OmpSs-2 User Guide

Release

BSC Programming Models

Nov 27, 2018

CONTENTS

CHAPTER

ONE

INSTALLATION OF OMPSS-2

1.1 Preparation

You must first choose a directory where you will install OmpSs-2. In this document this directory will be referred to as the `TARGET` directory. We recommend you to set an environment variable `TARGET` with the desired installation directory. For instance:

```
$ export TARGET=$HOME/ompss-2
```

1.2 Installation of Extrae (optional)

This is just a quick summary of the installation of Extrae. For a more detailed information check Extrae Homepage

1. Get Extrae from https://tools.bsc.es/downloads (choose *Source tarball* of Extrae tool).
2. Unpack the tarball and enter the just created directory:

   ```
   $ tar xfz extrae-xxx.tar.gz
   $ cd extrae-xxx
   ```

3. Configure it:

   ```
   $ ./configure --prefix=$TARGET
   ```

4. Build and install:

   ```
   $ make
   $ make install
   ```

Note: Extrae may have other packages's dependences and may use several options with the configure script. Do not hesitate to check the Extrae User's Manual

1.3 Installation of Mercurium compiler

You can find the build requirements, the configuration flags and the instructions to build Mercurium in the following link: https://github.com/bsc-pm/mcxx/blob/master/README.md

1.4 Installation of Nanos6

Nanos6 is a runtime that implements the OmpSs-2 parallel programming model, developed by the Programming Models group at the Barcelona Supercomputing Center.

Nanos6 can be obtained from the github public repository or by contacting us at pm-tools@bsc.es.

1.4.1 Build requirements

To install Nanos6 the following tools and libraries must be installed:

1. automake, autoconf, libtool, make and a C and C++ compiler
2. boost >= 1.59
3. hwloc
4. numactl
5. Finally, it's highly recommended to have a installation of Mercurium with OmpSs-2 support enabled. When installing OmpSs-2 for the first time, you can break the chicken and egg dependence between Nanos6 and Mercurium in both sides: on one hand, you can install Nanos6 without specifying a valid installation of Mercurium. On the other hand, you can install Mercurium without a valid installation of Nanos6 using the `--enable-nanos6-bootstrap` configuration flag.

1.4.2 Optional libraries and tools

In addition to the build requirements, the following libraries and tools enable additional features that are useful for debugging and analysing the performance of applications:

1. extrae to generate execution traces for offline performance analysis with paraver
2. elfutils and libunwind to generate sample-based profiling
3. graphviz and pdfjam or pdfjoin from TeX to generate graphical representations of the dependency graph
4. parallel to generate the graph representation in parallel
5. PAPI to generate statistics that include hardware counters

1.4.3 Build procedure

Nanos6 uses the standard GNU automake and libtool toolchain. When cloning from a repository, the building environment must be prepared through the following command:

```
$ autoreconf -f -i -v
```

When the code is distributed through a tarball, it usually does not need that command.

Then execute the following commands:

```
$ ./configure --prefix=TARGET ...other options...
$ make all check
$ make install
```

where `TARGET` is the directory into which to install Nanos6.

The configure script accepts the following options:

1. `--with-nanos6-mercurium=prefix` to specify the prefix of the Mercurium installation
2. `--with-boost` to specify the prefix of the Boost installation
3. `--with-libunwind=prefix` to specify the prefix of the libunwind installation
4. `--with-papi=prefix` to specify the prefix of the PAPI installation
5. `--with-libnuma=prefix` to specify the prefix of the numactl installation
6. `--with-extrae=prefix` to specify the prefix of the extrae installation

The location of elfutils and hwloc is always retrieved through pkg-config. The location of PAPI can also be retrieved through pkg-config if it is not specified through the `--with-papi` parameter. If they are installed in non-standard locations, pkg-config can be told where to find them through the `PKG_CONFIG_PATH` environment variable. For instance:

```
$ export PKG_CONFIG_PATH=$HOME/installations-mn4/elfutils-0.169/lib/pkgconfig:/apps/
↪HWLOC/2.0.0/INTEL/lib/pkgconfig:$PKG_CONFIG_PATH
```

After Nanos6 has been installed, it can be used by compiling your C, C++ and Fortran codes with Mercurium using the `--ompss-2` flag. Example:

```
$ mcc -c --ompss-2 a_part_in_c.c
$ mcxx -c --ompss-2 a_part_in_c_plus_plus.cxx
$ mcxx --ompss-2 a_part_in_c.o a_part_in_c_plus_plus.o -o app
```

