Gaining Insight into Parallel Program Performance Using HPCToolkit

John Mellor-Crummey

Department of Computer Science
Rice University

Challenges for Application Developers

- **Rapidly evolving platforms and applications**
  - architecture
    - rapidly changing microprocessor designs
    - rise of accelerated computing
    - increasing scale of parallel systems
  - applications
    - transition from MPI everywhere to threaded implementations
    - augment computational capabilities
- **Application developer needs**
  - adapt to changes in emerging architectures
  - improve scalability within and across nodes
    - assess weaknesses in algorithms and their implementations

Performance tools can play an important role as a guide

Performance Analysis Challenges

- Complex node architectures are hard to use efficiently
  - multi-level parallelism: multiple cores, ILP, SIMD, accelerators
  - multi-level memory hierarchy
  - result: gap between typical and peak performance is huge
- Complex applications present challenges
  - measurement and analysis
  - understanding behaviors and tuning performance
- Multifaceted performance concerns
  - computation
  - data movement
  - communication
  - I/O
- Supercomputers compound the complexity
  - unique hardware & microkernel-based operating systems

What Users Want

- Multi-platform, programming model independent tools
- Accurate measurement of complex parallel codes
  - large, multi-lingual programs
  - (heterogeneous) parallelism within and across nodes
  - optimized code: loop optimization, templates, inlining
  - binary-only libraries, sometimes partially stripped
  - complex execution environments
    - dynamic binaries on clusters; static binaries on supercomputers
    - batch jobs
- Effective performance analysis
  - insightful analysis that pinpoints and explains problems
    - correlate measurements with code for actionable results
    - support analysis at the desired level
      - intuitive enough for application scientists and engineers
      - detailed enough for library developers and compiler writers
- Scalable to full systems

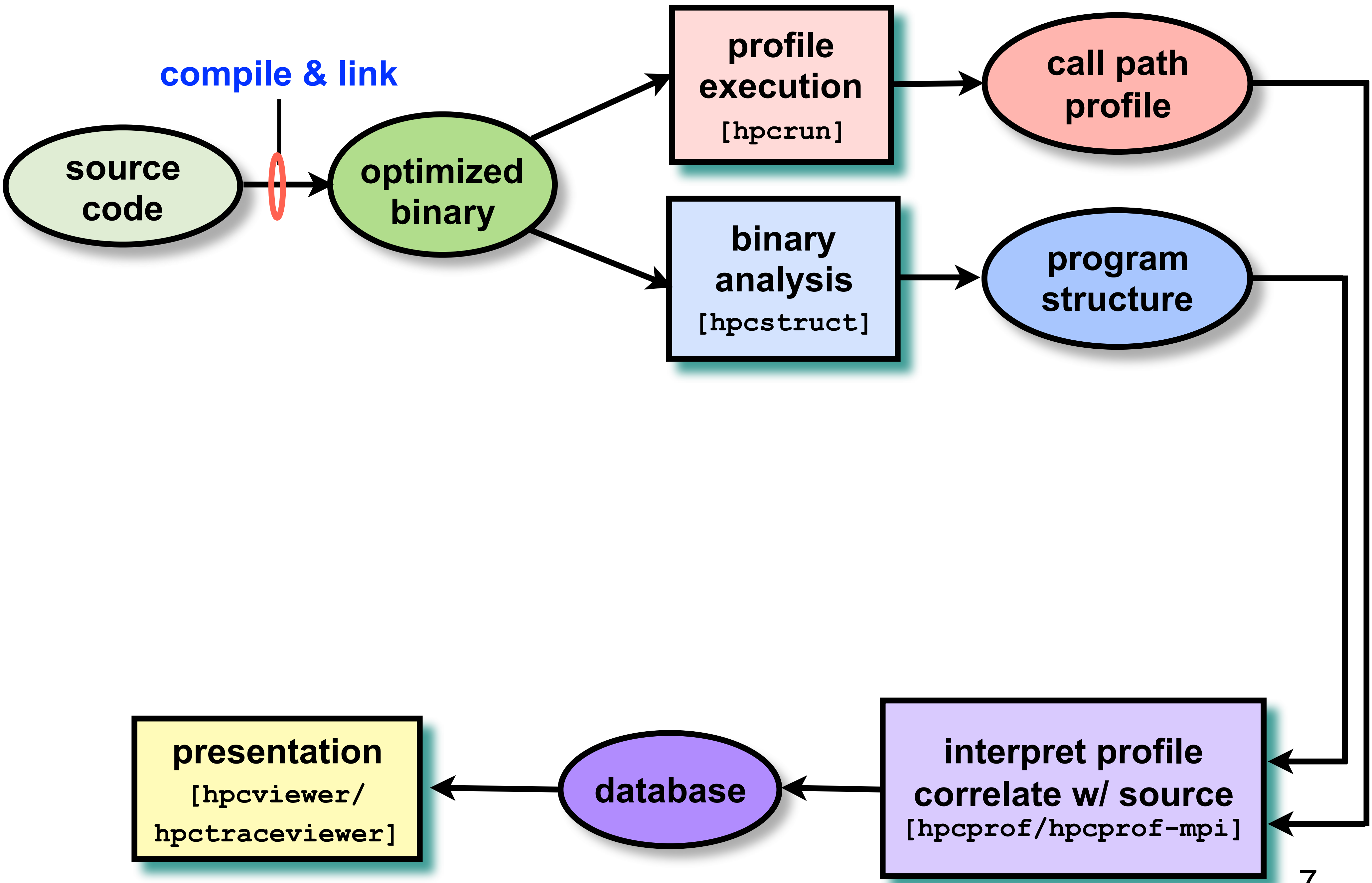
Rice University's HPCToolkit

- Employs binary-level measurement and analysis
  - observe fully optimized, dynamically linked executions
  - support multi-lingual codes with external binary-only libraries
- Uses sampling-based measurement (avoid instrumentation)
  - controllable overhead
  - minimize systematic error and avoid blind spots
  - enable data collection for large-scale parallelism
- Collects and correlates multiple derived performance metrics
  - diagnosis typically requires more than one species of metric
- Associates metrics with both static and dynamic context
  - loop nests, procedures, inlined code, calling context
- Supports top-down performance analysis
  - natural approach that minimizes burden on developers

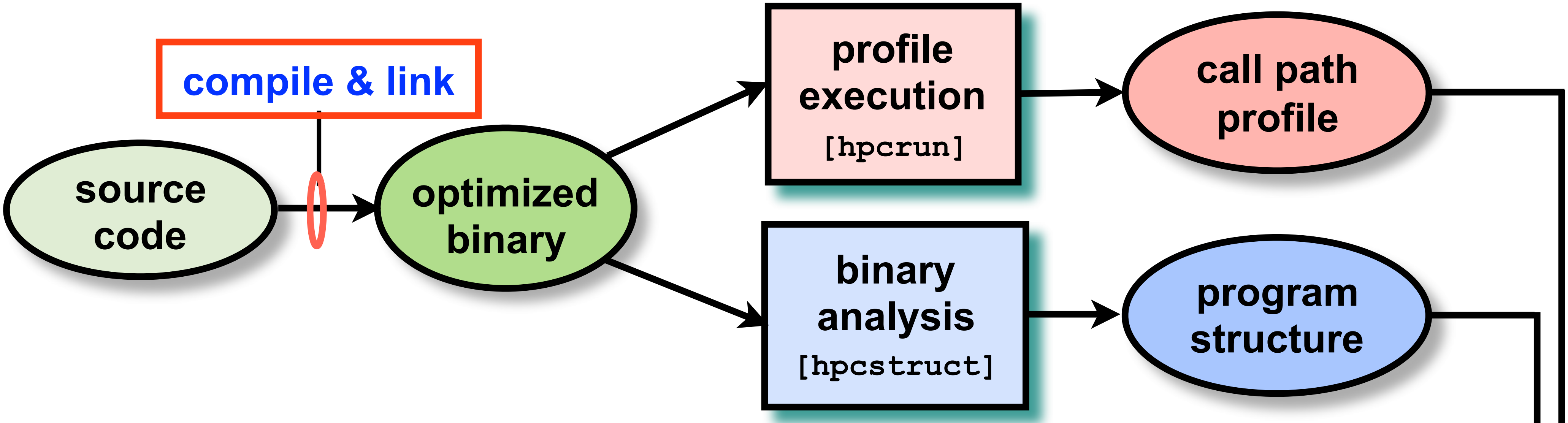
Outline

- **Overview of Rice's HPCToolkit**
- **Pinpointing scalability bottlenecks**
  - scalability bottlenecks on large-scale parallel systems
  - scaling on multicore processors
- **Understanding temporal behavior**
- **Assessing process variability**
- **Understanding threading and memory hierarchy**
  - blame shifting
  - attributing memory hierarchy costs to data
- **Today and the future**

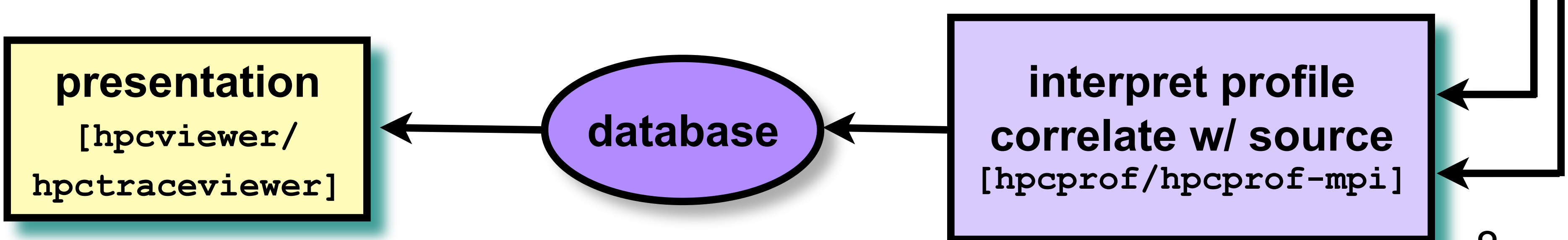
HPCToolkit Workflow



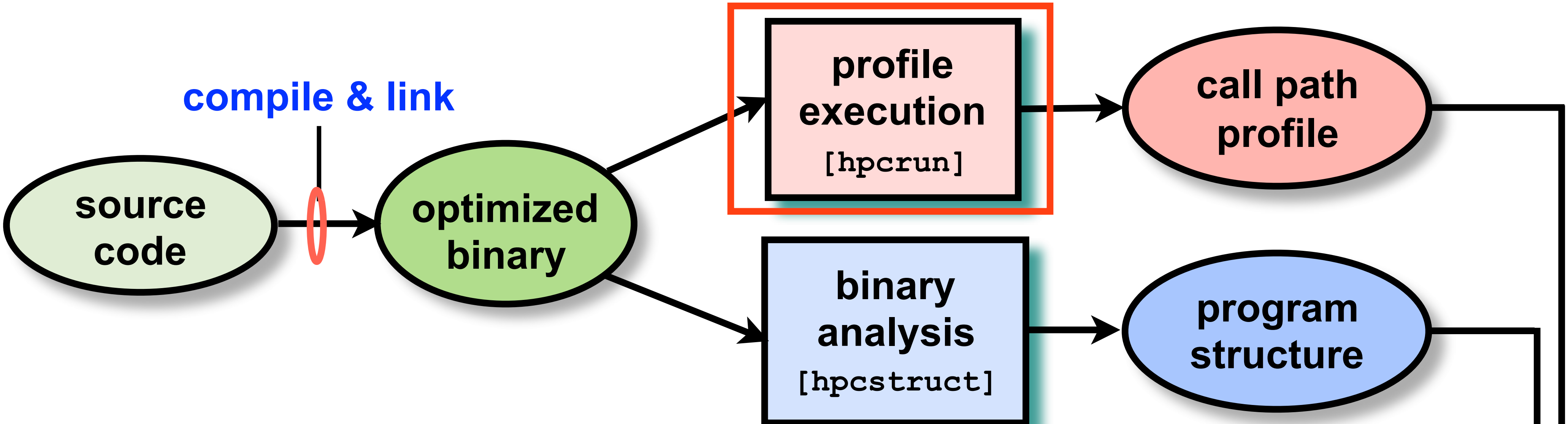
HPCToolkit Workflow



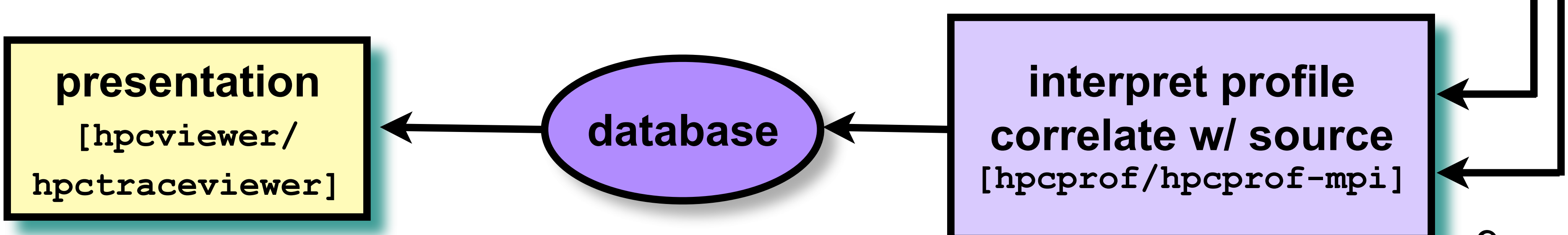
- For dynamically-linked executables on stock Linux
  — compile and link as you usually do
- For statically-linked executables (e.g., for Blue Gene, Cray)
  — add monitoring by using `hpclink` as prefix to your link line



