OMPT and OMPD: Emerging Tool Interfaces for OpenMP

John Mellor-Crummey
Department of Computer Science
Rice University

Acknowledgments

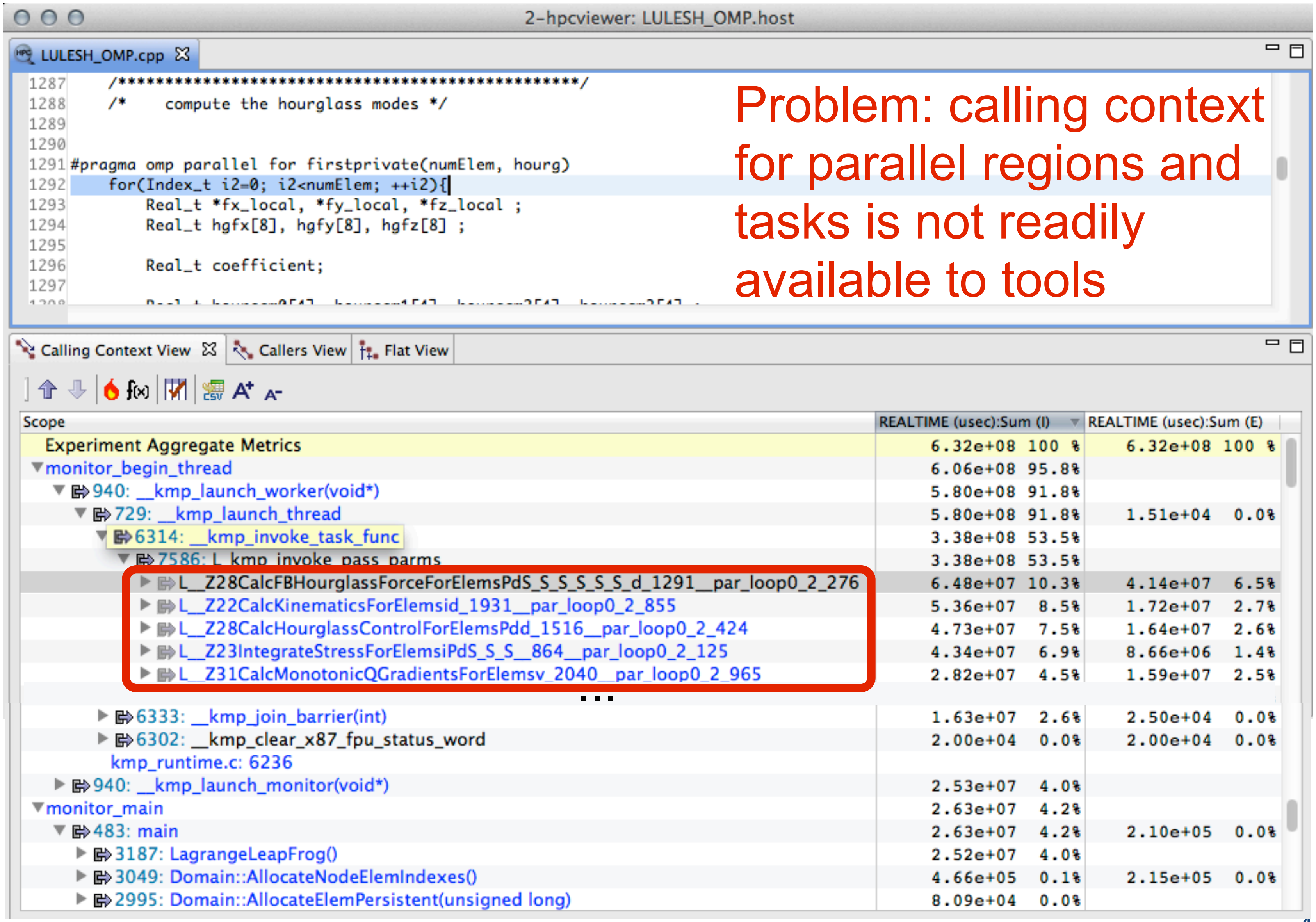
OpenMP tools subcommittee

- Executive lead
  - Martin Schulz - LLNL
- Technical leads
  - Alexandre Eichenberger - IBM
  - John Mellor-Crummey - Rice
- Active subcommittee members
  - Nawal Copty - Oracle
  - James Cownie - Intel
  - John DelSignore - Rogue Wave
  - Robert Dietrich - TU Dresden
  - Xu Liu - Rice
  - Eugene Loh - Oracle
  - Daniel Lorenz - Juelich

