

# Gaining Insight into Parallel Program Performance Using Sampling

**John Mellor-Crummey**

**DOE Center for Scalable Application Development Software**

**[johnmc@cs.rice.edu](mailto: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

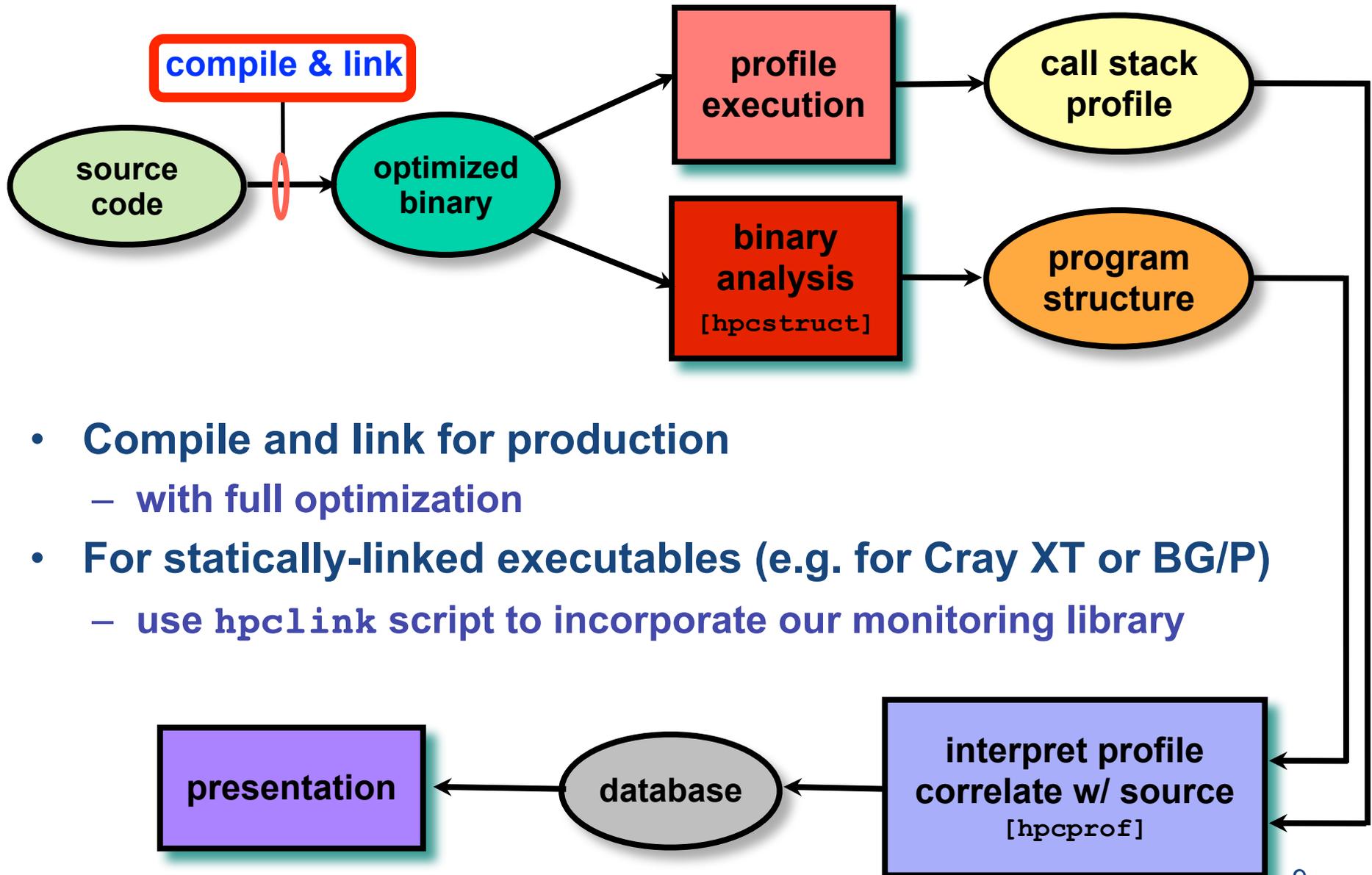
- **Understand where and why performance losses occur in sophisticated parallel codes on complex parallel hardware**
- **Identify opportunities for improvement**
- **Quantify potential benefits**

# Performance Analysis Goals

---

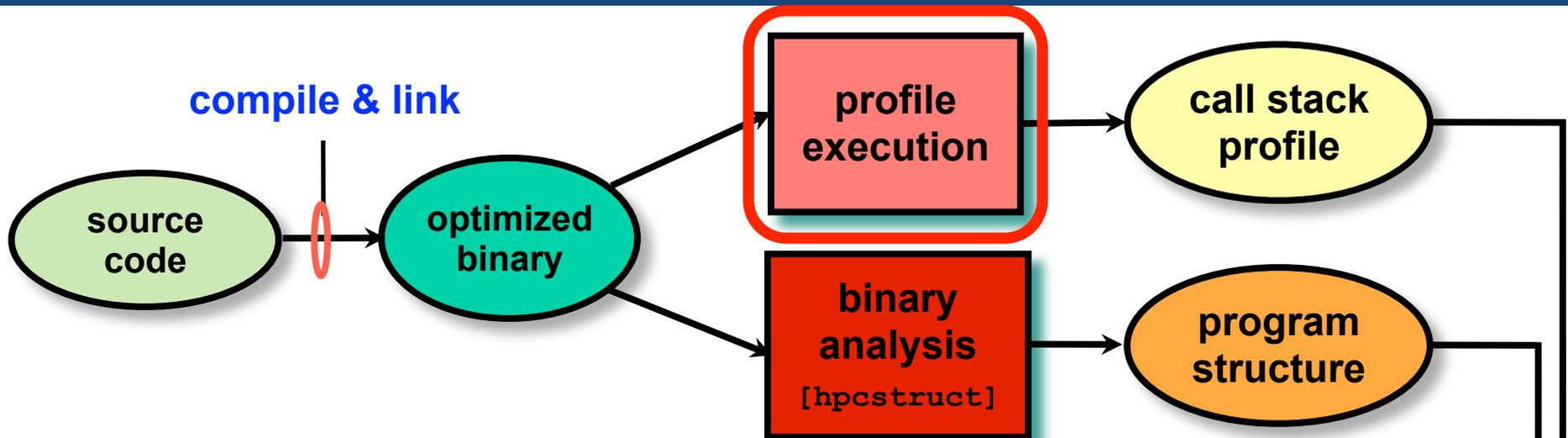
- **Accurate measurement of parallel scientific codes**
  - large, multi-lingual programs
  - fully optimized code: loop optimization, templates, inlining
  - binary-only libraries, sometimes partially stripped
  - complex execution environments
    - dynamic loading or static binaries
    - SPMD parallel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



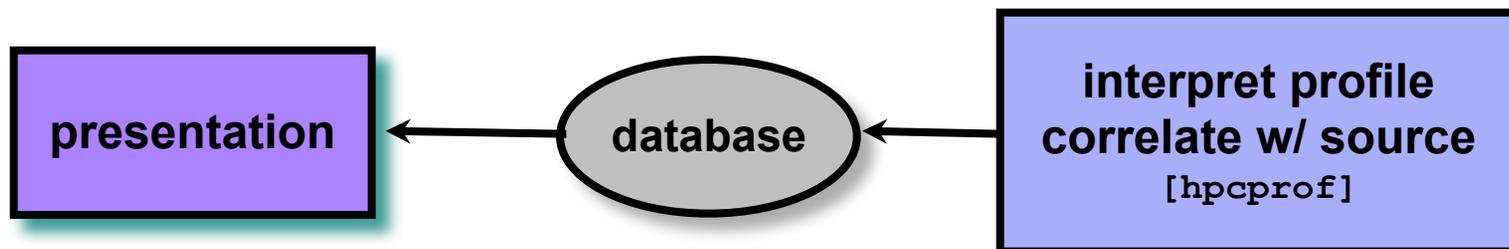
- **Compile and link for production**
  - with full optimization
- **For statically-linked executables (e.g. for Cray XT or BG/P)**
  - use `hpcLink` script to incorporate our monitoring library

