TAU Performance System

Tuning and Analysis Utilities

Sameer Shende
Performance Research Lab, University of Oregon
http://TAU.uoregon.edu
• Parallel performance framework and toolkit
  – Supports all HPC platforms, compilers, runtime system
  – Provides portable instrumentation, measurement, analysis

TAU Performance System® (http://tau.uoregon.edu)

- **Instrumentation**
  - *Source*
    - C, C++, Fortran
    - Python, UPC, Java
    - Robust parsers (PDT)

  - **Wrapping**
    - Interposition (PMPI)
    - Wrapper generation

  - **Linking**
    - Static, dynamic
    - Preloading

  - **Executable**
    - Dynamic (Dyninst)
    - Binary (Dyninst, MAQAO)

- **Measurement**
  - **Events**
    - static/dynamic
    - routine, basic block, loop
    - threading, communication
    - heterogeneous

  - **Profiling**
    - flat, callpath, phase, parameter, snapshot
    - probe, sampling, hybrid

  - **Tracing**
    - TAU / Scalasca tracing
    - Open Trace Format (OTF)

  - **Metadata**
    - system, user-defined

- **Analysis**
  - **Profiles**
    - ParaProf parallel profile analyzer / visualizer
    - PeriDMF parallel profile database
    - PeriExplorer parallel profile data mining

  - **Tracing**
    - TAU trace translation
      - OTF, SLOG-2
    - Trace analysis / visualizer
      - Vampir, Jumpshot

  - **Online**
    - event unification
    - statistics calculation
TAU Performance System®

• Instrumentation
  – Fortran, C++, C, UPC, Java, Python, Chapel
  – Automatic instrumentation

• Measurement and analysis support
  – MPI, OpenSHMEM, ARMCI, PGAS, DMAPP
  – pthreads, OpenMP, hybrid, other thread models
  – GPU, CUDA, OpenCL, OpenACC
  – Parallel profiling and tracing
  – Use of Score-P for native OTF2 and CUBEX generation
  – Efficient callpath profiles and trace generation using Score-P

• Analysis
  – Parallel profile analysis (ParaProf), data mining (PerfExplorer)
  – Performance database technology (PerfDMF, TAUdb)
  – 3D profile browser

VI–HPS TW15: VI–HPS Tuning Workshop, Saclay, France
• Tuning and Analysis Utilities (18+ year project)
• Comprehensive performance profiling and tracing
  – Integrated, scalable, flexible, portable
  – Targets all parallel programming/execution paradigms

• Integrated performance toolkit
  – Instrumentation, measurement, analysis, visualization
  – Widely-ported performance profiling / tracing system
  – Performance data management and data mining
  – Open source (BSD-style license)

• Integrates with application frameworks
Understanding Application Performance using TAU

• How much time is spent in each application routine and outer loops? Within loops, what is the contribution of each statement?
• How many instructions are executed in these code regions? Floating point, Level 1 and 2 data cache misses, hits, branches taken?
• What is the memory usage of the code? When and where is memory allocated/de-allocated? Are there any memory leaks?
• What are the I/O characteristics of the code? What is the peak read and write bandwidth of individual calls, total volume?
• What is the contribution of each phase of the program? What is the time wasted/spent waiting for collectives, and I/O operations in Initialization, Computation, I/O phases?
• How does the application scale? What is the efficiency, runtime breakdown of performance across different core counts?
What Can TAU Do?

• Profiling and tracing
  – **Profiling** shows you *how much* (total) time was spent in each routine
  – **Tracing** shows you *when* the events take place on a timeline

• Multi-language debugging
  – Identify the source location of a crash by unwinding the system callstack
  – Identify memory errors (off-by-one, etc.)

• Profiling and tracing can measure time as well as hardware performance counters (cache misses, instructions) from your CPU
• TAU can automatically instrument your source code using a package called PDT for routines, loops, I/O, memory, phases, etc.
• TAU runs on all HPC platforms and it is free (BSD style license)
• TAU includes instrumentation, measurement and analysis tools
What does TAU support?

