Introduction to OpenMP
Extensions in OpenMP-4.0 and 4.5 (and 5.0)

Rolf Rabenseifner
rabenseifner@hlrs.de
www.hlrs.de/people/rabenseifner/

University of Stuttgart
High-Performance Computing-Center Stuttgart (HLRS)
www.hlrs.de

 ● REC → online

Major Extensions in OpenMP 4.0 (Released July 2013)

OpenMP 4.0

Version 3.1 to 4.0 Differences (page numbers in OpenMP 4.0) → p. 303

- Initial support of Fortran 2003 (extensions to Fortran 95) → p. 22
- New Section 2.4 on array sections (in Fortran and C/C++) → p. 42
- Thread affinity and OpenMP places: → p. 49, 44, 241
  - **`proc_bind`** & **`OMP_PLACES, OMP_PROC_BIND`**
- **`simd`** construct to vectorize serial and parallelized loops → p. 68
- Support for accelerators through device constructs → p. 77
- Tasking extensions, e.g., → p. 116, 126
  - **`depend`** clause, **`taskgroup`** construct,
- User-defined reductions → `declare reduction` directive → p. 180
- Enhancements to atomic: → p. 127
  - New **`seq_cst`** clause and atomic swap with **`capture`** clause
- **`cancel`** and **`cancellation point`** construct → p. 140, 143, 199, 246
- **`OMP_DISPLAY_ENV`** to display all settings → p. 247

 ● REC → online

OMP_PLACES and Thread Affinity (see OpenMP-4.0 page 7 lines 29-32, p. 241-243)

A *place* consists of one or more *processors*.
Pinning on the level of *places*.
Free migration of the threads on a place between the *processors* of that place.

processor is the smallest unit to run a thread or task

- **`export OMP_PLACES=threads`** *abstract_name*
  → Each place corresponds to the single *processor* of a single hardware thread (hyper-thread)
- **`export OMP_PLACES=cores`**
  → Each place corresponds to the processors (one or more hardware threads) of a single core
- **`export OMP_PLACES=sockets`**
  → Each place corresponds to the processors of a single socket (consisting of all hardware threads of one or more cores)
- **`export OMP_PLACES=`***abstact_name(num_places)*
  → In general, the number of places may be explicitly defined

<lower-bound>:<number of entries>[:<stride>]

- Or with explicit numbering, e.g. 8 places, each consisting of 4 processors:
  - `export OMP_PLACES="{0,1,2,3},{4,5,6,7},{8,9,10,11}, … {28,29,30,31}"`
  - `export OMP_PLACES="{0:4},{4:4},{8:4}, … {28:4}"`
  - `export OMP_PLACES="{0:4}:8:4"`

CAUTION:
The numbers highly depend on hardware and operating system, e.g.,
{0,1} = hyper-threads of 1st core of 1st socket, or
{0,1} = 1st hyper-thread of 1st core of 1st and 2nd socket, or …

OpenMP places and proc_bind (see OpenMP-4.0 pages 49f, 239, 241-243)

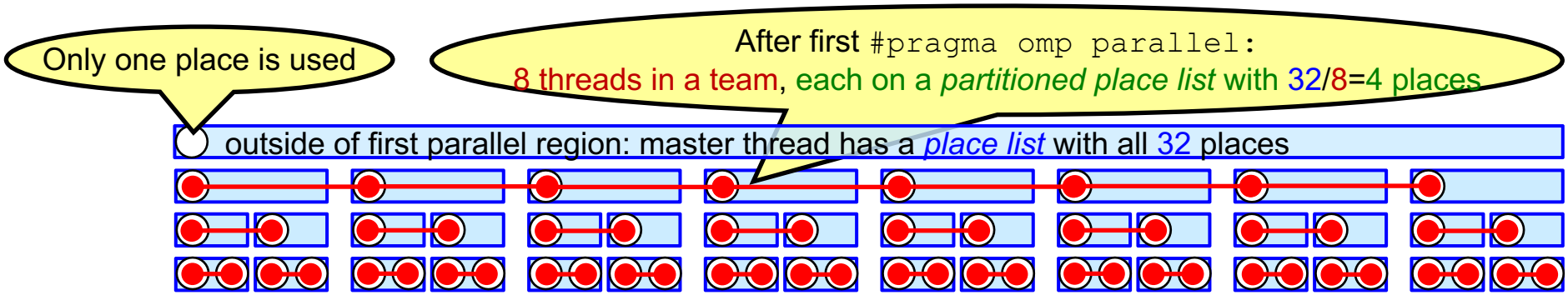
```
export OMP_PLACES="{0},{1},{2}, … {29},{30},{31}"
```
or

```
export OMP_PLACES=threads
```
(example with P=32 places)

- ```
  export OMP_NUM_THREADS="8,2,2"
  export OMP_PROC_BIND="spread,spread,close"
  ```
- Master thread encounters nested parallel regions:

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

  → uses: num_threads(8) proc_bind(spread)
  → uses: num_threads(2) proc_bind(spread)
  → uses: num_threads(2) proc_bind(close)



spread: Sparse distribution of the 8 threads among the 32 places; partitioned place lists.

close: New threads as close as possible to the parent's place; same place lists.

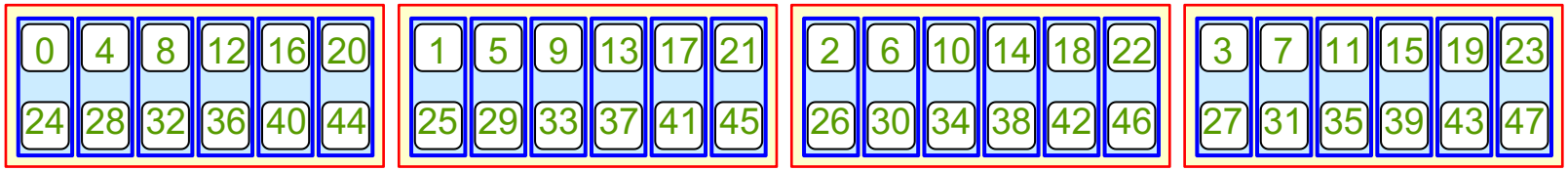
master: All new threads at the same place as the parent.

 ● REC → online
  ```

Goals behind OMP_PLACES and proc_bind

Example: 4 sockets x 6 cores x 2 hyper-threads = 48 *processors*

Vendor's numbering: round robin over the sockets, over cores, and hyperthreads



`export OMP_PLACES=threads` (="{0},{24},{4},{28},{8},{32},{12},{36},{16},{40},{20},{44},{1},{25}, … , {23},{47}")
→ OpenMP threads/tasks are **pinned** to hardware hyper-threads

`export OMP_PLACES=cores` (="{0,24}, {4,28}, {8,32}, {12,36}, {16,40}, {20,44}, {1,25}, … , {23,47}")
→ OpenMP threads/tasks are **pinned** to hardware cores
and can migrate between hyper-threads of the core

`export OMP_PLACES=sockets` (="**{**0, 24, 4, 28, 8, 32, 12, 36, 16, 40, 20, 44**}**, **{**1,25,…**}**, **{**…**}** , **{**…,23,47**}**")
→ OpenMP threads/tasks are **pinned** to hardware sockets
and can migrate between cores & hyper-threads of the socket

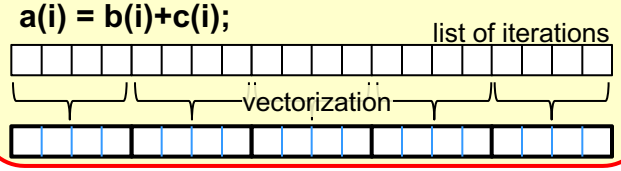
Examples should be **independent** of vendor's numbering & chosen pinning!

- Without nested parallel regions:
  #pragma omp parallel num_threads(4*6) proc_bind(spread) → one thread per core
- With nested regions:
  #pragma omp parallel num_threads(4) proc_bind(spread) → one thread per socket
  #pragma omp parallel num_threads(6) proc_bind(spread) → one thread per core
  #pragma omp parallel num_threads(2) proc_bind(close) → one thread per hyper-thread

 ● REC → online

Vectorization = SIMD Constructs

