OpenMP
TASKS - examples

SCIENTIFIC & DATA-INTENSIVE COMPUTING

Luca Tornatore - I.N.A.F.

Advanced HPC 2024-2025 @ Università di Trieste

Outline

The following slides detail some of the examples discussed in the lectures and introduced in the main pdf `tasks.pdf`
The list of the examples discussed here is as follows:

	The problem	What we use	Relevant files
●	variable workload ; how to manage an unpredictable workload on data chunks. Reduction among tasks. Creation of tasks in a for loop assigning chunks of iterations, “by hands” and by omp constructs	basic task creation syntax, taskgroup	`03_*.c` `04_*.c` `05_*.c`
●	unpredictable workload examples • how to traverse a sorted linked-list • how to build a balanced binary tree and traverse it • how to solve a random DAG with unpredictable dependencies among the nodes	basic task creation syntax, taskwait	`linked_list.traverse.c` `linked_list.generate_nodes.c` `dag.c` `AVLtree.c`
●	how to build a heap with a linked-list solving the insertion problem	locks	`linked_list.c`
●	parallelizing the quick-sort and the merge-sort	if and final clauses	`quicksort.c` `mergesort.c`

Simple task management with irregular workload; reduction among tasks

| Unpredictable workload

We’ll explore variations on how to manage a variable workload.

1. A code that allows you to compare the run time of dealing with random work associated with the entries of an array using either a for loop or the tasks
2. A code that “receives” chunks of data with ranbdom workloads and generate tasks to process them
3. …tbc

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.

example: variable workload

```c
#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

OpenMP

| 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

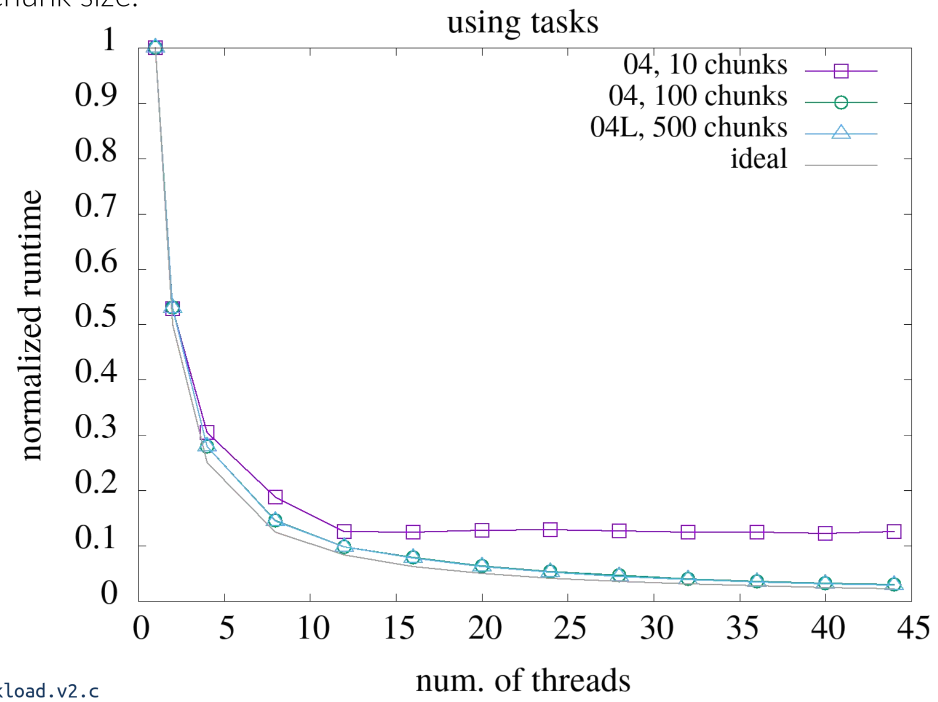
The case we adopted is "perfectly suited" for a `for dynamic` only if all your data are already in place, i.e. you do have an array to cycle over.
Quite the opposite, if your data are "arriving" the task solution is a very elegant and efficient one, while a `for` loop would be impossible.

OpenMP

| 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

| Using 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

Using taskgroup

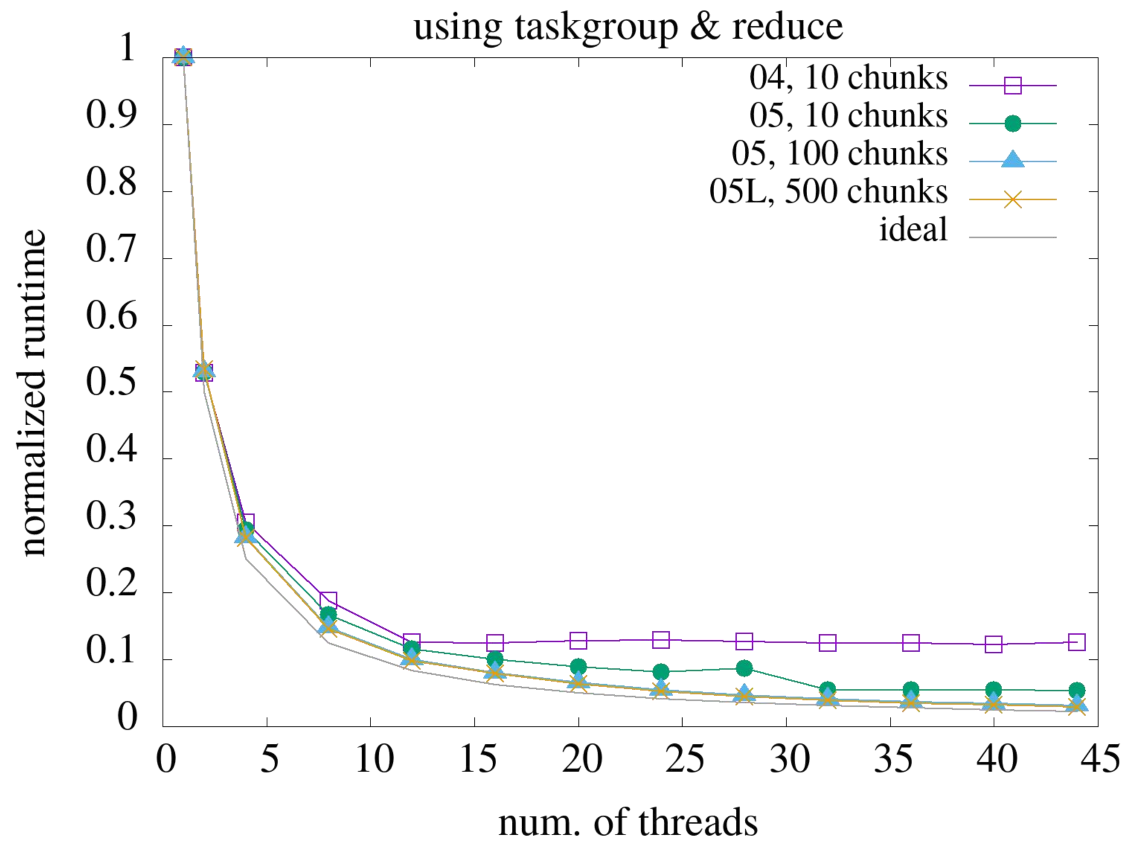
```c
#pragma omp parallel proc_bind(close)
  {

    #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

| Using taskgroup



04_tasks_reduction.c

05_taskgroup_reduction.c

Unpredictable workload

linked-list and balanced tree traverse

| Unpredictable workload

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

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

Traversing a linked-list

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

A classical example: traversing a linked list

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

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

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

Traversing a linked-list

`linked_list.generate_nodes.c`

Helper code: generates a random set of values to be used as stream of values to build a linked list with `linked_list.traversal.c`.
The N values may between 0 and a given max value, or around a give average value with a given dispersion.

`linked_list.traverse.c`

Actual code of interest.
You can choose to process the linked list (input values from a file, generated by `generate_nodes_for_linked_list.c`) by:
- generating an array of active list's nodes and
  - processing with a omp for static
  - processing with a omp for dynamic
- generating a task per active node
- generating a task per bunch of active nodes

The chunk size for the for loop, or of the bunch of active nodes for the tasks, is a commad-line parameter

OpenMP

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 );
   }
}
```

If the nodes to be processed are many, more than a gigazillion say, the overhead of tasks management may be critical.
I would be great if at some point the task creation could be frozen until some tasks are drained from the task pool..