# HPCToolkit Performance Tools

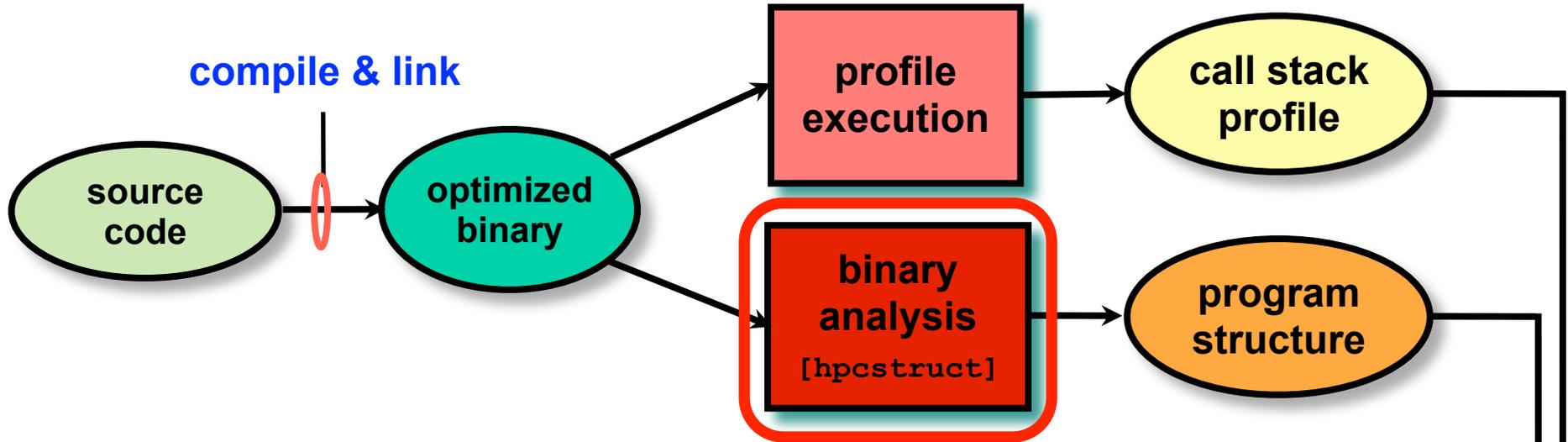


## Measure execution unobtrusively

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

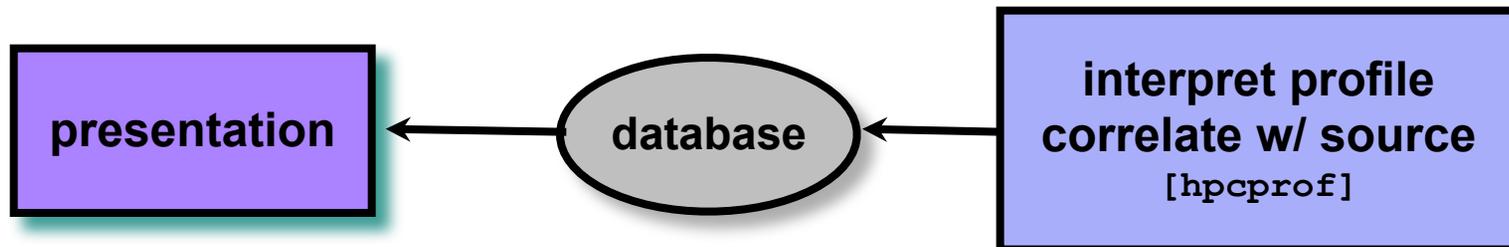


# HPCToolkit Performance Tools

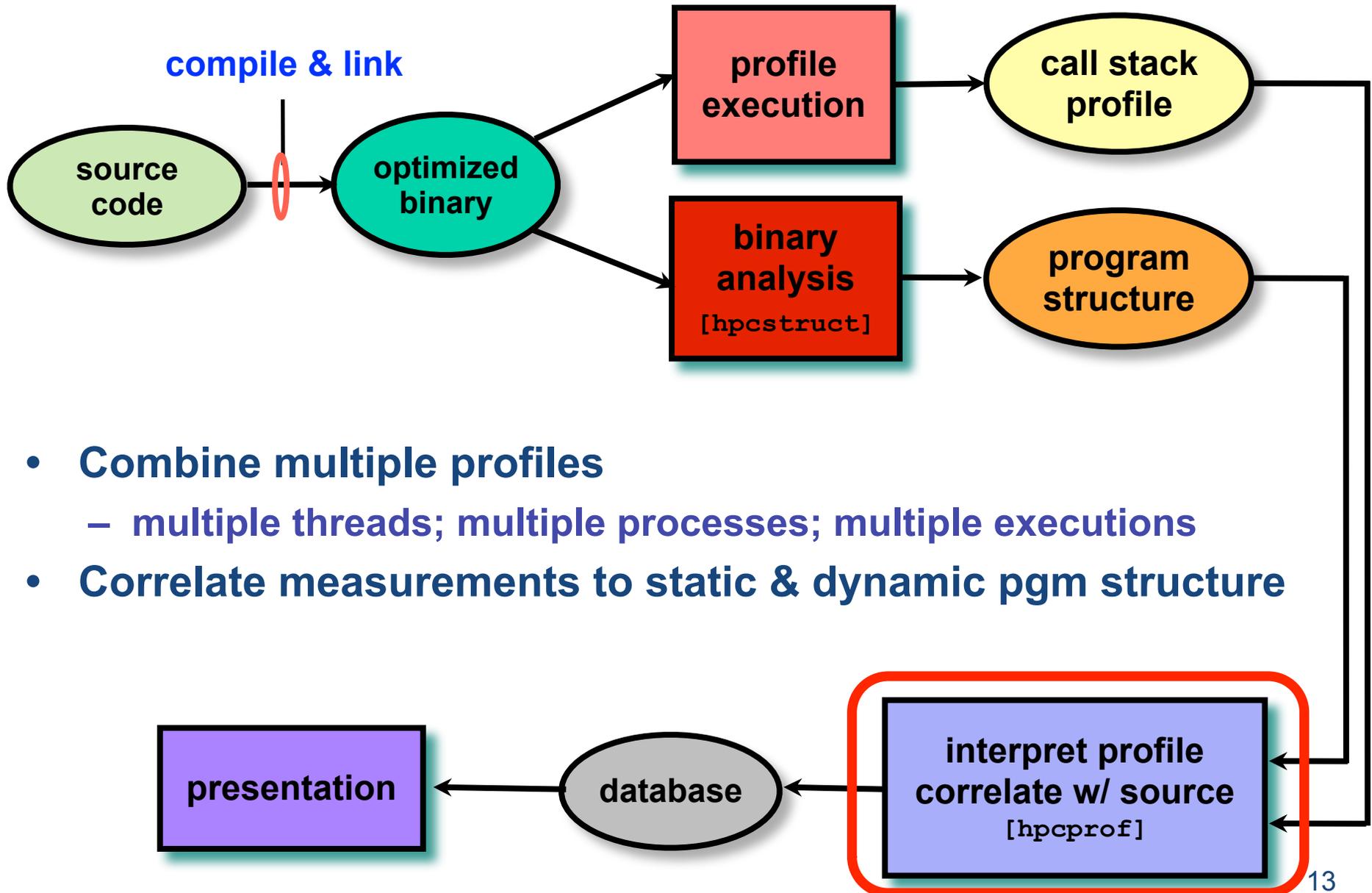


## Analyze binary to recover program structure

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

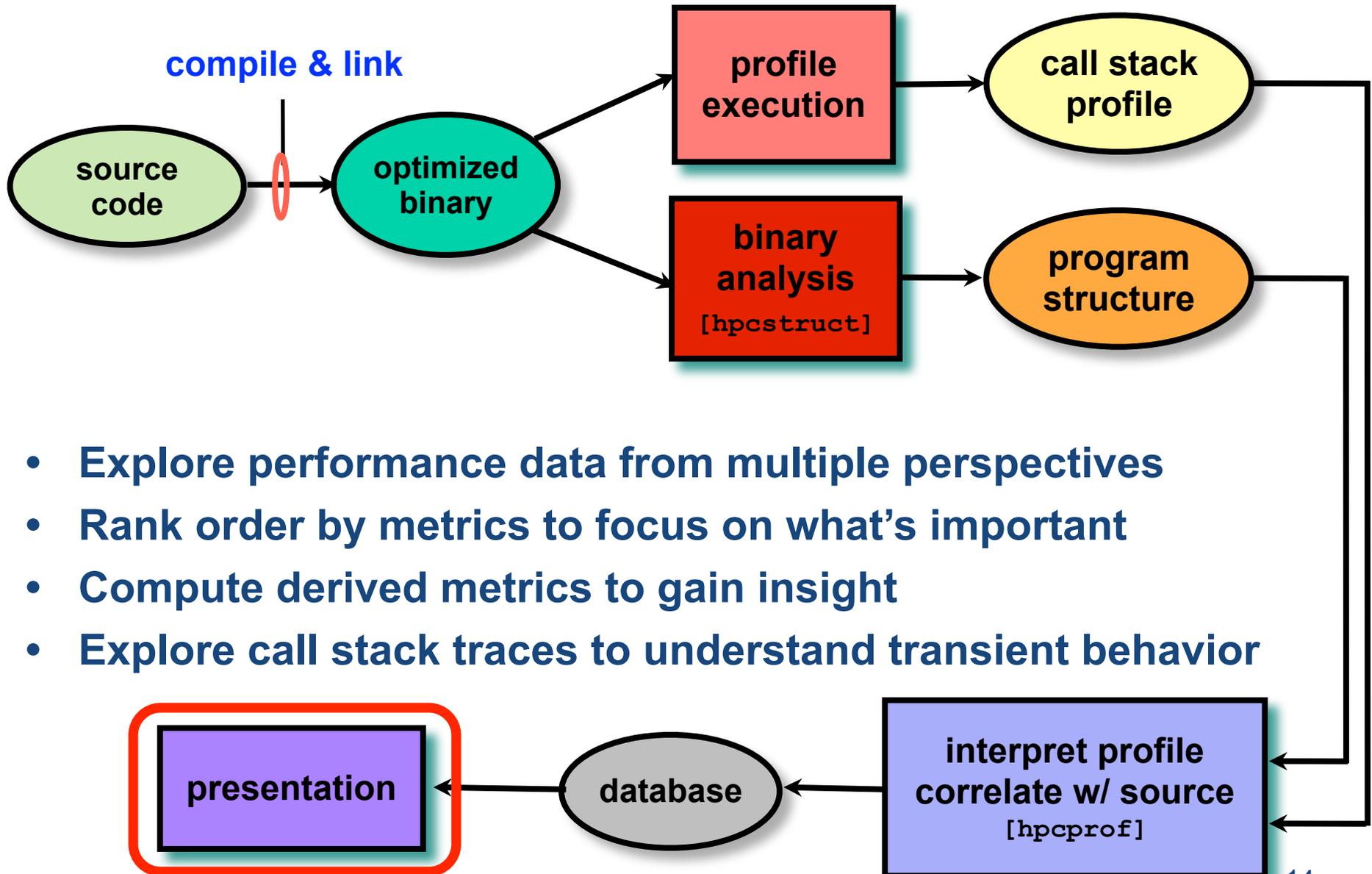


# HPCToolkit Performance Tools



- **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

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

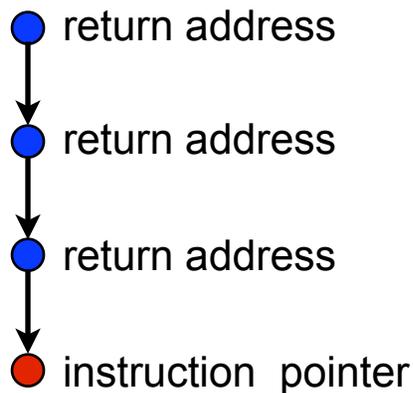
- **Call path profiling in HPCToolkit**
- **Pinpointing and quantifying scalability bottlenecks**
- **Blame shifting**
  - analyzing multithreaded computations based on work stealing
  - quantifying the impact of lock contention on threaded code
  - pinpointing load imbalance in parallel codes
- **Understanding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

