HPCToolkit: Sampling-based Performance Tools for Leadership Computing

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

`http://hpctoolkit.org`

Acknowledgments

- Staff
  - Laksono Adhianto, Mike Fagan, Mark Krentel, Nathan Tallent
- Grad Student
  - Xu Liu
- High School Summer Interns
  - Sinchan Banerjee, Michael Franco, Bowden Kelly, Chas Jhin, Reed Landrum, Gaurav Sanghani
- Alumni
  - Gabriel Marin (ORNL), Robert Fowler (RENCI)
- 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 pose challenges
  - for measurement and analysis
  - for understanding and tuning
- Leadership computing platforms: additional complexity
  - more than just computation: communication, I/O
  - immense scale
  - unique microkernel-based operating systems

Performance Analysis Principles

- **Without accurate measurement, analysis is irrelevant**
  - — avoid systematic measurement error
    - – instrumentation-based measurement is often problematic
  - — measure actual system, not a mock up
    - – fully optimized production code on the platform of interest
- **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 node parallelism + multithreading

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 leadership computing systems

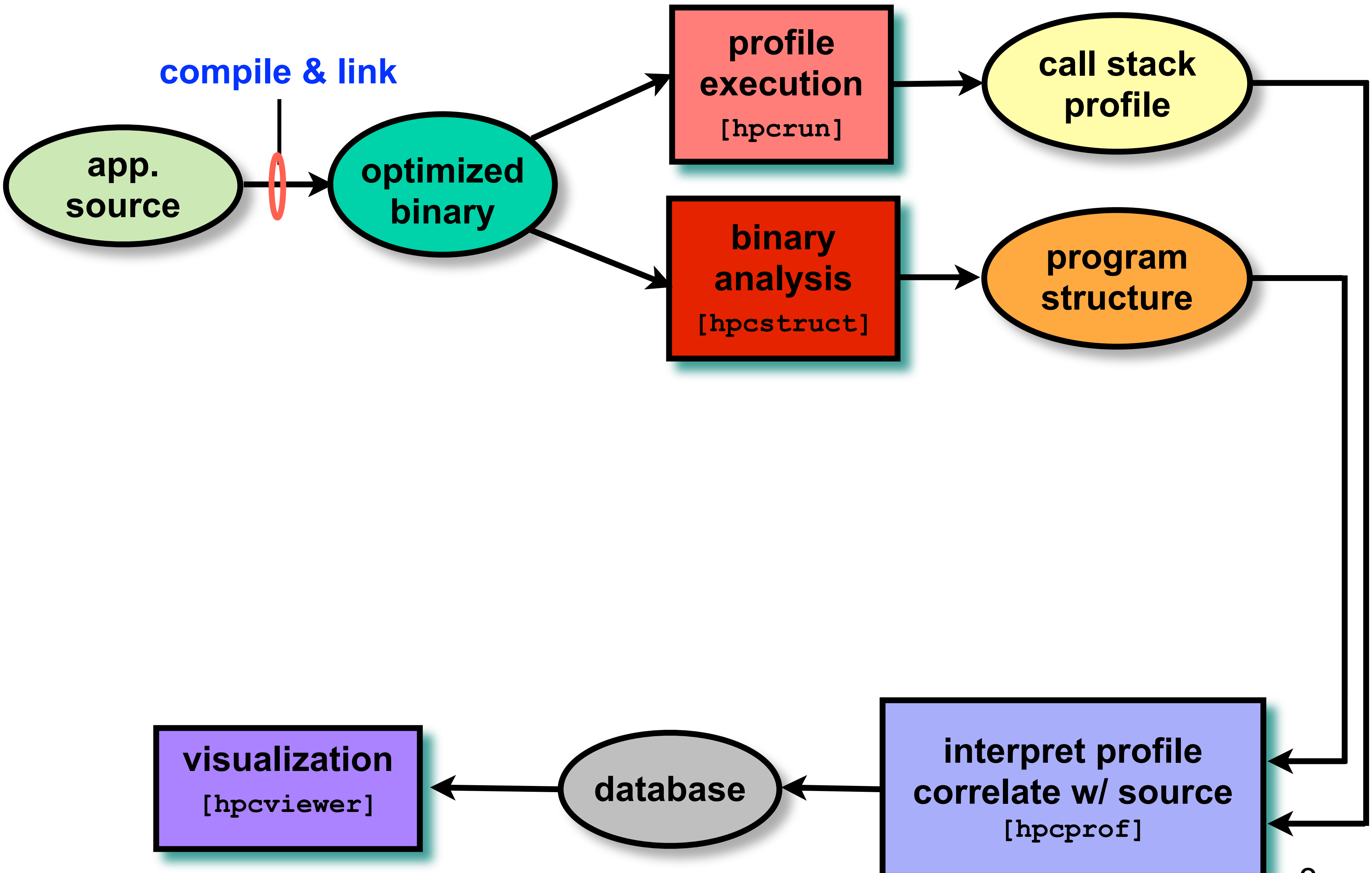
HPCToolkit Design Principles

- **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

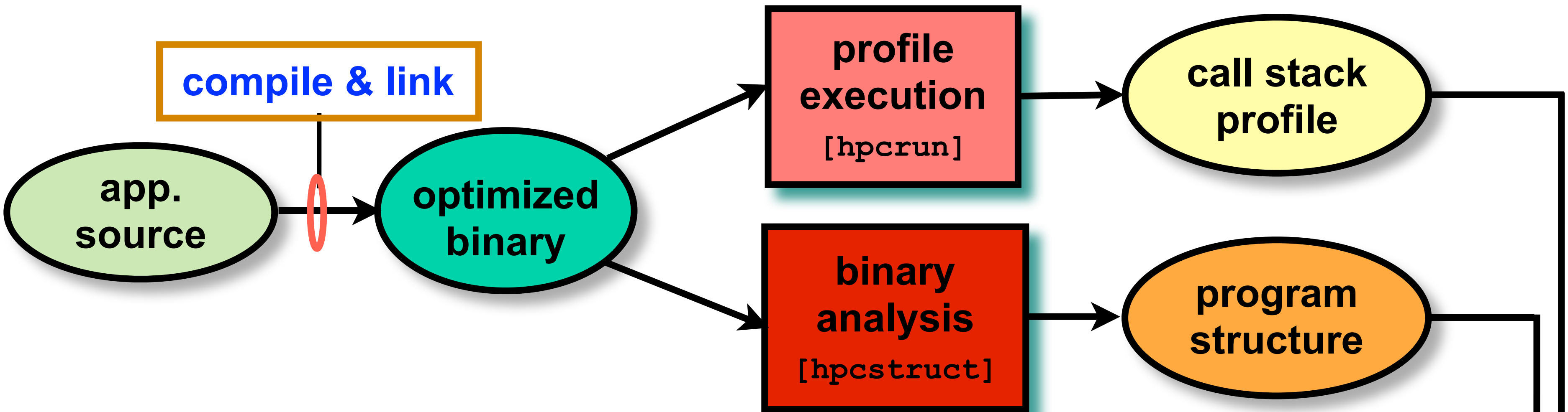
Outline

- **Overview of Rice's HPCToolkit**
- **Accurate measurement**
- **Useful source-level feedback**
- **Effective performance analysis**
  - — **derived metrics for understanding performance**
  - — **pinpointing scalability bottlenecks [SC09]**
  - — **analyzing lock contention in threaded codes [PPoPP10]**
  - — **pinpointing load imbalance [SC10]**
  - — **understanding temporal dynamics of parallel codes**
- **Using HPCToolkit**
- **Coming attractions**

HPCToolkit Workflow

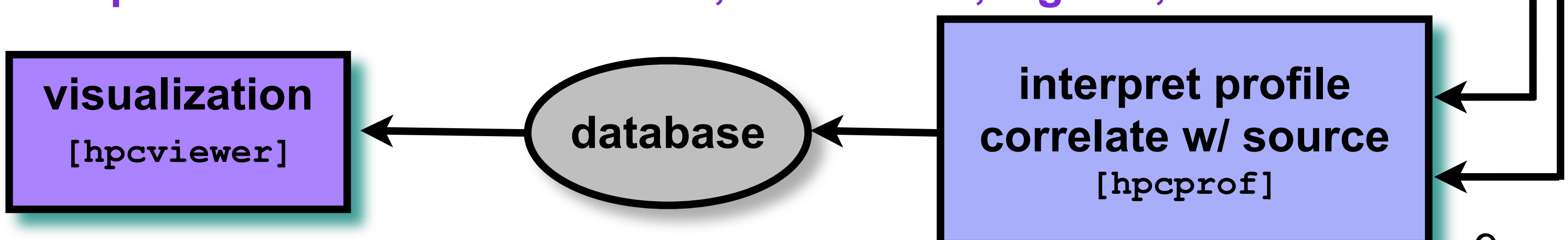


HPCToolkit Workflow

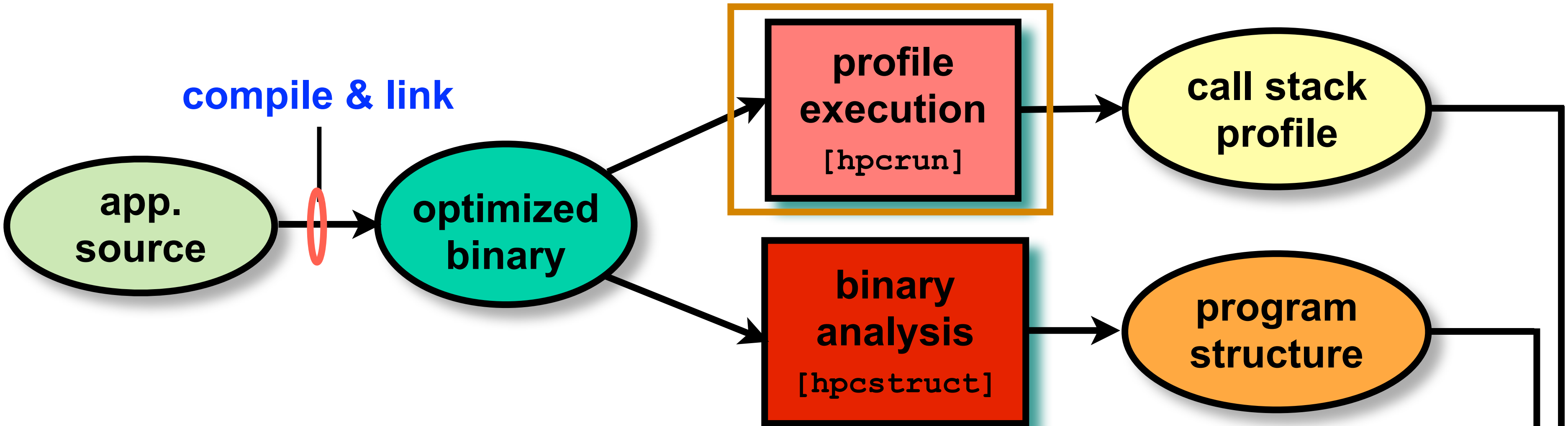


- For dynamically-linked executables on stock Linux
  — compile and link as you usually do: nothing special needed
- For statically-linked executables (e.g. for BG/P, Cray XT)
  — add monitoring by using **`hpclink`** as prefix to your link line
    – uses "linker wrapping" to catch "control" operations

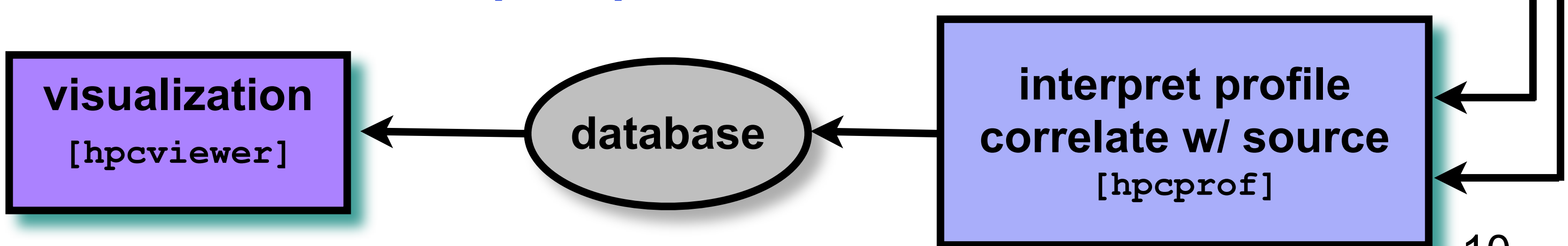
      process and thread creation, finalization, signals, ...