```
#pragma parallel region
{
  #pragma omp single
  #pragma omp task untied
    walk_list_and_create_tasks(..),
}

void walk_list_and_create_tasks ( ..) {
  while( !end_of_list(node) ) {
    if( node_is_to_be_processesed(node) )
      #pragma omp task
       process_node ( node );
         node = next_node( node );  }
}
```

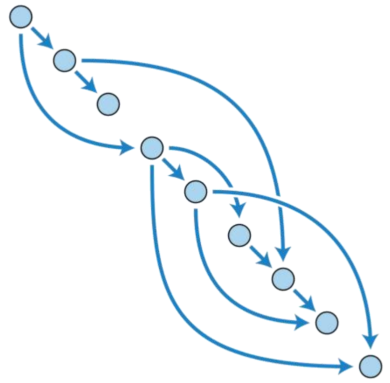
Now the task creation is itself a task that can be suspended.
Being untied, it can be resumed by any thread.

linked_list.traverse.b.c

OpenMP

| Solving a graph

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



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

▷ `dag.c`

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

| Solving a graph

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

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

| 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`..

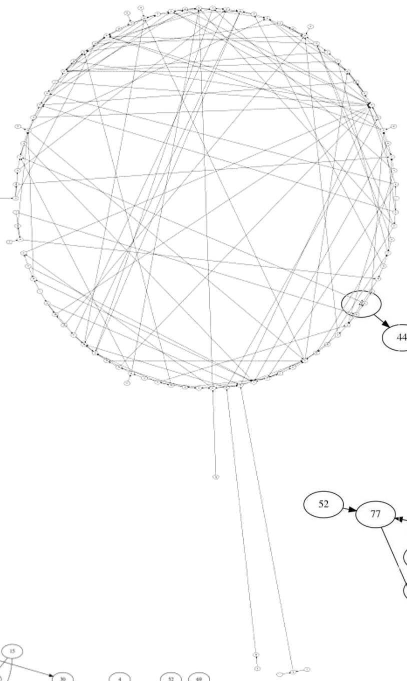
Building the DAG

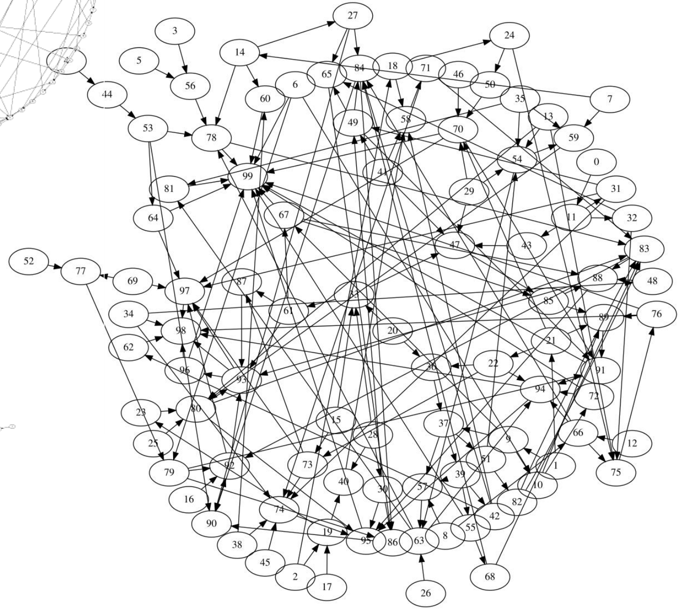
The routine that generates the dag is named `generate_dag()` and it is pretty simple. The free parameters are: the total number of nodes, the number of root nodes, the minimum and maximum number of children per node, and the baseline workload per each node.

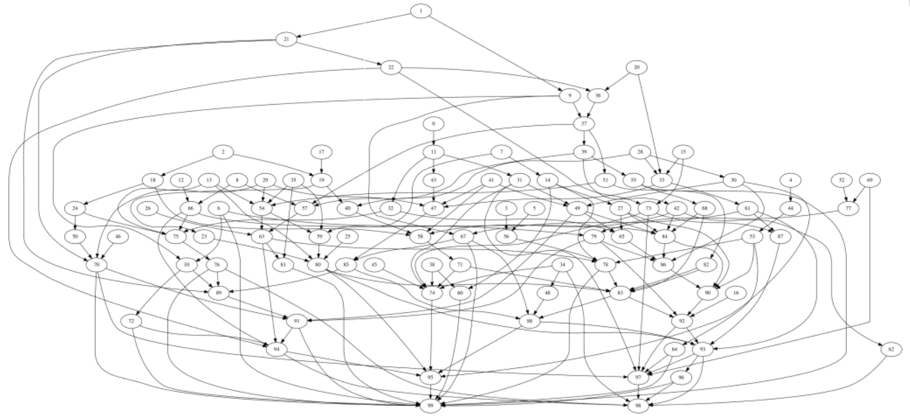
Here you find 3 different representations of the same small dag, having just 100 nodes (you can generate your own using the aforementioned routine).

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.

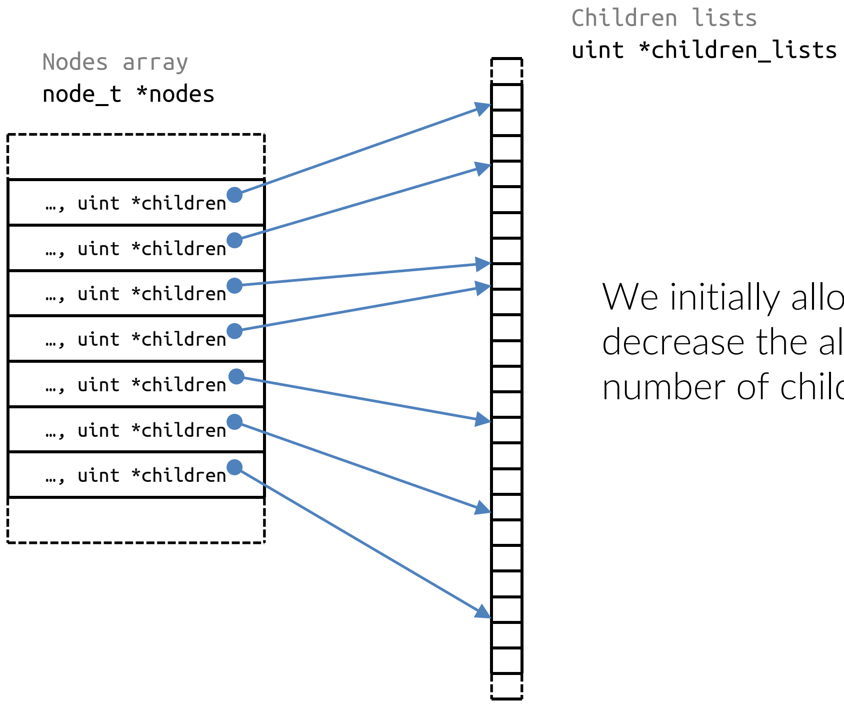
1) The basic point is how to avoid loops inside the directed graph. A simple way to achieve the goal is to enforce that each node of the graph has children that only live "forward" to it.
If we store the *N* nodes in an array, we can implement that by committing each node *i* to have children with an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

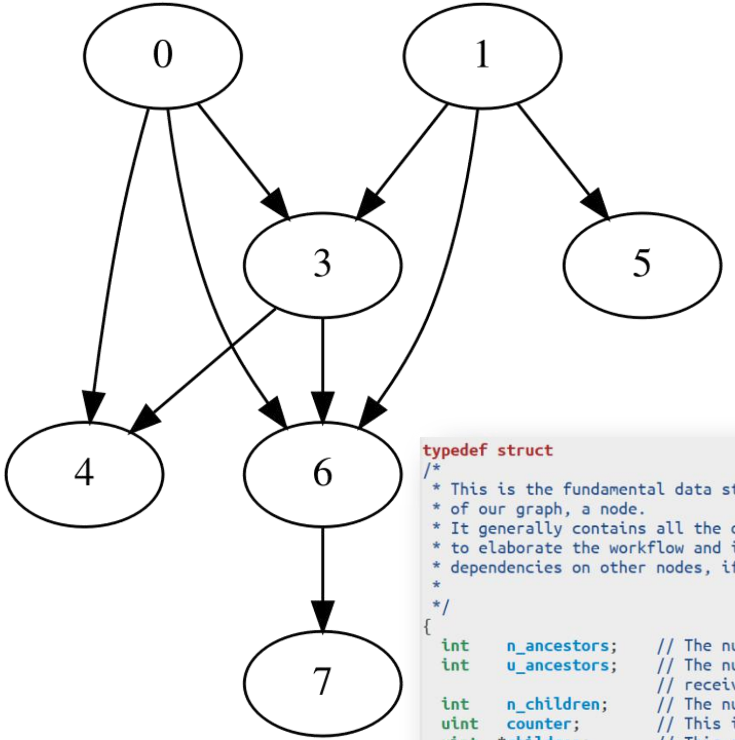
Building the DAG

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.

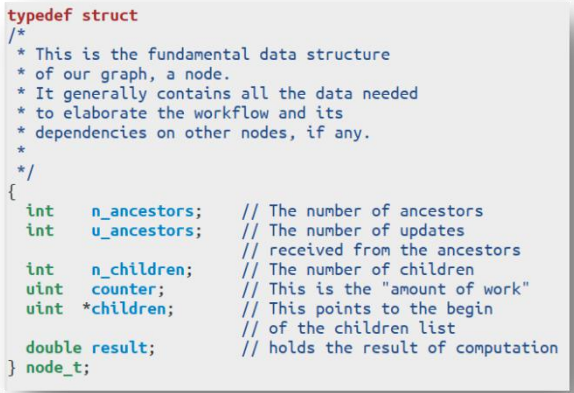
Building the DAG



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



