Gaining Insight into Parallel Program Performance Using Sampling

John Mellor-Crummey
DOE Center for Scalable Application Development Software
johnmc@cs.rice.edu

Collaborators: Nathan Tallent, Michael Fagan, Mark Krentel, Laksono Adhianto, Xu Liu, Reed Landrum, Michael Franco

SciDAC Scientific Discovery through Advanced Computing

RICE

Motivation

- **Complex hardware**
  - — **multi-level parallelism**
    - **ILP, short vectors, multiple cores, multiple sockets, multiple nodes**
  - — **large-scale parallelism**
- **Sophisticated software**
  - — **multiphysics, multiscale, adaptive**
- **Wide gap between peak and typical performance**

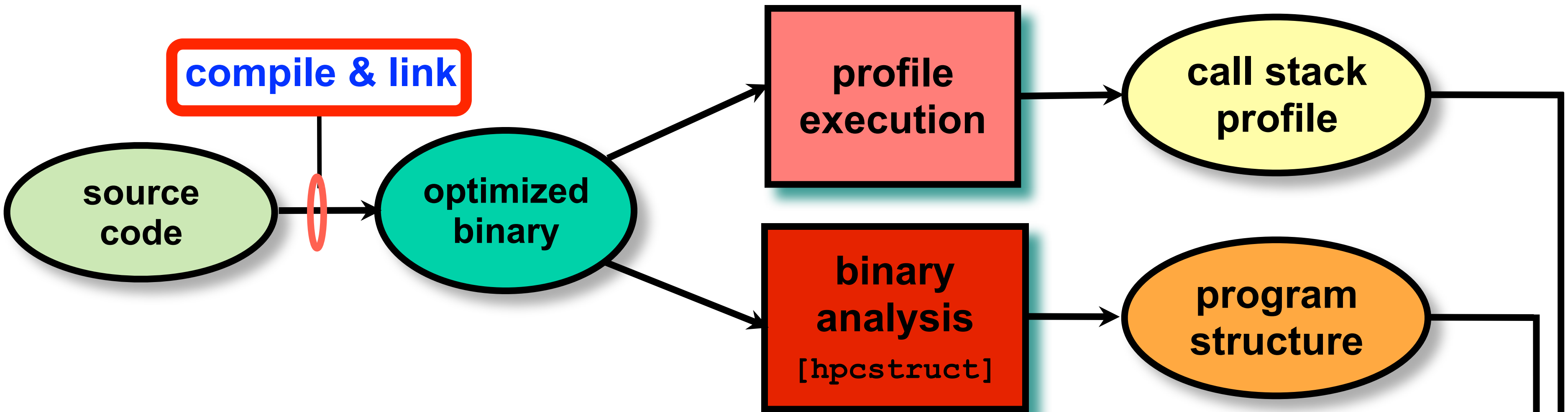
Challenges

- **Understand where and why performance losses occur in sophisticated parallel codes on complex parallel hardware**
- **Identify opportunities for improvement**
- **Quantify potential benefits**

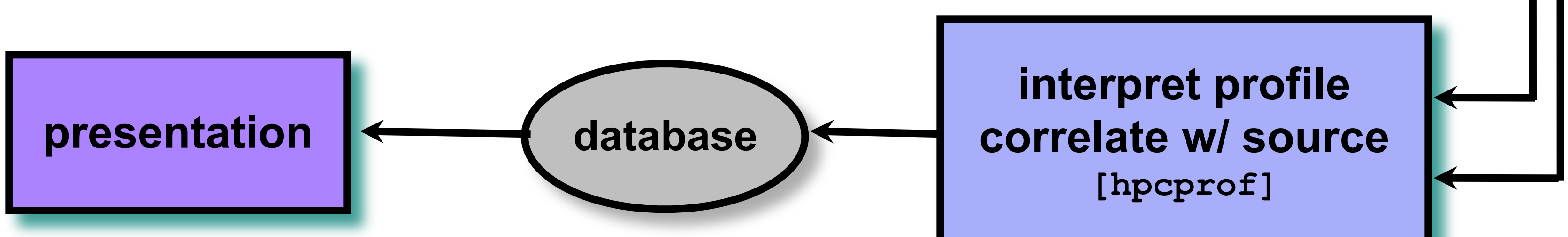
Performance Analysis Goals

- **Accurate measurement of parallel scientific codes**
  - — large, multi-lingual programs
  - — fully optimized code: loop optimization, templates, inlining
  - — binary-only libraries, sometimes partially stripped
  - — complex execution environments
    - dynamic loading or static binaries
    - SPMD parallel codes with threaded node programs
    - batch jobs
  - — production executions
- **Effective performance analysis**
  - — pinpoint and explain problems
    - intuitive enough for scientists and engineers
    - detailed enough for compiler writers
  - — yield actionable results
- **Scalable to petascale systems**

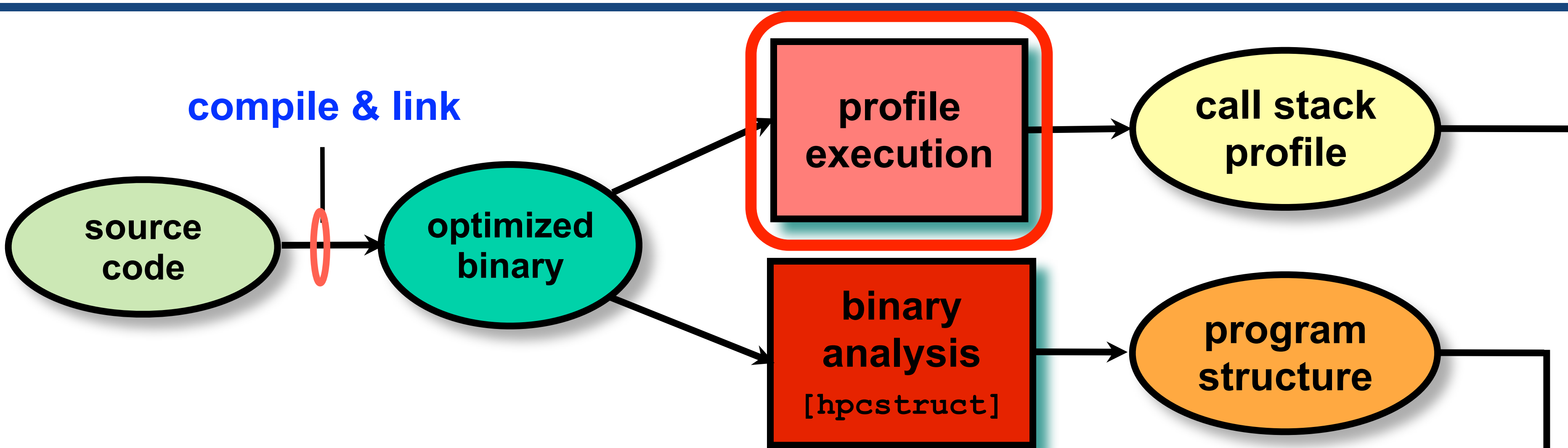
HPCToolkit Performance Tools



- Compile and link for production
  - with full optimization
- For statically-linked executables (e.g. for Cray XT or BG/P)
  - use `hpclink` script to incorporate our monitoring library

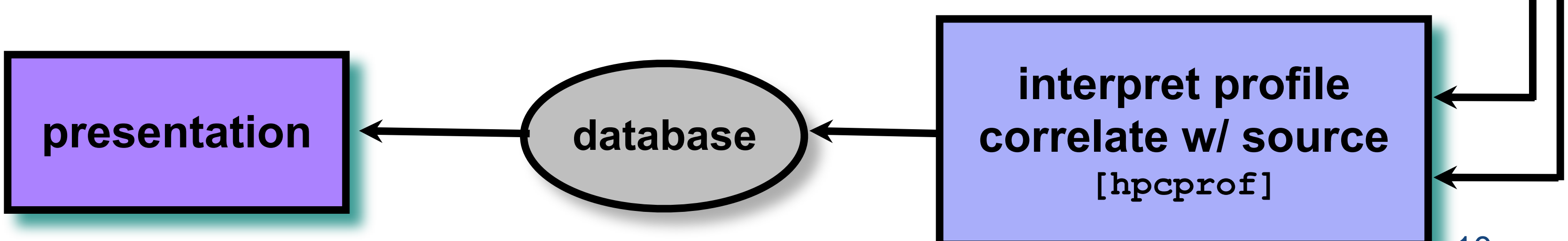


HPCToolkit Performance Tools

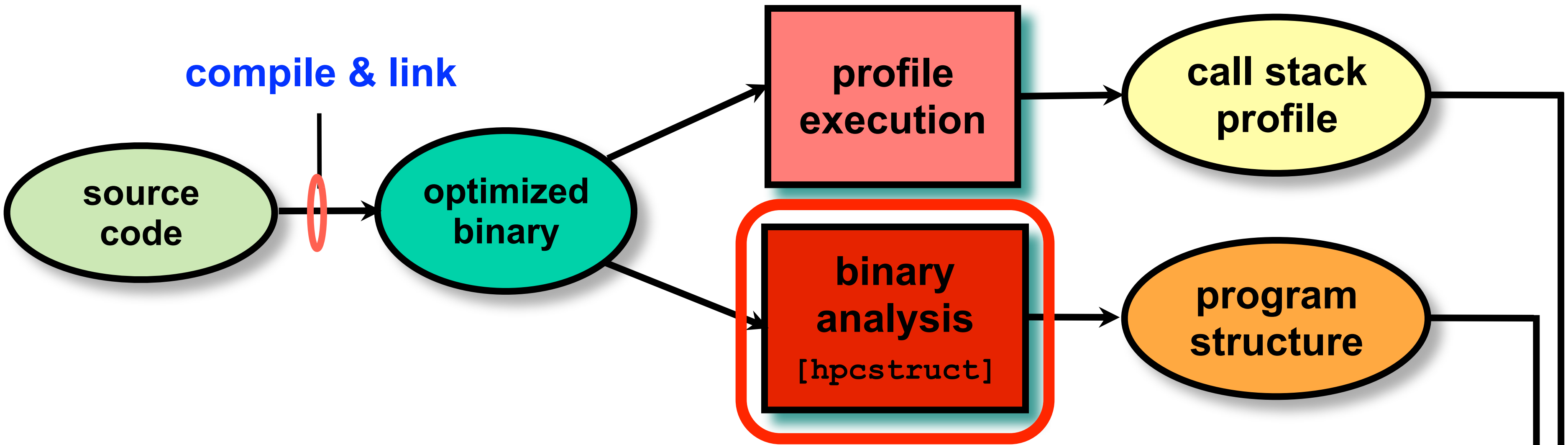


Measure execution unobtrusively

- launch optimized application binaries
- collect call path profiles of events of interest

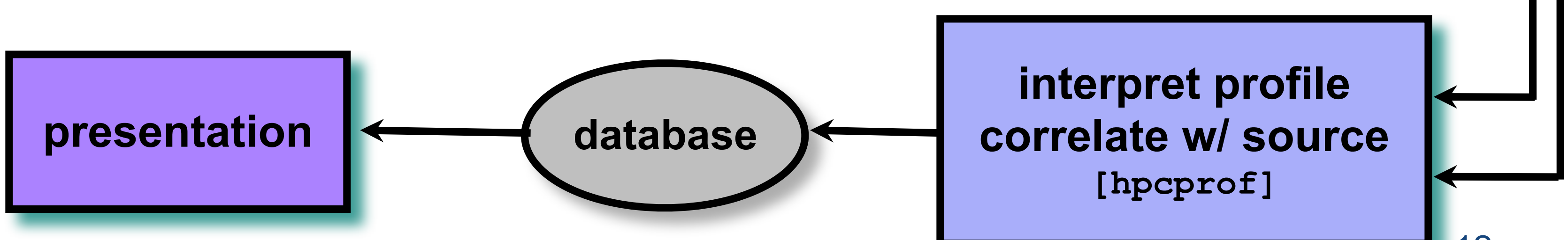


HPCToolkit Performance Tools

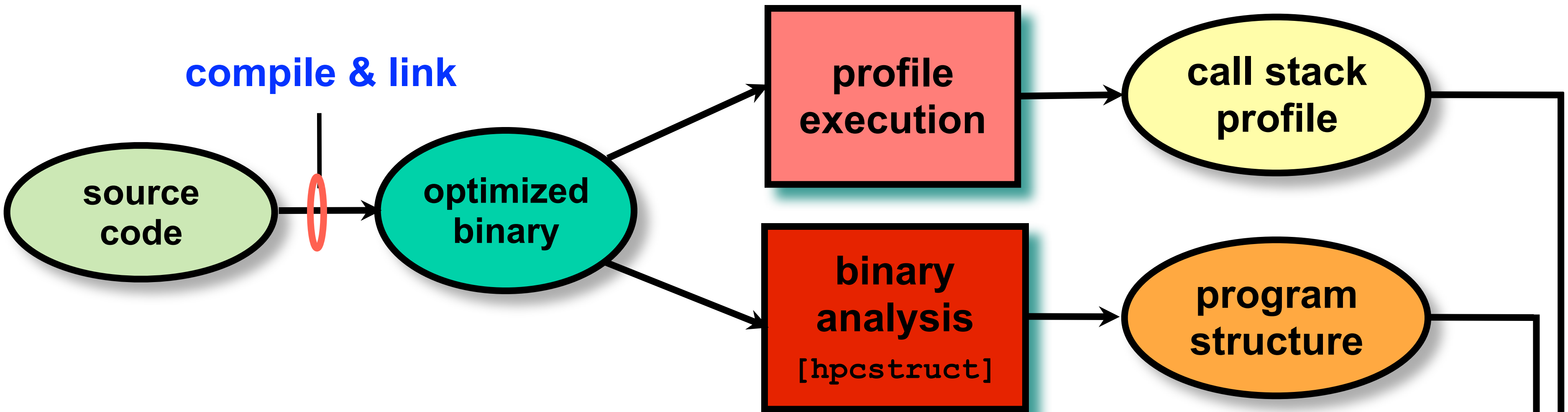


Analyze binary to recover program structure

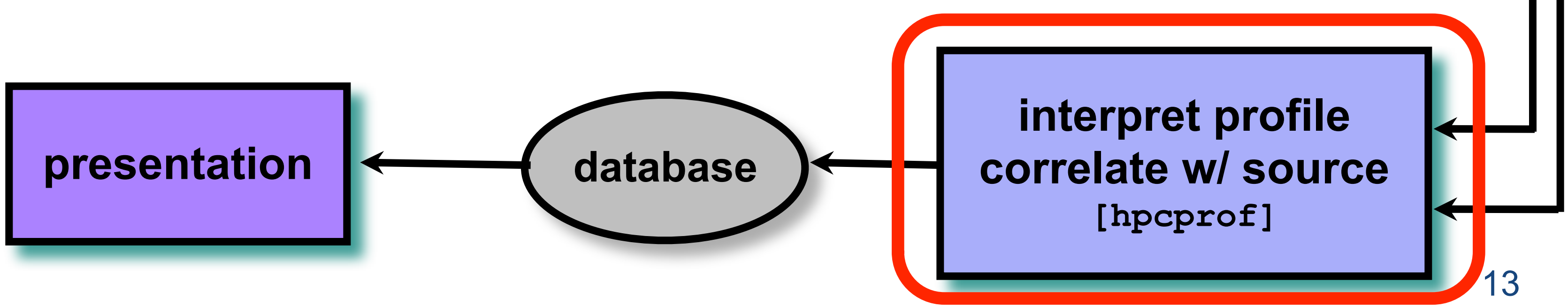
- analyze machine code, line map, and debugging information
- extract loop nesting information and identify inlined procedures
- map transformed loops and procedures back to source