# Call Path Profiling

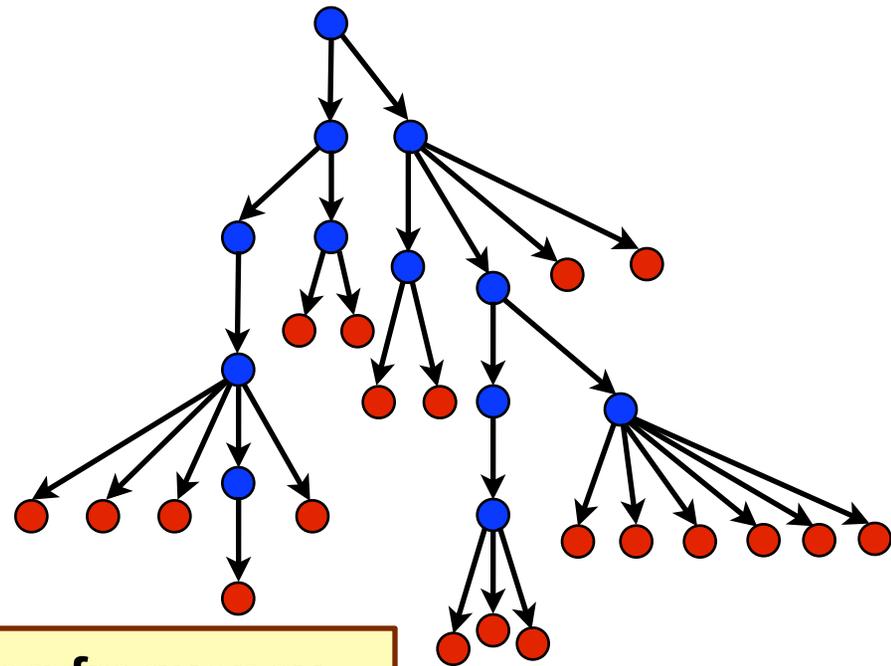
Measure and attribute costs in their *calling* context

