# Productive Performance Engineering of Petascale Applications with POINT and VI-HPS

<table>
<thead>
<tr>
<th>Name</th>
<th>Institution</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allen D. Malony</td>
<td>University of Oregon</td>
<td><a href="mailto:malony@cs.uoregon.edu">malony@cs.uoregon.edu</a></td>
</tr>
<tr>
<td>Felix Wolf</td>
<td>Jülich Supercomputing Centre</td>
<td><a href="mailto:f.wolf@fz-juelich.de">f.wolf@fz-juelich.de</a></td>
</tr>
<tr>
<td>Shirley Moore</td>
<td>Innovative Computing Laboratory</td>
<td><a href="mailto:shirley@eecs.utk.edu">shirley@eecs.utk.edu</a></td>
</tr>
<tr>
<td>Rick Kufrin</td>
<td>National Center for Supercomputing Applications</td>
<td><a href="mailto:rkufrin@illinois.edu">rkufrin@illinois.edu</a></td>
</tr>
<tr>
<td>Sameer Shende</td>
<td>University of Oregon</td>
<td><a href="mailto:sameer@cs.uoregon.edu">sameer@cs.uoregon.edu</a></td>
</tr>
<tr>
<td>Brian Wylie</td>
<td>Jülich Supercomputing Centre</td>
<td><a href="mailto:b.wylie@fz-juelich.de">b.wylie@fz-juelich.de</a></td>
</tr>
<tr>
<td>Andreas Knüpfer</td>
<td>Technische Universität Dresden</td>
<td><a href="mailto:andreas.knuepfer@tu-dresden.de">andreas.knuepfer@tu-dresden.de</a></td>
</tr>
<tr>
<td>Nick Nystrom</td>
<td>Pittsburgh Supercomputing Center</td>
<td><a href="mailto:nystrom@psc.edu">nystrom@psc.edu</a></td>
</tr>
</tbody>
</table>
## Tutorial Agenda

<table>
<thead>
<tr>
<th>Time</th>
<th>Topic</th>
<th>Speaker</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:30 – 9:00</td>
<td>Introduction to POINT and VI-HPS</td>
<td>Malony, Wolf</td>
</tr>
<tr>
<td>9:00 – 9:45</td>
<td>Introduction to Performance Engineering</td>
<td>Malony</td>
</tr>
<tr>
<td>9:45 – 10:00</td>
<td>POINT/VI-HPS LiveDVD</td>
<td>All</td>
</tr>
<tr>
<td>10:00 -10:30</td>
<td>Break</td>
<td></td>
</tr>
<tr>
<td>10:30 – 11:00</td>
<td>PAPI</td>
<td>Moore</td>
</tr>
<tr>
<td>11:00 – 11:30</td>
<td>PerfSuite</td>
<td>Kufrin</td>
</tr>
<tr>
<td>11:30 – 12:00</td>
<td>TAU</td>
<td>Shende</td>
</tr>
<tr>
<td>12:00 – 1:30</td>
<td>Lunch</td>
<td></td>
</tr>
<tr>
<td>1:30 – 2:00</td>
<td>TAU, continued</td>
<td>Shende</td>
</tr>
<tr>
<td>2:00 – 3:00</td>
<td>Scalasca</td>
<td>Wylie</td>
</tr>
<tr>
<td>3:00 – 3:30</td>
<td>Break</td>
<td></td>
</tr>
<tr>
<td>3:30 – 4:00</td>
<td>Vampir/VampirTrace</td>
<td>Knüpfer</td>
</tr>
<tr>
<td>4:30 – 5:00</td>
<td>Application Case Studies</td>
<td>Nystrom</td>
</tr>
</tbody>
</table>

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
INTRODUCTION TO POINT AND VI-HPS

Allen D. Malony
Performance Research Laboratory
University of Oregon

Felix Wolf
Jülich Supercomputing Centre
POINT Project

• “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

• POINT project
  – Petascale Productivity from Open, Integrated Tools
  – 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

Binary and dynamic instrumentation

Library interposition

Runtime control

Scalable profile and trace analysis

Scalable performance visualization

Automatic performance diagnosis

Performance data mining

Application and workload characterization

Memory access / usage analysis

Measurement

Hardware counter access

Parallel profiling

Statistical profiling

Scalable trace generation

Memory and I/O measurement

Kernel–level measurement

Performance database

Web–based portal

Automatic configuration

Testing and validation

Management

Performance Engineering Process

Level 2 (Advanced)

Level 1 (Intermediate)

Level 0 (Entry)

Testbed Apps
ENZO
NAMD
NEMO3D
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
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
- MARMOT
- SCALASCA
- VAMPIR

- Hardware monitoring
- Automatic trace analysis
- Visual trace analysis
- Error correction
- Execution
- Optimization

- POINT
- VI-HPS
Key tool components also provided as open-source
  – Program/library instrumentation
    • OPARI, POMP
  – MPI library/tool integration
    • UniMCI
  – Scalable I/O
    • SIONlib
  – Libraries & tools for handling (and converting) traces
    • PEARL, EARL, EPILOG, OTF, Œ
  – Analysis algebra & hierarchical/topological presentation
    • CUBE
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
VI-HPS Training & Tuning Workshops

• Goals
  – Give an overview of the programming tools suite
  – Explain the functionality of individual tools
  – Teach how to use the tools effectively
  – Offer hands-on experience and expert assistance using tools
  – Receive feedback from users to guide future development
• For best results, bring & analyse/tune your own code(s)!

• VI-HPS Tuning Workshop series
  – Aachen (Mar'08), Dresden (Oct'08), Jülich (Feb'09), Bremen (Sep'09), ...
• Joint POINT/VI-HPS Tutorial series
  – Austin/SC (Nov'08), Baton Rouge/ICCS (May'09), Portland/SC (Nov‘09), ...
• Training with individual tools & platforms (e.g., BlueGene)
INTRODUCTION TO PERFORMANCE ENGINEERING

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
- hypotheses

Performance Diagnosis
- properties

Performance Experimentation
- characterization

Performance Observation

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
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
• 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(EXIT, 2);
    ...
    trace(RECV, A);
    ...
    trace(EXIT, 2);
}
```

MONITOR

|   | master | worker | ...
|---|--------|--------|---
| 1 | A      | ENTER  | 1
| 2 | B      | ENTER  | 2
| 3 | A      | SEND   | B
| 4 | A      | EXIT   | 1
| 5 | B      | RECV   | A
| 6 | B      | EXIT   | 2
|   | ...    | ...    |...

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
Tracing Analysis and Visualization

<table>
<thead>
<tr>
<th></th>
<th>master</th>
<th></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>

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

SC '09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
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

[Diagram showing the process of build, execute, and simple analysis with feedback, linked to a performance database and offline analysis, resulting in 72% faster performance.]
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
Metadata Collection

• 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
Metadata and Knowledge Role

Context Knowledge

Source Code  Build Environment  Run Environment

Execution

Performance Knowledge

Application  Machine

Performance Problems

You have to capture these...

...to understand this

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS

53
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
HANSDS-ON:
LIVE-DVD AND NPB-MPI BT

POINT/VI-HPS Teams
NPB-MPI suite

• 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

```bash
% 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                               *
***************************************************************
Building an NPB benchmark

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

% make 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 -0 -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_sub.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

```bash
% 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
# 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 prepposition macro for instrumentation preparation

#MPIF77 = $(PREP) mpif77

# This links MPI Fortran programs; usually the same as ${MPIF77}

FLINK = $(MPIF77)

...
PAPI

Shirley Moore, Dan Terpstra
Innovative Computing Lab
University of Tennessee, Knoxville
Hardware Counters

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, SiCortex ...
PAPI Counter Interfaces

PAPI provides 3 interfaces to the underlying counter hardware:

1. A Low Level API manages hardware events in user defined groups called EventSets, and provides access to advanced features.
2. A High Level API provides the ability to start, stop and read the counters for a specified list of events.
3. Graphical and end-user tools provide facile data collection and visualization.
## PAPI Preset Events

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

### Level 1 Cache

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_L1_DCH</td>
<td>Level 1 data cache hits</td>
</tr>
<tr>
<td>PAPI_L1_DCA</td>
<td>Level 1 data cache accesses</td>
</tr>
<tr>
<td>PAPI_L1_DCR</td>
<td>Level 1 data cache reads</td>
</tr>
<tr>
<td>PAPI_L1_DCW</td>
<td>Level 1 data cache writes</td>
</tr>
<tr>
<td>PAPI_L1_DCM</td>
<td>Level 1 data cache misses</td>
</tr>
<tr>
<td>PAPI_L1_ICH</td>
<td>Level 1 instruction cache hits</td>
</tr>
<tr>
<td>PAPI_L1_ICA</td>
<td>Level 1 instruction cache accesses</td>
</tr>
<tr>
<td>PAPI_L1_ICR</td>
<td>Level 1 instruction cache reads</td>
</tr>
<tr>
<td>PAPI_L1_ICW</td>
<td>Level 1 instruction cache writes</td>
</tr>
<tr>
<td>PAPI_L1_ICM</td>
<td>Level 1 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_L1_TCH</td>
<td>Level 1 total cache hits</td>
</tr>
<tr>
<td>PAPI_L1_TCA</td>
<td>Level 1 total cache accesses</td>
</tr>
<tr>
<td>PAPI_L1_TCR</td>
<td>Level 1 total cache reads</td>
</tr>
<tr>
<td>PAPI_L1_TCW</td>
<td>Level 1 total cache writes</td>
</tr>
<tr>
<td>PAPI_L1_TCM</td>
<td>Level 1 total cache misses</td>
</tr>
<tr>
<td>PAPI_L1_LDM</td>
<td>Level 1 load misses</td>
</tr>
<tr>
<td>PAPI_L1_STM</td>
<td>Level 1 store misses</td>
</tr>
</tbody>
</table>

### Level 3 Cache

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_L3_DCH</td>
<td>Level 1 data cache hits</td>
</tr>
<tr>
<td>PAPI_L3_DCA</td>
<td>Level 1 data cache accesses</td>
</tr>
<tr>
<td>PAPI_L3_DCR</td>
<td>Level 1 data cache reads</td>
</tr>
<tr>
<td>PAPI_L3_DCW</td>
<td>Level 1 data cache writes</td>
</tr>
<tr>
<td>PAPI_L3_DCM</td>
<td>Level 1 data cache misses</td>
</tr>
<tr>
<td>PAPI_L3_ICH</td>
<td>Level 1 instruction cache hits</td>
</tr>
<tr>
<td>PAPI_L3_ICA</td>
<td>Level 1 instruction cache accesses</td>
</tr>
<tr>
<td>PAPI_L3_ICR</td>
<td>Level 1 instruction cache reads</td>
</tr>
<tr>
<td>PAPI_L3_ICW</td>
<td>Level 1 instruction cache writes</td>
</tr>
<tr>
<td>PAPI_L3_ICM</td>
<td>Level 1 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_L3_TCH</td>
<td>Level 1 total cache hits</td>
</tr>
<tr>
<td>PAPI_L3_TCA</td>
<td>Level 1 total cache accesses</td>
</tr>
<tr>
<td>PAPI_L3_TCR</td>
<td>Level 1 total cache reads</td>
</tr>
<tr>
<td>PAPI_L3_TCW</td>
<td>Level 1 total cache writes</td>
</tr>
<tr>
<td>PAPI_L3_TCM</td>
<td>Level 1 total cache misses</td>
</tr>
<tr>
<td>PAPI_L3_LDM</td>
<td>Level 1 load misses</td>
</tr>
<tr>
<td>PAPI_L3_STM</td>
<td>Level 1 store misses</td>
</tr>
</tbody>
</table>

### Cache Sharing

- **PAPI_CA_SNP:** Requests for a snoop
- **PAPI_CA_SHR:** Requests for exclusive access to shared cache line

---

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
PAPI Native Events

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

• Use `papi_event_chooser` utility to select a compatible set of events

