Effective Performance Measurement and Analysis of Multithreaded Applications

Nathan Tallent
John Mellor-Crummey

Rice University

CSCaDS

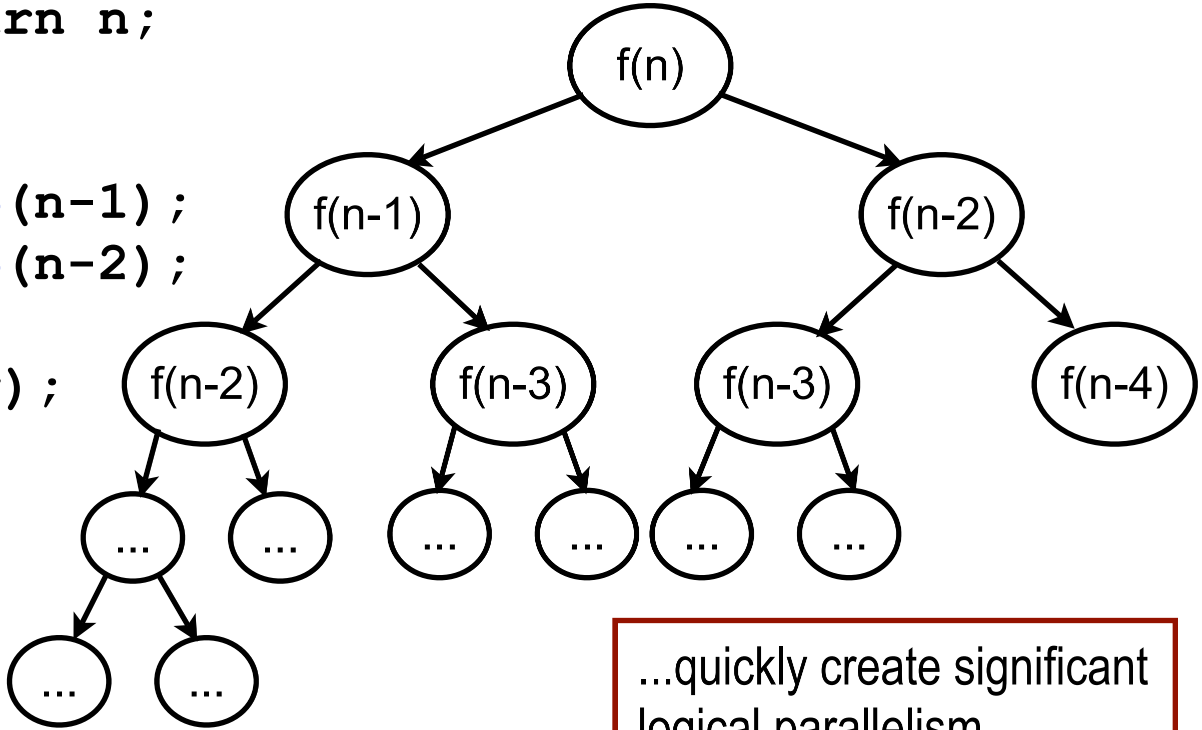
Wanted: Multicore Programming Models

- Simple
  - — well-defined semantics
    - e.g., language may guarantee races never occur
  - — Pthreads is analogous to assembly language
- Expressive
  - — task and data parallelism
  - — nested and irregular parallelism
- High performance
  - — dynamic work balancing
- Future: Transparent scaling to increasing core counts
  - — performance ≈ scaling (weak or strong)

Cilk is an early exemplar.
(TBB, X10/Habanero, MS Concurrency Runtime)

