HPC2N, UmU

The Task directive

Joachim Hein

LUNARC & Centre of Mathematics
Lund University

1

There is more than regular loops

- So far discussed: regular structures
  - Loops with known start and end
  - Fortran array constructs
- Many problems have irregular structures:
  - Recursive
  - Linked lists
  - Loops with unknown end (e.g. while)
  - Divide and Conquer
  - …
- Depending on the details they might still be **parallelisable**

2

Lund University

LUNARC

HPC2N, UmU

THE TASK CONSTRUCT

The task directive in Fortran

- creates "explicit task" from
  - code body
  - data environment at that point
- place inside parallel region
- execution:
  - now or later
  - encountering or other thread

```
!$omp task [clauses]
  code body
!$omp end task
```

The task directive in C

- creates “explicit task” from
  - code body
  - data environment at that point
- place inside parallel region
- execution:
  - now or later
  - encountering or other thread

```
#pragma omp task [clauses]
  {
    code body
  }
```


Allowed data sharing attributes for tasks

- `private`
  - data is private to the task
- `firstprivate`
  - data is private to the task
  - data initialised when task directive is encountered
- `shared`
  - data is shared – **only** way to return a result!
- `default`
  - Fortran: `shared | private | firstprivate | none`
  - C: `shared | none`

Data sharing without a default

- When no default is declared on a "task" directive:
- If `shared` by **all** implicit tasks in the current team:

  Variable is: `shared`
- Otherwise

  Variable is: `firstprivate`
- **My recommendation:** `default(none)`

Example for execution: Thread encountering the following

- Executes "code block 1"
- Creates a task for "code block 2"
- May
  - Execute the task for "code block 2"
  - Pick up another task
  - Continue with "code block 3"
- At some point:
  - Has to execute code block 3
- No control
  - Who executes code block 2
  - When code block 2 is finished

```
code block 1

!$omp task
  code block 2
!$omp end task

code block 3
```

Controlling when a task finishes

```
!$omp taskwait
```

- Ensures **child tasks** have completed
- Does not consider grand-children etc.

```
!$omp barrier
```

- Ensures all tasks in the innermost parallel region have finished
- Remark: Instead of waiting, thread can execute task generated elsewhere

Allowing suspension of the current task

- At a taskyield construct
  - Can suspend the current task to:
  - Execute a different task

Fortran:

```
!$omp taskyield
```

C:

```
#pragma omp taskyield
```

Using `taskgroup` to control tasks finishing (OpenMP 4.0)

- A taskgroup construct, defines a region with an implied task scheduling point at the end
- Current task suspended until **all** descendant tasks (incl. grand children etc.) completed

```
!$omp taskgroup
  do i=1, n
    !$omp task ...
       call processing(...)
    !$omp end task
  end do
!$omp end taskgroup
```

This waits for all task, incl. tasks generated in `processing`

11

Using `taskgroup` to control tasks finishing (OpenMP 4.0)

- A taskgroup construct, defines a region with an implied task scheduling point at the end
- Current task suspended until **all** descendant tasks (incl. grand children etc.) completed

```
#pragma omp taskgroup
  { for (int i=0; i<n; i++)
     {
      #pragma omp task ...
        {  processing(...);
        }
     }
  }
```

This waits for all task, incl. tasks generated in `processing`

12

Controlling `task` creation

- Creating a `task` encounters significant overhead
  - Requires significant work inside to pay off
- Use `if`-clause to control task creation

```
!$OMP task if(level .lt. 10) ...
    ...
!$OMP end task
```

- In case expression evaluates to `.false.`
  - Encountering thread executes code body directly (included task)

Final tasks

- A task can carry a `final` clause

```
!OMP task final( level .gt. 30) ...
   ...
!OMP end task
```

- If expression evaluates to `.true.`, all encountered tasks will be:
  - included
  - final

Mergeable tasks

- A task can be declared as mergeable

```
!$omp task mergable ...
```

```
#pragma omp task mergable ...
```

- In case of an undeferred or included task, the implementation may:
  - Use the data environment of the generating task (incl. internal control variables)
  - Use for optimisation, e.g. with `final`

Task scheduling points

- Threads may switch to different task at ***task scheduling point***
- Task scheduling points are:
  - Immediately after generation of explicit task
  - After point of completion of a task
  - At `taskwait`, `taskyield`
  - At `barrier` (explicit or implicit)
  - At the end of `taskgroup`
- Untied tasks (not in course) may switch at any point
  - Care with `critical`, `locks` etc.
  - E.g. task may switch out of critical region
    ➔deadlock!

