HPC Software – Debugger and Performance Tools

November 2015  |  Michael Knobloch
Outline

- Local module setup
- Compilers*
- Libraries*

Debugger:
- TotalView
- DDT
- STAT
- MUST

Performance Tools:
- Score-P
- Scalasca
- Vampir
- HPCToolkit
- Performance Reports
- TAU
- mpiP
- Extrae/Paraver
- PAPI

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

*Kent Beck*
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 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
  ▪ spider product
  ▪ avail
  ▪ list

  ▪ 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 and MPI libraries

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

- **JURECA**
  - Intel C/C++ and Fortran compiler
  - GNU C/C++ and Fortran compiler
  - Intel MPI
  - Parastation MPI
Debuggers
Debugging Tools (status: November 2015)

- Debugging
  - TotalView debugger
  - DDT debugger
  - STAT
  - MUST MPI verification tool

<table>
<thead>
<tr>
<th></th>
<th>JUQUEEN</th>
<th>JURECA</th>
</tr>
</thead>
<tbody>
<tr>
<td>TotalView debugger</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DDT debugger</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>STAT</td>
<td>✓</td>
<td>(✗)</td>
</tr>
<tr>
<td>MUST MPI verification tool</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
- http://www.roguewave.com
- NOTE: License limited to 2048 processes (shared between all users)
TotalView: Main Window

- Toolbar for common options
- Local variables for selected stack frame
- Source code window
- Stack trace
- Breakpoints
TotalView: Tools Menu

- Call Graph
- Message queue graph
- Data visualization
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)
  - Job submission from within debugger

- http://www.allinea.com

- **NOTE:** License limited to 32 processes (shared between all users)
DDT: Main Window

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

- Multi-Dimensional Array Viewer
- Message queue graph
- 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: Equivalence Classes
STAT: Equivalence Classes (cont.)

```
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() => north() => _sleep
north() => _sleep
west() => east() => south() => north() => _sleep
east() => south() => north() => _sleep
```
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

A send and a receive operation use datatypes that do not match! Mismatch occurs at (contiguous) [0](MPI_INT) 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, committed 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)

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)
- [0]
- MPI_INT
- MPI_BYTE
- MPI_Sendrecv:recv
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**

Comm: A

MPI_COMM_WORLD

**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

**Legend**

- **Active MPI Call**
  - Sub Operation
  - A waits for B and C
  - B
  - C

**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

<table>
<thead>
<tr>
<th>Tool</th>
<th>JUQUEEN</th>
<th>JURECA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score-P measurement system</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Scalasca2 performance analyzer</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Vampir[Server] trace visualizer</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>HPCToolkit sampling profiler</td>
<td>✓</td>
<td>(✗)</td>
</tr>
<tr>
<td>Allinea Performance Reports</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>TAU performance system</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>mpiP MPI profiling library*</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Extrae/Paraver tracing tool*</td>
<td>✓</td>
<td>(✓)</td>
</tr>
<tr>
<td>PAPI hardware counter library*</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
What is the Key Bottleneck?

- Generate **flat MPI profile** using Score-P/Scalasca (or mpiP)
  - 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_Ppnx_t_sum
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 mpxlf90 -O3 -qsmp=omp -c foo.f90
% scorep mpxlf90 -O3 -qsmp=omp -c bar.f90
% scorep \
     mpxlf90 -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
```
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 + OpenMP/MPI)
  - ANY/ALL (aggregate of all region types)
Call-path Profile: Example (cont.)

% less scorep_myprog_Ppmxt_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/visits[us]</th>
<th>region</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td></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>
</tr>
<tr>
<td>USR</td>
<td></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>
</tr>
<tr>
<td>OMP</td>
<td></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>
</tr>
<tr>
<td>MPI</td>
<td></td>
<td>676,342</td>
<td>550,010</td>
<td>208.98</td>
<td>0.6</td>
<td>379.96</td>
<td>MPI</td>
</tr>
<tr>
<td>COM</td>
<td></td>
<td>371,930</td>
<td>735,040</td>
<td>34.61</td>
<td>0.1</td>
<td>47.09</td>
<td>COM</td>
</tr>
</tbody>
</table>

USR 921,918,660 2,110,313,472 3290.11 9.0 1.56 matmul_sub
USR 921,918,660 2,110,313,472 5914.98 16.2 2.80 binvcrhs
USR 921,918,660 2,110,313,472 3822.64 10.4 1.81 matvec_sub
USR 41,071,134  87,475,200  358.56  1.0  4.10 lhsinit
USR 41,071,134  87,475,200  145.42  0.4  1.66 binvrhs
USR 29,194,256  68,892,672  86.15  0.2  1.25 exact_solution
OMP 3,280,320  3,293,184  15.81  0.0  4.80 !$omp parallel

[...]
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

