Periscope User’s Guide

PTF Version: 2.1
Periscope Version: xx.xx

Michael Gerndt, Anca Berariu

28.01.2016
Contents

1 Introduction 2

2 Quick Start 4
  2.1 Installation ................................................. 4
    2.1.1 periscope configuration file ..................... 5
    2.1.2 SSH access ........................................... 5
    2.1.3 GUI .................................................. 6
  2.2 Basic analysis run ........................................ 6
    2.2.1 Specify the phase region in NPB-MZ BT ............ 6
    2.2.2 Modify the Makefile ................................. 7
    2.2.3 Build the application ............................... 8
    2.2.4 Start Periscope analysis ............................ 8
    2.2.5 Explore the results ................................. 8

3 Analysis Flow within Periscope 9
  3.1 Specification of a phase region ....................... 9
  3.2 Enabling instrumentation - scorep .................... 11
  3.3 Starting performance analysis - psc_frontend .......... 13
  3.4 Exploring the results - GUI ........................... 15

4 Performance Tuning with Periscope 16
  4.1 Tuning plugins ........................................... 16
  4.2 Tuning advice ............................................ 18
  4.3 The tuning flow ......................................... 18
  4.4 Uninstrumented applications ............................ 19

5 Configuration Options 20
  5.1 Environment Variables ................................... 20
  5.2 The frontend - psc_frontend ............................ 21

6 Advanced user information - technical details 25
  6.1 Agent hierarchy ....................................... 25

7 Known Issues 27
Chapter 1

Introduction

Periscope is a scalable automatic performance analysis tool currently under development at Technische Universität München and is part of the Periscope Tuning Framework (PTF), along with tools like Pathway and tuning plugins.

Periscope provides two main functionalities for Fortran and C/C++ applications: performance analysis and performance tuning.

Performance analysis is performed at runtime, using an iterative approach. There is a starting set of performance properties, which is then refined based on the measurements and the chosen search strategy. In the end, the appropriate set of performance properties is provided for the application being analysed. The search threshold, the confidence value, and the severity are defined by means of a formal specification of the properties.

Based on expert knowledge, Periscope uses several strategies to identify possible performance issues. Such strategies include exploiting parallel MPI or OpenMP regions, as well as system specific approaches, like for example for Power6 machines.

The second functionality, performance tuning, is provided through the tuning framework. Periscope offers the necessary support for measurements and search logic for a series of tuning plugins. Different application and environment setups are tested within the plugins and the best configuration is provided as an advice at the end of the tuning.

Periscope consists of four main components: the frontend, the hierarchy of communication and analysis agents, the monitoring library and the GUI.

- The frontend is responsible for starting both the application to be analysed, as well as all the internal components of Periscope. All settings regarding the execution of Periscope can be selected by means of command-line parameters of the frontend process.
• The *agent hierarchy* is transparent for the common users. At the bottom layer of the hierarchy there are the analysis agents. They control and configure the measurements for each application node/process. They can start, halt, or resume the execution, and they also retrieve the performance data. The strategy is communicated upon startup by the frontend and at the end of the local search, the performance properties are communicated back to the frontend via the agent hierarchy.

• The *monitoring library* is also transparent to the user and it provides the measurement and communication layer between the application being tested and the performance tool.

• The *GUI* is used to visualise and explore the performance results. It is an Eclipse plugin which can be easily used to identify the most severe performance properties, as well as the corresponding source lines responsible for the performance issues.

**Periscope Tuning Framework**

Alongside Periscope, the Periscope Tuning Framework (PTF) also provides PAThWay, a workflow management tool for HPC experiments, as well as a series of tuning plugins for automatic tuning of applications.
Chapter 2

Quick Start

2.1 Installation

Periscope can be installed from the source files, following the common process of configuring and building using Autotools.

