Gaining Insight into Parallel Program Performance Using Sampling

John Mellor-Crummey, Laksono Adhianto,
Mike Fagan, Mark Krentel, Nathan Tallent

Department of Computer Science
Rice University

Performance Analysis Goals

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

Outline

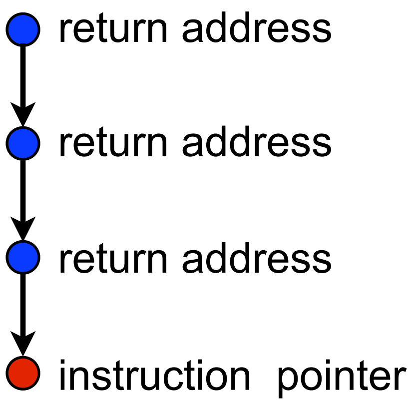
- Evaluating context-sensitive behavior
- Pinpointing and quantifying scalability bottlenecks
- Analyzing multithreaded computations with work stealing
- Quantifying the impact of lock contention on threaded code
- Understanding how computations evolve
- Work in progress

State of the Art: Call Path Profiling

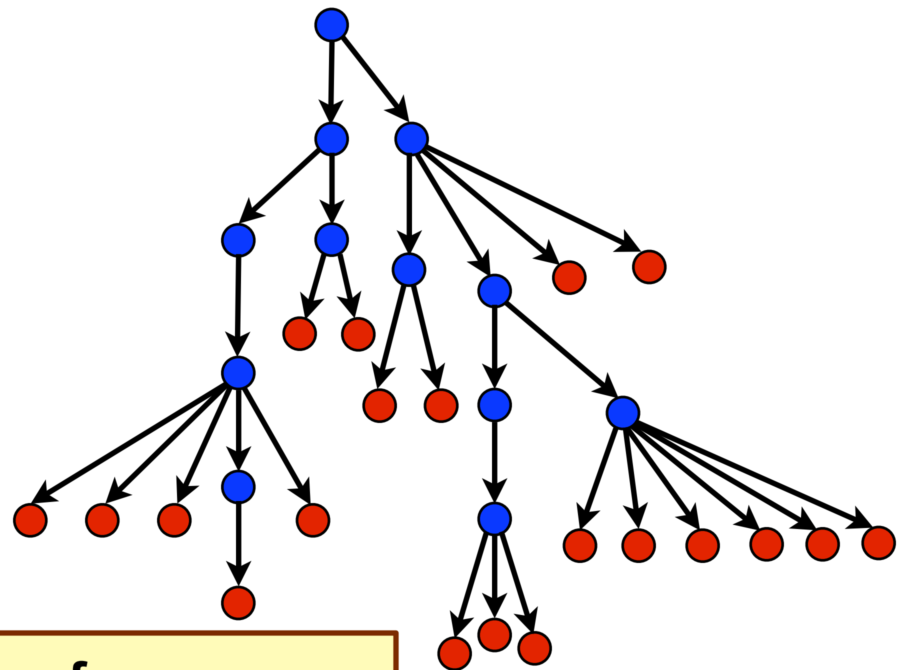
Measure and attribute costs in their *calling* context

- **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

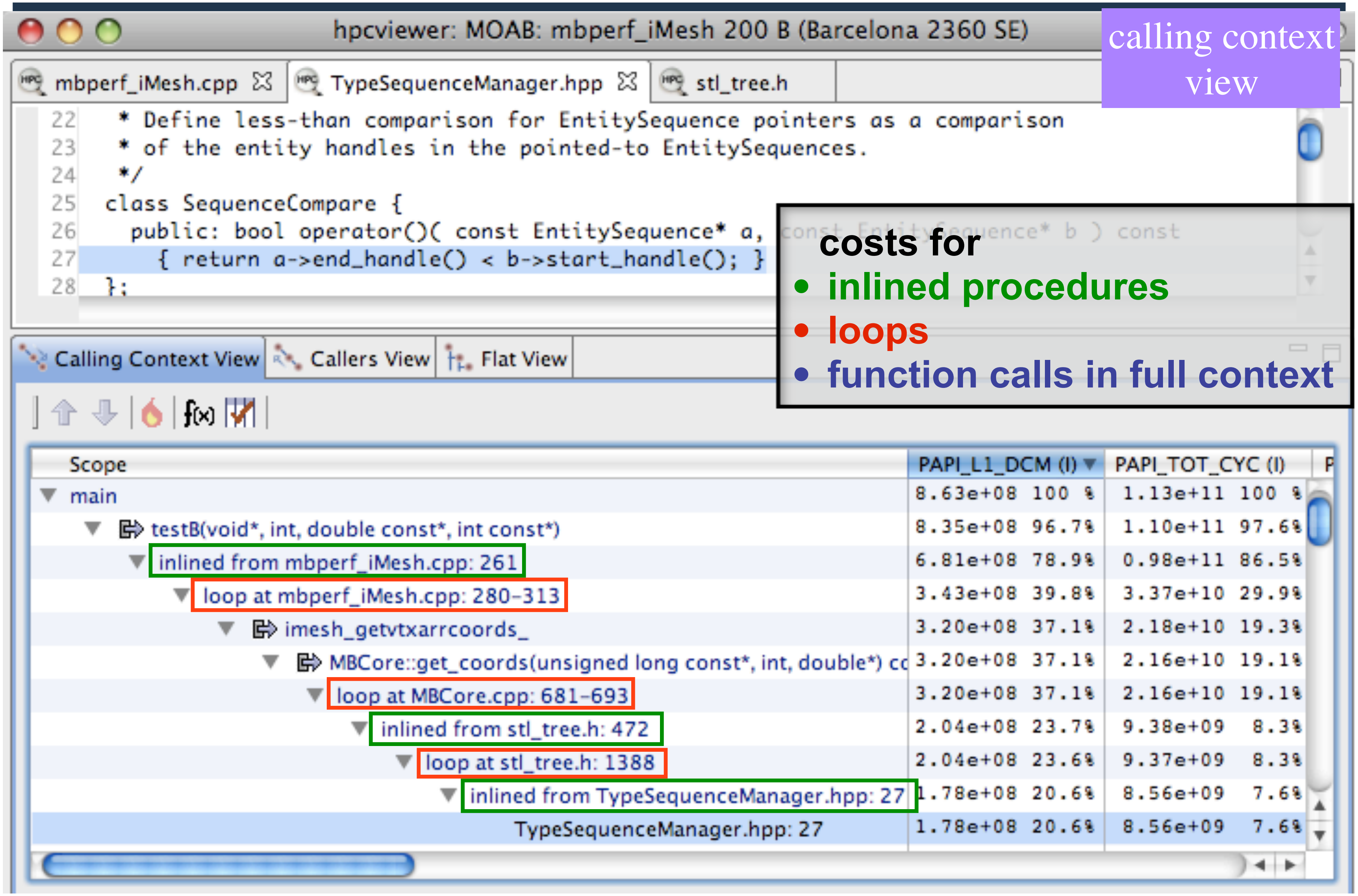
Unwinding Fully-optimized Parallel Code

Unwinding based on 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**
  - — 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**
  - — almost flawless unwinding
  - — 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.**

