Performance Observation

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

- Motivation
- Introduction to TAU
- Optimizing instrumentation: approaches
- Perturbation compensation
- Conclusion
Tools for performance problem solving

- Empirical-based performance optimization process
- Performance technology concerns

Performance Technology

- Experiment management
- Performance database

Performance Tuning

- Hypotheses

Performance Diagnosis

- Properties

Performance Experimentation

- Characterization

Performance Observation

Performance Technology

- Instrumentation
- Measurement
- Analysis
- Visualization
TAU Performance System

- Tuning and Analysis Utilities (11+ year project effort)
- Performance system framework for scalable parallel and distributed high-performance computing
- Targets a general complex system computation model
  - nodes / contexts / threads
  - Multi-level: system / software / parallelism
  - Measurement and analysis abstraction
- Integrated toolkit for performance instrumentation, measurement, analysis, and visualization
  - Portable performance profiling and tracing facility
  - Open software approach with technology integration
- University of Oregon, Forschungszentrum Jülich, LANL
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 slave {
    trace(EXIT, 2);
    ...
    recv(A, tag, buf);
    ...
    trace(RECV, A);
    ...
    trace(EXIT, 2);
}
```

Event definition

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>master</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>slave</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

MONITOR

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>58</td>
<td>A</td>
<td>ENTER</td>
</tr>
<tr>
<td>60</td>
<td>B</td>
<td>ENTER</td>
</tr>
<tr>
<td>62</td>
<td>A</td>
<td>SEND</td>
</tr>
<tr>
<td>64</td>
<td>A</td>
<td>EXIT</td>
</tr>
<tr>
<td>68</td>
<td>B</td>
<td>RECV</td>
</tr>
<tr>
<td>69</td>
<td>B</td>
<td>EXIT</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CPU A:

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

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>58</td>
<td>A</td>
<td>ENTER 1</td>
</tr>
<tr>
<td>60</td>
<td>B</td>
<td>ENTER 2</td>
</tr>
<tr>
<td>62</td>
<td>A</td>
<td>SEND B</td>
</tr>
<tr>
<td>64</td>
<td>A</td>
<td>EXIT 1</td>
</tr>
<tr>
<td>68</td>
<td>B</td>
<td>RECV A</td>
</tr>
<tr>
<td>69</td>
<td>B</td>
<td>EXIT 2</td>
</tr>
</tbody>
</table>

![Timeline Visualization]

- **main**
- **master**
- **slave**
TAU Performance System Architecture

Instrumentation
- Source Code
- Pre-processor
- Instrumented Source Code
- Compiler
- Object Code
- Linker
- Executable Code
- Binary Rewrite

TAU API
- PROFILE
- Run-Time Library Modules
- TRACE

Measurement
- Profiling Data Files
- Profile Groups
- Function Database
- Statistics
- Hardware Counters
- User-Level Timers
- Event Traces
- Event Tables

Analysis
- Racy
- pprof
- ASCII Report
- Trace Logs
- Paraver
- Vampir
- EPILOG
TAU Analysis

- Parallel profile analysis
  - *Pprof*
    - parallel profiler with text-based display
  - *ParaProf*
    - Graphical, scalable, parallel profile analysis and display

- Trace analysis and visualization
  - Trace merging and clock adjustment (if necessary)
  - Trace format conversion (ALOG, SDDF, VTF, Paraver)
  - Trace visualization using *Vampir* (Pallas/Intel)
## Pprof Output (NAS Parallel Benchmark – LU)

<table>
<thead>
<tr>
<th>%Time</th>
<th>Exclusive msec</th>
<th>Inclusive msec</th>
<th>#Call</th>
<th>#Subs</th>
<th>Inclusive Name usec/call</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0</td>
<td>1</td>
