Gaining Insight into Parallel Program Performance using HPCToolkit

John Mellor-Crummey
Department of Computer Science
Rice University
johnmc@rice.edu

http://hpctoolkit.org
Acknowledgments

• Funding sources
  — DOE Office of Science SciDAC-2
    – Center for Scalable Application Development Software
      Cooperative agreement number DE-FC02-07ER25800
    – Performance Engineering Research Institute
      Cooperative agreement number DE-FC02-06ER25762
  — Corporate: AMD, Western Geco

• Core project team
  — Research Staff
    – Laksono Adhianto, Mike Fagan, Mark Krentel
  — Students
    – Xu Liu, Milind Chabbi, Karthik Murthy
  — Collaborator
    – Nathan Tallent (PNNL)
Challenges for Computational Scientists

• Execution environments and applications are rapidly evolving
  — architecture
    – rapidly changing multicore microprocessor designs
    – increasing scale of parallel systems
    – growing use of accelerators
  — applications
    – MPI everywhere to threaded implementations
    – adding additional scientific capabilities to existing applications
    – maintaining multiple variants or configurations for particular problems

• Steep increase in application development effort to attain performance, evolvability, and portability

• Application developers need to
  — assess weaknesses in algorithms and their implementations
  — improve scalability of executions within and across nodes
  — adapt to changes in emerging architectures
  — overhaul algorithms & data structures as needed

Performance tools can play an important role as a guide
Performance Analysis Challenges

- Complex architectures are hard to use efficiently
  - multi-level parallelism: multi-core, ILP, SIMD instructions
  - multi-level memory hierarchy
  - result: gap between typical and peak performance is huge

- Complex applications present challenges
  - for measurement and analysis
  - for understanding and tuning

- Supercomputer platforms compound the complexity
  - unique hardware
  - unique microkernel-based operating systems
  - multifaceted performance concerns
    - computation
    - communication
    - I/O
Performance Analysis Principles

- Without accurate measurement, analysis is irrelevant
  - avoid systematic measurement error
  - measure actual executions of interest, not an approximation
    - fully optimized production code on the target platform

- Without effective analysis, measurement is irrelevant
  - quantify and attribute problems to source code
  - compute insightful metrics
    - e.g., “scalability loss” or “waste” rather than just “cycles”

- Without scalability, a tool is irrelevant for supercomputing
  - large codes
  - large-scale threaded parallelism within and across nodes
Performance Analysis Goals

• Programming model independent tools

• Accurate measurement of complex parallel codes
  — large, multi-lingual programs
  — fully optimized code: loop optimization, templates, inlining
  — binary-only libraries, sometimes partially stripped
  — complex execution environments
    – dynamic loading (Linux clusters) vs. static linking (Cray, Blue Gene)
    – SPMD parallel codes with threaded node programs
    – batch jobs

• Effective performance analysis
  — insightful analysis that pinpoints and explains problems
    – correlate measurements with code for actionable results
    – support analysis at the desired level
      intuitive enough for application scientists and engineers
      detailed enough for library developers and compiler writers

• Scalable to petascale and beyond
HPCToolkit Design Principles

- Employ binary-level measurement and analysis
  - observe fully optimized, dynamically linked executions
  - support multi-lingual codes with external binary-only libraries

- Use sampling-based measurement (avoid instrumentation)
  - controllable overhead
  - minimize systematic error and avoid blind spots
  - enable data collection for large-scale parallelism

- Collect and correlate multiple derived performance metrics
  - diagnosis typically requires more than one species of metric

- Associate metrics with both static and dynamic context
  - loop nests, procedures, inlined code, calling context

- Support top-down performance analysis
  - natural approach that minimizes burden on developers
Outline

- Overview of Rice’s HPCToolkit
  - Accurate measurement
  - Effective performance analysis
  - Pinpointing scalability bottlenecks
    - scalability bottlenecks on large-scale parallel systems
    - scaling on multicore processors
  - Understanding temporal behavior
  - Assessing process variability
  - Understanding threading and memory hierarchy
    - blame shifting
    - attributing memory hierarchy costs to data
  - Summary and conclusions
HPCToolkit Workflow

