Performance Analysis of Multithreaded Programs

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@rice.edu

Papers for Today

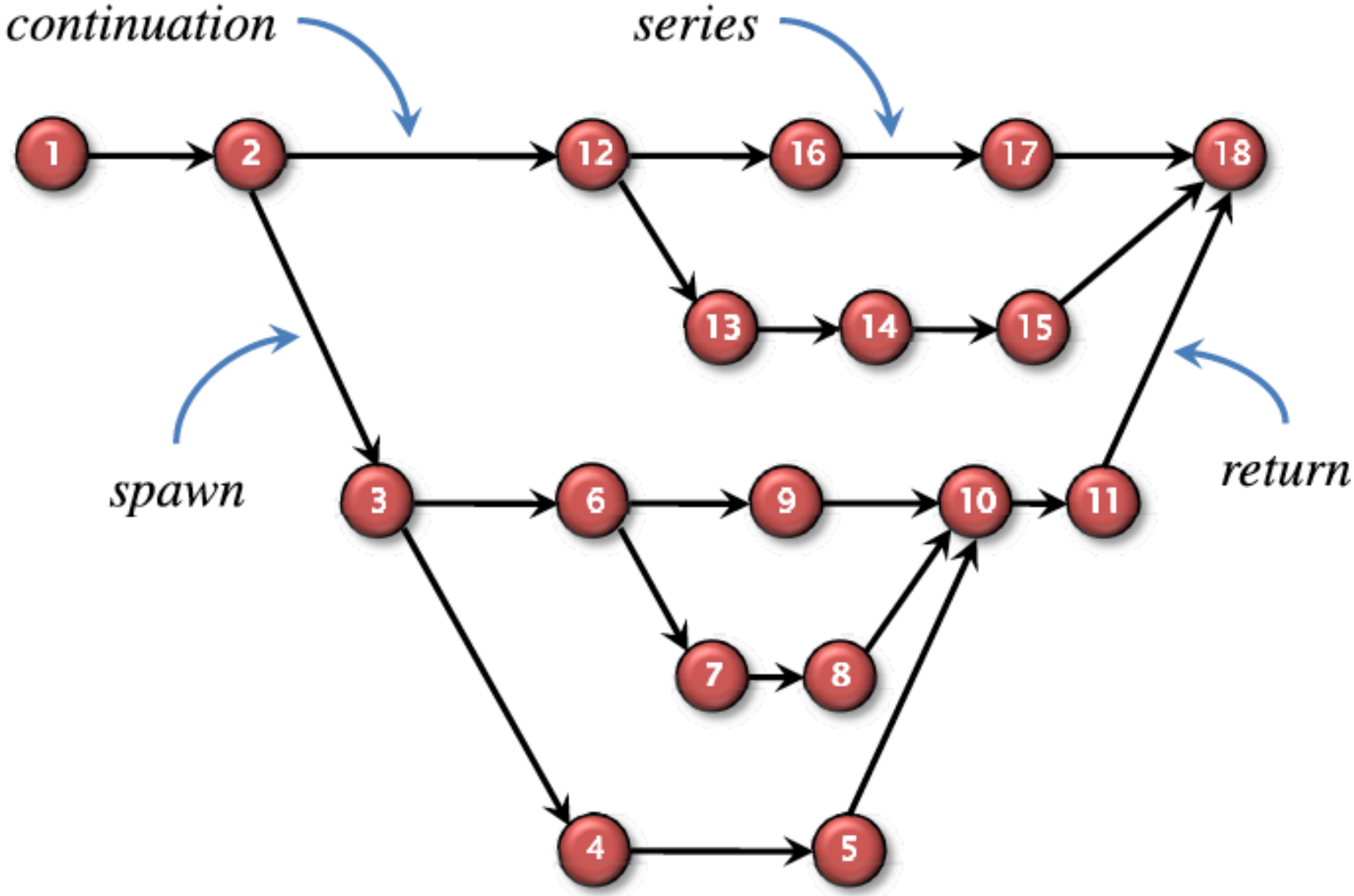
- **The Cilkview scalability analyzer.** Yuxiong He, Charles E. Leiserson, and William M. Leiserson. In Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures (SPAA '10). 2010. ACM, New York, NY, USA, 145-156.
- **A new approach for performance analysis of OpenMP programs.** Xu Liu, John Mellor-Crummey, and Michael Fagan. In Proceedings of the 27th ACM International conference on supercomputing (ICS '13). ACM, New York, NY, USA, 69-80.
- **The Cilkprof Scalability Profiler.** Tao B. Schardl, Bradley C. Kuszmaul, I-Ting Angelina Lee, William M. Leiserson, and Charles E. Leiserson. 2015. In Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures (SPAA '15). ACM, New York, NY, USA, 89-100.

Cilkview

Four Reasons for Scaling Losses in Cilk

- **Insufficient parallelism**
  - — e.g. serial code sections
- **Scheduling overhead**
  - — work is too fine grained to be distributed productively
- **Insufficient data bandwidth**
  - — contention for cache or memory bandwidth
- **Contention**
  - — for locks, false sharing

Cilk Execution DAG



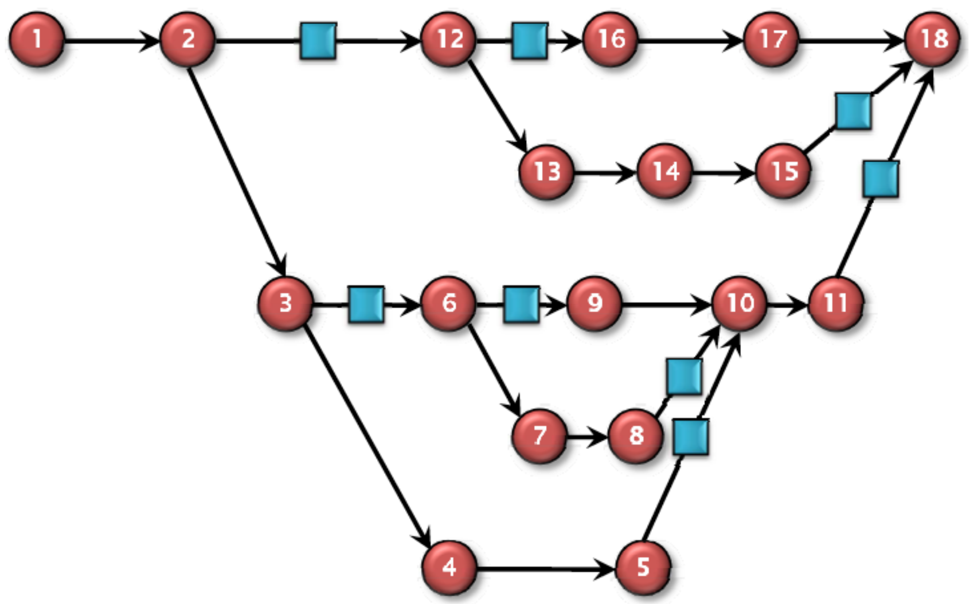
vertex = strand
edge = ordering dependencies

Upper Bounds on Speedup

- **Background**
  - work law
    - $T_p \geq T_1 / P$
  - span law
    - $T_p \geq T_\infty$
- **Bounds on speedup**
  - work bound
    - $T_1 / T_p \leq P$
  - span bound
    - $T_1 / T_p \leq T_1 / T_\infty$

Burdened DAG Model

- Performance determined not just by intrinsic parallelism, but also by the overhead of the scheduler
  - — thread migration by a steal is not free
- Model cost of potential thread migration by charging 15K cycles for each continuation and return edge



squares on return and continuation edges represent potential migration overhead

Cilkview Approach

- Use Pin binary instrumentation tool
- Insert instrumentation into the program to measure
  - number instructions along edges (work)
  - number of syncs
  - number of spawns
  - estimate addition to the critical path due to costs associated with steals along continuation and return edges
    - assume each steal may cost 15K instructions
- Perform measurements in a serial execution of the DAG
- Use projections to estimate parallel performance under a range of conditions

Performance Metrics

- **Measured metrics**
  - Work
  - Span
    - longest path through the DAG
  - Burdened span
    - longest path through the burdened DAG
  - Spawns
  - Syncs
- **Derived metrics**
  - Parallelism
    - Work / Span
  - Burdened parallelism
    - Work / (Burdened span)
  - Average maximal strand
    - Work / (1 + 2 * Spawns + Syncs)

Expected Speedup

Theorem: Let T_1 be the work of an application, and let T_b be its burdened span. Then, a work-stealing scheduler running on P processors can execute the application in expected time

