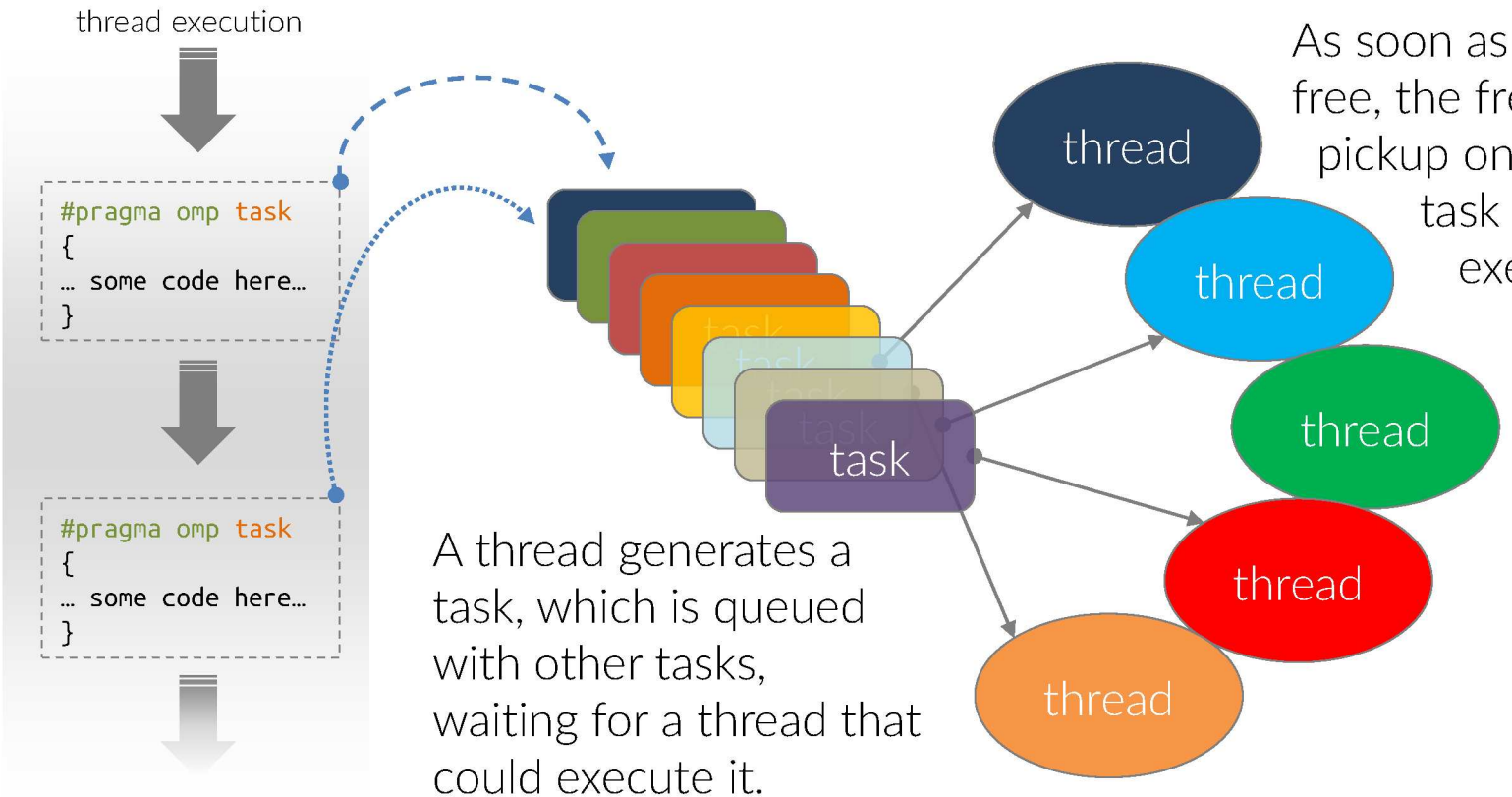
As we have seen in the previous example (`02_sections_nested_irregular.c`), it is sometimes possible to parallelize a workflow which is irregular or runtime-dependent using OpenMP sections.
However, often the solution is quite ugly and convoluted.

Since version 3.0, OpenMP offers a new elegant construct designed for this class of problems: the OpenMP task.

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.

OpenMP tasks



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", 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.