Please check the *Periscope Installation Manual* for a thorough guide on how to install Periscope on your machine. The basic installation steps are:

1. check and install prerequisites: ACE, Boost, etc.
2. checkout the source files from the Periscope repository
   
   \$ git clone https://periscope.in.tum.de/git/Periscope.git
   
   periscope
3. configure your installation choosing appropriate options\(^1\) for example:
   
   \$ configure --prefix=$HOME/install/psc
4. build the files
   
   \$ make -j 16
5. install the files
   
   \$ make install

If you are using SuperMUC, Periscope is already installed on the system. In order to use it, you have to add to your `.bashrc` file:

\$ module load periscope

\(^1\)Please refer to the *PTF Installation Manual* for further details regarding available options.
and then issue in your home directory:

$ source .bashrc

Note: Please make sure to add the command for loading the periscope module into your .bashrc. Just issuing the command at the command line is not going to work properly.

### 2.1.1 .periscope configuration file

Before using Periscope, the .periscope setup file has to be created in your home directory. You may create a new one, or copy it from the Periscope installation directory:

$ cp $PSC_ROOT/templates/.periscope

The setup file contains a list of `<option>=<value>` pairs, as follows:

- `MACHINE = SuperMUC`
- `SITE = LRZ`
- `REGSERVICE_HOST = localhost`
- `REGSERVICE_PORT = 50001`
- `REGSERVICE_HOST_INIT = localhost`
- `REGSERVICE_PORT_INIT = 50001`
- `APPL_BASEPORT = 51000`
- `AGENT_BASEPORT = 50002`

If running on your local machine only, then the `MACHINE` option above should be set to `localhost`.

MACHINE = localhost

Please refer to the PTF Periscope Installation Manual for a detailed description on how to choose the proper option values for your particular system.

### 2.1.2 SSH access

In order to run Periscope, a private key based ssh access has to be provided on the machine running the tool. If not already configured, you can do so in few steps:

1. $ mkdir ~/.ssh
2. $ cd ~/.ssh
3. $ ssh-keygen -t rsa -N '' -f id_rsa
4. $ cat id_rsa.pub >> authorized_keys
5. $ chmod 600 authorized_keys

The ssh access is not required if running on your local machine, i.e. the MACHINE option is set to localhost in your .periscope file.

2.1.3 GUI

The Periscope GUI used for analysing the performance measurements is provided as an Eclipse plugin. You can install the GUI from this location following the common plugin installation process in Eclipse.[2]

2.2 Basic analysis run

Having Periscope properly installed, there are only few steps required for a basic analysis of a test application:

1. specify a phase region by instrumenting the source code of the application with the Score-P pragma;
2. modify the Makefile to enable instrumentation;
3. build the application;
4. start the analysis;
5. visualize and explore the performance results.

For the remainder of this section we consider as the test application the NPB-MZ BT benchmark.[3]

2.2.1 Specify the phase region in NPB-MZ BT

Periscope uses an iterative analysis approach. It starts first with a set of performance properties which are measured for the test application throughout an experiment run. Based on the measurements result, it then determines new candidate properties which are going to be evaluated in the next experiment. The iteration stops when there are no new candidate properties.

CHAPTER 2. QUICK START

If the test application has a repetitive region, like for example the body of a main loop, then the consecutive experiments could be performed without the need of restarting the entire application. In order to do so, the repetitive region has to be marked in the source code as a phase region.

For the BT application, the phase region can be defined in file bt.f, lines 188 to 198, by inserting score-p pragma as shown below:

```c
---
c    start the benchmark time step loop
c---

  do step = 1, niter
c      lines omitted here ...

SCOREP_USER_OA_PHASE_BEGIN( scorep_bt_loop , "ITERATION" ,
SCOREP_USER_REGION_TYPE_COMMON)
call exch_qbc(u,qbc,nx,nxmax,ny,nz)

  do zone = 1, num_zones
     call adi(rho_(start1(zone)),us(start1(zone)) , $
     vs(start1(zone)),ws(start1(zone)) , $
     qs(start1(zone)),square(start1(zone)) , $
     rhs(start5(zone)),forcing(start5(zone)) , $
     u(start5(zone)) , $
     nx(zone),nxmax(zone),ny(zone),nz(zone))
  end do
SCOREP_USER_OA_PHASE_END( scorep_bt_loop )
end do
```

2.2.2 Modify the Makefile

In order to enable performance measurements, the test application has to be instrumented by the performance tool. To enable instrumentation, one has to substitute the compile/link commands usually defined in the `Makefile`.

