CIS 631
Parallel Processing

Parallel Performance Tools: TAU
http://tau.uoregon.edu/tau.ppt

Allen D. Malony and Sameer Shende
{malony, sameer}@cs.uoregon.edu

Department of Computer and Information Science
University of Oregon
TAU Performance System

- [ ] http://tau.uoregon.edu/
- [ ] Multi-level performance instrumentation
  - [ ] Multi-language automatic source instrumentation
- [ ] Flexible and configurable performance measurement
- [ ] Widely-ported parallel performance profiling system
  - [ ] Computer system architectures and operating systems
  - [ ] Different programming languages and compilers
- [ ] Support for multiple parallel programming paradigms
  - [ ] Multi-threading, message passing, mixed-mode, hybrid
- [ ] Integration in complex software, systems, applications
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
- TAU uses a package called PDT for automatic instrumentation of the source code
- Profiling and tracing can measure time as well as hardware performance counters from your CPU
- TAU can automatically instrument your source code (routines, loops, I/O, memory, phases, etc.)
- TAU runs on all 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 need to set a couple of environment variables and substitute the name of your compiler with a TAU shell script
Inclusive and Exclusive Profiles

- Performance with respect to code regions
- Exclusive measurements for region only
- Inclusive measurements includes child regions
TAU Parallel Performance System Goals

- Portable (open source) parallel performance system
  - Computer system architectures and operating systems
  - Different programming languages and compilers
- Multi-level, multi-language performance instrumentation
- Flexible and configurable performance measurement
- Support for multiple parallel programming paradigms
  - Multi-threading, message passing, mixed-mode, hybrid, object oriented (generic), component-based
- Support for performance mapping
- Integration of leading performance technology
- Scalable (very large) parallel performance analysis
TAU Performance System Components

**TAU Architecture**
- Instrumentation
  - event selection
  - source code
  - object code
  - library wrapper
  - binary code
  - virtual machine
- Measurement
  - event identifier
  - event selection
  - measurement
- Profiling
  - statistics
  - atomic profiles
  - trace profiles
  - event logging
  - profile sampling
  - event filtering
  - event mapping
- Tracing
  - trace saving
  - trace buffering
  - trace recording
  - trace analysis
- Performance data sources
  - OS and runtime system modules
  - system counters
  - kernel counters
  - hardware counters

**Program Analysis**
- Application/Library
  - C/C++ programs
  - Fortran programs
- Program Database
  - DUCTAPE

**Performance Data Mining**
- GUI
- Scripting interface
- Data Components
  - Data Mining
  - Metadatas
  - Performance Data
- Analysis Components
  - Statistics Analysis
  - Data Mining
  - Performance Analysis

**PerfExplorer**
- Process Control
- Analysis Interface

**TAUoverSupermon**
- Monitoring
- Analysis

---

Lecture 14

CIS 631 - Parallel Processing
TAU Performance System Architecture

Instrumentation

Event selection

Measurement

Event creation and management

Profiling

Statistics

Atomic profiles

Entry/exit profiles

Mapping (callpath)

Profile I/O

Sampling profiles

Tracing

Trace buffering

Record creation

Timestamp generation

Trace filtering

Performance data sources

Timing

Hardware counters

System counters

... OS and runtime system modules

Threading

Interrupts

Runtime system

...
**TAU Performance System Architecture**

1. **Instrumentation**
   - Event selection
   - Information

2. **Measurement**
   - Profiles
   - Traces

3. **Analysis**
   - Profile Data Management (PerfDMF)
     - Profile translators
     - Metadata (XML)
     - Profile database
   - Trace Data Management
     - Trace translators
     - Trace storage

4. **Profile Analysis (ParaProf)**
   - Profile data mining

5. **Profile Data Mining (PerfExplorer)**
   - Visualizers
     - Vampir
     - JumpShot
     - ProfileGen
   - Analyzers
     - Expert
     - Paraver
     - Vampir Server

