Advances in the TAU Performance System for Extreme Performance Engineering

Allen D. Malony
Department of Computer and Information Science
University of Oregon
Outline

Part 1
- Performance engineering and productivity
- Role of performance knowledge
- Model-based performance engineering and expectations

Part 2
- Introduction to the TAU performance system

Part 3
- Performance data mining (S3D, NWChem)
- Hybrid performance measurement (TAUebs)
- Heterogeneous performance measurement (GPU)
- Parallel performance visualization
Parallel Performance Engineering and Productivity

- Scalable, optimized applications deliver HPC promise
- Optimization through performance engineering process
  - Understand performance complexity and inefficiencies
  - Tune application to run optimally on high-end machines
- How to make the process more effective and productive?
- What performance technology should be used?
  - Performance technology part of larger environment
  - Programmability, reusability, portability, robustness
  - Application development and optimization productivity
- Process, performance technology, and its use will change as parallel systems evolve
- Goal is to deliver effective performance with high productivity value now and in the future
Parallel Performance Engineering Process

- Traditionally an empirically-based approach
  observation ⇔ experimentation ⇔ diagnosis ⇔ tuning
- Performance technology developed for each level

```
Performance Tuning
hypotheses

Performance Diagnosis
properties

Performance Experimentation
characterization

Performance Observation
```

Performance Technology
- Data mining
- Models
- Expert systems

Performance Technology
- Instrumentation
- Measurement
- Analysis
- Visualization

Performance Technology
- Experiment management
- Performance storage
Performance Technology and Scale

- What does it mean for performance technology to scale?
  - Instrumentation, measurement, analysis, tuning
- Scaling complicates observation and analysis
  - Performance data size and value
    - standard approaches deliver a lot of data with little value
  - Measurement overhead, intrusion, perturbation, noise
    - measurement overhead $\rightarrow$ intrusion $\rightarrow$ perturbation
    - tradeoff with analysis accuracy
    - “noise” in the system distorts performance
- Analysis complexity increases
  - More events, larger data, more parallelism
- Traditional empirical process breaks down at extremes
Performance Engineering Process

- What will enhance productive application development with a goal to improve performance optimization?
- Current performance engineering process decouples the application from the performance analysis
  - Little sharing of application knowledge with the tools
- Performance engineering process and tools must be more application-aware
- Support application-specific performance views
  - What are the important events and performance metrics?
  - How are these tied to the application structure and computational model?
  - How can knowledge about the application domain and algorithms be used to improve performance understanding?
Role of Intelligence, Automation, and Knowledge

- Evolution of high-end computing towards greater scale and heterogeneity forces the process to be more intelligent
- Even with intelligent and application-specific tools, the decisions of what to analyze is difficult
- More automation and knowledge-based decision making
- Build these capabilities into performance tools
  - Support broader experimentation methods and refinement
  - Access and correlate data from several sources
  - Automate performance data analysis / mining / learning
  - Include predictive features and experiment refinement
- Knowledge-driven adaptation and optimization guidance
- Address scale issues through increased expertise
Whole Performance Evaluation

- Extreme scale performance is an optimized orchestration
  - Application, processor, memory, network, I/O
- Reductionist approaches to performance will be unable to support optimization and productivity objectives
- Application-level only performance view is myopic
  - Interplay of hardware, software, and system components
  - Ultimately determines how performance is delivered
- Performance should be evaluated *in toto*
  - Application and system components
  - Understand effects of performance interactions
  - Identify opportunities for optimization across levels
- Need *whole performance evaluation practice*
Role of Structure in Performance Understanding

- Dealing with increased complexity of the problem (measurement and analysis) should not be the only focus
- There is fundamental structure in the computational model and an application's behavior

- Performance is a consequence of structural forms and behaviors
- It is the relationship between performance data and application semantics that is important to understand
  - This need not be complex (not as complex as it seems)

Shigeo Fukuda, 1987

Lunch with a Helmut On
Parallel Performance Diagnosis

- Parallel System
- Parallel Program

- Constraints on observational capabilities, invocation of measurement tools
- Analysis, modeling, and presentation of empirical data

Performance Hypothesis

- Hypothesis refinement from empirical results
- General performance results add to performance knowledge base

Stored Performance Knowledge

- System/program characteristics plus performance knowledge used for initial hypotheses
“Extreme” Performance Engineering

- Improve performance problem solving process
- Strategy to respond to challenges of technology change and disruptions
  - Evolve performance engineering process and technology
  - Build on robust, integrated performance infrastructure
  - Carry forward performance expertise and knowledge
- Model-oriented with knowledge-based reasoning
  - Community-driven knowledge engineering
  - Automated data / decision analytics
- Closer integration with application development and execution environment
Empirical performance data evaluated with respect to performance expectations at various levels of abstraction.
Models-based Performance Engineering

- Performance engineering tools and practice must incorporate a performance knowledge discovery process
- Focus on and automate performance problem identification and use to guide tuning decisions
- Model-oriented knowledge
  - Computational semantics of the application
  - Symbolic models for algorithms
  - Performance models for system architectures / components
  - Use to define performance expectations
- Higher-level abstraction for performance investigation
- Application developers can be more directly involved in the performance engineering process
Extreme Performance Engineering Claims

Claim 1: “Scaling up” current performance measurement and analysis techniques are insufficient for extreme scale (ES) performance diagnosis and tuning.

Claim 2: Performance of ES applications and systems should be evaluated in toto, to understand the effects of performance interactions and identify opportunities for optimization based on whole-performance engineering.

Claim 3: Adaptive ES applications might require performance monitoring, dynamic control, and tight coupling between the application and system at run time.

Claim 4: Optimization of ES applications requires model-level knowledge of parallel algorithms, parallel computation, system, and performance.
Performance Expectations

- Context for understanding the performance behavior of the application and system

- Traditional performance implicitly requires an expectation
  - Users are forced to reason from the perspective of absolute performance for every performance experiment and every application operation
  - Peak measures provide an absolute upper bound
  - Empirical data alone does not lead to performance understanding and is insufficient for optimization

- Empirical measurements lack a context for determining whether the operations under consideration are performing well or performing poorly
Sources of Performance Expectations

*Computation Model* – Operational semantics of a parallel application specified by its algorithms and structure define a space of relevant performance behavior.

*Symbolic Performance Model* – Symbolic or analytical model provides the relevant parameters for the performance model, which generate expectations.

*Historical Performance Data* – Provides real execution history and empirical data on similar platforms.

*Relative Performance Data* – Used to compare similar application operations across architectural components.

*Architectural parameters* – Based on what is known of processor, memory, and communications performance.
Model Use

- Performance models derived from application knowledge, performance experiments, and symbolic analysis
- They will be used to predict the performance of individual system components, such as communication and computation, as well as the application as a whole.
- The models can then be compared to empirical measurements—manually or automatically—to pinpoint or dynamically rectify performance problems.
- Further, these models can be evaluated at runtime in order to reduce perturbation and data management burdens, and to dynamically reconfigure system software.
Where are we now?

- Large-scale performance tools infrastructure
  - Instrumentation (source, binary, interposition, compiler)
  - Measurement and analysis
    - TAU performance system, Scalasca/Score-P, HPCToolkit
  - Performance database and data mining
  - Performance visualization
  - Monitoring middleware
- Automatic performance tuning (auto-tuning)
  - Block-level and application-level
- Some model/language integration (MPI, OpenMP, hybrid)
- Scalable parallel libraries and algorithms
What do we need?

- Performance knowledge engineering
  - At all levels
  - Support the building of models and expectations
  - Represented in form the tools can reason about

- Understanding of “performance” interactions
  - Between integrated components
  - Control and data interactions

- Properties of concurrency and dependencies
  - With respect to scientific problem formulation
  - Translate to performance requirements
  - Understand in presence of high concurrency and # cores

- More robust tools that are being used more broadly
TAU Performance System® (http://tau.uoregon.edu)

- Tuning and Analysis Utilities (20+ year project)
- Performance problem solving framework for HPC
  - Integrated, scalable, flexible, portable
  - Target all parallel programming / execution paradigms
- Integrated performance toolkit
  - Multi-level performance instrumentation
  - Flexible and configurable performance measurement
  - Widely-ported performance profiling / tracing system
  - Performance data management and data mining
  - Open source (BSD-style license)
- Broad use in complex software, systems, applications
General Target Computation Model in TAU

- **Node**: physically distinct shared memory machine
  - Message passing node interconnection network
- **Context**: distinct virtual memory space within node
- **Thread**: execution threads (user/system) in context
TAU Performance System Components

TAU Architecture
Instrumentation
- source code
- object code
- library wrapper
- binary code
- virtual machine

Measurement
- event identifier
- event source
- atomic events
- event mapping
- event context

Profiling
- statistics
- phase profiles
- I/O profiles
- profile sampling
- trace buffering
- record creation
- trace I/O

PerfDMF
- performance data sources
- OS and runtime system modules
- timing
- hardware counters
- system counters
- kernel
- threading
- interrupts
- runtime system
- IO

