# SCORE-P
## USER MANUAL
3.0 (revision 11303)


Thu Sep 15 2016 17:48:53

# SCORE-P LICENSE AGREEMENT

# Contents

# Chapter 1

# Introduction

This document provides an introduction to Score-P: the *Scalable Performance Measurement Infrastructure for Parallel Codes*. It is a software system that provides a measurement infrastructure for profiling, event trace recording, and online analysis of High Performance Computing (HPC) applications. It has been developed within the framework of the Scalable Infrastructure for the Automated Performance Analysis of Parallel Codes (SILC) project funded by the German Federal Ministry of Education and Research (BMBF) under its HPC programme and the Performance Refactoring of Instrumentation, Measurement, and Analysis Technologies for Petascale Computing (PRIMA) project,funded by the United States Department of Energy (DOE) with the goals of being highly scalable and easy to use.

The partners involved in the development of this system within the SILC and PRIMA projects were:

- Forschungszentrum Jülich,
- German Research School for Simulation Sciences,
- Gesellschaft für numerische Simulation mbH,
- Gesellschaft für Wissens- und Technologietransfer der TU Dresden (GWT-TUD GmbH),
- Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen,
- Technische Universität Dresden,
- Technische Universität München,
- and University of Oregon

The goal of Score-P is to simplify the analysis of the behavior of high performance computing software and to allow the developers of such software to find out where and why performance problems arise, where bottlenecks may be expected and where their codes offer room for further improvements with respect to the run time. A number of tools have been around to help in this respect, but typically each of these tools has only handled a certain subset of the questions of interest. A software developer who wanted to have a complete picture of his code therefore was required to use a multitude of programs to obtain the desired information. Most of these utilities work along similar principles. The first step is usually an instrumentation of the code to be investigated. Next, the instrumented programs are executed and write out (often very large amounts of) performance data. These data are then finally graphically displayed and analyzed after the end of the program run. In certain special cases, the visualization and analysis of the program behavior is also done while the program is running.

A crucial problem in the traditional approach used to be the fact that each analysis tool had its own instrumentation system, so the user was commonly forced to repeat the instrumentation procedure if more than one tool was to be employed. In this context, Score-P offers the user a maximum of convenience by providing the Opari2 instrumenter as a common infrastructure for a number of analysis tools like Periscope, Scalasca, Vampir, and Tau that obviates the need for multiple repetitions of the instrumentation and thus substantially reduces the amount of work required. It is open for other tools as well. Moreover, Score-P provides the new Open Trace Format Version 2

(OTF2) for the tracing data and the new CUBE4 profiling data format which allow a better scaling of the tools with respect to both the run time of the process to be analyzed and the number of cores to be used.

Score-P supports the following programming paradigms:

**Multi-process paradigms:**
- *MPI*
- *SHMEM*

**Thread-parallel paradigms:**
- *OpenMP*
- *Pthreads*

**Accelerator-based paradigms:**
- *CUDA*
- *OpenCL*
- *OpenACC*

And possible combinations from these including simple *serial* programs.

## 1.1 About this Document

This document consists of three parts. This chapter is devoted to a basic introduction to performance analysis in general and the components of the Score-P system in particular. Chapter 2 is a beginner's guide to using the Score-P tool suite. It demonstrates the basic steps and commands required to initiate a performance analysis of a parallel application. In the Chapters 3, 4, and 5, the reader can find more detailed information about the components of Score-P. Chapter 7 provides a typical workflow of performance analysis with Score-P and detailed instructions.

## 1.2 Getting Help and Support

The Score-P project uses various mailing lists to coordinate the development and to provide support to the user community. An overview of the available mailing lists can be found in 1.1.

You can subscribe to the news@score-p.org and support@score-p.org by ...

Table 1.1: Score-P mailing lists

| List Address | Subscription | Posting | Usage |
|---|---|---|---|
| news@score-p.org | open | core team | Important news regarding the Score-P software, e.g. announcements of new releases. |
| support@score-p.org | closed | anyone | Bug reports and general user support for the Score-P software. |

## 1.3 Basics of Performance Optimization

Performance optimization is a process that is usually executed in a work cycle consisting of a number of individual steps as indicated in Figure 1.1.

Thus, the process always begins with the original application in its unoptimized state. This application needs to be instrumented, i. e. it must be prepared in order to enable the measurement of the performance properties to take place. There are different ways to do this, including manual instrumentation of the source code by the user, automatic instrumentation by the compiler, or linking against pre-instrumented libraries. All these options are available in Score-P.

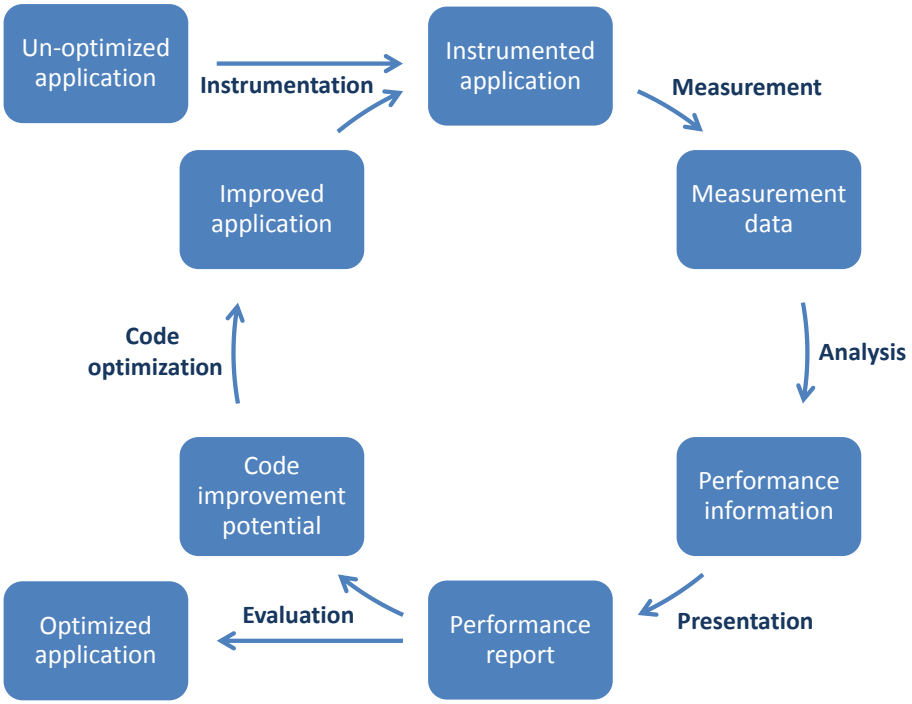


Figure 1.1: The performance optimization cycle

When the instrumented application obtained in this way is executed, the additional commands introduced during the instrumentation phase collect the data required to evaluate the performance properties of the code. Depending on the user's requirements, Score-P allows to store these data either as a profile or as event traces. The user must keep in mind here that the execution of the additional instructions of course requires some run time and storage space. Thus the measurement itself has a certain influence of the performance of the instrumented code. Whether the perturbations introduced in this way have a significant effect on the behavior depends on the specific structure of the code to be investigated. In many cases the perturbations will be rather small so that the overall results can be considered to be a realistic approximation of the corresponding properties of the uninstrumented code. However, certain constructions like regions with very small temporal extent that are executed frequently are likely to suffer from significant perturbations. It is therefore advisable not to measure such regions.

The next step is the analysis of the data obtained in the measurement phase. Traditionally this has mainly been done post mortem, i. e. after the execution of the instrumented application has ended. This is of course possible in Score-P too, but Score-P offers the additional option to go into the analysis in the so-called on-line mode, i. e. to investigate the performance data while the application is still running. If the collected data are event traces then a more detailed investigation is possible than in the case of profiles. In particular, one can then also look at more sophisticated dependencies between events happening on different processes.

The optimization cycle then continues with the presentation of the analysis results in a report. Here it is important to eliminate the part of the information that is irrelevant for the code optimization from the measured data. The reduction of the complexity achieved in this way will simplify the evaluation of the data for the user. However, care must be taken in order not to present the results in a too abstract fashion which would hide important facts from the user.

The performance report then allows the user to evaluate the performance of the code. One can then either conclude that the application behaves sufficiently well and exit the optimization cycle with the optimized version of the software being chosen as the final state, or one can proceed to identify weaknesses that need to be addressed and the potential for improvements of the code.

In the latter case, one then continues by changing the source code according to the outcome of the previous step and thus obtains an improved application that then can again be instrumented to become ready for a re-entry into the optimization cycle.

## 1.4 Score-P Software Architecture Overview

In order to allow the user to perform such an optimization of his code (typically written in Fortran, C, or C++ and implemented in a serial way or using a parallelization via an multi-process, thread-parallel, accelerator-based paradigm, or a combination thereof), the Score-P system provides a number of components that interact with each other and with external tools. A graphical overview of this structure is given in Fig. 1.2. We shall now briefly introduce the elements of this structure; more details will be given in the later chapters of this document.

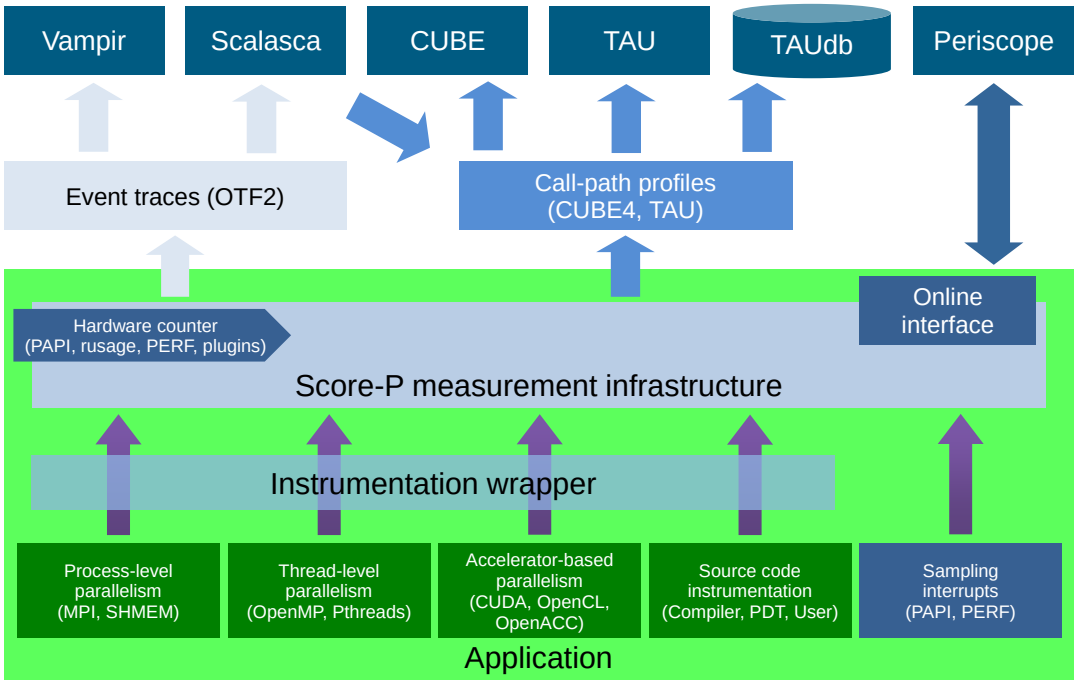


Figure 1.2: Overview of the Score-P measurement system architecture and the tools interface.

In order to instrument an application, the user needs to recompile the application using the Score-P instrumentation command, which is added as a prefix to the original compile and link command lines. It automatically detects the programming paradigm by parsing the original build instructions and utilizes appropriate and configurable methods of instrumentation. These are currently:

- compiler instrumentation,
- MPI and SHMEM library interposition,
- source code instrumentation via the TAU instrumenter,
- OpenMP source code instrumentation using Opari2.
- Pthread and OpenCL instrumentation via GNU ld library wrapping.
- CUDA instrumentation via the NVIDIA CUDA Profiling Tools Interface (CUPTI)
- OpenACC instrumentation using the OpenACC Profiling Interface

While the first three of these methods are based on using tools provided externally, the Opari2 instrumenter for OpenMP programs is a part of the Score-P infrastructure itself. It is an extension of the well known and frequently used `OpenMP Pragma And Region Instrumenter` system (Opari) that has been successfully used in the past in combination with tools like Scalasca, VampirTrace and ompP. The fundamental concept of such a system is a

source-to-source translation that automatically adds all necessary calls to a runtime measurement library allowing to collect runtime performance data of Fortran, C, or C++ OpenMP applications. This translation is based on the idea of OpenMP pragma/directive rewriting. The key innovation in Opari2, as compared to its predecessor, is the capability to support features introduced in version 3.0 of the OpenMP standard, in particular its new tasking functionality and OpenMP nesting. Opari used to work by automatically wrapping OpenMP constructs like parallel regions with calls to the portable OpenMP monitoring interface POMP. In order to reflect the above-mentioned extensions, this interface also had to be replaced by an enhanced version, POMP2.

Additionally, the user may instrument the code manually with convenient macros provided by Score-P. Score-P also supports sampling functionality that provides an alternative to direct instrumentation.

During measurement, the system records several performance metrics including execution time, communication metrics, and optionally hardware counters. Performance data is stored in appropriately sized chunks of a preallocated memory buffer that are assigned to threads on demand, efficiently utilizing the available memory and avoiding measurement perturbation by flushing the data to disk prematurely.

Without recompilation, measurement runs can switch between *tracing* and *profiling* mode. In tracing mode, the performance events are passed to the tracing back-end of Score-P and are written to files for subsequent post mortem analysis using Scalasca or Vampir. This backend uses the newly developed Open Trace Format 2 (OTF2), the joint successor of the Open Trace Format used by Vampir and the Epilog format used by Scalasca. The Score-P system contains a new library with reading and writing routines for OTF2. Basically, OTF2 is a full merge of its two predecessors that retains all their features, and it is planned to become the default data source for future versions of both Vampir and Scalasca. In this way, the user is free to choose between these two complementary tools to investigate the trace files and may select the one that is more appropriate for the specific question at hand. As an alternative to writing the trace data to disk and evaluating them post mortem, it is also possible to directly hand over the data to on-line analysis tools like Periscope. The corresponding interface that allows this on-line access is also an integral part of Score-P.

In profiling mode, the performance events are summarized at runtime separately for each call-path like in Scalasca. Additionally, support for phases, dynamic regions and parameter-based profiling has been integrated. The collected data is passed to the Score-P's profiling back-end CUBE4 for post mortem analysis using Scalasca or TAU or is used directly through the on-line access interface by Periscope. Also in profiling mode, Score-P supports the automatic detection of MPI wait states. Usually such inefficiencies are important bottlenecks and are thoroughly investigated by means of automatic trace analysis and subsequent visual analysis using a time-line representation. In the case of Score-P wait time profiling, inefficiencies are detected immediately when the respective MPI call is completed and stored as an additional metric in the call-path profile. In comparison to earlier versions of CUBE, this new one features a more powerful data model, more flexibility in the specification of system resource hierarchies and display parameters, and various techniques to enhance the efficiency that result in a much better scaling behavior of the analysis tool even in a range of tens of thousands of processes.

As a rough guideline for users who are uncertain which of these two modes to employ, we provide a brief comparison of their main advantages and disadvantages. Specifically, tracing mode allows to retain temporal and spatial connections, and it can reflect the dynamical behavior to an arbitrary precision. Moreover, statistical information and profiles may be derived from the program traces. On the other hand, the amount of data that is produced in the tracing mode can become prohibitively large; profiles tend to require much less storage space. In addition, the additional load that is imposed on the process, and hence the perturbations of the behavior of the code to be analyzed, are much smaller in profiling mode than in tracing mode. And finally we mention that the accurate synchronization of the clocks is an important aspect in tracing mode that may cause difficulties.

## 1.5 Acknowledgment

The development of Score-P was sponsored by a grant from the German Federal Ministry of Education and Research (Grant No. 01IH08006) within the framework of its High Performance Computing programme and with a grant from the US Department of Energy (Award No. DE-SC0001621). This support is gratefully acknowledged.

# Chapter 2

# Getting Started

In order to quickly introduce the user to the Score-P system, we explain how to build and install the tool and look at a simple example. We go through the example in full detail.

As mentioned above, the three core steps of a typical work cycle in the investigation of the behavior of a software package can be described as follows:

- *Instrumentation of user code*: Calls to the measurement system are inserted into the application. This can be done either fully automatically or with a certain amount of control handed to the software developer.
- *Measurement and analysis*: The instrumented application is executed under the control of the measurement system and the information gathered during the run time of this process is stored and analyzed.
- *Examination of results*: The information about the behavior of the code at run time is visualized and the user gets the opportunity to examine the reported results.

After building and installing the tool, we shall go through these three steps one after the other in the next sections. This will be followed by a full workflow example. For getting detailed presentations of available features, see Section 3 for the instrumentation step and Section 5 for the measurement.

## 2.1 Score-P Quick Installation

The Score-P performance analysis tool uses the GNU Autotools (Autoconf, Automake, Libtool and M4) build system. The use of Autotools allows Score-P to be build in many different systems with varying combinations of compilers, libraries and MPI implementations.

Autotools based projects are build as follows:

1. The available compilers and tools available are detected from the environment by the configure script.
2. Makefiles are generated based on the detected compilers and tools.
3. The generated Makefile project is then built and installed.

Score-P will have features enabled or disabled, based on the detection made by the Autotools generated configure script. The following 2 sub-sections cover mandatory prerequisites as well as optional features that are enabled based on what is available in the configured platform.

### 2.1.1 Prerequisites

To build Score-P, C, C++ and Fortran compilers and related tools are required. These can be available as modules (typically on super-computer environments) or as packages (on most Linux or BSD distributions).

For Debian based Linux systems using the APT package manager, the following command (as root) is sufficient to build Score-P with minimal features enabled:

```
apt-get install gcc g++ gfortran mpich2
```

On Red-Hat and derivative Linux systems running the YUM package manager, in a similar way:

```
yum install gcc g++ gfortran mpich2
```

For users of the SuperMUC, it is recommended to load the following modules:

```
module load ccomp/intel/12.1 fortran/intel/12.1 \
            mpi.ibm/5.2_PMR-fixes papi/4.9
```

### 2.1.2 General Autotools Build Options

System administrators can build Score-P with the familiar:

```
mkdir _build
cd _build
../configure && make && make install
```

The previous sequence of commands will detect compilers, libraries and headers, and then build and install Score-P in the following system directories:

```
/opt/scorep/bin
/opt/scorep/lib
/opt/scorep/include
/opt/scorep/share
```

Users that are not administrators on the target machine may need to install the tool in an different location (due to permissions). The prefix flag should be specified with the target directory:

```
../configure --prefix=<installation directory>
```

For example, in the `install/scorep` directory on his/her home folder:

```
../configure --prefix=$HOME/install/scorep
```

In this case, the user's `PATH` variable needs to be updated to include the `bin` directory of Score-P, and the appropriate library and include folders specified (with `-L` and `-I`) when instrumenting and building applications.

Users of the SuperMUC (after loading the required modules mentioned previously), can issue the following command to configure Score-P:

```
../configure --prefix=$HOME/install/scorep --enable-static    \
     --disable-shared --with-nocross-compiler-suite=intel     \
     --with-mpi=openmpi --with-papi-header=$PAPI_BASE/include \
     --with-papi-lib=$PAPI_BASE/lib
```

### 2.1.3 Score-P Specific Build Options

In addition to general options available in all Autotools based build systems, there are Score-P configuration flags. These can be printed out by passing the –help flag to the configure script.

They are usually self explanatory. Here is a list of them with a short explanation:

- `--with-nocross-compiler-suite= (gcc|ibm|intel|`
  `pathscale|pgi|studio)`

  Specifies the compiler suite to use when not cross-compiling. Selecting one of the options sets all relevant variables to their expected names. These are CC, FC, F77, as well as the linker, preprocessor, etc.

- `--with-frontend-compiler-suite= (gcc|ibm|intel| pathscale|pgi|studio)`

  Similar to the previous configuration flag, but for cross-compiling environments.

- `-with-mpi=(mpich2|impi|openmpi)` The MPI compiler and runtime suite to use. Currently there are entries for MPICH2, Intel MPI and Open MPI.

- `-with-shmem=(openshmem|openmpi|sgimpt)` The SHMEM compiler suite to build this package in non cross-compiling mode. Usually autodetected. Needs to be in $PATH.

- `-with-otf2=(yes|<otf2-bindir>)` An already install OTF2 can be specified with this flag. This is usually not necessary since OTF2 is built together with Score-P. Specify yes if the tool is in your $PATH, otherwise specify the full path.

- `-with-opari2=(yes|<opari2-bindir>)` Similar to the previous configuration flag, but for OPARI2.

- `-with-cube=(yes|<cube-bindir>)` Similar to the previous two configuration flags, but for CUBE.

## 2.2 Instrumentation

Various analysis tools are supported by the Score-P infrastructure. Most of these tools are focused on certain special aspects that are significant in the code optimization process, but none of them provides the full picture. In the traditional workflow, each tool used to have its own measurement system, and hence its own instrumenter, so the user was forced to instrument his code more than once if more than one class of features of the application was to be investigated. One of the key advantages of Score-P is that it provides an instrumentation system that can be used for all the performance measurement and analysis tools, so that the instrumentation work only needs to be done once.

Internally, the instrumentation itself will insert special measurement calls into the application code at specific important points (events). This can be done in an almost automatic way using corresponding features of typical compilers, but also semi-automatically or in a fully manual way, thus giving the user complete control of the process. In general, an automatic instrumentation is most convenient for the user. However, this approach may lead to too many and/or too disruptive measurements, and for such cases it is then advisable to use selective manual instrumentation and measurement instead. For the moment, we shall however start the procedure in an automatic way to keep things simple for novice users.

To this end, we need to ask the Score-P instrumenter to take care of all the necessary instrumentation of user and MPI functions. This is done by using the `scorep` command that needs to be prefixed to all the compile and link commands usually employed to build the application. Thus, an application executable `app` that is normally generated from the two source files `app1.f90` and `app2.f90` via the command:

```
mpif90 app1.f90 app2.f90 -o app
```

will now be built by:

```
scorep  mpif90 app1.f90 app2.f90 -o app
```

using the Score-P instrumenter.

In practice one will usually perform compilation and linking in separate steps, and it is not necessary to compile all source files at the same time (e.g., if makefiles are used). It is possible to use the Score-P instrumenter in such a case too, and this actually gives more flexibility to the user. Specifically, it is often sufficient to use the instrumenter not in all compilations but only in those that deal with source files containing MPI code. However, when invoking the linker, the instrumenter must *always* be used.

When makefiles are employed to build the application, it is convenient to define a placeholder variable to indicate whether a “preparation" step like an instrumentation is desired or only the pure compilation and linking. For example, if this variable is called `PREP` then the lines defining the C compiler in the makefile can be changed from:

```
MPICC = mpicc
```

to

```
MPICC = $(PREP) mpicc
```

(and analogously for linkers and other compilers). One can then use the same makefile to either build an instrumented version with the

```
make PREP="scorep"
```

command or a fully optimized and not instrumented default build by simply using:

```
make
```

in the standard way, i.e. without specifying `PREP` on the command line. Of course it is also possible to define the same compiler twice in the makefile, once with and once without the `PREP` variable, as in:

```
MPICC          = $(PREP) mpicc
MPICC_NO_INSTR =         mpicc
```

and to assign the former to those source files that must be instrumented and the latter to those files that do not need this.

## 2.3 Measurement and Analysis

Once the code has been instrumented, the user can initiate a measurement run using this executable. To this end, it is sufficient to simply execute the target application in the usual way, i.e.:

```
mpiexec $MPIFLAGS app [app_args]
```

in the case of an MPI or hybrid code, or simply:

```
app [app_args]
```

for a serial or pure OpenMP program. Depending on the details of the local MPI installation, in the former case the `mpiexec` command may have to be substituted by an appropriate replacement.

When running the instrumented executable, the measurement system will create a directory called `scorep-YYYYMMDD_HHMM_XXXXXXXX` where its measurement data will be stored. Here `YYYYMMDD` and `HHMM` are the date (in year-month-day format) and time, respectively, when the measurement run was started, whereas `XXXXXX` is an additional identification number. Thus, repeated measurements, as required by the optimization work cycle, can easily be performed without the danger of accidentally overwriting results of earlier measurements. The environment variables `SCOREP_ENABLE_TRACING` and `SCOREP_ENABLE_PROFILING` control whether event trace data or profiles are stored in this directory. By setting either variable to `true`, the corresponding data will be written to the directory. The default values are `true` for `SCOREP_ENABLE_PROFILING` and `false` for `SCOREP_ENABLE_TRACING`.

## 2.4 Report Examination

After the completion of the execution of the instrumented code, the requested data (traces or profiles) is available in the indicated locations. Appropriate tools can then be used to visualize this information and to generate reports, and thus to identify weaknesses of the code that need to be modified in order to obtain programs with a better performance. A number of tools are already available for this purpose. This includes, in particular, the `CUBE4 performance report explorer` for viewing and analyzing profile data, `Vampir` for the investigation of trace information, and the corresponding components of the `TAU` toolsuite.

Alternatively, the `Periscope` system may be used to analyze the behaviour of the code on-line during its run time, i.e. (in contrast to the approaches mentioned above) *before* the end of the program run.

## 2.5 Simple Example

As a specific example, we look at a short C code for the solution of a Poisson equation in a hybrid (MPI and OpenMP) environment. The corresponding source code comes as part of the Score-P distribution under the *scorep/test/jacobi/* folder. Various other versions are also available - not only hybrid but also for a pure MPI parallelization, a pure OpenMP approach, and in a non-parallel way; and, in each case, not only in C but also in C++ and Fortran.

As indicated above, the standard call sequence:

```
mpicc -std=c99 -g -O2 -fopenmp -c jacobi.c
mpicc -std=c99 -g -O2 -fopenmp -c main.c
mpicc -std=c99 -g -O2 -fopenmp -o jacobi jacobi.o main.o -lm
```

that would first compile the two C source files and then link everything to form the final executable needs to be modified by prepending `scorep` to each of the three commands, i.e. we now have to write:

```
scorep mpicc -std=c99 -g -O2 -fopenmp -c jacobi.c
scorep mpicc -std=c99 -g -O2 -fopenmp -c main.c
scorep mpicc -std=c99 -g -O2 -fopenmp -o jacobi jacobi.o \
             main.o -lm
```

This call sequence will create a number of auxiliary C source files containing the original source code and a number of commands introduced by the measurement system in order to enable the latter to create the required measurements when the code is actually run. These modified source files are then compiled and linked, thus producing the desired executable named `jacobi`.

The actual measurement process is then initiated, e.g., by the call:

```
mpiexec -n 2 ./jacobi
```

The output data of this process will be stored in a newly created experiment directory `scorep-YYYYMMDD_↩ HHMM_XXXXXXXX` whose name is built up from the date and time when the measurement was started and an identification number.

As we had not explicitly set any Score-P related environment variables, the profiling mode was active by default. We obtain a file called `profile.cubex` containing profiling data in the experiment directory as the result of the measurement run. This file can be visually analyzed with the help of CUBE.

If we had set the variable `SCOREP_ENABLE_TRACING` to `true`, we would additionally have obtained trace data, namely the so called anchor file `traces.otf2` and the global definitions `traces.def` as well as a subdirectory `traces` that contains the actual trace data. This trace data is written in Open Trace Format 2 (OTF2) format. OTF2 is the joint successor of the classical formats OTF (used, e. g., by Vampir) and Epilog (used by Scalasca). A tool like Vampir can then be used to give a visual representation of the information contained in these files.

# Chapter 3

# Application Instrumentation

Score-P provides several possibilities to instrument user application code. Besides the automatic compiler-based instrumentation (Section 3.1), it provides manual instrumentation using the Score-P User API (Section 3.2), semi-automatic instrumentation using POMP2 directives (Section 3.6) and, if configured, automatic source-code instrumentation using the PDToolkit-based instrumenter (Section 3.8).

As well as user routines and specified source regions, Score-P currently supports the following kinds of events:

- **MPI library calls**:
  Instrumentation is accomplished using the standard MPI profiling interface PMPI. To enable it, the application program has to be linked against the Score-P MPI (or hybrid) measurement library plus MPI-specific libraries. Note that the Score-P libraries must be linked *before* the MPI library to ensure interposition will be effective.
- **SHMEM library calls**:
  Instrumentation is accomplished using the SHMEM profiling interface or the GNU linker for library wrapping. To enable it, the application program has to be linked against the Score-P SHMEM (or hybrid) measurement library plus SHMEM-specific libraries. Note that the Score-P libraries must be linked *before* the SHMEM library to ensure interposition will be effective.
- **OpenMP directives & API calls**:
  The Score-P measurement system uses the OPARI2 tool for instrumentation of OpenMP constructs. See the OPARI2 documentation on how to instrument OpenMP source code. In addition, the application must be linked with the Score-P OpenMP (or hybrid) measurement library.
- **Pthread library calls**:
  The Score-P measurement system uses GNU linker for instrumentation of Pthreads library calls. At the moment only a few library calls are supported.

The Score-P instrumenter command `scorep` automatically takes care of compilation and linking to produce an instrumented executable, and should be prefixed to compile and link commands. Often this only requires prefixing definitions for `CC` or `MPICC` (and equivalents) in Makefiles.

Usually the Score-P instrumenter `scorep` is able to automatically detect the programming paradigm from the set of compile and link options given to the compiler. In some cases however, when the compiler or compiler wrapper enables specific programming paradigm by default (e.g., Pthreads on Cray and Blue Gene/Q systems), `scorep` needs to be made aware of the programming paradigm in order to do the correct instrumentation. Please see `scorep -help` for the available options.

When using Makefiles, it is often convenient to define a "preparation preposition" placeholder (e.g., `PREP`) which can be prefixed to (selected) compile and link commands:

```
MPICC  = $(PREP) mpicc
MPICXX = $(PREP) mpicxx
MPIF90 = $(PREP) mpif90
```

These can make it easier to prepare an instrumented version of the program with

```
make PREP="scorep"
```

while default builds (without specifying `PREP` on the command line) remain fully optimized and without instrumentation.

In order to instrument applications which employ GNU Autotools for building, following instrumentation procedure has to be used:

1. Configure application as usual, but provide additional argument:
   `-disable-dependency-tracking`
2. Build application using `make` command with compiler specification variables set as follows:
   ```
   make CC="scorep <your-cc-compiler>" \
        CXX="scorep <your-cxx-compiler>" \\
        FC="scorep <your-fc-compiler>" ...
   ```

When compiling without the Score-P instrumenter, the `scorep-config` command can be used to simplify determining the appropriate linker flags and libraries, or include paths:

```
scorep-config [--mpp=none|--mpp=mpi|--mpp=shmem] \
[--thread=none|--thread=omp|--thread=pthread] --libs
```

The –mpp=<paradigm> switch selects which message passing paradigm is used. Currently, Score-P supports applications using MPI ( –mpp=mpi ) or SHMEM ( –mpp=shmem ) and applications without any message passing paradigm. It is not possible to specify two message passing systems for the same application. The –thread=<paradigm> switch selects which threading system is used in Score-P. You may use OpenMP ( –thread=omp ), no threading system ( –thread=none ) or POSIX threading system ( –thread=pthread ). It is not possible to specify two threading systems for the same application. However, you may combine a message passing system with a threading system.

**Note**

A particular installation of Score-P may not offer all measurement configurations!

The `scorep-config` command can also be used to determine the right compiler flags for specifying the include directory of the `scorep/SCOREP_User.h` or `scorep/SCOREP_User.inc` header files. When compiling without using the Score-P instrumenter, necessary defines and compiler instrumentation flags can be obtained by calling one of the following, depending on the language:

```
scorep-config --cflags [<options>]
scorep-config --cxxflags [<options>]
scorep-config --fflags [<options>]
```

If you compile a C file, you should use `-cflags`. If you use a C++ program, you should use `-cxxflags`. And if you compile a Fortran source file, you should use `-flags`.

With the additional options it is possible to select the used adapter, the threading system and the message passing system. For each adapter, we provides a pair of flags of the form `-adapter`, and `-noadapter` (please replace `adapter` by the name of the adapter). This allows to get options for non-default instrumentation possibilities. E.g., `-user` enables the manual instrumentation with the Score-P user API, the `-nocompiler` option disables compiler instrumentation.

**Note**

Disabling OpenMP measurements with the `-noopenmp` flag, disables all except parallel regions. Internally Score-P needs to track events on a per-thread basis and thus needs to be aware of the creation and destruction of OpenMP threads. Accordingly these regions will also show up in the measurements.

Score-P supports a variety of instrumentation types for user-level source routines and arbitrary regions, in addition to fully-automatic MPI and OpenMP instrumentation, as summarized in Table 3.1.

Table 3.1: Score-P instrumenter option overview

| Type of instrumentation | Instrumenter switch | Default value | Instrumented routines | Runtime measurement control |
|---|---|---|---|---|
| MPI | `--mpp=mpi/ --mpp=none` | (auto) | configured by install | see Sec. 5.7.1 |
| SHMEM | `--mpp=shmem/ --mpp=none` | (auto) | configured by install | – |
| OpenCL | `--opencl/ --noopencl` | enabled | configured by install | see Sec. 5.9 |
| OpenACC | `--openacc/ --noopenacc` | enabled | configured by install | see Sec. 5.10 |
| OpenMP | `--thread=omp/ --thread=none --openmp --noopenmp` | (auto) | all parallel constructs | – |
| Pthread | `--thread= pthread` | (auto) | Basic Pthread library calls | – |
| Compiler, Sec. 3.1 | `--compiler/ --nocompiler` | enabled | all | Filtering, Sec. 5.3 |
| PDT, Sec. 3.8 | `--pdt/ --nopdt` | disabled | all | Filtering, Sec. 5.3 |
| POMP2 user regions, Sec. 3.6 | `--pomp/ --nopomp` | disabled | manually annotated | Filtering, Sec. 5.3 |
| Manual, Sec. 3.2 | `--user/ --nouser` | disabled | manually annotated | Filtering, Sec. 5.3, and selective recording, Sec. 5.4 |

When the instrumenter determines that MPI or OpenMP are being used, it automatically enables MPI library instrumentation or OPARI2-based OpenMP instrumentation, respectively. The default set of instrumented MPI library functions is specified when Score-P is installed. All OpenMP parallel constructs and API calls are instrumented by default.

**Note**

To fine-tune instrumentation of OpenMP regions, use the `-opari=<parameter-list>` option. For available parameters please refer to the OPARI2 manual.

By default, automatic instrumentation of user-level source routines by the compiler is enabled (equivalent to specifying `-compiler`). The compiler instrumentation can be disabled with `-nocompiler` when desired, such as when using PDToolkit, or POMP2 or Score-P user API manual source annotations, are enabled with `-pdt`, `-pomp` and `-user`, respectively. Compiler, PDToolkit, POMP2 and Score-P user API instrumentation can all be used simultaneously, or in arbitrary combinations, however, it is generally desirable to avoid instrumentation duplication (which would result if all are used to instrument the same routines). Note that enabling PDToolkit instrumentation automatically enables Score-P user instrumentation, because it inserts Score-P user macros into the source code.

**Note**

There are two ways of internal data handling for measurements involving the OpenMP threading model. The possible options are:
`--thread=omp:pomp_tpd`
`--thread=omp:ancestry`
These options should be identical in behavior. If you specify `--thread=omp` or OpenMP is automatically detected, the default is `pomp_tpd`.

Sometimes it is desirable to explicitly direct the Score-P instrumenter to do nothing except execute the associated compile/link command. For such cases it is possible to disable default instrumentation with `-nocompiler`, `-thread=none`, and/or `-mpp=none`. Although no instrumentation is performed, this can help verify that the Score-P instrumenter correctly handles the compile/link commands.

**Note**

Disabling OpenMP in the instrumenter for OpenMP applications will cause errors during program execution if any event occurs inside of a parallel region.

Each thread model uses a default internal locking mechanism for the Score-P measurement system. For the standard use case there is no need to specify an explicit locking mode. However, on certain systems or for performance reasons it might be useful to change the locking mode. For these cases the instrumenter provides the option `--mutex=[omp|pthread|pthread:spinlock|pthread:wrap|none]`. Current possibilities are the OpenMP locking (omp), Pthread mutex (pthread), Pthread spinlock (pthread:spinlock), Pthread mutex, where original functions replaced with __real functions (pthread:wrap), and none at all (none). Which of these are available for a given installation will be determined at configure time.

**Note**

Not all combinations of thread model and explicit choice of locking are useful. Currently, only the combination of no locking with a real threading system is overwritten by the thread model default to ensure thread safety.

## 3.1 Automatic Compiler Instrumentation

Most current compilers support automatic insertion of instrumentation calls at routine entry and exit(s), and Score-P can use this capability to determine which routines are included in an instrumented measurement.

Compiler instrumentation of all routines in the specified source file(s) is enabled by default by Score-P, or can be explicitly requested with `--compiler`. Compiler instrumentation is disabled with `--nocompiler`.

**Note**

Depending on the compiler, and how it performs instrumentation, insertion of instrumentation may disable in-lining and other significant optimizations, or in-lined routines may not be instrumented at all (and therefore "invisible").