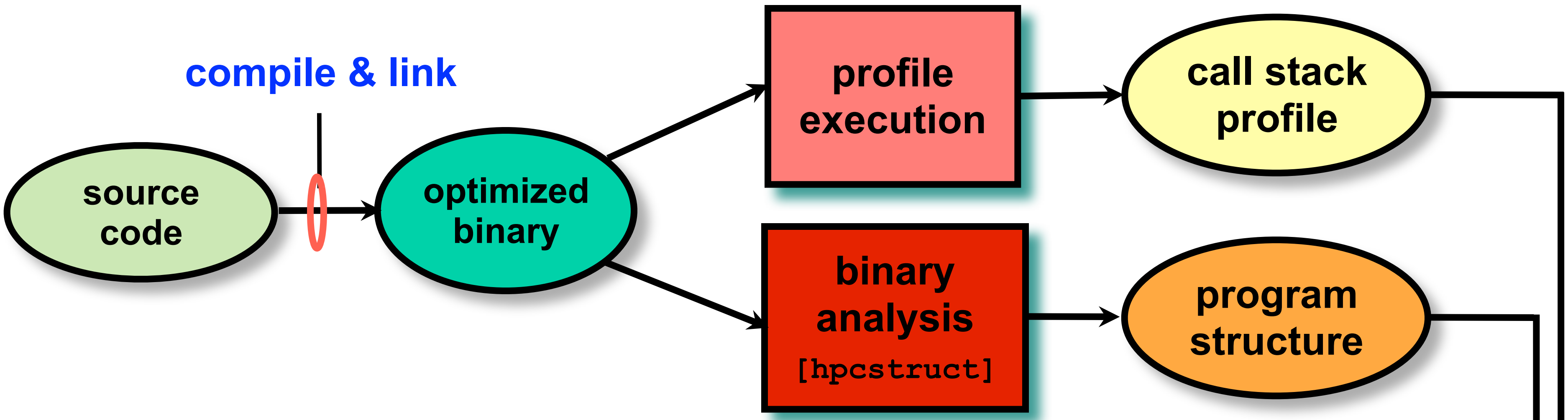
HPCToolkit Performance Tools



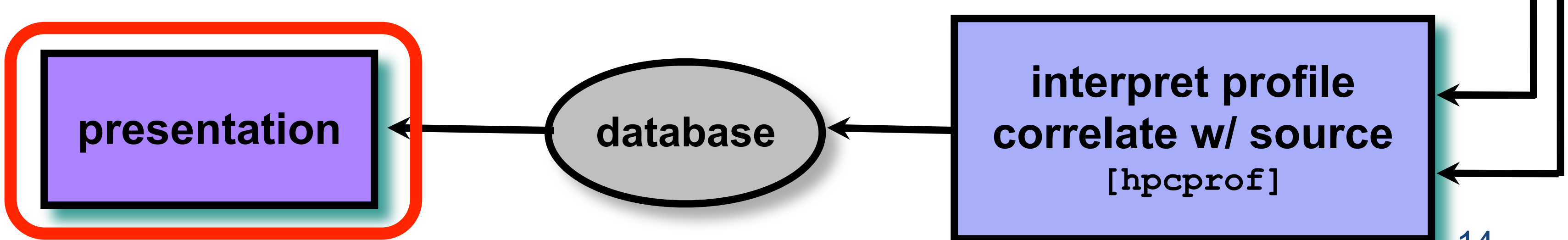
- Combine multiple profiles
  - multiple threads; multiple processes; multiple executions
- Correlate measurements to static & dynamic pgm structure



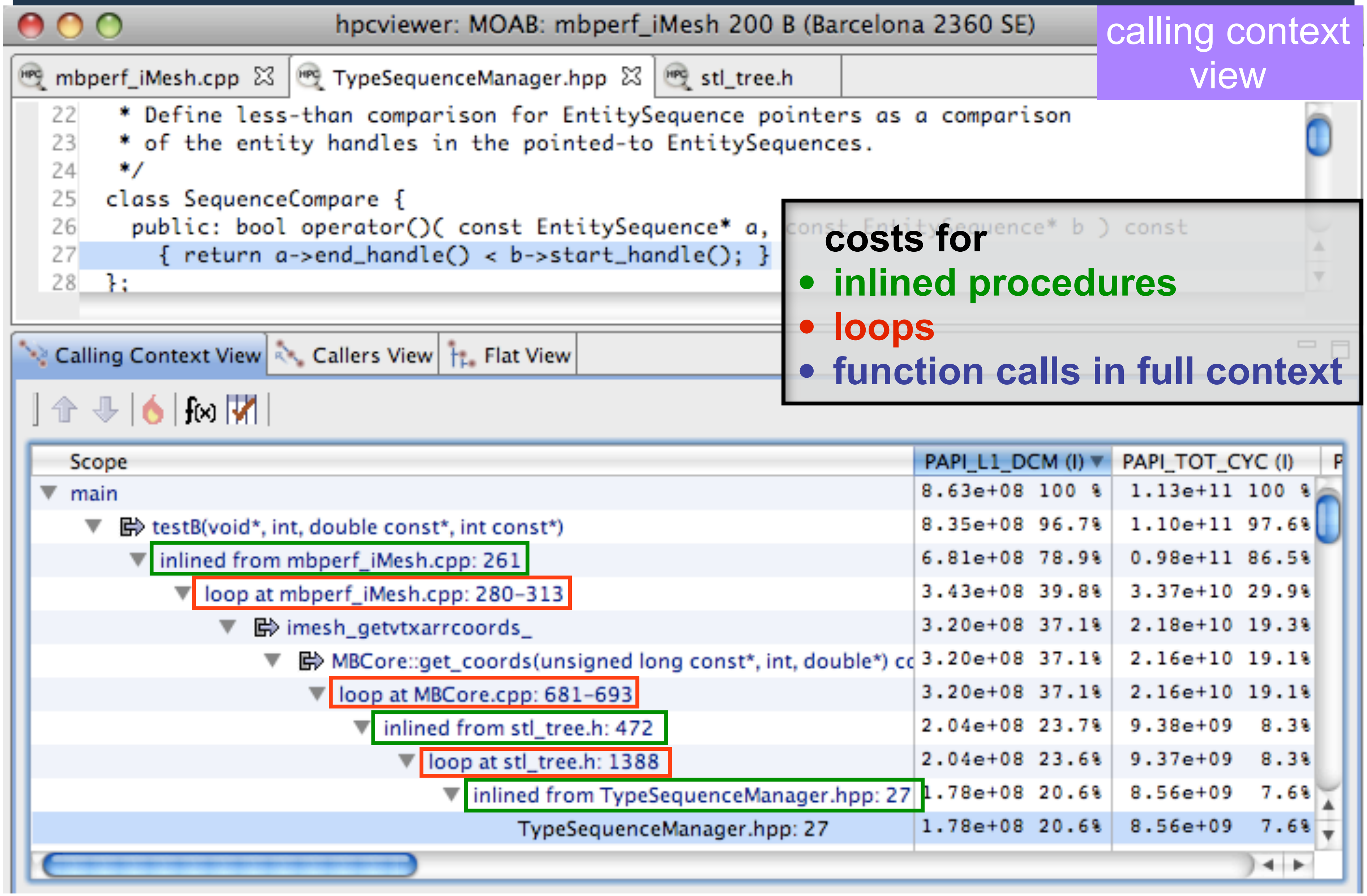
HPCToolkit Performance Tools



- Explore performance data from multiple perspectives
- Rank order by metrics to focus on what's important
- Compute derived metrics to gain insight
- Explore call stack traces to understand transient behavior



Attribution to Static + Dynamic Context



calling context view

costs for
- inlined procedures
- loops
- function calls in full context

Outline

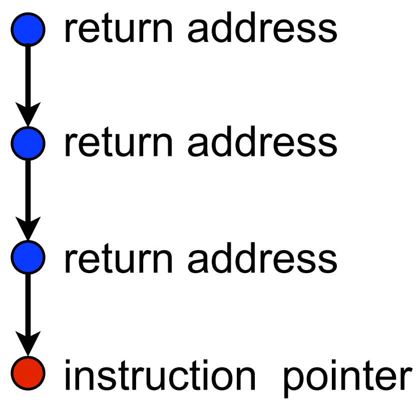
- **Call path profiling in HPCToolkit**
- **Pinpointing and quantifying scalability bottlenecks**
- **Blame shifting**
  - **analyzing multithreaded computations based on work stealing**
  - **quantifying the impact of lock contention on threaded code**
  - **pinpointing load imbalance in parallel codes**
- **Understanding execution behavior over time**
- **Associating memory hierarchy inefficiency with data**
- **Conclusions**
- **Challenges ahead**
- **Related work**

Call Path Profiling

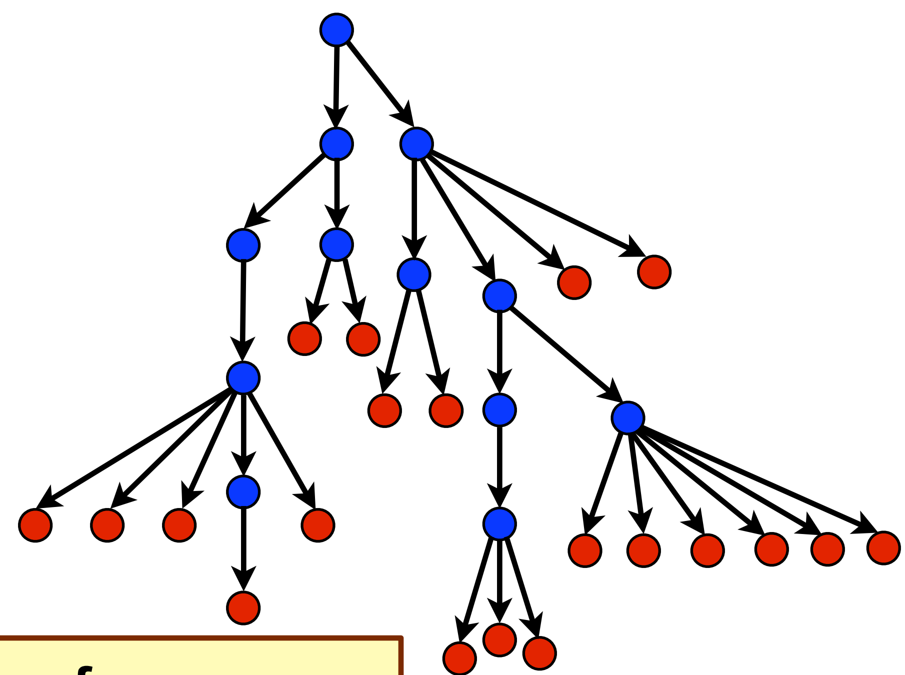
Measure and attribute costs in their *calling* context

- **Sample timer or hardware counter overflows**
- **Gather calling context using stack unwinding**

Call path sample



Calling Context Tree (CCT)



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

Unwinding Fully-optimized Parallel Code

Unwinding using demand-driven binary analysis

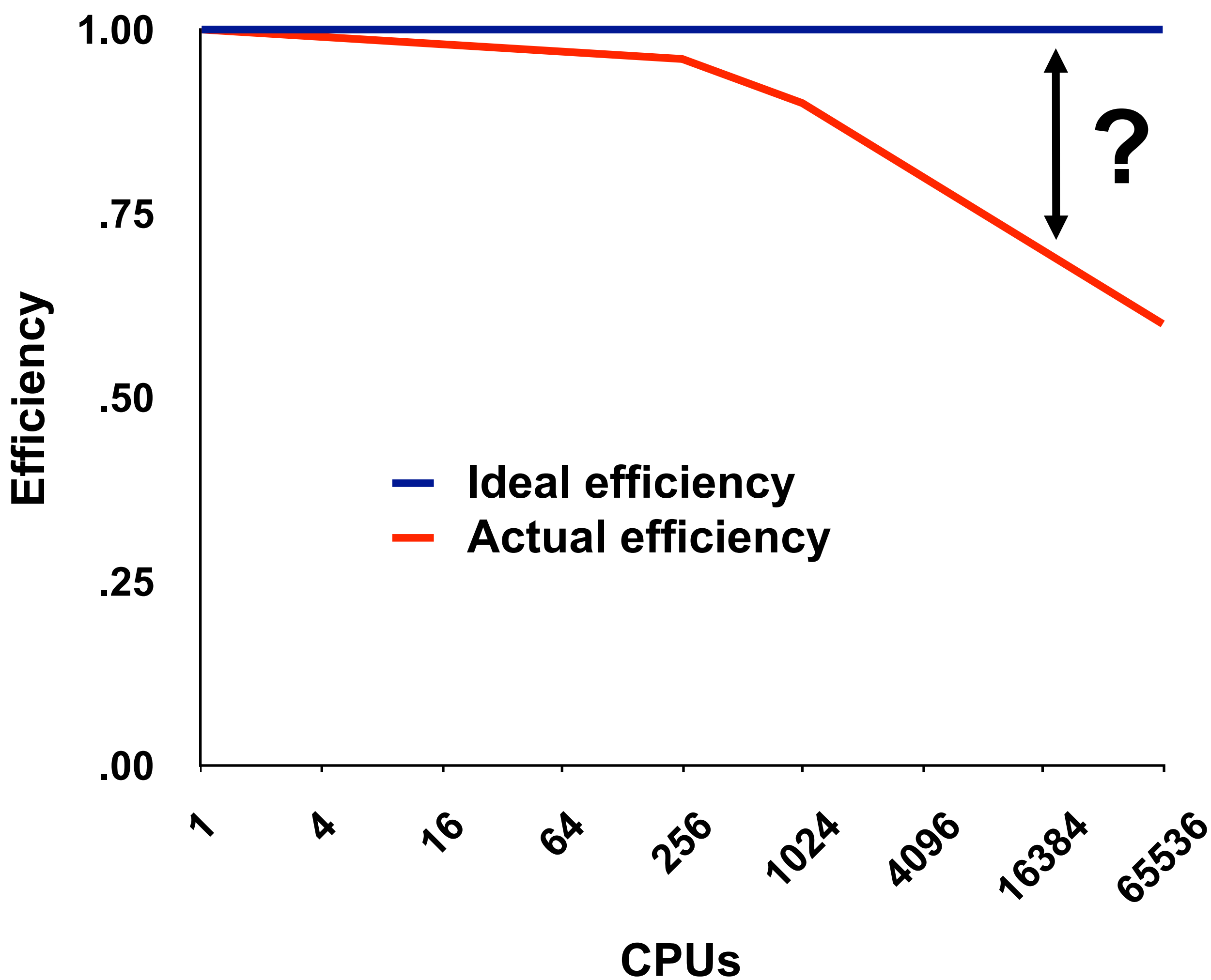
- **Identify procedure bounds**
  - for dynamically-linked code, do this at runtime
  - for statically-linked code, do this at compile time
- **Compute unwind recipes for a procedure on the fly**
  - scan the procedure's object code, tracking the locations of
    - caller's program counter
    - caller's frame and stack pointer
  - create unwind recipes between pairs of frame-relevant instructions
- **Processors: x86-64, PowerPC (BG/P), MIPS (SiCortex)**
- **Results**
  - accurate call path profiles
  - overheads of < 2% for sampling frequencies of 200/s

Nathan Tallent, John Mellor-Crummey, and Michael Fagan. Binary analysis for measurement and attribution of program performance. PLDI 2009, Dublin, Ireland, **Distinguished Paper Award.**

