Tasks in the OpenMP API
(A behind-the-scenes glimpse)

Dr.-Ing. Michael Klemm

Chief Executive Officer
OpenMP Architecture Review Board
michael.klemm@openmp.org

Principal Member of Technical Staff
HPC Center of Excellence at AMD
michael.klemm@amd.com

Task Execution Model

- Suited for unstructured parallelism
  - unbounded loops

```
while ( <expr> ) {
    ...
}
```

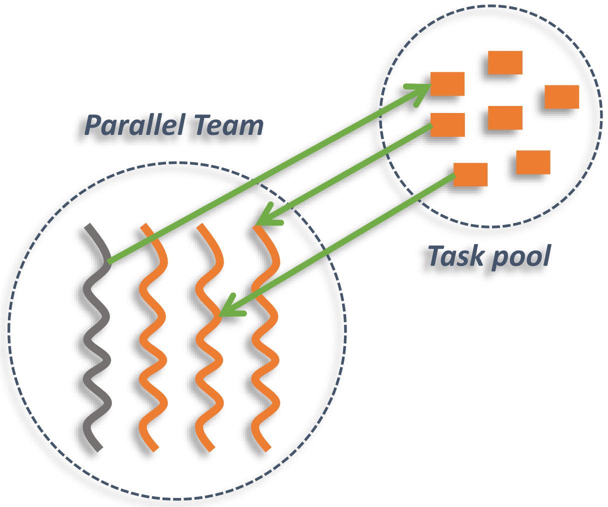
  - recursive functions

```
void myfunc( <args> )
{
    ...; myfunc( <newargs> ); ...;
}
```

- Several scenarios are possible:
  - single creator, multiple creators, nested tasks (tasks & worksharing)
- All threads in the team are candidates to execute tasks

- Example: traversal of a linked list

```
#pragma omp parallel
#pragma omp master
while (elem != NULL) {
   #pragma omp task
      compute(elem);
   elem = elem->next;
}
```

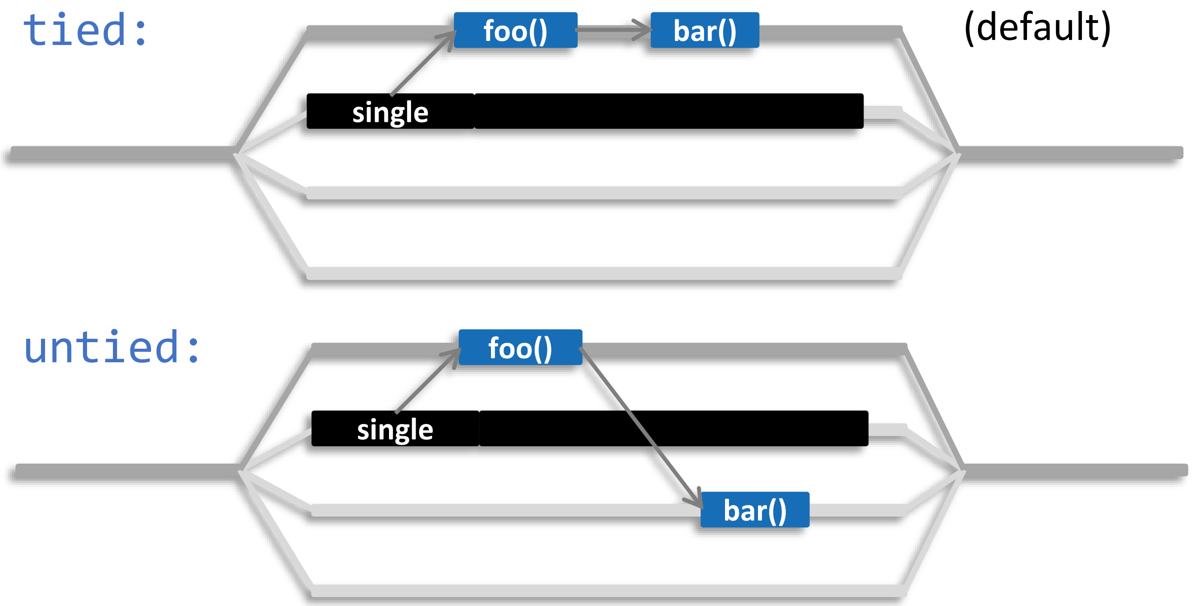


Task Scheduling (taskyield Directive)

