Advanced OpenMP
Tasks

SCIENTIFIC &
DATA-INTENSIVE COMPUTING

Luca Tornatore - I.N.A.F.

Advanced HPC 2023-2024 @ Università di Trieste

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

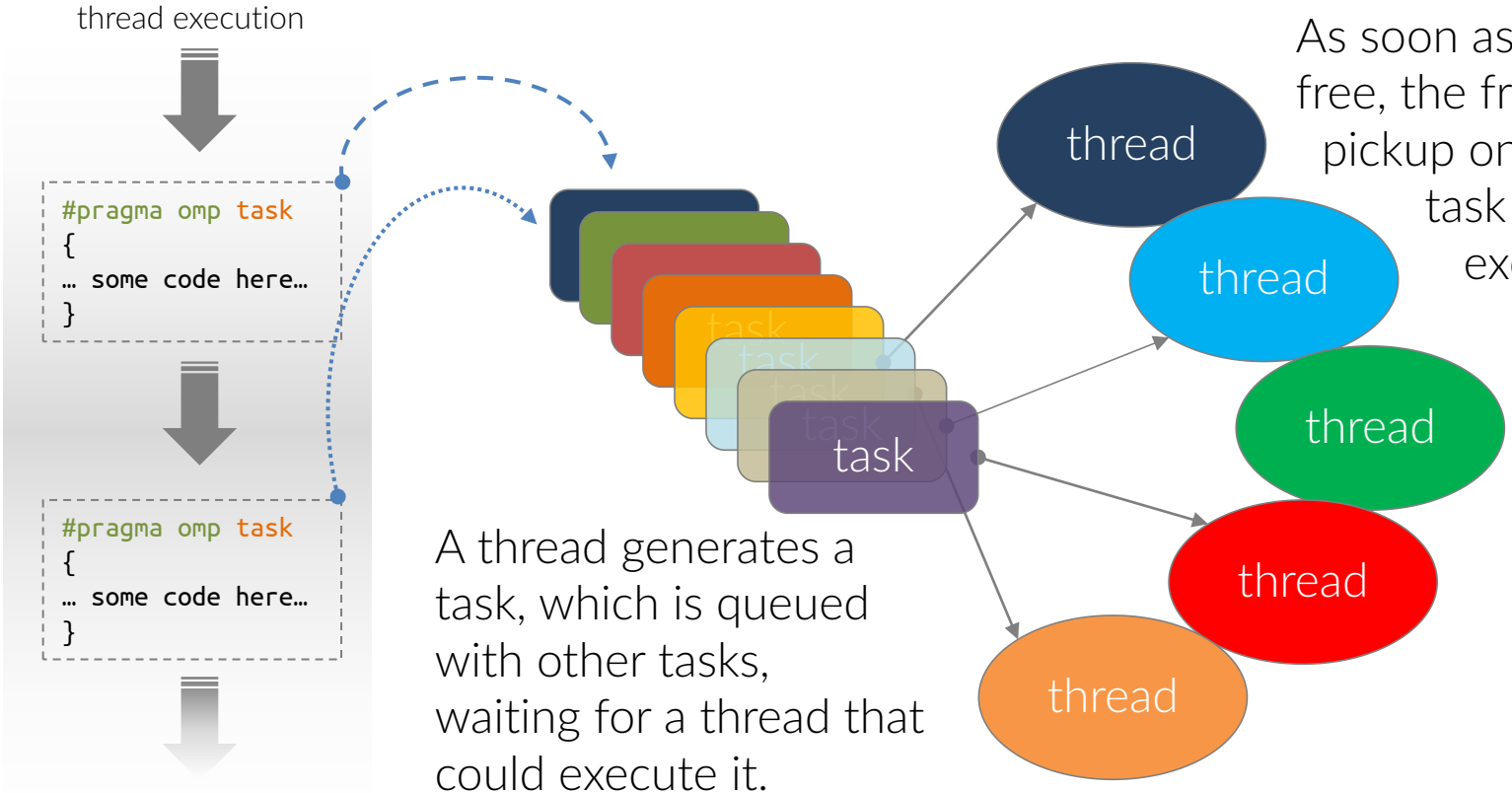
Concurrent access

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**.

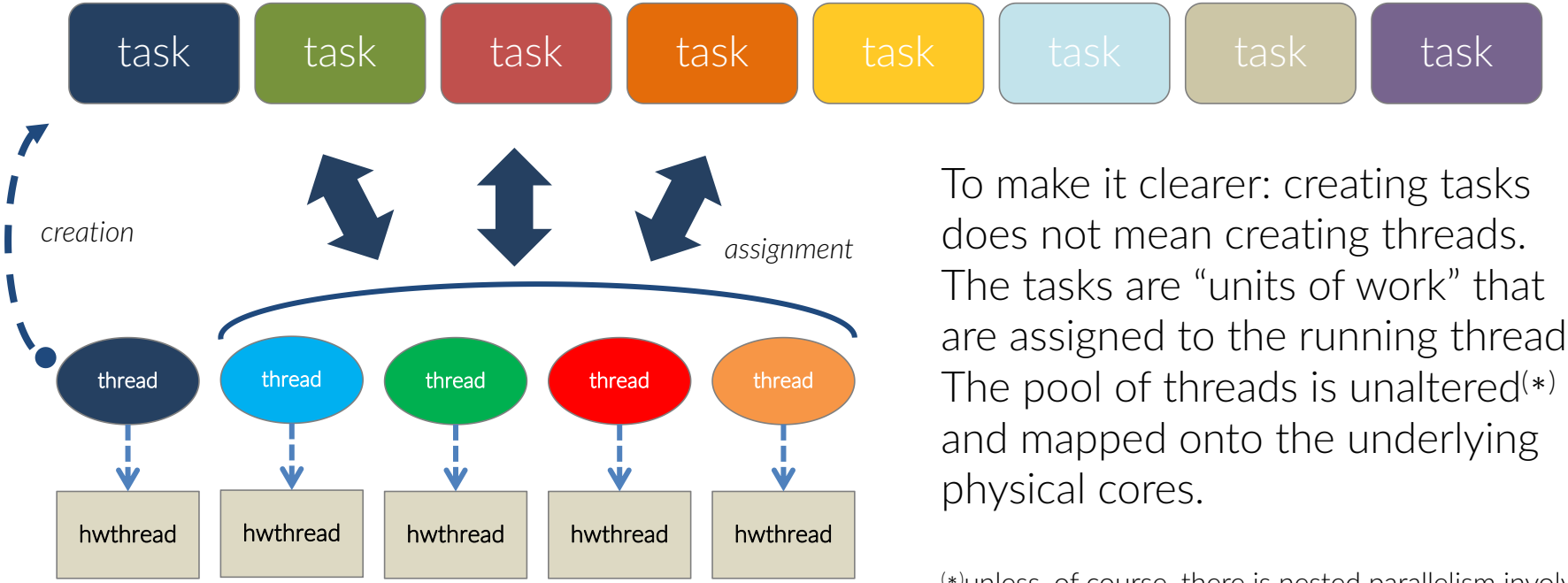
What happens under the hood is that OpenMP creates a “bunch of work” along with the data and the local variables it needs, and *schedules* it for execution at some point in the future.
Then, under the hood, a queuing system orchestrates the assignment of each task to the available threads.

OpenMP tasks



As soon as they are free, the free threads pickup one queued task each, and execute it

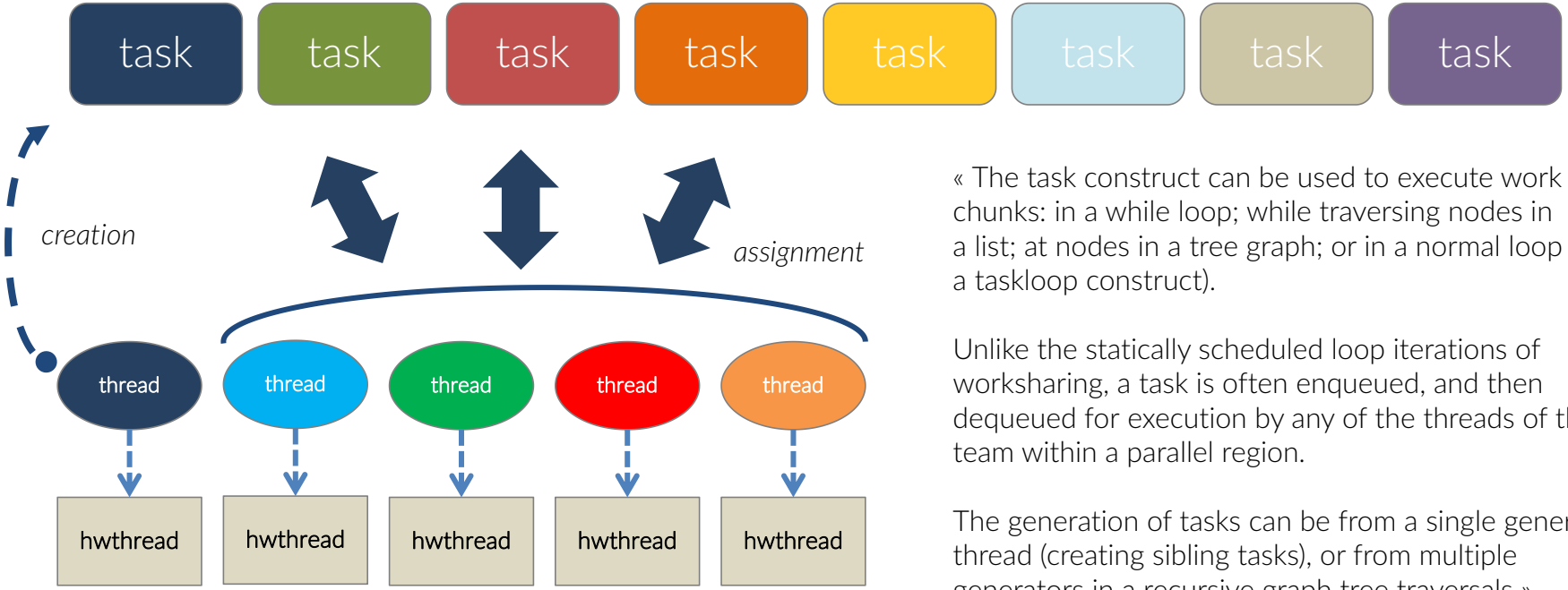
A thread generates a task, which is queued with other tasks, waiting for a thread that could execute it.



