TAU Portable Performance Profiling Tools

Sameer Shende
Department of CIS, University of Oregon,
Advanced Computing Laboratory, Los Alamos National Laboratory
sameer@cs.uoregon.edu
TAU Profiling Team Members

(In alphabetical order)

Peter Beckman (LANL)
Prof. Janice Cuny (UO)
Steve Karmesin (LANL)
Kathleen Lindlan (UO)
Prof. Allen D. Malony (UO)
Sameer Shende (UO, LANL)
TAU: Tuning and Analysis Utilities

Tau is...

• An extensible tool framework supporting tool interactions with the program, the user, and each other

• A graphical, program development environment with several distinct and unique, but completely integrated, tools

• A performance analysis environment facilitating static and dynamic interactions with programs

Tau is designed to...

• Operate on language-level program objects of high-level parallel programming languages

• Be extensible in many dimensions, easing creation of additional tools, retargeting to new languages, and porting to new machine environments

• Be tightly integrated through well-defined interfaces with both compilers and runtime systems
TAU Static Callgraph Display Tool (CAGEY)
TAU Class Hierarchy Browser (CLASSY)
TAU File and Class Browser (FANCY)
TAU Performance Profiling Tool (RACY)
“The purpose of performance profiling is to find out where a program is spending its time - in precisely which procedures or lines of code”
Performance Profiling Approaches

• Callstack sampling:
  Periodically examine and record the program counter and call-stack and resource consumption. Estimate performance based on samples. (e.g., CC -p, Speedshop SGI)

• Instrumentation:
  For counting the exact number of times a function executed. Insert binary instrumentation to generate a trap at function exit. (e.g., ideal CPU time (pixie) experiment SGI).

• Binary instrumentation:
  Runtime instrumentation to identify bottlenecks (e.g., Dyn-Inst, Paradyne U. Wisconsin).

• Trace based performance prediction:
  Generate trace of each function entry/exit and report performance statistics (e.g., Pablo UIUC, Vampir Pallas, Germany)

• Exception Trace:
  Records the location of floating point exceptions (e.g., Speedshop SGI)

• Event based sampling:
  (hardware performance counters, VTune)
TAU Performance Profiling Model

Instrumentation of source code

- Instrumentation is inserted either manually or by a preprocessor in the source code.

- Function (block) entry and exit points are used to record the exact time spent in the instrumented function.

- Statistics maintained in the instrumented program and small profile files generated at the end of execution.

- Supported for C++ and its parallel derivatives (pC++, HPC++)

- Compiler support needed for automatic instrumentation (Sage++, EDG).

- Higher level profiling. Suitable for C++ libraries, applications where source code is available for instrumentation.
Design goals for TAU Profiling and Tracing

• Provide a consistent, portable profiling and tracing library that would work with multiple compilers, operating systems and platforms for parallel C++ libraries and applications.

• Profiling should report the exact time spent in each instrumented function instead of an estimate.

• Overhead of profiling or tracing should be limited to the groups of profiled functions selected at runtime.

• Lightweight profiling facility.

• When profiling and tracing are disabled, the instrumented code should run without any runtime overhead (default).

• Time spent in templates should be reported based on distinct template instantiations.

• TAU should support the ISO/ANSI standard for C++.

• It should be possible to use statement level user defined timers for profiling a set of statements.

• Hardware performance counters may be used instead of time.
Terminology: Exclusive and Inclusive

“Inclusive time is the total time spent in a function and all the other functions it calls directly or indirectly.

Exclusive time is the time spent in the function only.”
What data can TAU generate?

