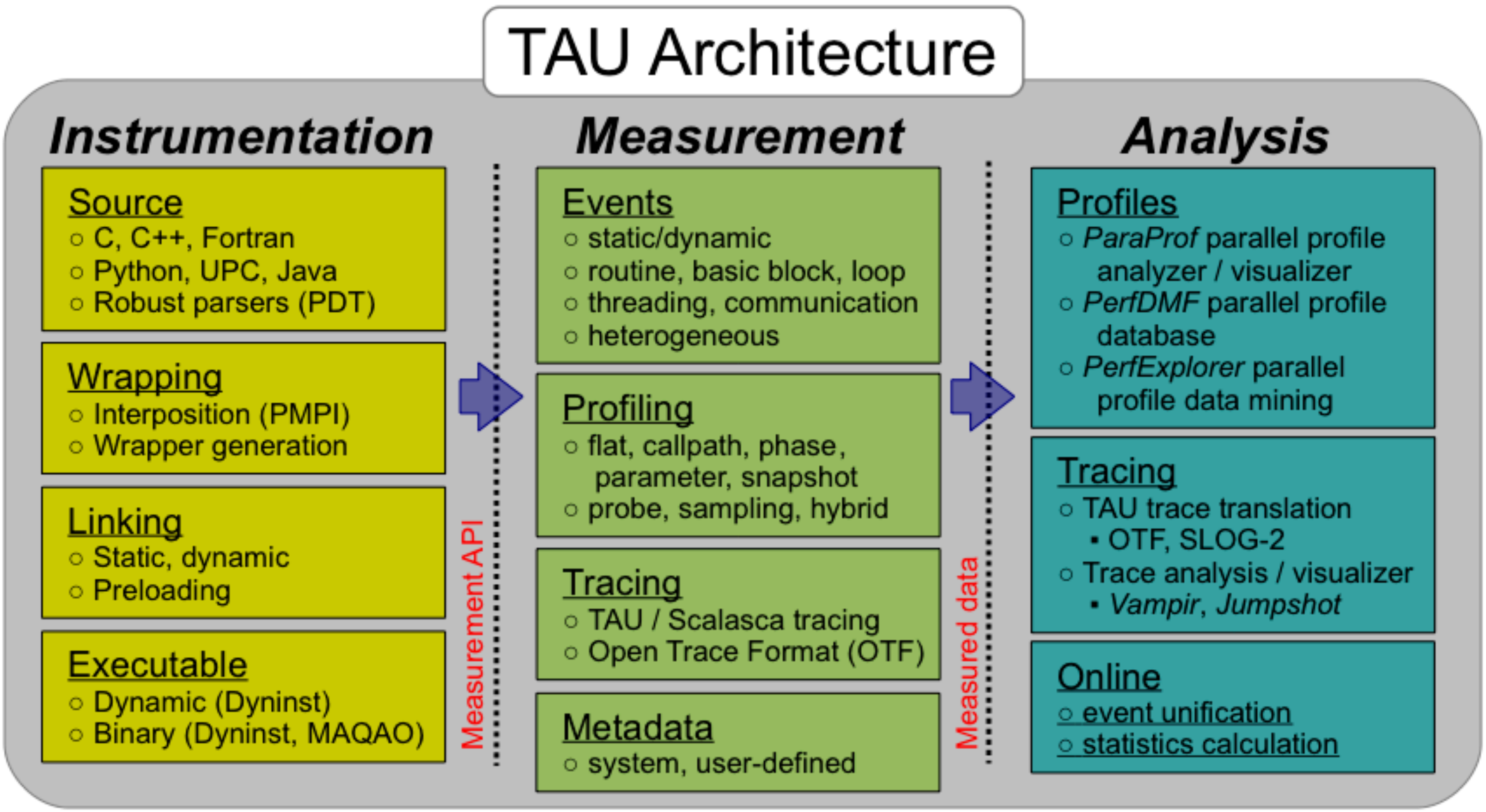
TAU Performance System

Tuning and Analysis Utilities

Sameer Shende
Performance Research Lab, University of Oregon
http://TAU.uoregon.edu

- Parallel performance framework and toolkit
  - Supports all HPC platforms, compilers, runtime system
  - Provides portable instrumentation, measurement, analysis



TAU Performance System®

- Instrumentation
  - Fortran, C++, C, UPC, Java, Python, Chapel
  - Automatic instrumentation
- Measurement and analysis support
  - MPI, OpenSHMEM, ARMCI, PGAS, DMAPP
  - pthreads, OpenMP, hybrid, other thread models
  - GPU, CUDA, OpenCL, OpenACC
  - Parallel profiling and tracing
  - Use of Score-P for native OTF2 and CUBEX generation
  - Efficient callpath proflles and trace generation using Score-P
- Analysis
  - Parallel profile analysis (ParaProf), data mining (PerfExplorer)
  - Performance database technology (PerfDMF, TAUdb)
  - 3D profile browser

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)

- Integrates with 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?

