Improving Application Performance Using the TAU Performance System

Sameer Shende, John C. Linford
\{sameer, jlinford\}@paratools.com

ParaTools, Inc and University of Oregon.

April 4-5, 2013, CG1, NCAR, UCAR

Download slides from:

http://www.paratools.com/sea13
Tutorial Goals

This tutorial is an introduction to portable performance evaluation tools. You should leave here with a better understanding of…

• Concepts and steps involved in performance evaluation
• Understanding key concepts in understanding code performance
• How to collect and analyze data from hardware performance counters (PAPI)
• How to instrument your programs with TAU
• Measurement options provided by TAU
• Environment variables used for choosing metrics, generating performance data
• How to use ParaProf, TAU’s profile browser
• General familiarity with TAU use for Fortran, C++, C, and mixed language
• How to generate trace data in different formats
Outline (1)

- Introduction to the TAU Performance System
- Configure your environment, selecting TAU_MAKEFILE
- Generating a flat MPI profile
  - Automatic instrumentation with TAU compiler wrappers
  - Visualization with ParaProf
- Generating a loop-level profile
- Profiling with multiple counters
  - Computing FLOPS in each loop
- Generate and visualize a callpath profile
- Generate and visualize the communication matrix
- Compiler-based Instrumentation (when PDT isn’t enough)
Outline (2)

- Binary rewriting instrumentation
- Generating event traces
  - Visualization with Vampir
- Runtime preloading with tau_exec
- Techniques for wrapping external libraries
  - tau_gen_wrapper
  - HDF5 library wrapping
  - Using the TAU I/O and memory wrappers
Outline (3)

- Profiling Python applications
- Profiling GPU-accelerated applications
  - GPU performance measuring tools
- Performance analysis
  - Snapshots in ParaProf
- PerfExplorer
  - Runtime breakdown
  - Scalability
  - Correlation analysis
  - Performance regression testing
Outline (4)

- Multi-language debugging
  - Unwind the callstack across Python, C++, and Fortran
- Memory debugging
  - Off-by-one errors
  - Zero-byte malloc
  - Incorrect allocation alignment
- Acknowledgements
- Reference
TAU Performance System®
http://tau.uoregon.edu/

- 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
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
Profiling and Tracing

**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** shows you *when* the events take place on a timeline

- **Profiling** shows you when the events took 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;
}
```
What does TAU support?

- C/C++
- Fortran
- CUDA
- UPC
- OpenACC
- Intel MIC
- pthreads
- MinGW
- OpenMP
- LLVM
- PGI
- Cray
- Intel MIC
- OpenCL
- Python
- GPI
- Java
- MPI
- Sun
- AIX
- ARM
- NVIDIA
- Fujitsu
- BlueGene
- Windows
- Linux
- OS X
- Kepler
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
The TAU Architecture

TAU Architecture

**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
  - PerlDMF 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 total mSec</th>
<th>#Call</th>
<th>#Subrs</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>

**Interval events** show **duration**

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

```bash
% export TAU_CALLPATH_DEPTH=0
% export TAU_TRACK_HEAP=1
```
Atomic Events and Context Events

### Atomic Events

Atomic events are events that are executed in the context of a specific event and are atomic in nature. They are shown in the table below, which illustrates the time taken by each event and its associated context.

#### Example Table

<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>0.253</td>
<td>1.106</td>
<td>1</td>
<td>44</td>
<td>int main(int, char **) C</td>
</tr>
<tr>
<td>93.2</td>
<td>1.031</td>
<td>1.031</td>
<td>1</td>
<td>10</td>
<td>1031311 MPT_Init()</td>
</tr>
<tr>
<td>6.0</td>
<td>1.081</td>
<td>66.0</td>
<td>40</td>
<td>320</td>
<td>1650 void func(int, int) C</td>
</tr>
<tr>
<td>5.7</td>
<td>63.0</td>
<td>63.0</td>
<td>40</td>
<td>0</td>
<td>1588 MPI_Barrier()</td>
</tr>
<tr>
<td>0.8</td>
<td>9.0</td>
<td>9.0</td>
<td>1</td>
<td>0</td>
<td>9119 MPI_Finalize()</td>
</tr>
<tr>
<td>0.1</td>
<td>1.1</td>
<td>1.1</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>0.0</td>
<td>0.0</td>
<td>1</td>
<td>0</td>
<td>0 MPI_Comm_rank()</td>
</tr>
</tbody>
</table>

### Context Events

Context events are events that are atomic and are used to track context events in the executing context. They are shown in the table above and control the depth of executing context shown in profiles.

#### Example Code

