Overview

- **TAU**
  - Comprehensive performance analysis framework and toolset
  - Can produce profiles and event traces
  - Supports a large variety of programming languages / paradigms and hardware platforms

- **KOJAK**
  - Automatically localizes, classifies, and ranks performance problems in event traces of MPI, OpenMP, SHMEM, or hybrid parallel Fortran, C, C++ programs

- **Vampir**
  - Well known, robust, and easy-to-use event trace visualization tool
KOJAK ↔ TAU ↔ VAMPIR

KOJAK
- OTF
TAU trace
EPILOG trace
KOJAK
TAU - TRACE
TAU - EPILOG
KOJAK
- PROFILE

TAU trace
EXPERT Analyzer
CUBE profile
CUBE Presenter

OTF / VTF3 trace
VAMPIR
VampirTrace <= 4.0

gprof / mpiP profile
PerfDMF
PARAPROF

covered now
covered later

Program Instrumentation and Measurement with KOJAK

Bernd Mohr
Forschungszentrum Jülich
John von Neumann Institute for Computing
Jülich, Germany
b.mohr@fz-juelich.de
KOJAK Instrumentation and Measurement

1. Instrument user application with EPILOG tracing library calls
   - User functions and regions:
     - Preferred: Manually using POMP directives
     - Also: Manually using EPILOG tracing API functions
     - Also: Automatically by Compiler
   - MPI calls:
     - Automatically by PMPI Wrapper Library (MPI 1.2 + MPI 2 RMA)
   - OpenMP
     - Automatically by OPARI source code instrumentor
   - Alternative: Configure TAU to generate EPILOG traces

2. Running instrumented executable will produce trace file named prefix.elg
   - Environment variables allow to control the measurement

KOJAK Manual User Region Instrumentation I

- Insert INST INIT directive as the first executable line of main
  - Fortran:
    ```fortran
    !$POMP INST INIT
    ```
  - C/C++:
    ```c
    #pragma pomp inst init
    ```

- Use INST BEGIN/END directive to mark any user-defined region
  - At least the main program/function has to be instrumented
  - All but last exit have to be instrumented by INST ALTEND
  - Fortran:
    ```fortran
    !$POMP INST BEGIN(name)
    ...  
    ![POMP INST ALTEND(name)]
    ...  
    !$POMP INST END(name)
    ```
  - C/C++:
    ```c
    #pragma pomp inst begin(name)
    ...  
    [#pragma pomp inst altend(name)]
    ...  
    #pragma pomp inst end(name)

- Add kinst-pomp to compile + link commands in Makefile and recompile
  - Fortran:
    ```
    FC = kinst-pomp [-otf] f90
    ```
  - C/C++:
    ```
    CC = kinst-pomp [-otf] cc
    ```
KOJAK Manual User Region Instrumentation II

- Which functions and user regions should be instrumented?
  - Typically upper levels in call tree plus deeper in iteration phase
  - Use (gprof) profile (especially if you don’t know the code)
    - To determine most important regions
    - To avoid frequently called (small) regions

- Advantages
  - Directives get ignored during “normal” compilation
  - Allows users to
    - Instrument arbitrary code regions
    - Set the focus of the analysis
    - Control optimal amount of instrumentation
  - Makes comparisons of different implementations easier

KOJAK Manual User Region Instrumentation III

- Alternatively, use EPILOG tracing API functions to mark regions

  Fortran:
  ```
  #include 'elg_user.inc'
  call ELG_USER_START('name');
  ...
  call ELG_USER_END('name');
  ```

  C:
  ```
  #include "elg_user.h"
  ELG_USER_START("name");
  ...
  ELG_USER_END("name");
  ```

  C++:
  ```
  #include "elg_user.h"
  {
    ELG_TRACER("name");
    ...
  }
  ```

- Note, Fortran source files instrumented this way have to be (cpp) preprocessed.
- Necessary Makefile changes:
  ```
  CFLAGS = ... -DEPILOG `kconfig --cflags [--64]`
  LDFLAGS = ... `kconfig --libs [--for] [--omp] --hybrid] [--64] [--otf]`
  ```

- Compile with "-DEPILOG" otherwise the "ELG_*" calls are ignored.
- Does not instrument OpenMP automatically! ⇒ use OPARI manually
KOJAK Automatic User Function Instrumentation