```c
{ .pme_name = "L2 ST",      .pme_code = 0x2a,
  .pme_flags = PPMLIB_CORE_CSPEC,
  .pme_desc = "L2 store requests",
  .pme_umasks = {
    { .pme_uname = "MESI",
      .pme_udesc = "Any cacheline access",
      .pme_ucode = 0xf
    },
    { .pme_uname = "I_STATE",
      .pme_udesc = "Invalid cacheline",
      .pme_ucode = 0x1
    },
    { .pme_uname = "S_STATE",
      .pme_udesc = "Shared cacheline",
      .pme_ucode = 0x2
    },
    { .pme_uname = "E_STATE",
      .pme_udesc = "Exclusive cacheline",
      .pme_ucode = 0x4
    },
    { .pme_uname = "M_STATE",
      .pme_udesc = "Modified cacheline",
      .pme_ucode = 0x8
    },
    { .pme_uname = "SELF",
      .pme_udesc = "This core",
      .pme_ucode = 0x40
    },
    { .pme_uname = "BOTH_CORES",
      .pme_udesc = "Both cores",
      .pme_ucode = 0xc0
    }
  },
  .pme_numasks = 7
},
```
PAPI High-level Interface

- Meant for application programmers wanting coarse-grained measurements
- Calls the lower level API
- Allows only PAPI preset events
- Easier to use and less setup (less additional code) than low-level
- Supports 8 calls in C or Fortran:
  
  - `PAPI_start_counters`
  - `PAPI_stop_counters`
  - `PAPI_read_counters`
  - `PAPI_accum_counters`
  - `PAPI_num_counters`
  - `PAPI_ipc`
  - `PAPI_flips`
  - `PAPI_flops`
#include "papi.h"
#define NUM_EVENTS 2
long long values[NUM_EVENTS];
unsigned int
  Events[NUM_EVENTS]={PAPI_TOT_INS,PAPI_TOT_CYC};

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

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

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

- Increased efficiency and functionality over the high level PAPI interface
- Obtain information about the executable, the hardware, and the memory environment
- Multiplexing
- Callbacks on counter overflow
- Profiling
- About 60 functions
#include "papi.h"
#define NUM_EVENTS 2
int Events[NUM_EVENTS]={PAPI_FP_INS,PAPI_TOT_CYC};
int EventSet;
long_long values[NUM_EVENTS];
/* Initialize the Library */
retval = PAPI_library_init(PAPI_VER_CURRENT);
/* Allocate space for the new eventset and do setup */
retval = PAPI_create_eventset(&EventSet);
/* Add Flops and total cycles to the eventset */
retval = PAPI_add_events(EventSet,Events(NUM_EVENTS);
/* Start the counters */
retval = PAPI_start(EventSet);

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

/*Stop counters and store results in values */
retval = PAPI_stop(EventSet,values);
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/\sim140 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 for "Uncore" events
    - Require global, not process level access
    - 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
Motivation

• Performance counters also exist in off-CPU resources
• All information is valuable for performance optimization
• Increasing cpu counts & power demands place greater importance on:
  – Thermal health and management
  – Power consumption
• Multicore systems require careful resource balancing
• Higher processor & core counts make communications metrics more critical:
  – Bandwidth
  – Latency
  – Dropped packets
  – Bytes transferred
Limitations

• Interfaces are often obscure, unexposed or non-standard
• Performance data (e.g. accelerators) can be vastly different than cpus
• Measurements are usually system-wide and asynchronous
  – May not matter on dedicated single-task OS’s like Cray Catamount or CLE and Blue Gene CNK
  – But matters more for Multicore
• Often very different time scales
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
PAPI PROVIDE PERFORMANCE MONITORING API

Monolithic ‘PAPI Classic’

PAPI PORTABLE LAYER

PAPI HARDWARE SPECIFIC LAYER

- Kernel Extension
- Operating System
- Perf Counter Hardware

Low Level User API

High Level User API

SC ‘09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
Component **PAPI**

- **PAPI FRAMEWORK**
  - **Low Level User API**
  - **High Level User API**
  - **Developer API**

**PAPI COMPONENT (NETWORK)**
- Operating System
- Counter Hardware

**PAPI COMPONENT (CPU)**
- Operating System
- Counter Hardware

**PAPI COMPONENT (THERMAL)**
- Operating System
- Counter Hardware

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS

83
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
<table>
<thead>
<tr>
<th>Myrinet MX Counters</th>
</tr>
</thead>
<tbody>
<tr>
<td>LANAI_UPTIME</td>
</tr>
<tr>
<td>COUNTERS_UPTIME</td>
</tr>
<tr>
<td>BAD_CRC8</td>
</tr>
<tr>
<td>BAD_CRC32</td>
</tr>
<tr>
<td>UNSTRIPPED_ROUTE</td>
</tr>
<tr>
<td>PKT_DESC_INVALID</td>
</tr>
<tr>
<td>RECV_PKT_ERRORS</td>
</tr>
<tr>
<td>PKT_MISROUTED</td>
</tr>
<tr>
<td>DATA_SRC_UNKNOWN</td>
</tr>
<tr>
<td>DATA_ENDPT_CLOSED</td>
</tr>
<tr>
<td>DATA_BAD_SESSION</td>
</tr>
<tr>
<td>PUSH_BAD_WINDOW</td>
</tr>
<tr>
<td>PUSH_DUPLICATE</td>
</tr>
<tr>
<td>PUSH_OBSOLETE</td>
</tr>
<tr>
<td>PUSH_RACE_DRIVER</td>
</tr>
<tr>
<td>PUSH_BAD_SEND_HANDLE</td>
</tr>
<tr>
<td>PUSH_BAD_SEND_MAGIC</td>
</tr>
<tr>
<td>PULL_OBSOLETE</td>
</tr>
<tr>
<td>PULL_NOTIFY_OBSOLETE</td>
</tr>
<tr>
<td>PULL_RACE_DRIVER</td>
</tr>
<tr>
<td>ACK_BAD_TYPE</td>
</tr>
<tr>
<td>ACK_BAD_MAGIC</td>
</tr>
<tr>
<td>ACK_RESEND_RACE</td>
</tr>
<tr>
<td>LATE_ACK</td>
</tr>
<tr>
<td>ACK_NACK_FRAMES_IN_PIPE</td>
</tr>
<tr>
<td>NACK_BAD_ENDPT</td>
</tr>
<tr>
<td>NACK_ENDPT_CLOSED</td>
</tr>
<tr>
<td>NACK_BAD_SESSION</td>
</tr>
<tr>
<td>NACK_BAD_RDMAWIN</td>
</tr>
<tr>
<td>NACK_EVENTQ_FULL</td>
</tr>
<tr>
<td>SEND_BAD_RDMAWIN</td>
</tr>
<tr>
<td>CONNECT_TIMEOUT</td>
</tr>
<tr>
<td>CONNECT_SRC_UNKNOWN</td>
</tr>
<tr>
<td>QUERY_BAD_MAGIC</td>
</tr>
<tr>
<td>QUERY_TIMED_OUT</td>
</tr>
<tr>
<td>QUERY_SRC_UNKNOWN</td>
</tr>
<tr>
<td>RAW SENDS</td>
</tr>
<tr>
<td>RAW RECEIVES</td>
</tr>
<tr>
<td>RAW_OVERSIZED_PACKETS</td>
</tr>
<tr>
<td>RAW_RECV_OVERRUN</td>
</tr>
<tr>
<td>RAW_DISABLED</td>
</tr>
<tr>
<td>CONNECT_SEND</td>
</tr>
<tr>
<td>CONNECT_RECV</td>
</tr>
<tr>
<td>ACK_SEND</td>
</tr>
<tr>
<td>ACK_RECV</td>
</tr>
<tr>
<td>PUSH_SEND</td>
</tr>
<tr>
<td>PUSH_RECV</td>
</tr>
<tr>
<td>QUERY_SEND</td>
</tr>
<tr>
<td>QUERY_RECV</td>
</tr>
<tr>
<td>REPLY_SEND</td>
</tr>
<tr>
<td>REPLY_RECV</td>
</tr>
<tr>
<td>QUERY_UNKNOWN</td>
</tr>
<tr>
<td>DATA_SEND_NULL</td>
</tr>
<tr>
<td>DATA_SEND_SMALL</td>
</tr>
<tr>
<td>DATA_SEND_MEDIUM</td>
</tr>
<tr>
<td>DATA_SEND_RNDV</td>
</tr>
<tr>
<td>DATA_SEND_PULL</td>
</tr>
<tr>
<td>DATA_RECV_NULL</td>
</tr>
<tr>
<td>DATA_RECV_SMALL_INLINE</td>
</tr>
<tr>
<td>DATA_RECV_SMALL_COPY</td>
</tr>
<tr>
<td>DATA_RECV_MEDIUM</td>
</tr>
<tr>
<td>DATA_RECV_RNDV</td>
</tr>
<tr>
<td>DATA_RECV_PULL</td>
</tr>
<tr>
<td>ETHER_SEND_UNICAST_CNT</td>
</tr>
<tr>
<td>ETHER_SEND_MULTICAST_CNT</td>
</tr>
<tr>
<td>ETHER_RECV_SMALL_CNT</td>
</tr>
<tr>
<td>ETHER_RECV_BIG_CNT</td>
</tr>
<tr>
<td>ETHER_OVERRUN</td>
</tr>
<tr>
<td>ETHER_OVERSIZED</td>
</tr>
<tr>
<td>DATA_RECV_NO_CREDITS</td>
</tr>
<tr>
<td>PACKETS_RESENT</td>
</tr>
<tr>
<td>PACKETS_DROPPED</td>
</tr>
<tr>
<td>MAPPER_ROUTES_UPDATE</td>
</tr>
<tr>
<td>ROUTE_DISPERSION</td>
</tr>
<tr>
<td>OUT_OF_SEND_HANDLES</td>
</tr>
<tr>
<td>OUT_OF_PULL_HANDLES</td>
</tr>
<tr>
<td>OUT_OF_PUSH_HANDLES</td>
</tr>
<tr>
<td>MEDIUM_CONT_RACE</td>
</tr>
<tr>
<td>CMD_TYPE_UNKNOWN</td>
</tr>
<tr>
<td>UREQ_TYPE_UNKNOWN</td>
</tr>
<tr>
<td>INTERRUPTS_OVERRUN</td>
</tr>
<tr>
<td>WAITING_FOR_INTERRUPT_DMA</td>
</tr>
<tr>
<td>WAITING_FOR_INTERRUPT_ACK</td>
</tr>
<tr>
<td>WAITING_FOR_INTERRUPT_TIMER</td>
</tr>
<tr>
<td>SLABS_RECYCLING</td>
</tr>
<tr>
<td>SLABS_PRESSURE</td>
</tr>
<tr>
<td>SLABS_STARVATION</td>
</tr>
<tr>
<td>OUT_OF_RDMA_HANDLES</td>
</tr>
<tr>
<td>EVENTQ_FULL</td>
</tr>
<tr>
<td>BUFFER_DROP</td>
</tr>
<tr>
<td>MEMORY_DROP</td>
</tr>
<tr>
<td>HARDWARE_FLOW_CONTROL</td>
</tr>
<tr>
<td>SIMULATED_PACKETS_LOST</td>
</tr>
<tr>
<td>LOGGING.FRAMES_DUMPED</td>
</tr>
<tr>
<td>WAKE_INTERRUPTS</td>
</tr>
<tr>
<td>AVERTED_WAKEUP_RACE</td>
</tr>
<tr>
<td>DMA_METADATA_RACE</td>
</tr>
</tbody>
</table>

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
## Myrinet MX Counters

<table>
<thead>
<tr>
<th>Counters</th>
<th>Counters</th>
</tr>
</thead>
<tbody>
<tr>
<td>LANAI_UPTIME</td>
<td>COUNTERS_UPTIME</td>
</tr>
<tr>
<td>BAD_CRC8</td>
<td>BAD_CRC32</td>
</tr>
<tr>
<td>UNSTRIPPED_ROUTE</td>
<td>PKT_DESC_INVALID</td>
</tr>
<tr>
<td>PKT_MISROUTED</td>
<td>RECV_PKT_ERRORS</td>
</tr>
<tr>
<td>DATA_SRC_UNKNOWN</td>
<td>DATA_ENDPT_CLOSED</td>
</tr>
<tr>
<td>DATA_BAD_SESSION</td>
<td>PUSH_BAD_WINDOW</td>
</tr>
<tr>
<td>PUSH_DUPLICATE</td>
<td>PUSH_OBSOLETE</td>
</tr>
<tr>
<td>PUSH_RACE_DRIVER</td>
<td>PUSH_BAD_SEND_HANDLE_MAGIC</td>
</tr>
<tr>
<td>RAW_SENDS</td>
<td>RAW_RECEIVES</td>
</tr>
<tr>
<td>RAW_OVERSIZED_PACKETS</td>
<td>RAW_RECV_OVERRUN</td>
</tr>
<tr>
<td>RAW_DISABLED</td>
<td>CONNECT_SEND</td>
</tr>
<tr>
<td>DATA_SEND_NULL</td>
<td>DATA_SEND_RNDV</td>
</tr>
<tr>
<td>DATA_RECV_NO_CREDITS</td>
<td>PACKETS_RESENT</td>
</tr>
<tr>
<td>PACKETS_DROPPED</td>
<td>MAPPER_ROUTES_UPDATE</td>
</tr>
<tr>
<td>ROUTE_DISPERSION</td>
<td>OUT_OF_SEND_HANDLES</td>
</tr>
<tr>
<td>OUT_OF_PULL_HANDLES</td>
<td>OUT_OF_PUSH_HANDLES</td>
</tr>
<tr>
<td>MEDIUM_CONT_RACE</td>
<td>CMD_TYPE_UNKNOWN</td>
</tr>
<tr>
<td>UREQ_TYPE_UNKNOWN</td>
<td>INTERRUPTS_OVERRUN</td>
</tr>
<tr>
<td>WAITING_FOR_INTERRUPT_DMA</td>
<td>WAITING_FOR_INTERRUPT_ACK</td>
</tr>
<tr>
<td>WAITING_FOR_INTERRUPT_TIMER</td>
<td>SLABS_RECYCLING</td>
</tr>
<tr>
<td>SLABS_PRESSURE</td>
<td>SLABS_STARVATION</td>
</tr>
<tr>
<td>OUT_OF_RDMA_HANDLES</td>
<td>EVENTQ_FULL</td>
</tr>
<tr>
<td>BUFFER_DROP</td>
<td>MEMORY_DROP</td>
</tr>
<tr>
<td>HARDWARE_FLOW_CONTROL</td>
<td>SIMULATED_PACKETS_LOST</td>
</tr>
<tr>
<td>LOGGING_FRAMES_DUMPED</td>
<td>WAKE_INTERRUPTS</td>
</tr>
<tr>
<td>AVERTED_WAKEUP_RACE</td>
<td>DMA_METADATA_RACE</td>
</tr>
</tbody>
</table>

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
PAPI-C Status

• PAPI 3.9 technology preview available with documentation
• PAPI 3.9.x under active development
• Implemented Components:
  – Myrinet MX
  – ACPI temperature sensor component
  – ‘toy’ network component
• Tested on HPC Challenge benchmarks
• Tested platforms include Pentium III, Pentium 4, Core2, Itanium and Opteron
• Platforms in development include Nehalem, POWER, SiCortex, BG/P
PAPI Release Status

• PAPI 3.7.0 September ‘09
  – Terminal releases of ‘PAPI Classic’
  – Nehalem, Atom
  – Shanghai, Istanbul, POWER7
  – Windows
  – Preliminary support for PCL

• Component PAPI 4.0 October ‘09
  – Components under development or consideration:
    • Lm-sensors
    • Infiniband
    • SeaStar II
    • GPGPUs??
    • Lustre
    • Nehalem Uncore
    • ...
### PAPI Event Tracker

- Web based event submission
- Generated by built-in PAPI xml tool
- Allows searching and comparison of multiple platforms

<table>
<thead>
<tr>
<th>REPORT VIEW</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vendor:</strong></td>
</tr>
<tr>
<td><strong>CPU Model:</strong></td>
</tr>
<tr>
<td><strong>Node Count:</strong></td>
</tr>
<tr>
<td><strong>CPU Count:</strong></td>
</tr>
<tr>
<td><strong>Node CPU Count:</strong></td>
</tr>
<tr>
<td><strong>CPU Revision:</strong></td>
</tr>
<tr>
<td><strong>CPU Clockrate:</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Submitter</th>
<th>Preset Event</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>CPU</td>
<td></td>
</tr>
<tr>
<td>PAPI_BR_UCL</td>
<td>PAPI_BR_IDL</td>
<td>PAPI_BR_IDL</td>
</tr>
<tr>
<td>PAPI_BR_CN</td>
<td>PAPI_BR_CN</td>
<td>PAPI_BR_CN</td>
</tr>
<tr>
<td>PAPI_BR_INS</td>
<td>PAPI_BR_INS</td>
<td>PAPI_BR_INS</td>
</tr>
<tr>
<td>PAPI_BR_MSP</td>
<td>PAPI_BR_MSP</td>
<td>PAPI_BR_MSP</td>
</tr>
<tr>
<td>PAPI_BR_NTK</td>
<td>PAPI_BR_NTK</td>
<td>PAPI_BR_NTK</td>
</tr>
<tr>
<td>PAPI_BR_PRC</td>
<td>PAPI_BR_PRC</td>
<td>PAPI_BR_PRC</td>
</tr>
<tr>
<td>PAPI_BR_TKN</td>
<td>PAPI_BR_TKN</td>
<td>PAPI_BR_TKN</td>
</tr>
<tr>
<td>PAPI_BR_UCN</td>
<td>PAPI_BR_UCN</td>
<td>PAPI_BR_UCN</td>
</tr>
<tr>
<td>PAPI_BTAC_M</td>
<td>PAPI_BTAC_M</td>
<td>PAPI_BTAC_M</td>
</tr>
<tr>
<td>PAPI_CA_CLN</td>
<td>PAPI_CA_CLN</td>
<td>PAPI_CA_CLN</td>
</tr>
<tr>
<td>PAPI_CA_INV</td>
<td>PAPI_CA_INV</td>
<td>PAPI_CA_INV</td>
</tr>
</tbody>
</table>

**Key:**
- event not supported (red)
- event is supported (yellow)
- all components support the event (blue)
Future Directions

• Power PAPI
  – Measure power consumption

• Multi-core support
  – Memory bandwidth
  – Cache sharing
  – Bus and other resource contention

• User-defined events
  – USER, L2 DTLB miss ratio, DERIVED_POSTFIX,\n    |N0|N1|N2|+/\T, DTLB_L1M_L2M, DTLB_L1M_L2H,\n    DTLB_L1M_L2M

• Multi-CPU PAPI
  – For heterogeneous systems like RoadRunner

• User-driven documentation
  – Wiki man pages
  – Wiki users guide
  – User submitted event configurations
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

Rick Kufrin
Rui Liu
National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign
Outline

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
• 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:~% 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
• 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

**Version**: 1.0  
**Created**: Mon Dec 30 11:31:53 AM Central Standard Time 2002  
**Generator**: psprocess 0.5  
**XML Source**: /u/ncsa/anyuser/performance/psrun-ia64.xml

#### Execution Information

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

#### Processor and System Information

<table>
<thead>
<tr>
<th>Node CPUs</th>
<th>2</th>
</tr>
</thead>
<tbody>
<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>Cache Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache levels : 3</td>
</tr>
</tbody>
</table>

Level 1
- Type : data
- Size (KB) : 16
- Linesize (B) : 32
- Assoc : 4
- Type : instruction
- Size (KB) : 16
- Linesize (B) : 32
- Assoc : 4

Level 2
- Type : unified
- Size (KB) : 96
- Linesize (B) : 64
- Assoc : 6
## psprocess: Text Mode, cont’d

<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

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graduated instructions per cycle</td>
<td>1.765</td>
</tr>
<tr>
<td>Graduated floating point instructions per cycle</td>
<td>0.145</td>
</tr>
<tr>
<td>Level 3 cache miss ratio (data)</td>
<td>0.957</td>
</tr>
<tr>
<td>Bandwidth used to level 3 cache (MB/s)</td>
<td>385.087</td>
</tr>
<tr>
<td>% cycles with no instruction issue</td>
<td>10.410</td>
</tr>
<tr>
<td>% cycles stalled on memory access</td>
<td>43.139</td>
</tr>
<tr>
<td>MFLOPS (cycles)</td>
<td>115.905</td>
</tr>
<tr>
<td>MFLOPS (wallclock)</td>
<td>114.441</td>
</tr>
</tbody>
</table>
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>
```
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
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).
• 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 --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_DOFOBK: if set (to anything), monitors child processes also