```
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;
```

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.

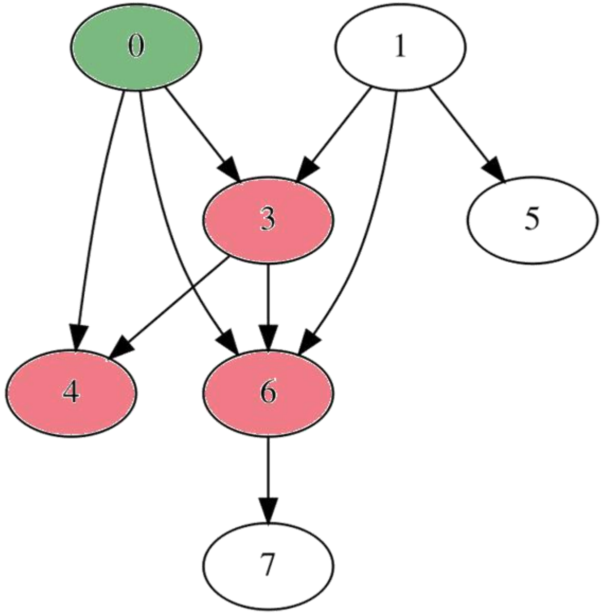
Notice that it is totally impossible to forecast what will be the execution pattern before the nodes are created.

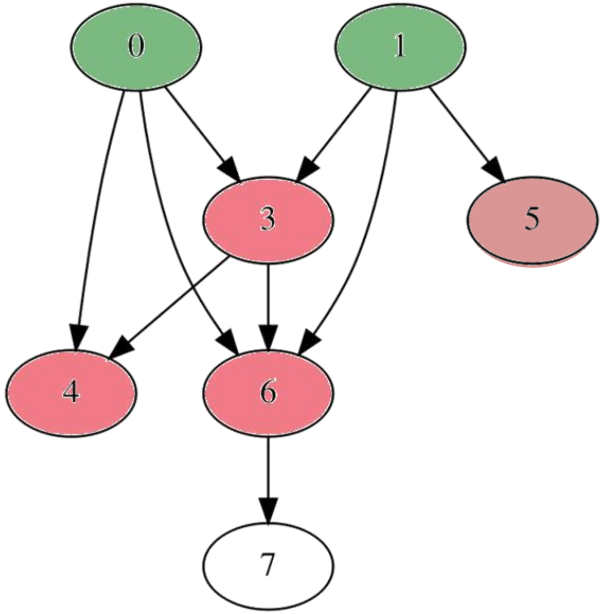
The general strategy

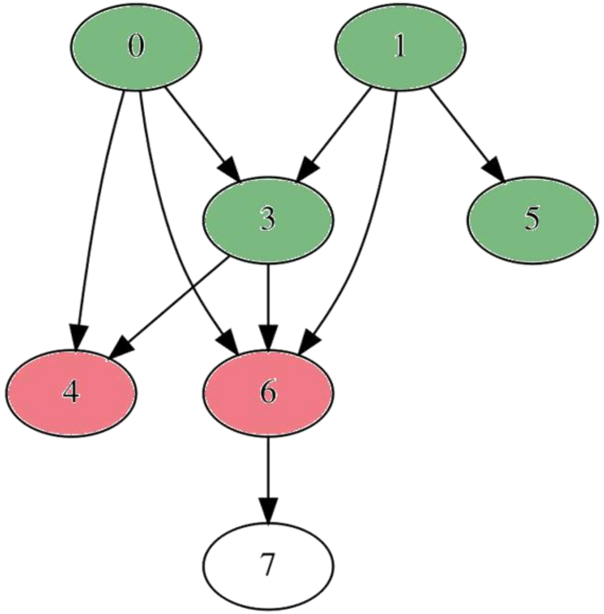
The root nodes are initialized, let's say we start from 0.
It updates its own descendants, that are then partially updated.

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

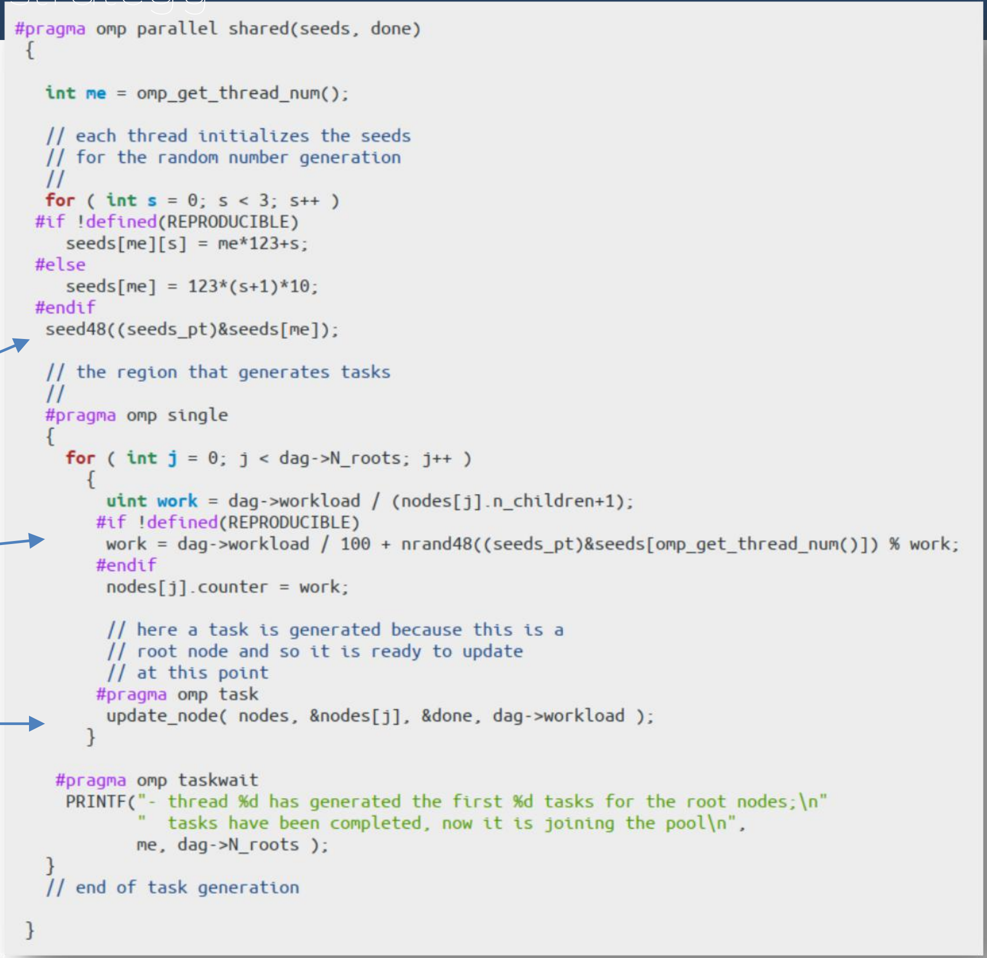
That triggers all the fully updated descendants to contribute to their children, and so on







The general strategy



```
#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

 }
```

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.

Generating the tasks

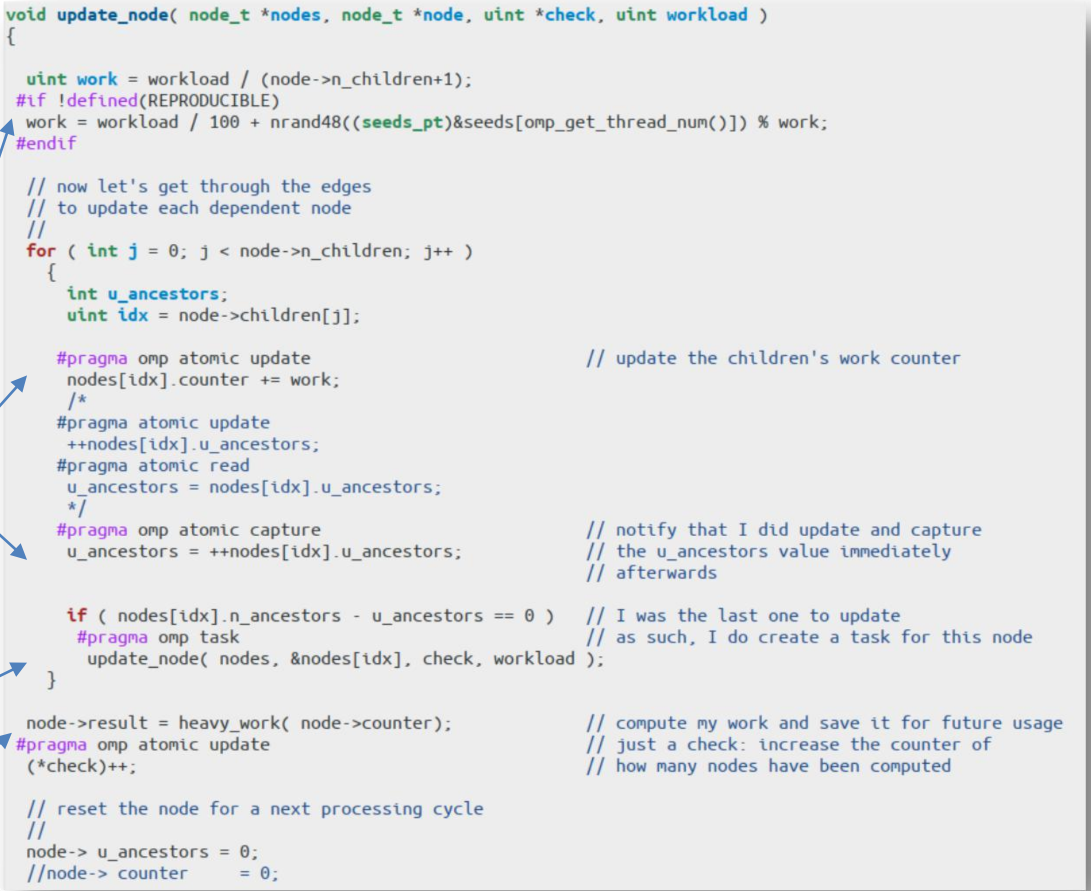
Inside **update_node()**, each task

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.