Cilk In a Nutshell

```
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**...

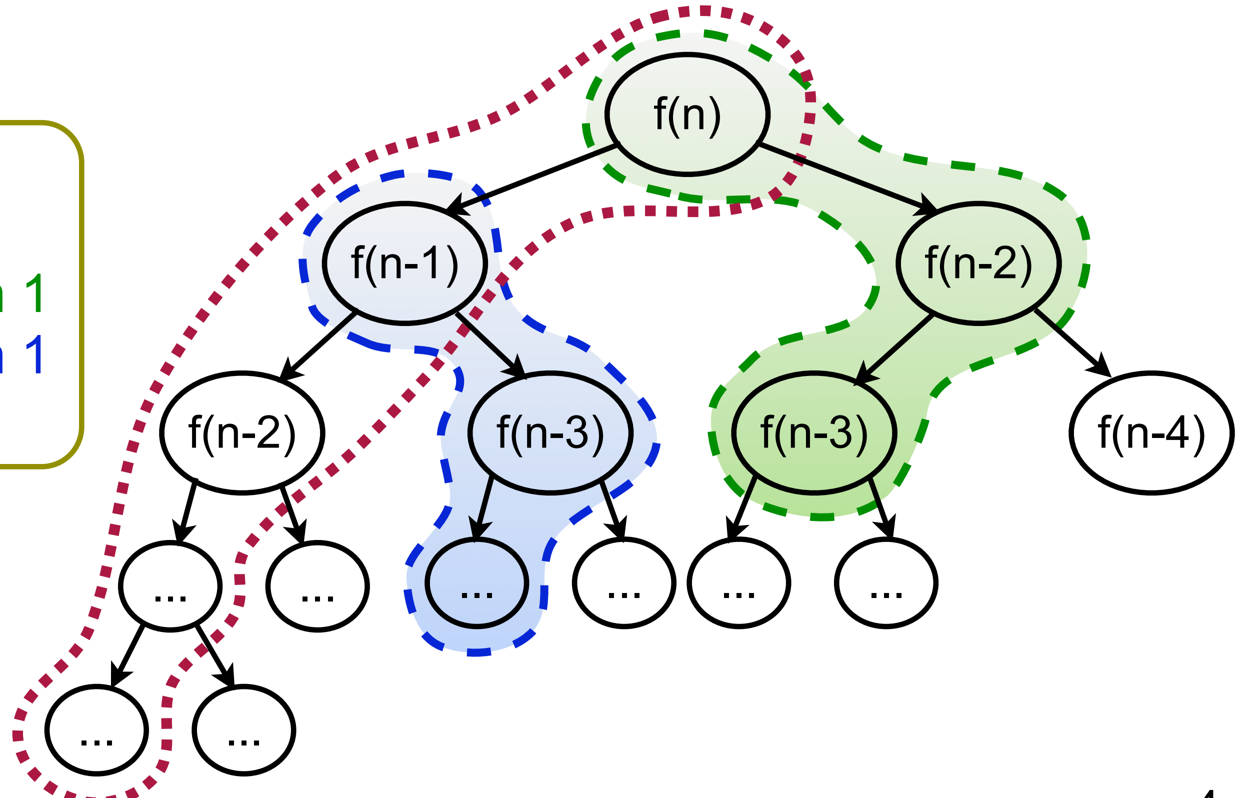
...quickly create significant logical parallelism.

Cilk Program Execution

- **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...

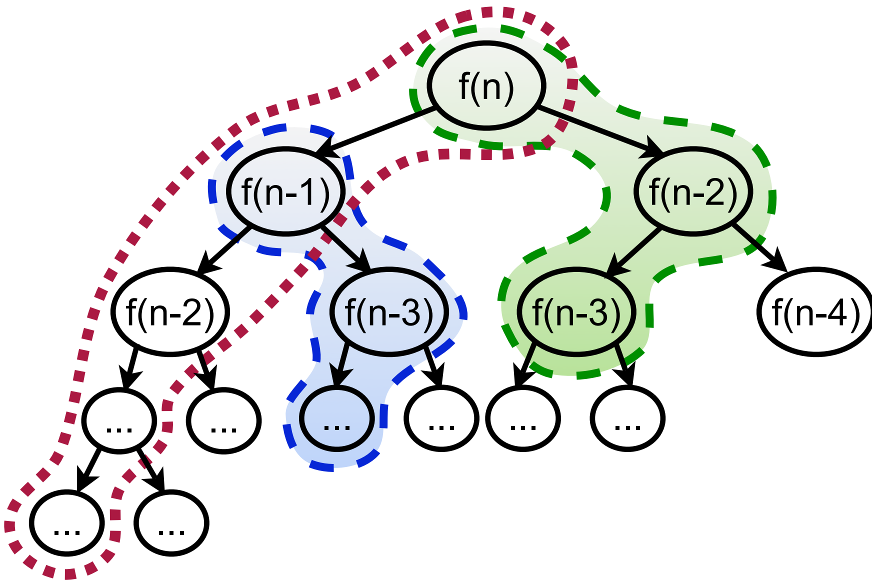


What If My Cilk Program Is Slow?

- **Cilk's metrics**
  - — measure of average parallelism for program + input
    - parallelism = work / critical path
    - lower bound on execution time (infinite number of cores)
- **Strengths**
  - — abstract measure of performance (machine independent)
  - — predictive insight for larger core counts
- **Weaknesses**
  - — not ***actionable***
    - if there is a bottleneck, where is it in my source code?
  - — abstract
    - hides important architectural details: e.g., memory effects
  - — computed via instrumentation
    - overhead perturbs application, affects compiler optimizations

Wanted: performance tools for threaded, parallel codes

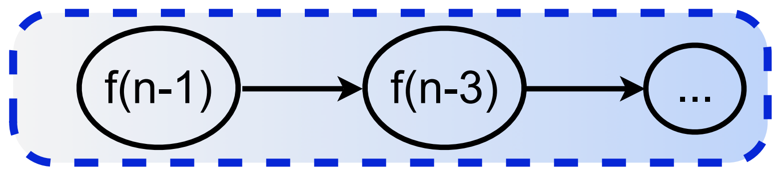
Wanted: Call Path Profiles of Cilk



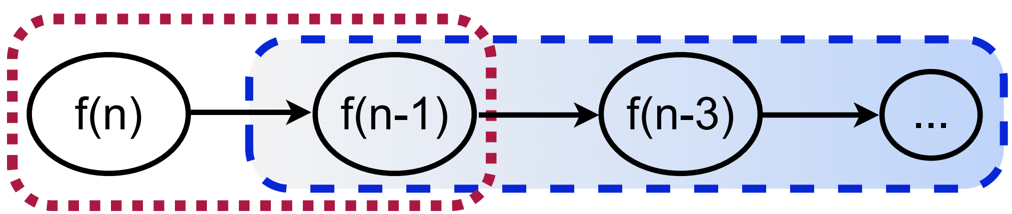
thread 1
thread 2
thread 3

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

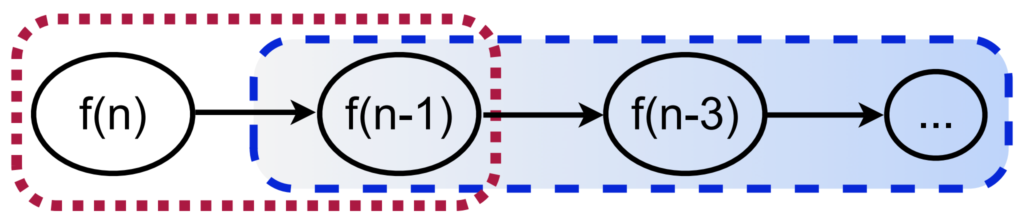
Performance Analysis of Work Stealing

Three Complementary Techniques:

- **Quantify *parallel idleness* (insufficient parallelism)**
- **Quantify *parallel overhead***
- **Recover *logical calling contexts* in presence of work-stealing**

- **Attribute *idleness* and *overhead* to *logical contexts***
  — **Pinpoint idleness and overhead to user-level code**

