HPCToolkit: Sampling-based Performance Tools for Leadership Computing

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

http://hpctoolkit.org

CScADS Workshop on Leadership Computing    July 19-22, 2010
Acknowledgments

• Staff
  — Laksono Adhianto, Mike Fagan, Mark Krentel, Nathan Tallent

• Grad Student
  — Xu Liu

• High School Summer Interns
  — Sinchan Banerjee, Michael Franco, Bowden Kelly, Chas Jhin, Reed Landrum, Gaurav Sanghani

• Alumni
  — Gabriel Marin (ORNL), Robert Fowler (RENCI)

• SciDAC project support
  — Center for Scalable Application Development Software
    – Cooperative agreement number DE-FC02-07ER25800
  — Performance Engineering Research Institute
    – Cooperative agreement number DE-FC02-06ER25762
Challenges

• Gap between typical and peak performance is huge

• Complex architectures are harder to program effectively
  — processors that are pipelined, out of order, superscalar
  — multi-level memory hierarchy
  — multi-level parallelism: multi-core, SIMD instructions

• Complex applications pose challenges
  — for measurement and analysis
  — for understanding and tuning

• Leadership computing platforms: additional complexity
  — more than just computation: communication, I/O
  — immense scale
  — unique microkernel-based operating systems
Performance Analysis Principles

• Without accurate measurement, analysis is irrelevant
  — avoid systematic measurement error
    – instrumentation-based measurement is often problematic
  — measure actual system, not a mock up
    – fully optimized production code on the platform of interest

• Without effective analysis, measurement is irrelevant
  — pinpoint and explain problems in terms of source code
    – binary-level measurements, source-level insight
  — compute insightful metrics
    – “unused bandwidth” or “unused flops” rather than “cycles”

• Without scalability, a tool is irrelevant
  — large codes
  — large-scale node parallelism + multithreading
Performance Analysis Goals

• 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 (e.g. Linux clusters) vs. static linking (Cray XT, BG/P)
    – SPMD parallel codes with threaded node programs
    – batch jobs

• Effective performance analysis
  — insightful analysis that pinpoints and explains problems
    – correlate measurements with code (yield actionable results)
    – intuitive enough for scientists and engineers
    – detailed enough for compiler writers

• Scalable to leadership computing systems
HPCToolkit Design Principles

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

• Sampling-based measurement (avoid instrumentation)
  — minimize systematic error and avoid blind spots
  — enable data collection for large-scale parallelism

• Collect and correlate multiple derived performance metrics
  — diagnosis requires more than one species of metric
  — derived metrics: “unused bandwidth” rather than “cycles”

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

• Support top-down performance analysis
  — intuitive enough for scientists and engineers to use
  — detailed enough to meet the needs of compiler writers
Outline

- Overview of Rice’s HPCToolkit
  - Accurate measurement
  - Useful source-level feedback
  - Effective performance analysis
    - derived metrics for understanding performance
    - pinpointing scalability bottlenecks [SC09]
    - analyzing lock contention in threaded codes [PPoPP10]
    - pinpointing load imbalance [SC10]
    - understanding temporal dynamics of parallel codes
- Using HPCToolkit
- Coming attractions
HPCToolkit Workflow
For dynamically-linked executables on stock Linux
   — compile and link as you usually do: nothing special needed

For statically-linked executables (e.g. for BG/P, Cray XT)
   — 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
• 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
• Combine multiple profiles  
  — multiple threads; multiple processes; multiple executions  
• Correlate metrics to static & dynamic program structure
**Visualization**

- 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
Outline

• Overview of Rice’s HPCToolkit
  • Accurate measurement
  • Useful source-level feedback
  • Effective performance analysis
    — derived metrics for understanding performance
    — pinpointing scalability bottlenecks [SC09]
    — analyzing lock contention in threaded codes [PPoPP10]
    — pinpointing load imbalance [SC10]
    — understanding temporal dynamics of parallel codes

• Using HPCToolkit
• Coming attractions
Measurement

- Compile & Link
  - App. Source
  - Optimized Binary

- Profile Execution
  - Call Stack Profile
  - Binary Analysis
    - Program Structure

- Interpret Profile
  - Correlate w/ Source
  - Database
  - Visualization
    - [hpcviewer]

- [hpcrun]
- [hpcstruct]
- [hpcprof]
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
Unwinding Optimized Code

- Optimized code presents challenges for unwinding
  - optimized code often lacks frame pointers
  - no compiler information about epilogues
  - routines may have multiple epilogues, multiple frame sizes
  - code may be partially stripped: no info about function bounds

- What we need
  - where is the return address of the current frame?
    - a register, relative to SP, relative to BP
  - where is the FP for the caller’s frame?
    - a register, relative to SP, relative to BP