<td>0.00</td>
<td>1</td>
<td>15</td>
<td>191293269 apcpu</td>
</tr>
<tr>
<td>99.6</td>
<td>3</td>
<td>0.00</td>
<td>3</td>
<td>37517</td>
<td>63487925 bcast_inputs</td>
</tr>
<tr>
<td>67.1</td>
<td>4.65</td>
<td>1.02</td>
<td>37200</td>
<td></td>
<td>3450 exchange_1</td>
</tr>
<tr>
<td>44.5</td>
<td>2</td>
<td>0.00</td>
<td>37200</td>
<td></td>
<td>9157 bufs</td>
</tr>
<tr>
<td>41.0</td>
<td>1.13</td>
<td>1.18</td>
<td>18600</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>29.5</td>
<td>5.91</td>
<td>5.40</td>
<td>9300</td>
<td>18600</td>
<td>4217 MPI_Recv()</td>
</tr>
<tr>
<td>26.2</td>
<td>50.14</td>
<td>50.14</td>
<td>301</td>
<td>602</td>
<td>5065 blts</td>
</tr>
<tr>
<td>16.2</td>
<td>24.451</td>
<td>31.031</td>
<td>4</td>
<td></td>
<td>2611 MPI_Send()</td>
</tr>
<tr>
<td>3.9</td>
<td>7.501</td>
<td>7.501</td>
<td>9300</td>
<td>1812</td>
<td>10918 exchange_3</td>
</tr>
<tr>
<td>3.4</td>
<td>8.38</td>
<td>6.594</td>
<td>604</td>
<td></td>
<td>709 jacu</td>
</tr>
<tr>
<td>3.4</td>
<td>5.590</td>
<td>6.590</td>
<td>9300</td>
<td>0</td>
<td>8205 MPI_Wait()</td>
</tr>
<tr>
<td>2.6</td>
<td>4.989</td>
<td>4.989</td>
<td>608</td>
<td>0</td>
<td>400081 init_comm</td>
</tr>
<tr>
<td>0.2</td>
<td>0.44</td>
<td>0.44</td>
<td>4</td>
<td></td>
<td>399334 MPI_Init()</td>
</tr>
<tr>
<td>0.1</td>
<td>1.41</td>
<td>1.24</td>
<td>1</td>
<td>1</td>
<td>247065 setiv</td>
</tr>
<tr>
<td>0.1</td>
<td>1.31</td>
<td>1.31</td>
<td>57252</td>
<td>0</td>
<td>2 exact</td>
</tr>
<tr>
<td>0.1</td>
<td>1.03</td>
<td>1.03</td>
<td>103</td>
<td>2</td>
<td>103168 erhs</td>
</tr>
<tr>
<td>0.1</td>
<td>0.956</td>
<td>0.96</td>
<td>1</td>
<td>2</td>
<td>96458 read_inout</td>
</tr>
<tr>
<td>0.0</td>
<td>95</td>
<td>95</td>
<td>9</td>
<td>0</td>
<td>10603 MPI_Ecast()</td>
</tr>
<tr>
<td>0.0</td>
<td>26</td>
<td>44</td>
<td>1</td>
<td>7937</td>
<td>440673 error</td>
</tr>
<tr>
<td>0.0</td>
<td>24</td>
<td>24</td>
<td>608</td>
<td>0</td>
<td>40 MPI_Irecv()</td>
</tr>
<tr>
<td>0.0</td>
<td>15</td>
<td>15</td>
<td>1</td>
<td>5</td>
<td>15630 MPI_Finalize()</td>
</tr>
<tr>
<td>0.0</td>
<td>4</td>
<td>12</td>
<td>1</td>
<td>1700</td>
<td>12335 setiv</td>
</tr>
<tr>
<td>0.0</td>
<td>7</td>
<td>7</td>
<td>3</td>
<td>3</td>
<td>2693 l2norm</td>
</tr>
<tr>
<td>0.0</td>
<td>3</td>
<td>3</td>
<td>8</td>
<td>0</td>
<td>491 MPI_Allreduce()</td>
</tr>
<tr>
<td>0.0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>3874 pintg</td>
</tr>
<tr>
<td>0.0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1007 MPI_Barrier()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.116</td>
<td>0.837</td>
<td>1</td>
<td>4</td>
<td>637 exchange_4</td>
</tr>
<tr>
<td>0.0</td>
<td>0.512</td>
<td>0.512</td>
<td>1</td>
<td>2</td>
<td>512 MPI_keyval_create()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.121</td>
<td>0.253</td>
<td>1</td>
<td>2</td>
<td>393 exchange_5</td>
</tr>
<tr>
<td>0.0</td>
<td>0.024</td>
<td>0.191</td>
<td>1</td>
<td>2</td>
<td>191 exchange_6</td>
</tr>
<tr>
<td>0.0</td>
<td>0.103</td>
<td>0.103</td>
<td>6</td>
<td>0</td>
<td>17 MPI_Type_contiguous()</td>
</tr>
</tbody>
</table>

- Intel Quad PIII Xeon
- F90 + MPICH
- Profile
  - Node
  - Context
  - Thread
- Events
  - code
  - MPI
Terminology – Example

- For routine “int main( )”:
  - Exclusive time
    - 100-20-50-20=10 secs
  - Inclusive time
    - 100 secs
  - Calls
    - 1 call
  - Subrs (no. of child routines called)
    - 3
  - Inclusive time/call
    - 100secs

```c
int main( )
{
    /* takes 100 secs */

    f1(); /* takes 20 secs */
    f2(); /* takes 50 secs */
    f1(); /* takes 20 secs */

