Performance Analysis at Scale: The Score-P Tools Infrastructure

23 May 2016

Frank Winkler
On-site contractor for Vampir, CSMD
Disclaimer

It is extremely easy to waste performance!

- Bad MPI (50-90%)
- No node-level parallelism (94%)
- No vectorization (75%)
- Bad memory access pattern (99%)
- In sum: 0.008% of the peak performance (about 2 teraflops of Titan)
Disclaimer (2)

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
Agenda

Performance Analysis Approaches
- Sampling vs. Instrumentation
- Profiling vs. Tracing

Score-P: Scalable Performance Measurement Infrastructure for Parallel Codes
- Motivation
- Functionality
- Architecture
- Workflow
- Advanced Features

Performance Analysis Tools
- Cube
- Vampir

Demo
- Performance Analysis of Jacobi Solver on Titan

Conclusions
Sampling

- Running program is periodically interrupted to take measurement
- Statistical inference of program behavior
  - Not very detailed information on highly volatile metrics
  - Requires long-running applications
- Works with unmodified executables
Instrumentation

- Measurement code is inserted such that every event of interest is captured directly
  - Can be done in various ways

- Advantage:
  - Much more detailed information

- Disadvantage:
  - Processing of source-code / executable necessary
  - Large relative overheads for small functions
Profilng vs. Tracing

- Statistics

Number of Invocations

- main
- bar
- foo

Execution Time

- foo
- bar
- main

- Timelines

Time
Terms Used and How They Connect

Analysis Layer

Data Acquisition

Data Recording

Data Presentation

Analysis Technique

Profiling

- Statistics
- Summarization

Tracing

- Timelines
- Logging
- Event-based Instrumentation

Sampling
So what is the right choice?
Agenda

- Performance Analysis Approaches
  - Sampling vs. Instrumentation
  - Profiling vs. Tracing

- Score-P: Scalable Performance Measurement Infrastructure for Parallel Codes
  - Motivation
  - Functionality
  - Architecture
  - Workflow
  - Advanced Features

- Performance Analysis Tools
  - Cube
  - Vampir

- Demo
  - Performance Analysis of Jacobi Solver on Titan

- Conclusions
Score-P: Motivation

- Several performance tools co-exist
- Separate measurement systems and output formats
- Complementary features and overlapping functionality
- Redundant effort for development and maintenance
- Limited or expensive interoperability
- Complications for user experience, support, training

<table>
<thead>
<tr>
<th>Vampir</th>
<th>Scalasca</th>
<th>TAU</th>
<th>Periscope</th>
</tr>
</thead>
<tbody>
<tr>
<td>VampirTrace</td>
<td>EPILOG /</td>
<td>TAU native</td>
<td></td>
</tr>
<tr>
<td>OTF</td>
<td>CUBE</td>
<td>formats</td>
<td>measurement</td>
</tr>
</tbody>
</table>

Performance Analysis at Scale: The Score-P Tools Infrastructure – Frank Winkler
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 (Prototype)
Score-P: Architecture

**Application (Process×Thread×Accelerator)**

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

**Score-P measurement infrastructure**

- Hardware counter (PAPI, rusage, PERF, plugins)
- Memory Recording (libc/C++ API)
- IO Recording (Posix, NETCDF, HD5F)

**Score-P: Architecture**

- Vampir: Event traces (OTF2)
- Scalasca: Call-path profiles (CUBE4, TAU)
- Cube: Call-path profiles (CUBE4, TAU)
- TAU: Call-path profiles (CUBE4, TAU)
- Periscope: Online interface

**Instrumentation wrapper**

- Process-level (MPI, SHMEM)
- Thread-level (OpenMP, Pthreads)
- Accelerator-based (CUDA, OpenCL)
- Source code instrumentation (Compiler, PDT, User)
Score-P: Workflow

0. Perform a reference run and note the run time to be able to refer to it later

1. Instrument your application with Score-P

   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:
   ...
   --nocompiler  Enables compiler instrumentation.
   --user        Enables user instrumentation.
   --cuda        Enables cuda instrumentation.
Score-P: Workflow

