

Gaining Insight into Parallel Program Performance using HPCToolkit

John Mellor-Crummey
Department of Computer Science
Rice University

<http://hpctoolkit.org>



Acknowledgments

- **Current funding**
 - DOE Exascale Computing Project (Subcontract 400015182)
 - NSF Software Infrastructure for Sustained Innovation (Collaborative Agreement 1450273)
 - ANL (Subcontract 4F-30241)
 - LLNL (Subcontract B633244)
- **Project team**
 - **Research Staff**
 - Laksono Adhianto, Mark Krentel, Scott Warren, Xiaozhu Meng
 - **Grad students:**
 - Keren Zhou, Lai Wei, Jonathon Anderson, Vladimir Indjic
 - **Undergraduates:**
 - Tijana Jovanovic, Aleksa Simovic

Challenges for Computational Scientists

- **Rapidly evolving platforms and applications**
 - **architecture**
 - rapidly changing designs for compute nodes
 - significant architectural diversity
 - multicore, manycore, accelerators
 - increasing parallelism within nodes
 - **applications**
 - exploit threaded parallelism in addition to MPI
 - leverage vector parallelism
 - augment computational capabilities
- **Computational scientists need to**
 - adapt codes to changes in emerging architectures
 - improve code 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
- **Supercomputer platforms compound the complexity**
 - unique hardware & microkernel-based operating systems
 - multifaceted performance concerns
 - computation
 - data movement
 - communication
 - I/O

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 petascale and beyond**

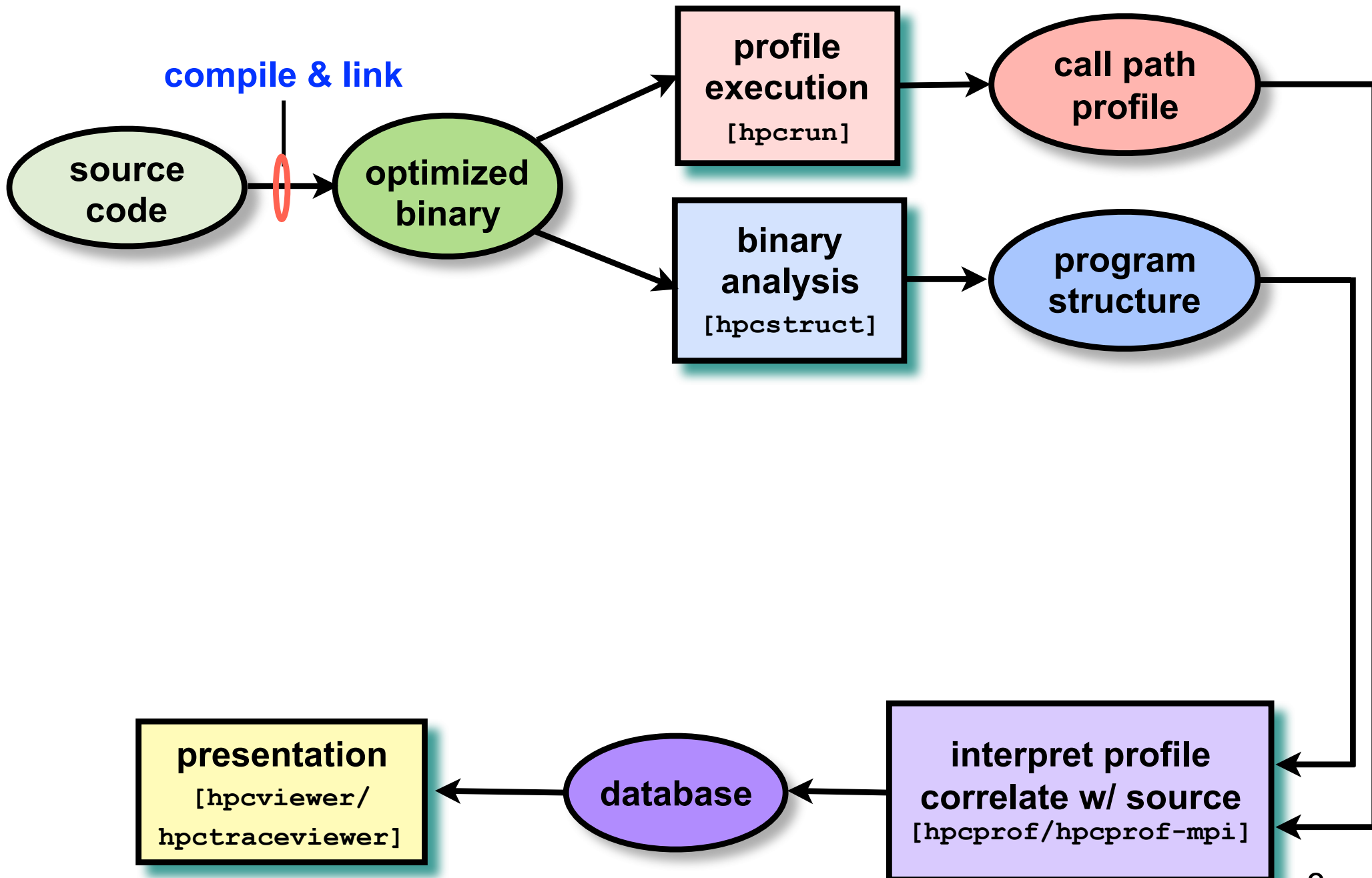
Outline

- **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 OpenMP performance**
 - blame shifting
 - assessing variability across threads and ranks
- **Understanding GPU-accelerated codes**
- **Other capabilities**
- **Ongoing work and future plans**

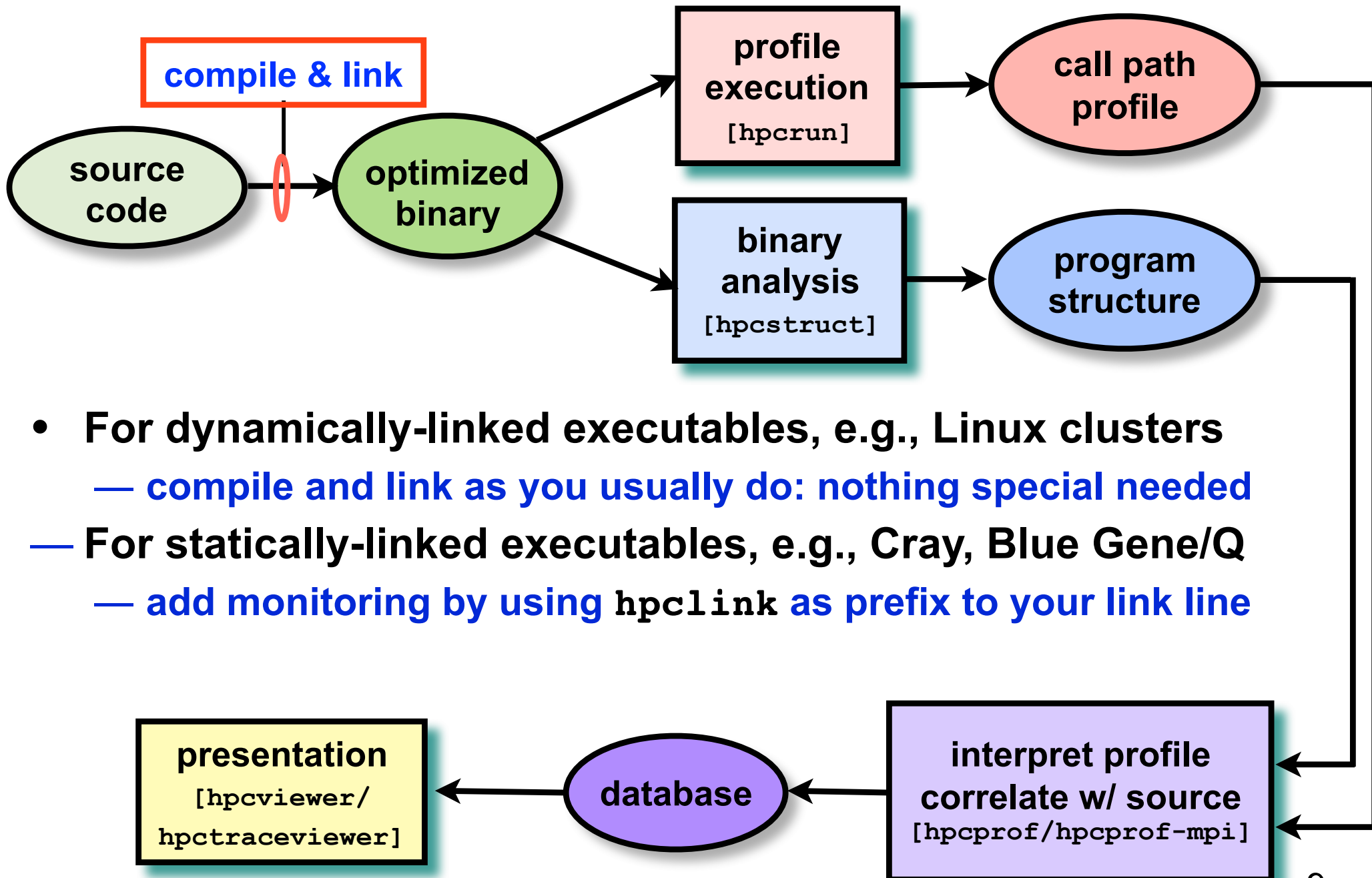
Rice University's HPCToolkit

- **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** often 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**
 - **identify costs of interest and drill down to causes**
 - **up and down call chains**
 - **over time**

HPCToolkit Workflow

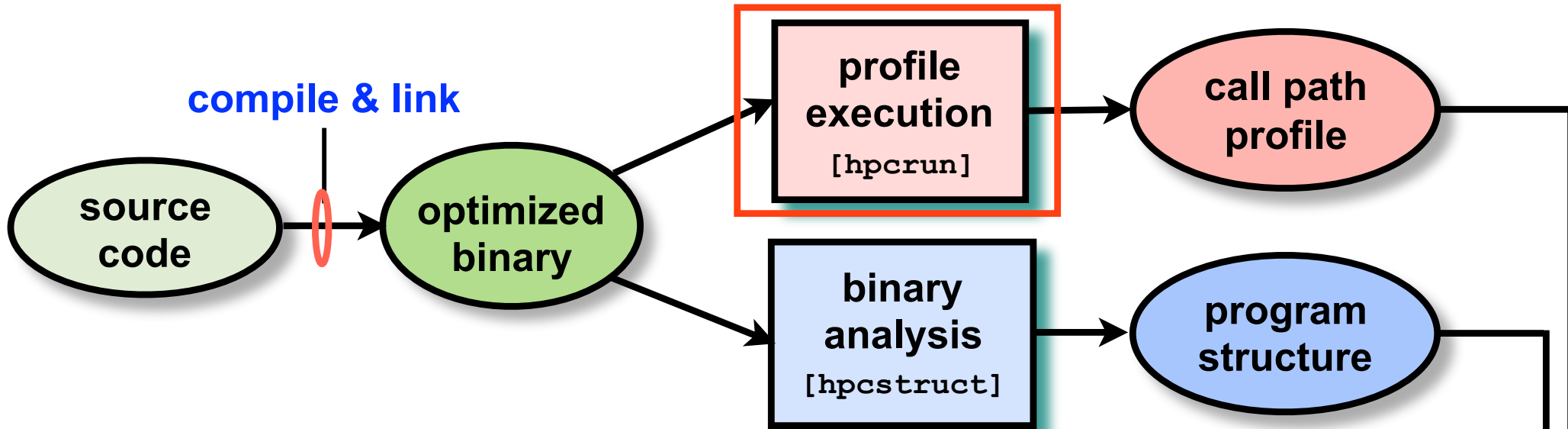


HPCToolkit Workflow



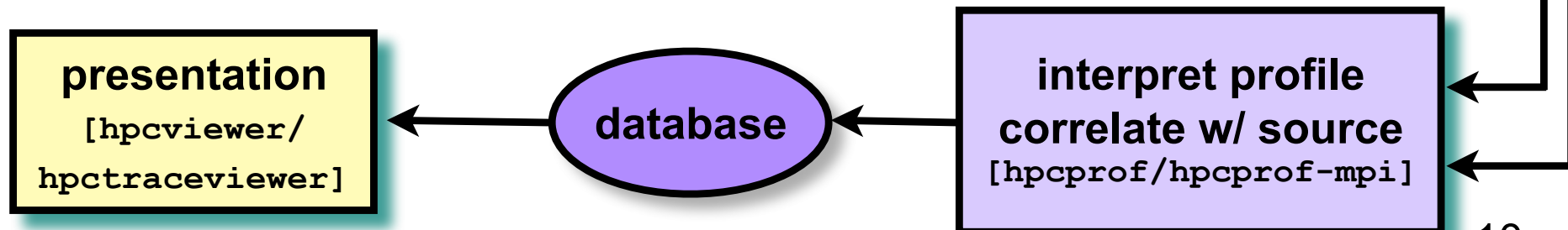
- For dynamically-linked executables, e.g., Linux clusters
 - **compile and link as you usually do: nothing special needed**
- For statically-linked executables, e.g., Cray, Blue Gene/Q
 - **add monitoring by using `hpcLink` as prefix to your link line**