1.4.4 Configuring GIT (contributing to github)

Please set up the following git configuration variables:

- user.name
- user.email

In addition we strongly suggest you to also set up the follwing pairs of variables and values:

- rebase.stat=true
- pull.rebase=true
- branch.autosetuprebase=always
- diff.submodule=log
- fetch.recursesubmodules=true
- status.submodulesummary=true
- rerere.enabled=true

CHAPTER

TWO

COMPILER OPTIONS

2.1 Mercurium drivers

The list of available drivers can be found here: https://github.com/bsc-pm/mcxx/blob/master/doc/md_pages/profiles.md

2.2 Common compilation flags

Usual flags like `-O`, `-O1`, `-O2`, `-O3`, `-D`, `-c`, `-o`, ... are recognized by Mercurium.

Almost every Mercurium-specific flag is of the form `--xxx`.

Mercurium drivers are deliberately compatible with `gcc`. This means that flags of the form `-fXXX`, `-mXXX` and `-Wxxx` are accepted and passed onto the backend compiler without interpretation by Mercurium drivers.

> **Warning:** In GCC a flag of the form `-fXXX` is equivalent to a flag of the form `--XXX`. This is **not** the case in Mercurium.

2.2.1 Getting command line help

You can get a summary of all the flags accepted by Mercurium using `--help` with any of the drivers:

```
$ mcc --help
Usage: mcc options file [file..]
Options:
  -h, --help               Shows this help and quits
  --version                Shows version and quits
  --v, --verbose           Runs verbosely, displaying the programs
                           invoked by the compiler
...
```

2.2.2 Passing vendor-specific flags

While almost every `gcc` of the form `-fXXX` or `-mXXX` can be passed directly to a Mercurium driver, some other vendor-specific flags may not be well known or be misunderstood by Mercurium. When this happens, Mercurium has a generic way to pass parameters to the backend compiler and linker.

--Wn,<comma-separated-list-of-flags> Passes comma-separated flags to the native compiler. These flags are used when Mercurium invokes the backend compiler to generate the object file (`.o`)

--Wl,<comma-separated-list-of-flags> Passes comma-separated-flags to the linker. These flags are used when Mercurium invokes the linker

--Wp,<comma-separated-list-of-flags> Passes comma-separated flags to the C/Fortran preprocessor. These flags are used when Mercurium invokes the preprocessor on a C or Fortran file.

These flags can be combined. Flags `--Wp,a --Wp,b` are equivalent to `--Wp,a,b`. Flag `--Wnp,a` is equivalent to `--Wn,a --Wp,a`

Important: Do not confuse `--Wl` and `--Wp` with the `gcc` similar flags `-Wl` and `-Wp` (note that `gcc` ones have a single `-`). The latter can be used with the former, as in `--Wl,-Wl,muldefs`. That said, Mercurium supports `-Wl` and `-Wp` directly, so `-Wl,muldefs` should be enough.

2.3 Compile OmpSs-2 programs

For OmpSs-2 programs that run in SMP or NUMA systems, you do not have to do anything. Just pick one of the drivers above.

Following is a very simple OmpSs-2 program in C:

```
/* test.c */
#include <stdio.h>

int main(int argc, char *argv[])
{
  int x = argc;
  #pragma oss task inout(x)
  {
    x++;
  }
  #pragma oss task in(x)
  {
    printf("argc + 1 == %d\n", x);
  }
  #pragma oss taskwait
  return 0;
}
```

Compile it using *mcc*:

```
$ mcc -o test --ompss-2 test.c
Nanos++ prerun
Nanos++ phase
```

Important: Do not forget the flag `--ompss-2` otherwise your program will be compiled without parallel support.

And run it:

```
$ ./test
argc + 1 == 2
```

2.3.1 Compile OpenMP programs with OmpSs-2 support

Warning: Experimental flag. There are no guarantees the generated code behaves as the OpenMP program.

`--openmp-compatibility` will support some constructs of OpenMP (e.g. taskloop, parallel for, etc.)

2.4 Problems during compilation

While we put big efforts to make a reasonably robust compiler, you may encounter a bug or problem with Mercurium.

There are several errors of different nature that you may run into.

- Mercurium ends abnormally with an internal error telling you to open a ticket.
- Mercurium does not crash but gives an error on your input code and compilation stops, as if your code were not valid.
- Mercurium does not crash, but gives an error involving an `internal-source`.
- Mercurium generates wrong code and native compilation fails on an intermediate file.
- Mercurium forgets something in the generated code and linking fails.