```c
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

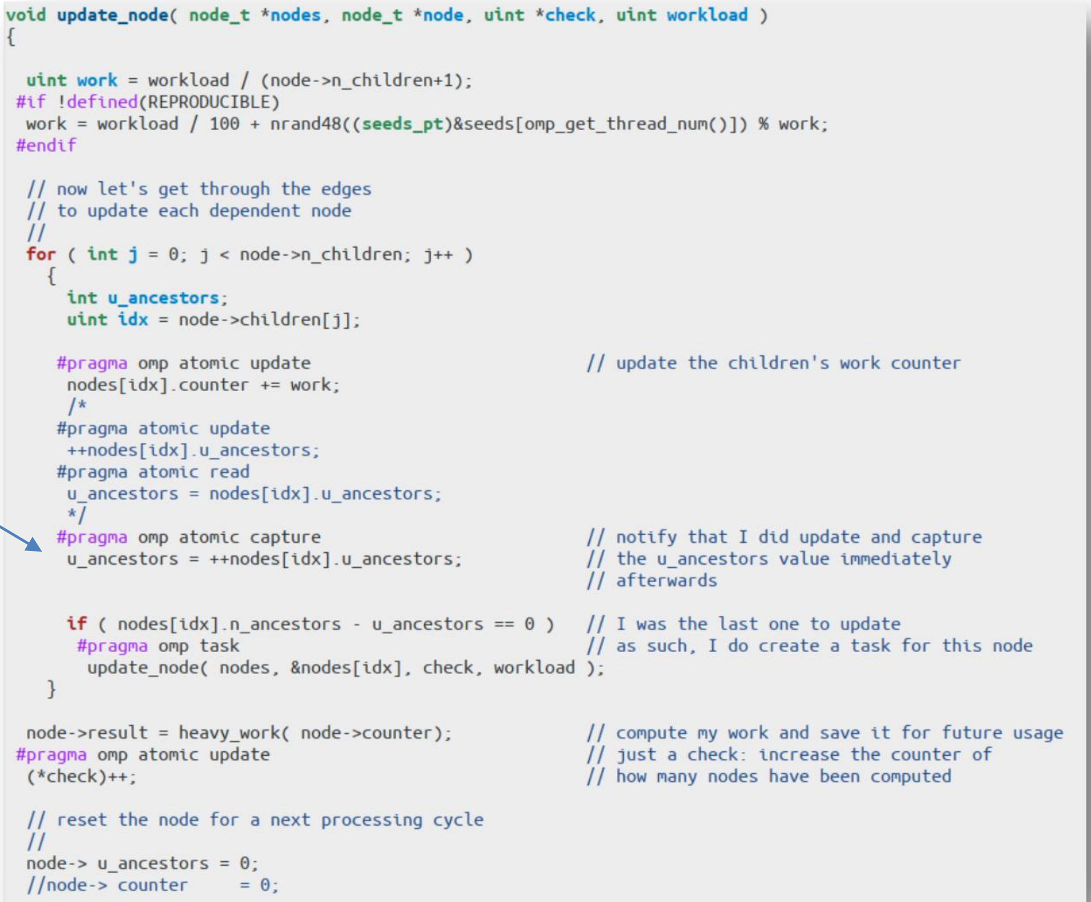
The usage of this `atomic capture` is an important detail to discuss.
Let's understand it more deeply.

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;
```

Generating the

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

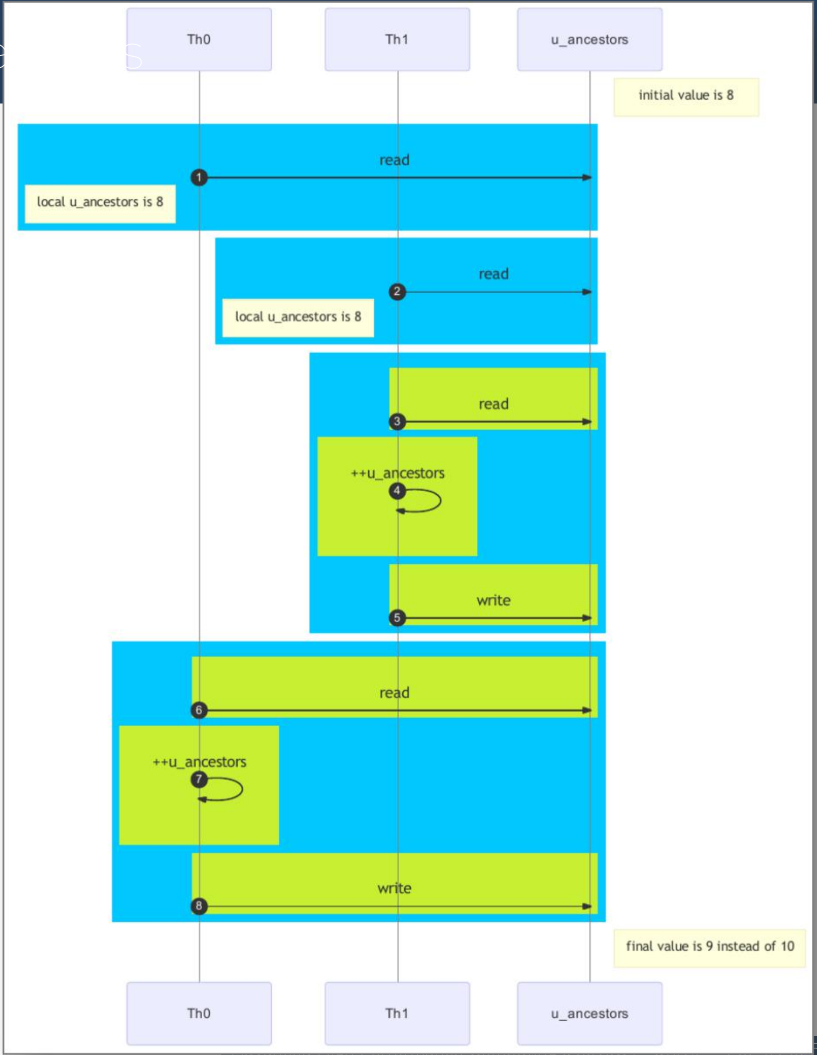
Give that the operation `++u_ancestors` requires 3 steps, namely

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;
```

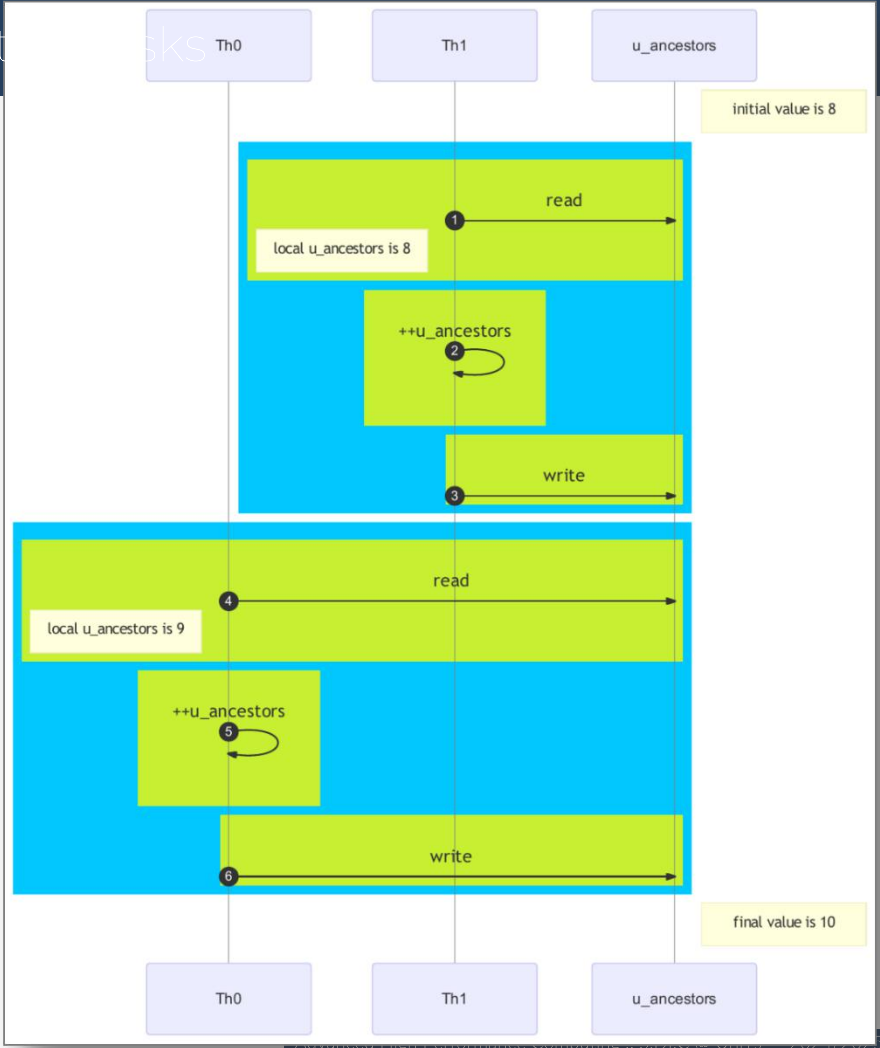
That could easily result the sequence presented here on the right (blue regions represent "exclusive accesses" – i.e. `omp atomic` – to `u_ancestors`).
Both thread 0 (Th0) and thread 1 (Th1) are convinced to be the 9th and none of the two realizes to be the 10th.
Then, the corresponding task for the 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: a note