Tracing
- traces
- symbol table

Analysis
- measurement
- instrument selection

Performance Data Mining
- GUI
- Scripting interface
- PerfExplorer Component Interfaces
- Analysis Components
- Data Components
- PerfDMF
- Expert Notes
- Analyze Results
- Inference Engine
- Prescriptions

Parallel Profile Analysis
- PDT
- C/C++/Fortran parser
- F77/90/95
- SLOOON
- CHASM
- C/C++/Fortran90/95
- Interoperability
- Automatic source instrumentation

ParaProf
- PARANALYSIS
- Call graphs
- Call trees
- Comparative Displays
- Test Displays
- 3D Displays
- Domain-specific Languages
- Scripting Interface
- Jython

TAUoverSupermon
- Performance Monitoring
- Monitoring Components
- Application Enablers
- Application Libraries
- Application Components
- Application Interface
- Application Data
- Application Presentation

CEA DAM
Advances in the TAU Performance System
August 25, 2011
TAU Instrumentation / Measurement

Instrumentation

*event selection*

- source code
- object code
- library wrapper
- binary code
- virtual machine

MEASUREMENT API

Measurement

**Event creation and management**

- event identifier
- entry/exit events
- atomic events
- event mapping
- event control

**Profiling**

- statistics
- atomic profiles
- entry/exit profiles
- phase profiles
- I/O profiles
- profile sampling

**Tracing**

- trace buffering
- record creation
- trace I/O
- timestamp generation
- trace filtering
- trace merging

**Performance data sources**

- timing
- hardware counters
- system counters
- kernel

**OS and runtime system modules**

- threading
- interrupts
- runtime system
- I/O
TAU Instrumentation Approach

- Direct and indirect performance instrumentation
  - Direct instrumentation of program (system) code (probes)
  - Indirect support via sampling
- Support for standard program events
  - Routines, classes and templates
  - Statement-level blocks
  - Begin/End events (Interval events)
- Support for user-defined events
  - Begin/End events specified by user
  - Atomic events (e.g., size of memory allocated/freed)
  - Flexible selection of event statistics
- Provides static events and dynamic events
- Instrumentation optimization
TAU Instrumentation Mechanisms

- **Source code**
  - Manual (TAU API, TAU component API)
  - Automatic (robust)
    - C, C++, F77/90/95 (Program Database Toolkit (PDT))
    - OpenMP (directive rewriting (Opari), POMP2 spec)
    - Powerful library header wrapping

- **Object code**
  - Pre-instrumented libraries (e.g., MPI using PMPI)
  - Statically- and dynamically-linked (with `LD_PRELOAD`)

- **Executable code / runtime**
  - Binary and dynamic instrumentation (Dyninst)
  - Virtual machine instrumentation (e.g., Java using JVMPI)
  - Python runtime

- **TAU_COMPILER** to automate instrumentation process
Automatic Source-level Instrumentation

PDT source analyzer

Parsed program

Instrumentation specification file

BEGIN_EXCLUDE_LIST
Foo
Bar
D#EMM
END_EXCLUDE_LIST

BEGIN_FILE_EXCLUDE_LIST
f*.f90
Foo?.cpp
END_FILE_EXCLUDE_LIST
BEGIN_FILE_INCLUDE_LIST
main.cpp
foo.f90
END_FILE_INCLUDE_LIST

Application source

Instrumented source

tau_instrumentor
MPI Wrapper Interposition Library

- Uses standard MPI Profiling Interface
  - Provides name shifted interface (weak bindings)
    - MPI_Send = PMPI_Send

- Create TAU instrumented MPI library
  - Interpose between MPI and TAU
    - `-lmpi` replaced by `-lTauMpi -lpmpi -lmpi`
  - No change to the source code, just re-link application!

- Can we interpose MPI for compiled applications?
  - Avoid re-compilation or re-linking
  - Requires shared library MPI
    - uses `LD_PRELOAD` for Linux
  - Approach will work with other shared libraries (see later slide)
  - Use TAU `tau_exec` (see later slide)
    - `% mpirun -np 4 tau_exec a.out`
Binary Instrumentation – DyninstAPI and tau_run

☐ TAU has been a long-time user of DyninstAPI

☐ Using DyninstAPI’s recent binary re-writing capabilities, created a binary re-writer tool for TAU (tau_run)
  ☐ Supports TAU's performance instrumentation
  ☐ Works with TAU instrumentation selection
    ➢ files and routines based on exclude/include lists
  ☐ TAU’s measurement library (DSO) is loaded by tau_run

☐ Runtime (pre-execution) and binary re-writing are supported

☐ Simplifies code instrumentation and tool usage greatly!

  % tau_run a.out -o a.inst
  % mpirun -np 4 ./a.inst
tau_run with NAS PBS

/home/livetau% cd ~/tutorial
/home/livetau/tutorial% # Build an uninstrumented bt NAS Parallel Benchmark
/home/livetau/tutorial% make bt CLASS=W NPROCS=4
/home/livetau/tutorial% cd bin
/home/livetau/tutorial/bin% # Run the instrumented code
/home/livetau/tutorial/bin% mpirun -np 4 ./bt_W.4
/home/livetau/tutorial/bin%
/home/livetau/tutorial/bin% # Instrument the executable using TAU with DyninstAPI
/home/livetau/tutorial/bin%
/home/livetau/tutorial/bin% tau_run ./bt_W.4 -o ./bt.i
/home/livetau/tutorial/bin% rm -rf profile.* MULT*
/home/livetau/tutorial/bin% mpirun -np 4 ./bt.i
/home/livetau/tutorial/bin%
/home/livetau/tutorial/bin% paraprof
/home/livetau/tutorial/bin%
/home/livetau/tutorial/bin% # Choose a different TAU configuration
/home/livetau/tutorial/bin% ls $TAU/libTAUsh
libTAUsh-depthlimit-mpi-pdt.so*
libTAUsh-mpi-pdt.so*
libTAUsh-mpi-pdt-upc.so*
libTAUsh-mpi-python-pdt.so*
libTAUsh-papi-mpi-pdt.so*
libTAUsh-papi-mpi-pdt-upc.so*
libTAUsh-papi-mpi-pdt-upc-udp.so*
libTAUsh-papi-mpi-pdt-vampirtrace-trace.so* libTAUsh-papi-python-pdt.so*
/home/livetau/tutorial/bin%
/home/livetau/tutorial/bin% tau_run -XrunTAUsh-papi-mpi-pdt-vampirtrace-trace bt_W.4 -o bt.vpt
/home/livetau/tutorial/bin% setenv VT_METRICS PAPI_FP_INS:PAPI_L1_DCM
/home/livetau/tutorial/bin% mpirun -np 4 ./bt.vpt
/home/livetau/tutorial/bin% vampir bt.vpt.otf &
Library Interposition

- Simplify TAU usage to assess performance properties
  - Application, I/O, memory, communication
- Designed a new tool that leverages runtime instrumentation by pre-loading measurement libraries
- Works on dynamic executables (default under Linux)
- Substitutes routines (e.g., I/O, MPI, memory allocation/deallocation) with instrumented calls
  - Interval events (e.g., time spent in write())
  - Atomic events (e.g., how much memory was allocated)
TAU Execution Command – tau_exec

- Uninstrumented execution
  \% mpirun -np 256 ./a.out

- Track MPI performance
  \% mpirun -np 256 tau_exec ./a.out

- Track I/O and MPI performance (MPI enabled by default)
  \% mpirun -np 256 tau_exec -io ./a.out

- Track memory operations
  \% setenv TAU_TRACK_MEMORY_LEAKS 1
  \% mpirun -np 256 tau_exec -memory ./a.out

- Track I/O performance and memory operations
  \% mpirun -np 256 tau_exec -io -memory ./a.out
POSIX I/O Calls Supported

- Unbuffered I/O
  - open, open64, close, read, write, readv, writev, creat, creat64

- Buffered I/O
  - fopen, fopen64, fdopen, freopen, fclose
  - fprintf, fscanf, fwrite, fread

- Communication
  - socket, pipe, socketpair, bind, accept, connect
  - recv, send, sendto, recvfrom, pclose

- Control
  - fcntl, rewind, lseek, lseek64, fseek, dup, dup2, mkstep, tmpfile

- Asynchronous I/O
  - aio_{read,write,suspend,cancel,return}, lio_listio
Library wrapping – tau_gen_wrapper

- How to instrument an external library without source?
  - Source may not be available
  - Library may be too cumbersome to build (with instrumentation)

- Build a library wrapper tools
  - Used PDT to parse header files
  - Generate new header files with instrumentation files
  - Three methods: runtime preloading, linking, redirecting headers

- Add to TAU_OPTIONS environment variable:
  - `--optTauWrapFile=<wrapperdir>/link_options.tau`

- Wrapped library
  - Redirects references at routine callsite to a wrapper call
  - Wrapper internally calls the original
  - Wrapper has TAU measurement code
HDF5 Library Wrapping

```
[sameer@zorak]$ tau_gen_wrapper hdf5.h /usr/lib/libhdf5.a -f select.tau
```

Usage: `tau_gen_wrapper <header> <library> [-r|-d|-w (default)] [-g groupname] [-i headerfile] [-c|-c++|-fortran] [-f <instr_req_file>]`