---

**Lecture 14**

**CIS 631 - Parallel Processing**
**Program Database Toolkit (PDT)**

- **Application / Library**
  - C / C++ parser
  - Fortran parser F77/90/95

- **C / C++ IL analyzer**
  - IL

- **Program Database Files**
  - DUCTAPE

Program documentation

Application component glue

C++ / F90/95 interoperability

Automatic source instrumentation
Automatic Source-Level Instrumentation in TAU using Program Database Toolkit (PDT)
Steps of Performance Evaluation

- Collect basic routine-level timing profile to determine where most time is being spent
- Collect routine-level hardware counter data to determine types of performance problems
- Collect callpath profiles to determine sequence of events causing performance problems
- Conduct finer-grained profiling and/or tracing to pinpoint performance bottlenecks
  - Loop-level profiling with hardware counters
  - Tracing of communication operations
Using TAU: A brief Introduction

- TAU supports several measurement options (profiling, tracing, profiling with hardware counters, etc.)
- Each measurement configuration of TAU corresponds to a unique stub makefile and library that is generated when you configure it
- To instrument source code using PDT
  - Choose an appropriate TAU stub makefile in <arch>/lib:
    % source /usr/local/packages/tau.bashrc
    % export TAU_MAKEFILE=/usr/local/packages/tau_latest/x86_64/lib/Makefile.tau-mpi-pdt
    % export TAU_OPTIONS=‘-optVerbose …’ (see tau_compiler.sh -help)
    And use tau_f90.sh, tau_cxx.sh or tau_cc.sh as Fortran, C++ or C compilers:
      % mpif90 foo.f90
      changes to
      % tau_f90.sh foo.f90
- Execute application and analyze performance data:
  % pprof  (for text based profile display)
  % paraprof  (for GUI)
TAU Measurement Configuration

% cd /usr/local/packages/tau_latest/x86_64/lib; ls Makefile.*
Makefile.tau-pdt
Makefile.tau mpi-pdt
Makefile.tau pthread-pdt
Makefile.tau papi mpi-pdt
Makefile.tau papi pthread-pdt
Makefile.tau mpi papi-pdt
Makefile.tau mpich2intel-icpc-papi-mpi-pdt
Makefile.tau intelmpi-papi-mpi-pdt
Makefile.tau mpi-pdt-vampirtrace-trace

☐ For an MPI+F90 application, you may want to start with:
Makefile.tau mpi-pdt
  ○ Supports MPI instrumentation & PDT for automatic source instrumentation
  ○ % export TAU_MAKEFILE=
     /usr/local/packages/tau latest/x86 64/lib/Makefile.tau mpi-pdt
  ○ % tau f90.sh matrix.f90 -o matrix
Usage Scenarios: Routine Level Profile

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

  Metric: P_VIRTUAL_TIME
  Value: Exclusive
  Units: seconds

  - LEQ_IKSWEEP: 9647.318
  - LEQ_BICGS: 4357.213
  - LEQ_MATVECT: 2669.887
  - SOLVE_SPECIES_EQ: 1777.752
  - SOLVE_LIN_EQ: 1417.986
  - PHYSICAL_PROP: 1028.448
  - RRATES: 783.402
  - LEQ_MSOLVE: 682.376
  - INIT_AB_M: 530.858
  - CALC_MASS_FLUX_SPHR: 463.788
  - INIT_MU_S: 446.025
  - CALC_RESID_S: 421.747
  - SOLVE_ENERGY_EQ: 381.363
  - SOURCE_Phi: 371.199
  - DRAG_GS: 258.829
Solution: Generating a flat profile with MPI

% export TAU_MAKEFILE=/usr/local/packages/tau_latest/x86_64/lib/Makefile.tau-mpi-pdt
% set path=(/usr/local/packages/tau_latest/x86_64/bin $path)
OR
% source /usr/local/packages/tau.bashrc
% tau_f90.sh matmult.f90 -o matmult
(Or edit Makefile and change F90=tau_f90.sh)

% mpirun -np 4 ./matmult
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.

