Introduction to OpenMP

Tasks

Nick Maclaren

Computing Service

nmm1@cam.ac.uk

June 2011
OpenMP Tasks

In OpenMP 3.0 with a slightly different model
A form of explicit but virtual threading
Mapped in a complex way to OpenMP threads

- This course will not cover the details of that

Useful for unstructured or irregular problems

Can be hierarchical (i.e. tasks within tasks)
Called descendent tasks, child tasks or subtasks
Their Major Gotcha

The structured block and aliasing rules apply
- And all in the context of a tree structure
Need iron-clad disciplined coding to avoid problems

⇒ This is seriously tricky to get right

In C/C++, watch out for implicit sharing
E.g. in class methods and some library functions

- This course will cover only their simplest use
Essentially just as dynamic, nestable sections
Basic Syntax

Fortran:

```fortran
!$OMP TASK [ clauses ]
< structured block >
!$OMP END TASK
```

C/C++:

```c
#pragma omp task [ clauses ]
< structured block >
```

Clause syntax is rather like `parallel`
I.e. `default, private, shared and firstprivate`
Data Environment (1)

This is very **poorly specified** and solid with **gotchas**

- If **task** construct is lexically within **parallel**
  Default is usually **inherited**, which is what is wanted

- **Otherwise**, default is generally **firstprivate**
  No problem when **reading** the **values** in **task** construct
  But it will generally **copy** the whole **variable**

- May need to specify **shared** for efficiency
  E.g. when tasks use separate **array sections**
  Still mustn’t update same element in **parallel tasks**
Behaviour (1)

Tasks can create **descendants** to form a task **tree**
Just use **task** within the **structured block**

The **descendant** may run in parallel to its **parent**
Or suspend the parent and run **synchronously**

• Do **not** write code that assumes **either behaviour**

Some clauses control this, to a limited extent
Specification is **bizarre** and **ambiguous**
Hierarchical Trees

Implicit (initial) task

Task 1

1.1

1.1.1

1.1.1.1

1.1.2

Task 2

1.2

1.2.1

Task 3

3.1

3.1.1

3.1.2

3.2
Behaviour (2)

• Avoid starting more tasks than available threads
  And that means available in the parallel region

Will work if you use just the facilities taught here
But there are lots of gotchas if you go beyond them

You can control some of that, but horribly complicated
There are some brief references to the features later

• It’s safe if you don’t use any synchronisation
  Except simple uses of critical and atomic (see later)
Data Environment (2)

Child task may need to return result to parent
Parent must share a private variable with the child
• You should use shared and be careful

• The variable must not move or go out of scope
So ensure that you call taskwait before it does

Unclear about Fortran pointers and allocatables
• Do not reallocate or change pointers
While shared by parent and any active tasks

Similar remarks may apply about C++ containers
Shared and Arguments/References

When you use `task` in a procedure
Can `task` have an argument that is shared?

- Yes, but always call `taskwait` before returning
I.e. do so in the same procedure that used `task`

Literal reading of specification states that is not so
At least for Fortran and C++ reference arguments
OpenMP’s specification conflicts with the standards

- Call `taskwait` before the name goes out of scope
Same applies to all block-scoped references
Thread-specific Data

Serious problems to do with thread-specific data
Including threadprivate, OpenMP thread ids, errno, IEEE 754 flags/modes, and even C++ exceptions

The details are far too foul to describe in this course

- Do not trust any of these over a task boundary
- Do not mark any of them shared, even indirectly
  E.g. by Fortran and C++ reference arguments
- Don’t use both threadprivate and tasks
Waiting for Completion (1)

The `taskwait` directive is a sort of `barrier`.
Waits for all immediate `child tasks` to finish.

**Fortran:**

```fortran
!$OMP TASKWAIT
```

**C/C++:**

```c
#pragma omp taskwait
```

Like `barrier`, mustn’t be executed `conditionally`.
No good reason for that restriction, but don’t do it.
Waiting for Completion (2)

At the end of the **structured block**, what happens? Does it wait for all of its **child tasks** or not? The **specification** says nothing useful – assume either

- End each **structured block** with a **taskwait**

It **does wait** at the end of a **parallel region**
For all tasks and descendants in that **parallel region**

- Relying on this has its uses but is **trickier**
E.g. can write a dynamic **parallel sections**
Barriers and Task Completion

*barrier* and *taskwait* are not interchangeable
Neither implies the other, though there are links

- Don’t use *barrier* with *active tasks*

⇒ And that means *implicit barriers*, too
That means all *workshare* constructs, like *DO*/for

Using *barrier* with *active tasks* is possible
It’s *tricky* and not covered in this course
Other Restrictions (1)

- No reduction operations inside tasks

- Rules for avoiding deadlock were given above. Just follow them with tasks replacing workshare.

Can use task within a workshare construct. A fairly insane idea, and probably very inefficient.

