TAU Performance System
(ACTS Workshop LBL)

Sameer Shende, Allen D. Malony
University of Oregon
{sameer, malony}@cs.uoregon.edu
Research Motivation

- Tools for performance problem solving
  - Empirical-based performance optimization process
  - Performance technology concerns

Performance Technology
- Experiment management
- Performance storage

Performance Tuning
- Hypotheses

Performance Diagnosis
- Properties

Performance Experimentation
- Characterization

Performance Observation

Performance Technology
- Instrumentation
- Measurement
- Analysis
- Visualization
Outline of Talk

- Performance problem solving
  - Scalability, productivity, and performance technology
  - Application-specific and autonomic performance tools
- TAU parallel performance system and advances
- Performance data management and data mining
  - Performance Data Management Framework (PerfDMF)
  - PerfExplorer
- Multi-experiment case studies
  - Clustering analysis
- Future work and concluding remarks
TAU Performance System

- Tuning and Analysis Utilities (13+ year project effort)
- Performance system framework for HPC systems
  - Integrated, scalable, flexible, and parallel
- Targets a general complex system computation model
  - Entities: nodes / contexts / threads
  - Multi-level: system / software / parallelism
  - Measurement and analysis abstraction
- Integrated toolkit for performance problem solving
  - Instrumentation, measurement, analysis, and visualization
  - Portable performance profiling and tracing facility
  - Performance data management and data mining
- University of Oregon, Research Center 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(ENTER, 1);
    ...
    trace(SEND, B);
    send(B, tag, buf);
    ...
    trace(EXIT, 1);
}
```

CPU B:
```c
void slave {
    trace(ENTER, 2);
    ...
    recv(A, tag, buf);
    ...
    trace(RECV, A);
    ...
    trace(EXIT, 2);
}
```

Event definition:
```
1 master
2 slave
3 ...
```

MONITOR

- CPU A: `timestamp`
- CPU B: `timestamp`

Event log:
```
58 A ENTER 1
60 B ENTER 2
62 A SEND B
64 A EXIT 1
68 B RECV A
69 B EXIT 2
...```

ACTS Workshop 2005

TAU Performance System
Event Tracing: “Timeline” Visualization

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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>...</td>
<td></td>
<td></td>
</tr>
<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>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Main** (green)
- **Master** (red)
- **Slave** (blue)

Time:
- 58
- 60
- 62
- 64
- 66
- 68
- 70

Diagram showing the timeline visualization of events between processes A and B.
TAU Parallel Performance System Goals

- 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
- Support for performance mapping
- Support for object-oriented and generic programming
- Integration in complex software, systems, applications
Advances in TAU Instrumentation

- Source instrumentation
  - *Program Database Toolkit (PDT)*
    - automated Fortran 90/95 support (Cleanscape Flint parser)
    - statement level support in C/C++ (Fortran soon)
  - *TAU_COMPILER* to automate instrumentation process
  - Automatic proxy generation for component applications
    - automatic CCA component instrumentation
  - Python instrumentation and automatic instrumentation

- Continued integration with dynamic instrumentation
- Update of OpenMP instrumentation (POMP2)
- Selective instrumentation and overhead reduction
- Improvements in performance mapping instrumentation
Program Database Toolkit (PDT)

Application / Library

C / C++ parser

C / C++ IL analyzer

Fortran parser F77/90/95

IL analyzer

Program Database Files

DUCTAPE

Fortran IL analyzer

PDBhtml

Program documentation

SILOON

Application component glue

CHASM

C++ / F90/95 interoperability

TAU_instr

Automatic source instrumentation
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 (e.g., size of memory allocated/freed)
  - Selection of event statistics

- **Support definition of “semantic” entities for mapping**

- **Support for event groups**

- **Instrumentation optimization (eliminate instrumentation in lightweight routines)**
TAU Instrumentation

- Flexible instrumentation mechanisms at multiple levels
  - Source code
    - manual (TAU API, TAU Component API)
    - 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)
  - Proxy Components
Using TAU – A tutorial

- Configuration
- Instrumentation
  - Manual
  - MPI – Wrapper interposition library
  - PDT- Source rewriting for C,C++, F77/90/95
  - OpenMP – Directive rewriting
  - Component based instrumentation – Proxy components
  - Binary Instrumentation
    - DyninstAPI – Runtime Instrumentation/Rewriting binary
    - Java – Runtime instrumentation
    - Python – Runtime instrumentation
- Measurement
- Performance Analysis
configure [OPTIONS]

- {-c++=<CC>, -cc=<cc>} Specify C++ and C compilers
- {-pthread, -sproc} Use pthread or SGI sproc threads
- -openmp Use OpenMP threads
- -jdk=<dir> Specify Java instrumentation (JDK)
- -opari=<dir> Specify location of Opari OpenMP tool
- -papi=<dir> Specify location of PAPI
- -pdt=<dir> Specify location of PDT
- -dyninst=<dir> Specify location of DynInst Package
- -mpi[inc/lib]=<dir> Specify MPI library instrumentation
- -shmemp[inc/lib]=<dir> Specify PSHMEM library instrumentation
- -python[inc/lib]=<dir> Specify Python instrumentation
- -epilog=<dir> Specify location of EPILOG
- -slog2[=<dir>] Specify location of SLOG2/Jumpshot
- -vtf=<dir> Specify location of VTF3 trace package
- -arch=<architecture> Specify architecture explicitly (bgl,ibm64,ibm64linux…)}
configure [OPTIONS]