- instruments using runtime preloading (-r), or -Wl,-wrap linker (-w), redirection of header file to redefine the wrapped routine (-d)
- instrumentation specification file (select.tau)
- group (hdf5)
- `tau_exec` loads `libhdf5_wrap.so` shared library using `-loadlib=<libwrap_pkg.so>`
- creates the wrapper/ directory with `-opt`

NODE 0; CONTEXT 0; THREAD 0:

```
%Time    Exclusive    Inclusive       #Call      #Subrs Inclusive Name
msec    total    msec      usec/call
---------------------------------------------------------------------------------------
100.0    0.057    1          13       1236 .TAU Application
70.8     0.875    0.875     1          0        875 hid_t H5Fcreate()
9.7     0.12      0.12      1          0        120 herr_t H5Fclose()
6.0     0.074    0.074      1          0        74 hid_t H5Dcreate()
3.1     0.038    0.038      1          0        38 herr_t H5Dwrite()
2.6     0.032    0.032      1          0        32 herr_t H5Dclose()
2.1     0.026    0.026      1          0        26 herr_t H5check_version()
0.6     0.008    0.008      1          0        8 hid_t H5Screate_simple()
0.2     0.002    0.002      1          0        2 herr_t H5Tset_order()
0.2     0.002    0.002      1          0        2 hid_t H5Tcopy()
0.1     0.001    0.001      1          0        1 herr_t H5Sclose()
0.1     0.001    0.001      2          0        0 herr_t H5open()
---------------------------------------------------------------------------------------
```
TAU Measurement Approach

- Portable and scalable parallel profiling solution
  - Multiple profiling types and options
  - Event selection and control (enabling/disabling, throttling)
  - Online profile access and sampling
  - Online performance profile overhead compensation

- Portable and scalable parallel tracing solution
  - Trace translation to OTF, EPILOG, Paraver, and SLOG2
  - Trace streams (OTF) and hierarchical trace merging

- Robust timing and hardware performance support

- Multiple counters (hardware, user-defined, system)

- Performance measurement of I/O and Linux kernel
TAU Measurement Mechanisms

- Parallel profiling
  - Function-level, block-level, statement-level
  - Supports user-defined events and mapping events
  - Support for flat, callgraph/callpath, phase profiling
  - Support for parameter and context profiling
  - Support for tracking I/O and memory (library wrappers)
  - Parallel profile stored (dumped, snapshot) during execution

- Tracing
  - All profile-level events
  - Inter-process communication events
  - Inclusion of multiple counter data in traced events
Parallel Performance Profiling

- Flat profiles
  - Metric (e.g., time) spent in an event (callgraph nodes)
  - Exclusive/inclusive, # of calls, child calls

- Callpath profiles (Calldepth profiles)
  - Time spent along a calling path (edges in callgraph)
  - "main=> f1 => f2 => MPI_Send" (event name)
  - TAU_CALLPATH_DEPTH environment variable

- Phase profiles
  - Flat profiles under a phase (nested phases are allowed)
  - Default "main" phase
  - Supports static or dynamic (per-iteration) phases

- Parameter and context profiling
TAU Analysis
Performance Analysis

- Analysis of parallel profile and trace measurement
- Parallel profile analysis (ParaProf)
  - Java-based analysis and visualization tool
  - Support for large-scale parallel profiles
- Performance data management framework (PerfDMF)
- Performance data mining (PerfExplorer)
- Parallel trace analysis
  - Translation to VTF (V3.0), EPILOG, OTF formats
  - Integration with Vampir / Vampir Server (TU Dresden)
  - Profile generation from trace data
- Online parallel analysis and visualization
- Integration with CUBE browser (Scalasca, UTK / FZJ)
ParaProf Profile Analysis Framework

Performance Data → PerfDMF → ParaProf
- Profiles: TAU, mpir, ompP, HPMTToolkit, Cube, HPCToolkit, Gprof, Dynaprof, PSRun
- Runtime Data Collection: Supermon, MRNet
- DBMS: PostgreSQL, MySQL, Oracle, DB2, Derby
- Internal Representation
- Profile Data
- Scripting Interface: Jython
- Call Graphs
- Histograms
- Call Trees
- Bar Charts
- Comparative Displays
- Text Displays
- Vis Package: JOGL
- 3D Displays
### Parallel Profile Analysis – pprof

#### Profiling Output

```
<table>
<thead>
<tr>
<th>Function</th>
<th>Time %</th>
<th>Inclusive</th>
<th>Excl. Call</th>
<th>Incl. Subsys</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100</td>
<td>3:11:233</td>
<td>1</td>
<td>15</td>
<td>191293268 applu</td>
</tr>
<tr>
<td></td>
<td>99.6</td>
<td>3:10:463</td>
<td>3</td>
<td>37517</td>
<td>63487925 bcast_inputs</td>
</tr>
<tr>
<td></td>
<td>67.1</td>
<td>2:08:326</td>
<td>37200</td>
<td>37200</td>
<td>3450 exchange_1</td>
</tr>
<tr>
<td></td>
<td>44.5</td>
<td>1:25:159</td>
<td>9300</td>
<td>18600</td>
<td>9157 bts</td>
</tr>
<tr>
<td></td>
<td>41.0</td>
<td>1:18:436</td>
<td>18600</td>
<td>0</td>
<td>4217 MPI_Recv()</td>
</tr>
<tr>
<td></td>
<td>29.5</td>
<td>1:16:392</td>
<td>3398</td>
<td>18600</td>
<td>6665 bts</td>
</tr>
<tr>
<td></td>
<td>26.2</td>
<td>50,142</td>
<td>50,142</td>
<td>0</td>
<td>2611 MPI_Send()</td>
</tr>
<tr>
<td></td>
<td>16.2</td>
<td>31,021</td>
<td>301</td>
<td>602</td>
<td>10096 rhs</td>
</tr>
<tr>
<td></td>
<td>3.9</td>
<td>7,591</td>
<td>8,300</td>
<td>0</td>
<td>807 jacl</td>
</tr>
<tr>
<td></td>
<td>3.4</td>
<td>6,590</td>
<td>6,590</td>
<td>0</td>
<td>10918 exchange_3</td>
</tr>
<tr>
<td></td>
<td>3.4</td>
<td>4,989</td>
<td>4,989</td>
<td>0</td>
<td>703 jacl</td>
</tr>
<tr>
<td></td>
<td>2.6</td>
<td>4,989</td>
<td>4,989</td>
<td>0</td>
<td>8206 MPI_Wait()</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>0.44</td>
<td>400</td>
<td>0</td>
<td>400081 init_com</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>398</td>
<td>399</td>
<td>1</td>
<td>399634 MPI_Init()</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>140</td>
<td>247</td>
<td>1</td>
<td>247066 setiv</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>131</td>
<td>131</td>
<td>0</td>
<td>2 exact</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>103</td>
<td>103</td>
<td>0</td>
<td>103168 emhs</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>966</td>
<td>96</td>
<td>1</td>
<td>96458 read_input</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>95</td>
<td>95</td>
<td>0</td>
<td>10603 MPI_Init()</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>26</td>
<td>44</td>
<td>0</td>
<td>7337 error</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>24</td>
<td>24</td>
<td>0</td>
<td>1 MPI_Irecv()</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>15</td>
<td>15</td>
<td>0</td>
<td>15630 MPI_Finalize()</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>4</td>
<td>12</td>
<td>1700</td>
<td>1 MPI_Type_contiguous()</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>7</td>
<td>8</td>
<td>3</td>
<td>2832 norm</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>3</td>
<td>3</td>
<td>8</td>
<td>491 MPI_Allreduce()</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>3874 pntgr</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1007 MPI_Barrier()</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>0.116</td>
<td>0.837</td>
<td>1</td>
<td>4 exchange_4</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>0.512</td>
<td>0.512</td>
<td>0</td>
<td>512 MPI_Request_create()</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>0.121</td>
<td>0.353</td>
<td>0</td>
<td>353 exchange_5</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>0.024</td>
<td>0.124</td>
<td>0</td>
<td>191 exchange_6</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>0.103</td>
<td>0.103</td>
<td>0</td>
<td>17 MPI_Type_contiguous()</td>
</tr>
</tbody>
</table>
```

---

**CEADAM**  
**Advances in the TAU Performance System**  
**August 25, 2011**
Performance Data Management

- Provide an open, flexible framework to support common data management tasks
  - Foster multi-experiment performance evaluation
- Extensible toolkit to promote integration and reuse across available performance tools (PerfDMF)
  - Supported multiple profile formats: TAU, CUBE (Scalasca), gprof, mpiP, psrun, ...
  - Supported DBMS: PostgreSQL, MySQL, Oracle, DB2, Derby/Cloudscape
Parallel Profile Analysis – ParaProf

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

Multiple PerfDMF DBs
ParaProf – Single Thread of Execution View

node, context, thread

8K processors

Miranda
- hydrodynamics
- Fortran + MPI
- LLNL BG/L
ParaProf – Full Profile and Comparative Views

- Full profile
  - threads
  - events

- Comparative
  - three event axes
  - one event color

bargraph view

landscape view
Using TAU

- Install TAU
  
  \%
  configure [options]; make clean install

- Modify application makefile and choose TAU configuration
  - Select TAU’s stub makefile
  - Change name of compiler in makefile

- Set environment variables
  - Directory where profiles/traces are to be stored/counter selection
  - TAU options

- Execute application
  
  \%
  mpirun -np <procs> a.out

- Analyze performance data
  - paraprof, vampir, pprof, paraver …
For more information

- TAU Website: http://tau.uoregon.edu
  - Software
  - Release notes
  - Documentation

- TAU LiveDVD: http://www.hpclinux.com
  - Boot up on your laptop or desktop
  - Includes TAU and variety of other packages
  - Include documentation and tutorial slides
Performance Data Mining / Analytics

- Conduct systematic and scalable analysis process
  - Multi-experiment performance analysis
  - Support automation, collaboration, and reuse
- Performance knowledge discovery framework
  - Data mining analysis applied to parallel performance data
    - comparative, clustering, correlation, dimension reduction, …
  - Use the existing TAU infrastructure
- PerfExplorer v1 performance data mining framework
  - Multiple experiments and parametric studies
  - Integrate available statistics and data mining packages
    - Weka, R, Matlab / Octave
  - Apply data mining operations in interactive enviroment
How to explain performance?

