

Towards OpenMP 3.0

Mark Bull

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

Status

- **OpenMP language committee is final stages of editing the OpenMP 3.0 specification**
- **A Draft Specification for public comment will be available by SC'07 (early November)**

Acknowledgements

Bronis de Supinski

Greg Bronevetsky

Dieter an Mey

Christian Terboven

Lei Huang

Barbara Chapman

Alex Duran

Eduard Ayguade

Michael Suess

Gabriele Jost

Randy Meyer

Kelvin Li

Guansong Zhang

Michael Wong

Priya Unnikrishnan

Diana King

Ernesto Su

Judy Ward

Tim Mattson

Xinmin Tian

Grant Haab

Jay Hoeflinger

Larry Meadows

Sanjiv Shah

Jeff Olivier

Henry Jin

Michael Wolfe

Eric Duncan

Nawal Coptý

Yuan Lin

James Beyer

Federico Massaoli

Brian Bliss

Tasks

- Adding tasking is the biggest addition for 3.0
- Worked on by a separate subcommittee
 - ◆ led by Jay Hoeflinger at Intel
- Re-examined issue from ground up
 - ◆ quite different from Intel taskq's

General task characteristics

- **A task has**
 - ◆ **Code to execute**
 - ◆ **A data environment (it *owns* its data)**
 - ◆ **An assigned thread that executes the code and uses the data**
- **Two activities: packaging and execution**
 - ◆ **Each encountering thread packages a new instance of a task (code and data)**
 - ◆ **Some thread in the team executes the task at some later time**

Definitions

- ***Task construct*** – task directive plus structured block
- ***Task*** – the package of code and instructions for allocating data created when a thread encounters a task construct
- ***Task region*** – the dynamic sequence of instructions produced by the execution of a task by a thread

Tasks and OpenMP

- Tasks have been fully integrated into OpenMP
- Key concept: OpenMP has always had tasks, we just never called them that.
 - ◆ Thread encountering `parallel` construct packages up a set of *implicit* tasks, one per thread.
 - ◆ Team of threads is created.
 - ◆ Each thread in team is assigned to one of the tasks (and *tied* to it).
 - ◆ Barrier holds original master thread until all implicit tasks are finished.
- We have simply added a way to create a task explicitly for the team to execute.
- Every part of an OpenMP program is part of one task or another!

task Construct

```
#pragma omp task [clause[[,clause] ...]  
    structured-block
```

where *clause* can be one of:

```
if (expression)  
untied  
shared (list)  
private (list)  
firstprivate (list)  
default( shared | none )
```

The `if` clause

- When the `if` clause argument is false
 - ◆ The task is executed immediately by the encountering thread.
 - ◆ The data environment is still local to the new task...
 - ◆ ...and it's still a different task with respect to synchronization.
- It's a user directed optimization
 - ◆ when the cost of deferring the task is too great compared to the cost of executing the task code
 - ◆ to control cache and memory affinity

When/where are tasks complete?

- **At thread barriers, explicit or implicit**
 - ◆ applies to all tasks generated in the current parallel region up to the barrier
 - ◆ matches user expectation
- **At task barriers**
 - ◆ applies only to tasks generated in the current task, not to “descendants”
 - ◆ **structured flavor:** `#pragma omp taskgroup`
 - ◆ **unstructured flavor:** `#pragma omp taskwait`

Task Synchronization

```
#pragma omp taskgroup  
    structured-block
```

Encountering task suspends at end of structured block until all children tasks created within structured block are complete.

```
#pragma omp taskwait
```

Encountering task suspends at the point of the directive until all children tasks created within the encountering task up to this point are complete.

Thread barrier (implicit or explicit) includes an implicit `taskwait`.