HPCToolkit Workflow



- **Measure execution unobtrusively**
  - —launch optimized application binaries
    - –dynamically-linked applications: launch with hpcrun
      - e.g., mpirun -np 8192 hpcrun -t -e REALTIME@5000 flash3 ...
    - –statically-linked applications: control with environment variables
  - —collect statistical call path profiles of events of interest

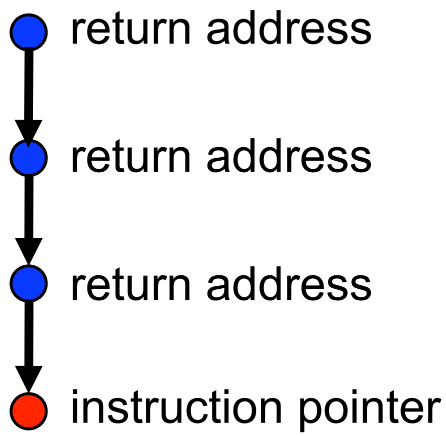


Call Path Profiling

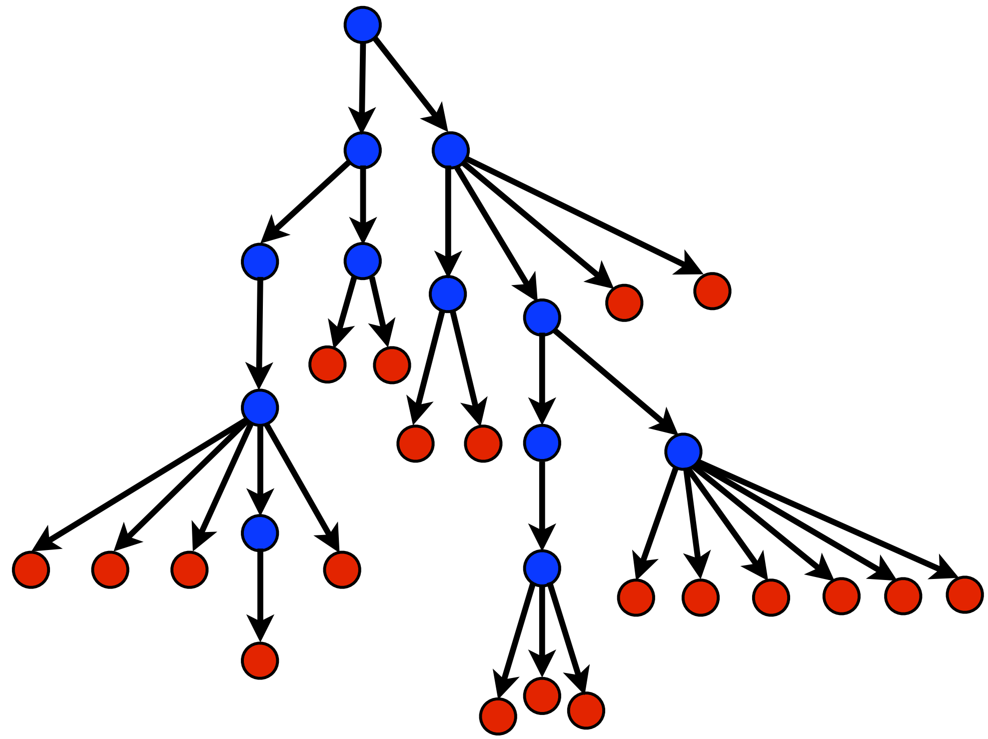
Measure and attribute costs in context

- **sample timer or hardware counter overflows**
- **gather calling context using stack unwinding**

Call path sample

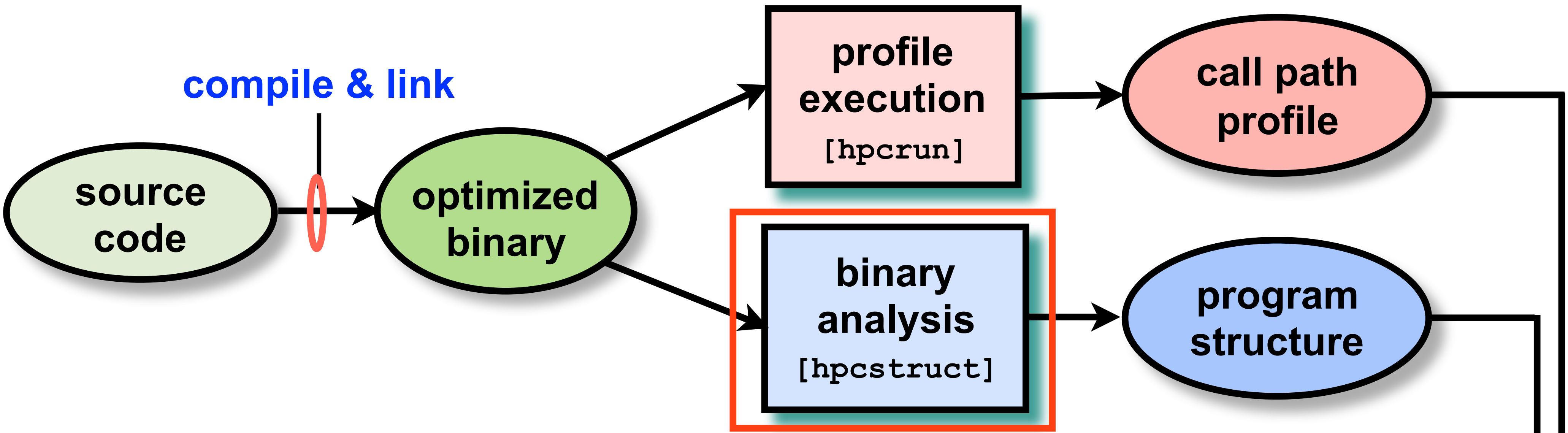


Calling context tree

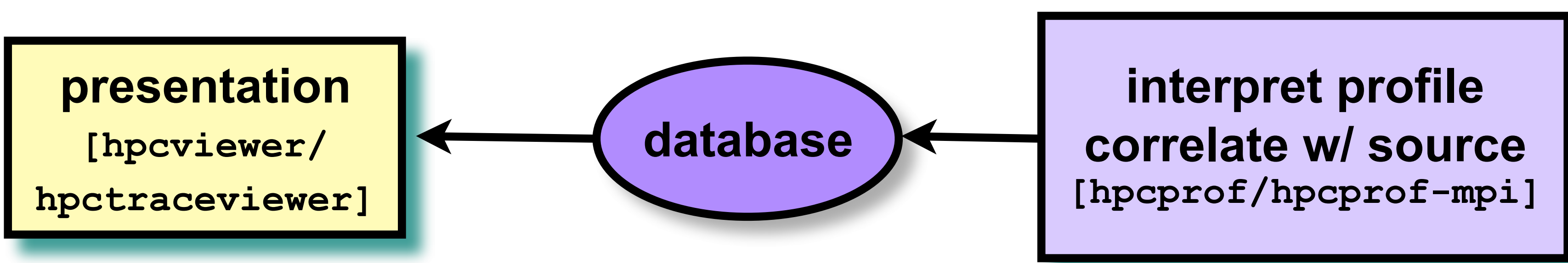


Overhead proportional to sampling frequency...
...not call frequency

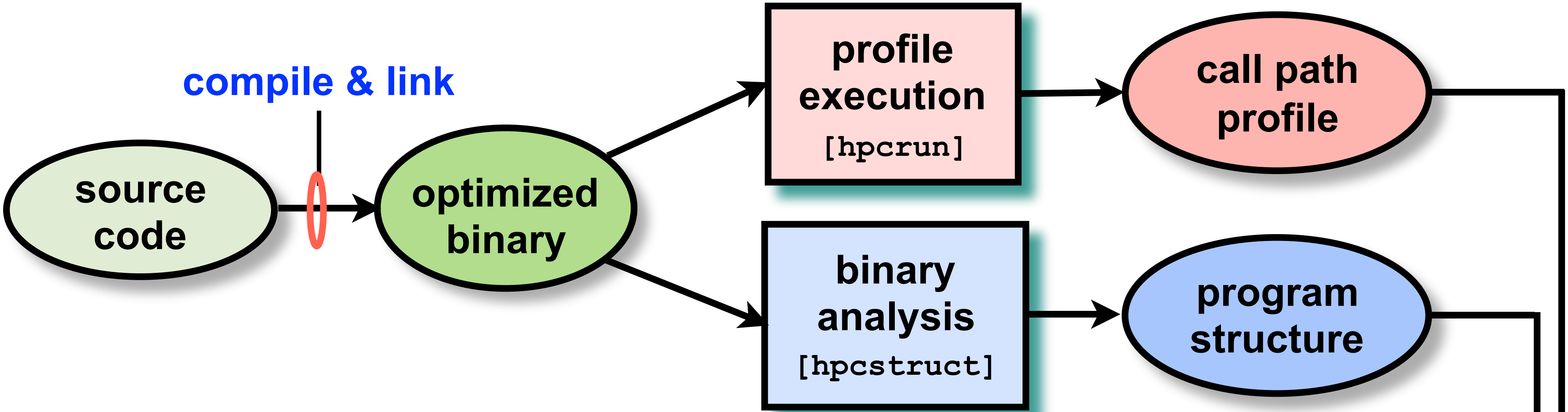
HPCToolkit Workflow



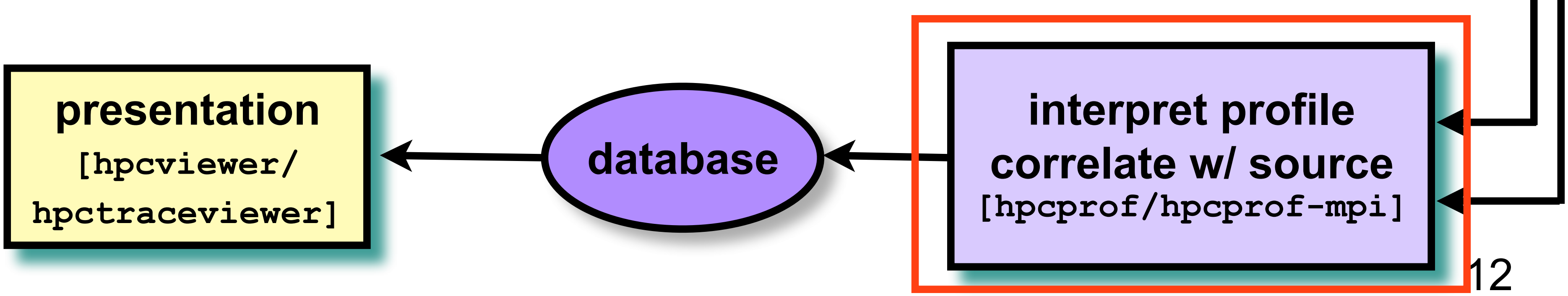
- **Analyze binary with `hpcstruct`: recover program structure**
  - —analyze machine code, line map, debugging information
  - —extract loop nesting & identify inlined procedures
  - —map transformed loops and procedures to source



