TAU PERFORMANCE SYSTEM

Sameer Shende
Chee Wai Lee, Wyatt Spear, Scott Biersdorff, Suzanne Millstein
Performance Research Lab

Allen D. Malony, Nick Chaimov, William Voorhees
Department of Computer and Information Science
University of Oregon
TAU Performance System®

• 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)

• Easy to integrate in application frameworks

http://tau.uoregon.edu
What is TAU?

- TAU is a performance evaluation tool
- It supports parallel profiling and tracing
- Profiling shows you how much (total) time was spent in each routine
- Tracing shows you *when* the events take place in each process along a timeline
- 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 most HPC platforms and it is free (BSD style license)
- TAU has instrumentation, measurement and analysis tools
  - paraprof is TAU’s 3D profile browser
- To use TAU’s automatic source instrumentation, you may set a couple of environment variables and substitute the name of your compiler with a TAU shell script
TAU: Usage Scenarios

- 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 peak heap memory usage of the code? When and where is memory allocated/de-allocated? Are there any memory leaks?

- How much time does the application spend performing I/O? What is the peak read and write bandwidth of individual calls, total volume?

- What is the contribution of different phases 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?
Using TAU: Simplest Case

• Uninstrumented code:
  – % mpirun -np 8 ./a.out

• With TAU:
  – % mpirun -np 8 tau_exec ./a.out
  – % paraprof
ParaProf: Mflops Sorted by Exclusive Time

<table>
<thead>
<tr>
<th>Metric: PAPL_FP_INS / LINUX_TIMERS</th>
<th>Value: Exclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Units: Derived metric shown in microseconds format</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Loop</th>
<th>Mflops</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHEMKIN_M::REACTION_RATE_VEC</td>
<td>1142.273</td>
</tr>
<tr>
<td>TRANSPORT_M::COMPUTECOEFFICIENTS</td>
<td>884.23</td>
</tr>
<tr>
<td>MPI_Wait()</td>
<td>0.014</td>
</tr>
<tr>
<td>TRANSPORT_M::COMPUTESPECIESDIFFFLUX</td>
<td>121.843</td>
</tr>
<tr>
<td>MPI_Send()</td>
<td>0.021</td>
</tr>
<tr>
<td>RHSF</td>
<td>0.06</td>
</tr>
<tr>
<td>INTEGRATE</td>
<td>183.176</td>
</tr>
<tr>
<td>MPI_Comm_split()</td>
<td>0</td>
</tr>
<tr>
<td>TRANSPORT_M::COMPUTEHEATFLUX</td>
<td>148.608</td>
</tr>
<tr>
<td>RHSF</td>
<td>269.897</td>
</tr>
<tr>
<td>TRANSPORT_M::COMPUTESCRIPT</td>
<td>163.177</td>
</tr>
<tr>
<td>RHSF</td>
<td>174.081</td>
</tr>
<tr>
<td>TRANSPORT_M::COMPUTESCRIPT</td>
<td>124.129</td>
</tr>
<tr>
<td>RHSF</td>
<td>149.472</td>
</tr>
<tr>
<td>DERIVATIVE_X_CALC</td>
<td>742.83</td>
</tr>
<tr>
<td>DERIVATIVE_X_COMM</td>
<td>2.177</td>
</tr>
<tr>
<td>DERIVATIVE_Y_CALC</td>
<td>178.357</td>
</tr>
<tr>
<td>VARIABLES_M::GET_MASS_FRAC</td>
<td>374.708</td>
</tr>
<tr>
<td>DERIVATIVE_Y_COMM</td>
<td>3.778</td>
</tr>
<tr>
<td>THERMCRYSTAL_MO::CALC_SPECEN Hairstyle</td>
<td>244.308</td>
</tr>
<tr>
<td>DERIVATIVE_Z_CALC</td>
<td>1.294</td>
</tr>
<tr>
<td>THERMCRYSTAL_MO::CALC_SPECEN Hairstyle</td>
<td>0.002</td>
</tr>
<tr>
<td>DERIVATIVE_Z_COMM</td>
<td>7.111</td>
</tr>
<tr>
<td>DERIVATIVE_X_CALC</td>
<td>733.265</td>
</tr>
<tr>
<td>DERIVATIVE_X_CALC</td>
<td>765.947</td>
</tr>
<tr>
<td>DERIVATIVE_Y_CALC</td>
<td>950.372</td>
</tr>
<tr>
<td>DERIVATIVE_Z_CALC</td>
<td>973.95</td>
</tr>
<tr>
<td>TRANSPORT_M::COMPUTESCRIPT</td>
<td>249.638</td>
</tr>
<tr>
<td>DERIVATIVE_X_CALC</td>
<td>1119.025</td>
</tr>
<tr>
<td>DERIVATIVE_Z_CALC</td>
<td>0.328</td>
</tr>
<tr>
<td>DERIVATIVE_Y_CALC</td>
<td>1257.338</td>
</tr>
<tr>
<td>DERIVATIVE_Y_CALC</td>
<td>84.464</td>
</tr>
<tr>
<td>DERIVATIVE_Z_CALC</td>
<td>30.583</td>
</tr>
<tr>
<td>DERIVATIVE_X_CALC</td>
<td>33.405</td>
</tr>
<tr>
<td>DERIVATIVE_Y_CALC</td>
<td>61.032</td>
</tr>
<tr>
<td>THERMCRYSTAL_MO::CALC_INN AVG MOL WT</td>
<td>45.93</td>
</tr>
<tr>
<td>GET_VELOCITY_VEC</td>
<td>103.526</td>
</tr>
<tr>
<td>TRANSPORT_M::COMPUTESCRIPT</td>
<td>196.839</td>
</tr>
<tr>
<td>TRANSPORT_M::COMPUTESCRIPT</td>
<td>293.594</td>
</tr>
<tr>
<td>TRANSPORT_M::COMPUTESCRIPT</td>
<td>141.127</td>
</tr>
<tr>
<td>TRANSPORT_M::COMPUTESCRIPT</td>
<td>24.465</td>
</tr>
<tr>
<td>INTEGRATE</td>
<td>0</td>
</tr>
<tr>
<td>VARIABLES_M::GET_MASS_FRAC</td>
<td>0.245</td>
</tr>
</tbody>
</table>
Parallel Profile Visualization: ParaProf
How does TAU work?

