Performance Analysis Tools

Karl Fuerlinger
karl@cs.utk.edu

With slides from Sameer Shende, Shirley Moore, Bernd Mohr, Hans Christian Hoppe, and others
Outline

- **Motivation**
  - Why do we care about performance

- **Concepts and Definitions**
  - The performance analysis cycle
  - Instrumentation
  - Measurement: profiling vs. tracing
  - Analysis: manual vs. automatic
  - Current trends: automation, dealing with extreme scale

- **Example Tools**
  - **PAPI**: Performance Application Programming Interface
  - **ompP**: Profiling of OpenMP applications
  - **KOJAK/SCALASCA**: Automated bottleneck detection of MPI/OpenMP applications
  - **Vampir**: Trace Visualization
  - **TAU**: Toolset for profiling and tracing of MPI/OpenMP/Java/Python applications
Motivation

- Performance Analysis is essential
  - Large investments in HPC systems
  - Solve larger problems
  - Solve problems faster
Concepts and Definitions

- The performance optimization cycle

- Code Development
- Measure
- Analyze
- Modify / Tune
- Instrumentation
- Functional complete and correct program
- Complete, correct and well-performing program
- Usage / Production
Instrumentation

- Add measurement probes to the code
- Different techniques for different levels
- Different overheads and levels of accuracy with each technique
Instrumentation Examples

Source code instrumentation
- **User added** time measurement, etc. (e.g., `printf()`, `gettimeofday()`)
- Many tools expose mechanisms to add user-defined instrumentation
- For example, to instrument functions, mark source code phases (initialization, termination, …)
- Examples:
  - KOJAK/Opapi mechanism:
    
    ```
    #pragma pomp inst begin(foo)
    #pragma pomp inst end(foo)
    ```
  - Scalasca/KOJAK mechanism:
    
    ```
    ELG_USER_START( "name" );
    . . .
    ELG_USER_END( "name" );
    ```
Instrumentation Examples

- Preprocessor Instrumentation
  - Example: Instrumenting OpenMP constructs with Opari
  - Preprocessor operation
    - Opari example:

```
POMP_Parallel_fork [master]
#pragma omp parallel {
  POMP_Parallel_begin [team]
  /* ORIGINAL CODE in parallel region */
  POMP_Barrier_Enter [team]
  #pragma omp barrier
  POMP_Barrier_Exit [team]
  POMP_Parallel_end [team]
}
POMP_Parallel_join [master]
```

Instrumentation added by Opari
Instrumentation Examples

- Compiler Instrumentation
  - Many compilers can instrument functions automatically
  - GNU compiler flag: `-finstrument-functions`
  - Automatically calls functions on function entry/exit that a tool can implement
  - Not standardized across compilers, often undocumented flags, sometimes not available at all
  - GNU compiler example:
    Parameters are addresses of caller and callee

```c
void __cyg_profile_func_enter(void *this, void *callsite) {
    /* called on function entry */
}

void __cyg_profile_func_exit(void *this, void *callsite) {
    /* called just before returning from function */
}
```
Instrumentation Examples

- Library Instrumentation:

  ![Diagram](image)

  - MPI library interposition
    - All functions are available under two names: MPI\_xxx and PMPI\_xxx, MPI\_xxx symbols are *weak*, can be over-written by interposition library
    - Measurement code in the interposition library measures begin, end, transmitted data, etc… and calls corresponding PMPI routine.
    - Not all MPI functions need to be instrumented
Instrumentation Examples

- **Binary Runtime Instrumentation**
  - Dynamic patching of the executing program
  - Example: Paradyn tool, Dyninst API

- **Base trampolines/Mini trampolines**
  - Base trampolines handle storing current state of program so instrumentations do not affect execution
  - *Mini trampolines* are the machine-specific realizations of predicates and primitives
  - One base trampoline may handle many mini-trampolines, but a base trampoline is needed for every instrumentation point

- **Binary instrumentation difficult!**
  - Have to deal with
    - Compiler optimizations
    - Branch delay slots
    - Different sizes of instructions for x86 (may increase the number of instructions that have to be relocated)
    - Creating and inserting mini trampolines somewhere in program (at end?)
    - Limited-range jumps may complicate this
Measurement

- Profiling vs. Tracing

- Profiling
  - Summary statistics of performance metrics
    - number of times a routine was invoked
    - exclusive, inclusive time/hpm counts spent executing it
    - number of instrumented child routines invoked, etc.
    - structure of invocations (calltrees/callgraphs)
    - memory, message communication sizes

- Tracing
  - When and where took events place along a global timeline
    - timestamped log of events
    - message communication events (sends/receives) are tracked
    - shows when and from/to where messages were sent
    - large volume of performance data generated leads to more perturbation in the program
Profiling

- Recording of summary information during execution
  - inclusive, exclusive time, # calls, hardware counter statistics, ...
- Reflects performance behavior of program entities
  - functions, loops, basic blocks
  - user-defined "semantic" entities
- Very good for low-cost performance assessment
- Helps to expose performance bottlenecks and hotspots
- Implemented through
  - **sampling**: periodic OS interrupts or hardware counter traps
  - **instrumentation**: direct insertion of measurement code
Measurement: Tracing

Tracing

- Recording of information about significant points (events) during program execution
  - entering/exiting code region (function, loop, block, …)
  - thread/process interactions (e.g., send/receive message)
- Save information in event record
  - timestamp
  - CPU identifier, thread identifier
  - Event type and event-specific information
- Event trace is a time-sequenced stream of event records
- Can be used to reconstruct dynamic program behavior
- Typically requires code instrumentation
Tracing Example: Instrumentation, Monitor, Trace

CPU A:

```c
void master {
    trace(ENTER, 1);
    ...
    trace(SEND, B);
    send(B, tag, buf);
    ...
    trace(EXIT, 1);
}
```

CPU B:

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

Event definition:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>master</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>slave</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

EVENTS:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>58</td>
<td>A</td>
<td>ENTER 1</td>
</tr>
<tr>
<td>60</td>
<td>B</td>
<td>ENTER 2</td>
</tr>
<tr>
<td>62</td>
<td>A</td>
<td>SEND B</td>
</tr>
<tr>
<td>64</td>
<td>A</td>
<td>EXIT 1</td>
</tr>
<tr>
<td>68</td>
<td>B</td>
<td>RECV A</td>
</tr>
<tr>
<td>69</td>
<td>B</td>
<td>EXIT 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Tracing: Timeline Visualization

<table>
<thead>
<tr>
<th></th>
<th>master</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>slave</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

... | A | ENTER | 1 |
58  | A | SEND  | B |
60  | B | ENTER | 2 |
62  | A | EXIT  | 1 |
64  | A | EXIT  | 1 |
68  | B | RECV  | A |
69  | B | EXIT  | 2 |

... |      |      |

- **A**: master
- **B**: slave

Timeline showing events and their timing.
Performance Data Analysis

- Draw conclusions from measured performance data
- Manual analysis
  - Visualization
  - Interactive exploration
  - Statistical analysis
- Automatic analysis
  - Try to cope with huge amounts of performance data with automatic analysis or automatic reduction
  - Examples: Paradyn/KOJAK, Scalasca
Vampir: Timeline view
Trace File Visualization

- Vampir: message communication statistics
3D performance data exploration

- Paraprof viewer (from the TAU toolset)
The Case for Automating Performance Analysis

- **Size of supercomputers**
  - Several thousand of processors
  - Trend to multicore

- **Trend for the future**
  - Single core performance maxed out
  - Increased levels of parallelism

- **Enormous amounts of performance data when tracing**
  - Several gigabytes or even terabytes
  - Overwhelms user

- **Not all programmers are performance experts**
  - Scientists want to focus on their domain
  - Need to keep up with new machines

- **Automation can solve some of these issues**
Idea: Automate the process of detecting performance problems
  – Detect problems automatically (and possibly recommend solutions)
  – Integrate expert knowledge into the tool

Another advantage: Decouple analysis wrt. time and space
  – Analysis can happen at any time
  – Analysis can happen distributed and automatically
Performance Optimization Best Practices

1. Collect basic routine-level timing profile to determine where most time is being spent

2. Collect routine-level hardware counter data to determine types of performance problems

3. Collect callpath profiles to determine sequence of events causing performance problems

4. Conduct finer-grained profiling and/or tracing to pinpoint performance bottlenecks
   - Loop-level profiling with hardware counters
   - Tracing of communication operations
PAPI – Performance Application Programming Interface
What is PAPI

- **Middleware** that provides a consistent programming interface for the performance counter hardware found in most major micro-processors.
- Started as a Parallel Tools Consortium project in 1998
- Goal was to produce a specification for a portable interface to the hardware performance counters available on most modern microprocessors.

Countable events are defined in two ways:
- Platform-neutral **Preset Events** (e.g., PAPI_TOT_INS)
- Platform-dependent **Native Events** (e.g., L3_MISSES)

- Preset Events can be derived from multiple Native Events
- All events are referenced by name and collected into EventSets for sampling
- Events can be **multiplexed** if counters are limited
- Statistical **sampling** and **profiling** is implemented by:
  - Software overflow with timer driven sampling
  - Hardware overflow if supported by the platform
**PAPI Hardware Events**

- **Preset Events**
  - Standard set of over 100 events for application performance tuning
  - Use `papi_avail` utility to see what preset events are available on a given platform
  - No standardization of the exact definition
  - Mapped to either single or linear combinations of native events on each platform

- **Native Events**
  - Any event countable by the CPU
  - Same interface as for preset events
  - Use `papi_native_avail` utility to see all available native events

- Use `papi_event_chooser` utility to select a compatible set of events
Where is PAPI

- PAPI runs on most modern processors and Operating Systems of interest to HPC:
  - IBM POWER{3, 4, 5} / AIX
  - POWER{4, 5, 6} / Linux
  - PowerPC{-32, -64, 970} / Linux
  - Blue Gene / L
  - Intel Pentium II, III, 4, M, Core, etc. / Linux
  - Intel Itanium{1, 2, Montecito?}
  - AMD Athlon, Opteron / Linux
  - Cray T3E, X1, XD3, XT{3, 4} Catamount
  - Altix, Sparc, SiCortex…
  - …and even Windows {XP, 2003 Server; PIII, Athlon, Opteron}!
  - …but not Mac 😞
PAPI Counter Interfaces

- PAPI provides 3 interfaces to the underlying counter hardware:
  1. The **low** level interface manages hardware events in user defined groups called *EventSets*, and provides access to advanced features.
  2. The **high** level interface provides the ability to start, stop and read the counters for a specified list of events.
  3. **Graphical** and end-user tools provide facile data collection and visualization.
PAPI High-level Interface

- Meant for application programmers wanting coarse-grained measurements
- Calls the lower level API
- Allows only PAPI preset events
- Easier to use and less setup (less additional code) than low-level
- Supports 8 calls in C or Fortran:

  PAPI_start_counters()   PAPI_stop_counters()
PAPI_read_counters()     PAPI_accum_counters()
PAPI_num_counters()      PAPI_flips()
PAPI_ipc()               PAPI_flops()
PAPI High-level Example

```c
#include "papi.h"
#define NUM_EVENTS 2
long_long values[NUM_EVENTS];
unsigned int Events[NUM_EVENTS]={PAPI_TOT_INS,PAPI_TOT_CYC};

/* Start the counters */
PAPI_start_counters((int*)Events,NUM_EVENTS);

/* What we are monitoring... */
do_work();

/* Stop counters and store results in values */
retval = PAPI_stop_counters(values,NUM_EVENTS);
```
PAPI Low-level Interface

- Increased efficiency and functionality over the high level PAPI interface
- Obtain information about the executable, the hardware, and the memory environment
- Multiplexing
- Callbacks on counter overflow
- Profiling
- About 60 functions
PAPI Low-level example

#include "papi.h"
#define NUM EVENTS 2
int Events[NUM EVENTS]={PAPI_FP_INS,PAPI_TOT_CYC};
int EventSet;
long long values[NUM EVENTS];

/* Initialize the Library */
retval = PAPI_library_init(PAPI_VER_CURRENT);

/* Allocate space for the new eventset and do setup */
retval = PAPI_create_eventset(&EventSet);

/* Add Flops and total cycles to the eventset */
retval = PAPI_add_events(EventSet,Events,NUM EVENTS);

/* Start the counters */
retval = PAPI_start(EventSet);

do_work(); /* What we want to monitor*/

/*Stop counters and store results in values */
retval = PAPI_stop(EventSet,values);
Some Tools that use PAPI

- TAU (U Oregon)  http://www.cs.uoregon.edu/research/tau/
- HPCToolkit (Rice Univ)  http://hipersoft.cs.rice.edu/hpctoolkit/
- KOJAK and SCALASCA (UTK, FZ Juelich)  http://icl.cs.utk.edu/kojak/
- PerfSuite (NCSA)  http://perfsuite.ncsa.uiuc.edu/
- Vampir (TU Dresden)  http://www.vamir.eu
- Open|Speedshop (SGI)  http://oss.sgi.com/projects/openspeedshop/
- SvPablo (UNC Renaissance Computing Institute)  http://www.renci.unc.edu/Software/Pablo/pablo.htm
- ompP (UTK)  http://www.ompp-tool.com
Monolithic PAPI Design

- PAPI High Level
- PAPI Low Level
- Hardware Independent Layer
- Portable Layer
- Machine Specific Layer
  - PAPI Machine Dependent Substrate
  - Kernel Extension
  - Operating System
  - Hardware Performance Counters
Component PAPI (PAPI-C)

**Motivation:**
- Hardware counters aren’t just for cpus anymore
  - Network counters; thermal & power measurement…
- Often insightful to measure multiple counter domains at once

**Goals:**
- Support simultaneous access to on- and off-processor counters
- Isolate hardware dependent code in a separable component module
- Extend platform independent code to support multiple simultaneous components
- Add or modify API calls to support access to any of several components
- Modify build environment for easy selection and configuration of multiple available components
Component PAPI design

Framework Layer

PAPI High Level

PAPI Low Level

Hardware Independent Layer

Component Layer

PAPI CPU Dependent Substrate

Kernel Extension

PAPI Network Dependent Substrate

Kernel Extension

Operating System

Hardware Performance Counters

Off-Processor Hardware Counters
Two exposed interfaces to the underlying counter hardware:

1. The **High Level API** provides the ability to start, stop and read the counters for a specified list of events.
2. The **Low Level API** manages hardware events in user defined groups called **EventSets**, and provides access to advanced features.
Component PAPI Status

- PAPI 3.9 technology pre-release available with documentation,
- Implemented Myrinet substrate (native counters)
- Implemented ACPI temperature sensor substrate
- Working on Infiniband and Cray Seastar substrates (access to Seastar counters not available under Catamount but expected under CNL)
- Tested on HPC Challenge benchmarks
- Tested platforms include Pentium III, Pentium 4, Core2Duo, Itanium (I and II) and AMD Opteron
OpenMP Performance Analysis with ompP

- **ompP**: Profiling tool for OpenMP
  - Based on source code instrumentation
  - Works with any combination of compiler and runtime
  - Tested and supported: Linux, Solaris, AIX and Intel, Pathscale, PGI, IBM, gcc, SUN studio compilers
  - Supports HW counters through PAPI

![Diagram of the profiling process]

1. **Source Code** → **Automatic instrumentation of OpenMP constructs, manual region instrumentation** → **Executable**
2. **Settings (env. Vars)** → **Execution on parallel machine** → **Profiling Report**
ompP’s Profiling Report

- Header
  - Date, time, duration of the run, number of threads, used hardware counters,…

- Region Overview
  - Number of OpenMP regions (constructs) and their source-code locations

- Flat Region Profile
  - Inclusive times, counts, hardware counter data

- Callgraph

- Callgraph Profiles
  - With Inclusive and exclusive times

- Overhead Analysis Report
  - Four overhead categories, per-parallel region breakdown, absolute times and percentages

- Performance Property Detection Report
  - Points out common inefficiency situations
Profiling Data

Example profiling data

**Code:**

```c
#pragma omp parallel
{
    #pragma omp critical
    {
        sleep(1)
    }
}
```

**Profile:**

<table>
<thead>
<tr>
<th>TID</th>
<th>execT</th>
<th>execC</th>
<th>bodyT</th>
<th>enterT</th>
<th>exitT</th>
<th>PAPI_TOT_INS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3.00</td>
<td>1</td>
<td>1.00</td>
<td>2.00</td>
<td>0.00</td>
<td>1595</td>
</tr>
<tr>
<td>1</td>
<td>1.00</td>
<td>1</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>6347</td>
</tr>
<tr>
<td>2</td>
<td>2.00</td>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
<td>1595</td>
</tr>
<tr>
<td>3</td>
<td>4.00</td>
<td>1</td>
<td>1.00</td>
<td>3.00</td>
<td>0.00</td>
<td>1595</td>
</tr>
<tr>
<td>SUM</td>
<td>10.01</td>
<td>4</td>
<td>4.00</td>
<td>6.00</td>
<td>0.00</td>
<td>11132</td>
</tr>
</tbody>
</table>

**Components:**

- Region number
- Source code location and region type
- Timing data and execution counts, **depending on the particular construct**
- One line per thread, last line sums over all threads
- Hardware counter data (if PAPI is available and HW counters are selected)
- Data is exact (measured, not based on sampling)
## Flat Region Profile (2)

- Times and counts reported by ompP for various OpenMP constructs

<table>
<thead>
<tr>
<th>construct</th>
<th>main</th>
<th>enter</th>
<th>body</th>
<th>barr</th>
<th>exit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>execT</td>
<td>execC</td>
<td>startT</td>
<td>bodyT</td>
<td>sectionT</td>
</tr>
<tr>
<td>MASTER</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATOMIC</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BARRIER</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLUSH</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>USER REGION</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRITICAL</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>LOCK</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>LOOP</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>WORKSHARE</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>SECTIONS</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>SINGLE</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>PARALLEL</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>PARALLEL LOOP</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>PARALLEL SECTIONS</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>PARALLEL WORKSHARE</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>
Callgraph View
- ‘Callgraph’ or ‘region stack’ of OpenMP constructs
- Functions can be included by using Opari’s mechanism to instrument user defined regions: #pragma pomp inst begin(...), #pragma pomp inst end(...)