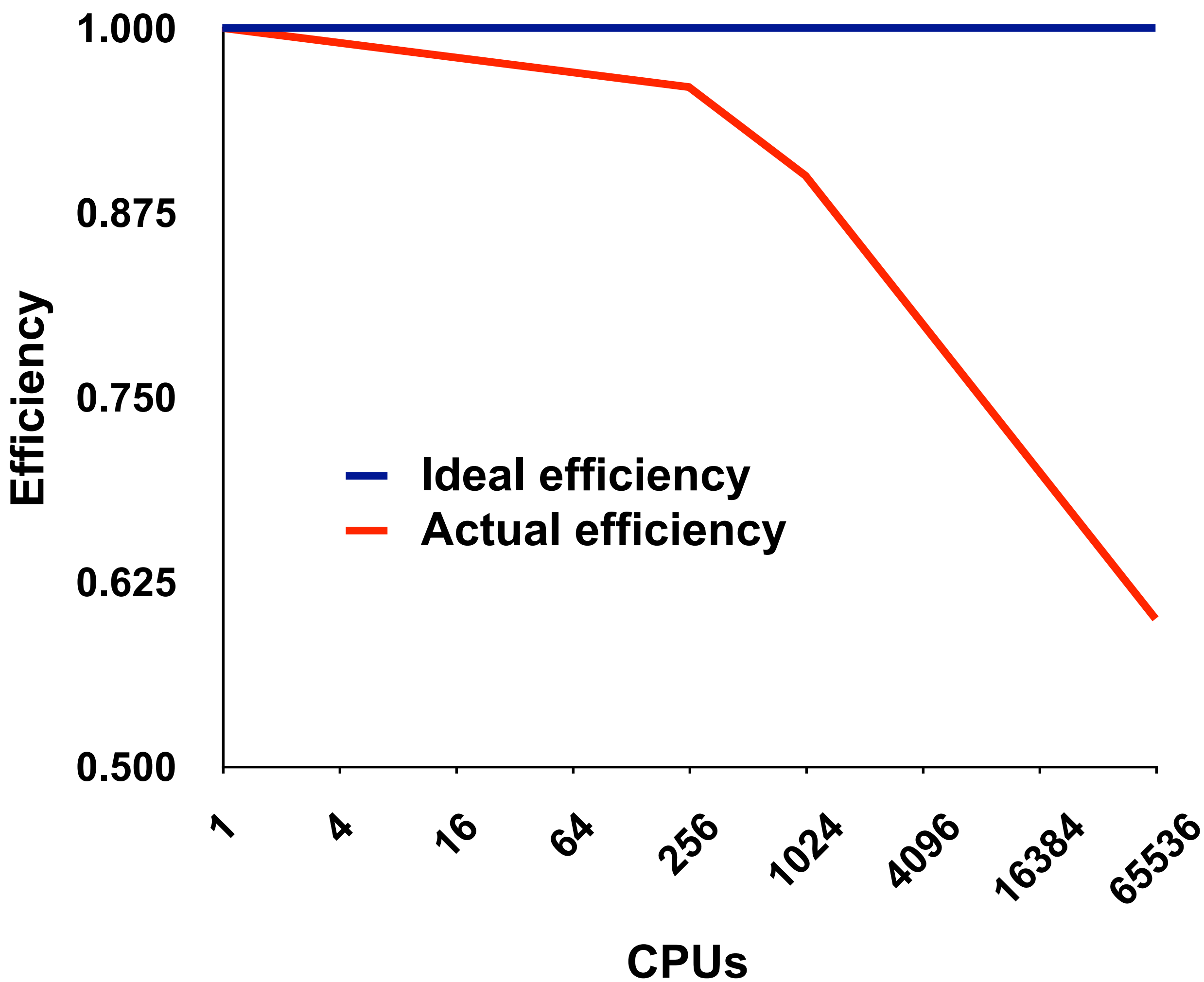
Detailed Attribution: MOAB Mesh Benchmark



