Scalable Identification of Load Imbalance in Parallel Executions Using Call Path Profiles

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

Rice University

CScADS Workshop
August 2, 2010
HPCToolkit Performance Tools

- Work at **binary level** for language independence
  - unmodified, fully optimized codes w/ binary-only libraries

- **Asynchronous-sampling-based** measurement
  - minimize overhead and distortion; avoid blind spots
  - compact data: well-suited for large-scale parallelism

- Collect and correlate multiple performance metrics
  - diagnosis requires more than one species of metric
  - derived metrics: “unused bandwidth” rather than “cycles”
  - ‘third-party’ metrics: if thread $x$ affects $y$, then blame $x$

- Associate metrics with both static and dynamic context
  - loop nests, procedures, inlined code, calling context

- Support top-down performance analysis
  - avoid overwhelming with the details
HPCToolkit Workflow

- Measure execution unobtrusively
  - hpcrun: launch unmodified dynamically linked applications
  - hpclink: link hpcrun into statically linked applications
  - collect call path profiles for sample-sources of interest
  - on-the-fly analysis: pinpoint idleness (work-stealing, locking)
Call Path Profiling

Measure and attribute costs in their *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)

"main"

Overhead proportional to sampling frequency...
...not call frequency
HPCToolkit Workflow

- Analyze binary to recover program structure
  - extract loop nesting & identify procedure inlining
  - map transformed loops and procedures to source
HPCToolkit Workflow

- **Scalably combine multiple large-scale profiles**
  - integrate dynamic call paths with static source structure
  - attribute measurements to source code
- **Post-mortem analysis:**
  - statistical summary metrics; load imbalance analysis

**Flowchart**:

- **app. source** → **optimized binary** → **compile & link (full optimization)** → **profile execution** → **call path profile** → **program structure**
- **profile execution** → **binary analysis** → **program structure**
- **binary analysis** → **analysis & attribution to source**
- **presentation** → **database** → **analysis & attribution to source**

**Notes**:

- **hpcrun/hpclink**
- **hpcstruct**
- **hpcprof/mpi**
- **hpcviewer**
Scalably* Combining A Large-Scale Profile

\[ \text{canonical CCT} = \bigcup \text{CCT}_i \quad (+ \text{symbolic info}) \]

\[ \sum \mu \sigma \]

\[ O(\text{CCT}_i) \quad O(m_i) \]

align metrics\(_i\) with canonical CCT to form database

* Yes, I know ‘scalably’ is not a word — but it should be.

† SPMD apps
Simple example: Given $x_1, \ldots, x_n$, compute arithmetic mean $\mu$

Accumulate inputs: Apply commutative & associative operations (order doesn’t matter)

Combine $P$ accumulator sets into one

Finalize: apply non-commutative & non-associative operators

$$\mu = a / n$$
Details: PFLOTRAN @ 8K cores, JaguarPF

- **hpcrun** generates a (binary) file per-thread
  - size of output: 7.2 GB (0.9 MB/process)
  - overhead: 3% (230 smpl/sec)
    - relative to unmonitored execution of 15.3 minutes
  - todo:
    - use parallel I/O (e.g., SIONlib)
    - apply sampling at multiple levels

- **hpcprof-mpi**
  - size of data: 3.6 GB
  - execution times:
    - 24 cores: 13.0 minutes
    - 48 cores: 8.0 minutes
    - 96 cores: 5.5 minutes
  - filters ‘uninteresting’ CCT paths
    - filter CCT node \( n \) if for all (additive) metrics \( m \) and all threads \( t \):
      \[- \sum_{t \in \text{threads}} m_I(n, t) < .001\% \sum_{t \in \text{threads}} m_I(root, t) \]  
      (I: inclusive)
  - todo: exploit sparsity of thread-level metric database
HPCToolkit Workflow

- **Presentation**
  - support top-down analysis with interactive viewer
  - show context-sensitive metrics in three views
  - generate plots of context-sensitive thread-level values
  - analyze results anytime, anywhere
Analyzing Load Imbalance Post-Mortem

Given a synchronized procedure $x$, compute imbalance waste, $\overline{B}(x)$.

(a) Actual Execution of $x$

(b) Ideal Execution of $x$ (same algorithm)

\[
\overline{B}(x) = W_{\text{max}}(x) + C_{\text{min}}(x) - [W_\mu(x) + C_{\text{min}}(x)] = W_{\text{max}}(x) - W_\mu(x)
\]

(time of actual exe) \quad (time of ideal exe) \quad (imbalance waste)
What If Procedure Is Not Synchronized?

- Cannot use prior equation to compute imbalance waste
  - e.g.: \( \mathcal{W}_{\text{max}}(x) + C_{\text{min}}(x) \neq \text{time of actual execution} \)
- May be *fine* that procedure \( x \) is imbalanced!
  - E.g.: each process performs different amount of work for \( x \)...
    - ...but is balanced at a \( x \)'s caller
Imbalanced vs. Balanced Procedures

- Cannot apply prior equation to left (a), but could to right (b)

(a) Execution of $x$

(b) Execution of $x$’s Caller

- Challenge: compute imbalance waste in its calling context
  - For what contexts can we compute imbalance waste?
  - Can we identify waste more precisely than using differences in communication time?
1. Identify idleness (exposed waiting): All imbalance is manifested in idleness (e.g., MPI_Cray_Progress_Wait)

2. Identify balance points (procedures or loops): A balance point cannot contribute to imbalance

3. Blame-shift idleness (effect) on closest ancestor balance point (nearer to cause)

4. Scatter plots of thread-level CCT metrics help expose patterns.
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.
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
The End