HPC Software – Debugger and Performance Tools

May 2017 | Michael Knobloch
Outline

- Local module setup
- Compilers
- Libraries

Make it work, make it right, make it fast.

*Kent Beck*

Debugger:
- TotalView
- DDT
- STAT
- MUST

Performance Tools:
- Score-P
- Scalasca
- Vampir
- HPCToolkit
- Performance Reports
- TAU
- NVIDIA Visual Profiler
- Darshan
- PAPI
Module setup & compiler
The Module Setup

- Tools are available through “modules”
  - Allows to easily manage different versions of programs
  - Works by dynamic modification of a user's environment

- **JUQUEEN**: Module setup of tools based on UNITE
  - Tools only visible after a module load UNITE
  - User has to take care of dependencies

- **JURECA**: Module setup based on EasyBuild and lmod
  - Staged, hierarchical setup
  - Automatically manages dependencies via toolchains
Most Important Module Commands

module
- spider # lmod only: show all products
- spider product # lmod only: show product details
- avail # show all available products
- list # list loaded products

- load product(s) # setup access to product
- unload product(s) # release access
- swap product1 product2 # replace v1 of product with v2

- whatis product(s) # print short description
- help product(s) # print longer description
- show product(s) # show what “settings” are performed
Compiler

- **JUQUEEN**
  - IBM XL C/C++ and Fortran compiler
  - GNU C/C++ and Fortran compiler
  - Clang C/C++ compiler

- **JURECA**
  - Intel C/C++ and Fortran compiler
  - GNU C/C++ and Fortran compiler
  - PGI C/C++ and Fortran compiler
  - Clang C/C++ compiler
  - NVIDIA CUDA compiler
MPI Libraries

- JUQUEEN
  - IBM MPI
  - MPICH3 → Not officially supported, use only if you have problems with IBM MPI or need MPI-3 features

- JURECA
  - Intel MPI
  - Parastation MPI
  - MVAPICH MPI (CUDA aware)
Debuggers
## Debugging Tools *(status: May 2017)*

<table>
<thead>
<tr>
<th>Tool</th>
<th>JUQUEEN</th>
<th>JURECA</th>
</tr>
</thead>
<tbody>
<tr>
<td>TotalView</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DDT</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>STAT</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>MUST</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Parallel Debugger

- UNIX Symbolic Debugger for C, C++, F77, F90, PGI HPF, assembler programs
- “Standard” debugger
- Special, non-traditional features
  - Multi-process and multi-threaded
  - C++ support (templates, inheritance, inline functions)
  - F90 support (user types, pointers, modules)
  - 1D + 2D Array Data visualization
  - Support for parallel debugging (MPI: automatic attach, message queues, OpenMP, pthreads)
  - Scripting and batch debugging
  - Memory Debugging
  - CUDA and OpenACC support
- http://www.roguewave.com
- **NOTE**: License limited to 2048 processes (shared between all users)
TotalView: Main Window

Stack trace

Break points

Toolbar for common options

Local variables for selected stack frame

Source code window
TotalView: Tools Menu

- Call Graph
- Data visualization
- Message queue graph
DDT Parallel Debugger

- UNIX Graphical Debugger for C, C++, F77, F90 programs
- Modern, easy-to-use debugger
- Special, non-traditional features
  - Multi-process and multi-threaded
  - 1D + 2D array data visualization
  - Support for MPI parallel debugging (automatic attach, message queues)
  - Support for OpenMP (Version 2.x and later)
  - Support for CUDA and OpenACC
  - Job submission from within debugger

- http://www.allinea.com
- **NOTE**: License limited to 64 processes (shared between all users)
DDT: Main Window

- **Process controls**
- **Process groups**
- **Source code**
- **Variables**
- **Expression evaluator**
- **Stack trace**
DDT: Non-standard Features

- **Message queue graph**

- **Multi-Dimensional Array Viewer**

- **Memory Usage**
STAT: Stack Trace Analysis Tool

- Very lightweight helper tool

- Shows merged call tree of whole program
- Useful to detect deadlocks
- Pinpoint individual problems

- Tailored for large-scale systems
  - Scales to millions of processes

- NOT a real/full debugger

- https://computing.llnl.gov/code/STAT/
STAT: Main Window
STAT: Zoom

Which ranks are following

- main
  - 512:[0-511]
  - body()
    - 128:{2,6,10,14,...}  128:{0,4,8,12,16,...}  128:{3,7,11,15,...}  128:{1,5,9,13,17,...}
      - south()
        - 128:{2,6,10,14,...}
          - north()
            - 128:{2,6,10,14,...}
              - __sleep
        - __sleep
      - __sleep
      - __sleep
STAT: Equivalence Classes