- Task scheduling points (and the taskyield directive)
  - tasks can be suspended/resumed at Task Scheduling Points (some additional constraints to avoid deadlocks)
  - implicit scheduling points (creation, synchronization, ...)
  - explicit scheduling point: `#pragma omp taskyield`

- Example:

```
#pragma omp parallel
#pragma omp single
{
   #pragma omp task untied
   {
      foo();
      #pragma omp taskyield
      bar();
   }
}
```



Outlining Tasks (here: clang/LLVM)

- clang/LLVM splits task creation
  - allocate task descriptor and data area for the new task
  - submit task to runtime system for execution

```
void create_task(int i, double d) {
    #pragma omp task firstprivate(i) \
                     firstprivate(d)
    {
      double answer = i * d;
      printf("The answer is %lf\n", answer);
    }
}

void caller() {
    create_task(2, 21.0);
}
```

```
void create_task(int i, double d) {
   void * task =
      __kmpc_omp_task_alloc(NULL, 0, NULL,
                            40 + 16, 16,
                            .omp_thunk_0);
    char * data = ((char **)task)[0];
    memcpy(data + 0, &i, sizeof(int));
    memcpy(data + 8, &d, sizeof(double));
    __kmpc_omp_task(NULL, 0, task);
}
```

Task descriptor

Task data

```
int32_t .omp_thunk_0(int32_t, void * task) {
    char * data = ((char **)task)[0];
    int i;
    double d;
    memcpy(&i, data + 0, sizeof(int));
    memcpy(&d, data + 8, sizeof(double));
    double answer = i * d;
    printf("The answer is %lf\n", answer);
    return 0;
}
```

Task Descriptor

- Tasks to be stored in a pool need to carry meta data
  - Code pointer, data pointer

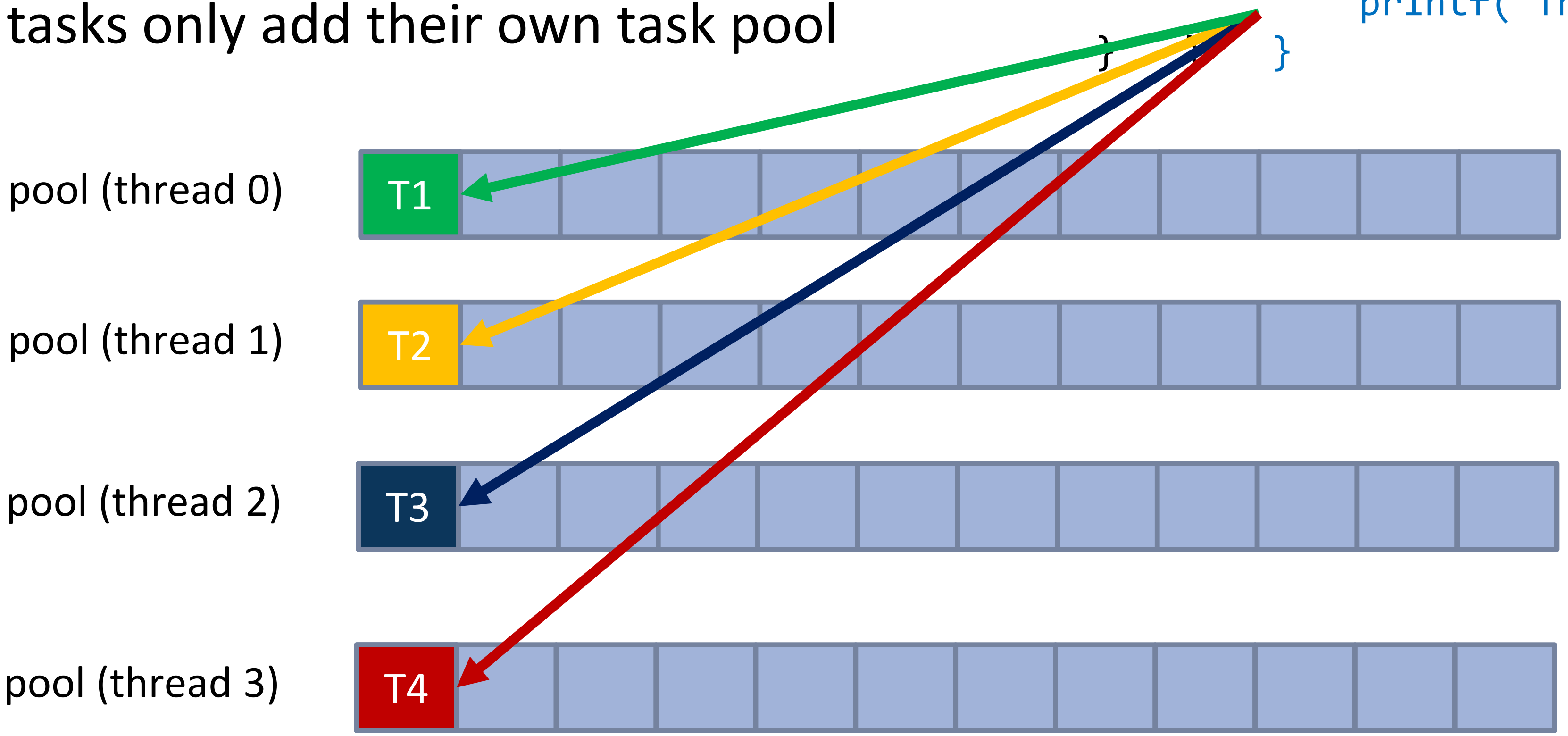
```
struct task_desc_t {
    void (*thunk)();
    void* dataenv;
    size_t env_sz;
    int flags;
    int priority;
    task_desc_t* parent;
    size_t wait_counter;
    taskgroup_t* taskgroup;
    task_depend_t* dephash;
};
```

