
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**

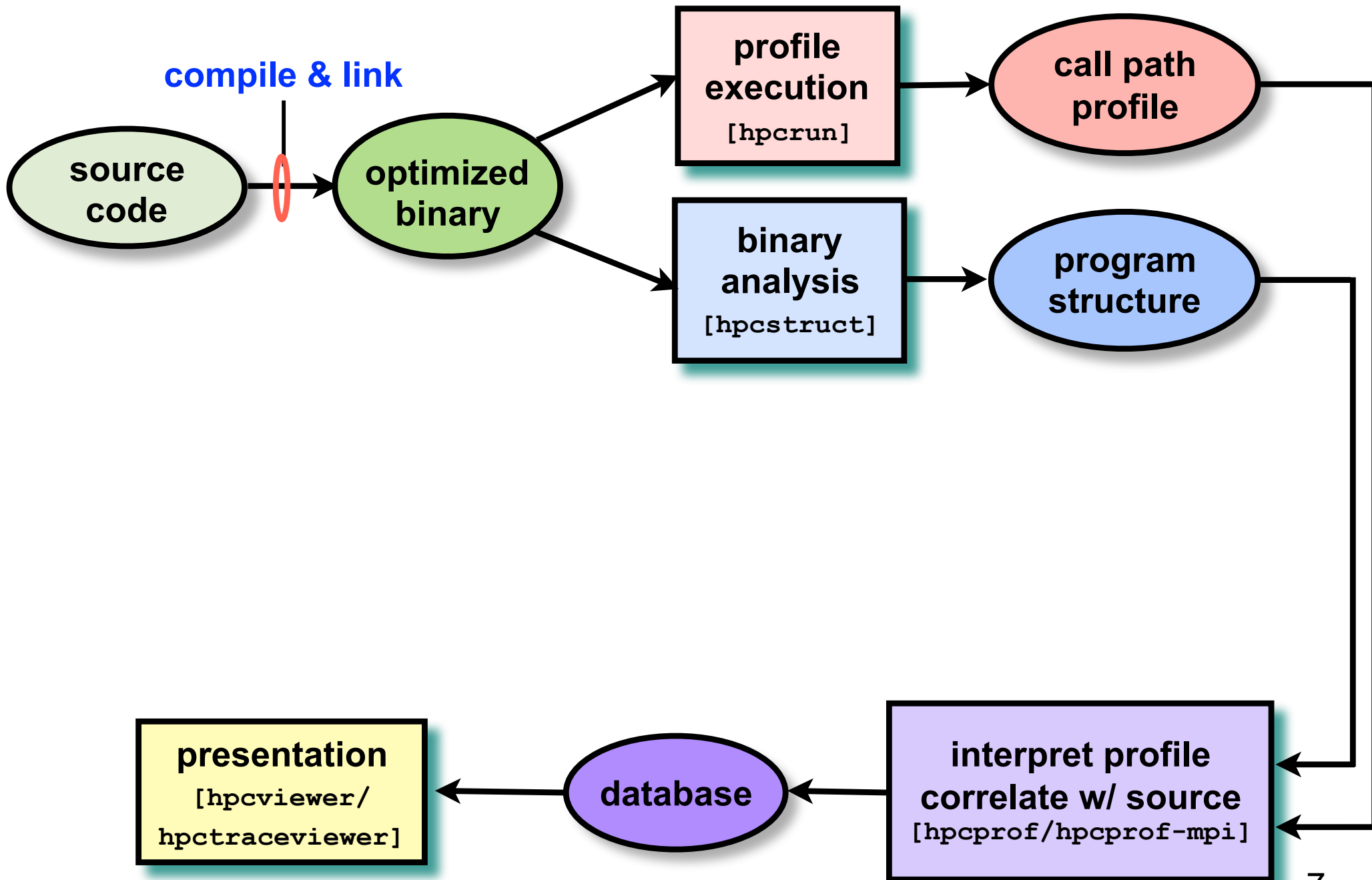
Rice University's HPCToolkit

- 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

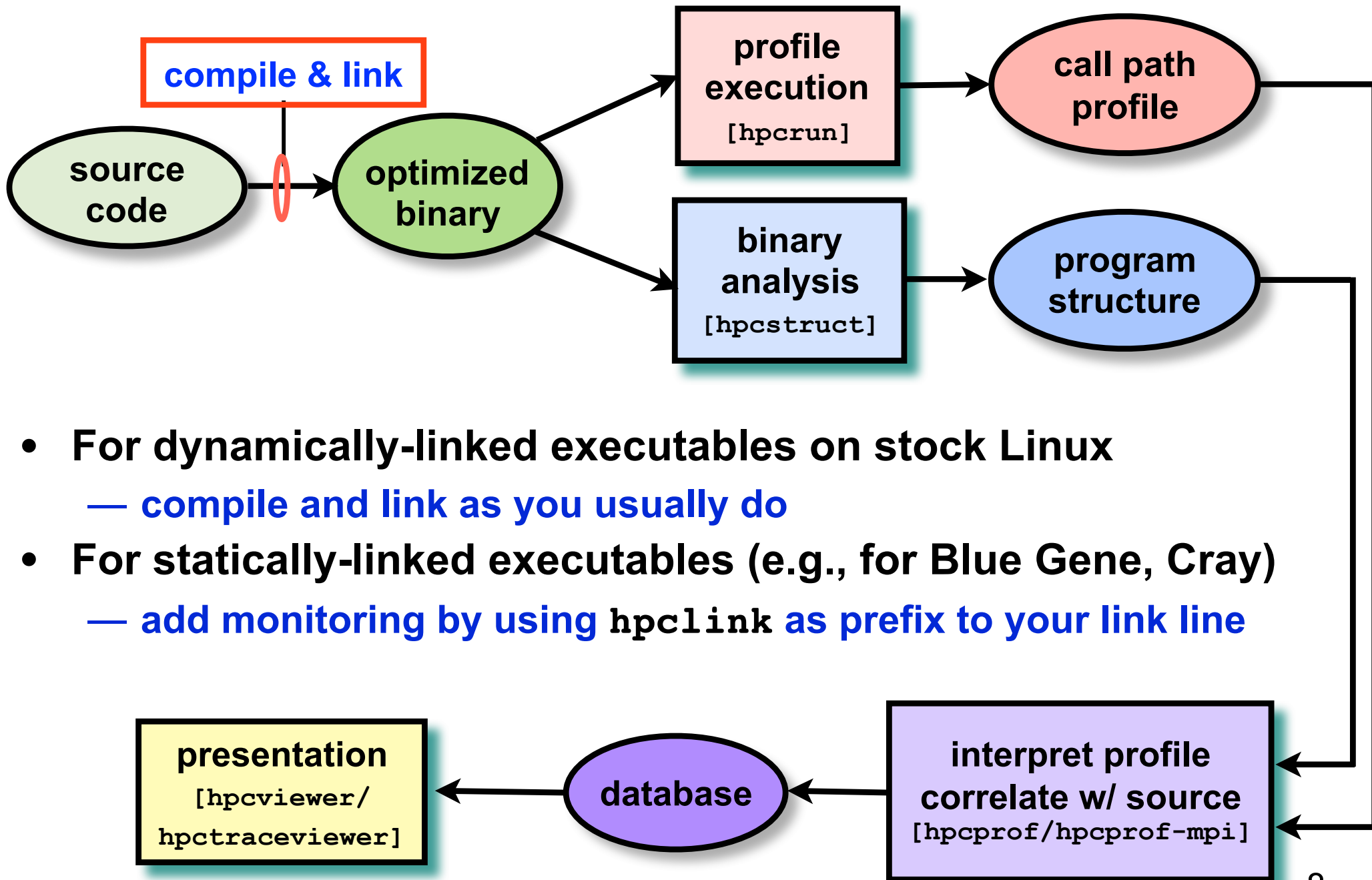
Outline

- 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

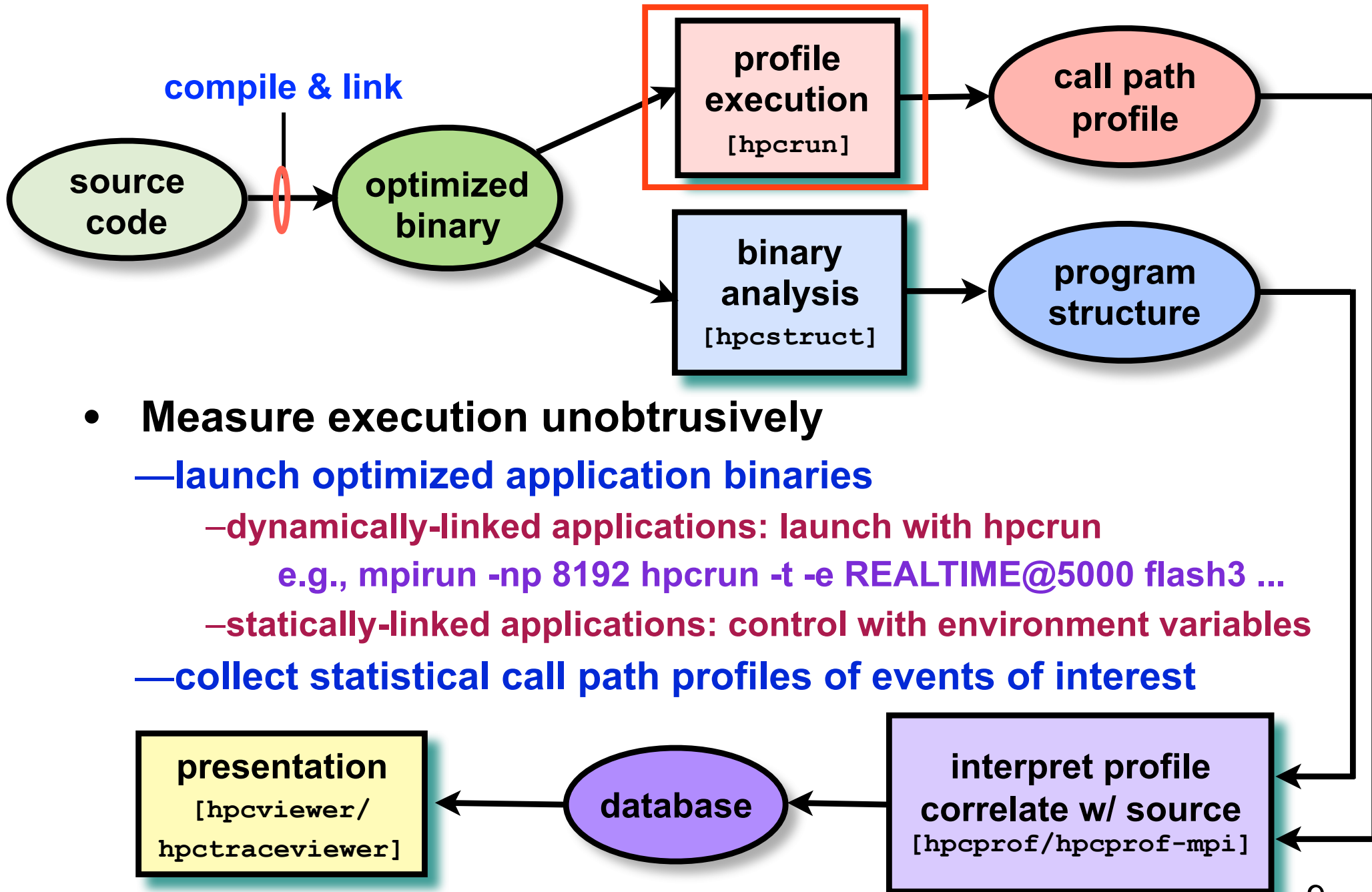


HPCToolkit Workflow



- 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



