Performance Analysis with Vampir

Matthias Weber
Technische Universität Dresden
Outline

- **Part I: Performance Analysis with Vampir & Score-P**
  - Introduction/Performance Engineering
  - Score-P – Measurement System
  - Vampir – Performance Data Visualization
- **Part II: Vampir Analysis Exercise**
  - Analysing Four Application Traces
- **Part III: Vampir Hands-On**
  - Visualizing and Analyzing NPB-MZ-MPI / BT
Introduction
Performance Engineering
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%)
- Total: \( \frac{1}{2} \times \frac{1}{16} \times \frac{1}{4} \times \frac{1}{100} \)
  \(~\frac{1}{10000}\) of the peak performance
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.
Virtual Institute – High Productivity Supercomputing

- **Goal**: Improve the quality and accelerate the development process of complex simulation codes running on highly-parallel computer systems
- **Start-up funding (2006–2011)**
  by Helmholtz Association of German Research Centres

**Activities**
- Development and integration of HPC programming tools
  - Correctness checking & performance analysis
- Academic workshops
- Training workshops
- Service
  - Support email lists
  - Application engagement

http://www.vi-hps.org
VI-HPS partners (founders)

Forschungszentrum Jülich
  - Jülich Supercomputing Centre

RWTH Aachen University
  - Centre for Computing & Communication

Technische Universität Dresden
  - Centre for Information Services & HPC

University of Tennessee (Knoxville)
  - Innovative Computing Laboratory
VI-HPS partners (cont.)

- Arm Ltd.
  - Allinea Software

- Barcelona Supercomputing Center
  - Centro Nacional de Supercomputación

- Lawrence Livermore National Lab.
  - Center for Applied Scientific Computing

- Leibniz Supercomputing Centre

- Technical University of Darmstadt
  - Laboratory for Parallel Programming
**VI-HPS partners (cont.)**

<table>
<thead>
<tr>
<th>Technical University of Munich</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chair for Computer Architecture</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>University of Oregon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance Research Laboratory</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>University of Stuttgart</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPC Centre</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>University of Versailles St-Quentin</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRC ITACA</td>
</tr>
</tbody>
</table>
Productivity tools

- **MUST & Archer**
  - MPI & OpenMP usage correctness checking
- **PAPI**
  - Interfacing to hardware performance counters
- **Periscope Tuning Framework**
  - Automatic analysis and Tuning
- **Scalasca**
  - Large-scale parallel performance analysis
- **TAU**
  - Integrated parallel performance system
- **Vampir**
  - Interactive graphical trace visualization & analysis
- **Score-P**
  - Community-developed instrumentation & measurement infrastructure

See VI-HPS Tools Guide for a overview of tools: https://www.vi-hps.org/tools
Productivity tools (cont.)

- **FORGE DDT/Map/PR**: Parallel debugging, profiling & performance reports
- **Extra-P**: Automated performance modelling
- **KcacheGrind**: Callgraph-based cache analysis [x86 only]
- **JUBE**: Workflow execution environment
- **MAQAO**: Assembly instrumentation & optimization [x86-64 only]
- **mpiP/mpiPview**: MPI profiling tool and analysis viewer
- **Open MPI**: Integrated memory checking
- **Open|SpeedShop**: Integrated parallel performance analysis environment
- **Paraver/Dimemas/Extrae**: Event tracing, graphical trace visualization & analysis
- **Rubik**: Process mapping generation & optimization [BG only]
- **SIONlib/Spindle**: Optimized native parallel file I/O & shared library loading
- **STAT**: Stack trace analysis tools
Technologies and their integration

- KCACHEGRIND
- MAP/PR / MPIP / O|SS / MAQAO
- TAUP
- EXTRA-P
- PERISCOPE
- SCALASCA
- SCORE-P / EXTRAE
- VAMPIR
- PARAVER
- JUBE
- Optimization
- Execution
- Visual trace analysis
- Automatic profile & trace analysis
- Debugging, error & anomaly detection
- Hardware monitoring
- PAPI
- MUST / ARCHER
- DDT
- STAT
- MEMCHECKER / SPINDLE / SIONLIB
- PTF / RUBIK / MAQAO
Recording and Studying a Program’s Behavior

- Performance analysis in practice:
  - Attach measurement system Score-P to application
  - Run application with Score-P monitor
    - Result: trace/profile data
- Study the trace data with Vampir
- Repeat
  - Adapt instrumentation ("what you measure")
  - Evaluate result of a change
Methodology: Sampling

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

- Measurement code is inserted such that every event of interest is captured directly
- Advantage:
  - Much more detailed information
- Disadvantage:
  - Processing of source-code / executable necessary
  - Large relative overheads for small functions
Instrumentation vs. Sampling

- **Instrumentation/Tracing:**
  - Typically more difficult than using a profiler
  - Does not guarantee anything about overhead or recording size
    - It depends on the event rate
      - E.g. an MPI-only trace has very low overhead
    - Filtering required

- **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
Methodology: Profiling vs. Tracing