- Task descriptors usually also store other useful bits that are hard to determine otherwise but are easy to save at creation time of a task.
- Examples:
  - Flags (e.g., tied/untied, status)
  - Scheduling priority
  - Pointer to parent task
  - Wait counter
  - Pointer to the taskgroup
  - Dependences to other tasks

Multiple Task Pools

- Single task pool creates contention
- Multiple, concurrent task pools
- LLVM OpenMP runtime: one per thread
  - Parent tasks only add their own task pool

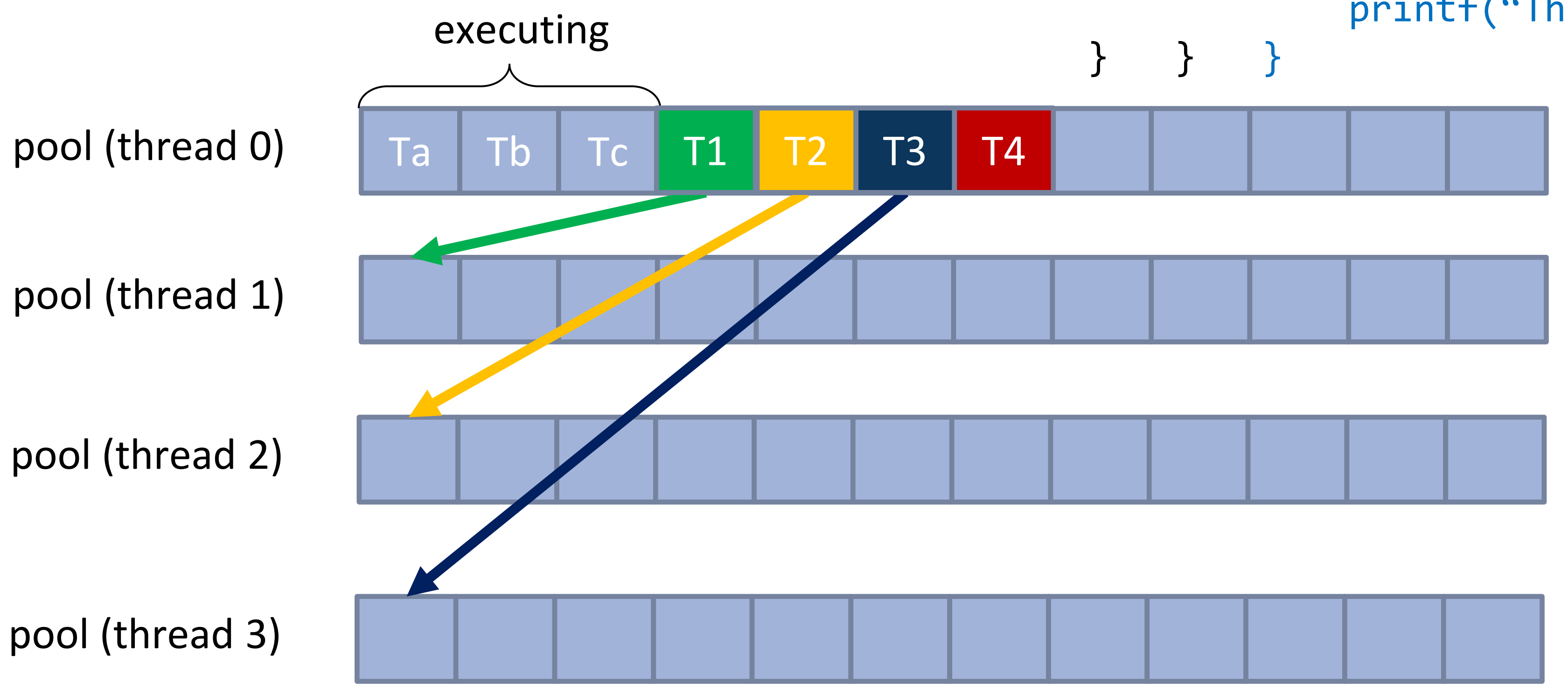
```
void example() {
#pragma omp parallel num_threads(4)
    {
        int tid = omp_get_thread_num();
#pragma omp task // T1-T4
        {
            printf(“Thread %d\n”, tid);
        }
    }
}
```