Outline

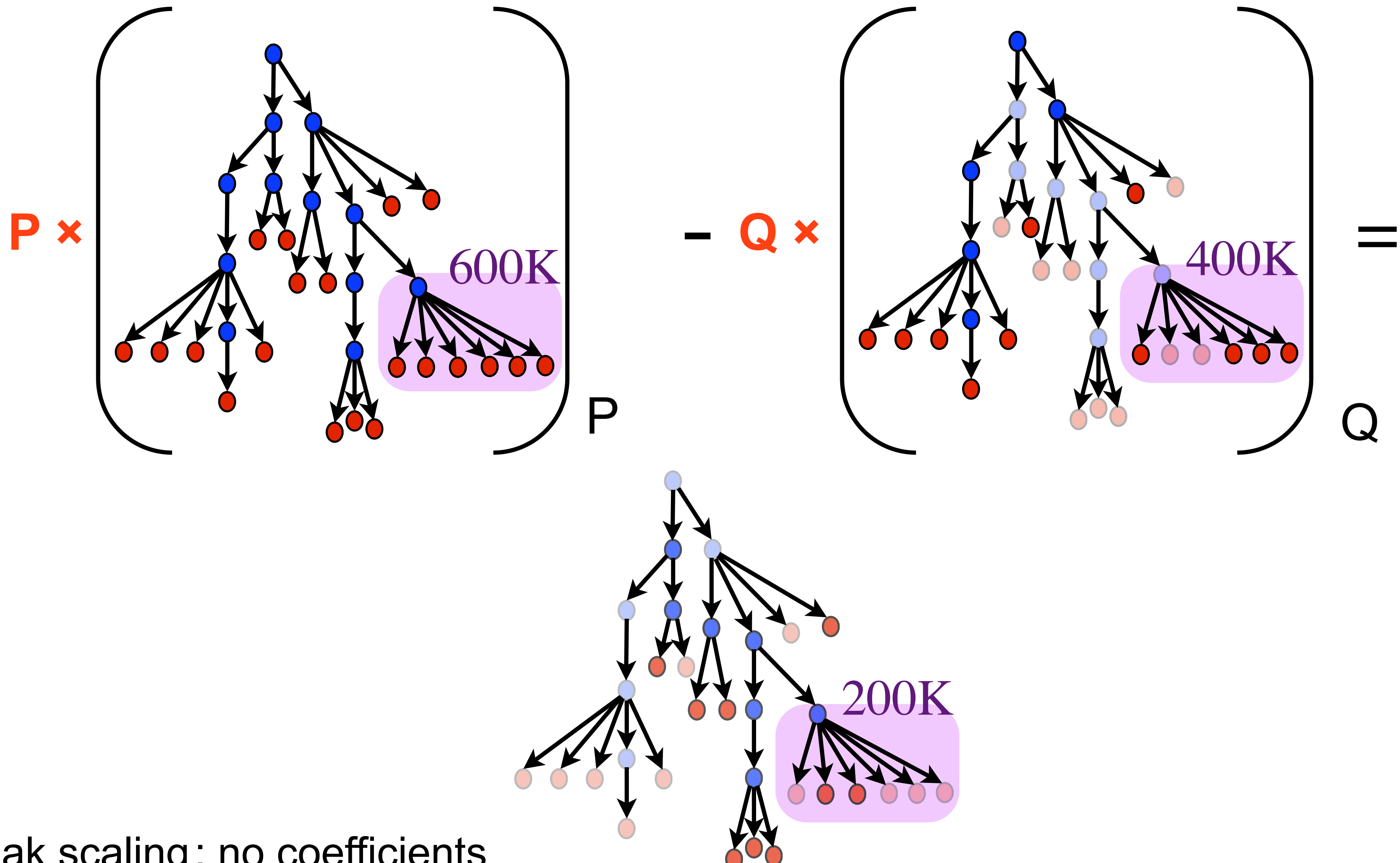
- Evaluating context-sensitive behavior
- Pinpointing and quantifying scalability bottlenecks
- Analyzing multithreaded computations with work stealing
- Quantifying the impact of lock contention on threaded code
- Understanding how computations evolve
- Work in progress

The Problem of Scaling



Note: higher is better

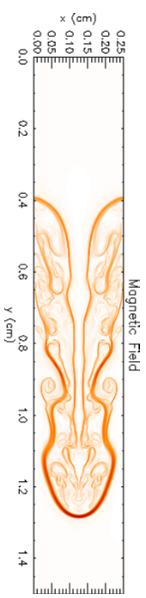
Pinpointing and Quantifying Scalability Bottlenecks



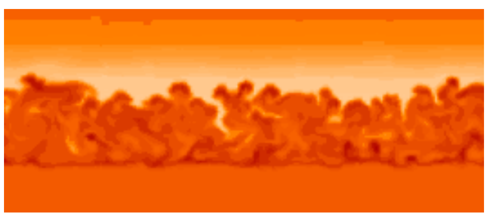
Weak scaling: no coefficients
Strong scaling: needs **red** coefficients

Scalability Analysis Demo

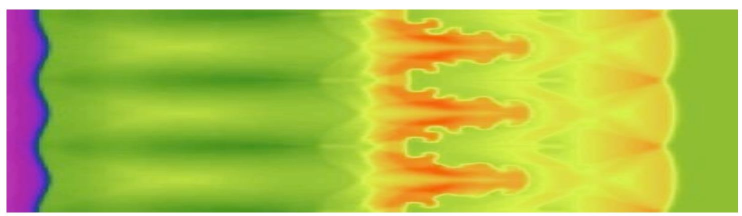
Code:	**University of Chicago FLASH**
Simulation:	**white dwarf collapse**
Platform:	**Blue Gene/P**
Experiment:	**8192 vs. 256 processors**
Scaling type:	**weak**



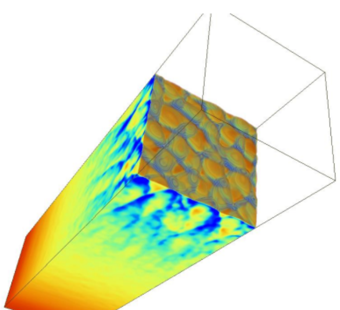
Magnetic Rayleigh-Taylor



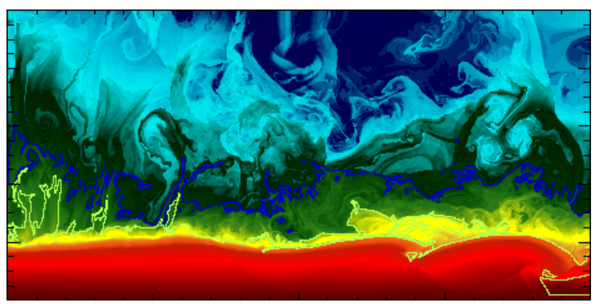
Nova outbursts on white dwarfs



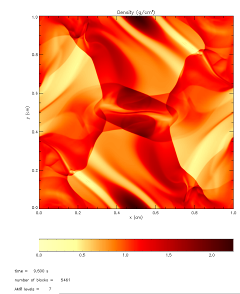
Laser-driven shock instabilities



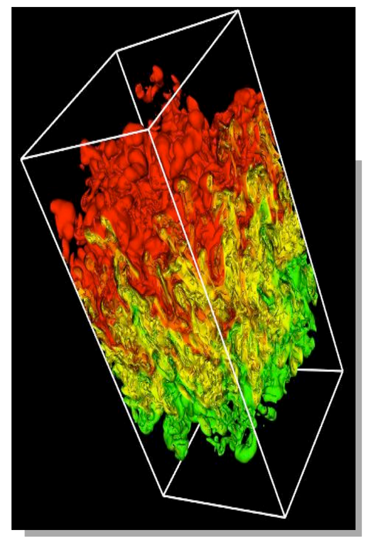
Cellular detonation



Helium burning on neutron stars



Orzag/Tang MHD vortex



Rayleigh-Taylor instability

Figures courtesy of FLASH Team, University of Chicago

S3D:Multicore Losses at the Procedure Level

hpcviewer: [Profile Name]