Join Equivalence Class
- Collapse
- Collapse Depth
- Hide
- Expand
- Expand All
- Focus
- View Source
STAT: Equivalence Classes (cont.)
MUST

- Next generation MPI correctness and portability checker
  - http://doc.itc.rwth-aachen.de/display/CCP/Project+MUST

- MUST reports
  - Errors: violations of the MPI-standard
  - Warnings: unusual behavior or possible problems
  - Notes: harmless but remarkable behavior
  - Further: potential deadlock detection

- Usage
  - Relink application with mustc, mustcxx, mustf90, …
  - Run application under the control of mustrun (requires one additional MPI process)
  - See MUST_Output.html report
**MUST Datatype Mismatch**

<table>
<thead>
<tr>
<th>Rank</th>
<th>Type</th>
<th>Message</th>
<th>From</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Error</td>
<td>A send and a receive operation use datatypes that do not match! Mismatch occurs at (contiguous) <a href="MPI_INT">0</a> in the send type and at (MPI_BYTE) in the receive type (consult the MUST manual for a detailed description of datatype positions). A graphical representation of this situation is available in a detailed type mismatch view (MUST Output-files/MUST_Typemismatch_0.html). The send operation was started at reference 1, the receive operation was started at reference 2. (Information on communicator: MPI_COMM_WORLD) (Information on send of count 1 with type:Datatype created at reference 3 is for C, commited at reference 4, based on the following type(s): { MPI_INT}Typemap = {(MPI_INT, 0), (MPI_INT, 4)}) (Information on receive of count 8 with type:MPI_BYTE).</td>
<td></td>
<td>reference 1 rank 0: MPI_Sendrecv called from: #0 <a href="mailto:main@example.c">main@example.c</a>:33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>reference 2 rank 1: MPI_Sendrecv called from: #0 <a href="mailto:main@example.c">main@example.c</a>:33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>reference 3 rank 0: MPI_Type_contiguous called from: #0 <a href="mailto:main@example.c">main@example.c</a>:29</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>reference 4 rank 0: MPI_Type_commit called from: #0 <a href="mailto:main@example.c">main@example.c</a>:30</td>
</tr>
</tbody>
</table>

**Message**

The application issued a set of MPI calls that mismatch in type signatures! The graph below shows details on this situation. The first differing item of each involved communication request is highlighted.

**Datatype Graph**

```
MPI_Sendrecv: send

MPI_Type_contiguous(count=2)  MPI_Sendrecv:recv

[0]

MPI_INT  MPI_BYTE
```
MUST Deadlock Detection

The application issued a set of MPI calls that can cause a deadlock! The graphs below show details on this situation. This includes a wait-for graph that shows active wait-for dependencies between the processes that cause the deadlock. Note that this process set only includes processes that cause the deadlock and no further processes. A legend details the wait-for graph components in addition, while a parallel call stack view summarizes the locations of the MPI calls that cause the deadlock. Below these graphs, a message queue graph shows active and unmatched point-to-point communications. This graph only includes operations that could have been intended to match a point-to-point operation that is relevant to the deadlock situation. Finally, a parallel call stack shows the locations of any operation in the parallel call stack. The leafs of this call stack graph show the components of the message queue graph that they span. The application still runs, if the deadlock manifested (e.g. caused a hang on this MPI implementation) you can attach to the involved ranks with a debugger or abort the application (if necessary).

**Active Communicators**

<table>
<thead>
<tr>
<th>Comm:</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_COMM_WORLD</td>
<td></td>
</tr>
</tbody>
</table>

**Wait-for Graph**

- MPI_Send@0
  - comm=A, tag=456
- MPI_Send@1
  - comm=A, tag=456
- MPI_Send@2
  - comm=A, tag=456
- MPI_Send@3
  - comm=A, tag=456

**Legend**

- Active MPI Call
- Sub Operation

**Call Stack**

- main@example.c:39
  - Ranks: 0-3
  - MPI_Send

**Active and Relevant Point-to-Point Messages: Overview**

**Active and Relevant Point-to-Point Messages: Callstack-view**
Performance Analysis Tools
Typical Performance Analysis Procedure

- Do I have a performance problem at all?
  - Time / speedup / scalability measurements
- What is the key bottleneck (computation / communication)?
  - MPI / OpenMP / flat profiling
- Where is the key bottleneck?
  - Call-path profiling, detailed basic block profiling
- Why is it there?
  - Hardware counter analysis
  - Trace selected parts (to keep trace size manageable)
- Does the code have scalability problems?
  - Load imbalance analysis, compare profiles at various sizes function-by-function, performance modeling
Remark: No Single Solution is Sufficient!

