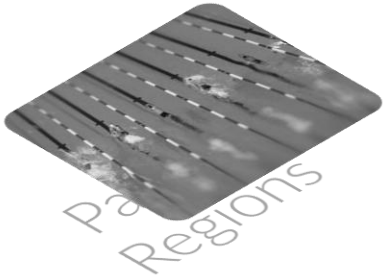
Advanced OpenMP
Tasks

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

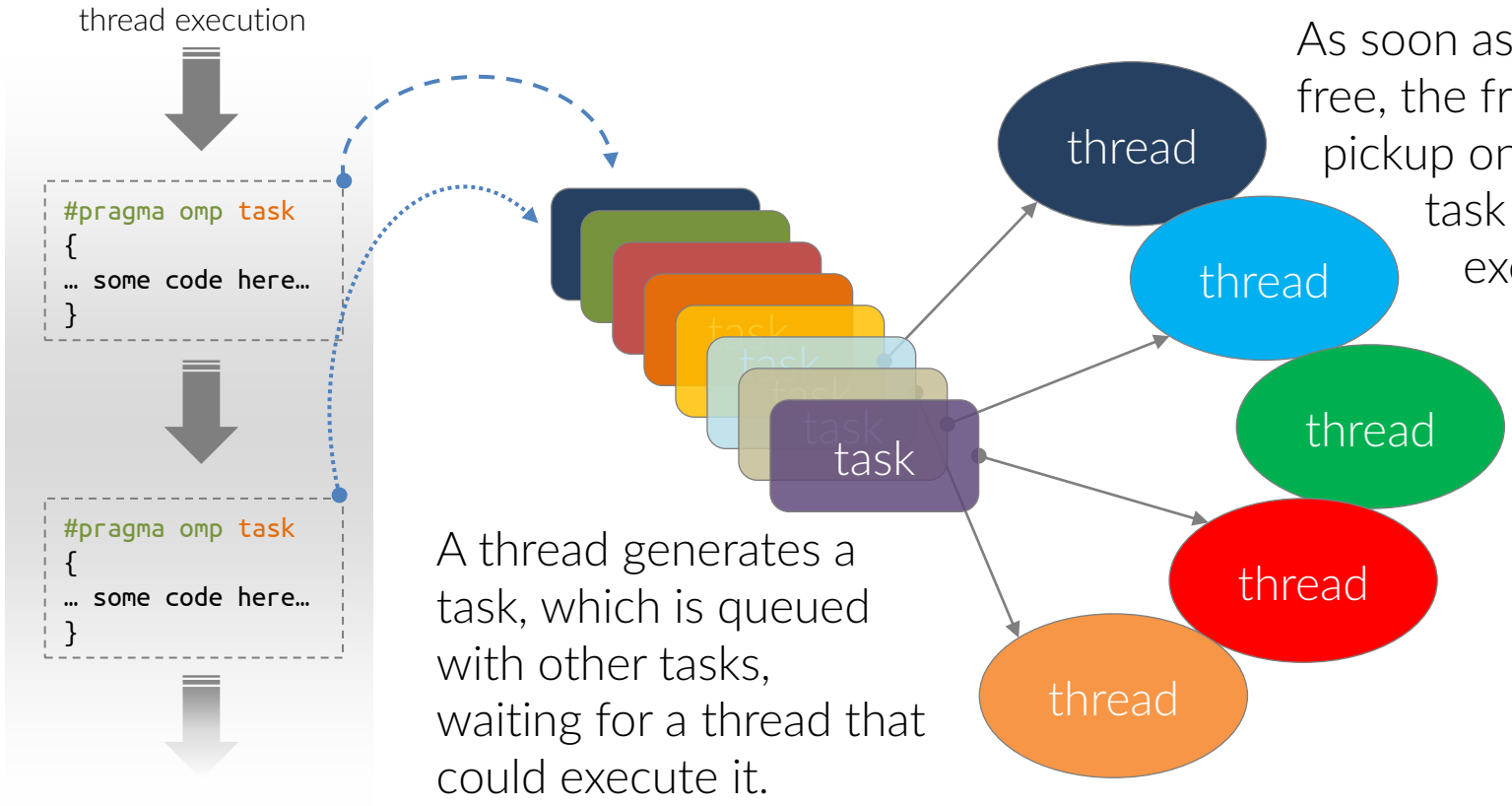
represents any piece of logically uniform “information”, that may be accessed by several threads; out-of-order access needs to be managed