```
% export TAU_CALLPATH_DEPTH=1
% export TAU_TRACK_HEAP=1
```
## Context Events with Callpath

### 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 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 **) 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>1885 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></td>
<td></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>44.39</td>
<td>3.091E+04</td>
<td>1.234E+04</td>
<td>Heap Memory Used (KB): Entry</td>
</tr>
<tr>
<td>1</td>
<td>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</td>
</tr>
<tr>
<td>1</td>
<td>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_rank()</td>
</tr>
<tr>
<td>1</td>
<td>2066</td>
<td>2066</td>
<td>2066</td>
<td>0</td>
<td>Heap Memory Used (KB): Entry: int main(int, char **) C =&gt; MPI_Comm_size()</td>
</tr>
<tr>
<td>1</td>
<td>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_Finalize()</td>
</tr>
<tr>
<td>1</td>
<td>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; MPI_Init()</td>
</tr>
<tr>
<td>40</td>
<td>5.036E+04</td>
<td>2069</td>
<td>3.011E+04</td>
<td>1.228E+04</td>
<td>Heap Memory Used (KB): Entry: int main(int, char **) C =&gt; void func(int, int) C</td>
</tr>
<tr>
<td>40</td>
<td>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_Barrier()</td>
</tr>
<tr>
<td>40</td>
<td>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_Bcast()</td>
</tr>
<tr>
<td>120</td>
<td>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_Recv()</td>
</tr>
<tr>
<td>120</td>
<td>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_Send()</td>
</tr>
<tr>
<td>365</td>
<td>5.139E+04</td>
<td>2065</td>
<td>3.116E+04</td>
<td>1.21E+04</td>
<td>Heap Memory Used (KB): Exit</td>
</tr>
</tbody>
</table>

% export TAU_CALLPATH_DEPTH=2
% export TAU_TRACK_HEAP=1

---

Callpath shown on context events

---

ParaTools
UNIVERSITY OF OREGON
20
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
Hands On
Configure your environment

Get workshop files from
http://www.paratools.com/seal3
% tar xvzf workshop.tgz
% module load workshop tau
% cd matmult
% make

Edit run.job to put in correct project id in -P:
% bsub run.job
Or
% bsub -Is -W 1:00 -n 8 -P <id> -q tutorial $SHELL
% pprof
% paraprof -pack foo.ppk
% paraprof foo.ppk (either remotely or locally)
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 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=$TAU/Makefile.tau-icpc-papi-mpi-pdt
% export TAU_OPTIONS=‘-optVerbose …’ (see tau_compiler.sh )
% export PATH=/glade/apps/opt/tau/2.22.2-intelpoe-intel-papi/x86_64/bin:$PATH

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

% mpi90 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)
Choosing TAU_MAKEFILE

% ls $TAU/Makefile.*
Makefile.tau
Makefile.tau-icpc-papi-mpi-pdt
Makefile.tau-icpc-papi-mpi-pdt-openmp-opari
Makefile.tau-icpc-papi-mpi-pdt-openmp-opari-scorep
Makefile.tau-icpc-papi-mpi-pdt-scorep
Makefile.tau-icpc-papi-pdt-openmp-opari
Makefile.tau-icpc-papi-pdt-openmp-opari-scorep

For an MPI+F90 application with Intel MPI, you may choose
Makefile.tau-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
% bsub -Is -W 1:00 -n 8 -P SCSG0004 -q tutorial $SHELL
% mpirun.lsf ./matrix

% paraprof
Automatic Instrumentation

• Use TAU’s compiler wrappers
  • Simply replace `CXX` with `tau_cxx.sh`, etc.
  • Automatically instruments source code, links with TAU libraries.
• Use `tau_cc.sh` for C, `tau_f90.sh` for Fortran, `tau_upc.sh` for UPC, etc.

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

app: $(OBJJS)
  $(CXX) $(LDFLAGS) $(OBJJS) -o $@
  $(LIBS)
.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) $(LDFLAGS) $(OBJJS) -o $@
  $(LIBS)
.cpp.o:
  $(CXX) $(CXXFLAGS) -c $<
```
Generating a flat profile with MPI

% module load workshop tau
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt
% make F90=tau_f90.sh
Or
% tau_f90.sh matmult.f90
% qsub -I -l select:ncpus=4; mpirun.lsf ./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.
Routine Level Profile with Score-P

How much time is spent in each application routine?