For NPB-MZ BT, one should edit the `config/make.def` file and update the `F77` variable as follows:

```
#--------------------------------------------------------
# This is the fortran compiler used for fortran programs
#--------------------------------------------------------
F77=scorep --user ./bin/bt-mz.$(CLASS).$(NPROCS) -mpif77
# This links fortran programs; usually the same as $(F77)
```
FLINK=$(F77)

2.2.3 Build the application

After the phase region was defined and the build command was adjusted, one can continue with the common build process of the test application.

For the NPB-MZ BT example, one should go to the root directory of the NPB-MZ series and issue:

\[
\begin{align*}
$ & \text{make clean} \\
$ & \text{make bt-mz CLASS=C NPROCS=16}
\end{align*}
\]

2.2.4 Start Periscope analysis

Periscope can be started via its frontend `psc_frontend`. Upon calling the executable with proper parameters, both Periscope’s internal components as well as the test application are being started and the performance measurements are then carried out.

For the NPB-MZ BT example, one should go to the `bin` directory and then call `psc_frontend` as follows:

\[
\begin{align*}
$ & \text{psc_frontend --apprun=./bt-mz.C.16 --mpinumprocs=16 --strategy=MPI --force-localhost}
\end{align*}
\]

2.2.5 Explore the results

Upon successful termination, Periscope generates a `*.psc` results file. This is a standard XML file and could be opened using any text editor. Periscope provides a Graphical User Interface (GUI) with enhanced visualisation and exploration functionalities for working with these performance result files.

Having started Periscope like described above for the NPB-MZ BT benchmark, the `properties_.psc` should have been created into the same `bin` directory. Please follow the instructions in section 3.4 for opening this file within the GUI.
Chapter 3

Analysis Flow within Periscope

Periscope follows an iterative analysis approach: it determines performance properties based on measurements, decides on possible new candidate properties and then it performs again new experiments to measure the data required to check whether the candidate properties hold. See also the cycle depicted in Figure 3.1.

The number of experiments carried out in one run of Periscope depends on both the execution time of the application itself and also the performance issues it might exhibit.

The number of experiments carried out in one run of Periscope depends on the performance issues it might detect. Thus the total execution time of one Periscope analysis will depend on both the execution time of the application itself, as well as the amount and severity of detected performance issues.

3.1 Specification of a phase region

The performance measurements carried out within one experiment of the iterative analysis could be applied to either the entire application or only a particular execution phase or code region. Periscope offers the possibility to define such a phase region by means of manual instrumentation of the source code.

Please explore the Score-P user manual for the manual instrumentation in more detail. We only mention here that a phase region can be in terms of Periscope code instrumentation any regular user region.
A user region can be defined by inserting the following directives into the source code:

**Fortran:**
```fortran
program myProg
  SCOREP_USER_REGION_DEFINE( my_region_handle )
  SCOREP_USER_REGION_BEGIN( my_region_handle, "my_region",
    SCOREP_USER_REGION_TYPE_COMMON )
  S1
  S2
  ...
  SCOREP_USER_REGION_END( my_region_handle )
end program myProg
```

**C/C++:**
```c
void myfunc()
{
  SCOREP_USER_REGION_DEFINE( my_region_handle )
  SCOREP_USER_REGION_BEGIN( my_region_handle, "my_region",
    SCOREP_USER_REGION_TYPE_COMMON )
  S1
```
Periscope allows the specification of several user regions, but only one such region can be defined as the phase region.

If several user regions are defined, but none of them is specified as the phase region, then the behaviour of Periscope is undefined.

If only one user region is specified, then this is automatically defined as the phase region.

If no phase region is specified, Periscope will automatically restart the application to perform new experiments, until no new candidate properties are found and the search terminates.

The use of phase regions is strongly recommended:

- it reduces the overall execution time of the Periscope performance measurements;
- it delivers more accurate results, as measurements are only performed for relevant execution fragments.

The best example for a phase region is the body of the main loop of an application. It is common that scientific applications have a main loop iterating through time steps or grid elements. If such a repetitive region is defined in the source code as a phase region, then the experiments can be done during the same application run. The application is suspended at the beginning of the phase region and new measurements are requested. The application is then released and the analysis is started. When the application encounters again the end of the region, it is suspended and the measured values are retrieved.

