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

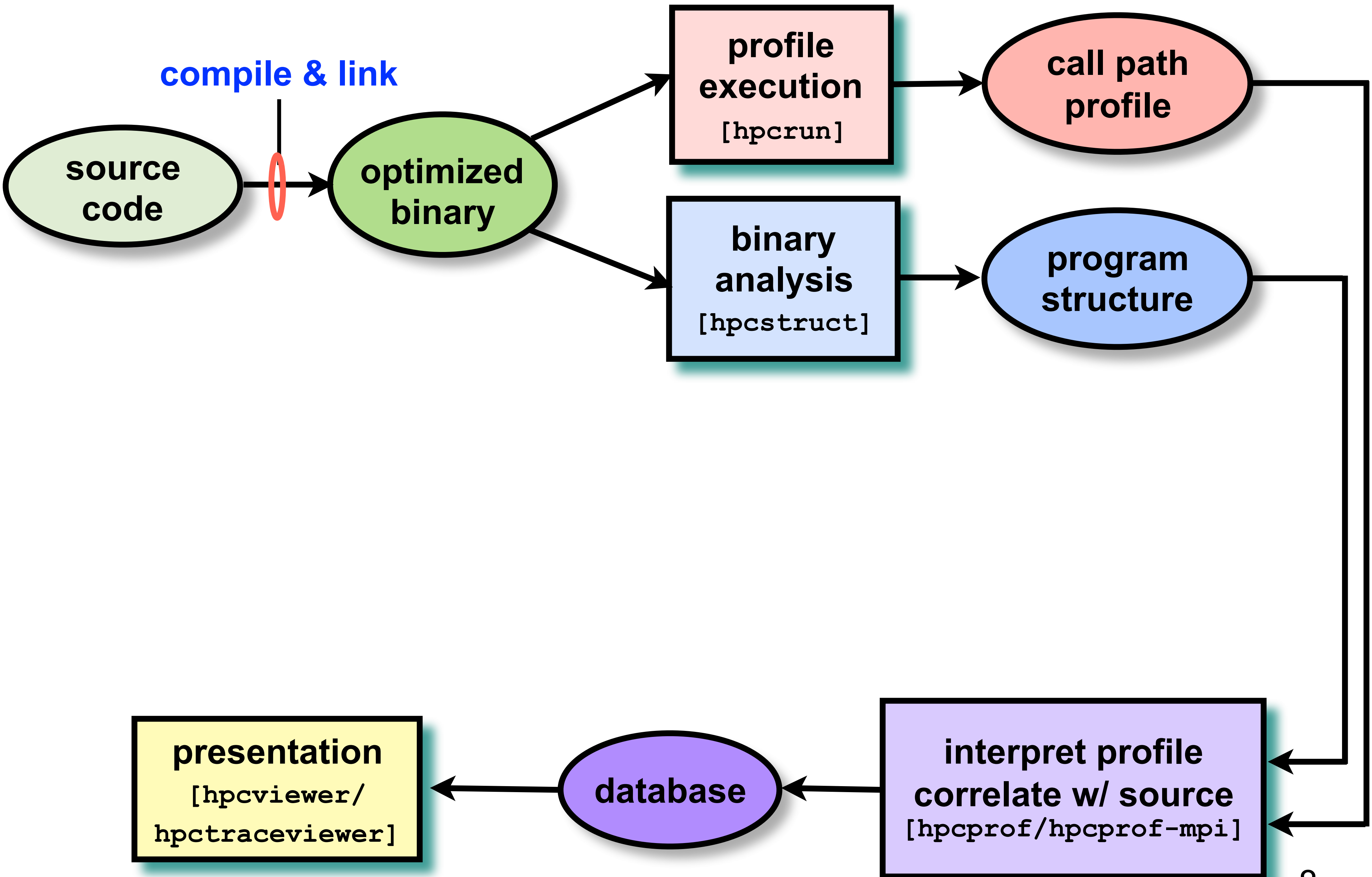
Outline

- **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**

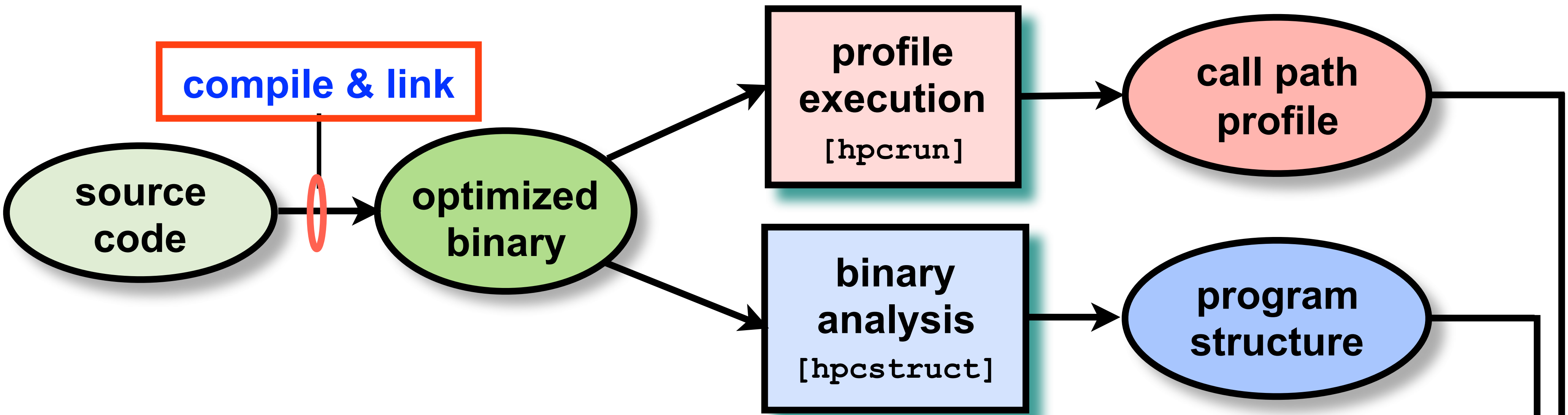
Rice University's HPCToolkit

- 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

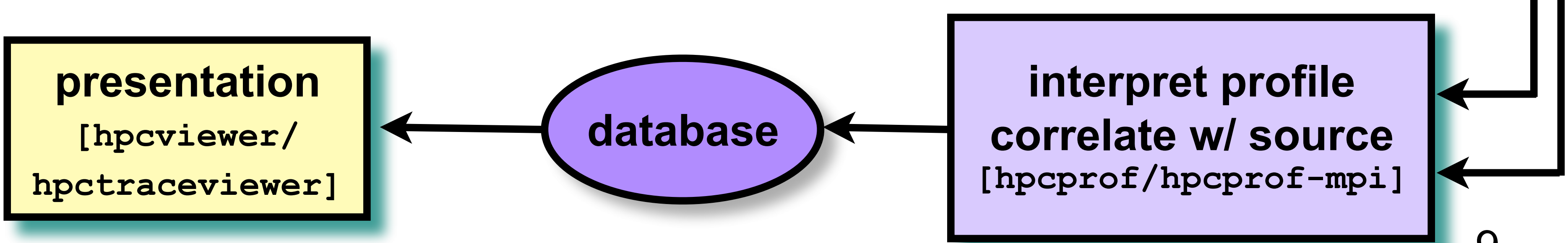


HPCToolkit Workflow

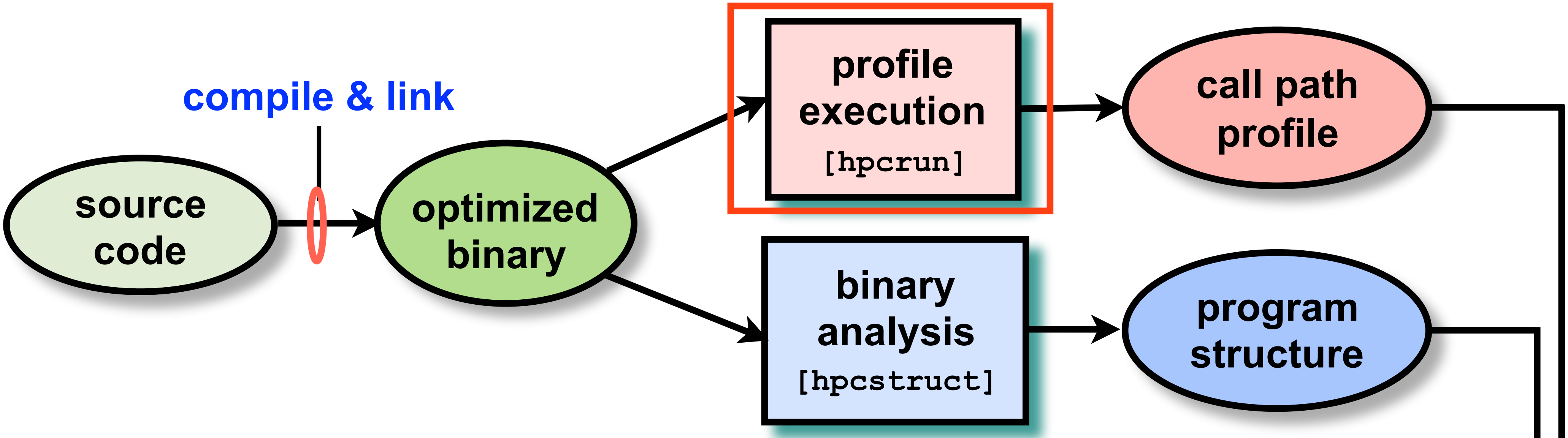


- For dynamically-linked executables, e.g., Linux clusters
  — compile and link as you usually do: nothing special needed

— For statically-linked executables, e.g., Cray, Blue Gene/Q
  — add monitoring by using `hpclink` as prefix to your link line

