Cineca
TRAINING
High Performance
Computing 2017

OpenMP: advanced features

Mirko Cestari – m.cestari@cineca.it

Eric Pascolo – eric.pascolo@cineca.it

SuperComputing Applications and Innovation Department

CINECA

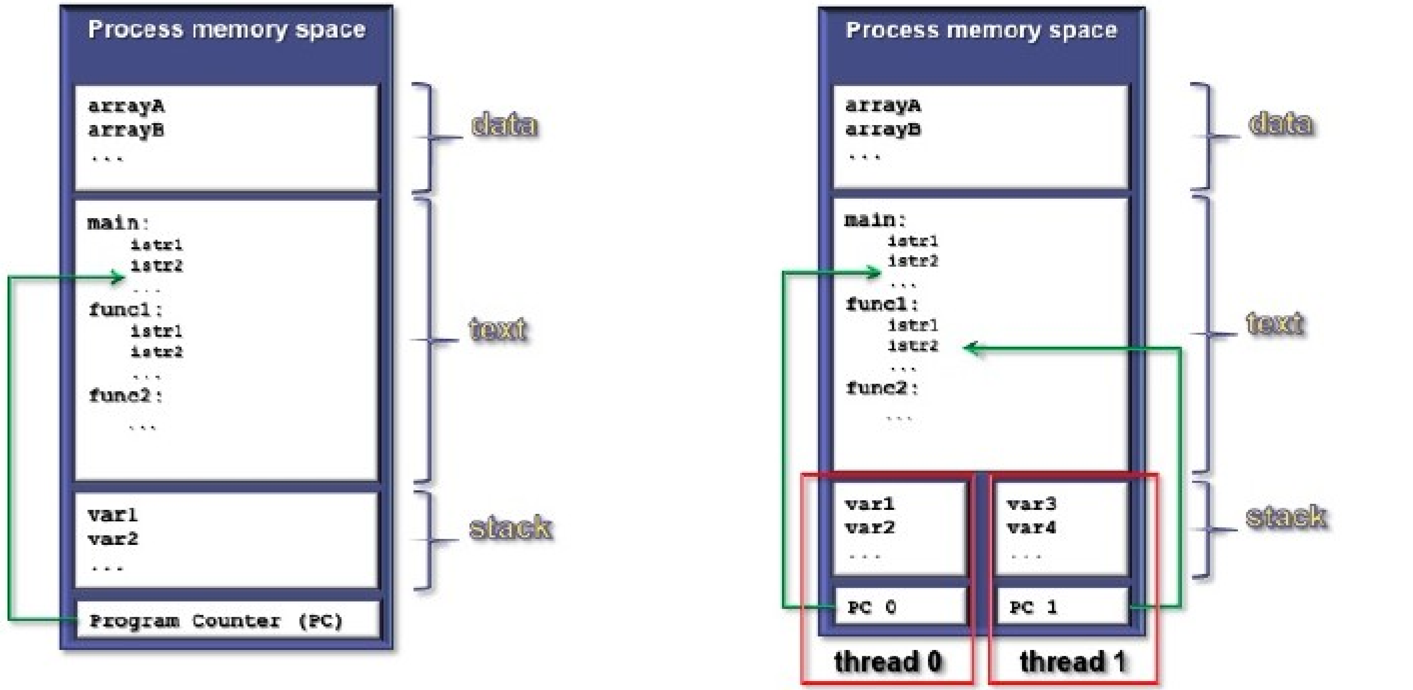
Outline

- **Introduction to OpenMP**
- Some technicalities
- General characteristics of Taks
- Some examples

