

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



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**

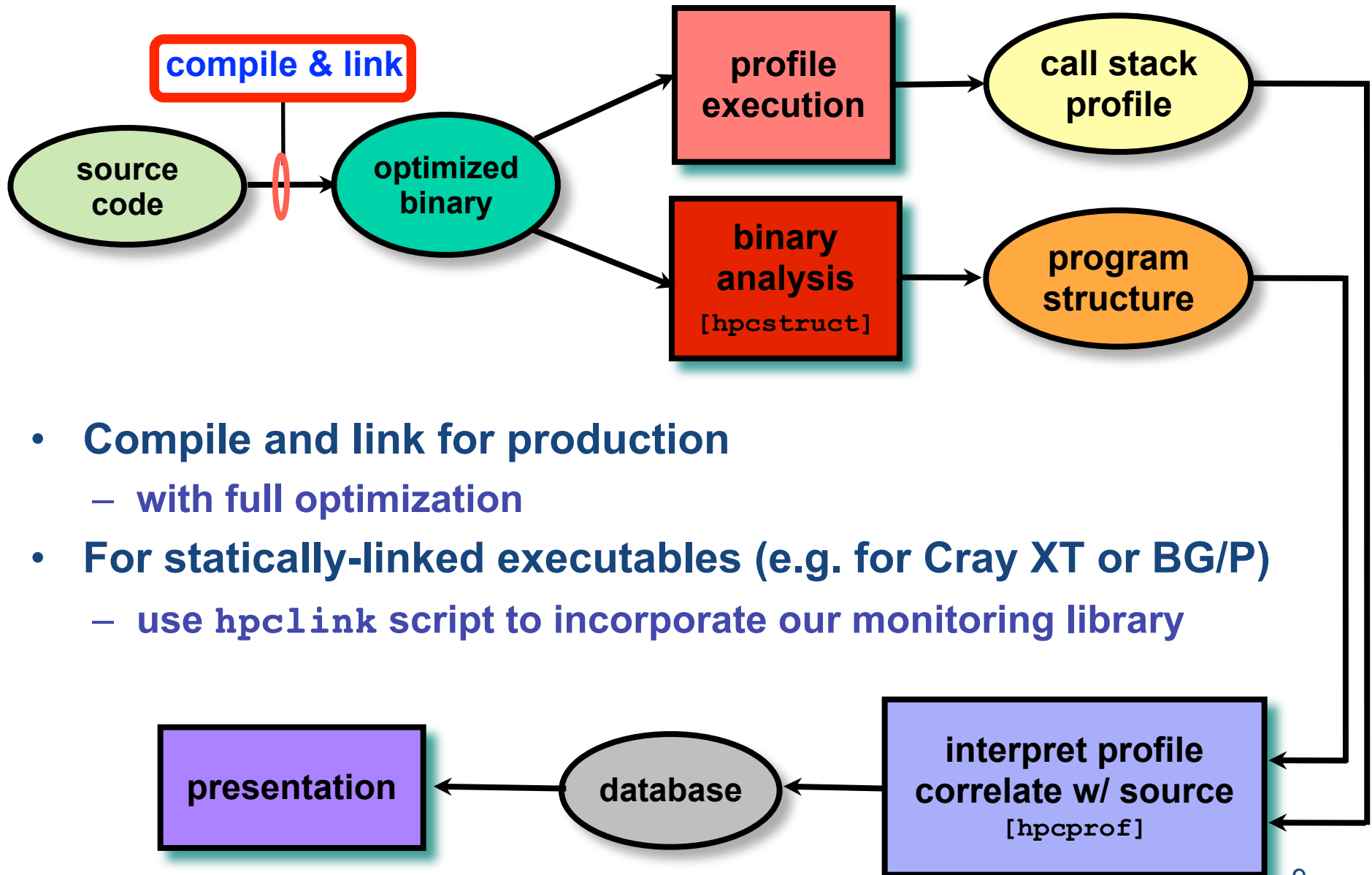
Challenges

- 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

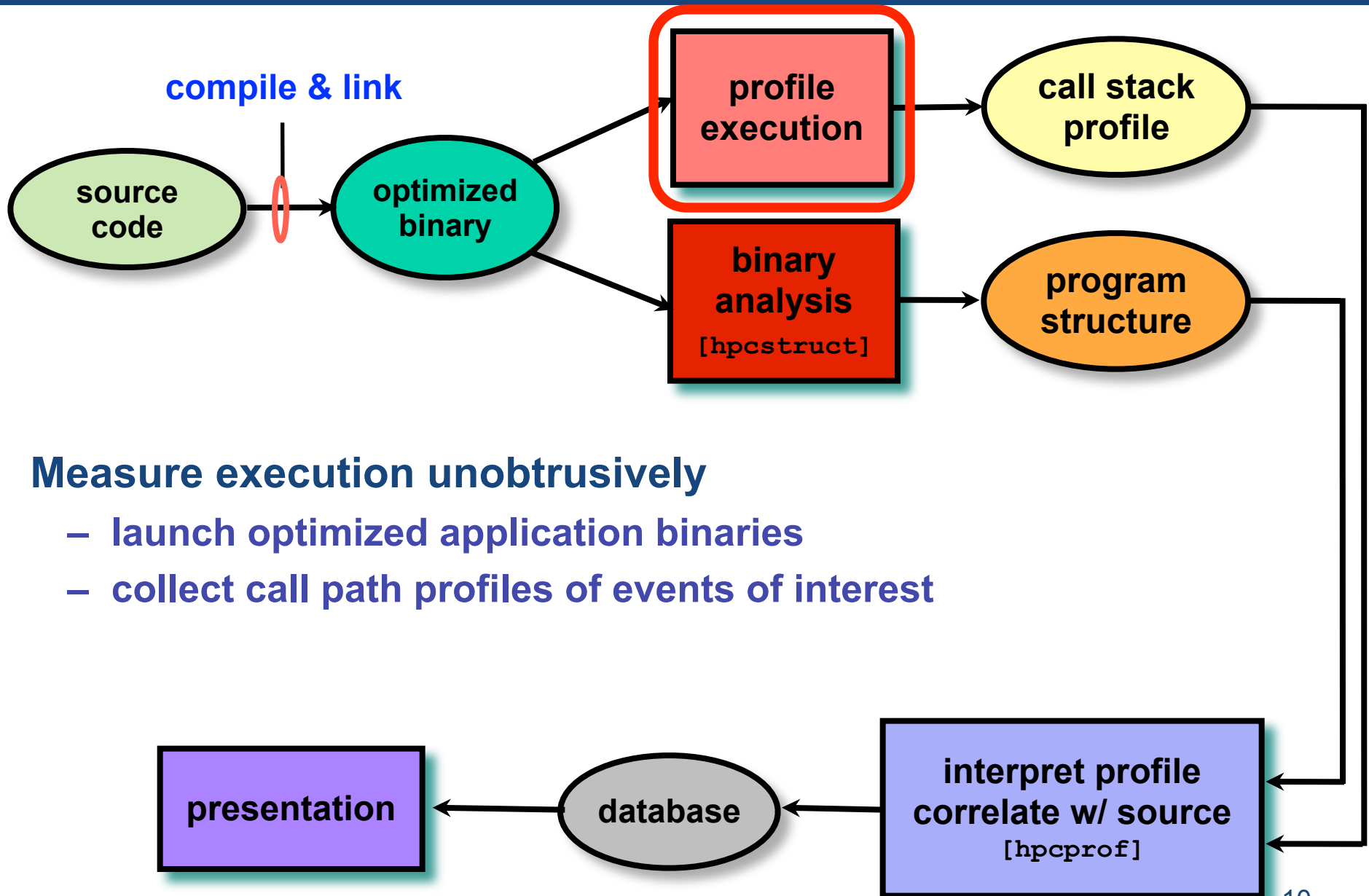
- **Accurate measurement of parallel scientific codes**
 - large, multi-lingual programs
 - fully optimized code: loop optimization, templates, inlining
 - binary-only libraries, sometimes partially stripped
 - complex execution environments
 - dynamic loading or static binaries
 - SPMD parallel codes 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