- C/C++
- Fortran
- pthreads
- Intel
- MinGW
- UPC
- OpenCL
- OpenACC
- Intel MIC
- LLVM
- PGI
- Cray
- Sun
- AIX
- OpenMP
- MPI
- Java
- GPU
- OpenCL
- ARM
- MPC
- BlueGene
- Fujitsu
- OS X

Insert yours here
Profiling and Tracing

Profiling

- Profiling shows you **how much** (total) time was spent in each routine

- Metrics can be time or hardware performance counters (cache misses, instructions)
- TAU can automatically instrument your source code using a package called PDT for routines, loops, I/O, memory, phases, etc.

Tracing

- Tracing shows you when the events take place on a timeline
Inclusive vs. Exclusive Measurements

- Performance with respect to code regions
- Exclusive measurements for region only
- Inclusive measurements includes child regions

```c
int foo()
{
    int a;
    a = a + 1;
    bar();
    a = a + 1;
    return a;
}
```
## Availability on New Systems

- Intel compilers with Intel MPI on Intel Xeon Phi\textsuperscript{TM} (MIC)
- GPI with Intel Linux x86_64 Infiniband clusters
- IBM BG/Q and Power 7 Linux with IBM XL UPC compilers
- NVIDIA Kepler K20 with CUDA 5.0 with NVCC
- Fujitsu Fortran/C/C++ MPI compilers on the K computer
- PGI compilers with OpenACC support on NVIDIA systems
- Cray CX30 Sandybridge Linux systems with Intel compilers
- Cray CCE compilers with OpenACC support on Cray XK7
- AMD OpenCL libs with GNU on AMD Fusion cluster systems
- MPC compilers on TGCC Curie system (Bull, Linux x86_64)
- GNU compilers on ARM Linux clusters (MontBlanc, BSC)
- Cray CCE compilers with OpenACC on Cray XK6 (K20)
- Microsoft MPI with Mingw compilers under Windows Azure
- LLVM and GNU compilers under Mac OS X, IBM BGQ
TAU Architecture and Workflow

Instrumentation
- Source
  - C, C++, Fortran
  - Python, UPC, Java
  - Robust parsers (PDT)
- Wrapping
  - Interposition (PMPI)
  - Wrapper generation
- Linking
  - Static, dynamic
  - Preloading
- Executable
  - Dynamic (Dyninst)
  - Binary (Dyninst, MAQAO)

Measurement
- Events
  - static/dynamic
  - routine, basic block, loop
  - threading, communication
  - heterogeneous
- Profiling
  - file, callpath, phase, parameter, snapshot
  - probe, sampling, hybrid
- Tracing
  - TAU / Scalasca tracing
  - Open Trace Format (OTF)
- Metadata
  - system, user-defined

Analysis
- Profiles
  - *ParaProf* parallel profile analyzer / visualizer
  - *PerfDMF* parallel profile database
  - *PerfExplorer* parallel profile data mining
- Tracing
  - TAU trace translation
    - OTF, SLOG-2
  - Trace analysis / visualizer
    - Vampir, Jumpshot
- Online
  - event unification
  - statistics calculation
TAU Architecture and Workflow

**Instrumentation:** Add probes to perform measurements
- Source code instrumentation using pre-processors and compiler scripts
- Wrapping external libraries (I/O, MPI, Memory, CUDA, OpenCL, pthread)
- Rewriting the binary executable

**Measurement:** Profiling or tracing using various metrics
- Direct instrumentation (Interval events measure exclusive or inclusive duration)
- Indirect instrumentation (Sampling measures statement level contribution)
- Throttling and runtime control of low-level events that execute frequently
- Per-thread storage of performance data
- Interface with external packages (e.g. PAPI hw performance counter library)

**Analysis:** Visualization of profiles and traces
- 3D visualization of profile data in paraprof or perfexplorer tools
- Trace conversion & display in external visualizers (Vampir, Jumpshot, ParaVer)
Instrumentation

• Direct and indirect performance observation
  – Instrumentation invokes performance measurement
  – Direct measurement with probes
  – Indirect measurement with periodic sampling or hardware performance counter overflow interrupts
  – Events measure performance data, metadata, context, etc.