Multiple Task Pools

- Multiple, concurrent task pools
- LLVM OpenMP runtime: one per thread
- Threads can run out of work.
  Solution: load distribution.

```
void example() {
#pragma omp parallel num_threads(4)
#pragma omp masked filter(0)
    {
        for (i=1; i<=4; i++)
#pragma omp task // T1-T4
        {
            printf("Thread %d\n", tid);
}   }   }
```



Load Distribution between Task Pools

- Load Balancing:
  - Task Sharing: generating thread pushes work from its pool into other pools.
  - Task Stealing: idle threads steal work from another thread’s task pool.

- Tasks to be stolen:
  - Child Stealing: The current task keeps executing and the child is sent to the pool.
  - Continuation Stealing: The current thread executes the child task and the remainder of the parent task is added to the pool.

- Tasks to be stolen, #2:
  - Steal from the tail of the pool.
  - Steal from the head of the pool.

Multiple Task Pools: FIFO Queues

- We can re-use the FIFO queues for the multi-task pool approach:
  - Each thread maintains its local FIFO queue.
  - Tasks added to the pool are added at the tail of the queue.
  - Tasks to be executed are taken from the head of the queue.
- FIFO queues are not the best data structure for load distribution:
  - Owner of the queue and thieves have a higher conflict potential for the head of the queue.
  - Heuristics considering locality indicate:
    - Youngest tasks are less likely to generate many new tasks (e.g., leaf tasks).
    - It is better to steal oldest tasks in the queue, as they are expected to generate more tasks.
    - Thus, thieves steal from the front of the “queue”, owner of the queue add/removes from rear.
- Use double-ended queue (deque) instead of FIFO queues.

Implementation: taskwait (Pseudo-code)

- Task descriptor typically contains a parent pointer!

```
struct task_desc_t {
    void (*thunk)();
    void* dataenv;
    size_t env_sz;
    int flags;
    int priority;
    task_desc_t* parent;
    size_t wait_counter;
    taskgroup_t* taskgroup;
    task_depend_t* dephash;
};
```

```
void __omp_task_fini(task_desc_t* task) {
    // …
    if (task->parent) {
        fetch-and-dec(parent->wait_counter);
    }
    // …
}

void __omp_taskwait(task_desc_t* task) {
    // …
    while(atomic_ld(task->wait_counter) > 0);
    // …
}
```

Spin waiting is suboptimal, as one thread is lost! Better: execute tasks!