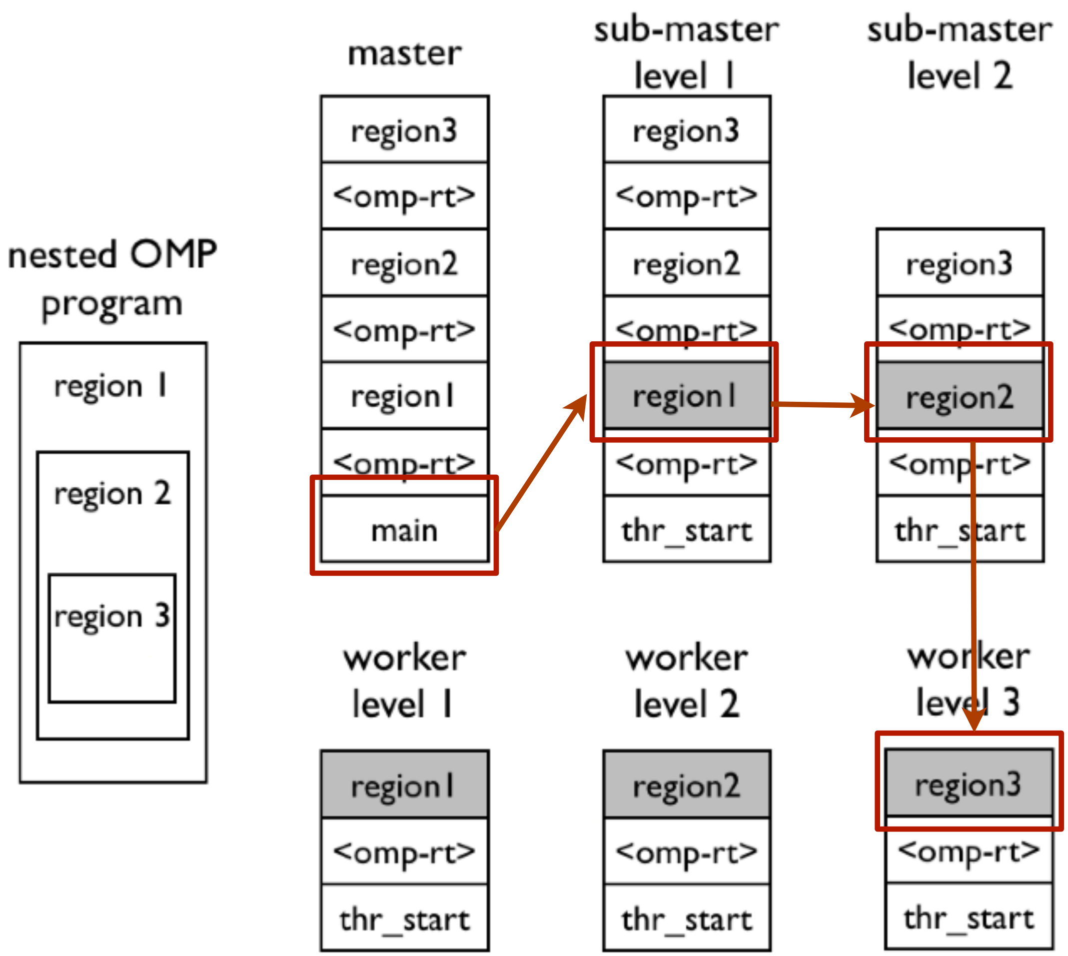
Motivation

- Highly-threaded multicore and manycore processors
  - Blue Gene/Q - 16 compute cores x 4-way SMT
  - Intel Xeon Phi - 60 compute cores x 4-way SMT
- OpenMP: important HPC threaded programming model for nodes
  - MPI + OpenMP increasingly common
- Large gap between source and implementation
  - tools must bridge this gap

Gap Between Source and Implementation



Calling Context Distributed Across OpenMP Threads



regions in gray have distributed calling contexts

Obstacles for Runtime-independent Tools

- No standard API for OpenMP tools
- Principal prior efforts
  - POMP - Mohr, Malony, Shende, Wolf
  - collector API - Itzkowitz, Mazurov, Copty, Lin
- Differences in OpenMP implementations
  - shepherd thread
  - cactus stack
  - ...
- Lack of standard hooks

Outline

- OMPT - emerging performance tool API for OpenMP
  - overview and goals
  - state tracking
  - event notification
  - API
- OMPD - emerging debugger interface for OpenMP
  - motivation
  - state inspection
  - control
- Status and next steps

OMPT Performance Tools API

Overview and Goals

- Create a standardized performance tool interface for OpenMP
  - prerequisite for portable performance tools
  - goal: inclusion in the OpenMP standard
  - role model: PMPI and MPI_T
- Focus on minimal set of functionality
  - provide essential support for sampling-based tools
  - only require support for tools attached at link-time or program launch
- Minimize runtime cost
  - reduce cost in runtime and tool where possible
  - enable integration into optimized runtimes
  - make support for higher-overhead features optional
    - callbacks for blame shifting
    - callbacks for full-featured tracing tools

Major OMPT Functionality

- State tracking
  - have runtime track keep track of its own state
  - allow tools to query this state at any time (async signal safe)
  - provide (limited) persistent storage for tool data in runtime system
- Call stack interpretation
  - provide hooks to enable recovery of complete calling context for computations in worker threads
    - hooks to support reconstruction of application-level call stacks
  - support identification of OpenMP runtime stack frames
- Event notification
  - provide callback mechanism for predefined events
  - support a few mandatory notifications and many optional ones

Runtime State Tracking

- OpenMP runtime keeps track of its own state
  - predefined states on next slide
- Query routine
  - **ompt_state_t ompt_get_state(ompt_wait_id_t *wait_id)**
  - routine must be async signal safe
- Wait IDs
  - only available for states that signify waiting
  - identifies the cause for waiting
    - e.g., address of a user lock or implicit lock for a critical region/atomic

Predefined States

```
/* work states (0..15) */
ompt_state_work_serial           = 0x00, /* serial work */
ompt_state_work_parallel         = 0x01, /* parallel work */
ompt_state_work_reduction        = 0x02, /* performing a reduction */
/* idle (16..31) */
ompt_state_idle                  = 0x10, /* waiting for work */
/* overhead states (32..63) */
ompt_state_overhead              = 0x20, /* non-wait overhead */
/* barrier wait states (64..79) */
ompt_state_wait_barrier          = 0x40, /* waiting at any barrier */
ompt_state_wait_explicit_barrier = 0x41, /* waiting at an explicit barrier */
/* task wait states (80..95) */
ompt_state_wait_taskwait         = 0x50, /* waiting at a taskwait */
ompt_state_wait_taskgroup        = 0x51, /* waiting at a taskgroup */
/* wait states mutex (96..111) */
ompt_state_wait_lock             = 0x60, /* waiting for lock */
ompt_state_wait_nest_lock        = 0x61, /* waiting for nest lock */
ompt_state_wait_critical         = 0x62, /* waiting for critical */
ompt_state_wait_atomic           = 0x63, /* waiting for atomic */
ompt_state_wait_ordered          = 0x64, /* waiting for ordered */
/* miscellaneous (112..127)*/
ompt_state_undefined             = 0x70, /* undefined thread state */
ompt_state_first                 = 0x71, /* initial enumeration state */
```

OMPT Event Notifications

- Mandatory events
- Blame-shifting events (optional)
- Trace events (optional)

Mandatory Events

Essential support for any performance tool

- Threads
- Parallel regions
- Tasks

create/exit event pairs

- Runtime shutdown
- User-level control API
  - e.g., support tool start/stop

Blame-shifting Events (Optional)

