OpenMP for Computational Scientists

6: Tasking and Tools

Tom Deakin
University of Bristol

Outline

- Quick recap of `target` exercise
- Poor man's tasking: sections
- The single and master constructs
- Tasking in OpenMP
  - Generating tasks
  - Synchronising tasks
  - Data sharing rules
  - Specifying task dependencies
- Tools
- OpenMP 5.0

Previous exercise

Take your 5-point stencil code and run on a GPU with the `target` directive.

1. Copy data to/from the device.

```
!$omp target enter data map(to: A, Atmp)

do t = 1, ntimes

  call stencil(...)

end do

!$omp target exit data map(from: A, Atmp)
```

Previous exercise

2. Offload parallel execution of the loops.

```
!$omp target map(tofrom:total)
!$omp teams distribute parallel do reduction(+:total) collapse(2)
do j = 1, ny
  do i = 1, nx
    Atmp(i,j) = (A(i-1,j) + A(i+1,j) + A(i,j) + A(i,j-1) + A(i,j+1)) *
    ↪  0.2
    total = total + Atmp(i,j)
  end do
end do
!$omp end teams distribute parallel do
!$omp end target
```

Don't forget to copy back the reduction variable.

The need for tasks

- What if your code doesn't follow a standard parallel loop pattern?
- OpenMP needs to know the loop count at runtime.
- Recursive and tree/graph based algorithms inconvenient to program with parallel loops.
- Tasking allows you to package work and data into units (tasks) and have them scheduled in parallel.

Example: processing a linked list:

```
p => head
do while(associated(p))
  call process(p)
  p => p%next
end do
```

When not to use tasks

Think!

If you have a regular parallel algorithm, tasks are *unlikely* to be the best approach.
Use the worksharing directives instead.

Sections

- Sections give you a way to assign different work to different threads.
- Useful for a producer/consumer pattern, but not really recursive algorithms.
- If fewer sections than threads, threads sit idle.
- If more sections than threads, sections assigned by implementation.

```fortran
!$omp parallel sections
    !$omp section
    call work1()

    !$omp section
    call work2()

!$omp end parallel sections
```

The single construct

- Often necessary for only one thread to do some work in a parallel region.
- Wrapping code with the `single` constructs means only one thread in the parallel region will execute that code.
- Not defined which thread will actually execute the block.
- There is an implicit barrier for this construct (can avoid with `nowait` clause).

```fortran
!$omp parallel
  call do_work()
  !$omp single
  call exchange_halos()
  !$omp end single
!$omp end parallel
```

The master construct

- Similar to `single` construct, except guaranteed the master thread will execute the block.
- There is *no* implicit barrier for this construct!
- Useful for cases where synchronisation isn't required:
  - Printing out messages to the screen.
  - **Generating tasks!**

```
!$omp parallel
  call do_work()
  if (conv .eq. .true.)
    !$omp master
    print *, "Converged!"
    !$omp end master
    exit
  end if
!$omp end parallel
```

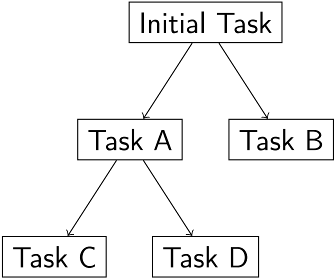
Tasks

- OpenMP 3.0 introduced real tasking.
- Later versions refined and added features.

```
!$omp parallel
  !$omp master
    !$omp task
    call do_task(x,y,z)
    !$omp end task

    !$omp task
    call do_task(i,j,k)
    !$omp end task
  !$omp end master
!$omp end parallel
```

- Use master thread to generate tasks for all threads to do.
- Idle threads take tasks off the task queue.
- Implicit barrier at the `end parallel` ensures all tasks complete.

Useful to think about tasks organised as a tree (or graph).



- The Initial Task is the *parent* of Task A.
- Task A is the *child* of the Initial Task.
- Task A and Task B are *siblings*.

