 0.6835623860116E+02 0.6835623860116E+02 0.3762882153975E-13
  5 0.6065737200368E+03 0.6065737200368E+03 0.2474004739002E-13
Verification Successful
```

BT Benchmark Completed.

Class	=	D
Size	=	408x 408x 408
Iterations	=	250
Time in seconds	=	477.71
Total processes	=	64
Compiled procs	=	64
Mop/s total	=	122114.71
Mop/s/process	=	1908.04
Operation type	=	floating point
Verification	=	SUCCESSFUL
Version	=	3.3
Compile date	=	29 Jan 2015

Compile options:

MPIF77	=	scorep mpif77
FLINK	=	\$(MPIF77)
FMPI_LIB	=	(none)
FMPI_INC	=	(none)
FFLAGS	=	-O2
FLINKFLAGS	=	-O2
RAND	=	(none)

Please send the results of this run to:

NPB Development Team

Internet: npb@nas.nasa.gov

If email is not available, send this to:

MS T27A-1
NASA Ames Research Center
Moffett Field, CA 94035-1000

Fax: 650-604-3957

```
S=C=A=N: Thu Jan 29 22:37:06 2015: Collect done (status=0) 484s
S=C=A=N: Thu Jan 29 22:37:06 2015: Analyze start
mpiexec -n 64 scout.mpi ./scorep_bt_64_trace/traces.otf2
SCOUT Copyright (c) 1998-2015 Forschungszentrum Juelich GmbH
        Copyright (c) 2009-2014 German Research School for Simulation
        Sciences GmbH
```

Analyzing experiment archive ./scorep_bt_64_trace/traces.otf2

```
Opening experiment archive ... done (0.017s).
Reading definition data ... done (0.018s).
Reading event trace data ... done (1.189s).
Preprocessing ... done (2.067s).
Analyzing trace data ...
  Wait-state detection (fwd) (1/5) ... done (1.137s).
  Wait-state detection (bwd) (2/5) ... done (0.538s).
  Synchpoint exchange (3/5) ... done (1.016s).
  Critical-path & delay analysis (4/5) ... done (0.888s).
  Propagating wait-state exchange (5/5) ... done (0.475s).
done (4.065s).
Writing analysis report ... done (0.137s).
```

Max. memory usage : 175.965MB

*** WARNING ***

9982 clock condition violations detected:

Point-to-point: 9982

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 : 7.628s

S=C=A=N: Thu Jan 29 22:37:15 2015: Analyze done (status=0) 9s

Warning: 2.968GB of analyzed trace data retained in ./scorep_bt_64_trace/traces!

S=C=A=N: ./scorep_bt_64_trace complete.

% ls scorep_bt_64_trace

```
scorep.cfg  scorep.log  scout.log  traces.def  trace.stat
scorep.filt scout.cubex traces      traces.otf2
```

After successful trace collection and analysis, the generated experiment directory `scorep_bt_64_trace` contains the measurement configuration `scorep.cfg`, the measurement log file `scorep.log` and a copy of the filtering file `scorep.filt`. In addition, an OTF2 trace archive is produced, consisting of the anchor file `traces.otf2`, the global definitions file `traces.def` and the per-process data in the `traces/` directory. Finally, the experiment also includes the trace analysis reports `scout.cubex` and `trace.stat`, as well as a log file storing the output of the trace analyzer (`scout.log`).

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, 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. An existing trace measurement can be re-analyzed using the `-a` option of the `scalasca -analyze` command, avoiding the need to collect a new experiment:

```
% export SCAN_ANALYZE_OPTS="--time-correct"
% scalasca -analyze -a mpiexec -n 64 ./bt.D.64
S=C=A=N: Scalasca 2.2 trace analysis
S=C=A=N: ./scorep_bt_64_trace experiment archive
S=C=A=N: Thu Jan 29 22:37:16 2015: Analyze start
mpiexec -n 64 scout.mpi --time-correct ./scorep_bt_64_trace/traces.otf2
SCOUT Copyright (c) 1998-2015 Forschungszentrum Juelich GmbH
        Copyright (c) 2009-2014 German Research School for Simulation
        Sciences GmbH
```

```
Analyzing experiment archive ./scorep_bt_64_trace/traces.otf2
```

```
Opening experiment archive ... done (0.005s).
Reading definition data    ... done (0.009s).
Reading event trace data  ... done (1.349s).
Preprocessing              ... done (2.063s).
Timestamp correction      ... done (0.683s).
Analyzing trace data      ...
  Wait-state detection (fwd) (1/5) ... done (1.170s).
  Wait-state detection (bwd) (2/5) ... done (0.570s).
  Synchpoint exchange      (3/5) ... done (1.009s).
  Critical-path & delay analysis (4/5) ... done (0.893s).
  Propagating wait-state exchange (5/5) ... done (0.502s).
done (4.150s).
Writing analysis report    ... done (0.185s).