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

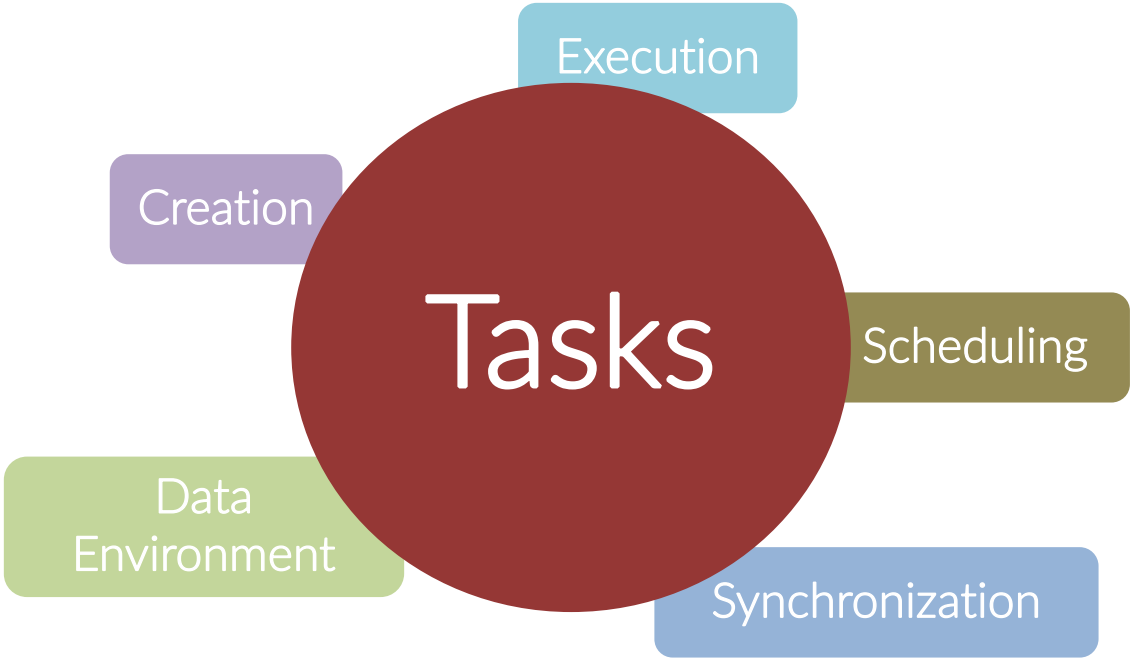
OpenMP `tasks`

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 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.

Key concepts in `tasks` management



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

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

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

example: 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) );
   }
 }
```

03_variable_workload.c

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

example: variable workload

	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.

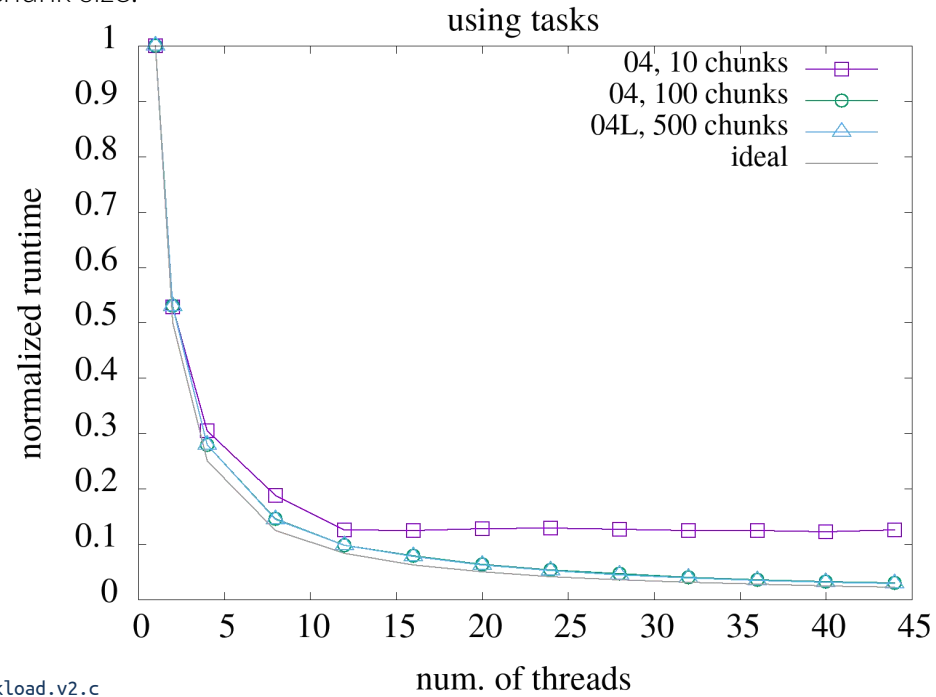
Message II

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.

example: 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
```

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.



parallel_tasks/
03_variable_workload.v2.c

The scope of the 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.

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.

02_tasks.c

```
#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 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.

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

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;
}
```

Unpredictable workload

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

1. Traversing a linked-list
2. solving a graph

Traversing a linked-list

A classical example: traversing a linked list

btw: there is a simpl 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 );
   }
   …;
}
```

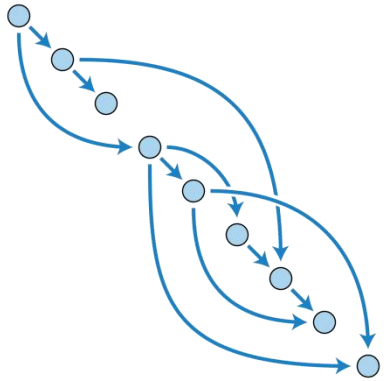
A task is generated for each node that must be processed

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

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

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.

Solving a graph

We'll be using only the following elementary features of OpenMP:

```
#pragma omp parallel
#pragma omp single

#pragma omp task

#pragma omp atomic update
#pragma atomic read
#pragma omp atomic capture
```

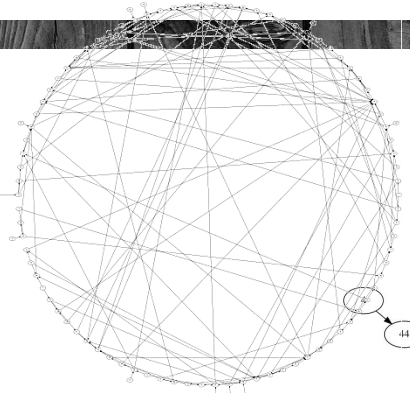
well, ok, there is also a **`taskwait`** directive that we'll see in the synchronization section, but that is just an eye-candy for a `printf`..

Building the DAG

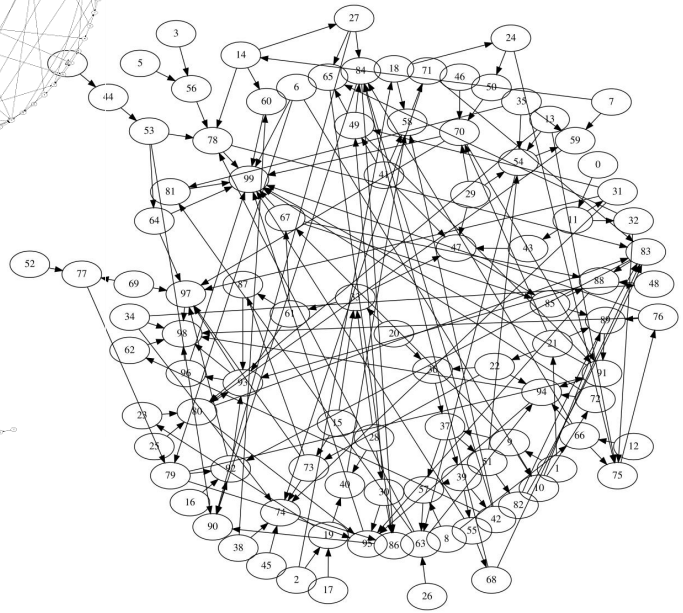
The routine that generates the dag is named `generate_dag()` and it is pretty simple. The free parameters are: the total number of nodes, the number of root nodes, the minimum and maximum number of children per node, and the baseline workload per each node.

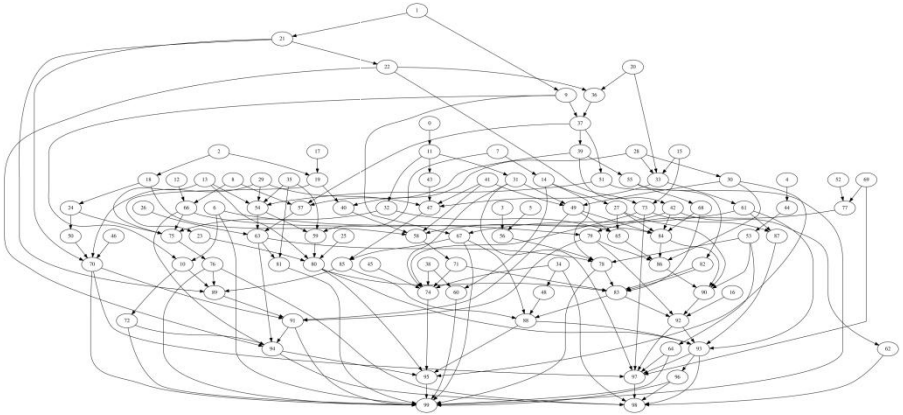
Here you find 3 different representations of the same small dag, having just 100 nodes (you can generate your own using the aforementioned routine).

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



dag.c





Building the DAG

To generate the DAG we choose a quite simple strategy ignoring some marginal issues that are not of major importance here. The comments in `examples_tasks/05_dag.c` should be sufficient to understand the details.

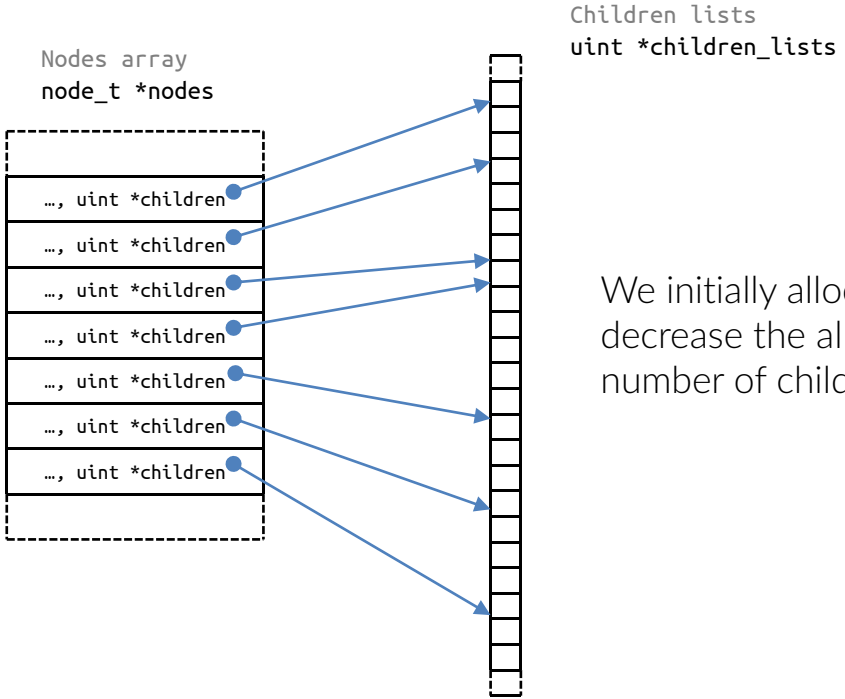
1) The basic point is how to avoid loops inside the directed graph. A simple way to achieve the goal is to enforce that each node of the graph has children that only live "forward" to it.
If we store the *N* nodes in an array, we can implement that by committing each node *i* to have children with an index $j \geq i$.

2) Then for each node *i* we randomly select a number n_i of children among the following *N-i* possible nodes.

3) We save the list of children for each node and we increase by 1 the number of ancestors of each children (note that we do not impose a maximum number of ancestors).

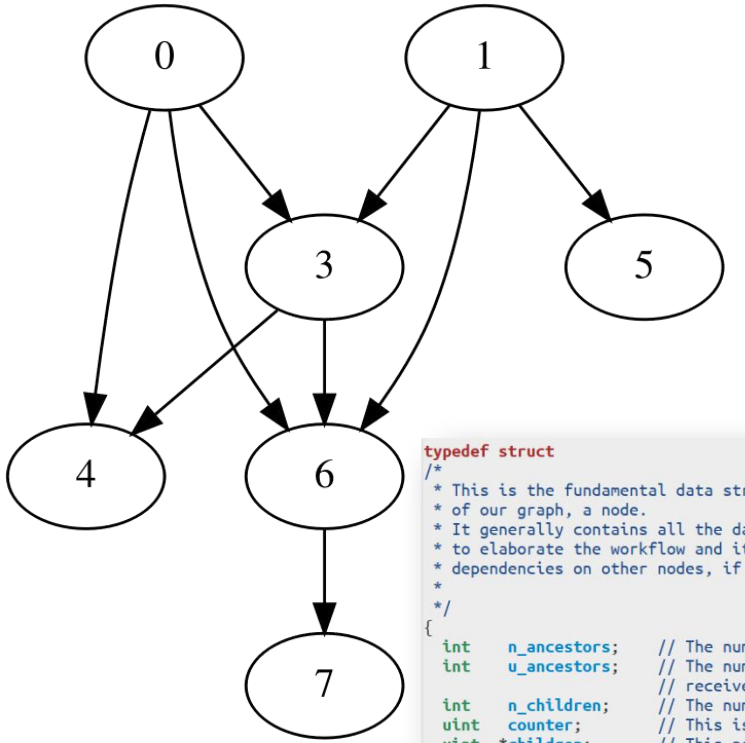
Building the DAG

4) We use a separated memory region to save the list of children of each node, since the lists have different lengths.



We initially allocate room for N*max_children, and we decrease the allocation size at the end when the actual number of children is known.

The general strategy




```
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 propagates 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.

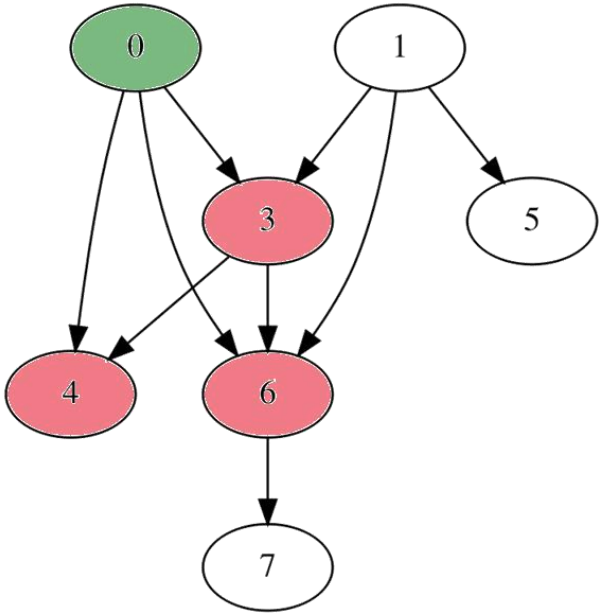
Notice that it is totally impossible to forecast what will be the execution pattern before the nodes are created.

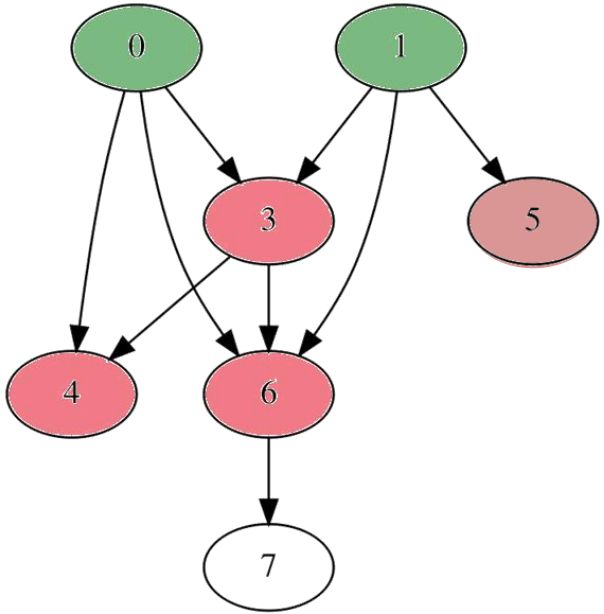
The general strategy

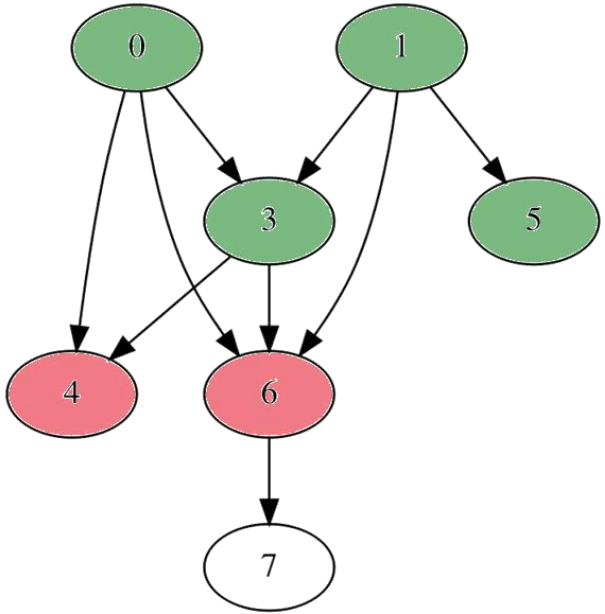
The root nodes are initialized, let's say we start from 0.
It updates its own descendants, that are then partially updated.

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

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







Generating the tasks

1) We initialize separately the pseudo-random number generators for each thread

2) For each root node, a random initial workload is generated.

