OpenMP
TASKS

SCIENTIFIC &
DATA-INTENSIVE COMPUTING

Luca Tornatore - I.N.A.F.

Advanced HPC 2024-2025 @ Università di Trieste

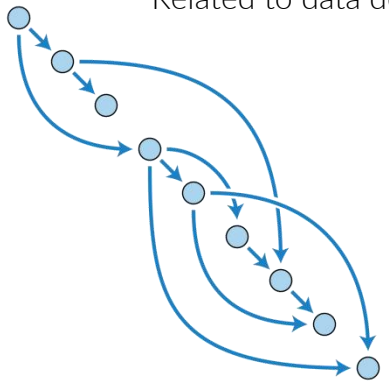
Tasks

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

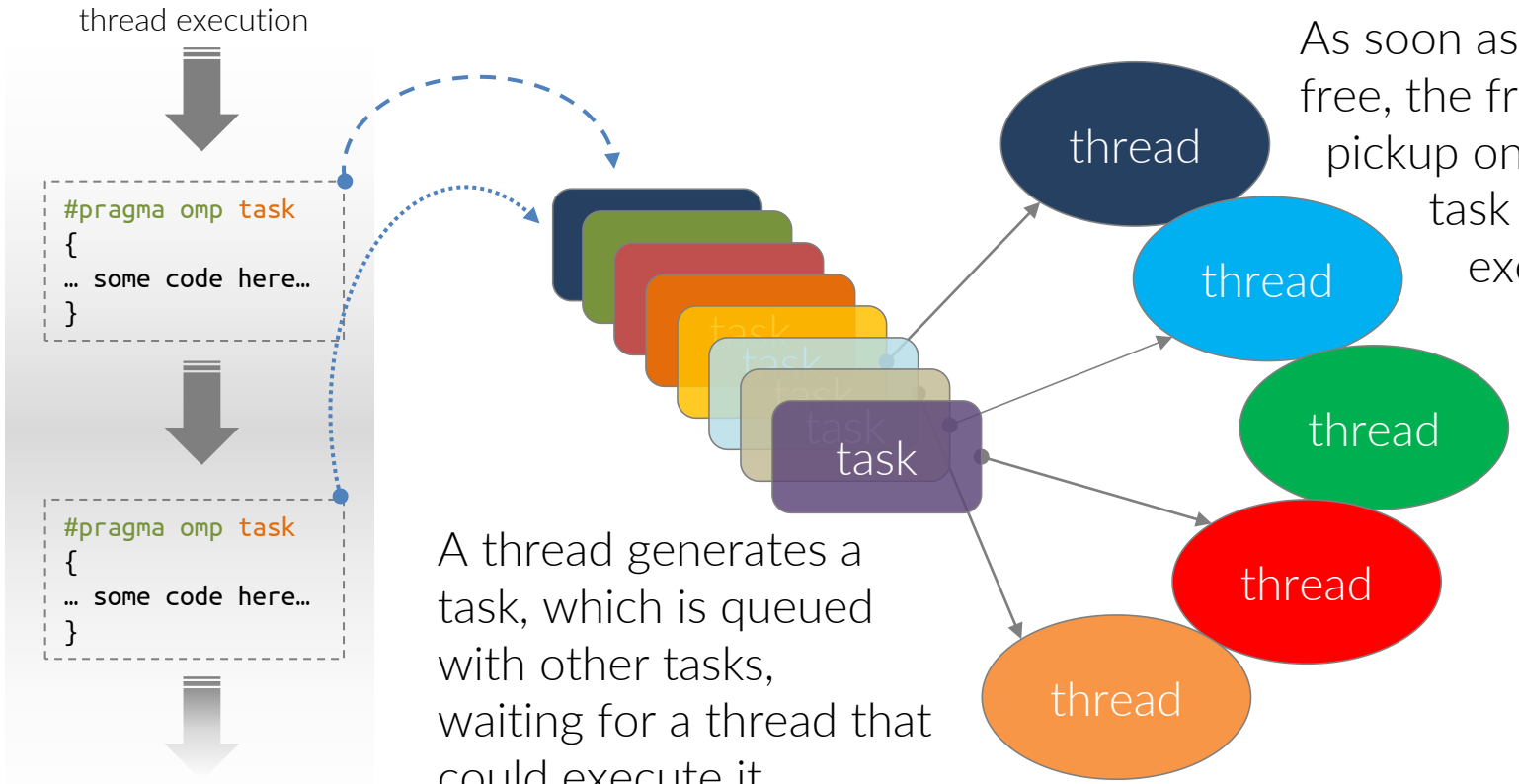
OpenMP `tasks`

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 and in any case it is nearly impossible to obviate to the intrinsic rigidity of the `sections` construct.

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.

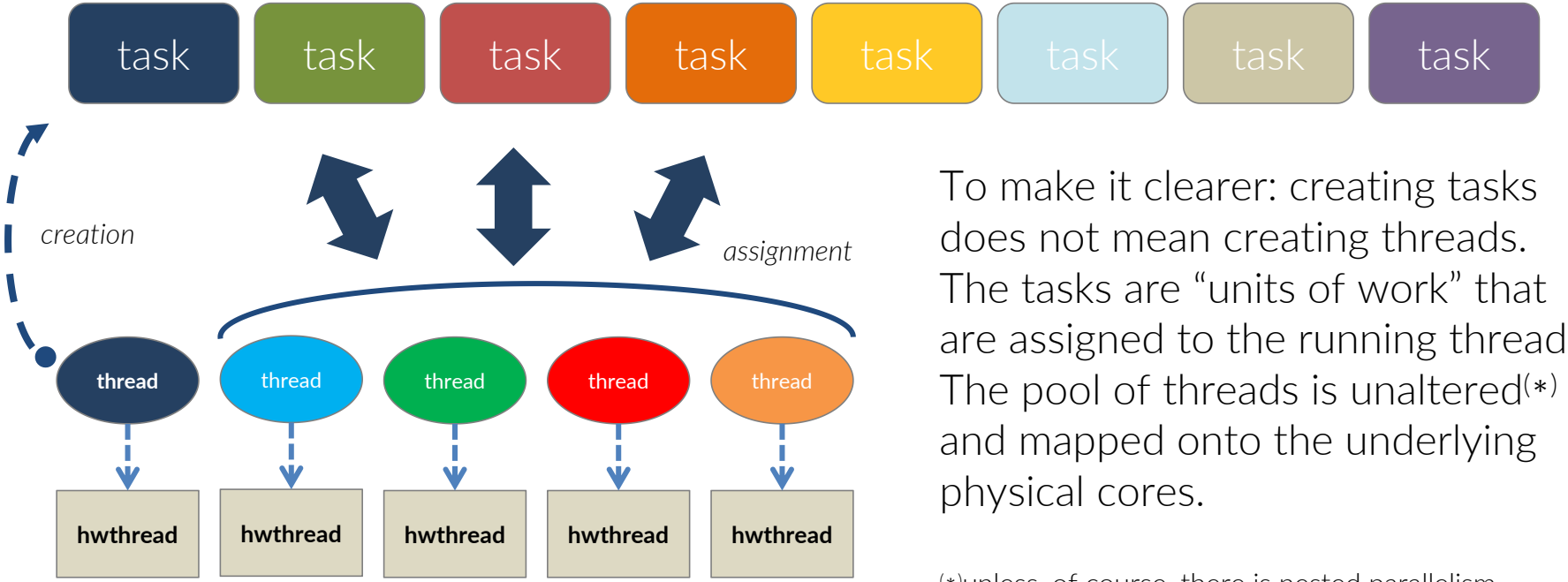
OpenMP tasks



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.