```
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

Outline

- **Motivation**
  - — multi-core: explicit shared memory parallelism
  - — languages: sophisticated, dynamically managed parallelism
- **Pinpointing and quantifying parallel bottlenecks**
  - — insufficient parallelism
  - — parallelization overhead
- **Logical call path profiling**
- **Conclusions**

Parallel Idleness

- **Parallel idleness:**
  - when a thread (core) is idle or blocked
- **Pinpoint idleness with call path profiling**
  - use statistical sampling
    - low, controllable overhead
    - on a sample, each thread receives an async signal
  - but...
    - idleness is manifested as samples within scheduler
    - blames the victim, not the perpetrator
    - not <u>actionable</u>!

Measuring Parallel Idleness

- **Metrics: Effort = “work” + “idleness”**
  - **— associate metrics with user-level calling contexts**
  - **— insight: attribute idleness to its cause: context of *working* thread**
- **Work stealing-scheduler: one thread per core (n cores)**
  - **— maintain $n_{\bar{w}}$ and n_w** (working/non-working threads)
    - **slight modifications to work-stealing run time**
      - **– maintain node-wide counter for n_w**
      - **– atomically decrement (incr.) when thread enters (exits) scheduler**
    - **when a sample event interrupts a working thread**
      - – $n_{\bar{w}} = n - n_w$
      - **– apportion idleness to it: $n_{\bar{w}} / n_w$**
- **Example: Dual quad-cores; on a sample, 5 are working:**

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

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

Summary

- **Idleness metric:**
  - — **identifies the cause of idleness: code with insufficient parallelism**
- **Measurement approach:**
  - — **requires only lightweight scheduler support**
  - — **negligible measurement overhead w/ sampling**

