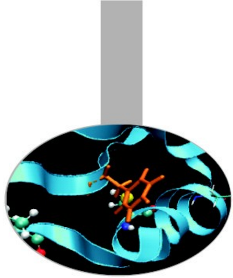
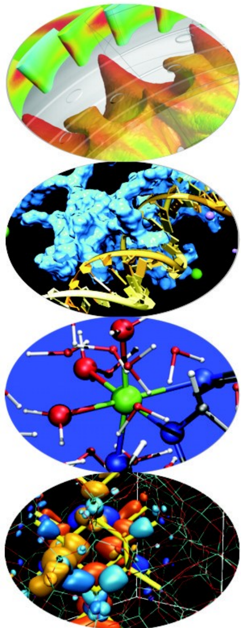
SCAI

SuperComputing Applications and Innovation



OpenMP

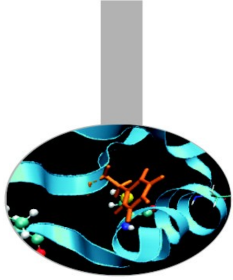
Tasking in OpenMP 4



Mirko Cestari - m.cestari@cineca.it
Marco Rorro - m.rorro@cineca.it

CINECA

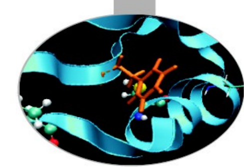
Outline

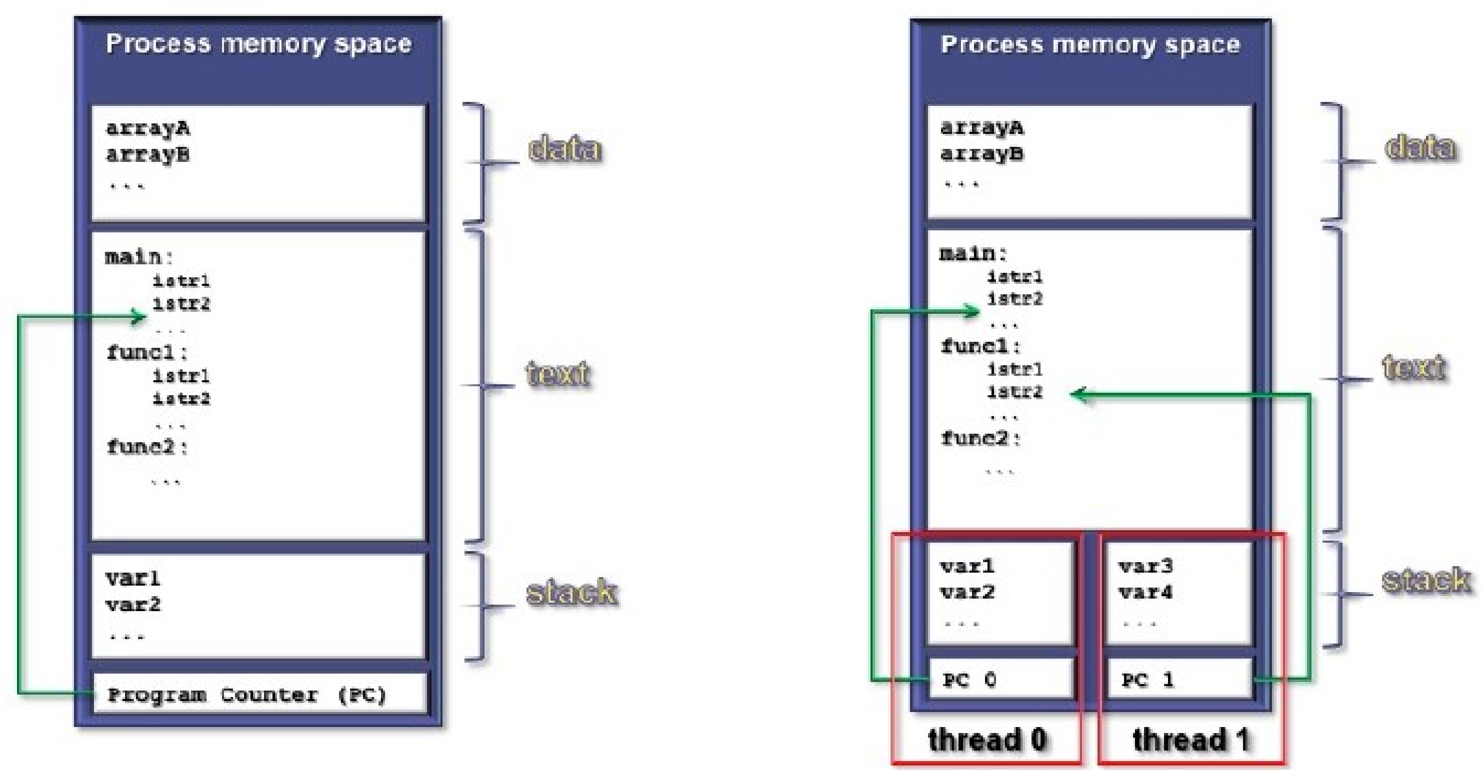


- **Introduction to OpenMP**
- General characteristics of Taks
- Some examples
- Live Demo