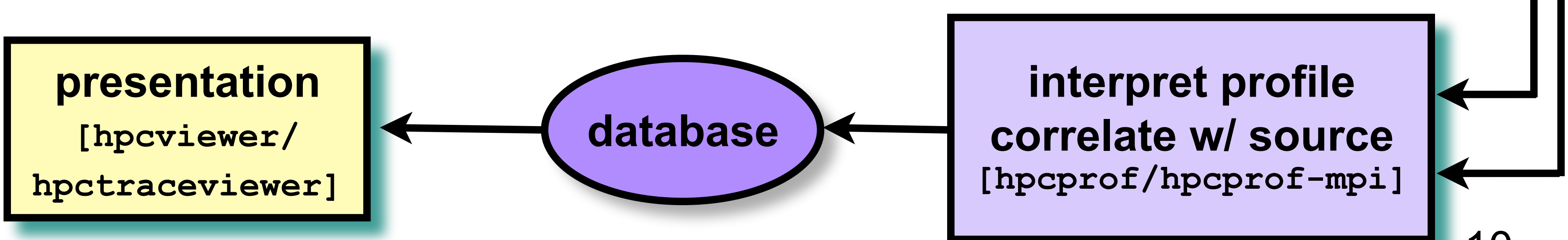


HPCToolkit Workflow



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

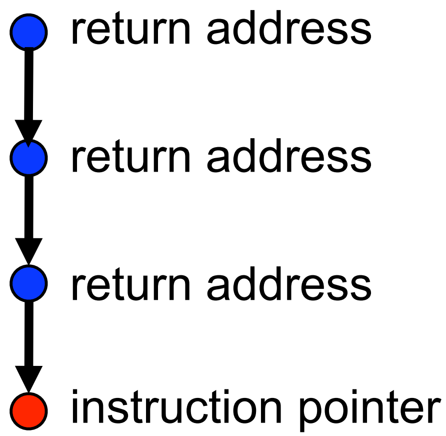


Call Path Profiling

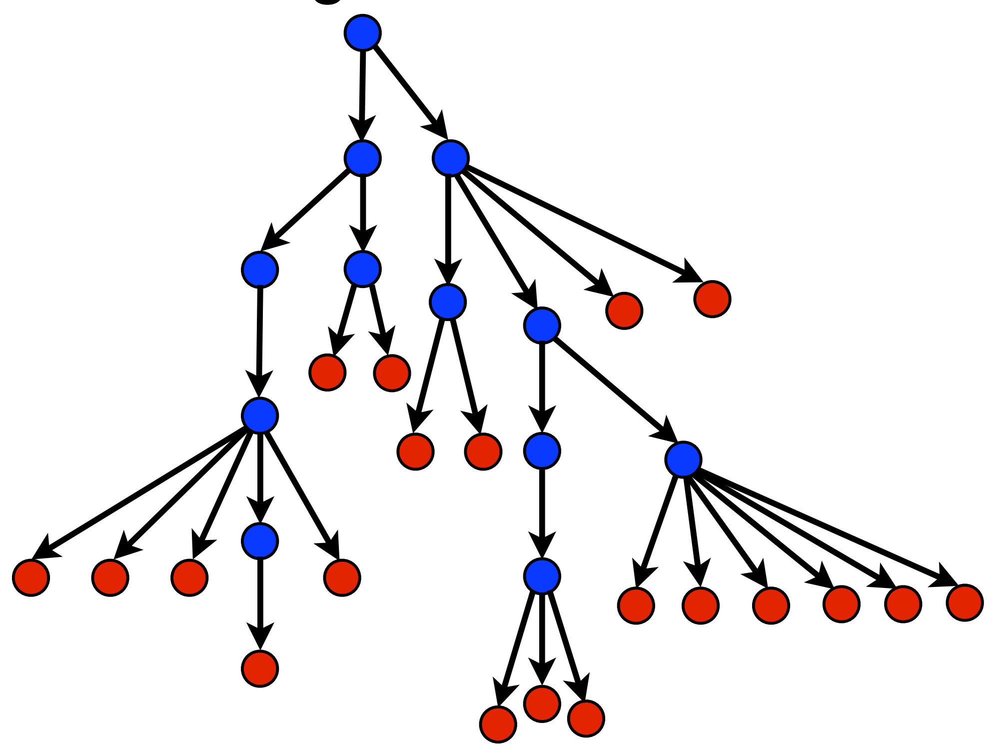
Measure and attribute costs in context

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

Call path sample

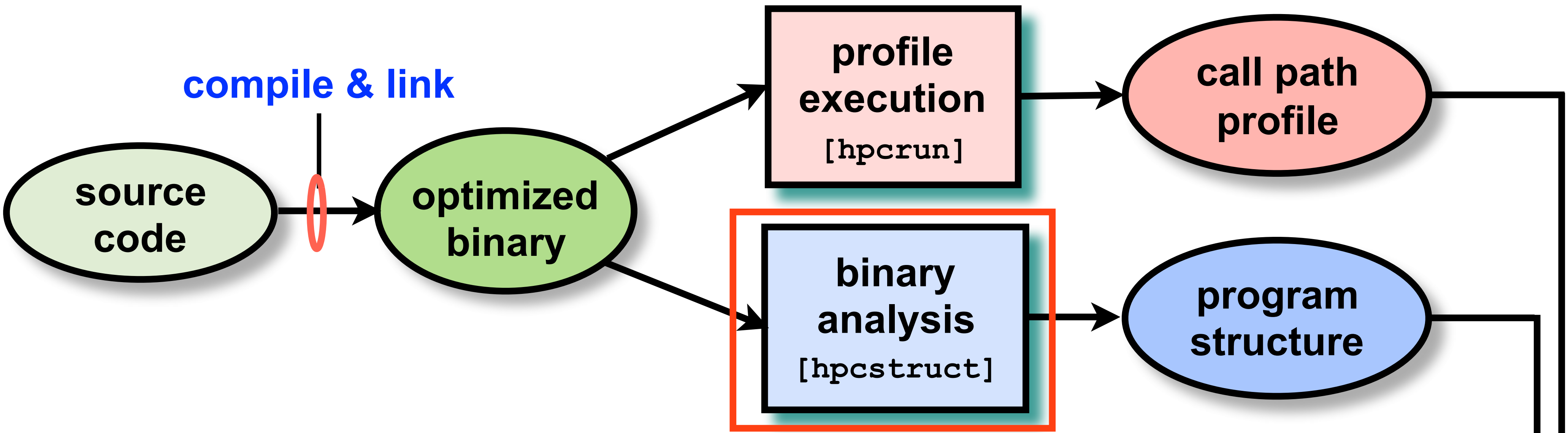


Calling context tree

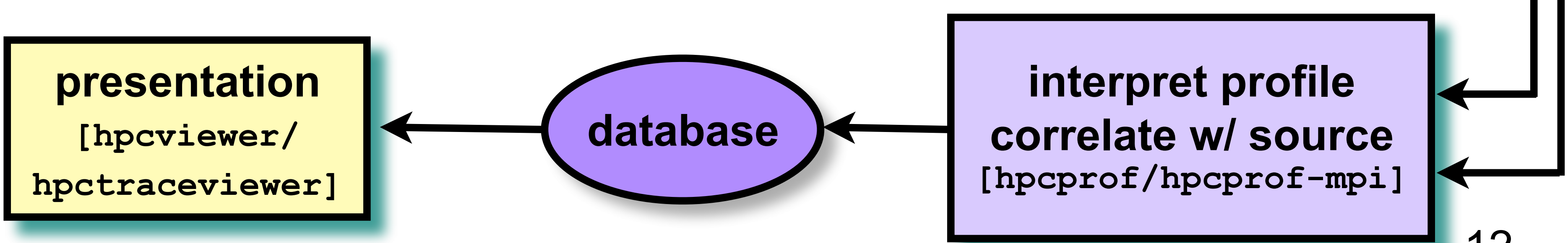


Overhead proportional to sampling frequency...
...not call frequency

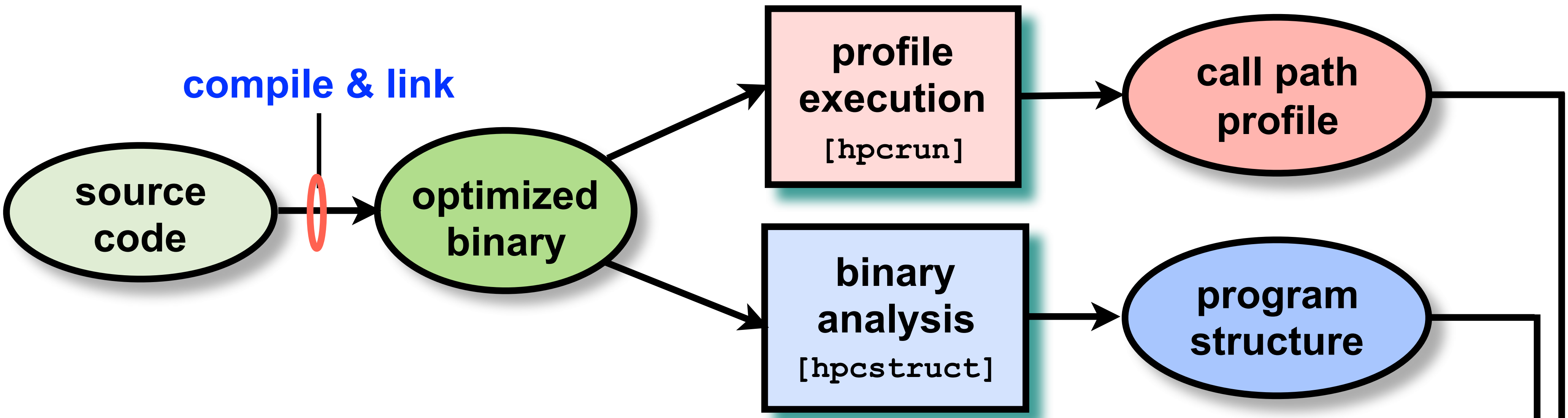
HPCToolkit Workflow



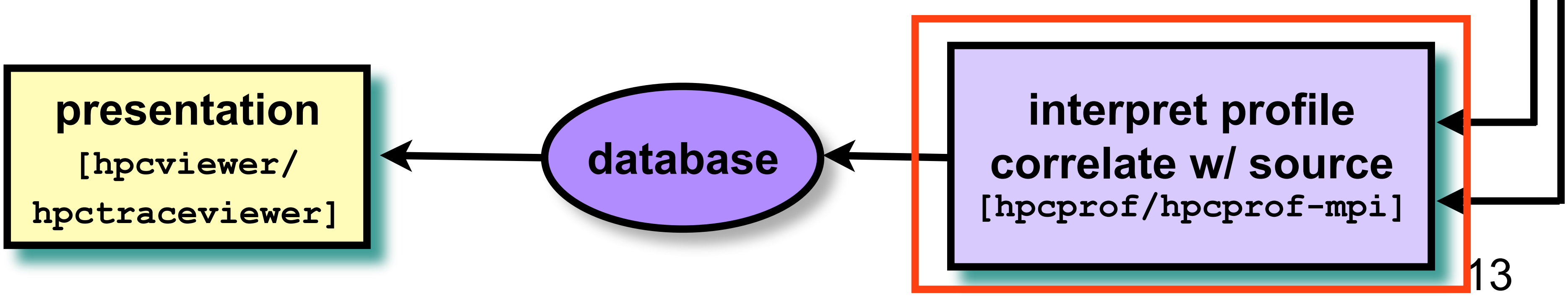
- Analyze binary with `hpcstruct`: recover program structure
  - — analyze machine code, line map, debugging information
  - — extract loop nests & identify inlined procedures
  - — map transformed loops and procedures to source



