www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Developing HPC applications with OpenMP and TA-X (MPI, CUDA)

Highly Efficient Accelerators and Reconfigurable Technologies (HEART) - 2024

Xavier Teruel and Kevin Sala

EXCELENCIA SEVERO OCHOA

Porto, June 21st, 2024

Agenda and lecturers

"Developing HPC applications with OpenMP, Task-Aware MPI (TAMPI) and Task-Aware CUDA (TACUDA)"

Time	Date: June 21st
10:30	**Introduction to the tasking model**
11:30	*Tasking - Q&A*
11:45	**Hybrid programming with (TAMPI)**
12:30	*- LUNCH -*
14:00	*Hybrid - Q&A*
14:15	**Heterogeneous systems (TACUDA)**
15:00	*Heterogeneous - Q&A (and wrap-up)*
15:30	***Adjourn***

Lecturers

Xavier Teruel

Team leader

Best Practices for Performance and Programmability

- *He will lecture tasking model*

Kevin Sala

PhD Candidate

Runtime systems for parallel programing models

- *He will lecture TAMPI and TACUDA*

Outline: introduction to the tasking model

OpenMP brief introduction

- Overview, main components, the fork-join model, syntax, parallel region and worksharing constructs

Task creation and scheduling

- Task execution model, task construct, data environmnet, tied vs untied, if, mergeable, final

Task synchronization

- Tasks and barriers, taskwait, taskgroup, dependences

Taskloop construct

- Number of tasks vs grain of the task, collapse, nogroup

OpenMP overview

Parallel Programming Model

- (initially) Designed for shared memory parallel computers
  - » single address space across the host memory system
- But now it also includes multi-device architectures (GPUs, Accelerators,…)
  - » it may imply additional (per device) address spaces
  - » support of data mapping from/to each address space

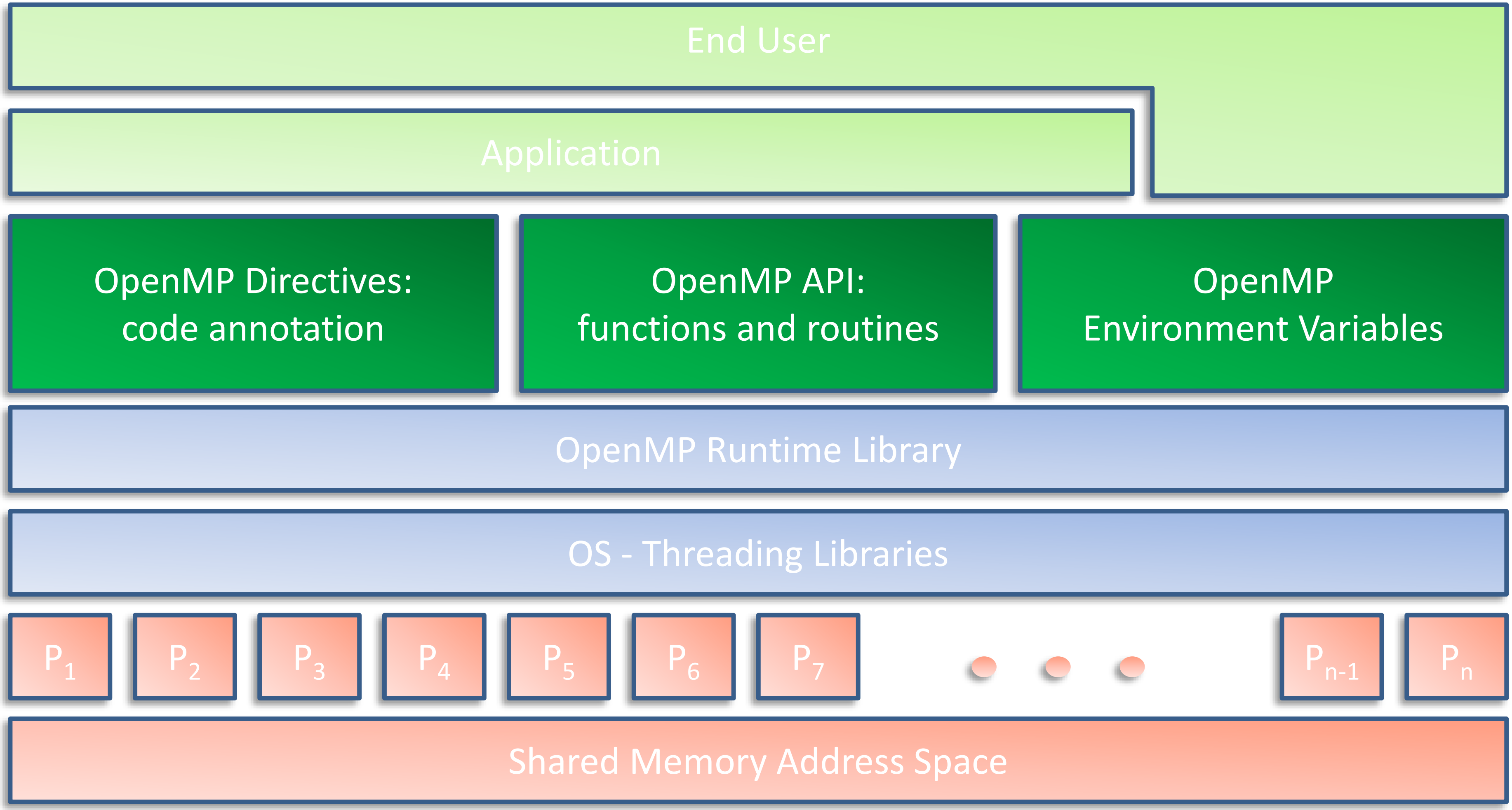
Maintained by the Architecture Review Board (ARB)

- Permanents members: AMD, ARM, Cray, Fujitsu, HP, IBM, Intel, Micron, NEC, NVIDIA, Oracle, Red Hat and Texas Instruments
- Auxiliary members: ANL, LLNL, BSC, cOMPunity, EPCC, LANL, LBNL, NASA, ORNL, RWTH, SNL, TACC and UH