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

```cmake
SCOREP_WRAPPER=OFF cmake .. \\ -DCMAKE_C_COMPILER=scorep-cc \\
-DCMAKE_CXX_COMPILER=scorep-CC \\
-DCMAKE_Fortran_COMPILER=scorep-ftn
```

```autotools
SCOREP_WRAPPER=OFF ../configure \\
 CC=scorep-cc \\
 CXX=scorep-CC \\
 FC=scorep-ftn \\
 --disable-dependency-tracking
```

• Pass instrumentation and compiler flags at make:

```make
make SCOREP_WRAPPER_INSTRUMENTER_FLAGS="--cuda" \ SCOREP_WRAPPER_COMPILER_FLAGS="-g -O2"
```

```shell
scorep --cuda <your_compiler> -g -O2
```
Score-P: Workflow

2. Perform a measurement run with profiling enabled

• Example for generating a profile:

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

$ aprun <instrumented binary>
```

• To see all environment variables for the measurement:

```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
  [...]
```
Score-P: Workflow

3. Compare profile runtime with reference runtime

- If overhead is too high:
  - Exclude short frequently called functions from measurement using hints from scorep-score

```bash
$ scorep-score -r profile/profile.cubex
[...]
Ft type        max_buf[B]   visits time[s]  time[%]  time/visit[us]  region
[...]
USR  3,421,305,420  522,844,416  137.49  10.7  0.26  matvec_sub
USR  3,421,305,420  522,844,416  174.16  13.5  0.33  matmul_sub
USR  3,421,305,420  522,844,416  226.67  17.6  0.43  binvcrhs
USR  150,937,332   22,692,096   6.73   0.5  0.30  binvrhs
USR  150,937,332   22,692,096  14.69   1.1  0.65  lhsinit
USR  112,194,160   17,219,840   4.70   0.4  0.27  exact_solution
OMP  1,312,128    102,912    0.06   0.0  0.58  !$omp_parallel
```

42% of the total time for these 3 regions, however, much of that is very likely measurement overhead due to short frequently called functions!
Score-P: Workflow

4. Create an optimized profile with filter applied if measurement overhead of full instrumented profile is too high

- Create a filter file and list functions to be excluded

```bash
$ vim scorep.filt
SCOREP_REGION_NAMES_BEGIN EXCLUDE
   matmul_sub
   matvec_sub
   binvcrhs
SCOREP_REGION_NAMES_END
```

- Example for generating a profile with filter applied:

```bash
$ export SCOREP_ENABLE_PROFILING=true
$ export SCOREP_ENABLE_TRACING=false
$ export SCOREP_FILTERING_FILE=scorep.filt
$ export SCOREP_EXPERIMENT_DIRECTORY=profile_with_filter
$ aprun <instrumented binary>
```
Score-P: Workflow

5. Perform analysis on (optimized) profile data

- Flat profile analysis with `cube_stat`:

```
cube_stat -t 3 -p profile_with_filter/profile.cubex
```

<table>
<thead>
<tr>
<th>Region</th>
<th>NumberOfCalls</th>
<th>ExclusiveTime</th>
<th>InclusiveTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>!$omp do @z_solve.f:52</td>
<td>51456.000000</td>
<td>131.579771</td>
<td>131.579771</td>
</tr>
<tr>
<td>!$omp do @y_solve.f:52</td>
<td>51456.000000</td>
<td>122.818761</td>
<td>122.818761</td>
</tr>
<tr>
<td>!$omp do @x_solve.f:54</td>
<td>51456.000000</td>
<td>117.027571</td>
<td>117.027571</td>
</tr>
</tbody>
</table>

- Call-path profile analysis with Cube:

```
cube profile_with_filter/profile.cubex
```
Score-P: Workflow

6. Define an appropriate filter for a tracing run
   • Exclude functions from measurement which require a large trace buffer to reduce total trace size
   • Use scorep-score with **full** instrumented profile

\[
\text{$ \textit{scorep-score} -r \textit{profile/profile.cubex}$}
\]

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

<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>137.49</td>
<td>10.7</td>
<td>0.26</td>
<td>matvec_sub</td>
</tr>
<tr>
<td>USR</td>
<td>3,421,305,420</td>
<td>522,844,416</td>
<td>174.16</td>
<td>13.5</td>
<td>0.33</td>
<td>matmul_sub</td>
</tr>
<tr>
<td>USR</td>
<td>3,421,305,420</td>
<td>522,844,416</td>
<td>226.67</td>
<td>17.6</td>
<td>0.43</td>
<td>binvcrhs</td>
</tr>
<tr>
<td>USR</td>
<td>150,937,332</td>
<td>22,692,096</td>
<td>6.73</td>
<td>0.5</td>
<td>0.30</td>
<td>binvrhs</td>
</tr>
<tr>
<td>USR</td>
<td>150,937,332</td>
<td>22,692,096</td>
<td>14.69</td>
<td>1.1</td>
<td>0.65</td>
<td>lhsinit</td>
</tr>
<tr>
<td>USR</td>
<td>112,194,160</td>
<td>17,219,840</td>
<td>4.70</td>
<td>0.4</td>
<td>0.27</td>
<td>exact_solution</td>
</tr>
<tr>
<td>OMP</td>
<td>1,312,128</td>
<td>102,912</td>
<td>0.06</td>
<td>0.0</td>
<td>0.58</td>
<td>!$omp parallel</td>
</tr>
</tbody>
</table>

About 10 GB just for these 6 regions per process!

• Test the effect of your filter on the trace file

\[
\text{$ \textit{scorep-score} -f \textit{scorep.filt} \textit{profile/profile.cubex}$}
\]
Score-P: Workflow

5. Perform a measurement run with tracing enabled and the filter applied

```bash
export SCOREP_ENABLE_PROFILING=false
export SCOREP_ENABLE_TRACING=true
export SCOREP_EXPERIMENT_DIRECTORY=trace
export SCOREP_FILTERING_FILE=scorep.filt
aprun <instrumented binary>
```

6. Perform analysis on the trace data with Vampir

```bash
vampir trace/traces.otf2
```
Score-P: Workflow Summary

Instrumentation

Profile Run

Profile Analysis

Trace Run

Trace Analysis

Filtering

Reduce run-time filter overhead

Reduce overhead if necessary

Reduce overhead and trace size

Compile-time filter

Run-time filter
Score-P Advanced Features: Sampling

• Alternative to compiler instrumentation to generate profiles or traces
• Regulate the trade-off between overhead and correctness
• Libunwind/1.1 to capture current stack
• Sampling interrupt sources:
  – Interval timer, PAPI, Perf
• Example for enabling sampling for measurement run:

```bash
$ export SCOREP_ENABLE_UNWINDING=true
$ export SCOREP_SAMPLING_EVENTS=PAPI_TOT_CYC@1000000
```

• Combination of instrumented and sampled events (not for compiler instrumented events)
• Calling context information for every event
Score-P Advanced Features: Memory Rec.

- Memory (de)allocations are recorded via the libc/C++ API
- Recording of memory location’s call-site in sampling mode
  - Debugging symbols required (-g)
- Interplay of memory usage and application’s execution
  - CUBE: (De)allocation size, maximum heap memory, leaked bytes
  - Vampir: Memory usage in “Counter Timelines”
- Enabling memory recording for measurement run:

```bash
$ export SCOREP_MEMORY_RECORDING=true
```
Agenda

Performance Analysis Approaches
- Sampling vs. Instrumentation
- Profiling vs. Tracing

Score-P: Scalable Performance Measurement Infrastructure for Parallel Codes
- Motivation
- Functionality
- Architecture
- Workflow
- Advanced Features

Performance Analysis Tools
- Cube
- Vampir

Demo
- Performance Analysis of Jacobi Solver on Titan

Conclusions
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
What kind of performance metric?

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

How is it distributed across the processes/threads?
Vampir

- Event trace analysis tool for displaying performance data of complex parallel programs
- 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
Vampir: Performance Charts Overview

Timeline Charts
- Master Timeline ➔ all threads’ activities over time per thread
- Summary Timeline ➔ all threads activities over time per activity
- Performance Radar ➔ all threads’ perf-metric over time
- Process Timeline ➔ single thread’s activities over time
- Counter Data Timeline ➔ single threads perf-metric over time

Summary Charts
- Function Summary
- Process Summary
- Message Summary
- Communication Matrix View
Vampir: Performance Charts

- Trace visualization of FDS (Fire Dynamics Simulator)
Vampir: Performance Charts

Master Timeline

Detailed information about functions, communication and synchronization events for collection of processes.
Vampir: Performance Charts

Summary Timeline

Fractions of the number of processes that are actively involved in given activities at a certain point in time.
Vampir: Performance Charts

Process Timeline

Detailed information about different levels of function calls in a stacked bar chart for an individual process.
Vampir: Performance Charts

Counter Timeline

Detailed counter information over time for an individual process.
Vampir: Performance Charts

Performance Radar

Detailed counter information over time for a collection of processes.
**Vampir: Where Do the Metrics Come From?**

- Custom Metrics Built-In Editor
Vampir: Performance Charts

Function Summary

Overview of the accumulated information across all functions and for a collection of processes.
Vampir: Performance Charts

Process Summary

Overview of the accumulated information across all functions and for every process independently.

Clustering: Grouping of similar processes by using summarized function information.
Vampir: Performance Charts

Communication Matrix View
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.
Vampir at Scale: LSMS (hybrid parallelism)

- 5831 processes: 343xMPI with 8xOpenMP and 8xCUDA
Vampir at Scale: LSMS (hybrid parallelism)

- Group threads and CUDA streams
Vampir at Scale: LSMS (hybrid parallelism)

• Collapse all MPI processes
Vampir at Scale: LSMS (hybrid parallelism)

- Fit to chart height for all collapsed MPI processes
Agenda

Performance Analysis Approaches
- Sampling vs. Instrumentation
- Profiling vs. Tracing

Score-P: Scalable Performance Measurement Infrastructure for Parallel Codes
- Motivation
- Functionality
- Architecture
- Workflow
- Advanced Features

Performance Analysis Tools
- Cube
- Vampir

Demo
- Performance Analysis of Jacobi Solver on Titan

Conclusions
Demo: Jacobi Solver

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

- Domain decomposition
  - Halo exchange at boundaries:
    - Via MPI between processes
    - Via CUDA between hosts and accelerators
Demo: Jacobi Solver / Setup

- Connect to Titan via X forwarding and copy sources

```bash
$ cd $MEMBERWORK/[projid]
$ cp /sw/sources/vampir/tutorial/jacobi.tar.gz .
$ tar xzvf jacobi.tar.gz
$ cd jacobi
```

- Change programming environment and load modules

```bash
$ module swap PrgEnv-{pgi,gnu}
$ module load cudatoolkit
$ module load scorep
```

- Compile benchmark and submit job

```bash
$ make
$ qsub -A [projid] run.pbs
$ less jacobi.o[JOB_ID]
```

Jacobi relaxation Calculation: 8192 x 8192 mesh with
2 processes and 16 threads + one Tesla K20X for each process.
614 of 2049 local rows are calculated on the CPU to balance the load
between the CPU and the GPU.
0, 0.489197
100, 0.002397
[...]
total: 8.425432 s

Keep time in mind!
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.pbs
  [...] export SCOREP_ENABLE_PROFILING=true
  export SCOREP_ENABLE_TRACING=false
  export SCOREP_EXPERIMENT_DIRECTORY=jacobi_mpi+openmp+cuda_profile
  export SCOREP_CUDA_ENABLE=yes
  export SCOREP_TIMER=(clock_gettime
  export SCOREP_MEMORY_RECORDING=yes
  [...] aprun -n 2 -d 16 -N 1 ./jacobi_mpi+openmp+cuda 8192 8192 0.15
  ```

  ```
  $ qsub -A [projid] run_profile.pbs
  $ less jacobi.o[JOB_ID]
  Jacobi relaxation Calculation: 8192 x 8192 mesh with
  2 processes and 16 threads + one Tesla K20X for each process.
  [...] total: 9.858350 s
  ```