Advantegs of OpenMP

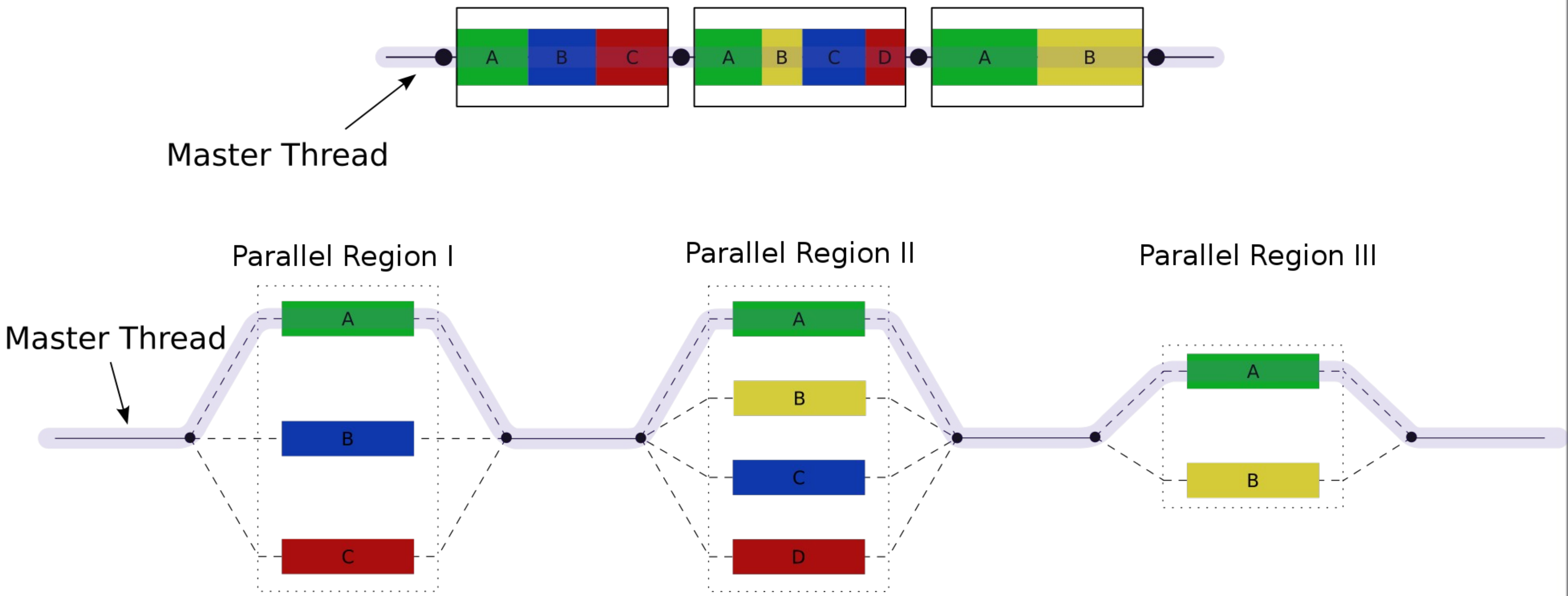
- **Standardized**
  - Enhance portability
- **Ease of use**
  - Limited set of directives
  - Fast (incremental) parallelization
- **Portability**
  - C, C++ And Fortran API
  - Part of many compilers
- **Can be used with MPI**
  - Hybrid code, I.e. to reduce impact of collective communications

