HPCToolkit: Sampling-based Performance Tools for Leadership Computing

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

http://hpctoolkit.org
Acknowledgments

• Funding sources
  — Center for Scalable Application Development Software
    – Cooperative agreement number DE-FC02-07ER25800
  — Performance Engineering Research Institute
    – Cooperative agreement number DE-FC02-06ER25762

• Project Team
  — Research Staff
    – Laksono Adhianto, Mike Fagan, Mark Krentel
  — Students
    – Xu Liu, Milind Chabbi, Karthik Murthy
  — Collaborator
    – Nathan Tallent (PNNL)
  — Alumni
    – Gabriel Marin (ORNL), Robert Fowler (RENCI), Nathan Froyd (Mozilla)
  — Summer Interns:
    – Reed Landrum, Michael Franco, Sinchan Banerjee, Philip Taffet
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 to add new capabilities

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

• 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
• Assessing process variability
• Understanding temporal behavior
• Using HPCToolkit
• Ongoing R&D
source

code

compiled & link

source code

optimized binary

profile execution
[hpcrun]

binary analysis
[hpcstruct]

call path profile

program structure

profile execution

interpret profile

correlate w/ source
[hpcprof/hpcprof-mpi]

presentation

[hpctoolkit/ hpctraceviewer]

database

interpret profile

correlate w/ source
[hpcprof/hpcprof-mpi]
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 \texttt{hpclink} as prefix to your link line
    - uses “linker wrapping” to catch “control” operations
      process and thread creation, finalization, signals, ...

presentation
\texttt{[hpcviewer/hpctraceviewer]}
database
interpret profile correlate w/ source
\texttt{[hpcprof/hpcprof-mpi]}
**HPCToolkit Workflow**

- **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

**Presentation**
- `hpcviewer/ hpctraceviewer`

**Database**
- interpret profile correlate w/ source
  - `hpcprof/hpcprof-mpi`
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

**Diagram:**
- Source code → Optimized binary
- Optimized binary → Profile execution [hpcrun]
- Profile execution [hpcrun] → Call path profile
- Call path profile → Program structure
- Program structure → Binary analysis [hpcstruct]
- Binary analysis [hpcstruct] → Interpret profile correlate w/ source [hpcprof/hpcprof-mpi]
- Interpret profile correlate w/ source [hpcprof/hpcprof-mpi] → Database
- Database → Presentation [hpcviewer/hpctraceviewer]
- Presentation [hpcviewer/hpctraceviewer] → Compile & link
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
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
• Assessing process variability
• Understanding temporal behavior
• Using HPCToolkit
• Ongoing R&D
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
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

• 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
• Assessing process variability
• Understanding temporal behavior
• Using HPCToolkit
• Ongoing R&D
Effective Analysis

source code → optimized binary

compile & link

profile execution [hpcrun] → call path profile

binary analysis [hpcstruct] → program structure

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

presentation [hpcviewer/hpctraceviewer]

database
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 \texttt{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 - 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
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
• Assessing process variability
• Understanding temporal behavior
• Using HPCToolkit
• Ongoing R&D
The Problem of Scaling

![Graph showing efficiency of CPUs vs. Number of CPUs. The graph compares ideal efficiency (blue line) and actual efficiency (red line). Efficiency is noted as higher is better. There is a question mark indicating a point of interest or a question related to the graph.](image-url)
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

• Putting 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

\[ Q \times \left(\begin{array}{c}
600K
\end{array}\right) - P \times \left(\begin{array}{c}
400K
\end{array}\right) =
\]

coefficients for analysis of strong 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

**Code:** University of Chicago FLASH

**Simulation:** white dwarf detonation

**Platform:** Blue Gene/P

**Experiment:** 8192 vs. 256 processors

**Scaling type:** weak

**Simulation Examples:**
- Nova outbursts on white dwarfs
- Helium burning on neutron stars
- Laser-driven shock instabilities
- Orzag/Tang MHD vortex
- Rayleigh-Taylor instability

*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: Multicore Losses at the Loop Level

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
• Effective performance analysis
• Pinpointing scalability bottlenecks
  — scalability bottlenecks on large-scale parallel systems
  — scaling on multicore processors
• Assessing process variability
• Understanding temporal behavior
• Using HPCToolkit
• Ongoing R&D
“Right click” on a node in the CCT view to graph values across all threads

Values for all threads graphed for the selected context

NOTE: Must analyze measurement data with hpcprof-mpi to include thread-centric metrics in the performance database
Radix Sort on 960 Cores: Barrier Time

- sorted by rank
- sorted by value
- value histogram
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
• Assessing process variability
  • Understanding temporal behavior
• Using HPCToolkit
• Ongoing R&D
Understanding Temporal Behavior

- 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
Process-Time Views of PFLOTRAN

8184-core execution on Cray XT5. Trace view rendered using hpctraceviewer on a Mac Book Pro Laptop. Insets show zoomed view of marked region at different call stack depths.
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
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
• Assessing process variability
• Understanding temporal behavior
• Using HPCToolkit
• Ongoing R&D
Where to Find HPCToolkit