- **Statistics**
  - Number of Invocations

- **Timelines**

```
main  foo  bar  foo
main  foo  bar  foo
```
Terms Used and How They Connect

- **Profiling**
  - **Statistics**
  - **Summarization**

- **Tracing**
  - **Timelines**
  - **Logging**
  - **Event-based Instrumentation**

- **Data Acquisition**
  - **Sampling**

- **Data Recording**
  - **Analysis Layer**

- **Data Presentation**
  - **Analysis Technique**
Score-P
Measurement System
Fragmentation of tools landscape

- 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 OTF</td>
<td>EPILOG / CUBE</td>
<td>TAU native formats</td>
<td>Online measurement</td>
</tr>
</tbody>
</table>
Score-P project idea

- Start a community effort for a common infrastructure
  - Score-P instrumentation and measurement system
  - Common data formats OTF2 and CUBE4
- Partners:
  - TU Dresden, FZ Jülich, TU München, University of Oregon, RWTH Aachen, TU Darmstadt
- Developers save manpower by sharing development resources
- Users need only a single installation, resulting in single learning curve
- Ensure a single official release version at all times which will always work with the tools
- Commitment to joint long-term cooperation
  - Development based on meritocratic governance model
  - Open for contributions and new partners
VIRTUAL INSTITUTE – HIGH PRODUCTIVITY SUPERCOMPUTING

Vampir  Scalasca  CUBE  TAU  TAUdb  Periscope

Event traces (OTF2)

Call-path profiles (CUBE4, TAU)

Score-P measurement infrastructure

Instrumentation wrapper

Process-level parallelism (MPI, SHMEM)
Thread-level parallelism (OpenMP, Pthreads)
Accelerator-based parallelism (CUDA, OpenCL, OpenACC)
I/O Activity Recording (Posix I/O, MPI-IO)
Source code instrumentation (Compiler, PDT, User)
Sampling interrupts (PAPI, PERF)

Hardware counter (PAPI, rusage, PERF, plugins)

Online interface
Data Collection with Score-P

- **Instrument (Compile & Link)**

  \[
  \begin{align*}
  \text{CC} & = \text{icc} \\
  \text{CXX} & = \text{icpc} \\
  \text{F90} & = \text{ifc} \\
  \text{MPICC} & = \text{mpicc}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{CC} & = \text{scorep} \ <\text{options}> \ \text{icc} \\
  \text{CXX} & = \text{scorep} \ <\text{options}> \ \text{icpc} \\
  \text{F90} & = \text{scorep} \ <\text{options}> \ \text{ifc} \\
  \text{MPICC} & = \text{scorep} \ <\text{options}> \ \text{mpicc}
  \end{align*}
  \]

- **Execute**

  \[
  \begin{align*}
  $ & ./\text{a.out} \\
  $ & ./\text{mpirun -np 2 .}/\text{a.out}
  \end{align*}
  \]

- **Inspect**

  \[
  \begin{align*}
  $ & \text{ls -R} \\
  \text{scorep-20170323_1309_7243761919249966 a.out}
  \end{align*}
  \]

  \[
  \begin{align*}
  ./\text{scorep-20170323_1309_7243761919249966: profile.cubex scorep.cfg}
  \end{align*}
  \]
Data Collection with Score-P

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
   [...]
```
Data Collection with Score-P

Example for generating and loading a trace:

```
$ 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

$ vampir scorep-20170323_1309_7243761919249966/traces.otf2
```
Data Collection with Score-P

- Score-P supports a combination of instrumentation/sampling:
  - Instrumentation for MPI/OpenMP events
  - Sampling for everything else

- Simple configuration, e.g.:

```
% export SCOREP_ENABLE_TRACING=true
% export SCOREP_ENABLE_UNWINDING=true
% export SCOREP_SAMPLING_EVENTS=perf_cycles@2000000
```
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

### Filter file:

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