-TRACE  Generate binary TAU traces
-PROFILE (default)  Generate profiles (summary)
-PROFILECALLPATH  Generate call path profiles
-PROFILEPHASE  Generate phase based profiles
-PROFILEMEMORY  Track heap memory for each routine
-PROFILEHEADROOM  Track memory headroom to grow
-MULTIPLECOUNTERS  Use hardware counters + time
-COMPENSATE  Compensate timer overhead
-CPUTIME  Use usertime+system time
-PAPIWALLCLOCK  Use PAPI’s wallclock time
-PAPIVIRTUAL  Use PAPI’s process virtual time
-SGITIMERS  Use fast IRIX timers
-LINUXTIMERS  Use fast x86 Linux timers
TAU Measurement Configuration – Examples

- `.configure -c++=xlC_r -pthread`
  - Use TAU with xlC_r and pthread library under AIX
  - Enable TAU profiling (default)

- `.configure -TRACE -PROFILE`
  - Enable both TAU profiling and tracing

- `.configure -c++=xlC_r -cc=xlc_r -papi=/usr/local/packages/papi
  -pdt=/usr/local/pdtoolkit-3.4 -arch=ibm64
  -mpiinc=/usr/lpp/ppe.poe/include
  -mpilib=/usr/lpp/ppe.poe/lib -MULTIPLECOUNTERS`
  - Use IBM’s xlC_r and xlc_r compilers with PAPI, PDT, MPI packages and multiple counters for measurements

- Typically configure multiple measurement libraries

- Each configuration creates a unique <arch>/lib/Makefile.tau-<options> stub makefile that corresponds to the configuration options specified. E.g.,
  - `/san/cca/tau/tau-2.14.7/x86_64/lib/Makefile.tau-icpc-mpi-pdt`
  - `/san/cca/tau/tau-2.14.7/x86_64/lib/Makefile.tau-icpc-mpi-pdt-trace`
tau-2.x>./tau_setup
Configuration Parameters in Stub Makefiles

- Each TAU Stub Makefile resides in <tau><arch>/lib directory
- Variables:
  - TAU_CXX: Specify the C++ compiler used by TAU
  - TAU_CC, TAU_F90: Specify the C, F90 compilers
  - TAU_DEFS: Defines used by TAU. Add to CFLAGS
  - TAU_LDFLAGS: Linker options. Add to LDFLAGS
  - TAU_INCLUDE: Header files include path. Add to CFLAGS
  - TAU_LIBS: Statically linked TAU library. Add to LIBS
  - TAU_SHLIBS: Dynamically linked TAU library
  - TAU_MPI_LIBS: TAU’s MPI wrapper library for C/C++
  - TAU_MPI_FLIBS: TAU’s MPI wrapper library for F90
  - TAU_FORTRANLIBS: Must be linked in with C++ linker for F90
  - TAU_CXXLIBS: Must be linked in with F90 linker
  - TAU_INCLUDE_MEMORY: Use TAU’s malloc/free wrapper lib
  - TAU_DISABLE: TAU’s dummy F90 stub library
  - TAU_COMPILER: Instrument using tau_compiler.sh script

Note: Not including TAU_DEFS in CFLAGS disables instrumentation in C/C++ programs (TAU_DISABLE for f90).
Using TAU

Step 1: Configure and install TAU:

```
% configure -pdt=<dir> -mpiinc=<dir> -mpilib=<dir>
   -c++=icpc -cc=icc -fortran=intel
% make clean; make install

Builds <taudir>/<arch>/lib/Makefile.tau-<options>
% set path=($path <taudir>/<arch>/bin)
```

Step 2: Choose target stub Makefile

```
% setenv TAU_MAKEFILE
   /san/cca/tau/tau-2.14.7/x86_64/lib/Makefile.tau-icpc-mpi-pdt
% setenv TAU_OPTIONS `--optVerbose --optKeepFiles`
(see tau_compiler.sh for all options)
```

Step 3: Use tau_f90.sh, tau_cxx.sh and tau_cc.sh as the F90, C++ or C compilers respectively.

```
% tau_f90.sh -c app.f90
% tau_f90.sh app.o -o app -lm -lblas
```

Or use these in the application Makefile.
AutoInstrumentation using TAU_COMPILER

- $(TAU_COMPILER) stub Makefile variable in 2.14+ release
- Invokes PDT parser, TAU instrumentor, compiler through \texttt{tau_compiler.sh} shell script
- Requires minimal changes to application Makefile
  - Compilation rules are not changed
  - User sets \texttt{TAU_MAKEFILE} and \texttt{TAU_OPTIONS} environment variables
  - User renames the compilers
    - F90=xlf90 to F90= \texttt{tau_f90.sh}
- Passes options from TAU stub Makefile to the four compilation stages
- Uses original compilation command if an error occurs
Tau_[cxx,cc,f90].sh – Improves Integration in Makefiles

OLD

#include /usr/tau-2.14/include/Makefile
CXX = mpCC
F90 = mpxlf90_r
PDTPARSE = $(PDTDIR)/
  $(PDTARCHDIR)/bin/cxxparse
TAUINSTR = $(TAUROOT)/$(CONFIG_ARCH)/
  bin/tau_instrumentor
CFLAGS = $(TAU_DEFS) $(TAU_INCLUDE)
LIBS = $(TAU_MPI_LIBS) $(TAU_LIBS)-lm
OBJS = f1.o f2.o f3.o ... fn.o
app: $(OBJS)
  $(CXX) $(LDFLAGS) $(OBJS) -o $@
  $(LIBS)
  .cpp.o:
    $(PDTPARSE) $<
    $(TAUINSTR) *.*.pdb $< -o
    *.*.i.cpp -f select.dat
    $(CC) $(CFLAGS) -c *.*.i.cpp