% paraprof app.ppk
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

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

% export TAU_MAKEFILE=/usr/local/packages/tau_latest/x86_64
   /lib/Makefile.tau-mpi-pdt
% export TAU_OPTIONS='--optTauSelectFile=select.tau --optVerbose'
% cat select.tau
  BEGIN_INSTRUMENT_SECTION
  loops routine="#"
  END_INSTRUMENT_SECTION

% set path=($(/usr/local/packages/tau_latest/x86_64/bin $path))
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% mpirun -np 4 ./a.out
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.

% paraprof app.ppk
Goal: Easily generate routine level performance data using the compiler instead of PDT for parsing the source code
Use Compiler-Based Instrumentation

% export TAU_MAKEFILE=/usr/local/packages/tau_latest/x86_64
   /lib/Makefile.tau-mpi
% export TAU_OPTIONS='--optCompInst --optVerbose'
% % set path=(/usr/local/packages/tau_latest/x86_64/bin $path)
% make F90=tau_f90.sh
   (Or edit Makefile and change F90=tau_f90.sh)

% mpirun -np 4 ./a.out
% paraprof --pack app.ppk
   Move the app.ppk file to your desktop.
% paraprof app.ppk
Usage Scenarios: Calculate mflops in Loops

- Goal: What MFlops am I getting in all loops?
- Flat profile with PAPI_FP_INS/OPS and time with loop instrumentation:

Metric: PAPI_FP_INS / GET_TIME_OF_DAY
Value: Exclusive
Units: Derived metric shown in microseconds format

- 770.699
- 223.39
- 223.24
- 171.855
- 170.862
- 122.96
- 37.549
- 21.367
- 13.795
- 8.935
- 1.131
- 0.794
- 0.647
- 0.355
- 0.171
- 0.115
- 0.023
Generate a PAPI profile with 2 or more counters

% export TAU_MAKEFILE=/usr/local/packages/tau_latest/x86_64/lib/Makefile.tau-papi-mpi-pdt
% export TAU_OPTIONS=`-optTauSelectFile=select.tau -optVerbose`
% cat select.tau
BEGIN_INSTRUMENT_SECTION
  loops routine="#"
END_INSTRUMENT_SECTION

% set path=($path /usr/local/packages/tau_latest/x86_64/bin)
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% export TAU_METRICS=TIME:PAPI_FP_INS:PAPI_L1_DCM
% mpirun -np 4 .a.out
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.
% paraprof app.ppk
  Choose Options -> Show Derived Panel -> “PAPI_FP_INS”, click “/”, “TIME”, click “Apply” choose.
Derived Metrics in ParaProf
Comparing Effects of Multi-Core Processors

AORSA2D
- magnetized plasma simulation
- Automatic loop level instrumentation
- Blue is single node
- Red is dual core
- Cray XT3 (4K cores)
ParaProf: Mflops Sorted by Exclusive Time

low mflops?
Goal: Who calls my MPI_Barrier()? Where?

Callpath profile for a given callpath depth:
Callpath Profile

- Generates program callgraph

![Callpath Profile Diagram]
Generate a Callpath Profile

% export TAU_MAKEFILE=/usr/local/packages/tau_latest/x86_64/lib/Makefile.tau-mpi-pdt
% set path=/usr/local/packages/tau_latest/x86_64/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

% mpirun -np 4 ./a.out
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.
% paraprof app.ppk
(Windows -> Thread -> Call Graph)
**Usage Scenario: Detect Memory Leaks**