Multi-threaded process



- Each thread may be regarded as a concurrent execution flow

Fork-Join parallel execution



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

OpenMP program

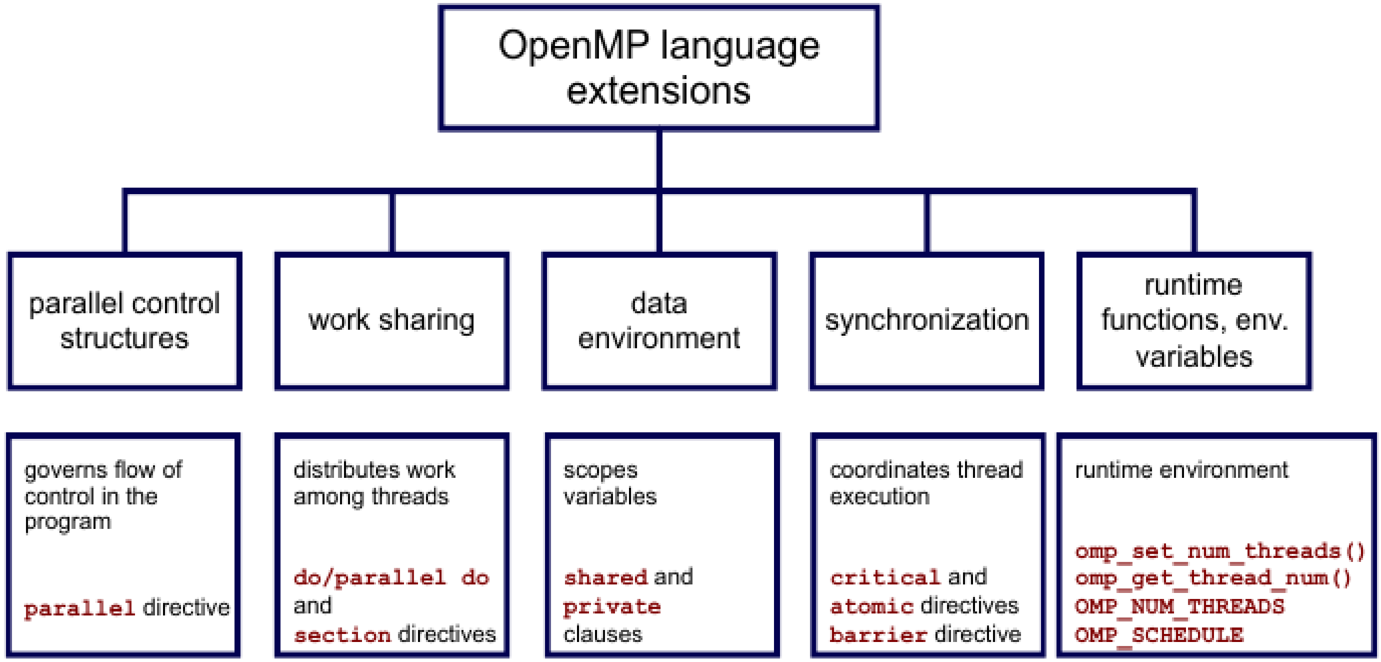
Shared-memory model

- 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 core elements



Outline

- Introduction to OpenMP
- **Some technicalities**
- General characteristics of Taks
- Some examples

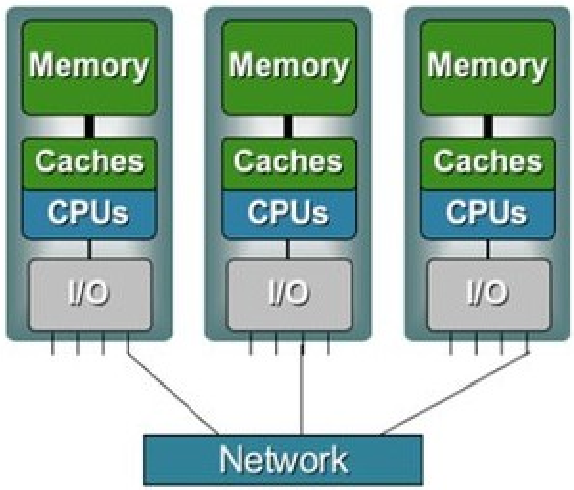
OpenMP: memory access

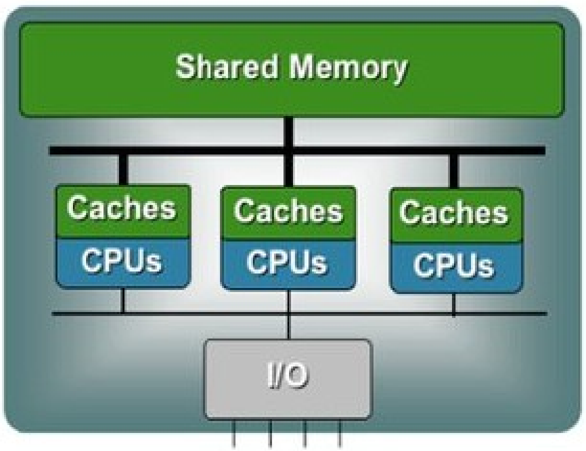
- OpenMP is not **cc-NUMA** aware
  - We need to take care of different **memory access**
  - Threads **placement** can be important

