HPCToolkit: Sampling-based Performance Tools for Leadership Computing

John Mellor-Crummey
Department of Computer Science
Rice University
johnmc@cs.rice.edu

SciDAC Scientific Discovery through Advanced Computing

`http://hpctoolkit.org`

RICE

Acknowledgments

- **Research Staff**
  - Nathan Tallent, Laksono Adhianto, Mike Fagan, Mark Krentel
- **Student**
  - Xu Liu
- **Alumni**
  - Gabriel Marin (ORNL), Robert Fowler (RENCI), Nathan Froyd (CodeSourcery)
- **SciDAC project support**
  - Center for Scalable Application Development Software
    - Cooperative agreement number DE-FC02-07ER25800
  - Performance Engineering Research Institute
    - Cooperative agreement number DE-FC02-06ER25762

Challenges

- **Gap between typical and peak performance is huge**
- **Complex architectures are harder to program effectively**
  - — processors that are pipelined, out of order, superscalar
  - — multi-level memory hierarchy
  - — multi-level parallelism: multi-core, SIMD instructions
- **Complex applications present challenges**
  - — for measurement and analysis
  - — for understanding and tuning
- **Leadership computing platforms pose additional challenges**
  - — unique microkernel-based operating systems
  - — immense scale
  - — more than just computation: communication, I/O

Performance Analysis Principles

- **Without accurate measurement, analysis is irrelevant**
  - avoid systematic measurement error
    - instrumentation-based measurement is often problematic
  - measure actual execution of interest, not an approximation
    - fully optimized production code on the target platform
- **Without effective analysis, measurement is irrelevant**
  - pinpoint and explain problems in terms of source code
    - binary-level measurements, source-level insight
  - compute insightful metrics
    - "unused bandwidth" or "unused flops" rather than "cycles"
- **Without scalability, a tool is irrelevant**
  - large codes
  - large-scale parallelism, including MPI + OpenMP hybrid

Performance Analysis Goals

- Accurate measurement of complex parallel codes
  - large, multi-lingual programs
  - fully optimized code: loop optimization, templates, inlining
  - binary-only libraries, sometimes partially stripped
  - complex execution environments
    - dynamic loading (e.g. Linux clusters) vs. static linking (Cray XT, BG/P)
    - SPMD parallel codes with threaded node programs
    - batch jobs
- Effective performance analysis
  - insightful analysis that pinpoints and explains problems
    - correlate measurements with code (yield actionable results)
    - intuitive enough for scientists and engineers
    - detailed enough for compiler writers
- Scalable to petascale systems

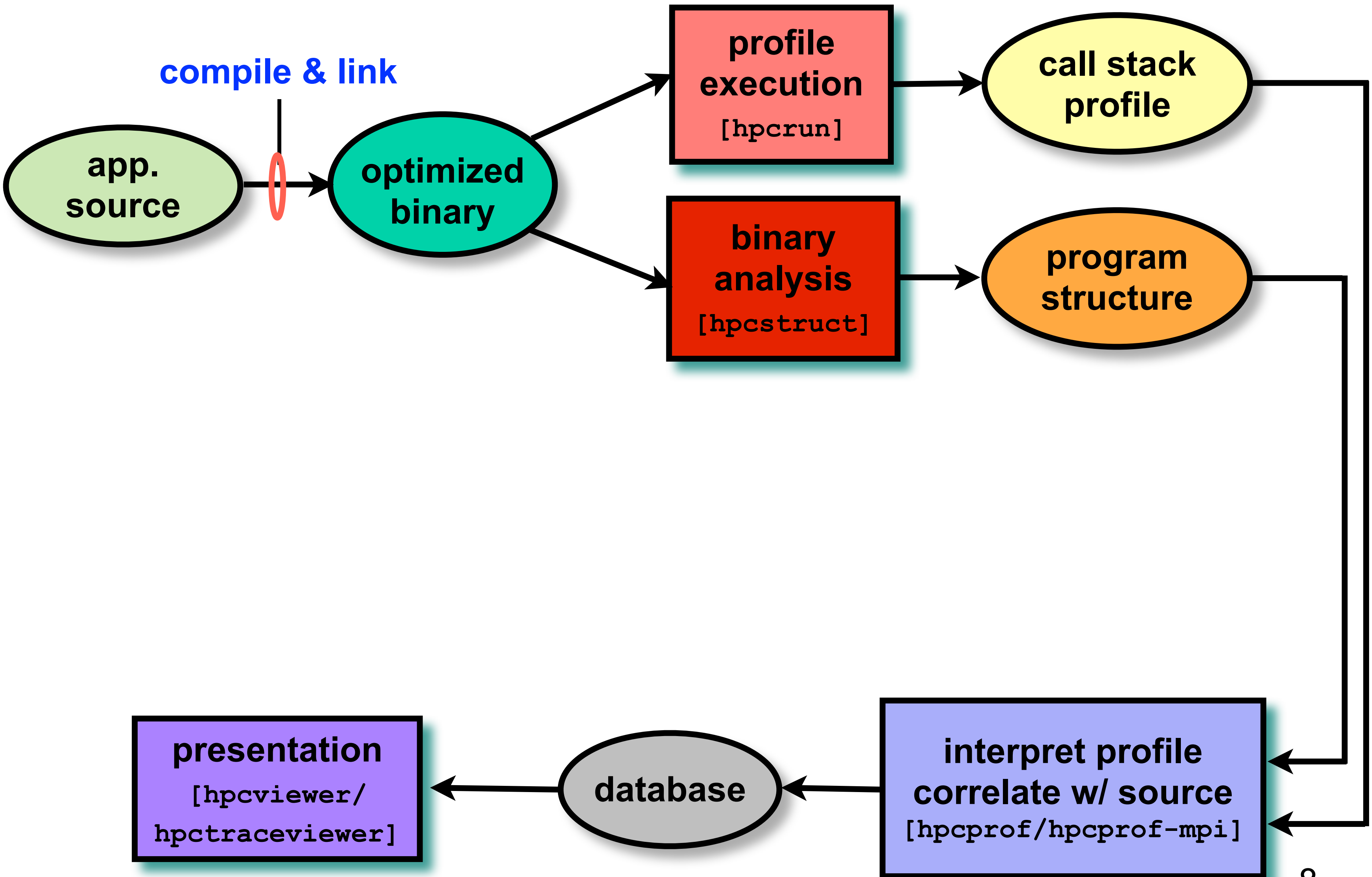
HPCToolkit Design Principles

- **Binary-level measurement and analysis**
  - — observe fully optimized, dynamically linked executions
  - — support multi-lingual codes with external binary-only libraries
- **Sampling-based measurement (avoid instrumentation)**
  - — minimize systematic error and avoid blind spots
  - — enable data collection for large-scale parallelism
- **Collect and correlate multiple derived performance metrics**
  - — diagnosis requires more than one species of metric
  - — derived metrics: “unused bandwidth” rather than “cycles”
- **Associate metrics with both static and dynamic context**
  - — loop nests, procedures, inlined code, calling context
- **Support top-down performance analysis**
  - — intuitive enough for scientists and engineers to use
  - — detailed enough to meet the needs of compiler writers

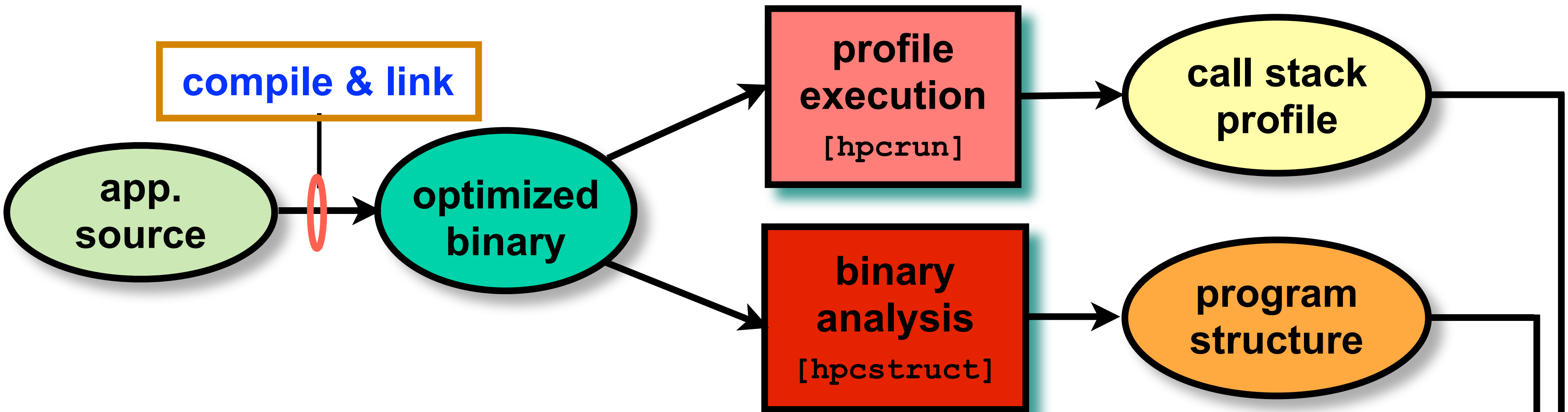
Outline

- **Overview of Rice's HPCToolkit**
- **Accurate measurement**
- **Effective performance analysis**
- **Pinpointing scalability bottlenecks**
  - — **scalability bottlenecks on large-scale parallel systems**
  - — **scaling on multicore processors**
- **Assessing process variability**
- **Understanding temporal behavior**
- **Using HPCToolkit**