Callgraph profile
- Similar to flat profile, but with inclusive/exclusive times

Example:

```c
void foo1()
{
    #pragma pomp inst begin(foo1)
    bar();
    #pragma pomp inst end(foo1)
}

void foo2()
{
    #pragma pomp inst begin(foo2)
    bar();
    #pragma pomp inst end(foo2)
}

void bar()
{
    #pragma omp critical
    {
        sleep(1.0);
    }
}

main()
{
    #pragma omp parallel
    {
        foo1();
        foo2();
    }
}
```
## Callgraph display

<table>
<thead>
<tr>
<th>Incl. CPU time</th>
<th>[APP 4 threads]</th>
</tr>
</thead>
<tbody>
<tr>
<td>32.22 (100.0%)</td>
<td>+R00004 main.c (42-46)</td>
</tr>
<tr>
<td>32.06 (99.50%)</td>
<td>+R00001 main.c (19-21) ('foo1')</td>
</tr>
<tr>
<td>10.02 (31.10%)</td>
<td>+R00003 main.c (33-36) (unnamed)</td>
</tr>
<tr>
<td>16.03 (49.74%)</td>
<td>-R00002 main.c (26-28) ('foo2')</td>
</tr>
</tbody>
</table>

## Callgraph profiles

```
[*00] critical.ia64.ompp
[+01] R00004 main.c (42-46) PARALLEL
[+02] R00001 main.c (19-21) ('foo1') USER REGION
TID execT/I execT/E execC
 0  1.00    0.00  1
 1  3.00    0.00  1
 2  2.00    0.00  1
 3  4.00    0.00  1
SUM 10.01    0.00  4
```

```
[*00] critical.ia64.ompp
[+01] R00004 main.c (42-46) PARALLEL
[+02] R00001 main.c (19-21) ('foo1') USER REGION
[=03] R00003 main.c (33-36) (unnamed) CRITICAL
TID execT execC bodyT/I bodyT/E enterT exitT
 0  1.00  1.00    1.00    0.00  0.00
 1  3.00  1.00    1.00    2.00  0.00
 2  2.00  1.00    1.00    1.00  0.00
 3  4.00  1.00    1.00    3.00  0.00
SUM 10.01 4.00   4.00   6.00  0.00
```
Detection of common inefficiency situations:

- Example: WaitAtBarrier
- Severity is fraction of total runtime lost due to the inefficiency
- Implemented performance properties:
  
  - WaitAtBarrier
  - ImbalanceInParallelRegion
  - ImbalanceInParallelLoop, -Workshare ,-Sections
  - ImbalanceDueToNotEnoughSections
  - ImbalanceDueToUnevenSectionDistribution
  - CriticalSectionContention, LockContention
  - FrequentAtomic
  - InsufficientWorkInParallelLoop
  - UnparallelizedInMasterRegion, -SingleRegion

PROPERTY 'ImbalanceInParallelRegion' holds for 'PARALLEL zaxpy.F (48-81)', with a severity of 0.041476
Overhead Analysis (1)

- Certain timing categories reported by ompP can be classified as overheads:
  - Example: `exitBarT`: Time wasted by threads idling at the exit barrier of work-sharing constructs. Reason is most likely an **imbalanced** amount of work.

- Four overhead categories are defined in ompP:
  - **Imbalance**: waiting time incurred due to an imbalanced amount of work in a worksharing or parallel region.
  - **Synchronization**: overhead that arises due to threads having to synchronize their activity, e.g. `barrier` call.
  - **Limited Parallelism**: idle threads due not enough parallelism being exposed by the program.
  - **Thread management**: overhead for the creation and destruction of threads, and for signaling critical sections, locks as available.
### Overhead Analysis (2)

<table>
<thead>
<tr>
<th>construct</th>
<th>main</th>
<th>enter</th>
<th>body</th>
<th>barr</th>
<th>exit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>execT</td>
<td>execC</td>
<td>enterT</td>
<td>startupt</td>
<td>bodyT</td>
</tr>
<tr>
<td>MASTER</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>ATOMIC</td>
<td>●(S)</td>
<td>●</td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>BARRIER</td>
<td>●(S)</td>
<td>●</td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>FLUSH</td>
<td>●(S)</td>
<td>●</td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>USER REGION</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>CRITICAL</td>
<td>●</td>
<td>●</td>
<td>●(S)</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>LOCK</td>
<td>●</td>
<td>●</td>
<td>●(S)</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>LOOP</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>WORKSHARE</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>SECTIONS</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>SINGLE</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>PARALLEL</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>PARALLEL LOOP</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>PARALLEL SECTIONS</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>PARALLEL WORKSHARE</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

**S**: Synchronization overhead  
**I**: Imbalance overhead  
**M**: Thread management overhead  
**L**: Limited Parallelism overhead
ompP’s Overhead Analysis Report

```
Total runtime (wallclock) : 172.64 sec [32 threads]
Number of parallel regions : 12
Parallel coverage           : 134.83 sec (78.10%)

Parallel regions sorted by wallclock time:

<table>
<thead>
<tr>
<th>Type</th>
<th>Location</th>
<th>Wallclock (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R00011 PARALL</td>
<td>mgrid.F (360-384)</td>
<td>55.75 (32.29)</td>
</tr>
<tr>
<td>R00019 PARALL</td>
<td>mgrid.F (403-427)</td>
<td>23.02 (13.34)</td>
</tr>
<tr>
<td>R00009 PARALL</td>
<td>mgrid.F (204-217)</td>
<td>11.94 (6.92)</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUM</td>
<td></td>
<td>134.83 (78.10)</td>
</tr>
</tbody>
</table>

Overheads wrt. each individual parallel region:

<table>
<thead>
<tr>
<th>Type</th>
<th>Location</th>
<th>Wallclock (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R00011 PARALL</td>
<td>mgrid.F (360-384)</td>
<td>55.75 (32.29)</td>
</tr>
<tr>
<td>R00019 PARALL</td>
<td>mgrid.F (403-427)</td>
<td>23.02 (13.34)</td>
</tr>
<tr>
<td>R00009 PARALL</td>
<td>mgrid.F (204-217)</td>
<td>11.94 (6.92)</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUM</td>
<td></td>
<td>134.83 (78.10)</td>
</tr>
</tbody>
</table>

Overheads wrt. whole program:

<table>
<thead>
<tr>
<th>Type</th>
<th>Location</th>
<th>Wallclock (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R00011 PARALL</td>
<td>mgrid.F (360-384)</td>
<td>55.75 (32.29)</td>
</tr>
<tr>
<td>R00019 PARALL</td>
<td>mgrid.F (403-427)</td>
<td>23.02 (13.34)</td>
</tr>
<tr>
<td>R00009 PARALL</td>
<td>mgrid.F (204-217)</td>
<td>11.94 (6.92)</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUM</td>
<td></td>
<td>134.83 (78.10)</td>
</tr>
</tbody>
</table>
```

Number of threads, parallel regions, parallel coverage

Wallclock time x number of threads

Overhead percentages wrt. this particular parallel region

Overhead percentages wrt. whole program
OpenMP Scalability Analysis

Methodology

- Analyze how overheads behave for increasing thread counts
- Graphs show accumulated runtime over all threads for fixed workload (strong scaling)
- Horizontal line = perfect scalability

Example: NAS parallel benchmarks

- Class C, SGI Altix machine (Itanium 2, 1.6 GHz, 6MB L3 Cache)
Application 314.mgrid_m
- Scales relatively poorly, application has 12 parallel loops, all contribute with increasingly severe load imbalance
- Markedly smaller load imbalance for thread counts of 32 and 16. Only three loops show this behavior
- In all three cases, the iteration count is always a power of two (2 to 256), hence thread counts which are not a power of two exhibit more load imbalance
SPEC OpenMP Benchmarks (2)

- Application 316.applu
  - Super-linear speedup
  - Only one parallel region (ssor.f 138-209) shows super-linear speedup, contributes 80% of accumulated total execution time
  - Most likely reason for super-linear speedup: increased overall cache size
Application 313.swim
- Dominating source of inefficiency is thread management overhead
- Main source: reduction of three scalar variables in a small parallel loop in swim.f 116-126.
- At 128 threads more than 6 percent of the total accumulated runtime is spent in the reduction operation
- Time for the reduction operation is larger than time spent in the body of the parallel region
Application 318.galgel
- Scales very badly, large fraction of overhead not accounted for by ompP (most likely memory access latency, cache conflicts, false sharing)
- One small parallel loop contributes significantly to the bad scaling:
  lapack.f90 5081-5092, accumulated CPU time increases from 107.9 (2 threads) to 1349.1 seconds (32 threads), 32 thread version is only 13% faster than 2 thread version (wall-clock time).
Incremental Profiling (1)

Profiling vs. Tracing
- Profiling:
  - low overhead
  - small amounts of data
  - easy to comprehend, even as simple ASCII text
- Tracing:
  - Large quantities of data
  - hard to comprehend manually
  - allows temporal phenomena to be explained
  - causal relationship of events are preserved

Idea: Combine advantages of profiling and tracing
- Add a temporal dimension to profiling-type performance data
- See what happens during the execution without capturing full traces
- Manual interpretation becomes harder since a new dimension is added to the performance data
Incremental Profiling (2)

- Implementation:
  - Capture and dump profiling reports not only at the end of the execution but several times while the application executes
  - Analyze how profiling reports change over time
  - Capture points need not be regular

"One-shot" Profiling

Incremental Profiling
Incremental Profiling (3)

Possible triggers for capturing profiles:

- **Timer-based, fixed**: capture profiles in regular, uniform intervals: predictable
  storage requirements (depends only on duration of program run, size of dataset).

- **Timer-based, adaptive**: Adapt the capture rate to the behavior of the application:
  dump often if application behavior changes, decrease rate if application behavior
  stays the same

- **Counter overflow based**: Dump a profile if a hardware counter overflows.
  Interesting for floating point intensive application

- **User-added**: Expose API for dumping profiles to the user aligned to outer loop
  iterations or phase boundaries
Incremental Profiling

- Trigger currently implemented in ompP:
  - Capture profiles in regular intervals
  - Timer signal is registered and delivered to profiler
  - Profiling data up to capture point stored to memory buffer
  - Dumped as individual profiling reports at the end of program execution
  - Perl scripts to analyze reports and generate graphs

- Experiments
  - 1 second regular dump interval
  - SPEC OpenMP benchmark suite
    • Medium variant, 11 applications
  - 32 CPU SGI Altix machine
    • Itanium-2 processors with 1.6 GHz and 6 MB L3 cache
    • Used in batch mode
1. Region invocation over time
   - See which OpenMP region was executed how often and when during the execution of the application
   - Either for a particular thread or summed over all threads
   - Two most time-consuming regions of application 328.fma3d:

   ![Graph showing region invocation over time]
   - Time [sec]

2. Region execution time over time
   - same as invocations but use time instead of execution count
3. Overheads over time
   - See how overheads change over the application run
   - How is each $\Delta t$ (1sec) spent for work or for one of the overhead classes:
     - Either for whole program or for a specific parallel region
     - Total incurred overhead=integral under this function

Application: 328.fma3d_m

Initialization in a critical section, effectively serializing the execution for approx. 15 seconds. Overhead=$31/32=96\%$
4. Performance Properties over time
   - Severity: negative impact on performance up to the capture point: percentage of CPU time lost due to inefficiency situation
Incremental Profiling

- Performance counter heatmaps
  - x-axis: Time, y-axis: Thread-ID
  - Color: number of hardware counter events observed during sampling period
  - Application “applu”, medium-sized variant, counter: LOADS RETIRED
  - Visible phenomena: iterative behavior, thread grouping (pairs)
Incremental Profiling

- Performance counter heatmaps (contd.)
  - Application “apsi”, medium-sized variant, counter: FP_OPS RETIRED
  - Visible phenomena: difference in thread behavior. Maybe related to placement of threads on processors
Incremental Profiling

- Performance counter heatmaps contd.
  - Application “galgel”, medium-sized variant, counter: DATA_EAR_CACHE_LAT1024
  - Visible phenomena: iterative behavior, stagger-pattern
  - Middle of the timeline cut-out
SCALASCA/KOJAK
Example: Timeline of MPI Ring Program

![Timeline of MPI Ring Program](image)
“Real World”
“Real World” Example
Example Automatic Analysis: Late Sender
Example Automatic Analysis (2): Wait at NxN
The KOJAK Project

- Kit for Objective Judgement and Automatic Knowledge-based detection of bottlenecks
- Collaborative Work
  - Forschungszentrum Jülich
  - Innovative Computing Laboratory, TN
- World-wide leading project in trace-based automatic analysis
- Started 1998

- Supports
  - C, C++, and Fortran parallel applications
  - Based on MPI, OpenMP, SHMEM, or hybrid

http://www.fz-juelich.de/jsc/kojak/
KOJAK Analysis Process

- **Instrument** parallel application
  - `kinst mpi cc -openmp example.c -o example`

- **Collect** event traces by running application as usual
  - `env OMP_NUM_THREADS=4 mpiexec -n 4 ./example`

- **Merge and unify** collected traces
  - `elg_merge epik_a`

- **Search** trace for event patterns representing inefficiencies

- **Categorize and rank** inefficiencies found
  - `expert epik_a`

- **Browse results**
  - `cube3 epik_a`
Performance properties: What problem?  
Region tree: Where in source code? In what context?  
Color coding: How severe is the problem?  
Location: How is the problem distributed across the machine?
Performance Analysis Tools

- How is the problem distributed across the machine?
- Where in source code?
- In what context?
- What problem?
- How severe is the problem?

Color Coding

Location

- How is the problem distributed across the machine?
- KOJAK in detail
**Basic Idea**

- **“Traditional” Tool**
  - Huge amount of Measurement data
  - For non-standard / tricky cases (10%)
  - For expert users

- **Automatic Tool**
  - Simple: 1 screen + 2 commands + 3 panes
  - Relevant problems and data
  - For standard cases (90% ?!)
  - For “normal” users
  - Starting point for experts

⇒ More productivity for performance analysis process!
The KOJAK Project

- Kit for Objective Judgement and Automatic Knowledge-based detection of bottlenecks
- Forschungszentrum Jülich
- Innovative Computing Laboratory, TN
- Started 1998

Approach
- Instrument C, C++, and Fortran parallel applications
  - Based on MPI, OpenMP, SHMEM, or hybrid
- Collect event traces
- Search trace for event patterns representing inefficiencies
- Categorize and rank inefficiencies found

http://www.fz-juelich.de/jsc/kojak/
KOJAK: Supported Platforms

- **Instrumentation and measurement only** (visual analysis on front-end or workstation)
  - Cray T3E, Cray XD1, Cray X1, and Cray XT3/XT4
  - IBM BlueGene/L
  - NEC SX
  - Fujitsu Primepower (Sparc, Solaris)
  - Hitachi SR-8000

- **Full support (instrumentation, measurement, and automatic analysis)**
  - Linux IA32, IA64, EMT64/x86_64, and PPC based clusters
  - IBM AIX Power3/4/5 based clusters (SP2, Regatta)
  - SGI Irix MIPS based clusters (Origin 2K, Origin 3K)
  - SGI Linux IA64 based clusters (Altix)
  - SUN Solaris Sparc and x86/x86_64 based clusters (SunFire, …)
  - DEC/HP Tru64 Alpha based clusters (Alphaserver, …)
Example Performance Property:

Wrong Order Late Sender

[Diagram showing the location and time with boxes labeled SEND and RECV, indicating a timing issue with wrong order and late sender.]
Performance Properties

- Describe a pattern of events which influences the performance of parallel programs (in a negative or positive way)

- Have a **severity**
  - Defined as execution time lost because of event pattern
  - Allow to **rank** properties found

- Are **classified** in a **hierarchy** from general to specific
  - More general properties simply describe execution time spent on specific parts of a program (e.g., MPI communication)
    - Can be calculated based on “local” events (⇒ profiles)
  - More specific properties describe waiting/blocking time triggered by interacting events on different processors
    - Require detection of “global” event patterns
KOJAK: Basic Pattern Hierarchy

Time

Execution

MPI
- Communication
- IO
- Synchronization

OpenMP
- Synchronization
- Fork
- Flush

SHMEM
- Communication
- Synchronization

Idle Threads

Overhead
KOJAK: MPI Pattern Hierarchy II

- MPI → Synchronization
  - Barrier
    - Wait at Barrier
    - Barrier Completion
  - RMA Sync.
    - Window Mngt.
      - Wait at Create
      - Wait at Free
    - Fence
      - Wait at Fence
    - Locks
    - Active Target
      - Early Wait
      - Late Complete
      - Late Post
KOJAK: OpenMP Pattern Hierarchy

- OpenMP
  - Synchronization
    - Barrier
      - Explicit
        - Wait at Barrier
      - Implicit
        - Wait at Barrier
    - Lock Competition
      - API
      - Critical
  - Fork
  - Flush
KOJAK: SHMEM Pattern Hierarchy

SHMEM

Communication

Collective
- Late Broadcast
- Wait at NxN

RMA

Synchronization

Barrier
- Wait at Barrier

Point to Point
- Lock Competition
- Wait Until

Init/Exit

Memory Mngt.
MPI-1 Pattern: Early Reduce

- Waiting time if the destination process (root) of a collective N-to-1 communication operation enters the operation earlier than its sending counterparts
- Applies to MPI calls MPI_Reduce(), MPI_Gather(), MPI_Gatherv()
Waiting times if the destination processes of a collective 1-to-N communication operation enter the operation earlier than the source process (root)

- MPI-1: Applies to `MPI_Bcast()`, `MPI_Scatter()`, `MPI_Scatterv()`
- SHMEM: Applies to `shmem_broadcast()`
Generic Pattern: Wait at #

- Time spent waiting in front of a collective synchronizing operation call until the last process reaches the operation
- Pattern instances:
  - Wait at NxN (MPI)
  - Wait at Barrier (MPI)
  - Wait at NxN (SHMEM)
  - Wait at Barrier (SHMEM)
  - Wait at Barrier (OpenMP)
  - Wait at Create (MPI-2)
  - Wait at Free (MPI-2)
  - Wait at Fence (MPI-2)
MPI-1 Pattern: Barrier Completion

- Time spent in MPI barriers after the first process has left the operation
**MPI-1 Pattern: Late Sender / Receiver**

- **Late Sender:** Time lost waiting caused by a blocking receive operation posted earlier than the corresponding send operation.