SuperComputing Applications and Innovation

Multi-threaded process



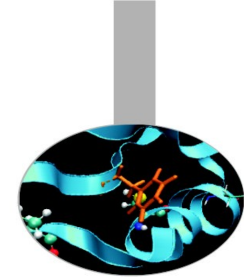


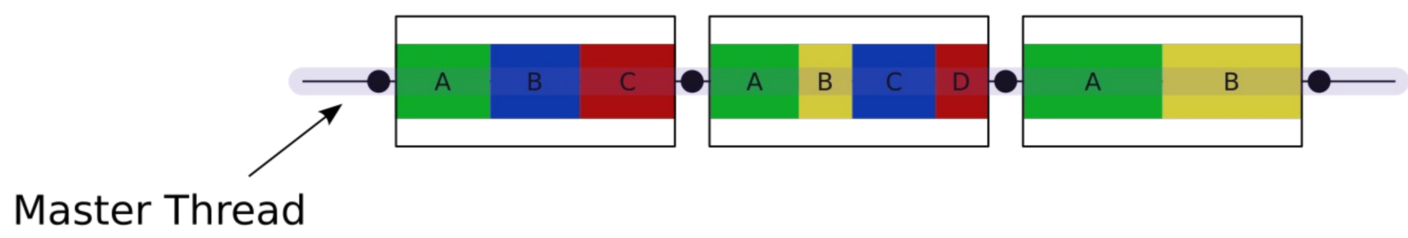
- Each thread may be regarded as a concurrent execution flow

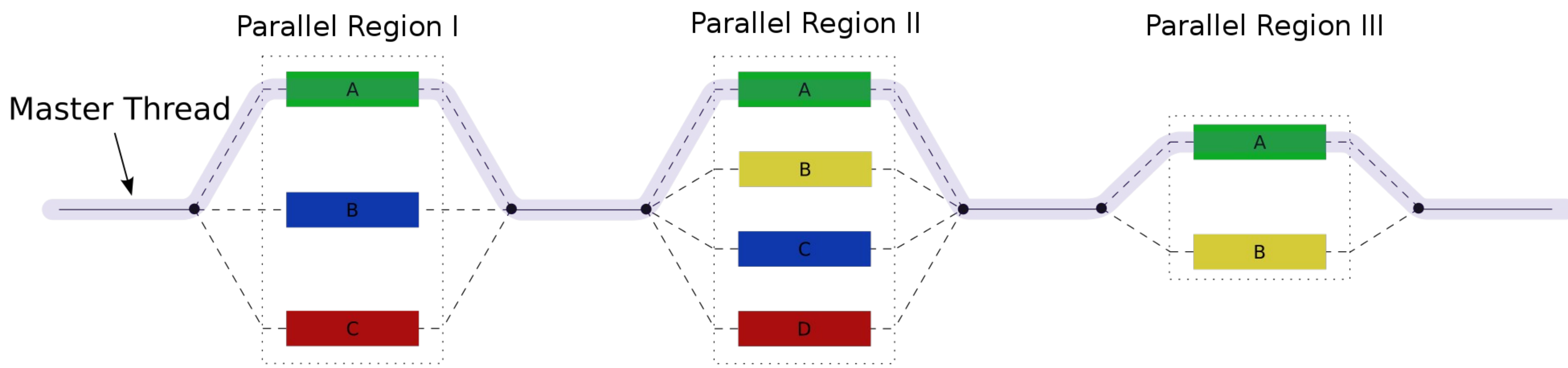
OpenMP

CINECA

Fork-Join parallel execution

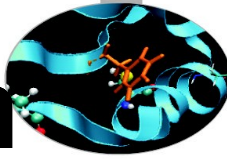






SCAI SuperComputing Applications and Innovation

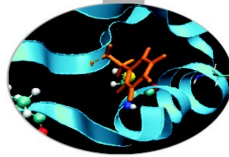
Structure of an OpenMP program



- **Execution model**
  - fork-join parallel execution
  - the program starts with an initial thread
  - when a parallel construct is encountered a team is created
  - parallel regions may be nested arbitrarily
  - worksharing constructs permit to divide work among threads
- **Shared-memory model**
  - all threads have access to the memory
  - each thread is allowed to have a temporary view of the memory
  - each thread has access to a thread-private memory
  - two kinds of data-sharing attributes: private and shared
  - data-races trigger undefined behavior
- **Programming model**
  - compiler directives + environment variables + run-time library

OpenMP CINECA

SCAI
SuperComputing Applications and Innovation

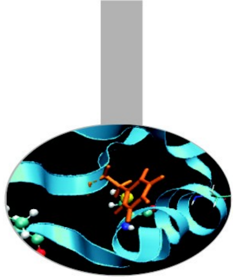


OpenMP core elements

OpenMP

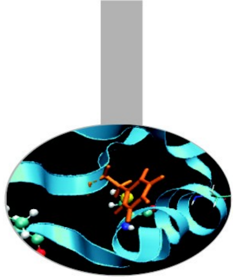
CINECA

OpenMP 2.5



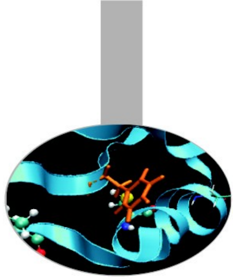
- 2 main worksharing contructs
  - **Loop construct**: the number of iterations is determined before entering the loop
  - Number of iterations cannot be changed
  - The **sections contruct**: sections are statically defined at compiled time

