

Gaining Insight into Parallel Program Performance using HPCToolkit

John Mellor-Crummey
Department of Computer Science
Rice University

<http://hpctoolkit.org>



Acknowledgments

- **Current funding**
 - DOE Office of Science ASCR X-Stack “PIPER” Award
 - ANL Subcontract 4F-30241
 - LLNL Subcontracts B609118, B614178
 - Intel gift funds
- **Project team**
 - **Research Staff**
 - Laksono Adhianto, Mark Krentel, Zung Nguyen
 - **Recent Alumni**
 - Xu Liu (William and Mary, 2014)
 - Milind Chabbi (HP Labs, 2015)
 - Mike Fagan (Rice)

Challenges for Computational Scientists

- **Rapidly evolving platforms and applications**
 - **architecture**
 - rapidly changing multicore microprocessor designs
 - increasing architectural diversity
 - multicore, manycore, accelerators
 - increasing parallelism within nodes
 - **applications**
 - exploit threaded parallelism in addition to MPI
 - enhance 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**

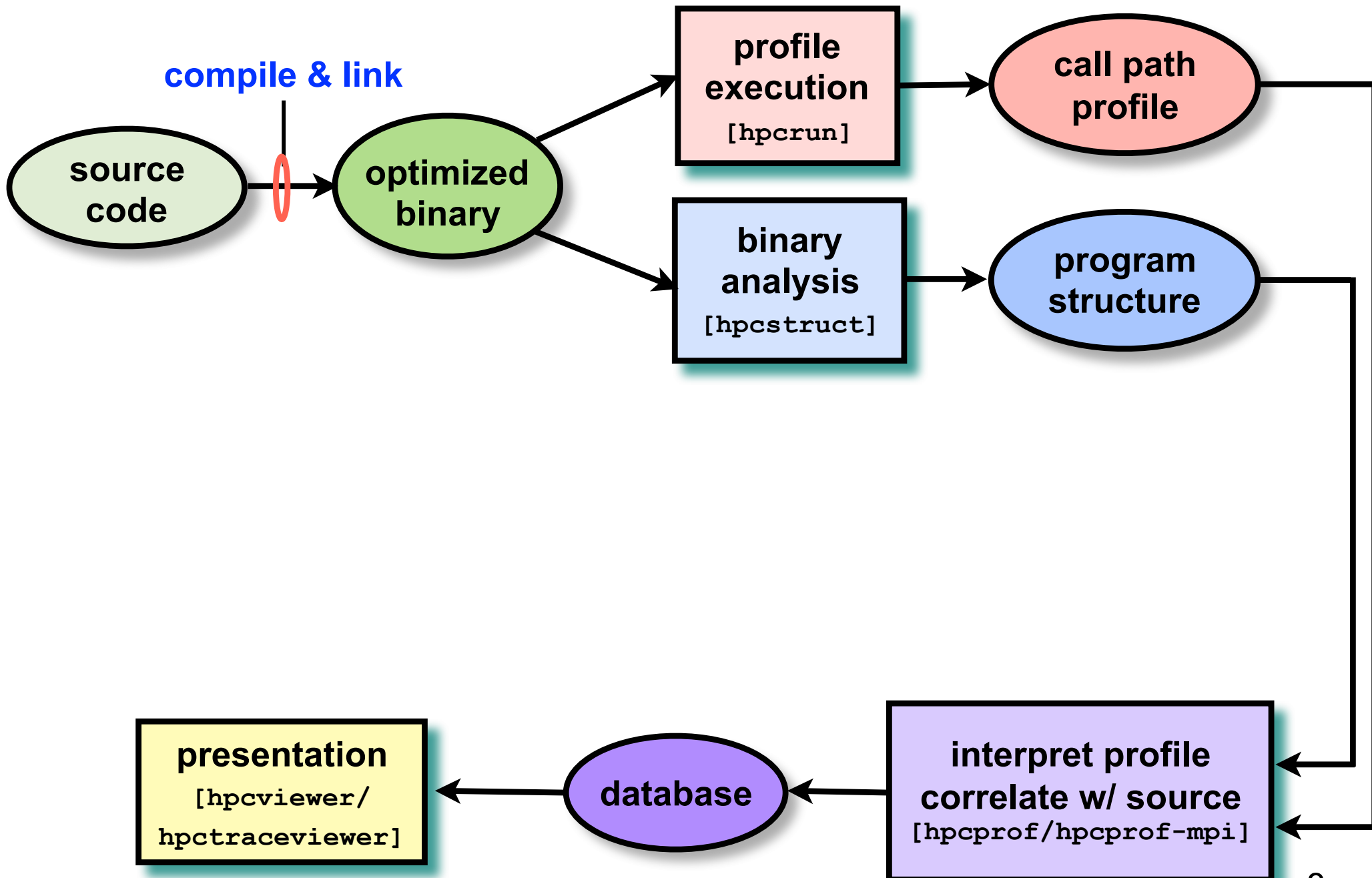
Outline

- **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 threading performance**
 - blame shifting
- **Today and the future**