- What was the time spent exclusively and inclusively in each function? template? (based on instantiations)
- How many times was each function called?
- How many profiled functions (subroutines) did it call?
- What is the mean time/call for each function?
- What was the mean time spent in a function over all nodes? contexts? threads?
- For each invocation of a function, what was the exclusive and inclusive time spent in it? (Trace)
- What is the standard deviation of exclusive time? (Statistics)
- Can we replace “Time” by “flops”? Instructions issued? Cycles? Secondary data cache misses?( R10000 HW counters)
- Can we profile only Communication functions? Comm + IO? (Selective Profiling)
- Can we profile a set of statements (finer granularity) instead of functions? Can we profile blocks such as for loops?
Profi ling Templates and Functions in C++

template<class ForwardIterator>
Message& Message::put(ForwardIterator beg,
   ForwardIterator end) {
   // Code
}

int main(int argc, char **argv) {
   // Code
}

• Identify functions based on:

  1. Name: “main()”
  2. Type information: “int (int, char **)”
     e.g., “Message::put() Message (vector<int>::iterator, vector<int>::iterator )”

• Group related functions in a profile group e.g., TAU_FIELD, TAU_IO, TAU_USER

• Insert instrumentation macro in each function.
TAU Profiling API

- **TAU_PROFILE(func_name, type_string, profile_group);**
  Inserted in each function that is to be profiled. e.g.,
  ```c
  int main(int argc, char **argv){
    TAU_PROFILE("main()", "int (int, char **)",
      TAU_DEFAULT);
  }
  ```

- **TAU_TYPE_STRING(string_varname, string);**
- **TAU_PROFILE(func_name, string_varname, profile_group);**
- **CT(object);**
  For profiling template member functions.

- **TAU_PROFILE_TIMER(var, name, type, group);**
- **TAU_PROFILE_START(var);**
- **TAU_PROFILE_STOP(var);**
  To start and stop a timer. To time one or more statements in the code.

  ```c
  TAU_PROFILE_TIMER(timer1,"main-loop1", "int (int, char **)", TAU_USER);
  ...
  TAU_PROFILE_START(timer1);
  for(i=0; i < N; i++) { // loop1 profiled using timer1 var }
  TAU_PROFILE_STOP(timer1);
  ```

- **TAU_PROFILE_STMT(stmt);**
  To declare a variable that is used only during profiling.

- **TAU_PROFILE_INIT(argc, argv);**
- **TAU_PROFILE_SET_NODE(myNode);**
  To initialize profile groups and to set the current node id for each context

- **TAU_PROFILE_EXIT(const char *message);**
  To abort the program, and dump profiling data to disk.

- **TAU_TRACE_SENDMSG(type, destination, length);**
- **TAU_TRACE_RECVMSG(type, source, length);**
pprof showing template instantiations

<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></td>
<td>usec/call</td>
</tr>
<tr>
<td>-------</td>
<td>-----------</td>
<td>-----------</td>
<td>-------</td>
<td>--------</td>
<td>----------------</td>
</tr>
<tr>
<td>100.0</td>
<td>23,145</td>
<td>36:36.980</td>
<td>1</td>
<td>14378</td>
<td>2196980000 main</td>
</tr>
<tr>
<td></td>
<td>4:26.306</td>
<td>4:26.306</td>
<td>45792</td>
<td>0</td>
<td>5816 apply BrickExpression&lt;3U, LField&lt;double, 3U&gt;::iterator, LField&lt;double, 3U&gt;::iterator, OpExtrapolate&lt;double&gt;&gt;</td>
</tr>
<tr>
<td>11.2</td>
<td>234</td>
<td>4:06.693</td>
<td>100</td>
<td>196</td>
<td>2466930 assign(IndexedBareField) PETE_TBTree&lt;OpDivide, PETE_TBTree&lt;OpAdd, PETE_TBTree&lt;OpAdd, PETE_TBTree&lt;OpAdd, PETE_TBTree&lt;OpAdd, PETE_TBTree&lt;OpAdd, PETE_TBTree&lt;OpAdd, PETE_TBTree&lt;OpAdd, PETE_TBTree&lt;OpAdd, IndexedBareField&lt;index, 3U, 3U&gt;::PETE_Expr_t, IndexedBareField&lt;index, 3U, 3U&gt;::PETE_Expr_t&gt;::PETE_Expr_t, IndexedBareField&lt;index, 3U, 3U&gt;::PETE_Expr_t&gt;::PETE_Expr_t, IndexedBareField&lt;index, 3U, 3U&gt;::PETE_Expr_t&gt;::PETE_Expr_t, IndexedBareField&lt;index, 3U, 3U&gt;::PETE_Expr_t&gt;::PETE_Expr_t, IndexedBareField&lt;index, 3U, 3U&gt;::PETE_Expr_t&gt;::PETE_Expr_t, IndexedBareField&lt;index, 3U, 3U&gt;::PETE_Expr_t&gt;::PETE_Expr_t&gt;</td>
</tr>
<tr>
<td>10.9</td>
<td>3:59.772</td>
<td>3:59.772</td>
<td>96</td>
<td>0</td>
<td>2497625 apply BrickExpression&lt;3U, LField&lt;double, 3U&gt;::iterator, LField&lt;double, 3U&gt;::iterator, OpExtrapolate&lt;double&gt;&gt;</td>
</tr>
</tbody>
</table>
# Selective Profiling

