

# OpenMP 3.0 Overview

**Mark Bull**

**EPCC, University of Edinburgh  
and OpenMP ARB**

# Status

- OpenMP language committee is active and working hard on OpenMP 3.0
- Weekly meetings + email + wiki
- Have agreed on ~15 topics to work on out of possible 50
  - ◆ some of these are stronger candidates than others
  - ◆ in various states of readiness
  - ◆ not all may make it into 3.0

# Tasks

- Adding tasking is the biggest addition for 3.0
- Being worked on by a separate subcommittee
  - ◆ led by Jay Hoeflinger at Intel
- Re-examined issue from ground up
  - ◆ not rubber-stamping Intel taskq's
- Main ideas are agreed, still working on some details.
- This is a snapshot of current status: not the final proposal
  - ◆ things may still change!

# Task directive

```
#pragma omp task [<clause>] ...  
    <structured block>
```

- A task is generated each time a thread (the encountering thread) encounters a task directive.
- A task is executed by a thread, called the task-thread.
- A task is possibly-deferred work. The task-thread may be the encountering thread or any other thread in the encountering thread's team.
- A task barrier blocks the thread that encounters it until a set of associated tasks is completed.
- Any thread may execute pending tasks when it is waiting at a task barrier that it encounters, or at a team barrier for its team.
- A given task must be completed by next task barrier to which it is associated or the next team barrier of the team containing its encountering thread, whichever comes first.

# Barriers

- Two types of task barriers
  - ◆ taskwait – thread waits here until all tasks generated in the current task (or thread if no task) are complete
    - #pragma omp taskwait
  - ◆ taskgroup – thread waits at end of structured block until all tasks generated by the execution of the structured block are complete
    - #pragma omp taskgroup  
<structured block>
- Thread team barriers (implicit and explicit)
  - ◆ #pragma omp barrier
  - ◆ Implicit barrier at end of structured block for parallel or any worksharing construct
  - ◆ Task behavior is the same as for taskwait

# Task directive

- Data sharing attribute clauses (final names not determined):
  - ◆ <captureaddress>
  - ◆ <capturevalue>
  - ◆ <taskprivate>
- Can be nested inside worksharing constructs
- Defaults for data sharing attributes are under discussion
- Each task is executed by a single thread, although can use a parallel directive within a task.
- Tasks can be nested inside other tasks.

# Tasking example

```
#pragma omp parallel
{
    #pragma omp single {
        for ( elem = l->first; elem; elem = elem->next)
            #pragma omp task capturevalue(elem)
                process(elem)
    }
    // all tasks are complete by this point
}
```

# Switching

- **Task switching point**

- ◆ Point in the program where a thread is allowed to switch from executing one task to executing another task.
  - **Task suspend and resume points**

- **Thread switching point**

- ◆ Point in a task where it is allowed for the task-thread to change. Only allowed when the `<switch>` clause is used on the enclosing task in the current lexical scope.
  - **Thread changes from a suspend to a resume point**

# Switching

- **Suspend points**
  - ◆ At a task directive
  - ◆ At the end of a taskgroup construct
  - ◆ At a taskwait
- **Resume points**
  - ◆ Immediately after task construct
  - ◆ Immediately after a taskgroup construct
  - ◆ Immediately after a taskwait