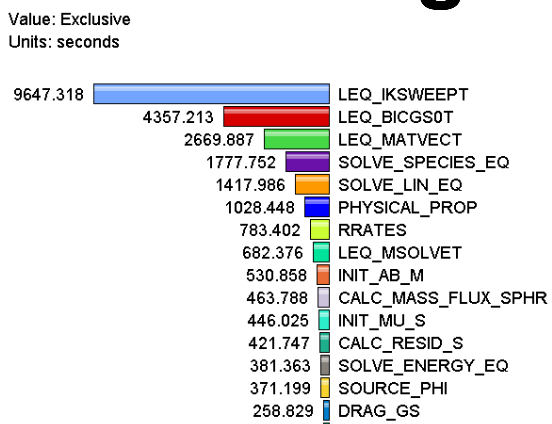
What Can TAU Do?

- 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

What does TAU support?

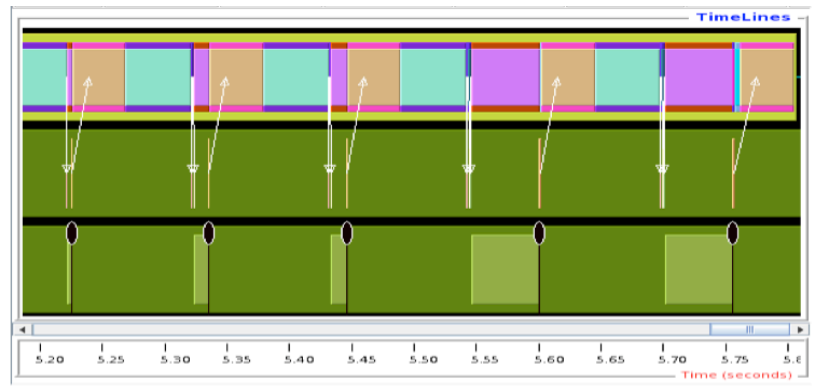
C/C++
UPC
OpenCL
GPI
Fortran
OpenACC
Java
MPI
pthreads
Intel MIC
OpenMP
Intel
LLVM
PGI
Cray
Sun
MinGW
Linux
Windows
AIX
Insert yours here
BlueGene
Fujitsu
ARM
MPC
OS X