Support designed for sampling-based performance tools

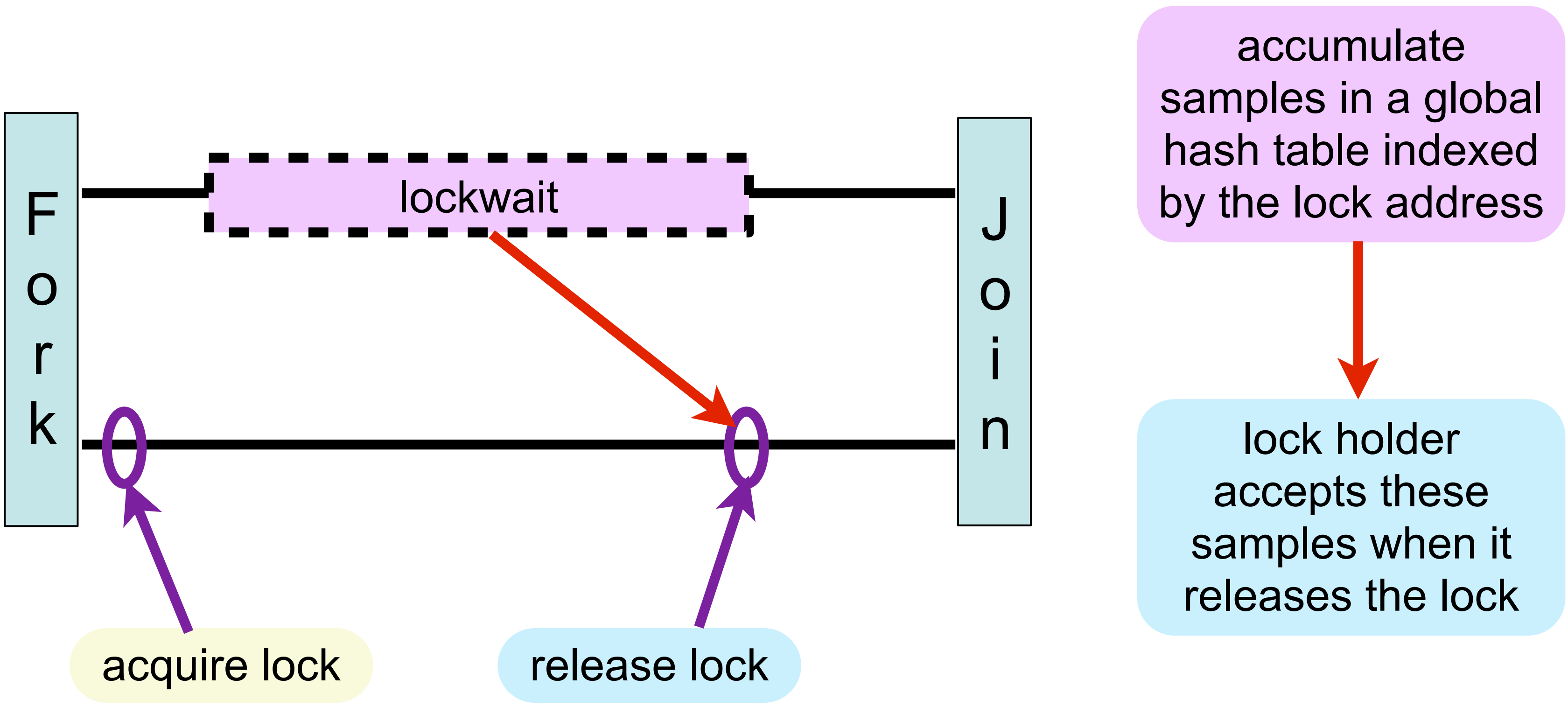
- Idle
- Wait
  - barrier
  - taskwait
  - taskgroup wait

