# Score-P Cheat Sheet

## General Workflow Loop

- **Preparation**: instrument target application and set up measurement environment
- **Measurement**: run application with measurement infrastructure enabled
- **Analysis**: analyse generated performance data
- **Examination**: find possible cause of performance anomalies in the code
- **Optimization**: apply optimizations to eliminate bottleneck
- **Repeat**: apply analysis workflow loop until acceptable performance achieved

## Performance Analysis Procedure

- Create a profile with full instrumentation
- Compare runtime to uninstrumented run to determine overhead
- (Incrementally) create filter file using hints from the scorep-score tool
- Create an optimized profile with filter applied
- Investigate profile with CUBE
- For in-depth analysis, generate a trace **with filter applied** and examine it using
- Scalasca and than Vampir

## Application Instrumentation

- Prefix all compile/link commands with scorep
- Compile as usual
- Advanced instrumentation options available to further adjust the measurement configuration

## Application Measurement

Set Score-P environment variables

| | |
|---|---|
| SCOREP_EXPERIMENT_DIRECTORY | Name of the experiment directory |
| SCOREP_ENABLE_PROFILING | Enable generation of profiles (default=true) |
| SCOREP_ENABLE_TRACING | Enable the generation of traces (default=false) |
| SCOREP_TOTAL_MEMORY | Total memory in bytes used for Score-P per process (default=16M) |
| SCOREP_FILTERING_FILE | Name of file containing filter rules |

... and many more (see manual or run `scorep-info config-vars --full`)

Run application as usual:

```
% export SCOREP_ENABLE_TRACING=false
% export SCOREP_ENABLE_PROFILING=true
% export SCOREP_EXPERIMENT_DIRECTORY=scorep_run
% export OMP_NUM_THREADS=4
% mpirun -np 4 ./binary_scorep
```

## Profile Examination with CUBE and Filter File Creation

Analyze profile with CUBE

```
% cube scorep_run/profile.cubex
```

Create filter file with hints from scorep-score

```
% scorep-score -r scorep_run/profile.cubex
% scorep-score -r -f ./scorep.filt scorep_run/profile.cubex
```

Create profile with filter applied

```
% export SCOREP_EXPERIMENT_DIRECTORY=scorep_run_filter
% export SCOREP_FILTERING_FILE=scorep.filt
% mpirun -np 4 ./binary_scorep
```

## Automatic Trace Analysis with Scalasca

Run the application using Scalasca with trace collection and analysis

```
% export SCOREP_EXPERIMENT_DIRECTORY=scorep_run_trace
% export OMP_NUM_THREADS=4
% export SCOREP_TOTAL_MEMORY=25M
% scan -f ./scorep.filt -t mpirun -np 4 ./binary_scorep
```

Produces and examine trace analysis report

```
% square scorep_run_trace
```

## Interactive Performance Analysis with Vampir

### Open small traces direclty in Vampir

```
% vampir scorep_run_trace/traces.otf2
```

### Open large traces using VampirServer

1. Launch analysis server on remote machine

   ```
   % ssh remote-machine
   % vampirserver start -n 4
   Running 4 analysis processes... (abort with vampirserver stop 17950)
   VampirServer <17950> listens on: node123:30085
   ```

2. Open SSH tunnel to connect remote VampirServer with GUI on your local machine

   ```
   % ssh -L30000:node123:30085 mymachine
   ```

3. Open Vampir and connect to VampirServer (listening on localhost:30000 via SSH tunnel)

   ```
   % vampir localhost:30000:scorep_run_trace/traces.otf2
   ```

4. Shutdown VampirServer on remote machine when finished

   ```
   % ssh remote-machine
   % vampirserver stop
   ```

# MUST Cheat Sheet

**General Workflow Loop**

- **Preparation**: Compile with debug flag: -g
- **Execution**: run application with mustrun wrapper
- **Analysis**: analyze generated MUST_Output.html
- **Fix Bugs**: fix the code issues pointed out by the tool
- **Repeat**: apply analysis workflow loop until the tool reports no issue

**Execution with mustrun**

- Default mode with 4 processes:

```
% mustrun –np 4 ./a.out
```

- For different mpiexec command (e.g., srun):

```
% mustrun --must:mpiexec srun –n 4 ./a.out
```

- Query the necessary number of processes:

```
% mustrun --must:info –np 4 ./a.out
```

- Set a directory for the temp files (must_temp is default):

```
% mustrun --must:temp must_temp –n 4 ./a.out
```

- Distributed analysis for crashing applications:

```
% mustrun --must:nodesize 8 --must:cleanshm –n 4 ./a.out
```

- Nodesize must be devisor of processes scheduled per shared memory node

**MUST with DDT workflow:**

- Execute with MUST to capture the detected errors:

```
% mustrun --must:capture –np 4 ./a.out
```

- Then execute again with DDT attached:

```
% mustrun --must:ddt --must:reproduce –np 4 ./a.out
```

# Archer Cheat Sheet

**General Workflow Loop**

- **Preparation**: Compile with TSAN and debug flag
- **Execution**: run application with Archer library
- **Analysis**: understand the output on command line
- **Fix Bugs**: fix the code issues pointed out by the tool
- **Repeat**: apply analysis workflow loop until the tool reports no issue

**Compilation**

- C/C++ code:

```
% clang –fsanitize=thread –fopenmp –g app.c
```

- Fortran code:

```
% gfortran –fsanitize=thread –fopenmp –g –c app.f
% clang –fsanitize=thread –fopenmp -lgfortran app.o
```

**Execution**

- Make sure to have Archer library loaded by OpenMP runtime:

```
% export OMP_TOOL_LIBARIES=libarcher.so
% export TSAN_OPTIONS="ignore_noninstrumented_modules=0"
```

- Then execute as any OpenMP application:

```
% OMP_NUM_THREADS=4 ./a.out
```

**Archer Options**

```
% export ARCHER_OPTIONS="opt1=<0/1> opt2=<0/1>"
```

- verbose Print startup information. (default=0)
- enable Use Archer runtime library during execution. (default=1)

**Archer GUI (Needs special version of Archer)**

```
% archer-gui <folder containing Archer race reports>
```

% git clone https://git.rwth-aachen.de/protze/tools-tutorial.git