NEW

# set TAU_MAKEFILE and TAU_OPTIONS env vars
CXX = tau_cxx.sh
F90 = tau_f90.sh
CFLAGS =
LIBS = -lm
OBJJS = f1.o f2.o f3.o ... fn.o
app: $(OBJJS)
  $(CXX) $(LDFLAGS) $(OBJJS) -o $@
  $(LIBS)
  .cpp.o:
    $(CC) $(CFLAGS) -c $<
TAU_COMPILER Options

- **-optVerbose**  Turn on verbose debugging messages
- **-optPdtDir=""**  PDT architecture directory. Typically \$(PDTDIR)/\$(PDTARCHDIR)
- **-optPdtF95Opts=""**  Options for Fortran parser in PDT (f95parse)
- **-optPdtCOpts=""**  Options for C parser in PDT (cparse). Typically \$\(\text{TAU_MPI_INCLUDE}\) \$\(\text{TAU_INCLUDE}\) \$\(\text{TAU_DEFS}\)
- **-optPdtCxxOpts=""**  Options for C++ parser in PDT (cxxparse). Typically \$\(\text{TAU_MPI_INCLUDE}\) \$\(\text{TAU_INCLUDE}\) \$\(\text{TAU_DEFS}\)
- **-optPdtF90Parser=""**  Specify a different Fortran parser. For e.g., f90parse instead of f95parse
- **-optPdtUser=""**  Optional arguments for parsing source code
- **-opt TauInstr=""**  Specify location of tau_instrumentor. Typically \$\(\text{TAUROOT}/\$(CONFIG_ARCH)/bin/tau_instrumentor\)
- **-opt TauSelectFile=""**  Specify selective instrumentation file for tau_instrumentor
- **-opt Tau=""**  Specify options for tau_instrumentor
- **-opt Compile=""**  Options passed to the compiler. Typically \$\(\text{TAU_MPI_INCLUDE}\) \$\(\text{TAU_INCLUDE}\) \$\(\text{TAU_DEFS}\)
- **-opt Linking=""**  Options passed to the linker. Typically \$\(\text{TAU_MPI_FLIBS}\) \$\(\text{TAU_LIBS}\) \$\(\text{TAU_CXXLIBS}\)
- **-opt NoMpi**  Removes -l*mpi* libraries during linking (default)
- **-opt KeepFiles**  Does not remove intermediate .pdb and .inst.* files

**Example:**

```
% setenv TAU_OPTIONS `\-opt TauSelectFile=select.tau \-optVerbose \-optPdtCOpts="\-I/home \-DFOO"`
% tau_cxx.sh matrix.cpp \-o matrix \-lm
```
% tau_instrumentor
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
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)

\[ [\text{GroupName:}] \text{ Field } \text{ Operator } \text{ Number} \]

- **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
Optimizing Instrumentation Overhead: Examples

- #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
Instrumentation of OpenMP Constructs

- **OpenMP Pragma And Region Instrumentor**
- Source-to-Source translator to insert POMP calls around OpenMP constructs and API functions
- **Done:** Supports
  - Fortran77 and Fortran90, OpenMP 2.0
  - C and C++, OpenMP 1.0
  - POMP Extensions
  - EPILOG and TAU POMP implementations
  - Preserves source code information (*#line line file*)
- **Work in Progress:**
  - Investigating standardization through OpenMP Forum
- **KOJAK Project website** http://icl.cs.utk.edu/kojak
OpenMP API Instrumentation

- **Transform**
  - `omp_#_lock()` → `pomp_#_lock()`
  - `omp_#_nest_lock()` → `pomp_#_nest_lock()`

  `[ # = init | destroy | set | unset | test ]`

- **POMP version**
  - Calls omp version internally
  - Can do extra stuff before and after call
Example: !$OMP PARALLEL DO Instrumentation

call pomp_parallel_fork(d)

!$OMP PARALLEL other-clauses...
    call pomp_parallel_begin(d)
    call pomp_do_enter(d)
    !$OMP DO schedule-clauses, ordered-clauses, lastprivate-clauses
        do loop
    !$OMP END DO NOWAIT
    call pomp_barrier_enter(d)
    !$OMP BARRIER
    call pomp_barrier_exit(d)
    call pomp_do_exit(d)
    call pomp_parallel_end(d)

    !$OMP END PARALLEL DO
    call pomp_parallel_join(d)
Opari Instrumentation: Example

- OpenMP directive instrumentation

```c
pomp_for_enter(&omp_rd_2);
#line 252 "stommel.c"
#pragma omp for schedule(static) reduction(+: diff) private(j)
    firstprivate (a1,a2,a3,a4,a5) nowait
for( i=i1;i<=i2;i++) {
    for(j=j1;j<=j2;j++){
        new_psi[i][j]=a1*psi[i+1][j] + a2*psi[i-1][j] + a3*psi[i][j+1]
        + a4*psi[i][j-1] - a5*the_for[i][j];
        diff=diff+fabs(new_psi[i][j]-psi[i][j]);
    }
}
pomp_barrier_enter(&omp_rd_2);
#pragma omp barrier
pomp_barrier_exit(&omp_rd_2);
pomp_for_exit(&omp_rd_2);
#line 261 "stommel.c"
```
Using Opari with TAU

Step I: Configure KOJAK/opari [Download from  http://www.fz-juelich.de/zam/kojak/]

```bash
% cd kojak-2.1; cp mf/Makefile.defs.ibm Makefile.defs;
   edit Makefile
% make
```

