Performance Analysis Tools: Vampir, VampirTrace, and Score-P

Joseph Schuchart

schuchartj@ornl.gov

April 9th, 2013
Performance tools will not automatically make your program run faster! However, they will help you understand the runtime behavior and find points for optimizations!
1. Introduction
2. The Vampir Toolset
   - Vampir: Graphical Performance Analysis
   - VampirTrace: Instrumentation and Measurement
3. Score-P: Consolidating Performance Measurement
4. Conclusion
5. Bonus: Visible Performance Bottlenecks
Why Performance Analysis?

• Resources are limited and expensive
  – Even if you don’t pay for them!
• Improve the scalability of your application
  – Allows to increase simulation size
• Faster turn-around times
Chances are there is potential for optimizations!
- Thread synchronization
- MPI Communication
- I/O behavior
- Cache usage
- ...
BigRed II is slower than machine XYZ!? 

- Most likely it’s not the machine itself!
- Codes might have to be adapted to new machines
- Reasons for performance degradations not always obvious
  - Problems starting at certain scales
  - Surprising side effects
  - MPI fine-tuning
An Example: How not to do I/O!

Large amount of I/O
Only little computation

Cause: Writing to one file!
An Example: Better! But wait…?!

User functions

All waiting for one!
Optimization in the Development Process

• Have an optimization phase
  – Just like you have a testing phase
  – Look at how well new code behaves

• Use available tools
  – Debugger
    • gdb
    • Totalview
  – Performance analysis tools
    • CrayPat
    • Vampir
  – No, printf() is not a suitable tool!
Overview

1. Introduction
2. The Vampir Toolset
   - Vampir: Graphical Performance Analysis
   - VampirTrace: Instrumentation and Measurement
3. Score-P: Consolidating Information Recording
4. Conclusion
5. Bonus: Visible Performance Bottlenecks
Profiling vs Tracing

- **Profiling:**
  - Collect aggregated information (time, counts, …)
  - Global or per process/thread

- **Tracing:**
  - Save individual event records
  - With precise timestamps
  - Per process/thread
  - Add event specific information
  - Represented as Timeline:
1. **Instrument** your application using VampirTrace
2. **Run** your application with an appropriate test-set
   - Should only run for a few minutes
3. **Analyze** your trace file with Vampir:
   - For small traces, you can use the (local) Vampir GUI
   - For larger traces (>32 processes):
     - Start a VampirServer (MPI Application)
     - Connect with your local Client

---

**Overview of Vampir – Jens Domke**

**2.2 Workflow Overview**

![Workflow Diagram]

Multi-Core Program → Vampir Trace → Trace File (OTF) → Vampir Toolset → Vampir Server → Trace Bundle → Vampir GUI
Vampir: Trace Visualization

- Arbitrary level of detail:
  - Overview of whole runtime
  - Function level details
- Interactive browsing, zooming, selection
Vampir: Process Information

Process stack view

Metric display
Performance Radar (Metric Heat Map)
Choose Opacity
Vampir: Function Summary