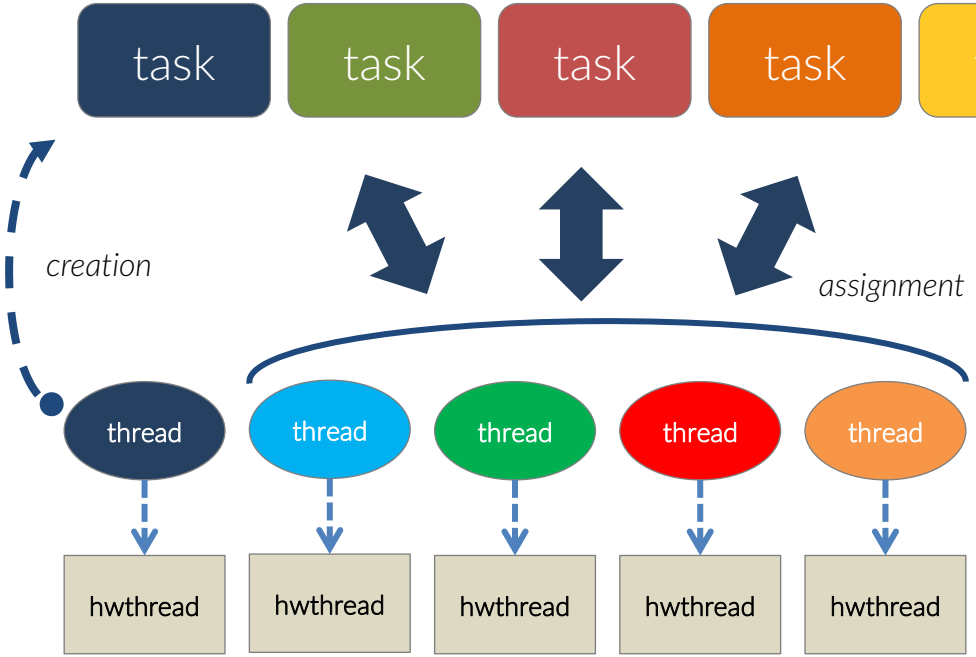
Concurrent access

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.



To make it clearer: creating tasks does not mean creating threads. The tasks are "units of work" that are assigned to the running threads. The pool of threads is unaltered(*) and mapped onto the underlying physical cores.

(*)unless, of course, there is nested parallelism involved

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

Creating tasks

```
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

Creating `tasks`

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.

Creating `tasks`

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

tasks variables scope

```
#pragma omp parallel
 {
   int me = omp_get_thread_num();

  #pragma omp single nowait
   {
     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", me );

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

    #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 after the end "
          "of the single region, I'm not waiting "
          "all the others\n", me );

   // does something change if we comment the
   // following taskwait ?
  #pragma omp taskwait

   // what if we comment/uncomment the following barrier ?
   //#pragma omp barrier

   printf(" +Hi there, finally that's me, thread %d "
          "at the end of the parallel region after all tasks ended\n",
          omp_get_thread_num());
 }
```

- It may seem a good idea to call the `omp_get_thread_num()` function only once instead of several times.

examples_tasks/
00_simple_taskwait_a.c

However, what we get is the following output:

```
:Hi, here is thread 2 after the end of the single region, I'm not waiting all the others
»Yuk yuk, here is thread 4 from within the single region
      Hi, here is thread 4 running task B
:Hi, here is thread 0 after the end of the single region, I'm not waiting all the others
:Hi, here is thread 1 after the end of the single region, I'm not waiting all the others
:Hi, here is thread 5 after the end of the single region, I'm not waiting all the others
:Hi, here is thread 6 after the end of the single region, I'm not waiting all the others
:Hi, here is thread 7 after the end of the single region, I'm not waiting all the others
      Hi, here is thread 4 running task A
«Yuk yuk, it is still me, thread 4 inside single region after all tasks ended
:Hi, here is thread 3 after the end of the single region, I'm not waiting all the others
:Hi, here is thread 4 after the end of the single region, I'm not waiting all the others
+Hi there, finally that's me, thread 2 at the end of the parallel region after all tasks ended
+Hi there, finally that's me, thread 1 at the end of the parallel region after all tasks ended
+Hi there, finally that's me, thread 7 at the end of the parallel region after all tasks ended
+Hi there, finally that's me, thread 0 at the end of the parallel region after all tasks ended
+Hi there, finally that's me, thread 5 at the end of the parallel region after all tasks ended
+Hi there, finally that's me, thread 4 at the end of the parallel region after all tasks ended
+Hi there, finally that's me, thread 6 at the end of the parallel region after all tasks ended
+Hi there, finally that's me, thread 3 at the end of the parallel region after all tasks ended
```

From which it looks like the same thread that creates the tasks also executes all of them.

tasks variables scope

What is wrong in the previous example (`examples_tasks/00_simple_taskwait_a.c`) is the scope of the variables.
The main point to consider here is that by its very nature, the tasks' creation is driven by the coeval data context and is not related to the values that any variable will have in the future at the moment of their execution.

As such, the rule-of-thumb is "data, unless otherwise stated, are *copied* in local copies so that to preserve the data context at the moment of creation".

Pragmatically, the effect is the same than declaring by default that data are **`firstprivate`**.

This fact is of paramount importance and ignoring it is a major source of bugs when dealing with tasks.

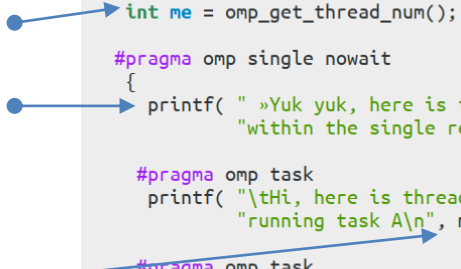
tasks variables scope

In our example `examples_tasks/00_simple_taskwait_a.c` the variable `me`, which is private for every thread, is inherited in the single region by the thread that enters there.

When the same thread enters in the task-creating region, the variable becomes `firstprivate`, and as such is copied in a local variable in the stack associated with the task. It then assumes the value it has for the creating task, i.e. its id.

Hence, when the task is executed, the executing thread receives this local variables with the value it had at the moment of creation, from which we can now understand the output of this code.

If, instead, we maintain the call to `omp_get_thread_num()`, that is executed by the executing tasks and then the correct id is stamped.



