Gaining Insight into Parallel Program Performance using HPCToolkit

John Mellor-Crummey
Department of Computer Science
Rice University

http://hpctoolkit.org
Challenges for Computational Scientists

• Rapidly evolving platforms and applications
  — architecture
    – rapidly changing multicore microprocessor designs
    – increasing architectural diversity
      CPU, GPU, APU, manycore (e.g., Xeon Phi)
    – increasing scale of parallel systems
  — applications
    – augment computational capabilities

• Computational scientist needs
  — adapt to changes in emerging architectures
    – adding threading and/or offloading to accelerators
  — improve scalability within and across nodes
  — assess weaknesses in algorithms and their implementations

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

• Complex node architectures are hard to use efficiently
  — multi-level parallelism: multiple cores, ILP, SIMD, accelerators
  — multi-level memory hierarchy
  — result: gap between typical and peak performance is huge

• Complex applications present challenges
  — measurement and analysis
  — understanding behaviors and tuning performance

• Supercomputer platforms compound the complexity
  — unique hardware & microkernel-based operating systems
  — multifaceted performance concerns
    – computation
    – data movement
    – communication
    – I/O
What Users Want

• Easy-to-use multi-platform, programming model independent tools

• Accurate measurement of complex parallel codes
  — large, multi-lingual programs
  — (heterogeneous) parallelism within and across nodes
  — optimized code: loop optimization, templates, inlining
  — binary-only libraries, sometimes partially stripped
  — complex execution environments
    – dynamic binaries on clusters; static binaries on supercomputers
    – 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
Rice University’s HPCToolkit

• Employs binary-level measurement and analysis
  — observe executions of optimized code
  — support multi-lingual codes with external binary-only libraries

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

• Collects and correlates multiple derived performance metrics
  — diagnosis typically requires more than one species of metric

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

• Supports top-down performance analysis
  — natural approach that minimizes burden on developers
Outline

• Overview of Rice’s HPCToolkit

• Pinpointing scalability bottlenecks
  — scalability bottlenecks on large-scale parallel systems
  — scaling on multicore processors

• Understanding temporal behavior

• Assessing process variability

• Understanding threading, GPU, and memory hierarchy
  — blame shifting
  — attributing memory hierarchy costs to data

• Summary and challenges ahead
HPCToolkit Workflow

Compile & Link

Source Code → Optimized Binary

Profile Execution [hpcrun]

Call Path Profile

Profile Analysis [hpcstruct]

Program Structure

Binary Analysis

Presentation

Presentation [hpcviewer/ hpctraceviewer]

Database

Interpret Profile
Correlate w/ Source [hpcprof/hpcprof-mpi]
HPCToolkit Workflow

• For dynamically-linked executables, e.g., Linux
  — compile and link as you usually do
• For statically-linked executables, e.g., Blue Gene/Q
  — add monitoring by using hpclink as prefix to your link line
HPCToolkit Workflow

- Measure execution unobtrusively
  - launch optimized application binaries
    - dynamically-linked applications: launch with `hpcrun`
      e.g., `mpirun -np 8192 hpcrun -t -e WALLCLOCK@5000 flash3 ...`
    - statically-linked applications: control with environment variables
  - collect statistical call path profiles of events of interest
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
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

presentation
[hpcviewer/hpctraceviewer]

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

database

program structure

call path profile

binary analysis
[hpcstruct]

profile execution
[hpcrun]

optimized binary

source code

compile & link
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

```
source code → optimized binary → profile execution [hpcrun] → call path profile
                                
optimized binary → binary analysis [hpcstruct] → program structure

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

presentation [hpcviewer/hpctraceviewer]
```

database
Analyzing Chombo@1024 cores with hpcviewer

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

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

Note: higher is better
Wanted: Scalability Analysis

- Isolate scalability bottlenecks
- Guide user to problems
- Quantify the magnitude of each 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 and/or different problem size
  — 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

\[
Q \times (600K) - P \times (400K) = P \times (200K)
\]

coefficients for analysis of strong scaling
Parallel, adaptive-mesh refinement (AMR) code

- Block structured AMR; a block is the unit of computation
- 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: FLASH3

**Code:**
- University of Chicago FLASH3

**Simulation:**
- white dwarf detonation

**Platform:**
- Blue Gene/P

**Experiment:**
- 8192 vs. 256 processors

**Scaling type:**
- weak

- Nova outbursts on white dwarfs
- Laser-driven shock instabilities
- Helium burning on neutron stars
- Rayleigh-Taylor instability

Figures courtesy of FLASH Team, University of Chicago
Scalability Analysis of Flash3 (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
subroutine rhsf accounts for 13.0% of the multicore scaling loss in the execution.