3) A task is generated for each root node, by calling **`update_node()`** with that root node as target.

```
#pragma omp parallel shared(seeds, done)
{

  int me = omp_get_thread_num();

  // each thread initializes the seeds
  // for the random number generation
  //
  for ( int s = 0; s < 3; s++ )
#if !defined(REPRODUCIBLE)
    seeds[me][s] = me*123+s;
#else
    seeds[me] = 123*(s+1)*10;
#endif
  seed48((seeds_pt)&seeds[me]);

  // the region that generates tasks
  //
  #pragma omp single
  {
    for ( int j = 0; j < dag->N_roots; j++ )
      {
        uint work = dag->workload / (nodes[j].n_children+1);
       #if !defined(REPRODUCIBLE)
        work = dag->workload / 100 + nrand48((seeds_pt)&seeds[omp_get_thread_num()]) % work;
       #endif
        nodes[j].counter = work;

        // here a task is generated because this is a
        // root node and so it is ready to update
        // at this point
       #pragma omp task
        update_node( nodes, &nodes[j], &done, dag->workload );
      }

   #pragma omp taskwait
    PRINTF("- thread %d has generated the first %d tasks for the root nodes;\n"
           "  tasks have been completed, now it is joining the pool\n",
          me, dag->N_roots );
  }
  // end of task generation

}
```

Generating the tasks

Inside **`update_node()`**, each task

1) Determines a random amount of work to be propagated to the children

2) Upgrades the children by modifying both the workload (the **`counter`** variable) *and* the **`u_ancestors`** variable which controls whether a node is ready for computation

3) If it was the last ancestors updating a node, it creates a new task for that node by calling the same **`update_node()`** with that node as target.

4) Performs the calculation for its target node.

```
void update_node( node_t *nodes, node_t *node, uint *check, uint workload )
{

 uint work = workload / (node->n_children+1);
#if !defined(REPRODUCIBLE)
 work = workload / 100 + nrand48((seeds_pt)&seeds[omp_get_thread_num()]) % work;
#endif

 // now let's get through the edges
 // to update each dependent node
 //
 for ( int j = 0; j < node->n_children; j++ )
   {
    int u_ancestors;
    uint idx = node->children[j];

   #pragma omp atomic update                          // update the children's work counter
    nodes[idx].counter += work;
    /*
   #pragma atomic update
    ++nodes[idx].u_ancestors;
   #pragma atomic read
    u_ancestors = nodes[idx].u_ancestors;
    */
   #pragma omp atomic capture                         // notify that I did update and capture
    u_ancestors = ++nodes[idx].u_ancestors;           // the u_ancestors value immediately
                                                      // afterwards

    if ( nodes[idx].n_ancestors - u_ancestors == 0 )  // I was the last one to update
     #pragma omp task                                 // as such, I do create a task for this node
      update_node( nodes, &nodes[idx], check, workload );
   }

 node->result = heavy_work( node->counter);           // compute my work and save it for future usage
#pragma omp atomic update                             // just a check: increase the counter of
 (*check)++;                                          // how many nodes have been computed

 // reset the node for a next processing cycle
 //
 node-> u_ancestors = 0;
 //node-> counter     = 0;
```

Generating the tasks: note 1

The usage of this `atomic capture` is an important detail to discuss.
Let's understand it more deeply.

What we want is that the last task updating the node starts a new task having that node as a target.
A task knows it is the last one because when it updates **`u_ancestors`** the condition

`u_ancestors == a_ancestors-1`

holds.

What would happen if we used a different way to read the value of `u_ancestors` ?

```
void update_node( node_t *nodes, node_t *node, uint *check, uint workload )
{

 uint work = workload / (node->n_children+1);
#if !defined(REPRODUCIBLE)
 work = workload / 100 + nrand48((seeds_pt)&seeds[omp_get_thread_num()]) % work;
#endif

 // now let's get through the edges
 // to update each dependent node
 //
 for ( int j = 0; j < node->n_children; j++ )
   {
    int u_ancestors;
    uint idx = node->children[j];

   #pragma omp atomic update                          // update the children's work counter
    nodes[idx].counter += work;
    /*
   #pragma atomic update
    ++nodes[idx].u_ancestors;
   #pragma atomic read
    u_ancestors = nodes[idx].u_ancestors;
    */
   #pragma omp atomic capture                         // notify that I did update and capture
    u_ancestors = ++nodes[idx].u_ancestors;           // the u_ancestors value immediately
                                                      // afterwards

    if ( nodes[idx].n_ancestors - u_ancestors == 0 )  // I was the last one to update
     #pragma omp task                                 // as such, I do create a task for this node
      update_node( nodes, &nodes[idx], check, workload );
   }

 node->result = heavy_work( node->counter);           // compute my work and save it for future usage
#pragma omp atomic update                             // just a check: increase the counter of
 (*check)++;                                          // how many nodes have been computed

 // reset the node for a next processing cycle
 //
 node-> u_ancestors = 0;
 //node-> counter     = 0;
```

What would happen if we used a different way to read and update the value of `u_ancestors ?`

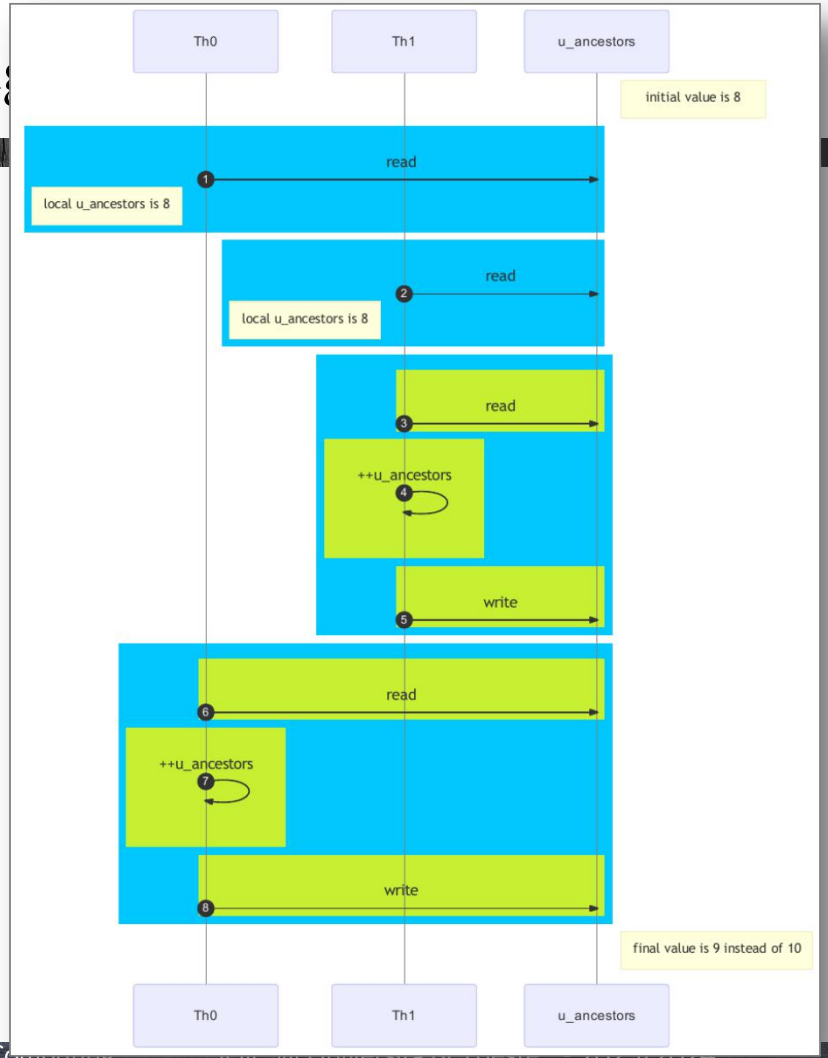
Give that the operation `++u_ancestors` requires 3 steps, namely

1. read the current value of `++u_ancestors`;
2. increase the value;
3. write back the updated value,

let's say that we coded that capture operation in a different way, for instance:

```
#pragma omp atomic read
 u_ancestor = nodes[idx].u_ancestors;
#pragma omp atomic update
 ++nodes[idx].u_ancestors;
```

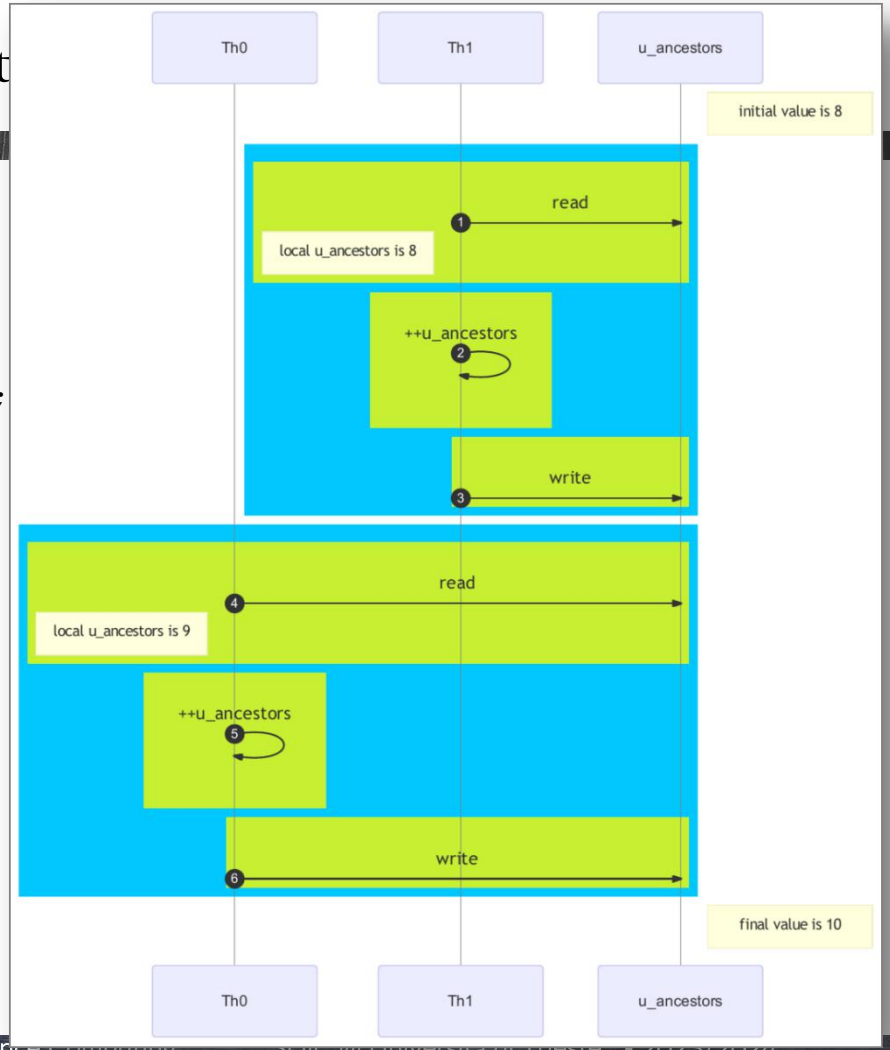
That could easily result the sequence presented here on the right (blue regions represent "exclusive accesses" – i.e. `omp atomic` – to `u_ancestors`).
Both thread 0 (Th0) and thread 1 (Th1) are convinced to be the 9th and none of the two realizes to be the 10th.
Then, the corresponding task for the node being updated is never created



Instead, with the given implementation

```
#pragma omp atomic capture
 u_ancestors = ++nodes[idx].u_ancestors;
```

the access to `u_ancestors` is secured and one of the two threads realizes to be the 10th and creates the corresponding task.