```
#pragma omp parallel
 {
  int me = omp_get_thread_num();

  #pragma omp single nowait
   {
    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", me );

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

    #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 after the end "
         "of the single region, I'm not waiting "
         "all the others\n", me );

  // does something change if we comment the
  // following taskwait ?
 #pragma omp taskwait

  // what if we comment/uncomment the following barrier ?
  //#pragma omp barrier

  printf(" +Hi there, finally that's me, thread %d "
         "at the end of the parallel region after all tasks ended\n",
         omp_get_thread_num());
}
```

tasks variables scope

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

  #pragma omp single
   {
     PRINTF(" : Thread %d is generating the tasks\n", me);

    #pragma omp task
     {
       PRINTF(" + Thread %d is executing T1\n", omp_get_thread_num());
       for( int jj = 0; jj < N; jj++ )
         result1 += heavy_work_0( array[jj] );
     }

    #pragma omp task
     {
       PRINTF(" + Thread %d is executing T2\n", omp_get_thread_num());
       for( int jj = 0; jj < N; jj++ )
         result2 += heavy_work_1( array[jj] );
     }

    #pragma omp task
     {
       PRINTF(" + Thread %d is executing T3\n", omp_get_thread_num());
       for( int jj = 0; jj < N; jj++ )
         result3 += heavy_work_2(array[jj] );
     }

   }

   PRINTF("\tThread %d is here (%g %g %g)\n", me, result1, result2, result3 );

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

Let's go deeper into this matter and examine the source code `examples_tasks/02_tasks_wrong.c` which is a code that insists with the same wrongdoings!

- `result*` are private variables (also note: they are *not* initialized; you must *always* initialize local accumulators).
- `result*` are inherited by the creating task
- `result*` are copied privately into the context of each created task; the local copies correctly performs as accumulators (although not initialized..)

Each thread sums its `result*` to the shared `result`. The intent here was that those tasks that executed the tasks sum the correct value while the others sum `0`.
However, these `result*` have nothing to do with those used inside the tasks.

tasks variables scope

This `examples_tasks/02_tasks.c` is a correct implementation of the previous strategy.

However, it is obvious that, as in the sections case, this strategy will never scale because it creates only 3 tasks and so, again, only 3 threads will perform all the calculations.

In the next slide we'll se how to manage such a simple case – even if it actually is perfectly suited for a `for` loop – using tasks.

```
#pragma omp parallel shared(result)
 {

  #pragma omp single  // having or not a taskwait here is irrelevant
                      // since there are no instructions after the
                      // single region
   {

    #pragma omp task   // result is shared, no need for "shared(result)" clause
     {
       double myresult = 0;
       for( int jj = 0; jj < N; jj++ )
         myresult += heavy_work_0( array[jj] );
      #pragma omp atomic update
       result += myresult;
     }

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

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

   }

   // all the threads will pile-up here, waiting for all
   // of them to arrive here.
 }
```

Variable workload