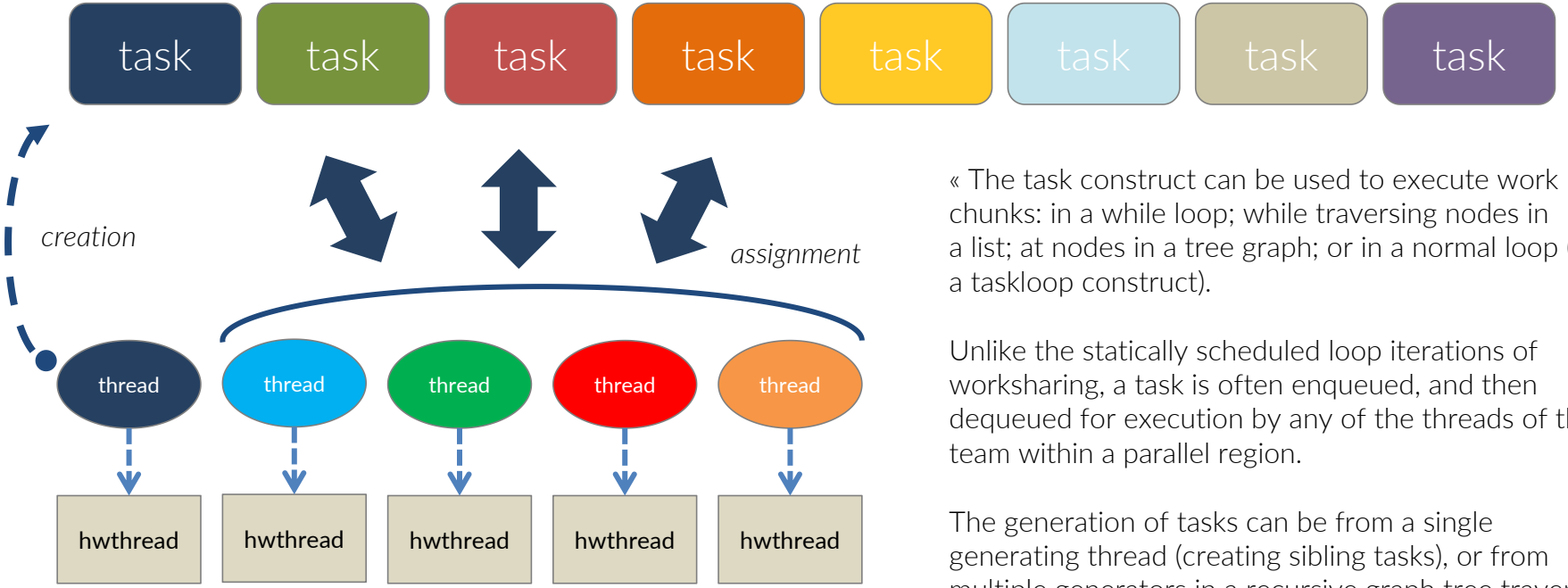
OpenMP tasks



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

OpenMP tasks



« The task construct can be used to execute work chunks: in a while loop; while traversing nodes in a list; at nodes in a tree graph; or in a normal loop (with a taskloop construct).

Unlike the statically scheduled loop iterations of worksharing, a task is often enqueued, and then dequeued for execution by any of the threads of the team within a parallel region.

The generation of tasks can be from a single generating thread (creating sibling tasks), or from multiple generators in a recursive graph tree traversals »

from OpenMP examples

OpenMP tasks

An effective way to understand how the tasks work is to imagine that “a task” is a sticky note on which somebody has written the instructions to be executed and the reference to the data involved (or the data themselves).

Then, people that are free will pass by, pick up a sticky note and execute the instructions.

In addition it will be possible to add constraints like “don’t pick a red one if there is any blue still there” or “before picking B A must have been executed”.

OpenMP `tasks`

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”, the task generation may 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 created only once, every task must be generated within a `single` or `master` region.

The `single` region may be 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.

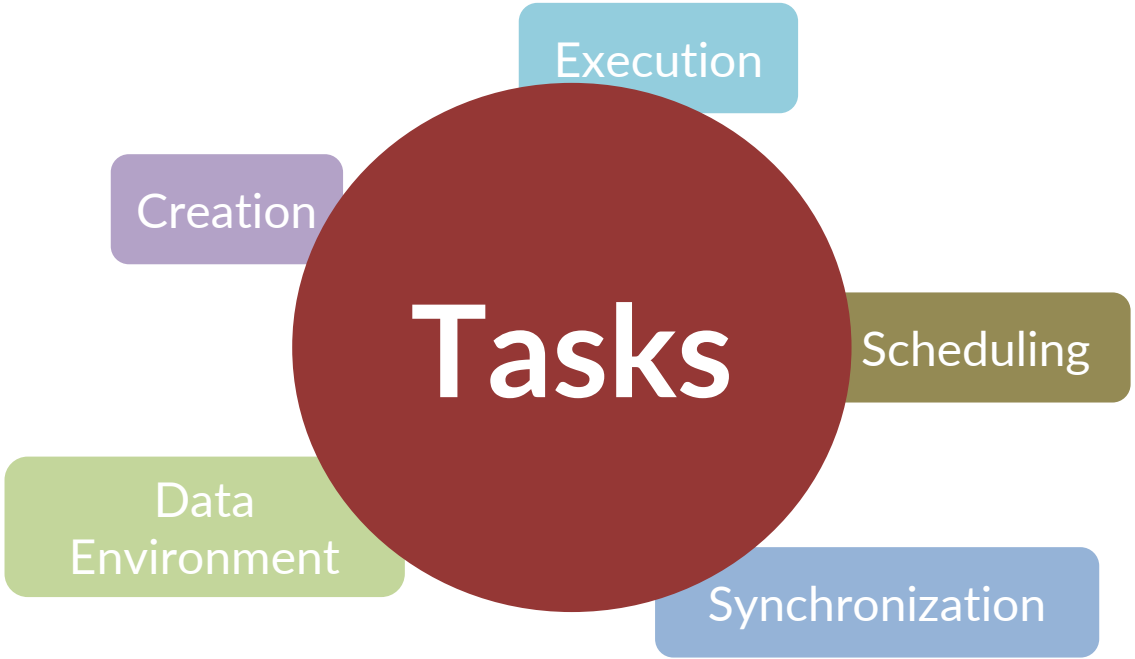
Key concepts in `tasks` management

Three key elements

- the code to execute
- data (the tasks *owns* the data)
- an n executor thread

The jargon

- **`task construct`**: the task directive, with clauses, and the structured code block
- **`task region`**: the region of code in which the tasks are created and the "execution" region where they are executed before the first sync point
- **`task`**: the code + the directions for the data



Key concepts in `tasks` management

Data environment

What data are assigned to each tasks?

How to deal with shared and private variables?

Creation & Execution

When are the tasks created?

How many of them are created?

When, and by who, are they executed?

Is there any priority ?

Synchronization & Dependence

How are the tasks synchronized ?

How are the tasks scheduled?

May they be dependent on other tasks?

Creating the tasks

Creating tasks

Let's start from a very basic example

The `single` region within which the tasks are created

```
#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() );
 }
```

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

While we do not know exactly when the task *will be* executed, do we know when we are sure that thay *have been* executed?

The last print will be printed before/while/after the task execution ?

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'll find that a random thread enters the region, while the others are waiting for the region to end,

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

After the conclusion of all the tasks, everybody can go

Creating tasks

Let's start from a very basic example

The `single` region within which the tasks are created

So we get that all the other threads are waiting here, at the implied barrier at the end of the `single` region.

That is what the OMP standard prescribes.

```
#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() );
 }
```

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

00_simple.c

Creating `tasks`

```
#pragma omp parallel
 {
   #pragma omp single nowait
   {
     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() );
 }
```

If we add a `nowait` clause to the `single` directive, we will get that the greeting message from all the other threads will almost certainly arrive before the greetings from the tasks execution.

In fact, the `nowait` lets the threads skip the `single` region entirely, and execute what is next, until the first scheduling point; which now is the very end of the parallel region.

`00_simple_nowait.c`

Creating `tasks`

```
#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());
 }
```

If we add a `taskwait` directive at the end of the `single` region, the single thread will not execute anything beyond that point *before that all the task created above had been executed*.

We'll see that this is a ***shallow* directive**: it *refers only* to the tasks directly created in that region and not to the tasks possibly created during the execution of the tasks created in the region.

`00_simple_taskwait.c`

When `tasks` are **guaranteed** to complete?

At the tasks scheduling points

- OpenMP **`barrier`** either *implicit* or *explicit*

  all tasks reated by *any* thread in the current parallel region are guaranteed to complete *after* the barrier exit

- task barrier **`taskwait`**

  all *children* tasks are completed, the encountering task is suspended until that is true
  it does not apply to *descendants*: i.e. it includes only direct children tasks.

- task construct: **`taskgroup`**

  All descendant tasks are guaranteed to be completed at the exit of the taskgroup region (see later); it baheves as an implicit `omp barrier`.

The scope of the tasks variables

The scope of `tasks` variables

```
#pragma omp parallel
 {

   #pragma omp single
   {
     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() );
     }

   }
```

00_simple.c

you may wonder why we call the `omp_get_thread_num()` function several times instead only once

The scope of `tasks` variables

```
#pragma omp parallel
 {
   #pragma omp single
   {
     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", me );
     }
```

What would be wrong in this snapshot 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.

(*) Of course we can create tasks that *purposely* process any data as they are at the moment of execution.