“psrun –h” will show a complete listing of recognized variables
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
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
The goals of this exercise are to gain experience with:
1. querying your computer with \textit{psinv}
2. collecting performance data with \textit{psrun}
3. examining the results with \textit{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 \texttt{psinv}

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

\% \texttt{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.NNNN.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 -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:

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

• Once again, use `psprocess` to view:

```
% 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:

% psprocess -o pstau.xml -x --glob ‘bt_profile*.xml’
% paraprof -f psrun pstau.xml

• ... and Scalasca’s Cube:

% 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
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”)
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)

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)

• 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
include 'fperfsuite.h'
call PSF_hwpc_init(ierr)
call PSF_hwpc_start(ierr)
do j = 1, n
    do i = 1, m
        do k = 1, 1
            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("perfdata");
```

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

• Currently supports HWPC report (“count” and “profile” mode), resource report and multi-HWPC 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 an optional user metric definition file. If no user metric definition file is given, it automatically uses the system provided one
• Includes detailed Javadoc documentation
• In toString() method, supports internationalization and localization in description strings of metrics (as in Tcl version) and number format of metric values
Example Use of the PS Java Metrics API

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

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

// The metric calculator object understands the format of
// PerfSuite metric definition files (default or custom)
PS_MetricCalculator metricCalculator =
    new PS_MetricCalculator(report);
    // or, to use a custom metric definition file:
    // new PS_MetricCalculator (report, metricFileName);

System.out.println("Calculated metric values:");
System.out.println(metricCalculator.getCalculatedMetrics());

System.out.println("Metric descriptions and values in the default
locale:");
System.out.println (metricCalculator.toString());
```
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 release in alpha cycle:**
      – 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:
  – http://perfsuite.ncsa.uiuc.edu/
  – http://www.sf.net/projects/perfsuite/
TAU PERFORMANCE SYSTEM

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

Allen D. Malony, Shangkar Mayanglambam, Suzanne Millstein
Department of Computer and Information Science
University of Oregon
• 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, ...)

TAU Performance System®
Building Bridges to Other Tools

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
TAU Instrumentation / Measurement

**Instrumentation**
- event selection
- source code
- object code
- library wrapper
- binary code
- virtual machine

**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
- Application source
- Parsed program
- tau_instrumentor
- Instrumented source
- Instrumentation specification file
Program Database Toolkit (PDT)

Application / Library

C / C++ parser

IL analyzer

C / C++

Fortran parser

F77/90/95

IL analyzer

Program Database Files

DUCTAPE

TAU instrumentor

Automatic source instrumentation

SC ‘09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
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
• 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
• 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

Profile Data Management (PerfDMF)
- profile translators
- Metadata (XML)
- profile database

Profile Analysis (ParaProf)

Profile Data Mining (PerfExplorer)

Trace Data Management
- trace translators
- trace storage

Trace Visualizers
- Vampir
- JumpShot
- Paraver

Trace Analyzers
- Expert
- ProfileGen
- Vampir Server

TAU Portal

Instrumentation

Analysis
- profiles
- traces

event selection

symbol table

event information
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)
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
- raw profiles
- profile metadata
- gprof
- mpiP
- psrun
- HPMtoolkit
- ...
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 environment
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...

Performance Problems
Application
Machine

Context Knowledge
Source Code
Build Environment
Run Environment

Execution

Performance Result

...to understand this
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
## Parallel Profile Analysis – pprof

### Buffers Files Tools Edit Search Mule Help

Reading Profile files in profile.*

<table>
<thead>
<tr>
<th>NODE</th>
<th>CONTEXT</th>
<th>THREAD</th>
<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>181293289</td>
<td>15</td>
<td>1912393289</td>
<td>applu</td>
</tr>
<tr>
<td>99.6</td>
<td>3,567</td>
<td>3:10:463</td>
<td>3</td>
<td>37517</td>
<td>63487925</td>
<td>3343</td>
<td>exchange_1</td>
<td></td>
</tr>
<tr>
<td>67.1</td>
<td>491</td>
<td>2:08:326</td>
<td>37200</td>
<td>37200</td>
<td>37200</td>
<td>3459</td>
<td>exchange_1</td>
<td></td>
</tr>
<tr>
<td>44.5</td>
<td>6,461</td>
<td>1:25:169</td>
<td>9300</td>
<td>9300</td>
<td>18600</td>
<td>9137</td>
<td>blts</td>
<td></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</td>
<td>MPI_Recv()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29.5</td>
<td>6,778</td>
<td>56,407</td>
<td>9300</td>
<td>18600</td>
<td>6065</td>
<td>blts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26.2</td>
<td>50,142</td>
<td>50,142</td>
<td>18204</td>
<td>0</td>
<td>2611</td>
<td>MPI_Send()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16.2</td>
<td>24,451</td>
<td>31,031</td>
<td>301</td>
<td>602</td>
<td>103096</td>
<td>rhm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.9</td>
<td>7,501</td>
<td>7,501</td>
<td>9300</td>
<td>0</td>
<td>807</td>
<td>jaccd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>6,594</td>
<td>6,594</td>
<td>604</td>
<td>1812</td>
<td>10918</td>
<td>exchange_3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>6,590</td>
<td>6,590</td>
<td>9300</td>
<td>0</td>
<td>709</td>
<td>jaccu</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.6</td>
<td>4,989</td>
<td>4,989</td>
<td>608</td>
<td>0</td>
<td>8205</td>
<td>MPI_Wait()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.2</td>
<td>0.44</td>
<td>400</td>
<td>1</td>
<td>4</td>
<td>400081</td>
<td>init_comm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.2</td>
<td>398</td>
<td>398</td>
<td>1</td>
<td>39</td>
<td>399634</td>
<td>MPI_Init()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.1</td>
<td>240</td>
<td>240</td>
<td>1</td>
<td>47816</td>
<td>247088</td>
<td>setiv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.1</td>
<td>134</td>
<td>134</td>
<td>57252</td>
<td>0</td>
<td>2</td>
<td>exact</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.1</td>
<td>89</td>
<td>103</td>
<td>1</td>
<td>2</td>
<td>103168</td>
<td>erhs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.1</td>
<td>966</td>
<td>96</td>
<td>1</td>
<td>2</td>
<td>96458</td>
<td>read_input</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>95</td>
<td>95</td>
<td>9</td>
<td>0</td>
<td>10603</td>
<td>MPI_Bcast()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>26</td>
<td>44</td>
<td>1</td>
<td>7937</td>
<td>44878</td>
<td>error</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>24</td>
<td>24</td>
<td>608</td>
<td>0</td>
<td>1490</td>
<td>MPI_Recv()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>15</td>
<td>15</td>
<td>1</td>
<td>5</td>
<td>15630</td>
<td>MPI_Finalize()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>4</td>
<td>12</td>
<td>1</td>
<td>1700</td>
<td>12335</td>
<td>setbv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>7</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>2893</td>
<td>12norm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>3</td>
<td>3</td>
<td>8</td>
<td>0</td>
<td>491</td>
<td>MPI_Allreduce()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>3</td>
<td>3</td>
<td>8</td>
<td>0</td>
<td>3824</td>
<td>pinter</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1007</td>
<td>MPI_Barrier()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>0.116</td>
<td>0.137</td>
<td>1</td>
<td>4</td>
<td>837</td>
<td>exchange_4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>0.512</td>
<td>0.512</td>
<td>1</td>
<td>0</td>
<td>512</td>
<td>MPI_Keyval_create()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>0.121</td>
<td>0.353</td>
<td>1</td>
<td>2</td>
<td>353</td>
<td>exchange_5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>0.024</td>
<td>0.191</td>
<td>1</td>
<td>2</td>
<td>131</td>
<td>exchange_6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>0.103</td>
<td>0.103</td>
<td>0</td>
<td>17</td>
<td>MPI_Type_contiguous()</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**SC '09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS**

172
Parallel Profile Analysis – ParaProf

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

**Applications**
- Standard Applications
  - Default App
    - Default
    - utonium
    - proton_mysql
    - proton_postgresql
    - proton_peri
    - perielmet
- Default Exp
  - Default
  - utonium
  - proton_mysql
  - proton_postgresql
  - proton_peri
- Default

**Multiple PerfDMF DBs**

<table>
<thead>
<tr>
<th>TrialField</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.uoregon.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.uoregon.edu</td>
</tr>
<tr>
<td>Memory Size</td>
<td>8161240 kB</td>
</tr>
<tr>
<td>Node Name</td>
<td>demon.nic.uoregon.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

node, context, thread

8K processors

Miranda
- hydrodynamics
- Fortran + MPI
- LLNL BG/L
ParaProf – Callpath Profile

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODULEHYDROS:HYDRO_SWEAP</td>
<td>26.47%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO_3D =&gt; MODULEHYDROS:HYDRO_SWEAP</td>
<td>26.47%</td>
</tr>
<tr>
<td>MODULEHYDROS:HYDRO_1D</td>
<td>24.55%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO_1D</td>
<td>24.55%</td>
</tr>
<tr>
<td>FLASH =&gt; EVOLVE =&gt; HYDRO:HYDRO_3D</td>
<td>14.35%</td>
</tr>
<tr>
<td>Flash == EVOLVE == HYDRO:HYDRO_3D</td>
<td>4.50%</td>
</tr>
<tr>
<td>MPL_Ssend</td>
<td>44.27%</td>
</tr>
</tbody>
</table>

Flash
- thermonuclear flashes
- Fortran + MPI
- Argonne
ParaProf – Scalable Histogram

8k processors

16k processors

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
ParaProf – 3D View (Full Profile)

LS3DF

64k 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

```c
Particle* P[MAX]; /* Array of particles */
int GenerateParticles() {
    /* distribute particles over all faces of the cube */
    for (int face=0, last=0; face < 6; face++) {
        /* particles on this face */
        int particles_on_this_face = num(face);
        for (int i=last; i < particles_on_this_face; i++) {
            /* particle properties are a function of face */
            P[i] = ... f(face);
            ...
        }
        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
How is MPI_Wait() distributed relative to solver direction? Application routine names reflect phase semantics.
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
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
Snapshot Replay in ParaProf

All windows dynamically update
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
- -shmemp[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
• 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-3.14.1
  -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/pkgs/tau/x86_64/lib/Makefile.tau-mpi-pdt-pgi
  /usr/pkgs/tau/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/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/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/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)
% cd /usr/local/packages/tau-2.19/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/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

% setenv TAU_MAKEFILE /opt/tau-2.19/x86_64
       /lib/Makefile.tau-mpi-pdt
% set path=(/opt/tau-2.19/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

% setenv TAU_MAKEFILE /opt/tau-2.19/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/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/x86_64
   /lib/Makefile.tau-mpi
% setenv TAU_OPTIONS ‘-optCompInst -optVerbose’
% % set path=(/opt/tau-2.19/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/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/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 -> Arg 1 = PAPI_FP_INS,
  Arg 2 = GET_TIME_OF_DAY, Operation = Divide -> Apply, choose.
-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_DEPTHLIMIT environment variable specifies depth
  – % setenv TAU_DEPTHLIMIT 1  
    – enables instrumentation in only “main”  
  – % setenv TAU_DEPTHLIMIT 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

```bash
% setenv TAU_MAKEFILE /opt/tau-2.19/x86_64/lib/Makefile.tau-mpi-pdt
% set path=($(opt/tau-2.19/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
% setenv TAU_MAKEFILE /opt/tau-2.19/x86_64/lib/Makefile.tau-mpi-pdt
% set path=(/opt/tau-2.19/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
Usage: tau_instrumentor <pdbfile> <sourcefile> [-o <outputfile>] [-noinline]
[-g groupname] [-i headerfile] [-c|-c++|-fortran] [-f <instr_req_file>]
For selective instrumentation, use -f option
% tau_instrumentor foo.pdb foo.cpp -o foo.inst.cpp -f selective.dat
% cat selective.dat
# Selective instrumentation: Specify an exclude/include list of routines/files.
BEGIN_EXCLUDE_LIST
void quicksort(int *, int, int)
void sort_5elements(int *)
void interchange(int *, int *)
END_EXCLUDE_LIST

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