Automatic compiler-based instrumentation has been tested with a number of different compilers:

- GCC (UNIX-like operating systems, not tested with Windows)
- IBM xlc, xlC (version 7 or later, IBM Blue Gene)
- IBM xlf (version 9.1 or later, IBM Blue Gene)
- PGI (on Linux)
- Intel compilers (version 10 or later, Linux)
- SUN Studio compilers (Linux, Fortran only)

In all cases, Score-P supports automatic instrumentation of C, C++ and, Fortran codes, except for the SUN Studio compilers which only provide appropriate support in their Fortran compiler.

**Note**

The automatic compiler instrumentation might create a significant relative measurement overhead on short function calls. This can impact the overall application performance during measurement. C++ applications are especially prone to suffer from this, depending on application design and whether C++ STL functions are also instrumented by the compiler. Currently, it is not possible to prevent the instrumentation of specific functions on all platforms when using automatic compiler instrumentation.
As an exception, the GCC plug-in based function instrumentation supports all filtering features when using the `--instrument-filter` flag to the Score-P instrumenter (see Sec. 5.3).

Names provided for instrumented routines depend on the compiler, which may add underscores and other decorations to Fortran and C++ routine names, and whether name "demangling" has been enabled when Score-P was installed and could be applied successfully.

## 3.2 Manual Region Instrumentation

In addition to the automatic compiler-based instrumentation (see Section 3.1), instrumentation can be done manually. Manual instrumentation can also be used to augment automatic instrumentation with region or phase annotations, which can improve the structure of analysis reports. Furthermore, it offers the possibility to record additional, user defined metrics. Generally, the main program routine should be instrumented, so that the entire execution is measured and included in the analysis.

Instrumentation can be performed in the following ways, depending on the programming language used.

Fortran:

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

subroutine foo
  SCOREP_USER_REGION_DEFINE( my_region_handle )
  ! more declarations

  SCOREP_USER_REGION_BEGIN( my_region_handle, "foo",
      SCOREP_USER_REGION_TYPE_COMMON )
  ! do something
  SCOREP_USER_REGION_END( my_region_handle )

end subroutine foo
```

C/C++:

```
#include <scorep/SCOREP_User.h>

void foo()
{
  SCOREP_USER_REGION_DEFINE( my_region_handle )

  // more declarations

  SCOREP_USER_REGION_BEGIN( my_region_handle, "foo",
      SCOREP_USER_REGION_TYPE_COMMON )

  // do something

  SCOREP_USER_REGION_END( my_region_handle )
}
```

C++ only:

```
#include <scorep/SCOREP_User.h>

void foo()
{
  SCOREP_USER_REGION( "foo", SCOREP_USER_REGION_TYPE_FUNCTION
      )

  // do something
}
```

**Note**

When using Fortran, make sure the C preprocessor expands the macros. In most cases, the fortran compiler invoke the C preprocessor if the source file suffix is in capital letters. However, some compilers provide extra flags to tell the compiler to use a C preprocessor. Furthermore, it is important to use the C-like `#include` with the leading '#'-character to include the `SCOREP_User.inc` header file. Otherwise, the inclusion may happen after the C preprocessor ran. As result the fortran compiler complains about unknown preprocessing directives.

Region handles (`my_region_handle`) should be registered in each annotated function/subroutine prologue before use within the associated body, and should not already be declared in the same program scope.

For every region, the region type can be indicated via the region type flag. Possible region types are:

**SCOREP_USER_REGION_TYPE_COMMON** Indicates regions without a special region type.

**SCOREP_USER_REGION_TYPE_FUNCTION** Indicates that the region is a function or subroutine

**SCOREP_USER_REGION_TYPE_LOOP** Indicates that the region is the body of a loop, with the same number of iterations in all locations.

**SCOREP_USER_REGION_TYPE_DYNAMIC** Set this type to create a separate branch in the call-tree for every execution of the region. See Section 5.1.3.

**SCOREP_USER_REGION_TYPE_PHASE** Indicates that this region belongs to a special phase. See Section 5.1.2.

To create a region of combined region types you can connect two or more types with the binary OR-operator, e.g.:

```
SCOREP_USER_REGION_BEGIN( handle, "foo",
                          SCOREP_USER_REGION_TYPE_LOOP |
                          SCOREP_USER_REGION_TYPE_PHASE |
                          SCOREP_USER_REGION_TYPE_DYNAMIC )
```

For function instrumentation in C and C++, Score-P provides macros, which automatically pass the name and function type to Score-P measurement system. The `SCOREP_USER_FUNC_BEGIN` macro contains a variable definition. Thus, compilers that require strict separation of declaration and execution part, may not work with this macro.

C/C++:

```
#include <scorep/SCOREP_User.h>

void foo()
{
    SCOREP_USER_FUNC_BEGIN()
    // do something
    SCOREP_USER_FUNC_END()
}
```

In some cases, it might be useful to have the possibility to define region handles with a global scope. In C/C++, a region handle can be defined at a global scope with `SCOREP_USER_GLOBAL_REGION_DEFINE`. In this case, the `SCOREP_USER_REGION_DEFINE` must be omitted. The `SCOREP_USER_GLOBAL_REGION_DEFINE` must only appear in one file. To use the same global variable in other files, too, declare the global region in other files with `SCOREP_USER_GLOBAL_REGION_EXTERNAL`.

File 1:

```
SCOREP_USER_GLOBAL_REGION_DEFINE( global_handle )

foo()
{
    SCOREP_USER_REGION_BEGIN( global_handle, "phase 1",
                              SCOREP_USER_REGION_TYPE_PHASE)
    // do something
    SCOREP_USER_REGION_END( global_handle )
}
```

File 2:

```
SCOREP_USER_GLOBAL_REGION_EXTERNAL( global_handle )

bar()
{
    SCOREP_USER_REGION_BEGIN( global_handle, "phase 1",
                              SCOREP_USER_REGION_TYPE_PHASE)
    // do something
    SCOREP_USER_REGION_END( global_handle )
}
```

**Note**

These macros are not available in Fortran.

In addition, the macros `SCOREP_USER_REGION_BY_NAME_BEGIN( name, type )` and `SCOREP_USER_REGION_BY_NAME_END( name )` are available. These macros might introduce more overhead than the standard macros but can annotate user regions without the need to take care about the handle struct. This might be useful for automatically generating instrumented code or to avoid global declaration of this variable.

C/C++:

```
#include <scorep/SCOREP_User.h>

/* Application functions are already instrumented with these two calls. */
void instrument_begin(const char* regionname)
{
  /* code added for Score-P instrumentation */
  SCOREP_USER_REGION_BY_NAME_BEGIN( regionname, SCOREP_USER_REGION_TYPE_COMMON
       )
}

void instrument_end(const char* regionname)
{
  SCOREP_USER_REGION_BY_NAME_END( regionname )
}
```

Fortran:

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

subroutine instrument_begin(regionname)
  character(len=*) :: regionname
  SCOREP_USER_REGION_BY_NAME_BEGIN( regionname, SCOREP_USER_REGION_TYPE_COMMON
```

```
      )
end subroutine instrument_begin

subroutine instrument_end(regionname)
  character(len=*) :: regionname
  SCOREP_USER_REGION_BY_NAME_END( regionname )
end subroutine instrument_end
```

**Note**

When using the "BY_NAME" macros in Fortran, be aware of section 12.4.1.1 of the F90/95/2003 standard. If you pass *name* through a dummy argument of a subroutine the length *len* of the character array *name* must be exactly the size of the actual string passed. In the Fortran examples above this is assured by `len=*`.
To ensure correct nesting, avoid automatic compiler instrumentation for these helper functions.

The source files instrumented with Score-P user macros have to be compiled with `-DSCOREP_USER_ENABLE` otherwise `SCOREP_*` calls expand to nothing and are ignored. If the Score-P instrumenter `-user` flag is used, the `SCOREP_USER_ENABLE` symbol will be defined automatically. Also note, that Fortran source files instrumented this way have to be preprocessed with the C preprocessor (CPP).

Manual routine instrumentation in combination with automatic source-code instrumentation by the compiler or PDT leads to double instrumentation of user routines, i.e., usually only user region instrumentation is desired in this case.

## 3.3 Instrumentation for Parameter-Based Profiling

The Score-P user API provides also macros for parameter-based profiling. In parameter-based profiling, the parameters of a function are used to split up the call-path for executions of different parameter values. In Score-P parameter-based profiling is supported for integer and string parameters. To associate a parameter value to a region entry, insert a call to `SCOREP_USER_PARAMETER_INT64` for signed integer parameters, `SCOREP_USER_PARAMETER_UINT64` for unsigned integer parameters, or `SCOREP_USER_PARAMETER_STRING` for string parameters after the region entry (e.g. after `SCOREP_USER_REGION_BEGIN` or `SCOREP_USER_FUNC_BEGIN`).

Fortran:

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

subroutine foo(i, s)
    integer :: i
    character (*) :: s

    SCOREP_USER_REGION_DEFINE( my_region_handle )
    SCOREP_USER_PARAMETER_DEFINE( int_param )
    SCOREP_USER_PARAMETER_DEFINE( string_param )
    SCOREP_USER_REGION_BEGIN( my_region_handle, "my_region",
      SCOREP_USER_REGION_TYPE_COMMON )
    SCOREP_USER_PARAMETER_INT64(int_param, "myint",i)
    SCOREP_USER_PARAMETER_UINT64(uint_param, "myuint",i)
    SCOREP_USER_PARAMETER_STRING(string_param, "mystring",s)

    // do something

    SCOREP_USER_REGION_END( my_region_handle )
 end subroutine foo
```

C/C++:

```
#include <scorep/SCOREP_User.h>

void foo(int64_t myint, uint64_t myuint, char *mystring)
{
  SCOREP_USER_REGION_DEFINE( my_region_handle )
  SCOREP_USER_REGION_BEGIN( my_region_handle, "foo",
      SCOREP_USER_REGION_TYPE_COMMON )
  SCOREP_USER_PARAMETER_INT64("myint",myint)
  SCOREP_USER_PARAMETER_UINT64("myuint",myuint)
  SCOREP_USER_PARAMETER_STRING("mystring",mystring)

  // do something
```

```
  SCOREP_USER_REGION_END( my_region_handle )
}
```

In C/C++, only a name for the parameter and the value needs to be provided. In Fortran, the handle must be defined first with SCOREP_USER_PARAMETER_DEFINE. The defined handle name must be unique in the current scope. The macro SCOREP_USER_PARAMETER_INT64 as well as the macro SCOREP_USER_PARAMETER_STR↩ ING need the handle as the first argument, followed by the name and the value.

## 3.4 Measurement Control Instrumentation

The Score-P user API also provides several macros for measurement control that can be incorporated in source files and activated during instrumentation. The macro SCOREP_RECORDING_OFF can be used to (temporarily) pause recording until a subsequent SCOREP_RECORDING_ON. Just like the already covered user-defined annotated regions, SCOREP_RECORDING_ON and the corresponding SCOREP_RECORDING_OFF must be correctly nested with other enter/exit events. Finally, with SCOREP_RECORDING_IS_ON you can test whether recording is switched on.

Events are not recorded when recording is switched off (though associated definitions are), resulting in smaller measurement overhead. In particular, traces can be much smaller and can target specific application phases (e.g., excluding initialization and/or finalization) or specific iterations. Since the recording switch is process-local, and effects all threads on the process, it can only be initiated outside of OpenMP parallel regions. Switching recording on/off is done independently on each MPI process without synchronization.

**Note**

Switching recording on/off may result in inconsistent traces or profiles, if not applied with care. In particular, if communication is recorded incomplete (e.g. if the send is missing but the corresponding receive event is recorded) it may result in errors during execution or analysis. Furthermore, it is not possible to switch recording on/off from within parallel OpenMP regions. We recommend to use the selective recording interface, instead of the manual on/off switch whenever possible. Special care is required in combination with selective recording (see Section 5.4, which also switches recording on/off.

## 3.5 Source-Code Instrumentation Enabling Online Access

The Online Access interface to the measurement system of Score-P allows remote control of measurement and access to the profile data. The online access interface may not be available on all platforms. To use the Online Access interface, Score-P must have been built with Online Access (OA) support.

The Online Access module requires the user to specify at least one *online access phase*. The online access phase does not show the behavior of a region of type phase as defined in Section 3.2. However, the way to specify an online access phase is similar to manual region instrumentation. The start and end of the online access phase defines the interaction points, where new measurement control commands are applied and data requests are answered.

To insert an online online access phase into the code, the user has to insert the macros SCOREP_USER_OA_P↩ HASE_BEGIN and the corresponding SCOREP_USER_OA_PHASE_END at appropriate locations. These macros must be

- correctly nested with all regions and
- must be potential global synchronization points.

Common practice is to mark the body of the application's main loop as online access phase, in order to utilize the main loop iterations for iterative online analysis. Only the measurements collected inside the OA phase could be configured and retrieved.

Instrumentation can be performed in the following ways, depending on the programming language used.

Fortran:

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

subroutine foo
  SCOREP_USER_REGION_DEFINE( my_region_handle )
  ! more declarations

  SCOREP_USER_OA_PHASE_BEGIN( my_region_handle, "foo",
      SCOREP_USER_REGION_TYPE_COMMON )
  ! do something
  SCOREP_USER_OA_PHASE_END( my_region_handle )

end subroutine foo
```

C/C++:

```
#include <scorep/SCOREP_User.h>

void foo()
{
  SCOREP_USER_REGION_DEFINE( my_region_handle )

  // do something

  SCOREP_USER_OA_PHASE_BEGIN( my_region_handle, "foo",
      SCOREP_USER_REGION_TYPE_COMMON )

  // do something

  SCOREP_USER_OA_PHASE_END( my_region_handle )
}
```

## 3.6 Semi-Automatic Instrumentation of POMP2 User Regions

**Note**

Since Score-P version 1.4, OpenMP instrumentation using OPARI2 no longer activates POMP2 instrumentation implicitly. You need to explicitly add the `-pomp` option to the Score-P instrumenter.

If you manually instrument the desired user functions and regions of your application source files using the POMP2 `INST` directives described below, the Score-P instrumenter `-pomp` flag will generate instrumentation for them. POMP2 instrumentation directives are supported for Fortran and C/C++. The main advantages are that

- being directives, the instrumentation is ignored during "normal" compilation and
- this semi-automatic instrumentation procedure can be used when fully automatic compiler instrumentation is not supported.

The `INST BEGIN/END` directives can be used to mark any user-defined sequence of statements. If this block has several exit points (as is often the case for functions), all but the last have to be instrumented by `INST ALTEND`.

Fortran:

```
subroutine foo(...)
    !declarations
    !POMP$ INST BEGIN(foo)
    ...
    if (<condition>) then
        !POMP$ INST ALTEND(foo)
        return
    end if
    ...
    !POMP$ INST END(foo)
end subroutine foo
```

C/C++:

```
void foo(...)
{
    /* declarations */
    #pragma pomp inst begin(foo)
```

```
    ...
    if (<condition>)
    {
        #pragma pomp inst altend(foo)
        return;
    }
    ...
    #pragma pomp inst end(foo)
}
```

At least the main program function has to be instrumented in this way, and additionally, one of the following should be inserted as the first executable statement of the main program:

Fortran:

```
program main
    ! declarations
    !POMP$ INST INIT
    ...
end program main
```

C/C++:

```
int main(int argc, char** argv)
{
    /* declarations */
    #pragma pomp inst init
    ...
}
```

By default, the source code is preprocessed before POMP2 instrumentation happens. For more information on the preprocessing, see Section 3.7.

## 3.7 Preprocessing before POMP2 and OpenMP instrumentation

By default, source files are preprocessed before the semi-automatic POMP2 instrumentation or the OpenMP construct instrumentation with OPARI2 happens. This ensures, that all constructs and regions that might be contained in header files, templates, or macros are properly instrumented. Furthermore, conditional compilation directives take effect, too. The necessary steps are performed by the Score-P instrumenter tool.

Some Fortran compilers do not regard information about the original source location that the preprocessing leaves in the preprocessed code. This causes wrong source code information for regions from compiler instrumentation, and manual source code instrumentation. However, these compilers also disregard the source code information left by OPARI2. Thus, for these compilers the source location information is incorrect anyway.

If the preprocessing is not desired, you can disable it with the `-nopreprocess` flag. In this case the instrumentation is performed before the preprocessing happens. In this case constructs and regions in header files, macros, or templates are not instrumented. Conditional compilation directives around constructs may also lead to broken instrumentation.

**Note**

If a parallel region is not instrumented, the application will crash during runtime.

The preprocessing does not work in combination with PDT source code instrumentation. Thus, if PDT instrumentation is enabled, it changes the default to not preprocess a source file. If you manually specify preprocessing and PDT source code instrumentation, the instrumenter will abort with an error.

## 3.8 Source-Code Instrumentation Using PDT

If Score-P has been configured with PDToolkit support, automatic source-code instrumentation can be used as an alternative instrumentation method. In this case, the source code of the target application is pre-processed before

compilation, and appropriate Score-P user API calls will be inserted automatically. However, please note that this feature is still somewhat experimental and has a number of limitations (see Section 3.8.1).

To enable PDT-based source-code instrumentation, call `scorep` with the `-pdt` option, e.g.,

```
scorep --pdt  mpicc -c foo.c
```

This will by default instrument all routines found in `foo.c`. (To avoid double instrumentation, automatic compiler instrumentation is disabled when using Source-Code Instrumentation with PDT. However, if you you can enforce additional compiler instrumentation with `--compiler`.) The underlying PDT instrumentor supports a set a instrumentation options, which can be set like

```
scorep --pdt="-f <inclusion/exclusion file>"  mpicc -c foo.c
```

This particular option for example can be used to manually include/exclude specific functions from the instrumentation process. The respective file format is described here. Please check the documentation about the `tau_instrumentor` for more valid options.

### 3.8.1 Limitations

Currently the support for the PDT-based source-code instrumenter still has a number of limitations:

- When instrumenting Fortran 77 applications, the inserted instrumentation code snippets do not yet adhere to the Fortran 77 line length limit. Typically, it is possible to work around this issue by supplying extra command line flags (e.g., `-ffixed-line-length-132` or `-qfixed=132`) to the compiler.
- Code in C/C++ header files as well as included code in Fortran (either using the C preprocessor or the `include` keyword) will currently not be instrumented.
- Support for C++ templates and classes is currently only partially implemented.
- Advanced TAU instrumentation features such as static/dynamic timers, loop, I/O and memory instrumentation are not yet supported. Respective entries in the selective instrumentation file will be ignored.

## 3.9 Enforce Linking of Static/Shared Score-P Libraries

If the Score-P was build with shared libraries and with static libraries, the instrumenter uses the compiler defaults for linking. E.g. if the compiler chooses shared libraries by default, the instrumenter will link your application with the shared Score-P libraries. Furthermore, the linking is affected by parameters in the original link command. E.g. if your link command contains a `-Bstatic` flag, afterwards appended Score-P libraries are also linked statically.

If you want to override the default and enforce linking of static or dynamic Score-P libraries, you can add the flag `-static` or `-dynamic` for the instrumenter. E.g. a command to enforce static linking can look like:

```
scorep --static mpicc foo.c -o foo
```

In this case, the linking against the static version of the Score-P libraries is enforced.

If enforcing static or dynamic linking is not possible on your system, e.g., because no static/dynamic Score-P libraries are installed, the instrumenter will abort with an error. You can determine whether `-static` or `-dynamic` is available from the output of `scorep -help`. If the `-static` or `-dynamic` flags are not shown, then they are not available.

# Chapter 4

# Application Sampling

This document describes how to use the sampling options within Score-P.

## 4.1 Introduction

Score-P supports sampling that can be used concurrently to instrumentation to generate profiles and traces. In the following, we will describe how sampling differs from instrumentation. Reading this text will help you to interpret resulting performance data. However, if you are aware of how sampling works, you can skip the preface.

In our context, we understand sampling as a technique to capture the behavior and performance of programs. We interrupt the running programs at a specified interval (the sampling period) and capture the current state of the program (i.e., the current stack) and performance metrics (e.g., PAPI). The obtained data is than further stored as a trace or a profile and can be used to analyze the behavior of the sampled program.

Before version 2.0 of Score-P, only instrumentation-based performance analysis had been possible. Such an instrumentation relies on callbacks to the measurement environment (`instrumentation points`), e.g., a function enter or exit. The resulting trace or profile presented the exact runtimes of the functions, augmented with performance data and communication information. However, instrumentation introduces a constant overhead for each of the instrumentation points. For small instrumented functions, this constant overhead can be overwhelming.

Sampling provides the opportunity to prevent this overwhelming overhead, and even more, the overhead introduced by sampling is controllable by setting the sampling rate. However, the resulting performance data is more "fuzzy". Not every function call is captured and thus the resulting data should be analyzed carefully. Based on the duration of a function and the sampling period, a function call might or might not be included in the gathered performance data. However, statistically, the profile information is correct. Additionally, the sampling rate allows to regulate the trade-off between overhead and correctness, which is not possible for instrumentation.

In Score-P we support both instrumentation and sampling. This allows you for example to get a statistical overview of your program as well as analyzing the communication behavior. If a sample hits a function that is known to the measurement environment via instrumentation (e.g., by OPARI2), the sample will show the same function in the trace and the profile.

## 4.2 Prerequisites

This version of Score-P provides support for sampling. To enable sampling, several prerequisites have to be met.

- **libunwind**:
  Additionally to the usual configuration process of Score-P, `libunwind` is needed. `libunwind` can be installed using a standard package manager or by downloading the latest version from

  `http://download.savannah.gnu.org/releases/libunwind/`

This library must be available at your system to enable sampling. In our tests, we used the most current stable version (1.1) as previous versions might result in segmentation faults.

- **Sampling Sources**:
  Sampling sources generate interrupts that trigger a sample. We interface three different interrupt generators, which can be chosen at runtime.

  1. **Interval timer**:
     Interval timers are POSIX compliant but provide a major drawback: They cannot be used for multi-threaded programs, but only for single-threaded ones. We check for `setitimer` that is provided by `sys/time.h`.
  2. **PAPI**:
     We interface the PAPI library, if it is found in the configure phase. The PAPI interrupt source uses overflowing performance counters to interrupt the program. This source can be used in multi-threaded programs. Due to limitations from the PAPI library, PAPI counters will not be available if PAPI sampling is enabled. However, you can use perf metrics. E.g.,

     ```
     export SCOREP_METRIC_PERF=instructions:page-faults
     ```

  3. **perf**:
     `perf` is comparable to PAPI but much more low-level. We directly use the system call. This source can be used in multi-threaded programs. PAPI counters are available if `perf` is used as an interrupt source. Currently we only provide a cycle based overflow counter via `perf`.

  We recommend using `PAPI` or `perf` as interrupt sources. However, these also pose a specific disadvantage when power saving techniques such as DVFS or idle states are active on a system. In this case, a constant sampling interval cannot be guaranteed. If, for example, an application calls a sleep routine, then the cycle counter might not increase as the CPU might switch to an idle state. This can also influence the result data. Such idling times can also be introduced by OpenMP runtimes and can be avoided by setting the block times accordingly or setting the environment variable `OMP_WAIT_POLICY` to `ACTIVE`.

## 4.3 Configure Options

### 4.3.1 libunwind

If libunwind is not installed in a standard directory, you can provide the following flags in the configure step:

```
--with-libunwind=(yes|no|<Path to libunwind installation>)
                        If you want to build scorep with libunwind but do
                        not have a libunwind in a standard location, you
                        need to explicitly specify the directory where it is
                        installed. On non-cross-compile systems we search
                        the system include and lib paths per default [yes];
                        on cross-compile systems, however, you have to
                        specify a path [no]. --with-libunwind is a shorthand
                        for --with-libunwind-include=<Path/include> and
                        --with-libunwind-lib=<Path/lib>. If these shorthand
                        assumptions are not correct, you can use the
                        explicit include and lib options directly.
--with-libunwind-include=<Path to libunwind headers>
--with-libunwind-lib=<Path to libunwind libraries>
```

## 4.4 Sampling Related Score-P Measurement Configuration Variables

The following lists the Score-P measurement configuration variables which are related to sampling. Please refer to the individual variables for a more detailed description.

- SCOREP_ENABLE_UNWINDING
- SCOREP_SAMPLING_EVENTS
- SCOREP_SAMPLING_SEP
- SCOREP_TRACING_CONVERT_CALLING_CONTEXT_EVENTS

## 4.5 Use Cases

### 4.5.1 Enable unwinding in instrumented programs

Additionally to the instrumentation, you now see where the instrumented region has been called. A pure MPI instrumentation for example does not tell you which functions have been issuing communications. With unwinding enabled, this is revealed and stored in the trace or profile.

Instrument your program, e.g. with MPI instrumentation enabled.

```
scorep mpicc my_mpi_code.c -o my_mpi_application
```

Set the following environment variables:

```
export SCOREP_ENABLE_UNWINDING=true
export SCOREP_SAMPLING_EVENTS=
```

Run your program

```
mpirun -np 16 ./my_mpi_application
```

### 4.5.2 Instrument a hybrid parallel program and enable sampling

In this example you get rid of a possible enormous compiler instrumentation overhead but you are still able to see statistical occurrences of small code regions. The NAS Parallel Benchmark BT-MZ for example uses small sub functions within OpenMP parallel functions that increase the measurement overhead significantly when compiler instrumentation is enabled.

Instrument your program, e.g. with MPI and OpenMP instrumentation enabled.

```
scorep mpicc -fopenmp my_hybrid_code.c -o my_hybrid_application
```

Note: If you use the GNU compiler and shared libraries of Score-P you might get errors due to undefined references depending on your gcc version. Please add `-no-as-needed` to your scorep command line. This flag will add a GNU ld linker flag to fix undefined references when using shared Score-P libraries. This happens on systems using `-as-needed` as linker default. It will be handled transparently in future releases of Score-P.

Set the following environment variables:

```
export SCOREP_ENABLE_UNWINDING=true
```

If you want to use a sampling event and period differing from the default settings you additionally set:

```
export SCOREP_SAMPLING_EVENTS=PAPI_TOT_CYC@1000000
```

Run your program

```
mpirun -np 16 ./my_mpi_application
```

## 4.6 Test Environment

Example

### 4.6.1 Instrument NAS BT-MZ code

```
cd <NAS_BT_MZ_SRC_DIR>
vim config/make.def
```

Set add the Score-P wrapper to your MPI Fortran compiler.

```
MPIF77 = scorep mpif77
```

Recompile the NAS BT-MZ code.

```
make clean
make bt-mz  CLASS=C  NPROCS=128
```

### 4.6.2 Run instrumented binary

```
cd bin
sbatch run.slurm
```

Batch script example:

```
#!/bin/bash
#SBATCH -J NAS_BT_C_128x2
#SBATCH --nodes=32
#SBATCH --tasks-per-node=4
#SBATCH --cpus-per-task=2
#SBATCH --time=00:30:00

export OMP_NUM_THREADS=2

export NPB_MZ_BLOAD=0

export SCOREP_ENABLE_TRACING=true
export SCOREP_ENABLE_PROFILING=false
export SCOREP_ENABLE_UNWINDING=true
export SCOREP_TOTAL_MEMORY=200M
export SCOREP_SAMPLING_EVENTS=perf_cycles@2000000
export SCOREP_EXPERIMENT_DIRECTORY='bt-mz_C.128x2_trace_unwinding'

srun ./bt-mz_C.128
```

# Chapter 5

# Application Measurement

If an application was instrumented with Score-P, you will get an executable, which you can execute like the uninstrumented application. After the application run, you will find an experiment directory in your current working directory, which contains all recorded data. The experiment directory has the format `scorep-YYYYMMDD_HHMM_XXXX↩XXXX`, where `YYYYMMDD` and `HHMM` encodes the date followed by a series of random numbers. You may specify the name of the experiment directory by setting the environment variable `SCOREP_EXPERIMENT_DIRECTO↩RY` to the desired name of the directory. If the directory already exists, the existing directory will be renamed by appending a date like above by default. You can let Score-P abort the measurement immediately by setting `SC↩OREP_OVERWRITE_EXPERIMENT_DIRECTORY` to `false` if the experiment directory already exists. This has only an effect if `SCOREP_EXPERIMENT_DIRECTORY` was set too.

In general, you can record a profile and/or a event trace. Whether a profile and/or a trace is recorded, is specified by the environment variables `SCOREP_ENABLE_PROFILING` and `SCOREP_ENABLE_TRACING`. If the value of this variables is zero or `false`, profiling/tracing is disabled. Otherwise Score-P will record a profile and/or trace. By default, profiling is enabled and tracing is disabled.

You may start with a profiling run, because of its lower space requirements. According to profiling results, you may configure the trace buffer limits, filtering or selective recording for recording traces.

Score-P allows to configure several parameters via environment variables. See Appendix E for a detailed description of how to configure the measurement.

## 5.1 Profiling

Score-P implements a call-tree based profiling system. Every node in the call tree represent a recorded region. The edges of the tree represent the caller-callee relationship: The children of a node are those regions, that are entered/exited within a region. The path from the root to an arbitrary node, represents a call-path. Thus, every node in the tree identifies also the call-path from the root to itself.

Together with a node, the statistics for the call-path are stored. By default, the runtime and the number of visits are recorded. Additionally, hardware counters can be configured and are stored for every call-path. User defined metrics are only stored in those nodes, where the metric was triggered.

For enabling profiling, set the `SCOREP_ENABLE_PROFILING` environment variable to 1 or `true`. After the execution of your application you will then find a file, named `profile.cubex` in your measurement directory, which you can display with the CUBE4 with `cube-qt profile.cubex`. The name of the profile can be changed through the environment variable `SCOREP_PROFILING_BASE_NAME`. The extension `.cubex` will be appended to the base name you specify in the environment variable `SCOREP_PROFILING_BASE_NAME`.

By default, Score-P writes the profile in CUBE4 base format. Hereby, for every metric contains one value, usually only the sum. However, Score-P allows to store the profile in two other formats. To change the default format, set the environment variable `SCOREP_PROFILING_FORMAT`. Please refer to the description of this variable for possible values.

Score-P records a call tree profile. The maximum call-path depth that is recorded is limited to 30, by default. This

avoids extremely large profiles for recursive calls. However, this limit can be changed with the environment variable SCOREP_PROFILING_MAX_CALLPATH_DEPTH.

### 5.1.1 Parameter-Based Profiling

Parameter-based profiling allows to separate the recoded statistics for a region, depending on the values of one or multiple parameters. In the resulting call-tree, each occurred parameter-value will create a sub-node of the region. Every parameter has a parameter name. Thus, if multiple parameters are used, they can be distinguished and split the call-tree in the order of the parameter events. In the final call-tree it looks like every parameter-name/parameter-value pair is a separate region.

Currently, the only source for parameter events is manual instrumentation (see Section 3.3).

### 5.1.2 Phase Profiling

Phase-profiling allows, to group the execution of the application into logical phases. Score-P records a separate call-tree for every phase in the application. A phase starts when a region of type SCOREP_USER_REGION_TYPE_PHASE (see Section 3.2) is entered. If the region is exited, the phase is left. If two phases are nested, then the outer phase is left, when the inner phase is entered. If the inner phase is exited, the outer phase is re-entered. Figure 5.1 shows the difference in the call-tree if the regions with the names phase1 and phase2 are not of type SCOREP_USER_REGION_TYPE_PHASE on the left side and the forest if they are of type SCOREP_USER_REGION_TYPE_PHASE on the right side.

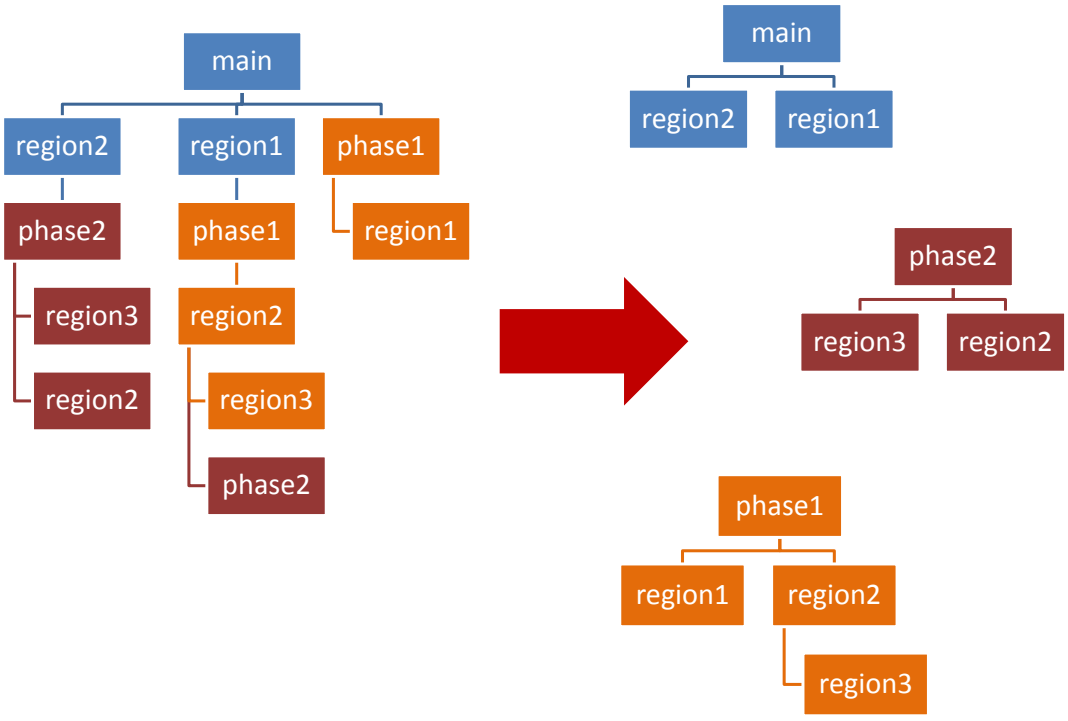


Figure 5.1: Call-tree changes when using phases. The left side shows the calltree if no region is of type phase. The right side shows the call-tree forest with phases.

If the phase consists of multiple partitions, and thus cannot be enclosed by a single code region, all code-regions that form the phase must have the same region handle. The possibility to define global region handles in C/C++ might be useful for the definition of phases that are have multiple partitions (see Section 3.2).

### 5.1.3 Dynamic Region Profiling

When profiling, multiple visits of a call-path are summarized. However, e.g, for investigations in time-dependent behavior of an application, each iteration of a main loop (or some other region) should create a separate profile sub-tree. For such cases, Score-P allows to define regions to by of type dynamic. For dynamic regions, each entry of the region will create a separate path. For this cause, the Score-P profiling system creates an extra parameter, named `instance`. On each visit to a dynamic region, the instance parameter for this call-path is increased and triggered automatically. Thus, the every visit to a dynamic region generates a separate subtree in the profile.

As an example, let us assume that an application contains the regions `foo` and `main`, where `main` calls `foo` three times. A regular profile would show two call-pathes:

- `main`
- `main/foo`

If `foo` is a dynamic region, the profile would contain additional sub-nodes for each visit of `foo`. The resulting profile would contain the following call-pathes:

- `main`
- `main/foo`
- `main/foo/instance=0`
- `main/foo/instance=1`
- `main/foo/instance=2`

In this case `main/foo` contains the summarized statistics for all 3 visits, while `main/foo/instance=0` contains the statistics for the first visits of the call-path.

**Note**

The enumeration of the instance is per call-path and not per dynamic region. In particular, if a dynamic region foo appears in 2 call-paths, it has 2 instance number 0, one in both call-paths. It is not a global enumeration of the visits to foo but enumerates the visits of foo in a particular call-path from 0 to N.

Currently, the only possibility to define dynamic regions is via the manual region instrumentation, described in Section 3.2.

**Note**

Using dynamic regions can easily create very large profiles. Thus, use this feature with care. If you are only interested in some parts of the application, selective recording (see Section 5.4) might be a memory space save alternative. Furthermore, you can use clustering (see Section 5.1.4) to reduce the memory requirements.

### 5.1.4 Clustering

Clustering allows to reduce the memory requirements of a dynamic region, by clustering similar sub-trees into one cluster. A visualization tool (like CUBE 4) might expand the clusters back to single iterations transparently. You can enable/disable clustering via the environment variable `SCOREP_PROFILING_ENABLE_CLUSTERING`. By default, clustering is enabled.

Currently, clustering is limited to the instances of one node in the call-tree. If a dynamic region appears on several call-paths, Score-P will only cluster one, and generate separate sub-trees for every iterations in all other call-paths. By default, Score-P will cluster the instances of that dynamic region call-path that it enters first. If you have only one call-path where a dynamic region occurs (e.g., if the body of the main loop is the only dynamic region), this region will be clustered automatically. Otherwise, we recommend to specify the region you want to cluster in the environment variable `SCOREP_PROFILING_CLUSTERED_REGION`.

**Note**

If the selected region appears on multiple call-paths, only one of them is clustered. Score-P chooses the call-path of that regions that it enters first. In particular, if the selected dynamic region is nested into itself, the outermost occurrence is clustered.

Furthermore, the clustered region must not be inside of a parallel region, but must be at a sequential part of the program. However, the clustered region may contain parallel regions.

Clustering is a lossy compression mechanism. The accuracy increases if more clusters are available. On the downside, more clusters require more memory. You can specify the number of clusters you want by setting the environment variable SCOREP_PROFILING_CLUSTER_COUNT to the number of cluster you want to have. The default cluster number is 64.

Furthermore, you can enforce a minimal structural similarity of instances of a cluster. Clusters that fit the minimal structural similarity requirements belong to the same equivalence class. Only instances of the same equivalence class will be clustered together. If you have more equivalence classes than the number of clusters you specified in SCOREP_PROFILING_CLUSTER_COUNT, the maximal number of clusters is increased. Thus, you might get more clusters than you specified.