Table 1: Selecting profile groups on the command line at runtime

<table>
<thead>
<tr>
<th>ProfileGroup</th>
<th>Description</th>
<th>Runtime Identifier</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAU_DEFAULT</td>
<td>All profiling groups enabled</td>
<td></td>
<td>--profile</td>
</tr>
<tr>
<td>TAU_MESSAGE</td>
<td>Message class includes MPI, PVM, ACLMPL</td>
<td>‘m’</td>
<td>--profile M</td>
</tr>
<tr>
<td>TAU_PETE</td>
<td>Portable Expression Template Engine</td>
<td>‘p’</td>
<td>--profile pete+message</td>
</tr>
<tr>
<td>TAU_VIZ</td>
<td>ACLVIZ</td>
<td>‘v’</td>
<td>--profile Pete+viz</td>
</tr>
<tr>
<td>TAU_ASSIGN</td>
<td>Assign expression</td>
<td>‘a’</td>
<td>--profile a+m</td>
</tr>
<tr>
<td>TAU_IO</td>
<td>IO functions</td>
<td>‘i’</td>
<td>--profile io</td>
</tr>
<tr>
<td>TAU_FIELD</td>
<td>Field functions</td>
<td>‘f’</td>
<td>--profile field+viz+assign</td>
</tr>
<tr>
<td>TAU_LAYOUT</td>
<td>Field Layout</td>
<td>‘l’</td>
<td>--profile layout</td>
</tr>
<tr>
<td>TAU_SPARSE</td>
<td>Sparse Index</td>
<td>‘s’</td>
<td>--profile sparse+assign</td>
</tr>
<tr>
<td>TAU_DOMAINMAP</td>
<td>Domain Map</td>
<td>‘d’</td>
<td>--profile i+d+viz</td>
</tr>
<tr>
<td>TAUUTILITY</td>
<td>Utility</td>
<td>‘ut’</td>
<td>--profile utils+io</td>
</tr>
<tr>
<td>TAU_USER</td>
<td>User</td>
<td>‘u’</td>
<td>--profile user+region</td>
</tr>
<tr>
<td>TAU_USER1</td>
<td>User1</td>
<td>‘1’</td>
<td>--profile 1+d</td>
</tr>
<tr>
<td>TAU_USER2</td>
<td>User2</td>
<td>‘2’</td>
<td>--profile 2+d</td>
</tr>
<tr>
<td>TAU_USER3</td>
<td>User3</td>
<td>‘3’</td>
<td>--profile 3+d</td>
</tr>
<tr>
<td>TAU_USER4</td>
<td>User4</td>
<td>‘4’</td>
<td>--profile 4+d</td>
</tr>
</tbody>
</table>

% mpirun -np 4 app --profile io+field+message
Hardware Performance Counters

- TAU approach: Function entry and exit points reset counters and read counter values from 64 bit register and accumulate these. Precise counts associated with each profiled function are reported. Counter lookup is expensive.

- SpeedShop: Program counter sampling done when a counter reaches a threshold. Kernel can preload the 64 bit register with some initial value. Interrupt received every “n” counter overflows.

- libperfex API provided by SGI is used.

