



OpenMP 3.0

Mark Bull
EPCC, University of Edinburgh

markb@epcc.ed.ac.uk

- History
- Tasks
- Nested Parallelism
- Parallel Loops
- Portable Control of Threads
- Odds and Ends

- Historical lack of standardisation in shared memory directives. Each vendor did their own thing.
 - mainly directive based, almost all for Fortran
 - previous attempt at standardisation (ANSI X3H5, based on work of Parallel Computing forum) failed due to political reasons and lack of vendor interest.
- OpenMP forum set up by Digital, IBM, Intel, KAI and SGI. Now includes most major vendors (and some academic organisations, including EPCC).
- OpenMP Fortran standard released October 1997, minor revision (1.1) in November 1999. Major revision (2.0) in November 2000.

- OpenMP C/C++ standard released October 1998. Major revision (2.0) in March 2002.
- Combined OpenMP Fortran/C/C++ standard (2.5) released in May 2005.
 - no new features, but extensive rewriting and clarification
- Version 3.0 released in May 2008
 - new features, including tasks, better support for loop parallelism and nested parallelism

- Addition of tasking is the biggest change between 2.5 and 3.0
- Re-examined issue from ground up
 - ended up with something quite different from Intel's `taskq` extension

- 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
- Nesting
 - Nesting of tasks is permitted: a task may itself generate other tasks.

- *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 have been fully integrated into OpenMP
- Key concept: OpenMP has always had tasks, but they were never called 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.
- 3.0 simply adds 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!

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

where *clause* can be one of:

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

- 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

- At barriers, explicit or implicit
 - applies to all tasks generated in the current parallel region up to the barrier
 - matches user expectation

- At a `taskwait` directive
 - applies only to child tasks of the current task, not to further “descendants”

Example 1

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) ;
        }
    }
}
```


one thread generates all the tasks

p is firstprivate by default here

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 ) ;
        }
    }
}
```

all threads generate tasks




Example 3

Postorder tree traversal:

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

parent task suspended until
child tasks complete



- Certain constructs have task scheduling points at defined locations within them
- When a thread encounters a task scheduling 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

```
#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 a previously generated task (draining the “*task pool*”), or
 - dive into the encountered task (can be very cache-friendly)

```
#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!

- 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_level()
```

```
omp_get_active_level()
```

```
omp_get_ancestor_thread_num(level)
```

```
omp_get_team_size(level)
```

(N.B. new, simpler definition of active parallel region: a parallel region executed by more than one thread)

- Guarantee that this works, by mandating that the same iterations are done by the same threads in both loops:

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

- Allows collapsing of perfectly nested loops

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

- Will form a single loop (with $n*m$ iterations, in the natural order) and then parallelise that.
- Parameter determines how many loops are collapsed

- Made **schedule(runtime)** more useful
 - can get/set it with library routines
 - `omp_set_schedule()`
 - `omp_get_schedule()`
 - allow implementations to implement their own schedule kinds as extensions
- Added a new schedule kind AUTO which gives full freedom to the runtime to determine the scheduling of iterations to threads.
- Allowed unsigned ints and C++ RandomAccessIterators as loop control variables in parallel loops

- 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 try to 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 (and runtime routine to query it)

OMP_THREAD_LIMIT

omp_get_thread_limit()

- Disallowed use of the original variable as master thread's private variable
- Made it clearer where/how private objects are constructed/destroyed
- Relaxed some restrictions on allocatable arrays
- Plugged some minor gaps in memory model
- Allowed C++ static class members to be threadprivate
- Minor fixes and clarifications to 2.5