The minimal structural similarity is defined by the clustering mode which can be set via the environment variable SCOREP_PROFILING_CLUSTERING_MODE. Please refer to the description of this variable for possible values.

### 5.1.5 Enabling additional debug output on inconsistent profiles

If the Score-P profiling system detects inconsistencies during measurement, it stops recording the profile and prints an error message. Examples for reasons of an inconsistent profile are, if the nesting order of function entries and exits is broken, or events appear for an uninitialized thread. This might indicate an bug of the profile, but typically the cause is an erroneous instrumentation. E.g. if manual instrumentation is applied, but not all possible exit points of a function are instrumented.

In order to support debugging of manual instrumentation, or during the development of own automatic instrumentation techniques, the profile can write additional information about its current state in a textual form into a file. This output may contain the following information:

- The current call stack of the failing thread
- The profile structure of the failing thread
- The complete profile structure

None of the three entries is guaranteed to appear in the output, it depends on the current state of the profile. It might not be possible to provide any output at all. Furthermore, the online representation of the profile structure may differ from the final profile structure.

You can enable this additional output by setting the environment variable SCOREP_PROFILING_ENABLE_CORE_FILES to `true`. Then, if the profile detects an inconsistency, it will write a core file into your measurement directory. If an inconstant profiles is detected on multiple locations, every location where an inconsistency is detected will write a core file. Thus, it is not recommended, to enable this feature for large scale runs.

## 5.2 Tracing

Score-P can write events to OTF2 traces. By setting the environment variable SCOREP_ENABLE_TRACING, you can control whether a trace is recorded. If the value is `0` or `false` no trace is recorded, if the value is non-zero or `true`, a trace is recorded. If the variable is not specified, Score-P records traces on default. After trace recording you will find the OTF2 anchor file, named `trace.otf2` in the experiment directory, along with the trace data.

## 5.3 Filtering

When automatic compiler instrumentation or automated source code instrumentation with PDT has been used to instrument user-level source-program routines, there are cases where measurement and associated analysis are degraded, e.g., by frequently-executed, small and/or generally uninteresting functions, methods and subroutines.

A measurement filtering capability is therefore only supported for compiler instrumented regions, regions instrumented with the user API from Score-P (see section 3.2), regions instrumented with the user API from OPARI2 (see section 3.6), and CUDA device and host activities (see Section 5.8). See section 5.7.1 to restrict the recording of MPI features and the OPARI2 documentation of `-disable` to restrict instrumentation of OpenMP directives. This `-disable` flag can than be passed on to the OPARI2 invocation with the `-opari=<parameter-list>` flag of the Score-P instrumenter. Because PDT instrumentation (Section 3.8) inserts Score-P user API instrumentation those regions can be filtered, too. Regions can be filtered based on their region name (e.g., their function name) or based on the source file, in which they are defined.

A file that contains the filter definition can be specified via the environment variable `SCOREP_FILTERING_FILE`. If no filter definition file is specified, all instrumented regions are recorded. For filtered regions, the enter/exit events are not recorded in trace and profile.

The filter definition file can contain two blocks:

- One block defines filter rules for filtering regions based on the source files they are defined in.
- One filter block defined rules for region names.

When the filter rules are applied, the source file name filter is evaluated first. If a region is filtered because it appears in a filtered source file, it cannot be included by the function name filter. If a region was defined in a not-filtered source file, the region name filter is evaluated. This means, events for a region are not recorded if they are filtered by the source file filter or the region name filter. Events for a region are recorded if the region is neither filtered by the source file filter nor by the region name filter. If one of the both filter blocks is not specified, it is equivalent to an empty filter block.

Beside the two filter blocks, you may use comments in the filter definition file. Comments start with the character '#' and is terminated by a new line. You may use comments also inside the filter blocks. If a region name or source file name contains '#', you must escape it with a backslash.

### 5.3.1 Source File Name Filter Block

The filter block for source file names, must be enclosed by `SCOREP_FILE_NAMES_BEGIN` and `SCOREP_FILE_NAMES_END`. In between you can specify an arbitrary number of include and exclude rules which are evaluated in sequential order. At the beginning all source files are included. Source files that are excluded after all rules are evaluated, are filtered.

An exclude rule starts with the keyword `EXCLUDE` followed by one or multiple white-space separated source file names. Respectively, include rules start with `INCLUDE` followed by one or multiple white-space separated file names. For the specification of file names, bash-like wild-cards are supported. In particular, the '*' wild-card matches an string of arbitrary length, the '?' matches exactly one arbitrary character, or within `[]` you may specify multiple options.

**Note**

Unlike bash, a '*' may match a string that contains slashes. E.g, you may use the '*' wild-card for path prefixes.

An example source file filter block could look like this:

```
SCOREP_FILE_NAMES_BEGIN # This is a comment
  EXCLUDE */filtering/filter*
  INCLUDE */filter_test.c
SCOREP_FILE_NAMES_END
```

**Note**

The keywords (SCOREP_FILE_NAMES_BEGIN, SCOREP_FILE_NAMES_END, EXCLUDE, and INCLUDE) are case-sensitive.

The filtering is based on the filenames as seen by the measurement system. Depending on instrumentation method and compiler the actual filename may contain the absolute path, a relative path or no path at all. The instrumentation tool tries to create as much absolute paths as possible. Paths are simplified before comparison to a rule. E.g. it removes path/../, /./ and multiple slashes. You may look up the actual filename in the resulting output of the measurement.

### 5.3.2 Region Name Filter Block

The filter block for the region names, must be enclosed by SCOREP_REGION_NAMES_BEGIN and SCOREP_REGION_NAMES_END. In between you can specify an arbitrary number of include and exclude rules which are evaluated in sequential order. At the beginning, all regions are included. Regions that are excluded after all rules are evaluated, are filtered.

**Note**

Regions that are defined in source files that are filtered, are excluded due to the source file filter. They cannot be included anymore by an include rule in the region filter block.

An exclude rule starts with the keyword EXCLUDE followed by one or multiple white-space separated region names. Respectively, include rules start with INCLUDE followed by one or multiple white-space separated expressions. For the specification of region names, bash-like wild-cards are supported. In particular, the '*' wild card matches an string of arbitrary length, the '?' matches exactly one arbitrary character, or within [] you may specify multiple options.

An example region filter block could look like this:

```
SCOREP_REGION_NAMES_BEGIN
  EXCLUDE *
  INCLUDE bar foo
          baz
          main
SCOREP_REGION_NAMES_END
```

In this example, all but the functions bar, foo, baz and main are filtered.

The filtering is based on the region names as seen by the measurement system. Depending on instrumentation method and compiler the actual region name may be mangled, or decorated. Thus, you may want to inspect the profile to determine the name of a region inside the measurement system.

In some cases, the instrumentation provides mangled names, which are demangled by Score-P. In this cases, Score-P uses the demangled form for display in profile and trace definitions, and thus, the demangled form should be used in the filter file. However, The MANGLED keyword marks a filter rule to be applied on the mangled name, if a different mangled name is available. If no mangled name is available, the rule is applied on the displayed name instead. The MANGLED keyword must appear inside of an include rule or exclude rule. All patterns of the rule that follow the MANGLED keyword, are applied to the mangled name, if the mangled name is available.

In the following example, foo and baz are applied to the mangled name, while bar and main are applied on the displayed name.

```
SCOREP_REGION_NAMES_BEGIN
  EXCLUDE *
  INCLUDE bar MANGLED foo
          baz
  INCLUDE main
SCOREP_REGION_NAMES_END
```

The displayed name may also be mangled if no demangled form is available. It is not necessary to prepend rules with the MANGLED keyword if the displayed name is mangled, but only if a mangled name is available that differs from the displayed name.

**Note**

The keywords (e.g., `EXCLUDE`, `INCLUDE`, `SCOREP_REGION_NAMES_BEGIN`, `SCOREP_REGION_NAMES_END`, and `MANGLED` are case-sensitive.
The GCC plug-in based function instrumentation supports all above mentioned filtering features when using the `--instrument-filter` flag to the Score-P instrumenter (see Sec. 3.1). The filter file is then used during compilation time but the use of such a filter file during runtime is still possible. The usage of the filter during compilation time removes the overhead of runtime filtering.

## 5.4 Selective Recording

Score-P experiments record by default all events during the whole execution run. If tracing is enabled the event data will be collected in buffers on each process that must be adequately sized to store events from the entire execution.

Instrumented routines which are executed frequently, while only performing a small amount of work each time they are called, have an undesirable impact on measurement. The measurement overhead for such routines is large in comparison to the execution time of the uninstrumented routine, resulting in measurement dilation. Recording these events requires significant space and analysis takes longer with relatively little improvement in quality. Filtering can be employed during measurement (described in Section 5.3) to ignore events from compiler-instrumented routines or user-instrumented routines.

Another possibility is not to record the whole application run. In many cases, only parts of the application are of interest for analysis (e.g. a frequently performed calculation) while other parts are of less interest (e.g., initialization and finalization) for performance analysis. Or the calculation itself shows iterative behavior, where recording of one iteration would be sufficient for analysis. Restricting recording to one or multiple time intervals during measurement would reduce the required space and overhead. This approach is called selective recording.

Score-P provides two possibilities for selective recording.

- A configuration file can specify recorded regions. The entry and exit of those regions define an interval during which events are recorded.
- With user instrumentation, the recording can be manually switched on /off. (See Section 3.2).

Switching recording on or off, can result in inconsistent traces or profiles, if not applied with care. Especially, switching recording on/off manually via `SCOREP_RECORDING_ON` and `SCOREP_RECORDING_OFF` from the Score-P user instrumentation macros is not recommended. Inconsistent traces may result in errors or deadlocks during analysis, or show unusable data. The consistency is endangered if:

- OpenMP events are missing in one thread while other threads have them. Furthermore, the OpenMP parallel region events are required if any event inside a parallel region is recorded. To prevent inconsistencies from incomplete recording of OpenMP events, it is not possible to switch recording on/off from inside a parallel region
- MPI a communication is only recorded partially, e.g. if a send is missing, but the corresponding receive on another process is recorded. To ensure recording of complete communication is the responsibility of the user.
- enter/exit events are not correctly nested.

How recording can be controlled through Score-P macros which are inserted in the application's source code, is explained in Section 3.2. Thus, this section focuses on first possibility, where the user specify recorded regions via a configuration file. Selective recording affects tracing and profiling.

For selective recording, you can specify one or multiple traced regions. The recording is enabled when a recorded region is entered. If the region is exited, recording of events is switched off again. If a recorded region is called inside another recorded region, thus, the recording is already enabled, it will not disable recording of it exits, but recording will be switched off, if all recorded regions are exited.

For recorded regions only regions from Score-P user instrumentation can be selected. If regions from other instrumentation methods are specified in the configuration file for selective recording, they are ignored.

For a recorded region, the recording can be restricted to certain executions of that region. Therefor, the enters for a recorded region are counted, and a particular execution can be specified by the number of its enter. If a recorded region is called recursively, the recording is only switched off, if the exit is reached, that corresponds to the enter that enabled recording.

The configuration file is a simple text file, where every line contains the name of exactly one region. Optionally, a comma-separated list of execution numbers or intervals of execution numbers can be specified. A configuration file could look like follows:

```
foo
bar 23:25, 50, 60:62
baz 1
```

This configuration file would record all executions of foo, the executions 23, 24, 25, 50, 60, 61, and 62 of bar, and the second (numbering starts with 0) execution of baz.

To apply the selective recording configuration file to a measurement run of your application, set the environment variable SCOREP_SELECTIVE_CONFIG_FILE to the configuration file and run your instrumented application. If SCOREP_SELECTIVE_CONFIG_FILE is empty, or the given file cannot be opened, the whole application run will be recorded (no selective recording will apply).

## 5.5 Trace Buffer Rewind

Introducing a long-term event-trace recording mode, the trace buffer rewind feature allows to discard the preceding section of the event trace at certain control points or phase markers. The live decision whether to keep or discard a section can depend on the presence or absence of certain behaviour patterns as well as on similarity or difference with other sections.

Based on user regions (see 3.2), three macros are given which control the rewind. These are:

```
  // to define a local region handle based on the function
  // SCOREP_USER_REGION_DEFINE( ... )
SCOREP_USER_REWIND_DEFINE( regionHandle )
  // similar to SCOREP_USER_REGION_BEGIN( ... )
SCOREP_USER_REWIND_POINT( regionHandle, "name" )
  // similar to SCOREP_USER_REGION_END( ... )
  // w/ additional parameter to control the rewind (yes or no)
SCOREP_USER_REWIND_CHECK( regionHandle, boolean )
```

The user has to specify whether or not a rewind is requested with a boolean variable in the SCOREP_USER_R↩ EWIND_CHECK function. There are two different approaches what to do with the rewind region in the trace based on the boolean variable. If the boolean variable is true, the trace buffer will be reset to an old snapshot and after that rewind region enter and leave events will be written into the trace buffer to mark the presence of the trace buffer rewind. This rewind region then looks like a normal user-defined region in the trace. If the variable is false, than no events of the rewind region are written into the trace, so that the trace buffer looks like the user never instrumented the code w/ rewind regions. Trace buffer flushes have an impact on the rewind regions, i.e. if a flush occurs all previous stored rewind points (which are not "checked", i.e. the flush is in between the region) will be deleted and the SCOREP_USER_REWIND_CHECK function won't write the enter/leave events into the trace independently from the boolean variable. Wrong nested rewind regions are handled as follows:

```
SCOREP_USER_REWIND_POINT( point 1, ...);
... do stuff ...
SCOREP_USER_REWIND_POINT( point 2, ...);
... do stuff ...
SCOREP_USER_REWIND_CHECK( point 1, true );
... do stuff ...
SCOREP_USER_REWIND_CHECK( point 2, true );
```

The check for point 2 would corrupt the trace buffer, so point 2 would be deleted and ignored in the second check.

## 5.6 Recording Performance Metrics

If Score-P has been built with performance metric support it is capable of recording performance counter information. To request the measurement of certain counters, the user is required to set individual environment variables. The user can leave these environment variables unset to indicate that no counters are requested. Requested counters will be recorded with every enter/exit event.

### 5.6.1 PAPI Hardware Performance Counters

Score-P provides the possibility to query hardware performance counters and include these metrics into the trace and/or profile. Score-P uses the Performance Application Programming Interface (PAPI) to access hardware performance counters. Recording of PAPI performance counters is enabled by setting the environment variable SCOREP_METRIC_PAPI to a comma-separated list of counter names. Counter names can be any PAPI preset names or PAPI native counter names.

Example:

```
SCOREP_METRIC_PAPI=PAPI_FP_OPS,PAPI_L2_TCM
```

This will record the number of floating point instructions and level 2 cache misses. If any of the requested counters is not recognized, program execution will be aborted with an error message. The PAPI utility programs `papi_avail` and `papi_native_avail` report information about the counters available on the current platform.

If you want to change the separator used in the list of PAPI counter names, set the environment variable SCOREP_METRIC_PAPI_SEP to the desired character.

**Note**

In addition it is possible to specify metrics that will be recorded only by the initial thread of a process. Please use SCOREP_METRIC_PAPI_PER_PROCESS for that reason.

### 5.6.2 Resource Usage Counters

Besides PAPI, Resource Usage Counters can be recorded. These metrics use the Unix system call `getrusage` to provide information about consumed resources and operating system events such as user/system time, received signals, and number of page faults. The manual page of `getrusage` provides a list of resource usage counters. Please note that the availability of specific counters depends on the operating system.

You can enable recording of resource usage counters by setting the SCOREP_METRIC_RUSAGE environment variable. The variable should contain a comma-separated list of counter names.

Example:

```
SCOREP_METRIC_RUSAGE=ru_utime,ru_stime
```

This will record the consumed user time and system time. If any of the requested counters is not recognized, program execution will be aborted with an error message.

**Note**

Please be aware of the scope of displayed resource usage statistics. Score-P records resource usage statistics for each individual thread, if the output while configuring your Score-P installation contains something like

```
RUSAGE_THREAD support: yes
```

Otherwise, the information displayed is valid for the whole process. That means, for multi-threaded programs the information is the sum of resources used by all threads in the process.

A shorthand to record all resource usage counters is

```
SCOREP_METRIC_RUSAGE=all
```

However, this is not recommended as most operating systems does not support all metrics.

If you want to change the separator used in the list of resource usage metrics, set the environment variable SCOREP_METRIC_RUSAGE_SEP to the desired character.

Example:

```
SCOREP_METRIC_RUSAGE_SEP=:
```

This indicates that counter names in the list are separated by colons.

**Note**

In addition it is possible to specify metrics that will be recorded only by the initial thread of a process. Please use SCOREP_METRIC_RUSAGE_PER_PROCESS for that reason.

### 5.6.3 Recording Linux Perf Metrics

This metric source uses the Linux Perf Interface to access hardware performance counters. First it is explained how to specify PERF metrics that will be recorded by every location.

You can enable the recording of PERF performance metrics by setting the environment variable SCOREP_METRIC_PERF to a comma-separated list of metric names. Metric names can be any PERF preset names or PAPI native counter names.

Example:

```
SCOREP_METRIC_PERF=cycles,page-faults,LLC-load-misses
```

In this example the number of CPU cycles, the number of page faults, and Last Level Cache Load Misses will be recorded. If any of the requested metrics is not recognized program execution will be aborted with an error message. The user can leave the environment variable unset to indicate that no metrics are requested. Use the tool `perf list` to get a list of available PERF events.

If you want to change the separator used in the list of PERF metrics, set the environment variable SCOREP_METRIC_PERF_SEP to the desired character.

Example:

```
SCOREP_METRIC_PERF_SEP=:
```

This indicates that counter names in the list are separated by colons.

**Note**

In addition it is possible to specify metrics that will be recorded per-process. Please use SCOREP_METRIC_PERF_PER_PROCESS for that reason.

### 5.6.4 Metric Plugins

Metric plugins extend the functionality of Score-P by providing additional counters as external libraries. The libraries are loaded when tracing or profiling your application. So there is no need to recompile your application or instrument it manually.

A simple example of a synchronous metric plugin can be found in Appendix C. Every plugin needs to include SCOREP_MetricPlugins.h. The commands to build the corresponding library of this plugin might look like:

```
gcc -c -fPIC hello_world.c   \
    -o libHelloWorld_plugin.so.o `scorep-config --cppflags`
gcc -shared -Wl,-soname,libHelloWorld_plugin.so  \
    -o libHelloWorld.so libHelloWorld_plugin.so.o
```

| Token | Module |
|---|---|
| ALL | Activate all available modules |
| DEFAULT | Activate the configured default modules of CG, COLL, ENV, IO, P2P, RMA, TOPO, XNONBLOCK. This can be used to easily activate additional modules. |
| CG | Communicators and groups |
| COLL | Collective communication |
| ENV | Environmental management |
| ERR | Error handlers |
| EXT | External interfaces |
| IO | I/O |
| MISC | Miscellaneous |
| P2P | Point-to-point communication |
| RMA | One-sided communication |
| SPAWN | Process management interface (aka Spawn) |
| TOPO | Topology communicators |
| TYPE | MPI Datatypes |
| XNONBLOCK | Extended non-blocking communication events |
| XREQTEST | Test events for tests of uncompleted requests |

To enable a metric plugin, add the plugin <PLUGINNAME> to the environment variable SCOREP_METRIC_PLUGINS and configure the used metrics through the environment variable SCOREP_METRIC_PLUGINNAME. In the following example we want to use the above `HelloWorld` plugin. We select two counters `metric1` and `metric2` from the plugin. Make sure that the metric plugin library is placed in a directory which is part of `LD_LIBRARY_PATH`.

```
SCOREP_METRIC_PLUGINS=HelloWorld_plugin
SCOREP_METRIC_HELLOWORLD_PLUGIN=metric1,metric2
```

**Note**

Plugins are not supposed to trigger events (e.g. via MPI, OpenMP, Pthreads or user instrumentation) during initialization and finalization of the plugin.
A set of open source metric plugins is available at GitHub.

## 5.7 MPI Performance Measurement

The Message Passing Interface (MPI) adapter of Score-P supports the tracing of most of MPI's 300+ function calls. MPI defines a so-called 'profiling interface' that supports the provision of wrapper libraries that can easily interposed between the user application and the MPI library calls.

### 5.7.1 Selection of MPI Groups

The general Score-P filtering mechanism is not applied to MPI functions. Instead, the user can decide whether event generation is turned on or off for a group of MPI functions, at start time of the application. These groups are the listed sub-modules of this adapter. Each module has a short string token that identifies this group. To activate event generation for a specific group, the user can specify a comma-separated list of tokens in the configuration variable SCOREP_MPI_ENABLE_GROUPS. Additionally, special tokens exist to ease the handling by the user. A complete list of available tokens that can be specified in the runtime configuration is listed below.

**Note**

Event generation in this context only relates to flow and transfer events. Tracking of communicators, groups, and other internal data is unaffected and always turned on.

Example:

```
SCOREP_MPI_ENABLE_GROUPS=ENV,P2P
```

This will enable event generation for environmental management, including `MPI_Init` and `MPI_Finalize`, as well as point-to-point communication, but will disable it for all other functions groups.

A shorthand to get event generation for all supported function calls is

```
SCOREP_MPI_ENABLE_GROUPS=ALL
```

A shorthand to add a single group, e.g. `TYPE`, to the configured default is

```
SCOREP_MPI_ENABLE_GROUPS=DEFAULT,TYPE
```

A detailed overview of the MPI functions associated with each group can be found in Appendix B.

A somehow special role plays the `XNONBLOCK` flag. This flag determines what kind of events are generated by non-blocking peer-to-peer MPI function calls. If `XNONBLOCK` is not set, an OTF2_MPI_Send event is created at the non-blocking send call and an OTF2_MPI_Recv event is recorded when a non-blocking receive request has completed. If `XNONBLOCK` is set, an OTF2_Isend event is recorded at the non-blocking send and an OTF2_IsendComplete event when the event was completed. Furthermore, on a non-blocking receive, it records an OTF2_IRecvRequest event. On request completion an OTF2_IRecv event is recorded. In both cases the group `P2P` must be enabled. Otherwise Score-P records no events for peer-to-peer communication functions.

### 5.7.2 Recording MPI Communicator Names

The measurement system tracks also the names of MPI communicators to easily identify them later in the analysis. This is done via the `MPI_Comm_set_name` call. But there are some restrictions. First, the name of a communicator is only recorded at the first call to `MPI_Comm_set_name` for this communicator. Later calls are ignored. Also this call is only honored when the call was made from the rank which is rank 0 in this communicator. Other calls from other ranks are ignored. And lastly the name will also be not recorded if the communicator has only one member.

## 5.8 CUDA Performance Measurement

If Score-P has been built with CUDA support it is capable of recording CUDA API function calls and GPU activities. The measurement is based on NVIDIA's **CU**DA **P**rofiling and **T**ool **I**nterface (CUPTI), which is an integral part of the CUDA Toolkit since version 4.1.

Score-P can wrap the NVIDIA compiler (**`scorep nvcc`**) to instrument .cu files. If Score-P has been built with the Intel compiler an additional flag has to be added for instrumentation:

```
-compiler-bindir=<path-to-intel-compiler-command>
```

Otherwise the program will not be instrumented, as `nvcc` uses the GNU compiler by default.

Setting the environment variable `SCOREP_CUDA_ENABLE` to **`yes`** enables CUDA measurement. Please refer to the description of this variable to enable a particular composition of CUDA measurement features.

CUPTI uses an extra buffer to store its activity records. If the size of this buffer is too small, Score-P will print a warning about the current buffer size and the number of dropped records. To avoid dropping of records increase the buffer size via the environment variable `SCOREP_CUDA_BUFFER` (default: 1M).

Since CUDA toolkit version 5.5 the chunk size for the CUPTI activity buffer can be specified via the environment variable `SCOREP_CUDA_BUFFER_CHUNK` (default: 8k). Buffer chunks are allocated whenever CUPTI requests a buffer (e.g. to record activities on a CUDA stream). `SCOREP_CUDA_BUFFER` specifies the upper bound of memory to be allocated for CUPTI activities. Therefore it should be a multiple of `SCOREP_CUDA_BUFFER_CHUNK`.

**Note**

Make sure to call `cudaDeviceReset()` or `cudaDeviceSynchronize()` before the exit of the program. Otherwise GPU activities might be missing in the trace.
For CUDA 5.5 there is an error in CUPTI buffer handling. The last activity in a CUPTI activity buffer (SCOREP_CUDA_BUFFER_CHUNK) gets lost, when the buffer is full. To avoid this issue specify SCOREP_CUDA_BUFFER_CHUNK as large as necessary to store all CUDA device activities until the CUDA device is synchronized with the host. In CUDA 6.0 this issue is fixed and CUPTI does not request buffers for individual streams any more.
Score-P supports CUDA monitoring since CUDA toolkit version 4.1. Make sure that the Score-P installation has configured CUDA support. The configure summary should contain the line:

```
CUDA support: yes
```

If not, for most systems it is sufficient to specify the CUDA toolkit directory at Score-P configuration time:

```
--with-libcudart=<path-to-cuda-toolkit-directory>
```

Otherwise check the configure help output to specify the location of the CUDA toolkit and CUPTI libraries and include files:

```
../configure --help=recursive | grep -E "(cuda|cupti)"
```

CUDA device and host activities can be filtered by name at runtime using the Score-P filter file (see Section 5.3). Filtering does not remove CUDA activities inserted by Score-P or CUDA data transfers inserted as RDMA events. If a kernel is filtered, no kernel launch properties activated in SCOREP_CUDA_ENABLE using `kernel_counter` are inserted for this kernel. GPU idle time is not affected by kernel filtering.

## 5.9 OpenCL Performance Measurement

If Score-P has been built with OpenCL support it is capable of recording OpenCL API function calls.

Setting the environment variable SCOREP_OPENCL_ENABLE to **yes** enables OpenCL measurement. Please refer to the description of this variable to enable a particular composition of OpenCL measurement features.

OpenCL measurement uses an extra buffer to store its activity records. If the size of this buffer is too small, Score-P will print a warning about the current buffer size and the number of dropped records. To avoid dropping of records increase the buffer size via the environment variable `SCOREP_OPENCL_BUFFER` (default: 1M). Memory in bytes for the OpenCL command queue buffer can be adjusted by setting the environment variable SCOREP_OPENCL_BUFFER_QUEUE (default: 8k).

## 5.10 OpenACC Performance Measurement

If Score-P has been built with OpenACC support it is capable of recording OpenACC regions as well as activities such as enqueuing kernels, data uploads, and data downloads. OpenACC activities that are implicitly generated by the compiler are attributed with `acc_implicit`.

To enable OpenACC measurement in Score-P the user has to:

- Build and install shared libraries of Score-P (use `--enable-shared` option when configuring Score-P).
- Set the environment variable `ACC_PROFLIB` to specify the OpenACC profiling library.
  Example:

  ```
  export ACC_PROFLIB=<path_to_scorep_installation>/lib/libscorep_adapter_openacc_event.so
  ```

- Set the environment variable SCOREP_OPENACC_ENABLE to **yes**. Please refer to the description of this variable to enable a particular composition of OpenACC measurement features.

**Note**

Score-P supports OpenACC monitoring, if the respective OpenACC compiler implements the OpenACC profiling interface that is part of the OpenACC standard since version 2.5. Make sure that the Score-P installation has configured OpenACC support. The configure summary should contain the line:

```
OpenACC support: yes
```

## 5.11 Online Access Interface

Online Access (OA) is an interface to the measurement system of Score-P allowing online analysis capable tools to configure and retrieve profile measurements remotely over sockets.

The Online Access interface implements a client-server paradigm, where Score-P acts as a server accepting connections from the remote tool. During the initialization, the OA module of the Score-P creates one socket for each application process. The network addresses and the ports of these sockets are published at the registry service and could be later queried by the remote tool. The hostname and the port of the registry service should be specified via the SCOREP_ONLINEACCESS_REG_HOST and SCOREP_ONLINEACCESS_REG_PORT environment variables, respectively. After publishing the socket addresses and ports, the OA module will accept connections. Once the connection is established the OA module will suspend the application execution and wait for requests. The format of the requests is plain text following the syntax below:

```
<request>             = <metric_configuration> | <execution> | <retrieval>
<metric_configuration> = BEGINREQUESTS GLOBAL <request_type>
                         ENDREQUESTS
<request_type>        = MPI | EXECUTION_TIME |
                         METRIC <metric_specification>
<metric_specification> = PERISCOPE <periscope_metric_code> |
                         PAPI "<papi_counter_name>" |
                         RUSAGE "<rusage_metric_name>" |
                         OTHER "metric_name"
<execution>           = TERMINATE | RUNTOSTART | RUNTOEND
<retrieval>           = GETSUMMARYDATA
```

where

- BEGINREQUESTS indicates the beginning of the request list,
- ENDREQUESTS indicates the end of the request list,
- GLOBAL indicates that the following measurement request is applied to all locations,
- MPI requests mpi wait states analysis,
- EXECUTION_TIME requests execution time,
- METRIC indicates the begin of the metric request,
- PERISCOPE <periscope_metric_code> requests a metric by the Periscope internal code,
- PAPI <papi_counter_name> requests a PAPI hardware counter metric by the counter name,
- RUSAGE <rusage_counter_name> requests a Resource Usage Counter metric by the counter name,
- OTHER <metric_name> requests a metric, to be defined in Score-P definition system, specified by the name,
- TERMINATE requests termination of the application,
- RUNTOSTART requests Score-P to run the beginning of the OA phase,
- RUNTOEND requests Score-P to run the end of the OA phase,
- GETSUMMARYDATA requests retrieval of the profile data.

When the GETSUMMARY request is received, the OA module will transform the call-path profile into a flat profile and send the data back to the remote tool. The flat profile is sent in two parts, where the first part carries the region definition data and the second part carries profile measurements. Each part starts with the key word MERGED_R↩EGION_DEFINITIONS or FLAT_PROFILE and followed by the number of the entries and the buffer containing the data.

# Chapter 6

# Usage of scorep-score

scorep-score is a tool that allows to estimate the size of an OTF2 trace from a CUBE4 profile. Furthermore, the effects of filters are estimated. The main goal is to define appropriate filters for a tracing run from a profile.

The general work-flow for performance analysis with Score-P is:

1. Instrument an application (see Section 3).
2. Perform a measurement run and record a profile (see Section 5). The profile already gives an overview what may happen inside the application.
3. Use scorep-score to define an appropriate filter for an application Otherwise the trace file may become too large. This step is explained in this Chapter.
4. Perform a measurement run with tracing enabled and the filter applied (see Section 5.2 and Section 5.3).
5. Perform in-depth analysis on the trace data.

## 6.1 Basic usage

To invoke `scorep-score` you must provide the filename of a CUBE4 profile as argument. Thus, the basic command looks like this:

```
scorep-score profile.cubex
```

The output of the command may look like this (taking an MPI/OpenMP hybrid application as an example):

```
Estimated aggregate size of event trace:                   20MB
Estimated requirements for largest trace buffer (max_buf): 20MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       24MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=24MB 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      19,377,048       786,577        27.48    100.0           34.93  ALL
     USR      16,039,680       668,320         0.36      1.3            0.53  USR
     OMP       3,328,344       117,881        26.92     98.0          228.37  OMP
     COM           9,024           376         0.20      0.7          532.17  COM
```

The first line of the output gives an estimation of the total size of the trace, aggregated over all processes. This information is useful for estimating the space required on disk. In the given example, the estimated total size of the event trace is 20MB.

The second line prints an estimation of the memory space required by a single process for the trace. The memory space that Score-P reserves on each process at application start must be large enough to hold the process' trace

in memory in order to avoid flushes during runtime, because flushes heavily disturb measurements. In addition to the trace, Score-P requires some additional memory to maintain internal data structures. Thus, it provides also an estimation for the total amount of required memory on each process. The memory size per process that Score-P reserves is set via the environment variable SCOREP_TOTAL_MEMORY. In the given example the per process memory should be larger than 24MB.

Beginning with the 6th line, `scorep-score` prints a table that show how the trace memory requirements and the runtime is distributed among certain function groups. The column `max_tbc` shows how much trace buffer is needed on a single process. The column `time(s)` shows how much execution time was spend in regions of that group in seconds, the column `%` shows the fraction of the overall runtime that was used by this group, and the column `time/visit(us)` shows the average time per visit in microseconds.

The following groups exist:

- ALL: Includes all functions of the application
- OMP: This group contains all regions that represent an OpenMP construct
- MPI: This group contains all MPI functions
- SHMEM: This group contains all SHMEM functions
- PTHREAD: This group contains all Pthread functions
- CUDA: This group contains all CUDA API functions and kernels
- OPENCL: This group contains all OpenCL API functions and kernels
- OPENACC: This group contains all OpenACC API functions and kernels
- MEMORY: This group contains all libc and C++ memory (de)allocation funcitons
- COM: This group contains all functions, implemented by the user that appear on a call-path to any functions from the above groups, except ALL.
- USR: This group contains all user functions except those in the COM group.

## 6.2 Additional per-region information

For a more detailed output, which shows the data for every region, you can use the `-r` option. The command could look like this.

```
scorep-score profile.cubex -r
```

This command adds information about the used buffer sizes and execution time of every region to the table. The additional lines of the output may look like this:

```
flt type      max_buf[B]        visits      time[s]  time[%]  time/visit[us]  region

     COM              24             4         0.00      0.0           67.78  Init
     COM              24             4         0.00      0.0           81.20  main
     USR              24             4         0.12      2.0        30931.14  InitializeMatrix
     COM              24             4         0.05      0.8        12604.78  CheckError
     USR              24             4         0.00      0.0           23.76  PrintResults
     COM              24             4         0.01      0.2         3441.83  Finish
     COM              24             4         0.48      7.7       120338.17  Jacobi
```

The region name is displayed in the column named `region`. The column `type` shows to which group this region belongs. In the example above the function main belongs to group `COM` required 24 bytes per process and used 0 s execution time. The regions are sorted by their buffer requirements.

By default scorep-score uses demangled function names. However, if you want to map data to tools which use mangled names you might want to display mangled names. Furthermore, if you have trouble with function signatures that contain characters that also have a wildcard meaning, defining filters on mangled names might be easier. To display mangled names instead of demangled names, you can use the -m flag, e.g.

```
scorep-score profile.cubex -r -m
```

**Note**

The -m flag takes only effect if you display region names. In particular it means that the -m flag is only effective if also the -r is specified.
In some cases, the same name is shown for the mangled and the demangled name. Some instrumentation methods, e.g. user instrumentation, provide only a demangled name. For C-compilers mangled and demangled names are usually identical. Or the demangling might have failed and only a mangled name is available. In these cases we show always the one name that is available.

## 6.3 Defining and testing a filter

For defining a filter, it is recommended to exclude short frequently called functions from measurement, because they require a lot of buffer space (represented by a high value under `max_tbc`) but incur a high measurement overhead. Furthermore, for communication analysis, functions that appear on a call-path to MPI functions and OpenMP constructs (regions of type `COM`) are usually of more interest than user functions of type `USR` which do not appear on call-path to communications. MPI functions and OpenMP constructs cannot be filtered. Thus, it is usually a good approach to exclude regions of type `USR` starting at the top of the list until you reduced the trace to your needs. Section 5.3 describes the format of a filter specification file.

If you have a filter file, you can test the effect of your filter on the trace file. Therefor, you need to pass a `-f` followed by the file name of your filter. E.g. if your filter file name is `myfilter`, the command looks like this:

```
scorep-score profile.cubex -f myfilter
```

An example output is:

```
Estimated aggregate size of event trace:                   7kB
Estimated requirements for largest trace buffer (max_buf): 1806 bytes
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       5MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=5MB 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           2,093           172         5.17    100.0        30066.64  ALL
 -   MPI           1,805           124         4.20     81.3        33910.31  MPI
 -   COM             240            40         0.84     16.3        21092.44  COM
 -   USR              48             8         0.12      2.4        15360.71  USR

 *   ALL           1,805           124         4.20     81.3        33910.31  ALL-FLT
 -   MPI           1,805           124         4.20     81.3        33910.31  MPI-FLT
 +   FLT             288            48         0.97     18.7        20137.15  FLT
```

Now, the output estimates the total trace size an the required memory per process, if you would apply the provided filter for the measurement run which records the trace. A new group `FLT` appears, which contains all regions that are filtered. Under `max_tbc` the group `FLT` displays how the memory requirements per process are reduced. Furthermore, the groups that end on `-FLT`, like `ALL-FLT` contain only the unfiltered regions of the original group. E.g. `USR-FLT` contains all regions of group `USR` that are not filtered.

Furthermore, the column `flt` is no longer empty but contain a symbol that indicates how this group is affected by the filter. A '-' means 'not filtered', a '+' means 'filtered' and a '*' appears in front of groups that potentially can be affected by the filter.

You may combine the `-f` option with a `-r` option. In this case, for each function a '+' or '-' indicates whether the function is filtered.

## 6.4 Calculating the effects of recording hardware counters

Recording additional metrics, e.g. hardware counters may significantly increase the trace size, because for many events additional metric values are stored. In order to estimate the effects of these metrics, you may add a `-c` followed by the number of metrics you want to record. E.g.

```
scorep-score profile.cubex -c 3
```

would mean that `scorep-score` estimates the disk and memory requirements for the case that you record 3 additional metrics.