Lund University

LUNARC

HPC2N, UmU

1st Case study

PARALLELISING A RECURSIVE ALGORITHM

Fibonacci numbers

- Mathematical series:
  $F_0 = 0$
  $F_1 = 1$
  $F_n = F_{n-1} + F_{n-2}$
- First numbers in series:
  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …
- Recursive program: not efficient!!!

Fortran code: Serial version

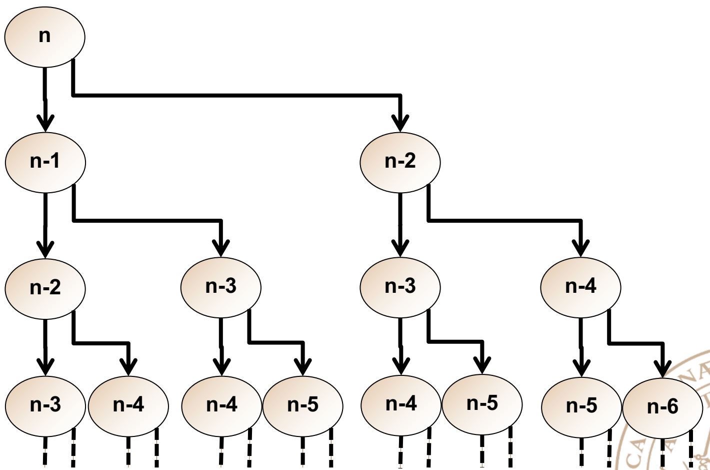
```
recursive function recursive_fib(in) result(fibnum)
    integer, intent(in) :: in
    integer(lint) :: fibnum, sub1, sub2
    if ( in .gt. 1) then

          sub1 = recursive_fib( in - 1 )


          sub2 = recursive_fib( in - 2 )


       fibnum = sub1 + sub2
    else
       fibnum = in
    endif
end function recursive_fib
```


Recursive Fibonacci



Fortran code: Serial version

```
recursive function recursive_fib(in) result(fibnum)
    integer, intent(in) :: in
    integer(lint) :: fibnum, sub1, sub2
    if ( in .gt. 1) then

          sub1 = recursive_fib( in - 1 )


          sub2 = recursive_fib( in - 2 )


       fibnum = sub1 + sub2
    else
       fibnum = in
    endif
end function recursive_fib
```


Fortran code: Towards a parallel version I

```
recursive function recursive_fib(in) result(fibnum)
    integer, intent(in) :: in
    integer(lint) :: fibnum, sub1, sub2
    If ( in .gt. 1) then
       !$OMP task shared(sub1) firstprivate(in)
          sub1 = recursive_fib( in - 1 )
       !$OMP end task

          sub2 = recursive_fib( in - 2 )


       fibnum = sub1 + sub2
    else
       fibnum = in
    endif
end function recursive_fib
```

sub1 shared
declared inside function

in firstprivate
initilised at task creation

Fortran code: Towards a parallel version II

```
recursive function recursive_fib(in) result(fibnum)
    integer, intent(in) :: in
    integer(lint) :: fibnum, sub1, sub2
    If ( in .gt. 1) then
       !$OMP task shared(sub1) firstprivate(in)
          sub1 = recursive_fib( in - 1 )
       !$OMP end task
       !$OMP task shared(sub2) firstprivate(in)
          sub2 = recursive_fib( in - 2 )
       !$OMP end task

       fibnum = sub1 + sub2
    else
       fibnum = in
    endif
end function recursive_fib
```