Builds opari

Step II: Configure TAU with Opari (used here with MPI and PDT)

```bash
% configure --opari=/usr/contrib/TAU/kojak-2.1/opari
   --mpiinc=/usr/lpp/ppe.poe/include
   --mpilib=/usr/lpp/ppe.poe/lib
   --pdt=/usr/contrib/TAU/pdtoolkit-3.4
% make clean; make install
% setenv TAU_MAKEFILE /tau/<arch>/lib/Makefile.tau-...opari-...
% tau_cxx.sh -c foo.cpp
% tau_cxx.sh -c bar.f90
% tau_cxx.sh *.o -o app
```
Advances in TAU Measurement

- Profiling (four types)
  - Memory profiling
    - global heap memory tracking (several options)
  - Callpath profiling and calldepth profiling
    - user-controllable callpath length and calling depth
  - Phase-based profiling

- Tracing
  - Generation of VTF3 / SLOG2 traces files (fully portable)
  - Inclusion of hardware performance counts in trace files
  - Hierarchical trace merging

- Online performance overhead compensation

- Component software proxy generation and monitoring
Building Bridges to Other Tools: TAU
TAU Tracing Enhancements

- Configure TAU with `-TRACE -vtf=<dir> -slog2` options
  - `% configure -TRACE -vtf=<dir> ...`
  - Generates tau_merge, tau2vtf tools in `<tau>/<arch>/bin` directory
  - `% configure -TRACE -slog2`
  - Generates tau2slog2 and jumpshot v4 tools bundled with TAU in `<tau>/<arch>/bin` directory
    - Need working javac [v1.4] in your path

- Execute application
  - `% mpirun -np 4 app`

- Merge and convert trace files to VTF3/SLOG2 format
  - `% tau_treemerge.pl`
  - `% tau2vtf tau.trc tau.edf app.vpt.gz`
  - `% traceanalyzer foo.vpt.gz`

  - `% tau2slog2 tau.trc tau.edf app.slog2`
  - `% jumpshot app.slog2`
Intel ® Traceanalyzer (Vampir) Global Timeline
Visualizing TAU Traces with Counters/Samples
Visualizing TAU Traces with Counters/Samples
Memory Profiling in TAU

- Configuration option –PROFILEMEMORY
  - Records global heap memory utilization for each function
  - Takes one sample at beginning of each function and associates the sample with function name

- Configuration option –PROFILEHEADROOM
  - Records headroom (amount of free memory to grow) for each function
  - Takes one sample at beginning of each function and associates it with the callstack [TAU_CALLPATH_DEPTH env variable]

- Independent of instrumentation/measurement options selected
- No need to insert macros/calls in the source code
- User defined atomic events appear in profiles/traces
### Memory Profiling in TAU

Flash2 code profile (-PROFILEMEMORY) on IBM BlueGene/L [MPI rank 0]

<table>
<thead>
<tr>
<th>NumSamples</th>
<th>Max</th>
<th>Min</th>
<th>Mean</th>
<th>Std. Dev</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>252032</td>
<td>2022.7</td>
<td>1181.2</td>
<td>1534.3</td>
<td>410.04</td>
<td>NGDULEHYDRO_ID::HYDRO_ID - Heap Memory (KB)</td>
</tr>
<tr>
<td>252032</td>
<td>2022.8</td>
<td>1181.7</td>
<td>1534.3</td>
<td>410.04</td>
<td>NGDULEINRKC::INRKC - Heap Memory (KB)</td>
</tr>
<tr>
<td>104559</td>
<td>2023.2</td>
<td>331.13</td>
<td>1526.6</td>
<td>409.54</td>
<td>NGDULEEOS3D::EOS3D - Heap Memory (KB)</td>
</tr>
<tr>
<td>63008</td>
<td>2022.7</td>
<td>1182</td>
<td>1534.3</td>
<td>410.01</td>
<td>NGDULEUPDATE_SOLN::UPDATE_SOLN - Heap Memory (KB)</td>
</tr>
<tr>
<td>55545</td>
<td>2023.3</td>
<td>333.07</td>
<td>1514.2</td>
<td>408.31</td>
<td>DEASSTREE::DEASSTREENEIGHBORBLOCKLIST - Heap Memory (KB)</td>
</tr>
<tr>
<td>51374</td>
<td>2023</td>
<td>1179.4</td>
<td>1497.7</td>
<td>402.53</td>
<td>AMR_PROLONG_GEN_UNK_FUN - Heap Memory (KB)</td>
</tr>
<tr>
<td>42120</td>
<td>2022.7</td>
<td>1187.5</td>
<td>1533.5</td>
<td>409.83</td>
<td>ABUNDANCE_RESTRICT - Heap Memory (KB)</td>
</tr>
<tr>
<td>41958</td>
<td>2023</td>
<td>346.12</td>
<td>1514.9</td>
<td>408.39</td>
<td>AMR_RESTRICT_UNK_FUN - Heap Memory (KB)</td>
</tr>
<tr>
<td>31832</td>
<td>2022.8</td>
<td>1187.4</td>
<td>1534.1</td>
<td>409.91</td>
<td>AMR_RESTRICT_RED - Heap Memory (KB)</td>
</tr>
<tr>
<td>31504</td>
<td>2022.7</td>
<td>1181.8</td>
<td>1534.3</td>
<td>410.04</td>
<td>AMR_RESTRICT_REFRESH - Heap Memory (KB)</td>
</tr>
<tr>
<td>26042</td>
<td>2023</td>
<td>1179.2</td>
<td>1501.9</td>
<td>403.51</td>
<td>AMR_PROLONG_UNK_FUN - Heap Memory (KB)</td>
</tr>
</tbody>
</table>
Memory Profiling in TAU