A combination of different methods, tools and techniques is typically needed!

- Analysis
  - Statistics, visualization, automatic analysis, data mining, ...
- Measurement
  - Sampling / instrumentation, profiling / tracing, ...
- Instrumentation
  - Source code / binary, manual / automatic, ...
Critical Issues

- Accuracy
  - Intrusion overhead
    - Measurement itself needs time and thus lowers performance
  - Perturbation
    - Measurement alters program behavior, e.g., memory access pattern
    - Might prevent compiler optimization, e.g. function inlining
  - Accuracy of timers & counters

- Granularity
  - How many measurements?
  - How much information / processing during each measurement?

© Tradeoff: Accuracy vs. Expressiveness of data
### Performance Tools (status: May 2017)

<table>
<thead>
<tr>
<th>Tool</th>
<th>JUQUEEN</th>
<th>JURECA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score-P</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Scalasca2</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Vampir[Server]</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>HPCToolkit</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Allinea Performance Reports</td>
<td>✗️</td>
<td>✔️</td>
</tr>
<tr>
<td>Darshan</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>NVIDIA Visual Profiler</td>
<td>✗️</td>
<td>✔️</td>
</tr>
<tr>
<td>TAU</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Intel VTune Amplifier XE*</td>
<td>✗️</td>
<td>✔️</td>
</tr>
<tr>
<td>mpiP*</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Extrae/Paraver*</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>PAPI*</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>
Score-P

- Community instrumentation and measurement infrastructure
  - Developed by a consortium of performance tool groups
  - Next generation measurement system of
    - Scalasca 2.x
    - Vampir
    - TAU
    - Periscope
  - Common data formats improve tool interoperability
  - http://www.score-p.org
Collection of trace-based performance analysis tools

- Specifically designed for large-scale systems
- Unique features:
  - Scalable, automated search for event patterns representing inefficient behavior
  - Scalable identification of the critical execution path
  - Delay / root-cause analysis
- Based on Score-P for instrumentation and measurement
  - Includes convenience / post-processing commands providing added value
- [http://www.scalasca.org](http://www.scalasca.org)
What is the Key Bottleneck?

- Generate **flat MPI profile** using Score-P/Scalasca
  - Only requires re-linking
  - Low runtime overhead

- Provides detailed information on MPI usage
  - How much time is spent in which operation?
  - How often is each operation called?
  - How much data was transferred?

- Limitations:
  - Computation on non-master threads and outside of MPI_Init/MPI_Finalize scope ignored
Flat MPI Profile: Recipe

1. Prefix your *link command* with “scorep --nocompiler”

2. Prefix your MPI *launch command* with “scalasca -analyze”

3. After execution, examine analysis results using “scalasca -examine scorep_<title>”
Flat MPI Profile: Example

```
% module load UNITE scorep scalasca
% mpixlf90 -O3 -qsmp=omp -c foo.f90
% mpixlf90 -O3 -qsmp=omp -c bar.f90
% scorep --nocompiler \
    mpixlf90 -O3 -qsmp=omp -o myprog foo.o bar.o

##########################
## In the job script:  ##
##########################
module load UNITE scalasca
scalasca -analyze \
    runjob --ranks-per-node P --np n [...] --exe ./myprog

##########################
## After job finished:  ##
##########################
% scalasca -examine scorep_myprog_Pp```
Flat MPI Profile: Example (cont.)

Aggregate execution time on master threads

Time spent in a particular MPI call

Time spent in selected call as percentage of total time
Where is the Key Bottleneck?

- Generate **call-path profile** using Score-P/Scalasca
  - Requires re-compilation
  - Runtime overhead depends on application characteristics
  - Typically needs some care setting up a good measurement configuration
    - Filtering
    - Selective instrumentation

- Option 1 (recommended):
  Automatic compiler-based instrumentation

- Option 2:
  Manual instrumentation of interesting phases, routines, loops
Call-path Profile: Recipe

1. Prefix your *compile & link commands* with “scorep”
2. Prefix your MPI *launch command* with “scalasca -analyze”
3. After execution, compare overall runtime with uninstrumented run to determine overhead
4. If overhead is too high
   1. Score measurement using “scalasca -examine -s scorep_<title>”
   2. Prepare filter file
   3. Re-run measurement with filter applied using prefix “scalasca -analyze -f <filter_file>”
5. After execution, examine analysis results using “scalasca -examine scorep_<title>”
Call-path Profile: Example

% module load UNITE scorep scalasca
% scorep mpirun -np 10 -qsmp=omp -c foo.f90
% scorep mpirun -np 10 -qsmp=omp -c bar.f90
% scorep \
    mpirun -np 10 -qsmp=omp -o myprog foo.o bar.o

#################################################################
##  In the job script:  ##
#################################################################

module load UNITE scalasca
scalasca -analyze \
    runjob --ranks-per-node P --np n [...] --exe ./myprog
Call-path Profile: Example (cont.)

- Estimates trace buffer requirements
- Allows to identify candidate functions for filtering
  - Computational routines with high visit count and low time-per-visit ratio
- Region/call-path classification
  - MPI (pure MPI library functions)
  - OMP (pure OpenMP functions/regions)
  - USR (user-level source local computation)
  - COM (“combined” USR + OpeMP/MPI)
  - ANY/ALL (aggregate of all region types)