Function Summary

All Processes, Accumulated Exclusive Time per Function

Function | Time (s)
---|---
viscosity_mod_biharmonic_wk | 168.124 s
viscosity_mod_calar_minmax | 161.472 s
prim_advance_nd_apply_flux | 157.74 s
prim_advection_dlayer_steo | 150.278 s
prim_advection_dlayer_mod | 49.288 s
prim_advection_dlayer_mode | 44.918 s
prim_advection_dlayer_viscosity | 41.657 s
prim_advection_no_hypervisc | 38.512 s
prim_advection_str | 31.974 s
prim_advection_volvisc | 30.334 s
prim_advection_volvisc_viscosity | 29.815 s
prim_differential | 26.555 s
prim_differential_viscosity | 20.18 s
prim_differential_viscosity_viscosity | 19.838 s
prim_differential_viscosity_viscosity_viscosity | 19.451 s
prim_differential_viscosity_viscosity_viscosity_viscosity | 18.526 s
prim_differential_viscosity_viscosity_viscosity_viscosity_viscosity | 15.906 s
prim_differential_viscosity_viscosity_viscosity_viscosity_viscosity_viscosity | 12.172 s
prim_differential_viscosity_viscosity_viscosity_viscosity_viscosity_viscosity_viscosity | 11.294 s
prim_differential_viscosity_viscosity_viscosity_viscosity_viscosity_viscosity_viscosity_viscosity | 9.608 s
derivative_mo_ce_sphere_wk | 9.405 s
derivative_mo_ce_sphere_wk_wk | 9.387 s
perf_mod_t_cropf | 9.187 s
perf_mod_t_cropf | 8.96 s
perf_mod_t_cropf | 8.64 s
tropopause_btwmo | 8.273 s	naoav_salicwme | 8.268 s
tfrac_fraction_cldfrc | 7.722 s
derivative_mo_ce_sphere_wk | 6.966 s
derivative_mo_ce_sphere_wk | 6.717 s
derivative_mo_ce_sphere_wk | 6.341 s
derivative_mo_ce_sphere_wk | 5.74 s
derivative_mo_ce_sphere_wk | 5.346 s
MPI_Allreduce | 5.204 s
time_manager_time_fromdate | 5.134 s
tfrac_fraction_cldfrc | 4.888 s
prim_advection_r_memap | 4.436 s
prim_advection_r_memap | 4.218 s
prim_advection_r_memap | 3.905 s
tracer_data_vert_interp | 3.871 s
prim_differential_hys_writeout | 3.826 s
prim_advection_hos_minmax | 3.796 s
vertical_diffusion_diffusion_tend | 3.727 s
derivative_mo_gen_sphere | 3.588 s
prim_advection_r_memap | 3.467 s
derivative_mod_tricity_sphere | 3.397 s
prim_advection_r_memap | 3.391 s
derivative_mo_ce_sphere_wk | 3.251 s
derivative_mo_dient_sphere | 3.209 s
time_manager_time_fromdate | 3.184 s
time_manager_time_fromdate | 3.026 s

MPI_Allreduce [2.12%]
MPI_Comm_commit [4.51%]
MPI_Comm_free [9.16%]
MPI_Barrier [12.35%]
MPI_Wait [19.17%]
MPI_Waitall [31.88%]
Vampir: Communication Matrix
• How to get the trace data?
  1. Use VampirTrace :)
  2. Change compiler to use compiler wrappers

    \[
    \begin{align*}
    \text{CC} & = \text{cc} \\
    \text{CXX} & = \text{CC} \\
    \text{F90} & = \text{ftn} \\
    \text{NVCC} & = \text{nvcc}
    \end{align*}
    \quad \rightarrow \quad
    \begin{align*}
    \text{CC} & = \text{vtcc} \\
    \text{CXX} & = \text{vtcxx} \\
    \text{F90} & = \text{vtf90} \\
    \text{NVCC} & = \text{vtnvcc}
    \end{align*}
    \]

  3. Recompile / Relink
  4. Run with appropriate test-set
• What does it do in the background?
  – Automatic function instrumentation:
    • Using specifics of the underlying compiler
    • Using tools like TAU and Dyninst
  – Links instrumentation libraries
    • MPI
    • POSIX Threads
    • libc: I/O, memory, and other system calls
    • CUDA, CUPTI
  – Instruments OpenMP statements (using Opari)
• Launch your application with an appropriate test-set
• Measurement library performs:
  – Event data collection
    • Triggered by function calls
  – Records selected performance metrics
  – Precise time measurement
    • Parallel timer synchronization
  – Filtering and grouping of function calls
  – Monitor accelerator
• Measurement is controlled via environment variables
• Event records stored in internal buffer
  – Written to file at the very end
VampirTrace: GPU Tracing