- Uses (sometimes hidden and undocumented) compiler flag to instrument every function in user application
  - Works for PGI, GNU, Intel, IBM, NEC, Sun f90, Hitachi compilers
- Advantage
  - Automatic ⇒ simple procedure
- Disadvantage
  - Instruments too much ⇒ often produces too large trace files

- Add kinst to compile + link commands in Makefile and recompile

```plaintext
Fortran:
FC = kinst [-otf] f90

C/C++:
CC = kinst [-otf] cc
```

- Note, kinst and kinst-pomp also automatically instrument OpenMP constructs and MPI functions if necessary

KOJAK Measurement I

- Running instrumented executable will produce trace file named `prefix.elg`
- Environment variables allow to control the measurement

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELG_METRICS</td>
<td>Specify counter metrics to be recorded with trace events as a colon-separated list of names (e.g., PAPI preset events)</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>ELG_FILE_PREFIX</td>
<td>Prefix used for trace filenames</td>
<td>&quot;a&quot;</td>
</tr>
<tr>
<td>ELG_VERBOSE</td>
<td>Print EPILOG related control information during measurement?</td>
<td>no</td>
</tr>
<tr>
<td>ELG_BUFFER_SIZE</td>
<td>Internal event trace buffer in bytes</td>
<td>10000000</td>
</tr>
</tbody>
</table>

- Make sure the environment variables have the same value for all processes of your application on ALL nodes of your machine.
Additional, advanced control environment variables

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELG_PFORM_GDIR</td>
<td>Name of global, cluster-wide directory to store final trace file</td>
<td>Platform specific, typically &quot;.&quot;</td>
</tr>
<tr>
<td>ELG_PFORM_LDIR</td>
<td>Name of node-local directory that can be used to store temporary trace files</td>
<td>Platform specific, typically &quot;/tmp&quot;</td>
</tr>
<tr>
<td>ELG_METRICS_SPEC</td>
<td>Name of file specifying metrics, groups and relations</td>
<td>KOJAK install directory &quot;doc/METRICS.SPEC&quot;</td>
</tr>
</tbody>
</table>

Value for the variables $ELG_FILE_PREFIX$, $ELG_PFORM_GDIR$ and $ELG_PFORM_LDIR$ can contain strings of the form $XYZ$ or ${XYZ}$

Evaluation of these environment variables is done during measurement of the instrumented application.
**TAU Instrumentation and Measurement**

- **Architecture**
- **Instrumentation**
  - Approaches and mechanisms
- **Measurement**
  - Approach and mechanisms
- **Using TAU**
  - Install TAU
  - Instrument application
  - Create performance experiments
  - Execute application
  - Analyze performance (in the afternoon)
**TAU Instrumentation Approach**

- Support for standard program events
  - Routines, classes and templates
  - Statement-level blocks
  - Interval events (begin/end events)
- Support for user-defined events
  - "User-defined timers"
  - Atomic events (e.g., size of memory allocated/freed)
  - Selection of event statistics
- Support for event groups (aggregation, selection)
- Definition of "semantic" entities for mapping
- Instrumentation optimization to eliminate lightweight routines
- Target TAU measurement API

**TAU Instrumentation Mechanisms**

- Source code
  - Manual (TAU API, TAU component API)
  - Automatic
    - C, C++, F77/90/95 (Program Database Toolkit (PDT))
    - OpenMP (directive rewriting (Opari), POMP specification)
- Object code
  - Pre-instrumented libraries (e.g., MPI using PMPI)
  - Statically-linked and dynamically-linked (e.g., LD_PRELOAD)
- Executable code
  - Dynamic instrumentation (pre-execution) (e.g., DynInstAPI)
  - Virtual machine instrumentation
  - Python interpreter-based instrumentation at runtime
- Proxy components
Multi-Level Instrumentation and Mapping

- Multiple instrumentation interfaces
- Information sharing
  - Between interfaces
- Event selection
  - Within/between levels
- Mapping
  - Associate performance data with high-level semantic abstractions
- Instrumentation targets measurement API with support for mapping

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 Open Trace Format (OTF)
  - Trace streams and hierarchical trace merging
- Robust timing and hardware performance support
- Multiple counters (hardware, user-defined, system)
- Performance measurement for CCA component software
TAU Measurement Mechanisms