- **app. source**
- **compile & link**
- **optimized binary**
  - **profile execution** [hpcrun]
  - **call stack profile**
- **binary analysis** [hpcstruct]
  - **program structure**
- **interpret profile correlate w/ source** [hpcprof/hpcprof-mpi]
- **presentation** [hpcviewer/hpctraceviewer]
- **database**
• For dynamically-linked executables on stock Linux
  — compile and link as you usually do: nothing special needed
• For statically-linked executables (e.g. for Blue Gene, Cray)
  — add monitoring by using `hpclink` as prefix to your link line
    – uses “linker wrapping” to catch “control” operations
      process and thread creation, finalization, signals, ...
• Measure execution unobtrusively
  — launch optimized application binaries
    – dynamically-linked applications: launch with `hpcrun` to measure
    – statically-linked applications: measurement library added at link time
      control with environment variable settings
  — collect statistical call path profiles of events of interest
HPCToolkit Workflow

- Analyze binary with **hpcstruct**: recover program structure
  - analyze machine code, line map, debugging information
  - extract loop nesting & identify inlined procedures
  - map transformed loops and procedures to source
HPCToolkit Workflow

- Combine multiple profiles
  — multiple threads; multiple processes; multiple executions
- Correlate metrics to static & dynamic program structure
HPCToolkit Workflow

• Presentation
  — explore performance data from multiple perspectives
    – rank order by metrics to focus on what’s important
    – compute derived metrics to help gain insight
      e.g. scalability losses, waste, CPI, bandwidth
  — graph thread-level metrics for contexts
  — explore evolution of behavior over time

interpret profile correlate w/ source
[hpccprof/hpccprof-mpi]

presentation
[hpcviewer/hpctraceviewer]
database

compile & link

app. source

optimized binary

profile execution
[hpcrun]
call stack profile

binary analysis
[hpcstruct]
program structure

[hpccrun]

Outline

• Overview of Rice’s HPCToolkit
• Accurate measurement
• Effective performance analysis
• Pinpointing scalability bottlenecks
  — scalability bottlenecks on large-scale parallel systems
  — scaling on multicore processors
• Understanding temporal behavior
• Assessing process variability
• Understanding threading and memory hierarchy
  — blame shifting
  — attributing memory hierarchy costs to data
• Summary and conclusions
Call Path Profiling

Measure and attribute costs in context
- sample timer or hardware counter overflows
- gather calling context using stack unwinding

Call path sample
- return address
- return address
- return address
- instruction pointer

Calling context tree

Overhead proportional to sampling frequency...
...not call frequency
Why Sampling?

The performance uncertainty principle implies that the accuracy of performance data is inversely correlated with the degree of performance instrumentation – Al Malony, PhD Thesis 1991

Instrumentation of MADNESS with TAU

<table>
<thead>
<tr>
<th>Method</th>
<th>Number of Profiled Events</th>
<th>Runtime (seconds)</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uninstrumented</td>
<td></td>
<td>654s</td>
<td></td>
</tr>
<tr>
<td>Compiler-based Instrumentation</td>
<td>1321</td>
<td>19625s</td>
<td>2901%</td>
</tr>
<tr>
<td>Regular Source Instrumentation</td>
<td>183</td>
<td>748s</td>
<td>14.4%</td>
</tr>
<tr>
<td>Source Instrumentation with headers (-optHeaderInst)</td>
<td>806</td>
<td>1628s</td>
<td>150%</td>
</tr>
<tr>
<td>-optHeaderInst and selective instrumentation (auto)</td>
<td>539</td>
<td>685s</td>
<td>4.7%</td>
</tr>
<tr>
<td>callpath depth 2, -optHeaderInst and selective instrumentation (auto)</td>
<td>1773</td>
<td>693s</td>
<td>6%</td>
</tr>
<tr>
<td>callpath depth 100, -optHeaderInst and selective instrumentation (auto)</td>
<td>8535</td>
<td>893s</td>
<td>36.5%</td>
</tr>
</tbody>
</table>