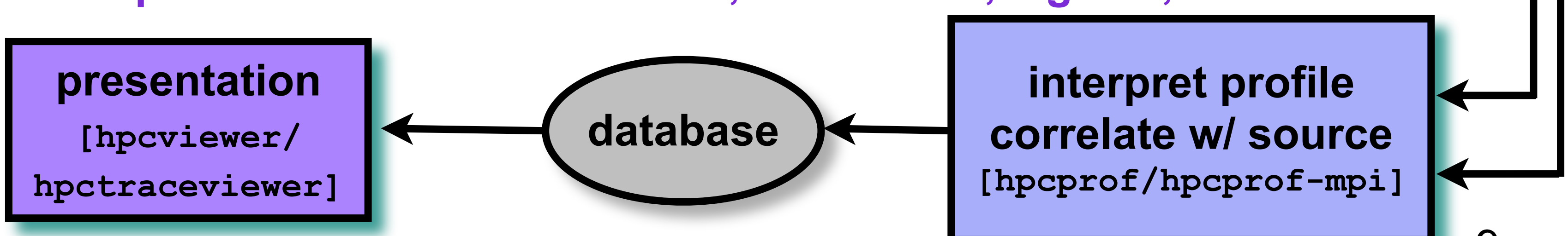
HPCToolkit Workflow



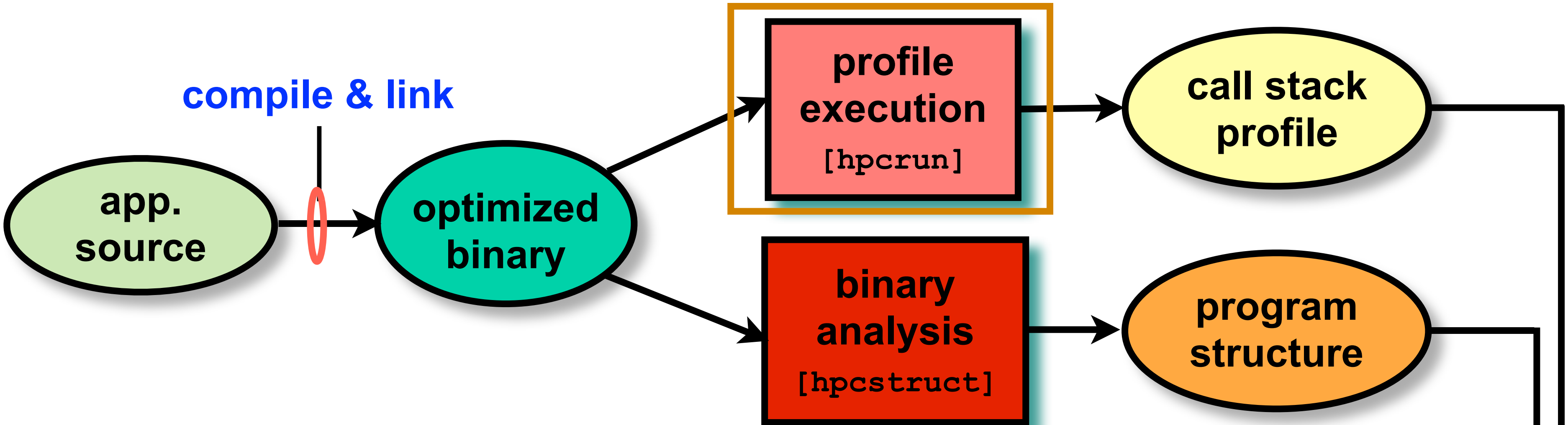
HPCToolkit Workflow



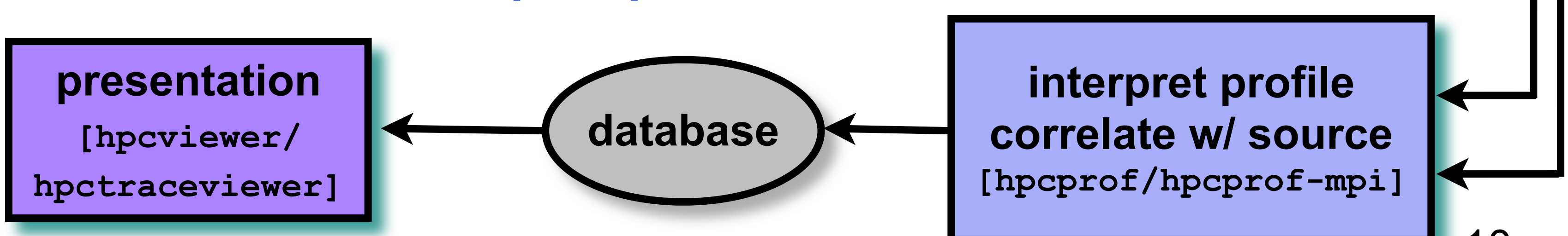
- For dynamically-linked executables on stock Linux
  — compile and link as you usually do: nothing special needed
- For statically-linked executables (e.g. for BG/P, Cray XT)
  — add monitoring by using **`hpclink`** as prefix to your link line
    - uses “linker wrapping” to catch “control” operations
      process and thread creation, finalization, signals, ...



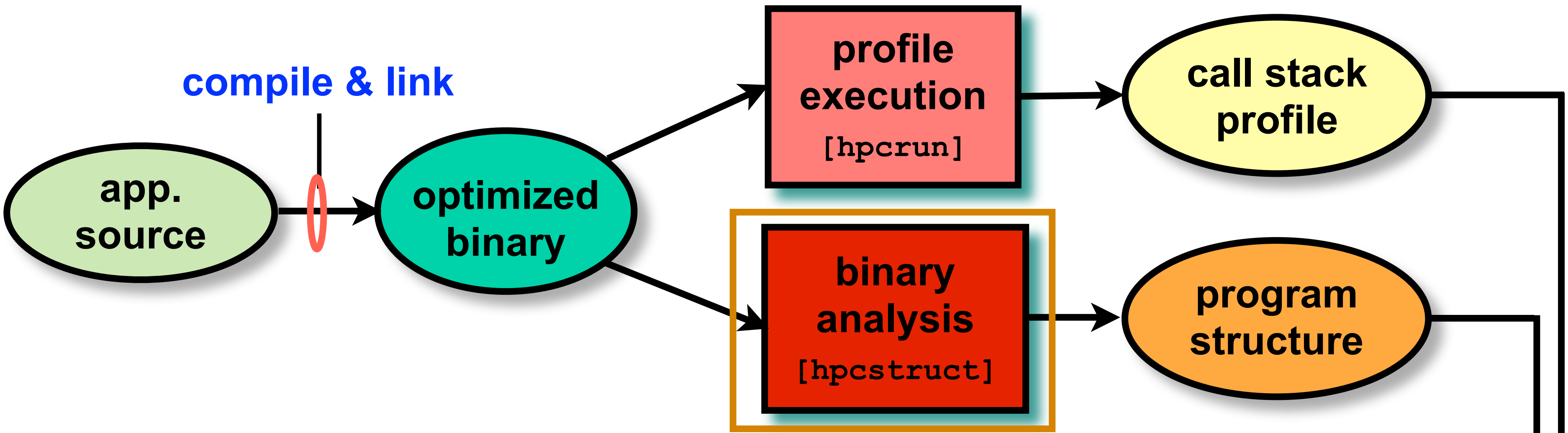
HPCToolkit Workflow



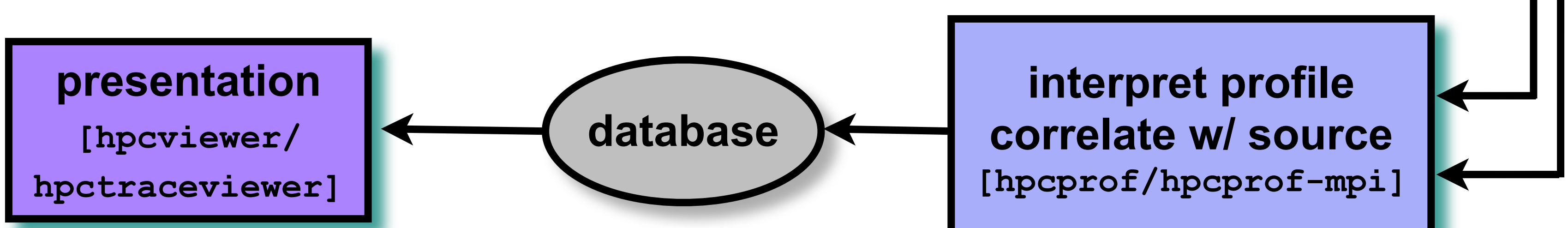
- **Measure execution unobtrusively**
  - launch optimized application binaries
    - dynamically-linked applications: launch with `hpcrun` to measure
    - statically-linked applications: measurement library added at link time
      - control with environment variable settings
  - collect statistical call path profiles of events of interest



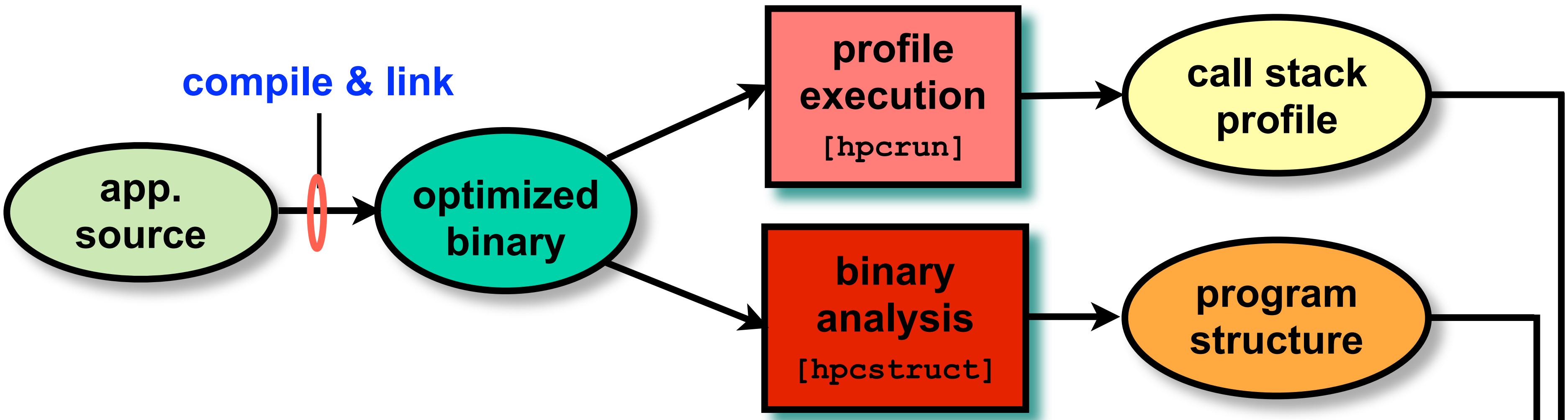
HPCToolkit Workflow



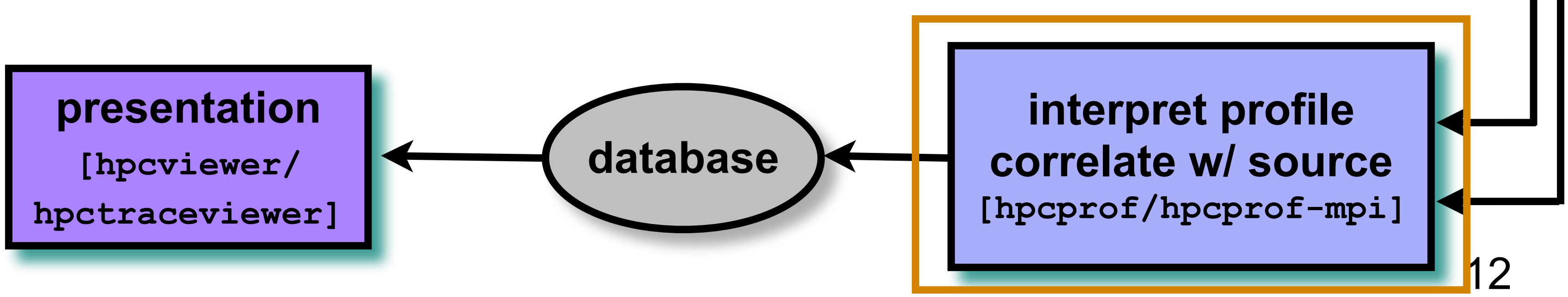
- Analyze binary with **hpcstruct**: recover program structure
  - analyze machine code, line map, debugging information
  - extract loop nesting & identify inlined procedures
  - map transformed loops and procedures to source