place task at 2nd call

Problem:
Need to have sub1 & sub2

Fortran code: Parallel version

```
recursive function recursive_fib(in) result(fibnum)
    integer, intent(in) :: in
    integer(lint) :: fibnum, sub1, sub2
    If ( in .gt. 1) then
       !$OMP task shared(sub1) firstprivate(in)
          sub1 = recursive_fib( in - 1 )
       !$OMP end task
       !$OMP task shared(sub2) firstprivate(in)
          sub2 = recursive_fib( in - 2 )
       !$OMP end task
       !$OMP taskwait
       fibnum = sub1 + sub2
    else
       fibnum = in
    endif
end function recursive_fib
```

Solved: taskwait
Waits for the 2 tasks above
Recursion takes care of grand-children

How to call?
The orginal serial code

```
Program fibonacci
  !$ use omp_lib
  integer, parameter :: lint = selected_int_kind(10)
  integer(lint) :: fibres
  integer :: input
  read (*,*) input


      fibres = recursive_fib(input)


  print *, "Fibonacci number", input," is:", fibres
End program fibonacci
```


How to call?
Need to start a parallel region

```
Program fibonacci
  !$ use omp_lib
  integer, parameter :: lint = selected_int_kind(10)
  integer(lint) :: fibres
  integer :: input
  read (*,*) input
  !$OMP parallel shared(input, fibres) default(none)

      fibres = recursive_fib(input)

  !$OMP end parallel
  print *, "Fibonacci number", input," is:", fibres
End program fibonacci
```

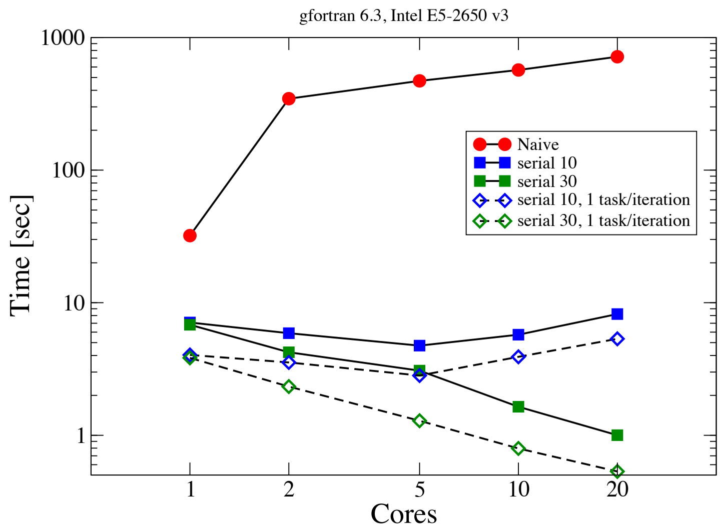
Problem:
Each thread starts
Fibonacci calculation :o

How to call?

```
Program fibonacci
  !$ use omp_lib
  integer, parameter :: lint = selected_int_kind(10)
  integer(lint) :: fibres
  integer :: input
  read (*,*) input
  !$OMP parallel shared(input, fibres) default(none)
    !$OMP single
       fibres = recursive_fib(input)
    !$OMP end single
  !$OMP end parallel
  print *, "Fibonacci number", input," is:", fibres
End program fibonacci
```