A classical example: traversing a linked list

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

A task is generated for each node that must be processed

The calling thread continues traversing the linked list

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

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

OpenMP `tasks` synchronization


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

    #pragma omp single nowait
    {

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

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

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

    }

    #pragma omp taskwait

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

You need the tasks to be completed *before* to arrive at this point where the final results are accumulated.

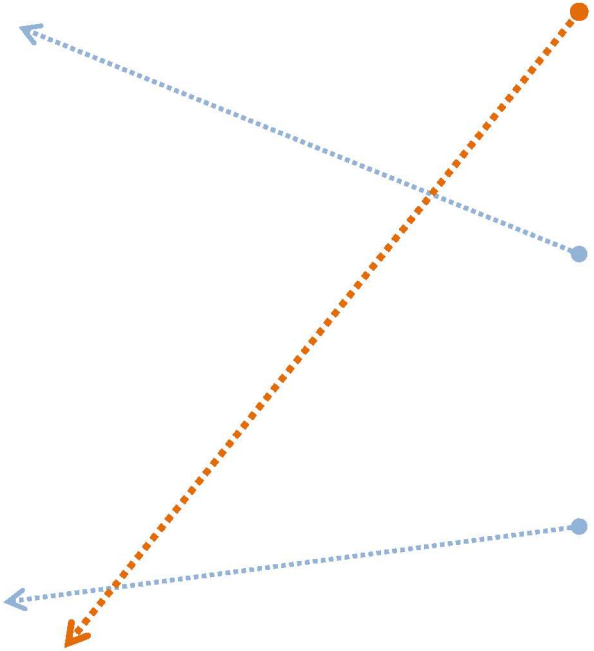
Due to the `nowait` clause, the implied barrier at the end of the `single` region is no respected and the threads are flowing freely beyond that region.

Without the `taskwait` directive, the threads that are not in the `single` region would execute the updates of `result` with meaningless data.