### 3.2 Enabling instrumentation - scorep

Measuring performance of an application is commonly based on the ability of the performance tool to ”communicate” with the application at runtime. This can be achieved through the instrumentation of the application, i.e. inserting tool specific calls inside the source code or the compiled binary of the application. See also the right hand side of figure 3.1.

As Score-P is compiled with the application, eventually the application itself is responsible for performance property measurements. The application
CHAPTER 3. ANALYSIS FLOW WITHIN PERISCOPE

opens a socket to connect with Periscope. Periscope then asks the application for certain measurement data, or to apply a tuning action.

In order to enable instrumentation with Score-P, one needs to prepend the compiling and linking commands with the call to the scorep script. This could usually be done by editing the Makefile of the application.

For example, one should replace

```bash
mpif90 -c <args>
```

with

```bash
scorep <scorep_options> mpif90 -c <args>
```

for a Fortran code, and

```bash
mpicc -c <args>
```

with

```bash
scorep <scorep_options> mpicc -c <args>
```

for a C/C++ code.

Do not forget to change both the compiling and the linking commands.

Please note that the script recognizes the -c argument passed to the compiler itself and uses it to decide between the instrumentation and the linking steps. It is thus required that the respective test application is built in two distinct steps: compilation and linking.

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.

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:

```bash
scorep-config [--mpp=none|--mpp=mpi|--mpp=shm] [--thread=none|--thread=omp|--thread=pthread] --libs
```

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 3.2. The detailed explanation can be found in Score-P user guide.
CHAPTER 3. ANALYSIS FLOW WITHIN PERISCOPE

<table>
<thead>
<tr>
<th>Type of instrumentation</th>
<th>Instrumenter switch</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI</td>
<td>--mpp=mpi/</td>
<td>auto</td>
</tr>
<tr>
<td></td>
<td>--mpp=none</td>
<td></td>
</tr>
<tr>
<td>SHMEM</td>
<td>--mpp=shmem/</td>
<td>auto</td>
</tr>
<tr>
<td></td>
<td>--mpp=none</td>
<td></td>
</tr>
<tr>
<td>OpenCL</td>
<td>--opencl/</td>
<td>enabled</td>
</tr>
<tr>
<td></td>
<td>--noopencl</td>
<td></td>
</tr>
<tr>
<td>OpenMP</td>
<td>--thread=omp/</td>
<td>auto</td>
</tr>
<tr>
<td></td>
<td>--thread=none</td>
<td></td>
</tr>
<tr>
<td>Pthread</td>
<td>--thread=pthread</td>
<td>auto</td>
</tr>
<tr>
<td>Compiler</td>
<td>--compiler/</td>
<td>enabled</td>
</tr>
<tr>
<td></td>
<td>--nocompiler</td>
<td></td>
</tr>
<tr>
<td>PDT</td>
<td>--pdt/</td>
<td>disabled</td>
</tr>
<tr>
<td></td>
<td>--nopdt</td>
<td></td>
</tr>
<tr>
<td>POMP2</td>
<td>--pomp/</td>
<td>depends on OpenMP usage</td>
</tr>
<tr>
<td>user regions</td>
<td>--nopomp</td>
<td></td>
</tr>
<tr>
<td>Manual</td>
<td>--user/</td>
<td>disabled</td>
</tr>
<tr>
<td></td>
<td>--nouser</td>
<td></td>
</tr>
</tbody>
</table>

3.3 Starting performance analysis - psc_frontend

The Periscope performance measurement and analysis process can be started via the psc_frontend executable. For example:

$ psc_frontend --apprun=./bt-mz_C.16 --mpinumprocs=16
   --force-localhost --debug=1

All needed configuration options can be passed to Periscope by means of the command line parameters.

The mandatory parameters which are required for Periscope analysis to start are:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--apprun=&lt;command line&gt;</td>
<td>Specify the command line to start the application. It will be passed to the mpirun command. The executable specified in the command line must exist when Periscope is started.</td>
</tr>
</tbody>
</table>
CHAPTER 3. ANALYSIS FLOW WITHIN PERISCOPE

```
--mpinumprocs=<np>
```

Number of MPI processes for the application.