- Parallel profiling
  - Function-level, block-level, statement-level
  - Supports user-defined events and mapping events
  - TAU parallel profile stored (dumped) during execution
  - Support for flat, callgraph/callpath, phase profiling
  - Support for memory profiling
- Tracing
  - All profile-level events
  - Inter-process communication events
  - Inclusion of multiple counter data in traced events

Types of Parallel Performance Profiling

- Flat profiles
  - Metric (e.g., time) spent in an event (e.g., MPI_Send, foo, …)
  - 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_LENGTH environment variable (paths ≤ this value)
- Phase profiles
  - Flat profiles within a phase (nested phases are allowed)
  - Default "main" phase has all phases and routines invoked outside
  - Supports static or dynamic (per-iteration) phases
  - Example: "IO => MPI_Send" is time spent in MPI_Send in IO phase
Using TAU

- Install TAU
  - Configuration
  - Measurement library creation
- Instrument application
  - Manual or automatic source instrumentation
  - Instrumented library (e.g., MPI - wrapper interposition library)
  - Binary instrumentation
- Create performance experiments
  - Integrate with application build environment
  - Set measurement variables
- Execute application
- Analyze performance

TAU Measurement System Configuration

- configure [OPTIONS]
  - {-c++=<CC>, -cc=<cc>} Specify C++ and C compilers
  - {-pthread, -sproc} Use pthread or SGI sproc threads
  - {-openmp} Use OpenMP threads
  - {-jdk=<dir>} Specify Java instrumentation (JDK)
  - {-opari=<dir>} Specify location of Opari OpenMP tool
  - {-papi=<dir>} Specify location of PAPI
  - {-pdt=<dir>} Specify location of PDT
  - {-dyninst=<dir>} Specify location of DynInst Package
  - {-mpi[inc/lib]=<dir>} Specify MPI library instrumentation
  - {-shmem[inc/lib]=<dir>} Specify PSHMEM library instrumentation
  - {-python[inc/lib]=<dir>} Specify Python instrumentation
  - {-epilog=<dir>} Specify location of EPILOG
  - {-slog2[=<dir>] Specify location of SLOG2/Jumpshot
  - {-vtf=<dir>} Specify location of VTF3 trace package
  - {-arch=<architecture>} Specify architecture explicitly (bgl, xt3, ibm64, ...)

© 2006 Allen Malony, University of Oregon, and Bernd Mohr, Forschungszentrum Jülich
TAU Measurement System Configuration

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

Configuration Example

- ./configure -pdt=/app/comenv/ALTIX/pkgs/pdtoolkit-3.6 -mpi
  Configure using PDT and MPI
- ./configure -papi=/usr/local/packages/papi
  -pdt=/usr/local/pdtoolkit-3.6 -mpiinc=/usr/local/include
  -mpilib=/usr/local/lib
  -MULTIPLECOUNTERS -c++=icpc -cc=icc -fortran=intel
  Use PAPI counters with C/C++/F90 automatic instrumentation
  Also instrument the MPI library
  Use Intel compilers.
- Build multiple measurement libraries for different purposes
- Each configuration creates a unique stub makefiles for options used
  - <arch>/lib/Makefile.tau<options>
  - /app/comenv/ALTIX/tau/ia64/lib/Makefile.tau-icpc-mpi-pdt
  - /app/comenv/ALTIX/tau/ia64/lib/Makefile.tau-icpc-mpi-pdt-trace
TAU_Conf: A GUI for Installing TAU

Using TAU

- Install TAU
  - Configuration
  - Measurement library creation
- Instrument application
  - Manual or automatic source instrumentation
  - Instrumented library (e.g., MPI - wrapper interposition library)
  - Binary instrumentation
- Create performance experiments
  - Integrate with application build environment
  - Set measurement variables
- Execute application
- Analyze performance
**Measurement API for Manual Instrumentation**

- Initialization and runtime configuration
  - TAU_PROFILE_INIT(argc, argv);
  - TAU_PROFILE_SET_NODE(myNode);
  - TAU_PROFILE_SET_CONTEXT(myContext);
  - TAU_PROFILE_EXIT(message);
  - TAU_REGISTER_THREAD();
- Function and class methods for C++ only:
  - TAU_PROFILE(name, type, group);
  - TAU_PROFILE( name, type, group);
- Template
  - TAU_TYPE_STRING(variable, type);
  - TAU_PROFILE(name, type, group);
  - CT (variable);
- User-defined timing
  - TAU_PROFILE_TIMER(timer, name, type, group);
  - TAU_PROFILE_START(timer);
  - TAU_PROFILE_STOP(timer);

