Tau
Introduction

Lars Koesterke
(& Kent Milfeld, Sameer Shende)

Cornell University
Ithaca, NY

March 13, 2009
Outline

• General
  – Measurements
  – Instrumentation & Control
  – Example: matmult

• Profiling and Tracing
  – Event Tracing
  – Steps for Performance Evaluation
  – Tau Architecture

• A look at a task-parallel MxM Implementation

• Paraprof Interface
General

- **Tuning and Analysis Utilities** (11+ year project effort)
  
  www.cs.uoregon.edu/research/paracomp/tau/

- *Performance system framework* for parallel, shared & distributed memory systems

- Targets a general complex system computation model
  - Nodes / Contexts / Threads

- *Integrated toolkit* for performance instrumentation, measurement, analysis, and visualization

**TAU = Profiler and Tracer + Hardware Counters + GUI + Database**
Tau: Measurements

• Parallel profiling
  – Function-level, block (loop)-level, statement-level
  – Supports user-defined events
  – TAU parallel profile data stored during execution
  – Hardware counter values
  – Support for multiple counters
  – Support for callgraph and callpath profiling

• Tracing
  – All profile-level events
  – Inter-process communication events
  – Trace merging and format conversion
Tau: Instrumentation

PDT is used to instrument your code.

Replace mpicc and mpif90 in make files with tau_f90.sh and tau_cc.sh

It is necessary to specify all the components that will be used in the instrumentation (mpi, openmp, profiling, counters [PAPI], etc. However, these come in a limited number of combinations.)

Combinations: First determine what you want to do (profiling, PAPI counters, tracing, etc.) and the programming paradigm (mpi, openmp), and the compiler. PDT is a required component:

<table>
<thead>
<tr>
<th>Instrumentation</th>
<th>Parallel Paradigm</th>
<th>Collectors</th>
<th>Compiler:</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDT Hand-code</td>
<td>MPI OMP Callpath</td>
<td>PAPI</td>
<td>intel pgi gnu</td>
</tr>
<tr>
<td></td>
<td></td>
<td>…</td>
<td></td>
</tr>
</tbody>
</table>
You can view the available combinations
(also tauTypes 'ls -C1 $TAU | grep Makefile ' ).

Your selected combination is made known to the compiler wrapper through the TAU_MAKEFILE environment variable.

E.g. the PDT instrumentation (pdt) for the Intel compiler (icpc) for MPI (mpi) is set with this command:

    setenv TAU_MAKEFILE /.../Makefile.tau-icpc-mpi-pdt

Other run-time and instrumentation options are set through TAU_OPTIONS. For verbose:

    setenv TAU_OPTIONS ' -optVerbose'
% tar –xvf ~train00/tau.tar

% cd tau
★ READ the Instructions file

% source sourceme.csh
or
% source sourceme.sh create env. (modules and TAU_MAKEFILE)

% make matmultf create executable(s)
or
% make matmultc

% qsub job submit job (edit and uncomment ibrun line)

% paraprof (for GUI) Analyze performance data:
Definitions – Profiling

- **Profiling**
  - Recording of summary information during execution
    - inclusive, exclusive time, # calls, hardware statistics, …
  - Reflects performance behavior of program entities
    - functions, loops, basic blocks
    - user-defined “semantic” entities
  - Very good for low-cost performance assessment
  - Helps to expose performance bottlenecks and hotspots
  - Implemented through
    - sampling: periodic OS interrupts or hardware counter traps
    - instrumentation: direct insertion of measurement code
Definitions – Tracing

Tracing

- Recording of information about significant points (events) during program execution
  - entering/exiting code region (function, loop, block, …)
  - thread/process interactions (e.g., send/receive message)
- Save information in event record
  - timestamp
  - CPU identifier, thread identifier
  - Event type and event-specific information
- Event trace is a time-sequenced stream of event records
- Can be used to reconstruct dynamic program behavior
- Typically requires code instrumentation
Event Tracing: Instrumentation, Monitor, Trace

CPU A:

```c
void master {
    trace(EXIT, 1);
    ...
    trace(SEND, B);
    send(B, tag, buf);
    ...
    trace(EXIT, 1);
}
```

CPU B:

```c
void worker {
    trace(EXIT, 2);
    ...
    recv(A, tag, buf);
    ...
    trace(RECV, A);
    ...
    trace(EXIT, 2);
}
```

Event definition:

<table>
<thead>
<tr>
<th></th>
<th>master</th>
<th>worker</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Event Log:

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>Event</th>
<th>CPU</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>58</td>
<td>A</td>
<td>ENTER</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>B</td>
<td>ENTER</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>A</td>
<td>SEND</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>A</td>
<td>EXIT</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>B</td>
<td>RECV</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>B</td>
<td>EXIT</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Event Tracing: “Timeline” Visualization

<table>
<thead>
<tr>
<th></th>
<th>master</th>
<th>worker</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

... | ...
---|---
58 | A ENTER 1
60 | B ENTER 2
62 | A SEND B
64 | A EXIT 1
68 | B RECV A
69 | B EXIT 2
... | ...

A

B

main
master
worker

58 60 62 64 66 68 70
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
TAU Performance System Architecture

Analysis
- Instrumentation
- Measurement
  - profiles
  - traces
- Profile Data Management (PerfDMF)
  - profile translators
  - Metadata (XML)
  - profile database
- Profile Analysis (ParaProf)
  - Profile Data Mining (PerfExplorer)
- Trace Data Management
  - trace translators
  - trace storage
- Trace Visualizers
  - Vampir
  - JumpShot
  - Paraver
- Trace Analyzers
  - Expert
  - ProfileGen
  - Vampir Server

TAU Performance System Architecture
Overview of Matmult: $C = A \times B$

**MASTER**
- Create $A$ & $B$

**Worker**
- Send $B$ → Receive $B$
- Send Row of $A$ → Receive a
- Multiply row $a \times B$ → $a \times j$
- Receive Row of $C$ → Send Back row of $C$

Order $N$ P Tasks
Preparation of Matmult: $C = A \times B$

**MASTER**

Generate A & B

Create A

Create B

Broadcast B to All by columns

PE 0 \rightarrow PE x

MPI_Bcast( b(1,i)…)

loop over i (i=1 \rightarrow n)

Order N P Tasks

$C = A \times B$
Master Ops of Matmult: $C = A \times B$

**MASTER**

Master (0) sends rows 1 through $(p-1)$ to slaves $(1 \rightarrow p-1)$ receives

Master (0) receives rows 1 through $(n)$ from Slaves.
Master Ops of Matmult: $C = A \times B$

**Worker**

- Pick up broadcast of B columns from PE 0
- Slave receives any A row from PE 0
- Slaves multiply all Columns of B into A (row $i$) to form row $i$ of Matrix C
- Slave(any) sends row $j$ of C to master, PE 0

```
MPI_Send( crow ... j
PE 0
```

```
loop over i (i=1 → n)
MPI_Recv( arow ... ANY, j
```

```
Matrix * Vector
```

```
MPI_Send( crou ... j
```
Paraprof and Pprof

- Execute application and analyze performance data:
- `% qsub job`
  - Look for files: profile.<task_no>.
  - With Multiple counters, look for directories for each counter.
- `% pprof (for text based profile display)`
- `% paraprof (for GUI)`
  - pprof and paraprof will discover files/directories.
  - paraprof runs on PCs, Files/Directories can be downloaded to laptop and analyzed there.
Tau Paraprof Overview

- Raw files
- PerfDMF managed (database)
- Application
- Experiment
- Trial

HPMToolkit
Metadata
MpiP
TAU
Tau Paraprof Manager Window

Provides Machine Details
Organizes Runs as: Applications, Experiments and Trials.
Routine Time Experiment

Profile Information is in “GET_TIME_OF_DAY” metric
Mean and Standard Deviation Statistics given.
Multiply_Matrices Routine Results

Function Data Window gives a closer look at a single function:

<table>
<thead>
<tr>
<th>Function Data Window</th>
<th>Multiplied Matrices ([matmult.f90] [25, 18])</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name: MULTIPLE_MATRICES [matmult.f90] [25, 18]</td>
<td></td>
</tr>
<tr>
<td>Metric Name: GET_TIME_OF_DAY</td>
<td></td>
</tr>
<tr>
<td>Value: Exclusive</td>
<td></td>
</tr>
<tr>
<td>Units: seconds</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time (seconds)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>59.999</td>
<td>n,c,t 12,0,0</td>
</tr>
<tr>
<td>59.995</td>
<td>n,c,t 13,0,0</td>
</tr>
<tr>
<td>59.955</td>
<td>n,c,t 14,0,0</td>
</tr>
<tr>
<td>59.94</td>
<td>n,c,t 3,0,0</td>
</tr>
<tr>
<td>59.939</td>
<td>n,c,t 1,0,0</td>
</tr>
<tr>
<td>59.916</td>
<td>n,c,t 10,0,0</td>
</tr>
<tr>
<td>59.904</td>
<td>n,c,t 15,0,0</td>
</tr>
<tr>
<td>59.896</td>
<td>n,c,t 9,0,0</td>
</tr>
<tr>
<td>59.896</td>
<td>mean</td>
</tr>
<tr>
<td>59.875</td>
<td>n,c,t 5,0,0</td>
</tr>
<tr>
<td>59.852</td>
<td>n,c,t 12,0,0</td>
</tr>
<tr>
<td>59.848</td>
<td>n,c,t 7,0,0</td>
</tr>
<tr>
<td>59.847</td>
<td>n,c,t 10,0,0</td>
</tr>
<tr>
<td>59.838</td>
<td>n,c,t 8,0,0</td>
</tr>
<tr>
<td>59.838</td>
<td>n,c,t 6,0,0</td>
</tr>
<tr>
<td>59.838</td>
<td>n,c,t 11,0,0</td>
</tr>
<tr>
<td>59.838</td>
<td>n,c,t 4,0,0</td>
</tr>
<tr>
<td>0.051</td>
<td>std. dev.</td>
</tr>
</tbody>
</table>

Not from same run
Float Point OPS trial

Hardware Counters provide Floating Point Operations (Function Data view).

![Graph showing floating point operations metrics](image-url)
L1 Data Cache Miss trial

Hardware Counters provide L1 Cache Miss Operations.

<table>
<thead>
<tr>
<th>Metric:</th>
<th>Value:</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_L1_DCM</td>
<td>Exclusive</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>std. dev.</th>
<th>mean</th>
<th>n,c,t 0,0,0</th>
<th>n,c,t 1,0,0</th>
<th>n,c,t 2,0,0</th>
<th>n,c,t 3,0,0</th>
<th>n,c,t 4,0,0</th>
<th>n,c,t 5,0,0</th>
<th>n,c,t 6,0,0</th>
<th>n,c,t 7,0,0</th>
<th>n,c,t 8,0,0</th>
<th>n,c,t 9,0,0</th>
<th>n,c,t 10,0,0</th>
<th>n,c,t 11,0,0</th>
<th>n,c,t 12,0,0</th>
<th>n,c,t 13,0,0</th>
<th>n,c,t 14,0,0</th>
<th>n,c,t 15,0,0</th>
</tr>
</thead>
</table>
Call Path

Call Graph Paths (Must select through “thread” menu.)
TAU_MAKEFILE = ...
Makefile.tau-callpath-icpc-mpi-pdt

<table>
<thead>
<tr>
<th>Standard Deviation</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>n,ct 0,0,0</td>
<td></td>
</tr>
<tr>
<td>n,ct 1,0,0</td>
<td></td>
</tr>
<tr>
<td>n,ct 2,0,0</td>
<td></td>
</tr>
<tr>
<td>n,ct 3,0,0</td>
<td></td>
</tr>
<tr>
<td>n,ct 4,0,0</td>
<td></td>
</tr>
<tr>
<td>n,ct 5,0,0</td>
<td></td>
</tr>
<tr>
<td>n,ct 6,0,0</td>
<td></td>
</tr>
<tr>
<td>n,ct 7,0,0</td>
<td></td>
</tr>
<tr>
<td>n,ct 8,0,0</td>
<td></td>
</tr>
<tr>
<td>n,ct 9,0,0</td>
<td></td>
</tr>
<tr>
<td>n,ct 10,0,0</td>
<td></td>
</tr>
<tr>
<td>n,ct 11,0,0</td>
<td></td>
</tr>
<tr>
<td>n,ct 12,0,0</td>
<td></td>
</tr>
<tr>
<td>n,ct 13,0,0</td>
<td></td>
</tr>
<tr>
<td>n,ct 14,0,0</td>
<td></td>
</tr>
<tr>
<td>n,ct 15,0,0</td>
<td></td>
</tr>
</tbody>
</table>
Derived Metrics

Select Argument 1 (green ball); Select Argument 2 (green ball); Select Operation; then Apply. Derived Metric will appear as a new trial.