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value: Exclusive</th>
<th>Units: milliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>int main(int, char **) C =&gt; MPI_Init</td>
<td>319.665</td>
<td></td>
</tr>
<tr>
<td>MPI_Init</td>
<td>319.665</td>
<td></td>
</tr>
<tr>
<td>int main(int, char **) C =&gt; double do_work(void) C =&gt; void compute(double **, double **, double **, int, int, int) C</td>
<td>278.297</td>
<td></td>
</tr>
<tr>
<td>void compute(double **, double **, double **, int, int, int) C</td>
<td>278.297</td>
<td></td>
</tr>
<tr>
<td>int main(int, char **) C =&gt; double do_work(void) C =&gt; void compute_interchange(double **, double **, double **, int, int, int) C</td>
<td>264.864</td>
<td></td>
</tr>
<tr>
<td>void compute_interchange(double **, double **, double **, int, int, int) C</td>
<td>264.864</td>
<td></td>
</tr>
<tr>
<td>int main(int, char **) C =&gt; MPI_Finalize</td>
<td>244.177</td>
<td></td>
</tr>
<tr>
<td>MPI_Finalize</td>
<td>244.177</td>
<td></td>
</tr>
<tr>
<td>int main(int, char **) C =&gt; double do_work(void) C =&gt; void initialize(double **, int, int) C</td>
<td>3.327</td>
<td></td>
</tr>
<tr>
<td>void initialize(double **, int, int) C</td>
<td>3.327</td>
<td></td>
</tr>
<tr>
<td>int main(int, char **) C =&gt; double do_work(void) C =&gt; double **allocateMatrix(int, int) C</td>
<td>2.601</td>
<td></td>
</tr>
<tr>
<td>double **allocateMatrix(int, int) C</td>
<td>2.601</td>
<td></td>
</tr>
<tr>
<td>int main(int, char **) C =&gt; double do_work(void) C</td>
<td>1.173</td>
<td></td>
</tr>
<tr>
<td>double do_work(void) C</td>
<td>1.173</td>
<td></td>
</tr>
<tr>
<td>int main(int, char **) C</td>
<td>0.185</td>
<td></td>
</tr>
</tbody>
</table>
Generating a routine level profile

% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt-scorep

% module load tau
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% bsub -Is -W 1:00 -n 8 -P SCSG0004 -q tutorial $SHELL
% mpirun.lsf ./matmult
Execute the program and then launch paraprof or cube:
% cd scorep-<dir> ;
% paraprof profile.cubex
ParaProf 3D Profile Browser
ParaProf

http://www.paratools.com/sea13
Generating a loop level profile

```bash
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-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)

% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.

% paraprof app.ppk
```
Loop Level Instrumentation

Goal: What loops account for the most time? How much?

Flat profile with wallclock time with loop instrumentation:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET_TIME_OF_DAY</td>
<td>Exclusive</td>
<td>microseconds</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time (microseconds)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1729975.333</td>
<td>Loop: MULTIPLY_MATRICES [[matmult.f90] {31.9}–{36.14}]</td>
</tr>
<tr>
<td>443194</td>
<td>MPI_Recv()</td>
</tr>
<tr>
<td>8105</td>
<td>MAIN</td>
</tr>
<tr>
<td>49569</td>
<td>MPI_Bcast()</td>
</tr>
<tr>
<td>45669</td>
<td>Loop: MAIN [[matmult.f90] {86.9}–{106.14}]</td>
</tr>
<tr>
<td>12412</td>
<td>MPI_Send()</td>
</tr>
<tr>
<td>8959</td>
<td>Loop: INITIALIZE [[matmult.f90] {17.9}–{21.14}]</td>
</tr>
<tr>
<td>8953</td>
<td>Loop: INITIALIZE [[matmult.f90] {10.9}–{14.14}]</td>
</tr>
<tr>
<td>5609.2</td>
<td>MPI_Finalize()</td>
</tr>
<tr>
<td>2932.667</td>
<td>MULTIPLY_MATRICES</td>
</tr>
<tr>
<td>2577.667</td>
<td>Loop: MAIN [[matmult.f90] {117.9}–{128.14}]</td>
</tr>
<tr>
<td>2091.8</td>
<td>MPI_Barrier()</td>
</tr>
<tr>
<td>1875.667</td>
<td>Loop: MAIN [[matmult.f90] {112.9}–{115.14}]</td>
</tr>
<tr>
<td>1833</td>
<td>Loop: MAIN [[matmult.f90] {71.9}–{74.14}]</td>
</tr>
<tr>
<td>107</td>
<td>Loop: MAIN [[matmult.f90] {77.9}–{84.14}]</td>
</tr>
<tr>
<td>30</td>
<td>INITIALIZE</td>
</tr>
<tr>
<td>14.25</td>
<td>MPI_Comm_rank()</td>
</tr>
<tr>
<td>1</td>
<td>MPI_Comm_size()</td>
</tr>
</tbody>
</table>
Profiling with multiple counters

% module load workshop tau
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-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)
% setenv REQUEST_SUSPEND_HPC_STAT 1
% bsub -Is -W 1:00 -n 8 -P SCSG0004 -q tutorial $SHELL
% export TAU_METRICS=TIME:PAPI_FP_INS:PAPI_L1_DCM
% mpirun.lsf ./matmult
% 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.
Computing FLOPS per loop

Goal: What is the execution rate of my loops in MFLOPS?