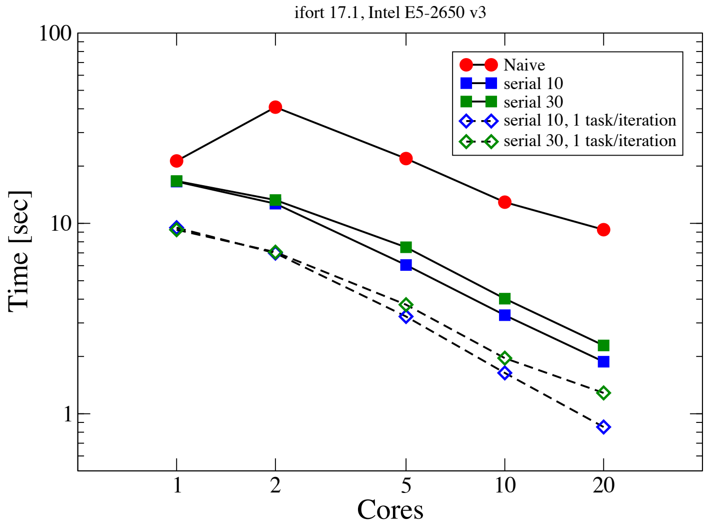

27

Performance for Fibonacci number 40



29

Performance for Fibonacci number 40



Discussion of Fibonacci performance

- Naïve implementation
  - 2 tasks per iteration
  - poor performance
- Using if-clause
  - No tasks created for low input value: helps
- Start only 1 task/iteration
  - Helps more
- Too little work per task
- Limit number of tasks

- Hardware:
  - 2 socket/server
  - Intel E5-2650 v3
  - 10 cores per Processor
- gfortran
  - Version 6.3
  - Thread binding
- Intel ifort
  - Version17.1
  - Thread binding

Lund University

LUNARC

HPC2N, UmU

2nd Casestudy

SELF-REFINING RECURSIVE INTEGRATOR

Mesh refinement

- Codes endloying irregular grids
- Dynamic grid refinement / coarsening allows effciency
  - Example: Fluiddynamic
    - refine grid were eddy develops
    - coarsen when eddy vanishes
- Case study: self refining integrator for 1D function

Basic algorithm

- Evaluate function at 5 regular space points in interval
- Estimate integral:
  - Polygon using all 5 points
  - Polygon using only 3 points (first, centre, last)
- Check difference between the two integrals
  - Compare to threshold times interval length
- If accurate add contribution to accumulation variable
- If not accurate:
  - Split interval into two pieces
  - Run integrator on both pieces (Recursion)

Implementation details: Parallel region

- Use shared variable `accumulator` for result
  - Declared as module variable
- Start recursive integrator from `single`
- The implied barrier ensures tasks are finished here
- Recursive subroutine: `rec_eval_shared_update`

```
accumulator = 0.0D0
!$OMP parallel default(none) &
   !$OMP  shared(accumulator) &
   !$OMP  shared(startv, stopv, &
   !$OMP  unit_err, gen_num)

   !$OMP single
    call rec_eval_shared_update( &
      startv, stopv, unit_err, gen_num)
   !$OMP end single
!$OMP end parallel
```

Implementation details: Task startup (in: rec_eval_shared_update)

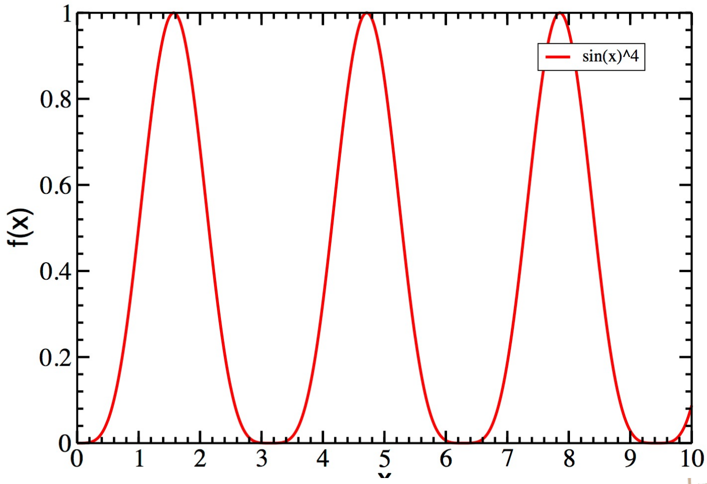
```
!$OMP task shared(accumulator) firstprivate(my_start, my_stop)  &
   !$OMP   default(none)  firstprivate(my_gen,u_err) &
   !$OMP    if(task_start)
   call rec_eval_shared_update( &
         my_start, 0.5_dpr * (my_start + my_stop), u_err, my_gen)
!$OMP end task

!$OMP task shared(accumulator) firstprivate(my_start, my_stop)  &
   !$OMP    default(none)  firstprivate(my_gen, u_err) &
   !$OMP    if(task_start)
   call rec_eval_shared_update( &
        0.5_dpr * (my_start + my_stop), my_stop, u_err, my_gen)
!$OMP end task
```