Supported by most compiler vendors

- Intel, IBM, PGI, TI, Sun, Cray, Fujitsu, MS, HP, LLVM, GCC,…

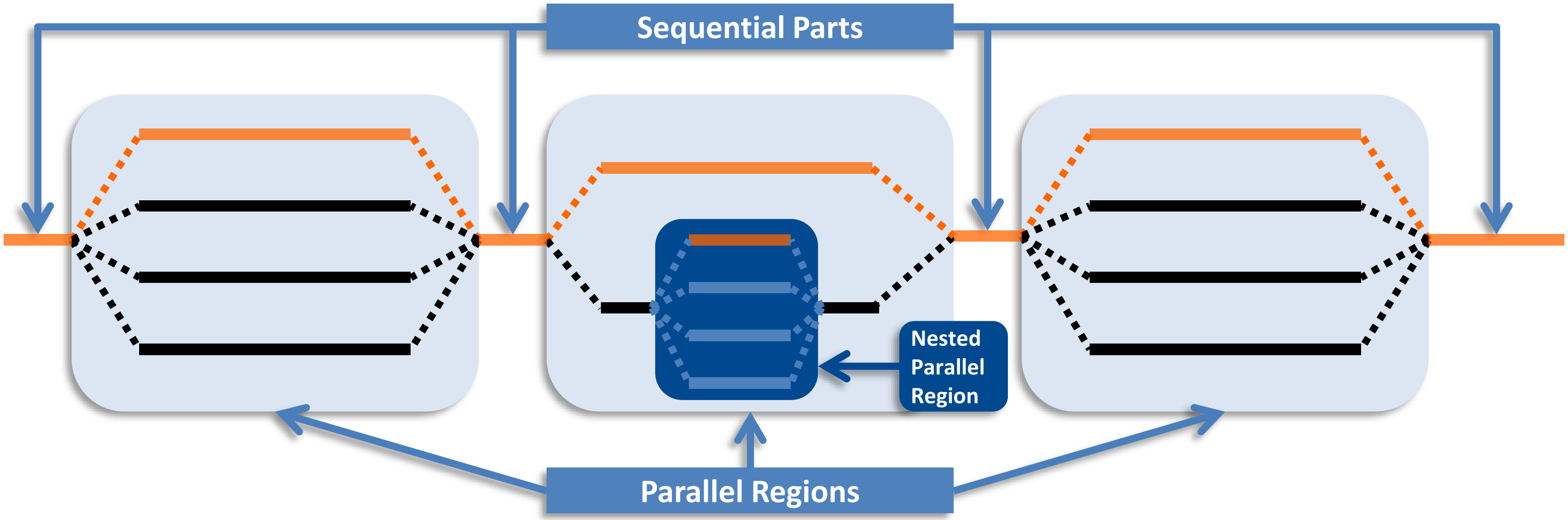
OpenMP components



Execution model

Based on the fork-join paradigm

- a thread team is a set of threads which co-operate on a region
- the **primary thread** is responsible for coordinating the team
- usually running **one thread** per **processor** (but could be more / or less)
- different threads may follow different control flows



OpenMP (directive) syntax

In Fortran language

- through a specially formatted comment

```
sentinel directive-name [clause[[,] clause]...]
```

- where sentinel is one of
  - » `!$OMP` or `C$OMP` or `*$OMP` in fixed format
  - » `!$OMP` in free format
- API runtime services
  - » omp_lib module contains the subroutine and function definitions

In C/C++ language

- using compiler directives*

```
#pragma omp directive-name [clause[[,] clause]...]
```

- API runtime services
  - » omp.h contains the API prototypes and data types definitions

** directives are ignored if compiler does not recognize OpenMP*

The parallel region

When two “blocks of code” may run in parallel…

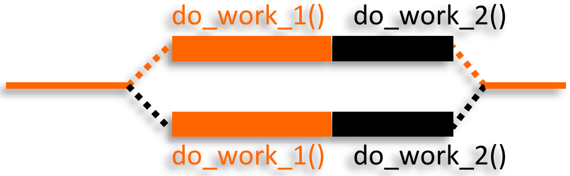
```
#include <stdio.h>

void main (void)
{
  do_work_1();
  do_work_2();
}
```

```
$ ./myProgram
```

… we just include them within a parallel region (replicate)

```
#include <stdio.h>
#include <omp.h>
void main (void)
{
  #pragma omp parallel num_threads(2)
  {
    do_work_1();
    do_work_2();
  }
}
```

```
$./myProgram
```



Worksharing: introduction

Divide the execution of a code region among the threads of a team

- threads cooperate to do some work (i.e. to share some work)
- better way to split work than using thread-ids
- lower overhead than using tasks → less flexible

In OpenMP, there are four worksharing constructs:

- single construct
- sections construct
- loop construct
- workshare construct (only Fortran)

Restriction: worksharings cannot be nested

Worksharing: the single construct

Serializing (1-thread) a portion of the parallel region

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

Where clause:

- private(list)
- firstprivate(list)
- nowait
- copyprivate(list)

Semantics: only one thread of the team executes the structured block

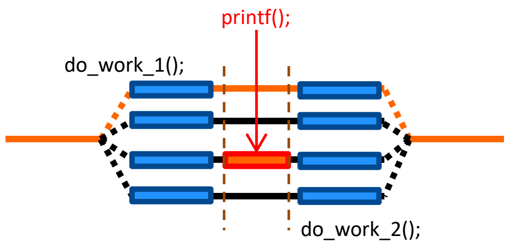
Very useful in I/O operations

Example:

```
#pragma omp parallel
 {
   do_work_1();
   #pragma omp single
   {
     printf ("Hello world!\n") ;
   }
   do_work_2();
 }
```

This program writes just one "Hello world!"

```
$ OMP_NUM_THREADS=4 ./myProgram
```



Worksharing: the sections construct

Set of structured blocks distributed among threads

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

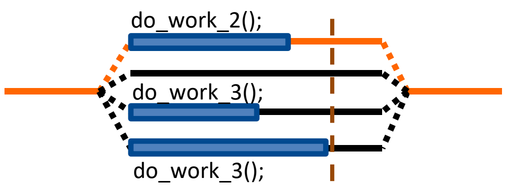
Where clause:
- private(list)
- firstprivate(list)
- lastprivate(list)
- reduction(operator: variable-list)
- nowait

Semantics: sections distributed among threads

Example:

```
#pragma omp parallel sections
 {
   do_work_1();
   #pragma omp section
   do_work_2();
   #pragma omp section
   do_work_3();
 }
```

```
$ OMP_NUM_THREADS=4 ./myProgram
```