```bash
% scalasca -examine -s epik_myprog_Ppnxt_sum
scorep-score -r ./epik_myprog_Ppnxt_sum/profile.cubex
INFO: Score report written to ./scorep_myprog_Ppnxt_sum/scorep.score
```
Call-path Profile: Example (cont.)

% less scorep_myprog_Ppmt_sum/scorep.score

Estimated aggregate size of event trace: 162GB
Estimated requirements for largest trace buffer (max_buf): 2758MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY): 2822MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=2822MB 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>2,891,417,902</td>
<td>6,662,521,083</td>
<td>36581.51</td>
<td>100.0</td>
<td>5.49</td>
<td>ALL</td>
<td></td>
</tr>
<tr>
<td>USR</td>
<td>2,858,189,854</td>
<td>6,574,882,113</td>
<td>13618.14</td>
<td>37.2</td>
<td>2.07</td>
<td>USR</td>
<td></td>
</tr>
<tr>
<td>OMP</td>
<td>54,327,600</td>
<td>86,353,920</td>
<td>22719.78</td>
<td>62.1</td>
<td>263.10</td>
<td>OMP</td>
<td></td>
</tr>
<tr>
<td>MPI</td>
<td>676,342</td>
<td>550,010</td>
<td>208.98</td>
<td>0.6</td>
<td>379.96</td>
<td>MPI</td>
<td></td>
</tr>
<tr>
<td>COM</td>
<td>371,930</td>
<td>735,040</td>
<td>34.61</td>
<td>0.1</td>
<td>47.09</td>
<td>COM</td>
<td></td>
</tr>
<tr>
<td>USR</td>
<td>921,918,660</td>
<td>2,110,313,472</td>
<td>3290.11</td>
<td>9.0</td>
<td>1.56</td>
<td>matmul_sub</td>
<td></td>
</tr>
<tr>
<td>USR</td>
<td>921,918,660</td>
<td>2,110,313,472</td>
<td>5914.98</td>
<td>16.2</td>
<td>2.80</td>
<td>binvcrhs</td>
<td></td>
</tr>
<tr>
<td>USR</td>
<td>921,918,660</td>
<td>2,110,313,472</td>
<td>3822.64</td>
<td>10.4</td>
<td>1.81</td>
<td>matvec_sub</td>
<td></td>
</tr>
<tr>
<td>USR</td>
<td>41,071,134</td>
<td>87,475,200</td>
<td>358.56</td>
<td>1.0</td>
<td>4.10</td>
<td>lhsinit</td>
<td></td>
</tr>
<tr>
<td>USR</td>
<td>41,071,134</td>
<td>87,475,200</td>
<td>145.42</td>
<td>0.4</td>
<td>1.66</td>
<td>binvrhs</td>
<td></td>
</tr>
<tr>
<td>USR</td>
<td>29,194,256</td>
<td>68,892,672</td>
<td>86.15</td>
<td>0.2</td>
<td>1.25</td>
<td>exact_solution</td>
<td></td>
</tr>
<tr>
<td>OMP</td>
<td>3,280,320</td>
<td>3,293,184</td>
<td>15.81</td>
<td>0.0</td>
<td>4.80</td>
<td>!$omp parallel [++]</td>
<td></td>
</tr>
</tbody>
</table>
Call-path Profile: Filtering

- In this example, the 6 most frequently called routines are of type USR.
- These routines contribute around 35% of total time.
  - However, much of that is most likely measurement overhead:
    - Frequently executed.
    - Time-per-visit ratio in the order of a few microseconds.

- Avoid measurements to reduce the overhead.
- List routines to be filtered in simple text file.
Filtering: Example

- Score-P filtering files support
  - Wildcards (shell globs)
  - Blacklisting
  - Whitelisting
  - Filtering based on filenames

% cat filter.txt
SCOREP_REGION_NAMES_BEGIN
  EXCLUDE
    binvcrhs
    matmul_sub
    matvec_sub
    binvrhs
    lhsinit
    exact_solution
SCOREP_REGION_NAMES_END
## To verify effect of filter:

```bash
% scalasca -examine -s -f filter.txt \n  scorep_myprog_Ppnxt_sum
```

## In the job script:

```bash
module load UNITE scalasca
scalasca -analyze -f filter.txt \n  runjob --ranks-per-node P --np n [...] --exe ./myprog
```

## After job finished:

```bash
% scalasca -examine scorep_myprog_Ppnxt_sum
```
Call-path Profile: Example (cont.)
Call-path Profile: Example (cont.)

Box plot view shows distribution across processes/threads.

Distribution of selected metric across call tree.

When expanding, value changes from inclusive to exclusive.

Selection updates columns to the right.
Split base metrics into more specific metrics.
Score-P: Advanced Features

- Sampling support
  - x86 only
- Measurement can be extensively configured via environment variables
  - Check output of “scorep-info config-vars” for details
- Allows for targeted measurements:
  - Selective recording
  - Phase profiling
  - Parameter-based profiling
  - …
- Please ask us or see the user manual for details
Measurement of CUDA events

- Record CUDA events using the CUPTI interface

  ```
  % export SCOREP_CUDA_ENABLE=gpu,kernel,idle
  ```

- Important record types:
  - runtime  CUDA runtime API
  - driver    CUDA driver API
  - gpu       GPU activities
  - kernel    CUDA kernels
  - Idle      GPU compute idle time
  - memcpy    CUDA memory copy

- For all record types consult the Score-P user guide
Why is the Bottleneck There?

- This is highly application dependent!
- Might require additional measurements
  - Hardware-counter analysis
    - CPU utilization
    - Cache behavior
  - Selective instrumentation
  - Manual/automatic event trace analysis
HW Counter Measurements w/ Score-P

- Score-P supports both PAPI preset and native counters
- Available counters: papi_avail or papi_native_avail

```plaintext
% module load UNITE papi/5.0.1
% less $PAPI_ROOT/doc/papi-5.0.1-avail.txt
% less $PAPI_ROOT/doc/papi-5.0.1-native_avail.txt
% less $PAPI_ROOT/doc/papi-5.0.1-avail-detail.txt
```

- Specify using “SCOREP_METRIC_PAPI” environment variable

```plaintext
############################################################################
## In the job script:  ##
############################################################################

module load UNITE scalasca
export SCOREP_METRIC_PAPI="PAPI_FP_OPS,PAPI_TOT_CYC"
scalasca -analyze -f filter.txt \
runjob --ranks-per-node P --np n [...] --exe ./myprog
```
Automatic Trace Analysis w/ Scalasca

- Idea: Automatic search for patterns of inefficient behavior
  - Identification of wait states and their root causes
  - Classification of behavior & quantification of significance
  - Scalable identification of the critical execution path

- Advantages
  - Guaranteed to cover the entire event trace
  - Quicker than manual/visual trace analysis
  - Helps to identify hot-spots for in-depth manual analysis
Trace Generation & Analysis w/ Scalasca

- Enable trace collection & analysis using “-t” option of “scalasca -analyze”:

```bash
module load UNITE scalasca
export SCOREP_TOTAL_MEMORY=120MB # Consult score report
scalasca -analyze -f filter.txt -t \
    runjob --ranks-per-node P --np n [...] --exe ./myprog
```

- ATTENTION:
  - Traces can quickly become extremely large!
  - Remember to use proper filtering, selective instrumentation, and Score-P memory specification
  - Before flooding the file system, ask us for assistance!
Scalasca Trace Analysis Example

Additional wait-state metrics from the trace analysis

Delay / root-cause metrics

Critical-path profile
Vampir Event Trace Visualizer

- **Offline** trace visualization for Score-P’s OTF2 trace files
- **Visualization of MPI, OpenMP** and application events:
  - **All** diagrams highly customizable (through context menus)
  - Large variety of displays for **ANY** part of the trace
- http://www.vampir.eu

- Advantage:
  - Detailed view of dynamic application behavior
- Disadvantage:
  - Requires event traces (huge amount of data)
  - Completely manual analysis
Vampir Displays
Vampir: Timeline Diagram

- Functions organized into groups
- Coloring by group
- Message lines can be colored by tag or size
- Information about states, messages, collective and I/O operations available through clicking on the representation
Vampir: Process and Counter Timelines

- Process timeline show call stack nesting
- Counter timelines for hardware and software counters
Vampir: Execution Statistics

- Aggregated profiling information: execution time, number of calls, inclusive/exclusive

- Available for all / any group (activity) or all routines (symbols)

- Available for any part of the trace \(\Rightarrow\) selectable through time line diagram
Vampir: Process Summary

- Execution statistics over all processes for comparison

- Clustering mode available for large process counts
Vampir: Communication Statistics

- Byte and message count, min/max/avg message length and min/max/avg bandwidth for each process pair
- Message length statistics

Available for any part of the trace
Vampir: CUDA Example

- Detailed information on kernel execution and memory transfers
- All statistics and displays also available for CUDA events
- Multi-platform sampling-based call-path profiler
- Works on unmodified, optimized executables
- http://hpctoolkit.org

Advantages:
- Overhead can be easily controlled via sampling interval
- Advantageous for complex C++ codes with many small functions
- Loop-level analysis (sometimes even individual source lines)
- Supports POSIX threads

Disadvantages:
- Statistical approach that might miss details
- MPI/OpenMP time displayed as low-level system calls
HPCToolkit: Metric Specification

- Specified via environment variable HPCRUN_EVENT_LIST

- General format:
  “name@interval [;name@interval ...]”

- Possible sample sources:
  - WALLCLOCK
  - PAPI counters
  - IO (use w/o interval spec)
  - MEMLEAK (use w/o interval spec)

- Interval: given in microseconds
  - E.g., 10000 → 100 samples per second
Example: hpcviewer

Callpath to hotspot

associated source code
Allinea Performance Reports

- **Single page** report provides quick overview of performance issues
- Works on unmodified, optimized executables
- Shows CPU, memory, network and I/O utilization
- Supports MPI, multi-threading and accelerators
- Saves data in HTML, CVS or text form

- **Note**: License limited to 512 processes (with unlimited number of threads)
Example Performance Reports

Summary: cp2k.popt is **CPU-bound** in this configuration

The total wallclock time was spent as follows:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td>56.5%</td>
</tr>
<tr>
<td><strong>MPI</strong></td>
<td>43.5%</td>
</tr>
<tr>
<td><strong>I/O</strong></td>
<td>0.0%</td>
</tr>
</tbody>
</table>

Time spent running application code. High values are usually good. This is **average**, check the CPU performance section for optimization advice.

Time spent in MPI calls. High values are usually bad. This is **average**, check the MPI breakdown for advice on reducing it.

Time spent in filesystem I/O. High values are usually bad. This is **negligible**, there's no need to investigate I/O performance.

This application run was **CPU-bound**. A breakdown of this time and advice for investigating further is in the CPU section below.

### CPU

A breakdown of how the 56.5% total CPU time was spent:

<table>
<thead>
<tr>
<th>Type</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar numeric ops</td>
<td>27.7%</td>
</tr>
<tr>
<td>Vector numeric ops</td>
<td>11.3%</td>
</tr>
<tr>
<td>Memory accesses</td>
<td>60.9%</td>
</tr>
<tr>
<td>Other</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

The per-core performance is **memory-bound**. Use a profiler to identify time-consuming loops and check their cache performance. Little time is spent in **vectorized instructions**. Check the compiler's vectorization advice to see why key loops could not be vectorized.

### MPI

Of the 43.5% total time spent in MPI calls:

<table>
<thead>
<tr>
<th>Type</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time in collective calls</td>
<td>8.2%</td>
</tr>
<tr>
<td>Time in point-to-point calls</td>
<td>91.8%</td>
</tr>
<tr>
<td>Estimated collective rate</td>
<td>169 Mb/s</td>
</tr>
<tr>
<td>Estimated point-to-point rate</td>
<td>50.6 Mb/s</td>
</tr>
</tbody>
</table>

The point-to-point transfer rate is low. This can be caused by inefficient message sizes, such as many small messages, or by imbalanced workloads causing processes to wait. Use an MPI profiler to identify the problematic calls and ranks.

### I/O

A breakdown of how the 0.0% total I/O time was spent:

<table>
<thead>
<tr>
<th>Type</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time in reads</td>
<td>0.0%</td>
</tr>
<tr>
<td>Time in writes</td>
<td>0.0%</td>
</tr>
<tr>
<td>Estimated read rate</td>
<td>0 bytes/s</td>
</tr>
<tr>
<td>Estimated write rate</td>
<td>0 bytes/s</td>
</tr>
</tbody>
</table>

No time is spent in I/O operations. There's nothing to optimize here!

### Memory

Per-process memory usage may also affect scaling:

<table>
<thead>
<tr>
<th>Type</th>
<th>Mb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean process memory usage</td>
<td>82.5</td>
</tr>
<tr>
<td>Peak process memory usage</td>
<td>89.3</td>
</tr>
<tr>
<td>Peak node memory usage</td>
<td>7.4%</td>
</tr>
</tbody>
</table>

The peak node memory usage is low. You may be able to reduce the total number of CPU hours used by running with fewer MPI processes and more data on each process.
Allinea Performance Reports: Example