Outline

- Motivation
  - — multi-core: explicit shared memory parallelism
  - — languages: sophisticated, dynamically managed parallelism
- Pinpointing and quantifying parallel bottlenecks
  - — insufficient parallelism
  - — parallelization overhead
- Logical call path profiling
- Conclusions

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
  - possible approaches:
    - instrumentation
      - must support instruction level granularity
      - not practical
    - sampling?
      - not clear how to distinguish overhead from work

Pinpointing Overhead In Parallel Languages

- **Conceptual model:**
  - — before: total effort = work + idleness
  - — refine: work = useful-work + overhead
- **Approach:**
  - — insight: compiler tags instructions contributing to overhead
    - compiler knows which instructions are for overhead
    - permits *full* and *aggressive* optimization
  - — call path profiling...
    - attributes samples to instructions in context
  - — post-mortem analysis...
    - partitions samples into useful-work and overhead

Pinpointing Overhead for Cilk

- **Benefits:**
  - — requires only lightweight compiler support
    - (similar to support for debugging)
  - — permits a hierarchy of overhead categories
    - cf. cycle accounting
  - — can even be implemented as a preprocessor
  - — compatible with fully optimized code
  - — no measurement overhead

Using Parallel Idleness & Overhead

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

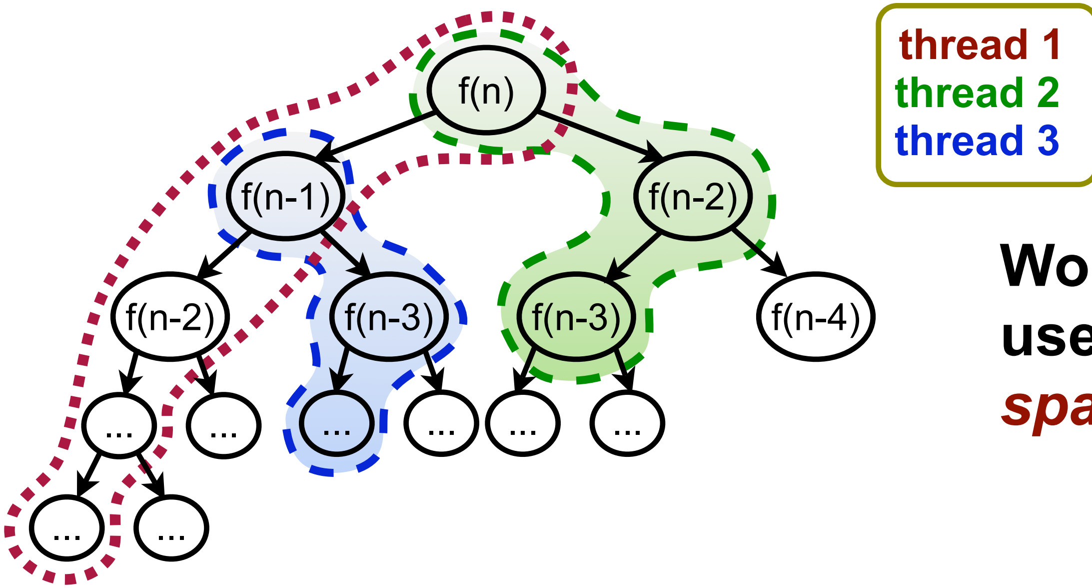
idleness	overhead	interpretation
low	low	effectively parallel
low	high	coarsen concurrency granularity
high	low	refine concurrency granularity
high	high	switch parallelization strategies

- **Normalize w.r.t. total effort to create**
  - **percent idleness or percent overhead**
- **Applicable to many programming models**
  - **Pthreads, OpenMP, Cilk, Intel TBB, etc.**

Outline

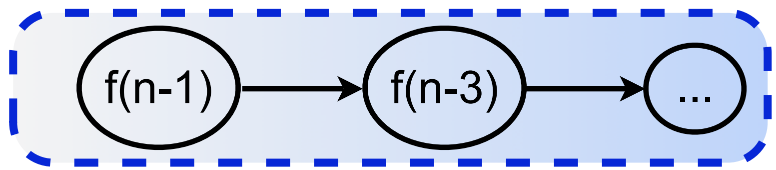
- **Motivation**
  - — multi-core: explicit shared memory parallelism
  - — languages: sophisticated, dynamically managed parallelism
- **Pinpointing and quantifying parallel bottlenecks**
  - — insufficient parallelism
  - — parallelization overhead
- **Logical call path profiling**
- **Conclusions**

Recall: Call Path Profiling...

