A Slice of CScADS: Performance Tools for Petascale Platforms

John Mellor-Crummey
Laksono Adhianto, Mike Fagan, Michael Franco, Mark Krentel, Reed Landrum, Xu Liu, Nathan Tallent

Department of Computer Science
Rice University
John Mellor-Crummey, Keith Cooper

Peter Beckman, Ewing Lusk

Jack Dongarra

Bart Miller

Katherine Yelick
CScADS Mission

• Provide open source software systems, tools, and components that address a spectrum of needs
  – directly usable by application experts
  – support development of enabling technologies by the CS community
• Target architectures of critical interest to DOE
  – Cray XT
  – Blue Gene/P
  – multicore processors in general
• Engage DOE application teams and vendors
• Engage the research community in SciDAC challenges

SciDAC-2 Mission

• Develop comprehensive scientific computing software infrastructure to enable petascale science
• Develop new generation of data management and knowledge discovery tools for large data sets
Vertical integration across the petascale software stack

- System software for leadership computing platforms
- Communication libraries
- Math libraries
- Open source compilers
- **Performance tools and infrastructure**
- Application engagement: analysis and tuning
  - e.g., Annual CScADS Workshop on Leadership Computing
    - experts worked with approximately 200 INCITE and SciDAC code developers to help them scale to DOE’s largest systems
Key Performance Questions

- Why doesn’t my application scale as well as I hoped?
- How can I identify bottlenecks in multithreaded node programs?
- How is my code performing relative to peak performance?
  - if my code is not performing well, what is the nature of its problems?
Performance Tool Requirements

• Cope with complex application characteristics
  – large, multi-lingual programs
  – fully optimized code: loop optimization, templates, inlining
  – binary-only libraries, sometimes partially stripped
  – hybrid programs: MPI + OpenMP

• Cope with complex execution environments
  – static or dynamic binaries
  – batch jobs

• Provide effective performance analysis
  – pinpoint and quantify problems
  – yield actionable results

• Scale to leadership computing platforms
Outline

• Introduction to HPCToolkit

• Five new approaches for analyzing parallel program performance
  – scalability analysis using call path profiles [SC09]
  – blame shifting to analyze lock contention in threaded codes [PPoPP10]
  – pinpointing load imbalance in parallel codes [SC10]
  – understanding temporal dynamics of parallel codes
  – data centric analysis of program performance

• Conclusions
• Compile and link for production
  – with full optimization
• For statically-linked executables (e.g. for Cray XT or BG/P)
  – use `hpclink` script to incorporate our monitoring library
Measure execution unobtrusively
- launch optimized application binaries
- collect 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
- instruction pointer

Call path sample
- return address
- return address
- return address

Calling Context Tree (CCT)

Overhead proportional to sampling frequency ...
... not call frequency
Analyze binary to recover program structure
- analyze machine code, line map, and debugging information
- extract loop nesting information and identify inlined procedures
- map transformed loops and procedures back to source
• Combine multiple profiles
  – multiple threads; multiple processes; multiple executions
• Correlate measurements to static and dynamic program structure
HPCToolkit Performance Tools

- Explore performance data from multiple perspectives
- Rank order by metrics to focus on what’s important
- Compute derived metrics to gain insight
- Explore call stack traces to understand transient behavior

presentation

interpret profile correlate w/ source [hpcprof]

database

program structure

binary analysis [hpcstruct]

profile execution

call stack profile

optimized binary

compile & link

source code
MOAB Mesh Library from ITAPS

calling context view

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

• Introduction to HPCToolkit
• Five new approaches for analyzing parallel program performance
  – scalability analysis using call path profiles [SC09]
  – blame shifting to analyze lock contention in threaded codes [PPoPP10]
  – pinpointing load imbalance in parallel codes [SC10]
  – understanding temporal dynamics of parallel codes
  – data centric analysis of program performance
• Conclusions
Pinpointing Scalability Bottlenecks

![Graph showing efficiency versus CPUs]

- **Efficiency**
- **CPUs**

- **Ideal efficiency**
- **Actual efficiency**

Note: higher is better
Scalability Analysis Challenges

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

- **Monitoring**
  - bottleneck nature: computation, data movement, synchronization?
  - pragmatics: need low data volume and low perturbation

Example climate code skeleton

```
main
\_ land
  \_ wait
\_ sea ice
  \_ wait
\_ ocean
  \_ wait
\_ atmosphere
  \_ wait
```
Analyzing Weak Scaling: 1K to 10K processors

Weak 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: FLASH

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

Figures courtesy of FLASH Team, University of Chicago
Pinpointing a Scalability Loss in Flash

21% of the program’s scaling loss is due to a loop over all processors in the adaptive mesh refinement setup called during program initialization.
Improved Flash Scaling of AMR Setup

Graph courtesy of Anshu Dubey, U Chicago
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
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

- Introduction to HPCToolkit
- Five new approaches for analyzing parallel program performance
  - scalability analysis using call path profiles [SC09]
  - blame shifting to analyze lock contention in threaded codes [PPoPP10]
  - pinpointing load imbalance in parallel codes [SC10]
  - understanding temporal dynamics of parallel codes
  - data centric analysis of program performance
- Conclusions
• 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

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

1-5% overhead

lock contention accounts for 23.5% of execution time.

Adding futures to shared global work queue.
Outline

- Introduction to HPCToolkit
- Five new approaches for analyzing parallel program performance
  - scalability analysis using call path profiles [SC09]
  - blame shifting to analyze lock contention in threaded codes [PPoPP10]
  - pinpointing load imbalance in parallel codes [SC10]
  - understanding temporal dynamics of parallel codes
  - data centric analysis of program performance
- Conclusions
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*

Text and figures courtesy of PFLOTRAN Team
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 arrivals at Allreduce
Outline

- Introduction to HPCToolkit
- Five new approaches for analyzing parallel program performance
  - scalability analysis using call path profiles [SC09]
  - blame shifting to analyze lock contention in threaded codes [PPoPP10]
  - pinpointing load imbalance in parallel codes [SC10]
  - understanding temporal dynamics of parallel codes
  - data centric analysis of program performance
- Conclusions
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

• Introduction to HPCToolkit
• Five new approaches for analyzing parallel program performance
  – scalability analysis using call path profiles [SC09]
  – blame shifting to analyze lock contention in threaded codes [PPoPP10]
  – pinpointing load imbalance in parallel codes [SC10]
  – understanding temporal dynamics of parallel codes
  – data centric analysis of program performance
• Conclusions
Data Centric Analysis

- 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
Conclusions

- Observe 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