Figure source: http://www.nic.uoregon.edu/tau-wiki/MADNESS
**Why Sampling?**

*The performance uncertainty principle* implies that the accuracy of performance data is inversely correlated with the degree of performance instrumentation – Al Malony, PhD Thesis 1991

### Instrumentation of MADNESS with TAU

<table>
<thead>
<tr>
<th>Method</th>
<th>Number of Profiled Events</th>
<th>Runtime (seconds)</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uninstrumented</td>
<td></td>
<td>654s</td>
<td></td>
</tr>
<tr>
<td>Compiler-based Instrumentation</td>
<td>1321</td>
<td>19625s</td>
<td>2901%</td>
</tr>
<tr>
<td>Partial instrumentation</td>
<td>188</td>
<td>719s</td>
<td>11%</td>
</tr>
<tr>
<td>instrumentation (auto)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>callpath depth 100, -optHeaderInst and selective instrumentation (auto)</td>
<td>8535</td>
<td>893s</td>
<td>36.5%</td>
</tr>
</tbody>
</table>

Each of these instrumentation approaches ignores any functions in libraries available only in binary form.

Figure source: [http://www.nic.uoregon.edu/tau-wiki/MADNESS](http://www.nic.uoregon.edu/tau-wiki/MADNESS)

*full instrumentation slows execution by 30x!*
Novel Aspects of Our Approach

• Unwind fully-optimized and even stripped code
  — use on-the-fly binary analysis to support unwinding

• Cope with dynamically-loaded shared libraries on Linux
  — note as new code becomes available in address space
  — problematic for instrumentation-based tools, unless using Dyninst or Pin

• Integrate static & dynamic context information in presentation
  — dynamic call chains including procedures, inlined functions, loops, and statements
Measurement Effectiveness

• **Accurate**
  
  — PFLOTRAN on Cray XT @ 8192 cores
    – 148 unwind failures out of 289M unwinds
    – 5e-5% errors
  
  — Flash on Blue Gene/P @ 8192 cores
    – 212K unwind failures out of 1.1B unwinds
    – 2e-2% errors
  
  — SPEC2006 benchmark test suite (sequential codes)
    – fully-optimized executables: Intel, PGI, and Pathscale compilers
    – 292 unwind failures out of 18M unwinds (Intel Harpertown)
    – 1e-3% error

• **Low overhead**
  
  — e.g. PFLOTRAN scaling study on Cray XT @ 512 cores
    – measured cycles, L2 miss, FLOPs, & TLB @ 1.5% overhead
  
  — suitable for use on production runs
Outline

• Overview of Rice’s HPCToolkit
• Accurate measurement
• **Effective performance analysis**
  • Pinpointing scalability bottlenecks
    — scalability bottlenecks on large-scale parallel systems
    — scaling on multicore processors
• Understanding temporal behavior
• Assessing process variability
• Research prototypes
  — blame shifting
  — attributing memory hierarchy costs to data
• Summary and conclusions
Recovering Program Structure

• Analyze an application binary
  — identify object code procedures and loops
    – decode machine instructions
    – construct control flow graph from branches
    – identify natural loop nests using interval analysis
  — map object code procedures/loops to source code
    – leverage line map + debugging information
    – discover inlined code
    – account for many loop and procedure transformations

Unique benefit of our binary analysis

• Bridges the gap between
  — lightweight measurement of fully optimized binaries
  — desire to correlate low-level metrics to source level abstractions
Analyzing Results with hpcviewer

- costs for
  - inlined procedures
  - loops
  - function calls in full context

- source pane
- view control
- metric display
- navigation pane
Principal Views

• Calling context tree view - “top-down” (down the call chain)
  — associate metrics with each dynamic calling context
  — high-level, hierarchical view of distribution of costs
  — example: quantify initialization, solve, post-processing

• Caller’s view - “bottom-up” (up the call chain)
  — apportion a procedure’s metrics to its dynamic calling contexts
  — understand costs of a procedure called in many places
  — example: see where PGAS put traffic is originating

• Flat view - ignores the calling context of each sample point
  — aggregate all metrics for a procedure, from any context
  — attribute costs to loop nests and lines within a procedure
  — example: assess the overall memory hierarchy performance within a critical procedure
Sampling Call Chains with Recursion