For serial applications, please set this value to 1. Periscope treats serial applications as 1-process MPI applications.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--debug=&lt;level&gt;</td>
<td>Level of debug output (default: 0).</td>
</tr>
<tr>
<td>--force-localhost</td>
<td>Locally start the agents instead of using SSH.</td>
</tr>
<tr>
<td>--strategy=&lt;strategy&gt;</td>
<td>Specify one of the following strategies: MPI, SCA, SCABF, P6, P6BF, P6BF_Memory, SCPS_BF, scalability_OMP. Please note: Some strategies are platform dependent (default: all).</td>
</tr>
<tr>
<td>--propfile=&lt;filename&gt;</td>
<td>Store the detected properties into filename (default: properties.psc)</td>
</tr>
<tr>
<td>--ompnumthreads=&lt;threads&gt;</td>
<td>Number of OpenMP threads (default: 1).</td>
</tr>
</tbody>
</table>

Please see table 5.2 for a complete list of options accepted by psc_frontend.

On startup, a hierarchy of analysis and communication agents is first created, then the application to be measured is started and the analysis agents attach to the application nodes. The performance data are gathered by means of the monitoring library and communicated to the low-level agents. There it is analysed using the strategy established at the beginning within the frontend and based on the results, the next step of the iterative analysis is established.

The final results are propagated through the agent hierarchy up to the frontend, which then stores them in the properties file.

The frontend is the control point of Periscope. Users can configure and direct the performance analysis process from here. The agent hierarchy and the monitoring library remain transparent to the common user.
3.4 Exploring the results - GUI

The frontend writes the found performance properties into a file called `properties.*` with the `.psc` extension. This file is in XML format and can be opened with any off-the-shelf text editor or a spreadsheet application.

Periscope also offers a Graphical User Interface (GUI) for an enhanced visualisation and exploration of the analysis results. It is an Eclipse based plugin, featuring a multi-functional table for displaying and organizing the textual data. Following functionalities are available:

- multiple criteria sorting algorithm
- complex categorization utility
- searching engine using regular expressions
- filtering operations
- direct navigation from the bottlenecks to their precise source location using the default IDE editor for that source file type (e.g. CDT/Pho-tran editor).

An outline view for the instrumented code regions that were used in an experiment is also available. The information it shows is a combination of the standard intermediate representation of the analyzed application and the distribution of its bottlenecks. The main goals of the view are to assist the navigation in the source code and attract developer’s attention to the most problematic code areas.

The multivariate statistical clustering is another key feature of the plug-in that enhances the scalability of the GUI and provides means of conducting Peta-scale performance analysis. It can effectively summarize the displayed information and identify a runtime behavior possibly hidden in the large amount of data.
Chapter 4

Performance Tuning with Periscope

Performance tuning using PTF (Periscope Tuning Framework) is based on the collaborative work performed by customized tuning plugins on the one side and Periscope as the host application of the plugins on the other side.

The high-level architecture of PTF can be seen in figure 4.1. Similar to using the analysis feature of PTF, users can start and configure the tuning process by calling the `psc_frontend` with appropriate parameters. The option enabling the tuning execution mode of Periscope is `--tune`:

```
$ psc_frontend --tune=<nameofplugin> ...
```

For example, the following will run compiler flags tuning (CFS) on the BT application:

```
psc_frontend --apprun="./bt-MZ.W" --mpinumprocs=1 --force-localhost --tune=compilerflags --cfs-config="cfs_config.cfg"
```

Depending on each particular plugin, there might be also other options available for configuration. Please consult the corresponding User’s Guide for details specific to each of the plugins.

All other components in figure 4.1 are transparent to the users of the plugins and of the PTF tuning feature.

4.1 Tuning plugins

For the current version, PTF provides the following tuning plugins:
Figure 4.1: Plugin architecture of the Periscope Tuning Framework.

**CFS:** the *Compiler Flags Selection* plugin tunes the application to find the combination of compiler flags with which the best execution time is achieved.

**DVFS:** the *Dynamic Voltage and Frequency Scaling* plugin tunes the energy consumption of an application.

**Master-Worker:** the *Master-Worker* plugin tunes the number of tasks and processes to be used by applications based on the master-worker paradigm.