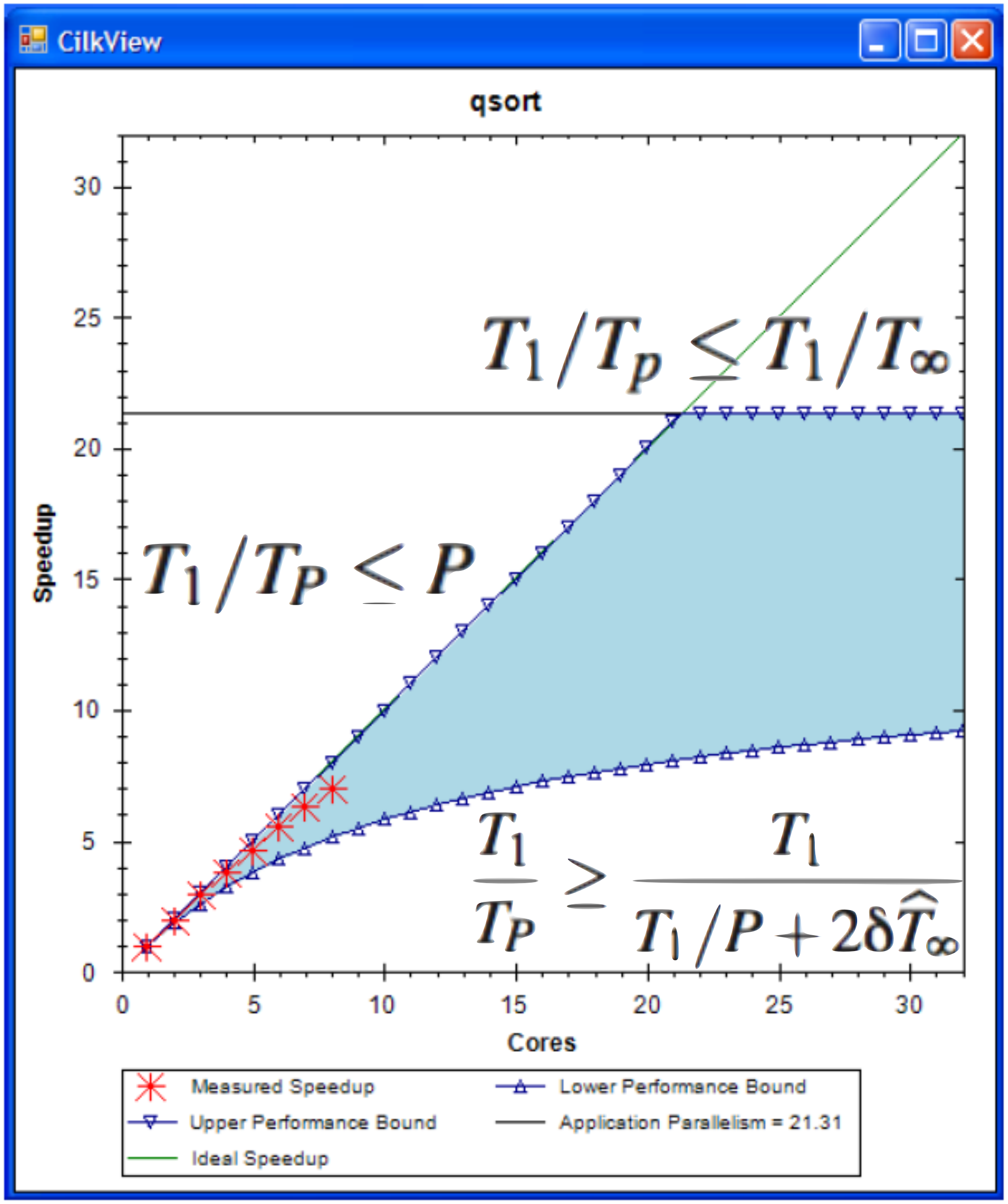
$$T_p \leq T_1 / P + 2 \delta T_b,$$

where δ is the span coefficient.

See the paper for the proof.

The proof considers the additional cost of the burden for the number of steals in the expected case and adds that to the work.

Cilkview Output for Quicksort (10M numbers)



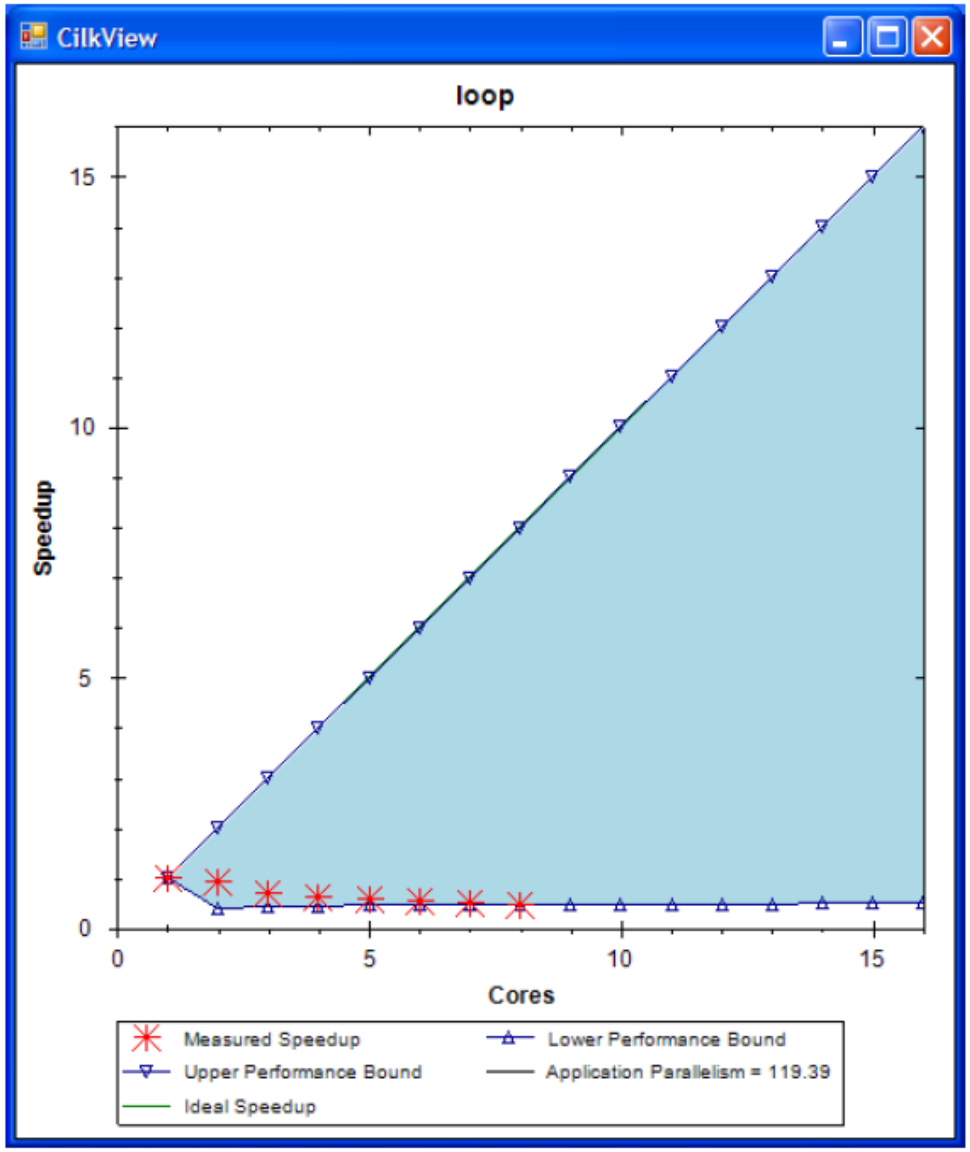
Case Study: A Stencil Computation - I

```
void stencil_loop (int t0, int t1,
        int x0, int x1, int y0, int y1,
        int z0, int z1){
  for(int t = t0; t < t1; ++t) {
    for(int z = z0; z < z1; ++z) {
      for(int y = y0; y < y1; ++y) {
        cilk_for(int x = x0; x < x1; ++x) {
          // stencil computation kernel
          stencil_kernel(t, x, y, z);
} } } } }
```

Case Study: A Stencil Computation - II

```
void stencil_loop (int t0, int t1,
     int x0, int x1, int y0, int y1,
     int z0, int z1){
  for(int t = t0; t < t1; ++t) {
    for(int z = z0; z < z1; ++z) {
      for(int y = y0; y < y1; ++y) {
        cilk_for(int x = x0; x < x1; ++x) {
          // stencil computation kernel
          stencil_kernel(t, x, y, z);
} } } } }
```

- **Parallelism ~119**
- **Large difference between span and burdened span**
- **Burdened parallelism ~.87**
  - **slowdown likely!**
- **Low burdened parallelism indicates that dynamic load balancing cost may swamp benefit of exploiting available parallelism**



Parallelizing outer loop rather than inner loop would help

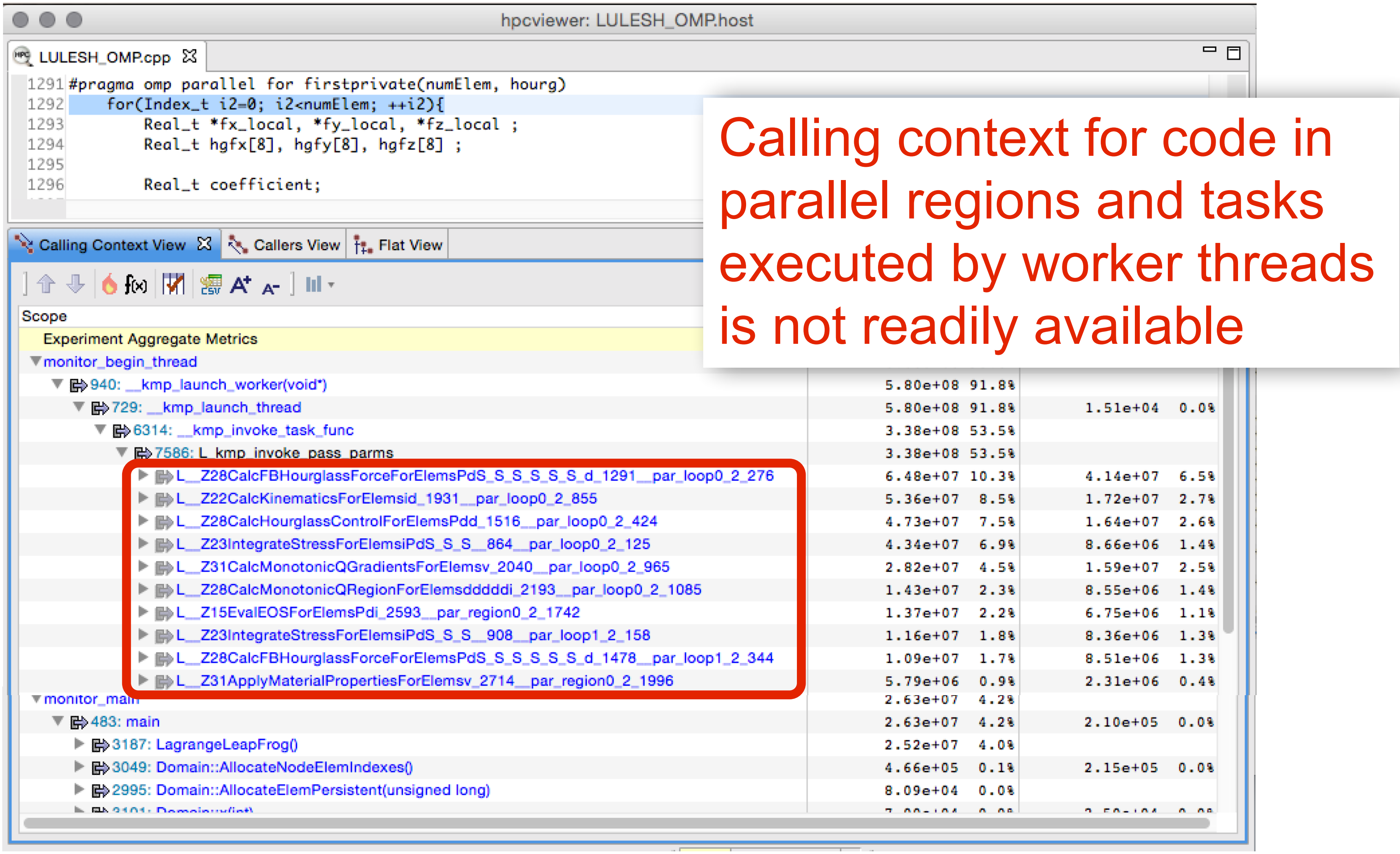
Limitations of Cilkview

- Analyzes the performance of the whole program
- Can analyze the performance of a region by inserting “start” and “stop” points in a program
  - cumbersome
  - error prone for large and complex code bases
- Tuning is equivalent to “guess and check”