• Problem: some recursive algorithms, e.g., quicksort have many long and unique call chains
  — each sample can expose a unique call chain
  — space overhead can be significant for recursive computations that have many unique call chains, e.g. broad and deep trees
    – for parallel programs, the total space overhead can be especially problematic when thread-level views are merged

• Approach
  — collapse recursive chains to save space
  — preserve one level of recursion so high-level properties of the recursive solution remain available
Example: Recursive Fibonacci

- Compact representation
- Summarizes costs for each subtree in the recursion
- $T_{\text{fib}(n-1)} / T_{\text{fib}(n-2)} = 1.619$
  (within .1% of the golden ratio)
Properties of our Approach to Recursion

- Accurate costs attributed to source code lines within the recursive routine at the top level
- Total costs attributed to the subtree below each recursive call
  - e.g., can see that the costs of recursive calls to fib are related by the golden ration
- Total costs attributed to the callsite of any function called from the recursive routine
  - can see the total cost of swap64 called from quicksort in MPBS
- Total costs attributed to each line of the recursive routine within each recursive subtree
  - in quicksort, can inspect the costs of loops comparing with the pivot and splitting into left/right groups
Configure HPCToolkit on Ranger/Lonestar

- Prepend the following to your path
  - /work/01055/johnmc/pkgs/hpctoolkit-nopapi/bin
  - /work/01055/johnmc/pkgs/jdk/bin

- Refresh your paths

- Check that everything is working
  - try starting up hpcviewer
    - you should get a splash screen and a file chooser
Exercise: Using hpcviewer

- cp -r /home/johnmc/work/examples/fib $HOME/fib
- cd $HOME/fib
- make run

- Use hpcviewer to look at
  - (1) hpctoolkit-fib-icc-database
  - (2) hpctoolkit-fib-gcc-database

- FAQ
  - why are there two calls to fib from main in database (1)
    - one instance of fib is inlined in main
  - why is there only one call and a loop in database (2)
    - gcc performs tail call optimization, replacing the last call with a branch back to the routine entry
  - why are the call chains only two levels deep
    - hpctoolkit compresses recursion to two levels to avoid excess space
Outline

• Overview of Rice’s HPCToolkit
• Accurate measurement
• Effective performance analysis
  • Pinpointing scalability bottlenecks
    — scalability bottlenecks on large-scale parallel systems
    — scaling on multicore processors
• Understanding temporal behavior
• Assessing process variability
• Understanding threading and memory hierarchy
  — blame shifting
  — attributing memory hierarchy costs to data
• Summary and conclusions
The Problem of Scaling

Note: higher is better
Goal: Automatic Scaling Analysis

- Pinpoint scalability bottlenecks
- Guide user to problems
- Quantify the magnitude of each problem
- Diagnose the nature of the problem
Challenges for Pinpointing Scalability Bottlenecks

- Parallel applications
  - modern software uses layers of libraries
  - performance is often context dependent

- Monitoring
  - bottleneck nature: computation, data movement, synchronization?
  - 2 pragmatic constraints
    - acceptable data volume
    - low perturbation for use in production runs

Example climate code skeleton
Performance Analysis with Expectations

- You have performance expectations for your parallel code
  - strong scaling: linear speedup
  - weak scaling: constant execution time

- Put your expectations to work
  - measure performance under different conditions
    - e.g. different levels of parallelism or different inputs
  - express your expectations as an equation
  - compute the deviation from expectations for each calling context
    - for both inclusive and exclusive costs
  - correlate the metrics with the source code
  - explore the annotated call tree interactively
Pinpointing and Quantifying Scalability Bottlenecks

\[ P \times \left( \begin{array}{c} \text{coefficients for analysis} \\ \text{of strong scaling} \end{array} \right) - Q \times \left( \begin{array}{c} 600K \\ 400K \end{array} \right) = \left( \begin{array}{c} 200K \end{array} \right) \]
Parallel, adaptive-mesh refinement (AMR) code

- Designed for compressible reactive flows
- Can solve a broad range of (astro)physical problems
- Portable: runs on many massively-parallel systems
- Scales and performs well
- Fully modular and extensible: components can be combined to create many different applications