```bash
% module load AllineaPerformanceReports

# In the job script:  
# Source the allinea module:

# Source the allinea module:

perf-report --mpi="slurm" \
  srun --procs-per-node=P --nodes=n [...] ./myprog [args]

# After job finished:  
% less myprog_<NP>_p_<DATE>.txt
% firefox myprog_<NP>_p_<DATE>.html
```
Darshan

- I/O characterization tool logging parallel application file access
- Summary report provides quick overview of performance issues
- Works on unmodified, optimized executables
- Shows counts of file access operations, times for key operations, histograms of accesses, etc.

- Supports POSIX, MPI-IO, HDF5, PnetCDF, …
  - Doesn’t support mpif90 on BlueGene systems (use mpif77)
  - Binary log file written at exit post-processed into PDF report

- Open Source: installed on many HPC systems
Example Darshan report extract

Job ID: | UID: | Nprocs: 4096 | Runtime: 175 seconds

Average I/O cost per process:
- POSIX
- MPI-I/O
  - Read
  - Write
  - Metadata
  - Other (including application compute)

I/O Operation Counts:
- Read
- Write
- Open
- Stat
- Seek
- Mmap
- Fsync

I/O Sizes:
- Count (Total, All Processes)

I/O Pattern:
- Total
- Sequential
- Consecutive
% load compiler and MPI module
% module load darshan-runtime darshan-util

##############################################
## In the job script:  ##
##############################################

export LD_PRELOAD=$EBROOTDARSHANMINRUNTIME/lib/libdarshan.so
export DARSHAN_LOG_PATH=$PWD
export DARSHAN_LOGFILE=darshan.log
srun --tasks-per-node $P --ntasks $n [...] ./myprog [args]

##############################################
## After job finished:  ##
##############################################

% darshan-job-summary.pl darshan.log
% gv darshan.pdf
% module load darshan
# (re)link executable using darshan compiler wrapper
# (if necessary, substitute mpif77 for Fortran90 compiler)
% mpixlc -O3 -qsmp=omp -o myprog foo.c
% mpixlf77 -O3 -qsmp=omp -o myprog foo.f90

##########################
## In the job script: ##
##########################
runjob --envs DARSHAN_LOG_PATH=$PWD \
      --envs DARSHAN_LOGFILE=darshan.log \
      --ranks-per-node P --np n [...] --exe ./myprog

##########################
## After job finished: ##
##########################
% darshan-job-summary.pl darshan.log
% gv darshan.pdf
NVIDIA Visual Profiler

- Part of the CUDA Toolkit
- Supports all CUDA enabled GPUs
- Supports CUDA and OpenACC on Windows, OS X and Linux

- Unified CPU and GPU Timeline
- CUDA API trace
  - Memory transfers, kernel launches, and other API functions
- Automated performance analysis
  - Identify performance bottlenecks and get optimization suggestions
- Guided Application Analysis
- Power, thermal, and clock profiling
NVIDIA Visual Profiler: Example

Timeline view

Detailed information on Kernel execution

Automatic analysis of performance bottlenecks
TAU

- Very portable tool set for instrumentation, measurement and analysis of parallel multi-threaded applications
  - http://tau.uoregon.edu/

- Supports
  - Various profiling modes and tracing
  - Various forms of code instrumentation
  - C, C++, Fortran, Java, Python
  - MPI, multi-threading (OpenMP, Pthreads, …)
  - Accelerators
TAU: Instrumentation

- Flexible instrumentation mechanisms at multiple levels
  - Source code
    - manual
    - automatic
      - C, C++, F77/90/95 (Program Database Toolkit (PDT))
      - OpenMP (directive rewriting with Opari)
  - Object code
    - pre-instrumented libraries (e.g., MPI using PMPI)
    - statically-linked and dynamically-loaded (e.g., Python)
  - Executable code
    - dynamic instrumentation (pre-execution) (DynInst)
    - virtual machine instrumentation (e.g., Java using JVMPi)
- Support for performance mapping
- Support for object-oriented and generic programming
TAU: Basic Profile View

![TAU Profile View](Image)

**Counter Name:** P_WALL_CLOCK_TIME (seconds)

- **MPI_Allreduce()**
- **algps::HyperbolicLevelIntegrator3::advance_bdry_fill_create**