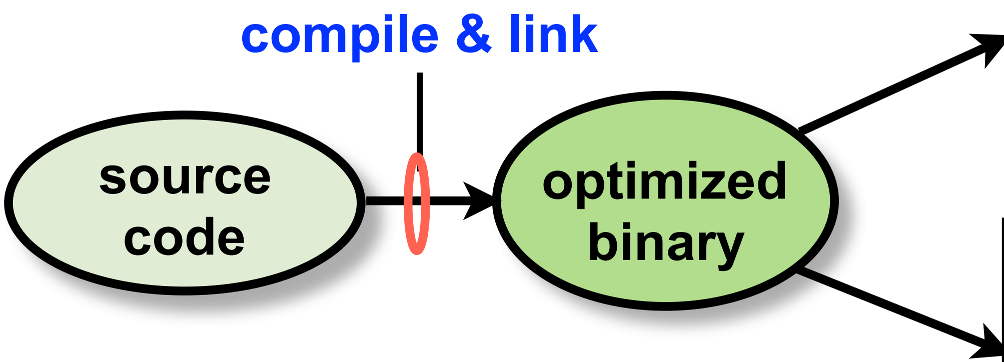
HPCToolkit Workflow

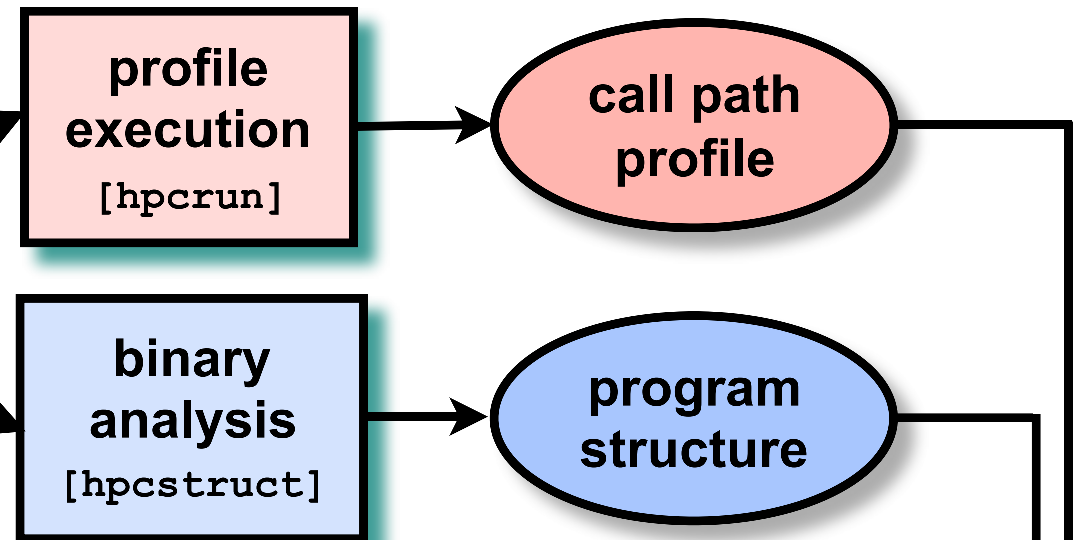


- **Combine multiple profiles**
  —multiple threads; multiple processes; multiple executions
- **Correlate metrics to static & dynamic program structure**



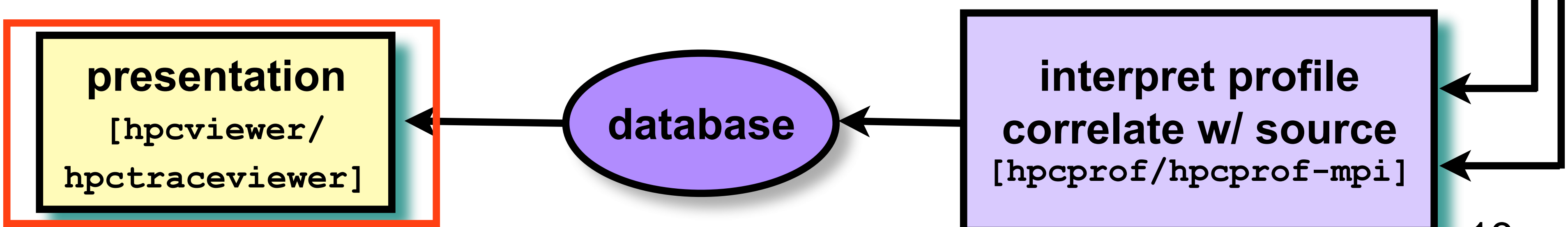
HPCToolkit Workflow



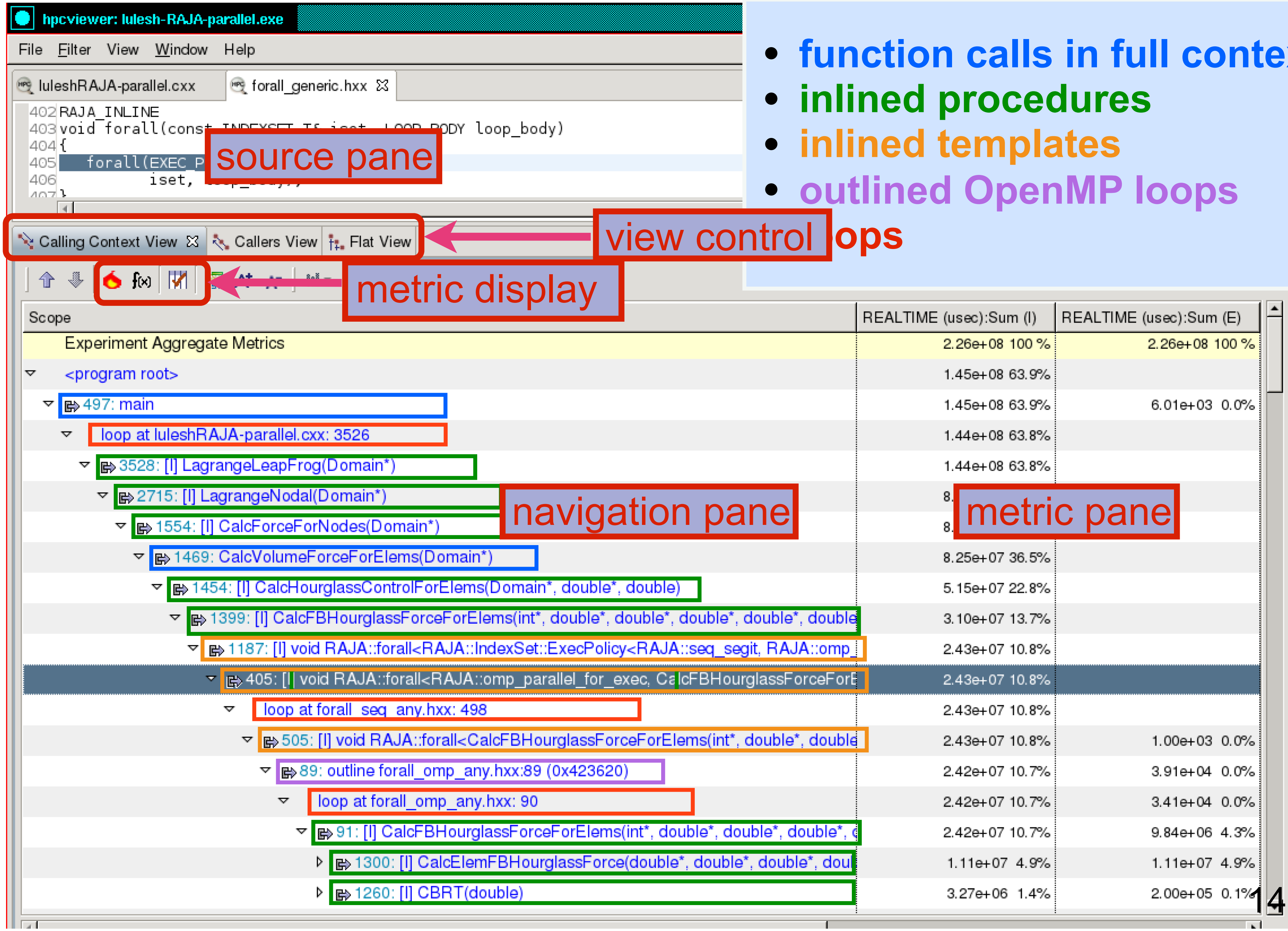


- **Presentation**
  - —explore performance data from multiple perspectives
    - –rank order by metrics to focus on what's important
    - –compute derived metrics to help gain insight

      e.g. scalability losses, waste, CPI, bandwidth
  - —graph thread-level metrics for contexts
  - —explore evolution of behavior over time



Code-centric Analysis with hpcviewer

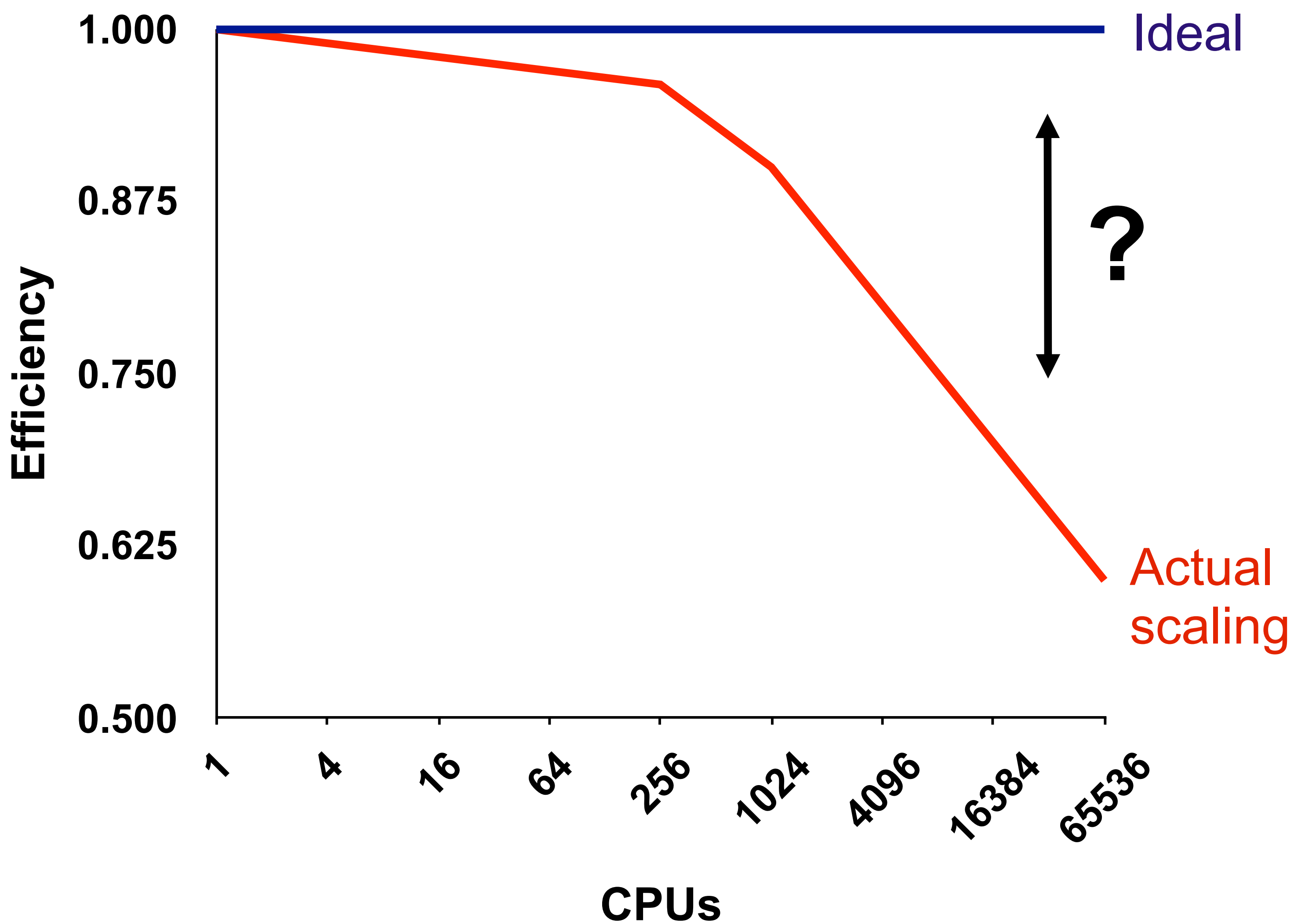


- function calls in full context
- inlined procedures
- inlined templates
- outlined OpenMP loops
- ops

Outline

- Overview of Rice's HPCToolkit
- **Pinpointing scalability bottlenecks**
  - — scalability bottlenecks on large-scale parallel systems
  - — scaling on multicore processors
- Understanding temporal behavior
- Assessing process variability
- Understanding threading and memory hierarchy
  - — blame shifting
  - — attributing memory hierarchy costs to data
- Today and the future