Outline

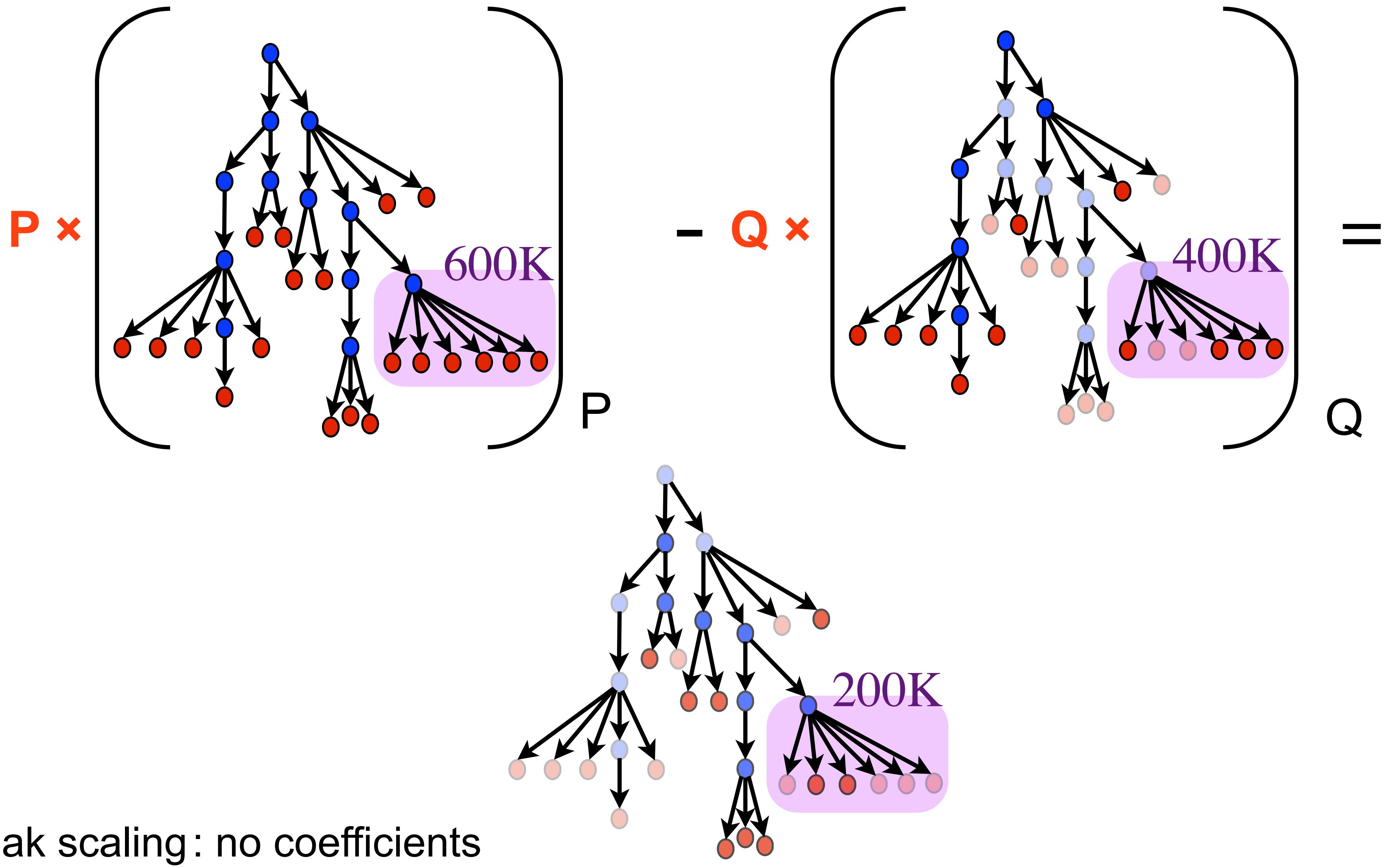
- Call path profiling in HPCToolkit
- Pinpointing and quantifying scalability bottlenecks
- Blame shifting
  - — analyzing multithreaded computations based on work stealing
  - — quantifying the impact of lock contention on threaded code
  - — pinpointing load imbalance in parallel codes
- Understanding execution behavior over time
- Associating memory hierarchy inefficiency with data
- Conclusions
- Challenges ahead
- Related work

The Problem of Scaling Losses



Note: higher is better

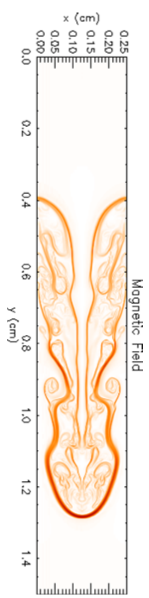
Pinpointing and Quantifying Scalability Bottlenecks



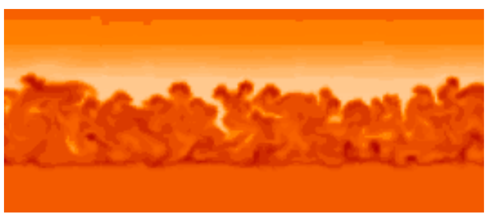
Weak scaling: no coefficients
Strong scaling: needs **red** coefficients

Scalability Analysis of Flash

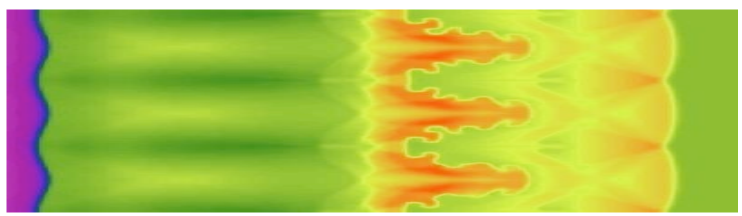
Code:	**University of Chicago FLASH**
Simulation:	**white dwarf detonation**
Platform:	**Blue Gene/P**
Experiment:	**8192 vs. 256 processors**
Scaling type:	**weak**



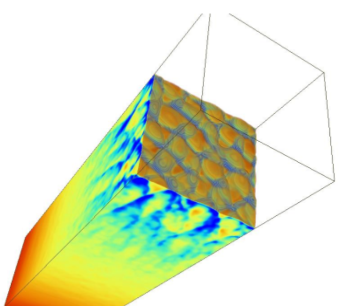
Magnetic Rayleigh-Taylor



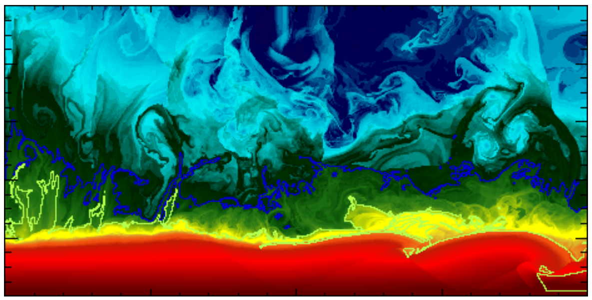
Nova outbursts on white dwarfs



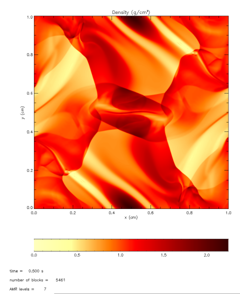
Laser-driven shock instabilities



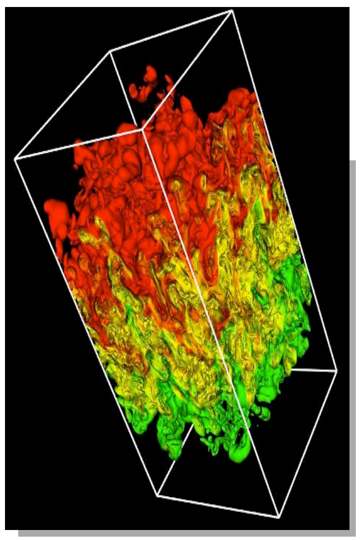
Cellular detonation



Helium burning on neutron stars



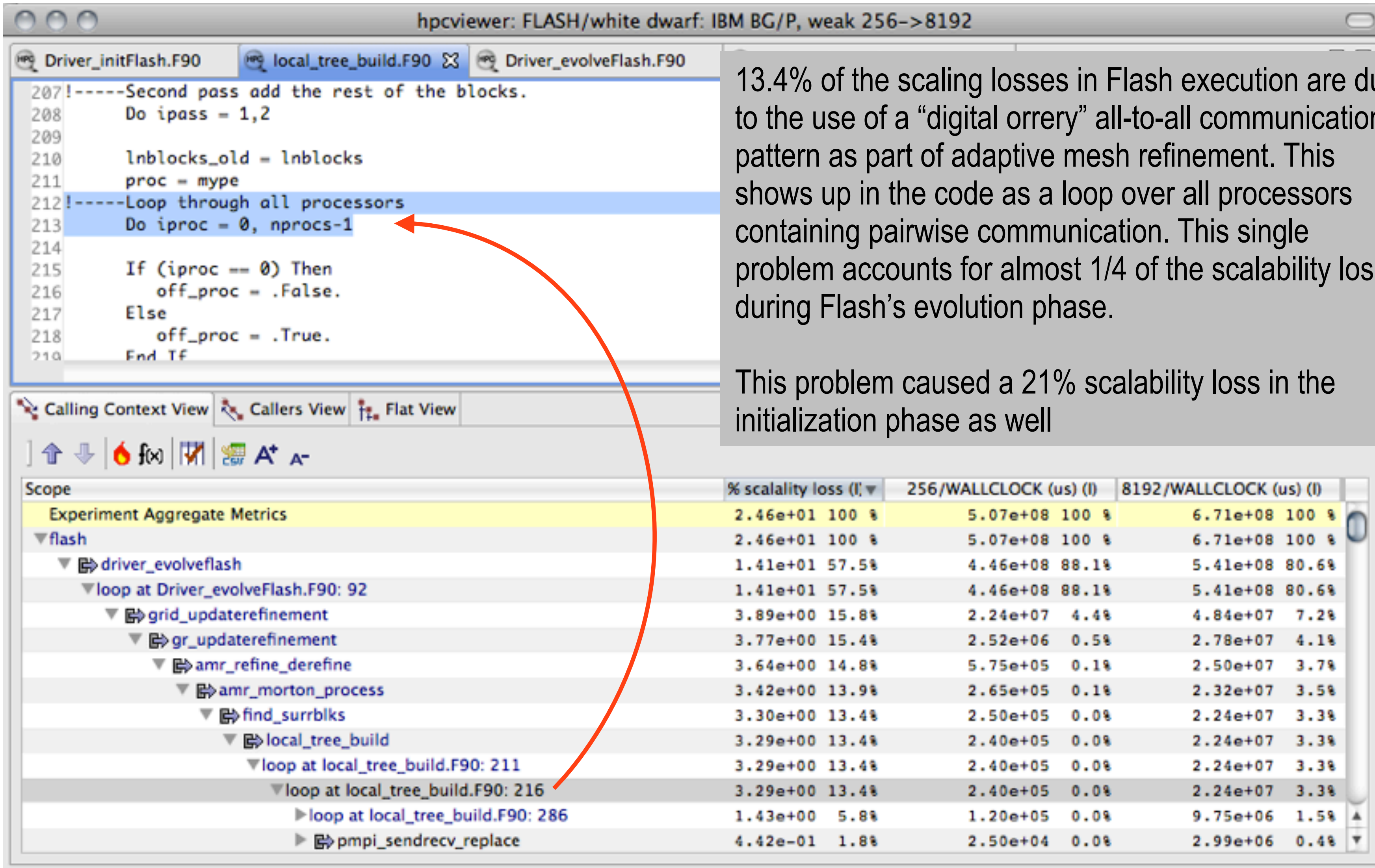
Orzag/Tang MHD vortex



Rayleigh-Taylor instability

Figures courtesy of FLASH Team, University of Chicago

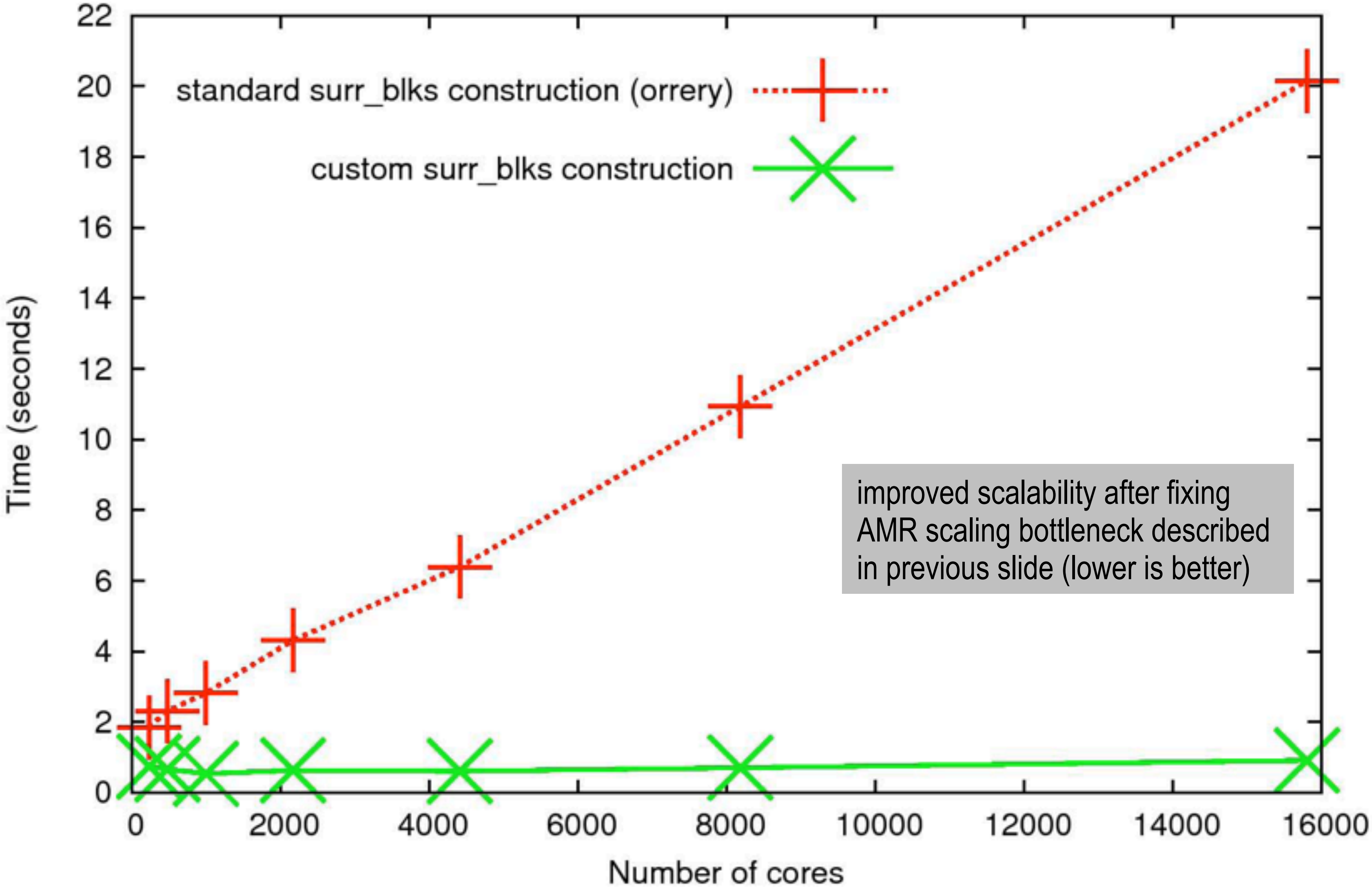
System-wide Scaling Losses in Flash



13.4% of the scaling losses in Flash execution are due to the use of a "digital orrery" all-to-all communication pattern as part of adaptive mesh refinement. This shows up in the code as a loop over all processors containing pairwise communication. This single problem accounts for almost 1/4 of the scalability loss during Flash's evolution phase.

