Performance Analysis at Scale: The Score-P Tools Infrastructure

Frank Winkler (frank.winkler@tu-dresden.de)
Performance tools will not automatically make your code run faster. They help you understand what your code does and where to put in work.
Performance engineering workflow

- Preparation
  - Prepare application with symbols
  - Insert extra code (probes/hooks)

- Measurement
  - Collection of performance data
  - Aggregation of performance data

- Optimization
  - Modifications intended to eliminate/reduce performance problem

- Analysis
  - Calculation of metrics
  - Identification of performance problems
  - Presentation of results
Performance Analysis Approaches: Profiling vs. Tracing

- **Data Acquisition**
  - Sampling
- **Data Recording**
  - Summarization
- **Data Presentation**
  - Statistics
  - Profiling
  - Tracing
  - Timelines
  - Logging
  - Event-based Instrumentation
So what is the right choice?

SO, YOU HAVE DECIDED TO UNDERSTAND WHAT A PROGRAM EXACTLY DOES?

CONGRATULATIONS!!!
YOU ARE AHEAD OF
99% OF YOUR COLLEAGUES

ARE YOU
SERIOUS
ABOUT
THIS?

WHAT KIND OF
PROFESSIONAL
ARE YOU?

SELECT
MAGNIFICATION

LOW
MED
HIGH

RUN IN
PROFILING
MODE

USE COMPILED
WRAPPER FILTERS

INSTRUMENT
THE CRITICAL
PARTS

FIND
INTERESTING
SPOTS

GO USE gprof
OR THE LIKE...

SLACKER

GET + INSTALL
THE TOOLS

SCIENTIST
ENGINEER
Score-P: Functionality

• Typical functionality for HPC performance tools
  – Instrumentation (various methods)
  – Sampling (experimental)

• Flexible measurement without re-compilation
  – Basic and advanced profile generation
  – Event trace recording

• Programming paradigms:
  – Multi-process
    • MPI, SHMEM
  – Thread-parallel
    • OpenMP, Pthreads
  – Accelerator-based
    • CUDA, OpenCL, OpenACC

Hybrid parallelism
Score-P: Architecture

Score-P measurement infrastructure

Hardware counter (PAPI, rusage, PERF, plugins)
Memory Recording (libc/C++ API)
IO Recording (Posix, NETCDF, HD5F, ADIOS)
3rd-Party Library Wrapping

Application (Process×Thread×Accelerator)

Process-level (MPI, SHMEM)
Thread-level (OpenMP, Pthreads)
Accelerator-based (CUDA, OpenACC)
Source code instrumentation (Compiler, PDT, User)
Sampling Interrupts (PAPI, PERF, timer)

Instrumentation wrapper
Score-P Workflow: Profiling

Instrumentation

Profile Run

Profile Analysis
Score-P Workflow: Instrumentation

CC = cc
CXX = CC
F90 = ftn

CC = scorep <options> cc
CXX = scorep <options> CC
F90 = scorep <options> ftn

• To see all available options for instrumentation:

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

Common options are:
...
--cuda Enables cuda instrumentation.
...
--user Enables user instrumentation.
...
--openacc Enables OpenACC instrumentation.
Score-P Workflow: Advanced Instrumentation

- For CMake and autotools based build systems it is recommended to use the scorep-wrapper script instances

**CMake**

```bash
#CMake
SCOREP_WRAPPER=OFF cmake .. \
-DCMAKE_C_COMPILER=scorep-icc \
-DCMAKE_CXX_COMPILER=scorep-icpc \
-DCMAKE_Fortran_COMPILER=scorep-ifc
```

**Autotools**

```bash
#Autotools
SCOREP_WRAPPER=OFF ../configure \
CC=scorep-icc \
CXX=scorep-icpc \
FC=scorep-ifc \
--disable-dependency-tracking
```

- Pass instrumentation and compiler flags at make

```bash
make SCOREP_WRAPPER_INSTRUMENTER_FLAGS="--user" \
SCOREP_WRAPPER_COMPILER_FLAGS="-g -O2"
```

```bash
scorep --user <your_compiler> –g –O2
```
Score-P Workflow: Measurement

• Measurements are configured via environment variables

```bash
$ scorep-info config-vars --full

SCOREP_ENABLE_PROFILING
  [...]
SCOREP_ENABLE_TRACING
  [...]
SCOREP_TOTAL_MEMORY
  Description: Total memory in bytes for the measurement system
  [...]
SCOREP_EXPERIMENT_DIRECTORY
  Description: Name of the experiment directory
  [...]
```

• Example for generating a profile:

```bash
$ export SCOREP_ENABLE_PROFILING=true
$ export SCOREP_ENABLE_TRACING=false
$ export SCOREP_EXPERIMENT_DIRECTORY=profile

$ aprun <instrumented binary>
```
Score-P: Cube

- Profile analysis tool for displaying performance data of parallel programs
- Originally developed as part of Scalasca toolset
- Available as a separate component of Score-P
- Representation of values (severity matrix) on three hierarchical axes
  - Performance property (metric)
  - Call-tree path (program location)
  - System location (process/thread)
- Three coupled tree browsers
Score-P: Cube Analysis Presentation

What kind of performance metric?