**MPI Paramenters:** automatically optimizes the values of a user selected subset of MPI configuration parameters.

**Patterns:** the *Parallel Patterns* plugin works on applications using a *Pipeline-based* execution to determine the best combination of the pipeline stages.
4.2 Tuning advice

As a result of the tuning process, Periscope generates an XML file describing:

- the final tuning advice to be applied to the application
- the tuning scenarios which were used in searching the best advice
- other information specific to the tuning plugin, like, for example, the tuning parameters, the execution times, or the energy consumption.

4.3 The tuning flow

Being the host of the tuning plugins, Periscope provides several services to build a standard tuning flow.

Data model

The main components of the tuning data model are:

- **tuning parameters**: represent the parameters based on which a tuning of the application can be done. These are plugin dependent and their semantics is strictly defined in each plugin. For example, the CFS plugin uses compiler flags as tuning parameters, while the MPI Parameters plugin uses MPI related switches and parameters.
  
  For most plugins, the tuning parameters are the given by user input through a configuration file.

- **tuning scenario**: represents a combination of tuning parameters. The application is analysed by Periscope using one scenario at a time.
  
  Scenarios are computed internally based on a chosen search algorithm. Users can choose between different search algorithms, but cannot directly define tuning scenarios.

- **tuning space**: the set of all valid tuning scenarios.

- **analysis result**: the analysis result associated with one specific tuning scenario. Results are partially displayed in the final tuning advice provided by Periscope.
CHAPTER 4. PERFORMANCE TUNING WITH PERISCOPE

Operations

On the functional side, the tuning flow is supported by means two main operations:

search algorithm: the search algorithm generates the tuning space and delivers the next scenario to be evaluated. For most tuning plugins, users can choose the preferred search algorithm.

There are several search algorithms available: exhaustive search, individual search, random search and GDE3 search (one genetic algorithm).

pre-analysis: some plugins require an analysis step before the tuning process can start. The Periscope performance analysis feature is being used in this case.

Required pre-analysis is very much plugin specific. Please consult the given User’s Guide to see whether user input is possible for each particular case.

4.4 Uninstrumented applications

The CFS Plugin an the MPI Plugin also allow tuning of uninstrumented applications, but this is strongly not encouraged. When measuring performance for uninstrumented applications, Periscope relies exclusively on the data retrieved from the system. This mostly leads to inaccuracies, especially for applications with a short execution time. If one does want to use the uninstrumented version, this can be done by passing the --uninstrumented option to the psc_frontend process at the command line.
Chapter 5

Configuration Options

5.1 Environment Variables

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSC_ROOT</td>
<td>Root directory of the Periscope installation.</td>
</tr>
<tr>
<td>PERISCOPE_DEBUG</td>
<td>0..2</td>
</tr>
<tr>
<td></td>
<td>0=quiet</td>
</tr>
<tr>
<td></td>
<td>1=startup, found properties in each search</td>
</tr>
<tr>
<td></td>
<td>2=candidate properties and found properties in each strategy step</td>
</tr>
</tbody>
</table>
### 5.2 The frontend - `psc_frontend`