Worksharing: the loop construct

Distributing a loop among threads

```
#pragma omp for [clause[[,] clause]...]
{structured-block: loop}
```

Where clause:

- private(list), firstprivate(list), lastprivate(list), reduction(operator: list)
- schedule(schedule-kind)
- nowait, collapse(n), ordered

Semantics: distributes the loop iteration space among the threads

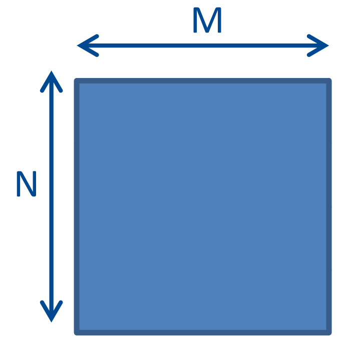
Matrix initialization (using the loop construct)

```
void foo ( int *m, int N, int M)
{
  int i, j ;
#pragma omp parallel for private( j )
  for ( i = 0; i < N; i ++ )
    for ( j = 0; j < M; j ++ )
      m[ i * N + j ] = 0;
}
```

New created threads cooperate to execute all the iterations of the loop

The i variable is automatically privatized

The j variable must be manually privatized



... but other distributions are also possible

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Task creation and scheduling

Highly Efficient Accel. and Reconfigurable Tech. (HEART) - 2024

Porto, June 21st, 2024

What is a task in OpenMP?

Tasks are work units whose execution may be deferred…
… or it can be executed immediately!!!

Tasks appears in OpenMP 3.0 specification (2008)

Tasks are composed of:
- code to execute (set of instructions, function calls, etc…)
- a data environment (initialized at creation time)
- internal control variables (ICVs)

In OpenMP tasks are created…
- when reaching a parallel region → implicit task are created per thread
- when encounters a task construct → explicit task is created
- when encounters a taskloop construct → explicit task per chunk is created
- when encounters a target construct → target task is created

Tasking execution model

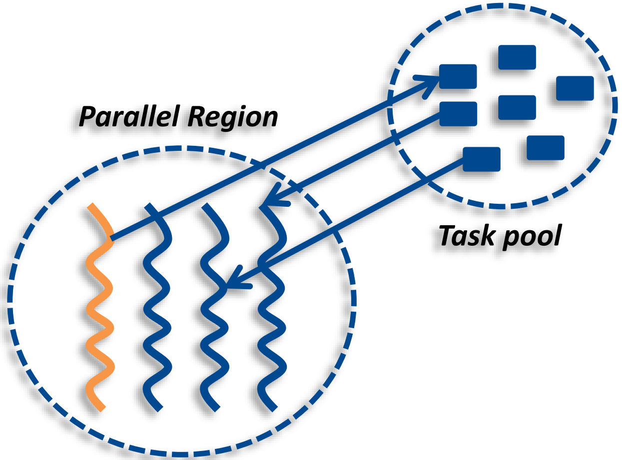
Supports unstructured parallelism

- unbounded loops

```
while ( <expr> ) {
   ...;
}
```

- recursive function calls

```
void myCode ( <args> ) {
   ...; myCode ( <args> ); ...;
}
```



Several scenarios are possible

- single creator vs. multiple creators…
- but all members in the team are candidates to execute these tasks

The task construct

Deferring a unit of work (executable for any member of the team)

- always attached to a structured block

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

Where clause:

- private(list), firstprivate(list), shared(list)
- default(shared | none)
- untied
- if(scalar-expression)
- mergeable
- final(scalar-expression)
- priority(priority-value)
- depend(dependence-type: list)

Data environment: role of a variable within a construct

Pre-determined data-sharing attributes
- threadprivate variables are threadprivate
- dynamic storage duration objects are shared (malloc, new,…)
- static data members are shared
- variables declared inside the construct
  - static storage duration variables are shared
  - automatic storage duration variables are private
- the loop iteration variable(s) are private

Explicit data-sharing clauses (shared, private, firstprivate,…)
- if default clause present, what the clause says
  - none means that the compiler will issue an error if the attribute is not explicitly set by the programmer (very useful!!!)

Implicit data-sharing rules for…

… worksharings:
- non pre-determined/explicit variables will be shared

… tasks:
- the shared attribute is lexically inherited
- in any other case the variable is firstprivate

Data sharing attributes: pre-determined

threadprivate variables are threadprivate **(1)**
dynamic storage duration objects are shared (malloc, new,…) **(2)**
static data members are shared **(3)**
variables declared inside the construct
- static storage duration variables are shared **(4)**
- automatic storage duration variables are private **(5)**

the loop iteration variable(s) are private

5
```
#pragma omp task
{
   int x = MN;
   // Scope of x: private
}
```

4
```
#pragma omp task
{
   static int y;
   // Scope of y: shared
}
```

1
```
int A[SIZE];
#pragma omp threadprivate(A)

// ...
#pragma omp task
{
  // A: threadprivate
}
```

2
```
int *p;

p = malloc(sizeof(float)*SIZE);

#pragma omp task
{
   // *p: shared
}
```

3
```
void foo(void){
   static int s = MN;
}

#pragma omp task
{
   foo(); // s@foo(): shared
}
```

Data sharing attributes: explicit and default

Explicit data-sharing clauses (shared, private and firstprivate)

```
#pragma omp task shared(a)
{
  // Scope of a: shared
}
```

```
#pragma omp task private(b)
{
  // Scope of b: private
}
```

```
#pragma omp task firstprivate(c)
{
  // Scope of c: firstprivate
}
```

If **default** clause present, what the clause says

- shared: data which is not explicitly included in any other data sharing clause will be **shared**
- none: compiler will issue an error if the attribute is not explicitly set by the programmer (very useful!!!)

```
#pragma omp task default(shared)
{
 // Scope of all the references, not explicitly
 // included in any other data sharing clause,
 // and with no pre-determined attribute: shared
}
```

```
#pragma omp task default(none)
{
 // Compiler will force to specify the scope for
 // every single variable referenced in the context
}
```

Hint: Use default(none) to be forced to think about every variable if you do not see clearly.

Data sharing attributes: implicit

Pre-determined data-sharing attributes

- threadprivate variables are threadprivate
- dynamic storage duration objects are shared (malloc, new,…)
- static data members are shared
- variables declared inside the construct
  - static storage duration variables are shared
  - automatic storage duration variables are private
- the loop iteration variable(s) are private

Explicit data-sharing clauses (shared, private, firstprivate,…)

- if default clause present, what the clause says
  - none means that the compiler will issue an error if the attribute is not explicitly set by the programmer (very useful!!!)