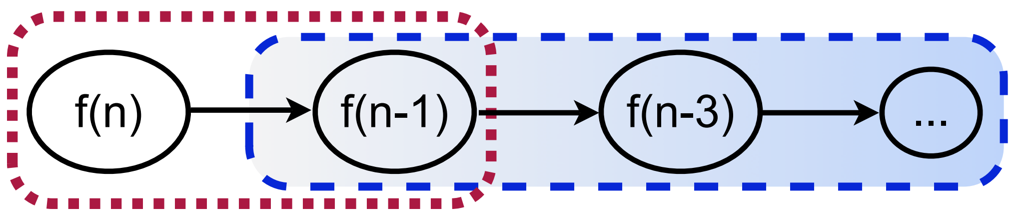


Work stealing *separates* user-level calling contexts in *space and time*

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



  - **logical call path:**

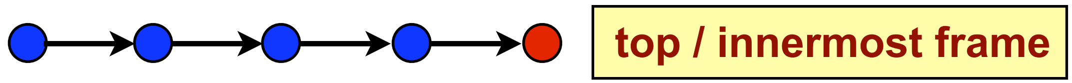


Logical call path profiling: Recover *full* relationship between *physical* and *user-level* execution

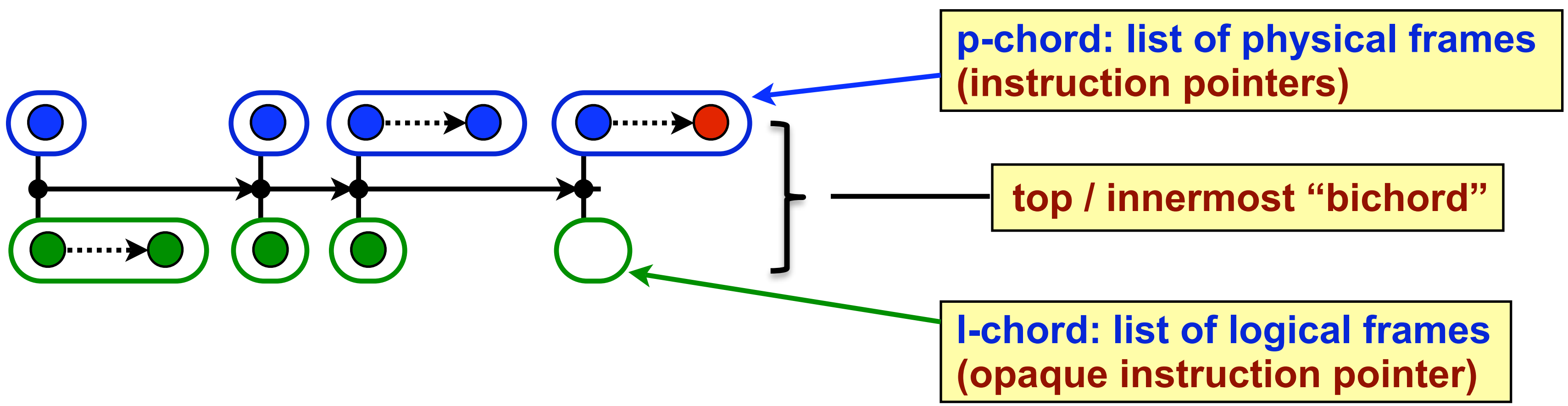
Logical Call Paths

Recover relationship between physical and user-level execution

- **Physical call path:**
  - a list of instruction pointers for active procedure frames