- Synchronization contructs affect the whole team of threads
  - Not just units of work



Tasks: motivations

- Modern applications are larger and more complex
- Irregular and dynamic structures are widely used
  - While loops
  - Recursive routines
- OpenMP 2.5 is not suitable to exploit this kind of concurrency

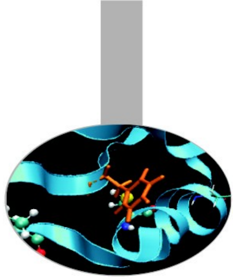


Pointer chasing using single

```
#pragma omp parallel private(p)
{
   p = head;
   while(p) {
      #pragma omp single nowait
         process(p);
      p=p->next;
   }
}
```

- Each thread performs the while loop (traverses the whole list)
- Each thread has to determine if another thread already executed the work on that element

CINECA SCAI
SuperComputing Applications and Innovation



Outline

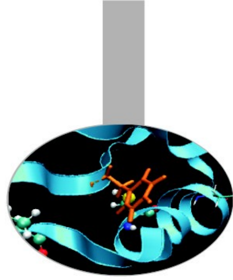
- Introduction to OpenMP
- **General characteristics of Taks**
- Some examples
- Live Demo

OpenMP

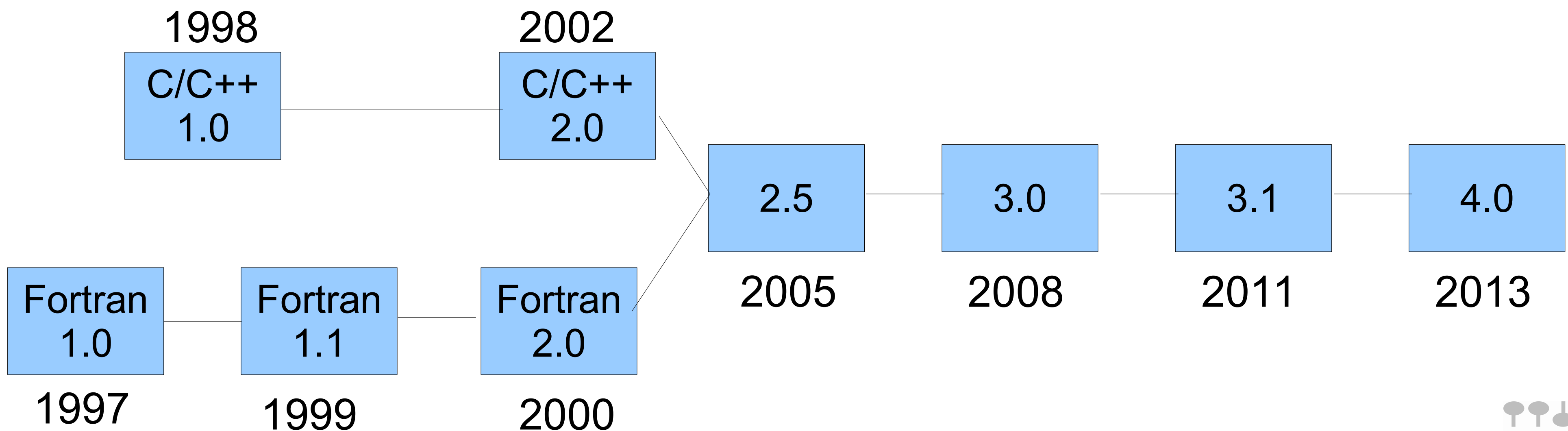
CINECA

SuperComputing Applications and Innovation

Tasks



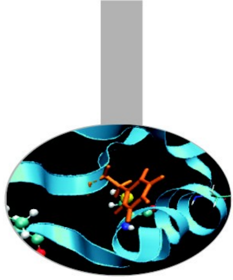
- **First Introduced in OpenMP 3.0**
  - has been the major addition from OpenMP 2.5
- **Refined in OpenMP 3.1 and OpenMP 4.0**



OpenMP

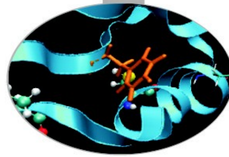
CINECA

Tasking



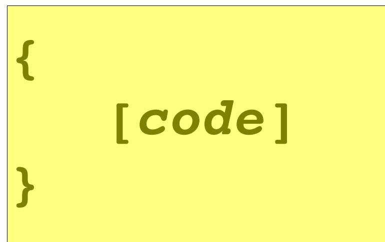
- From a thread-centric model to a task centric-model
- A model in which users identify independent unit of works and rely on the system to schedule these units
- Irregular parallelism: dynamically generated units of work that can be executed asynchronously