The Problem of Scaling



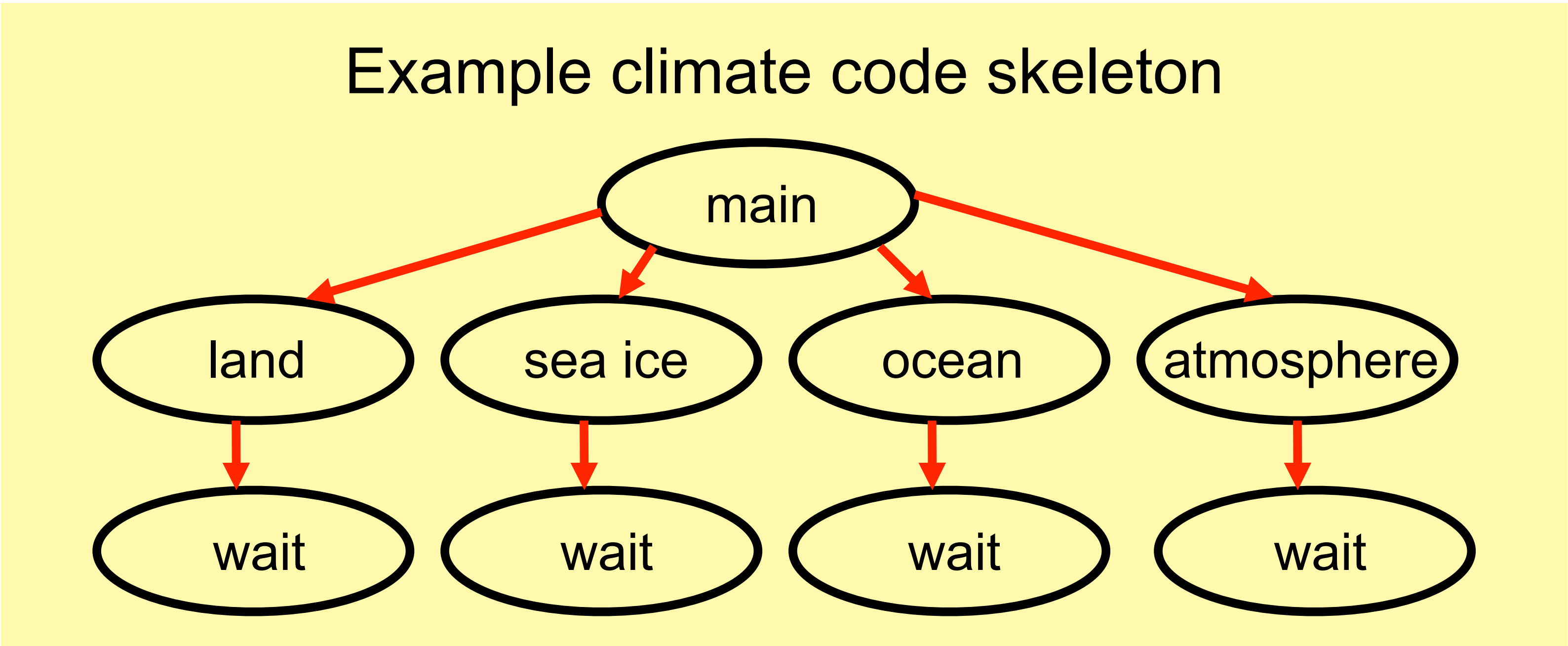
Note: higher is better

Wanted: Scalability Analysis

- Isolate scalability bottlenecks
- Guide user to problems
- Quantify the magnitude of each problem

Challenges for Pinpointing Scalability Bottlenecks

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



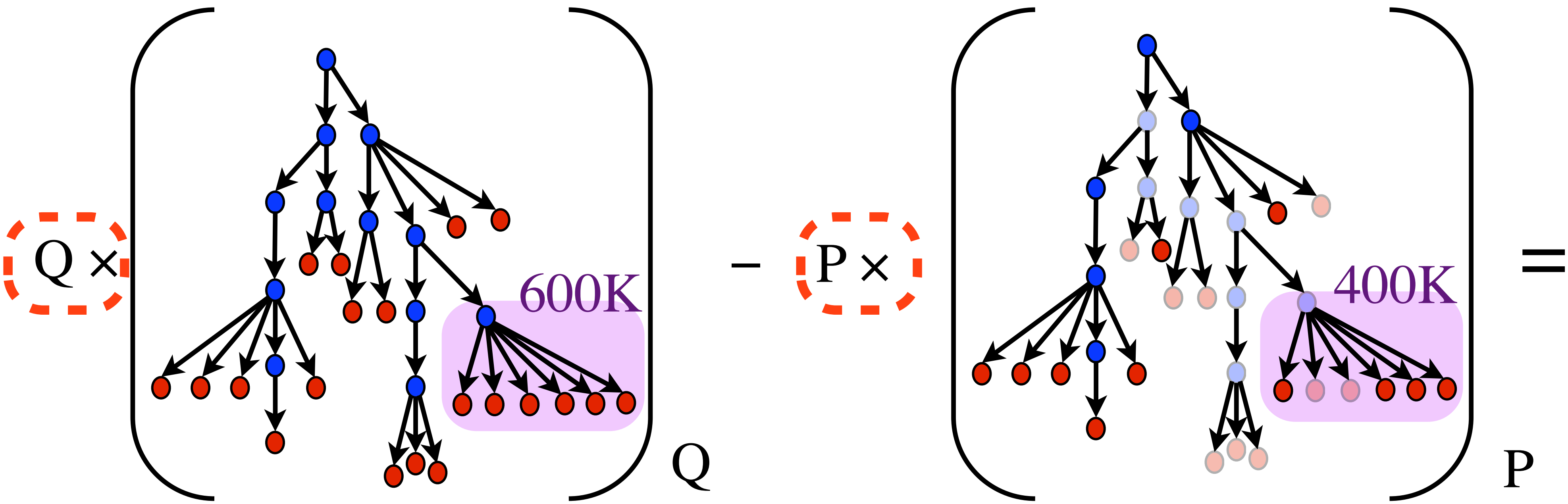
- **Monitoring**
  - **bottleneck nature: computation, data movement, synchronization?**
  - **2 pragmatic constraints**
    - **acceptable data volume**
    - **low perturbation for use in production runs**

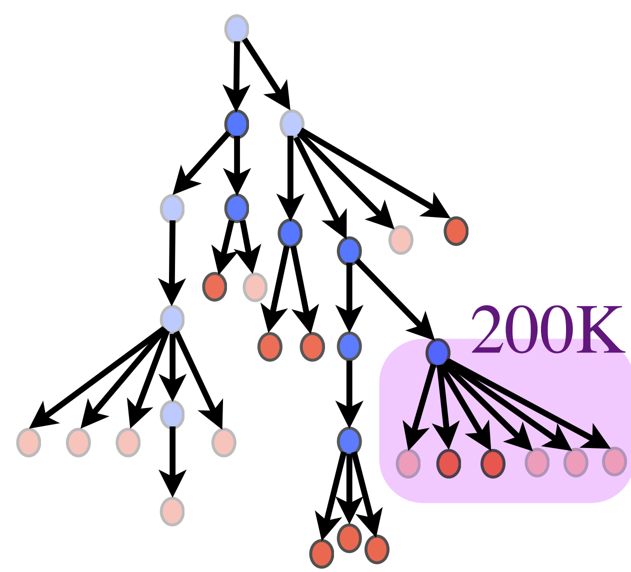
Performance Analysis with Expectations

- You have performance expectations for your parallel code
  - strong scaling: linear speedup
  - weak scaling: constant execution time

- Put your expectations to work
  - measure performance under different conditions
    - e.g. different levels of parallelism and/or different problem size
  - express your expectations as an equation
  - compute the deviation from expectations for each calling context
    - for both inclusive and exclusive costs
  - correlate the metrics with the source code
  - explore the annotated call tree interactively

Pinpointing and Quantifying Scalability Bottlenecks

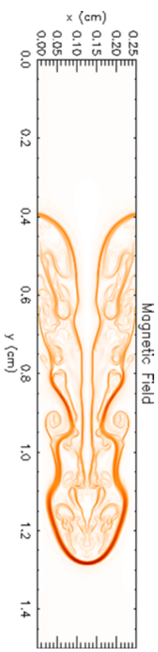




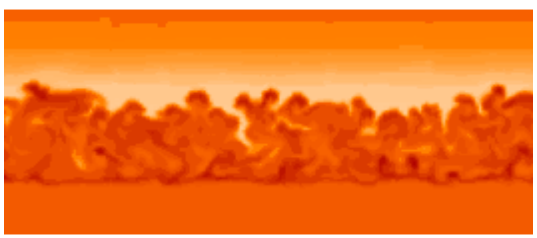
coefficients for analysis of strong scaling

Scalability Analysis Demo: FLASH3

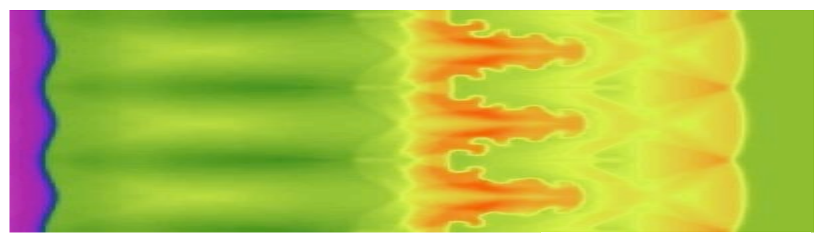
Code:	**University of Chicago FLASH3**
Simulation:	**white dwarf detonation**
Platform:	**Blue Gene/P**
Experiment:	**8192 vs. 256 cores**
Scaling type:	**weak**