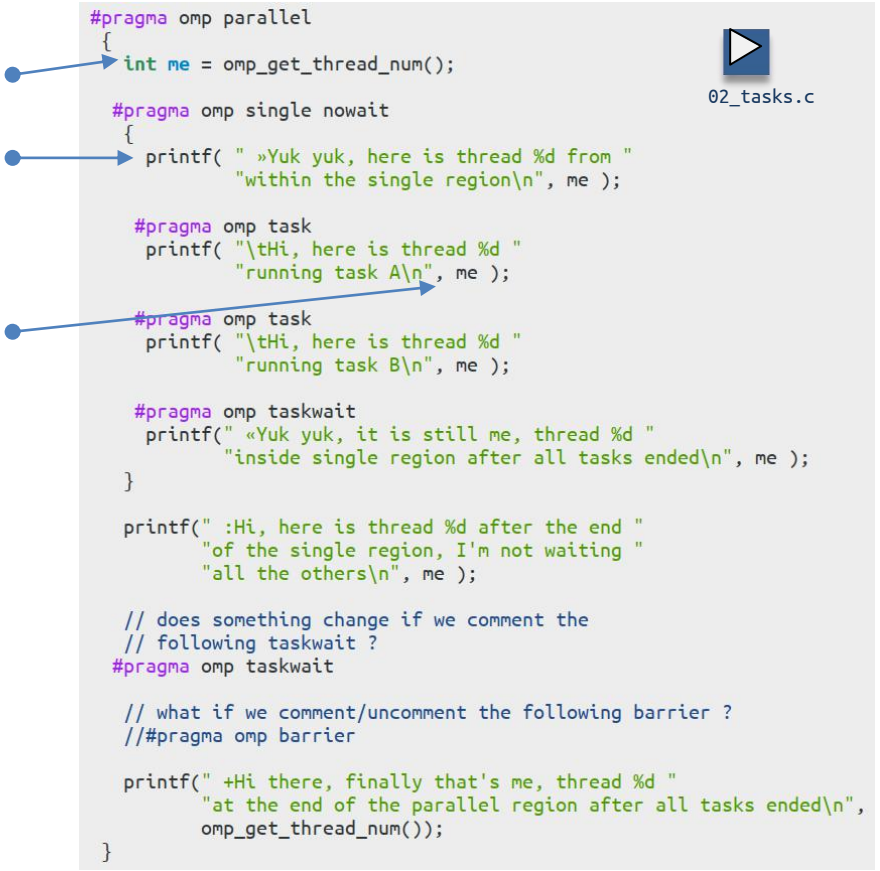
The scope of `tasks` variables

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());
}
```

The scope of `tasks` variables

```
#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;
```

02_tasks_wrong.c

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 perform as accumulators.

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.

The scope of `tasks` variables

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.

`examples_tasks/02_tasks.c`

```
#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.
 }
```

The scope of `tasks` variables

```
double pi = 3.1415;
double a = 0.0;

#pragma omp parallel
 {
    // pi, a : shared
   #pragma omp single private(a)
    {
      int i = 1;
      // a : private but not initialized
      // i : private because it's a local variable
     #pragma omp task
      {
        int j = 0;
        // i, a : firstprivate by default (a's value is undefined)
        // j    : is a tasks's private variable
        // pi   : shared

      }  // end of task
    } // end of single
 } // end of parallel
```

The scope of `tasks` variables

- When a variable is shared on the task creation
  the storage used is that referred with that name at the point where the task was created

- When a variable is private on the task creation
  the references to it (inside the task code region) use the uninitialized storage that is created when the task is executed

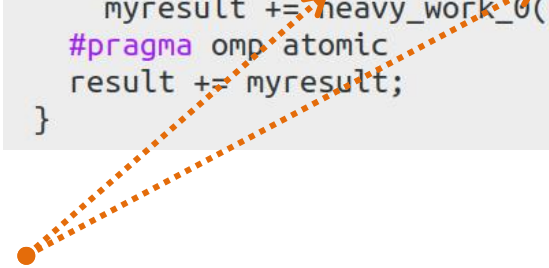
- When a variable is firstprivate on a task creation
  the references to it inside the task code region are to the new storage that is created and initialized with the value of the existing storage of that name when the task is created

The scope of `tasks` variables

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.

```
#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 are key variables for the task execution.

What if they were shared variables and hence they kept 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.

The scope of `tasks` variables

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;
}
```

The scope of `tasks` variables

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;
}
```

example: variable workload

check the relative section in the accompanying file `tasks.examples.pdf`

Unpredictable workload

We'll explore three examples of unpredictable workload that are perfectly suited for the task paradigm:

1. Traversing a linked-list
2. Solving a graph
3. Traversing a binary tree

Traversing a linked-list

A classical example: traversing a linked list

btw: there is a simple way to solve this problem using a for-loop. as an exercise, figure it out.

```
#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

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

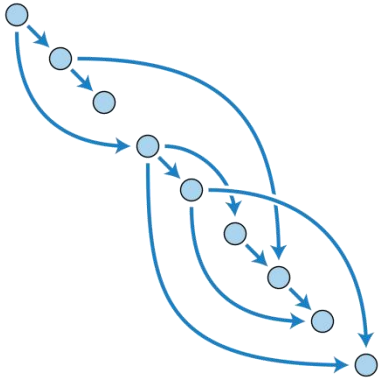
Due to the `nowait` clause, all the threads skip the implied barrier at the end of the `single` region and wait here for being assigned a task

Traversing a linked-list

check the relative section in the accompanying file `tasks.examples.pdf`

Solving a graph

The linked list walk is a pretty simple case and the sketch from the previous slide is sufficient to describe it.
We'll explore a more interesting case: the traversing of 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.

dag.c

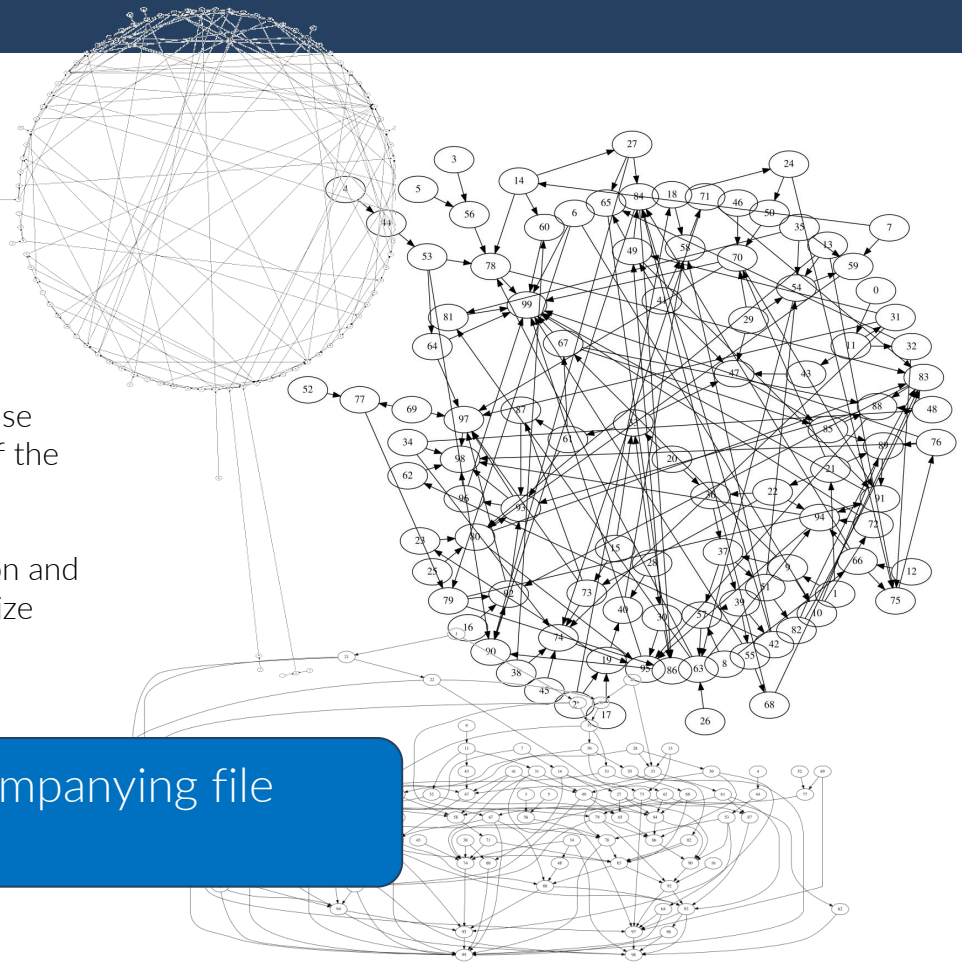
(*) you find a starting point on the wiki https://en.wikipedia.org/wiki/Directed_acyclic_graph

Solving a graph

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.



check the relative section in the accompanying file `tasks.examples.pdf`

Traversing a binary tree

Text TBD

check the relative section in the accompanying file `tasks.examples.pdf`

Synchronization and Dependence

Controlling Tasks Synchronization

OpenMP tasks synchronization

A key point to catch about 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.

In general, the synchronization tools that work for *thread-execution model* are not effective for the *tasks-execution*.
Summarizing:

- mutual exclusion like critical sections, atomic operations, mutex locks are ok
- event synchronization like idle wait barrier, boolean locks or other event synchronization are unlikely to be succesfull (and sooner or later they may lead to a deadlock)

OpenMP tasks synchronization

task 0

