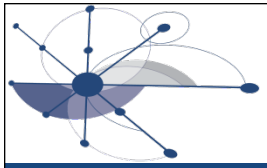
RICE

A Data-centric Profiler for Parallel Programs

Xu Liu
John Mellor-Crummey

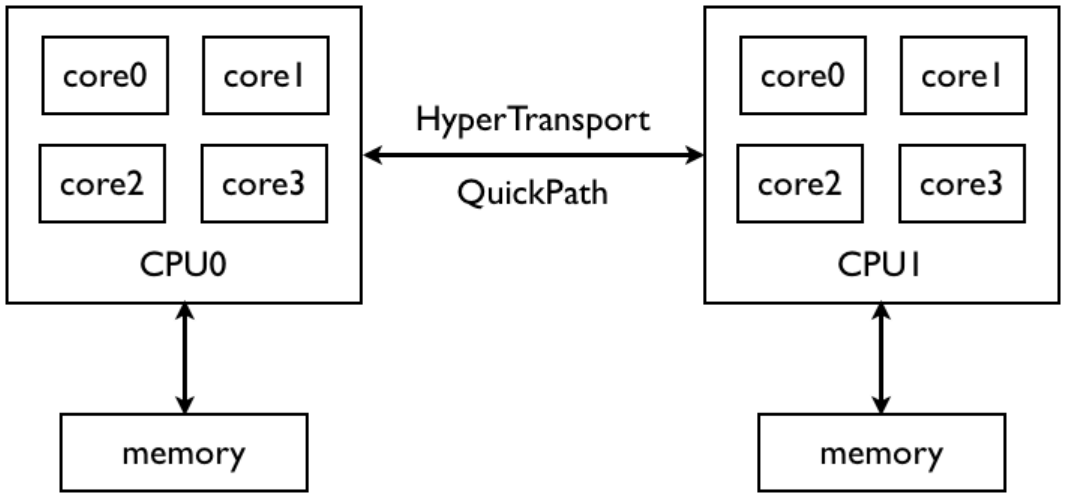
Department of Computer Science
Rice University

Petascale Tools Workshop - Madison, WI - July 16, 2013



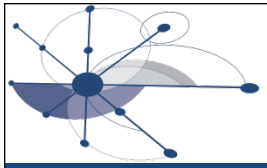
Motivation

- Good data locality is important
  - high performance
  - low energy consumption
- Types of data locality
  - temporal/spatial locality
    - reuse distance
    - data layout
  - NUMA locality
    - remote v.s. local
    - memory bandwidth



remote accesses: high latency, low bandwidth

- Performance tools are needed to identify data locality problems
  - code-centric analysis
  - data-centric analysis



Code-centric v.s. data-centric

- Code-centric attribution
  - problematic code sections
    - instruction, loop, function
- Data-centric attribution
  - problematic variable accesses
  - aggregate metrics of different memory accesses to the same variable
- Code-centric + data-centric
  - data layout ~~match~~ access pattern
  - data layout ~~match~~ computation distribution

```
1: for (i = 0; i < n; i++) {
2:   for(j = 0; j < n; j++) {
3:     for(k = 0; k < n; k++) {
4:       A[i, j, k] = A[i, j, k,] + B[j, i, k] + C[k, j, i];
5:     }
6:   }
7: }
```

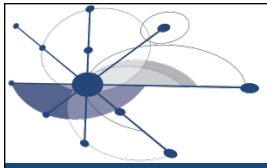
code-centric profiling

line 4: 100% latency

data-centric profiling

array A:
  line 4: 1% latency
array B
  line 4: 10% latency
array C
  line 4: 89% latency

Combination of code-centric and data-centric attributions provides insights



Previous work

- Simulation methods
  - Memspy, SLO, ThreadSpotter ...
  - disadvantages
    - Memspy and SLO have large overhead
    - difficult to simulate complex memory hierarchies
- Measurement methods
  - temporal/spatial locality
    - HPCToolkit, Cache Scope
  - NUMA locality
    - Memphis, MemProf

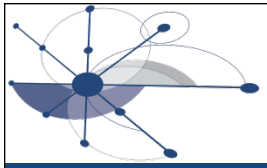
Support both static and heap-allocated variable attributions