Performance analysis of OpenMP

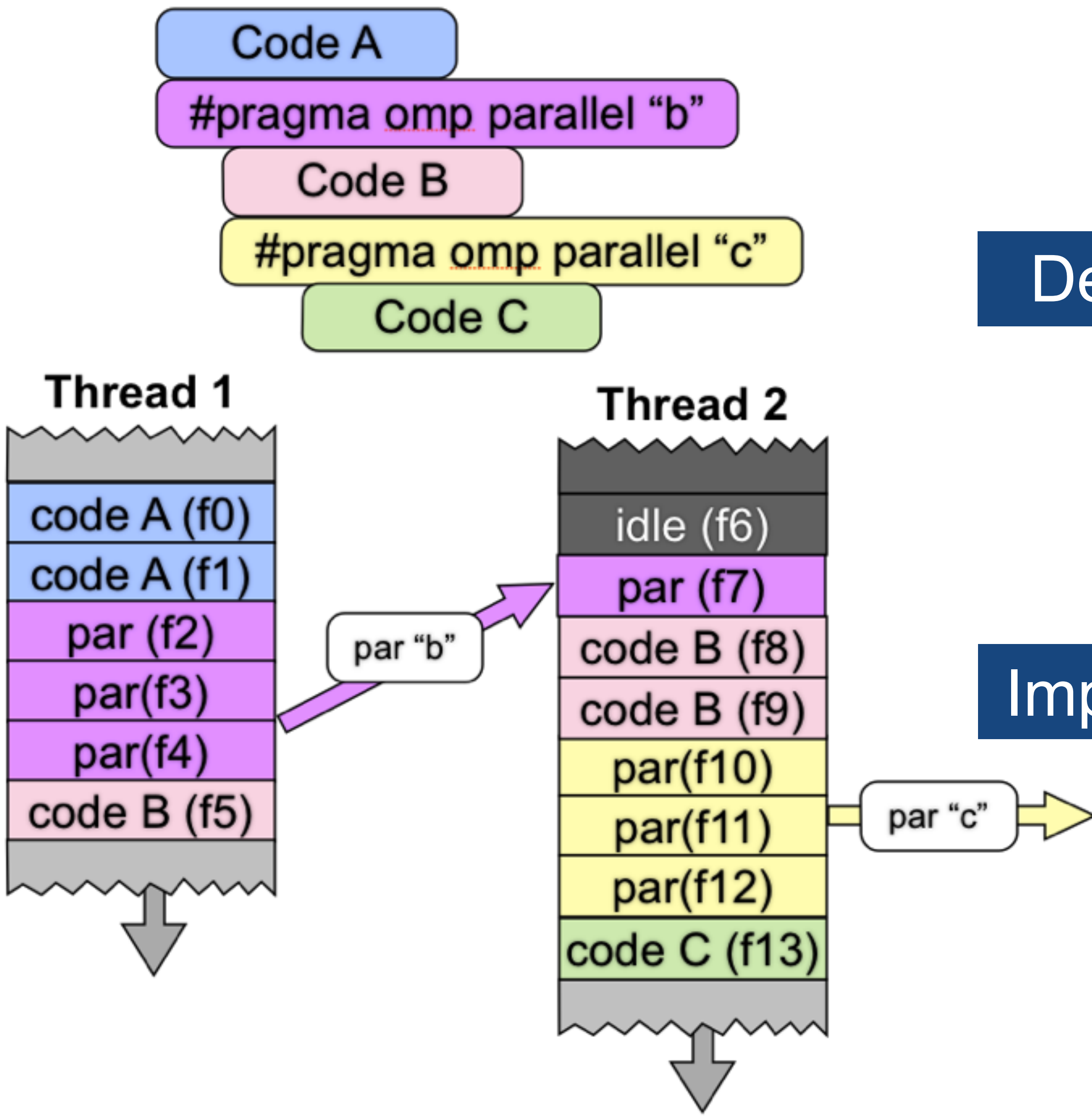
Challenge for OpenMP Tools

Typically, large gap between OpenMP source and implementation



Difficulty: OpenMP Context is Distributed

Problem: full calling context may be distributed among threads



Developer view

Implementation view

Additional Obstacles for Tools

- Differences in OpenMP implementations
  - static vs. dynamic linking
    - Oracle's collector interface for tools supports only dynamic linking
    - static linking is often preferred for supercomputers
  - threads
    - Intel: extra *shepherd* thread
    - IBM: none
  - call stack
    - GOMP: master calls outlined function from user code
    - Intel and IBM: master calls outlined function from runtime
    - PGI: cactus stack
- No standard API for runtime inquiry

OMPT: An OpenMP Tools API

- Goal: a standardized tool interface for OpenMP
  - prerequisite for portable tools for debugging and performance analysis
  - missing piece of the OpenMP language standard
- Design objectives
  - enable tools to measure and attribute costs to application source and runtime system
    - support low-overhead tools based on asynchronous sampling
    - attribute to user-level calling contexts
    - associate a thread's activity at any point with a descriptive state
  - minimize overhead if OMPT interface is not in use
    - features that may increase overhead are optional
  - define interface for trace-based performance tools
  - don't impose an unreasonable development burden
    - runtime implementers
    - tool developers

Major OMPT Functionality

- **State tracking**
  — threads maintain state at all times (e.g., working, waiting, idle)
  — a tool can query this state at any time (async signal safe)
- **Call stack interpretation**
  — inquiry functions enable tools to reconstruct application-level call stacks from implementation-level information
    – identify which frames on the call stack belong to the runtime system
- **Event notification callbacks for predefined events**
  — mandatory callbacks for threads, parallel regions, and tasks
  — optional callbacks for identifying idleness and attributing blame
  — optional callbacks for tracing activity for all OpenMP constructs
- **Target device monitoring**
  — collect event trace on target
  — inspect, process, and record target events on host

OMPT Callbacks

```
ompt_callback_thread_begin
ompt_callback_thread_end
ompt_callback_parallel_begin
ompt_callback_parallel_end
ompt_callback_task_create
ompt_callback_task_schedule
ompt_callback_implicit_task
ompt_callback_target
ompt_callback_target_data_op
ompt_callback_target_submit
ompt_callback_control_tool
ompt_callback_device_initialize
ompt_callback_device_finalize
ompt_callback_device_load
ompt_callback_device_unload
ompt_callback_sync_region_wait
```

```
ompt_callback_mutex_released
ompt_callback_dependences
ompt_callback_task_dependence
ompt_callback_work
ompt_callback_master
ompt_callback_target_map
ompt_callback_sync_region
ompt_callback_lock_init
ompt_callback_lock_destroy
ompt_callback_mutex_acquire
ompt_callback_mutex_acquired
ompt_callback_nest_lock
ompt_callback_flush
ompt_callback_cancel
ompt_callback_reduction
ompt_callback_dispatch
```

OMPT Callback API Requirements

Return code abbreviation	N	S/P	A
ompt_callback_thread_begin			*
ompt_callback_thread_end			*
ompt_callback_parallel_begin			*
ompt_callback_parallel_end			*
ompt_callback_task_create			*
ompt_callback_task_schedule			*
ompt_callback_implicit_task			*
ompt_callback_target			*
ompt_callback_target_data_op			*
ompt_callback_target_submit			*
ompt_callback_control_tool			*
ompt_callback_device_initialize			*
ompt_callback_device_finalize			*
ompt_callback_device_load			*
ompt_callback_device_unload			*
ompt_callback_sync_region_wait	*	*	*
ompt_callback_mutex_released	*	*	*
ompt_callback_dependences	*	*	*
ompt_callback_task_dependence	*	*	*
ompt_callback_work	*	*	*
ompt_callback_master	*	*	*
ompt_callback_target_map	*	*	*
ompt_callback_sync_region	*	*	*
ompt_callback_reduction	*	*	*
ompt_callback_lock_init	*	*	*
ompt_callback_lock_destroy	*	*	*
ompt_callback_mutex_acquire	*	*	*
ompt_callback_mutex_acquired	*	*	*
ompt_callback_nest_lock	*	*	*
ompt_callback_flush	*	*	*
ompt_callback_cancel	*	*	*
ompt_callback_dispatch	*	*	*

N = ompt_set_never   S = ompt_set_sometimes
P = ompt_set_sometimes_paired   A = ompt_set_always

OMPT Introspection API

“ompt_enumerate_states”

“ompt_enumerate_mutex_impls”

“ompt_set_callback”

“ompt_get_callback”

“ompt_get_thread_data”

“ompt_get_num_places”

“ompt_get_place_proc_ids”

“ompt_get_place_num”

“ompt_get_partition_place_nums”

“ompt_get_proc_id”

“ompt_get_state”

“ompt_get_parallel_info”

“ompt_get_task_info”

“ompt_get_task_memory”

“ompt_get_num_devices”