```
... ;  // do something
release( lock0 );
...;   // continuing
```

task 1

```
... ;  // do something
get_idle_or_spin( lock0 );
...;   // do something
release( lock1 );
...;   // continuing
```

task 2

```
... ;  // do something
get_idle_or_spin( lock1 );
...;   // continuing
```

`Task0` releases the lock tested in `Task1`;
`Task1` releases the lock tested in `Task2`.

Hence it is vital that `Task0` is executed.

Otherwise `Task1` will continue spinning when trying to acquire the `lock0` and it will never release `lock1`. Then, also `Task2` will be spinning with no end.

OpenMP tasks synchronization

How a bad synchronization may lead to a deadlock.
If the 3 tasks are executed by 2 threads, the second execution pattern ends in a deadlock

task 0

```
... ;  // do something
release( lock0 );
...;   // continuing
```

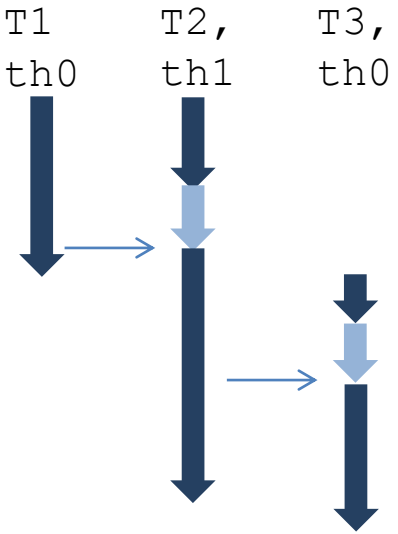
task 1

```
... ;  // do something
get_idle_or_spin( lock0 );
...;   // do something
release( lock1 );
...;   // continuing
```

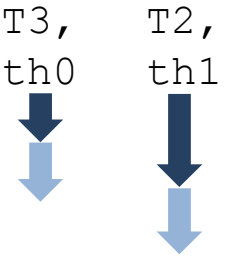
task 2

```
... ;  // do something
get_idle_or_spin( lock1 );
...;   // continuing
```

good execution pattern of tasks



execution pattern leading to a deadlock



OpenMP tasks synchronization

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 (see later in the section “controlling task creation”)

We have seen that some constructs enforce synchronization by ensuring that the tasks complete:

`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

Synchronizing `tasks:nowait`

Let's add a detail..

`nowait`

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

00_simple_nowait.c

```
#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() );
}
```

Synchronizing `tasks:nowait`

```
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 each task will have been executed at some special and well-defined points in the code.

Synchronizing `tasks:taskwait`

Let's add one more detail..

`taskwait`

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 threads of the `taskwait` region is the current team.

When a thread encounters a taskwait construct, the current task region is suspended until all child tasks that it generated *before* the taskwait region complete the 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() );
 }
```

00_simple_taskwait.c

Synchronizing `tasks:taskwait`

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
it
th
execution.

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() );
 }
```

00_simple_taskwait.c

Synchronizing `tasks:taskgroup`

The `taskwait` construct works well enough if you do not need a *deeper* task synchronization (remind: `taskwait` enforces to wait only for the task generated in the current task region by the generating thread, not for the possible children tasks generated by the threads executing the tasks).

Instead, **`taskgroup`** guarantees the completion of all the descendant.

```
#pragma omp taskgroup
structured-block
```

Synchronizing `tasks:taskgroup`

The **`taskgroup`** construct allows for a more sophisticated control of complex workflows in which you may want to control different groups of generated tasks

```
#pragma omp parallel
#pragma omp single
 {
   #pragma omp task
    start_receiving_data( );

   #pragma omp task
    background_work( );

   while ( data_queue_not_empty() )
    {
       tree_t *last_tree;
       #pragma omp taskgroup
        {
          #pragma omp task
           last_tree = build_tree(data_queue[last]);
        } // wait on tree building to be finished
       #pragma omp task
        communicate_tree(last_tree);
       #pragma omp taskgroup
        {
          #pragma omp task
           compute_tree(last_tree);
        } // wait on tree computation to be finished
       #pragma omp task
        communicate_tree_computation(last_tree);
     }
    } // only now is receiving_data() and background_work()
      // required to be complete
```

Synchronizing `tasks:taskgroup`

We enforce that *all* the tasks previsouly generated are concluded here, i.e. that the tree has been complete fpr the current bunch of data

All computations on the tree must be finished before we communicate the results

```
#pragma omp parallel
#pragma omp single
 {
   #pragma omp task
    start_receiving_data( );

   #pragma omp task
    background_work( );

   while ( data_queue_not_empty() )
    {
       tree_t *last_tree;
       #pragma omp taskgroup
        {
          #pragma omp task
           last_tree = build_tree(data_queue[last]);
        } // wait on tree building to be finished
       #pragma omp task
        communicate_tree(last_tree);
       #pragma omp taskgroup
        {
          #pragma omp task
           compute_tree(last_tree);
        } // wait on tree computation to be finished
       #pragma omp task
        communicate_tree_computation(last_tree);
    }
   } // only now is receiving_data() and background_work()
     // required to be complete
```

Synchronizing `tasks:taskgroup`

check the relative section in the accompanying file `tasks.examples.pdf`

Memory consistency model

let A = 0 and B = 0

Thread **a**	Thread **b**
[a1] A = 1	[b1] B = 1
[a2] print B	[b2] print A

what values will be printed for A and B ?

Let's start with a preliminary definition: that of Sequential Consistency, introduced by Leslie Lamport in 1979.
In simple terms, you may consider it as the assumption that the execution respects what is *intuitive* for *parallel threads* accessing the *same main memory*

1. each threads issues memory requests in the same sematic order than specified in the code
2. memory requests from all processors issued to an individual memory module are serviced from a single FIFO queue. Issuing a memory request consists of entering the request on this queue.

In other terms: every threads execute mem instructions in order, while the access to the same main memory implies that the mem instructions will be executed in *some* order.

nice introduction at this URL ; don't miss the OpenMP's page on memory model

Memory consistency model

let A = 0 and B = 0

Thread **a**	Thread **b**
[a1] A = 1	[b1] B = 1
[a2] print B	[b2] print A

what values will be printed for A and B ?

Relying on the sequential consistency, we see that any of the following execution orders is possible

```
[a1]→[a2]→[b1]→[b2] ⇒ print A=1, B=0
[b1]→[b2]→[a1]→[a2] ⇒ print A=0, B=1

[a1]→[b1]→[a2]→[b2] ⇒ print A=1, B=1
[a1]→[b1]→[b2]→[a2] ⇒ print A=1, B=1
[b1]→[a1]→…         ⇒ print A=1, B=1
```

and few other with similar effect: either 10, 01 or 11
What is ***not* possible** is to get `00` because that would imply that both `[a2]` and `[b2]` happen before `[a1]` and `[b1]` respectively.

the notation
`x → y`
means "`x` happens before `y`"

nice introduction at this URL ; don't miss the OpenMP's page on memory model

Memory consistency model

let A = 0

Thread **a**	Thread **b**	Thread **c**
`[a1] A = 1`	`[b1] A = 2`	`[b1] print A`

what values will be printed for A?

Even in presence of local memory, which is admitted in the OpenMP's memory model, the coherency protocol guarantees that the writes on the *same memory location* are seen by all the threads in the *same order*, whichever is the order.

By the assumption 2) of the sequential consistency, we know that either `[a1]` or `[b1]` will arrive first in the FIFO queue at the main memory and that order will appear the same to all the threads.

The problem with the sequential consistency is the performance, because it basically amounts to a serialization of the instructions execution.

That is why all the modern systems implement relaxed memory models that are more adequate for a modern out-of-order CPU with a cache hierarchy.

Memory consistency model

let A = 0 and B = 0

Thread **a**	Thread **b**
[a1] A = 1	[b1] B = 1
[a2] print B	[b2] print A

what if the *write* instructions could be buffered to a local storage (for instance, the cache memory?)