Flat profile with PAPI_FP_INS and time with loop instrumentation:

Metric: PAPI_FP_INS / GET_TIME_OF_DAY
Value: Exclusive
Units: Derived metric shown in microseconds format
% module load workshop tau
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)

% bsub -I -q tutorial ...
% export TAU_CALLPATH=1
% export TAU_CALLPATH_DEPTH=100
(truncates all calling paths to a specified depth)
% mpirun.lsf ./a.out
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.
% paraprof app.ppk
(Windows -> Thread -> Call Graph)
ParaProf Call Graph Window
Generating Communication Matrix

% module load workshop tau
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)

% qsub -I -l ncpus=4;
% export TAU_COMM_MATRIX=1
% mpirun.lsf ./a.out

% paraprof
(Windows -> Communication Matrix)
(Windows -> 3D Communication Matrix)
Communication Matrix Display

Goal: What is the volume of inter-process communication? Along which calling path?
Compiler-based Instrumentation

- Compiler automatically emits instrumentation calls in the object code instead of parsing the source code using PDT
- To enable: export TAU_OPTIONS="-optCompInst"
- Configure TAU with "-bfd=download" for best results
Use Compiler-Based Instrumentation

% module load workshop tau
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-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.lsf./a.out
% paraprof --pack app.ppk
   Move the app.ppk file to your desktop.
% paraprof app.ppk
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
- **Dyninst:**
  \[ \% \text{tau\_run} -T [\text{tags}] \text{a.out} -o \text{a.inst} \]
- **MAQAO:**
  \[ \% \text{tau\_rewrite} -T [\text{tags}] \text{a.out} -o \text{a.inst} \]
- **Pebil:**
  \[ \% \text{tau\_pebil\_rewrite} -T [\text{tags}] \text{a.out} \ -o \text{a.inst} \]
- Execute the application to get measurement data:
  \[ \% \text{mpirun\_lsf} ./\text{a.inst} \]
Use Binary Rewriting Instrumentation

% module load workshop tau
% mpif90 -g matmult.f90 -o matmult
% tau_rewrite matmult matmult.i

Or
% tau_rewrite -T icpc,mpi,papi,pdt ./matmult -o matmult.i
% mpirun.lsf ./matmult.i
% paraprof

Score-P:
Binary Rewriting with Score-P

```bash
% mpif90 matmult.f90 -o matmult -g
% tau_rewrite -T scorep ./matmult -o ./matmult.i

% bsub -I ...
Uninstrumented:
% mpirun.lsf ./matmult

Instrumented with Score-P:

% mpirun.lsf tau_exec -T scorep -loadlib=/glade/apps/opt/unite/packages/scorep/1.1.1-intelpoe-intel-papi/lib/libscorep_mpi.so ./matmult.i

% cd scorep-<dir>; paraprof profile.cubex &
% export SCOREP_ENABLE_TRACING=1
% mpirun.lsf tau_exec -T scorep -loadlib=/glade/apps/opt/unite/packages/scorep/1.1.1-intelpoe-intel-papi/lib/libscorep_mpi.so ./matmult.i
% export PATH=/glade/apps/opt/unite/packages/vampir/8.0.1/bin/:$PATH
% cd scorep-<dir>; vampir traces.otf2 &
```
Compiler-based Instrumentation

```bash
% module load workshop tau
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-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.lsf./a.out
% paraprof --pack app.ppk
   Move the app.ppk file to your desktop.
% paraprof app.ppk
```
Generating Event Traces

% module load workshop tau
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)

% bsub -I
% export TAU_TRACE=1
% mpirun.lsf ./a.out

Merge and convert the tracefiles:
% 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
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
Stencil2D Parallel Profile / Trace in Vampir

Metric: TAUGPU_TIME
Value: Exclusive
Runtime Preloading: tau_exec

- 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 TAU Library

% ./configure --pdt=<dir> -mpi --papi=<dir>; make install
Creates in $TAU:
Makefile.tau--papi--mpi--pdt
shared--papi--mpi--pdt/libTAU.so

% ./configure --pdt=<dir> -mpi; make install creates
Makefile.tau--icpc--papi--mpi--pdt
shared--mpi--pdt/libTAU.so

To explicitly choose preloading of shared--<options>/libTAU.so change:
% mpirun.lsf./a.out to
% mpirun.lsf tau_exec --T <comma_separated_options> ./a.out

% mpirun.lsf tau_exec --T papi,mpi,pdt ./a.out
Preloads $TAU/shared--papi--mpi--pdt/libTAU.so
% mpirun.lsf tau_exec --T papi ./a.out
Preloads $TAU/shared--papi--mpi--pdt/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.
NOTE: --mpi configuration is selected by default. Use --T serial for Sequential programs.
TAU Execution Command (tau_exec)