begin/end event pairs

- Release
  - lock
  - nest lock
  - critical
  - atomic
  - ordered section

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

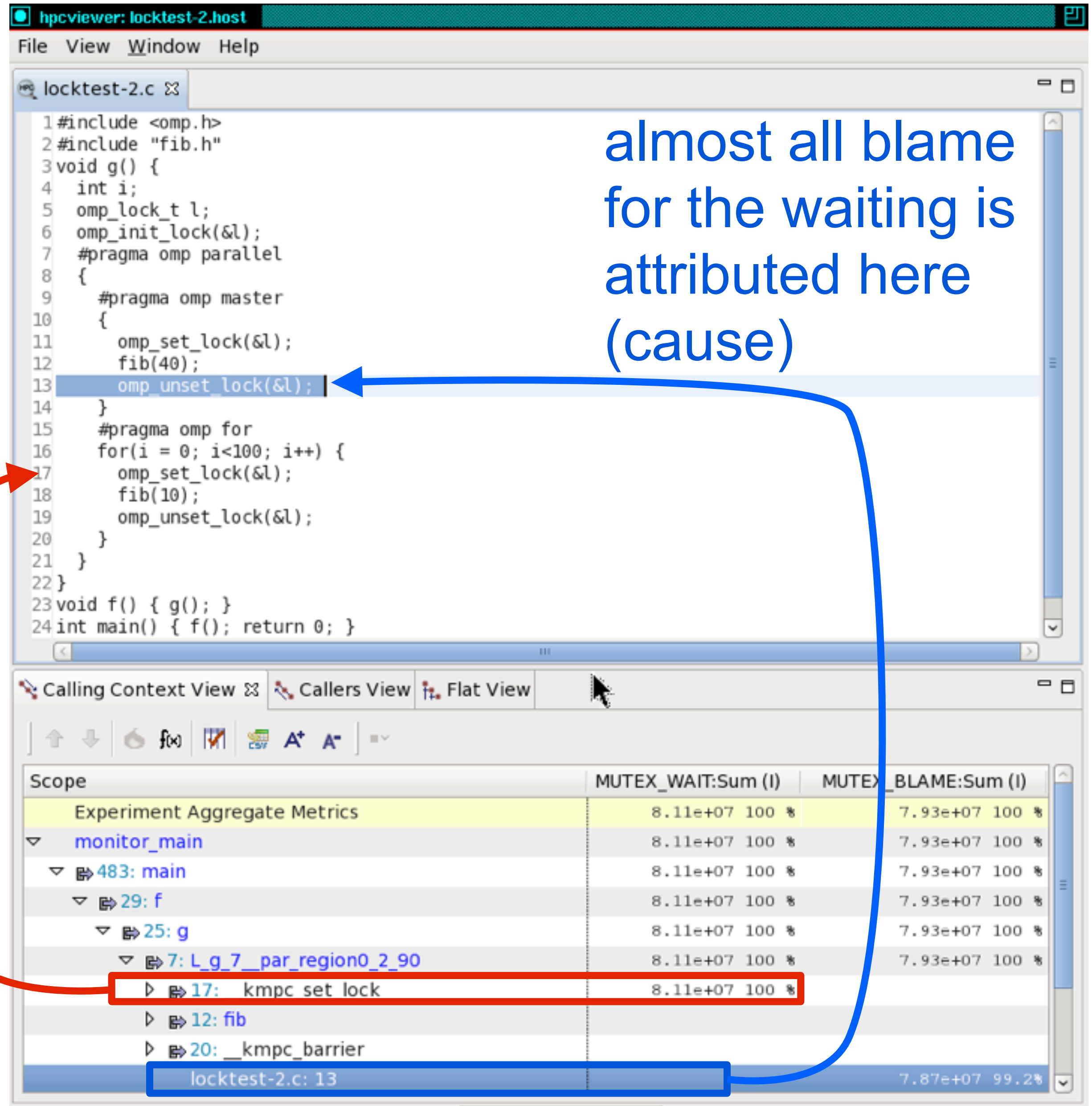


Example: Directed Blame Shifting for Locks

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)



Trace Events (Optional)

ompt_event_implicit_task_create	ompt_event_taskgroup_end
ompt_event_implicit_task_exit	ompt_event_release_nest_lock_prev
ompt_event_task_switch	ompt_event_wait_lock
ompt_event_loop_begin	ompt_event_wait_nest_lock
ompt_event_loop_end	ompt_event_wait_critical
ompt_event_section_begin	ompt_event_wait_atomic
ompt_event_section_end	ompt_event_wait_ordered
ompt_event_single_in_block_begin	ompt_event_acquired_lock
ompt_event_single_in_block_end	ompt_event_acquired_nest_lock_first
ompt_event_single_others_begin	ompt_event_acquired_nest_lock_next
ompt_event_single_others_end	ompt_event_acquired_critical
ompt_event_master_begin	ompt_event_acquired_atomic
ompt_event_master_end	ompt_event_acquired_ordered
ompt_event_barrier_begin	ompt_event_init_lock
ompt_event_barrier_end	ompt_event_init_nest_lock
ompt_event_taskwait_begin	ompt_event_destroy_lock
ompt_event_taskwait_end	ompt_event_destroy_nest_lock
ompt_event_taskgroup_begin	ompt_event_flush

Thread State/Data & Query Functions

- Runtime maintains some state for a tool
  - persists between entry/exit events
  - lifetime equals that of associated thread or region
  - support for a single tool / single data item
- Data structure

```
typedef union ompt_data_t {
    long long value;
    void *ptr;
} ompt_data_t;
```

  - suitable for holding a pointer or an integer
- Query thread data
  - routine: **ompt_data_t *ompt_get_thread_data()**
  - async signal safe

Parallel Region IDs

- Each parallel region instance has a unique ID
  - region IDs are not required to be consecutive
- Ability to query parallel region IDs
  - **ompt_parallel_id_t ompt_get_parallel_id(int ancestor_level)**
  - async signal safe
  - current region: ancestor_level = 0
  - query IDs of ancestor regions using higher ancestor levels
- Query function pointer of current and parent functions
  - **void *ompt_get_parallel_function(int ancestor_level)**
  - async signal safe