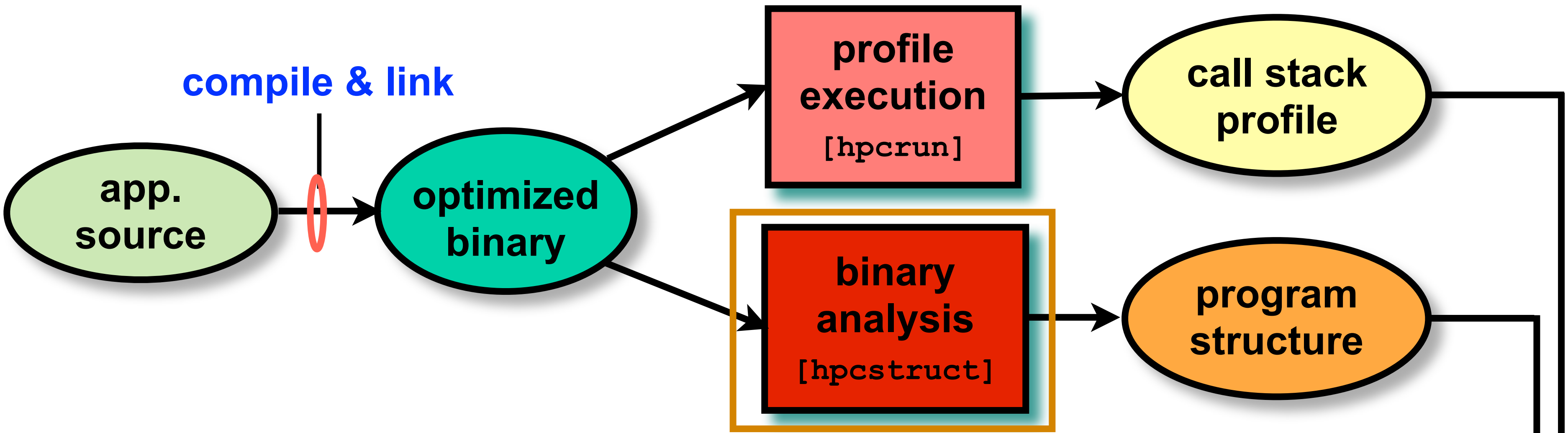
HPCToolkit Workflow



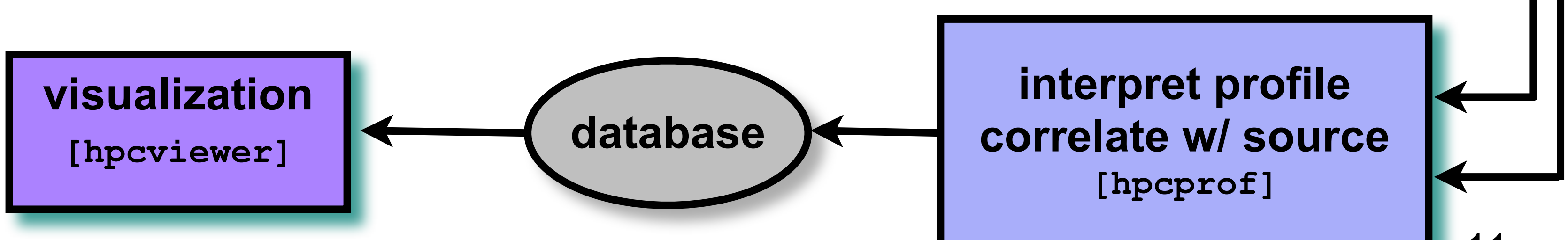
- **Measure execution unobtrusively**
  - launch optimized application binaries
    - dynamically-linked applications: launch with `hpcrun` to measure
    - statically-linked applications: measurement library added at link time
      - control with environment variable settings
  - collect statistical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



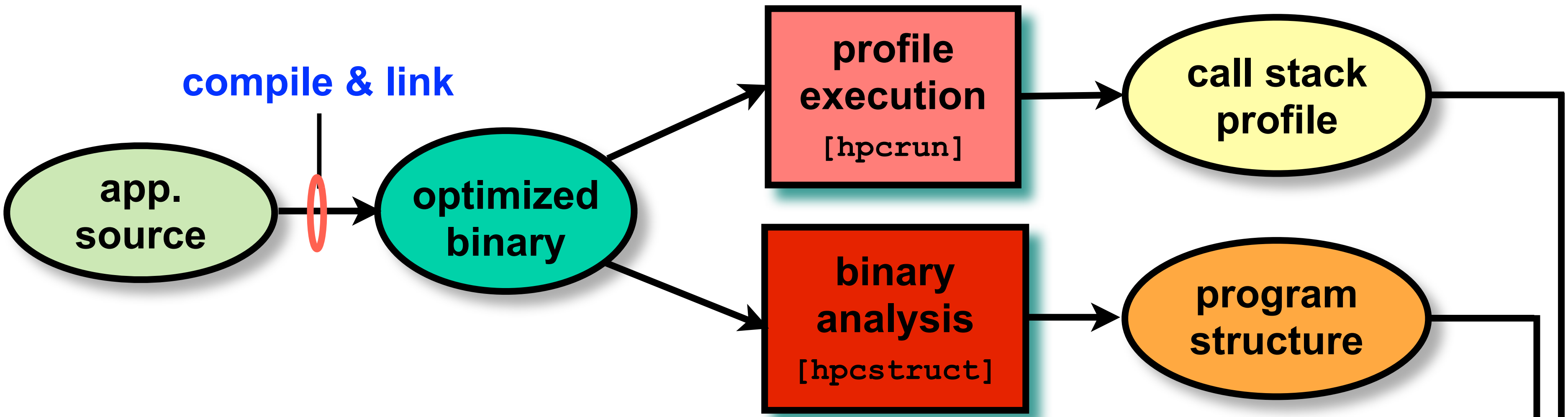
HPCToolkit Workflow



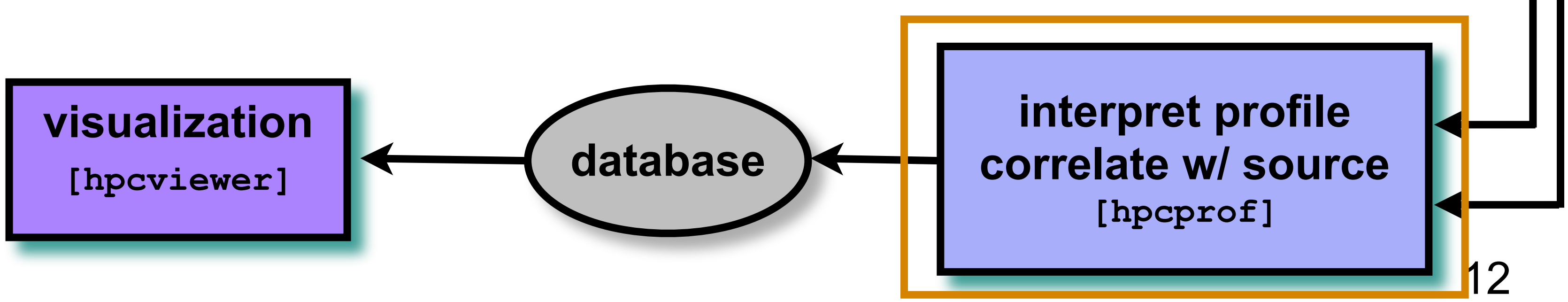
- Analyze binary with `hpcstruct`: recover program structure
  - analyze machine code, line map, debugging information
  - extract loop nesting & identify inlined procedures
  - map transformed loops and procedures to source



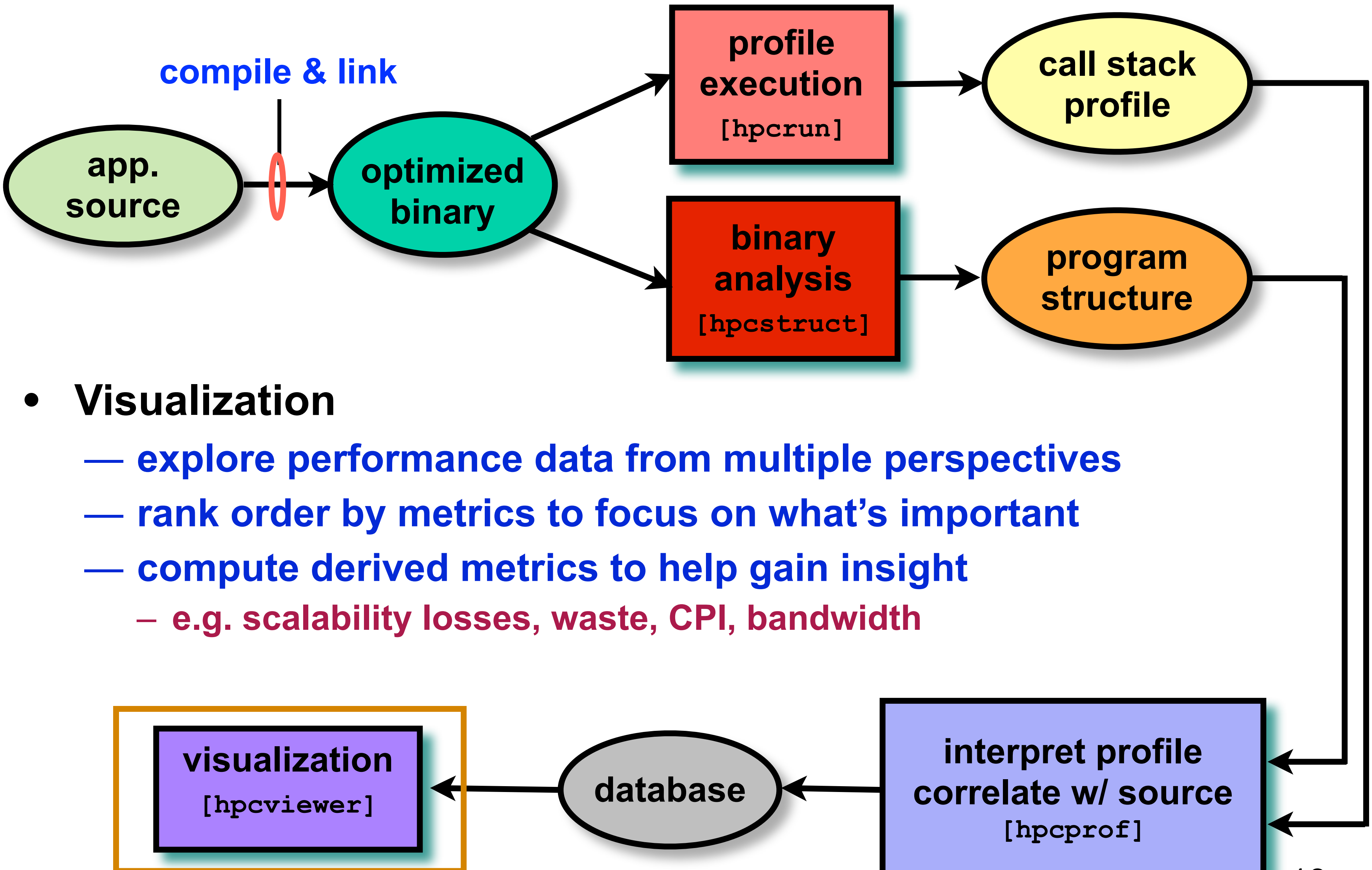
HPCToolkit Workflow



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



HPCToolkit Workflow

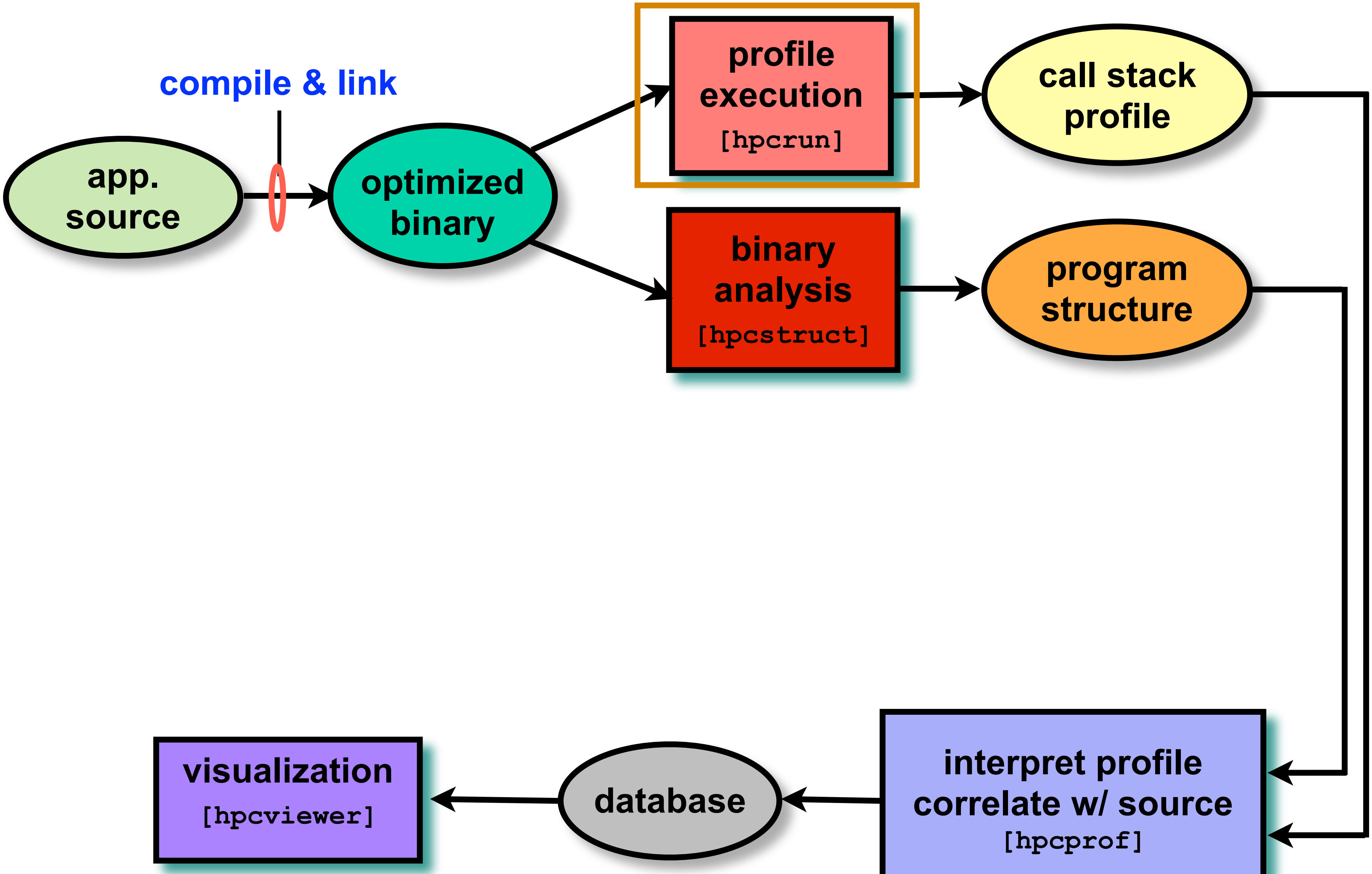


- **Visualization**
  — explore performance data from multiple perspectives
  — rank order by metrics to focus on what's important
  — compute derived metrics to help gain insight
    – e.g. scalability losses, waste, CPI, bandwidth