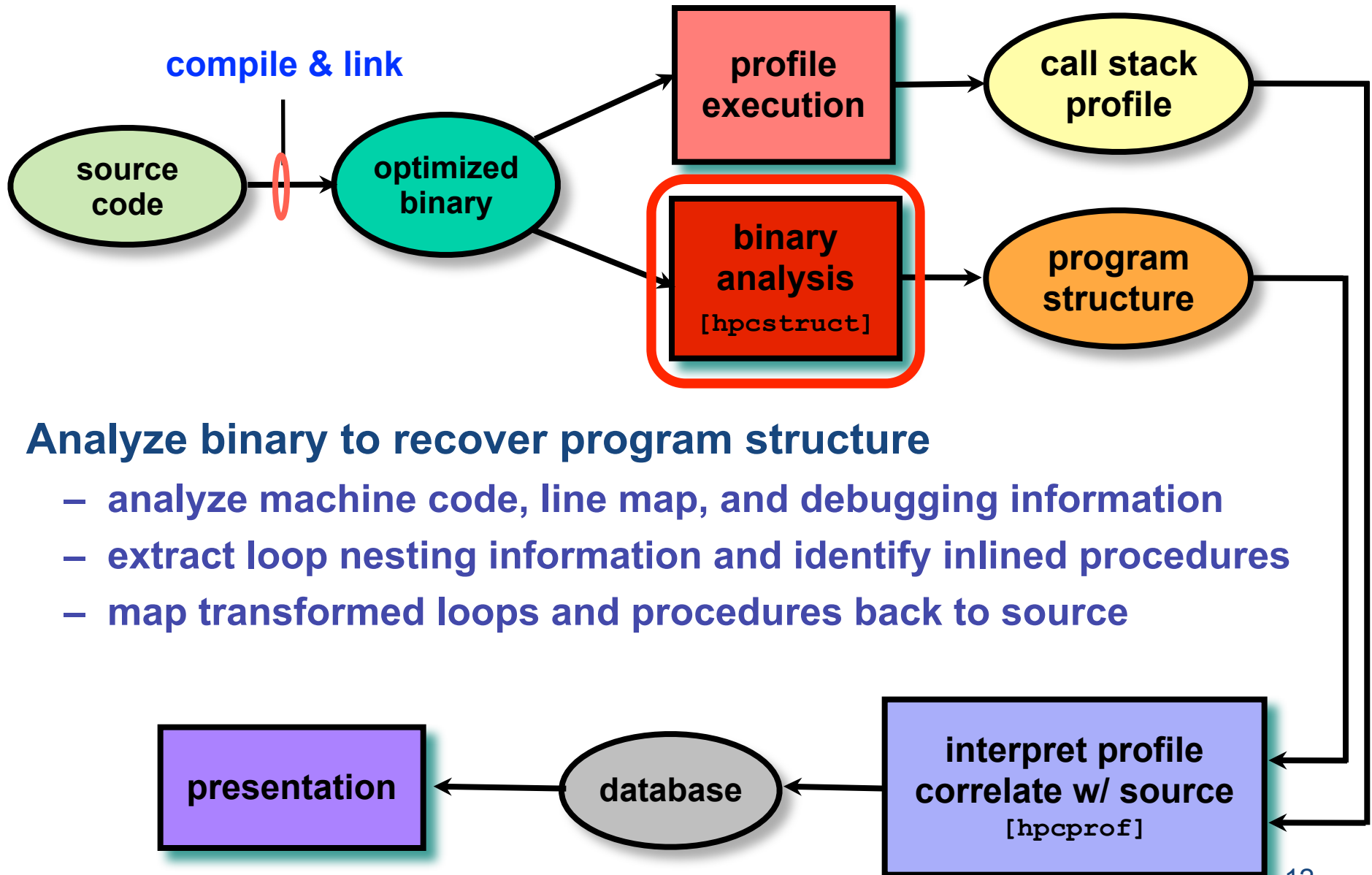
HPCToolkit Performance Tools



Measure execution unobtrusively

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

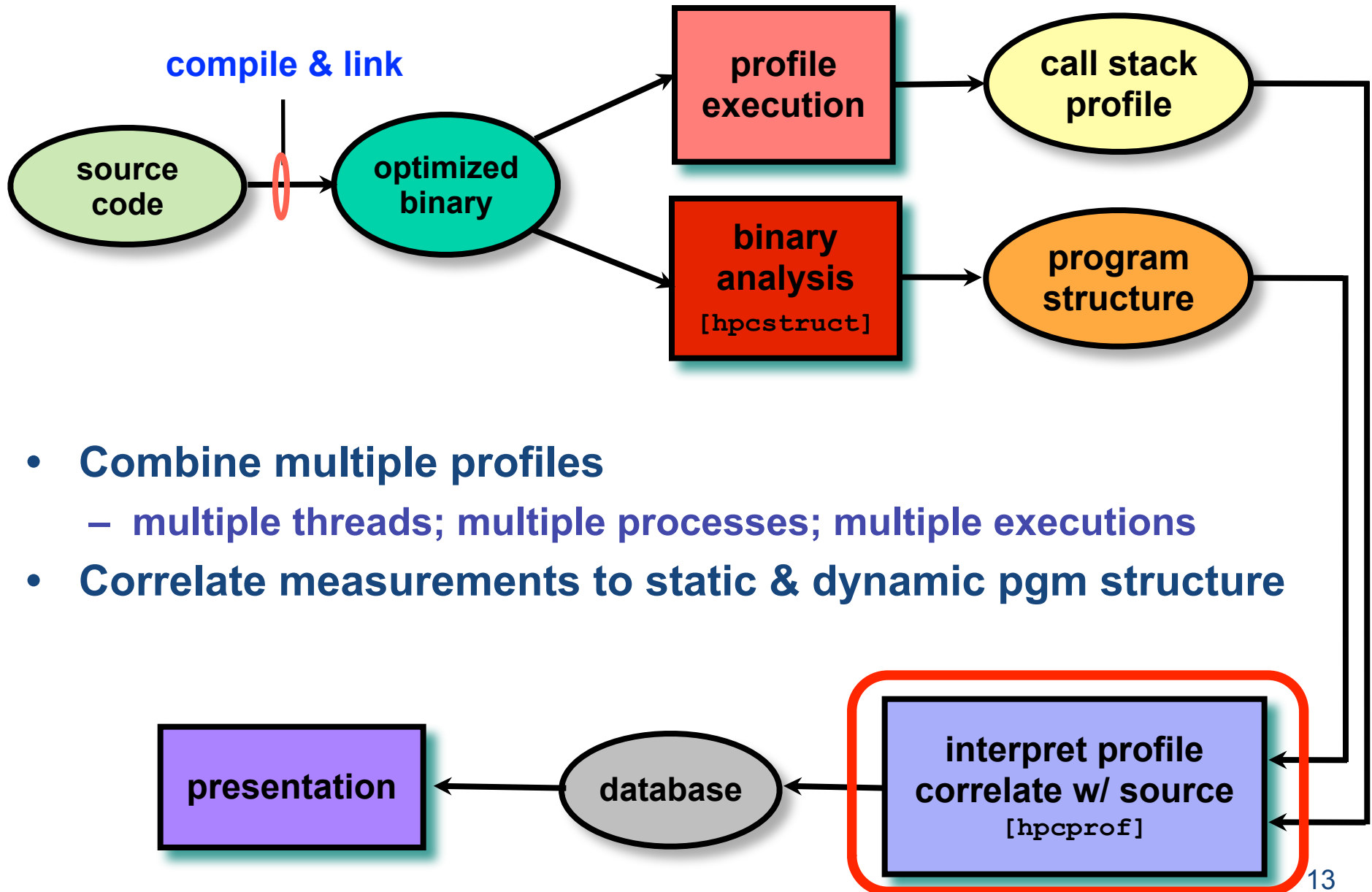
HPCToolkit Performance Tools



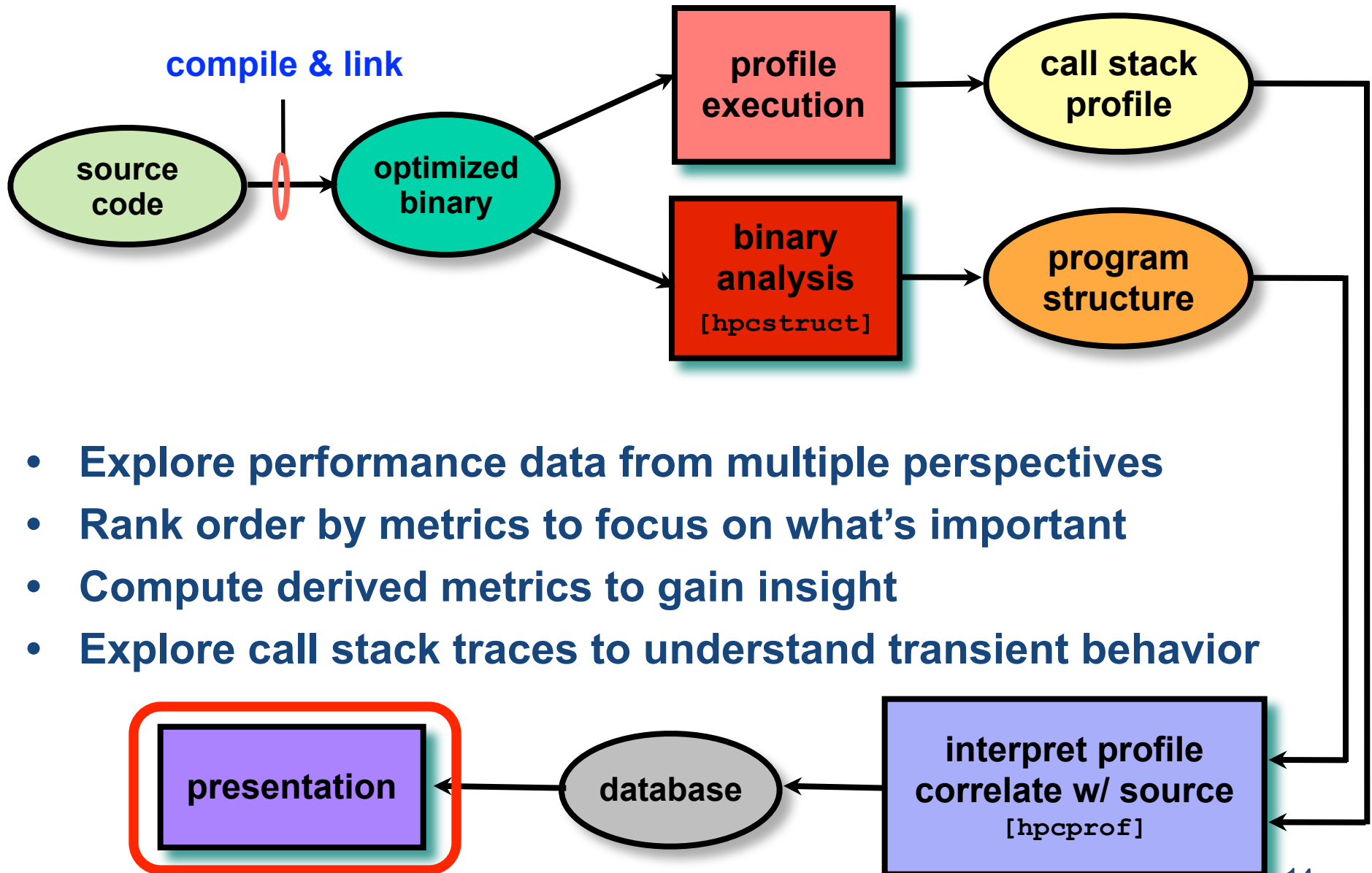
Analyze binary to recover program structure

- 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



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

hpcviewer: MOAB: mbperf_iMesh 200 B (Barcelona 2360 SE)

calling context view

```
mbperf_iMesh.cpp  TypeSequenceManager.hpp  stl_tree.h
```

```
22  * Define less-than comparison for EntitySequence pointers as a comparison
23  * of the entity handles in the pointed-to EntitySequences.
24  */
25  class SequenceCompare {
26  public: bool operator()( const EntitySequence* a, const EntitySequence* b ) const
27  { return a->end_handle() < b->start_handle(); }
28  };
```

costs for

- inlined procedures
- loops
- function calls in full context

Calling Context View Callers View Flat View

Scope PAPI_L1_DCM (I) PAPI_TOT_CYC (I) P

main	8.63e+08 100 %	1.13e+11 100 %	
testB(void*, int, double const*, int const*)	8.35e+08 96.7%	1.10e+11 97.6%	
inlined from mbperf_iMesh.cpp: 261	6.81e+08 78.9%	0.98e+11 86.5%	
loop at mbperf_iMesh.cpp: 280-313	3.43e+08 39.8%	3.37e+10 29.9%	
imesh_getvtxarrcoords_	3.20e+08 37.1%	2.18e+10 19.3%	
MBCore::get_coords(unsigned long const*, int, double*) c	3.20e+08 37.1%	2.16e+10 19.1%	
loop at MBCore.cpp: 681-693	3.20e+08 37.1%	2.16e+10 19.1%	
inlined from stl_tree.h: 472	2.04e+08 23.7%	9.38e+09 8.3%	
loop at stl_tree.h: 1388	2.04e+08 23.6%	9.37e+09 8.3%	
inlined from TypeSequenceManager.hpp: 27	1.78e+08 20.6%	8.56e+09 7.6%	
TypeSequenceManager.hpp: 27	1.78e+08 20.6%	8.56e+09 7.6%	

Outline

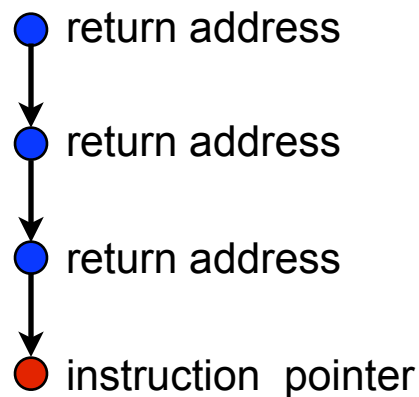
- **Call path profiling in HPCToolkit**
- **Pinpointing and quantifying scalability bottlenecks**
- **Blame shifting**
 - analyzing multithreaded computations based on work stealing
 - quantifying the impact of lock contention on threaded code
 - pinpointing load imbalance in parallel codes
- **Understanding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

Call Path Profiling

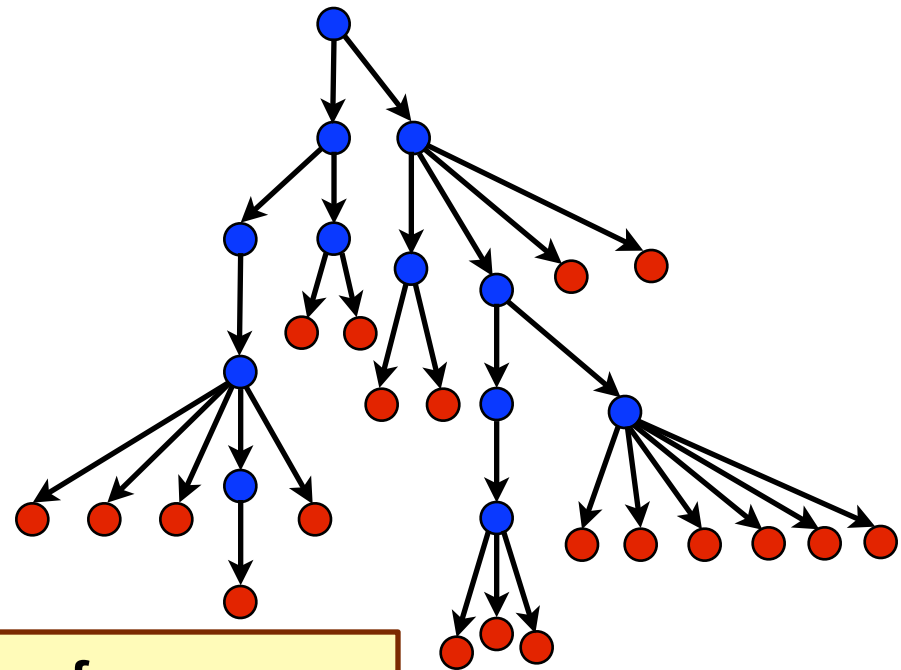
Measure and attribute costs in their *calling* context