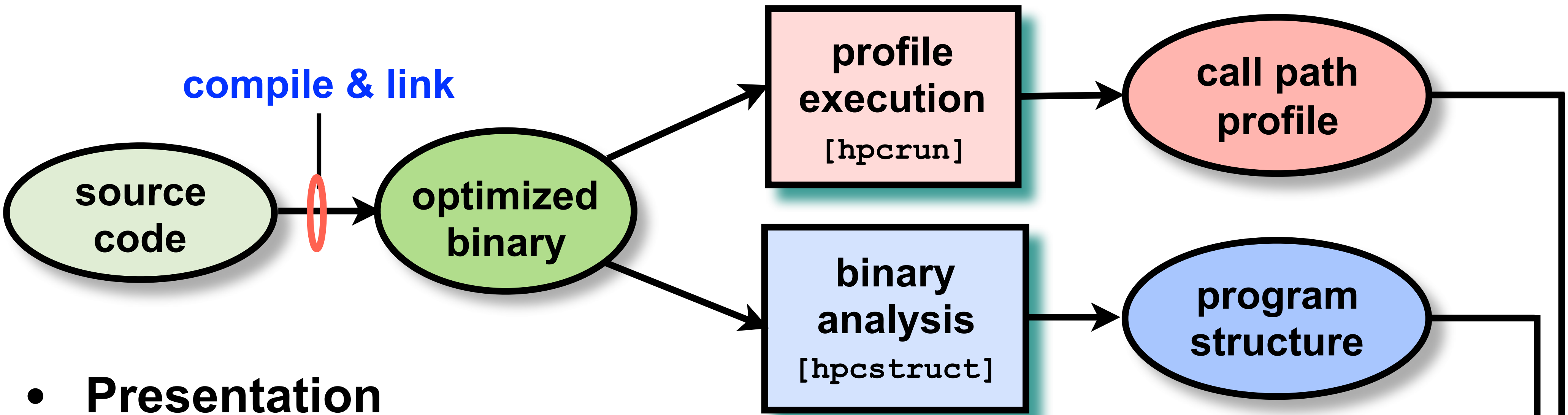
HPCToolkit Workflow



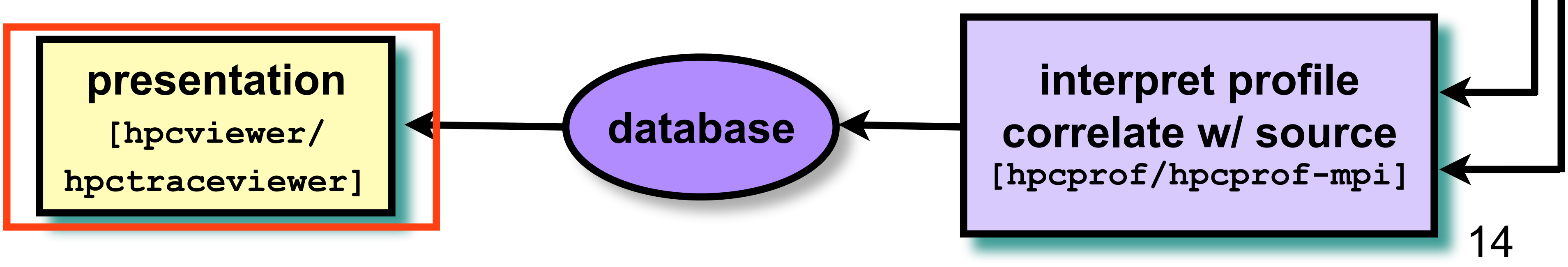
- **Combine multiple profiles**
  - multiple threads; multiple processes; multiple executions
- **Correlate metrics to static & dynamic program structure**