• User-defined events
  – Interval (start/stop) events to measure exclusive & inclusive duration
  – Atomic events take measurements at a single point
    – Measures total, samples, min/max/mean/std. deviation statistics
  – Context events are atomic events with executing context
    – Measures above statistics for a given calling path
Direct Observation Events

• Interval events (begin/end events)
  – Measures exclusive & inclusive durations between events
  – Metrics monotonically increase
  – Example: Wall-clock timer

• Atomic events (trigger with data value)
  – Used to capture performance data state
  – Shows extent of variation of triggered values (min/max/mean)
  – Example: heap memory consumed at a particular point

• Code events
  – Routines, classes, templates
  – Statement-level blocks, loops
  – Example: for-loop begin/end
Interval and Atomic Events in TAU

<table>
<thead>
<tr>
<th>%Time</th>
<th>Exclusive msec</th>
<th>Inclusive msec</th>
<th>#Call</th>
<th>#Subs</th>
<th>Inclusive Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0</td>
<td>0.187</td>
<td>1.105</td>
<td>1</td>
<td>44</td>
<td>1105659 int main(int, char **) C</td>
</tr>
<tr>
<td>93.2</td>
<td>1.030</td>
<td>1.030</td>
<td>1</td>
<td>0</td>
<td>1030654 MPI_Init()</td>
</tr>
<tr>
<td>5.9</td>
<td>0.879</td>
<td>65</td>
<td>40</td>
<td>320</td>
<td>1637 void func(int, int) C</td>
</tr>
<tr>
<td>4.6</td>
<td>51</td>
<td>51</td>
<td>40</td>
<td>0</td>
<td>1277 MPI_Barrier()</td>
</tr>
<tr>
<td>1.2</td>
<td>13</td>
<td>13</td>
<td>120</td>
<td>0</td>
<td>111 MPI_Recv()</td>
</tr>
<tr>
<td>0.8</td>
<td>9</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>9328 MPI_Finalize()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.137</td>
<td>0.137</td>
<td>120</td>
<td>0</td>
<td>1 MPI_Send()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.086</td>
<td>0.086</td>
<td>40</td>
<td>0</td>
<td>2 MPI_Bcast()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.002</td>
<td>0.002</td>
<td>1</td>
<td>0</td>
<td>2 MPI_Comm_size()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.001</td>
<td>0.001</td>
<td>1</td>
<td>0</td>
<td>1 MPI_Comm_rank()</td>
</tr>
</tbody>
</table>

**USER EVENTS Profile:** NODE 0, CONTEXT 0, THREAD 0

<table>
<thead>
<tr>
<th>NumSamples</th>
<th>MaxValue</th>
<th>MinValue</th>
<th>MeanValue</th>
<th>Std. Dev.</th>
<th>Event Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>365</td>
<td>5.138E+04</td>
<td>44.39</td>
<td>3.09E+04</td>
<td>1.234E+04</td>
<td>Heap Memory Used (KB) : Entry</td>
</tr>
<tr>
<td>365</td>
<td>5.138E+04</td>
<td>2064</td>
<td>3.115E+04</td>
<td>1.21E+04</td>
<td>Heap Memory Used (KB) : Exit</td>
</tr>
<tr>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>0</td>
<td>Message size for broadcast</td>
</tr>
</tbody>
</table>

% export TAU_CALLPATH_DEPTH=0
% export TAU_TRACK_HEAP=1
### Atomic Events and Context Events

#### Controls depth of executing context shown in profiles

% export TAU_CALLPATH_DEPTH=1
% export TAU_TRACK_HEAP=1

#### Atomic Events

<table>
<thead>
<tr>
<th>Time</th>
<th>Exclusive</th>
<th>Inclusive</th>
<th>#Call</th>
<th>#Subrs</th>
<th>Inclusive Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.253</td>
<td>1.106</td>
<td>1</td>
<td>44</td>
<td>int main(char **) C</td>
</tr>
<tr>
<td>93.2</td>
<td>1.031</td>
<td>1.031</td>
<td>1</td>
<td>0</td>
<td>MPT_Init()</td>
</tr>
<tr>
<td>6.0</td>
<td>1</td>
<td>66</td>
<td>40</td>
<td>320</td>
<td>void func(int, int) C</td>
</tr>
<tr>
<td>5.7</td>
<td>63</td>
<td>63</td>
<td>40</td>
<td>0</td>
<td>MPT_Barrier()</td>
</tr>
<tr>
<td>0.8</td>
<td>9</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>MPT_Finalize()</td>
</tr>
<tr>
<td>0.1</td>
<td>1</td>
<td>1</td>
<td>120</td>
<td>0</td>
<td>10 M Recv()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.141</td>
<td>0.141</td>
<td>120</td>
<td>0</td>
<td>1 M Send()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.085</td>
<td>0.085</td>
<td>40</td>
<td>0</td>
<td>2 M Bcast()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.001</td>
<td>0.001</td>
<td>1</td>
<td>0</td>
<td>1 M Comm_size()</td>
</tr>
<tr>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0 M Comm_rank()</td>
</tr>
</tbody>
</table>