Outline

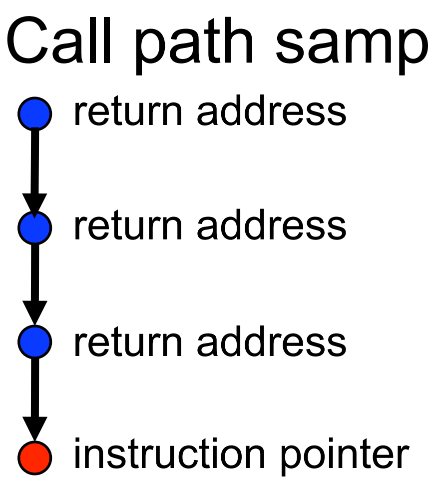
- Overview of Rice's HPCToolkit
- **Accurate measurement**
- Useful source-level feedback
- Effective performance analysis
  - derived metrics for understanding performance
  - pinpointing scalability bottlenecks [SC09]
  - analyzing lock contention in threaded codes [PPoPP10]
  - pinpointing load imbalance [SC10]
  - understanding temporal dynamics of parallel codes
- Using HPCToolkit
- Coming attractions

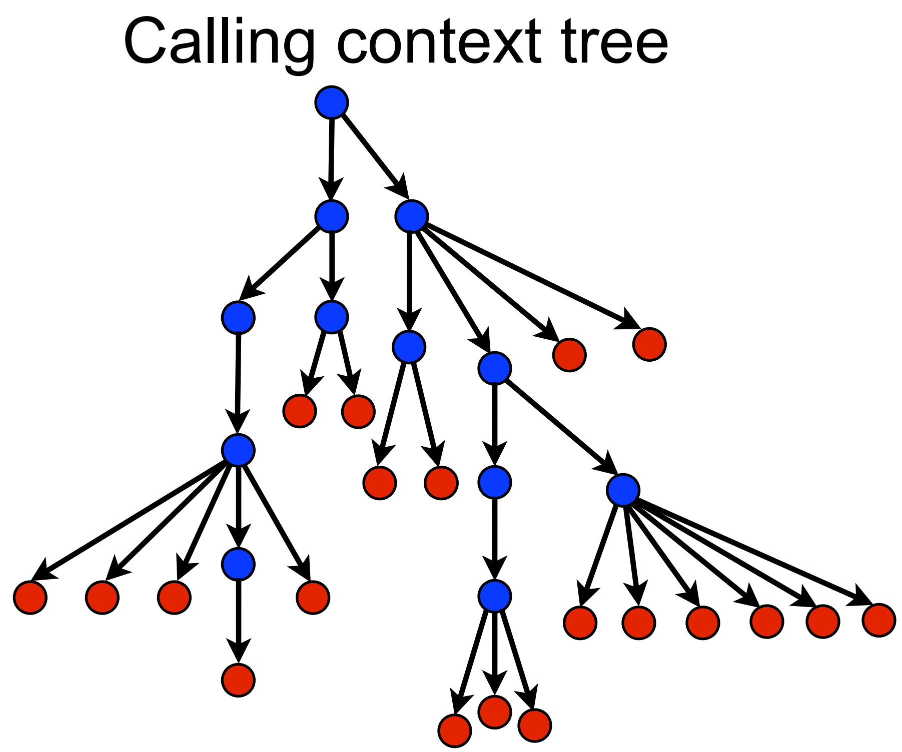
Measurement



Call Path Profiling

- **Measure and attribute costs in context**
  - **sample timer or hardware counter overflows**
  - **gather calling context using stack unwinding**





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

Unwinding Optimized Code

- **Optimized code presents challenges for unwinding**
  - —optimized code often lacks frame pointers
  - —no compiler information about epilogues
  - —routines may have multiple epilogues, multiple frame sizes
  - —code may be partially stripped: no info about function bounds
- **What we need**
  - —where is the return address of the current frame?
    - – a register, relative to SP, relative to BP
  - —where is the FP for the caller's frame?
    - – a register, relative to SP, relative to BP
- **Approach: use binary analysis to support unwinding**

Dynamically Loaded Code (Linux)

New code may be loaded/unloaded at any time

- **When a new module is loaded**
  - —note new code segment mappings
  - —build table of new procedure bounds
- **When a module is unloaded**
  - —mark end of profiler epoch: code addresses no longer apply
  - —flush stale cached information

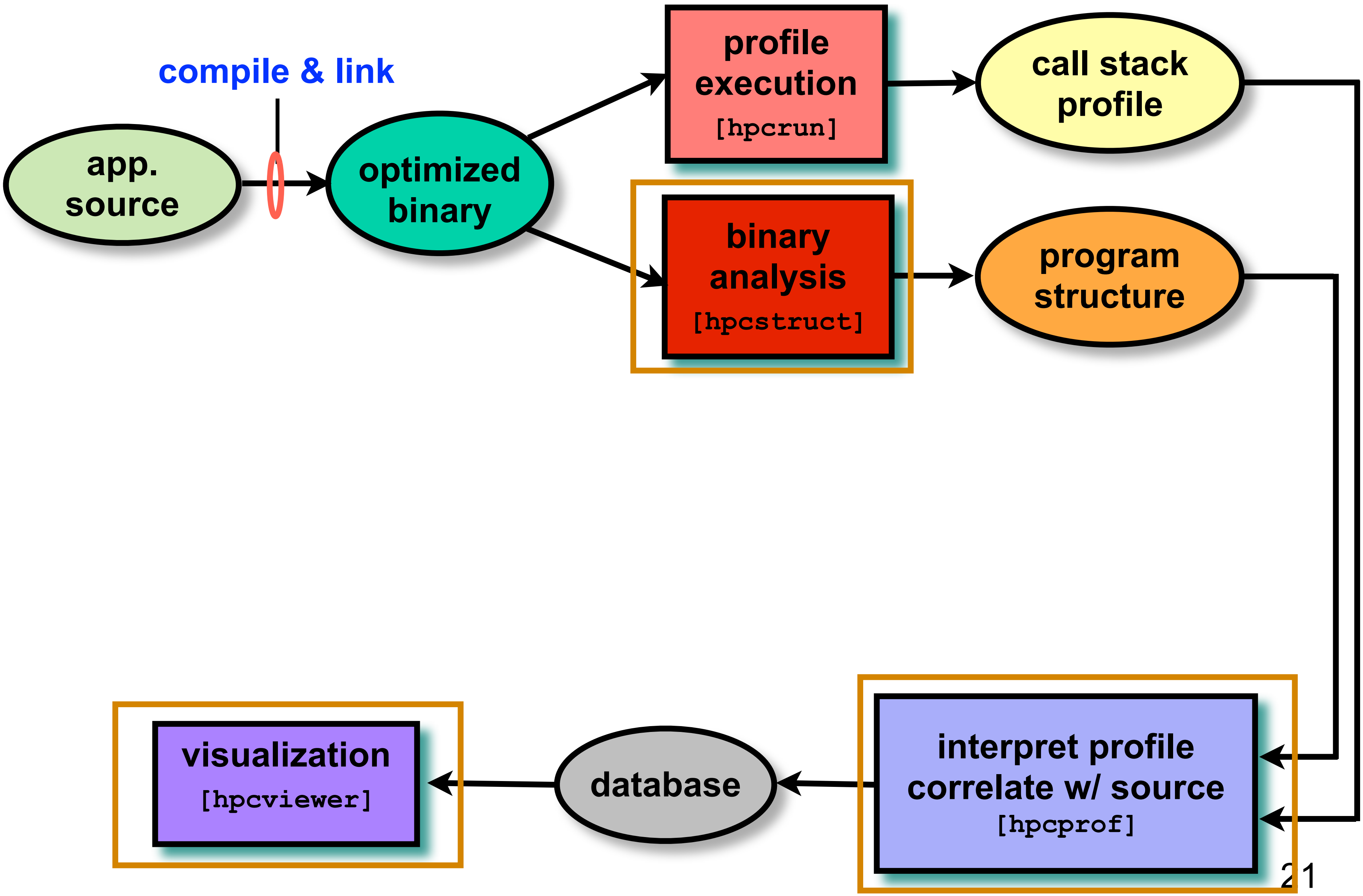
Measurement Effectiveness

- **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
- Useful source-level feedback
- Effective performance analysis
  - derived metrics for understanding performance
  - pinpointing scalability bottlenecks [SC09]
  - analyzing lock contention in threaded codes [PPoPP10]
  - pinpointing load imbalance [SC10]
  - understanding temporal dynamics of parallel codes
- Using HPCToolkit
- Coming attractions