- Should not just redescribe the performance results
- Should explain performance phenomena
  - What are the causes for performance observed?
  - What are the factors and how do they interrelate?
  - Performance analytics, forensics, and decision support
- Need to add knowledge to do more intelligent things
  - Automated analysis needs good informed feedback
    - iterative tuning, performance regression testing
  - Performance model generation requires interpretation
- We need better methods and tools for
  - Integrating meta-information
  - Knowledge-based performance problem solving
Metadata Collection

- Integration of XML metadata for each parallel profile
- Three ways to incorporate metadata
  - Measured hardware/system information
    - CPU speed, memory in GB, MPI node IDs, …
  - Application instrumentation (application-specific)
    - TAU_METADATA() used to insert any name/value pair
    - Application parameters, input data, domain decomposition
  - PerfDMF data management tools can incorporate an XML file of additional metadata
    - Compiler flags, submission scripts, input files, …
PerfExplorer 2.0

- Performance data mining framework
  - Parallel profile performance data and metadata
- Programmable, extensible workflow automation
- Rule-based inference for expert system analysis
S3D Scalability Study

- S3D flow solver for simulation of turbulent combustion
  - Targeted application by DOE SciDAC PERI tiger team
- Performance scaling study (C2H4 benchmark)
  - Cray XT3+XT4 hybrid, XT4 (ORNL, Jaguar)
  - IBM BG/P (ANL, Intrepid)
  - Weak scaling (1 to 12000 cores)
  - Evaluate scaling of code regions and MPI
- Demonstration of scalable performance measurement, analysis, and visualization
- Understanding scalability
  - Requires environment for performance investigation
  - Performance factors relative to main events
S3D Total Runtime Breakdown by Events (Jaguar)
S3D FP Instructions / L1 Data Cache Miss (Jaguar)
S3D Total Runtime Breakdown by Events (Intrepid)

Total BGP Timers Breakdown for s3d:intrepid-scaling-c2h4

Percentage of Total BGP Timers

Number of Processors

- GETRATES_I [getrates_i.pp.f] (40,7)–(99,9)
- Loop: DERIVATIVE_X_CALC [[derivative_x.pp.f90] (429,10)–(438,15)]
- Loop: DERIVATIVE_Y_CALC [[derivative_y.pp.f90] (428,10)–(437,15)]
- Loop: DERIVATIVE_Z_CALC [[derivative_z.pp.f90] (432,10)–(441,15)]
- Loop: INTEGRATE [[integrate.erl.pp.f90] (75,3)–(95,13)]
- Loop: RATT_I [getrates_i.pp.f] (455,7)–(666,12)
- Loop: RATT_I [getrates_i.pp.f] (668,7)–(670,11)
- Loop: RATX_I [getrates_i.pp.f] (1560,7)–(1760,11)
- Loop: RATX_I [getrates_i.pp.f] (1761,7)–(1961,12)
- Loop: RHF [rhost.pp.f90] (213,3)–(213,7)
- Loop: RHF [rhost.pp.f90] (539,3)–(545,16)
- Loop: RHF [rhost.pp.f90] (547,3)–(553,16)
- Loop: THERMCHM_M::CALC_TEMP [[thermchem_m.pp.f90] (177,5)–(189,9)]
- Loop: TRANSPORT_M::COMPUTE_COEFFICIENTS [[mixavg_transport_m.pp.f90] (494,5)–(522,9)]
- Loop: TRANSPORT_M::COMPUTE_HEATFLUX [[mixavg_transport_m.pp.f90] (777,5)–(785,19)]
- Loop: TRANSPORT_M::COMPUTESPECIES_DIFFFLUX [[mixavg_transport_m.pp.f90] (631,5)–(657,19)]
- MPL.Barrier()  MPL.Isend()  MPL.Wait()  RATT_I [getrates_i.pp.f] (103,7)–(716,9)  RATX_I [getrates_i.pp.f] (1013,7)–(1964,9)  RHF [rhost.pp.f90] (3,1)–(711,19)  other

MPI
S3D Relative Efficiency / Speedup by Event (Jaguar)
S3D Relative Efficiency by Event (Intrepid)
S3D Event Correlation to Total Time (Jaguar)

$r = 1$ implies direct correlation
S3D Event Correlation to Total Time (Intrepid)
S3D 3D Correlation Cube (Intrepid, MPI_Wait)

12K cores

GETRATES_I vs. RATI_I vs. RATX_I vs. MPI_Wait
S3D Computational Structure

- Domain decomposition with wavefront evaluation and recursion dependences in all 3 grid directions
- Communication affected by cell location

4x4 example:
- 2 neighbors
- 3 neighbors
- 4 neighbors

Data: Sweep3D on Linux Cluster, 16 processes

CEA DAM

Advances in the TAU Performance System

August 25, 2011
S3D on a Hybrid System (Cray XT3 + XT4)

- 6400 core execution on transitional Jaguar configuration

MPI_Wait

ORNL Jaguar
* Cray XT3/XT4
* 6400 cores
Gap happens to be only XT3 nodes

\textit{MPI\_Wait} takes less time, other routines take more time
Scatterplot (S3D, XT3+XT4)

- Scatterplot of top three events colored by total time
- Two clusters
- Memory speed accounts for performance difference!!!
- Process metadata can identify the different nodes
S3D Run on XT4 Only

- Better balance across nodes
- More uniform performance
Capturing Analysis and Inference Knowledge

Create PerfExplorer workflow for S3D case study?

1. Load Data
2. Extract Non-callpath data
3. Extract top 5 events
4. Load metadata
5. Correlate events With metadata
6. Process inference rules
PerfExplorer Workflow Applied to S3D Performance

--------- JPython test script start ---------
doing single trial analysis for sweep3d on jaguar
Loading Rules...
Reading rules: rules/GeneralRules.drl... done.
Reading rules: rules/ApplicationRules.drl... done.
Reading rules: rules/MachineRules.drl... done.
loading the data...
Getting top 10 events (sorted by exclusive time)... Firing rules...
MPI_Recv(): "CALLS" metric is correlated with the metadata field "total neighbors".
The correlation is 1.0 (direct).
MPI_Send(): "CALLS" metric is correlated with the metadata field "total Neighbors".
The correlation is 1.0 (direct).
MPI_Send(): "P_WALL_CLOCK_TIME:EXCLUSIVE" metric is correlated with the metadata field "total neighbors".
The correlation is 0.8596 (moderate).
SOURCE [{source.f} {2,18}]: "PAPI_FP_INS:EXCLUSIVE" metric is inversely correlated with the metadata field "Memory Speed (MB/s)".
The correlation is -0.9792 (very high).
SOURCE [{source.f} {2,18}]: "PAPI_FP_INS:EXCLUSIVE" metric is inversely correlated with the metadata field "Seastar Speed (MB/s)".
The correlation is -0.9785 (very high).
SOURCE [{source.f} {2,18}]: "PAPI_L1_TCA:EXCLUSIVE" metric is inversely correlated with the metadata field "Memory Speed (MB/s)".
The correlation is -0.9818 (very high).
SOURCE [{source.f} {2,18}]: "PAPI_L1_TCA:EXCLUSIVE" metric is inversely correlated with the metadata field "Seastar Speed (MB/s)".
The correlation is -0.982 (very high).
SOURCE [{source.f} {2,18}]: "P_WALL_CLOCK_TIME:EXCLUSIVE" metric is inversely correlated with the metadata field "Memory Speed (MB/s)".
The correlation is -0.998 (very high).
SOURCE [{source.f} {2,18}]: "P_WALL_CLOCK_TIME:EXCLUSIVE" metric is inversely correlated with the metadata field "Seastar Speed (MB/s)".
The correlation is -0.996 (very high).
...done with rules.
--------- JPython test script end ---------
NWChem and One-sided Communication

