Gaining Insight into Parallel Program Performance Using Sampling

John Mellor-Crummey
DOE Center for Scalable Application Development Software
johnmc@cs.rice.edu

Collaborators: Nathan Tallent, Michael Fagan, Mark Krentel, Laksono Adhianto, Xu Liu, Reed Landrum, Michael Franco
Motivation

- Complex hardware
  - multi-level parallelism
    - ILP, short vectors, multiple cores, multiple sockets, multiple nodes
  - large-scale parallelism
- Sophisticated software
  - multiphysics, multiscale, adaptive
- Wide gap between peak and typical performance

Challenges

- Understand where and why performance losses occur in sophisticated parallel codes on complex parallel hardware
- Identify opportunities for improvement
- Quantify potential benefits
Performance Analysis Goals

• Accurate measurement of parallel scientific codes
  — large, multi-lingual programs
  — fully optimized code: loop optimization, templates, inlining
  — binary-only libraries, sometimes partially stripped
  — complex execution environments
    • dynamic loading or static binaries
    • SPMD parallel codes with threaded node programs
    • batch jobs
  — production executions

• Effective performance analysis
  — pinpoint and explain problems
    • intuitive enough for scientists and engineers
    • detailed enough for compiler writers
  — yield actionable results

• Scalable to petascale systems
• 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

source code

optimized binary

profile execution

binary analysis

[hpcref]

interpret profile correlate w/ source

[hpcref]

call stack profile

program structure

presentation

database

compile & link
HPCToolkit Performance Tools

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

presentation

interpret profile correlate w/ source [hpcprof]
database

compile & link
source code
optimized binary
profile execution
binary analysis [hpcstruct]
call stack profile
program structure
HPCToolkit Performance Tools

- Combine multiple profiles
  - multiple threads; multiple processes; multiple executions
- Correlate measurements to static & dynamic program structure

presentation

optimized binary

source code

profile execution

call stack profile

binary analysis [hpcstruct]

program structure

interpret profile correlate w/ source [hpcprof]

database
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

source code → optimized binary → profile execution

presentation → database → interpret profile correlate w/ source [hpcprof]

call stack profile → program structure

binary analysis [hpcstruct]
Attribution to Static + Dynamic Context

- inlined procedures
- loops
- function calls in full context
• Call path profiling in HPCToolkit
• Pinpointing and quantifying scalability bottlenecks
• Blame shifting
  — analyzing multithreaded computations based on work stealing
  — quantifying the impact of lock contention on threaded code
  — pinpointing load imbalance in parallel codes
• Understanding execution behavior over time
• Associating memory hierarchy inefficiency with data
• Conclusions
• Challenges ahead
• Related work
Call Path Profiling

Measure and attribute costs in their \textit{calling} 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 (CCT)

Overhead proportional to sampling frequency... 
...not call frequency
Unwinding Fully-optimized Parallel Code

Unwinding using demand-driven binary analysis

• Identify procedure bounds
  — for dynamically-linked code, do this at runtime
  — for statically-linked code, do this at compile time

• Compute unwind recipes for a procedure on the fly
  — scan the procedure’s object code, tracking the locations of
    • caller’s program counter
    • caller’s frame and stack pointer
  — create unwind recipes between pairs of frame-relevant instructions

• Processors: x86-64, PowerPC (BG/P), MIPS (SiCortex)

• Results
  — accurate call path profiles
  — overheads of < 2% for sampling frequencies of 200/s

Outline

- Call path profiling in HPCToolkit
- Pinpointing and quantifying scalability bottlenecks
  - Blame shifting
    - analyzing multithreaded computations based on work stealing
    - quantifying the impact of lock contention on threaded code
    - pinpointing load imbalance in parallel codes
- Understanding execution behavior over time
- Associating memory hierarchy inefficiency with data
- Conclusions
- Challenges ahead
- Related work
The Problem of Scaling Losses

Note: higher is better
Pinpointing and Quantifying Scalability Bottlenecks

Weak scaling: no coefficients
Strong scaling: needs red coefficients
Scalability Analysis of 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
13.4% of the scaling losses in Flash execution are due to the use of a “digital orrery” all-to-all communication pattern as part of adaptive mesh refinement. This shows up in the code as a loop over all processors containing pairwise communication. This single problem accounts for almost 1/4 of the scalability loss during Flash’s evolution phase.

