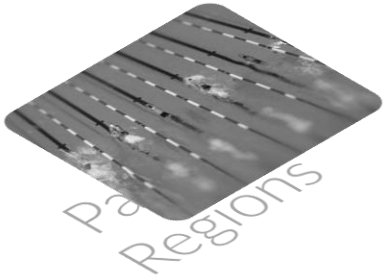
Advanced OpenMP
Tasks

Luca Tornatore - I.N.A.F.

"**Foundation of HPC" course**

DATA SCIENCE &
SCIENTIFIC COMPUTING
2020-2021 @ Università di Trieste

OpenMP Outline



Advanced Parallelism

Advanced Parallelism Outline

Advanced Parallelism in OpenMP

- Sections
- Tasks

This lecture

Task abstraction

represents any contained sequence of instructions in the code, logically defining a finite work/function/assignment

Asynchronous + Interleaved execution + dependencies

Dependency graph among task.
Must be acyclic.
Related to data dependencies.

Data abstraction

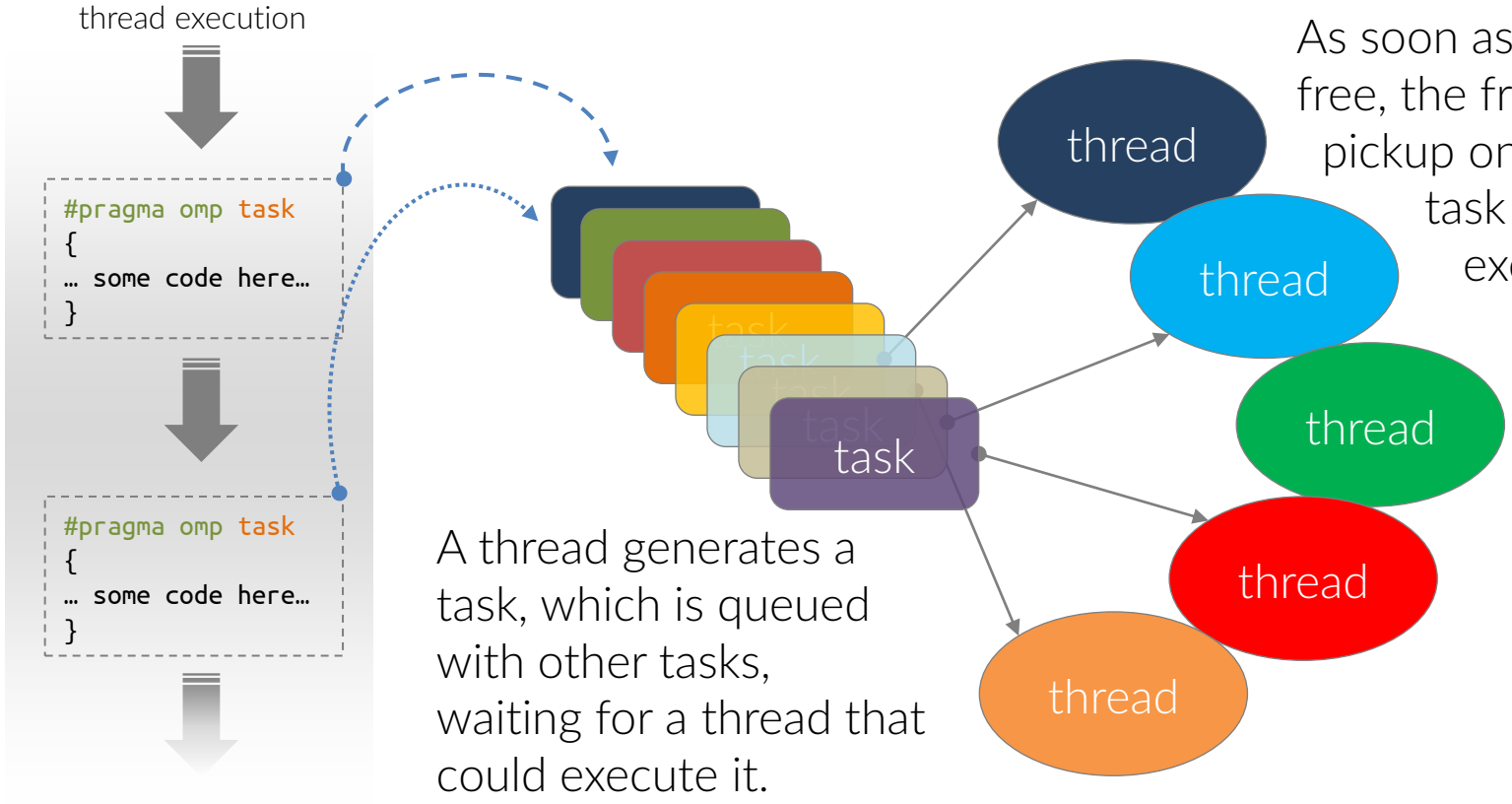
represents any piece of logically uniform “information”, that may be accessed by several threads; out-of-order access needs to be managed

Concurrent access

As we have seen in the previous example (`02_sections_nested_irrecular.c`), it is *sometimes* possible to parallelize a workflow which is irregular or runtime-dependent using OpenMP sections.
However, often the solution is quite ugly and convoluted.