- Cache coherency plays a role
  - **False sharing**

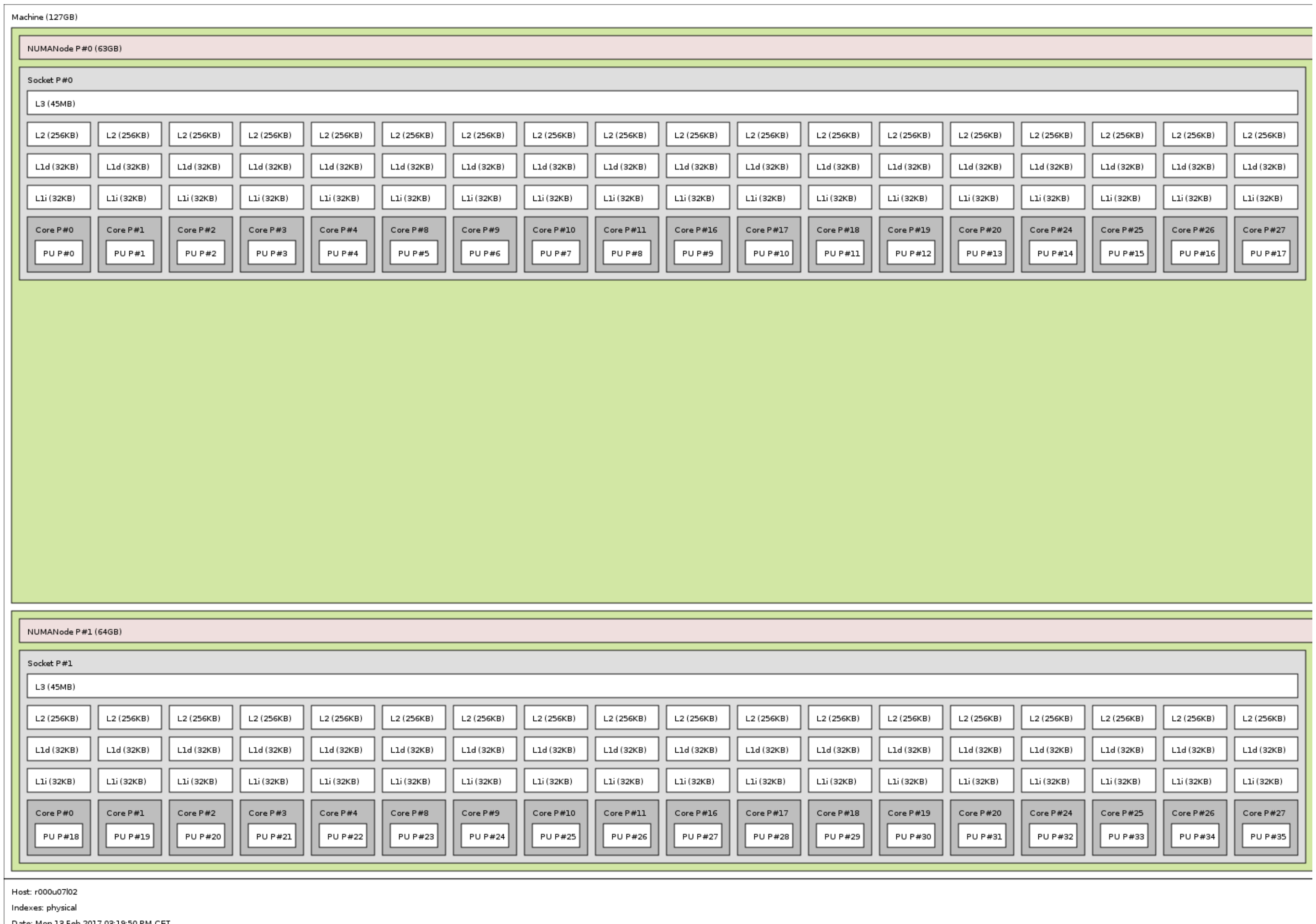
Shared memory systems

- Memory is **shared**
- Distinction between NUMA/UMA systems





cc-NUMA

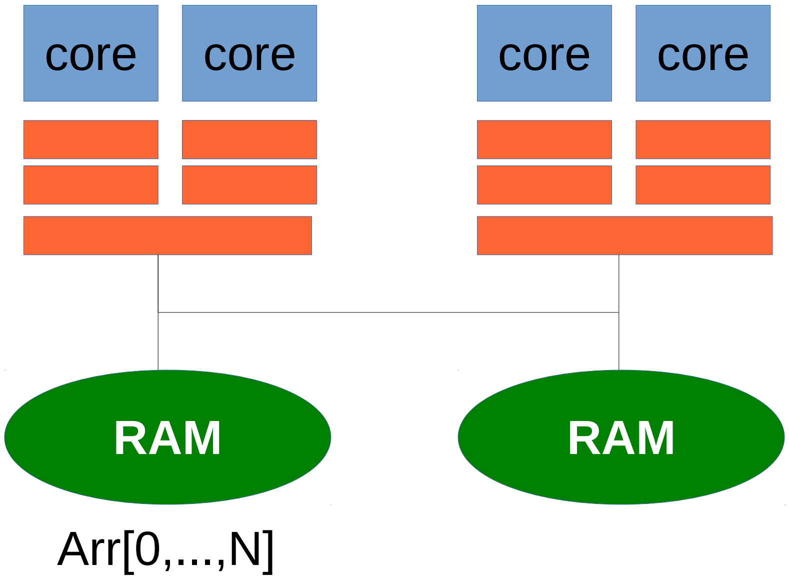

Machine (127GB)
NUMANode P#0 (63GB)
Socket P#0
L3 (45MB)
L2 (256KB)
L1d (32KB)
L1i (32KB)
Core P#0
PU P#0
Core P#1
PU P#1
Core P#2
PU P#2
Core P#3
PU P#3
Core P#4
PU P#4
Core P#8
PU P#5
Core P#9
PU P#6
Core P#10
PU P#7
Core P#11
PU P#8
Core P#16
PU P#9
Core P#17
PU P#10
Core P#18
PU P#11
Core P#19
PU P#12
Core P#20
PU P#13
Core P#24
PU P#14
Core P#25
PU P#15
Core P#26
PU P#16
Core P#27
PU P#17
NUMANode P#1 (64GB)
Socket P#1
L3 (45MB)
L2 (256KB)
L1d (32KB)
L1i (32KB)
Core P#0
PU P#18
Core P#1
PU P#19
Core P#2
PU P#20
Core P#3
PU P#21
Core P#4
PU P#22
Core P#8
PU P#23
Core P#9
PU P#24
Core P#10
PU P#25
Core P#11
PU P#26
Core P#16
PU P#27
Core P#17
PU P#28
Core P#18
PU P#29
Core P#19
PU P#30
Core P#20
PU P#31
Core P#24
PU P#32
Core P#25
PU P#33
Core P#26