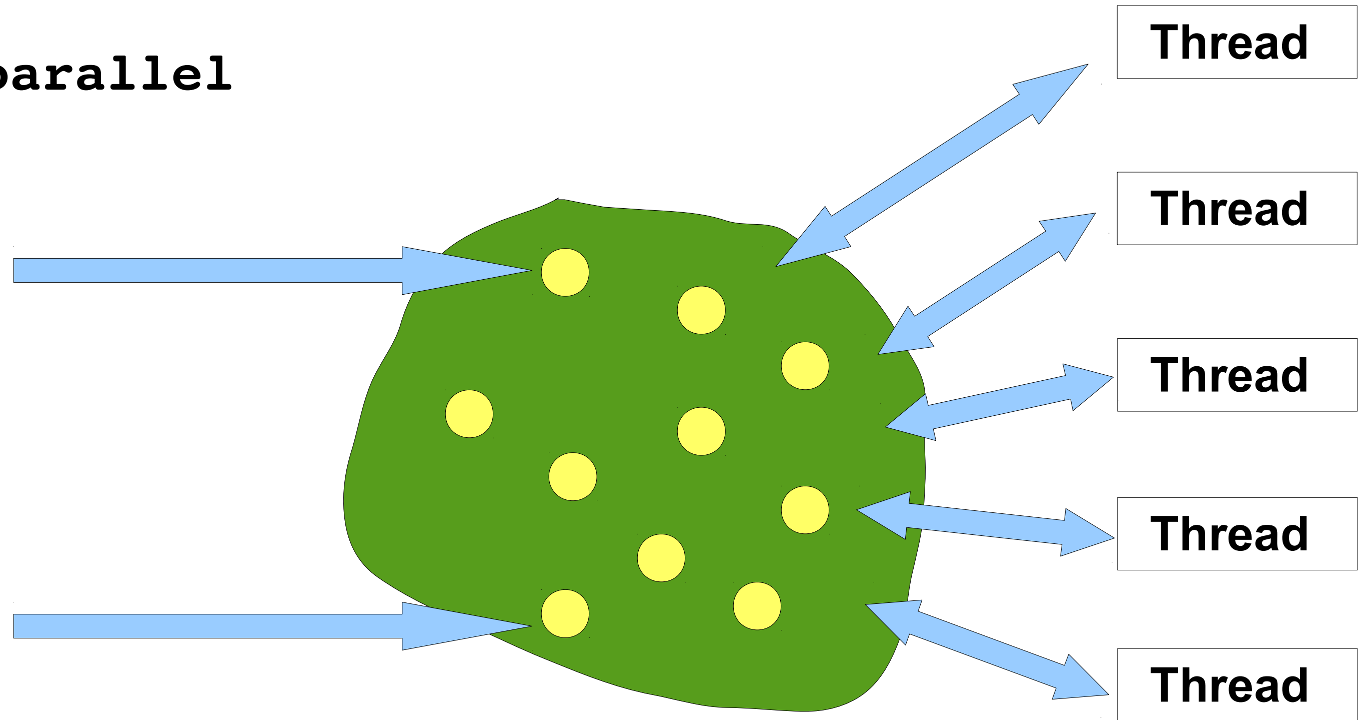
Tasking in OpenMP



```
#pragma omp parallel
...
```

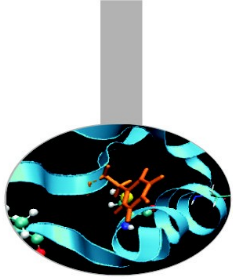
```
...
```






```
...
```

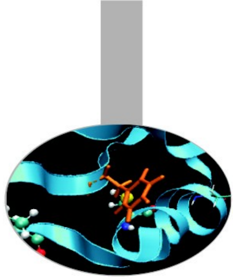
The assumption here is that tasks are **independent**



Task construct

```
#pragma omp task [clause[[,]clause] ...]
{
      structured-block
}
```

- Explicit task construct
- a task can be executed immediately or delayed (deferred)
- Runtime system will decide when the task is executed
- Tasks can be nested

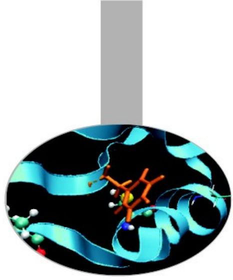


Definitions

- **Task construct** – task directive plus structured block

```
#pragma omp task [clause[[,]clause] ...]
   structured-block
```

- **Task** – instructions and data created when a thread encounters a task contruct
  - Different encounters of the same task construct generate different tasks
- **Task region** – all the code encountered during the execution of a task



Task example

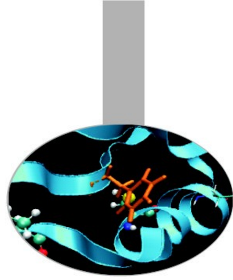
- Let's write a code that prints

  “A“
  “long“
  “run”