Since version 3.0, OpenMP **tasks** offer a new elegant construct designed for this class of problems: **irregular and run-time dependent execution flow**.

What happens under the hood is that OpenMP creates a "bunch of work" along with the data and the local variables it needs, and *schedules* it for execution at some point in the future.
Then, under the hood, a queuing system orchestrates the assignment of each task to the available threads.


thread execution
#pragma omp task
{
… some code here…
}
#pragma omp task
{
… some code here…
}
task
thread
thread
thread
thread
thread
As soon as they are free, the free threads pickup one queued task each, and execute it
A thread generates a task, which is queued with other tasks, waiting for a thread that could execute it.

As almost everything else in OpenMP, a task must be generated *inside* a parallel region and it is linked to a specific block of code.
If its execution is not properly "protected", it might be executed by *more* than one thread (i.e. by all threads that encounter the task definition), which is not in general what we want.

To guarantee that each task is executed only once, every task must be generated within a `single` or `master` region.

The `single` region is preferable because of its implied barrier that makes all tasks to be completed before passing. In case you use a `master` region, pay attention to the execution flow.
Moreover, the `master` has often the heavier burden so it's best to user a `single` region, possibly with the `nowait` clause.

Let's start from a very basic example

The `single` region within which the tasks are created

Tasks generation by the thread that entered in the `single` region

All the other threads are waiting here, at the implied barrier at the end of the `single` region

```
#pragma omp parallel
 {

   #pragma omp single
   {
     printf( " »Yuk yuk, here is thread %d from "
             "within single region\n", omp_get_thread_num() );

     #pragma omp task
     {
       printf( "\tHi, here is thread %d "
               "running task A\n", omp_get_thread_num() );
     }

     #pragma omp task
     {
       printf( "\tHi, here is thread %d "
               "running task B\n", omp_get_thread_num() );
     }

   }

   printf(" :Hi, here is thread %d at the end "
          "of the single region, stuck waiting "
          "all the others\n", omp_get_thread_num() );
 }
```

examples_tasks/
00_simple.c

```
tasks:> gcc -fopenmp -o 00_simple 00_simple.c
tasks:> ./00_simple
 »Yuk yuk, here is thread 5 from within the single region
        Hi, here is thread 2 running task A
        Hi, here is thread 1 running task B
 :Hi, here is thread 5 at the end of the single region, stuck waiting all the others
 :Hi, here is thread 1 at the end of the single region, stuck waiting all the others
 :Hi, here is thread 0 at the end of the single region, stuck waiting all the others
 :Hi, here is thread 6 at the end of the single region, stuck waiting all the others
 :Hi, here is thread 4 at the end of the single region, stuck waiting all the others
 :Hi, here is thread 3 at the end of the single region, stuck waiting all the others
 :Hi, here is thread 7 at the end of the single region, stuck waiting all the others
 :Hi, here is thread 2 at the end of the single region, stuck waiting all the others
tasks:> ./00_simple
 »Yuk yuk, here is thread 1 from within the single region
        Hi, here is thread 6 running task A
        Hi, here is thread 2 running task B
 :Hi, here is thread 6 at the end of the single region, stuck waiting all the others
 :Hi, here is thread 1 at the end of the single region, stuck waiting all the others
 :Hi, here is thread 5 at the end of the single region, stuck waiting all the others
 :Hi, here is thread 0 at the end of the single region, stuck waiting all the others
 :Hi, here is thread 3 at the end of the single region, stuck waiting all the others
 :Hi, here is thread 7 at the end of the single region, stuck waiting all the others
 :Hi, here is thread 4 at the end of the single region, stuck waiting all the others
 :Hi, here is thread 2 at the end of the single region, stuck waiting all the others
```

If you run it several times, at each shot you find that a random thread enters the region, while the others are waiting for the region to end,

Some of them (even one, potentially) will pick up the tasks generated in the single region.