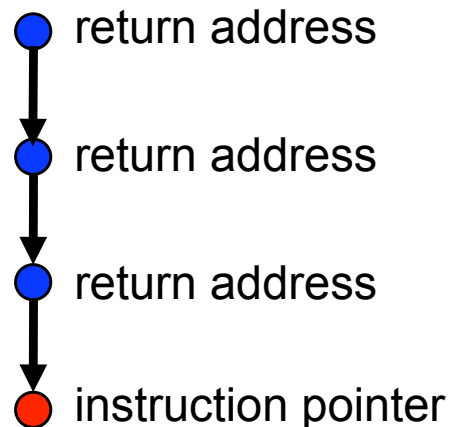
Call Path Profiling

Measure and attribute costs in context

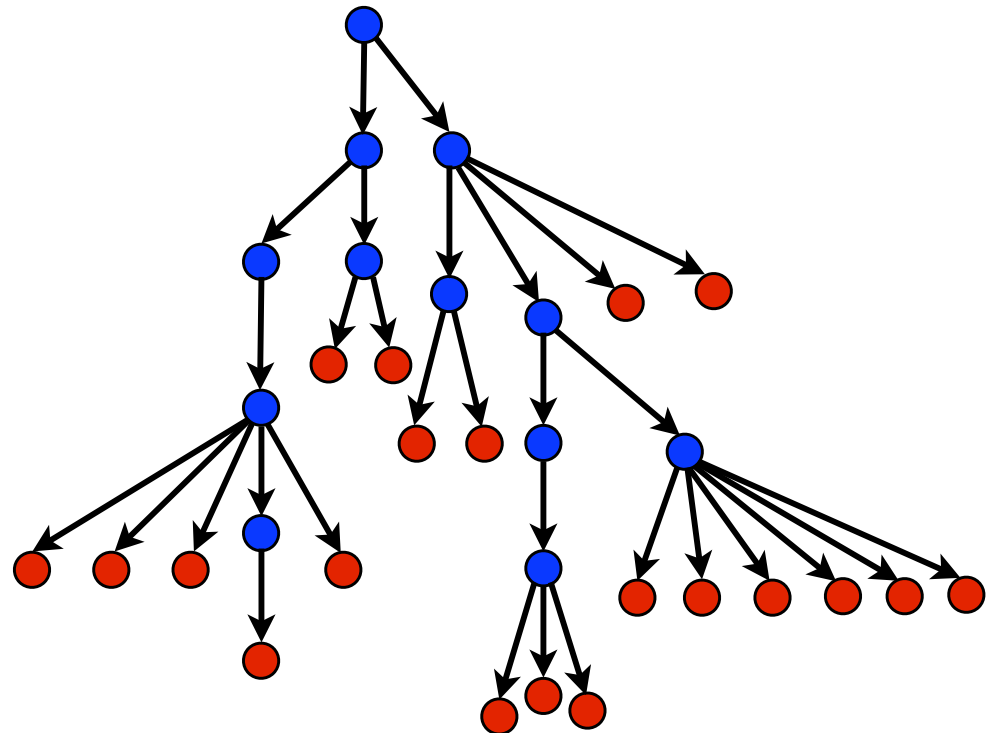
sample timer or hardware counter overflows

gather calling context using stack unwinding

Call path sample

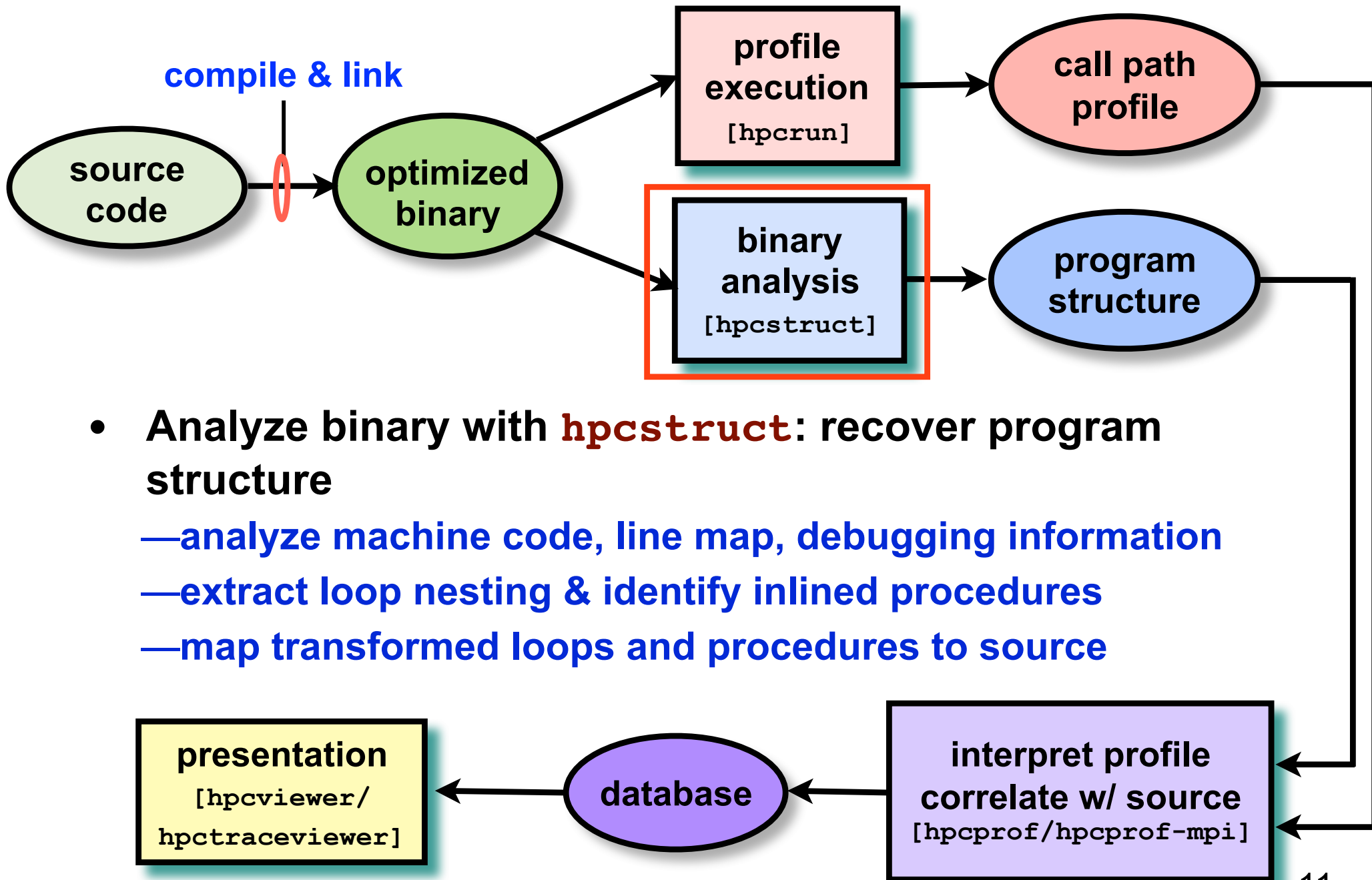


Calling context tree



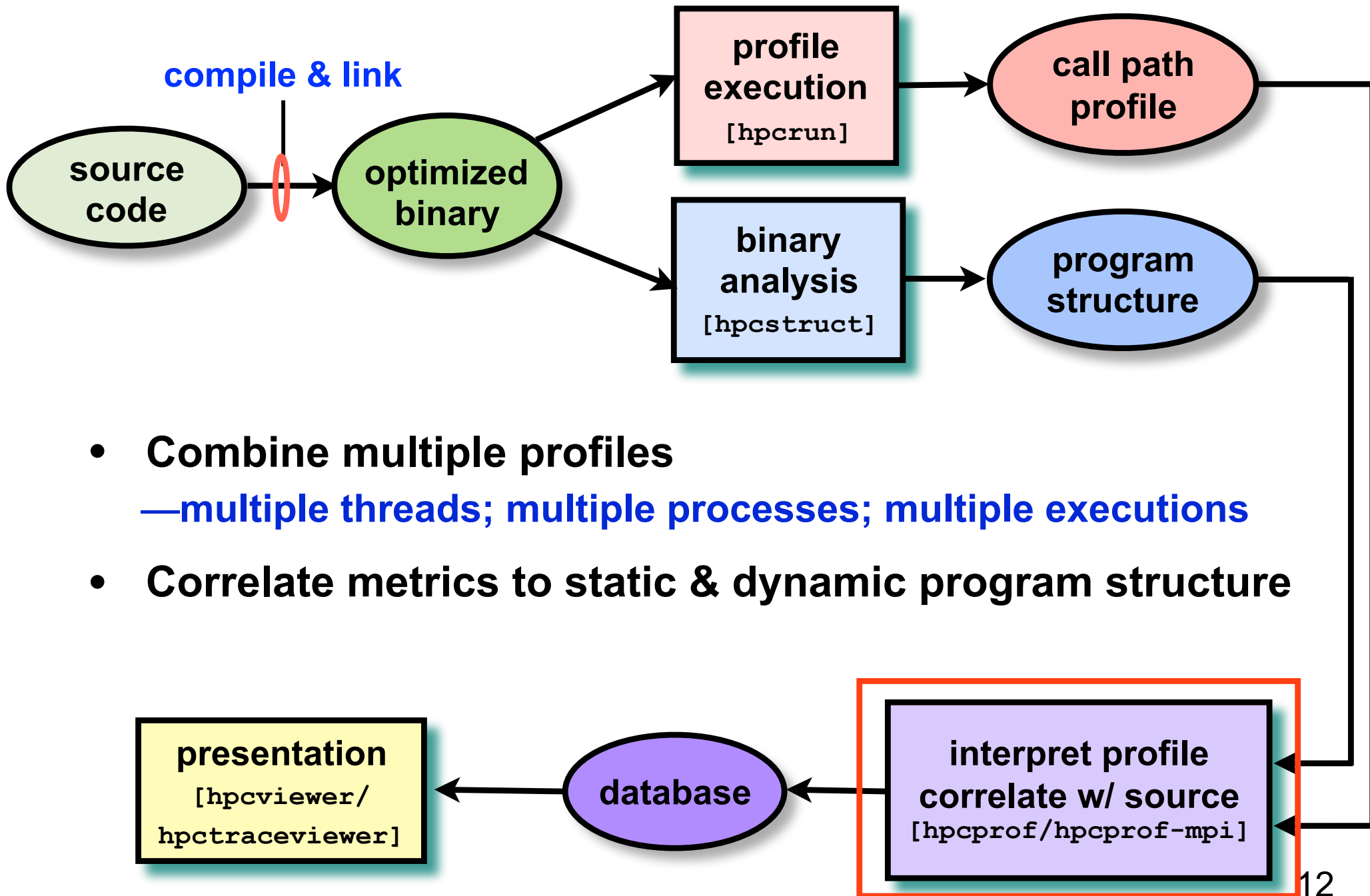
**Overhead proportional to sampling frequency...
...not call frequency**