Example – parallel pointer chasing using tasks

```
#pragma omp parallel
{
  #pragma omp single private(p)
  {
    p = listhead ;
    while (p) {
      #pragma omp task
      process (p)
      p=next (p) ;
    }
  }
}
```

p is firstprivate by default here



Example – parallel pointer chasing on multiple lists using tasks

```
#pragma omp parallel
{
    #pragma omp for private(p)
    for ( int i =0; i <numlists ; i++) {
        p = listheads [ i ] ;
        while (p ) {
            #pragma omp task
                process (p)
            p=next (p ) ;
        }
    }
}
```

Example: postorder tree traversal

```
void postorder(node *p) {  
    #pragma omp taskgroup {  
        if (p->left)  
            #pragma omp task  
                postorder(p->left);  
        if (p->right)  
            #pragma omp task  
                postorder(p->right);  
    } ← suspend point  
    process(p->data);  
}
```

- Parent task suspended until children tasks complete
- Structured directive (as opposed to standard barriers)
 - ◆ clearly defines the scope

Task switching

- Certain constructs have suspend/resume points at defined locations within them
- When a thread encounters a suspend/resume point, it is allowed to suspend the current task and execute another (called *task switching*)
- It can then return to the original task and resume

Task switching example

```
#pragma omp single
{
    for (i=0; i<ONEZILLION; i++)
        #pragma omp task
            process(item[i]);
}
```

- Too many tasks generated in an eye-blink
- Generating task will have to suspend for a while
- With task switching, the executing thread can:
 - ◆ execute an already generated task (draining the “*task pool*”)
 - ◆ dive into the encountered task (could be very cache-friendly)