Useful Source-level Feedback



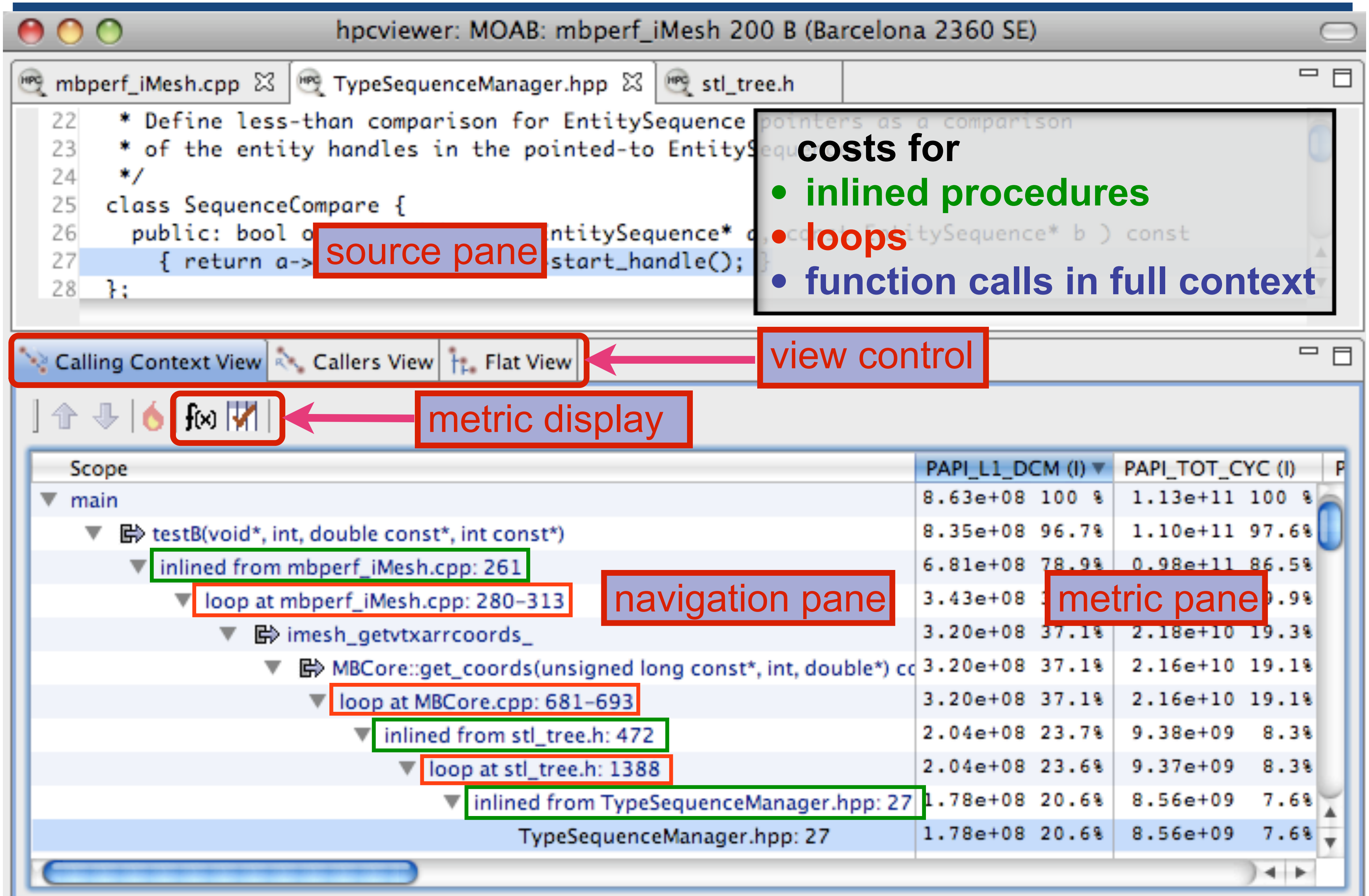
Recovering Program Structure

- 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`



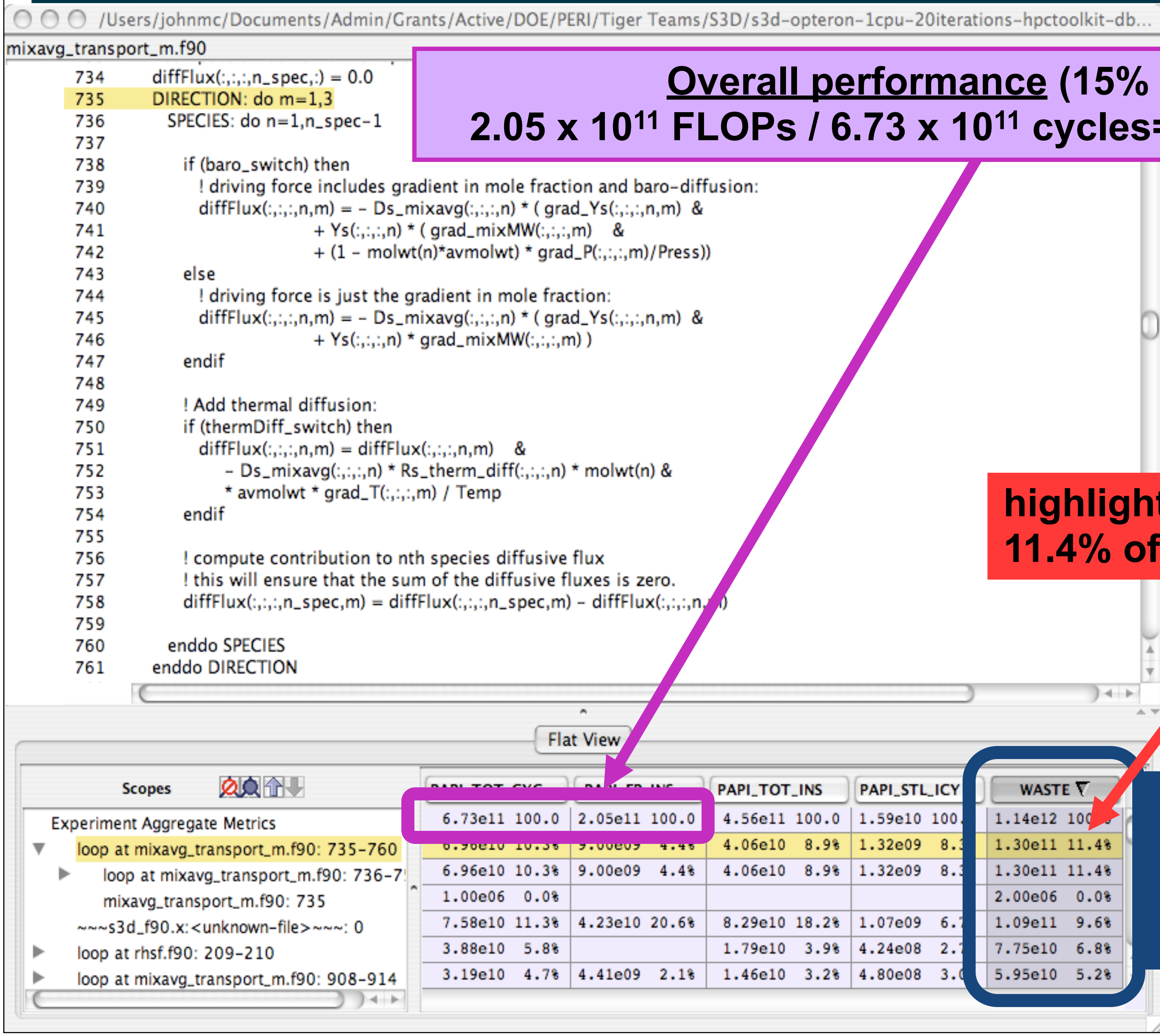
Principal Views

- **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**
  - — "flatten" 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
- Useful source-level feedback
- Effective performance analysis
  - — derived metrics for understanding performance
  - — pinpointing scalability bottlenecks [SC09]
  - — analyzing lock contention in threaded codes [PPoPP10]
  - — pinpointing load imbalance [SC10]
  - — understanding temporal dynamics of parallel codes
- Using HPCToolkit
- Coming attractions

S3D Solver for Turbulent, Reacting Flows



/Users/johnmc/Documents/Admin/Grants/Active/DOE/PERI/Tiger Teams/S3D/s3d-opteron-1cpu-20iterations-hpctoolkit-db...

mixavg_transport_m.f90

```
734       diffFlux(:,:,:,n_spec,:) = 0.0
735       DIRECTION: do m=1,3
736         SPECIES: do n=1,n_spec-1
737
738           if (baro_switch) then
739             ! driving force includes gradient in mole fraction and baro-diffusion:
740             diffFlux(:,:,:,n,m) = - Ds_mixavg(:,:,:,n) * ( grad_Ys(:,:,:,n,m)  &
741                            + Ys(:,:,:,n) * ( grad_mixMW(:,:,:,m)    &
742                            + (1 - molwt(n)*avmolwt) * grad_P(:,:,:,m)/Press))
743           else
744             ! driving force is just the gradient in mole fraction:
745             diffFlux(:,:,:,n,m) = - Ds_mixavg(:,:,:,n) * ( grad_Ys(:,:,:,n,m)  &
746                            + Ys(:,:,:,n) * grad_mixMW(:,:,:,m) )
747           endif
748
749           ! Add thermal diffusion:
750           if (thermDiff_switch) then
751             diffFlux(:,:,:,n,m) = diffFlux(:,:,:,n,m)    &
752                - Ds_mixavg(:,:,:,n) * Rs_therm_diff(:,:,:,n) * molwt(n) &
753                * avmolwt * grad_T(:,:,:,m) / Temp
754           endif
755
756           ! compute contribution to nth species diffusive flux
757           ! this will ensure that the sum of the diffusive fluxes is zero.
758           diffFlux(:,:,:,n_spec,m) = diffFlux(:,:,:,n_spec,m) - diffFlux(:,:,:,n,m)
759
760         enddo SPECIES
761       enddo DIRECTION
```

Flat View

Scopes	[illegible]	[illegible]	PAPI_TOT_INS	PAPI_STL_ICY	WASTE
Experiment Aggregate Metrics	6.73e11 100.0	2.05e11 100.0	4.56e11 100.0	1.59e10 100.	1.14e12 100
loop at mixavg_transport_m.f90: 735-760	[illegible]	[illegible]	4.06e10 8.9%	1.32e09 8.3	1.30e11 11.4%
loop at mixavg_transport_m.f90: 736-7	6.96e10 10.3%	9.00e09 4.4%	4.06e10 8.9%	1.32e09 8.3	1.30e11 11.4%
mixavg_transport_m.f90: 735	1.00e06 0.0%				2.00e06 0.0%
~~s3d_f90.x:<unknown-file>~~~: 0	7.58e10 11.3%	4.23e10 20.6%	8.29e10 18.2%	1.07e09 6.7	1.09e11 9.6%
loop at rhsf.f90: 209-210	3.88e10 5.8%		1.79e10 3.9%	4.24e08 2.7	7.75e10 6.8%
loop at mixavg_transport_m.f90: 908-914	3.19e10 4.7%	4.41e09 2.1%	1.46e10 3.2%	4.80e08 3.0	5.95e10 5.2%

Overall performance (15% of peak)

2.05×10^{11} FLOPs / 6.73×10^{11} cycles= .305 FLOPs/cycle

highlighted loop accounts for 11.4% of total program waste

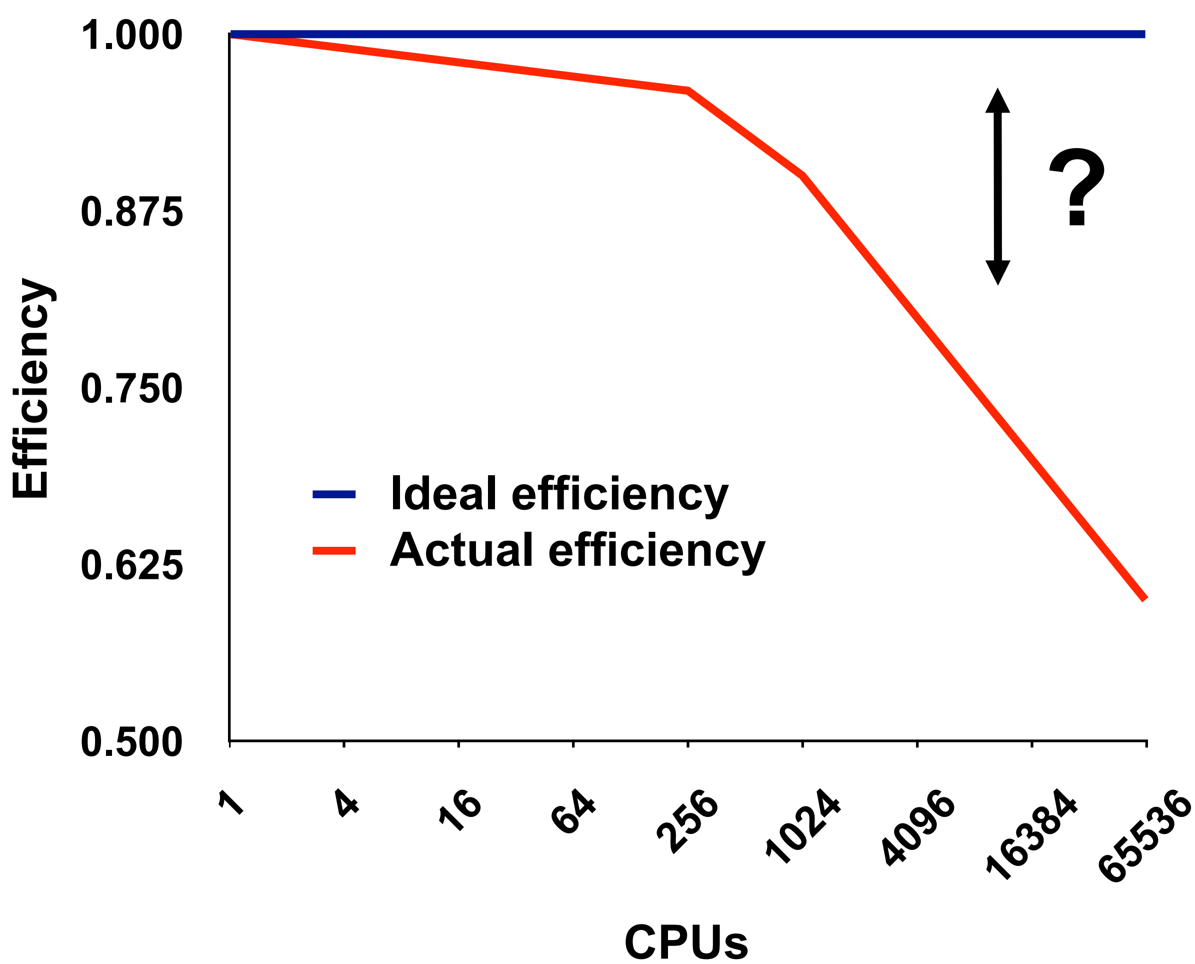
Wasted Opportunity

(Maximum FLOP rate * cycles - (actual FLOPs))

Outline

- Overview of Rice's HPCToolkit
- Accurate measurement
- Useful source-level feedback
- Effective performance analysis
  - derived metrics for understanding performance
  - pinpointing scalability bottlenecks [SC09]
  - analyzing lock contention in threaded codes [PPoPP10]
  - pinpointing load imbalance [SC10]
  - understanding temporal dynamics of parallel codes
- Using HPCToolkit
- Coming attractions