Generating the tasks: note 2

```
if ( nodes[idx].n_ancestors - u_ancestors == 0)
 #pragma omp task
  update_node( nodes, &nodes[idx], check, workload );
```

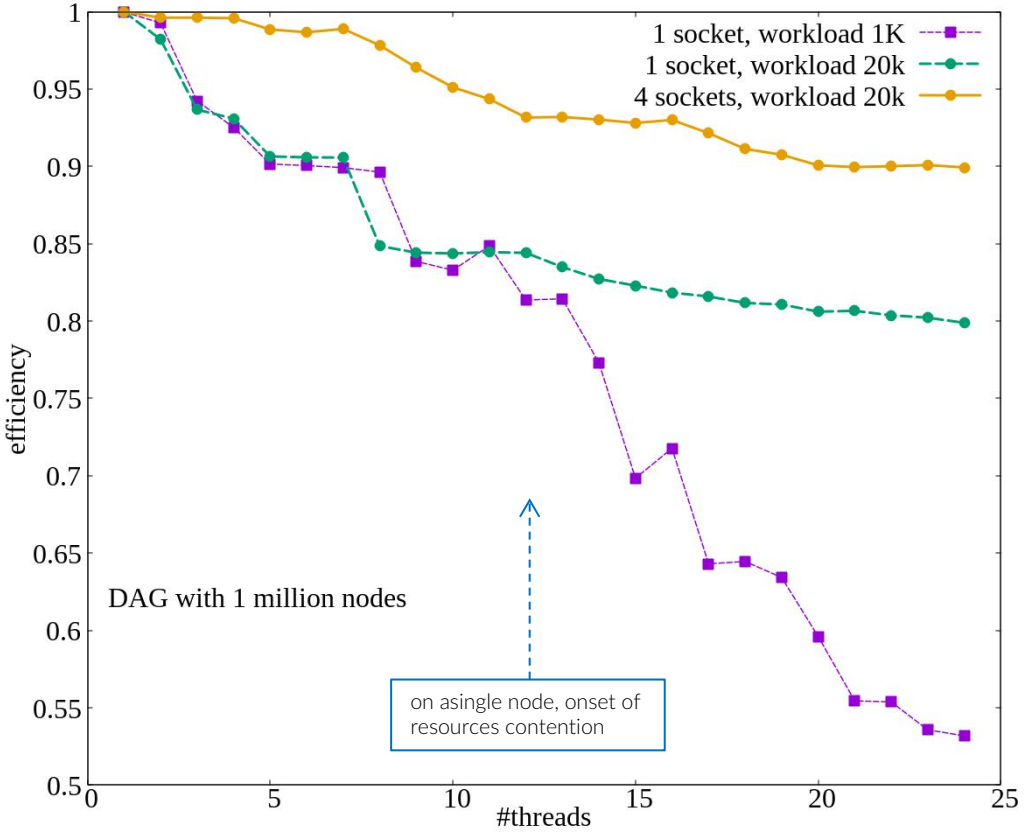
The task creation is *not* a recursive call to `update_node()`.
In fact, recursion happens when the call (i.e. a code jump and the relative creation of the stack) happens in that very moment, and the stack of the called function lives right under the stack of the caller function.

What happens at the moment of the task creation is somehow similar to the creation of a "description" of a bunch of work: imagine that the creating thread sends to the task queuing system a note like

« tell to the thread that will be assigned to this task: call the `update_node()` function with the following arguments: < `nodes, &nodes[idx], check, workload` > »

where the embedded value of `idx` is the value at the moment of the task creation (and the same holds of course for all the other variables, which however in this example do not change).

efficiency



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
```

small work → } 1 socket

large work { ← 4 sockets

"small work" ~30sec for a single thread
"large work" ~10min for a single thread

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

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

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

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
```

`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
```

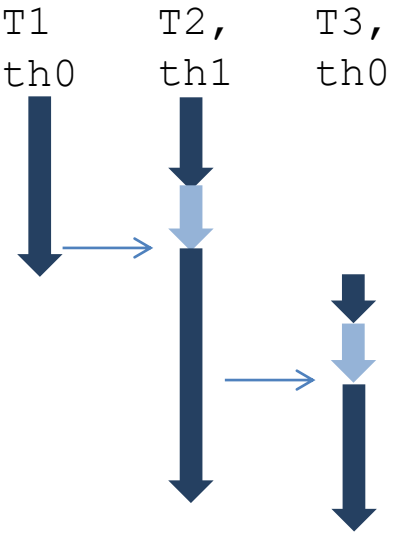
task 1

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

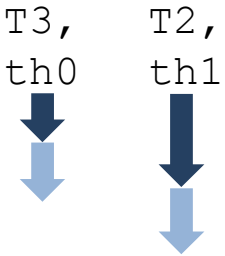
task 2

```
... ;  // 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.

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

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

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() );
}
```

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`


```
double gravity_tree ( particle_t *p, tree_t *tree )
{
  double gravity_force = 0;
 #pragma omp taskgroup task_reduction(+: gravity_force)
  {

    while(  )
    {
      #pragma omp task in_reduction(+: res)
        res += sum_up_data();
    }
}
```

An interesting feature coupled with the `taskgroup` construct, is the `task_reduction,` that allows a reduction operation among tasks declared with an `in_reduction` clause

Synchronizing `tasks:taskgroup`

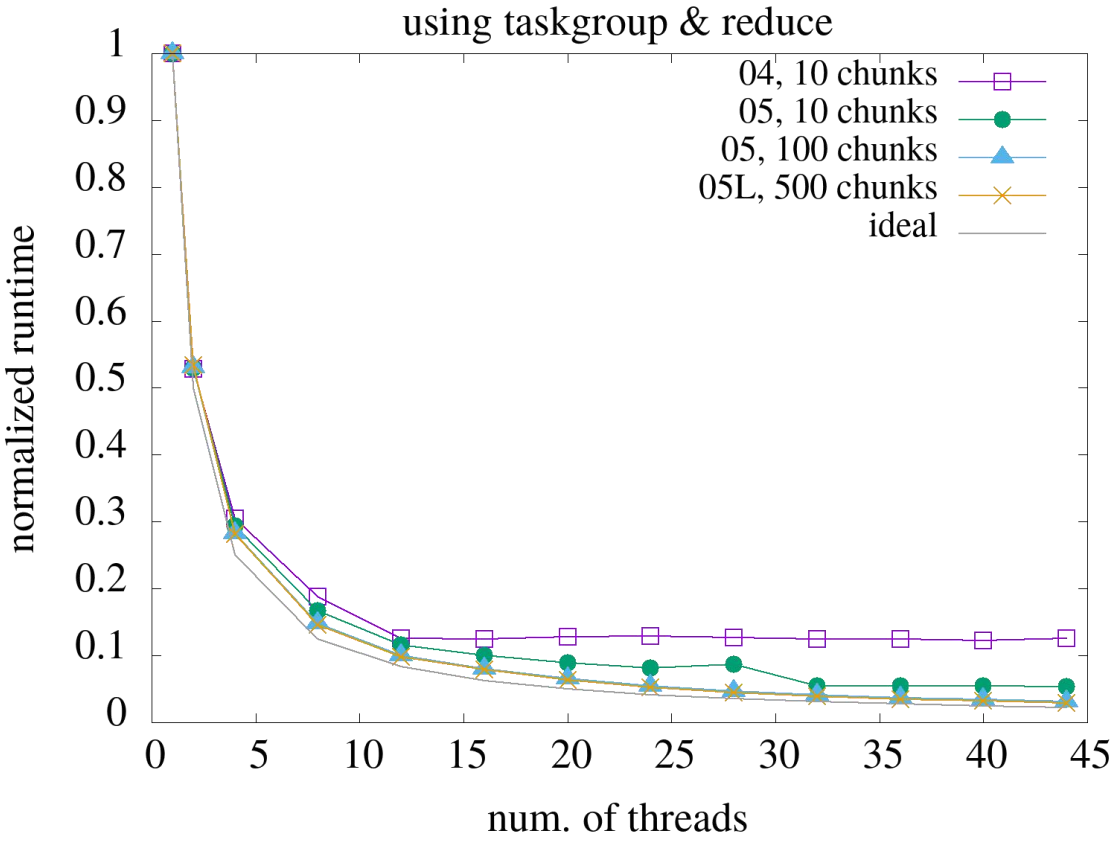

```
#pragma omp parallel proc_bind(close)
  {

    #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
```

05_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

Synchronizing tasks:taskgroup



04_tasks_reduction.c

05_taskgroup_reduction.c

An additional mechanism that you can use to manage the thread (and task) synchronization is the **lock**, via the **omp_lock_t** type.
A lock may be nested: **omp_nest_lock_t**.

`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;
```

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

Building a heap with a double-linked list

The code here on the right builds a double-linked list by inserting the new data (int values) with a total order. Walking the list the data will always be presented in asceding order.