  **15% Overhead!**
Demo: Jacobi Solver / Profile Analysis

- Perform flat profile analysis with `cube_stat`

```bash
$ cd bin.scorep
$ cube_stat -t 10 -p jacobi_mpi+openmp+cuda_profile/profile.cubex
```

<table>
<thead>
<tr>
<th>cube::Region</th>
<th>NumberOfCalls</th>
<th>ExclusiveTime</th>
<th>InclusiveTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>!$omp for @jacobi_cuda.c:188</td>
<td>32000.000000</td>
<td>131.797289</td>
<td>131.797289</td>
</tr>
<tr>
<td>!$omp implicit barrier</td>
<td>32000.000000</td>
<td>104.298683</td>
<td>104.298683</td>
</tr>
<tr>
<td>!$omp for @jacobi_cuda.c:258</td>
<td>32000.000000</td>
<td>42.999056</td>
<td>50.568642</td>
</tr>
</tbody>
</table>

- Perform call-path profile analysis with Cube

```bash
$ cube jacobi_mpi+openmp+cuda_profile/profile.cubex
```
Demo: Jacobi Solver / Scoring

- Do we need a filter? (Overhead and memory footprint)

$\texttt{scorep-score\_jacobi\_mpi+openmp+cuda\_profile/profile.cubex}$

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

$ cd ..
$ less run_trace.pbs

[...]  
export SCOREP_ENABLE_PROFILING=false  
export SCOREP_ENABLE_TRACING=true  
export SCOREP_EXPERIMENT_DIRECTORY=jacobi_mpi+openmp+cuda_trace  
export SCOREP_CUDA_ENABLE=yes  
export SCOREP_TIMER= clock_gettime  
export SCOREP_MEMORY_RECORDING=yes  
export SCOREP_TOTAL_MEMORY=50MB

[...]  
aprun -n 2 -d 16 -N 1 ./jacobi_mpi+openmp+cuda 8192 8192 0.15

$ qsub -A [projid] run_trace.pbs
$ less jacobi.o[JOB_ID]

Jacobi relaxation Calculation: 8192 x 8192 mesh with  
2 processes and 16 threads + one Tesla K20X for each process.  
614 of 2049 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

```
$ cd bin.scorep
$ module load vampir
$ vampir jacobi_mpi+openmp+cuda_trace/traces.otf2
```
Agenda

Performance Analysis Approaches
- Sampling vs. Instrumentation
- Profiling vs. Tracing

Score-P: Scalable Performance Measurement Infrastructure for Parallel Codes
- Motivation
- Functionality
- Architecture
- Workflow
- Advanced Features

Performance Analysis Tools
- Cube
- Vampir

Demo
- Performance Analysis of Jacobi Solver on Titan

Conclusions
Conclusions

Score-P

- Common instrumentation and measurement infrastructure for various analysis tools
- Hides away complicated details
- Provides many options and switches for experts

General Workflow

- Instrument your application with Score-P
- Perform a measurement run with **profiling enabled**
- Perform profile analysis with **Cube**
- Use `scorep-score` to define an appropriate filter
- Perform a measurement run with **tracing enabled** and the filter applied
- Perform in-depth analysis on the trace data with **Vampir**
If you have any questions or need help, please don't hesitate to contact me under winklerf@ornl.gov.