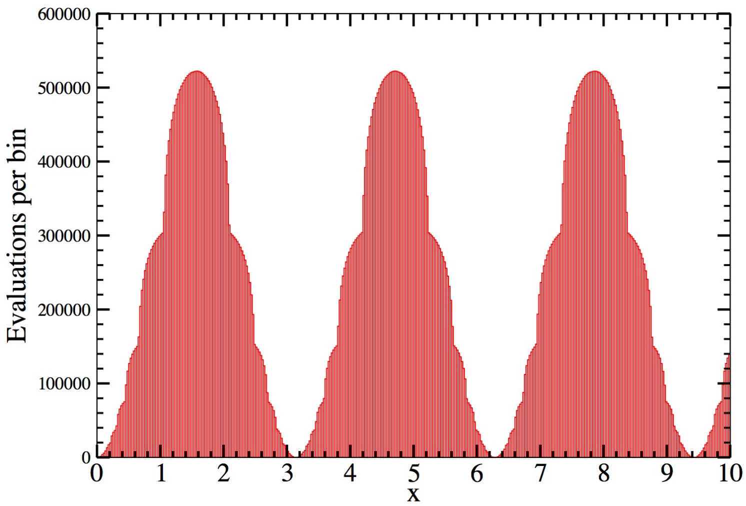

Implementation details: Result accumulation

- Declare `shared` variable (`global` to avoid issue in ifort 19u5): `atomic` update when accurate
- Declare `threadprivate` variables
  - Thread executing task achieving accuracy adds to his `threadprivate` copy
  - After `barrier` (implied in `end single`) atomic update of `threadprivate` data into `shared` variable
- **Remarks:**
  - Carefull: `threadprivate` and task scheduling points
    - Value can be changed after scheduling point
  - `threadprivate` isn't private to the task
  - OpenMP 5.0 has reduction constructs for tasks

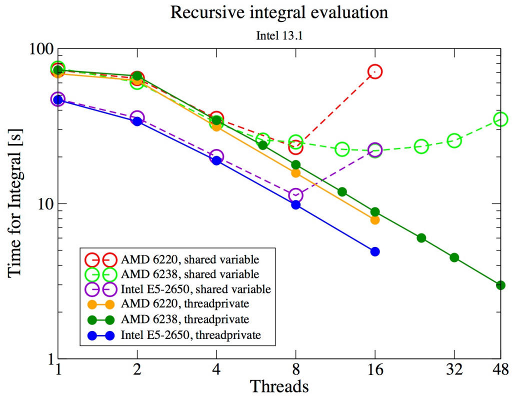
Testfunction: $\sin^2(10000\,x)\,\sin^4(x)$