### TAU: ParaProf - Mean Context Events - `mem.ppk`

<table>
<thead>
<tr>
<th>Name</th>
<th>NumSamples</th>
<th>MaxValue</th>
<th>MinValue</th>
<th>MeanValue</th>
<th>Std Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MAIN</strong> [[matrix.f90] {141,7}→{146,22}]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEMORY LEAK! malloc size &lt;file=matrix.f90, variable=C, line=11&gt;</td>
<td>1</td>
<td>8,000,000</td>
<td>8,000,000</td>
<td>8,000,000</td>
<td>0</td>
</tr>
<tr>
<td>malloc size &lt;file=matrix.f90, variable=A, line=11&gt;</td>
<td>1</td>
<td>8,000,000</td>
<td>8,000,000</td>
<td>8,000,000</td>
<td>0</td>
</tr>
<tr>
<td>malloc size &lt;file=matrix.f90, variable=B, line=11&gt;</td>
<td>1</td>
<td>8,000,000</td>
<td>8,000,000</td>
<td>8,000,000</td>
<td>0</td>
</tr>
<tr>
<td>malloc size &lt;file=matrix.f90, variable=C, line=11&gt;</td>
<td>1</td>
<td>8,000,000</td>
<td>8,000,000</td>
<td>8,000,000</td>
<td>0</td>
</tr>
<tr>
<td><strong>MATRICES::DEALLOCATE_MATRICES</strong> [[matrix.f90] {14,7}→{17,40}]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>free size &lt;file=matrix.f90, variable=A, line=15&gt;</td>
<td>1</td>
<td>8,000,000</td>
<td>8,000,000</td>
<td>8,000,000</td>
<td>0</td>
</tr>
<tr>
<td>free size &lt;file=matrix.f90, variable=B, line=15&gt;</td>
<td>1</td>
<td>8,000,000</td>
<td>8,000,000</td>
<td>8,000,000</td>
<td>0</td>
</tr>
</tbody>
</table>

### User Event Window: `mem.ppk`

Name: MEMORY LEAK! malloc size <file=matrix.f90, variable=C, line=11> : MAIN [[matrix.f90] {141,7}→{146,22}] → MATRICES::ALLOCATE_MATRICES [[matrix.f90] {10,7}→{13,38}]

Value Type: Max Value

![Graph showing memory usage]

---

**Lecture 14**

---

**CIS 631 - Parallel Processing**
Detect Memory Leaks

% export TAU_MAKEFILE=/usr/local/packages/tau_latest/x86_64/lib/Makefile.tau-mpi-pdt
% export TAU_OPTIONS='-optDetectMemoryLeaks -optVerbose'
% set path=/usr/local/packages/tau_latest/x86_64/bin $path
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% export TAU_CALLPATH_DEPTH=100

% mpirun -np 4 ./a.out
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.
% paraprof app.ppk
(Windows -> Thread -> Context Event Window -> Select thread -> select...
  expand tree)
(Windows -> Thread -> User Event Bar Chart -> right click LEAK
  -> Show User Event Bar Chart)
Usage Scenarios: Instrument a Python program

- Goal: Generate a flat profile for a Python program
Usage Scenarios: Instrument a Python program

Original code:

% cat foo.py
#!/usr/bin/env python
import numpy
ra=numpy.random
la=numpy.linalg

size=2000
a=ra.standard_normal((size,size))
b=ra.standard_normal((size,size))
c=la.linalg.dot(a,b)
print c

Create a wrapper:

% cat wrapper.py
#!/usr/bin/env python

# setenv PYTHONPATH $PET_HOME/pkgs/tau-2.17.3/ppc64/lib/bindings-gnu-python-pdt

import tau

def OurMain():
    import foo

    tau.run('OurMain()')
Generate a Python Profile

% export TAU_MAKEFILE=/usr/local/packages/tau_latest/ibm64
   /lib/Makefile.tau-python-pdt
% set path=/usr/local/packages/tau_latest/ibm64/bin $path
% cat wrapper.py
   import tau
   def OurMain():
       import foo
       tau.run(‘OurMain()’)  
Uninstrumented:
% ./foo.py
Instrumented:
% export PYTHONPATH= <taudir>/ibm64/<lib>/bindings-python-pdt
   (same options string as TAU_MAKEFILE)
% export LD_LIBRARY_PATH=<taudir>/x86_64/lib/bindings-python-pdt:
   $LD_LIBRARY_PATH
% ./wrapper.py

Wrapper invokes foo and generates performance data
% pprof/paraprof
Usage Scenarios: Mixed Python+F90+C+pyMPI

- **Goal:** Generate multi-level instrumentation for Python+MPI+C+F90+C++ ...
Generate a Multi-Language Profile w/ Python