**Measurement API for Manual Instrumentation**

- Defining application phases
  - TAU_PHASE_CREATE_STATIC( var, name, type, group);
  - TAU_PHASE_CREATE_DYNAMIC( var, name, type, group);
  - TAU_PHASE_START(var);
  - TAU_PHASE_STOP (var);
- User-defined events
  - TAU_REGISTER_EVENT(variable, event_name);
    - TAU_EVENT(variable, value);
    - TAU_PROFILE_STMT(statement);
- Heap Memory Tracking:
  - TAU_TRACK_MEMORY();
  - TAU.Track_MEMORY_HEADROOM();
  - TAU_SET_INTERRUPT_INTERVAL(seconds);
  - TAU_DISABLE_TRACKING_MEMORY[_HEADROOM]();
  - TAU_ENABLE_TRACKING_MEMORY[_HEADROOM]();
#include <TAU.h>
int main(int argc, char **argv)
{
    TAU_PROFILE("int main(int, char **)", " ", TAU_DEFAULT);
    TAU_PROFILE_INIT(argc, argv);
    TAU_PROFILE_SET_NODE(0); /* for sequential programs */
    foo();
    return 0;
}
int foo(void)
{
    TAU_PROFILE("int foo(void)", " ", TAU_DEFAULT); // measures entire foo()
    TAU_PROFILE_TIMER(t, "foo(): for loop", ":[23:45 file.cpp]", TAU_USER);
    TAU_PROFILE_START(t);
    for(int i = 0; i < N ; i++)
    {
        work(i);
    }
    TAU_PROFILE_STOP(t);
    // other statements in foo ...
}

PROGRAM SUM_OF_CUBES
integer profiler(2)
save profiler
INTEGER :: H, T, U
call TAU_PROFILE_INIT()
call TAU_PROFILE_TIMER(profiler, 'PROGRAM SUM_OF_CUBES')
call TAU_PROFILE_START(profiler)
call TAU_PROFILE_SET_NODE(0)
! This program prints all 3-digit numbers that equal the sum of the cubes of
! their digits.
DO H = 1, 9
  DO T = 0, 9
    DO U = 0, 9
      IF (100*H + 10*T + U == H**3 + T**3 + U**3) THEN
        PRINT "(3I1)", H, T, U
      ENDIF
    END DO
  END DO
END DO
call TAU_PROFILE_STOP(profiler)
END PROGRAM SUM_OF_CUBES
Using Program Database Toolkit (PDT)

1. Parse the Program to create foo.pdb:
   % cxxparse foo.cpp -I/usr/local/mydir -DMYFLAGS ...
   or
   % cparsse foo.c -I/usr/local/mydir -DMYFLAGS ...
   or
   % f95parse foo.f90 -I/usr/local/mydir ...
   % f95parse *.f -omerged.pdb -I/usr/local/mydir -R free
2. Instrument the program:
   % tau_instrumentor foo.pdb foo.f90 -o foo.inst.f90
   -f select.tau
3. Compile the instrumented program:
   % ifort foo.inst.f90 -c -I/usr/local/mpi/include
   -o foo.o
**Optimization of Program Instrumentation**

- Selective instrumentation file to filter events
  
  ```
  % tau_instrumentor [options] -f <file>
  or
  % setenv TAU_OPTIONS 'optTauSelectFile=tau.txt'
  ```

- Eliminate of instrumentation in lightweight routines
  
  - Throttling of events at runtime:
    ```
    % setenv TAU_THROTTLE 1
    ```
    - Turns off instrumentation in routines that execute over 10000 times (TAU_THROTTLE_NUMCALLS) and take less than 10 microseconds of inclusive time per call (TAU_THROTTLE_PERCALL)

- Compensation of local instrumentation overhead
  ```
  % configure -COMPENSATE
  ```