```
#pragma omp parallel
 {
  #pragma omp single nowait
   {
     for ( int i = 0; i < N; i++ )
      #pragma omp task
       heavy_work( function_of_i(i) );
   }
 }
```

examples_tasks/
03_variable_workload.c

The basic strategy in `examples_tasks/03_variable_workload.c` is to create a task for each of the *N* iterations.
We can control the *task granularity* by creating, for instance, a task that executes bunches of *n* iterations.

This strategy is not that different than what actually happens when the same problem is solved by using a `for` loop with `dynamic` schedule.

Here below, we present a table of the timing results for the execution of this code with a comparison of the `for dynamic` and tasks solution (see the code's comment for the details)

Results obtained on a single socket, 12 cores with 12 omp threads
Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz

The figures are the average among 10 repetitions on 10000 iterations with a workload base of 40000 (see the provided code for the details).
The total work in the case "decreasing" is larger than in the "random" case.

	Granularity = 1		**Granularity = 10**		**Granularity = 50**	
	for loop	*tasks*	*for loop*	*tasks*	*for loop*	*tasks*
Random workload	1.067	1.069	1.074	1.063	1.095	1.106
Decreasing workload	1.83	1.83	1.85	1.84	1.87	1.87

	Granularity = 1		Granularity = 10		Granularity = 50	
	for loop	*tasks*	*for loop*	*tasks*	*for loop*	*tasks*
Random workload	1.067	1.069	1.074	1.063	1.095	1.106
Decreasing workload	1.83	1.83	1.85	1.84	1.87	1.87

Message I

In spite of the fact that this case is perfectly suited for a `for dynamic` loop, generating the tasks – even 1 task per iteration, i.e. 10 thousands tasks in this example – results to be not less efficient.
Actually it would be reasonable to expect that under the hood of the `for dynamic` loop there was exactly the same queue technology.

Message II

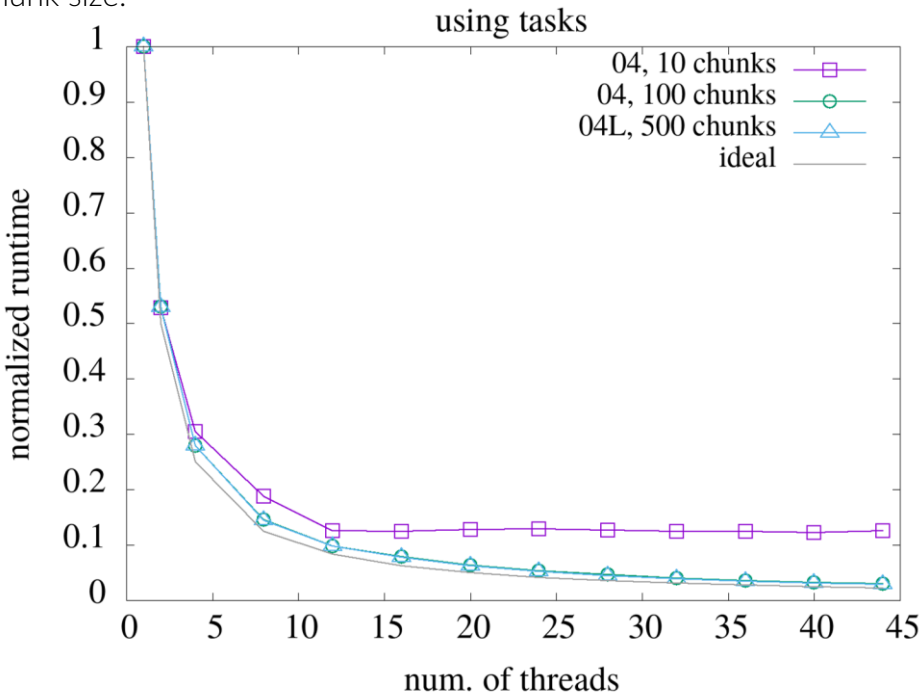
The case we adopted is "perfectly suited" for a `for dynamic` only if all your data are already in place, i.e. you do have an array to cycle over.
Quite the opposite, if your data are "arriving" the task solution is a very elegant and efficient one, while a `for` loop would be impossible.

Variable workload

```
#pragma omp parallel proc_bind(close) reduction(+:result)
  {
    #pragma omp single nowait
    {
      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 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 update
            result += myresult;
          }
          #pragma omp task firstprivate(first, last) shared(result) untied
          {
            double myresult    = 0;
            for( int ii = first; ii < last; ii++)
              myresult += heavy_work_1(array[ii]);
            #pragma omp atomic update
            result += myresult;
          }
          #pragma omp task firstprivate(first, last) shared(result) untied
          {
            double myresult    = 0;
            for( int ii = first; ii < last; ii++)
              myresult += heavy_work_2(array[ii]);
            #pragma omp atomic update
            result += myresult;
          }

          first += chunk;
          last  += chunk;


          #if defined (MIMIC_SLOWER_INITIALIZATION)
          nanot.tv_nsec = 200*uSEC + lrand48() % 100*uSEC;
          nanosleep( &nanot, NULL );
          #endif

        }
    }
  } // close parallel region
```

parallel_tasks/
02_variable_workload.v2.c

A different implementation, in which data are generated in chunks (they may be irregular, though) and a task is generated for each chunck. Here the parameter that regulates the granularity is the chunk size.



We stress that a 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.

examples_tasks/
03_variable_workload.v2.c

```
#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.

tasks variables scope

We stress that a 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;
}
```

tasks variables scope

We stress that a 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;
}
```

Unpredictable workload

Many times both the actual workload *and* the execution pattern are unpredictable and rigid thread-centric methods are either impossible or very hard to implement efficiently.