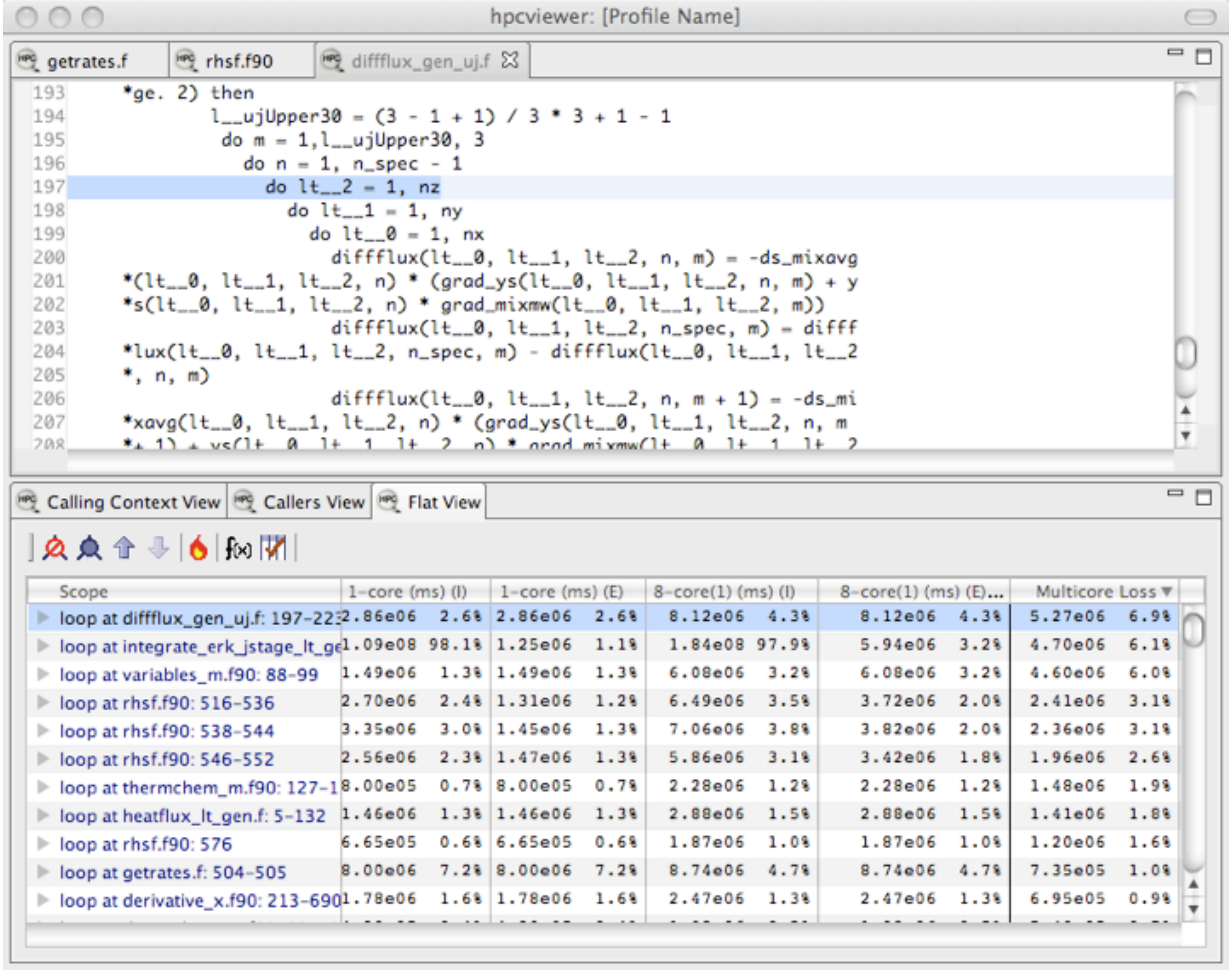
getrates.f | rhsf.f90 | diffflux_gen_uj.f

```
subroutine rhsf( q, rhs )
!-----------------------------------------------------------------------
! Changes
! Ramanan Sankaran - 01/04/05
! 1. Diffusive fluxes are computed without having to convert units.
! Ignore older comments about conversion to CGS units.
! This saves a lot of flops.
! 2. Mixavg and Lewis transport modules have been made interchangeable
! by adding dummy arguments in both.
  !-----------------------------------------------------------------------
  !                      Author:  James Sutherland
  !                      Date:    April, 2002
  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  ! This routine calculates the time rate of change for the
  ! momentum, continuity, energy, and species equations.
  !
```

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 ▼
Experiment Aggregate Metrics	1.11e08 100 %	1.11e08 100 %	1.88e08 100 %	1.88e08 100 %	7.64e07 100 %
rhsf	1.07e08 96.5%	6.60e06 5.9%	1.77e08 94.1%	1.65e07 8.8%	9.92e06 13.0%
diffflux_proc_looptool	2.86e06 2.6%	2.86e06 2.6%	8.12e06 4.3%	8.12e06 4.3%	5.27e06 6.9%
integrate_erk_jstage_lt	1.09e08 98.1%	1.25e06 1.1%	1.84e08 97.9%	5.94e06 3.2%	4.70e06 6.1%
GET_MASS_FRAC.in.VARIABLES_M	1.49e06 1.3%	1.49e06 1.3%	6.08e06 3.2%	6.08e06 3.2%	4.59e06 6.0%
ratx	1.01e07 9.1%	1.00e07 9.0%	4.41e07 23.5%	1.40e07 7.4%	3.95e06 5.2%
qssa	3.52e06 3.2%	3.52e06 3.2%	5.71e06 3.0%	5.71e06 3.0%	2.18e06 2.9%
ratt	3.26e07 29.2%	1.48e07 13.3%	4.38e07 23.3%	1.66e07 8.8%	1.76e06 2.3%
CALC_INV_AVG_MOL_WT.in.THER	9.70e05 0.9%	9.70e05 0.9%	2.68e06 1.4%	2.68e06 1.4%	1.70e06 2.2%
computeheatflux_looptool	1.46e06 1.3%	1.46e06 1.3%	2.88e06 1.5%	2.88e06 1.5%	1.41e06 1.8%
rdwdot	3.09e06 2.8%	3.09e06 2.8%	4.33e06 2.3%	4.33e06 2.3%	1.24e06 1.6%