- NWChem is a leading chemistry modeling code
- NWChem relies on Global Arrays (GA)
  - GA is a PGAS programming model
  - Provides a global view of a physically distributed array
  - One-sided access to arbitrary patches of data
  - Developed as a library (fully interoperable with MPI)
- Aggregate Remote Memory Copy Interface (ARMCI)
  - GA communication substrate for one-sided communication
  - Portable high-performance one-sided communication library
  - Rich set of remote memory access primitives
- Difficult to test representative workloads for NWChem
  - Lack of use cases for one-side programming models
NWChem Characterization

- Strong-scaling of modest problems helps to understand the behavior of larger scientifically significant problems
  - Represent behavior of real calculations on future systems
- Understand interplay between data-server and compute processes as a function of scaling
  - Large numerical computation per node at small scale can obscure the cost of maintaining passive-target progress
  - Larger scale decreases numerical work per node and increases the fragmentation of data, increasing messages
  - Vary #nodes, cores-per-node, and memory buffer pinning
- Understand trade-off of core allocation
  - All to computation versus some to communication
- CCPE paper just accepted on NWChem performance analysis
NWChem Instrumentation

- Source-base instrumentation of NWChem application
- Developed an ARMCI interposition library (PARMCI)
  - Defines weak symbols and name-shifted PARMCI interface
  - Similar to PMPI for MPI
- Developed a TAU PARMCI library
  - Intervals events around interface routines
  - Atomic events capture communication size and destination
- Wrapped external libraries
  - BLAS (DGEMM)
- Need portable instrumentation for cross-platform runs
- Systems
  - Fusion: Linux cluster, Pacific Northwest National Lab
  - Intrepid: IBM BG/P, Argonne National Lab
FUSION Tests with Varying Cores (no pinning)

- Scaling on 24, 32, 48, 64, 96 and 128 nodes
- Test on 8 and 7 cores with pinning disabled
  - Dedicated data server with 7 cores
- Relative ARMI communication overhead increases with greater number of nodes (cores)
FUSION Tests with Varying Cores (with pinning)

- Pinning communication buffers shows dramatic effects
- Speedups now possible
- Relative ARMI communication overhead increases, but does not grow to dominate
Intrepid Tests

- Scaling on 64, 128, 256 and 512 nodes
- Tests with interrupt or communication helper thread (CHT)
  - CHT requires a core to be allocated
- ARMCI calls are barely noticeable
- DAXPY calculation shows up more
- CHT performs better in both SMP and DUAL modes
Hybrid Performance Measurement

- Different approaches for observing parallel performance
- *Sampling-based measurement* (SBM)
  - Event-based sampling (EBS)
  - Instruction-based sampling (IBS)
  - Example: PerfSuite, HPCToolkit, ...
- *Probe-based measurement* (PBM)
  - Instrumentation of program code
  - Example: TAU, Scalasca, ...
- Combine to exploit advantages of both PBM+SBM
- TAUebs
  - TAU for probe-based instrumentation and measurement
  - EBS measurement with callstack unwinding
  - Capture profiles and traces
Integrated PBS + EBS Measurement Design (1)

Sampling Timeline

Sample-Based Profile
- PC addr range
  - 0x2110
  - 0x3820
  - 0x4894

Integration?

Probe-Based Profile
- TAU event stack

Instrumentation Timeline

Integration?
Context Merging

- Get more information if can merge contexts
  - SBM context is the PC + callstack
  - PBM context is the event stack
- At interrupt, SBM can reach over into PBM
  - “Sample” PBM context and couple with SBM context
  - Encoding TAU context
    - map TAU event stack into a TAU key
Integrated PBS + EBS Measurement Design (2)

Sampling Timeline

Probe-Based Profile

TAU event stack (TAUkey)

0x2110
0x3820
0x4894

Sampling-Based Profile

PC Histogram

PC addr range

0x3820
0x4894
0x2110

Sampling-Based Trace

Samples since last flush

0x3820
0x4894
0x2110

TAUkey: 0 1 2

Integration
TAU Event Stack and PC Callstack

Loop1 and Loop2 are not routines!
TAUebs Measurement

- What type of measurement is made with each sample?
  - Capture a trace of EBS samples (ICPP 2010 paper)
    - Timestamp
    - TAUkey (key to current TAU event stack)
    - PCkey (key to current PC call stack)
    - Hardware counters
    - Parallel profiles can be calculated from TAUebs traces

- Capture EBS sample histograms at runtime
  - Parallel profiles are maintained during execution
  - Histograms are associated with TAU event path
  - Produce TAUebs profiles in standard TAU profile format
Hybrid profiling

○ Instance of new sample contextualized by TAUkey
○ Integrate into TAU profile structures at runtime

New Sample:
Function a(), address: 0x21098

TAUkey:
“main-> ... -> foo() -> loop”

PC Histogram:
0x21098 == 15
...  
0x45362 == 23
Hybrid Profiling Implementation

- Uses existing timer-interrupt framework to trigger samples
- With each sample:
  - Query active TAU event context to determine TAUkey
  - Create/update PC address histogram for the active TAU event context represented by its key
- Addresses are resolved to meaningful symbol information via BFD at the end of the run
- TAU event context controlled by the event path depth
## TAUebs Flat Profiling

- **Pure sampling** – only `main()` is instrumented.

### TAU: ParaProf Thread Statistics:

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Exclusive TIME</th>
<th>Inclusive TIME</th>
<th>Calls</th>
<th>Child Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