# Chapter 7

# Performance Analysis Workflow Using Score-P

This chapter demonstrates a typical performance analysis workflow using Score-P. It consist of the following steps:

1. Program instrumentation (Section 7.1)
2. Summary measurement collection (Section 7.2)
3. Summary report examination (Section 7.3)
4. Summary experiment scoring (Section 7.4)
5. Advanced summary measurement collection (Section 7.5)
6. Advanced summary report examination (Section 7.6)
7. Event trace collection and examination (Section 7.7)

The workflow is demonstrated using NPB BT-MZ benchmark as an example. BT-MZ solves a discretized version of unsteady, compressible Navier-Stokes equations in three spatial dimensions. It performs 200 time-steps on a regular 3-dimensional grid using ADI and verifies solution error within acceptable limit. It uses intra-zone computation with OpenMP and inter-zone communication with MPI. The benchmark can be build with a predefined data class (S,W,A,B,C,D,E,F) and any number of MPI processes and OpenMP threads.

NPB BT-MZ distribution already prepared for this example could be obtained from here.

## 7.1 Program Instrumentation

In order to collect performance measurements, BT-MZ has to be instrumented. There are various types of instrumentation supported by Score-P which cover a broad spectrum of performance analysis use cases (see Chapter 3 for more details).

In this example we start with automatic compiler instrumentation by prepending compiler/linker specification variable `MPIF77` found in `config/make.def` with `scorep`:

```
#            SITE- AND/OR PLATFORM-SPECIFIC DEFINITIONS
#---------------------------------------------------------------------
# Items in this file may need to be changed for each platform.
#---------------------------------------------------------------------
...
#---------------------------------------------------------------------
# The Fortran compiler used for MPI programs
#---------------------------------------------------------------------
#MPIF77 = mpif77
# Alternative variants to perform instrumentation
...
MPIF77 = scorep mpif77
# This links MPI Fortran programs; usually the same as ${MPIF77}
FLINK   = $(MPIF77)
...
```

After the makefile is modified and saved, it is recommended to return to the root folder of the application and clean-up previously build files:

```
% make clean
```

Now the application is ready to be instrumented by simply issuing the standard build command:

```
% make bt-mz CLASS=W NPROCS=4
```

After the command is issued, the make command should produce the output similar to the one below:

```
cd BT-MZ; make CLASS=W NPROCS=4 VERSION=
make: Entering directory 'BT-MZ'
cd ../sys; cc  -o setparams setparams.c -lm
../sys/setparams bt-mz 4 W
scorep mpif77 -c  -O3 -fopenmp bt.f
[...]
cd ../common;  scorep --user mpif77 -c  -O3 -fopenmp timers.f
scorep mpif77 -O3 -fopenmp -o ../bin.scorep/bt-mz_W.4 \
bt.o initialize.o exact_solution.o exact_rhs.o set_constants.o \
adi.o rhs.o zone_setup.o x_solve.o y_solve.o exch_qbc.o \
solve_subs.o z_solve.o add.o error.o verify.o mpi_setup.o \
../common/print_results.o ../common/timers.o
Built executable ../bin.scorep/bt-mz_W.4
make: Leaving directory 'BT-MZ'
```

When make finishes, the built and instrumented application could be found under `bin.scorep/bt-mz_W.4`.

## 7.2 Summary Measurement Collection

Now instrumented BT-MZ is ready to be executed and to be profiled by Score-P at the same time. However before doing so, one has an opportunity to configure Score-P measurement by setting Score-P environment variables. For the complete list of variables please refer to Appendix E.

The typical Score-P performance analysis workflow implies collecting summary performance information first and then in detail performance exploration using execution traces. Therefore Score-P has to be configured to perform profiling and tracing has to be disabled. This is done by setting corresponding environment variables:

```
% export SCOREP_ENABLE_PROFILING=1
% export SCOREP_ENABLE_TRACING=0
```

Performance data collected by Score-P will be stored in an experiment directory. In order to efficiently manage multiple experiments, one can specify a meaningful name for the experiment directory by setting

```
% export SCOREP_EXPERIMENT_DIRECTORY=scorep_bt-mz_W_4x4_sum
```

After Score-P is prepared for summary collection, the instrumented application can be started as usual:

```
% cd bin.scorep
% export OM_NUM_THREADS=4
% mpiexec -np 4 ./bt-mz_W.4
```

The BT-MZ output should look similar to the listing below:

```
NAS Parallel Benchmarks (NPB3.3-MZ-MPI) BT-MZ MPI+OpenMP Benchmark

Number of zones:   4 x   4
Iterations: 200    dt:   0.000800
Number of active processes:     4

Use the default load factors with threads
Total number of threads:     16  (  4.0 threads/process)

Calculated speedup =     15.78

Time step    1
[... More application output ...]
```

After application execution is finished, the summary performance data collected by Score-P is stored in the experiment directory `bin.scorep/scorep_bt-mz_W_4x4_sum`. The directory contains the following files:

- `scorep.cfg` - a record of the measurement configuration used in the run
- `profile.cubex` - the analysis report that was collated after measurement

## 7.3 Summary report examination

After BT-MZ finishes execution, the summary performance data measured by Score-P can be investigated with CUBE or ParaProf interactive report exploration tools.

CUBE:

```
% cube scorep_bt-mz_W_4x4_sum/profile.cubex
```

ParaProf:

```
% paraprof scorep_bt-mz_W_4x4_sum/profile.cubex
```

Both tools will reveal the call-path profile of BT-MZ annotated with metrics: *Time*, *Visits count*, MPI message statistics (bytes sent/received). For more information on using the tool please refer to the corresponding documentation (CUBE, ParaProf).

## 7.4 Summary experiment scoring

Though we were able to collect the profile data, one can mention that the execution took longer in comparison to uninstrumented run, even when the time spent for measurement start-up/finalization is disregarded. Longer execution times of the instrumented application is a sign of high instrumentation/measurement overhead. Furthermore, this overhead might result in large trace files and buffer flushes in the later tracing experiment if Score-P is not properly configured.

In order to investigate sources of the overhead and to tune measurement configuration for consequent trace collection with Score-P, `scorep-score` tool (see Section 6 for more details about `scorep-score`) can be used:

```
% scorep-score scorep_bt-mz_W_4x4_sum/profile.cubex
Estimated aggregate size of event trace (total_tbc):
                  990247448 bytes
Estimated requirements for largest trace buffer (max_tbc):
                  256229936 bytes
(hint: When tracing set SCOREP_TOTAL_MEMORY > max_tbc to avoid
intermediate flushes
or reduce requirements using file listing names of USR regions
to be filtered.)
flt type         max_tbc         time      % region
     ALL       256229936      5549.78  100.0 ALL
     USR       253654608      1758.27   31.7 USR
     OMP         5853120      3508.57   63.2 OMP
     COM          343344       183.09    3.3 COM
     MPI           93776        99.86    1.8 MPI
```

The textual output of the tool generates an estimation of the size of an OTF2 trace produced, should Score-P be run using the current configuration. Here the trace size estimation could be also seen as a measure of overhead, since both are proportional to the number of recorded events. Additionally, the tool shows the distribution of the required trace size over call-path classes. From the report above one can see that the estimated trace size needed is equal to 1 GB in total or 256 MB per MPI rank, which is significant. From the breakdown per call-path class one can see that most of the overhead is due to user-level computations. In order to further localize the source of the overhead, `scorep-score` can print the breakdown of the buffer size on per-region basis:

```
% scorep-score -r scorep_bt-mz_W_4x4_sum/profile.cubex
  [...]
flt  type    max_tbc       time      % region
     ALL   256229936    5549.78  100.0 ALL
     USR   253654608    1758.27   31.7 USR
     OMP     5853120    3508.57   63.2 OMP
     COM      343344     183.09    3.3 COM
     MPI       93776      99.86    1.8 MPI
     USR    79176312     559.15   31.8 binvcrhs_
     USR    79176312     532.73   30.3 matvec_sub_
     USR    79176312     532.18   30.3 matmul_sub_
     USR     7361424      50.51    2.9 binvrhs_
     USR     7361424      56.35    3.2 lhsinit_
     USR     3206688      27.32    1.6 exact_solution_
     OMP     1550400    1752.20   99.7 !$omp implicit barrier
     OMP      257280       0.44    0.0 !$omp parallel @exch_qbc.f
     OMP      257280       0.61    0.0 !$omp parallel @exch_qbc.f
     OMP      257280       0.48    0.0 !$omp parallel @exch_qbc.f
```

The regions marked as USR type contribute to around 32% of the total time, however, much of that is very likely to be measurement overhead due to frequently-executed small routines. Therefore, it is highly recommended to remove these routines from measurements. This can be achieved by placing them into a filter file (please refer to Section 6.3 for more details about filter file specification) as regions to be excluded from measurements. There is already a filter file prepared for BT-MZ which can be used:

```
% cat ../config/scorep.filt
SCOREP_REGION_NAMES_BEGIN EXCLUDE
binvcrhs*
matmul_sub*
matvec_sub*
exact_solution*
binvrhs*
lhs*init*
timer_*
```

One can use `scorep-score` once again to verify the effect of the filter file :

```
% scorep-score -f ../config/scorep.filt scorep_bt-mz_W_4x4_sum
Estimated aggregate size of event trace (total_tbc):
                    20210360 bytes
Estimated requirements for largest trace buffer (max_tbc):
                    6290888 bytes
(hint: When tracing set SCOREP_TOTAL_MEMORY > max_tbc to avoid
intermediate flushes
or reduce requirements using file listing names of USR regions
to be filtered.)
```

Now one can see that the trace size is reduced to just 20MB in total or 6MB per MPI rank. The regions filtered out will be marked with "+" in the left-most column of the per-region report.

## 7.5 Advanced summary measurement collection

After the filtering file is prepared to exclude the sources of the overhead, it is recommended to repeat summary collection, in order to obtain more precise measurements.

In order to specify the filter file to be used during measurements, the corresponding environment variable has to be set:

```
% export SCOREP_FILTERING_FILE=../config/scorep.filt
```

It is also recommended to adjust the experiment directory name for the new run:

```
% export SCOREP_EXPERIMENT_DIRECTORY=\
         scorep_bt-mz_W_4x4_sum_with_filter
```

Score-P also has a possibility to record hardware counters (see Section 5.6.1) and operating system resource usage (see Section 5.6.2) in addition to default time and number of visits metrics. Hardware counters could be configured by setting Score-P environment variable `SCOREP_METRIC_PAPI` to the comma-separated list of P↩ API events (other separator could be specified by setting `SCOREP_METRIC_PAPI_SEP`):

```
% export SCOREP_METRIC_PAPI=PAPI_TOT_INS,PAPI_FP_INS
```

**Note**

The specified combination of the hardware events has to be valid, otherwise Score-P will abort execution. Please run `papi_avail` and `papi_native_avail` in order to get the list of the available PAPI generic and native events.

Operating system resource usage metrics are configured by setting the following variable:

```
% export SCOREP_METRIC_RUSAGE=ru_maxrss,ru_stime
```

Additionally Score-P can be configured to record only a subset of the mpi functions. This is achieved by setting `SCOREP_MPI_ENABLE_GROUPS` variable with a comma-separated list of sub-groups of MPI functions to be recorded (see Appendix B):

```
% export SCOREP_MPI_ENABLE_GROUPS=cg,coll,p2p,xnonblock
```

In case performance of the CUDA code is of interest, Score-P can be configured to measure CUPTI metrics as follows (see Section 5.8):

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

In case performance of the OpenCL code is of interest, Score-P can be configured to measure OpenCL events as follows (see Section 5.9):

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

When the granularity offered by the automatic compiler instrumentation is not sufficient, one can use Score-P manual user instrumentation API (see Section 3.2) for more fine-grained annotation of particular code segments. For example initialization code, solver or any other code segment of interest can be annotated.

In order to enable user instrumentation, an `-user` argument has to be passed to Score-P command prepending compiler and linker specification:

```
% MPIF77 = scorep --user mpif77
```

Below, the loop found on line ... in file ... is annotated as a user region:

```
#include "scorep/SCOREP_User.inc"
subroutine foo(...)
  ! Declarations
  SCOREP_USER_REGION_DEFINE( solve )
  ! Some code...
  SCOREP_USER_REGION_BEGIN( solve, "<solver>", \
                            SCOREP_USER_REGION_TYPE_LOOP )
  do i=1,100
    [...]
  end do
  SCOREP_USER_REGION_END( solve )
  ! Some more code...
end subroutine
```

This will appear as an additional region in the report.

BT-MZ has to be recompiled and relinked in order to complete instrumentation.

```
% make clean
% make bt-mz CLASS=W NPROCS=4
```

After applying advanced configurations described above, summary collection with Score-P can be started as usual:

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

## 7.6 Advanced summary report examination

After execution is finished, one can use `scorep-score` tool to verify the effect of filtering:

```
% scorep-score scorep_bt-mz_W_4x4_sum_with_filter/profile.cubex
Estimated aggregate size of event trace (total_tbc):
                    20210360 bytes
Estimated requirements for largest trace buffer (max_tbc):
                    6290888 bytes
(hint: When tracing set SCOREP_TOTAL_MEMORY > max_tbc to avoid
intermediate flushes
or reduce requirements using file listing names of USR regions
to be filtered.)
flt type         max_tbc         time      % region
     ALL         6290888       241.77  100.0 ALL
     OMP         5853120       168.94   69.9 OMP
     COM          343344        35.57   14.7 COM
     MPI           93776        37.25   15.4 MPI
     USR             672         0.01    0.0 USR
```

The report above shows significant reduction in runtime (due to elimination of the overhead) not only in USR regions but also in MPI/OMP regions as well.

Now, the extended summary report can be interactively explored using CUBE:

```
% cube scorep_bt-mz_W_4x4_sum_with_filter/profile.cubex
```

or ParaProf:

```
% paraprof scorep_bt-mz_W_4x4_sum_with_filter/profile.cubex
```

## 7.7 Event trace collection and examination

After exploring extended summary report, additional insight into performance of BT-MZ can be gained by performing trace collection. In order to do so, Score-P has to be configured to perform tracing by setting corresponding variable to `true` and disabling profile generation:

```
% export SCOREP_ENABLE_TRACING=true
% export SCOREP_ENABLE_PROFILING=false
```

Additionally it is recommended to set a meaningful experiment directory name:

```
% export SCOREP_EXPERIMENT_DIRECTORY=scorep_bt-mz_W_4x4_trace
```

After BT-MZ execution is finished, a separate trace file per thread is written into the new experiment directory. In order to explore it, `Vampir` tool can be used:

```
% vampir scorep_bt-mz_W_4x4_trace/traces.otf2
```

Please consider that traces can become extremely large and unwieldy, because the size of the trace is proportional to number of processes/threads (width), duration (length) and detail (depth) of measurement. When the trace is too large to hold in the memory allocated by Score-P, flushes can happen. Unfortunately the resulting traces are of little value, since uncoordinated flushes result in cascades of distortion.

Traces should be written to a parallel file system, e.g., to `/work` or `/scratch` which are typically provided for this purpose.

# Appendices

# Appendix A

# Score-P INSTALL



```
                        Score-P INSTALL GUIDE
                        =====================

This file describes how to configure, compile, and install the Score-P
measurement infrastructure.  If you are not familiar with using the
configure scripts generated by GNU autoconf, read the "Generic
Installation Instructions" section below; then return here.  Also,
make sure to carefully read and follow the platform-specific
installation notes (especially when building for the Intel Xeon Phi
platform).


Quick start
===========

In a nutshell, configuring, building, and installing Score-P can be as
simple as executing the shell commands

    mkdir _build
    cd _build
    ../configure --prefix=<installdir>
    make
    make install
```

```
If you don't specify --prefix, /opt/scorep will be used.

Depending on your system configuration and specific needs, the build
process can be customized as described below.

Note Score-P requires a case sensitive file system to build correctly.


Configuration
=============

The configure script in this package tries to automatically determine
the platform for which Score-P will be compiled in order to provide
reasonable defaults for backend (i.e., compute-node) compilers,
MPI compilers, and, in case of cross-compiling environments, frontend
(i.e., login-node) compilers.

Depending on the environment it is possible to override the platform
defaults by using the following configure options:

  --with-machine-name=<default machine name>
              The default machine name used in profile and trace
              output. We suggest using a unique name, e.g., the
              fully qualified domain name. If not set, a name
              based on the detected platform is used. Can be
              overridden at measurement time by setting the
              environment variable SCOREP_MACHINE_NAME.

Score-P requires a full compiler suite with language support for C99,
C++98, Fortran 77, and Fortran 90. The following section describes how
to select supported compiler suits.

In non-cross-compiling environments, the compiler suite used to build
the backend parts can be specified explicitly if desired. On Linux
clusters it is currently recommended to use this option to select a
compiler suite other than GCC.

  --with-nocross-compiler-suite=(gcc|ibm|intel|pgi|studio)
              The compiler suite used to build this package in
              non-cross-compiling environments. Needs to be in $PATH.
              [Default: gcc]

In cross-compiling environments, the compiler suite used to build the
frontend parts can be specified explicitly if desired.

  --with-frontend-compiler-suite=(gcc|ibm|intel|pgi|studio)
              The compiler suite used to build the frontend parts of
              this package in cross-compiling environments.  Needs to
              be in $PATH.
              [Default: gcc]

The MPI compiler, if in $PATH, is usually autodetected. If there are
several MPI compilers in $PATH the user is requested to select one
using the configure option:

  --with-mpi=(bullxmpi|hp|ibmpoe|intel|intel2|intel3|intelpoe|lam| \
             mpibull2|mpich|mpich2|mpich3|openmpi|platform|scali| \
             sgimpt|sun)
              The MPI compiler suite to build this package in non
              cross-compiling mode. Usually autodetected. Needs to be
              in $PATH.

Note that there is currently no consistency check if backend and MPI
compiler are from the same vendor. If they are not, linking problems
(undefined references) might occur.

The SHMEM compiler, if in $PATH, is usually autodetected. If there are
several SHMEM compilers in $PATH the user is requested to select one
using the configure option:

  --with-shmem=(openshmem|openmpi|sgimpt)
              The SHMEM compiler suite to build this package in
```

```
              non cross-compiling mode. Usually autodetected.
              Needs to be in $PATH.

If a particular system requires to use compilers different to those
Score-P currently supports, please edit the three files
vendor/common/build-config/platforms/platform-*-user-provided to your
needs and use the following configure option:

  --with-custom-compilers
              Customize compiler settings by 1. copying the three
              files
              <srcdir>/vendor/common/build-config/platforms/platform-*-user-provided
              to the directory where you run configure <builddir>,
              2. editing those files to your needs, and 3. running
              configure. Alternatively, edit the files under <srcdir>
              directly. Files in <builddir> take precedence. You are
              entering unsupported terrain.  Namaste, and good luck!

On cross-compile systems the default frontend compiler is IBM XL for
the Blue Gene series and GCC on all other platforms.  The backend
compilers will either be automatically selected by the platform
detection (IBM Blue Gene series) or by the currently loaded
environment modules (Cray X series). If you want to customize these
settings please use the configure option '--with-custom-compilers' as
described above.

Although this package comes with recent versions of the OTF2 and Cube
libraries as well as the OPARI2 instrumenter included, it is possible
to use existing installations instead. Here, the --without option
means 'without external installation', i.e., the component provided
with the tarball will be used:

  --with-otf2[=<otf2-bindir>]
              Use an already installed and compatible OTF2 library
              (v2.0 or newer). Provide path to otf2-config.
              Auto-detected if already in $PATH.
  --with-cube[=<cube-bindir>]
              Use already installed and compatible Cube libraries
              (v4.3 or newer). Provide path to cube-config.
              Auto-detected if already in $PATH.
  --with-opari2[=<opari2-bindir>]
              Use an already installed and compatible OPARI2 (v2.0
              or newer). Provide path to opari2-config.
              Auto-detected if already in $PATH.

For the components otf2, cube, and opari2, the corresponding
--without-<component> or --with-<component>=no options will ignore the
<component>-config in $PATH but use the Score-P internal components.

Options to further specify which features and external packages should
be used to build Score-P are as follows:

  --enable-platform-mic   Force build for Intel MIC platform [no]
  --enable-debug          activate internal debug output [no]
  --enable-shared[=PKGS]  build shared libraries [default=no]
  --enable-static[=PKGS]  build static libraries [default=yes]
  --enable-backend-test-runs
              Enable execution of tests during 'make check' [no]
              (does not affect building of tests, though). If
              disabled, the files 'check-file-*' and/or
              'skipped_tests' listing the tests are generated in the
              corresponding build directory.
  --enable-cuda
              Enable or disable support for CUDA. Fail if support
              can't be satisfied but was requested.
  --enable-openacc
              Enable or disable support for OpenACC. (defaults to yes)
  --disable-gcc-plugin
              Disable support for the GCC plug-in
              instrumentation. Default is to determine support
              automatically. This disables it by request and fail
              if support can't be satisfied but was requested.
  --with-pdt=<path-to-binaries>
```

```
              Specifies the path to the program database toolkit
              (PDT) binaries, e.g., cparse.
  --with-extra-instrumentation-flags=flags
              Add additional instrumentation flags.
  --with-sionlib[=<sionlib-bindir>]
              Use an already installed sionlib. Provide path to
              sionconfig. Auto-detected if already in $PATH. This
              option is not used by Score-P itself but passed to an
              internal OTF2.
  --with-papi-header=<path-to-papi.h>
              If papi.h is not installed in the default location,
              specify the dirname where it can be found.
  --with-papi-lib=<path-to-libpapi.*>
              If libpapi.* is not installed in the default location,
              specify the dirname where it can be found.
  --with-libunwind=(yes|no|<Path to libunwind installation>)
              If you want to build with libunwind support but do
              not have a libunwind in a standard location, you
              need to explicitly specify the directory where it is
              installed. On non-cross-compile systems we search
              the system include and lib paths per default [yes];
              on cross-compile systems, however, you have to
              specify a path [no]. --with-libunwind is a shorthand
              for --with-libunwind-include=<Path/include> and
              --with-libunwind-lib=<Path/lib>. If these shorthand
              assumptions are not correct, you can use the
              explicit include and lib options directly.
  --with-libunwind-include=<Path to libunwind headers>
  --with-libunwind-lib=<Path to libunwind libraries>
  --with-libcudart=<Path to libcudart installation>
              If you want to build scorep with libcudart but do not
              have a libcudart in a standard location then you need
              to explicitly specify the directory where it is
              installed. On non-cross-compile systems we search the
              system include and lib paths per default [yes], on
              cross-compile systems however,you have to specify a
              path [no]. --with-libcudart is a shorthand for
              --with-libcudart-include=<Path/include> and
              --with-libcudart-lib=<Path/lib>. If these shorthand
              assumptions are not correct, you can use the explicit
              include and lib options directly.
  --with-libcudart-include=<Path to libcudart headers>
  --with-libcudart-lib=<Path to libcudart libraries>
  --with-libcuda=<Path to libcuda installation>
              Usually not needed, specifying --with-libcudart should
              be fine!
              If you want to build scorep with libcuda but do not
              have a libcuda in a standard location then you need to
              explicitly specify the directory where it is
              installed. On non-cross-compile systems we search the
              system include and lib paths per default [yes], on
              cross-compile systems however,you have to specify a
              path [no]. --with-libcuda is a shorthand for
              --with-libcuda-include=<Path/include> and
              --with-libcuda-lib=<Path/lib>. If these shorthand
              assumptions are not correct, you can use the explicit
              include and lib options directly.
  --with-libcuda-include=<Path to libcuda headers>
  --with-libcuda-lib=<Path to libcuda libraries>
  --with-libcupti=(yes|no|<Path to libcupti installation>)
              If you want to build with libcupti support but do
              not have a libcupti in a standard location, you need
              to explicitly specify the directory where it is
              installed. On non-cross-compile systems we search
              the system include and lib paths per default [yes];
              on cross-compile systems, however, you have to
              specify a path [no]. --with-libcupti is a shorthand
              for --with-libcupti-include=<Path/include> and
              --with-libcupti-lib=<Path/lib>. If these shorthand
              assumptions are not correct, you can use the
              explicit include and lib options directly.
  --with-libcupti-include=<Path to libcupti headers>
  --with-libcupti-lib=<Path to libcupti libraries>
```

```
  --with-openacc-include=<path-to-openacc.h>
              If openacc.h is not installed in the default
              location, specify the directory where it can be
              found.
  --with-openacc-prof-include=<path-to-acc_prof.h>
              If acc_prof.h is not installed in the default
              location, specify the directory where it can be
              found.
  --with-libOpenCL=(yes|no|<Path to libOpenCL installation>)
              If you want to build with libOpenCL support but do
              not have a libOpenCL in a standard location, you
              need to explicitly specify the directory where it is
              installed. On non-cross-compile systems we search
              the system include and lib paths per default [yes];
              on cross-compile systems, however, you have to
              specify a path [no]. --with-libOpenCL is a shorthand
              for --with-libOpenCL-include=<Path/include> and
              --with-libOpenCL-lib=<Path/lib>. If these shorthand
              assumptions are not correct, you can use the
              explicit include and lib options directly.
  --with-libOpenCL-include=<Path to libOpenCL headers>
  --with-libOpenCL-lib=<Path to libOpenCL libraries>
  --with-libpmi=(yes|no|<Path to libpmi installation>)
              If you want to build with libpmi support but do not
              have a libpmi in a standard location, you need to
              explicitly specify the directory where it is
              installed. On non-cross-compile systems we search
              the system include and lib paths per default [yes];
              on cross-compile systems, however, you have to
              specify a path [no]. --with-libpmi is a shorthand
              for --with-libpmi-include=<Path/include> and
              --with-libpmi-lib=<Path/lib>. If these shorthand
              assumptions are not correct, you can use the
              explicit include and lib options directly.
  --with-libpmi-include=<Path to libpmi headers>
  --with-libpmi-lib=<Path to libpmi libraries>
  --with-librca=(yes|no|<Path to librca installation>)
              If you want to build with librca support but do not
              have a librca in a standard location, you need to
              explicitly specify the directory where it is
              installed. On non-cross-compile systems we search
              the system include and lib paths per default [yes];
              on cross-compile systems, however, you have to
              specify a path [no]. --with-librca is a shorthand
              for --with-librca-include=<Path/include> and
              --with-librca-lib=<Path/lib>. If these shorthand
              assumptions are not correct, you can use the
              explicit include and lib options directly.
  --with-librca-include=<Path to librca headers>
  --with-librca-lib=<Path to librca libraries>
  --with-libbfd=<Path to libbfd installation>
              If you want to build scorep with libbfd but do not have
              a libbfd in a standard location then you need to
              explicitly specify the directory where it is
              installed. On non-cross-compile systems we search the
              system include and lib paths per default [yes], on
              cross-compile systems however,you have to specify a
              path [no]. --with-libbfd is a shorthand for
              --with-libbfd-include=<Path/include> and
              --with-libbfd-lib=<Path/lib>. If these shorthand
              assumptions are not correct, you can use the explicit
              include and lib options directly.
  --with-libbfd-include=<Path to libbfd headers>
  --with-libbfd-lib=<Path to libbfd libraries>
  --without-gui
              This option is passed to Cube to prevent building the
              Cube GUI. It is relevant only if the internal version
              of Cube is used (see also the --with-cube option).

Instead of passing command-line options to the 'configure' script, the package
configuration can also be influenced by setting the following environment
variables:
```

```
CC          C compiler command
CFLAGS      C compiler flags
LDFLAGS     linker flags, e.g. -L<lib dir> if you have libraries in a
            nonstandard directory <lib dir>
LIBS        libraries to pass to the linker, e.g. -l<library>
CPPFLAGS    (Objective) C/C++ preprocessor flags, e.g. -I<include dir> if
            you have headers in a nonstandard directory <include dir>
LT_SYS_LIBRARY_PATH
            User-defined run-time library search path.
CPP         C preprocessor
CXX         C++ compiler command
CXXFLAGS    C++ compiler flags
CXXCPP      C++ preprocessor
CCAS        assembler compiler command (defaults to CC)
CCASFLAGS   assembler compiler flags (defaults to CFLAGS)
CXXCPP      C++ preprocessor
F77         Fortran 77 compiler command
FFLAGS      Fortran 77 compiler flags
FC          Fortran compiler command
FCFLAGS     Fortran compiler flags
CC_FOR_BUILD
            C compiler command for the frontend build
CXX_FOR_BUILD
            C++ compiler command for the frontend build
F77_FOR_BUILD
            Fortran 77 compiler command for the frontend build
FC_FOR_BUILD
            Fortran compiler command for the frontend build
CPPFLAGS_FOR_BUILD
            (Objective) C/C++ preprocessor flags for the frontend build,
            e.g. -I<include dir> if you have headers in a nonstandard
            directory <include dir>
CFLAGS_FOR_BUILD
            C compiler flags for the frontend build
CXXFLAGS_FOR_BUILD
            C++ compiler flags for the frontend build
FFLAGS_FOR_BUILD
            Fortran 77 compiler flags for the frontend build
FCFLAGS_FOR_BUILD
            Fortran compiler flags for the frontend build
LDFLAGS_FOR_BUILD
            linker flags for the frontend build, e.g. -L<lib dir> if you
            have libraries in a nonstandard directory <lib dir>
LIBS_FOR_BUILD
            libraries to pass to the linker for the frontend build, e.g.
            -l<library>
MPICC       MPI C compiler command
MPICXX      MPI C++ compiler command
MPIF77      MPI Fortran 77 compiler command
MPIFC       MPI Fortran compiler command
MPI_CPPFLAGS
            MPI (Objective) C/C++ preprocessor flags, e.g. -I<include dir>
            if you have headers in a nonstandard directory <include dir>
MPI_CFLAGS  MPI C compiler flags
MPI_CXXFLAGS
            MPI C++ compiler flags
MPI_FFLAGS  MPI Fortran 77 compiler flags
MPI_FCFLAGS MPI Fortran compiler flags
MPI_LDFLAGS
            MPI linker flags, e.g. -L<lib dir> if you have libraries in a
            nonstandard directory <lib dir>
MPI_LIBS    MPI libraries to pass to the linker, e.g. -l<library>
SHMEMCC     SHMEM C compiler command
SHMEMCXX    SHMEM C++ compiler command
SHMEMF77    SHMEM Fortran 77 compiler command
SHMEMFC     SHMEM Fortran compiler command
SHMEM_CPPFLAGS
            SHMEM (Objective) C/C++ preprocessor flags, e.g. -I<include dir>
            if you have headers in a nonstandard directory <include dir>
SHMEM_CFLAGS
            SHMEM C compiler flags
SHMEM_CXXFLAGS
            SHMEM C++ compiler flags
```

```
  SHMEM_FFLAGS
              SHMEM Fortran 77 compiler flags
  SHMEM_FCFLAGS
              SHMEM Fortran compiler flags
  SHMEM_LDFLAGS
              SHMEM linker flags, e.g. -L<lib dir> if you have libraries in a
              nonstandard directory <lib dir>
  SHMEM_LIBS  SHMEM libraries to pass to the linker, e.g. -l<library>
  SHMEM_LIB_NAME
              name of the SHMEM library
  SHMEM_NAME  name of the implemented SHMEM specification
  CXXFLAGS_FOR_BUILD_SCORE
              C++ compiler flags for building scorep-score
  YACC        The 'Yet Another Compiler Compiler' implementation to use.
              Defaults to the first program found out of: 'bison -y', 'byacc',
              'yacc'.
  YFLAGS      The list of arguments that will be passed by default to $YACC.
              This script will default YFLAGS to the empty string to avoid a
              default value of '-d' given by some make applications.
  PTHREAD_CFLAGS
              CFLAGS used to compile Pthread programs
  PTHREAD_LIBS
              LIBS used to link Pthread programs
  RUNTIME_MANAGEMENT_TIMINGS
              Whether to activate time measurements for Score-P's
              SCOREP_InitMeasurement() and scorep_finalize() functions.
              Activation values are '1', 'yes', and 'true'. For developer use.
  PAPI_INC    Include path to the papi.h header.
  PAPI_LIB    Library path to the papi library.
  LIBUNWIND_INCLUDE
              Path to libunwind headers.
  LIBUNWIND_LIB
              Path to libunwind libraries.
  LIBCUDART_INCLUDE
              Path to libcudart headers.
  LIBCUDART_LIB
              Path to libcudart libraries.
  LIBCUDA_INCLUDE
              Path to libcuda headers.
  LIBCUDA_LIB Path to libcuda libraries.
  LIBCUPTI_INCLUDE
              Path to libcupti headers.
  LIBCUPTI_LIB
              Path to libcupti libraries.
  LIBOPENCL_INCLUDE
              Path to libOpenCL headers.
  LIBOPENCL_LIB
              Path to libOpenCL libraries.
  LIBBFD_INCLUDE
              Path to libbfd headers.
  LIBBFD_LIB  Path to libbfd libraries.
  OPENACC_INCLUDE
              Path to openacc.h header.
  OPENACC_PROFILING_INCLUDE
              Path to acc_prof.h header.
  LIBPMI_INCLUDE
              Path to libpmi headers.
  LIBPMI_LIB  Path to libpmi libraries.
  LIBRCA_INCLUDE
              Path to librca headers.
  LIBRCA_LIB  Path to librca libraries.

Building & Installing
=====================

Before building Score-P, carefully check whether the configuration summary
printed by the configure script matches your expectations (i.e., whether MPI
and/or OpenMP support is correctly enabled/disabled, external libraries are
used, etc). If everything is OK, Score-P can be built and installed using

        make
        make install
```

```
Note that parallel builds (i.e., using 'make -j <n>') are fully supported.


                     Platform-specific Instructions
                     ==============================

GNU Compiler Plug-In
====================

   On some system the necessary header files, for compiling with support for
the GNU Compiler plug-in instrumentation, are not installed by default. Therefore
an extra package needs to be installed.

On Debian and it's derivates the package is called:

     gcc-<version>-plugin-dev

On Fedora and it's derivates the mentioned package is called:

     gcc-plugin-devel


Intel Xeon Phi (aka. MIC)
=========================

   Building Score-P for the Intel MIC platform requires some extra
care, and in some cases two installations into the same location.
Therefore, we strongly recommend to strictly follow the procedure as
described below.

  1. Ensure that Intel compilers and Intel MPI (if desired) are
     installed and available in $PATH, and that the Intel Manycore
     Platform Software Stack (MPSS) is installed.

  2. Configure Score-P to use the MIC platform:

        mkdir _build-mic
        cd _build-mic
        ../configure --enable-platform-mic [other options, e.g., '--prefix']

  3. Build and install:

        make; make install

   In case a native MIC-only installation serves your needs, that's
it.  However, if the installation should also support instrumentation
and measurement of host code, a second installation *on top* of the
just installed one is required:

  4. Create a new build directory for the host build:

        cd ..
        mkdir _build-host
        cd _build-host

  5. Reconfigure for the host using *identical directory options* (e.g.,
     '--prefix' or '--bindir') as in step 2:

        ../configure [other options as used in step 2]

     This will automatically detect the already existing native MIC
     build and enable the required support in the host tools.  On
     non-cross-compile systems (e.g., typical Linux clusters), make
     sure to explicitly select Intel compiler support by passing
     '--with-nocross-compiler-suite=intel' to the configure script.

  6. Build and install:

        make; make install

   Note that this approach also works with VPATH builds (even with two
separate build directories) as long as the same options defining directory
```

locations are passed in steps 2 and 5.

# Generic Installation Instructions



## Basic Installation

Briefly, the shell commands ‘./configure; make; make install’ should configure, build, and install this package. The following more-detailed instructions are generic; see the section above for instructions specific to this package. Some packages provide this ‘INSTALL’ file but do not implement all of the features documented below. The lack of an optional feature in a given package is not necessarily a bug.

The ‘configure’ shell script attempts to guess correct values for various system-dependent variables used during compilation. It uses those values to create a ‘Makefile’ in each directory of the package. It may also create one or more ‘.h’ files containing system-dependent definitions. Finally, it creates a shell script ‘config.status’ that you can run in the future to recreate the current configuration, and a file ‘config.log’ containing compiler output (useful mainly for debugging ‘configure’).

It can also use an optional file (typically called ‘config.cache’ and enabled with ‘--cache-file=config.cache’ or simply ‘-C’) that saves the results of its tests to speed up reconfiguring. Caching is disabled by default to prevent problems with accidental use of stale cache files.

If you need to do unusual things to compile the package, please try to figure out how ‘configure’ could check whether to do them, and mail diffs or instructions to support@score-p.org so they can be considered for the next release. If you are using the cache, and at some point ‘config.cache’ contains results you don’t want to keep, you may remove or edit it.

The file ‘configure.ac’ (or ‘configure.in’) is used to create ‘configure’ by a program called ‘autoconf’. You need ‘configure.ac’ if you want to change it or regenerate ‘configure’ using a newer version of ‘autoconf’.

The simplest way to compile this package is:

1. ‘cd’ to the directory containing the package’s source code and type ‘./configure’ to configure the package for your system.

   Running ‘configure’ might take a while. While running, it prints some messages telling which features it is checking for.

2. Type ‘make’ to compile the package.

3. Optionally, type ‘make check’ to run any self-tests that come with the package, generally using the just-built uninstalled binaries.

4. Type ‘make install’ to install the programs and any data files and documentation. When installing into a prefix owned by root, it is recommended that the package be configured and built as a regular user, and only the ‘make install’ phase executed with root privileges.