- **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

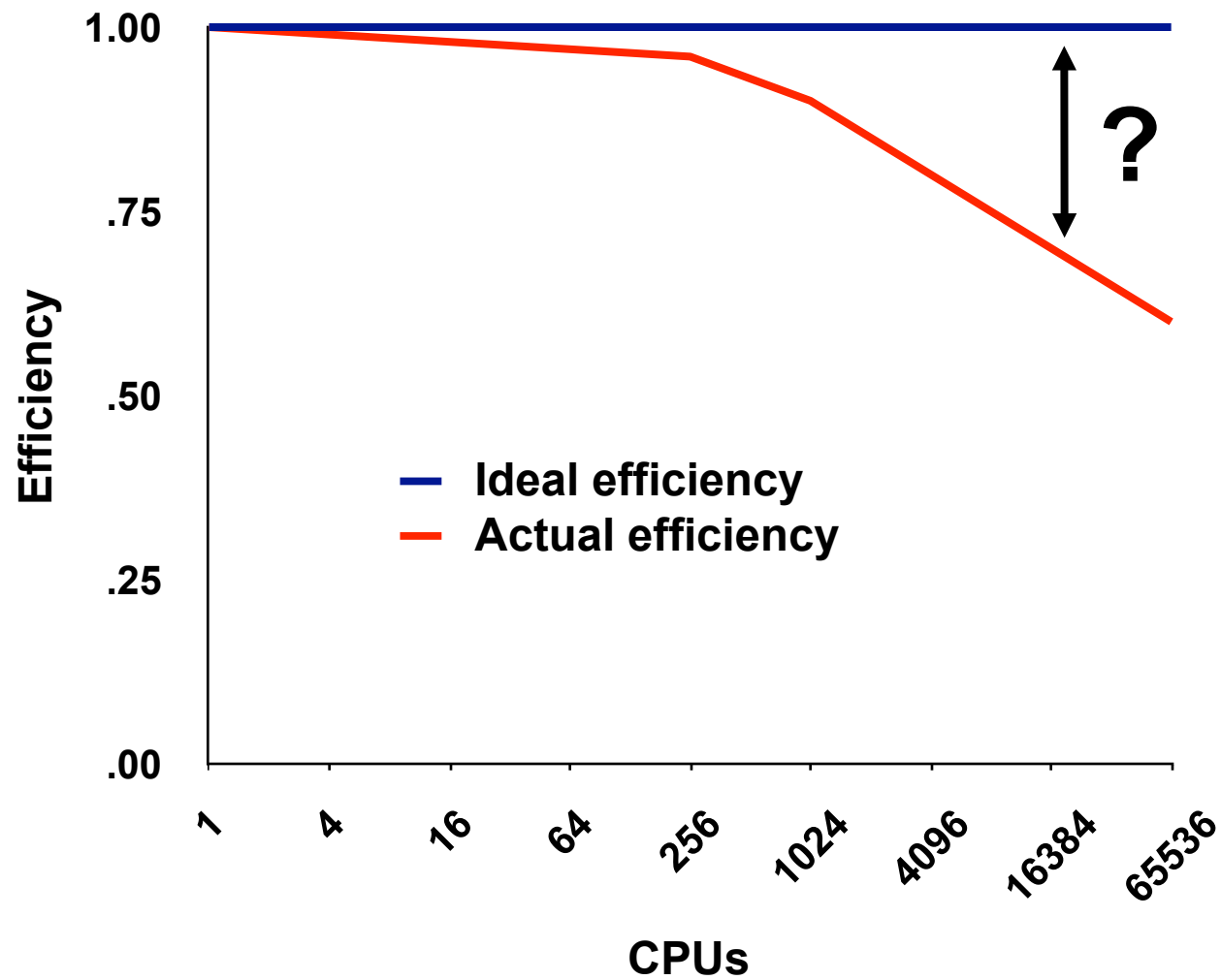
- **Identify procedure bounds**
 - for dynamically-linked code, do this at runtime
 - for statically-linked code, do this at compile time
- **Compute unwind recipes for a procedure on the fly**
 - scan the procedure's object code, tracking the locations of
 - caller's program counter
 - caller's frame 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

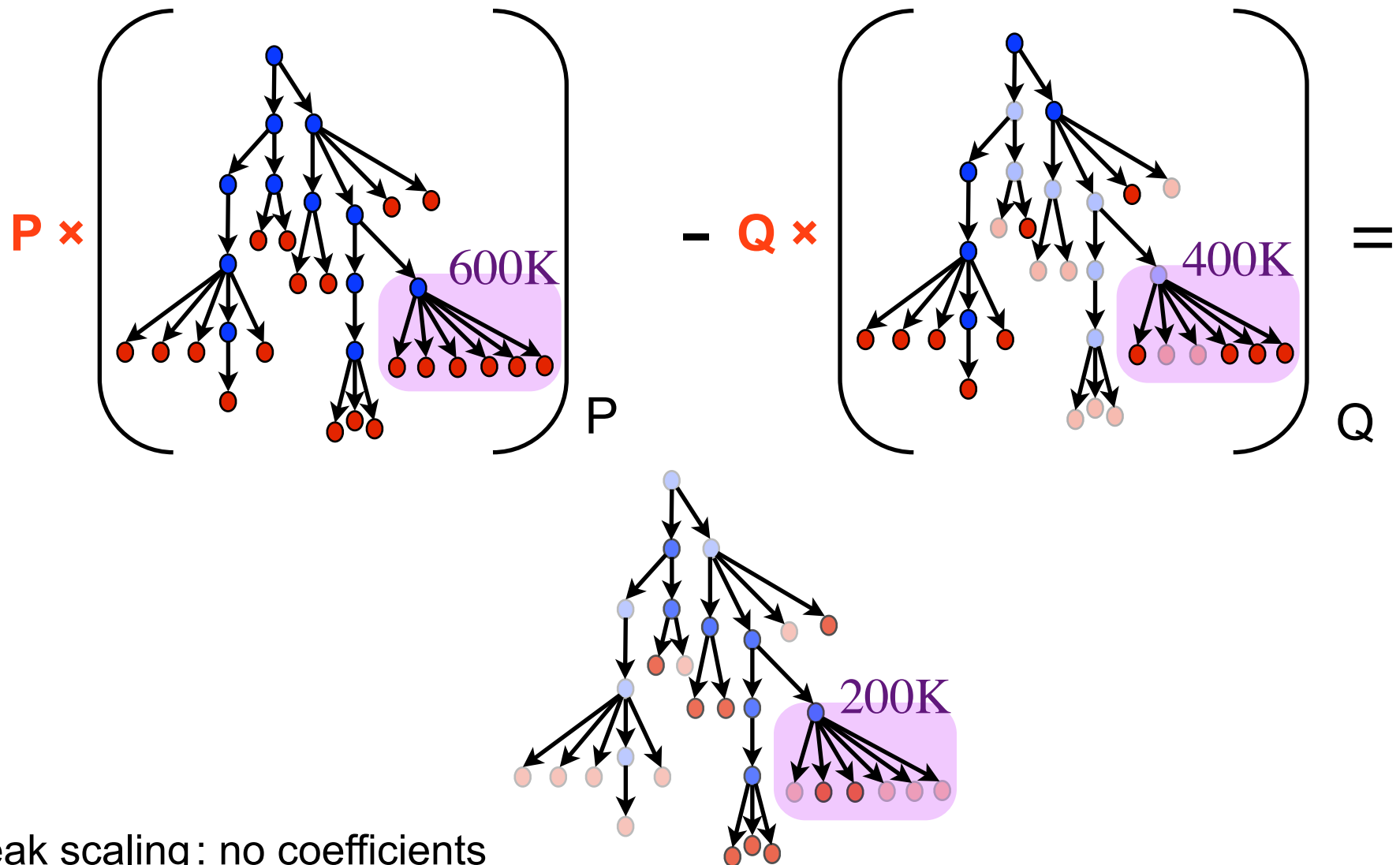
- Call path profiling in HPCToolkit
- Pinpointing and quantifying scalability bottlenecks
- Blame shifting
 - analyzing multithreaded computations based on work stealing
 - quantifying the impact of lock contention on threaded code
 - pinpointing load imbalance in parallel codes
- Understanding execution 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:

Simulation:

Platform:

Experiment:

Scaling type:

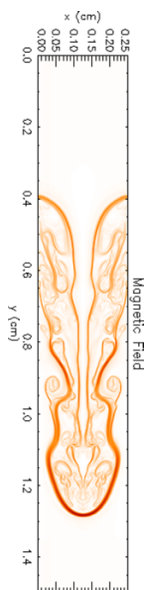
University of Chicago FLASH

white dwarf detonation

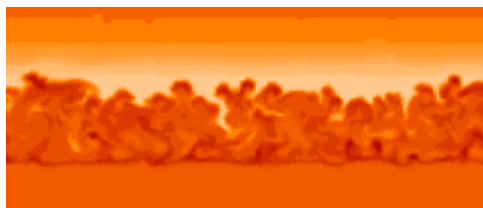
Blue Gene/P

8192 vs. 256 processors

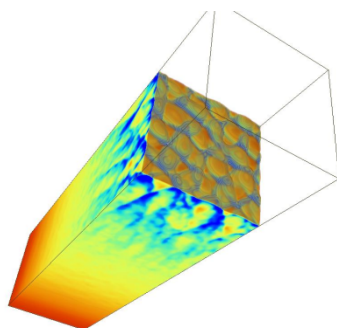
weak



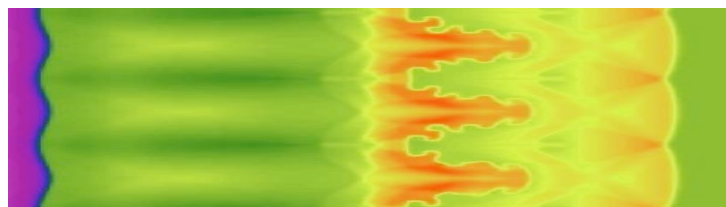
*Magnetic
Rayleigh-Taylor*



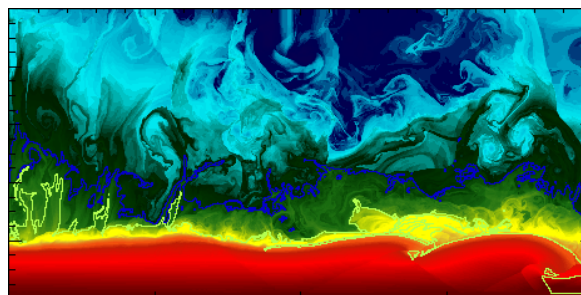
Nova outbursts on white dwarfs



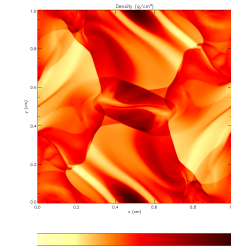
Cellular detonation



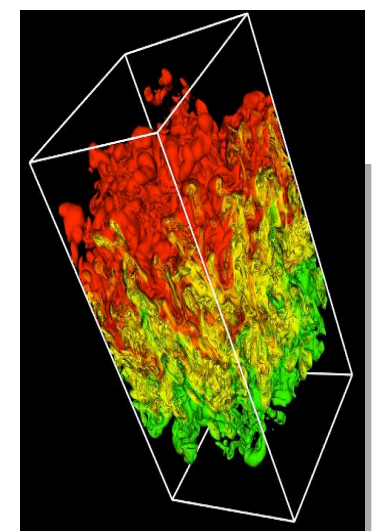
Laser-driven shock instabilities



Helium burning on neutron stars



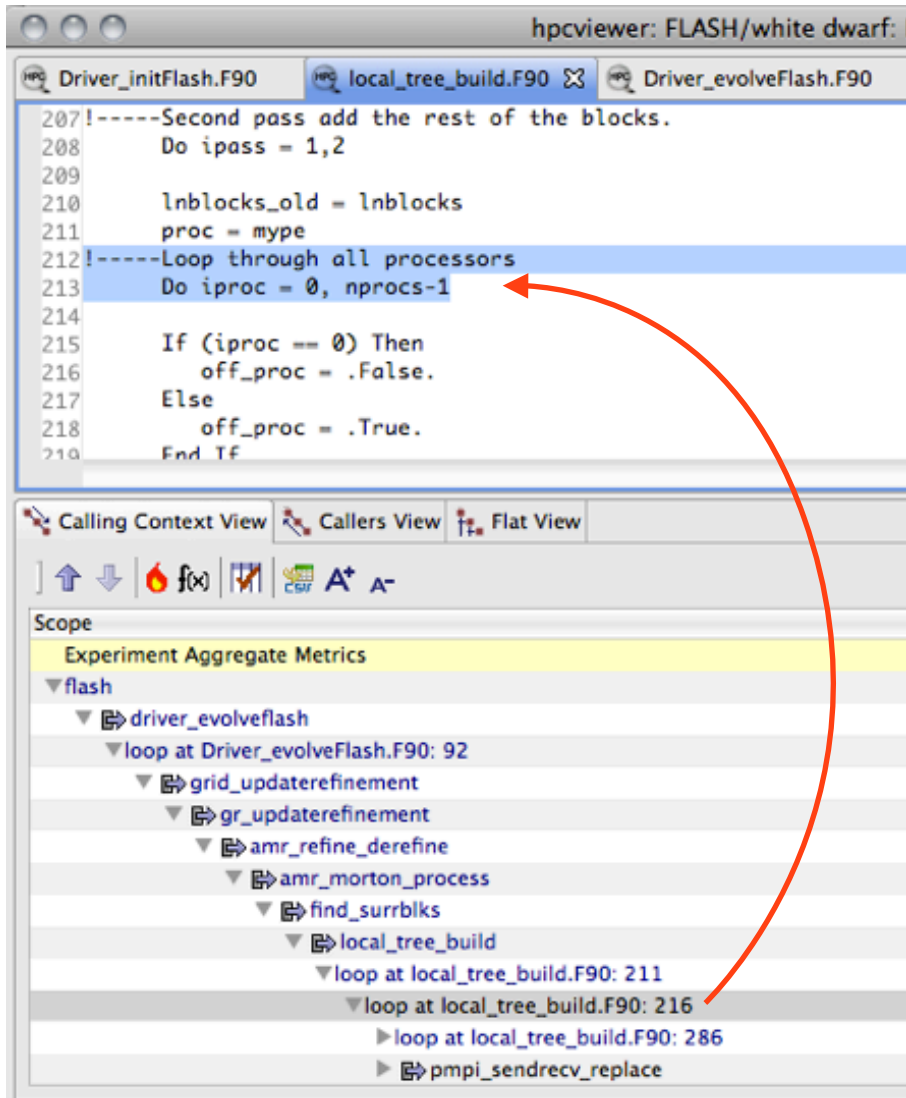
*Orzag/Tang MHD
vortex*



Rayleigh-Taylor instability

Figures courtesy of FLASH Team, University of Chicago

System-wide Scaling Losses in Flash



```

207!-----Second pass add the rest of the blocks.
208  Do ipass = 1,2
209
210    lnblocks_old = lnblocks
211    proc = mype
212!-----Loop through all processors
213  Do iproc = 0, nprocs-1
214
215    If (iproc == 0) Then
216      off_proc = .False.
217    Else
218      off_proc = .True.
219    End If

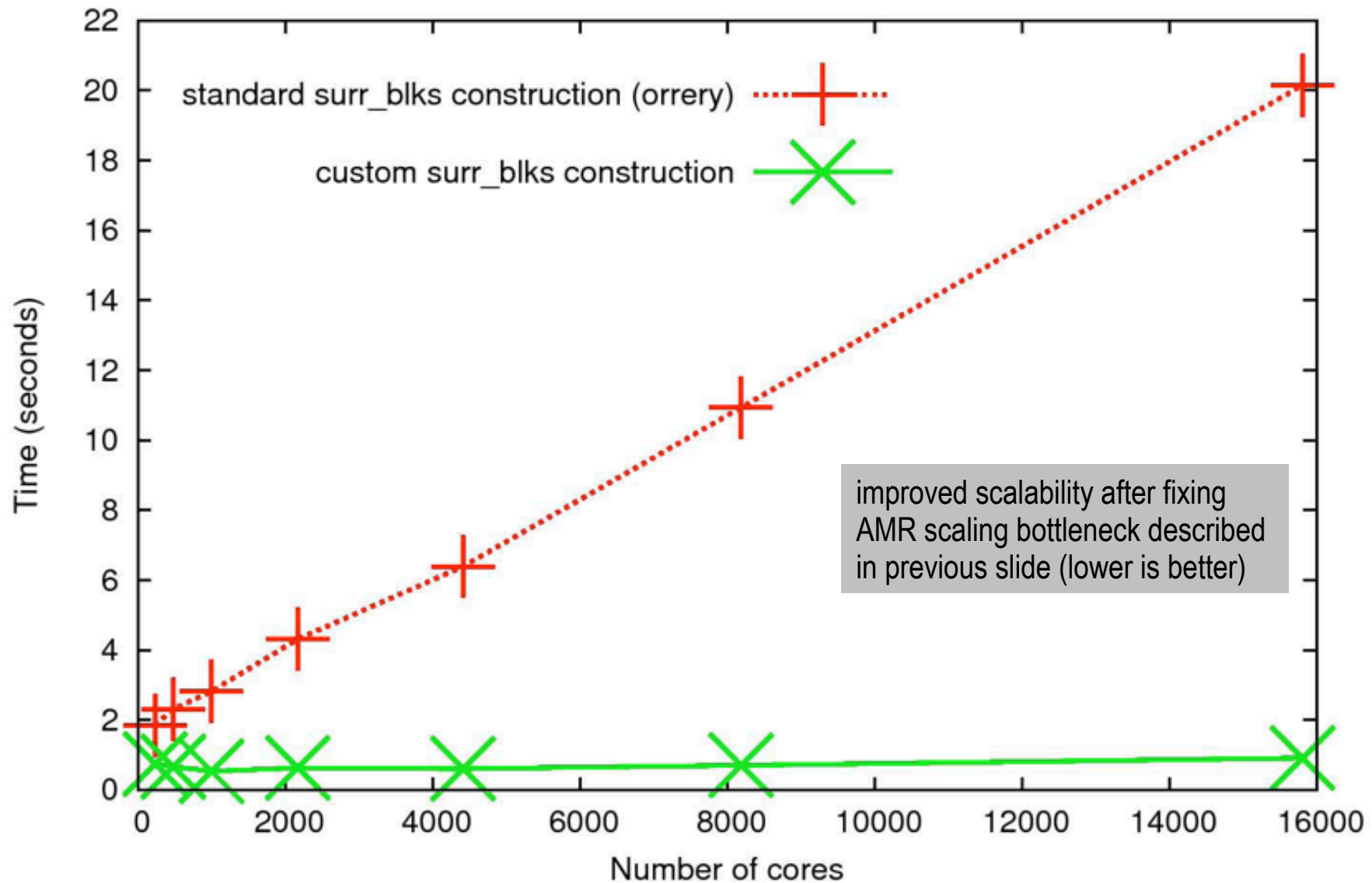
```

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

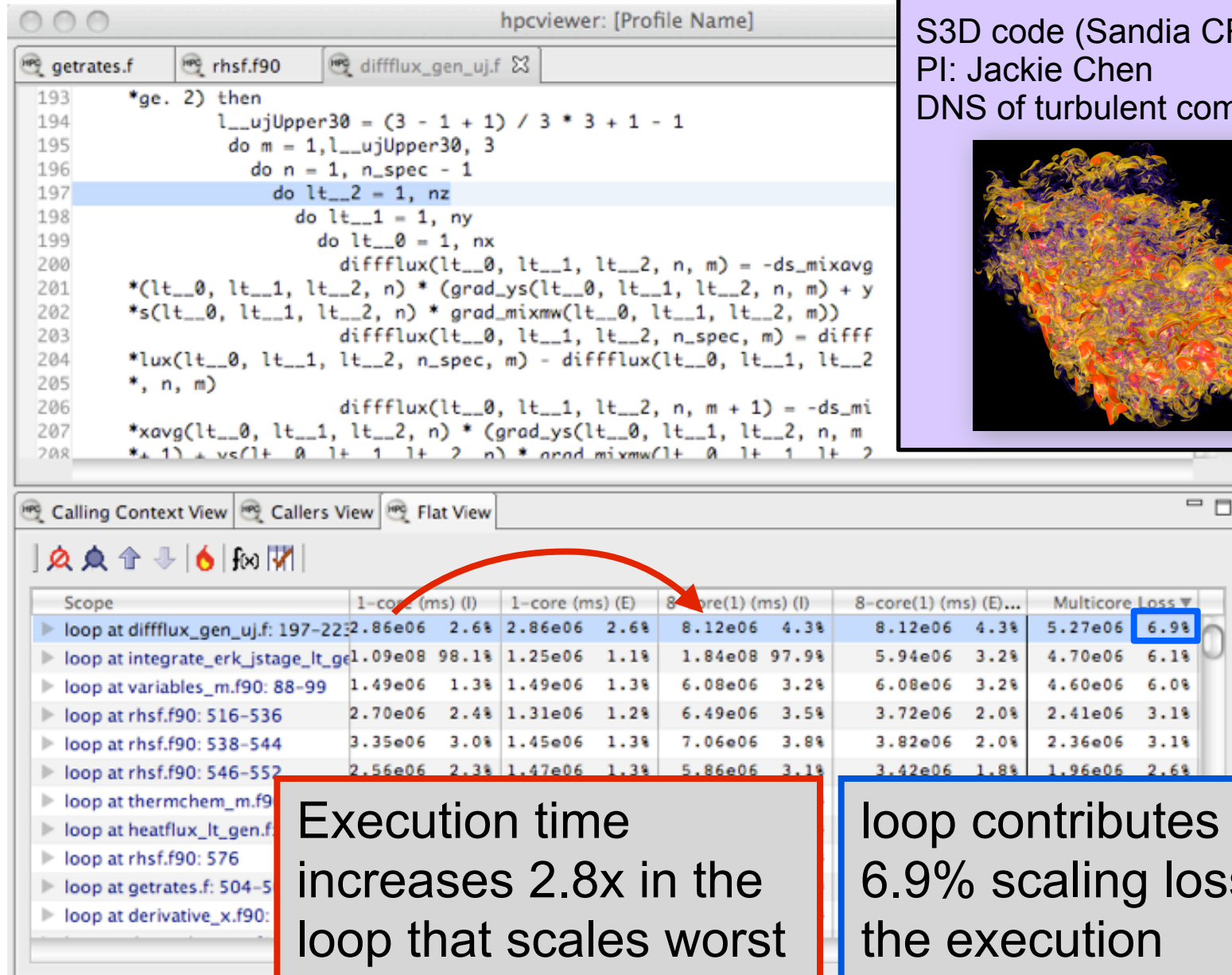
Scope	% scalability loss (l)	256/WALLCLOCK (us) (l)	8192/WALLCLOCK (us) (l)
Experiment Aggregate Metrics	2.46e+01 100 %	5.07e+08 100 %	6.71e+08 100 %
flash	2.46e+01 100 %	5.07e+08 100 %	6.71e+08 100 %
driver_evolveflash	1.41e+01 57.5%	4.46e+08 88.1%	5.41e+08 80.6%
loop at Driver_evolveFlash.F90: 92	1.41e+01 57.5%	4.46e+08 88.1%	5.41e+08 80.6%
grid_updaterefinement	3.89e+00 15.8%	2.24e+07 4.4%	4.84e+07 7.2%
gr_updaterefinement	3.77e+00 15.4%	2.52e+06 0.5%	2.78e+07 4.1%
amr_refine_derefine	3.64e+00 14.8%	5.75e+05 0.1%	2.50e+07 3.7%
amr_morton_process	3.42e+00 13.9%	2.65e+05 0.1%	2.32e+07 3.5%
find_surrblks	3.30e+00 13.4%	2.50e+05 0.0%	2.24e+07 3.3%
local_tree_build	3.29e+00 13.4%	2.40e+05 0.0%	2.24e+07 3.3%
loop at local_tree_build.F90: 211	3.29e+00 13.4%	2.40e+05 0.0%	2.24e+07 3.3%
loop at local_tree_build.F90: 216	3.29e+00 13.4%	2.40e+05 0.0%	2.24e+07 3.3%
loop at local_tree_build.F90: 286	1.43e+00 5.8%	1.20e+05 0.0%	9.75e+06 1.5%
pmpi_sendrecv_replace	4.42e-01 1.8%	2.50e+04 0.0%	2.99e+06 0.4%