#### Context Events

Context events are atomic events with executing context.
% export TAU_CALLPATHDEPTH=2
% export TAU_TRACK_HEAP=1

Context Events with Callpath

<table>
<thead>
<tr>
<th>ZTime</th>
<th>Exclusive msec</th>
<th>Inclusive total msec</th>
<th>#Call</th>
<th>#Subrs</th>
<th>Inclusive Name usec/call</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0</td>
<td>0.357</td>
<td>1.114</td>
<td>1</td>
<td>44</td>
<td>1114040 int main(int, char **)</td>
</tr>
<tr>
<td>92.6</td>
<td>1.031</td>
<td>1.031</td>
<td>1</td>
<td>0</td>
<td>1031066 MPI_Init()</td>
</tr>
<tr>
<td>6.7</td>
<td>72</td>
<td>74</td>
<td>40</td>
<td>320</td>
<td>1865 void func(int, int) C</td>
</tr>
<tr>
<td>0.7</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>8002 MPI_Finalize()</td>
</tr>
<tr>
<td>0.1</td>
<td>1</td>
<td>1</td>
<td>120</td>
<td>0</td>
<td>12 MPI_Recv()</td>
</tr>
<tr>
<td>0.1</td>
<td>0.608</td>
<td>0.608</td>
<td>40</td>
<td>0</td>
<td>15 MPI_Barrier()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.136</td>
<td>0.136</td>
<td>120</td>
<td>0</td>
<td>1 MPI_Send()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.095</td>
<td>0.095</td>
<td>40</td>
<td>0</td>
<td>2 MPI_Bcast()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.001</td>
<td>0.001</td>
<td>1</td>
<td>0</td>
<td>1 MPI_Comm_size()</td>
</tr>
<tr>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0 MPI_Comm_rank()</td>
</tr>
</tbody>
</table>

USER EVENTS Profile : NODE 0, CONTEXT 0, THREAD 0

<table>
<thead>
<tr>
<th>NumSamples</th>
<th>MaxValue</th>
<th>MinValue</th>
<th>MeanValue</th>
<th>Std. Dev.</th>
<th>Event Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>365</td>
<td>5.139E+04</td>
<td>3.091E+04</td>
<td>1.234E+04</td>
<td>Heap</td>
<td>Memory Used (KB) : Entry</td>
</tr>
<tr>
<td>1</td>
<td>44.39</td>
<td>44.39</td>
<td>44.39</td>
<td>Heap</td>
<td>Memory Used (KB) : Entry : int main(int, char **)</td>
</tr>
<tr>
<td>1</td>
<td>2068</td>
<td>2068</td>
<td>2068</td>
<td>Heap</td>
<td>Memory Used (KB) : Entry : int main(int, char **) C =&gt; MPI_Comm_rank()</td>
</tr>
<tr>
<td>1</td>
<td>5.139E+04</td>
<td>5.139E+04</td>
<td>5.139E+04</td>
<td>Heap</td>
<td>Memory Used (KB) : Entry : int main(int, char **) C =&gt; MPI_Comm_size()</td>
</tr>
<tr>
<td>1</td>
<td>57.58</td>
<td>57.58</td>
<td>57.58</td>
<td>Heap</td>
<td>Memory Used (KB) : Entry : int main(int, char **) C =&gt; MPI_Finalize()</td>
</tr>
<tr>
<td>40</td>
<td>5.036E+04</td>
<td>3.011E+04</td>
<td>1.228E+04</td>
<td>Heap</td>
<td>Memory Used (KB) : Entry : int main(int, char **) C =&gt; void func(int, int)</td>
</tr>
<tr>
<td>40</td>
<td>3.114E+04</td>
<td>3.114E+04</td>
<td>3.114E+04</td>
<td>Heap</td>
<td>Memory Used (KB) : Entry : void func(int, int) C =&gt; MPI_Bcast()</td>
</tr>
<tr>
<td>120</td>
<td>5.139E+04</td>
<td>1.136E+04</td>
<td>1.187E+04</td>
<td>Heap</td>
<td>Memory Used (KB) : Entry : void func(int, int) C =&gt; MPI_Send()</td>
</tr>
<tr>
<td>365</td>
<td>5.139E+04</td>
<td>2065</td>
<td>3.116E+04</td>
<td>Heap</td>
<td>Memory Used (KB) : Exit</td>
</tr>
</tbody>
</table>