Profiling



- **Profiling** shows you **how much** (total) time was spent in each routine

Tracing

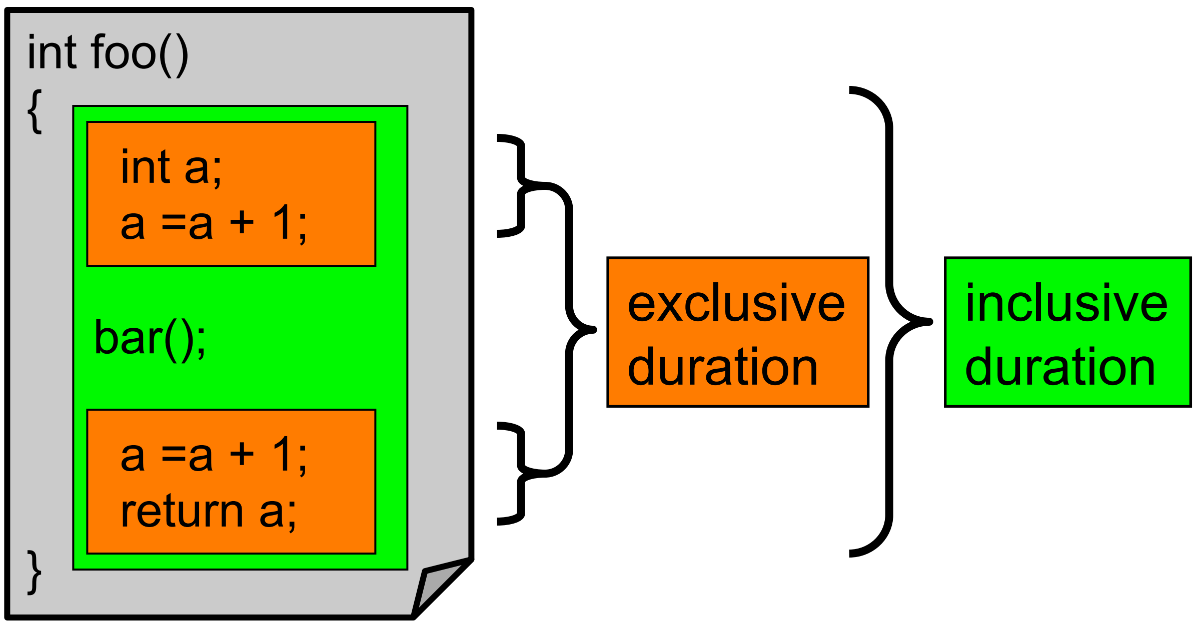


- Tracing shows you when the events take place on a timeline

- 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.

Inclusive vs. Exclusive Measurements

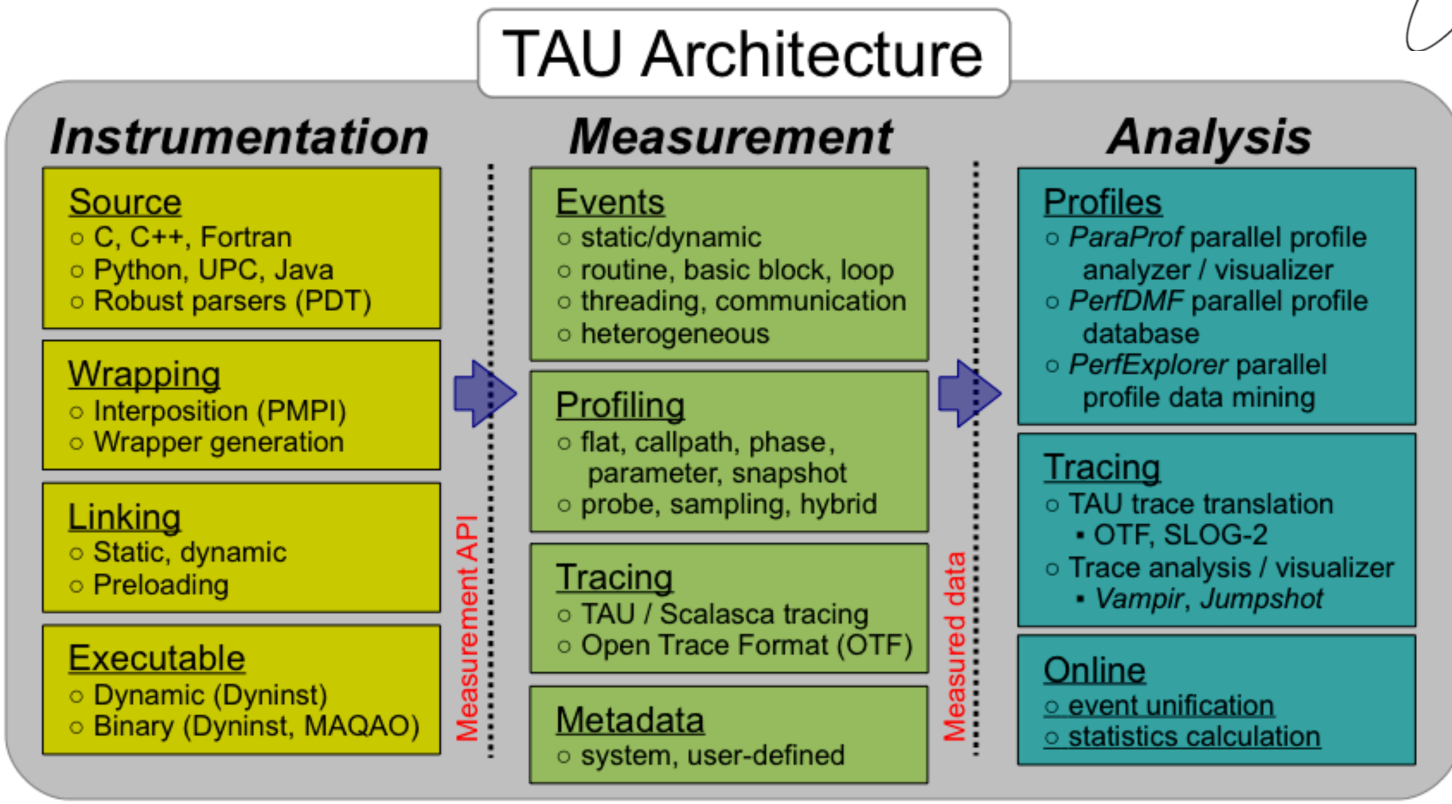
- Performance with respect to code regions
- Exclusive measurements for region only
- Inclusive measurements includes child regions