```c
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 msec</th>
<th>Inclusive total msec</th>
<th>#Call</th>
<th>#Subrs</th>
<th>Inclusive Name usec/call</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0</td>
<td>0.12</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() [ file = &lt;loop_test.cpp&gt; line,col = &lt;23,3&gt; to &lt;30,3&gt; ]</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() [ file = &lt;loop_test.cpp&gt; line,col = &lt;38,3&gt; to &lt;46,7&gt; ]</td>
</tr>
<tr>
<td>0.1</td>
<td>33</td>
<td>33</td>
<td>1</td>
<td>0</td>
<td>33964 Loop: double multiply() [ file = &lt;loop_test.cpp&gt; line,col = &lt;16,10&gt; to &lt;21,12&gt; ]</td>
</tr>
</tbody>
</table>
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/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

9647.318 LEQ_IKSWEEPET
4357.213 LEQ_BICGS0T
2669.887 LEQ_MATVECT
1777.752 SOLVE_SPECIES_EQ
1417.986 SOLVE_LIN_EQ
1028.448 PHYSICAL_PROP
783.402 RRATES
682.376 LEQ_MSOLVET
530.858 INIT_AB_M
463.788 CALC_MASS_FLUX_SPHR
446.025 INIT_MU_S
421.747 CALC_RESID_S
381.363 SOLVE_ENERGY_EQ
371.199 SOURCE_PHI
258.829 DRAG_GS
Solution: Generating a flat profile with MPI

% setenv TAU_MAKEFILE /opt/tau-2.19/x86_64
   /lib/Makefile.tau-mpi-pdt
% set path=(/opt/tau-2.19/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:**

Metric: GET_TIME_OF_DAY
Value: Exclusive
Units: microseconds

1729975.333

Loop: MULTIPLY_MATRICES [{matmul.f90} {31.9}-{36.14}]

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

```bash
% setenv TAU_MAKEFILE /opt/tau-2.19/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}]
- Loop: INITIALIZE [[matmult.f90] {10,9}-{14,14}]
- Loop: INITIALIZE [[matmult.f90] {17,9}-{21,14}]
- Loop: MAIN [[matmult.f90] {71,9}-{74,14}]
- Loop: MAIN [[matmult.f90] {112,9}-{115,14}]
- Loop: MAIN [[matmult.f90] {117,9}-{128,14}]
- MULTIPLY_MATRICES

- 770.699
- 233.39
- 223.24
- 171.855
- 170.862
- 122.36
- 37.549
- 21.367
- 13.795
- 8.935
- 1.131
- 0.794
- 0.647
- 0.355
- 0.171
- 0.115
- 0.023
- MAIN
Generate a PAPI profile with 2 or more counters

% setenv TAU_MAKEFILE /opt/tau-2.19/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.
Usage Scenarios: Compiler-based Instrumentation

- 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/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

SC '09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
Callpath Profile

- Generates program callgraph
Generate a Callpath Profile

```
% setenv TAU_MAKEFILE /opt/tau-2.19/x86_64
    /lib/Makefile.tau-mpi-pdt
% set path=/opt/tau-2.19/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

File Windows Help

<table>
<thead>
<tr>
<th>Name</th>
<th>NumSamples</th>
<th>MaxValue</th>
<th>MinValue</th>
<th>MeanValue</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MATRICES:ALLOCATE_MATRICES [matrix.f90] {10,7}–{13,38}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEMORY LEAK! malloc size &lt;file=matrix.f90, variable=C, line=11&gt;</td>
<td>1</td>
<td>8,000,000</td>
<td>8,000,000</td>
<td>8,000,000</td>
<td>0</td>
</tr>
<tr>
<td>malloc size &lt;file=matrix.f90, variable=A, line=11&gt;</td>
<td>1</td>
<td>8,000,000</td>
<td>8,000,000</td>
<td>8,000,000</td>
<td>0</td>
</tr>
<tr>
<td>malloc size &lt;file=matrix.f90, variable=B, line=11&gt;</td>
<td>1</td>
<td>8,000,000</td>
<td>8,000,000</td>
<td>8,000,000</td>
<td>0</td>
</tr>
<tr>
<td>malloc size &lt;file=matrix.f90, variable=C, line=11&gt;</td>
<td>1</td>
<td>8,000,000</td>
<td>8,000,000</td>
<td>8,000,000</td>
<td>0</td>
</tr>
<tr>
<td>MATRICES:DEALLOCATE_MATRICES [matrix.f90] {14,7}–{17,40}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>free size &lt;file=matrix.f90, variable=A, line=15&gt;</td>
<td>1</td>
<td>8,000,000</td>
<td>8,000,000</td>
<td>8,000,000</td>
<td>0</td>
</tr>
<tr>
<td>free size &lt;file=matrix.f90, variable=B, line=15&gt;</td>
<td>1</td>
<td>8,000,000</td>
<td>8,000,000</td>
<td>8,000,000</td>
<td>0</td>
</tr>
</tbody>
</table>

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

80000000 80000000 80000000 80000000 80000000 80000000 80000000 80000000 80000000 ...

Mean: n.c.t 0,0 0,0 0,0 0,0 0,0 0,0 0,0 0,0 0,0
Std. Dev.: n.c.t 0,0 0,0 0,0 0,0 0,0 0,0 0,0 0,0 0,0

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS

POINT

VI-HPS
Detect Memory Leaks

% setenv TAU_MAKEFILE /opt/tau-2.19/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

Interval event
e.g., routines (start/stop)

Atomic events
(trigger with value)

% setenv TAU_CALLPATH_DEPTH 0
% setenv TAU_TRACK_HEAP 1

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
Atomic Events, Context Events

<table>
<thead>
<tr>
<th>%Time</th>
<th>Exclusive</th>
<th>Inclusive</th>
<th>#Call</th>
<th>#Subs</th>
<th>Inclusive</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>msec</td>
<td>total msec</td>
<td></td>
<td></td>
<td>usec/call</td>
<td></td>
</tr>
<tr>
<td>100.0</td>
<td>0.253</td>
<td>1.106</td>
<td>1</td>
<td>44</td>
<td></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></td>
<td>1031311 MPI_Init()</td>
</tr>
<tr>
<td>6.0</td>
<td>1</td>
<td>66</td>
<td>40</td>
<td>320</td>
<td></td>
<td>1650 void func(int, int) C</td>
</tr>
<tr>
<td>5.7</td>
<td>66</td>
<td>66</td>
<td>40</td>
<td>0</td>
<td></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></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></td>
<td>10 MPI_Recv()</td>
</tr>
<tr>
<td>0.0</td>
<td>0.141</td>
<td>0.141</td>
<td>120</td>
<td>0</td>
<td></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></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></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></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>5.139E+04</td>
<td>44.39</td>
<td>3.091E+04</td>
<td>1.234E+04</td>
<td>Message size for broadcast</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</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_Barrier()</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_Bcast()</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>0.00069055</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

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
Context Events (default)

% setenv TAU_CALLPATH_DEPTH 2
% setenv TAU_TRACK_HEAP 1

Context event
= atomic event
+ executing context
## Environment Variables in TAU

<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 TAU_TRACK_HEADROOM</td>
<td>0</td>
<td>Setting to 1 turns on tracking heap memory/headroom at routine entry &amp; exit</td>
</tr>
<tr>
<td></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 separted 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>
Measuring Performance of PGI Accelerator Code

TAU: ParaProf: n,c,t 0,0,0 - mat1k.ppk

Metric: TIME
Value: Exclusive percent

68.044%
24.417%
3.206%
1.572%
1.572%
0.782%
0.142%
0.122%
0.12%
0.017%
0.005%
0.002%
2.1E-4%
1.2E-4%
1.2E-4%

__pgi_cu_launch multiply_matrices (pgi_kernel_7,gx=32,gy=32,gz=1,bx=16,by=16,bz=1) [[mm2.f90][15]]
__pgi_cu_init multiply_matrices [[mm2.f90][9]]
__pgi_cu_download2 multiply_matrices var=a [[mm2.f90][20]]
__pgi_cu_upload2 multiply_matrices var=b [[mm2.f90][9]]
__pgi_cu_upload2 multiply_matrices var=c [[mm2.f90][9]]
mymatrixmultiply [[mmxdrv.f90][1,0]]
__pgi_cu_free multiply_matrices [[mm2.f90]]
__pgi_cu alloc multiply_matrices [[mm2.f90][9]]
multiply_matrices [[mm2.f90][5,0]]
pgi accelerator region
__pgi_cu_module multiply_matrices [[mm2.f90][9]]
__pgi_cu_module_function multiply_matrices [[mm2.f90][11]]
__pgi_cu_paramset multiply_matrices [[mm2.f90]]

TAU: ParaProf: Thread Statistics: n,c,t 0,0,0 - mat1k.ppk

<table>
<thead>
<tr>
<th>Name</th>
<th>Exclusive TIME</th>
<th>Inclusive TIME</th>
<th>Calls</th>
<th>Child Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>__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>mymatrixmultiply [[mmxdrv.f90][1,0]]</td>
<td>0.125</td>
<td>0.1621</td>
<td>1</td>
<td>1</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>0.023</td>
<td>0.023</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>__pgi_cu_free multiply_matrices [[mm2.f90]]</td>
<td>0.002</td>
<td>0.002</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>__pgi_cu alloc multiply_matrices [[mm2.f90][9]]</td>
<td>0.019</td>
<td>0.019</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>multiply_matrices [[mm2.f90][5,0]]</td>
<td>0.003</td>
<td>15.895</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>pgi accelerator region</td>
<td>0.001</td>
<td>15.893</td>
<td>5</td>
<td>85</td>
</tr>
<tr>
<td>__pgi_cu_module multiply_matrices [[mm2.f90][9]]</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</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>
Usage Scenarios: Mixed Python+F90+C+pyMPI

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

![Graph showing performance metrics and code snippets related to usage scenarios.]

---

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
Generate a Multi-Language Profile w/ Python

% setenv TAU_MAKEFILE /opt/tau-2.19/x86_64
   /lib/Makefile.tau-python-mpi-pdt
% set path=/opt/tau-2.19/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

[Diagram showing a bar chart for S3D Jaguar CNL:Scaling]

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
Performance Regression Testing

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

- int main(int, char **)  
- std::vector<double, std::allocator<double>> FcCoreCellUpdate...
- void FcTmCoreFluxCalc::computeFluxes()  
- MPI_Recv()
- double FcDataAssimilator::getValue(const std::string & cons...
- MPI_Init()
- FcHdf5Tpl <DATATYPE>::writeDataSet
- void FcDataAssimilatorUfiles::parseUfiles(const std::vector<...
- void FcUpdaterComponent::dumpToFile(const std::string & con...
- other
Evaluate Scalability using PerfExplorer Charts

% setenv TAU_MAKEFILE /opt/tau-2.19/x86_64
   /lib/Makefile.tau-mpi-pdt
% set path=/opt/tau-2.19/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:
% perfdfm_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 perfdfm_loadtrial on the commandline)
% perfexplorer
(Charts -> Speedup)
• Goal: What is the volume of inter-process communication? Along which calling path?

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
Evaluate Scalability using PerfExplorer Charts

```
% setenv TAU_MAKEFILE
   $TAU/Makefile.tau-mpi-pdt
% set path=/usr/local/packages/tau-2.19/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)
```
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

Brian J. N. Wylie
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 July 2009: Scalasca v1.2
    • Tutorial Live-DVD
    • Scalasca 1.3β currently in testing
Scalasca features

• Open source, New BSD license
• Portable
  – BG/P, BG/L, IBM SP & blade clusters, Cray XT, SGI Altix,
    NEC SX, SiCortex, 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
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 instrumenter
- 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.

Diagram:

- Program sources
- Compiler
- Instrumenter
- Instrumented executable
- Application + measurement lib
- Summary analysis
- Analysis report examiner

SC '09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
Measurement event tracing & analysis

- During measurement, time-stamped events are 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
- Application + measurement lib
- Unified defs+maps
- Trace 1, 2, .., N
- Parallel trace analyzer
- Trace analysis
- Analysis report examiner

SC '09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
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 1.2
  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
• 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 instrumenter 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 3.2 (July 2009)
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)

How is it distributed across the processes?

What kind of performance problem?

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 transprt/ct/hsmoc
- With 23.2% std dev over 512 processes
- Lowest values in 3\textsuperscript{rd} and 4\textsuperscript{th} planes of the Cartesian grid
- 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
Scalasca 1.2 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
  – limited analyses of non-blocking point-to-point, RMA, ...
• 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
Scalasca 1.2 added functionality

- OpenMP measurement & analysis
  - run-time summaries include OpenMP metrics (for all threads)
    - not all threads need to participate in parallel regions
  - trace collection & analysis unchanged
- Hybrid OpenMP/MPI measurement & analysis
  - run-time summaries include OpenMP metrics (for all threads)
    - not all threads need to participate in parallel regions
  - trace collection complemented with hybrid trace analysis
    - requires all threads to participate in parallel regions
- MPI File I/O analysis
  - file operations (reads/writes), collective read/write time
- Improved support for Cray XT, NEC SX
- Improved support for PGI compilers
Tutorial Exercise
NPB-MPI BT
Performance analysis steps

1. Reference preparation for validation
2. Program instrumentation: skin
3. Summary measurement collection & analysis: scan [-s]
4. Summary analysis report examination: square
5. Summary experiment scoring: square -s
6. Event trace collection & analysis: scan -t
7. Event trace analysis report examination: square

- Configuration & customization
  - Instrumentation, Measurement, Analysis, Presentation
Live-DVD exercise sources

• Additional exercise sources provided for several programs (implemented in various languages & parallelizations)
  – workshop-scalasca/Exercise
    • jacobi # MPI/OpenMP/hybrid x C/C++/Fortran
    • sweep3d # MPI/Fortran
    • smg2000 # MPI/C
    • NPB3.3-MPI # MPI/Fortran & C
    • NPB3.3-OMP # OpenMP/Fortran & C
    • NPB3.3-MZ-MPI # hybrid OpenMP+MPI/Fortran

• This tutorial concentrates on NPB3.3-MPI-BT
  – but can be repeated substituting other codes as desired
NPB-BT

- Intermediate-level tutorial example
- Available in MPI, OpenMP, hybrid OpenMP/MPI variants
  - also MPI File I/O variants (collective & individual)
- Summary measurement collection & analysis
  - Automatic instrumentation
    - OpenMP, MPI & application functions
  - Summary analysis report examination
  - PAPI hardware counter metrics
- Trace measurement collection & analysis
  - Filter determination, specification & configuration
  - Automatic trace analysis report patterns
- Manual instrumentation
- Measurement configuration
- Analysis report algebra
Load the Scalasca module

• Load the module

% module load UNITE
UNITE loaded
% module load scalasca
scalasca/1.2 loaded

• ... and run **scalasca** for brief usage information

% scalasca
Scalasca 1.2
Toolset for scalable performance analysis of large-scale applications
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