### Application Stack Trace:

```
<table>
<thead>
<tr>
<th>Function</th>
<th>Time</th>
<th>Calls</th>
<th>Child Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

### Example Stack Trace:

- Application
  - .a
  - .so

---

**CEA DAM**

**Advances in the TAU Performance System**

**August 25, 2011**
TAUebs Hybrid Profile

- Mix of sampling and probed-based instrumentation

C: main

C: main → a

C: main → a → main_foo

C: main → a → main_foo → main_bar
## TAUebs Hybrid Profiling with Loop Events

- **TAU** events not restricted to routines (e.g., blocks, loops)

### Screen Shot

<table>
<thead>
<tr>
<th>Name</th>
<th>Exclusive TIME</th>
<th>Inclusive TIME</th>
<th>Calls</th>
<th>Child Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>int main(int, char **)</td>
<td>27.654</td>
<td>38.81</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>double a(int)</td>
<td>0</td>
<td>27.579</td>
<td>27.579</td>
<td>0</td>
</tr>
<tr>
<td>Loop: double a(int)</td>
<td>0</td>
<td>7.619</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>(INTERMEDIATE) Loop: double a(int)</td>
<td>0</td>
<td>1.901</td>
<td>1.901</td>
<td>0</td>
</tr>
</tbody>
</table>

**C: main → a → loop[38,3]**

*Samples show performance relative to line number*
Hybrid Profiling with NAMD

- NAMD is a molecular modeling application implemented with Charm++ programming model
- Charm++ exposes its programming and runtime constructs
- Callback system for TAU for probed-based measurement

Uninstrumented runtime routines

Hopper, Cray XE6
144 processors
Hybrid Profiling with FLASH4

Simulation of astrophysical thermonuclear flashes

Follow probed event path to end and pick up samples to highlight significant code points

Hopper, Cray XE6
144 processors
## Advances in the TAU Performance System

### FLASH4 – Reverse Event/Call Path

- **Aggregate events / samples performance data**

![TAU events within which this memcpy sample occurs](image)

<table>
<thead>
<tr>
<th>Name</th>
<th>Executable TIME</th>
<th>Inclusive TIME</th>
<th>Calls</th>
<th>Child Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY_PPM_BLOCK [hy_ppm_block.F90]</td>
<td>3.642</td>
<td>27.069</td>
<td>52,339,778</td>
<td>523,398,778</td>
</tr>
<tr>
<td>AMR_1BLK_GUARDCELL_SRL [amr_1blk_guardecell_srl.F90]</td>
<td>2.907</td>
<td>2.92</td>
<td>77,447,889</td>
<td>404,75</td>
</tr>
<tr>
<td>[SAMPLE] amr_1blk_ec_cp_remole (!(null))[0.0]-[0.0])</td>
<td>2.737</td>
<td>2.737</td>
<td>2,736,556</td>
<td></td>
</tr>
<tr>
<td>MPI_Waitall()</td>
<td>2.457</td>
<td>2.457</td>
<td>47,013,688</td>
<td></td>
</tr>
<tr>
<td>[SAMPLE] interp (!(null))[0.0]-[0.0])</td>
<td>2.426</td>
<td>2.426</td>
<td>2,426,354</td>
<td></td>
</tr>
<tr>
<td>[SAMPLE] memcpy [usr/src/packages/BUILD/ribc-2.9/string/...sysdeps/x86_64/memcpy.S]</td>
<td>2.408</td>
<td>2.408</td>
<td>2,408,014</td>
<td></td>
</tr>
<tr>
<td>[INTERMEDIATE] MPI_Send()</td>
<td>1.727</td>
<td>1.727</td>
<td>1,726,59</td>
<td></td>
</tr>
<tr>
<td>[INTERMEDIATE] MPI_AMR_READ_RESTRICT_COMM [mpi_amr_store_comm_info.F90]</td>
<td>0.229</td>
<td>0.229</td>
<td>228,535</td>
<td></td>
</tr>
<tr>
<td>[INTERMEDIATE] MPI_Waitall()</td>
<td>0.172</td>
<td>0.172</td>
<td>172,465</td>
<td></td>
</tr>
<tr>
<td>[INTERMEDIATE] MPI_AMR_READ_FLUX_COMM [mpi_amr_store_comm_info.F90]</td>
<td>0.13</td>
<td>0.13</td>
<td>130,285</td>
<td></td>
</tr>
<tr>
<td>[INTERMEDIATE] MPI_AMR_WRITE_COMM [mpi_amr_store_comm_info.F90]</td>
<td>0.093</td>
<td>0.093</td>
<td>93,41</td>
<td></td>
</tr>
<tr>
<td>[INTERMEDIATE] AMR_CHECK_DEREFINE [mpi_amr_derefine_blocks.F90]</td>
<td>0.023</td>
<td>0.023</td>
<td>22,562</td>
<td></td>
</tr>
<tr>
<td>[INTERMEDIATE] MPI_Init()</td>
<td>4.462</td>
<td>4.462</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[INTERMEDIATE] MPI_WARN()</td>
<td>4.438</td>
<td>4.438</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[INTERMEDIATE] MPIadastro()</td>
<td>3.923</td>
<td>3.923</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[INTERMEDIATE] MPI_Read()</td>
<td>3.879</td>
<td>3.879</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[INTERMEDIATE] MPI_Bcast()</td>
<td>0.004</td>
<td>0.004</td>
<td>3.705</td>
<td></td>
</tr>
<tr>
<td>[INTERMEDIATE] MPI_AMR_WRITE_COMM [mpi_amr_store_comm_info.F90]</td>
<td>0.003</td>
<td>0.003</td>
<td>3.133</td>
<td></td>
</tr>
<tr>
<td>[INTERMEDIATE] MPI_AMR_WRITE_COMM [mpi_amr_store_comm_info.F90]</td>
<td>0.003</td>
<td>0.003</td>
<td>2.617</td>
<td></td>
</tr>
<tr>
<td>[INTERMEDIATE] MPI_BARRIER()</td>
<td>0.002</td>
<td>0.002</td>
<td>1.9</td>
<td></td>
</tr>
<tr>
<td>[INTERMEDIATE] MPI_Recv()</td>
<td>0.001</td>
<td>0.001</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>[INTERMEDIATE] MPI_Intercomm()</td>
<td>0.001</td>
<td>0.001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[SAMPLE] multispecies_setsmufrac [!(null)][0.0]-[0.0])</td>
<td>2.238</td>
<td>2.238</td>
<td>2,238,749</td>
<td></td>
</tr>
<tr>
<td>[SAMPLE] multispecies_getproperty [!(null)][0.0]-[0.0])</td>
<td>2.023</td>
<td>2.023</td>
<td>2,022,986</td>
<td></td>
</tr>
<tr>
<td>[SAMPLE] states [!(null)][0.0]-[0.0])</td>
<td>1.822</td>
<td>1.822</td>
<td>1,822,16</td>
<td></td>
</tr>
<tr>
<td>[SAMPLE] eos [eos.F90]</td>
<td>1.795</td>
<td>44.124</td>
<td>1,451,798.5</td>
<td>1,451,798.5</td>
</tr>
<tr>
<td>[SAMPLE] monot [!(null)][0.0]-[0.0])</td>
<td>1.73</td>
<td>1.73</td>
<td>1,730,194</td>
<td></td>
</tr>
<tr>
<td>[SAMPLE] amr_guardecell ![null][0.0]-[0.0])</td>
<td>1.645</td>
<td>1.645</td>
<td>1,644,826</td>
<td></td>
</tr>
<tr>
<td>[SAMPLE] HY_PPM_SWEEP [hy_ppm_sweep.F90]</td>
<td>1.578</td>
<td>119.099</td>
<td>8,000</td>
<td>644,084,333</td>
</tr>
<tr>
<td>[SAMPLE] hydro_1d ![null][0.0]-[0.0])</td>
<td>1.527</td>
<td>1.527</td>
<td>1,527,404</td>
<td></td>
</tr>
<tr>
<td>[SAMPLE] eigensystem.1622.clone.1 ![Interp_char.instr.F90][0.0]-[0.0])</td>
<td>1.5</td>
<td>1.5</td>
<td>1,499,903</td>
<td></td>
</tr>
<tr>
<td>[SAMPLE] multispecies_setsmunw [!(null)][0.0]-[0.0])</td>
<td>1.473</td>
<td>1.473</td>
<td>1,473,479</td>
<td></td>
</tr>
<tr>
<td>[SAMPLE] burn [burn.F90]</td>
<td>1.396</td>
<td>55.819</td>
<td>4,009</td>
<td>1,597,228,667</td>
</tr>
</tbody>
</table>

---

**CEA DAM**

**Advances in the TAU Performance System**

**August 25, 2011**
TAUebs Trace with Performance Metrics

- TAU events: main, multiply, slow_matmut, fast_matmult
- Extract hardware counters from EBS trace
Vampir call stack color-coded by file name

Flops rate shown (with respect to context)
GPAW – Density Function Theory

- Mixed Python, C, and MPI run on 128 processes
- Python performance interface and LD_PRELOAD
TAUebs Trace for FLASH (240 processes, 60 ranks)

Color-coded by filename

Alternating patterns due to dual mode

40 secs
Heterogeneous Parallel Systems and Performance

- Heterogeneous parallel systems are highly relevant today
  - Multi-CPU, multicore shared memory nodes
  - Manycore (throughput) accelerators with high-BW I/O
  - Cluster interconnection network

- Performance is the main driving concern
  - Heterogeneity is an important path to extreme scale

- Heterogeneous software technology to get performance
  - More sophisticated parallel programming environments
  - Integrated parallel performance tools
    - support heterogeneous performance model and perspectives
Implications for Parallel Performance Tools

- Current status quo is somewhat comfortable
  - Mostly homogeneous parallel systems and software
  - Shared-memory multithreading – OpenMP
  - Distributed-memory message passing – MPI

- Parallel computational models are relatively stable (simple)
  - Corresponding performance models are relatively tractable
  - Parallel performance tools can keep up and evolve

- Heterogeneity creates richer computational potential
  - Results in greater performance diversity and complexity

- Heterogeneous systems will utilize more sophisticated programming and runtime environments

- Performance tools have to support richer computation models and more versatile performance perspectives
Heterogeneous Performance Views

- Want to create performance views that capture heterogeneous concurrency and execution behavior
  - Reflect interactions between heterogeneous components
  - Capture performance semantics relative to computation model
  - Assimilate performance for all execution paths for shared view

- Existing parallel performance tools are CPU (host)-centric
  - Event-based sampling (not appropriate for accelerators)
  - Direct measurement (through instrumentation of events)

- What perspective does the host have of other components?
  - Determines the semantics of the measurement data
  - Determines assumptions about behavior and interactions

- Performance views may have to work with reduced data
Heterogeneous Performance Measurement

- Heterogeneous performance perspectives
- Inter-node communication
  - Message communication, overhead, synchronization
- Intra-node execution
  - Multicore thread execution and interactions
- Host-GPU interactions
  - Kernel setup, memory transfer, concurrency overlap, synchronization
- GPU kernel execution
  - Use of GPU compute and memory resources
Host (CPU) - GPU Scenarios

- **Single GPU**
  - Host (CPU): Open device, Move data, Launch kernel(s), Wait, Move data
  - GPU: Run kernel(s)
  - Implemented as asynchronous calls

- **Multi-stream**
  - Host (CPU): Open device, Move data, Launch kernel(s), Wait, Move data
  - GPU: Stream 1: Run kernel(s), Stream 2: Run kernel(s)
  - Time

- **Multi-CPU, Multi-GPU**
  - Thread (CPU 1): Open device, Move data, Launch kernel(s), Wait, Move data
  - GPU 1, ..., GPU k: Run kernel(s)
  - Thread (CPU k): Open device, Move data, Launch kernel(s), Wait, Move data
  - Time
Host-GPU Measurement – Synchronous Method

- Consider three measurement approaches
  - Synchronous, Event queue, Callback
- Synchronous events on the CPU
- Some inaccuracy with actual kernel start/stop
Host-GPU Measurement – Event Queue Method

- Events queued in GPU stream
- Events are measured by GPU
- Performance information read at sync points on CPU
- Events must be placed around kernel launch
Host-GPU Measurement – Callback Method

- GPU driver libraries provide callbacks for certain routines
- Measurement tool registers the callbacks
  - Application code is not modified
- Measurements occur at callback
  - GPU performance measurements might be made internally
Method Support and Implementation

- Synchronous method
  - Place instrumentation appropriately around GPU calls (kernel launch, library routine, …)
  - Wrap (synchronous) library with performance tool

- Event queue method
  - Utilize CUDA and OpenCL event support
  - Again, need instrumentation to create and insert events in the streams with kernel launch and process events
  - Can be implemented with driver library wrapping

- Callback method
  - Utilize language-level callback support in OpenCL
  - Utilize NVIDIA CUDA Performance Tool Interface (CUPTI)
  - Need to appropriately register callbacks
GPU Performance Measurement Tools

- Support the Host-GPU performance perspective
- Provide integration with existing measurement system to facilitate tool use
- Utilize support in GPU driver library and device

Tools
- TAU performance system
- Vampir
- PAPI
- NVIDIA CUPTI
GPU Performance Tool Interoperability

The diagram illustrates the interoperability between various GPU performance tools and frameworks. The tools include CUDA, OpenCL, CUPTI, TAU, PAPI, VampirTrace, ParaProf, and Vampir. The diagram shows how these tools communicate and exchange data through the use of interfaces like tau2otf.

Key elements of the diagram:
- CUDA and OpenCL are connected to CUPTI, indicating integration at the performance monitoring level.
- TAU, PAPI, and VampirTrace are central nodes for data exchange, with TAU receiving parallel profile data and VampirTrace connecting to Vampir.
- ParaProf is shown handling parallel profiles, with an event queue and callback mechanism for data flow.

This diagram provides a visual representation of how different GPU performance tools can be integrated for comprehensive monitoring and analysis.
NVIDIA CUPTI

- NVIDIA is developing CUPTI to enable the creation of profiling and tracing tools
- Callback API
  - Interject tool code at the entry and exist to each CUDA runtime and driver API call
- Counter API
  - Query, configure, start, stop, and read the counters on CUDA-enabled devices
- CUPTI is delivered as a dynamic library
- CUPTI is released with CUDA 4.0
TAU for Heterogeneous Measurement

- Support multiple performance perspectives
- Integrate Host-GPU support in TAU measurement
  - Enable use of each measurement approach
  - Include use of PAPI and CUPTI
  - Provide profiling and tracing support
- Additional support
  - TAU wrapping of libraries
  - Use of tau_exec to work with binaries
Example: SDK simpleMultiGPU

- Demonstration of multiple GPU device use
- Program structure:
  
  \[ \text{main} \rightarrow \text{solverThread} \rightarrow \text{reduceKernel} \]

- One node with three GPUs
- Performance profile for:
  - One \textit{main} thread
  - Three \textit{solverThread} threads
  - Three \textit{reduceKernel} “threads”
Identified a known overhead in GPU context creation
# simpleMultiGPU Profile

## Main Thread

<table>
<thead>
<tr>
<th>Name</th>
<th>Exclusive TIME</th>
<th>Inclusive TIME ▼</th>
<th>Calls</th>
<th>Child Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>main [{simpleMultiGPU.cpp} {105,0}]</td>
<td>1,543.18</td>
<td>20,997.786</td>
<td>12</td>
<td>17</td>
</tr>
<tr>
<td>cutWaitForThreads [{multithreading.cpp} {65,0}]</td>
<td>0.006</td>
<td>19,450.196</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>cutEndThread [{multithreading.cpp} {55,0}]</td>
<td>19,450.19</td>
<td>19,450.19</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>cudaError_t cudaGetDeviceCount(int *) C</td>
<td>4.342</td>
<td>4.342</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>cutStartThread [{multithreading.cpp} {48,0}]</td>
<td>0.064</td>
<td>0.064</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>cudaError_t cudaThreadExit(void) C</td>
<td>0.004</td>
<td>0.004</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

## Solver Thread

<table>
<thead>
<tr>
<th>Name</th>
<th>Exclusive TIME</th>
<th>Inclusive TIME ▼</th>
<th>Calls</th>
<th>Child Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>solverThread(TGPUplan*) [{simpleMultiGPU.cpp} {47,0}]</td>
<td>0.103</td>
<td>18,767.447</td>
<td>9</td>
<td>21</td>
</tr>
<tr>
<td>cudaError_t cudaMalloc(void **, size_t) C</td>
<td>18,743.914</td>
<td>18,743.914</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>cudaError_t cudaMemcpy(void *, const void *, size_t, enum cudaMemcpyKind) C</td>
<td>20.108</td>
<td>20.108</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>cudaError_t cudaFree(void *) C</td>
<td>2.007</td>
<td>2.007</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>cudaError_t cudaThreadSynchronize(void) C</td>
<td>1.286</td>
<td>1.286</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>cudaError_t cudaSetDevice(int) C</td>
<td>0.013</td>
<td>0.013</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>cudaError_t cudaLaunch(const char *) C</td>
<td>0.012</td>
<td>0.012</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>cudaError_t cudaConfigureCall(dim3, dim3, size_t, cudaStream_t) C</td>
<td>0.002</td>
<td>0.002</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>cudaError_t cudaSetupArgument(const void *, size_t, size_t) C</td>
<td>0.002</td>
<td>0.002</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

## GPU Execution

<table>
<thead>
<tr>
<th>Name</th>
<th>Exclusive TIME</th>
<th>Inclusive TIME ▼</th>
<th>Calls</th>
<th>Child Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>.TAU application</td>
<td>0</td>
<td>0.865</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>reduceKernel</td>
<td>0.865</td>
<td>0.865</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
SHOC FFT Profile with Callsite Info

- Scalable HeterOgeneous Computing benchmarks (ORNL)
  - Programs to test performance on heterogeneous systems
  - CUDA and OpenCL versions
- TAU associate callsite information with kernel launch
  - Different kernel calls can be distinguished

Each kernel (ifft1D_512, fft1D_512 and chk1D_512) is broken down by call-site, either during the **single** precession or **double** precession step.
CUDA Linpack Profile (4 processes, 4 GPUs)

- GPU-accelerated Linpack benchmark (NVIDIA)
CUDA Linpack Trace

CUDA memory transfer (white)  MPI communication (yellow)
SHOC Stencil2D

- Compute 2D, 9-point stencil
  - Multiple GPUs using MPI
  - CUDA and OpenCL versions

- Experiments:
  - One node with 3 GPUs
  - Eight nodes with 24 GPUs

- Performance profile and trace
  - Application events
  - Communication events
  - Kernel execution
Stencil2D Parallel Profile / Trace

Metric: TAUGPU_TIME
Value: Exclusive

- Std. Dev.
- Mean
- node 0, thread 0
- node 0, thread 1
- node 1, thread 0
- node 1, thread 1
- node 2, thread 0
- node 2, thread 1

Process 0
CUDA[0] 0:1
Process 1
CUDA[1] 1:1
Process 2
CUDA[0] 2:1
Process 3
CUDA[1] 3:1

CUDA[0] 0:1, Values of Counter "l1_shared_bank_conflict" over Time
CUDA[0] 0:1, Values of Counter "threads_per_kernel" over Time
CUDA[0] 0:1, Values of Counter "threads_per_block" over Time

All Processes, Accumulated Exclusive Time per Function
18.997 ms
14.885 ms
12.158 ms
4.891 ms
4.797 ms
Application
CUDA_KERNEL
CUDA_KERNEL
CUDART_API
CUDA_SYNC
Communication Matrix View
Average Bandwidth
Process 0
CUDA[0] 0:1
320 MiB/s
280 MiB/s
240 MiB/s
Process 1
CUDA[1] 1:1
200 MiB/s
160 MiB/s
Process 2
CUDA[0] 2:1
120 MiB/s
80 MiB/s
Process 3
CUDA[1] 3:1
40 MiB/s
0 MiB/s
Stencil2D Parallel Profile
Stencil2D Trace (512 iterations, 4 CPUxGPU)

CUDA memory transfer (white)
NAMD Trace using CUPTI
Performance Visualization – Motivation

- Large performance data presents interpretation challenges
- Visualization aids in data exploration and pattern analysis
  - 3D visualization can help in identifying relations between events/metrics
- Existing tools provide “canned” views
  - TAU provides a few
    - 2D: bargraph, histogram
    - 3D: full profile, correlation
- Developing new visualizations is a challenge
  - Strategy 1: Create new view for each problem
  - Strategy 2: Use external visualization environment
- Provide high-level support to use within existing framework
Extending Visualization Support in Profile Tool

- User defines visualization
  - Based on performance data model
- Specifies layout based on events, metrics, and metadata
- UI provides control of data binding and visualization
Using Process Topology Metadata

- Inspired by the Scalasca CUBE topology display
- Each point represents a thread of execution (MPI process)
  - Positioned according to the Cartesian \((x,y,z,t)\) coordinates
- Color is determined by selected event/metric value
- Topology information can be recorded in TAU metadata
- ParaProf reads metadata to determine topology and create layout
- Sweep3D is a 3D neutron transport application
  - 16K run on BG/P
  - Color is exclusive time in the “sweep” function
Topology Control UI

- **Layout** tab allows customization of the position and visibility of data points.
- Performance event/metric data used to define color and position is selected in the **Event** tab.
- Additional rendering options, such as color scale and point size are available.
- 4k-core S3D run on BG/P.
Alternate Topologies

- Certain views may hide deeper inter-process behavior
- Spatially dependent performance issues may be revealed by manipulating topology
- Sweep3D profile with alternative Cartesian mapping exposes distribution of computational effort
- Topology has direct effect on communication
- Visualization mapped to hardware topologies can suggest better node/rank mapping

- `MPI_AllReduce()` values for Sweep3D highlights waiting distribution from rank 0 (lower left) to the most distant rank (upper right)
Viewing Internal Structure

- Dense topologies can hide internal structure
- Restrict visibility by color value to expose performance patterns
- ParaProf visualization UI now allows for range filtering
  - Mid-level values can be excluded
  - Remaining points are:
    - high outliers (hotspots)
    - low outliers (underutilized nodes)
Slicing to Reduce Dimensionality

- Restrict visibility to slices along the spatial axes
- Multiple axis controls allow selection of planes, lines, or an individual point
- ParaProf visualization UI provides filtering control
  - Averaging the color value for all points in the selected area
Cray XE6 Topology

- Now have ability to capture Cray XE6 topology metadata
  - Node topology and core identity
- GCRM is a global cloud resolving mode
- Visualization of 10K execution
  - Topological layout attempts to capture node cores

12-core node

MPI_Init

MPI_Allreduce
Visual Layout Specification

- Want to allow creation of explicit layouts
- Define a specification “language” that allows mathematical expressions to describe features of performance display
  - Equations define X, Y, Z coordinates and color per process
  - Event and metrics are seen as variables
    - $eventX.val$ : value for $X$th specified event and metric
    - $eventX\{min,max,mean\}$ : global aggregate values
    - $atomicY$ : $Y$th atomic event value
  - Intermediate variables can be used in the calculation
  - Defined global variables (e.g., max rank) are provided
- Specifications are loaded and processed by ParaProf
  - Use the MESP expression parser
Sphere Layout Specification

- Spatially mediated performance behavior may not be represented directly in topology metadata
  - Applications allocate resources with respect to a data-driven model
- The position of each point can be defined by custom equations in terms of event/metric, aggregate, atomic event and metadata
- Sweep3D profile mapped to a sphere