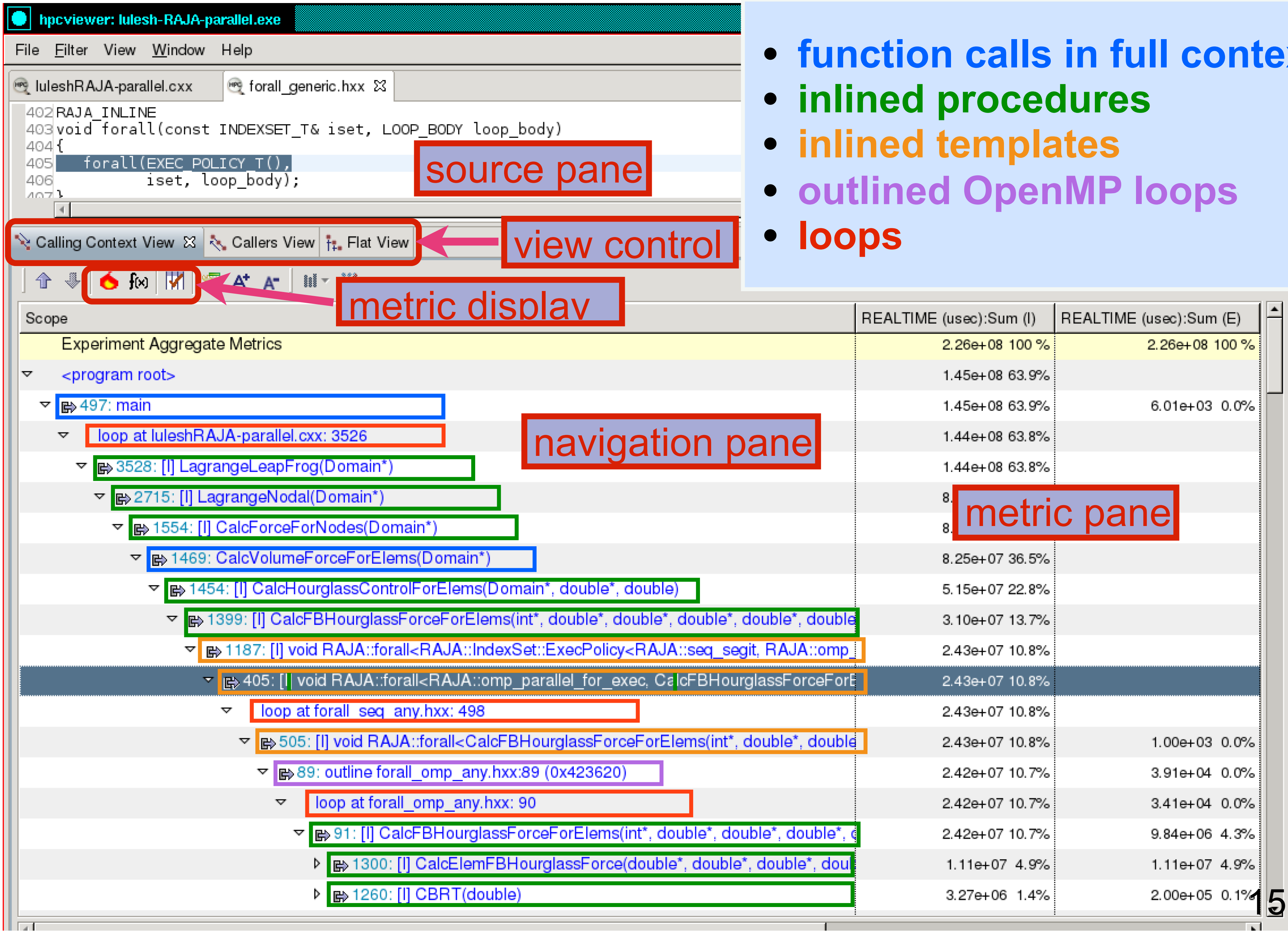
HPCToolkit Workflow



- **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

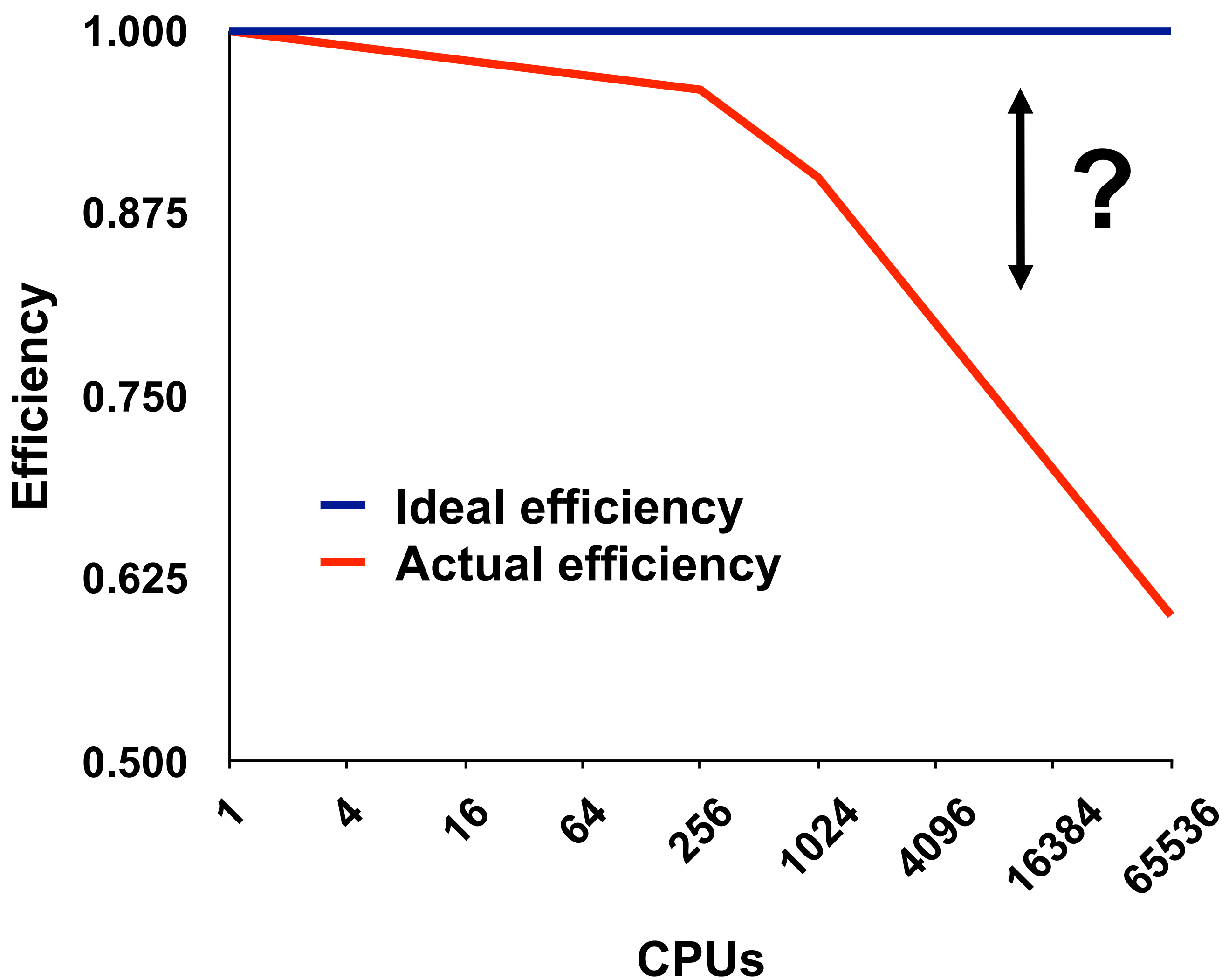


Code-centric Analysis with hpcviewer



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

The Problem of Scaling



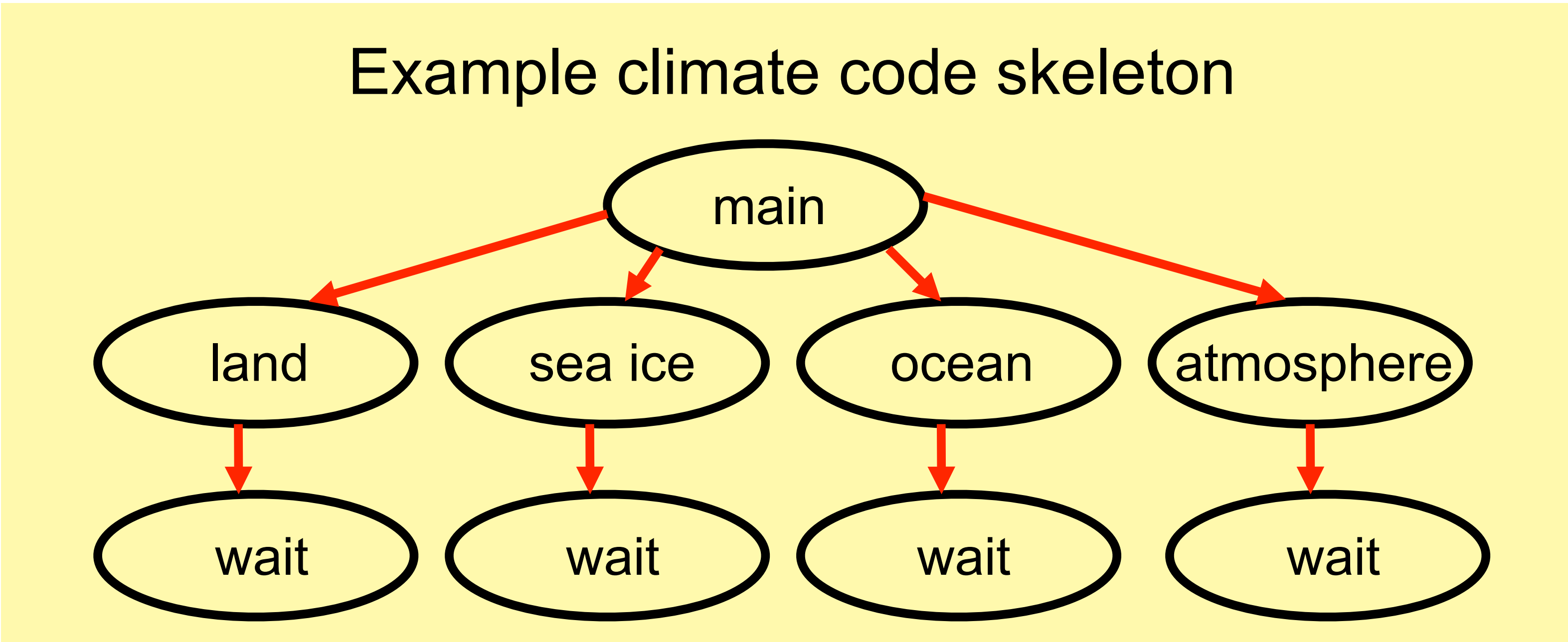
Note: higher is better

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**



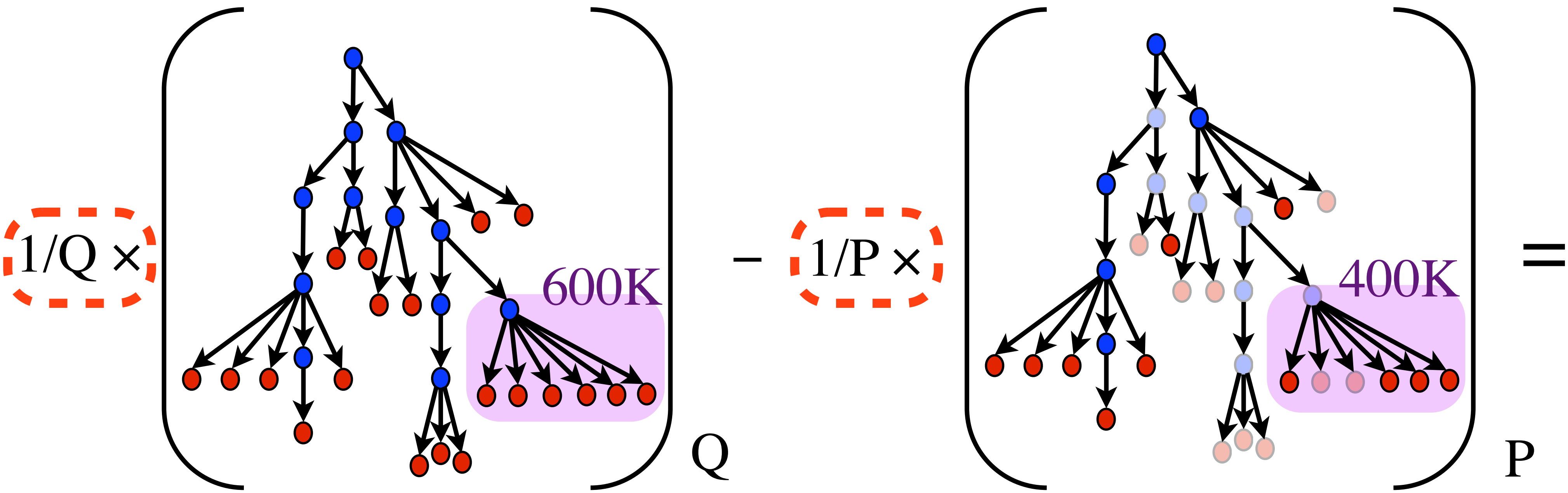
- **Monitoring**
  - — **bottleneck nature: computation, data movement, synchronization?**
  - — **2 pragmatic constraints**
    - – **acceptable data volume**
    - – **low perturbation for use in production runs**

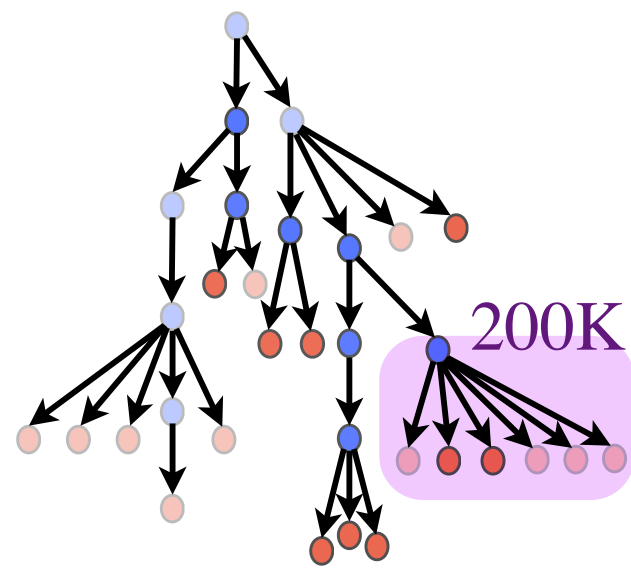
Performance Analysis with Expectations

- 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

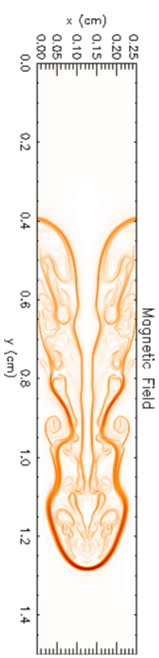




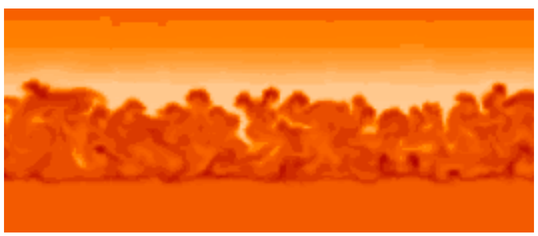
coefficients for analysis of weak scaling

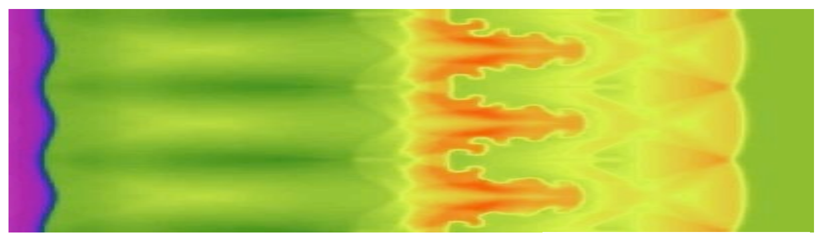
Scalability Analysis Demo

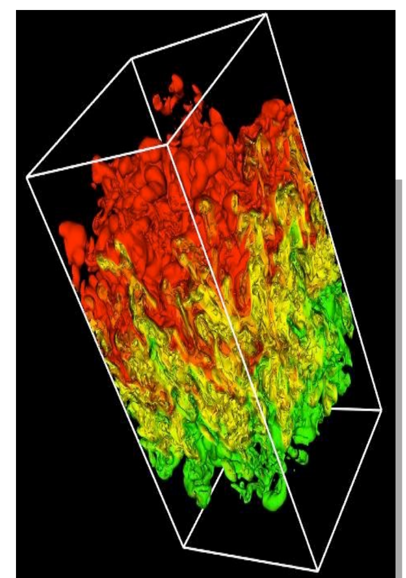
Code:	**University of Chicago FLASH**
Simulation:	**white dwarf detonation**
Platform:	**Blue Gene/P**
Experiment:	**8192 vs. 256 processors**
Scaling type:	**weak**