Max. memory usage        : 207.742MB
```

```

# passes      : 1
# violated    : 2679
# corrected   : 18791313
# reversed-p2p : 2679
# reversed-coll : 0
# reversed-omp : 0
# events      : 263738186
max. error    : 0.000072 [s]
error at final. : 0.000000 [%]
Max slope     : 0.010000000

Total processing time      : 8.580s
S=C=A=N: Thu Jan 29 22:37:25 2015: Analyze done (status=0) 9s
Warning: 2.968GB of analyzed trace data retained in ./scorep_bt_64_trace/traces!
S=C=A=N: ./scorep_bt_64_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 postprocessed and interactively explored using the Cube report browser, again using the `scalasca -examine` convenience command:

```

% scalasca -examine scorep_bt_64_trace
INFO: Post-processing trace analysis report...
INFO: Displaying ./scorep_bt_64_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 roughly half of the time spent in MPI point-to-point communication is waiting time, equally split into *Late Sender* and *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.

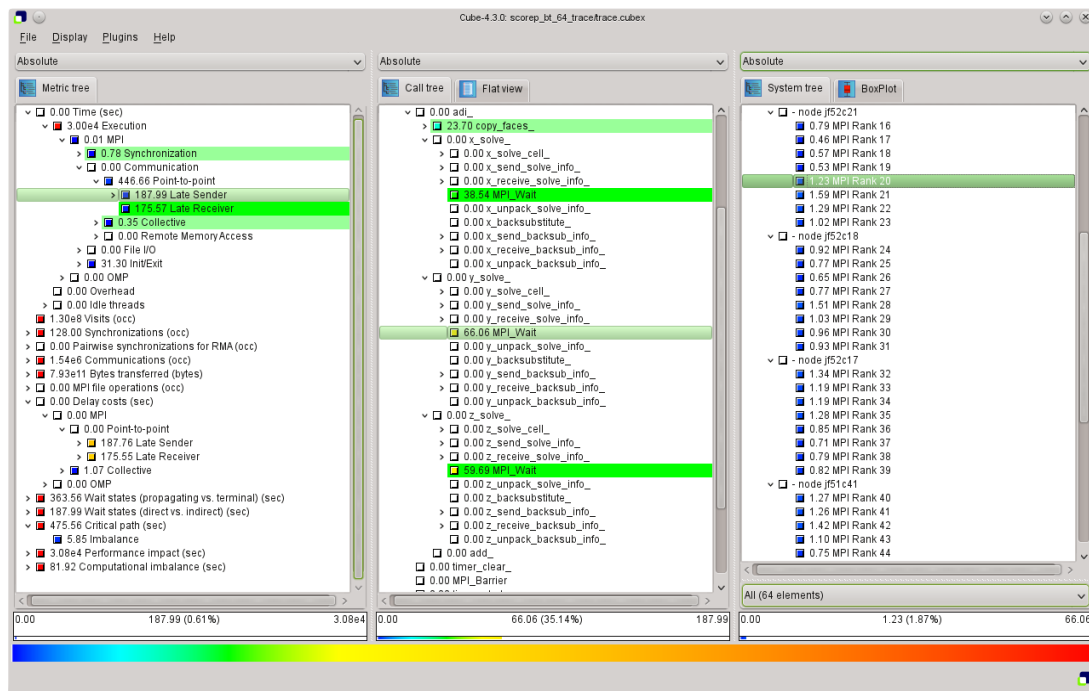


Figure 2.2: Screenshot of a trace analysis result in the Cube report browser.

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] Cube User Guide. Available as part of the Cube installation or online at <http://www.scalasca.org/software/cube-4.x/documentation.html>. [19](#)
- [4] 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](#)
- [5] 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](#)
- [6] 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](#)
- [7] 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](#)
- [8] Message Passing Interface Forum. MPI: A message-passing interface standard, Version 3.0. <http://www.mpi-forum.org>, Sept. 2012. [1](#)
- [9] NASA Advanced Supercomputing Division. NAS Parallel Benchmarks website. <https://www.nas.nasa.gov/publications/npb.html>. [7](#)
- [10] OpenMP Architecture Review Board. OpenMP application program interface, Version 4.0. <http://www.openmp.org>, July 2013. [1](#)
- [11] Scalasca 2.x series documentation web page. <http://www.scalasca.org/software/scalasca-2.x/documentation.html>. [7](#)

- [12] Score-P User Manual. Available as part of the Score-P installation or online at <http://www.score-p.org>. 3, 5, 6, 14, 16
- [13] 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