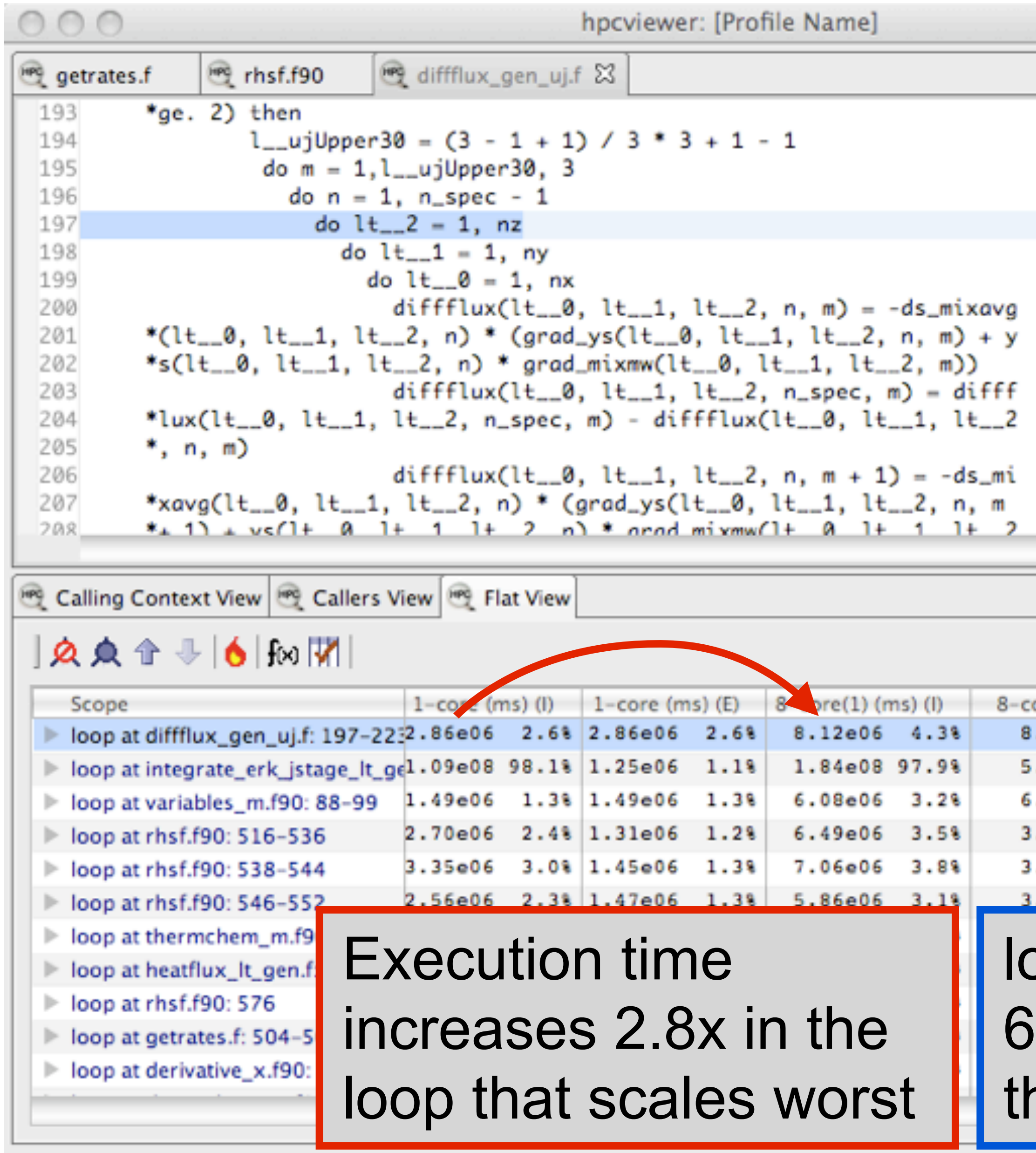
This problem caused a 21% scalability loss in the initialization phase as well

Improved Flash Scaling of AMR Setup

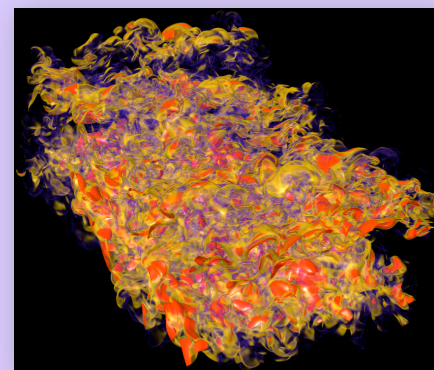


Graph courtesy of Anshu Dubey, U Chicago

Scalability Losses at the Loop Level



S3D code (Sandia CRF)
PI: Jackie Chen
DNS of turbulent combustion



Execution time increases 2.8x in the loop that scales worst

loop contributes a 6.9% scaling loss to the execution

Outline

- Call path profiling in HPCToolkit
- Pinpointing and quantifying scalability bottlenecks
- Blame shifting
  - analyzing multithreaded computations based on work stealing
  - quantifying the impact of lock contention on threaded code
  - pinpointing load imbalance in parallel codes
- Understanding execution behavior over time
- Associating memory hierarchy inefficiency with data
- Conclusions
- Challenges ahead
- Related work

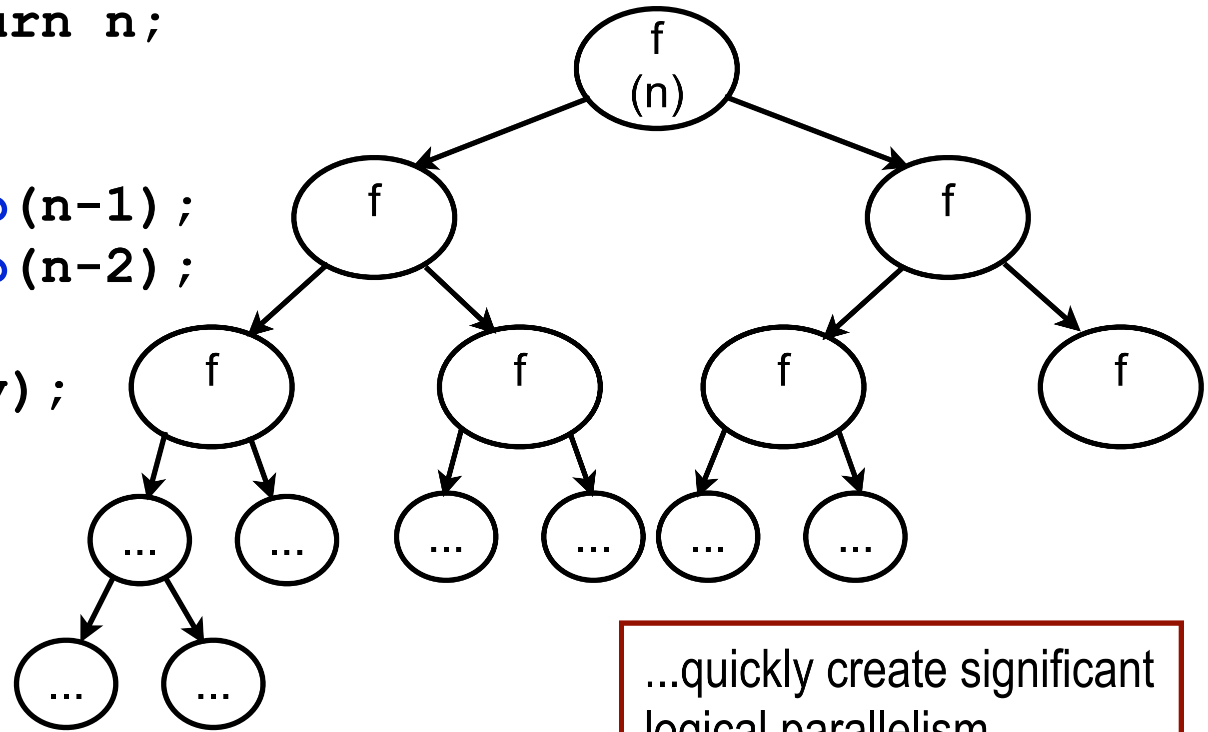
Blame Shifting

- **Problem: in many circumstances sampling measures symptoms of performance losses rather than causes**
  - worker threads waiting for work
  - threads waiting for a lock
  - MPI process waiting for peers in a collective communication
- **Approach: shift blame for losses from victims to perpetrators**
- **Flavors**
  - active measurement
  - analysis only

Cilk: A Multithreaded Language

```
cilk int fib(n) {
  if (n < 2) return n;
  else {
    int x, y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x + y);
  }
}
```

asynchronous calls create logical tasks that only block at a **sync**...



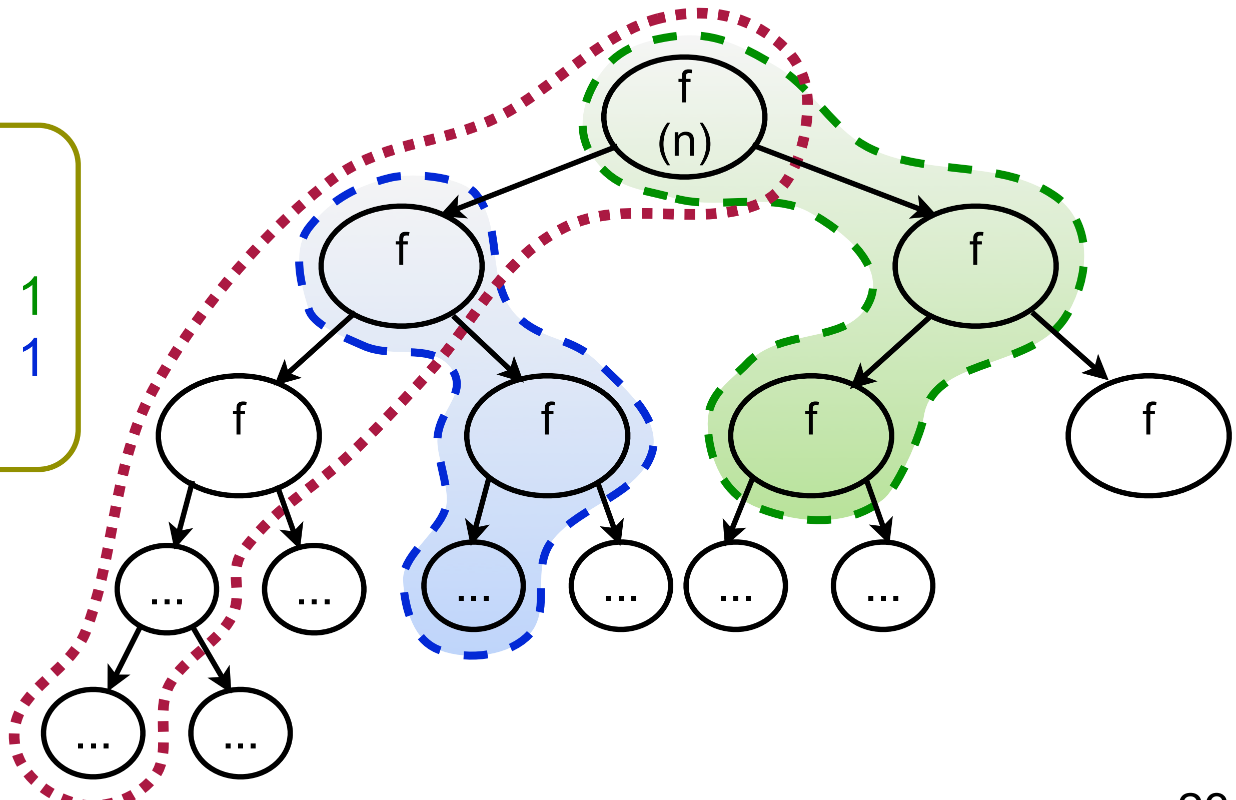
...quickly create significant logical parallelism.

Cilk Program Execution using Work Stealing

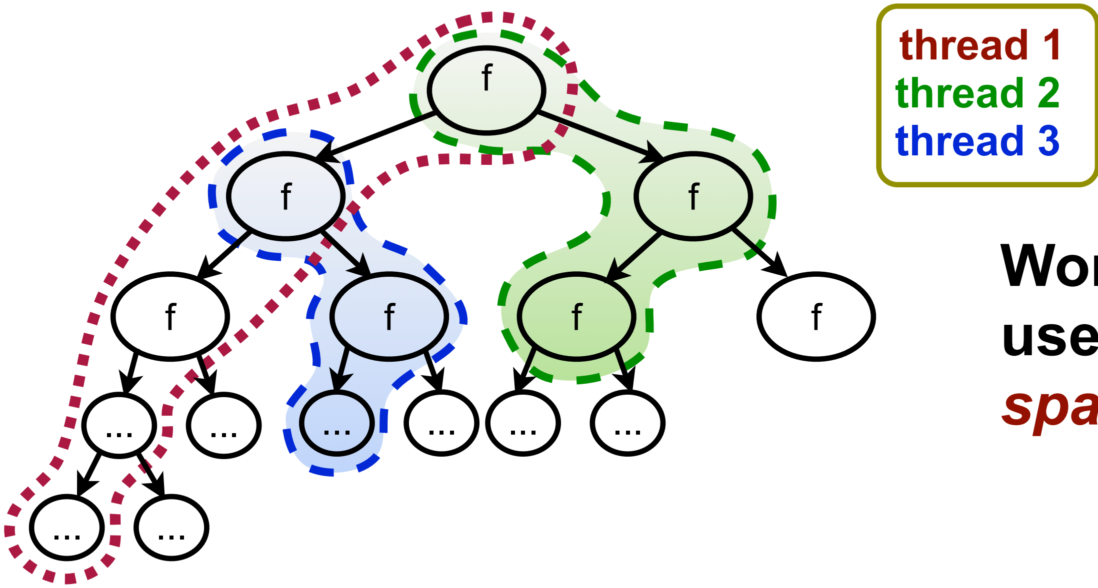
- **Challenge: Mapping logical tasks to compute cores**
- **Cilk approach:**
  - **— lazy thread creation plus work-stealing scheduler**
    - **`spawn`: a potentially parallel task is available**
    - **an idle thread steals tasks from a random working thread**

Possible Execution:

thread 1 begins
thread 2 steals from 1
thread 3 steals from 1
etc**...**

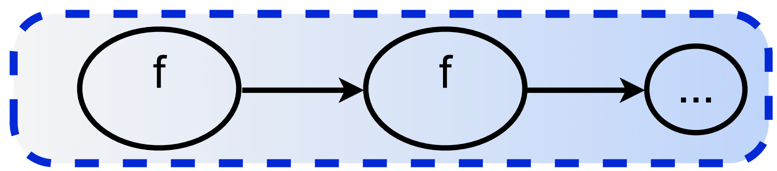


Wanted: Call Path Profiles of Cilk

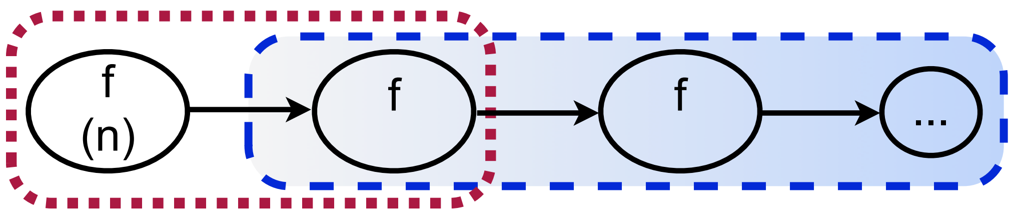


Work stealing *separates* user-level calling contexts in *space and time*

- **Consider thread 3:**
  - **physical call path:**



  - **logical call path:**



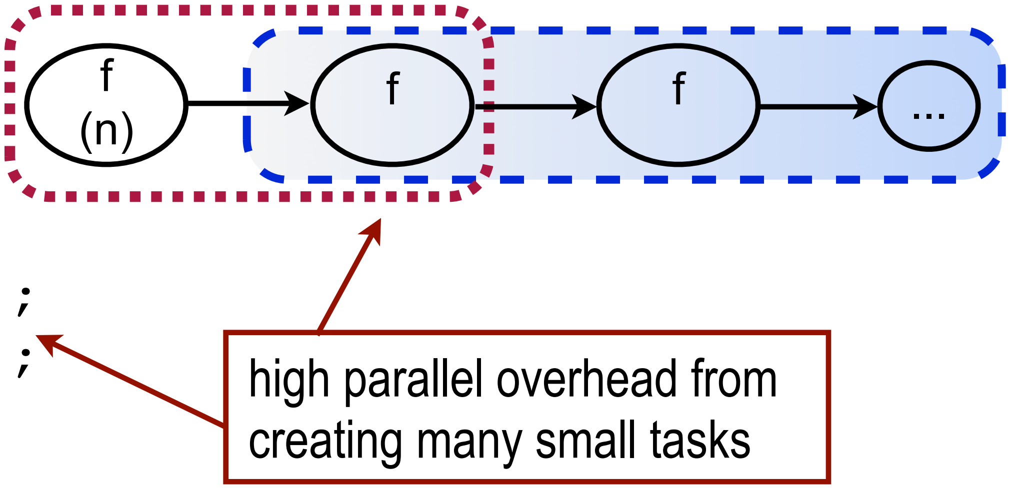
Logical call path profiling: Recover *full* relationship between *physical* and *user-level* execution

Effective Performance Analysis

Three Complementary Techniques:

- Recover ***logical calling contexts*** in presence of work-stealing

```
cilk int fib(n) {
  if (n < 2) {...}
  else {
    int x, y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x + y);
```



- Quantify ***parallel idleness*** (insufficient parallelism)
- Quantify ***parallel overhead***
- Attribute ***idleness*** and ***overhead*** to ***logical contexts***
  — at the source level