PU P#34
Core P#27
PU P#35
Host: r000u07l02
Indexes: physical
Date: Mon 13 Feb 2017 03:19:50 PM CET

Numa memory access

- Windows, Linux and other OS by default uses a **first touch** policy to place data in memory
- The core that *touches* the memory *owns* the data

```
Arr[0,...,N]

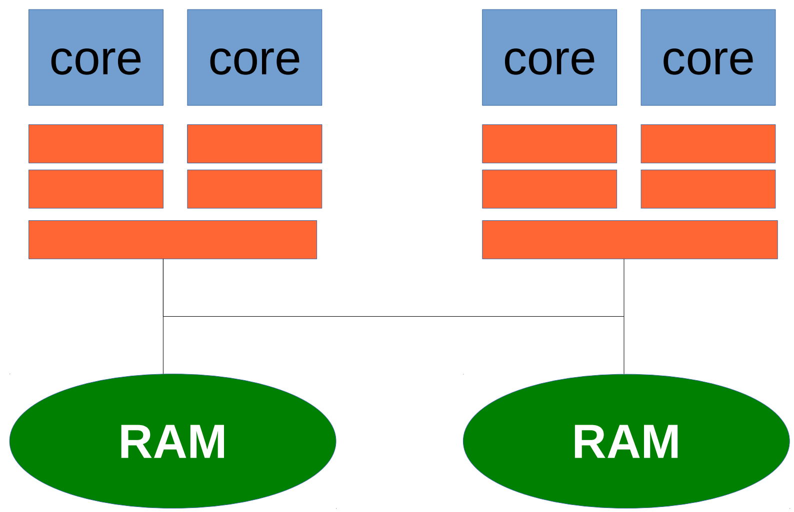
for (i=0;i<N;i++)
      Arr[i] = 1.0
```



Numa memory access

- Increases aggregated memory bandwidth (reduce latency)

for (i=0;i<N/2;i++)
Arr[i] = 1.0



Arr[0,...,N/2]

Arr[N/2,...,N]

for (i=N/2;i<N;i++)
Arr[i] = 1.0

OpenMP: thread placing

- Give more control to the programmer

- You need info on the system topology
  - **lstopo** from hwloc (cache dimension, topology)
  - **cpuinfo** (proc ids, physical cores, hardware threads)

OpenMP: thread placing

- Set OpenMP places: **OMP_PLACES**
    - sockets
    - cores
    - threads
    - …

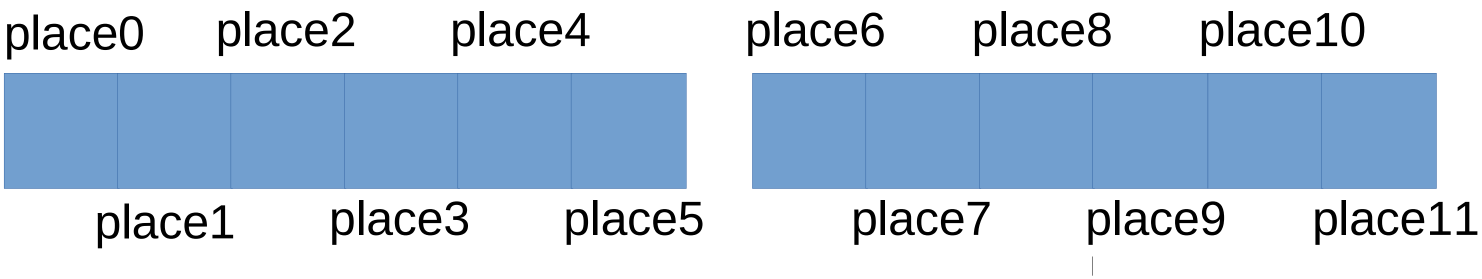
- Define thread binding: **OMP_PROC_BIND**
    - spread
    - close
    - master