**Scalability Analysis Demo**

**Code:**

**Simulation:**

**Platform:**

**Experiment:**

**Scaling type:**

- University of Chicago FLASH
- white dwarf detonation
- Blue Gene/P
- 8192 vs. 256 processors
- weak

**Figures courtesy of FLASH Team, University of Chicago**
Scalability Analysis of Flash (Demo)
Improved Flash Scaling of AMR Setup

Graph courtesy of Anshu Dubey, U Chicago
Scaling on Multicore Processors

• Compare performance
  — single vs. multiple processes on a multicore system

• Strategy
  — differential performance analysis
    – subtract the calling context trees as before, unit coefficient for each
S3D: Multicore Losses at the Procedure Level

**Execution time increases 1.65x in subroutine rhsf**

**subroutine rhsf accounts for 13.0% of the multicore scaling loss in the execution**
S3D: Multicore Losses at the Loop Level

Execution time increases 2.8x in the loop that scales worst

loop contributes 6.9% of the scaling loss for the whole execution
Outline

• Overview of Rice’s HPCToolkit
• Accurate measurement
• Effective performance analysis
• Pinpointing scalability bottlenecks
  — scalability bottlenecks on large-scale parallel systems
  — scaling on multicore processors
• Understanding temporal behavior
• Assessing process variability
• Understanding threading and memory hierarchy
  — blame shifting
  — attributing memory hierarchy costs to data
• Summary and conclusions
Profiling compresses out the temporal dimension—temporal patterns, e.g. serialization, are invisible in profiles.

What can we do? Trace call path samples—sketch:
- N times per second, take a call path sample of each thread
- organize the samples for each thread along a time line
- view how the execution evolves left to right
- what do we view?
  assign each procedure a color; view a depth slice of an execution
Exposes Temporal Call Path Patterns

PFLOTRAN, 8184 processes, Cray XT5

Process-time view at selected depth

Depth-time view for selected rank
Presenting Large Traces on Small Displays

• How to render an arbitrary portion of an arbitrarily large trace?
  — we have a display window of dimensions $h \times w$
  — typically many more processes (or threads) than $h$
  — typically many more samples (trace records) than $w$

• Solution: sample the samples!

Trace with $n$ processes

samples (of samples)

each sample defines a pixel

process

process

$w$

$h$
Exercise: Using hpctraceviewer

- cd /home/johnmc/work/data
- hpctraceviewer hpctoolkit-chombo-database-1024pe
- Move down the call stack by selecting a routine at a deeper level in the call stack pane
- Move down several levels one at a time and see how details of the execution are revealed
- Zoom in by sweeping out a box
- Find a reduction instance
  - reductions generally have a jagged left edge and a sharp right edge in the space time diagram
- Questions
  - is there a serialization bottleneck?
Exercise: Using hpcviewer on Chombo

- `cd /home/johnmc/work/data`
- `hpcviewer hpctoolkit-chombo-database-1024pe`
- **Explore the Calling Context View**
  - use the flame button to see a top-down view of where the code spends its time
- **Use the Caller’s View**
  - sort by exclusive time
  - see where the code spends its time
  - use the flame button to expose callers to see how the execution got there
- **Use the Flat View**
  - see how much time was spent in each routine regardless of context
  - what is the most costly routine in the execution?
  - what is the most costly loop?
Outline

• Overview of Rice’s HPCToolkit
• Accurate measurement
• Effective performance analysis
• Pinpointing scalability bottlenecks
  — scalability bottlenecks on large-scale parallel systems
  — scaling on multicore processors
• Understanding temporal behavior
• Assessing process variability
• Understanding threading and memory hierarchy
  — blame shifting
  — attributing memory hierarchy costs to data
• Summary and conclusions
Example: Massive Parallel Bucket Sort (MPBS)

Program execution consists of two phases

- Produces a large number of files
  - each file has a fixed numbered sequence of buckets
  - each bucket has a fixed number of records
  - each record is a 4, 8, or 16-byte integer
  - each file produced by sequentially filling each bucket with integer records
    - most significant bits set to bucket number
    - file complete when all buckets filled and file written to disk

