Using POINT Performance Tools (PAPI, PerfSuite, TAU, Scalasca and Vampir) to Understand and Optimize Multicore Codes

Sameer Shende  
University of Oregon  
sameer@cs.uoregon.edu

David Cronk  
Innovative Computing Laboratory  
University of Tennessee  
cronk@eecs.utk.edu

Rui Liu  
National Center for Supercomputing Applications  
ruiliu@ncsa.illinois.edu

Nick Nystrom  
Pittsburgh Supercomputing Center  
nystrom@psc.edu

LCI 2010, Pittsburgh, March 8, 2010
<table>
<thead>
<tr>
<th>Time</th>
<th>Topic</th>
<th>Speaker</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:30 – 8:45</td>
<td>Introduction to POINT</td>
<td>Nick Nystrom</td>
</tr>
<tr>
<td>8:45 – 9:00</td>
<td>Multicore Processors: Resource Contention &amp; Performance</td>
<td>David Cronk</td>
</tr>
<tr>
<td>9:00 – 9:45</td>
<td>Introduction to Performance Engineering</td>
<td>Sameer Shende</td>
</tr>
<tr>
<td>9:45 – 10:00</td>
<td>POINT LiveDVD</td>
<td>All</td>
</tr>
<tr>
<td>10:00 – 10:30</td>
<td>Break</td>
<td></td>
</tr>
<tr>
<td>10:30 – 11:15</td>
<td>PAPI</td>
<td>David Cronk</td>
</tr>
<tr>
<td>11:15 – 12:00</td>
<td>PerfSuite</td>
<td>Rui Liu</td>
</tr>
<tr>
<td>12:00 – 1:00</td>
<td>Lunch</td>
<td></td>
</tr>
<tr>
<td>1:00 – 3:00</td>
<td>TAU</td>
<td>Sameer Shende</td>
</tr>
<tr>
<td>3:00 – 3:30</td>
<td>Break</td>
<td></td>
</tr>
<tr>
<td>3:30 – 4:00</td>
<td>Vampir</td>
<td>Sameer Shende</td>
</tr>
<tr>
<td>4:00 – 4:30</td>
<td>Scalasca</td>
<td>Sameer Shende</td>
</tr>
<tr>
<td>4:30 – 5:00</td>
<td>Application Case Studies</td>
<td>Nick Nystrom</td>
</tr>
</tbody>
</table>
INTRODUCTION TO POINT

Nick Nystrom
Pittsburgh Supercomputing Center

Allen D. Malony
Performance Research Laboratory, University of Oregon
POINT Project

• Petascale Productivity from Open, Integrated Tools
• “High-Productivity Performance Engineering (Tools, Methods, Training) for NSF HPC Applications”
  – NSF SDCI, Software Improvement and Support
  – University of Oregon, University of Tennessee, National Center for Supercomputing Applications, Pittsburgh Supercomputing Center
  – http://www.nic.uoregon.edu/point
Motivation

• Promise of HPC through scalable scientific and engineering applications
• Performance optimization through effective performance engineering methods
  – Performance analysis / tuning “best practices”
• Productive petascale HPC will require
  – Robust parallel performance tools
  – Training good performance problem solvers
Objectives

• Robust parallel performance environment
  – Uniformly available across NSF HPC platforms

• Promote performance engineering
  – Training in performance tools and methods
  – Leverage NSF TeraGrid EOT

• Work with petascale applications teams

• Community building
Challenges

• Consistent performance tool environment
  – Tool integration, interoperation, and scalability
  – Uniform deployment across NSF HPC platforms

• Useful evaluation metrics and process
  – Make part of code development routine
  – Recording performance engineering history

• Develop performance engineering culture
  – Proceed beyond “hand holding” engagements
Performance Engineering Levels

• Target different performance tool users
  – Different levels of expertise
  – Different performance problem solving needs

• Level 0 (entry)
  – Simpler tool use, limited performance data

• Level 1 (intermediate)
  – More tool sophistication, increased information

• Level 2 (advanced)
  – Access to powerful performance techniques
POINT Project Organization

Vendor / Research HPC Tools

Performance engineering competency

Education and Training

HPC Tools Improvement

NSF TeraGrid / HPC Centers (TACC, SDSC, NCSA)

Tools Support and Application Engagement

Open source parallel performance analysis infrastructure

- University of Oregon (TAU)
- National Center for Supercomputing Applications (PerfSuite)
- University of Tennessee (PAPI, KOJAK)

HPC S&E applications and systems

- Pittsburgh Supercomputing Center (lead pilot site)

Performance Technology Expertise

- Source instrumentation
- Library interposition
- Scalable profile and trace analysis
- Automatic performance diagnosis
- Application and workload characterization

- Binary and dynamic instrumentation
- Runtime control
- Scalable performance visualization
- Performance data mining
- Memory access / usage analysis

- Hardware counter access
- Statistical profiling
- Memory and I/O measurement
- Performance database
- Automatic configuration

- Parallel profiling
- Scalable trace generation
- Kernel–level measurement
- Web-based portal
- Testing and validation

Performance Engineering Process

Level 2 (Advanced)

Level 1 (Intermediate)

Level 0 (Entry)

Testbed Apps
- ENZO
- NAMD
- NEMO3D
- UNRES
- ARPS
Parallel Performance Technology

- PAPI
  - University of Tennessee, Knoxville
- PerfSuite
  - National Center for Supercomputing Applications
- TAU Performance System
  - University of Oregon
- Kojak / Scalasca
  - Research Centre Juelich
- Vampir and VampirTrace
  - T.U. Dresden
Parallel Engineering Training

• User engagement
• User support in TeraGrid
• Training workshops
• Quantify tool impact
• POINT lead pilot site
  – Pittsburgh Supercomputing Center
  – NSF TeraGrid site
Testbed Applications

• ENZO
  – Adaptive mesh refinement (AMR), grid-based hybrid code (hydro+Nbody) designed to do simulations of cosmological structure formation

• NAMD
  – Mature community parallel molecular dynamics application deployed for research in large-scale biomolecular systems

• NEMO3D
  – Quantum mechanical based simulation tool created to provide quantitative predictions for nanometer-scale semiconductor devices
Virtual Institute – High Productivity Supercomputing

**Goal:** Improve the quality and accelerate the development process of complex simulation codes running on highly-parallel computer systems

- Funded by Helmholtz Association of German Research Centres
- Activities
  - Development and integration of HPC programming tools
    - Correctness checking & performance analysis
  - Training workshops
  - Service
    - Support email lists
    - Application engagement
  - Academic workshops

[www.vi-hps.org](http://www.vi-hps.org)

LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes
Partners

Forschungszentrum Jülich
  – Jülich Supercomputing Centre

RWTH Aachen University
  – Centre for Computing and Communication

Technical University of Dresden
  – Centre for Information Services and HPC

University of Tennessee (Knoxville)
  – Innovative Computing Laboratory

Technical University of Munich
  – Chair for Computer Architecture

University of Stuttgart
  – High Performance Computing Centre
Productivity tools

- Marmot
  - Free MPI correctness checking tool
- PAPI
  - Free library interfacing to hardware performance counters
- Scalasca
  - Open-source toolset for analysing the performance behaviour of parallel applications to automatically identify inefficiencies
- Vampir
  - Commercial framework and graphical analysis tool to display and analyse event traces
- VampirTrace
  - Open-source tool generating event traces for analysis and visualization by Vampir
- [Tutorial Live-DVD contains latest tools releases]
Technologies and their integration

- **PAPI**

- **SCALASCA**

- **MARMOT**

- **VAMPIR**

1. Hardware monitoring
2. Error correction
3. Execution
4. Optimization
5. Automatic trace analysis
6. Visual trace analysis

**POINT**

*INT Performance Tools to Understand and Optimize Multicore Codes*
POINT/VI-HPS tool interoperability

- PerfSuite can generate reports in CUBE format
- TAU can use Scalasca & VampirTrace measurement libs and can present reports in PerfSuite & CUBE formats
- TAU & Vampir use OPARI to instrument OpenMP sources, and Scalasca can use TAU source instrumenter
- Scalasca & Vampir traces can be inter-converted
MULTICORE PROCESSORS:
RESOURCE CONTENTION
AND PERFORMANCE

David Cronk
Innovative Computing Lab
University of Tennessee, Knoxville
Introduction

• The number of cores on a chip is only going to continue to increase
  – Chips with 4 cores per socket are common today
  – Chips with 6 or 8 cores per socket are available
    • Chips with 16, 32, 64, etc cores per chip are on their way
• Many believe we cannot scale past 8 cores per socket and continue to see performance improvement
  – Cache coherency adds overhead
  – Multi-core causes contention for shared resources
  – This contention causes performance degradation
    • In extreme cases, this performance degradation can cause execution times to increase when using multiple cores
Intel Core i7 (Nehalem)
Shared Resources

- Main memory
- Last Level Cache (LLC)
- Memory controller
- Pre-fetch hardware
Performance Issues

• Thread/process allocation
  – Threads accessing the same data should be within the same socket
  – Minimize cross-socket communication
• Hyperthreading
• Cache coherency
• False sharing
  – Causes thrashing
• Memory contention
Thread/process allocation

• If two threads share some data, being within the same socket can increase performance
  – However, cache coherency mechanisms can have a negative affect

• If two processes communicate with each other, communicating across sockets is more costly than within a socket
Hyperthreading

• Many modern microarchitectures offer hyperthreading
  – Allows for more threads than cores
    • Sun UltraSPARC – 8 threads per core
    • Intel i7 – 2 threads per core
  – Basically allows multiple pipelines. As long as the pipelines don’t converge, concurrency is possible
  – The hardware reports as if there are more cores than physically exist
Cache Coherency

• If two threads share data, even if that data is in cache there can be performance problems
  – If the data is only read, no problems
  – If one or more writes to the data, then updates must be reflected in both caches, resulting in contention for the memory bus and/or memory controller
False Sharing

- Generally, cores have private level one cache; sometimes private level two
- If each core stays in local cache, there should be little contention
- However, if two cores have the same cache line in their local cache, updates in one must be reflected in the other, even if the data being updated is different
- This can cause thrashing
Memory Contention

- If two cores have frequent private cache misses, then they must access the LLC. This causes contention for this resource, which slows response time.
- Contention for the LLC can also increase the eviction rate, leading to more frequent accesses to main memory.
- If two cores are competing for the same main memory, contention for the memory bus and memory controller results in slower response time.
- Aggressive prefetching can put additional pressure on the LLC.
Possible solutions/ideas

• Try to ensure that most frequently communicating threads reside on cores within the same socket
• If threads must share data, ensure they reside on cores within the same socket
  – If possible, minimize data sharing. Let each thread have its own private copy of data and aggregate when necessary
• Pad or control alignment to avoid false sharing
• Write local cache friendly code
  – Tiling can help
Possible solutions/ideas

• Make use of hyper threading when appropriate
  – This may include using different numbers of threads at different locations of the code
  – Separate tasks into compute intensive and memory intensive portions
  – Maybe have MPI operations handled by a separate thread/process (mostly integer instructions, should not interfere with compute intensive portions)

• Experiment to see what number of threads is best
  – It is possible that leaving one or more cores idle may result in better performance
Conclusions

• Getting the best performance out of your code is difficult for sequential, single core systems
• It has always been difficult to tune your parallel application
• The arrival of multi-core systems only makes the task that much harder
• There are techniques that can help lead to better performance
• There are tools available to help you out
INTRODUCTION TO PERFORMANCE ENGINEERING

Sameer Shende and Allen D. Malony
Performance Research Laboratory
University of Oregon
Performance Engineering

- Optimization process
- Effective use of performance technology

**Performance Technology**

- Experiment management
- Performance storage

**Performance Tuning**

- Performance Diagnosis
  - Hypotheses

**Performance Experimentation**

- Performance Observation
  - Characterization

**Performance Technology**

- Data mining
- Models
- Expert systems

**Performance Technology**

- Instrumentation
- Measurement
- Analysis
- Visualization
Performance Optimization Cycle

- Expose factors
- Collect performance data
- Calculate metrics
- Analyze results
- Visualize results
- Identify problems
- Tune performance

- Instrumentation
- Measurement
- Analysis
- Presentation
- Optimization
Parallel Performance Properties

• Parallel code performance is influenced by both sequential and parallel factors?

• Sequential factors
  – Computation and memory use
  – Input / output

• Parallel factors
  – Thread / process interactions
  – Communication and synchronization
Performance Observation

• Understanding performance requires observation of performance properties
• Performance tools and methodologies are primarily distinguished by what observations are made and how
  – What aspects of performance factors are seen
  – What performance data is obtained
• Tools and methods cover broad range
Metrics and Measurement

• Observability depends on measurement
• A metric represents a type of measured data
  – Count, time, hardware counters
• A measurement records performance data
  – Associates with program execution aspects
• Derived metrics are computed
  – Rates (e.g., flops)
• Metrics / measurements decided by need
Execution Time

- Wall-clock time
  - Based on realtime clock
- Virtual process time
  - Time when process is executing
    - User time and system time
  - Does not include time when process is stalled
- Parallel execution time
  - Runs whenever any parallel part is executing
  - Global time basis
Direct Performance Observation

- Execution *actions* exposed as *events*
  - In general, actions reflect some execution state
    - presence at a code location or change in data
    - occurrence in parallelism context (thread of execution)
  - Events encode actions for observation

- Observation is *direct*
  - Direct instrumentation of program code (probes)
  - Instrumentation invokes performance measurement
  - Event measurement = performance data + context

- Performance experiment
  - Actual events + performance measurements
Indirect Performance Observation

• Program code instrumentation is not used
• Performance is observed indirectly
  – Execution is interrupted
    • can be triggered by different events
  – Execution state is queried (sampled)
    • different performance data measured
  – *Event-based sampling* (EBS)
• Performance attribution is inferred
  – Determined by execution context (state)
  – Observation resolution determined by interrupt period
  – Performance data associated with context for period
Direct Observation: Events

- Event types
  - Interval events (begin/end events)
    - measures performance between begin and end
    - metrics monotonically increase
  - Atomic events
    - used to capture performance data state
- Code events
  - Routines, classes, templates
  - Statement-level blocks, loops
- User-defined events
  - Specified by the user
- Abstract mapping events
Direct Observation: Instrumentation

- Events defined by instrumentation access
- Instrumentation levels
  - Source code
  - Object code
  - Runtime system
  - Library code
  - Executable code
  - Operating system
- Different levels provide different information
- Different tools needed for each level
- Levels can have different granularity
Direct Observation: Techniques

- Static instrumentation
  - Program instrumented prior to execution

- Dynamic instrumentation
  - Program instrumented at runtime

- Manual and automatic mechanisms

- Tool required for automatic support
  - Source time: preprocessor, translator, compiler
  - Link time: wrapper library, preload
  - Execution time: binary rewrite, dynamic

- Advantages / disadvantages
Direct Observation: Mapping

- Associate performance data with high-level semantic abstractions
- Abstract events at user-level provide semantic context
Indirect Observation: Events/Triggers

• Events are actions external to program code
  – Timer countdown, HW counter overflow, …
  – Consequence of program execution
  – Event frequency determined by:
    • Type, setup, number enabled (exposed)

• Triggers used to invoke measurement tool
  – Traps when events occur (interrupt)
  – Associated with events
  – May add differentiation to events
Indirect Observation: Context

• When events trigger, execution context determined at time of trap (interrupt)
  – Access to PC from interrupt frame
  – Access to information about process/thread
  – Possible access to call stack
    • requires call stack unwinder

• Assumption is that the context was the same during the preceding period
  – Between successive triggers
  – Statistical approximation valid for long running programs
Direct / Indirect Comparison

- **Direct performance observation**
  - 😊 Measures performance data exactly
  - 😊 Links performance data with application events
  - 😞 Requires instrumentation of code
  - 😞 Measurement overhead can cause execution intrusion and possibly performance perturbation

- **Indirect performance observation**
  - 😊 Argued to have less overhead and intrusion
  - 😊 Can observe finer granularity
  - 😊 No code modification required (may need symbols)
  - 😞 Inexact measurement and attribution
Measurement Techniques

• When is measurement triggered?
  – External agent (indirect, asynchronous)
    • interrupts, hardware counter overflow, ...
  – Internal agent (direct, synchronous)
    • through code modification

• How are measurements made?
  – Profiling
    • summarizes performance data during execution
    • per process / thread and organized with respect to context
  – Tracing
    • trace record with performance data and timestamp
    • per process / thread
Measured Performance

• Counts
• Durations
• Communication costs
• Synchronization costs
• Memory use
• Hardware counts
• System calls
Critical issues

• Accuracy
  – Timing and counting accuracy depends on resolution
  – Any performance measurement generates overhead
    • Execution on performance measurement code
  – Measurement overhead can lead to intrusion
  – Intrusion can cause perturbation
    • alters program behavior

• Granularity
  – How many measurements are made
  – How much overhead per measurement

• Tradeoff (general wisdom)
  – Accuracy is inversely correlated with granularity
Profiling

• Recording of aggregated information
  – Counts, time, ...

• ... about program and system entities
  – Functions, loops, basic blocks, ...
  – Processes, threads

• Methods
  – Event-based sampling (indirect, statistical)
  – Direct measurement (deterministic)
Inclusive and Exclusive Profiles

- Performance with respect to code regions
- Exclusive measurements for region only
- Inclusive measurements includes child regions

```c
int foo()
{
    int a;
    a = a + 1;
    bar();
    a = a + 1;
    return a;
}
```
Flat and Callpath Profiles

• Static call graph
  – Shows all parent-child calling relationships in a program
• Dynamic call graph
  – Reflects actual execution time calling relationships
• Flat profile
  – Performance metrics for when event is active
  – Exclusive and inclusive
• Callpath profile
  – Performance metrics for calling path (event chain)
  – Differentiate performance with respect to program execution state
  – Exclusive and inclusive
Tracing Measurement

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

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

MONITOR

|   | master | worker | ...
|---|--------|--------|---
| 1 |        |        |   
| 2 |        |        |   
| 3 |        |        |   

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

LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes
Tracing Analysis and Visualization

<table>
<thead>
<tr>
<th></th>
<th>master</th>
<th>worker</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>master</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>worker</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A ENTER</th>
<th>B ENTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>58</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>A SEND</td>
<td>B</td>
</tr>
<tr>
<td>64</td>
<td>A EXIT</td>
<td>1</td>
</tr>
<tr>
<td>68</td>
<td>B RECV</td>
<td>A</td>
</tr>
<tr>
<td>69</td>
<td>B EXIT</td>
<td>2</td>
</tr>
</tbody>
</table>

![Diagram showing interactions between master and worker processes over time]
Trace Formats

• Different tools produce different formats
  – Differ by event types supported
  – Differ by ASCII and binary representations
    • Vampir Trace Format (VTF)
    • KOJAK (EPILOG)
    • Jumpshot (SLOG-2)
    • Paraver

• Open Trace Format (OTF)
  – Supports interoperation between tracing tools
Profiling / Tracing Comparison

• Profiling
  😊 Finite, bounded performance data size
  😊 Applicable to both direct and indirect methods
  😞 Loses time dimension (not entirely)
  🙁 Lacks ability to fully describe process interaction

• Tracing
  😊 Temporal and spatial dimension to performance data
  😊 Capture parallel dynamics and process interaction
  😞 Some inconsistencies with indirect methods
  🙁 Unbounded performance data size (large)
  🙁 Complex event buffering and clock synchronization
Performance Problem Solving Goals

• Answer questions at multiple levels of interest
  – High-level performance data spanning dimensions
    • machine, applications, code revisions, data sets
    • examine broad performance trends
  – Data from low-level measurements
    • use to predict application performance

• Discover general correlations
  – performance and features of external environment
  – Identify primary performance factors

• Benchmarking analysis for application prediction

• Workload analysis for machine assessment
Performance Analysis Questions

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

Build application \rightarrow Execute application \rightarrow Simple analysis feedback

72% Faster!

- Build information
- Environment / performance data
- Offline analysis

Performance database
Performance Data Management

- Performance diagnosis and optimization involves multiple performance experiments
- Support for common performance data management tasks augments tool use
  - Performance experiment data and metadata storage
  - Performance database and query
- What type of performance data should be stored?
  - Parallel profiles or parallel traces
  - Storage size will dictate
  - Experiment metadata helps in meta analysis tasks
- Serves tool integration objectives
Integration of metadata with each parallel profile
  - Separate information from performance data

Three ways to incorporate metadata
  - Measured hardware/system information
    - CPU speed, memory in GB, MPI node IDs, ...
  - Application instrumentation (application-specific)
    - Application parameters, input data, domain decomposition
    - Capture arbitrary name/value pair and save with experiment
  - Data management tools can read additional metadata
    - Compiler flags, submission scripts, input files, ...
    - Before or after execution

Enhances analysis capabilities
Performance Data Mining

• Conduct parallel performance analysis in a systematic, collaborative and reusable manner
  – Manage performance complexity and automate process
  – Discover performance relationship and properties
  – Multi-experiment performance analysis

• Data mining applied to parallel performance data
  – Comparative, clustering, correlation, characterization, ...
  – Large-scale performance data reduction

• Implement extensible analysis framework
  – Abstraction / automation of data mining operations
  – Interface to existing analysis and data mining tools
How to explain performance?

• Should not just redescribe performance results
• Should explain performance phenomena
  – What are the causes for performance observed?
  – What are the factors and how do they interrelate?
  – Performance analytics, forensics, and decision support
• Add *knowledge* to do more intelligent things
  – Automated analysis needs good informed feedback
  – Performance model generation requires interpretation
• Performance knowledge discovery framework
  – Integrating meta-information
  – Knowledge-based performance problem solving
You have to capture these...