thread placing: examples

Cineca TRAINING High Performance Computing 2017

2 sockets, 6 cores per socket

OMP_PLACES=**cores**

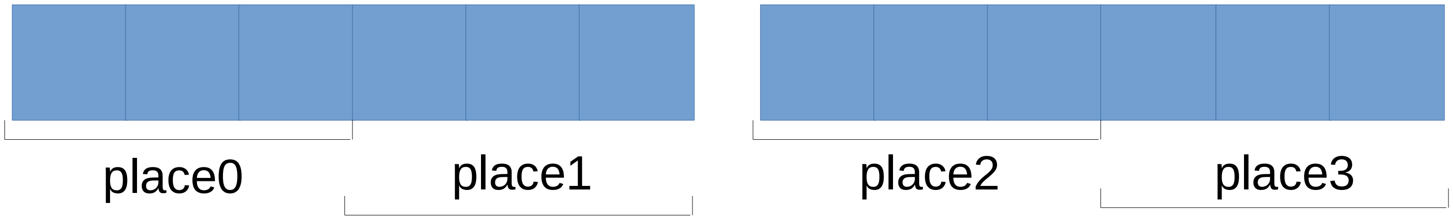


OMP_PLACES=**sockets**

thread placing: examples

2 sockets, 6 cores per socket

OMP_PLACES="{0:3},{3:3},{6:3},{9:3}"



OpenMP: thread binding

Define thread binding: **OMP_PROC_BIND**

- **true/false**
- **spread:** spread threads evenly among the places (useful for nesting)
- **close**: starts by placing T/P threads in parent thread's place, and then proceeds in a round-robin fashion allocating T/P threads in each place
- **master**: the threads in the team are assigned to the same place as the master thread

Thread binding API

New in **OpenMP 4.5**

- int omp_get_num_places()
- int omp_get_place_num_procs(int place_num)
- void omp_get_place_proc_ids(int place_num, int *ids)
- int omp_get_place_num(void)

Goals of binding

- Avoid that threads move among the cores (losing all caches)
- If threads share cache
  - useful if they are working on same data
    - caches do not replicate same data (increasing effective cache size, decreasing cache misses)
  - false sharing can be mitigated
- If threads do not share cache
  - useful if the don't work on the same data
  - cache don't compete for data (increasing effective cache size, decreasing cache misses)

CPU caches (1)

- All data read or written by the CPU cores is stored in the cache
- If the CPU needs data caches are searched first
- Data is stored in lines (typically 64 bytes)

  `sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size`
- When memory content is needed by the CPU the entire cache line is loaded into L1d
- A cache line which holds values that are not yet written to main memory or higher-level caches is said to be "**dirty**"
- Eventually the dirty bit will tell the processor to write the data back before discarding the data

CPU caches (2)

- All processors are supposed to see the same memory content at all times
  - cache coherency
- When a second processor needs a value from a dirty line of a processor:
  - the processor **sends the content** of the cache line to the second processor
  - if the value is required to be written on, the first processor needs to **invalid** the cache line. It will need to read the new content from a higher-level cache or main memory
    - False sharing

False sharing

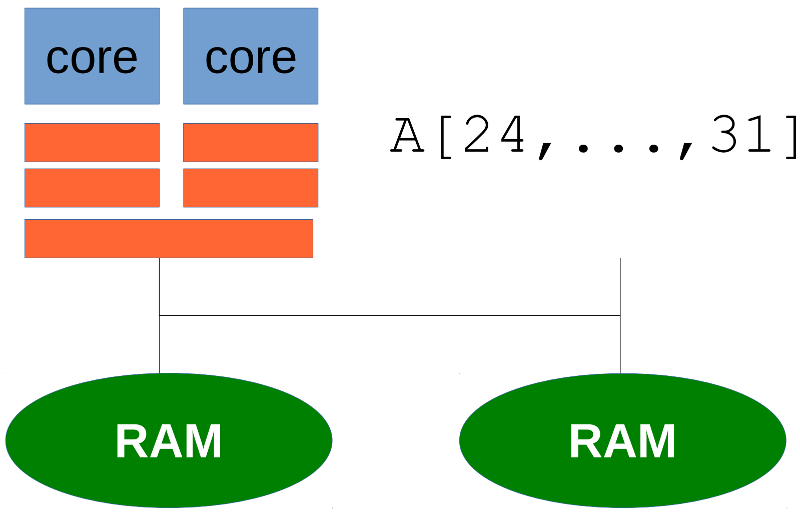
All cores working on same data: A[0,...,100]