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

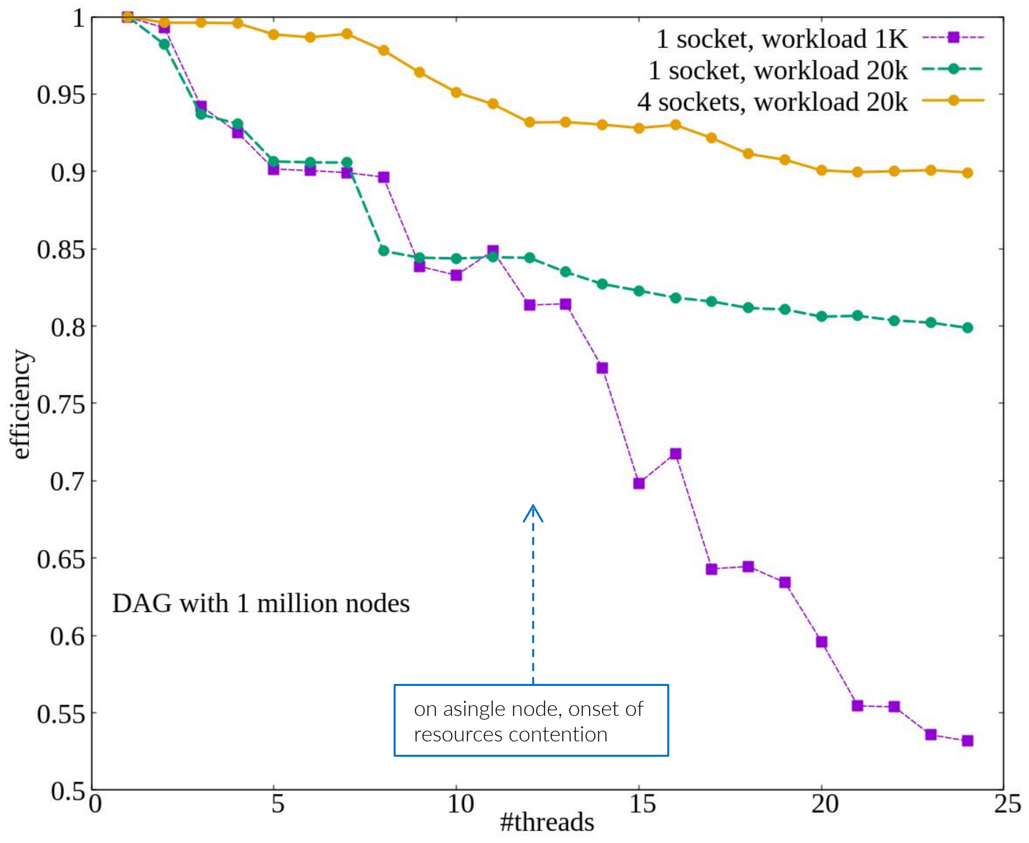
The task creation is *not* a recursive call to `update_node()`.
In fact, recursion happens when the call (i.e. a code jump and the relative creation of the stack) happens in that very moment, and the stack of the called function lives right under the stack of the caller function.

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` **>** »

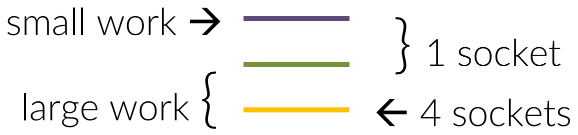
where the embedded value of `idx` is the value at the moment of the task creation (and the same holds of course for all the other variables, which however in this example do not change).

Generating the tasks : a note



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" ~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
```

Synchronization and Dependence

Tasks Synchronization

Building a heap via linked-list using locks

Example: Building a heap

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;
}
```

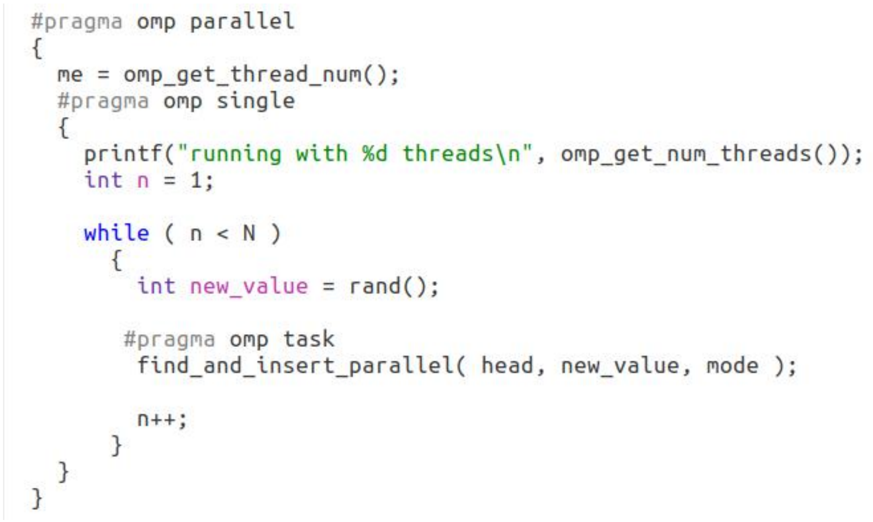
linked_list.c

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.



```c
#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

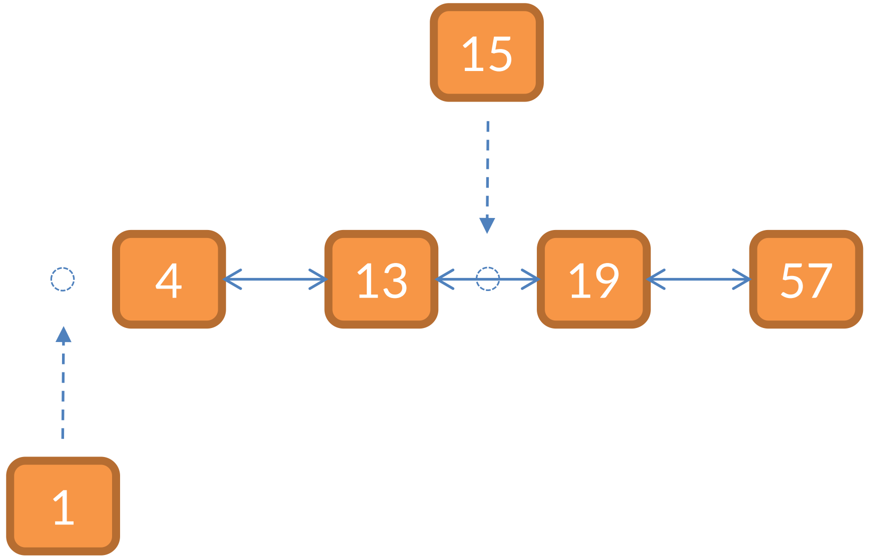
The very first step for a new ordered insertion will be to find the Left and Right nodes that are the largest smaller and the smallest largest than the new value to be inserted.
For the sake of simplicity our data we'll be integers.

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

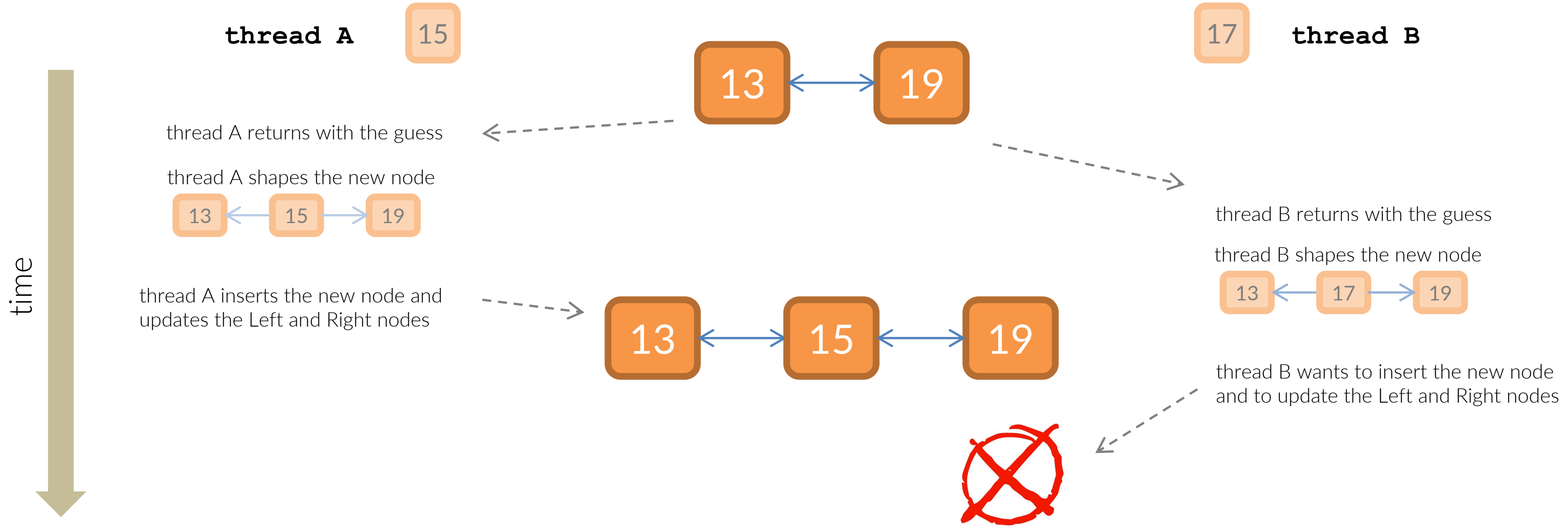
```
int find ( llnode_t *head, int value,
           llnode_t **prev, llnode_t **next );
```

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 it to be the head of the list)



Building a heap with a double-linked 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.



thread A
15
17
thread B
13
19
time
thread A returns with the guess
thread A shapes the new node
13
15
19
thread B returns with the guess
thread B shapes the new node
13
17
19
thread A inserts the new node and updates the Left and Right nodes
13
15
19
thread B wants to insert the new node and to update the Left and Right nodes

OpenMP

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 contiguous 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;
      }
  }
```