HPCToolkit Workflow

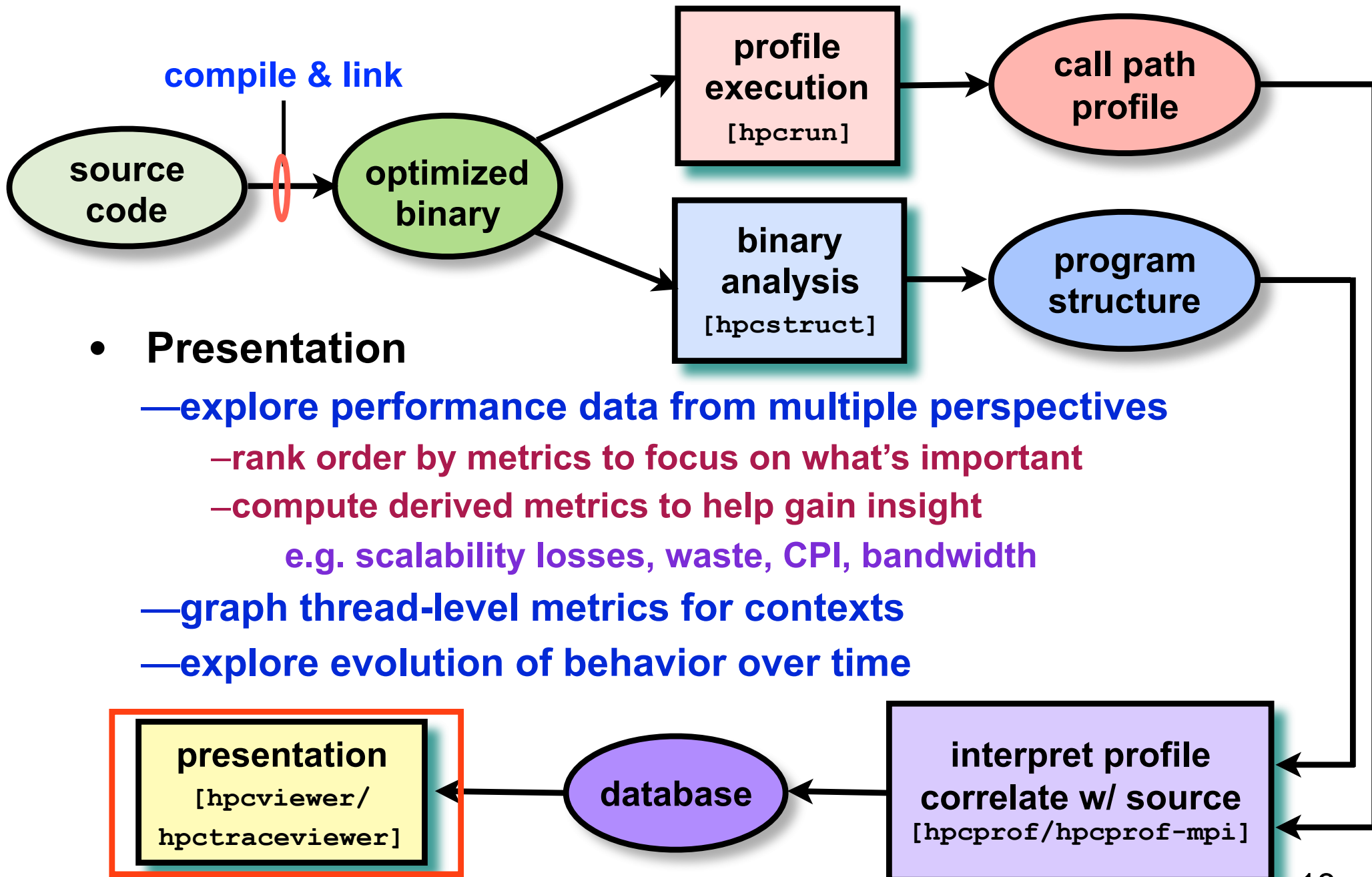


- Analyze binary with **hpcstruct**: recover program structure
 - analyze machine code, line map, debugging information
 - extract loop nesting & identify inlined procedures
 - map transformed loops and procedures to source

HPCToolkit Workflow



HPCToolkit Workflow



Code-centric Analysis with hpcviewer

The screenshot shows the hpcviewer interface for the executable 'lulesh-RAJA-parallel.exe'. The top menu bar includes File, Filter, View, Window, and Help. Below the menu, there are tabs for 'luleshRAJA-parallel.cxx' and 'forall_generic.hxx'. The main area displays the source code of 'forall' function. A red box labeled 'source pane' highlights the code editor. Below the code editor, there are three view controls: 'Calling Context View', 'Callers View', and 'Flat View'. A red box labeled 'view control ops' points to these controls. Below the view controls, there is a 'metric display' area with various icons. A red box labeled 'navigation pane' points to the 'Scope' pane on the left, which shows a hierarchical tree of the program's execution. The 'Scope' pane is expanded to show the 'main' function, which contains a loop at 'luleshRAJA-parallel.cxx: 3526'. This loop contains a call to 'LagrangeLeapFrog', which in turn calls 'LagrangeNodal', which calls 'CalcForceForNodes', which calls 'CalcVolumeForceForElems', which calls 'CalcHourglassControlForElems', which calls 'CalcFBHourglassForceForElems', which calls 'void RAJA::forall<RAJA::IndexSet::ExecPolicy<RAJA::seq_segit, RAJA::omp_parallel_for_exec, CalcFBHourglassForceForElems>>'. The 'metric display' area shows a table of metrics for each function call, including 'REALTIME (usec):Sum (I)' and 'REALTIME (usec):Sum (E)'. A red box labeled 'metric pane' points to this table.

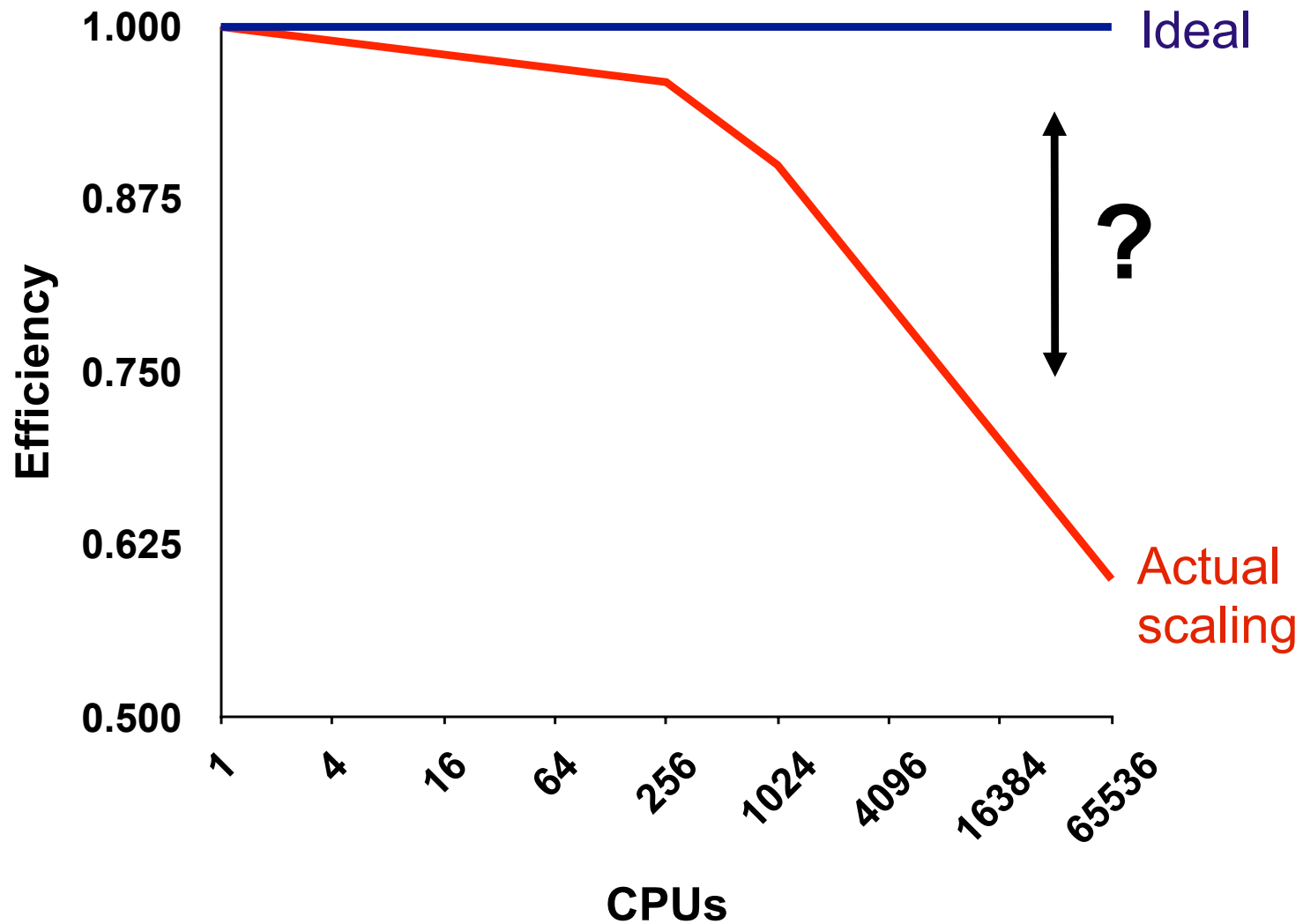
- function calls in full context
- inlined procedures
- inlined templates
- outlined OpenMP loops