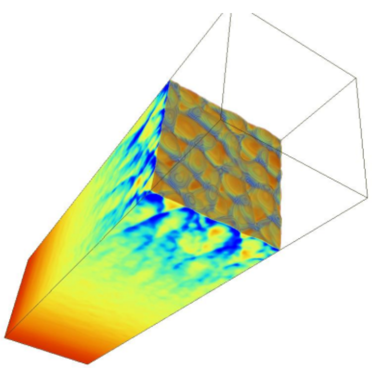


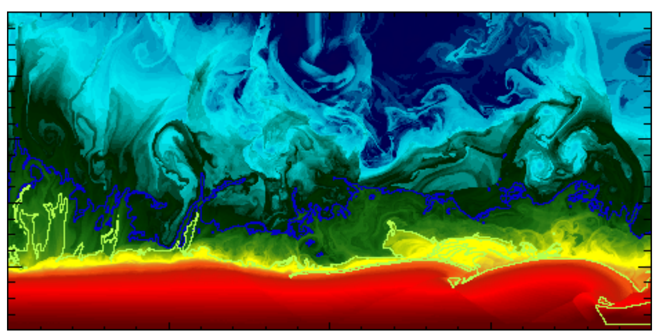
Magnetic Rayleigh-Taylor

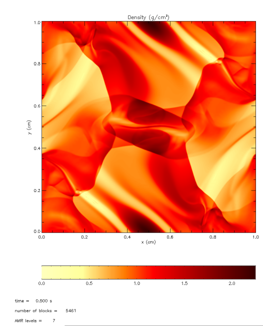

Nova outbursts on white dwarfs


Laser-driven shock instabilities


Rayleigh-Taylor instability

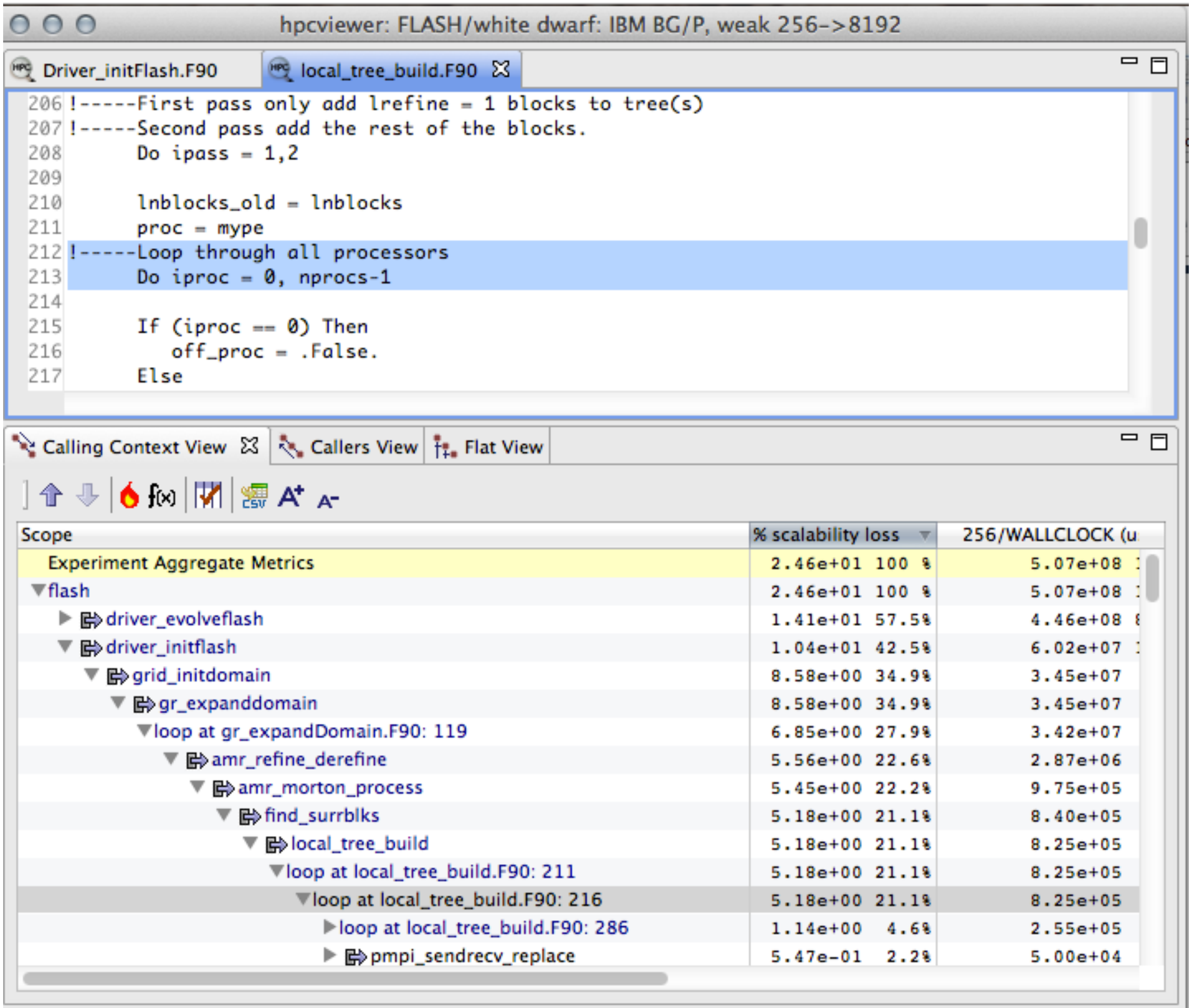

Cellular detonation


Helium burning on neutron stars


Orzag/Tang MHD vortex

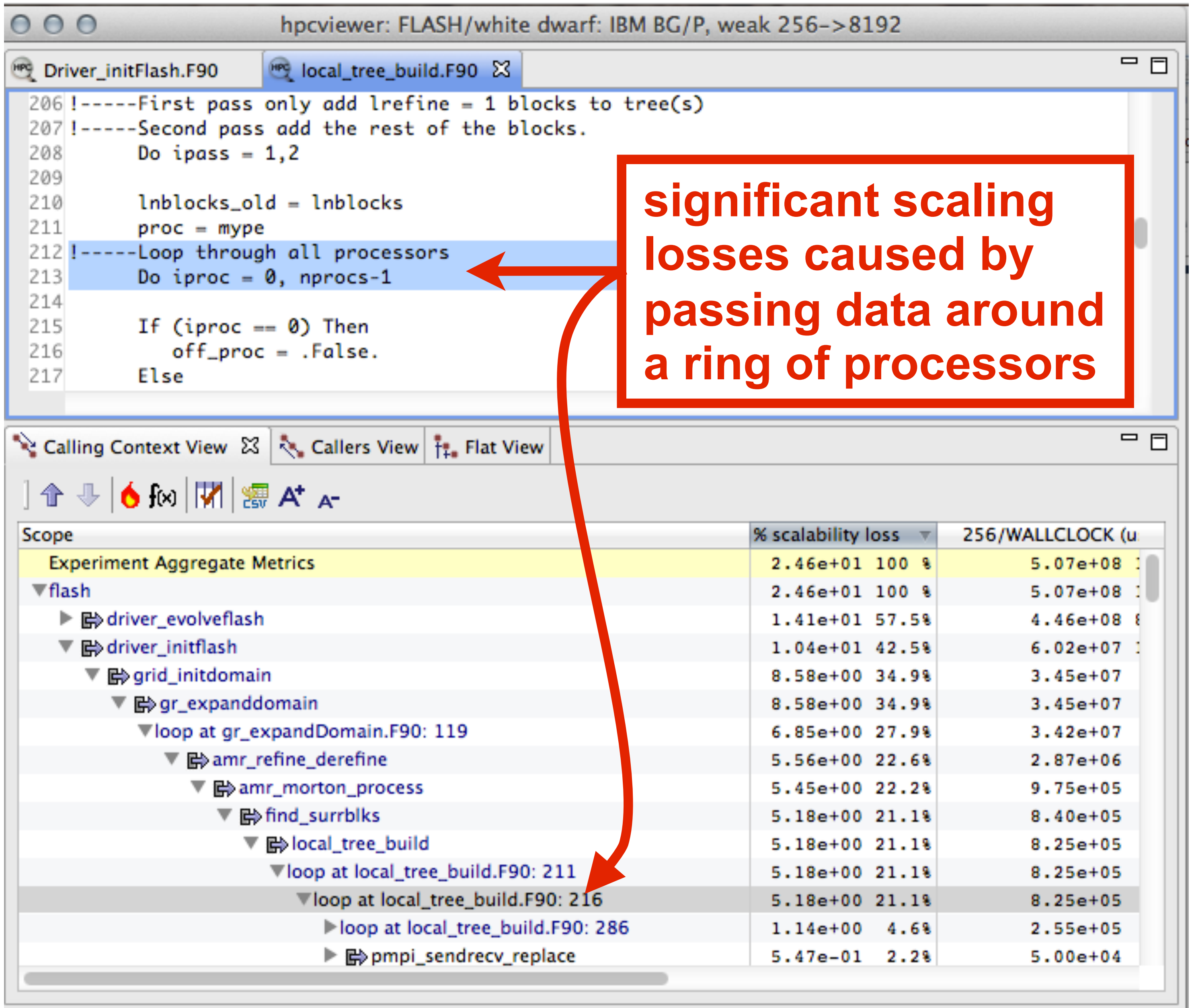
Figures courtesy of FLASH Team, University of Chicago

Scalability Analysis of Flash (Demo)