taskyield directive

- User-inserted suspend/resume point

```
#pragma omp taskyield
```

Thread switching

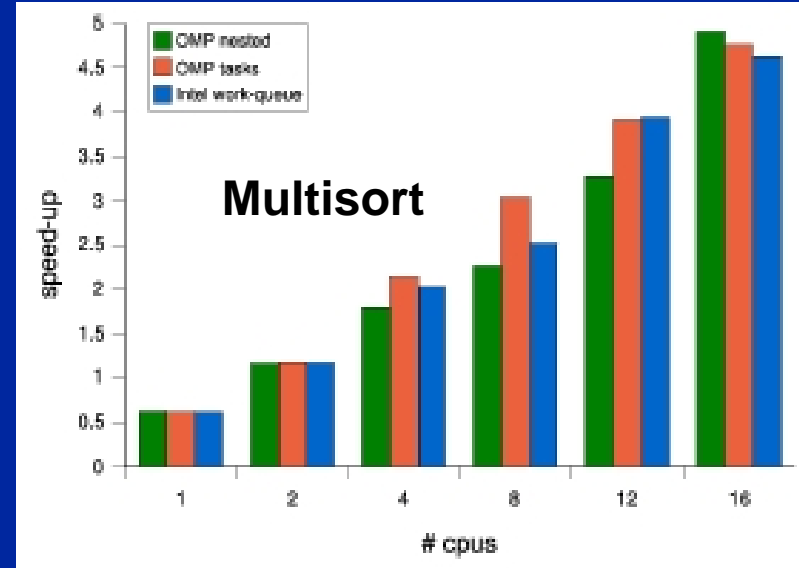
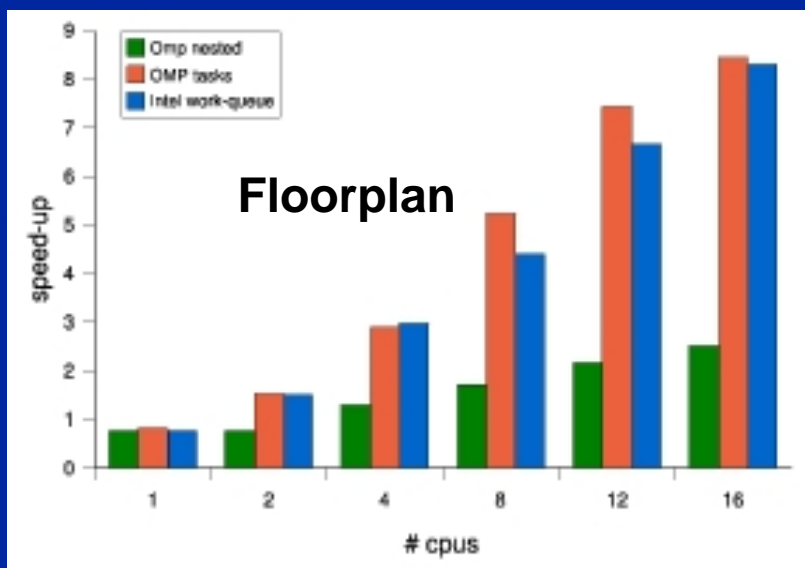
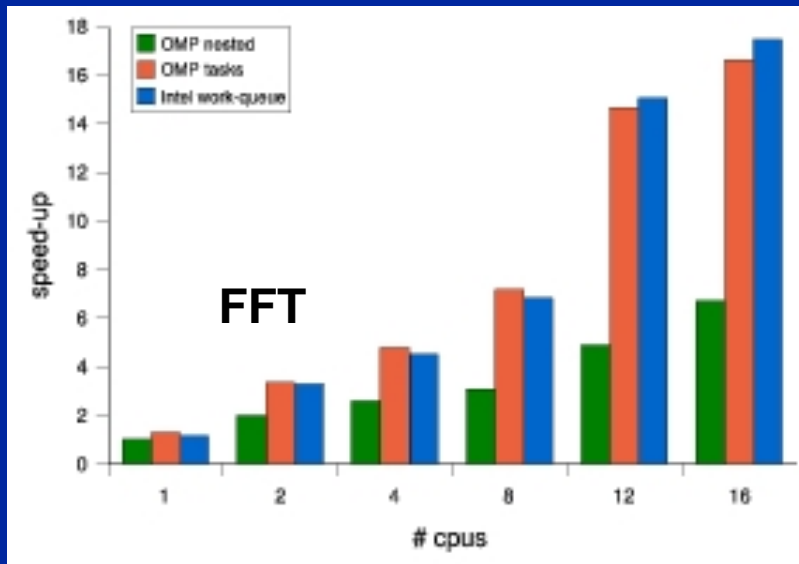
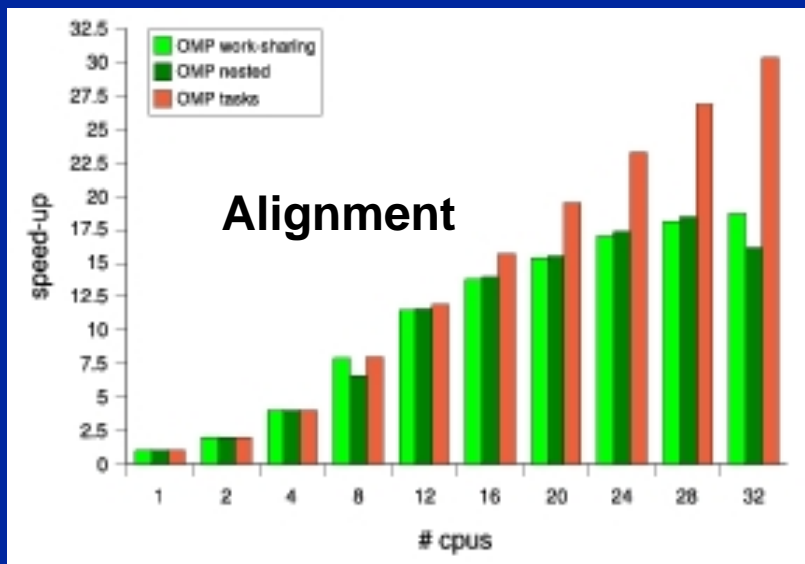
```
#pragma omp single
{
    #pragma omp task untied
    for (i=0; i<ONEZILLION; i++)
        #pragma omp task
        process(item[i]);
}
```

- Eventually, too many tasks are generated
- Generating task is suspended and executing thread switches to a long and boring task
- Other threads get rid of all already generated tasks, and start starving...
- With thread switching, the generating task can be resumed by a different thread, and starvation is over
- Too strange to be the default: the programmer is responsible!