- 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.**

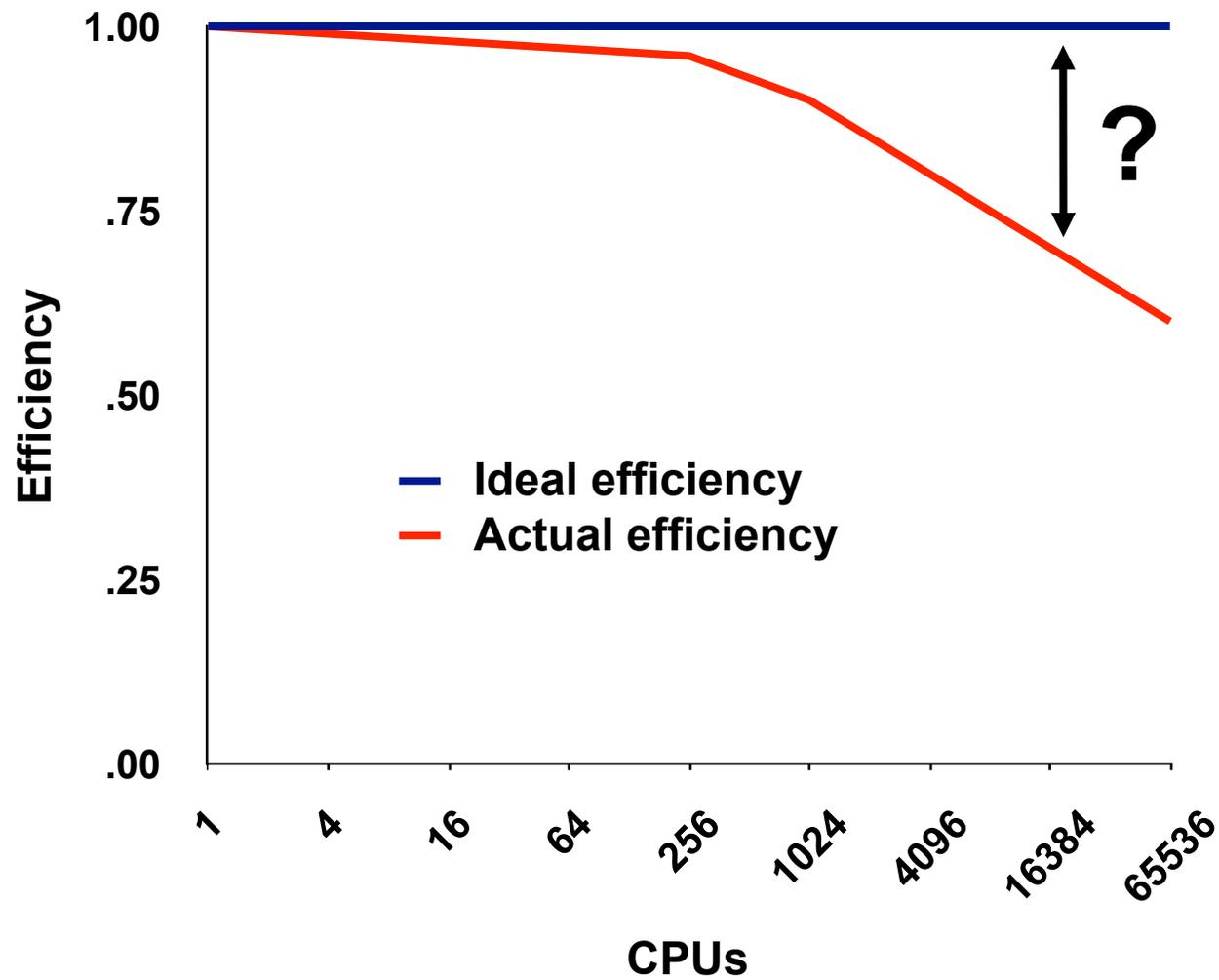
# Outline

---

- **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**

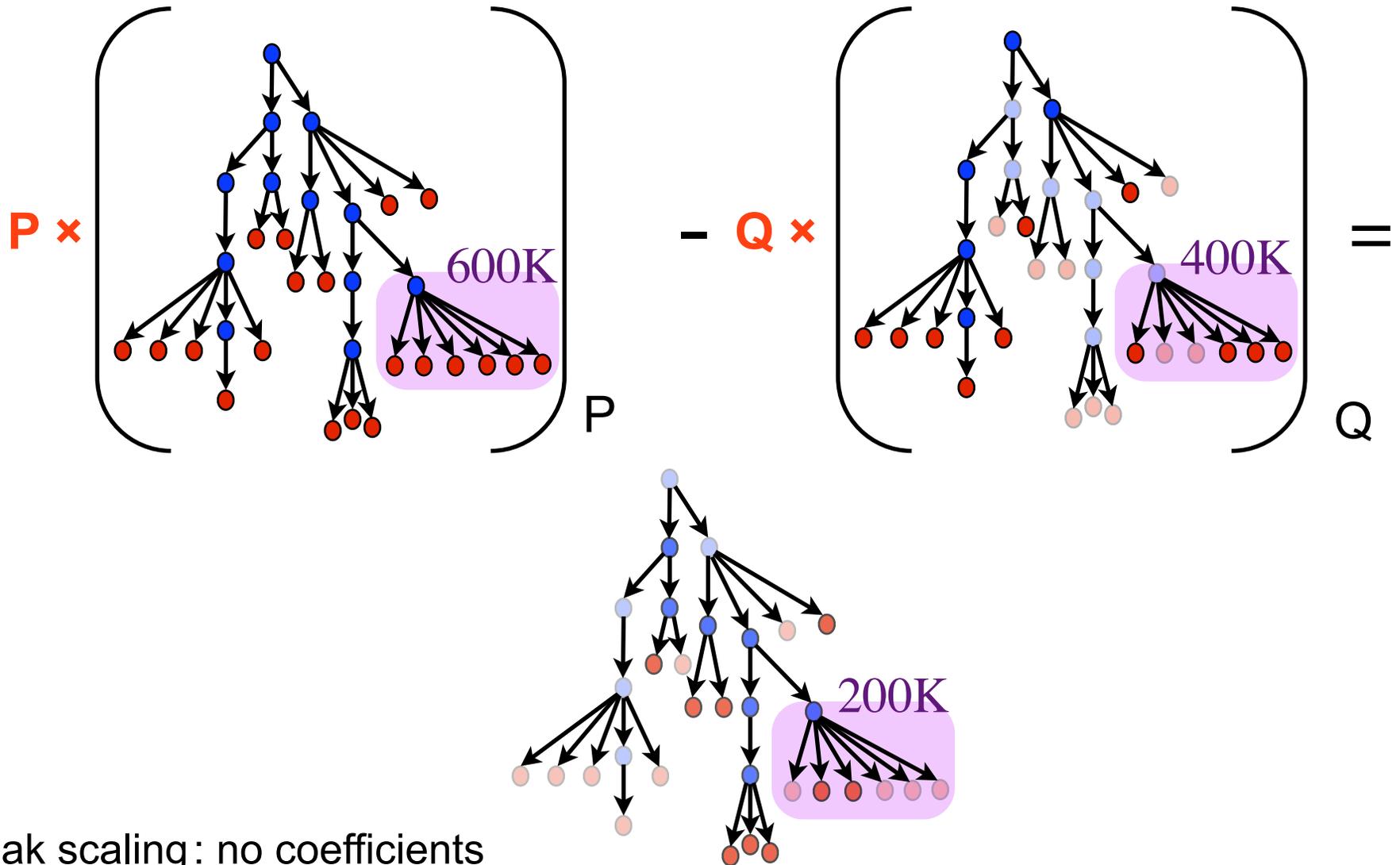
# The Problem of Scaling Losses

---



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:**

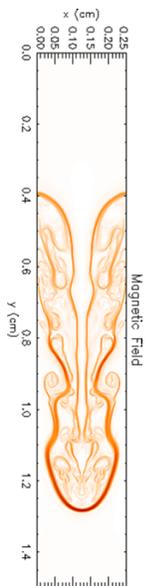
**University of Chicago FLASH**

**white dwarf detonation**

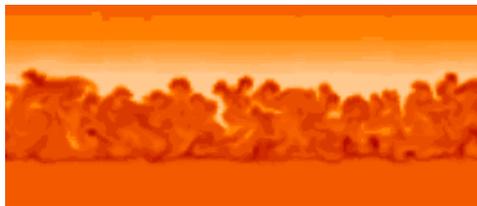
**Blue Gene/P**

**8192 vs. 256 processors**

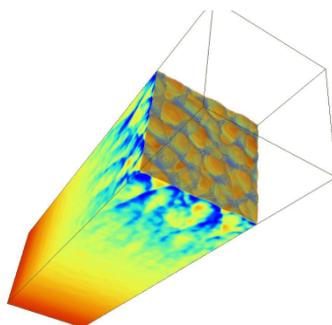
**weak**



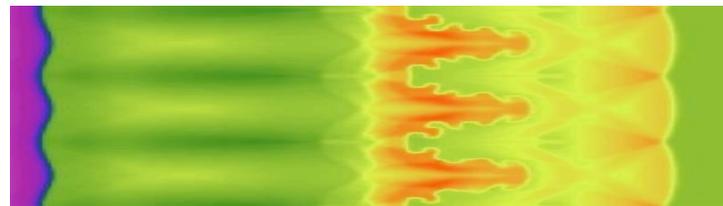
*Magnetic  
Rayleigh-Taylor*



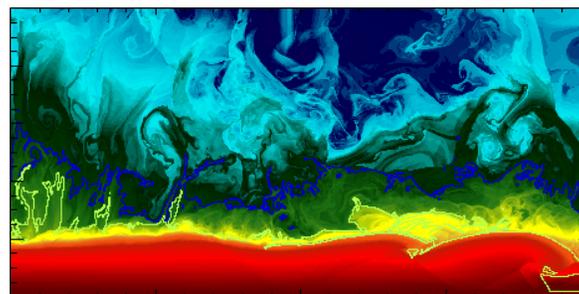
*Nova outbursts on white dwarfs*



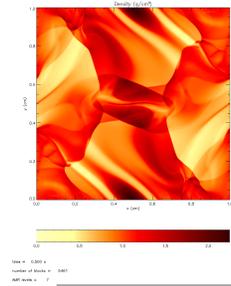
*Cellular detonation*



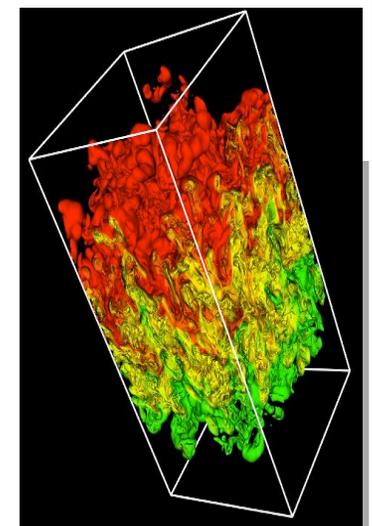
*Laser-driven shock instabilities*



*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

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

```
Driver_initFlash.F90 local_tree_build.F90 Driver_evolveFlash.F90
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
```

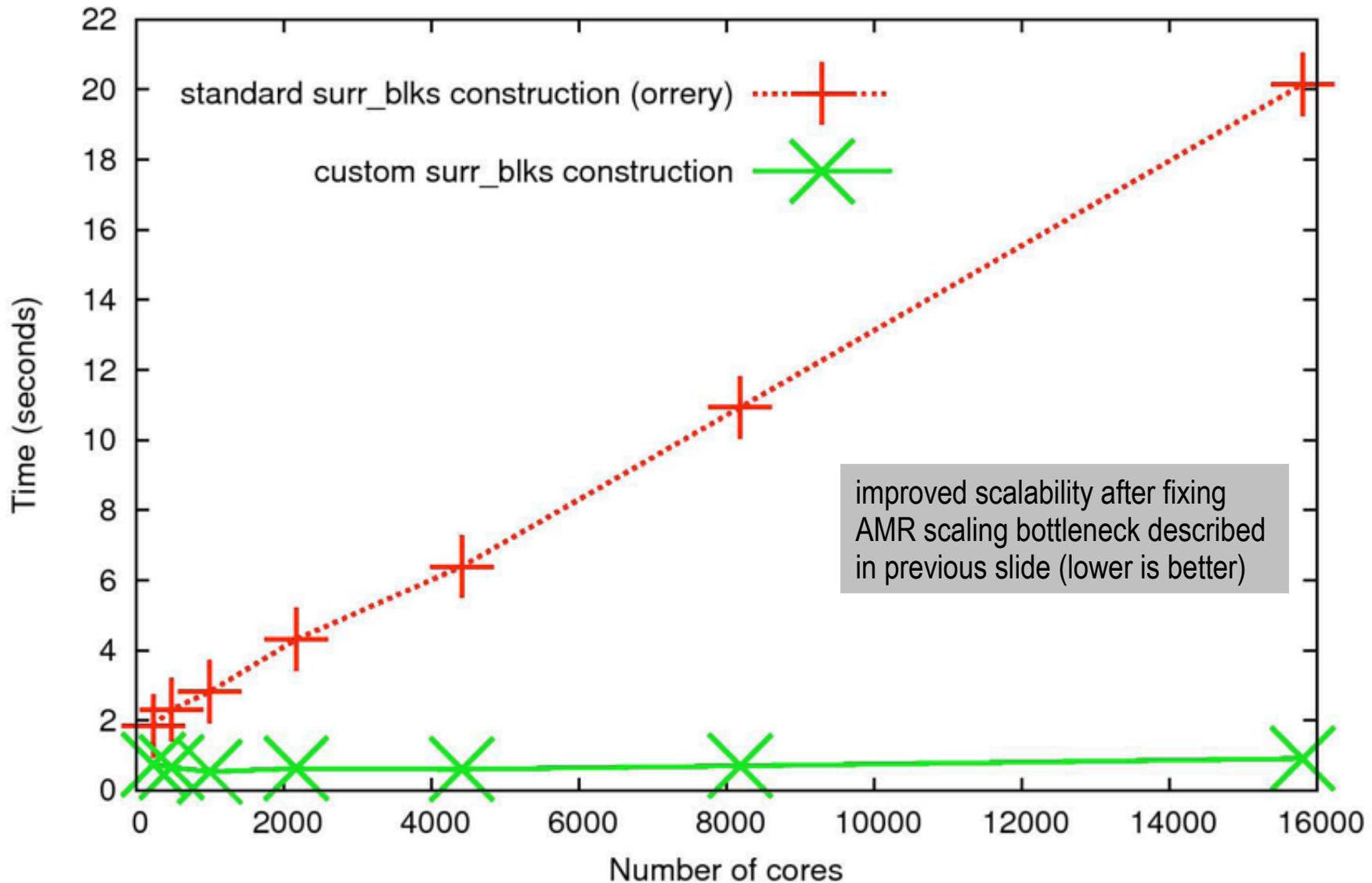
Calling Context View Callers View Flat View

Scope	% scalality 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%
pmi_sendrecv_replace	4.42e-01 1.8%	2.50e+04 0.0%	2.99e+06 0.4%

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

hpcviewer: [Profile Name]

getrates.f rhsf.f90 diffflux\_gen\_uj.f

```

193 *ge. 2) then
194   l__ujUpper30 = (3 - 1 + 1) / 3 * 3 + 1 - 1
195   do m = 1, l__ujUpper30, 3
196     do n = 1, n_spec - 1
197       do lt__2 = 1, nz
198         do lt__1 = 1, ny
199           do lt__0 = 1, nx
200             diffflux(lt__0, lt__1, lt__2, n, m) = -ds_mixavg
201             *(lt__0, lt__1, lt__2, n) * (grad_ys(lt__0, lt__1, lt__2, n, m) + y
202             *s(lt__0, lt__1, lt__2, n) * grad_mixmw(lt__0, lt__1, lt__2, m))
203             diffflux(lt__0, lt__1, lt__2, n_spec, m) = diff
204             *lux(lt__0, lt__1, lt__2, n_spec, m) - diffflux(lt__0, lt__1, lt__2
205             *, n, m)
206             diffflux(lt__0, lt__1, lt__2, n, m + 1) = -ds_mi
207             *xavg(lt__0, lt__1, lt__2, n) * (grad_ys(lt__0, lt__1, lt__2, n, m
208             * + 1) + ys(lt__0, lt__1, lt__2, n) * grad_mixmw(lt__0, lt__1, lt__2