This problem caused a 21% scalability loss in the initialization phase as well.
Improved Flash Scaling of AMR Setup

Graph courtesy of Anshu Dubey, U Chicago

improved scalability after fixing AMR scaling bottleneck described in previous slide (lower is better)
Scalability Losses at the Loop Level

Execution time increases 2.8x in the loop that scales worst

S3D code (Sandia CRF)
PI: Jackie Chen
DNS of turbulent combustion

loop contributes a 6.9% scaling loss to the execution
Outline

- Call path profiling in HPCToolkit
- Pinpointing and quantifying scalability bottlenecks
  - Blame shifting
    - analyzing multithreaded computations based on work stealing
    - quantifying the impact of lock contention on threaded code
    - pinpointing load imbalance in parallel codes
- Understanding execution behavior over time
- Associating memory hierarchy inefficiency with data
- Conclusions
- Challenges ahead
- Related work
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
- Approach: shift blame for losses from victims to perpetrators
- Flavors
  - active measurement
  - analysis only
cilk int fib(n) {
    if (n < 2) return n;
    else {
        int x, y;
        x = spawn fib(n-1);
        y = spawn fib(n-2);
        sync;
        return (x + y);
    }
}

asynchronous calls create logical tasks that only block at a sync...

...quickly create significant logical parallelism.
Cilk Program Execution using Work Stealing

- Challenge: Mapping logical tasks to compute cores
- Cilk approach:
  - lazy thread creation plus work-stealing scheduler
    - spawn: a potentially parallel task is available
    - an idle thread steals tasks from a random working thread

Possible Execution:
thread 1 begins
thread 2 steals from 1
thread 3 steals from 1 etc...
Wanted: Call Path Profiles of Cilk

- Consider thread 3:
  - physical call path:
    - logical call path:

Logical call path profiling: Recover full relationship between physical and user-level execution

Work stealing separates user-level calling contexts in space and time
Effective Performance Analysis

Three Complementary Techniques:

- Recover *logical calling contexts* in presence of work-stealing

```cilk
int fib(n) {
  if (n < 2) {...}
  else {
    int x, y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x + y);
  }
}
```

- Quantify *parallel idleness* (insufficient parallelism)
- Quantify *parallel overhead*
- Attribute *idleness* and *overhead* to *logical contexts* — at the source level
Measuring & Attributing Parallel Idleness

• Metrics: Effort = “work” + “idleness”
  — associate metrics with user-level calling contexts
  — insight: attribute idleness to its cause: context of working thread
    • a thread looks past itself when ‘bad things’ happen to others

• Work stealing-scheduler: one thread per core
  — maintain W (# working threads) and I (# idling threads)
    • slight modifications to work-stealing run time
      – atomically incr/decr W when thread exits/enters scheduler
    • when a sample event interrupts a working thread
      – I = #cores − W
      – apportion others’ idleness to me: I / W

• Example: Dual quad-cores; on a sample, 5 are working:
  for each worker:
    Wald += 1
    I = 3/5
  \[ \sum Wald = 5 \]
  \[ \sum I = 3 \]
  idle: drop sample (it’s in the scheduler!)
Parallel Overhead

- Parallel overhead
  - when a thread works on something other than user code
    - (we classify waiting for work as idleness)

- Pinpointing overhead with call path profiling
  - impossible, without prior arrangement
    - work and overhead are both machine instructions
  - insight: have compiler tag instructions as overhead
  - quantify samples attributed to instructions that represent ovhd
    - use post-mortem analysis
Top-down Work for Cilk ‘Cholesky’

13.5% of cilk_main’s total effort was spent in idleness...

