Gaining Insight into Parallel Program Performance Using Sampling

John Mellor-Crummey, Laksono Adhianto, Mike Fagan, Mark Krentel, Nathan Tallent

Department of Computer Science
Rice University
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 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
Outline

- Evaluating context-sensitive behavior
- Pinpointing and quantifying scalability bottlenecks
- Analyzing multithreaded computations with work stealing
- Quantifying the impact of lock contention on threaded code
- Understanding how computations evolve
- Work in progress
State of the Art: 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 based on 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
  — 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
  — almost flawless unwinding
  — overheads of < 2% for sampling frequencies of 200/s

Detailed Attribution: MOAB Mesh Benchmark

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

• Evaluating context-sensitive behavior

• Pinpointing and quantifying scalability bottlenecks

• Analyzing multithreaded computations with work stealing

• Quantifying the impact of lock contention on threaded code

• Understanding how computations evolve

• Work in progress
The Problem of Scaling

Note: higher is better
Weak scaling: no coefficients
Strong scaling: needs red coefficients

\[ P \times 600K \rightleftharpoons 400K \times Q \]

C. Coarfa et al. Scalability analysis of SPMD codes using expectations. ICS 2007, Seattle, WA.
N. Tallent et al. Diagnosing scalability bottlenecks in emerging petascale applications. SC 2009, Portland, OR.
Scalability Analysis Demo

**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
S3D: Multicore Losses at the Procedure Level

```
subroutine rhsf(q, rhs)
!
Changes
!
Ramanan Sankaran - 01/04/05
!
1. Diffusive fluxes are computed without having to convert units.
2. Ignore older comments about conversion to CGS units.
3. This saves a lot of flops.
4. Mixavg and Lewis transport modules have been made interchangeable
   by adding dummy arguments in both.
!
Author: James Sutherland
Date: April, 2002
!
This routine calculates the time rate of change for the
momentum, continuity, energy, and species equations.
```

<table>
<thead>
<tr>
<th>Scope</th>
<th>1-core (ms) (l)</th>
<th>1-core (ms) (E)</th>
<th>8-core(1) (ms) (l)</th>
<th>8-core(1) (ms) (E)</th>
<th>Multicore Loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>rhsf</td>
<td>1.07e08 96.5%</td>
<td>6.60e06 5.9%</td>
<td>1.77e08 94.1%</td>
<td>1.65e07 8.8%</td>
<td>9.92e06 13.0%</td>
</tr>
<tr>
<td>difflux_proc_looptool</td>
<td>2.68e06 2.6%</td>
<td>2.68e06 2.6%</td>
<td>8.12e06 4.3%</td>
<td>8.12e06 4.3%</td>
<td>5.27e06 6.9%</td>
</tr>
<tr>
<td>integrate_erk_stage_lt</td>
<td>1.09e08 98.1%</td>
<td>1.25e06 1.1%</td>
<td>1.84e08 97.9%</td>
<td>5.94e06 3.2%</td>
<td>4.70e06 6.1%</td>
</tr>
<tr>
<td>GET_MASS_FRAC.in.VARIABLES_M</td>
<td>1.49e06 1.3%</td>
<td>1.49e06 1.3%</td>
<td>6.08e06 3.2%</td>
<td>6.08e06 3.2%</td>
<td>4.59e06 6.0%</td>
</tr>
<tr>
<td>ratx</td>
<td>1.01e07 9.1%</td>
<td>1.00e07 9.0%</td>
<td>4.41e07 23.5%</td>
<td>1.40e07 7.4%</td>
<td>3.95e06 5.2%</td>
</tr>
<tr>
<td>qssa</td>
<td>3.52e06 3.2%</td>
<td>3.52e06 3.2%</td>
<td>5.71e06 3.0%</td>
<td>5.71e06 3.0%</td>
<td>2.18e06 2.9%</td>
</tr>
<tr>
<td>ratt</td>
<td>3.26e07 29.2%</td>
<td>1.48e07 13.3%</td>
<td>4.38e07 23.3%</td>
<td>1.66e07 8.8%</td>
<td>1.76e06 2.3%</td>
</tr>
<tr>
<td>CALC_INV_AVG_MOL_WT.in.THER</td>
<td>7.9e05 0.9%</td>
<td>9.7e05 0.9%</td>
<td>2.68e06 1.4%</td>
<td>2.68e06 1.4%</td>
<td>1.76e06 2.2%</td>
</tr>
<tr>
<td>computeheatflux_looptool</td>
<td>1.46e06 1.3%</td>
<td>1.46e06 1.3%</td>
<td>2.88e06 1.5%</td>
<td>2.88e06 1.5%</td>
<td>1.41e06 1.8%</td>
</tr>
<tr>
<td>rdwdot</td>
<td>3.09e06 2.8%</td>
<td>3.09e06 2.8%</td>
<td>4.33e06 2.3%</td>
<td>4.33e06 2.3%</td>
<td>1.24e06 1.6%</td>
</tr>
</tbody>
</table>
S3D: Multicore Losses at the Loop Level
- Evaluating context-sensitive behavior
- Pinpointing and quantifying scalability bottlenecks
- Analyzing multithreaded computations with work stealing
- Quantifying the impact of lock contention on threaded code
- Understanding how computations evolve
- Work in progress
Cilk: A Multithreaded Language