- **Instrumentation**: Adds 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 wallclock time or hardware counters
  - 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 (Scalasca, VampirTrace, Score-P, PAPI)

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

• TAU supports several measurement and thread options
  – Phase profiling, profiling with hardware counters, trace with Score-P...

• Each measurement configuration of TAU corresponds to a unique stub makefile 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-mpi-pdt
    % export TAU_OPTIONS=‘-optVerbose ...’ (see tau_compiler.sh )
  Use tau_f90.sh, tau_cxx.sh or tau_cc.sh as F90, C++ or C compilers:
    % 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)
% cd $TAUROOTDIR/<arch>/lib; ls Makefile.*
Makefile.tau-pdt
Makefile.tau-mpi-pdt
Makefile.tau-pthread-pdt
Makefile.tau-papi-mpi-pdt
Makefile.tau-mpi-pthread-pdt
Makefile.tau-papi-pthread-pdt
Makefile.tau-opari-openmp-mpi-pdt
Makefile.tau-papi-mpi-pdt-epilog-scalasca-trace
Makefile.tau-papi-mpi-pdt-vampirtrace-trace ...

• **For an MPI+F90 application, you may choose** Makefile.tau-mpi-pdt
  – Supports MPI instrumentation & PDT for automatic source instrumentation
  – % export TAU_MAKEFILE=$TAU/Makefile.tau-mpi-pdt
  – % tau_f90.sh matrix.f90 -o matrix
  – % mpirun –np 8 ./matrix
  – % paraprof
TAU Instrumentation Approach

• Supports both direct and indirect performance observation
  – Direct instrumentation of program (system) code (probes)
  – Instrumentation invokes performance measurement
  – Event measurement: performance data, meta-data, context
  – Indirect mode supports sampling based on periodic timer or hardware performance counter overflow based interrupts

• Support for user-defined events
  – *Interval* (Start/Stop) events to measure exclusive & inclusive duration
  – *Atomic events* (Trigger at a single point with data, e.g., heap memory)
    • 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

• Event types
  – Interval events (begin/end events)
    • Measures exclusive & inclusive durations between events
    • Metrics monotonically increase
  – Atomic events (trigger with data value)
    • Used to capture performance data state
    • Shows extent of variation of triggered values (min/max/mean)

• Code events
  – Routines, classes, templates
  – Statement-level blocks, loops
Inclusive and Exclusive Profiles