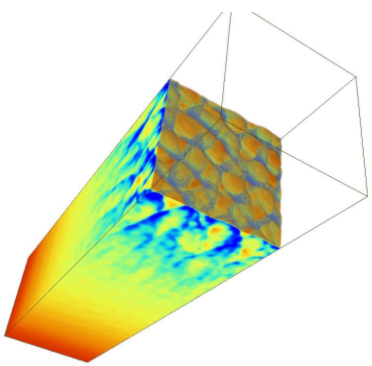
Magnetic Rayleigh-Taylor



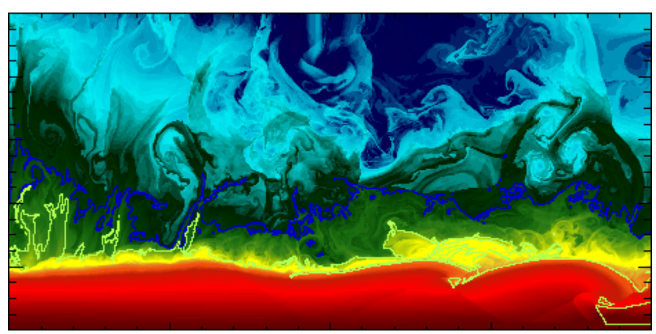
Nova outbursts on white dwarfs



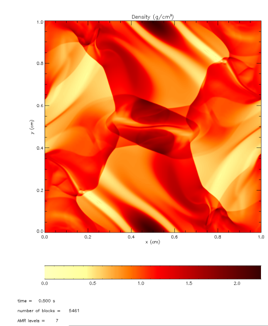
Laser-driven shock instabilities



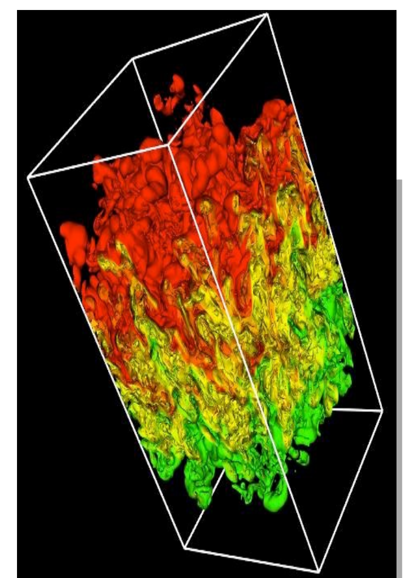
Cellular detonation



Helium burning on neutron stars



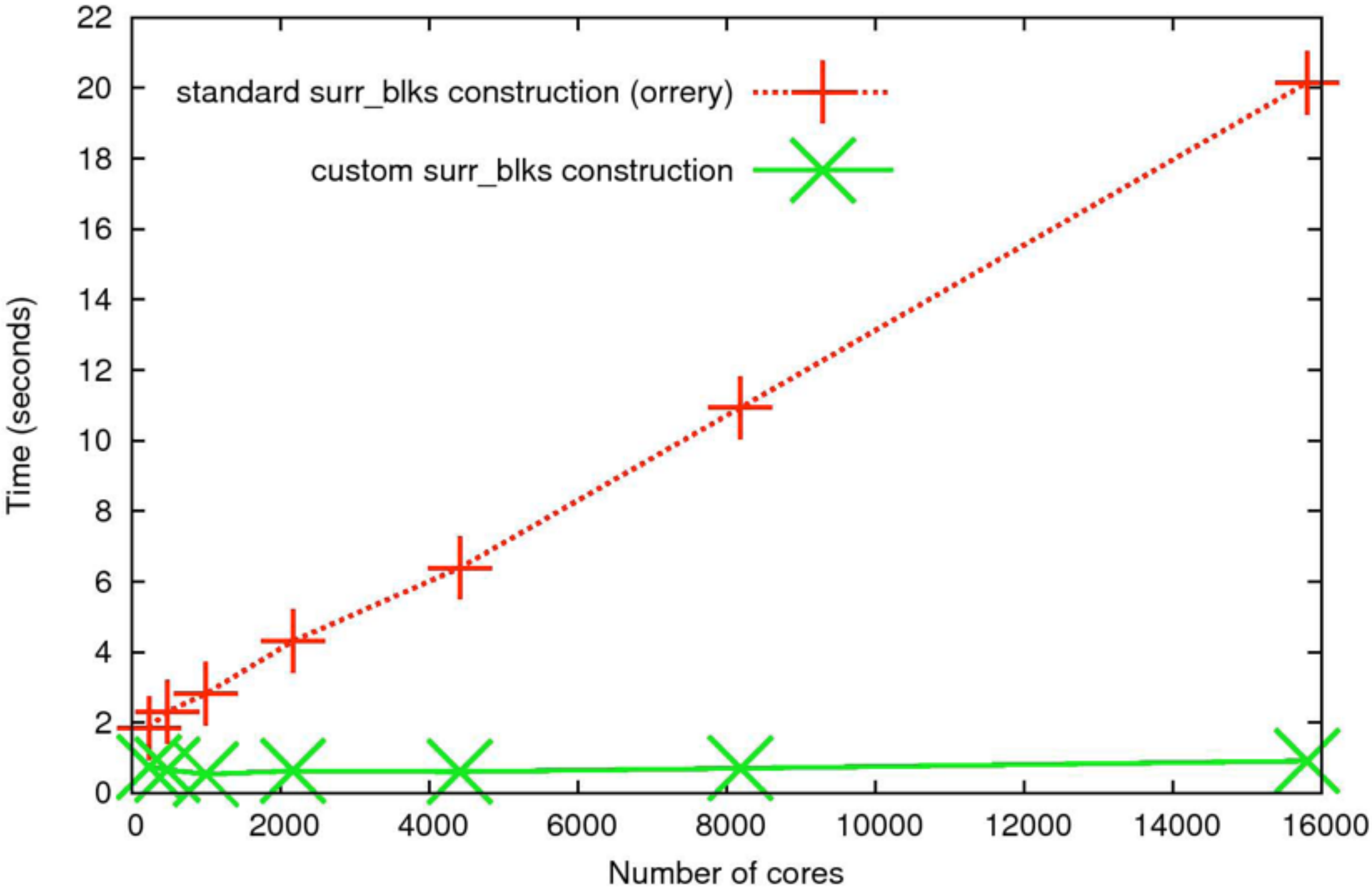
Orzag/Tang MHD vortex



Rayleigh-Taylor instability

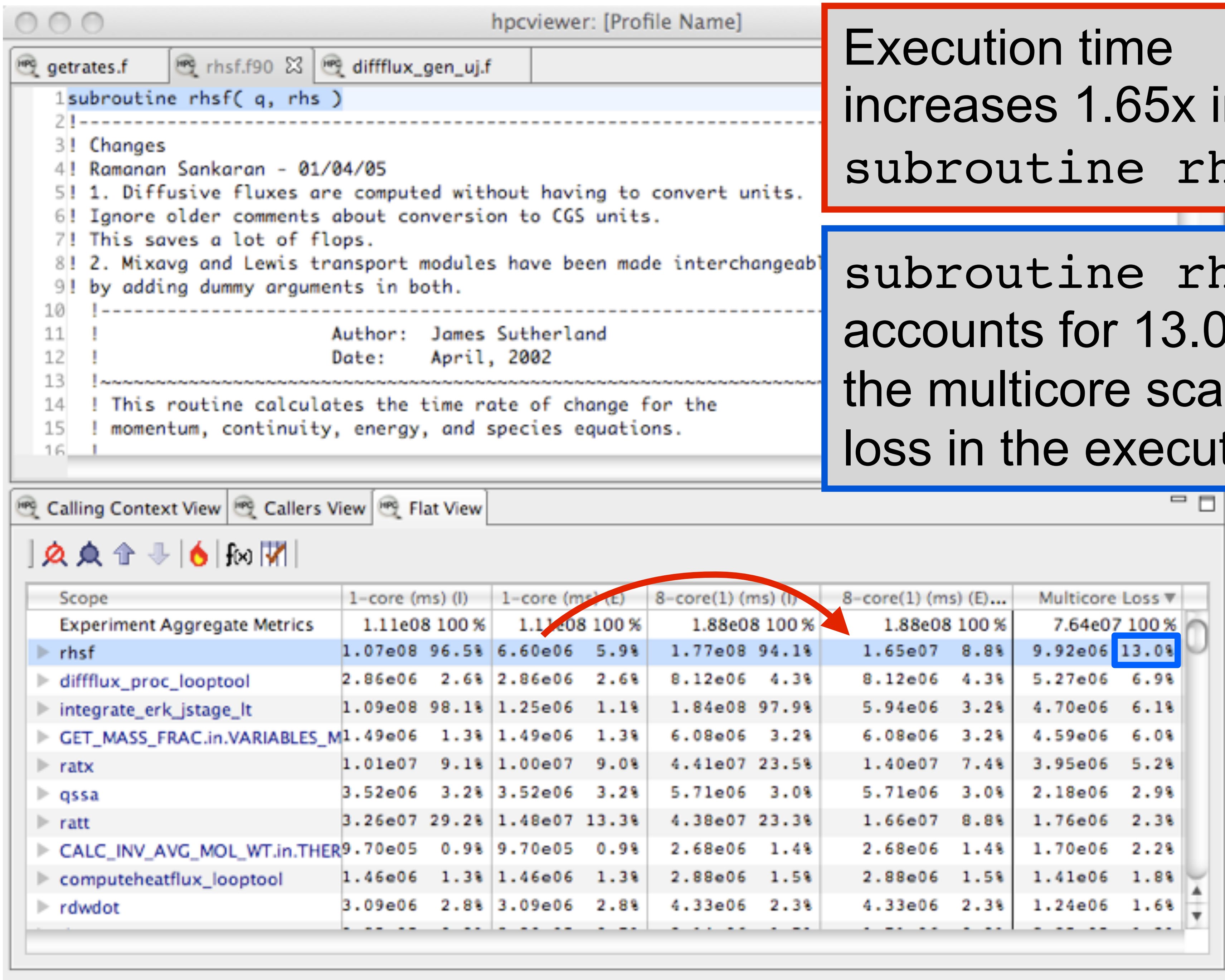
Figures courtesy of FLASH Team, University of Chicago