2.4.1 How can you help us to solve the problem quicker?

In order for us to fix your problem we need the *preprocessed* file.

If your program is C/C++ we need you to do:

1. Figure out the compilation command of the file that fails to compile. Make sure you can replicate the problem using that compilation command alone.
2. If your compilation command includes `-c`, replace it by `-E`. If it does not include `-c` simply add `-E`.
3. If your compilation command includes `-o file` (or `-o file.o`) replace it by `-o file.ii`. If it does not include `-o`, simply add `-o file.ii`.
4. Now run the compiler with this modified compilation command. It should have generated a `file.ii`.
5. These files are usually very large. Please compress them with `gzip` (or `bzip2` or any similar tool).

Send us an email to pm-tools at bsc.es with the error message you are experiencing and the (compressed) preprocessed file attached.

If your program is Fortran just the input file may be enough, but you may have to add all the `INCLUDE`d files and modules.

CHAPTER

THREE

RUNTIME OPTIONS

3.1 Common runtime options

3.1.1 Determining the number of CPUs

Nanos6 applications can be executed as is. The number of cores that are used is controlled by running the application through the `taskset` command. For instance:

```
$ taskset -c 0-2,4 ./app
```

would run `app` on cores 0, 1, 2 and 4.

In addition, the runtime comes with several variants that can be selected through the NANOS6 environment variable.

3.1.2 Stack Size

Nanos6 by default allocates stacks of 8 MB for its worker threads. In some codes this may not be enough. For instance, when converting Fortran codes, some global variables may need to be converted into local variables. This may increase substantially he amount of stack required to run the code and may surpass the space that is available. To solve that problem, the stack size can be set through the `NANOS6_STACK_SIZE` environment variable. Its value is expressed in bytes but it also accepts the `K`, `M`, `G`, `T` and `E` suffixes, that are interpreted as power of 2 multipliers.

Example:

```
$ export NANOS6_STACK_SIZE=16M
```

3.1.3 Values for the NANOS6 Environment Variable

The `NANOS6` environment variable selects the variant of the runtime that will be used. Currently it supports the following variants:

optimized This is the default value and selects the standard runtime based on pthreads.

debug Runtime compiled without optimization and with all assertions turned on.

extrae Instrumented to produce paraver traces. See: *Generating extrae traces*

verbose Instrumented to emit a log of the execution. See: *Verbose instrumentation*

verbose-debug Instrumented to emit a log of the execution and compiled without optimization and with all assertions turned on. See: *Verbose instrumentation*

graph Instrumented to produce a graph of the execution. Only practical for small graphs. See: *Generating a graphical representation of the dependency graph*

profile Instrumented to produce a function and source code execution profile. See: *Sampling-based profiling*

stats Instrumented to produce a summary of metrics of the execution. See: *Obtaining statistics*

stats-papi Instrumented to produce a summary of metrics of the execution including hardware counters. See: *Obtaining statistics*

3.1.4 Loading runtime verbosity

The `NANOS6_LOADER_VERBOSE` environment controls the verbosity of the Nanos6 Loader. By default (`0` value), it is quiet. If the environment variable has value `1` it will emit to standard error the actions that it takes and their outcome.

By default the loader will attempt to load the actual runtime library from the path determined by the operating system (taking into account the rpath and the `LD_LIBRARY_PATH` environment variable). If it fails to load the library, then it will attempt to locate the library at the same location as the nanos6 loader.

The default search path can be overriden through the `NANOS6_LIBRARY_PATH` environment variable. If it exists the first attempt at loading the runtime will be performed at the directory specified in that variable. The loader does not accept multiple directories in that variable.

The nanos6 loader resolves the addresses of the API functions to the actual runtime implementation. In addition it also checks for the implementation of some features, and if they are not found, it will either complain of emit a warning an fall back to a compatible but less powerful implementation. More specifically, the loader accepts running applications that make use of weak dependencies and will fall back to strong dependencies if the runtime does not have support for them.

3.1.5 Common environment variables

The following environment variables are valid for the runtime implementations.

- `NANOS6_HANDLE_SIGINT`: (default value: 0), Intercept the SIGINT signal to perform an unclean shutdown that does actually finalize the instrumentation (if any).
- `NANOS6_HANDLE_SIGTERM`: (default value: 0), Intercept the SIGTERM signal to perform an unclean shutdown that does actually finalize the instrumentation (if any).
- `NANOS6_HANDLE_SIGQUIT`: (default value: 0), Intercept the SIGQUIT signal to perform an unclean shutdown that does actually finalize the instrumentation (if any).