Where does it sample

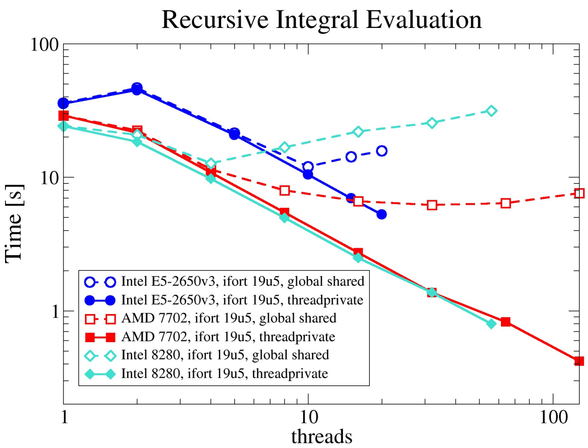


Performance results



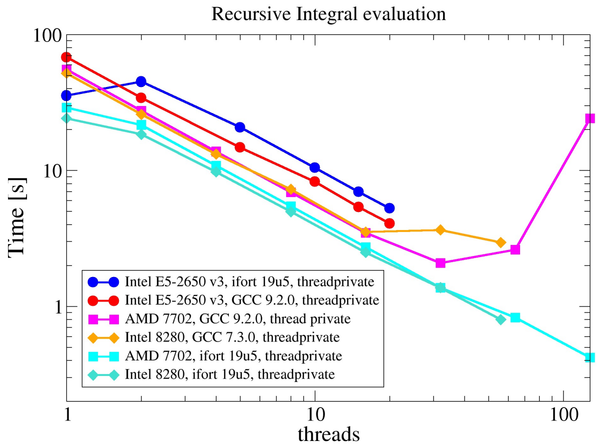
- Task started every 5th generation
- Poor results with atomic updates (there are millions)
- `threadprivate` accumulation satisfactory result

Performance results



- Task started every 5th generation
- Poor results with atomic updates (there are millions)
- `threadprivate` accumulation satisfactory result
- Efficient utilization of up to 128 cores

Comparing compilers



- GCC: inferior scalabilty beyond 20 cores

42

Task dependencies (OpenMP 4.0)

- One can declare dependencies

```
!$omp task depend (type : list)
```

```
#pragma omp task depend (type : list)
```

- 3 dependcency types:
  - `in`: task depends on all previous siblings with an `out` or `inout` dependency on one or more of the list items
  - `out, inout`: task depends on all previous siblings with an `in`, `out` or `inout` dependency on one or more of the list items
- The list are variables, that my include array sections

43

Example for task dependency

- Code snippet:

```
#pragma omp task depend (out: a)
    task_function_1( &a );
#pragma omp task depend (in: a)
    task_function_2( a );
#pragma omp task depend (in: a)
    task_function_3( a );
```

- Wait for `task_function_1` to be finished
- Can execute `task_function_2` and `task_function_3` in any order on any thread

Tasks and reductions (OpenMP 5.0)

- A `taskgroup` around the reduction area is required
- Reduction variable included in a `task_reduction` clause of the `taskgroup`:

```
task_reduction( operator : variable_list )
```

- Reduction variable include in a `in_reduction` clause of the task construct

```
in_reduction( operator : variable_list )
```

Example in C

```
double sum = 0.0;
#pragma omp single
{
#pragma omp taskgroup task_reduction(+: sum)
  {
    for (int i = 0; i < N; i++)
#pragma omp task in_reduction(+: sum)
    {
      sum += evalution(i);
    }
  }
}
```


Taskloop (OpenMP 4.5)

- The `taskloop` construct distributes loops on tasks
  - Similar to the loop construct
- By default the `taskloop` implies a `taskgroup`

Basic syntax

```
!$OMP taskloop default(none) shared(…) private(…)
  do i = 1, N
     ...
  enddo
```

```
#pragma omp taskloop default(none) shared(…) private(…)
   for (i=0; i<N; i++)
     {
       ...
     }
```


48

Clauses for taskloop

- Clauses introduced before
  `if( scalar-expr ), shared, private, firstprivate, lastprivate, default, collapse, final( scalar-expr )`
- A `reduction` clause was introduced for the `taskloop` was introduced in OpenMP 5.0
- An `in_reduction` clause is available if the `taskloop` is embedded in a `taskgroup` with a `task_reduction`
- The clause `nogroup` removes the implied taskgroup
- There is a `taskloop simd` construct

49

Controlling the number of task created

- Use `grainsize` or `num_tasks` to control task number
  - Only one allowed
- `grainsize` controls number of loop iterations per task
  - each tasks gets between `grainsize` and `2*grainsize` iterations
- `num_tasks` specifies the number of tasks created
- Additional restrictions from iteration count

Summary

- Explained the task construct
  - Task scheduling
  - Task completion
- Discussed performance aspects in case studies
  - Need to control the number
  - Decent amount of work per task