Implicit data-sharing rules for…

… worksharings:

- non pre-determined/explicit variables will be shared

… tasks:

- the shared attribute is lexically inherited
- in any other case the variable is firstprivate

Task default data-sharing attributes (in practice)

```
int a ;
void foo ( int b ) {
   int c;
   #pragma omp parallel private( c )
   {
      int d ;
      #pragma omp task
      {
         int e;
         a = <expr>;
         b = <expr>;
         c = <expr>;
         d = <expr>;
         e = <expr>;
         g = <expr>;
      }
   }
}
```

– default(none) may help when you are not sure of understand the default

Task scheduling: tied vs untied tasks (1)

Tasks are tied by default (when no untied clause present)

- tied tasks are executed always by the same thread (*not necessarily creator*)
- tied tasks "may" run into performance problems

Programmers may specify tasks to be untied (relax scheduling)

```
#pragma omp task untied
{structured-block}
```

- can potentially switch to any thread (of the team)
- bad mix with thread based features: thread-id, threadprivate, critical regions...
- gives the runtime more flexibility to schedule tasks

Task scheduling: tied vs untied tasks (2)

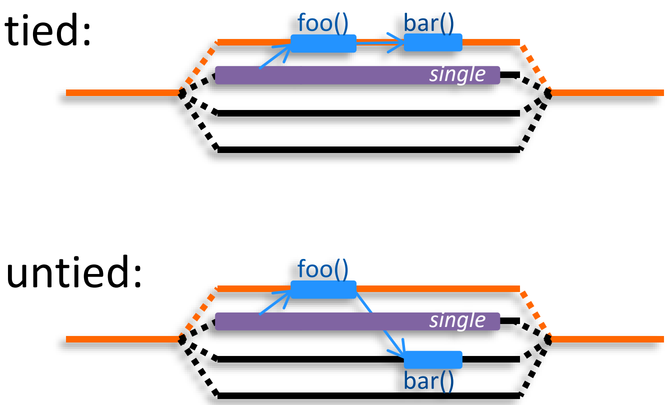
Task scheduling points (and the taskyield directive)

- tasks can be suspended/resumed at these points
- some additional constraints to avoid deadlock problems
- implicit scheduling points (creation, synchronization, ...)
- explicit scheduling point: the taskyield directive

```
#pragma omp taskyield
```

Scheduling untied tasks: example

```
#pragma omp parallel
#pragma omp single
{
   #pragma omp task [untied]
   {
      foo ();
      #pragma omp taskyield
      bar ();
   }
}
```



Controlling task scheduling (1)

The if clause of a task construct

- allows to optimize task creation/execution → reduces parallelism but also reduces the pressure in the runtime's task pool
- for "very" fine grain tasks you may need to do your own (manual) if

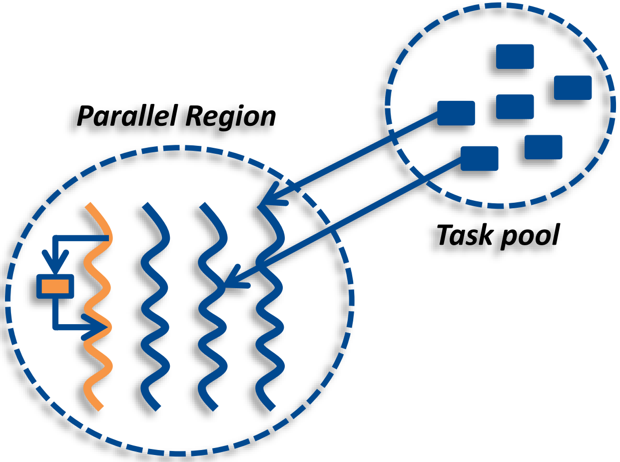
```
#pragma omp task if(expresion)
{structured-block}
```

If the expression of the "if" clause evaluates to false

- the encountering task is suspended
- the new task is executed immediately
- the parent task resumes when the task finishes

This is known as undeferred task

…more combined with mergeable clause!!!



Controlling task scheduling (2)

The mergeable clause of a task construct

- allows to optimize task creation/execution (combined with the if clause)
- under certain circustances it may avoid the whole task overhead

```
#pragma omp task mergeable [if(expression)]
{structured-block}
```

if-clause evaluates to false → task is executed immediately

- But with its own data environment and ICVs

Combined with the semantic of the mergeable clause

- “a task for which the data environment (inclusive of ICVs) may be the same as that of its generating task region”
- so the user agrees (if posible) on relaxing the previous restriction

Undeferred and mergeable task may execute as a function call

Controlling task scheduling (3)

The final clause of a task construct

- allows to omit future task creation → reduces parallelism & overhead

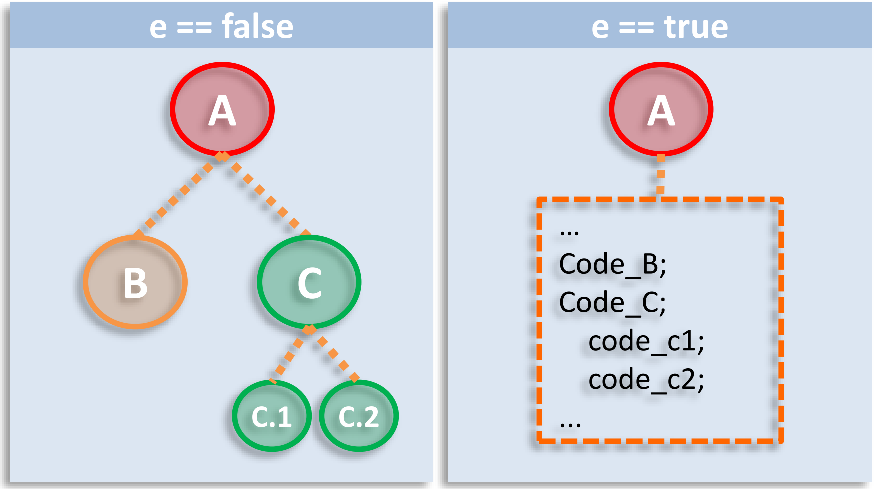
```
#pragma omp task final(expresion)
{structured-block}
```