Measuring & Attributing Parallel Idleness

- **Metrics: Effort = "work" + "idleness"**
  - — **associate metrics with user-level calling contexts**
  - — **insight: attribute idleness to its cause: context of *working* thread**
    - **a thread looks past itself when 'bad things' happen to <u>others</u>**
- **Work stealing-scheduler: one thread per core**
  - — **maintain W (# working threads) and I (# idling threads)**
    - **slight modifications to work-stealing run time**
      - **atomically incr/decr W when thread exits/enters scheduler**
    - **when a sample event interrupts a <u>working</u> thread**
      - **I = #cores − W**
      - **apportion <u>others'</u> idleness to <u>me</u>: I / W**
- **Example: Dual quad-cores; on a sample, 5 are working:**

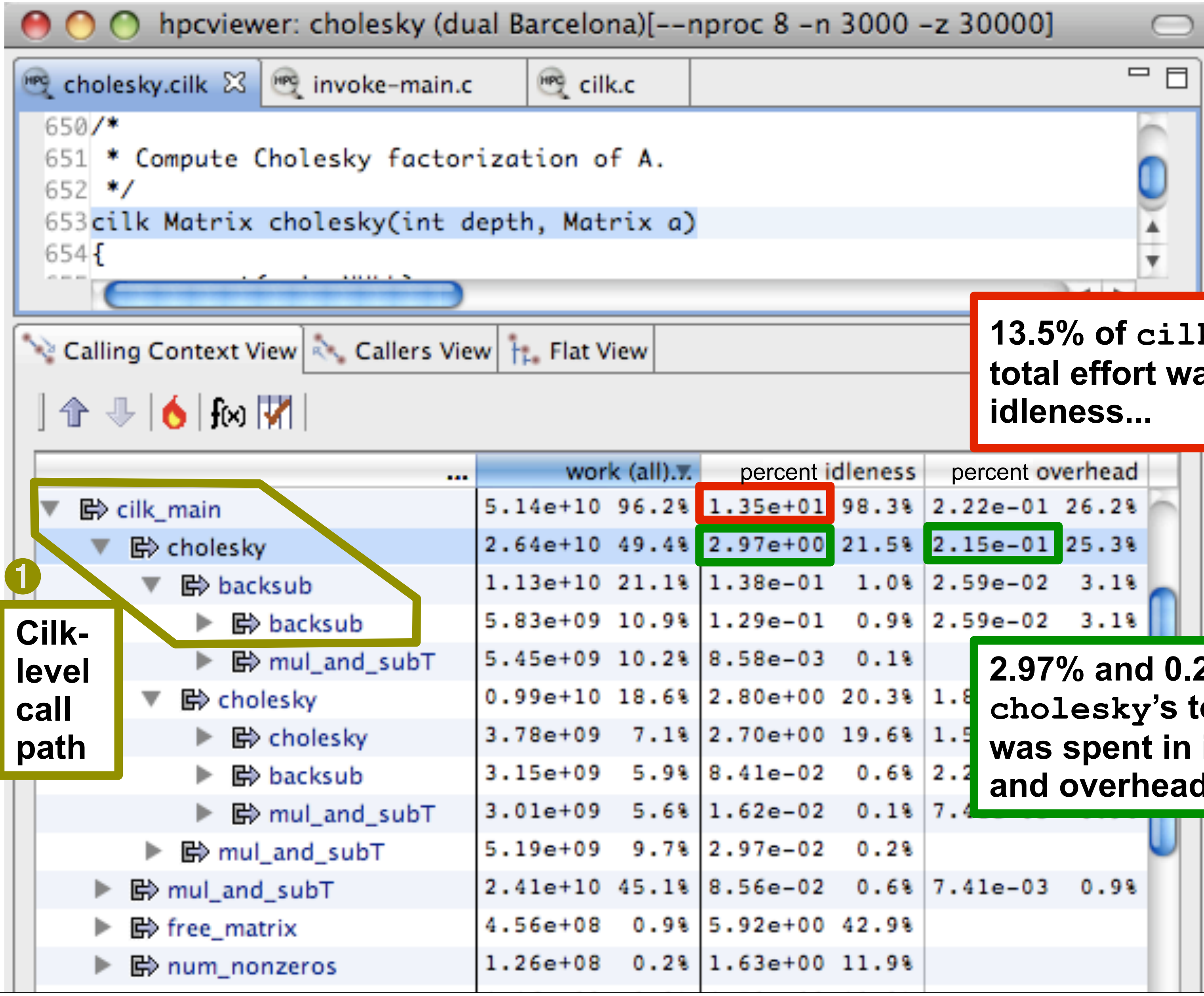
for each worker: $\mathcal{W} \mathrel{+}= 1$, $\mathcal{I} \mathrel{+}= 3/5$ $\qquad$ $\sum \mathcal{W} = 5$, $\sum \mathcal{I} = 3$

idle: drop sample (it's in the scheduler!)

Parallel Overhead

- **Parallel overhead**
  - — when a thread works on something other than user code
    - (we classify waiting for work as idleness)
- **Pinpointing overhead with call path profiling**
  - — impossible, without prior arrangement
    - work and overhead are both machine instructions
  - — insight: have compiler tag instructions as overhead
  - — quantify samples attributed to instructions that represent ovhd
    - use post-mortem analysis

Top-down Work for Cilk 'Cholesky'



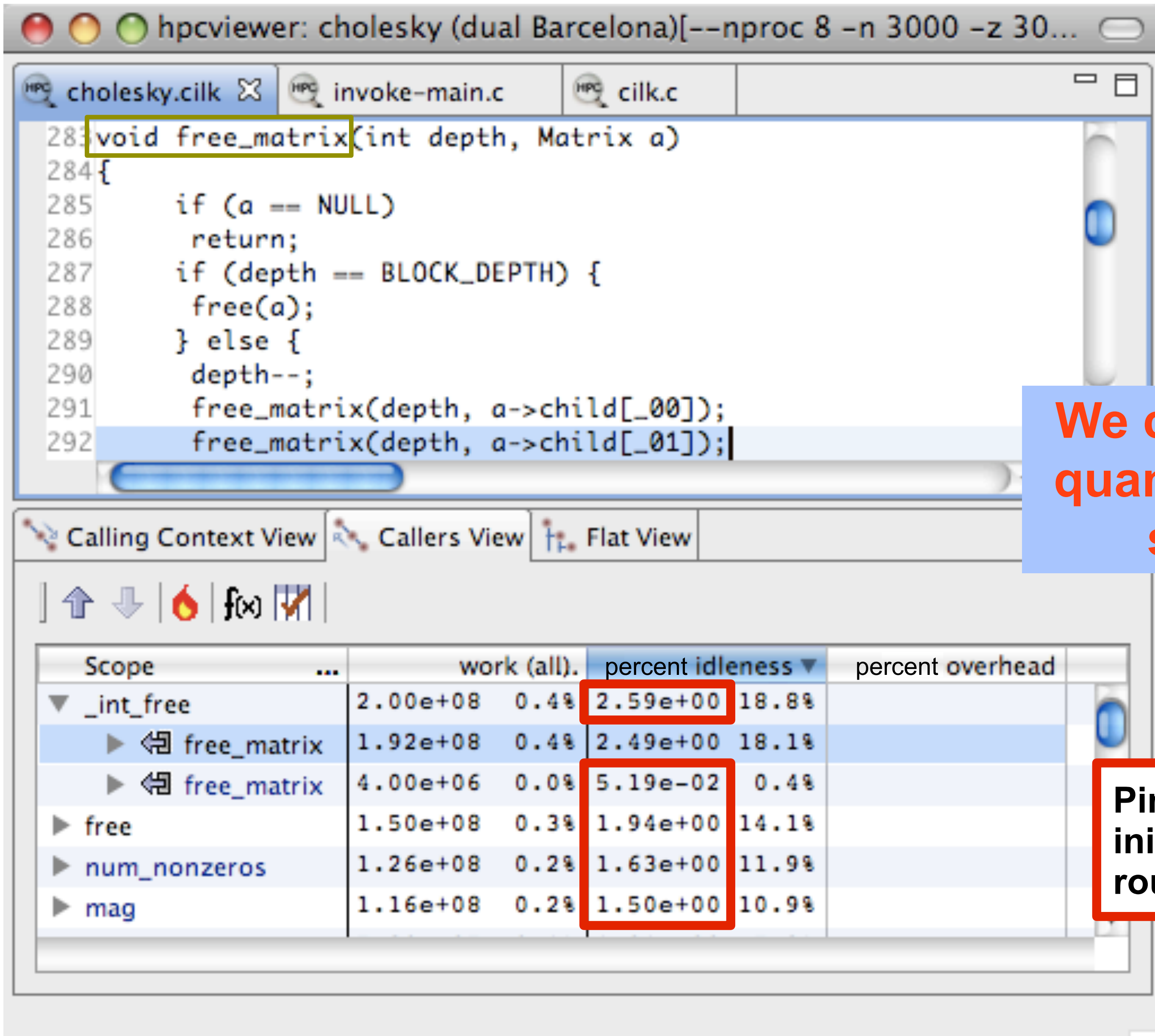
...	work (all)		percent idleness		percent overhead	
cilk_main	5.14e+10	96.2%	1.35e+01	98.3%	2.22e-01	26.2%
cholesky	2.64e+10	49.4%	2.97e+00	21.5%	2.15e-01	25.3%
backsub	1.13e+10	21.1%	1.38e-01	1.0%	2.59e-02	3.1%
backsub	5.83e+09	10.9%	1.29e-01	0.9%	2.59e-02	3.1%
mul_and_subT	5.45e+09	10.2%	8.58e-03	0.1%		
cholesky	0.99e+10	18.6%	2.80e+00	20.3%	1.8	
cholesky	3.78e+09	7.1%	2.70e+00	19.6%	1.5	
backsub	3.15e+09	5.9%	8.41e-02	0.6%	2.2	
mul_and_subT	3.01e+09	5.6%	1.62e-02	0.1%	7.4	
mul_and_subT	5.19e+09	9.7%	2.97e-02	0.2%		
mul_and_subT	2.41e+10	45.1%	8.56e-02	0.6%	7.41e-03	0.9%
free_matrix	4.56e+08	0.9%	5.92e+00	42.9%		
num_nonzeros	1.26e+08	0.2%	1.63e+00	11.9%		

❶ **Cilk-level call path**

13.5% of `cilk_main`'s total effort was spent in idleness... ❸

2.97% and 0.215% of `cholesky`'s total effort was spent in idleness and overhead. ❷

Bottom-up Idleness for Cilk ‘Cholesky’



We can pinpoint and quantify the effect of serialization.

Pinpoints serial initialization/finalization routines.

Using Parallel Idleness & Overhead

- **Total effort = useful work + idleness + overhead**
- **Enables powerful and precise interpretations**

idleness	overhead	interpretation
low	low	effectively parallel
low	high	coarsen concurrency granularity
high	low	refine concurrency granularity
high	high	switch parallelization strategies

- **Normalize w.r.t. total effort to create**
  - **percent idleness or percent overhead**

Nathan Tallent, John Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. PPoPP 2009, Raleigh, NC.

Outline

- Call path profiling in HPCToolkit
- Pinpointing and quantifying scalability bottlenecks
- Blame shifting
  — analyzing multithreaded computations based on work stealing
  — quantifying the impact of lock contention on threaded code
  — pinpointing load imbalance in parallel codes
- Understanding execution behavior over time
- Associating memory hierarchy inefficiency with data
- Conclusions
- Challenges ahead
- Related work

Understanding Lock Contention

- Lock contention causes idleness
  - — explicitly threaded programs (Pthreads, etc)
  - — implicitly threaded programs (critical sections in OpenMP, Cilk...)
- Use “blame-shifting” to shift blame 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 assoc. with 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

*different technique handles blocking

Lock contention in MADNESS

```
    add(MEMFUN_OBJT(memfunT)& obj,
        memfunT memfun,
        const arg1T& arg1, const arg2T& arg2, const arg3T& arg3, const TaskAttributes& a
        Future<REMFUTURE(MEMFUN_RETURNT(memfunT))> result;
        add(new TaskMemfun<memfunT>(result,obj,memfun,arg1,arg2,arg3,attr));
        return result;
    }
```

quantum chemistry; MPI + pthreads

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

- Call path profiling in HPCToolkit
- Pinpointing and quantifying scalability bottlenecks
- Blame shifting
  - — analyzing multithreaded computations based on work stealing
  - — quantifying the impact of lock contention on threaded code
  - — pinpointing load imbalance in parallel codes
- Understanding execution behavior over time
- Associating memory hierarchy inefficiency with data
- Conclusions
- Challenges ahead
- Related work

PFLOTRAN

8K cores, Cray XT5

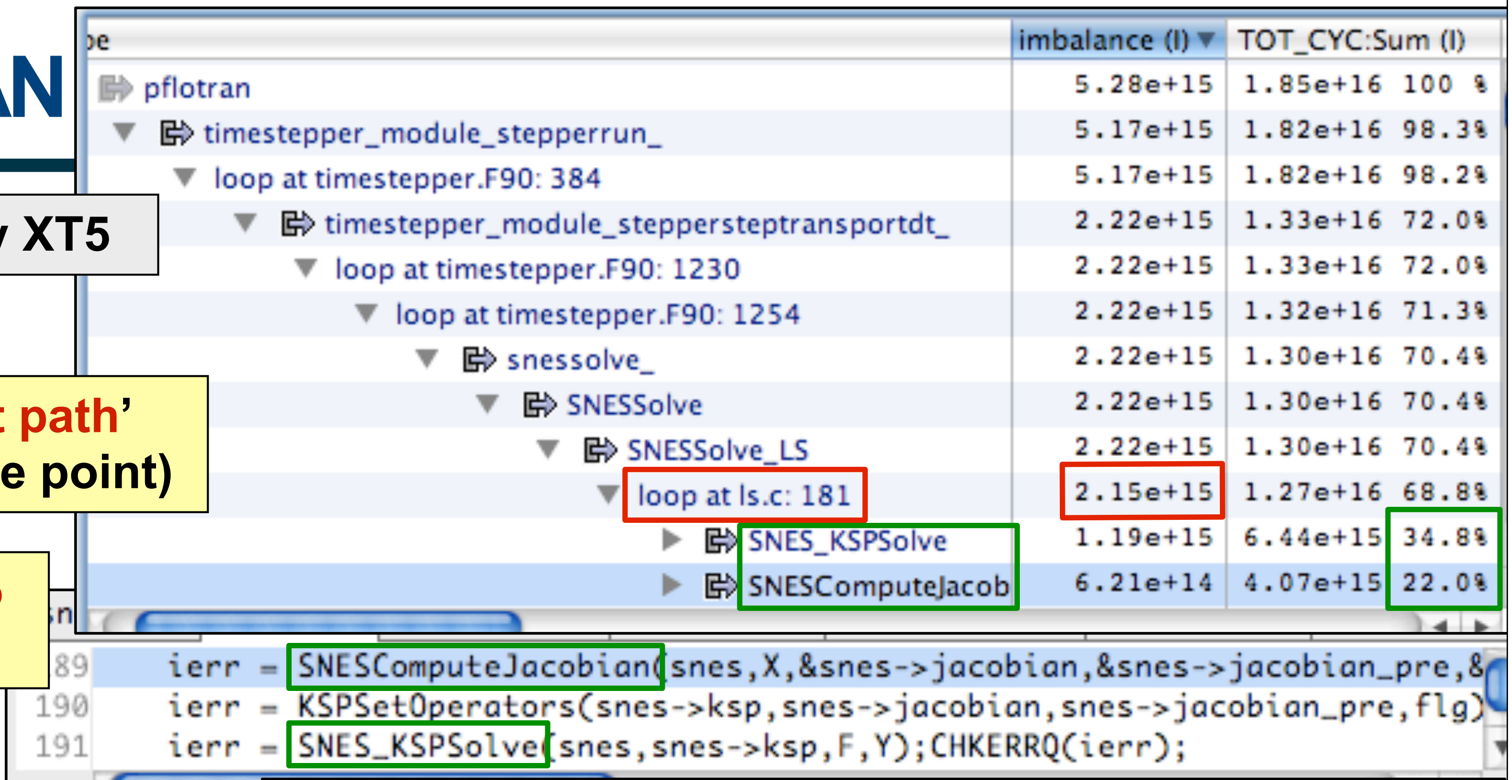
1. Drill down 'hot path' to loop (a balance point)

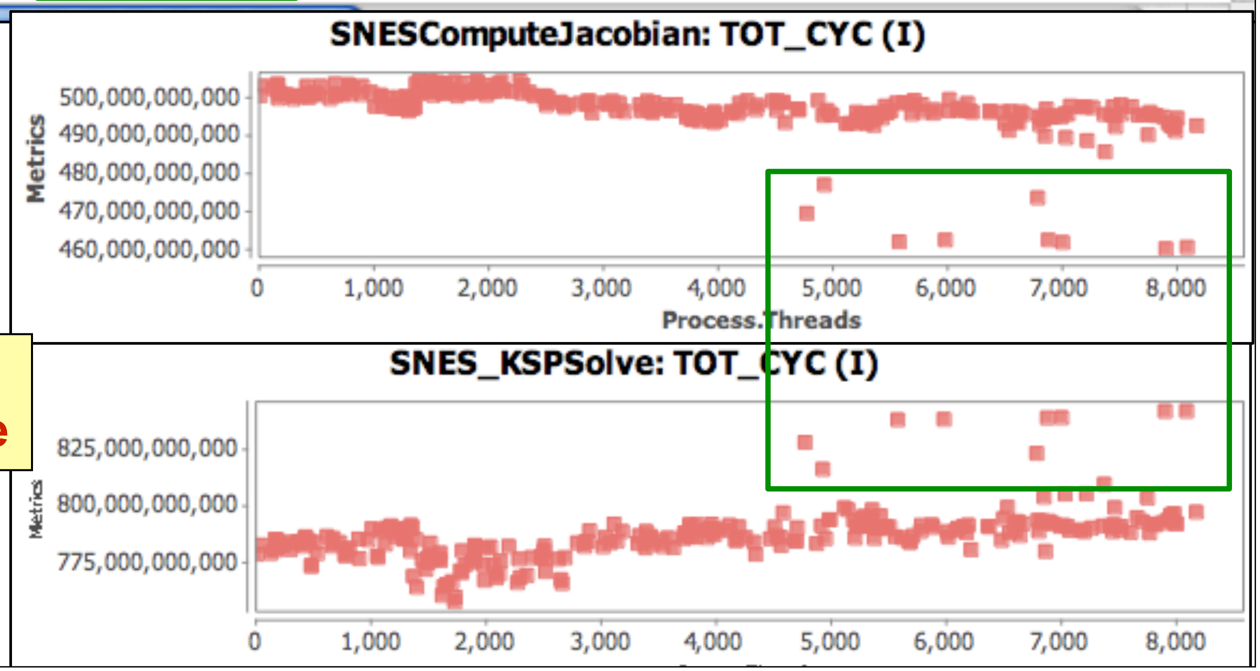
2. Notice top two call sites...