HPCToolkit Workflow



Measure execution unobtrusively

- launch optimized application binaries
 - dynamically-linked: launch with `hpcrun`, arguments control monitoring
 - statically-linked: environment variables control monitoring
- collect statistical call path profiles of events of interest



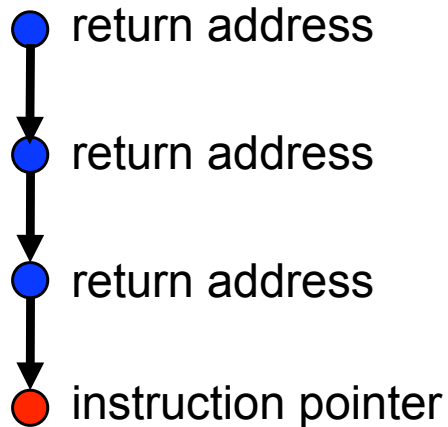
Call Path Profiling

Measure and attribute costs in context

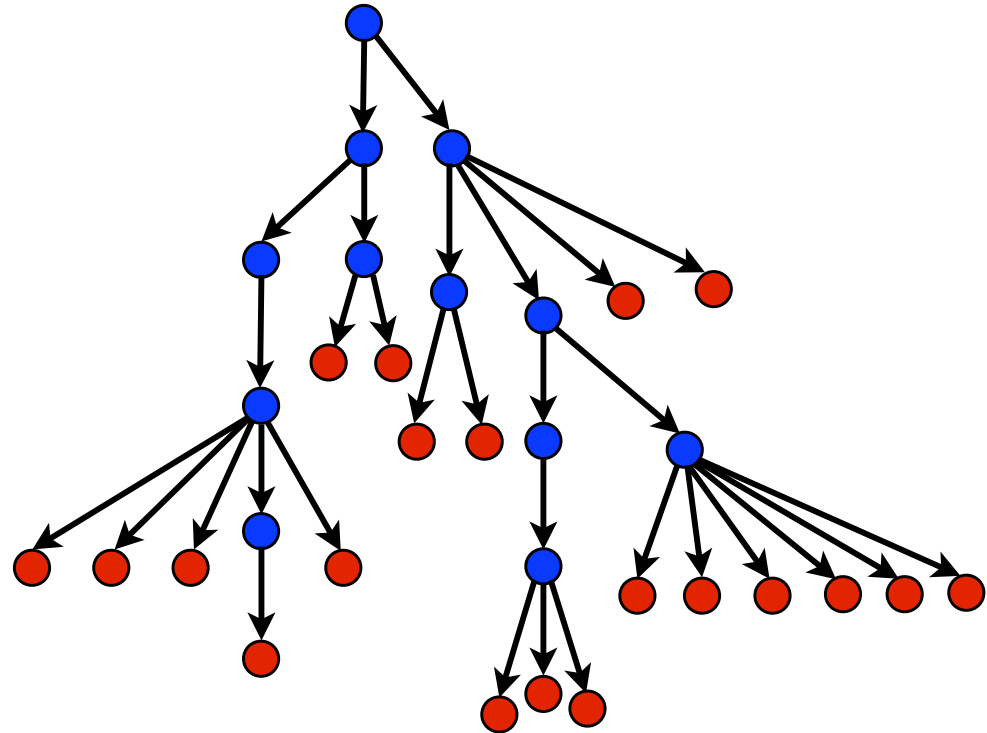
sample timer or hardware counter overflows

gather calling context using stack unwinding

Call path sample

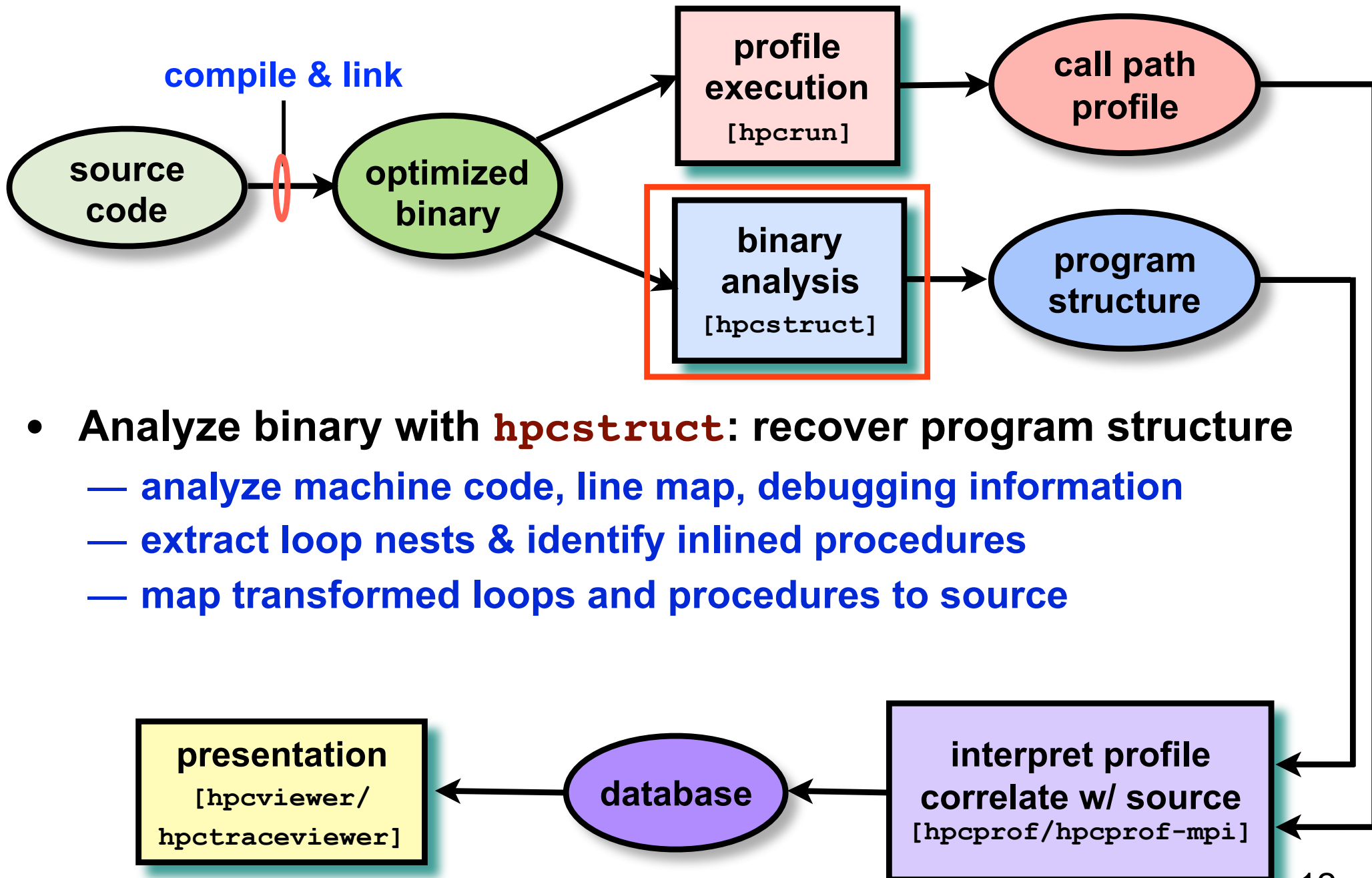


Calling context tree



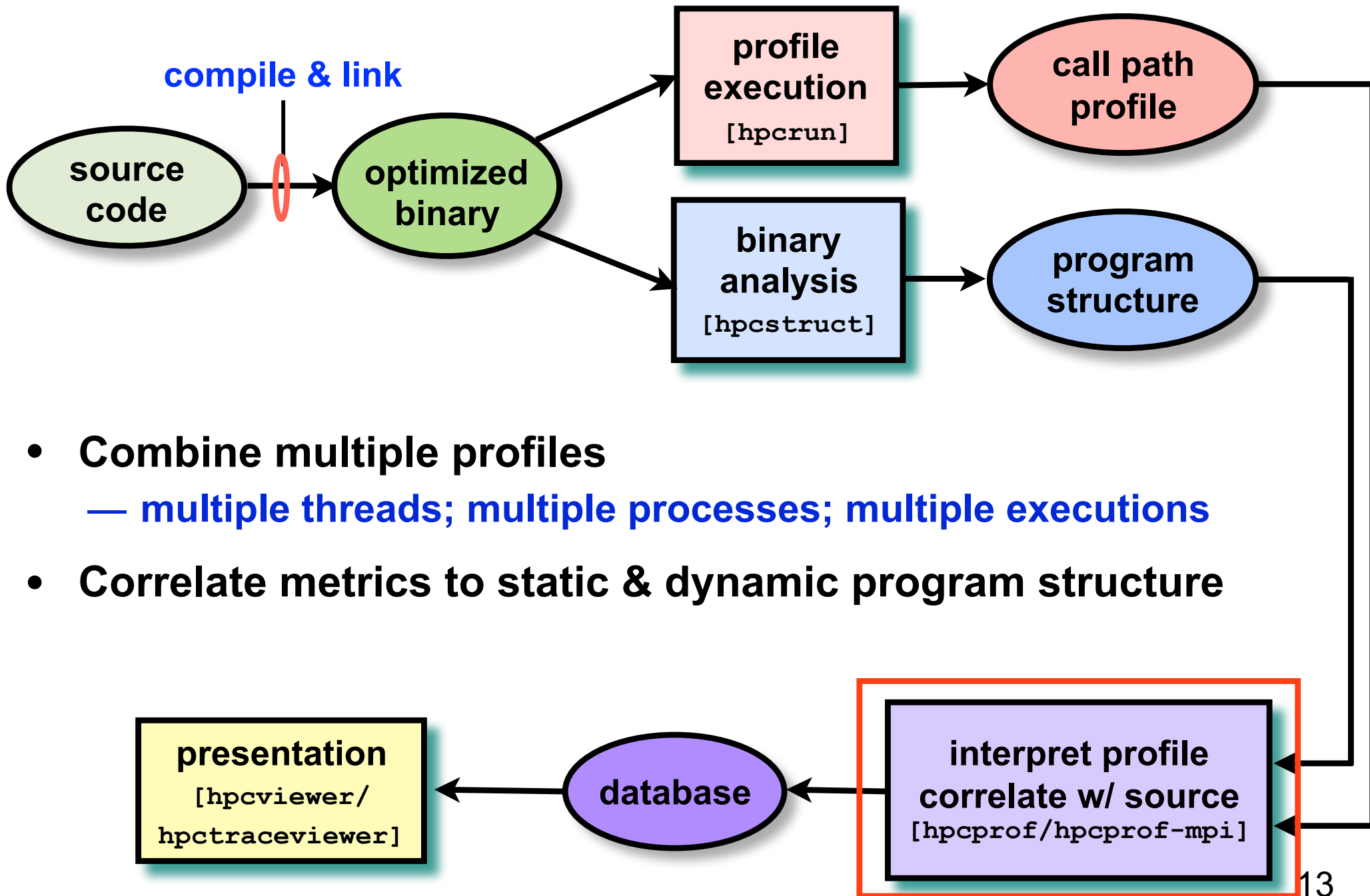
**Overhead proportional to sampling frequency...
...not call frequency**

