Score-P & Vampir

Comprehensive Multi-Paradigm Performance Analysis

Ronny Brendel
ORNL / TU Dresden
brendelr@ornl.gov

Argonne Training Program on Extreme-Scale Computing (ATPESC), St. Charles, Illinois, 8th August 2017
Introduction > Methods

**Profile**
- Information accumulated into buckets
- Typically small overhead
- Static representation

<table>
<thead>
<tr>
<th>Time (%)</th>
<th>Function name</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.44</td>
<td>QListData::isEmpty</td>
</tr>
<tr>
<td>2.96</td>
<td>QHash::findNode</td>
</tr>
<tr>
<td>2.67</td>
<td>QList::last</td>
</tr>
<tr>
<td>1.71</td>
<td>handleEnter</td>
</tr>
<tr>
<td>0.58</td>
<td>QHash::find</td>
</tr>
</tbody>
</table>

**Trace**
- Event log
- Possibly large overhead
- Interactive presentation

<table>
<thead>
<tr>
<th>Time (s)</th>
<th>Function name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.21</td>
<td>QListData::isEmpty</td>
</tr>
<tr>
<td>0.66</td>
<td>QHash::findNode</td>
</tr>
<tr>
<td>0.60</td>
<td>QList::last</td>
</tr>
<tr>
<td>0.38</td>
<td>handleEnter</td>
</tr>
<tr>
<td>0.13</td>
<td>QHash::find</td>
</tr>
</tbody>
</table>
Introduction > Methods

**Sampling**
- Interrupt with given interval (typically ~10ms)
- Statistical guarantees

**Instrumentation**
- Callback before/after event
- Exact time and call counts
- Wrappers have access to function arguments
Introduction > Methods

Presentation

Statistics

Timelines

Recording

Summarization

Logging

Interception

Sampling

Instrumentation
Introduction > Methods

• Myth 1: Tracing has a giant overhead
  ▪ **It depends on the event rate**
    - E.g. an MPI-only trace has very low overhead
  ▪ Admittedly:
    - Main problem 1: Compiler instrumentation
      - → Compiler plugins to the rescue
    - Wrappers are mostly fine and widely used
    - Main problem 2: Filtering workflows are inconvenient
      - Tool-specific problem, not a general „Tracing“-problem
Introduction > Methods

• Myth 2: Tracing produces giant recordings
  • Analogous to Myth 1: It depends on the number of events
  • Score-P has a simple filtering workflow that copes with that
  • The majority of our archived tracefiles are below 1 GB

• Personally, I always configure Score-P so that the trace is easy to handle on my laptop
Introduction > Methods

• Myth 3: One technique is superior
  • Not true

• Sampling:
  • Does not give an absolutely accurate picture of a run

```c
int main(int argc, char** args) {
    QApplication app(argc, argv);
    QWidget w;
    QImage i;
    w.show();
    return app.exec();
}
```
Introduction > Methods

• Myth 3: One technique is superior
  • Not true

• Sampling:
  • Does not give an absolutely accurate picture of a run
  • Cannot count function calls
  • Cannot record exact timings
  • Cannot record exact performance counters
  • It is statistical sampling

• It cannot capture semantics of APIs, i.e. it cannot follow API usage and analyze passed arguments, e.g. transferred bytes
Introduction > Methods

• Myth 3: One technique is superior
  • Not true

• Instrumentation/Tracing:
  • Typically more difficult than using a profiler
  • Does not guarantee anything about overhead or recording size
    ▪ (But it’s not inconceivable to achieve this)
Introduction > Methods

- In practice, most tools use a combination
  - Coarse-grained sampling + call stack unwinding
  - Wrappers for library functions of interest (MPI_Send, cudaMalloc, dlopen)
Introduction
Introduction > Vampir

- Comprehensive, powerful performance data visualization
- Developed since 1996
- Commercial

http://vampir.eu
Introduction > Score-P

- Jointly developed next-generation performance data collector
- Developed since 2009
- Open-source (3-clause BSD)
- Partners:
  - TU Dresden, GER
  - FZ Jülich, GER
  - TU München, GER
  - University of Oregon, USA
  - RWTH Aachen; TU Darmstadt;
  - Gesellschaft für numerische Simulation mbH;
  - German Research School for Simulation Sciences GmbH (all GER)

http://score-p.org
Introduction > Score-P

- **Supports:**
  - C, C++, Fortran
  - MPI, SHMEM
  - OpenMP, PThreads
  - CUDA, OpenACC, OpenCL

- **Compilers:**
  - Cray, GNU, IBM, Intel, Pathscale, PGI, LLVM
Tutorial
Tutorial > Usage

- **Load Score-P**
  
  ```
  $ module load scorep
  $ echo "+vampir" >> ~/.soft & & resoft
  ```

- **Compile & Link**
  
  (with MPI)
  
  ```
  $ scorep ... gcc ... main.c
  $ scorep mpicc main.c
  ```

  (with SHMEM)
  