After the conclusion of all the tasks, everybody is let go

Let's add a detail..

All the other threads skipe the the `single` region, and continue the execution at the next barrier (in this case, implicit) where they will receive a task.

```
#pragma omp parallel
{
  #pragma omp single nowait
  {
    printf( " »Yuk yuk, here is thread %d from "
            "within the single region\n", omp_get_thread_num() );

    #pragma omp task
    {
      printf( "\tHi, here is thread %d "
              "running task A\n", omp_get_thread_num() );
    }

   #pragma omp task
    {
      printf( "\tHi, here is thread %d "
              "running task B\n", omp_get_thread_num() );
    }

  }

  printf(" :Hi, here is thread %d at the end "
         "of the single region, stuck waiting "
         "all the others\n", omp_get_thread_num() );
}
```

examples_tasks/
00_simple_nowait.c

```
tasks:> gcc -fopenmp -o 00_simple_nowait 00_simple_nowait.c
tasks:> ./00_simple_nowait
 :Hi, here is thread 6 at the end of the single region, stuck waiting all the others
 »Yuk yuk, here is thread 7 from within the single region
 :Hi, here is thread 1 at the end of the single region, stuck waiting all the others
       Hi, here is thread 1 running task A
 :Hi, here is thread 0 at the end of the single region, stuck waiting all the others
 :Hi, here is thread 4 at the end of the single region, stuck waiting all the others
 :Hi, here is thread 3 at the end of the single region, stuck waiting all the others
 :Hi, here is thread 2 at the end of the single region, stuck waiting all the others
 :Hi, here is thread 7 at the end of the single region, stuck waiting all the others
       Hi, here is thread 6 running task B
 :Hi, here is thread 5 at the end of the single region, stuck waiting all the others
tasks:> ./00_simple_nowait
 :Hi, here is thread 0 at the end of the single region, stuck waiting all the others
 :Hi, here is thread 7 at the end of the single region, stuck waiting all the others
 :Hi, here is thread 3 at the end of the single region, stuck waiting all the others
 »Yuk yuk, here is thread 1 from within the single region
 :Hi, here is thread 6 at the end of the single region, stuck waiting all the others
 :Hi, here is thread 4 at the end of the single region, stuck waiting all the others
 :Hi, here is thread 2 at the end of the single region, stuck waiting all the others
 :Hi, here is thread 5 at the end of the single region, stuck waiting all the others
       Hi, here is thread 3 running task A
 :Hi, here is thread 1 at the end of the single region, stuck waiting all the others
       Hi, here is thread 0 running task B
```

Now the threads are free to flow beyond the single region, up to the next barrier (either implied or explicit).

The order of execution of the tasks is in general not guaranteed.

It is only guaranteed that the task will be executed at some special and well-defined points in the code.

Let's add one more detail..

This directive requires that all the children task of the current task must be completed.
It binds to the current task region, the set of binding thread of the `taskwait` region is the current team.

When a thread encounters a taskwait construct, the current task is suspended until all child tasks that it generated *before* the taskwait region complete execution.

```
#pragma omp parallel
 {

   #pragma omp single nowait
   {
     int me = omp_get_thread_num();
     printf( " »Yuk yuk, here is thread %d from "
             "within the single region\n", me );

     #pragma omp task
     {
       printf( "\tHi, here is thread %d "
               "running task A\n", omp_get_thread_num() );
     }

    #pragma omp task
     {
       printf( "\tHi, here is thread %d "
               "running task B\n", omp_get_thread_num() );
     }

   #pragma omp taskwait
     printf(" «Yuk yuk, it is still me, thread %d "
            "inside single region after all tasks ended\n", me);

   }

   printf(" :Hi, here is thread %d at the end "
          "of the single region, stuck waiting "
          "all the others\n", omp_get_thread_num() );
 }
```

examples_tasks/
00_simple_taskwait.c

Let's add one more detail..

This directive requires that all the children task of the current task must b
It
se
re

W
a
is
g
c

A tricky point:

can you explain the behaviour of the code `examples_tasks/00_simple_taskwait_a.c`

