TAU Performance System®

- Tuning and Analysis Utilities (18+ year project)
- Comprehensive performance profiling and tracing
  - Integrated, scalable, flexible, portable
  - Targets all parallel programming/execution paradigms
  - Integrated performance toolkit
- Instrumentation, measurement, analysis, visualization
  - Widely-ported performance profiling / tracing system
  - Performance data management and data mining
  - Open source (BSD-style license)
- Easy to integrate in application frameworks

http://tau.uoregon.edu
Parallel performance framework and toolkit

- Goal: to support all HPC platforms, compilers, and runtime systems
- Provides portable instrumentation, measurement, analysis

**TAU Architecture**

**Instrumentation**
- **Source**
  - C, C++, Fortran
  - Python, UPC, Java
  - Robust parsers (PDT)
- **Wrapping**
  - Interposition (PMPI)
  - Wrapper generation
- **Linking**
  - Static, dynamic
  - Preloading
- **Executable**
  - Dynamic (Dyninst)
  - Binary (Dyninst, MAQAO)

**Measurement**
- **Events**
  - static/dynamic
  - routine, basic block, loop
  - threading, communication
  - heterogeneous
- **Profiling**
  - flat, callpath, phase, parameter, snapshot
  - probe, sampling, hybrid

**Analysis**
- **Profiles**
  - ParaProf parallel profile analyzer / visualizer
  - PerfDMF parallel profile database
  - PerfExplorer parallel profile data mining
- **Tracing**
  - TAU trace translation
    - OTF, SLOG-2
  - Trace analysis / visualizer
    - Vampir, Jumpshot
- **Online**
  - event unification
  - statistics calculation
How does TAU work?

- Instrumentation: Adds probes to perform measurements
  - Source code instrumentation using pre-processors and compiler scripts
  - Wrapping external libraries (I/O, MPI, Memory, CUDA, OpenCL, pthread)
  - Rewriting the binary executable

- Measurement: Profiling or Tracing using wallclock time or hardware counters
  - Direct instrumentation (Interval events measure exclusive or inclusive duration)
  - Indirect instrumentation (Sampling measures statement level contribution)
How does TAU work, cont.

- Throttling and runtime control of low-level events that execute frequently
- Per-thread storage of performance data
- Interface with external packages (e.g. PAPI performance counter library)

Analysis: Visualization of profiles and traces
- 2D, 3D visualization of profile data in paraprof, perfexplorer tools
- Trace conversion & display in external visualizers (Vampir, Jumpshot, ParaVer)
Using TAU: A brief Introduction to instrumentation

- TAU supports several measurement and thread options
  - Each measurement configuration of TAU corresponds to a unique stub makefile and library that is generated when you configure it
- To instrument source code automatically using PDT
  
  ```
  $ export TAU_MAKEFILE=/home/sshende/apps/tau/x86_64/lib/Makefile.tau-icpc-mpi-pdt
  $ export PATH=/home/sshende/apps/tau/x86_64/bin:$PATH
  ```
- Use tau_f90.sh, tau_cxx.sh, tau_upc.sh, or tau_cc.sh:
  $ mpif90 foo.f90 ...changes to... $ tau_f90.sh foo.f90
- Set variables, execute and analyze performance data:
  $ pprof or paraprof
Using TAU: A brief introduction to tau_exec

☐ TAU can collect data without instrumentation

☐ Instead of Makefile.tau-…, use tau_exec –T
  ○ mpirun –np 4 tau_exec –T icpc,pdt,mpi ./a.out
  ○ tau_exec –T icpc,papi,openmp ./a.out
  ○ tau_exec –T icpc,openmp –ebs ./a.out
  ○ tau_exec –T icpc,pthread –memory_debug ./a.out
  ○ tau_exec –T icpc,mpi,openmp –io ./a.out
  ○ tau_exec –T icpc,openmp –loadlib=<lib> ./a.out
  ○ …

☐ -T tags are used in other TAU tools:
  ○ tau_run, tau_rewrite, tau_exec, tau_gen_wrapper
Interval and Atomic Events in TAU

**Interval events**
- e.g., routines (start/stop) show duration

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

% export TAU_CALLPATH_DEPTH=0
% export TAU_TRACK_HEAP=1
Atomic Events, Context Events

% export TAU_CALLPATH_DEPTH=1
% export TAU_TRACK_HEAP=1

Control depth of executing context shown in profiles
Instrumentation Options in TAU

- Source Code Instrumentation
  - Manual instrumentation
  - Automatic instrumentation using pre-processor based on static analysis of source code (PDT), creating an instrumented copy
  - Compiler generates instrumented object code