Improved Flash Scaling of AMR Setup

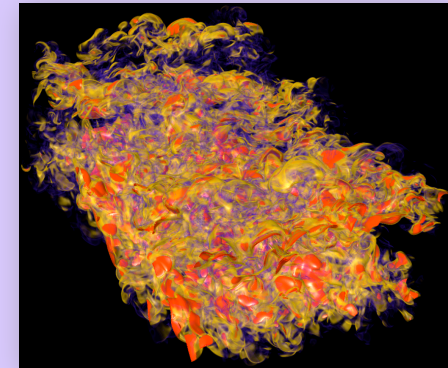


Graph courtesy of Anshu Dubey, U Chicago

Scalability Losses at the Loop Level



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



Outline

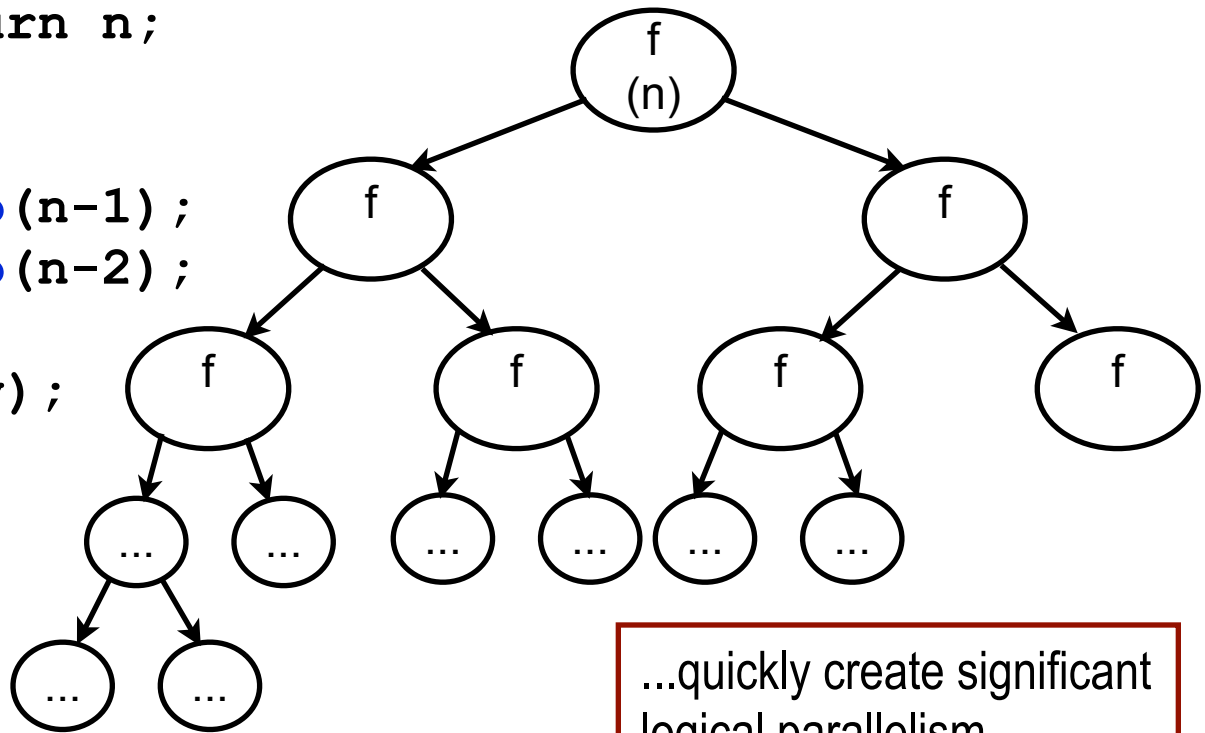
- Call path profiling in HPCToolkit
- Pinpointing and quantifying scalability bottlenecks
- Blame shifting
 - analyzing multithreaded computations based on work stealing
 - quantifying the impact of lock contention on threaded code
 - pinpointing load imbalance in parallel codes
- Understanding execution 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...**

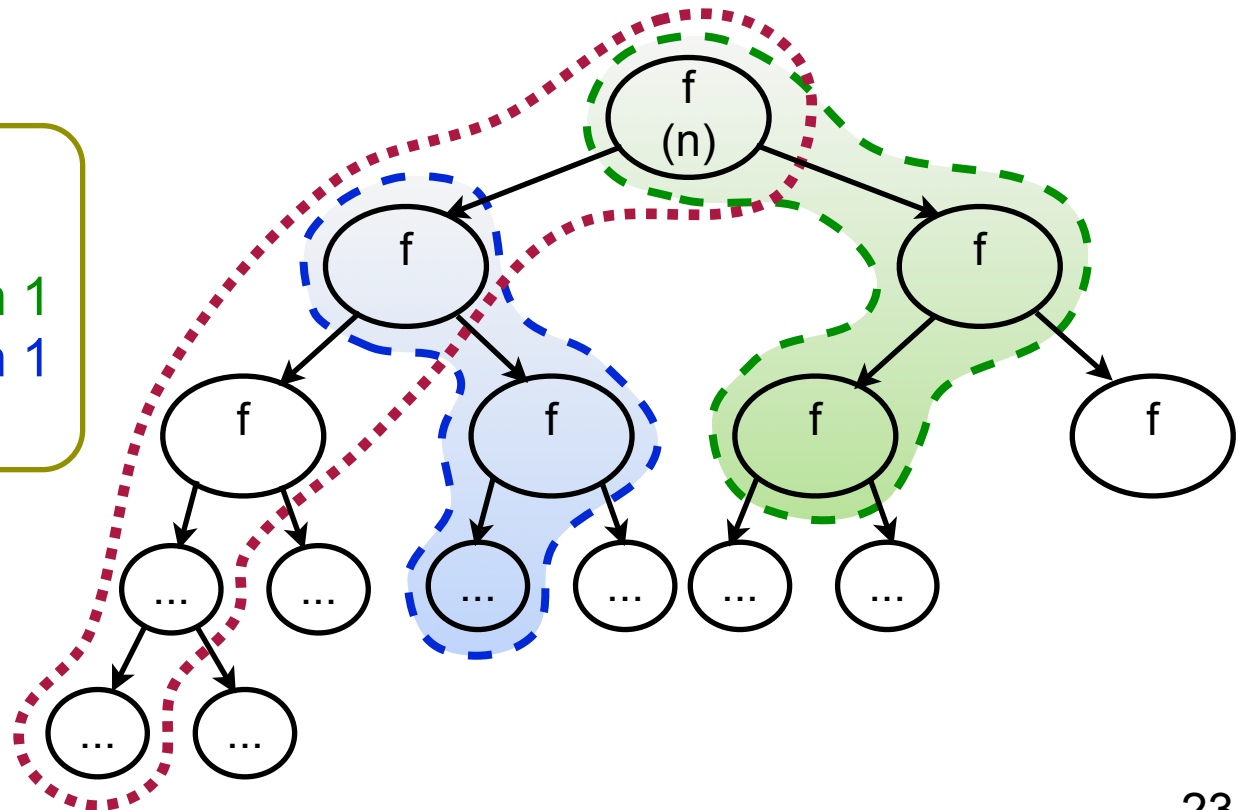
...quickly create significant logical parallelism.

Cilk Program Execution using Work Stealing

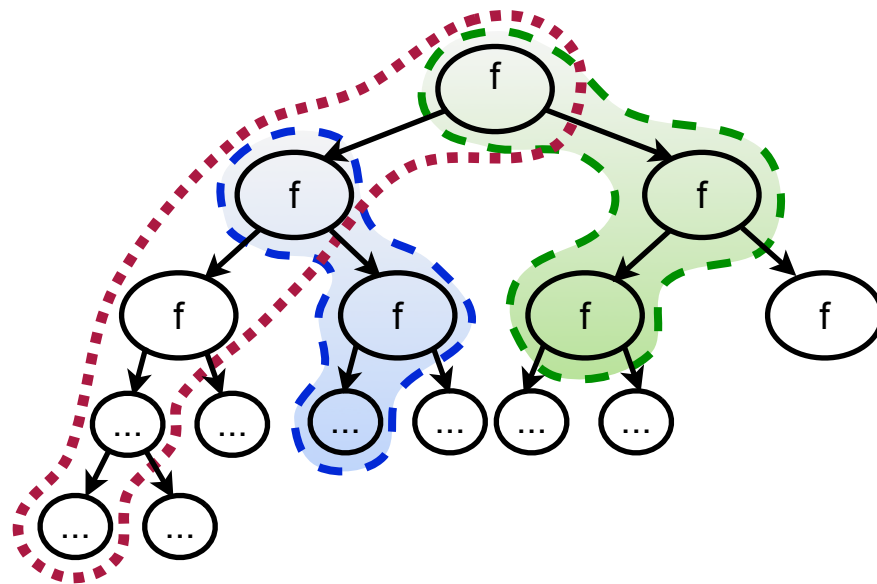
- Challenge: Mapping logical tasks to compute cores
- Cilk approach:
 - lazy thread creation plus work-stealing scheduler
 - **spawn**: a potentially parallel task is available
 - an idle thread steals tasks from a random working thread

Possible Execution:

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



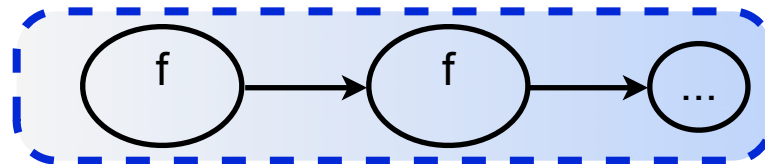
Wanted: Call Path Profiles of Cilk



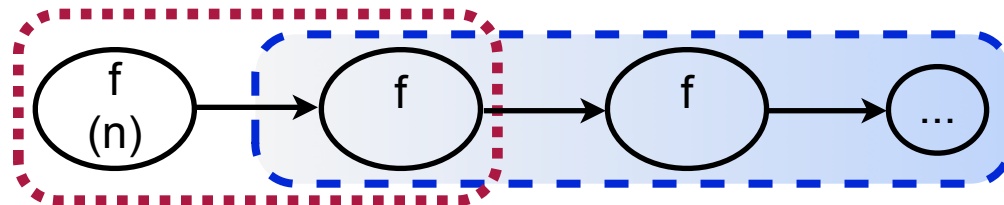
thread 1
thread 2
thread 3

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

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



- logical call path:



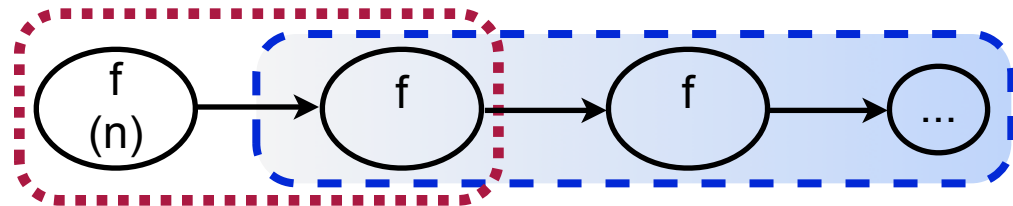
Logical call path profiling: Recover *full* relationship
between *physical* and *user-level* execution

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);  
  }  
}
```



high parallel overhead from
creating many small tasks

- 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 others
- **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 working thread
 - $I = \text{\#cores} - W$
 - apportion others' idleness to me: I / W
- **Example: Dual quad-cores; on a sample, 5 are **working**:**



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

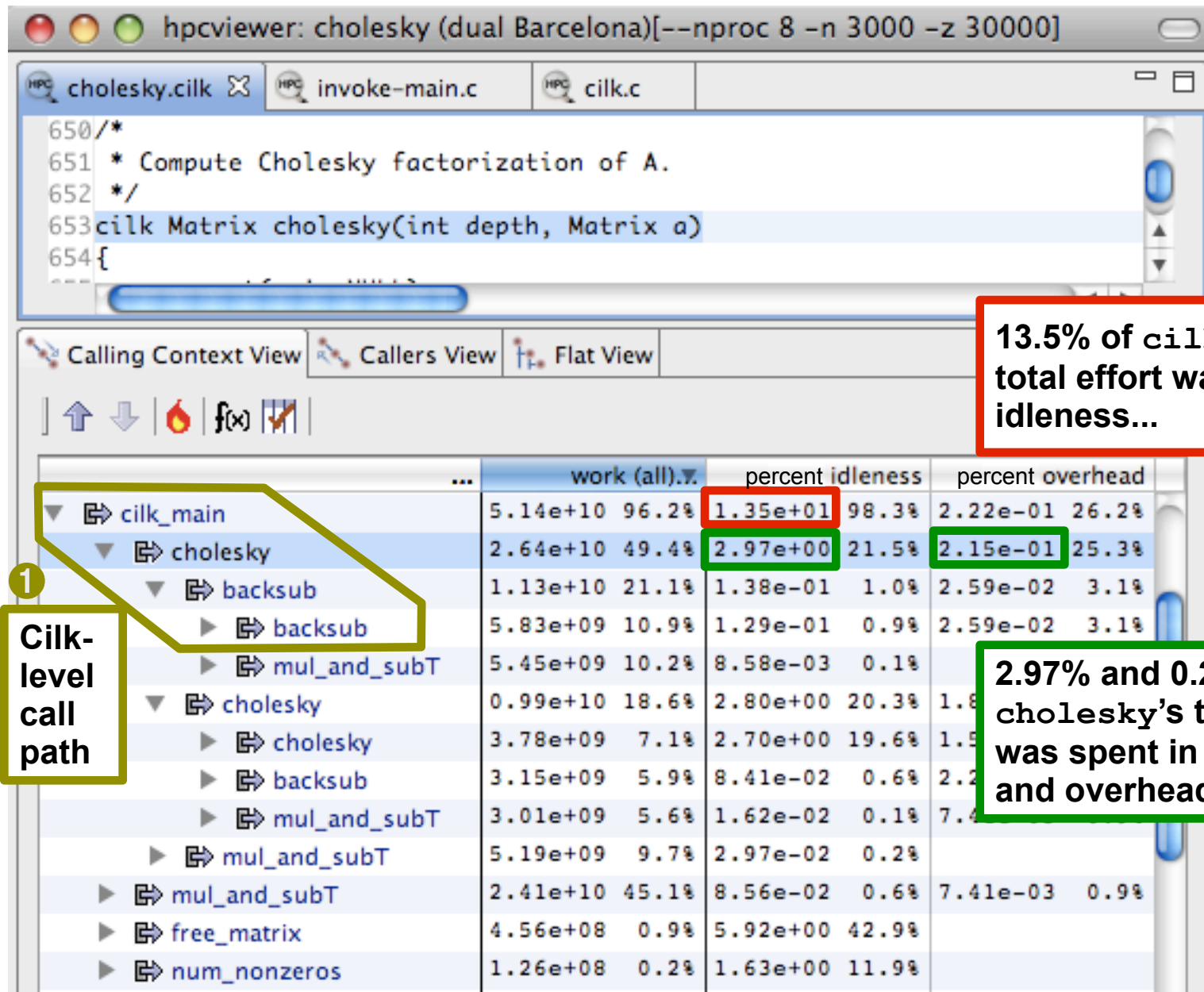


idle: drop sample
(it's in the scheduler!)

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'



Bottom-up Idleness for Cilk 'Cholesky'

The screenshot shows the hpcviewer interface for a Cilk 'Cholesky' program. The top window displays the source code for `free_matrix`, with lines 283-292 visible. The bottom window shows the 'Calling Context View' with a table of performance metrics. The table has columns for Scope, work (all), percent idleness, and percent overhead. The 'percent idleness' column is highlighted, and the values for the 'free' and 'num_nonzeros' scopes are circled in red.