% export TAU_MAKEFILE=/usr/local/packages/tau_latest/x86_64
   /lib/Makefile.tau-python-mpi-pdt
% set path=('/usr/local/packages/tau_latest/x86_64/bin $path')
% export TAU_OPTIONS='-optShared -optVerbose…'
(Python needs shared object based TAU library)
% make F90=tau_f90.sh CXX=tau_cxx.sh CC=tau_cc.sh (build libs, pyMPI w/TAU)
% cat wrapper.py
   import tau
   def OurMain():
       import App
       tau.run('OurMain()')

Uninstrumented:
% mpirun -np 4 pyMPI ./App.py
Instrumented:
% export PYTHONPATH= <taudir>/x86_64/<lib>/bindings-python-mpi-pdt
(same options string as TAU_MAKEFILE)
% export LD_LIBRARY_PATH=<taudir>/x86_64/lib/bindings-python-mpi-pdt:
$LD_LIBRARY_PATH
% mpirun -np 4 <pkgs>/pyMPI-2.5b0-TAU/bin/pyMPI
./wrapper.py (Instrumented pyMPI with wrapper.py)
Usage Scenarios: 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
VNG Process Timeline with PAPI Counters
Vampir Counter Timeline Showing I/O BW
Generate a Trace File

% export TAU_MAKEFILE=/usr/local/packages/tau_latest/x86_64/lib/Makefile.tau-mpi-pdt

% export TAU_TRACE=1
% set path=(/usr/local/packages/tau_latest/x86_64/bin $path)
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% mpirun -np 4 ./a.out
% tau_treemerge.pl
(merges binary traces to create tau.trc and tau.edf files)
JUMPSHOT:
% tau2slog2 tau.trc tau.edf -o app.slog2
% jumpshot app.slog2
OR
VAMPIR:
% tau2otf tau.trc tau.edf app.otf -n 4 -z
(4 streams, compressed output trace)
% vampir app.otf
Goal: How does my application scale? What bottlenecks occur at what core counts?

Load profiles in PerfDMF database and examine with PerfExplorer
Usage Scenarios: Evaluate Scalability
Performance Regression Testing

![Graph showing FACETS Bassi Regression: 32 Procs (events above 2%)](image)

- int main(int, char **)  - std::vector<double, std::allocator<double>> FcCoreCellUpdate...
- void FcTmCoreFluxCalc::computeFluxes()  - MPI_Recv()
- double FcDataAssimilator::getValue(const std::string &, cons...
- MPI_Init()
- FcHdfSTmpl<DATATYPE>::writeDataSet
- void FcDataAssimilatorUfiles::parseUfiles(const std::vector<
- void FcUpdaterComponent::dumpToFile(const std::string &) con...
- other
Evaluate Scalability using PerfExplorer Charts

% export TAU_MAKEFILE=/usr/local/packages/tau_latest/x86_64
   /lib/Makefile.tau-mpi-pdt
% set path=(/usr/local/packages/tau_latest/x86_64/bin $path)
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% mpirun -np 1 ./a.out
% paraprof --pack 1p.ppk
% mpirun -np 2 ./a.out ...
% paraprof --pack 2p.ppk ... and so on.
On your client:
% perfdfm_configure --create-default
(Chooses derby, blank user/passwd, yes to save passwd, defaults)
% perfexplorer_configure
(Yes to load schema, defaults)
% paraprof
(load each trial: DB -> Add Trial -> Type (Paraprof Packed Profile) -> OK) OR use
   perfdfm_loadtrial
Then,
% perfexplorer
(Select experiment, Menu: Charts -> Speedup)
Goal: What is the volume of inter-process communication? Along which calling path?
Evaluate Scalability using PerfExplorer Charts

% export TAU_MAKEFILE=/usr/local/packages/tau_latest/x86_64
   /lib/Makefile.tau-mpi-pdt
% set path=/usr/local/packages/tau_latest/x86_64/bin $path
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% export TAU_COMM_MATRIX=1