Performance Knowledge

...to understand this

Context Knowledge

Source Code
Build Environment
Run Environment

Execution

Performance Result
Performance Optimization Process

• Performance characterization
  – Identify major performance contributors
  – Identify sources of performance inefficiency
  – Utilize timing and hardware measures

• Performance diagnosis (Performance Debugging)
  – Look for conditions of performance problems
  – Determine if conditions are met and their severity
  – What and where are the performance bottlenecks

• Performance tuning
  – Focus on dominant performance contributors
  – Eliminate main performance bottlenecks
HANDS-ON:
LIVE-DVD AND NPB-MPI BT

POINT Team
The NAS Parallel Benchmark suite (sample MPI version)
- Available from http://www.nas.nasa.gov/Software/NPB
- 9 benchmarks (7 in Fortran77, 2 in C)
- Configurable for various sizes & classes

Move into the NPB3.3-MPI root directory

```
% cd tutorial/NPB3.3-MPI; ls
BT/     CG/     DT/     EP/     FT/     IS/     LU/     MG/     SP/
bin/      common/  config/  Makefile      README  README.tutorial  sys/
```

Subdirectories contain source code for each benchmark
- plus additional configuration and common code

The provided distribution has already been configured for the tutorial, such that it's ready to “make” one or more of the benchmarks and install them into a (tool-specific) “bin” subdirectory
Building an NPB-MPI benchmark

• Type “make” for instructions

```
% make
=========================================                      
= NAS Parallel Benchmarks 3.3 = 
= MPI/F77/C =                          
=========================================                      

To make a NAS benchmark type

    make <benchmark-name> NPROCS=<number> CLASS=<class>

To make a set of benchmarks, create the file config/suite.def according to the instructions in config/suite.def.template and type

    make suite
```

**************************************************************
* Custom build configuration is specified in config/make.def *
* Suggested tutorial benchmark specification:             *
*    make bt CLASS=W NPROCS=16                           *
**************************************************************

LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes
Building an NPB benchmark

- Specify the benchmark configuration
  - benchmark name: \texttt{bt}, \texttt{cg}, \texttt{dt}, \texttt{ep}, \texttt{ft}, \texttt{is}, \texttt{lu}, \texttt{mg}, \texttt{sp}
  - the number of MPI processes: \texttt{NPROC=16}
  - the benchmark class (S, W, A, B, C, D, E): \texttt{CLASS=W}

% make \texttt{bt NPROCS=16 CLASS=W}
cd BT; make NPROCS=16 CLASS=W SUBTYPE= VERSION=
gmake: Entering directory 'BT'
cd ..../sys; cc -o setparams setparams.c
../sys/setparams bt 16 W
mpif77 -c -O bt.f
...
cd ../common; mpif77 -c -O timers.f
mpif77 -c -O btio.f
mpif77 -O -o ..../bin/bt_W.16
bt.o make_set.o initialize.o exact_solution.o exact_rhs.o 
set_constants.o adi.o define.o copy_faces.o rhs.o solve_subs.o 
x_solve.o y_solve.o z_solve.o add.o error.o verify.o setup_mpi.o 
../common/print_results.o ../common/timers.o btio.o
Built executable ../bin/bt_W.16
gmake: Leaving directory 'BT'
NPB-MPI BT (Block Tridiagonal solver)

- **What does it do?**
  - Solves a discretized version of unsteady, compressible Navier-Stokes equations in three spatial dimensions
  - Performs 200 time-steps on a regular 3-dimensional grid
- **Can be configured to include various forms of parallel I/O**
  - e.g., MPI collective file I/O: SUBTYPE=full
- **Implemented in 20 or so Fortran77 source modules**

- **Needs a square number of processes**
  - 16 should be reasonable (decrease to 9 or 4, if necessary)
    - excess processes idle when run with more than compiled
    - don’t expect to see speed-up when run on a laptop!
  - **bt_W** should run in around 5 to 12 seconds
  - **bt_A** should take around 16-20x longer (90-100 seconds)
BT-MPI reference execution

• Launch as an MPI application

```
% cd bin; mpiexec -np 16 ./bt_W.16
NAS Parallel Benchmarks 3.3 -- BT Benchmark
Size: 24x 24x 24
Iterations: 200  dt: 0.0008000
Number of active processes: 16

Time step 1
Time step 20
Time step 40
Time step 60
...
Time step 160
Time step 180
Time step 200
Verification Successful

BT Benchmark Completed.
Time in seconds = 4.70
```

Hint: save the benchmark output (or note the run time) to be able to refer to it later
Tutorial exercise steps

• The tutorial steps are similar and repeated for each tool
• Edit config/make.def to adjust build configuration
  – Modify specification of compiler/linker: MPIF77
• Make clean and build new tool-specific executable

```
% make clean
% make bt CLASS=W NPROCS=16
Built executable ../bin.$(TOOL)/bt_W.16
```

• Change to the directory containing the new executable before running it with the desired tool configuration

```
% cd bin.$(TOOL)
% export ...
% mpiexec -np 16 bt_W.16
```
NPB-MPI/BT: config/make.def

# SITE- AND/OR PLATFORM-SPECIFIC DEFINITIONS
#-------------------------------------------------------------------------------------
# Items in this file may need to be changed for each platform.
#-------------------------------------------------------------------------------------

# The Fortran compiler used for MPI programs
#-------------------------------------------------------------------------------------

MPIF77 = mpif77

# Alternative variants to perform instrumentation
#-------------------------------------------------------------------------------------

#MPIF77 = marmotf77
#MPIF77 = tau_f90.sh
#MPIF77 = scalasca -instrument mpif77
#MPIF77 = vtf77 -vt:f77 mpif77

# PREP is a generic preposition macro for instrumentation preparation
#-------------------------------------------------------------------------------------

#MPIF77 = $(PREP) mpif77

# This links MPI Fortran programs; usually the same as ${MPIF77}
FLINK = $(MPIF77)

...
PAPI

David Cronk
Innovative Computing Lab
University of Tennessee, Knoxville
Outline

• Introduction
• PAPI Utilities
• An Example
• Some PAPI counters
• PAPI and Multi-core
• Component PAPI – The New Wave
Hardware performance counters available on most modern microprocessors can provide insight into:

1. Whole program timing
2. Cache behaviors
3. Branch behaviors
4. Memory and resource access patterns
5. Pipeline stalls
6. Floating point efficiency
7. Instructions per cycle

Hardware counter information can be obtained with:

1. Subroutine or basic block resolution
2. Process or thread attribution
What’s PAPI?

• Middleware to provide a consistent programming interface for the performance counter hardware found in most major micro-processors.

• Countable events are defined in two ways:
  – Platform-neutral *preset* events
  – Platform-dependent native events

• Presets can be *derived* from multiple *native events*

• All events are referenced by name and collected in EventSets for sampling

• Events can be *multiplexed* if counters are limited

• Statistical sampling implemented by:
  – Hardware overflow if supported by the platform
  – Software overflow with timer driven sampling
Where’s PAPI

- PAPI runs on most modern processors and operating systems of interest to HPC:
  - IBM POWER / AIX / Linux
  - Blue Gene / L / P...
  - Intel Pentium, Core2, Core i7, Atom / Linux
  - Intel Itanium / Linux
  - AMD Athlon, Opteron / Linux
  - Cray XT(n) / CLE
  - Altix, Sparc, Niagara...
$ utils/papi_cost -h
This is the PAPI cost program.
It computes min / max / mean / std. deviation for PAPI start/stop pairs; for PAPI reads, and for PAPI_accums.
Usage:

cost [options] [parameters]
cost TESTS_QUIET

Options:

- **-b BINS** set the number of bins for the graphical distribution of costs. Default: 100
- **-d** show a graphical distribution of costs
- **-h** print this help message
- **-s** show number of iterations above the first 10 std deviations
- **-t THRESHOLD** set the threshold for the number of iterations. Default: 100,000
PAPI Utilities: \textit{papi\_cost}

$ \texttt{utils/papi\_cost}$

Cost of execution for PAPI start/stop and PAPI read.
This test takes a while. Please be patient...
Performing start/stop test...

Total cost for \texttt{PAPI\_start/stop(2 counters)} over 1000000 iterations
\begin{itemize}
  \item min cycles : 63
  \item max cycles : 17991
  \item mean cycles : 69.000000
  \item std deviation: 34.035263
\end{itemize}
Performing start/stop test...

Performing read test...

Total cost for \texttt{PAPI\_read(2 counters)} over 1000000 iterations
\begin{itemize}
  \item min cycles : 288
  \item max cycles : 102429
  \item mean cycles : 301.000000
  \item std deviation: 144.694053
\end{itemize}
\begin{flushright}
  \texttt{cost.c PASSED}
\end{flushright}
PAPI Utilities: \textit{papi\_cost}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{cost_distribution_profile.png}
\caption{Cost distribution profile}
\end{figure}
PAPI Utilities: `papi_avail`

$ utils/papi_avail -h
Usage: utils/papi_avail [options]
Options:

General command options:
- `-a, --avail` Display only available preset events
- `-d, --detail` Display detailed information about all preset events
- `-e EVENTNAME` Display detail information about specified preset or native event
- `-h, --help` Print this help message

Event filtering options:
- `--br` Display branch related PAPI preset events
- `--cache` Display cache related PAPI preset events
- `--cnd` Display conditional PAPI preset events
- `--fp` Display Floating Point related PAPI preset events
- `--ins` Display instruction related PAPI preset events
- `--idl` Display Stalled or Idle PAPI preset events
- `--l1` Display level 1 cache related PAPI preset events
- `--l2` Display level 2 cache related PAPI preset events
- `--l3` Display level 3 cache related PAPI preset events
- `--mem` Display memory related PAPI preset events
- `--msc` Display miscellaneous PAPI preset events
- `--tlb` Display Translation Lookaside Buffer PAPI preset events

This program provides information about PAPI preset and native events.
PAPI preset event filters can be combined in a logical OR.
$ utils/papi_avail
Available events and hardware information.

-----------------------------------------------
PAPI Version : 4.0.0.0
Vendor string and code : GenuineIntel (1)
Model string and code : Intel Core i7 (21)
CPU Revision : 5.000000
CPUID Info : Family: 6 Model: 26 Stepping: 5
CPU Megahertz : 2926.000000
CPU Clock Megahertz : 2926
Hdw Threads per core : 1
Cores per Socket : 4
NUMA Nodes : 2
CPU's per Node : 4
Total CPU's : 8
Number Hardware Counters : 7
Max Multiplex Counters : 32
-----------------------------------------------

The following correspond to fields in the PAPI_event_info_t structure.

[MORE...]
The following correspond to fields in the PAPI_event_info_t structure.

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Avail</th>
<th>Deriv</th>
<th>Description (Note)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_L1_DCM</td>
<td>0x80000000</td>
<td>No</td>
<td>No</td>
<td>Level 1 data cache misses</td>
</tr>
<tr>
<td>PAPI_L1_ICM</td>
<td>0x80000001</td>
<td>Yes</td>
<td>No</td>
<td>Level 1 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_L2_DCM</td>
<td>0x80000002</td>
<td>Yes</td>
<td>Yes</td>
<td>Level 2 data cache misses</td>
</tr>
<tr>
<td>PAPI_VEC_SP</td>
<td>0x80000069</td>
<td>Yes</td>
<td>No</td>
<td>Single precision vector/SIMD instr.</td>
</tr>
<tr>
<td>PAPI_VEC_DP</td>
<td>0x8000006a</td>
<td>Yes</td>
<td>No</td>
<td>Double precision vector/SIMD instr.</td>
</tr>
</tbody>
</table>

Of 107 possible events, 34 are available, of which 9 are derived.
PAPI Utilities: `papi_avail`

```bash
$ utils/papi_avail -e PAPI_FP_OPS
[...]

The following correspond to fields in the PAPI_event_info_t structure.

Event name: PAPI_FP_OPS
Event Code: 0x80000066
Number of Native Events: 2
Short Description: |FP operations|
Long Description: |Floating point operations|
Developer's Notes: |
Derived Type: |DERIVED_ADD|
Postfix Processing String: |
Native Code[0]: 0x4000801b |FP_COMP_OPS_EXE:SSE_SINGLE_PRECISION|
Number of Register Values: 2
Register[ 0]: 0x0000000f |Event Selector|
Register[ 1]: 0x00004010 |Event Code|
Native Event Description: |Floating point computational micro-ops, masks:SSE* FP single precision Uops|

Native Code[1]: 0x4000081b |FP_COMP_OPS_EXE:SSE_DOUBLE_PRECISION|
Number of Register Values: 2
Register[ 0]: 0x0000000f |Event Selector|
Register[ 1]: 0x00008010 |Event Code|
Native Event Description: |Floating point computational micro-ops, masks:SSE* FP double precision Uops|
```
UNIX> utils/papi_native_avail
Available native events and hardware information.

[...]
<table>
<thead>
<tr>
<th>Event Code</th>
<th>Symbol</th>
<th>Long Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x40000010</td>
<td>BR_INST_EXEC</td>
<td>Branch instructions executed</td>
</tr>
<tr>
<td></td>
<td>:ANY</td>
<td>Branch instructions executed</td>
</tr>
<tr>
<td></td>
<td>:COND</td>
<td>Conditional branch instructions executed</td>
</tr>
<tr>
<td></td>
<td>:DIRECT</td>
<td>Unconditional branches executed</td>
</tr>
<tr>
<td></td>
<td>:DIRECT_NEAR_CALL</td>
<td>Unconditional call branches executed</td>
</tr>
<tr>
<td></td>
<td>:INDIRECT_NEAR_CALL</td>
<td>Indirect call branches executed</td>
</tr>
<tr>
<td></td>
<td>:INDIRECT_NON_CALL</td>
<td>Indirect non call branches executed</td>
</tr>
<tr>
<td></td>
<td>:NEAR_CALL</td>
<td>Retired near call instructions (Precise Event)</td>
</tr>
</tbody>
</table>

[...]
UNIX> utils/papi_native_avail -e DATA_CACHE_REFILLS
Available native events and hardware information.
--------------------------------------------------------------------------------
[...]
--------------------------------------------------------------------------------
The following correspond to fields in the PAPI_event_info_t structure.

Event name: DATA_CACHE_REFILLS
Event Code: 0x4000000b
Number of Register Values: 2
Description: Data Cache Refills from L2 or System

Register[ 0]: 0x0000000f |Event Selector|
Register[ 1]: 0x00000042 |Event Code|

Unit Masks:
Mask Info: SYSTEM|Refill from System|
Register[ 0]: 0x0000000f |Event Selector|
Register[ 1]: 0x00000142 |Event Code|
Mask Info: L2_SHARED|Shared-state line from L2|
Register[ 0]: 0x0000000f |Event Selector|
Register[ 1]: 0x00000242 |Event Code|
Mask Info: L2_EXCLUSIVE|Exclusive-state line from L2|
Register[ 0]: 0x0000000f |Event Selector|
Register[ 1]: 0x00000442 |Event Code|
$ utils/papi_event_chooser PRESET PAPI_FP_OPS
Event Chooser: Available events which can be added with given events.
--------------------------------------------------------------------------------

Name        Code    Deriv Description (Note)
PAPI_L1_DCM  0x80000000  No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  No   Level 1 instruction cache misses
PAPI_L2_ICM  0x80000003  No   Level 2 instruction cache misses
PAPI_L1_DCA  0x80000040  No   Level 1 data cache accesses
PAPI_L2_DCR  0x80000044  No   Level 2 data cache reads
PAPI_L2_DCW  0x80000047  No   Level 2 data cache writes
PAPI_L1_ICA  0x8000004c  No   Level 1 instruction cache accesses
PAPI_L2_ICA  0x8000004d  No   Level 2 instruction cache accesses
PAPI_L2_TCA  0x80000059  No   Level 2 total cache accesses
PAPI_L2_TCW  0x8000005f  No   Level 2 total cache writes
PAPI_FML_INS 0x80000061  No   Floating point multiply instructions
PAPI_FDV_INS 0x80000063  No   Floating point divide instructions
--------------------------------------------------------------------------------
Total events reported: 34
event_chooser.c             PASSED
PAPI Utilities: `papi_event_chooser`

```bash
$ utils/papi_event_chooser PRESET PAPI_FP_OPS PAPI_L1_DCM
Event Chooser: Available events which can be added with given events.
--------------------------------------------------------------------------------
[...]
--------------------------------------------------------------------------------

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Deriv</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_TOT_INS</td>
<td>0x80000032</td>
<td>No</td>
<td>Instructions completed</td>
</tr>
<tr>
<td>PAPI_TOT_CYC</td>
<td>0x8000003b</td>
<td>No</td>
<td>Total cycles</td>
</tr>
</tbody>
</table>

Total events reported: 2

event_chooser.c Passed
```
$ utils/papi_event_chooser NATIVE RESOURCE_STALLS:LD_ST X87_OPS_RETIRED
   INSTRUCTIONS_RETIRED
[...]
-----------------------------------------------------------------

UNHALTED_CORE_CYCLES  0x40000000

| count core clock cycles whenever the clock signal on the specific core is running 
  (not halted). Alias to event CPU_CLK_UNHALTED:CORE_P |
| Register Value[0]: 0x20003    Event Selector |
| Register Value[1]: 0x3c       Event Code |

-----------------------------------------------------------------

UNHALTED_REFERENCE_CYCLES  0x40000002

| Unhalted reference cycles. Alias to event CPU_CLK_UNHALTED:REF |
| Register Value[0]: 0x40000    Event Selector |
| Register Value[1]: 0x13c      Event Code |

-----------------------------------------------------------------

CPU_CLK_UNHALTED  0x40000028

| Core cycles when core is not halted |
| Register Value[0]: 0x60000    Event Selector |
| Register Value[1]: 0x3c       Event Code |

  0x40001028 :CORE_P  |Core cycles when core is not halted |
  0x40008028 :NO_OTHER |Bus cycles when core is active and the other is halted |

-----------------------------------------------------------------

Total events reported: 3

event_chooser.c       PASSED
$ papi_command_line PAPI_FP_OPS
Successfully added: PAPI_FP_OPS

PAPI_FP_OPS : 100000000

Verification: None.
This utility lets you add events from the command line interface to see if they work.
command_line.c PASSED

$ papi_command_line PAPI_FP_OPS PAPI_L1_DCA
Successfully added: PAPI_FP_OPS
Successfully added: PAPI_L1_DCA

PAPI_FP_OPS : 100000000
PAPI_L1_DCA : 120034404

Verification: None.
This utility lets you add events from the command line interface to see if they work.
command_line.c PASSED
The Code

```c
#define ROWS 1000 // Number of rows in each matrix
#define COLUMNS 1000 // Number of columns in each matrix

void classic_matmul()
{
    // Multiply the two matrices
    int i, j, k;
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLUMNS; j++) {
            float sum = 0.0;
            for (k = 0; k < COLUMNS; k++) {
                sum += matrix_a[i][k] * matrix_b[k][j];
            }
            matrix_c[i][j] = sum;
        }
    }
    // Note that the nesting of the innermost loops
    // has been changed. The index variables j and k
    // change the most frequently and the access
    // pattern through the operand matrices is
    // sequential using a small stride (one.) This
    // change improves access to memory data through
    // the data cache. Data translation lookaside
    // buffer (DTLB) behavior is also improved.
}

void interchanged_matmul()
{
    // Multiply the two matrices
    int i, j, k;
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLUMNS; j++) {
            float sum = 0.0;
            for (k = 0; k < COLUMNS; k++) {
                sum += matrix_a[i][k] * matrix_b[k][j];
            }
            matrix_c[i][j] = sum;
        }
    }
}
```

LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes
IPC – instructions per cycle

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Classic mat_mul</th>
<th>Reordered mat_mul</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PAPI_IPC Test (PAPI_ipc)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Real time</td>
<td>13.6093 sec</td>
<td>2.9796 sec</td>
</tr>
<tr>
<td>Processor time</td>
<td>13.5359 sec</td>
<td>2.9556 sec</td>
</tr>
<tr>
<td>IPC</td>
<td>0.3697</td>
<td>1.6936</td>
</tr>
<tr>
<td>Instructions</td>
<td>9007035063</td>
<td>9009011383</td>
</tr>
<tr>
<td><strong>High Level IPC Test (PAPI_{start,stop}_counters)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Real time</td>
<td>13.6106 sec</td>
<td>2.9762 sec</td>
</tr>
<tr>
<td>IPC</td>
<td>0.3697</td>
<td>1.6939</td>
</tr>
<tr>
<td>PAPI_TOT_CYC</td>
<td>24362605525</td>
<td>5318626915</td>
</tr>
<tr>
<td>PAPI_TOT_INS</td>
<td>9007034503</td>
<td>9009011245</td>
</tr>
<tr>
<td><strong>Low Level IPC Test (PAPI low level calls)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Real time</td>
<td>13.6113 sec</td>
<td>2.9772 sec</td>
</tr>
<tr>
<td>IPC</td>
<td>0.3697</td>
<td>1.6933</td>
</tr>
<tr>
<td>PAPI_TOT_CYC</td>
<td>24362750167</td>
<td>5320395138</td>
</tr>
<tr>
<td>PAPI_TOT_INS</td>
<td>9007034381</td>
<td>9009011130</td>
</tr>
</tbody>
</table>

- All three PAPI methods consistent
- Roughly 460% improvement in reordered code
Data Cache Access

Data Cache Misses can be considered in 3 categories:

• **Compulsory**: Occurs on first reference to a data item.
  – Prefetching

• **Capacity**: Occurs when the working set exceeds the cache capacity.
  – Spatial locality
  – Smaller working set (blocking/tiling algorithms)

• **Conflict**: Occurs when a data item is referenced after the cache line containing the item was evicted earlier.
  – Temporal locality
  – Data layout; memory access patterns
L1 Data Cache Access

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Classic mat_mul</th>
<th>Reordered mat_mul</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA_CACHE_ACCESSSES</td>
<td>2002807841</td>
<td>3008528961</td>
</tr>
<tr>
<td>DATA_CACHE_REFILLS:L2_MODIFIED:L2_OWNED:L2_EXCLUSIVE:L2_SHARED</td>
<td>205968263</td>
<td>60716301</td>
</tr>
<tr>
<td>DATA_CACHE_REFILLS_FROM_SYSTEM:MODIFIED:OWNED:EXCLUSIVE:SHARED</td>
<td>61970925</td>
<td>1950282</td>
</tr>
</tbody>
</table>

---

Data Cache Request Rate
- 0.2224 req/inst
- 0.3339 req/inst

Data Cache Miss Rate
- 0.0298 miss/inst
- 0.0070 miss/inst

Data Cache Miss Ratio
- 0.1338 miss/req
- 0.0208 miss/req

- Two techniques
  - First uses native events
  - Second uses PAPI presets only

- ~50% more requests from reordered code

- 1/4 as many misses per instruction
- 1/6 as many misses per request
### Branching

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Classic mat_mul</th>
<th>Reordered mat_mul</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_BR_INS</td>
<td>1001028240</td>
<td>1001006987</td>
</tr>
<tr>
<td>PAPI_BR_MSP</td>
<td>1028256</td>
<td>1006984</td>
</tr>
<tr>
<td>PAPI_BR_TKN</td>
<td>1000027233</td>
<td>1000005980</td>
</tr>
<tr>
<td>Branch Rate</td>
<td>0.1111 br/inst</td>
<td>0.1111 br/inst</td>
</tr>
<tr>
<td>Branch Miss Rate</td>
<td>0.0001 miss/inst</td>
<td>0.0001 miss/inst</td>
</tr>
<tr>
<td>Branch Miss Ratio</td>
<td>0.0010 miss/br</td>
<td>0.0010 miss/br</td>
</tr>
<tr>
<td>Branch Taken Rate</td>
<td>0.1110 tkn/inst</td>
<td>0.1110 tkn/inst</td>
</tr>
<tr>
<td>Branch Taken Ratio</td>
<td>0.9990 tkn/br</td>
<td>0.9990 tkn/br</td>
</tr>
<tr>
<td>Instr / Branch</td>
<td>8.9978 inst/br</td>
<td>8.9999 inst/br</td>
</tr>
</tbody>
</table>

- **Uses all PAPI Presets!**
- **Branch behavior nearly identical in both codes**
- **Roughly 1 branch every 9 instructions**
- **1 miss per 1000 branches (remember ROWS?)**
- **Branching and branch misses can be reduced with loop unrolling, loop fusion and function in-lining.**
Performance Measurement Categories

• Efficiency
  – Instructions per cycle (IPC)
  – Memory bandwidth
• Caches
  – Data cache misses and miss ratio
  – Instruction cache misses and miss ratio
• Lower level cache misses and miss ratio
• Translation lookaside buffers (TLB)
  – Data TLB misses and miss ratio
  – Instruction TLB misses and miss ratio
• Control transfers
  – Branch mispredictions
  – Near return mispredictions
• Special cases
  – Unaligned data access
  – Floating point operations
  – Floating point exceptions
The Multicore Dilemma

• Multicore is the (near term) future of Petascale computing

• Minimizing Resource contention is key
  – Memory bandwidth
  – Cache sharing & collisions
  – Bus and other resource contention

• Current tools don’t support first-person counting of shared events

• Current architectures don’t encourage first-person counting of shared events
Current “State of the Art”

• Counter support for shared resources is broken
  – Every vendor has a different approach
  – Often 3rd person, not 1st person
  – Counts often polluted by other cores
  – No exclusive reservation of shared counter resources
  – No migration of events with tasks

• PAPI research is underway to address this
Multicore counter support

• Intel Core2 Duos:
  – SELF/ANY
  – L2 shared cache, bus, snoop
  – 39 events/≈140 are core qualified
• AMD Opteron Shanghai
  – 4 L3 shared cache events:
    • READ_REQUEST_TO_L3_CACHE
    • L3_CACHE_MISSES
    • L3_FILLS_CAUSED_BY_L2_EVictions
    • L3_EVictions
  – First 3 are qualified per core:
    • CORE0, CORE1, CORE2, CORE3
    • Only 1 core can (safely) count these events at a time
Multicore counter support (cont.)