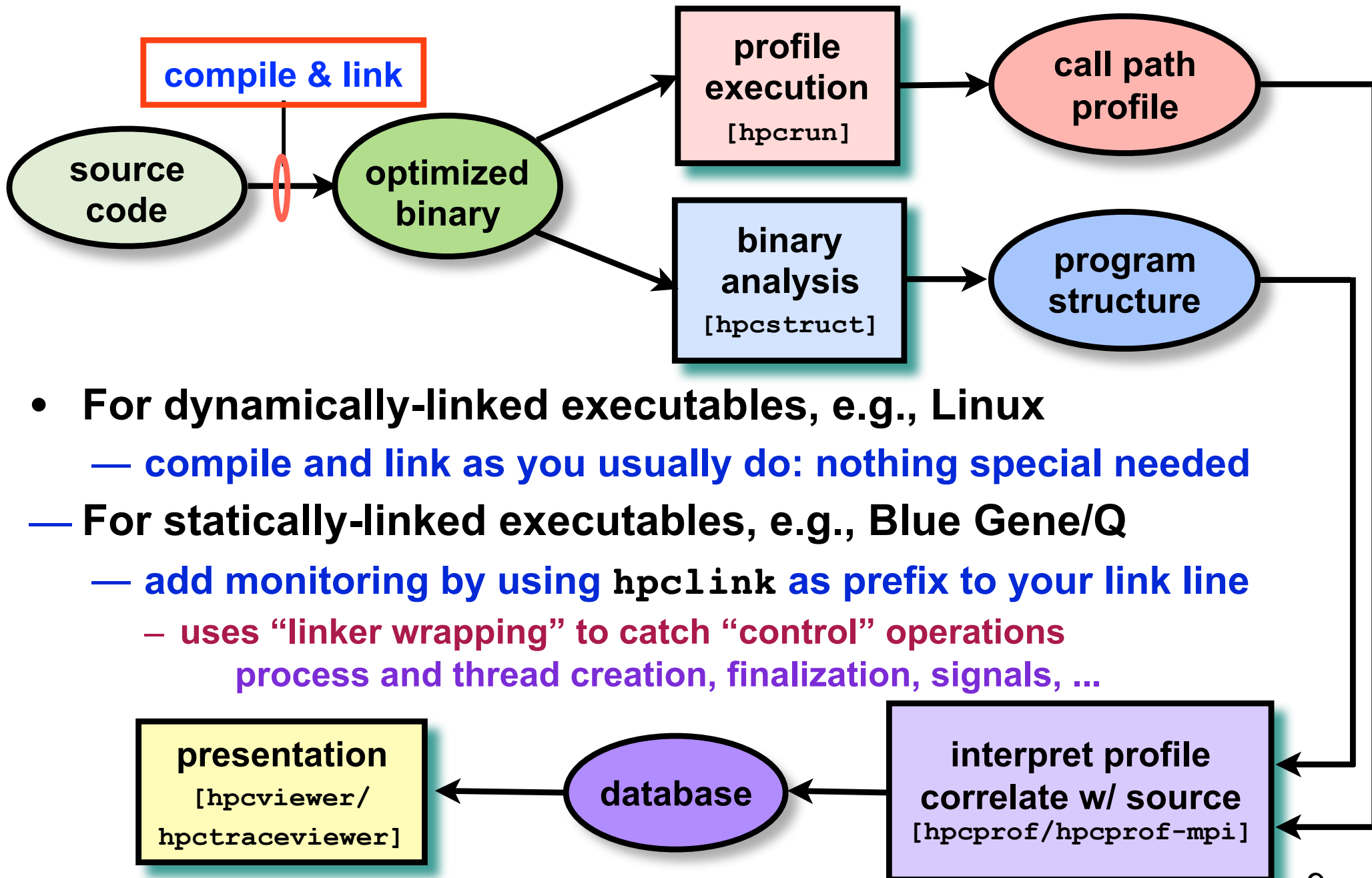
Rice University's HPCToolkit

- **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

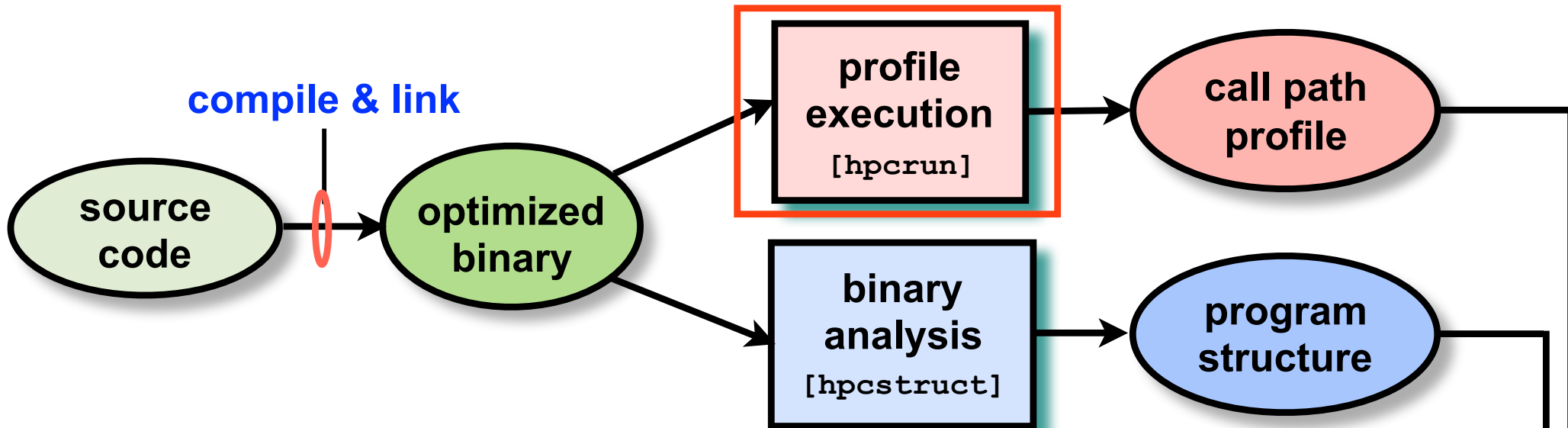


HPCToolkit Workflow



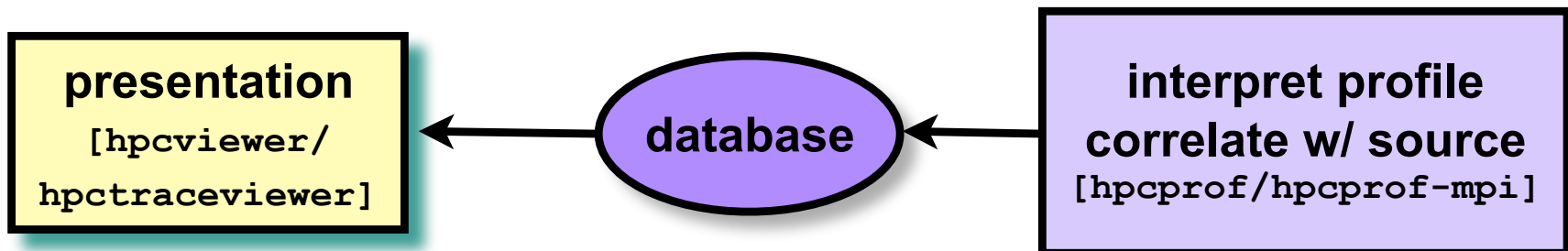
- For dynamically-linked executables, e.g., Linux
 - **compile and link as you usually do: nothing special needed**
- For statically-linked executables, e.g., Blue Gene/Q
 - **add monitoring by using `hpcLink` as prefix to your link line**
 - uses “linker wrapping” to catch “control” operations
process and thread creation, finalization, signals, ...

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



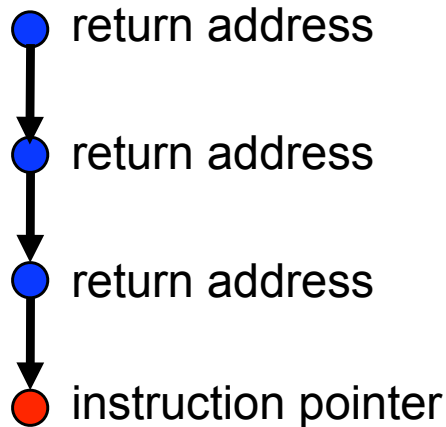
Call Path Profiling

Measure and attribute costs in context

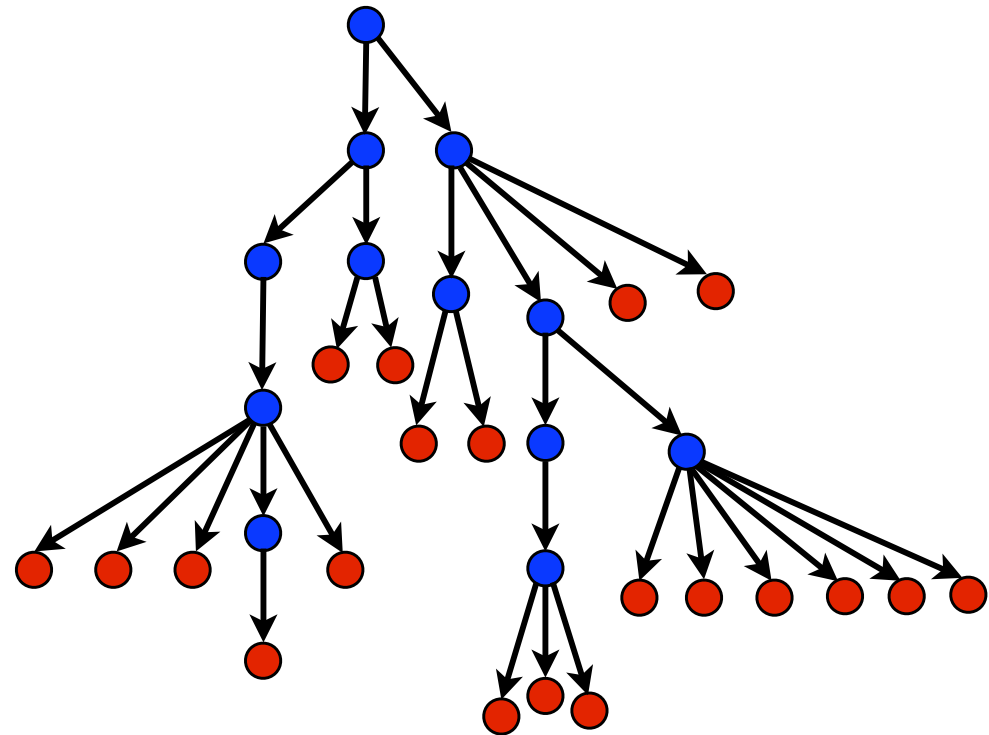
sample timer or hardware counter overflows