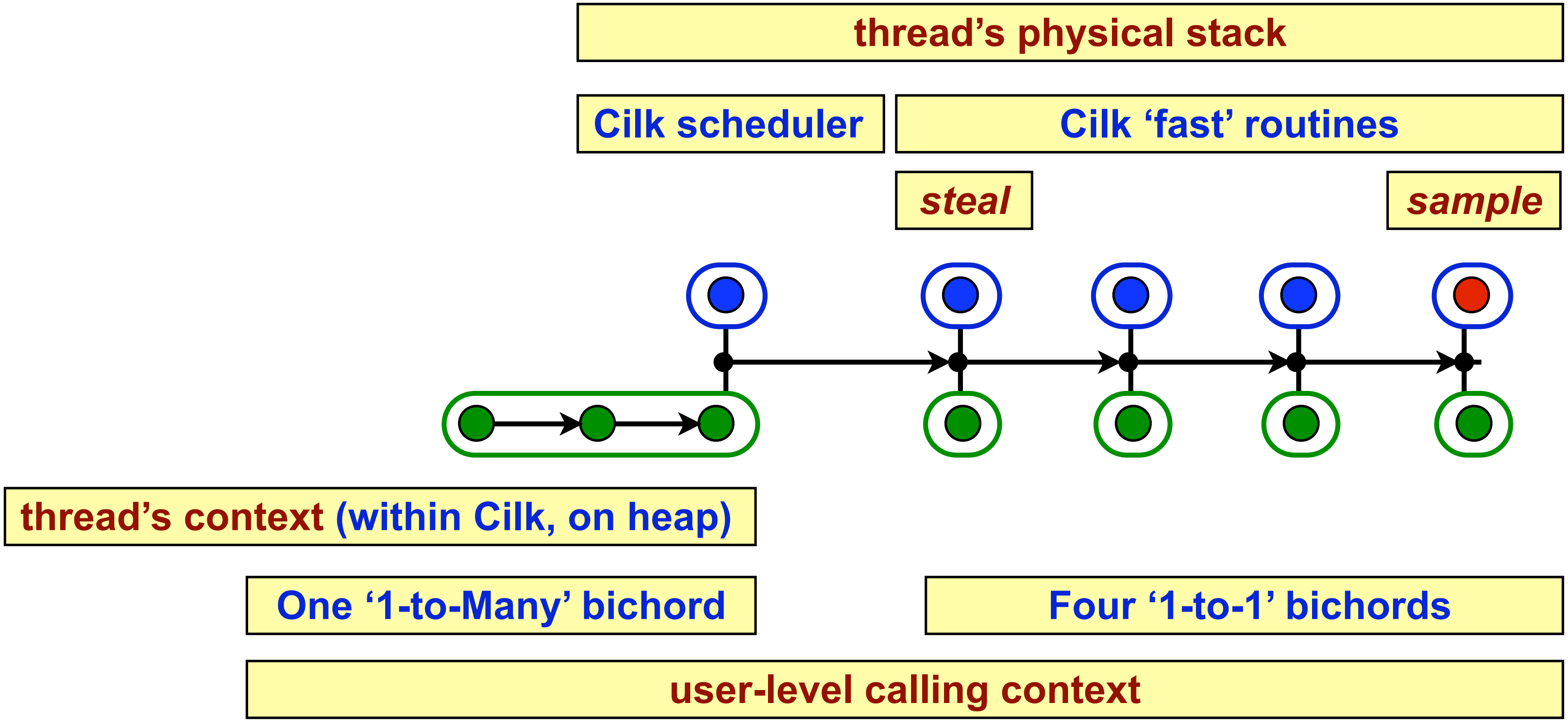


- **Logical call path: generalization of physical call path**
  - a list of 'bichords' for physical-user frame relationships