The Problem of Scaling



Note: higher is better

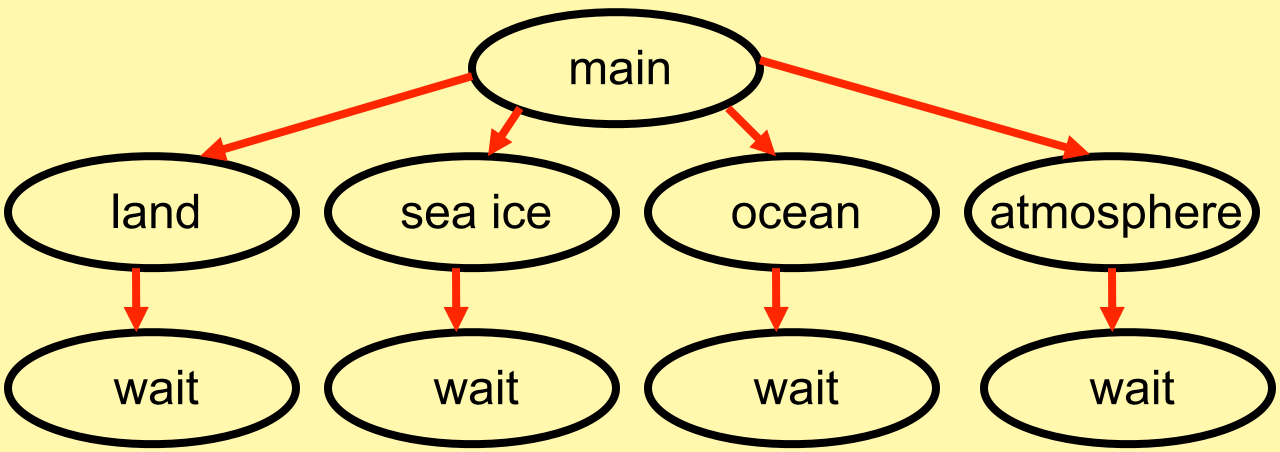
Goal: Automatic Scaling Analysis

- **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**

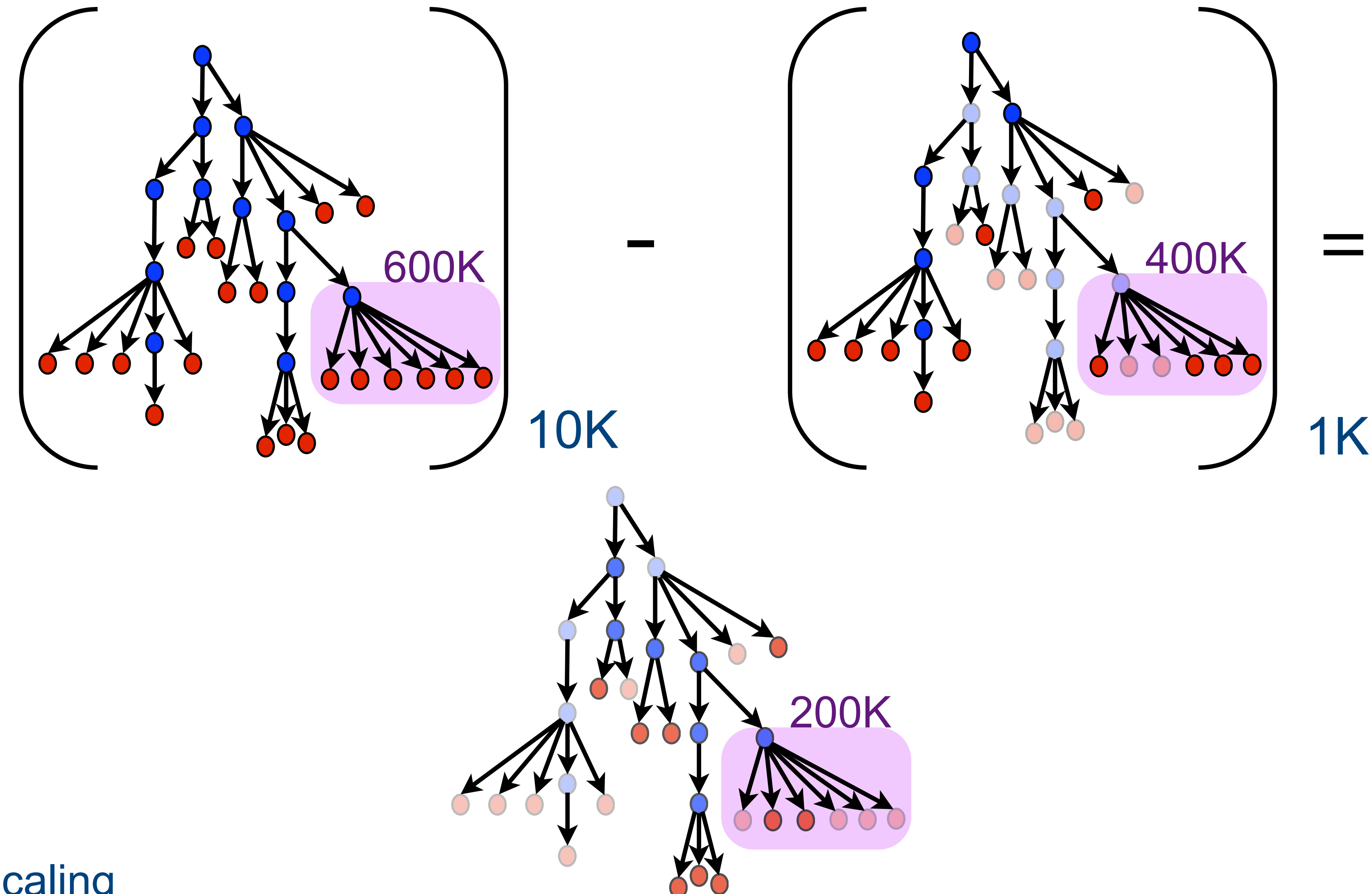
Example: community earth system model



Performance Analysis with Expectations

- **Users have performance expectations for parallel codes**
  - strong scaling: linear speedup
  - weak scaling: constant execution time
- **Putting 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

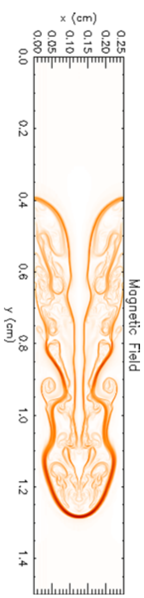
Analyzing Weak Scaling: 1K to 10K processors



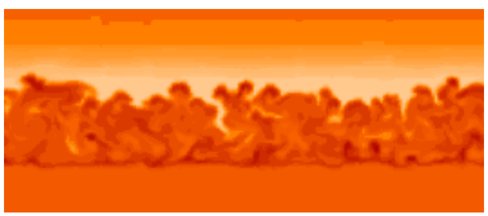
Weak scaling

Scalability Analysis Demo: FLASH

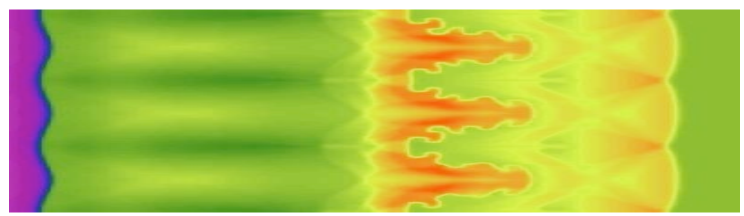
Code:	University of Chicago FLASH
Simulation:	white dwarf detonation
Platform:	Blue Gene/P
Experiment:	8192 vs. 256 processors
Scaling type:	weak



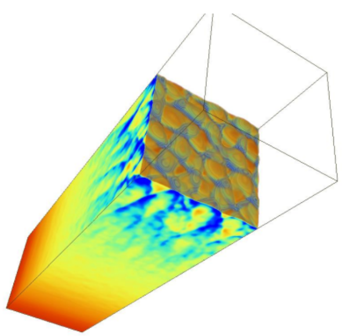
Magnetic Rayleigh-Taylor



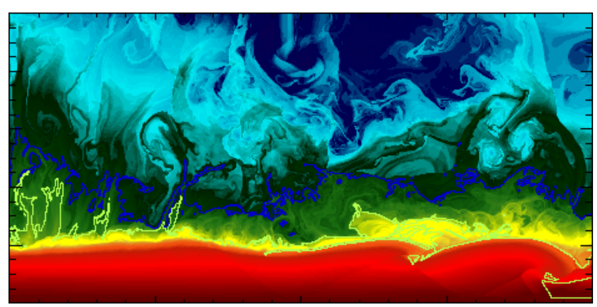
Nova outbursts on white dwarfs



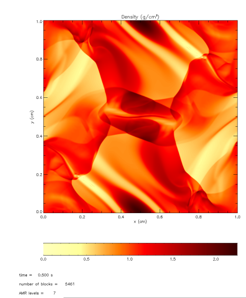
Laser-driven shock instabilities



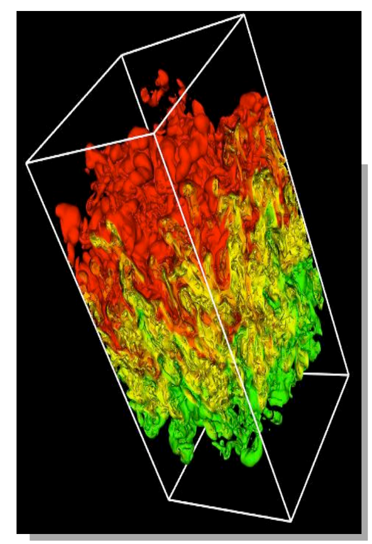
Cellular detonation



Helium burning on neutron stars



Orzag/Tang MHD vortex



Rayleigh-Taylor instability

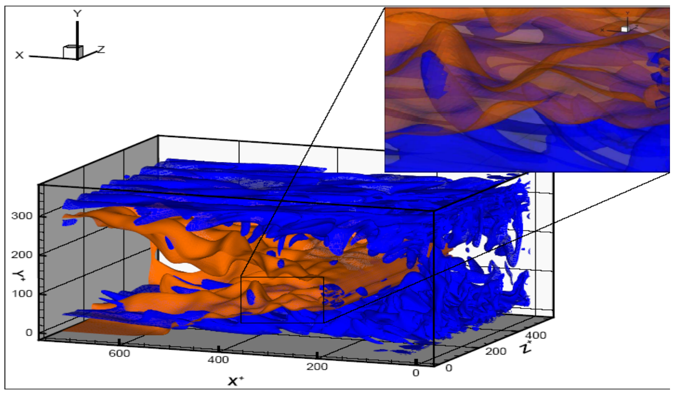
Figures courtesy of FLASH Team, University of Chicago

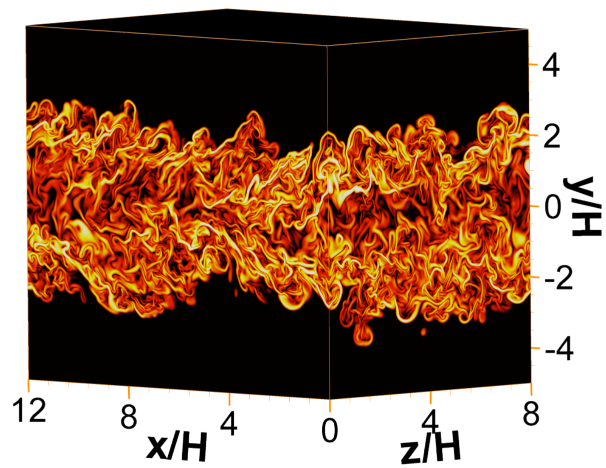
Scaling on Multicore Processors

- **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 - DNS Solver

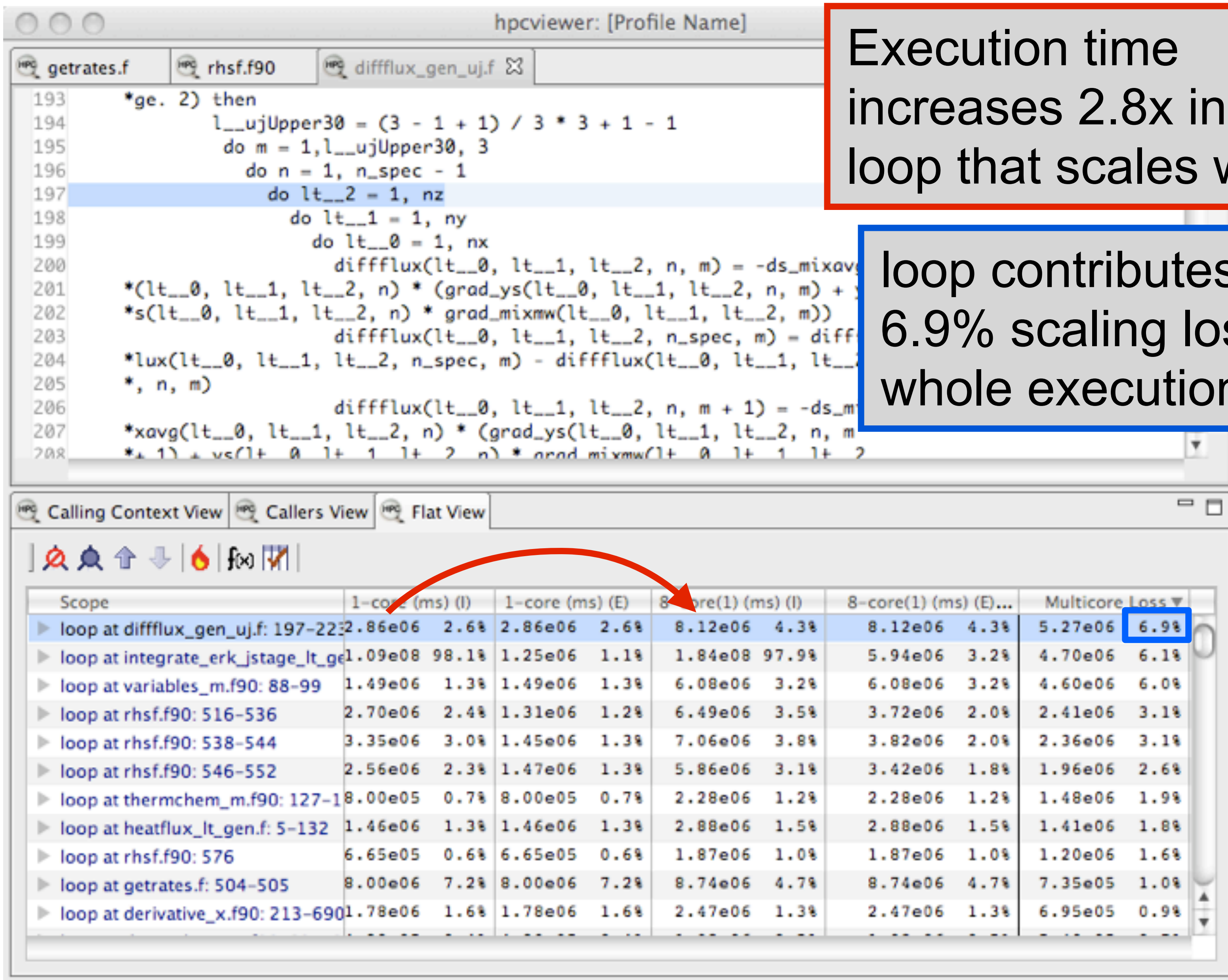
- Solves compressible reacting Navier-Stokes equations
- High fidelity numerical methods
  - 8th order finite-difference
  - 4th order explicit RK integrator
- Hierarchy of molecular transport models
- Detailed chemistry
- Multi-physics (sprays, radiation and soot)
  - from SciDAC-TSTC (Terascale Simulation of Turbulent Combustion)