HPCToolkit Workflow



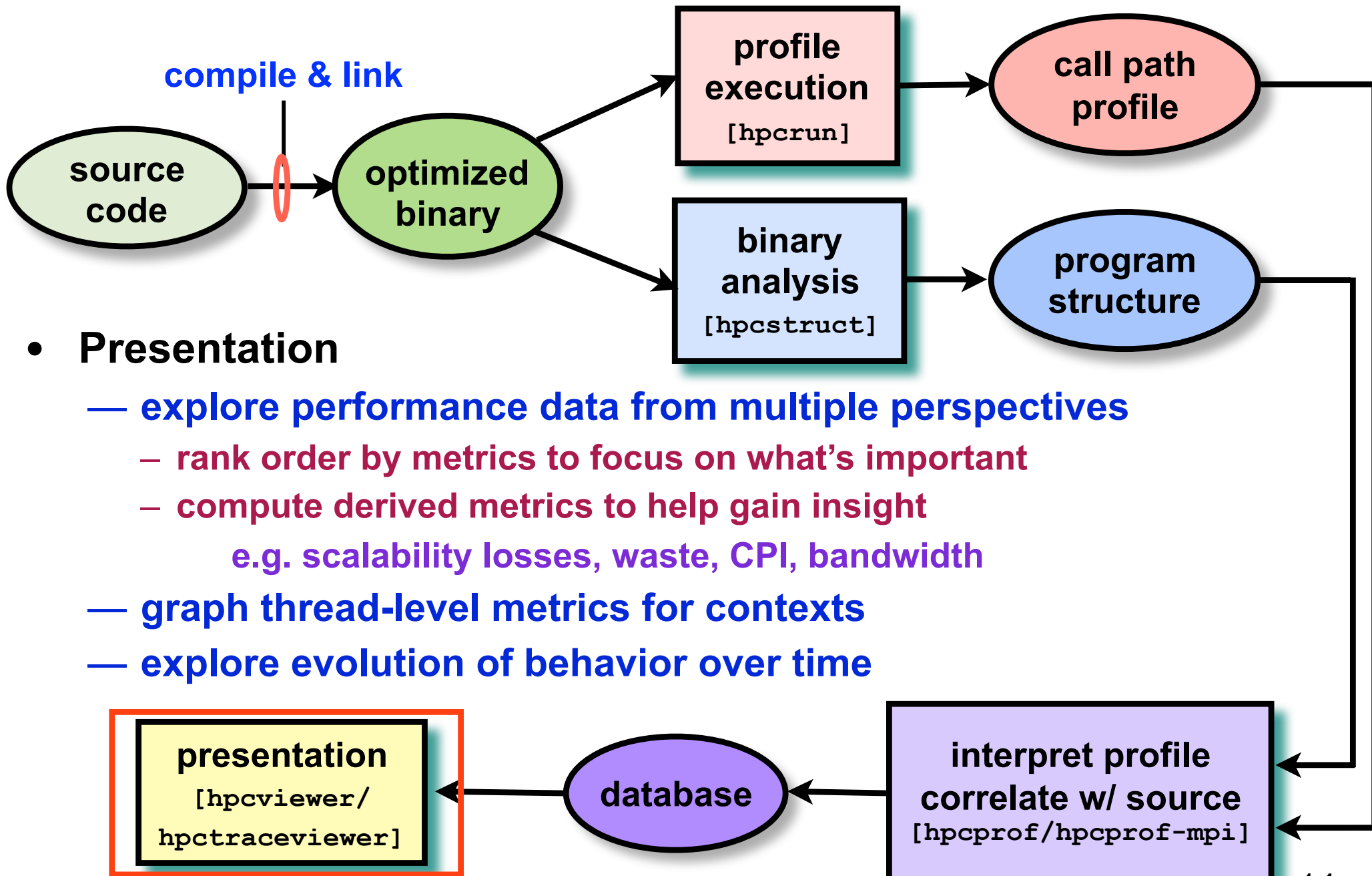
- Analyze binary with **hpcstruct**: recover program structure
 - analyze machine code, line map, debugging information
 - extract loop nests & identify inlined procedures
 - map transformed loops and procedures to source

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

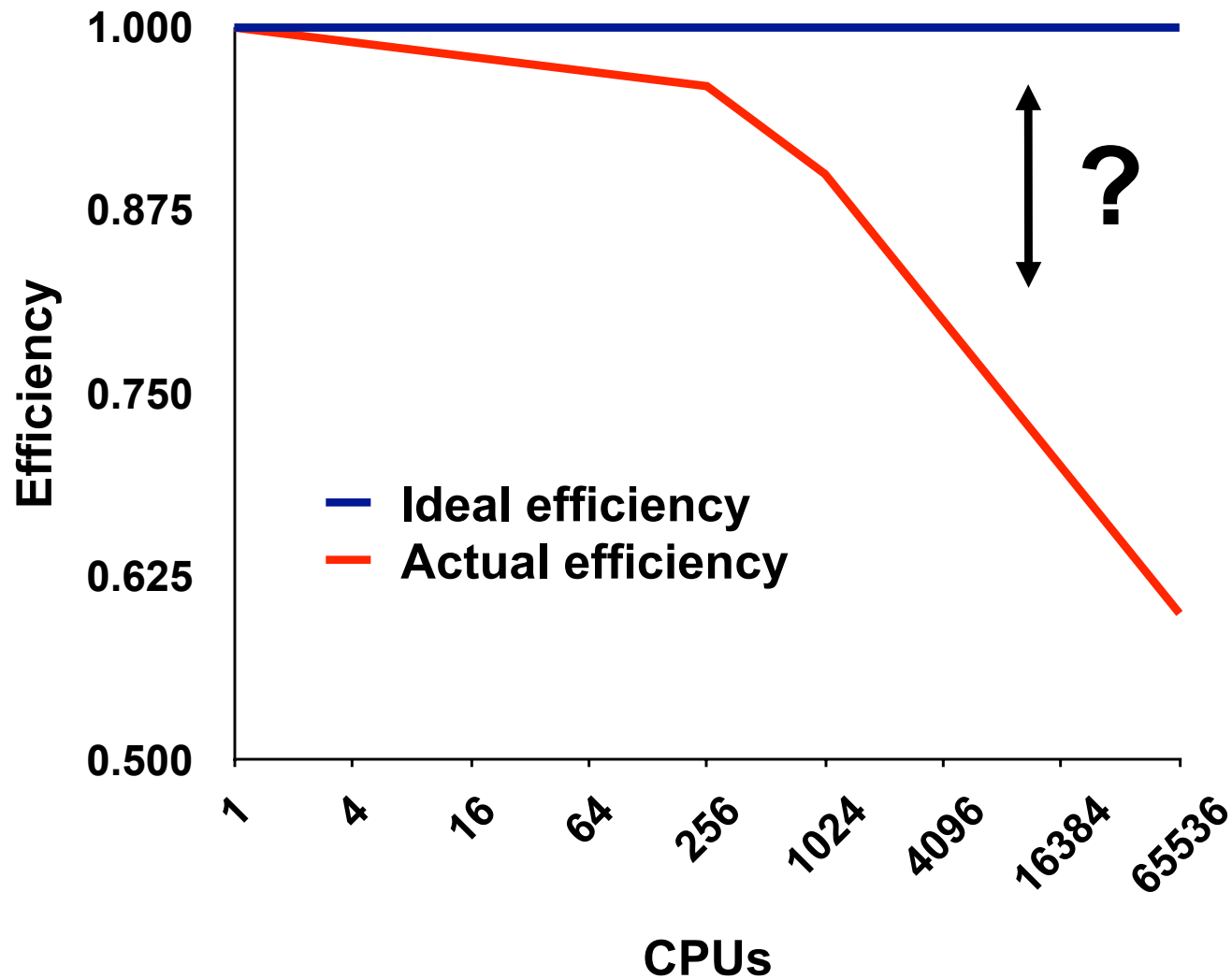
Code-centric Analysis with hpcviewer

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

The screenshot displays the hpcviewer interface for the application 'lulesh-RAJA-parallel.exe'. The top pane shows the source code for 'forall_generic.hxx' with a red box labeled 'source pane' highlighting the function definition. Below the code is a 'view control' bar with buttons for 'Calling Context View', 'Callers View', and 'Flat View'. A 'metric display' bar contains icons for various metrics, with a red box labeled 'metric display' pointing to the 'f(x)' icon. The main area is a 'navigation pane' showing a tree view of the execution context, with a red box labeled 'navigation pane' pointing to the tree structure. The right side of the interface is a 'metric pane' table showing performance metrics for various scopes, with a red box labeled 'metric pane' pointing to the table. The table has three columns: 'Scope', 'REALTIME (usec):Sum (I)', and 'REALTIME (usec):Sum (E)'. The table data is as follows:

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: [] LagrangeLeapFrog(Domain*)	1.44e+08 63.8%	
2715: [] LagrangeNodal(Domain*)	8	
1554: [] CalcForceForNodes(Domain*)	8	
1469: CalcVolumeForceForElems(Domain*)	8.25e+07 36.5%	
1454: [] CalcHourglassControlForElems(Domain*, double*, double)	5.15e+07 22.8%	
1399: [] CalcFBHourglassForceForElems(int*, double*, double*, double*, double*, double)	3.10e+07 13.7%	
1187: [] void RAJA::forall<RAJA::IndexSet::ExecPolicy<RAJA::seq_segit, RAJA::omp_parallel_for_exec, C2lcFBHourglassForceForElems(Domain*, double*, double)>>(const INDEXSET_T& iset, LOOP_BODY loop_body)	2.43e+07 10.8%	
405: [] void RAJA::forall<RAJA::omp_parallel_for_exec, C2lcFBHourglassForceForElems(Domain*, double*, double)>(const INDEXSET_T& iset, LOOP_BODY loop_body)	2.43e+07 10.8%	
loop at forall_seq_any.hxx: 498	2.43e+07 10.8%	
505: [] void RAJA::forall<CalcFBHourglassForceForElems(int*, double*, double*, double*, double*, double)>(const INDEXSET_T& iset, LOOP_BODY loop_body)	2.43e+07 10.8%	1.00e+03 0.0%
89: outline forall_omp_any.hxx:89 (0x423620)	2.42e+07 10.7%	3.91e+04 0.0%
loop at forall_omp_any.hxx: 90	2.42e+07 10.7%	3.41e+04 0.0%
91: [] CalcFBHourglassForceForElems(int*, double*, double*, double*, double*, double)	2.42e+07 10.7%	9.84e+06 4.3%
1300: [] CalcElemFBHourglassForce(double*, double*, double*, double)	1.11e+07 4.9%	1.11e+07 4.9%
1260: [] CBRT(double)	3.27e+06 1.4%	2.00e+05 0.1%