- Library Level Instrumentation
  - Wrapper libraries for standard MPI libraries using PMPI interface
  - Wrapping external libraries where source is not available

- Runtime pre-loading and interception of library calls

- Binary Code instrumentation
  - Rewrite the binary, runtime instrumentation

- Virtual Machine, Interpreter, OS level instrumentation
Source Instrumentation Using PDT

- **TAU source analyzer**
- **Application source**
- **Parsed program**
- **tau_instrumentor**
- **Instrumentation specification file**
- **Instrumented copy of source**
TAU Instrumentor for generic instrumentation

- “Improving the flexibility of the TAU instrumentor”
- Markus Geimer, JSC
- Cooperative effort during summer of 2008
- Make TAU instrumentor generic
“file” construct

file=”<file>” line=<line> code=”<code>” [lang=”<lang>”]

- Inserts the given <code> at <line> in <file>
  - *Files can be specified using wildcards*

- New, optional “lang” token restricts construct to the given language(s)
  - *Comma-separated list of languages*
  - *Recognized: c, c++, fortran*
“entry” construct

```
entry [file="<file>"] [routine="<r>"] code="<code>"
[lang="<lang>"]
```

- Inserts the given <code> at the beginning of the code section of routine <r> in <file>
  - *Files can be specified using wildcards*
  - *Routines can be specified using wildcards*

☑ Specifying a routine name is now optional (default: “#”)
☑ Construct is subject to file/routine filter rules
☑ New, optional “lang” token restricts construct to the given language(s)
  - *Comma-separated list of languages*
  - *Recognized: c, c++, fortran*
“exit” construct

exit [file="<file>"] [routine="<r>"] code="<code>
[lang="<lang>"]

- Inserts the given <code> at each exit of routine <r> in <file>
  - Files can be specified using wildcards
  - Routines can be specified using wildcards

☑ Specifying a routine is now optional (default: “#”)
☑ Construct is subject to file/routine filter rules
☑ New, optional “lang” token restricts construct to the given language(s)
  - Comma-separated list of languages
  - Recognized: c, c++, fortran
“init” construct

init code="<code>" [lang="<lang>"]

☑ Inserts the given <code> at the beginning of main() for C/C++
or a program in Fortran

☑ Construct is subject to file/routine filter rules

☑ Optional “lang” token restricts construct to the given language(s)
“decl” construct

```
decl [file="<file>"] [routine="<r>"] code="<code>
[lang="<lang>"]
```

☑ Adds the given <code> to the declaration section of routine <r> in <file>

- Files can be specified using wildcards
- Routines can be specified using wildcards

☑ Construct is subject to file/routine filter rules

☑ Optional “lang” token restricts construct to the given language(s)
To use “instrumentor knowledge” in the inserted code snippets, some substitutions are performed:

- **file, decl, init, entry, exit**
  - @FILE@ → Filename
  - @LINE@ → Insertion line
  - @COL@ → Insertion column

- **decl, init, entry, exit**
  - @ROUTINE@ → Routine name
  - @BEGIN_LINE@ → Line of routine begin
  - @BEGIN_COL@ → Column of routine begin
  - @END_LINE@ → Line of routine end
  - @END_COL@ → Column of routine end

- **init**
  - @ARGV@ → First argument to main()
Putting it all together: an example

BEGIN_INSTRUMENT_SECTION
    file="*"  line=1  code="#include <stdio.h>"

    init  code="printf("Argument count: %d\n", @ARGC@);"
    decl  code="static int ex_count=0;"
    entry code="printf("Entered @ROUTINE@ [%d]\n", ex_count++);"
    exit  code="printf("Leaving @ROUTINE@ at line @LINE@\n");"

END_INSTRUMENT_SECTION
int foo(int value) {
  if (value > 1)
    return 1;
  else
    return 0;
}

int main(int count, char** args)
{
  int i;

  for (i=0; i<4; ++i)
    foo(i);

  return 0;
}

#include <stdio.h>
int foo(int value) {
  int tau_ret_val;
  static int ex_count=0;

  printf(“Entered int foo(int) C”
         “ [%d]\n”, ex_count++);
  {
    if (value > 1) {
      tau_ret_val=1;
      printf(“Leaving int foo(int) C”
              “ at line 3\n”);
      return (tau_ret_val);
    } else {
      tau_ret_val=0;
      printf(“Leaving int foo(int) C”
              “ at line 5\n”);
      return (tau_ret_val);
    }
  }
  printf(“Leaving int foo(int) C”
          “ at line 6\n”);
}
...
Example output