**Instrumentation Specification File**

```
% tau_instrumentor

For selective instrumentation, use -f option

% tau_instrumentor foo.pdb foo.cpp -o foo.inst.cpp -f selective.dat
% cat selective.dat

# Selective instrumentation: Specify an exclude/include list of routines/files.
BEGIN_EXCLUDE_LIST
void quicksort(int *, int, int)
void sort_5elements(int *)
void interchange(int *, int *)
END_EXCLUDE_LIST

BEGIN_FILE_INCLUDE_LIST
Main.cpp
Foo?.c
*.[C]
END_FILE_INCLUDE_LIST

# Instruments routines in Main.cpp, Foo?.c and *.c files only
# Use BEGIN_[FILE]_INCLUDE_LIST with END_[FILE]_INCLUDE_LIST
```
TAU's MPI Wrapper Interposition Library

- Uses standard MPI Profiling Interface
  - Provides name shifted interface
    - MPI_Send = PMPI_Send
  - Weak bindings
- Interpose TAU's MPI wrapper library between MPI and TAU
  - -lmpi replaced by -lTauMpi -lpmpi -lmpi
- No change to the source code!
  - Just re-link the application to generate performance data
  - setenv TAU_MAKEFILE <dir>/<arch>/lib/Makefile.tau-mpi -[options]
  - Use tau_cxx.sh, tau_f90.sh and tau_cc.sh as compilers

Runtime MPI Shared Library Instrumentation

- We can now interpose the MPI wrapper library for applications that have already been compiled
  - No re-compilation or re-linking necessary!
- Uses LD_PRELOAD for Linux
- Soon on AIX using MPI_EUILIB / MPI_EUILIBPATH
- Simply compile TAU with MPI support and prefix your MPI program with tau_load.sh
  - % mpirun -np 4 tau_load.sh a.out
- Requires shared library MPI
- Approach will work with other shared libraries
Using TAU

- Install TAU
  - Configuration
  - Measurement library creation
- Instrument application
  - Manual or automatic source instrumentation
  - Instrumented library (e.g., MPI - wrapper interposition library)
  - Binary instrumentation
- Create performance experiments
  - Integrate with application build environment
  - Set experiment variables
- Execute application
- Analyze performance

Integration with Application Build Environment

- Try to minimize impact on user’s application build procedures
- Handle process of parsing, instrumentation, compilation, linking
- Dealing with Makefiles
  - Minimal change to application Makefile
  - Avoid changing compilation rules in application Makefile
  - No explicit inclusion of rules for process stages
- Some applications do not use Makefiles
  - Facilitate integration in whatever procedures used
- Two techniques:
  - TAU shell scripts (tau_<compiler>.sh)
    - Invokes all PDT parser, TAU instrumenter, and compiler
  - TAU_COMPILER
Tau_[cxx,cc,f90].sh

```bash
# set TAU_MAKEFILE and TAU_OPTIONS environment vars
CC = tau_cc.sh
F90 = tau_f90.sh
CFLAGS =
LIBS = -lm
OBJS = f1.o f2.o f3.o ... fn.o

app: $(OBJS)
  $(F90) $(LDFLAGS) $(OBJS) -o $@ $(LIBS)
.c.o:
  $(CC) $(CFLAGS) -c $<
.f90.o:
  $(F90) $(FFLAGS) -c $<
```

Auto Instrumentation using TAU_COMPILER

- $(TAU_COMPILER) variable defined in stub Makefile
- Invokes PDT parser, TAU instrumentor, compiler through tau_compiler.sh shell script
- Requires minimal changes to application Makefile
  - Compilation rules are not changed
  - User adds $(TAU_COMPILER) before compiler name
  - F90=mpxlF90
    - Changes to
      - F90=$(TAU_COMPILER) mpxlF90
- Passes options from TAU stub Makefile to the compilation stages
- Use tau_cxx.sh, tau_cc.sh, tau_f90.sh scripts OR $(TAU_COMPILER)
- Uses original compilation command if an error occurs
TAU_COMPILER - Improving Integration in Makefiles

```
include /usr/tau-2.15.2/rs6000/Makefile.tau-mpi-pdt
CXX = $(TAU_COMPILER) mpCC
F90 = $(TAU_COMPILER) mpxlf90_r
CFLAGS =
LIBS = -lm
OBJJS = f1.o f2.o f3.o ... fn.o

app: $(OBJJS)
   $(CXX) $(LDFLAGS) $(OBJJS) -o $@ $(LIBS)
.cpp.o:
   $(CXX) $(CFLAGS) -c $<
```