- Approach: use binary analysis to support unwinding
Dynamically Loaded Code (Linux)

New code may be loaded/unloaded at any time

- **When a new module is loaded**
  - note new code segment mappings
  - build table of new procedure bounds

- **When a module is unloaded**
  - mark end of profiler epoch: code addresses no longer apply
  - flush stale cached information
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

• Useful source-level feedback

• Effective performance analysis
  — derived metrics for understanding performance
  — pinpointing scalability bottlenecks [SC09]
  — analyzing lock contention in threaded codes [PPoPP10]
  — pinpointing load imbalance [SC10]
  — understanding temporal dynamics of parallel codes

• Using HPCToolkit
• Coming attractions
Useful Source-level Feedback

compile & link

app. source → optimized binary

profile execution [hpcrun] → call stack profile

binary analysis [hpcstruct] → program structure

interpret profile correlate w/ source [hpcprof]

database → visualization [hpcviewer]
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**

**Metric 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

• 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

• Flat view
  — “flatten” 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
Outline

• Overview of Rice’s HPCToolkit
• Accurate measurement
• Useful source-level feedback
• Effective performance analysis
  — derived metrics for understanding performance
  — pinpointing scalability bottlenecks [SC09]
  — analyzing lock contention in threaded codes [PPoPP10]
  — pinpointing load imbalance [SC10]
  — understanding temporal dynamics of parallel codes

• Using HPCToolkit
• Coming attractions
S3D Solver for Turbulent, Reacting Flows

Overall performance (15% of peak)
2.05 \times 10^{11} \text{ FLOPs} / 6.73 \times 10^{11} \text{ cycles} = 0.305 \text{ FLOPs/cycle}

Wasted Opportunity (Maximum FLOP rate * cycles - (actual FLOPs))

highlighted loop accounts for 11.4% of total program waste
Outline

• Overview of Rice’s HPCToolkit
• Accurate measurement
• Useful source-level feedback
• Effective performance analysis
  — derived metrics for understanding performance
  — pinpointing scalability bottlenecks [SC09]
  — analyzing lock contention in threaded codes [PPoPP10]
  — pinpointing load imbalance [SC10]
  — understanding temporal dynamics of parallel codes
• Using HPCToolkit
• Coming attractions
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: community earth system model
Performance Analysis with Expectations

• Users have performance expectations for parallel codes
  — strong scaling: linear speedup
  — weak scaling: constant execution time

• Putting 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
Analyzing Weak Scaling: 1K to 10K processors

Weak scaling
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: FLASH

- Code: University of Chicago FLASH
- Simulation: white dwarf detonation
- Platform: Blue Gene/P
- Experiment: 8192 vs. 256 processors
- Scaling type: weak

Figures courtesy of FLASH Team, University of 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 - DNS Solver

• Solves compressible reacting Navier-Stokes equations
• High fidelity numerical methods
  – 8th order finite-difference
  – 4th order explicit RK integrator
• Hierarchy of molecular transport models
• Detailed chemistry
• Multi-physics (sprays, radiation and soot)
  – from SciDAC-TSTC (Terascale Simulation of Turbulent Combustion)

Text and figures courtesy of Jacqueline H. Chen, SNL
Execution time increases 2.8x in the loop that scales worst.

Loop contributes a 6.9% scaling loss to whole execution.
Outline

• Overview of Rice’s HPCToolkit
• Accurate measurement
• Useful source-level feedback
• Effective performance analysis
  — derived metrics for understanding performance
  — pinpointing scalability bottlenecks [SC09]
  — analyzing lock contention in threaded codes [PPoPP10]
  — pinpointing load imbalance [SC10]
  — understanding temporal dynamics of parallel codes
• Using HPCToolkit
• Coming attractions
Lock contention => idleness
  – explicitly threaded programs (Pthreads, etc)
  – implicitly threaded programs (critical sections in OpenMP, ...)

Strategy: “blame-shifting” of contention from victim to perpetrator
  – use shared state (locks) to communicate blame

How it works
  – consider spin-waiting
  – sample a working thread:
    • charge to ‘work’ metric
  – sample an idle thread
    • accumulate in idleness counter associated with a lock (atomic add)
  – working thread releases a lock
    • atomically swap 0 with lock’s idleness counter
    • exactly represents contention while that thread held the lock
    • unwind the call stack to attribute lock contention to a calling context
Lock Contention in MADNESS

Quantum chemistry; MPI + pthreads
- 65M distinct locks
- max. of 340K live locks
- 30K lock acquisitions/sec/thread

1-5% overhead

16 cores; 1 thread/core (4 x Barcelona)

lock contention accounts for 23.5% of execution time.