Availability on New Systems

- Intel compilers with Intel MPI on Intel Xeon Phi^{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

TAU Architecture and Workflow



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

```
NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0        0.187        1,105           1          44    1105659 int main(int, char **) C
 93.2        1,030        1,030           1           0    1030654 MPI_Init()
  5.9        0.879           65          40         320       1637 void func(int, int) C
  4.6           51           51          40           0       1277 MPI_Barrier()
  1.2           13           13         120           0        111 MPI_Recv()
  0.8            9            9           1           0       9328 MPI_Finalize()
  0.0        0.137        0.137         120           0          1 MPI_Send()
  0.0        0.086        0.086          40           0          2 MPI_Bcast()
  0.0        0.002        0.002           1           0          2 MPI_Comm_size()
  0.0        0.001        0.001           1           0          1 MPI_Comm_rank()
---------------------------------------------------------------------------------------

USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0
---------------------------------------------------------------------------------------
NumSamples   MaxValue   MinValue  MeanValue  Std. Dev.  Event Name
---------------------------------------------------------------------------------------
       365  5.138E+04      44.39   3.09E+04  1.234E+04  Heap Memory Used (KB) : Entry
       365  5.138E+04       2064  3.115E+04    1.21E+04  Heap Memory Used (KB) : Exit
        40         40         40         40          0  Message size for broadcast
---------------------------------------------------------------------------------------
                                                              27,1          1%
```

Interval events show ***duration***

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

% export TAU_CALLPATH_DEPTH=0
% export TAU_TRACK_HEAP=1

Atomic Events and Context Events

```
xterm
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0        0.253        1,106           1          44    1106701 int main(int, char **) C
 93.2        1,031        1,031           1           0    1031311 MPI_Init()
  6.0            1           66          40         320       1650 void func(int, int) C
  5.7           63           63          40           0       1588 MPI_Barrier()
  0.8            9            9           1           0       9119 MPI_Finalize()
  0.1            1            1         120           0         10 MPI_Recv()
  0.0        0.141        0.141         120           0          1 MPI_Send()
  0.0        0.085        0.085          40           0          2 MPI_Bcast()
  0.0        0.001        0.001           1           0          1 MPI_Comm_size()
  0.0            0            0           1           0          0 MPI_Comm_rank()
---------------------------------------------------------------------------------------

USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0
---------------------------------------------------------------------------------------
NumSamples   MaxValue   MinValue  MeanValue  Std. Dev.  Event Name
---------------------------------------------------------------------------------------
        40         40         40         40          0  Message size for broadcast
       365  5.139E+04      44.39  3.091E+04  1.234E+04  Heap Memory Used (KB) : Entry
        40  5.139E+04       3097  3.114E+04  1.227E+04  Heap Memory Used (KB) : Entry : MPI_Barrier()
        40  5.139E+04   1.13E+04  3.134E+04  1.187E+04  Heap Memory Used (KB) : Entry : MPI_Bcast()
         1       2067       2067       2067          0  Heap Memory Used (KB) : Entry : MPI_Comm_rank()
         1       2066       2066       2066          0  Heap Memory Used (KB) : Entry : MPI_Comm_size()
         1  5.139E+04  5.139E+04  5.139E+04  0.0006905  Heap Memory Used (KB) : Entry : MPI_Finalize()
         1      57.56      57.56      57.56          0  Heap Memory Used (KB) : Entry : MPI_Init()
       120  5.139E+04   1.13E+04  3.134E+04  1.187E+04  Heap Memory Used (KB) : Entry : MPI_Recv()
       120  5.139E+04  1.129E+04  3.134E+04  1.187E+04  Heap Memory Used (KB) : Entry : MPI_Send()
         1      44.39      44.39      44.39          0  Heap Memory Used (KB) : Entry : int main(int, char **) C
        40  5.036E+04       2068  3.011E+04  1.227E+04  Heap Memory Used (KB) : Entry : void func(int, int) C
                                                                              4,9            1%
```