• Analyze usage of GPU:
  – Interaction with CPU
  – Kernel activity and GPU related metrics
• Measurement controlled via environment variables
• GPU streams displayed as:
  CUDA[device:stream] process:thread
Tracing: Pros and Cons

• **Advantages:**
  – Preserve spatial and temporal relationship of events
  – Allows reconstruction of dynamic behavior
  – Easy to calculate profiles from trace data

• **Disadvantages:**
  – Tracing may cause perturbation
  – Traces may become very large
1. Introduction
2. The Vampir Toolset
   – Vampir: Graphical Performance Analysis
   – VampirTrace: Instrumentation and Measurement
3. **Score-P: Consolidating Performance Measurement**
4. Conclusion
5. **Bonus: Visible Performance Bottlenecks**
• Current tools ecosystem fragmented
  – Limited compatibility
  – High learning effort

• Goal: common performance measurement runtime infrastructure
  – Concentrate development efforts
  – Improve user experience
  – Meant to replace existing measurement tools, e.g. VampirTrace

• Supported tools:
  – Periscope
  – TAU
  – Scalasca
  – Vampir
The Whole Ensemble

Periscope

Online interface

Score-P

Instr. target application

TAU ParaProf

CUBE4 report

CUBE4 report

Scalasca wait-state analysis

OTF2 traces

TAU PerfExplorer

TAU

Instr. target application

PAPI

TAU

ParaProf

CUBE

Vampir

SC’12: Workshop on Extreme-Scale Performance Tools
Score-P: Usage

- Similar to VampirTrace
  - Only one compiler wrapper
  - Detects parallelization paradigms automatically
  - Links instrumentation libraries
- Recompile and run
- Customizable via environment Variables
  - Switch between tracing and profiling
  - (De-)Activate parts of the instrumentation (metrics, MPI function groups)
- Customizable via files (filter functions)
Further Improvements

• Opapi2: OpenMP instrumentation
  – Support OpenMP task constructs
  – Upcoming support for the OpenMP tools API

• Support for RMA
  – SHMEM
  – MPI One-Sided
  – Any PGAS

• Online Access Interface
• Lower measurement overhead
• Robust and stable
• Ensure a single release usable with all tools

Scalasca
• Scalable performance analysis toolkit for parallel codes
  – Specifically targeting large-scale applications running on 10,000s to 100,000s of cores

• Integrated performance analysis process
  – Performance overview via call-path profiles
  – In-depth study of application behavior via event tracing, automatic trace analysis identifying wait states
  – Switching between both without re-compilation or re-linking

• Supports MPI 2.2 and basic OpenMP

TAU Performance System
• Very portable tool set for instrumentation, measurement and analysis of parallel applications
• The “Swiss army knife” of performance analysis
• Instrumentation API supports choice – between profiling and tracing of metrics (e.g., time, HW counter, ...)
• Supports – C, C++, Fortran, HPF, HPC++, Java, Python
  – MPI, OpenMP, POSIX threads, Java, Win32, ...
• License: Open Source
Overview

1. Introduction
2. The Vampir Toolset
   - Vampir: Graphical Performance Analysis
   - VampirTrace: Instrumentation and Measurement
3. Score-P: Consolidating Information Recording
4. Conclusion
5. Bonus: Visible Performance Bottlenecks
Summary

• VampirTrace
  – Convenient instrumentation and measurement infrastructure
  – Hides complex details
  – Highly configurable
    • Provides many options and switches for expert users
  – Available under BSD-like license
    • Part of Open MPI >=1.3
    • Standalone download at www.tu-dresden.de/zih/vampirtrace
Summary

• **Score-P**
  - Consolidates development efforts
  - One measurement tool for different analysis tools
    • TAU, Scalasca, Vampir, Periscope
  - Simplified use of wrappers
  - Improved measurement overhead
  - Target for future developments
  - Will replace VampirTrace
Summary