- Instrumentation based observation of global heap memory (not per function)
  - call TAU.Track_MEMORY()
  - call TAU.Track_MEMORY_HEADROOM()
    - Triggers one sample every 10 secs
  - call TAU.Track_MEMORY_HERE()
  - call TAU.Track_MEMORY_HEADROOM_HERE()
    - Triggers sample at a specific location in source code
  - call TAU_SET_INTERRUPT_INTERVAL(seconds)
    - To set inter-interrupt interval for sampling
  - call TAU.DISABLE_TRACKING_MEMORY()
  - call TAU.DISABLE_TRACKING_MEMORY_HEADROOM()
    - To turn off recording memory utilization
  - call TAU.ENABLE_TRACKING_MEMORY()
  - call TAU.ENABLE_TRACKING_MEMORY_HEADROOM()
    - To re-enable tracking memory utilization
Profile Measurement – Three Flavors

- **Flat profiles**
  - Time (or counts) spent in each routine (nodes in callgraph).
  - Exclusive/inclusive time, no. of calls, child calls
  - E.g.: MPI_Send, foo, ...

- **Callpath Profiles**
  - Flat profiles, **plus**
  - Sequence of actions that led to poor performance
  - Time spent along a calling path (edges in callgraph)
  - E.g., “main=> f1 => f2 => MPI_Send” shows the time spent in MPI_Send when called by f2, when f2 is called by f1, when it is called by main. Depth of this callpath = 4 (TAU_CALLPATH_DEPTH environment variable)

- **Phase based profiles**
  - Flat profiles, **plus**
  - Flat profiles under a phase (nested phases are allowed)
  - Default “main” phase has all phases and routines invoked outside phases
  - Supports static or dynamic (per-iteration) phases
  - E.g., “IO => MPI_Send” is time spent in MPI_Send in IO phase
TAU Timers and Phases

- **Static timer**
  - Shows time spent in all invocations of a routine (foo)
  - E.g., “foo()” 100 secs, 100 calls

- **Dynamic timer**
  - Shows time spent in each invocation of a routine
  - E.g., “foo() 3” 4.5 secs, “foo 10” 2 secs (invocations 3 and 10 respectively)

- **Static phase**
  - Shows time spent in all routines called (directly/indirectly) by a given routine (foo)
  - E.g., “foo() => MPI_Send()” 100 secs, 10 calls shows that a total of 100 secs were spent in MPI_Send() when it was called by foo.

- **Dynamic phase**
  - Shows time spent in all routines called by a given invocation of a routine.
  - E.g., “foo() 4 => MPI_Send()” 12 secs, shows that 12 secs were spent in MPI_Send when it was called by the 4th invocation of foo.
Advances in TAU Performance Analysis

- Enhanced parallel profile analysis (*ParaProf*)
  - Callpath analysis integration in ParaProf
  - Event callgraph view
- *Performance Data Management Framework* (*PerfDMF*)
  - First release of prototype
- Integration with *Vampir Next Generation* (*VNG*)
  - Online trace analysis
- 3D Performance visualization prototype
- Component performance modeling and QoS
Pprof – Flat Profile (NAS PB LU)

- Intel Linux cluster
- F90 + MPICH
- Profile
  - Events
    - code
    - MPI
- Metric
  - time
- Text display

Buffers Files Tools Edit Search Mute Help

Reading Profile files in profile.*

%Time  Exclusive msec  Inclusive total msec  #Call  #Subrs  Inclusive Name usec/call

100.0%  11 3:11.293 15 191293269 apply
99.6%  3,667 3:10.463 3 37157 63462925 bcast inputs
67.1%  491 2:08.323 37200 37200 3350 exchange_1
44.5%  5,461 1:26.159 9300 9300 9957 bputc
41.0%  1,128 1:18.436 18600 18600 4227 MPI_RECV()
29.5%  6,770 56.407 9300 9300 5065 bsts
26.2%  50,142 50,142 19204 19204 2611 MPI_Send()
16.2%  24,451 31,031 301 602 103096 rsh
3.9%  7,501 7,501 9300 9300 807 jacld
3.4%  830 6,594 604 1812 10818 exchange_3
3.4%  6,590 6,590 9300 9300 709 jacu
2.6%  7,999 7,999 608 608 8206 MPI_Wait()
0.2%  0.44 400 1 4 400081 init_comm
0.2%  399 399 1 39 399634 MPI_InitT()
0.1%  140 247 1 47616 247085 setiv
0.1%  131 131 57252 0 2 exact
0.1%  99 103 1 2 103168 ithc
0.1%  966 966 1 2 96458 read_input
0.0%  95 95 9 0 10603 MPI_Bcast()
0.0%  99 99 1 7937 44878 error
0.0%  26 24 24 603 0 40 MPI_Irecv()
0.0%  24 15 15 1 5 15630 MPI_Finalize()
0.0%  3 3 3 3 12335 setiv
0.0%  4 12 12 1700 12353 setiv
0.0%  3 3 3 0 2803 lenorm
0.0%  3 3 3 0 491 MPI_Allreduce()
0.0%  3 3 3 0 3874 pintgr
0.0%  1 1 1 0 1007 MPI_BARRIER()
0.0%  0.116 0.837 0.837 0 2 837 exchange_4
0.0%  0.512 0.512 0.512 1 2 512 MPI_Keyval_create()
0.0%  0.121 0.353 0.353 1 0 353 exchange_5
0.0%  0.24 0.191 0.191 1 2 191 MPI_Send()
0.0%  0.103 0.103 0.103 0 1 17 MPI_Type_contiguous()