- 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;
}
```
Interval Events, Atomic Events in TAU

**Interval events**
- e.g., routines (start/stop)
- show duration

**Atomic events**
- (triggered with value)
- show extent of variation (min/max/mean)

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

<table>
<thead>
<tr>
<th>Time</th>
<th>Exclusive msec</th>
<th>Inclusive total msec</th>
<th>#Call</th>
<th>#Subs</th>
<th>Inclusive Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0</td>
<td>0.253</td>
<td>1.106</td>
<td>1</td>
<td>44</td>
<td>1106701 int main(int, char **) C</td>
</tr>
<tr>
<td>93.2</td>
<td>1.031</td>
<td>1.031</td>
<td>1</td>
<td>0</td>
<td>1031311 MPI_Init()</td>
</tr>
<tr>
<td>6.0</td>
<td>1</td>
<td>66</td>
<td>40</td>
<td>320</td>
<td>1850 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>1588 MPI_Barrier()</td>
</tr>
<tr>
<td>0.8</td>
<td>9</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>9119 MPI_Finalize()</td>
</tr>
<tr>
<td>0.1</td>
<td>1</td>
<td>1</td>
<td>120</td>
<td>0</td>
<td>10 MPI_Recv()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.141</td>
<td>0.141</td>
<td>120</td>
<td>0</td>
<td>1 MPI_Send()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.085</td>
<td>0.085</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>

### Context Events

- % export TAU_CALLPATH_DEPTH=1
- % export TAU_TRACK_HEAP=1

**SC 11: Hands-on Practical Hybrid Parallel Application Performance Engineering**

**Controlling depth of executing context shown in profiles**

**Context events**

+ executing context

**Atomic events**

- Controls depth of executing context shown in profiles

---

% export TAU_CALLPATH_DEPTH=1

% export TAU_TRACK_HEAP=1
% export TAU_CALLPATH_DEPTH=2
% export TAU_TRACK_HEAP=1

Context Events (Default)

NODE 0: CONTEXT 0: THREAD 0:

<table>
<thead>
<tr>
<th>%Time</th>
<th>Exclusive msec</th>
<th>Inclusive total msec</th>
<th>#Call</th>
<th>#Subrs</th>
<th>Inclusive Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0</td>
<td>0.39/</td>
<td>1.114</td>
<td>1</td>
<td>44</td>
<td>1114040 int main(int, char **) C</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 5.139E+04</td>
<td>44.39</td>
<td>3.091E+04</td>
<td>1.234E+04</td>
<td>Heap Memory Used (KB) : Entry</td>
<td></td>
</tr>
<tr>
<td>1 44.39</td>
<td>44.39</td>
<td>44.39</td>
<td>0</td>
<td>Heap Memory Used (KB) : Entry : int main(int, char **) C =&gt; MPI_Comm_rank()</td>
<td></td>
</tr>
<tr>
<td>1 2068</td>
<td>2068</td>
<td>2068</td>
<td>0</td>
<td>Heap Memory Used (KB) : Entry : int main(int, char **) C =&gt; MPI_Comm_size() =&gt; MPI_Finalize()</td>
<td></td>
</tr>
<tr>
<td>1 5.139E+04</td>
<td>5.139E+04</td>
<td>5.139E+04</td>
<td>0</td>
<td>Heap Memory Used (KB) : Entry : int main(int, char **) C =&gt; MPI_Init() =&gt; MPI_Bcast() =&gt;MPI_Send() =&gt; MPI_Recv()</td>
<td></td>
</tr>
<tr>
<td>1 57.58</td>
<td>57.58</td>
<td>57.58</td>
<td>0</td>
<td>Heap Memory Used (KB) : Entry : int main(int, char **) C =&gt; void func(int, int) C =&gt; void func(int, int) C =&gt; void func(int, int) C</td>
<td></td>
</tr>
<tr>
<td>40 5.036E+04</td>
<td>2069</td>
<td>3.011E+04</td>
<td>1.228E+04</td>
<td>Heap Memory Used (KB) : Entry : void func(int, int) C =&gt; MPI_Bcast() =&gt; MPI_Send() =&gt; MPI_Recv()</td>
<td></td>
</tr>
<tr>
<td>40 5.139E+04</td>
<td>3098</td>