3. Plot the per-process values:

Early finishers...

... become early arrivers at Allreduce





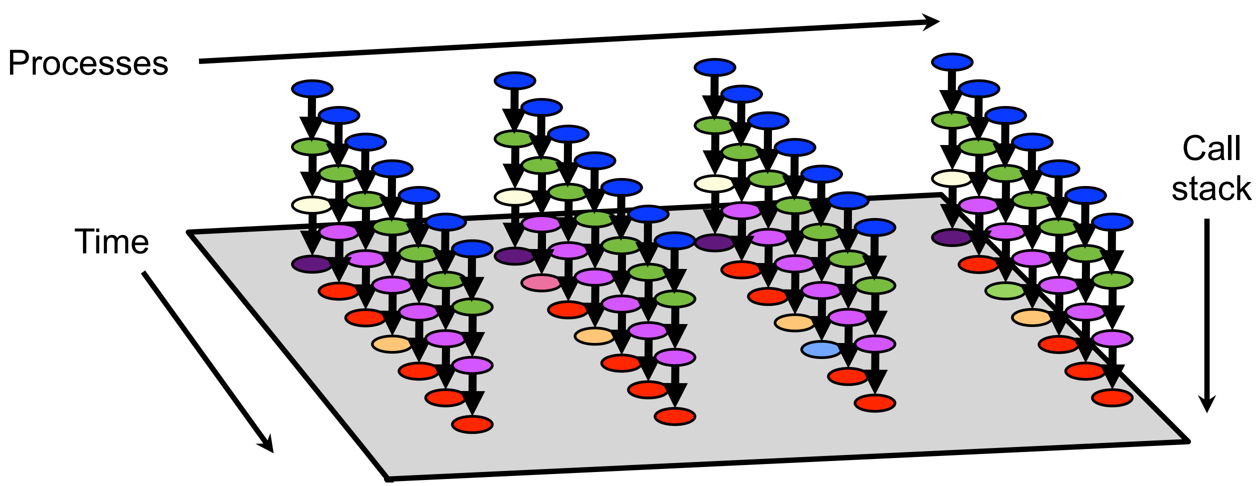
Outline

- Call path profiling in HPCToolkit
- Pinpointing and quantifying scalability bottlenecks
- Blame shifting
  — analyzing multithreaded computations based on work stealing
  — quantifying the impact of lock contention on threaded code
  — pinpointing load imbalance in parallel codes
- Understanding execution behavior over time
- Associating memory hierarchy inefficiency with data
- Conclusions
- Challenges ahead
- Related work

Understanding Temporal Behavior

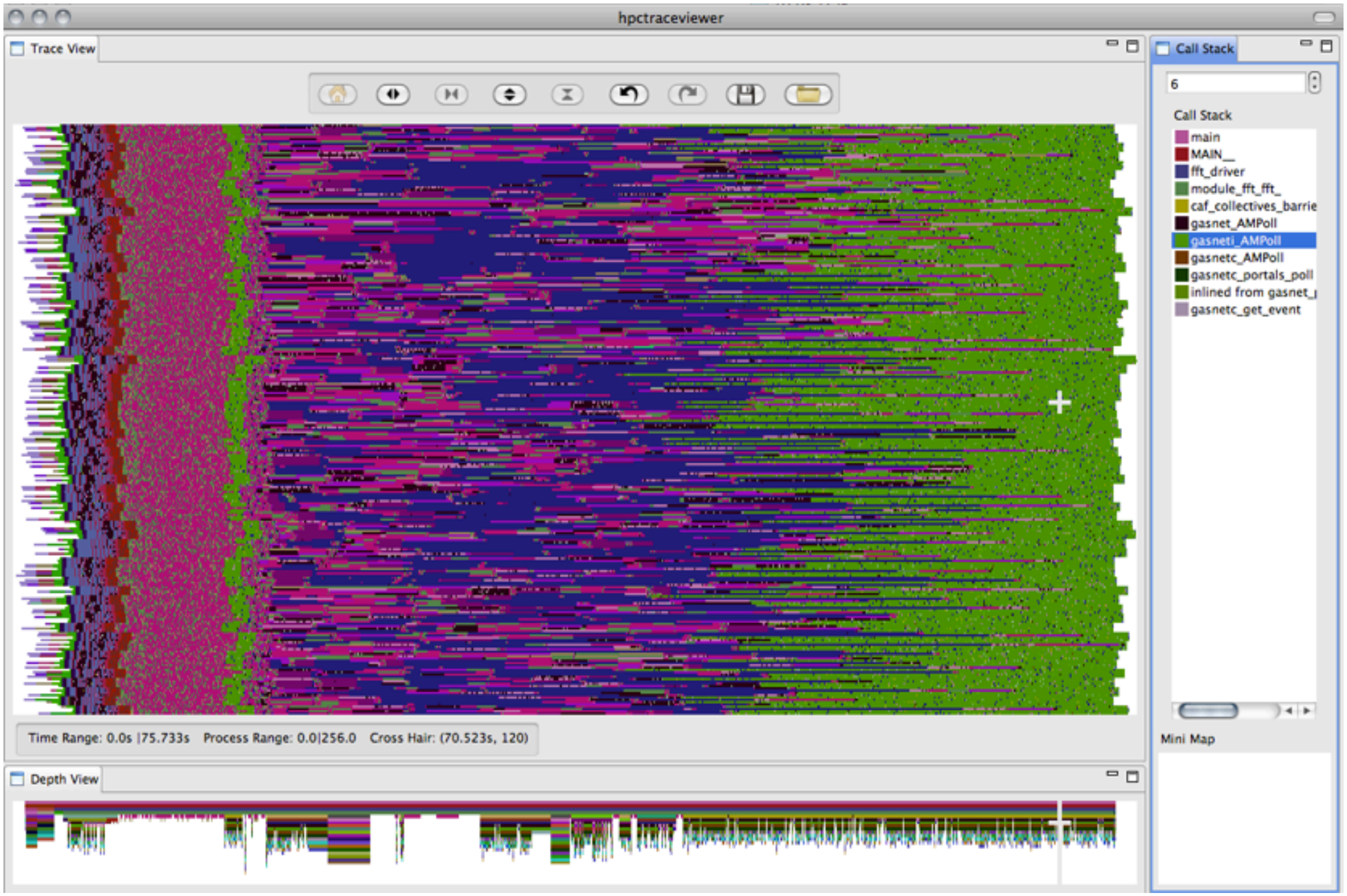
- **Profiling compresses out the temporal dimension**
  - **temporal patterns, e.g. serialization, are invisible in profiles**
- **What can we do? Trace call path samples**
  - **sketch:**
    - **N times per second, take a call path sample of each thread**
    - **organize the samples for each thread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



Call Path Tracing for Parallel Programs

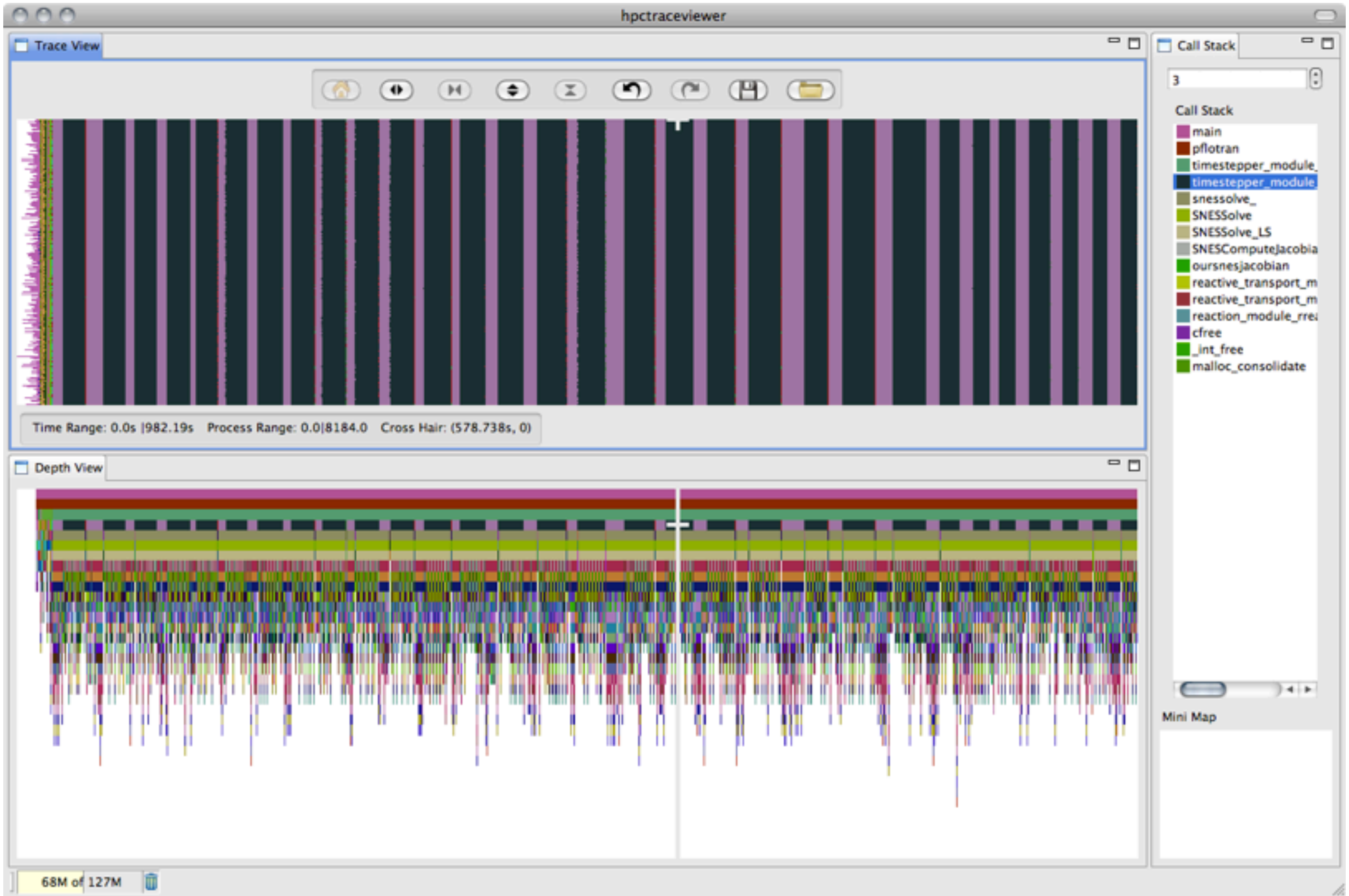
1D FFT, CAF 2.0, 256 processes, Cray XT, 128M/core



Scalable Fine-grained Call Path Tracing, Submitted to IPDPS 2011.

Call Path Tracing for Parallel Programs

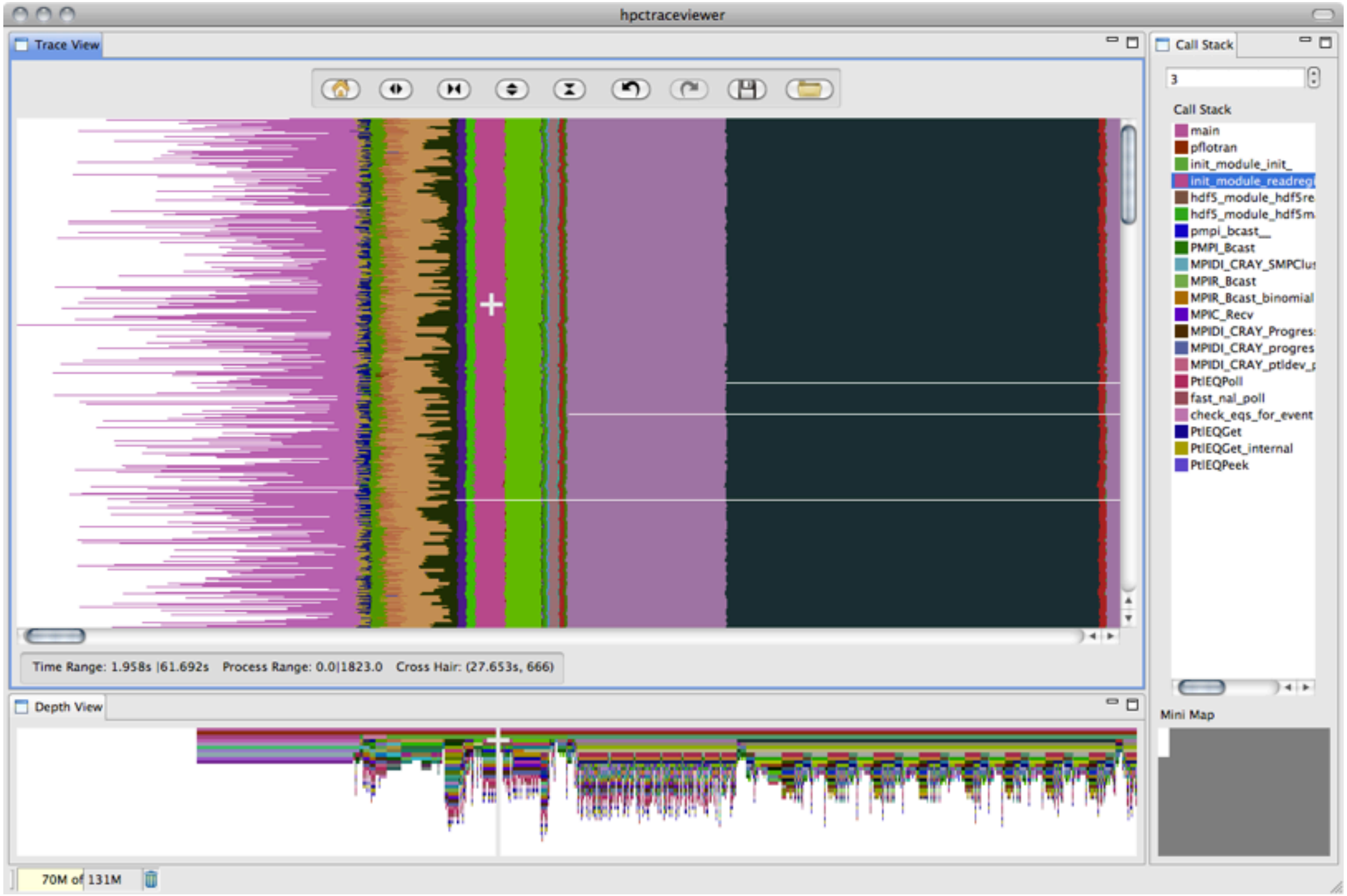
PFLOTRAN: Fortran+MPI, 8184 cores, Cray XT (982s)



Scalable Fine-grained Call Path Tracing, Submitted to IPDPS 2011.