The `NANOS6_HANDLE_SIGINT`, `NANOS6_HANDLE_SIGTERM` and `NANOS6_HANDLE_SIGQUIT` environment variables are useful to get the output of the instrumentation at a given point in the execution. For instance, if the application misbehaves, running it with `NANOS6_HANDLE_SIGINT=1` with an instrumented version of the runtime and pressing Ctrl-C at the precise moment will terminate the application with the instrumentation information generated up to that point.

Alternatively, if the misbehavior is deterministic, the code may perform a call to raise(`SIGINT`) to produce the same effect programmatically.

3.2 Runtime task scheduler

The scheduler can be specified through the `NANOS6_SCHEDULER` environment variable. Currently it accepts the following values:

default, priority The default priority-aware scheduler with one immediate successor reservation per CPU

naive A very simple scheduler in LIFO mode

fifo A very simple scheduler in FIFO mode

immediatesuccessor A scheduler that reserves an immediate successor for each CPU

iswp A scheduler that reserves an immediate successor for each CPU and that when starved, leaves one thread polling for new work.

iswpfifo A scheduler that reserves an immediate successor for each CPU and that when starved, leaves one thread polling for new work. This is the FIFO version.

3.3 Runtime variants

3.3.1 Verbose instrumentation

To enable verbose logging, run the application with the `NANOS6` environment variable.

By default it generates a lot of information. This is controlled by the `NANOS6_VERBOSE` environment variable, which can contain a comma separated list of areas. The areas are the following:

AddTask Task creation

Blocking Blocking and unblocking within a task through calls tot he blocking API

ComputePlaceManagement Starting and stopping compute places (CPUs, GPUs, ...)

DependenciesByAccess Dependencies by their accesses

DependenciesByAccessLinks Dependencies by the links between the accesses to the same data

DependenciesByGroup Dependencies by groups of tasks that determine common predecessors and common successors

LeaderThread Execution of the leader thread.

LoggingMessages Additional logging messages

TaskExecution Task execution

TaskStatus Task status transitions

TaskWait Entering and exiting taskwaits

ThreadManagement Thread creation, activation and suspension

UserMutex User-side mutexes (critical)

The case is ignored, and the `all` keyword enables all of them. Additionally, and area can have the `!` prepended to it to disable it. For instance, `NANOS6_VERBOSE=AddTask,TaskExecution,TaskWait` is a good starting point.

The default value is `all,!ComputePlaceManagement,!DependenciesByAccess,!DependenciesByAccessLinks,!DependenciesByGroup,!LeaderThread,!TaskStatus,!ThreadManagement`.

By default events are recorded with their timestamp and ordered accordingly. This can be disabled by setting the `NANOS6_VERBOSE_TIMESTAMPS` environment variable to `0`.

The output is emitted by default to standard error, but it can be sent to a file by specifying it through the `NANOS6_VERBOSE_FILE` environment variable. Also the `NANOS6_VERBOSE_DUMP_ONLY_ON_EXIT` can be set to `1` to delay the output to the end of the program to avoid getting it mixed with the output of the program.

3.3.2 Generating a graphical representation of the dependency graph

To generate the graph, run the application with the `NANOS6` environment variable set to `graph`.

The graph instrumentation creates a subdirectory with the graph in several stages and a script that can be executed to generate a PDF that combines each step in a different page. The progress of the execution can be visualized by advancing the pages. This PDF is intended to be viewed in whole page mode, instead of continuous mode.

The most graph instrumentation environment variables are boolean variables and can take either the 0 or the 1 value.

`NANOS6_GRAPH_SHORTEN_FILENAMES` (default value: 0) When generating nodes, do not emit the directory together with the source code file name

`NANOS6_GRAPH_SHOW_ALL_STEPS` (default value: 0) Instead of trying to collapse in one step as many related changes as possible, show one at a time.

`NANOS6_GRAPH_DISPLAY` (default value: 0) Automatically process the graph through graphviz and display it.

`NANOS6_GRAPH_DISPLAY_COMMAND` (default value: xdg-open or evince or okular or acroread) Command to display the final PDF of the graph. Only effective if NANOS6_GRAPH_DISPLAY is 1.

`NANOS6_GRAPH_SHOW_LOG` (default value: 0) Emit a table next to the graph with a description of the changes in each frame.