OpenMP

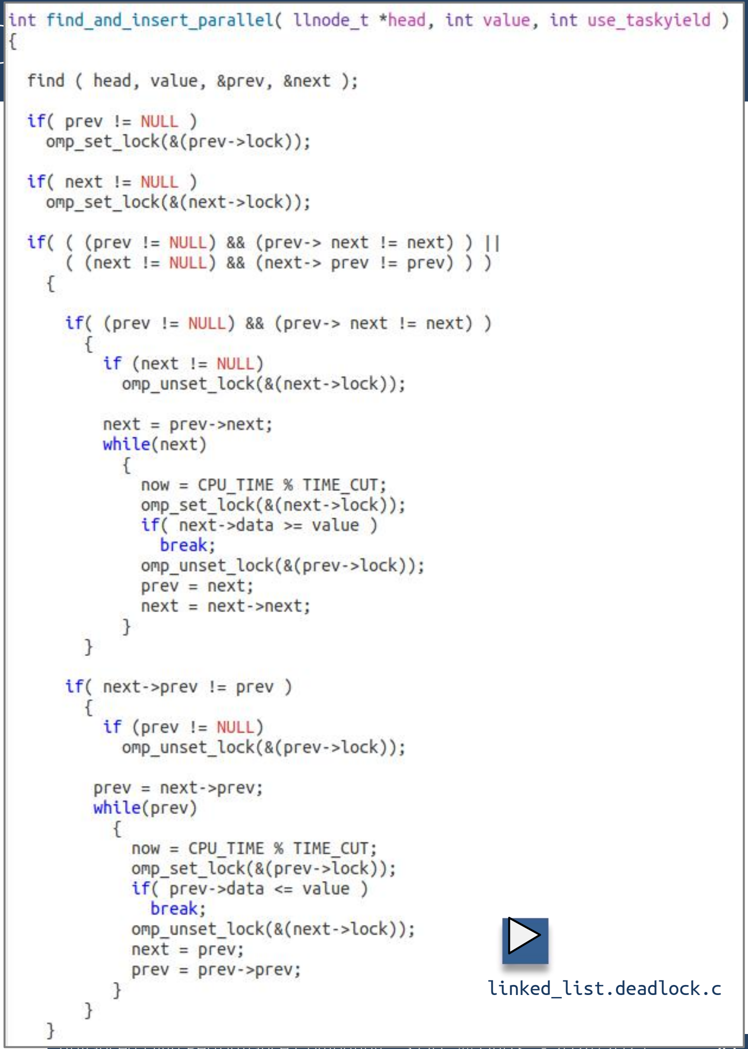
Building a heap with a c

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 ?



```c
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

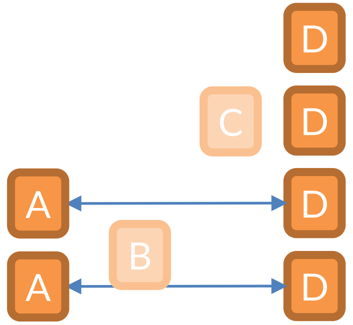
thread z finds guess for B
LEFT = A, RIGHT = D

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

OpenMP `tasks`

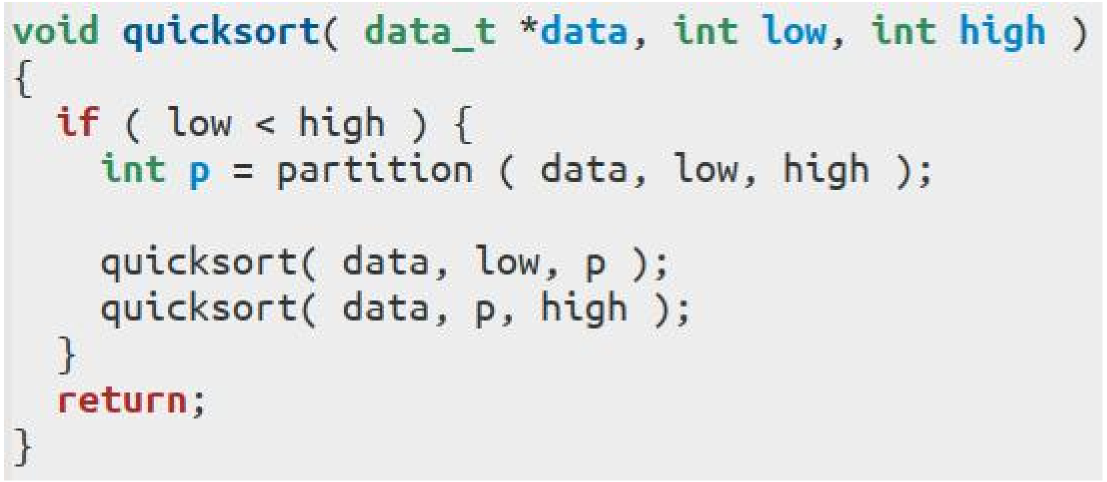
Controlling the task creation
Clauses

Example: Building a heap

Parallelizing the Quicksort

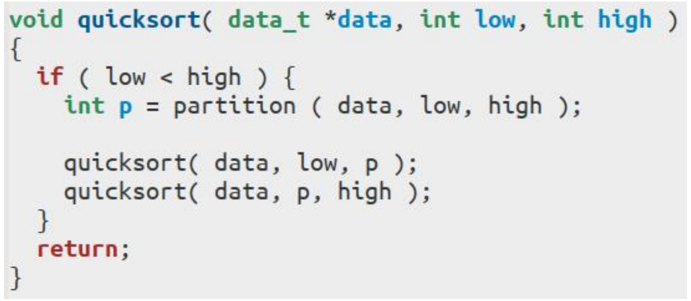
Task creation example: QuickSort

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;
}
```