- v: enable verbose commentary
- n: show actions without taking them
- h: show quick reference guide (only)
NPB instrumentation

• Prefix compile/link commands in Makefile definitions (config/make.def) with Scalasca instrumenter

```
MPIF77 = scalasca -instrument mpif77
FLINK = $(MPIF77)
FFLAGS = -O

mpi-bt: $(OBJECTS)
    $(FLINK) $(FFLAGS) -o mpi-bt $(OBJECTS)
.f.o:
    $(MPIF77) $(FFLAGS) -c $<
```

• or use PREP macro as customizable preparation preposition

```
MPIF77 = $(PREP) mpif77
```

• By default, PREP macro is not set and no instrumentation is performed for a regular “production” build
• Specifying a PREP value in the Makefile or on the make command line uses it as a preposition, e.g., for instrumentation
  – make PREP=”scalasca -instrument” ...
    scalasca -instrument mpif77 -O -c bt.f
NPB-MPI-BT instrumented build

- Return to root directory and clean-up

  ```
  % make clean
  ```

- Re-build specifying Scalasca instrumenter as PREP

  ```
  % make bt NPROCS=16 CLASS=W PREP="scalasca -instrument"
  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
  scalasca -instrument mpif77 -c -O bt.f
  ... 
  cd ../common; scalasca -instrument mpif77 -c -O timers.f
  scalasca -instrument mpif77 -c -O btio.f
  scalasca -instrument mpif77 -O -o ../bin.scalasca/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
  INFO: Instrumented executable for MPI measurement
  gmake: Leaving directory 'BT'
  ```
BT-MPI summary measurement

- Run the application using the Scalasca measurement collection & analysis nexus prefixed to launch command

```bash
% cd bin.scalasca; scalasca -analyze mpiexec -np 16 ./bt_W.16
S=C=A=N: Scalasca 1.2 runtime summarization
S=C=A=N: ./epik_bt_W_16_sum experiment archive
S=C=A=N: Sun Mar 29 16:36:31 2009: Collect start
mpiexec -np 16 ./bt_W.16
[00000]EPIK: Created new measurement archive ./epik_bt_W_16_sum
[00000]EPIK: Activated ./epik_bt_W_16_sum [NO TRACE] (0.006s)

[... Application output ...]

[00000]EPIK: Closing experiment ./epik_bt_W_16_sum
[00000]EPIK: 102 unique paths (102 max paths, 4 max frames, 0 unknown)
[00000]EPIK: Unifying... done (0.023s)
[00000]EPIK: Collating... done (0.049s)
[00000]EPIK: Closed experiment ./epik_bt_W_16_sum (0.073s)
S=C=A=N: Sun Mar 29 16:36:45 2009: Collect done (status=0) 14s
S=C=A=N: ./epik_bt_W_16_sum complete.
```

- Produces experiment directory ./epik_bt_W_16_sum
BT-MPI summary analysis report examination

• Interactive exploration with Scalasca GUI

```bash
% scalasca -examine epik_bt_W_16_sum
INFO: Post-processing runtime summarization result...
INFO: Displaying ./epik_bt_W_16_sum/summary.cube...
```

[GUI showing summary analysis report]

• Report scoring as textual output

```bash
% scalasca -examine -s epik_bt_W_16_sum
cube3_score ./epik_bt_W_16_sum/summary.cube
Reading ./epik_bt_W_16_sum/summary.cube... done.
Estimated aggregate size of event trace (total_tbc): 513,823,960 bytes
Estimated size of largest process trace (max_tbc): 32,528,680 bytes
(When tracing set ELG_BUFFER_SIZE to avoid intermediate flushes or reduce requirements using filter file listing names of USR regions.)
```

<table>
<thead>
<tr>
<th>flt</th>
<th>type</th>
<th>max_tbc</th>
<th>time</th>
<th>%</th>
<th>region</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANY</td>
<td></td>
<td>32528680</td>
<td>220.22</td>
<td>100.00</td>
<td>(summary) ALL</td>
</tr>
<tr>
<td>MPI</td>
<td></td>
<td>642712</td>
<td>194.57</td>
<td>88.35</td>
<td>(summary) MPI</td>
</tr>
<tr>
<td>COM</td>
<td></td>
<td>197928</td>
<td>1.03</td>
<td>0.47</td>
<td>(summary) COM</td>
</tr>
<tr>
<td>USR</td>
<td></td>
<td>31688040</td>
<td>24.62</td>
<td>11.18</td>
<td>(summary) USR</td>
</tr>
</tbody>
</table>
Analysis report exploration (opening view)
Analysis report exploration (system tree)

Distribution of selected metric for call path by process/thread
Analysis report exploration (call tree)

Distribution of selected metric across the call tree

Selection updates metrics shown in columns to right

SC '09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
Split base metrics into more specific metrics
Analysis report exploration (source browser)

```
c - in our terminology stage is the number of the cell in the y-dir
c - i.e. stage = 1 means the start of the line stage=nCells means end
do stage = 1, nCells
  c = slice(3, stage)
  isize = cell_size(1, c) - 1
  jsize = cell_size(2, c) - 1
  ksize = cell_size(3, c) - 1

- set last-cell flag
  if (stage .eq. nCells) then
    last = 1
  else
    last = 0
  endif

- This is the first cell, so solve without receiving data
  first = 1
  call lhs(c)
  call z_solve_cell(first, last, c)
  else
  - Not the first cell of this line, so receive info from processor working on preceding cell
  first = 0
  call z_receive_solve_info(recv_id, c)

- overlap computations and communications
  call lhs(c)
  call rhs(c)
  wait for completion
  call mpi_wait(send_id, r_status, error)
call mpi_wait(recv_id, r_status, error)

- install C'(#start+1) and rhs'(#start+1) to be used in this cell
  call z_unpack_solve_info(c)
call z_solve_cell(first, last, c)
endif
if (last .eq. 0) call z_send_solve_info(send_id, c)
endif
```

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS

288
BT-MPI summary analysis score

• Summary measurement analysis score reveals
  – Total size of event trace would be over 500MB
  – Maximum trace buffer size would be over 30MB per process
    • smaller buffer would require flushes to disk during measurement resulting in substantial perturbation
  – 97% of the trace requirements are for USR regions
    • purely computational routines never found on COM call-paths common to communication routines
  – These USR regions contribute around 10% of total time
    • however, much of that is very likely to be measurement overhead for frequently-executed small routines

• Advisable to tune measurement configuration
  – Specify an adequate trace buffer size
  – Specify a filter file listing (USR) regions not to be measured
BT-MPI summary analysis report breakdown

- Report scoring with region breakdown

```bash
% cube3_score -r ./epik_bt_W_16_sum/summary.cube
```

<table>
<thead>
<tr>
<th>flt</th>
<th>type</th>
<th>max_tbc</th>
<th>time</th>
<th>% region</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANY</td>
<td>ANY</td>
<td>32528680</td>
<td>220.22</td>
<td>100.00 (summary) ALL</td>
</tr>
<tr>
<td>MPI</td>
<td>MPI</td>
<td>642712</td>
<td>194.57</td>
<td>88.35 (summary) MPI</td>
</tr>
<tr>
<td>COM</td>
<td>COM</td>
<td>197928</td>
<td>1.03</td>
<td>0.47 (summary) COM</td>
</tr>
<tr>
<td>USR</td>
<td>USR</td>
<td>31688040</td>
<td>24.62</td>
<td>11.18 (summary) USR</td>
</tr>
</tbody>
</table>

USR 10231704 4.44 2.02 binvcrhs_
USR 10231704 3.06 1.39 matvec_sub_
USR 10231704 3.53 1.60 matmul_sub_
USR 492048 0.16 0.07 binvrhs_
USR 360576 0.12 0.05 exact_solution_
MPI 241500 0.27 0.12 MPI_Isend
MPI 222180 0.12 0.06 MPI_Irecv
MPI 173664 173.02 78.57 MPI_Wait
USR 57888 0.06 0.03 lhsabinit_
USR 19296 3.53 1.60 y_solve_cell_
...
# BT-MPI summary analysis report filtering

- **Report scoring with prospective filter listing 6 USR regions**

```bash
% cube3_score -r -f npb.filt ./epik_bt_W_16_sum/summary.cube
```

Applying filter "./npb.filt":

Estimated aggregate size of event trace (total_tbc): 16,852,888 bytes

Estimated size of largest process trace (max_tbc): 1,053,304 bytes

<table>
<thead>
<tr>
<th>flt</th>
<th>type</th>
<th>max_tbc</th>
<th>time</th>
<th>%</th>
<th>region</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>FLT</td>
<td>31475376</td>
<td>11.37</td>
<td>5.16</td>
<td>(summary) FLT</td>
</tr>
<tr>
<td>*</td>
<td>ANY</td>
<td>1053328</td>
<td>208.85</td>
<td>94.84</td>
<td>(summary) ALL-FLT</td>
</tr>
<tr>
<td>-</td>
<td>MPI</td>
<td>642712</td>
<td>194.57</td>
<td>88.35</td>
<td>(summary) MPI-FLT</td>
</tr>
<tr>
<td>*</td>
<td>COM</td>
<td>197928</td>
<td>1.03</td>
<td>0.47</td>
<td>(summary) COM-FLT</td>
</tr>
<tr>
<td>*</td>
<td>USR</td>
<td>212688</td>
<td>13.25</td>
<td>6.02</td>
<td>(summary) USR-FLT</td>
</tr>
<tr>
<td>+</td>
<td>USR</td>
<td>10231704</td>
<td>4.44</td>
<td>2.02</td>
<td>binvcrhs_</td>
</tr>
<tr>
<td>+</td>
<td>USR</td>
<td>10231704</td>
<td>3.06</td>
<td>1.39</td>
<td>matvec_sub_</td>
</tr>
<tr>
<td>+</td>
<td>USR</td>
<td>10231704</td>
<td>3.53</td>
<td>1.60</td>
<td>matmul_sub_</td>
</tr>
<tr>
<td>+</td>
<td>USR</td>
<td>492048</td>
<td>0.16</td>
<td>0.07</td>
<td>binvrhs_</td>
</tr>
<tr>
<td>+</td>
<td>USR</td>
<td>360576</td>
<td>0.12</td>
<td>0.05</td>
<td>exact_solution_</td>
</tr>
<tr>
<td>-</td>
<td>MPI</td>
<td>241500</td>
<td>0.27</td>
<td>0.12</td>
<td>MPI_Isend</td>
</tr>
<tr>
<td>-</td>
<td>MPI</td>
<td>222180</td>
<td>0.12</td>
<td>0.06</td>
<td>MPI_Irecv</td>
</tr>
<tr>
<td>-</td>
<td>MPI</td>
<td>173664</td>
<td>173.02</td>
<td>78.57</td>
<td>MPI_Wait</td>
</tr>
<tr>
<td>+</td>
<td>USR</td>
<td>57888</td>
<td>0.06</td>
<td>0.03</td>
<td>lhsabinit_</td>
</tr>
</tbody>
</table>

Filtered routines marked with ‘+’

```bash
% cat npb.filt
# filter for bt
binvcrhs_
matvec_sub_
matmul_sub_
binvrhs_
exact_solution_
lhsabinit_
```

---

**Summary Report**

- The report includes a summary analysis with filtering.
- The filter listing highlights specific regions.
- Estimated aggregate size: 16,852,888 bytes.
- Estimated max size: 1,053,304 bytes.
- Various types of regions and routines are analyzed.
- Filtered routines are marked with `+`.

---

**Routines**

- `binvcrhs_`, `matvec_sub_`, `matmul_sub_`, `binvrhs_`, `exact_solution_`, `lhsabinit_` are considered for filtering.

---

**Event Trace Analysis**

- Event tracing is conducted with specific routines.
- Timings are recorded for each event.
- Filters are applied to reduce the data size.

---

**Filtering Benefits**

- Reduces data size for efficient analysis.
- Focuses on critical regions for performance engineering.

---

**Conclusion**

- Effective application of filters aids in productive performance engineering.
- Helps in identifying bottlenecks for further optimization.

---

**Further Analysis**

- Additional filtering parameters can be explored for more refined analysis.
- Integration with benchmarking tools for comprehensive performance evaluation.

---

**References**

- SC '09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS.

---

**Appendix**

- Detailed filtering parameters and scoring criteria.
- Extensive documentation on event tracing and filtering techniques.
BT-MPI filtered summary measurement

- Rename former measurement archive directory, set new filter configuration and re-run the measurement

```
% mv epik_bt_W_16_sum epik_bt_W_16_sum.nofil
% export EPK_FILTER=npb.filt
% scalasca -analyze mpiexec -np 16 ./bt_W.16
S=C=A=N: Scalasca 1.2 runtime summarization
S=C=A=N: ./epik_bt_W_16_sum experiment archive
S=C=A=N: Sun Mar 29 16:58:34 2009: Collect start
mpiexec -np 16 ./bt_W.16
[00000]EPIK: Created new measurement archive ./epik_bt_W_16_sum
[00000]EPIK: EPK_FILTER "npb.filt" filtered 6 of 96 functions
[00000]EPIK: Activated ./epik_bt_W_16_sum [NO TRACE] (0.071s)

[... Application output ...]

[00000]EPIK: Closing experiment ./epik_bt_W_16_sum
[00000]EPIK: 84 unique paths (84 max paths, 4 max frames, 0 unknowns)
[00000]EPIK: Unifying... done (0.014s)
[00000]EPIK: Collating... done (0.059s)
[00000]EPIK: Closed experiment ./epik_bt_W_16_sum (0.075s)
S=C=A=N: Sun Mar 29 16:58:41 2009: Collect done (status=0) 7s
S=C=A=N: ./epik_bt_W_16_sum complete.
```
BT-MPI tuned summary analysis report score

- Scoring of new analysis report as textual output

```
% scalasca -examine -s epik_bt_W_16_sum
INFO: Post-processing runtime summarization result...
cube3_score ./epik_bt_W_16_sum/summary.cube
```

Estimated aggregate size of event trace (total_tbc): 16,852,888 bytes
Estimated size of largest process trace (max_tbc): 1,053,328 bytes
(When tracing set ELG_BUFFER_SIZE to avoid intermediate flushes or reduce requirements using filter file listing names of USR regions.)

<table>
<thead>
<tr>
<th>flt type</th>
<th>max_tbc</th>
<th>time</th>
<th>% region</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANY</td>
<td>1053328</td>
<td>98.39</td>
<td>100.00 (summary) ALL</td>
</tr>
<tr>
<td>MPI</td>
<td>642712</td>
<td>86.83</td>
<td>88.25 (summary) MPI</td>
</tr>
<tr>
<td>COM</td>
<td>197928</td>
<td>1.68</td>
<td>1.71 (summary) COM</td>
</tr>
<tr>
<td>USR</td>
<td>212688</td>
<td>9.88</td>
<td>10.04 (summary) USR</td>
</tr>
</tbody>
</table>

- Significant reduction in runtime (measurement overhead)
  - Not only reduced time for USR regions, but MPI reduced too!
- Further measurement tuning (filtering) may be appropriate
  - e.g., use “timer_*” to filter timer_start_, timer_read_, etc.
Summary analysis report exploration (filtered)

Same message statistics, but times greatly reduced

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
BT-MPI trace measurement collection...