Improved Flash Scaling of AMR Setup



Graph courtesy of Anshu Dubey, U 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.
! Ignore older comments about conversion to CGS units.
! This saves a lot of flops.
! 2. Mixavg and Lewis transport modules have been made interchangeabl
! 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.
  !
```

Scope	1-core (ms) (I)		1-core (ms) (E)		8-core(1) (ms) (I)		8-core(1) (ms) (E)...		Multicore Loss ▼	
Experiment Aggregate Metrics	1.11e08	100 %	1.11e08	100 %	1.88e08	100 %	1.88e08	100 %	7.64e07	100 %
rhsf	1.07e08	96.5%	6.60e06	5.9%	1.77e08	94.1%	1.65e07	8.8%	9.92e06	13.0%
diffflux_proc_looptool	2.86e06	2.6%	2.86e06	2.6%	8.12e06	4.3%	8.12e06	4.3%	5.27e06	6.9%
integrate_erk_jstage_lt	1.09e08	98.1%	1.25e06	1.1%	1.84e08	97.9%	5.94e06	3.2%	4.70e06	6.1%
GET_MASS_FRAC.in.VARIABLES_M	1.49e06	1.3%	1.49e06	1.3%	6.08e06	3.2%	6.08e06	3.2%	4.59e06	6.0%
ratx	1.01e07	9.1%	1.00e07	9.0%	4.41e07	23.5%	1.40e07	7.4%	3.95e06	5.2%
qssa	3.52e06	3.2%	3.52e06	3.2%	5.71e06	3.0%	5.71e06	3.0%	2.18e06	2.9%
ratt	3.26e07	29.2%	1.48e07	13.3%	4.38e07	23.3%	1.66e07	8.8%	1.76e06	2.3%
CALC_INV_AVG_MOL_WT.in.THER	9.70e05	0.9%	9.70e05	0.9%	2.68e06	1.4%	2.68e06	1.4%	1.70e06	2.2%
computeheatflux_looptool	1.46e06	1.3%	1.46e06	1.3%	2.88e06	1.5%	2.88e06	1.5%	1.41e06	1.8%
rdwdot	3.09e06	2.8%	3.09e06	2.8%	4.33e06	2.3%	4.33e06	2.3%	1.24e06	1.6%

Execution time increases 1.65x in `subroutine rhsf`

`subroutine rhsf` accounts for 13.0% of the multicore scaling loss in the execution

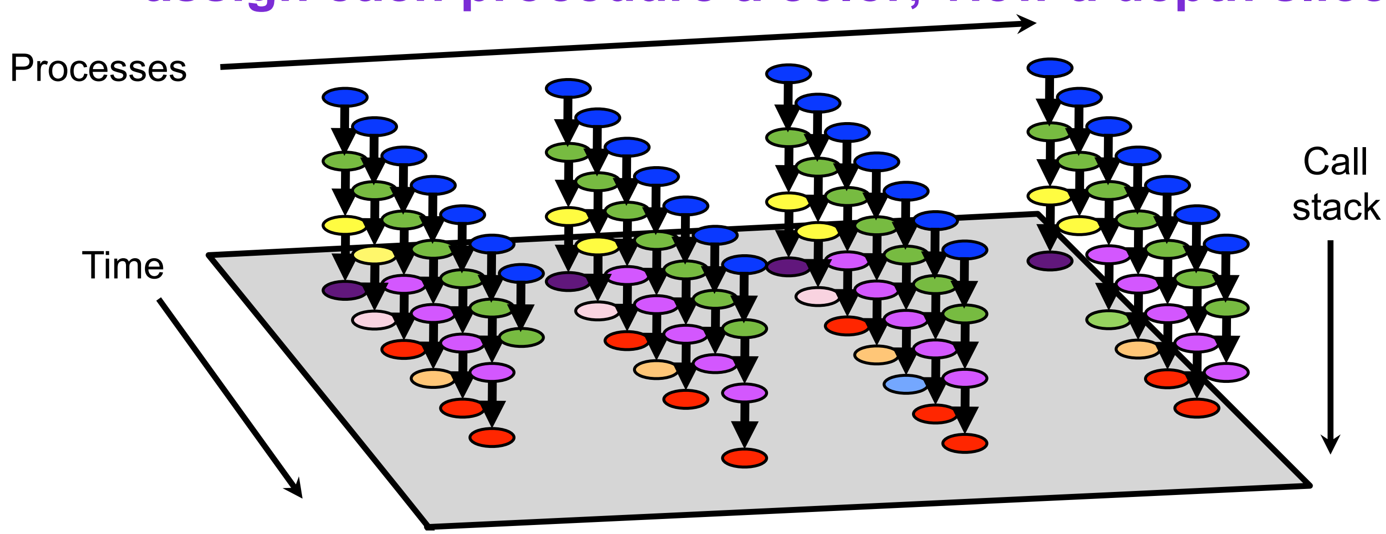
Outline

- Overview of Rice's HPCToolkit
- Pinpointing scalability bottlenecks
  — scalability bottlenecks on large-scale parallel systems
  — scaling on multicore processors
- Understanding temporal behavior
- Assessing process variability
- Understanding threading and memory hierarchy
  — blame shifting
  — attributing memory hierarchy costs to data
- Today and the future

Understanding Temporal Behavior

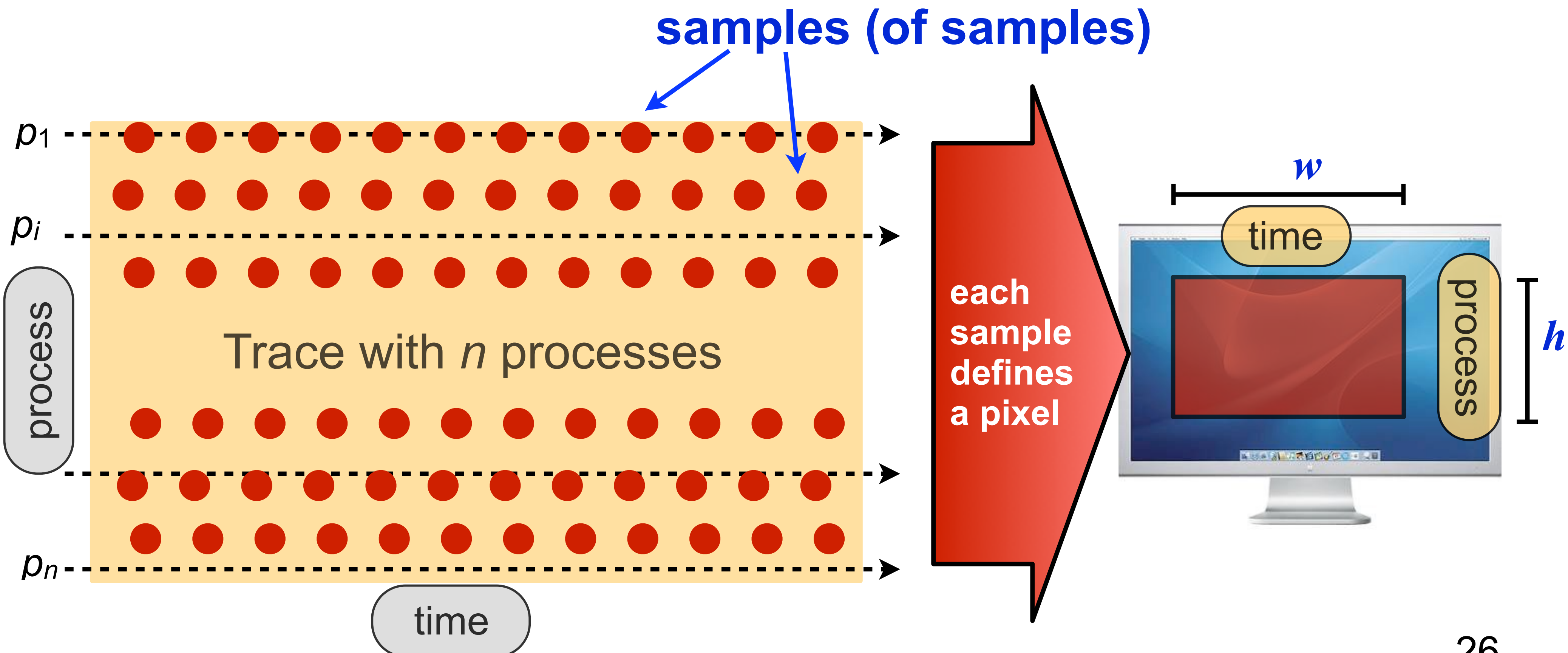
- **Profiling compresses out the temporal dimension**
  - —temporal patterns, e.g. serialization, are invisible in profiles
- **What can we do? Trace call path samples**
  - —sketch:
    - – N times per second, take a call path sample of each thread
    - – organize the samples for each thread along 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



Presenting Large Traces on Small Displays

- **How to render an arbitrary portion of an arbitrarily large trace?**
  - **— we have a display window of dimensions h × w**
  - **— typically many more processes (or threads) than h**
  - **— typically many more samples (trace records) than w**
- **Solution: sample the samples!**

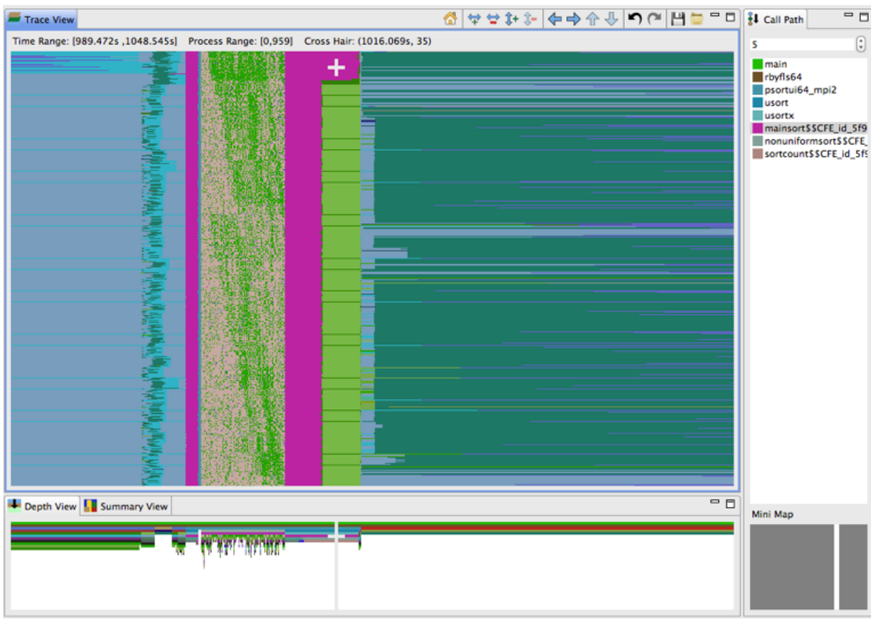


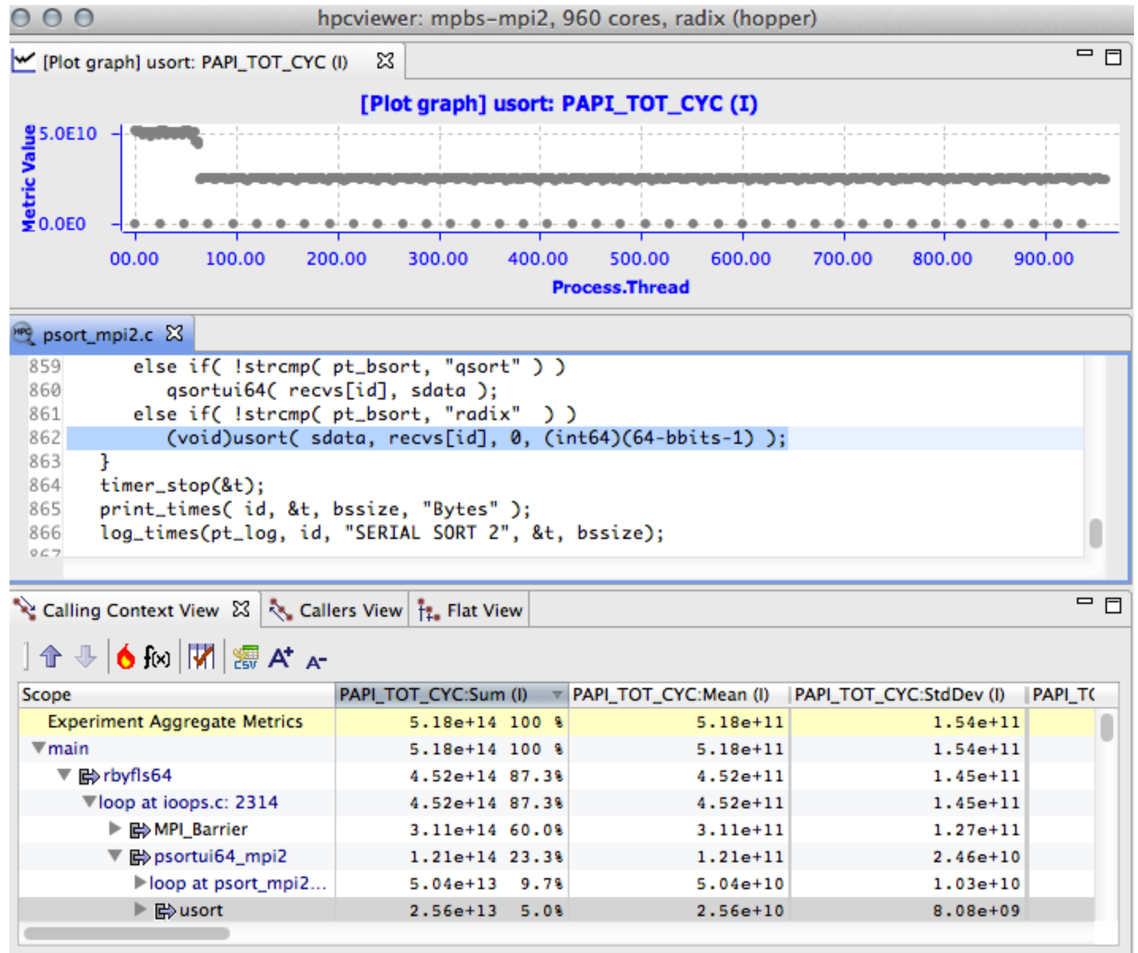
Outline

- Overview of Rice's HPCToolkit
- Pinpointing scalability bottlenecks
  - scalability bottlenecks on large-scale parallel systems
  - scaling on multicore processors
- Understanding temporal behavior
- **Assessing process variability**
- Understanding threading and memory hierarchy
  - blame shifting
  - attributing memory hierarchy costs to data
- Today and the future

MPBS @ 960 cores, radix sort

Two views of load imbalance since not on a 2^k cores



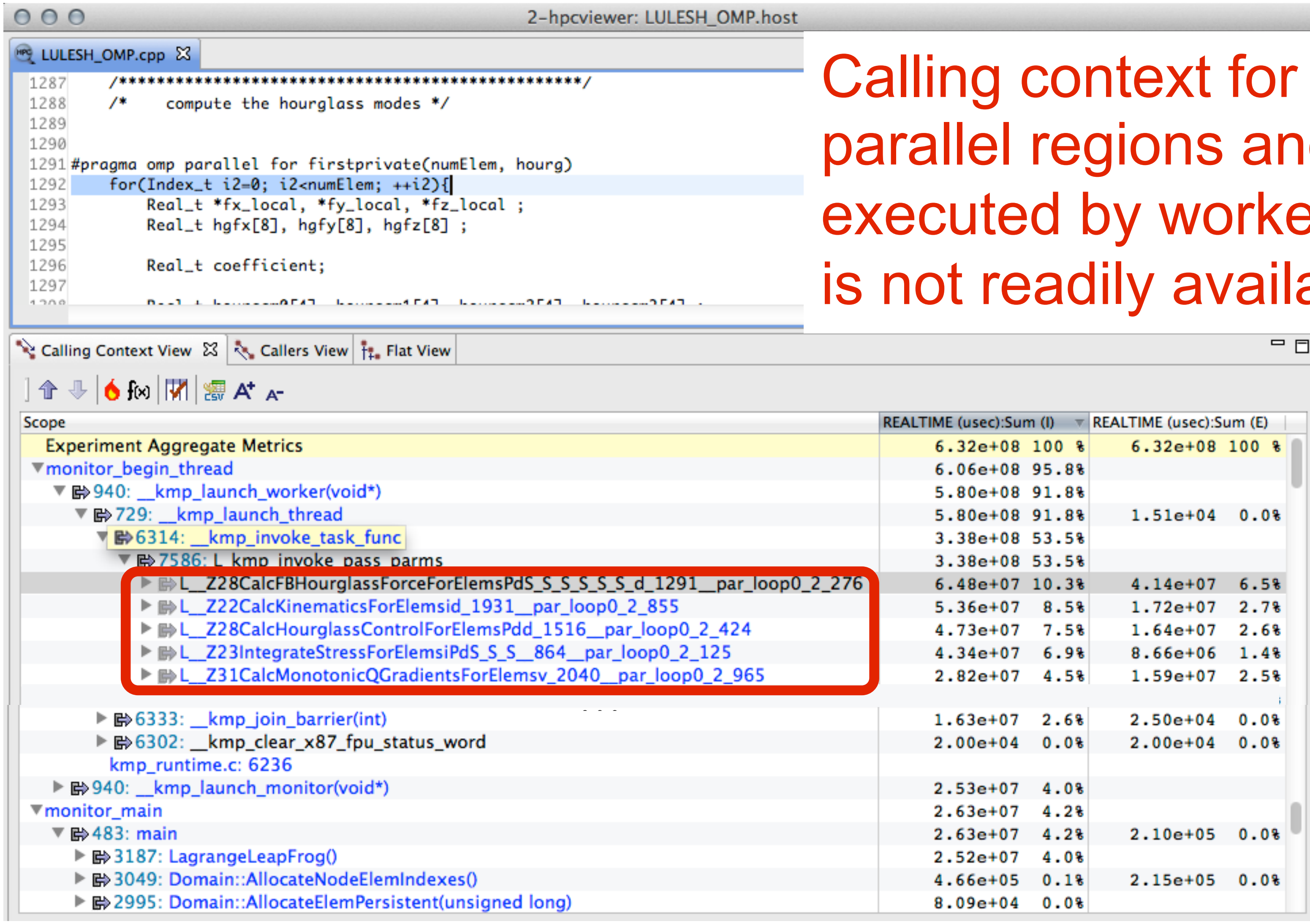


Outline

- Overview of Rice's HPCToolkit
- Pinpointing scalability bottlenecks
  - scalability bottlenecks on large-scale parallel systems