Scope	REALTIME (usec):Sum (I)	REALTIME (usec):Sum (E)
Experiment Aggregate Metrics	2.26e+08 100 %	2.26e+08 100 %
<program root>	1.45e+08 63.9%	
497: main	1.45e+08 63.9%	6.01e+03 0.0%
loop at luleshRAJA-parallel.cxx: 3526	1.44e+08 63.8%	
3528: [I] LagrangeLeapFrog(Domain*)	1.44e+08 63.8%	
2715: [I] LagrangeNodal(Domain*)	8.25e+07 36.5%	
1554: [I] CalcForceForNodes(Domain*)	5.15e+07 22.8%	
1469: CalcVolumeForceForElems(Domain*)	3.10e+07 13.7%	
1454: [I] CalcHourglassControlForElems(Domain*, double*, double)	2.43e+07 10.8%	
1399: [I] CalcFBHourglassForceForElems(int*, double*, double*, double*, double*, double)	2.43e+07 10.8%	
1187: [I] void RAJA::forall<RAJA::IndexSet::ExecPolicy<RAJA::seq_segit, RAJA::omp_parallel_for_exec, CalcFBHourglassForceForElems>>	2.43e+07 10.8%	
405: [I] void RAJA::forall<RAJA::omp_parallel_for_exec, CalcFBHourglassForceForElems>>	2.43e+07 10.8%	1.00e+03 0.0%
loop at forall_seq_any.hxx: 498	2.43e+07 10.8%	
505: [I] void RAJA::forall<CalcFBHourglassForceForElems(int*, double*, double)	2.42e+07 10.7%	3.91e+04 0.0%
89: outline forall_omp_any.hxx:89 (0x423620)	2.42e+07 10.7%	3.41e+04 0.0%
loop at forall_omp_any.hxx: 90	2.42e+07 10.7%	9.84e+06 4.3%
91: [I] CalcFBHourglassForceForElems(int*, double*, double*, double*, double)	1.11e+07 4.9%	1.11e+07 4.9%
1300: [I] CalcElemFBHourglassForce(double*, double*, double*, double)	3.27e+06 1.4%	2.00e+05 0.1%
1260: [I] CBRT(double)		

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

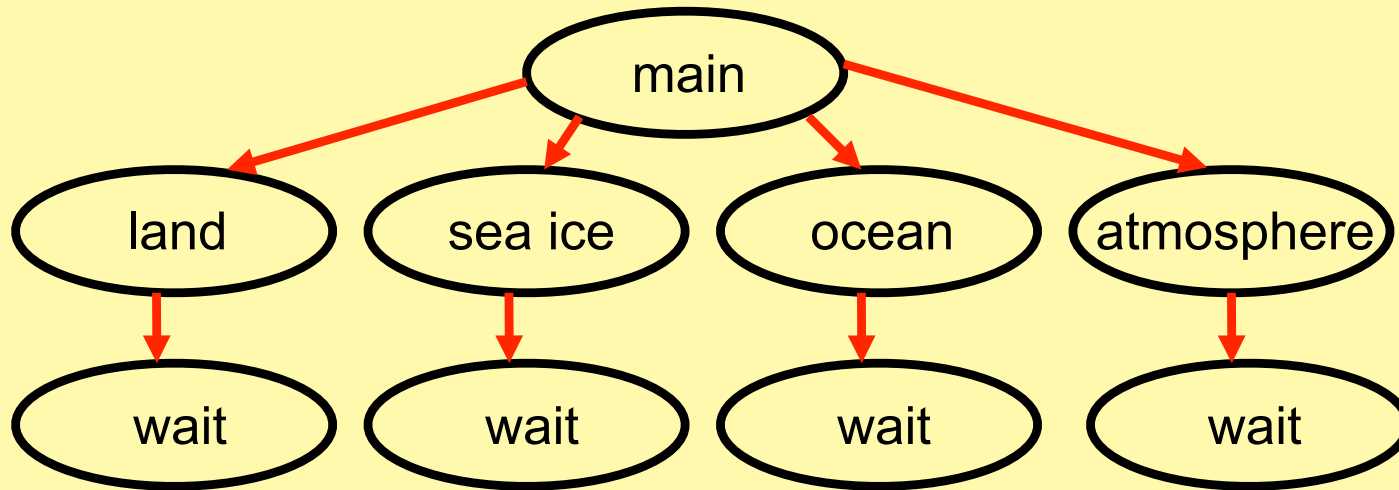
- **Isolate scalability bottlenecks**
- **Guide user to problems**
- **Quantify the magnitude of each problem**

Challenges for Pinpointing Scalability Bottlenecks

- **Parallel applications**

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

Example climate code skeleton



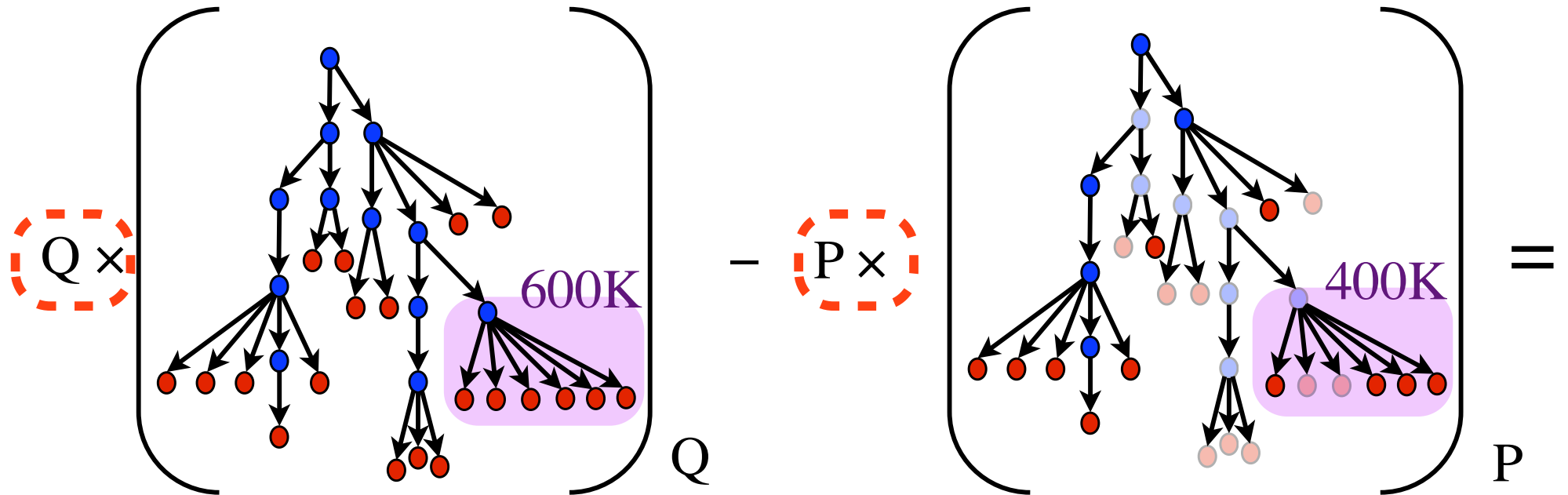
- **Monitoring**

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

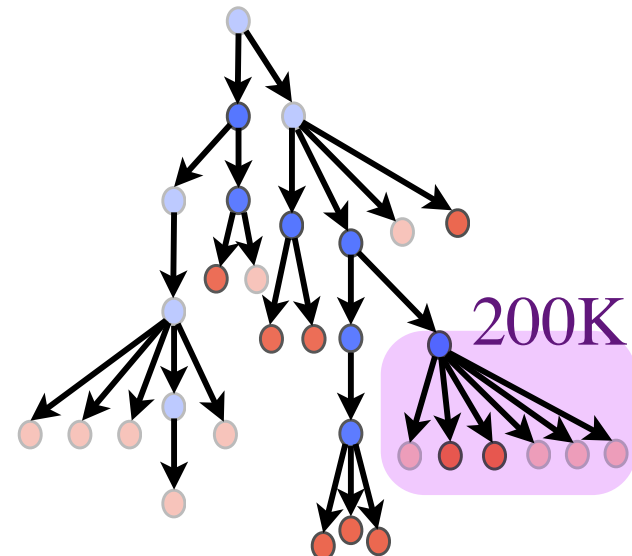
Performance Analysis with Expectations

- 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

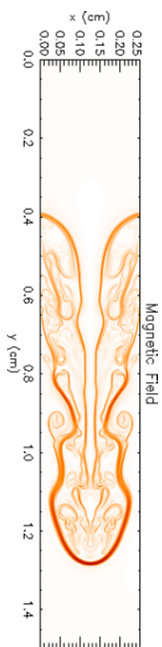


coefficients for analysis
of strong scaling

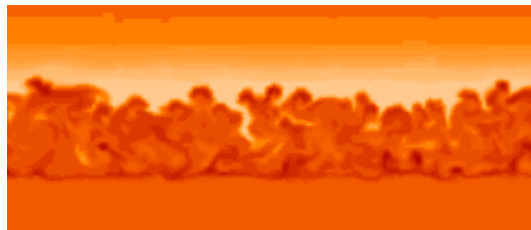


Scalability Analysis Demo: FLASH3

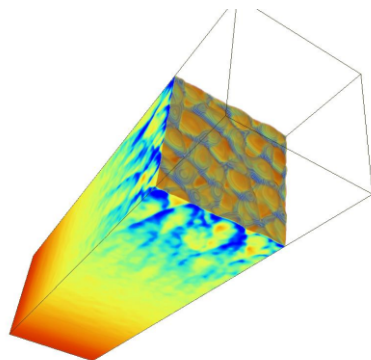
Code:	University of Chicago FLASH3
Simulation:	white dwarf detonation
Platform:	Blue Gene/P
Experiment:	8192 vs. 256 cores
Scaling type:	weak



