Introduction to OpenMP

Tasks

N.M. Maclaren

nmm1@cam.ac.uk

September 2019

6.1 OpenMP Tasks

6.1.1 Introduction

These were introduced by OpenMP 3.0 and use a slightly different parallelism model from the previous facilities. They are a form of explicit but virtual threading, where the virtual threads are mapped in a complex way to OpenMP threads. This course will not cover the details of that mapping.

They can be useful for unstructured or irregular problems, and can be hierarchical (i.e. tasks can be started from within tasks). These are called any of descendent tasks, child tasks or subtasks, and the originating task is the parent task.

The biggest difficulty in using such features is that the *structured block* and aliasing rules apply, unchanged, but in the context of a tree structure; you need iron-clad self-discipline to code in such a way as to avoid illegal aliasing. *Experience shows that this is seriously tricky to get right.* In C and C++, watch out for implicit sharing, such as caused by class methods and some library functions. This course will cover only their simplest use, which is essentially just as dynamic, nestable `sections`. The syntax is:

Fortran:

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

C/C++:

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

The clause syntax is rather like the `parallel` construct: i.e. `default`, `private`, `shared` amd `firstprivate`.

The data environment is very poorly specified and solid with *gotchas*, so be careful. If the `task` construct is lexically within a `parallel` one, the default is usually inherited, which is what is wanted.; otherwise, the default is generally `firstprivate`. This does not cause problems when merely reading the parent's values within the task construct, but will generally copy the whole variable. You may need to specify `shared` for efficiency, such as when separate tasks use separate sections of an array, but you still must not update the same element in parallel tasks.

Tasks can create descendants to form a task tree – to do that, just use `task` within the structured block that runs the previous task. The descendant may run in parallel to its parent, or it may suspend the parent and run synchronously using the parent's thread (the parent will restart when it finishes). Do **not** write code that assumes either behaviour, or it may fail unexpectedly. Some clauses control this, to a limited extent, but their specification is bizarre and ambiguous and mentioned later only briefly.

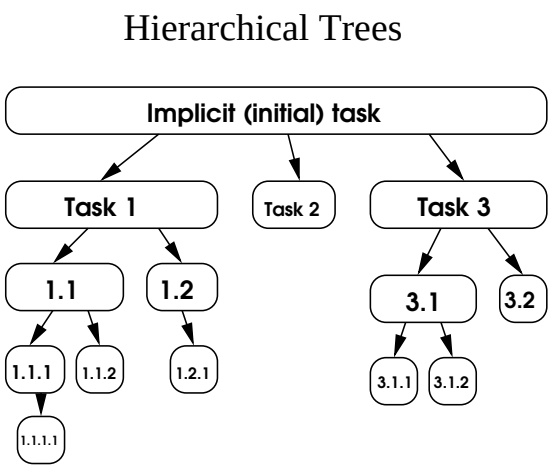

Figure 1.1

Do not start more tasks than there are available threads, which means available in the enclosing parallel region. It 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 it is horribly complicated; there are some brief references to the features later. Generally, it is safe if you do not use any synchronisation, except for simple uses of `critical` and `atomic` (covered later).

A child task may need to return a result to its parent, which means thast the parent must share one of its private variables with the child. To do this, use `shared` – but be careful. The variable must not move or go out of scope, so ensure that you call `taskwait` before it does. The specification is also unclear about the effect on C++ containers and Fortran pointers and allocatable variables. Do not reallocate them or change the pointer association; this restriction applies only while they are being shared by the parent and an active task, of course. This also means not adding or removing elements, as that can cause reallocation.

When you use `task` in a procedure, can it have an argument in its list of shared variables? The answer is *yes*, but you should **always** call `taskwait` before returning from the procedure that used `task`. The *gotcha* is that a literal reading of the specification states that it is not always needed, at least for Fortran and C++ reference arguments. Unfortunately, OpenMP's specification conflicts with the standards', and implementations will usually not support this.

- Call `taskwait` before the **name** goes out of scope, not just the variable, and do the same for all block-scoped references.

There are serious problems to do with thread-specific data, including `threadprivate`, OpenMP thread identifiers (i.e. the results from `omp_get_thread_num`), `errno`, IEEE 754 flags and modes and even C++ exceptions. The details are far too foul to describe in this course

- Do not trust any of these over a boundary.
- Do not mark any of them as `shared`, not even indirectly, which can happen when using Fortran or C++ reference arguments.
- Do not use both `threadprivate` and tasks.

There is a `taskwait` directive, which is a sort of barrier, and waits for all immediate child tasks to finish. Its syntax is:

Fortran:

```
!$OMP TASKWAIT
```

C/C++:

```
#pragma omp taskwait
```

Like `barrier`, it must not be executed conditionally (e.g. in an `if` statement or a loop). In principle, there is no good reason for that restriction, but you should still not write such code. Also, does a task wait for all of its child tasks to complete at the end of the structured block or not? The specification says nothing useful, so you should assume either, and end each structured block with a `taskwait` directive.

It does wait for all of its descendant tasks to complete at the end of a parallel region, but only for all tasks and their descendants in that parallel region. Relying on this has its uses but is trickier; for example, you can write a dynamic form of `parallel sections`.