- Intel i7 (Nehalem)
  - 7 counters per core
    - 3 fixed, 4 programmable
  - 8 counters shared on-chip
    - Require global, not process
    - Not currently supported by PAPI
  - 117 native events available to PAPI users
  - 28 PAPI PRESET events
Extending PAPI beyond the CPU

• PAPI has historically targeted on on-processor performance counters
• Several categories of off-processor counters exist
  – network interfaces: Myrinet, Infiniband, GigE
  – memory interfaces: Cray SeaStar, Gemini
  – thermal and power interfaces: ACPI, lm-sensors
  – accelerators?
• CHALLENGE:
  – Extend the PAPI interface to address multiple counter domains
  – Preserve the PAPI calling semantics, ease of use, and platform independence for existing applications
Component PAPI Goals

• Support simultaneous access to on- and off-
  processor counters
• Isolate hardware dependent code in separable
  ‘component’ modules
• 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
Multi Component Measurements

• HPCC HPL benchmark on Opteron with 3 performance metrics:
  – FLOPS; Temperature; Network Sends/Receives
  • Temperature is from an on-chip thermal diode
For more information

- PAPI Website: [http://icl.cs.utk.edu/papi/](http://icl.cs.utk.edu/papi/)
  - Software
  - Release notes
  - Documentation
  - Links to tools that use PAPI
  - Mailing/discussion lists
PERFSUITE

Rui Liu and Rick Kufrin
National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign
I. Overview
II. PerfSuite Tools
III. Hands-On Exercise with LiveDVD
IV. PerfSuite Libraries / APIs
V. PerfSuite and Java
VI. Current Issues and Status
PerfSuite Background

• Active development since Linux clusters were adopted at NCSA in 2001
  – No tools then available for CPU beyond gprof
• UI/NCSA Open Source license approved 2003
• Targeted to users of all levels of expertise
  – The intent is to provide an easy-to-use mechanism for measuring application performance, and to expose problem areas for further exploration
• Low measurement overhead also important
• Close collaboration/sharing with UTK from outset
PerfSuite and POINT

- NSF SDCI program enables maintenance, enhancement, interoperability, and integration
- PerfSuite fills the Level 0 (entry) role for performance measurement within POINT
  - Simple (in most cases, no code change/relink needed)
  - Low overhead (default case is nearly non-intrusive)
  - Limited information... but still very useful and in some cases sufficient
- PerfSuite has never attempted to supply sophisticated graphical/visualization or data management capabilities
  - By partnering with TAU, advanced graphical tools come as a natural by-product
  - PerfDMF infrastructure is mature, and well-suited for importing data collected by PerfSuite
- POINT’s application and training thrust (PSC) will expose to wider user base
What Does PerfSuite Provide?

• Overall hardware performance event counts for all or a portion of your application
• Profiling with statistical sampling using either time- or event-based triggers
  – Generalization of the approach used by gprof
• Flexible XML-based output along with various techniques for display, manipulation, combining, transformation
• Information about processor in use (type, cache/TLB specs, etc) – this “metadata” is stored along with measurement
• Functionality available through easy-to-use command line tool that can be used with most applications without need for modification
• Also available through several libraries for finer control
PerfSuite and XML

• In PerfSuite, nearly all data (input, output, configuration, etc) is represented as XML (eXtensible Markup Language) documents
• This provides the ability to manipulate & transform the data in many ways using standard software / skills
• Machine-independent (no binary files)
  – ... opens the data up to the user
• There are numerous high-quality XML-aware libraries available from either compiled or interpreted languages that can make it easy to transform the data for your needs
  – New in PS version 1.0.0: Java API for accessing data
• The structured, well-defined nature of XML makes it natural for import into DB-driven infrastructure such as PerfDMF
PerfSuite Counter-Related Software

• Four performance counter-related utilities:
  – `psconfig` - configure / select performance events
  – `psinv` - query events and machine information
  – `psrun` - generate raw counter or statistical profiling data from an unmodified binary
  – `psprocess` - pre- and post-process data

• Three libraries (shared and static, serial and threaded)
  – `libperfsuite` – the “core” library that can be used standalone and will be built regardless of the availability of other software
  – `libpshwpc` – HardWare Performance Counter library, also built regardless of other software. Without counter support, will only perform time-based profiling through profil() or interval timers
  – `libpshwpc_mpi` – a convenience library based on the MPI standard PMPI interface

• *PerfSuite does not require kernel patches*
psinv: Processor Inventory

- Lists information about the characteristics of the computer
- This same information is also stored in PerfSuite XML output and is useful for later generating derived metrics (or for remembering where you ran your program!)
- x86/x86-64 version also shows processor features and descriptions
- Lists available hardware performance events

```
titan:~3% psinv -v
System Information -
Processors: 2
Total Memory: 2007.16 MB
System Page Size: 16.00 KB

Processor Information -
Vendor: Intel
Processor family: IPF
Model (Type): Itanium
Revision: 6
Clock Speed: 800.136 MHz

Cache and TLB Information -
Cache levels: 3
Caches/TLBs: 7

Cache Details -
Level 1:
  Type: Data
  Size: 16 KB
  Line size: 32 bytes
  Associativity: 4-way set associative

  Type: Instruction
  Size: 16 KB
  Line size: 32 bytes
  Associativity: 4-way set associative
```
**psinv: PAPI Event Summary**

```
% psinv -p
PAPI Standard Event Information -
Standard events: 43
Non-derived events: 26
Derived events: 17

PAPI Standard Event Details -
Non-derived:
- PAPI_BR_INS: Branch instructions
- PAPI_BR_PRC: Conditional branch instructions correctly predicted
- PAPI_L1_DCA: Level 1 data cache accesses
- PAPI_L1_DCM: Level 1 data cache misses
- PAPI_L1_ICM: Level 1 instruction cache misses
- PAPI_L2_DCA: Level 2 data cache accesses
- PAPI_L2_DCR: Level 2 data cache reads
- PAPI_L2_DCW: Level 2 data cache writes
- PAPI_L2_ICM: Level 2 instruction cache misses
- PAPI_L2_STM: Level 2 store misses
- PAPI_L2_TCM: Level 2 cache misses

Derived:
- PAPI_BR_MSP: Conditional branch instructions mispredicted
- PAPI_BR_NTK: Conditional branch instructions not taken
- PAPI_BR_TKN: Conditional branch instructions taken
- PAPI_FLOPS: Floating point instructions per second
- PAPI_FP_INS: Floating point instructions
- PAPI_L1_DCH: Level 1 data cache hits
```
psrun: Performance Measurement

- Hardware performance counting and profiling with unmodified dynamically-linked executables
- Available for x86, x86-64, and ia64
- POSIX threads support
- Automatic multiplexing
- Can be used with MPI and OpenMP
- Optionally collects resource usage
- Supports all PAPI standard and CPU-native events
- Input/Output = XML documents (can request plain text)
psrun “Cookbook”

# First, be sure to set all paths properly (can do in .cshrc/.profile)

% set PSDIR=/opt/perfsuite
% source $PSDIR/bin/psenv.csh

# Use psrun on your program to generate the data,
# then use psprocess to produce an HTML file (default is plain text)

% psrun myprog
% psprocess --html myprog.12345.xml > myprog.html

# Take a look at the results

% your-web-browser myprog.html

# Second run, but this time profiling instead of counting

% psrun -C -c papi_profile_cycles.xml myprog
% psprocess -e myprog myprog.67890.xml
psprocess: Post-process Results

- This style of output is customizable by you.
- By default, the information it contains and its visual appearance are based on PerfSuite-provided defaults, but these can be easily replaced to suit your preference.
- This output is generated by psprocess using XML Transformations. The stylesheet is in the share/perfsuite/xml/pshwpc subdirectory, with a “xsl” file extension.
# PerfSuite Hardware Performance Summary Report

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>1.0</td>
</tr>
<tr>
<td>Created</td>
<td>Mon Dec 30 11:31:53 AM Central Standard Time 2002</td>
</tr>
<tr>
<td>Generator</td>
<td>psprocess 0.5</td>
</tr>
<tr>
<td>XML Source</td>
<td>/u/ncsa/anyuser/performance/psrun-ia64.xml</td>
</tr>
</tbody>
</table>

## Execution Information

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>Sun Dec 15 21:01:20 2002</td>
</tr>
<tr>
<td>Host</td>
<td>user01</td>
</tr>
</tbody>
</table>

## Processor and System Information

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node CPUs</td>
<td>2</td>
</tr>
<tr>
<td>Vendor</td>
<td>Intel</td>
</tr>
<tr>
<td>Family</td>
<td>IPF</td>
</tr>
<tr>
<td>Model</td>
<td>Itanium</td>
</tr>
<tr>
<td>CPU Revision</td>
<td>6</td>
</tr>
<tr>
<td>Clock (MHz)</td>
<td>800.136</td>
</tr>
<tr>
<td>Memory (MB)</td>
<td>2007.16</td>
</tr>
<tr>
<td>Pagesize (KB)</td>
<td>16</td>
</tr>
</tbody>
</table>
The reports (text or HTML) generated by psprocess have several sections, covering:

- Report creation details
- Run details
- Machine information
- Raw counter listings
- Counter explanations and index
- Derived metrics
- Run annotation defined by you

Derived metrics are evaluated at run-time and can be extended (text mode only)
<table>
<thead>
<tr>
<th>Index Description</th>
<th>Counter Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Conditional branch instructions mispredicted</td>
<td>4831072449</td>
</tr>
<tr>
<td>4 Floating point instructions</td>
<td>86124489172</td>
</tr>
<tr>
<td>5 Total cycles</td>
<td>594547754568</td>
</tr>
<tr>
<td>6 Instructions completed</td>
<td>1049339828741</td>
</tr>
</tbody>
</table>

Statistics

| Graduated instructions per cycle                                                  | 1.765                  |
| Graduated floating point instructions per cycle                                   | 0.145                  |
| Level 3 cache miss ratio (data)                                                   | 0.957                  |
| Bandwidth used to level 3 cache (MB/s)                                            | 385.087                |
| % cycles with no instruction issue                                                | 10.410                 |
| % cycles stalled on memory access                                                | 43.139                 |
| MFLOPS (cycles)                                                                  | 115.905                |
| MFLOPS (wallclock)                                                              | 114.441                |
Configuring Your Measurement

• All PerfSuite runs are configured according to an XML document that specifies what is to be measured
  – if you don’t specify a custom configuration, a default is used
• A custom configuration document (file) is supplied in one of two ways
  – psrun option “-c filename”
  – PS_HWPC_CONFIG environment variable, which can be set to filename
• Creating new configuration files is easy, and can be done with either a text editor or the tool psconfig
Example Configuration

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<ps_hwpc_eventlist class="PAPI">
    <ps_hwpc_event type="preset" name="PAPI_BR_MSP" />
    <ps_hwpc_event type="preset" name="PAPI_BR_PRC" />
    <ps_hwpc_event type="preset" name="PAPI_BR_TKN" />
    <ps_hwpc_event type="preset" name="PAPI_FP_INS" />
    <ps_hwpc_event type="preset" name="PAPI_TOT_CYC" />
    <ps_hwpc_event type="preset" name="PAPI_TOT_INS" />
    <ps_hwpc_event type="preset" name="PAPI_L1_DCA" />
    <ps_hwpc_event type="preset" name="PAPI_L1_DCM" />
    <ps_hwpc_event type="preset" name="PAPI_L1_TCM" />
    <ps_hwpc_event type="preset" name="PAPI_L2_DCA" />
    <ps_hwpc_event type="preset" name="PAPI_L2_DCM" />
</ps_hwpc_eventlist>
```

- You can edit this file like any text file
- The XML document root element “ps_hwpc_eventlist” indicates this configuration is to be used for aggregate counting (not profiling)
Using Processor “Native Events”

• It’s easy to work with native events in addition to PAPI standard events by modifying the configuration file slightly.
• Instead of using the XML attributes type="preset" name="PAPI_EVENTNAME", use the attribute type="native" and enclose the event name as the content of the element.
• Can be used with profiling configurations.

<ps_hwpc_event type="native">NOPS RETIRED</ps_hwpc_event>
<ps_hwpc_event type="native">BACK END BUBBLE ALL</ps_hwpc_event>
Configuring for Profiling

- Setting up for profiling is similar to counting - all you have to do is modify the XML configuration document:
- The XML document “root element” is now `<ps_hwpc_profile>`, not `<ps_hwpc_eventlist>`
- You can supply an optional “threshold”, or sampling rate
- Only one event is allowed in the document
- `psconfig` does not yet support profiling, need to edit by hand

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<ps_hwpc_profile class="PAPI">
  <ps_hwpc_event type="preset"
    name="PAPI_BR_MSP" threshold="100000" />
</ps_hwpc_profile>
```
**Profile Information**

<table>
<thead>
<tr>
<th>Class</th>
<th>PAPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>3.6.2</td>
</tr>
<tr>
<td>Event</td>
<td>PAPI TOT_CYC (Total cycles)</td>
</tr>
<tr>
<td>Period</td>
<td>100000</td>
</tr>
<tr>
<td>Samples</td>
<td>200471</td>
</tr>
<tr>
<td>Domain</td>
<td>user</td>
</tr>
<tr>
<td>Run Time</td>
<td>27.19 (seconds)</td>
</tr>
<tr>
<td>Min Self %</td>
<td>(all)</td>
</tr>
</tbody>
</table>

**Module Summary**

<table>
<thead>
<tr>
<th>Samples</th>
<th>Self %</th>
<th>Total %</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>186068</td>
<td>92.82%</td>
<td>92.82%</td>
<td>/home/rkufrin/apps/aspcg/aspcg</td>
</tr>
<tr>
<td>14182</td>
<td>7.07%</td>
<td>99.89%</td>
<td>/opt/intel/cc/9.0/lib/libguide.so</td>
</tr>
<tr>
<td>187</td>
<td>0.09%</td>
<td>99.98%</td>
<td>/lib/ld-2.3.6.so</td>
</tr>
<tr>
<td>18</td>
<td>0.01%</td>
<td>99.99%</td>
<td>/lib/tls/libc-2.3.6.so</td>
</tr>
<tr>
<td>15</td>
<td>0.01%</td>
<td>100.00%</td>
<td>/lib/tls/libpthread-2.3.6.so</td>
</tr>
<tr>
<td>1</td>
<td>0.00%</td>
<td>100.00%</td>
<td>/tmp/perfsuite/lib/libpsrun_r.so.0.0.1</td>
</tr>
</tbody>
</table>

**File Summary**

<table>
<thead>
<tr>
<th>Samples</th>
<th>Self %</th>
<th>Total %</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>154346</td>
<td>76.99%</td>
<td>76.99%</td>
<td>/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f</td>
</tr>
<tr>
<td>14506</td>
<td>7.24%</td>
<td>84.23%</td>
<td>/home/rkufrin/apps/aspcg/cg3_blk.f</td>
</tr>
<tr>
<td>14505</td>
<td>7.24%</td>
<td>91.46%</td>
<td>??</td>
</tr>
<tr>
<td>10185</td>
<td>5.08%</td>
<td>96.54%</td>
<td>/home/rkufrin/apps/aspcg/matrixvec2d_blk3.f</td>
</tr>
<tr>
<td>3042</td>
<td>1.52%</td>
<td>98.06%</td>
<td>/home/rkufrin/apps/aspcg/dot_prod2d_blk3.f</td>
</tr>
<tr>
<td>2366</td>
<td>1.18%</td>
<td>99.24%</td>
<td>/home/rkufrin/apps/aspcg/add_exchange2d_blk3.f</td>
</tr>
<tr>
<td>834</td>
<td>0.42%</td>
<td>99.66%</td>
<td>/home/rkufrin/apps/aspcg/main3.f</td>
</tr>
<tr>
<td>687</td>
<td>0.34%</td>
<td>100.00%</td>
<td>/home/rkufrin/apps/aspcg/cs_jac2d_blk3.f</td>
</tr>
</tbody>
</table>
## Text-based profiles, cont’d

### Function Summary

<table>
<thead>
<tr>
<th>Samples</th>
<th>Self %</th>
<th>Total %</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>154346</td>
<td>76.99%</td>
<td>76.99%</td>
<td>pc_jac2d_blk3</td>
</tr>
<tr>
<td>14506</td>
<td>7.24%</td>
<td>84.23%</td>
<td>cg3_blk</td>
</tr>
<tr>
<td>10185</td>
<td>5.08%</td>
<td>89.31%</td>
<td>matxvec2d_blk3</td>
</tr>
<tr>
<td>6937</td>
<td>3.46%</td>
<td>92.77%</td>
<td>__kmp_x86_pause</td>
</tr>
<tr>
<td>4711</td>
<td>2.35%</td>
<td>95.12%</td>
<td>__kmp_wait_sleep</td>
</tr>
<tr>
<td>3042</td>
<td>1.52%</td>
<td>96.64%</td>
<td>dot_prod2d_blk3</td>
</tr>
<tr>
<td>2366</td>
<td>1.18%</td>
<td>97.82%</td>
<td>add_exchange2d_blk3</td>
</tr>
</tbody>
</table>

### Function:File:Line Summary

<table>
<thead>
<tr>
<th>Samples</th>
<th>Self %</th>
<th>Total %</th>
<th>Function:File:Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>39063</td>
<td>19.49%</td>
<td>19.49%</td>
<td>pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:20</td>
</tr>
<tr>
<td>24134</td>
<td>12.04%</td>
<td>31.52%</td>
<td>pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:19</td>
</tr>
<tr>
<td>15626</td>
<td>7.79%</td>
<td>39.32%</td>
<td>pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:33</td>
</tr>
<tr>
<td>15028</td>
<td>7.50%</td>
<td>46.82%</td>
<td>pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:24</td>
</tr>
<tr>
<td>13878</td>
<td>6.92%</td>
<td>53.74%</td>
<td>pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:24</td>
</tr>
<tr>
<td>11880</td>
<td>5.93%</td>
<td>59.66%</td>
<td>pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:31</td>
</tr>
<tr>
<td>8896</td>
<td>4.44%</td>
<td>64.10%</td>
<td>pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:22</td>
</tr>
<tr>
<td>7863</td>
<td>3.92%</td>
<td>68.02%</td>
<td>matxvec2d_blk3:/home/rkufrin/apps/aspcg/matxvec2d_blk3.f:19</td>
</tr>
<tr>
<td>7145</td>
<td>3.56%</td>
<td>71.59%</td>
<td>pc_jac2d_blk3:/home/rkufrin/apps/aspcg/pc_jac2d_blk3.f:32</td>
</tr>
</tbody>
</table>
psconfig: Graphical Configuration

- Graphical user interface makes it easy to select events
- Can read in or write out valid XML documents to be used by psrun
- Provides text description of events with mouse click
- Searching capabilities
- Profiling not yet supported

LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes
Searching Events with psconfig

- Selecting “Edit”, “Search Events…” brings up a window like this that allows you to search events for keywords.
- Can restrict the search to only events available on your computer.
- The search is based on the event’s description, not its standard event name (PAPI_TOT_CYC).

Search events for: cache

Event Available

Event Not Available

PAPI_L1_DCA: Level 1 data cache accesses
PAPI_L1_DCH: Level 1 data cache hits
PAPI_L1_DCM: Level 1 data cache misses
PAPI_L1_DCR: Level 1 data cache reads
PAPI_L1_DCW: Level 1 data cache writes
PAPI_L1_ICA: Level 1 instruction cache accesses
PAPI_L1_ICH: Level 1 instruction cache hits
PAPI_L1_ICM: Level 1 instruction cache misses
PAPI_L1_ICR: Level 1 instruction cache reads
PAPI_L1_ICW: Level 1 instruction cache writes
PAPI_L1_TCA: Level 1 total cache accesses
PAPI_L1_TCH: Level 1 total cache hits
PAPI_L1_TCM: Level 1 total cache misses
PAPI_L1_TCR: Level 1 total cache reads
PAPI_L1_TCW: Level 1 total cache writes
PAPI_L2_DCA: Level 2 data cache accesses
PAPI_L2_DCH: Level 2 data cache hits
PAPI_L2_DCM: Level 2 data cache misses
PAPI_L2_DCR: Level 2 data cache reads
PAPI_L2_DCW: Level 2 data cache writes
PAPI_L2_ICA: Level 2 instruction cache accesses
PAPI_L2_ICH: Level 2 instruction cache hits
PAPI_L2_ICM: Level 2 instruction cache misses
PAPI_L2_ICR: Level 2 instruction cache reads
PAPI_L2_ICW: Level 2 instruction cache writes
PAPI_L2_TCA: Level 2 total cache accesses
PAPI_L2_TCH: Level 2 total cache hits
Browsing Predefined Event Configurations

- Selecting “File”, “Default Hardware Event Configurations...” brings up the directory with pre-selected configuration documents
- Opening one of them will show you which events will be used
- You can base custom configuration files using these as a start
psrun: Advanced Use

• psrun supports a few options that can be useful in working with shared or distributed memory programs:

• `-p / --pthreads`
  - uses a POSIX thread-aware variant of the library that captures thread creation and measures performance of each, depositing the results in an XML document with the thread ID embedded

• `-f / --fork`
  - monitors child processes that are created. Not enabled by default

• `-a / --annotate`
  - inserts an XML “element” with a user-supplied annotation (text)
psprocess: Advanced Use

- **psprocess** is meant to be a “generic” processor for different XML document types generated by PerfSuite. For hardware counting, the most common type is `<hwpcreport>`

- Individual documents can be combined into a “multi-document” with the option `-c` / `--combine`. With hardware counter data, `psprocess` summarizes the information contained in them with descriptive statistics (mean, max, min, sum, stddev)

- `-s LIST` is a very useful option to be used with profiling runs. LIST is a comma-separated list of modules, files, functions, lines used to limit the amount of output

- `-t THRESHOLD` is also helpful in limiting the output of profiling runs. **THRESHOLD** is a number that specifies the minimum % of samples required for a given entry to be displayed. Example: “-t 2” means “don’t show me anything that didn’t account for at least 2% of the samples collected”

- `psprocess help output` (“-h”) lists all available options and types
psprocess: User-defined Metrics

- **psprocess** allows the creation of user-defined metrics
- User-defined metrics are stored in a file of your choice that contains expression templates (reminiscent of MathML)
- Select via PS_HWPC_METRICS environment variable or “psprocess -m”