gather calling context using stack unwinding

Call path sample

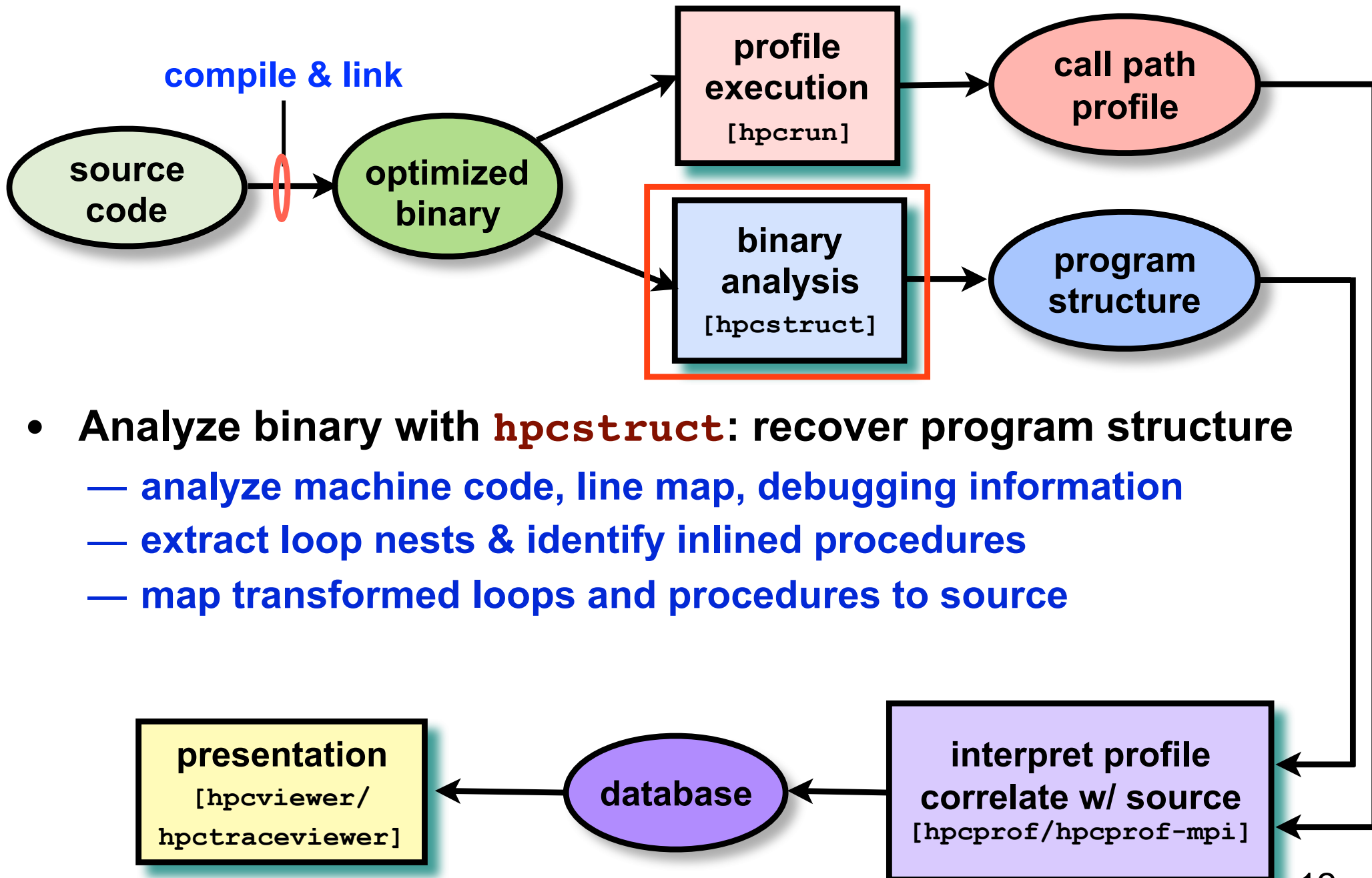


Calling context tree



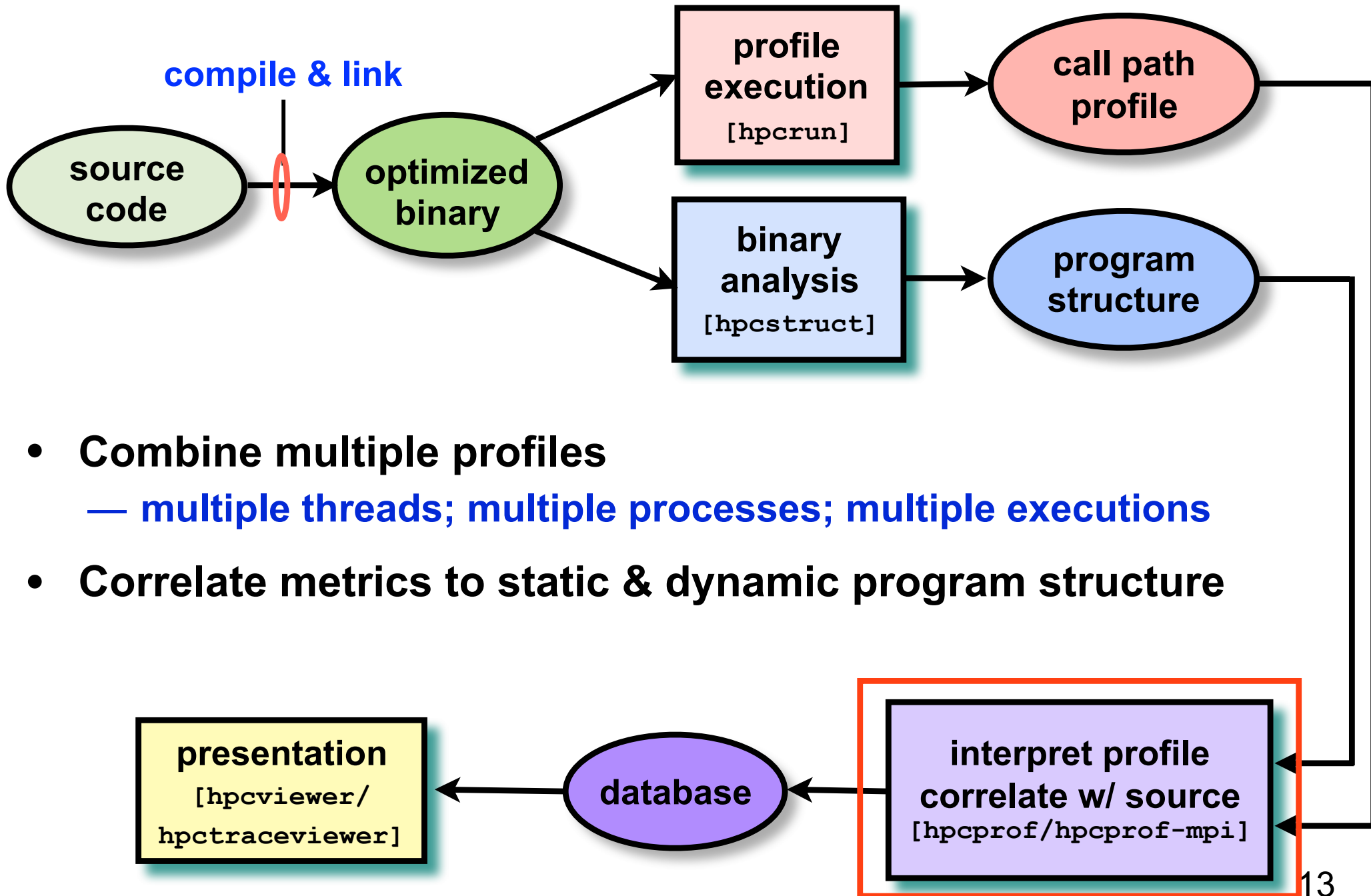
**Overhead proportional to sampling frequency...
...not call frequency**

HPCToolkit Workflow



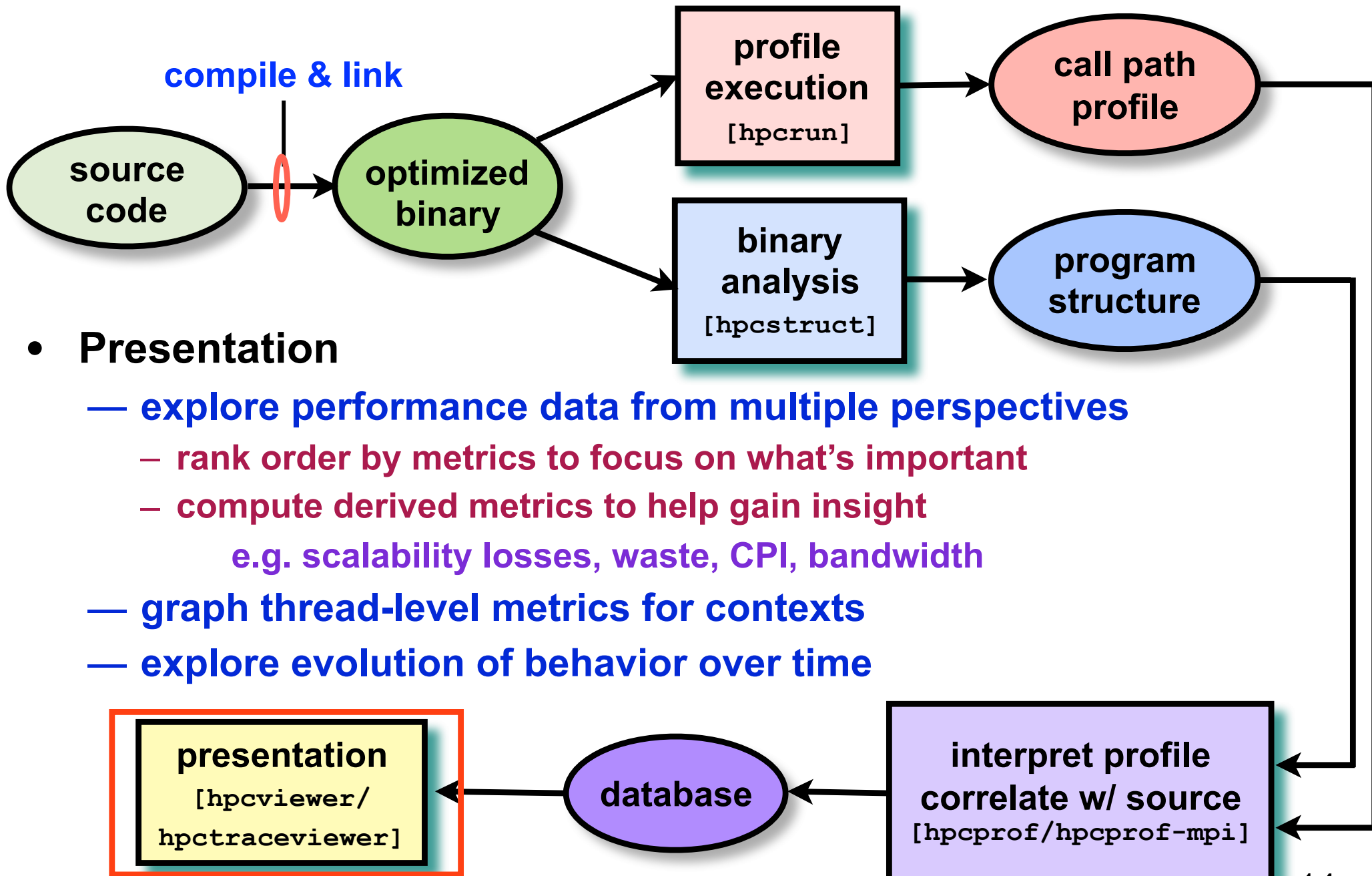
- Analyze binary with **hpcstruct**: recover program structure
 - analyze machine code, line map, debugging information
 - extract loop nests & identify inlined procedures
 - map transformed loops and procedures to source