Identify both locality problems

Work for both MPI and threaded programs

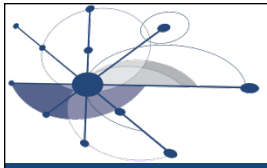
GUI for intuitive analysis

Widely applicable



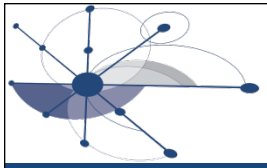
Approach

- A scalable sampling-based call path profiler which
  - performs both code-centric and data-centric attribution
  - identifies locality and NUMA bottlenecks
  - monitors MPI+threads programs running on clusters
  - works on almost all modern architectures
  - incurs low runtime and space overhead
  - has a friendly graphic user interface for intuitive analysis



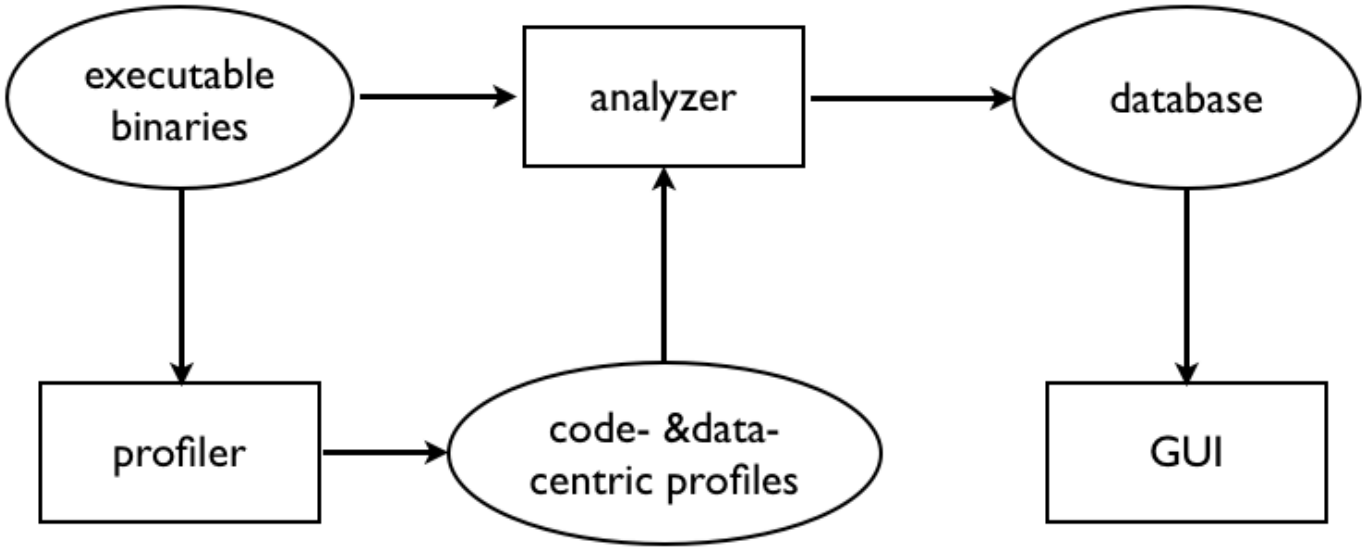
Prerequisite: sampling support

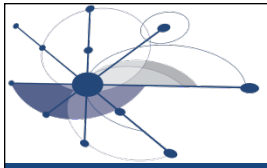
- Sampling features that HPCToolkit needs
  - necessary features
    - sample memory-related events (memory accesses, NUMA events)
    - capture effective addresses
    - record precise IP of sampled instructions or events
  - optional features
    - record useful metrics: data access latency (in CPU cycle)
    - sample instructions/events not related to memory
- Support in modern processors
  - hardware support
    - AMD Opteron 10h and above: instruction-based sampling (IBS)
    - IBM POWER 5 and above: marked event sampling (MRK)
    - Intel Itanium 2: data event address register sampling (DEAR)
    - Intel Pentium 4 and above: precise event based sampling (PEBS)
    - Intel Nehalem and above: PEBS with load latency (PEBS-LL)
  - software support: instrumentation-based sampling (Soft-IBS)



HPCToolkit workflow

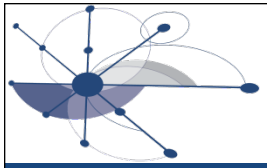
- Profiler: collect and attribute samples
- Analyzer: merge profiles and map to source code
- GUI: display metrics in both code-centric and data-centric views