Adding futures to shared global work queue.
Outline

• Overview of Rice’s HPCToolkit
• Accurate measurement
• Useful source-level feedback
• Effective performance analysis
  — derived metrics for understanding performance
  — pinpointing scalability bottlenecks [SC09]
  — analyzing lock contention in threaded codes [PPoPP10]
  — pinpointing load imbalance [SC10]
  — understanding temporal dynamics of parallel codes
• Using HPCToolkit
• Coming attractions
Identifying Load Imbalance Post Mortem

1. Identify exposed waiting: all imbalance is manifested in waiting

2. Identify balance points (procedures or loops that cannot contribute to imbalance)

3. Blame imbalance on the computation subtree in which it originates

4. Associate each (summary) node with thread-level metric values
Load Imbalance Analysis Example

**PFLOTRAN**: modeling multi-scale, multiphase, multi-component subsurface reactive flows

Example use: modeling sequestration of CO$_2$ in deep geologic formations, where resolving density-driven fingering patterns is necessary to accurately describe the rate of dissipation of the CO$_2$ plume.

Strong scaling study on Cray XT
PFLOTRAN

8K cores, Cray XT5

1. Drill down ‘hot path’ to loop (a balance point)

2. Notice top two call sites...

3. Plot the per-process values:

   Early finishers...

   ... become early arrivers at Allreduce
Outline

• Overview of Rice’s HPCToolkit
• Accurate measurement
• Useful source-level feedback
• Effective performance analysis
  — derived metrics for understanding performance
  — pinpointing scalability bottlenecks [SC09]
  — analyzing lock contention in threaded codes [PPoPP10]
  — pinpointing load imbalance [SC10]
  — understanding temporal dynamics of parallel codes
• Using HPCToolkit
• Coming attractions
Profiling compresses out the temporal dimension
  - that’s why serialization is 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 execution with a depth slice
Flash White Dwarf Collapse on 256 Cores

Full execution at call stack depth 2
Flash White Dwarf Collapse on 256 Cores

Full execution at call stack depth 5
Flash White Dwarf Collapse on 256 Cores

Execution detail at call stack depth 5
Outline

• Overview of Rice’s HPCToolkit
• Accurate measurement
• Useful source-level feedback
• Effective performance analysis
  — derived metrics for understanding performance
  — pinpointing scalability bottlenecks [SC09]
  — analyzing lock contention in threaded codes [PPoPP10]
  — pinpointing load imbalance [SC10]
  — understanding temporal dynamics of parallel codes

• Using HPCToolkit

• Coming attractions
Where to Find HPCToolkit

• DOE Systems
  — jaguar: /ccs/proj/hpctoolkit/pkgs/hpctoolkit
  — intrepid: /home/projects/hpctoolkit/pkgs/hpctoolkit
  — franklin: /project/projectdirs/hpctk/pkgs/hpctoolkit

• NSF Systems
  — ranger: /scratch/projects/hpctoolkit/pkgs/hpctoolkit

• For your local Linux systems, you can download and install it
  — documentation, build instructions, link to our svn repository
    – svn repository: https://outreach.scidac.gov/svn/hpctoolkit
  — we recommend downloading and building from svn
  — important notes:
    – obtaining information from hardware counters requires downloading and installing PAPI
    – installing PAPI
      on Linux 2.6.32 or better: built-in kernel support for counters
      earlier Linux needs a kernel patch (perfmon2 or perfctr)
Available Guides

http://hpctoolkit.org/documentation.html

- Using HPCToolkit with statically linked programs [pdf]
  — a guide for using hpctoolkit on BG/P and Cray XT

- Quick start guide [pdf]
  — essential overview that almost fits on one page

- The hpctoolkit user interface [pdf]

- Effective strategies for analyzing program performance with HPCToolkit [pdf]
  — analyzing scalability, waste, multicore performance ...

- HPCToolkit and MPI [pdf]

- HPCToolkit Troubleshooting [pdf]
  — why don’t I have any source code in the viewer?
  — hpctoolkit isn’t working well over the network ... what can I do?
Setup

• Add hpctoolkit’s bin directory to your path
  — see earlier slide for HPCToolkit’s HOME directory on your system

• Adjust your compiler flags (if you want full attribution to src)
  — add -g flag after any optimization flags

• 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
  — 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 XT, BG/P)
    – use hpclink to link your executable
    – launch executable with environment var HPCRUN_EVENT_LIST=LIST
      (currently BG/P hardware counters unsupported)
Launching your Job

• Modify your run script to enable monitoring
  — Cray XT: set environment variable in your PBS script
    – e.g. setenv HPCRUN_EVENT_LIST "PAPI_TOT_CYC@3000000
      PAPI_L2_DCM@400000 PAPI_TLB_DM@400000
      PAPI_FP_OPS@400000"
    – to collect a trace of WALLCLOCK samples
      setenv HPCRUN_OPT_TRACE=1
  — Blue Gene/P: pass environment settings to qsub
    – qsub -A YourAllocation -q prod -t 30 -n 2048 \
      --proccount 8192 --mode vn --env \n      HPCRUN_EVENT_LIST=WALLCLOCK@1000 flash3.hpc
    – to collect a trace of WALLCLOCK samples use
      HPCRUN_EVENT_LIST=WALLCLOCK:HPCRUN_OPT_TRACE=1
Binary Analysis and Data Assessment