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<psmetrics class="hwpc">
  <metric namespace="PAPI" type="ratio">
    <name>PS_RATIO_GINS_CYC</name>
    <description lang="en_US">Graduated instructions per cycle</description>
    <definition>
      <apply>
        <divide>
          <ci>PAPI_TOT_INS</ci>
          <ci>PAPI_TOT_CYC</ci>
        </divide>
      </apply>
    </definition>
  </metric>
</psmetrics>
```
PerfSuite Environment Variables

- PS_HWPC: “off” or “on”, controls whether measurement takes place at all (for API)
- PS_HWPC_CONFIG: set to the name of the XML event file created with psconfig or “by hand”. A default is used if not set
- PS_HWPC_FILE: controls the prefix of the XML output document (default is the name of the command being measured)
- PS_HWPC_ANNOTATION - adds an arbitrary “note” to the XML output
- PS_HWPC_DOMAIN: controls whether counting at user or system level (or both)
- PS_HWPC_THRESHOLD: sets threshold for profiling
- PS_HWPC_FORMAT: “text” or “xml”, controls whether output is in an XML document or plain text (similar to a psprocess report)
- PSRUN_DOFORK: if set (to anything), monitors child processes also

“psrun –h” will show a complete listing of recognized variables
PerfSuite’s XML Document Hierarchy

• The basic per-thread XML document that is created by PerfSuite is called an “hwpcreport”
  – These are in either “counting” or “profiling” mode
• Logical collections of the basic documents can be grouped together using the “-c” (“combine”) option to psprocess. The result is called a “multihwpcreport”
  – This is where the notion of a parallel run of arbitrary scale enters and can be applied to shared- or distributed-memory runs
  – Subsequent processing with psprocess recognizes these “multi” documents and provides different statistics, more appropriate for parallel runs
• The basic concept is extensible to further logical collections of one or more runs, threads, tasks, etc
PerfSuite Profiles with ParaProf and Cube3

TAU’s ParaProf can display PerfSuite profiles after being mapped to source and stored as XML (psprocess –x)

Development version of psprocess produces Cube XML files directly
LiveDVD Hands-On Exercise

• The goals of this exercise are to gain experience with:
  1. querying your computer with `psinv`
  2. collecting performance data with `psrun`
  3. examining the results with `psprocess`
  4. using PerfSuite data with other POINT/VI-HPS tools

• Your environment is already set up properly to access PerfSuite after booting the LiveDVD, so no need to “source” the “psenv” file

• We will use one of the NAS Parallel Benchmarks (NPB-MPI, BT), which are already on the LiveDVD
Querying Your System with `psinv`

- Open a new shell (terminal), and run the `psinv` command, asking it to display everything it can by supplying the option `–v`
- You will probably want to pipe the output into a pager like `less`

```
% psinv –v | less
```

- Output is broken into several logical sections:

<table>
<thead>
<tr>
<th>System/OS</th>
<th>Cache/TLB</th>
<th>PAPI Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Type/Speed</td>
<td>Processor Features</td>
<td></td>
</tr>
</tbody>
</table>
Building and Running NPB-MPI/BT

• Position yourself in the source directory for the NPB-MPI benchmark, build the program, and run it without performance tools

• Symbols are not included in the build by default (and they will be needed for profiling), so we request their inclusion via FFLAGS

```
% cd tutorial
% make bt NPROCS=4 CLASS=W FFLAGS="-O -g"
% cd bin
% mpirun -np 4 ./bt_W.4
```
Obtaining Hardware Event Counts

• Next, run the application again, but this time include `psrun` in the launch command:

```
% mpirun -np 4 psrun ./bt_W.4
```

• Examine the data from one or more of the resulting XML documents with `psprocess`:

```
% psprocess bt_W.4.NNNNN.localhost.xml
```

• Combine all tasks’ data into a single document and view the aggregate statistics:

```
% psprocess -c -o combined.xml bt_W*.xml
% psprocess combined.xml
```

This is a process ID
Modifying Your Configuration

• You may have to adjust the default configuration file that `psrun` uses, to correspond to the type of CPU you have. You can check the default configuration to be used by executing `psrun -h`

• Use environment variable `PS_HWPC_CONFIG` or supply the option `-c` to `psrun`

• Likely values that may work for your system:
  – “papi3_core.xml” (Intel Core)
  – “papi3_p4.xml” (Intel Pentium 4)
  – “papi3_p6.xml” (Intel Pentium Pro, II, III)

• If you find that you need to adjust the configuration:

  % mpirun -np 4 psrun -C -c conf.xml ./bt_W.4
Obtaining A Profile

• Profiling the application involves using `psrun` again, but using a different configuration file. We will also add the option `-o`, which allows us to explicitly name the output files:

```bash
% mpirun -np 4 psrun -o bt_profile -C \ -c papi_profile_cycles.xml ./bt_W.4
```

• Once again, use `psprocess` to view:

```bash
% psprocess bt_profile.NNNN.xml
```
Using POINT and VI-HPS Tools

• There is no built-in support within PerfSuite for working with profiles generated from parallel applications as a whole. However, you can use TAU’s ParaProf:

```bash
% psprocess -o pstau.xml -x --glob 'bt_profile*.xml'
% paraprof -f psrun pstau.xml
```

• ... and Scalasca’s Cube:

```bash
% psprocess -o pscube.xml --cube --glob 'bt_profile*.xml'
% cube3 pscube.xml
```

The `--glob` option provides pattern-based filename matching without exceeding shell limits. *Use quotes to protect from shell interpretation/expansion!*
PerfSuite Library Access (API)

• All of the functionality is also available from within your program (C/C++/Fortran) through a small API
• Same XML documents are read, same XML documents are written, small additional functionality
• Why would you want to use this?
  – Primarily to gain finer control over where measurements are taken in your program. For example, you might defer measurement until program initialization has completed
• For complex uses, you are probably better off using an “industrial-strength” performance library
• The intent of the API is to “abstract out” the process of performance measurement to a very high level
libperfsuite: Core Library

- This library is available regardless of the presence of hardware counter support.
- Small number of useful routines callable from either C or FORTRAN (use “PSF_” instead of “ps_” with FORTRAN).

```c
int ps_cpuspeed         (double *mhz);
int ps_cpuusage         (pid_t pid, ps_time_t *utime,
                          ps_time_t *stime);
int ps_dmemusage        (float *total_mb, float *used_mb,
                          float *free_mb);
int ps_memusage         (pid_t pid, float *vsize_mb,
                          float *rss_mb);
int ps_procstat         (pid_t pid,
                          ps_procstat_t *p);
int ps_RTC              (unsigned long long *rtcval);
int ps_RTCinit          (void);
const char *ps_strerror (int code);
```

- `#include <perfsuite.h>` (or “fperfsuite.h”)

LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes
libpshwpc: Performance Collection API

C / C++

ps_hwpc_init (void)
ps_hwpc_start (void)
ps_hwpc_read (long long *values)
ps_hwpc_suspend (void)
ps_hwpc_stop (char *prefix)
ps_hwpc_shutdown (void)

• Call “init” once, call “start”, “read” and “suspend” as many times as you like. Call “stop” (supplying a file name prefix of your choice) to get the performance data XML document

• Optionally, call “shutdown”

• Example programs demonstrating use are installed in PerfSuite “examples” subdirectory

• Additional routines
  ps_hwpc_numevents() and
  ps_hwpc_eventnames() allow querying current configuration

Fortran

call psf_hwpc_init (ierr)
call psf_hwpc_start (ierr)
call psf_hwpc_read (integer*8 values,ierr)
call psf_hwpc_suspend (ierr)
call psf_hwpc_stop (prefix, ierr)
call psf_hwpc_shutdown (ierr)
FORTRAN API Example

```fortran
include 'fperfsuite.h'
call PSF_hwpc_init(ierr)
call PSF_hwpc_start(ierr)
do j = 1, n
    do i = 1, m
        do k = 1, l
            c(i,j) = c(i,j) + a(i,k)*b(k,j)
        end do
    end do
end do
call PSF_hwpc_stop('perf', ierr)
call PSF_hwpc_shutdown(ierr)
```

% ifort -c matmult.f -I/opt/perfsuite/include
% ifort matmult.o -L/opt/perfsuite/lib/intel -L/opt/papi/lib
    -lpshwpc -lperfsuite -lpapi
Java-based Performance Measurement

- PerfSuite 1.0.0 provides the capability of monitoring unmodified Java applications in a manner similar to the `psrun` command.
- Implemented using the core PerfSuite C libraries and the Java Virtual Machine Tool Interface (JVMTI).
- Use involves a slightly modified command line:
  ```java
  java -agentlib:psjrun MyClass
  ```
- Results are contained in XML documents that can be post-processed in the usual “PerfSuite way”.
psprocess (Java version)

- **psprocess** is being reimplemented in Java (to date, has been in Tcl). Requires Java 1.5 or newer
- **Motivation:** long-term development and maintenance
- **Retains most features of Tcl version, but some infrequently-used options are deprecated:**
  - `-h/--html` generate HTML format output
  - `-P/--peak` specify peak MFLOP rate
  - `--vmon` generate VProf-format output
- **Developed using Java XML API and metric calculation API previously released**
Java/Tcl Versions Co-exist

• Plan is to have a transition period, during which both Tcl and Java versions are installed (Tcl remains default at present)
• The `psprocess` command supports new options to select which version to use: "--tcl" and "--java"
• For feature stability, continuity, and ease of debugging if issues arise
• We encourage use of the Java version, feedback, bug reports, etc
  – This will help accelerate full movement to the Java version, while the older Tcl prototype is deprecated
PerfSuite Java Performance API

• PerfSuite supports a new Java-based API for performance measurement from within a Java application

• Analogous to the PerfSuite C/Fortran libraries, and follows a similar model, but in Java style

```java
import org.perfsuite.hwpc.*;

PS_hwpcThreaded hwpc = PS_hwpcThreaded.getInstance();
hwpc.start();
computeSomething();
hwpc.stop("perfdatal);
```

Subclass of abstract base class PS_hwpc for threaded programs. Serial equivalent is PS_hwpcSerial
PerfSuite XML Java API

• Provides programmatic access to the information contained in PerfSuite reports through Java

• Includes detailed Javadoc documentation:
  – $PREFIX/share/perfsuite/doc/javadoc

• Supports all PerfSuite XML reports; parses all elements in them and places the data in Java objects that can be accessed via “getter” methods

$ JARFILE=$PREFIX/share/perfsuite/javalib/perfsuite.jar
$ javac -classpath $JARFILE MyClass.java
$ java -classpath $JARFILE:. MyClass <arguments>
Example Use of the PS XML Java API

```java
import java.util.*;
import org.perfsuite.xml.*;

// The “newInstance” method is used to parse any supported type of XML
// document that PerfSuite generates. It accepts the name of the
// file to parse and a flag to indicate whether XML validation is done.
PS_Report report0 = PS_Report.newInstance (filename, false);

// Use “instanceof” to determine the type of report that was parsed.
// This example shows how to handle a report with event totals.
if (report0 instanceof PS_HwpcCountingReport) {
    PS_HwpcCountingReport report = (PS_HwpcCountingReport) report0;
    Map<String, PS_HwpcEvent> eventMap = report.getEvents();
    for (Iterator it = eventMap.entrySet().iterator(); it.hasNext(); ) {
        Map.Entry entry = (Map.Entry) it.next();
        PS_HwpcEvent event = (PS_HwpcEvent) entry.getValue();
        System.out.println ("Event: " + event.getName() +
        ", Count: " + event.getCount() +
        ", Type: " + event.getType() +
        ", Derived: " + event.getDerived());
    }
}
```
PerfSuite Java Metrics API

• Provides calculation of metrics for a given PerfSuite counting report, and a user metric definition object created by the PS XML Java API

• The calculated metrics are stored in a map where the keys are the metric descriptions and the associated values are the metric values

• Supports internationalization and localization in the description strings of the metrics (as in the Tcl version)

• Includes detailed Javadoc documentation
Example Use of the PS Java Metrics API

```java
import org.perfsuite.xml.*;
import org.perfsuite.metrics.PS_MetricCalculator;

// Use the PerfSuite XML API to parse data
PS_Report report = PS_Report.newInstance (xmlFileName, false);

// First create a metric definition object which is then used to
// obtain derived metric calculations, given an input report.
// Note: “metDef” can be reused with additional input reports
PS_MetricDefinition metDef =
    new PS_MetricDefinition (metricFileName, true);
Map<String,Double> metResult =
    PS_MetricCalculator.calculate (report, metDef);

System.out.println ("Calculated metric values:");
for (Map.Entry<String,Double> entry : metResult.entrySet()) {
    System.out.println (String.format
        ("%-64s: %15.3f", entry.getKey(), entry.getValue()));
}
```
Issues at Higher Scales of Parallelism

• How well can PerfSuite be expected to scale to extreme levels of parallelism?
  – All monitoring is contained within the context of a single core/processor/thread. No communication or synchronization required between threads as measurement proceeds, so not impacted
  – Currently, results/output are written to local disk files; PerfSuite enforces serialized output from multithreaded programs to minimize filesystem contention. Not an issue to date, but warrants rethinking
  – PC-to-source code mapping (for profiling runs) is currently done through the `psprocess` command, and can consume significant times for large programs at high levels of parallelism

• While PerfSuite has been used successfully on core counts of hundreds to thousands, further work needs to be done to improve existing barriers to scalability. These issues are a key piece of work ongoing under the POINT collaboration
Recent and Upcoming in PerfSuite

• Current stable release is version 0.6.2
  – Provides nearly all of the features covered in this presentation
• Version 1.0 is now in alpha release state
  – Alpha releases are for incorporating new features, major modifications
  – Much new functionality and reengineering on the roadmap:
    • *Highlights of current alpha*:
      – New Java API for user access to PerfSuite XML documents (do what you like with the data PerfSuite collects)
      – New Java API for derived metric calculation
      – New support for Cube3 output
      – New support for collecting performance data from Java applications (using either an API or a new JVMTI agent)
      – New Java-based implementation of `psprocess`
    • *For later releases*:
      – Enhanced profiling capabilities, including substantial reduction in memory requirements for profiling runs
      – Improved scalability of profiling output and post-processing for parallel runs
  – Current and potential users’ feedback, bug reports, encouraged
For More Information and Downloads

• PerfSuite web sites:
TAU PERFORMANCE SYSTEM

Sameer Shende
Alan Morris, Wyatt Spear, Scott Biersdorff
Performance Research Lab

Allen D. Malony, Suzanne Millstein
Department of Computer and Information Science
University of Oregon
TAU Performance System®

• Tuning and Analysis Utilities (15+ year project)
• Performance problem solving framework for HPC
  – Integrated, scalable, flexible, portable
  – Target all parallel programming / execution paradigms
• Integrated performance toolkit (open source)
  – Instrumentation, measurement, analysis, visualization
  – Widely-ported performance profiling / tracing system
  – Performance data management and data mining
• Broad application use (NSF, DOE, DOD, …)
LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes
TAU Instrumentation / Measurement

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
- I/O profiles
- profile sampling

Tracing

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

Performance data sources

- timing
- hardware counters
- system counters
- kernel

OS and runtime system modules

- threading
- interrupts
- runtime system
- I/O
Direct Performance Observation

• Execution actions of interest exposed as events
  – In general, actions reflect some execution state
    • presence at a code location or change in data
    • occurrence in parallelism context (thread of execution)
  – Events encode actions for performance system to observe

• Observation is direct
  – Direct instrumentation of program (system) code (probes)
  – Instrumentation invokes performance measurement
  – Event measurement: performance data, meta-data, context

• Performance experiment
  – Actual events + performance measurements

• Contrast with (indirect) event-based sampling
TAU Instrumentation Approach

• Support for standard program events
  – Routines, classes and templates
  – Statement-level blocks
  – Begin/End events (Interval events)

• Support for user-defined events
  – Begin/End events specified by user
  – Atomic events (e.g., size of memory allocated/freed)
  – Flexible selection of event statistics

• Provides static events and dynamic events
• Enables “semantic” mapping
• Specification of event groups (aggregation, selection)
• Instrumentation optimization
TAU Event Interface

- Events have a type, a group association, and a name
- TAU events names are character strings
  - Powerful way to encode event information
  - Inefficient way to communicate each event occurrence
- TAU maps a new event name to an event ID
  - Done when event is first encountered (get event handle)
  - Event ID is used for subsequent event occurrences
  - Assigning a uniform event ID a priori is problematic
- A new event is identified by a new event name in TAU
  - Can create new event names at runtime
  - Allows for dynamic events (TAU renames events)
  - Allows for context-based, parameter-based, phase events
TAU Instrumentation Mechanisms

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

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

• Executable code
  – Binary and dynamic instrumentation (Dyninst)
  – Virtual machine instrumentation (e.g., Java using JVMPI)

• TAU_COMPILER to automate instrumentation process
Automatic Source-level Instrumentation

TAU source analyzer

Parsed program

Instrumentation specification file

tau_instrumentor

Application source

Instrumented source
Program Database Toolkit (PDT)

Applicacion / Library

C / C++ parser

Fortran parser F77/90/95

C / C++ IL analyzer

IL

Fortran IL analyzer

Program Database Files

DUCTAPE

TAU instrumentor

Automatic source instrumentation

LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes
MPI Wrapper Interposition Library

• Uses standard MPI Profiling Interface
  – Provides name shifted interface
    • MPI_Send = PMPI_Send
    • Weak bindings

• Create TAU instrumented MPI library
  – Interpose between MPI and TAU
  – Done during program link
    • -lmpi replaced by –lTauMpi –lpmpi –lmpi
  – No change to the source code!
  – Just re-link application to generate performance data
MPI Shared Library Instrumentation

- Interpose the MPI wrapper library for applications that have already been compiled
  - Avoid re-compilation or re-linking
- Requires shared library MPI
  - Uses LD_PRELOAD for Linux
  - On AIX use MPI_EUILIB / MPI_EUILIBPATH
  - Does not work on XT3
- Approach will work with other shared libraries
- Use TAU tauex
  - `% mpirun -np 4 tauex a.out`
Selective Instrumentation File

- Specify a list of events to exclude or include
- # is a wildcard in a routine name