- Re-run the application using Scalasca nexus with “-t” flag

```bash
% scalasca -analyze -t mpiexec -np 16 ./bt_W.16
S=C=A=N: Scalasca trace collection and analysis
S=C=A=N: ./epik_bt_W_16_trace experiment archive
S=C=A=N: Sun Apr  5 18:50:57 2009: Collect start
mpiexec -np 16 ./bt_W.16
[00000]EPIK: Created new measurement archive ./epik_bt_W_16_trace
[00000]EPIK: EPK_FILTER "npb.filt" filtered 6 of 96 functions
[00000]EPIK: Activated ./epik_bt_W_16_trace [10000000 bytes] (0.051s)

[... Application output ...]

[00000]EPIK: Closing experiment ./epik_bt_W_16_trace [0.016GB] (max 1053310)
[00000]EPIK: Flushed 1053330 bytes to file ./epik_bt_W_16_trace/ELG/00000
[00000]EPIK: 84 unique paths (84 max paths, 4 max frames, 0 unknowns)
[00000]EPIK: Unifying... done (0.021s)
[00013]EPIK: Flushed 1053306 bytes to file ./epik_bt_W_16_trace/ELG/00013

[00001]EPIK: Flushed 1053306 bytes to file ./epik_bt_W_16_trace/ELG/00001
[00000]EPIK: 1flush=0.001GB@2.582MB/s, Pflush=0.015GB@35.458MB/s
[00000]EPIK: Closed experiment ./epik_bt_W_16_trace (0.178s)
S=C=A=N: Sun Apr  5 18:51:05 2009: Collect done (status=0) 8s
[. continued ...]
```

- Separate trace file per MPI rank written straight into new experiment directory ./epik_bt_W_16_trace
BT-MPI trace measurement ... analysis

• Continues with automatic (parallel) analysis of trace files

    S=C=A=N: Sun Apr  5 18:51:05 2009: Analyze start
    mpiexec -np 16 scout ./epik_bt_W_16_trace
    SCOUT   Copyright (c) 1998-2009 Forschungszentrum Juelich GmbH

    Analyzing experiment archive ./epik_bt_W_16_trace

    Reading definitions file ... done (0.563s).
    Reading event trace files ... done (0.495s).
    Preprocessing             ... done (0.134s).
    Analyzing event traces    ... done (2.186s).
    Writing CUBE report       ... done (0.160s).

    Total processing time : 3.737s
    Max. memory usage      : 7.000MB

    S=C=A=N: Sun Apr  5 18:51:09 2009: Analyze done (status=0) 4s
    S=C=A=N: ./epik_bt_W_16_trace complete.

• Produces trace analysis report in experiment directory

    % scalasca -examine epik_bt_W_16_trace
    INFO: Post-processing runtime summarization result...
    INFO: Post-processing trace analysis report ...
    INFO: Displaying ./epik_bt_W_16_trace/trace.cube...
Trace analysis report exploration

Additional trace-based metrics in metric hierarchy
Further information

• Consult quick reference guide for further information

% scalasca -h
Scalasca 1.2 - quick reference guide
pdfview /UNITE/packages/scalasca/1.2/doc/manuals/QuickReference.pdf

[PDF viewer showing quick reference guide]

• CUBE GUI provides context sensitive help and on-line metric descriptions

• EPIK archive directories contain analysis report(s), measurement collection & analysis logs, etc.

• Instrumentation, measurement, analysis & presentation can all be extensively customized

• Visit www.scalasca.org or mail scalasca@fz-juelich.de
EPIK user instrumentation API

• EPIK user instrumentation API
  – #include “epik_user.h”
  – EPIK_USER_REG(epik_solve, “<<Solve>>”)
  – EPIK_USER_START(epik_solve)
  – EPIK_USER_END(epik_solve)

• Can be used to mark initialization, solver & other phases
  – Annotation macros ignored by default
  – Instrumentation enabled with “-user” flag
  – Also available for Fortran
    • #include “epik_user.inc” and use C preprocessor

• Appear as additional regions in analyses
  – Distinguishes performance of important phase from rest
EPIK measurement configuration

• Via ./EPIK.CONF file

  EPK_FILTER=smg2000.filt
  ELG_BUFFER_SIZE=40000000

• Via environment variables

  % export EPK_FILTER=smg2000.filt
  % export ELG_BUFFER_SIZE=40000000

• Via command-line flags (partially)

  % scalasca -analyze -f smg2000.filt ...

• To show current/default configuration

  % epik_conf

• Actual Scalasca measurement configuration saved in experiment archive as epik.conf
Summary analysis report with HWC metrics

EPK_METRICS=PAPI_TOT_INS:PAPI_FP_OPS:PAPI_L2_TCM:PAPI_TOT_CYC

Hardware counter metrics each shown as root metrics
CUBE algebra utilities

- Extracting solver sub-tree from analysis report
  
  ```
  % cube3_cut -r '<<SMG.Solve>>' epik_smg2000_12_trace/trace.cube
  Writing cut.cube... done.
  ```

- Calculating difference of two reports
  
  ```
  % cube3_diff epik_smg2000_12_trace/trace.cube cut.cube
  Writing diff.cube... done.
  ```

- Additional utilities for merging, calculating mean, etc.
  - Default output of cube3_utility is a new report utility.cube

- Further utilities for report scoring & statistics

- Run utility with “-h” (or no arguments) for brief usage info
Scalasca usage recap

1. Reference preparation for validation
2. Program instrumentation: skin
3. Summary measurement collection & analysis: scan [-s]
4. Summary analysis report examination: square
5. Summary experiment scoring: square -s
6. Event trace collection & analysis: scan -t
7. Event trace analysis report examination: square
   • General usage/help: scalasca [-h]
   • Instrumentation, measurement, analysis & presentation can all be extensively customized

• Visit www.scalasca.org or mail scalasca@fz-juelich.de
Prepares application objects & executables for measurement

- `skin = scalasca -instrument`
- `skin [options] <compile-or-link-command>`
  - defaults to automatic function instrumentation by compiler
    - available for most compilers, but not all
  - for OpenMP, includes source-level pre-processing of directives to insert POMP instrumentation
- `[-pomp]`
  - source-level pre-processing of OpenMP & POMP directives *instead* of automatic compiler instrumentation
- `[-user]`
  - additionally enable EPIK user instrumentation API
  - offers complementary program structure information for analyses via user-provided annotations (e.g., phases, loops, ...)

SC '09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
scan – Scalasca measurement collection/analysis

• Runs application under control of measurement system to collect and analyze an execution experiment
  – scan = scalasca -analyze
  – scan [options] <application-launch-command>
    • e.g., scan [options] [$MPIEXEC [mpiexec-options]] [target [args]]
  – [-s] collect summarization experiment [default]
  – [-t] collect event traces and then analyze them automatically
  – Additional options
    • [-e] experiment archive (directory) name
    • [-f filter] specify file listing routines to ignore during measurement
    • [-m metric1:metric2:...] include hardware counter metrics
    • [-n] preview scan and perform checks but don't execute
    • [-q] quiesce (disable most) measurement collection
    • [-a] (re-)analyze a previously collected experiment
Prepares and presents measurement analysis report(s) for interactive exploration

- square = scalasca -examine
- square [options] <experiment-archive/report>
  - e.g., square epik_title
- Post-processes intermediate measurement analysis reports
- Launches GUI and presents default analysis report (if multiple reports available)
  - trace analysis given precedence over summary analysis
  - select other reports via File/Open menu
- [-s] skip display and output textual score report
  - estimate total trace size and maximum rank trace size
  - breakdown of USR vs. MPI/OMP vs. COM region requirements
Performance analysis & tuning case studies
Additional Live-DVD example experiments

• Example experiment archives provided for examination (see workshop-scalasca/Examples subdirectory):
  – jugene_sweep3d
    • 294,912 MPI processes on BG/P (trace)
  – jump_zeusmp2
    • 512 MPI processes on p690 cluster (summary & trace)
  – marenostrum_wrf-nmm
    • 1600 MPI processes on JS21 blade cluster, solver extract
    • summary analysis with 8 PowerPC hardware counters
    • trace analysis showing NxN completion problem on some blades
  – neptun_jacobi
    • 12 MPI processes, or 12 OpenMP threads, or 4x3 hybrid parallelizations implemented in C, C++ & Fortran on SGI Altix
  – ranger_smg2000
    • 12,288 MPI processes on Sun Constellation cluster, solve extract
Scalasca NPB-BT experiments

• Comparison of NPB-BT class A in various configurations run on a single dedicated 16-core cluster compute node
  – 16 MPI processes
    • optionally built using MPI File I/O (e.g., SUBTYPE=full)
    • optionally including PAPI counter metrics in measurement (e.g., EPK_METRICS=PAPI_FP_OPS:DISPATCHSTALLS)
  – 16 OpenMP threads
  – 4 MPI processes each with 4 OpenMP threads (MZ-MPI)
• NPB-BT-MZ class B on Cray XT5 (8-core compute nodes)
  – 32 MPI processes with OMP_NUM_THREADS=8
    • More threads created on some processes (and fewer on others) as application attempts to balance work distribution
• NPB-MPI-BT on BlueGene/P with 144k processes
  – 1536x1536x1536 gridpoints distributed on 384x384 processes
16-process trace analysis
16-process summary analysis with HWC metrics

EPK_METRICS = PAPI_TOT_CYC:PAPI_TOT_INS:PAPI_FP_OPS:DISPATCH_STALLS

CH 3.0 QT: epik_bt_A_16_sum_PAPI/summary.cube.gz

EPK_METRICS = PAPI_TOT_CYC:PAPI_TOT_INS:PAPI_FP_OPS:DISPATCH_STALLS

Metric tree

Call tree

Absolute

Absolute

Peer percent

System tree

Topology 0

Floating point operations.

[RETIRRED_SSE_OPERATIONS:SINGLE_ADD_SUB_OPS:SINGLE_MUL_OPS:DOUBLE_ADD_SUB_OPS:DOUBLE_MUL_OPS]

OK
16-process summary analysis: MPI File I/O time

BT-MPI SUBTYPE=full using collective MPI file I/O has imbalance during read

MPI file statistics
16-process summary analysis: MPI File I/O time

BT-MPI SUBTYPE=simple using individual MPI file I/O is balanced but much slower

MPI file statistics
16-thread summary analysis: Execution time

Thread 15 finishes its work fastest...
16-thread summary analysis: Implicit barrier time

... but must then wait longest at end of loop
16-thread summary analysis: Thread fork time

Less than 1% overhead for thread management
16-thread summary analysis: Idle threads time

99.74% of execution time found in parallel regions
Fourth thread in team generally has lighter load
4x4 summary analysis: OpenMP time

9% OpenMP time mostly found at implicit barriers
4x4 summary analysis: Idle threads time

9% of total time wasted with idle/unused threads

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
4x4 summary analysis: MPI time

2.75% MPI time only found on master threads
4x4 combined summary & trace analysis
32x8 summary analysis: Excl. execution time

<table>
<thead>
<tr>
<th>Metric tree</th>
<th>Absolute</th>
<th>Peer percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00 Time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1581.71 Execution</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.04 MPI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39.28 Synchronization</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00 Communication</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1456.00 Point-to-point</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.43 Collective</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00 File I/O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.33 Init/Exit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00 OMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00 Flush</td>
<td></td>
<td></td>
</tr>
<tr>
<td>827.66 Management</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00 Synchronization</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00 Barrier</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00 Explicit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1275.00 Implicit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.01 Critical</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00 Lock API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.37 Overhead</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10499.94 Idle threads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>103.91 Limited parallelism</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Generally good process and thread load balance**
32x8 summary analysis: Limited parallelism

Even through a number of threads are never used
However, serial execution sections are prevalent...
32x8 summary analysis: MPI communication time

... typically while master thread communicating
Thread imbalance also results in substantial loss
32x8 summary analysis: Thread management

Thread management cost high with over-subscription
### 32x8 summary analysis: Critical section time

<table>
<thead>
<tr>
<th>Metric tree</th>
<th>Call tree</th>
<th>Peer percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00 Time</td>
<td>0.00 MPI_Waitall</td>
<td>0.00 100.00 100.00</td>
</tr>
<tr>
<td>0.04 MPI</td>
<td>0.00 adi</td>
<td>0.00 100.00 100.00</td>
</tr>
<tr>
<td>39.28 Synchronization</td>
<td>0.00 compute_rhs</td>
<td>0.00 100.00 100.00</td>
</tr>
<tr>
<td>0.00 Communication</td>
<td>0.00 x_solve</td>
<td>0.00 100.00 100.00</td>
</tr>
<tr>
<td>1456.00 Point-to-point</td>
<td>0.00 y_solve</td>
<td>0.00 100.00 100.00</td>
</tr>
<tr>
<td>0.43 Collective</td>
<td>0.00 z_solve</td>
<td>0.00 100.00 100.00</td>
</tr>
<tr>
<td>0.00 File I/O</td>
<td>0.00 add</td>
<td>0.00 100.00 100.00</td>
</tr>
<tr>
<td>2.33 Init/Exit</td>
<td>0.00 MPI_BARRIER</td>
<td>0.00 100.00 100.00</td>
</tr>
<tr>
<td>0.00 OMP</td>
<td>0.00 error_norm</td>
<td>0.00 100.00 100.00</td>
</tr>
<tr>
<td>0.00 Flush</td>
<td>0.00 $omp parallel @error.f:27</td>
<td>0.00 100.00 100.00</td>
</tr>
<tr>
<td>827.66 Management</td>
<td>0.00 $omp do @error.f:33</td>
<td>0.00 100.00 100.00</td>
</tr>
<tr>
<td>0.00 Synchronization</td>
<td>0.01 $omp atomic @error.f:51</td>
<td>0.00 100.00 100.00</td>
</tr>
<tr>
<td>0.00 Barrier</td>
<td>0.00 $omp barrier @error.f:54</td>
<td>0.00 100.00 100.00</td>
</tr>
<tr>
<td>0.00 Explicit</td>
<td>0.00 compute_rhs</td>
<td>0.00 100.00 100.00</td>
</tr>
<tr>
<td>1275.00 Implicit</td>
<td>0.00 rhs_norm</td>
<td>0.00 100.00 100.00</td>
</tr>
<tr>
<td>0.01 Critical</td>
<td>0.01 $omp atomic @error.f:104</td>
<td>0.00 100.00 100.00</td>
</tr>
<tr>
<td>0.00 Lock API</td>
<td>0.00 $omp barrier @error.f:107</td>
<td>0.00 100.00 100.00</td>
</tr>
<tr>
<td>0.00 Overhead</td>
<td>0.00 MPI_Reduce</td>
<td>0.00 100.00 100.00</td>
</tr>
<tr>
<td>10499.94 Idle threads</td>
<td>0.00 100.00 100.00</td>
<td></td>
</tr>
<tr>
<td>103.91 Limited parallelism</td>
<td>0.00 100.00 100.00</td>
<td></td>
</tr>
</tbody>
</table>