- **algps::HyperbolicLevelIntegrator3::advanceLevel()**
- **algps::HyperbolicLevelIntegrator3::fill_new_level_create**
- **Mesh::GriddingAlgorithm3::load_balance_boxes**
- **algps::HyperbolicLevelIntegrator3::advance_bdry_fill_comm**
- **Mesh::GriddingAlgorithm3::findRefinementBoxes()**
- **algps::HyperbolicLevelIntegrator3::coarsen_fluxsum_create**
- **algps::HyperbolicLevelIntegrator3::coarsen_sync_create**
- **Mesh::GriddingAlgorithm3::find_boxes_containing_tags**
- **MPI_Init()**
- **Mesh::GriddingAlgorithm3::bdry_fill_tags_create**
- **MPI_Bcast()**
- **MPI_Wait()**
- **algps::HyperbolicLevelIntegrator3::error_bdry_fill_comm**
- **MPI_Finalize()**
- **MPI_Isend()**
- **MPI_Waitall()**
- **Mesh::GriddingAlgorithm3::remove_intersections_reggrid_all**
- **MPI_Test()**
- **algps::HyperbolicLevelIntegrator3::fill_new_level_comm**
- **MPI_Comm_rank()**
Box width and color indicate different metrics.
TAU: 3D Profile View

Height and color indicate different metrics
Documentation

- To check latest status and versions
  - **JUQUEEN**: use "module avail"
  - **JURECA**: use "module spider"

- Websites
  - [http://www.fz-juelich.de/iasjsc/juqueen/](http://www.fz-juelich.de/iasjsc/juqueen/)
  - [http://www.fz-juelich.de/iasjsc/jureca/](http://www.fz-juelich.de/iasjsc/jureca/)
  - User Info
    - Parallel Debugging (⚠️)
    - Parallel Performance Analysis (⚠️)
  - [http://www.vi-hps.org/training/material/](http://www.vi-hps.org/training/material/)
  - Performance Tools LiveDVD image
  - Links to tool websites and documentation
  - Tutorial slides
Support

- For general support: sc@fz-juelich.de
- Tool-specific support via corresponding mailing lists
  - Score-P: support@score-p.org
  - Scalasca: scalasca@fz-juelich.de

- Workshops and Trainings:
  - Regular VI-HPS Tuning Workshops
    - Several days
    - Multiple tools, e.g. Score-P, Scalasca, Vampir, TAU, …
    - Bring-your-own-code
    - http://www.vi-hps.org/training/tws/
  - JUQUEEN Porting and Tuning Workshop Series
  - WS’s and trainings at several HPC conferences
POP CoE

- **A Center of Excellence**
  - On Performance Optimization and Productivity
  - Promoting best practices in parallel programming
- **Providing Services**
  - Precise understanding of application and system behaviour
  - Suggestions/support on how to refactor code in the most productive way
- **Horizontal**
  - Transversal across application areas, platforms and scales
  - For (your?) academic AND industrial codes and users

The best: Currently free of charge
Motivation

- **Why?**
  - Complexity of machines and codes
    → Frequent lack of quantified understanding of actual behaviour
    → Not clear most productive direction of code refactoring
  - Important to **maximize efficiency** (performance, power) of compute intensive applications and productivity of the development efforts

- **What?**
  - Parallel programs, mainly MPI/OpenMP
    - Also CUDA, OpenCL, OpenACC, Python, …
The process ...

When?
October 2015 – March 2018

How?
- Apply
  - Fill in small questionnaire describing application and needs [https://pop-coe.eu/request-service-form](https://pop-coe.eu/request-service-form)
  - Questions? Ask pop@bsc.es
- Selection/assignment process
- Install tools @ your production machine (local, PRACE, …)
- Interactively: Gather data → Analysis → Report
Services provided by the CoE

? Parallel Application Performance Audit ⇒ Report
- Primary service
- Identify performance issues of customer code (at customer site)
- Small effort (< 1 month)

! Parallel Application Performance Plan ⇒ Report
- Follow-up on the audit service
- Identifies the root causes of the issues found and qualifies and quantifies approaches to address them
- Longer effort (1-3 months)

✔ Proof-of-Concept ⇒ Software Demonstrator
- Experiments and mock-up tests for customer codes
- Kernel extraction, parallelisation, mini-apps experiments to show effect of proposed optimisations
- 6 months effort