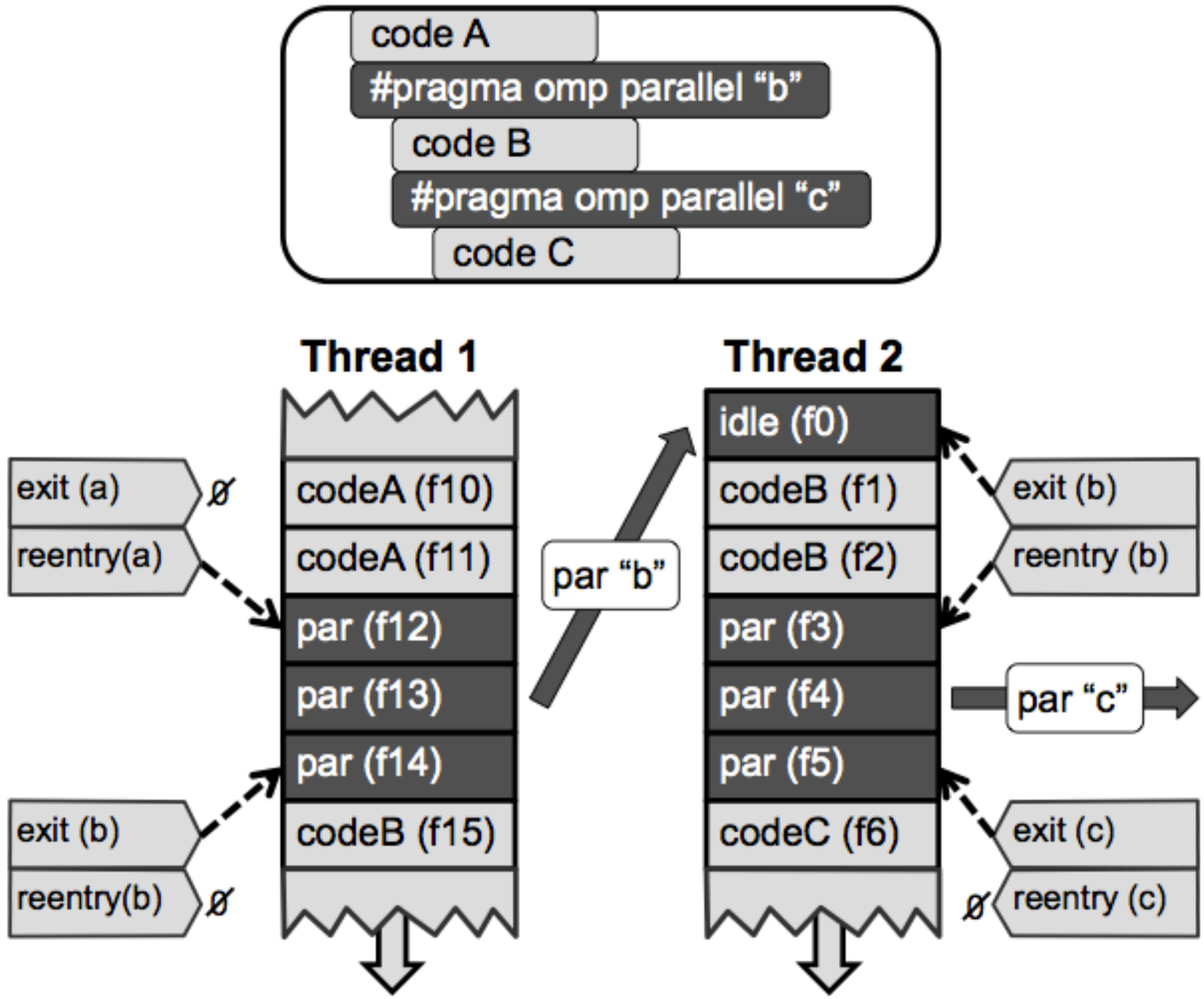
Call Stack Interpretation

- Tool saves some frame information to support stack unwinding

```
typedef struct ompt_frame_t {
   void *reenter_runtime_frame;
   void *exit_runtime_frame;
} ompt_frame_t;
```

  - per task; lifetime: duration of task
  - **ompt_frame_t *ompt_get_task_frame(int ancestor_level)**
  - async signal safe
- Reenter_runtime_frame
  - set each time a current task enters the runtime to create a new task
  - points to the stack above the return address of the last user frame
- Exit_runtime_frame
  - set when a task exits the runtime to execute user code
  - points to the stack above the return address of the last runtime frame

Call Stack Interpretation Example



Task Inquiry Functions

Inquiry functions async signal safe

- Query task function
  - **void *ompt_get_task_function(int ancestor_level)**
- Query task data
  - **ompt_data_t *ompt_get_task_data(int ancestor_level)**

Miscellaneous API Features

- Tool-facing API functions
  - initialization
    - **int ompt_initialize(void)**
    - **int ompt_set_callback(ompt_event_t e, ompt_callback_t cb)**
  - tool support version inquiry
    - **int ompt_get_ompt_version(void)**
  - state enumeration
    - **int ompt_enumerate_state(int current_state, int *next_state,**
      const char **next_state_name)
- User-facing API functions
  - version inquiry
    - **int ompt_get_runtime_version(char *buffer, int length)**
  - tool control
    - **void ompt_control(uint64_t command, uint64_t modifier)**
- OMPD debugger support shared-library locations
  - char **ompd_dll_locations
    - argv-style list of filename strings

Outline

- OMPT - emerging performance tool API for OpenMP
  - overview and goals
  - state tracking
  - event notification
  - API
- OMPD - emerging debugger interface for OpenMP
  - motivation
  - state inspection
  - control
- Status and next steps

OMPD Debugger Support Library

- A standard plug-in library to be dynamically-loaded by debuggers
  - enable a debugger to interact with any OpenMP runtime
- Strategy used for pthreads and MPI
- Historical precedent for OpenMP
  - J. Cownie, J. DelSignore, B. R. de Supinski, and K. Warren. DMPL: an OpenMP DLL debugging interface. In *Proceedings of the OpenMP applications and tools 2003 international conference on OpenMP shared memory parallel programming*, WOMPAT'03, pages 137–146, Berlin, Heidelberg, 2003. Springer-Verlag.

Unimplemented Design

OMPD Design Objectives

- Enable a debugger to inspect state of live process or core file
  - provide debugger with third-party versions of OpenMP runtime functions
  - provide debugger with third-party versions of OMPT inquiry functions
- Facilitate interactive control of a live process
  - help debugger place breakpoints
    - intercept enter/exit of parallel regions
    - intercept first instruction in a parallel region or task region
- API should not impose an unreasonable development burden
  - runtime implementers
  - tool implementers

OMPD Initialization

- ompd_rc_t ompd_initialize(ompd_callbacks_t *cb)
  - debugger informs ompd library about debugger entry points

```
typedef struct {
  /*------------------------------------------------------------------------------*/
  /* debugger interface
  /*------------------------------------------------------------------------------*/

  /* interface for ompd to allocate/free memory in the debugger's address space */
  ompd_dmemory_alloc_fn_t  d_alloc_memory;      /* allocate memory in the debugger        */
  ompd_dmemory_free_fn_t   d_free_memory;       /* free memory in the debugger            */

  /* errors */
  ompd_error_string_fn_t get_error_string;  /* retrieve an error string for an error code  */

  /* printing */
  ompd_print_string_fn_t print_string;    /* have the debugger print a string for OMPD     */

  /*------------------------------------------------------------------------------*/
  /* target interface
  /*------------------------------------------------------------------------------*/

  /* obtain information about the size of primitive types in the target   */
  ompd_tsizeof_prim_fn_t   t_sizeof_prim_type;    /* return the size of a primitive type   */

  /* obtain information about symbols and structure offsets in the target  */
  ompd_tsymbol_addr_fn_t   t_symbol_addr_lookup;  /* look up the address of a symbol       */

  ompd_ttype_fn_t   t_type_lookup;                /* look up a type in the target          */
  ompd_ttype_sizeof_fn_t   t_type_sizeof;         /* look up the size of of a type         */
  ompd_ttype_offset_fn_t   t_type_field_offset;   /* look up a field offset in a type      */

  /* access data in the target   */
  ompd_tmemory_access_fn_t t_read_memory;       /* read from target address into buffer    */
  ompd_tmemory_access_fn_t t_write_memory;      /* write from buffer to target address     */

  /* convert byte ordering */
  ompd_target_host_fn_t target_to_host;

} ompd_callbacks_t;
```