Does the additional `taskwait` directive added after the `single` region affect the expected behaviour ?

```
#pragma omp parallel
 {

   #pragma omp single nowait
   {
     int me = omp_get_thread_num();
     printf( " »Yuk yuk, here is thread %d from "
             "within the single region\n", me );

     #pragma omp task
     {
       printf( "\tHi, here is thread %d "
               "running task A\n", omp_get_thread_num() );
     }

    #pragma omp task
     {
       printf( "\tHi, here is thread %d "
               "running task B\n", omp_get_thread_num() );
     }

   #pragma omp taskwait
     printf(" «Yuk yuk, it is still me, thread %d "
            "inside single region after all tasks ended\n", me);

   }

   printf(" :Hi, here is thread %d at the end "
          "of the single region, stuck waiting "
          "all the others\n", omp_get_thread_num() );
 }
```

examples_tasks/
00_simple_taskwait.c

examples_tasks/
03_*.c

A classical example: traversing a linked list

```
#pragma parallel region
{
   …;
   #pragma omp single nowait
   {
      while( !end_of_list(node) ) {
        if( node_is_to_be_processesed(node) )
           #pragma omp task
           process_node ( node );
        node = next_node( node );
   }
   …;
}
```

A task is generated for each node that must be processed

The calling thread continues traversing the linked list

Due to the `nowait` clause, all the threads skip the implied barrier at the end of the `single` region and wait here for being assigned a task

A second key point to account for when dealing with the asynchronous execution is the *data* environment.
A task is a confined code section that performs some operations on a data set, that is referred at the moment of the task creation.

You are in charge of ensuring that that reference will still be valid *at the moment of execution*, which is somewhere in the future.

```
#pragma omp task shared(result) untied
{
  double myresult    = 0;
  for( int ii = first; ii < last; ii++)
    myresult += heavy_work_0(array[ii]);
  #pragma omp atomic
  result += myresult;
}
```

Both first and last, which are two shared variables, are key variables for the task execution.

What if they are keep changing?
At the moment of execution, their value could be different than at the moment of task creation, and then the processing would be totally different than the original intention.

A second key point to account for when dealing with the asynchronous execution is the *data* environment.
A task is a confined code section that performs some operations on a data set, that is referred at the moment of the task creation.

You are in charge of ensuring that that reference will still be valid *at the moment of execution*, which is somewhere in the future.

The values of variables that are susceptible to change and that enter in the execution of the task must be protected to ensure the correctness of the task itself.

With the `firstprivate` clause, we are creating private local variables that will be referred to at the moment of the execution and will still have the correct value.

```
#pragma omp task firstprivate(first, last) shared(result) untied
{
  double myresult    = 0;
  for( int ii = first; ii < last; ii++)
    myresult += heavy_work_0(array[ii]);
  #pragma omp atomic
  result += myresult;
}
```

A second key point to account for when dealing with the asynchronous execution is the *data* environment.
A task is a confined code section that performs some operations on a data set, that is referred at the moment of the task creation.

You are in charge of ensuring that that reference will still be valid *at the moment of execution*, which is somewhere in the future.

With the `untied` clause, you are signalling that this task – if ever suspended – can be resumed by *any* free thread. The default is the opposite, a task to be `tied` to the thread that initially starts it.
If untied, you must take care of the data environment, of course: for instance, no `threadprivate` variables can be used, nor the thread number, and so on.

```
#pragma omp task firstprivate(first, last) shared(result) untied
{
  double myresult    = 0;
  for( int ii = first; ii < last; ii++)
    myresult += heavy_work_0(array[ii]);
  #pragma omp atomic
  result += myresult;
}
```

A third key point to catch with asynchronous execution, is about the *timing*, i.e. when a task is executed and how to synchronize them.

At the moment of creation, a task may be *deferred* or not, i.e. its execution may be scheduled for the future or immediately taken while the task region that has generated it is frozen.

There are some constructs that enforce synchronization:

barrier	Implicit or explicit barrier
taskwait	Wait on the completion of all child tasks of the current task
taskgroup	Wait on the completion of all child tasks of the current task **and** of their descendant