Note that for thread a there is no relation between the write of A and the read of B, and the other way around for thread b.
Hence, both of them could buffer the writes on A and B to their local store buffer and issue the read of B amd A, respectively.

In absence of an explicit synchronization, thread a and b will read the maini memory's value of B and A, which are both 0.
That would lead to the 00 result that the intuitive sequential consistency enforced.

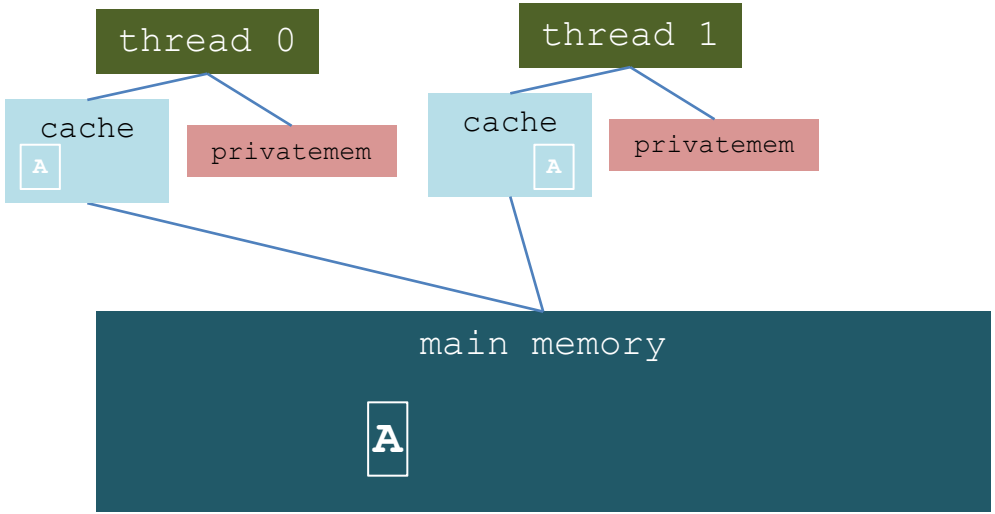
nice introduction at this URL ; don't miss the OpenMP's page on memory model

Memory consistency model

Then, it is all about the memory model which is defined in terms of:

- Coherence, i.e. the prescribed behaviour of the memory system when multiple accesses to the same memory locaiton are preformed by different threads
- Consistency, i.e. the order of the memory accesses by different threads

> The OpenMP API provides a relaxed-consistency, shared-memory model. All OpenMP threads have access to a place to store and to retrieve variables, called the memory. In addition, each thread is allowed to have its own temporary view of the memory. The temporary view of memory for each thread is not a required part of the OpenMP memory model, but can represent any kind of intervening structure, such as machine registers, cache, or other local storage, between the thread and the memory. The temporary view of memory allows the thread to cache variables and thereby to avoid going to memory for every reference to a variable. Each thread also has access to another type of memory that must not be accessed by other threads, called threadprivate memory.



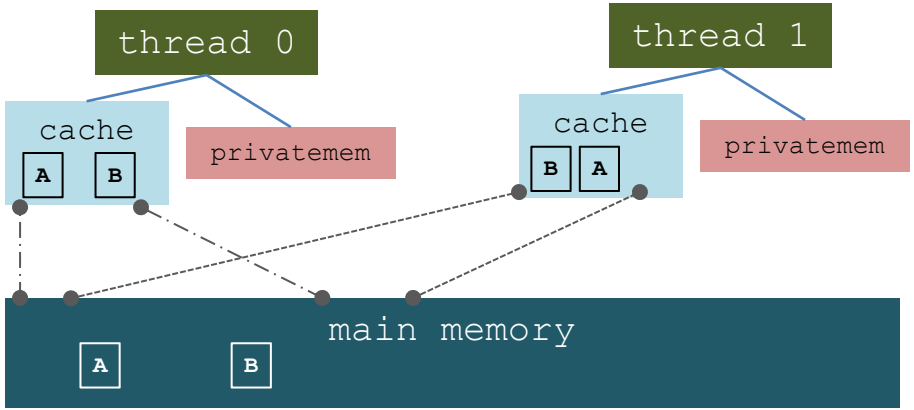
nice introduction at this URL ; don't miss the OpenMP's page on memory model

Memory consistency model

```
program semantic order
   W_A  R_A  W_B  R_B  S  ...
```

compilation

```
executable code
W_B  W_A  R_B  R_A S  ...
```



(1) The compiler re-orders the source code's program semantic order in any *equivalent* code order which it devises as more efficient.

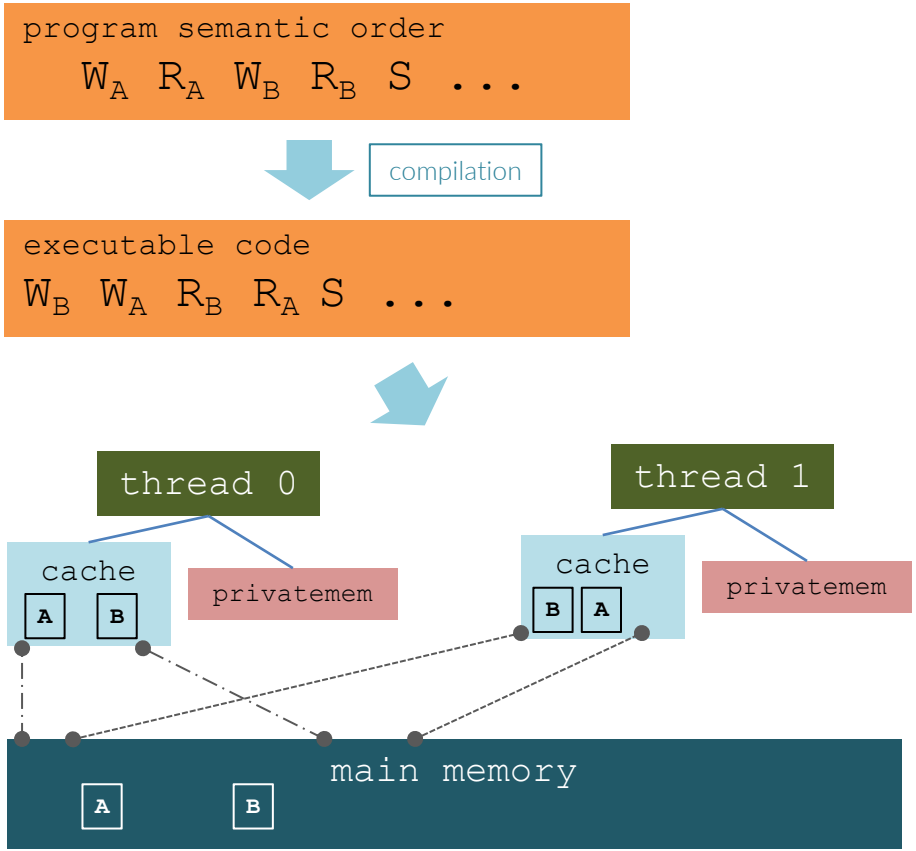
(2) The hardware re-orders the code order into the memory commit order that is the run-time realization of the code execution

(3) The "temporary view" of the memory for each thread may differ from the content of the main memory for every thread

(4) The consistency model relies on the admitted ordering of **R**eads, **W**rites and **S**ynchronizations
R→R. W→W, R→W, R→S, S→S, W→S