`barrier` and `taskwait` are not interchangeable, and neither implies the other, though there are connections. You are recommended **not** to use `barrier` with active tasks; this also applies to constructs that include an implicit barrier, such as `DO/for` (whether or not `nowait` is used). Using `barrier` with active tasks is possible, but is tricky and not covered in this course.

You should not use reduction operations inside tasks – it is unclear why this is forbidden, but it is. Rules for avoiding deadlock were given above; just follow them with tasks replacing worksharing constructs in the description. You can also use `task` within a worksharing construct, but this is a fairly insane idea, and probably very inefficient. The exception is the `single` usage, which (together with `master`) is described below.

Worksharing directives cannot be used directly within a task, though you can use parallel worksharing constructs; be warned – this is nested parallelism. Do **not** do this without learning about nesting. You must enable it explicitly, and tuning is tricky; it is too complicated to cover in this course.

Tasks can create recursion in even non-recursive code; this applies to **all** procedures called from within tasks, not just ones called directly. What happens is that Task **A** is executing inside such a procedure and is suspended; task **B** is then scheduled, and runs on

the same thread as task **A**. So, within tasks, you should make all procedures pure, which is much stronger than merely recursive. Any static data, including Fortran initialised variables, must be read-only and not changed by anything else while the task might be running, and you must not call any procedure that doesn't do that. In Fortran, mark all such procedures `RECURSIVE`. And, especially in C and C++, see the next lecture about Program Global State.

If you use synchronisation inside tasks, do not use `master` or explicit checks on thread number, because threads are bound to a single, arbitrary OpenMP thread, and this can cause deadlock; with `untied`, they can even change thread dynamically. `single` is almost certainly asking for trouble, but `critical` can be used for task synchronisation and so on. You should watch out for deadlock if you use features not covered in this course. Some non-obvious warnings include:

- Do **not** use tasking within `critical` sections.
- Do **not** call procedures from SMP-capable libraries in `critical` sections.
- `critical` is **not** safe in untied tasks, because they may be suspended almost anywhere.

6.1.2 Using Tasks for Worksharing

One simple use of tasks is to write your own worksharing construct (essentially something that is like a dynamic `sections`), and then use that just like any other worksharing construct (e.g. `DO`/`for`).

You need to embed it in a `single` or barriered (worksharing) `master` construct; that thread then starts all of the top-level tasks, and waits for all of them before leaving the `single` construct. Each task waits for all of its subtasks before exiting; you can omit that if it creates no subtasks but be careful, because that is easy to get wrong.

You can create tasks in loops, tasks can create subtasks, and so on. In this code, each task waits for all of its descendants before it exits. The main warning is that you must make sure that all subtrees are independent of all other subtrees at the same level; a subtree is a task and all of its descendants. This is **by far** the most common cause of errors, because it is it is terribly easy to think of just one level.

Fortran Task Worksharing

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

```
        !$OMP TASKWAIT
    !$OMP END TASK
END DO
!$OMP TASKWAIT
!$OMP END SINGLE
```

C/C++ Task Worksharing

```
#pragma single [clauses]
{
    for (. . .) {
        #pragma task [clauses]
        {
            . . .
            #pragma task
            {
                . . .
            }
            . . .
            #pragma taskwait
        }
    }
    #pragma taskwait
}
```

Passing dynamic parameters to the task is tricky; for example, these will not work, because `index` is private:

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

```
#pragma omp single
{
    for (index = 0 ; index < count; ++index)
    {
        #pragma omp task firstprivate(index)
        fred(index);
     }
     #pragma omp taskwait
}
```

Leaving out the `firstprivate` does not work, either. We need to share `index`, which is best done indirectly for arcane reasons.

```
!$OMP PARALLEL SHARED ( copy )
!$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
```

```
#pragma omp parallel shared(copy)
{
     #pragma omp single
     {
         for (index = 0 ; index < count; ++index)
         {
             copy = index;
             #pragma omp task firstprivate(index)
             fred(index);
          }
          #pragma omp taskwait
     }
}
```

Note that `copy` is accessed in only one thread, and so there is no race condition. C and C++ programmers should do the same, though they are more likely to get away with declaring `index` as shared; it is still not advisable.

⇒ **Even the above code is not entirely safe.**

The OpenMP specification is unclear about exactly when the `firstprivate` is 'executed', and some compilers might do it as part of the new task, in parallel with the next iteration of the parent. To defend against that, we could allocate and array of length `count` before the loop and use separate elements in each iteration. Remember to deallocate it **after** the `taskwait`, of course. This applies to Fortran, C and C++.

Another simple usage is even more like a dynamic `parallel sections` construct. All you do is to create a parallel region and do all the waiting at the end. In this form, **all** tasks, and not just subtrees, must be independent. This is a bit more flexible than either `parallel section` or `parallel DO/for`, but is otherwise similar. You can also combine these two simple usages, by using this form at the top level, and the previous (worksharing) form for the subtasks.

Fortran Parallel Task Worksharing

```
!$OMP PARALLEL [clauses]
!$OMP MASTER [clauses]
   . . .
    !$OMP TASK
        . . .
        !$OMP TASK
            . . .
        !$OMP END TASK
        . . .
    !$OMP END TASK
    . . .
!$OMP END SINGLE
!$OMP END PARALLEL
```

C/C++ Parallel Task Worksharing

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

You can use `single` instead of `master` if you prefer (possibly with `nowait`).

And remember that, beyond these simple usages, *there be dragons* ....