Atomic statements during verification are efficient.
NPB-MPI-BT on BlueGene/P case study

• 3D solution of unsteady, compressible Navier-Stokes eqs
  – NASA NAS parallel benchmark suite Block-Tridiagonal solver
  – series of ADI solve steps in X, Y & Z dimensions
  – ~9,500 lines (20 source modules), mostly Fortran 77

• Run on IBM BlueGene/P in VN mode with 144k processes
  – Good scaling when problem size matched to architecture
    • 1536x1536x1536 gridpoints mapped onto 384x384 processes
  – Measurement collection took 53 minutes
  – 38% dilation for summarization measurement compared to uninstrumented execution (using 10 function filter)
  – MPI trace size would be 18.6 TB
  – 25% of time in ADI is point-to-point communication time
    • 13% copy_faces, 23% x_solve, 33% y_solve, 31% z_solve
  – 128s for a single MPI_Comm_split during setup
NPB-MPI-BT on jugene@144k summary analysis

Highest waiting times in corners and along edges of BG/P torus
NPB-MPI-BT on jugene@144k summary analysis

Regular grid pattern with curious crossover along middle partition
AMMP on Altix case study

• Molecular mechanics simulation
  – original version developed by Robert W. Harrison
• SPEC OMP benchmark parallel version
  – ~14,000 lines (in 28 source modules): 100% C
• Run with 32 threads on SGI Altix 4700 at TUD-ZIH
  – Built with Intel compilers
  – 333 simulation timesteps for 9582 atoms
• Scalasca summary measurement
  – Minimal measurement dilation
  – 60% of total time lost in synchronization with lock API
  – 12% thread management overhead
ammp on jupiter@32 OpenMP lock analysis

Lots of explicit lock synchronization is a scalability inhibitor
OpenMP metrics reworked with v1.2

Thread management costs vary by parallel region & num_threads
WRF/MareNostrum case study

- Numerical weather prediction
  - public domain code developed by US NOAA
  - flexible, state-of-the-art atmospheric simulation
  - Non-hydrostatic Mesoscale Model (NMM)
- MPI parallel version 2.1.2 (Jan-2006)
  - >315,000 lines (in 480 source modules): 75% Fortran, 25% C
- Eur-12km dataset configuration
  - 3-hour forecast (360 timesteps) with checkpointing disabled
- Run with 1600 processes on MareNostrum
  - IBM BladeCenter cluster at BSC
- Scalasca summary and trace measurements
  - 15% measurement dilation with 8 hardware counters
  - 23GB trace analysis in 5 mins
WRF on MareNostrum@1600 with HWC metrics

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Selection percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00 Time</td>
<td>1.3 solve_nmm</td>
</tr>
<tr>
<td>0.00 MPI</td>
<td></td>
</tr>
<tr>
<td>0.00 Synchronization</td>
<td></td>
</tr>
<tr>
<td>0.00 Communication</td>
<td></td>
</tr>
<tr>
<td>3639.53 Collective</td>
<td></td>
</tr>
<tr>
<td>15824.94 Point-to-point</td>
<td></td>
</tr>
<tr>
<td>0.00 MPI I/O</td>
<td>0.2 module_tiles</td>
</tr>
<tr>
<td>0.00 Init/Exit</td>
<td>1.2 rsl_lite_init_exch</td>
</tr>
<tr>
<td>0.00 Overhead</td>
<td>9.5 rsl_lite_pack</td>
</tr>
<tr>
<td>224096053 Visits</td>
<td>3.4 rsl_lite_exch_y</td>
</tr>
<tr>
<td>0 Synchronizations</td>
<td>3.7 rsl_lite_exch_x</td>
</tr>
<tr>
<td>50227200</td>
<td>13.5 module_igwave_adjust_prdht</td>
</tr>
<tr>
<td>1390691467550</td>
<td>0.6 module_igwave_adjust_ddamp</td>
</tr>
<tr>
<td>50274153042403 CYCLES</td>
<td>0.3 module_igwave_adjust_pde</td>
</tr>
<tr>
<td>21015191824572 DATA_ACCESS</td>
<td>11.1 module_advection_adve</td>
</tr>
<tr>
<td>0 DATA_LOAD</td>
<td>0.4 module_igwave_adjust_vtca</td>
</tr>
<tr>
<td>1689476387779 LOAD_HIT_L1$</td>
<td>0.7 module_nonhy_dynam_vad2</td>
</tr>
<tr>
<td>245601672754 LOAD_HIT_L2$</td>
<td>3.0 module_nonhy_dynam_hadz</td>
</tr>
<tr>
<td>10148888948 LOAD_HIT_MEM</td>
<td>4.3 module_nonhy_dynam_eps</td>
</tr>
<tr>
<td>245,801,627,254 (1.4%)</td>
<td>0.7 module_advection_vad2</td>
</tr>
<tr>
<td>1.715e+13</td>
<td>3.7 module_advection_had2</td>
</tr>
<tr>
<td>25,778,752,076 (10.5%)</td>
<td>4.0 module_physics_calls_radiation</td>
</tr>
<tr>
<td>2,458e+11</td>
<td>0.3 rdtemp</td>
</tr>
<tr>
<td></td>
<td>10.5 module_physics_calls_turbi</td>
</tr>
<tr>
<td></td>
<td>0.1 module_physics_calls_uv_h_to</td>
</tr>
</tbody>
</table>

Distribution of data load hits in level 2 cache on application MPI 2D grid topology.
WRF on MareNostrum@1600 trace analysis

MareNostrum JS21 topology shows blades

Imbalance at exit from Allreduce
Some ranks require extra 1.75s to complete 51st MPI_Allreduce.
WRF/MareNostrum experience

- Limited system I/O requires careful management
  - Selective instrumentation and measurement filtering
- PowerPC hardware counter metrics included in summary
  - Memory/cache data access hierarchy constructed
- Automated trace analysis quantified impact of imbalanced exit from MPI_Allreduce in “NxN completion time” metric
  - Intermittent but serious MPI library/system problem, that restricts application scalability
  - Only a few processes directly impacted, however, communication partners also quickly blocked
- Presentation using logical and physical topologies
  - MPI Cartesian topology provides application insight
  - Hardware topology helps localize system problems
XNS on BlueGene/L case study

• CFD simulation of unsteady flows
  – developed by RWTH CATS group of Marek Behr
  – exploits finite-element techniques, unstructured 3D meshes, iterative solution strategies

• MPI parallel version (Dec-2006)
  – >40,000 lines of Fortran & C
  – DeBakey blood-pump dataset (3,714,611 elements)
XNS-DeBakey on jubl@4096 summary analysis

- Point-to-point msgs w/o data
- Masses of P2P synch operations
- Primarily in scatter & gather
- Processes all equally responsible

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
XNS-DeBakey scalability on BG/L

Original performance peaked at 132 ts/hr

3.5x overall improvement to 461 ts/hr
XNS on BlueGene/L experience

- Globally synchronized high-resolution clock facilitates efficient measurement & analysis
- Restricted compute node memory limits trace buffer size and analyzable trace size
- Summarization identified bottleneck due to unintended P2P synchronizations (messages with zero-sized payload)
- 4x solver speedup after replacing MPI_Sendrecv operations with size-dependant separate MPI_Send and MPI_Recv
- Significant communication imbalance remains due to mesh partitioning and mapping onto processors
- MPI_Scan implementation found to contain implicit barrier
  - responsible for 6% of total time with 4096 processes
  - decimated when substituted with simultaneous binomial tree
PEPC-B on BG/P & Cray XT case study

• Coulomb solver used for laser-plasma simulations
  – Developed by Paul Gibbon (JSC)
  – Tree-based particle storage with dynamic load-balancing

• MPI version
  – PRACE benchmark configuration, including file I/O

• Run on BlueGene/P in dual mode with 1024 processes
  – 2 processes per quad-core PowerPC node, 1100 seconds
  – IBM xl compilers, MPI library and torus/tree interconnect

• Run on Cray XT in VN (4p) mode with 1024 processes
  – 4 processes per quad-core Opteron node, 360 seconds
  – PGI compilers and Cray MPI, CNL, SeaStar interconnect
PEPC@1024 on BlueGene/P: Wait at \( N \times N \) time

Time waiting for last rank to enter MPI_Allgather

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
PEPC@1024 on Cray XT4: Wait at NxN time

Time waiting for last rank to enter MPI_Allgather
PEPC-B on BG/P & Cray XT experience

• Despite very different processor and network performance, measurements and analyses can be easily compared
  – different compilers affect function naming & in-lining
• Both spend roughly two-thirds of time in computation
  – tree_walk has expensive computation & communication
• Both waste 30% of time waiting to enter MPI_Barrier
  – not localized to particular processes, since particles are regularly redistributed
• Most of collective communication time is also time waiting for last ranks to enter MPI_Allgather & MPI_Alltoall
  – imbalance for MPI_Allgather twice as severe on BlueGene/P, however, almost 50x less for MPI_Alltoall
  – collective completion times also notably longer on Cray XT
Sweep3d on BlueGene/P case study

- 3D neutron transport simulation
  - ASC benchmark
  - direct order solve uses diagonal sweeps through grid cells
- MPI parallel version 2.2b using 2D domain decomposition
  - ~2,000 lines (12 source modules), mostly Fortran77
- Run on IBM BlueGene/P in VN mode with 288k processes
  - 790 GB trace written in 47 minutes, analyzed in 7 minutes
    - plus 86 minutes just to create 294,912 files (one per MPI rank)
    - SIONlib being developed to address management of sets of files
  - Mapping of 576x512 grid of processes onto 3D physical torus results in regular performance artifacts
sweep3d on jugene@288k trace analysis

Computation time of 86.71±1.8% seconds varies by BG/P node
sweep3d on jugene@288k trace analysis

Computation time distribution likely due to mapping onto torus
sweep3d on jugene@288k trace (wait) analysis

Late Receiver time complements sweep computation time
Acknowledgements

• Ths Scalasca development team
• The application and benchmark developers who provided their codes and/or measurement archives
• The facilities who made their HPC resources available
  – ALCF, BSC, CSC, CSCS, EPCC, JSC, HLRN, HLRS, LRZ, NCAR, NCCS/ORN, NICS, RWTH, RZG, SARA, TeraGrid/TACC, TUD/ZIH, UTK/ICL
Scalable performance analysis of large-scale parallel applications
– toolset for scalable performance measurement & analysis of MPI, OpenMP & hybrid parallel applications
– supporting most popular HPC computer systems
– available under New BSD open-source license
– sources, documentation & publications:
  • http://www.scalasca.org
  • mailto: scalasca@fz-juelich.de
Event Tracing with VAMPIRIRTRACE & VAMPIR

Andreas Knüpfer
Technical University Dresden
Overview

• Introduction
• Vampir Displays
• VampirTrace Instrumentation & Measurement
• Hands-on
  – First Steps
  – Buffer Management
  – Filtering and Grouping
  – PAPI Hardware Performance Counters
• Finding Performance Bottlenecks
• Conclusion & Outlook
Overview

• Introduction
• Vampir Displays
• VampirTrace Instrumentation & Measurement
• Hands-on
  – First Steps
  – Buffer Management
  – Filtering and Grouping
  – PAPI Hardware Performance Counters
• Finding Performance Bottlenecks
• Conclusion & Outlook
Introduction

Why bother with performance analysis?
• Well, why are you here after all?
• Efficient usage of expensive and limited resources
• Scalability to achieve next bigger simulation

Profiling and Tracing
• Have an optimization phase!
  – just like the testing and debugging phases
• Use tools!
• Avoid do-it-yourself-with-printf solutions, really!
Introduction: Profiling & Tracing

Program Instrumentation
• Detect run-time events (points of interest)
• Pass information to run-time measurement library

Profile Recording
• Collect aggregated information (Time, Counts, ...)
• About program and system entities
  – functions, loops, basic blocks
  – application, processes, threads, ...

Trace Recording
• Save individual event records together with precise time stamp and process or thread ID
• Plus event specific information
Event Trace Visualization

Trace Visualization

• Alternative and supplement to automatic analysis
• Show dynamic run-time behavior graphically
• Provide statistics and performance metrics
  — global timeline for parallel processes/threads
  — process timeline plus performance counters
  — statistic summary display
  — message statistics
  — more
• Interactive browsing, zooming, selecting
  — adapt statistics to zoom level (time interval)
  — also for very large and highly parallel traces
VampirServer Architecture

Parallel Program

Monitor System

Process

Event Streams

File System

Trace 1
Trace 2
Trace 3
Trace N

Parallel I/O

Message Passing

Analysis Server

Worker 1

Worker 2

Worker m

Visualization Client

Timeline with 16 Traces visible

768 Processes

Thumbnail View

768 Processes

Segment Indicator

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
Overview

• Introduction
• Vampir Displays
• VampirTrace Instrumentation & Measurement
• Hands-on  
  – First Steps  
  – Buffer Management  
  – Filtering and Grouping  
  – PAPI Hardware Performance Counters
• Finding Performance Bottlenecks
• Conclusion & Outlook
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
Process Timeline with Counters
### Statistic Summary Display

![Statistic Summary Display](image)