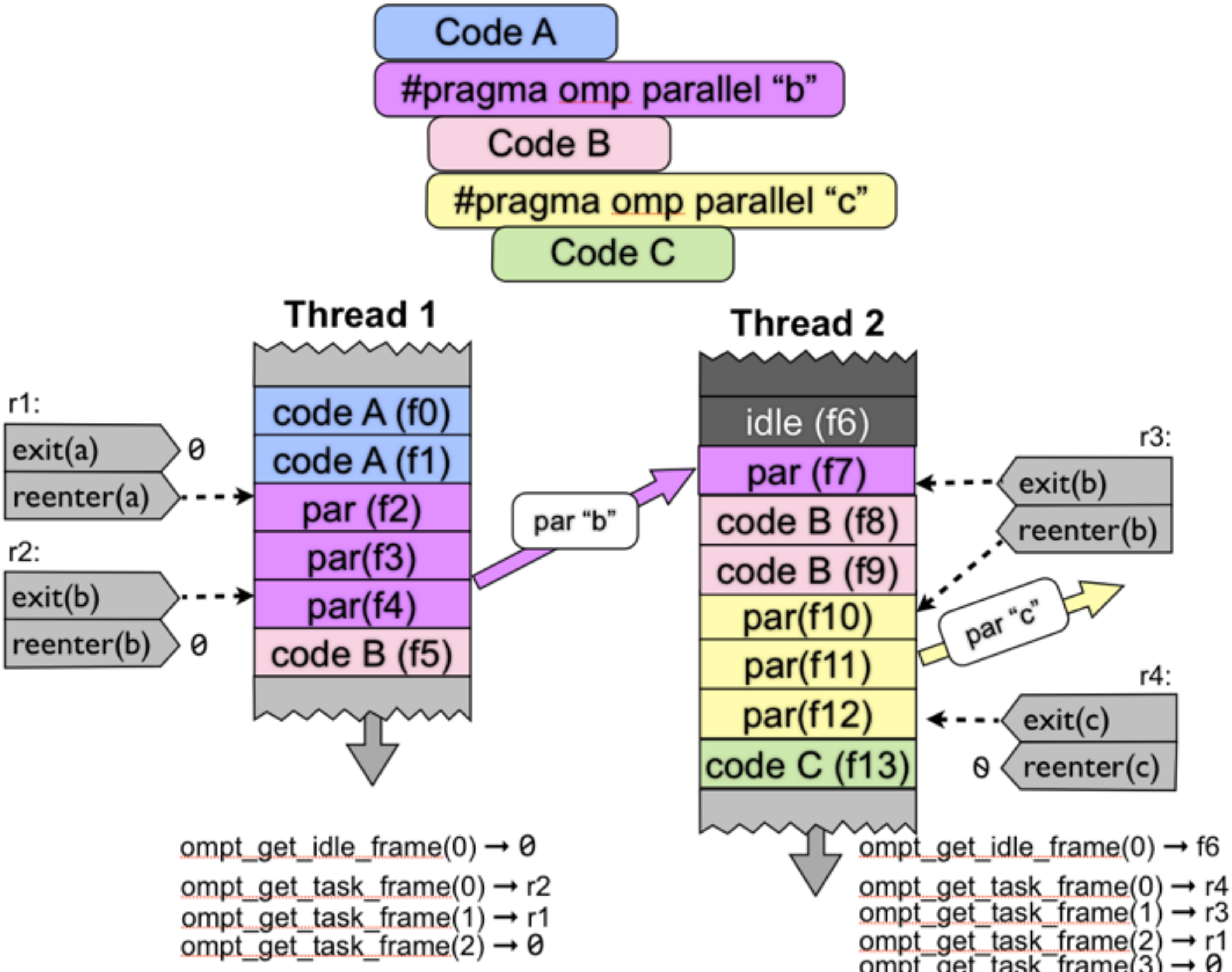
“ompt_get_num_procs”

“ompt_get_target_info”

“ompt_get_unique_id”

“ompt_finalize_tool”

Understanding Call Stacks of OpenMP



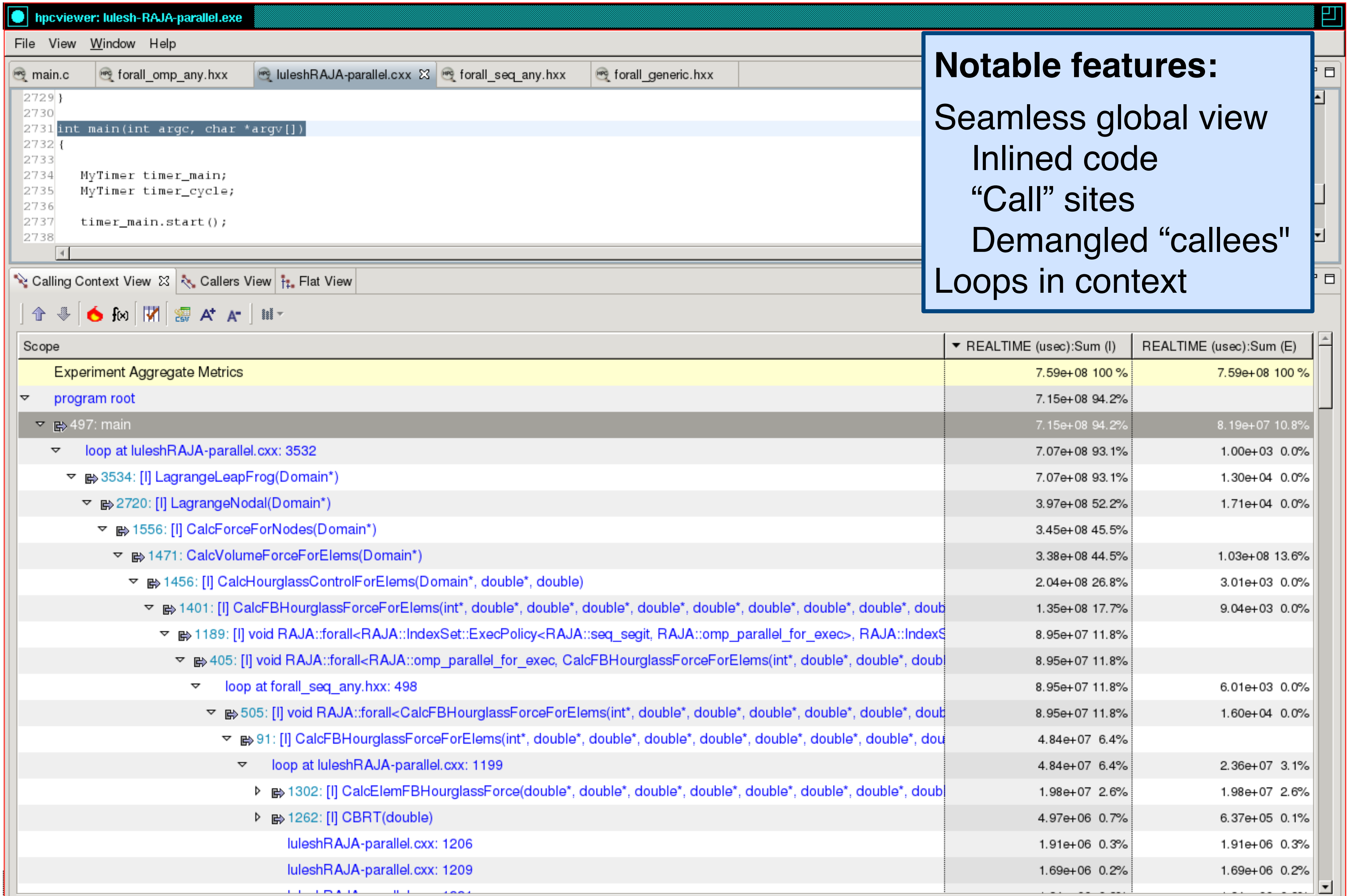
Case Study: LLNL's LULESH with RAJA

Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics

- Implementation using RAJA portability model
- Compiled with high optimization
  — icpc -g -O3 -msse4.1 -align -inline-max-total-size=20000 -inline-forceinline -ansi-alias -std=c++0x -openmp -debug inline-debug-info -parallel-source-info=2 -debug all
- Linked with OMPT-enabled LLVM OpenMP runtime
- Data collection
  — hpcrun -e REALTIME@1000 ./lulesh-RAJA-parallel.exe
    – implicitly uses the OMPT performance tools interface, which is enabled in our OMPT-enhanced version of the Intel LLVM OpenMP runtime

2 18-core Haswell
72+1 threads

Case Study: LLNL's LULESH with



Notable features:

Seamless global view
- Inlined code
- "Call" sites
- Demangled "callees"

Loops in context

Blame-shifting: Analyze Thread Performance

	Problem	Approach
Undirected Blame Shifting[1,3]	A thread is idle waiting for work	Apportion blame among working threads for not shedding enough parallelism to keep all threads busy
Directed Blame Shifting[2,3]	A thread is idle waiting for a mutex	Blame the thread holding the mutex for idleness of threads waiting for the mutex

[1]Tallent & Mellor-Crummey: PPoPP 2009

[2]Tallent, Mellor-Crummey, Porterfield: PPoPP 2010

[3]Liu, Mellor-Crummey, Fagan: ICS 2013

Blame-shifting Metrics for OpenMP

- **OMP_IDLE**
  - attribute idleness to insufficiently-parallel code being executed by other threads
- **OMP_MUTEX**
  - attribute waiting for locks to code holding the lock
    - attribute to the lock release as a proxy

- **Measuring these metrics requires sampling using using a time-based sample source**
  - REALTIME, CPUTIME, cycles

HPCToolkit's Support for OMPT & OpenMP

Simplified sketch

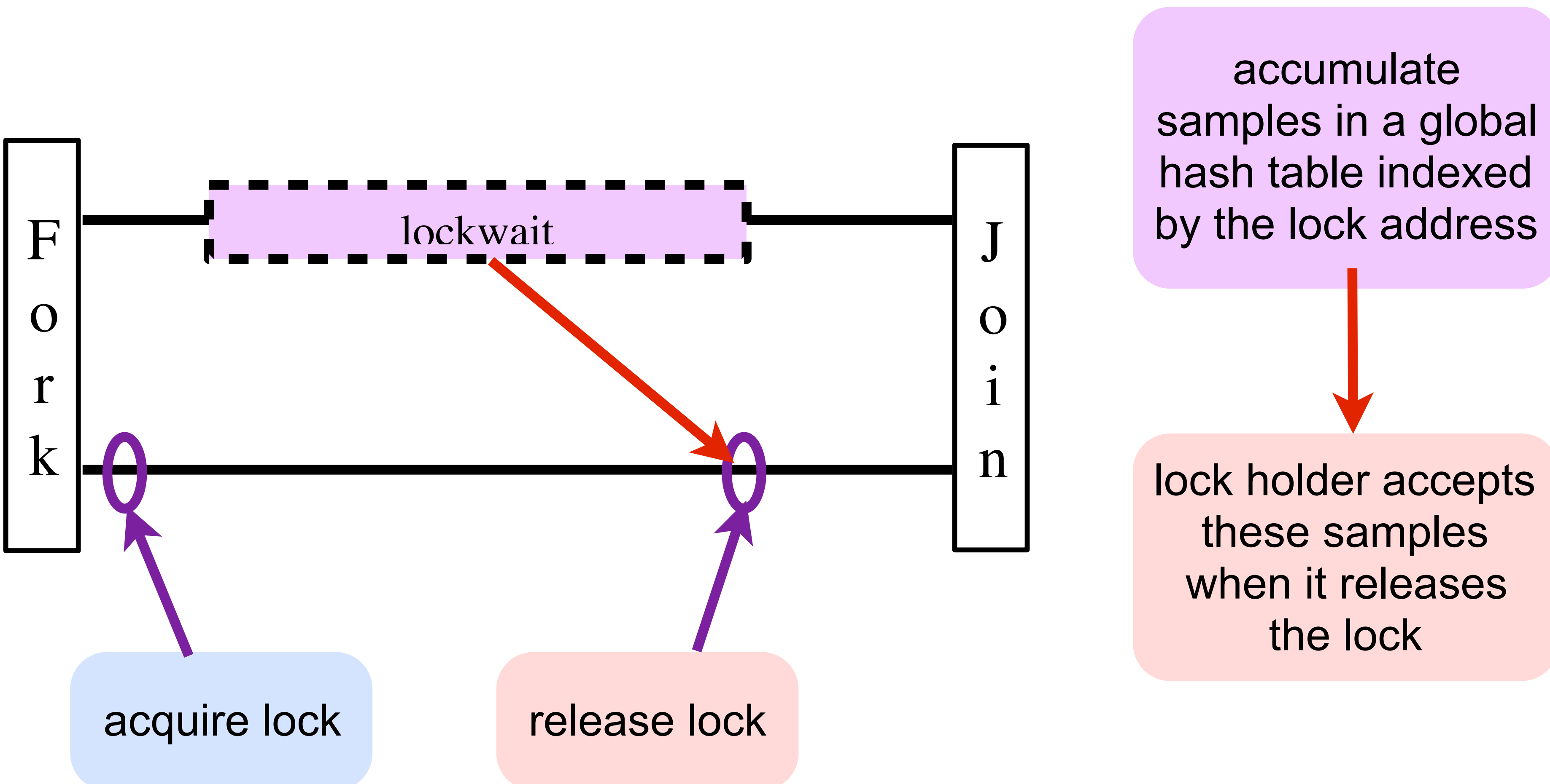
- **Initialization: install callbacks**
  - — mandatory: thread begin/end, parallel region & task begin/end
  - — blame shifting: wait begin/end, mutex release
- **When a profiling trigger fires**
  - — if thread is waiting
    - – apply blame shifting to attribute idleness to working threads
  - — if thread is working
    - – accept undirected blame for idleness of others
    - – attribute work and blame to application-level calling context
- **When a mutex release occurs**
  - — accept directed blame charged to that mutex
  - — attribute blame to application-level calling context