The Problem of Scaling



Note: higher is better

Goal: Automatic Scalability Analysis

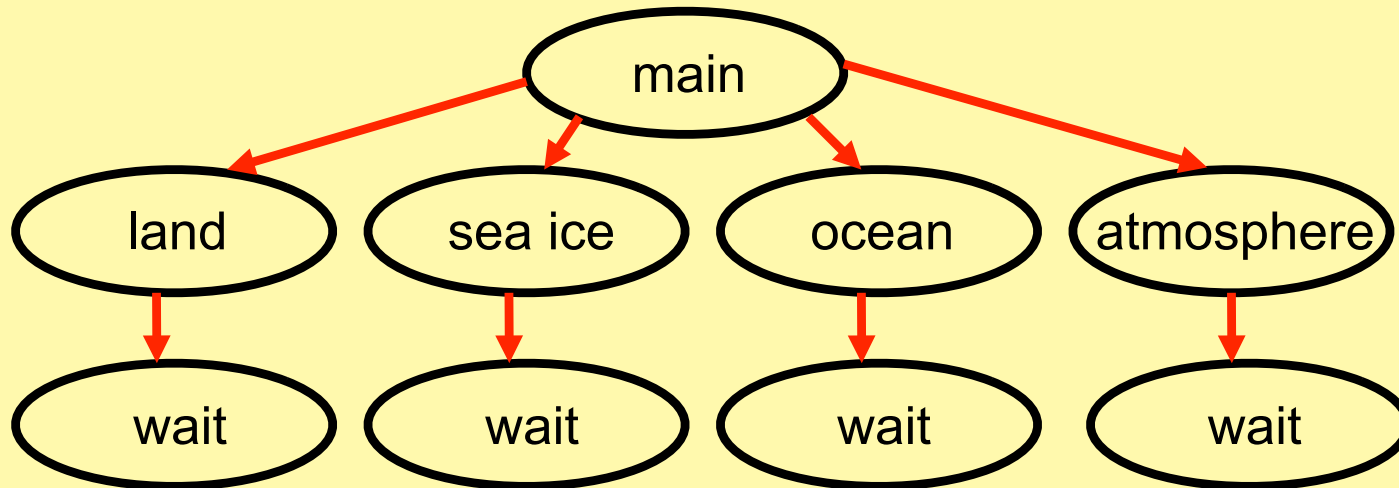
- Pinpoint scalability bottlenecks
- Guide user to problems
- Quantify the magnitude of each problem
- Diagnose the nature of the problem

Challenges for Pinpointing Scalability Bottlenecks

- **Parallel applications**

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

Example climate code skeleton



- **Monitoring**

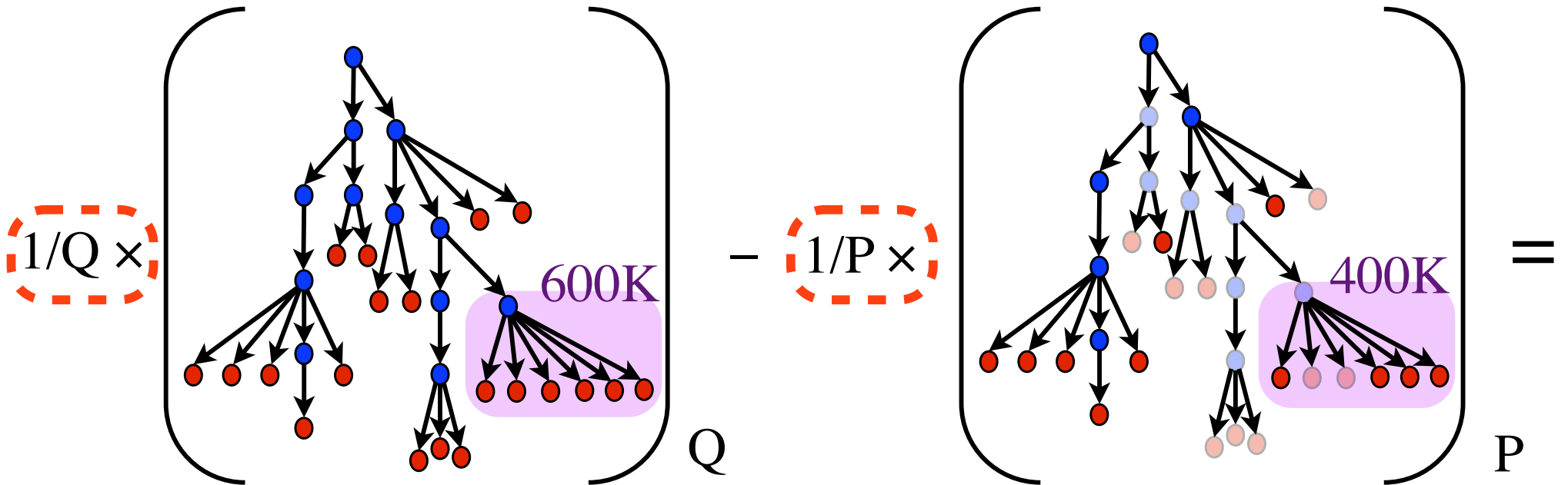
- bottleneck nature: computation, data movement, synchronization?
- 2 pragmatic constraints
 - acceptable data volume
 - low perturbation for use in production runs

Performance Analysis with Expectations

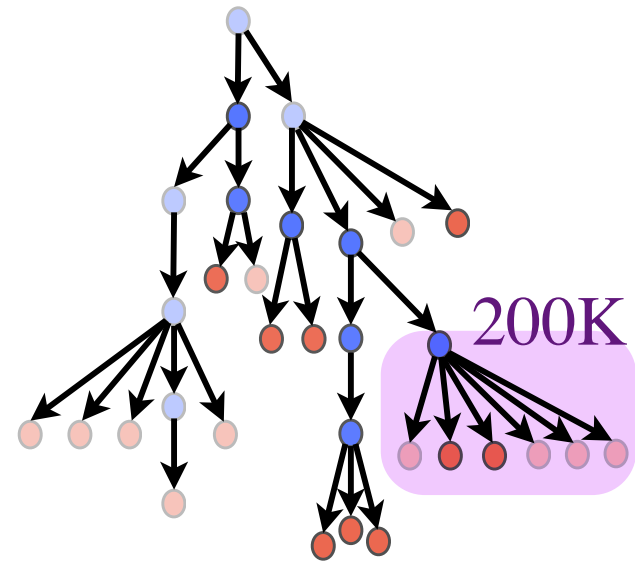
- You have performance expectations for your parallel code
 - strong scaling: linear speedup
 - weak scaling: constant execution time

- Put your expectations to work
 - measure performance under different conditions
 - e.g. different levels of parallelism or different inputs
 - 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 weak scaling



Scalability Analysis Demo

Code:

Simulation:

Platform:

Experiment:

Scaling type:

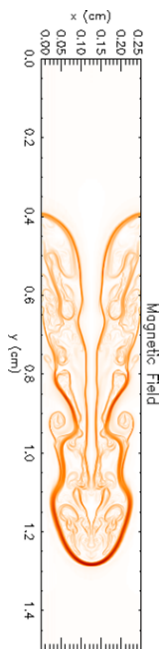
University of Chicago FLASH

white dwarf detonation

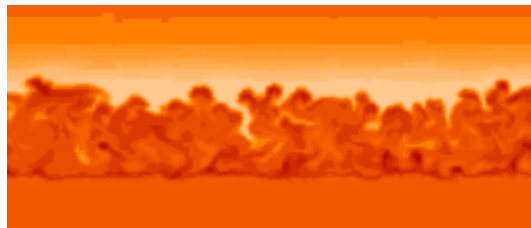
Blue Gene/P

8192 vs. 256 processors

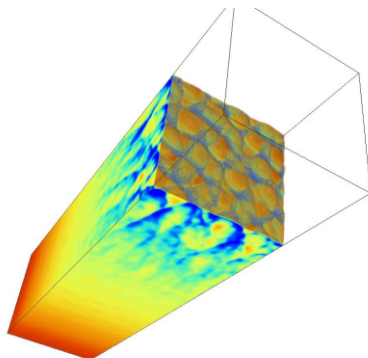
weak



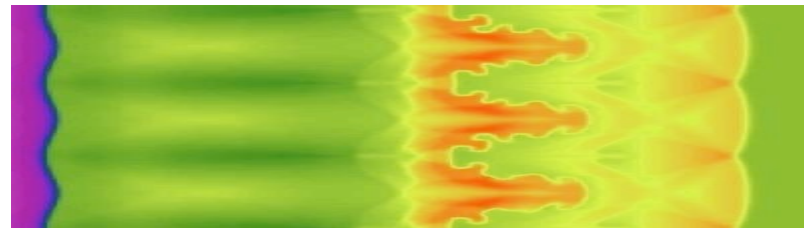
*Magnetic
Rayleigh-Taylor*



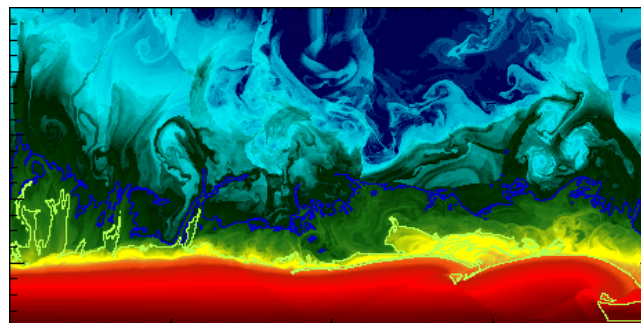
Nova outbursts on white dwarfs



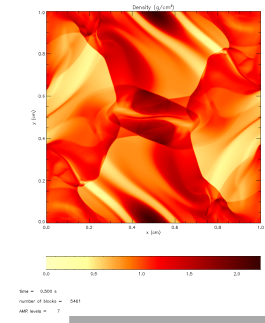
Cellular detonation



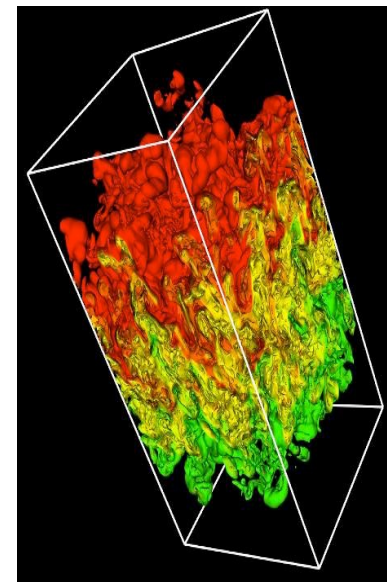
Laser-driven shock instabilities



Helium burning on neutron stars



*Orzag/Tang MHD
vortex*



Rayleigh-Taylor instability

Scalability Analysis of Flash (Demo)

hpcviewer: FLASH/white dwarf: IBM BG/P, weak 256->8192

Driver_initFlash.F90 local_tree_build.F90

```

206 !-----First pass only add lrefine = 1 blocks to tree(s)
207 !-----Second pass add the rest of the blocks.
208     Do ipass = 1,2
209
210         lnblocks_old = lnblocks
211         proc = mype
212 !-----Loop through all processors
213     Do iproc = 0, nprocs-1
214
215         If (iproc == 0) Then
216             off_proc = .False.
217         Else
  
```

Calling Context View Callers View Flat View

Scope % scalability loss 256/WALLCLOCK (u)

Scope	% scalability loss	256/WALLCLOCK (u)
Experiment Aggregate Metrics	2.46e+01 100 %	5.07e+08
flash	2.46e+01 100 %	5.07e+08
driver_evolveflash	1.41e+01 57.5%	4.46e+08
driver_initflash	1.04e+01 42.5%	6.02e+07
grid_initdomain	8.58e+00 34.9%	3.45e+07
gr_expanddomain	8.58e+00 34.9%	3.45e+07
loop at gr_expandDomain.F90: 119	6.85e+00 27.9%	3.42e+07
amr_refine_derefine	5.56e+00 22.6%	2.87e+06
amr_morton_process	5.45e+00 22.2%	9.75e+05
find_surrblks	5.18e+00 21.1%	8.40e+05
local_tree_build	5.18e+00 21.1%	8.25e+05
loop at local_tree_build.F90: 211	5.18e+00 21.1%	8.25e+05
loop at local_tree_build.F90: 216	5.18e+00 21.1%	8.25e+05
loop at local_tree_build.F90: 286	1.14e+00 4.6%	2.55e+05
pmpi_sendrecv_replace	5.47e-01 2.2%	5.00e+04