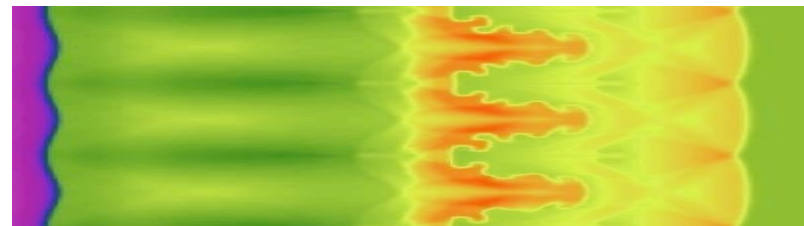
*Magnetic
Rayleigh-Taylor*



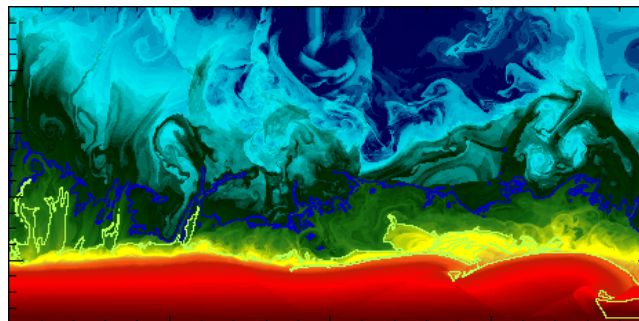
Nova outbursts on white dwarfs



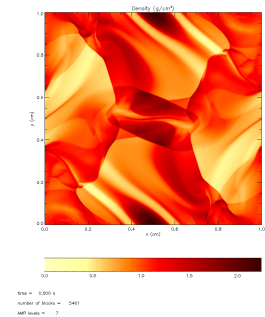
Cellular detonation



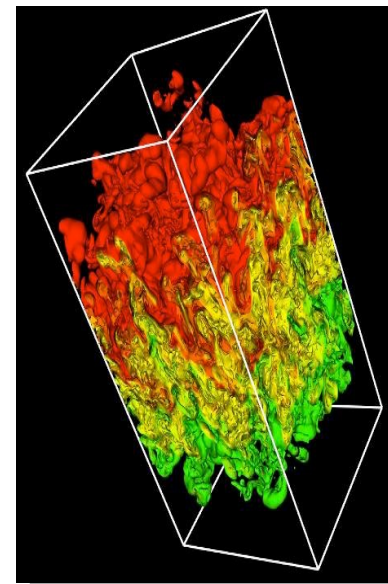
Laser-driven shock instabilities



Helium burning on neutron stars



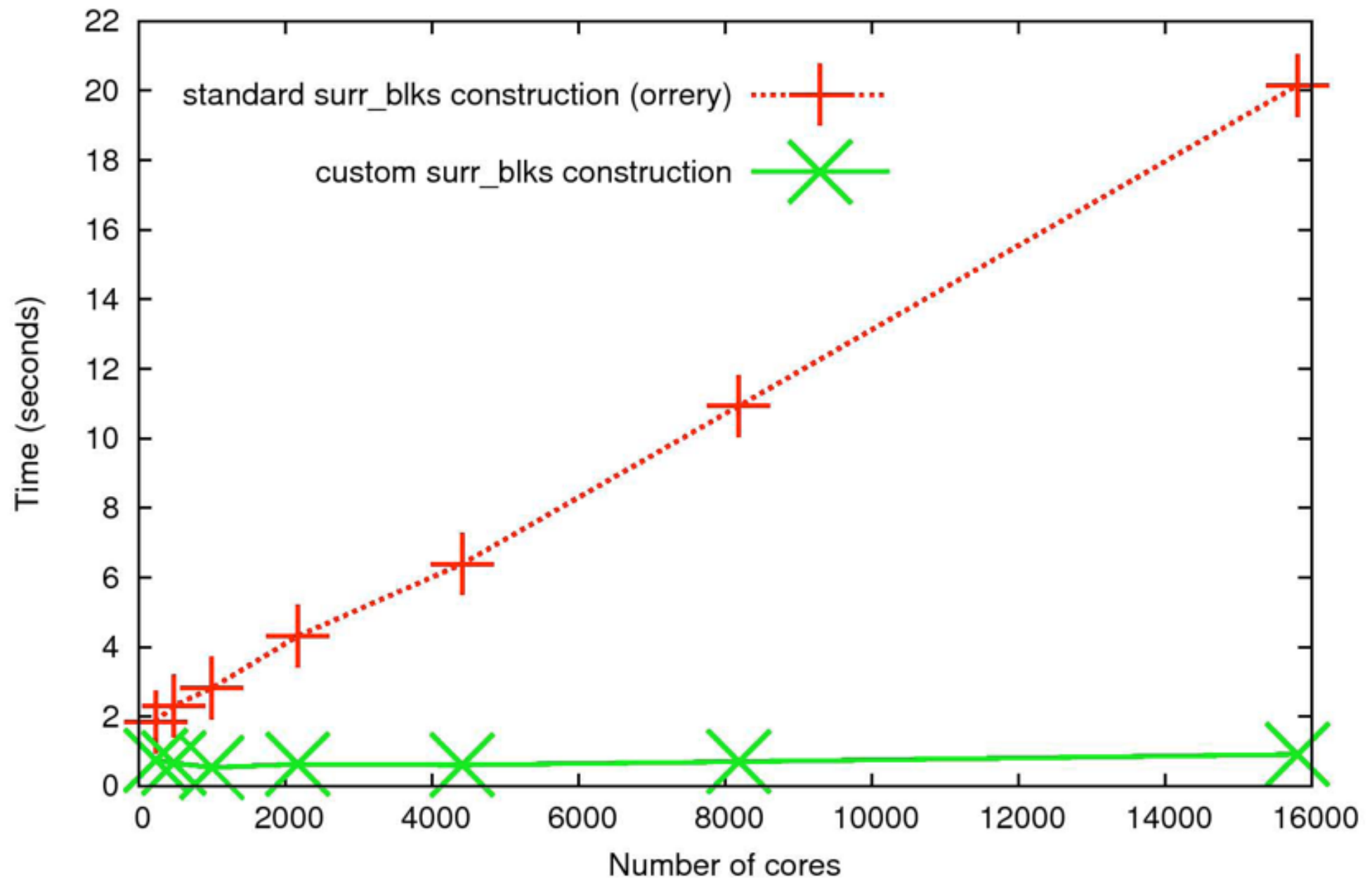
*Orzag/Tang MHD
vortex*



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

The screenshot displays the hpcviewer interface. The top pane shows the source code for the subroutine `rhsf`. The bottom pane shows a performance table with columns for Scope, 1-core (ms) (I), 1-core (ms) (E), 8-core(1) (ms) (I), 8-core(1) (ms) (E), and Multicore Loss. The `rhsf` row is highlighted, and its Multicore Loss value of 13.0% is circled in blue. A red arrow points from this value to a text box on the right.

```
1 subroutine rhsf( q, rhs )
2 !-----
3 ! Changes
4 ! Ramanan Sankaran - 01/04/05
5 ! 1. Diffusive fluxes are computed without having to convert units.
6 ! Ignore older comments about conversion to CGS units.
7 ! This saves a lot of flops.
8 ! 2. Mixavg and Lewis transport modules have been made interchangeable
9 ! by adding dummy arguments in both.
10 !-----
11 !
12 !           Author:  James Sutherland
13 !           Date:    April, 2002
14 !-----
15 ! This routine calculates the time rate of change for the