In addition, the following advanced environment variables can be used to debug the runtime:

`NANOS6_GRAPH_SHOW_DEPENDENCY_STRUCTURES` (default value: 0) Show the internal data structures that determine when tasks are ready.

`NANOS6_GRAPH_SHOW_SPURIOUS_DEPENDENCY_STRUCTURES` (default value: 0) Do not hide internal data structures that do not determine dependencies or that are redundant by transitivity.

`NANOS6_GRAPH_SHOW_DEAD_DEPENDENCY_STRUCTURES` (default value: 0) Do not hide the internal data structures after they are no longer relevant.

`NANOS6_GRAPH_SHOW_REGIONS` (default value: 0) When showing internal data structures, include the information about the range of data or region that is covered.

3.3.3 Sampling-based profiling

To enable sampling-based profiling, run the application with the `NANOS6` environment variable set to `profile`.

The profile instrumentation uses the following environment variables:

- `NANOS6_PROFILE_NS_RESOLUTION`: (default value: 1000), sampling interval in nanoseconds
- `NANOS6_PROFILE_BACKTRACE_DEPTH`: (default value: 4), number of stack frames to collect (excluding inlines) in each sample.
- `NANOS6_PROFILE_BUFFER_SIZE`: (default value: 1000000000), number of sampling events to preallocate together in a chunk. The default value corresponds to 1 second of samples.

At the end of the execution, the runtime generates four files that contain entries sorted by decreasing frequency. Their first column contains the sample count, and the rest, the actual entry values. Their contents are the following:

line-profile-PID.txt: Source code lines

function-profile-PID.txt: Function names

inline-profile-PID.txt: Function names and source code lines including inlines > Since the sampling is performed over the return addresses in the stack, if the compiler performs inlining, a given address can correspond to several functions. This file shows for the number of samples that have the same associated source code lines.

backtrace-profile-by-line-PID.txt: Function names and source code lines including inlines of a full backtrace > Shows the number of samples that have a full backtrace that corresponds to the same exact source code lines.

backtrace-profile-by-address-PID.txt: Function names and source code lines including inlines of a full backtrace > Shows the number of samples that have a full backtrace with the same exact return addresses.

When compiling, Mercurium performs transformations to the original source code. At this time, Mercurium cannot preserve the original source code lines and function names. Hence, the outputs of the profiler are based on the transformed code. However, the transformed source code can be preserved by passing the `-keep` parameter to Mercurium.

Mercurium generates additional functions that wrap the task code. These appear in the backtraces and their names begin with `nanos6_ol_` and `nanos6_unpack_` followed by a number.

3.3.4 Obtaining statistics

To enable collecting statistics, run the application with the `NANOS6` environment variable set to either `stats` or `stats-papi`. The first collects timing statistics and the second also records hardware counters.

By default, the statistics are emitted standard error when the program ends. The output can be sent to a file through the `NANOS6_STATS_FILE` environment variable.

The contents of the output contains the average for each task type and the total task average of the following metrics:

- Number of instances
- Mean instantiation time
- Mean pending time (not ready due to dependencies)
- Mean ready time
- Mean execution time
- Mean blocked time (due to a critical or a taskwait)
- Mean zombie time (finished but not yet destroyed)
- Mean lifetime (time between creation and destruction)

The output also contains information about:

- Number of CPUs
- Total number of threads
- Mean threads per CPU
- Mean tasks per thread
- Mean thread lifetime
- Mean thread running time

Most codes consist of an initialization phase, a calculation phase and final phase for verification or writing the results. Usually these phases are separated by a taskwait. The runtime uses the taskwaits at the outermost level to identify phases and emit individual metrics for each phase.

3.3.5 Generating extrae traces

To generate an paraver trace using extrae, the `NANOS6` environment variable must be set to `extrae` before running the application.

By default, the runtime will generate a trace as if `EXTRAE_ON` was set to 1. In addition, the `EXTRAE_CONFIG_FILE` environment variable can be set to an extrae configuration file for fine tuning, for instance, to enable recording hardware counters. See: the extrae documentation.

The amount of information generated in the extrae traces can be controlled through the `NANOS6_EXTRAE_DETAIL_LEVEL` environment variable. Its value determines a level of detail that goes from 0, which is the least detailed, up to 8. The default level is 1. Lower levels incur in less overhead and produce smaller traces. Higher levels have more overhead, produce bigger traces, but are more precise and comtain more information. The information generated at each level is incremental and is the following:

Level 0 Basic level

Includes basic information about the runtime state and the execution of tasks.