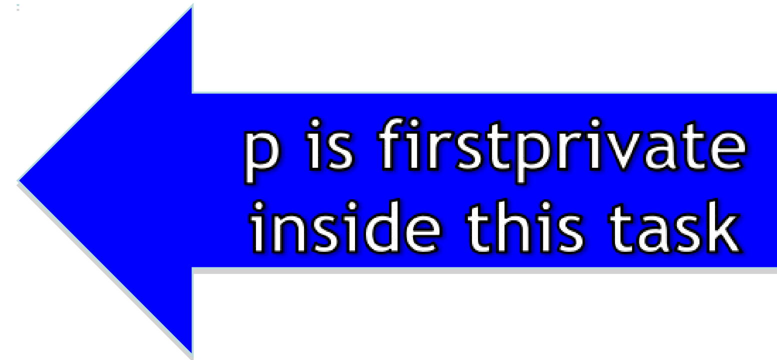
  in any order exploiting all the cores of the system

SCAI SuperComputing Applications and Innovation

Pointer chasing using tasks



```
#pragma omp parallel private(p)
    #pragma omp single
    {
        p = head;
        while(p) {
            #pragma omp task
                process(p);
            p=p->next;
        }
    }
```

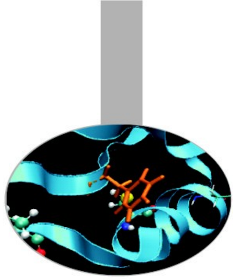


- One thread creates tasks
  - packages code and data environment
- When it finishes, it reaches the implicit barrier and starts to execute the tasks
- The other threads reach straight the implicit barrier and start to execute tasks

OpenMP

CINECA

Data scoping in explicit tasks



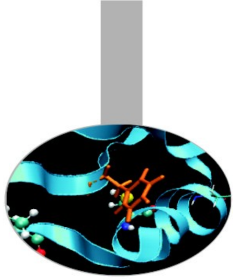
- **`private`** and **`firstprivate`**: business as usual

  Example:

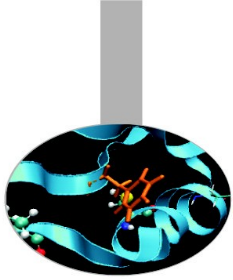
  a = 1, b = 1, c = 1
  #pragma omp parallel private(b) firstprivate(c)

- Inside the parallel region
  - a (shared) 1
  - b (private) undefined
  - c (private) 1

Data scoping in explicit tasks

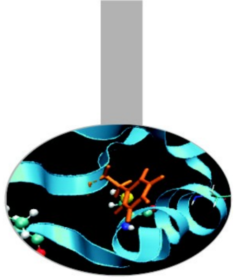


- **`private`** and **`firstprivate`**: business as usual
  - If a variable is private on a task construct, the references to it inside the construct are to new uninitialized storage that is created when the task is executed
  - If a variable is firstprivate on a construct, the references to it inside the construct are to new storage that is created and initialized with the value of the existing storage of that name when the task is encountered



Data scoping in explicit tasks

- **`shared`**: same business, from a new perspective
  - shared among all tasks (“horizontal”)
  - shared among a task and a descendant (“vertical”)
  - If a variable is shared on a task construct, the references to it inside the construct are to the storage with that name at the point where the task was encountered



Data scoping in explicit tasks

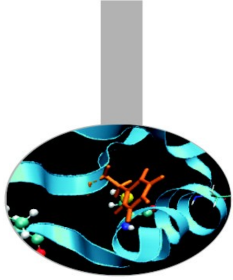
The behavior you want for tasks is usually **firstprivate**, because the task may not be executed until later (and variables may have gone out of scope)

Variables that are private when the task construct is encountered are **firstprivate** by default

Variables that are shared in all constructs starting from the innermost enclosing parallel construct are **shared** by default

Use default(none) to help avoid races!!!

CINECA SCAI
SuperComputing Applications and Innovation



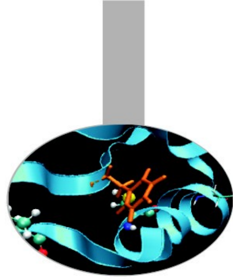
Task data scoping example

```
#pragma omp parallel shared(a) private(b)
{
    ...
    #pragma omp task
        int c;
        process(a,b,c);
    }
}
```

a is shared
b is firstprivate
c is private

OpenMP

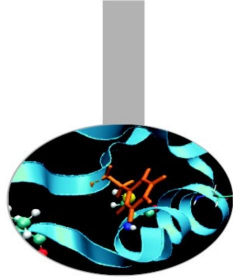
CINECA



Task data scoping example

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d = 4;
        #pragma omp task
        {
                int e = 5;

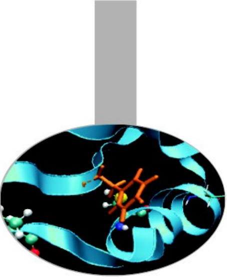
                // Scope of a:
                // Scope of b:
                // Scope of c:
                // Scope of d:
                // Scope of e:
} } }
```



Task data scoping example

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d = 4;
        #pragma omp task
        {
                int e = 5;

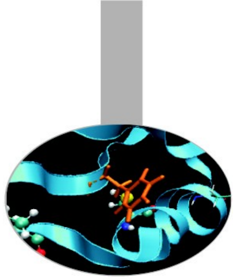
                // Scope of a: shared
                // Scope of b: firstprivate
                // Scope of c: shared
                // Scope of d: firstprivate
                // Scope of e: private
} } }
```