The frontend starts up the application and the agent hierarchy.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--apprun=&lt;appl cmdline&gt;</code></td>
<td>This is the command line used to start the application. It should be the same as in <code>mpirun -np &lt;procs&gt; &lt;appl cmdline&gt;</code>. This value is also used to determine the name of the SIR file, when <code>--sir</code> is missing. The executable specified in the command line must exist when Periscope is started. This is true also for the cases where the tuning feature of Periscope is used in combination with plugins which by themselves re-build the application from its source files (e.g. the CFS plugin).</td>
</tr>
<tr>
<td><code>--debug=level</code></td>
<td>Level of debugging. All debug output up to that level will be printed. Default: <code>PERISCOPE_DEBUG</code> or 0</td>
</tr>
<tr>
<td><code>--selective-debug=list</code></td>
<td>Individual debugging level names separated by comma</td>
</tr>
<tr>
<td><code>--delay=&lt;n&gt;</code></td>
<td>Number of phase executions that are skipped before the search is started. This is useful for applications that have a different behaviour at the beginning.</td>
</tr>
<tr>
<td><code>--duration=&lt;n&gt;</code></td>
<td>Search delay in seconds of phase</td>
</tr>
<tr>
<td><code>--dontcluster</code></td>
<td>Do not use online clustering for the detected bottlenecks.</td>
</tr>
<tr>
<td><code>--force-localhost</code></td>
<td>Locally start the agents instead of using SSH.</td>
</tr>
<tr>
<td><code>--help</code></td>
<td>Help information</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>--maxcluster=&lt;n&gt;</code></td>
<td>Maximum number of MPI processes analyzed by a single analysis agent.</td>
</tr>
<tr>
<td></td>
<td>It is not used on the Bluegene since the analysis agents are running on the</td>
</tr>
<tr>
<td></td>
<td>IO nodes. All processes on the compute nodes of an IO nodes connect to its</td>
</tr>
<tr>
<td></td>
<td>analysis agent.</td>
</tr>
<tr>
<td></td>
<td>Default: 64</td>
</tr>
<tr>
<td><code>--maxfan=&lt;n&gt;</code></td>
<td>Determines the fan-out of the tree of high-level agents in interactive mode.</td>
</tr>
<tr>
<td></td>
<td>Default: 4</td>
</tr>
<tr>
<td><code>--mpinumprocs=&lt;n&gt;</code></td>
<td>Number of MPI processes to be started.</td>
</tr>
<tr>
<td><code>--nprops=&lt;n&gt;</code></td>
<td>Specifies the number of properties the frontend prints to standard output.</td>
</tr>
<tr>
<td></td>
<td>Regardless of this value, all properties are output to the properties file.</td>
</tr>
<tr>
<td></td>
<td>Default: 50.</td>
</tr>
<tr>
<td><code>--ompnumthreads=&lt;n&gt;</code></td>
<td>Number of OMP threads to be started per MPI process.</td>
</tr>
<tr>
<td></td>
<td>Default: 1.</td>
</tr>
<tr>
<td><code>--pedantic</code></td>
<td>Shows all detected properties.</td>
</tr>
<tr>
<td><code>--phase=&lt;fileid:rfl&gt;</code></td>
<td>Specifies the phase region via the fileid and the region first line number.</td>
</tr>
<tr>
<td></td>
<td>If no phase region is specified, a user region is selected if at least one</td>
</tr>
<tr>
<td></td>
<td>is given in the code. If multiple are given, it is undefined which is</td>
</tr>
<tr>
<td></td>
<td>selected. If no user region is given, the main program is the user region</td>
</tr>
<tr>
<td></td>
<td>and the program will be restarted for each strategy step.</td>
</tr>
<tr>
<td></td>
<td>If you mark the phase region via a user region and would like to use user</td>
</tr>
<tr>
<td></td>
<td>regions also to guide analysis, you have to give the fileid and rfl for the</td>
</tr>
<tr>
<td></td>
<td>phase region.</td>
</tr>
</tbody>
</table>
### CHAPTER 5. CONFIGURATION OPTIONS

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--propfile=&lt;filename&gt;</td>
<td>Specify the file to use when exporting the properties. Default: properties.psc</td>
</tr>
<tr>
<td>--cfs-config=filename</td>
<td>Relative path to the CFS plugin configuration file</td>
</tr>
</tbody>
</table>
| --strategy=<strategyname> | Strategy used by analysis agent. Currently one of:  
  - MPI - MPI Communication analysis  
  - OMP - OpenMP analysis  
  - P6 - Power6 Analysis (only on Power6 machines)  
  - P6BF - Power6 Breadth First (only on Power6 machines)  
  - P6BF_MEMORY - Power6 Memory Behavior Analysis (only on Power6 machines)  
  - SCPS_BF - Generic memory analysis strategy scalabilityOMP - Automatic OpenMP scalability analysis |
| --timeout=<secs> | Timeout for startup of the agent hierarchy. Default: varying depending on the number of processes |
| --uninstrumented | **Autotuning only**: instructs Periscope to tune an uninstrumented application. Use with caution. See also Section 4.2. |
| --name           | Defines the name of the application                                           |
| --version        | Displays the version of Periscope.                                            |
| --starter=name   | Specifies resource manager name. E.g. Fast Interactive, Interactive, SLURM   |
| --statemachine-trace | Collects and prints state-machine transitions                               |
| --registry=<host:port> | This option defines the address of the registry service                      |
| --port=<n>       | Local port number                                                            |
| --maxthreads=<n> | Maximum number of threads assigned to a node agent                           |
| --masterhost=<hostname> | Name of the host where the frontend starts                                   |
| --hlagenthosts=<list> | List with host names of HL agents, e.g. –hlagenthost=h1,h2,...                |
### CHAPTER 5. CONFIGURATION OPTIONS