Implementation: taskwait (Pseudo-code)

- Task descriptor typically contains a parent pointer!

```
struct task_desc_t {
    void (*thunk)();
    void* dataenv;
    size_t env_sz;
    int flags;
    int priority;
    task_desc_t* parent;
    size_t wait_counter;
    taskgroup_t* taskgroup;
    task_depend_t* dephash;
};
```

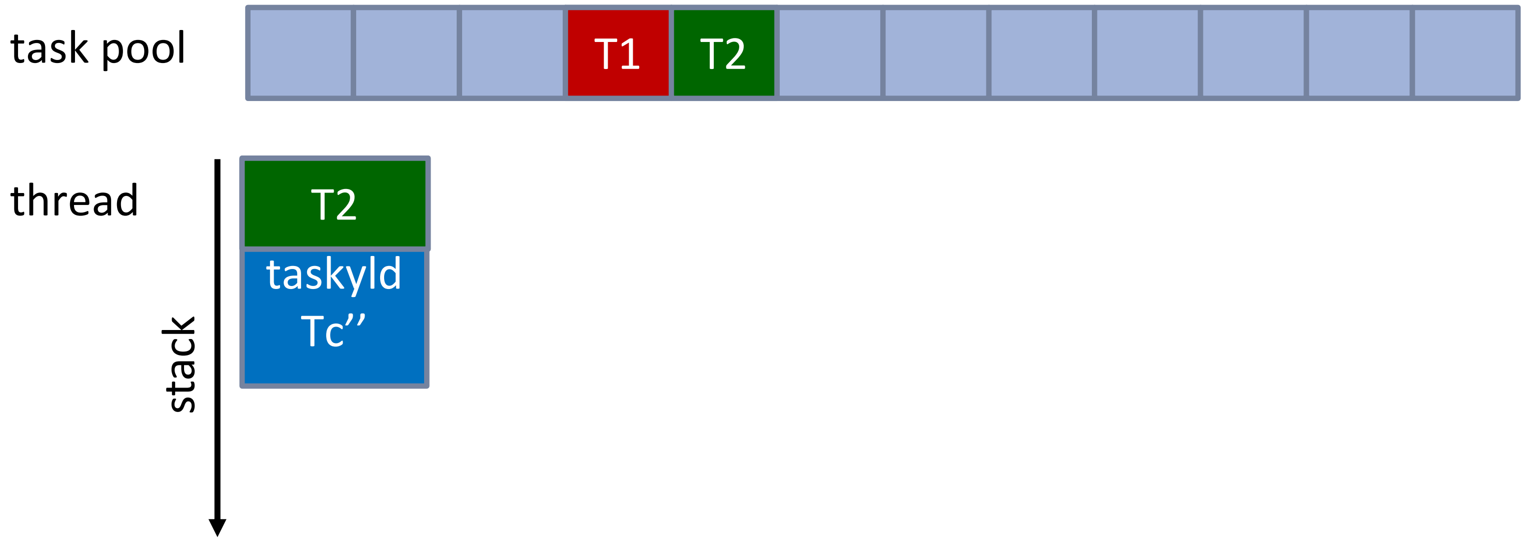
```
void __omp_task_fini(task_desc_t* task) {
    // …
    if (task->parent) {
        fetch-and-dec(parent->wait_counter);
    }
    // …
}

void __omp_taskwait(task_desc_t* task) {
    // …
    while(atomic_ld(task->wait_counter) > 0) {
        taskqueue_t* queue = thread->get_queue();
        task_desc_t* invoke = queue->pop_task();
        __omp_exec_task(invoke);
    }
    // …
}
```

Get next task from queue and execute it.