Callpath shown on context events
Direct Instrumentation Options in TAU

• Source Code Instrumentation
  – Automatic instrumentation using pre-processor based on static analysis of source code (PDT), creating an instrumented copy
  – Compiler generates instrumented object code
  – Manual instrumentation
• Library Level Instrumentation
  – Statically or dynamically linked wrapper libraries
    • MPI, I/O, memory, etc.
  – Wrapping external libraries where source is not available
• Runtime pre-loading and interception of library calls
• Binary Code instrumentation
  – Rewrite the binary, runtime instrumentation
• Virtual Machine, Interpreter, OS level instrumentation
Using TAU

• TAU supports several measurement and thread options
  Phase profiling, profiling with hardware counters, MPI library, CUDA…
  Each measurement configuration of TAU corresponds to a unique stub
  makefile (configuration file) and library that is generated when you
  configure it

• To instrument source code automatically using PDT
  Choose an appropriate TAU stub makefile in <arch>/lib:

  % export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-mpi-pdt
  % export TAU_OPTIONS=‘-optVerbose …’ (see tau_compiler.sh)
  % export PATH=$TAU_ROOT/x86_64/bin:$PATH
  % export TAU=$TAU_ROOT/x86_64/lib

  Use tau_f90.sh, tau_cxx.sh, tau_upc.sh, or tau_cc.sh as F90, C++,
  UPC, or C compilers respectively:

  % mpif90 foo.f90 changes to
  % tau_f90.sh foo.f90

• Set runtime environment variables, execute application and
  analyze performance data:

  % pprof (for text based profile display)  % paraprof (for GUI)
Automatic Source Instrumentation using PDT

% module load UNITE VI-HPS-TW; ls $TAU/Makefile.*
Makefile.tau-icpc
Makefile.tau-icpc-cupti-pdt
Makefile.tau-icpc-mpi-cupti-pdt
Makefile.tau-icpc-mpi-pdt
Makefile.tau-icpc-mpi-pdt-openmp
Makefile.tau-icpc-mpi-pdt-openmp-opari
Makefile.tau-icpc-mpi-thread-pdt
Makefile.tau-icpc-ompt-mpi-pdt-openmp
Makefile.tau-icpc-papi-mpi-pdt-openmp-opari-scorep
Makefile.tau-icpc-papi-mpi-pdt-scorep
Makefile.tau-icpc-papi-ompt-mpi-pdt-openmp
Makefile.tau-mpc250-mpc-mpi-pdt

• For an MPI+F90 application with Intel MPI, you may choose
  Makefile.tau-mpi-pdt
    - Supports MPI instrumentation & PDT for automatic source instrumentation
  % export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-mpi-pdt
  % tau_f90.sh matmult.f90 -o matmult
  % mpirun -np 4 ./matmult
  % paraprof
Using TAU with Score-P

% export TAU=$TAU_ROOT/x86_64/lib
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt-openmp-opari-scorep
% export OMP_NUM_THREADS=10
% make CC=tau_cc.sh CXX=tau_cxx.sh F90=tau_f90.sh

% mpirun -np 4 ./matmult

% cd score*; paraprof profile.cubex &
% export TAU=$TAU_ROOTDIR/mic_linux/lib
% ls $TAU/Makefile.*
Makefile.tau-intelmpi-icpc-mpi-pdt
Makefile.tau-intelmpi-icpc-papi-mpi-pdt
Makefile.tau-intelmpi-icpc-papi-mpi-pdt-openmp-opari