- Configuration:
  
  `% ./configure -PROFILECOUNTERS
  % make install
  % setenv T5_EVENT0 26
  # profile secondary data cache misses
  % mpirun -np 4 app --profile io+field
SGI R10000 Performance Counters

Table 1: Values of T5_EVENT0 environment variable

<table>
<thead>
<tr>
<th>Quantity Profiled</th>
<th>Quantity Profiled</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Cycles</td>
<td>16 Graduated cycles</td>
</tr>
<tr>
<td>1 Issued instructions</td>
<td>17 Graduated instructions</td>
</tr>
<tr>
<td>2 Issued loads</td>
<td>18 Graduated loads</td>
</tr>
<tr>
<td>3 Issued stores</td>
<td>19 Graduated stores</td>
</tr>
<tr>
<td>4 Issued store conditionals</td>
<td>20 Graduated store conditionals</td>
</tr>
</tbody>
</table>
| 5 Failed store conditionals                           | 21 Graduated floating point instruc-
|                                                       | tions                             |
| 6 Decoded branches                                    | 22 Quadwords written back from pri-
|                                                       | mary data cache                    |
| 7 Quadwords written back from scache                  | 23 TLB misses                      |
| 8 Correctable scache data array ECC errors            | 24 Mispredicted branches           |
| 9 Primary instruction cache misses                    | 25 Primary data cache misses       |
| 10 Secondary instruction cache misses                 | 26 Secondary data cache misses     |
| 11 Instruction misprediction from scache way prediction table | 27 Data misprediction from scache way prediction table |
| 12 External interventions                             | 28 External intervention hits in scache |
| 13 External invalidations                             | 29 External invalidation hits in scache |
| 14 Virtual coherency conditions                       | 30 Store/prefetch exclusive to clean block in scache |
| 15 Graduated instructions                             | 31 Store/prefetch exclusive to shared block in scache |
Tuning and Analysis Utilities
http://www.acl.lanl.gov/tau

TAU Portable Profiling and Tracing

Stage I:
Instrumentation

C++ source
instrument
compile
run

instrumented
program

profile
profile
groups
data base
statistics

runtime library modules
data
data files
function
callstack
hardware
counters
user level
timers

profile
profile
data files
function
callstack
hardware
counters
user level
timers

Stage II:
Runtime Data Generation

trace
profile
runtime library modules

Stage III:
Performance Analysis

post-process:
merge and convert

pprof
RACY profile browser

ASCII
report

Vampir

event traces, event
table

pprof
RACY profile browser
**pprof - Profile Data Sorter and Display**

<table>
<thead>
<tr>
<th>Time</th>
<th>Exclusive</th>
<th>Inclusive</th>
<th>Call</th>
<th>Subs</th>
<th>Inclusive Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0</td>
<td>23.145</td>
<td>36.36.990</td>
<td>1</td>
<td>14378</td>
<td>main int(int,char*)[]</td>
</tr>
<tr>
<td>12.1</td>
<td>4:26.306</td>
<td>4:26.306</td>
<td>45792</td>
<td>0</td>
<td>apply ErickExpression3U, LField&lt;double*&gt;</td>
</tr>
<tr>
<td>11.2</td>
<td>234</td>
<td>4:06.693</td>
<td>100</td>
<td>196</td>
<td>assign(IndexedBareField), PTE_TETree&lt;Cp*</td>
</tr>
<tr>
<td>10.9</td>
<td>3:59.772</td>
<td>3:59.772</td>
<td>96</td>
<td>0</td>
<td>apply ErickExpression3U, LField&lt;double&gt;</td>
</tr>
<tr>
<td>7.7</td>
<td>576</td>
<td>2:48.948</td>
<td>3705</td>
<td>23775</td>
<td>Field::fillGuardCells, TechMatField&lt;double&gt;</td>
</tr>
<tr>
<td>6.2</td>
<td>201</td>
<td>2:16.607</td>
<td>10</td>
<td>20</td>
<td>13660700 assign(IndexedBareField), PTE_TETree&lt;Cp*</td>
</tr>
<tr>
<td>6.1</td>
<td>2:14.165</td>
<td>2:14.165</td>
<td>10</td>
<td>0</td>
<td>13416500 apply ErickExpression3U, LField&lt;Vektor&gt;</td>
</tr>
</tbody>
</table>