Scalability Analysis

- Difference call path profile from two executions
 - different number of nodes
 - different number of threads
- Pinpoint and quantify scalability bottlenecks within and across nodes

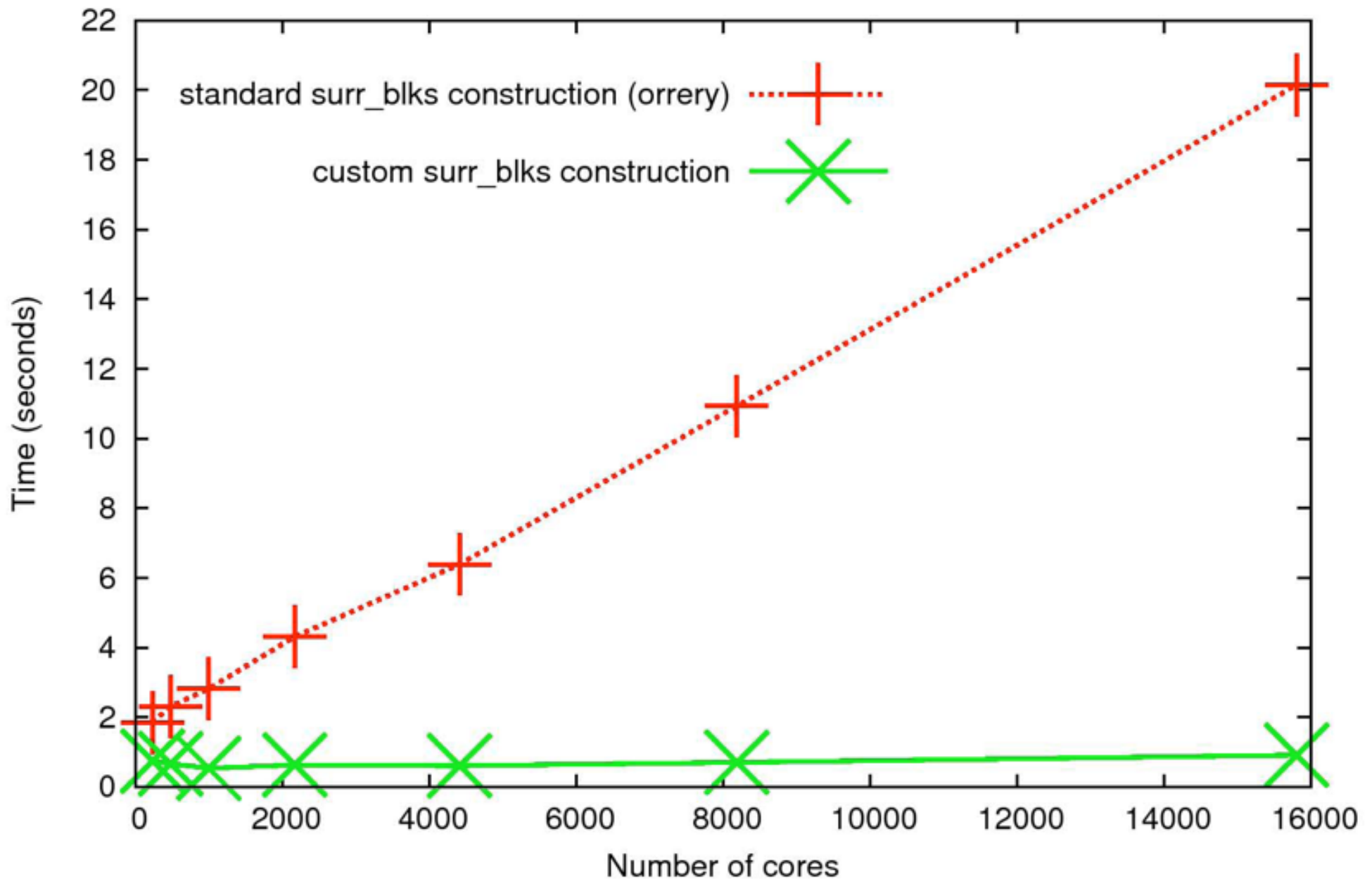
hpcviewer: FLASH/white dwarf: IBM BG/P, weak 256->8192

```
Driver_initFlash.F90 local_tree_build.F90
206 !-----First pass only add lrefine = 1 blocks to tree(s)
207 !-----Second pass add the rest of the blocks.
208     Do ipass = 1,2
209
210     lnblocks_old = lnblocks
211     proc = mype
212 !-----Loop through all processors
213     Do iproc = 0, nprocs-1
214
215     If (iproc == 0) Then
216         off_proc = .False.
217     Else
```

significant scaling losses caused by passing data around a ring of processors

Scope	% scalability loss	256/WALLCLOCK (u)
Experiment Aggregate Metrics	2.46e+01 100 %	5.07e+08
flash	2.46e+01 100 %	5.07e+08
driver_evolveflash	1.41e+01 57.5%	4.46e+08
driver_initflash	1.04e+01 42.5%	6.02e+07
grid_initdomain	8.58e+00 34.9%	3.45e+07
gr_expanddomain	8.58e+00 34.9%	3.45e+07
loop at gr_expandDomain.F90: 119	6.85e+00 27.9%	3.42e+07
amr_refine_derefine	5.56e+00 22.6%	2.87e+06
amr_morton_process	5.45e+00 22.2%	9.75e+05
find_surrblks	5.18e+00 21.1%	8.40e+05
local_tree_build	5.18e+00 21.1%	8.25e+05
loop at local_tree_build.F90: 211	5.18e+00 21.1%	8.25e+05
loop at local_tree_build.F90: 216	5.18e+00 21.1%	8.25e+05
loop at local_tree_build.F90: 286	1.14e+00 4.6%	2.55e+05
pmpi_sendrecv_replace	5.47e-01 2.2%	5.00e+04

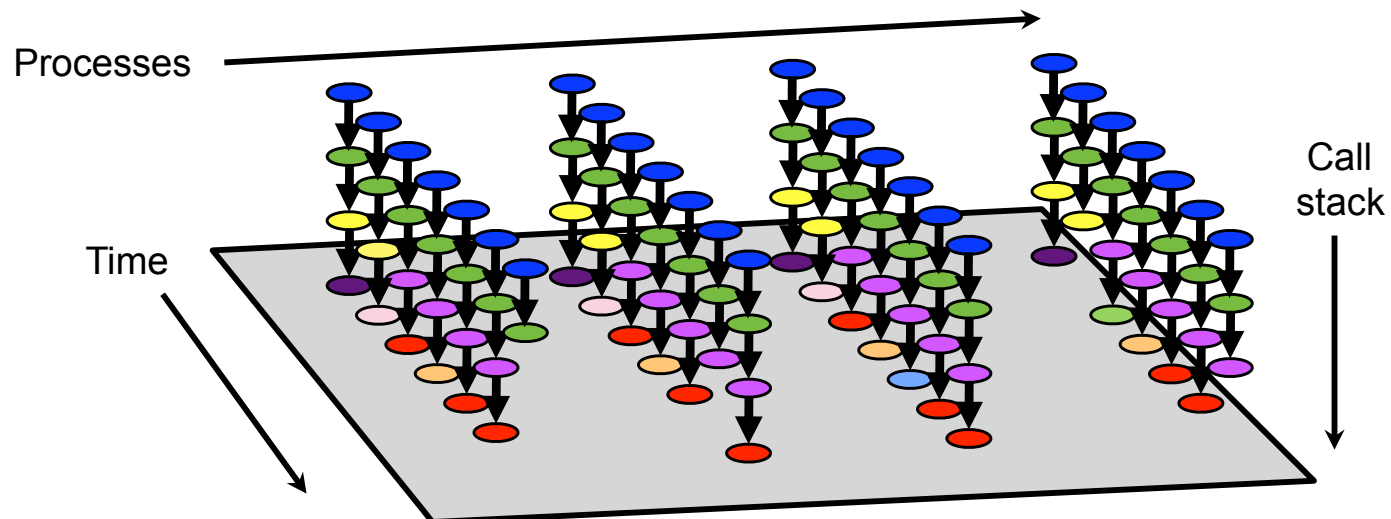
Improved Flash Scaling of AMR Setup



Graph courtesy of Anshu Dubey, U Chicago

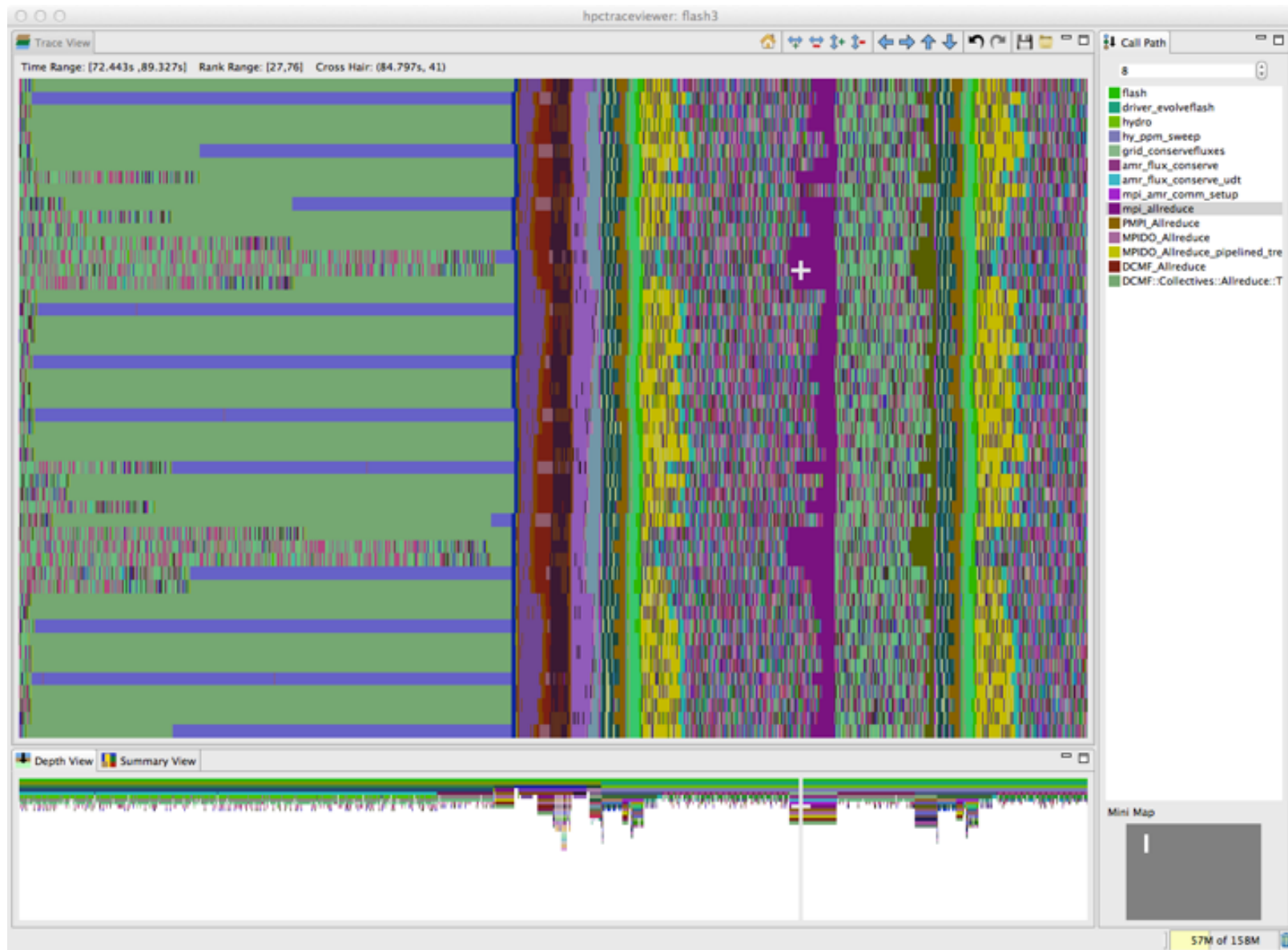
Understanding Temporal Behavior

- 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



hpctraceviewer: detail of FLASH@256PE

Time-centric analysis: load imbalance among threads appears as different lengths of colored bands along the x axis



OpenMP: A Challenge for Tools

- Large gap between between threaded programming models and their implementations

The screenshot shows the hpcviewer interface for the file LULESH_OMP.cpp. The source code is visible at the top, with a parallel region starting at line 1291. Below the code, the 'Calling Context View' is displayed, showing a tree of function calls. A red box highlights a specific path in the tree: L_Z28CalcFBHourglassForceForElemsPdS_S_S_S_S_d_1291__par_loop0_2_276, L_Z22CalcKinematicsForElemsid_1931__par_loop0_2_855, L_Z28CalcHourglassControlForElemsPdd_1516__par_loop0_2_424, L_Z23IntegrateStressForElemsiPdS_S_S_864__par_loop0_2_125, and L_Z31CalcMonotonicQGradientsForElemsv_2040__par_loop0_2_965. The table below the tree shows performance metrics for each node.