Task creation example: QuickSort



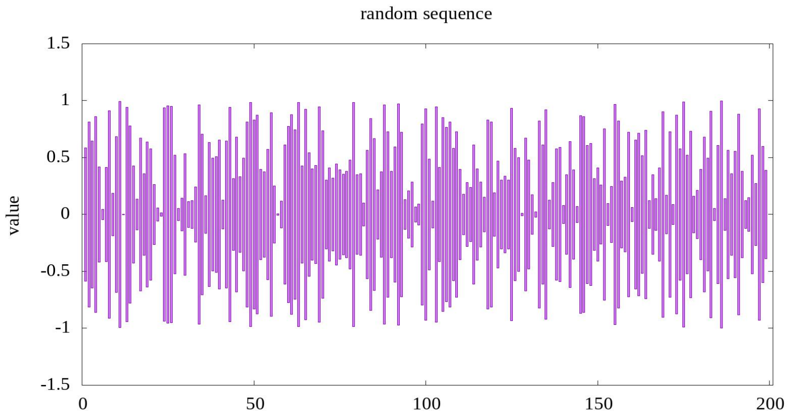
```
void quicksort( data_t *data, int low, int high )
{
  if ( low < high ) {
    int p = partition ( data, low, high );

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

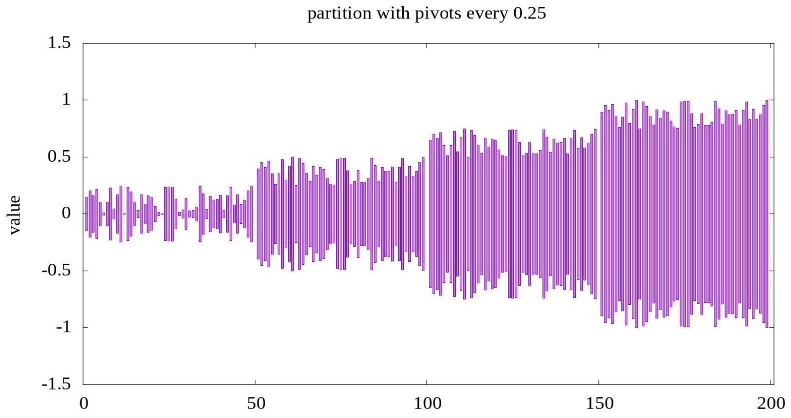
The `partition` function divides the array data in (hopefully) 2 sections.
It individuates the (hopefully) median element `p`, and move all the entries `a[i]<p` in the left part and all the entries `a[i]≤p` in the right part.

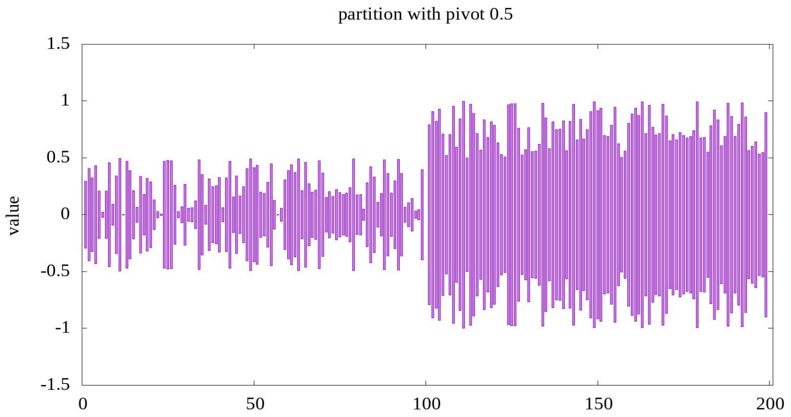
There are *lots* of subtleties to consider and tricks to implement in order to make this algorithm as efficient as possible, but the big picture is the one we have just seen.

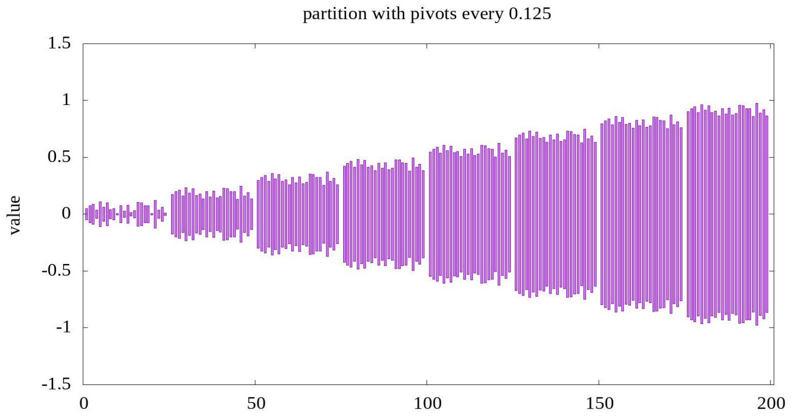
It performs as *NlogN* in the average case, and as N^2 in the worst case (can you figure out which is the worst case?)

Task creation example: QuickSort









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

QSort example

Task creation example: QuickSort

```c
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

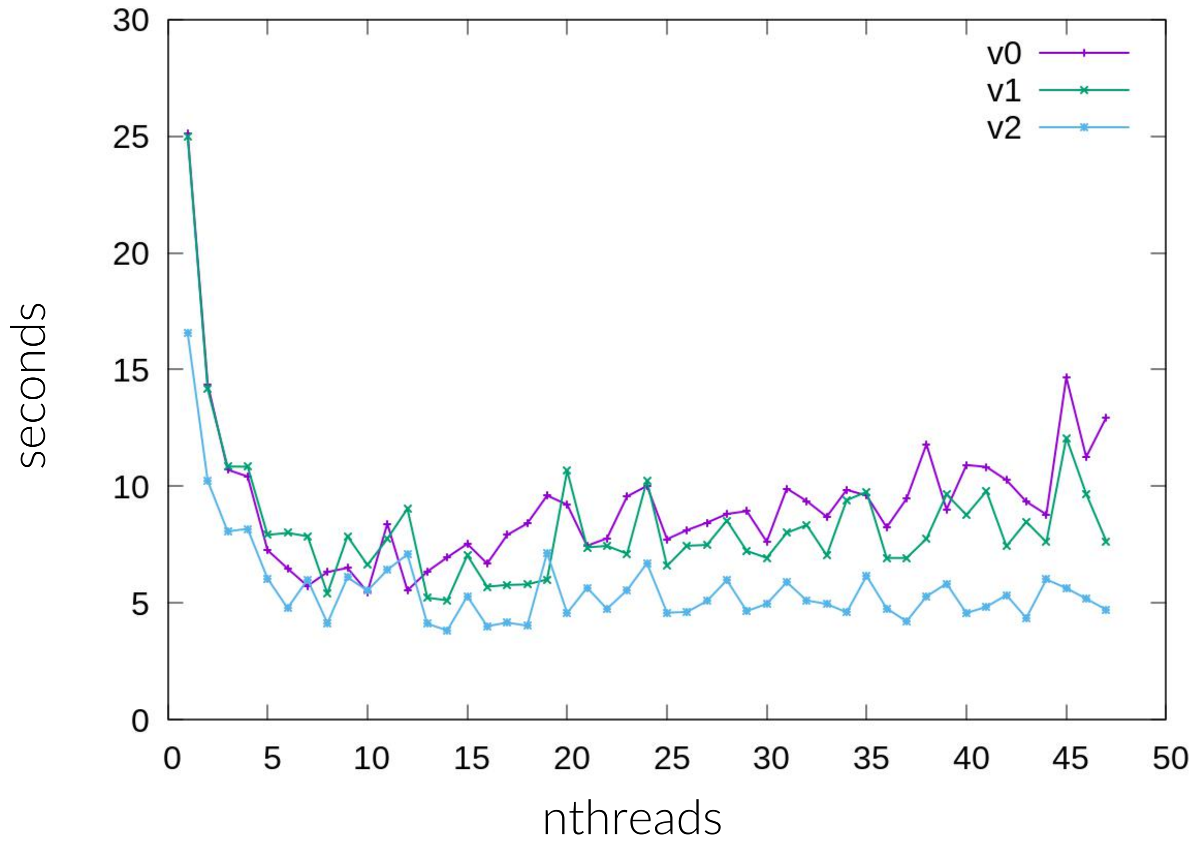
Task creation example: QuickSort

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

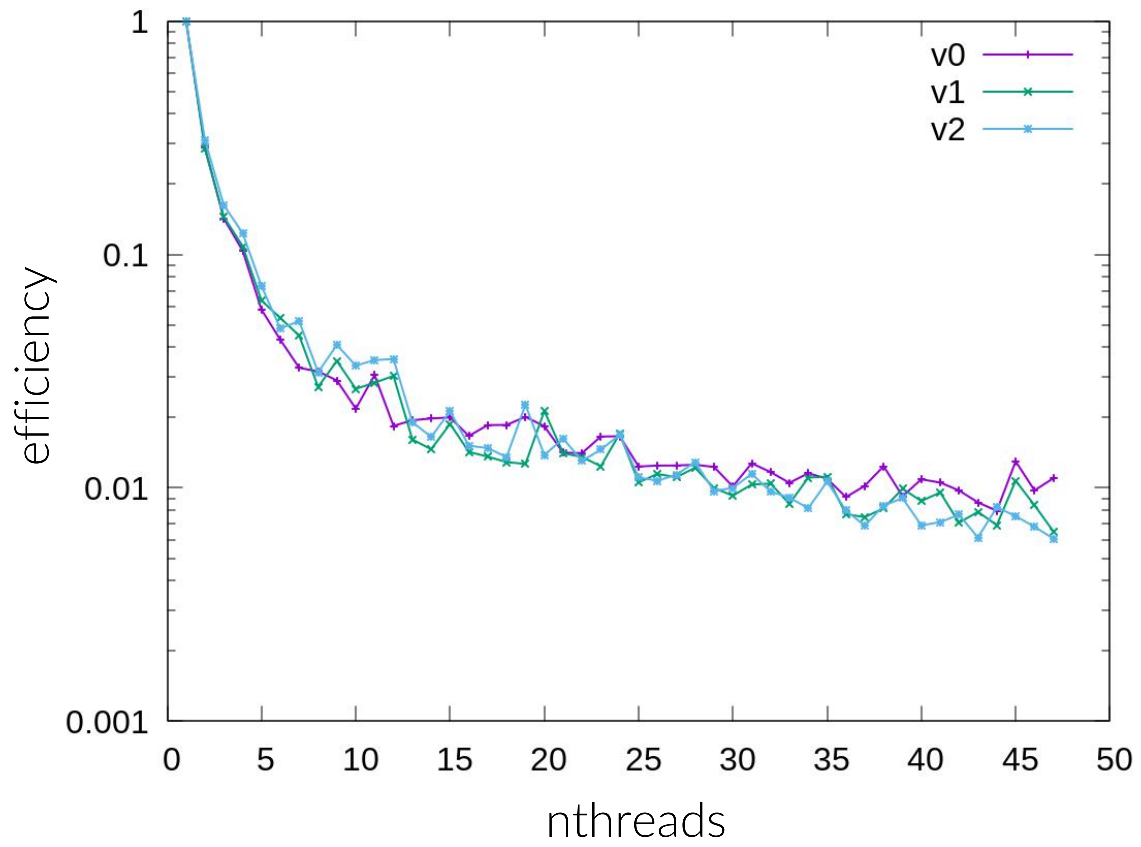
	v0	v1	v2
tasking	-	Just added the untied clause	final and mergeable clauses added
sorting	-	Added the sorting networks for few elements	Added the insertion sort for few elements