5. Optionally, type ‘make installcheck’ to repeat any self-tests, but this time using the binaries in their final installed location.

```
     This target does not install anything.  Running this target as a
     regular user, particularly if the prior `make install' required
     root privileges, verifies that the installation completed
     correctly.

  6. You can remove the program binaries and object files from the
     source code directory by typing `make clean'.  To also remove the
     files that `configure' created (so you can compile the package for
     a different kind of computer), type `make distclean'.  There is
     also a `make maintainer-clean' target, but that is intended mainly
     for the package's developers.  If you use it, you may have to get
     all sorts of other programs in order to regenerate files that came
     with the distribution.

  7. Often, you can also type `make uninstall' to remove the installed
     files again.  In practice, not all packages have tested that
     uninstallation works correctly, even though it is required by the
     GNU Coding Standards.

  8. Some packages, particularly those that use Automake, provide `make
     distcheck', which can by used by developers to test that all other
     targets like `make install' and `make uninstall' work correctly.
     This target is generally not run by end users.

Compilers and Options
=====================

   Some systems require unusual options for compilation or linking that
the `configure' script does not know about.  Run `./configure --help'
for details on some of the pertinent environment variables.

   You can give `configure' initial values for configuration parameters
by setting variables in the command line or in the environment.  Here
is an example:

     ./configure CC=c99 CFLAGS=-g LIBS=-lposix

   *Note Defining Variables::, for more details.

Compiling For Multiple Architectures
====================================

   You can compile the package for more than one kind of computer at the
same time, by placing the object files for each architecture in their
own directory.  To do this, you can use GNU `make'.  `cd' to the
directory where you want the object files and executables to go and run
the `configure' script.  `configure' automatically checks for the
source code in the directory that `configure' is in and in `..'.  This
is known as a "VPATH" build.

   With a non-GNU `make', it is safer to compile the package for one
architecture at a time in the source code directory.  After you have
installed the package for one architecture, use `make distclean' before
reconfiguring for another architecture.

   On MacOS X 10.5 and later systems, you can create libraries and
executables that work on multiple system types--known as "fat" or
"universal" binaries--by specifying multiple `-arch' options to the
compiler but only a single `-arch' option to the preprocessor.  Like
this:

     ./configure CC="gcc -arch i386 -arch x86_64 -arch ppc -arch ppc64" \
                 CXX="g++ -arch i386 -arch x86_64 -arch ppc -arch ppc64" \
                 CPP="gcc -E" CXXCPP="g++ -E"

   This is not guaranteed to produce working output in all cases, you
may have to build one architecture at a time and combine the results
using the `lipo' tool if you have problems.

Installation Names
==================

   By default, `make install' installs the package's commands under
```

'/usr/local/bin', include files under '/usr/local/include', etc.  You
can specify an installation prefix other than '/usr/local' by giving
'configure' the option '--prefix=PREFIX', where PREFIX must be an
absolute file name.

   You can specify separate installation prefixes for
architecture-specific files and architecture-independent files.  If you
pass the option '--exec-prefix=PREFIX' to 'configure', the package uses
PREFIX as the prefix for installing programs and libraries.
Documentation and other data files still use the regular prefix.

   In addition, if you use an unusual directory layout you can give
options like '--bindir=DIR' to specify different values for particular
kinds of files.  Run 'configure --help' for a list of the directories
you can set and what kinds of files go in them.  In general, the
default for these options is expressed in terms of '${prefix}', so that
specifying just '--prefix' will affect all of the other directory
specifications that were not explicitly provided.

   The most portable way to affect installation locations is to pass the
correct locations to 'configure'; however, many packages provide one or
both of the following shortcuts of passing variable assignments to the
'make install' command line to change installation locations without
having to reconfigure or recompile.

   The first method involves providing an override variable for each
affected directory.  For example, 'make install
prefix=/alternate/directory' will choose an alternate location for all
directory configuration variables that were expressed in terms of
'${prefix}'.  Any directories that were specified during 'configure',
but not in terms of '${prefix}', must each be overridden at install
time for the entire installation to be relocated.  The approach of
makefile variable overrides for each directory variable is required by
the GNU Coding Standards, and ideally causes no recompilation.
However, some platforms have known limitations with the semantics of
shared libraries that end up requiring recompilation when using this
method, particularly noticeable in packages that use GNU Libtool.

   The second method involves providing the 'DESTDIR' variable.  For
example, 'make install DESTDIR=/alternate/directory' will prepend
'/alternate/directory' before all installation names.  The approach of
'DESTDIR' overrides is not required by the GNU Coding Standards, and
does not work on platforms that have drive letters.  On the other hand,
it does better at avoiding recompilation issues, and works well even
when some directory options were not specified in terms of '${prefix}'
at 'configure' time.

Optional Features
=================

   If the package supports it, you can cause programs to be installed
with an extra prefix or suffix on their names by giving 'configure' the
option '--program-prefix=PREFIX' or '--program-suffix=SUFFIX'.

   Some packages pay attention to '--enable-FEATURE' options to
'configure', where FEATURE indicates an optional part of the package.
They may also pay attention to '--with-PACKAGE' options, where PACKAGE
is something like 'gnu-as' or 'x' (for the X Window System).

   For packages that use the X Window System, 'configure' can usually
find the X include and library files automatically, but if it doesn't,
you can use the 'configure' options '--x-includes=DIR' and
'--x-libraries=DIR' to specify their locations.

   Some packages offer the ability to configure how verbose the
execution of 'make' will be.  For these packages, running './configure
--enable-silent-rules' sets the default to minimal output, which can be
overridden with 'make V=1'; while running './configure
--disable-silent-rules' sets the default to verbose, which can be
overridden with 'make V=0'.

Particular systems
==================

```
   On HP-UX, the default C compiler is not ANSI C compatible.  If GNU
CC is not installed, it is recommended to use the following options in
order to use an ANSI C compiler:

     ./configure CC="cc -Ae -D_XOPEN_SOURCE=500"

and if that doesn't work, install pre-built binaries of GCC for HP-UX.

   On OSF/1 a.k.a. Tru64, some versions of the default C compiler cannot
parse its '<wchar.h>' header file.  The option '-nodtk' can be used as
a workaround.  If GNU CC is not installed, it is therefore recommended
to try

     ./configure CC="cc"

and if that doesn't work, try

     ./configure CC="cc -nodtk"

   On Solaris, don't put '/usr/ucb' early in your 'PATH'.  This
directory contains several dysfunctional programs; working variants of
these programs are available in '/usr/bin'.  So, if you need '/usr/ucb'
in your 'PATH', put it _after_ '/usr/bin'.

   On Haiku, software installed for all users goes in '/boot/common',
not '/usr/local'.  It is recommended to use the following options:

     ./configure --prefix=/boot/common

Specifying the System Type
==========================

   There may be some features 'configure' cannot figure out
automatically, but needs to determine by the type of machine the package
will run on.  Usually, assuming the package is built to be run on the
_same_ architectures, 'configure' can figure that out, but if it prints
a message saying it cannot guess the machine type, give it the
'--build=TYPE' option.  TYPE can either be a short name for the system
type, such as 'sun4', or a canonical name which has the form:

     CPU-COMPANY-SYSTEM

where SYSTEM can have one of these forms:

     OS
     KERNEL-OS

   See the file 'config.sub' for the possible values of each field.  If
'config.sub' isn't included in this package, then this package doesn't
need to know the machine type.

   If you are _building_ compiler tools for cross-compiling, you should
use the option '--target=TYPE' to select the type of system they will
produce code for.

   If you want to _use_ a cross compiler, that generates code for a
platform different from the build platform, you should specify the
"host" platform (i.e., that on which the generated programs will
eventually be run) with '--host=TYPE'.

Sharing Defaults
================

   If you want to set default values for 'configure' scripts to share,
you can create a site shell script called 'config.site' that gives
default values for variables like 'CC', 'cache_file', and 'prefix'.
'configure' looks for 'PREFIX/share/config.site' if it exists, then
'PREFIX/etc/config.site' if it exists.  Or, you can set the
'CONFIG_SITE' environment variable to the location of the site script.
A warning: not all 'configure' scripts look for a site script.

Defining Variables
```

```
==================

   Variables not defined in a site shell script can be set in the
environment passed to 'configure'.  However, some packages may run
configure again during the build, and the customized values of these
variables may be lost.  In order to avoid this problem, you should set
them in the 'configure' command line, using 'VAR=value'.  For example:

     ./configure CC=/usr/local2/bin/gcc

causes the specified 'gcc' to be used as the C compiler (unless it is
overridden in the site shell script).

Unfortunately, this technique does not work for 'CONFIG_SHELL' due to
an Autoconf bug.  Until the bug is fixed you can use this workaround:

     CONFIG_SHELL=/bin/bash /bin/bash ./configure CONFIG_SHELL=/bin/bash

'configure' Invocation
======================

   'configure' recognizes the following options to control how it
operates.

'--help'
'-h'
     Print a summary of all of the options to 'configure', and exit.

'--help=short'
'--help=recursive'
     Print a summary of the options unique to this package's
     'configure', and exit.  The 'short' variant lists options used
     only in the top level, while the 'recursive' variant lists options
     also present in any nested packages.

'--version'
'-V'
     Print the version of Autoconf used to generate the 'configure'
     script, and exit.

'--cache-file=FILE'
     Enable the cache: use and save the results of the tests in FILE,
     traditionally 'config.cache'.  FILE defaults to '/dev/null' to
     disable caching.

'--config-cache'
'-C'
     Alias for '--cache-file=config.cache'.

'--quiet'
'--silent'
'-q'
     Do not print messages saying which checks are being made.  To
     suppress all normal output, redirect it to '/dev/null' (any error
     messages will still be shown).

'--srcdir=DIR'
     Look for the package's source code in directory DIR.  Usually
     'configure' can determine that directory automatically.

'--prefix=DIR'
     Use DIR as the installation prefix.  *note Installation Names::
     for more details, including other options available for fine-tuning
     the installation locations.

'--no-create'
'-n'
     Run the configure checks, but stop before creating any output
     files.

'configure' also accepts some other, not widely useful, options.  Run
'configure --help' for more details.
```

# Appendix B

# MPI wrapper affiliation

Some wrapper functions are affiliated with a function group that has not been described for direct user access in section 5.7.1. These groups are subgroups that contain function calls that are only enabled when both main groups are enabled. The reason for this is to control the amount of events generated during measurement, a user might want to turn off the measurement of non-critical function calls before the measurement of the complete main group is turned off. Subgroups can either be related to `MISC` (miscellaneous functions, e.g. handle conversion), `EXT` (external interfaces, e.g. handle attributes), or `ERR` (error handlers).

For example, the functions in group `CG_MISC` will only generate events if both groups `CG` and `MISC` are enabled at runtime.

## B.1 Function to group

| Function | Group |
|---|---|
| MPI_Abort | EXT |
| MPI_Accumulate | RMA |
| MPI_Add_error_class | ERR |
| MPI_Add_error_code | ERR |
| MPI_Add_error_string | ERR |
| MPI_Address | MISC |
| MPI_Allgather | COLL |
| MPI_Allgatherv | COLL |
| MPI_Alloc_mem | MISC |
| MPI_Allreduce | COLL |
| MPI_Alltoall | COLL |
| MPI_Alltoallv | COLL |
| MPI_Alltoallw | COLL |
| MPI_Attr_delete | CG_EXT |
| MPI_Attr_get | CG_EXT |
| MPI_Attr_put | CG_EXT |
| MPI_Barrier | COLL |
| MPI_Bcast | COLL |
| MPI_Bsend | P2P |
| MPI_Bsend_init | P2P |
| MPI_Buffer_attach | P2P |

| | |
|---|---|
| MPI_Buffer_detach | P2P |
| MPI_Cancel | P2P |
| MPI_Cart_coords | TOPO |
| MPI_Cart_create | TOPO |
| MPI_Cart_get | TOPO |
| MPI_Cart_map | TOPO |
| MPI_Cart_rank | TOPO |
| MPI_Cart_shift | TOPO |
| MPI_Cart_sub | TOPO |
| MPI_Cartdim_get | TOPO |
| MPI_Close_port | SPAWN |
| MPI_Comm_accept | SPAWN |
| MPI_Comm_c2f | CG_MISC |
| MPI_Comm_call_errhandler | CG_ERR |
| MPI_Comm_compare | CG |
| MPI_Comm_connect | SPAWN |
| MPI_Comm_create | CG |
| MPI_Comm_create_errhandler | CG_ERR |
| MPI_Comm_create_group | CG |
| MPI_Comm_create_keyval | CG_EXT |
| MPI_Comm_delete_attr | CG_EXT |
| MPI_Comm_disconnect | SPAWN |
| MPI_Comm_dup | CG |
| MPI_Comm_dup_with_info | CG |
| MPI_Comm_f2c | CG_MISC |
| MPI_Comm_free | CG |
| MPI_Comm_free_keyval | CG_EXT |
| MPI_Comm_get_attr | CG_EXT |
| MPI_Comm_get_errhandler | CG_ERR |
| MPI_Comm_get_info | CG_EXT |
| MPI_Comm_get_name | CG_EXT |
| MPI_Comm_get_parent | SPAWN |
| MPI_Comm_group | CG |
| MPI_Comm_idup | CG |
| MPI_Comm_join | SPAWN |
| MPI_Comm_rank | CG |
| MPI_Comm_remote_group | CG |
| MPI_Comm_remote_size | CG |
| MPI_Comm_set_attr | CG_EXT |
| MPI_Comm_set_errhandler | CG_ERR |
| MPI_Comm_set_info | CG_EXT |
| MPI_Comm_set_name | CG_EXT |
| MPI_Comm_size | CG |
| MPI_Comm_spawn | SPAWN |
| MPI_Comm_spawn_multiple | SPAWN |
| MPI_Comm_split | CG |
| MPI_Comm_split_type | CG |

| | |
|---|---|
| MPI_Comm_test_inter | CG |
| MPI_Compare_and_swap | RMA |
| MPI_Dims_create | TOPO |
| MPI_Dist_graph_create | TOPO |
| MPI_Dist_graph_create_adjacent | TOPO |
| MPI_Dist_graph_neighbors | TOPO |
| MPI_Dist_graph_neighbors_count | TOPO |
| MPI_Errhandler_create | ERR |
| MPI_Errhandler_free | ERR |
| MPI_Errhandler_get | ERR |
| MPI_Errhandler_set | ERR |
| MPI_Error_class | ERR |
| MPI_Error_string | ERR |
| MPI_Exscan | COLL |
| MPI_Fetch_and_op | RMA |
| MPI_File_c2f | IO_MISC |
| MPI_File_call_errhandler | IO_ERR |
| MPI_File_close | IO |
| MPI_File_create_errhandler | IO_ERR |
| MPI_File_delete | IO |
| MPI_File_f2c | IO_MISC |
| MPI_File_get_amode | IO |
| MPI_File_get_atomicity | IO |
| MPI_File_get_byte_offset | IO |
| MPI_File_get_errhandler | IO_ERR |
| MPI_File_get_group | IO |
| MPI_File_get_info | IO |
| MPI_File_get_position | IO |
| MPI_File_get_position_shared | IO |
| MPI_File_get_size | IO |
| MPI_File_get_type_extent | IO |
| MPI_File_get_view | IO |
| MPI_File_iread | IO |
| MPI_File_iread_all | IO |
| MPI_File_iread_at | IO |
| MPI_File_iread_at_all | IO |
| MPI_File_iread_shared | IO |
| MPI_File_iwrite | IO |
| MPI_File_iwrite_all | IO |
| MPI_File_iwrite_at | IO |
| MPI_File_iwrite_at_all | IO |
| MPI_File_iwrite_shared | IO |
| MPI_File_open | IO |
| MPI_File_preallocate | IO |
| MPI_File_read | IO |
| MPI_File_read_all | IO |
| MPI_File_read_all_begin | IO |

| | |
|---|---|
| MPI_File_read_all_end | IO |
| MPI_File_read_at | IO |
| MPI_File_read_at_all | IO |
| MPI_File_read_at_all_begin | IO |
| MPI_File_read_at_all_end | IO |
| MPI_File_read_ordered | IO |
| MPI_File_read_ordered_begin | IO |
| MPI_File_read_ordered_end | IO |
| MPI_File_read_shared | IO |
| MPI_File_seek | IO |
| MPI_File_seek_shared | IO |
| MPI_File_set_atomicity | IO |
| MPI_File_set_errhandler | IO_ERR |
| MPI_File_set_info | IO |
| MPI_File_set_size | IO |
| MPI_File_set_view | IO |
| MPI_File_sync | IO |
| MPI_File_write | IO |
| MPI_File_write_all | IO |
| MPI_File_write_all_begin | IO |
| MPI_File_write_all_end | IO |
| MPI_File_write_at | IO |
| MPI_File_write_at_all | IO |
| MPI_File_write_at_all_begin | IO |
| MPI_File_write_at_all_end | IO |
| MPI_File_write_ordered | IO |
| MPI_File_write_ordered_begin | IO |
| MPI_File_write_ordered_end | IO |
| MPI_File_write_shared | IO |
| MPI_Finalize | ENV |
| MPI_Finalized | ENV |
| MPI_Free_mem | MISC |
| MPI_Gather | COLL |
| MPI_Gatherv | COLL |
| MPI_Get | RMA |
| MPI_Get_accumulate | RMA |
| MPI_Get_address | MISC |
| MPI_Get_count | EXT |
| MPI_Get_elements | EXT |
| MPI_Get_elements_x | EXT |
| MPI_Get_library_version | ENV |
| MPI_Get_processor_name | EXT |
| MPI_Get_version | MISC |
| MPI_Graph_create | TOPO |
| MPI_Graph_get | TOPO |
| MPI_Graph_map | TOPO |
| MPI_Graph_neighbors | TOPO |

| | |
|---|---|
| MPI_Graph_neighbors_count | TOPO |
| MPI_Graphdims_get | TOPO |
| MPI_Grequest_complete | EXT |
| MPI_Grequest_start | EXT |
| MPI_Group_c2f | CG_MISC |
| MPI_Group_compare | CG |
| MPI_Group_difference | CG |
| MPI_Group_excl | CG |
| MPI_Group_f2c | CG_MISC |
| MPI_Group_free | CG |
| MPI_Group_incl | CG |
| MPI_Group_intersection | CG |
| MPI_Group_range_excl | CG |
| MPI_Group_range_incl | CG |
| MPI_Group_rank | CG |
| MPI_Group_size | CG |
| MPI_Group_translate_ranks | CG |
| MPI_Group_union | CG |
| MPI_Iallgather | COLL |
| MPI_Iallgatherv | COLL |
| MPI_Iallreduce | COLL |
| MPI_Ialltoall | COLL |
| MPI_Ialltoallv | COLL |
| MPI_Ialltoallw | COLL |
| MPI_Ibarrier | COLL |
| MPI_Ibcast | COLL |
| MPI_Ibsend | P2P |
| MPI_Iexscan | COLL |
| MPI_Igather | COLL |
| MPI_Igatherv | COLL |
| MPI_Improbe | P2P |
| MPI_Imrecv | P2P |
| MPI_Ineighbor_allgather | TOPO |
| MPI_Ineighbor_allgatherv | TOPO |
| MPI_Ineighbor_alltoall | TOPO |
| MPI_Ineighbor_alltoallv | TOPO |
| MPI_Ineighbor_alltoallw | TOPO |
| MPI_Info_c2f | MISC |
| MPI_Info_create | MISC |
| MPI_Info_delete | MISC |
| MPI_Info_dup | MISC |
| MPI_Info_f2c | MISC |
| MPI_Info_free | MISC |
| MPI_Info_get | MISC |
| MPI_Info_get_nkeys | MISC |
| MPI_Info_get_nthkey | MISC |
| MPI_Info_get_valuelen | MISC |

| | |
|---|---|
| MPI_Info_set | MISC |
| MPI_Init | ENV |
| MPI_Init_thread | ENV |
| MPI_Initialized | ENV |
| MPI_Intercomm_create | CG |
| MPI_Intercomm_merge | CG |
| MPI_Iprobe | P2P |
| MPI_Irecv | P2P |
| MPI_Ireduce | COLL |
| MPI_Ireduce_scatter | COLL |
| MPI_Ireduce_scatter_block | COLL |
| MPI_Irsend | P2P |
| MPI_Is_thread_main | ENV |
| MPI_Iscan | COLL |
| MPI_Iscatter | COLL |
| MPI_Iscatterv | COLL |
| MPI_Isend | P2P |
| MPI_Issend | P2P |
| MPI_Keyval_create | CG_EXT |
| MPI_Keyval_free | CG_EXT |
| MPI_Lookup_name | SPAWN |
| MPI_Mprobe | P2P |
| MPI_Mrecv | P2P |
| MPI_Neighbor_allgather | TOPO |
| MPI_Neighbor_allgatherv | TOPO |
| MPI_Neighbor_alltoall | TOPO |
| MPI_Neighbor_alltoallv | TOPO |
| MPI_Neighbor_alltoallw | TOPO |
| MPI_Op_c2f | MISC |
| MPI_Op_commutative | MISC |
| MPI_Op_create | MISC |
| MPI_Op_f2c | MISC |
| MPI_Op_free | MISC |
| MPI_Open_port | SPAWN |
| MPI_Pack | TYPE |
| MPI_Pack_external | TYPE |
| MPI_Pack_external_size | TYPE |
| MPI_Pack_size | TYPE |
| MPI_Pcontrol | PERF |
| MPI_Probe | P2P |
| MPI_Publish_name | SPAWN |
| MPI_Put | RMA |
| MPI_Query_thread | ENV |
| MPI_Raccumulate | RMA |
| MPI_Recv | P2P |
| MPI_Recv_init | P2P |
| MPI_Reduce | COLL |

| | |
|---|---|
| MPI_Reduce_local | COLL |
| MPI_Reduce_scatter | COLL |
| MPI_Reduce_scatter_block | COLL |
| MPI_Register_datarep | IO |
| MPI_Request_c2f | MISC |
| MPI_Request_f2c | MISC |
| MPI_Request_free | P2P |
| MPI_Request_get_status | MISC |
| MPI_Rget | RMA |
| MPI_Rget_accumulate | RMA |
| MPI_Rput | RMA |
| MPI_Rsend | P2P |
| MPI_Rsend_init | P2P |
| MPI_Scan | COLL |
| MPI_Scatter | COLL |
| MPI_Scatterv | COLL |
| MPI_Send | P2P |
| MPI_Send_init | P2P |
| MPI_Sendrecv | P2P |
| MPI_Sendrecv_replace | P2P |
| MPI_Sizeof | TYPE |
| MPI_Ssend | P2P |
| MPI_Ssend_init | P2P |
| MPI_Start | P2P |
| MPI_Startall | P2P |
| MPI_Status_c2f | MISC |
| MPI_Status_f2c | MISC |
| MPI_Status_set_cancelled | EXT |
| MPI_Status_set_elements | EXT |
| MPI_Status_set_elements_x | EXT |
| MPI_Test | P2P |
| MPI_Test_cancelled | P2P |
| MPI_Testall | P2P |
| MPI_Testany | P2P |
| MPI_Testsome | P2P |
| MPI_Topo_test | TOPO |
| MPI_Type_c2f | TYPE_MISC |
| MPI_Type_commit | TYPE |
| MPI_Type_contiguous | TYPE |
| MPI_Type_create_darray | TYPE |
| MPI_Type_create_f90_complex | TYPE |
| MPI_Type_create_f90_integer | TYPE |
| MPI_Type_create_f90_real | TYPE |
| MPI_Type_create_hindexed | TYPE |
| MPI_Type_create_hindexed_block | TYPE |
| MPI_Type_create_hvector | TYPE |
| MPI_Type_create_indexed_block | TYPE |

| | |
|---|---|
| MPI_Type_create_keyval | TYPE_EXT |
| MPI_Type_create_resized | TYPE |
| MPI_Type_create_struct | TYPE |
| MPI_Type_create_subarray | TYPE |
| MPI_Type_delete_attr | TYPE_EXT |
| MPI_Type_dup | TYPE |
| MPI_Type_extent | TYPE |
| MPI_Type_f2c | TYPE_MISC |
| MPI_Type_free | TYPE |
| MPI_Type_free_keyval | TYPE_EXT |
| MPI_Type_get_attr | TYPE_EXT |
| MPI_Type_get_contents | TYPE |
| MPI_Type_get_envelope | TYPE |
| MPI_Type_get_extent | TYPE |
| MPI_Type_get_extent_x | TYPE |
| MPI_Type_get_name | TYPE_EXT |
| MPI_Type_get_true_extent | TYPE |
| MPI_Type_get_true_extent_x | TYPE |
| MPI_Type_hindexed | TYPE |
| MPI_Type_hvector | TYPE |
| MPI_Type_indexed | TYPE |
| MPI_Type_lb | TYPE |
| MPI_Type_match_size | TYPE |
| MPI_Type_set_attr | TYPE_EXT |
| MPI_Type_set_name | TYPE_EXT |
| MPI_Type_size | TYPE |
| MPI_Type_size_x | TYPE |
| MPI_Type_struct | TYPE |
| MPI_Type_ub | TYPE |
| MPI_Type_vector | TYPE |
| MPI_Unpack | TYPE |
| MPI_Unpack_external | TYPE |
| MPI_Unpublish_name | SPAWN |
| MPI_Wait | P2P |
| MPI_Waitall | P2P |
| MPI_Waitany | P2P |
| MPI_Waitsome | P2P |
| MPI_Win_allocate | RMA |
| MPI_Win_allocate_shared | RMA |
| MPI_Win_attach | RMA |
| MPI_Win_c2f | RMA_MISC |
| MPI_Win_call_errhandler | RMA_ERR |
| MPI_Win_complete | RMA |
| MPI_Win_create | RMA |
| MPI_Win_create_dynamic | RMA |
| MPI_Win_create_errhandler | RMA_ERR |
| MPI_Win_create_keyval | RMA_EXT |

| | |
|---|---|
| MPI_Win_delete_attr | RMA_EXT |
| MPI_Win_detach | RMA |
| MPI_Win_f2c | RMA_MISC |
| MPI_Win_fence | RMA |
| MPI_Win_flush | RMA |
| MPI_Win_flush_all | RMA |
| MPI_Win_flush_local | RMA |
| MPI_Win_flush_local_all | RMA |
| MPI_Win_free | RMA |
| MPI_Win_free_keyval | RMA_EXT |
| MPI_Win_get_attr | RMA_EXT |
| MPI_Win_get_errhandler | RMA_ERR |
| MPI_Win_get_group | RMA |
| MPI_Win_get_info | RMA_EXT |
| MPI_Win_get_name | RMA_EXT |
| MPI_Win_lock | RMA |
| MPI_Win_lock_all | RMA |
| MPI_Win_post | RMA |
| MPI_Win_set_attr | RMA_EXT |
| MPI_Win_set_errhandler | RMA_ERR |
| MPI_Win_set_info | RMA_EXT |
| MPI_Win_set_name | RMA_EXT |
| MPI_Win_shared_query | RMA |
| MPI_Win_start | RMA |
| MPI_Win_sync | RMA |
| MPI_Win_test | RMA |
| MPI_Win_unlock | RMA |
| MPI_Win_unlock_all | RMA |
| MPI_Win_wait | RMA |
| MPI_Wtick | EXT |
| MPI_Wtime | EXT |

## B.2 Group to function

| **CG** - **Communicators and Groups** |
|---|
| MPI_Comm_compare, MPI_Comm_create, MPI_Comm_create_group, MPI_Comm_dup, MPI_Comm_dup_with_info, MPI_Comm_free, MPI_Comm_group, MPI_Comm_idup, MPI_Comm_rank, MPI_Comm_remote_group, MPI_Comm_remote_size, MPI_Comm_size, MPI_Comm_split, MPI_Comm_split_type, MPI_Comm_test_inter, MPI_Group_compare, MPI_Group_difference, MPI_Group_excl, MPI_Group_free, MPI_Group_incl, MPI_Group_intersection, MPI_Group_range_excl, MPI_Group_range_incl, MPI_Group_rank, MPI_Group_size, MPI_Group_translate_ranks, MPI_Group_union, MPI_Intercomm_create, MPI_Intercomm_merge, |

| **CG_ERR** - **Error handlers for Communicators and Groups** |
|---|
| MPI_Comm_call_errhandler, MPI_Comm_create_errhandler, MPI_Comm_get_errhandler, MPI_Comm_set_errhandler, |

| **CG_EXT** - **External interfaces for Communicators and Groups** |
|---|
| MPI_Attr_delete, MPI_Attr_get, MPI_Attr_put, MPI_Comm_create_keyval, MPI_Comm_delete_attr, MPI_Comm_free_keyval, MPI_Comm_get_attr, MPI_Comm_get_info, MPI_Comm_get_name, MPI_Comm_set_attr, MPI_Comm_set_info, MPI_Comm_set_name, MPI_Keyval_create, MPI_Keyval_free, |

| **CG_MISC** - **Miscellaneous functions for Communicators and Groups** |
|---|
| MPI_Comm_c2f, MPI_Comm_f2c, MPI_Group_c2f, MPI_Group_f2c, |

| **COLL** - **Collective communication** |
|---|

MPI_Allgather, MPI_Allgatherv, MPI_Allreduce, MPI_Alltoall, MPI_Alltoallv, MPI_Alltoallw, MPI_Barrier, MPI_Bcast, MPI_Exscan, MPI_Gather, MPI_Gatherv, MPI_Iallgather, MPI_Iallgatherv, MPI_Iallreduce, MPI_Ialltoall, MPI_Ialltoallv, MPI_Ialltoallw, MPI_Ibarrier, MPI_Ibcast, MPI_Iexscan, MPI_Igather, MPI_Igatherv, MPI_Ireduce, MPI_Ireduce_scatter, MPI_Ireduce_scatter_block, MPI_Iscan, MPI_Iscatter, MPI_Iscatterv, MPI_Reduce, MPI_Reduce_local, MPI_Reduce_scatter, MPI_Reduce_scatter_block, MPI_Scan, MPI_Scatter, MPI_Scatterv,

**ENV** - **Environmental management**

MPI_Finalize, MPI_Finalized, MPI_Get_library_version, MPI_Init, MPI_Init_thread, MPI_Initialized, MPI_Is_thread_main, MPI_Query_thread,

**ERR** - **Common error handlers**

MPI_Add_error_class, MPI_Add_error_code, MPI_Add_error_string, MPI_Errhandler_create, MPI_Errhandler_free, MPI_Errhandler_get, MPI_Errhandler_set, MPI_Error_class, MPI_Error_string,

**IO** - **Parallel I/O**

MPI_File_close, MPI_File_delete, MPI_File_get_amode, MPI_File_get_atomicity, MPI_File_get_byte_offset, MPI_File_get_group, MPI_File_get_info, MPI_File_get_position, MPI_File_get_position_shared, MPI_File_get_size, MPI_File_get_type_extent, MPI_File_get_view, MPI_File_iread, MPI_File_iread_all, MPI_File_iread_at, MPI_File_iread_at_all, MPI_File_iread_shared, MPI_File_iwrite, MPI_File_iwrite_all, MPI_File_iwrite_at, MPI_File_iwrite_at_all, MPI_File_iwrite_shared, MPI_File_open, MPI_File_preallocate, MPI_File_read, MPI_File_read_all, MPI_File_read_all_begin, MPI_File_read_all_end, MPI_File_read_at, MPI_File_read_at_all, MPI_File_read_at_all_begin, MPI_File_read_at_all_end, MPI_File_read_ordered, MPI_File_read_ordered_begin, MPI_File_read_ordered_end, MPI_File_read_shared, MPI_File_seek, MPI_File_seek_shared, MPI_File_set_atomicity, MPI_File_set_info, MPI_File_set_size, MPI_File_set_view, MPI_File_sync, MPI_File_write, MPI_File_write_all, MPI_File_write_all_begin, MPI_File_write_all_end, MPI_File_write_at, MPI_File_write_at_all, MPI_File_write_at_all_begin, MPI_File_write_at_all_end, MPI_File_write_ordered, MPI_File_write_ordered_begin, MPI_File_write_ordered_end, MPI_File_write_shared, MPI_Register_datarep,

**IO_ERR** - **Error handlers for Parallel I/O**

MPI_File_call_errhandler, MPI_File_create_errhandler, MPI_File_get_errhandler, MPI_File_set_errhandler,

**IO_MISC** - **Miscellaneous functions for Parallel I/O**

MPI_File_c2f, MPI_File_f2c,

**EXT** - **Common external interfaces**

MPI_Abort, MPI_Get_count, MPI_Get_elements, MPI_Get_elements_x, MPI_Get_processor_name, MPI_Grequest_complete, MPI_Grequest_start, MPI_Status_set_cancelled, MPI_Status_set_elements, MPI_Status_set_elements_x, MPI_Wtick, MPI_Wtime,

**MISC** - **Miscellaneous functions**

MPI_Address, MPI_Alloc_mem, MPI_Free_mem, MPI_Get_address, MPI_Get_version, MPI_Info_c2f, MPI_Info_create, MPI_Info_delete, MPI_Info_dup, MPI_Info_f2c, MPI_Info_free, MPI_Info_get, MPI_Info_get_nkeys, MPI_Info_get_nthkey, MPI_Info_get_valuelen, MPI_Info_set, MPI_Op_c2f, MPI_Op_commutative, MPI_Op_create, MPI_Op_f2c, MPI_Op_free, MPI_Request_c2f, MPI_Request_f2c, MPI_Request_get_status, MPI_Status_c2f, MPI_Status_f2c,

**P2P** - **Point-to-point communication**

MPI_Bsend, MPI_Bsend_init, MPI_Buffer_attach, MPI_Buffer_detach, MPI_Cancel, MPI_Ibsend, MPI_Improbe, MPI_Imrecv, MPI_Iprobe, MPI_Irecv, MPI_Irsend, MPI_Isend, MPI_Issend, MPI_Mprobe, MPI_Mrecv, MPI_Probe, MPI_Recv, MPI_Recv_init, MPI_Request_free, MPI_Rsend, MPI_Rsend_init, MPI_Send, MPI_Send_init, MPI_Sendrecv, MPI_Sendrecv_replace, MPI_Ssend, MPI_Ssend_init, MPI_Start, MPI_Startall, MPI_Test, MPI_Test_cancelled, MPI_Testall, MPI_Testany, MPI_Testsome, MPI_Wait, MPI_Waitall, MPI_Waitany, MPI_Waitsome,

**PERF** - **Profiling Interface**

MPI_Pcontrol,

**RMA** - **One-sided communication (Remote Memory Access)**

| |
|---|
| MPI_Accumulate, MPI_Compare_and_swap, MPI_Fetch_and_op, MPI_Get, MPI_Get_accumulate, MPI_Put, MPI_Raccumulate, MPI_Rget, MPI_Rget_accumulate, MPI_Rput, MPI_Win_allocate, MPI_Win_allocate_shared, MPI_Win_attach, MPI_Win_complete, MPI_Win_create, MPI_Win_create_dynamic, MPI_Win_detach, MPI_Win_fence, MPI_Win_flush, MPI_Win_flush_all, MPI_Win_flush_local, MPI_Win_flush_local_all, MPI_Win_free, MPI_Win_get_group, MPI_Win_lock, MPI_Win_lock_all, MPI_Win_post, MPI_Win_shared_query, MPI_Win_start, MPI_Win_sync, MPI_Win_test, MPI_Win_unlock, MPI_Win_unlock_all, MPI_Win_wait, |

| **RMA_ERR** - **Error handlers for One-sided communication (Remote Memory Access)** |
|---|
| MPI_Win_call_errhandler, MPI_Win_create_errhandler, MPI_Win_get_errhandler, MPI_Win_set_errhandler, |

| **RMA_EXT** - **External interfaces for One-sided communication (Remote Memory Access)** |
|---|
| MPI_Win_create_keyval, MPI_Win_delete_attr, MPI_Win_free_keyval, MPI_Win_get_attr, MPI_Win_get_info, MPI_Win_get_name, MPI_Win_set_attr, MPI_Win_set_info, MPI_Win_set_name, |

| **RMA_MISC** - **Miscellaneous functions for One-sided communication (Remote Memory Access)** |
|---|
| MPI_Win_c2f, MPI_Win_f2c, |

| **SPAWN** - **Process spawning** |
|---|
| MPI_Close_port, MPI_Comm_accept, MPI_Comm_connect, MPI_Comm_disconnect, MPI_Comm_get_parent, MPI_Comm_join, MPI_Comm_spawn, MPI_Comm_spawn_multiple, MPI_Lookup_name, MPI_Open_port, MPI_Publish_name, MPI_Unpublish_name, |

| **TOPO** - **Topology (cartesian and graph) communicators** |
|---|
| MPI_Cart_coords, MPI_Cart_create, MPI_Cart_get, MPI_Cart_map, MPI_Cart_rank, MPI_Cart_shift, MPI_Cart_sub, MPI_Cartdim_get, MPI_Dims_create, MPI_Dist_graph_create, MPI_Dist_graph_create_adjacent, MPI_Dist_graph_neighbors, MPI_Dist_graph_neighbors_count, MPI_Graph_create, MPI_Graph_get, MPI_Graph_map, MPI_Graph_neighbors, MPI_Graph_neighbors_count, MPI_Graphdims_get, MPI_Ineighbor_allgather, MPI_Ineighbor_allgatherv, MPI_Ineighbor_alltoall, MPI_Ineighbor_alltoallv, MPI_Ineighbor_alltoallw, MPI_Neighbor_allgather, MPI_Neighbor_allgatherv, MPI_Neighbor_alltoall, MPI_Neighbor_alltoallv, MPI_Neighbor_alltoallw, MPI_Topo_test, |

| **TYPE** - **Datatypes** |
|---|
| MPI_Pack, MPI_Pack_external, MPI_Pack_external_size, MPI_Pack_size, MPI_Sizeof, MPI_Type_commit, MPI_Type_contiguous, MPI_Type_create_darray, MPI_Type_create_f90_complex, MPI_Type_create_f90_integer, MPI_Type_create_f90_real, MPI_Type_create_hindexed, MPI_Type_create_hindexed_block, MPI_Type_create_hvector, MPI_Type_create_indexed_block, MPI_Type_create_resized, MPI_Type_create_struct, MPI_Type_create_subarray, MPI_Type_dup, MPI_Type_extent, MPI_Type_free, MPI_Type_get_contents, MPI_Type_get_envelope, MPI_Type_get_extent, MPI_Type_get_extent_x, MPI_Type_get_true_extent, MPI_Type_get_true_extent_x, MPI_Type_hindexed, MPI_Type_hvector, MPI_Type_indexed, MPI_Type_lb, MPI_Type_match_size, MPI_Type_size, MPI_Type_size_x, MPI_Type_struct, MPI_Type_ub, MPI_Type_vector, MPI_Unpack, MPI_Unpack_external, |

| **TYPE_EXT** - **External interfaces for datatypes** |
|---|
| MPI_Type_create_keyval, MPI_Type_delete_attr, MPI_Type_free_keyval, MPI_Type_get_attr, MPI_Type_get_name, MPI_Type_set_attr, MPI_Type_set_name, |

| **TYPE_MISC** - **Miscellaneous functions for datatypes** |
|---|
| MPI_Type_c2f, MPI_Type_f2c, |