2.97% and 0.215% of cholesky’s total effort was spent in idleness and overhead.
Bottom-up Idleness for Cilk ‘Cholesky’

We can pinpoint and quantify the effect of serialization.

Pinpoints serial initialization-finalization routines.
Using Parallel Idleness & Overhead

- Total effort = useful work + idleness + overhead
- Enables powerful and precise interpretations

<table>
<thead>
<tr>
<th>idleness</th>
<th>overhead</th>
<th>interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>low</td>
<td>low</td>
<td>effectively parallel</td>
</tr>
<tr>
<td>low</td>
<td>high</td>
<td>coarsen concurrency granularity</td>
</tr>
<tr>
<td>high</td>
<td>low</td>
<td>refine concurrency granularity</td>
</tr>
<tr>
<td>high</td>
<td>high</td>
<td>switch parallelization strategies</td>
</tr>
</tbody>
</table>

- Normalize w.r.t. total effort to create
  — percent idleness or percent overhead

Nathan Tallent, John Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. PPoPP 2009, Raleigh, NC.
Outline

• Call path profiling in HPCToolkit
• Pinpointing and quantifying scalability bottlenecks
• Blame shifting
  — analyzing multithreaded computations based on work stealing
  — quantifying the impact of lock contention on threaded code
  — pinpointing load imbalance in parallel codes
• Understanding execution behavior over time
• Associating memory hierarchy inefficiency with data
• Conclusions
• Challenges ahead
• Related work
Understanding Lock Contention

• Lock contention causes idleness
  — explicitly threaded programs (Pthreads, etc)
  — implicitly threaded programs (critical sections in OpenMP, Cilk...)

• Use “blame-shifting” to shift blame 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 assoc. with 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

*different technique handles blocking
Lock contention in MADNESS

Quantum chemistry; MPI + pthreads

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

- Call path profiling in HPCToolkit
- Pinpointing and quantifying scalability bottlenecks
- Blame shifting
  - analyzing multithreaded computations based on work stealing
  - quantifying the impact of lock contention on threaded code
  - pinpointing load imbalance in parallel codes
- Understanding execution behavior over time
- Associating memory hierarchy inefficiency with data
- Conclusions
- Challenges ahead
- Related work
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
• Call path profiling in HPCToolkit
• Pinpointing and quantifying scalability bottlenecks
• Blame shifting
  — analyzing multithreaded computations based on work stealing
  — quantifying the impact of lock contention on threaded code
  — pinpointing load imbalance in parallel codes
• Understanding execution behavior over time
• Associating memory hierarchy inefficiency with data
• Conclusions
• Challenges ahead
• Related work
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
Call Path Tracing for Parallel Programs

1D FFT, CAF 2.0, 256 processes, Cray XT, 128M/core

Scalable Fine-grained Call Path Tracing, Submitted to IPDPS 2011.
Call Path Tracing for Parallel Programs

PFLOTRAN: Fortran+MPI, 8184 cores, Cray XT (982s)

Scalable Fine-grained Call Path Tracing, Submitted to IPDPS 2011.
Call Path Tracing for Parallel Programs

PFLOTRAN: Fortran+MPI, 8184 cores, Cray XT (1st minute)
Outline

- Call path profiling in HPCToolkit
- Pinpointing and quantifying scalability bottlenecks
- Blame shifting
  - analyzing multithreaded computations based on work stealing
  - quantifying the impact of lock contention on threaded code
  - pinpointing load imbalance in parallel codes
- Understanding execution behavior over time
- **Associating memory hierarchy inefficiency with data**
- Conclusions
- Challenges ahead
- Related work
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
Conclusions

• 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
  — where insufficient parallelism lurks
  — sources of lock contention
  — load imbalance
  — temporal dynamics
  — problematic data structures
Some Challenges Ahead