Where is it in the source code? In what context?

How is it distributed across the processes/threads?
Score-P Workflow: Profiling + Tracing

Instrumentation

Profile Run

Profile Analysis

Trace Run

Trace Analysis

Filtering

Reduce overhead and trace size
Score-P Workflow: Filtering

• Use scorep-score to define a filter
  – Exclude short frequently called functions from measurement
    • For profiling: reduce measurement overhead (if necessary)
    • For tracing: reduce measurement overhead and total trace size

```bash
$ scorep-score -r profile/profile.cubex
```

Estimated aggregate size of event trace: 40GB
Estimated requirements for largest trace buffer (max_buf): 10GB
Estimated memory requirements (SCOREP_TOTAL_MEMORY): 10GB

<table>
<thead>
<tr>
<th>Flt type</th>
<th>max_buf[B]</th>
<th>visits</th>
<th>time[s]</th>
<th>time[%]</th>
<th>time/visit[us]</th>
<th>region</th>
</tr>
</thead>
<tbody>
<tr>
<td>USR</td>
<td>3,421,305,420</td>
<td>522,844,416</td>
<td>144.46</td>
<td>13.4</td>
<td>0.28</td>
<td>matmul_sub</td>
</tr>
<tr>
<td>USR</td>
<td>3,421,305,420</td>
<td>522,844,416</td>
<td>102.40</td>
<td>9.5</td>
<td>0.20</td>
<td>matvec_sub</td>
</tr>
<tr>
<td>USR</td>
<td>3,421,305,420</td>
<td>522,844,416</td>
<td>200.94</td>
<td>18.6</td>
<td>0.38</td>
<td>binvrhs</td>
</tr>
<tr>
<td>USR</td>
<td>150,937,332</td>
<td>22,692,096</td>
<td>5.58</td>
<td>0.5</td>
<td>0.25</td>
<td>binvrhs</td>
</tr>
<tr>
<td>USR</td>
<td>150,937,332</td>
<td>22,692,096</td>
<td>13.21</td>
<td>1.2</td>
<td>0.58</td>
<td>lhsinit</td>
</tr>
</tbody>
</table>

About 10 GB just for these 3 regions per process!

• Filter file:

```bash
$ vim scorep.filt
```

```
SCOREP_REGION_NAMES_BEGIN EXCLUDE
matmul_sub
matvec_sub
binvrhs
```

Reduce measurement overhead and size of event trace to about 40 GB!
(Example uses 4 processes)
Score-P Workflow: Measurement

- Example for generating a trace

```bash
$ export SCOREP_ENABLE_PROFILING=false
$ export SCOREP_ENABLE_TRACING=true
$ export SCOREP_EXPERIMENT_DIRECTORY=trace
$ export SCOREP_TOTAL_MEMORY=2G
$ export SCOREP_FILTERING_FILE=scorep.filt

$ aprun <instrumented binary>
```
Vampir: Event Trace Visualization

- Show dynamic run-time behavior graphically at a fine level of detail
- Provide summaries (profiles) on performance metrics

**Timeline charts**
- Show application activities and communication along a time axis

**Summary charts**
- Provide quantitative results for the currently selected time interval
Detailed information about functions, communication and synchronization events for collection of processes.
Vampir: Performance Charts

- Trace visualization of FDS (Fire Dynamics Simulator)
Vampir at Scale: FDS with 8192 cores

- Fit to chart height feature in Master Timeline

Overview of the entire application run across all processes based on available pixels on screen.
Demo: Jacobi Solver

• Jacobi Example
  - Iterative solver for system of equations
    \[ U_{old} = U \]
    \[ u_{i,j} = b u_{old,i,j} + a_x (u_{old,i-1,j} + u_{old,i+1,j}) + a_y (u_{old,i,j-1} + u_{old,i,j+1}) - rHs / b \]
  - Code uses OpenMP, CUDA and MPI for parallelization

• Domain decomposition
  - Halo exchange at boundaries:
    • Via MPI between processes
    • Via CUDA between hosts and accelerators
The Score-P Tools Infrastructure

- Documentation at https://www.olcf.ornl.gov/support/software/

Thank you for your attention!
• Connect to Summit-dev and copy sources

```bash
$ cp /ccs/home/winklerf/scorep_tutorial/jacobi.tar.gz .
$ tar xzvf jacobi.tar.gz
$ cd jacobi
```

• Change programming environment and load modules

```bash
$ module load gcc/5.4.0
$ module load cuda
$ module load scorep
```

• Compile benchmark and submit job

```bash
$ make
$ cd bin
$ bsub < run.lsf
$ less run.lsf
Jacobi relaxation Calculation: 8192 x 8192 mesh with
2 processes and 6 threads + one Tesla P100-SXM2-16GB for each process.
614 of 4097 local rows are calculated on the CPU to balance the load
between the CPU and the GPU.
0, 0.489197
100, 0.002397
[...]
total: 9.409952 s
```
Demo: Jacobi Solver / Profiling