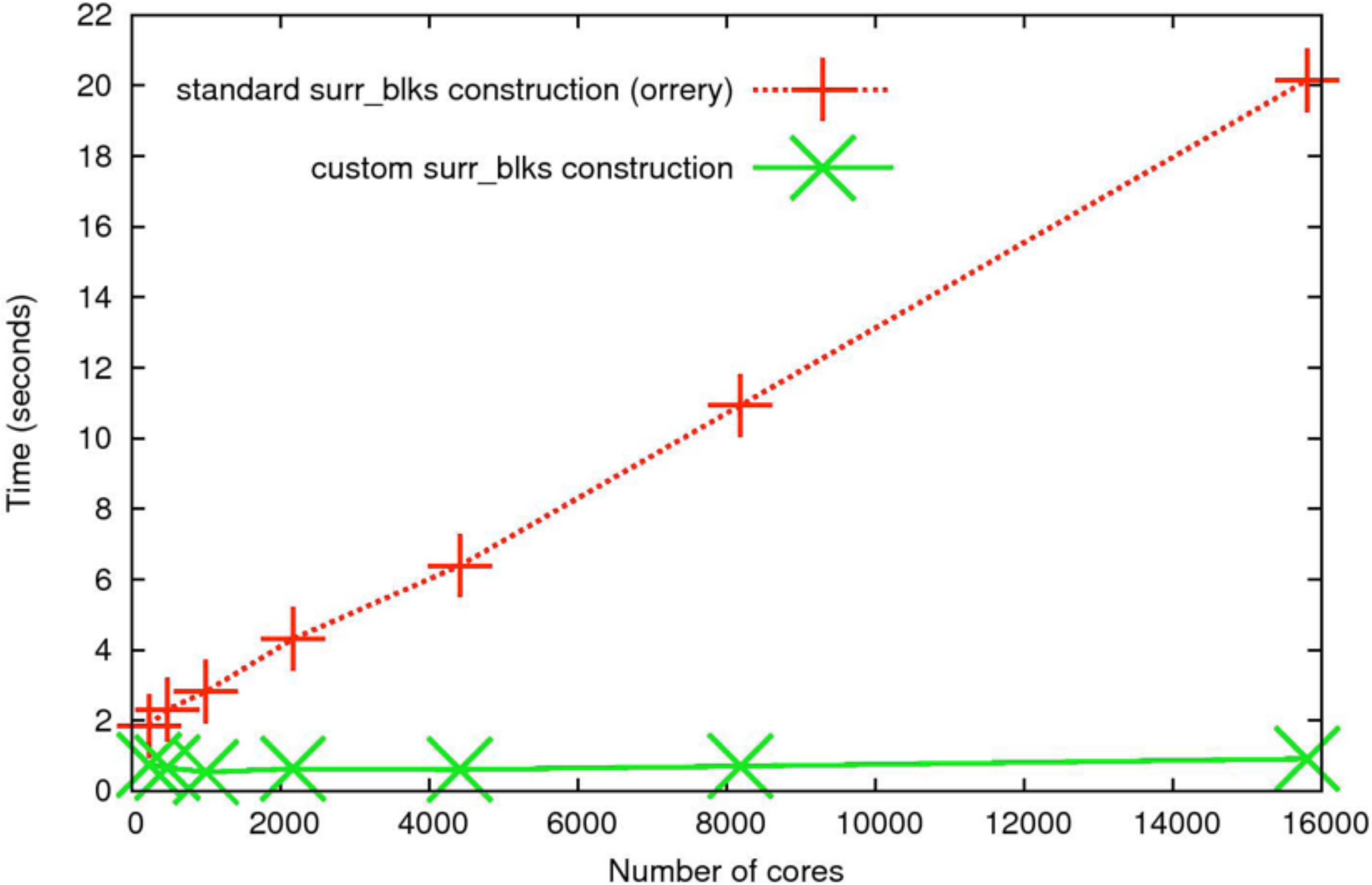


Scalability Analysis

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



Improved Flash Scaling of AMR Setup

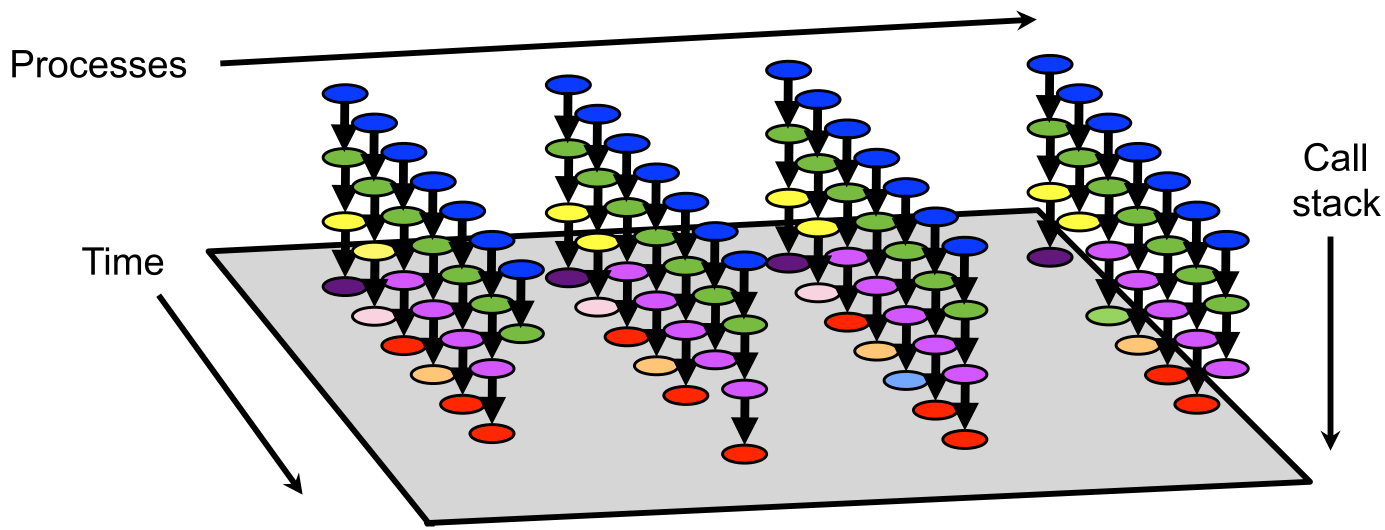


Graph courtesy of Anshu Dubey, U Chicago

Understanding Temporal Behavior

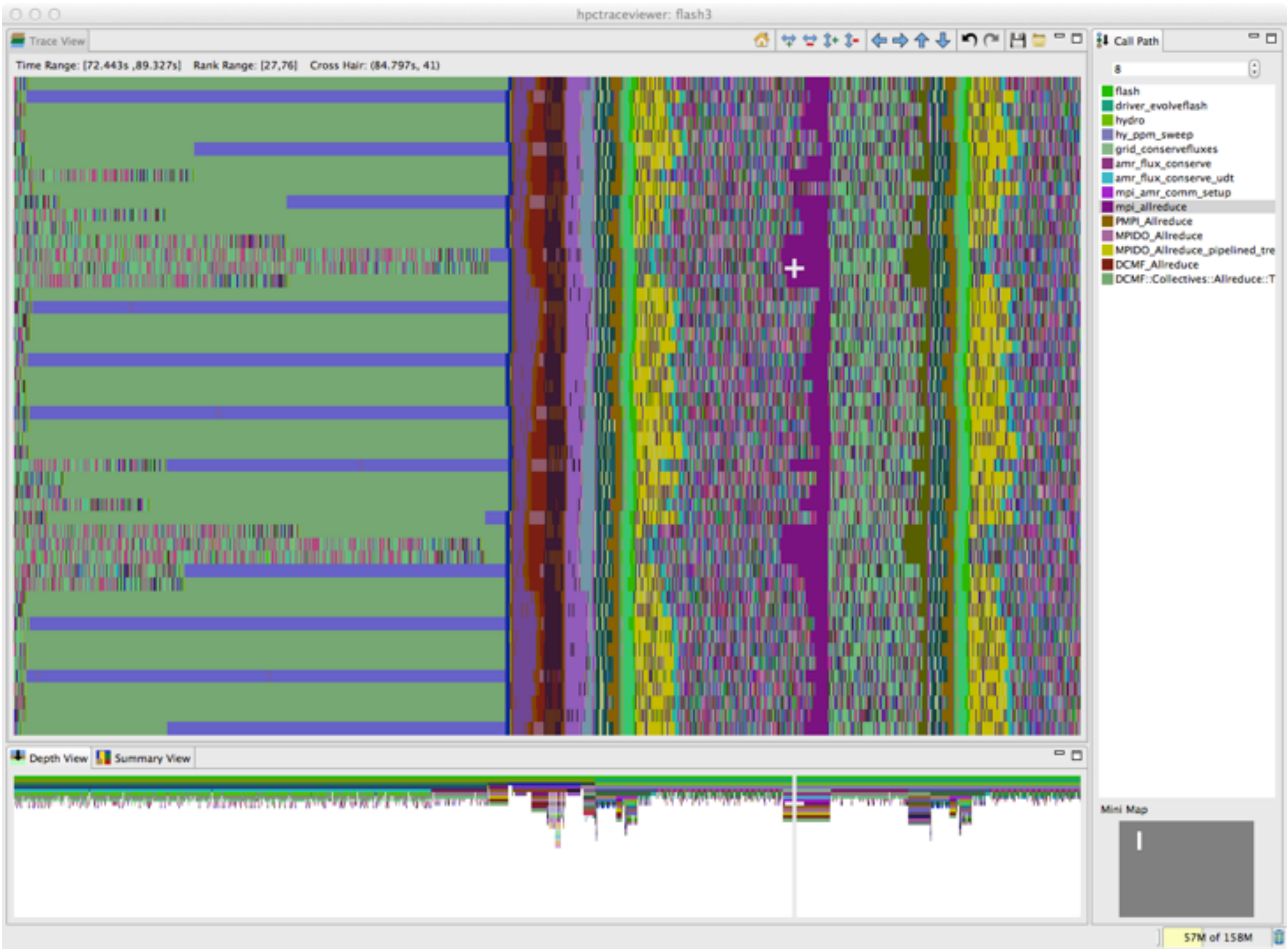
- **Profiling compresses out the temporal dimension**
  - —**temporal patterns, e.g. serialization, are invisible in profiles**
- **What can we do? Trace call path samples**
  - —**sketch:**
    - **N times per second, take a call path sample of each thread**
    - **organize the samples for each thread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