$ ./example 2 3 4
Argument count: 4
Entered int main(int, char **) C [0]
Entered int foo(int) C [0]
Leaving int foo(int) C at line 5
Entered int foo(int) C [1]
Leaving int foo(int) C at line 5
Entered int foo(int) C [2]
Leaving int foo(int) C at line 3
Entered int foo(int) C [3]
Leaving int foo(int) C at line 3
Leaving int main(int char **) C at line 15
$
Three Techniques for Wrapping External Libraries

- **Pre-processor based substitution by re-defining a call**
  - Tool defined header file with same name `<unistd.h>` takes precedence
  - Header redefines a routine as a different routine using macros
  - Substitution: `read()` substituted by preprocessor as `tau_read()` at callsite

- **Preloading a library at runtime**
  - Library preloaded (LD_PRELOAD env var in Linux) in the address space of executing application intercepts calls from a given library
  - Tool’s wrapper library defines `read()`, gets address of global `read()` symbol (`dlsym`), internally calls timing calls around call to global `read`

- **Linker based substitution**
  - Wrapper library defines `__wrap_read` which calls `__real_read` and linker is passed `-Wl,-wrap,read` to substitute all references to `read` from application’s object code with the `__wrap_read` defined by the tool
Issues: Preprocessor based substitution

- Pre-processor based substitution by re-defining a call
  - Compiler replaces read() with tau_read() in the body of the source code

- Advantages:
  - Simple to instrument
  - Preprocessor based replacement
  - A header file redefines the calls
  - No special linker or runtime flags required

- Disadvantages
  - Only works for C & C++ for replacing calls in the body of the code.
  - Incomplete instrumentation: fails to capture calls in uninstrumented libraries (e.g., libhdf5.a)
Issues: Linker based substitution

- Linker based substitution

  - Wrapper library defines \_\_wrap\_read which calls \_\_real\_read and linker is passed -Wl,-wrap, read

- Advantages

  - Tool can intercept all references to a given call
  - Works with static as well as dynamic executables
  - No need to recompile the application source code, just re-link the application objects and libraries with the tool wrapper library

- Disadvantages

  - Wrapping an entire library can lengthen the linker command line with multiple -Wl,-wrap,<func> arguments. It is better to store these arguments in a file and pass the file to the linker
  - Approach does not work with un-instrumented binaries
Preloading a wrapper library at runtime

- Tool defines read(), gets address of global read() symbol (dlsym), internally calls timing calls around call to global read
- tau_exec tool uses this mechanism to intercept library calls

- Advantages
  - No need to re-compile or re-link the application source code
  - Drop-in replacement library implemented using LD_PRELOAD environment variable under Linux, Cray CNL, IBM BG/P CNK, Solaris...

- Disadvantages
  - Only works with dynamic executables. Default compilation mode under Cray XE6 and IBM BG/P is to use static executables
  - Not all operating systems support preloading of dynamic shared objects (DSOs)
tau_gen_wrapper

- Automates creation of wrapper libraries using TAU

- Input:
  - header file (foo.h), library to be wrapped (/path/to/libfoo.a)
  - technique for wrapping
    - Preprocessor based redefinition (-d)
    - Runtime preloading (-r)
    - Linker based substitution (-w: default)
  - Optional selective instrumentation file (-f select)
    - Exclude list of routines, or
    - Include list of routines

- Output:
  - wrapper library, optional link_options.tau file (-w), pass –optTauWrapFile=<file> in TAU_OPTIONS environment variable
tau_wrap

- TAU source analyzer
- Parsed program
- Instrumentation specification file
- tau_wrap
- Application source
- Instrumented source
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_spec_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

**NODE 0;CONTEXT 0;THREAD 0:**

<table>
<thead>
<tr>
<th>%Time</th>
<th>Exclusive msec</th>
<th>Inclusive total msec</th>
<th>#Call</th>
<th>#Subrs</th>
<th>Inclusive Name usec/call</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0</td>
<td>0.057</td>
<td>1</td>
<td>1</td>
<td>13</td>
<td>1236 .TAU Application</td>
</tr>
<tr>
<td>70.8</td>
<td>0.875</td>
<td>0.875</td>
<td>1</td>
<td>0</td>
<td>875 hid_t H5Fcreate()</td>
</tr>
<tr>
<td>9.7</td>
<td>0.12</td>
<td>0.12</td>
<td>1</td>
<td>0</td>
<td>120 herr_t H5Fclose()</td>
</tr>
<tr>
<td>6.0</td>
<td>0.074</td>
<td>0.074</td>
<td>1</td>
<td>0</td>
<td>74 hid_t H5Dcreate()</td>
</tr>
<tr>
<td>3.1</td>
<td>0.038</td>
<td>0.038</td>
<td>1</td>
<td>0</td>
<td>38 herr_t H5Dwrite()</td>
</tr>
<tr>
<td>2.6</td>
<td>0.032</td>
<td>0.032</td>
<td>1</td>
<td>0</td>
<td>32 herr_t H5Dclose()</td>
</tr>
<tr>
<td>2.1</td>
<td>0.026</td>
<td>0.026</td>
<td>1</td>
<td>0</td>
<td>26 herr_t H5check_version()</td>
</tr>
</tbody>
</table>
TAU Binary Instrumentation