- **Build instrumented executable**
  
  ```
  $ make clean
  $ make scorep
  scorep --cuda cc ... -o bin/jacobi_mpi+openmp+cuda
  ```

- **Submit job for profiling run**
  
  ```
  $ less run_profile.lsf
  [...] export SCOREP_ENABLE_PROFILING=true
  export SCOREP_ENABLE_TRACING=false
  export SCOREP_EXPERIMENT_DIRECTORY=jacobi_profile
  export SCOREP_CUDA_ENABLE=yes
  [...] mpirun -n 2 ./jacobi_mpi+openmp+cuda 8192 8192 0.15

  $ bsub < run_profile.lsf
  $ less jacobi.o[JOB_ID]
  Jacobi relaxation Calculation: 8192 x 8192 mesh with
  2 processes and 6 threads + one Tesla P100-SXM2-16GB for each process.
  [...] total: 10.678350 s
  ```
Demo: Jacobi Solver / Profile Analysis

- Perform flat profile analysis with `cube_stat`

```bash
$ cd bin
$ cube_stat -t 10 -p jacobi_mpi+openmp+cuda_profile/profile.cubex
```
```
cube::Region     NumberOfCalls   ExclusiveTime   InclusiveTime
!$omp for @jacobi_cuda.c:188  32000.000000  131.797289  131.797289
!$omp implicit barrier  32000.000000  104.298683  104.298683
!$omp for @jacobi_cuda.c:258  32000.000000   42.999056  50.568642
[...]
```

- Perform call-path profile analysis with Cube

```bash
$ cube jacobi_profile/profile.cubex
```
Do we need a filter? (Overhead and memory footprint)

```bash
$ scorep-score jacobi_profile/profile.cubex
Estimated aggregate size of event trace: 10MB
Estimated requirements for largest trace buffer (max_buf): 5MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY): 41MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=41MB to avoid intermediate flushes or reduce requirements using USR regions filters.)
```

<table>
<thead>
<tr>
<th>flt</th>
<th>type</th>
<th>max_buf[B]</th>
<th>visits</th>
<th>time[s]</th>
<th>time[%]</th>
<th>time/visit[us]</th>
<th>region</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>4,924,060</td>
<td>310,504</td>
<td>308.53</td>
<td>100.0</td>
<td>993.63</td>
<td>ALL</td>
<td></td>
</tr>
<tr>
<td>OMP</td>
<td>4,135,850</td>
<td>256,417</td>
<td>287.31</td>
<td>93.1</td>
<td>1120.46</td>
<td>OMP</td>
<td></td>
</tr>
<tr>
<td>CUDA</td>
<td>494,338</td>
<td>38,025</td>
<td>10.40</td>
<td>3.4</td>
<td>273.53</td>
<td>CUDA</td>
<td></td>
</tr>
<tr>
<td>COM</td>
<td>156,260</td>
<td>12,020</td>
<td>10.46</td>
<td>3.4</td>
<td>870.58</td>
<td>COM</td>
<td></td>
</tr>
<tr>
<td>MPI</td>
<td>137,222</td>
<td>4,012</td>
<td>0.30</td>
<td>0.1</td>
<td>73.96</td>
<td>MPI</td>
<td></td>
</tr>
<tr>
<td>MEMORY</td>
<td>260</td>
<td>20</td>
<td>0.06</td>
<td>0.0</td>
<td>2972.15</td>
<td>MEMORY</td>
<td></td>
</tr>
<tr>
<td>USR</td>
<td>130</td>
<td>10</td>
<td>0.00</td>
<td>0.0</td>
<td>10.26</td>
<td>USR</td>
<td></td>
</tr>
</tbody>
</table>

No filtering required.
Demo: Jacobi Solver / Tracing

- Submit job for tracing run

```bash
$ cd ..
$ less run_trace.lsf
[...]
export SCOREP_ENABLE_PROFILING=false
export SCOREP_ENABLE_TRACING=true
export SCOREP_EXPERIMENT_DIRECTORY=jacobi_trace
export SCOREP_CUDA_ENABLE=yes
export SCOREP_TOTAL_MEMORY=50MB
[...]
mpirun -n 2 ./jacobi_mpi+openmp+cuda 8192 8192 0.15

$ bsub < run_trace.lsf
$ less jacobi.o[JOB_ID]

Jacobi relaxation Calculation: 8192 x 8192 mesh with
2 processes and 6 threads + one Tesla P100-SXM2-16GB for each process.
614 of 4097 local rows are calculated on the CPU to balance the load
between the CPU and the GPU.
  0, 0.489197
  100, 0.002397
  [...]
  900, 0.000269
total: 9.895828 s
Demo: Jacobi Solver / Trace Analysis

- Perform analysis on the trace data with Vampir

```bash
$ cd bin
$ module load vampir
$ vampir jacobi_trace/traces.otf2
```