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

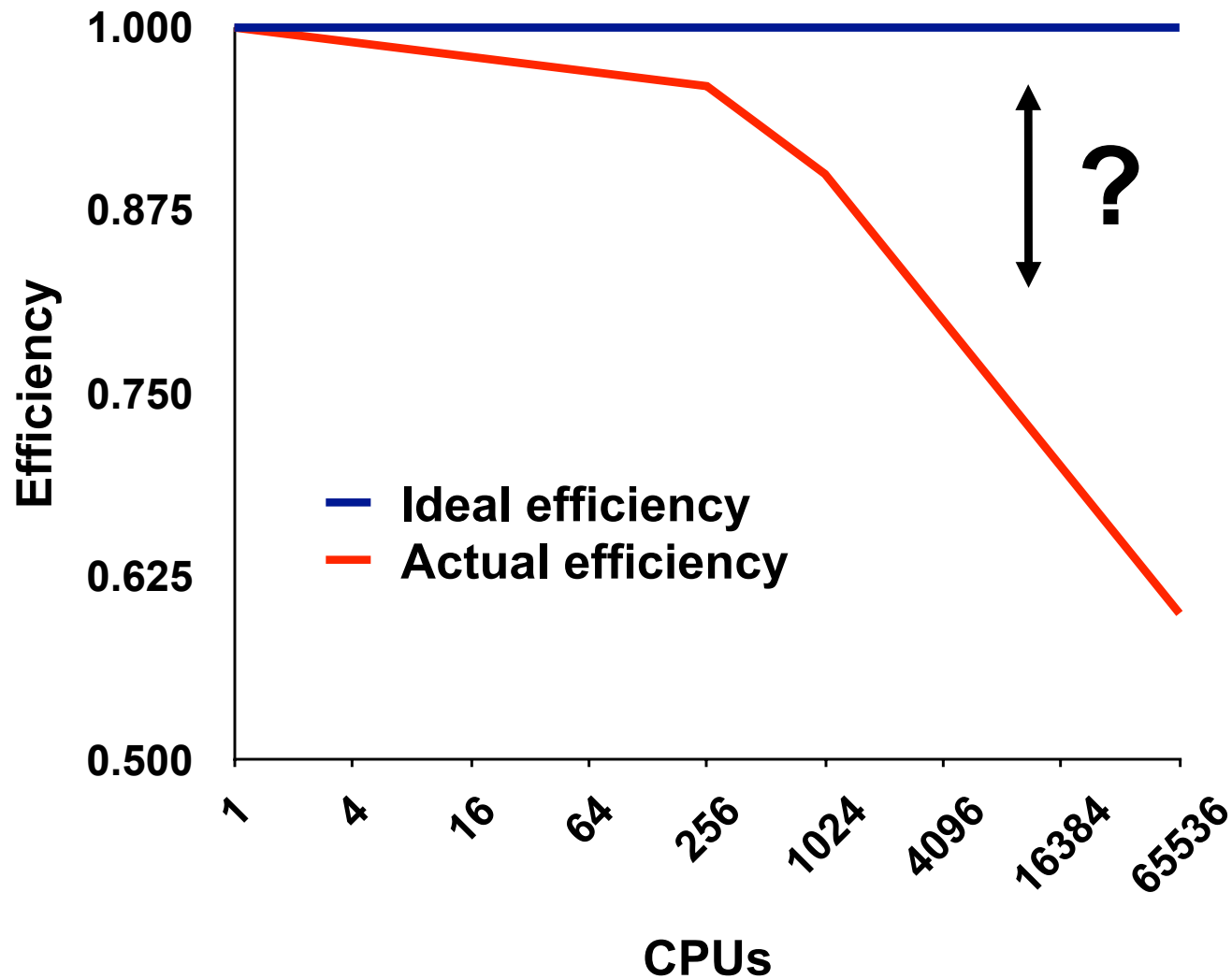
Code-centric Analysis with hpcviewer

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

The screenshot displays the hpcviewer interface for the executable 'lulesh-RAJA-parallel.exe'. The source pane shows C++ code for a 'forall' function. The view control pane includes 'Calling Context View', 'Callers View', and 'Flat View'. The metric display pane shows a flame graph with various function calls and their metrics. The navigation pane shows a tree view of the code structure. The metric pane shows a table of performance metrics.

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(int*, double*, double*, double*, 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(int*, double*, double*, double*, 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*, 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