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: __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_Z23IntegrateStressForElemsiPdS_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%	

User-level calling context for code in OpenMP parallel regions and tasks executed by worker threads is not readily available

- Runtime support is necessary for tools to bridge the gap

Challenges for OpenMP Node Programs

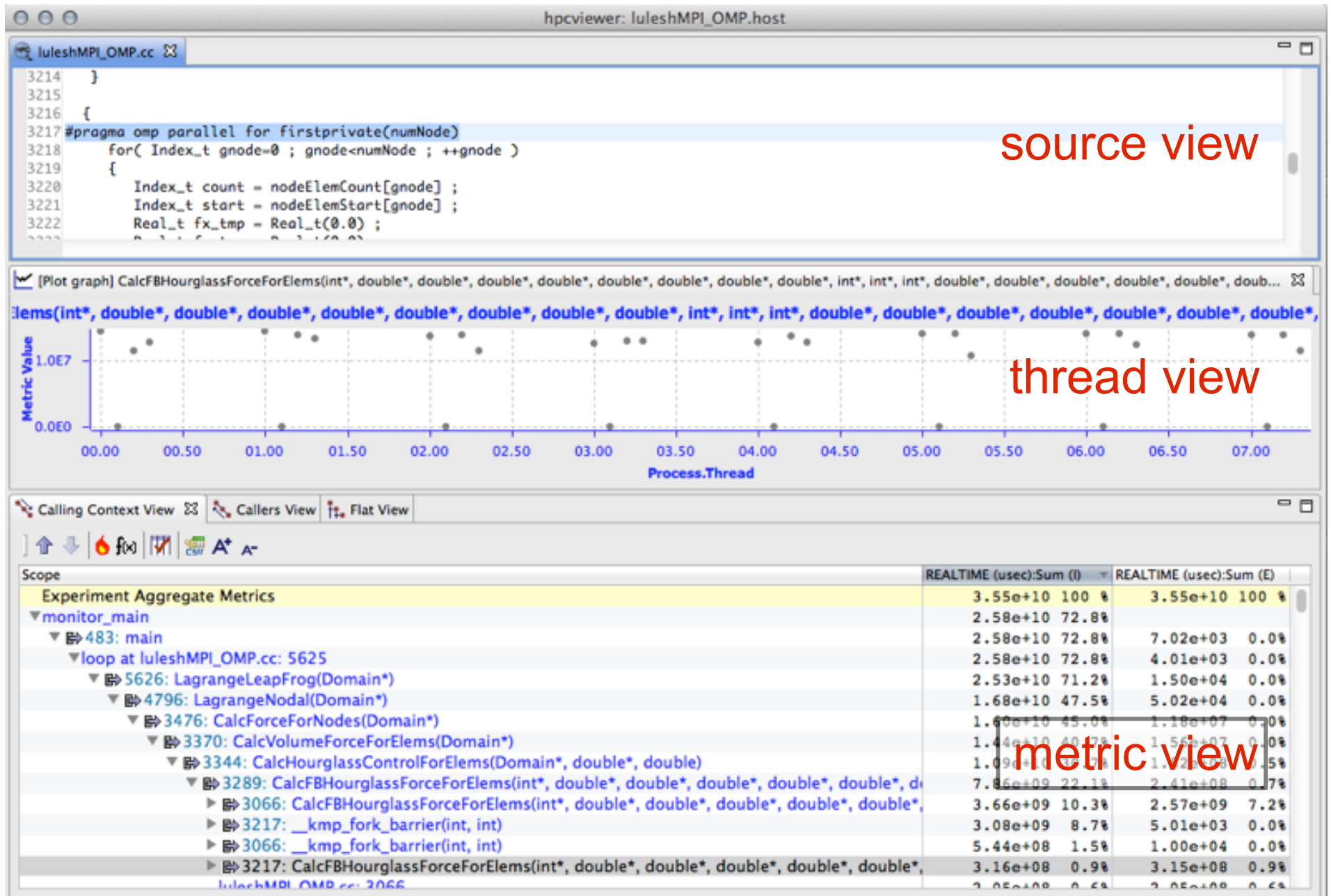
- **Tools provide implementation-level view of OpenMP threads**
 - **asymmetric threads**
 - **master thread**
 - **worker thread**
 - **run-time frames are interspersed with user code**
- **Hard to understand causes of idleness**
 - **long serial sections**
 - **load imbalance in parallel regions**
 - **waiting for critical sections or locks**

OMPT: An OpenMP Tools API

- **Goal: a standardized tool interface for OpenMP**
 - prerequisite for portable tools
 - missing piece of the OpenMP language standard
- **Design objectives**
 - enable tools to measure and attribute costs to application source and runtime system
 - support low-overhead tools based on asynchronous sampling
 - attribute to user-level calling contexts
 - associate a thread's activity at any point with a descriptive state
 - minimize overhead if OMPT interface is not in use
 - features that may increase overhead are optional
 - define interface for trace-based performance tools
 - don't impose an unreasonable development burden
 - runtime implementers
 - tool developers

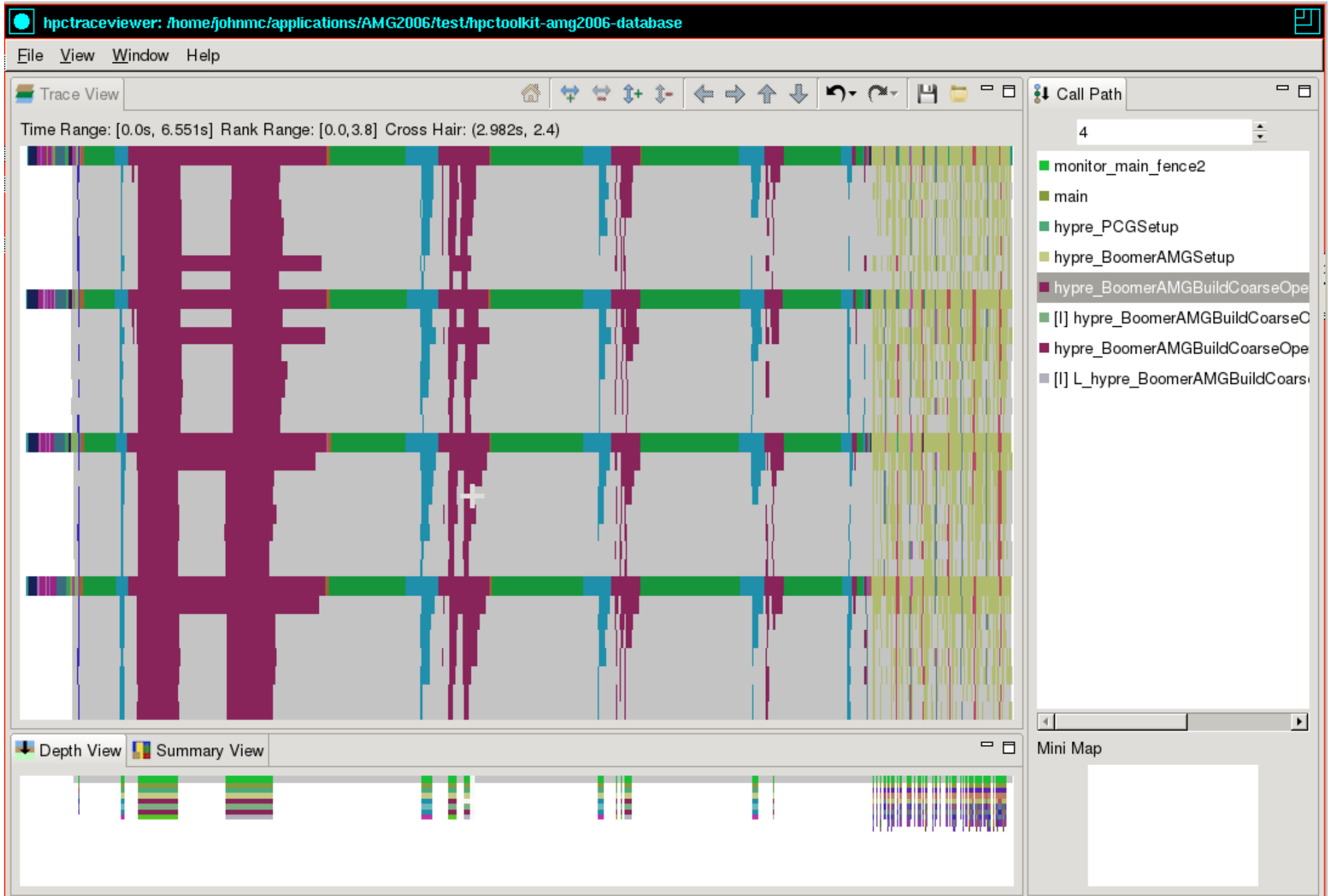
Integrated View of MPI+OpenMP with OMPT

LLNL's luleshMPI_OMP (8 MPI x 3 OMP), 30, REALTIME@1000



2 18-core Haswell
4 MPI ranks
6+3 threads per rank

Case Study: AMG2006



OpenMP Tool API Status

- **HPCToolkit supports OpenMP 5.0 OMPT**
- **OMPT prototype implementations**
 - LLVM (emerging: OpenMP 5.0)
 - interoperable with GNU, Intel compilers
 - IBM LOMP (currently targets OpenMP 4.5)
- **Ongoing work**
 - refining OpenMP 5.0 OMPT support in LLVM OpenMP
 - refining OpenMP 5.0 OMPT support in HPCToolkit
 - asynchronous call stack assembly for lightweight monitoring

HPCToolkit Measurement on NVIDIA GPUs

- **Monitor GPU events using NVIDIA's CUPTI API**
 - kernel invocations
 - explicit data copies
 - implicit data copies (page faults)
 - PC samples
- **Register for callbacks associated with target devices**
 - device initialization/finalization
 - enable selected monitoring upon initialization
 - device load/unload
 - upon load: relocate CUBIN to interpret PC samples
 - add CUBINs to the load map
 - buffer request/complete
 - request: supply a buffer for the GPU to record events
 - complete: process CUPTI event records into a profile