- DynInst, MACAO, Pebil (in progress)
- Support for both static and dynamic executables
- Specify the list of routines to instrument/exclude from instrumentation
- Specify the TAU measurement library to be injected

Simplify the usage of TAU:

- To instrument:
  
  $$\text{	exttt{tau\_run} --T [tags] a.out --o a.inst}$$

- To perform measurements, execute the application:
  
  $$\text{	exttt{mpiexec./a.inst}}$$

- To analyze the data
  
  $$\text{	exttt{paraprof}}$$
TAU Sampling

- Periodic sampling support
- Libunwind, StackWalkerAPI support
- Variable sampling frequency
- SIGPROF / ITIMER_PROF for multithreaded signaling
- PAPI counter overflow timer support
TAU Metadata Collection

- Automatically collected from `/proc/cpuinfo`, `/proc/meminfo`, BGP, BGQ personality information, Cray Topology information
- Context-sensitive
- Structured

![Metadata for n,c,t 9,0,0](image.png)
GPU: 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
Profiling GPGPU Executions

- GPGPU compilers (e.g., CAPS hmpp and PGI) can now automatically generate GPGPU code using manual annotation of loop-level constructs and routines (hmpp)
- The loops (and routines for HMPP) are transferred automatically to the GPGPU
- TAU intercepts the runtime library routines and examines the arguments
- Shows events as seen from the host
- Profiles and traces GPGPU execution
Host (CPU) - GPU Scenarios

**Single GPU**

1. **Host (CPU)**
   - Open device
   - Move data
   - Launch kernel(s)
   - Wait
   - Move data

2. **GPU**
   - Run kernel(s)

   implemented as asynchronous calls

**Multi-stream**

1. **Host (CPU)**
   - Open device
   - Move data
   - Launch kernel(s)
   - Wait
   - Move data
   - Open device
   - Move data
   - Launch kernel(s)
   - Wait
   - Move data

2. **GPU**
   - Stream 1
   - Run kernel(s)
   - Stream 2
   - Run kernel(s)

**Multi-CPU, Multi-GPU**

1. **Thread (CPU 1)**
   - Open device
   - Move data
   - Launch kernel(s)
   - Wait
   - Move data

2. **GPU**
   - GPU 1
   - Run kernel(s)
   - ...
   - GPU k
   - Run kernel(s)

3. **Thread (CPU k)**
   - Open device
   - Move data
   - Launch kernel(s)
   - Wait
   - Move data
Host-GPU Measurement – Callback Method

- GPU driver libraries provide callbacks for certain routines and captures measurements
- Measurement tool registers the callbacks and processes performance data
- Application code is not modified
NVIDIA has developed 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+.
GPU Performance Tool Interoperability

Diagram showing the integration of CUDA, OpenCL, and various profiling tools such as TAU, PAPI, and VampirTrace through the use of CUPTI.
Profiling PGI Accelerator Primatives

- 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, and routines.
TAU Profile, Trace Analysis

- Profiles:
  - ParaProf
  - PerfExplorer
  - CUBE

- Traces:
  - Vampir (OTF, VTF)
  - Jumpshot (SLOG2)
  - Paraver (PRV)
  - Expert
  - ProfileGen
  - Vampir Server
ParaProf Profile Analysis & Visualization

Performance Data
- Profiles
  - TAU, mpiP, ompP, HPMToolkit, Cube, HPC Toolkit, Gprof, Dynaprop, PSRun
- Runtime Data Collection
  - Supermon, MRNet
- DBMS
  - PostgreSQL, MySQL, Oracle, DB2, Derby

PerfDMF
- Parsers and Importers
- Basic Analysis + Derived Data
- Internal Representation
- Profile Data

ParaProf
- Call Graphs
- Histograms
- Call Trees
- Bar Charts
- Comparative Displays
- Text Displays
- Vis Package
  - JOGL
  - 3D Displays

Scripting Interface
- Jython

XPRESS Workshop on System Performance, Feb 6th-8th 2013 @ LSU
ParaProf: NAS BT MPI Flat Profile

Application routine names reflect phase semantics

How is \texttt{MPI\_Wait()} distributed relative to solver direction?