`quicksort.v[0-2].c`

Task creation example: QuickSort



Task creation example: QuickSort



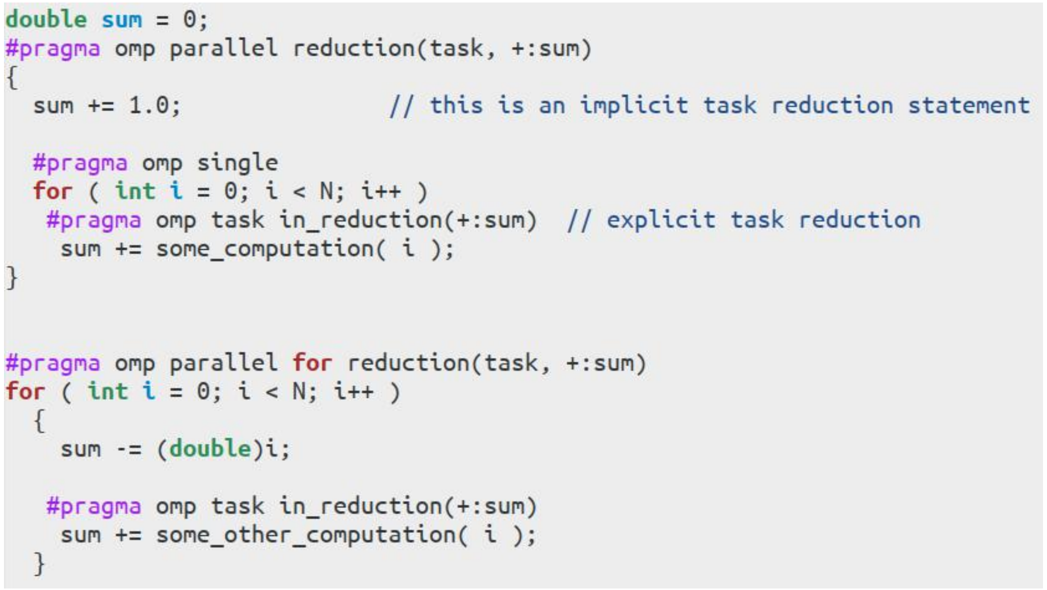
Controlling task creation and execution

Task creation in ***loops***

Reduction *operations with tasks*

| OpenMP task reduction

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



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

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


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

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

| OpenMP `taskloop`

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

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

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

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

| OpenMP **taskloop**

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

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

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

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

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

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

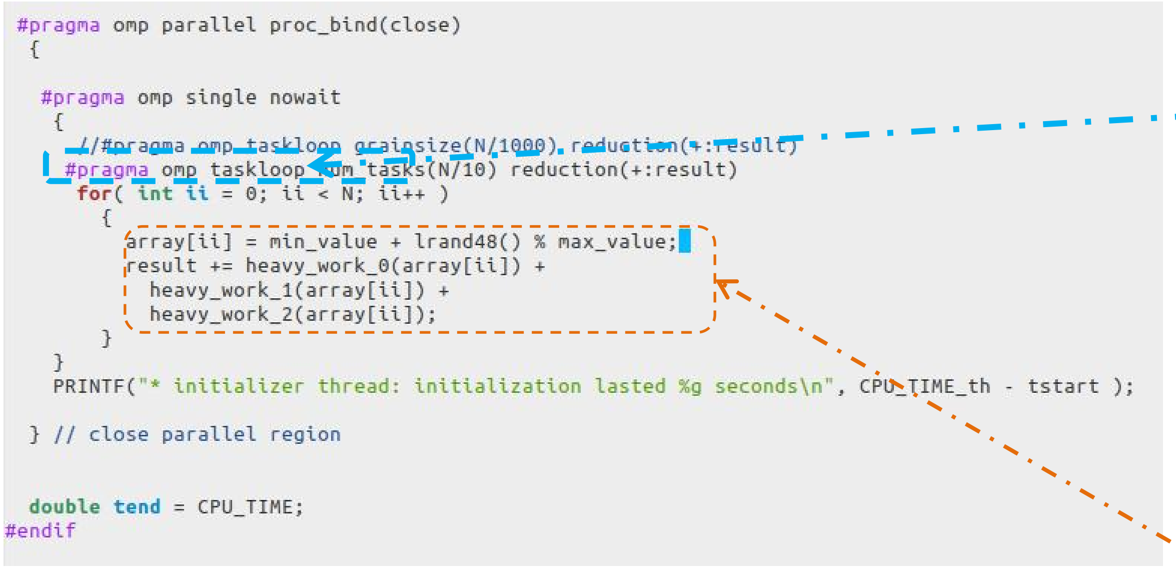
`nogroup`
The tasking construct is not embedded in an otherwise implied `taskgroup` construct.

| OpenMP taskloop

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

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

 } // close parallel region


  double tend = CPU_TIME;
#endif
```

day26/example_tasks/
07_taskloop.c

| OpenMP taskloop



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

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

 } // close parallel region


  double tend = CPU_TIME;
#endif
```

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

Tasks are created for each iteration

day26/example_tasks/
07_taskloop.c

| OpenMP taskloop

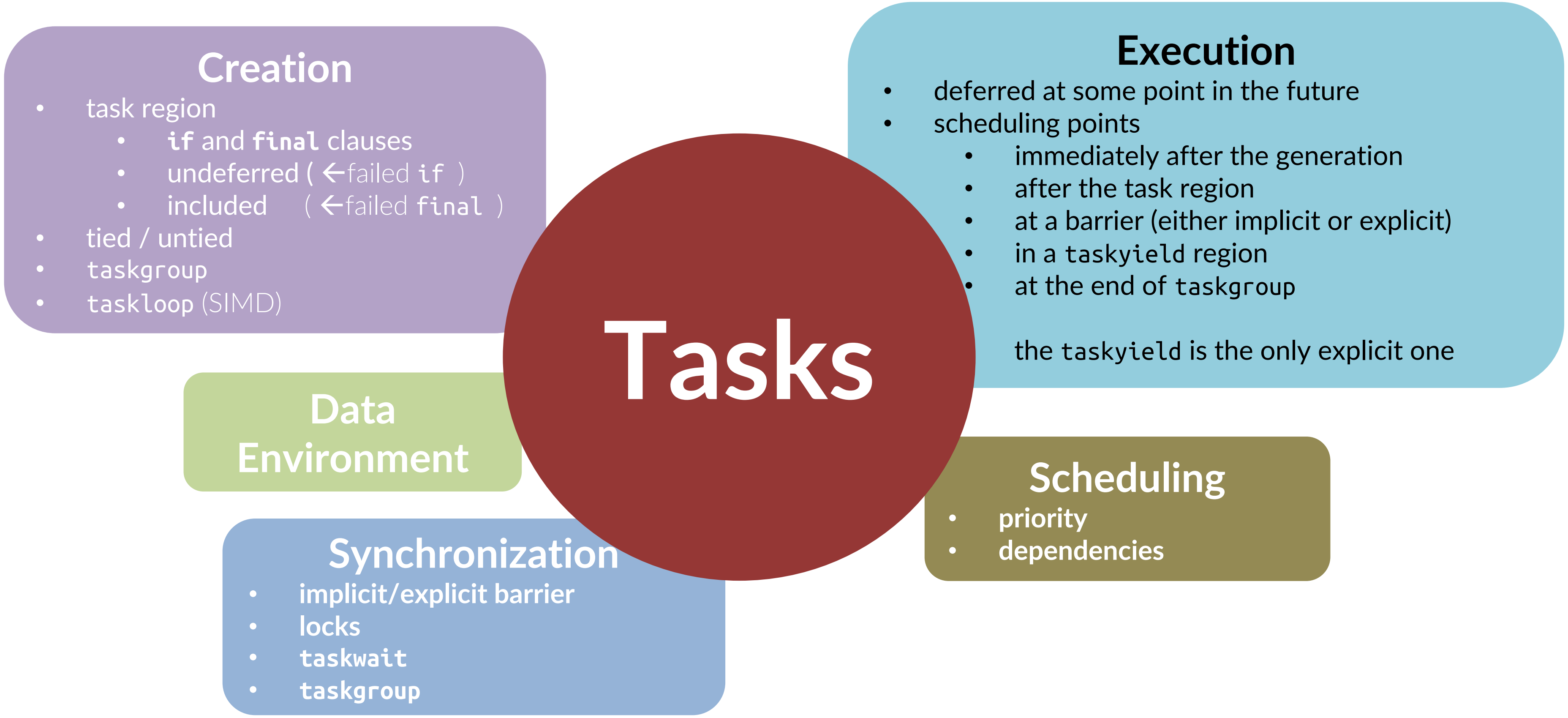
```
#pragma omp parallel proc_bind(close)
 {

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

 } // close parallel region


  double tend = CPU_TIME;
#endif
```

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

taskloop.c

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

- implicit/explicit barrier
- locks
- `taskwait`
- `taskgroup`

OpenMP

that's all folks, have fun