```

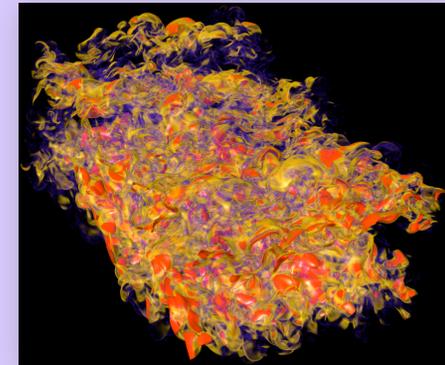
Calling Context View Callers View Flat View

Scope	1-core (ms) (I)	1-core (ms) (E)	8-core(1) (ms) (I)	8-core(1) (ms) (E)...	Multicore Loss
▶ loop at diffflux_gen_uj.f: 197-222	2.86e06 2.6%	2.86e06 2.6%	8.12e06 4.3%	8.12e06 4.3%	5.27e06 6.9%
▶ loop at integrate_erk_jstage_lt_ge	1.09e08 98.1%	1.25e06 1.1%	1.84e08 97.9%	5.94e06 3.2%	4.70e06 6.1%
▶ loop at variables_m.f90: 88-99	1.49e06 1.3%	1.49e06 1.3%	6.08e06 3.2%	6.08e06 3.2%	4.60e06 6.0%
▶ loop at rhsf.f90: 516-536	2.70e06 2.4%	1.31e06 1.2%	6.49e06 3.5%	3.72e06 2.0%	2.41e06 3.1%
▶ loop at rhsf.f90: 538-544	3.35e06 3.0%	1.45e06 1.3%	7.06e06 3.8%	3.82e06 2.0%	2.36e06 3.1%
▶ loop at rhsf.f90: 546-552	2.56e06 2.3%	1.47e06 1.3%	5.86e06 3.1%	3.42e06 1.8%	1.96e06 2.6%

Execution time increases 2.8x in the loop that scales worst

loop contributes a 6.9% scaling loss to the execution

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**

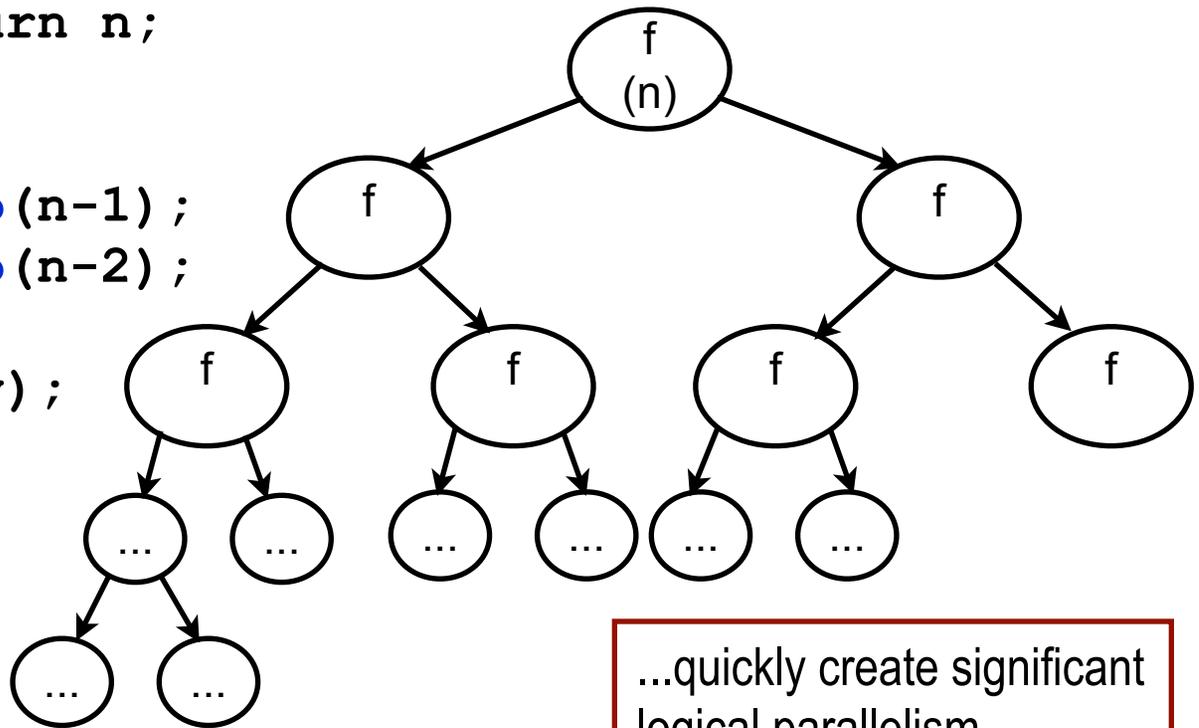
# Blame Shifting

---

- **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);  
  }  
}
```

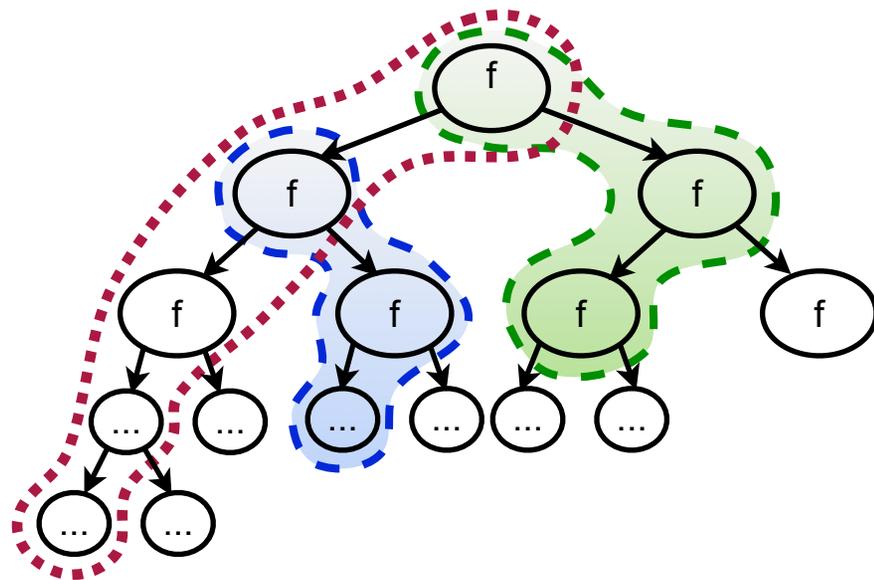


asynchronous calls  
create logical tasks that  
only block at a **sync**...

...quickly create significant  
logical parallelism.



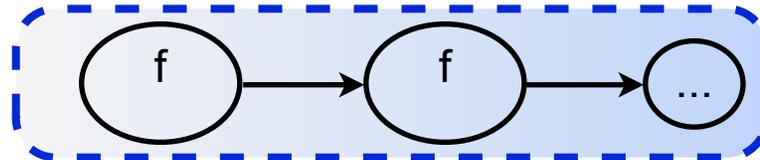
# Wanted: Call Path Profiles of Cilk



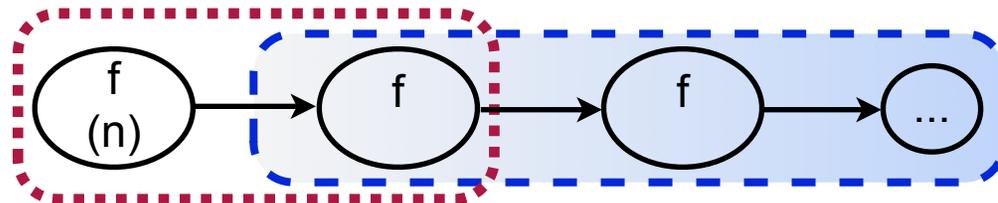
thread 1  
thread 2  
thread 3

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

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



- logical call path:



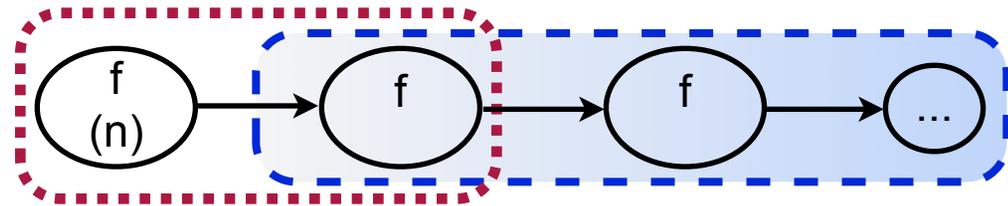
**Logical call path profiling: Recover *full* relationship  
between *physical* and *user-level* execution**

# 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**

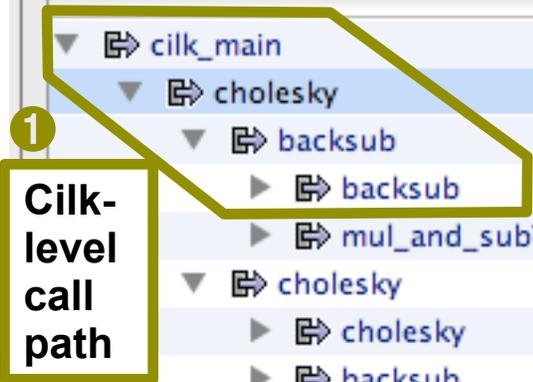
# Top-down Work for Cilk 'Cholesky'