- **Late Receiver:** Time lost waiting in a blocking send operation until the corresponding receive operation is called.
MPI-1 Pattern: Wrong Order

- Late Sender / Receiver patterns caused by messages received/sent in wrong order
- **Sub patterns** of Late Sender / Receiver
MPI-2 Pattern: Early Transfer

- Time lost being blocked in a RMA operation until exposure epoch is opened with \texttt{MPI\_Win\_n\_post}
MPI-2 Pattern: Late Post

- **MPI_Win_start** (variant A) or **MPI_Win_complete** (variant B)
- Block until exposure epoch is opened with **MPI_Win_post**
MPI-2 Pattern: Early Wait

- Blocking time of `MPI_Win_wait` until access epoch is closed by last `MPI_Win_complete`
**MPI-2 Pattern: Late Complete**

- Blocking time of `MPI_Win_wait` until access epoch is closed by last `MPI_Win_complete`
  - **Sub pattern of Early Wait**
  - Portion of blocking time because of unnecessary pause between last RMA operation and last `MPI_Win_complete`
Generic Pattern: Lock Competition

- Time waiting for a lock that had been previously acquired by another thread/process
- Pattern instances:
  - Lock competition, API (OpenMP)
  - Lock competition, Critical (OpenMP)
  - Lock competition (SHMEM)
KOJAK Architecture

- **User program**
  - [TAU instr. +] OPARI
  - Modified program
  - Compiler / Linker
  - PAPI library
  - **Semi-automatic Instrumentation**
    - POMP+PMPI libraries
    - EPILOG trace library

- **Executable**
  - [TAU instr. +] OPARI
  - Modified program
  - Compiler / Linker
  - PAPI library

- **Execute**
  - **Manual Analysis**
    - POMP+PMPI libraries
    - VAMP/Paraver trace converter
    - VTF3/OTF/PRV trace
    - Manual Analysis

- **Event trace**
  - EPILOG
  - EXPERT Analyzer
  - EARL
  - Analysis result
  - EPILOG pattern trace
  - CUBE3 Presenter
  - Automatic Analysis
KOJAK Tool Components

- Instrument user application with **EPILOG** tracing library calls
  - User functions and regions:
    - Manually using **POMP** directives or Tracing API calls
    - Automatically by **compiler**
    - Automatically by **TAU** source instrumentor
  - MPI calls: Automatically by **PMPI** Wrapper Library
  - OpenMP: Automatically by **OPARI** source instrumentor
  - Record HW counters with **PAPI**

- Analyze measured event trace
  - Automatically with **EXPERT** trace analyzer
    (based on **EARL** trace analysis language) and **CUBE3** result visualizer
  - Manually with **VAMPIR**
    (+ **EPILOG-VTF3/OTF** converters)
  - or **Paraver**
    (+ **EPILOG-PRV** converter)
KOJAK Usage Overview

1. Instrument user application
   - Use POMP directives + `kinst-pomp` OR use `kinst`.

2. Run instrumented executable and merge traces
   - Will produce experiment archive `epik_title`
   - `elg_merge epik_title`

3. Search for inefficiency patterns
   - `expert epik_title`

4. Investigate search results
   - `cube3 epik_title`

5. [If necessary] convert traces for trace visualization
   - `elg2otf / elg2prv epik_title`

Default title: “a”

More? See “doc/USAGE”
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
  - Unsupported: Manually using VT 3.0 tracing API functions

- 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
Insert **INST INIT** directive as the first executable line of main

Fortran:

```fortran
!$POMP INST INIT
```

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

Fortran:

```fortran
!$POMP INST BEGIN(name)
...
[ !$POMP INST ALTEND(name) ]
...
!$POMP INST END(name)
```

C/C++:

```
#pragma pomp inst begin(name)
...
[ #pragma pomp inst altend(name) ]
...
#pragma pomp inst end(name)
```

Fortran:

```fortran
FC = kinst-pomp f90
```

C/C++:

```c
CC = kinst-pomp cc
```
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 of manual instrumentation (with directives)
- 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
Alternatively, use EPILOG tracing API functions to mark regions

Fortran:

```fortran
#include 'el_g_user.inc'
call ELG_USER_START('name');
...
call ELG_USER_END('name');
```

C:

```c
#include "el_g_user.h"
ELG_USER_START("name");
...
ELG_USER_END("name");
```

C++:

```c
#include "el_g_user.h"
{
    ELG_TRACER("name");
    ...
}
```

Note, Fortran source files instrumented this way have to be (cpp) preprocessed.

Necessary Makefile changes:

```make
CFLAGS = ... -DEPILOG `kconfig --cflags [--64]`
LDFLAGS= ... `kconfig --libs [--for] [--omp|--hybrid] [--64]`
```

Compile with "-DEPILOG" otherwise the "ELG_*" calls are ignored.

Does not instrument OpenMP automatically! ⇒ use OPARI manually
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
- Blacklisting functions at runtime not always successful

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

- **Fortran:**
  \[ FC = \text{kinst} \ f90 \]

- **C/C++:**
  \[ CC = \text{kinst cc} \]
kinst and kinst-pomp also automatically instrument OpenMP constructs and MPI functions with OPARI if necessary

Current major OPARI limitations

- Does not yet support
  - Varying number of threads in different parallel regions
    - .. user specified
    - .. because of “sequentialized” parallel regions (if, nested)
  - Nested parallelism

- Executed before compiler preprocessor
  ⇒ issues with macros, conditional compilation, includes!

- Fortran: !$OMP END [Parallel] DO directives required
  ⇒ OPARI “ERROR: unbalanced pragma nesting”
Further OPARI Issues

- Note that OPARI is used
  - With `kinst` when compiling in OpenMP mode
  - Always with `kinst-pomp`

- OPARI stores information about instrumented regions
  - By default in file `opari.rc`
    ⇒ add “`rm -f opari.rc`” to “make clean”
  - Needs special care if building ...
    - ... more than one application in one directory
    - ... applications spread over multiple directories
      ⇒ use `kinst --rcfile <file> --`

- See sections “OpenMP” and “OPARI” in `doc/OPEN_ISSUES`
Running instrumented executable will produce experiment archive named epik_title

Measurements aborts if archive directory already exists!

Produces a trace per MPI process inside experiment archive

Needs to be
  – Merged (combined in one global trace file sorted by timestamp)
  – Unified (replace process local IDs with globally valid ones)
  – Command: `elg_merge <experiment-archive>`
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>EPK_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>EPK_TITLE</td>
<td>Title used for experiment archive</td>
<td>&quot; a&quot;</td>
</tr>
<tr>
<td>EPK_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><code>EPK_GDIR</code></td>
<td>Name of global, cluster-wide directory to store final trace file</td>
<td>Platform specific, typically <code>&quot;. &quot;</code></td>
</tr>
<tr>
<td><code>EPK_LDIR</code></td>
<td>Name of node-local directory that can be used to store temporary trace files</td>
<td>Platform specific, typically <code>&quot;/ tmp&quot;</code></td>
</tr>
<tr>
<td><code>EPK_METRICS_SPEC</code></td>
<td>Name of file specifying metrics, groups and relations</td>
<td>KOJAK install directory <code>&quot;doc/METRICS.SPEC&quot;</code></td>
</tr>
</tbody>
</table>

Value for the variables `$EPK_TITLE`, `EPK_GDIR` and `EPK_LDIR` can contain strings of the form `$XYZ` or `$\{XYZ\}$

Evaluation of these environment variables is done during measurement of the instrumented application.
Automatic Analysis Process

- Analysis process implemented by Expert
  - Scan event trace sequentially for patterns
  - if instance found:
    - Determine call path and process/thread affected
    - Calculate severity ::= percentage of total execution time “lost” because of pattern
  - Result stored in “cube” file containing
    - For each pattern: distribution of severity
      - Over all call paths
      - Over machine / nodes / processes / threads

- Usage: `expert <experiment-archive>`
CUBE3 performance result browser

- Representation of results (severity matrix) along three hierarchical axes
  - Performance metric (general → specific)
  - Region tree path
  - System location (machine → node → process → thread)
- Three coupled tree browsers
- Each node displays severity
  - As color: for easy identification of bottlenecks
  - As value: for precise comparison
- Displayed severity depends on state
  - Collapsed (inclusive severity)
  - Expanded (exclusive severity)
- Usage: `cube3 <experiment-archive-or-cube-file>`
Performance Analysis Tools

Location
- How is the problem distributed across the machine?

Region Tree
- Where in source code?
- In what context?

Color Coding
- How severe is the problem?

Performance Property
- What problem?
Performance Analysis Tools

Location
- How is the problem distributed across the machine?

Region Tree
- Where in source code?
- In what context?

Color Coding
- How severe is the problem?
CUBE3 Display Modes

- **Absolute**
  - Absolute values in seconds/number of occurrences
- **Root percent**
  - Percentage relative to root node of the hierarchy
- **External percent**
  - Similar to “Root percent”, but relative to another data set
- **Selection percent**
  - Percentage relative to the node selected in pane to the left

**System tree only:**
- **Peer percent**
  - Percentage relative to maximum of peer values (all values of the current leaf level)
- **Peer distribution**
  - Percentage relative to maximum and minimum ≠ 0 of peer values
EPILOG traces can be converted to Vampir or Paraver traces
- Usage: `elg2otf <trace>.elg`
  ⇒ produces `<trace>.otf` plus OTF auxiliary files
  `elg2otf <experiment-archive>`
  ⇒ produces `<experiment-archive>/epik.otf`

- Usage: `elg2prv <trace>.elg`
  ⇒ produces `<trace>.prv` plus PRV auxiliary files
  `elg2prv <experiment-archive>`
  ⇒ produces `<experiment-archive>/epik.prv`

Vampirtrace MPI traces can be converted to EPILOG traces
- Usage: `vtotf2elg <trace>.otf`
Other Major Features

- **CUBE file post-processing**
  - Process region tree [prune, cut]
  - Comparative analysis of multiple executions [mean, merge, diff]

- **Holistic analysis**
  - Combined hardware counter metrics + time-based analysis

- **Topology-aware analysis**
  - MPI and user-defined topologies
  - Hardware topology (IBM BG and Cray XT only)

- **Extensive one-sided communication analysis**
  - MPI-2 RMA and SHMEM

- **Pattern trace visualization**
  - Investigate pattern instances and relationships
Cutting / Pruning Call Tree Nodes I

- Apps often have:
  - Initialization
  - Main loop
  - Wrapup structure

- Re-rooting of subtrees possible

Karl Fuerlinger
Usage:

- `cube3_cut`  [-r <nodename>]  [-p <nodename>]
  [-o <outfile>]  <cubefile>

- `-r`: Re-root calltree at named node
- `-p`: Prune call tree at named node
- `-o`: Name of output cube file (default: cut.cube)
CUBE Performance Algebra

- “Compute” with CUBE result files
  - Mean
  - Merge
  - Diff

- Raised relief / positive values show improvement
- Sunken relief / negative values show degradation
Holistic Performance Analysis

- Collect
  - series of experiments
  - with different but matching sets of hardware counters

- Merge results using
  - Generic
  - Platform-specific counter hierarchy Specification

- Not yet available for CUBE3!
CUBE Performance Algebra Tools

- **Usage:**

  - `cube3_diff [ -o <outfile> ] <minuend> <subtrahend>`
    - `- o`: Name of output cube file (default: diff.cube)

  - `cube3_merge [ -o <outfile> ] <file1> <file2> ...`
    - `- o`: Name of output cube file (default: merge.cube)

  - `cube3_mean [ -o <outfile> ] <file1> <file2> ...`
    - `- o`: Name of output cube file (default: mean.cube)
    - Can eliminate “noise” by averaging results of several experiments

- **Note that cube3 utilities directly work on CUBE files and NOT on experiment archives!**
KOJAK: sPPM run on (8x16x14) 1792 PEs

- New topology display
- Shows distribution of pattern over HW topology
- Easily scales to even larger systems
Pattern Trace Visualization

- In addition to provide pattern summary profile (CUBE), Expert can also provide a **pattern instance trace**

- **Usage:**

  - `expert [-p] [-s] [-%] [-l <filterlist>] <trace>.elg`
    - `-p`: Pattern trace: generate pattern events
    - `-s`: Pattern trace: generate severity events
    - `-%`: Statistics: print statistics about patterns
    - `-l`: Filter: only patterns listed in `<filterlist>` are considered for `-%`, `-p`, and `-s`. Pattern names can be followed by a minimum severity for the pattern to be considered.
  - Generated pattern trace name: `<trace>_[p][s].elg`
  - For visualization convert to OTF or PRV
Pattern Trace Visualization Example 1

Original Trace

Pattern Trace

Very little P2P is LateSender
Pattern Trace Visualization Example II

OMP imbalance triggers MPI imbalance
Basic EPILOG Trace Tools I

- “Physical” listing of all EPILOG records in trace file
  - `elg_print <elgtrace-or-exparchive>`

- “Logical” listing and simple statistics of EPILOG records in trace file
  - `elg_stat [-d] [-e] [-l] <elgtrace-or-exparchive>`
    - `-d`: print declaration event info
    - `-e`: print event info
    - `-l`: statistics per process/thread instead of global statistics

- Print information about incorrect message order in trace file
  - `elg_msgord <elgtrace-or-exparchive>`
Basic EPILOG Trace Tools II

- Simple time correction for MPI traces [OBSOLETE]
  - `elg_timecorrect <elgtrace-or-exparchive> <outfile>`
  - See `elg_timecorrect -h` for more detailed instructions

- Filter epilog traces in various ways
  - `elg_filter <elgtrace-or-exparchive> <outfile>`
  - See `elg_filter -h` for more detailed instructions
  - Examples:
    - `-T <typename>`: filter all records of this type
    - `-R <regionname>`: filter <ENTER> and <EXITS> of this region
    - `-O <regionname>`: filter all events outside this region
    - `-I <regionname>`: filter all records inside this region
KOJAK ⇔ TAU ⇔ VAMPIR ⇔ Paraver

KOJAK
- OTF

TAU
- VT

TAU
- TRACE

TAU
- EPILOG

TAU
- PROFILE

KOJAK

TAU

EPILOG
trace

EXPERT
Analyzer

CUBE
profile

OTF / VTF3
trace

PRV
trace

pattern
trace

gprof / mpiP
profile

VAMPIR

Paraver

CUBE
Presenter

PerfDMF

PARAPROF
KOJAK Installation Overview

1. **Configure**
   - `configure --prefix=DIR`
   - `[--papidir=DIR]`
   - `[--otfdir=DIR]`
   - Read configure output carefully and follow instructions

2. **[If necessary] Tweak Makefile.defs**

3. **Build**
   - `make`

4. **Install**
   - `make install`

---

Needed for CUBE:
- libxml2
- wxGTK-2.6.X

More? See “INSTALL”