```plaintext
BEGIN_VIZ=Sphere
rootRanks=sqrt(maxRank)
theta=2*pi()/rootRanks*mod(rank,rootRanks)
phi=pi()/rootRanks*(ceil(rank/rootRanks))
x=cos(theta)*sin(phi)*100
y=sin(theta)*sin(phi)*100
z=cos(phi)*100
END_VIZ
```
ParaProf Events Panel

- Events / metrics get bound in ParaProf UI
- Example:
  - `event0` is the FLOP count for function `foo`
  - `event1` is the time value for function `foo`
  - To set the X coordinate for each process point to the FLOPS for event `foo`:
    \[ x = \text{event0.val} / \text{event1.val} \]
  - To set the Y coordinate for each process point to the global average FLOPS for event `foo`:
    \[ y = \text{event0.mean} / \text{event1.mean} \]
Adding Dimensionality

- Topologies can involve more than three dimensions (e.g., intranode)
- Mirror actual machine layout to capture communication structure and cores
- Custom layouts allow specification of multiple points from a single process/rank
- 4K-core S3D run on BG/P
- Default topology only covers X, Y, Z coordinates
- A custom topology divides each nth core into its own block

BEGIN_VIZ=4K_8x8x16Block
xdim=8
ydim=8
zdim=16

x=mod(rank,xdim)+16*floor(rank/1024)
y=mod(floor(rank/xdim),ydim)
z=mod(floor(rank/xdim/ydim),zdim)

END_VIZ
Non-Spatial Relationships

- Positioning of points needs not be with respect to physical or data topology
- Correlation of metrics within the same events or events between processes can indicate relevant performance effects
- Partitioning or clustering of different processes based on selected performance criteria
- 3D scatterplot for 10240 core run of GCRM/ZGrd application
- Correlates four selected events, one for each spatial axis plus color