- **ALCF Systems**
  - intrepid: `/home/projects/hpctoolkit/ppc64/pkgs/hpctoolkit`
  - vesta: `/home/projects/hpctoolkit/pkgs/hpctoolkit`
  - eureka: `/home/projects/hpctoolkit/x86_64/pkgs/hpctoolkit`

- **OLCF (Interlagos)**
  - `/ccs/proj/hpctoolkit/pkgs/hpctoolkit-interlagos`
  - `/ccs/proj/hpctoolkit/pkgs/hpcviewer`

- **NERSC (Hopper)**
  - `/project/projectdirs/hpctk/hpctoolkit-hopper`
  - `/project/projectdirs/hpctk/hpcviewer`

- **For your local Linux systems, you can download and install it**
  - documentation, build instructions, and software
    - see http://hpctoolkit.org for instructions
  - we recommend downloading and building from svn
  - important notes:
    - using hardware counters requires downloading and installing PAPI
      - on Linux 2.6.32 or better: built-in kernel support for counters
      - earlier Linux needs a kernel patch (perfmon2 or perfctr)
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 and hpctraceviewer user interfaces
  — 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 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
  — statically-linked executables (e.g., Cray XT, BG/P)
    – use hpclink to link your executable
    – launch executable with environment var HPCRUN_EVENT_LIST=LIST
      (BG/P hardware counters supported)
  — 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”
Collecting Performance Data

• Collecting traces
  — dynamically-linked: hpcrun -t ...
  — statically-linked: set environment variable HPCRUN_TRACE=1

• Launching your job using hpctoolkit
  — Blue Gene
    – qsub -q prod-devel -t 10 -n 2048 -c 8192 \ 
      --env OMP_NUM_THREADS=2: \ 
      HPCRUN_EVENT_LIST=WALLCLOCK@5000: \ 
      HPCRUN_TRACE=1 your_app
  — Cray (with WALLCLOCK)
    setenv HPCRUN_EVENT_LIST “WALLCLOCK@5000”
    setenv HPCRUN_TRACE 1
    aprun your_app
  — Cray (with hardware performance counters)
    – setenv HPCRUN_EVENT_LIST “PAPI_TOT_CYC@3000000 \ 
      PAPI_L2_MISS@400000 PAPI_TLB_MISS@400000 PAPI_FP_OPS@400000”
    setenv HPCRUN_TRACE 1
    aprun your_app
Digesting your Performance Data

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

- **Correlate measurements to source code with hpcprof and hpcprof-mpi**
  - run `hpcprof` on the front-end node to analyze a few processes
    - no per-thread profiles
  - run `hpcprof-mpi` on the compute nodes to analyze data in parallel
    - includes per-thread profiles to support thread-centric graphical view

- **Digesting performance data in parallel with hpcprof-mpi**
  - `run_cmd`
    - `/path/to/hpcprof-mpi`
    - `-S your_app.hpcstruct`
    - `-l /path/to/your_app/src/"*"`
    - `hpctoolkit-your_app-measurements.jobid`
  - `runcmd`
    - Cray: `aprun`
    - Blue Gene: `qsub -q prod-devel -t 20 -n 32 -m co`
Analysis and Visualization

• Use hpcviewer to open resulting database
  — warning: first time you graph any data, it will pause to combine info from all threads into one file

• Use hpctraceviewer to explore traces
  — warning: first time you open a trace database, the viewer will pause to combine info from all threads into one file

• Try our our user interfaces before collecting your own data
  — example performance data for Chombo on hpctoolkit.org
A Special Note About hpccstruct and xlf

• IBM’s xlf compiler emits machine code for Fortran that have an unusual mapping back to source

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

• Why?
  — get meaningful results when measuring a shorter execution than would really be representative.
  — only want to measure solver without measuring initialization.

• How
  — Environment variable
    – HPCTOOLKIT_DELAY_SAMPLING=1
  — API
    – hpctoolkit_sampling_start()
    – hpctoolkit_sampling_stop()
  — Include file
    – -I /home/projects/hpctoolkit/ppc64/pkgs/hpctoolkit/include
    – #include <hpctoolkit.h>
  — Always against API library
    – -L /home/projects/hpctoolkit/ppc64/pkgs/hpctoolkit/lib/hpctoolkit /Lhpctoolkit
  — API is a no-op unless used with hpclink or hpcrun
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
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
• Assessing process variability
• Understanding temporal behavior
• Using HPCToolkit
• Ongoing R&D
Ongoing R&D

• Available in prototype form
  — memory leak detection
  — performance analysis of multithreaded code
    – pinpoint & quantify insufficient parallelism and parallel overhead
    – pinpoint & quantify idleness due to serialization at locks

• Emerging capabilities
  — data-centric profiling
  — GPU support
  — enhanced analysis of OpenMP and multithreading

• Future work
  — improving measurement scalability by using parallel file I/O
Ask Me About

- Filtering traces
- Derived metrics
- Profiling OpenMP
- Profiling hybrid CPU+GPU code
- Data centric performance analysis
- Profiling programs with recursion
- Scalable trace server