<td>3.114E+04</td>
<td>1.227E+04</td>
<td>Heap Memory Used (KB) : Entry : void func(int, int) C =&gt; MPI_Finalize()</td>
<td></td>
</tr>
<tr>
<td>120 5.139E+04</td>
<td>1.13E+04</td>
<td>3.134E+04</td>
<td>1.187E+04</td>
<td>Heap Memory Used (KB) : Entry : void func(int, int) C =&gt; MPI_Finalize()</td>
<td></td>
</tr>
<tr>
<td>120 5.139E+04</td>
<td>1.13E+04</td>
<td>3.134E+04</td>
<td>1.187E+04</td>
<td>Heap Memory Used (KB) : Entry : void func(int, int) C =&gt; MPI_Finalize()</td>
<td></td>
</tr>
<tr>
<td>365 5.139E+04</td>
<td>2065</td>
<td>3.116E+04</td>
<td>1.21E+04</td>
<td>Heap Memory Used (KB) : Entry</td>
<td></td>
</tr>
</tbody>
</table>

Context event = atomic event + executing context
• Source Code Instrumentation
  – Manual instrumentation
  – Automatic instrumentation using pre-processor based on static analysis of source code (PDT), creating an instrumented copy
  – Compiler generates instrumented object code

• Library Level Instrumentation
  – Wrapper libraries for standard MPI libraries using PMPI interface
  – 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
TAU’s Static Analysis System: Program Database Toolkit (PDT)

- Application/ Library
  - C/C++ parser
  - Fortran parser F77/90/95
  - IL
  - C/C++ IL analyzer

- Automatic source instrumentation
- DUCTAPE
- Program Database Files
Automatic Source Instrumentation using PDT

- TAU source analyzer
- Application source
- Parsed program
- tau_instrumentor
- Instrumentation specification file
- Instrumented copy of source
To instrument source code using PDT

- Choose an appropriate TAU stub makefile from
  `<taudir>/<arch>/lib/Makefile.tau*:
  (typically, `arch=i386_linux, x86_64, craycnl, bgp, cygwin ... and
  `taudir=/usr/local/packages/tau on LiveDVD)

  % export TAU_MAKEFILE=$TAU/Makefile.tau-mpi-pdt
  % make CC=tau_cc.sh CXX=tau_cxx.sh F90=tau_f90.sh

- Execute application and analyze performance data:
  % pprof  (for text based profile display)
  % paraprof (for GUI)
How much time is spent in each application routine?

Value: Exclusive
Units: seconds

- LEQ_IKSWEPT: 9647.318 seconds
- LEQ_BICGSOT: 4357.213 seconds
- LEQ_MATVECT: 2669.887 seconds
- SOLVE_SPECIES_EQ: 1777.752 seconds
- SOLVE_LIN_EQ: 1417.986 seconds
- PHYSICAL_PROP: 1028.448 seconds
- RRATES: 783.402 seconds
- LEQ_MSOLVET: 682.376 seconds
- INIT_AB_M: 530.858 seconds
- CALC_MASS_FLUX_SPHR: 463.788 seconds
- INIT_MU_S: 446.025 seconds
- CALC_RESID_S: 421.747 seconds
- SOLVE_ENERGY_EQ: 381.363 seconds
- SOURCE_PHI: 371.199 seconds
- DRAG_GS: 258.829 seconds
Solution: Generating a flat profile with MPI

% export TAU_MAKEFILE=<taudir>/<arch>/lib/Makefile.tau-mpi-pdt
% export PATH=<taudir>/<arch>/bin:$PATH

Or

% module load tau
% make F90=tau_f90.sh

Or

% tau_f90.sh matmult.f90
% mpirun -np 8 ./a.out
% paraprof

To view. To view the data locally on the workstation,
% paraprof --pack app.ppk
    Move the app.ppk file to your desktop.

% paraprof app.ppk

Click on the “node 0” label to see profile for that node. Right click to see other options. Windows -> 3D Visualization for 3D window.
Automatic Instrumentation

- We now provide compiler wrapper scripts
  - Simply replace CC with tau_cxx.sh
  - Automatically instruments C++ and C source code, links with TAU MPI Wrapper libraries.
- Use tau_cc.sh and tau_f90.sh for C and Fortran

**Before**