# Appendix C

# Score-P Metric Plugin Example

Simple example of a Score-P metric plugin

```
#include <scorep/SCOREP_MetricPlugins.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>


/* Maximum number of metrics */
#define NUMBER_METRICS 5

/* Maximum string length */
#define MAX_BUFFER_LENGTH 255

/* Number of individual metrics */
static int32_t num_metrics = 0;

int32_t
init()
{
    return 0;
}

int32_t
add_counter( char* eventName )
{
    int id = num_metrics;
    num_metrics++;

    return id;
}

SCOREP_Metric_Plugin_MetricProperties*
get_event_info( char* eventName )
{
    SCOREP_Metric_Plugin_MetricProperties* return_values;
    char                                   name_buffer[ MAX_BUFFER_LENGTH ];
    int                                    i;

    /* If wildcard is specified, add some default counters */
    if ( strcmp( eventName, "*" ) == 0 )
    {
        return_values = malloc( ( NUMBER_METRICS + 1 ) * sizeof(
      SCOREP_Metric_Plugin_MetricProperties ) );
        for ( i = 0; i < NUMBER_METRICS; i++ )
        {
            snprintf( name_buffer, MAX_BUFFER_LENGTH, "sync counter #%i", i );
            return_values[ i ].name        = strdup( name_buffer );
            return_values[ i ].description = NULL;
            return_values[ i ].unit        = NULL;
            return_values[ i ].mode        =
      SCOREP_METRIC_MODE_ABSOLUTE_LAST;
            return_values[ i ].value_type  =
      SCOREP_METRIC_VALUE_UINT64;
            return_values[ i ].base        = SCOREP_METRIC_BASE_DECIMAL;
            return_values[ i ].exponent    = 0;
        }
        return_values[ NUMBER_METRICS ].name = NULL;
    }
    else
    {
        /* If no wildcard is given create one counter with the passed name */
```

```
        return_values = malloc( 2 * sizeof(
      SCOREP_Metric_Plugin_MetricProperties ) );
        snprintf( name_buffer, MAX_BUFFER_LENGTH, "sync counter #%s", eventName );
        return_values[ 0 ].name        = strdup( name_buffer );
        return_values[ 0 ].description = NULL;
        return_values[ 0 ].unit        = NULL;
        return_values[ 0 ].mode        = SCOREP_METRIC_MODE_ABSOLUTE_LAST
      ;
        return_values[ 0 ].value_type  = SCOREP_METRIC_VALUE_UINT64;
        return_values[ 0 ].base        = SCOREP_METRIC_BASE_DECIMAL;
        return_values[ 0 ].exponent    = 0;
        return_values[ 1 ].name        = NULL;
    }
    return return_values;
}

bool
get_value( int32_t   counterIndex,
           uint64_t* value )
{
    *value = counterIndex;

    return true;
}

void
fini()
{
    return;
}

SCOREP_METRIC_PLUGIN_ENTRY( HelloWorld )
{
    /* Initialize info data (with zero) */
    SCOREP_Metric_Plugin_Info info;
    memset( &info, 0, sizeof( SCOREP_Metric_Plugin_Info ) );

    /* Set up */
    info.plugin_version     = SCOREP_METRIC_PLUGIN_VERSION;
    info.run_per            = SCOREP_METRIC_PER_PROCESS;
    info.sync               = SCOREP_METRIC_SYNC;
    info.initialize         = init;
    info.finalize           = fini;
    info.get_event_info     = get_event_info;
    info.add_counter        = add_counter;
    info.get_optional_value = get_value;

    return info;
}
```

### See also

SCOREP_MetricPlugins.h

# Appendix D

# Score-P Tools

## D.1 scorep

A call to `scorep` has the following syntax:

```
This is the Score-P instrumentation tool. The usage is:
scorep <options> <original command>

Common options are:
  --help, -h      Show help output. Does not execute any other command.
  --config=<file> Specifies file for the instrumentation configuration.
  -v, --verbose[=<value>] Specifies the verbosity level. The following
                  levels are available:
                  0 = No output
                  1 = Executed commands are displayed (default if no
                      value is specified)
                  2 = Detailed information is displayed
  --dry-run       Only displays the executed commands. It does not
                  execute any command.
  --keep-files    Do not delete temporarily created files after successful
                  instrumentation. By default, temporary files are deleted
                  if no error occurs during instrumentation.
  --instrument-filter=<file>
                  Specifies the filter file for filtering functions during
                  compile-time. Not supported by all instrumentation methods.
                  It applies the same syntax, as the one used by Score-P during
                  run-time.
  --version       Prints the Score-P version and exits.
  --static        Enforce static linking of the Score-P libraries.
  --dynamic       Enforce dynamic linking of the Score-P libraries.
  --no-as-needed  Adds a GNU ld linker flag to fix undefined references
                  when using shared Score-P libraries. This happens on
                  systems using --as-needed as linker default. It will
                  be handled transparently in future releases of Score-P.
  --thread=<paradigm>[:<variant>]
                  Possible paradigms and variants are:
              none
                  No thread support.
              omp:pomp_tpd
                  OpenMP support using OPARI2 thread tracking.
                  It requires and, thus, automatically enables OPARI2
                  instrumentation.
              omp:ancestry
                  OpenMP support using thread tracking with ancestry functions
                  in OpenMP 3.0 and later.
                  It requires and, thus, automatically enables OPARI2
                  instrumentation.
              pthread
                  Pthread support using thread tracking via library wrapping
                  It conflicts and, thus, automatically disables OPARI2
                  instrumentation.
  --mpp=<paradigm>[:<variant>]
                  Possible paradigms and variants are:
```

```
              none
                  No multi-process support.
              mpi
                  MPI support using library wrapping
              shmem
                  SHMEM support using library wrapping
  --mutex=<paradigm>[:<variant>]
                  Possible paradigms and variants are:
              none
                  serial case, no locking
              pthread
                  Pthread mutex locks
              pthread:spinlock
                  Pthread spinlocks
              omp
                  OpenMP locks
  --compiler      Enables compiler instrumentation.
                  By default, it disables pdt instrumentation.
  --nocompiler    Disables compiler instrumentation.
  --cuda          Enables cuda instrumentation.
  --nocuda        Disables cuda instrumentation.
  --online-access Enables online-access support. It is disabled by default
  --noonline-access Disables online-access support.
  --pomp          Enables pomp user instrumentation. By default, it also
                  enables preprocessing.
  --nopomp        Disables pomp user instrumentation. (Default)
  --openmp        Enables instrumentation of OpenMP directives. By default,
                  it also enables preprocessing. (Default for compile units
                  with enabled OpenMP support during the compilation)
  --noopenmp      Disables instrumentation of OpenMP directives.
                  Note: To ensure thread-safe execution of the measurement,
                  parallel regions still need to be tracked and will appear
                  in the results. (Default for compile units without OpenMP
                  enabled compilation)
  --opari=<parameter-list>
                  Pass options to the source-to-source instrumenter OPARI2
                  to have finer control over the instrumentation process.
                  Please refer to the OPARI2 user documentation for more
                  details.
  --pdt[=<parameter-list>]
                  Enables pdt instrumentation.
                  You may add additional parameters that are passed to pdt.
                  It requires and, thus, automatically enables user
                  instrumentation.
                  It conflicts and, thus, automatically disables preprocess
                  instrumentation.
                  By default, it disables compiler instrumentation.
  --nopdt         Disables pdt instrumentation.
  --preprocess    Enables preprocess instrumentation.
                  It cannot be enabled, if not at least one of the following is
                  enabled: OPARI2 instrumentation.
                  It conflicts and, thus, automatically disables pdt
                  instrumentation.
  --nopreprocess  Disables preprocess instrumentation.
  --user          Enables user instrumentation.
  --nouser        Disables user instrumentation.
  --opencl        Enables OpenCL instrumentation.
  --noopencl      Disables OpenCL instrumentation.
  --openacc       Enables OpenACC instrumentation.
  --noopenacc     Disables OpenACC instrumentation.
  --memory        Enables memory usage instrumentation. It is enabled by default.
  --nomemory      Disables memory usage instrumentation.
```

## D.2 scorep-config

A call to `scorep-config` has the following syntax:

```
Usage:
scorep-config <command> [<options>]
  Commands:
```

```
  --cflags    prints additional compiler flags for a C compiler. They already
              contain the include flags.
  --cxxflags  prints additional compiler flags for a C++ compiler. They already
              contain the include flags.
  --fflags    prints additional compiler flags for a Fortran compiler. They already
              contain the include flags.
  --cppflags[=language]
              prints the include flags. They are already contained in the
              output of the --cflags, --cxxflags, and --fflags commands.
              language may be one of c (default), c++, or fortran
  --ldflags   prints the library path flags for the linker
  --libs      prints the required linker flags
  --cc        prints the C compiler name
  --cxx       prints the C++ compiler name
  --fc        prints the Fortran compiler name
  --mpicc     prints the MPI C compiler name
  --mpicxx    prints the MPI C++ compiler name
  --mpifc     prints the MPI Fortran compiler name
  --help      prints this usage information
  --version   prints the version number of the Score-P package
  --scorep-revision prints the revision number of the Score-P package
  --common-revision prints the revision number of the common package
  --remap-specfile  prints the path to the remapper specification file
Options:
  --nvcc      Convert flags to be suitable for the nvcc compiler.
  --static    Use only static Score-P libraries if possible.
  --dynamic   Use only dynamic Score-P libraries if possible.
  --online-access|--noonline-access
           Specifies whether online access (needed by Periscope) is enabled.
           On default it is enabled.
  --compiler|--nocompiler
           Specifies whether compiler instrumentation is used.
           On default compiler instrumentation is enabled.
  --user|--nouser
           Specifies whether user instrumentation is used.
           On default user instrumentation is disabled.
  --pomp|--nopomp
           Specifies whether pomp instrumentation is used.
           On default pomp instrumentation is disabled.
  --cuda|--nocuda
           Specifies whether cuda instrumentation is used.
           On default cuda instrumentation is enabled.
  --openacc|--noopenacc
           Specifies whether openacc instrumentation is used.
           On default openacc instrumentation is enabled.
  --opencl|--noopencl
           Specifies whether opencl instrumentation is used.
           On default opencl instrumentation is enabled.
  --preprocess|--nopreprocess
           Specifies whether preprocess instrumentation is used.
           On default preprocess instrumentation is disabled.
  --memory=<memory-api-list>|--nomemory
           Specifies whether memory usage recording is used.
           On default memory usage recording is enabled.
           The following memory interfaces may be recorded:
            libc:
             malloc,realloc,calloc,free,memalign,posix_memalign,valloc
            libc11:
             aligned_alloc
            c++L32|c++L64:
             new,new[],delete,delete[] (IA-64 C++ ABI)
            pgCCL32|pgCCL64:
             new,new[],delete,delete[] (old PGI/EDG C++ ABI)
  --thread=<threading system>[:<variant>]
           Available threading systems are:
         none   This is the default.
         omp:pomp_tpd
         omp:ancestry
         pthread
            If no variant is specified the first matching
            threading system is used.
  --mutex=<locking system>[:<variant>]
           Available locking systems are:
```

```
        none
        omp
        pthread
        pthread:spinlock
        pthread:wrap
           If no variant is specified the default for the respective
           threading system is used.
  --mpp=<multi-process paradigm>
           Available multi-process paradigms are:
        mpi    This is the default.
        shmem
        none
```

## D.3 scorep-info

A call to `scorep-info` has the following syntax:

```
Usage: scorep-info <info command> <command options>
       scorep-info --help
This is the Score-P info tool.

Available info commands:

  config-vars:
    Shows the list of all measurement config variables with a short description.

    Info command options:
      --help        Displays a description of the Score-P measurement configuration system.
      --full        Displays a detailed description for each config variable.
      --values      Displays the current values for each config variable.
                    Warning: These values may be wrong, please consult the
                             manual of the batch system how to pass the values
                             to the measurement job.

  config-summary:
    Shows the configure summary of the Score-P package.

  open-issues:
    Shows open and known issues of the Score-P package.
```

## D.4 scorep-score

A call to `scorep-score` has the following syntax:

```
Usage: scorep-score <profile> [options]
Options:
 -r          Show all regions.
 -h, --help  Show this help and exit.
 -f <filter> Shows the result with the filter applied.
 -c <num>    Specifes the number of hardware counters that shall be measured.
             By default, this value is 0, which means that only a timestamp
             is measured on each event. If you plan to record hardware counters
             specify the number of hardware counters. Otherwise, scorep-score
             may underestimate the required space.
 -m          Prints mangled region names instead of demangled names.
```

## D.5 scorep-backend-info

**Note**

This tool is intended to run as a batch job. Please consult the manual of the batch system how to submit jobs.

A call to `scorep-backend-info` has the following syntax:

```
Usage: scorep-backend-info <info command> <command options>
       scorep-backend-info --help
This is the Score-P backend info tool.

Available info commands:

  system-tree:
    Shows the available system tree levels, starting with the root.

  config-vars:
    Shows the current values of all measurement config variables.
```

# Appendix E

# Score-P Measurement Configuration

## Introduction

Score-P allows to configure several measurement parameters via environment variables. After the measurement run you can find a 'scorep.cfg' file in your experiment directory which contains the configuration of the measurement run. If you did not set any configuration values explicitly, this file will contain the default values. This file is safe to be used as input for a POSIX shell. For example, if you want to reuse the same configuration from a previous measurement run, do something like this:

```
$ set -a
$ . scorep.cfg
$ set +a
```

Measurement configuration variables have a specific type which accepts certain values.

## Supported Types

### String

An arbitrary character sequence, no white space trimming is done.

### Path

Like String but a path is expected. Though no validation is performed.

### Boolean

A Boolean value, no white space trimming is done. Accepted Boolean values for true are case insensitive and the following:

- 'true'
- 'yes'
- 'on'
- any positive decimal number

Everything else is interpreted as the Boolean value false.

#### Number

A decimal number, white space trimming is done.

#### Number with size suffixes

Like Number, but also accepts unit case insensitive suffixes after optional white space:

- 'B', 'Kb', 'Mb', 'Gb', 'Tb', 'Pb', 'Eb'

The 'b' suffix can be omitted.

#### Set

A symbolic set. Accepted members are listed in the documentation of the variable. Multiple values are allowed, are case insensitive, and are subject to white space trimming. They can be separated with one of the following characters:

- ' ' - space
- ',' - comma
- ':' - colon
- ';' - semicolon

Acceptable values can also have aliases, which are listed in the documentation and separated by '/'.

#### Option

Like Set, but only one value allowed to be selected.

## List of Configuration Variables

This is the list of all known configure variables to control a Score-P measurement.

**Note**

Not all variables are supported by one Score-P installation. Use the `scorep-info config-vars` command to list only those supported by the used installation.

**`SCOREP_ENABLE_PROFILING`** Enable profiling

**Type:** Boolean
**Default:** true

**`SCOREP_ENABLE_TRACING`** Enable tracing

**Type:** Boolean

**Default:** false

**SCOREP_ENABLE_UNWINDING** Enables recording calling context information for every event

**Type:** Boolean

**Default:** false

The calling context is the call chain of functions to the current position in the running program. This call chain will also be annotated with source code information if possible.
This is a prerequisite for sampling but also works with instrumented applications.
Note that when tracing is also enabled, Score-P does not write the usual Enter/Leave records into the OTF2 trace, but new records.
See also SCOREP_TRACING_CONVERT_CALLING_CONTEXT_EVENTS.
Note also that this supresses events from the compiler instrumentation.

**SCOREP_VERBOSE** Be verbose

**Type:** Boolean

**Default:** false

**SCOREP_TOTAL_MEMORY** Total memory in bytes per process for the measurement system

**Type:** Number with size suffixes

**Default:** 16000k

**SCOREP_PAGE_SIZE** Memory page size in bytes

**Type:** Number with size suffixes

**Default:** 8k

TOTAL_MEMORY will be split up into pages of size PAGE_SIZE.

**SCOREP_EXPERIMENT_DIRECTORY** Name of the experiment directory

**Type:** Path

**Default:** ""

When no experiment name is given (the default) Score-P names the experiment directory 'scorep-measurement-tmp' and renames this after a successful measurement to a generated name based on the current time.

**SCOREP_OVERWRITE_EXPERIMENT_DIRECTORY** Overwrite an existing experiment directory

**Type:** Boolean

**Default:** true

If you specified a specific experiment directory name, but this name is already given, you can force overwriting it with this flag. The previous experiment directory will be renamed.

**SCOREP_MACHINE_NAME** The machine name used in profile and trace output

**Type:** String

**Default:** "Linux"

We suggest using a unique name, e.g., the fully qualified domain name. The default machine name was set at configure time (see the INSTALL file for customization options).

**SCOREP_TIMER** Timer used during measurement

**Type:** Option

**Default:** tsc

The following timers are available for this installation:

**tsc** Low overhead time stamp counter (X86_64) timer.

**gettimeofday** gettimeofday timer.

**clock_gettime** clock_gettime timer with CLOCK_MONOTONIC_RAW as clock.

**SCOREP_EXECUTABLE** Executable of the application

**Type:** Path

**Default:** ""

File name, preferably with full path, of the application's executable. It is used for evaluating the symbol table of the application, which is required by some compiler adapters.

**SCOREP_NM_SYMBOLS** Application's symbol table obtained via 'nm -l' for compiler instrumentation

**Type:** Path

**Default:** ""

File name, preferably with full path, of <file> that contains the <application>'s symbol table that was obtained by the command:

$ nm -l <application> 2> /dev/null > <file>

Only needed if generating the file at measurement initialization time fails, e.g., if using the 'system()' command from the compute nodes isn't possible.

**SCOREP_PROFILING_TASK_EXCHANGE_NUM** Number of foreign task objects that are collected before they are put into the common task object exchange buffer

**Type:** Number with size suffixes

**Default:** 1K

The profiling creates a record for every task instance that is running. To avoid locking, the required memory is taken from a preallocated memory block. Each thread has its own memory block. On task completion, the created object can be reused by other tasks. However, if tasks migrate, the data structure migrates with them. Thus, if there is an imbalance in the migration from a source thread that starts the execution of tasks towards a sink thread that completes the tasks, the source thread may continually creating new task objects while in the sink, released task objects are collected. Thus, if the sink collected a certain number of tasks it should trigger a backflow of its collected task objects. However, this requires locking which should be avoided as much as possible. Thus, we do not want the locking to happen on every migrated task, but only if a certain imbalance occurs. This environment variable determines the number of migrated task instances that must be collected before the backflow is triggered.

**SCOREP_PROFILING_MAX_CALLPATH_DEPTH** Maximum depth of the calltree

**Type:** Number

**Default:** 30

**SCOREP_PROFILING_BASE_NAME** Base for construction of the profile filename

**Type:** String

**Default:** "profile"

String which is used as based to create the filenames for the profile files.

**SCOREP_PROFILING_FORMAT** Profile output format

**Type:** Option
**Default:** default

Sets the output format for the profile.

The following formats are supported:

**none** No profile output. This does not disable profile recording.
**tau_snapshot** Tau snapshot format.
**cube4** Stores the sum for every metric per callpath in Cube4 format.
**cube_tuple** Stores an extended set of statistics in Cube4 format.
**default** Default format. If Cube4 is supported, Cube4 is the default else the Tau snapshot format is default.

**SCOREP_PROFILING_ENABLE_CLUSTERING** Enable clustering

**Type:** Boolean
**Default:** true

**SCOREP_PROFILING_CLUSTER_COUNT** Maximum cluster count for iteration clustering

**Type:** Number with size suffixes
**Default:** 64

**SCOREP_PROFILING_CLUSTERING_MODE** Specifies the level of strictness when comparing call trees for equivalence

**Type:** Option
**Default:** subtree

Possible levels:

**none/0** No structural similarity required.
**subtree/1** The sub-trees structure must match.

**subtree_visits/2** The sub-trees structure and the number of visits must match.

**mpi/3** The structure of the call-path to MPI calls must match.
Nodes that are not on an MPI call-path may differ.

**mpi_visits/4** Like above, but the number of visits of the MPI calls must match, too.

**mpi_visits_all/5** Like above, but the number of visits must match also match on all nodes on the call-path to an MPI function.

**SCOREP_PROFILING_CLUSTERED_REGION** Name of the clustered region

**Type:** String

**Default:** ""

The clustering can only cluster one dynamic region. If more than one dynamic region are defined by the user, the region is clustered which is exited first. If another region should be clustered instead you can specify the region name in this variable. If the variable is unset or empty, the first exited dynamic region is clustered.

**SCOREP_PROFILING_ENABLE_CORE_FILES** Write .core files if an error occurred

**Type:** Boolean

**Default:** false

If an error occures inside the profiling system, the profiling is disabled. For debugging reasons, it might be feasible to get the state of the local stack at these points. It is not recommended to enable this feature for large scale measurements.

**SCOREP_TRACING_USE_SION** Whether or not to use libsion as OTF2 substrate

**Type:** Boolean

**Default:** false

**SCOREP_TRACING_MAX_PROCS_PER_SION_FILE** Maximum number of processes that share one sion file (must be $> 0$)

**Type:** Number with size suffixes

**Default:** 1K

All processes are than evenly distributed over the number of needed files to fulfill this constraint. E.g., having 4 processes and setting the maximum to 3 would result in 2 files each holding 2 processes.

**SCOREP_TRACING_COMPRESS** Whether or not to compress traces with libz

**Type:** Boolean

**Default:** false

**SCOREP_TRACING_CONVERT_CALLING_CONTEXT_EVENTS** Write calling context information as a sequence of Enter/Leave events to trace

**Type:** Boolean

**Default:** false

When recording the calling context of events (instrumented or sampled) than these could be stored in the trace either as the new CallingContext records from OTF2 or they could be converted to the legacy Enter/↩ Leave records. This can be controlled with this variable, where the former is the false value.
This is only in effect if SCOREP_ENABLING_UNWINDING is on.
Note that enabling this will result in an increase of records per event and also of the loss of the source code locations.
This option exists only for backwards compatibility for tools, which cannot handle the new OTF2 records. This option my thus be removed in future releases.

**SCOREP_ONLINEACCESS_ENABLE** Enable online access interface

**Type:** Boolean

**Default:** false

**SCOREP_ONLINEACCESS_REG_PORT** Online access registry service port

**Type:** Number

**Default:** 50100

**SCOREP_ONLINEACCESS_REG_HOST** Online access registry service hostname

**Type:** String

**Default:** "localhost"

**SCOREP_ONLINEACCESS_BASE_PORT** Base port for online access server

**Type:** Number
**Default:** 50010

**SCOREP_ONLINEACCESS_APPL_NAME** Application name to be registered

**Type:** String
**Default:** "appl"

**SCOREP_FILTERING_FILE** A file name which contain the filter rules

**Type:** Path
**Default:** ""

**SCOREP_METRIC_PAPI** PAPI metric names to measure

**Type:** String
**Default:** ""

List of requested PAPI metric names that will be collected during program run.

**SCOREP_METRIC_PAPI_PER_PROCESS** PAPI metric names to measure per-process

**Type:** String
**Default:** ""

List of requested PAPI metric names that will be recorded only by first thread of a process.

**SCOREP_METRIC_PAPI_SEP** Separator of PAPI metric names

**Type:** String
**Default:** ","

Character that separates metric names in SCOREP_METRIC_PAPI and SCOREP_METRIC_PAPI_PER_↩ PROCESS.

**SCOREP_METRIC_RUSAGE** Resource usage metric names to measure

**Type:** String

**Default:** ""

List of requested resource usage metric names that will be collected during program run.

**SCOREP_METRIC_RUSAGE_PER_PROCESS** Resource usage metric names to measure per-process

**Type:** String

**Default:** ""

List of requested resource usage metric names that will be recorded only by first thread of a process.

**SCOREP_METRIC_RUSAGE_SEP** Separator of resource usage metric names

**Type:** String

**Default:** ","

Character that separates metric names in SCOREP_METRIC_RUSAGE and SCOREP_METRIC_RUSAG↩ E_PER_PROCESS.

**SCOREP_METRIC_PLUGINS** Specify list of used plugins

**Type:** String

**Default:** ""

List of requested metric plugin names that will be used during program run.

**SCOREP_METRIC_PLUGINS_SEP** Separator of plugin names

**Type:** String

**Default:** ","

Character that separates plugin names in SCOREP_METRIC_PLUGINS.

**SCOREP_METRIC_PERF** PERF metric names to measure

**Type:** String

**Default:** ""

List of requested PERF metric names that will be collected during program run.

**SCOREP_METRIC_PERF_PER_PROCESS** PERF metric names to measure per-process

**Type:** String

**Default:** ""

List of requested PERF metric names that will be recorded only by first thread of a process.

**SCOREP_METRIC_PERF_SEP** Separator of PERF metric names

**Type:** String

**Default:** ","

Character that separates metric names in SCOREP_METRIC_PERF and SCOREP_METRIC_PERF_PE↩
R_PROCESS.

**SCOREP_SAMPLING_EVENTS** Set the sampling event and period: <event>[<period>]

**Type:** String

**Default:** "perf_cycles@10000000"

This selects the interrupt source for sampling.
This is only in effect if SCOREP_ENABLE_UNWINDING is on.

Possible values:
- perf event (perf_<event>, see "perf list")
period in number of events, default: 10000000
e.g., perf_cycles@2000000
- timer (POSIX timer, invalid for multi-threaded)
period in us, default: 10000
e.g., timer@2000

**SCOREP_SAMPLING_SEP** Separator of sampling event names

**Type:** String
**Default:** ","

Character that separates sampling event names in SCOREP_SAMPLING_EVENTS

**SCOREP_SELECTIVE_CONFIG_FILE** A file name which configures selective recording

**Type:** Path
**Default:** ""

**SCOREP_MPI_MAX_COMMUNICATORS** Determines the number of concurrently used communicators per process

**Type:** Number
**Default:** 50

**SCOREP_MPI_MAX_WINDOWS** Determines the number of concurrently used windows for MPI one-sided communication per process

**Type:** Number
**Default:** 50

**SCOREP_MPI_MAX_ACCESS_EPOCHS** Maximum amount of concurrently active access or exposure epochs per process

**Type:** Number
**Default:** 50

**SCOREP_MPI_MAX_GROUPS** Maximum number of concurrently used MPI groups per process

**Type:** Number

**Default:** 50

**SCOREP_MPI_ENABLE_GROUPS** The names of the function groups which are measured

**Type:** Set
**Default:** default

Other functions are not measured.

Possible groups are:

**all** All MPI functions
**cg** Communicator and group management
**coll** Collective functions
**default** Default configuration
**env** Environmental management
**err** MPI Error handling
**ext** External interface functions
**io** MPI file I/O
**p2p** Peer-to-peer communication
**misc** Miscellaneous
**perf** PControl
**rma** One sided communication
**spawn** Process management
**topo** Topology
**type** MPI datatype functions
**xnonblock** Extended non-blocking events
**xreqtest** Test events for uncompleted requests
**none/no** Disable feature

**SCOREP_MPI_MEMORY_RECORDING** Enable tracking of memory allocations done by calls to MPI_ALLOC_↩ MEM and MPI_FREE_MEM

**Type:** Boolean
**Default:** false

Requires that the MISC group is also recorded.

**SCOREP_MPI_ONLINE_ANALYSIS** Enable online mpi wait states analysis

**Type:** Boolean

**Default:** false

**SCOREP_SHMEM_MEMORY_RECORDING** Enable tracking of memory allocations done by calls to the SHMEM allocation API

**Type:** Boolean

**Default:** false

**SCOREP_CUDA_ENABLE** CUDA measurement features

**Type:** Set

**Default:** no

Sets the CUDA measurement mode to capture:

**runtime** CUDA runtime API

**driver** CUDA driver API

**kernel** CUDA kernels

**kernel_serial** Serialized kernel recording

**kernel_counter** Fixed CUDA kernel metrics

**memcpy** CUDA memory copies

**sync** Record implicit and explicit CUDA synchronization

**idle** GPU compute idle time

**pure_idle** GPU idle time (memory copies are not idle)

**gpumemusage** Record CUDA memory (de)allocations as a counter

**references** Record references between CUDA activities

**flushatexit** Flush CUDA activity buffer at program exit

**default/yes/1** CUDA runtime API and GPU activities

**none/no** Disable feature

**SCOREP_CUDA_BUFFER** Total memory in bytes for the CUDA record buffer

**Type:** Number with size suffixes

**Default:** 1M

**SCOREP_CUDA_BUFFER_CHUNK** Chunk size in bytes for the CUDA record buffer (ignored for CUDA 5.5 and earlier)

**Type:** Number with size suffixes

**Default:** 8k

**SCOREP_OPENCL_ENABLE** OpenCL measurement features

**Type:** Set

**Default:** no

Sets the OpenCL measurement mode to capture:

**api** OpenCL runtime API

**kernel** OpenCL kernels

**memcpy** OpenCL buffer reads/writes

**default/yes/true/1** OpenCL API and GPU activities

**none/no** Disable feature

**SCOREP_OPENCL_BUFFER_QUEUE** Memory in bytes for the OpenCL command queue buffer

**Type:** Number with size suffixes

**Default:** 8k

**SCOREP_OPENACC_ENABLE** OpenACC measurement features

**Type:** Set

**Default:** no

Sets the OpenACC measurement mode to capture:

**regions** OpenACC regions

**wait** OpenACC wait operations

**enqueue** OpenACC enqueue operations (kernel, upload, download)

**device_alloc** OpenACC device memory allocations

**kernel_properties** Record kernel properties such as the kernel name as well as the gang, worker and vector size for kernel launch operations

**variable_names** Record variable names for OpenACC data allocation and enqueue upload/download

**default/yes/1** OpenACC regions, enqueue and wait operations

**none/no** Disable feature

**SCOREP_MEMORY_RECORDING** Memory recording

**Type:** Boolean

**Default:** false

Memory (de)allocations are recorded via the libc/C++ API.

# Appendix F

# Score-P wrapper usage

## Usage

```
scorep-wrapper --create COMPILER [BINDIR]
scorep-<compiler> [COMPILER_FLAGS...]
```

## Description

The `scorep-wrapper` script instances (like `scorep-gcc`, see below for a list of provided instances) are intended to simplify configuration and instrumentation of applications where the usual means of instrumentation, i.e., prefixing the compilation command with `scorep`, does not work out of the box. This applies, e.g., to applications that use autotools or CMake.

The intended usage of the wrapper instances is to replace the application's compiler and linker with the corresponding wrapper at configuration time so that they will be used at build time. As compiler and linker commands are usually assigned to build variables like `CC`, `CXX`, or `F77` (e.g., `CC=gcc`), using the corresponding wrapper is as simple as prefixing the value with `scorep-` (e.g., `CC=scorep-gcc`).

E.g., say the original compile command is

```
$ gcc COMPILER_FLAGS...
```

Using the wrapper instead

```
$ scorep-gcc COMPILER_FLAGS...
```

will expand to the following call:

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

Used at build time, this expansion performs the desired Score-P instrumentation.

The variables `SCOREP_WRAPPER_INSTRUMENTER_FLAGS` and `SCOREP_WRAPPER_COMPILER_FLAGS` can be used to pass extra arguments to `scorep` and to the compiler, respectively. Please see the *Examples* section below for details.

If the application's build system includes a configuration step (like it is the case for autotools and CMake based projects) the expansion needs to be prevented at this stage (the configuration step compiles and runs lots of small test programs; instrumenting these would in almost all cases result in failure of the configuration). To do so, one needs to set the variable `SCOREP_WRAPPER` to `off` when invoking the configuration command. The wrapper command from above will than expand to the original compile command:

```
$ gcc COMPILER_FLAGS...
```

See also the EXAMPLES section below.

## Wrapper Instances

The installation provides wrapper instances based on the compilers used to build Score-P. Run `scorep-wrapper --help` to see a listing of all default instances of the used Score-P installation.

Additional wrapper instances can be created with `scorep-wrapper --create`.

## Examples

- The most prominent use case is the CMake build system. As CMake prohibits to change the compiler after the `CMake` step and it also prohibits that the compiler variable value includes any flags (which renders the usual prefixing `scorep gcc` to a non-functional value). One needs to use a wrapper script which introduces the Score-P instrumenter as a compiler replacement to CMake as early as possible so that they are hard-coded into the generated build files. Apart from that one needs to make sure that the wrappers don't perform their usual instrumentation at this early stage or else the configuration step is likely to fail. However, at make time we want the wrappers to do the actual instrumentation. These goals can be achieved by invoking `cmake` like follows:

  ```
  $ SCOREP_WRAPPER=off cmake .. \
      -DCMAKE_C_COMPILER=scorep-gcc \
      -DCMAKE_CXX_COMPILER=scorep-g++
  ```

  The `SCOREP_WRAPPER=off` disables the instrumentation only in the environment of the `cmake` command. Subsequent calls to `make` are not affected and will instrument the application as expected.

- For autotools based build systems it is recommended to configure in the following way:

  ```
  $ SCOREP_WRAPPER=off ../configure \
      CC=scorep-gcc \
      CXX=scorep-g++ \
      --disable-dependency-tracking
  ```

- Both autoconf and CMake based build systems, may automatically re-configure the build tree when calling `make`, because some build related files have changed (i.e., `Makefile.am` or `CMakeLists.txt` files). This usage is not supported by the Score-P wrapper. Please re-start the configuration from an empty build directory again as described above.

- To pass options to the `scorep` command in order to diverge from the default instrumentation or to activate verbose output, use the variable `SCOREP_WRAPPER_INSTRUMENTER_FLAGS` at make time:

  ```
  $ make SCOREP_WRAPPER_INSTRUMENTER_FLAGS=--verbose
  ```

  This will result in the execution of:

  ```
  $ scorep --verbose gcc ...
  ```

- The wrapper also allows to pass flags to the wrapped compiler call by using the variable `SCOREP_WRAP↩PER_COMPILER_FLAGS`:

  ```
  $ make SCOREP_WRAPPER_COMPILER_FLAGS="-D_GNU_SOURCE"
  ```

  Will result in the execution of:

  ```
  $ scorep gcc -D_GNU_SOURCE ...
  ```

- If there is a need to create additional wrapper instances, e.g., if your build system already uses compiler wrappers, you can do so by calling the `scorep-wrapper` script with the `--create` option:

  ```
  $ scorep-wrapper --create cc
  ```

  This will create a new wrapper instance for the `cc` compiler named `scorep-cc` in the same directory where `scorep-wrapper` resides.