Detailed information under:

http://www.vi-hps.org/projects/score-p or
https://www.olcf.ornl.gov/support/software/
Score-P Advanced Features: Metrics

• Available PAPI metrics
  – Preset events: common set of events deemed relevant and useful for application performance tuning

```bash
$ papi_avail
```

  – Native events: set of all events that are available on the CPU (platform dependent)

```bash
$ papi_native_avail
```

• Available resource usage metrics

```bash
$ man getrusage
[... Output ...]
```

```c
struct rusage {
  struct timeval ru_utime; /* user CPU time used */
  struct timeval ru_stime; /* system CPU time used */
  [... More output ...]
```
Score-P Advanced Features: Metrics (2)

- Recording hardware counters via PAPI

\[
\textit{export \ SCOREP\_METRIC\_PAPI=PAPI\_TOT\_INS, PAPI\_FP\_INS}
\]

- Recording operating system resource usage

\[
\textit{export \ SCOREP\_METRIC\_RUSAGE=ru\_maxrss, ru\_stime}
\]
Vampir: Visualization Modes (1)

- Directly on front end or local machine

```
vampir
```
Vampir: Visualization Modes (2)

- On local machine with remote VampirServer

```
$ vampirserver start –n 16
```

```
$ vampir
```

Trace File (OTF2)

Many-Core Program

Score-P

Vampir Server

LAN/WAN

Large Trace File (stays on remote machine)

Parallel application

Performance Analysis at Scale: The Score-P Tools Infrastructure – Frank Winkler
Vampir Bonus: Case Study of FDS

- Identification of program phases

- Initialization Phase
  - Master thread: 1
  - Master thread: 207
  - Master thread: 364
  - Master thread: 480
  - Master thread: 664
  - Master thread: 864
  - Master thread: 1004
  - Master thread: 1306
  - Master thread: 1506
  - Master thread: 1769
  - Master thread: 2000
  - Master thread: 2138
  - Master thread: 2320
  - Master thread: 2500
  - Master thread: 2643
  - Master thread: 2909
  - Master thread: 3222
  - Master thread: 3522
  - Master thread: 3678
  - Master thread: 3967
  - Master thread: 4210
  - Master thread: 4414
  - Master thread: 4601
  - Master thread: 4746
  - Master thread: 4910
  - Master thread: 5005
  - Master thread: 5311
  - Master thread: 5410
  - Master thread: 5605
  - Master thread: 5799
  - Master thread: 5975
  - Master thread: 6176
  - Master thread: 6400
  - Master thread: 6608
  - Master thread: 6798
  - Master thread: 7003
  - Master thread: 7153
  - Master thread: 7322
  - Master thread: 844

- Computation Phase

  - Function Summary
    - All Processes, Accumulated Exclusive Time per Function
      - 2,537,706.44 s
      - 2,300,760.654 s
      - 1,997,224.331 s
      - 464,258.627 s
      - 258,161.183 s
      - 98,383.802 s
      - 77,675.037 s
      - 51,696.982 s
      - 48,432.479 s
      - 41,030.476 s
      - 14,804.909 s
      - 2,256.608 s
      - 1,976.264 s

  - Context View
    - Property: Trace Info
      - File: Chester:/lustre/atlas/proj-shared/stf010/winkler/MPI_SCALE/reference_8192_8_procs_trace_filter/traces.of2
      - Creator: Score-P 2.0-trunk
      - Version: 2.0
      - Number of Processes: 8,192
      - Timer Resolution: 454.540618 ps
**Vampir Bonus: Case Study of FDS**

- Load imbalance in initialization phase

Master thread:0 is reading input files. All other processes are waiting in MPI_Barrier.
Vampir Bonus: Case Study of FDS

• Load imbalance in initialization phase (2)

Initialization time increases with the process index.
Vampir Bonus: Case Study of FDS

- Computation phase

12% communication and 88% computation during computation phase.
Vampir Bonus: Case Study of FDS

- Unnecessary synchronization in computation phase

MPI_Barrier unneeded.
Vampir Bonus: Case Study of FDS

- Inefficient cache usage in computation phase

Low Flops/s rate due to a higher L2 cache miss rate.