A Simple GPU-accelerated Example

Two threads launch vecAdd kernels concurrently

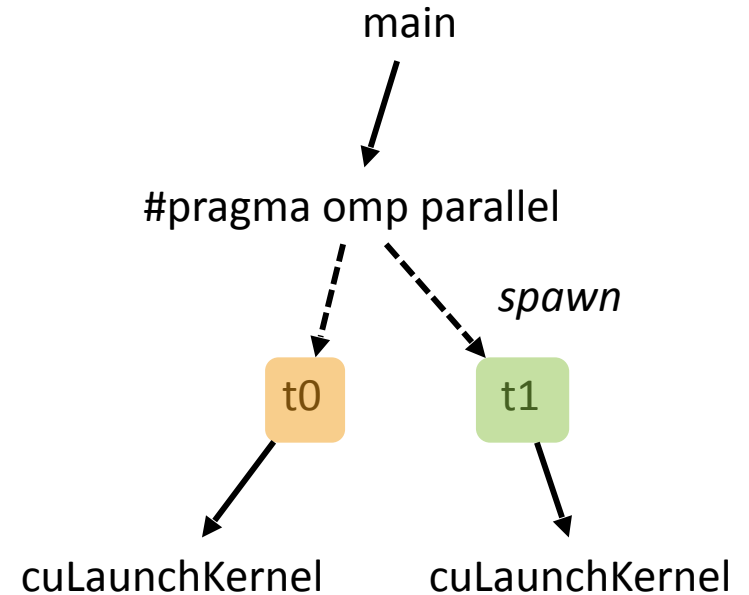
```
1  #omp parallel num_threads(2)
2  cuLaunchKernel(vecAdd, ...)
3
4  int __noinline__ add(int a, int b) {
5      return a + b;
6  }
7
8  void vecAdd(int *l, int *r, int *p, size_t iter1, size_t iter2) {
9      size_t idx = blockDim.x * blockIdx.x + threadIdx.x;
10     for (size_t i = 0; i < iter1; ++i) {
11         p[idx] = add(l[idx], r[idx]);
12     }
13     for (size_t i = 0; i < iter2; ++i) {
14         p[idx] = add(l[idx], r[idx]);
15     }
16 }
```

Collect CPU Calling Context for GPU Work

CPU Calling Context

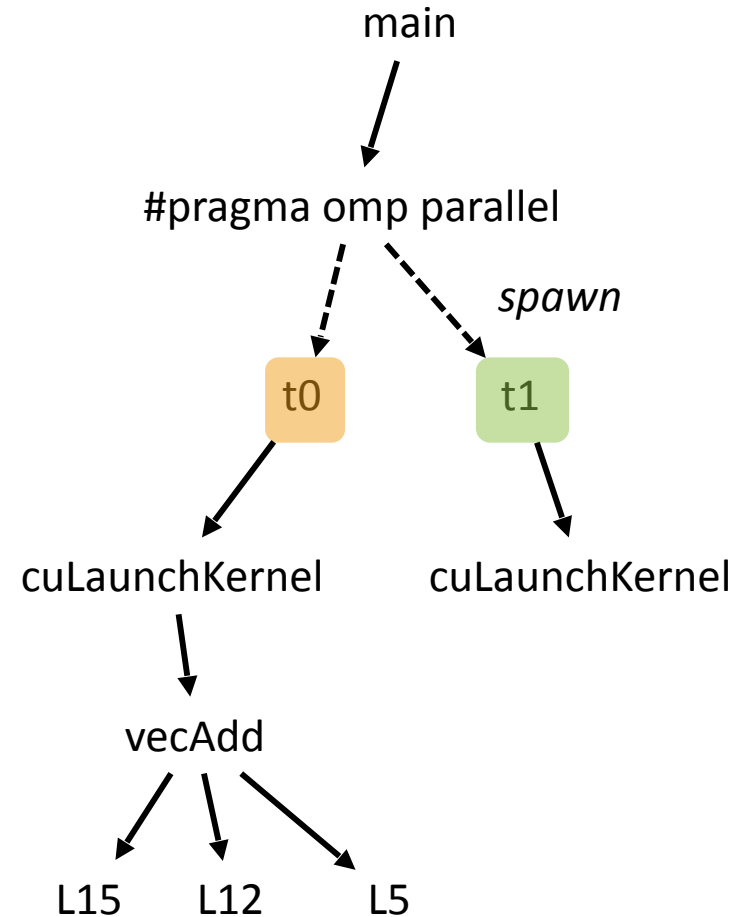
```
1 #omp parallel num_threads(2)
2 cuLaunchKernel(vecAdd, ...)
3
4 int __noinline__ add(int a, int b) {
5     return a + b;
6 }
7
8 void vecAdd(int *l, int *r, int *p,
9     size_t iter1, size_t iter2) {
10     size_t idx = blockDim.x * blockIdx.x
11         + threadIdx.x;
12     for (size_t i = 0; i < iter1; ++i) {
13         p[idx] = add(l[idx], r[idx]);
14     }
15     for (size_t i = 0; i < iter2; ++i) {
16         p[idx] = add(l[idx], r[idx]);
17     }

```



Collecting GPU PC Samples

```
1 #omp parallel num_threads(2)
2 cuLaunchKernel(vecAdd, ...)
3
4 int __noinline__ add(int a, int b) {
5     return a + b;
6 }
7
8 void vecAdd(int *l, int *r, int *p,
9     size_t iter1, size_t iter2) {
10     size_t idx = blockDim.x * blockIdx.x
11         + threadIdx.x;
12     for (size_t i = 0; i < iter1; ++i) {
13         p[idx] = add(l[idx], r[idx]);
14     }
15     for (size_t i = 0; i < iter2; ++i) {
16         p[idx] = add(l[idx], r[idx]);
17     }
18 }
```



Lx: samples collected at Line x

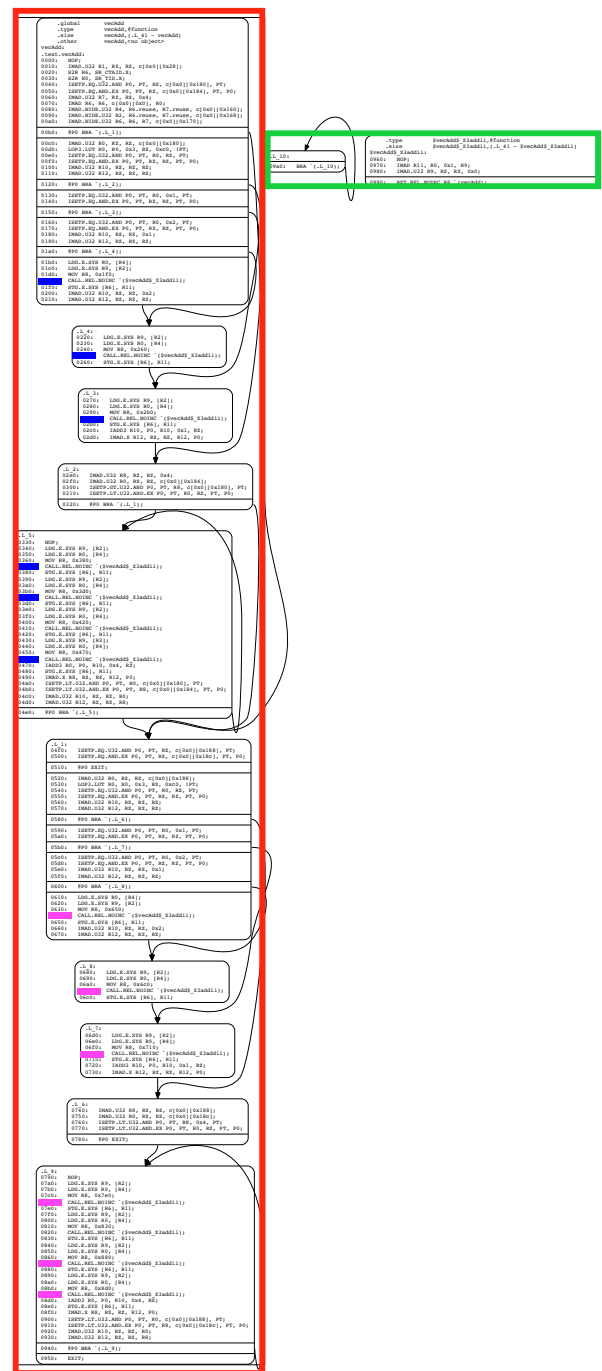
NVIDIA PC Sampling records flat samples

Attribution for GPU binaries

- **Analyze loop nests in NVIDIA CUBIN GPU binaries**
 - **invoke “hpcstruct” on an hpctoolkit measurement directory to analyze any CUBINs collected at runtime’**
 - **results of such analysis will be automatically be integrated into the profiling result**
- **Approximately reconstruct GPU call paths from flat samples by using PC sampling on the GPU**
 - **analyze instructions to identify static calls**
 - **use PC samples of call instructions to help apportion cost of callee to callers**

Optimized GPU Machine Code for VecAdd

```
1 __device__
2 int __attribute__((noinline)) add(int a, int b) {
3     return a + b;
4 }
5
6
7 extern "C"
8 __global__
9 void vecAdd(int *l, int *r, int *p, size_t N, size_t iter1,
10            size_t iter2) {
11     size_t idx = blockDim.x * blockIdx.x + threadIdx.x;
12     for (size_t i = 0; i < iter1; ++i) {
13         p[idx] = add(l[idx], r[idx]);
14     }
15     for (size_t i = 0; i < iter2; ++i) {
16         p[idx] = add(l[idx], r[idx]);
17     }
18 }
```



Profiling Result for VecAdd CUDA Example

3-hpcviewer: main

vecAdd.cu

```

11  for (size_t i = 0; i < iter1; ++i) {
12      p[idx] = add(l[idx], r[idx]);
13  }
14  for (size_t i = 0; i < iter2; ++i) {
15      p[idx] = add(l[idx], r[idx]);
16  }

```

GPU kernel

loop 14

loop 11

device fn calls

device fn calls

Calling Context View

Callers View

Flat View

↑ ↓ 🔥 f(x) 📄 📄 CSV A+ A- ||| 🔍

Scope	GPU_ISAMP.[0,0] (I) ↓	GPU_ISAMP.[0,0] (E)
Experiment Aggregate Metrics	1.78e+07 100 %	1.78e+07 100 %
<program root>	1.78e+07 100 %	
500: main	1.78e+07 100 %	
63: main._omp_fn.0	1.78e+07 100 %	
85: cupti correlation callback cuda	1.78e+07 100 %	
301: vecAdd	1.78e+07 100 %	1.52e+07 85.5%
loop at vecAdd.cu: 14	1.07e+07 60.3%	8.99e+06 50.6%
vecAdd.cu: 15	1.70e+06 9.6%	1.70e+06 9.6%
15: \$vecAdd\$_Z3addii	1.46e+06 8.2%	1.46e+06 8.2%
15: \$vecAdd\$_Z3addii	1.36e+06 7.6%	1.36e+06 7.6%
15: \$vecAdd\$_Z3addii	1.33e+06 7.5%	1.33e+06 7.5%
15: \$vecAdd\$_Z3addii	1.22e+06 6.9%	1.22e+06 6.9%
vecAdd.cu: 15	9.92e+05 5.6%	9.92e+05 5.6%
vecAdd.cu: 15	9.20e+05 5.2%	9.20e+05 5.2%
vecAdd.cu: 15	9.04e+05 5.1%	9.04e+05 5.1%
vecAdd.cu: 15	8.29e+05 4.7%	8.29e+05 4.7%
loop at vecAdd.cu: 11	5.26e+06 29.6%	4.42e+06 24.9%
vecAdd.cu: 12	8.71e+05 4.9%	8.71e+05 4.9%
12: \$vecAdd\$_Z3addii	6.95e+05 3.9%	6.95e+05 3.9%
12: \$vecAdd\$_Z3addii	6.70e+05 3.8%	6.70e+05 3.8%
12: \$vecAdd\$_Z3addii	6.62e+05 3.7%	6.62e+05 3.7%
12: \$vecAdd\$_Z3addii	5.90e+05 3.3%	5.90e+05 3.3%
vecAdd.cu: 12	4.71e+05 2.7%	4.71e+05 2.7%
vecAdd.cu: 12	4.55e+05 2.6%	4.55e+05 2.6%