```

hpcviewer: cholesky (dual Barcelona)[--nproc 8 -n 3000 -z 30000]
cholesky.cilk invoke-main.c cilk.c
650/*
651 * Compute Cholesky factorization of A.
652 */
653 cilk Matrix cholesky(int depth, Matrix a)
654 {

```

13.5% of cilk\_main's total effort was spent in idleness... ③



2.97% and 0.215% of cholesky's total effort was spent in idleness and overhead. ②

...	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%	1.8e-01 22.1%
▶ cholesky	3.78e+09 7.1%	2.70e+00 19.6%	1.5e-01 18.6%
▶ backsub	3.15e+09 5.9%	8.41e-02 0.6%	2.2e-02 2.7%
▶ mul_and_subT	3.01e+09 5.6%	1.62e-02 0.1%	7.4e-03 0.9%
▶ 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%	

# Bottom-up Idleness for Cilk 'Cholesky'

The screenshot shows the hpcviewer interface for a Cilk program named 'cholesky'. The code editor displays the `free_matrix` function, which is recursive and uses `cilk_spawn` to parallelize the work. The function signature is `void free_matrix(int depth, Matrix a)`. The code includes a null check, a return statement, and a recursive call to `free_matrix` on child nodes.

Below the code editor, the 'Flat View' is selected, showing a table of performance metrics for various scopes. The table has columns for 'Scope', 'work (all)', 'percent idleness', and 'percent overhead'. The 'percent idleness' column is highlighted in blue, and several values are circled in red to indicate high idleness.

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

↑ ↓ 🔥 f(x) 📄 CSV A+ A-

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

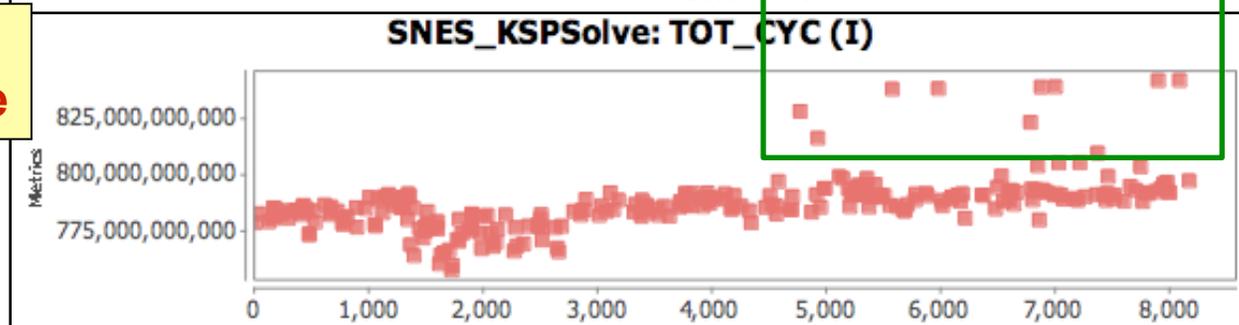
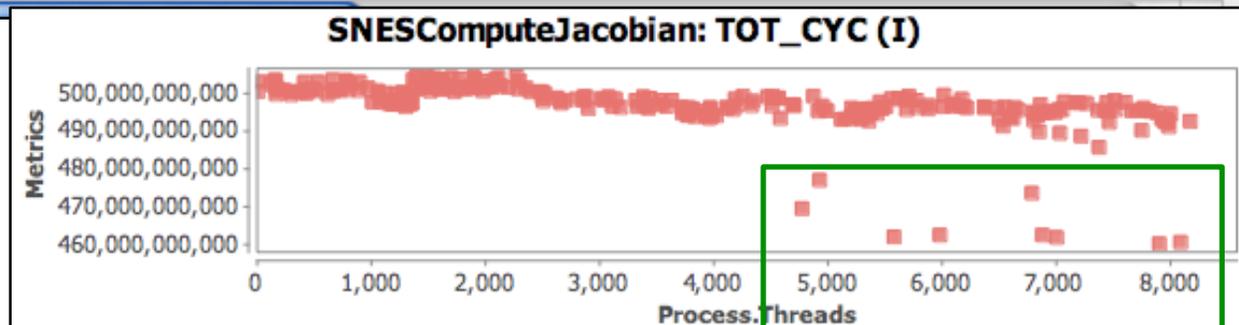
3. Plot the per-process values:

Early finishers...

... become early arrivers at **Allreduce**

Process	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%