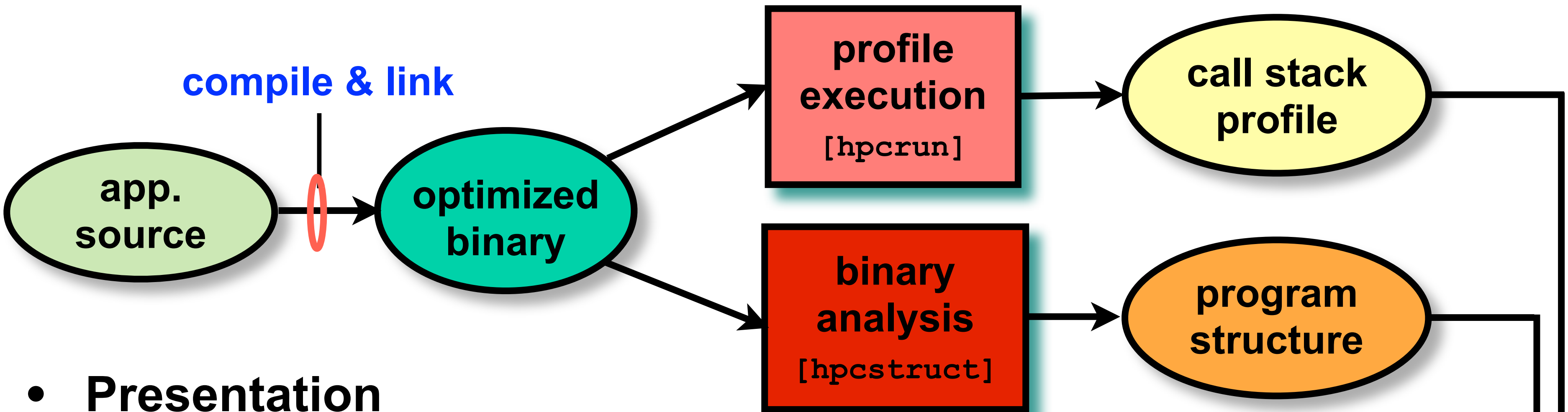
HPCToolkit Workflow



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

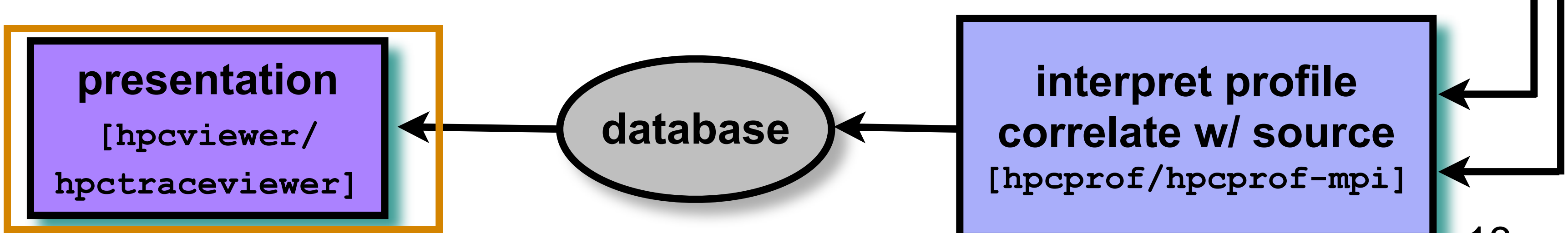


HPCToolkit Workflow



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

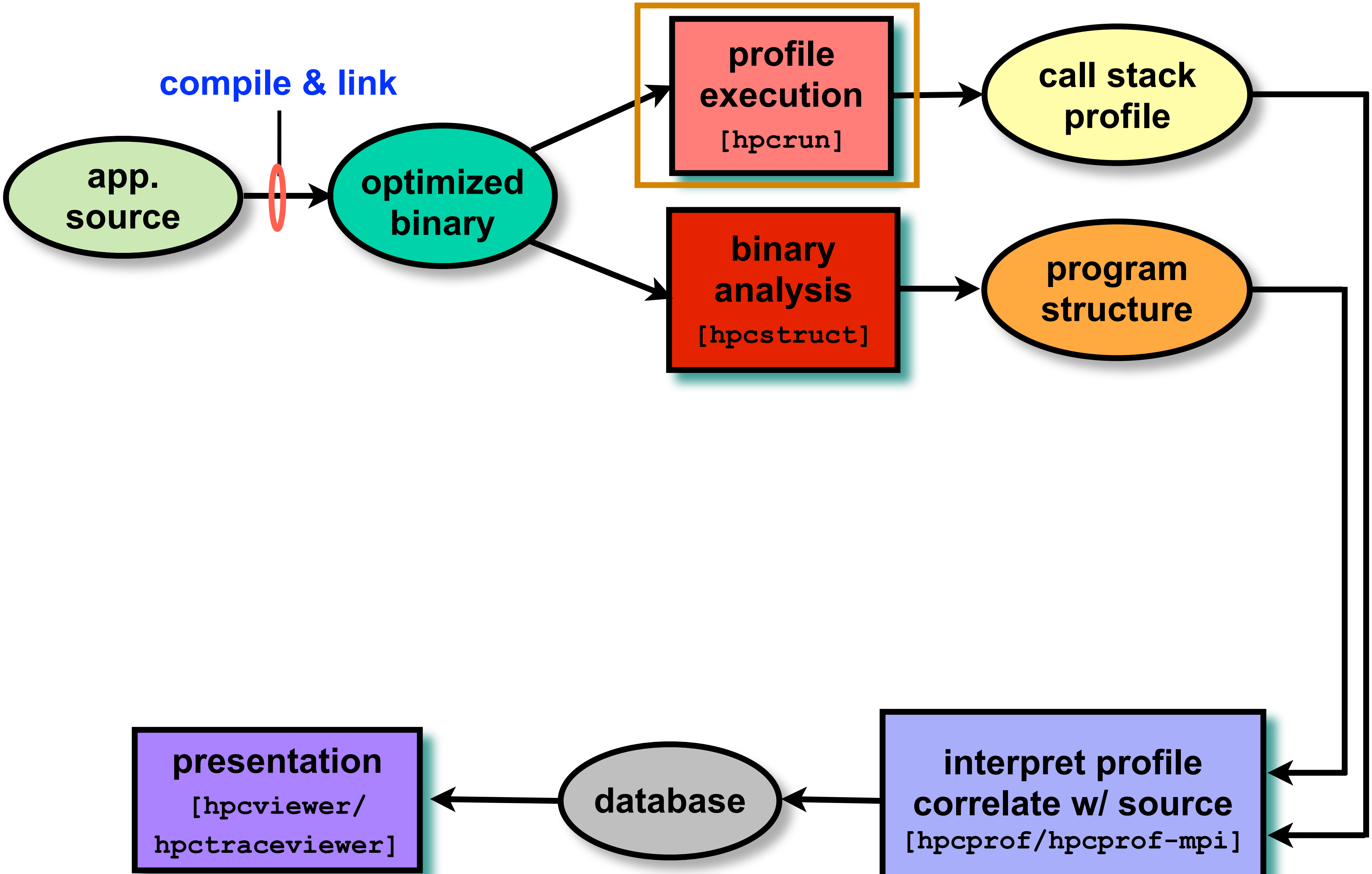
      e.g. scalability losses, waste, CPI, bandwidth
  - — graph thread-level metrics for contexts
  - — explore evolution of behavior over time



Outline

- Overview of Rice's HPCToolkit
- **Accurate measurement**
- Effective performance analysis
- Pinpointing scalability bottlenecks
  - — scalability bottlenecks on large-scale parallel systems
  - — scaling on multicore processors
- Assessing process variability
- Understanding temporal behavior
- Using HPCToolkit

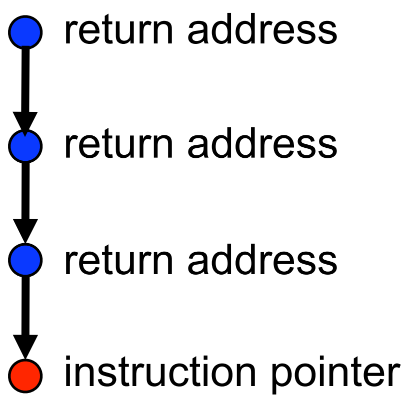
Measurement



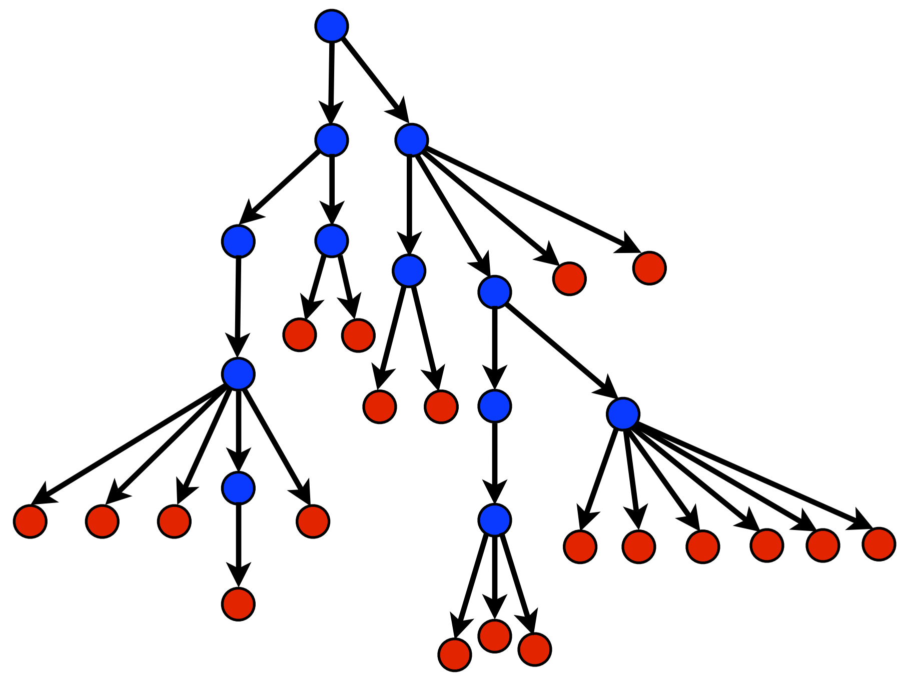
Call Path Profiling

- **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

Novel Aspects of Our Approach

- **Unwind fully-optimized and even stripped code**
  - use on-the-fly binary analysis to support unwinding
- **Cope with dynamically-loaded shared libraries on Linux**
  - note as new code becomes available in address space
- **Integrate static & dynamic context information in presentation**
  - dynamic call chains including procedures, inlined functions, loops, and statements