```
CXX = mpicxx
F90 = mpif90
CXXFLAGS =
LIBS = -lm
OBJJS = f1.o f2.o f3.o ... fn.o

app: $(OBJJS)
  $(CXX) $(CXXFLAGS) -c $<

.cpp.o:
  $(CXX) $(CXXFLAGS) -c $<
```

**After**

```
CXX = tau_cxx.sh
F90 = tau_f90.sh
CXXFLAGS =
LIBS = -lm
OBJJS = f1.o f2.o f3.o ... fn.o

app: $(OBJJS)
  $(CXX) $(CXXFLAGS) $(LIBS) $(OBJJS) -o $@

.cpp.o:
  $(CXX) $(CXXFLAGS) $(LIBS) -c $<
```
• Go to tutorial directory

```bash
% cd tutorial/NPB3.3-MZ-MPI/
```

• Activate the TAU compiler wrappers

```bash
% vim config/make.def

#MPIF77 = mpif77
MPIF77 = tau_f90.sh -tau_makefile=<file>
```
Hands-on: NPB 3.3 BT-MZ-MPI

- Re-compile

% make clean; make bt-mz CLASS=W NPROCS=4

Reconsider number of ranks and buffer sizes when running the LiveDVD on a laptop computer.

% cd bin.tau
% export OMP_NUM_THREADS=4
% mpiexec -np 4 ./bt-mz.W.4
% paraprof &
Passing Optional Parameters to TAU Compiler Scripts

- See `<taudir>/<arch>/bin/tau_compiler.sh -help`

Compilation:

% ftn -c foo.f90

Changes to

% gparse foo.f90 $(OPT1)
% tau_instrumentor foo.pdb foo.f90 -o foo.inst.f90 $(OPT2)
% ftn -c foo.inst.f90 -o foo.o $(OPT3)

- Linking:

% ftn foo.o bar.o -o app

Changes to

% ftn foo.o bar.o -o app <taulibs> $(OPT4)

- Where options OPT[1-4] default values may be overridden by the user:
  
  F90 = `tau_f90.sh`
Compile-Time Environment Variables

• 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)

- optKeepFiles  
  Does not remove intermediate .pdb and .inst.* files

- optPreProcess  
  Preprocess Fortran sources 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

- 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/gfpars)

- optPdtF95Reset=""  
  Reset options for Fortran parser in PDT (f95parse/gfpars)

- optPdtCOpts=""  
  Options for C parser in PDT (cparse). Typically
  $(TAU_MPI_INCLUDE) $(TAU_INCLUDE) $(TAU_DEFS)

- optPdtCxxOpts=""  
  Options for C++ parser in PDT (cxxparse). Typically
  $(TAU_MPI_INCLUDE) $(TAU_INCLUDE) $(TAU_DEFS)
• 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 -optDetectMemoryLeaks' 

• To use an instrumentation specification file:
  % export TAU_OPTIONS='"-optTauSelectFile=select.tau -optVerbose -optPreProcess' 
  % cat select.tau
  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
### Runtime Environment Variables in TAU

<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 tau_exec –memory ./a.out)</td>
</tr>
<tr>
<td>TAU_TRACK_HEAP or TAU_TRACK_HEADROOM</td>
<td>0</td>
<td>Setting to 1 turns on tracking heap memory/headroom at routine entry &amp; exit using context events (e.g., Heap at Entry: main=&gt;foo=&gt;bar)</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_SAMPLING</td>
<td>1</td>
<td>Generates sample based profiles</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_&lt;event&gt;:&lt;subevent&gt;)</td>
</tr>
</tbody>
</table>
Usage Scenarios: Loop Level Instrumentation

- Goal: What loops account for the most time? How much?
- Flat profile with wallclock time with loop instrumentation:

Metric: GET_TIME_OF_DAY
Value: Exclusive
Units: microseconds

1729975.333  Loop: MULTIPLY_MATRICES [{matmul.f90} {31.9}-{36.14}]
443194  MPI_Recv()
81095  MAIN
49569  MPI_Bcast()
45669  Loop: MAIN [{matmul.f90} {86.9}-{106.14}]
12412  MPI_Send()
8959  Loop: INITIALIZE [{matmul.f90} {17.9}-{21.14}]
8953  Loop: INITIALIZE [{matmul.f90} {10.9}-{14.14}]
5609.2  MPI_Finalize()
2932.667  MULTIPLY_MATRICES
2577.667  Loop: MAIN [{matmul.f90} {117.9}-{128.14}]
2091.8  MPI_Barrier()
1875.667  Loop: MAIN [{matmul.f90} {112.9}-{115.14}]
1833  Loop: MAIN [{matmul.f90} {71.9}-{74.14}]
107  Loop: MAIN [{matmul.f90} {77.9}-{84.14}]
30  INITIALIZE
14.25  MPI_Comm_rank()
1  MPI_Comm_size()
Solution: Generating a loop level profile

% export TAU_MAKEFILE=<taudir>/<arch>/lib/Makefile.tau-mpi-pdt
% export TAU_OPTIONS=-optTauSelectFile=select.tau -optVerbose
% cat select.tau

BEGIN_INSTRUMENT_SECTION
loops routine="#"
END_INSTRUMENT_SECTION

% module load tau
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% mpirun -np 8 ./a.out