```
189 ierr = SNESComputeJacobian(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);
```



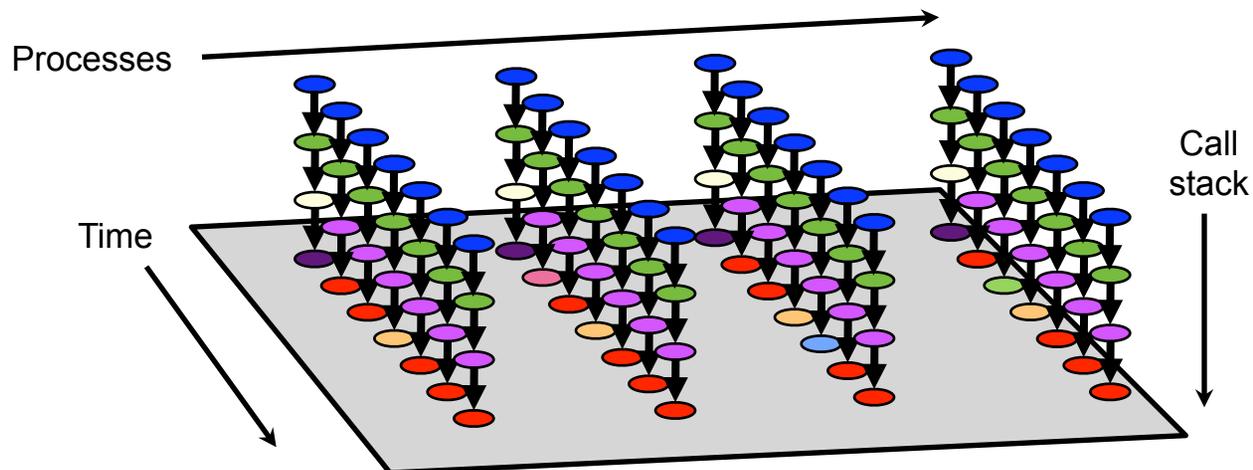
# Outline

---

- **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**

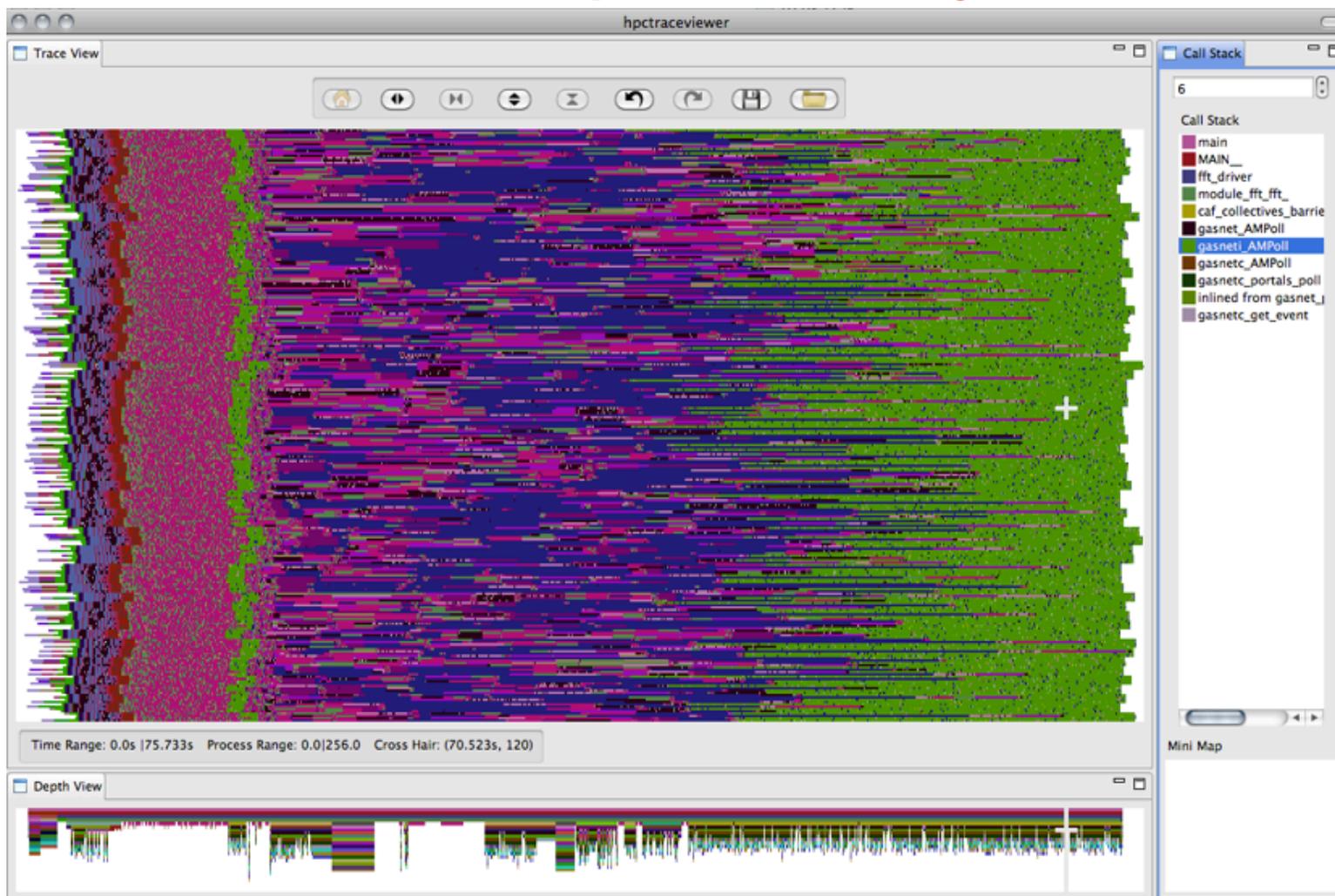
# Understanding Temporal Behavior

- 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



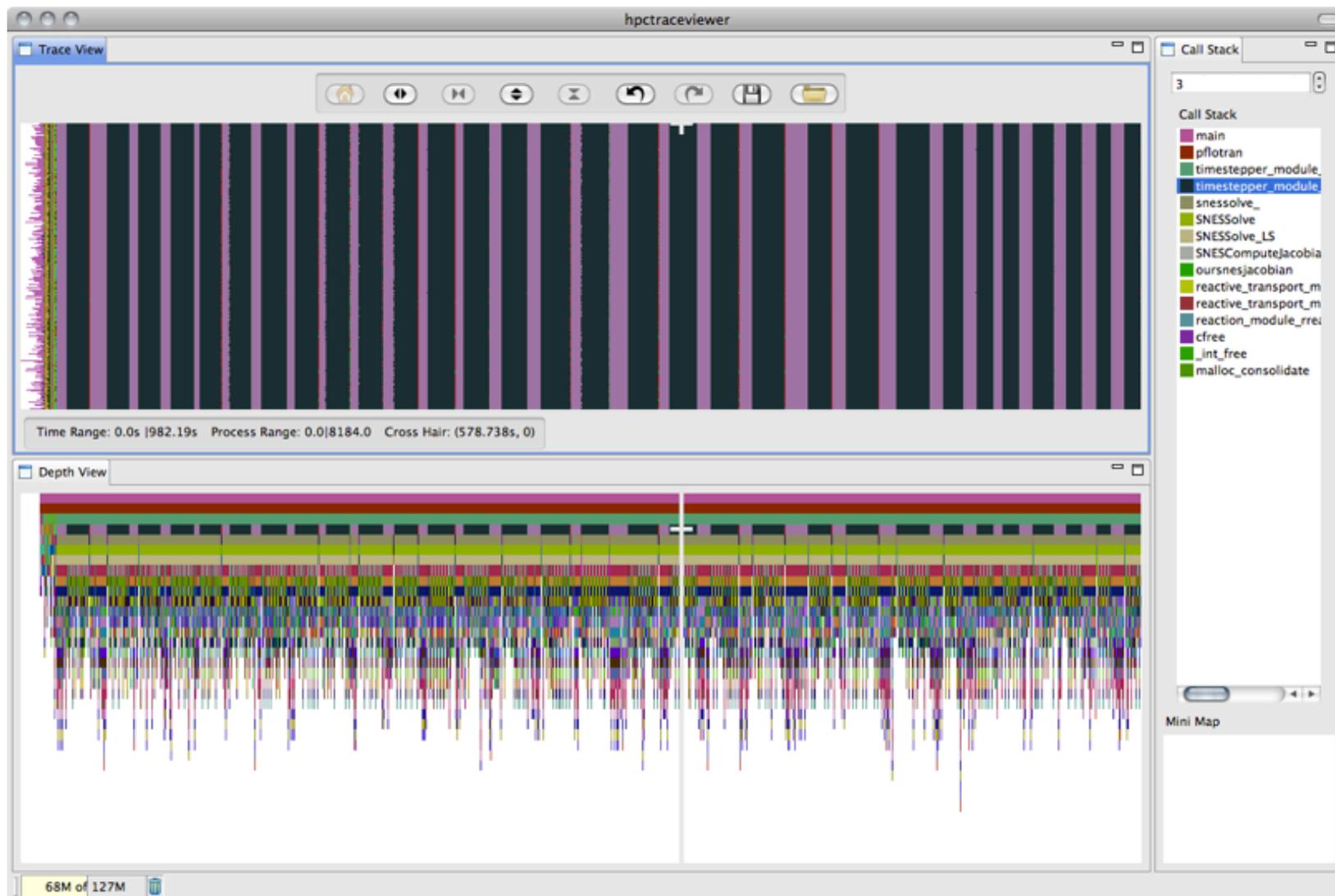
# Call Path Tracing for Parallel Programs

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



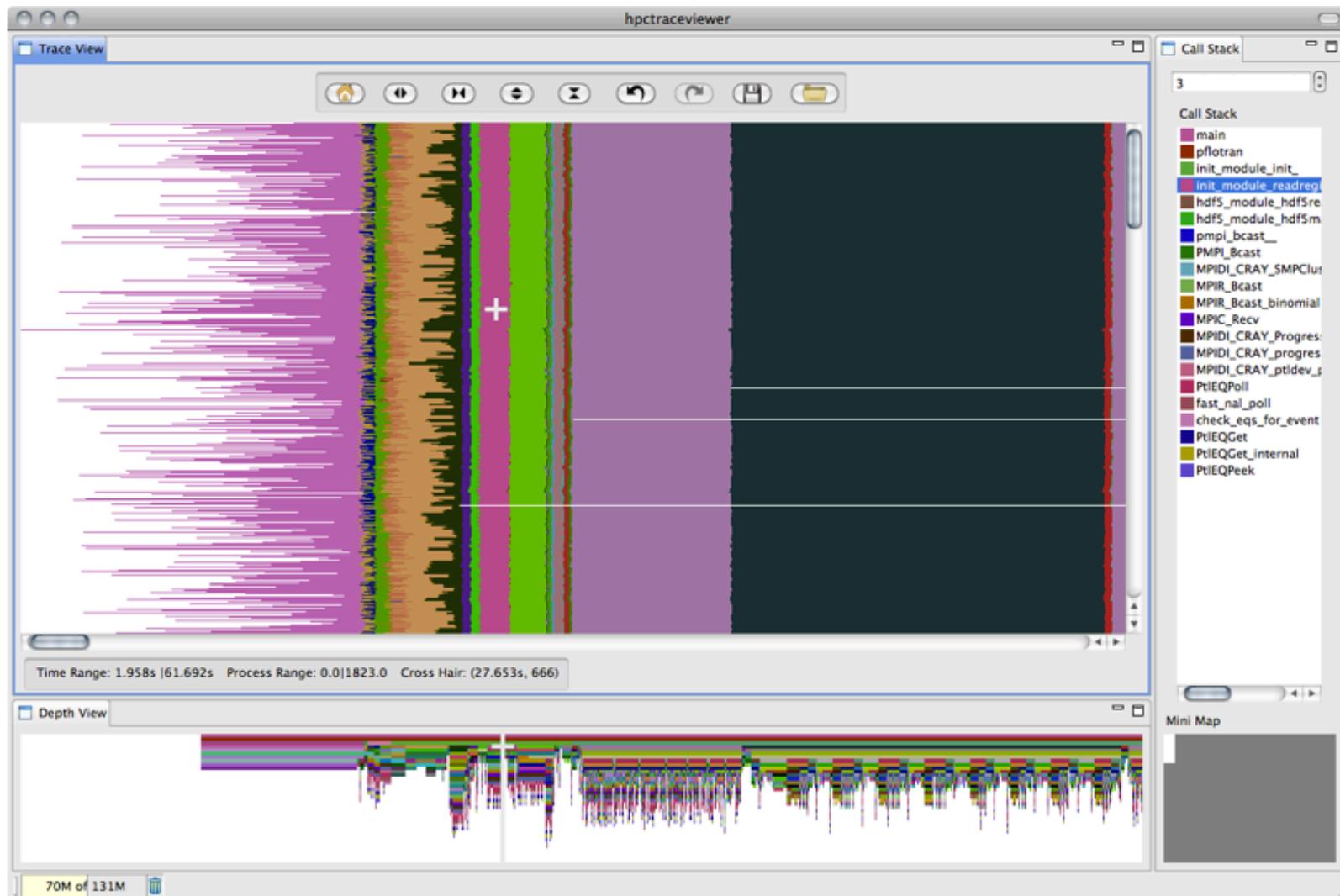
# Call Path Tracing for Parallel Programs

**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 file s3d\_f90.x. The top pane shows Fortran code with a loop at line 389 highlighted in blue. A blue arrow points from a text box to this line. The bottom pane shows a performance table with a red box around the entry for the loop at chemkin\_m.f90: 389, and a red arrow pointing from a text box to it.