HPCToolkit Capabilities for GPU Code

MPI + OpenMP 4.5 or CUDA GPU accelerated applications

3-hpcviewer: lulesh

```

lulesh.cc  ompt-state-placeholders.c
901         hourg, numthreads) if(USE_GPU == 1)
902     {
903
904     # pragma omp target teams num_teams(TEAMS) thread_limit(THREADS) if (USE_GPU == 1)
905     # pragma omp distribute parallel for
906     for(Index_t i2=0;i2<numElem;++i2){
907         Real_t gamma[4][8];
908
909         gamma[0][0] = Real_t( 1.);
    
```

Top-down view Bottom-up view Flat view

Scope	GPU INST:Sum (I)	GPU INST:Sum (E)	STALL:MEM_DEP:Sum	STALL:MEM_DEP:Sum	XDMOV:TIME (us):Sun	XDMOV:TIME (us):Sun	KERNEL:TIME (us):Sur	KERNEL:TIME (us):Sur	KERNEL:COUNT :Sum
<program root>	3.09e+07 100 %		1.18e+07 100 %		3.35e+05 100 %		3.75e+05 100 %		1.43e+04 100 %
516: main	3.09e+07 100 %		1.18e+07 100 %		3.35e+05 100 %		3.75e+05 100 %		1.43e+04 100 %
loop at lulesh.cc: 3222	3.09e+07 100 %		1.18e+07 100 %		3.35e+05 100.0		3.75e+05 100 %		1.43e+04 100 %
3225: LagrangeLeapFrog(Domain&)	3.09e+07 100 %		1.18e+07 100 %		3.35e+05 100.0		3.75e+05 100 %		1.43e+04 100 %
3048: [I] LagrangeNodal(Domain&)	2.44e+07 79.0%		1.00e+07 84.8%		1.15e+05 34.4%		2.93e+05 78.1%		9.87e+03 68.8%
1570: [I] CalcForceForNodes(Domain&)	2.24e+07 72.4%		9.01e+06 76.1%		9.86e+04 29.4%		2.64e+05 70.4%		6.28e+03 43.8%
1397: [I] CalcVolumeForceForElems(Domain&)	2.21e+07 71.5%		8.95e+06 75.6%		9.86e+04 29.4%		2.59e+05 69.1%		5.38e+03 37.5%
1353: [I] CalcHourglassControlForElems(Domain&)	1.61e+07 52.2%		6.91e+06 58.3%		5.58e+04 16.6%		1.84e+05 49.2%		2.69e+03 18.8%
1279: [I] CalcFBHourglassForceForElems(Domain&)	9.05e+06 29.3%		4.02e+06 33.9%		4.10e+04 12.2%		1.02e+05 27.3%		1.79e+03 12.5%
904: <omp tgt kernel>	6.70e+06 21.7%		2.29e+06 19.3%				7.57e+04 20.2%	7.57e+04 20.2%	8.97e+02 6.3%
217: _omp_offloading_2d_e6aac4_ZL28C	6.70e+06 21.7%	6.70e+06 21.7%	2.29e+06 19.3%	2.29e+06 19.3%					
loop at lulesh.cc: 905	6.10e+06 19.7%	1.40e+05 0.5%	2.16e+06 18.3%	9.91e+04 0.8%					
loop at lulesh.cc: 906	5.96e+06 19.3%	5.11e+06 16.6%	2.06e+06 17.4%	1.60e+06 13.5%					
lulesh.cc: 906	1.37e+05 0.4%	1.37e+05 0.4%	9.91e+04 0.8%	9.91e+04 0.8%					
lulesh.cc: 905	3.09e+03 0.0%	3.09e+03 0.0%							
[I] inlined from lulesh.cc: 904	2.93e+05 0.9%	2.93e+05 0.9%	8.90e+04 0.8%	8.90e+04 0.8%					
lulesh.cc: 905	2.49e+05 0.8%	2.49e+05 0.8%	2.92e+04 0.2%	2.92e+04 0.2%					
loop at lulesh.cc: 905	3.96e+04 0.1%	3.96e+04 0.1%							
lulesh.cc: 906	1.72e+04 0.1%	1.72e+04 0.1%							
loop at lulesh.cc: 905	7.34e+03 0.0%	7.34e+03 0.0%	6.58e+03 0.1%	6.58e+03 0.1%					
ompt-state-placeholders.c: 217							7.57e+04 20.2%	7.57e+04 20.2%	8.97e+02 6.3%
1132: <omp tgt kernel>	2.34e+06 7.6%		1.73e+06 14.6%				2.67e+04 7.1%	2.67e+04 7.1%	8.97e+02 6.3%
901: <omp tgt copyin>					1.78e+04 5.3%	1.78e+04 5.3%			
1129: <omp tgt copyin>					2.32e+04 6.9%	2.32e+04 6.9%			
1198: <omp tgt kernel>	7.07e+06 22.9%		2.89e+06 24.4%				8.21e+04 21.9%	8.21e+04 21.9%	8.97e+02 6.3%
1196: <omp tgt copyin>					1.47e+04 4.4%	1.47e+04 4.4%			
1332: [I] IntegrateStressForElems(Domain&, doubl	5.63e+06 18.2%		1.94e+06 16.4%		2.30e+04 6.9%		6.98e+04 18.6%		1.79e+03 12.5%
1322: [I] InitStressTermsForElems(Domain&, doubl	3.25e+05 1.1%		9.79e+04 0.8%		1.43e+04 4.3%		5.20e+03 1.4%		8.97e+02 6.3%
1338: <omp tgt copyout>					5.59e+03 1.7%	5.59e+03 1.7%			
1384: <omp tgt kernel>	2.87e+05 0.9%		6.49e+04 0.5%				4.75e+03 1.3%	4.75e+03 1.3%	8.97e+02 6.3%

Other Capabilities

- **Measure hardware counters using Linux perf_events**
 - available events can be listed with
 - `hpcrun -L`
 - launching a binary created by `hpclink` with environment setting `HPCRUN_EVENT_LIST=LIST`
 - frequency based sampling: 300/s per thread or machine max
 - no need to set periods or frequencies unless you want precise control
 - hardware event multiplexing
 - measure more events than hardware counters
- **Kernel sampling**
 - measure activity in the Linux kernel in addition to your program
 - e.g., allocating and clearing memory pages
 - not available on BG/Q
 - measurement and attribution subject to system permissions
 - detailed attribution not available on NERSC or ANL systems

Ongoing Work and Future Plans

- **Ongoing work**
 - compliance with emerging OpenMP 5.0 standard
 - improving support for measuring GPU-accelerated nodes
 - sampling-based measurement and analysis of CUDA and OpenMP 5
 - add support for ANL's Aurora/A21 and ORNL's Frontier
 - data-centric analysis: associate costs with variables
 - analysis and attribution of performance to optimized code
 - automated analysis to deliver performance insights
- **Future plans**
 - scale measurement and analysis for exascale
 - support top-down analysis methods using hardware counters
 - resource-centric performance analysis
 - within and across nodes

HPCToolkit at ALCF

- **ALCF systems (theta, cooley)**
 - **on theta**
 - **source /projects/Tools/hpctoolkit/pkgs-theta/setup-ompt.sh**
 - **on cooley**
 - **source /projects/Tools/hpctoolkit/pkgs-cooley/setup-ompt.sh**
- **Man pages**
 - **automatically added to MANPATH by the aforementioned command**
- **ALCF guide to HPCToolkit**
 - **<http://www.alcf.anl.gov/user-guides/hpctoolkit>**

HPCToolkit at ORNL

- **On Summit**
 - **module use /gpfs/alpine/csc322/world-shared/modulefiles**
 - **module load hpctoolkit**
- **Man pages**
 - **automatically added to MANPATH by the aforementioned command**

GUIs for your Laptop

- **Download binary packages for HPCToolkit's user interfaces on your laptop**
 - <http://hpctoolkit.org/download/hpcviewer>

Detailed HPCToolkit Documentation

<http://hpctoolkit.org/documentation.html>

- **Comprehensive user manual:**

- <http://hpctoolkit.org/manual/HPCToolkit-users-manual.pdf>

- **Quick start guide**

- **essential overview that almost fits on one page**

- **Using HPCToolkit with statically linked programs**

- **a guide for using hpctoolkit on BG/Q and Cray platforms**

- **The hpcviewer and hpctraceviewer user interfaces**

- **Effective strategies for analyzing program performance with HPCToolkit**

- **analyzing scalability, waste, multicore performance ...**

- **HPCToolkit and MPI**

- **HPCToolkit Troubleshooting**

- **why don't I have any source code in the viewer?**

- **hpcviewer isn't working well over the network ... what can I do?**

- **Installation guide**

Advice for Using HPCToolkit

Using HPCToolkit

- Add hpctoolkit's bin directory to your path using softenv
- Adjust your compiler flags (if you want full attribution to src)
 - add **-g** flag after any optimization flags
- Add hpclink as a prefix to your Makefile's link line
 - e.g. **hpclink mpixlf -o myapp foo.o ... lib.a -lm ...**
- See what sampling triggers are available on BG/Q
 - use **hpclink** to link your executable
 - launch executable with environment variable **HPCRUN_EVENT_LIST=LIST**
 - you can launch this on 1 core of 1 node
 - no need to provide arguments or input files for your program
they will be ignored

Monitoring Large Executions

- **Collecting performance data on every node is typically not necessary**
- **Can improve scalability of data collection by recording data for only a fraction of processes**
 - **set environment variable HPCRUN_PROCESS_FRACTION**
 - **e.g. collect data for 10% of your processes**
 - **set environment variable HPCRUN_PROCESS_FRACTION=0.10**

Digesting your Performance Data

- Use `hpcstruct` to reconstruct program structure
 - e.g. `hpcstruct your_app`
 - creates `your_app.hpcstruct`
- Correlate measurements to source code with `hpcprof` and `hpcprof-mpi`
 - run `hpcprof` on the front-end to analyze data from small runs
 - run `hpcprof-mpi` on the compute nodes to analyze data from lots of nodes/threads in parallel
 - notes
 - much faster to do this on an `x86_64` vis cluster (cooley) than on BG/Q
 - avoid expensive per-thread profiles with `--metric-db no`
- Digesting performance data in parallel with `hpcprof-mpi`
 - `qsub -A ... -t 20 -n 32 --mode c1 --proccount 32 --cwd `pwd` \`
`/projects/Tools/hpctoolkit/pkgsvesta/hpctoolkit/bin/hpcprof-mpi \`
`-S your_app.hpcstruct \`
`-I /path/to/your_app/src/+ \`
`hpctoolkit-your_app-measurements.jobid`
- Hint: you can run `hpcprof-mpi` on the `x86_64` vis cluster (cooley)

Analysis and Visualization

- **Use hpcviewer to open resulting database**
 - **warning: first time you graph any data, it will pause to combine info from all threads into one file**
- **Use hpctraceviewer to explore traces**
 - **warning: first time you open a trace database, the viewer will pause to combine info from all threads into one file**
- **Try our our user interfaces before collecting your own data**
 - **example performance data**
<http://hpctoolkit.org/examples.html>

Installing HPCToolkit GUIs on your Laptop

- See <http://hpctoolkit.org/download/hpcviewer>
- Download the latest for your laptop (Linux, Mac, Windows)
 - hpctraceviewer
 - hpcviewer

A Note for Mac Users

When installing HPCToolkit GUIs on your Mac laptop, don't simply download and double click on the zip file and have Finder unpack them. Follow the Terminal-based installation directions on the website to avoid interference by Mac Security.