Trees

```fortran
!$omp parallel
  !$omp master
    !$omp task
    call do_task(x,y,z)
    !$omp end task

    !$omp task
    call do_task(i,j,k)
    !$omp end task
  !$omp end master
!$omp end parallel
```

Trees

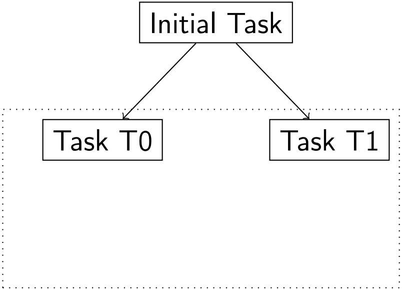
```
!$omp parallel
  !$omp master
    !$omp task
    call do_task(x,y,z)
    !$omp end task

    !$omp task
    call do_task(i,j,k)
    !$omp end task
  !$omp end master
!$omp end parallel
```

Initial Task

1. Initial thread begins serial execution.

Trees

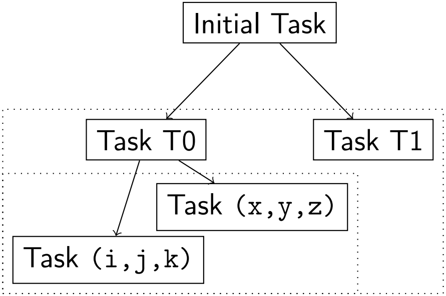
```
!$omp parallel
  !$omp master
    !$omp task
    call do_task(x,y,z)
    !$omp end task

    !$omp task
    call do_task(i,j,k)
    !$omp end task
  !$omp end master
!$omp end parallel
```



1. Initial thread begins serial execution.
2. Parallel region encountered, implicit task generated per thread.

Trees

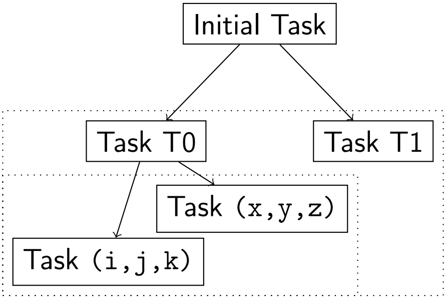
```
!$omp parallel
  !$omp master
    !$omp task
    call do_task(x,y,z)
    !$omp end task

    !$omp task
    call do_task(i,j,k)
    !$omp end task
  !$omp end master
!$omp end parallel
```



1. Initial thread begins serial execution.
2. Parallel region encountered, implicit task generated per thread.
3. Master thread T0 generates tasks. Tasks are *children* of T0, and *siblings* of each other.

Trees

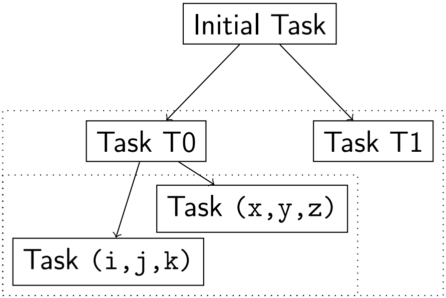
```
!$omp parallel
  !$omp master
    !$omp task
    call do_task(x,y,z)
    !$omp end task

    !$omp task
    call do_task(i,j,k)
    !$omp end task
  !$omp end master
!$omp end parallel
```



1. Initial thread begins serial execution.
2. Parallel region encountered, implicit task generated per thread.
3. Master thread T0 generates tasks. Tasks are *children* of T0, and *siblings* of each other.
4. When Tasks T0 and T1 are done they reach the implicit barrier. Threads free to execute other tasks.

Trees

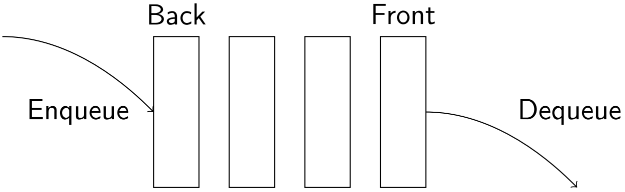
```fortran
!$omp parallel
  !$omp master
    !$omp task
    call do_task(x,y,z)
    !$omp end task

    !$omp task
    call do_task(i,j,k)
    !$omp end task
  !$omp end master
!$omp end parallel
```



1. Initial thread begins serial execution.
2. Parallel region encountered, implicit task generated per thread.
3. Master thread T0 generates tasks. Tasks are *children* of T0, and *siblings* of each other.
4. When Tasks T0 and T1 are done they reach the implicit barrier. Threads free to execute other tasks.
5. Initial task resumes after implicit barrier at `end parallel`.