Call Path Tracing for Parallel Programs

PFLOTRAN: Fortran+MPI, 8184 cores, Cray XT (1st minute)



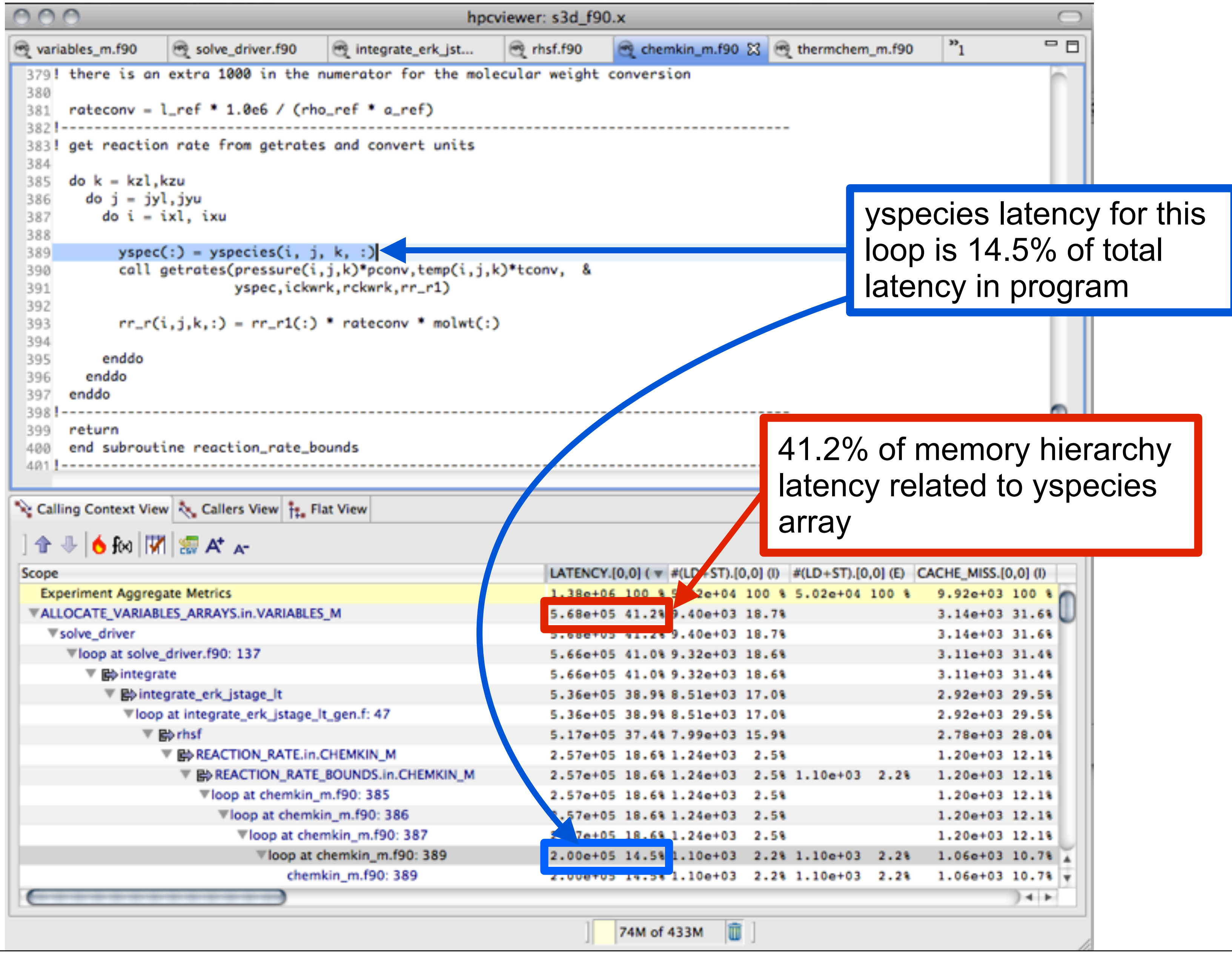
Outline

- Call path profiling in HPCToolkit
- Pinpointing and quantifying scalability bottlenecks
- Blame shifting
  — analyzing multithreaded computations based on work stealing
  — quantifying the impact of lock contention on threaded code
  — pinpointing load imbalance in parallel codes
- Understanding execution behavior over time
- Associating memory hierarchy inefficiency with data
- Conclusions
- Challenges ahead
- Related work

Data Centric Analysis

- Goal: associate memory hierarchy performance losses with data
- Approach
  - — intercept allocations to associate with their data ranges
  - — associate latency with data using "instruction-based sampling" on 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 quantitative results using hpcviewer

Data Centric Analysis of S3D



Conclusions

- **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
  - where insufficient parallelism lurks
  - sources of lock contention
  - load imbalance
  - temporal dynamics
  - problematic data structures

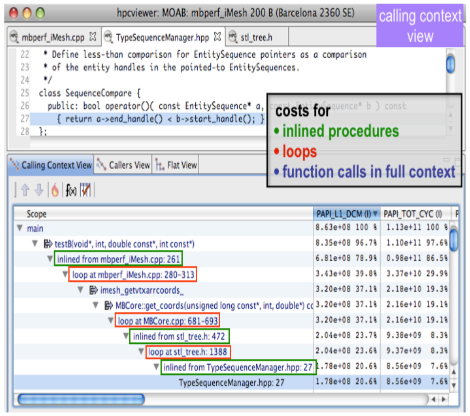
Some Challenges Ahead

- Support characteristics of emerging hardware and software
  - heterogeneous hardware
    - manycore, CPU+GPU
    - dynamic power and frequency scaling
  - software
    - one-sided communication
    - asynchronous operations
    - dynamic parallelism
    - adaptation
    - failure recovery
- Augment monitoring capabilities throughout the stack
  - hardware, OS, runtime, language-level API
- Improve data management for extreme scale parallelism
- Transition from descriptive to prescriptive feedback
- Guide online adaptation and tuning

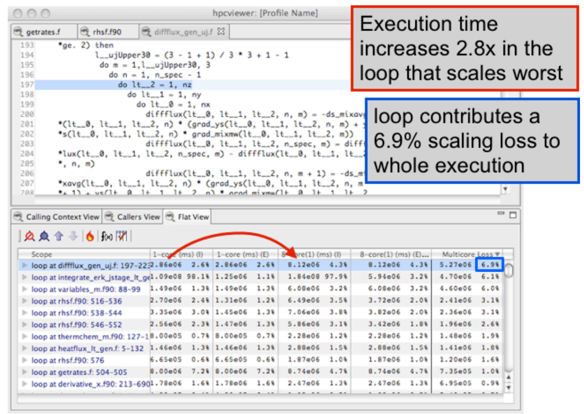
Some Related Work

- **Sampling**
  - — e.g., gprof, Speedshop, Shark, PTU, DCPI, Oprofile, CrayPat
- **Instrumentation**
  - — e.g., Tau, Vtune, IBM HPC Toolkit, Dyninst, CrayPat, Pin
- **Tracing**
  - — e.g., vt, Tau, CEBPA,
- **Call stack profiling**
  - — e.g., mpiP, Tau, PTU, Shark
- **Visualization**
  - — e.g., Paraver, Projections, Vampir, Jumpshot, EXPERT
- **Parallel Analysis**
  - — e.g., Scalasca
- **Analysis**
  - — e.g., IBM HPCS Toolkit, Cray Apprentice, EXPERT, PerfExpert

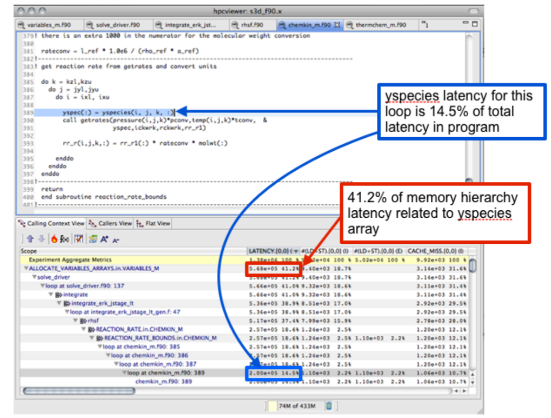
HPCToolkit Capabilities at a Glance



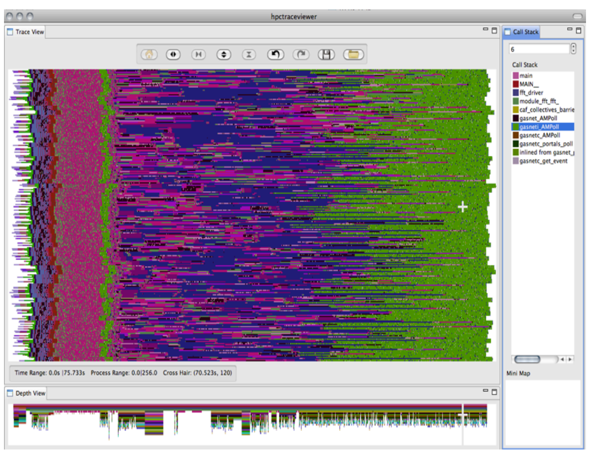
Attribute Costs to Code



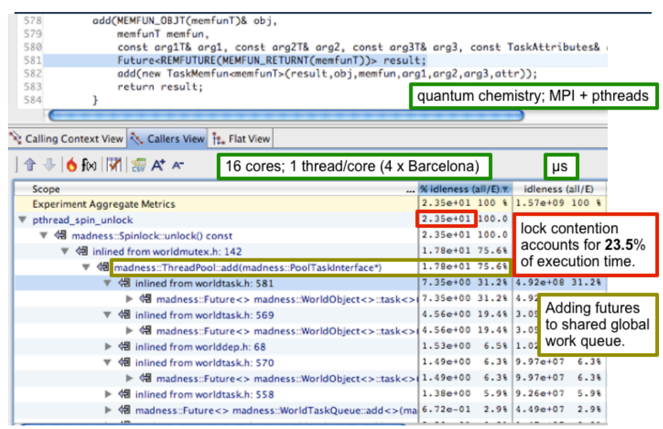
Pinpoint & Quantify Scaling Bottlenecks



Associate Costs with Data



Analyze Behavior over Time



Shift Blame from Symptoms to Causes

Assess Imbalance and Variability

hpctoolkit.org

HPCToolkit Publications

Measurement

- Binary analysis for (1) recovering functions in partially stripped code, (2) unwinding fully-optimized code, (3) recovering program structure
- Nearly perfect call stack sampling of fully optimized code with low overhead

Binary Analysis for Measurement and Attribution of Program Performance, PLDI, June 2009. Distinguished Paper Award.

Pinpointing Locality Problems Using Data-centric Analysis,

Submitted to CGO 2011, April 2011

Pinpoint Scalability Bottlenecks using Differential Profiling

Scalability Analysis of SPMD Codes using Expectations, ICS, June 2007

Pinpoint Performance Losses in Multithreaded Executions

Effective Performance Measurement and Analysis of Multithreaded Applications, PPoPP, February 2009.

Analyzing Lock Contention in Multithreaded Applications,

PPoPP, January 2010

Novel Capabilities of HPCToolkit - II

Performance Analysis using Sampling on Leadership Platforms

Diagnosing Performance Bottlenecks in Emerging Petascale Applications, *SC09, November 2009*

Scalable Identification of Load Imbalance using Call Path Profiles, *SC10, November 2010*

User Interfaces

Effectively Presenting Call Path Profiles of Application Performance, PSTI, September 2010.

Scalable Fine-grained Call Path Tracing, Submitted to IPDPS 2011.

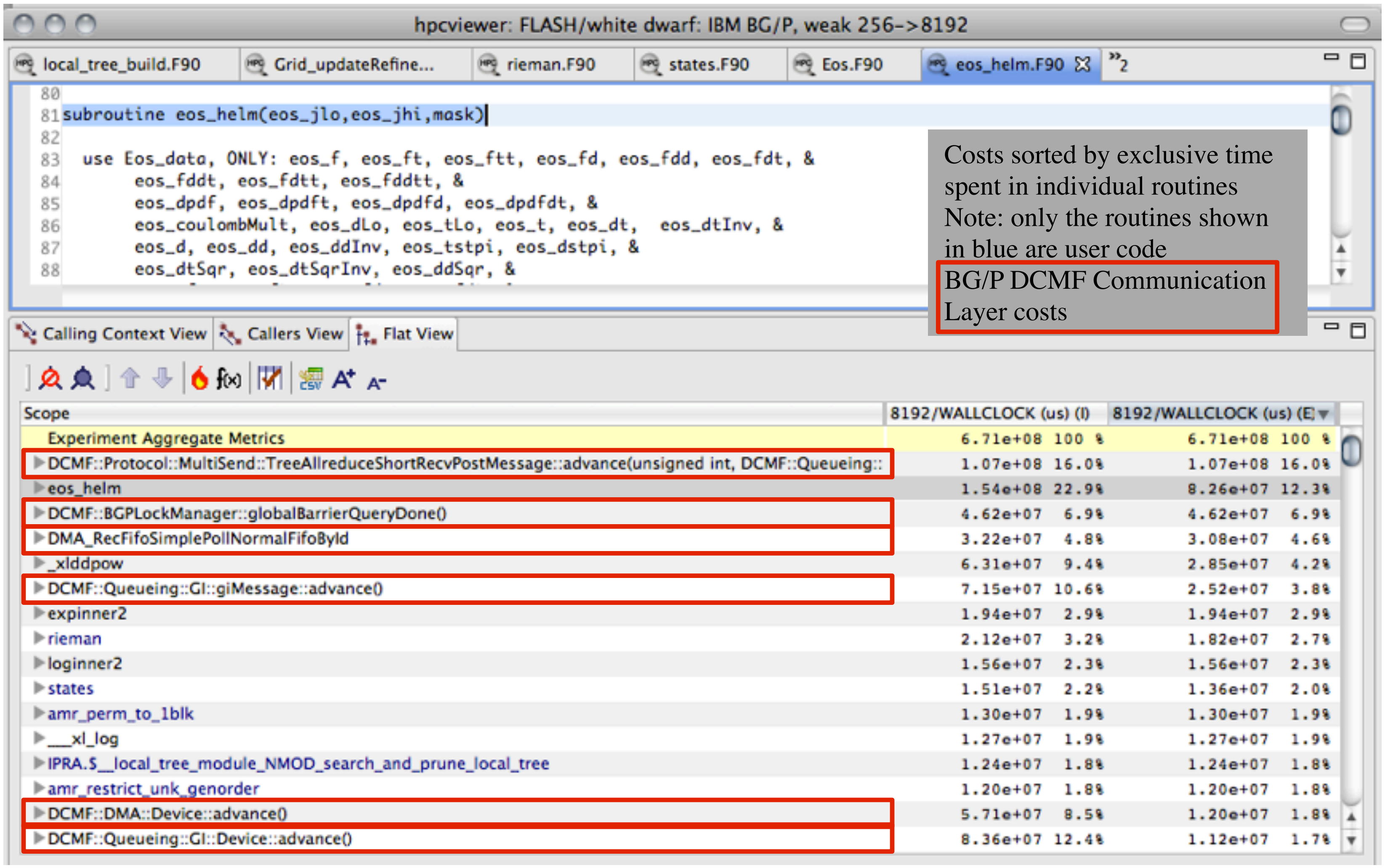
Overview Paper

HPCToolkit: Tools for performance analysis of optimized parallel programs, Concurrency & Computation: Practice and Experience, January 2010