```
283 void free_matrix(int depth, Matrix a)
284 {
285     if (a == NULL)
286         return;
287     if (depth == BLOCK_DEPTH) {
288         free(a);
289     } else {
290         depth--;
291         free_matrix(depth, a->child[_00]);
292         free_matrix(depth, a->child[_01]);
293     }
```

Scope	work (all)	percent idleness	percent overhead
▼ _int_free	2.00e+08	0.4%	2.59e+00 18.8%
▶ free_matrix	1.92e+08	0.4%	2.49e+00 18.1%
▶ free_matrix	4.00e+06	0.0%	5.19e-02 0.4%
▶ free	1.50e+08	0.3%	1.94e+00 14.1%
▶ num_nonzeros	1.26e+08	0.2%	1.63e+00 11.9%
▶ mag	1.16e+08	0.2%	1.50e+00 10.9%

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

```
578     add(MEMFUN_OBJT(memfunT)& obj,  
579         memfunT memfun,  
580         const arg1T& arg1, const arg2T& arg2, const arg3T& arg3, const TaskAttributes&  
581         Future<REMFUTURE(MEMFUN_RETURNT(memfunT))> result;  
582         add(new TaskMemfun<memfunT>(result,obj,memfun,arg1,arg2,arg3,attr));  
583         return result;  
584     }
```

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%	4.92e+08 31.2%
▼ inlined from worldtask.h: 569		4.56e+00 19.4%	3.09e+07 19.4%
▶ madness::Future<> madness::WorldObject<>::task<>		4.56e+00 19.4%	3.09e+07 19.4%
▶ inlined from worlddep.h: 68		1.53e+00 6.5%	1.02e+07 6.5%
▼ 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...

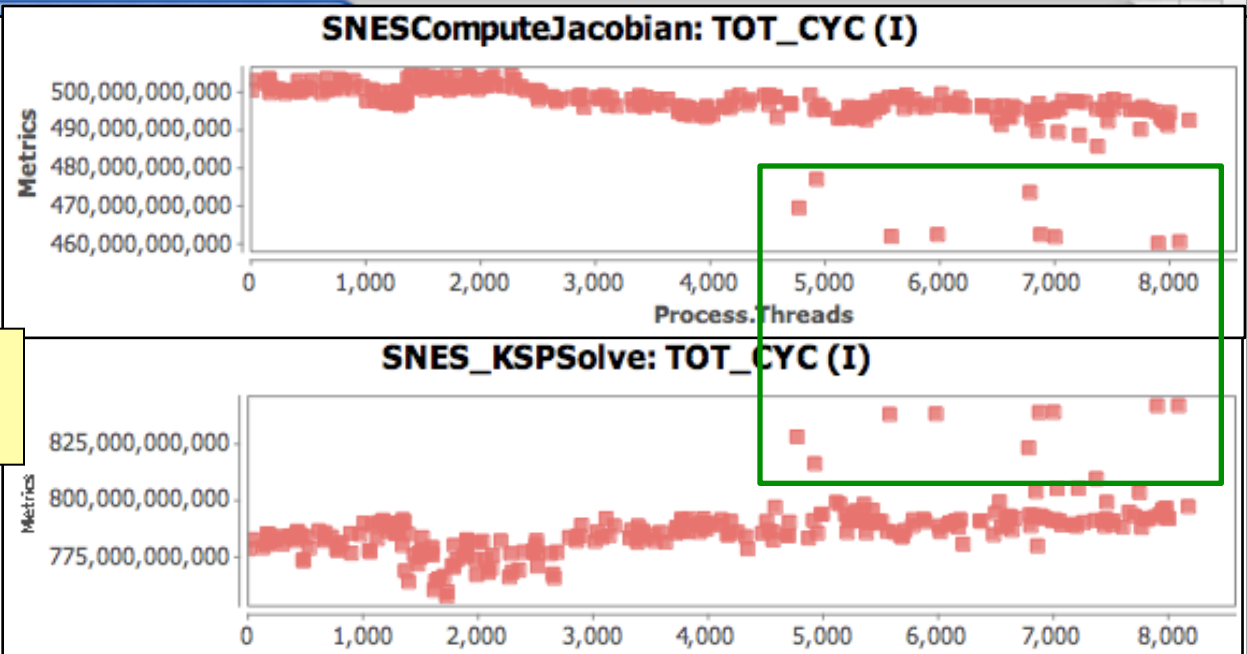
3. Plot the per-process values:

Early finishers...

... become early arrivers at Allreduce

	imbalance (I)	TOT_CYC:Sum (I)	
pflotran	5.28e+15	1.85e+16	100 %
timestepper_module_stepperrun_	5.17e+15	1.82e+16	98.3 %
loop at timestepper.F90: 384	5.17e+15	1.82e+16	98.2 %
timestepper_module_steppersteptransportdt_	2.22e+15	1.33e+16	72.0 %
loop at timestepper.F90: 1230	2.22e+15	1.33e+16	72.0 %
loop at timestepper.F90: 1254	2.22e+15	1.32e+16	71.3 %
snessolve_	2.22e+15	1.30e+16	70.4 %
SNESSolve	2.22e+15	1.30e+16	70.4 %
SNESSolve_LS	2.22e+15	1.30e+16	70.4 %
loop at ls.c: 181	2.15e+15	1.27e+16	68.8 %
SNES_KSPSolve	1.19e+15	6.44e+15	34.8 %
SNESComputeJacob	6.21e+14	4.07e+15	22.0 %

```
89 ierr = SNESComputeJacob(snes,X,&snes->jacobian,&snes->jacobian_pre,&
190 ierr = KSPSetOperators(snes->ksp,snes->jacobian,snes->jacobian_pre,flg)
191 ierr = SNES_KSPSolve(snes,snes->ksp,F,Y);CHKERRQ(ierr);
```

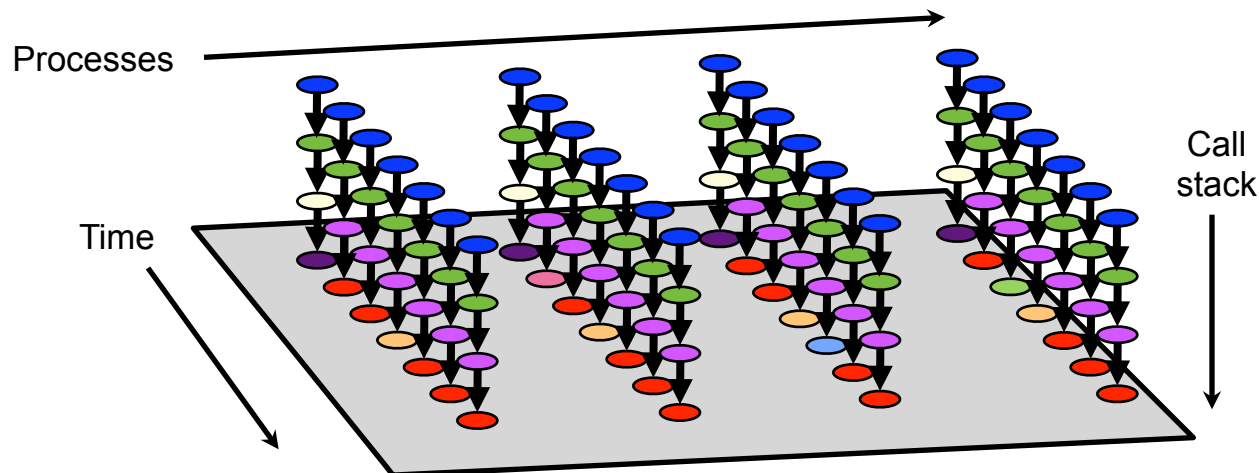


Outline

- **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**

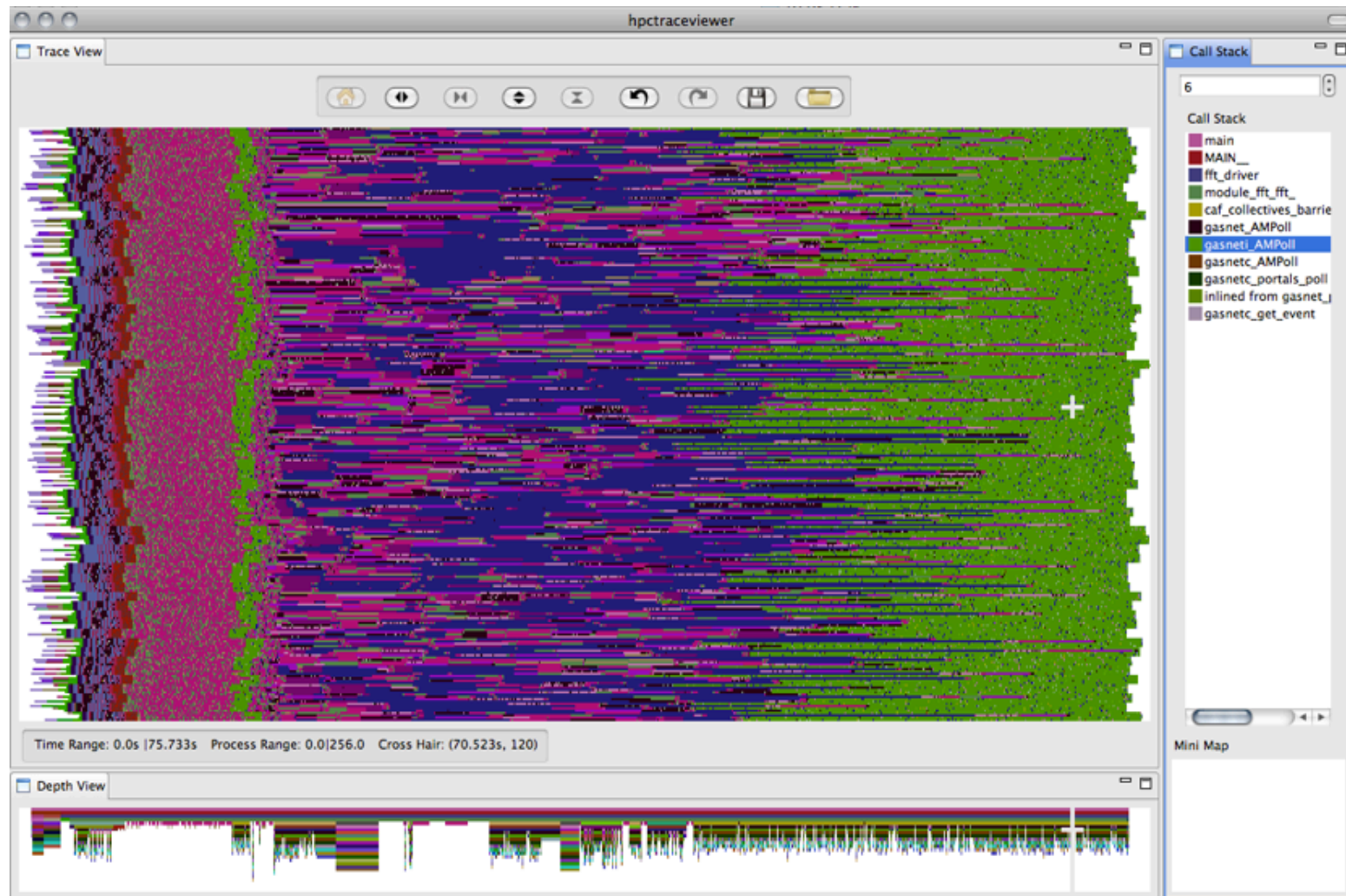
Understanding Temporal Behavior

- 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

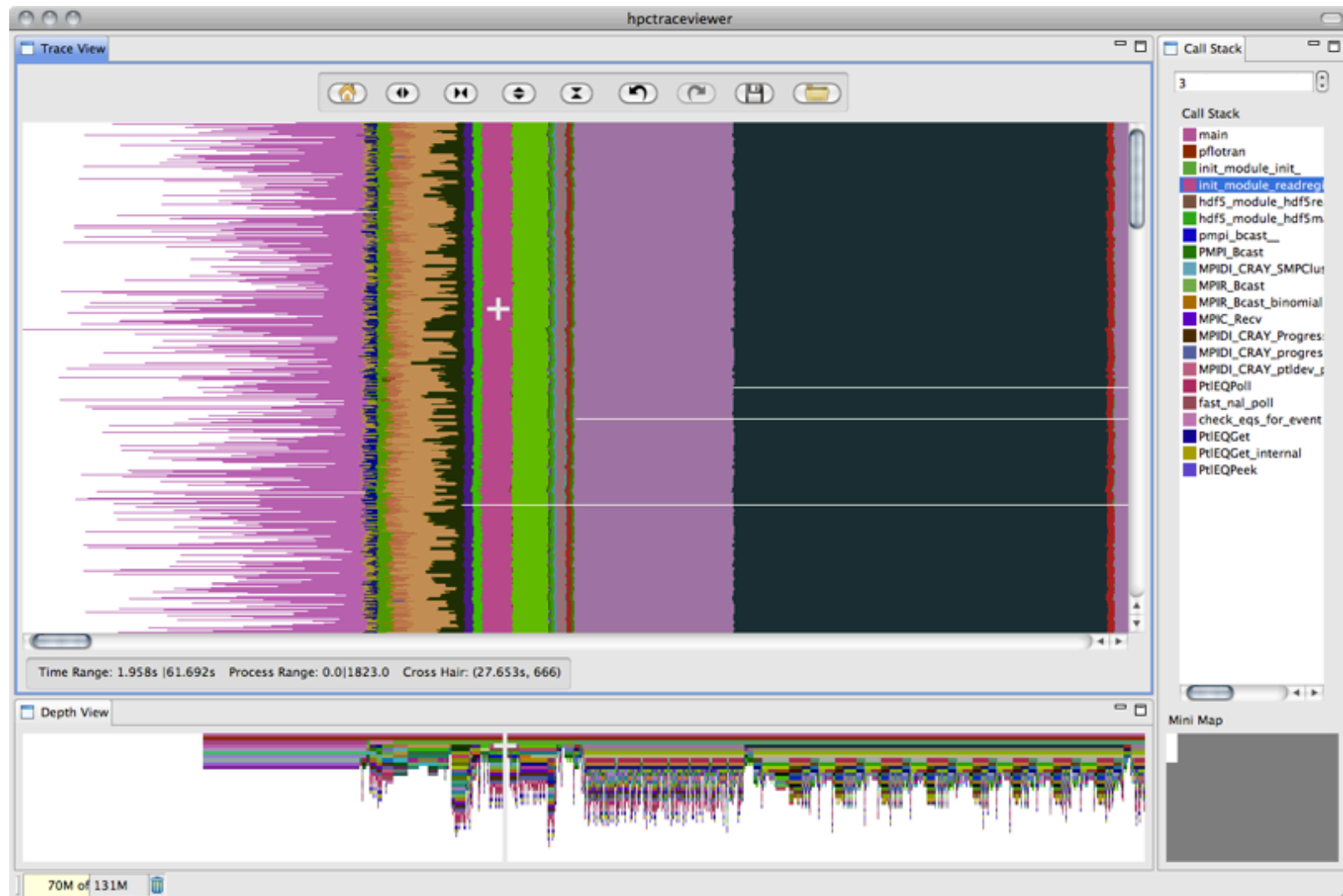


PFLOTRAN: Fortran+MPI, 8184 cores, Cray XT (982s)



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

The screenshot displays the hpcviewer interface for the s3d_f90.x application. The top pane shows Fortran code with a loop at line 389 highlighted. The bottom pane shows a performance table with columns for LATENCY, # of LD+ST, and CACHE_MISS. A blue arrow points from the code line to the table, and a red arrow points from the table to a callout box.

Code snippet (lines 389-391):