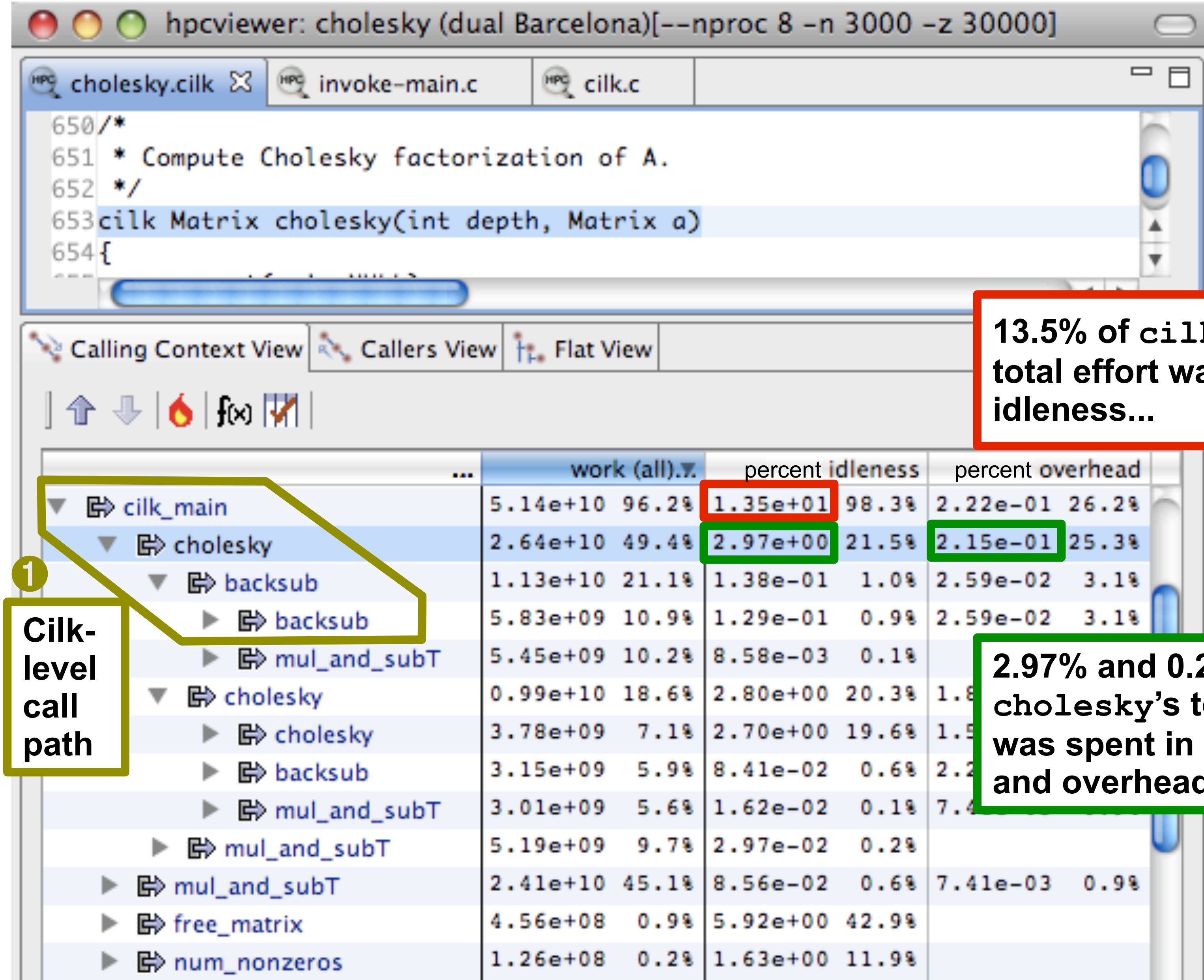
Logical Unwinding of Cilk

- The typical case (simplified):



- More details in the paper
  - — theoretical
  - — implementation

Top-down Work for Cilk ‘Cholesky’