OMPD Handle Management

- Each OMPD call that is dependent on a context must provide that context as a handle
- Handle types
  - target process
  - threads
  - parallel regions
  - tasks

OMPD Handle Inquiry Operations

- Threads
  - retrieve array of handles for all OpenMP threads
  - retrieve array of handles for OpenMP threads in a parallel region
- Parallel regions
  - retrieve handle for innermost parallel region for an OpenMP thread
  - retrieve handle for enclosing parallel region
- Tasks
  - retrieve handle for innermost task for an OpenMP thread
  - retrieve handle for enclosing task
  - retrieve implicit task handle for parallel region

OMPD Setting Inquiry Operations

- Process
  - OMP info
    - thread limit
    - number of procs
- Parallel regions
  - OMP info
    - number of threads
    - depth of a parallel region instance
    - number of enclosing active parallel regions
  - OMPT info
    - parallel id
    - parallel function
- OS thread inquiry
  - thread handle ↔ OS thread
  - OMPT info
    - thread state

OMPD Task Inquiry Operations

- OMP API analogues
  - get max threads
  - get thread num
  - in parallel
  - in final
  - get dynamic
  - get nested
  - get max active levels
  - get schedule
  - get proc bind
- OMPT analogues
  - get task frame
  - get task function

Note: no OMP API counterparts in OMPT interface because OMPT can call OMP runtime functions directly

OMPD Breakpoint Interface

- Neither a debugger nor OpenMP runtime knows what application code a program will launch in a parallel region or task until a code address is provided as an argument to an OpenMP runtime call
- Inform debugger where breakpoints can be placed to intercept parallel regions and tasks

```
typedef struct ompd_breakpoints_s {
  ompd_taddr_t parallel_pre_execute;
  ompd_taddr_t parallel_post_execute;
  ompd_taddr_t task_pre_execute;
  ompd_taddr_t task_post_execute;
} ompd_breakpoints_t;

EXTERN ompd_rc_t ompd_get_breakpoints(
  ompd_context_t *context,  /* debugger handle for the target */
  ompd_breakpoints_t *bkpt_locations
);
```

Breakpoints in Parallel Region and Task Code

- Parallel regions
  - debugger gains control with trap at pre_execute
  - debugger maps OS thread to OpenMP thread using OMPD
  - inquires about top parallel region
  - inquires about user function executed by parallel region
- Tasks
  - similar to above

Miscellaneous API Operations

- Function to inquire about control variable settings
- Function to enable/disable performance tool support at next clean point (if possible)

Outline

- OMPT - emerging performance tool API for OpenMP
  - overview and goals
  - state tracking
  - event notification
  - API
- OMPD - emerging debugger interface for OpenMP
  - motivation
  - state inspection
  - control
- Status and next steps

Status Next Steps

- Specifications
  - OMPT
    - apply last bit of polish to API
      - nits with barriers
      - worker idle frame
    - submit it to OpenMP language committee for comment
    - turn it into an official OpenMP TR
  - OMPD
    - will anyone implement it?
- Runtime implementations
  - IBM will release OMPT interface on BG/Q and Power
  - Rice and Oregon will finish draft of OMPT in Intel runtime
- Tools
  - HPCToolkit OpenMP branch will be folded into trunk

Additional Details

Supplemental Material

- A few examples of OMPT implementation issues in Intel Runtime
- HPCToolkit capabilities using OMPT

OMPT Callbacks in Intel OpenMP Runtime

- Add callbacks for blame shifting
  - if action warrants
  - if tracking enabled
  - if callback provided
- Example
  - release nested lock
    - if outer release
    - and tool callbacks enabled
    - and callback provided
    - make the callback and pass a "wait id"

```
/* release the lock */
void
__kmpc_unset_nest_lock( ident_t *loc, kmp_int32 gtid, void **user_lock )
{
    kmp_user_lock_p lck;

    /* Can't use serial interval since not block structured */

    if ( ( __kmp_user_lock_kind == lk_tas ) && ( sizeof( lck->tas.lk.poll )
      + sizeof( lck->tas.lk.depth_locked ) <= OMP_NEST_LOCK_T_SIZE ) ) {
#if KMP_OS_LINUX && (KMP_ARCH_X86 || KMP_ARCH_X86_64)
        // "fast" path implemented to fix customer performance issue
        kmp_tas_lock_t *tl = (kmp_tas_lock_t*)user_lock;
        if ( --(tl->lk.depth_locked) == 0 ) {
            TCW_4(tl->lk.poll, 0);
        }
        KMP_MB();
        return;
#else
        lck = (kmp_user_lock_p)user_lock;
#endif
    }
#if KMP_OS_LINUX && (KMP_ARCH_X86 || KMP_ARCH_X86_64)
    else if ( ( __kmp_user_lock_kind == lk_futex )
     && ( sizeof( lck->futex.lk.poll ) + sizeof( lck->futex.lk.depth_locked )
     <= OMP_NEST_LOCK_T_SIZE ) ) {
        lck = (kmp_user_lock_p)user_lock;
    }
#endif
    else {
        lck = __kmp_lookup_user_lock( user_lock, "omp_unset_nest_lock" );
    }

    int release_status = RELEASE_NESTED_LOCK( lck, gtid );

#if OMPT_SUPPORT
    if ((release_status == KMP_NESTED_LOCK_RELEASED) &&
       (ompt_status == ompt_status_track_callback) &&
       (ompt_callbacks.ompt_callback(ompt_event_release_nest_lock_last))) {
      ompt_callbacks.ompt_callback(ompt_event_release_nest_lock_last)((uint64_t) lck);
    }
#endif
}
```