  - scaling on multicore processors
- Understanding temporal behavior
- Assessing process variability
- Understanding threading and memory hierarchy
  - blame shifting
  - attributing memory hierarchy costs to data
- Today and the future

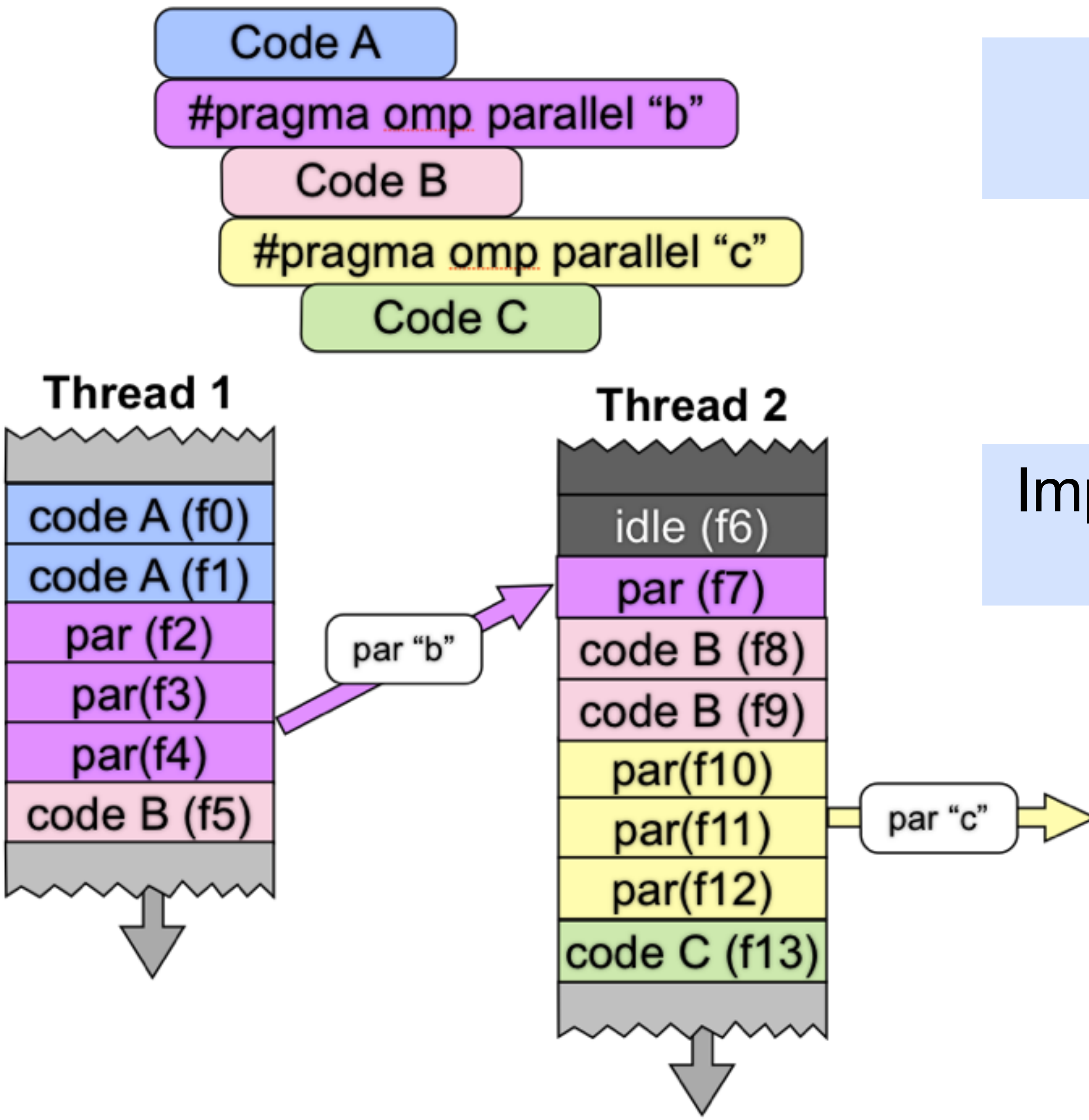
OpenMP: Gap between Source and Implementation

Calling context for code in parallel regions and tasks executed by worker threads is not readily available



Tools must bridge this gap to explain program performance

OpenMP Application-level Context is Distributed



Developer view

Implementation view

OMPT Design Objectives

- Enable tools to gather information and associate costs with application source and runtime system
  - —construct low-overhead tools based on asynchronous sampling
  - —identify application stack frames vs. runtime frames
  - —associate a thread's activity at any point with a descriptive state
    - – parallel work, idle, lock wait, ...
- Negligible overhead if OMPT interface is not in use
  - —features that may increase overhead are optional
- Define support for trace-based performance tools
- Don't impose an unreasonable development burden
  - —runtime implementers
  - —tool developers

Principal OMPT Features

- **State tracking**
  - — have runtime track keep track of thread states

work (serial, parallel)	task wait
idle	mutual exclusion
barrier	overhead

  - — async signal safe query
    - – if in a waiting state, handle identifies what is being awaited
- **Call stack interpretation**
  - — provide hooks that enable tools to reconstruct application-level call stacks from implementation-level information
- **Event notification**
  - — provide callbacks for predefined events
  - – few mandatory notifications
  - – optional events for blame shifting, tracing

OMPT Mandatory Events

Essential support for any performance tool

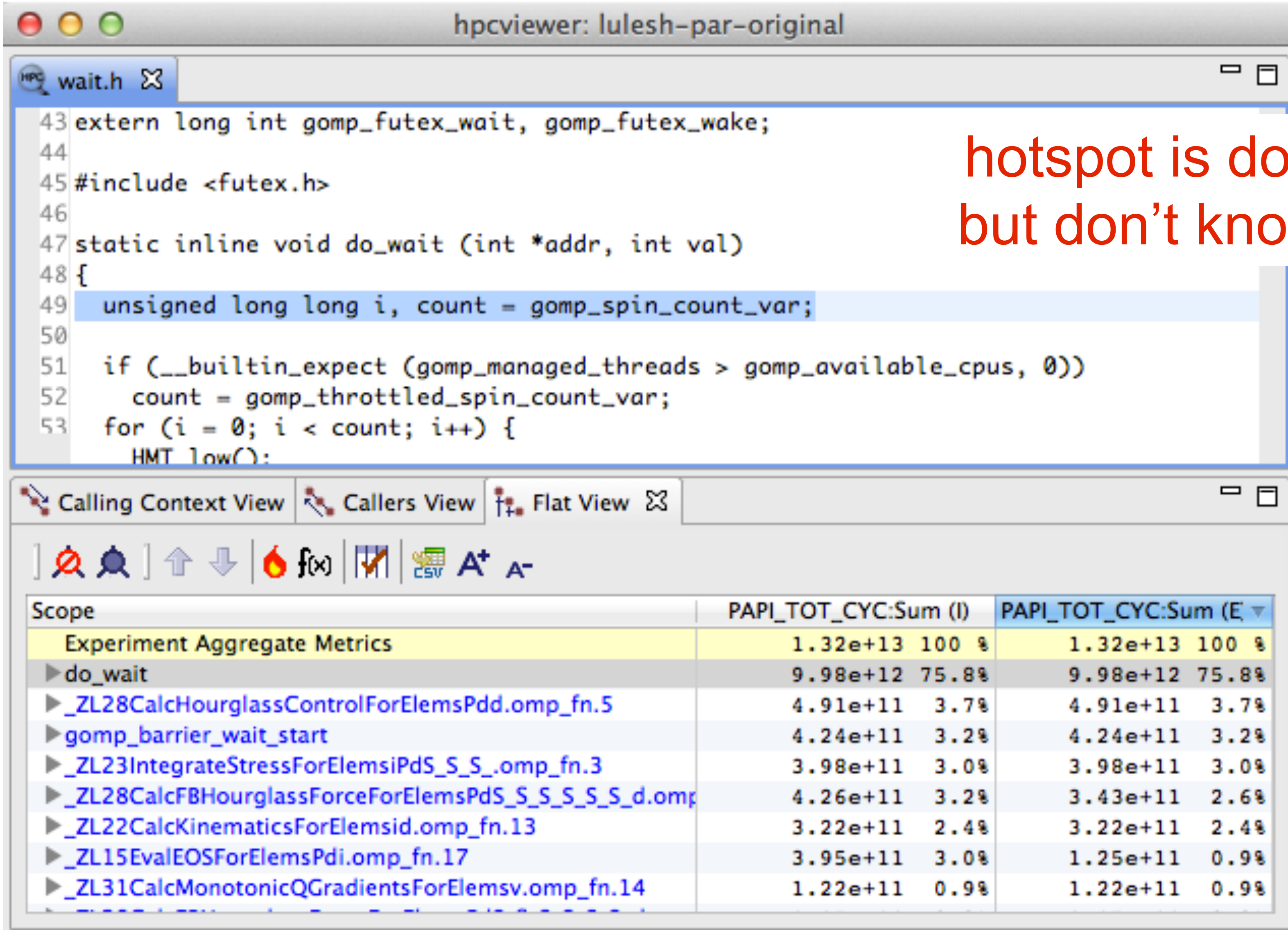
- **Threads**
- **Parallel regions**
- **Tasks**

create/exit event pairs

- **Runtime shutdown**
- **User-level control API**
  - **e.g., support tool start/stop**

singleton events

OpenMP: Meaningless Hotspots



hotspot is do_wait,
but don't know why

Identifying Causes of Bottlenecks

- **Problem: in many circumstances sampling measures symptoms of performance losses rather than causes**
  - worker threads waiting for work
  - threads waiting for a lock
  - process waiting for peers to arrive at a barrier
  - idle GPU waiting for work
- **Approach: shift blame for losses from victims to perpetrators**
  - blame code executing while other threads are idle
  - blame code executed by lock holder when thread(s) are waiting
  - blame processes that arrive late to collectives
  - shift blame between CPU and GPU for hybrid code

Blame Shifting from Symptoms to Causes

- Approach
  - — shift blame for idleness to code executing while other threads are idle
- Implementation of undirected blame shifting
  - — callback at thread transitions idle ↔ working
  - — maintain two global counters
    - – thread created (or dedicated HW resources that are reserved)
    - – number of threads that are working
    - – idleness is the difference between the two counters
  - — at a sample event
    - – if the thread is actively working
      - attribute a sample of work to the present context
      - attribute partial blame for idleness to the present context