If the expression of the "final" clause evaluates to true

- the new task is created and executed normally
- in the context of this task no new tasks will be created

```
#pragma omp parallel
#pragma omp single
{
  #pragma omp task final(e)
  {
    #pragma omp task
    { code_B; }
    #pragma omp task
    { code_C; }
  #pragma omp taskwait
  }
}
```

Children tasks may have additional task constructs



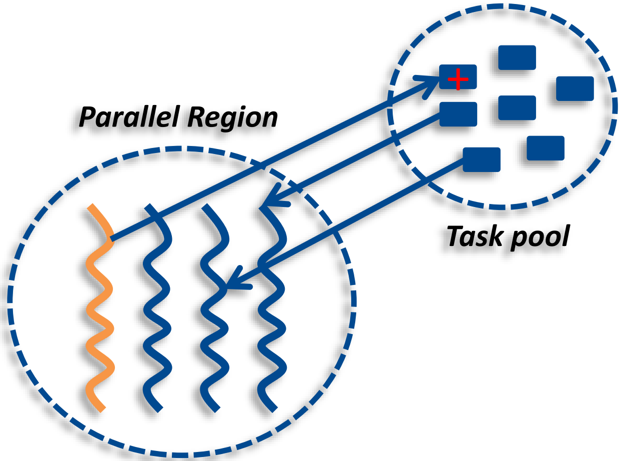
Programmer's hints for task scheduler

Programmers may specify a priority value when creating a task

```
#pragma omp task priority(pvalue)
{structured-block: loop}
```

- pvalue: the higher → the best (will be scheduled earlier)
- all ready tasks are inserted in an ordered ready queue
- once a thread becomes idle, gets one of the highest priority tasks

```
#pragma omp parallel
#pragma omp single
{
  for ( i = 0; i < SIZE; i++) {
    #pragma omp task priority(1)
    { code_A; }
  }
  #pragma omp task priority(100)
  { code_C; }
  ...
}
```



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Task Synchronization

Synchronizing the execution of threads / tasks

Threads need “some” order in the sequence of their actions

- execute in a logical order certain regions
- mutual exclusion in the execution of a given region
- wait in a location until all other threads have reach the same location
- wait until a given condition is accomplished

OpenMP provides different synchronization mechanisms

- masked / master construct, selecting thread within a parallel region
- critical construct, mutual exclusion when executing a region
- **barrier** directive [and implicits], all threads reaching the “barrier” before continuing
- atomic construct, load/update with hardware support
- flush directive [and implicits], make visible changes in the relaxed consistency model
- ordered clause/construct, forces a logical order among loop iterations
- **taskwait** directive, waiting for tasks (shallow)
- **taskgroup** construct, waiting for tasks (deep)
- **depend** clause, establish an order among tasks: pre-decessor, successor

The barrier directive

Threads cannot proceed after a barrier point until

- all threads reach the barrier
- and all previously generated work is completed

```
#pragma omp barrier
```

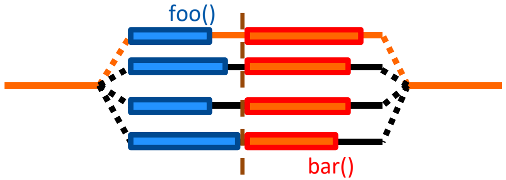
- some constructs have an implicit barrier at the end (e.g., the parallel construct)

Synchronizing threads between two phases in a parallel region

```
#pragma omp parallel
{
  foo ();
  #pragma omp barrier
  bar ();
}
```

Forces all foo()'s too happen before all bar()'s

Implicit barrier



The barrier directive

Threads cannot proceed after a barrier point until

- all threads reach the barrier
- and all previously generated work is completed

```
#pragma omp barrier
```

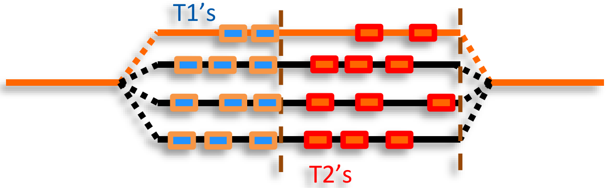
- some constructs have an implicit barrier at the end (e.g., the parallel construct)

Using barrier to force task completion

```
#pragma omp parallel
{
   #pragma omp single
   generate_taks_T1 ();

   #pragma omp barrier

   #pragma omp single
   generate_taks_T2 ();
}
```

Forces all tasks (T1) to be executed

Implicit barrier: also forces tasks to complete



The taskwait directive (shalow task synchronization)

– It is a stand-alone directive

```
#pragma omp taskwait
```

– wait on the completion of child tasks of the current task
– just direct children, not descendants
– includes an implicit task scheduling point

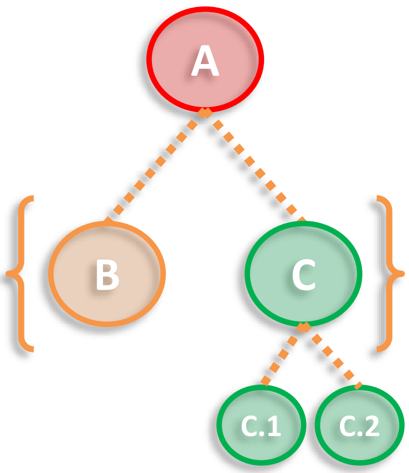
Using the taskwait directive

```
#pragma omp parallel
#pragma omp single
{
   #pragma omp task
   {
      #pragma omp task
      { … }
      #pragma omp task
      { … }
   #pragma omp taskwait
   }
}
```

Children tasks may create additional tasks

Wait only for direct descendant tasks

wait for…



Waiting for all descendant tasks

The taskgroup construct (deep task synchronization)

– always attached to a structured block

```
#pragma omp taskgroup
{structured-block}
```