```bash
SCOREP_REGION_NAMES_BEGIN EXCLUDE
matmul_sub
matvec_sub
binvcrhs
```

<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>binvcrhs</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>
Good Tracing Practices

- Traces can become large; Their handling challenging
  - Trace size proportional to: number of processes/threads (width), duration (length), and measurement detail (depth)

- Intermediate flushes => High degree of perturbation
  - Either use less detail or larger trace buffers

- Traces should be written to a parallel file systems
  - E.g. “/work”, “/scratch”, or “/p/lscratchc”

- Moving large traces can become challenging
  - However, systems with more memory/core can analyze larger traces
  - Alternatively, undersubscribe for increased memory/core ratios
Vampir
Performance Data Visualization
Event Trace Visualization with Vampir

- Visualization of dynamic runtime behaviour at any level of detail along with statistics and performance metrics
- Alternative and supplement to automatic analysis

**Typical questions that Vampir helps to answer**

- What happens in my application execution during a given time in a given process or thread?
- How do the communication patterns of my application execute on a real system?
- Are there any imbalances in computation, I/O or memory usage and how do they affect the parallel execution of my application?

**Timeline charts**
- Application activities and communication along a time axis

**Summary charts**
- Quantitative results for the currently selected time interval
Event Trace Visualization with Vampir
The value of seeing how an application executes on the machine

- Application code computing coulomb forces
- The workload was distributed evenly across available processes
- The user expected perfect parallelized code
- However the underlying algorithm worked differently than expected

Visualization of the application execution instantly shows a problem in the parallelization approach

Large imbalance instantly visible

More than 50% application time wasted!
Main Performance Charts of Vampir

Timeline Charts
- Master Timeline: all threads’ activities
- Process Timeline: single thread’s activities
- Summary Timeline: all threads’ function call statistics
- Performance Radar: all threads’ performance metrics
- Counter Data Timeline: single threads’ performance metrics
- I/O Timeline: all threads’ I/O activities

Summary Charts
- Function Summary
- Message Summary
- I/O Summary
- Process Summary
- Communication Matrix View
- Call Tree
Visualization Modes (1)
Directly on front end or local machine

% vampir

Multi-Core Program

Score-P

Trace File

Vampir

Small/Medium sized trace

Thread parallel
Visualization Modes (2)
On local machine with remote VampirServer

% vampirserver start

% vampir

VampirServer

Many-Core Program

Score-P

Vampir

Trace File

LAN/WAN

Large Trace File (stays on remote machine)

Parallel application
Visualization: After Tracing

I/O System

Compute Nodes (Batch jobs)

Login Nodes

Dekstop System

Trace File (OTF2)
Visualization: Most simple (Analysis on Desktop)

- Minimal setup (no installations, no batch job)
- Copying of traces to desktop
- Only small traces

Visualization and analysis: Vampir
Visualization: Best Option (Analysis on HPC system)

Analysis: **VampirServer**

- **TCP Socket connection**

**Visualization:** **Vampir**

- **I/O System**
  - Trace File (OTF2)

- **Compute Nodes (Batch jobs)**
  - **Login Nodes**

- **Desktop System**

+ Best performance, low response time
- Tunneling to connect to batch job
- Installation on desktop system needed
Visualization: Alternative (Analysis on HPC system)

+ Simpler setup, no installation on desktop
- X11 forwarding needed (use: `ssh -X` ...)
- Bandwidth and response time can be critical

**Analysis:** VampirServer

**Visualization:** Vampir
Visualization: Most simple (Analysis on Frontend)

- Minimal setup (no installations, no batch job)
- X11 forwarding, bandwidth, and response
- Only small traces

Visualization and analysis: **Vampir**
Example Analysis:
Visualizing and Analyzing NPB-MZ-MPI / BT
Visualization of the NPB-MZ-MPI / BT trace
Visualization of the NPB-MZ-MPI / BT trace

Master Timeline

Detailed information about functions, communication and synchronization events for collection of processes.
Visualization of the NPB-MZ-MPI / BT trace

Process Timeline

Detailed information about different levels of function calls in a stacked bar chart for an individual process.
Visualization of the NPB-MZ-MPI / BT trace

Typical program phases

Initialisation Phase

Computation Phase
Visualization of the NPB-MZ-MPI / BT trace

Counter Data Timeline

Detailed counter information over time for an individual process.
Visualization of the NPB-MZ-MPI / BT trace Performance Radar

Detailed counter information over time for a collection of processes.
Visualization of the NPB-MZ-MPI / BT trace
Zoom in: Initialisation Phase

Context View: Detailed information about function "initialize_".
Visualization of the NPB-MZ-MPI / BT trace

Find Function

Execution of function “initialize_” results in higher page fault rates.
Visualization of the NPB-MZ-MPI / BT trace

Computation Phase

Computation phase results in higher floating point operations.
Visualization of the NPB-MZ-MPI / BT trace
Zoom in: Computation Phase

MPI communication results in lower floating point operations.
Visualization of the NPB-MZ-MPI / BT trace
Zoom in: Finalisation Phase

“Early reduce” bottleneck.
Visualization of the NPB-MZ-MPI / BT trace

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

Function Summary:
Overview of the accumulated information across all functions and for a collection of processes.
Visualization of the NPB-MZ-MPI / BT trace

Process Summary

Find groups of similar processes and threads by using summarized function information.
I/O Features
Operations over Time

Individual I/O operation

I/O runtime contribution
I/O Features
Data Rates over Time

![Graph showing I/O data rates over time](image)

I/O data rate of single thread
I/O Features
Summaries with Totals

Other metrics:
- IOPS
- I/O time
- I/O size
- I/O bandwidth
I/O Features
Summaries per File

Aggregated data for specific resource
I/O Features
Operations per File

Focus on specific resource
Show all resources
Summary and Conclusion
Summary

- **Vampir & VampirServer**
  - Interactive trace visualization and analysis
  - Intuitive browsing and zooming
  - Scalable to large trace data sizes (20 TiByte)
  - Scalable to high parallelism (200,000 processes)

- **Vampir for Linux, Windows, and Mac OS X**