A[0,...,24]

A[25,...,49]

A[24,...,31]

A[24,...,31]



A[0,...,100]

Outline

- Introduction to OpenMP
- Some technicalities
- **General characteristics of Tasks**
- Some examples

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
- **sections contruct**: sections are statically defined at compiled time

Synchronization constructs 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 leverage these kinds of concurrency

Pointer chasing using single

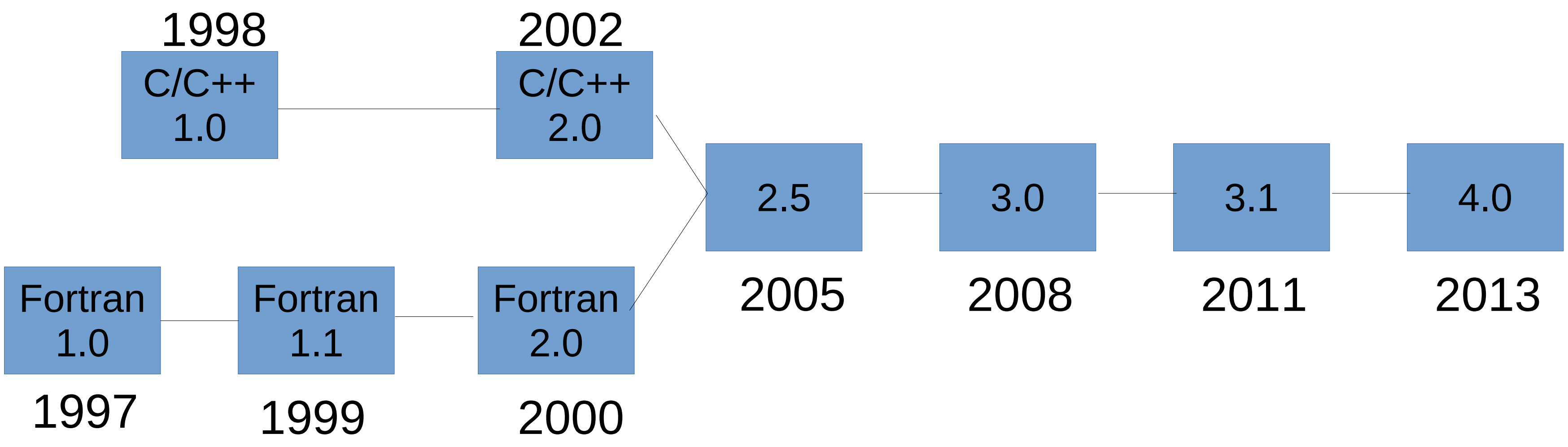
```
#pragma omp parallel private(p)
{
   p = head;
   while(p) {
      #pragma omp single nowait
         process(p);
      p=p->next;
   }
}
```

- 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

Tasks

- **First Introduced in OpenMP 3.0**
  - has been the major addition from OpenMP 2.5