16 ! 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

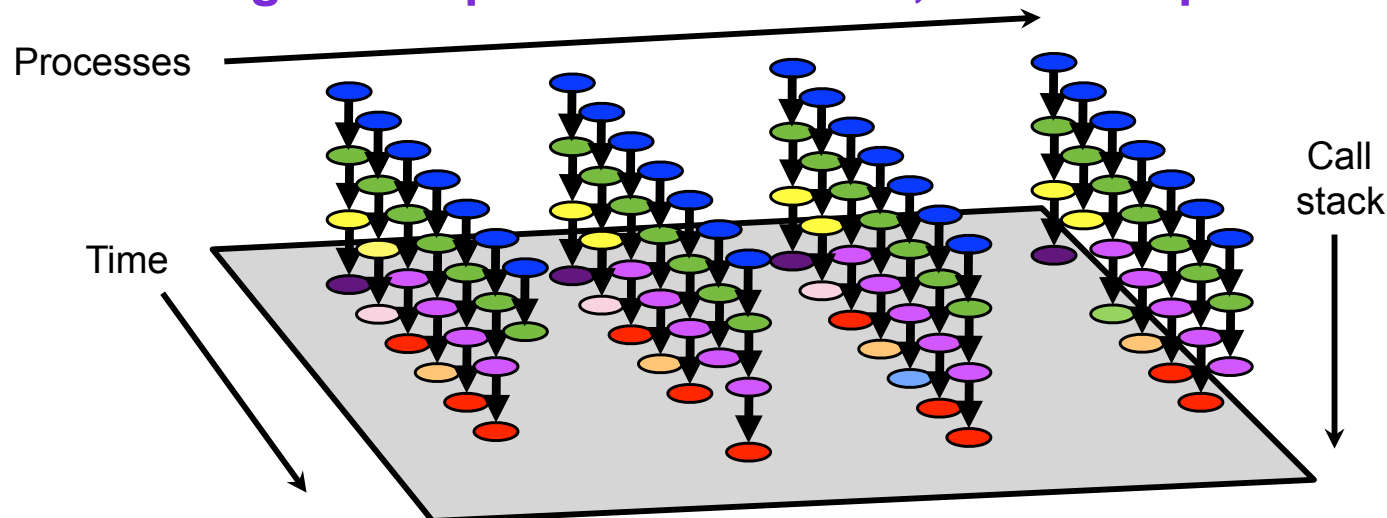
Understanding Temporal Behavior

- Profiling compresses out the temporal dimension
 - temporal patterns, e.g. serialization, are invisible in profiles
- What can we do? Trace call path samples

—sketch:

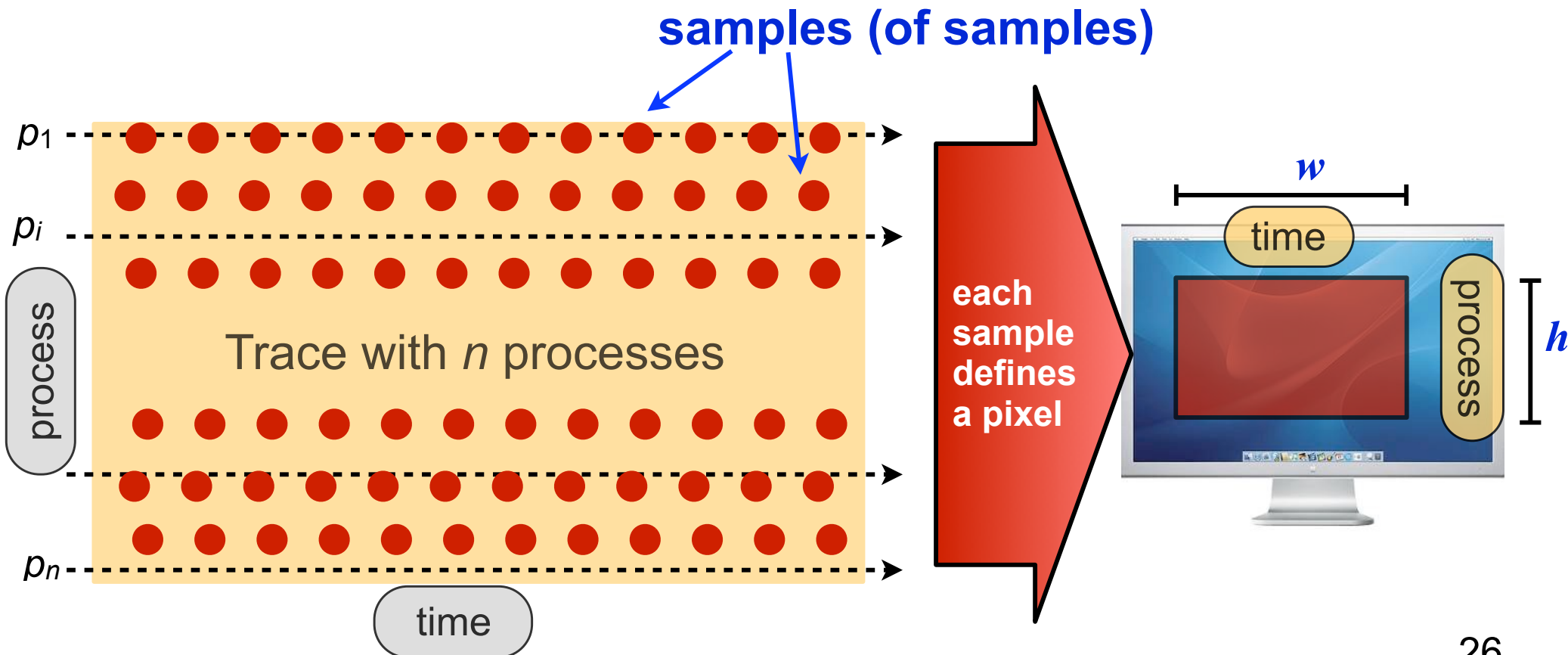
- N times per second, take a call path sample of each thread
- organize the samples for each thread along a time line
- view how the execution evolves left to right
- what do we view?

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 $h \times w$
 - typically many more processes (or threads) than h
 - typically many more samples (trace records) than w
- Solution: sample the samples!

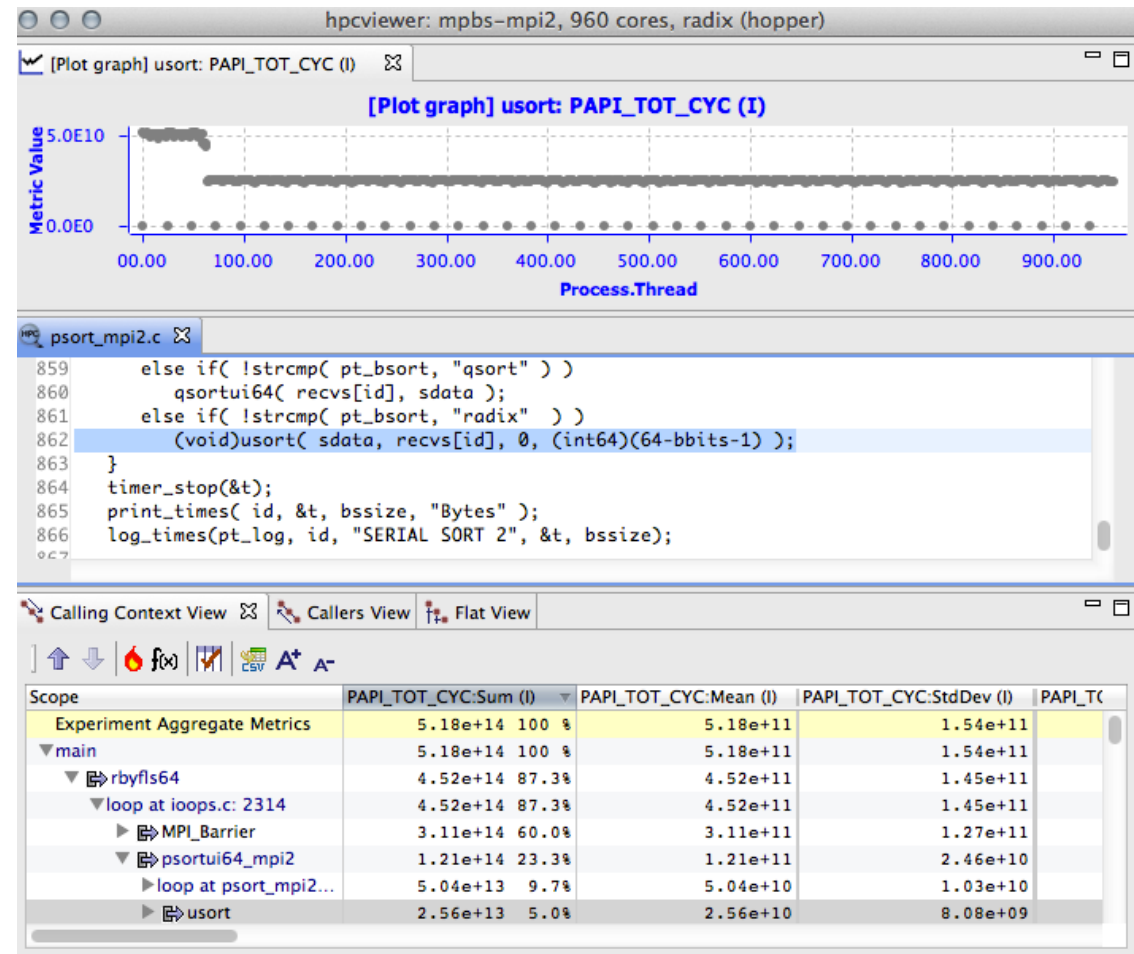
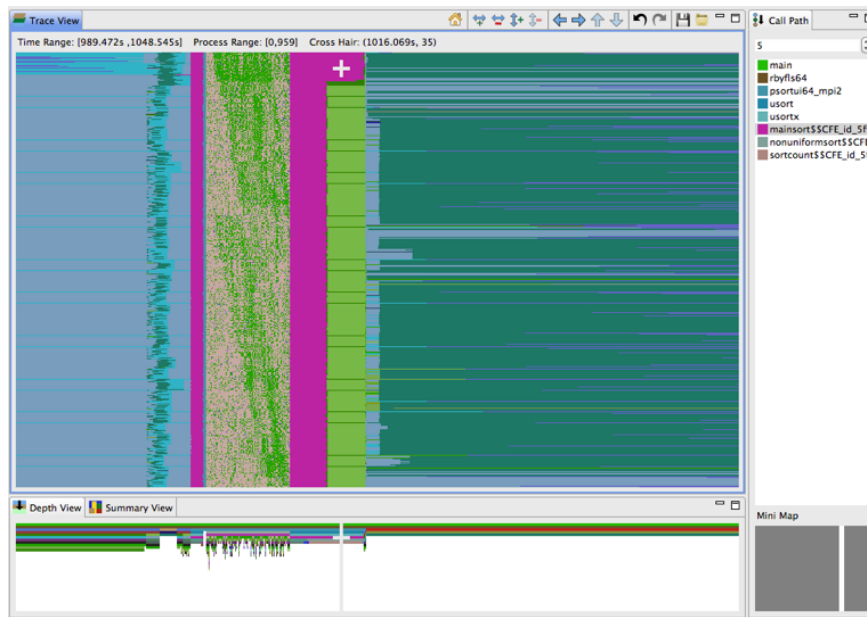


Outline

- 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

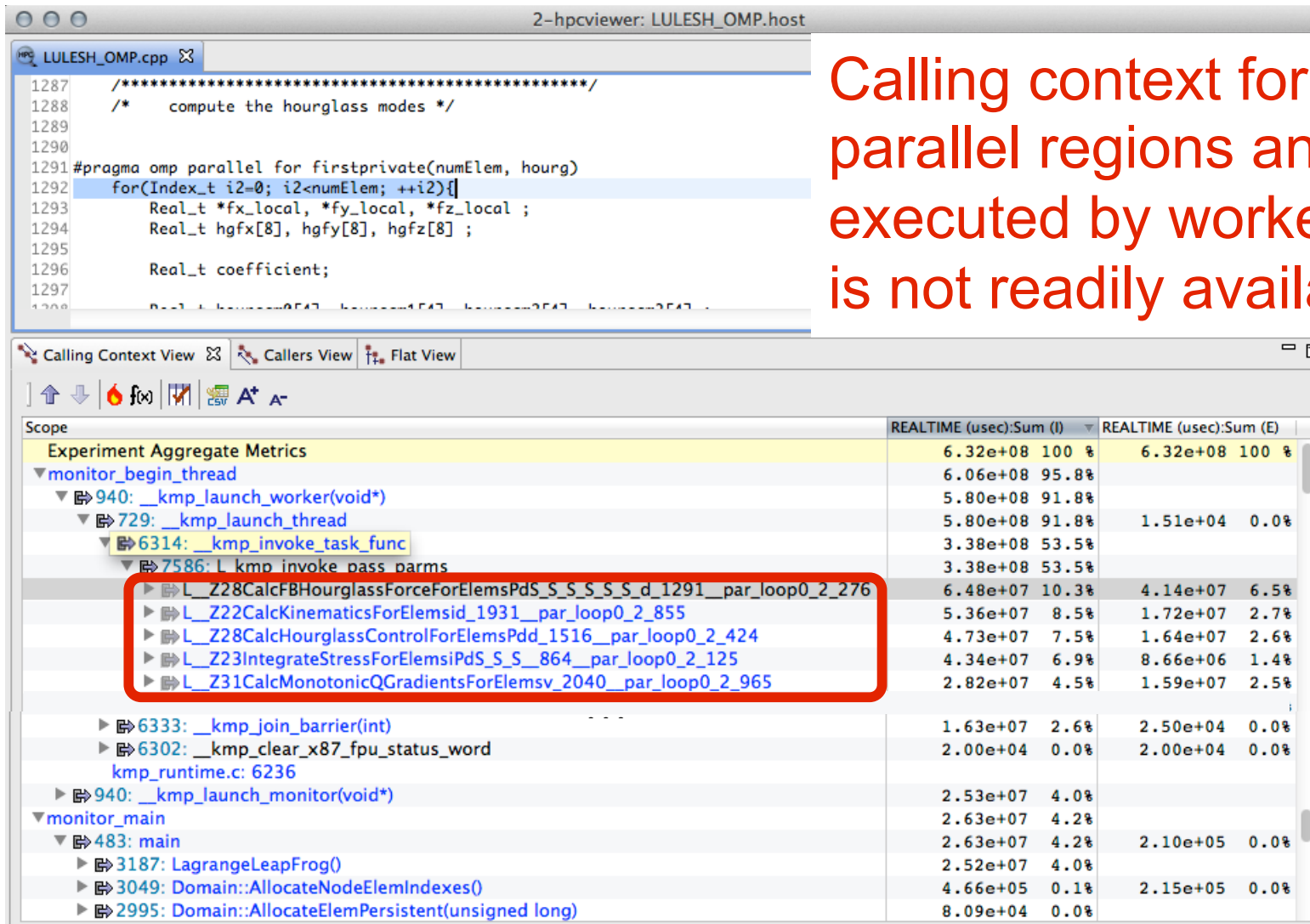
Two views of load imbalance since not on a 2^k cores



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



The screenshot shows the hpcviewer interface for the LULESH_OMP.host. The top pane displays the source code of LULESH_OMP.cpp, with a loop of tasks highlighted. The bottom pane shows the Calling Context View, which lists the tasks and their performance metrics. A red box highlights a specific task in the profile.

Source code snippet (LULESH_OMP.cpp):