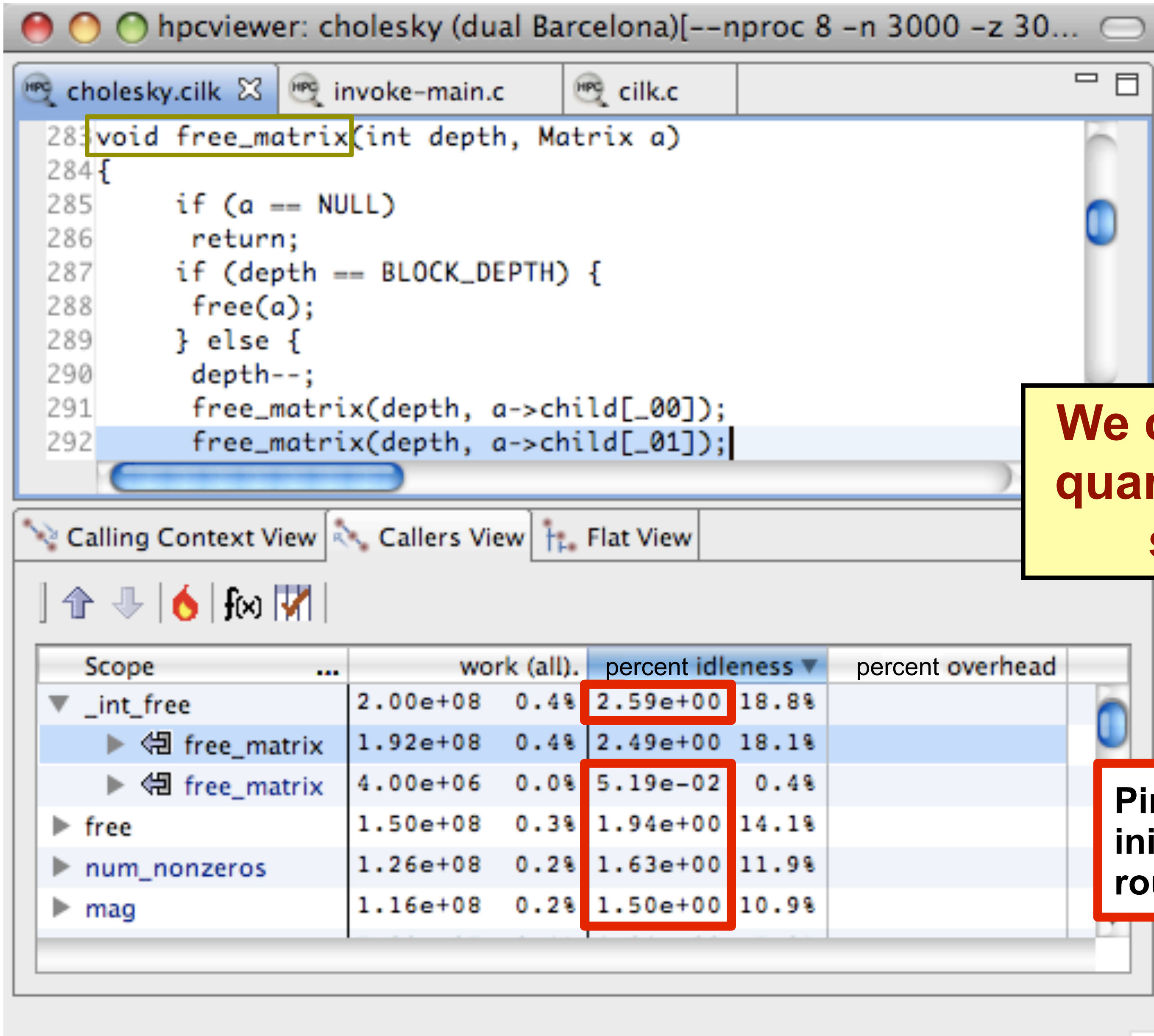


1 Cilk-level call path

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

3 13.5% of `cilk_main`’s total effort was spent in idleness...

Bottom-up Idleness for Cilk 'Cholesky'



We can pinpoint and quantify the effect of serialization.

Pinpoints serial initialization/finalization routines.

Conclusion: Effective for Work Stealing

- **Summary:**
  - — Attribute *idleness* and *overhead* to *logical contexts*
  - — Pinpoint idleness and overhead to user-level code
  - — These metrics complement traditional hardware counters
- **We have shown it is possible to:**
  - — construct efficient, effective tools for complex multithreaded languages
    - intuitive metrics
    - user-level insight
  - — provide user-level insight with only minor run-time effects
    - bridge chasm between user-level and run-time execution models
    - permit full optimization
      - – the version of the code that matters
  - — project detailed metrics to a much higher level of abstraction

What about lock contention?

- **Lock contention => idleness:**
  - explicitly threaded programs (Pthreads, etc)
  - implicitly threaded programs (critical sections in OpenMP, Cilk...)
- **Extend work stealing idea for locks:**
  - Work-stealing: blame idleness on working threads
  - Extension: blame lock waiting on lock holders
- **Maintain:**
  - W_L: threads working in a lock critical section
  - W_O: threads working otherwise
  - I_L: threads idling at a lock
  - I_O: threads idling otherwise (e.g., condition variable)
- **On sampling a working thread:**
  - if in state W_L: work = 1, idleness = I_L / W_L
  - if in state W_O: work = 1, idleness = I_O / W_O

Blame shifting: perpetrator, not suspects

- Problem with prior approach:
  - — blame is too diffuse for complex programs
  - — global counters leads to scalability problems
- Idea: communicate blame via locks (shared state)
  - — assume spin-waiting (contra sleep-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 locate lock contention in calling context
- “Blame shifting”: blames the perpetrator
  - — rather than the suspects or the victim

Blame shifting: implementation

- **Ground rules:**
  - cannot change lock library (mem. overhead when not profiling)
  - cannot have two lock libraries (requires recompilation/relink)
- **Implementation challenges for Pthreads:**
  - must
    - instrument locks to track working/idling
    - alloc out-of-band shared state (spin lock only 32-bits)
    - dynamically manage out-of-band state (cannot leak mem)
      - consider a linked structure where each node has a lock
  - problems
    - locks are used by
      - `malloc` and other glibc routines
    - locks are used very early (before profiler state may be initialized)
      - library constructors, static constructors
    - alloc shared state in a 'racy' environment
      - profiler may not be able to alloc at lock init point

The End