Execution time increases 1.65x in subroutine rhsf.
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

• Pinpointing scalability bottlenecks
  — scalability bottlenecks on large-scale parallel systems
  — scaling on multicore processors

• Understanding temporal behavior

• Assessing process variability

• Understanding threading, GPU, and memory hierarchy
  — blame shifting
  — attributing memory hierarchy costs to data

• Summary and challenges ahead
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
Outline

• Overview of Rice’s HPCToolkit

• Pinpointing scalability bottlenecks
  — scalability bottlenecks on large-scale parallel systems
  — scaling on multicore processors

• Understanding temporal behavior

• Assessing process variability

• Understanding threading, GPU, and memory hierarchy
  — blame shifting
  — attributing memory hierarchy costs to data

• Summary and challenges ahead
MPBS @ 960 cores, radix sort

Two views of load imbalance since not on a $2^k$ cores
Outline

• Overview of Rice’s HPCToolkit

• Pinpointing scalability bottlenecks
  — scalability bottlenecks on large-scale parallel systems
  — scaling on multicore processors

• Understanding temporal behavior

• Assessing process variability

• Understanding threading, GPU, and memory hierarchy
  — blame shifting
  — attributing memory hierarchy costs to data

• Summary and challenges ahead
Blame Shifting

- Problem: in many circumstances sampling measures symptoms of performance losses rather than causes
  - worker threads waiting for work
  - threads waiting for a lock
  - MPI process waiting for peers in a collective communication
  - idle GPU waiting for work

- Approach: shift blame for losses from victims to perpetrators
  - blame code executing while other threads are idle
  - blame code executed by lock holder when thread(s) are waiting
  - blame processes that arrive late to collectives
  - shift blame between CPU and GPU for hybrid code
Directed Blame Shifting

• Example:
  — threads waiting at a lock are the symptom
  — the cause is the lock holder

• Approach: blame lock waiting on lock holder

![Diagram showing directed blame shifting process]

lockwait

acquire lock
release lock

accumulate samples in a global hash table indexed by the lock address

lock holder accepts these samples when it releases the lock
Example: Directed Blame Shifting for Locks

Blame a lock holder for delaying waiting threads

- Charge all samples that threads receive while awaiting a lock to the lock itself
- When releasing a lock, accept blame at the lock

almost all blame for the waiting is attributed here (cause)

all of the waiting occurs here (symptom)
Undirected Blame Shifting

- **Example:**
  - threads idling waiting for work are the symptom
  - the cause is insufficiently parallel work being executed by others

- **Approach:** each working threads proportionally blames itself for instantaneous idling by others when it is sampled
Performance Expectations for Hybrid Code with Blame Shifting

GPU Successes with HPCToolkit

- **LAMMPS**: identified hardware problem with Keeneland system
  - improperly seated GPUs were observed to have lower data copy bandwidth

- **LLNL’s LULESH**: identified that dynamic memory allocation using cudaMalloc and cudaFree accounted for 90% of the idleness of the GPU
**Data Centric Analysis**

- **Goal:** associate memory hierarchy performance losses with data
- **Approach**
  - intercept allocations to associate with their data ranges
  - measure latency with various PMU capabilities
    - instruction-based sampling (AMD Opteron)
    - precise event-based sampling + load latency facility (Intel)
    - marked instructions (IBM Power)
  - 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

• Pinpointing scalability bottlenecks
  — scalability bottlenecks on large-scale parallel systems
  — scaling on multicore processors

• Understanding temporal behavior

• Assessing process variability

• Understanding threading, GPU, and memory hierarchy
  — blame shifting
  — attributing memory hierarchy costs to data

• Summary and challenges ahead
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
  - bottlenecks in hybrid code
  - problematic data structures
  - hardware counters for detailed diagnosis
- Other capabilities
  - attribute memory leaks back to their full calling context
HPCToolkit Status

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

• Available as open source software at http://hpctoolkit.org

• Emerging capabilities
  — NVIDIA GPU
    • measurement and reporting using GPU hardware counters
  — data centric analysis
  — OpenMP analysis using OMPT
Problem: calling context for parallel regions and tasks is not readily available to tools.
Key OMPT Design Objectives

- Enable tools to gather information and associate costs with application source and runtime system
  - provide interface for low-overhead sampling-based tools
  - enable tools to reconstruct application-level profiles
    - alternative to implementation-level view
  - associate activity of a thread at any point in time with a state
    - enable performance tools to monitor behavior

- Negligible overhead if OMPT interface is not in use

- Define support for trace-based performance tools
Integrated View of MPI+OpenMP with OMPT

LLNL’s luleshMPI_OMP (8 MPI x 3 OMP), 30, REALTIME@1000

source view

thread view

metric view
Tool Challenges Ahead

- Address challenges of emerging systems
  - heterogeneity (e.g., on-chip; host + accelerator)
  - growth in thread counts: MIC supports 200+ threads
  - increasing scale of systems (e.g., Sequoia)

- Identify causes rather than symptoms (blame shifting)

- Measure and analyze all facets of application performance
  - CPU, accelerator, data movement, synchronization, I/O, power
  - interactions: HW, other jobs, system software

- Analyze asynchronous activities

- Support dynamic adaptation of software
  - measurements and decision algorithms to drive adaptation
  - assessment of adaptation policies

- Provide analysis to support higher level insight, diagnosis, and guidance
HPCToolkit Capabilities at a Glance

- **Attribute Costs to Code**
- **Pinpoint & Quantify Scaling Bottlenecks**
- **Assess Imbalance and Variability**
- **Analyze Behavior over Time**
- **Shift Blame from Symptoms to Causes**
- **Associate Costs with Data**

hpctoolkit.org