- Performs a two-stage sort on the contents of all files
  - records are sorted for a given bucket number across all of the generated files
  - then written to a single file
  - this is repeated for each bucket
  - this yields a single sorted file as a result

Sample execution: radix sort, 960 cores, 512MB/core
MPBS @ 960 cores, radix sort

Two views of load imbalance since not on a $2^k$ cores
Exercise: Using hpcviewer on Chombo

- cd /home/johnmc/work/data
- hpcviewer hpctoolkit-chombo-database-1024pe
- Explore the Calling Context View
  - use the flame button to see a top-down view of where the code spends its time
- Use the Caller’s View
  - sort by exclusive time
  - see where the code spends its time
  - use the flame button to expose callers to see how the execution got there
- Use the Flat View
  - see how much time was spent in each routine regardless of context
  - what is the most costly routine in the execution?
  - what is the most costly loop?
Outline

• Overview of Rice’s HPCToolkit
• Accurate measurement
• Effective performance analysis
• Pinpointing scalability bottlenecks
  — scalability bottlenecks on large-scale parallel systems
  — scaling on multicore processors
• Understanding temporal behavior
• Assessing process variability
• Understanding threading and memory hierarchy
  — blame shifting
  — attributing memory hierarchy costs to data
• Summary and conclusions
Challenges for Highly Threaded Architectures

- Problem: MPI everywhere is no longer enough
  - must use threading to fully exploit the power of BG/Q’s A2
    - 16 compute cores; 4 hardware thread contexts per core
  - expected threading approach: OpenMP

- Performance concern: underutilizing threads
  - execution outside OpenMP parallel regions
  - inefficient parallel regions
    - load imbalance
    - serialization
    - overhead significant w.r.t. computation

- Developers need guidance to tune threading
Understanding Performance of Threaded Programs

- Symptoms are often separated from the cause
  - threads idling and waiting for work are a symptom of a problem

- Idea: “blame shifting” from symptoms to causes
  - performance analysis of multithreaded Cilk programs
    - spinning and waiting to steal work is a symptom of insufficient parallelism; shift blame to active workers
      Tallent & Mellor-Crummey [PPoPP 2009]
  - applied to locking in threaded programs
    - spinning for a lock is a symptom; shift blame to lock holders
      Tallent & Mellor-Crummey [PPoPP 2010]

- Implementation sketch
  - special measurement strategy for proper attribution of blame
  - collect new metrics: idleness, work, overhead
Root Cause Analysis of Threading Performance Losses in MPI+OpenMP

• Approach
  — lightweight instrumentation of OpenMP runtime system
    – callbacks when threads begin work or become idle maintain quantitative information about instantaneous idleness of threads
  — efficient sampling-based measurement
    – assigns quantitative credit for work and blame for idleness to application loops, procedures, and call chains
  — post-mortem analysis techniques
    – compute insightful assessments of performance losses and quantify the potential benefits of code improvements

• Prototype implementation in Rice’s HPCToolkit
  — code-centric views precisely attribute time, work, idleness
  — space-time diagrams illustrate executions over time

• Used HPCToolkit’s guidance to identify and fix load imbalance in LLNL’s AMG2006 solver which employs MPI+OpenMP
OpenMP loop in `hypre_BoomerAMGRelax` using static scheduling has load imbalance; threads idle for a significant fraction of their time.
Code-centric view: `hypre_BoomerAMGRelax`

Note: The highlighted OpenMP loop in `hypre_BoomerAMGRelax` accounts for only 4.6% of the execution time for this benchmark run. In real runs, solves using this loop are a dominant cost across all instances of this OpenMP loop in `hypre_BoomerAMGRelax`. 19.7% of time in this loop is spent idle idle w.r.t. total effort in this loop.
Serial Code in AMG2006 8 PE, 8 Threads

7 worker threads are idle in each process while its main MPI thread is working.
Root Cause Analysis for CPU+GPU

• Problem: code not running well on CPU+GPU

• What is are the causes?
  — CPU not offloading work to GPU?
  — inadequate CPU/GPU overlap?
  — inefficient GPU code?