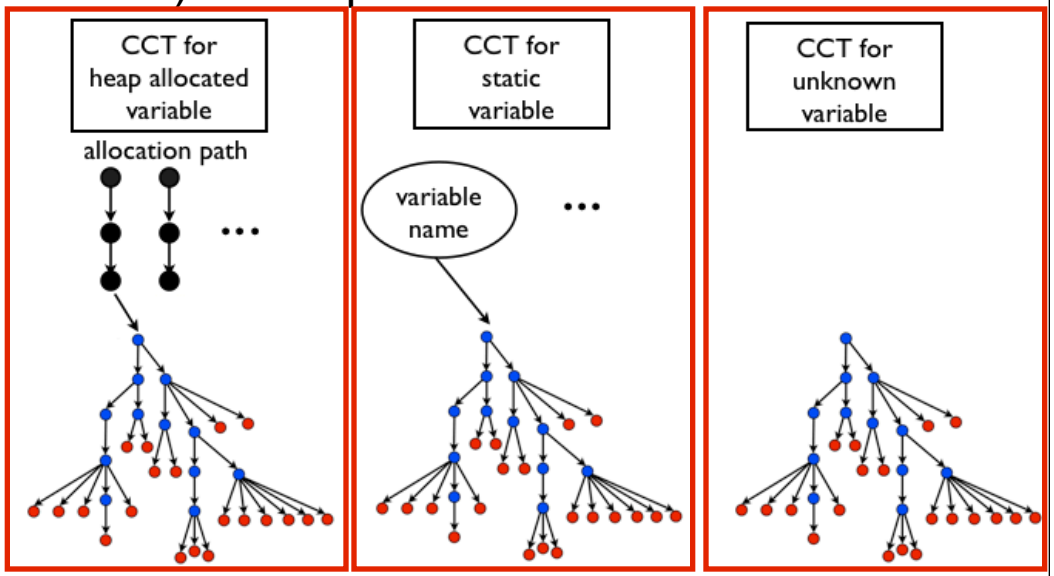
HPCToolkit profiler

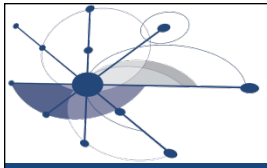
- Record data allocation
  - heap-allocated variables
    - overload memory allocation functions: malloc, calloc, realloc, ...
    - determine the allocation call stack
    - record the pair (allocated memory range, call stack) into a map
  - static variables
    - read symbol tables of the executable and dynamic libraries in use
    - identify the name and memory range for each static variable
    - record the pair (memory range, name) in a map
- Record samples
  - determine the calling context of the sample
  - update the precise IP
  - attribute to data (allocation call path or static variable name) according to effective address touched by instruction



HPCToolkit profiler (cont.)

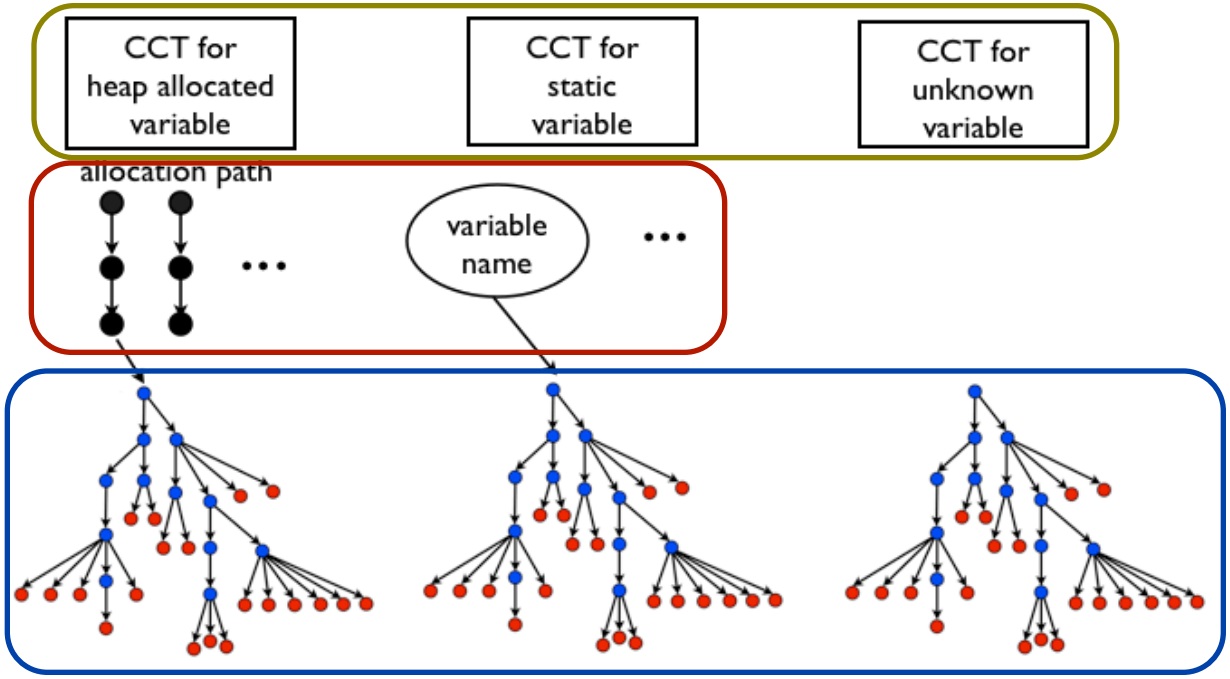
- Data-centric attribution for each sample
  - create three CCTs
  - look up the effective address in the map
    - heap-allocated variables
      - use the allocation call path as a prefix for the current context
      - insert in first CCT
    - static variables
      - copy the name (as a CCT node) as the prefix
      - insert in second CCT
    - unknown variables
      - insert in third CCT
- Record per-thread profiles