```
BEGIN_VIZ=ScatterTest
restrictDim=1
x=event0.val
y=event1.val
z=event2.val
END_VIZ
```
3D Scatterplot Correlation of TAUebs Data (Sweep3D)

Sample under MPI_Allreduce

MPI_Allreduce() => [INTERMEDIATE,...]

MPI_Allreduce() (Exclusive, TIME)

MPI_Allreduce

CEA DAM Advances in the TAU Performance System August 25, 2011
Multiple Samples used to Expose Structure

Sample 1 under MPI_Allreduce

Sample 2 under MPI_Allreduce

MPI_Allreduce() => (INTERMEDIATE, EXCLUSIVE, TIME)
Visualizaton Next Steps

- Collect topology data from additional platforms (e.g. Cray)
- Expand UI for more general access to performance data model
- Allow independent manipulation of unconnected segments
- Improve presentation of data values, ranks, and metrics
- Better functionality for automatic higher-dimensional layouts
- Add representation of communication channels
Profiling PGI Accelerator Primitives

- PGI compiler allows users to annotate source code to identify loops that should be accelerated.
- When a program is compiled with TAU, its measurement library intercepts the PGI runtime library layer to measure time spent in the runtime library routines and data transfers.
- TAU also captures the arguments:
  - array data dimensions and sizes, strides
  - upload and download times
  - variable names, source file names
  - row and column information
  - routines
Example: PGI GPU-accelerated MM
PGI MM Computational Kernel

![Image of a software interface showing the execution time and source code of a computational kernel. The interface displays the details of a function call and the source code for a matrix multiplication subroutine. The code snippet includes directives for performance analysis and optimization within the TAU Profiler framework.](image)
Scalable Parallel Performance Monitoring

- Performance problem analysis is increasingly complex
  - Multi-core, heterogeneous, and extreme scale computing
  - Adaptive algorithms and runtime application tuning
  - Performance dynamics variability within/between executions

- Neo-performance measurement and analysis perspective
  - Static, offline analysis $\leftrightarrow$ dynamic, online analysis
  - Scalable runtime analysis of parallel performance data
  - Performance feedback to application for adaptive control
  - Integrated performance monitoring (measurement + query)
    - Co-allocation of additional (tool specific) system resources

- Goal
  - Scalable, integrated parallel performance monitoring
Parallel Performance Measurement and Data

- Parallel performance tools measure concurrently
  - Scaling dictates “local” measurements (profile, trace)
    - save data with "local context" (processes or threads)
  - Done without synchronization or central control
- Parallel performance state is globally distributed as a result
  - Logically part of application’s global data space
  - Offline: outputs data at end for post-mortem analysis
  - Online: access to performance state for runtime analysis
- Definition: Monitoring
  - Online access to parallel performance (data) state
  - May or may not involve runtime analysis
Monitoring for Performance Dynamics

- Runtime access to parallel performance data
  - Scalable and lightweight
  - Raises concerns of overhead and intrusion
  - Support for performance-adaptive, dynamic applications

- Alternative 1: Extend existing performance measurement
  - Create own integrated monitoring infrastructure
  - Disadvantage: maintain own monitoring framework

- Alternative 2: Couple with other monitoring infrastructure
  - Leverage scalable middleware from other supported projects
  - Challenge: measurement system / monitor integration
Performance Dynamics: Parallel Profile Snapshots

- Profile snapshots are parallel profiles recorded at runtime.
- Shows performance profile dynamics (all types allowed).

Parallel Profile Snapshots of FLASH 3.0 (UIC)

- Simulation of astrophysical thermonuclear flashes
- Snapshots show profile differences since last snapshot
  - Captures all events since beginning per thread
  - Mean profile calculated post-mortem
  - Highlight change in performance per iteration and at checkpointing

![Graph showing profile breakdown with categories Initialization, Checkpointing, and Finalization]
FLASH 3.0 Performance Dynamics (Periodic)
Empirical performance data evaluated with respect to performance expectations at various levels of abstraction.
Performance Tool Integration (PRIMA) (UO, Juelich)

- Performance tool community is small
- Strong need to integrate technology
  - Integration of instrumentation and measurement
  - Create core infrastructure (avoid duplication, share)
- DOE PRIMA project
  - University of Oregon and Research Centre Juelich
  - Focus on TAU and Scalasca
  - Refactor instrumentation, measurement, and analysis
  - Build next-generation tools on new common foundation
- Extend to involve the SILC project
  - Juelich, TU Dresden, TU Munich
- Fully-integrated measurement infrastructure in Score-P
Score-P Architecture

Event traces (OTF2)  Call-path profiles (CUBE4)  Online interface

Score-P measurement infrastructure

Application (MPI, OpenMP, hybrid)

Instrumentation  MPI wrapper

supplemental instrumentation + measurement support

Compiler  TAU instrumentor  OPARI 2  COBI

TAU adaptor

Hardware counter (PAPI)
Heterogeneous Exascale Software (Vancouver)

- DOE X-stack program
- Partners:
  Oak Ridge National Laboratory
  University of Oregon
  University of Illinois
  Georgia Institute of Technology
- Components
  - Compilers
  - Scheduling and runtime resource management
  - Libraries
  - Performance measurement, analysis, modeling
Support Acknowledgements

- Department of Energy (DOE)
  - Office of Science
  - ASC/NNSA
- Department of Defense (DoD)
  - HPC Modernization Office (HPCMO)
- NSF Software Development for Cyberinfrastructure (SDCI)
- Research Centre Juelich
- Argonne National Laboratory
- Technical University Dresden
- ParaTools, Inc.
- NVIDIA