S3D: Multicore Losses at the Loop Level




```
     *ge. 2) then
             l__ujUpper30 = (3 - 1 + 1) / 3 * 3 + 1 - 1
              do m = 1,l__ujUpper30, 3
                do n = 1, n_spec - 1
                  do lt__2 = 1, nz
                    do lt__1 = 1, ny
                      do lt__0 = 1, nx
                        diffflux(lt__0, lt__1, lt__2, n, m) = -ds_mixavg
     *(lt__0, lt__1, lt__2, n) * (grad_ys(lt__0, lt__1, lt__2, n, m) + y
     *s(lt__0, lt__1, lt__2, n) * grad_mixmw(lt__0, lt__1, lt__2, m))
                        diffflux(lt__0, lt__1, lt__2, n_spec, m) = difff
     *lux(lt__0, lt__1, lt__2, n_spec, m) - diffflux(lt__0, lt__1, lt__2
     *, n, m)
                        diffflux(lt__0, lt__1, lt__2, n, m + 1) = -ds_mi
     *xavg(lt__0, lt__1, lt__2, n) * (grad_ys(lt__0, lt__1, lt__2, n, m
```

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

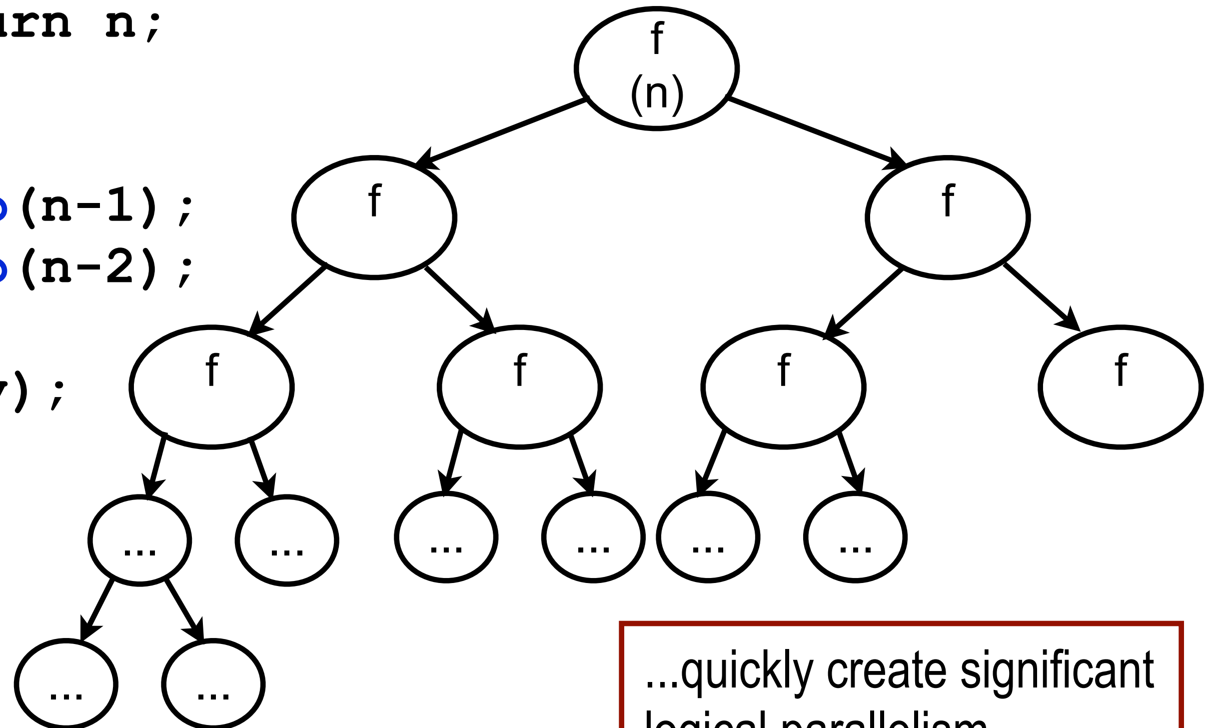
Outline

- Evaluating context-sensitive behavior
- Pinpointing and quantifying scalability bottlenecks
- Analyzing multithreaded computations with work stealing
- Quantifying the impact of lock contention on threaded code
- Understanding how computations evolve
- Work in progress

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**...



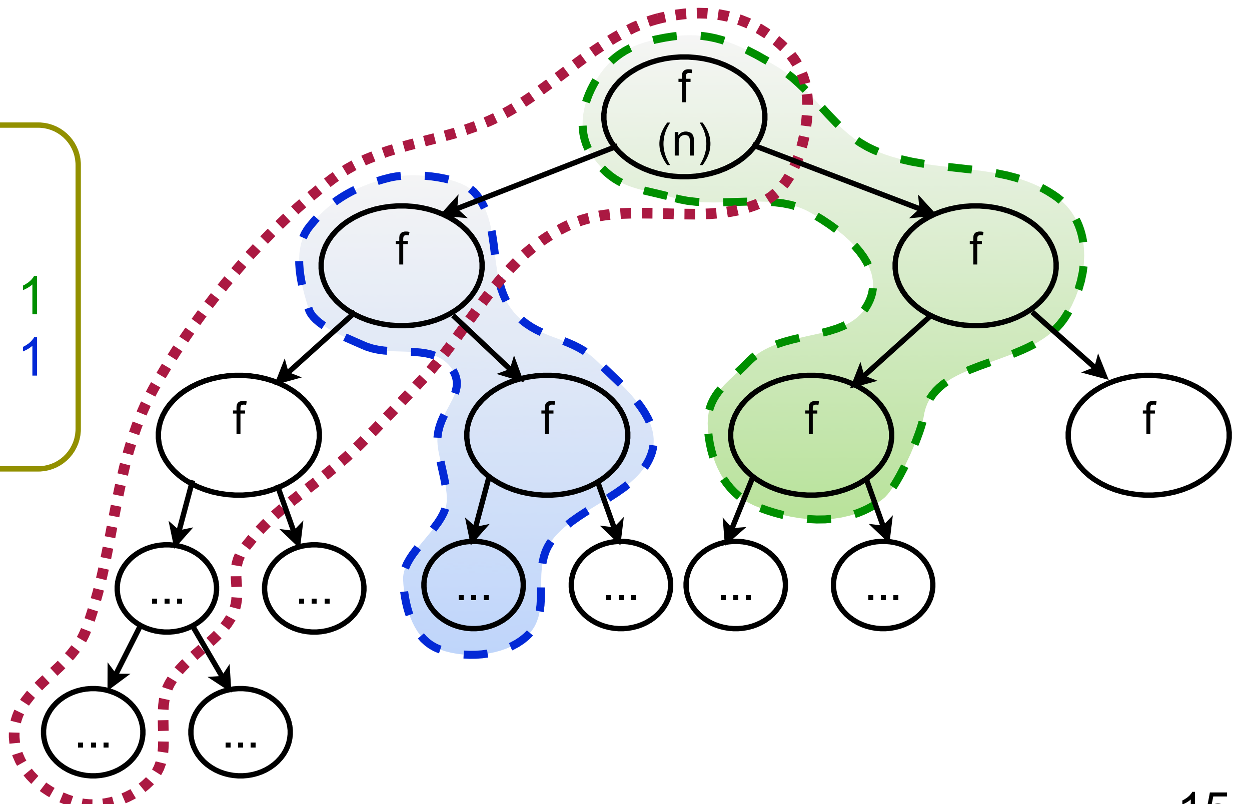
...quickly create significant logical parallelism.