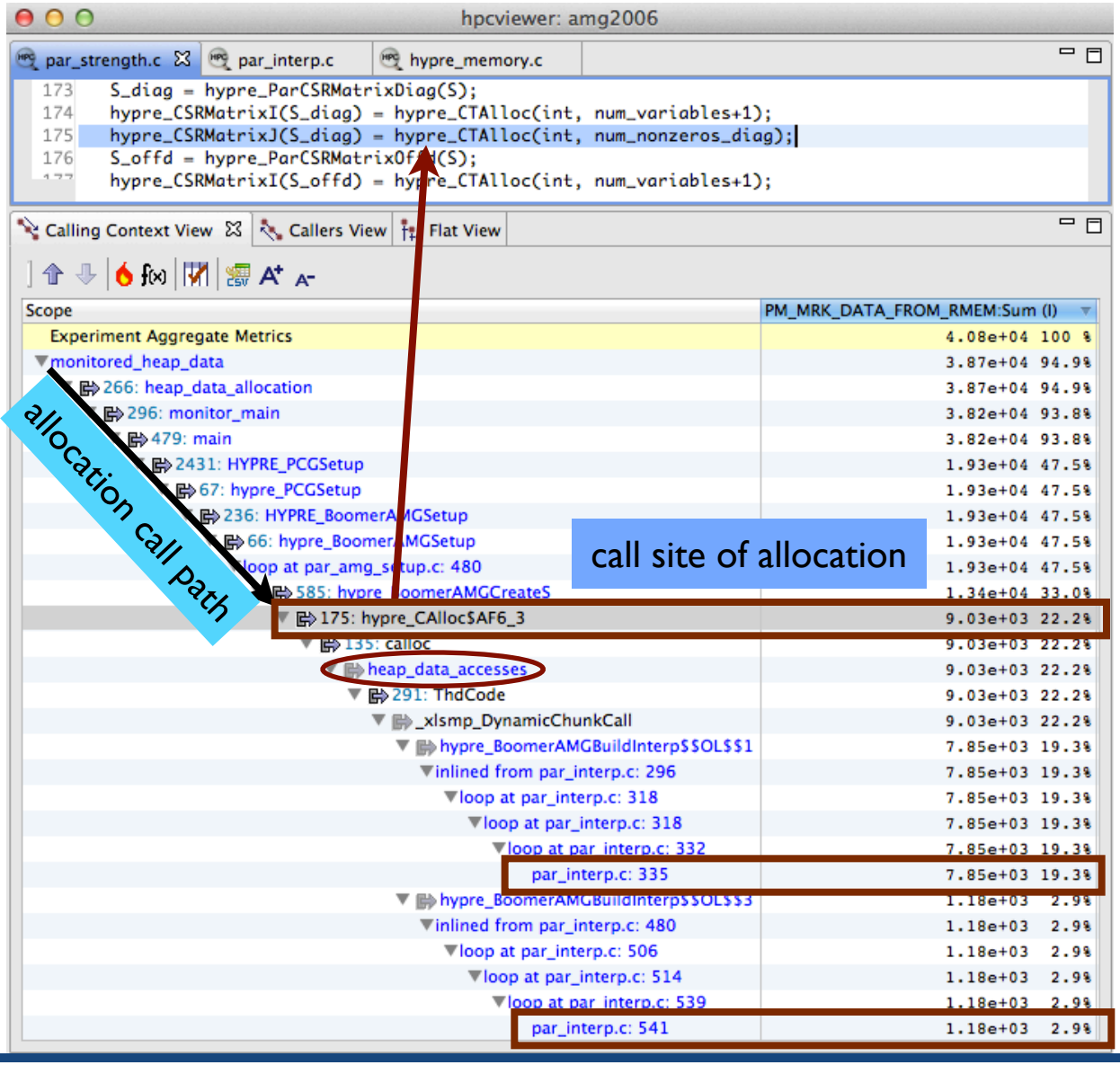


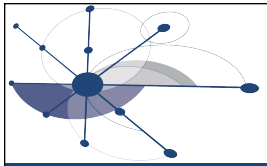
HPCToolkit analyzer

- Merge profiles across threads
  - begin at the root of each CCT
  - merge variables next
    - variables have the same name or allocation call path
  - merge sample call paths finally



GUI: intuitive display





Assess bottleneck impact

- Determine memory bound v.s. CPU bound
  - metric: latency/instruction (>0.1 cycle/instruction → memory bound)

$$l_{ins} = \frac{latency}{\#ins} = \frac{latency}{\#mem} \times \frac{\#mem}{\#ins}$$

average latency per memory access

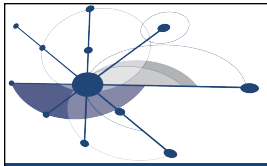
percentage of memory instructions

Sphot: 0.097
S3D: 0.02

- Identify problematic variables and memory accesses
  - metric: latency

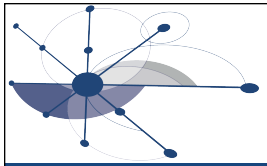
for a variable or a program region:

l_{ins}	$latency$	optimization strategy
low	low	no optimization needed
low	high	optimization would yield little benefit
high	low	low priority for optimization
high	high	high priority for optimization



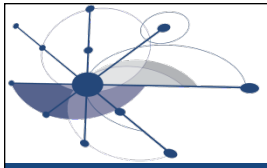
Experiments

- AMG2006
  - MPI+OpenMP: 4 MPI × 128 threads
  - sampling method: MRK on IBM POWER 7
- LULESH
  - OpenMP: 48 threads
  - sampling method: IBS on AMD Magny-Cours