    /* other work */
}

/*
Time can be replaced by counts */
```
ParaProf (NAS Parallel Benchmark – LU)

Node, context, thread

Global profiles

Routine profile across all nodes

Event legend

Individual profile

Performance Measurement
Trace Visualization using Vampir [Intel/Pallas]

Timeline display

Callgraph display

Parallelism display

Communications display
PETSc ex19 (Tracing)

Commonly seen communication behavior
MPI_Alltoall is an execution bottleneck
Strategies for Empirical Performance Evaluation

Empirical performance evaluation as a series of performance experiments

- Experiment trials describing instrumentation and measurement requirements
- Where/When/How axes of empirical performance space
  - where are performance measurements made in program
    - routines, loops, statements…
  - when is performance instrumentation done
    - compile-time, while pre-processing, runtime…
  - how are performance measurement/instrumentation chosen
    - profiling with hw counters, tracing, callpath profiling…
TAU Instrumentation Approach

- Support for standard program events
  - Routines
  - Classes and templates
  - Statement-level blocks
- Support for user-defined events
  - Begin/End events ("user-defined timers")
  - Atomic events
  - Selection of event statistics
- Support definition of “semantic” entities for mapping
- Support for event groups
- Instrumentation optimization
Flexible instrumentation mechanisms at multiple levels

- **Source code**
  - manual
  - automatic
    - C, C++, F77/90/95 (Program Database Toolkit (*PDT*))
    - OpenMP (directive rewriting (*Opari*), *POMP* spec)

- **Object code**
  - pre-instrumented libraries (e.g., MPI using *PMPI*)
  - statically-linked and dynamically-linked

- **Executable code**
  - dynamic instrumentation (pre-execution) (*DynInstAPI*)
  - virtual machine instrumentation (e.g., Java using *JVMPi*)
Multi-Level Instrumentation

- Targets common measurement interface
  - TAU API
- Multiple instrumentation interfaces
  - Simultaneously active
- Information sharing between interfaces
  - Utilizes instrumentation knowledge between levels
- Selective instrumentation
  - Available at each level
  - Cross-level selection
- Targets a common performance model
- Presents a unified view of execution
  - Consistent performance events
TAU Measurement Options

- Parallel profiling
  - Function-level, block-level, statement-level
  - Supports user-defined events
  - TAU parallel profile data stored during execution
  - Hardware counts 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
Optimizing Instrumentation

- **Grouping**
  - Enable/disable profile groups at runtime

- **Selective Instrumentation**
  - Include/exclude events (or files) for instrumentation

- **Re-instrumentation**
  - Profile, overhead analysis, exclude events, re-instrument

- **Compensation**
  - Overhead calibration, removal
Grouping Performance Data in TAU

Profile Groups

- A group of related routines forms a profile group
- Statically defined
  - TAU_DEFAULT, TAU_USER[1-5], TAU_MESSAGE, TAU_IO, …
- Dynamically defined
  - group name based on string, such as “adlib” or “particles”
  - runtime lookup in a map to get unique group identifier
  - uses tau_instrumentor to instrument
- Ability to change group names at runtime
- Group-based instrumentation and measurement control
Selective Instrumentation

- Selection of which performance events to observe
  - Could depend on scope, type, level of interest
  - Could depend on instrumentation overhead

- How is selection supported in instrumentation system?
  - No choice
  - Include / exclude routine and file lists (TAU)
  - Environment variables
  - Static vs. dynamic
Automatic Instrumentation of Source Code

% cxxparse file.cpp -I/dir -Dflags       [PDT: Program Database Toolkit]
% tau_instrumentor

Usage : tau_instrumentor <pdbfile> <sourcefile> [-o <outputfile>] [-noinline]
[-g groupname] [-i headerfile] [-c|-c++|-fortran] [-f <instr_req_file> ]
For selective instrumentation, use -f option
% tau_instrumentor foo.pdb foo.cpp -o foo.inst.cpp -f selective.dat
% cat selective.dat

# Selective instrumentation: Specify an exclude/include list of routines/files.
BEGIN_EXCLUDE_LIST
void quicksort(int *, int, int)
void sort_5elements(int *)
void interchange(int *, int *)
END_EXCLUDE_LIST

BEGIN_FILE_INCLUDE_LIST
Main.cpp
Foo?.c
*.C
END_FILE_INCLUDE_LIST

# Instruments routines in Main.cpp, Foo?.c and *.C files only
# Use BEGIN_[FILE]_INCLUDE_LIST with END_[FILE]_INCLUDE_LIST
Distortion of Performance Data

- Problem: Controlling instrumentation of small routines
  - High relative measurement overhead
  - Significant intrusion and possible perturbation
- Solution: Re-instrument the application!
  - Weed out frequently executing lightweight routine
  - Feedback to instrumentation system
Re-instrumentation

- **Tau_reduce**: rule based overhead analysis
- Analyze the performance data to determine events with high (relative) overhead performance measurements
- Create a select list for excluding those events
- Rule grammar (used in *tau_reduce* tool [N. Trebon, UO])