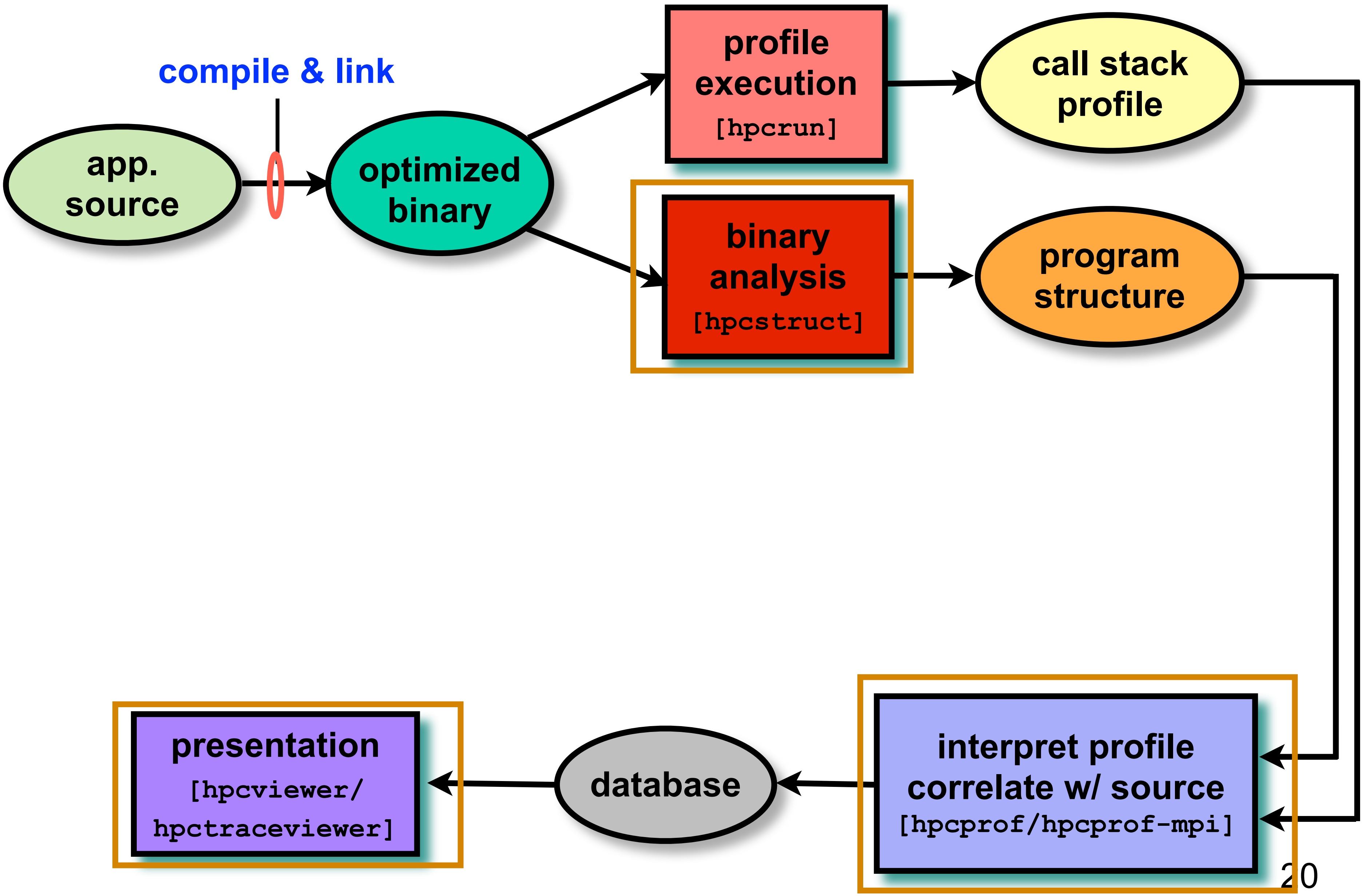
Measurement Effectiveness

- **Accurate**
  - PFLOTRAN on Cray XT @ 8192 cores
    - 148 unwind failures out of 289M unwinds
    - 5e-5% errors
  - Flash on Blue Gene/P @ 8192 cores
    - 212K unwind failures out of 1.1B unwinds
    - 2e-2% errors
  - SPEC2006 benchmark test suite (sequential codes)
    - fully-optimized executables: Intel, PGI, and Pathscale compilers
    - 292 unwind failures out of 18M unwinds (Intel Harpertown)
    - 1e-3% error
- **Low overhead**
  - e.g. PFLOTRAN scaling study on Cray XT @ 512 cores
    - measured cycles, L2 miss, FLOPs, & TLB @ 1.5% overhead
  - suitable for use on production runs

Outline

- Overview of Rice's HPCToolkit
- Accurate measurement
- **Effective performance analysis**
- Pinpointing scalability bottlenecks
  - scalability bottlenecks on large-scale parallel systems
  - scaling on multicore processors
- Assessing process variability
- Understanding temporal behavior
- Using HPCToolkit

Effective Analysis



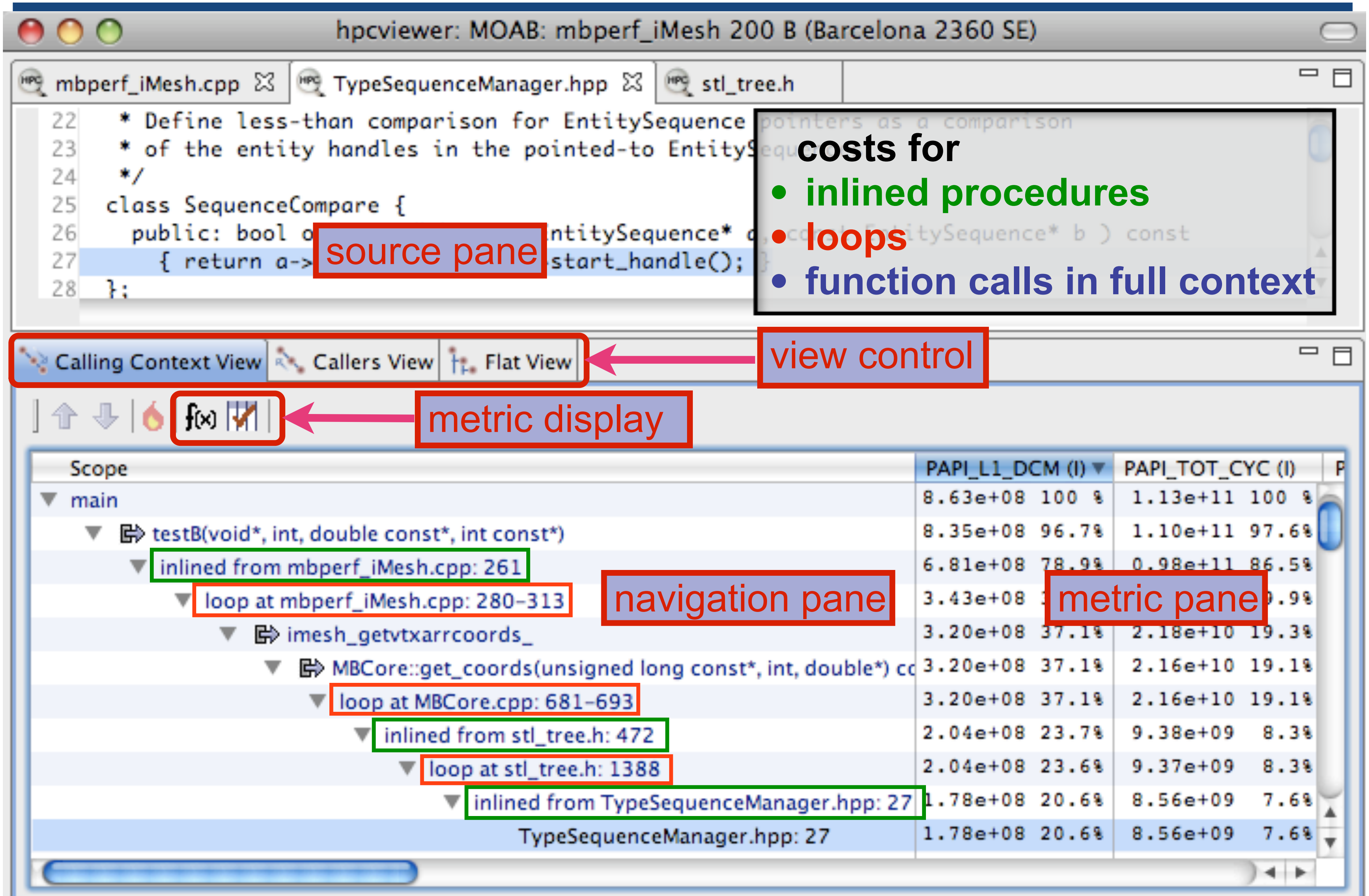
Recovering Program Structure

- Analyze an application binary
  - identify object code procedures and loops
    - decode machine instructions
    - construct control flow graph from branches
    - identify natural loop nests using interval analysis
  - map object code procedures/loops to source code
    - leverage line map + debugging information
    - discover inlined code
    - account for many loop and procedure transformations

Unique benefit of our binary analysis

- Bridges the gap between
  - lightweight measurement of fully optimized binaries
  - desire to correlate low-level metrics to source level abstractions