```
BEGIN_EXCLUDE_LIST
  Foo
  Bar
  D#EMM
END_EXCLUDE_LIST

BEGIN_INCLUDE_LIST
  int main(int, char **)
  F1
  F3
END_INCLUDE_LIST
```
Selective Instrumentation File

- Optionally specify a list of files
- * and ? may be used as wildcard characters

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

- User instrumentation commands
  - Placed in INSTRUMENT section
  - Routine entry/exit
  - Arbitrary code insertion
  - Outer-loop level instrumentation

BEGIN_INSTRUMENT_SECTION
loops file="foo.f90" routine="matrix#"
memory file="foo.f90" routine="#"
io routine="matrix#"
[static/dynamic] phase routine="MULTIPLY"
dynamic [phase/timer] name="foo" file="foo.cpp" line=22 to line=35
file="foo.f90" line = 123 code = " print *, \" Inside foo\"
exit routine = “int foo()” code = "cout <<\"exiting foo\"<<endl;”
END_INSTRUMENT_SECTION
TAU Measurement Approach

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

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

• Robust timing and hardware performance support

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

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

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

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

• Flat profiles
  – Metric (e.g., time) spent in an event (callgraph nodes)
  – Exclusive/inclusive, # of calls, child calls
• Callpath profiles (Calldepth profiles)
  – Time spent along a calling path (edges in callgraph)
  – “main=> f1 => f2 => MPI_Send” (event name)
  – TAU_CALLPATH_DEPTH environment variable
• Phase profiles
  – Flat profiles under a phase (nested phases are allowed)
  – Default “main” phase
  – Supports static or dynamic (per-iteration) phases
  – Phase profiles may be generated from full callpath profiles in paraprof by choosing events as phases
TAU Analysis

Using POINT Performance Tools to Understand and Optimize Multicore Codes
Performance Analysis

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

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

Parser and Importers
- Basic Analysis + Derived Data
- Internal Representation
- Profile Data

PerfDMF
- Call Graphs
- Histograms
- Call Trees
- Bar Charts
- Comparative Displays
- Text Displays

ParaProf
- Vis Package
  - 3D Displays
- Scripting Interface
  - Jython
Performance Data Management

• Provide an open, flexible framework to support common data management tasks
  – Foster multi-experiment performance evaluation
• Extensible toolkit to promote integration and reuse across available performance tools (PerfDMF)
  – Originally designed to address critical TAU requirements
  – Supported profile formats:
    TAU, CUBE (Scalasca), HPC Toolkit (Rice), HPM Toolkit (IBM), gprof, mpiP, psrun (PerfSuite), Open|SpeedShop, ...
  – Supported DBMS:
    PostgreSQL, MySQL, Oracle, DB2, Derby/Cloudscape
  – Profile query and analysis API
• Reference implementation for PERI-DB project
PerfDMF Architecture

TAU Performance System

Profile metadata

raw profiles

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

XML document

formatted profile data

Performance Analysis Programs

scalability analysis
ParaProf
cluster analysis

Query and Analysis Toolkit

Java PerfDMF API

SQL (PostgreSQL, MySQL, DB2, Oracle)

Data Mining (Weka)

Statistics (R / Omega)
Metadata Collection

• Integration of XML metadata for each parallel profile
• Three ways to incorporate metadata
  – Measured hardware/system information (TAU, PERI-DB)
    • CPU speed, memory in GB, MPI node IDs, ...
  – Application instrumentation (application-specific)
    • TAU_Metadata() used to insert any name/value pair
    • Application parameters, input data, domain decomposition
  – PerfDMF data management tools can incorporate an XML file of additional metadata
    • Compiler flags, submission scripts, input files, ...
• Metadata can be imported from / exported to PERI-DB
Performance Data Mining / Analytics

• Conduct systematic and scalable analysis process
  – Multi-experiment performance analysis
  – Support automation, collaboration, and reuse

• Performance knowledge discovery framework
  – Data mining analysis applied to parallel performance data
    • comparative, clustering, correlation, dimension reduction, ...
  – Use the existing TAU infrastructure

• PerfExplorer v1 performance data mining framework
  – Multiple experiments and parametric studies
  – Integrate available statistics and data mining packages
    • Weka, R, Matlab / Octave
  – Apply data mining operations in interactive enviroment
How to explain performance?

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

You have to capture these...

Context Knowledge

Source Code
Build Environment
Run Environment

Execution

Performance Knowledge

Performance Problems
Application
Machine

...to understand this

Performance Result
PerfExplorer v2 – Requirements

• Component-based analysis process
  – Analysis operations implemented as modules
  – Linked together in analysis process and workflow
• Scripting
  – Provides process/workflow development and automation
• Metadata input, management, and access
• Inference engine
  – Reasoning about causes of performance phenomena
  – Analysis knowledge captured in expert rules
• Persistence of intermediate analysis results
• Provenance
  – Provides historical record of analysis results
PerfExplorer v2 Architecture
### Parallel Profile Analysis – pprof

![Image of a Profiling Tool]

**Buffers Files Tools Edit Search Mule Help**

**Node** 0 **Context** 0 **Thread** 0:

<table>
<thead>
<tr>
<th>Time</th>
<th>Exclusive</th>
<th>Inclusive</th>
<th>#Call</th>
<th>#Subrs</th>
<th>Inclusive Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0</td>
<td>1</td>
<td>3:11.293</td>
<td>1</td>
<td>15</td>
<td>191293269 applu</td>
</tr>
<tr>
<td>99.6</td>
<td>3.667</td>
<td>3:10.463</td>
<td>3</td>
<td>37517</td>
<td>63087325 bcast_inputs</td>
</tr>
<tr>
<td>67.1</td>
<td>6.491</td>
<td>2:08.326</td>
<td>37200</td>
<td>37200</td>
<td>3450 exchange_1</td>
</tr>
<tr>
<td>44.5</td>
<td>6.451</td>
<td>1:25.189</td>
<td>5300</td>
<td>18600</td>
<td>9157 bts</td>
</tr>
<tr>
<td>41.0</td>
<td>1:18.436</td>
<td>1:18.436</td>
<td>18600</td>
<td>0</td>
<td>4217 MPI_Recv()</td>
</tr>
<tr>
<td>29.5</td>
<td>5.778</td>
<td>55.407</td>
<td>9300</td>
<td>18600</td>
<td>6058 bits</td>
</tr>
<tr>
<td>26.2</td>
<td>50.142</td>
<td>50.142</td>
<td>16304</td>
<td>0</td>
<td>2611 MPI_Send()</td>
</tr>
<tr>
<td>16.2</td>
<td>24.451</td>
<td>31.031</td>
<td>301</td>
<td>602</td>
<td>103096 mhs</td>
</tr>
<tr>
<td>3.9</td>
<td>7.501</td>
<td>7.501</td>
<td>9300</td>
<td>0</td>
<td>807 jac1d</td>
</tr>
<tr>
<td>3.4</td>
<td>838</td>
<td>6.594</td>
<td>604</td>
<td>1812</td>
<td>10918 exchange_3</td>
</tr>
<tr>
<td>3.4</td>
<td>6.590</td>
<td>6.590</td>
<td>9300</td>
<td>0</td>
<td>709 jacu</td>
</tr>
<tr>
<td>2.6</td>
<td>4.989</td>
<td>4.989</td>
<td>608</td>
<td>0</td>
<td>8206 MPI_Wait()</td>
</tr>
<tr>
<td>0.2</td>
<td>0.44</td>
<td>0.44</td>
<td>1</td>
<td>4</td>
<td>400081 init_comm</td>
</tr>
<tr>
<td>0.2</td>
<td>338</td>
<td>339</td>
<td>1</td>
<td>39</td>
<td>399634 MPI_Init()</td>
</tr>
<tr>
<td>0.1</td>
<td>240</td>
<td>247</td>
<td>1</td>
<td>47610</td>
<td>247006 suv</td>
</tr>
<tr>
<td>0.1</td>
<td>131</td>
<td>131</td>
<td>57252</td>
<td>0</td>
<td>2 exact</td>
</tr>
<tr>
<td>0.1</td>
<td>89</td>
<td>103</td>
<td>1</td>
<td>2</td>
<td>103168 erho</td>
</tr>
<tr>
<td>0.1</td>
<td>956</td>
<td>96</td>
<td>1</td>
<td>2</td>
<td>96458 read_input</td>
</tr>
<tr>
<td>0.0</td>
<td>95</td>
<td>95</td>
<td>9</td>
<td>0</td>
<td>10603 MPI_Bcast()</td>
</tr>
<tr>
<td>0.0</td>
<td>25</td>
<td>44</td>
<td>1</td>
<td>7937</td>
<td>44878 error</td>
</tr>
<tr>
<td>0.0</td>
<td>24</td>
<td>0</td>
<td>608</td>
<td>0</td>
<td>40 MPI_Recv()</td>
</tr>
<tr>
<td>0.0</td>
<td>15</td>
<td>15</td>
<td>1</td>
<td>5</td>
<td>15630 MPI_Finalize()</td>
</tr>
<tr>
<td>0.0</td>
<td>4</td>
<td>12</td>
<td>1</td>
<td>1700</td>
<td>12335 setbv</td>
</tr>
<tr>
<td>0.0</td>
<td>7</td>
<td>8</td>
<td>1</td>
<td>2384</td>
<td>19994 19994</td>
</tr>
<tr>
<td>0.0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>491 MPI_Allreduce()</td>
</tr>
<tr>
<td>0.0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3874 pint</td>
</tr>
<tr>
<td>0.0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1007 MPI_BARRIER()</td>
</tr>
<tr>
<td>0.0</td>
<td>116</td>
<td>0.837</td>
<td>1</td>
<td>4</td>
<td>837 exchange_4</td>
</tr>
<tr>
<td>0.0</td>
<td>512</td>
<td>0.512</td>
<td>1</td>
<td>0</td>
<td>512 MPI_Keyval_create()</td>
</tr>
<tr>
<td>0.0</td>
<td>121</td>
<td>0.353</td>
<td>1</td>
<td>2</td>
<td>353 exchange_5</td>
</tr>
<tr>
<td>0.0</td>
<td>224</td>
<td>0.191</td>
<td>1</td>
<td>2</td>
<td>131 exchange_6</td>
</tr>
<tr>
<td>0.0</td>
<td>103</td>
<td>0.103</td>
<td>0</td>
<td>0</td>
<td>17 MPI_Type_contiguous()</td>
</tr>
</tbody>
</table>

**POINT**

LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes
Parallel Profile Analysis – ParaProf

Raw files

PerfDMF managed (database)

Application

Experiment

Trial

HPMToolkit

Metadata

MpiP

TAU

Performance Tools to Understand and Optimize Multicore Codes
### Metadata for Each Experiment

#### Multiple PerfDMF DBs

<table>
<thead>
<tr>
<th>Trial Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>f90/pdt_mpi/examples/tau2/amorris/home/</td>
</tr>
<tr>
<td>Application ID</td>
<td>0</td>
</tr>
<tr>
<td>Experiment ID</td>
<td>0</td>
</tr>
<tr>
<td>Trial ID</td>
<td>0</td>
</tr>
<tr>
<td>CPU Cores</td>
<td>2</td>
</tr>
<tr>
<td>CPU MHz</td>
<td>2992.505</td>
</tr>
<tr>
<td>CPU Type</td>
<td>Intel(R) Xeon(R) CPU 5160 @ 3.00GHz</td>
</tr>
<tr>
<td>CPU Vendor</td>
<td>GenuineIntel</td>
</tr>
<tr>
<td>CWD</td>
<td>/home/amorris/tau2/examples/pdt_mpi/f90</td>
</tr>
<tr>
<td>Cache Size</td>
<td>4096 KB</td>
</tr>
<tr>
<td>Executable</td>
<td>/home/amorris/tau2/examples/pdt_mpi/f90</td>
</tr>
<tr>
<td>Hostname</td>
<td>demon.nic.oregon.edu</td>
</tr>
<tr>
<td>Local Time</td>
<td>2007-07-04T04:21:14-07:00</td>
</tr>
<tr>
<td>MPI Processor Name</td>
<td>demon.nic.oregon.edu</td>
</tr>
<tr>
<td>Memory Size</td>
<td>8161240 kB</td>
</tr>
<tr>
<td>Node Name</td>
<td>demon.nic.oregon.edu</td>
</tr>
<tr>
<td>OS Machine</td>
<td>x86_64</td>
</tr>
<tr>
<td>OS Name</td>
<td>Linux</td>
</tr>
<tr>
<td>OS Release</td>
<td>2.6.9-42.0.3.EL.perfctrsmp</td>
</tr>
<tr>
<td>OS Version</td>
<td>#1 SMP Fri Nov 3 07:34:13 PST 2006</td>
</tr>
<tr>
<td>Starting Timestamp</td>
<td>118354807220996</td>
</tr>
<tr>
<td>TAU Architecture</td>
<td>x86_64</td>
</tr>
<tr>
<td>TAU Config</td>
<td>-papi=/usr/local/packages/papi-3.5.0 -M...</td>
</tr>
<tr>
<td>Timestamp</td>
<td>1183548074317538</td>
</tr>
<tr>
<td>pid</td>
<td>11395</td>
</tr>
<tr>
<td>username</td>
<td>amorris</td>
</tr>
</tbody>
</table>
ParaProf – Flat Profile

8K processors

node, context, thread

Miranda
- hydrodynamics
- Fortran + MPI
- LLNL BG/L

192
Comparing Effects of Multi-Core Processors

Metric: PAPI_RES_STL
Value: Exclusive
Units: counts

AORSA2D
- magnetized plasma simulation
- Blue is single node
- Red is dual core
- Cray XT3 (4K cores)
Comparing FLOPS (AORSA2D, Cray XT3)

AORSA2D
- Blue is dual core
- Red is single node
- Cray XT3 (4K cores)
- Data generated by Richard Barrett, ORNL

LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes
ParaProf – Stacked View
ParaProf – Callpath Profile

Flash
● thermonuclear flashes
● Fortran + MPI
● Argonne

196
ParaProf – Scalable Histogram

8k processors

16k processors

LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes
ParaProf – 3D View (Full Profile)

128k processors
ParaProf – 3D View (Full Profile)

Miranda

16k processors
ParaProf – 3D Scatterplot

- Each point is a “thread” of execution
- A total of four metrics shown in relation
- ParaProf’s visualization library
  - JOGL
- Miranda, 32k cores
Performance Mapping

- Example: Particles distributed on cube surface

\[
\text{Particle* P[MAX]; /* Array of particles */}
\]

\[
\text{int GenerateParticles()} 
\]

\[
\text{/* distribute particles over all faces of the cube */}
\]

\[
\text{for (int face=0, last=0; face < 6; face++)}{}
\]

\[
\text{{/* particles on this face */}
\]

\[
\text{int particles_on_this_face = num(face);
}\]

\[
\text{for (int i=last; i < particles_on_this_face; i++) {
}\}
\]

\[
\text{{/* particle properties are a function of face */}
\]

\[
\text{P[i] = ... f(face);
}\]

\[
\text{...}
\]

\[
\text{last+= particles_on_this_face;
}\]

\[
\}\
\]

\[
\}\
\]
int ProcessParticle(Particle *p) {
    /* perform some computation on p */
}
int main() {
    GenerateParticles();
    /* create a list of particles */
    for (int i = 0; i < N; i++)
        /* iterates over the list */
        ProcessParticle(P[i]);
}

• How much time (flops) spent processing face i particles?
• What is the distribution of performance among faces?
No Mapping versus Mapping

- Typical performance tools report performance with respect to routines
- Does not provide support for mapping

- TAU’s performance mapping can observe performance with respect to scientist’s programming and problem abstractions

![TAU (no mapping)](image)

![TAU (w/ mapping)](image)
NAS BT – Flat Profile

How is MPI_Wait() distributed relative to solver direction?

Application routine names reflect phase semantics

How is MPI_Wait() distributed relative to solver direction?
NAS BT – Phase Profile

Main phase shows nested phases and immediate events
Phase Profiling of HW Counters

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

*Increasing phase execution time*

*Decreasing flops rate*

*Declining cache performance*
Profile Snapshots in ParaProf

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

- Only show main loop
- Percentage breakdown

LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes
Snapshot Replay in ParaProf

All windows dynamically update
PerfExplorer – Runtime Breakdown

Total Runtime Breakdown for S3D (Jaguar, ORNL): Harness Scaling Study:

GET_TIME_OF_DAY

WRITE_SAVEFILE

MPI_Waitall

DERIVATIVE.X_COMM [derivative_x.pp.f90] (53.14)
Loop: CHEMKIN.M::REACTION_RATE_BOUNDS [chemkin_m.pp.f90] (374.3-386.7)
Loop: DERIVATIVE.X_CALC [derivative_x.pp.f90] (432.10-441.15)
Loop: DERIVATIVE.X_CALC [derivative_x.pp.f90] (566.19-589.24)
Loop: DERIVATIVE.Y_CALC [derivative_y.pp.f90] (431.10-440.15)
Loop: DERIVATIVE.Z_CALC [derivative_z.pp.f90] (435.10-444.15)
Loop: INTEGRATE [integrate_erk.pp.f90] (73.3-93.13)
Loop: RHSF [rhsf.pp.f90] (209.3-211.7)
Loop: RHSF [rdsf.pp.f90] (515.3-535.16)
Loop: RHSF [rshf.pp.f90] (537.3-543.16)
Loop: RHSF [rshf.pp.f90] (545.3-551.16)
Loop: THERMCHM.M::CALC_INV_AVG_MOL_WT [thermchem_m.pp.f90] (127.5-129.9)
Loop: THERMCHM.M::CALC_SPECENTH_ALLPTS [thermchem_m.pp.f90] (506.3-512.8)
Loop: THERMCHM.M::CALC_TEMP [thermchem_m.pp.f90] (175.5-216.9)
Loop: TRANSPORT.M::COMPUTE_COEFFICIENTS [mixavg_transport_m.pp.f90] (492.5-520.9)
Loop: TRANSPORT.M::COMPUTE_HEATFLUX [mixavg_transport_m.pp.f90] (782.5-790.19)
Loop: TRANSPORT.M::COMPUTE_SPECIES_DIFFFLUX [mixavg_transport_m.pp.f90] (630.5-656.19)
Loop: VARIABLES.M::GET_MASS_SPECIES [variables_m.pp.f90] (96.3-99.7)
Loop: MPI_Comm_compare
Loop: MPI_Wait0
READWRITE_SAVEFILE_DATA [io.pp.f90] (544.14)
RHSF [rshf.pp.f90] (1.12)
WRITE_SAVEFILE [io.pp.f90] (240.14)
PerfExplorer – Relative Comparisons

- Total execution time
- Timesteps per second
- Relative efficiency
- Relative efficiency per event
- Relative speedup
- Relative speedup per event
- Group fraction of total
- Runtime breakdown
- Correlate events with total runtime
- Relative efficiency per phase
- Relative speedup per phase
- Distribution visualizations
Strong negative linear correlation between CALC_CUT_BLOCK_CONTRIBUTIONS and MPI_Barrier.
-0.995 indicates strong, negative relationship

As CALC_CUT_BLOCK_CONTRIBUTIONS() increases in execution time, MPI_BARRIER() decreases
PerfExplorer – Cluster Analysis
PerfExplorer – Cluster Analysis

- Four significant events automatically selected
- Clusters and correlations are visible
PerfExplorer – Performance Regression
Other Projects in TAU

• TAU Portal
  – Support collaborative performance study
• Kernel-level system measurements (KTAU)
  – Application to OS noise analysis and I/O system analysis
• TAU performance monitoring
  – TAUoverSupermon and TAUoverMRNet
• PerfExplorer integration and expert-based analysis
  – OpenUH compiler optimizations
  – Computational quality of service in CCA
• Eclipse CDT and PTP integration
• Performance tools integration (NSF POINT project)
Using TAU

- Install TAU
  - `% configure [options]; make clean install`
- Modify application makefile and choose TAU configuration
  - Select TAU’s stub makefile
  - Change name of compiler in makefile
- Set environment variables
  - Directory where profiles/traces are to be stored/counter selection
  - TAU options
- Execute application
  - `% mpirun -np <procs> a.out;`
- Analyze performance data
  - paraprof, vampir, pprof, paraver ...
Application Build Environment

• Minimize impact on user’s application build procedures
• Handle 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
Configuring TAU

• TAU can measure several metrics with profiling and tracing approaches
• Different tools can also be invoked to instrument programs for TAU measurement
• Each configuration of TAU produces a measurement library for an architecture
• Each measurement configuration of TAU also creates a corresponding stub makefile that can be used to compile programs
• Typically configure multiple measurement libraries
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 (bgl, xt3,x86_64,x86_64linux...)
  - `{-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
  - Use hardware counters + time
  - -COMPENSATE Compensate timer overhead
  - -CPUTIME Use usertime+system time
  - -PAPIWALLCLOCK Use PAPI’s wallclock time
  - -PAPIVIRTUAL Use PAPI’s process virtual time
  - -SGITIMERS Use fast IRIX timers
  - -LINUXTIMERS Use fast x86 Linux timers
TAU Configuration – Examples

• Configure using PDT and MPI for x86_64 Linux
  ./configure –pdt=/usr/pkgs/pkgs/pdtoolkit-3.15
  -mpiinc=/usr/pkgs/mpich/include -mpilib=/usr/pkgs/mpich/lib
  -mpilibrary=‘-lmpich -L/usr/gm/lib64 -lgm -lpthread -ldl’

• Use PAPI counters (one or more) with C/C++/F90
  automatic instrumentation for Cray CNL. Also instrument
  the MPI library. Use PGI compilers.
  ./configure -arch=craycnl -papi=/opt/xt-tools/papi/3.6.2 -mpi; make
  clean install

• Stub makefiles
  /usr/pkgstau/x86_64/lib/Makefile.tau-mpi-pdt-pgi
  /usr/pkgstau/x86_64/lib/Makefile.tau-mpi-papi-pdt-pgi
Stub Makefiles Configuration Parameters

- TAU scripts use stub makefiles to select performance measurements
- 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
TAU Measurement Configuration

- % cd /opt/tau-2.19.1/x86_64/lib; ls Makefile.*
  - Makefile.tau-pdt
  - Makefile.tau-mpi-pdt
  - Makefile.tau-mpi-papi-pdt
  - Makefile.tau-mpi-papi-pdt-trace
  - Makefile.tau-pthread-pdt...
- For an MPI+F90 application, you may want to start with:
  - Makefile.tau-mpi-pdt
  - Supports MPI instrumentation & PDT for automatic source instrumentation
- % setenv TAU_MAKEFILE /opt/tau-2.19.1/x86_64/lib/Makefile.tau-mpi-pdt
Using TAU: A brief Introduction

- To instrument source code using PDT
  - Choose an appropriate TAU stub makefile in <arch>/lib:
    % setenv TAU_MAKEFILE
    /opt/tau-2.19.1/x86_64/lib/Makefile.tau-mpi-pdt
    % setenv TAU_OPTIONS ‘-optVerbose …’ (see tau_compiler.sh)
    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 analyze performance data:
  % pprof (for text based profile display)
  % paraprof (for GUI)
TAU Measurement Configuration – Examples

% cd /usr/local/packages/tau-2.19.1/i386_linux/lib; ls Makefile.* on LiveDVD
Makefile.tau-pdt
Makefile.tau-mpi-pdt
Makefile.tau-papi-mpi-pdt
Makefile.tau-vampirtrace-papi-mpi-pdt
Makefile.tau-scalasca-papi-mpi-pdt
Makefile.tau-pthread-pdt
Makefile.tau-pthread-mpi-pdt
Makefile.tau-openmp-opari-pdt
Makefile.tau-openmp-opari-mpi-pdt
Makefile.tau-papi-openmp-opari-mpi-pdt
...

• For an MPI+F90 application, you may want to start with:
  Makefile.tau-mpi-pdt
    – Supports MPI instrumentation & PDT for automatic source instrumentation
    – % setenv TAU_MAKEFILE
      /usr/local/packages/tau-2.19.1/i386_linux/lib/Makefile.tau-mpi-pdt
PROFILE 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
Generating a Flat Profile with MPI

```bash
% setenv TAU_MAKEFILE /opt/tau-2.19.1/x86_64
  /lib/Makefile.tau-mpi-pdt
% set path=(/opt/tau-2.19.1/x86_64/bin $path)
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)

% qsub run.job
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.

% paraprof app.ppk
```
Generating a Loop-level Profile

```bash
% setenv TAU_MAKEFILE /opt/tau-2.19.1/x86_64
               /lib/Makefile.tau-mpi-pdt
% setenv TAU_OPTIONS `-optTauSelectFile=select.tau -optVerbose`
% cat select.tau
BEGIN_INSTRUMENT_SECTION
loops routine="#"
END_INSTRUMENT_SECTION

% set path=/opt/tau-2.19.1/x86_64/bin $path
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% qsub run.job
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.
% paraprof app.ppk
```
% setenv TAU_MAKEFILE /opt/tau-2.19.1/x86_64
    /lib/Makefile.tau-mpi
% setenv TAU_OPTIONS '–optCompInst –optVerbose'
% % set path=('/opt/tau-2.19.1/x86_64/bin $path)
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)

% qsub run.job
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.
% paraprof app.ppk
-papi Option

• Instead of one metric, profile or trace with more than one metric
  – Set environment variable TAU_METRICS to specify the metric
    • % setenv TAU_METRICS TIME:PAPI_FP_INS:PAPI_L1_DCM...
    • % setenv TAU_METRICS TIME:PAPI_NATIVE_<native_event>...

• When used with tracing (TAU_TRACE=1) option, the first counter must be TIME
  • % setenv TAU_METRICS TIME:PAPI_FP_INS...
  • Provides a globally synchronized real time clock for tracing

• -papi appears in the name of the stub Makefile
• papi_avail, papi_event_chooser, and papi_native_avail are useful tools
Generate a PAPI profile

\% setenv TAU_MAKEFILE /opt/tau-2.19.1/x86_64
\% lib/Makefile.tau-papi-mpi-pdt
\% setenv TAU_OPTIONS '-optTauSelectFile=select.tau -optVerbose'
\% cat select.tau
BEGIN_INSTRUMENT_SECTION
loops routine="#"
END_INSTRUMENT_SECTION

\% set path=(/opt/tau-2.19.1/x86_64/bin $path)
\% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
\% setenv TAU_METRICS TIME:PAPI_FP_INS

\% qsub run.job
\% paraprof --pack app.ppk
Move the app.ppk file to your desktop.
\% paraprof app.ppk
Choose Options -> Show Derived Panel -> Click PAPI_FP_INS, Click / , Click TIME, Apply, choose the metric
-PROFILECALLPATH Option

• Generates profiles that show the calling order (edges and 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 environment variable
  – -callpath in the name of the stub Makefile name or setting TAU_CALLPATH= 1 at runtime (TAU v2.18.1+)
-DEPTHLIMIT 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
• TAU_DEPTH_LIMIT environment variable specifies depth
  – % setenv TAU_DEPTH_LIMIT 1
  – enables instrumentation in only “main”
  – % setenv TAU_DEPTH_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-mpi-depthlimit-pdt
Generate a Callpath Profile

% setenv TAU_MAKEFILE /opt/tau-2.19.1/x86_64
    /lib/Makefile.tau-mpi-pdt
% set path=/opt/tau-2.19.1/x86_64/bin $path
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% setenv TAU_CALLPATH 1
% setenv TAU_CALLPATH_DEPTH 100

to generate the callpath profiles without any recompilation.
% qsub run.job
% paraprof --pack app.ppk
    Move the app.ppk file to your desktop.
% paraprof app.ppk
    (Windows -> Thread -> Call Graph)
Tracing in TAU

• Generates event-trace logs, rather than summary profiles
  – setenv TAU_TRACE 1
• Traces show when and where an event occurred in terms of location and the process that executed it
• Traces from multiple processes are merged:
  – % tau_treemerge.pl
    • generates tau.trc and tau.edf as merged trace and event definition file
• TAU traces can be converted to Vampir’s OTF/VTF3, Jumpshot SLOG2, Paraver trace formats:
  – % tau2otf tau.trc tau.edf app.otf
  – % tau2vtf tau.trc tau.edf app.vpt.gz
  – % tau2slog2 tau.trc tau.edf -o app.slog2
  – % tau_convert -paraver tau.trc tau.edf app.prv
Generate a Trace File