What did I do last time? See `PREFIX/example/configure.log`  
`PREFIX/example/Makefile.defs`
The People Behind It

- JSC Research group
  “Performance Optimization and Programming Environments”
  - Bernd Mohr, Marc-Andre Hermanns, …

- Helmholtz-University Young Investigators Group
  “Performance Analysis of Parallel Programs”
  - Felix Wolf, Brian Wylie, Markus Geimer, Erika Abraham, …

- Helmholtz Virtual Institute High-Productivity Supercomputing
  - With RWTH Aachen, TU Dresden, ICL Tennessee

- EU ITEA2 Project
  “Parallel Programming for Multi-core Architectures (ParMA)”

- Close collaborations with
  - TAU group, University of Oregon
  - Vampir group, TU Dresden
  - Paraver group, BSC/CEPBA Barcelona
Scalasca
## Increasing Importance of Scaling

- Number of Processors share for TOP 500 11/2007

<table>
<thead>
<tr>
<th>NProc</th>
<th>Count</th>
<th>Share</th>
<th>$\Sigma R_{max}$</th>
<th>Share</th>
<th>$\Sigma N_{Proc}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;= 512</td>
<td>11</td>
<td>2.2%</td>
<td>89 TF</td>
<td>1.3%</td>
<td>7,066</td>
</tr>
<tr>
<td>513-1024</td>
<td>60</td>
<td>12.0%</td>
<td>444 TF</td>
<td>6.4%</td>
<td>57,460</td>
</tr>
<tr>
<td>1025-2048</td>
<td>268</td>
<td>53.6%</td>
<td>2,208 TF</td>
<td>31.7%</td>
<td>400,352</td>
</tr>
<tr>
<td>2049-4096</td>
<td>99</td>
<td>19.8%</td>
<td>1,137 TF</td>
<td>16.3%</td>
<td>291,328</td>
</tr>
<tr>
<td>&gt; 4096</td>
<td>62</td>
<td>12.4%</td>
<td>3,088 TF</td>
<td>44.3%</td>
<td>891,889</td>
</tr>
<tr>
<td>Total</td>
<td>500</td>
<td>100%</td>
<td>6,966 TF</td>
<td>100%</td>
<td>1,648,095</td>
</tr>
</tbody>
</table>
Personal Motivation: JSC Machine Hall

1342 Core Pwr4

16K Core BGL

64K Core BGP
Factors Leading to Large Traces

- **Temporal coverage**
  - full
  - partial
  - disabled

- **Granularity / event rate**
  - high
  - low

- **Problem size / execution time**
  - large
  - small

- **Number of event parameters**
  - many
  - few

⇒ **Long Traces**

⇒ **Wide Traces**
Sequential Analysis Process (KOJAK)

Semi-automatic Instrumentation
- source code → multilevel instrumentation → instrumented executable

Measurement
- execution on parallel machine
  - unification + merge
  - rank-local trace files

Automatic Analysis
- sequential trace analyzer
  - pattern profile → CUBE result presenter
  - Global unified trace file
    - Vampir + Paraver visualizer

Manual Analysis
The Scalasca Project

- Scalable Analysis of Large Scale Applications
- Follow-up project to KOJAK
- Started in January 2006
- BSD open-source release v0.9 in August 2007

- Funded by Helmholtz Initiative and Networking Fund

- Objective: develop a scalable version of KOJAK
  - Basic idea: parallelization of analysis
  - Also support scalable profiling
  - Current focus: single-threaded MPI-1 applications

- http://www.scalasca.org/
Scalasca Runtime Summarization

- Event measurements summarized for each call-path during runtime execution
- Summary report produced at finalization

- Provides overview of measured execution
  - Visit frequency
  - Time
  - MPI message statistics
    - Synchronizations := messages w/o data payload
    - Communications := messages with data payload
    - Bytes transferred
  - plus optional hardware counter metrics

- Size independent of length of execution
- Scales to long execution measurements
Parallel Analysis Process I (Scalasca)

Semi-automatic Instrumentation
- Source code
- Multilevel instrumentation
- Instrumented executable

Measurement
- Execution on parallel machine

Parallel Automatic Analysis
- CUBE result presenter
- Unified+collated summary profile
“pruned” result file (solver only)
Runtime Summarization Overview

1. Instrument MPI application as for KOJAK

2. Run instrumented executable with runtime summarization
   - Produces experiment archive `epik_<exe>_<nprocs>_sum`
   - `scan -q -s mpiexec -n <nprocs> <exe> ...`

3. Explore summary analysis report
   - Does necessary post-processing and executes cube3
   - `square epik_<exe>_<numprocs>_sum`
Scalable Automatic Trace Analysis

- **Parallel pattern search to address wide traces**
  - As many processes / threads as used to run the application
    ⇒ Can run in same batch job!!
  - Each process / thread responsible for its “own” local trace data
- **Idea: “parallel replay” of application**
  - Analysis uses communication mechanism that is being analyzed
  - Use MPI P2P operation to analyze MPI P2P communication,
    use MPI collective operation to analyze MPI collectives, ...
  - Communication requirements not significantly higher and (often lower) than
    requirements of target application
- **In-memory trace analysis**
  - Available memory scales with number of processors used
  - Local memory usually large enough to hold local trace
Parallel Analysis Process (Scalasca)

Semi-automatic Instrumentation
- Source code
- Multilevel instrumentation
- Instrumented executable

Measurement
- Execution on parallel machine
- Unified defs + mappings
- Rank-local trace files

Parallel Automatic Analysis
- Parallel trace analyzer
- Collated pattern profile
- CUBE result presenter
- Collated summary profile
Parallel Trace Analysis Overview

1. Instrument **MPI** application as for KOJAK

2. Run instrumented executable **with parallel trace analysis**
   - Produces experiment archive `epik_<exe>_<nprocs>_trace`
   - `env EPK_LDI R='.' scan -q -t mpiexec -n <nprocs> <exe> ...`

3. Explore **trace analysis report**
   - Does necessary post-processing and executes `cube3`
   - `square epik_<exe>_<nprocs>_trace`
**Example: Late Sender**

Sender
- Triggered by send event
- Determine enter event
- Send both events to receiver

Receiver
- Triggered by receive event
- Determine enter event
- Receive remote events
- Detect Late Sender situation
- Calculate & store waiting time
Example: Wait at $N \times N$

- Triggered by collective exit event
- Determine enter events
- Determine & distribute timestamp of latest enter event (max reduction to all)
- Calculate & store waiting time
Trace Analysis SMG2000@22k

- More detailed pattern tree
- Includes now cross-process patterns
Summary
- KOJAK: unique portable automatic performance analysis
- Scalasca: (b)leading-edge parallel automatic performance analysis

Future work
- More scalability enhancements
  - Better parallel unification and collation algorithms
  - Avoiding of trace I/O ??
  - New enhanced client/server-based result presentation
- More and improved support for programming models
  - Multi-threading: OpenMP, POSIX, ...
  - PGAS: UPC, CAF, ...
- Meta-pattern analysis
Software Availability

- All software available under BSD open-source licence
- Sources, documentation & publications

- http://www.fz-juelich.de/jsc/kojak/
- kojak@fz-juelich.de

- http://www.scalasca.org/
- scalasca@fz-juelich.de
Vampir
Vampir main window

Vampir 2.5 main window

- Tracefile loading can be interrupted at any time
- Tracefile loading can be resumed
- Tracefile can be loaded starting at a specified time offset
- Tracefile can be re-written
Aggregated profiling information
- Execution time
- Number of calls

Inclusive or exclusive of called routines.
Vampir state model

- User specifies activities and symbol grouping
- Look at all/any activities or all symbols
Timeline display

- To **zoom**, mark region with the mouse
Timeline display – zoomed
Timeline display – contents

- Shows all selected processes
- Shows state changes (activity color)
- Shows messages, collective and MPI–IO operations

- Can show parallelism display at the bottom
Timeline display – message details

Message information

Click on message line

Message send op

Message receive op
Communication statistics

- **Message statistics** for each process/node pair:
  - Byte and message count
  - min/max/avg message length, bandwidth
Message histograms

- Message statistics by length, tag or communicator
  - Byte and message count
  - Min/max/avg bandwidth
Collective operations

- For each process: mark operation locally

- Connect start/stop points by lines
Collective operations

- Filter collective operations
- Change display style
Collective operations statistics

- Statistics for collective operations:
  - operation counts, Bytes sent/received
  - transmission rates
- Filter for collective operation

MPI_Gather only

All collective operations
Activity chart

- Profiling information for all processes
Global calling tree

- Display for each symbol:
  - Number of calls, min/max. execution time
- Fold/unfold or restrict to subtrees
Process–local displays

- Timeline (showing calling levels)
- Activity chart
- Calling tree (showing number of calls)
Other displays

- **Parallelism display**
- **Pending Messages display**
- **Trace Comparison feature**
  - compare different runs (scalability analysis)
  - compare different processes
Focus on a time interval

- Chose a time interval by **zooming** with the timeline display
- Enable the **Show Timeline Portion** option
- All statistics windows are updated for the selected interval

- Use to focus on one application phase or iteration!
Effects of zooming

- Updated message statistics
- Updated summary
- Select one iteration
Compare traces

- Compare profiling information
  - To check load balance (between processes)
  - To evaluate scalability (different runs)
  - To look at optimization effects (different code versions)

Comparison by routine

Compare processes 6 and 19
TAU Parallel Performance System

- [http://www.cs.uoregon.edu/research/tau/](http://www.cs.uoregon.edu/research/tau/)
- Multi-level performance instrumentation
  - Multi-language automatic source instrumentation
- Flexible and configurable performance measurement
- Widely-ported parallel performance profiling system
  - Computer system architectures and operating systems
  - Different programming languages and compilers
- Support for multiple parallel programming paradigms
  - Multi-threading, message passing, mixed-mode, hybrid
- Integration in complex software, systems, applications
Using TAU: A brief Introduction

- To instrument source code:
  
  `% setenv TAU_MAKEFILE /usr/pkgs/tau/xt3/lib/Makefile.tau-mpi-pdt-pgi`
  
  And use tau_f90.sh, tau_cxx.sh or tau_cc.sh as Fortran, C++ or C compilers:

  `% mpif90 foo.f90`
  
  changes to

  `% tau_f90.sh foo.f90`

- Execute application and then run:

  `% pprof` (for text based profile display)
  
  `% paraprof` (for GUI)
TAU Performance System Architecture

Instrumentation

- 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
- profile I/O
- sampling profiles
- mapping (callpath)

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

Performance data sources
- timing
- hardware counters
- system counters

OS and runtime system modules
- threading
- interrupts
- runtime system
- …

event selection
TAU Performance System Architecture
Program Database Toolkit (PDT)

- **Application / Library**
  - C / C++ parser
  - Fortran parser F77/90/95
  - IL analyzer

Program Database Files

- **Program documentation**
- **Application component glue**
- **C++ / F90/95 interoperability**
- **Automatic source instrumentation**
TAU Instrumentation Approach

- Support for standard program events
  - Routines
  - Classes and templates
  - Statement-level blocks
- Support for user-defined events
  - Begin/End events ("user-defined timers")
  - Atomic events (e.g., size of memory allocated/freed)
  - Selection of event statistics
- Support definition of “semantic” entities for mapping
- Support for event groups
- Instrumentation optimization (eliminate instrumentation in lightweight routines)
Flexible instrumentation mechanisms at multiple levels

- **Source code**
  - manual (TAU API, TAU Component API)
  - automatic
    - C, C++, F77/90/95 (Program Database Toolkit (*PDT*))
    - OpenMP (directive rewriting (*Opari*), *POMP* spec)
- **Object code**
  - pre-instrumented libraries (e.g., MPI using *PMPI*)
  - statically-linked and dynamically-linked
- **Executable code**
  - dynamic instrumentation (pre-execution) (*DynInstAPI*)
  - virtual machine instrumentation (e.g., Java using *JVMPI*)
  - 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

![Diagram showing the flow of information and mapping from source code to performance data.](Diagram.png)
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
Using TAU

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

**configure [OPTIONS]**

{-c++=<CC>, -cc=<cc>}

Specify C++ and C compilers

-pdt=<dir>

Specify location of PDT

-opari=<dir>

Specify location of Opari OpenMP tool

-papi=<dir>

Specify location of PAPI

-vampirtrace=<dir>

Specify location of VampirTrace

-mpi[inc/lib]=<dir>

Specify MPI library instrumentation

-dyninst=<dir>

Specify location of DynInst Package

-shmem[inc/lib]=<dir>

Specify PSHMEM library instrumentation

-python[inc/lib]=<dir>

Specify Python instrumentation

-tag=<name>

Specify a unique configuration name

-epilog=<dir>

Specify location of EPILOG

-slog2

Build SLOG2/Jumpshot tracing package

-otf=<dir>

Specify location of OTF trace package

-arch=<architecture>

Specify architecture explicitly

{-pthread, -sproc}

Use pthread or SGI sproc threads

-openmp

Use OpenMP threads

-jdk=<dir>

Specify Java instrumentation (JDK)

-fortran=[vendor]

Specify Fortran compiler
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
  - CPU TIME Use usertime+system time
  - PAPIWALLCLOCK Use PAPI’s wallclock time
  - PAPIVIRTUAL Use PAPI’s process virtual time
  - SGITIMERS Use fast IRIX timers
  - LINUXTIMERS Use fast x86 Linux timers
TAU Measurement Configuration – Examples

- `./configure --pdt=/usr/pkgs/pkgs/pdtoolkit-3.11 -mpi -arch=xt3`
  - Configure using PDT and MPI for Cray XT3
- `./configure -arch=xt3 -papi=/opt/xt-tools/papi/3.2.1 -mpi -MULTIPLECOUNTERS; make clean install`
  - Use PAPI counters (one or more) with C/C++/F90 automatic instrumentation. Also instrument the MPI library. Use PGI compilers.

- Typically configure multiple measurement libraries
- Each configuration creates a unique `<arch>/lib/Makefile.tau<options>` stub makefile. It corresponds to the configuration options used. e.g.,
  - `/usr/pkgs/tau/xt3/lib/Makefile.tau-mpi-pdt-pgi`
  - `/usr/pkgs/tau/xt3/lib/Makefile.tau-multiplecounters-mpi-papi-pdt-pgi`
% cd /usr/pkgs/tau/xt3/lib; ls Makefile.*pgi
Makefile.tau-pdt-pgi
Makefile.tau-mpi-pdt-pgi
Makefile.tau-callpath-mpi-pdt-pgi
Makefile.tau-mpi-pdt-trace-pgi
Makefile.tau-mpi-compensate-pdt-pgi
Makefile.tau-multiplecounters-mpi-papi-pdt-pgi
Makefile.tau-multiplecounters-mpi-papi-pdt-trace-pgi
Makefile.tau-mpi-papi-pdt-epilog-trace-pgi
Makefile.tau-pdt-pgi...

For an MPI+F90 application, you may want to start with:
Makefile.tau-mpi-pdt-pgi
  Supports MPI instrumentation & PDT for automatic source instrumentation for PGI compilers
Each TAU stub Makefile resides in `<tau>/<arch>/lib` directory

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

Each stub makefile encapsulates the parameters that TAU was configured with.
It represents a specific instance of the TAU libraries. TAU scripts use stub makefiles to identify what performance measurements are to be performed.
Using TAU

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

- **Typically modify application makefile and choose TAU configuration**
  - Select TAU’s stub makefile, change name of compiler in Makefile
  
  % setenv TAU_MAKEFILE /usr/pkgs/tau/xt3/lib/Makefile.tau-mpi-pdt-pgi
  % setenv TAU_OPTIONS ‘-optVerbose -optKeepFiles ...’
  - F90 = tau_f90.sh  CXX = tau_cxx.sh CC = tau_cc.sh

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

- **Execute application**
  
  % mpirun –np <procs> a.out;

- **Analyze performance data**
  - paraprof, vampir, pprof, paraver …
ParaProf Main Window

click left mouse button

% paraprof matmult.ppk
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**
  - -Impi replaced by -lTauMpi -lpmpi -Impi

- **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
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
- On AIX, TAU uses 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 - does not work on XT3
- Approach will work with other shared libraries
Instrumenting MPI Applications

- Under Linux you may use tau_load.sh to launch un-instrumented programs under TAU
  - Without TAU:
    % mpirun -np 4 ./a.out
  - With TAU:
    % ls /usr/pkgs/tau/xt3/lib/libTAU*intel91*
    % mpirun -np 4 tau_load.sh ./a.out
    % mpirun -np 4 tau_load.sh -XrunTAUsh-mpi-pdt-trace.so a.out
    loads <taudir>/<arch>/lib/libTAUsh-mpi-pdt-trace.so shared object

- Under AIX, use tau_poe instead of poe
  - Without TAU:
    % poe a.out -procs 8
  - With TAU:
    % tau_poe a.out -procs 8
    % tau_poe -XrunTAUsh-mpi-pdt-trace.so a.out -procs 8
    chooses <taudir>/<arch>/lib/libTAUsh-mpi-pdt-trace.so

- No change to source code or executables! No need to re-link!
- Only instruments MPI routines. To instrument user routines, you may need to parse the application source code!
-PROFILE Configuration Option

- Generates flat profiles (one for each MPI process)
  - It is the default option.
- Uses wallclock time (gettimeofday() sys call)
- Calculates exclusive, inclusive time spent in each timer and number of calls

% pprof
For routine “int main( )”:

- **Exclusive time**
  - 100-20-50-20=10 secs
- **Inclusive time**
  - 100 secs
- **Calls**
  - 1 call
- **Subrs (no. of child routines called)**
  - 3
- **Inclusive time/call**
  - 100secs

```c
int main( )
{ /* takes 100 secs */
    f1(); /* takes 20 secs */
    f2(); /* takes 50 secs */
    f1(); /* takes 20 secs */
    /* other work */
}

/*
Time can be replaced by counts from PAPI e.g., PAPI_FP_OPS. */
```
-MULTIPLECOUNTERS Configuration Option

- Instead of one metric, profile or trace with more than one metric
  - Set environment variables COUNTER[1-25] to specify the metric
    - `% setenv COUNTER1 GET_TIME_OF_DAY`
    - `% setenv COUNTER2 PAPI_L2_DCM`
    - `% setenv COUNTER3 PAPI_FP_OPS`
    - `% setenv COUNTER4 PAPI_NATIVE_<native_event>`
    - `% setenv COUNTER5 P_WALL_CLOCK_TIME` ...