• For an MPI+F90 application with Intel MPI, you may choose
  Makefile.tau-intelmpi-icpc-papi-mpi-pdt
  – Supports MPI instrumentation & PDT for automatic source instrumentation
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt
• % tau_f90.sh matrix.f90 -o matrix
• % idev -m 50;
• % export MIC_PPN=6
• % export MIC_OMP_NUM_THREADS=10
• ibrun.symm -m ./matrix
• % paraprof
安装和配置TAU

• 安装PDT:
  - `wget http://tau.uoregon.edu/pdt_lite.tgz`
  - `./configure --prefix=<dir>; make; make install`

• 安装TAU:
  - `wget http://tau.uoregon.edu/tau.tgz`
  - `./configure --arch=x86_64 -bfd=download -pdt=<dir> -papi=<dir> ...
    - For MIC:
    - `./configure --arch=mic_linux -pdt=<dir> -pdt_c++=g++ -papi=dir ...
    - `make install`

• 使用TAU:
  - `export TAU_MAKEFILE=<taudir>/x86_64/lib/Makefile.tau-<TAGS>`
  - `make CC=tau_cc.sh CXX=tau_cxx.sh F90=tau_f90.sh`
Compile-Time Options

- Optional parameters for the TAU_OPTIONS environment variable:
  
  % tau_compiler.sh

  -optVerbose Turn on verbose debugging messages
  -optCompInst Use compiler based instrumentation
  -optNoCompInst Do not revert to compiler instrumentation if source instrumentation fails.
  -optTrackIO Wrap POSIX I/O call and calculates vol/bw of I/O operations (Requires TAU to be configured with --iowrapper)
  -optMemDbg Runtime bounds checking (see TAU_MEMDBG_* env vars)
  -optKeepFiles Does not remove intermediate .pdb and .inst.* files
  -optPreProcess Preprocess sources (OpenMP, Fortran) before instrumentation
  -optTauSelectFile="<file>" Specify selective instrumentation file for tau_instrumentor
  -optTauWrapFile="<file>" Specify path to link_options.tau generated by tau_gen_wrapper
  -optHeaderInst Enable Instrumentation of headers
  -optTrackUPCR Track UPC runtime layer routines (used with tau_upc.sh)
  -optLinking="" Options passed to the linker. Typically $(TAU_MPI_FLIBS) $(TAU_LIBS) $(TAU_CXXLIBS)
  -optCompile="" Options passed to the compiler. Typically $(TAU_MPI_INCLUDE) $(TAU_INCLUDE) $(TAU_DEFS)
  -optPdtF95Opts="" Add options for Fortran parser in PDT (f95parse/gfparse) …
## Runtime Environment Variables