Attribute costs to application-level calling context

- – unwind call stack
- – elide OpenMP runtime frames using OMPT frame information
- – use info about nesting of tasks & regions to reconstruct full context

Directed Blame Shifting

- **Example:**
  - — **threads waiting at a lock are the symptom**
  - — **the cause is the lock holder**
- **Approach: blame lock waiting on lock holder**

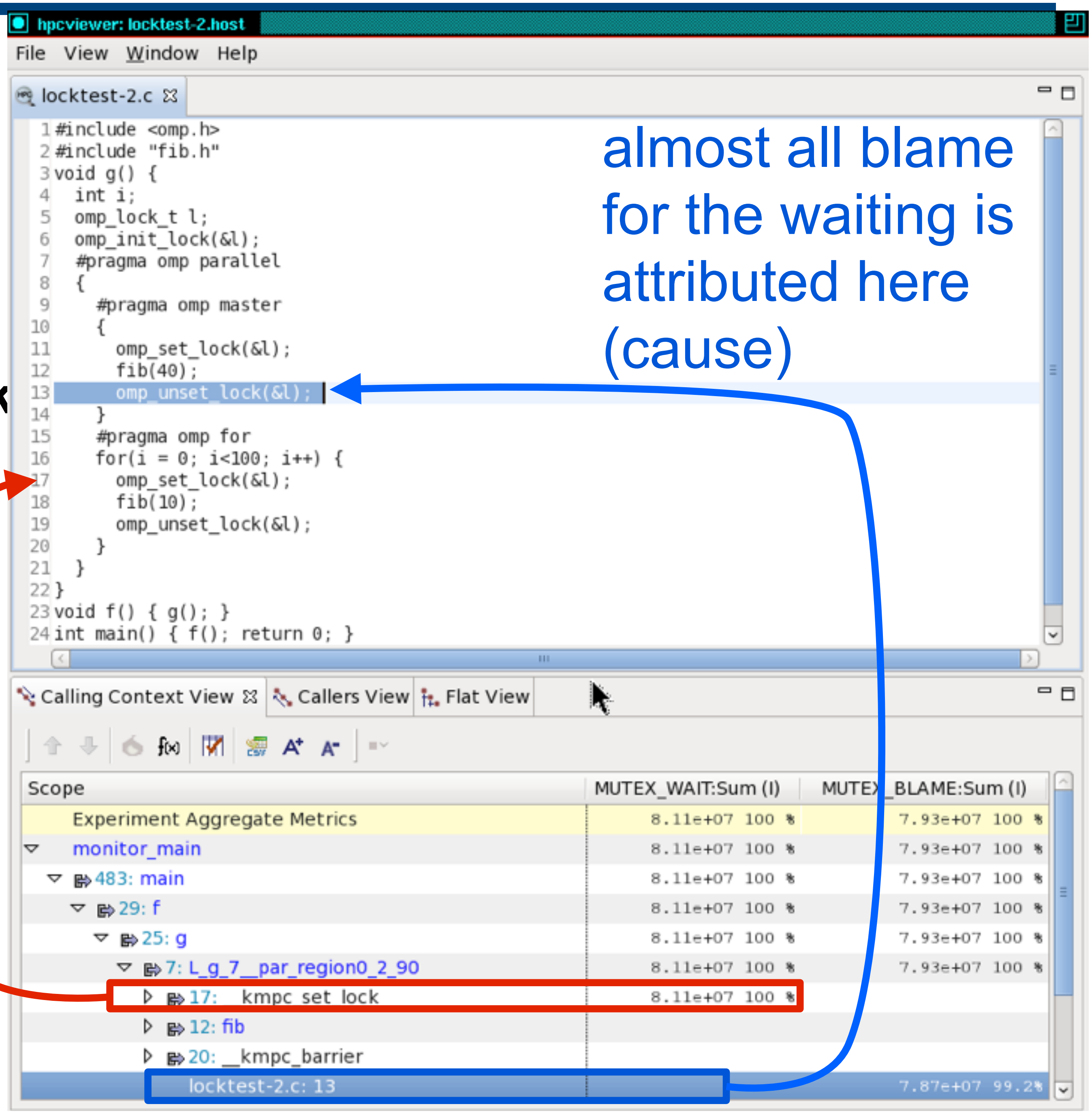


Example: Directed Blame Shifting for Locks

Blame a lock holder for delaying waiting threads

- **Charge all samples that threads receive while awaiting a lock to the lock itself**
- **When releasing a lock, accept blame at the lock**

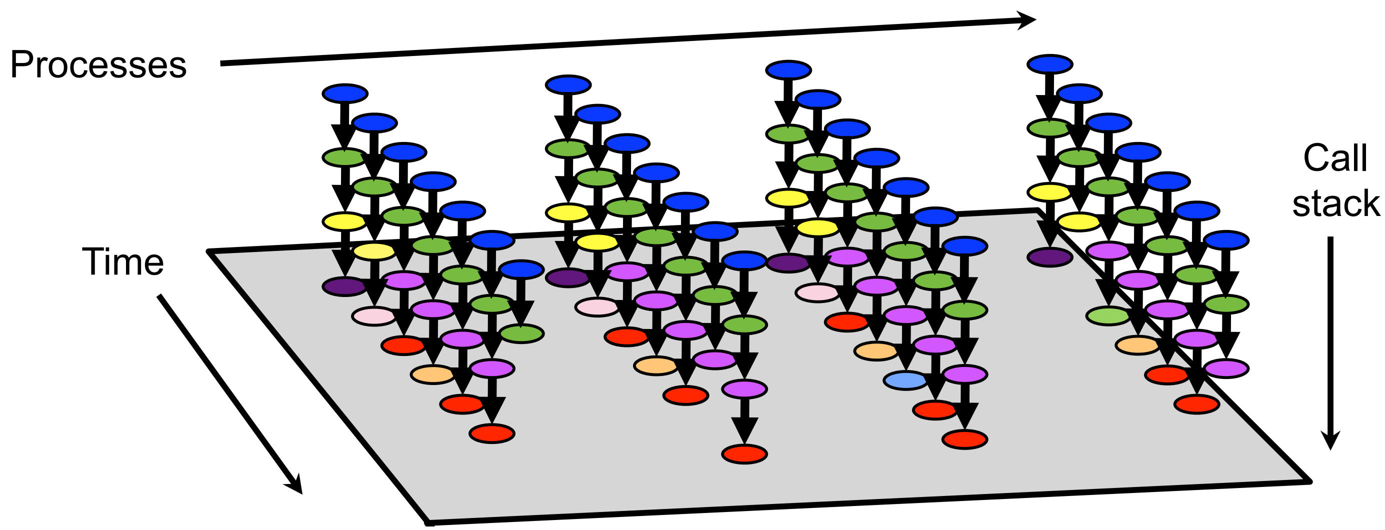
all of the waiting occurs here (symptom)