Outline

- Introduction to OpenMP
- General characteristics of Taks
- **Some examples**
- Live Demo

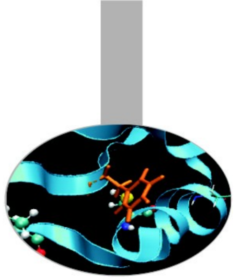
SuperComputing Applications and Innovation



Load balancing on lists with tasks

```
#pragma omp parallel
{
   #pragma omp for private(p)
   for (i=0; i<num_lists; i++) {
      p = heads[i];
      while(p) {
      #pragma omp task
         process(p);
      p=p->next;
      }
   }
}
```

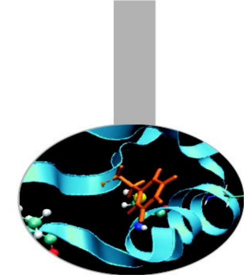
- Assign one list per thread could be unbalanced
- Multiple threads create tasks
- All the team cooperates executing them



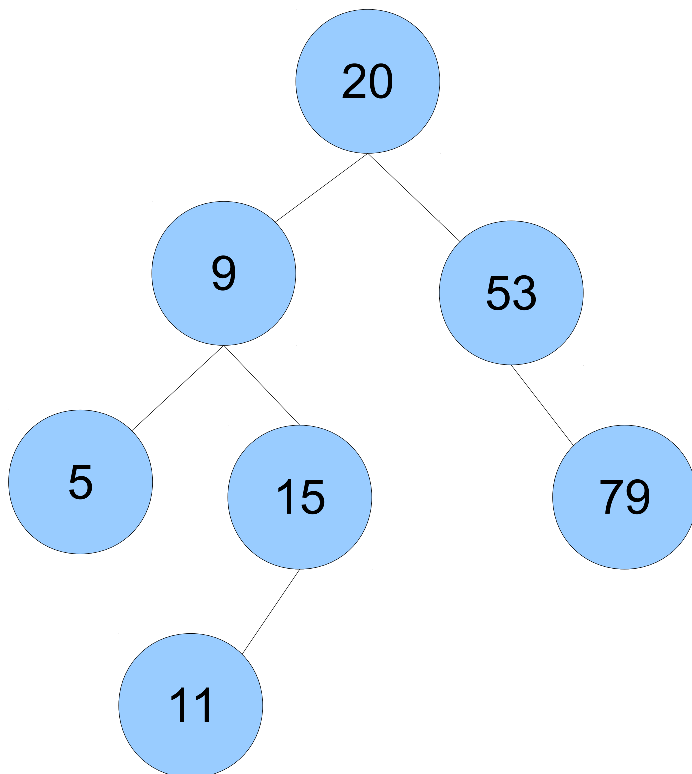
Tree traversal with task

```
void preorder(node *p) {
   process(p->data);
   if (p->left)
   #pragma omp task
      preorder(p->left);
   if (p->right)
   #pragma omp task
      preorder(p->right);
}
```

- Tasks are composable
- It isn’t a worksharing construct
- But what about postorder traversal?



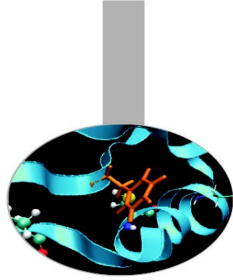
Tree traversal with task



Postorder: LRN

5, 11, 15, 9, 79, 53, 20

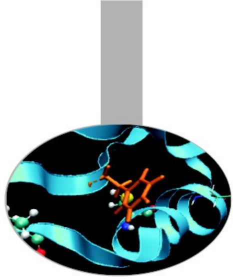
Postorder tree traversal



```
void postorder(node *p) {
  if (p->left)
    #pragma omp task
      postorder(p->left);
  if (p->right)
    #pragma omp task
      postorder(p->right);
  #pragma omp taskwait

  process(p->data);
}
```

- Parent task suspended until children tasks complete



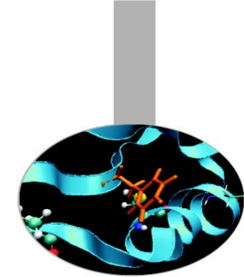
When/where explicit tasks complete?