Additional Tool Screenshots

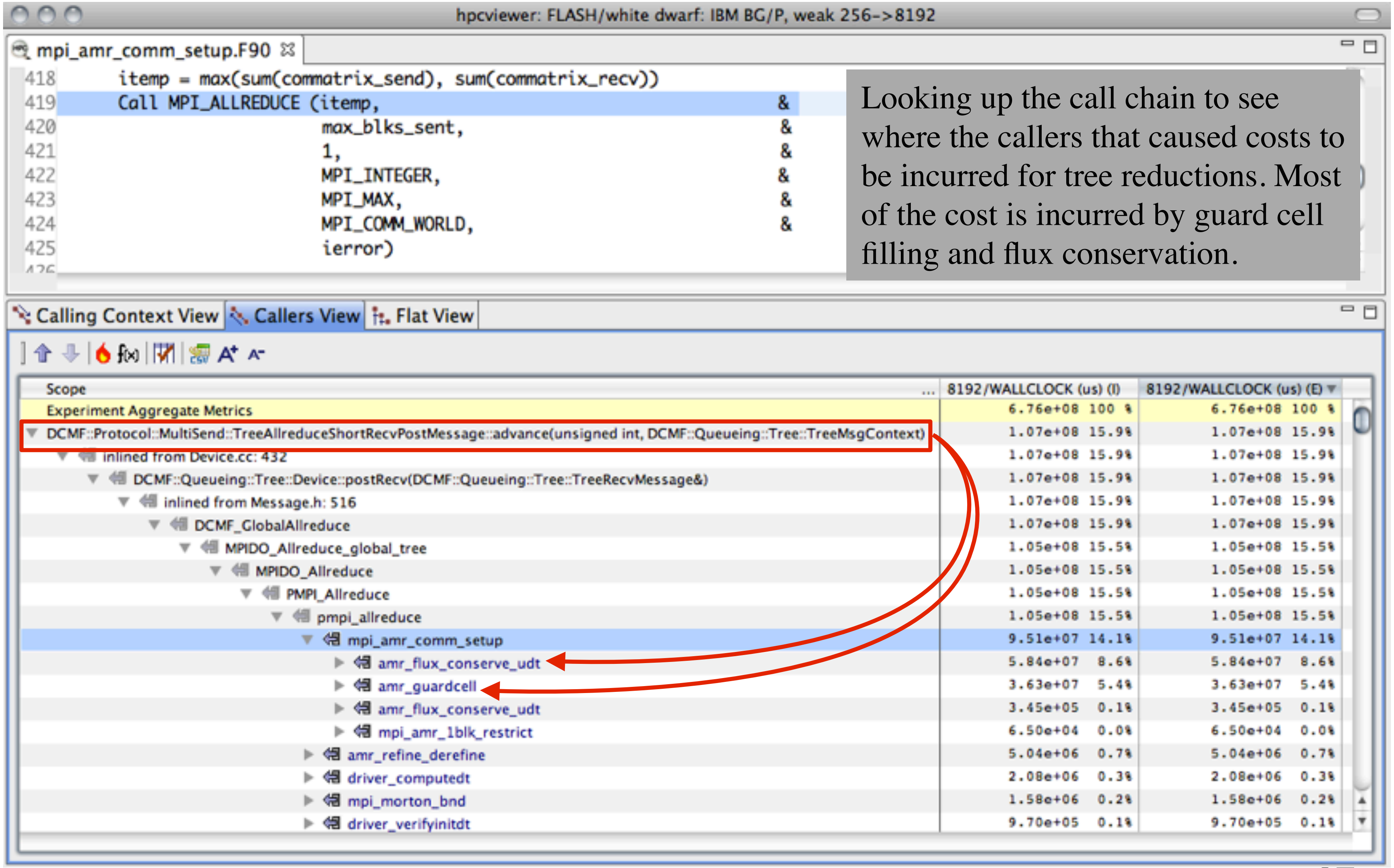
Execution Cost Breakdown (Routine-Level)

Flash on Blue Gene/P, 8K cores, white dwarf detonation



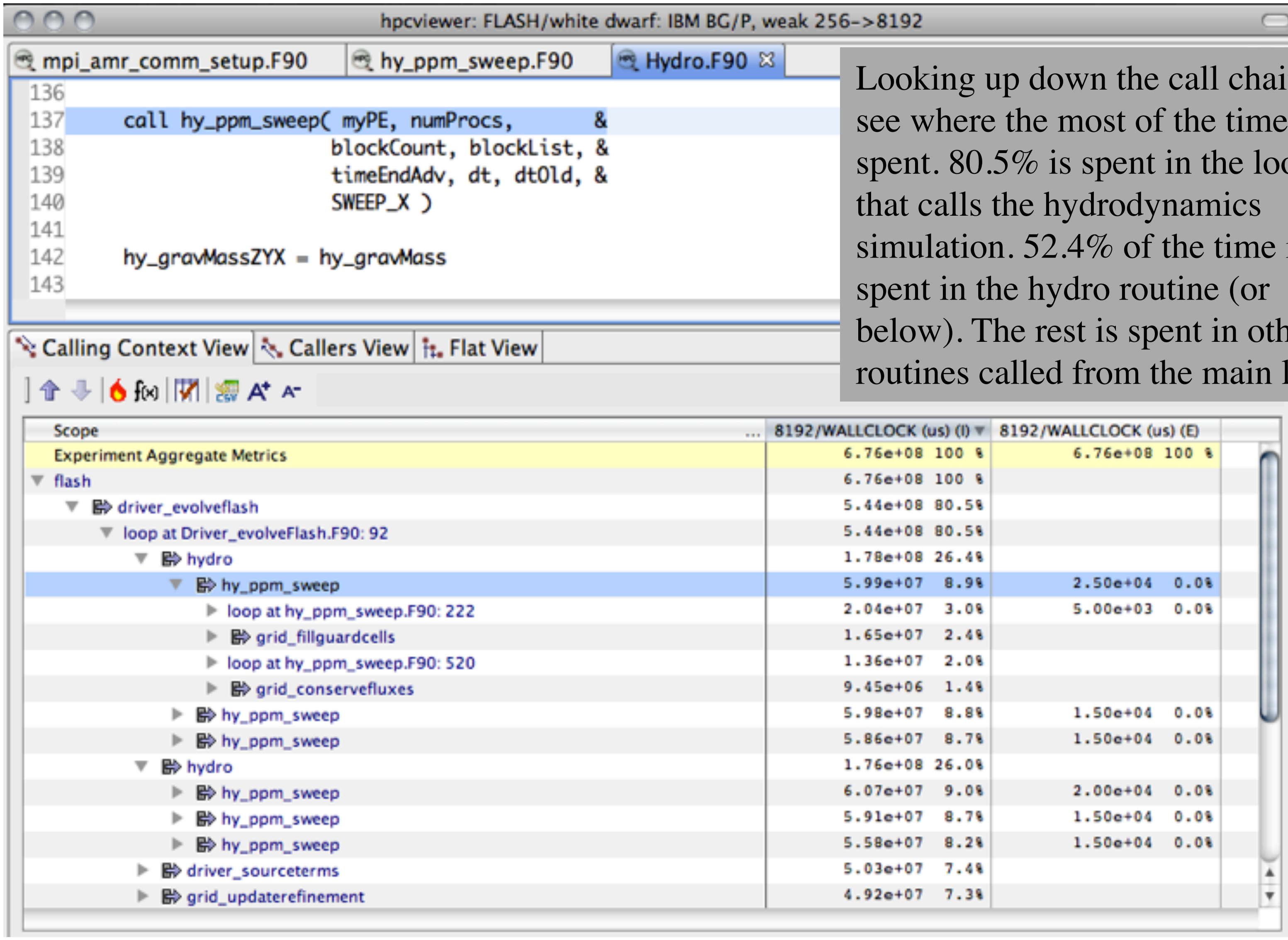
Execution Cost Attribution (Callers View)

Flash on Blue Gene/P, 8K cores, white dwarf detonation



Execution Cost Attribution (Top Down)

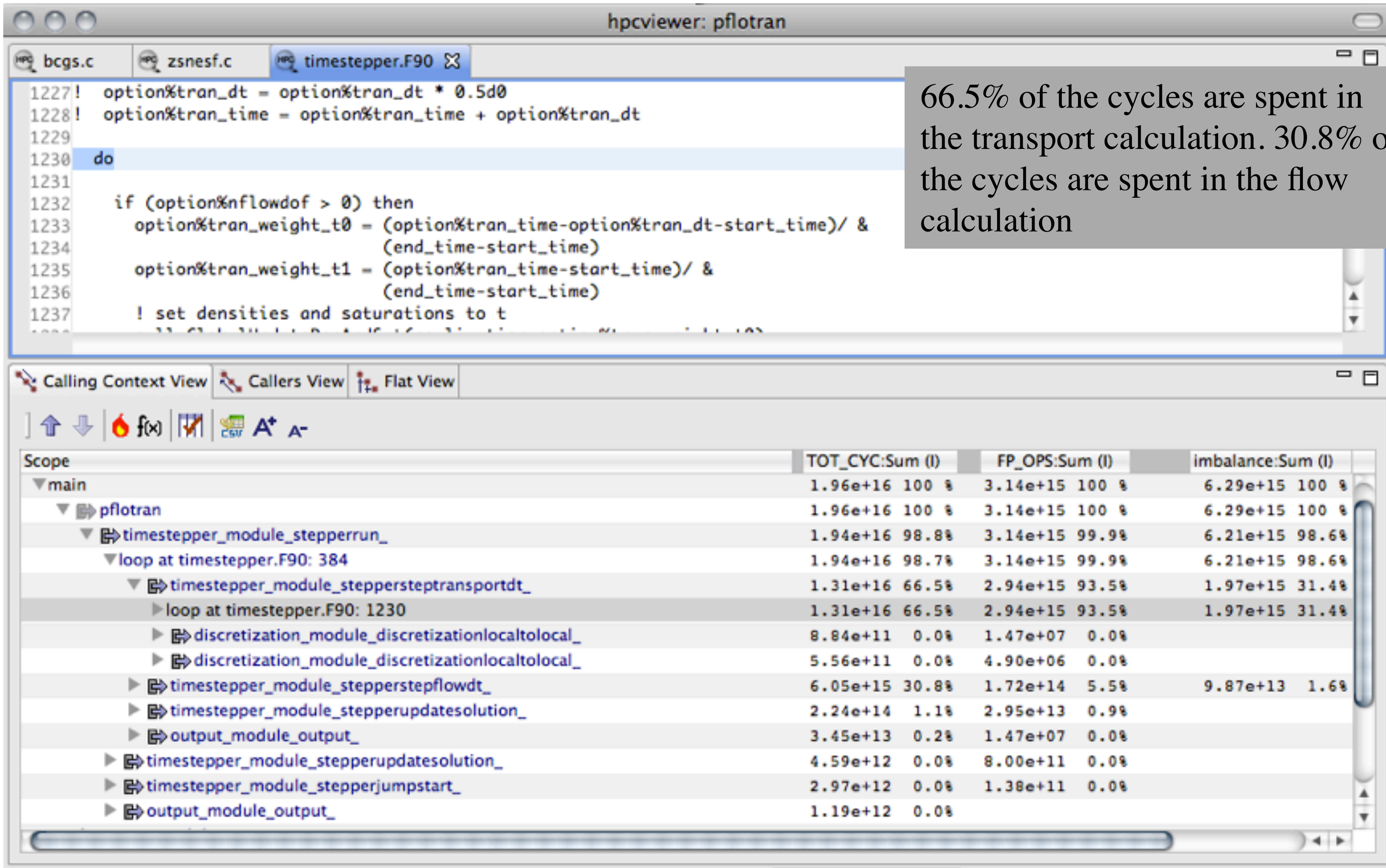
Flash on Blue Gene/P, 8K cores, white dwarf detonation



Looking up down the call chain to see where the most of the time was spent. 80.5% is spent in the loop that calls the hydrodynamics simulation. 52.4% of the time is spent in the hydro routine (or below). The rest is spent in other routines called from the main loop

Execution Cost Attribution (Top-Down)

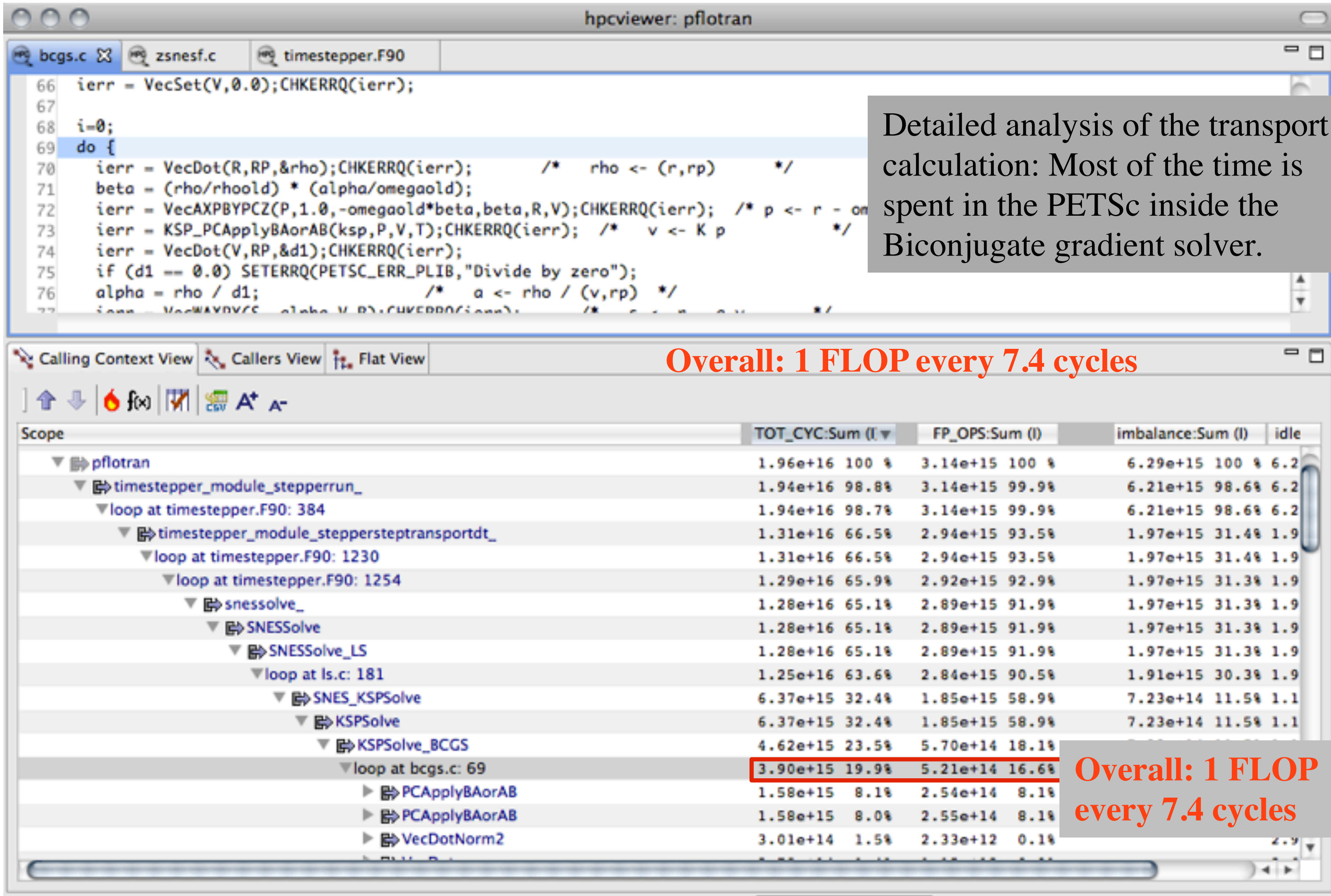
PFLOTRAN, Cray XT, 8184 cores, Hanford problem



66.5% of the cycles are spent in the transport calculation. 30.8% of the cycles are spent in the flow calculation

Execution Cost Attribution (Top-Down)

PFLOTRAN, Cray XT, 8184 cores, Hanford problem



Detailed analysis of the transport calculation: Most of the time is spent in the PETSc inside the Biconjugate gradient solver.

Overall: 1 FLOP every 7.4 cycles

Overall: 1 FLOP every 7.4 cycles