% mpirun -np 4 ./a.out (setting the environment variables)

% paraprof
(Windows -> Communication Matrix)
ParaProf: Communication Matrix Display
Measuring Performance of PGI GPGPU Accelerated Code
Scaling NAMD with CUDA (Jumpshot with TAU)

Data transfer
Parallel Profile Visualization: ParaProf
Scalable Visualization: ParaProf (128k cores)
Scatter Plot: ParaProf (128k cores)
Automatic Instrumentation

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

Before

CXX = mpicxx
F90 = mpif90
CFLAGS =
LIBS = -lm
OBJS = f1.o f2.o f3.o ... fn.o

app: $(OBJS)
  $(CXX) $(LDFLAGS) $(OBJS) -o $@
  $(LIBS)

.cpp.o:
  $(CC) $(CFLAGS) -c $<

After

CXX = tau_cxx.sh
F90 = tau_f90.sh
CFLAGS =
LIBS = -lm
OBJS = f1.o f2.o f3.o ... fn.o

app: $(OBJS)
  $(CXX) $(LDFLAGS) $(OBJS) -o $@
  $(LIBS)

.cpp.o:
  $(CC) $(CFLAGS) -c $<
TAU_COMPILER Commandline Options

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

- Compilation:
  - `% mpif90 -c foo.f90`
  - Changes to
    - `% gfparse foo.f90 $(OPT1)`
    - `% tau_instrumentor foo.pdb foo.f90 -o foo.inst.f90 $(OPT2)`
    - `% mpif90 -c foo.f90 $(OPT3)`

- Linking:
  - `% mpif90 foo.o bar.o -o app`
  - Changes to
    - `% mpif90 foo.o bar.o -o app $(OPT4)`

- Where options OPT[1-4] default values may be overridden by the user:
  - F90 = `tau_f90.sh`
Optional parameters for TAU_OPTIONS: [tau_compiler.sh –help]

- optVerbose
  Turn on verbose debugging messages

- optCompInst
  Use compiler based instrumentation

- optNoCompInst
  Do not revert to compiler instrumentation if source instrumentation fails.

- optDetectMemoryLeaks
  Turn on debugging memory allocations/de-allocations to track leaks

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

- optPreProcess
  Preprocess Fortran sources before instrumentation

- optTauSelectFile=""
  Specify selective instrumentation file for tau_instrumentor

- 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)

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

- 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)
# 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_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_SYNCHRONIZE_CLOCKS</td>
<td>1</td>
<td>Synchronize clocks across nodes to correct timestamps in traces</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:linuxtimers:PAPI_FP_OPS:PAPI_NATIVE_&lt;event&gt;)</td>
</tr>
</tbody>
</table>
Optimization of Program Instrumentation

- Need to eliminate instrumentation in frequently executing lightweight routines
- Throttling of events at runtime (default in tau-2.17.2+):
  % export TAU_THROTTLE=1
  Turns off instrumentation in routines that execute over 100000 times (TAU_THROTTLE_NUMCALLS) and take less than 10 microseconds of inclusive time per call (TAU_THROTTLE_PERCALL). Use TAU_THROTTLE=0 to disable.
- Selective instrumentation file to filter events
  % tau_instrumentor [options] -f <file>  OR
  % export TAU_OPTIONS=’-optTauSelectFile=tau.txt’
- Compensation of local instrumentation overhead
  % configure -COMPENSATE
  or
  % export TAU_COMPENSATE=1  (in tau-2.19.2+)
### ParaProf: Creating Selective Instrumentation File