    - – else, ignore the sample event

Blame-shifting for Analyzing Thread Performance

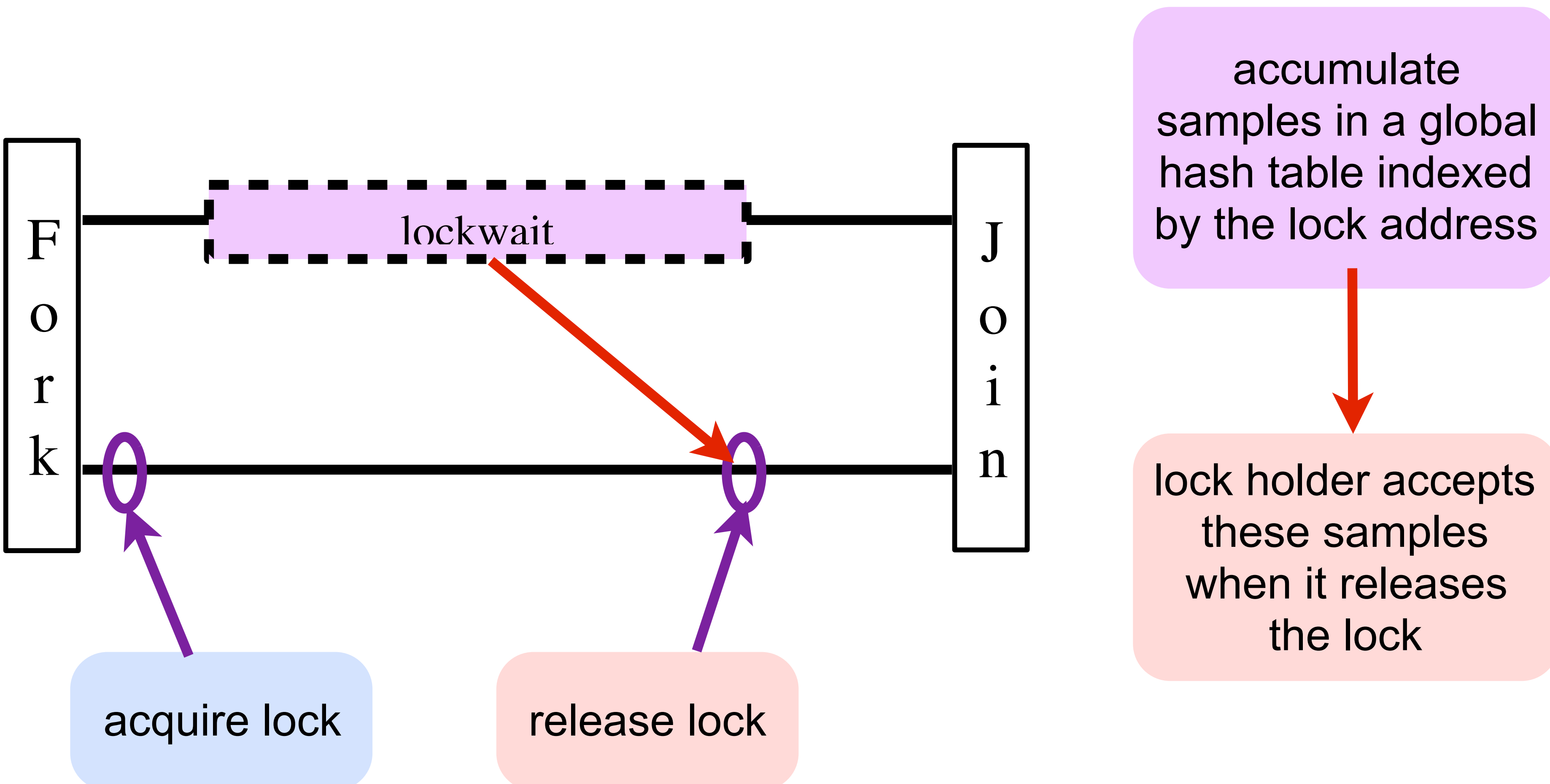
	Problem	Approach
Undirected Blame Shifting[1]	A thread is idle waiting for work	**Apportion blame among working threads** for not shedding enough parallelism to keep all threads busy
Directed Blame Shifting[2]	A thread is idle waiting for a mutex	**Blame the thread holding the mutex** for idleness of threads waiting for the mutex

[1]Tallent & Mellor-Crummey: PPoPP 2009

[2]Tallent, Mellor-Crummey, Porterfield: PPoPP 2010

Directed Blame Shifting

- **Example:**
  - **threads waiting at a lock are the symptom**
  - **the cause is the lock holder**
- **Approach: blame lock waiting on lock holder**



OMPT Events for Blame-shifting

Support designed for sampling-based performance tools

- **Idle**
- **Wait**
  - barrier
  - taskwait
  - taskgroup wait

begin/end event pairs

- **Release**
  - lock
  - nest lock
  - critical
  - atomic
  - ordered section

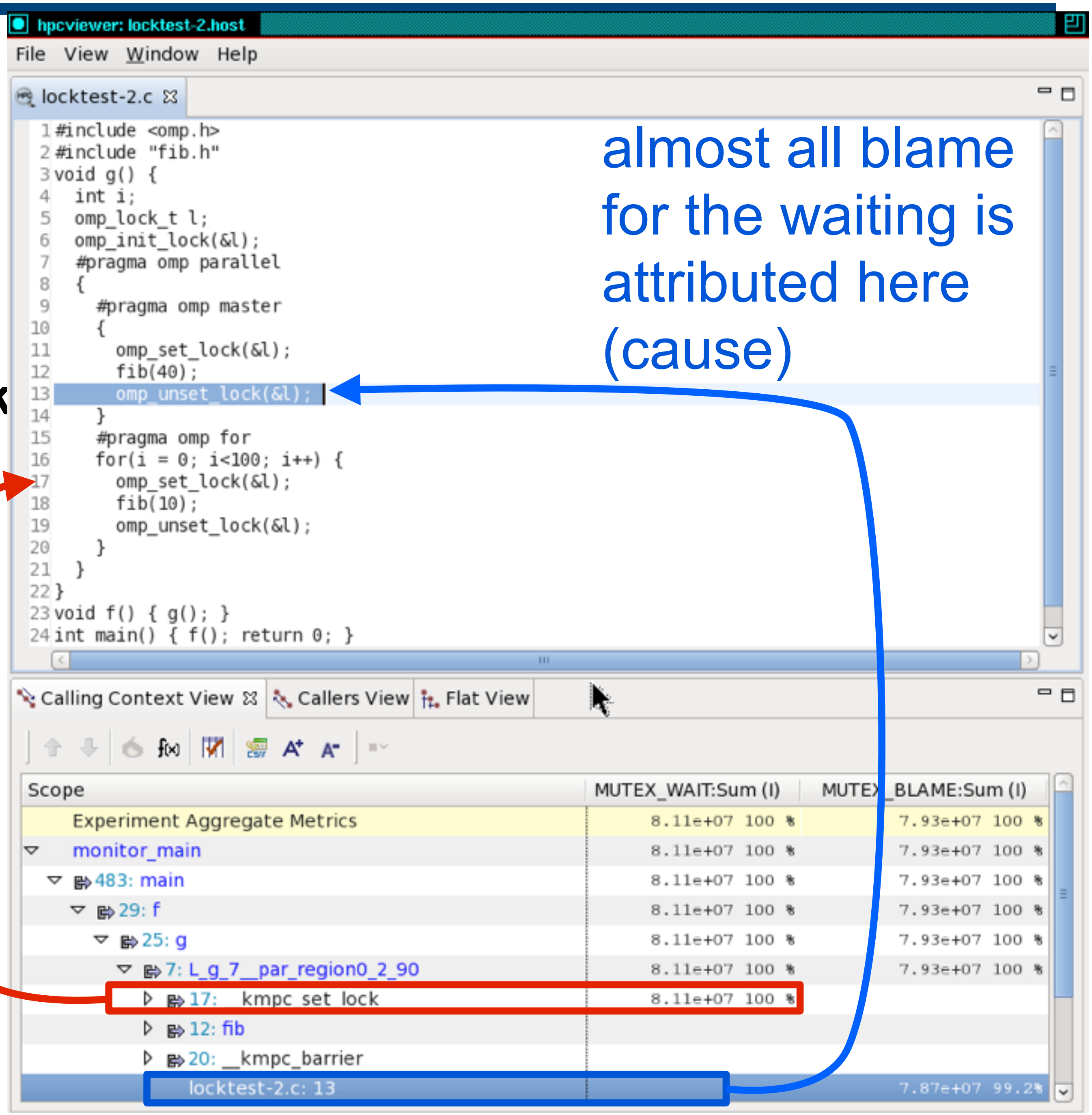
singleton events

Example: Directed Blame Shifting for Locks

Blame a lock holder for delaying waiting threads

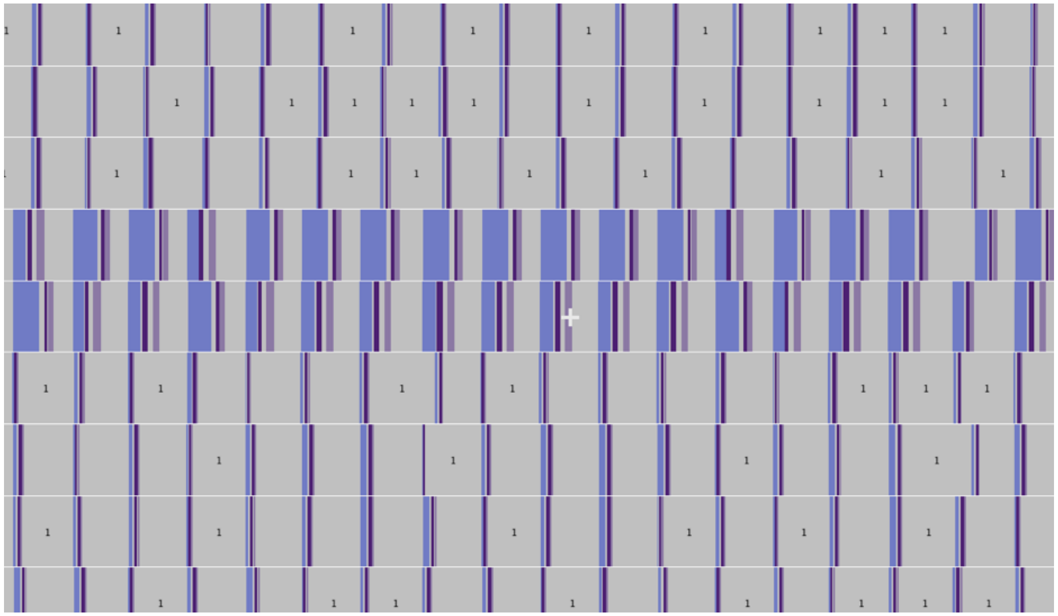
- **Charge all samples that threads receive while awaiting a lock to the lock itself**
- **When releasing a lock, accept blame at the lock**

all of the waiting occurs here (symptom)



GPU Successes with HPCToolkit

- **LAMMPS: identified hardware problem with Keeneland system**
  - — **improperly seated GPUs were observed to have lower data copy bandwidth**

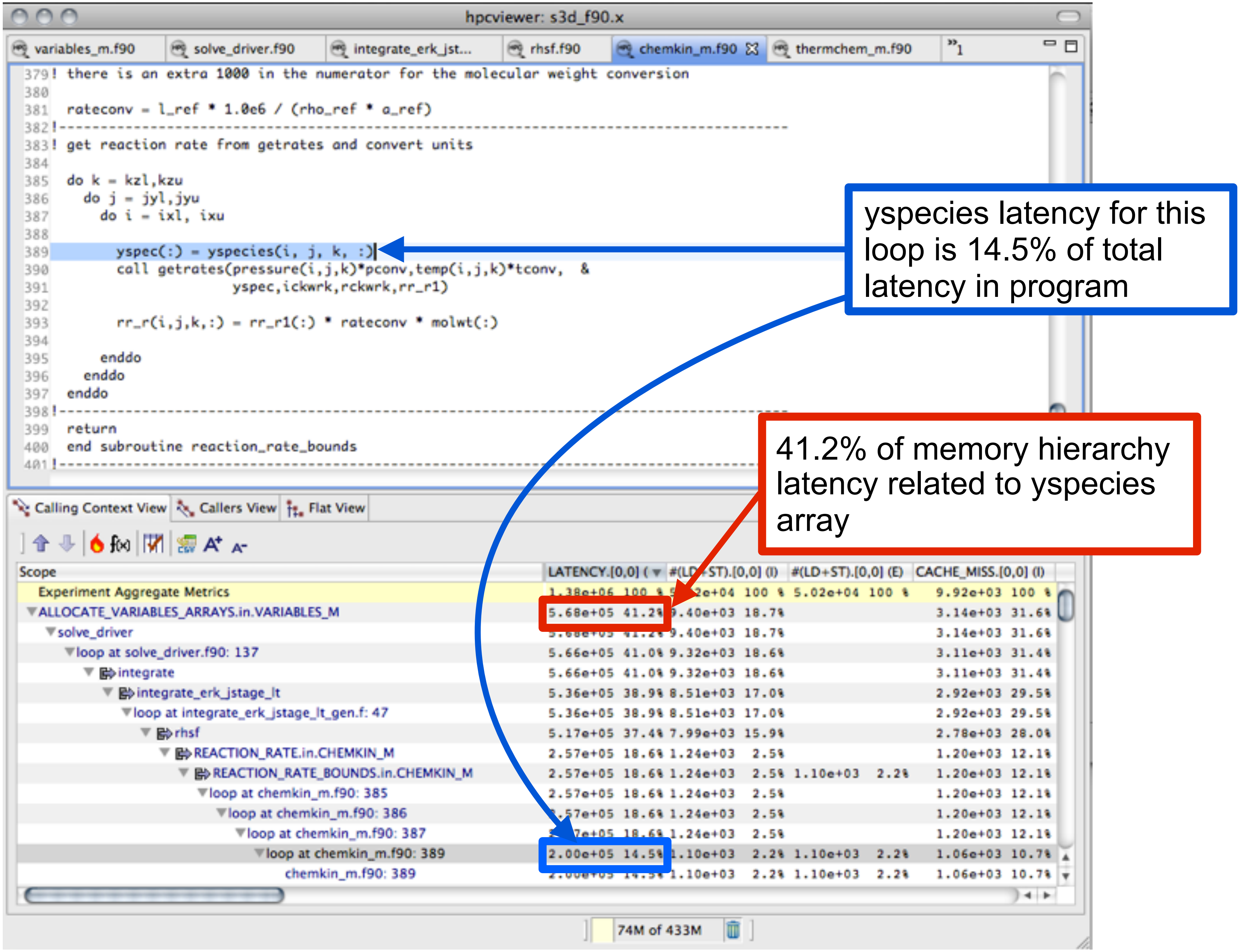


- **LLNL's LULESH: identified that dynamic memory allocation using cudaMalloc and cudaFree accounted for 90% of the idleness of the GPU**

Data Centric Analysis

- **Goal: associate memory hierarchy access costs with data**
- **Approach**
  - **intercept allocations to associate with their data ranges**
  - **measure latency**
    - **“instruction-based sampling” (AMD Opteron)**
    - **“marked instructions” (IBM Power)**
    - **“load latency facility” (Intel x86_64)**
  - **present quantitative results using hpcviewer**

Data Centric Analysis of S3D



Outline

- Overview of Rice's HPCToolkit
- Pinpointing scalability bottlenecks
  — scalability bottlenecks on large-scale parallel systems
  — scaling on multicore processors
- Understanding temporal behavior
- Assessing process variability
- Understanding threading, GPU, and memory hierarchy
  — blame shifting
  — attributing memory hierarchy costs to data
- Summary

Summary

- **Sampling provides low overhead measurement**
- **Call path profiling + binary analysis + blame shifting = insight**
  - scalability bottlenecks
  - where insufficient parallelism lurks
  - sources of lock contention
  - load imbalance
  - temporal dynamics
  - bottlenecks in hybrid code
  - problematic data structures
  - hardware counters for detailed diagnosis
- **Other capabilities**
  - attribute memory leaks back to their full calling context

Challenges Ahead

- Measure and analyze accelerated computations
- Measure and diagnose performance with billions of threads
- Understand how an application should best exploit complex, exposed memory hierarchies
- Help users optimize extreme-scale applications to reduce power consumption and improve energy efficiency
  - dynamic voltage and frequency scaling
- Evaluate behavior of adaptive runtime systems