OMPT Frame Tracking in Intel OpenMP Runtime

- Add frame tracking to enable reconstruction of application-level call stacks
- Support:
  - __kmpc_fork_call
    - record frame address
    - the call in user code is below this point
  - __kmp_invoke_microtask
    - record “exit” SP location above return address for call

```
Do the actual fork and call the microtask in the relevant number of thread:
*/
void
__kmpc_fork_call(ident_t *loc, kmp_int32 argc, kmpc_micro microtask, ...)
{
  int          gtid = __kmp_entry_gtid();
  // maybe to save thr_state is enough here
  {
    va_list     ap;
    va_start(   ap, microtask );

#if OMPT_SUPPORT
    kmp_info_t *master_th = __kmp_threads[ gtid ];
    kmp_team_t *parent_team = master_th->th.th_team;
    int tid = __kmp_tid_from_gtid( gtid );
    parent_team->t.t_implicit_task_taskdata[tid].
      ompt_task_info.frame.reenter_runtime_frame =
        __builtin_frame_address(0);
#endif

    __kmp_fork_call( loc, gtid, TRUE,
            argc,
            VOLATILE_CAST(microtask_t) microtask,
            VOLATILE_CAST(launch_t)    __kmp_invoke_task_func,
```

```
// int
// __kmp_invoke_microtask( void (*pkfn) (int *gtid, int *tid, ...),
//                         int gtid, int tid,
//                         int argc, void *p_argv[] ) {
//    (*pkfn)( & gtid, & tid, argv[0], ... );
//    return 1;
// }
...
// begin OMPT SUPPORT
        leaq    -8(%rsp),%r11   // Address after the return address has been pushed (r11
        movq    %r11, (%r9)     // save exit_frame
// end OMPT SUPPORT
...
        call    *%rbx           // call (*pkfn)();
        movq    $1, %rax        // move 1 into return register;
```

State Tracking, Callbacks, Frames, & More

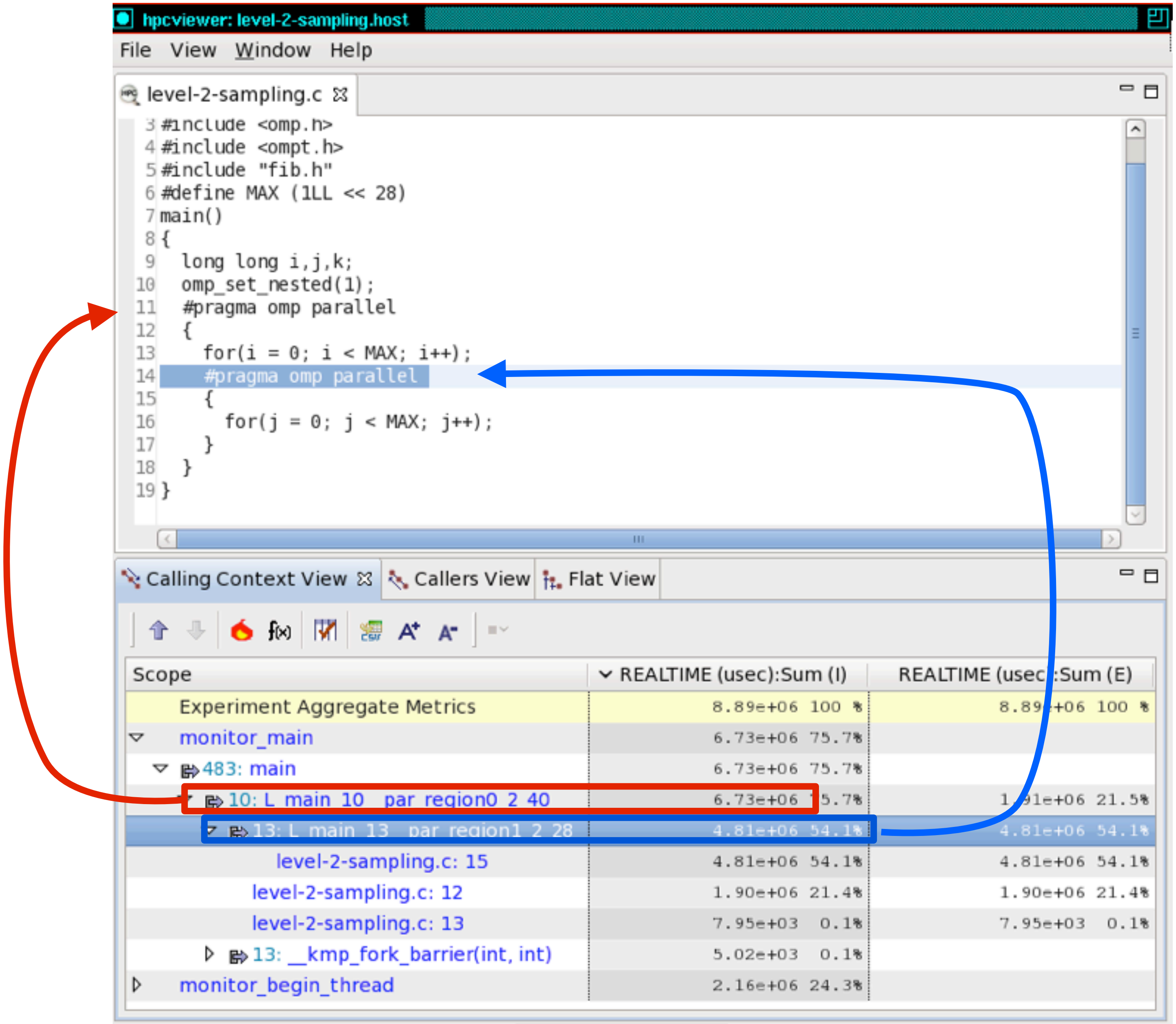
- __kmp_fork_call
- Shown: handling for degenerate case with singleton team
  - need a lightweight team record on the stack to maintain OMPT info
  - state changes from overhead to “parallel work” when invoking microtask
  - returns to overhead afterward
  - create/exit callbacks for parallel region
  - after microtask, clear exit_frame

```
#if OMPT_SUPPORT
        ompt_lw_taskteam_t lw_taskteam;
        void **exit_runtime_p =
          &(lw_taskteam.ompt_task_info.frame.exit_runtime_frame);
        if (ompt_status & ompt_status_track) {
          __ompt_lw_taskteam_init(&lw_taskteam, master_th, gtid, microtask);

          /* OMPT state */
          master_th->th.ompt_thread_info.state = ompt_state_work_parallel;

          /* OMPT event */
          if ((ompt_status & ompt_status_track_callback) &&
              ompt_callbacks.ompt_callback(ompt_event_parallel_create)) {
            ompt_callbacks.ompt_callback(ompt_event_parallel_create)
              (&lw_taskteam.ompt_task_info.data,
               &lw_taskteam.ompt_task_info.frame,
               lw_taskteam.ompt_team_info.parallel_id);
          }
        }
#else
        void *dummy;
        void **exit_runtime_p = &dummy;
#endif

        __kmp_invoke_microtask( microtask, gtid, 0, argc, args, exit_runtime_p )

#if OMPT_SUPPORT
        if (ompt_status & ompt_status_track) {
          lw_taskteam.ompt_task_info.frame.exit_runtime_frame = 0;
          if ((ompt_status & ompt_status_track_callback) &&
              ompt_callbacks.ompt_callback(ompt_event_parallel_exit)) {
            ompt_callbacks.ompt_callback(ompt_event_parallel_exit)
              (&lw_taskteam.ompt_task_info.data,
               &lw_taskteam.ompt_task_info.frame,
               lw_taskteam.ompt_team_info.parallel_id);
          }
          __ompt_lw_taskteam_fini(&lw_taskteam, master_th);
          master_th->th.ompt_thread_info.state = prev_state;
        }
#endif
```