**Argument 1:** 0:0:0:1 - PAPI_FP_OPS

**Argument 2:** 0:0:0:2 - PAPI_L1_DCM

**Divide**  

[Apply operation]
Be careful→ even though ratios are constant, cores may do different amounts of work/operations per call.

Since FP/Miss ratios are constant→ must be memory access problem.
PAPI Implementation

**Portable Layer**
- PAPI Low Level

**Machine Specific Layer**
- PAPI Machine Dependent Substrate
  - Kernel Extension
  - Operating System
  - Hardware Performance Counter

**Tools**
- PAPI High Level
PAPI Performance Monitor

- Provides high level counters for events:
  - Floating point instructions/operations,
  - Total instructions and cycles
  - Cache accesses and misses
  - Translation Lookaside Buffer (TLB) counts
  - Branch instructions taken, predicted, mispredicted

- PAPI_flops routine for basic performance analysis
  - Wall and processor times
  - Total floating point operations and MFLOPS
    http://icl.cs.utk.edu/projects/papi

- Low level functions are thread-safe, high level are not
PAPI Preset Events

• Proposed standard set of events deemed most relevant for application performance tuning
• Defined in papiStdEventDefs.h
• Mapped to native events on a given platform
  – Run tests/avail to see list of PAPI preset events available on a platform
High-level Interface

- Meant for application programmers wanting coarse-grained measurements
- Not thread safe
- Calls the lower level API
- Allows only PAPI preset events
- Easier to use and less setup (additional code) than low-level
High-level API

• C interface
  PAPI_start_counters
  PAPI_read_counters
  PAPI_stop_counters
  PAPI_accum_counters
  PAPI_num_counters
  PAPI_flips
  PAPI_ipc

• Fortran interface
  PAPIF_start_counters
  PAPIF_read_counters
  PAPIF_stop_counters
  PAPIF_accum_counters
  PAPIF_num_counters
  PAPIF_flips
  PAPIF_ipc
Low-level Interface

- Increased efficiency and functionality over the high level PAPI interface
- About 40 functions
- Obtain information about the executable and the hardware
- Thread-safe
- Fully programmable
- Callbacks on counter overflow
PAPI counters in Tau

• Instead of one metric, profile or trace with more than one metric
• Set environment variables COUNTER[1-25] to specify the metric
  % setenv COUNTER1 GET_TIME_OF_DAY
  % setenv COUNTER2 PAPI_L2_DCM
  % setenv COUNTER3 PAPI_FP_OPS
  % setenv COUNTER4 PAPI_NATIVE_<native_event>
• % setenv COUNTER5 P_WALL_CLOCK_TIME ...
• When used with –TRACE option, the first counter must be GET_TIME_OF_DAY
  % setenv COUNTER1 GET_TIME_OF_DAY
  Provides a globally synchronized real time clock for tracing
• -multiplecounters appears in the name of the stub Makefile
• Often used with –papi=<dir> to measure hardware performance counters and time
• papi_native_avail and papi_avail are two useful tools.
Important Environment Variables

• Choose the measurement option and compile your code:
• `setenv TAU_MAKEFILE $TAU/Makefile.tau-icpc-mpi-pdt`
• `setenv TAU_OPTIONS '-optVerbose -optKeepFiles -optPreProcess'`
• `setenv TAU_THROTTLE 1  At runtime to keep instrumentation overhead in check`
Fortran TAU Tips

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

- If it uses several module files, you may switch from the default Cleanscape Inc. parser in PDT to the GNU gfortran parser to generate PDB files:
  ```
  % setenv TAU_OPTIONS `-optPdtGnuFortranParser -optVerbose`
  ```

- If your Fortran code uses C preprocessor directives (#include, #ifdef, #endif):
  ```
  % setenv TAU_OPTIONS `-optPreProcess -optVerbose -optDetectMemoryLeaks`
  ```

- To use an instrumentation specification file:
  ```
  % setenv TAU_OPTIONS `-optTauSelectFile=mycmd.tau -optVerbose -optPreProcess`
  % cat mycmd.tau
  ```
  ```
  BEGIN_INSTRUMENT_SECTION
  memory file="foo.f90" routine="#"
  # instruments all allocate/deallocate statements in all routines in foo.f90
  loops file="*" routine="#"
  io file="abc.f90" routine="FOO"
  END_INSTRUMENT_SECTION
  ```
References

• Performance Research Laboratory, University of Oregon, Eugene, sameer@cs.uoregon.edu
• http://www.cs.uoregon.edu/research/tau/about.php