- `#pragma omp taskwait`
  - applies only to siblings, not to descendants
  - task is suspendended until siblings complete

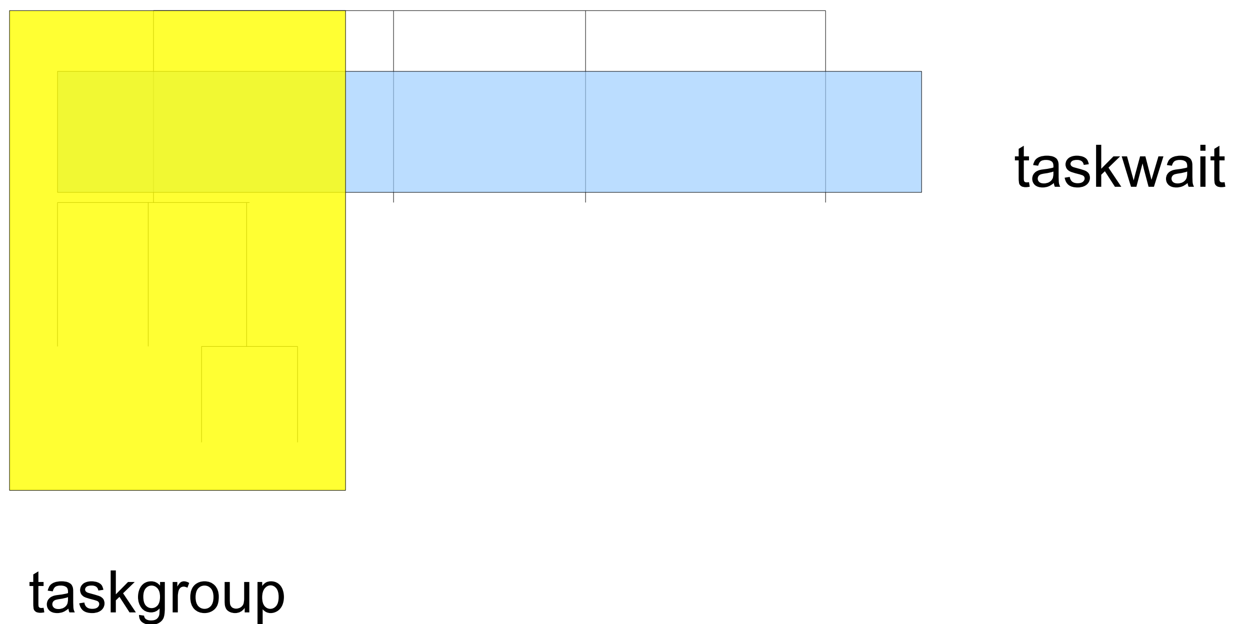
- ```
  #pragma omp taskgroup
  {
   create_a_group_of_tasks(could_create_nested_task)
   }
  ```
  - at the end of the region current task is suspended until all child tasks generated in the region and their descendants complete execution

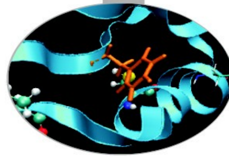
- `#pragma omp barrier`
  – applies to all tasks generated in the current parallel region up to the barrier
  – matches user expectation
  – obviously applies also to implicit barriers
  ```

SuperComputing Applications and Innovation



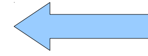
When/where explicit tasks complete?

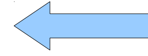


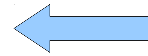


SparseLU

```
1  int sparseLU() {
2    int ii, jj, kk;
3    #pragma omp parallel
4      #pragma omp single nowait
5        for (kk=0; kk<NB; kk++) {
6          lu0(A[kk][kk]);
7          /* fwd phase */
8          for (jj=kk+1; jj < NB; jj++)
9            if (A[kk][jj] != NULL)
10             /* only create tasks for non-empty blocks */
11             #pragma omp task
12               fwd(A[kk][kk], A[kk][jj]);
13         /* bdiv phase */
14         for (ii=kk+1; ii < NB; ii++)
15           if (A[ii][kk] != NULL)
16             /* only create tasks for non-empty blocks */
17             #pragma omp task
18               bdiv (A[kk][kk], A[ii][kk]);
19         /* wait for previous tasks */
20         #pragma omp taskwait
21         /* bmod phase */
22         for (ii=kk+1; ii < NB; ii++)
23           if (A[ii][kk] != NULL)
24             for (jj=kk+1; jj < NB; jj++)
25               if (A[kk][jj] != NULL)
26                 /* only create tasks for non-empty blocks */
27                 #pragma omp task
28                 {
29                   if (A[ii][jj]==NULL)
30                     A[ii][jj]=allocate_clean_block();
31                   bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
32                 }
33         /* wait for all previous tasks */
34         #pragma omp taskwait
35       }
36 }
```

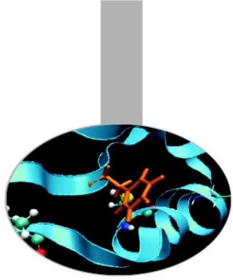






- **Fwd, bdiv** and **bmod** phases are responsible for load imbalance

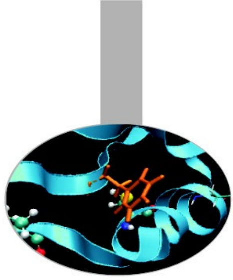
- Using task to operate only on non-empty blocks



task switching

- **What**: the ability of a thread to suspend the execution of a task and execute another one before resuming
- **Where**: at *task scheduling points:* **`task`**, **`taskwait`**, **`barrier`** directives, and implicit barriers

- **When**:
  - whenever is needed or useful
  - up to the implementation
- **Why**:
  - to lift pressure on runtime data structures

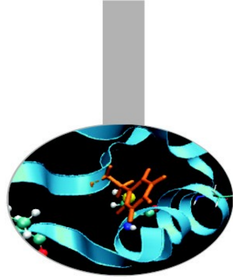


Lifting pressure on runtime