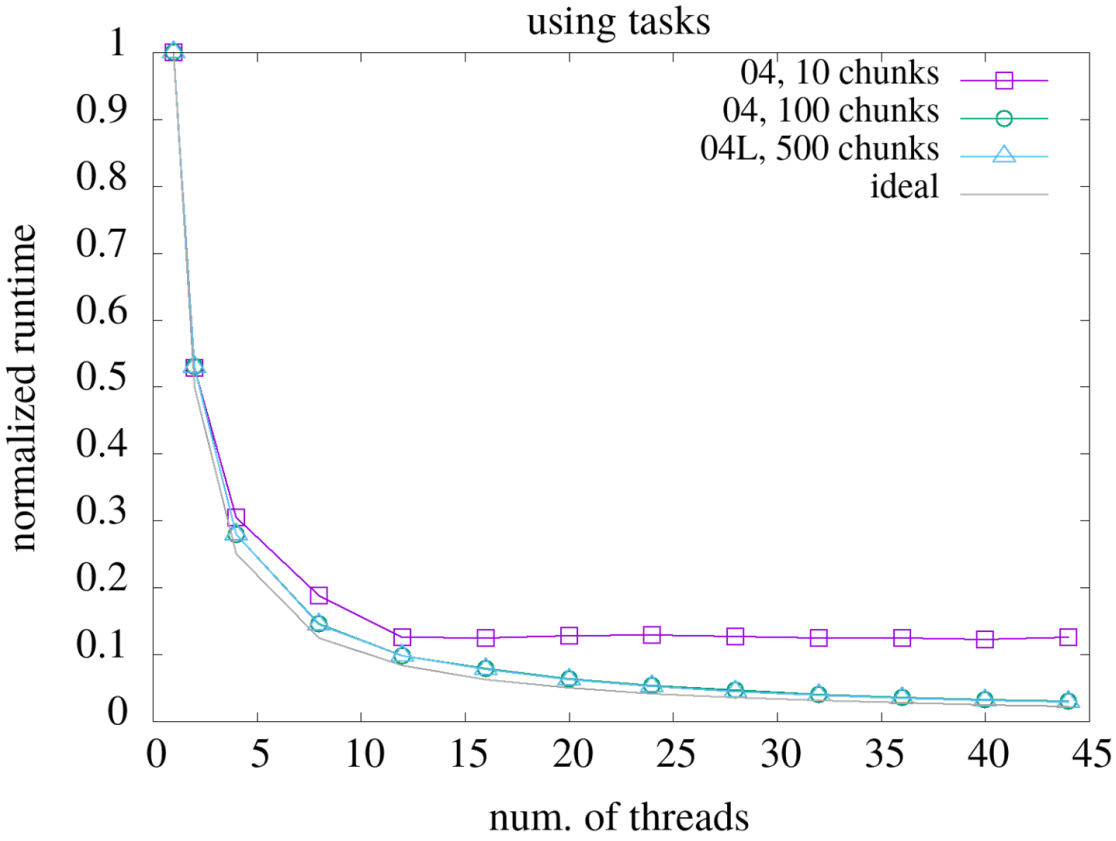
```
#pragma omp parallel shared(result)
  {
    double result1, result2, result3;

    #pragma omp single nowait
    {

      #pragma omp task  shared(result1)
      {
        double myresult = 0;
        for( int jj = 0; jj < N; jj++ )
          myresult += heavy_work_0( array[jj] );
       #pragma omp atomic update
        result1 += myresult;
      }

      #pragma omp  task shared(result2)
      {
        double myresult = 0;
        for( int jj = 0; jj < N; jj++ )
          myresult += heavy_work_1( array[jj] );
       #pragma omp atomic update
        result2 += myresult;
      }

      #pragma omp task  shared(result3)
      {
        double myresult = 0;
        for( int jj = 0; jj < N; jj++ )
          myresult += heavy_work_2(array[jj] );
       #pragma omp atomic update
        result3 += myresult;
      }

    }

    #pragma omp taskwait

    #pragma omp atomic update
    result += result1;
    #pragma omp atomic update
    result += result2;
    #pragma omp atomic update
    result += result3;
  }
```

- You need the tasks to be completed *before* to arrive at this point where the final results are accumulated.
- Due to the `nowait` clause, the implied barrier at the end of the `single` region is no respected and the threads are flowing freely beyond that region.
- Without the `taskwait` directive, the threads that are not in the `single` region would execute the updates of `result` with meaningless data.

`parallel_tasks/03_tasks_wrong.c` → `parallel_tasks/04_tasks.c`

finer implementation

Tasks performance



Tasks priorities

Even if you want your tasks to run concurrently, sometimes it is advisable that some tasks run earlier than others.
For instance, it may be good that the tasks that are receiving data have an higher *priority* than the tasks that post-process them.

You can suggest this to the OpenMP scheduler by using the `priority(p)` clause.
The higher the value of `p`, the sooner the corresponding task will be scheduled for execution.

```
#pragma omp parallel
#pragma #omp single
{
   …;
  #pragma omp task priority(100)
   read_data(…);
  #pragma omp task priority(50)
   process_and_save_data(…);
  #pragma omp task priority(10)
   postprocess_and_send_data(…);
}
```

Tasks dependencies

Often, there are **dependencies** among different tasks:

a given tasks may have to use the results of another one, or in any case to wait for its operations to terminate

Often, there are **dependencies** among different tasks:
a given tasks may have to use the results of another one, or in any case to wait for its operations to terminate

RaW
Read after Write
"flow dependence"