- When used with –TRACE option, the first counter must be GET_TIME_OF_DAY
  - `% setenv COUNTER1 GET_TIME_OF_DAY`
  - Provides a globally synchronized real time clock for tracing

- -multiplecounters appears in the name of the stub Makefile
- Often used with –papi=<dir> to measure hardware performance counters and time
- papi_native_avail and papi_avail are two useful tools
-PROFILECALLPATH Configuration Option

- Generates profiles that show the calling order (edges & nodes in callgraph)
  - A=>B=>C shows the time spent in C when it was called by B and B was called by A
  - Control the depth of callpath using TAU_CALLPATH_DEPTH env. Variable
  - -callpath in the name of the stub Makefile name
-PROFILECALLPATH Configuration Option

- Generates program callgraph

![Call Graph for n,c,t, 0,0,0 - tmp/private/](image)
Profile Measurement – Three Flavors

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

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

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

- Allows users to enable instrumentation at runtime based on the depth of a calling routine on a callstack.
  - Disables instrumentation in all routines a certain depth away from the root in a callgraph
- TAUDEPTH_LIMIT environment variable specifies depth
  % setenv TAUDEPTH_LIMIT 1
  enables instrumentation in only "main"
  % setenv TAUDEPTH_LIMIT 2
  enables instrumentation in main and routines that are directly called by main

- Stub makefile has -depthlimit in its name:
  setenv TAU_MAKEFILE <taudir>/<arch>/lib/Makefile.tau-icpc-mpi-depthlimit-pdt
-COMPENSATE Configuration Option

- Specifies online compensation of performance perturbation
- TAU computes its timer overhead and subtracts it from the profiles
- Works well with time or instructions based metrics
- Does not work with level 1/2 data cache misses
-TRACE Configuration Option

- Generates event-trace logs, rather than summary profiles
- Traces show when and where an event occurred in terms of location and the process that executed it
- Traces from multiple processes are merged:
  \[
  \text{\% tau\_treemerge.pl}
  \]
  \[
  \begin{itemize}
    \item generates tau.trc and tau.edf as merged trace and event definition file
  \end{itemize}
\]
- TAU traces can be converted to Vampir’s OTF/VTF3, Jumpshot SLOG2, Paraver trace formats:
  \[
  \begin{align*}
    \text{\% tau2otf} & \text{ tau.trc tau.edf app.otf} \\
    \text{\% tau2vtf} & \text{ tau.trc tau.edf app.vpt.gz} \\
    \text{\% tau2slog2} & \text{ tau.trc tau.edf -o app.slog2} \\
    \text{\% tau\_convert -paraver} & \text{ tau.trc tau.edf app.prv}
  \end{align*}
  \]
- Stub Makefile has -trace in its name
  \[
  \text{\% setenv TAU\_MAKEFILE <taudir>/<arch>/lib/Makefile.tau-icpc-mpi-pdt\_trace}
  \]
Performance Evaluation Alternatives

Depthlimit profile

Parameter profile

Callpath/callgraph profile

Flat profile

Phase profile

Trace

Each alternative has:
- one metric/counter
- multiple counters

Volume of performance data
-PROFILEPARAM Configuration Option

- Idea: partition performance data for individual functions based on runtime parameters
- Enable by configuring with `-PROFILEPARAM`
- TAU call: TAU_PROFILE_PARAM1L (value, “name”)
- Simple example:

```c
void foo(long input) {
    TAU_PROFILE("foo", ",", TAU_DEFAULT);
    TAU_PROFILE_PARAM1L(input, "input");
    ...
}
```
Workload Characterization

- 5 seconds spent in function “foo” becomes
  - 2 seconds for “foo [ <input> = <25> ]”
  - 1 seconds for “foo [ <input> = <5> ]”
  - ...

- Currently used in MPI wrapper library
  - Allows for partitioning of time spent in MPI routines based on parameters (message size, message tag, destination node)
  - Can be extrapolated to infer specifics about the MPI subsystem and system as a whole
Simple example, send/receive squared message sizes (0-32MB)

```c
#include <stdio.h>
#include <mpi.h>
int buffer[8*1024*1024];

int main(int argc, char **argv) {
    int rank, size, i, j;
    MPI_Init(&argc, &argv);
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    for (i=0;i<1000;i++)
        for (j=1;j<=8*1024*1024;j*=2) {
            if (rank == 0) {
                MPI_Send(buffer,j,MPI_INT,1,42,MPI_COMM_WORLD);
            } else {
                MPI_Status status;
                MPI_Recv(buffer,j,MPI_INT,0,42,MPI_COMM_WORLD,&status);
            }
        }
    MPI_Finalize();
}
```
Workload Characterization

- Use `tau_load.sh` to instrument MPI routines (SGI Altix)

% icc mpi.c -lmpi
% mpirun -np 2 tau_load.sh -XrunTAU-icpc-mpi-pdt.so a.out
Workload Characterization

- MPI Results (NAS Parallel Benchmark 3.1, LU class D on 16 processors of SGI Altix)
Workload Characterization

- Two different message sizes (~3.3MB and ~4K)
LU spent 0.162 seconds sending messages of size 44880

It got 833.82 Mflops!
Memory Profiling in TAU

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

- **Configuration option -PROFILEHEADROOM**
  - Records headroom (amount of free memory to grow) for each function.
  - Takes one sample at beginning of each function and associates it with the *callstack*.
  - Useful for debugging memory usage on IBM BG/L.

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

Sorted By: number of userEvents

<table>
<thead>
<tr>
<th>NumSamples</th>
<th>Max</th>
<th>Min</th>
<th>Mean</th>
<th>Std. Dev</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>252032</td>
<td>2022.7</td>
<td>1181.2</td>
<td>1534.3</td>
<td>410.04</td>
<td>MODULEHYDRO_ID::HYDRO_ID - Heap Memory (KB)</td>
</tr>
<tr>
<td>252032</td>
<td>2022.8</td>
<td>1181.7</td>
<td>1534.3</td>
<td>410.04</td>
<td>MODULEINTRFC::INTRFC - Heap Memory (KB)</td>
</tr>
<tr>
<td>104559</td>
<td>2023.2</td>
<td>331.13</td>
<td>1526.6</td>
<td>409.54</td>
<td>MODULEEOS3D::EOS3D - Heap Memory (KB)</td>
</tr>
<tr>
<td>63008</td>
<td>2022.7</td>
<td>1182</td>
<td>1534.3</td>
<td>410.01</td>
<td>MODULEUPDATE_SOLN::UPDATE_SOLN - Heap Memory (KB)</td>
</tr>
<tr>
<td>55545</td>
<td>2023.3</td>
<td>333.07</td>
<td>1514.2</td>
<td>408.31</td>
<td>DBASETREE::DBASENEIGBOREBLOCKLIST - Heap Memory (KB)</td>
</tr>
<tr>
<td>51374</td>
<td>2023</td>
<td>1179.4</td>
<td>1497.7</td>
<td>402.53</td>
<td>AMR_PROLONG_GEN_UNK_FUN - Heap Memory (KB)</td>
</tr>
<tr>
<td>42120</td>
<td>2022.7</td>
<td>1187.5</td>
<td>1533.5</td>
<td>409.83</td>
<td>ABUNDANCE_RESTRICT - Heap Memory (KB)</td>
</tr>
<tr>
<td>41958</td>
<td>2023</td>
<td>346.12</td>
<td>1514.9</td>
<td>406.39</td>
<td>AMR_RESTRICT_UNK_FUN - Heap Memory (KB)</td>
</tr>
<tr>
<td>31832</td>
<td>2022.8</td>
<td>1187.8</td>
<td>1534.1</td>
<td>409.91</td>
<td>AMR_RESTRICT_RED - Heap Memory (KB)</td>
</tr>
<tr>
<td>31504</td>
<td>2022.7</td>
<td>1181.8</td>
<td>1534.3</td>
<td>410.04</td>
<td>DIFFUSE - Heap Memory (KB)</td>
</tr>
<tr>
<td>26042</td>
<td>2023</td>
<td>1179.2</td>
<td>1501.9</td>
<td>403.61</td>
<td>AMR_PROLONG_UNK_FUN - Heap Memory (KB)</td>
</tr>
</tbody>
</table>

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

- Instrumentation based observation of global heap memory (not per function)
  - call TAU_TRACK_MEMORY()
  - call TAU_TRACK_MEMORY_HEADROOM()
    - Triggers one sample every 10 secs
  - call TAU_TRACK_MEMORY_HERE()
  - call TAU_TRACK_MEMORY_HEADROOM_HERE()
    - Triggers sample at a specific location in source code
  - call TAU_SET_INTERRUPT_INTERVAL(seconds)
    - To set inter-interrupt interval for sampling
  - call TAU_DISABLE_TRACKING_MEMORY()
  - call TAU_DISABLE_TRACKING_MEMORY_HEADROOM()
    - To turn off recording memory utilization
  - call TAU_ENABLE_TRACKING_MEMORY()
  - call TAU_ENABLE_TRACKING_MEMORY_HEADROOM()
    - To re-enable tracking memory utilization
Detecting Memory Leaks in C/C++

- TAU wrapper library for malloc/realloc/free
- During instrumentation, specify
  - `optDetectMemoryLeaks` option to TAU_COMPILER
  - `% setenv TAU_OPTIONS '-optVerbose -optDetectMemoryLeaks'
  - `% setenv TAU_MAKEFILE <taudir>/<arch>/lib/Makefile.tau-icpc-mpi-pdt...
    - tau_cxx.sh foo.cpp ...
- Tracks each memory allocation/de-allocation in parsed files
- Correlates each memory event with the executing callstack
- At the end of execution, TAU detects memory leaks
- TAU reports leaks based on allocations and the executing callstack
- Set `TAU_CALLPATH_DEPTH` environment variable to limit callpath data
  - default is 2
- Future work
  - Support for C++ new/delete planned
  - Support for Fortran 90/95 allocate/deallocate planned
include /opt/tau/xt3/lib/Makefile.tau-icpc-mpi-pdt
MYOPTS = -optVerbose -optDetectMemoryLeaks
CC= $(TAU_COMPILER) $(MYOPTS) $(TAU_CXX)
LIBS = -lm
OBJS = f1.o f2.o ...
TARGET= a.out
TARGET: $(OBJJS)
   $(F90) $(LDFLAGS) $(OBJJS) -o $@ $(LIBS)
.c.o:
   $(CC) $(CFLAGS) -c $< -o $@
Memory Leak Detection

```plaintext
malloc size <file=simpleinst.cpp, line=26>
malloc size <file=simpleinst.cpp, line=26> : int main(int, char **) => int foo(int) => int bar(int)
free size <file=simpleinst.cpp, line=28>
free size <file=simpleinst.cpp, line=28> : int main(int, char **) => int foo(int) => int bar(int)
malloc size <file=simpleinst.cpp, line=18>
malloc size <file=simpleinst.cpp, line=18> : int main(int, char **) => int foo(int) => int g(int) => int bar(int)
free size <file=simpleinst.cpp, line=21>
free size <file=simpleinst.cpp, line=21> : int main(int, char **) => int foo(int) => int g(int) => int bar(int)
free size <file=simpleinst.cpp, line=21>
```

Karl Fuerlinger

Performance Analysis Tools | 212
Detecting Memory Leaks in Fortran

```fortran
subroutine foo(x)
  integer:: x
  integer, allocatable :: A(:), B(:), C(:)

  print *, "inside foo"
  allocate(A(x), B(x), C(x))
  deallocate(A, C)
  print *, "exiting foo"

end subroutine foo

program main
  call foo(5)
end program main
```
Detecting Memory Leaks in Fortran

<table>
<thead>
<tr>
<th>NumSamples</th>
<th>MaxValue</th>
<th>MinValue</th>
<th>MeanValue</th>
<th>Std. Dev.</th>
<th>Event Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>MEMORY LEAK! malloc size &lt;file=simple.f, variable=B, line=6&gt; : MAIN =&gt; FOO</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>free size &lt;file=simple.f, variable=A, line=7&gt;</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>free size &lt;file=simple.f, variable=A, line=7&gt; : MAIN =&gt; FOO</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>free size &lt;file=simple.f, variable=C, line=7&gt;</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>malloc size &lt;file=simple.f, variable=A, line=6&gt;</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>malloc size &lt;file=simple.f, variable=B, line=6&gt;</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>malloc size &lt;file=simple.f, variable=C, line=6&gt;</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>malloc size &lt;file=simple.f, variable=C, line=6&gt; : MAIN =&gt; FOO</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>malloc size &lt;file=simple.f, variable=B, line=6&gt; : MAIN =&gt; FOO</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>malloc size &lt;file=simple.f, variable=C, line=6&gt; : MAIN =&gt; FOO</td>
</tr>
</tbody>
</table>
TAU_SETUP: A GUI for Installing TAU

- Compilers
  - C Compiler [-cc=]: Default
  - User Defined
  - C++ Compiler [-c++=]: Default
  - User Defined
  - Fortran Compiler [-fortran=]: Default
  - User Defined
  - PDT [-pdt=]: Browse
  - PDT C++ Compiler [-pdt_c++=]: Default
  - User Defined
  - PAPI [-papi=]:
    - PAPI Wallclock [-PAPIWALLCLOCK]
    - PAPI Virtual [-PAPIVIRTUAL]
    - Multiple Counters [-MULTIPLECOUNTERS]

- Message Passing

- Tracing/Profiling
  - Profile [-PROFILE]
  - Profile Headroom [-PROFILEHEADROOM]
  - Profile Memory [-PROFILEMEMORY]

- Threads

- Data Tools

- Misc

- Profile Compensate [-COMPENSATE]
  - Profile Callpath Compensate [-PROFILECALLPATH]

- Trace [-TRACE]
  - Epilog [-epilog=]:
  - SLOG2 [-slog2=]:
  - Use External SLOG2SDK [-slog2=]:

- -COMPENSATE
  - Specifies online compensation of performance perturbation. When this option is used, TAU computes its overhead and subtracts it from the profiles. It can be only used when profiling is chosen. This option works with MULTIPLECOUNTERS as well, but while it is relevant for removing perturbation with wallclock time, it cannot accurately account for perturbation with hardware performance counts (e.g., L1 Data cache misses). See TAU Publication (Europa04) for further information on this option.