Atomic events

Context events are atomic events with executing context

```
% export TAU_CALLPATH_DEPTH=1
% export TAU_TRACK_HEAP=1
```

Controls depth of executing context shown in profiles

Context Events with Callpath

xterm

```
NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0        0.357        1,114           1          44    1114040 int main(int, char **) C
 92.6        1,031        1,031           1           0    1031066 MPI_Init()
  6.7           72           74          40         320       1865 void func(int, int) C
  0.7            8            8           1           0       8002 MPI_Finalize()
  0.1            1            1         120           0         12 MPI_Recv()
  0.1        0.608        0.608          40           0         15 MPI_Barrier()
  0.0        0.136        0.136         120           0          1 MPI_Send()
  0.0        0.095        0.095          40           0          2 MPI_Bcast()
  0.0        0.001        0.001           1           0          1 MPI_Comm_size()
  0.0            0            0           1           0          0 MPI_Comm_rank()
---------------------------------------------------------------------------------------

USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0
---------------------------------------------------------------------------------------
NumSamples   MaxValue   MinValue  MeanValue  Std. Dev.  Event Name
---------------------------------------------------------------------------------------
       365  5.139E+04      44.39  3.091E+04  1.234E+04  Heap Memory Used (KB) : Entry
         1      44.39      44.39      44.39          0  Heap Memory Used (KB) : Entry : int main(int, char **) C
         1       2068       2068       2068          0  Heap Memory Used (KB) : Entry : int main(int, char **) C => MPI_Comm_rank()
         1       2066       2066       2066          0  Heap Memory Used (KB) : Entry : int main(int, char **) C => MPI_Comm_size()
         1  5.139E+04  5.139E+04  5.139E+04          0  Heap Memory Used (KB) : Entry : int main(int, char **) C => MPI_Finalize()
         1      57.58      57.58      57.58          0  Heap Memory Used (KB) : Entry : int main(int, char **) C => MPI_Init()
        40  5.036E+04       2069  3.011E+04  1.228E+04  Heap Memory Used (KB) : Entry : int main(int, char **) C => void func(int, int) C
        40  5.139E+04       3098  3.114E+04  1.227E+04  Heap Memory Used (KB) : Entry : void func(int, int) C => MPI_Barrier()
        40  5.139E+04    1.13E+04  3.134E+04  1.187E+04  Heap Memory Used (KB) : Entry : void func(int, int) C => MPI_Bcast()
       120  5.139E+04    1.13E+04  3.134E+04  1.187E+04  Heap Memory Used (KB) : Entry : void func(int, int) C => MPI_Recv()
       120  5.139E+04    1.13E+04  3.134E+04  1.187E+04  Heap Memory Used (KB) : Entry : void func(int, int) C => MPI_Send()
       365  5.139E+04       2065  3.116E+04   1.21E+04  Heap Memory Used (KB) : Exit
                                                                                      3,7          1%
```

% export TAU_CALLPATH_DEPTH=2
% export TAU_TRACK_HEAP=1

Callpath shown on context events

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

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 (configuration file) 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-mpi-pdt

% export TAU_OPTIONS='-optVerbose …' (see tau_compiler.sh)

% export PATH=$TAU_ROOT/x86_64/bin:$PATH

% export TAU=$TAU_ROOT/x86_64/lib

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

% mpif90 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)

Automatic Source Instrumentation using PDT

```
% module load UNITE VI-HPS-TW; ls $TAU/Makefile.*
Makefile.tau-icpc
Makefile.tau-icpc-cupti-pdt
Makefile.tau-icpc-mpi-cupti-pdt
Makefile.tau-icpc-mpi-pdt
Makefile.tau-icpc-mpi-pdt-openmp
Makefile.tau-icpc-mpi-pdt-openmp-opari
Makefile.tau-icpc-mpi-pthread-pdt
Makefile.tau-icpc-ompt-mpi-pdt-openmp
Makefile.tau-icpc-papi-mpi-pdt-openmp-opari-scorep
Makefile.tau-icpc-papi-mpi-pdt-scorep
Makefile.tau-icpc-papi-ompt-mpi-pdt-openmp
Makefile.tau-mpc250-mpc-mpi-pdt
```

- For an MPI+F90 application with Intel MPI, you may choose `Makefile.tau-mpi-pdt`
  - Supports MPI instrumentation & PDT for automatic source instrumentation

```
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-mpi-pdt
% tau_f90.sh matmult.f90 -o matmult
% mpirun -np 4 ./matmult
% paraprof
```