```
[GroupName:] Field Operator Number

- **GroupName** indicates rule applies to events in group
- **Field** is an event metric attribute (from profile statistics)
  - numcalls, numsubs, percent, usec, cumusec, count [PAPI],
    totalcount, stdev, usecs/call, counts/call
- **Operator** is one of >, <, or =
- **Number** is any number
- Compound rules possible using & between simple rules
```
Example Rules

- #Exclude all events that are members of TAU_USER and use less than 1000 microseconds
  TAU_USER:usec < 1000

- #Exclude all events that have less than 100 microseconds and are called only once
  usec < 1000 & numcalls = 1

- #Exclude all events that have less than 1000 usecs per call OR have a (total inclusive) percent less than 5
  usecs/call < 1000
  percent < 5

- Scientific notation can be used
  - usec>1000 & numcalls>400000 & usecs/call<30 & percent>25
TAU_REDUCE

- Reads profile files and rules
- Creates selective instrumentation file
  - Specifies which routines should be excluded from instrumentation
Compensation of Overhead

- Runtime estimation of a single timer overhead
- Evaluation of number of timer calls along a calling path
- Compensation by subtracting timer overhead
- Recalculation of performance metrics
Introduce a pair of timer calls (start/stop)

\[
T_{\text{actual}} = T_{\text{measured}} - (b+c)
\]

\[
t_1 = n \times (b+c)
\]

\[
t_2 = b+n \times (a+b+c+d)+c
\]

\[
T_{\text{overhead}} = a+b+c+d = \frac{t_2 - (t_1/n)}{n}
\]

\[
T_{\text{null}} = b+c = \frac{t_1}{n}
\]
Recalculating Inclusive Time

- Number of children/grandchildren... nodes
- Traverse callstack

\[ T_{\text{actual}} = T_{\text{measured}} - (b+c) - n_{\text{descendants}} \times T_{\text{overhead}} \]
- Compensate for synchronization operations

Diagram:
- Timer start/stop call
- Message

Process A

Process B

wait

Time
Lamport’s Logical Time [Lamport 1978]

- Logical time incremented by timer start/stop
- Accumulate timer overhead on local process
- Send local timer overhead with message

**Diagram:**
- Process A
  - $t^A_{\text{overhead}}$
  - Timer start/stop call
  - Message
- Process B
  - $t^A_{\text{overhead}}$
  - $t^A_{\text{overhead}} > t^B_{\text{overhead}}$?
  - Yes: $t^B_{\text{overhead}} = t^A_{\text{overhead}}$
    - $t_{\text{wait}} = t_{\text{wait}} - (t^A_{\text{overhead}} - t^B_{\text{overhead}})$
    - $= 0$ (if negative)
  - $t^A_{\text{overhead}}$
  - $t^A_{\text{overhead}}$

**Notes:**
- Logical time incremented by timer start/stop
- Accumulate timer overhead on local process
- Send local timer overhead with message
Compensation (contd.)

- **Message passing programs**
  - Adjust wait times (MPI_Recv, MPI_Wait…)
  - Adjust barrier wait times (MPI_Barrier)
    - Each process sends its timer overheads to all other tasks
    - Each task compares its overhead with max overhead

- **Shared memory multi-threaded programs**
  - Adjust barrier synchronization wait times
    - Each task compares its overhead to max overhead from all participating threads
  - Adjust semaphore/condition variable wait times
    - Each task compares its overhead with other thread’s overhead
Conclusions

- Complex software and parallel computing systems pose challenging performance analysis problems that require robust methodologies and tools.
- Optimizing instrumentation is a key step towards balancing the volume of performance data with accuracy of measurements.
- Present new research in the area of performance perturbation compensation techniques for profiling.
- [http://www.cs.uoregon.edu/research/paracomp/tau](http://www.cs.uoregon.edu/research/paracomp/tau)
Support Acknowledgements

- Department of Energy (DOE)
  - Office of Science contracts
  - University of Utah DOE ASCI Level 1 sub-contract
  - DOE ASCI Level 3 (LANL, LLNL)

- NSF National Young Investigator (NYI) award

- Research Centre Juelich
  - John von Neumann Institute for Computing
  - Dr. Bernd Mohr

- Los Alamos National Laboratory