Karl Fuerlinger
TAU Manual Instrumentation API for C/C++

- **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);`

- **Name-based API**
  - `TAU_START("timer_name");`
  - `TAU_STOP("timer_name");`

- **User-defined timing**
  - `TAU_PROFILE_TIMER(timer, name, type, group);`
  - `TAU_PROFILE_START(timer);`
  - `TAU_PROFILE_STOP(timer);`
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_START ("big-loop")

    for(int i = 0; i < N; i++){
        work(i);
    }

    TAU_STOP ("big-loop");
}
#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)
!
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
TAU Timers and Phases

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

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

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

- **Dynamic phase**
  - Shows time spent in all routines called by a given invocation of a routine.
  - E.g., “foo() 4 => MPI_Send()” 12 secs, shows that 12 secs were spent in MPI_Send when it was called by the 4th invocation of foo.
SUBROUTINE SUM_OF_CUBES
    integer profiler(2)
    save profiler
    INTEGER :: H, T, U
    
    call TAU_PROFILE_TIMER(profiler, 'SUM_OF_CUBES')
    call TAU_PROFILE_START(profiler)
    ! 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 SUBROUTINE SUM_OF_CUBES
Static Phases and Timers

SUBROUTINE FOO
integer profiler(2)
save profiler

call TAU_PHASE_CREATE_STATIC(profiler, 'foo')
call TAU_PHASE_START(profiler)
call bar()
! Here bar calls MPI_Barrier and we evaluate foo=>MPI_Barrier and foo=>bar
call TAU_PHASE_STOP(profiler)
END SUBROUTINE SUM_OF_CUBES

SUBROUTINE BAR
integer profiler(2)
save profiler
call TAU_PROFILE_TIMER(profiler, 'bar')
call TAU_PROFILE_START(profiler)
call MPI_Barrier()
call TAU_PROFILE_STOP(profiler)
END SUBROUTINE BAR
Dynamic Phases

SUBROUTINE ITERATE(IER, NIT)
   IMPLICIT NONE
   INTEGER IER, NIT
   character(11) taucharary
   integer tauiteration / 0 /
   integer profiler(2) / 0, 0 /
   save profiler, tauiteration

   write (taucharary, '(a8,i3)') 'ITERATE ', tauiteration
   ! Taucharary is the name of the phase e.g., 'ITERATION 23'
   tauiteration = tauiteration + 1

   call TAU_PHASE_CREATE_DYNAMIC(profiler,taucharary)
   call TAU_PHASE_START(profiler)

   IER = 0
   call SOLVE_K_EPSILON_EQ(IER)
   ! Other work
   call TAU_PHASE_STOP(profiler)
### TAU’s ParaProf Profile Browser: Static Timers

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

<table>
<thead>
<tr>
<th>%Total Time</th>
<th>Exclusive</th>
<th>Inclusive</th>
<th>#Calls</th>
<th>#Child Calls</th>
<th>Total Time/Call</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>81.5</td>
<td>2025.003</td>
<td>2025.003</td>
<td>96969</td>
<td>0</td>
<td>0.002</td>
<td>INT_RTE</td>
</tr>
<tr>
<td>6.0</td>
<td>148.335</td>
<td>148.335</td>
<td>11511</td>
<td>0</td>
<td>0.013</td>
<td>MPI_BARRIER()</td>
</tr>
<tr>
<td>2.1</td>
<td>52.692</td>
<td>52.692</td>
<td>124265.5</td>
<td>0</td>
<td>4.24E-04</td>
<td>MPI_RECV()</td>
</tr>
<tr>
<td>2.0</td>
<td>49.561</td>
<td>49.561</td>
<td>1201</td>
<td>0</td>
<td>0.041</td>
<td>CHEMKIN_M::REACTION_RATE</td>
</tr>
<tr>
<td>1.3</td>
<td>33.289</td>
<td>33.289</td>
<td>1200</td>
<td>0</td>
<td>0.028</td>
<td>SOOT_M::GET_SOOT_RATE</td>
</tr>
<tr>
<td>94.6</td>
<td>31.215</td>
<td>2349.911</td>
<td>1200</td>
<td>40800</td>
<td>1.95E-09</td>
<td>HRSF_NEW</td>
</tr>
<tr>
<td>0.6</td>
<td>15.683</td>
<td>15.683</td>
<td>1401</td>
<td>0</td>
<td>0.011</td>
<td>THERMOCHEM_M::CALC_TEMP</td>
</tr>
<tr>
<td>0.6</td>
<td>15.57</td>
<td>15.57</td>
<td>9884</td>
<td>0</td>
<td>0.002</td>
<td>MPI_ALLREDUCE()</td>
</tr>
<tr>
<td>0.6</td>
<td>14.482</td>
<td>14.482</td>
<td>2</td>
<td>0</td>
<td>7.24E-01</td>
<td>READWRITE_SAVEFILE_DATA</td>
</tr>
<tr>
<td>2.6</td>
<td>11.066</td>
<td>65.491</td>
<td>1200</td>
<td>27600</td>
<td>0.055</td>
<td>SOOT_RHSF</td>
</tr>
<tr>
<td>98.1</td>
<td>10.295</td>
<td>2436.084</td>
<td>200</td>
<td>8000</td>
<td>12.18e-08</td>
<td>TSTEP_ERR</td>
</tr>
<tr>
<td>0.4</td>
<td>10.113</td>
<td>10.113</td>
<td>2400</td>
<td>0</td>
<td>0.004</td>
<td>TRANSPORT_M::COMPUOECOIFFICIENTS</td>
</tr>
<tr>
<td>0.3</td>
<td>7.163</td>
<td>7.163</td>
<td>124253</td>
<td>0</td>
<td>5.76E-05</td>
<td>MPI_WAIT()</td>
</tr>
<tr>
<td>88.2</td>
<td>5.442</td>
<td>2191.51</td>
<td>1200</td>
<td>995752</td>
<td>1.82E-04</td>
<td>DPM</td>
</tr>
<tr>
<td>0.5</td>
<td>5.214</td>
<td>12.598</td>
<td>3400</td>
<td>30600</td>
<td>0.004</td>
<td>FILTER_M::FILTER</td>
</tr>
<tr>
<td>0.2</td>
<td>4.912</td>
<td>4.912</td>
<td>1200</td>
<td>0</td>
<td>0.004</td>
<td>TRANSPORT_M::COMPUTESPECIESDIFFFLUX</td>
</tr>
<tr>
<td>0.2</td>
<td>4.884</td>
<td>5.969</td>
<td>1200</td>
<td>7423</td>
<td>0.005</td>
<td>RADIUS_M::GET_ABSORPTION</td>
</tr>
<tr>
<td>1.2</td>
<td>4.635</td>
<td>29</td>
<td>34806</td>
<td>156627</td>
<td>8.31E-04</td>
<td>DERIVATIVE_X</td>
</tr>
<tr>
<td>1.0</td>
<td>4.226</td>
<td>25.686</td>
<td>30606</td>
<td>137727</td>
<td>8.39E-04</td>
<td>DERIVATIVE_8</td>
</tr>
<tr>
<td>0.1</td>
<td>3.669</td>
<td>3.669</td>
<td>1200</td>
<td>0</td>
<td>0.003</td>
<td>TRANSPORT_M::COMPUTHEATFLUX</td>
</tr>
<tr>
<td>0.1</td>
<td>3.382</td>
<td>3.382</td>
<td>124265.5</td>
<td>0</td>
<td>2.72E-05</td>
<td>MPI_Isend()</td>
</tr>
</tbody>
</table>
## Dynamic Timers

![Dynamic Timers](image)

### Performance Analysis Tools

<table>
<thead>
<tr>
<th>Metric Name: Time</th>
<th>Sorted By: Inclusive</th>
<th>Units: hour:minute:seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>%Total Time</strong></td>
<td><strong>Exclusive</strong></td>
<td><strong>Inclusive</strong></td>
</tr>
<tr>
<td>100.0</td>
<td>0:10:0.005</td>
<td>0:10:16.779</td>
</tr>
<tr>
<td>99.8</td>
<td>0:10:0.006</td>
<td>0:10:13.509</td>
</tr>
<tr>
<td>97.5</td>
<td>0:10:7.403</td>
<td>0:10:9.044</td>
</tr>
<tr>
<td>87.3</td>
<td>0:10:23.927</td>
<td>0:10:33.803</td>
</tr>
<tr>
<td>65.8</td>
<td>0:10:0.104</td>
<td>0:10:6.601</td>
</tr>
<tr>
<td>65.7</td>
<td>0:10:0.01</td>
<td>0:10:6.497</td>
</tr>
<tr>
<td>64.3</td>
<td>0:10:56.217</td>
<td>0:10:56.217</td>
</tr>
<tr>
<td>8.2</td>
<td>0:10:45.564</td>
<td>0:10:45.564</td>
</tr>
<tr>
<td>8.2</td>
<td>0:10:0.008</td>
<td>0:10:45.541</td>
</tr>
<tr>
<td>7.7</td>
<td>0:10:0.012</td>
<td>0:10:42.599</td>
</tr>
<tr>
<td>7.7</td>
<td>0:10:8.289</td>
<td>0:10:42.947</td>
</tr>
<tr>
<td><strong>5.1</strong></td>
<td><strong>0:10:0.037</strong></td>
<td><strong>0:10:20.181</strong></td>
</tr>
<tr>
<td>4.3</td>
<td>0:10:23.902</td>
<td>0:10:23.902</td>
</tr>
<tr>
<td>3.0</td>
<td>0:10:16.848</td>
<td>0:10:16.848</td>
</tr>
<tr>
<td>2.7</td>
<td>0:10:0.043</td>
<td>0:10:14.713</td>
</tr>
<tr>
<td>2.6</td>
<td>0:10:16.459</td>
<td>0:10:16.459</td>
</tr>
<tr>
<td>2.6</td>
<td>0:10:13.624</td>
<td>0:10:14.201</td>
</tr>
<tr>
<td>2.3</td>
<td>0:10:12.725</td>
<td>0:10:12.725</td>
</tr>
<tr>
<td>1.9</td>
<td>0:10:3.436</td>
<td>0:10:10.666</td>
</tr>
<tr>
<td>1.9</td>
<td>0:10:0.128</td>
<td>0:10:10.414</td>
</tr>
<tr>
<td>1.7</td>
<td>0:10:9.494</td>
<td>0:10:9.494</td>
</tr>
<tr>
<td>0.9</td>
<td>0:10:0.01</td>
<td>0:10:6.772</td>
</tr>
<tr>
<td>0.8</td>
<td>0:10:6.628</td>
<td>0:10:6.628</td>
</tr>
<tr>
<td>0.7</td>
<td>0:10:0.456</td>
<td>0:10:4.113</td>
</tr>
<tr>
<td>0.7</td>
<td>0:10:2.635</td>
<td>0:10:3.684</td>
</tr>
</tbody>
</table>
Static Phases

MPI_Barrier took 4.85 secs out of 13.48 secs in the DTM Phase.
Dynamic Phases

The first iteration was expensive for INT_RTE. It took 27.89 secs. Other iterations took less time – 14.2, 10.5, 10.3, 10.5 seconds.
Dynamic Phases

Time spent in MPI_Barrier, MPI_Recv,... in DTM ITERATION 1

Breakdown of time spent in MPI_Isend based on its static and dynamic parent phases
Using TAU – A tutorial

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

Application / Library

C / C++ parser

Fortran parser F77/90/95

C / C++ IL analyzer

IL analyzer

Program Database Files

DUCTAPE

PDBhtml

Program documentation

Application component glue

SILOON

CHASM

C++ / F90/95 interoperability

tau_instrumentor

Automatic source instrumentation
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
Using Program Database Toolkit (PDT)

1. Parse the Program to create foo.pdb:
   % cxxparse foo.cpp -I/usr/local/mydir -DMYFLAGS ...
   or
   % cparsing 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
# set TAU_MAKEFILE and TAU_OPTIONS env vars
CC = tau_cc.sh
F90 = tau_f90.sh
CFLAGS =
LIBS = -lm
OBJJS = f1.o f2.o f3.o ... fn.o

app: $(OBJJS)
   $(F90) $(LDFLAGS) $(OBJJS) -o $@ $(LIBS)
.c.o:
   $(CC) $(CFLAGS) -c $<
.f90.o:
   $(F90) $(FFLAGS) -c $<
AutoInstrumentation using TAU_COMPILER

- $(TAU_COMPILER) stub Makefile variable
- 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=mpxf90
      - Changes to
        - F90= $(TAU_COMPILER) mpxf90
- Passes options from TAU stub Makefile to the four compilation stages
- Use `tau_cxx.sh`, `tau_cc.sh`, `tau_f90.sh` scripts OR $(TAU_COMPILER)
- Uses original compilation command if an error occurs
Automatic Instrumentation

- We now provide compiler wrapper scripts
  - Simply replace `mpxlf90` with `tau_f90.sh`
  - Automatically instruments Fortran source code, links with TAU MPI Wrapper libraries.
- Use `tau_cc.sh` and `tau_cxx.sh` for C/C++

Before

```
CXX = mpCC
F90 = mpxlf90_r
CFLAGS =
LIBS = -lm
OBJS = f1.o f2.o f3.o ... fn.o

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

After

```
CXX = tau_cxx.sh
F90 = tau_f90.sh
CFLAGS =
LIBS = -lm
OBJS = f1.o f2.o f3.o ... fn.o

app: $(OBJS)
    $(CXX) $(LDFLAGS) $(OBJS) -o $@
    $(LIBS)
    .cpp.o:
        $(CC) $(CFLAGS) -c $<
```
TAU_COMPILER – Improving Integration in Makefiles

```bash
include /usr/tau-2.15.5/xt3/Makefile.tau-mpi-pdt
CXX = $(TAU_COMPILER) mpicxx
F90 = $(TAU_COMPILER) mpif90
CFLAGS =
LIBS = -lm
OBJJS = f1.o f2.o f3.o ... fn.o

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

- See `<taudir>/<arch>/bin/tau_compiler.sh -help`
- Compilation:
  % mpxlf90 -c foo.f90
  Changes to
  % f95parse foo.f90 $(OPT1)
  % tau_instrumentor foo.pdb foo.f90 -o foo.inst.f90 $(OPT2)
  % mpxlf90 -c foo.f90 $(OPT3)
- Linking:
  % mpxlf90 foo.o bar.o -o app
  Changes to
  % mpxlf90 foo.o bar.o -o app $(OPT4)
- Where options OPT[1-4] default values may be overridden by the user:
  F90 = $(TAU_COMPILER) $(MYOPTIONS) mpxlf90
Optional parameters for $(TAU_COMPILER): [tau_compiler.sh –help]

- **-optVerbose** Turn on verbose debugging messages
- **-optDetectMemoryLeaks** Turn on debugging memory allocations/de-allocations to track leaks
- **-optPdtGnuFortranParser** Use gfparse (GNU) instead of f95parse (Cleanscape) for parsing Fortran source code
- **-optKeepFiles** Does not remove intermediate .pdb and .inst.* files
- **-optPreProcess** Preprocess Fortran sources before instrumentation
- **-optTauSelectFile=””** Specify selective instrumentation file for tau_instrumentor
- **-optLinking=””** Options passed to the linker. Typically
  
  $(TAU_MPI_FLIBS) $(TAU_LIBS) $(TAU_CXXLIBS)

- **-optCompile=””** Options passed to the compiler. Typically
  
  $(TAU_MPI_INCLUDE) $(TAU_INCLUDE) $(TAU_DEFS)

- **-optPdtF95Opts=””** Add options for Fortran parser in PDT (f95parse/gfparse)
- **-optPdtF95Reset=””** Reset options for Fortran parser in PDT (f95parse/gfparse)
- **-optPdtCOpts=””** Options for C parser in PDT (cparse). Typically
  
  $(TAU_MPI_INCLUDE) $(TAU_INCLUDE) $(TAU_DEFS)

- **-optPdtCxxOpts=””** Options for C++ parser in PDT (cxxparse). Typically
  
  $(TAU_MPI_INCLUDE) $(TAU_INCLUDE) $(TAU_DEFS)
Overriding Default Options: TAU_COMPILER

```bash
include /usr/pkgs/tau/xt3/lib/

# Fortran .f files in free format need the -R free option for parsing
# Are there any preprocessor directives in the Fortran source?
MYOPTIONS= -optVerbose -optPreProcess -optPdtF95Opts=''-R free''
F90 = $(TAU_COMPILER) $(MYOPTIONS) ifort
OBJS = f1.o f2.o f3.o ...
LIBS = -Lappdir -lapplib1 -lapplib2 ...

app: $(OBJS)
  $(F90) $(OBJS) -o app $(LIBS)
.f.o:
  $(F90) -c $<
```
Overriding Default Options: TAU_COMPILER

```bash
% cat Makefile
F90 = tau_f90.sh
OBJS = f1.o f2.o f3.o ...
LIBS = -Lappdir -lapplib1 -lapplib2 ...

app: $(OBJS)
    $(F90) $(OBJS) -o app $(LIBS)
.f90.o:
    $(F90) -c $<
% setenv TAU_OPTIONS '-optVerbose -optTauSelectFile=select.tau
    -optKeepFiles'
% setenv TAU_MAKEFILE <taudir>/xt3/lib/Makefile.tau-mpi-pdt
```
Optimization of Program Instrumentation

- Need to eliminate instrumentation in frequently executing lightweight routines
- Throttling of events at runtime:
  \[
  \text{% setenv TAU_THROTTLE 1}
  \]
  Turns off instrumentation in routines that execute over 100000 times (TAU_THROTTLE_NUMCALLS)
  and take less than 10 microseconds of inclusive time per call (TAU_THROTTLE_PERCALL)
- Selective instrumentation file to filter events
  \[
  \text{% tau_instrumentor [options] \(-f \ <file>\) OR}
  \text{% setenv TAU_OPTIONS \(-optTauSelectFile=tau.txt\) }
  \]
- Compensation of local instrumentation overhead
  \[
  \text{% configure \(-COMPENSATE\) }
  \]
Selective Instrumentation File

- Specify a list of routines to exclude or include (case sensitive)
- # is a wildcard in a routine name. It cannot appear in the first column.
  