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.

% paraprof app.ppk
• Goal: What execution rate do my application loops get in mflops?
• Flat profile with PAPI_FP_INS and time with loop instrumentation:
% export TAU_MAKEFILE=<taudir>/<arch>/lib/Makefile.tau-papi-mpi-pdt
% export TAU_OPTIONS='-optTauSelectFile=select.tau -optVerbose'
% cat select.tau
   BEGIN_INSTRUMENT_SECTION
   loops routine="#"
   END_INSTRUMENT_SECTION

% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% export TAU_METRICS=TIME:PAPI_FP_INS
% mpirun -np 8 ./a.out
% paraprof --pack app.ppk
   Move the app.ppk file to your desktop.
% paraprof app.ppk
   Choose Options -> Show Derived Panel -> Click PAPI_FP_INS, Click "/", Click TIME, Apply, Choose new metric by double clicking.
Usage Scenarios: Compiler-based Instrumentation

- Use the compiler to automatically emit instrumentation calls in the object code instead of parsing the source code using PDT.

![Image of performance data analysis]

SC 11: Hands-on Practical Hybrid Parallel Application Performance Engineering
% export TAU_MAKEFILE=<taudir>/<arch>/lib/Makefile.tau-mpi-pdt
% export TAU_OPTIONS='–optCompInst –optQuiet'

% make CC=tau_cc.sh CXX=tau_cxx.sh F90=tau_f90.sh

NOTE: You may also use the short-hand scripts taucc, tauf90, taucxx instead of specifying TAU_OPTIONS and using the traditional tau_<cc,cxx,f90>.sh scripts. These scripts use compiler-based instrumentation by default.

% make CC=taucc CXX=taucxx F90=tauf90
% mpirun –np 8 ./a.out
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.
% paraprof app.ppk
Callpath Profile

- Generates program callgraph
% export TAU_MAKEFILE=<taudir>/<arch>/lib/Makefile.tau-mpi-pdt
% export PATH=<taudir>/<arch>/bin:$PATH
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)

% export TAU_CALLPATH=1
% export TAU_CALLPATH_DEPTH=100
(truncates all calling paths to a specified depth)

% mpirun -np 8 ./a.out
% paraprof --pack app.ppk
   Move the app.ppk file to your desktop.
% paraprof app.ppk
(Windows -> Thread -> Call Graph)
• Goal: What is the volume of inter-process communication? Along which calling path?
% export TAU_MAKEFILE=$TAU/Makefile.tau-mpi-pdt
% export PATH=<taudir>/<arch>/bin:$PATH
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% export TAU_COMM_MATRIX=1

% mpirun -np 8 ./a.out

% paraprof
(Windows -> Communication Matrix)
(Windows -> 3D Communication Matrix)
Generating a Trace File

- Goal: Identify the temporal aspect of performance. What happens in my code at a given time? When?
- Event trace visualized in Vampir/Jumpshot/Paraver
Evaluate Scalability using PerfExplorer Charts

% export TAU_MAKEFILE=$TAU/Makefile.tau-mpi-pdt
% export PATH=<taudir>/<arch>/bin:$PATH
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% export TAU_TRACE=1

% mpirun -np 8 ./a.out

Merge and convert the travefiles:
% tau_treemerge.pl
For Vampir (OTF):
% tau2otf tau.trc tau.edf app.otf; vampir app.otf

For Jumpshot (SLOG2):
% tau2slog2 tau.trc tau.edf -o app.slog2; jumpshot app.slog2

For ParaVer:
% tau_convert -paraver tau.trc tau.edf app.prv; paraver app.prv
Three Instrumentation Techniques for Wrapping External Libraries

- Pre-processor based substitution by re-defining a call (e.g., read)
  - Tool defined header file with same name <unistd.h> takes precedence
  - Header redefines a routine as a different routine using macros
  - Substitution: read() substituted by preprocessor as tau_read() at callsite

- Preloading a library at runtime
  - Library preloaded (LD_PRELOAD env var in Linux) in the address space of executing application intercepts calls from a given library
  - Tool’s wrapper library defines read(), gets address of global read() symbol (dlsym), internally calls timing calls around call to global read

- Linker based substitution
  - Wrapper library defines __wrap_read which calls __real_read and linker is passed -Wl,-wrap,read to substitute all references to read from application’s object code with the __wrap_read defined by the tool
• Pre-processor based substitution by re-defining a call