Cilk Program Execution using Work Stealing

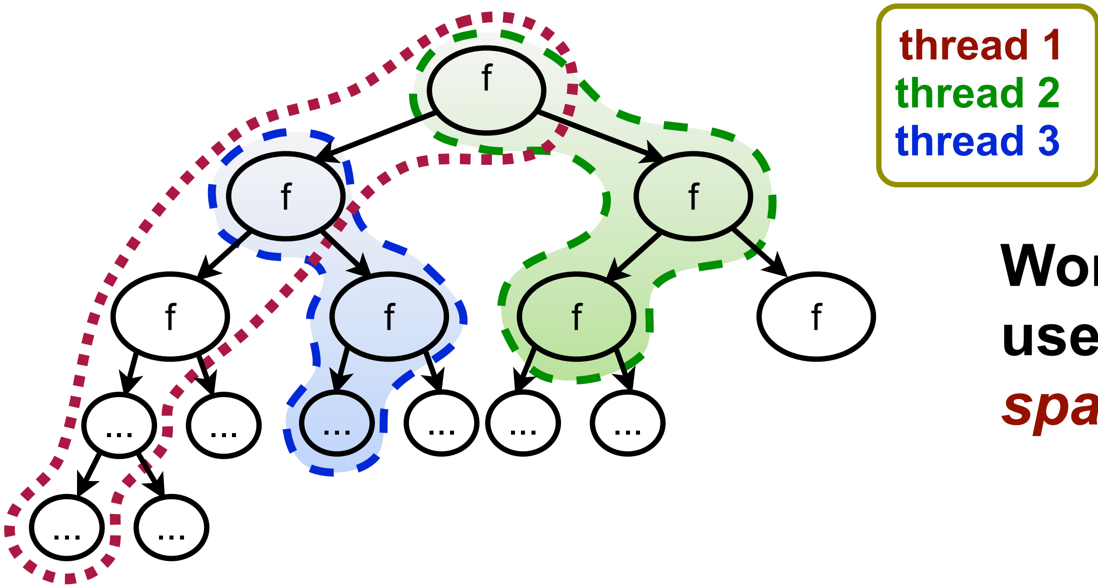
- **Challenge: Mapping logical tasks to compute cores**
- **Cilk approach:**
  - — **lazy thread creation plus work-stealing scheduler**
    - **`spawn`: a potentially parallel task is available**
    - **an idle thread steals tasks from a random working thread**

Possible Execution:

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

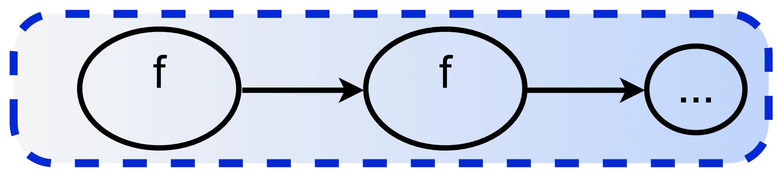


Wanted: Call Path Profiles of Cilk

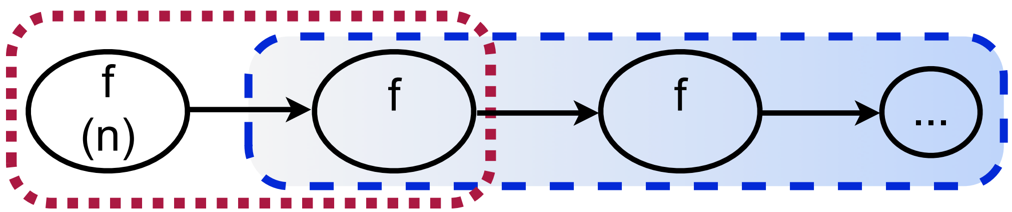


Work stealing ***separates*** **user-level calling contexts in** ***space and time***

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



  - **logical call path:**



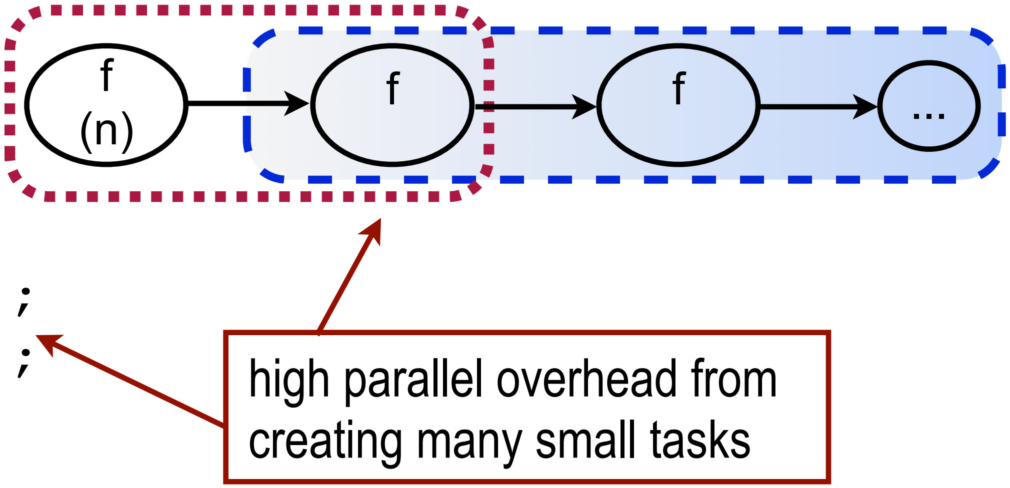
Logical call path profiling: Recover *full* relationship between *physical* and *user-level* execution