  ```
  BEGIN_EXCLUDE_LIST
  Foo
  Bar
  D#EMM
  END_EXCLUDE_LIST
  ```

- Specify a list of routines to include for instrumentation
  
  ```
  BEGIN_INCLUDE_LIST
  int main(int, char **)
  F1
  F3
  END_EXCLUDE_LIST
  ```

- Specify either an include list or an exclude list!
Selective Instrumentation File

- Optionally specify a list of files to exclude or include (case sensitive)
- * and ? may be used as wildcard characters in a file name
  
  ```
  BEGIN_FILE_EXCLUDE_LIST
  f*.f90
  Foo?.cpp
  END_FILE_EXCLUDE_LIST
  ```

- Specify a list of routines to include for instrumentation
  
  ```
  BEGIN_FILE_INCLUDE_LIST
  main.cpp
  foo.f90
  END_FILE_INCLUDE_LIST
  ```
Selective Instrumentation File

- User instrumentation commands are placed in INSTRUMENT section
- `?` and `*` used as wildcard characters for file name, `#` for routine name
- `\` as escape character for quotes
- Routine entry/exit, arbitrary code insertion
- Outer-loop level instrumentation

```plaintext
BEGIN_INSTRUMENT_SECTION
loops file="foo.f90" routine="matrix#"
memory file="foo.f90" routine="#"
file="foo.f90" line = 123 code = "  print *, \" Inside foo\"
exit routine = "int foo()" code = "cout <<\"exiting foo\"<<endl;"
END_INSTRUMENT_SECTION
```
Instrumentation Specification

% tau_instrumentor
Usage : tau_instrumentor <pdbfile> <sourcefile> [-o <outputfile>] [-noinline] [-g groupname] [-i headerfile] [-c][-c++][-fortran] [-f <instr_req_file> ]
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
Automatic Outer Loop Level Instrumentation

BEGIN_INSTRUMENT_SECTION
loops file="loop_test.cpp" routine="multiply"
# it also understands # as the wildcard in routine name
# and * and ? wildcards in file name.
# You can also specify the full
# name of the routine as is found in profile files.
#loops file="loop_test.cpp" routine="double multiply#"
END_INSTRUMENT_SECTION

%pprof
NODE 0;CONTEXT 0;THREAD 0:

%Time    Exclusive    Inclusive       #Call      #Subrs Inclusive Name
msec total msec usec/call
---------------------------------------------------------------------------------------
100.0         0.12       25,162    ... 1   25162827 int main(int, char **)
100.0        0.175       25,162           1           4   25162707 double multiply()
90.5       22,778       22,778           1           0   22778959 Loop: double multiply()
[ file = <loop_test.cpp> line,col = <23,3> to <30,3> ]
9.3        2,345        2,345           1           0    2345823 Loop: double multiply()
[ file = <loop_test.cpp> line,col = <38,3> to <46,7> ]
0.1           33           33           1           0      33964 Loop: double multiply()
[ file = <loop_test.cpp> line,col = <16,10> to <21,12> ]
TAU_REDUCE

- Reads profile files and rules
- Creates selective instrumentation file
  - Specifies which routines should be excluded from instrumentation
Optimizing Instrumentation Overhead: Rules

- #Exclude all events that are members of TAU_USER
  #and use less than 1000 microseconds
  TAU_USER:usec < 1000
- #Exclude all events that have less than 100
  microseconds and are called only once
  usec < 1000 & numcalls = 1
- #Exclude all events that have less than 1000 usecs per
  #call OR have a (total inclusive) percent less than 5
  usecs/call < 1000
  percent < 5
- Scientific notation can be used
  - usec > 1000 & numcalls > 400000 & usecs/call < 30 & percent > 25
- Usage:
  % pprof -d > pprof.dat
  % tau_reduce -r rules.txt -o select.tau
Instrumentation of OpenMP Constructs

- **OpenMP Pragma And Region Instrumentor** [UTK, FZJ]
- Source-to-Source translator to insert **POMP** calls around OpenMP constructs and API functions
- **Done:** Supports
  - Fortran77 and Fortran90, OpenMP 2.0
  - C and C++, OpenMP 1.0
  - POMP Extensions
  - EPILOG and TAU POMP implementations
  - Preserves source code information (#line line file)
- **tau_ompcheck**
  - Balances OpenMP constructs (DO/END DO) and detects errors
  - Invoked by tau_compiler.sh prior to invoking Opari
- **KOJAK Project website** [http://icl.cs.utk.edu/kojak]
OpenMP API Instrumentation

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

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

- POMP version
  - Calls omp version internally
  - Can do extra stuff before and after call
call pomp_parallel_fork(d)
!$OMP PARALLEL other-clauses...
call pomp_parallel_begin(d)
call pomp_do_enter(d)
!$OMP DO schedule-clauses, ordered-clauses, lastprivate-clauses
   do loop
   !$OMP END DO NOWAIT
   call pomp_barrier_enter(d)
   !$OMP BARRIER
   call pomp_barrier_exit(d)
call pomp_do_exit(d)
call pomp_parallel_end(d)
!$OMP END PARALLEL DO
call pomp_parallel_join(d)
OpenMP directive instrumentation

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

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

% cd kojak-2.1.1; cp mf/Makefile.defs.ibm Makefile.defs;
    edit Makefile
% make

Builds opari

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

% configure -opari=/usr/contrib/TAU/kojak-2.1.1/opari
  -mpiinc=/usr/lpp/ppe.poe/include
  -mpilib=/usr/lpp/ppe.poe/lib
  -pdt=/usr/contrib/TAU/pdtoolkit-3.9
% make clean; make install
% setenv TAU_MAKEFILE /tau/<arch>/lib/Makefile.tau-...opari-...
% tau_cxx.sh -c foo.cpp
% tau_cxx.sh -c bar.f90
% tau_cxx.sh *o -o app
Dynamic Instrumentation

- TAU uses DyninstAPI for runtime code patching
- Developed by U. Wisconsin and U. Maryland
- http://www.dyninst.org
- \textit{tau\_run} (mutator) loads measurement library
- Instruments mutatee
- MPI issues:
  - one mutator per executable image [TAU, DynaProf]
  - one mutator for several executables [Paradyn, DPCL]
Step I: Install DyninstAPI
[Download from http://www.dyninst.org]
% cd dyninstAPI-4.2.1/core; make
Set DyninstAPI environment variables (including LD_LIBRARY_PATH)
Step II: Configure TAU with Dyninst
% configure --dyninst=/usr/local/dyninstAPI-4.2.1
% make clean; make install
Builds <taudir>/<arch>/bin/tau_run
% tau_run [-o outfile] [-Xrun<libname>] [-f <select_inst_file>] [-v] <infile>
% tau_run -o a.inst.out a.out
Rewrites a.out
% tau_run klargest
Instruments klargest with TAU calls and executes it
% tau_run -XrunTAUsh-papi a.out
Loads libTAUsh-papi.so instead of libTAU.so for measurements
Integrate performance system with VM
- Captures robust performance data (e.g., thread events)
- Maintain features of environment
  - portability, concurrency, extensibility, interoperation
- Allow use in optimization methods

JVM Profiling Interface (JVMPI)
- Generation of JVM events and hooks into JVM
- Profiler agent (TAU) loaded as shared object
  - registers events of interest and address of callback routine
- Access to information on dynamically loaded classes
- No need to modify Java source, bytecode, or JVM
Using TAU with Java Applications

Step I: Sun JDK 1.4+ [download from www.javasoft.com]
Step II: Configure TAU with JDK (v 1.2 or better)

% configure –jdk=/usr/java2 –TRACE –PROFILE
% make clean; make install
Builds <taudir>/<arch>/lib/libTAU.so

For Java (without instrumentation):
% java application

With instrumentation:
% java -XrunTAU application
% java -XrunTAU:exclude=sun/io,java application
Excludes sun/io/* and java/* classes
TAU Profiling of Java Application (SciVis)

Profile for each Java thread

Captures events for different Java packages

24 threads of execution!

Profile for each Java thread

Global routine profile

java/lang/Object wait (J)V
java/io/BufferedInputStream read (J)
java/io/DataInputStream readInt (I)
java/io/DataInputStream readFloat (F)
java/sw/awt/motif/X11Graphics createMenu (Ljava/awt/motif/X11LockViewResources (Lsun/awt/motif/X11Graphics;
java/sw/awt/motif/X11Graphics doDraw (Lsun/awt/motif/X11Graphics;
java/sw/kernel/TimeData2D makeVertex (ZF)J

Profile for each Java thread

Captures events for different Java packages

24 threads of execution!
Using TAU with Python Applications

Step I: Configure TAU with Python

% configure --pythoninc=/usr/include/python2.4/include
% make clean; make install

Builds <taudir>/<arch>/lib/<bindings>/pytau.py and tau.py packages
for manual and automatic instrumentation respectively
% setenv PYTHONPATH $PYTHONPATH:<taudir>/<arch>/lib/[<dir>]

Karl Fuerlinger
Python Automatic Instrumentation Example

```python
#!/usr/bin/env python

import tau
from time import sleep

def f2():
    print "In f2: Sleeping for 2 seconds"
    sleep(2)

def f1():
    print "In f1: Sleeping for 3 seconds"
    sleep(3)

def OurMain():
    f1()
    tau.run('OurMain()')
```

Running:
% setenv PYTHONPATH
<tau>/<arch>/lib/bindings-python
% ./auto.py
Instruments OurMain, f1, f2, print...
Python Instrumentation: SciPy
Performance Analysis

- paraprof profile browser (GUI)
- pprof (text based profile browser)
- TAU traces can be exported to many different tools
  - Vampir/VNG [T.U. Dresden] (formerly Intel (R) Trace Analyzer)
  - EXPERT [FZJ]
  - Jumpshot (bundled with TAU) [Argonne National Lab] ...
Building Bridges to Other Tools: TAU

LEGEND

- **Profile/Trace File Generator**
- **End User Analysis Tool**
- **Analysis/Converter Tool**
- **Profile Database**

- **Profile Format file(s)**
- **Data from a Profile/Trace file generator**
- **Profile file data output**
- **Tracefile data output**

**Profile/Trace File Generator**
- MPE (MPICH)
- OMPtrace
- MPITrace
- OMPITrace
- SCOPUs
- JIS/JACIT infoPerfex
- NanosCompiler Dimemas

**Analysis/Converter Tool**
- Trace format recognition coming soon.

**End User Analysis Tool**
- Trace Format file(s)

**Profile Database**
- Profile file data output

**Data from a Profile/Trace file generator**

**Profile/Trace File Generator**

**Analysis/Converter Tool**

**End User Analysis Tool**

**Profile Database**

**Data from a Profile/Trace file generator**

**Profile/Trace File Generator**

**Analysis/Converter Tool**

**End User Analysis Tool**

**Profile Database**

**Data from a Profile/Trace file generator**

**Profile/Trace File Generator**

**Analysis/Converter Tool**

**End User Analysis Tool**

**Profile Database**

**Data from a Profile/Trace file generator**
TAU Performance System Interfaces

- PDT [U. Oregon, LANL, FZJ] for instrumentation of C++, C99, F95 source code
- PAPI [UTK] for accessing hardware performance counters data
- DyninstAPI [U. Maryland, U. Wisconsin] for runtime instrumentation
- KOJAK [FZJ, UTK]
  - Epilog trace generation library
  - CUBE callgraph visualizer
  - Opari OpenMP directive rewriting tool
- Vampir/VNG Trace Analyzer [TU Dresden]
- VTF3/OTF trace generation library [TU Dresden] (available from TAU website)
- Paraver trace visualizer [CEPBA]
- Jumpshot-4 trace visualizer [MPICH, ANL]
- JVMPI from JDK for Java program instrumentation [Sun]
- Paraprof profile browser/PerfDMF database supports:
  - TAU format
  - Gprof [GNU]
  - HPM Toolkit [IBM]
  - MpiP [ORNL, LLNL]
  - Dynaprof [UTK]
  - PSRun [NCSA]
Performance Database: Storage of MetaData
ParaProf – Flat Profile (Miranda)

8K processors!

node, context, thread

Miranda
- hydrodynamics
- Fortran + MPI
- LLNL
ParaProf – Histogram View (Miranda)

**8k processors**

```
MPI_Alltoall()
```

**16k processors**

```
MPI_Barrier()
```
ParaProf – 3D Full Profile (Miranda)

16k processors
ParaProf – 3D Scatterplot (Miranda)

- Each point is a “thread” of execution
- A total of four metrics shown in relation
- ParaVis 3D profile visualization library – JOGL

32k processors
ParaProf – 3D Scatterplot (SWEEP3D CUBE)
ParaProf – Flat Profile (NAS BT)

How is MPI_Wait() distributed relative to solver direction?

Application routine names reflect phase semantics.
ParaProf – Phase Profile (NAS BT)

Main phase shows nested phases and immediate events
### ParaProf – Callpath Profile (Flash)

**Flash**
- thermonuclear flashes
- Fortran + MPI
- U. Chicago

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODULEHYDROSWEEP:HYDRO_SWEEP</td>
<td>26.474%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO_3D =&gt; MODULEHYDROSWEEP:HYDRO_SWEEP</td>
<td>26.474%</td>
</tr>
<tr>
<td>MODULEHYDRO:HYDRO_1D</td>
<td>24.556%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO_3D =&gt; MODULEHYDROSWEEP:HYDRO_SWEEP =&gt; MODULEHYDRO:HYDRO_1D</td>
<td>24.556%</td>
</tr>
<tr>
<td>MODULEINTRFC:INTRFC</td>
<td>14.931%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO_3D =&gt; MODULEHYDROSWEEP:HYDRO_SWEEP =&gt; MODULEHYDRO:HYDRO_1D =&gt; MODULEINTRFC:INTRFC</td>
<td>14.931%</td>
</tr>
<tr>
<td>MODULEEOS:EOS 3D</td>
<td>4.301%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO_3D =&gt; MODULEHYDROSWEEP:HYDRO_SWEEP =&gt; MODULEHYDRO:HYDRO_1D =&gt; MODULEEOS:EOS 3D</td>
<td>4.301%</td>
</tr>
<tr>
<td>MPI_Ssend</td>
<td>2.678%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO_3D =&gt; MODULEHYDROSWEEP:HYDRO_SWEEP =&gt; MODULEHYDRO:HYDRO_1D =&gt; MODULEEOS:EOS 3D</td>
<td>2.678%</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>3.536%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO_3D =&gt; MODULEHYDROSWEEP:HYDRO_SWEEP =&gt; MODULEHYDRO:HYDRO_1D =&gt; MODULEEOS:EOS 3D</td>
<td>3.536%</td>
</tr>
<tr>
<td>MPI_Waitall</td>
<td>2.727%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO_3D =&gt; MODULEHYDROSWEEP:HYDRO_SWEEP =&gt; MODULEHYDRO:HYDRO_1D =&gt; MODULEEOS:EOS 3D</td>
<td>2.727%</td>
</tr>
</tbody>
</table>

**Fortran + MPI**
- U. Chicago

** thermonuclear flashes **

---

**ParaProf – Callpath Profile (Flash)**

**Flash**
- thermonuclear flashes
- Fortran + MPI
- U. Chicago

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODULEHYDROSWEEP:HYDRO_SWEEP</td>
<td>26.474%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO_3D =&gt; MODULEHYDROSWEEP:HYDRO_SWEEP</td>
<td>26.474%</td>
</tr>
<tr>
<td>MODULEHYDRO:HYDRO_1D</td>
<td>24.556%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO_3D =&gt; MODULEHYDROSWEEP:HYDRO_SWEEP =&gt; MODULEHYDRO:HYDRO_1D</td>
<td>24.556%</td>
</tr>
<tr>
<td>MODULEINTRFC:INTRFC</td>
<td>14.931%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO_3D =&gt; MODULEHYDROSWEEP:HYDRO_SWEEP =&gt; MODULEHYDRO:HYDRO_1D =&gt; MODULEINTRFC:INTRFC</td>
<td>14.931%</td>
</tr>
<tr>
<td>MODULEEOS:EOS 3D</td>
<td>4.301%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO_3D =&gt; MODULEHYDROSWEEP:HYDRO_SWEEP =&gt; MODULEHYDRO:HYDRO_1D =&gt; MODULEEOS:EOS 3D</td>
<td>4.301%</td>
</tr>
<tr>
<td>MPI_Ssend</td>
<td>2.678%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO_3D =&gt; MODULEHYDROSWEEP:HYDRO_SWEEP =&gt; MODULEHYDRO:HYDRO_1D =&gt; MODULEEOS:EOS 3D</td>
<td>2.678%</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>3.536%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO_3D =&gt; MODULEHYDROSWEEP:HYDRO_SWEEP =&gt; MODULEHYDRO:HYDRO_1D =&gt; MODULEEOS:EOS 3D</td>
<td>3.536%</td>
</tr>
<tr>
<td>MPI_Waitall</td>
<td>2.727%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO_3D =&gt; MODULEHYDROSWEEP:HYDRO_SWEEP =&gt; MODULEHYDRO:HYDRO_1D =&gt; MODULEEOS:EOS 3D</td>
<td>2.727%</td>
</tr>
</tbody>
</table>
ParaProf – 3D Full Profile Bar Plot (Flash)

128 processors
ParaProf Bar Plot (Zoom in/out +/-)
ParaProf – Callgraph Zoomed (Flash)
ParaProf - Thread Statistics Table (GSI)