Our scope here is to implement the same functionality using tasks.

The node data structure that we adopt is:

```
typedef struct llnode
{
  int data;
 #if defined(_OPENMP)
  omp_lock_t lock;
 #endif

  struct llnode *next;
  struct llnode *prev;
} llnode_t;
```

```
int find_and_insert( llnode_t *head, int value )
{
  if ( head == NULL )
    return -1;

  llnode_t *ptr  = head->next;
  llnode_t *prev = head;
  while ( (ptr != NULL) && (ptr->data < value) )
    {
      prev = ptr;
      ptr  = ptr->next;
    }

  llnode_t *new = (llnode_t*)malloc( sizeof(llnode_t) );
  if ( new == NULL )
    return -2;

  new->data = value;
  new->prev = prev;
  new->next = ptr;
  prev->next = new;

  return 0;
}
```

linked_list.c

note: the code shown here only works in forward directions, i.e. the very first node may not be the smallest one. For the complete code look at the example source.

Building a heap with a double-linked list

First, let's analyze the problem, assuming that we'll generate a task for every new insertion, as depicted in the code snapshot in the right.

N.B. for the sake of simplicity we'll allocate the memory needed for a new node at the moment of insertion; please account for the fact that this *may no be* the best way.

```
#pragma omp parallel
{
  me = omp_get_thread_num();
  #pragma omp single
  {
    printf("running with %d threads\n", omp_get_num_threads());
    int n = 1;

    while ( n < N )
      {
        int new_value = rand();

       #pragma omp task
        find_and_insert_parallel( head, new_value, mode );

        n++;
      }
  }
}
```

Building a heap with a double-linked list

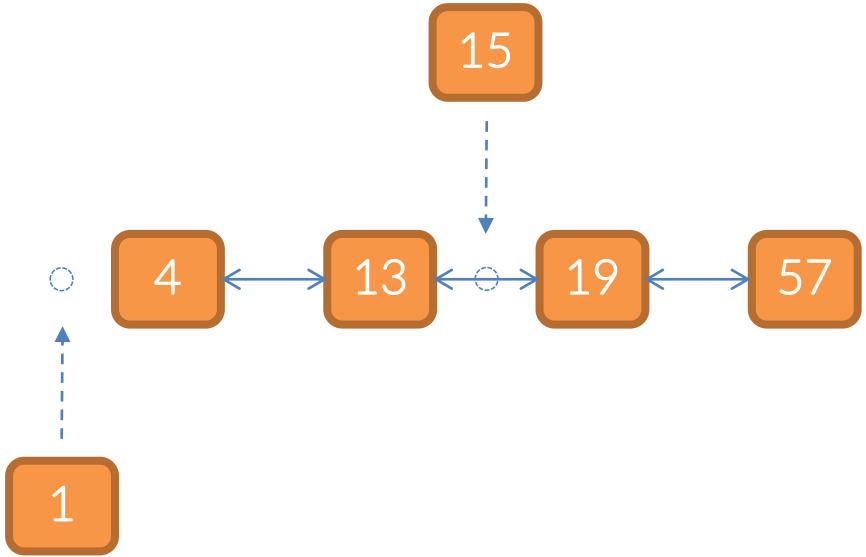
The very first step for a new ordered insertion will be to find the Left and Right nodes that are the largest smaller and the smallest largest than the new value to be inserted.
For the sake of simplicity our data we'll be integers.

In the example on the right, 13 and 19 we'll be the Left and Right nodes for 15, while 4 we'll be the Right one for 1, which, in turn, we'll have a NULL Left node.

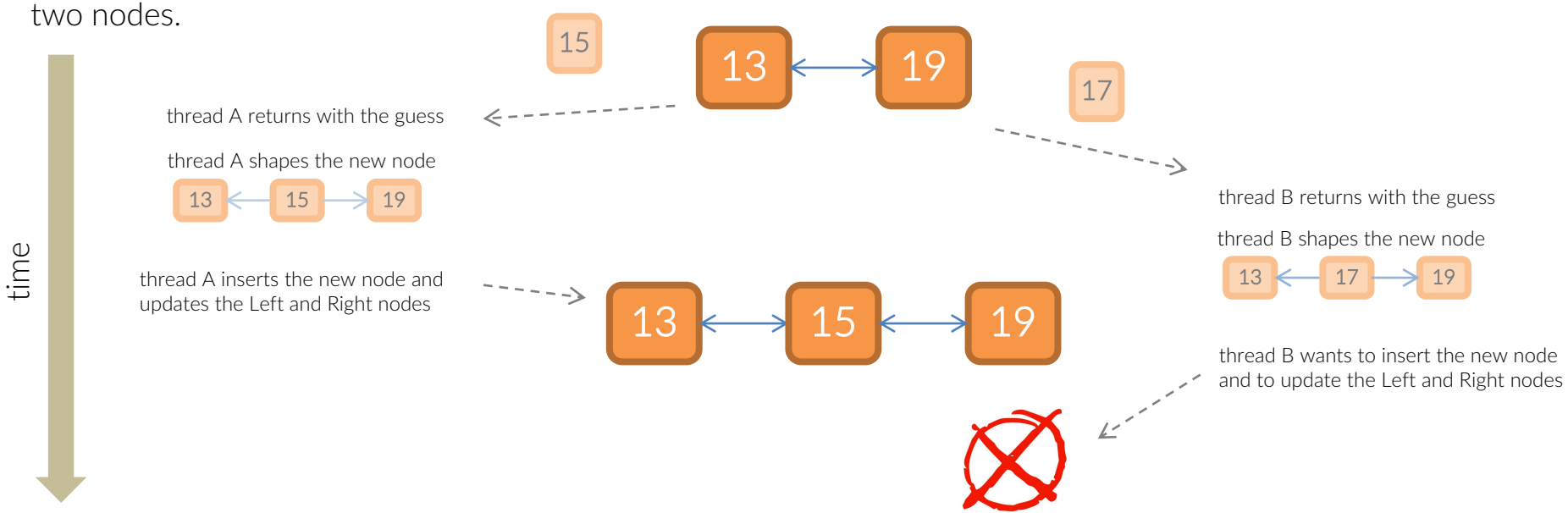
In the example source file linked_list.c, the routine that accomplish this task is

```
int find ( llnode_t *head, int value,
           llnode_t **prev, llnode_t **next );
```

that returns in ***prev** and ***next** the pointers to the Left and Right nodes of the new **value**; **head** is the pointer of a starting point (it is not needed that it to be the head of the list)



The Left and Right nodes returned by the search are just a first guess: in fact, meanwhile the thread walked the tree and returned with the result, some other thread may have inserted new data in between of the two nodes.



Building a heap with a double-linked list

Hence, when the thread returns from the search with the pointers to the Left and Right nodes, it has to check whether they are still contigous nodes (or that they are still the head or tail of the list).

In turn, before checking, it will have to acquire the locks of both of them (or of just one of them if it arrived at the head or at the tail of the list); that is mandatory because otherwise another thread may be able to update their **`prev`** and **`next`** pointers.

After that, if both the conditions (either prev or next may be NULL if the value to be inserted was the smallest or the largest at the moment of the search)

```
prev->next = next
next->prev = prev
```

are met, the thread can safely insert the new node and release the locks.

What if that is not the case, like for the **`thread B`** in the previous slide ?

Building a heap with a double-linked list

If some new nodes have been inserted in between of **prev** and **next**, additional operations are needed.
Two symmetric situations may be at stake:

```
A) (prev != NULL) && (prev-> next != next)
```

prev exists, but a different node is its new **next** node
the thread will start from the **prev**, which is still a valid guess, to walk ahead until it finds the first node whose key is larger than the value to be inserted

```
B) (next != NULL) && (next->prev != prev)
```

prev was NULL (so we were at the list's head), but the **next** has a non-NULL **prev** node
the thread will start from the **next**, which is still a valid guess, to walk back until it finds the first node whose key is smaller than the value to be inserted

Building a heap with a double-linked list

The algorithms that solves A) and B) are perfectly symmetric. As such, let's describe how to solve A.

first, release the old **next** lock, not to block other threads

start the walk ahed, from the valid **prev**

acquire the lock on the current **next**

exit if the **next**'s key is larger than value; at this moment. the thread owns both the locks at **prev** and **next**

the **(prev,next)** pair not found yet; release the **next**'s lock

walk on: **prev** becomes **next**, **next** becomes **next->next**

```
if( (prev != NULL) && (prev-> next != next) )
  {

    if (next != NULL)
      omp_unset_lock(&(next->lock));

    next = prev->next;
    while(next)
      {
        omp_set_lock(&(next->lock));
        if( next->data >= value )
          break;
        omp_unset_lock(&(prev->lock));
        prev = next;
        next = next->next;
      }
  }
```

Building a heap with a do

The core of the **`find_and_insert`** is shown in the snapshot on the right.

Right after it, one should the insertion code and the code to release the locks.

However, the code deployed in `linked_list.deadlock.c` leads sometimes to a deadlock.