```
yspec(:) = yspecies(i, j, k, :)
```

Callout box (blue border):

yspecies latency for this loop is 14.5% of total latency in program

Callout box (red border):

41.2% of memory hierarchy latency related to yspecies array

Scope	LATENCY.[0,0] (v)	#(LD+ST).[0,0] (I)	#(LD+ST).[0,0] (E)	CACHE_MISS.[0,0] (I)
Experiment Aggregate Metrics	1.38e+05	100 %	5.02e+04	100 %
ALLOCATE_VARIABLES_ARRAYS.in.VARIABLES_M	5.68e+05	41.2 %	9.40e+03	18.7 %
solve_driver	5.66e+05	41.0 %	9.32e+03	18.6 %
loop at solve_driver.f90: 137	5.66e+05	41.0 %	9.32e+03	18.6 %
integrate	5.36e+05	38.9 %	8.51e+03	17.0 %
integrate_erk_jstage_it	5.36e+05	38.9 %	8.51e+03	17.0 %
loop at integrate_erk_jstage_it.gen.f: 47	5.17e+05	37.4 %	7.99e+03	15.9 %
rhsf	2.57e+05	18.6 %	1.24e+03	2.5 %
REACTION_RATE.in.CHEMKIN_M	2.57e+05	18.6 %	1.24e+03	2.5 %
REACTION_RATE_BOUNDS.in.CHEMKIN_M	2.57e+05	18.6 %	1.24e+03	2.5 %
loop at chemkin_m.f90: 385	2.57e+05	18.6 %	1.24e+03	2.5 %
loop at chemkin_m.f90: 386	2.57e+05	18.6 %	1.24e+03	2.5 %
loop at chemkin_m.f90: 387	2.00e+05	14.5 %	1.10e+03	2.2 %
loop at chemkin_m.f90: 389	2.00e+05	14.5 %	1.10e+03	2.2 %
chemkin_m.f90: 389	2.00e+05	14.5 %	1.10e+03	2.2 %

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

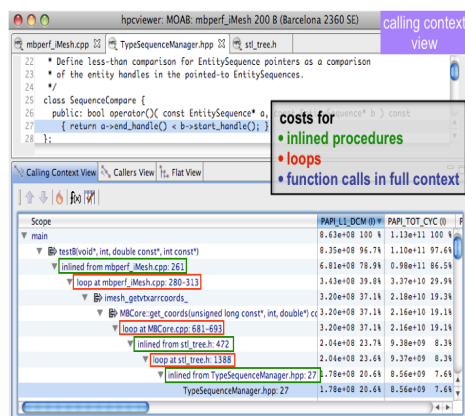
Some Challenges Ahead

- **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**

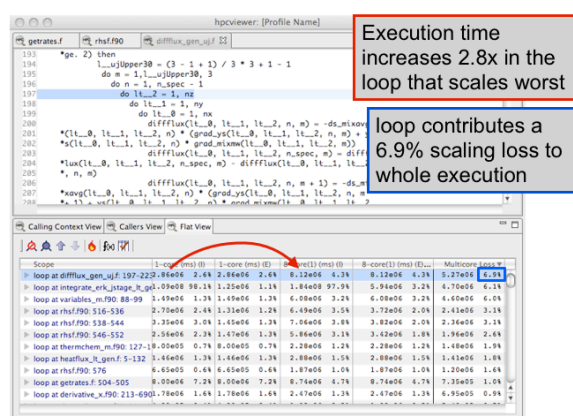
Some Related Work

- **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

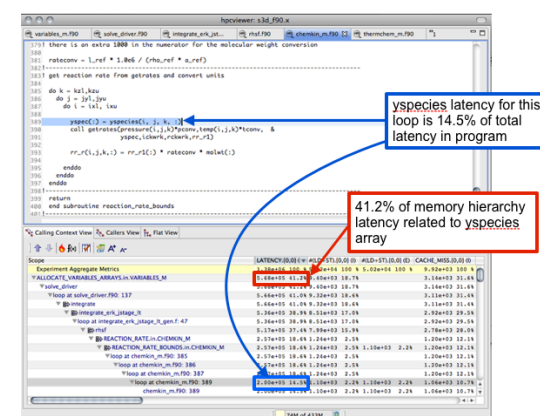
HPCToolkit Capabilities at a Glance



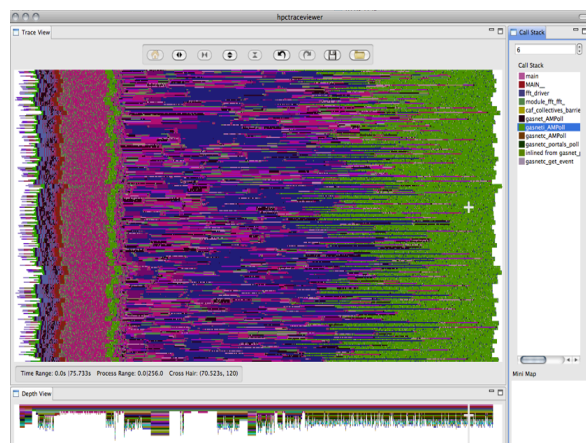
Attribute Costs to Code



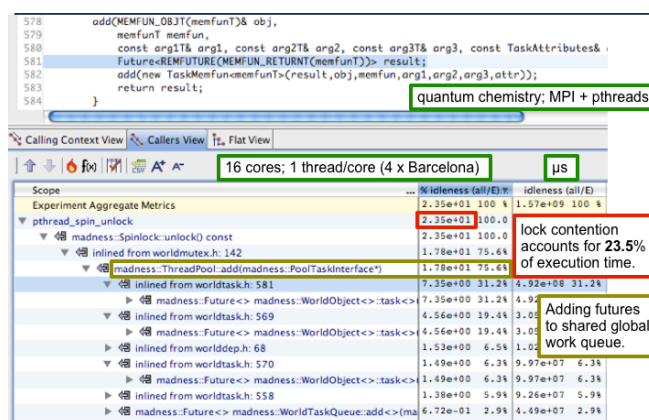
Pinpoint & Quantify Scaling Bottlenecks



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.

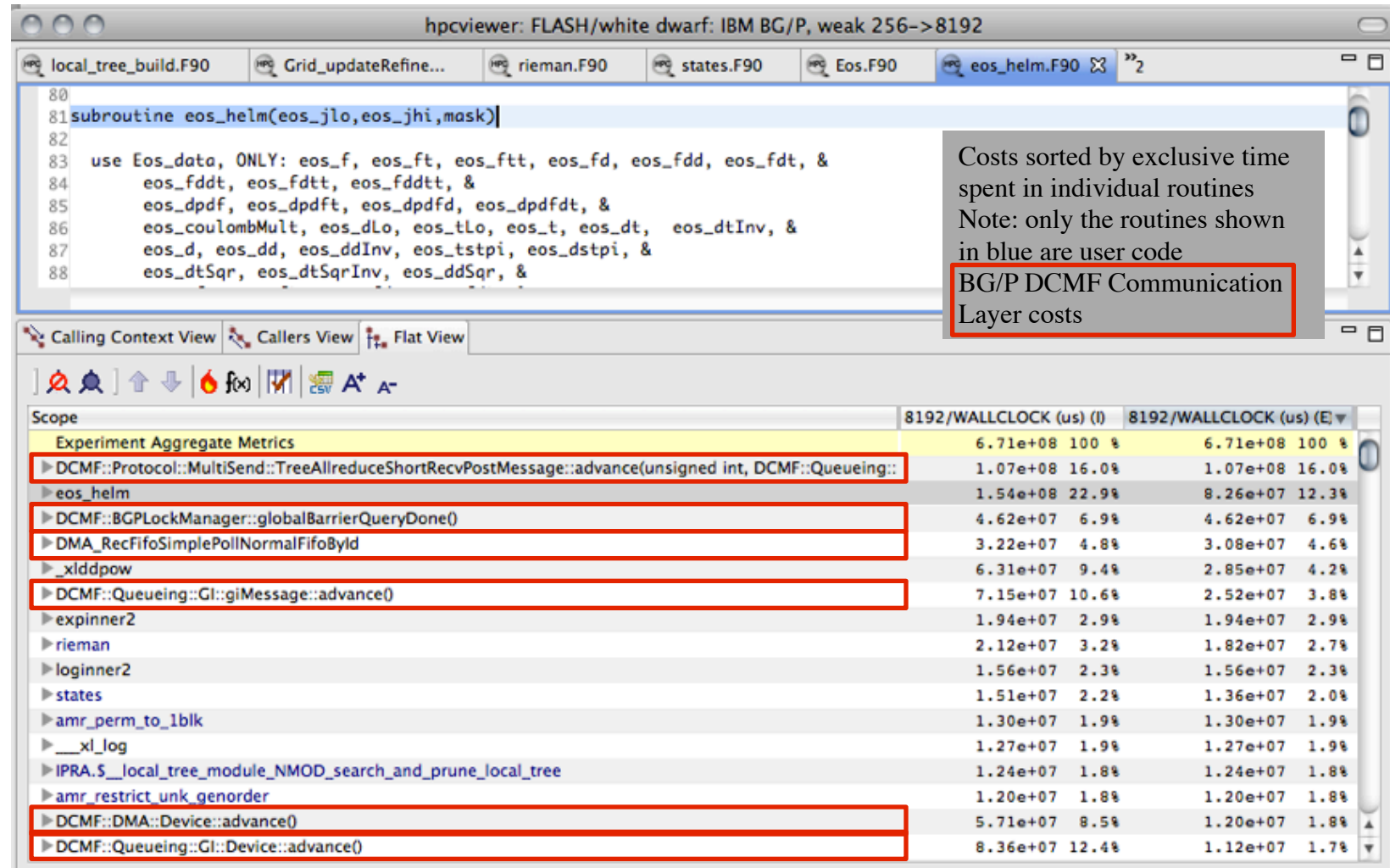
Overview Paper

*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

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