Analyzing Results with `hpcviewer`



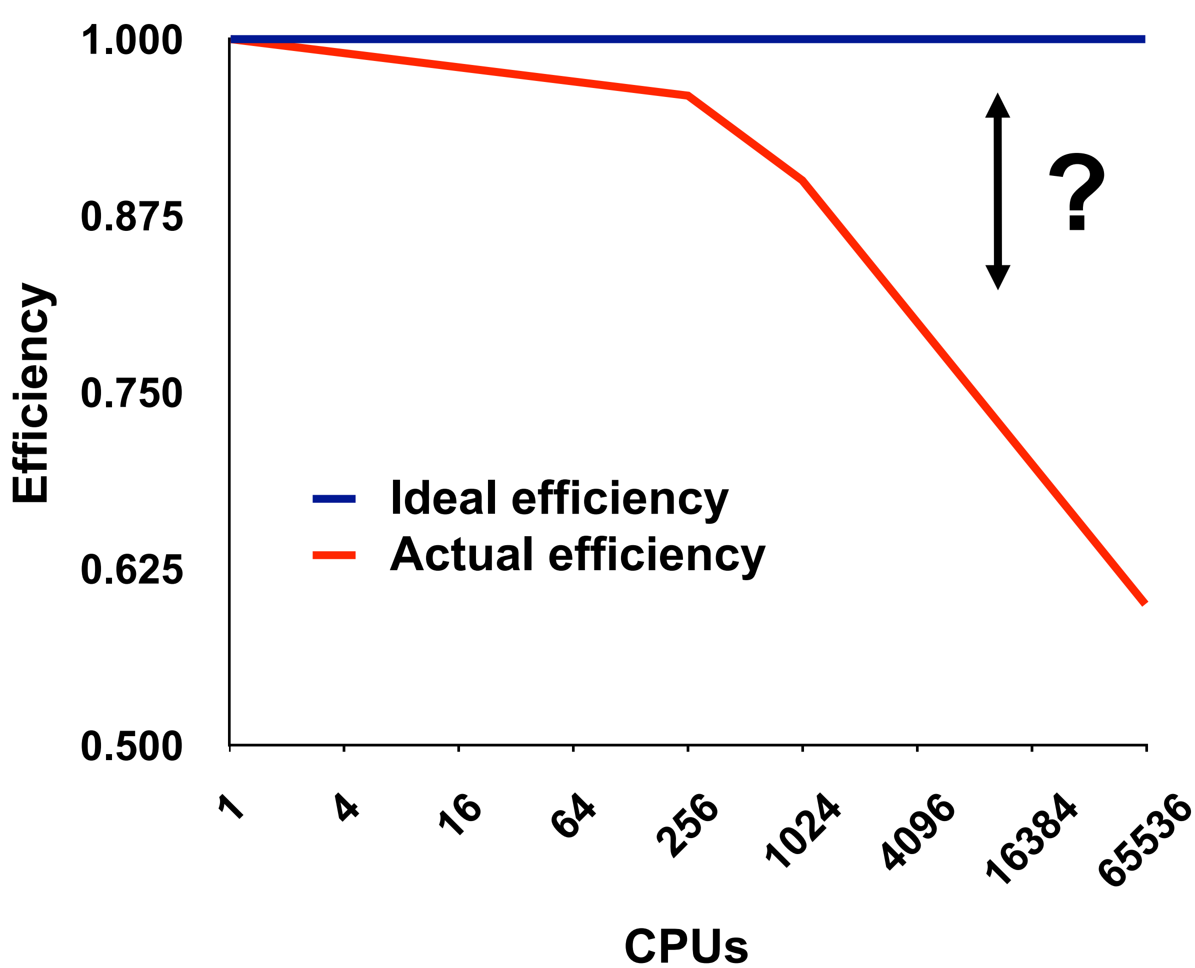
Principal Views

- Calling context tree view - “top-down” (down the call chain)
  - — associate metrics with each dynamic calling context
  - — high-level, hierarchical view of distribution of costs
- Caller’s view - “bottom-up” (up the call chain)
  - — apportion a procedure’s metrics to its dynamic calling contexts
  - — understand costs of a procedure called in many places
- Flat view - ignores the calling context of each sample point
  - — aggregate all metrics for a procedure, from any context
  - — attribute costs to loop nests and lines within a procedure

Outline

- Overview of Rice's HPCToolkit
- Accurate measurement
- Effective performance analysis
- Pinpointing scalability bottlenecks
  - scalability bottlenecks on large-scale parallel systems
  - scaling on multicore processors
- Assessing process variability
- Understanding temporal behavior
- Using HPCToolkit

The Problem of Scaling



Note: higher is better

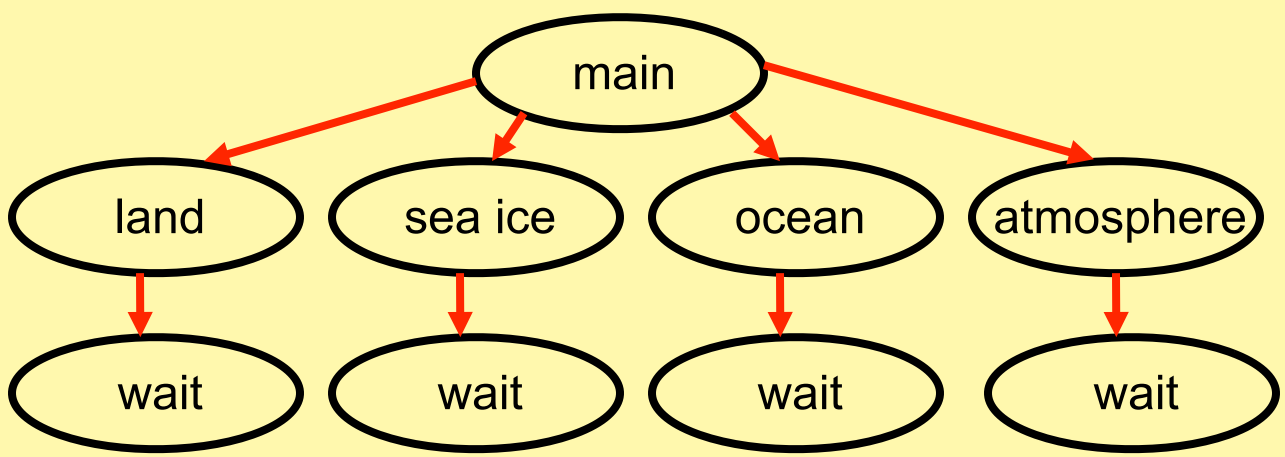
Goal: Automatic Scaling Analysis

- 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

Example climate code skeleton

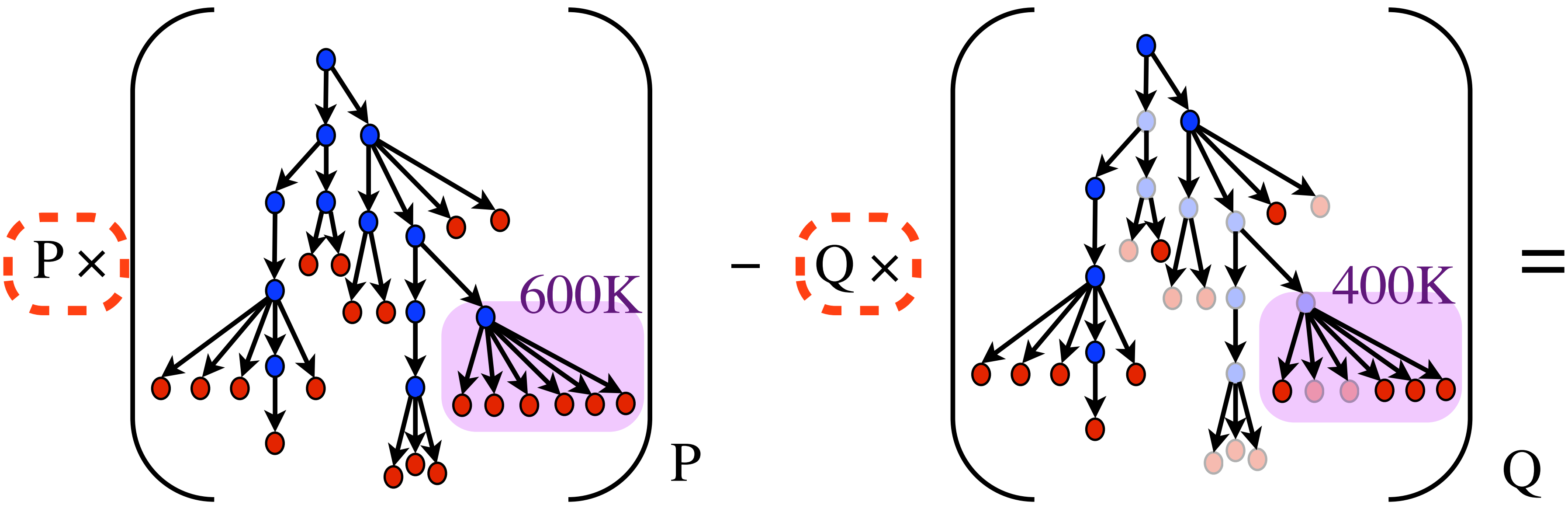


Performance Analysis with Expectations

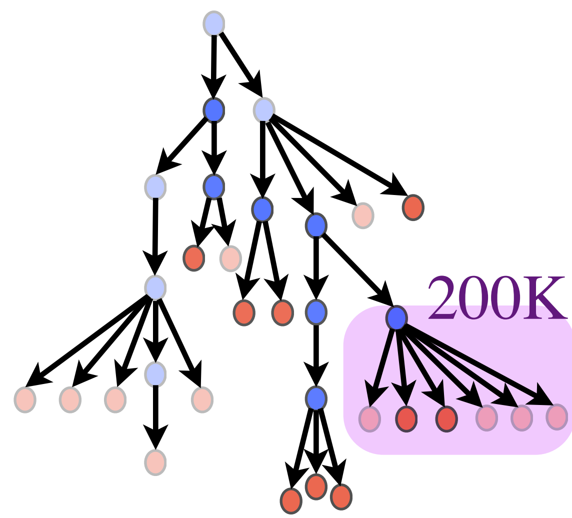
- You have performance expectations for your parallel code
  - strong scaling: linear speedup
  - weak scaling: constant execution time