  – Compiler replaces read() with tau_read() in the body of the source code
• Advantages:
  – Simple to instrument
    • Preprocessor based replacement
    • A header file redefines the calls
    • No special linker or runtime flags required
• Disadvantages
  – Only works for C & C++ for replacing calls in the body of the code.
  – Incomplete instrumentation: fails to capture calls in uninstrumented libraries (e.g., libhdf5.a)
Issues: Linker based substitution

- Linker based substitution
  - Wrapper library defines __wrap_read which calls __real_read and linker is passed -Wl,-wrap, read

- Advantages
  - Tool can intercept all references to a given call
  - Works with static as well as dynamic executables
  - No need to recompile the application source code, just re-link the application objects and libraries with the tool wrapper library

- Disadvantages
  - Wrapping an entire library can lengthen the linker command line with multiple -Wl,-wrap,<func> arguments. It is better to store these arguments in a file and pass the file to the linker
  - Approach does not work with un-instrumented binaries
Solution: tau_gen_wrapper

• Automates creation of wrapper libraries using TAU
• Input:
  – header file (foo.h)
  – library to be wrapped (/path/to/libfoo.a)
  – technique for wrapping
    • Preprocessor based redefinition (-d)
    • Runtime preloading (-r)
    • Linker based substitution (-w: default)
  – Optional selective instrumentation file (-f select)
    • Exclude list of routines, or
    • Include list of routines
• Output:
  – wrapper library
  – optional link_options.tau file (-w), pass –optTauWrapFile=<file>
    in TAU_OPTIONS environment variable
Design of wrapper generator (tau_gen_wrapper)

- **tau_gen_wrapper** shell script:
  - Parses source of header file using static analysis tool Program Database Toolkit (PDT)
  - Invokes `tau_wrap`, a tool that generates
    - instrumented wrapper code,
    - an optional `link_options.tau` file (for linker-based substitution, -w)
    - Makefile for compiling the wrapper interposition library
  - Builds the wrapper library using `make`

- Use TAU_OPTIONS environment variable to pass location of `link_options.tau` file using
  ```bash
  % export TAU_OPTIONS=’-optVerbose
  -optTauWrapFile=<path/to/link_options.tau>’
  ```

- Use `tau_exec` `-loadlib=<wrapperlib.so>` to pass location of wrapper library for preloading based substitution
HDF5 Library Wrapping

Usage:  
```bash
tau_gen_wrapper <header> <library> [-r|-d|-w (default)] [-g groupname] [-i headerfile] [-c|-c++|-fortran] [-f <instr_spec_file>]
```

- instruments using runtime preloading (-r), or -Wl,-wrap linker (-w), redirection of header file to redefine the wrapped routine (-d)
- instrumentation specification file (select.tau)
- group (hdf5)
- tau_exec loads libhdf5_wrap.so shared library using -loadlib=<libwrap_pkg.so>
- creates the wrapper/ directory

<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>100.0</td>
<td>0.057</td>
<td>1</td>
<td>1</td>
<td>13</td>
<td>1236 .TAU Application</td>
</tr>
<tr>
<td>70.8</td>
<td>0.875</td>
<td>0.875</td>
<td>1</td>
<td>0</td>
<td>875 hid_t H5Fcreate()</td>
</tr>
<tr>
<td>9.7</td>
<td>0.12</td>
<td>0.12</td>
<td>1</td>
<td>0</td>
<td>120 herr_t H5Fclose()</td>
</tr>
<tr>
<td>6.0</td>
<td>0.074</td>
<td>0.074</td>
<td>1</td>
<td>0</td>
<td>74 hid_t H5Dcreate()</td>
</tr>
<tr>
<td>3.1</td>
<td>0.038</td>
<td>0.038</td>
<td>1</td>
<td>0</td>
<td>38 herr_t H5Dwrite()</td>
</tr>
<tr>
<td>2.6</td>
<td>0.032</td>
<td>0.032</td>
<td>1</td>
<td>0</td>
<td>32 herr_t H5Dclose()</td>
</tr>
<tr>
<td>2.1</td>
<td>0.026</td>
<td>0.026</td>
<td>1</td>
<td>0</td>
<td>26 herr_t H5check_version()</td>
</tr>
<tr>
<td>0.6</td>
<td>0.008</td>
<td>0.008</td>
<td>1</td>
<td>0</td>
<td>8 hid_t H5Screate_simple()</td>
</tr>
<tr>
<td>0.2</td>
<td>0.002</td>
<td>0.002</td>
<td>1</td>
<td>0</td>
<td>2 herr_t H5Tset_order()</td>
</tr>
<tr>
<td>0.2</td>
<td>0.002</td>
<td>0.002</td>
<td>1</td>
<td>0</td>