• Apply “blame shifting” approach to hybrid architectures
  — when CPU idle: blame GPU code executing
  — when GPU idle: blame CPU code executing
Heterogeneous Code-Centric View - I

Blaming GPU for CPU idleness
Heterogeneous Code-Centric View - II

Blaming CPU for GPU idleness
Heterogeneous Code-Centric View - III

Crediting CPU-GPU overlap
Space-Time View of Heterogeneous Performance

- Explore space-time view of GPU & CPU activity together
  - CPU: sampled activity trace; GPU: event-based trace
  - GPU activity identified by calling contexts of kernels
- Scale this heterogeneous view per PE
Outline

• Overview of Rice’s HPCToolkit
• Accurate measurement
• Effective performance analysis
• Pinpointing scalability bottlenecks
  — scalability bottlenecks on large-scale parallel systems
  — scaling on multicore processors
• Understanding temporal behavior
• Assessing process variability
• Understanding threading and memory hierarchy
  — blame shifting
  — attributing memory hierarchy costs to data
• Summary and conclusions
Data Centric Analysis

- Goal: associate memory hierarchy performance losses with data

- Approach
  - intercept allocations to associate with their data ranges
  - associate latency with data using “instruction-based sampling” on AMD Opteron CPUs
    - identify instances of loads and store instructions
    - identify the data structure an access touches based on L/S address
    - measure the total latency associated with each L/S
  - present quantitative results using hpcviewer
Data Centric Analysis of S3D

41.2% of memory hierarchy latency related to yspecies array

yspecies latency for this loop is 14.5% of total latency in program
Outline

• Overview of Rice’s HPCToolkit
• Accurate measurement
• Effective performance analysis
• Pinpointing scalability bottlenecks
  — scalability bottlenecks on large-scale parallel systems
  — scaling on multicore processors
• Understanding temporal behavior
• Assessing process variability
• Understanding threading and memory hierarchy
  — blame shifting
  — attributing memory hierarchy costs to data

• Summary and conclusions
Summary

• Sampling provides low overhead measurement

• Call path profiling + binary analysis + blame shifting = insight
  — scalability bottlenecks
  — where insufficient parallelism lurks
  — sources of lock contention
  — load imbalance
  — temporal dynamics
  — problematic data structures

• Other capabilities
  — attribute memory leaks back to their full calling context
Status

- Operational today on
  - 64- and 32-bit x86 systems running Linux (including Cray XT/E/K)
  - IBM Blue Gene/P
  - IBM Power7 systems running Linux

- Emerging capabilities
  - IBM Blue Gene/Q
  - Idleness analysis of OpenMP programs
  - NVIDIA GPU
    - measurement and reporting using GPU hardware counters
    - data centric analysis

- Available as open source software at hpctoolkit.org
HPCToolkit Capabilities at a Glance

Attribute Costs to Code

Analyze Behavior over Time

Pinpoint & Quantify Scaling Bottlenecks

Shift Blame from Symptoms to Causes

Assess Imbalance and Variability

Associate Costs with Data

hpctoolkit.org
Demos

- **PFlotran - groundwater flow code**
  - 8K cores, traces, profile

- **CESM (community earth system model)**
  - coupled code, 312 cores, traces, profile

- **FLASH (structured adaptive mesh refinement)**
  - 256 cores, traces, profile
  - 256 cores vs. 8K cores, scalability analysis profile

- **MPBS**
  - 1K cores, mpi2, radix sort, quick sort, HW counters, wall clock time, profiles, traces
  - traces with load imbalance: 960 core radix and quick sort
  - pgas communication: 1K cores, UPC, radix sort
  - 16K cores

- **OMEN**: 11K cores, load imbalance

- **GTC** - gyrokinetic toroidal code, 192 cores MPI+OpenMP

- **S3D** - data centric analysis
Trying out HPCToolkit’s User Interfaces

• Recommendation
  — try our interfaces it before investing time with your application
  — see if you like what we have to offer

• Example database for hpcviewer and hpctraceviewer
  — Chombo AMR framework on 1024 Cray XE6 cores
    – available from http://hpctoolkit.org/examples.html
HPCToolkit Documentation

http://hpctoolkit.org/documentation.html

• Comprehensive user manual:
  
  
  — Quick start guide
    – essential overview that almost fits on one page
  
  — Using HPCToolkit with statically linked programs
    – a guide for using hpctoolkit on BG/P and Cray XT
  
  — The hpcviewer user interface
  
  — Effective strategies for analyzing program performance with HPCToolkit
    – analyzing scalability, waste, multicore performance ...
  