  ```
  $ scorep oshcc main.c
  ```

- **CMake**

  ```
  $ SCOREP_WRAPPER=OFF cmake -DCMAKE_C_COMPILER=scorep-gcc ..
  $ SCOREP_WRAPPER_INSTRUMENTER_FLAGS="..." SCOREP_WRAPPER_COMPILER_FLAGS="..." make
  ```

- **Autotools**

  ```
  $ SCOREP_WRAPPER=OFF ./configure CC=scorep-gcc MPICC=scorep-mpicc ..
  $ SCOREP_WRAPPER_INSTRUMENTER_FLAGS="..." SCOREP_WRAPPER_COMPILER_FLAGS="..." make
  ```
Tutorial > Usage

• **Execute**

$ ./a.out

$ mpirun -np 2 ./a.out

$ shmemrun -np 2 ./a.out

• **Inspect**

$ ls -R

scorep-20170323_1309_7243761919249966 a.out

./scorep-20170323_1309_7243761919249966:
profile.cubex scorep.cfg

• **Inspect > Cube**

$ cube scorep-20170323_1309_7243761919249966/profile.cubex
Tutorial > Usage

- Inspect > Cube
Tutorial > Usage

- Runtime Options
  - Profiling (default)
  - Tracing
  - Performance counters
  - Filtering

- Memory (default: 16M)

- And many more...

```
$ export SCOREP_ENABLE_PROFILING=true
$ export SCOREP_ENABLE_TRACING=true
$ export SCOREP_METRIC_PAPI=PAPI_L2_TCM,...
$ export SCOREP_FILTERING_FILE=my.filt
$ export SCOREP_TOTAL_MEMORY=1G
$ scorep-info config-vars
```
Tutorial > Usage

• Inspect > Vampir

```bash
$ export SCOREP_ENABLE_PROFILING=false
$ export SCOREP_ENABLE_TRACING=true
$ export SCOREP_METRIC_PAPI=PAPI_TOT_INS,PAPI_TOT_CYC

$ mpirun -np 4 ./a.out

$ ls -R
scorep-20170323_1309_7243761919249966  a.out
./scorep-20170323_1309_7243761919249966:
scorep.cfg  traces/  traces.def  traces.otf2

$ module load vampir

$ vampir scorep-20170323_1309_7243761919249966/traces.otf2
```
Tutorial > Usage

- Inspect > Vampir
Tutorial > Overhead

- Trace size and overhead varies greatly with event rate
  - Make a reference run and check wall clock time!
  - Rule of thumb: Try to stay below 10% overhead

→ Filtering is an integral part of Score-P’s workflow
Tutorial > Profiling Workflow

1) Instrument & build
2) Execute
3) Analyze profile using Cube

Mind that overhead can be too high. Runtime filtering does not help, if the event rate is extremely high. → Compile-time filtering with our GCC instrumentation compiler plugin solves this:

```
$ scorep --instrument-filter=<filter_file> gcc main.c
```
Tutorial > Tracing Workflow

1) Instrument & build
2) Execute (profiling)
3) Analyze overhead
   - If the estimated trace size is too large, filter and goto (3)
4) Execute using the filter (tracing)
5) Analyze trace using Vampir
Tutorial > Tracing Workflow

3) Analyze Overhead

$ scorep-score scorep-20170323_1309_7243761919249966/profile.cubex

Estimated aggregate size of event trace: 40GB
Estimated requirements for largest trace buffer (max_buf): 6GB
Estimated memory requirements (SCOREP_TOTAL_MEMORY): 6GB
(warning: The memory requirements cannot be satisfied by Score-P to avoid intermediate flushes when tracing. Set SCOREP_TOTAL_MEMORY=4G to get the maximum supported memory 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>5,383,272,006</td>
<td>1,635,443,611</td>
<td>579.23</td>
<td>100.0</td>
<td>0.35</td>
<td>ALL</td>
<td></td>
</tr>
<tr>
<td>USR</td>
<td>5,358,738,138</td>
<td>1,631,138,913</td>
<td>253.00</td>
<td>43.7</td>
<td>0.16</td>
<td>USR</td>
<td></td>
</tr>
<tr>
<td>OMP</td>
<td>23,580,522</td>
<td>4,089,856</td>
<td>318.79</td>
<td>55.0</td>
<td>77.95</td>
<td>OMP</td>
<td></td>
</tr>
<tr>
<td>COM</td>
<td>665,210</td>
<td>182,120</td>
<td>0.90</td>
<td>0.2</td>
<td>4.95</td>
<td>COM</td>
<td></td>
</tr>
<tr>
<td>MPI</td>
<td>288,136</td>
<td>32,722</td>
<td>6.55</td>
<td>1.1</td>
<td>200.11</td>
<td>MPI</td>
<td></td>
</tr>
</tbody>
</table>
### Tutorial > Tracing Workflow