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

finer implementation

OpenMP `tasks` synchronization

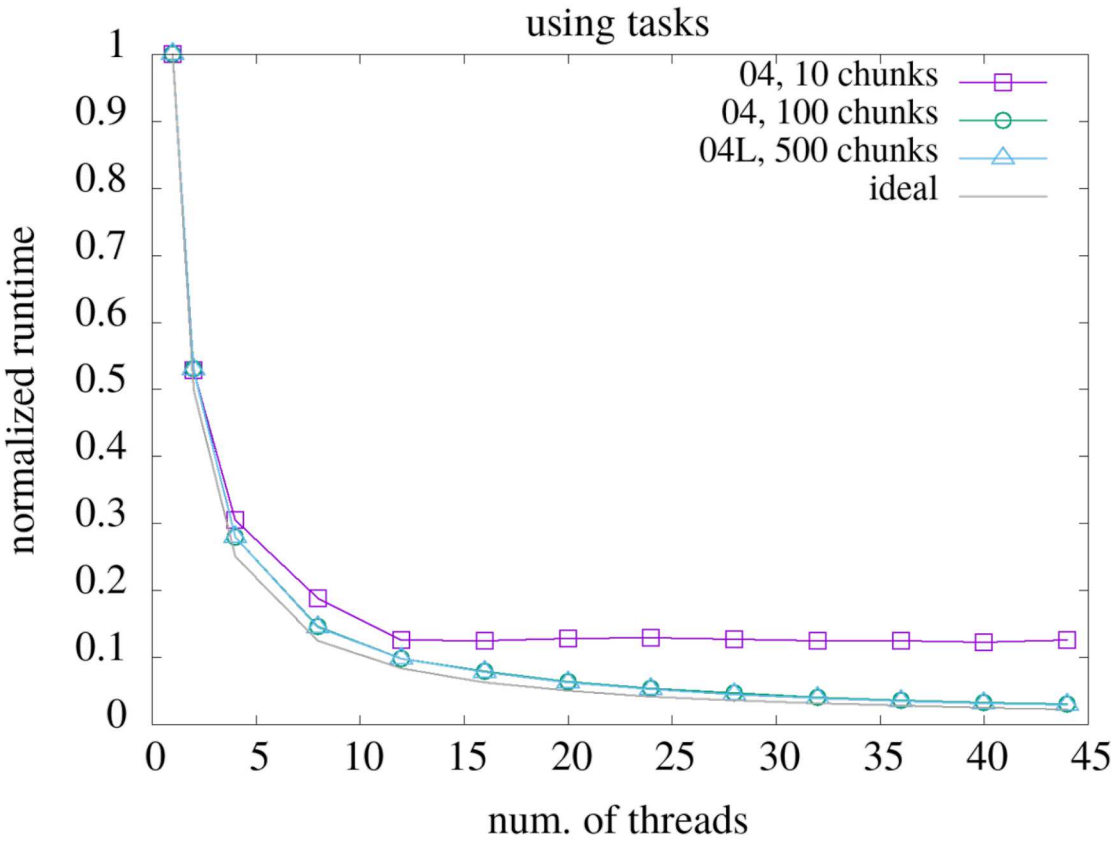


parallel_tasks/
03_tasks_wrong.c

You need the tasks to be completed *before* to arrive at this point where the final results are accumulated.

Due to the `nowait` clause, the implied barrier at the end of the `single` region is no respected and the threads are flowing freely beyond that region.

Without the `taskwait` directive, the threads that are not in the `single` region would execute the updates of `result` with meaningless data.


using tasks
04, 10 chunks
04, 100 chunks
04L, 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

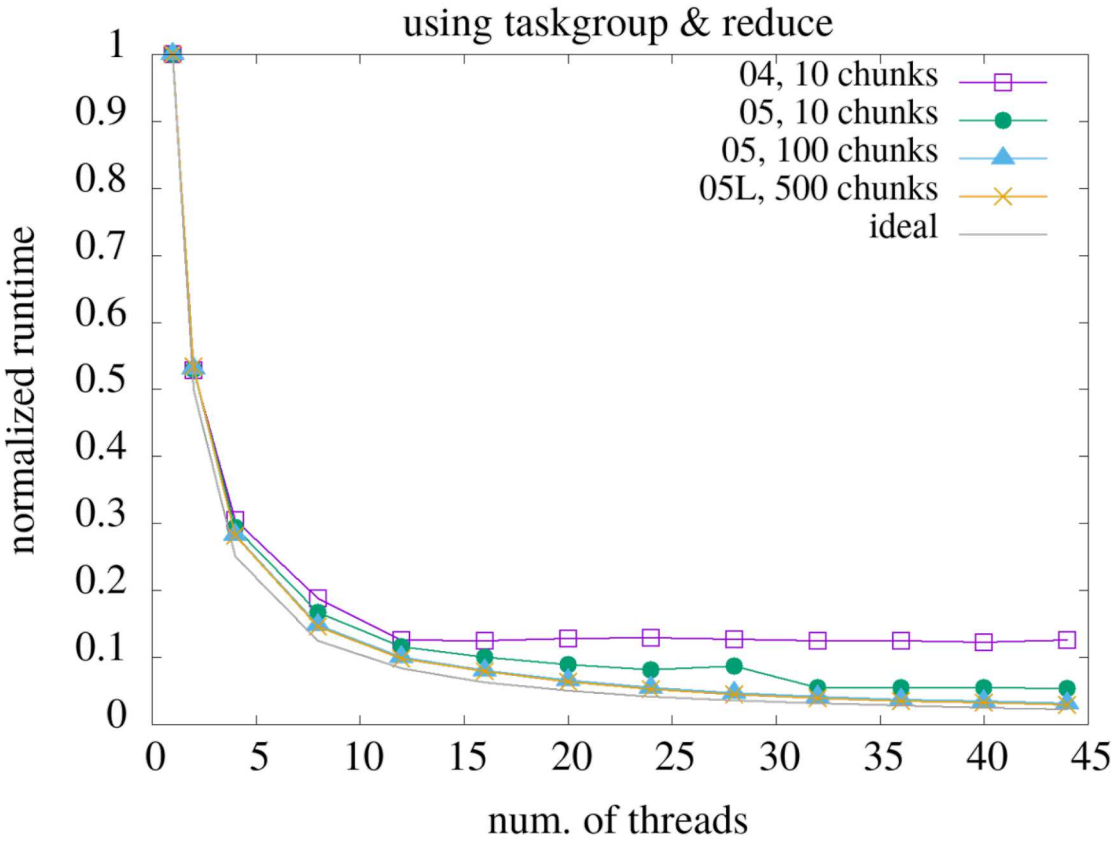
OpenMP `taskgroup`


```
#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 easeof loops

Tasks are created for each iteration

parallel_tasks/
05_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
```

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/
05_task_taskloop.c