• **Vampir GUI and VampirServer**
  – Interactive trace visualization and analysis
  – Intuitive browsing and zooming
  – Scalable to large trace data sizes (1.5 TB)
  – Scalable to high parallelism (200,000 trace processes)
  – Vampir GUI available for Windows, Linux/Unix, Mac OS X
  – Further information: [www.vampir.eu](http://www.vampir.eu)

• **Support:**
  – [vampirsupport@zih.tu-dresden.de](mailto:vampirsupport@zih.tu-dresden.de)
  – Your friendly UITS staff
Overview

1. Introduction
2. The Vampir Toolset
   - Vampir: Graphical Performance Analysis
   - VampirTrace: Instrumentation and Measurement
3. Score-P: Consolidating Information Recording
4. Conclusion
5. Bonus: Visible Performance Bottlenecks
Visible Performance Bottlenecks

• Inefficient communication patterns
• Load imbalances (processes / threads) or serial parts
• Memory bound computation
  – Inefficient cache usage and TLB misses
  – Use PAPI counter to detect
• I/O bottlenecks
• Most time consuming functions
  – Good start for optimizations
• …
• Too much runtime share for MPI all-together
Sequential P2P

7.3 Sequential point-to-point messages

- Chains of MPI Sendrecv

→ Replace by faster overlapping series of Send, Irecv, and Waitall
Many MPI barriers are unneeded

- Remove barriers if possible, following operations start less regular but are faster.
Low Parallel Efficiency

- Mostly idle OpenMP threads

(Zoom)
Inefficient L1/L2/L3 Cache Usage

High rate of Flop/s with low rate of L3 cache misses

Low Flop/s rate due to a high L3 miss rate
Interested?

Contact UITS for help:
researchtechnologies@iu.edu
Additional Information
**VampirTrace: Environment Variables**

- Measurement controlled via environment variables

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>VT_BUFFER_SIZE</td>
<td>Size of event buffer</td>
<td>32M</td>
</tr>
<tr>
<td>VT_MAX_FLUSHES</td>
<td>Maximum number of flushes</td>
<td>1</td>
</tr>
<tr>
<td>VT_PFORM_GDIR</td>
<td>Directory to hold trace data</td>
<td>./</td>
</tr>
<tr>
<td>VT_METRICS</td>
<td>Colon-separated list of PAPI metrics to record</td>
<td>-</td>
</tr>
<tr>
<td>VT_FILTER_SPEC</td>
<td>Define a filter file for functions</td>
<td>-</td>
</tr>
<tr>
<td>VT_IOTRACE</td>
<td>Enable I/O tracing</td>
<td>no</td>
</tr>
<tr>
<td>VT_MEMTRACE</td>
<td>Enable memory tracing</td>
<td>no</td>
</tr>
<tr>
<td>VT_MODE</td>
<td>Colon-separated list of VampirTrace modes: Tracing (TRACE) or Profiling (STAT)</td>
<td>TRACE</td>
</tr>
<tr>
<td>VT_VERBOSE</td>
<td>Controls verbosity: Quiet (0), Warnings (1), Information (2)</td>
<td>0</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>cuda</td>
<td>Enable support for CUDA</td>
<td></td>
</tr>
<tr>
<td>runtime</td>
<td>CUDA runtime API tracing</td>
<td></td>
</tr>
<tr>
<td>driver</td>
<td>CUDA driver API tracing</td>
<td></td>
</tr>
<tr>
<td>kernel</td>
<td>Record kernel activity</td>
<td></td>
</tr>
<tr>
<td>idle</td>
<td>Record GPU idle times</td>
<td></td>
</tr>
<tr>
<td>memcpy</td>
<td>Record GPU memory copies</td>
<td></td>
</tr>
<tr>
<td>stream_reuse</td>
<td>force reusing of CUDA streams after cudaStreamDestroy()</td>
<td></td>
</tr>
<tr>
<td>memusage</td>
<td>Record GPU memory usage</td>
<td></td>
</tr>
<tr>
<td>debug</td>
<td>CUDA tracing debug mode</td>
<td></td>
</tr>
<tr>
<td>error</td>
<td>CUDA errors will exit the program</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>default</td>
<td>Same as “cuda,runtime,kernel,memcpy”</td>
</tr>
<tr>
<td>no</td>
<td>Disable CUDA measurement</td>
<td></td>
</tr>
</tbody>
</table>
### Additional Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VT_GPUTRACE_KERNEL=[yes</td>
<td>2]</td>
</tr>
<tr>
<td>VT_CUDATRACE_SYNC=[0</td>
<td>1</td>
</tr>
<tr>
<td>VT_CUPTI_METRICS</td>
<td>Record additional CUPTI metrics, e.g., VT_CUPTI_METRICS=local_store:local_load</td>
</tr>
<tr>
<td>VT_CUPTI_SAMPLING=[yes</td>
<td>no]</td>
</tr>
<tr>
<td>VT_GPUTRACE_MEMUSAGE=[yes</td>
<td>2]</td>
</tr>
</tbody>
</table>