• Support characteristics of emerging hardware and software
  — heterogeneous hardware
    • manycore, CPU+GPU
    • dynamic power and frequency scaling
  — software
    • one-sided communication
    • asynchronous operations
    • dynamic parallelism
    • adaptation
    • failure recovery

• Augment monitoring capabilities throughout the stack
  — hardware, OS, runtime, language-level API

• Improve data management for extreme scale parallelism

• Transition from descriptive to prescriptive feedback

• Guide online adaptation and tuning
Some Related Work

- Sampling
  - e.g., gprof, Speedshop, Shark, PTU, DCPI, Oprofile, CrayPat

- Instrumentation
  - e.g., Tau, Vtune, IBM HPC Toolkit, Dyninst, CrayPat, Pin

- Tracing
  - e.g., vt, Tau, CEBPA,

- Call stack profiling
  - e.g., mpiP, Tau, PTU, Shark

- Visualization
  - e.g., Paraver, Projections, Vampir, Jumpshot, EXPERT

- Parallel Analysis
  - e.g., Scalasca

- Analysis
  - e.g., IBM HPCS Toolkit, Cray Apprentice, EXPERT, PerfExpert
HPCToolkit Capabilities at a Glance

Attribute Costs to Code

Pinpoint & Quantify Scaling Bottlenecks

Associate Costs with Data

Analyze Behavior over Time

Shift Blame from Symptoms to Causes

Assess Imbalance and Variability

hpctoolkit.org
Measurement

• Binary analysis for (1) recovering functions in partially stripped code, (2) unwinding fully-optimized code, (3) recovering program structure
• Nearly perfect call stack sampling of fully optimized code with low overhead

*Binary Analysis for Measurement and Attribution of Program Performance, PLDI, June 2009. Distinguished Paper Award.*

*Pinpointing Locality Problems Using Data-centric Analysis, Submitted to CGO 2011, April 2011*

Pinpoint Scalability Bottlenecks using Differential Profiling

*Scalability Analysis of SPMD Codes using Expectations, ICS, June 2007*

Pinpoint Performance Losses in Multithreaded Executions

*Effective Performance Measurement and Analysis of Multithreaded Applications, PPoPP, February 2009.*

*Analyzing Lock Contention in Multithreaded Applications, PPoPP, January 2010*
Novel Capabilities of HPCToolkit - II

Performance Analysis using Sampling on Leadership Platforms
- Diagnosing Performance Bottlenecks in Emerging Petascale Applications, SC09, November 2009
- Scalable Identification of Load Imbalance using Call Path Profiles, SC10, November 2010

User Interfaces
- Effectively Presenting Call Path Profiles of Application Performance, PSTI, September 2010.
- Scalable Fine-grained Call Path Tracing, Submitted to IPDPS 2011.

Overview Paper
Additional Tool Screenshots
Execution Cost Breakdown (Routine-Level)

Flash on Blue Gene/P, 8K cores, white dwarf detonation

Costs sorted by exclusive time spent in individual routines
Note: only the routines shown in blue are user code
BG/P DCMF Communication Layer costs
Execution Cost Attribution (Callers View)

Flash on Blue Gene/P, 8K cores, white dwarf detonation

Looking up the call chain to see where the callers that caused costs to be incurred for tree reductions. Most of the cost is incurred by guard cell filling and flux conservation.
Looking up down the call chain to see where the most of the time was spent. 80.5% is spent in the loop that calls the hydrodynamics simulation. 52.4% of the time is spent in the hydro routine (or below). The rest is spent in other routines called from the main loop.
Execution Cost Attribution (Top-Down)

PFLOTRAN, Cray XT, 8184 cores, Hanford problem

66.5% of the cycles are spent in the transport calculation. 30.8% of the cycles are spent in the flow calculation.
Detailed analysis of the transport calculation: Most of the time is spent in the PETSc inside the Biconjugate gradient solver.

Overall: 1 FLOP every 7.4 cycles