  — HPCToolkit and MPI
  
  — HPCToolkit Troubleshooting
    – why don’t I have any source code in the viewer?
    – hpcviewer isn’t working well over the network ... what can I do?

• Installation guide
Using HPCToolkit

- Add java to your path
- Add hpctoolkit’s bin directory to your path
- Perhaps adjust your compiler flags for your application
  - sadly, most compilers throw away the line map unless -g is on the command line. add -g flag after any optimization flags if using anything but the Cray compilers/ Cray compilers provide attribution to source without -g.
- Add hpclink as a prefix to your Makefile’s link line
  - e.g. hpclink mpixlf -o myapp foo.o ... lib.a -lm ...
- Decide what hardware counters to monitor (WALLCLOCK is good to start)
  - dynamically-linked executables (e.g., Linux)
    - use hpcrun -L to learn about counters available for profiling
    - use papi_avail
      - you can sample any event listed as “profilable”
  - statically-linked executables (e.g., Cray, Blue Gene)
    - use hpclink to link your executable
    - launch executable with environment var HPCRUN_EVENT_LIST=LIST (BG/P hardware counters supported)
Using Profiling and Tracing Together

• When tracing, good to have an event that represents a measure of time
  — e.g., WALLCLOCK or PAPI_TOT_CYC

• Turn on tracing while sampling using one of the above events
  — Linux: use hpcrun
    hpcrun -e PAPI_TOT_CYC@3000000 -t your_app
  — Blue Gene/P at ANL: pass environment settings to cqsub
    cqsub -p YourAllocation -q prod-devel -t 30 -n 2048 -c 8192 \ 
    --mode vn --env HPCRUN_EVENT_LIST=WALLCLOCK@1000 \ 
    --env HPCRUN_TRACE=1 your_app
  — Cray XT/E/K: set environment variable in your launch script
    setenv HPCRUN_EVENT_LIST “PAPI_TOT_CYC@3000000”
    setenv HPCRUN_TRACE 1
    aprun your_app
Monitoring Using Hardware Counters

- **Linux:** use *hpcrun*

  ```
  hpcrun -e PAPI_TOT_CYC@3000000 -e PAPI_L2_MISS@400000 \ 
  -e PAPI_TLB_MISS@400000 -e PAPI_FP_OPS@400000 \ 
  your_app
  ```

- **Blue Gene/P at ANL:** pass environment settings to *cqsub*

  ```
  cqsub -p YourAllocation  -q prod-devel -t 30  -n 2048  -c 8192 \ 
  --mode vn  --env HPCRUN_EVENT_LIST=WALLCLOCK@1000 \ 
  your_app
  ```

- **Cray XT/E/K:** set environment variable in your launch script

  ```
  setenv HPCRUN_EVENT_LIST “PAPI_TOT_CYC@3000000 \ 
  PAPI_L2_MISS@400000 PAPI_TLB_MISS@400000 PAPI_FP_OPS@400000”
  aprun your_app
  ```
Analysis and Visualization

- Use hpcstruct to reconstruct program structure
  - e.g. hpcstruct your_app
    - creates your_app.hpcstruct

- Use hpcprof to correlate measurements to source code
  - run hpcprof on the front-end node
  - run hpcprof-mpi on the compute nodes to analyze data in parallel

- Use hpcviewer to open resulting database

- Use hpctraceviewer to explore traces (collected with -t option)
Memory Leak Detection with HPCToolkit

• Statically linked code
  — hpclink --memleak -o your_app foo.o ... lib.a -lm ...
  — at launch time
    – setenv HPCTOOLKIT_EVENT_LIST=MEMLEAK
    – your_app

• Dynamically linked code
  — hpcrun -e MEMLEAK your_app