The task 1 read a memory region written by task 0

```
#pragma omp task depend(OUT:the_answer)
 function_wise( *the_answer );

#pragma omp task depend(IN:the_answer)
 function_curious( *the_answer );
```

Tasks dependencies

Often, there are **dependencies** among different tasks:
a given tasks may have to use the results of another one, or in any case to wait for its operations to terminate

RaW
Read after Write

The task 1 read a memory region written by task 0

```
#pragma omp task depend(OUT:the_answer)
 function_wise( *the_answer );
#pragma omp task depend(IN:the_answer)
 function_curious( *the_answer );
```

WaR
Write after Read
"anti-dependence"

The task 0 reads a memory region written by task 1

```
#pragma omp task depend(IN:the_question)
 function_sage( *the_question );
#pragma omp task depend(OUT:the_question)
 function_curious( *the_question );
```

Tasks dependencies

Often, there are **dependencies** among different tasks:
a given tasks may have to use the results of another one, or in any case to wait for its operations to terminate

RaW
Read after Write

The task 1 read a memory region written by task 0

```
#pragma omp task depend(OUT:the_answer)
 function_wise( *the_answer );
#pragma omp task depend(IN:the_answer)
 function_curious( *the_answer );
```

WaR
Write after Read

The task 0 reads a memory region written by task 1

```
#pragma omp task depend(IN:the_question)
 function_sage( *the_question );
#pragma omp task depend(OUT:the_question)
 function_curious( *the_question );
```

WaW
Write after Write
"output depend."

Both task 0 and task 1 write the same memory region;

```
#pragma omp task depend(OUT:the_question)
 function_sage( *the_question );
#pragma omp task depend(OUT:the_question)
 function_curious( *the_question );
```

Often, there are **dependencies** among different tasks:
a given tasks may have to use the results of another one, or in any case to wait for its operations to terminate

RaW
Read after Write

The task 1 read a memory region written by task 0

```
#pragma omp task depend(OUT:the_answer)
 function_wise( *the_answer );
#pragma omp task depend(IN:the_answer)
 function_curious( *the_answer );
```

WaR
Write after Read

The task 0 reads a memory region written by task 1

```
#pragma omp task depend(IN:the_question)
 function_sage( *the_question );
#pragma omp task depend(OUT:the_question)
 function_curious( *the_question );
```

WaW
Write after Write

Both task 0 and task 1 write the same memory region;

```
#pragma omp task depend(OUT:the_question)
 function_sage( *the_question );
#pragma omp task depend(OUT:the_question)
 function_curious( *the_question );
```

RaR
Read after Read

Both task 0 and task 1 read the same memory region; no particular order is needed

```
#pragma omp task depend(IN:the_question)
 function_sage( *the_question );
#pragma omp task depend(OUT:the_question)
 function_curious( *the_question );
```

Tasks dependencies

dependency types:

- **`IN`**: the task will be dependent on a previously generated task if that task has an `out`, `inout` or `mutexinoutset` dependence on the same memory region.
- **`OUT, INOUT`** : the task will be dependent on a previously generated task if that task has an `in`, `out`, `inout` or `mutexinoutset` dependence on the same memory region.
- **`MUTEXINOUTSET`**: the task will be dependent on a previously generated task if that task has an `in`, `out`, `inout` or dependence on the same memory region;
  it will be *mutually exclusive* with another `mutexinoutset` sibling task.

Tasks dependencies

Flow-dependence: will write "x=2"

```
int x = 1;
…;
#pragma omp task shared(x) depend(out:x)
 x = 2;

#pragma omp task depend(in:x)
 printf("x = %d\n", x);
```

Anti-dependence: will write "x=1"

```
int x = 1;
…;
#pragma omp task shared(x) depend(in:x)
 printf("x = %d\n", x);

#pragma omp task shared(x) depend(out:x)
 x = 2;
```

output-dependence: will write "x=3", the dep is enforced by the generation order

```
#pragma omp single
{
  #pragma omp task shared(x) depend(out:x)
   x = 2;
  #pragma omp task shared(x) depend(out:x)
   x = 3;
  #pragma omp taskwait
   printf("x = %d\n", x);
}
```