<table>
<thead>
<tr>
<th>Name</th>
<th>Token</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>hypre_StructMatvecCompute</td>
<td>299</td>
<td>2:15.199</td>
</tr>
<tr>
<td>hypre_StructAxpy</td>
<td>306</td>
<td>2:10.744</td>
</tr>
<tr>
<td>hypre_StructInnerProd</td>
<td>295</td>
<td>1:14.087</td>
</tr>
<tr>
<td>MPI_Finalize</td>
<td>82</td>
<td>1:04.179</td>
</tr>
<tr>
<td>hypre_StructCopy</td>
<td>297</td>
<td>51.516 s</td>
</tr>
<tr>
<td>MPI_Waitall</td>
<td>163</td>
<td>20.135 s</td>
</tr>
<tr>
<td>hypre_StructVectorSetConstantV</td>
<td>303</td>
<td>20.124 s</td>
</tr>
<tr>
<td>hypre_StructScale</td>
<td>308</td>
<td>15.980 s</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>9</td>
<td>13.283 s</td>
</tr>
<tr>
<td>MPI_Isend</td>
<td>115</td>
<td>9.010 s</td>
</tr>
<tr>
<td>hypre_StructMatrixSetBoxValues</td>
<td>229</td>
<td>8.455 s</td>
</tr>
<tr>
<td>sync</td>
<td>2</td>
<td>5.654 s</td>
</tr>
<tr>
<td>main</td>
<td>184</td>
<td>4.661 s</td>
</tr>
<tr>
<td>hypre_CAlloc</td>
<td>186</td>
<td>2.050 s</td>
</tr>
<tr>
<td>hypre_StructVectorSetBoxValues</td>
<td>260</td>
<td>1.827 s</td>
</tr>
<tr>
<td>hypre_StructMatrixInitializeDat</td>
<td>224</td>
<td>0.738 s</td>
</tr>
<tr>
<td>hypre_StructKrylovAxpy</td>
<td>305</td>
<td>0.668 s</td>
</tr>
<tr>
<td>MPI_Init</td>
<td>108</td>
<td>0.436 s</td>
</tr>
<tr>
<td>hypre_StructKrylovCopyVector</td>
<td>296</td>
<td>0.221 s</td>
</tr>
<tr>
<td>hypre_StructKrylovMatvec</td>
<td>298</td>
<td>0.215 s</td>
</tr>
<tr>
<td>hypre_PCGSolve</td>
<td>293</td>
<td>0.212 s</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>113</td>
<td>0.190 s</td>
</tr>
<tr>
<td>hypre_BoxGetSize</td>
<td>227</td>
<td>0.182 s</td>
</tr>
<tr>
<td>hypre_Free</td>
<td>187</td>
<td>0.169 s</td>
</tr>
<tr>
<td>hypre_InitializeCommunication</td>
<td>250</td>
<td>0.160 s</td>
</tr>
<tr>
<td>MPI_Waitall</td>
<td>327</td>
<td>0.131 s</td>
</tr>
</tbody>
</table>

Sorted by Value Down All Symbols: Exclusive Times
Summary Timeline Display
Message Statistics Display

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
Collective Operation Statistics
Counter Timeline Display

SC '09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
Call Tree Display
Overview

• Introduction
• Vampir Displays
• VampirTrace Instrumentation & Measurement
• Hands-on
  – First Steps
  – Buffer Management
  – Filtering and Grouping
  – PAPI Hardware Performance Counters
• Finding Performance Bottlenecks
• Conclusion & Outlook
Profiling and Tracing

• Tracing Advantages
  – preserve temporal and spatial relationships
  – allow reconstruction of dynamic behavior on any required abstraction level
  – profiles can be calculated from trace

• Tracing Disadvantages
  – traces can become very large
  – may cause perturbation
  – instrumentation and tracing is complicated (event buffering, clock synchronization, ...)

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
Common Event Types

• Enter/leave of function/routine/region
  – time, process/thread, function ID

• Send/receive of P2P message (MPI)
  – time, sender, receiver, length, tag, comm.

• Collective communication (MPI)
  – time, process, root, communicator, # bytes

• Hardware performance counter values
  – time, process, counter ID, value
Instrumentation

- Instrumentation: Process of modifying programs to detect and report events
  - call instrumentation functions
  - provided by trace library
  - call for every run-time event of interest

- there are various ways of instrumentation
What do you need to do for it?

- Instrumentation (automatic with compiler wrappers)

  \[
  \begin{align*}
  &\text{CC}=\text{icc} \\
  &\text{CXX}=\text{icpc} \\
  &\text{F90}=\text{ifc} \\
  &\text{MPICC}=\text{mpicc}
  \end{align*}
  \]

  \[
  \begin{align*}
  &\text{CC}=\text{vtcc} \\
  &\text{CXX}=\text{vtcxx} \\
  &\text{F90}=\text{vtf90} \\
  &\text{MPICC}=\text{vtcc}
  \end{align*}
  \]

- Re-compile & re-link
- Trace Run (run with appropriate test data set)
- More details later
What does VampirTrace do?

• Instrumentation:
  – via compiler wrappers
  – by underlying compiler with specific options
  – MPI instrumentation with replacement lib
  – OpenMP instrumentation with Opari
  – also binary instrumentation with Dyninst
  – partial manual instrumentation
What does VampirTrace do?

• Trace run:
  – event data collection
  – precise time measurement
  – parallel timer synchronization
  – collecting parallel process/thread traces
  – collecting performance counters (from PAPI, memory usage, POSIX I/O calls and fork/system/exec calls, and more ...)
  – filtering and grouping of function calls
The Tools

• VampirTrace
  – convenient instrumentation and measurement
  – hides away complicated details
  – provides many options and switches for experts
• VampirTrace is part of Open MPI 1.3

• Vampir & VampirServer
  – interactive trace visualization and analysis
  – intuitive browsing and zooming
  – scalable to large trace data sizes (100GB)
  – scalable to high parallelism (2000 processes)
• Vampir for Windows in progress, beta available
The Tools

The Open Trace Format (OTF)

• Open source trace file format

• Available at [http://www.tu-dresden.de/zih/otf/](http://www.tu-dresden.de/zih/otf/)

• Includes powerful libotf for reading/parsing/writing in custom applications

• Multi-level API:
  – High level interface for analysis tools
  – Low level interface for trace libraries

• Actively developed at TU Dresden in cooperation with the University of Oregon and the Lawrence Livermore National Laboratory (LLNL)
Overview

• Introduction
• Vampir Displays
• VampirTrace Instrumentation & Measurement
• Hands-on
  – First Steps
  – Buffer Management
  – Filtering and Grouping
  – PAPI Hardware Performance Counters
• Finding Performance Bottlenecks
• Conclusion & Outlook
Hands-on: First Steps

• Default build and normal run
• Standard prerequisite, never even try tracing if this fails

%> cd NPB3.3-MPI/BT/
%> gedit config/make.def               # look at plain setup
%> make clean
%> make bt NPROCS=16 CLASS=W
%> time mpirun -np 16 ./bin/bt.W.16

Thinkpad Laptop 13.178s, SGI Altix 1.761s
Hands-on: First Steps

• Build and run with VampirTrace

%> gedit config/make.def               # look at plain setup
   MPIF77 = vtf77 -vt:verbose -vt:f77 mpif77
   MPICC  = vtcc -vt:verbose -vt:cc mpicc

%> make clean
%> make bt NPROCS=16 CLASS=W
%> export VT_FILE_PREFIX=bt_1_initial
%> time mpirun -np 16 ./bin/bt.W.16

Thinkpad Laptop 1m33.712s, SGI Altix 2m21.331s
Hands-on: First Steps

• Start VampirServer

  %> mpirun –np 5 vngd

• Start client in separate terminal

  %> vng &

• Connect client: File → Connect Server
Hands-on: First Vampir Displays
Hands-on: First Vampir Displays
Hands-on: First Vampir Displays

<table>
<thead>
<tr>
<th>Process</th>
<th>0</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.5 M</td>
</tr>
<tr>
<td>0</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1</td>
<td>235.7</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>2</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>3</td>
<td>1.235 M</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>1.441 M</td>
</tr>
<tr>
<td>4</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>5</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>6</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>1.073 M</td>
<td>---</td>
</tr>
<tr>
<td>7</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>8</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>1.356 M</td>
<td>---</td>
</tr>
<tr>
<td>9</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>1.635 M</td>
<td>---</td>
</tr>
<tr>
<td>10</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>1.172 M</td>
<td>1.279 M</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>11</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>12</td>
<td>1.183 M</td>
<td>1.375 M</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>1.954 M</td>
</tr>
<tr>
<td>13</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>14</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>15</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>325.7 M</td>
</tr>
<tr>
<td>Summary</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>1.21 M</td>
<td>---</td>
</tr>
</tbody>
</table>
Hands-on: Trace Buffer

• Allow multiple flushes of the trace buffer (limited to one by default)

%> cd NPB3.3-MPI/BT/

%> export VT_FILE_PREFIX=bt_2_multiple_flushes
%> export VT_MAX_FLUSHES=10
%> time mpirun -np 16 ./bin/bt.W.16

Thinkpad Laptop 3m41.346s, SGI Altix 2m23.154s
Hands-on: Trace Buffer
Hands-on: Trace Buffer
Hands-on: Trace Buffer

- Increase trace buffer size
- Not more than the available memory minus the application’s allocation

```bash
%> export VT_FILE_PREFIX=bt_3_buffer_120M
%> export VT_MAX_FLUSHES=1
%> export VT_BUFFER_SIZE=120M # with 2GB RAM
%> time mpirun -np 16 ./bin/bt.W.16
```

Thinkpad Laptop 3m42.402s, SGI Altix 2m24.103s
Hands-on: Trace Buffer

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
Hands-on: Trace Buffer
Hands-on: Trace Filtering

• Filter function calls from trace

%> vtfilter -gen -fo filter.txt -r 10 -stats -p bt_3_buffer_120M
%> gedit filter.txt

• Automatically generate filter file to receive approx. 10% of the original trace size
• Create manually or modify if desired
• Wildcards allowed

• The vtfilter tool can also shrink existing traces
Hands-on: Trace Filtering

- Re-run with filter specification

```bash
%> export VT_FILE_PREFIX=bt_4_filter
%> export VT_FILTER_SPEC=filter.txt
%> time mpirun -np 16 ./bin/bt.W.16
```

Thinkpad Laptop 0m33.467s, SGI Altix 0m33.082s
Hands-on: Trace Filtering
Hands-on: Trace Filtering

SC '09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS

POINT

VI-HPS
Hands-on: Grouping

• Grouping of functions

```bash
%> export VT_GROUPS_SPEC = /home/user/groups.txt
%> time mpirun -np 16 ./bin/bt.W.16
```

| CALC=calculate | MISC=my*;test | UNKNOWN=* |

• Allows wildcards

• Groups get separate colors in Vampir
Hands-on: PAPI Counters

• Activate PAPI if available

```bash
%> export VT_FILE_PREFIX=bt_5_papi
%> papi_avail
%> papi_event Chooser PRESET PAPI_FP_OPS
%> export VT_METRICS=PAPI_FP_OPS:PAPI_L2_TCM
%> time mpirun -np 16 ./bin/bt.W.16
```

• Use papi_avail and papi_native_avail to find available counters

Thinkpad Laptop 0m46.120s, SGI Altix 0m38.098s
Hands-on: PAPI Counters

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
Hands-on: PAPI Counters
Hands-on: PAPI Counters
Control options by environment variables:

- **VT_PFORM_GDIR**: Directory for final trace files
- **VT_PFORM_LDIR**: Directory for intermediate files
- **VT_FILE_PREFIX**: Trace file name
- **VT_BUFFER_SIZE**: Internal trace buffer size
- **VT_MAX_FLUSHES**: Max number of buffer flushes
- **VT_MEMTRACE**: Enable memory allocation tracing
- **VT_IOTRACE**: Enable I/O tracing
- **VT_MPITRACE**: Enable MPI tracing
- **VT_FILTER_SPEC**: Name of filter definition file
- **VT_GROUPS_SPEC**: Name of grouping definition file
- **VT_METRICS**: PAPI counter selection
Overview

• Introduction
• Vampir Displays
• VampirTrace Instrumentation & Measurement
• Hands-on
  – First Steps
  – Buffer Management
  – Filtering and Grouping
  – PAPI Hardware Performance Counters
• Finding Performance Bottlenecks
• Conclusion & Outlook
Finding Bottlenecks

• Trace Visualization
  – Vampir provides a number of display types
  – each allows many different options

• Advice
  – identify essential parts of an application (initialization, main iteration, I/O, finalization)
  – identify important components of the code (serial computation, MPI P2P, collective MPI, OpenMP)
  – make a hypothesis about performance problems
  – consider application's internal workings if known
  – select the appropriate displays
  – use statistic displays in conjunction with timelines
Finding Bottlenecks

- Communication
- Computation
- Memory, I/O, etc
- Tracing itself
Bottlenecks in Communication

– communication as such (dominating over computation)
– late sender, late receiver
– point-to-point messages instead of collective communication
– unmatched messages
– overcharge of MPI’s buffers
– bursts of large messages (bandwidth)
– frequent short messages (latency)
– unnecessary synchronization (barrier)

The above usually result in a high MPI time share
Bottlenecks in Communication

Prevalent communication

SC '09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
Bottlenecks in Communication

Prevalent communication: MPI_Allreduce

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
Bottlenecks in Communication

Prevalent communication: Timeline view

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
Bottlenecks in Communication

Propagated Delays in MPI_SendReceiveReplace

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
Bottlenecks in Communication

Unnecessary MPI_Barsriers

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
Bottlenecks in Communication

Patterns of successive MPI_Allreduce Calls

SC ’09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
Further Bottlenecks

• Unbalanced computation  
  – single late comer

• Strictly serial parts of program  
  – idle processes/threads

• Very frequent tiny function calls
 • Sparse loops
Further Bottlenecks

Idle OpenMP threads
Bottlenecks in Computation

• Memory bound computation
  – inefficient L1/L2/L3 cache usage, TLB misses
  – detectable via HW performance counters

• I/O bound computation
  – slow input/output
  – sequential I/O with a single process
  – I/O load imbalance

• Exception handling
Bottlenecks in Computation

Low FP rate due to heavy cache misses
Bottlenecks in Computation

Low FP rate due to heavy FP exceptions
Bottlenecks in Computation

Irregular slow I/O operations

SC '09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
Effects due to Tracing Itself

• Measurement overhead
  – esp. grave for tiny function calls
  – solve with selective instrumentation
• Frequent/asynchronous trace buffer flushes
• Too many concurrent counters

• Heisenbugs
Effects due to Tracing Itself

- Buffer flushes are marked explicitly
- Harmless at the end of a trace
Overview

• Introduction
• Vampir Displays
• VampirTrace Instrumentation & Measurement
• Hands-on
  – First Steps
  – Buffer Management
  – Filtering and Grouping
  – PAPI Hardware Performance Counters
• Finding Performance Bottlenecks
• Conclusion & Outlook
Conclusions

• Performance analysis is very important
• Use tools!
• Do not spend effort in DIY solutions or printf-debugging
• Use tracing with some precautions
  – overhead
  – data volume
• Let us know about problems and feature wishes: vampirsupport@zih.tu-dresden.de
Vampir and VampirTraces are available at http://www.vampir.eu and http://www.tu-dresden.de/zih/vampirtrace/, support via vampirsupport@zih.tu-dresden.de
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

- Green: Bigben
- Blue: Ideal

Sc '09: Productive Performance Engineering of Petascale Applications with POINT and VI-HPS
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!
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
• 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
• 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:

```plaintext
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.ppk (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:

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:

double precision function scalar(u,v)
double precision u(3),v(3)
    call TAUPROFILE_TIMER(profiler, 'SCALAR [...])'
call TAUPROFILE_START(profiler)
    scalar=u(1)*v(1)+u(2)*v(2)+u(3)*v(3)
call TAUPROFILE_STOP(profiler)
return
    call TAUPROFILE_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
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
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’
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

Observe multiple causes of load imbalance, as well as the serial bottleneck
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
Repeating pattern for each timestep
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 choice of (serial and parallel) algorithm is 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
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

• 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