```
379! there is an extra 1000 in the numerator for the molecular weight conversion
380
381 rateconv = l_ref * 1.0e6 / (rho_ref * a_ref)
382!-----
383! get reaction rate from getrates and convert units
384
385 do k = kz1,kzu
386 do j = jyl,jyu
387 do i = ixl,ixu
388
389 yspec(:) = yspecies(i, j, k, :)
390 call getrates(pressure(i,j,k)*pconv,temp(i,j,k)*tconv, &
391 yspec,ickwrk,rckwrk,rr_r1)
392
393 rr_r(i,j,k,:) = rr_r1(:) * rateconv * molwt(:)
394
395 enddo
396 enddo
397 enddo
398!-----
399 return
400 end subroutine reaction_rate_bounds
401!-----
```

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	41.2%	9.40e+03	18.7%
loop at solve_driver.f90: 137	5.66e+05	41.0%	9.32e+03	18.6%
integrate	5.66e+05	41.0%	9.32e+03	18.6%
integrate_erk_jstage_it	5.36e+05	38.9%	8.51e+03	17.0%
loop at integrate_erk_jstage_it_gen.f: 47	5.36e+05	38.9%	8.51e+03	17.0%
rhsf	5.17e+05	37.4%	7.99e+03	15.9%
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	2.57e+05	18.6%	1.24e+03	2.5%
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%

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

41.2% of memory hierarchy latency related to yspecies array

# 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 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

The screenshot shows the hpcviewer interface for the routine eos\_helm.F90. The code editor displays the subroutine definition, with user code highlighted in blue. A call stack table below shows the execution cost breakdown for various routines, with several entries highlighted in red. A text box on the right explains that costs are sorted by exclusive time and that only routines shown in blue are user code.

Costs sorted by exclusive time spent in individual routines  
Note: only the routines shown in blue are user code  
BG/P DCMF Communication Layer costs

Scope	8192/WALLCLOCK (us) (I)	8192/WALLCLOCK (us) (E)
Experiment Aggregate Metrics	6.71e+08 100 %	6.71e+08 100 %
DCMF::Protocol::MultiSend::TreeAllreduceShortRecvPostMessage::advance(unsigned int, DCMF::Queueing::	1.07e+08 16.0%	1.07e+08 16.0%
eos_helm	1.54e+08 22.9%	8.26e+07 12.3%
DCMF::BGPLockManager::globalBarrierQueryDone()	4.62e+07 6.9%	4.62e+07 6.9%
DMA_RecFifoSimplePollNormalFifoByld	3.22e+07 4.8%	3.08e+07 4.6%
_xlddpow	6.31e+07 9.4%	2.85e+07 4.2%
DCMF::Queueing::GI::giMessage::advance()	7.15e+07 10.6%	2.52e+07 3.8%
expinner2	1.94e+07 2.9%	1.94e+07 2.9%
rieman	2.12e+07 3.2%	1.82e+07 2.7%
loginner2	1.56e+07 2.3%	1.56e+07 2.3%
states	1.51e+07 2.2%	1.36e+07 2.0%
amr_perm_to_1blk	1.30e+07 1.9%	1.30e+07 1.9%
__xl_log	1.27e+07 1.9%	1.27e+07 1.9%
IPRA.\$_local_tree_module_NMOD_search_and_prune_local_tree	1.24e+07 1.8%	1.24e+07 1.8%
amr_restrict_unk_genorder	1.20e+07 1.8%	1.20e+07 1.8%
DCMF::DMA::Device::advance()	5.71e+07 8.5%	1.20e+07 1.8%
DCMF::Queueing::GI::Device::advance()	8.36e+07 12.4%	1.12e+07 1.7%

# Execution Cost Attribution (Callers View)

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

The screenshot shows the hpcviewer interface. The top pane displays the source code for `mpi_amr_comm_setup.F90`, with line 419 highlighted: `Call MPI_ALLREDUCE (itemp, max_blks_sent, 1, MPI_INTEGER, MPI_MAX, MPI_COMM_WORLD, ierror)`. The bottom pane shows the 'Callers View' call stack. A red box highlights the entry `DCMF::Protocol::MultiSend::TreeAllreduceShortRecvPostMessage::advance(unsigned int, DCMF::Queueing::Tree::TreeMsgContext)`. Red arrows point from this entry to `amr_guardcell` and `amr_flux_conserve_udt` in the caller list.

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.

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

The screenshot shows the hpcviewer interface for the file 'Hydro.F90'. The top pane displays source code lines 136-143, with line 137 highlighted: `call hy_ppm_sweep( myPE, numProcs, & blockCount, blockList, & timeEndAdv, dt, dtOld, & SWEEP_X )`. Below the code, the 'Calling Context View' is active, showing a tree of execution scopes. A table at the bottom provides performance metrics for each scope.

Scope	8192/WALLCLOCK (us) (I)	8192/WALLCLOCK (us) (E)
Experiment Aggregate Metrics	6.76e+08 100 %	6.76e+08 100 %
flash	6.76e+08 100 %	
driver_evolveflash	5.44e+08 80.5%	
loop at Driver_evolveFlash.F90: 92	5.44e+08 80.5%	
hydro	1.78e+08 26.4%	
hy_ppm_sweep	5.99e+07 8.9%	2.50e+04 0.0%
loop at hy_ppm_sweep.F90: 222	2.04e+07 3.0%	5.00e+03 0.0%
grid_fillguardcells	1.65e+07 2.4%	
loop at hy_ppm_sweep.F90: 520	1.36e+07 2.0%	
grid_conservefluxes	9.45e+06 1.4%	
hy_ppm_sweep	5.98e+07 8.8%	1.50e+04 0.0%
hy_ppm_sweep	5.86e+07 8.7%	1.50e+04 0.0%
hydro	1.76e+08 26.0%	
hy_ppm_sweep	6.07e+07 9.0%	2.00e+04 0.0%
hy_ppm_sweep	5.91e+07 8.7%	1.50e+04 0.0%
hy_ppm_sweep	5.58e+07 8.2%	1.50e+04 0.0%
driver_sourceterms	5.03e+07 7.4%	
grid_updaterefinement	4.92e+07 7.3%	

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

The screenshot shows the hpcviewer interface for the pflotran application. The top pane displays Fortran code from timestepper.F90, with lines 1227-1237 visible. A callout box highlights that 66.5% of cycles are spent in transport calculation and 30.8% in flow calculation. The bottom pane shows a 'Flat View' of the execution cost attribution table.

Scope	TOT_CYC:Sum (I)	FP_OPS:Sum (I)	imbalance:Sum (I)
main	1.96e+16 100 %	3.14e+15 100 %	6.29e+15 100 %
pflotran	1.96e+16 100 %	3.14e+15 100 %	6.29e+15 100 %
timestepper_module_stepperrun_	1.94e+16 98.8 %	3.14e+15 99.9 %	6.21e+15 98.6 %
loop at timestepper.F90: 384	1.94e+16 98.7 %	3.14e+15 99.9 %	6.21e+15 98.6 %
timestepper_module_steppersteptransportdt_	1.31e+16 66.5 %	2.94e+15 93.5 %	1.97e+15 31.4 %
loop at timestepper.F90: 1230	1.31e+16 66.5 %	2.94e+15 93.5 %	1.97e+15 31.4 %
discretization_module_discretizationlocaltolocal_	8.84e+11 0.0 %	1.47e+07 0.0 %	
discretization_module_discretizationlocaltolocal_	5.56e+11 0.0 %	4.90e+06 0.0 %	
timestepper_module_stepperstepflowdt_	6.05e+15 30.8 %	1.72e+14 5.5 %	9.87e+13 1.6 %
timestepper_module_stepperupdatesolution_	2.24e+14 1.1 %	2.95e+13 0.9 %	
output_module_output_	3.45e+13 0.2 %	1.47e+07 0.0 %	
timestepper_module_stepperupdatesolution_	4.59e+12 0.0 %	8.00e+11 0.0 %	
timestepper_module_stepperjumpstart_	2.97e+12 0.0 %	1.38e+11 0.0 %	
output_module_output_	1.19e+12 0.0 %		

# 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

Scope	TOT_CYC:Sum (l)	FP_OPS:Sum (l)	imbalance:Sum (l)	idle
pflotran	1.96e+16 100 %	3.14e+15 100 %	6.29e+15 100 %	6.2
timestepper_module_stepperrun_	1.94e+16 98.8%	3.14e+15 99.9%	6.21e+15 98.6%	6.2
loop at timestepper.F90: 384	1.94e+16 98.7%	3.14e+15 99.9%	6.21e+15 98.6%	6.2
timestepper_module_steppersteptransportdt_	1.31e+16 66.5%	2.94e+15 93.5%	1.97e+15 31.4%	1.9
loop at timestepper.F90: 1230	1.31e+16 66.5%	2.94e+15 93.5%	1.97e+15 31.4%	1.9
loop at timestepper.F90: 1254	1.29e+16 65.9%	2.92e+15 92.9%	1.97e+15 31.3%	1.9
snessolve_	1.28e+16 65.1%	2.89e+15 91.9%	1.97e+15 31.3%	1.9
SNESolve	1.28e+16 65.1%	2.89e+15 91.9%	1.97e+15 31.3%	1.9
SNESolve_LS	1.28e+16 65.1%	2.89e+15 91.9%	1.97e+15 31.3%	1.9
loop at ls.c: 181	1.25e+16 63.6%	2.84e+15 90.5%	1.91e+15 30.3%	1.9
SNES_KSPSolve	6.37e+15 32.4%	1.85e+15 58.9%	7.23e+14 11.5%	1.1
KSPSolve	6.37e+15 32.4%	1.85e+15 58.9%	7.23e+14 11.5%	1.1
KSPSolve_BCGS	4.62e+15 23.5%	5.70e+14 18.1%		
loop at bcgs.c: 69	3.90e+15 19.9%	5.21e+14 16.6%		
PCApplyBAorAB	1.58e+15 8.1%	2.54e+14 8.1%		
PCApplyBAorAB	1.58e+15 8.0%	2.55e+14 8.1%		
VecDotNorm2	3.01e+14 1.5%	2.33e+12 0.1%		

Overall: 1 FLOP every 7.4 cycles