Text and figures courtesy of Jacqueline H. Chen, SNL

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

Outline

- Overview of Rice's HPCToolkit
- Accurate measurement
- Useful source-level feedback
- Effective performance analysis
  — derived metrics for understanding performance
  — pinpointing scalability bottlenecks [SC09]
  — analyzing lock contention in threaded codes [PPoPP10]
  — pinpointing load imbalance [SC10]
  — understanding temporal dynamics of parallel codes
- Using HPCToolkit
- Coming attractions

Understanding Lock Contention in Threaded Code

- Lock contention => idleness
  - explicitly threaded programs (Pthreads, etc)
  - implicitly threaded programs (critical sections in OpenMP, ...)
- Strategy: “blame-shifting” of contention from victim to perpetrator
  - use shared state (locks) to communicate blame
- How it works
  - consider spin-waiting
  - sample a working thread:
    - charge to ‘work’ metric
  - sample an idle thread
    - accumulate in idleness counter associated with a lock (atomic add)
  - working thread releases a lock
    - atomically swap 0 with lock’s idleness counter
    - exactly represents contention while that thread held the lock
    - unwind the call stack to attribute lock contention to a calling context

Lock Contention in MADNESS

```
add(MEMFUN_OBJT(memfunT)& obj,
    memfunT memfun,
    const arg1T& arg1, const arg2T& arg2, const a
    Future<REMFUTURE(MEMFUN_RETURNT(memfunT))> re
    add(new TaskMemfun<memfunT>(result,obj,memfun
    return result;
}
```

Quantum chemistry; MPI + pthreads
- **65M distinct locks**
- **max. of 340K live locks**
- **30K lock acquisitions/sec/thread**

1-5% overhead

Calling Context View | Callers View | Flat View

16 cores; 1 thread/core (4 x Barcelona)

μs

Scope	% idleness (all/E)	idleness (all/E)
Experiment Aggregate Metrics	2.35e+01 100 %	1.57e+09 100 %
pthread_spin_unlock	2.35e+01 100.0	
madness::Spinlock::unlock() const	2.35e+01 100.0	
inlined from worldmutex.h: 142	1.78e+01 75.6%	
madness::ThreadPool::add(madness::PoolTaskInterface*)	1.78e+01 75.6%	
inlined from worldtask.h: 581	7.35e+00 31.2%	4.92e+08 31.2%
madness::Future<> madness::WorldObject<>::task<>(	7.35e+00 31.2%	4.92
inlined from worldtask.h: 569	4.56e+00 19.4%	3.05
madness::Future<> madness::WorldObject<>::task<>(	4.56e+00 19.4%	3.05
inlined from worlddep.h: 68	1.53e+00 6.5%	1.02
inlined from worldtask.h: 570	1.49e+00 6.3%	9.97e+07 6.3%
madness::Future<> madness::WorldObject<>::task<>(	1.49e+00 6.3%	9.97e+07 6.3%
inlined from worldtask.h: 558	1.38e+00 5.9%	9.26e+07 5.9%
madness::Future<> madness::WorldTaskQueue::add<>(ma	6.72e-01 2.9%	4.49e+07 2.9%

lock contention accounts for **23.5**% of execution time.

Adding futures to shared global work queue.

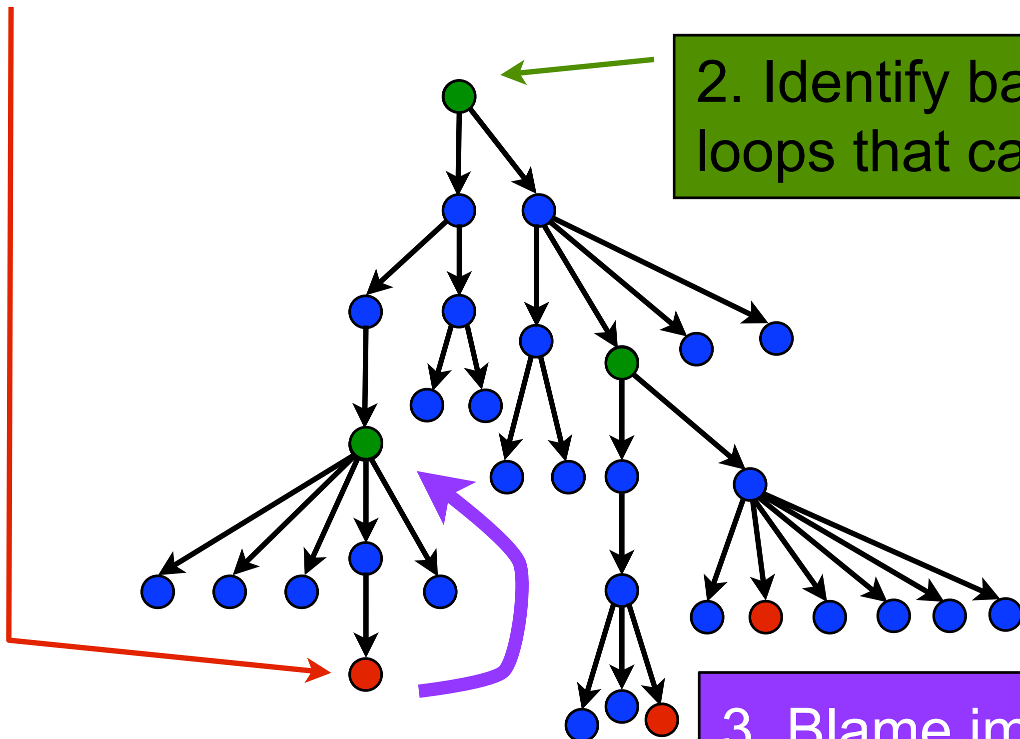
Outline

- Overview of Rice's HPCToolkit
- Accurate measurement
- Useful source-level feedback
- Effective performance analysis
  - — derived metrics for understanding performance
  - — pinpointing scalability bottlenecks [SC09]
  - — analyzing lock contention in threaded codes [PPoPP10]
  - — pinpointing load imbalance [SC10]
  - — understanding temporal dynamics of parallel codes
- Using HPCToolkit
- Coming attractions

Identifying Load Imbalance Post Mortem

1. Identify exposed waiting: all imbalance is manifested in waiting

2. Identify balance points (procedures or loops that cannot contribute to imbalance)



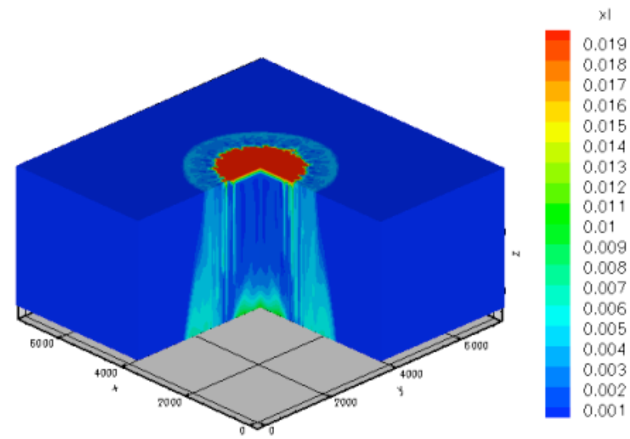
3. Blame imbalance on the computation subtree in which it originates

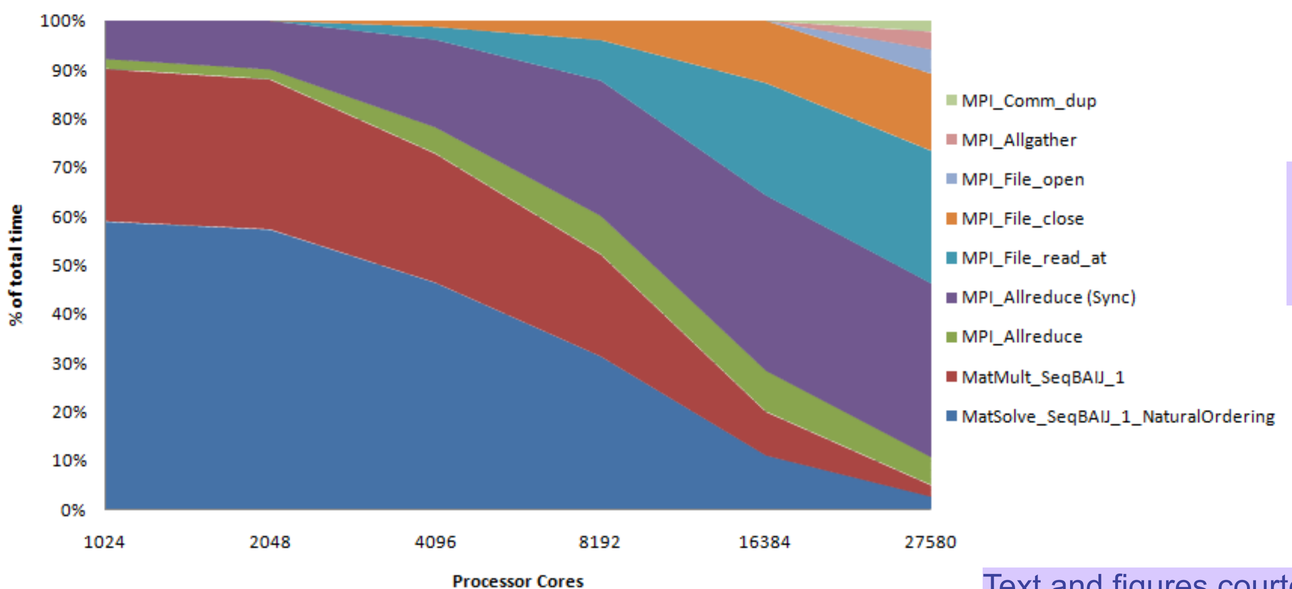
4. Associate each (summary) node with thread-level metric values

Load Imbalance Analysis Example

PFLOTRAN: modeling multi-scale, multiphase, multi-component subsurface reactive flows

Example use: modeling sequestration of CO_2 in deep geologic formations, where resolving density-driven fingering patterns is necessary to accurately describe the rate of dissipation of the CO_2 plume





Strong scaling study on Cray XT

Text and figures courtesy of PFLOTRAN Team

PFLOTRAN

8K cores, Cray XT5

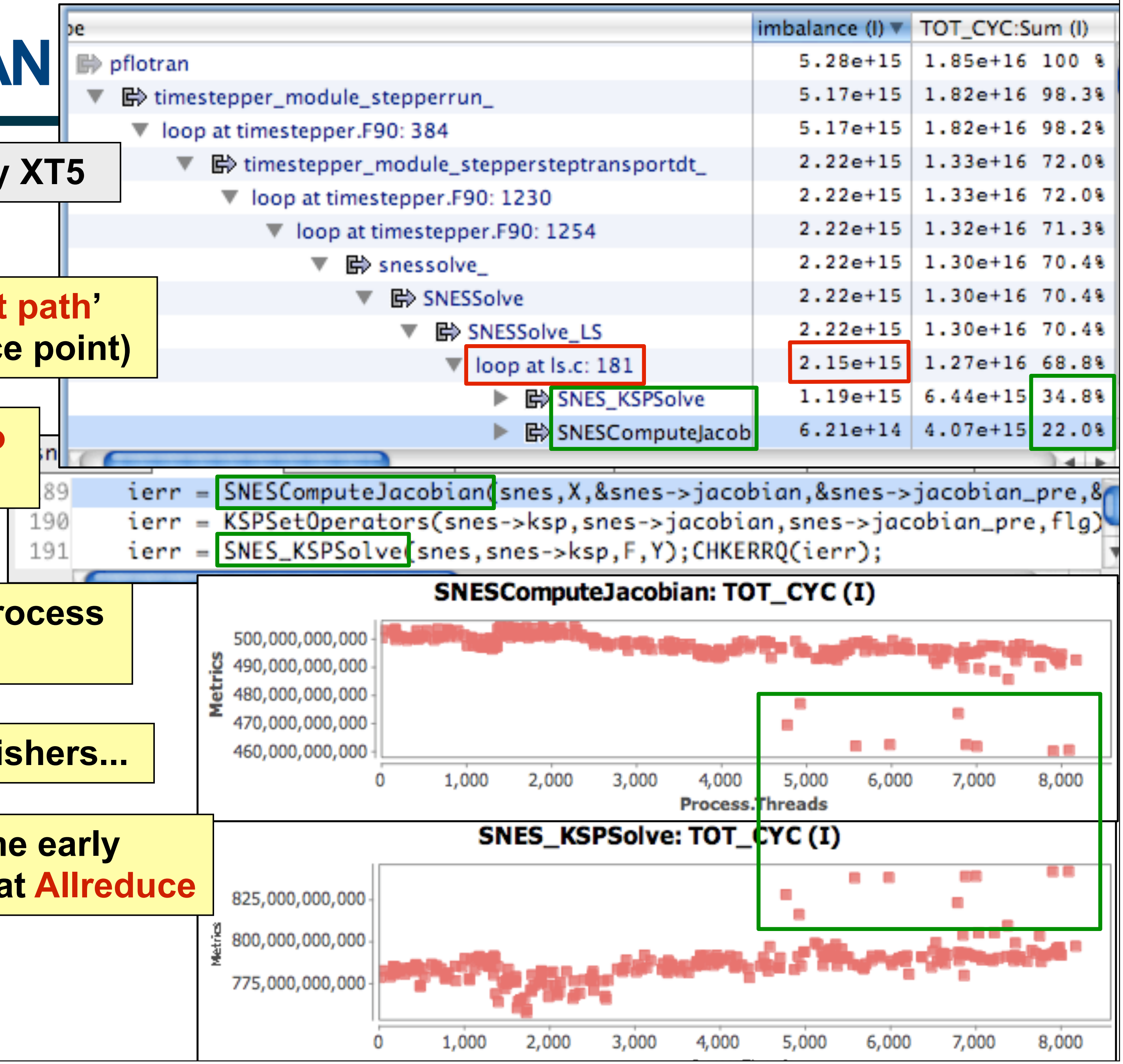
1. Drill down 'hot path' to loop (a balance point)

2. Notice top two call sites...

3. Plot the per-process values:

Early finishers...