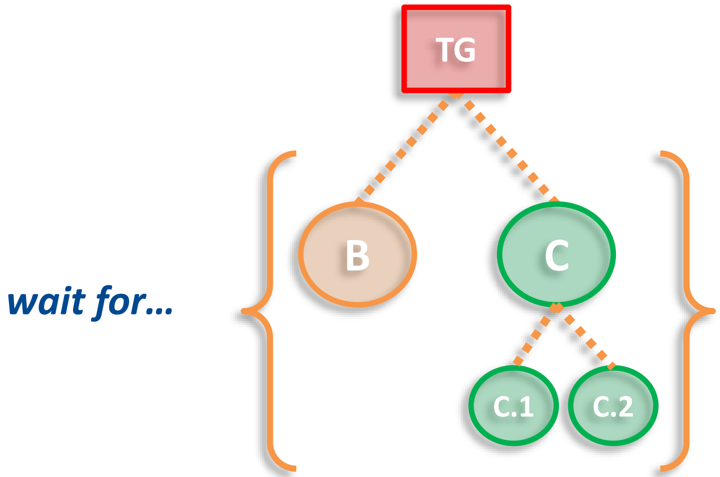
– wait on the completion of all descendant tasks of the current task
– includes an implicit task scheduling point at the end of the construct

Using the taskgroup construct

```
#pragma omp parallel
#pragma omp single
{
  #pragma omp taskgroup
  {
    #pragma omp task
    { … }
    #pragma omp task
    { … }
  }
}
```

Children tasks may create additional tasks

Wait for all descendant tasks



Using task dependences

The depend clause of the task construct

```
#pragma omp task depend(dependence-type: list)
{structured-block}
```

- used to compute dependences, but actually it is not a dependence
- specify the data directionality of a list of variables

Where dependence-type can be:

- in: the task only reads from the data specified
- out: the task only writes to the data specified
- inout: the task reads from and writes to the data

And where list items are

- variables, a named data storage block (memory address)
- array sections, a designated subset of the elements of an array
  - A[lower:length]

Computing task dependences (1)

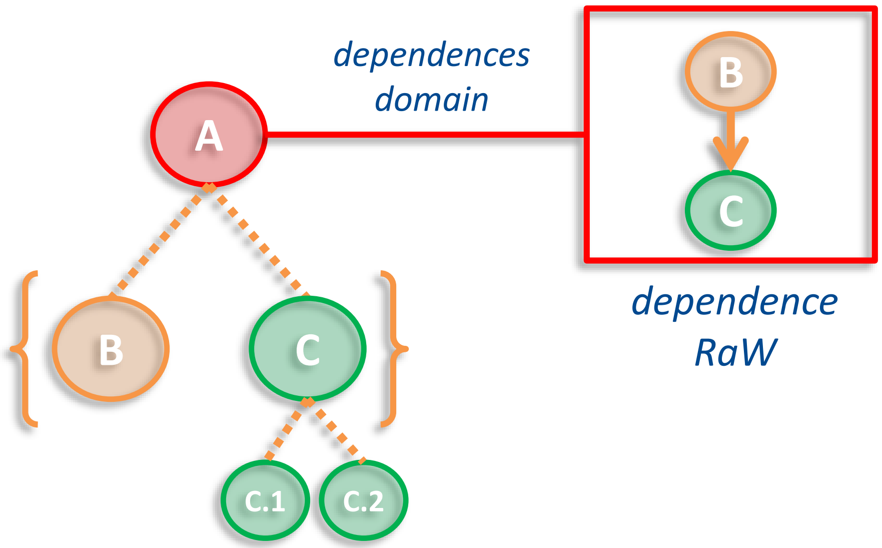
If a task does “in” on a given data variable

– the task will depend on all previously generated sibling tasks that reference at least one of the list items in an out or inout dependence list

If a task does “out” or “inout” on a given data variable

– on both out and inout dependence types, the task will depend on all previously generated sibling tasks that reference at least one of list items in an in, out or inout dependence list

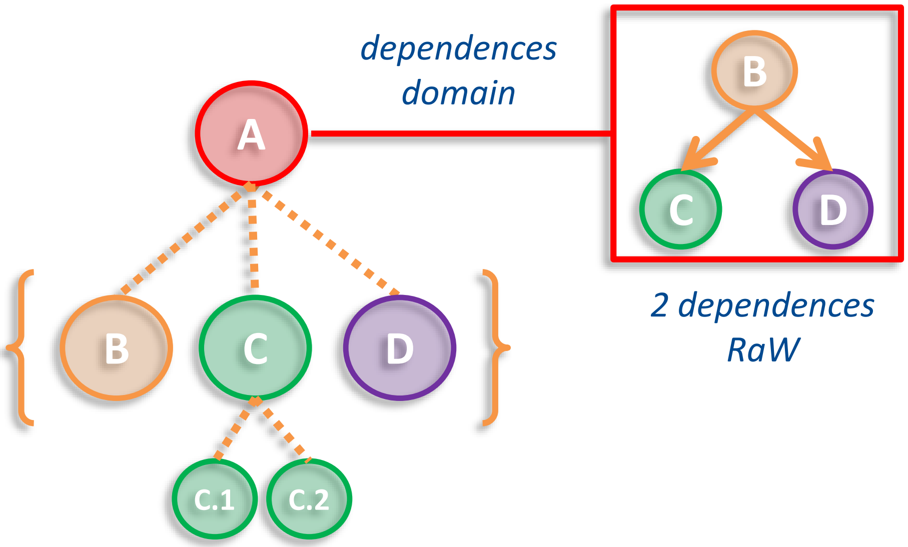
```
#pragma omp parallel
#pragma omp single
{
  #pragma omp task
  {
    #pragma omp task depend(out:a)
    { … }
    #pragma omp task depend(in:a)
    { … }
  #pragma omp taskwait
  }
}
```



Computing task dependences (2)

Computing dependences between one writer and n-readers

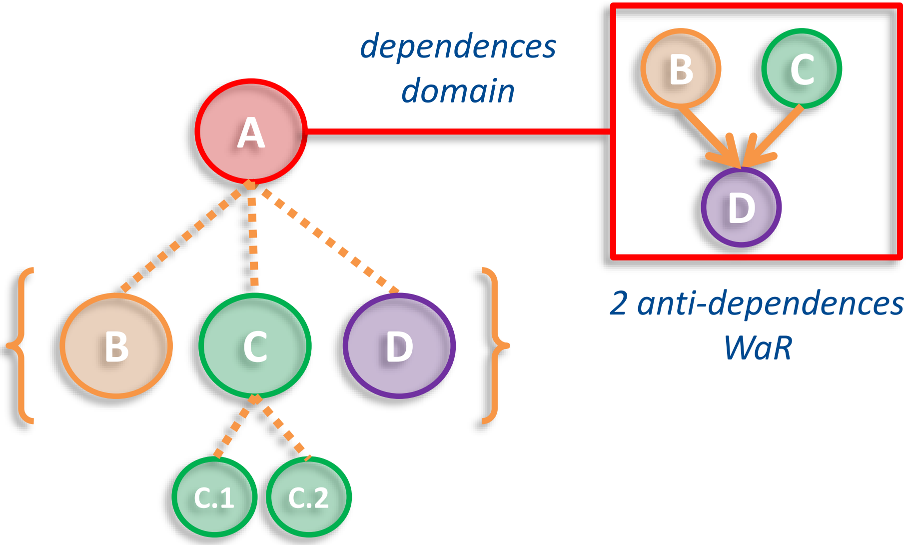
```
#pragma omp parallel
#pragma omp single
{
  #pragma omp task
  {
    #pragma omp task depend(out:a)
    { … }
    #pragma omp task depend(in:a)
    { … }
    #pragma omp task depend(in:a)
    { … }
  #pragma omp taskwait
  }
}
```