- **Refined in OpenMP 3.1, 4.0 and 4.5**



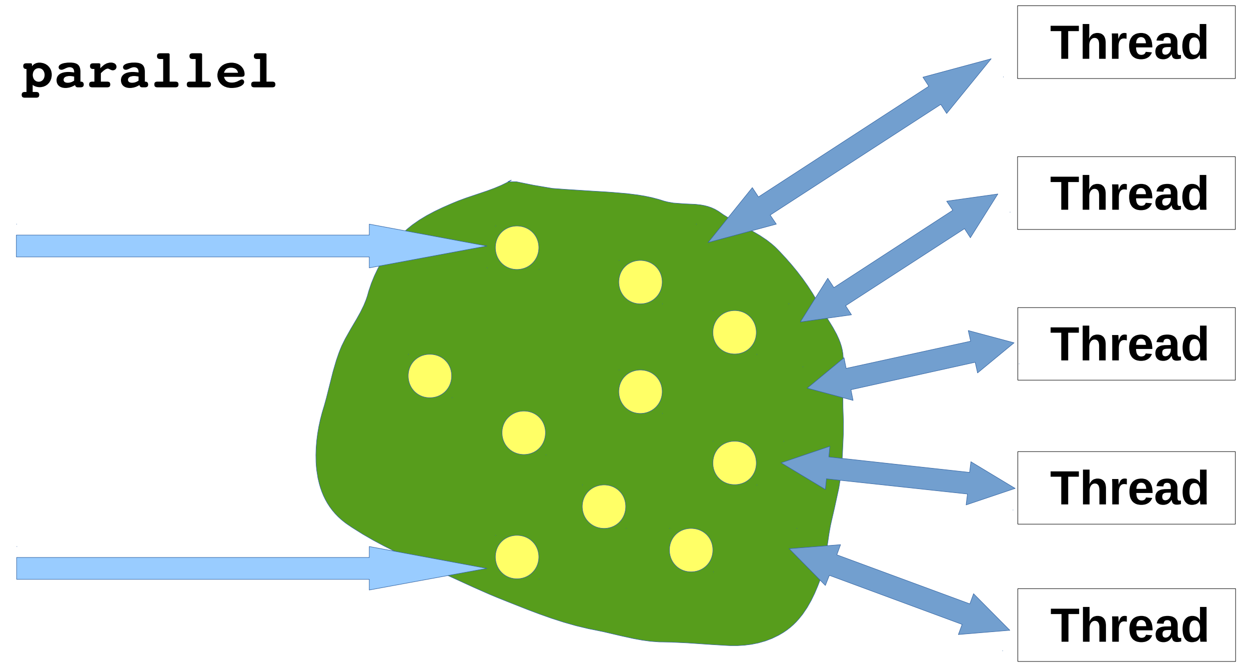
Tasking

- From a thread-centric model to a **task centric-model**
- A model in which users identify **independent unit of work**
  - i.e. intrinsically unbalanced
  - 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
...
{
    [code]
}
...
{
    [code]
}

...
```



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 **deferred**
- **runtime system** will decide when the task is executed
- tasks can also 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

Pointer chasing using tasks

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

p is firstprivate inside this task

- One thread creates tasks
- When it finishes, it reaches the implicit barrier and starts to execute the tasks
- The other threads directly go the implicit barrier and start to execute the tasks

Data scoping in 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 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 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 tasks

The behavior you want for tasks is usually **firstprivate**, because tasks 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!!!

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

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
- Some technicalities
- General characteristics of Taks
- **Some examples**

Load balancing of lists

```
#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?

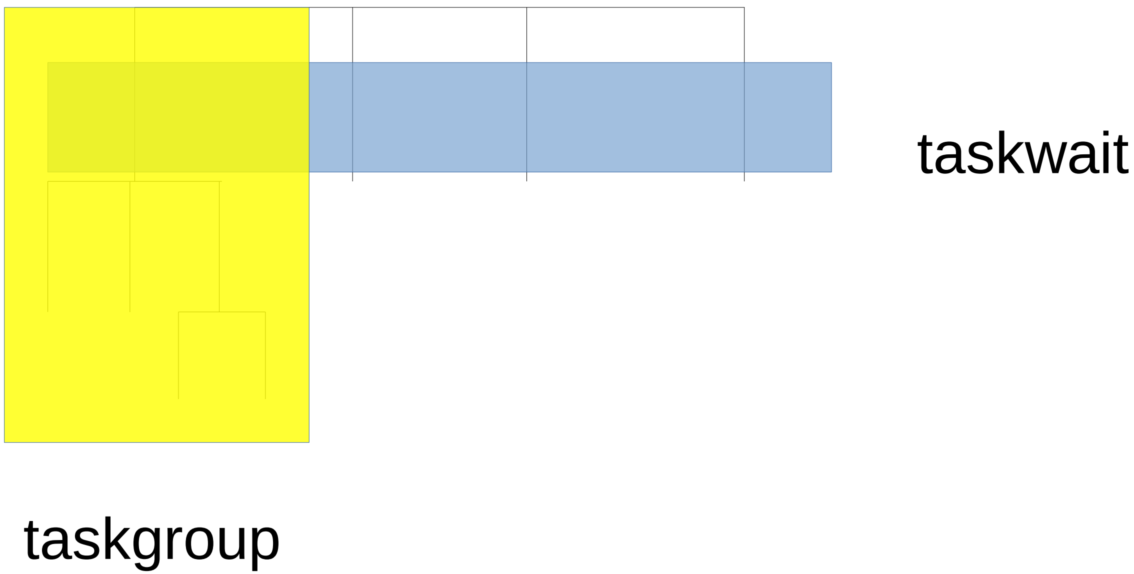
When/where explicit tasks complete?

- `#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
  ```

When/where explicit tasks complete?



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 with 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

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