... become early arrivers at Allreduce

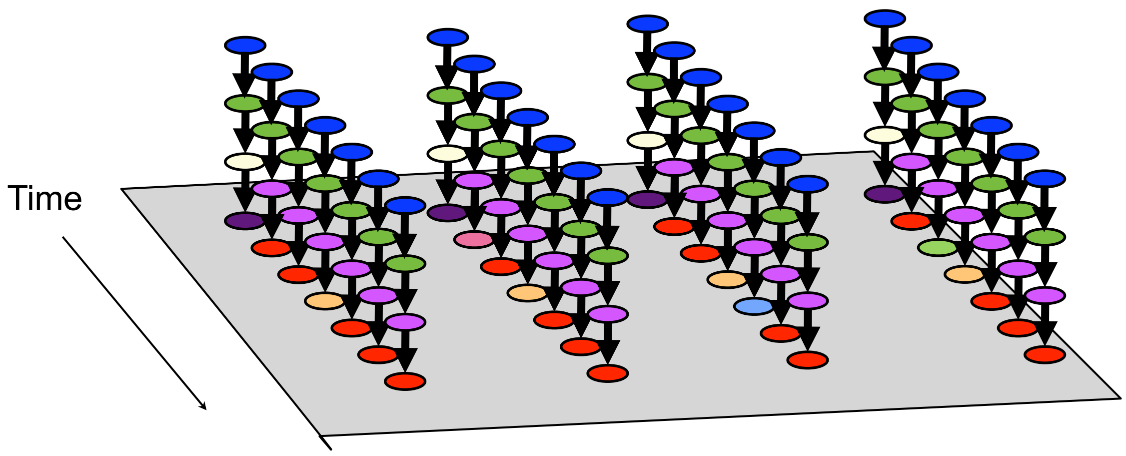


Outline

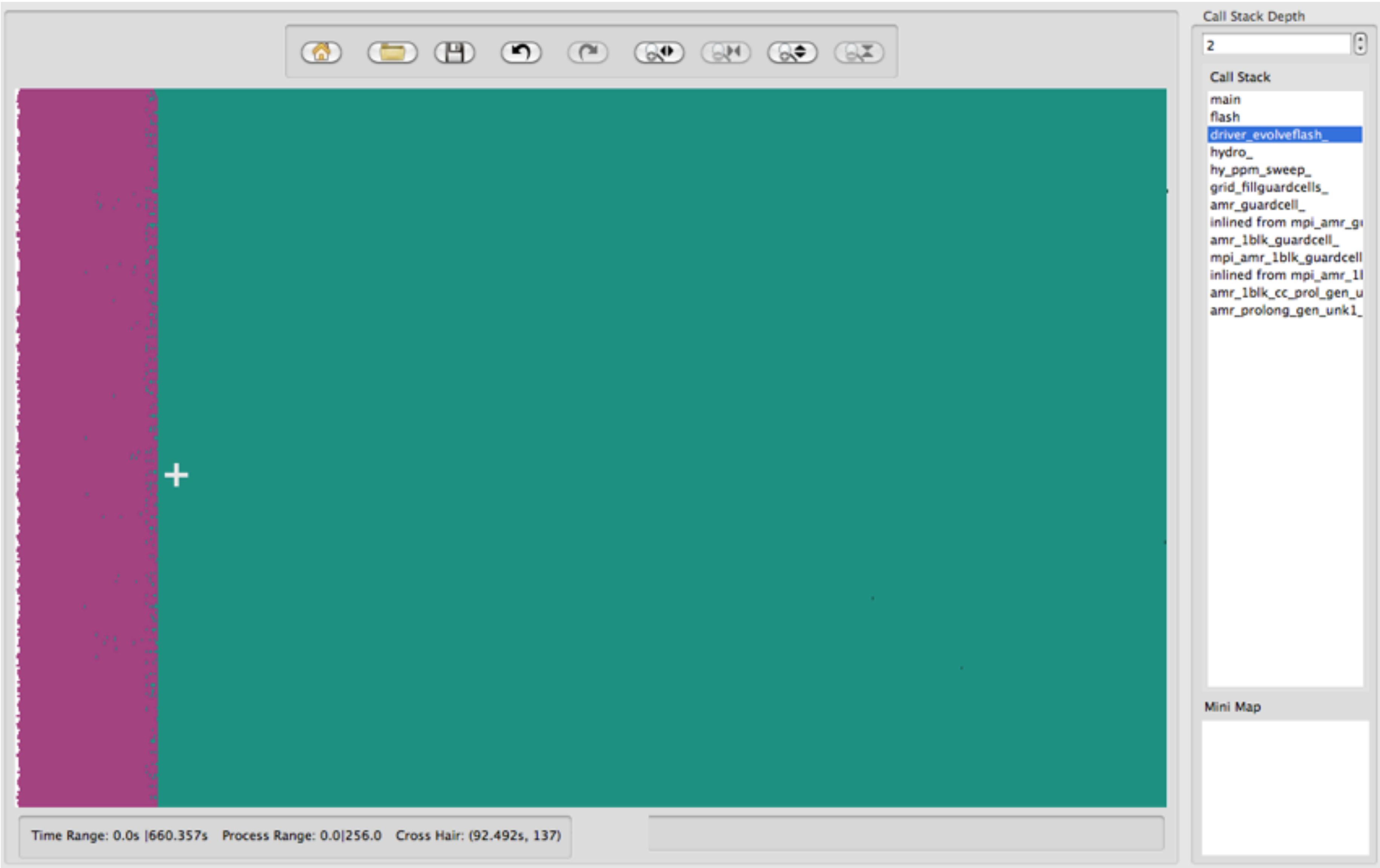
- **Overview of Rice's HPCToolkit**
- **Accurate measurement**
- **Useful source-level feedback**
- **Effective performance analysis**
  - — **derived metrics for understanding performance**
  - — **pinpointing scalability bottlenecks [SC09]**
  - — **analyzing lock contention in threaded codes [PPoPP10]**
  - — **pinpointing load imbalance [SC10]**
  - — **understanding temporal dynamics of parallel codes**
- **Using HPCToolkit**
- **Coming attractions**

Understanding Temporal Behavior

- Profiling compresses out the temporal dimension
  - that's why serialization is 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 execution with a depth slice