Can you figure out why ?

```
int find_and_insert_parallel( llnode_t *head, int value, int use_taskyield )
{
  find ( head, value, &prev, &next );

  if( prev != NULL )
    omp_set_lock(&(prev->lock));

  if( next != NULL )
    omp_set_lock(&(next->lock));

  if( ( (prev != NULL) && (prev-> next != next) ) ||
      ( (next != NULL) && (next-> prev != prev) ) )
    {

      if( (prev != NULL) && (prev-> next != next) )
        {
          if (next != NULL)
            omp_unset_lock(&(next->lock));

          next = prev->next;
          while(next)
            {
              now = CPU_TIME % TIME_CUT;
              omp_set_lock(&(next->lock));
              if( next->data >= value )
                break;
              omp_unset_lock(&(prev->lock));
              prev = next;
              next = next->next;
            }
        }

      if( next->prev != prev )
        {
          if (prev != NULL)
            omp_unset_lock(&(prev->lock));

          prev = next->prev;
          while(prev)
            {
              now = CPU_TIME % TIME_CUT;
              omp_set_lock(&(prev->lock));
              if( prev->data <= value )
                break;
              omp_unset_lock(&(next->lock));
              next = prev;
              prev = prev->prev;
            }
        }
    }
```

linked_list.deadlock.c

Building a heap with a double-linked list

To understand a typical configuration that throws the previous code in a deadlock, let's consider the inital state

time

D is the list's head

thread y finds guess for C
LEFT = NULL , RIGHT = D

thread x inserts A

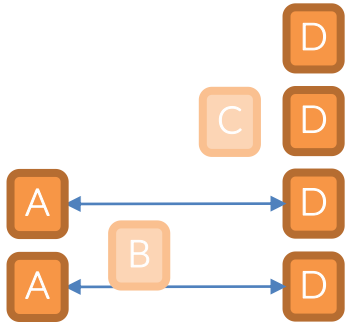
thread z finds guess for B
LEFT = A, RIGHT = D

thread y locks D

thread z locks A

thread y niotices that D→prev ≠ NULL and starts walking back
thread y requires lock for A

thread z requires lock for D



linked_list.deadlock.c

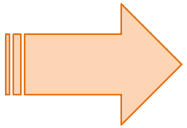
That is the typical deadlock situation:
`thread y` enters in the blocking
`omp_set_lock( A )`
and thread z enters in the blocking
`omp_set_lock( D )`

SInce A belongs to z and D belongs to y, they will forever wait for each other

Building a heap with a double-linked list

A solution of the issue is conveyd in `linked_list.c` : it consists in a slightly more complex sequence to acquire the lock of **`prev`** and **`next`**

```
if( prev != NULL )
  omp_set_lock(&(prev->lock));

if( next != NULL )
  omp_set_lock(&(next->lock));
```



```
int locks_acquired = 0;
while( !locks_acquired )
  {
    if( prev != NULL )
      {
        omp_set_lock(&(prev->lock));
        locks_acquired = 1;
      }

    if ( next != NULL )
      {
        locks_acquired = omp_test_lock(&(next->lock));
        if( !locks_acquired && (prev!=NULL) )
          omp_unset_lock(&(prev->lock));
      }
  }
```

linked_list.deadlock.c

linked_list.c

Controlling the task creation
Clauses

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, ...
  { ... }
```

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 new task is 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.
`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 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;
}
```

partitioning

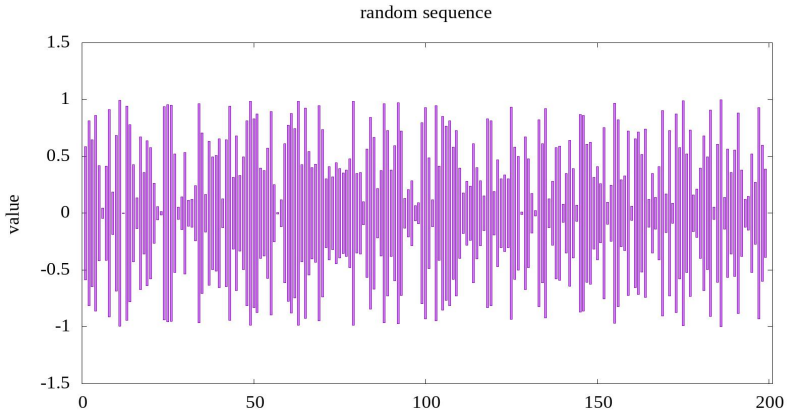
```
void quicksort( data_t *data, int low, int high )
{
  if ( low < high ) {
    int p = partition ( data, low, high );

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

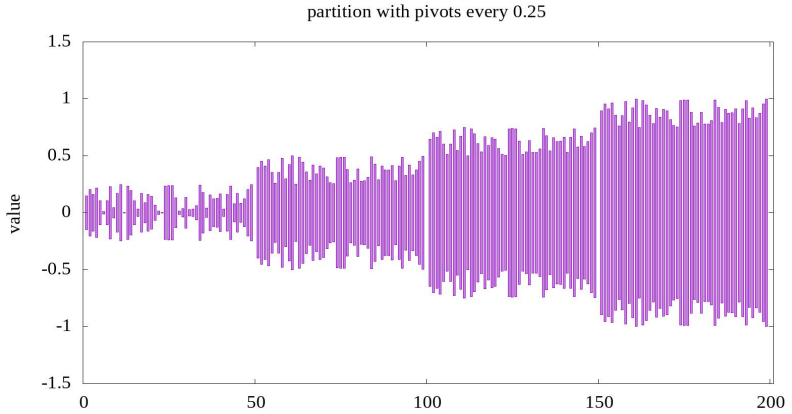
The `partition` function divides the array data in (hopefully) 2 sections.
It individuates the (hopefully) median element `p`, and move all the entries `a[i]<p` in the left part and all the entries `a[i]≤p` in the right part.

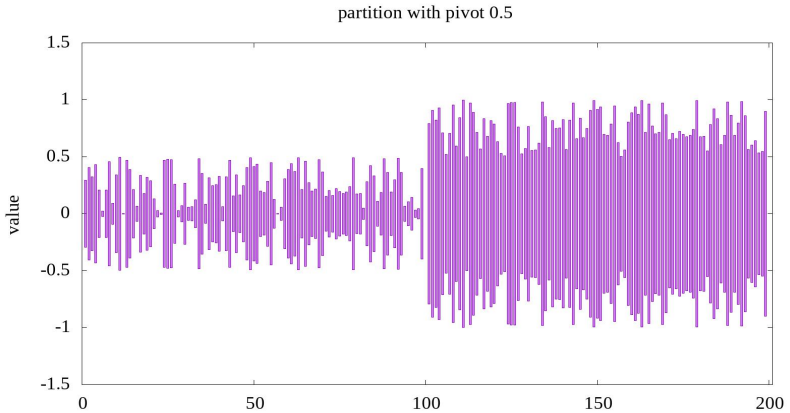
There are *lots* of subtleties to consider and tricks to implement in order to make this algorithm as efficient as possible, but the big picture is the one we have just seen.

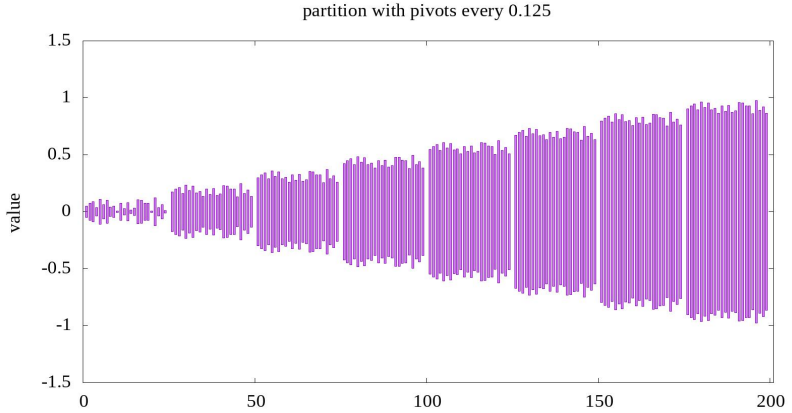
It performs as *NlogN* in the average case, and as N^2 in the worst case (can you figure out which is the worst case?)

partitioning









partitioning is at the core of the *divide-et-impera* strategy of the QuickSort algorithm

```
inline int partitioning( data_t *data, int start, int end, compare_t cmp_ge )
{
  --end;
  void *pivot = (void*)&data[end];

  int pointbreak = end-1;
  for ( int i = start; i <= pointbreak; i++ )
    if( cmp_ge( (void*)&data[i], pivot ) )
      {
        while( (pointbreak > i) && cmp_ge( (void*)&data[pointbreak], pivot ) ) pointbreak--;
        if (pointbreak > i ) {
        SWAP( (void*)&data[i], (void*)&data[pointbreak], sizeof(data_t) );
        pointbreak--; }
      }
  pointbreak += !cmp_ge( (void*)&data[pointbreak], pivot ) ;
  SWAP( (void*)&data[pointbreak], pivot, sizeof(data_t) );

  return pointbreak;
}


void pqsort( data_t *data, int start, int end, compare_t cmp_ge )
{

  int size = end-start;
  if ( size > 2 )
    {
      int mid = partitioning( data, start, end, cmp_ge );