--- NPB ему.out (Fundamental)--L8--Top---
ParaProf – Manager Window

- Performance database

- Derived performance metrics

ACTS Workshop 2005
ParaProf – Full Profile (Miranda)

8K processors!
ParaProf– Flat Profile (Miranda)
ParaProf– Callpath Profile (Flash)
ParaProf– Callpath Profile (ESMF)

21-level callpath

Value Type: exclusive

<table>
<thead>
<tr>
<th>Percentage</th>
<th>Description</th>
<th>n, c, t</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.7487%</td>
<td>mean</td>
<td>1.00</td>
</tr>
<tr>
<td>11.2785%</td>
<td>n, c, t</td>
<td>1.00</td>
</tr>
<tr>
<td>10.9582%</td>
<td>n, c, t</td>
<td>3.00</td>
</tr>
<tr>
<td>10.4453%</td>
<td>n, c, t</td>
<td>2.00</td>
</tr>
<tr>
<td>10.3146%</td>
<td>n, c, t</td>
<td>0.00</td>
</tr>
</tbody>
</table>
## Gprof Style Callpath View in Paraprof (SAGE)

### Metric Name: Time
Sorted By: exclusive
Units: seconds

<table>
<thead>
<tr>
<th>Exclusive</th>
<th>Inclusive</th>
<th>Calls/Tot.Calls</th>
<th>Name[id]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.8584</td>
<td>1.8584</td>
<td>1196/13188</td>
<td>TOKEN_MODULE::TOKEN_GS_I [521]</td>
</tr>
<tr>
<td>0.584</td>
<td>0.584</td>
<td>234/13188</td>
<td>TOKEN_MODULE::TOKEN_GS_L [544]</td>
</tr>
<tr>
<td>25.0819</td>
<td>25.0819</td>
<td>11758/13188</td>
<td>TOKEN_MODULE::TOKEN_GS_R8 [734]</td>
</tr>
<tr>
<td>---</td>
<td>27.5242</td>
<td>13188</td>
<td>MPI_Waitall() [525]</td>
</tr>
<tr>
<td>17.9579</td>
<td>39.1657</td>
<td>156/156</td>
<td>DERIVATIVE_MODULE::DERIVATIVES_NOFACE [841]</td>
</tr>
<tr>
<td>---</td>
<td>17.9579</td>
<td>156</td>
<td>DERIVATIVE_MODULE::DERIVATIVES_FACE [843]</td>
</tr>
<tr>
<td>0.0156</td>
<td>0.0195</td>
<td>312/312</td>
<td>TIMER_MODULE::TIMERSET [77]</td>
</tr>
<tr>
<td>0.1133</td>
<td>9.1269</td>
<td>2340/2340</td>
<td>MESSAGE_MODULE::CLONE_GET_R8 [808]</td>
</tr>
<tr>
<td>0.1602</td>
<td>11.4608</td>
<td>4056/4056</td>
<td>MESSAGE_MODULE::CLONE_PUT_R8 [850]</td>
</tr>
<tr>
<td>0.0059</td>
<td>0.6006</td>
<td>117/117</td>
<td>MESSAGE_MODULE::CLONE_PUT_I [856]</td>
</tr>
<tr>
<td>14.1151</td>
<td>21.6209</td>
<td>5/5</td>
<td>MATRIX_MODULE::MCGDS [1443]</td>
</tr>
<tr>
<td>---</td>
<td>14.1151</td>
<td>5</td>
<td>MATRIX_MODULE::CSR_CG_SOLVER [1470]</td>
</tr>
<tr>
<td>0.0654</td>
<td>1.2617</td>
<td>1005/1005</td>
<td>TOKEN_MODULE::TOKEN_GET_R8 [769]</td>
</tr>
<tr>
<td>0.0557</td>
<td>5.2714</td>
<td>1005/1005</td>
<td>TOKEN_MODULE::TOKEN_REDUCTION_R8_S [1475]</td>
</tr>
<tr>
<td>0.0703</td>
<td>0.9726</td>
<td>1000/1000</td>
<td>TOKEN_MODULE::TOKEN_REDUCTION_R8_V [208]</td>
</tr>
</tbody>
</table>
ParaProf – Phase Profile (MFIX)

In 51st iteration, time spent in MPI_Waitall was 85.81 secs

Total time spent in MPI_Waitall was 4137.9 secs across all 92 iterations.
ParaProf - Statistics Table (Uintah)