Computing task dependences (3)

Computing dependences between n-readers and one writer

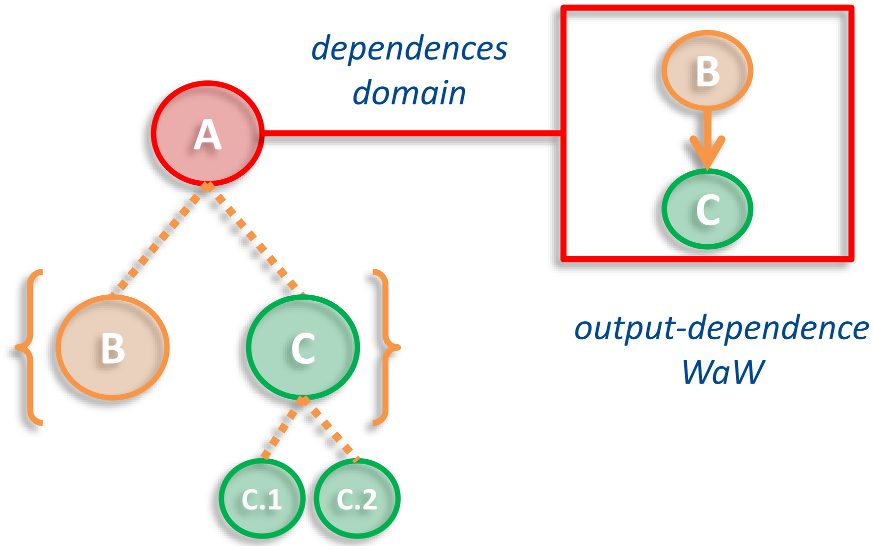
```
#pragma omp parallel
#pragma omp single
{
  #pragma omp task
  {
    #pragma omp task depend(in:a)
    { … }
    #pragma omp task depend(in:a)
    { … }
    #pragma omp task depend(out:a)
    { … }
  #pragma omp taskwait
  }
}
```



Computing task dependences (4)

Computing dependences between 2 writers

```
#pragma omp parallel
#pragma omp single
{
  #pragma omp task
  {
    #pragma omp task depend(out:a)
    { … }
    #pragma omp task depend(out:a)
    { … }
  #pragma omp taskwait
  }
}
```



Using task dependences (cont.)

The depend clause of the task construct

```
#pragma omp task depend(dependence-type: list)
{structured-block}
```

Restrictions on list items

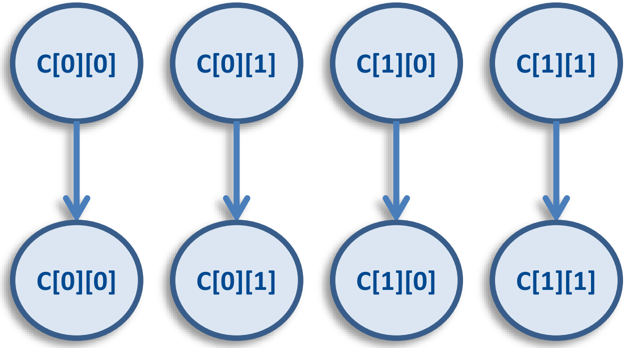
- list items used in depend clauses of the same task or sibling tasks must indicate identical storage or disjoint storage
- list items used in depend clauses cannot be zero-length array sections
- a variable that is part of another variable (such as a field of a structure) but is not an array element or an array section cannot appear in a depend clause

```
#define N 100

#pragma omp task depend(out: a[0:N])
{ … }

#pragma omp task depend(in: a[25:50])
{ … }
```

Example: matrix multiply (dependences)

```
void matmul_block ( int N, int BS, float *A, float *B, float *C) ;

// Assume BS divides N perfectly
void matmul ( int N, int BS, float A[N][N], float B[N][N], float C[N][N] )
{
   #pragma omp parallel
   #pragma omp single
   {
      int i, j, k;
      for ( i = 0; i < N; i+=BS) {
         for ( j = 0; j < N; j+=BS) {
            for ( k = 0; k < N; k+=BS) {
               #pragma omp task depend ( in:A[i:BS][k:BS],B[k:BS][j:BS] )\
                                depend ( inout:C[i:BS][j:BS] )
                  matmul_block (N, BS, &A[i][k], &B[k][j], &C[i][j] );
            }
         }
      }
   }
}
```

- avoid “blocks” to be written before read
- input deps useless in this particular example (*still recommended*)
- example on a matrix of 2x2 blocks:



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Taskloop construct

Highly Efficient Accel. and Reconfigurable Tech. (HEART) - 2024

Porto, June 21st, 2024

Task loop: motivation

Loop (worksharing) construct restrictions

- all threads (in the current team) must reach the worksharing construct
- taskloop constructs comes to break this specific restriction (using tasks)

So if we are executing a single or a section…

```
#include "synthetic.h“

void main (void)
{
  #pragma omp parallel
  #pragma omp sections
  {
    #pragma omp section
    synthetic_phase1();
    #pragma omp section
    synthetic_phase2();
    #pragma omp section
    synthetic_phase3();
  }
}
```

```
#include "synthetic.h“

void synthetic_phase2()
{
  #pragma omp for
  for ( i = 0; i < N ; i ++ ) { … }
}
```

✗

```
#include "synthetic.h“

void synthetic_phase2()
{
  #pragma omp taskloop
  for ( i = 0; i < N ; i ++ ) { … }
}
```