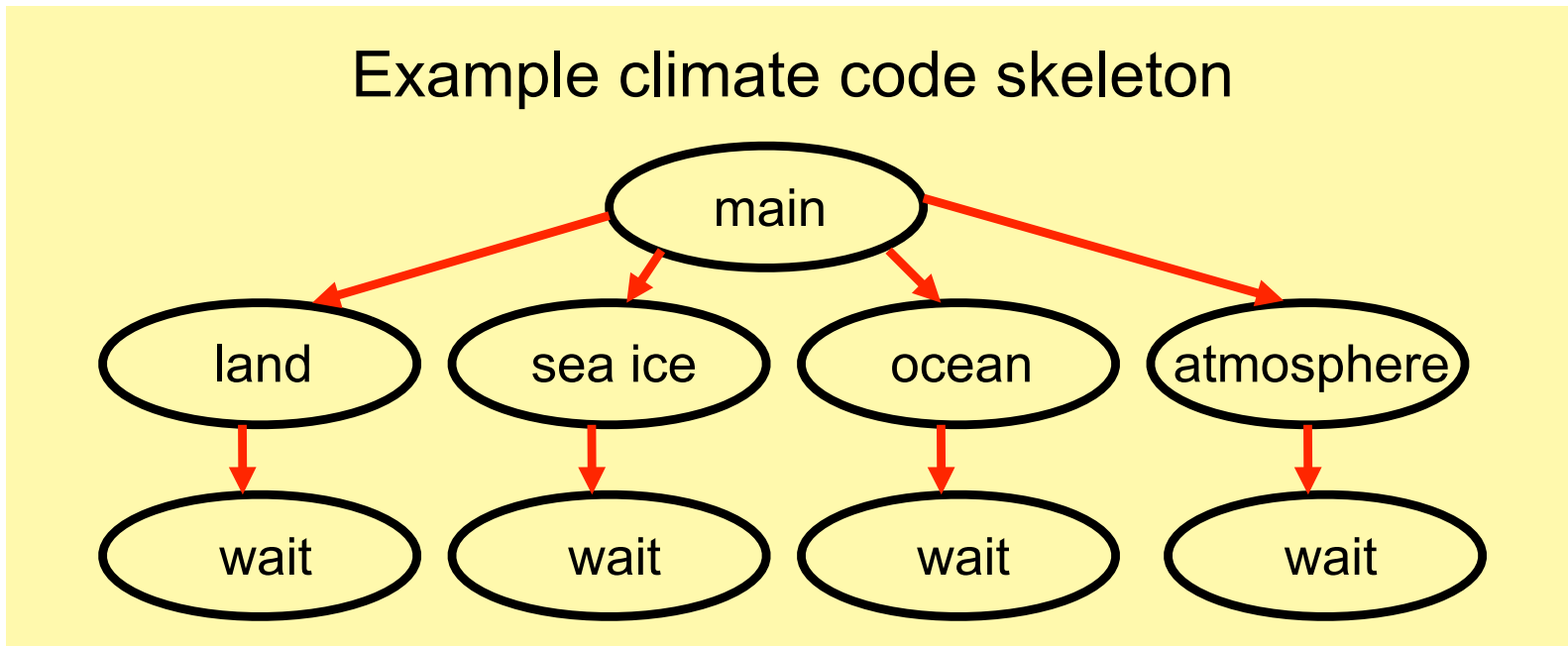
Goal: Automatic Scaling Analysis

- 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



- **Monitoring**

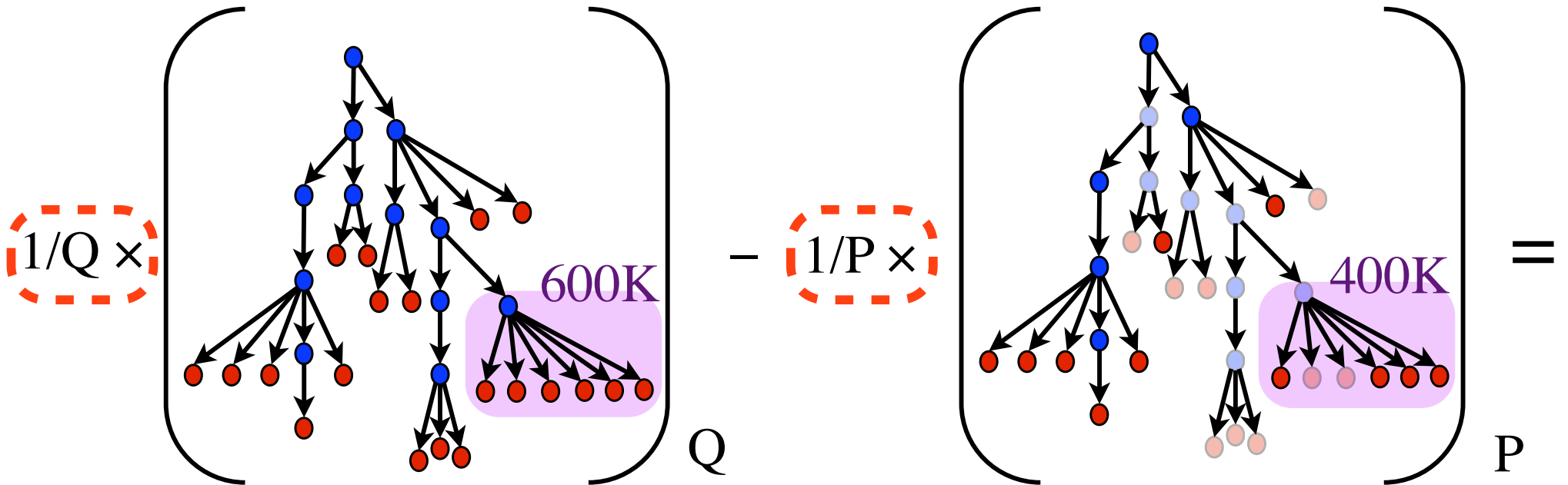
- bottleneck nature: computation, data movement, synchronization?
- 2 pragmatic constraints
 - acceptable data volume
 - low perturbation for use in production runs

Performance Analysis with Expectations

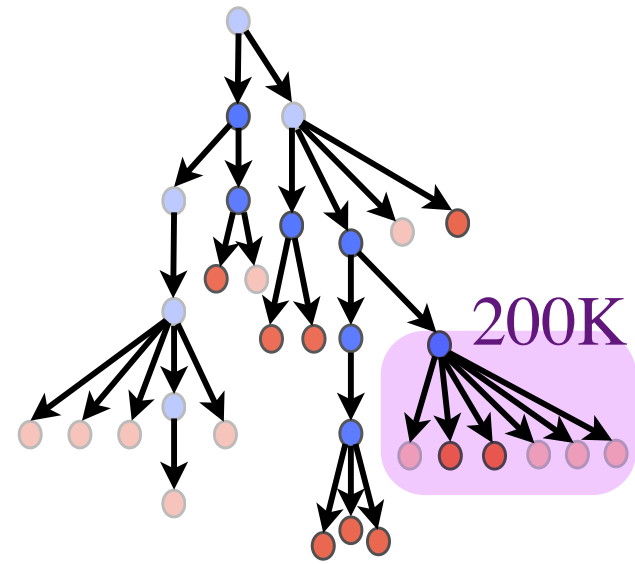
- You have performance expectations for your parallel code
 - strong scaling: linear speedup
 - weak scaling: constant execution time

- 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:

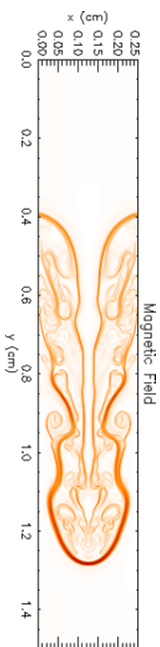
University of Chicago FLASH

white dwarf detonation

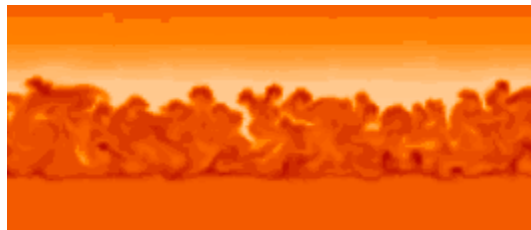
Blue Gene/P

8192 vs. 256 processors

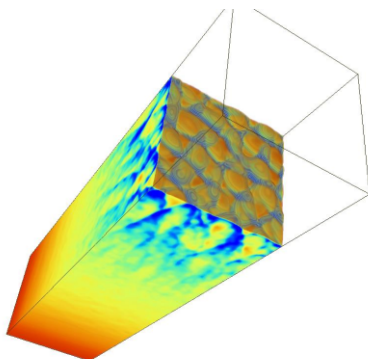
weak



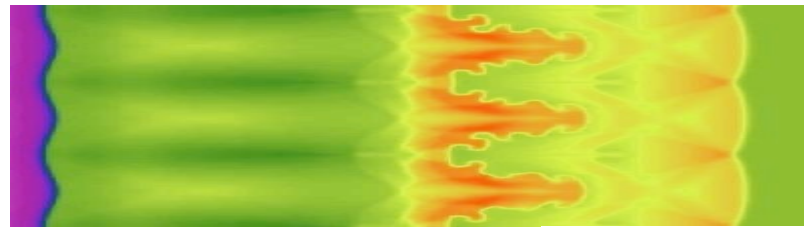
Magnetic Rayleigh-Taylor



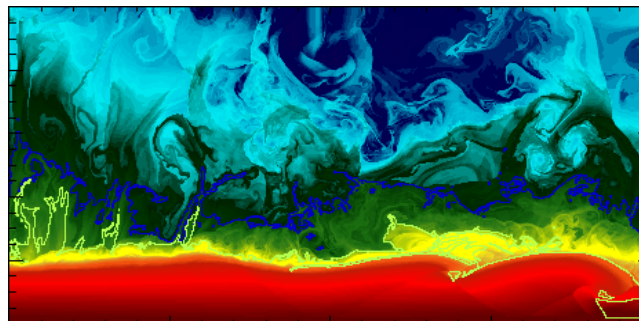
Nova outbursts on white dwarfs



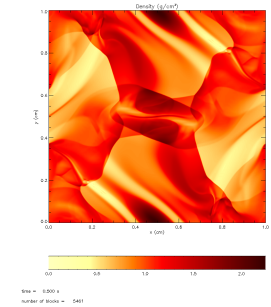
Cellular detonation



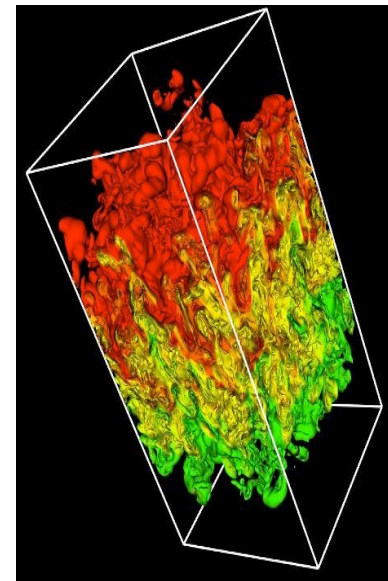
Laser-driven shock instabilities



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

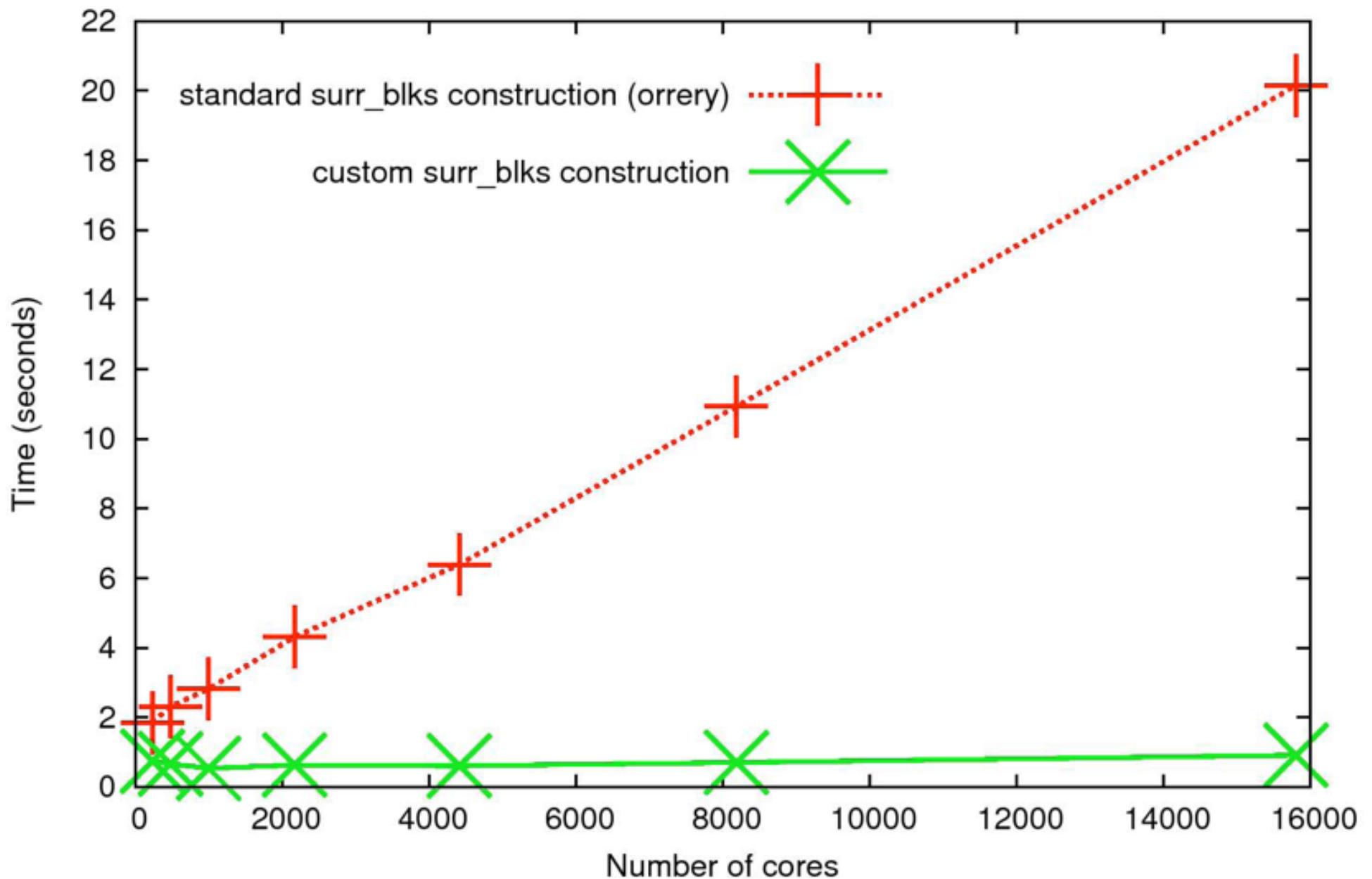
hpcviewer: FLASH/white dwarf: IBM BG/P, weak 256->8192

```
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

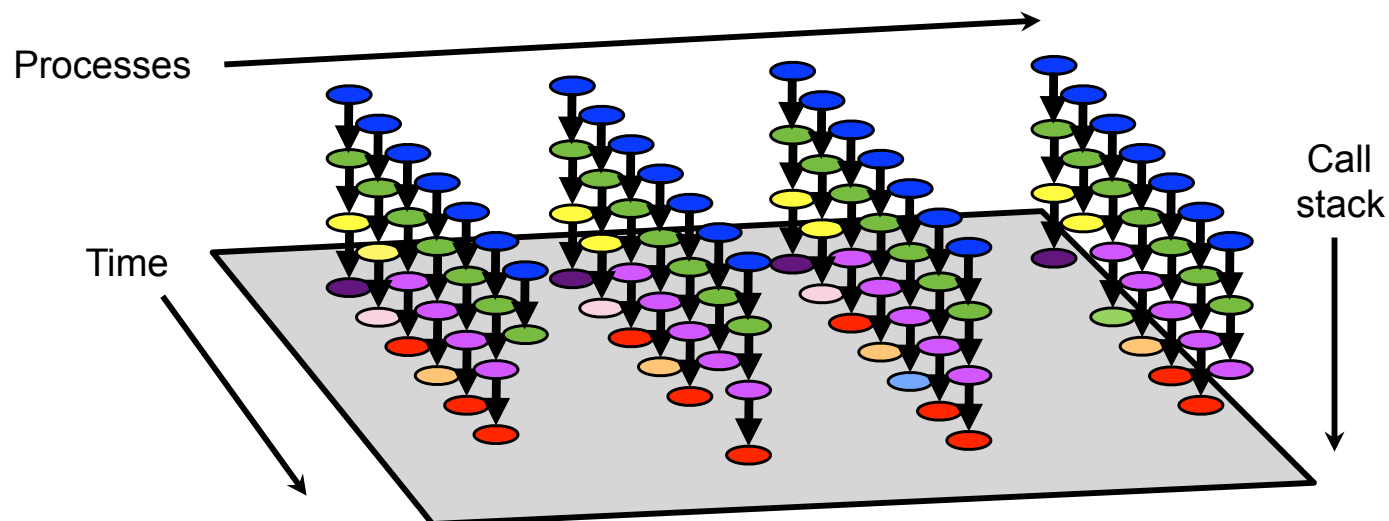
Improved Flash Scaling of AMR Setup



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



Measurement & Attribution of L2 Activity

- **L2Unit measurement capabilities**
 - e.g., counts load/store activity
 - node-wide counting; not thread-centric
 - global or per slice counting
 - supports threshold-based sampling
 - samples delivered late: about 800 cycles after threshold reached
 - each sample delivered to ALL threads/cores
- **HPCToolkit approach**
 - attribute a share of L2Unit activity to each thread context for each sample
 - e.g., when using a threshold of 1M loads and T threads, attribute 1M/T events to the active context in each thread when each sample event occurs
 - best effort attribution
 - strength: correlate L2Unit activity with regions of your code
 - weakness: some threads may get blamed for activity of others

OpenMP: A Challenge for Tools

- Large gap between between threaded programming models and their implementations

2-hpcviewer: LULESH_OMP.host