<td>2 hid_t H5Tcopy()</td>
</tr>
<tr>
<td>0.1</td>
<td>0.001</td>
<td>0.001</td>
<td>1</td>
<td>0</td>
<td>1 herr_t H5Sclose()</td>
</tr>
<tr>
<td>0.1</td>
<td>0.001</td>
<td>0.001</td>
<td>2</td>
<td>0</td>
<td>0 herr_t H5open()</td>
</tr>
<tr>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0 herr_t H5Tclose()</td>
</tr>
</tbody>
</table>
Using POSIX I/O wrapper library in TAU

• Setting environment variable TAU_OPTIONS=-optTrackIO links in TAU’s wrapper interposition library using linker-based substitution
• Instrumented application generates bandwidth, volume data
• Workflow:
  – % export TAU_OPTIONS='--optTrackIO --optVerbose'
  – % export TAU_MAKEFILE=/path/to/tau/x86_64/lib/Makefile.tau-mpi-pdt
  – % make CC=tau_cc.sh CXX=tau_cxx.sh F90=tau_f90.sh
  – % mpirun --np 8 ./a.out
  – % paraprof

• Get additional data regarding individual arguments by setting environment variable TAU_TRACK_IO_PARAMS=1 prior to running
• Preloading a library at runtime
  – Tool defines read(), gets address of global read() symbol
    (dlsym), internally calls timing calls around call to global read
  – tau_exec tool uses this mechanism to intercept library calls

• Advantages
  – No need to re-compile or re-link the application source code
  – Drop-in replacement library implemented using LD_PRELOAD
    environment variable under Linux, Cray CNL, IBM BG/P CNK, Solaris…

• Disadvantages
  – Only works with dynamic executables. Default compilation mode
    under Cray XE6 and IBM BG/P is to use static executables
  – Not all operating systems support preloading of dynamic shared
    objects (DSOs)
• Runtime instrumentation by pre-loading the measurement library
• Works on dynamic executables (default under Linux)
• Can substitute I/O, MPI, SHMEM, CUDA, OpenCL, and memory allocation/deallocation routines with instrumented calls
• Track interval events (e.g., time spent in write()) as well as atomic events (e.g., how much memory was allocated) in wrappers
• Accurately measure I/O and memory usage
• Preload any wrapper interposition library in the context of the executing application
Preloading a Specific TAU Measurement Library

% ./configure \(-\text{pdt}=<\text{dir}>\) -mpi \(-\text{papi}=<\text{dir}>\); make install
Creates in \(<\text{taudir}>/<\text{arch}>/\text{lib}>:
Makefile.tau-papi-mpi-pdt
shared-papi-mpi-pdt/libTAU.so

% ./configure \(-\text{pdt}=<\text{dir}>\) -mpi; make install creates
Makefile.tau-mpi-pdt
shared-mpi-pdt/libTAU.so

To explicitly choose preloading of shared-\(<\text{options}>\)/libTAU.so change:
% mpirun –np 8 ./a.out \(\) to
% mpirun –np 8 tau_exec –T \(<\text{comma}_{-}\text{separated}_{-}\text{options}>\) ./a.out

% mpirun –np 8 tau_exec –T papi,mpi,pdt ./a.out
Preloads \(<\text{taudir}>/<\text{arch}>/\text{shared}-\text{papi}-\text{mpi}-\text{pdt}/\text{libTAU}.so

% mpirun –np 8 tau_exec –T papi ./a.out
Preloads \(<\text{taudir}>/<\text{arch}>/\text{shared}-\text{papi}-\text{mpi}-\text{pdt}/\text{libTAU}.so\) by matching.

% mpirun –np 8 tau_exec –T papi,mpi,pdt \(-s\) ./a.out
Does not execute the program. Just displays the library that it will preload if executed without
the \(-s\) option.

\textbf{NOTE:} \(-\text{mpi}\) configuration is selected by default. Use \(-\text{T serial}\) for Sequential programs.
TAU Execution Command (tau_exec)

- **Uninstrumented execution**
  - % mpirun -np 8 ./a.out
- **Track MPI performance**
  - % mpirun -np 8 tau_exec ./a.out
- **Track POSIX I/O and MPI performance (MPI enabled by default)**
  - % mpirun -np 8 tau_exec -io ./a.out
- **Track memory operations**
  - % setenv TAU_TRACK_MEMORY_LEAKS 1
  - % mpirun -np 8 tau_exec -memory ./a.out
- **Use event based sampling (compile with -g)**
  - % mpirun -np 8 tau_exec -ebs ./a.out
  - Also -ebs_source=<PAPI_COUNTER> -ebs_period=<overflow_count>
- **Load wrapper interposition library**
  - % mpirun -np 8 tau_exec -loadlib=<path/libwrapper.so> ./a.out
- **Track GPGPU operations**
  - % mpirun -np 8 tau_exec -cuda ./a.out
  - % mpirun -np 8 tau_exec -opencl ./a.out