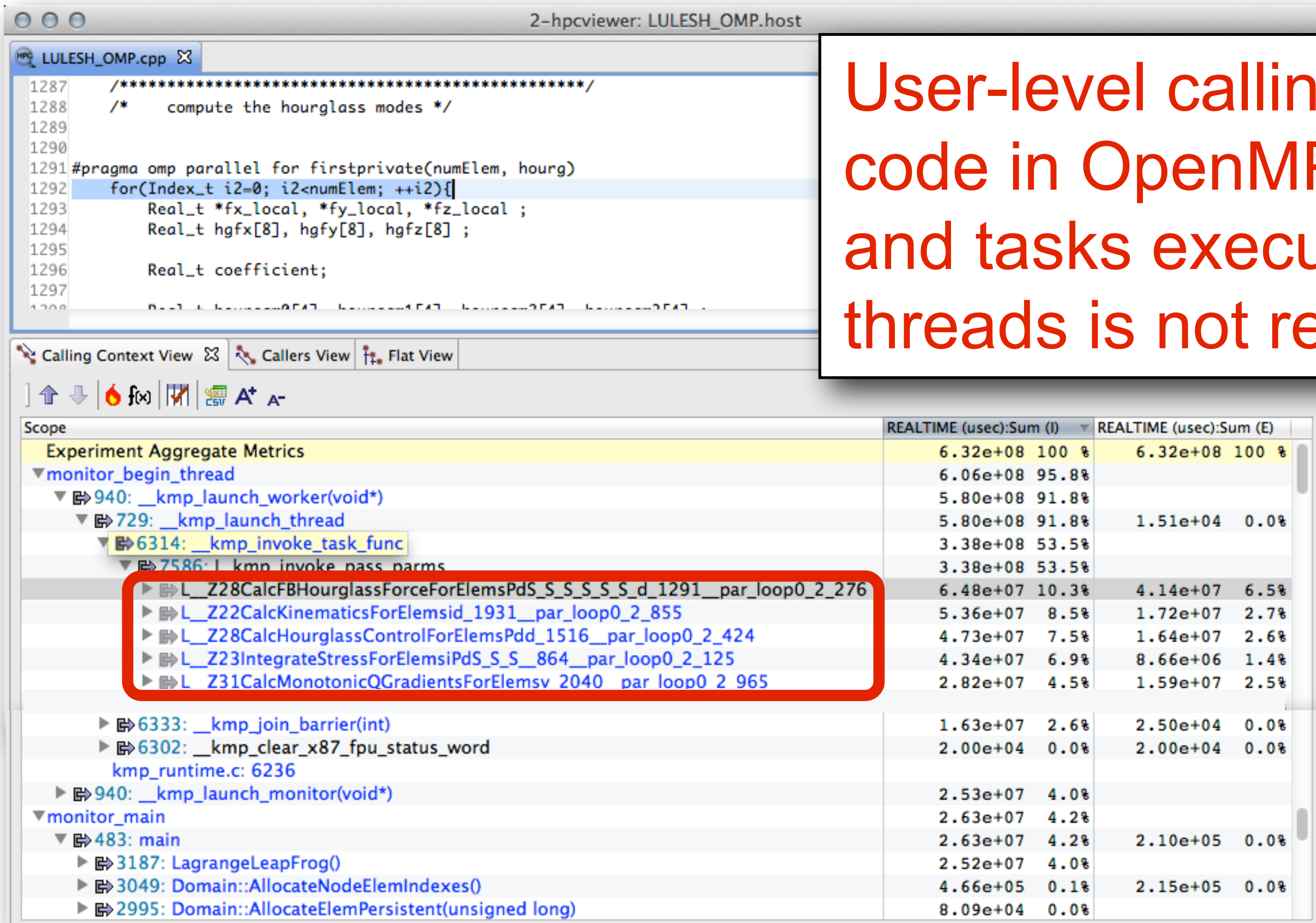
hpctraceviewer: detail of FLASH@256PE

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**



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

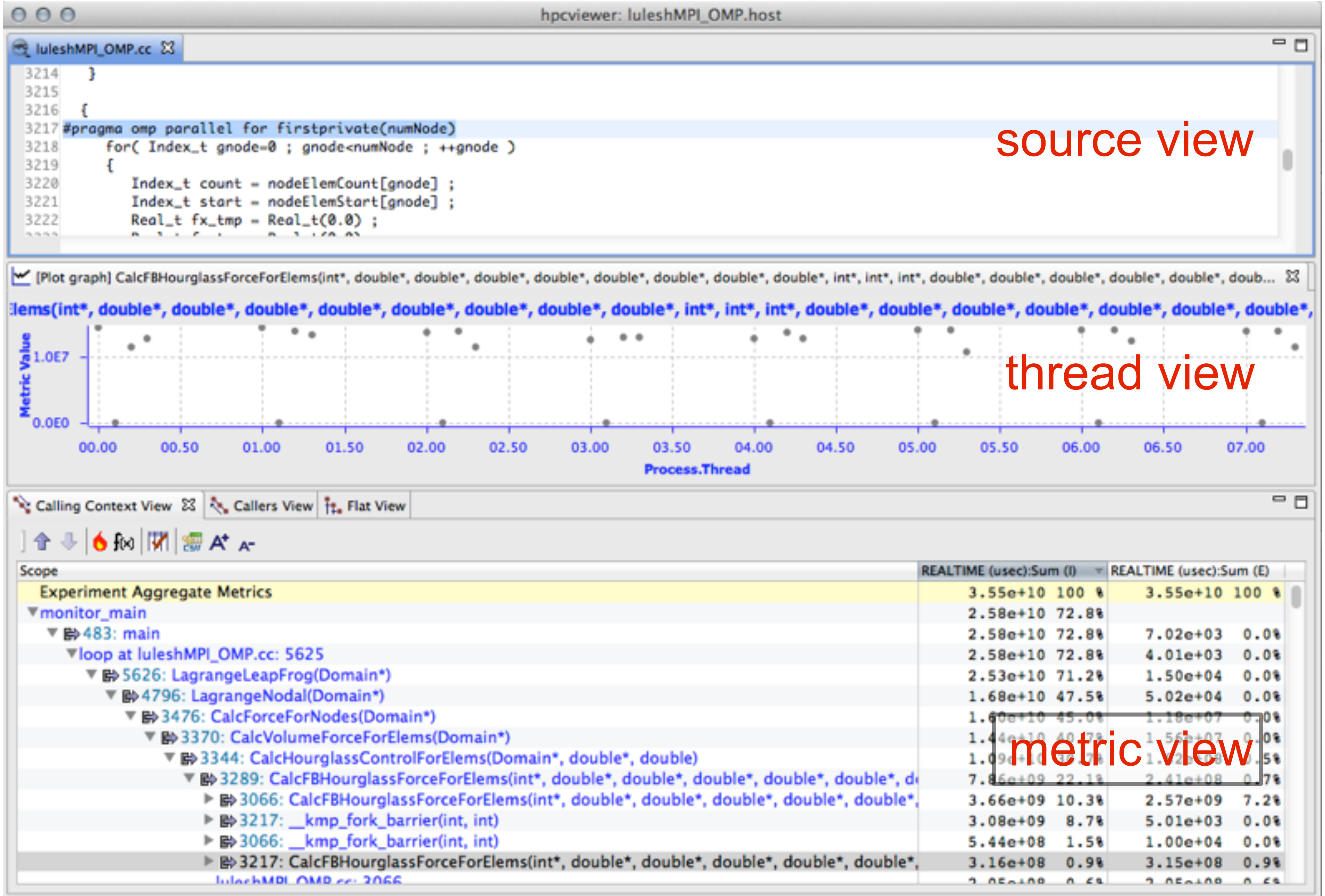
- Tools provide implementation-level view of OpenMP threads
  - asymmetric threads
    - master thread
    - worker thread
  - run-time frames are interspersed with user code
- Hard to understand causes of idleness
  - long serial sections
  - load imbalance in parallel regions
  - waiting for critical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

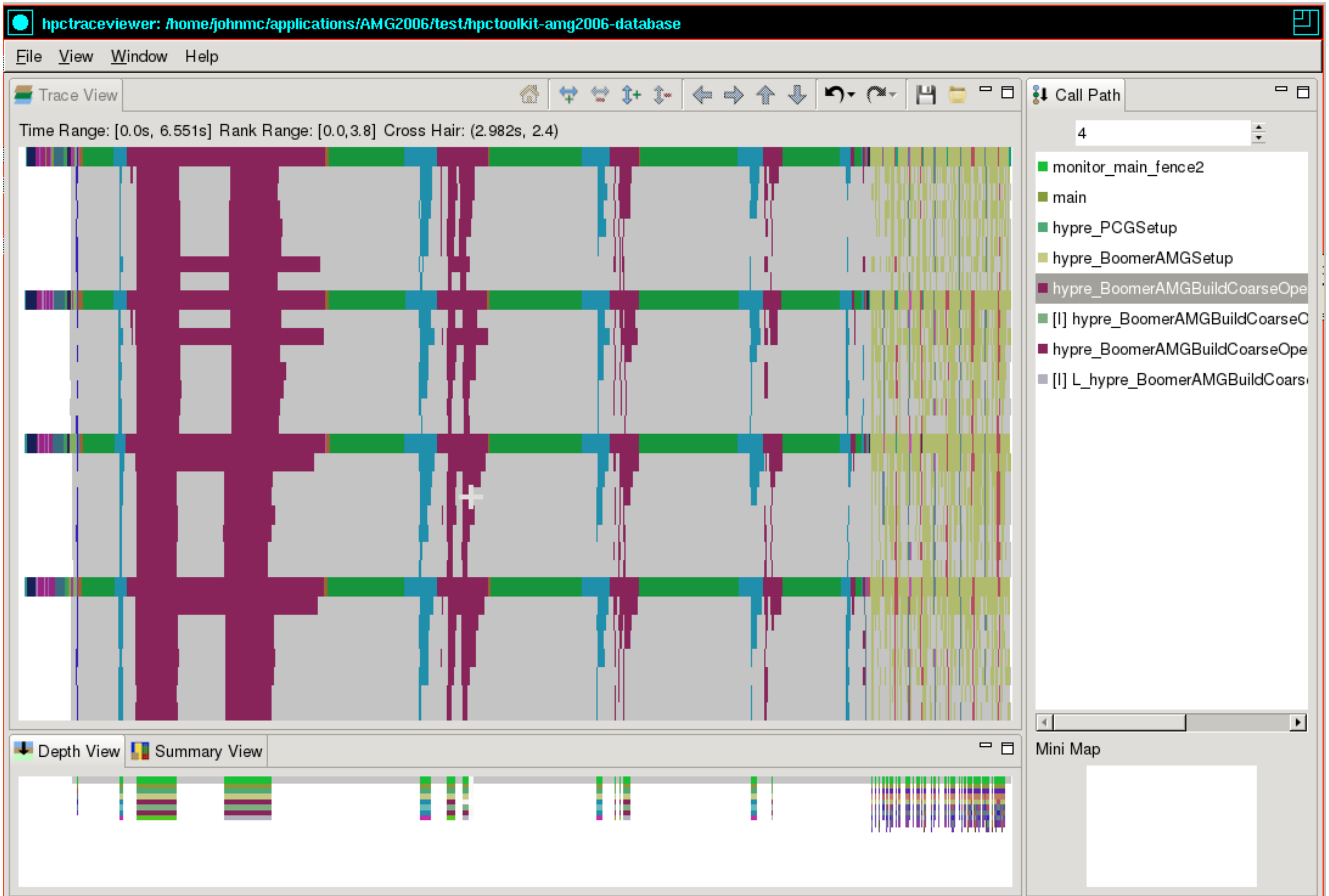
Integrated View of MPI+OpenMP with OMPT

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

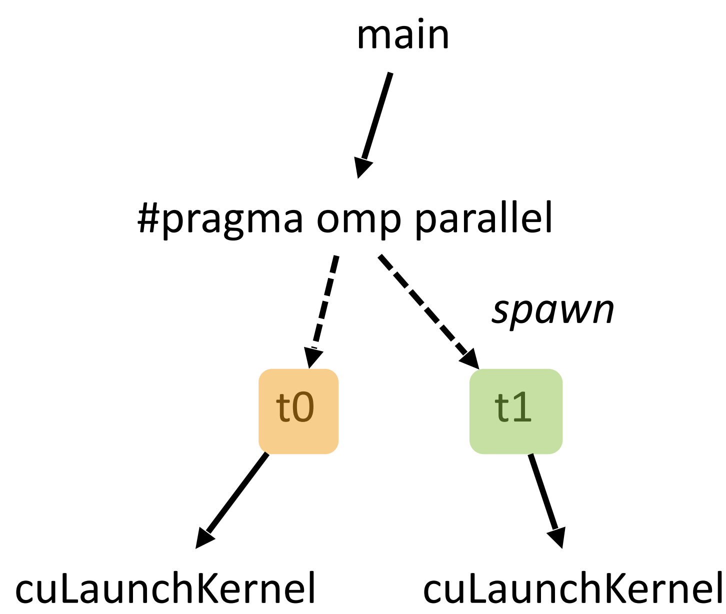
Two threads launch vecAdd kernels concurrently

```
#omp parallel num_threads(2)
  cuLaunchKernel(vecAdd, ...)

int __noinline__ add(int a, int b) {
  return a + b;
}

void vecAdd(int *l, int *r, int *p, size_t iter1, size_t iter2) {
  size_t idx = blockDim.x * blockIdx.x + threadIdx.x;
  for (size_t i = 0; i < iter1; ++i) {
    p[idx] = add(l[idx], r[idx]);
  }
  for (size_t i = 0; i < iter2; ++i) {
    p[idx] = add(l[idx], r[idx]);
  }
}
```

Collect CPU Calling Context for GPU Work

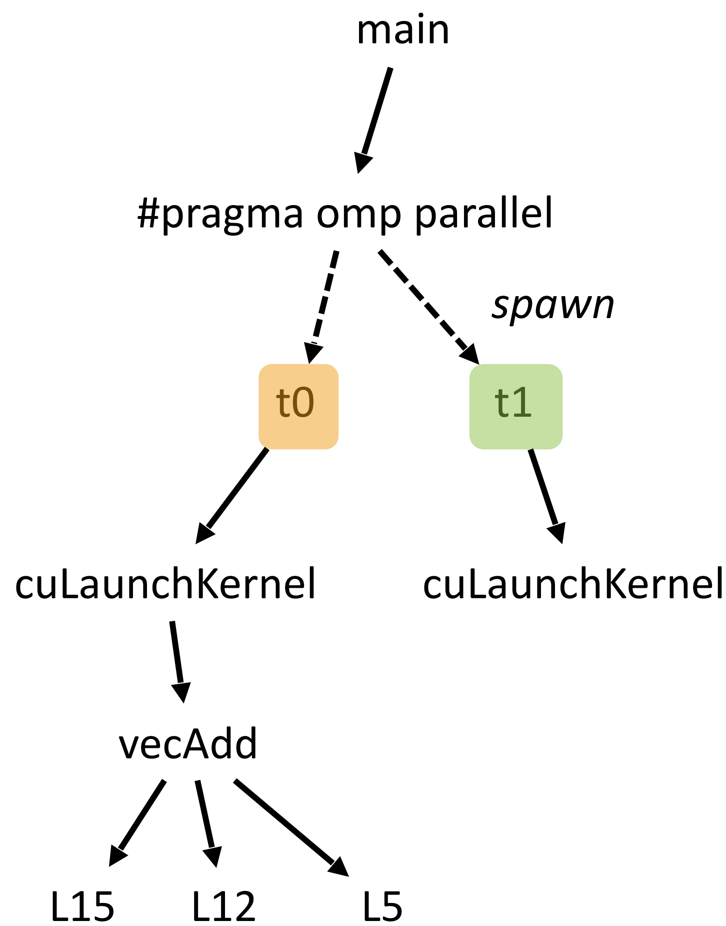
```
#omp parallel num_threads(2)
  cuLaunchKernel(vecAdd, ...)

int __noinline__ add(int a, int b) {
  return a + b;
}

void vecAdd(int *l, int *r, int *p,
  size_t iter1, size_t iter2) {
  size_t idx = blockDim.x * blockIdx.x
                  + threadIdx.x;
  for (size_t i = 0; i < iter1; ++i) {
    p[idx] = add(l[idx], r[idx]);
  }
  for (size_t i = 0; i < iter2; ++i) {
    p[idx] = add(l[idx], r[idx]);
  }
```