```
#pragma omp single
{
  for (i=0; i<ONEZILLION; i++)
    #pragma omp task
      process(item[i]);
}
```

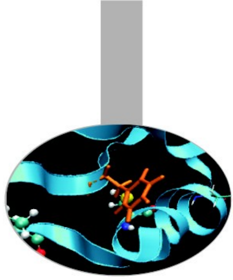
- Too many tasks generated in an eye-blink
- Generating task will have to suspend for a while
- With task switching, the executing thread can:
  - execute an already generated task (draining the "*task pool*")
  - dive into the encountered task (could be very cache-friendly)

thread switching



```
#pragma omp single
{
  #pragma omp task   untied
    for (i=0; i<ONEZILLION; i++)
      #pragma omp task
        process(item[i]);
}
```

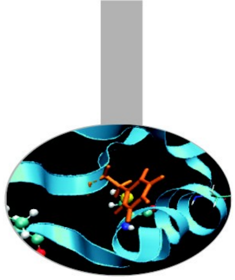
- Eventually, too many tasks are generated
- Generating task is suspended and executing thread switches to a long and boring task
- Other threads get rid of all already generated tasks, and start starving…
- With thread switching, the generating task can be resumed by a different thread, and starvation is over
- Too unsafe to be the default, the programmer is responsible!



The `if` clause

- When the **`if`** clause argument is false
  - the encountered task is executed immediately by the encountering thread, and the enclosing task is suspended up to its end
  - the data environment is still local to the new task
  - and it's still a different task wrt. Synchronization
  - does not apply to descendants

- It's a user directed optimization
  - when the cost of the task is comparable to the runtime overhead
  - to control cache and memory affinity

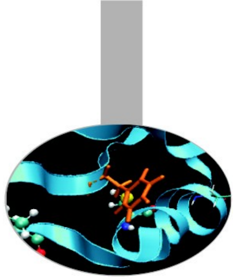


The `final` clause

- When the **`final`** clause argument is true
  - the generated task will be a final task
  - all tasks encountered during execution of a final task will generate included tasks
    - an included task is a task for which execution is sequentially included in the generating task region; that is, it is undeferred and executed immediately by the encountering threads
- It's another user directed optimization

- **`omp_in_final()`** returns true if the enclosing task region is final. Otherwise, it returns false

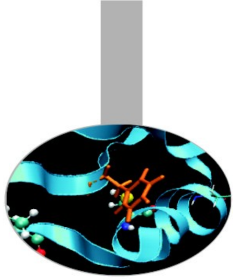
SuperComputing Applications and Innovation



Example: if and final

```
#pragma omp task if(0) // This is undeferred
{
  #pragma omp task // This is a regular task
     for( i = 0; i < 3; i++ ) {
       #pragma omp task // This is a regular task
         bar();
     }
}
#pragma omp task final(1) // This is a regular task
{
  #pragma omp task // This is included
    for(i=0;i<3;i++){
      #pragma omp task // This is also included
        bar();
    }
}
```

CINECA SCAI
SuperComputing Applications and Innovation



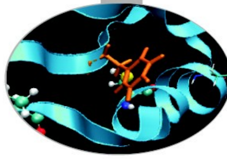
Conclusions on tasks

- Tasks allow to express a lot of irregular parallelism
- The tasking concept opens up opportunities to parallelize a wider range of applications

OpenMP

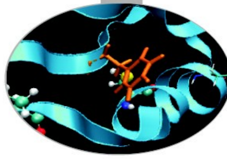
CINECA

Outline



- Introduction to OpenMP
- General characteristics of Taks
- Some examples
- **Live Demo**

SCAI SuperComputing Applications and Innovation



GOAL

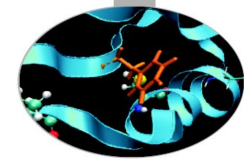
Impress your Grandma by implementing a parallel sudoku solver with OpenMP enad tasks

(credits Cristian Terboven)

OpenMP

CINECA

	6						8	11			15	14			16
15	11				16	14				12			6		
13		9	12					3	16	14		15	11	10	
2		16		11		15	10	1							
	15	11	10			16	2	13	8	9	12				
12	13			4	1	5	6	2	3					11	10
5		6	1	12		9		15	11	10	7	16			3
	2				10		11	6		5			13		9
10	7	15	11	16				12	13						6
9						1			2		16	10			11
1		4	6	9	13			7		11		3	16		
16	14			7		10	15	4	6	1				13	8
11	10		15				16	9	12	13			1	5	4
		12		1	4	6		16				11	10		
		5		8	12	13		10			11	2			14
3	16			10			7			6				12	



A brute force algorithm visits the empty cells in some order, filling in digits sequentially from the available choices, or backtracking (removing failed choices) when a dead-end is reached.

(wikipedia)