---


- **-c**: Sort according to number of Calls
- **-b**: Sort according to number of subroutines called by a function
- **-m**: Sort according to Milliseconds (exclusive time total)
- **-t**: Sort according to Total milliseconds (inclusive time total) (default)
- **-e**: Sort according to Exclusive time per call (mssec/call)
- **-l**: Sort according to Inclusive time per call (total msec/call)
- **-y**: Sort according to Standard Deviation (excl usec)
- **-r**: Reverse sorting order
- **-s**: print only Summary profile information
- **-n <num>**: print only first <num> number of functions
- **-f <filename>**: specify full path and Filename without node ids
- **-l**: List all functions and exit
- **[node numbers]**: prints only info about all contexts/threads of given node numbers
Routine Access and Data Display (RACY)
RACY
TAU Tracing

• Tracing library can be used with or without profiling.

• Reuses TAU profiling instrumentation for function entry/exit.

• Shows details of message passing and function entry/exit.

• Converts traces from TAU to:

  1. ASCII text format
  2. VAMPIR trace format
  3. ALOG (Upshot) trace format
  4. SDDF (Pablo) trace format
Trace of a POOMA sparse index application displayed in VAMPIR
Vampir and TAU

VAMPIR displays templates in C++ using TAU instrumentation
VAMPIR and TAU

Comparison of two nodes for different TAU groups in VAMPIR
Profiling Expression Templates

(Todd Veldhuizen, Lawrence Berkeley National Laboratory)

- Blitz++ numerical library
- Expression templates
Tracing Expression Templates

(Todd Veldhuizen, Lawrence Berkeley National Laboratory)
TAU: Current Focus

- KAI (Kuck and Associates) - static program database for static TAU tools (callgraph display, file and class browsers, class hierarchy browsers)
- KAI - Profiling instrumentation triggered at runtime
- EDG - Automatic insertion of profiling instrumentation
- Profiling expression templates (Todd Veldhuizen, Blitz++)
- New TAU tools: template browser, performance callstack monitor, performance callstack trace
- VAMPIR - binary trace data generation, new tools to merge and convert to binary trace format
- Support for other trace visualization tools (VAMPIR, ALOG, SDDF formats supported)
Performance Callstack Trace
(under implementation, prototype released in ver 2.2)
Performance Callstack Trace
(under implementation)
**TAU Status**

- TAU profiling and tracing instrumentation API defined and implemented and available for downloading at: [http://www.acl.lanl.gov/tau](http://www.acl.lanl.gov/tau)
- Profiling deliverable: profiling library, profile data visualization tools - RACY, pprof
- Trace deliverable: tracing library, trace merging and conversion tools
- Support for trace visualization with VAMPIR (PALLAS, KFA Jülich)
- Instrumentation of SciTL components:
  - POOMA (Parallel Object Oriented Methods and Applns)
  - Tecolote
  - ACLMPL (ACL Message Passing Library)
  - A++/P++ (Array library and Parallel Array Library)
  - PADRE (Parallel Asynchronous Data Routing Env.)
  - ACLVIS (ACL Visualization Library)
  - PETE (Portable Expression Template Engine)
  - MC++ (Montecarlo Simulation)
  - among others...
Conclusion

• TAU provides the building blocks for implementing different performance profiling models and features.

• Profiling overhead depends on modules selected.

• Common API for profiling and tracing.

• Performance visualization tools.

• Tracing and profiling portable across multiple compilers (KAI, SGI CC, gnu g++), platforms (SGI Origin 2000, ASCI Red Intel Teraflop, Cray T3E, Linux PC Cluster, Solaris, HP)

<table>
<thead>
<tr>
<th>Statistics</th>
<th>Counters</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lightweight profiling core.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Selective profiling</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>