Uninstrumented execution

• % mpirun.lsf ./a.out

Track MPI performance

• % mpirun.lsf tau_exec ./a.out

Track POSIX I/O and MPI performance (MPI enabled by default)

• % mpirun.lsf tau_exec –io ./a.out

Track memory operations

• % export TAU_TRACK_MEMORY_LEAKS=1
• % mpirun –np 8 tau_exec –memory_debug ./a.out (bounds check)

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 –cupti ./a.out
• % mpirun –np 8 tau_exec –opencl ./a.out
Tags in tau_exec and other tools

% cd $TAU; ls Makefile.*
Makefile.tau-icpc-papi-mpi-pdt

% qsub -I -l select:ncpus=4; mpirun.lsf./matrix
% tau_exec -T icpc,mpi,pdt ./a.out

Chooses Makefile.tau-icpc-mpi,pdt and associated libraries.

% tau_exec -T serial,pdt ./a.out
Chooses Makefile.tau-pdt or the shortest Makefile name without -mpi.

-T <list_of_tags> is used in several TAU tools:
- tau_run
- tau_rewrite
- tau_exec
- tau_gen_wrapper
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
Issues: Preprocessor based substitution

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
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

```
% export TAU_OPTIONS='-optTauWrapFile=<path/to/link_options.tau> -optVerbose'
```

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

- TAU source analyzer
- Application source
- Parsed program
- tau_wrap
- Instrumented source
- Instrumentation specification file
HDF5 Library Wrapping

```
[sameer@zorak]$ tau_gen_wrapper hdf5.h /usr/lib/libhdf5.a -f select.tau
```

Usage: `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

NODE 0;CONTEXT 0;THREAD 0:

<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></td>
<td>msec</td>
<td>total msec</td>
<td></td>
<td>usec/call</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>-----------</td>
<td>-----------</td>
<td>-------</td>
<td>---------</td>
<td></td>
</tr>
<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</td>
</tr>
</tbody>
</table>

H5check_version()
Using POSIX I/O wrapper library

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=$TAU/Makefile.tau-icpc-papi-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 wrapper library

Preloading a library at runtime

- Tool defines read(), gets address of global read() symbol (dlsym), internally calls timing calls around call to global read
- $\text{tau_exec}$ 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 $\text{LD_PRELOAD}$ 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)
Profiling Python applications

• Create a top-level Python wrapper
• Launch with `tau_exec -T python ...`

```python
#!/bin/env python
import tau
import sys

def OurMain():
    try:
        import <your_application>
    except SystemExit:
        # Intercept the exit call so Tau writes profiles
        print 'SystemExit intercepted by wrapper'

tau.run('OurMain()')

% tau_exec -T python wrapper.py
```
Profiling Python applications
Profiling GPGPU Executions

- GPGPU compilers (e.g. CAPS Compilers and PGI) can automatically generate GPGPU code using manual annotation of loop-level constructs and routines
- The loops (and routines for HMPP) are transferred automatically to the GPGPU
- TAU intercepts the runtime library routines and examines the arguments
- Shows events as seen from the host
- Profiles and traces GPGPU execution
Host (CPU) - GPU Scenarios

**Single GPU**

- Host (CPU): Open device → Move data → Launch kernel(s) → Wait → Move data
- GPU: Run kernel(s)

- Time: implemented as asynchronous calls

**Multi-stream**

- Host (CPU): Open device → Move data → Launch kernel(s) → Wait → Move data
- GPU: Run kernel(s) for Stream 1
- GPU: Run kernel(s) for Stream 2

**Multi-CPU, Multi-GPU**

- Thread (CPU 1): Open device → Move data → Launch kernel(s) → Wait → Move data
- GPU 1: Run kernel(s)
- GPU k: Run kernel(s)
- Time

Paratools

UNIVERSITY OF OREGON
Host-GPU Measurement – Callback Method

• GPU driver libraries provide callbacks for certain routines and captures measurements
• Measurement tool registers the callbacks and processes performance data
• Application code is not modified
Method Support and Implementation

Synchronous method
• Place instrumentation appropriately around GPU calls (kernel launch, library routine, …)
• Wrap (synchronous) library with performance tool

Event queue method
• Utilize CUDA and OpenCL event support
• Again, need instrumentation to create and insert events in the streams with kernel launch and process events
• Can be implemented with driver library wrapping

Callback method
• Utilize language-level callback support in OpenCL
• Utilize NVIDIA CUDA Performance Tool Interface (CUPTI)
• Need to appropriately register callbacks
GPU Performance Measurement Tools

Support the Host-GPU performance perspective
Provide integration with existing measurement system to facilitate tool use
Utilize support in GPU driver library and device Tools

- TAU performance system
- Vampir
- PAPI
- NVIDIA CUPTI
GPU Performance Tool Interoperability