<table>
<thead>
<tr>
<th>Name</th>
<th>P_WALL_CLOCK_TIME</th>
<th>Calls</th>
<th>Child Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>main() void (int, char **)</td>
<td>0.015</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>Uintah::ProcessorGroup *Uintah::Parallel::getRootProcessor()</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Uintah::SimpleSimulationController &amp;Uintah::SimpleSimulation</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Uintah::SimulationController &amp;Uintah::SimulationController::Si</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>bool Uintah::Parallel::usingMPI()</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>int Uintah::Parallel::getMPIRank()</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>void Uintah::OnDemandDataWarehouse::~OnDemandDataW</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>void Uintah::Parallel::determinIfRunningUnderMPI(int, char</td>
<td>0.002</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>void Uintah::Parallel::finalizeManager(Uintah::Parallel::Circur</td>
<td>0.011</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>void Uintah::Parallel::initializeManager(int &amp; char **&amp; const</td>
<td>0.001</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Comm_rank()</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Comm_size()</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MPI_Init_thread()</td>
<td>6.327</td>
<td>1</td>
<td>39</td>
</tr>
<tr>
<td>void Uintah::Parallel::noThreading()</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>void Uintah::SimpleSimulationController::run() Uintah::Simple</td>
<td>0.074</td>
<td>1</td>
<td>154</td>
</tr>
<tr>
<td>MPIScheduler::actuallyCompile()</td>
<td>0.109</td>
<td>2</td>
<td>44</td>
</tr>
<tr>
<td>MPIScheduler::execute()</td>
<td>27.68</td>
<td>11</td>
<td>3,460</td>
</tr>
<tr>
<td>MPI_Reduce()</td>
<td>0.001</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>Uintah::DataWarehouse::ScrubMode Uintah::OnDemandC</td>
<td>0</td>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>Uintah::OnDemandDataWarehouse &amp;Uintah::OnDemandC</td>
<td>0</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>bool Uintah::OnDemandDataWarehouse::timestepAborde</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>bool Uintah::OnDemandDataWarehouse::timestepRestar</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>bool Uintah::SimpleSimulationController::needRecompiler</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>void Uintah::OnDemandDataWarehouse::get(Uintah::Red</td>
<td>0.001</td>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td>void Uintah::OnDemandDataWarehouse::override(const l</td>
<td>0.001</td>
<td>20</td>
<td>40</td>
</tr>
</tbody>
</table>
ParaProf – Histogram View (Miranda)

- Scalable 2D displays

8k processors

16k processors
ParaProf – Callgraph View (MFIX)
Profiling of Miranda on BG/L

- Profile code performance (automatic instrumentation)
- Scaling studies (problem size, number of processors)
- Run on 8K, 16K and 32K processors!
ParaProf – 3D Full Profile (Miranda)

16k processors
ParaProf Bar Plot (Zoom in/out +/-)
ParaProf – 3D Scatterplot (Miranda)

- Each point is a “thread” of execution
- A total of four metrics shown in relation
- ParaVis 3D profile visualization library
  - JOGL
- Visualization and Analysis of MPI Programs
- Originally developed by Forschungszentrum Jülich
- Current development by Technical University Dresden, Germany
- Distributed by Intel
- [http://www.vampir-ng.de](http://www.vampir-ng.de)
Performance Tracing on Miranda

- Use TAU to generate VTF3 traces for Vampir analysis
  - MPI calls with HW counter information (not shown)
  - Detailed code behavior to focus optimization efforts
S3D on Lemieux (tau2vtf, Vampir)
S3D on Lemieux (Zoomed)
Jumpshot Trace Visualizer [ANL] (S3D)
Jumpshot Trace Visualizer (S3D on Tru64)
Computing platforms (selected)
- IBM SP/pSeries/BGL, SGI Altix/Origin, Cray T3E/SV-1/X1/XT3, HP (Compaq) SC (Tru64), Sun, Hitachi SR8000, NEC SX-5/6, Linux clusters (IA-32/64, Alpha, PPC, PA-RISC, Power, Opteron), Apple (G4/5, OS X), Windows

Programming languages
- C, C++, Fortran 77/90/95, HPF, Java, Python

Thread libraries (selected)
- pthrethreads, SGI sproc, Java, Windows, OpenMP

Compilers (selected)
- Intel, PGI, GNU, Fujitsu, Sun, PathScale, SGI, Cray, IBM, HP, NEC, Absoft, Lahey, Nagware
Project Affiliations (selected)

- Center for Simulation of Accidental Fires and Explosion
  - University of Utah, ASCI ASAP Center, C-SAFE
  - Uintah Computational Framework (UCF) (C++)

- Center for Simulation of Dynamic Response of Materials
  - California Institute of Technology, ASCI ASAP Center
  - Virtual Testshock Facility (VTF) (Python, Fortran 90)

- Earth Systems Modeling Framework (ESMF)
  - NSF, NOAA, DOE, NASA, …
  - Instrumentation for ESMF framework and applications
  - C, C++, and Fortran 95 code modules
  - MPI wrapper library for MPI calls
Project Affiliations (selected) (continued)

- Lawrence Livermore National Lab
  - Hydrodynamics (Miranda)
- Sandia National Lab and Los Alamos National Lab
  - DOE CCTTSS SciDAC project
  - Common component architecture (CCA) integration
- Argonne National Lab
  - Jumpshot SLOG2 SDK project
  - ZeptoOS - scalable components for petascale architectures
  - KTAU - integration of TAU infrastructure in Linux kernel
- Oak Ridge National Lab
  - Contribution to the Joule Report: S3D, AORSA3D
Important Questions for Application Developers

- How does performance vary with different compilers?
- Is poor performance correlated with certain OS features?
- Has a recent change caused unanticipated performance?
- How does performance vary with MPI variants?
- Why is one application version faster than another?
- What is the reason for the observed scaling behavior?
- Did two runs exhibit similar performance?
- How are performance data related to application events?
- Which machines will run my code the fastest and why?
- Which benchmarks predict my code performance best?
Performance Problem Solving Goals

- Answer questions at multiple levels of interest
  - Data from low-level measurements and simulations
    - use to predict application performance
  - High-level performance data spanning dimensions
    - machine, applications, code revisions, data sets
    - examine broad performance trends
- Discover general correlations application performance and features of their external environment
- Develop methods to predict application performance on lower-level metrics
- Discover performance correlations between a small set of benchmarks and a collection of applications that represent a typical workload for a given system
Performance Data Management Framework

TAU Performance System

Performance Analysis Programs

scalability analysis
ParaProf
cluster analysis

Query and Analysis Toolkit

Data Mining
(Weka)

Statistics
(R / Omega)

Java PerfDMF API

SQL (PostgreSQL, MySQL, DB2, Oracle)

XML document
formatted profile data

* gprof
* mpiP
* psrun
* HPMtoolkit
* ...