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--nodeagenthosts=&lt;list&gt;</td>
<td>List with host names of analysis agents, e.g. –nodeagenthosts=h1,h2,...</td>
</tr>
<tr>
<td>--agenthostfile=&lt;name&gt;</td>
<td>File containing host configuration</td>
</tr>
<tr>
<td>--manual</td>
<td>Run Periscope in manual mode</td>
</tr>
<tr>
<td>--property=&lt;name&gt;</td>
<td>Name of a property to export</td>
</tr>
</tbody>
</table>
Chapter 6

Advanced user information -
technical details

The application and the agent network are started through the `psc_frontend` process. First the set of available processors is analysed and based on this the mapping of application and analysis agent processes are determined. Both the application and the agent hierarchy are then started and a command is propagated from the frontend down to the analysis agents to start the search. The search is performed according to a search strategy selected when the frontend is started.

Each of the analysis agents, i.e. the nodes of the agent hierarchy, searches autonomously for inefficiencies in a subset of the application processes.

The application processes are linked with a monitoring system that provides the Monitoring Request Interface (MRI). The agents attach to the monitor via sockets. The MRI allows the agent to configure the measurements, to start, to halt, to resume the execution, and to retrieve the performance data. The monitor currently only supports summary information.

At the end of the local search, the detected performance properties are reported back via the agent hierarchy to the frontend.

6.1 Agent hierarchy

The layout of the agent hierarchy can be controlled by the user by means of the specific parameters of the `psc_frontend` executable:

`maxfan`: determines the fan-out of the tree of high-level agents. By default this is set to 4.
**maxcluster:** gives the maximum number of MPI processes analysed by a single analysis agent. The default number is 64.

Further information on how the agents work within a specific run of PTF can be gathered by using the `--selective-debug` parameter of the same `psc_frontend` executable:

```
--selective-debug= <level1>,<level2>...
```

with the following *levels* being relevant for the agent hierarchy:

- **AgentApplComm:** displays information regarding the communication between the agents and the application nodes.
- **AutotuneAgentStrategy:** displays information regarding the analysis strategy used in the analysis agent for tuning. To be used only when the tuning feature of PTF is being used.

Other values for the `--selective-debug` parameter can be found in the *PTF Developer’s Guide*.

Using a proper layout of the agent hierarchy is very important especially when performing analysis and tuning of applications on large systems.

Please note that, if the `--force-localhost` option of the `psc_frontend` executable is being used, then the entire agent hierarchy will be started on a single node. This is not recommended for applications using a large number of processes, as the communication between the agents and the application nodes would result in a bottleneck with a negative influence on the overall analysis time.
Chapter 7

Known Issues

- Automatic restart of the application does not work on the Bluegene. Make sure, you specify a user region that is executed repetitively.

- C instrumentation: The name of an OMP pragma should not occur again as a string in another context in this pragma, e.g., in a variable name.

- Measurements might be wrong in recursive algorithms.

- Multiple running instances of Periscope might not work on some systems.
Examples

You can find one example with the adapted makefile in ~/Periscope/testcases/add.

Example on SuperMUC

Periscope can be used in batch jobs.

Example batch script:

```bash
#@ wall_clock_limit = 00:30:00
#@ job_name = mytest
#@ job_type = MPICH
#@ class = test
#@ island_count = 1
#@ node = 1
#@ total_tasks = 1
#@ node_usage = not_shared
#@ initialdir = .
#@ output = out.txt
#@ error = out.txt
#@ notification = never
#@ queue
 . /etc/profile
 . /etc/profile.d/modules.sh

export OMP_NUM_THREADS=1
psc_frontend -apprun=../add.exe -mpinumprocs=1 -tune=compilerflags -phase="mainRegion" -force-localhost
```