- CUDA
- OpenCL
- CUPTI
- TAU
- PAPI
- VampirTrace
- ParaProf
- Vampir

Event queue
Callback
NVIDIA is developing CUPTI to enable the creation of profiling and tracing tools

Callback API

• Interject tool code at the entry and exist to each CUDA runtime and driver API call

Counter API

• Query, configure, start, stop, and read the counters on CUDA-enabled devices

CUPTI is delivered as a dynamic library

CUPTI is released with CUDA 4.0+
TAU for Heterogeneous Measurement

Multiple performance perspectives
Integrate Host-GPU support in TAU measurement framework
  • Enable use of each measurement approach
  • Include use of PAPI and CUPTI
  • Provide profiling and tracing support

Tutorial
  • Use TAU library wrapping of libraries
  • Use `tau_exec` to work with binaries
    % ./a.out  (uninstrumented)
    % tau_exec -T serial,cupti --cupti ./a.out
    % paraprof
Example: SDK simpleMultiGPU

Demonstration of multiple GPU device use

main → solverThread → reduceKernel

One Keeneland node with three GPUs

Performance profile for:

• One main thread
• Three solverThread threads
• Three reduceKernel “threads”
simpleMultiGPU Profile

Identified a known overhead in GPU context creation

Overall profile

Comparison profile

Metric: TIME
Value: Exclusive
Units: milliseconds

19450
18744
cutEndThread {{multithreading.cpp} {55.0}}
cudaError_t cudaMalloc(void **, size_t) C
cudaError_t cudaSetDevice(int) C
main {{simpleMultiGPU.cpp} {105.0}}
TAU is able to associate callsite context information with kernel launch so that different kernel calls can be distinguished.

Each kernel (ifft1D_512, fft1D_512 and chk1D_512) is broken down by callsite, either during the single precession or double precession step.
Example: SHOC Stencil2D

Compute 2D, 9-point stencil
- Multiple GPUs using MPI
- CUDA and OpenCL versions

One Keeneland node with 3 GPUs
Eight Keeneland nodes with 24 GPUs

Performance profile and trace
- Application events
- Communication events
- Kernel execution
Stencil2D Parallel Profile
Performance Analysis

Profile Data Management (PerfDMF)
- profile translators
- Metadata (XML)
- profile database

Trace Data Management
- trace translators
- trace storage

Profile Analysis (ParaProf)

Profile Data Mining (PerfExplorer)

Trace Visualizers
- Vampir
- JumpShot
- Paraver

Trace Analyzers
- Expert
- ProfileGen
- Vampir Server
Performance Analysis

- Parallel profile analysis (ParaProf)
  - Java-based analysis and visualization tool
  - Support for large-scale parallel profiles
- Parallel trace analysis
  - Translation to ParaVer, SLOG-2, OTF formats
  - Integration with Vampir / Vampir Pro (TU Dresden)
  - Profile generation from trace data
- Performance data management framework (PerfDMF)
  - Online parallel analysis and visualization
  - Integration with CUBE browser (Scalasca, FZJ)
ParaProf Profile Comparison Window

<table>
<thead>
<tr>
<th>Metric: TIME</th>
<th>Value: Exclusive</th>
<th>Units: seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.609</td>
<td>1.655 (46.13%)</td>
<td>mm_f10.ppp - Mean</td>
</tr>
<tr>
<td></td>
<td>1.656</td>
<td>mm_f10_03.ppp - Mean</td>
</tr>
</tbody>
</table>

void compute(double **, double **, double **, int, int, int) C [matmult.c] {46,1}--{61,1}

void compute_interchange(double **, double **, double **, int, int, int) C [matmult.c] {63,1}--{78,1}

0.057 (110.53%)

0.051  

0.01 (94.129%)

0.009  

0.003 (34.613%)

0.002  

2.3E-4  

2.7E-4 (120.638%)

8.7E-5  

8.3E-5 (95.677%)
Profile Snapshots in ParaProf

- Profile snapshots are profiles recorded at runtime
- Shows performance profile dynamics (all types allowed)
Profile Snapshot Views