<table>
<thead>
<tr>
<th>Thread Name</th>
<th>Inclusive Time</th>
<th>Exclusive Time</th>
<th>Calls</th>
<th>Child Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECMOD::INIT_SPEC_VARS</td>
<td>0.26</td>
<td>0.26</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MPLInit()</td>
<td>0.056</td>
<td>0.054</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>RADINFO::RADINFO_READ</td>
<td>0.103</td>
<td>0.101</td>
<td>1</td>
<td>1,196</td>
</tr>
<tr>
<td>PCPINFO::PCPINFO_READ</td>
<td>0.042</td>
<td>0.042</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>GLBSOI</td>
<td>5,212.171</td>
<td>0.024</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>MPL_Finalize()</td>
<td>1.004</td>
<td>1.004</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>OBS_PARA</td>
<td>3.635</td>
<td>0.181</td>
<td>1</td>
<td>56</td>
</tr>
<tr>
<td>JFUNC::CREATE_JFUNC</td>
<td>0.142</td>
<td>0.142</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>GUESS_GRIDS::CREATE_CES_BIAS_GRIDS</td>
<td>0.059</td>
<td>0.059</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>READ::GUESS</td>
<td>1,406.412</td>
<td>0.023</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>READ::OBS</td>
<td>3,770.188</td>
<td>0.016</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>READ::BUFFTOVS</td>
<td>44.369</td>
<td>0.254</td>
<td>1</td>
<td>871,535</td>
</tr>
<tr>
<td>W3FS2</td>
<td>0.025</td>
<td>0.012</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>INITIALIZE::INITIALIZE_RTM</td>
<td>0.099</td>
<td>0.001</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>GUESS_GRIDS::CREATE_FRC_GRIDS</td>
<td>0.050</td>
<td>0.050</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MP_FVANAGRID::ALLGETLIST</td>
<td>30.582</td>
<td>0.0</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>ERROR::HANDLER::DISPLAY_MESSAGE</td>
<td>0.0</td>
<td>0.0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>JFUNC::SET_POINTER</td>
<td>0.0</td>
<td>0.0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>OZINFO::OZINFO_READ</td>
<td>0.016</td>
<td>0.016</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>DETER_SUBDOMAIN</td>
<td>0.008</td>
<td>0.008</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>GRIDMOD::CREATE_MAPPING</td>
<td>0.005</td>
<td>0.005</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>INIT_COMMVARS</td>
<td>0.004</td>
<td>0.004</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MP_FVANAGRID::ALLGETLIST_104</td>
<td>10.711</td>
<td>0.0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>GRIDMOD::CREATE_GRID_VARS</td>
<td>0.0</td>
<td>0.0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
### ParaProf - Callpath Thread Relations Window

#### Parent

<table>
<thead>
<tr>
<th>Parent Routine</th>
<th>Children Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.023</td>
<td>3/430</td>
</tr>
<tr>
<td>0.33</td>
<td>3/430</td>
</tr>
<tr>
<td>1.639</td>
<td>1/430</td>
</tr>
<tr>
<td>3725.802</td>
<td>3/430</td>
</tr>
<tr>
<td>216.294</td>
<td>6/430</td>
</tr>
<tr>
<td>20.069</td>
<td>20/430</td>
</tr>
<tr>
<td>3964.18</td>
<td>430</td>
</tr>
</tbody>
</table>

#### Routine

<table>
<thead>
<tr>
<th>Routine</th>
<th>Calls/Tot Calls</th>
<th>Name/id</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.023</td>
<td>2/430</td>
<td>COMPUTE_DERIVED[56]</td>
</tr>
<tr>
<td>0.33</td>
<td>2/430</td>
<td>INSTALLMOD:INSTALL[1708]</td>
</tr>
<tr>
<td>1.639</td>
<td>2/430</td>
<td>OBS_PARA[1808]</td>
</tr>
<tr>
<td>3725.802</td>
<td>2/430</td>
<td>READ_OBS[1860]</td>
</tr>
<tr>
<td>216.294</td>
<td>2/430</td>
<td>SETPUARKDAT[1900]</td>
</tr>
<tr>
<td>20.069</td>
<td>2/430</td>
<td>STPCALCMOD:STPCALC[1942]</td>
</tr>
</tbody>
</table>

#### Children

<table>
<thead>
<tr>
<th>Child Routine</th>
<th>Calls/Tot Calls</th>
<th>Name/id</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.003</td>
<td>1/430</td>
<td>MPI_Allreduce()[1762]</td>
</tr>
<tr>
<td>0.017</td>
<td>1/430</td>
<td>MPI_Boast()[1768]</td>
</tr>
<tr>
<td>0.006</td>
<td>1/430</td>
<td>MPI_Comm_rank()[1765]</td>
</tr>
</tbody>
</table>
Vampir – Trace Analysis (TAU-to-VTF3) (S3D)
Vampir – Trace Zoomed (S3D)
PerfDMF: Performance Data Mgmt. Framework

TAU Performance System

Performance Analysis Programs

Query and Analysis Toolkit

Data Mining (Weka)

Statistics (R / Omega)

Java PerfDMF API

SQL (PostgreSQL, MySQL, DB2, Oracle)

---

Karl Fuerlinger
TAU Portal - www.paratools.com/tauportal
TAU Portal

Performance Analysis Tools| Karl Fuerlinger
Using Performance Database (PerfDMF)

- Configure PerfDMF (Done by each user)
  % perfdmf_configure
  - Choose derby, PostgreSQL, MySQL, Oracle or DB2
  - Hostname
  - Username
  - Password
  - Say yes to downloading required drivers (we are not allowed to distribute these)
  - Stores parameters in your ~/.ParaProf/perfdmf.cfg file

- Configure PerfExplorer (Done by each user)
  % perfexplorer_configure

- Execute PerfExplorer
  % perfexplorer
Performance Data Mining (Objectives)

- Conduct parallel performance analysis process
  - In a systematic, collaborative and reusable manner
  - Manage performance complexity
  - Discover performance relationship and properties
  - Automate process

- Multi-experiment performance analysis

- Large-scale performance data reduction
  - Summarize characteristics of large processor runs

- Implement extensible analysis framework
  - Abstraction / automation of data mining operations
  - Interface to existing analysis and data mining tools
Performance Data Mining (PerfExplorer)

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

- Technology integration
  - Java API and toolkit for portability
  - PerfDMF
  - R-project/Omegahat, Octave/Matlab statistical analysis
  - WEKA data mining package
  - JFreeChart for visualization, vector output (EPS, SVG)
Performance Data Mining (PerfExplorer)
PerfExplorer - Analysis Methods

- Data summaries, distributions, scatter plots
- Clustering
  - $k$-means
  - Hierarchical
- Correlation analysis
- Dimension reduction
  - PCA
  - Random linear projection
  - Thresholds
- Comparative analysis
- Data management views
PerfExplorer - Cluster Analysis

- Performance data represented as vectors - each dimension is the cumulative time for an event
- \( k \)-means: \( k \) random centers are selected and instances are grouped with the "closest" (Euclidean) center
- New centers are calculated and the process repeated until stabilization or max iterations
- Dimension reduction necessary for meaningful results
- Virtual topology, summaries constructed
PerfExplorer - Cluster Analysis (sPPM)
PerfExplorer - Cluster Analysis

- Four significant events automatically selected (from 16K processors)
- Clusters and correlations are visible
PerfExplorer - Correlation Analysis (Flash)

- Describes strength and direction of a linear relationship between two variables (events) in the data
-0.995 indicates strong, negative relationship

As CALC_CUT_BLOCK_CONTRIBUTIONS() increases in execution time, MPI_Barrier() decreases
PerfExplorer - Comparative Analysis

- Relative speedup, efficiency
  - total runtime, by event, one event, by phase
- Breakdown of total runtime
- Group fraction of total runtime
- Correlating events to total runtime
- Timesteps per second
- Performance Evaluation Research Center (PERC)
  - PERC tools study (led by ORNL, Pat Worley)
  - In-depth performance analysis of select applications
  - Evaluation performance analysis requirements
  - Test tool functionality and ease of use
PerfExplorer - Interface

Select experiments and trials of interest

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

Experiment metadata
PerfExplorer - Interface

Select analysis
PerfExplorer - Relative Efficiency Plots

![Relative Efficiency Plot](image-url)

- **Value** vs. **Number of Processors**

Legend:
- B1-std.cheetah
- B1-std.phoenix.dfs.dft
- B1-std.phoenix.dfs.fft
- B1-std.phoenix.scratch.dft
- B1-std.phoenix.scratch.fft
- B1-std.seaborg
- B1-std.tg
PerfExplorer - Relative Efficiency by Routine

Relative Efficiency by Event for GYRO:Time

- Value vs. Number of Processors
- Graph shows efficiency for various event types.
- Event categories include Coll, Coll_tr, I/O, NL, NL_tr, extras, field, lin_RHS, other.
PerfExplorer - Relative Speedup

![Graph showing relative speedup vs. number of processors for different benchmarks and ideal performance.](image_url)
PerfExplorer - Timesteps Per Second

![Graph showing Timesteps Per Second versus Number of Processors](image)
- Cray X1 is the fastest to solution
  - In all 3 tests
- FFT (nl2) improves time
  - B3-gtc only
- TeraGrid faster than p690
  - For B1-std?
- All plots generated automatically
PerfExplorer - Relative Efficiency (B1-std)

- By experiment (B1-std)
  - Total runtime (Cheetah (red))
- By event for one experiment
  - Coll_tr (blue) is significant
- By experiment for one event
  - Shows how Coll_tr behaves for all experiments
PerfExplorer - Runtime Breakdown
Communication grows to over 60% of total runtime

At each timestep, 230 messages between all boundaries: MPI_Bcast = 26%, MPI_Wait = 25% of total for N=1024
TAU Performance System Status

- **Computing platforms (selected)**
  - IBM SP/pSeries/BGL/Cell PPE, SGI Altix/Origin, Cray T3E/SV-1/X1/XT3, HP (Compaq) SC
    (Tru64), Sun, Linux clusters (IA-32/64, Alpha, PPC, PA-RISC, Power, Opteron), Apple (G4/5, OS
    X), Hitachi SR8000, NEC SX Series, Windows …

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

- **Thread libraries (selected)**
  - pthreads, OpenMP, SGI sproc, Java, Windows, Charm++

- **Compilers (selected)**
  - Intel, PGI, GNU, Fujitsu, Sun, PathScale, SGI, Cray, IBM, HP, NEC, Absoft, Lahey, Nagware, …
Vampir, VNG, and OTF

- Commercial trace based tools developed at ZiH, T.U. Dresden
  - Wolfgang Nagel, Holger Brunst and others...
- Vampir Trace Visualizer (aka Intel ® Trace Analyzer v4.0)
  - Sequential program
- Vampir Next Generation (VNG)
  - Client (vng) runs on a desktop, server (vngd) on a cluster
  - Parallel trace analysis
  - Orders of magnitude bigger traces (more memory)
  - State of the art in parallel trace visualization
- Open Trace Format (OTF)
  - Hierarchical trace format, efficient streams based parallel access with VNGD
  - Replacement for proprietary formats such as STF
  - Tracing library available with a evaluation license now. Open source package at SC’06.

http://www.vampir-ng.de
Vampir Next Generation (VNG) Architecture

Parallel Program
- Monitor System
- Process

File System
- Event Streams
- Parallel I/O
- Trace 1
- Trace 2
- Trace 3
- Trace N

Analysis Server
- Worker 1
- Worker 2
- Worker m
- Message Passing
- Master

Visualization Client
- Timeline with 16 visible Traces
- Internet
- Segment Indicator
- 768 Processes Thumbnail

Performance Analysis Tools | 313
VNG Parallel Analysis Server

Worker
- Message Passing
- Session Thread
- Analysis Module
- Event Databases
- Trace Format Driver

M Worker
N Session Threads

Master
- Message Passing
- Session Thread
- Analysis Merger
- Endian Conversion
- Socket Communication

N Session Threads

Visualization
Client

Traces
Scalability of VNG [Holger Brunst, WAPA 2005]

- sPPM
- 16 CPUs
- 200 MB

<table>
<thead>
<tr>
<th>Number of Workers</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Time</td>
<td>47.33</td>
<td>22.48</td>
<td>10.80</td>
<td>5.43</td>
<td>3.01</td>
<td>3.16</td>
</tr>
<tr>
<td>Timeline</td>
<td>0.10</td>
<td>0.09</td>
<td>0.06</td>
<td>0.08</td>
<td>0.09</td>
<td>0.09</td>
</tr>
<tr>
<td>Summary Profile</td>
<td>1.59</td>
<td>0.87</td>
<td>0.47</td>
<td>0.30</td>
<td>0.28</td>
<td>0.25</td>
</tr>
<tr>
<td>Process Profile</td>
<td>1.32</td>
<td>0.70</td>
<td>0.38</td>
<td>0.26</td>
<td>0.17</td>
<td>0.17</td>
</tr>
<tr>
<td>Com. Matrix</td>
<td>0.06</td>
<td>0.07</td>
<td>0.08</td>
<td>0.09</td>
<td>0.09</td>
<td>0.09</td>
</tr>
<tr>
<td>Stack Tree</td>
<td>2.57</td>
<td>1.39</td>
<td>0.70</td>
<td>0.44</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>
VNG Analysis Server Architecture

- Implementation using MPI and Pthreads
- Client/server approach
- MPI and pthreads are available on most platforms
- Workload and data distribution among “physical” MPI processes
- Support of multiple visualization clients by using virtual sessions handled by individual threads
- Sessions are scheduled as threads
TAU Tracing Enhancements

- Configure TAU with `-TRACE -otf=<dir>` option
  ```
  % configure -TRACE -otf=<dir> ...
  Generates tau_merge, tau2vtf, tau2otf tools in <tau>/<arch>/bin directory
  % tau_f90.sh app.f90 -o app
  ```
- Instrument and execute application
  ```
  % mpirun -np 4 app
  ```
- Merge and convert trace files to OTF format
  ```
  % tau_treemerge.pl
  % tau2otf tau.trc tau.edf app.otf [-z][-n <nstreams>]
  % vampir app.otf
  ```
  OR use VNG to analyze OTF/VTF trace files
Environment Variables

- Configure TAU with `-TRACE -otr=<dir>` option
  
  ```
  % configure -TRACE -otr=<dir>
  -MULTIPLECOUNTERS -papi=<dir> -mpi
  -pdt=dir ...
  ```

- Set environment variables
  
  ```
  % setenv TRACEDIR /p/gm1/<login>/traces
  % setenv COUNTER1 GET_TIME_OF_DAY (reqd)
  % setenv COUNTER2 PAPI_FP_INS
  % setenv COUNTER3 PAPI_TOT_CYC ...
  ```

- Execute application
  
  ```
  % mpirun -np 32 ./a.out [args]

  % tau_treemerge.pl
  % tau2otf tau.trc tau.edf app.otf -z
  ```
Using VampirTrace to generate OTF traces

- Configure TAU with `-TRACE -vampirtrace=<dir>` option
  ```
  % configure -TRACE -vampirtrace=<dir> -papi=<dir> -mpi
  -pdt=dir ...
  ```
- Set environment variables
  ```
  % setenv VT_METRICS PAPI_FP_OPS:PAPI_TOT_CYC
  ```
- Execute application
  ```
  % yod -sz 20 ./a.out [args]
  On Cray XT3, this will a.[1..n].uctl, a.[1..n].events.z...
  % vtunify 20 a
  On IBM AIX, running the application will create a.otf after unifying the events
  Unifies the descriptors to generate a.otf
  % vampir a.otf &
Using Vampir Next Generation (VNG v1.4)

```bash
mcr36{shende1}32: srun -N2 -n4 -p pdebug /usr/local/tools/vampir/vngd
Service process resides on "mcr98"
Found license file: /usr/global/tools/vampir/chaos_3_x86_elan3/vng-1.4.0/lic.dat
running...
```

```bash
mcr36{shende1}22: /usr/local/tools/vampir/vng &
[1] 12427
mcr36{shende1}23: Found license file: /usr/global/tools/vampir/chaos_3_x86_elan3/vng-1.4.0/lic.dat
```

```
Server: mcr98.1nl.gov
Port: 30000
Use secure connection (SSL)
Certificate File: [ ]
```
Documentation, Manuals, User Guides

- **PAPI References:**
  - PAPI documentation page available from the PAPI website: 

- **ompP**
  - User guide and manual
    [http://www.ompp-tool.com](http://www.ompp-tool.com)

- **TAU References:**
  - TAU Users Guide and papers available from the TAU website: 
    [http://www.cs.uoregon.edu/research/tau/](http://www.cs.uoregon.edu/research/tau/)

- **VAMPIR References**
  - VAMPIR-NG website
    [http://www.vampir-ng.de/](http://www.vampir-ng.de/)

- **KOJAK References**
  - KOJAK documentation page
    [http://www.fz-juelich.de/zam/kojak/documentation/](http://www.fz-juelich.de/zam/kojak/documentation/)

- **Scalasca**
  - Scalasca documentation
    [http://www.scalasca.org](http://www.scalasca.org)