nice introduction at this URL ; don't miss the OpenMP's page on memory model

Memory consistency model

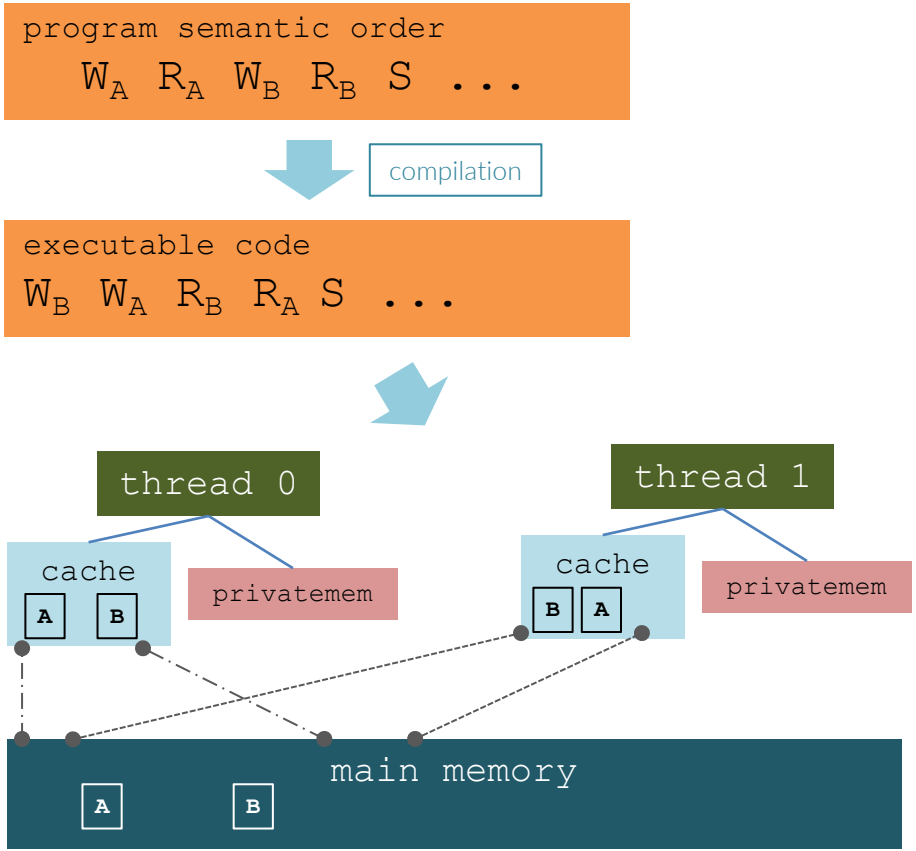


(4) The consistency model relies on the admitted ordering of **R**eads, **W**rites and **S**ynchronizations
R→R. W→W, R→W, R→S, S→S, W→S

As we have seen, in a multiprocessors system the (R,W,S) ops are sequentially consistent if
- the program order is respected for each cpu
- they are seen in the same order by all the other cpus
With sequential consistency the program orderm the code order and the commit order are all equivalent.

If we implement a relaxed consistency, some of the constraints on the ordering of memory ops (R,W,S) must be lifted

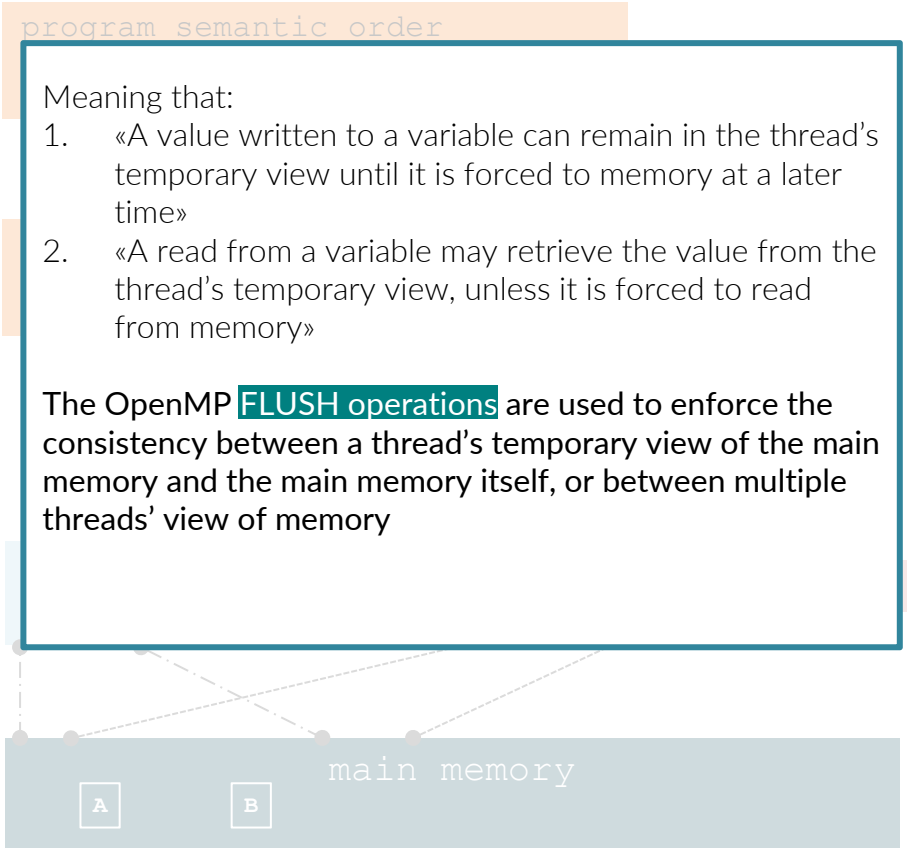
Memory consistency model



OpenMP defines a
RELAXED CONSISTENCY model

- The threads' temporary view of the memory is *not* required to be always consistent with the main memory until it is forced to be so
- S ops must be in sequential order accross threads
- On the same thread, R or W can not be reordered with S ops:
  the weak consistency guarantees that
  S→W. S→R, R→S, W→S, S→S

Memory consistency model



Meaning that:

1. «A value written to a variable can remain in the thread's temporary view until it is forced to memory at a later time»
2. «A read from a variable may retrieve the value from the thread's temporary view, unless it is forced to read from memory»

The OpenMP FLUSH operations are used to enforce the consistency between a thread's temporary view of the main memory and the main memory itself, or between multiple threads' view of memory

OpenMP defines a RELAXED CONSISTENCY model

- The threads' temporary view of the memory is *not* required to be always consistent with the main memory until it is forced to be so
- S ops must be in sequential order accross threads
- On the same thread, R or W can not be reordered with S ops:
  the weak consistency guarantees that
  `S→W. S→R, R→S, W→S, S→S`

nice introduction at this URL ; don't miss the OpenMP's page on memory model

FLUSH operations

see https://www.openmp.org/spec-html/5.2/openmp.html sec. 1.4.4

`strong`
`FLUSH`

Enforces consistency between a thread's temporary view and memory.
A strong flush operation is applied to a set of variables called the **flush-set**.
A strong flush **restricts reordering of memory operations** that an implementation might otherwise do.
Implementations must not reorder the code for a memory operation for a given variable, or the code for a flush operation for the variable, with respect to a strong flush operation that refers to the same variable.

A strong flush operation **provides a guarantee of consistency between a thread's temporary view and memory**. Therefore, a strong flush can be used to **guarantee that a value written to a variable by one thread may be read by a second thread**. To accomplish this, the programmer must ensure that the second thread has not written to the variable since its last strong flush of the variable, and that the following sequence of events are completed in this specific order:

[1] the value is written to the variable by thread 1
[2] the variable is flushed, with a strong flush, by thread 1
[3] the variable is flushed, with a strong flush, by thread 2
[4] the value is read from the variable by thread 2

FLUSH operations

see https://www.openmp.org/spec-html/5.2/openmp.html sec. 1.4.4

`release or acquire FLUSH`	It can enforce consistency between the views of memory of two synchronizing threads.
`release`	A release flush guarantees that any prior operation that writes or reads a shared variable will appear to be completed before any operation that writes or reads the same shared variable and follows an acquire flush with which the release flush synchronizes. A release flush will propagate the values of all shared variables in its temporary view to memory prior to the thread performing any subsequent atomic operation that may establish a synchronization.
`acquire`	An acquire flush will discard any value of a shared variable in its temporary view to which the thread has not written since last performing a release flush, and it will load any value of a shared variable propagated by a release flush that synchronizes with it into its temporary view so that it may be subsequently read.

Release and Acquire flushes may also be used to guarantee that a value written to a variable by one thread may be read by a second thread. To accomplish this, the programmer must ensure that the second thread has not written to the variable since its last acquire flush, and that the following sequence of events happen in this specific order:

[1] the value is written to the variable by thread 1
[2] thread 1 performs a release flush
[3] thread 2 performs an acquire flush
[4] the value is read from the variable by thread 2

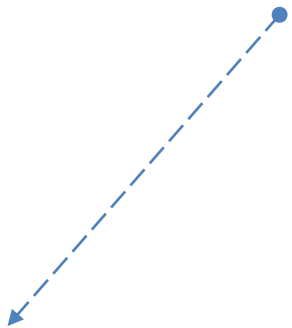
nice introduction at `this URL` ; don't miss the OpenMP's `page on memory model`

FLUSH operations

Consider this code snippet

```
int    counter = 0;
#pragma omp parallel
{
   ...
  #pragma omp atomic update
   counter++;

   ...
   double x = f(counter, ...)
}
```

how can you ensure that *all* the threads have update counter before that any thread calls `f` ?