Counters about the number of tasks in the system are approximate and are sampled at periodic intervals.

Level 1 Default level

Counters about the number of tasks in the system are precise in terms of time and value.

Adds communication records that link the point of instantiation of a task to the point where they start their execution.

Adds communication records that link the point where a task get blocked to the point where a task unblocks it (makes it ready), and from that point to the point where the task actually resumes its execution.

Adds communication records that show task dependency relations. The set of predecessors a task that is shown is limited to the set of tasks that have not finished their execution once said task has been instantiated. The links go from the end of the execution of a predecessor to the start of the execution of the successor.

Levels 2 to 7 Unused

Currently they do not add further information.

Level 8 Very detailed level

Adds communication records that link the end of the execution of a task to the point where its parent returns from a taskwait. Similarly to the graph information, the trace only contains links for the tasks that have not finished once their parent enters the taskwait. This information is only available at level 8, since it may make the trace significantly bigger.

The runtime installation contains a set of already made paraver configuration files at the following subdirectory: `share/doc/nanos6/paraver-cfg/nanos6`

Support for hardware counters is enabled through file specified in the `EXTRAE_CONFIG_FILE` environment variable. The procedure is explained in the extrae documentation. However, the `NANOS6_EXTRAE_AS_THREADS` environment variable must also be set to 1. This is a temporary measure that is needed to produce correct hardware counter information. The resulting trace will expose the actual runtime threads, as opposed to the CPU view that is generated by default.

3.4 Debugging

By default, the runtime is optimized for speed and will assume that the application code is correct. Hence, it will not perform most validity checks. To enable validity checks, run the application with the `NANOS6` environment variable set to `debug`. This will enable many internal validity checks that may be violated when the application code is incorrect. In the future we may include a validation mode that will perform extensive application code validation.

To debug an application with a regular debugger, please compile its code with the regular debugging flags and also the `-keep` flag. This flag will force Mercurium to dump the transformed code in the local file system, so that it will be available for the debugger.

To debug dependencies, it is advised to reduce the problem size so that very few tasks trigger the problem, and then use let the runtime make a graphical representation of the dependency graph. See *Generating a graphical representation of the dependency graph*.

Processing the NANOS6 environment variable involves selecting at run time a runtime compiled for the corresponding instrumentation. This part of the bootstrap is performed by a component of the runtime called "loader. To debug problems due to the installation, run the application with the NANOS6_LOADER_VERBOSE environment variable set to any value.

3.4.1 Runtime information

Information about the runtime may be obtained by running the application with the NANOS6_REPORT_PREFIX environment variable set, or by invoking the following command:

```
$ nanos6-info --runtime-details
Runtime path /opt/nanos6/lib/libnanos6-optimized.so.0.0.0
Runtime Version 2017-11-07 09:26:03 +0100 5cb1900
Runtime Branch master
Runtime Compiler Version g++ (Debian 7.2.0-12) 7.2.1 20171025
Runtime Compiler Flags -DNDEBUG -Wall -Wextra -Wdisabled-optimization -Wshadow -
↪fvisibility=hidden -O3 -flto
Initial CPU List 0-3
NUMA Node 0 CPU List 0-3
Scheduler priority
Dependency Implementation linear-regions-fragmented
Threading Model pthreads
```

The NANOS6_REPORT_PREFIX environment variable may be defined as an empty string or it may contain a string that will be prepended to each line. For instance, it can contain a sequence that starts a comment in the output of the program. Example:

```
$ NANOS6_REPORT_PREFIX="#" ./app
Some application output ...
#   string  version 2017-11-07 09:26:03 +0100 5cb1900       Runtime Version
#   string  branch  master      Runtime Branch
#   string  compiler_version    g++ (Debian 7.2.0-12) 7.2.1 20171025        Runtime␣
↪Compiler Version
#   string  compiler_flags  -DNDEBUG -Wall -Wextra -Wdisabled-optimization -Wshadow -
↪fvisibility=hidden -O3 -flto       Runtime Compiler Flags
#   string  initial_cpu_list    0-3     Initial CPU List
#   string  numa_node_0_cpu_list    0-3     NUMA Node 0 CPU List
#   string  scheduler   priority        Scheduler
#   string  dependency_implementation   linear-regions-fragmented       Dependency␣
↪Implementation
#   string  threading_model pthreads        Threading Model
```

INDEX

R

S

V

X