```
mpi_amr_comm_setup.F90
418   itemp = max(sum(commatrix_send), sum(commatrix_recv))
419   Call MPI_ALLREDUCE (itemp,
420                      max_blks_sent,
421                      1,
422                      MPI_INTEGER,
423                      MPI_MAX,
424                      MPI_COMM_WORLD,
425                      ierror)
```

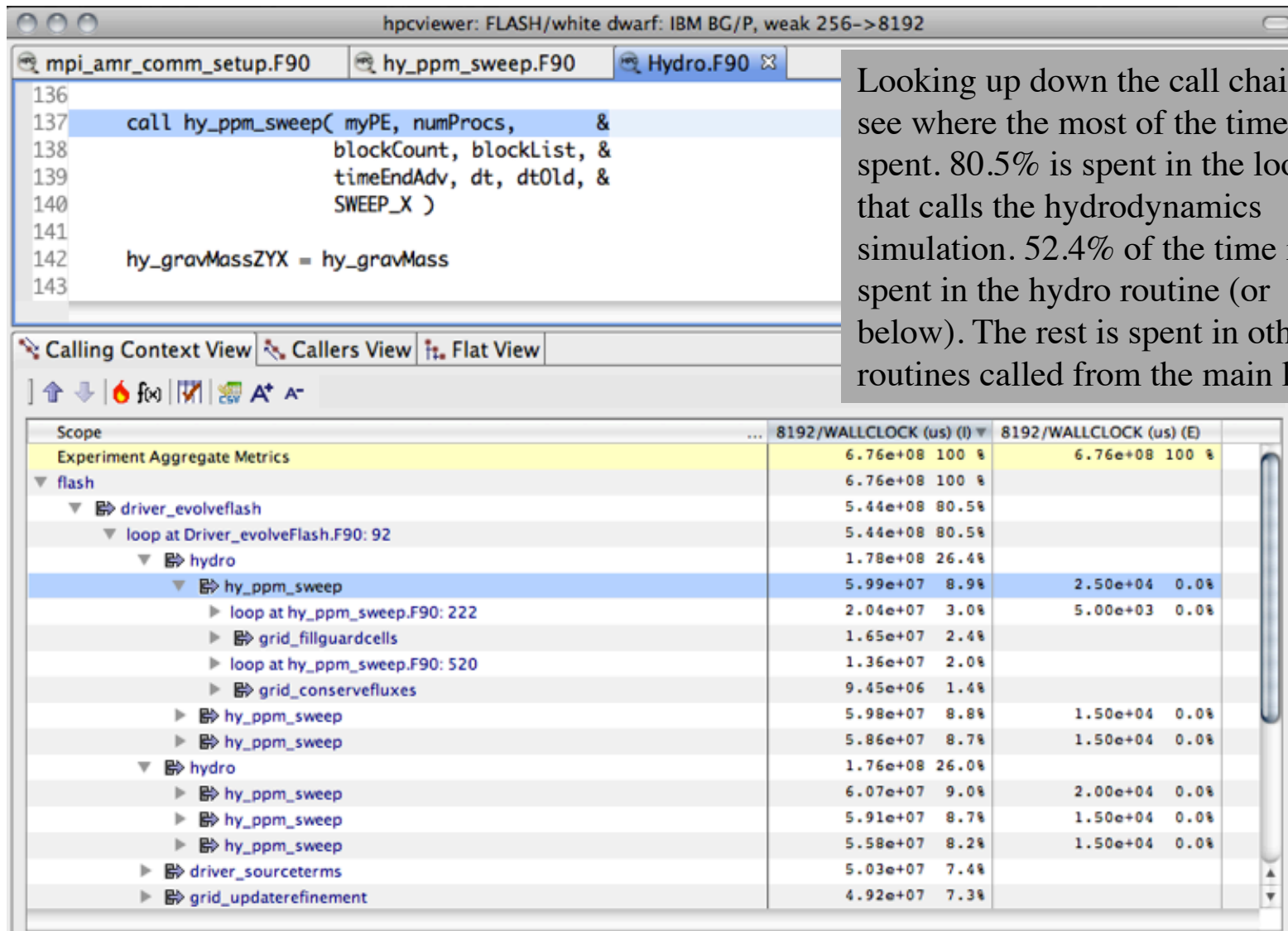
Looking up the call chain to see where the callers that caused costs to be incurred for tree reductions. Most of the cost is incurred by guard cell filling and flux conservation.

Calling Context View Callers View Flat View

Scope	8192/WALLCLOCK (us) (I)	8192/WALLCLOCK (us) (E)
Experiment Aggregate Metrics	6.76e+08 100 %	6.76e+08 100 %
DCMF::Protocol::MultiSend::TreeAllreduceShortRecvPostMessage::advance(unsigned int, DCMF::Queueing::Tree::TreeMsgContext)	1.07e+08 15.9 %	1.07e+08 15.9 %
inlined from Device.cc: 432	1.07e+08 15.9 %	1.07e+08 15.9 %
DCMF::Queueing::Tree::Device::postRecv(DCMF::Queueing::Tree::TreeRecvMessage&)	1.07e+08 15.9 %	1.07e+08 15.9 %
inlined from Message.h: 516	1.07e+08 15.9 %	1.07e+08 15.9 %
DCMF_GlobalAllreduce	1.07e+08 15.9 %	1.07e+08 15.9 %
MPIDO_Allreduce_global_tree	1.05e+08 15.5 %	1.05e+08 15.5 %
MPIDO_Allreduce	1.05e+08 15.5 %	1.05e+08 15.5 %
PMPI_Allreduce	1.05e+08 15.5 %	1.05e+08 15.5 %
pmpi_allreduce	1.05e+08 15.5 %	1.05e+08 15.5 %
mpi_amr_comm_setup	9.51e+07 14.1 %	9.51e+07 14.1 %
amr_flux_conserve_udt	5.84e+07 8.6 %	5.84e+07 8.6 %
amr_guardcell	3.63e+07 5.4 %	3.63e+07 5.4 %
amr_flux_conserve_udt	3.45e+05 0.1 %	3.45e+05 0.1 %
mpi_amr_1blk_restrict	6.50e+04 0.0 %	6.50e+04 0.0 %
amr_refine_derefine	5.04e+06 0.7 %	5.04e+06 0.7 %
driver_computedt	2.08e+06 0.3 %	2.08e+06 0.3 %
mpi_morton_bnd	1.58e+06 0.2 %	1.58e+06 0.2 %
driver_verifyinitdt	9.70e+05 0.1 %	9.70e+05 0.1 %

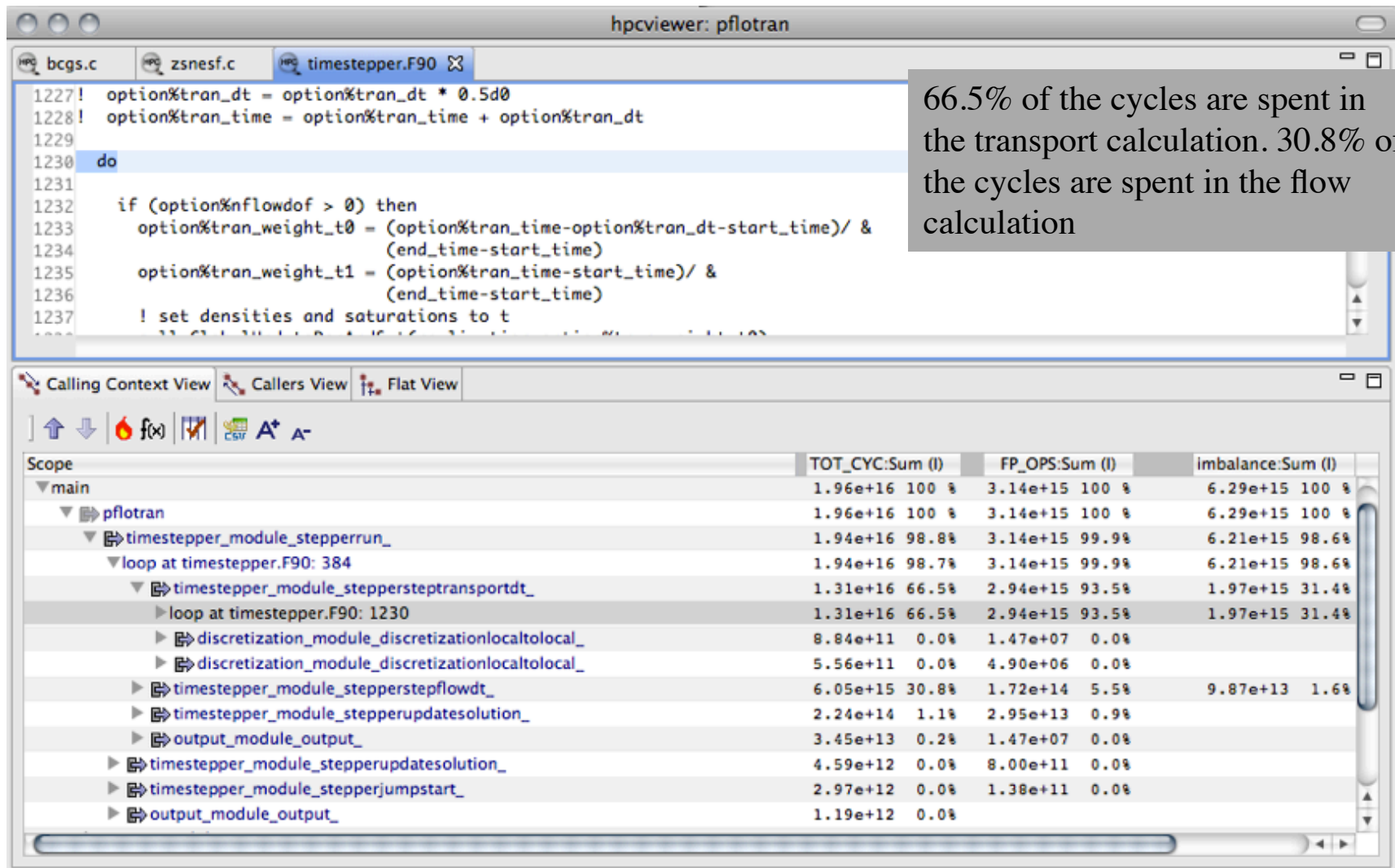
Execution Cost Attribution (Top Down)

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



Execution Cost Attribution (Top-Down)

PFLOTRAN, Cray XT, 8184 cores, Hanford problem



Execution Cost Attribution (Top-Down)

PFLOTRAN, Cray XT, 8184 cores, Hanford problem