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAU_TRACE</td>
<td>0</td>
<td>Setting to 1 turns on tracing</td>
</tr>
<tr>
<td>TAU_CALLPATH</td>
<td>0</td>
<td>Setting to 1 turns on callpath profiling</td>
</tr>
<tr>
<td>TAU_TRACK_MEMORY_LEAKS</td>
<td>0</td>
<td>Setting to 1 turns on leak detection (for use with –optMemDbg or tau_exec)</td>
</tr>
<tr>
<td>TAU_MEMDBG_PROTECT_ABOVE</td>
<td>0</td>
<td>Setting to 1 turns on bounds checking for dynamically allocated arrays. (Use with –optMemDbg or tau_exec –memory_debug).</td>
</tr>
<tr>
<td>TAU_CALLPATH DEPTH</td>
<td>2</td>
<td>Specifies depth of callpath. Setting to 0 generates no callpath or routine information, setting to 1 generates flat profile and context events have just parent information (e.g., Heap Entry: foo)</td>
</tr>
<tr>
<td>TAU_TRACK_IO_PARAMS</td>
<td>0</td>
<td>Setting to 1 with –optTrackIO or tau_exec –io captures arguments of I/O calls</td>
</tr>
<tr>
<td>TAU_TRACK_SIGNALS</td>
<td>0</td>
<td>Setting to 1 generate debugging callstack info when a program crashes</td>
</tr>
<tr>
<td>TAU_COMM_MATRIX</td>
<td>0</td>
<td>Setting to 1 generates communication matrix display using context events</td>
</tr>
<tr>
<td>TAU_THROTTLE</td>
<td>1</td>
<td>Setting to 0 turns off throttling. Enabled by default to remove instrumentation in lightweight routines that are called frequently</td>
</tr>
<tr>
<td>TAU_THROTTLE_NUMCALLS</td>
<td>100000</td>
<td>Specifies the number of calls before testing for throttling</td>
</tr>
<tr>
<td>TAU_THROTTLE_PERCALL</td>
<td>10</td>
<td>Specifies value in microseconds. Throttle a routine if it is called over 100000 times and takes less than 10 usec of inclusive time per call</td>
</tr>
<tr>
<td>TAU_COMPENSATE</td>
<td>0</td>
<td>Setting to 1 enables runtime compensation of instrumentation overhead</td>
</tr>
<tr>
<td>TAU_PROFILE_FORMAT</td>
<td>Profile</td>
<td>Setting to “merged” generates a single file. “snapshot” generates xml format</td>
</tr>
<tr>
<td>TAU_METRICS</td>
<td>TIME</td>
<td>Setting to a comma separated list generates other metrics. (e.g., TIME:P_VIRTUAL_TIME:PAPI_FP_INS:PAPI_NATIVE_{event}&lt;subevent&gt;)</td>
</tr>
</tbody>
</table>
Compiling Fortran Codes with TAU

- If your Fortran code uses free format in .f files (fixed is default for .f), you may use:%
  export TAU_OPTIONS=‘-optPdtF95Opts="-R free" -optVerbose’

- To use the compiler based instrumentation instead of PDT (source-based):
  export TAU_OPTIONS=‘ -optCompInst -optVerbose’

- If your Fortran code uses C preprocessor directives (#include, #ifdef, #endif):
  export TAU_OPTIONS=‘-optPreProcess -optVerbose’

- To use an instrumentation specification file:
  export TAU_OPTIONS=‘-optTauSelectFile=select.tau -optVerbose -optPreProcess’
  cat select.tau
  BEGIN_EXCLUDE_LIST
  FOO
  END_EXCLUDE_LIST
  BEGIN_INSTRUMENT_SECTION
  loops routine="#"
  # this statement instruments all outer loops in all routines. # is wildcard as well as comment in first column.
  END_INSTRUMENT_SECTION
Binary Rewriting Instrumentation

- Support for both static and dynamic executables
- Specify a list of routines to instrument
- Specify the TAU measurement library to be injected

**MAQAO:**

```
% tau_rewrite -T [tags] [-f select.tau] a.out
[-o] a.inst
```

**Dyninst:**

```
% tau_run -T [tags] [-f select.tau] a.out -o a.inst
```

**Pebil:**

```
% tau_pebil_rewrite -T [tags] [-f select.tau] a.out
-o a.inst
```

**Execute the application to get measurement data:**

```
% mpirun -np 4 ./a.inst
```
Selective Instrumentation in tau_rewrite

- `tau_rewrite -T icpc,mpi,pdt a.out a.inst`
- `cat select.tau`

```
BEGIN_EXCLUDE_LIST
compute#
foo
END_EXCLUDE_LIST
```

- `tau_rewrite -f select.tau -T icpc,mpi,pdt a.out a.inst`
- `mpirun -np 4 ./a.inst`

Rewrites `a.out` and runs the instrumented code. Routine names must match profiles.
Support Acknowledgments

• US Department of Energy (DOE)
  – Office of Science contracts
  – SciDAC, LBL contracts
  – LLNL-LANL-SNL ASC/NNSA contract
  – Battelle, PNNL contract
  – ANL, ORNL contract
• Department of Defense (DoD)
  – PETTT, HPCMP
• National Science Foundation (NSF)
  – Glassbox, SI-2
• University of Tennessee, Knoxville
• T.U. Dresden, GWT
• Juelich Supercomputing Center
Download TAU from U. Oregon

http://tau.uoregon.edu

http://www.hpclinux.com [LiveDVD]

Free download, open source, BSD license