Supplemental Material

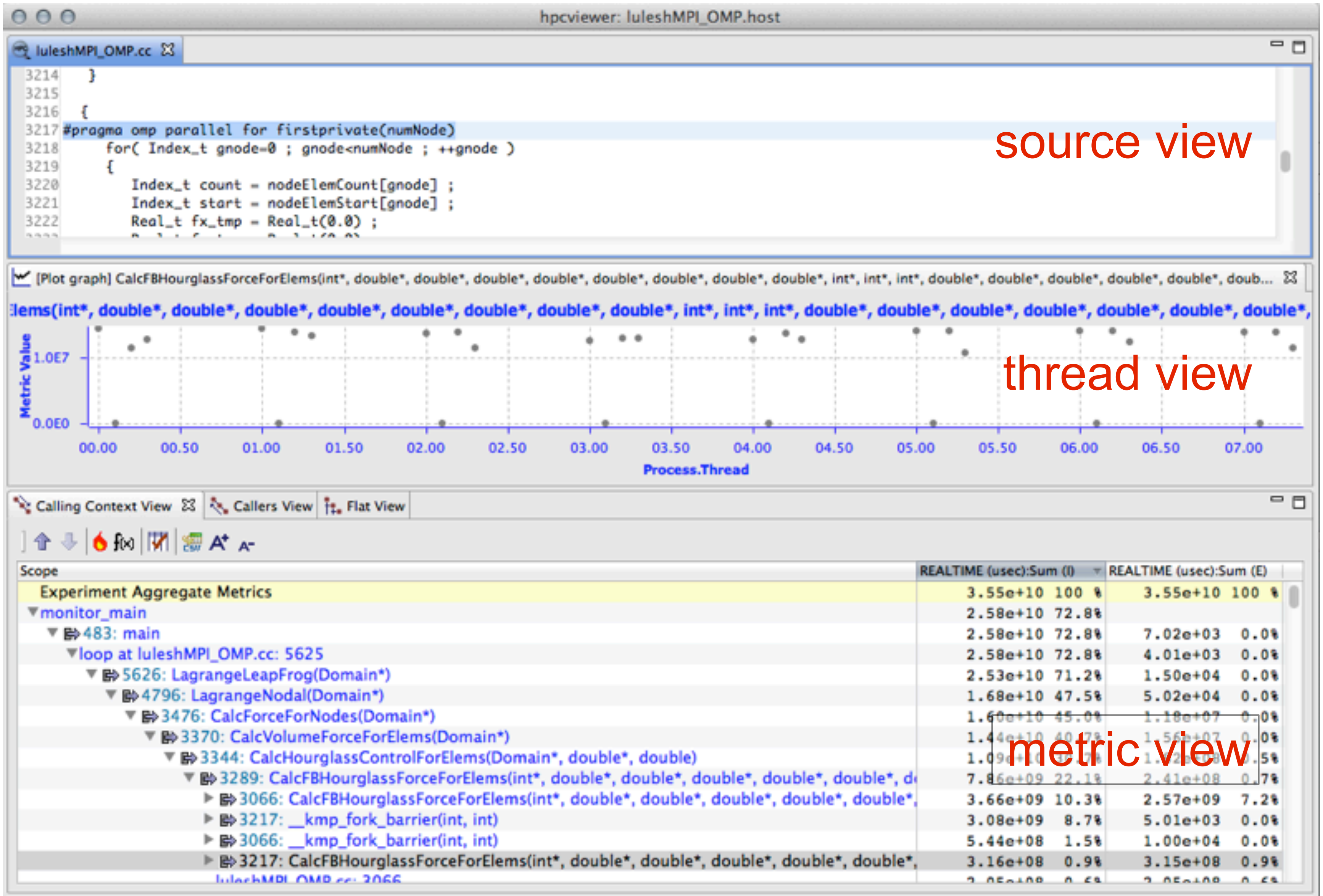
- A few examples of OMPT implementation issues in Intel Runtime
- HPCToolkit capabilities using OMPT

Assembly of Nested Regions with HPCToolkit



Integrated View of MPI+OpenMP with OMPT

LLNL's luleshMPI_OMP (8 MPI x 3 OMP), 30, REALTIME@1000



Integrated View of MPI+OpenMP with OMPT

LLNL's luleshMPI_OMP (8 MPI x 3 OMP), 30, REALTIME@1000

time-centric view