- Putting 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

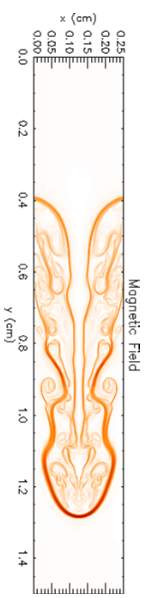


coefficients for analysis
of strong scaling

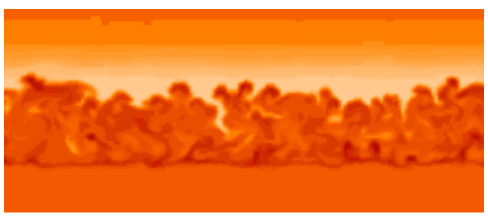


Scalability Analysis Demo

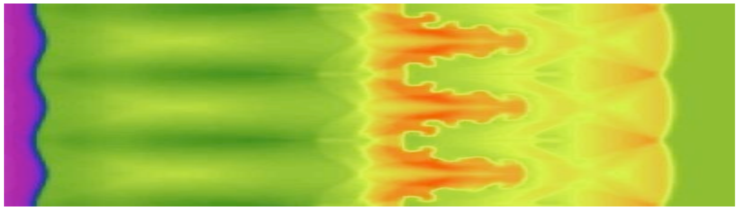
Code:	**University of Chicago FLASH**
Simulation:	**white dwarf detonation**
Platform:	**Blue Gene/P**
Experiment:	**8192 vs. 256 processors**
Scaling type:	**weak**



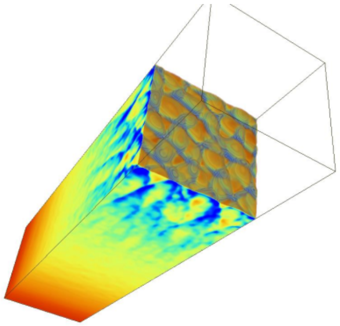
Magnetic Rayleigh-Taylor



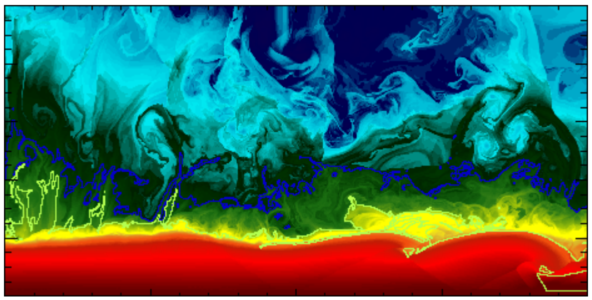
Nova outbursts on white dwarfs



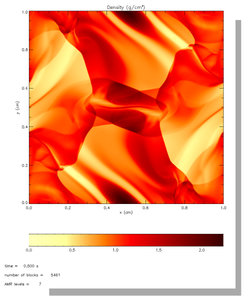
Laser-driven shock instabilities



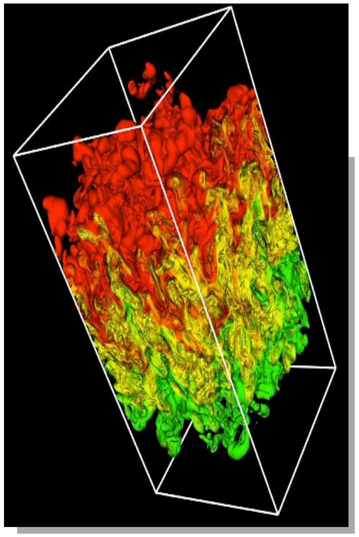
Cellular detonation



Helium burning on neutron stars



Orzag/Tang MHD vortex



Rayleigh-Taylor instability

Figures courtesy of FLASH Team, University of Chicago

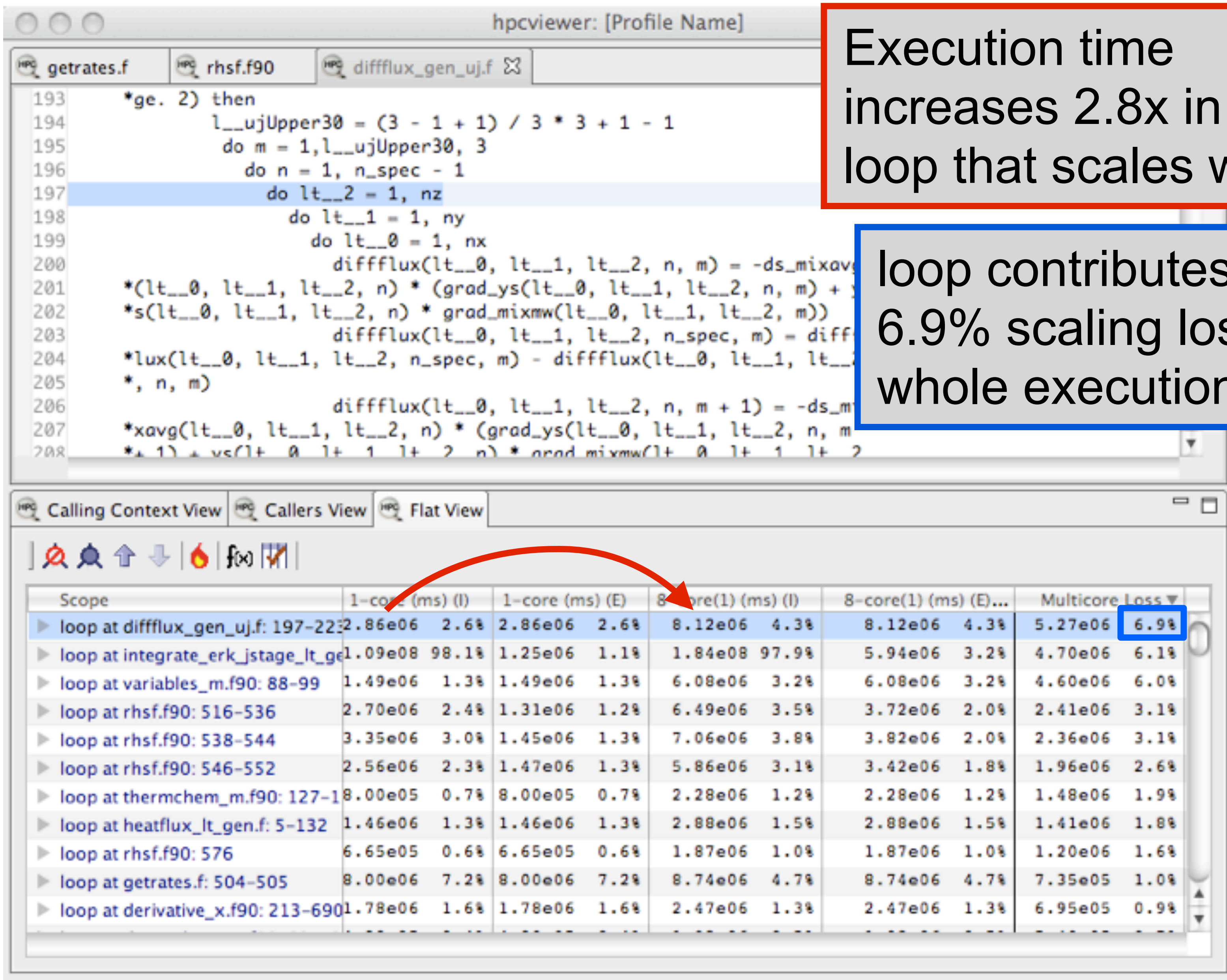
Scaling on Multicore Processors

- **Compare performance**
  - **single vs. multiple processes on a multicore system**
- **Strategy**
  - **differential performance analysis**
    - **subtract the calling context trees as before, unit coefficient for each**

S3D: Multicore Losses at the Loop Level

Execution time increases 2.8x in the loop that scales worst

loop contributes a 6.9% scaling loss to whole execution



Scope	1-core (ms) (I)		1-core (ms) (E)		8-core(1) (ms) (I)		8-core(1) (ms) (E)...		Multicore Loss	
loop at diffflux_gen_uj.f: 197-22	2.86e06	2.6%	2.86e06	2.6%	8.12e06	4.3%	8.12e06	4.3%	5.27e06	6.9%
loop at integrate_erk_jstage_lt_ge	1.09e08	98.1%	1.25e06	1.1%	1.84e08	97.9%	5.94e06	3.2%	4.70e06	6.1%
loop at variables_m.f90: 88-99	1.49e06	1.3%	1.49e06	1.3%	6.08e06	3.2%	6.08e06	3.2%	4.60e06	6.0%
loop at rhsf.f90: 516-536	2.70e06	2.4%	1.31e06	1.2%	6.49e06	3.5%	3.72e06	2.0%	2.41e06	3.1%
loop at rhsf.f90: 538-544	3.35e06	3.0%	1.45e06	1.3%	7.06e06	3.8%	3.82e06	2.0%	2.36e06	3.1%
loop at rhsf.f90: 546-552	2.56e06	2.3%	1.47e06	1.3%	5.86e06	3.1%	3.42e06	1.8%	1.96e06	2.6%
loop at thermchem_m.f90: 127-1	8.00e05	0.7%	8.00e05	0.7%	2.28e06	1.2%	2.28e06	1.2%	1.48e06	1.9%
loop at heatflux_lt_gen.f: 5-132	1.46e06	1.3%	1.46e06	1.3%	2.88e06	1.5%	2.88e06	1.5%	1.41e06	1.8%
loop at rhsf.f90: 576	6.65e05	0.6%	6.65e05	0.6%	1.87e06	1.0%	1.87e06	1.0%	1.20e06	1.6%
loop at getrates.f: 504-505	8.00e06	7.2%	8.00e06	7.2%	8.74e06	4.7%	8.74e06	4.7%	7.35e05	1.0%
loop at derivative_x.f90: 213-690	1.78e06	1.6%	1.78e06	1.6%	2.47e06	1.3%	2.47e06	1.3%	6.95e05	0.9%