CPU Calling Context



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   }
```



Lx: samples collected at Line x

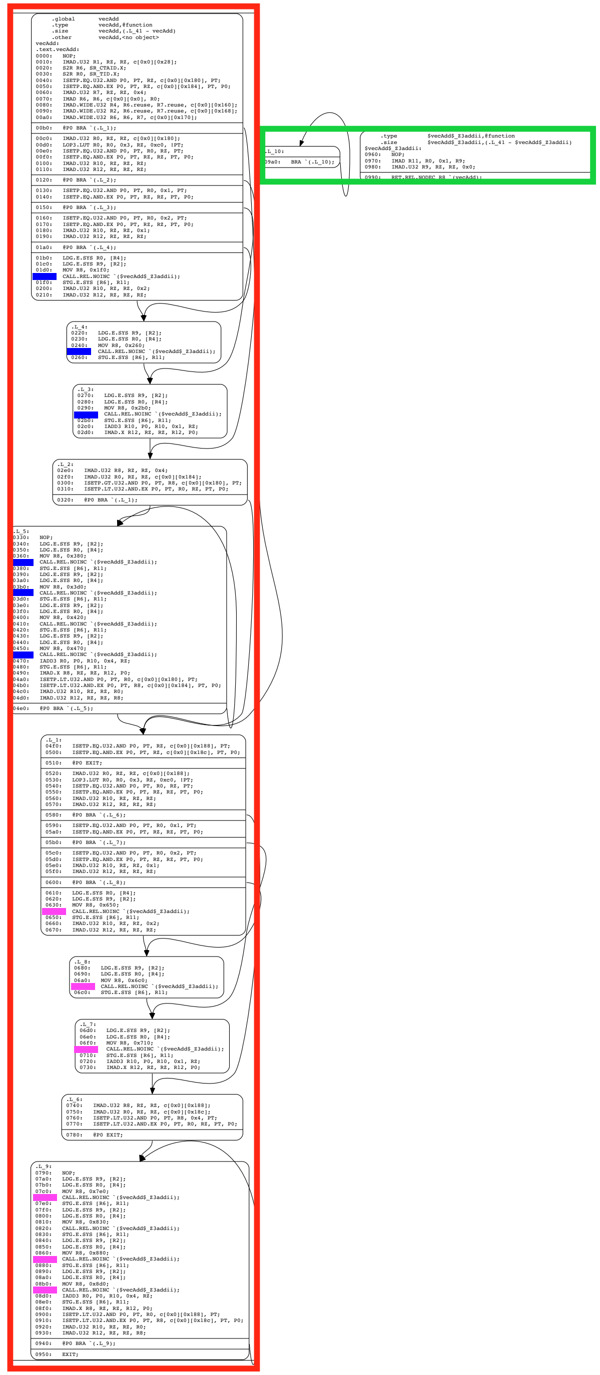
NVIDIA PC Sampling records flat samples

Attribution for GPU binaries

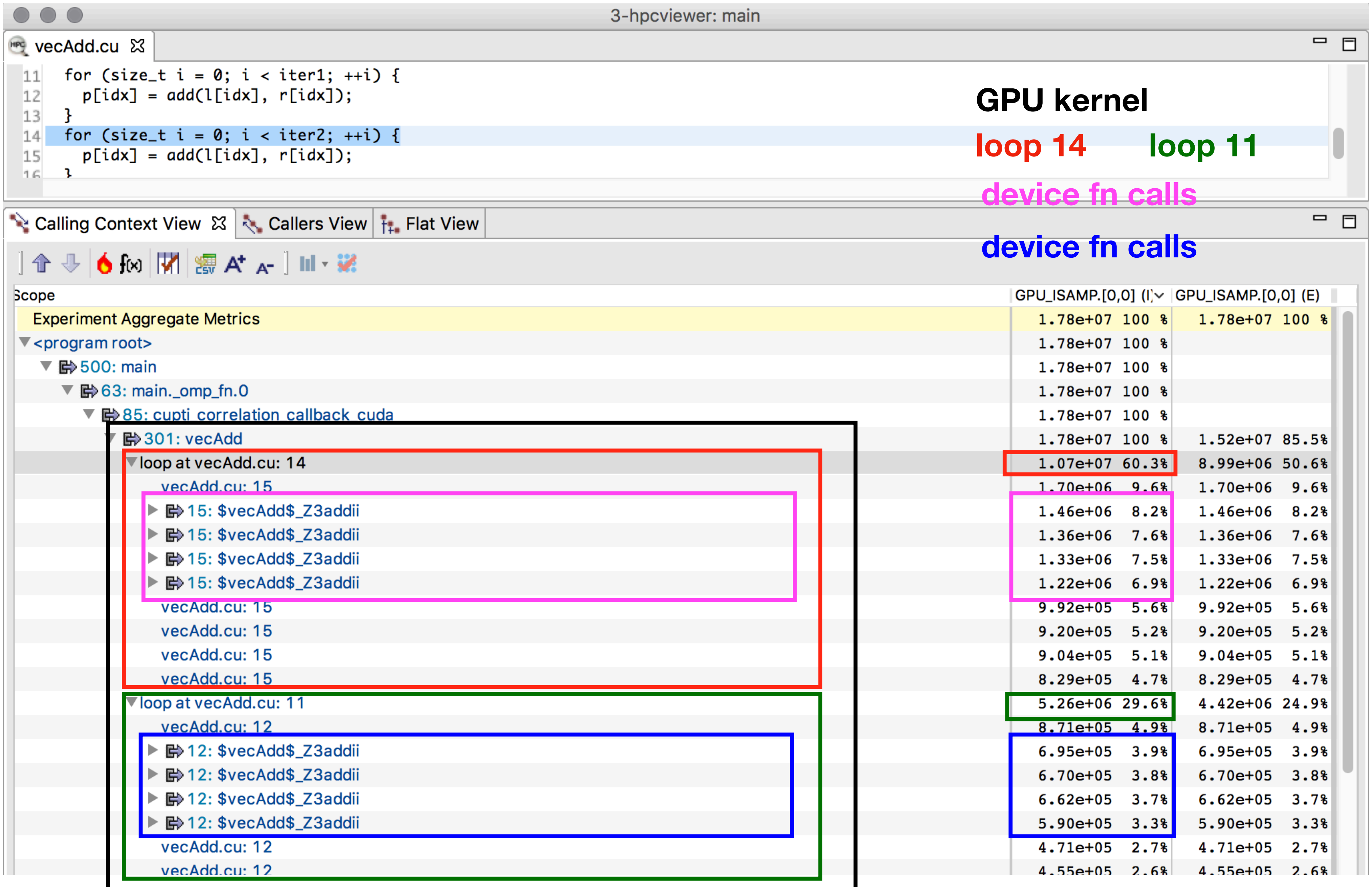
- **Analyze loop nests in NVIDIA CUBIN GPU binaries**
  - — invoke "hpcstruct" on an hpctoolkit measurement directory to analyze any CUBINs collected at runtime'
  - — results of such analysis will be automatically be integrated into the profiling result
- **Approximately reconstruct GPU call paths from 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

```
__device__
int __attribute__ ((noinline)) add(int a, int b) {
  return a + b;
}


extern "C"
__global__
void vecAdd(int *l, int *r, int *p, size_t N, size_t iter1,
      size_t iter2) {
  size_t idx = blockDim.x * blockIdx.x + threadIdx.x;
  for (size_t i = 0; i < iter1; ++i) {
    p[idx] = add(l[idx], r[idx]);
  }
  for (size_t i = 0; i < iter2; ++i) {
    p[idx] = add(l[idx], r[idx]);
  }
}

```

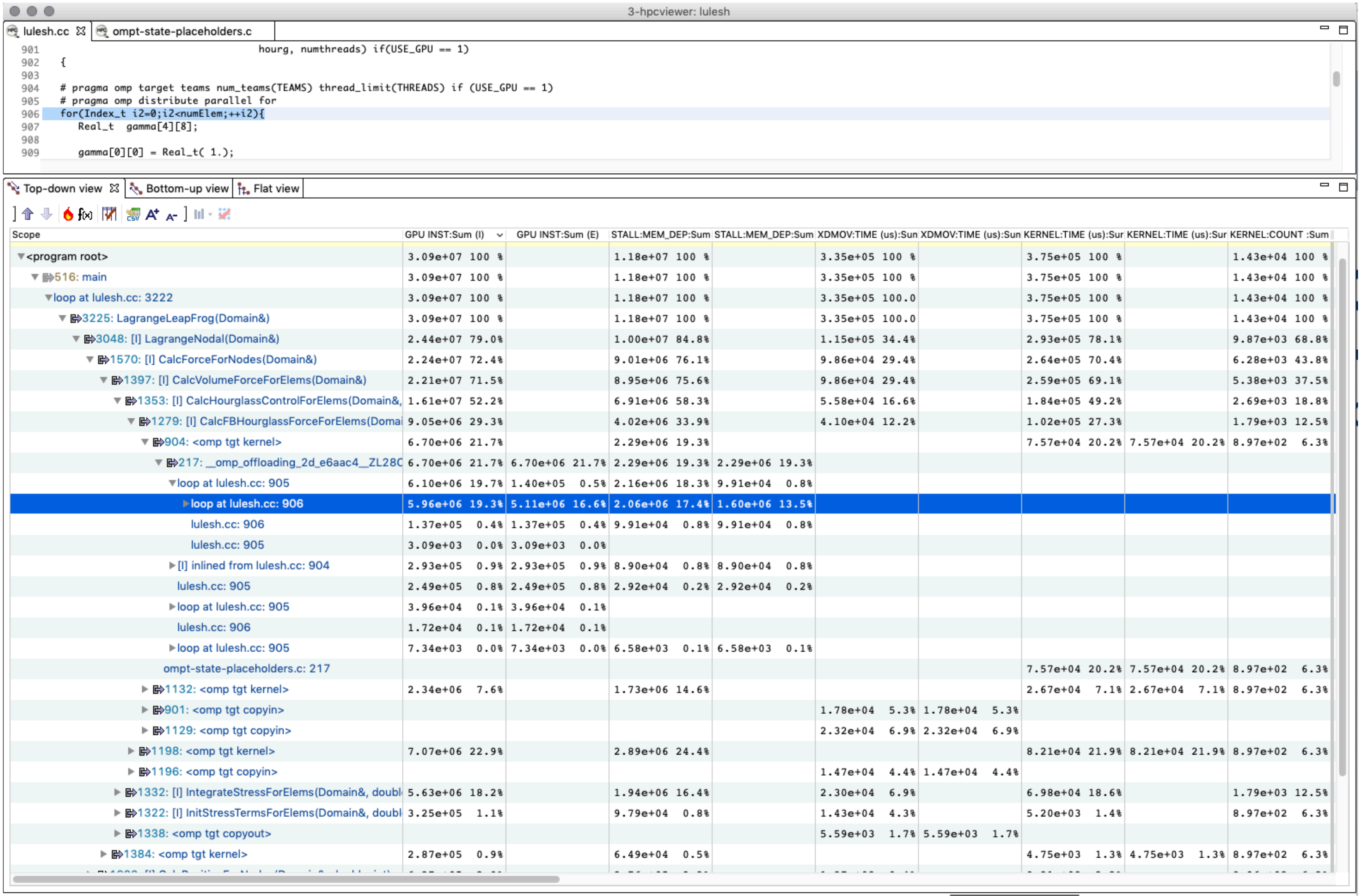


Profiling Result for VecAdd CUDA Example



HPCToolkit Capabilities for GPU Code

MPI + OpenMP 4.5 or CUDA GPU accelerated applications



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/pkgs-vesta/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.