```plaintext
% 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:

```
% scalasca -examine -s -f filter.txt \
  scorep_myprog_Ppnpnt_sum
```

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

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

```
- 1.44e4 Time (sec)
  - 8.77e7 Visits (occ)
  - 0.00 Synchronizations (occ)
  - 0.00 Communications (occ)
  - 9.57e9 Bytes transferred (bytes)
  - 0.00 MPI file operations (occ)
  - 2696.16 Computational imbalance (sec)
  - 0.00 Minimum Inclusive Time (sec)
  - 6.99 Maximum Inclusive Time (sec)
  - 64.00 task_migration_loss
  - 0.00 task_migration_win

0.00  1.44e4 (100.00%)  1.44e4
0.00  1.44e4 (100.00%)  1.44e4
0.00  1.44e4 (100.00%)  1.44e4
```
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
Call-path Profile: Example (cont.)

Split base metrics into more specific metrics
Score-P: Advanced Features

- 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
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`

```bash
% 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

```bash
##########################
## 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”:

```
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](http://www.vampir.eu)

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

Associated source code

Callpath to hotspot
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:

<table>
<thead>
<tr>
<th>Component</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>56.5%</td>
</tr>
<tr>
<td>MPI</td>
<td>43.5%</td>
</tr>
<tr>
<td>I/O</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

- **CPU**
  - A breakdown of how the 56.5% total CPU time was spent:
    - Scalar numeric ops 27.7%
    - Vector numeric ops 11.3%
    - Memory accesses 60.9%
    - Other 0.0%
  - 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:
    - Time in collective calls 8.2%
    - Time in point-to-point calls 91.8%
    - Estimated collective rate 169 Mb/s
    - Estimated point-to-point rate 50.6 Mb/s
  - 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:
    - Time in reads 0.0%
    - Time in writes 0.0%
    - Estimated read rate 0 bytes/s
    - Estimated write rate 0 bytes/s
  - No time is spent in I/O operations. There’s nothing to optimize here!

- **Memory**
  - Per-process memory usage may also affect scaling:
    - Mean process memory usage 82.5 Mb
    - Peak process memory usage 89.3 Mb
    - Peak node memory usage 7.4%
  - 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.
**TAU**

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

File Options Windows Help

M. Knobloch
SC Introduction, November 2015
TAU: Callgraph Profile View

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/jureca/
    - User Info
      - Parallel Debugging
      - Parallel Performance Analysis
  - 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
Appendix

- Tool recipes for JUQUEEN
STAT: Recipe

- Compile and link your program with debug option: `-g`
- Load modules

```bash
% ssh -X user@juqueen
[...]
juqueen% module load UNITE stat
UNITE loaded
stat/2.1 loaded
juqueen% STATGUI
```

- Submit job and attach to it from GUI

**CAVEAT:**
- Job needs to be started by login node where GUI is running
- Add the following entry to the submission script
  ```bash
  #@ requirements = (Machine == "juqueen<n>")
  with <n> being the login node id
  ```
STAT: Attaching to a Job