- For CUPTI counter overview:
  - [http://docs.nvidia.com/cuda/cupti/index.html#r_metric_api](http://docs.nvidia.com/cuda/cupti/index.html#r_metric_api)
## Score-P: Environment Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCOREP_ENABLE_PROFILING</td>
<td>true</td>
</tr>
<tr>
<td><strong>SCOREP_ENABLE_TRACING</strong></td>
<td>false</td>
</tr>
<tr>
<td>SCOREP_VERBOSE</td>
<td>false</td>
</tr>
<tr>
<td>SCOREP_TOTAL_MEMORY</td>
<td>16000k</td>
</tr>
<tr>
<td>SCOREP_EXPERIMENT_DIRECTORY</td>
<td>scorep_&lt;date_and_time&gt;</td>
</tr>
<tr>
<td>SCOREP_OVERWRITE_EXPERIMENT_DIRECTORY</td>
<td>true</td>
</tr>
<tr>
<td>SCOREP_TRACING_COMPRESS</td>
<td>false</td>
</tr>
<tr>
<td>SCOREP_FILTERING_FILE</td>
<td></td>
</tr>
<tr>
<td>SCOREP_METRIC_PAPI</td>
<td></td>
</tr>
<tr>
<td>SCOREP_METRIC_RUSAGE</td>
<td></td>
</tr>
<tr>
<td>SCOREP_SELECTIVE_CONFIG_FILE</td>
<td></td>
</tr>
<tr>
<td>SCOREP_CUDA_ENABLE</td>
<td>false</td>
</tr>
<tr>
<td>SCOREP_MPI_ENABLE_GROUPS</td>
<td>default</td>
</tr>
<tr>
<td>SCOREP_MPI_HIERARCHICAL_UNIFY</td>
<td>true</td>
</tr>
</tbody>
</table>
• Compiler Instrumentation
  – Default
  – Instrumentation points inserted by compiler
  – Easy, but quite verbose
    • Fills up buffers fast
    • Problematic for C++ code
  – Solution: use filter
    • At runtime: VT_FILTER_SPEC=filter.txt
      – Reduces trace data size
      – Does not reduce overhead
    • If using the GNU compiler:
      -finstrument-functions-exclude-file-list=include/g++,math_extra
      -finstrument-functions-exclude-function-list=map,timing,Timer,operator
VampirTrace: Instrumentation Types