Under the hood

- It's useful to think about how the OpenMP runtime itself implements tasking.
- In general, the runtime maintains a queue of tasks.
- Threads enqueue and dequeue tasks from the queue.



- Intel/Clang runtime has one task queue per thread, and allows work-stealing.
- Gives lower contention than case where all threads access a single queue.

Task completion

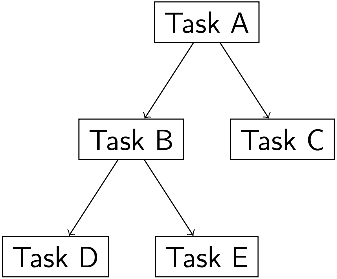
Three places where tasks synchronise:

1. At thread barriers:
   - Implicit barriers (like at end of `parallel` and `single` regions).
   - Explicit `barrier` constructs.
   - *All* previously generated tasks will complete.
2. At the `taskwait` construct:
   - Waits on all child tasks of the current task before continuing.
   - Only applies to tasks that the current task *generated*.
   - Everything in OpenMP is defined as task, so here the "outer/current" task is the one that called the first `task` construct.
3. At the end of the `taskgroup` construct:
   - Waits on all child tasks generated within the `taskgroup` region.
   - *Includes* waiting on descendants.
   - Like `taskwait`, but allows grouping of child tasks.

Taskwait

!$omp taskwait: waits for *generated* (child) tasks to finish.

```fortran
subroutine do_a()
  !$omp task
  call do_b()
  !$omp end task

  !$omp task
  call do_c()
  !$omp end task
end subroutine

subroutine do_b()
  !$omp task
  call do_d()
  !$omp end task

  !$omp task
  call do_e()
  !$omp end task
end subroutine
```



Taskwait

!$omp taskwait: waits for *generated* (child) tasks to finish.

```
subroutine do_a()
  !$omp task
  call do_b()
  !$omp end task

  !$omp task
  call do_c()
  !$omp end task
end subroutine

subroutine do_b()
  !$omp task
  call do_d()
  !$omp end task

  !$omp task
  call do_e()
  !$omp end task
end subroutine
```

```
$ OMP_NUM_THREADS=4 ./tasks
 Task A starting
 Task A finished
 Task C starting
 Task B starting
 Task C finished
 Task B finished
 Task D starting
 Task E starting
 Task E finished
 Task D finished
```

Tasks finish even when child tasks haven't started.

Taskwait: usage

- A `taskwait` in Task A will wait for Tasks B and C to finish.
- Task B finishes when Tasks D and E have been *generated*, not completed.
- So this `taskwait` *will not* wait for Tasks D and E to finish.

```fortran
subroutine do_a()
  !$omp task
  call do_b()
  !$omp end task

  !$omp task
  call do_c()
  !$omp end task

  !$omp taskwait
end subroutine
```

```
$ OMP_NUM_THREADS=4 ./tasks
 Task A starting
 Task B starting
 Task C starting
 Task B finished
 Task C finished
 Task D starting
 Task A finished
 Task E starting
 Task D finished
 Task E finished
```

Taskwait: usage

- A `taskwait` *also* in Task B will ensure Task B waits for Tasks D and E to finish before finishing itself.
- Task A waits for Tasks B to finish, which is waiting for Tasks D and E to finish.
- So Task A will now finish when Tasks B, C, D and E are finished.

```fortran
subroutine do_b()
  !$omp task
  call do_d()
  !$omp end task

  !$omp task
  call do_e()
  !$omp end task

  !$omp taskwait
end subroutine
```

```
$ OMP_NUM_THREADS=4 ./tasks
 Task A starting
 Task B starting
 Task C starting
 Task E starting
 Task D starting
 Task C finished
 Task E finished
 Task D finished
 Task B finished
 Task A finished
```

Taskgroup

- Similar to `taskwait` but also waits on children of child tasks.
- Creates a set of tasks containing *all* generated tasks, including descendants, in the region.
- Encountering task suspended until all the tasks in the set complete.

```
!$omp taskgroup

  ! Generate some tasks...

!$omp end taskgroup

! Here, all tasks generated in the taskgroup region
! have finished
```