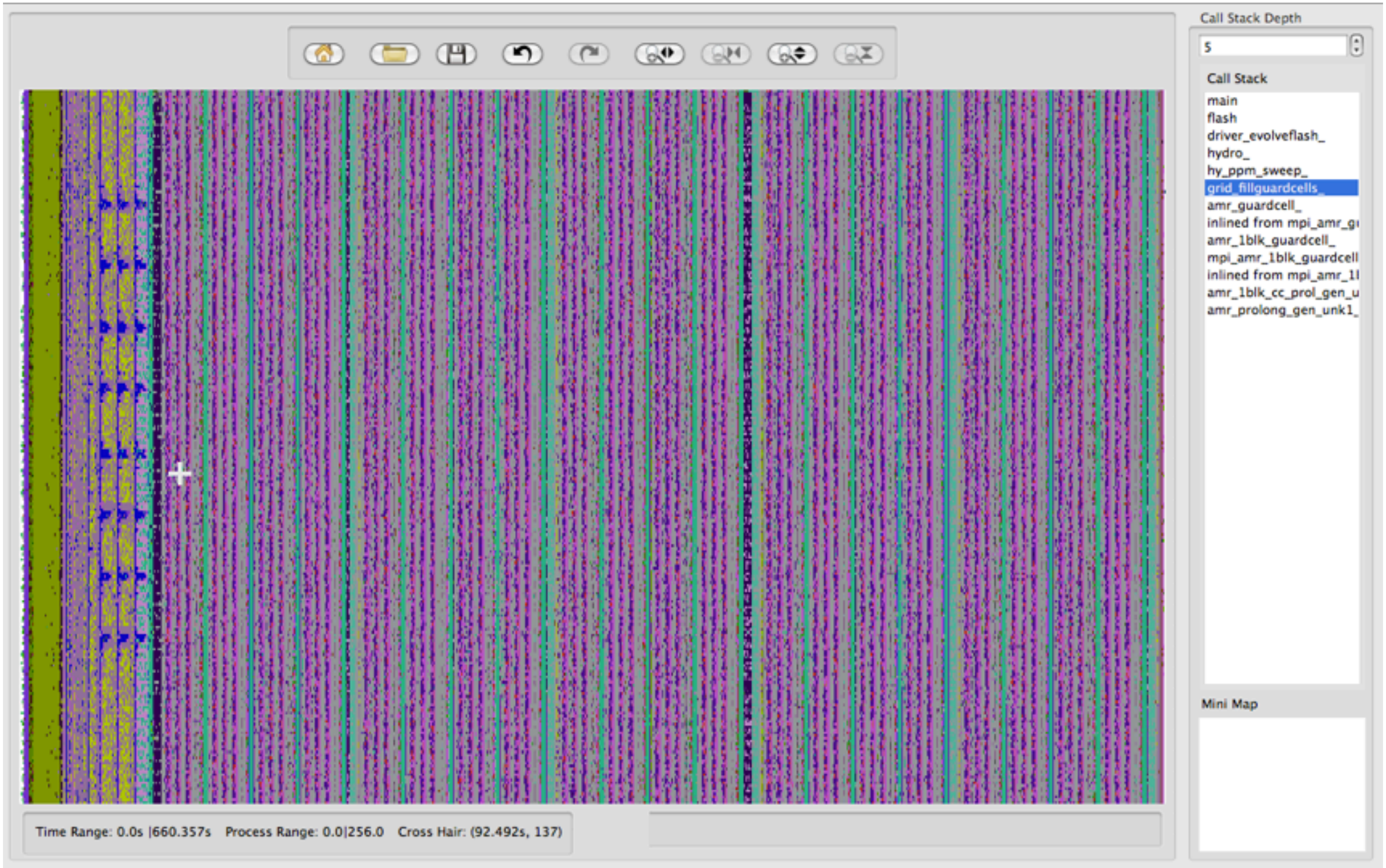


Flash White Dwarf Collapse on 256 Cores



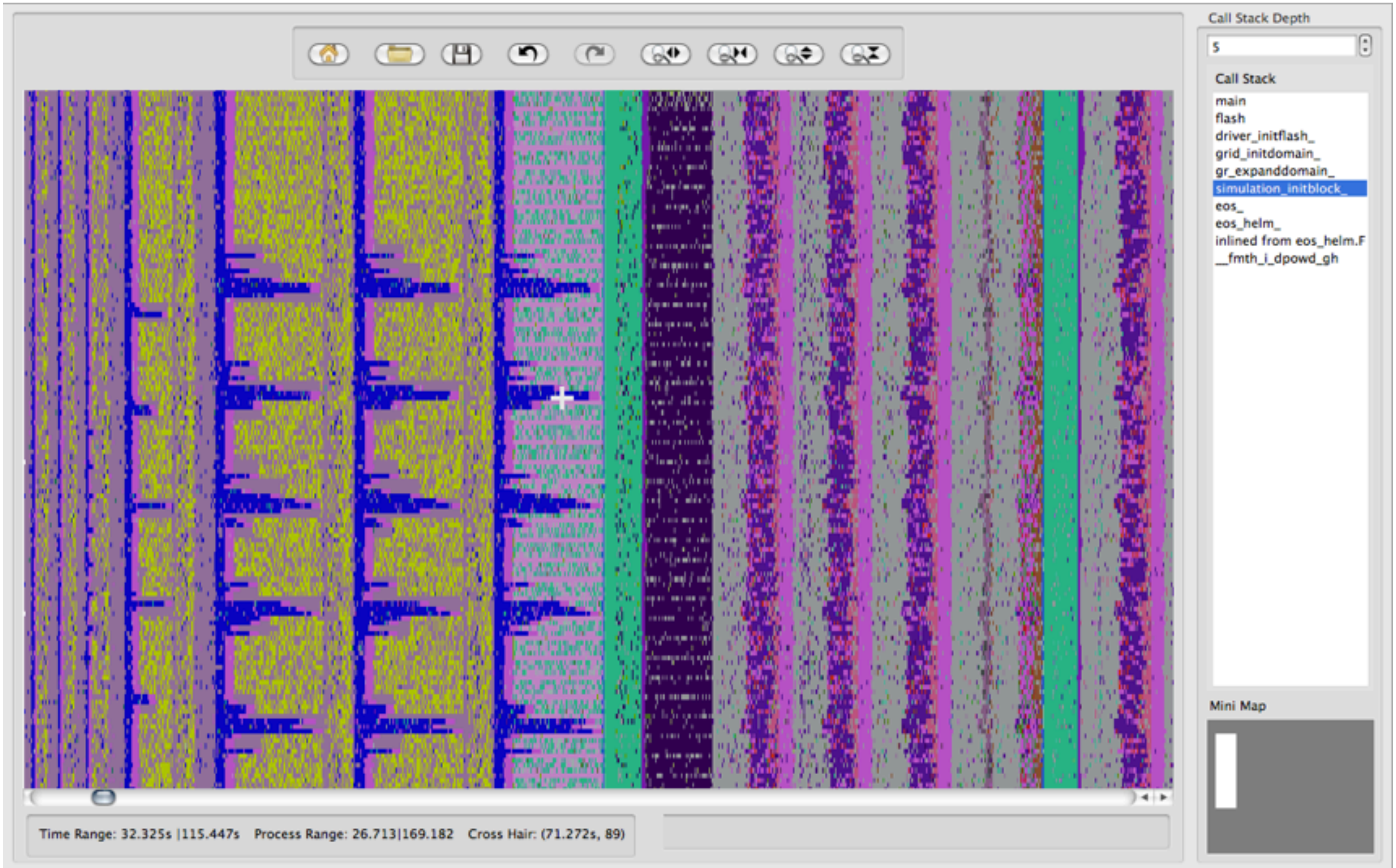
Full execution at call stack depth 2

Flash White Dwarf Collapse on 256 Cores



Full execution at call stack depth 5

Flash White Dwarf Collapse on 256 Cores



Execution detail at call stack depth 5

Outline

- **Overview of Rice's HPCToolkit**
- **Accurate measurement**
- **Useful source-level feedback**
- **Effective performance analysis**
  - — derived metrics for understanding performance
  - — pinpointing scalability bottlenecks [SC09]
  - — analyzing lock contention in threaded codes [PPoPP10]
  - — pinpointing load imbalance [SC10]
  - — understanding temporal dynamics of parallel codes
- **Using HPCToolkit**
- **Coming attractions**

Where to Find HPCToolkit

- DOE Systems
  - jaguar: /ccs/proj/hpctoolkit/pkgs/hpctoolkit
  - intrepid: /home/projects/hpctoolkit/pkgs/hpctoolkit
  - franklin: /project/projectdirs/hpctk/pkgs/hpctoolkit
- NSF Systems
  - ranger: /scratch/projects/hpctoolkit/pkgs/hpctoolkit
- For your local Linux systems, you can download and install it
  - documentation, build instructions, link to our svn repository
    - svn repository: https://outreach.scidac.gov/svn/hpctoolkit
  - we recommend downloading and building from svn
  - important notes:
    - obtaining information from hardware counters requires downloading and installing PAPI
    - installing PAPI
      - on Linux 2.6.32 or better: built-in kernel support for counters
      - earlier Linux needs a kernel patch (perfmon2 or perfctr)

Available Guides

http://hpctoolkit.org/documentation.html

- **Using HPCToolkit with statically linked programs [pdf]**
  - a guide for using hpctoolkit on BG/P and Cray XT
- **Quick start guide [pdf]**
  - essential overview that almost fits on one page
- **The `hpcviewer` user interface [pdf]**
- **Effective strategies for analyzing program performance with HPCToolkit [pdf]**
  - analyzing scalability, waste, multicore performance ...
- **HPCToolkit and MPI [pdf]**
- **HPCToolkit Troubleshooting [pdf]**
  - why don't I have any source code in the viewer?
  - hpcviewer isn't working well over the network ... what can I do?

Setup

- 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
  - 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"
  - statically-linked executables (e.g., Cray XT, BG/P)
    - use hpclink to link your executable
    - launch executable with environment var HPCRUN_EVENT_LIST=LIST

      (currently BG/P hardware counters unsupported)

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_DCM@400000 PAPI_TLB_DM@400000 PAPI_FP_OPS@400000"`
    - to collect a trace of WALLCLOCK samples

      `setenv HPCRUN_OPT_TRACE=1`
  - Blue Gene/P: pass environment settings to qsub
    - ```
      qsub -A YourAllocation  -q prod -t 30  -n 2048  \
      --proccount 8192 --mode vn --env \
      HPCRUN_EVENT_LIST=WALLCLOCK@1000 flash3.hpc
      ```
    - to collect a trace of WALLCLOCK samples use

      `HPCRUN_EVENT_LIST=WALLCLOCK:HPCRUN_OPT_TRACE=1`
      ```

Binary Analysis and Data Assessment

- **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`

Analyzing Data with hpcprof-mpi

- **Analyze call graph profiles from <u>all cores</u> together**
  - perform analysis in parallel for acceptable analysis time
- **Purpose:**
  - compute summary statistics across nodes
    - enables top-down investigation of node differences
  - provide access to thread-level data for detailed comparisons
- **Mechanics**
  - hpcprof-mpi is just an MPI program
    - launch it on an appropriate number of nodes to reduce analysis time
      - e.g. analysis of PFLOTRAN on Cray XT: 8K profiles, 48 nodes, 10 min
    - e.g. `qsub -A YourAllocation -q prod-devel -t 20 -n 64 hpcprof-mpi -S myapp.hpcstruct -I "path_to_src/*" hpctoolkit-myapp-measurements-5912`
  - produces hpctoolkit-myapp-database-5912

Analyzing Data with hpcprof

- **This runs on the head node; can't analyze all performance data there for large parallel executions**
- **Use hpcprof to analyze one (or a few) measurement files**
  - select one or a few files from your measurements to analyze
  - e.g. `hpcprof -S myapp.hpcstruct -I "path_to_src/*" hpctoolkit-myapp-measurements-5912/myapp-0000-000-983409-764.hpcrun`
  - produces hpctoolkit-myapp-database-5912

Using hpcviewer and hpctraceview

- Notes
  - if you collected traces or used hpcprof-mpi, your performance database will be large
    - best approach: analyze it on the leadership computing platform
  - you can tar up a database for analysis on your laptop
    - with patience: copy whole database to laptop
    - impatient way: tar up database without thread or trace data
- Use hpcviewer to open a performance database
  - if using hpcviewer on a the leadership computing platform, add recent Java implementation to your path (for hpcviewer)
    - Cray XT: module load java
    - Blue Gene/P: add /opt/soft/.../java/bin to your path
  - on a front-end node, run hpcviewer with the performance database as an argument
  - ALCF: can also run hpcviewer on gadzooks or eureka
- Use hpctraceview to open call stack traces of core activity
  - run hpctraceview and open performance database

hpcviewer and hpctraceview tip

- **When running interactive viewers on leadership platforms**
  - create a virtual desktop with vncserver
  - view the virtual desktop with vncviewer
  - run hpcviewer or hpctraceview inside your virtual desktop

A Note About `hpcstruct` and Fortran

- **Fortran compilers emit machine code 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**
- **Useful for IBM's xlf, PGI's pgf90 and others**

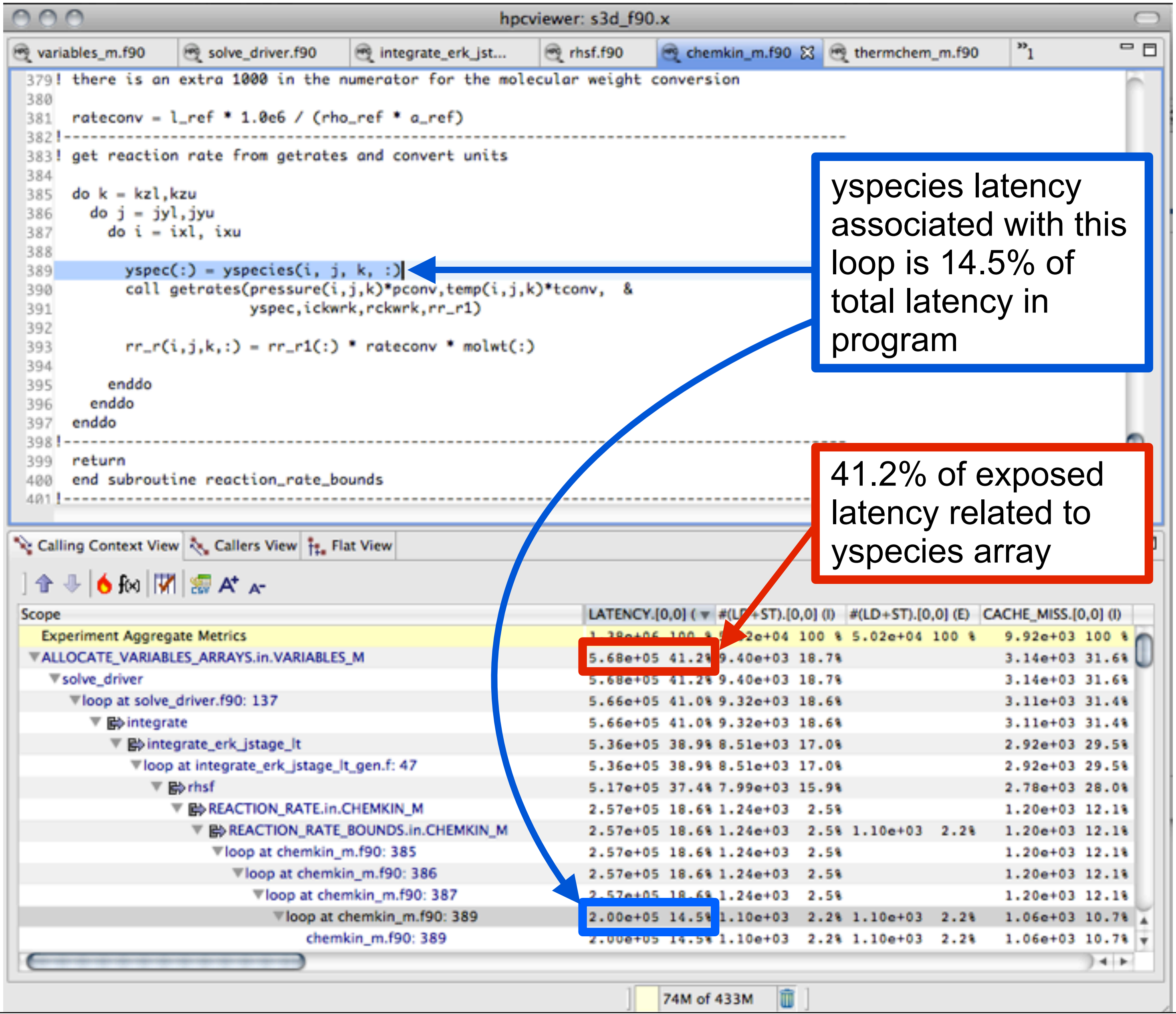
Outline

- Overview of Rice's HPCToolkit
- Accurate measurement
- Useful source-level feedback
- Effective performance analysis
  - — derived metrics for understanding performance
  - — pinpointing scalability bottlenecks [SC09]
  - — analyzing lock contention in threaded codes [PPoPP10]
  - — pinpointing load imbalance [SC10]
  - — understanding temporal dynamics of parallel codes
- Using HPCToolkit
- **Coming attractions**

Coming Attraction: Data Centric Analysis

- Goal: associate memory hierarchy locality problems with particular data structures
- Approach
  - intercept memory allocations to associate data range with allocation
  - associate latency with data structures using “instruction based sampling” capability of AMD Opteron CPUs
    - identify instances of loads and store instructions
    - identify the data structure an access touches based on L/S address
    - measure the total latency associated with each L/S
  - present results in hpcviewer

Data Centric Analysis of S3D



HPCToolkit Summary

- Obtain insight, accuracy & precision by combining call path profiling, binary analysis, and blame shifting
- Show surprisingly effective measurement and source-level attribution for fully optimized code (1-3% overhead)
  - statements in their full static and dynamic context
  - project low-level measurements to much higher levels
- Sampling-based measurements can deliver insight into a range of phenomena
  - scalability bottlenecks
  - sources of lock contention
  - load imbalance
  - temporal dynamics
  - problematic data structures

Some Challenges Ahead

- Data management for scalable measurement and analysis
- Moving from descriptive to prescriptive feedback
- Increasing importance of threading as core counts increase
- Heterogeneous architectures, e.g. GPU accelerators