⇒ Except for single, as described below. That (and master) is a trick to get tasks started.
Other Restrictions (2)

- **Worksharing** cannot be used within a **task**
  Though you can use **parallel workshare** constructs

⇒ Be warned – this is **nested parallelism**

Do **NOT** do this without learning about **nesting**
Must enable it **explicitly**, and **tuning** is **tricky**
- **It is too complicated to cover in this course**

- **Same applies to many other complications**
A Recursion Gotcha

Tasks can create recursion in non-recursive code.
Applies to all procedures called from within tasks.

Task A is suspended inside such a procedure.
Task B is scheduled on the same thread as task A.

- **Within tasks**, make all procedures recursive.

In Fortran, mark all such procedures **RECURSIVE**.
In C/C++, use **non-auto** data only **read-only**.
And see the next lecture about **Program Global State**.
Synchronisation Inside Tasks

- Avoid **master** and explicit **thread id** checks
  Tasks bound to a **single**, **arbitrary** OpenMP thread
  With **untied**, they can **change thread** dynamically

**single** is almost certainly **asking for trouble**

**critical** can be used for task synchronisation etc.
**Watch out** if you use features not in this course
  - Do **NOT** use tasking within **critical**
  - Do **NOT** call **SMP-capable** libraries in it
  - **critical** is **NOT** safe in **untied** tasks
Using Tasks for Workshare (1)

One simple use is write your own workshare
Then use that just like any other workshare construct

Need to embed it in single (or master+barrier)
That thread then starts all the top-level tasks
Waits for all tasks before exiting the single

- Each task waits for all subtasks before exiting
Can omit that if no subtasks but be careful
Using Tasks for Workshare (2)

Can create tasks in loops, tasks create subtasks etc. Each task waits for all descendants before exit

- Use taskwait at the end of all tasks

- Make sure that all subtrees are independent
  A subtree is a task and all its descendants

This is BY FAR the most common cause of errors
It is terribly easy to think of just one level
Fortran Task Workshare

!$OMP SINGLE [clauses]
DO . . .
  !$OMP TASK [clauses]
  . . .
  !$OMP TASK
  . . .
  !$OMP END TASK
  . . .
  !$OMP TASKWAIT
  !$OMP END TASK
END DO

!$OMP TASKWAIT
!$OMP END SINGLE
C/C++ Task Workshare

#pragma single [clauses]
{
    for (...)
    {
        #pragma task [clauses]
        {
            ...
            #pragma task
            {
                ...
            }
            ...
            #pragma taskwait
        }
    }
    #pragma taskwait
}
Task Parameters (1)

Passing **dynamic parameters** to the task is tricky. E.g. this will not work, because **index** is private:

```fortran
 !$OMP SINGLE
 DO index = 1 , count
   !$OMP TASK FIRSTPRIVATE ( index )
   CALL Fred ( index )
 !$OMP END TASK
 END DO
 !$OMP TASKWAIT
 !$OMP END SINGLE
```

Leaving out the **FIRSTPRIVATE** doesn’t work, either
Task Parameters (2)

Need to share index, but best done indirectly

```c
!$OMP PARALLEL SHARED ( copy )
!$OMP SINGLE
DO index = 1 , count
  copy = index
  !$OMP TASK FIRSTPRIVATE ( copy )
  CALL Fred ( index )
!$OMP END TASK
END DO
!$OMP TASKWAIT
!$OMP END SINGLE
!$OMP END PARALLEL
```

• Note that copy is accessed in only one thread
Task Parameters (3)

C and C++ people should do the same
Though declaring index as shared might work

⇒ Even the above code is not entirely safe

Unclear when FIRSTPRIVATE is executed
 Might be in parallel with next iteration

• Could allocate array and use separate elements
Remember to deallocate after the TASKWAIT
Exactly the same applies to C and C++
Dynamic Sections

Just create a **parallel region** and do all waiting **at end**
- **All tasks** (not just **subtrees**) must be independent

This is just a **dynamic** form of **parallel sections**
A bit more **flexible** than either that or **parallel DO/for**

- **Can combine** these two simple usages
Use this form at **top level**, and previous for **subtasks**
Fortran Parallel Task Workshare

!$OMP PARALLEL [clauses]
!$OMP MASTER

...  
!$OMP TASK
...  
!$OMP TASK
...  
!$OMP END TASK
...  
!$OMP END TASK
...  
!$OMP MASTER
!$OMP END PARALLEL

Use **SINGLE** if you prefer (possibly with **NOWAIT**
C/C++ Parallel Task Workshare

```c
#pragma parallel [clauses]
{
    #pragma master
    {
        ....
        #pragma task
        {
            ....
            #pragma task
            {
                ....
                ....
            }
            ....
        }
    }
    ....

}
Last Word

Beyond these simple usages, there be dragons ...