That is the case, for instance, of walking a linked-list and processing one of the encountered nodes under some given condition, like we depict in the next slide

```
#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 classical example: traversing a linked list

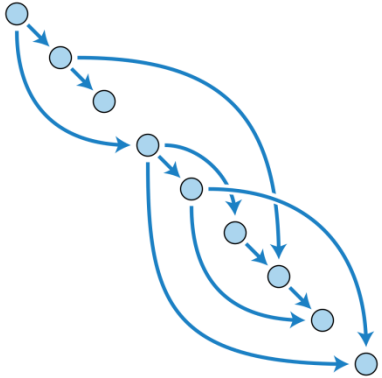
A task is generated for each node that must be processed

The calling thread continues traversing the linked list

Something else to do for the threads team, while the tasks are generated

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

Unpredictable workload



However, since as you know I do not like the linked-lists :)
we'll explore a different
an much more interesting case.
We'll traverse a Directed Acyclic Graph (DAG).

We're not studying in detail (*) what graphs, directed graphs and DAG are. Let's just say that DAG are data structures made of *vertices* (which are the data) and *edges* (which are data connections/dependences) each of whose is *directed* from a vertex to another so that there is an "ordered flow" that never loops.
Actually, we've used a pictorial view of a DAG in the forefront of this lecture to render clear what tasks are about.

examples_tasks/
05_dag.c

(*) you find a starting point on the wiki https://en.wikipedia.org/wiki/Directed_acyclic_graph

Unpredictable workload

In this example we first build a random DAG whose nodes contains some work to be done and whose edges represent dependences among nodes and their ancestors.
Each node could update its children and perform its work only when it has received updates by all its ancestors and so on.
A fraction of nodes are “great ancestors”, or root nodes, because they do not have any ancestors, and they trigger the update of the entire graph.

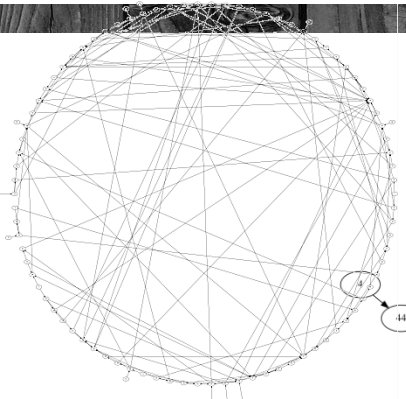
Such class of problems, which is very ubiquitous in computation and data analytics, would be *very* difficult, or impossible, to parallelize without the task approach.

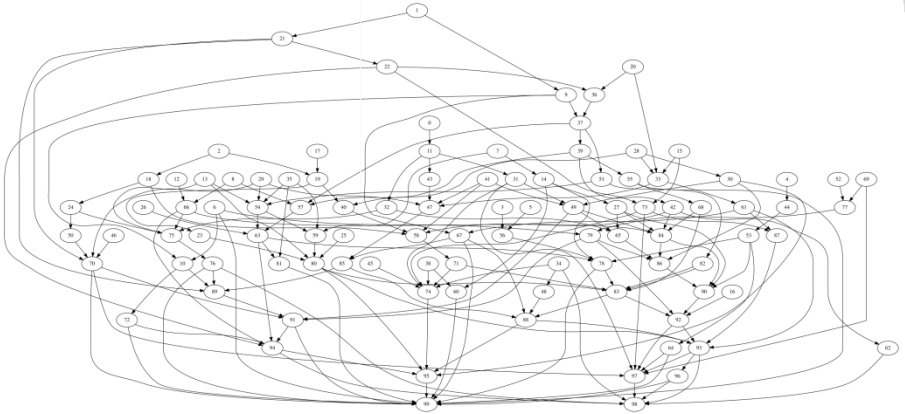
Unpredictable workload

The routine that generates the dag is named `generate_dag()` and it is pretty simple. The free parameters are: the total number of nodes, the number of root nodes, the minimum and maximum number of children per node, and the baseline workload per each node.

Here you find 3 different representation of the same small dag, having just 100 nodes (you can generate your own using the aforementioned routine).

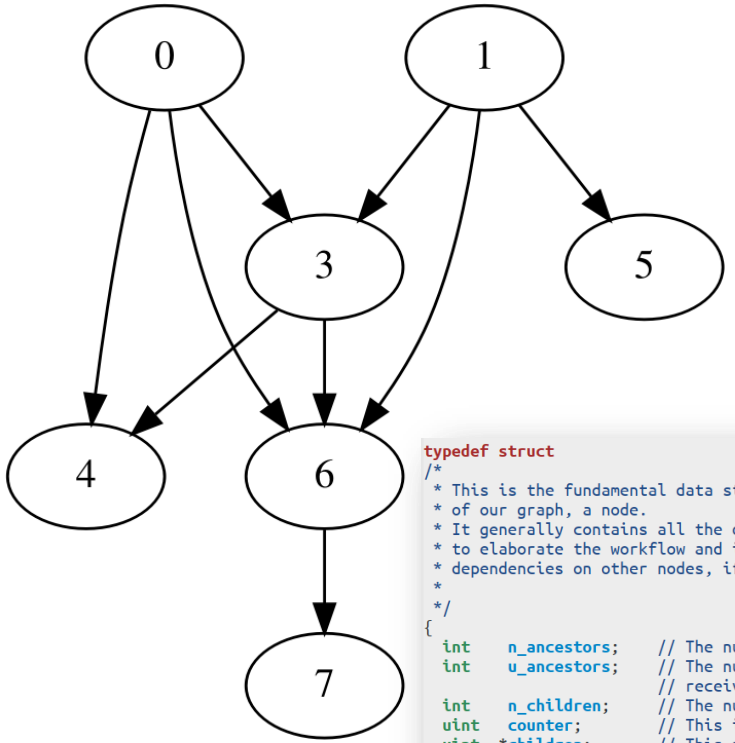
In the computational examples we'll use millions of nodes.





`examples_tasks/05_dag.c`

Unpredictable workload