FLUSH operations

Consider this code snippet

```
int    counter = 0;
#pragma omp parallel
{
   ...
  #pragma omp atomic update
   counter++;
   ...
  #pragma omp barrier
   double x = f(counter, ...)
}
```

how can you ensure that *all* the threads have update counter before that any thread calls **f** ?

you need to put a barrier here, which which implies a a strong **flush**

nice introduction at this URL ; don't miss the OpenMP's page on memory model

FLUSH operations

Consider this code snippet

```
int    counter = 0;
#pragma omp parallel
{
   ...
  #pragma omp atomic update
   counter++;
   ...
  #pragma omp barrier
   double x = f(counter, ...)
}
```

how can you ensure that *all* the threads have update counter before that any thread calls `f` ?

you need to put a barrier here, which which implies a a strong **`flush`**

however, this is very costly and inherently serializing

FLUSH operations

```
#pragma omp parallel
{
   ...
  #pragma omp atomic update
   counter++;

  #pragma omp flush(counter) acquire
   double z = f(counter...);
   ...
}
```

strong flush implied at entry and exit of the atomic operation

specific **flush** used here to force the reading from main memory instead of local cache

nice introduction at this URL ; don't miss the OpenMP's page on memory model

FLUSH operations

```
#pragma omp parallel
{
   ...
  #pragma omp critical (write_x)
   x = x + result;
   ...
  #pragma omp critical (write_z)
   double z = foo(...);
   ...
}
```

It is better to use **named** critical **regions** that implies a strong flush

OpenMP tasks synchronization

An additional mechanism that you can use to manage threads (and hence tasks) synchronization is the **lock**, via the **omp_lock_t** type.
A lock may be nested: **omp_nest_lock_t**.
A lock implies a memory fence, i.e. a “flush”, for all the thread’s visible variables

`omp_[nest]_lock_[init |destroy] (omp_lock_t *)`	Initialize or destroy a lock.
`omp_[nest]_lock_[set|unset] (omp_lock_t *)`	Acquire or release a lock; that is a blocking function, ir returns only when the lock is acquired.
`omp_[nest]_lock_test (omp_lock_t *)`	Test whether a lock is available; returns 1 on success (lock is acquired) returns 0 on failure.

When a lock is nested, it can be acquired multiple times *always by the same thread that acquired it before.*

Note: the locks can be initialized with an ***hint*** about what will be their typical usage:

```
void omp_init-lock_with-hint ( omp_lock_t *t, omp_lock_hint_t hint);
void omp_init_nest_lock_wi th-1lint ( omp_lock_t *t, omp_lock_hint_t hint);

typedef enum omp_lock_hint_t {
         omp_lock_hint_none = 0,
         omp_lock_hint_uncontended = 1,
         omp_lock_hint_contended = 2,
         omp_lock_hint_nonspeculative = 4,
         omp_lock_hint_speculative = 8
         } omp_lock_hint_t;
```

Note: the locks can be initialized with an ***hi*** no hint given will be their typical usage:

```
void omp_init-lock_with-hint ( omp_lock_t *t, omp_lock_hint_t hint);
void omp_init_nest_lock_wi th-1lint ( omp_lock_t

typedef enum omp_lock_hint_t {
        omp_lock_hint_none = 0,
        omp_lock_hint_uncontended = 1,
        omp_lock_hint_contended = 2,
        omp_lock_hint_nonspeculative = 4,
        omp_lock_hint_speculative = 8
        } omp_lock_hint_t;
```

It will be rare (`uncontended`) / common (`contended`) that multiple threads run it concurrently

The lock should (`speculative`) / should **not** (`nonspeculative`) use speculative techniques, like transactional memory

OpenMP tasks synchronization

Note: the lock ownership is bound to task regions and not (just) to threads!

```
omp_lock_t lock;
omp_init_lock( &lock );
omp_set_lock( &lock );              // the thread 0 is acquiring the lock

#pragma omp parallel
 {
     #pragma omp master
      {
         omp_unset_lock( &lock)     // the thread 0 wants to release the lock
                                    // but that is not allowed because the task
                                    // region is not the same
      }
 }
```

Example: Building a heap

Now we will inspect an examples of non-trivial usage of tasks locks:

building a heap with double-linked list

check the relative section in the accompanying file `tasks.examples.pdf`

OpenMP tasks

Controlling the task creation
Clauses

Controlling the task creation

The task creation construct

```
#pragma omp task
  { ... }
```

adtmits a number of clauses that allow to control the creation of tasks

```
#pragma omp task clause, clause, ...
  { ... }
```

Controlling the task creation

task creation clauses

`if(expr)`	when `expr` evaluates false, the encountering thread does not create a new task in the tasks queue; instead, the execution of the current task is suspendend the execution of (what it would have been) the newly created task is *undeferred* and started; The task suspended is resumed afterwards.
`final(expr)`	The `final(expr)` clause can be used to suppress the task creation and then to control the tasking overhead, especially in recursive task creation. If `expr` evaluates *true*, no more tasks are generated and the code is executed immediately. That is propagated to all the children tasks. That is called an *included* task and is *always* undeferred. Use `omp_in_final()` to check if the task is a final one.
`mergeable`	This clause avoid a separated data environment to be created if a task is undeferred or included
`tied, untied`	This clause let a suspended task to be resumed to a different thread than the one that started it. The defalt option is tied.
`depend, priority`	*we will cover this afterwards*

Controlling the task creation

task creation clauses

`if(expr)`

This is a "**shallow**" clause.
It only affects the encountering task

`final(expr)`

This is a "**deep**" clause.
If a task is final, all the child tasks are final too.

`mergeable`

This clause avoid a separated data environment to be created.

`tied,untied`

This clause let a suspended task to be resumed to a different thread than the one that started it. The defalt option is tied.

`depend, priority`

we will cover this afterwards

Task creation example: QuickSort

Let's consider a classical example among the *sorting algorithms*, i.e. the **quicksort.**
That is a *divide-et-impera* algorithm which subdivides a problem in smaller similar problems and solve them.
The easiest formulation is recursive:

```
void quicksort( data_t *data, int low, int high )
{
  if ( low < high ) {
    int p = partition ( data, low, high );

    quicksort( data, low, p );
    quicksort( data, p, high );
  }
  return;
}
```

check the relative section in the accompanying file `tasks.examples.pdf`

Task schedulilng and switching

```
#include <omp.h>
void something_useful();
void something_critical();
void foo(omp_lock_t * lock, int n)
{
  for(int i = 0; i < n; i++) {
    #pragma omp task
     {
       something_useful();
       while (!omp_test_lock(lock)){
          #pragma omp taskyield
       }
       something_critical();
       omp_unset_lock(lock);
     }
  }
}
```

the taskyield introduces an explicit task scheduling point, and may lead to the suspension of the calling task.

OpenMP tasks

Controlling task execution
*Task **priorities** and **dependencies***

Task 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 data 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

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`, 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` or `out` dependence on the same memory region;
  it will be *mutually exclusive* with another `mutexinoutset` sibling task, meaning that they can be executed in any order but not at the same time.

(*)`INOUT` is a relic, no longer used.
It is the same than `OUT`.

Tasks dependencies

When an instruction (task) depends on the result of another instruction (task), that is called a “flow dependency” or “true dependency” and referred as Read-After-Write

(you need to read a result after it is written)

RaW
Read after Write
“flow dependence”

The task 1 reads 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_courious( *the_answer );
```

Tasks dependencies

When an instruciton (task) may change the result of another instruction (task), the oredering must be strict.
That is called “anti-dependency” and referred as Write-After-Read

GET(Y)
X = SQRT(Y)
Y *= 2

RaW
Read after Write

```
The task 1 reads 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_courious( *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_wise( *the_question );
#pragma omp task depend(OUT:the_question)
 function_courious( *the_question );
```

Tasks dependencies

When two instructions (tasks) are independent of each other but writes in the same memory location, then there is an “output dependency” or a Write-After-Write

That may also be considered a “false dependency” or “name dependency”, as it may be suffiecient a variable renaming to remove it

RaW
Read after Write

```
The task 1 reads 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_wise( *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_courious1( *the_question );
#pragma omp task depend(OUT:the_question)
 function_courious2( *the_question );
```

Tasks dependencies

When two instructions (tasks) read the same memory region, there is actually no dependency

Read-After-Read

RaW
Read after Write

The task 1 reads 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_wise1( *the_question );
#pragma omp task depend(IN:the_question)
 function_wise2( *the_question );
```

Tasks dependencies

RaW
Read after Write

The task 1 reads 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_wise1( *the_question );
#pragma omp task depend(IN:the_question)
 function_wise2( *the_question );
```

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);
}
```