Taskgroup: usage

```
1  !$omp parallel
2  !$omp single
3
4  !$omp task
5  call background_work()
6  !$omp end task
7
8  !$omp taskgroup
9    !$omp task
10   call recursive_work()
11   !$omp end task
12 !$omp end taskgroup
13
14 call check_work()
15
16 !$omp end single
17 !$omp end parallel
```

- Using `single` region to generate tasks, which has implicit barrier.
- Start a background task, and perform a recursive task which generates more tasks.
- The `taskgroup` waits for all the tasks in the group only, *excluding* the background task.
- Background task can continue while executing `check_work()`.
- If used a `taskwait` before line 14, it would also wait for the *background* task. Also need to ensure wait on recursive task's descendants.

Data sharing clauses

- Can use these three data sharing clauses from `parallel` regions on the `task` construct.
- The definitions are really the same, but applied to tasks, not threads.
- Need to think carefully about the appropriate clauses for each variable.
- `shared(x)`: There is one copy of the `x` variable, shared between the current and child tasks.
- `private(x)`: Each task gets its own local `x` variable. It is not initialised.
- `firstprivate(x)`: Each task gets its own local `x` variable. It is initialised to the value of the original variable when *encountered*.

Default data sharing

- Tasks and parallel regions have different default data sharing rules.
- By default, data is *shared* for `parallel` regions.
- By default, data is *firstprivate* for `task` constructs.
  - Unless, it's shared by the enclosing (the outer) region.
- Sensible behaviour because task execution may be delayed, and original variables may no longer be in scope.
- Using `default(none)` is recommended here especially.

Data sharing example

```
!$omp parallel shared(A) private(B)

  !$omp task
    call compute(A,B)
  !$omp end task

!$omp end parallel
```

Within the task:

- `A` is `shared`.
- `B` is `firstprivate`.
- Any variables inside the subroutine are `private`.

Serial Fibonacci

```fortran
recursive integer function fib(n) result(res)
  integer :: n, i, j
  if (n .lt. 2) then
    res = n
  else
    i = fib(n-1)
    j = fib(n-2)

    res = i+j
  end if
end function

program fibonacci
  integer :: res

  res = fib(40)
  print *, res
end program
```

Note: there are better ways to calculate Fibonacci numbers. . .

Parallel Fibonacci

Create the parallel region:

```fortran
program fibonacci

  integer :: res

  !$omp parallel
    !$omp master
      res = fib(40)
    !$omp end master
  !$omp end parallel

  print *, res

end program
```

Parallel Fibonacci

```fortran
recursive integer function
↪  fib(n) result(res)
  integer :: n, i, j
  if (n .lt. 2) then
    res = n
  else
    !$omp task shared(i)
    i = fib(n-1)
    !$omp end task
    !$omp task shared(j)
    j = fib(n-2)
    !$omp end task
    !$omp taskwait
    res = i+j
  end if
end function
```

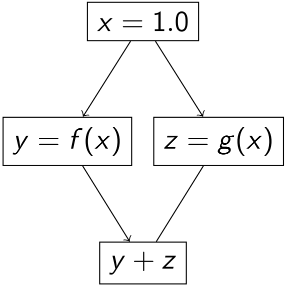
- Recursively creates a binary tree of tasks.
- This first task `fib(40)` creates two tasks: `fib(39)` and `fib(38)`.
- The `taskwait` construct ensures that the child tasks are finished before summing their return values.
- `i` and `j` *must* be `shared` so that the results are retained by the calling task.
- They are local variables so would have been `firstprivate` for the tasks by default.

Task dependencies

- OpenMP gives you the ability to define an ordering of tasks based on input and output data dependencies.
- In other words, can make tasks wait for other sibling tasks to finish before starting.
- Need to specify input and output dependencies.
- Only checks for previously generated siblings with a dependency specified.
- Add a `depend` clause to the `task` directive:
  - `depend(in: A)`: task depends on siblings with `out` or `inout` dependency `A`.
  - `depend(out: A)`: task depends on siblings with any dependency on `A`.
  - `depend(inout: A)`: same as `depend(out: A)`.