Using TAU with Score-P

```
% export TAU=$TAU_ROOT/x86_64/lib
% export TAU_MAKEFILE=$TAU/Makefile.tau-icpc-papi-mpi-pdt-openmp-opari-scorep
% export OMP_NUM_THREADS=10
% make CC=tau_cc.sh CXX=tau_cxx.sh F90=tau_f90.sh

% mpirun -np 4  ./matmult

% cd score*; paraprof profile.cubex &
```

MIC Architecture

```
% export TAU=$TAU_ROOTDIR/mic_linux/lib
% ls $TAU/Makefile.*
Makefile.tau-intelmpi-icpc-mpi-pdt
Makefile.tau-intelmpi-icpc-papi-mpi-pdt
Makefile.tau-intelmpi-icpc-papi-mpi-pdt-openmp-opari
```

- For an MPI+F90 application with Intel MPI, you may choose

```
Makefile.tau-intelmpi-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`
- `% idev -m 50;`
- `% export MIC_PPN=6`
- `% export MIC_OMP_NUM_THREADS=10`
- `ibrun.symm -m ./matrix`
- `% paraprof`

- Installing PDT:
  - wget http://tau.uoregon.edu/pdt_lite.tgz
  - ./configure –prefix=<dir>; make ; make install
- Installing TAU:
  - wget http://tau.uoregon.edu/tau.tgz
  - ./configure –arch=x86_64 -bfd=download -pdt=<dir> -papi=<dir> ...
  - For MIC:
  - ./configure –arch=mic_linux –pdt=<dir> -pdt_c++=g++ -papi=dir …
  - make install
- Using TAU:
  - export TAU_MAKEFILE=<taudir>/x86_64/ lib/Makefile.tau-<TAGS>
  - 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

Environment Variable	Default	Description
TAU_TRACE	0	Setting to 1 turns on tracing
TAU_CALLPATH	0	Setting to 1 turns on callpath profiling
TAU_TRACK_MEMORY_LEAKS	0	Setting to 1 turns on leak detection (for use with –optMemDbg or tau_exec)
TAU_MEMDBG_PROTECT_ABOVE	0	Setting to 1 turns on bounds checking for dynamically allocated arrays. (Use with –optMemDbg or tau_exec –memory_debug).
TAU_CALLPATH_DEPTH	2	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)
TAU_TRACK_IO_PARAMS	0	Setting to 1 with –optTrackIO or tau_exec –io captures arguments of I/O calls
TAU_TRACK_SIGNALS	0	Setting to 1 generate debugging callstack info when a program crashes
TAU_COMM_MATRIX	0	Setting to 1 generates communication matrix display using context events
TAU_THROTTLE	1	Setting to 0 turns off throttling. Enabled by default to remove instrumentation in lightweight routines that are called frequently
TAU_THROTTLE_NUMCALLS	100000	Specifies the number of calls before testing for throttling
TAU_THROTTLE_PERCALL	10	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
TAU_COMPENSATE	0	Setting to 1 enables runtime compensation of instrumentation overhead
TAU_PROFILE_FORMAT	Profile	Setting to “merged” generates a single file. “snapshot” generates xml format
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= '-optCompInst -optVerbose'

- If your Fortran code uses C preprocessor directives (#include, #ifdef, #endif):

  % export TAU_OPTIONS= '-optPreProcess -optVerbose'

- 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
```

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
- MAQAO:

```
% tau_rewrite -T [tags] [-f select.tau] a.out
    [-o] a.inst
```

- Dyninst:

```
% tau_run -T [tags] [-f select.tau]  a.out -o a.inst
```

- Pebil:

```
% tau_pebil_rewrite -T [tags] [-f select.tau] a.out
-o a.inst
```

- Execute the application to get measurement data:

```
% mpirun -np 4 ./a.inst
```

Selective Instrumentation in tau_rewrite

- `tau_rewrite -T icpc,mpi,pdt a.out a.inst`
- `cat select.tau`

```
BEGIN_EXCLUDE_LIST
compute#
foo
END_EXCLUDE_LIST
```

- `tau_rewrite -f select.tau -T icpc,mpi,pdt a.out a.inst`
- `mpirun -np 4 ./a.inst`

Rewrites a.out and runs the instrumented code. Routine names must match profiles.

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

Download TAU from U. Oregon

http://tau.uoregon.edu

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

Free download, open source, BSD license