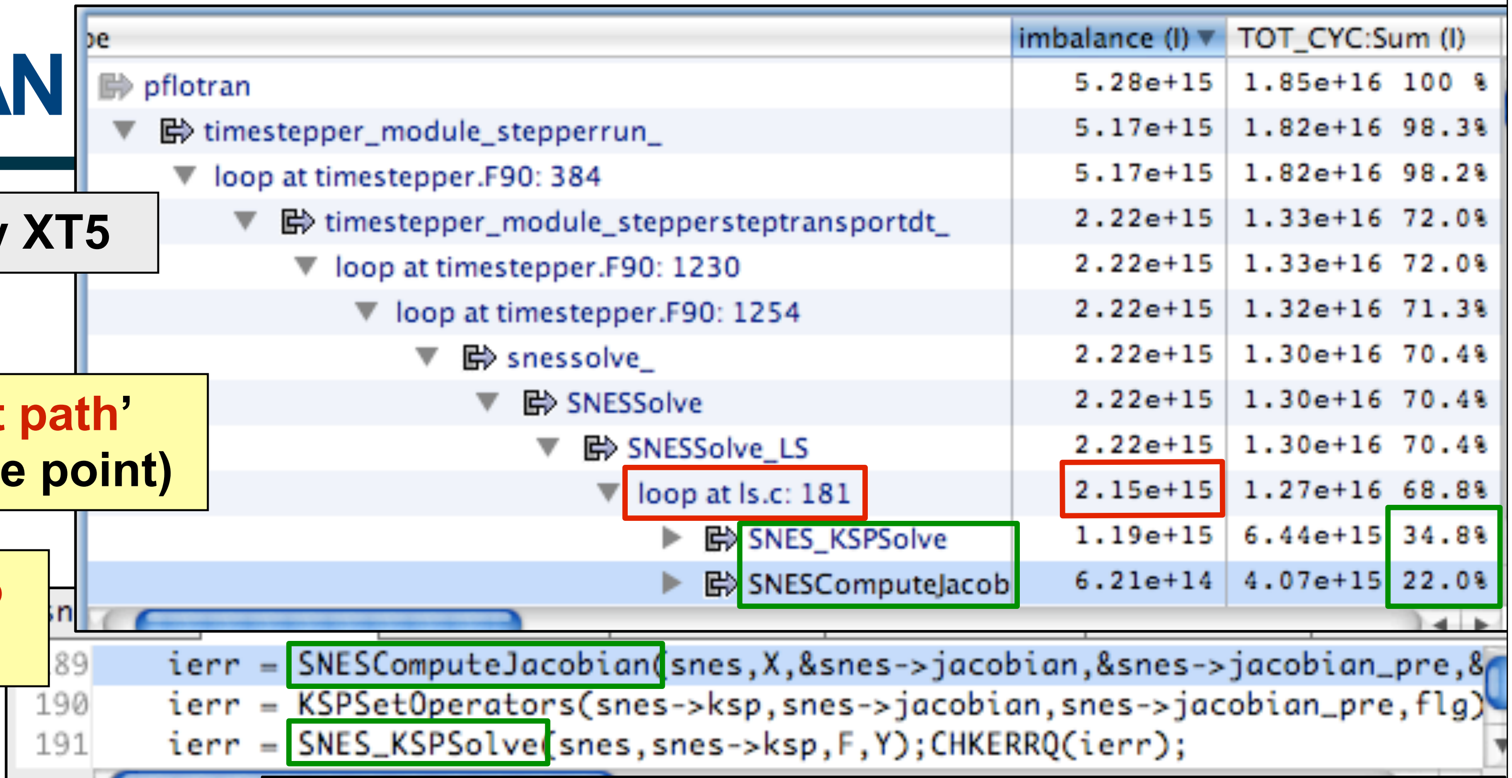
Outline

- Overview of Rice's HPCToolkit
- Accurate measurement
- Effective performance analysis
- Pinpointing scalability bottlenecks
  - — scalability bottlenecks on large-scale parallel systems
  - — scaling on multicore processors
- Assessing process variability
- Understanding temporal behavior
- Using HPCToolkit

PFLOTRAN

8K cores, Cray XT5

1. Drill down 'hot path' to loop (a balance point)

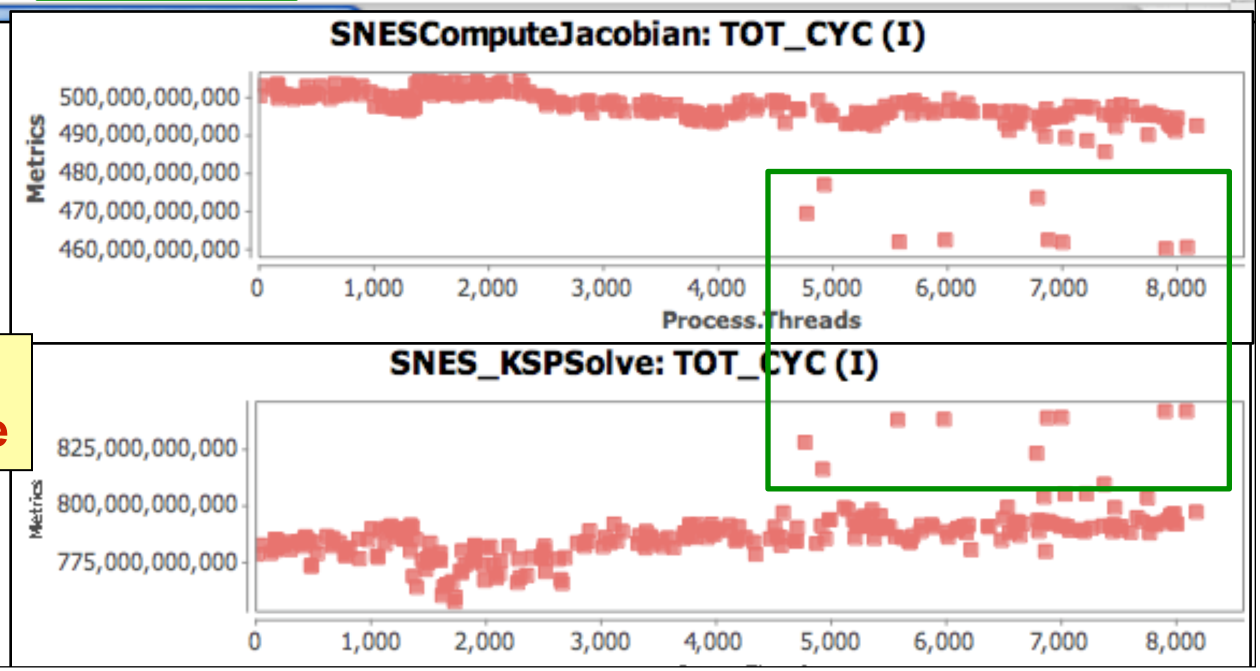


2. Notice top two call sites...

3. Plot the per-process values:

Early finishers...

... become early arrivers at Allreduce



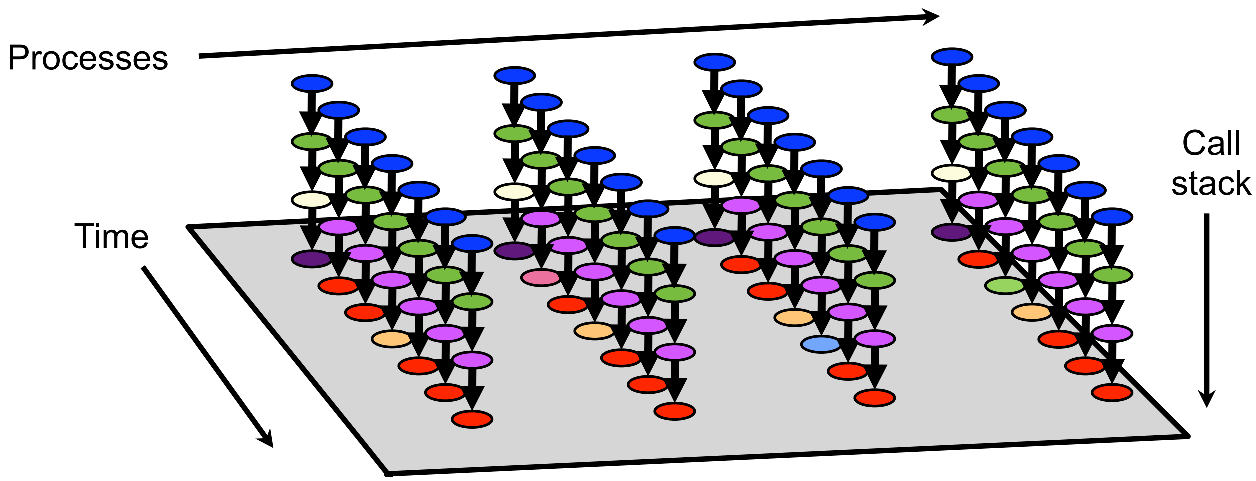
Outline

- Overview of Rice’s HPCToolkit
- Accurate measurement
- Effective performance analysis
- Pinpointing scalability bottlenecks
  - scalability bottlenecks on large-scale parallel systems
  - scaling on multicore processors
- Assessing process variability
- **Understanding temporal behavior**
- Using HPCToolkit