```
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], hgyf[8], hgfh[8] ;  
1295  
1296     Real_t coefficient;  
1297  
1298     ...  
1299 }
```

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: __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 IuleshMPI_OMP (8 MPI x 3 OMP), 30, REALTIME@1000

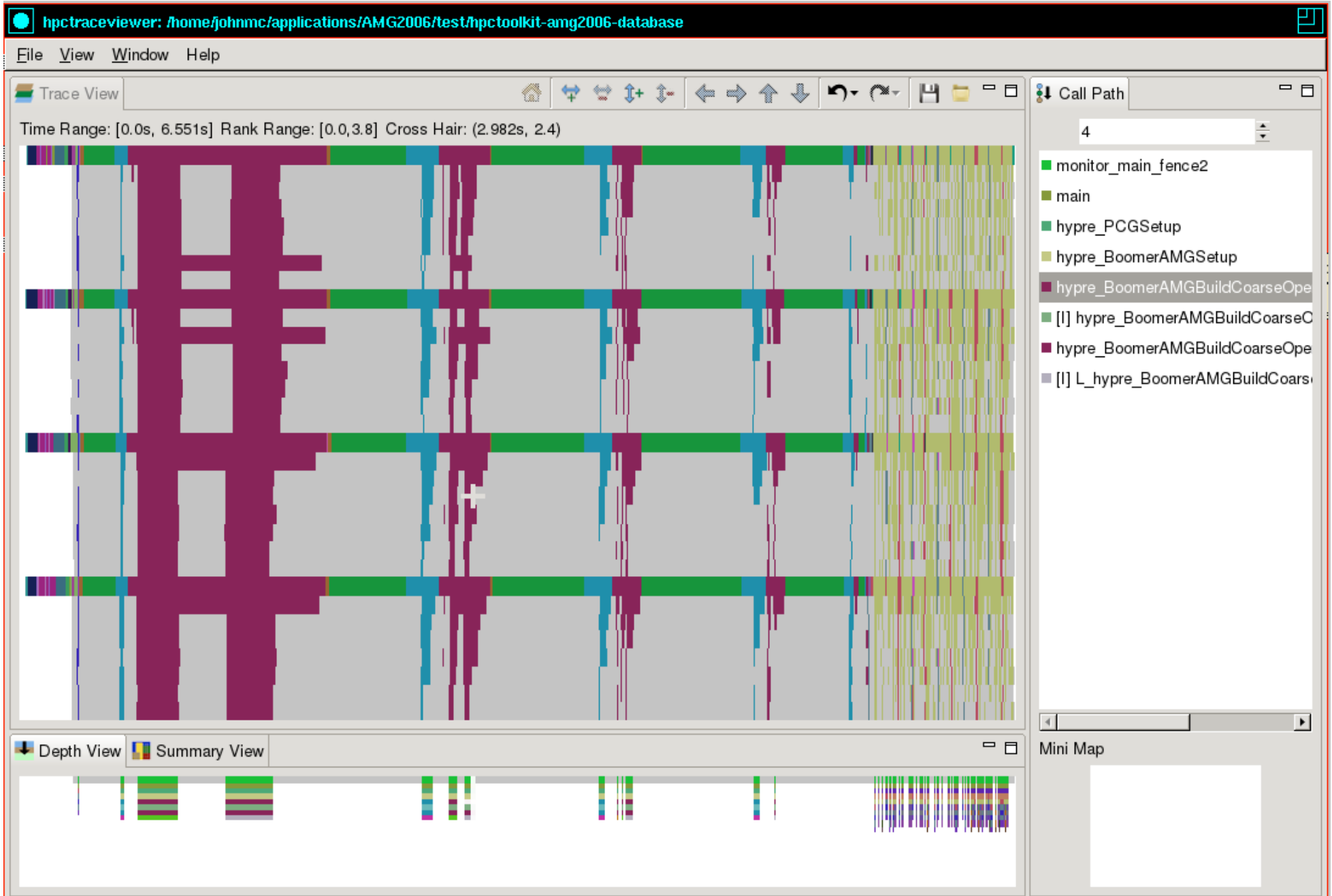
The screenshot displays the hpcviewer interface for the application IuleshMPI_OMP. It is divided into three main sections:

- source view:** Shows the source code for IuleshMPI_OMP.cc. A parallel loop is highlighted, starting at line 3217 with `#pragma omp parallel for firstprivate(numNode)` and ending at line 3222. The loop body includes calculations for node element counts and starting indices.
- thread view:** A scatter plot showing the execution time of the thread `Process.Thread` (x-axis, 00.00 to 07.00) against the `Metric Value` (y-axis, 0.0E0 to 1.0E7). The plot shows a regular pattern of data points, indicating periodic execution of the thread.
- metric view:** A table showing the execution metrics for the application. The table has columns for `Scope`, `REALTIME (usec):Sum (I)`, `REALTIME (usec):Sum (E)`, and `REALTIME (usec):Sum (R)`. The table shows the execution time of the application and its components, including the main function and the parallel loop.

Scope	REALTIME (usec):Sum (I)	REALTIME (usec):Sum (E)	REALTIME (usec):Sum (R)
Experiment Aggregate Metrics	3.55e+10 100 %	3.55e+10 100 %	
monitor_main	2.58e+10 72.8%		
483: main	2.58e+10 72.8%	7.02e+03 0.0%	
loop at IuleshMPI_OMP.cc: 5625	2.58e+10 72.8%	4.01e+03 0.0%	
5626: LagrangeLeapFrog(Domain*)	2.53e+10 71.2%	1.50e+04 0.0%	
4796: LagrangeNodal(Domain*)	1.68e+10 47.5%	5.02e+04 0.0%	
3476: CalcForceForNodes(Domain*)	1.60e+10 45.0%	1.18e+07 0.0%	
3370: CalcVolumeForceForElems(Domain*)	1.44e+10 40.7%	1.56e+07 0.0%	
3344: CalcHourglassControlForElems(Domain*, double*, double)	1.09e+10 30.7%	1.21e+08 0.3%	
3289: CalcFBHourglassForceForElems(int*, double*, double*, double*, double*, double*, double*, double*, double*, int*, int*, int*, double*, double*, double*, double*, double*, double*, double*, double*)	7.86e+09 22.1%	2.41e+08 0.7%	
3066: CalcFBHourglassForceForElems(int*, double*, double*, double*, double*, double*, double*, double*, double*, int*, int*, int*, double*, double*, double*, double*, double*, double*, double*, double*)	3.66e+09 10.3%	2.57e+09 7.2%	
3217: __kmp_fork_barrier(int, int)	3.08e+09 8.7%	5.01e+03 0.0%	
3066: __kmp_fork_barrier(int, int)	5.44e+08 1.5%	1.00e+04 0.0%	
3217: CalcFBHourglassForceForElems(int*, double*, double*, double*, double*, double*, double*, double*, double*, int*, int*, int*, double*, double*, double*, double*, double*, double*, double*, double*)	3.16e+08 0.9%	3.15e+08 0.9%	

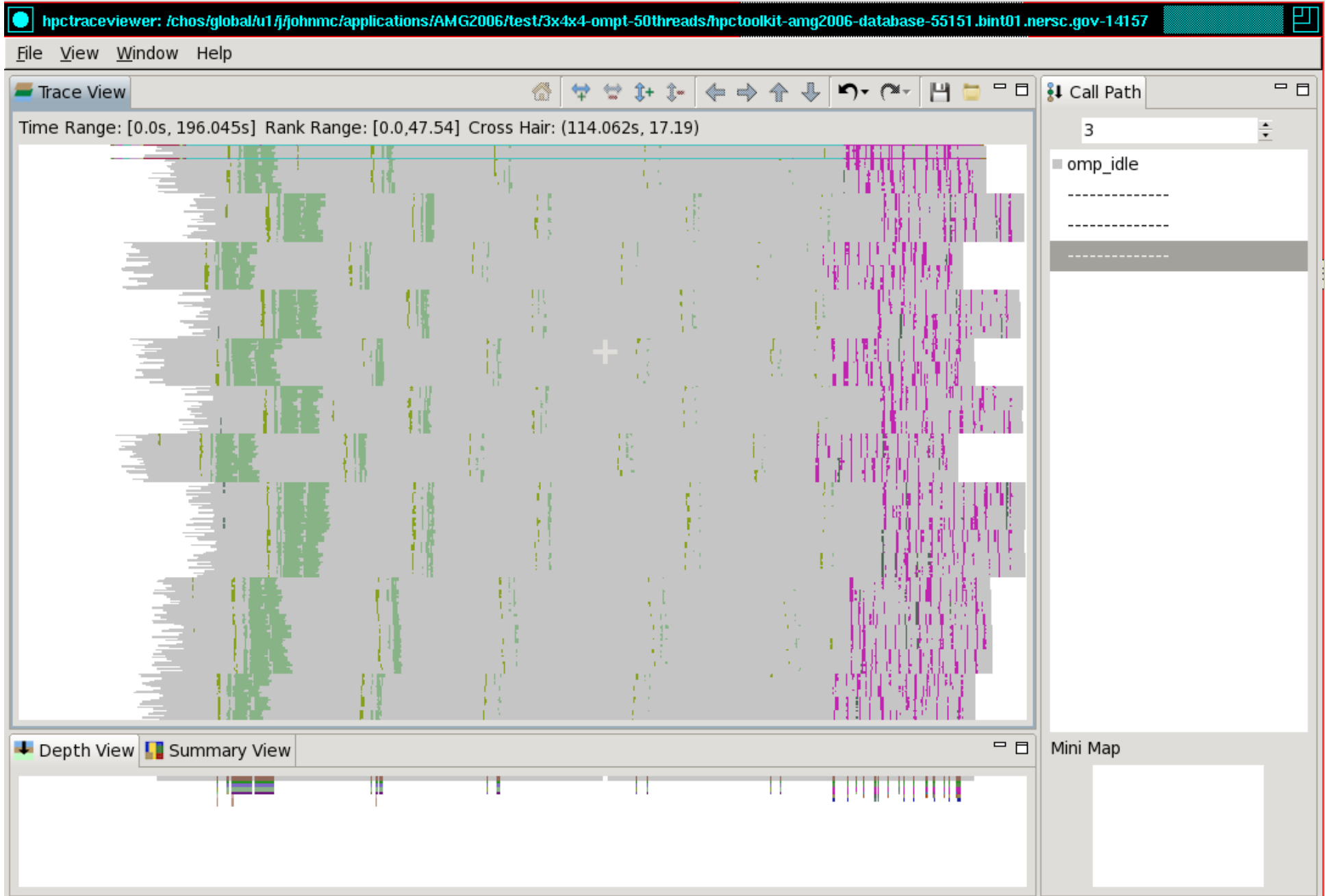
2 18-core Haswell
4 MPI ranks
6+3 threads per rank

Case Study: AMG2006



12 nodes on Babbage@NERSC
24 Xeon Phi
48 MPI ranks
50+5 threads per rank

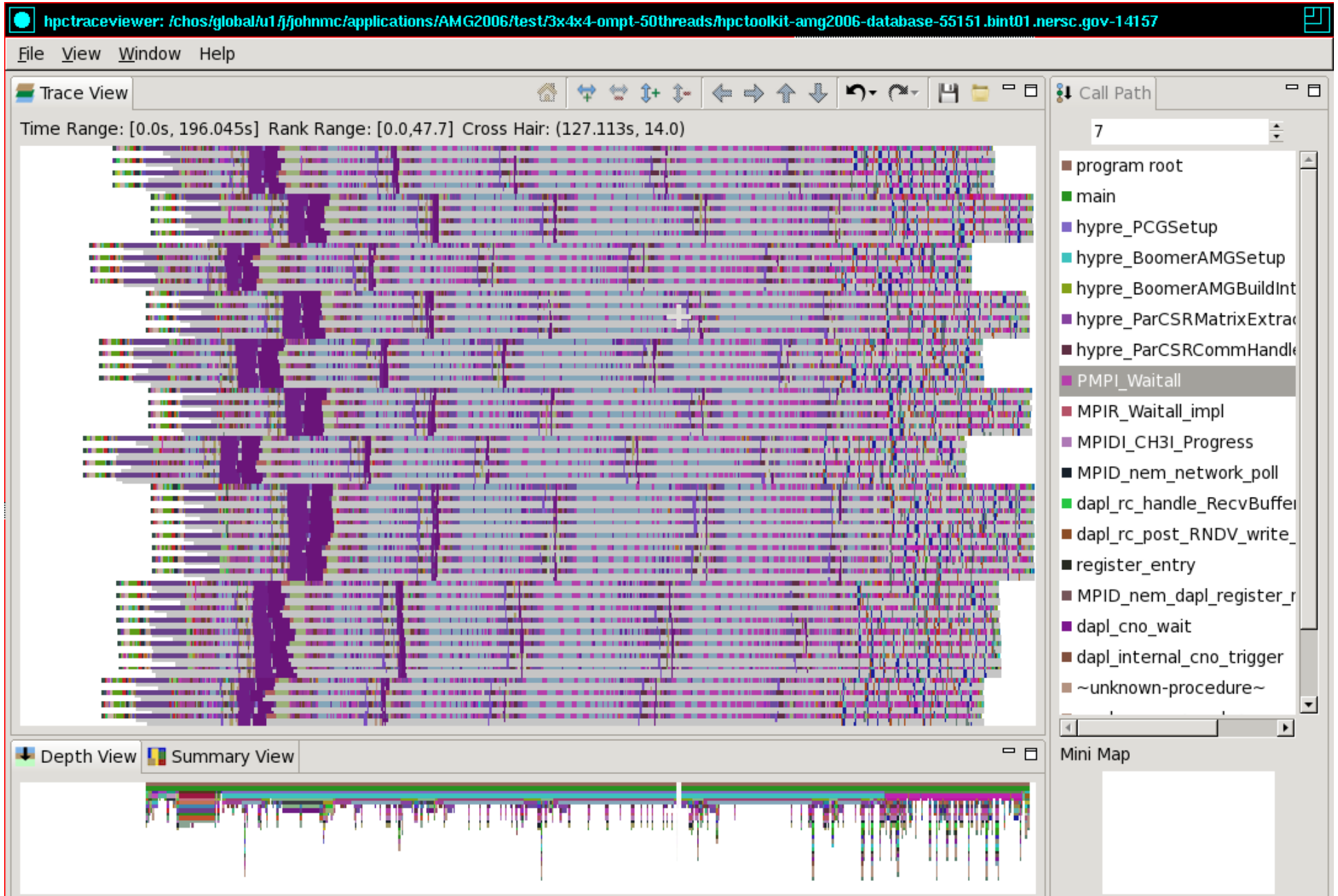
Case Study: AMG2006



12 nodes on Babbage@NERSC
24 Xeon Phi
48 MPI ranks
50+5 threads per rank

Case Study: AMG2006

Slice
Thread 0 from each MPI rank
First two OpenMP workers



Blame-shifting: Analyze Thread Performance

	Problem	Approach
Undirected Blame Shifting ^{1,3}	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 ^{2,3}	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

³Liu, Mellor-Crummey, Fagan: ICS 2013

Blame Shifting: Idleness in AMG2006

The screenshot shows the hpcviewer interface for the amg2006 project. The top pane displays the source code for main.c, with the hypre_BoomerAMGCoarsenFalgout function highlighted. The bottom pane shows a performance table with columns for Scope, REALTIME (usec):Sum (I), REALTIME (usec):Sum (E), OMP_IDLE:Sum (I), OMP_IDLE:Sum (E), and OMP_WORK:Sum (I). The row for hypre_BoomerAMGCoarsenFalgout is highlighted with a red box, showing a significant amount of OMP_IDLE time (6.62e+07 usec, 50.1% of the total for that scope).

Scope	REALTIME (usec):Sum (I)	REALTIME (usec):Sum (E)	OMP_IDLE:Sum (I)	OMP_IDLE:Sum (E)	OMP_WORK:Sum (I)
Experiment Aggregate Metrics	1.97e+08 100 %	1.97e+08 100 %	1.32e+08 100 %	1.32e+08 100 %	6.36e+07 100 %
monitor_main_fence2	6.87e+07 34.8%		1.32e+08 99.9%		6.35e+07 99.9%
497: main	6.87e+07 34.8%	9.02e+03 0.0%	1.32e+08 99.9%		6.35e+07 99.9%