Understanding Temporal Behavior

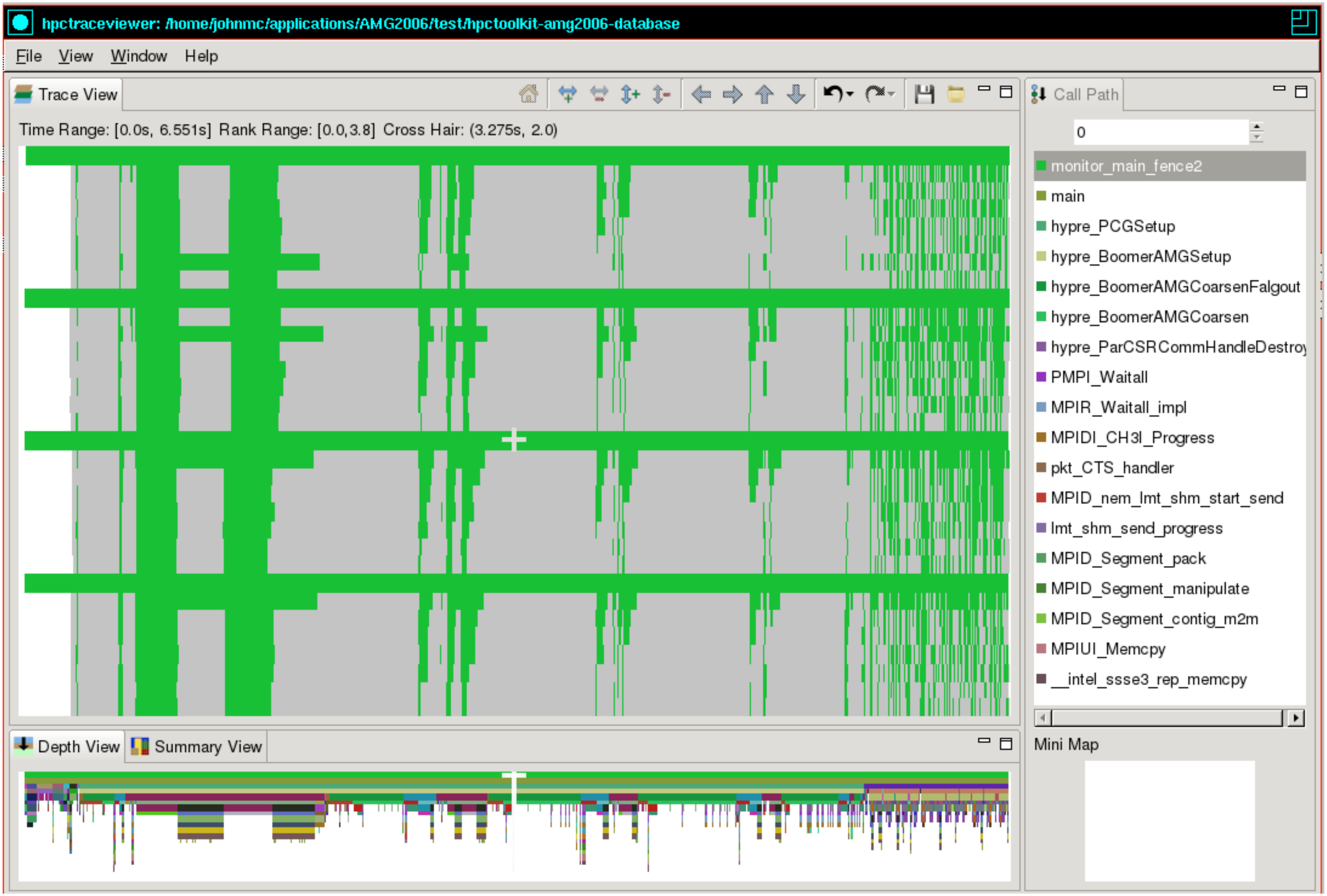
- **Profiling compresses out the temporal dimension**
  — **temporal patterns, e.g. serialization, are invisible in profiles**
- **What can we do? Trace call path samples**
  — **sketch:**
    - **N times per second, take a call path sample of each thread**
    - **organize the samples for each thread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



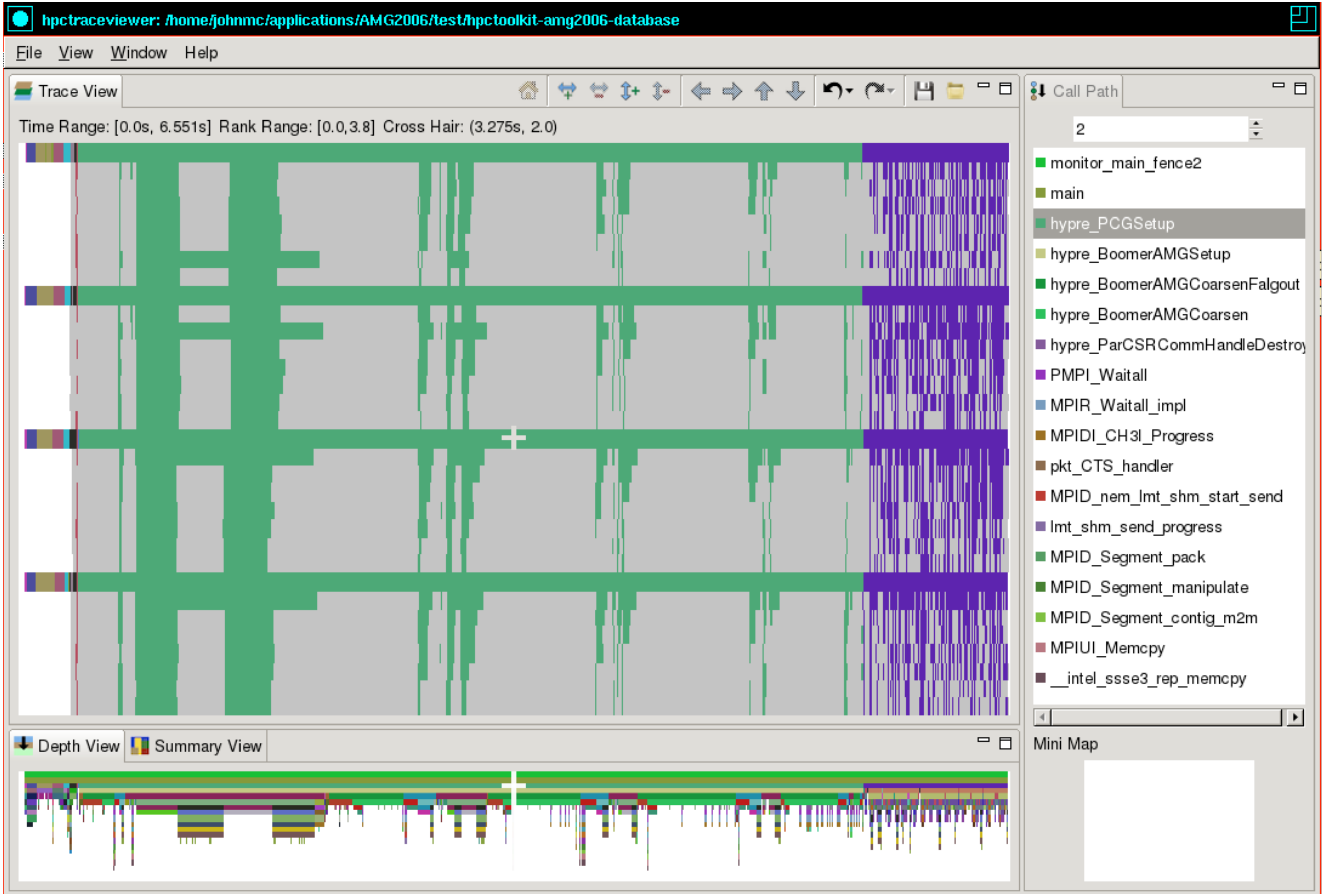
2 18-core Haswell
4 MPI ranks
6+3 threads per rank

Case Study: AMG2006



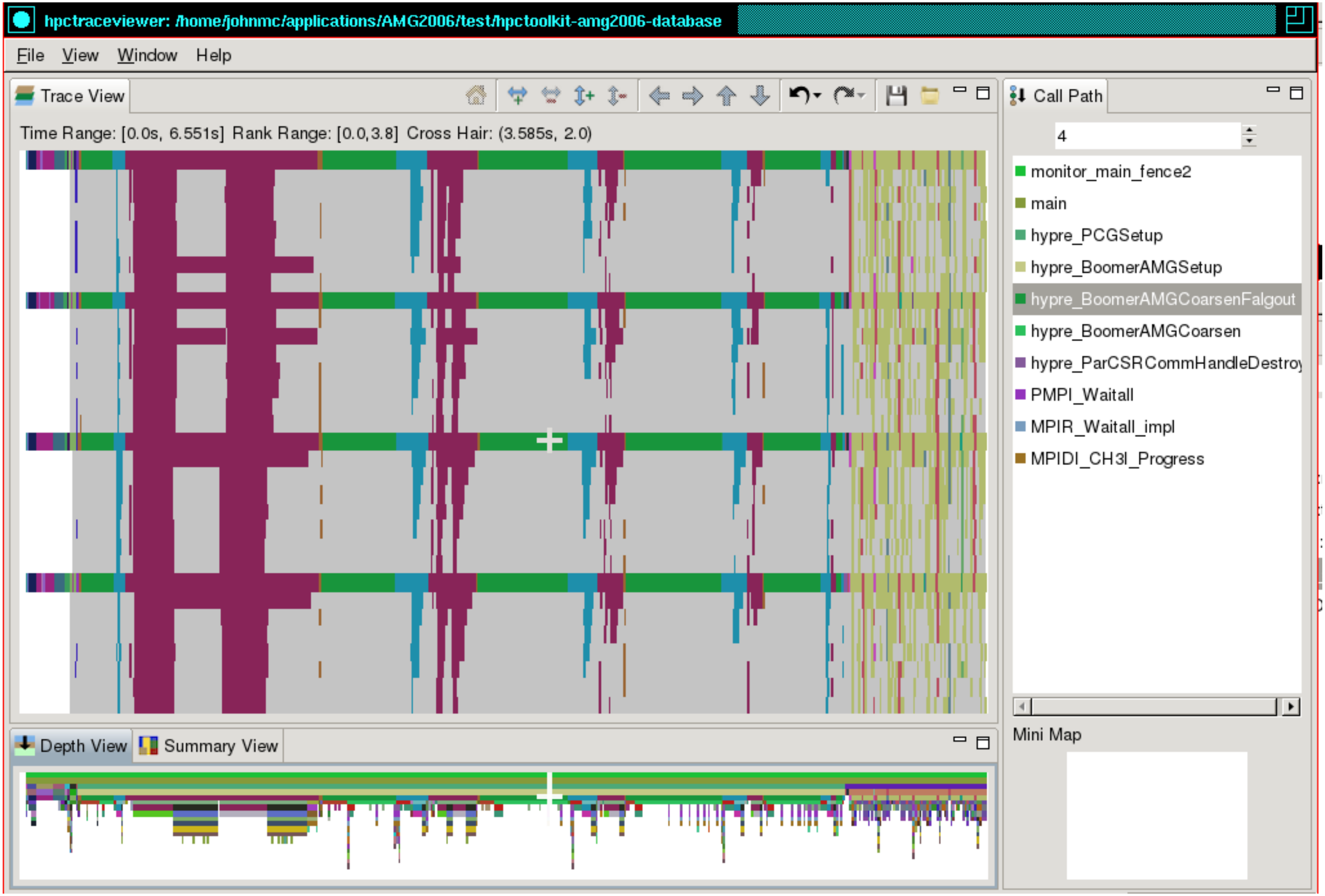
2 18-core Haswell
4 MPI ranks
6+3 threads per rank