```
17375 runjob --exe /home/jzam11/jzam1143/scalasca/juqueen/topompi/cart --
```

TotalView: Recipe for JUQUEEN

- Compile and link your program with debug option: -g
- Use absolute paths for source code info: -qfullpath
- In case of optimized codes (XL), keep function call parameters: –qkeepparm
- Load modules

```plaintext
% ssh -X user@juqueen
[...]
juqueen% module load UNITE totalview
UNITE loaded
totalview/8.14.0-16-mrnet loaded

juqueen% mpixlcxx hello.cpp -qfullpath -qkeepparm -g -o helloworld
juqueen%
```
TotalView: Interactive Startup

- Interactively: call the \texttt{lltv} script
  - Creates a LoadLeveler batch script with required TotalView parameters
- If user cancels the script, it cancels the debugging job (does not eat your computing quota)

- \textbf{NOTE:}
  - License limited to 2048 MPI ranks (shared between all users)
  - Attaching to subset is recommended
TotalView: *lltv* Launch Script

% *lltv* -n <nodes> : -default_parallel_attach_subset=

<rank-range> runjob -a --exe <program> -p <num>

- Starts <program> with <nodes> and <num> processes per node, attaches to <rank-range>:
  - Rank: that rank only
  - RankX-RankZ: all ranks, both inclusive
  - RankX-RankZ: stride every *stride*th between RankX and RankZ
- Example:

  % *lltv* -n 2 : -default_parallel_attach_subset=
  
  2-6 runjob -a --exe helloworld -p 64

Creating LoadLeveler Job

Submitting LoadLeveler Interactive Job for Totalview

Wait for job juqueen1c1.32768.0 to be started:.............
TotalView: Execution Recipe

- TotalView tries to debug “runjob” and shows no source code
  - Ignore it and press “GO”

- After some seconds, TotalView will detect parallel execution and ask if it should stop. Yes, it should stop.
- To find the correct point file/function to debug, use the “File-Open” command.
- Set your breakpoints, and press “GO” again. Debugging session will then start.

- To see a variable’s contents, double click on it in the source.
Vampir: Recipe (JUQUEEN)

1. module load UNITE vampirserver

2. Start Vampir server component (on frontend) using “vampirserver start smp”
   - Check output for port and pid

3. Connect to server from remote machine (see next slide) and analyze the trace

4. vampirserver stop <pid>
   - See above (2.)
Vampir: Recipe (local system)

1. Open SSH tunnel to JUQUEEN using
   
   "ssh -L30000:localhost:<port> juqueen<\n>

2. Start Vampir client component
   - For example: "/usr/local/zam/unite/bin/vampir"

3. Select
   1. “Open other…”
   2. “Remote file”
   3. “Connect” (keep defaults)
   4. File “traces.otf2” from Score-P trace measurement directory
HPCToolkit: Recipe

1. Compile your code with “-g -qnoipa”
   - For MPI, also make sure your application calls MPI_Comm_rank first on MPI_COMM_WORLD

2. Prefix your *link command* with “hpclink”
   - Ignore potential linker warnings ;-)

3. Run your application as usual, specifying requested metrics with sampling intervals in environment variable “HPCRUN_EVENT_LIST”

4. Perform static binary analysis with “hpcstruct --loop-fwd-subst=no <app>”

5. Combine measurements with “hpcprof -S <struct file> \
   -I “<path_to_src>/*” <measurement_dir>”

6. View results with “hpcviewer <hpct_database>”
TAU: Recipe

1. Load TAU module\# once per session

   module load UNITE tau

1. Specify programming model by setting TAU_MAKEFILE to one of $TAU_MF_DIR/Makefile.tau-*

2. Compile and link with
   - tau_cc.sh file.c ...
   - tau_cxx.sh file.cxx...
   - tau_f90.sh file.f90 ...

3. Execute with real input data
   Environment variables control measurement mode
   - TAU_PROFILE, TAU_TRACE, TAU_CALLPATH, ...

4. Examine results with paraprof