<table>
<thead>
<tr>
<th>Metric Name: Time</th>
<th>Value Type: exclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>std. dev.</td>
<td></td>
</tr>
<tr>
<td>mean</td>
<td></td>
</tr>
<tr>
<td>n,c,t 0,0,0</td>
<td></td>
</tr>
<tr>
<td>n,c,t 1,0,0</td>
<td></td>
</tr>
<tr>
<td>n,c,t 2,0,0</td>
<td></td>
</tr>
<tr>
<td>n,c,t 3,0,0</td>
<td></td>
</tr>
<tr>
<td>n,c,t 4,0,0</td>
<td></td>
</tr>
<tr>
<td>n,c,t 5,0,0</td>
<td></td>
</tr>
<tr>
<td>n,c,t 6,0,0</td>
<td></td>
</tr>
<tr>
<td>n,c,t 7,0,0</td>
<td></td>
</tr>
<tr>
<td>n,c,t 8,0,0</td>
<td></td>
</tr>
</tbody>
</table>

Notes:
- Y\_SOLVE\_CELL
- LHSZ
- LHSY
- Z\_SOLVE\_CELL
- X\_SOLVE\_CELL
- MPI\_Wait()
- LHSX
- MPI\_Waitall()
ParaProf: NAS BT MPI Phase Profile

Main phase shows nested phases and immediate events
ParaProf: 3D Display
ParaProf: 3D Topology
ParaProf: Snapshot profiles

- Profile snapshots are parallel profiles recorded at runtime
- Shows performance profile dynamics (all types allowed)
ParaProf: Profile Snapshot Views

Percentage breakdown    Only show main loop

[Images of ParaProf profiles showing snapshot breakdowns]
ParaProf: Snapshot Replay

All windows dynamically update
ParaProf: Callgraph Visualizations

Call Graph for n,c,t; 0,0,0 – tmp/private/

- main() (calls f1, f3)
- f5() (sleeps 5 sec)
- f1() (sleeps 1 sec, calls f2, f4)
- f4() (sleeps 4 sec, calls f2)
- f2() (sleeps 2 sec, calls f3)
- f3() (sleeps 3 sec)

TAU: ParaProf: Statistics for: node 0 – /Users/khuck/src/tau2/examples/userevent

<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() (calls f1, f3)</td>
<td>0</td>
<td>20.014</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>f1() (sleeps 1 sec, calls f2, f4)</td>
<td>1.007</td>
<td>20.014</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>f2() (sleeps 2 sec, calls f3)</td>
<td>2.001</td>
<td>5.003</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>f3() (sleeps 3 sec)</td>
<td>3.001</td>
<td>3.001</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>f4() (sleeps 4 sec, calls f2)</td>
<td>4.001</td>
<td>9.004</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>f5() (sleeps 5 sec)</td>
<td>5.001</td>
<td>5.001</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
ParaProf: Communication Matrix – 2D, 3D
PerfExplorer: Parametric Study Support
PerfExplorer: Phase Profiling of HW Counters

- GTC particle-in-cell simulation of fusion turbulence
- Phases assigned to iterations
- Poor temporal locality for one important data
- Automatically generated by PE2 python script

Graphs showing:
- Increasing phase execution time
- Decreasing flops rate
- Declining cache performance
-0.995 indicates strong, negative relationship. As `CALC_CUT_BLOCK_CONTRIBUTIONS()` increases in execution time, `MPI_Barrier()` decreases.
PerfExplorer Cluster Analysis
PerfExplorer Cluster Analysis

- Four significant events automatically selected
- Clusters and correlations are visible
PerfExplorer: Regression Visualization
Final Items of Note

- **KTAU**
  - Instrumentation of Linux kernel space
    - System calls, exceptions, interrupts, scheduling, signals
  - Kernel-wide, process-specific perspectives
  - `/proc/ktau` filesystem for interaction between kernel space and user space
  - User space measurements given kernel space context

- **OpenMP**
  - Collector API support (OpenUH)
  - GOMP Wrapper
Support Acknowledgements

- US Department of Energy (DOE)
  - Office of Science contracts
  - SciDAC, LBL contracts
  - LLNL-LANL-SNL ASC/NNSA contract
  - Battelle, PNNL contract
  - ANL, ORNL contract
- Department of Defense (DoD)
  - PETTT, HPCMP
- National Science Foundation (NSF)
  - Glassbox, SI-2

Partners:
- University of Oregon
- ParaTools, Inc.
- University of Tennessee, Knoxville
- T.U. Dresden, GWT
- Juelich Supercomputing Center