Case Study: AMG2006



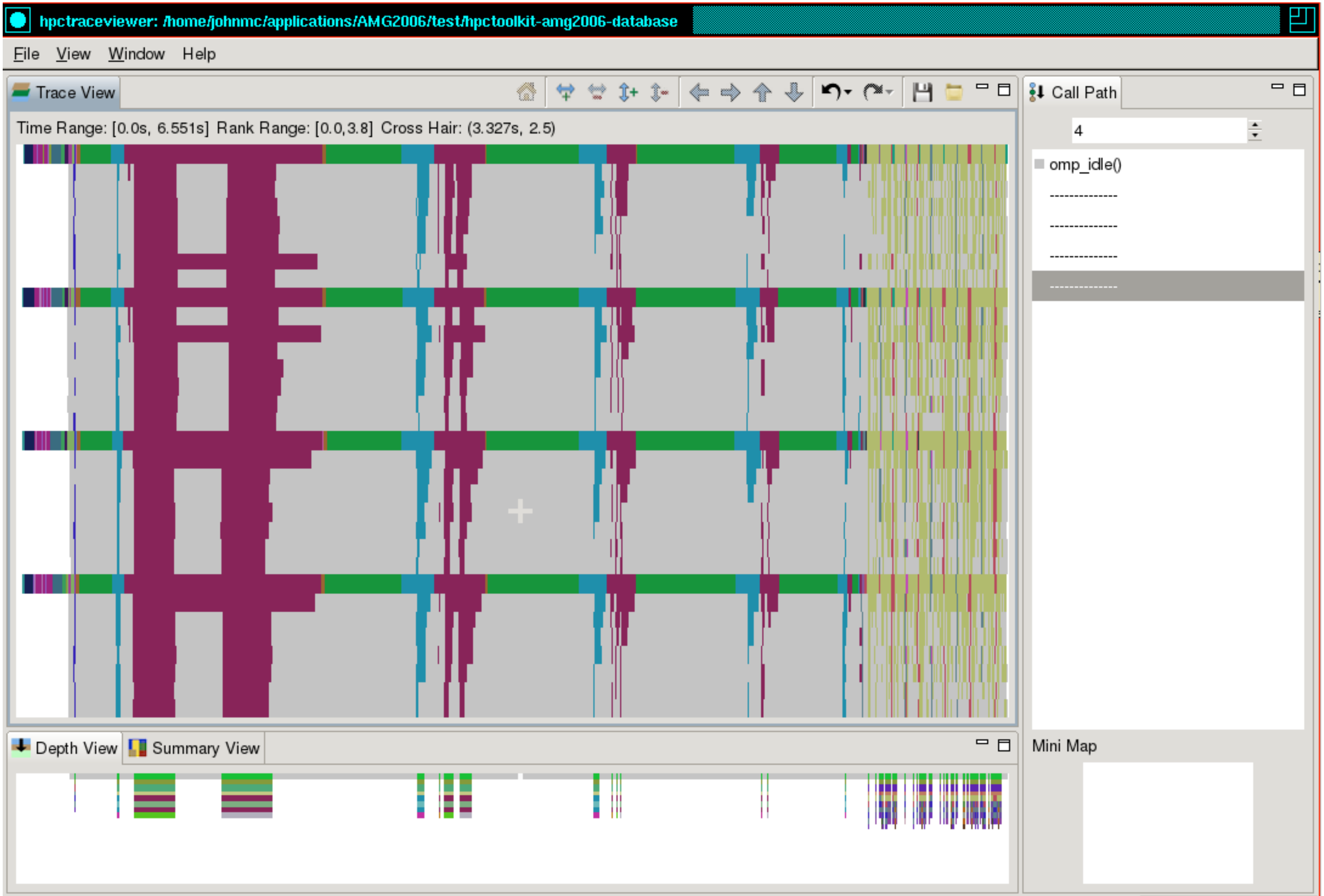
2 18-core Haswell
4 MPI ranks
6+3 threads per rank

Case Study: AMG2006



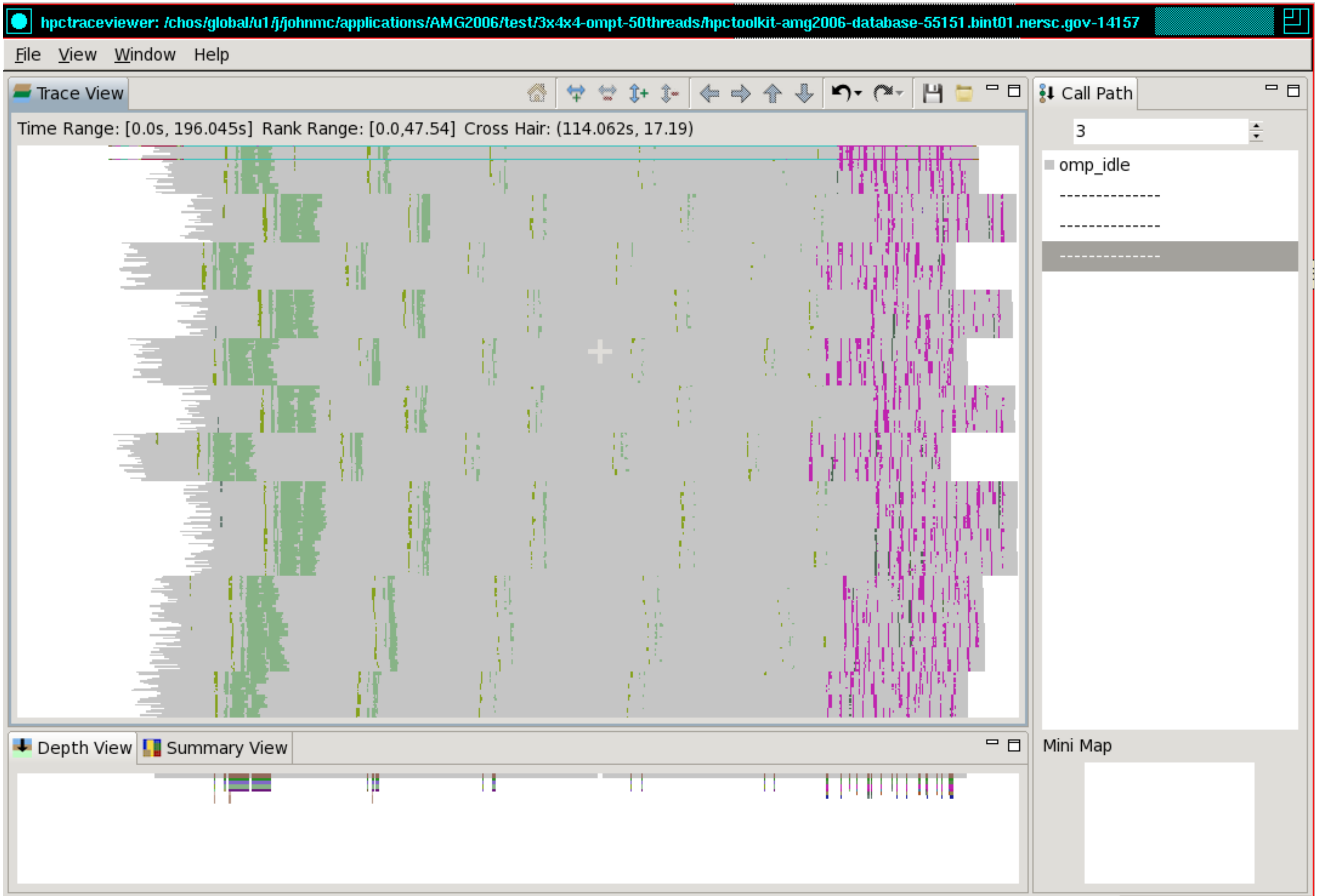
2 18-core Haswell
4 MPI ranks
6+3 threads per rank