`taskprivate` directive

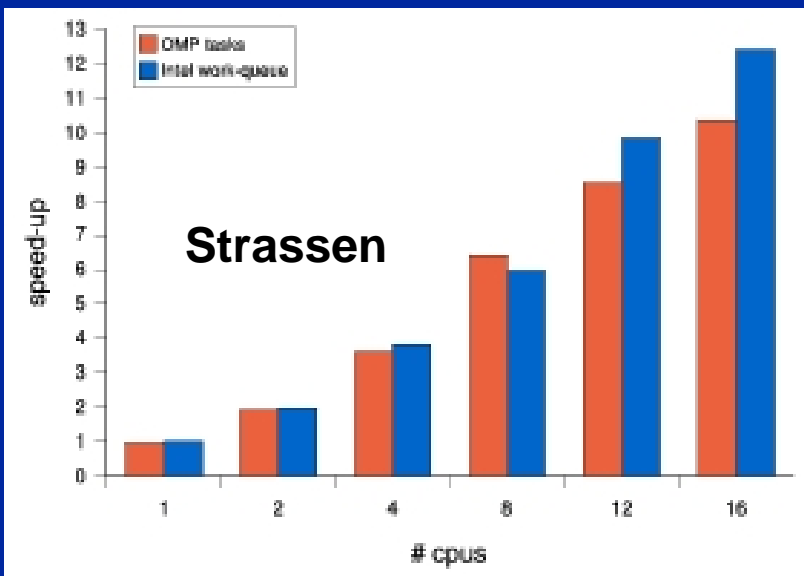
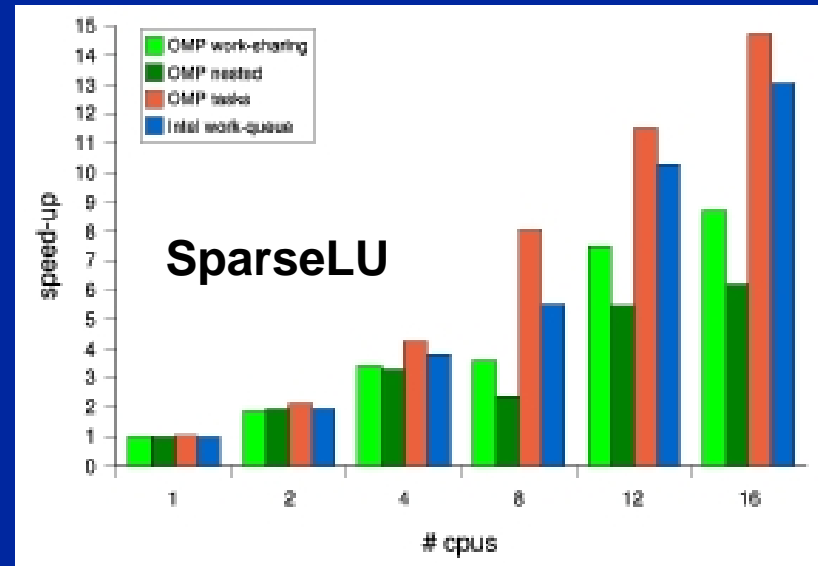
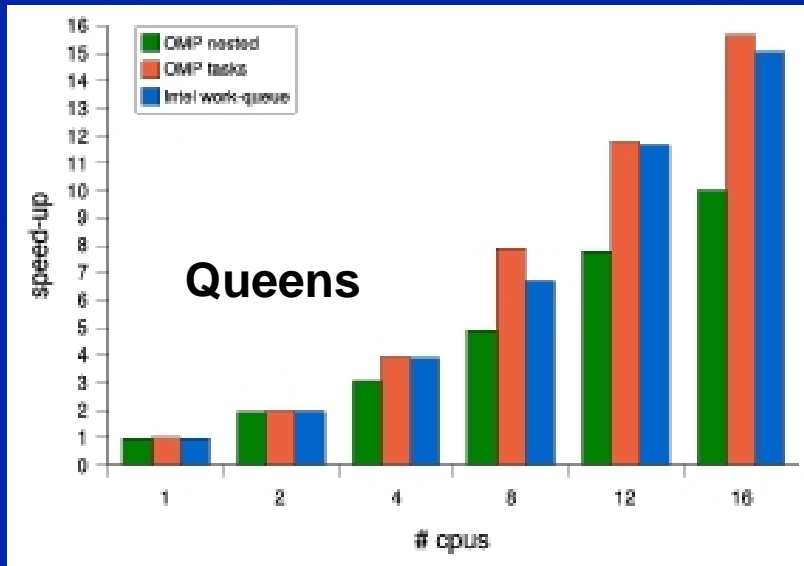
- Threadprivate variables and tasks don't mix well
 - ◆ Don't know which thread will execute the task
 - ◆ For untied tasks, more than one thread may execute different parts of the task
- Added `taskprivate` directive which gives one copy of global-lifetime data to each task