2431: hypre_PCGSetup	5.02e+07 25.4%		1.16e+08 88.1%		4.70e+07 77.7%
236: hypre_BoomerAMGSetup	5.02e+07 25.4%		1.16e+08 88.1%		4.70e+07 77.7%
609: hypre_BoomerAMGCoarsenFalgout	9.46e+06 4.8%		6.62e+07 50.1%		9.46e+06 100 %
1953: hypre_BoomerAMGCoarsen	7.78e+06 3.9%	5.13e+06 2.6%	5.44e+07 41.2%	3.59e+07 27.2%	7.78e+06 100 %
loop at par_coarsen.c: 621	6.56e+06 3.3%		4.59e+07 34.8%		6.56e+06 100 %
loop at par_coarsen.c: 621	4.93e+06 2.5%	2.10e+04 0.0%	3.45e+07 26.1%	1.47e+05 0.1%	4.93e+06 100 %
loop at par_coarsen.c: 725	3.00e+06 1.5%	2.89e+05 0.1%	2.10e+07 15.9%	2.02e+06 1.5%	3.00e+06 100 %
loop at par_coarsen.c: 732	2.10e+06 1.1%	2.10e+06 1.1%	1.47e+07 11.1%	1.47e+07 11.1%	2.10e+06 100 %
par_coarsen.c: 738	1.33e+06 0.7%	1.33e+06 0.7%	9.30e+06 7.0%	9.30e+06 7.0%	1.33e+06 100 %
par_coarsen.c: 732	3.79e+05 0.2%	3.79e+05 0.2%	2.65e+06 2.0%	2.65e+06 2.0%	3.79e+05 100 %
par_coarsen.c: 735	3.53e+05 0.2%	3.53e+05 0.2%	2.47e+06 1.9%	2.47e+06 1.9%	3.53e+05 100 %
par_coarsen.c: 734	4.01e+04 0.0%	4.01e+04 0.0%	2.80e+05 0.2%	2.80e+05 0.2%	4.01e+04 100 %

OpenMP Tools API Status

- **April 2014: OpenMP TR2**
 - OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis**
 - **Alexandre Eichenberger (IBM), John Mellor-Crummey (Rice), Martin Schulz (LLNL), and other members of the OpenMP tools subcommittee**
 - major step toward having a tools API added to OpenMP standard**
- **OMPT prototype implementations: IBM, LLVM (prototype)**
- **Next steps**
 - refine OMPT in LLVM OpenMP runtime for production use**
 - **contributors: Rice, University of Oregon, RWTH Aachen, TU Dresden**
 - working to add OMPT to the language standard**
 - refine HPCToolkit OMPT to match emerging standard**

Ongoing Work and Future Plans

- **Argonne**
 - **scale I/O strategy**
 - **one file per node rather than one file per thread**
 - **finish scalable traceviewer**
 - **split traceviewer into client server**
 - server runs as a parallel program on vis cluster
 - client runs on your laptop
- **Other work**
 - **support for many threads**
 - **data-centric analysis: associate costs with variables**
 - **analysis and attribution of performance to optimized code**
- **Future plans**
 - **resource-centric performance analysis**
 - **within and across nodes**
 - **scale measurement and analysis for exascale**
 - **automated analysis to deliver performance insights**

HPCToolkit at ALCF

- **ALCF systems (vesta, mira, cetus)**
 - **in your .soft file, add one of the following lines below**
 - **+hpctoolkit-devel**
 - **(this package is always the most up-to-date)**
- **Man pages**
 - **automatically added to MANPATH by the aforementioned softenv command**
- **ALCF guide to HPCToolkit**
 - **<http://www.alcf.anl.gov/user-guides/hpctoolkit>**
- **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**

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

Collecting Performance Data on BG/Q

- **Collecting traces on BG/Q**
 - set environment variable `HPCRUN_TRACE=1`
 - use `WALLCLOCK` or `PAPI_TOT_CYC` as one of your sample sources when collecting a trace
- **Launching your job on BG/Q using hpctoolkit**
 - `qsub -A ... -t 10 -n 1024 --mode c1 --proccount 16384 \
--cwd `pwd` \
--env OMP_NUM_THREADS=2:\`
 - `HPCRUN_EVENT_LIST=WALLCLOCK@5000:\`
 - `HPCRUN_TRACE=1\
your_executable`

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 platform (Linux, Mac, Windows)
 - hpctraceviewer
 - hpcviewer

A Note for Mac Users

When installing HPCToolkit GUIs on your Mac laptop, don't double click on the zip file and have the Finder unpack them. Instead, unzip them from Terminal. Otherwise, you will end up with broken GUIs, courtesy of overbearing security.

Troubleshooting Deadlock or SEGV on BG/Q

- **Sadly, IBM's PAMI (the implementation layer below MPI) and IBM's XL OpenMP implementations have race conditions that can cause them to fail**
- **Measuring applications with sampling-based performance tools can increase the likelihood that the race conditions will resolve the wrong way, causing deadlock (PAMI) or failure (XL OpenMP)**
- **If you run into problems, the following environment variable settings can disable buggy optimizations in IBM's software**
 - **PAMID_COLLECTIVES=0**
 - **ATOMICS_OPT_LEVEL=0**
- **If you don't run into problems, don't use these settings as they reduce performance**