ACTS Workshop 2005
ParaProf Performance Profile Analysis

Raw files
PerfDMF managed (database)
Application
Experiment
Trial
HPMToolkit
Metadata
MpiP
TAU

ACTS Workshop 2005  TAU Performance System  75
PerfExplorer

- Performance knowledge discovery framework
  - Use the existing TAU infrastructure
    - TAU instrumentation data, PerfDMF
  - Client-server based system architecture
  - Data mining analysis applied to parallel performance data
    - comparative, clustering, correlation, dimension reduction, ...

- Technology integration
  - Relational DatabaseManagement Systems (RDBMS)
  - Java API and toolkit
  - R-project / Omegahat statistical analysis
  - WEKA data mining package
  - Web-based client
PerfExplorer Architecture

PerfExplorer Client
- Analysis Requestor
- Analysis Monitor
- Analysis Graphs
- Analysis Data
- Raw Performance Data
- Scalability Charts

PerfExplorer Server
- RMI Client
- RMI Server
- Raw Performance Data
- Analysis Data
- Analysis Graphs

Analysis Engine
- Weka
- R
- OCTAVE

DBMS
- JFreeChart

PerfDMF
- JFreeChart
PerfExplorer Client GUI
Hierarchical and K-means Clustering (sPPM)
Miranda Clustering on 16K Processors

(a) Processes Per Cluster

(b) Virtual Topology

(c) Wall Clock Time Per Event
PERC Tool Requirements and Evaluation

- Performance Evaluation Research Center (PERC)
  - DOE SciDAC
  - Evaluation methods/tools for high-end parallel systems

- PERC tools study (led by ORNL, Pat Worley)
  - In-depth performance analysis of select applications
  - Evaluation performance analysis requirements
  - Test tool functionality and ease of use

- Applications
  - Start with fusion code – GYRO
  - Repeat with other PERC benchmarks
  - Continue with SciDAC codes
Primary Evaluation Machines

- *Phoenix* (ORNL – Cray X1)
  - 512 multi-streaming vector processors
- *Ram* (ORNL – SGI Altix (1.5 GHz Itanium2))
  - 256 total processors
- *TeraGrid*
  - ~7,738 total processors on 15 machines at 9 sites
- *Cheetah* (ORNL – p690 cluster (1.3 GHz, HPS))
  - 864 total processors on 27 compute nodes
- *Seaborg* (NERSC – IBM SP3)
  - 6080 total processors on 380 compute nodes
GYRO Execution Parameters

- Three benchmark problems
  - \textit{B1-std} : 16n processors, 500 timesteps
  - \textit{B2-cy} : 16n processors, 1000 timesteps
  - \textit{B3-gtc} : 64n processors, 100 timesteps (very large)
- Test different methods to evaluate nonlinear terms:
  - Direct method
  - FFT (“nl2” for B1 and B2, “nl1” for B3)
- Task affinity enabled/disabled (p690 only)
- Memory affinity enabled/disabled (p690 only)
- Filesystem location (Cray X1 only)
PerfExplorer Analysis of Self-Instrumented Data

- PerfExplorer
  - Focus on comparative analysis
  - Apply to PERC tool evaluation study
- Look at user timer data
  - Aggregate data
    - no per process data
    - process clustering analysis is not applicable
  - Timings output every $N$ timesteps
    - some phase analysis possible
- Goal
  - Recreate manually generated performance reports
**PerfExplorer Interface**

Select experiments and trials of interest

Data organized in application, experiment, trial structure (will allow arbitrary in future)

Experiment metadata
PerfExplorer Interface

Select analysis
Timesteps per Second

- Cray X1 is the fastest to solution in all 3 tests
- FFT (nl2) improves time for B3-gtc only
- TeraGrid faster than p690 for B1-std?
- Plots generated automatically
Relative Efficiency (B1-std)

- By experiment (B1-std)
  - Total runtime (Cheetah (red))
- By event for one experiment
  - Coll_tr (blue) is significant
- By experiment for one event
  - Shows how Coll_tr behaves for all experiments
Current and Future Work

- ParaProf
  - Developing phase-based performance displays

- PerfDMF
  - Adding new database backends and distributed support
  - Building support for user-created tables

- PerfExplorer
  - Extending comparative and clustering analysis
  - Adding new data mining capabilities
  - Building in scripting support

- Performance regression testing tool (PerfRegress)

- Integrate in Eclipse Parallel Tool Project (PTP)
Concluding Discussion

- Performance tools must be used effectively
- More intelligent performance systems for productive use
  - Evolve to application-specific performance technology
  - Deal with scale by “full range” performance exploration
  - Autonomic and integrated tools
  - Knowledge-based and knowledge-driven process
- Performance observation methods do not necessarily need to change in a fundamental sense
  - More automatically controlled and efficiently use
- Develop next-generation tools and deliver to community
- Open source with support by ParaTools, Inc.
- http://www.cs.uoregon.edu/research/tau
Hands-On Session

- Login to odin.cs.indiana.edu and get software
  
  ```
  % cp /san/cca/tau/tautraining.tar.gz .
  % tar zxf tautraining.tar.gz
  ```

- Follow instructions in the README file
Support Acknowledgements

- Department of Energy (DOE)
  - Office of Science contracts
  - University of Utah ASCI Level 1 sub-contract
  - ASC/NNSA Level 3 contract
- NSF
  - High-End Computing Grant
- Research Centre Juelich
  - John von Neumann Institute
  - Dr. Bernd Mohr
- Los Alamos National Laboratory