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: $\mathcal{W} \mathrel{+}= 1$ $\quad \sum \mathcal{W} = 5$

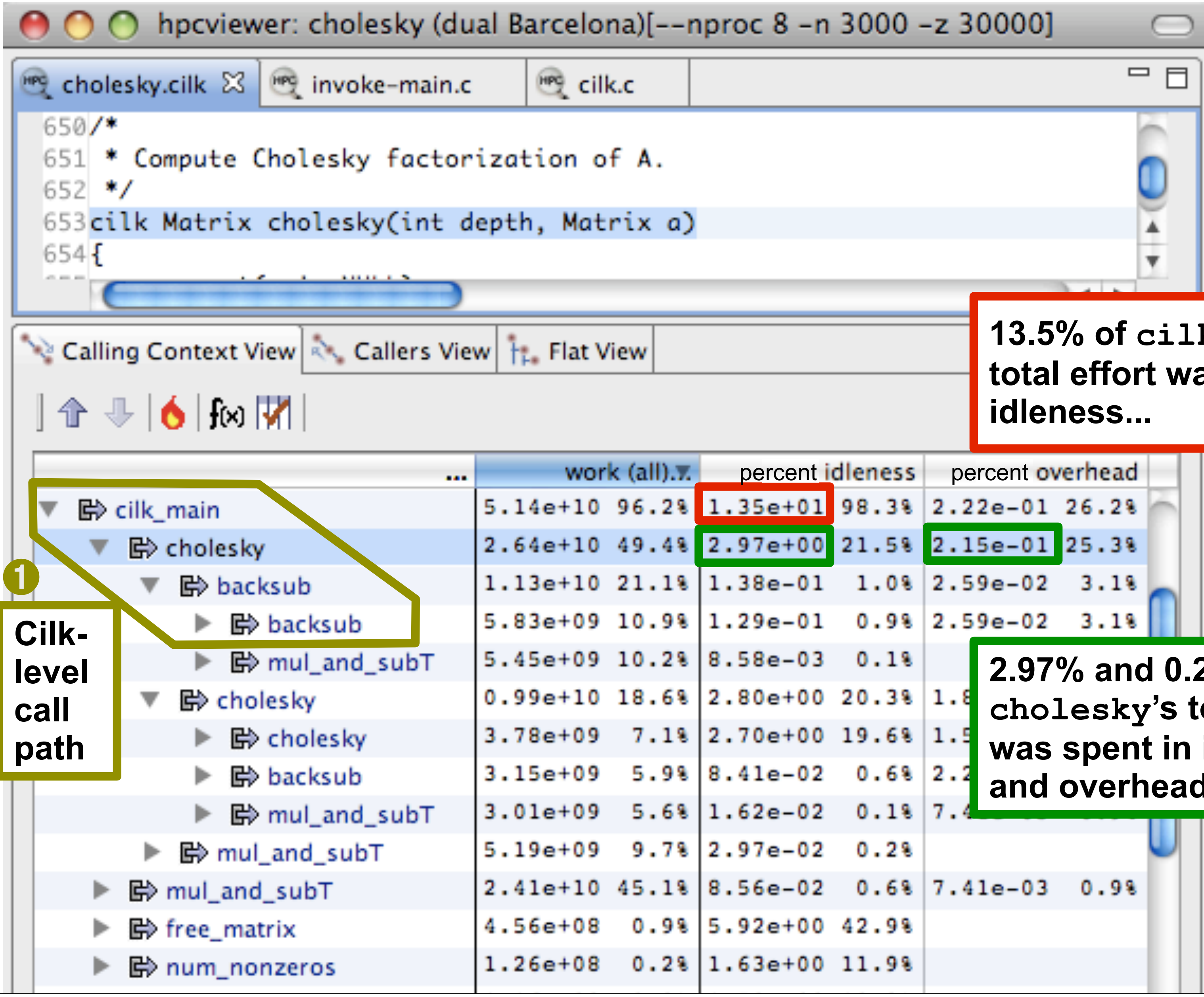
$\mathcal{I} \mathrel{+}= 3/5$ $\quad \sum \mathcal{I} = 3$

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

Parallel Overhead

- **Parallel overhead:**
  - — when a thread works on something other than user code
    - (we classify delays -- e.g., wait time -- 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'



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

❶ **Cilk-level call path**

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

- Evaluating context-sensitive behavior
- Pinpointing and quantifying scalability bottlenecks
- Analyzing multithreaded computations with work stealing
- Quantifying the impact of lock contention on threaded code
- Understanding how computations evolve
- Work in progress

Understanding Lock Contention

- **Lock contention => idleness:**
  - **explicitly threaded programs (Pthreads, etc)**
  - **implicitly threaded programs (critical sections in OpenMP, Cilk...)**
- **Use "blame-shifting:" 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& 
            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.