     #pragma omp task shared(data) firstprivate(start, mid)
      pqsort( data, start, mid, cmp_ge );
     #pragma omp task shared(data) firstprivate(mid, end)
      pqsort( data, mid+1, end , cmp_ge );
    }
  else
    {
      if ( (size == 2) && cmp_ge ( (void*)&data[start], (void*)&data[end-1] ) )
        SWAP( (void*)&data[start], (void*)&data[end-1], sizeof(data_t) );
    }
}
```

Let's consider a first simple omp implementation

day26/examples_tasks/
08_quicksort.v0.c

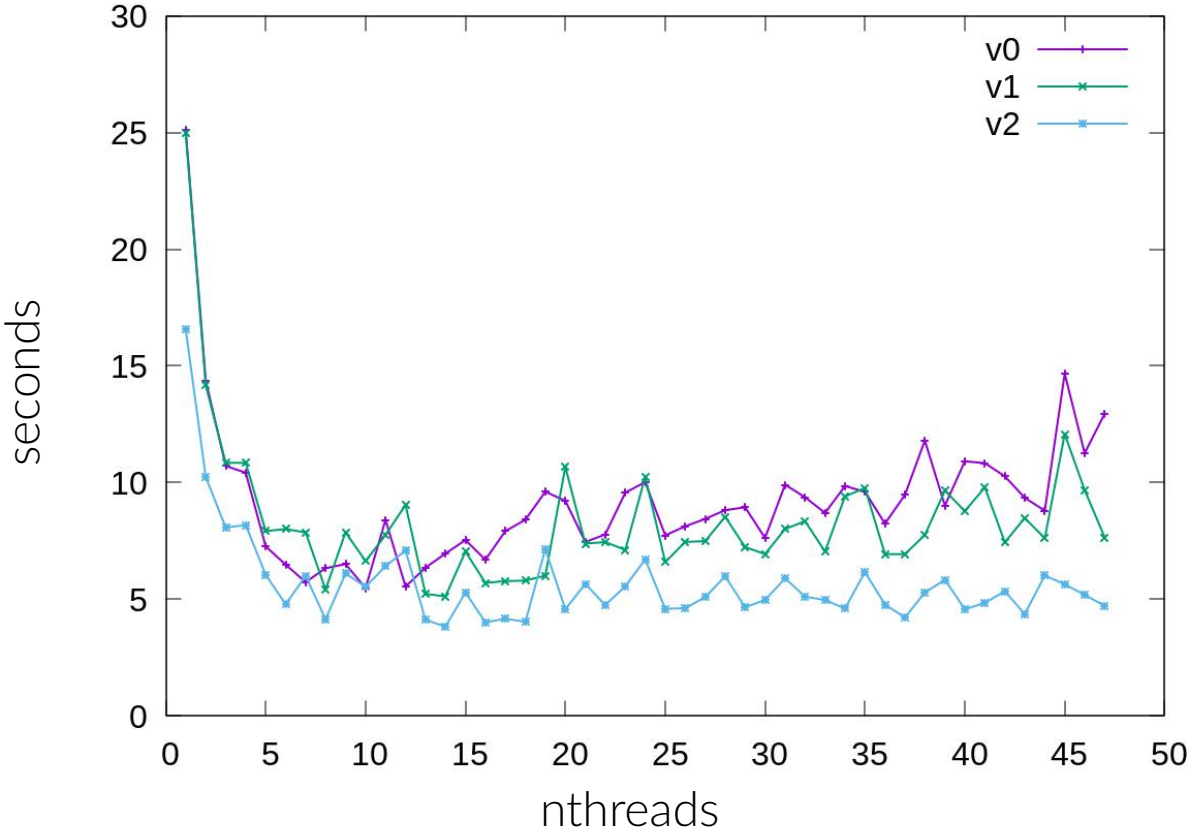
refining the task creation

Now, let's discuss the differences among 3 different implementations

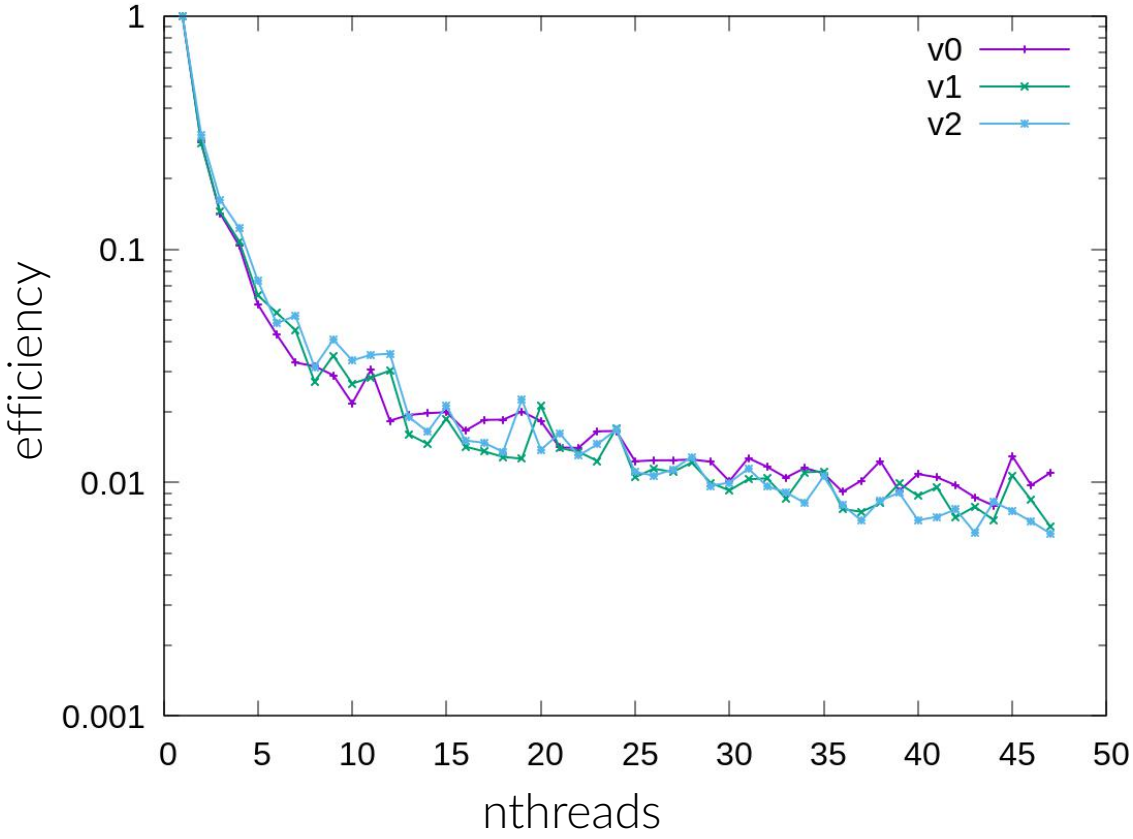
	v0	v1	v2
tasking	-	Just added the untied clause	final and mergeable clauses added
sorting	-	Added the sorting networks for few elements	Added the insertion sort for few elements

quicksort.v[0-2].c

Scaling



Efficiency


v0
v1
v2
efficiency
nthreads
1
0.1
0.01
0.001
0
5
10
15
20
25
30
35
40
45
50

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

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(…);
}
```

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 );
```

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 );
```

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 );
```

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);
}
```

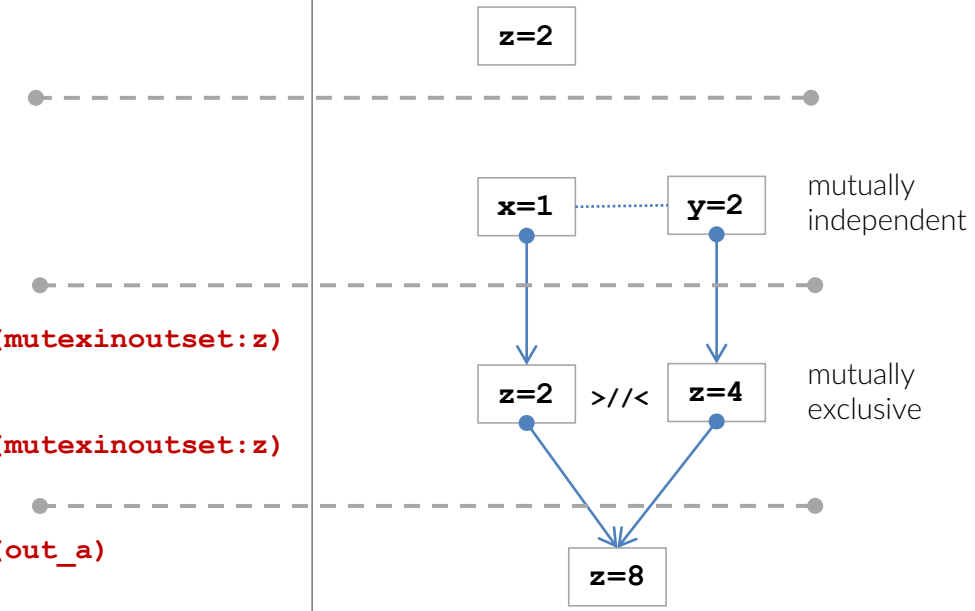
No dependence: output is variable, the printing tasks are independent off each other

```
#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

```
… 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

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(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
```

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 ?

examples taken from openmp.org

Controlling task creation and execution

*Task creation in **loops***

***Reduction** operations with tasks*

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 );
  }
```

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`

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

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

Clauses are very similar to both the
`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.

```
#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

```
#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

```
#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

Creation

- task region
  - **`if`** and **`final`** clauses
- undeferred (←failed **`if`**)
- included (←failed **`final`**)
- tied / untied
- **`taskgroup`**
- **`taskloop`** (SIMD)

Execution

- deferred at some point in the future
- scheduling points
  - immediately after the generation
  - after the task region
  - at a barrier (either implicit or explicit)
  - in a **`taskyield`** region
  - at the end of **`taskgroup`**

the **`taskyield`** is the only explicit one

Tasks

Data Environment

Scheduling

- priority
- dependencies

Synchronization

- implciti/explicit barrier
- **`taskwait`**
- **`taskgroup`**

that's all, have fun