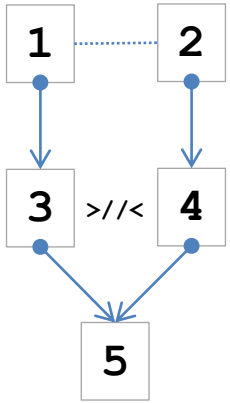
No dependence: output is variable, the printing tasks are independent off each other

```
#pragma omp single
{
  #pragma omp task shared(x) depend(out:x)
   x = 2;
  #pragma omp task shared(x) depend(in:x)
   printf("x + 1 = %d\n", x+1);
  #pragma omp task shared(x) depend(in:x)
   printf("x + 2 = %d\n", x+2);
}
```

examples taken from openmp.org

Mutually exclusive dependency

```
…;
#pragma omp task shared(x) depend(out:x)
 x = get_x();                          // task 1

#pragma omp task shared(y) depend(out:y)
 y = get_y();                          // task 2

#pragma omp task shared(z,x) depend(in:x) depend(mutexinoutset:z)
 z *= x;                               // task 3

#pragma omp task shared(z,y) depend(in:y) depend(mutexinoutset:z)
 z *= y;                               // task 4

#pragma omp task shared(a,z) depend(in:z) depend(out_a)
 a = z;                                // task 5
```



mutually independent

mutually exclusive

examples taken from openmp.org

You can enforce to wait for some particular dependence

```
int x = 1, y = 2;

// task 1
#pragma omp task shared(x) depend(inout:x)
 x += 1;

// task 2
#pragma omp task shared(y)
 y *= 2;

#pragma omp taskwait depend(in:x)   // wait for task 1 only

printf("x = %d\n", x);              // this print is safe
printf("y = %d\n", y);              // this print is unsafe

#pragma omp taskwait

printf("y = %d\n", y);              // *now* this print is
                                    // safe too
```

examples taken from openmp.org

You can enforce to wait for some particular dependence

At this point, the `in:x` dependence is fullfilled and the generating thread can prosecute to the `printf` instructions, without waiting for the `task 2` which is not modifying `x`. What would you modify to make both prints safe and eliminate the last taskwait ?

```
int x = 1, y = 2;

// task 1
#pragma omp task shared(x) depend(inout:x)
 x += 1;

// task 2
#pragma omp task shared(x, y) depend(in:x) depend(inout:y)
 y *= x;

#pragma omp taskwait depend(in:x)   // wait for task 1 only

printf("x = %d\n", x);              // this print is safe
printf("y = %d\n", y);              // this print is unsafe

#pragma omp taskwait

printf("y = %d\n", y);              // *now* this print is
                                    // safe too
```

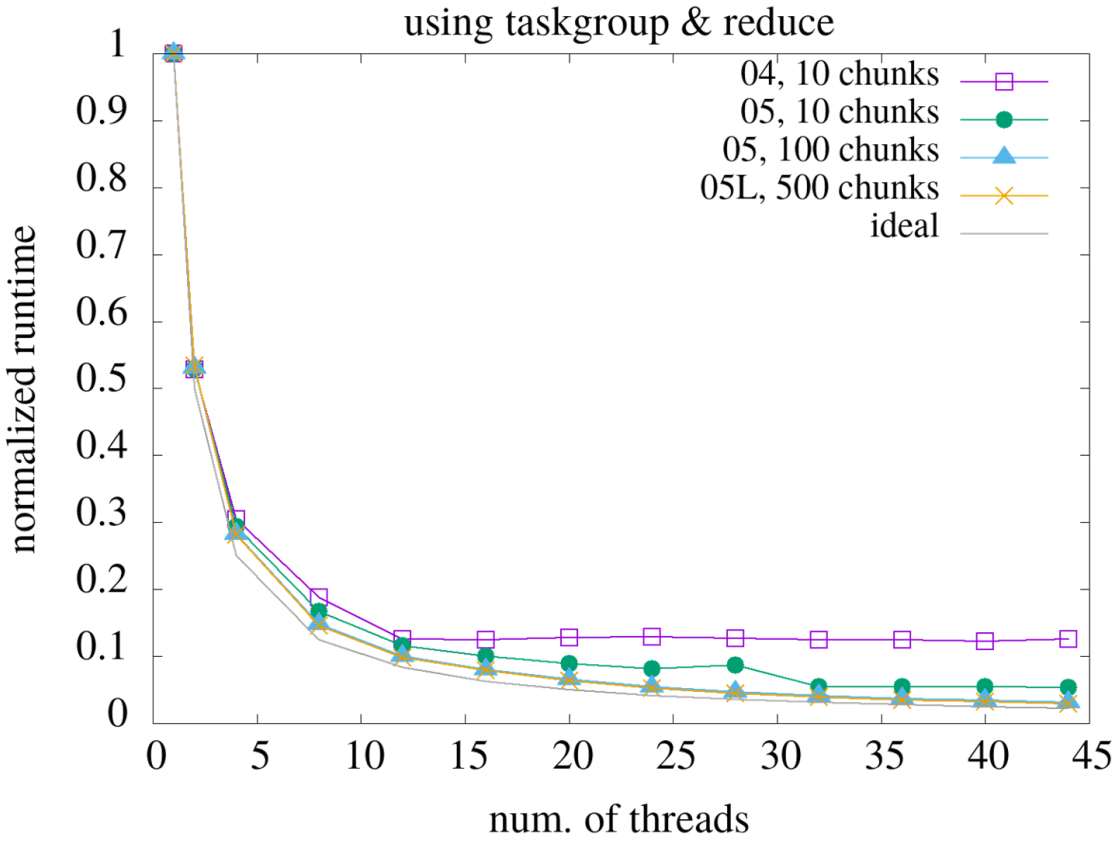
examples taken from openmp.org

```
#pragma omp parallel proc_bind(close)
  {
    #pragma omp single nowait
    {
      #pragma omp taskgroup task_reduction(+:result)
      {
        int idx   = 0;
        int first = 0;
        int last  = chunk;

        while( first < N )
          {
            last = (last >= N)?N:last;
            for( int kk = first; kk < last; kk++, idx++ )
              array[idx] = min_value + lrand48() % max_value;

            #pragma omp task in_reduction(+:result) firstprivate(first, last) untied
            {
              ...
            }
            #pragma omp task in_reduction(+:result) firstprivate(first, last) untied
            {
              ...
            }
            #pragma omp task in_reduction(+:result) firstprivate(first, last) untied
            {
              ...
            }

            first += chunk;
            last  += chunk;
          }
      }
    }

    #pragma omp taskwait
  } // close parallel region
```