- Sweep3D
  - MPI: 48 MPI processes
  - sampling method: IBS on AMD Magny-Cours
- Streamcluster and NW
  - OpenMP: 128 threads
  - sampling method: MRK on IBM POWER 7



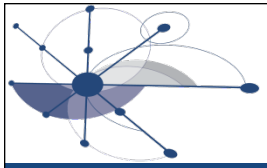
Optimization results

Benchmark	Optimization	Improvement
AMG2006	match data with computation	24% for solver
Sweep3D	change data layout to match access patterns	15%
LULESH	1. interleave data allocation 2. change data layout	13%
Streamcluster	interleave data allocation	28%
NW	interleave data allocation	53%



Overhead

Benchmark	Execution time	
	Native	With profiling
AMG2006	551s	604s (+9.6%)
Sweep3D	88s	90s (+2.3%)
LULESH	17s	19s (+12%)
Streamcluster	25s	27s (+8.0%)
NW	77s	80s (+3.9%)



Conclusion

- HPCToolkit capabilities
  - identify data locality bottlenecks
  - assess the impact of data locality bottlenecks
  - provide guidance for optimization
- HPCToolkit features
  - code-centric and data-centric analysis
  - widely applicable on modern architectures
  - work for MPI+thread programs
  - intuitive GUI for analyzing data locality bottlenecks
  - low overhead and high accuracy
- HPCToolkit utilities
  - identify CPU bound and memory bound programs
  - provide feedback to guide data locality optimization