```
1287 /*****  
1288  /* compute the hourglass modes */  
1289  
1290  
1291 #pragma omp parallel for firstprivate(numElem, hourg)  
1292 for(Index_t i2=0; i2<numElem; ++i2){  
1293     Real_t *fx_local, *fy_local, *fz_local ;  
1294     Real_t hgfx[8], hgy[8], hgz[8] ;  
1295  
1296     Real_t coefficient;  
1297  
1298     Real_t hourglass1[8], hourglass2[8], hourglass3[8], hourglass4[8] ;  
1299 }
```

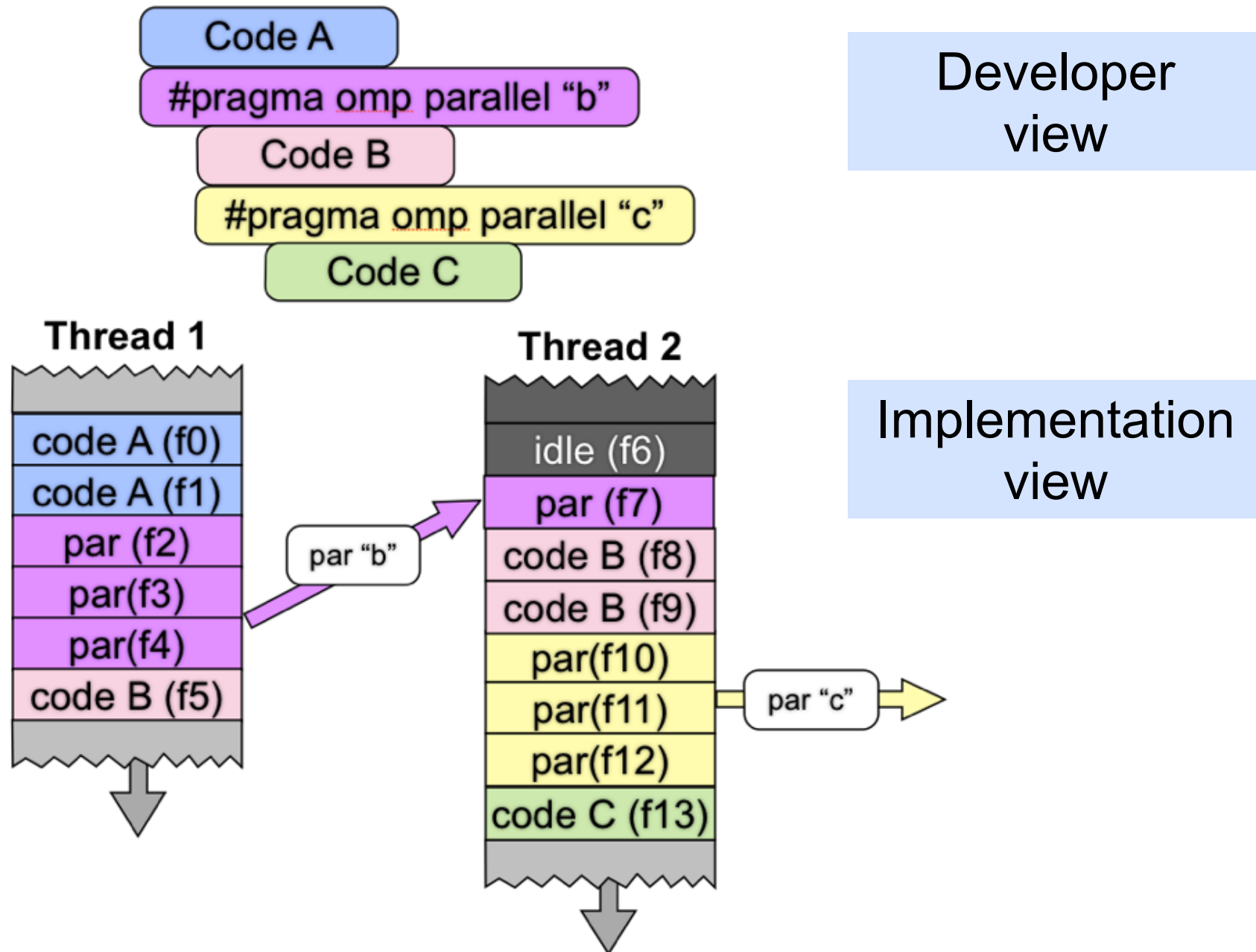
Performance Profile (Calling Context View):

Scope	REALTIME (usec):Sum (I)	REALTIME (usec):Sum (E)
Experiment Aggregate Metrics	6.32e+08 100 %	6.32e+08 100 %
monitor_begin_thread	6.06e+08 95.8%	
940: __kmp_launch_worker(void*)	5.80e+08 91.8%	
729: __kmp_launch_thread	5.80e+08 91.8%	1.51e+04 0.0%
6314: __kmp_invoke_task_func	3.38e+08 53.5%	
7586: L kmp invoke pass parms	3.38e+08 53.5%	
L_Z28CalcFBHourglassForceForElemsPdS_S_S_S_S_d_1291__par_loop0_2_276	6.48e+07 10.3%	4.14e+07 6.5%
L_Z22CalcKinematicsForElemsid_1931__par_loop0_2_855	5.36e+07 8.5%	1.72e+07 2.7%
L_Z28CalcHourglassControlForElemsPdd_1516__par_loop0_2_424	4.73e+07 7.5%	1.64e+07 2.6%
L_Z23IntegrateStressForElemsPdS_S_S_864__par_loop0_2_125	4.34e+07 6.9%	8.66e+06 1.4%
L_Z31CalcMonotonicQGradientsForElemsv_2040__par_loop0_2_965	2.82e+07 4.5%	1.59e+07 2.5%
6333: __kmp_join_barrier(int)	1.63e+07 2.6%	2.50e+04 0.0%
6302: __kmp_clear_x87_fpu_status_word	2.00e+04 0.0%	2.00e+04 0.0%
kmp_runtime.c: 6236		
940: __kmp_launch_monitor(void*)	2.53e+07 4.0%	
monitor_main	2.63e+07 4.2%	
483: main	2.63e+07 4.2%	2.10e+05 0.0%
3187: LagrangeLeapFrog()	2.52e+07 4.0%	
3049: Domain::AllocateNodeElemIndexes()	4.66e+05 0.1%	2.15e+05 0.0%
2995: Domain::AllocateElemPersistent(unsigned long)	8.09e+04 0.0%	

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



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

- | | |
|--|-------------------------|
| <ul style="list-style-type: none">• Threads• Parallel regions• Tasks | create/exit event pairs |
| <ul style="list-style-type: none">• Runtime shutdown• User-level control API<ul style="list-style-type: none">— e.g., support tool start/stop | singleton events |

OpenMP: Meaningless Hotspots

hpcviewer: lulesh-par-original

wait.h