• Instrumentation using TAU/PDT
  – TAU/PDT rewrites source code to insert instrumentation points
  – Specify on command line:

  vtcc -vt:inst tauinst

  – Allows for fine-grained instrumentation filtering
    • Compile-time filter
    • Reduces runtime overhead
    • Matches * and ?
    • Specify filter using

  vtcc -vt:inst tauinst \ -vt:tau -f filter.txt

  BEGIN_EXCLUDE_LIST
  init_*
  boost::*
  END_EXCLUDE_LIST

  BEGIN_FILE_EXCLUDE_LIST
  /usr/include/*
  END_FILE_EXCLUDE_LIST
VampirTrace: Instrumentation Types

- **Binary Instrumentation using Dyninst**
  - Relink application using
    - **vtcc -vt:inst dyninst**
  - Launch using **vtdyn**
    - Rewrites program binary to insert function instrumentation
  - **Pros:**
    - Shows compiler-optimized program behavior
    - Reduces overhead
  - **Cons:**
    - Does not instrument inline functions
    - Trace might not correlate with source code
• **Manual Instrumentation**
  – Additional information not captured by automatic instrumentation
  – Function calls: VT_START, VT_END
    • Mark certain phases in the execution
  – Counter values: VT_COUNT_[TYPE]_VAL
    • Record loop iteration counts
    • Record function arguments
  – Communication: VT_SEND, VT_RECV
    • Add (virtual) communication
    • Example: Track lock acquisition
  – Marker: VT_MARKER
    • Mark interesting parts of the trace, e.g., based on conditions
• Turn tracing on/off: VT_ON, VT_OFF
  – Only trace interesting areas of the application
• Buffer rewind: VT_SET_REWIND_MARK(), VT_REWIND()
  – Rewind the buffer up to a set mark
  – Drops all records after the mark
  – Usage: selective tracing based on run-time behavior
• Force buffer flush
• Force timer synchronization
• Force counter update
Periscope

• Automatic online performance analysis
• Iterative online profiling using Score-P

• Automatic Search for performance Inefficiencies:
  – MPI Wait states
  – Single core stalls
  – OpenMP overhead
  – OpenMP scalability

• Eclipse integration
• Scalable performance-analysis toolkit for parallel codes
• Specifically targeting large-scale applications
  – 10,000s to 100,000s of cores
• Integrated performance-analysis process

• Performance overview via call-path profiles
• Automatic event trace analysis
  – Focus on communication & synchronization
• License: NewBSD
• Very portable toolset for instrumentation, measurement and analysis of parallel applications
• The “swiss army knife” of performance analysis
• Uses Score-P for profiling and tracing

• Supports
  – C, C++, UPC, Fortran, HPF, Java, Python, Chapel
  – MPI, OpenMP, POSIX Threads, Java, Win32, OpenCL...
• 3D profile browser, ParaProf, data mining and cross experiment analysis
• BSD-Style license
Using VampirTrace on BRII

• VampirTrace:
  – Wrappers might not determine the parallelization paradigm:
    • Use –vt:mpi for MPI applications
    • Use –vt:mt for threaded applications
    • Use –vt:hyb for MPI + threaded applications
  – Increase the size of the buffers:
    • VT_BUFFER_SIZE=128M or more
    • But make sure there is enough memory left for the application
  – Create a profile first to get an overview:
    • VT_MODE=STAT
    • Use it to create instrumentation filters (if necessary)

• Vampir
  – Avoid using X11 forwarding (it’s slow!)
  – Make sure there is enough memory available (~ trace size x 4)
VampirTrace: Instrumentation Types

• Default: Compiler Instrumentation
  – Easy, instruments all functions
    • problematic for C++ applications
  – Use filters to restrict amount of data generated

• TAU/PDT (-vt:inst tauinst)
  – Rewrites code to insert instrumentation
  – Supports compile-time filters

• Dyninst (-vt:inst dyninst)
  – Instruments compiled (and optimized) binary
  – Does not capture inline-functions

• Manual Instrumentation
  – Manually insert function calls, counter, messages

• Details included as additional information at the end!
Main Timeline

Metric (FLOPS)

Function Stack

Performance Radar

Function and Function Group Summary

Example Trace: CESM