Performance Results 1



All tests run on SGI Altix 4700 with 128 processors

Performance Results 2



All tests run on SGI Altix 4700 with 128 processors

Reference Implementation

- URL:

<http://mercurium.pc.ac.upc.edu/nanos>

- Made by Xavier Teruel, Roger Ferrer,
Alex Duran, Eduard Ayguadé,
Xavier Martorell

Conclusions on tasks

- Enormous amount of work by many people
- Tightly integrated into 2.5 spec
- Flexible model for irregular parallelism
- Provides balanced solution despite often conflicting goals
- Appears that performance can be reasonable

Nested parallelism

- Better support for nested parallelism
- Per-thread internal control variables
 - ◆ Allows, for example, calling `omp_set_num_threads()` inside a parallel region.
 - ◆ Controls the team sizes for next level of parallelism
- Library routines to determine depth of nesting, IDs of parent/grandparent etc. threads, team sizes of parent/grandparent etc. teams

```
omp_get_nested_level()  
omp_get_ancestor(level)  
omp_get_teamsize(level)
```

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.)

- Made `schedule(runtime)` more useful
 - ◆ can get/set it with library routines

```
omp_set_run_sched()  
omp_get_run_sched()
```
 - ◆ allow implementations to implement their own schedule kinds
- Added a new schedule kind AUTO which gives full freedom to the runtime to determine the scheduling of iterations to threads.
- Allowed C++ RandomAccessIterators as loop control variables in parallel loops

Portable control of threads

- Added environment variable to control the size of child threads' stack

`OMP_STACKSIZE`

- Added environment variable to hint to runtime how to treat idle threads

`OMP_WAIT_POLICY`

`ACTIVE` keep threads alive at barriers/locks

`PASSIVE` try to release processor at barriers/locks

- **Added environment variable and runtime routines to get/set the maximum number of active levels of nested parallelism**

`OMP_MAX_NESTED_LEVELS`

`omp_set_max_nested_levels()`

`omp_get_max_nested_levels()`

- **Added environment variable to set maximum number of threads in use**

`OMP_THREAD_LIMIT`

`omp_get_thread_limit()`

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
- Allow C++ static class members to be threadprivate
- Improve C/C++ grammar
- Minor fixes and clarifications to 2.5

Summary

- **OpenMP 3.0 is almost ready**
- **Been a lot of hard work by a lot of people**
- **We hope you like it: let us know via the public comment process what you think!**