#### 3) Analyze Overhead

```bash
$ scorep-score -r scorep-20170323_1309_7243761919249966/profile.cubex

...]
```

<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>5,383,272,006</td>
<td>1,635,443,611</td>
<td>579.23</td>
<td>100.0</td>
<td>0.35</td>
<td>ALL</td>
<td></td>
</tr>
<tr>
<td>USR</td>
<td>5,358,738,138</td>
<td>1,631,138,913</td>
<td>253.00</td>
<td>43.7</td>
<td>0.16</td>
<td>USR</td>
<td></td>
</tr>
<tr>
<td>OMP</td>
<td>23,580,522</td>
<td>4,089,856</td>
<td>318.79</td>
<td>55.0</td>
<td>77.95</td>
<td>OMP</td>
<td></td>
</tr>
<tr>
<td>COM</td>
<td>665,210</td>
<td>182,120</td>
<td>0.90</td>
<td>0.2</td>
<td>4.95</td>
<td>COM</td>
<td></td>
</tr>
<tr>
<td>MPI</td>
<td>288,136</td>
<td>32,722</td>
<td>6.55</td>
<td>1.1</td>
<td>200.11</td>
<td>MPI</td>
<td></td>
</tr>
</tbody>
</table>

| USR | 1,716,505,830 | 522,844,416 | 79.32 | 13.7   | 0.15    | matmul_sub_    |
| USR | 1,716,505,830 | 522,844,416 | 53.44 | 9.2    | 0.10    | matvec_sub_    |
| USR | 1,716,505,830 | 522,844,416 | 111.47| 19.2   | 0.21    | binvcrhs_      |
| USR | 76,195,080    | 22,692,096  | 2.76  | 0.5    | 0.12    | binvrhs_       |
| USR | 76,195,080    | 22,692,096  | 4.37  | 0.8    | 0.19    | lhsinit_       |
| USR | 56,825,184    | 17,219,840  | 1.63  | 0.3    | 0.09    | exact_solution_ |
3) Filter

```bash
$ cat myfilter.filt
SCOREP_REGION_NAMES_BEGIN
  EXCLUDE
    matmul_sub*
    matvec_sub*
    binvcrhs*
    Binvrhs*
    exact_solution*
    lhs*init*
    timer_*
SCOREP_REGION_NAMES_END

$ scorep-score -f myfilter.filt scorep-20170323*/profile.cubex

Estimated aggregate size of event trace: 409MB
Estimated requirements for largest trace buffer (max_buf): 58MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY): 70MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=70M to avoid [...])
4) Execute using the filter (Tracing)

$ export SCOREP_ENABLE_TRACING=true
$ export SCOREP_TOTAL_MEMORY=70M
$ export SCOREP_FILTERING_FILE=myfilter.filt

$ mpirun -np 8 ./a.out

4) GCC-only: Compile-time filtering

$ scorep --instrument-filter=myfilter.filt gcc main.c

$ export SCOREP_ENABLE_TRACING=true
$ export SCOREP_TOTAL_MEMORY=70M

$ mpirun -np 8 ./a.out # no runtime filtering needed
Tutorial > Vampir Demo (Live)
Tutorial > Getting Help

- `$ scorep --help`
- `$ scorep-wrapper --help`
- `$ scorep-info config-vars`

- Manuals: 
  - `$SCOREP_DIR/share/doc/scorep/pdf/scorep.pdf`
- https://www.alcf.anl.gov/vampir
- support@score-p.org, service@vampir.eu

- VI-HPS offers trainings (Invite them!)
  - http://www.vi-hps.org/training/tws/
  - http://www.vi-hps.org/training/material/
Conclusions

- Holistic, powerful and detailed software performance analysis
  - Everything in one picture
  - Extremely customizable
  - Extremely scalable

- Advanced features
- Very active in adopting new features

- Active research community

- Continuously selected by OLCF
- Enabler for science at extreme scale
Future Work

- User library wrapping
- I/O analysis (POSIX, ADIOS, HDF5, NetCDF)
- Non-volatile memory analysis
- POWER8/9 & Clang support
- Instrumentation compiler plugin for LLVM
- OpenMP tools interface (OMPT)
- MPI RMA analysis
- KNL-specific metrics and topology information
Sponsors & Projects
Contributors

- **Score-P**
  

- **Vampir**
  
  Andreas Knüpfer, Bert Wesarg, Frank Winkler, Hartmut Mix, Heide Rohling, Holger Brunst, Jens Doleschal, Matthias Weber, Laszlo Barabas, Michael Heyde, Michael Peter, Reinhard Neumann, Ronald Geisler, Ronny Brendel, Thomas William
Hands-On

- Prepared example:
  
  ```plaintext
  ssh -Y titan
  cp -r /lustre/atlas/world-shared/stf010/brendel/heat ~
  cd ~/heat
  less instructions.txt
  ```

- https://www.alcf.anl.gov/vampir
  - Please install Vampir on your laptop

- (https://www.olcf.ornl.gov/kb_articles/software-scorep/)