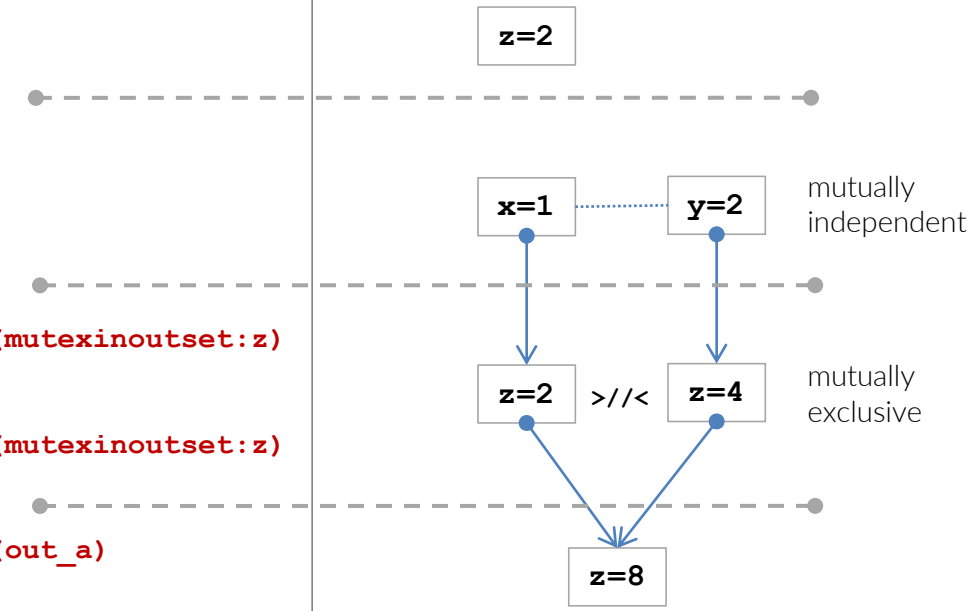
```
No dependence: output is variable, the
printing tasks are independent off each
other
#pragma omp single
{
  #pragma omp task shared(x) depend(out:x)
   x = 2;
  #pragma omp task shared(x) depend(in:x)
   printf("x + 1 = %d\n", x+1);
  #pragma omp task shared(x) depend(in:x)
   printf("x + 2 = %d\n", x+2);
}
```

examples taken from openmp.org

Tasks dependencies

Mutually exclusive dependency

```
… z is assigned some value here …;
#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
```



examples taken from openmp.org

Tasks dependencies

You can couple the `taskwait` directive with a `depend` clause to enforce the sync of tasks with a given dependence

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

// task 1
#pragma omp task shared(x) depend(out: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 couple the `taskwait` directive with a `depend` clause to enforce the sync of tasks with a given 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(out:x)
 x += 1;

// task 2
#pragma omp task shared(x, y) depend(in:x) depend(out: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

Controlling task creation and execution

*Task creation in **loops***

***Reduction** operations with tasks*

OpenMP task reduction

In OpenMP 5.0 the *task* modifier to the reduction clause has been introduced also for the ordinary parallel regions and work-sharing constructs

```
double sum = 0;
#pragma omp parallel reduction(task, +:sum)
{
  sum += 1.0;               // this is an implicit task reduction statement

  #pragma omp single
  for ( int i = 0; i < N; i++ )
   #pragma omp task in_reduction(+:sum)  // explicit task reduction
    sum += some_computation( i );
}


#pragma omp parallel for reduction(task, +:sum)
for ( int i = 0; i < N; i++ )
  {
    sum -= (double)i;

   #pragma omp task in_reduction(+:sum)
    sum += some_other_computation( i );
  }
```

OpenMP `taskloop`

Many times happens that you need to create tasks in a loop (for instance, a task for every entry, or sections, of an array).
The **`taskloop`** construct has been conceived to ease this cases, combining the `for` loops and the `tasks` natively.

```
#pragma omp taskloop [clause[[,] clause]…]
for-loops        (perfectly nested)
```

Clauses are very similar to both the usual `for` and `task` constructs:
`private, firstprivate, lastprivate, shared, default, if, final, priority, untied, mergeable`

There are 3 peculiar clauses, instead:
`grainsize, num_tasks, nogroup`

OpenMP `taskloop`

Many times happens that you nee
task for every entry, or sections, o
The **`taskloop`** construct has been
the `for` loops and the `tasks` nativ

```
#pragma omp tasklo
for-loops          (perfe
```

Clauses are very similar to both th
`private, firstprivate, lastprivate, s`
`mergeable`

There are 3 peculiar clauses, instea
`grainsize, num_tasks, nogrou`

`grainsize (arg)`
arg is a positive integer.
It is used to regulate the granularity of the work assignment, so that the amount of work per task be not too small.
The number of loop iterations assigned to a task is the minimum btw grainsize and the number of loop iterations, but does not exceed 2*grainsize

`num_tasks (arg)`
arg is a positive integer.
It is used to limit the tasking overhead.
That is the maximum number of tasks generated at run-time.

`nogroup`
The tasking construct is not embedded in an otherwise implied `taskgroup` construct.

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
```

day26/example_tasks/
07_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

day26/example_tasks/
07_taskloop.c

OpenMP `taskloop`

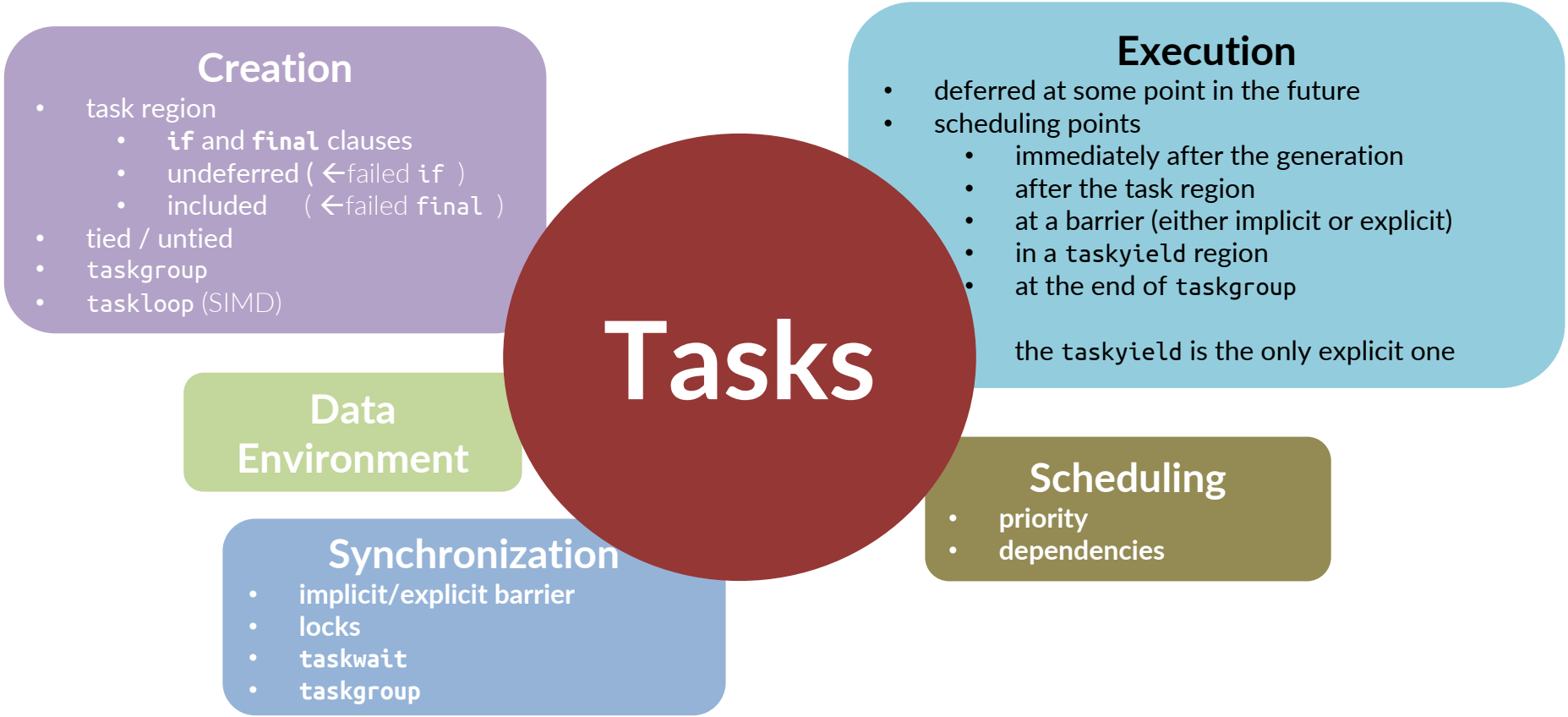
```
#pragma omp parallel proc_bind(close)
 {

  #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
```

taskloop.c

To limit the 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

Key concepts in `tasks` management



that's all folks, have fun