# Appendix G

# Module Documentation

## G.1 Score-P User Adapter

### Files

- file SCOREP_User.h

  *This file contains the interface for the manual user instrumentation.*
- file SCOREP_User_Types.h

  *This file contains type definitions for manual user instrumentation.*

### Macros for region instrumentation

- #define SCOREP_USER_OA_PHASE_BEGIN(handle, name, type)
- #define SCOREP_USER_OA_PHASE_END(handle) SCOREP_User_OaPhaseEnd( handle );
- #define SCOREP_USER_REGION_BEGIN(handle, name, type)
- #define SCOREP_USER_REGION_INIT(handle, name, type)
- #define SCOREP_USER_REGION_END(handle) SCOREP_User_RegionEnd( handle );
- #define SCOREP_USER_REGION_ENTER(handle) SCOREP_User_RegionEnter( handle );
- #define SCOREP_USER_REGION_DEFINE(handle) static SCOREP_User_RegionHandle handle = SCO↩ REP_USER_INVALID_REGION;
- #define SCOREP_USER_FUNC_DEFINE()
- #define SCOREP_USER_FUNC_BEGIN()
- #define SCOREP_USER_FUNC_END() SCOREP_User_RegionEnd( scorep_user_func_handle );
- #define SCOREP_USER_GLOBAL_REGION_DEFINE(handle) SCOREP_User_RegionHandle handle = SCOREP_USER_INVALID_REGION;
- #define SCOREP_USER_GLOBAL_REGION_EXTERNAL(handle) extern SCOREP_User_RegionHandle handle;

### Macros for parameter instrumentation

- #define SCOREP_USER_PARAMETER_INT64(name, value)
- #define SCOREP_USER_PARAMETER_UINT64(name, value)
- #define SCOREP_USER_PARAMETER_STRING(name, value)

### Macros to provide user metrics

- #define SCOREP_USER_METRIC_LOCAL(metricHandle)
- #define SCOREP_USER_METRIC_GLOBAL(metricHandle)

- #define SCOREP_USER_METRIC_EXTERNAL(metricHandle) extern SCOREP_SamplingSetHandle metricHandle;
- #define SCOREP_USER_METRIC_INIT(metricHandle, name, unit, type, context) SCOREP_User_InitMetric( &metricHandle, name, unit, type, context );
- #define SCOREP_USER_METRIC_INT64(metricHandle, value)
- #define SCOREP_USER_METRIC_UINT64(metricHandle, value)
- #define SCOREP_USER_METRIC_DOUBLE(metricHandle, value)

#### C++ specific macros for region instrumentation

- #define SCOREP_USER_REGION(name, type)

#### Macros for measurement control

- #define SCOREP_RECORDING_ON() SCOREP_User_EnableRecording();
- #define SCOREP_RECORDING_OFF() SCOREP_User_DisableRecording();
- #define SCOREP_RECORDING_IS_ON() SCOREP_User_RecordingEnabled()

#### Region types

- #define SCOREP_USER_REGION_TYPE_COMMON 0
- #define SCOREP_USER_REGION_TYPE_FUNCTION 1
- #define SCOREP_USER_REGION_TYPE_LOOP 2
- #define SCOREP_USER_REGION_TYPE_DYNAMIC 4
- #define SCOREP_USER_REGION_TYPE_PHASE 8

#### Metric types

- #define SCOREP_USER_METRIC_TYPE_INT64 0
- #define SCOREP_USER_METRIC_TYPE_UINT64 1
- #define SCOREP_USER_METRIC_TYPE_DOUBLE 2

#### Metric contexts

- #define SCOREP_USER_METRIC_CONTEXT_GLOBAL 0
- #define SCOREP_USER_METRIC_CONTEXT_CALLPATH 1

### G.1.1 Detailed Description

The user adapter provides a set of macros for user manual instrumentation. The macros are inserted in the source code and call functions of the Score-P runtime system. The user should avoid calling the Score-P runtime functions directly.

For every macro, two definitions are provided: The first one inserts calls to the Score-P runtime system, the second definitions resolve to nothing. Which implementation is used, depends on the definition of SCOREP_USER↩ _ENABLE. If SCOREP_USER_ENABLE is defined, the macros resolve to calls to the Score-P runtime system. If SCOREP_USER_ENABLE is undefined, the user instrumentation is removed by the preprocessor. This flag SCOREP_USER_ENABLE should be set through the instrumentation wrapper tool automatically if user manual instrumentation is enabled.

Every source file which is instrumented must include a header file with the Score-P user instrumentation header. For C/C++ programs, the header file is 'scorep/SCOREP_User.h', for Fortran files, 'scorep/SCOREP_User.inc' must be included. Because the Fortran compilers cannot expand macros, the Fortran source code must be preprocessed

by a C or C++ preprocessor, to include the headers and expand the macros. Which Fortran files are passed to the preprocessor depends on the suffix. Usually, suffixes .f and .f90 are not preprocessed, .F and .F90 files are preprocessed. However, this may depend on the used compiler.

### G.1.2 Macro Definition Documentation

#### G.1.2.1 #define SCOREP_RECORDING_IS_ON( ) SCOREP_User_RecordingEnabled()

In C/C++ it behaves like a function call which returns whether recording is enabled or not. It returns false if the recording of events is disabled, else it returns true.

C/C++ example:

```
1 void foo()
2 {
3   if ( SCOREP_RECORDING_IS_ON() )
4   {
5     // do something
6   }
7 }
```

In Fortran, this macro has a different syntax. An integer variable must be specified as parameter, which is set to non-zero if recording is enabled, else the value is set to zero.

Fortran example:

```
1 subroutine foo
2   integer :: l
3
4   SCOREP_RECORDING_IS_ON( l )
5   if (l .eq. 0) then
6     ! do something
7   end if
8
9 end subroutine foo
```

#### G.1.2.2 #define SCOREP_RECORDING_OFF( ) SCOREP_User_DisableRecording();

Disables recording of events. If already disabled, this command has no effect. The control is not restricted to events from the user adapter, but disables the recording of all events.

C/C++ example:

```
1 void foo()
2 {
3   SCOREP_RECORDING_OFF()
4
5   // do something
6
7  SCOREP_RECORDING_ON()
8 }
```

Fortran example:

```
1 subroutine foo
2
3   SCOREP_RECORDING_OFF()
4 ! do something
5   SCOREP_RECORDING_ON()
6
7 end subroutine foo
```

#### G.1.2.3 #define SCOREP_RECORDING_ON( ) SCOREP_User_EnableRecording();

Enables recording of events. If already enabled, this command has no effect. The control is not restricted to events from the user adapter, but enables the recording of all events.

C/C++ example:

```
1 void foo()
2 {
3   SCOREP_RECORDING_OFF()
4
5   // do something
6
7  SCOREP_RECORDING_ON()
8 }
```

Fortran example:

```
1 subroutine foo
2
3   SCOREP_RECORDING_OFF()
4 ! do something
5   SCOREP_RECORDING_ON()
6
7 end subroutine foo
```

#### G.1.2.4 #define SCOREP_USER_FUNC_BEGIN( )

**Value:**

```
static SCOREP_User_RegionHandle        \
    scorep_user_func_handle =  SCOREP_USER_INVALID_REGION; \
    SCOREP_User_RegionBegin( &scorep_user_func_handle, &SCOREP_User_LastFileName, \
                             &SCOREP_User_LastFileHandle, SCOREP_USER_FUNCTION_NAME, \
                             SCOREP_USER_REGION_TYPE_FUNCTION, __FILE__,
      __LINE__ );
```

This macro marks the start of a function. It should be inserted at the beginning of the instrumented function. It will generate a region, with the function name.

The C/C++ version of this command takes no arguments. It contains a variable declaration and a function call. Compilers that require a strict separation between declaration block and execution block may fail if this macro is used.

In Fortran one argument is required for the name of the function. Furthermore, the handle must be declared explicitly in Fortran.

**Parameters**

| | |
|---|---|
| *name* | Fortan only: A string containing the name of the function. |

C/C++ example:

```
1 void myfunc()
2 {
3   // declarations
4
5   SCOREP_USER_FUNC_BEGIN()
6
7   // do something
8
9   SCOREP_USER_FUNC_END()
10 }
```

Fortran example:

```
1 subroutine myfunc
2   SCOREP_USER_FUNC_DEFINE()
3   ! more declarations
4
```

```
5   SCOREP_USER_FUNC_BEGIN( "myfunc" )
6   ! do something
7   SCOREP_USER_FUNC_END()
8
9 end subroutine myfunc
```

Note that in Fortran the function need to be declared using SCOREP_USER_FUNC_DEFINE before.

#### G.1.2.5 #define SCOREP_USER_FUNC_DEFINE( )

This macro is for Fortran only. It declares the handle for a function. Every function handle must be declared in the declaration part of the subroutine or function if the SCOREP_USER_FUNC_BEGIN and SCOREP_USER_FUN↩ C_END macros are used.

Example:

```
1 subroutine myfunc
2   SCOREP_USER_FUNC_DEFINE()
3   ! more declarations
4
5   SCOREP_USER_FUNC_BEGIN( "myfunc" )
6   ! do something
7   SCOREP_USER_FUNC_END()
8
9 end subroutine myfunc
```

Note that in Fortran the function need to be declared using SCOREP_USER_FUNC_DEFINE before.

#### G.1.2.6 #define SCOREP_USER_FUNC_END( ) SCOREP_User_RegionEnd( scorep_user_func_handle );

This macro marks the end of a function. It should be inserted at every return point of the instrumented function.

C/C++ example:

```
1 void myfunc()
2 {
3   // declarations
4
5   SCOREP_USER_FUNC_BEGIN()
6
7   // do something
8   if ( some_expression )
9   {
10     SCOREP_USER_FUNC_END()
11     return;
12   }
13
14   SCOREP_USER_FUNC_END()
15 }
```

Fortran example:

```
1 subroutine myfunc
2   SCOREP_USER_FUNC_DEFINE()
3   ! more declarations
4
5   SCOREP_USER_FUNC_BEGIN( "myfunc" )
6   ! do something
7   SCOREP_USER_FUNC_END()
8
9 end subroutine myfunc
```

Note that in Fortran the function need to be declared using SCOREP_USER_FUNC_DEFINE before.

#### G.1.2.7 #define SCOREP_USER_GLOBAL_REGION_DEFINE( *handle* ) SCOREP_User_RegionHandle handle = SCOREP_USER_INVALID_REGION;

This macro defines a region handle in a global scope for usage in more than one code block. If a region is used in multiple source files, only one of them must contain the definition using SCOREP_USER_GLOBAL_REGION↩

_DEFINE. All other files, in which the global handle is accessed, must only declare the global handle with SCOREP_USER_GLOBAL_REGION_EXTERNAL( handle ). It is possible to use the global handle in more than one code-block. However, code-blocks that share a handle, are handled as they were all the same region. Enter and exit events for global regions are created with SCOREP_USER_REGION_BEGIN and SCOREP_USER_REGION_END, respectively. Its name and type is determined at the first enter event and is not changed on later events, even if other code blocks contains a different name or type in their SCOREP_USER_REGION_BEGIN statement.

This macro is not available in Fortran.

**Parameters**

| | |
|---|---|
| *handle* | A unique name for the handle must be provided. This handle is declared in the macro. This handle is used in the SCOREP_USER_REGION_BEGIN and SCOREP_USER_REGION_END statements to specify which region is started, or ended. If you are using a Fortran version which has a limited length of code lines, the length of the *handle* parameter must be at most 4 characters, else the declaration line exceeds the allowed length. |

C/C++ example:

```
1 // In File1:
2 SCOREP_USER_GLOBAL_REGION_DEFINE( my_global_handle )
3
4 void myfunc()
5 {
6   SCOREP_USER_REGION_BEGIN( my_global_handle, "my_global", SCOREP_USER_REGION_TYPE_PHASE )
7
8   // do something
9
10   SCOREP_USER_REGION_END( my_global_handle )
11 }
```

```
1 // In File2:
2 SCOREP_USER_GLOBAL_EXTERNAL( my_global_handle )
3
4 void foo()
5 {
6   SCOREP_USER_REGION_BEGIN( my_global_handle, "my_global", SCOREP_USER_REGION_TYPE_PHASE )
7
8   // do something
9
10   SCOREP_USER_REGION_END( my_global_handle )
11 }
```

#### G.1.2.8 #define SCOREP_USER_GLOBAL_REGION_EXTERNAL( *handle* ) extern SCOREP_User_RegionHandle handle;

This macro declares an externally defined global region. If a region is used in multiple source files, only one of them must contain the definition using SCOREP_USER_GLOBAL_REGION_DEFINE. All other files, in which the global handle is accessed, must only declare the global handle with SCOREP_USER_GLOBAL_REGION_EXTERNAL( handle ). It is possible to use the global handle in more than one code-block. However, code-blocks that share a handle, are handled as they were all the same region. Enter and exit events for global regions are created with SCOREP_USER_REGION_BEGIN and SCOREP_USER_REGION_END, respectively. Its name and type is determined at the first enter event and is not changed on later events, even if other code blocks contains a different name or type in their SCOREP_USER_REGION_BEGIN statement.

This macro is not available in Fortran

**Parameters**

| | |
|---|---|
| *handle* | A name for a variable must be provided. This variable name must be the same like for the corresponding SCOREP_USER_GLOBAL_REGION_DEFINE statement. The handle is used in the SCOREP_USER_REGION_BEGIN and SCOREP_USER_REGION_END statements to specify which region is started, or ended. |

C/C++ example:

```
1 // In File 1
2 SCOREP_USER_GLOBAL_REGION_DEFINE( my_global_handle )
3
4 void myfunc()
5 {
```

```
6   SCOREP_USER_REGION_BEGIN( my_global_handle, "my_global", SCOREP_USER_REGION_TYPE_PHASE )
7
8   // do something
9
10   SCOREP_USER_REGION_END( my_global_handle )
11 }
```

```
1 // In File 2
2 SCOREP_USER_GLOBAL_EXTERNAL( my_global_handle )
3
4 void foo()
5 {
6   SCOREP_USER_REGION_BEGIN( my_global_handle, "my_global", SCOREP_USER_REGION_TYPE_PHASE )
7
8   // do something
9
10   SCOREP_USER_REGION_END( my_global_handle )
11 }
```

#### G.1.2.9 #define SCOREP_USER_METRIC_CONTEXT_CALLPATH 1

Indicates that a user counter is is measured for every callpath.

#### G.1.2.10 #define SCOREP_USER_METRIC_CONTEXT_GLOBAL 0

Indicates that a user counter is is measured for the global context.

#### G.1.2.11 #define SCOREP_USER_METRIC_DOUBLE( *metricHandle, value* )

**Value:**

```
SCOREP_User_TriggerMetricDouble( \
        metricHandle, value );
```

Triggers a new event for a user counter of a double precision floating point data type. Each user metric must be declared with SCOREP_USER_COUNTER_LOCAL, SCOREP_USER_COUNTER_GLOBAL, or SCOREP_USE↩R_COUNTER_EXTERNAL and initialized with SCOREP_USER_COUNTER_INIT before it is triggered for the first time.

**Parameters**

| | |
|---|---|
| *metricHandle* | The handle of the metric for which a value is given in this statement. |
| *value* | The value of the counter. It must be possible for implicit casts to cast it to a double. |

Example:

```
1 SCOREP_USER_METRIC_LOCAL( my_local_metric )
2
3 int main()
4 {
5   SCOREP_USER_METRIC_INIT( my_local_metric, "My Metric", "seconds", \
6                            SCOREP_USER_METRIC_TYPE_DOUBLE, \
7                            SCOREP_USER_METRIC_CONTEXT_GLOBAL )
8   // do something
9 }
10
11 void foo()
12 {
13   double my_double = get_some_double_value();
14   SCOREP_USER_METRIC_DOUBLE( my_local_metric, my_double )
15 }
```

Fortran example:

```
1 program myProg
2   SCOREP_USER_METRIC_LOCAL( my_local_metric )
3   real (kind=selected_int_kind(14,200)):: my_real = 24.5
4 ! more declarations
```

```
5
6
7   SCOREP_USER_METRIC_INIT( my_local_metric, "My Metric", "seconds", &
8                            SCOREP_USER_METRIC_TYPE_DOUBLE, &
9                            SCOREP_USER_METRIC_CONTEXT_GLOBAL )
10
11 ! do something
12
13   SCOREP_USER_METRIC_DOUBLE( my_local_metric, my_real )
14 end program myProg
```

#### G.1.2.12 #define SCOREP_USER_METRIC_EXTERNAL( *metricHandle* ) extern SCOREP_SamplingSetHandle metricHandle;

Declares an externally defined handle for a user metric. Every global metric must be declared only in one file using SCOREP_USER_METRIC_GLOBAL. All other files in which this handle is accessed must declare it with SCOREP_USER_METRIC_EXTERNAL.

This macro is not available in Fortran.

**Parameters**

| | |
|---|---|
| *metricHandle* | The variable name of the handle. it must be the same name as used in the corresponding SCOREP_USER_METRIC_GLOBAL statement. |

C/C++ example:

```
1 // In File 1
2 SCOREP_USER_METRIC_GLOBAL( my_global_metric )
3
4 int main()
5 {
6   SCOREP_USER_METRIC_INIT( my_global_metric, "My Global Metric", "seconds", \
7                            SCOREP_USER_METRIC_TYPE_UINT64, \
8                            SCOREP_USER_METRIC_CONTEXT_GLOBAL )
9   // do something
10 }
11
12 void foo()
13 {
14   uint64 my_int = get_some_int_value();
15   SCOREP_USER_METRIC_UINT64( my_global_metric, my_int )
16 }
```

```
1 // In File 2
2 SCOREP_USER_METRIC_EXTERNAL( my_global_metric )
3
4 void bar()
5 {
6   uint64 my_int = get_some_int_value();
7   SCOREP_USER_METRIC_UINT64( my_global_metric, my_int )
8 }
```

#### G.1.2.13 #define SCOREP_USER_METRIC_GLOBAL( *metricHandle* )

**Value:**

```
SCOREP_SamplingSetHandle metricHandle \
        = SCOREP_INVALID_SAMPLING_SET;
```

Declares a handle for a user metric as a global variable. It must be used if a metric handle is accessed in more than one file. Every global metric must be declared only in one file using SCOREP_USER_METRIC_GLOBAL. All other files in which this handle is accessed must declare it with SCOREP_USER_METRIC_EXTERNAL.

This macro is not available in Fortran.

**Parameters**

| | |
|---|---|
| *metricHandle* | The variable name for the handle. If you are using a Fortran version which has a limited length of code lines, the length of the *handle* parameter must be at most 4 characters, else the declaration line exceeds the allowed length. |

C/C++ example:

```
1 // In File 1
2 SCOREP_USER_METRIC_GLOBAL( my_global_metric )
3
4 int main()
5 {
6   SCOREP_USER_METRIC_INIT( my_global_metric, "My Global Metric", "seconds", \
7                            SCOREP_USER_METRIC_TYPE_UINT64, \
8                            SCOREP_USER_METRIC_CONTEXT_GLOBAL )
9   // do something
10 }
11
12 void foo()
13 {
14   uint64 my_int = get_some_int_value();
15   SCOREP_USER_METRIC_UINT64( my_global_metric, my_int )
16 }
```

```
1 // In File 2
2 SCOREP_USER_METRIC_EXTERNAL( my_global_metric )
3
4 void bar()
5 {
6   uint64 my_int = get_some_int_value();
7   SCOREP_USER_METRIC_UINT64( my_global_metric, my_int )
8 }
```

#### G.1.2.14 #define SCOREP_USER_METRIC_INIT( *metricHandle, name, unit, type, context* ) SCOREP_User_InitMetric( &metricHandle, name, unit, type, context );

Initializes a new user counter. Each counter must be initialized before it is triggered the first time. The handle must be declared using SCOREP_USER_METRIC_LOCAL, SCOREP_USER_METRIC_GLOBAL, or SCOREP_USE↩R_METRIC_EXTERNAL.

**Parameters**

| | |
|---|---|
| *metricHandle* | Provides a variable name of the variable to store the metric handle. The variable is declared by the macro. |
| *name* | A string containing a unique name for the counter. |
| *unit* | A string containing a the unit of the data. |
| *type* | Specifies the data type of the counter. It must be one of the following: SCOREP_USER_↩METRIC_TYPE_INT64, SCOREP_USER_METRIC_TYPE_UINT64, SCOREP_USER_M↩ETRIC_TYPE_DOUBLE. In Fortran is SCOREP_USER_METRIC_TYPE_UINT64 not available. |
| *context* | Specifies the context for which the counter is measured. IT must be one of the following: SCOREP_USER_METRIC_CONTEXT_GLOBAL, or SCOREP_USER_METRIC_CONTE↩XT_CALLPATH. |

C/C++ example:

```
1 SCOREP_USER_METRIC_LOCAL( my_local_metric )
2
3 int main()
4 {
5   SCOREP_USER_METRIC_INIT( my_local_metric, "My Metric", "seconds", \
6                            SCOREP_USER_METRIC_TYPE_UINT64, \
7                            SCOREP_USER_METRIC_CONTEXT_GLOBAL )
8   // do something
9 }
10
11 void foo()
12 {
13   uint64 my_int = get_some_int_value();
14   SCOREP_USER_METRIC_UINT64( my_local_metric, my_int )
15 }
```

Fortran example:

```
1 program myProg
2   SCOREP_USER_METRIC_LOCAL( my_local_metric )
3   integer (kind=selected_int_kind(8)):: my_int = 19
4 ! more declarations
5
6
7   SCOREP_USER_METRIC_INIT( my_local_metric, "My Metric", "seconds", &
8                            SCOREP_USER_METRIC_TYPE_INT64, &
9                            SCOREP_USER_METRIC_CONTEXT_GLOBAL )
10
11 ! do something
12
13   SCOREP_USER_METRIC_INT64( my_local_metric, my_int )
14 end program myProg
```

#### G.1.2.15 #define SCOREP_USER_METRIC_INT64( *metricHandle, value* )

**Value:**

```
SCOREP_User_TriggerMetricInt64( \
        metricHandle, value );
```

Triggers a new event for a user counter of a 64 bit integer data type. Each user metric must be declared with SCOREP_USER_COUNTER_LOCAL, SCOREP_USER_COUNTER_GLOBAL, or SCOREP_USER_COUNTE↩ R_EXTERNAL and initialized with SCOREP_USER_COUNTER_INIT before it is triggered for the first time.

**Parameters**

| | |
|---|---|
| *metricHandle* | The handle of the metric for which a value is given in this statement. |
| *value* | The value of the counter. It must be possible for implicit casts to cast it to a 64 bit integer. |

C/C++ example:

```
1 SCOREP_USER_METRIC_LOCAL( my_local_metric )
2
3 int main()
4 {
5   SCOREP_USER_METRIC_INIT( my_local_metric, "My Metric", "seconds", \
6                            SCOREP_USER_METRIC_TYPE_INT64, \
7                            SCOREP_USER_METRIC_CONTEXT_GLOBAL )
8   // do something
9 }
10
11 void foo()
12 {
13   int64 my_int = get_some_int_value();
14   SCOREP_USER_METRIC_INT64( my_local_metric, my_int )
15 }
```

Fortran example:

```
1 program myProg
2   SCOREP_USER_METRIC_LOCAL( my_local_metric )
3   integer (kind=selected_int_kind(8)):: my_int = 19
4 ! more declarations
5
6
7   SCOREP_USER_METRIC_INIT( my_local_metric, "My Metric", "seconds", &
8                            SCOREP_USER_METRIC_TYPE_INT64, &
9                            SCOREP_USER_METRIC_CONTEXT_GLOBAL )
10
11 ! do something
12
13   SCOREP_USER_METRIC_INT64( my_local_metric, my_int )
14 end program myProg
```

#### G.1.2.16 #define SCOREP_USER_METRIC_LOCAL( *metricHandle* )

**Value:**

```
static SCOREP_SamplingSetHandle \
    metricHandle                                                    \
        = SCOREP_INVALID_SAMPLING_SET;
```

Declares a handle for a user metric. It defines a variable which must be in scope at all places where the metric is used. If it is used in more than one place it need to be a global definition.

**Parameters**

| | |
|---|---|
| *metricHandle* | The name of the variable which will be declared for storing the meric handle. |

C/C++ example:

```
1 SCOREP_USER_METRIC_LOCAL( my_local_metric )
2
3 int main()
4 {
5   SCOREP_USER_METRIC_INIT( my_local_metric, "My Metric", "seconds", \
6                            SCOREP_USER_METRIC_TYPE_UINT64, \
7                            SCOREP_USER_METRIC_CONTEXT_GLOBAL )
8   // do something
9 }
10
11 void foo()
12 {
13   uint64 my_int = get_some_int_value();
14   SCOREP_USER_METRIC_UINT64( my_local_metric, my_int )
15 }
```

Fortran example:

```
1 program myProg
2   SCOREP_USER_METRIC_LOCAL( my_local_metric )
3   integer (kind=selected_int_kind(8)):: my_int = 19
4 ! more declarations
5
6
7   SCOREP_USER_METRIC_INIT( my_local_metric, "My Metric", "seconds", &
8                            SCOREP_USER_METRIC_TYPE_INT64, &
9                            SCOREP_USER_METRIC_CONTEXT_GLOBAL )
10
11 ! do something
12
13   SCOREP_USER_METRIC_INT64( my_local_metric, my_int )
14 end program myProg
```

#### G.1.2.17 #define SCOREP_USER_METRIC_TYPE_DOUBLE 2

Indicates that a user counter is of data type double.

#### G.1.2.18 #define SCOREP_USER_METRIC_TYPE_INT64 0

Indicates that a user counter is of data type signed 64 bit integer.

#### G.1.2.19 #define SCOREP_USER_METRIC_TYPE_UINT64 1

Indicates that a user counter is of data type unsigned 64 bit integer.

#### G.1.2.20 #define SCOREP_USER_METRIC_UINT64( *metricHandle, value* )

**Value:**

```
SCOREP_User_TriggerMetricUint64( \
        metricHandle, value );
```

Triggers a new event for a user counter of a 64 bit unsigned integer data type. Each user metric must be declared with SCOREP_USER_COUNTER_LOCAL, SCOREP_USER_COUNTER_GLOBAL, or SCOREP_USER_COU↩ NTER_EXTERNAL and initialized with SCOREP_USER_COUNTER_INIT before it is triggered for the first time.

In Fortran is the unsigned integer type metric not available.

**Parameters**

| | |
|---|---|
| *metricHandle* | The handle of the metric for which a value is given in this statement. |
| *value* | The value of the counter. It must be possible for implicit casts to cast it to a 64 bit unsigned integer. |

Example:

```
1 SCOREP_USER_METRIC_LOCAL( my_local_metric )
2
3 int main()
4 {
5   SCOREP_USER_METRIC_INIT( my_local_metric, "My Metric", "seconds", \
6                            SCOREP_USER_METRIC_TYPE_UINT64, \
7                            SCOREP_USER_METRIC_CONTEXT_GLOBAL )
8   // do something
9 }
10
11 void foo()
12 {
13   uint64 my_int = get_some_int_value();
14   SCOREP_USER_METRIC_UINT64( my_local_metric, my_int )
15 }
```

#### G.1.2.21 #define SCOREP_USER_OA_PHASE_BEGIN( *handle, name, type* )

**Value:**

```
SCOREP_User_OaPhaseBegin( \
        &handle, &SCOREP_User_LastFileName, &SCOREP_User_LastFileHandle, name, \
        type, __FILE__, __LINE__ );
```

This macro marks the start of a user defined Online Access phase region. The SCOREP_USER_OA_PHASE_B↩ EGIN and SCOREP_USER_OA_PHASE_END must be correctly nested and be a potential global synchronization points, also it is recommended to mark the body of the application's main loop as a Online Access phase in order to utilize main loop iterations for iterative online analysis.

**Parameters**

| | |
|---|---|
| *handle* | The handle of the associated user region, which will become a root of the profile call-tree. This handle must be declared using SCOREP_USER_REGION_DEFINE or SCOREP_U↩ SER_GLOBAL_REGION_DEFINE before. |
| *name* | A string containing the name of the new region. The name should be unique. |
| *type* | Specifies the type of the region. Possible values are SCOREP_USER_REGION_TYPE_↩ COMMON, SCOREP_USER_REGION_TYPE_FUNCTION, SCOREP_USER_REGION_↩ TYPE_LOOP, SCOREP_USER_REGION_TYPE_DYNAMIC, SCOREP_USER_REGION↩ _TYPE_PHASE, or a combination of them. |

C/C++ example:

```
1 void main()
2 {
3   SCOREP_USER_REGION_DEFINE( my_region_handle )
4
5   // application initialization
6
7   for ( ) // main loop of the application
8       {
9               SCOREP_USER_OA_PHASE_BEGIN( my_region_handle, "main loop",SCOREP_USER_REGION_TYPE_COMMON )
10
11               // do something
12
13               SCOREP_USER_OA_PHASE_END( my_region_handle )
14       }
15
16       // application finalization
17 }
```

Fortran example:

```
1 program myProg
2   SCOREP_USER_REGION_DEFINE( my_region_handle )
3
4   ! applications initialization
5
6       ! main loop of the application
7       do ...
8
9   SCOREP_USER_OA_PHASE_BEGIN( my_region_handle, "main loop",SCOREP_USER_REGION_TYPE_COMMON )
10
11   ! do something
12
13   SCOREP_USER_OA_PHASE_END( my_region_handle )
14
15       enddo
16
17       !application finalization
18
19 end program myProg
```

#### G.1.2.22 #define SCOREP_USER_OA_PHASE_END( *handle* ) SCOREP_User_OaPhaseEnd( handle );

This macro marks the end of a user defined Online Access phase region. The SCOREP_USER_OA_PHASE_B↩ EGIN and SCOREP_USER_OA_PHASE_END must be correctly nested and be a potential global synchronization points, also it is recommended to mark the body of the application's main loop as a Online Access phase in order to utilize main loop iterations for iterative online analysis.

**Parameters**

| | |
|---|---|
| *handle* | The handle of the associated user region, which will become a root of the profile call-tree. This handle must be declared using SCOREP_USER_REGION_DEFINE or SCOREP_U↩ SER_GLOBAL_REGION_DEFINE before. C/C++ example: |

```
1 void main()
2 {
3   SCOREP_USER_REGION_DEFINE( my_region_handle )
4
5   // application initialization
6
7   for ( ) // main loop of the application
8       {
9               SCOREP_USER_OA_PHASE_BEGIN( my_region_handle, "main loop",SCOREP_USER_REGION_TYPE_COMMON
10
11               // do something
12
13               SCOREP_USER_OA_PHASE_END( my_region_handle )
14       }
15
16       // application finalization
17 }
```

Fortran example:

```
1 program myProg
2   SCOREP_USER_REGION_DEFINE( my_region_handle )
3
4   ! applications initialization
5
6       ! main loop of the application
7       do ...
8
9   SCOREP_USER_OA_PHASE_BEGIN( my_region_handle, "main loop",SCOREP_USER_REGION_TYPE_COMMON )
10
11   ! do something
12
13   SCOREP_USER_OA_PHASE_END( my_region_handle )
14
15       enddo
16
17       !application finalization
18
19 end program myProg
```

#### G.1.2.23 #define SCOREP_USER_PARAMETER_INT64( *name, value* )

**Value:**

```
{ \
        static SCOREP_User_ParameterHandle scorep_param =
      SCOREP_USER_INVALID_PARAMETER; \
        SCOREP_User_ParameterInt64( &scorep_param, name, value ); }
```

This statement adds a 64 bit signed integer type parameter for parameter-based profiling to the current region. The call-tree for the region is split according to the different values of the parameters with the same name. It is possible to add an arbitrary number of parameters to a region. Each parameter must have a unique name. However, it is not recommended to use more than 1 parameter per region.

**Parameters**

| | |
|---|---|
| *name* | A string containing the name of the parameter. |
| *value* | The value of the parameter. It must be possible for implicit casts to cast it to a 64 bit integer. |

C/C++ example:

```
1 void myfunc(int64 myint)
2 {
3   SCOREP_USER_REGION_DEFINE( my_region_handle )
4   SCOREP_USER_REGION_BEGIN( my_region_handle, "my_region",SCOREP_USER_REGION_TYPE_COMMON )
5   SCOREP_USER_PARAMETER_INT64("A nice int",myint)
6
7   // do something
8
9   SCOREP_USER_REGION_END( my_region_handle )
10 }
```

#### G.1.2.24 #define SCOREP_USER_PARAMETER_STRING( *name, value* )

**Value:**

```
{ \
        static SCOREP_User_ParameterHandle scorep_param =
      SCOREP_USER_INVALID_PARAMETER; \
        SCOREP_User_ParameterString( &scorep_param, name, value ); }
```

This statement adds a string type parameter for parameter-based profiling to the current region. The call-tree for the region is split according to the different values of the parameters with the same name. It is possible to add an arbitrary number of parameters to a region. Each parameter must have a unique name. However, it is not recommended to use more than 1 parameter per region. During one visit it is not allowed to use the same name twice for two different parameters.

**Parameters**

| | |
|---|---|
| *name* | A string containing the name of the parameter. |
| *value* | The value of the parameter. It must be a pointer to a C-string (a NULL-terminated string). |

C/C++ Example:

```
1 void myfunc(char *mystring)
2 {
3   SCOREP_USER_REGION_DEFINE( my_region_handle )
4   SCOREP_USER_REGION_BEGIN( my_region_handle, "my_region",SCOREP_USER_REGION_TYPE_COMMON )
5   SCOREP_USER_PARAMETER_STRING("A nice string",mystring)
6
7   // do something
8
9   SCOREP_USER_REGION_END( my_region_handle )
10 }
```

#### G.1.2.25 #define SCOREP_USER_PARAMETER_UINT64( *name, value* )

**Value:**

```
{ \
        static SCOREP_User_ParameterHandle scorep_param =
      SCOREP_USER_INVALID_PARAMETER; \
        SCOREP_User_ParameterUint64( &scorep_param, name, value ); }
```

This statement adds a 64 bit unsigned integer type parameter for parameter-based profiling to the current region. The call-tree for the region is split according to the different values of the parameters with the same name. It is possible to add an arbitrary number of parameters to a region. Each parameter must have a unique name. However, it is not recommended to use more than 1 parameter per region.

**Parameters**

| | |
|---|---|
| *name* | A string containing the name of the parameter. |
| *value* | The value of the parameter. It must be possible for implicit casts to cast it to a 64 bit unsigned integer. |

C/C++ example:

```
1 void myfunc(uint64 myuint)
2 {
3   SCOREP_USER_REGION_DEFINE( my_region_handle )
4   SCOREP_USER_REGION_BEGIN( my_region_handle, "my_region",SCOREP_USER_REGION_TYPE_COMMON )
5   SCOREP_USER_PARAMETER_UINT64("A nice unsigned int",myuint)
6
7   // do something
8
9   SCOREP_USER_REGION_END( my_region_handle )
10 }
```

#### G.1.2.26 #define SCOREP_USER_REGION( *name, type* )

Instruments a code block as a region with the given name. It inserts a local variable of the type class SCOREP_↩ User_Region. Its constructor generates the enter event and its destructor generates the exit event. Thus, only one statement is necessary to instrument the code block. This statement is only in C++ available.

**Parameters**

| | |
|---|---|
| *name* | A string containing the name of the new region. The name should be unique. |
| *type* | Specifies the type of the region. Possible values are SCOREP_USER_REGION_TYPE_↩ COMMON, SCOREP_USER_REGION_TYPE_FUNCTION, SCOREP_USER_REGION_↩ TYPE_LOOP, SCOREP_USER_REGION_TYPE_DYNAMIC, SCOREP_USER_REGION↩ _TYPE_PHASE, or a combination of them. |

Example:

```
1 void myfunc()
2 {
3   SCOREP_USER_REGION_( "myfunc", SCOREP_USER_REGION_TYPE_FUNCTION )
4
5   // do something
6 }
```

#### G.1.2.27 #define SCOREP_USER_REGION_BEGIN( *handle, name, type* )

**Value:**

```
SCOREP_User_RegionBegin( \
        &handle, &SCOREP_User_LastFileName, &SCOREP_User_LastFileHandle, name, \
        type, __FILE__, __LINE__ );
```

This macro marks the start of a user defined region. The SCOREP_USER_REGION_BEGIN and SCOREP_US↩ ER_REGION_END calls of all regions must be correctly nested.

**Parameters**

| | |
|---|---|
| *handle* | The handle of the region to be started. This handle must be declared using SCOREP_USER_REGION_DEFINE or SCOREP_USER_GLOBAL_REGION_DEFINE before. |
| *name* | A string containing the name of the new region. The name should be unique. |
| *type* | Specifies the type of the region. Possible values are SCOREP_USER_REGION_TYPE_COMMON, SCOREP_USER_REGION_TYPE_FUNCTION, SCOREP_USER_REGION_TYPE_LOOP, SCOREP_USER_REGION_TYPE_DYNAMIC, SCOREP_USER_REGION_TYPE_PHASE, or a combination of them. |

C/C++ example:

```
1 void myfunc()
2 {
3   SCOREP_USER_REGION_DEFINE( my_region_handle )
4
5   // do something
6
7   SCOREP_USER_REGION_BEGIN( my_region_handle, "my_region",SCOREP_USER_REGION_TYPE_COMMON )
8
9   // do something
10
11   SCOREP_USER_REGION_END( my_region_handle )
12 }
```

Fortran example:

```
1 program myProg
2   SCOREP_USER_REGION_DEFINE( my_region_handle )
3   ! more declarations
4
5   SCOREP_USER_REGION_BEGIN( my_region_handle, "my_region",SCOREP_USER_REGION_TYPE_COMMON )
6   ! do something
7   SCOREP_USER_REGION_END( my_region_handle )
8
9 end program myProg
```

#### G.1.2.28 #define SCOREP_USER_REGION_DEFINE( *handle* ) static SCOREP_User_RegionHandle handle = SCOREP_USER_INVALID_REGION;

This macro defines a user region handle in a local context. Every user handle must be defined, before it can be used.

**Parameters**

| | |
|---|---|
| *handle* | A unique name for the handle must be provided. This handle is declared in the macro. This handle is used in the SCOREP_USER_REGION_BEGIN and SCOREP_USER_REGION_END statements to specify which region is started, or ended. If you are using a Fortran version which has a limited length of code lines, the length of the *handle* parameter must be at most 4 characters, else the declaration line exceeds the allowed length. |

C/C++ example:

```
1 void myfunc()
2 {
3   SCOREP_USER_REGION_DEFINE( my_region_handle )
4
5   // do something
6
7   SCOREP_USER_REGION_BEGIN( my_region_handle, "my_region",SCOREP_USER_REGION_TYPE_COMMON )
8
9   // do something
10
11   SCOREP_USER_REGION_END( my_region_handle )
12 }
```

Fortran example:

```
1 program myProg
2   SCOREP_USER_REGION_DEFINE( my_region_handle )
```

```
3   ! more declarations
4
5   SCOREP_USER_REGION_BEGIN( my_region_handle, "my_region",SCOREP_USER_REGION_TYPE_COMMON )
6   ! do something
7   SCOREP_USER_REGION_END( my_region_handle )
8
9 end program myProg
```

#### G.1.2.29 #define SCOREP_USER_REGION_END( *handle* ) SCOREP_User_RegionEnd( handle );

This macro marks the end of a user defined region. The SCOREP_USER_REGION_BEGIN and SCOREP_USE↩ R_REGION_END calls of all regions must be correctly nested.