parallel_tasks/
05_task_taskgroup.c

A `taskgroup` region is declared: at its end, the completion of all tasks generated within it, and of their descendant, is explicitly ensured.

This task are participating to the reduction


using taskgroup & reduce
normalized runtime
num. of threads
04, 10 chunks
05, 10 chunks
05, 100 chunks
05L, 500 chunks
ideal
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
5
10
15
20
25
30
35
40
45

OpenMP taskloop


```
#pragma omp parallel proc_bind(close)
 {

  #pragma omp single nowait
   {
     //#pragma omp taskloop grainsize(N/1000) reduction(+:result)
    #pragma omp taskloop num_tasks(N/10) reduction(+:result)
     for( int ii = 0; ii < N; ii++ )
       {
         array[ii] = min_value + lrand48() % max_value;
         result += heavy_work_0(array[ii]) +
           heavy_work_1(array[ii]) +
           heavy_work_2(array[ii]);
       }
   }
   PRINTF("* initializer thread: initialization lasted %g seconds\n", CPU_TIME_th - tstart );

 } // close parallel region


  double tend = CPU_TIME;
#endif
```

parallel_tasks/
06_task_taskloop.c

OpenMP `taskloop`


```
#pragma omp parallel proc_bind(close)
 {

  #pragma omp single nowait
   {
   //#pragma omp taskloop grainsize(N/1000) reduction(+:result)
   #pragma omp taskloop num_tasks(N/10) reduction(+:result)
     for( int ii = 0; ii < N; ii++ )
       {
         array[ii] = min_value + lrand48() % max_value;
         result += heavy_work_0(array[ii]) +
           heavy_work_1(array[ii]) +
           heavy_work_2(array[ii]);
       }
   }
   PRINTF("* initializer thread: initialization lasted %g seconds\n", CPU_TIME_th - tstart );

 } // close parallel region


  double tend = CPU_TIME;
#endif
```

A `taskloop` region is declared: it blends the flexibility of tasking with the ease of loops

Tasks are created for each iteration

parallel_tasks/
06_task_taskloop.c

```
#pragma omp parallel proc_bind(close)
 {

  #pragma omp single nowait
   {
     //#pragma omp taskloop grainsize(N/1000) reduction(+:result)
    #pragma omp taskloop num_tasks(N/10) reduction(+:result)
     for( int ii = 0; ii < N; ii++ )
       {
         array[ii] = min_value + lrand48() % max_value;
         result += heavy_work_0(array[ii]) +
           heavy_work_1(array[ii]) +
           heavy_work_2(array[ii]);
       }
   }
   PRINTF("* initializer thread: initialization lasted %g seconds\n", CPU_TIME_th - tstart );

 } // close parallel region


  double tend = CPU_TIME;
#endif
```

To limit overhead, you can control the task generation by using of `num_tasks` and `grainsize` clauses

~~Tasks are created for each iteration~~ Tasks are created accordingly to clauses

parallel_tasks/
06_task_taskloop.c

that's all, have fun