#### TAU: ParaProf Manager

<table>
<thead>
<tr>
<th>TrialField</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>200m4_p256.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>BCP Coords</td>
<td>(7,3,7)</td>
</tr>
<tr>
<td>BCP DiskSize (MB)</td>
<td>2048</td>
</tr>
<tr>
<td>BCP Location</td>
<td>R00-M1-N15-J32</td>
</tr>
<tr>
<td>BCP Node Mode</td>
<td>Cprocessor (22270844)</td>
</tr>
<tr>
<td>BCP Processor ID</td>
<td>0</td>
</tr>
<tr>
<td>BCP Size</td>
<td>(8,4,8)</td>
</tr>
<tr>
<td>BCP Itorpus</td>
<td>(0,0,0)</td>
</tr>
<tr>
<td>BCP numberInPset</td>
<td>1</td>
</tr>
<tr>
<td>BCP numPsets</td>
<td>256</td>
</tr>
<tr>
<td>BCP psetnum</td>
<td>3</td>
</tr>
<tr>
<td>BCP rankPset</td>
<td>24</td>
</tr>
<tr>
<td>CPU Type</td>
<td>450 Blue Gene/P DG2</td>
</tr>
<tr>
<td>CADW</td>
<td>/gpfs/home/kbann/Frontier/str/</td>
</tr>
<tr>
<td>Executable</td>
<td>/bin/taud/proxy</td>
</tr>
<tr>
<td>Hostname</td>
<td>lon-16</td>
</tr>
<tr>
<td>Local Time</td>
<td>2008-08-22T12:50:33-05:00</td>
</tr>
<tr>
<td>MPI Processor Name</td>
<td>Rank 255 of 256 &lt;7,3,7,0&gt; R00-M1-N15-J32</td>
</tr>
<tr>
<td>Memory Size</td>
<td>1816068 KB</td>
</tr>
<tr>
<td>Node Name</td>
<td>lon-16</td>
</tr>
<tr>
<td>OS Machine</td>
<td>BGP</td>
</tr>
<tr>
<td>OS Name</td>
<td>CNK</td>
</tr>
<tr>
<td>OS Release</td>
<td>2.6.19.2</td>
</tr>
<tr>
<td>OS Version</td>
<td>1</td>
</tr>
<tr>
<td>Starting Timestamp</td>
<td>1219427192054274</td>
</tr>
<tr>
<td>TAU Architecture</td>
<td>b gp</td>
</tr>
<tr>
<td>TAU Config</td>
<td>-arch=bgp -pdir=/soft/apps/tau/pdtoolkit-3.12 -...</td>
</tr>
<tr>
<td>TAU Version</td>
<td>2.17.1</td>
</tr>
<tr>
<td>Timestamp</td>
<td>12194271926121879</td>
</tr>
<tr>
<td>UTC Time</td>
<td>2008-08-22T17:50:33Z</td>
</tr>
<tr>
<td>p Id</td>
<td>355</td>
</tr>
</tbody>
</table>
Choosing Rules for Excluding Routines

Output File: /mnt/epsilon/Users/sameer/rs/taudata/frontier/select.tau

- Exclude Throttled Routines
- Exclude Lightweight Routines

<table>
<thead>
<tr>
<th>Lightweight Routine Exclusion Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microseconds per call:</td>
</tr>
<tr>
<td>Number of calls:</td>
</tr>
</tbody>
</table>

10
100000

Excluded Routines

- bool debugging(const char *) C
- double DGam_star(double, double, double, double, double, double) C
- double Mfluxsqr(double, double, double, double, double, double, double, double, double) C
- double compute_max(double, double *, double, double *, int) C
- double compute_min(double, double *, double, double *, int) C

save close
Lab Instructions

To profile a code using TAU:
1. Change the compiler name to tau_cxx.sh, tau_f90.sh, tau_cc.sh:
   
   \[ F90 = \text{tau}_\text{f90.sh} \]

2. Choose TAU stub makefile
   
   \[
   \% \text{export TAU\_MAKEFILE=} \\
   /usr/local/packages/tau\_latest/x86_64/lib/Makefile.tau-\text{[options]} \\
   \]

3. If stub makefile has \texttt{-papi} in its name, set the TAU\_METRICS environment variable:
   
   \[
   \% \text{export} \\
   \text{TAU\_METRICS=}\text{TIME:}\text{PAPI\_L2\_DCM:}\text{PAPI\_TOT\_CYC}... \\
   \]

4. Build and run workshop examples, then run \texttt{pprof/paraprof}