```bash
% setenv TAU_MAKEFILE /opt/tau-2.19.1/x86_64
    /lib/Makefile.tau-mpi-pdt
% set path=/opt/tau-2.19.1/x86_64/bin $path
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% setenv TAU_TRACE 1
% qsub run.job
% tau_treemerge.pl
(merges binary traces to create tau.trc and tau.edf files)
JUMPSHOT:
% tau2slog2 tau.trc tau.edf -o app.slog2
% jumpshot app.slog2
OR
VAMPIR:
% tau2otf tau.trc tau.edf app.otf -n 4 -z
(4 streams, compressed output trace)
% vampir app.otf
(or vng client with vngd server)
```
% tau_instrumentor
For selective instrumentation, use -f option
% tau_instrumentor foo.pdb foo.cpp -o foo.inst.cpp -f selective.dat
% cat selective.dat
# Selective instrumentation: Specify an exclude/include list of routines/files.
BEGIN_EXCLUDE_LIST
void quicksort(int *, int, int)
void sort_5elements(int *)
void interchange(int *, int *)
END_EXCLUDE_LIST

BEGIN_FILE_INCLUDE_LIST
Main.cpp
Foo?.c
*.C
END_FILE_INCLUDE_LIST
# Instruments routines in Main.cpp, Foo?.c and *.C files only
# Use BEGIN_[FILE]_INCLUDE_LIST with END_[FILE]_INCLUDE_LIST
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:

<table>
<thead>
<tr>
<th>Time</th>
<th>Exclusive</th>
<th>Inclusive</th>
<th>#Call</th>
<th>#Subrs</th>
<th>Inclusive Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>msec</td>
<td>total msec</td>
<td></td>
<td></td>
<td>usec/call</td>
</tr>
<tr>
<td>-------</td>
<td>-----------</td>
<td>-----------</td>
<td>-------</td>
<td>--------</td>
<td>-----------------</td>
</tr>
<tr>
<td>100.0</td>
<td>0.12</td>
<td>25,162</td>
<td>1</td>
<td>1</td>
<td>25162827 int main(int, char **)</td>
</tr>
<tr>
<td>100.0</td>
<td>0.175</td>
<td>25,162</td>
<td>1</td>
<td>4</td>
<td>25162707 double multiply()</td>
</tr>
<tr>
<td>90.5</td>
<td>22,778</td>
<td>22,778</td>
<td>1</td>
<td>0</td>
<td>22778959 Loop: double multiply()</td>
</tr>
<tr>
<td>9.3</td>
<td>2,345</td>
<td>2,345</td>
<td>1</td>
<td>0</td>
<td>2345823 Loop: double multiply()</td>
</tr>
<tr>
<td>9.3</td>
<td>2,345</td>
<td>2,345</td>
<td>1</td>
<td>0</td>
<td>2345823 Loop: double multiply()</td>
</tr>
<tr>
<td>0.1</td>
<td>33</td>
<td>33</td>
<td>1</td>
<td>0</td>
<td>33964 Loop: double multiply()</td>
</tr>
</tbody>
</table>

file = <loop_test.cpp> line,col = <23,3> to <30,3>
file = <loop_test.cpp> line,col = <38,3> to <46,7>
file = <loop_test.cpp> line,col = <16,10> to <21,12>
Using TAU: A brief Introduction

• To instrument source code using PDT
  – Choose an appropriate TAU stub makefile in <arch>/lib:
    % setenv TAU_MAKEFILE
    /opt/tau-2.19.1/x86_64/lib/Makefile.tau-mpi-pdt
    % setenv TAU_OPTIONS ‘-optVerbose ...’ (see tau_compiler.sh)
    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 analyze performance data:
  % pprof  (for text based profile display)
  % paraprof  (for GUI)
Goal: What routines account for the most time? How much?

Metric: P_VIRTUAL_TIME
Value: Exclusive
Units: seconds

- LEQ_IKSWEPT: 9647.318
- LEQ_BICGSO: 4357.213
- LEQ_MATVE: 2669.887
- SOLVE_SPECIES_EQ: 1777.752
- SOLVE_LIN_EQ: 1417.986
- PHYSICAL_PROP: 1028.448
- RRATES: 783.402
- LEQ_MSOLVET: 682.376
- INIT_AB_M: 530.858
- CALC_MASS_FLUX_SPHR: 463.788
- INIT_MU_S: 446.025
- CALC_RESID_S: 421.747
- SOLVE_ENERGY_EQ: 381.363
- SOURCE_PHI: 371.199
- DRAG_GS: 258.829
Solution: Generating a flat profile with MPI

```bash
% setenv TAU_MAKEFILE /opt/tau-2.19.1/x86_64
       /lib/Makefile.tau-mpi-pdt
% set path=/opt/tau-2.19.1/x86_64/bin $path
Or
% module load tau
% make F90=tau_f90.sh
Or
% tau_f90.sh matmult.f90 -o matmult
(Or edit Makefile and change F90=tau_f90.sh)
% qsub run.job
% paraprof
To view. To view the data locally on the workstation,
% paraprof --pack app.ppk
   Move the app.ppk file to your desktop.
% paraprof app.ppk
```
Usage Scenarios: Loop Level Instrumentation

- **Goal:** What loops account for the most time? How much?
- **Flat profile with wallclock time with loop instrumentation:**

<table>
<thead>
<tr>
<th>Metric: GET_TIME_OF_DAY</th>
<th>Value: Exclusive</th>
<th>Units: microseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1729975.333</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>Time (microseconds)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>443194</td>
<td>Loop: MULTIPLY_MATRICES [{matmult.f90} {31.9}-{36.14}]</td>
</tr>
<tr>
<td>81095</td>
<td>MAIN</td>
</tr>
<tr>
<td>49569</td>
<td>MPI_Bcast()</td>
</tr>
<tr>
<td>45669</td>
<td>Loop: MAIN [{matmult.f90} {86.9}-{106.14}]</td>
</tr>
<tr>
<td>12412</td>
<td>MPI_Send()</td>
</tr>
<tr>
<td>8959</td>
<td>Loop: INITIALIZE [{matmult.f90} {17.9}-{21.14}]</td>
</tr>
<tr>
<td>8953</td>
<td>Loop: INITIALIZE [{matmult.f90} {10.9}-{14.14}]</td>
</tr>
<tr>
<td>5609.2</td>
<td>MPI_Finalize()</td>
</tr>
<tr>
<td>2932.667</td>
<td>MULTIPLY_MATRICES</td>
</tr>
<tr>
<td>2577.667</td>
<td>Loop: MAIN [{matmult.f90} {117.9}-{128.14}]</td>
</tr>
<tr>
<td>2091.8</td>
<td>MPI_Barrier()</td>
</tr>
<tr>
<td>1875.667</td>
<td>Loop: MAIN [{matmult.f90} {112.9}-{115.14}]</td>
</tr>
<tr>
<td>1833</td>
<td>Loop: MAIN [{matmult.f90} {71.9}-{74.14}]</td>
</tr>
<tr>
<td>107</td>
<td>Loop: MAIN [{matmult.f90} {77.9}-{84.14}]</td>
</tr>
<tr>
<td>30</td>
<td>INITIALIZE</td>
</tr>
<tr>
<td>14.25</td>
<td>MPI_Comm_rank()</td>
</tr>
<tr>
<td>1</td>
<td>MPI_Comm_size()</td>
</tr>
</tbody>
</table>
```
Solution: Generating a loop level profile

```bash
% setenv TAU_MAKEFILE /opt/tau-2.19.1/x86_64
    /lib/Makefile.tau-mpi-pdt
% setenv TAU_OPTIONS '–optTauSelectFile=select.tau –optVerbose'
% cat select.tau
BEGIN_INSTRUMENT_SECTION
loops routine="#"
END_INSTRUMENT_SECTION

% module load tau
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% qsub run.job
% paraprof --pack app.ppk
    Move the app.ppk file to your desktop.

% paraprof app.ppk
```
Usage Scenarios: MFlops in Loops

- **Goal**: What execution rate do my application loops get in mflops?
- **Flat profile with PAPI_FP_INS/OPS and time (-papi) with loop instrumentation**:

Metric: PAPI_FP_INS / GET_TIME_OF_DAY
Value: Exclusive
Units: Derived metric shown in microseconds format

```
Loop: MULTIPLY_MATRICES ([matmult.f90] {31.9}-{36.14})
  770.699
  Loop: INITIALIZE ([matmult.f90] {10.9}-{14.14})
  223.39
  Loop: INITIALIZE ([matmult.f90] {17.9}-{21.14})
  223.24
  Loop: MAIN ([matmult.f90] {71.9}-{74.14})
  171.855
  Loop: MAIN ([matmult.f90] {112.9}-{115.14})
  170.862
  Loop: MAIN ([matmult.f90] {117.9}-{128.14})
  122.96
  MULTIPLY_MATRICES
  37.549
  INITIALIZE
  21.867
  Loop: MAIN ([matmult.f90] {86.9}-{106.14})
  13.795
  MPI_Comm_size()
  11
  Loop: MAIN ([matmult.f90] {77.9}-{84.14})
  8.935
  MPI_Send()
  1.131
  MPI_Comm_rank()
  0.794
  MPI_Bcast()
  0.647
  MPI_Recv()
  0.355
  MPI_Barrier()
  0.171
  MPI_Finalize()
  0.115
  MAIN
  0.023
```

LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes
Generate a PAPI profile with 2 or more counters

% setenv TAU_MAKEFILE /opt/tau-2.19.1/x86_64
   /lib/Makefile.tau-papi-mpi-pdt
% setenv TAU_OPTIONS ‘-optTauSelectFile=select.tau -optVerbose’
% cat select.tau
  BEGIN_INSTRUMENT_SECTION
  loops routine="#"
  END_INSTRUMENT_SECTION

% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% setenv TAU_METRICS TIME:PAPI_FP_INS
% qsub run.job
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.
% paraprof app.ppk
  Choose Options -> Show Derived Panel -> Arg 1 = PAPI_FP_INS,
  Arg 2 = GET_TIME_OF_DAY, Operation = Divide -> Apply, choose.
Goal: Easily generate routine level performance data using the compiler instead of PDT for parsing the source code
Use Compiler-Based Instrumentation

```bash
% setenv TAU_MAKEFILE /opt/tau-2.19.1/x86_64
          /lib/Makefile.tau-mpi-pdt
% setenv TAU_OPTIONS ‘-optCompInst -optVerbose’
% module load tau
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)

% qsub run.job
% paraprof --pack app.ppk
   Move the app.ppk file to your desktop.
% paraprof app.ppk
```
Generate a Callpath Profile

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Time (cts)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODIFIEDHYDROSWEEP:HYDRO_SWEEP</td>
<td>26.474%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO, 3D =&gt; MODIFIEDHYDROSWEEP:HYDRO_SWEEP</td>
<td>24.559%</td>
</tr>
<tr>
<td>MODIFIEDHYDRO:HYDRO, 3D</td>
<td>24.350%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO, 3D =&gt; MODIFIEDHYDROSWEEP:HYDRO_SWEEP =&gt; MODIFIEDHYDRO_1D:HYDRO_1D</td>
<td>14.351%</td>
</tr>
<tr>
<td>MODIFIEDHYDRO_1D:HYDRO_1D</td>
<td>14.351%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO, 3D =&gt; MODIFIEDHYDROSWEEP:HYDRO_SWEEP =&gt; MODIFIEDHYDRO_1D:HYDRO_1D =&gt; MODULEINTRFC:INTRFC</td>
<td>4.581%</td>
</tr>
<tr>
<td>MODULEINTRFC:INTRFC</td>
<td>4.477%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO, 3D =&gt; MODIFIEDHYDROSWEEP:HYDRO_SWEEP =&gt; MODULEINTRFC:INTRFC =&gt; MPI_Ssend()</td>
<td>3.678%</td>
</tr>
<tr>
<td>MPI_Ssend()</td>
<td>3.536%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO, 3D =&gt; MODIFIEDHYDROSWEEP:HYDRO_SWEEP =&gt; MODULEINTRFC:INTRFC =&gt; MPI_Ssend() =&gt; MPI_Waitall()</td>
<td>2.727%</td>
</tr>
<tr>
<td>MPI_Waitall()</td>
<td>2.323%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO, 3D =&gt; MODIFIEDHYDROSWEEP:HYDRO_SWEEP =&gt; MODULEINTRFC:INTRFC =&gt; MPI_Ssend() =&gt; MPI_Waitall() =&gt; MODULEUPDATE_SOL:UPDATE_SOLN</td>
<td>2.242%</td>
</tr>
<tr>
<td>MODULEUPDATE_SOL:UPDATE_SOLN</td>
<td>2.059%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO, 3D =&gt; MODULEHYDROSWEEP:HYDRO_SWEEP =&gt; MODULEUPDATE_SOL:UPDATE_SOLN =&gt; AMR_GUARDCELL, c.c.s.l.</td>
<td>1.793%</td>
</tr>
<tr>
<td>AMR_GUARDCELL, c.c.s.l</td>
<td>1.560%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO, 3D =&gt; MODULEHYDROSWEEP:HYDRO_SWEEP =&gt; MODULEUPDATE_GRID_REFINEMENT =&gt; MESH_REFINE:DEREFINE =&gt; AMR_Refine:DEREFINE =&gt; AMR_Morton:ORDER</td>
<td>1.496%</td>
</tr>
<tr>
<td>MESH_REFINE:DEREFINE =&gt; AMR_Morton:ORDER</td>
<td>1.361%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; MESH_UPDATE_GRID_REFINEMENT =&gt; MESH_REFINE:DEREFINE =&gt; AMR.Refine:DEREFINE =&gt; AMR_Morton:ORDER</td>
<td>1.319%</td>
</tr>
<tr>
<td>AMR_Refine:DEREFINE =&gt; AMR_Morton:ORDER</td>
<td>1.272%</td>
</tr>
<tr>
<td>AMR_PROLONG:GEN.REFINE</td>
<td>1.093%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO, 3D =&gt; MODULEHYDROSWEEP:HYDRO_SWEEP =&gt; MESH_GUARDCELL =&gt; AMR_GUARDCELL_C, TO, F</td>
<td>1.027%</td>
</tr>
<tr>
<td>MESH_GUARDCELL =&gt; AMR_GUARDCELL_C, TO, F</td>
<td>1.027%</td>
</tr>
<tr>
<td>ABUNDANCE:RESTRICT</td>
<td>1.027%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO, 3D =&gt; MODULEHYDROSWEEP:HYDRO_SWEEP =&gt; MESH_GUARDCELL =&gt; AMR_GUARDCELL_C, TO, F =&gt; ABUNDANCE:RESTRICT</td>
<td>1.027%</td>
</tr>
<tr>
<td>ABUNDANCE:RESTRICT</td>
<td>1.027%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO, 3D =&gt; MODULEHYDROSWEEP:HYDRO_SWEEP =&gt; MESH_GUARDCELL =&gt; AMR_GUARDCELL_C, TO, F =&gt; ABUNDANCE:RESTRICT =&gt; DEASEREF:DEASEREF:DEASEREF</td>
<td>1.064%</td>
</tr>
<tr>
<td>DEASEREF:DEASEREF:DEASEREF</td>
<td>0.987%</td>
</tr>
<tr>
<td>MESH:CONSERVE:CONSERVE:CONSERVE:CONSERVE:UDDT:UDDT:UDDT</td>
<td>0.916%</td>
</tr>
<tr>
<td>MPI_Barrier()</td>
<td>0.872%</td>
</tr>
<tr>
<td>AMR_PROLONG:GEN.REFINE</td>
<td>0.884%</td>
</tr>
<tr>
<td>AMR_PROLONG:GEN.REFINE</td>
<td>0.735%</td>
</tr>
<tr>
<td>DIFFUSE =&gt; AMR_PROLONG:GEN.REFINE</td>
<td>0.699%</td>
</tr>
<tr>
<td>AMR_PROLONG:GEN.REFINE</td>
<td>0.671%</td>
</tr>
<tr>
<td>AMR_PROLONG:GEN.REFINE =&gt; AMR_PROLONG:GEN.REFINE</td>
<td>0.657%</td>
</tr>
<tr>
<td>AMR_PROLONG:GEN.REFINE =&gt; AMR_PROLONG:GEN.REFINE =&gt; AMR_PROLONG:GEN.REFINE</td>
<td>0.638%</td>
</tr>
<tr>
<td>AMR_PROLONG:GEN.REFINE =&gt; AMR_PROLONG:GEN.REFINE =&gt; AMR_PROLONG:GEN.REFINE =&gt; AMR_PROLONG:GEN.REFINE</td>
<td>0.618%</td>
</tr>
<tr>
<td>AMR_PROLONG:GEN.REFINE =&gt; AMR_PROLONG:GEN.REFINE =&gt; AMR_PROLONG:GEN.REFINE =&gt; AMR_PROLONG:GEN.REFINE =&gt; AMR_PROLONG:GEN.REFINE</td>
<td>0.598%</td>
</tr>
<tr>
<td>AMR_PROLONG:GEN.REFINE =&gt; AMR_PROLONG:GEN.REFINE =&gt; AMR_PROLONG:GEN.REFINE =&gt; AMR_PROLONG:GEN.REFINE =&gt; AMR_PROLONG:GEN.REFINE =&gt; AMR_PROLONG:GEN.REFINE</td>
<td>0.454%</td>
</tr>
</tbody>
</table>

LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes
Callpath Profile

- Generates program callgraph
Generate a Callpath Profile

```bash
% setenv TAU_MAKEFILE /opt/tau-2.19.1/x86_64
   /lib/Makefile.tau-mpi-pdt
% set path=/opt/tau-2.19.1/x86_64/bin $path
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% setenv TAU_CALLPATH 1
% setenv TAU_CALLPATH_DEPTH 100

% qsub run.job
% paraprof --pack app.ppk
   Move the app.ppk file to your desktop.
% paraprof app.ppk
(Windows -> Thread -> Call Graph)
```
Usage Scenario: Detect Memory Leaks

**TALI: ParaProf: Mean Context Events - mem.ppk**

- **Name**: MAIN [{matrix.f90} {141,7}–{146,22}]
- **MATRICES:ALLOCATE_MATRICES** [{matrix.f90} {10,7}–{13,38}]
  - MEMORY LEAK! malloc size <file=matrix.f90, variable=C, line=11> 1 8,000,000 8,000,000 8,000,000 0
  - malloc size <file=matrix.f90, variable=A, line=11> 1 8,000,000 8,000,000 8,000,000 0
  - malloc size <file=matrix.f90, variable=B, line=11> 1 8,000,000 8,000,000 8,000,000 0
  - malloc size <file=matrix.f90, variable=C, line=11> 1 8,000,000 8,000,000 8,000,000 0
- **MATRICES:DEALLOCATE_MATRICES** [{matrix.f90} {14,7}–{17,40}]
  - free size <file=matrix.f90, variable=A, line=15> 1 8,000,000 8,000,000 8,000,000 0
  - free size <file=matrix.f90, variable=B, line=15> 1 8,000,000 8,000,000 8,000,000 0

**User Event Window: mem.ppk**

- **Name**: MEMORY LEAK! malloc size <file=matrix.f90, variable=C, line=11> : MAIN [{matrix.f90} {141,7}–{146,22}] => MATRICES:ALLOCATE_MATRICES [{matrix.f90} {10,7}–{13,38}]
- **Value Type**: Max Value
- Mean
  - n.r.t 0.0
  - n.r.t 1.0
  - n.r.t 2.0
  - n.r.t 3.0
- Std. Dev. 0

**LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes**

253
% setenv TAU_MAKEFILE /opt/tau-2.19.1/x86_64/lib/Makefile.tau-mpi-pdt
% setenv TAU_OPTIONS ‘-optDetectMemoryLeaks -optVerbose’
% module load tau
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% setenv TAU_CALLPATH_DEPTH 100

% qsub run.job
% paraprof --pack app.ppk
   Move the app.ppk file to your desktop.
% paraprof app.ppk
(Windows -> Thread -> Context Event Window -> Select thread ->
   select... expand tree)
(Windows -> Thread -> User Event Bar Chart -> right click LEAK
   -> Show User Event Bar Chart)

NOTE: setenv TAU_TRACK_HEAP 1 and setenv TAU_TRACK_HEADROOM 1 may be used to track
heap and headroom utilization at the entry and exit of each routine.
TAU_CALLPATH_DEPTH=1 shows just the routine name, and 0 shows just one event for the
entire program.
Interval Events, Atomic Events in TAU

<table>
<thead>
<tr>
<th>%Time</th>
<th>Exclusive msecs</th>
<th>Inclusive total msecs</th>
<th>#Call</th>
<th>#Subrs</th>
<th>Inclusive msecs/call</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0</td>
<td>0.187</td>
<td>1.105</td>
<td>1</td>
<td>44</td>
<td>1105659</td>
<td>int main(int. char **) C</td>
</tr>
<tr>
<td>93.2</td>
<td>1.030</td>
<td>1.030</td>
<td>1</td>
<td>0</td>
<td>1020654</td>
<td>MPI_Init()</td>
</tr>
<tr>
<td>5.9</td>
<td>0.879</td>
<td>65</td>
<td>40</td>
<td>320</td>
<td>1637</td>
<td>void func(int. int) C</td>
</tr>
<tr>
<td>4.6</td>
<td>51</td>
<td>51</td>
<td>40</td>
<td>0</td>
<td>1277</td>
<td>MPI_Barrier()</td>
</tr>
<tr>
<td>1.2</td>
<td>13</td>
<td>13</td>
<td>120</td>
<td>0</td>
<td>111</td>
<td>MPI_Recev()</td>
</tr>
<tr>
<td>0.8</td>
<td>9</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>9328</td>
<td>MPI_Finalize()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.137</td>
<td>0.137</td>
<td>120</td>
<td>0</td>
<td>1</td>
<td>MPI_Send()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.086</td>
<td>0.086</td>
<td>40</td>
<td>0</td>
<td>2</td>
<td>MPI_Bcast()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.002</td>
<td>0.002</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>MPI_Comm_size()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.001</td>
<td>0.001</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>MPI_Comm_rank()</td>
</tr>
</tbody>
</table>

**USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0**

<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>365</td>
<td>5.138E+04</td>
<td>44.39</td>
<td>3.09E+04</td>
<td>1.234E+04</td>
<td>Heap Memory Used (KB) : Entry</td>
</tr>
<tr>
<td>365</td>
<td>5.138E+04</td>
<td>2064</td>
<td>3.115E+04</td>
<td>1.21E+04</td>
<td>Heap Memory Used (KB) : Exit</td>
</tr>
</tbody>
</table>

% setenv TAU_CALLPATH_DEPTH          0
% setenv TAU_TRACK_HEAP             1
Atomic Events, Context Events

<table>
<thead>
<tr>
<th>%Time</th>
<th>Exclusive msec</th>
<th>Inclusive total msec</th>
<th>#Call</th>
<th>#Subs</th>
<th>Inclusive Name usec/call</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0</td>
<td>0.253</td>
<td>1.106</td>
<td>1</td>
<td>44</td>
<td>1106701 int main(int, char **) C</td>
</tr>
<tr>
<td>93.2</td>
<td>1.031</td>
<td>1.031</td>
<td>1</td>
<td>0</td>
<td>103111 MPI_Init()</td>
</tr>
<tr>
<td>6.0</td>
<td>1</td>
<td>66</td>
<td>40</td>
<td>320</td>
<td>1650 void func(int, int) C</td>
</tr>
<tr>
<td>5.7</td>
<td>63</td>
<td>63</td>
<td>40</td>
<td>0</td>
<td>1588 MPI_BARRIER()</td>
</tr>
<tr>
<td>0.8</td>
<td>9</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>9119 MPI_Finalize()</td>
</tr>
<tr>
<td>0.1</td>
<td>1</td>
<td>1</td>
<td>120</td>
<td>0</td>
<td>10 MPI_Send()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.141</td>
<td>0.141</td>
<td>120</td>
<td>0</td>
<td>1 MPI_Send()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.085</td>
<td>0.085</td>
<td>40</td>
<td>0</td>
<td>2 MPI_Bcast()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.001</td>
<td>0.001</td>
<td>1</td>
<td>0</td>
<td>1 MPI_Comm_size()</td>
</tr>
<tr>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0 MPI_Comm_rank()</td>
</tr>
</tbody>
</table>