• Use hpcstruct to reconstruct program structure
  — e.g. hpcstruct myapp
    – creates myapp.hpcstruct

• Use hpcsummary script to summarize measurement data
  — e.g. hpcsummary hpctoolkit-myapp-measurements-5912
Analyzing Data with hpcprof-mpi

- Analyze call graph profiles from all cores together
  - perform analysis in parallel for acceptable analysis time

- Purpose:
  - compute summary statistics across nodes
    - enables top-down investigation of node differences
  - provide access to thread-level data for detailed comparisons

- Mechanics
  - hpcprof-mpi is just an MPI program
    - launch it on an appropriate number of nodes to reduce analysis time
    - e.g. analysis of PFLOTRAN on Cray XT: 8K profiles, 48 nodes, 10 min
      - e.g. qsub -A YourAllocation -q prod-devel -t 20 -n 64
        hpcprof-mpi -S myapp.hpcstruct -I "path_to_src/*"
        hpctoolkit-myapp-measurements-5912
  - produces hpctoolkit-myapp-database-5912
Analyzing Data with hpcprof

• This runs on the head node; can’t analyze all performance data there for large parallel executions

• Use hpcprof to analyze one (or a few) measurement files
  — select one or a few files from your measurements to analyze
  — e.g. `hpcprof -S myapp.hpcstruct -I "path_to_src/*"`
  hpctoolkit-myapp-measurements-5912/
  myapp-0000-000-983409-764.hpcrun
  — produces hpctoolkit-myapp-database-5912
Using hpcviewer and hpctraceview

- **Notes**
  - if you collected traces or used hpcprof-mpi, your performance database will be large
    - best approach: analyze it on the leadership computing platform
  - you can tar up a database for analysis on your laptop
    - with patience: copy whole database to laptop
    - impatient way: tar up database without thread or trace data

- **Use hpcviewer to open a performance database**
  - if using hpcviewer on a the leadership computing platform, add recent Java implementation to your path (for hpcviewer)
    - Cray XT: module load java
    - Blue Gene/P: add /opt/soft/.../java/bin to your path
  - on a front-end node, run hpcviewer with the performance database as an argument
  - ALCF: can also run hpcviewer on gadzooks or eureka

- **Use hpctraceview to open call stack traces of core activity**
  - run hpctraceview and open performance database
**hpcviewer and hpctraceview tip**

- When running interactive viewers on leadership platforms
  - create a virtual desktop with vncserver
  - view the virtual desktop with vncviewer
  - run hpcviewer or hpctraceview inside your virtual desktop
A Note About hpcstruct and Fortran

• Fortran compilers emit machine code that have an unusual mapping back to source

• To compensate, hpcstruct needs a special option
  — --loop-fwd-subst=no
  — without this option, many nested loops will be missing in hpcstruct’s output and (as a result) hpcviewer

• Useful for IBM’s xlf, PGI’s pgf90 and others
Outline

• Overview of Rice’s HPCToolkit
• Accurate measurement
• Useful source-level feedback
• Effective performance analysis
  — derived metrics for understanding performance
  — pinpointing scalability bottlenecks [SC09]
  — analyzing lock contention in threaded codes [PPoPP10]
  — pinpointing load imbalance [SC10]
  — understanding temporal dynamics of parallel codes

• Using HPCToolkit

• Coming attractions
• Goal: associate memory hierarchy locality problems with particular data structures

• Approach
  – intercept memory allocations to associate data range with allocation
  – associate latency with data structures using “instruction based sampling” capability of 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 results in hpcviewer
Data Centric Analysis of S3D

41.2% of exposed latency related to yspecies array

yspecies latency associated with this loop is 14.5% of total latency in program
HPCToolkit Summary

- Obtain insight, accuracy & precision by combining call path profiling, binary analysis, and blame shifting
- Show surprisingly effective measurement and source-level attribution for fully optimized code (1-3% overhead)
  - statements in their full static and dynamic context
  - project low-level measurements to much higher levels
- Sampling-based measurements can deliver insight into a range of phenomena
  - scalability bottlenecks
  - sources of lock contention
  - load imbalance
  - temporal dynamics
  - problematic data structures
Some Challenges Ahead

- Data management for scalable measurement and analysis
- Moving from descriptive to prescriptive feedback
- Increasing importance of threading as core counts increase
- Heterogeneous architectures, e.g. GPU accelerators