```
typedef struct
/*
 * This is the fundamental data structure
 * of our graph, a node.
 * It generally contains all the data needed
 * to elaborate the workflow and its
 * dependencies on other nodes, if any.
 *
 */
{
  int    n_ancestors;    // The number of ancestors
  int    u_ancestors;    // The number of updates
                         // received from the ancestors
  int    n_children;     // The number of children
  uint   counter;        // This is the "amount of work"
  uint  *children;       // This points to the begin
                         // of the children list
  double result;         // holds the result of computation
} node_t;
```

Each node may have a variable number of ancestors and children. The directionality is accordingly to the sematic:
ANCESTOR -> NODE -> CHILD NODE

0 and 1 are “great ancestors” or “root nodes”, whereas 4, 5 and 7 are “leaves”.

Each ancestor propagate some information to the children by
- Increasing their work (the `counter` variable) by some amount
- Increasing the `u_ancestors` counter that keeps track of how many ancestors did update the node

Once a node has been updated by all its ancestors (i.e. `n_ancestors == u_ancestors`), it could both undergo its own calculation *and* propagate the relevant information to its own children.

The children list is stored elsewhere and not in the node data structure.

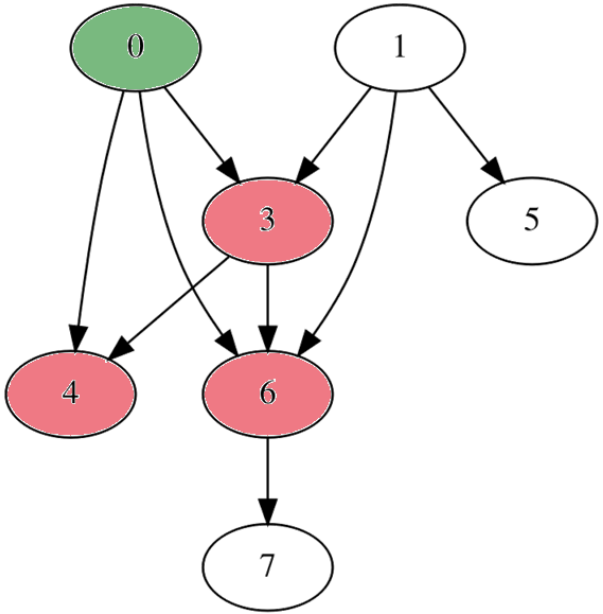
Notice that it is totally impossible to forecast what will be the execution pattern before the nodes are created.

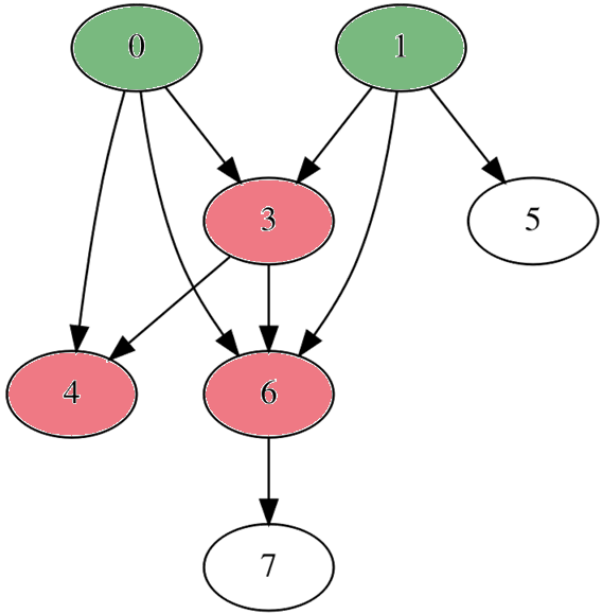
Unpredictable workload

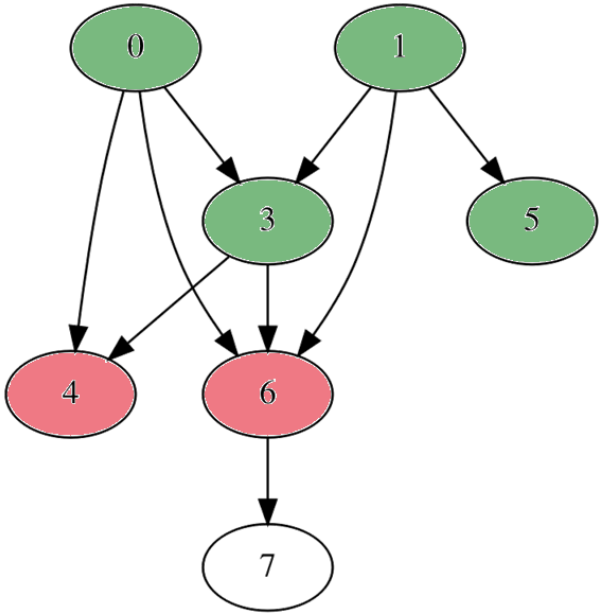
The root nodes are initialized, let's say we start from 0.
It updates its own descendants, that are then partially updated.

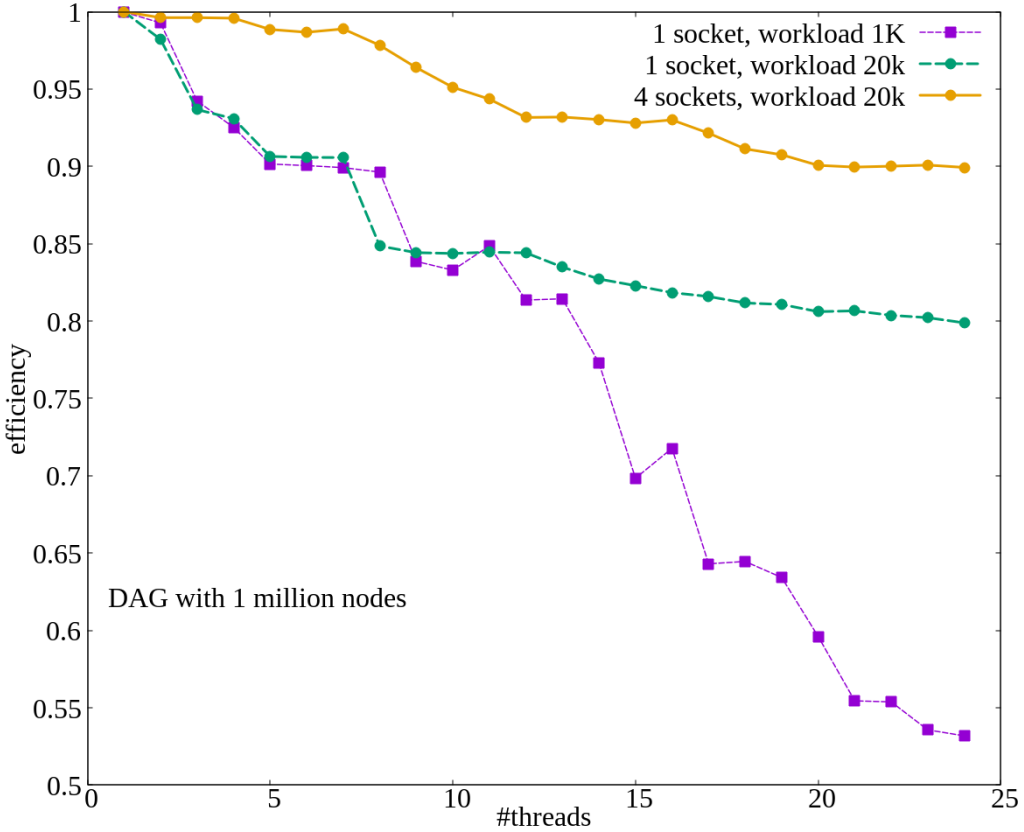
The root node 1 also is initialized and it propagate information through its edges

That triggers all the fully updated descendants to contribute to their children, and so on









These are some scaling results for a randomly generated dag with 1 million nodes having ~2.5 children in average, on a system