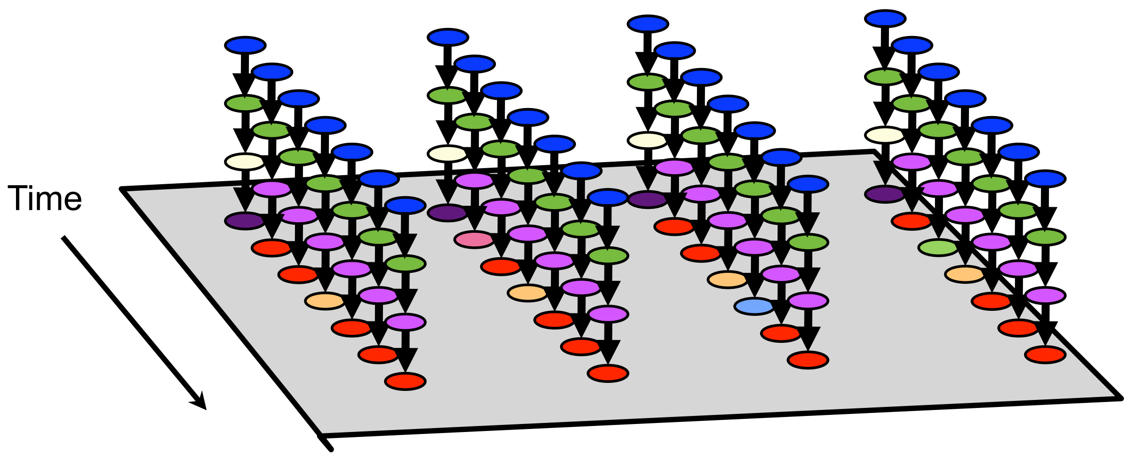
Outline

- Evaluating context-sensitive behavior
- Pinpointing and quantifying scalability bottlenecks
- Analyzing multithreaded computations with work stealing
- Quantifying the impact of lock contention on threaded code
- Understanding how computations evolve
- Work in progress

Understanding Temporal Behavior

- **Profiling compresses out the temporal dimension**
  - **—that's why serialization is invisible in profiles**
- **What can we do? Trace call path samples**
  - **—sketch:**
    - **N times per second, take a call path sample of each thread**
    - **organize the samples for each thread along a time line**
    - **view how the execution evolves left to right**
    - **what do we view?**

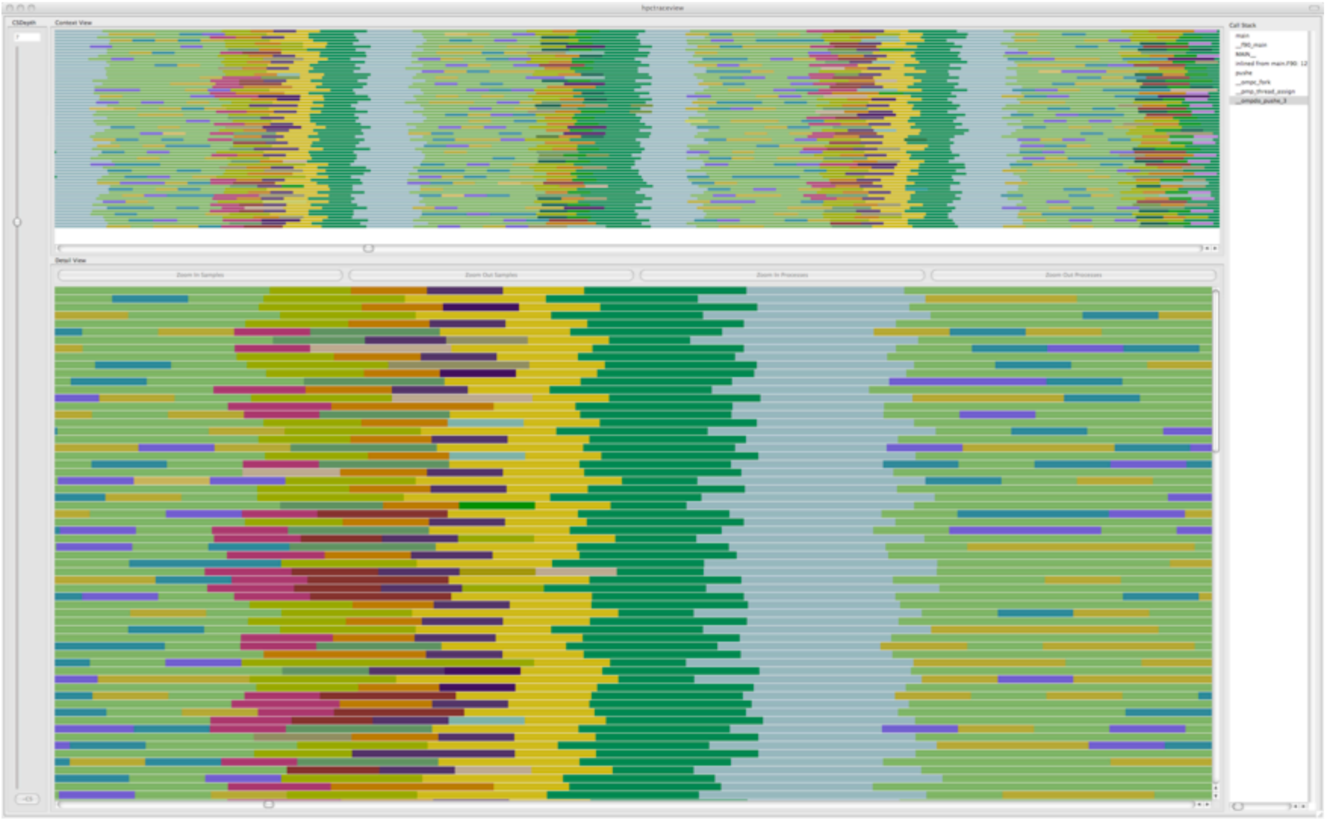
      assign each procedure a color; view execution with a depth slice



Call Path Sample Trace for GTC

Gyrokinetic Toroidal Code (GTC)

- **32 process MPI program**
- **Each process has a pair of threads managed with OpenMP**



L. Adhianto et al. *HPCToolkit: Tools for performance analysis of optimized parallel programs*, Concurrency and Computation: Practice and Experience. To appear.

Outline

- Evaluating context-sensitive behavior
- Pinpointing and quantifying scalability bottlenecks
- Analyzing multithreaded computations with work stealing
- Quantifying the impact of lock contention on threaded code
- Understanding how computations evolve
- Work in progress

Work in Progress

- **Analyze call path profiles for 100K+ cores in parallel**
  - — **aggregate profile CCTs for different cores to get union CCT**
  - — **compute summary statistics (e.g. min, mean, max, std. deviation)**
  - — **hypothesis: we can apply our top-down methodology for analyzing CCTs to assess profile differences**
    - **pinpoint and quantify profile differences at a high level**
    - **drill down using differential analysis of sample profiles**
- **Develop GUI support for sorting and histogramming profile values to cope with data from thousands of cores**
- **Using hardware monitoring capabilities to gain insight into data access patterns**
  - — **identify potential for improving locality and data reuse**
- **Visualize sampled traces for thousands of cores**