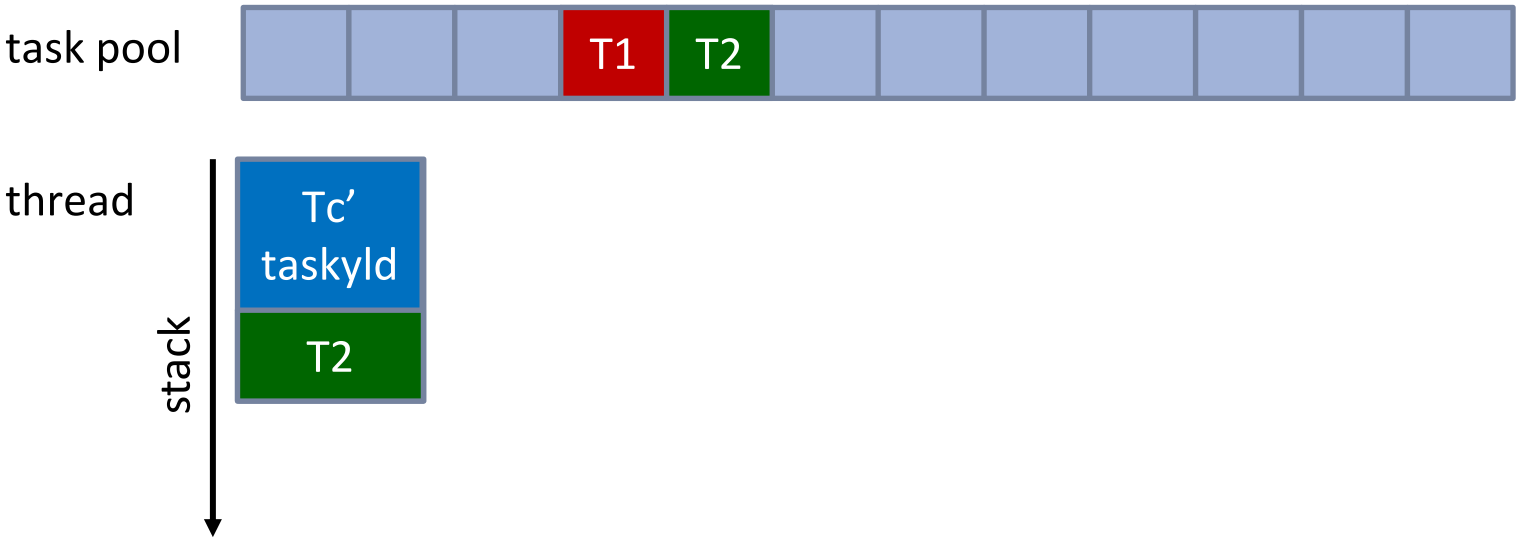
`taskyield` Implementation: No-op

- Simplification: assume only one task pool.
- Task “Tc” contains a taskyield directive.