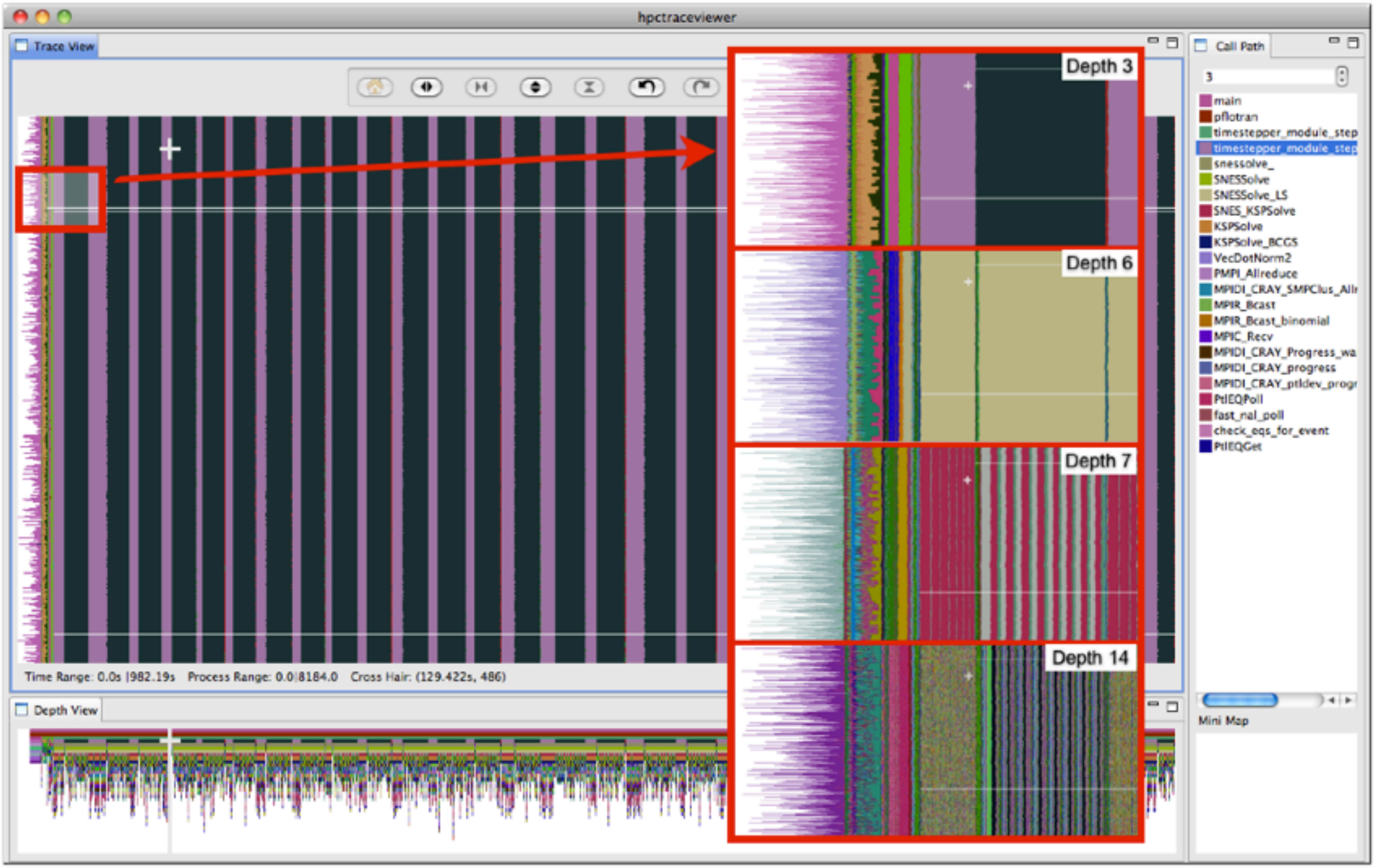
Understanding Temporal Behavior

- **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



Process-Time Views of PFLOTRAN



8184-core execution on Cray XT5. Trace view rendered using hpctraceviewer on a Mac Book Pro Laptop. Insets show zoomed view of marked region at different call stack depths.

Outline

- Overview of Rice's HPCToolkit
- Accurate measurement
- Effective performance analysis
- Pinpointing scalability bottlenecks
  - scalability bottlenecks on large-scale parallel systems
  - scaling on multicore processors
- Assessing process variability
- Understanding temporal behavior
- Using HPCToolkit

Where to Find HPCToolkit

- DOE Systems
  - jaguar: /ccs/proj/hpctoolkit/pkgs/hpctoolkit
  - intrepid: /home/projects/hpctoolkit/pkgs/hpctoolkit
  - franklin: /project/projectdirs/hpctk/pkgs/hpctoolkit-franklin
  - hopper: /project/projectdirs/hpctk/pkgs/hpctoolkit-hopper
- See examples subdirectory for chombo x 1024 data
- For your local Linux systems, you can download and install it
  - documentation, build instructions, and software
    - see http://hpctoolkit.org for instructions
  - we recommend downloading and building from svn
  - important notes:
    - using hardware counters requires downloading and installing PAPI
    - kernel support for hardware counters

      on Linux 2.6.32 or better: built-in kernel support for counters

      requires PAPI newer than 4.1.1 (CVS version at present)

      earlier Linux needs a kernel patch (perfmon2 or perfctr)

Using HPCToolkit at ORNL, NERSC, ANL

- jaguarpf, franklin, hopper, freedom
  - module load java
  - module load hpctoolkit
- intrepid, surveyor
  - add the following to your .softenvrc before @default
    - +ibmjava6
    - +hpctoolkit
  - resoft

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/P and Cray XT**

  — **The hpcviewer user interface**

  — **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
  - see earlier slide for HPCToolkit's HOME directory on your system
- Adjust your compiler flags (if you want <u>full</u> 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 ...`
- Decide what hardware counters to monitor
  - statically-linked executables (e.g., Cray XT, BG/P)
    - use hpclink to link your executable
    - launch executable with environment var HPCRUN_EVENT_LIST=LIST

      (BG/P hardware counters supported)
  - dynamically-linked executables (e.g., Linux)
    - use hpcrun -L to learn about counters available for profiling
    - use papi_avail

      you can sample any event listed as "profilable"

HPCToolkit Examples on Intrepid

- **Example script for monitoring an application using hpctoolkit**
  — /home/projects/hpctoolkit/pkgs/hpctoolkit/share/examples/bgp-scripts/run-bgp.sh
- **Example script for launching hpcprof-mpi**
  — /home/projects/hpctoolkit/pkgs/hpctoolkit/share/examples/bgp-scripts/run-hpcprof-bgp.sh
- **Example performance data**
  — /home/projects/hpctoolkit/pkgs/hpctoolkit/share/examples/data/hpctoolkit-fft-crayxt-256

Launching your Job

- **Modify your run script to enable monitoring**
  - **Cray XT: set environment variable in your PBS script**
    - **e.g. setenv HPCRUN_EVENT_LIST "PAPI_TOT_CYC@3000000 PAPI_L2_MISS@400000 PAPI_TLB_MISS@400000 PAPI_FP_OPS@400000"**
  - **Blue Gene/P: pass environment settings to cqsub**
    - **cqsub -p YourAllocation -q prod-devel -t 30 -n 2048 -c 8192 \ --mode vn --env HPCRUN_EVENT_LIST=WALLCLOCK@1000 \ flash3.hpc**

Analysis and Visualization

- Use hpcstruct to reconstruct program structure
  - e.g. `hpcstruct myapp`
    - creates myapp.hpcstruct
- Use hpcsummary script to summarize measurement data
  - e.g. `hpcsummary hpctoolkit-myapp-measurements-5912`
- Use hpcprof to correlate measurements to source code
  - run hpcprof on the front-end node
  - run hpcprof-mpi on the back-end nodes to analyze data in parallel
- Use hpcviewer to open resulting database
- Use hpctraceviewer to explore traces (collected with -t option)

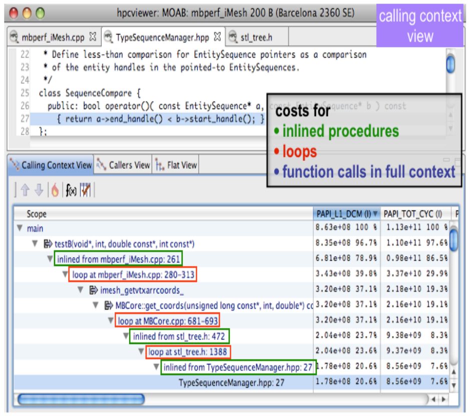
A Special Note About `hpcstruct` and `xlf`

- IBM's xlf compiler emits machine code for Fortran that have an unusual mapping back to source
- To compensate, hpcstruct needs a special option
  - --loop-fwd-subst=no
  - without this option, many nested loops will be missing in hpcstruct's output and (as a result) hpcviewer

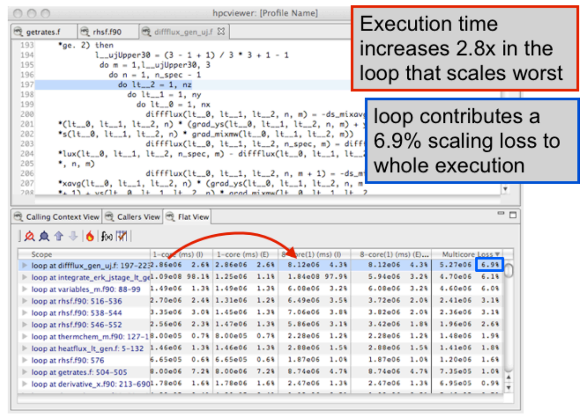
Other Useful Features

- **Leak detection**
  - hpclink --memleak `-o myapp foo.o ... lib.a -lm ...`
  - `when you run`
  - `setenv HPCTOOLKIT_EVENT_LIST=MEMLEAK`

HPCToolkit Capabilities at a Glance

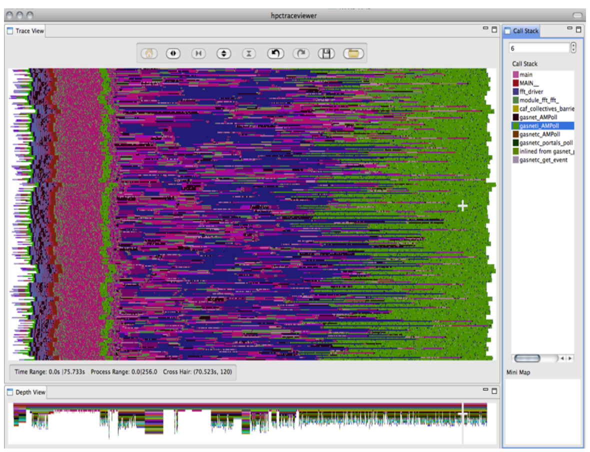


Attribute Costs to Code

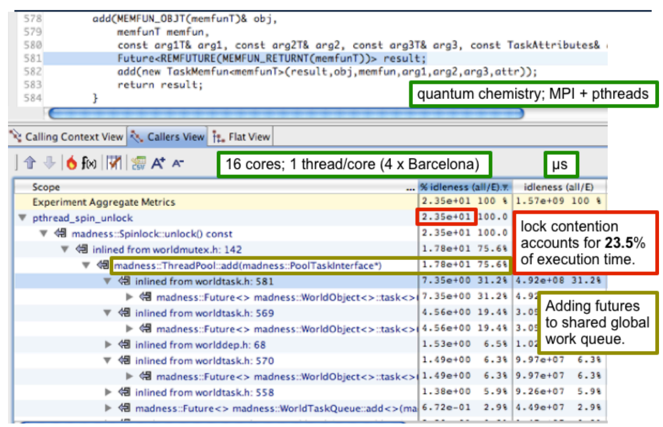


Pinpoint & Quantify Scaling Bottlenecks

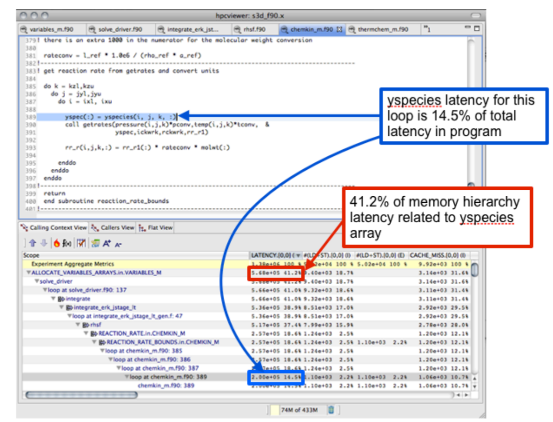
Assess Imbalance and Variability



Analyze Behavior over Time



Shift Blame from Symptoms to Causes



Associate Costs with Data

RICE

hpctoolkit.org