**Parameters**

| | |
|---|---|
| *handle* | The handle of the region which ended here. It must be the same handle which is used as the start of the region. |

C/C++ example:

```
1 void myfunc()
2 {
3   SCOREP_USER_REGION_DEFINE( my_region_handle )
4
5   // do something
6
7   SCOREP_USER_REGION_BEGIN( my_region_handle, "my_region",SCOREP_USER_REGION_TYPE_COMMON )
8
9   // do something
10
11   SCOREP_USER_REGION_END( my_region_handle )
12 }
```

Fortran example:

```
1 program myProg
2   SCOREP_USER_REGION_DEFINE( my_region_handle )
3   ! more declarations
4
5   SCOREP_USER_REGION_BEGIN( my_region_handle, "my_region",SCOREP_USER_REGION_TYPE_COMMON )
6   ! do something
7   SCOREP_USER_REGION_END( my_region_handle )
8
9 end program myProg
```

#### G.1.2.30 #define SCOREP_USER_REGION_ENTER( *handle* ) SCOREP_User_RegionEnter( handle );

This macro marks the beginning of a user defined and already initialized region. The SCOREP_USER_REGI↩ ON_BEGIN/SCOREP_USER_REGION_ENTER and SCOREP_USER_REGION_END calls of all regions must be correctly nested. To initialize the region handle, SCOREP_USER_REGION_INIT or SCOREP_USER_REGION_↩ BEGIN must be called before.

**Parameters**

| | |
|---|---|
| *handle* | The handle of the region which ended here. It must be the same handle which is used as the start of the region. |

C/C++ example:

```
1 void myfunc()
2 {
3   SCOREP_USER_REGION_DEFINE( my_region_handle )
4
5   // do something
6
7   SCOREP_USER_REGION_INIT( my_region_handle, "my_region",SCOREP_USER_REGION_TYPE_COMMON )
8   SCOREP_USER_REGION_ENTER( my_region_handle )
9
10   // do something
11
12   SCOREP_USER_REGION_END( my_region_handle )
13 }
```

Fortran example:

```
1 program myProg
2   SCOREP_USER_REGION_DEFINE( my_region_handle )
3   ! more declarations
4
5   SCOREP_USER_REGION_INIT( my_region_handle, "my_region",SCOREP_USER_REGION_TYPE_COMMON )
6   SCOREP_USER_REGION_ENTER( my_region_handle )
7   ! do something
8   SCOREP_USER_REGION_END( my_region_handle )
9
10 end program myProg
```

#### G.1.2.31 #define SCOREP_USER_REGION_INIT( *handle, name, type* )

**Value:**

```
SCOREP_User_RegionInit( \
        &handle, &SCOREP_User_LastFileName, &SCOREP_User_LastFileHandle, name, \
        type, __FILE__, __LINE__ );
```

This macro initializes a user defined region. If the region handle is already inititialized, no operation is executed.

**Parameters**

| | |
|---|---|
| *handle* | The handle of the region to be started. This handle must be declared using SCOREP_USER_REGION_DEFINE or SCOREP_USER_GLOBAL_REGION_DEFINE before. |
| *name* | A string containing the name of the new region. The name should be unique. |
| *type* | Specifies the type of the region. Possible values are SCOREP_USER_REGION_TYPE_COMMON, SCOREP_USER_REGION_TYPE_FUNCTION, SCOREP_USER_REGION_TYPE_LOOP, SCOREP_USER_REGION_TYPE_DYNAMIC, SCOREP_USER_REGION_TYPE_PHASE, or a combination of them. |

C/C++ example:

```
1 void myfunc()
2 {
3   SCOREP_USER_REGION_DEFINE( my_region_handle )
4
5   // do something
6
7   SCOREP_USER_REGION_INIT( my_region_handle, "my_region",SCOREP_USER_REGION_TYPE_COMMON )
8   SCOREP_USER_REGION_ENTER( my_region_handle )
9
10   // do something
11
12   SCOREP_USER_REGION_END( my_region_handle )
13 }
```

Fortran example:

```
1 program myProg
2   SCOREP_USER_REGION_DEFINE( my_region_handle )
3   ! more declarations
4
5   SCOREP_USER_REGION_INIT( my_region_handle, "my_region",SCOREP_USER_REGION_TYPE_COMMON )
6   SCOREP_USER_REGION_ENTER( my_region_handle )
7   ! do something
8   SCOREP_USER_REGION_END( my_region_handle )
9
10 end program myProg
```

#### G.1.2.32 #define SCOREP_USER_REGION_TYPE_COMMON 0

Region without any specific type.

#### G.1.2.33 #define SCOREP_USER_REGION_TYPE_DYNAMIC 4

Marks the regions as dynamic.

#### G.1.2.34 #define SCOREP_USER_REGION_TYPE_FUNCTION 1

Marks the region as being the codeblock of a function.

#### G.1.2.35 #define SCOREP_USER_REGION_TYPE_LOOP 2

Marks the region as being the codeblock of a loop with the same number of iterations on all processes.

#### G.1.2.36 #define SCOREP_USER_REGION_TYPE_PHASE 8

Marks the region as being a root node of a phase.

# Appendix H

# Data Structure Documentation

## H.1 SCOREP_Metric_Plugin_Info Struct Reference

`#include <SCOREP_MetricPlugins.h>`

### Data Fields

- int32_t(∗ add_counter )(char ∗metric_name)
- uint64_t delta_t
- void(∗ finalize )(void)
- uint64_t(∗ get_all_values )(int32_t id, SCOREP_MetricTimeValuePair ∗∗time_value_list)
- uint64_t(∗ get_current_value )(int32_t id)
- SCOREP_Metric_Plugin_MetricProperties ∗(∗ get_event_info )(char ∗token)
- bool(∗ get_optional_value )(int32_t id, uint64_t ∗value)
- int32_t(∗ initialize )(void)
- uint32_t plugin_version
- uint64_t reserved [92]
- SCOREP_MetricPer run_per
- void(∗ set_clock_function )(uint64_t(∗clock_time)(void))
- SCOREP_MetricSynchronicity sync
- void(∗ synchronize )(bool is_responsible, SCOREP_MetricSynchronizationMode sync_mode)

### H.1.1 Detailed Description

Information on that defines the plugin. All values that are not explicitly defined should be set to 0

### H.1.2 Field Documentation

#### H.1.2.1 int32_t( ∗ SCOREP_Metric_Plugin_Info::add_counter) (char ∗metric_name)

Depending on run_per, this function is called per thread, per process, per host, or only on a single thread. Further it is called for each metric as returned by the calls to get_event_info.

The function sets up the measurement for the requested metric and returns a non-negative unique ID which is from now on used to refer to this metric.

**Parameters**

| | |
|---|---|
| *metric_name* | Name of an individual metric |

**Returns**

non-negative ID of requested metric or negative value in cased of failure

#### H.1.2.2 uint64_t SCOREP_Metric_Plugin_Info::delta_t

Set a specific interval for reading metric values. Score-P will request metric values of a plugin, at the earliest, after delta_t ticks after it was last read. NOTE: This is only a lower limit for the time between two reads

- there is no upper limit. This setting is used by plugins of synchronicity type SCOREP_METRIC_SYNC, SCOREP_METRIC_ASYNC_EVENT, and SCOREP_METRIC_ASYNC. In combination with SCOREP_METRIC_SYNC, it can be used for metrics that update at known intervals and therefore reduce the over head of reading unchanged values. In combination with SCOREP_METRIC_ASYNC_EVENT it can be used similarly. This value is ignored for SCOREP_METRIC_STRICTLY_SYNC metrics.

#### H.1.2.3 void( * SCOREP_Metric_Plugin_Info::finalize) (void)

This functions is called once per process to clean up all resources used by the metric plugin.

#### H.1.2.4 uint64_t( * SCOREP_Metric_Plugin_Info::get_all_values) (int32_t id, SCOREP_MetricTimeValuePair **time_value_list)

This function provides the recorded value of the selected metric. It must be implemented by asynchronous metric plugins. The timestamps in the returned list should correspond to the clock provided by set_clock_function. Further, all values (timestamps) should lie within within the following interval: synchronize(SCOREP_METRIC_SYNCHRONIZATION_MODE_BEGIN|SCOREP_METRIC_SYNCHRONIZATION_MODE_BEGIN_MPP), synchronize(SCOREP_METRIC_SYNCHRONIZATION_MODE_END). Score-P takes ownership of the `*time_value_list` memory. Make sure the memory remains valid and is never modified by the plugin. Score-P will release the memory using `free`. The pointer must be created directly by malloc/calloc etc. directly. Do not use a memory pool / pointer with offset into a larger memory etc.

**Parameters**

| | | |
|---|---|---|
| | *id* | Metric id (see add_counter). |
| `out` | *time_value_list* | Pointer to list with return values (pairs of timestamp and value). |

**See also**

SCOREP_MetricSynchronizationMode

**Returns**

Number of elements within `*time_value_list`

#### H.1.2.5 uint64_t( * SCOREP_Metric_Plugin_Info::get_current_value) (int32_t id)

This function shall provide the current value of a metric. It must be implemented by strictly synchronous metric plugins. It is called according to the run_per specification.

**Parameters**

| | |
|---|---|
| *id* | Metric id (see add_counter). |

**Returns**

Current value of requested metric. For metrics of value_type other than UINT64, the data should be reinterpreted to a UINT64 using a union.

#### H.1.2.6 SCOREP_Metric_Plugin_MetricProperties*( * SCOREP_Metric_Plugin_Info::get_event_info) (char *token)

A user specifies a SCOREP_METRIC_EXAMPLE_PLUGIN=token1,token2,... This function is called once per process and token. Each token can result in any number of metrics (wildcards). The function shall provide the properties of the metrics for this token.

The total list of metrics returned by the calls for all tokens comprises the metrics that will be recorded by the plugin.

Note: The properties-array must contain an additional end entry with name = `NULL`.

Note: The properties-array memory and all indirect pointers are managed by Score-P now. Make sure the memory remains valid and unmodified. All memory may be released with `free` by Score-P. Make sure that all provided pointers are created by malloc/strdup/....

**Parameters**

| | |
|---|---|
| *token* | String that describes one or multiple metrics. |

**Returns**

properties Meta data about the metrics available for this token.

#### H.1.2.7 bool( * SCOREP_Metric_Plugin_Info::get_optional_value) (int32_t id, uint64_t *value)

This function provides the current value of a metric if available. It must be implemented by synchronous metric plugins. It is called according to the run_per specification.

**Parameters**

| | | |
|---|---|---|
| | *id* | Metric id (see add_counter). |
| `out` | *value* | Current value of requested metric. For metrics of value_type other than UIN↩ T64, the data should be reinterpreted to a UINT64 using a union. |

**Returns**

True if value of requested metric was written, otherwise false.

#### H.1.2.8 int32_t( * SCOREP_Metric_Plugin_Info::initialize) (void)

This function is called once per process. It should check that all requirements are met (e.g., are special libraries needed and available, has the user appropriate rights to access implemented metrics). If all requirements are met, data structures used by the plugin can be initialized within this function.

**Returns**

0 if successful, error code if failure

#### H.1.2.9 uint32_t SCOREP_Metric_Plugin_Info::plugin_version

Should be set to SCOREP_PLUGIN_VERSION (needed for back- and forward compatibility)

#### H.1.2.10 uint64_t SCOREP_Metric_Plugin_Info::reserved[92]

Reserved space for future features, should be zeroed

#### H.1.2.11 SCOREP_MetricPer SCOREP_Metric_Plugin_Info::run_per

Defines how many threads should record the metrics of a plugin. For the available options see SCOREP_MetricPer.

#### H.1.2.12 void( * SCOREP_Metric_Plugin_Info::set_clock_function) (uint64_t(*clock_time)(void))

When this callback is implemented, Score-P calls it once to provide a clock function that allows the plugin to read the current time in Score-P ticks. This should be used by asynchronous metric plugins.

Note: This function is called before initialize.

**Parameters**

| | |
|---|---|
| *clock_time* | Function pointer to Score-P's clock function. |

#### H.1.2.13 SCOREP_MetricSynchronicity SCOREP_Metric_Plugin_Info::sync

Defines how metrics are measured over time and how they are collected by Score-P. This setting influences when and which callback functions are called by Score-P. For the available options see SCOREP_MetricSynchronicity.

#### H.1.2.14 void( * SCOREP_Metric_Plugin_Info::synchronize) (bool is_responsible, SCOREP_MetricSynchronizationMode sync_mode)

This callback is used for stating and stopping the measurement of asynchronous metrics and can also be used for time synchronization purposes. This function is called for all threads in the application, but the threads that handle the metric plugin according to run_per will be marked as is_responsible. The function will be called approximately at the same time for all threads:

- Once the beginning with SCOREP_METRIC_SYNCHRONIZATION_MODE_BEGIN or SCOREP_METRIC_SYNCHRONIZATION_MODE_BEGIN_MPP for (non-)MPI programs respectively.
- Once at the end with SCOREP_METRIC_SYNCHRONIZATION_MODE_END For asynchronous metrics, starting and stopping a measurement should be done in this function, not in add_counter.

  **Parameters**

  | | |
  |---|---|
  | *is_responsible* | Flag to mark responsibility as per run_per |
  | *sync_mode* | Mode of synchronization point, e.g. SCOREP_METRIC_SYNCHRONIZATION_MODE_BEGIN, SCOREP_METRIC_SYNCHRONIZATION_MODE_BEGIN_MPP, SCOREP_METRIC_SYNCHRONIZATION_MODE_END |

  **See also**

  SCOREP_MetricSynchronizationMode

The documentation for this struct was generated from the following file:

- SCOREP_MetricPlugins.h

## H.2 SCOREP_Metric_Plugin_MetricProperties Struct Reference

Properties describing a metric. Provided by the get_event_info function.

`#include <SCOREP_MetricPlugins.h>`

### Data Fields

- SCOREP_MetricBase base
- char * description
- int64_t exponent
- SCOREP_MetricMode mode
- char * name
- char * unit
- SCOREP_MetricValueType value_type

### H.2.1 Detailed Description

Properties describing a metric. Provided by the get_event_info function.

### H.2.2 Field Documentation

#### H.2.2.1 SCOREP_MetricBase SCOREP_Metric_Plugin_MetricProperties::base

Base of metric: decimal, binary

#### H.2.2.2 char* SCOREP_Metric_Plugin_MetricProperties::description

Additional information about the metric

#### H.2.2.3 int64_t SCOREP_Metric_Plugin_MetricProperties::exponent

Exponent to scale metric: e.g., 3 for kilo

#### H.2.2.4 SCOREP_MetricMode SCOREP_Metric_Plugin_MetricProperties::mode

Metric mode: valid combination of ACCUMULATED|ABSOLUTE|RELATIVE + POINT|START|LAST|NEXT

**See also**

SCOREP_MetricMode

#### H.2.2.5 char* SCOREP_Metric_Plugin_MetricProperties::name

Plugin name

#### H.2.2.6 char* SCOREP_Metric_Plugin_MetricProperties::unit

Unit string of recorded metric

#### H.2.2.7 SCOREP_MetricValueType SCOREP_Metric_Plugin_MetricProperties::value_type

Value type: signed 64 bit integer, unsigned 64 bit integer, double

The documentation for this struct was generated from the following file:

- SCOREP_MetricPlugins.h

## H.3 SCOREP_Metric_Properties Struct Reference

`#include <SCOREP_MetricTypes.h>`

### Data Fields

- SCOREP_MetricBase base
- const char * description
- int64_t exponent
- SCOREP_MetricMode mode
- const char * name
- SCOREP_MetricProfilingType profiling_type
- SCOREP_MetricSourceType source_type
- const char * unit
- SCOREP_MetricValueType value_type

### H.3.1 Detailed Description

Properties desribing a metric.

### H.3.2 Field Documentation

#### H.3.2.1 SCOREP_MetricBase SCOREP_Metric_Properties::base

Base of metric values (DECIMAL or BINARY).

#### H.3.2.2 const char* SCOREP_Metric_Properties::description

Long description of the metric.

#### H.3.2.3 int64_t SCOREP_Metric_Properties::exponent

Exponent to scale metric values (metric_value = value * base^exponent).

#### H.3.2.4 SCOREP_MetricMode SCOREP_Metric_Properties::mode

Metric mode (valid combination of ACCUMULATED|ABSOLUTE|RELATIVE and POINT|START|LAST|NEXT).

#### H.3.2.5 const char* SCOREP_Metric_Properties::name

Name of the metric.

#### H.3.2.6 SCOREP_MetricProfilingType SCOREP_Metric_Properties::profiling_type

Profiling type of the metric.

#### H.3.2.7 SCOREP_MetricSourceType SCOREP_Metric_Properties::source_type

Type of the metric source (e.g. PAPI).

#### H.3.2.8 const char* SCOREP_Metric_Properties::unit

Unit of the metric.

#### H.3.2.9 SCOREP_MetricValueType SCOREP_Metric_Properties::value_type

Type of the metric value (INT64, UINT64, or DOUBLE).

The documentation for this struct was generated from the following file:

- SCOREP_MetricTypes.h

## H.4 SCOREP_MetricTimeValuePair Struct Reference

`#include <SCOREP_MetricTypes.h>`

### Data Fields

- uint64_t timestamp
- uint64_t value

### H.4.1 Detailed Description

Pair of Score-P timestamp and corresponding metric value (used by asynchronous metrics).

### H.4.2 Field Documentation

#### H.4.2.1 uint64_t SCOREP_MetricTimeValuePair::timestamp

Timestamp in Score-P time!

#### H.4.2.2 uint64_t SCOREP_MetricTimeValuePair::value

Current metric value

The documentation for this struct was generated from the following file:

- SCOREP_MetricTypes.h

# Appendix I

# File Documentation

## I.1 SCOREP_MetricPlugins.h File Reference

Description of the metric plugin header. For information on how to use metric plugins, please refer to **??**.

```
#include <stdbool.h>
#include <scorep/SCOREP_MetricTypes.h>
```

### Data Structures

- struct SCOREP_Metric_Plugin_Info
- struct SCOREP_Metric_Plugin_MetricProperties

  *Properties describing a metric. Provided by the get_event_info function.*

### Macros

- #define SCOREP_METRIC_PLUGIN_ENTRY(_name)
- #define SCOREP_METRIC_PLUGIN_VERSION 1

### I.1.1 Detailed Description

Description of the metric plugin header. For information on how to use metric plugins, please refer to **??**.

### I.1.2 Macro Definition Documentation

#### I.1.2.1 #define SCOREP_METRIC_PLUGIN_ENTRY( *_name* )

**Value:**

```
EXTERN SCOREP_Metric_Plugin_Info \
    SCOREP_MetricPlugin_ ## _name ## _get_info( void )
```

Macro used for implementation of the 'get_info' function

#### I.1.2.2 #define SCOREP_METRIC_PLUGIN_VERSION 1

The developer of a metric plugin should provide a README file which explains how to compile, install and use the plugin. In particular, the supported metrics should be described in the README file.

Each metric plugin has to include SCOREP_MetricPlugins.h and implement a 'get_info' function. Therefore, use the SCOREP_METRIC_PLUGIN_ENTRY macro and provide the name of the plugin library as the argument. For example, the metric plugin libexample_plugin.so should use SCOREP_METRIC_PLUGIN_ENTRY( example_plugin ).

It is encouraged to use the "_plugin" suffix on the name to avoid conflicts with existing libraries, e.g., libsensors_plugin.so using the existing libsensors.so.

### I.1.3 Mandatory functions

See each function for details.

initialize

Check requirements and initialize the plugin.

get_event_info

A user specifies a SCOREP_METRIC_EXAMPLE_PLUGIN=token1,token2,... This function provides information about the metric(s) corresponding to this token. The total list of metrics returned for all tokens will then be recorded by the plugin.

add_counter

The function is called for and sets each of the metrics to be recorded by the plugin. It provides a unique ID for each metric.

finalize

Clean up the resources used by the metric plugin.

### I.1.4 Mandatory variables

run_per

Defines how many threads should record the metrics of a plugin.

sync

Defines synchronicity type of a metric plugin. A metric plugin can

- provide a metric value for each event (SCOREP_METRIC_STRICTLY_SYNC)
- optionally provide a metric value for each Score-P event (SCOREP_METRIC_SYNC)
- measure metric values independently of Score-P events, but collect them in Score-p during a Score-P event (SCOREP_METRIC_ASYNC_EVENT)
- measure all metric values independently of events and collect them once at the very end of execution (SCOREP_METRIC_ASYNC)

plugin_version

Should be set to SCOREP_METRIC_PLUGIN_VERSION

Depending on the plugin's synchronicity type there are some optional functions and variables.

### I.1.5 Optional functions

get_current_value

Used by strictly synchronous metric plugins only. Returns value of requested metric.

get_optional_value

Used by synchronous metric plugins, but not by strictly synchronous ones. This function requests current value of a metric, but it is valid that no value is returned (read: no update for this metric available).

get_all_values

Used by asynchronous metric plugins. This function is used to request values of a asynchronous metric. The metric will return an arbitrary number of timestamp-value-pairs.

set_clock_function

Used by asynchronous metric plugins. This function passes a function to the plugin, which can be used by the plugin to get a Score-P valid timestamp.

### I.1.6 Optional variables

delta_t

Defines interval between two calls to update metric value. Ignored for strictly synchronous plugins.Current version of Score-P metric plugin interface

## I.2 SCOREP_MetricTypes.h File Reference

Types used by metric service.

```
#include <stdint.h>
```

### Data Structures

- struct SCOREP_Metric_Properties
- struct SCOREP_MetricTimeValuePair

### Enumerations

- enum SCOREP_MetricBase {
  SCOREP_METRIC_BASE_BINARY = 0,
  SCOREP_METRIC_BASE_DECIMAL = 1,
  SCOREP_INVALID_METRIC_BASE }
- enum SCOREP_MetricMode {
  SCOREP_METRIC_MODE_ACCUMULATED_START = 0,
  SCOREP_METRIC_MODE_ACCUMULATED_POINT = 1,
  SCOREP_METRIC_MODE_ACCUMULATED_LAST = 2,
  SCOREP_METRIC_MODE_ACCUMULATED_NEXT = 3,
  SCOREP_METRIC_MODE_ABSOLUTE_POINT = 4,
  SCOREP_METRIC_MODE_ABSOLUTE_LAST = 5,
  SCOREP_METRIC_MODE_ABSOLUTE_NEXT = 6,
  SCOREP_METRIC_MODE_RELATIVE_POINT = 7,
  SCOREP_METRIC_MODE_RELATIVE_LAST = 8,
  SCOREP_METRIC_MODE_RELATIVE_NEXT = 9 }
- enum SCOREP_MetricPer {
  SCOREP_METRIC_PER_THREAD = 0,
  SCOREP_METRIC_PER_PROCESS,
  SCOREP_METRIC_PER_HOST,
  SCOREP_METRIC_ONCE }

- enum SCOREP_MetricProfilingType {
  SCOREP_METRIC_PROFILING_TYPE_EXCLUSIVE = 0,
  SCOREP_METRIC_PROFILING_TYPE_INCLUSIVE = 1,
  SCOREP_METRIC_PROFILING_TYPE_SIMPLE = 2,
  SCOREP_METRIC_PROFILING_TYPE_MAX = 3,
  SCOREP_METRIC_PROFILING_TYPE_MIN = 4 }
- enum SCOREP_MetricSourceType {
  SCOREP_METRIC_SOURCE_TYPE_PAPI = 0,
  SCOREP_METRIC_SOURCE_TYPE_RUSAGE = 1,
  SCOREP_METRIC_SOURCE_TYPE_USER = 2,
  SCOREP_METRIC_SOURCE_TYPE_OTHER = 3,
  SCOREP_METRIC_SOURCE_TYPE_TASK = 4,
  SCOREP_METRIC_SOURCE_TYPE_PLUGIN = 5,
  SCOREP_METRIC_SOURCE_TYPE_PERF = 6 }
- enum SCOREP_MetricSynchronicity {
  SCOREP_METRIC_STRICTLY_SYNC = 0,
  SCOREP_METRIC_SYNC,
  SCOREP_METRIC_ASYNC_EVENT,
  SCOREP_METRIC_ASYNC }
- enum SCOREP_MetricSynchronizationMode {
  SCOREP_METRIC_SYNCHRONIZATION_MODE_BEGIN,
  SCOREP_METRIC_SYNCHRONIZATION_MODE_BEGIN_MPP,
  SCOREP_METRIC_SYNCHRONIZATION_MODE_END }
- enum SCOREP_MetricValueType {
  SCOREP_METRIC_VALUE_INT64,
  SCOREP_METRIC_VALUE_UINT64,
  SCOREP_METRIC_VALUE_DOUBLE }

### I.2.1 Detailed Description

Types used by metric service.

### I.2.2 Enumeration Type Documentation

#### I.2.2.1 enum SCOREP_MetricBase

Types to be used in defining metric base (SCOREP_Definitions_NewMetric()).

**Enumerator**

***SCOREP_METRIC_BASE_BINARY*** Binary base.

***SCOREP_METRIC_BASE_DECIMAL*** Decimal base.

***SCOREP_INVALID_METRIC_BASE*** For

#### I.2.2.2 enum SCOREP_MetricMode

Types to be used in defining metric mode (SCOREP_Definitions_NewMetric()). The mode consists of a timing and a value semantic. The possible value semantics are:

- Accumulated for classic counters, e.g. number of floating point operations. While they are stored monotonically increasing in the trace, they are often differentiated as rate over time.
- Absolute values, e.g. temperature. They are stored as is in the trace and typically also displayed as is.
- Relative values.

The possible timing semantics are:

- Start: The value is valid for the interval from the beginning of the trace to the associated timestamp.
- Point: The value is only valid for the point in time given by the timestamp.
- Last: The value is valid for the interval from the previous to the current timestamp.
- Next: The value is valid for the interval from the current to the next timestamp.

**Enumerator**

***SCOREP_METRIC_MODE_ACCUMULATED_START*** Accumulated metric, 'START' timing.

***SCOREP_METRIC_MODE_ACCUMULATED_POINT*** Accumulated metric, 'POINT' timing.

***SCOREP_METRIC_MODE_ACCUMULATED_LAST*** Accumulated metric, 'LAST' timing.

***SCOREP_METRIC_MODE_ACCUMULATED_NEXT*** Accumulated metric, 'NEXT' timing.

***SCOREP_METRIC_MODE_ABSOLUTE_POINT*** Absolute metric, 'POINT' timing.

***SCOREP_METRIC_MODE_ABSOLUTE_LAST*** Absolute metric, 'LAST' timing.

***SCOREP_METRIC_MODE_ABSOLUTE_NEXT*** Absolute metric, 'NEXT' timing.

***SCOREP_METRIC_MODE_RELATIVE_POINT*** Relative metric, 'POINT' timing.

***SCOREP_METRIC_MODE_RELATIVE_LAST*** Relative metric, 'LAST' timing.

***SCOREP_METRIC_MODE_RELATIVE_NEXT*** Relative metric, 'NEXT' timing.

#### I.2.2.3 enum SCOREP_MetricPer

Enumeration to define how many threads should record the metrics of a plugin. Used by SCOREP_Metric_Plugin↩ _Info::run_per.

**Enumerator**

***SCOREP_METRIC_PER_THREAD*** Metric values are recorded on all threads of all processes

***SCOREP_METRIC_PER_PROCESS*** If processes use multiple threads, the metric is recorded on the main thread of each process.

***SCOREP_METRIC_PER_HOST*** Metric values are recorded on a single thread of each node in a parallel program running on multiple nodes (hosts). Nodes are determined by the platform-specific Score-P node identifier.

***SCOREP_METRIC_ONCE*** Metric values recorded once within the parallel program. They are recorded on the first node, first process, first thread.

#### I.2.2.4 enum SCOREP_MetricProfilingType

Types used to define type of profiling.

**Enumerator**

***SCOREP_METRIC_PROFILING_TYPE_EXCLUSIVE*** Exclusive values (excludes values from subordinated items)

***SCOREP_METRIC_PROFILING_TYPE_INCLUSIVE*** Inclusive values (sum including values from subordinated items )

***SCOREP_METRIC_PROFILING_TYPE_SIMPLE*** Single value

***SCOREP_METRIC_PROFILING_TYPE_MAX*** Maximum values

***SCOREP_METRIC_PROFILING_TYPE_MIN*** Minimum values

#### I.2.2.5 enum SCOREP_MetricSourceType

Metric sources to be used in defining a metric member (SCOREP_Definitions_NewMetric()).

**Enumerator**

***SCOREP_METRIC_SOURCE_TYPE_PAPI*** PAPI counter.

***SCOREP_METRIC_SOURCE_TYPE_RUSAGE*** Resource usage counter.

***SCOREP_METRIC_SOURCE_TYPE_USER*** User metrics.

***SCOREP_METRIC_SOURCE_TYPE_OTHER*** Any other metrics.

***SCOREP_METRIC_SOURCE_TYPE_TASK*** Generated by task profiling.

***SCOREP_METRIC_SOURCE_TYPE_PLUGIN*** Plugin metrics.

***SCOREP_METRIC_SOURCE_TYPE_PERF*** Linux perf metrics

#### I.2.2.6 enum SCOREP_MetricSynchronicity

Enumeration to define the synchronicity type of a metric. Used by SCOREP_Metric_Plugin_Info::sync.

**Enumerator**

***SCOREP_METRIC_STRICTLY_SYNC*** The current value of each metric is queried by Score-P whenever an enter/leave event occurs via get_current_value. The plugin must always be able to provide a current value. The plugin provides the value itself, the timestamp is provided by Score-P. This setting is used for metrics that can be measured with minimal runtime costs and updated very frequently.

***SCOREP_METRIC_SYNC*** The current value of each metric is queried by Score-P whenever an enter/leave event occurs via get_optional_value. Providing a value is optional in case no new value is available in the plugin. The plugin provides the value itself, the timestamp is provided by Score-P. This setting is used for metrics that can be measured with minimal runtime costs but do not necessarily always change.

***SCOREP_METRIC_ASYNC_EVENT*** Metric values are be measured at arbitrary points in time, but are collected at enter/leave events. Whenever an enter/leave event occurs, Score-P queries the plugin via get_all_values for a list of timestamp-value-pairs. This setting can be used for some special cases, SCOREP_METRIC_ASYNC is usually easier to implement.

***SCOREP_METRIC_ASYNC*** Metric values are be measured at arbitrary points in time. All values are collected once at the very end of the execution. Score-P collects the values and associated timestamps via get_all_values. This setting is used for metrics that are recorded on external systems or within a separate thread. While it does require additional memory buffers to store the measurement, it usually reduces the overhead by decoupling the measurement from collection. It is also called post-mortem processing.

#### I.2.2.7 enum SCOREP_MetricSynchronizationMode

Possible modes of a synchronization point. Express the time when a synchronization happens.

**Enumerator**

***SCOREP_METRIC_SYNCHRONIZATION_MODE_BEGIN*** Synchronization at the beginning of the measurement

***SCOREP_METRIC_SYNCHRONIZATION_MODE_BEGIN_MPP*** Synchronization at the initialization of a multi-process paradigm (e.g., MPI)

***SCOREP_METRIC_SYNCHRONIZATION_MODE_END*** Synchronization at the end of the measurement

#### I.2.2.8 enum SCOREP_MetricValueType

Types to be used in defining type of metric values (SCOREP_Definitions_NewMetric()). The interface uses UINT64 for all values, the other types should be reinterpreted using a union.

**Enumerator**

***SCOREP_METRIC_VALUE_INT64*** 64 bit integer

***SCOREP_METRIC_VALUE_UINT64*** 64 bit unsigned integer

***SCOREP_METRIC_VALUE_DOUBLE*** double precision floating point

## I.3 SCOREP_User.h File Reference

This file contains the interface for the manual user instrumentation.

```
#include <scorep/SCOREP_User_Variables.h>
#include <scorep/SCOREP_User_Functions.h>
```

### Macros

#### Macros for region instrumentation

- #define SCOREP_USER_FUNC_DEFINE()
- #define SCOREP_USER_OA_PHASE_BEGIN(handle, name, type)
- #define SCOREP_USER_OA_PHASE_END(handle) SCOREP_User_OaPhaseEnd( handle );
- #define SCOREP_USER_REGION_DEFINE(handle) static SCOREP_User_RegionHandle handle = S↩ COREP_USER_INVALID_REGION;
- #define SCOREP_USER_REGION_ENTER(handle) SCOREP_User_RegionEnter( handle );
- #define SCOREP_USER_REGION_BEGIN(handle, name, type)
- #define SCOREP_USER_REGION_INIT(handle, name, type)
- #define SCOREP_USER_REGION_END(handle) SCOREP_User_RegionEnd( handle );
- #define SCOREP_USER_FUNC_BEGIN()
- #define SCOREP_USER_FUNC_END() SCOREP_User_RegionEnd( scorep_user_func_handle );
- #define SCOREP_USER_GLOBAL_REGION_DEFINE(handle) SCOREP_User_RegionHandle handle = SCOREP_USER_INVALID_REGION;
- #define SCOREP_USER_GLOBAL_REGION_EXTERNAL(handle) extern SCOREP_User_Region↩ Handle handle;

#### Macros for parameter instrumentation

- #define SCOREP_USER_PARAMETER_INT64(name, value)
- #define SCOREP_USER_PARAMETER_UINT64(name, value)
- #define SCOREP_USER_PARAMETER_STRING(name, value)

#### Macros to provide user metrics

- #define SCOREP_USER_METRIC_LOCAL(metricHandle)
- #define SCOREP_USER_METRIC_GLOBAL(metricHandle)
- #define SCOREP_USER_METRIC_EXTERNAL(metricHandle) extern SCOREP_SamplingSetHandle metricHandle;
- #define SCOREP_USER_METRIC_INIT(metricHandle, name, unit, type, context) SCOREP_User_Init↩ Metric( &metricHandle, name, unit, type, context );
- #define SCOREP_USER_METRIC_INT64(metricHandle, value)
- #define SCOREP_USER_METRIC_UINT64(metricHandle, value)
- #define SCOREP_USER_METRIC_DOUBLE(metricHandle, value)

#### C++ specific macros for region instrumentation

- #define SCOREP_USER_REGION(name, type)

**Macros for measurement control**

- #define SCOREP_RECORDING_ON() SCOREP_User_EnableRecording();
- #define SCOREP_RECORDING_OFF() SCOREP_User_DisableRecording();
- #define SCOREP_RECORDING_IS_ON() SCOREP_User_RecordingEnabled()

### I.3.1 Detailed Description

This file contains the interface for the manual user instrumentation.

## I.4 SCOREP_User_Types.h File Reference

This file contains type definitions for manual user instrumentation.

```
#include <scorep/SCOREP_PublicTypes.h>
```

### Macros

- #define SCOREP_USER_INVALID_PARAMETER -1
- #define SCOREP_USER_INVALID_REGION NULL

**Region types**

- #define SCOREP_USER_REGION_TYPE_COMMON 0
- #define SCOREP_USER_REGION_TYPE_FUNCTION 1
- #define SCOREP_USER_REGION_TYPE_LOOP 2
- #define SCOREP_USER_REGION_TYPE_DYNAMIC 4
- #define SCOREP_USER_REGION_TYPE_PHASE 8

**Metric types**

- #define SCOREP_USER_METRIC_TYPE_INT64 0
- #define SCOREP_USER_METRIC_TYPE_UINT64 1
- #define SCOREP_USER_METRIC_TYPE_DOUBLE 2

**Metric contexts**

- #define SCOREP_USER_METRIC_CONTEXT_GLOBAL 0
- #define SCOREP_USER_METRIC_CONTEXT_CALLPATH 1

### Typedefs

- typedef uint32_t SCOREP_User_MetricType
- typedef uint64_t SCOREP_User_ParameterHandle
- typedef struct SCOREP_User_Region * SCOREP_User_RegionHandle
- typedef uint32_t SCOREP_User_RegionType

### I.4.1 Detailed Description

This file contains type definitions for manual user instrumentation.

### I.4.2 Macro Definition Documentation

#### I.4.2.1 #define SCOREP_USER_INVALID_PARAMETER -1

Marks an parameter handle as invalid or uninitialized

#### I.4.2.2 #define SCOREP_USER_INVALID_REGION NULL

Value for uninitialized or invalid region handles

### I.4.3 Typedef Documentation

#### I.4.3.1 typedef uint32_t SCOREP_User_MetricType

Type for the user metric type

#### I.4.3.2 typedef uint64_t SCOREP_User_ParameterHandle

Type for parameter handles

#### I.4.3.3 typedef struct SCOREP_User_Region* SCOREP_User_RegionHandle

Type for region handles in the user adapter.

#### I.4.3.4 typedef uint32_t SCOREP_User_RegionType

Type for the region type

# Index