✓

The taskloop construct

Deferring several units of work (exec. for any team member)

- always attached to a “for” loop (“do” in Fortran)

```
#pragma omp taskloop [clause[[,] clause]...]
{structured-block: loop}
```

Where clause:

- shared(list), private(list), firstprivate(list), lastprivate(list) and default(dtype)
- if(scalar-expr) → already explained (applies to each created task)
- final(scalar-expr) → already explained (applies to each created task)
- priority(priority-value) → already explained (applies to each created task)
- untied → already explained (applies to each created task)
- mergeable → already explained (applies to each created task)
- grainsize(grain-size) and num_tasks(num-tasks)
- collapse(n)
- nogroup

Using grainsize in taskloop construct

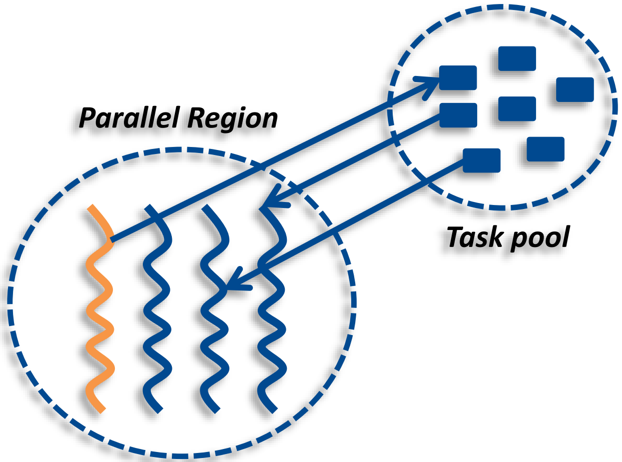
The grainsize clause of the taskloop construct

```
#pragma omp taskloop grainsize(<grain-size>)
{structured-block: loop}
```

- allow to specify the grain size of the generated chunks (tasks)
  - greater or equal than min(grain-size, iters)
  - less than two times grain-size (2 x grain-size)

- cannot be combined with num_tasks clause

```
#include "synthetic.h“

void synthetic_phase2() {
  #pragma omp taskloop grainsize(10)
  for ( i = 0; i < N ; i ++ ) { ... }
}
```



Philosophy: amount of work that is worthy to execute as a task

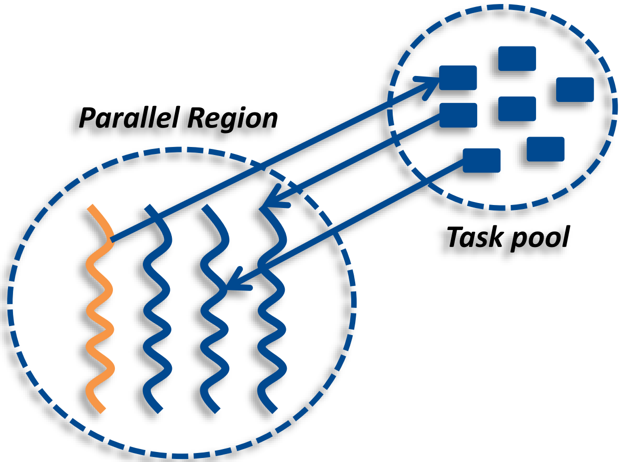
Using num_tasks in taskloop construct

The num_tasks clause of the taskloop construct

```
#pragma omp taskloop num_tasks(<num-tasks>)
{structured-block: loop}
```

- allow to specify the number of chunks (tasks)
  - greater or equal than min(num-tasks, iters)
  - each task should have as minimum one iteration
- cannot be combined with the grainsize clause

```
#include "synthetic.h"

void synthetic_phase2() {
  #pragma omp taskloop num_tasks(10)
  for ( i = 0; i < N ; i ++ ) { ... }
}
```



Philosophy: amount of parallelism we want to create

The collapse clause

Allows to distribute work from a set of *n*-nested loops

- loops must be perfectly nested (no instruction in between)
- the nest must traverse a rectangular iteration space (triangular also allowed)
- combines both iteration spaces to create a single one

Using the collapse clause over two loops

```
#define N ??
#define M ???

void main (void) {
  int i, j;
  #pragma omp parallel
  #pragma omp single
  {
    #pragma omp taskloop collapse(2) num_tasks(128)
    for ( i = 0; i < N; i ++ )
      for ( j = 0; j < M; j ++ )
        foo ( i , j ) ;
  }
}
```

- useful when first loop (or both) have only a few iterations (e.g., N = 64)
- increase the amount of created parallelism

→

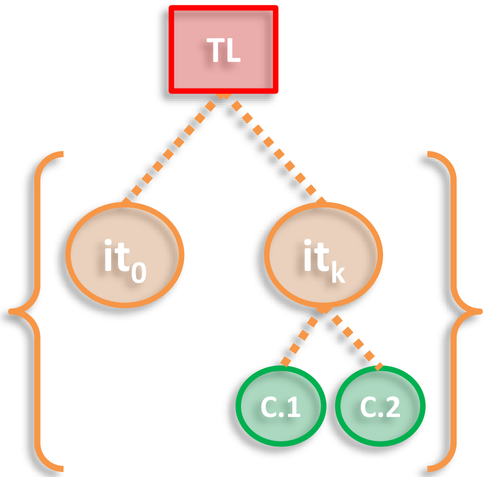
```
#pragma omp taskloop num_tasks(128)
for ( idx = 0; idx < (N * M); idx ++ ) {
    foo ( fi(idx) , fj(idx) ) ;
}
```

Taskgroup associated with a taskloop

```
#include "synthetic.h"
void synthetic_phase2()
{
  #pragma omp taskgroup
  {
    #pragma omp taskloop
    for ( i = 0; i < N ; i ++ ) { … }
  }
  foo();
  bar();
}
```

```
#include "synthetic.h"
void synthetic_phase2()
{

  {
    #pragma omp taskloop nogroup
    for ( i = 0; i < N ; i ++ ) { … }
  }
  foo();
  bar();
}
```

wait for…



The nogroup clause of the taskloop construct

```
#pragma omp taskloop nogroup
{structured-block: loop}
```

– allow to continue the execution of the encountering task without waiting for all created tasks

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Thank you!

For further information please visit/contact

http://www.linkedin.com/in/xteruel

xavier.teruel@bsc.es

Porto, June 21st, 2024