USER EVENTS Profile : NODE 0, CONTEXT 0, THREAD 0

<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>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>Message size for broadcast</td>
</tr>
<tr>
<td>365</td>
<td>5.139E+04</td>
<td>44.39</td>
<td>3.091E+04</td>
<td>1.234E+04</td>
<td>Heap Memory Used (KB) : Entry</td>
</tr>
<tr>
<td>40</td>
<td>5.139E+04</td>
<td>3097</td>
<td>3.114E+04</td>
<td>1.227E+04</td>
<td>Heap Memory Used (KB) : Entry : MPI_BARRIER()</td>
</tr>
<tr>
<td>40</td>
<td>5.139E+04</td>
<td>1.13E+04</td>
<td>3.134E+04</td>
<td>1.187E+04</td>
<td>Heap Memory Used (KB) : Entry : MPI_Bcast()</td>
</tr>
<tr>
<td>1</td>
<td>2067</td>
<td>2067</td>
<td>2067</td>
<td>0</td>
<td>Heap Memory Used (KB) : Entry : MPI_Comm_rank()</td>
</tr>
<tr>
<td>1</td>
<td>2066</td>
<td>2066</td>
<td>2066</td>
<td>0</td>
<td>Heap Memory Used (KB) : Entry : MPI_Comm_size()</td>
</tr>
<tr>
<td>1</td>
<td>5.139E+04</td>
<td>5.139E+04</td>
<td>5.139E+04</td>
<td>0.0006905</td>
<td>Heap Memory Used (KB) : Entry : MPI_Finalize()</td>
</tr>
<tr>
<td>1</td>
<td>57.56</td>
<td>57.56</td>
<td>57.56</td>
<td>0</td>
<td>Heap Memory Used (KB) : Entry : MPI_Init()</td>
</tr>
<tr>
<td>120</td>
<td>5.139E+04</td>
<td>1.13E+04</td>
<td>3.134E+04</td>
<td>1.187E+04</td>
<td>Heap Memory Used (KB) : Entry : MPI_Recv()</td>
</tr>
<tr>
<td>120</td>
<td>5.139E+04</td>
<td>1.129E+04</td>
<td>3.134E+04</td>
<td>1.187E+04</td>
<td>Heap Memory Used (KB) : Entry : MPI_Send()</td>
</tr>
<tr>
<td>1</td>
<td>44.39</td>
<td>44.39</td>
<td>44.39</td>
<td>0</td>
<td>Heap Memory Used (KB) : Entry : int main(int, char **) C</td>
</tr>
<tr>
<td>40</td>
<td>5.036E+04</td>
<td>2068</td>
<td>3.011E+04</td>
<td>1.227E+04</td>
<td>Heap Memory Used (KB) : Entry : void func(int, int) C</td>
</tr>
</tbody>
</table>

% setenv TAU_CALLPATH_DEPTH  1
% setenv TAU_TRACK_HEAP 1

LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes
Context Events (default)

% setenv TAU_CALLPATH_DEPTH 2
% setenv TAU_TRACK_HEAP 1

LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes
<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAU_TRACE</td>
<td>0</td>
<td>Setting to 1 turns on tracing</td>
</tr>
<tr>
<td>TAU_CALLPATH</td>
<td>0</td>
<td>Setting to 1 turns on callpath profiling</td>
</tr>
<tr>
<td>TAU_TRACK_HEAP or</td>
<td>0</td>
<td>Setting to 1 turns on tracking heap memory/headroom at routine entry &amp; exit</td>
</tr>
<tr>
<td>TAU_TRACK_HEADROOM</td>
<td></td>
<td>using context events (e.g., Heap at Entry: main=&gt;foo=&gt;bar)</td>
</tr>
<tr>
<td>TAU_CALLPATH_DEPTH</td>
<td>2</td>
<td>Specifies depth of callpath. Setting to 0 generates no callpath or routine</td>
</tr>
<tr>
<td></td>
<td></td>
<td>information, setting to 1 generates flat profile and context events have just</td>
</tr>
<tr>
<td></td>
<td></td>
<td>parent information (e.g., Heap Entry: foo)</td>
</tr>
<tr>
<td>TAU_SYNCHRONIZE_CLOCKS</td>
<td>1</td>
<td>Synchronize clocks across nodes to correct timestamps in traces</td>
</tr>
<tr>
<td>TAU_COMM_MATRIX</td>
<td>0</td>
<td>Setting to 1 generates communication matrix display using context events</td>
</tr>
<tr>
<td>TAU_THROTTLE</td>
<td>1</td>
<td>Setting to 0 turns off throttling. Enabled by default to remove instrumentation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in lightweight routines that are called frequently</td>
</tr>
<tr>
<td>TAU_THROTTLE_NUMCALLS</td>
<td>100000</td>
<td>Specifies the number of calls before testing for throttling</td>
</tr>
<tr>
<td>TAU_THROTTLE_PERCALL</td>
<td>10</td>
<td>Specifies value in microseconds. Throttle a routine if it is called over 100000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>times and takes less than 10 usec of inclusive time per call</td>
</tr>
<tr>
<td>TAU_COMPENSATE</td>
<td>0</td>
<td>Setting to 1 enables runtime compensation of instrumentation overhead</td>
</tr>
<tr>
<td>TAU_PROFILE_FORMAT</td>
<td>Profile</td>
<td>Setting to “merged” generates a single file. “snapshot” generates xml format</td>
</tr>
<tr>
<td>TAU_METRICS</td>
<td>TIME</td>
<td>Setting to a comma separated list generates other metrics. (e.g.,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TIME:linuxtimers:PAPI_FP_OPS:PAPI_NATIVE_&lt;event&gt;)</td>
</tr>
</tbody>
</table>
Optional parameters for TAU_OPTIONS: [tau_compiler.sh –help]

- **-optVerbose**  
  Turn on verbose debugging messages

- **-optCompInst**  
  Use compiler based instrumentation

- **-optDetectMemoryLeaks**  
  Turn on debugging memory allocations/de-allocations to track leaks

- **-optKeepFiles**  
  Does not remove intermediate .pdb and .inst.* files

- **-optPreFiles**  
  Preprocess Fortran sources before instrumentation

- **-optTauSelectFile=""**  
  Specify selective instrumentation file for tau_instrumentor

- **-optLinking=""**  
  Options passed to the linker. Typically
  
  $$\text{\$(TAU\_MPI\_FLIBS) \$(TAU\_LIBS) \$(TAU\_CXXLIBS)}$$

- **-optCompile=""**  
  Options passed to the compiler. Typically
  
  $$\text{\$(TAU\_MPI\_INCLUDE) \$(TAU\_INCLUDE) \$(TAU\_DEFS)}$$

- **-optTauSelectFile=""**  
  Specify selective instrumentation file for tau_instrumentor

- **-optNoCompInst**  
  Do not revert to compiler-based instrumentation if source instrumentation fails

- **-optPdtF95Opts=""**  
  Add options for Fortran parser in PDT (f95parse/gfparse)

- **-optPdtF95Reset=""**  
  Reset options for Fortran parser in PDT (f95parse/gfparse)

- **-optPdtCxxOpts=""**  
  Options for C parser in PDT (cparse). Typically
  
  $$\text{\$(TAU\_MPI\_INCLUDE) \$(TAU\_INCLUDE) \$(TAU\_DEFS)}$$

- **-optPdtCxxOpts=""**  
  Options for C++ parser in PDT (cxpparse). Typically
  
  $$\text{\$(TAU\_MPI\_INCLUDE) \$(TAU\_INCLUDE) \$(TAU\_DEFS)}$$
Measuring Performance of PGI Accelerator Code

### Metric: TIME
Value: Exclusive percent

<table>
<thead>
<tr>
<th>Function</th>
<th>Exclusive Time</th>
<th>Inclusive Time</th>
<th>Calls</th>
<th>Child Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>__pgi_cu_launch multiply_matrices (pgi_kernel_7,gx=32,gy=32,gz=1,bx=16,by=16,bz=1) [mm2.f90][15]</td>
<td>10.901</td>
<td>10.901</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>__pgi_cu_init multiply_matrices [mm2.f90][9]</td>
<td>3.912</td>
<td>3.912</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>__pgi_cu_download2 multiply_matrices var=a [mm2.f90][20]</td>
<td>0.514</td>
<td>0.514</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>__pgi_cu_upload2 multiply_matrices var=b [mm2.f90][9]</td>
<td>0.252</td>
<td>0.252</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>__pgi_cu_upload2 multiply_matrices var=c [mm2.f90][9]</td>
<td>0.252</td>
<td>0.252</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>__pgi_cu_launch multiply_matrices (pgi_kernel_2,gx=32,gy=32,gz=1,bx=16,by=16,bz=1) [mm2.f90][11]</td>
<td>0.125</td>
<td>16.021</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>__pgi_cu_free multiply_matrices [mm2.f90][1]</td>
<td>0.023</td>
<td>0.023</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>__pgi_cu_launch multiply_matrices (pgi_kernel_2,gy=32,gy=32,gz=1,bx=16,by=16,bz=1) [mm2.f90][11]</td>
<td>0.019</td>
<td>0.019</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>__pgi_cu_alloc multiply_matrices [mm2.f90][9]</td>
<td>0.003</td>
<td>15.895</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>__pgi_cu_module multiply_matrices [mm2.f90][9]</td>
<td>0.001</td>
<td>15.893</td>
<td>5</td>
<td>85</td>
</tr>
<tr>
<td>__pgi_cu_module_function multiply_matrices [mm2.f90][11]</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>__pgi_cu_paramset multiply_matrices [mm2.f90]</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

---

**LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes**

260
Usage Scenarios: Mixed Python+F90+C+pyMPI

• Goal: Generate multi-level instrumentation for Python+MPI+C+F90+C++ ...

LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes
Generate a Multi-Language Profile w/ Python

```bash
% setenv TAU_MAKEFILE /opt/tau-2.19.1/x86_64
    /lib/Makefile.tau-python-mpi-pdt
% set path=/opt/tau-2.19.1/x86_64/bin $path
% setenv TAU_OPTIONS `-optShared -optVerbose…`

(Python needs shared object based TAU library)
% make F90=tau_f90.sh CXX=tau_cxx.sh CC=tau_cc.sh  (build pyMPI w/TAU)
% cat wrapper.py
    import tau
def OurMain():
    import App
    tau.run('OurMain()')

Uninstrumented:
% poe <dir>/pyMPI-2.4b4/bin/pyMPI ./App.py --procs 4

Instrumented:
% setenv PYTHONPATH <taudir>/x86_64/lib/bindings-python-mpi-pdt-pgi
    (same options string as TAU_MAKEFILE)
% setenv LD_LIBRARY_PATH <taudir>/x86_64/lib/bindings-icpc-python-mpi-pdt-pgi
    :$LD_LIBRARY_PATH
% poe <dir>/pyMPI-2.5b0-TAU/bin/pyMPI ./wrapper.py --procs 4
    (Instrumented pyMPI with wrapper.py)
```
Usage Scenarios: Evaluate Scalability

- Goal: How does my application scale? What bottlenecks at what cpu counts?
- Load profiles in PerfDMF database and examine with PerfExplorer
Usage Scenarios: Evaluate Scalability

LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes
Performance Regression Testing

FACETS Bassi Regression: 32 Procs (events above 2%)

Exclusive Time (seconds)

Date


int main(int, char **) std::vector<double, std::allocator<double>> FcCoreCellUpdate...
void FcTmCoreFluxCalc::computeFluxes() MPI_Recv()
double FcDataAssimilator::getValue(const std::string &, const ...) MPI_Init()
FcHdfSTmpl <DATATYPE>::writeDataSet
void FcDataAssimilatorUfiles::parseUfiles(const std::vector<...
void FcUpdaterComponent::dumpToFile(const std::string &)
other
Evaluate Scalability using PerfExplorer Charts

% setenv TAU_MAKEFILE /opt/tau-2.19.1/x86_64
       /lib/Makefile.tau-mpi-pdt
% set path=/opt/tau-2.19.1/x86_64/bin $path
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% qsub run1p.job
% paraprof --pack 1p.ppk
% qsub run2p.job ...
% paraprof --pack 2p.ppk ... and so on.
On your client:
% perfdmf_configure
(Choose derby, blank user/passwd, yes to save passwd, defaults)
% perfexplorer_configure
(Yes to load schema, defaults)
% paraprof
(load each trial: DB -> Add Trial -> Type (Paraprof Packed Profile) -> OK, OR use perfdmf_loadtrial on the commandline)
% perfexplorer
(Charts -> Speedup)
Goal: What is the volume of inter-process communication? Along which calling path?
Evaluate Scalability using PerfExplorer Charts

% setenv TAU_MAKEFILE
   $TAU/Makefile.tau-mpi-pdt
% set path=(/usr/local/packages/tau-2.19.1/x86_64/bin $path)
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)
% setenv TAU_COMM_MATRiX 1

% qsub run.job (setting the environment variables)

% paraprof
(Windows -> Communication Matrix)
(Windows -> 3D Communication Matrix)
TAU Integration with IDEs

- High performance software development environments
  - Tools may be complicated to use
  - Interfaces and mechanisms differ between platforms / OS
- Integrated development environments
  - Consistent development environment
  - Numerous enhancements to development process
  - Standard in industrial software development
- Integrated performance analysis
  - Tools limited to single platform or programming language
  - Rarely compatible with 3rd party analysis tools
  - Little or no support for parallel projects
TAU and Eclipse

- Provide an interface for configuring TAU’s automatic instrumentation within Eclipse’s build system
- Manage runtime configuration settings and environment variables for execution of TAU instrumented programs
TAU and Eclipse

PerfDMF

ECF 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes
Choosing PAPI Counters with TAU in Eclipse

% /usr/local/packages/eclipse/eclipse
VAMPIRTRACE & VAMPIR
INTRODUCTION AND OVERVIEW

Sameer Shende
Performance Research Laboratory
University of Oregon
Overview

• Introduction
• Event Trace Visualization
• Vampir & VampirServer
• The Vampir Displays
  – Timeline
  – Process Timeline with Performance Counters
  – Summary Display
  – Message Statistics
• VampirTrace
  – Instrumentation & Run-Time Measurement
• Conclusions
VampirServer Architecture

Parallel Program
- Monitor System
- Process
- Event Streams
- Parallel I/O

File System
- Trace 1
- Trace 2
- Trace 3
- Trace N

Analysis Server
- Worker 1
- Worker 2
- Worker m

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

LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes
Vampir Displays

The main displays of Vampir:

• Global Timeline
• Process Timeline w/o Counters
• Statistic Summary
• Summary Timeline
• Message Statistics
• Collective Operation Statistics
• Counter Timeline
• Call Tree
Vampir Global Timeline Display

LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes
Process Timeline Display

Using POINT Performance Tools to Understand and Optimize Multicore Codes
Process Timeline with Counters

![Vampir - Process Timeline](image)

LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes
Statistic Summary Display

LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes
Vampir and VampirTraces are available at [http://www.vampir.eu](http://www.vampir.eu) and [http://www.tu-dresden.de/zh/vampirtrace/](http://www.tu-dresden.de/zh/vampirtrace/),

get support via vampirsupport@zh.tu-dresden.de
Jumpshot

- Developed at Argonne National Laboratory as part of the MPICH project
  - Also works with other MPI implementations
  - Installed on IBM BG/P
  - Jumpshot is bundled with the TAU package
- Java-based tracefile visualization tool for postmortem performance analysis of MPI programs
- Latest version is Jumpshot-4 for SLOG-2 format
  - Scalable level of detail support
  - Timeline and histogram views
  - Scrolling and zooming
  - Search/scan facility
Jumpshot

LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes
Support Acknowledgements

- Department of Energy (DOE)
  - Office of Science
    - MICS, Argonne National Lab
  - ASC/NNSA
    - University of Utah ASC/NNSA Level 1
    - ASC/NNSA, Lawrence Livermore National Lab
- Department of Defense (DoD)
  - HPC Modernization Office (HPCMO)
- NSF Software Development for Cyberinfrastructure (SDCI)
- Research Centre Juelich
- ANL, NASA Ames, LANL, SNL
- TU Dresden
- ParaTools, Inc.
For more information

• TAU Website:
  http://tau.uoregon.edu
  – Software
  – Release notes
  – Documentation
SCALABLE PERFORMANCE ANALYSIS OF LARGE-SCALE PARALLEL APPLICATIONS

Sameer Shende
Performance Research Laboratory, University of Oregon

Shirley Moore
Innovative Computing Laboratory, University of Tennessee, Knoxville

Brian J. N. Wylie and Markus Geimer
Jülich Supercomputing Centre
Performance analysis, tools & techniques

• Profile analysis
  – Summary of aggregated metrics
    • per function/call-path and/or per process/thread
  – Most tools (can) generate and/or present such profiles
    • but they do so in very different ways, often from event traces!
    – e.g., mpiP, ompP, TAU, PerfSuite, Scalasca, Sun Studio, Vampir, ...

• Time-line analysis
  – Visual representation of the space/time sequence of events
  – Requires an execution trace
  – e.g., Vampir, Paraver, Sun Studio Performance Analyzer, ...

• Pattern analysis
  – Search for characteristic event sequences in event traces
  – Can be done manually, e.g., via visual time-line analysis
  – Can be done automatically, e.g., KOJAK, Scalasca
Automatic trace analysis

• Idea
  – Automatic search for patterns of inefficient behaviour
  – Classification of behaviour & quantification of significance

  – Guaranteed to cover the entire event trace
  – Quicker than manual/visual trace analysis
  – Parallel replay analysis exploits memory & processors to deliver scalability
The Scalasca project

• Overview
  – Helmholtz Initiative & Networking Fund project started in 2006
  – Headed by Prof. Felix Wolf (RWTH Aachen University & JSC)
  – Follow-up to pioneering KOJAK project (started 1998)
    • Automatic pattern-based trace analysis

• Objective
  – Development of a scalable performance analysis toolset
  – Specifically targeting large-scale parallel applications
    • such as those running on BlueGene/P or Cray XT with 10,000s to 100,000s of processes
  – Latest release in February 2010: Scalasca v1.3
    • Tutorial Live-DVD
Scalasca features

• Open source, New BSD license
• Portable
  – BG/P, BG/L, IBM SP & blade clusters, Cray XT, SGI Altix, NEC SX, Solaris & Linux clusters, ...
• Supports parallel programming paradigms & languages
  – MPI, OpenMP & hybrid OpenMP/MPI
  – Fortran, C, C++
• Integrated measurement & analysis toolset
  – Runtime summarization (aka profiling)
  – Automatic event trace analysis
Scalasca 1.3 functionality

- Automatic function instrumentation (and filtering)
  - GCC, IBM, Intel, PathScale & PGI compilers
- MPI measurement & analyses
  - scalable runtime summarization & event tracing
  - only requires application executable re-linking
  - MPI File I/O operation analysis
  - Full MPI 2.2 support
  - MPI-2 RMA analysis
- OpenMP measurement & analysis
  - requires (automatic) application source instrumentation
  - runtime summaries include OpenMP metrics
  - serial event trace analysis (of merged traces)
- Hybrid OpenMP/MPI measurement & analysis
  - combined requirements/capabilities
  - parallel trace analysis requires uniform thread teams
Generic MPI application build & run

- Application code compiled & linked into executable using MPICC/CXX/FC
- Launched with MPIEXEC
- Application processes interact via MPI library
Application instrumentation

• Automatic/manual code instrumentor
• Program sources processed to add instrumentation and measurement library into application executable
• Exploits MPI standard profiling interface (PMPI) to acquire MPI events
Measurement runtime summarization

• Measurement library manages threads & events produced by instrumentation

• Measurements summarized by thread & call-path during execution

• Analysis report unified & collated at finalization

• Presentation of summary analysis
Measurement event tracing & analysis

- During measurement time-stamped events buffered for each thread
- Flushed to files along with unified definitions & maps at finalization
- Follow-up analysis replays events and produces extended analysis report
- Presentation of analysis report

Diagram:
- Program sources
  - Compiler
  - Instrumenter
  - Instrumented executable
  - Expt config
- Application + Measurement lib
  - Unified defs+maps
  - Trace 1 2 .. N
  - Parallel trace analyzer
  - Trace analysis
  - Analysis report examiner
Generic parallel tools architecture

- Automatic/manual code instrumenter
- Measurement library for runtime summary & event tracing
- Parallel (and/or serial) event trace analysis when desired
- Analysis report examiner for interactive exploration of measured execution performance properties
Scalasca toolset components

- Scalasca instrumenter = SKIN
- Scalasca measurement collector & analyzer = SCAN
- Scalasca analysis report examiner = SQUARE
EPIK

• Measurement & analysis runtime system
  – Manages runtime configuration and parallel execution
  – Configuration specified via EPIK.CONF file or environment
    • epik_conf reports current measurement configuration
  – Creates experiment archive (directory): `epik_<title>`
  – Optional runtime summarization report
  – Optional event trace generation (for later analysis)
  – Optional filtering of (compiler instrumentation) events
  – Optional incorporation of HWC measurements with events
    • via PAPI library, using PAPI preset or native counter names

• Experiment archive directory
  – Contains (single) measurement & associated files (e.g., logs)
  – Contains (subsequent) analysis reports
scalasca

• One command for everything
  
  % scalasca
  Scalasca 1.3
  Toolset for scalable performance analysis of large-scale apps
  usage: scalasca [-v][-n] {action}
  1. prepare application objects and executable for measurement:
     scalasca -instrument <compile-or-link-command>  # skin
  2. run application under control of measurement system:
     scalasca -analyze <application-launch-command>  # scan
  3. interactively explore measurement analysis report:
     scalasca -examine <experiment-archive|report>     # square

[-h] show quick reference guide (only)
scalasca actions

• One command for everything
  % scalasca -usage
  % scalasca -instrument [options] <compile-or-link-command>
  % scalasca -analyze [options] <application-launch-command>
  % scalasca -examine [options] <experiment-archive|report>

... that does nothing!
  – simply a shell script wrapper for action commands:
    % skin [options] <compile-or-link-command>
      • prepare application objects and executable for measurement
    % scan [options] <application-launch-command>
      • run application under control of measurement system
    % square [options] <experiment-archive|report>
      • interactively explore measurement analysis report
OPARI

- Automatic instrumentation of OpenMP & POMP directives via source pre-processor
  - Parallel regions, worksharing, synchronization
  - Currently limited to OpenMP 2.5
    - No special handling of guards, dynamic or nested thread teams
  - Configurable to disable instrumentation of locks, etc.
  - Typically invoked internally by instrumentation tools
- Used by Scalasca/Kojak, ompP, TAU, VampirTrace, etc.
  - Provided with Scalasca, but also available separately
    - OPARI 1.1 (October 2001)
    - OPARI 2.0 currently in development
CUBE3

• Parallel program analysis report exploration tools
  – Libraries for XML report reading & writing
  – Algebra utilities for report processing
  – GUI for interactive analysis exploration
    • requires Qt4 or wxGTK widgets library
    • can be installed independently of Scalasca instrumentor and measurement collector/analyzer, e.g., on laptop or desktop

• Used by Scalasca/Kojak, Marmot, ompP, PerfSuite, etc.
  – Provided with Scalasca, but also available separately
    • CUBE 3.3 (February 2010)
Analysis presentation and exploration

- Representation of values (severity matrix) on three hierarchical axes
  - Performance property (metric)
  - Call-tree path (program location)
  - System location (process/thread)

- Three coupled tree browsers

- CUBE3 displays severities
  - As value: for precise comparison
  - As colour: for easy identification of hotspots
  - Inclusive value when closed & exclusive value when expanded
  - Customizable via display mode
Scalasca analysis report explorer (summary)

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Selection percent</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00 Time</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>41731.81 Execution</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>60.96 MPI</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>425.44 Collective</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00 Communication</td>
<td>6851.77 Point-to-point</td>
<td>6851.77</td>
</tr>
<tr>
<td>2991.08 Collective</td>
<td>2991.08 Collective</td>
<td>2991.08</td>
</tr>
<tr>
<td>0.00 I/O</td>
<td>1300.43 Init/Exit</td>
<td>1300.43</td>
</tr>
<tr>
<td>37.54 Overhead</td>
<td>57579621 Visits</td>
<td>57579621</td>
</tr>
<tr>
<td>1536 Synchronizations</td>
<td>0 Communications</td>
<td>0 Communications</td>
</tr>
<tr>
<td>35492352 Point-to-point</td>
<td>227328 Collective</td>
<td>227328</td>
</tr>
<tr>
<td>0 Bytes transferred</td>
<td>713051766784 Point-to-point</td>
<td>713051766784</td>
</tr>
<tr>
<td>1692668944 Collective</td>
<td>6851.77 Point-to-point</td>
<td>6851.77</td>
</tr>
</tbody>
</table>

What kind of performance problem?