Case Study: AMG2006



12 nodes on Babbage@NERSC
24 Xeon Phi
48 MPI ranks
50+5 threads per rank

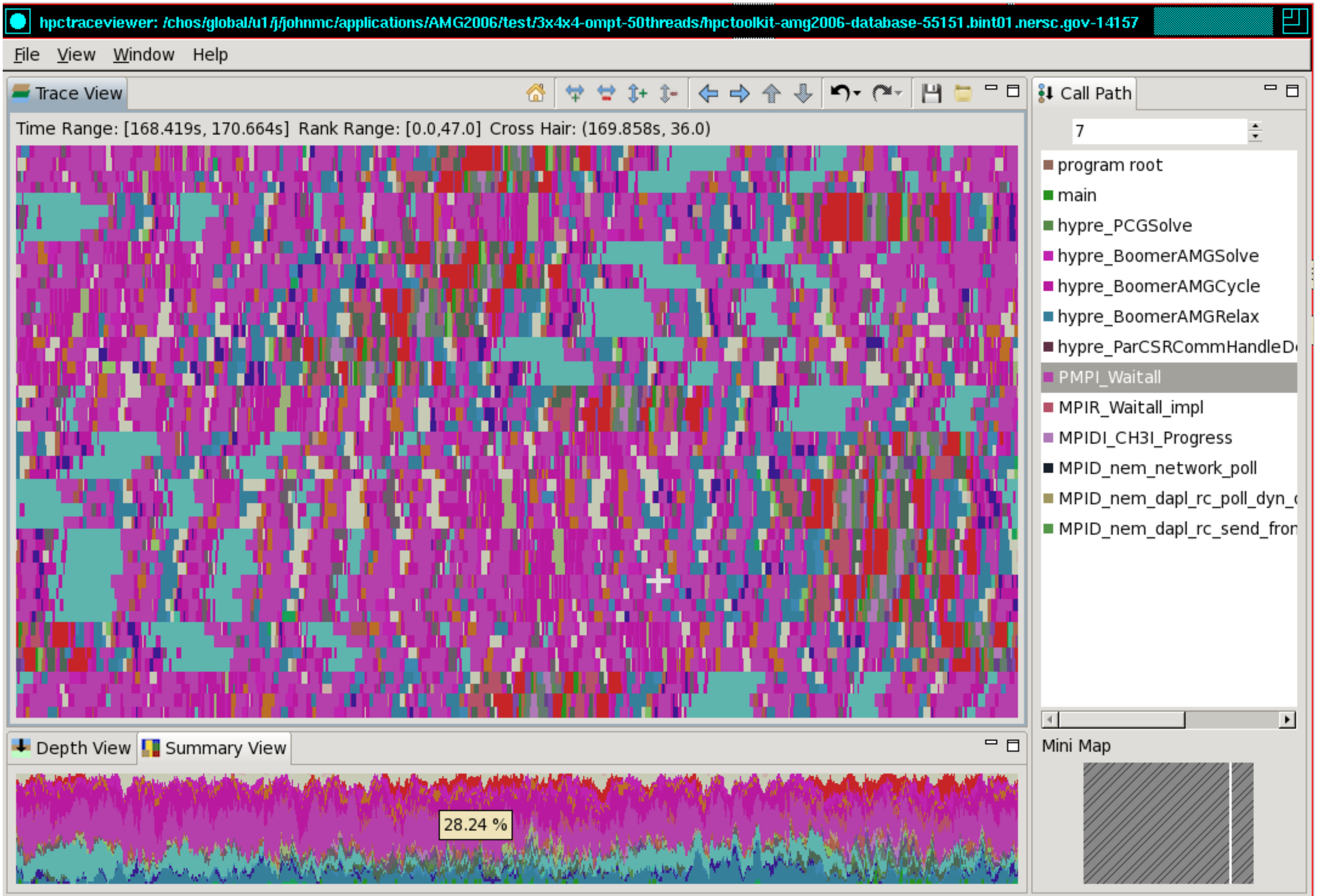
Case Study: AMG2006



12 nodes on Babbage@NERSC
24 Xeon Phi
48 MPI ranks
50+5 threads per rank

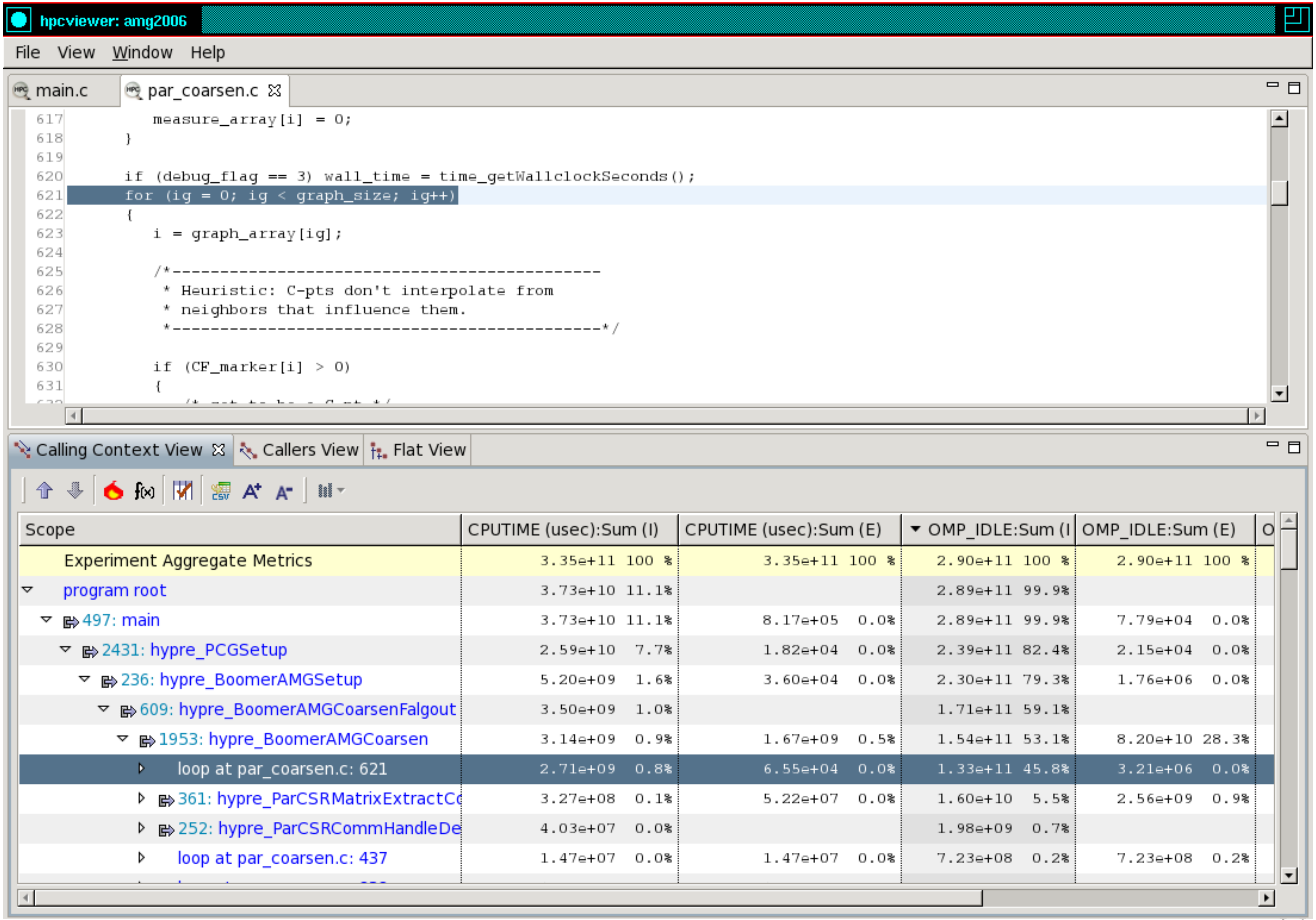
Case Study: AMG2006

Slice
Thread 0 from each MPI rank



12 nodes on Babbage@NERSC
24 Xeon Phi
48 MPI ranks
50+5 threads per rank

Case Study: AMG2006



Cilkprof

Cilkprof

- Cilkprof uses compiler instrumentation to gather detailed information about a Cilk program execution*
  - measures how much work and span of the overall computation is attributable to the subcomputation that begins when the function invoked at that call site is called or spawned and that ends when that function returns
  - analysis enables a programmer to evaluate the scalability of that call site — the scalability of the computation attributable to that call site — and how it affects the overall computation's scalability
- Currently, the tool lacks a user interface: it merely dumps a spreadsheet that relates costs to each call site

*Cilkview uses dynamic binary instrumentation with Pin to measure work.

Maintaining Work-Span Variables

For each function F, maintain work-span variables in shadow stack alongside the function call stack

- **Let u represent the spawn of F's child with the longest span so far. u is initialized to the beginning of F on entry to F.**
- **F.w: work**
  - **— work executed in the function so far**
- **F.p: prefix**
  - **— span of the trace starting from the first instruction of *F* and ending with *u***
  - **— *F.p* is guaranteed to be on the critical path of *F***
- ***F*.l: *longest-child***
  - **— span of the trace from the start of *F* through the return of the child that *F* spawns at *u***
- ***F.c: continuation***
  - **— the span of the trace from the continuation of *u* through the most recently executed instruction in *F***

Cilkprof Algorithm

F spawns or calls G:

$G.w = 0$
$G.p = 0$
$G.\ell = 0$
$G.c = 0$

Called G returns to F:

$G.p \mathrel{+}= G.c$
$F.w \mathrel{+}= G.w$
$F.c \mathrel{+}= G.p$

Spawned G returns to F:

$G.p \mathrel{+}= G.c$
$F.w \mathrel{+}= G.w$
if $F.c + G.p > F.\ell$
    $F.\ell = G.p$
    $F.p \mathrel{+}= F.c$
    $F.c = 0$

F syncs:

if $F.c > F.\ell$
    $F.p \mathrel{+}= F.c$
else
    $F.p \mathrel{+}= F.\ell$
$F.c = 0$
$F.\ell = 0$

F executes an instruction:

$F.w \mathrel{+}= 1$
$F.c \mathrel{+}= 1$

Performance Metrics

- A Cilkprof measurement for a call site s consists of the following values for a set of invocations of s
  - execution count
    - the number of invocations of s accumulated in the profile
  - call-site work
    - the sum of the work of those invocations
  - the call-site span
    - the sum of the spans of those invocations
- Cilkprof additionally computes the parallelism of s as the ratio of s's call-site work and call-site span
  - without recursive functions, Cilkprof could simply aggregate all executions of each call site
  - for recursive functions, must avoid overcounting the call-site work and call-site span

Space and Time Complexity

- For a Cilk program that
  - executes in T1 time
  - has stack depth D
- Cilkprof's work-span algorithm
  - runs in O(T1) time
  - using O(D) extra storage

Case Study with Quicksort

```
1  int partition(long array[], int low, int high) {
2    long pivot = array[low + rand(high - low)];
3    int l = low - 1;
4    int r = high;
5    while (true) {
6      do { ++l; } while (array[l] < pivot);
7      do { --r; } while (array[r] > pivot);
8      if (l < r) {
9        long tmp = array[l];
10       array[l] = array[r];
11       array[r] = tmp;
12     } else {
13       return (l == low ? l + 1 : l);
14 } } }

16 void pqsort(long array[], int low, int high) {
17   if (high - low < COARSENING) {
18     // base case: sort using insertion sort
19   } else {
20     int part = partition(array, low, high);
21     cilk_spawn pqsort(array, low, part);
22     pqsort(array, part, high);
23     cilk_sync;
24 } }

26 int main(int argc, char *argv[]) {
27   int n;
28   long *A;
29   // parse arguments
30   // initialize array A of size n
31   pqsort(A, 0, n);
32   // do something with A
33   return 0;
34 }
```

On work

Line	T_1	T_∞	T_1/T_∞
20	408,150,528	408,150,528	1.0
21	741,312,781	116,591,841	6.4
22	761,041,165	125,360,000	6.1
31	790,518,060	141,902,681	5.6

On span

T_1	T_∞	T_1/T_∞	*Local* T_1	*Local* T_∞
141,891,291	141,891,291	1.0	141,891,291	141,891,291
597,298,216	98,119,730	6.1	4,340	3,823
691,808,220	118,447,199	5.8	7,068	6,682
790,518,060	141,902,681	5.6	885	885

Cilkprof Overhead

Benchmark	*Input size*	*Description*	*Overhead*
`mm`	2048×2048 matrix	Square matrix multiplication	0.99
`dedup`	large	Compression program	1.03
`lu`	2048×2048 matrix	LU matrix decomposition	1.04
`strassen`	2048×2048 matrix	Strassen matrix multiplication	1.06
`heat`	$4096 \times 1024 \times 40$ spacetime	Heat diffusion stencil	1.07
`cilksort`	$10,000,000$ elements	Parallel mergesort	1.08
`pbfs`	$\|V\| = 8$M, $\|E\| = 55.8$M	Parallel breadth-first search	1.10
`fft`	$8,388,608$	Fast Fourier transform	1.15
`quicksort`	$100,000,000$ elements	Parallel quicksort	1.20
`nqueens`	12×12 board	n-Queens problem	1.27
`ferret`	large	Image similarity search	2.04
`leiserchess`	5.8M nodes	Speculative game-tree search	3.72
`collision`	$528,032$ faces	Collision detection in 3*D*	4.37
`cholesky`	2000×2000 matrix, 16000 nonzeros	Cholesky decomposition	4.54
`hevc`	5 frames	H265 video encoding and decoding	6.25
`fib`	35	Recursive Fibonacci	7.36