Dependency example

```fortran
!$omp task depend(out: x)
x = 1.0
!$omp end task

!$omp task depend(in: x) depend(out: y)
y = f(x)
!$omp end task

!$omp task depend(in: x) depend(out: z)
z = g(x)
!$omp end task

!$omp task depend(in: y, z)
print *, y+z
!$omp end task
```



Dependency example

```fortran
!$omp task depend(out: x)
x = 1.0
!$omp end task

!$omp task depend(in: x) depend(out: y)
y = f(x)
!$omp end task

!$omp task depend(in: x) depend(out: z)
z = g(x)
!$omp end task

!$omp task depend(in: y, z)
print *, y+z
!$omp end task
```

- Must specify first out dependency.
- Then generate $y = f(x)$ tasks which depends on it.
- Then generate $z = g(x)$ task, which depends on first but not second.
- Generate task which depends on middle two tasks.
- Note, OpenMP still sees all these tasks as *siblings*.

New dependency clauses

OpenMP 5.0 will introduce the `mutexinoutset` dependency qualifier.

- Ensures commutativity between tasks depending on `x`.
- Sibling tasks with `depend(inout: x)` would need to be run in the order the tasks were generated.
- `mutexinoutset` means they can run in any order, not just the generated order.
- Tasks still won't run in parallel.

mutexinoutset

```fortran
!$omp task (inout: y)
call slow(y) ! Task A
!$omp end task

!$omp task (inout: z)
call fast(z) ! Task B
!$omp end task

!$omp task depend(inout: res) depend(in: y)
res = res + y ! Task C
!$omp end task

!$omp task depend(inout: res) depend(in: z)
res = res + z ! Task D
!$omp end task
```

- Task A and B will start.
- Task C can start once Task A has finished.
- Task D can start once both Tasks B and C have finished.
- Ideally should just wait on Task B, not C.
- Solved by replacing `depend(inout: res)` with `depend(mutexinoutset: res)`.

Suspending tasks with taskyield

- Once a task starts executing, it goes to completion unless it encounters a *task scheduling point*.
- At these points, the threads can pick another task to start executing or resume a task.
- These points occur when you:
  - Generate a task
  - Finish a task
  - Reach the end of a `taskgroup` construct.
  - Synchronise with `barrier` or `taskwait`.
  - And, at a `taskyield` construct.
- The `taskyield` construct allows tasks to be suspended so another task can start.
- Really useful when polling for a lock, otherwise the task never "sleeps".
- Not all implementations actually suspend tasks (Intel/Clang does).

taskyield example

```
!$omp task
  call do_something()

  ! Wait for the lock
  do while (.not. omp_test_lock(lock))
    ! Couldn't get lock, so suspend task and schedule
    ↪  another
    !$omp taskyield
  end do

  ! Only one task should call this function
  call critical_code()

  ! Release the lock
  call omp_unset_lock(lock)
!$omp end task
```

The taskloop construct

- Generates a task for each loop iteration.
- Shorthand for generating tasks in a tiled `parallel do` loop.
- Tasks form part of a `taskgroup`, unless use the `nogroup` clause.
- `grainsize(num)` clause: sets the minimum number of iterations for each generated task.
- `num_tasks(num)` clause: set number of tasks to create. Unlikely to want to specify this in practice.
- `collapse` clause can also be used.

```
!$omp taskloop
do i = 1, N
  call do_work(i)
end do
!$omp end taskloop
```

Other clauses

Some rare clauses you can use when generating tasks, but unlikely to use them in most cases.

- `untied`: If the task gets suspended, any thread can start it. Without this clause, the thread which started the task must resume it. Helpful for load balancing when used with `taskyield`.
- `mergeable`: The task can be merged into the parent task, sharing its data environment. Execution might happen straight away, or be delayed.
- `priority(n)`: The task is given a priority value $n \geq 0$. Gives a hint to the runtime about which tasks to run first.
- `final(expr)`: If the expression is true, this and all further child tasks are included in the parent task, and just executed immediately. Useful for specifying a minimum tasks size in recursive algorithms.

Tasking advice

- Getting the correct data sharing clauses can be tricky.
- Don't use tasks for patterns supported by other parts of OpenMP (parallel loops).
- Tasking comes with overheads.
- The runtime is good, but can't work miracles.
- Best results obtained where programmer controls number and granularity of tasks.