Percentage breakdown

Only show main loop
Snapshot Replay in ParaProf

All windows dynamically update
ParaProf Metadata Window

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUJITSU Coords</td>
<td>(3,1,0)</td>
</tr>
<tr>
<td>FUJITSU Dimension</td>
<td>3</td>
</tr>
<tr>
<td>FUJITSU Size</td>
<td>(6,6,6)</td>
</tr>
<tr>
<td>File Type Index</td>
<td>0</td>
</tr>
<tr>
<td>File Type Name</td>
<td>ParaProf Packed Profile</td>
</tr>
<tr>
<td>Hostname</td>
<td>e09t14226</td>
</tr>
<tr>
<td>Local Time</td>
<td>2012-11-12T02:14:16+09:00</td>
</tr>
<tr>
<td>MPI Processor Name</td>
<td>e09t14226</td>
</tr>
<tr>
<td>Memory Size</td>
<td>32836968 kB</td>
</tr>
<tr>
<td>Node Name</td>
<td>e09t14226</td>
</tr>
<tr>
<td>OS Machine</td>
<td>s64fx</td>
</tr>
<tr>
<td>OS Name</td>
<td>Linux</td>
</tr>
<tr>
<td>OS Release</td>
<td>2.6.25.8</td>
</tr>
<tr>
<td>OS Version</td>
<td>#1 SMP Tue Sep 11 11:04:02 JST 2...</td>
</tr>
<tr>
<td>Starting Timestamp</td>
<td>1352654056461761</td>
</tr>
<tr>
<td>TAU Architecture</td>
<td>sparc64fx</td>
</tr>
</tbody>
</table>

% pjsub -interact -L node="6x6x6"
% mpirun.lsf -n 216 ./a.out
PerfExplorer – Runtime Breakdown

Total Runtime Breakdown for S3D (Jaguar, ORNL): Harness Scaling Study:
GET_TIME_OF_DAY

WRITE_SAVEFILE

MPI_Waitall

Number of Processors

Percentage of Total Runtime
Evaluate Scalability
Runtime Breakdown
PerfExplorer – Relative Comparisons

Total execution time
Timesteps per second
Relative efficiency
Relative efficiency per event
Relative speedup
Relative speedup per event
Group fraction of total
Runtime breakdown
Correlate events with total runtime
Relative efficiency per phase
Relative speedup per phase
Distribution visualizations

ParaTools
PerfExplorer – Correlation Analysis

Strong negative linear correlation between
CALC_CUT_BLOCK_CONTRIBUTION S
and MPI_Barrier
PerfExplorer – Correlation Analysis

-0.995 indicates strong, negative relationship. As CALC_CUT_BLOCK_CONTRIBUTIONS() increases in execution time, MPI_Barrier() decreases.
PerfExplorer – Cluster Analysis
PerfExplorer – Cluster Analysis

Four significant events automatically selected
Clusters and correlations are visible
PerfExplorer – Performance Regression
Evaluate Scalability

Goal: How does my application scale? What bottlenecks at what CPU counts?
Load profiles in PerfDMF database and examine with PerfExplorer
Usage Scenarios: Evaluate Scalability
TAUdb: Framework for Managing Performance Data
Evaluate Scalability using PerfExplorer Charts

% module load workshop tau
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% qsub run1p.job
% paraprof --pack 1p.ppk
% qsub run2p.job ...
% paraprof --pack 2p.ppk ... and so on.
On your client:
% taudb_configure -create-default
% perfexplorer_configure
(Enter, Yes to load jars, schema, defaults)
% paraprof
(load each trial: DB -> Add Trial -> Type (Paraprof Packed Profile) -> OK, OR use taudb_configure on the commandline)
% perfexplorer
(Charts -> Speedup)
Multi-language Application Debugging

```bash
% module load workshop tau
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt
% export TAU_OPTIONS='--optMemDbg --optVerbose'
% make F90=tau_f90.sh CC=tau_cc.sh CXX=tau_cxx.sh
% bsub -Is -W 1:00 -n 8 -P SCSG0004 -q tutorial $SHELL

% export TAU_MEMDBG_PROTECT_ABOVE=1
% export TAU_MEMDBG_PROTECT_BELOW=1
% export TAU_MEMDBG_PROTECT_FREE=1
% mpirun.lsf ./matmult
% paraprof
```
## Multi-language Application Debugging

![TAU: ParaProf Manager](image)