# Suspend and Resume points

task suspend  
point

```
#pragma omp task
```

```
{
```

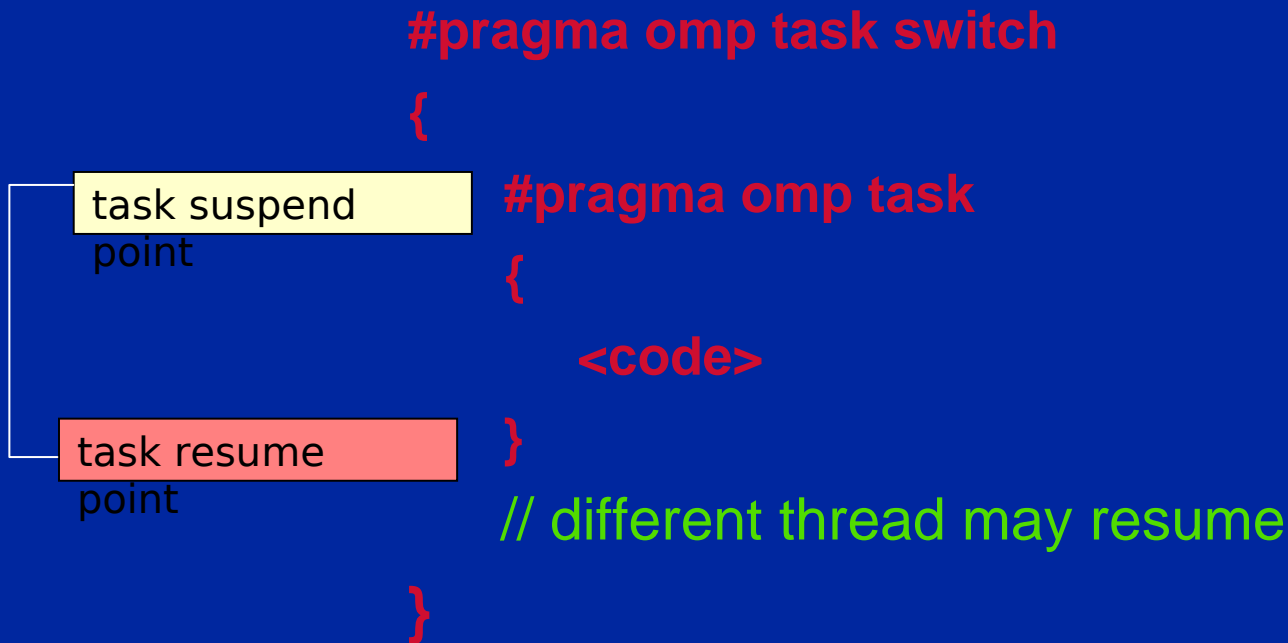
```
<code>
```

```
}
```

task resume  
point

# Suspend and Resume points

```
#pragma omp task switch
{
  task suspend point
  #pragma omp task
  {
    <code>
  }
  task resume point
  // different thread may resume
}
}
```



# Thread-switching

- User can allow thread-switching within a task by using a `<switch>` clause on the task directive.
- Thread-switching allows the encountering thread to start executing tasks, and a different thread to take its place.
  - ◆ Depth-first execution of tasks is desirable to keep the cache hot (encountering-thread becomes task-thread). Thread switching allows task parallelism with depth-first execution.
  - ◆ On breadth-first execution of tasks, if task pool becomes full, encountering thread can execute some tasks to empty it. Without thread-switching you can get starvation if the encountering thread executes a very long-running task.
- Thread-switching has the problem that the thread number will appear to have changed from one part of the task to another and any threadprivate location used will change. Hence it is not the default!

# Still To Be Done

- Determine other clauses for a task
  - ◆ Reduction?
  - ◆ Ordered?
- What is the default data sharing attribute for variables not specified in a task clause?
- Decide names for everything
- Make a reference implementation, evaluate it.
- Craft good wording for a proposal.

# Nested parallelism

- Better support for nested parallelism
- Multiple internal control variables
  - ◆ Allows, for example, calling `omp_set_num_threads()` inside a parallel region.
- Library routines to determine depth of nesting and IDs of parent/grandparent etc. threads.
- Allow threadprivate variables to persist across inner parallel regions
- Looking for a way of describing the nesting structure up-front so the runtime can make intelligent thread placement decisions

# Parallel loops

- Guarantee that this works:

```
!$omp do schedule(static)
do i=1,n
    a(i) = ....
end do
!$omp end do nowait
!$omp do schedule(static)
do i=1,n
    .... = a(i)
end do
```

# Loops (cont.)

- Allow collapsing of perfectly nested loops

```
!$omp parallel do collapse(2)
do i=1,n
    do j=1,n
        .....
    end do
end do
```

- Will form a single loop and then parallelise that

# Loops (cont.)

- Make `schedule(runtime)` more useful
  - ◆ can set it with library routine
  - ◆ allow implementations to implement their own schedule kinds
- Add a new schedule kind which gives full freedom to the runtime to determine the scheduling of iterations to threads.

# Portable control of threads

- Add environment variable to control the size of child threads' stack
- Add environment variable to hint to runtime how to treat idle threads

**ACTIVE**      keep threads alive at barriers/locks

**PASSIVE**    try to release processor at barriers/locks

**not set**      use implementation specific controls

# NUMA support

- **Leading candidate is a `MIGRATE_NEXT_TOUCH` directive**
  - ◆ **move the data to the node where the next thread to access it is running**
  - ◆ **would be ignored on UMA systems or NUMA systems that can't support it**
- **Still up for discussion and not certain to make it into 3.0**

# Odds and ends

- Allow unsigned ints in parallel for loops
- Disallow use of the original variable as master threads private variable
- Make it clearer where/how private objects are constructed/destroyed
- Relax some restrictions on allocatable arrays and Fortran pointers
- Plug some minor gaps in memory model
- Improve C/C++ grammar