Tools

Intel Parallel Studio XE

- Intel Advisor
  - Shows vectorisation efficiency and Roofline analysis.
  - Can show you where might be good to thread, but obviously can't refactor your code for you to make it parallelisable.
- Intel Inspector
  - Debugger for memory errors and thread errors (such as deadlock).
- Intel vTune Amplifier
  - Performance tuning using hardware counters and metrics.
  - Very detailed information presented, but tries to summarise it.
- The documentation on the Intel website is really helpful.

Tools

ARM HPC Tools

- From ARM, but cross-platform. Used to be the Allinea Forge tools.
- DDT: Graphical debugger. Really good for parallel programs in MPI and OpenMP (and both).
- Map: Simple to use profiler. Can show thread activity, synchronisation and communication overheads.
- `https://www.arm.com/products/development-tools/hpc-tools/cross-platform/forge`

Tools

Cray PerfTools

- CrayPat
  - Performance analysis tool for Cray systems.
  - Gives overview of expensive routines and communication and synchronisation costs.
  - Also uses hardware counters for more details information.
- Reveal
  - Analyses loop dependencies and uses performance data to automatically try to insert OpenMP constructs.
  - Doesn't do a complete job, still need to check the constructs are correct.
  - Obviously can't refactor a loop to make is parallelisable.
  - `http://www.nersc.gov/users/software/performance-and-debugging-tools/craypat/reveal/`

Tools

- Can use the NVIDIA debuggers and profilers for OpenMP `target` constructs.
- Extrae and Paraver profiler and visualiser from Barcelona Supercomputing Center: `https://tools.bsc.es`.
- Score-P and Vampir: good for generating timelines of multi-threaded/process code: `https://www.alcf.anl.gov/files/Vampir.pdf`
- TAU profiler and program tracer: `https://www.cs.uoregon.edu/research/tau/tau.ppt`
- TotalView debugger: `https://computing.llnl.gov/tutorials/totalview/`

Resources

- OpenMP website: `https://www.openmp.org`
- cOMPunity: `http://www.compunity.org`
- Tim Mattson's YouTube tutorial: `https://youtu.be/nE-xN4Bf8XI`
- SC'08 tutorial from Tim Mattson and Larry Meadows: `https://openmp.org/mp-documents/omp-hands-on-SC08.pdf`
- From Lawrence Livermore National Lab: `https://computing.llnl.gov/tutorials/openMP/`
- SC'16 Tutorial from Tim Mattson and Alice Koniges: `https://www.nersc.gov/assets/Uploads/SC16-Programming-Irregular-Applications-with-OpenMP.pdf`
- OpenMPCon'15 Tutorial from Christian Terboven and Michael Klemm: `https://openmpcon.org/wp-content/uploads/openmpcon2015-michael-klemm-tasking.pdf`
- Patrick Atkinson and Simon McIntosh-Smith, *On the Performance of Parallel Tasking Runtimes for an Irregular Fast Multipole Method Application*, IWOMP 2017. `https://link.springer.com/chapter/10.1007/978-3-319-65578-9_7`.

OpenMP 5.0

Due to be released the week before SC'18.
Highlighted additions:

- `mutexinout` dependency.
- Task reductions.
- Scan reduction operations.
- Task affinity hints to say where tasks might execute based on data.
- `metadirective` and `declare variant` for selection of directive and function variants based on context.
- Memory space allocators and traits to support different kinds of memories.
- `loop` construct for compiler optimisation and parallelisation of loops; specifies iterations can execute in any order.

Plus many others…

Exercise

Asynchronous offload:

- Start with the `target` version of the 5-point stencil.
- Use the `nowait` clause to asynchronously offload part of the computation.
- Complete the remainder of the work on the host in parallel concurrently with the device.
- Ensure correct synchronisation with the tasking constructs.

Summary

- Covered a lot of content:
  - Shared memory parallelism with OpenMP.
  - Parallelising loops with OpenMP worksharing.
  - Code optimisations.
  - Performance analysis, including the Roofline model.
  - Vectorisation.
  - NUMA aware programming.
  - Hybrid MPI+OpenMP codes.
  - Programming a GPU with OpenMP.
  - Task based parallelism with OpenMP.
- Trying the concepts out yourself is the best way of learning a parallel programming model.
- OpenMP gives you the tools to write performance portable programs.
- The OpenMP standard, runtimes and compilers continue to develop.