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

\[\text{cilk int fib(n)} \{\]
\[\text{if (n < 2) \{} \ldots \}
\[\text{else} \{\]
\[\text{int x, y;}\]
\[\text{x = spawn fib(n-1);}\]
\[\text{y = spawn fib(n-2);}\]
\[\text{sync;}\]
\[\text{return (x + y);}\]

• 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: \( W \) += 1 \( \sum W = 5 \)
  idle: drop sample
  \( I \) += 3/5 \( \sum I = 3 \)
  (it’s in the scheduler!)
Parallel Overhead

• Parallel overhead:
  — when a thread *works* on something other than user code
    • (we classify delays -- e.g., wait time -- 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.
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

• Evaluating context-sensitive behavior
• Pinpointing and quantifying scalability bottlenecks
• Analyzing multithreaded computations with work stealing
• Quantifying the impact of lock contention on threaded code
• Understanding how computations evolve
• Work in progress
Understanding Lock Contention

- Lock contention => idleness:
  - explicitly threaded programs (Pthreads, etc)
  - implicitly threaded programs (critical sections in OpenMP, Cilk...)

- Use “blame-shifting:” 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

Lock contention accounts for 23.5% of execution time. Adding futures to shared global work queue.

quantum chemistry; MPI + pthreads

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

- Evaluating context-sensitive behavior
- Pinpointing and quantifying scalability bottlenecks
- Analyzing multithreaded computations with work stealing
- Quantifying the impact of lock contention on threaded code
- Understanding how computations evolve
- Work in progress
Understanding Temporal Behavior

• 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
Call Path Sample Trace for GTC

Gyrokinetic Toroidal Code (GTC)

- 32 process MPI program
- Each process has a pair of threads managed with OpenMP

Outline

- Evaluating context-sensitive behavior
- Pinpointing and quantifying scalability bottlenecks
- Analyzing multithreaded computations with work stealing
- Quantifying the impact of lock contention on threaded code
- Understanding how computations evolve
- Work in progress
Work in Progress

- Analyze call path profiles for 100K+ cores in parallel
  - aggregate profile CCTs for different cores to get union CCT
  - compute summary statistics (e.g. min, mean, max, std. deviation)
  - hypothesis: we can apply our top-down methodology for analyzing CCTs to assess profile differences
    - pinpoint and quantify profile differences at a high level
    - drill down using differential analysis of sample profiles

- Develop GUI support for sorting and histogramming profile values to cope with data from thousands of cores

- Using hardware monitoring capabilities to gain insight into data access patterns
  - identify potential for improving locality and data reuse

- Visualize sampled traces for thousands of cores