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>debug.ppk</td>
</tr>
<tr>
<td>Application ID</td>
<td>0</td>
</tr>
<tr>
<td>Experiment ID</td>
<td>0</td>
</tr>
<tr>
<td>Trial ID</td>
<td>0</td>
</tr>
<tr>
<td>BACKTRACE(1)</td>
<td>/glade/u/home/sshende/pkgs/workshop/debug/foo.c:17</td>
</tr>
<tr>
<td>BACKTRACE(2)</td>
<td>/glade/u/home/sshende/pkgs/workshop/debug/foo.c:24</td>
</tr>
<tr>
<td>BACKTRACE(3)</td>
<td>/glade/u/home/sshende/pkgs/workshop/debug/foo.c:31</td>
</tr>
<tr>
<td>BACKTRACE(4)</td>
<td>/lib64/libc-2.12.so</td>
</tr>
<tr>
<td>BACKTRACE(5)</td>
<td>/glade/u/home/sshende/pkgs/workshop/debug/foo.tau rewritten</td>
</tr>
<tr>
<td>CPU Cores</td>
<td>8</td>
</tr>
<tr>
<td>CPU MHz</td>
<td>2600.00</td>
</tr>
<tr>
<td>CPU Type</td>
<td>Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz</td>
</tr>
<tr>
<td>CPU Vendor</td>
<td>GenuineIntel</td>
</tr>
<tr>
<td>CWD</td>
<td>/glade/u/home/sshende/pkgs/workshop/debug</td>
</tr>
<tr>
<td>Cache Size</td>
<td>20480 KB</td>
</tr>
<tr>
<td>Command Line</td>
<td>/glade/u/home/sshende/pkgs/workshop/debug/foo.tau rewritten</td>
</tr>
<tr>
<td>Executable</td>
<td>/glade/u/home/sshende/pkgs/workshop/debug/foo.tau rewritten</td>
</tr>
<tr>
<td>File Type Index</td>
<td>0</td>
</tr>
<tr>
<td>File Type Name</td>
<td>ParaProf Packed Profile</td>
</tr>
<tr>
<td>Hostname</td>
<td>ys0101</td>
</tr>
<tr>
<td>Local Time</td>
<td>2013-03-31T17:13:40-06:00</td>
</tr>
<tr>
<td>MPI Processor Name</td>
<td>ys0101</td>
</tr>
<tr>
<td>Memory Size</td>
<td>52988340 kB</td>
</tr>
<tr>
<td>Node Name</td>
<td>ys0101</td>
</tr>
<tr>
<td>OS Machine</td>
<td>x86_64</td>
</tr>
<tr>
<td>OS Name</td>
<td>Linux</td>
</tr>
<tr>
<td>OS Release</td>
<td>2.6.32-220.13.1.66.x86_64</td>
</tr>
<tr>
<td>OS Version</td>
<td>#1 SMP Thu Mar 29 11:46:40 EDT 2012</td>
</tr>
<tr>
<td>SIGNAL</td>
<td>Segmentation fault</td>
</tr>
<tr>
<td>Starting Timestamp</td>
<td>1364771620575991</td>
</tr>
<tr>
<td>TAU Architecture</td>
<td>x86_64</td>
</tr>
</tbody>
</table>
| TAU Config                    | -lowrmm -mplnc=-opt/hpc/pe1209/mpich2/intel/include64 -mpilib=/opt/hpc/pe1209/mpich2/intel/lib64 -cc=icc -c++ ...
| TAU Makefile                  | /glade/u/home/sshende/pkgs/tau-2.22.2/x86_64/lib/Makfile.tau-iccpc-mpi-pdt |
| TAU Version                   | 2.22.2                                                              |

Location of segmentation violation

```c
#include <stdio.h>
#include <stdlib.h>

struct node {
    int id;
    struct node *next;
};

int bar(int x) {
    int y;
    struct node *t = (struct node *)malloc(sizeof(struct node));
    t->next = NULL;
    t->id = x;
    printf("t->id = %d\n", t->id);
    y = t->next->id;
    printf("y = %d\n", y);
    return x;
}

int foo(int x) {
    printf("foo: x = %d\n", x);
    bar(x);
    return x;
}

int main(int argc, char **argv) {
    int ret;
    MPI_Init(&argc, &argv);
    ret = foo(29);
    MPI_Finalize();
    return ret;
}
```
Memory Leak Detection

```
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt
% export PATH=$TAU/../bin:$PATH
% export TAU_OPTIONS='--optMemDbg --optVerbose'
% make F90=tau_f90.sh CC=tau_cc.sh CXX=tau_cxx.sh
% bsub -Is -W 1:00 -n 8 -P SCSG0004 -q tutorial $SHELL

% export TAU_TRACK_MEMORY_LEAKS=1
% mpirun.lsf ./matmult
% paraprof
```
Multi-language Memory Leak Detection
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

And a special thanks to UCAR!
Download TAU from U. Oregon

http://tau.uoregon.edu

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

Free download, open source, BSD license
Reference
Using TAU on Yellowstone

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

• **Configuration of TAU:**
  - `wget http://tau.uoregon.edu/tau.tgz`
  - `./configure --pdt=<dir> -c++=icpc --c=icc --fortran=intel --prefix=<dir> -papi=<dir>`
  - `make install`

• **Compiling:**
  - `export TAU_MAKEFILE=<taudir>/x86_64/lib/Makefile.tau-icpc-papi-mpi-pdt`
  - `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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>information, setting to 1 generates flat profile and context events have</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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>
</tbody>
</table>
| TAU_METRICS                          | TIME    | Setting to a comma separated list generates other metrics. (e.g., TIME:P_VIRTUAL_TIME:PAPI_FP_INS:PAPI_NATIVE_<event>:<subevent>)}
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= ‘-optComplInst -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_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