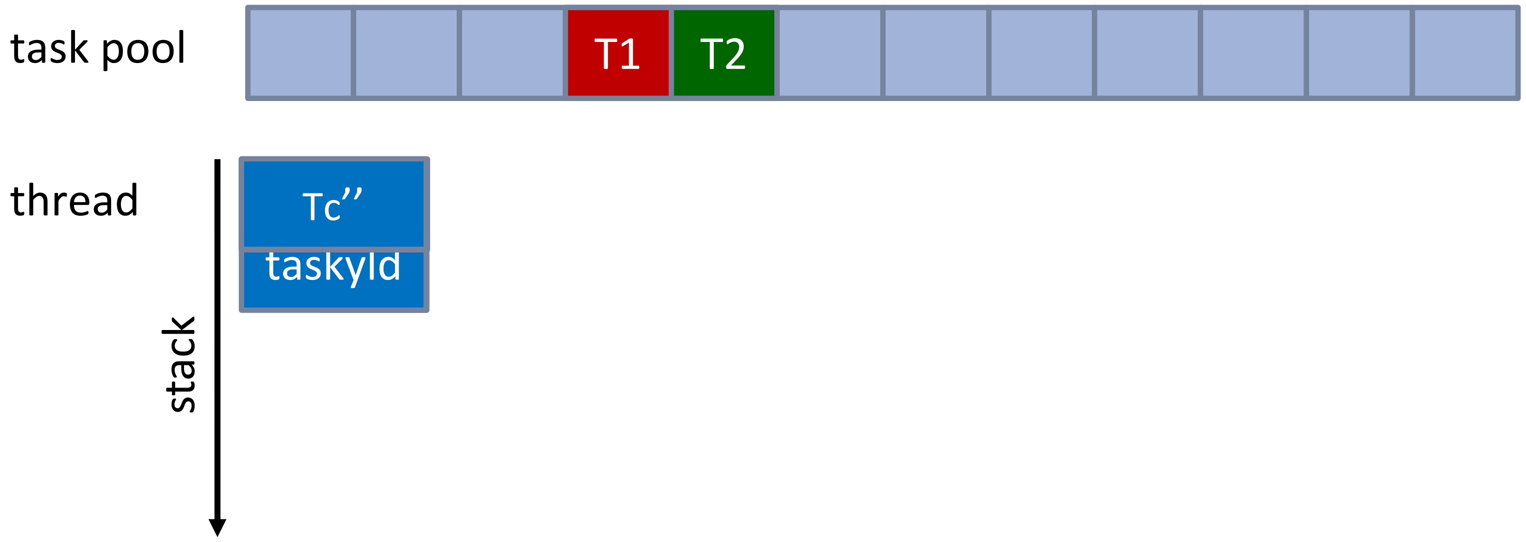
taskyield Implementation: Stack

- Simplification: assume only one task pool.
- Task "Tc" contains a taskyield directive.



`taskyield` Implementation: Cyclic

- Simplification: assume only one task pool.
- Task “Tc” contains a taskyield directive.



Implementation Choices for `taskyield` Directive (and TSPs)

- No-op: simply ignore the `taskyield` directive.
  - Simplest solution.
  - But executing thread is not freed (might lead to deadlocks with locks!)

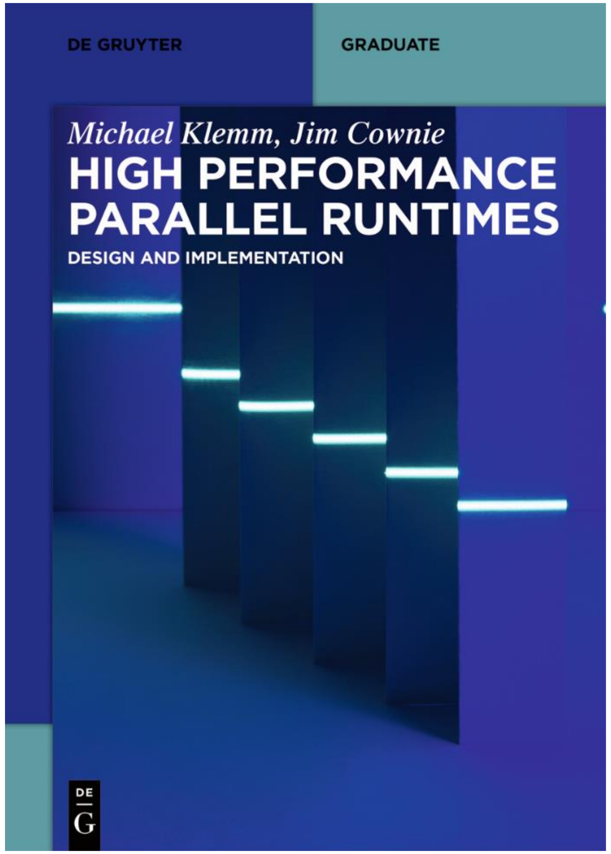
- Stack-based: suspend the current task but keep it on the execution stack.
  - Simple solution, new tasks are invoked while suspended task is still “active”.
  - If stack depth is exceeded, implementation will need to fall back to no-op implementation.

- Cyclic (for `untied` tasks): suspend current task and put into the task pool.
  - Most complex solution: continuation needs to be store in the task descriptor (or: split the tasks at TSPs into many individual sub-tasks w/ scheduling constraint).
  - Only works for `untied` tasks, as resuming thread might be different from suspending thread.

Summary

- OpenMP tasking requires a tight interplay between the compiler and the runtime system.
- OpenMP task data is allocated from the parent task.
- OpenMP implementations are based on multiple task pools for improved locality properties.
- Task waiting is optimized for throughput not for wake-up latency.

Shameless plug!



Tasks in the OpenMP API

(A behind-the-scenes glimpse)

Dr.-Ing. Michael Klemm

Chief Executive Officer
OpenMP Architecture Review Board
michael.klemm@openmp.org

Principal Member of Technical Staff
HPC Center of Excellence at AMD
michael.klemm@amd.com