How is it distributed across the processes?

Where is it in the source code? In what context?
Scalasca analysis report explorer (trace)

Additional metrics determined from trace
ZeusMP2/JUMP case study

- Computational astrophysics
  - (magneto-)hydrodynamic simulations on 1-, 2- & 3-D grids
  - part of SPEC MPI2007 1.0 benchmark suite (132.zeusmp2)
  - developed by UCSD/LLNL
  - >44,000 lines Fortran 90 (in 106 source modules)
  - provided configuration scales to 512 MPI processes

- Run with 512 processes on JUMP
  - IBM p690+ eServer cluster with HPS at JSC

- Scalasca summary and trace measurements
  - ~5% measurement dilation (full instrumentation, no filtering)
  - 2 GB trace analysis in 19 seconds
  - application’s 8x8x8 grid topology automatically captured from MPI Cartesian
Scalasca summary analysis: zeusmp2 on jump

- 12.8% of time spent in MPI point-to-point communication
- 45.0% of which is on program callpath transp/ct/hsmoc
- With 23.2% std dev over 512 processes
- Lowest values in 3rd and 4th planes of the Cartesian grid
Scalasca trace analysis: zeusmp2 on jump

- MPI point-to-point communication time separated into transport and Late Sender fractions
- Late Sender situations dominate (57%)
- Distribution of transport time (43%) indicates congestion in interior of grid
UNDERSTANDING APPLICATION PERFORMANCE WITH POINT: APPLICATION CASE STUDY

Nick Nystrom, Phil Blood, Raghu Reddy, and Mahin Mahmoodi
Pittsburgh Supercomputing Center
Code Development and Optimization Process

- Choice of algorithm most important consideration (serial and parallel)
- Measurement may reveal need for new algorithm or completely different implementation rather than optimization
- Check compiler optimizations and MPI tuning parameters before code optimizations
- Serial performance should be checked/optimized before major parallel effort
Applying Performance Tools to Improve Parallel Performance of the UNRES MD code

The UNRES molecular dynamics (MD) code utilizes a carefully-derived mesoscopic protein force field to study and predict protein folding pathways by means of molecular dynamics simulations.

Structure of UNRES

• Two issues
  – Master/Slave code
    if (myrank==0)
      MD=>...=>EELEC
    else
      ERGASTULUM=>...=>EELEC
    endif
  – Significant startup time: must remove from profiling
    • Setup time: 300 sec
    • MD Time: 1 sec/step
    • Only MD time is significant for production runs, which consist of millions of steps
    • For measurement -- could run for 30,000 steps to amortize startup!
Performance Engineering: Procedure

• Serial
  – Assess overall serial performance (percent of peak)
  – Identify functions where code spends most time
  – Instrument those functions
  – Measure code performance using hardware counters
  – Identify inefficient regions of source code and cause of inefficiencies

• Parallel
  – Assess overall parallel performance (scaling)
  – Identify functions where code spends most time (this may change at high core counts)
  – Instrument those functions
  – Identify load balancing issues, serial regions
  – Identify communication bottlenecks--use tracing to help identify cause and effect
Is There a Performance Problem?

- What does it mean for a code to perform “poorly”?
- HPL on 4K cores can take a couple of hrs
- Quantum calculations involving a few atoms may take a week
- Depends on the work being done

• Where does performance need to be improved?
  - Serial performance problem?
  - Parallel performance problem?
Detecting Performance Problems

• Serial Performance: Fraction of Peak
  – 20% peak (overall) is usually decent; After that you decide how much effort it is worth
  – 80:20 rule

• Parallel Performance: Scalability
  – Does run time decrease by 2x when I use 2x cores?
    • Strong scalability
  – Does run time remain the same when I keep the amount of work per core the same?
    • Weak scalability
PerfSuite

- Great for getting overall picture of application performance
  - Easy: no need to recompile
  - Minimal overhead
  - Provides function-level information
  - Works with OpenMP
  - Available on x86, x86-64, em64t, and ia64 architectures

http://perfsuite.ncsa.uiuc.edu/
% set PSDIR=/opt/perfsuite
% source $PSDIR/bin/psenv.csh

# Use psrun on your program to generate the data,
# then use psprocess to produce an output file (default is plain text)

# First run: this will give you a summary of performance information over
# total program execution (e.g. MFLOPS)
% psrun myprog

% psprocess myprog.12345.xml > myprog.txt

# Second run: this will break down cycles spent in each function
% psrun -C -c papi_profile_cycles.xml myprog
% psprocess -e myprog myprog.67890.xml > myprog_functions.txt
UNRES: Serial Performance

Processor and System Information (abbreviated output from PerfSuite)
=======================================================================
Node CPUs : 768
Vendor : Intel
Family : Itanium 2
Clock (MHz) : 1669.001

Statistics
=======================================================================
Floating point operations per cycle........................................ 0.597
MFLOPS (cycles).............................................................. 995.801
CPU time (seconds)......................................................... 1404.675

• Theoretical peak on Itanium2: 4 FLOP/cycle *1669 MHz = 6676 MFLOPS
• UNRES getting 15% of peak on Itanium--needs serial optimization on Itanium
• Much better on Bigben (x86_64): 1720 MFLOPS, 33% peak
• Make sure compiler is inlining (-ipo needed for ifort, –Minline=reshape needed for pgf90)
UNRES: Parallel Performance

UNRES Performance: Cray XT3

- **Cores**
  - 1
  - 2
  - 4
  - 8
  - 16
  - 32
  - 64
  - 128
  - 256

- **Time steps/sec**
  - 1
  - 2
  - 4
  - 8
  - 16
  - 32
  - 64
  - 128
  - 256
  - 512
  - 1024

- **Lines**
  - Green: Bigben
  - Blue: Ideal
Performance Engineering: Procedure

• Serial
  – Assess overall serial performance (percent of peak)
  – **Identify functions where code spends most time**
  – Instrument those functions
  – Measure code performance using hardware counters
  – Identify inefficient regions of source code and cause of inefficiencies

• Parallel
  – Assess overall parallel performance (scaling)
  – **Identify functions where code spends most time** (this may change at high core counts)
  – Instrument those functions
  – Identify load balancing issues, serial regions
  – Identify communication bottlenecks—use tracing to help identify cause and effect
Which Functions are Important?

- Often a handful of functions account for 90% of the execution time.

- Make sure you are measuring the production part of your code (make sure startup time, etc. is eliminated or insignificant).

- For parallel apps, measure at high core counts – insignificant functions become significant!
### PerfSuite Function Summary

**Function Summary**

<table>
<thead>
<tr>
<th>Samples</th>
<th>Self %</th>
<th>Total %</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>2905589</td>
<td>51.98%</td>
<td>51.98%</td>
<td>eelecij</td>
</tr>
<tr>
<td>827023</td>
<td>14.79%</td>
<td>66.77%</td>
<td>egb</td>
</tr>
<tr>
<td>634107</td>
<td>11.34%</td>
<td>78.11%</td>
<td>setup_md_matrices</td>
</tr>
<tr>
<td>247353</td>
<td>4.42%</td>
<td>82.54%</td>
<td>escp</td>
</tr>
<tr>
<td>220089</td>
<td>3.94%</td>
<td>86.48%</td>
<td>etrbk3</td>
</tr>
<tr>
<td>183492</td>
<td>3.28%</td>
<td>89.76%</td>
<td>einvit</td>
</tr>
<tr>
<td>144851</td>
<td>2.59%</td>
<td>92.35%</td>
<td>banach</td>
</tr>
<tr>
<td>132058</td>
<td>2.36%</td>
<td>94.71%</td>
<td>ginv_mult</td>
</tr>
<tr>
<td>66182</td>
<td>1.18%</td>
<td>95.89%</td>
<td>multibody_hb</td>
</tr>
<tr>
<td>39495</td>
<td>0.71%</td>
<td>96.60%</td>
<td>etred3</td>
</tr>
<tr>
<td>38111</td>
<td>0.68%</td>
<td>97.28%</td>
<td>eelec</td>
</tr>
</tbody>
</table>

- Short runs include some startup functions amongst top functions.
- To eliminate startup, perform a full production run with PerfSuite.
- Can use PerfSuite during production runs due to low overhead—essentially no impact on application performance.
Performance Engineering: Procedure

• Serial
  – Assess overall serial performance (percent of peak)
  – Identify functions where code spends most time
  – Instrument key functions
  – Measure code performance using hardware counters
  – Identify inefficient regions of source code and cause of inefficiencies

• Parallel
  – Assess overall parallel performance (scaling)
  – Identify functions where code spends most time (this may change at high core counts)
  – Instrument key functions
  – Identify load balancing issues, serial regions
  – Identify communication bottlenecks--use tracing to help identify cause and effect
Instrument Key Functions

- Instrumentation: insert functions into source code to measure performance
- Pro: Gives precise information about where things happen
- Con: High overhead and perturbation of application performance
- Thus essential to only instrument important functions
TAU: Tuning and Analysis Utilities

• Useful for a more detailed analysis
  – Routine level
  – Loop level
  – Performance counters
  – Communication performance

• A more sophisticated tool
  – Performance analysis of Fortran, C, C++, Java, and Python
  – Portable: Tested on all major platforms
  – Steeper learning curve, must recompile code

http://www.cs.uoregon.edu/research/tau/home.php
General Instructions for TAU

• Use a TAU Makefile stub (even if you don’t use makefiles for your compilation)
• Use TAU scripts for compiling (tau_cc.sh tau_f90.sh)
• Example (most basic usage):

  module load tau
  setenv TAU_MAKEFILE <path>/Makefile.tau-papi-pdt-pgi
  setenv TAU_OPTIONS "-optVerbose -optKeepFiles"
  tau_f90.sh -o hello hello_mpi.f90

• Excellent “Cheat Sheet”!
  – Everything you need to know?! (Almost)
    http://www.psc.edu/general/software/packages/tau/TAU-quickref.pdf
Using TAU with Makefiles

• Fairly simple to use with well written makefiles:

  setenv TAU_MAKEFILE <path>/Makefile.tau-papi-mpi-pdt-pgi
  setenv TAU_OPTIONS "-optVerbose –optKeepFiles –optPreProcess"
  make FC=tau_f90.sh

  – run code as normal
  – run pprof (text) or paraprof (GUI) to get results
  – paraprof --pack file.pp (packs all of the profile files into one file, easy to copy back to local workstation)

• Example scenarios

  – Typically you can do cut and paste from here: http://www.cs.uoregon.edu/research/tau/docs/scenario/index.html
Tiny Routines: High Overhead

Before:

```plaintext
double precision function scalar(u,v)
double precision u(3),v(3)
    scalar=u(1)*v(1)+u(2)*v(2)+u(3)*v(3)
return
end
```

After:

```plaintext
double precision function scalar(u,v)
double precision u(3),v(3)
call TAU_PROFILE_TIMER(profiler, 'SCALAR [...]')
call TAU_PROFILE_START(profiler)
scalar=u(1)*v(1)+u(2)*v(2)+u(3)*v(3)
call TAU_PROFILE_STOP(profiler)
return
call TAU_PROFILE_STOP(profiler)
end
```
Reducing Overhead

Overhead (time in sec):

MD steps base: 51.4 seconds
MD steps with TAU: 315 seconds

Must reduce overhead to get meaningful results:

- In paraprof go to “File” and select “Create Selective Instrumentation File”
TAU automatically generates a list of routines that you can save to a selective instrumentation file.
Selective Instrumentation File

• Automatically generated file essentially eliminates overhead in instrumented UNRES

• In addition to eliminating overhead, use this to specify:
  – Files to include/exclude
  – Routines to include/exclude
  – Directives for loop instrumentation
  – Phase definitions

• Specify the file in TAU_OPTIONS and recompile:

  setenv TAU_OPTIONS "-optVerbose –optKeepFiles
  –optPreProcess -optTauSelectFile=select .tau"

• http://www.cs.uoregon.edu/research/tau/docs/newguide/bk03ch01.html
Key UNRES Functions in TAU (with Startup Time)
Phase Profiling: Isolate regions of code execution

- Eliminated overhead, now we need to deal with startup time
  - Choose a region of the code of interest: e.g. the main computational kernel
  - Determine where in the code that region begins and ends (call path can be helpful)
  - Then put something like this in selective instrumentation file:
    ```
    static phase name="foo1_bar" file="foo.c" line=26 to line=27
    ```
  - Recompile and rerun
Key UNRES Functions (MD Time Only)

Phase: PHASE_MD
Metric: TIME
Value: Exclusive
Units: seconds

6.100

<table>
<thead>
<tr>
<th>Function</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EESC [energy_p_new_barrier.pp.F]</td>
<td>2204.7</td>
</tr>
<tr>
<td>SUM_GRADIENT [energy_p_new_barrier.pp.F]</td>
<td>2172.9</td>
</tr>
<tr>
<td>MULTIBODY_HB [energy_p_new_barrier.pp.F]</td>
<td>4924.9</td>
</tr>
<tr>
<td>INT_FROM_CART [checkder pp.F]</td>
<td>4935.7</td>
</tr>
<tr>
<td>ZEROGRAD [gradient.pp.F]</td>
<td>3387.9</td>
</tr>
<tr>
<td>SET_MATRICES [energy_p_new_barrier.pp.F]</td>
<td>2202.9</td>
</tr>
<tr>
<td>MPI_Reduce()</td>
<td>0.138</td>
</tr>
<tr>
<td>MPI_Allreduce()</td>
<td>0.135</td>
</tr>
<tr>
<td>MPI_Waitall()</td>
<td>0.135</td>
</tr>
<tr>
<td>VEC_AND_DERIV [energy_p_new_barrier.pp.F]</td>
<td>1918.9</td>
</tr>
<tr>
<td>MPI_Bcast()</td>
<td>0.084</td>
</tr>
<tr>
<td>ESSEND [energy_p_new_barrier.pp.F]</td>
<td>3926.9</td>
</tr>
<tr>
<td>ETURN4 [energy_p_new_barrier.pp.F]</td>
<td>3252.9</td>
</tr>
<tr>
<td>MPI_Scatter()</td>
<td>0.036</td>
</tr>
<tr>
<td>ADD_HB_CONTACT [energy_p_new_barrier.pp.F]</td>
<td>961.9</td>
</tr>
<tr>
<td>MPI_Isend()</td>
<td>0.027</td>
</tr>
<tr>
<td>ETURN3 [energy_p_new_barrier.pp.F]</td>
<td>4923.7</td>
</tr>
<tr>
<td>ESC [energy_p_new_barrier.pp.F]</td>
<td>4106.0</td>
</tr>
<tr>
<td>CHAINBUILD_CART [inccartderiv.pp.F]</td>
<td>331.9</td>
</tr>
<tr>
<td>MPI_irecv()</td>
<td>0.017</td>
</tr>
<tr>
<td>PHASE_MD</td>
<td>0.014</td>
</tr>
<tr>
<td>INTCARTDERIV [inccartderiv.pp.F]</td>
<td>113.9</td>
</tr>
</tbody>
</table>
Performance Engineering: Procedure

• Serial
  – Assess overall serial performance (percent of peak)
  – Identify functions where code spends most time
  – Instrument those functions
  – Measure code performance using hardware counters
  – Identify inefficient regions of source code and cause of inefficiencies

• Parallel
  – Assess overall parallel performance (scaling)
  – Identify functions where code spends most time (this may change at high core counts)
  – Instrument those functions
  – Identify load balancing issues, serial regions
  – Identify communication bottlenecks--use tracing to help identify cause and effect
Getting Performance Counter Information

• Compile with TAU as described before
  – With selective instrumentation (to eliminate overhead)
  – With phases designated (defining your region of interest)
• At run time:

setenv TAU_METRICS GET_TIME_OF_DAY:PAPI_FP_OPS:PAPI_TOT_CYC

• Requires that PAPI be installed on your system
• Run ‘papi_avail’ or ‘papi_native_avail’ on compute nodes to get names of PAPI counters
Example: Measuring FLOP/cycle (peak is 2)

Most important UNRES routines perform well
Performance Engineering: Procedure

• Serial
  – Assess overall serial performance (percent of peak)
  – Identify functions where code spends most time
  – Instrument those functions
  – Measure code performance using hardware counters
  – Identify inefficient regions of source code and cause of inefficiencies

• Parallel
  – Assess overall parallel performance (scaling)
  – Identify functions where code spends most time (this may change at high core counts)
  – Instrument those functions
  – Identify load balancing issues, serial regions
  – Identify communication bottlenecks--use tracing to help identify cause and effect
Detecting Parallel Performance Issues: Serial Bottlenecks

• To identify scaling bottlenecks, do the following for each run in a scaling study (e.g. 2-64 cores):
  1) In Paraprof manager right-click “Default Exp” and select “Add Trial”. Find packed profile file and add it.
  2) If you defined a phase, from main paraprof window select: Windows -> Function Legend-> Filter->Advanced Filtering
  3) Type in the name of the phase you defined, and click ‘OK’
  4) Return to Paraprof manager, right-click the name of the trial, and select “Add to Mean Comparison Window”

• Compare functions across increasing core counts
**Serial Bottleneck Detection in UNRES**

- Examine timings of functions in your region of interest as you scale up.
- Identify functions that do not scale well or that need to be parallelized.
- Find communication routines that are starting to dominate runtime.
- **Caution:** Looking at mean execution time will not reveal some scaling problems (load imbalance).

**Serial function begins to dominate runtime**

---

**Diagram:**

- **08.803**
  - 38.03 (56.146%)
- **38.254**
  - 30.251 (53.591%)
  - 10.482 (26.677%)
  - 5.144 (13.443%)
  - 2.306 (6.364%)
- **11.437**
  - 0.129 (3.393%)
  - 3.016 (26.895%)
  - 1.530 (13.477%)
  - 0.716 (6.257%)
- **11.105**
  - 4.689 (41.327%)
  - 1.425 (12.835%)
  - 0.056 (5.356%)
  - 0.373 (3.392%)
- **9.043**
  - 4.853 (53.64%)
  - 2.44 (26.872%)
  - 1.225 (13.544%)
  - 0.572 (3.321%)
- **1.463**
  - 1.597 (109.159%)
  - 1.652 (103.52%)
  - 1.8 (109.365%)
  - 1.495 (101.523%)
- **1.409**
  - 1.229 (87.25%)
  - 0.927 (65.839%)
  - 0.8 (66.775%)

**Legend:**

- **PHASE_MD <= PROGRAM**
- **EELEC <= PHASE_MD**
- **ESCP <= PHASE_MD**
- **MPI_Send0 <= PHASE_MD**
- **EGS <= PHASE_MD**
- **G_INV_MULT <= PHASE_MD**
- **MPI_AllgatherV <= PHASE_MD**

---

**LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes**
Detecting Parallel Performance Issues: Load Imbalance

• Examine timings of functions in your region of interest
  – If you defined a phase, from paraprof window, right-click on phase name and select: ‘Show profile for this phase’

• To look at load imbalance in a particular function:
  – Left-click on function name to look at timings across all processors

• To look at load imbalance across all functions:
  – In Paraprof window go to ‘Options’
  – Uncheck ‘Normalize’ and ‘Stack Bars Together’
Load Imbalance Detection in UNRES

Only looking at time spent in the important MD phase

- In this case: Developers unaware that chosen algorithm would create load imbalance
- Reexamined available algorithms and found one with much better load balance – also faster in serial!
- Also parallelized serial function causing bottleneck

LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes
Major Serial Bottleneck and Load Imbalance in UNRES Eliminated

- Due to 4x faster serial algorithm the balance between computation and communication has shifted – communication must be more efficient to scale well

- Code is undergoing another round of profiling and optimization
• Load imbalance on one processor apparently causing other processors to idle in MPI_Barrier
Performance Engineering: Procedure

• Serial
  – Assess overall serial performance (percent of peak)
  – Identify functions where code spends most time
  – Instrument those functions
  – Measure code performance using hardware counters
  – Identify inefficient regions of source code and cause of inefficiencies

• Parallel
  – Assess overall parallel performance (scaling)
  – Identify functions where code spends most time (this may change at high core counts)
  – Instrument those functions
  – Identify load balancing issues, serial regions
  – Identify communication bottlenecks--use tracing to help identify cause and effect
Getting a Call Path with TAU

• Why do I need this?
  – To optimize a routine, you often need to know what is above and below it
  – e.g. Determine which routines make significant MPI calls

• To get callpath info, do the following at runtime:
  setenv TAU_CALLPATH 1 (this enables callpath)
  setenv TAU_CALLPATH_DEPTH 50 (defines depth)
Use call path information to find routines from which key MPI calls are made. Include these routines in tracing experiment.

To show source locations select: File -> Preferences
Generating a Trace

• Creating TAU trace and analyzing in Vampir
  – At runtime: setenv TAU_TRACE 1
  – After the run, in the trace directory do:
    tau_treemerge.pl
tau2otf tau.trc tau.edf mytrace.otf
  – Creates three files: .otf, .def, .events
  – Run Vampir in the same directory:
vampir mytrace.otf

• Insight into causes of communication bottlenecks
  – Duration of individual MPI calls
  – Unnecessary use of blocking calls
  – Posting MPI calls too early or too late
  – Opportunities to overlap computation and communication
TAU Trace of UNRES Timesteps in Vampir

Repeating pattern for each timestep

LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes

349
Focus on “problem” functions: MPI_Barrier (green) and Multibody_HB (yellow)

As observed in profile, last process is taking longer in Multibody_HB

In addition, trace reveals that Multibody_HB always starts early on process 0 and late on the last process
Issues with Tracing

• At high processor counts the amount of data becomes overwhelming
• Very selective instrumentation is critical to manage data
• Also need to isolate the computational kernel and trace for minimum number of iterations to see patterns
• Complexity of manually analyzing traces on thousands of processors is an issue
• SCALASCA does automated analysis of traces to determine communication problems
Some Take-Home Points

• Good choices for algorithms (serial and parallel) are most important
• Performance measurement can help you determine if algorithm and implementation is good
• Do compiler and MPI parameter optimizations first
• Check/optimize serial performance before investing a lot of time in improving scaling
• Choose the right tool for the job
• Know when to stop: 80:20 rule
• PSC staff and tool developers collaborate with code developers to help with performance engineering of parallel codes

LCI 2010: Using POINT Performance Tools to Understand and Optimize Multicore Codes
Additional Resources: Obtaining Advanced Support

• Advanced Support Program (ASP)
  – associates one or more staff members agreed-upon level of effort with a research project for a sustained period of collaboration
  – can be requested with all types of resource allocations
  – [Link](http://www.teragrid.org/userinfo/asp.php)

• NSF Resource Providers & other HPC centers
  – intermediate to advanced workshops address system-specific optimizations
  – computational science staff

• POINT Application Engagement
  – Contact the POINT team at [point@nic.uoregon.edu](mailto:point@nic.uoregon.edu)