```
43 extern long int gomp_futex_wait, gomp_futex_wake;
44
45 #include <futex.h>
46
47 static inline void do_wait (int *addr, int val)
48 {
49     unsigned long long i, count = gomp_spin_count_var;
50
51     if (__builtin_expect (gomp_managed_threads > gomp_available_cpus, 0))
52         count = gomp_throttled_spin_count_var;
53     for (i = 0; i < count; i++) {
54         HMT_low();
55     }
56 }
```

hotspot is do_wait,
but don't know why

Calling Context View Callers View Flat View

Scope PAPI_TOT_CYC:Sum (I) PAPI_TOT_CYC:Sum (E)

Experiment Aggregate Metrics	1.32e+13 100 %	1.32e+13 100 %
do_wait	9.98e+12 75.8%	9.98e+12 75.8%
_ZL28CalcHourglassControlForElemsPdd.omp_fn.5	4.91e+11 3.7%	4.91e+11 3.7%
gomp_barrier_wait_start	4.24e+11 3.2%	4.24e+11 3.2%
_ZL23IntegrateStressForElemsiPdS_S_S_.omp_fn.3	3.98e+11 3.0%	3.98e+11 3.0%
_ZL28CalcFBHourglassForceForElemsPdS_S_S_S_S_d.omp_fn.12	4.26e+11 3.2%	3.43e+11 2.6%
_ZL22CalcKinematicsForElemsid.omp_fn.13	3.22e+11 2.4%	3.22e+11 2.4%
_ZL15EvalEOSForElemsPdi.omp_fn.17	3.95e+11 3.0%	1.25e+11 0.9%
_ZL31CalcMonotonicQGradientsForElemsv.omp_fn.14	1.22e+11 0.9%	1.22e+11 0.9%

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 \leftrightarrow 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

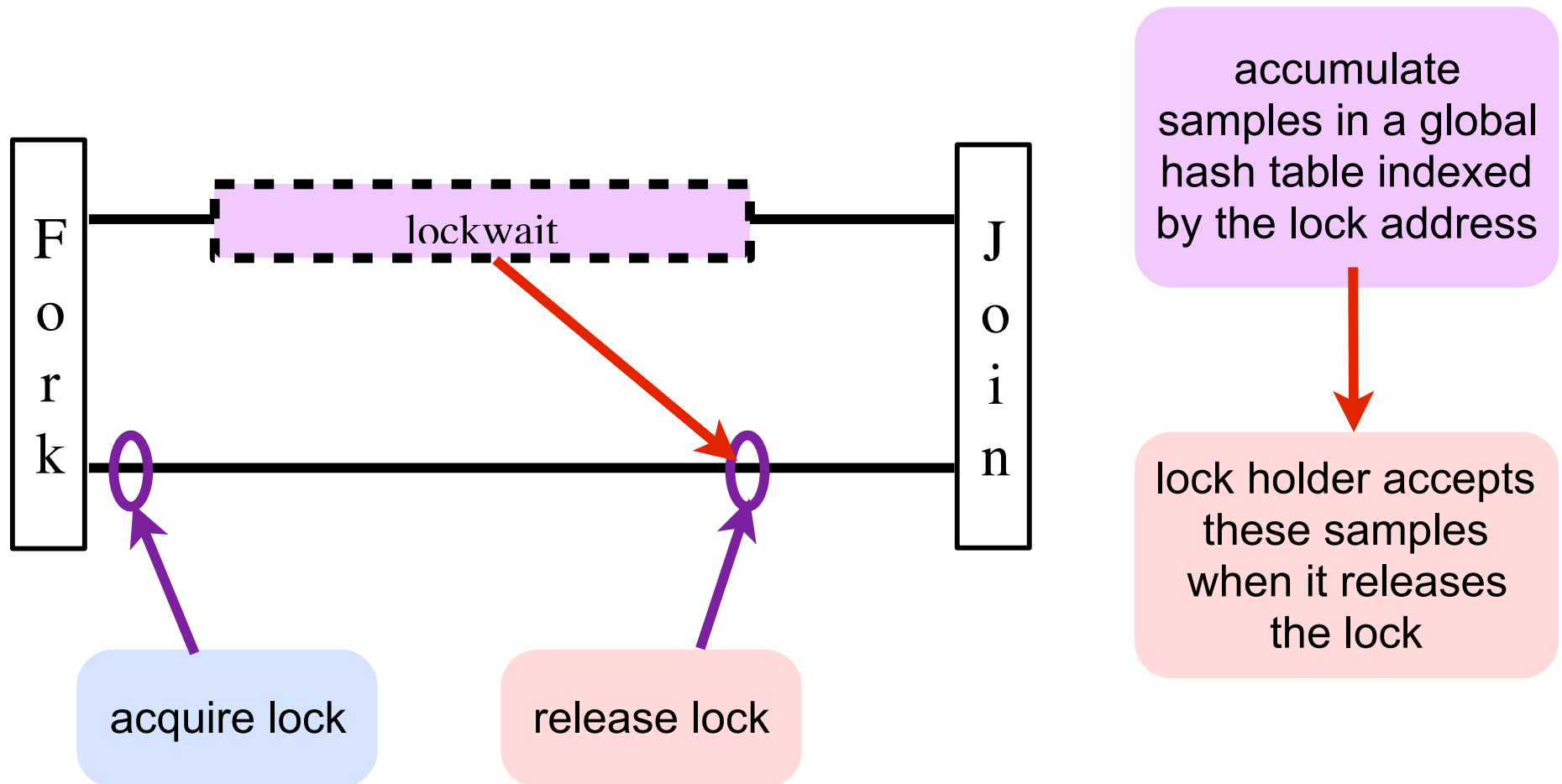
	Problem	Approach
Undirected Blame Shifting¹	A thread is idle waiting for work	Apportion blame among working threads for not shedding enough parallelism to keep all threads busy
Directed Blame Shifting²	A thread is idle waiting for a mutex	Blame the thread holding the mutex for idleness of threads waiting for the mutex

¹Tallent & Mellor-Crummey: PPOPP 2009

²Tallent, 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 **all of the waiting occurs here (symptom)**

hpcviewer: locktest-2.host

locktest-2.c

```
1 #include <omp.h>
2 #include "fib.h"
3 void g() {
4     int i;
5     omp_lock_t l;
6     omp_init_lock(&l);
7     #pragma omp parallel
8     {
9         #pragma omp master
10        {
11            omp_set_lock(&l);
12            fib(40);
13            omp_unset_lock(&l);
14        }
15        #pragma omp for
16        for(i = 0; i < 100; i++) {
17            omp_set_lock(&l);
18            fib(10);
19            omp_unset_lock(&l);
20        }
21    }
22 }
23 void f() { g(); }
24 int main() { f(); return 0; }
```

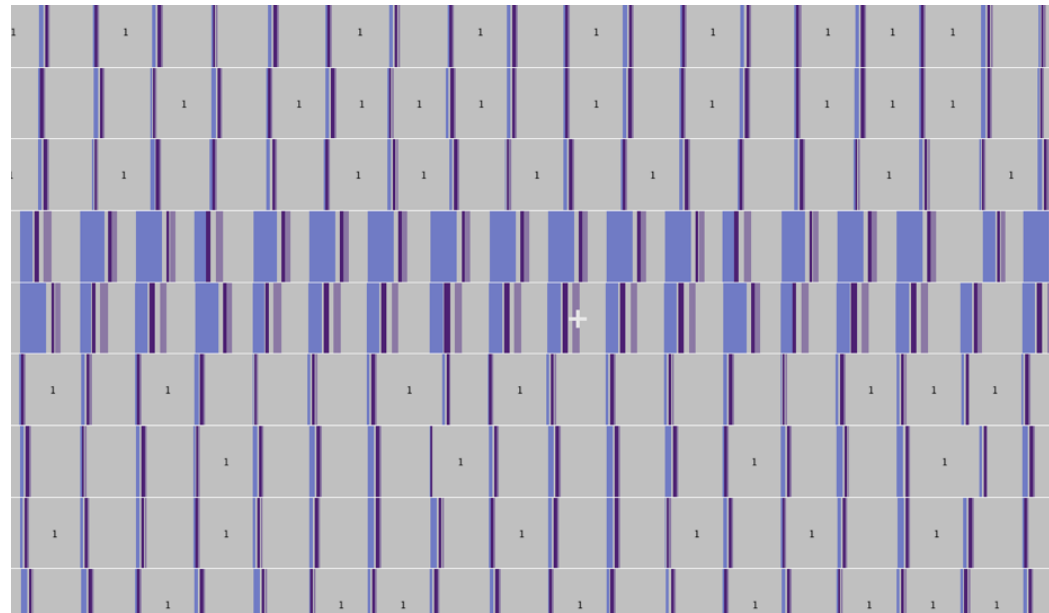
almost all blame for the waiting is attributed here (cause)

Calling Context View

Scope	MUTEX_WAIT:Sum (I)	MUTEX_BLAKE:Sum (I)
Experiment Aggregate Metrics	8.11e+07 100 %	7.93e+07 100 %
monitor_main	8.11e+07 100 %	7.93e+07 100 %
483: main	8.11e+07 100 %	7.93e+07 100 %
29: f	8.11e+07 100 %	7.93e+07 100 %
25: g	8.11e+07 100 %	7.93e+07 100 %
7: L_g_7_par_region0_2_90	8.11e+07 100 %	7.93e+07 100 %
17: kmopc set lock	8.11e+07 100 %	
12: fib		
20: _kmopc_barrier		
locktest-2.c: 13		7.87e+07 99.2%

GPU Successes with HPCToolkit

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

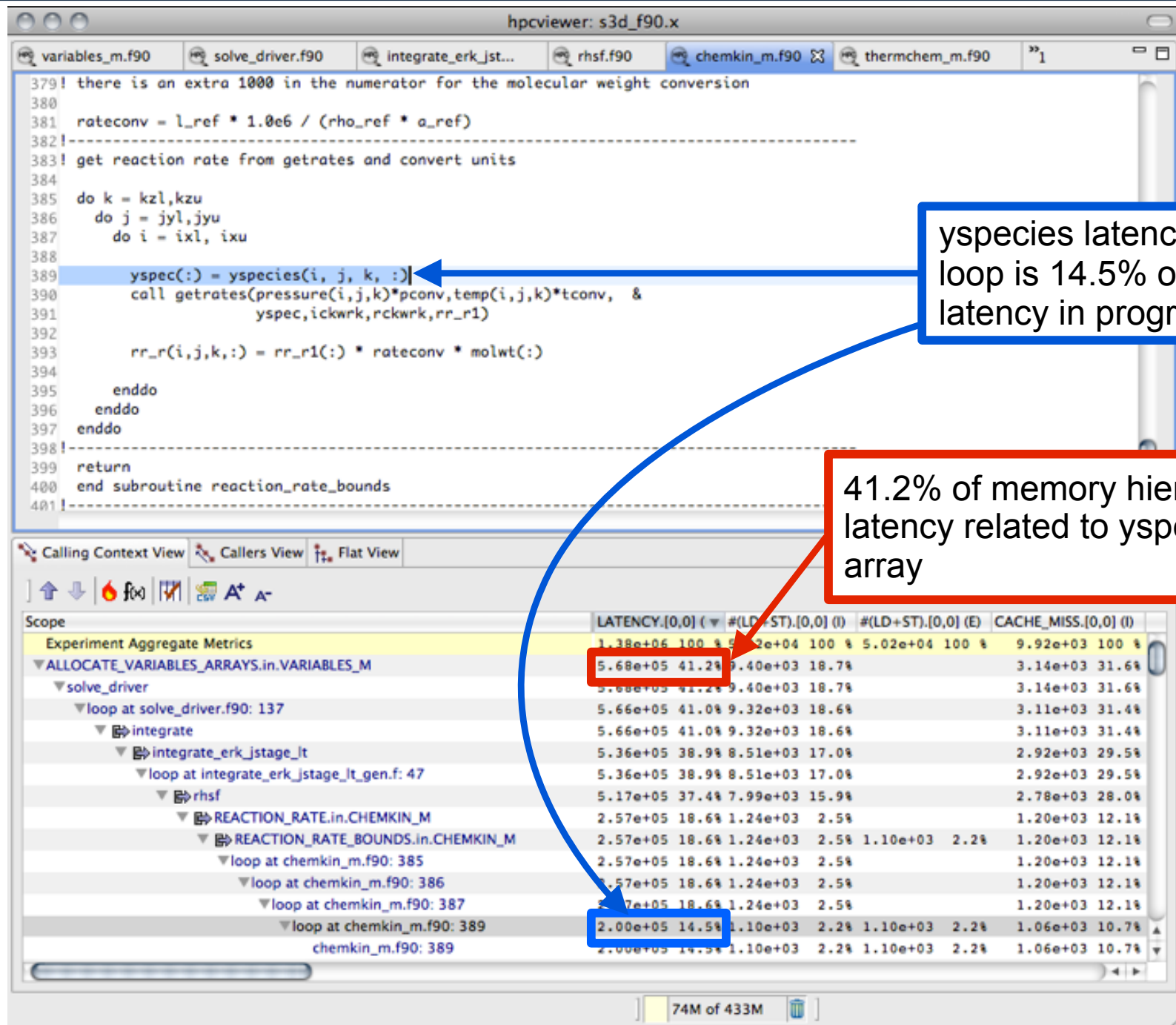


- **LLNL's LULESH:** identified that dynamic memory allocation using cudaMalloc and cudaFree accounted for 90% of the idleness of the GPU

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**