```
Intel® Xeon ® Gold 5118 CPU @ 2.30GHz
4 sockets, 12 cores/socket, 2 hwthreads/core
```

When using a single socket (violet and green lines) or 4 sockets (yellow line), for a small amount of work (violet line, ~30sec run for a single thread) or a much larger amount of work (green and yellow line, ~10min run for a single thread).

The scaling when using 4 sockets is very good, almost perfect up to 2 threads/socket.
That is a sign that memory access is not dominating this case (see comments in the source code).

`OMP_PLACES=sockets`
Violet and Green lines:
`OMP_PROC_BIND=master`
Yellow line:
`OMP_PROC_BIND=spread`

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

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(IN: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.

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);
}
```

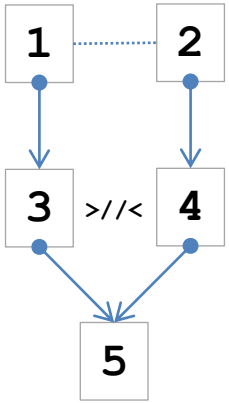
No dependence: output is variable, the printing tasks are independent off each other

```
#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

Tasks dependencies

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
```

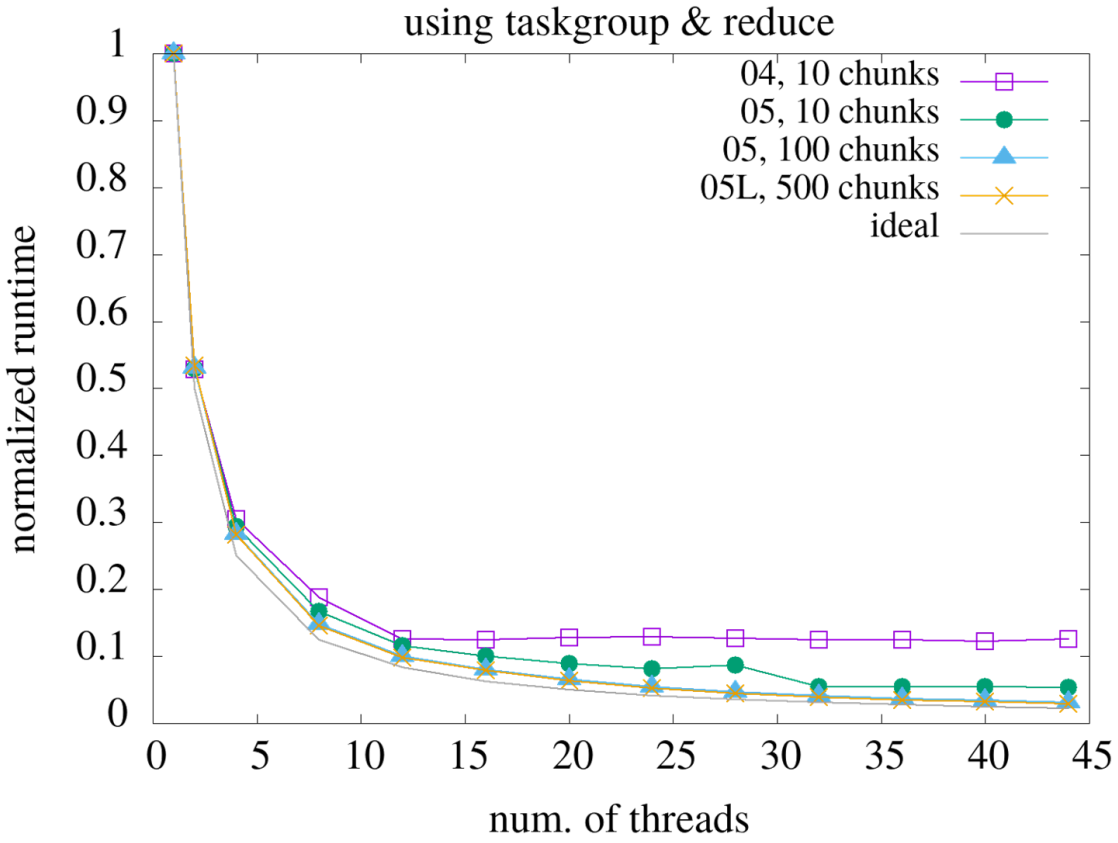
examples taken from openmp.org

```
#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
04, 10 chunks
05, 10 chunks
05, 100 chunks
05L, 500 chunks
ideal
normalized runtime
num. of threads
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

```
#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