```
#pragma omp simd
for (i=0; i<n, i++)
  a(i) = b(i)+c(i);
```



SIMD Construct (OpenMP 4.0, page 68)

C/C++

- `#pragma omp simd` *[clause [[,] clause] ...]*
  for-loops

Loop iterations must be independent, i.e., they can be executed in parallel

Fortran

- `!$omp simd` *[clause [[,] clause] ...]*
  do-loops
  [`!$omp end simd`*]*

OpenMP 5.0: Standard is **changed**, i.e., then `safelen(5)` will be best and correct.
Expectation: Compilers already implemented OpenMP-5.0 since … (nobody knows :-)

Clauses:

- `safelen`(*length*)
- `linear`(*list[: linear-step]*)
- `private`(*list*)
- `lastprivate`(*list*)
- `reduction`(*reduction-identifier : list*)
- `collapse`(*n*)
- `aligned`(*list[: alignment]*)

See #pragma omp for/do

```
!$omp simd safelen(4)
DO i=100, 1000, 20
  → i=100, 120, 140, 160, 180, 200, 220, …  real iteration numbers
  → L= 0  1  2  3  4  5  6 …  Logical iteration numbers
      Always maximal 5 elements can be together in a vector
  a(i) = a(i-100)*b(i)
  → Parallelization of iterations with ΔL = 5,10,… → race-conditions
END DO
```

4 = 5 – 1 in OpenMP 4.0 / 4.5

k=*expression*; **kstep=***expression*

```
!$omp simd linear(k:kstep)
DO i=1,n
  a(i) = b(i)+c(k)
  k=k+kstep
END DO
```

The integer variable's value is in linear relationship with the iteration index. k gets private. Default *linear-step* = 1.
See OpenMP-4.0, page 172.

Specifies that the list items have a given alignment.
Important after aligned allocation, e.g., with
- malloc_align(64)
- __attribute__((aligned(64)))

Default is alignment for the architecture.

Parallelization & SIMD

Loop SIMD construct:

C/C++

- `#pragma omp for simd` *[clause [[,] clause] ...]*
  for-loops

Fortran

- `!$omp do simd` *[clause [[,] clause] ...]*
  do-loops
  [`!$omp end do simd` *[*`nowait`*]]*

- Cannot be specified separately.
- Worksharing on parallel region.
- Resulting chunks of iterations will then be converted to a SIMD loop.
- Clauses apply to omp & for/do

Parallel loop SIMD construct:

C/C++

- `#pragma omp parallel for simd` *[clause [[,] clause] ...]*
  for-loops

Fortran

- `!$omp parallel do simd` *[clause [[,] clause] ...]*
  do-loops
  `!$omp end parallel do simd`

- Purely a convenience that combines `omp parallel` with `omp for/do simd`
- Clauses first apply to `omp for/do simd` and remaining clauses then to `omp parallel`

 ● REC → online

Vectorized subroutines and functions

```
#pragma omp declare simd notinbranch
float sqrdist(float x1, float y1, float x2, float y2) {
   return (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2);
}

void example() {
  #pragma omp parallel for simd
    for (int i=0; i<N; i++)
      d[i] = sqrdist(x1[i],y1[i],x2[i],y2[i]);
}
```

Generates an **additional vector version** of this routine.

`notinbranch`: function **never** called from inside a conditional statement of a SIMD loop

Uses the vector version of this routine.

Other available clauses:

- `inbranch` → see next slide
- `simdlen(`*length*`)`
- `aligned(`*argument-list[:alignment]*`)`
- `uniform(`*argument-list*`)`
- `linear (`*argument-list[:constant-linear-step]*`)`

Determines the vector length of the generated vector routine. Length must be a constant expression. Several `#pragma omp declare simd` with different `simdlen(`*length*`)` values or sets of clauses are allowed.

Invariant value for all concurrent invocations of the function in the execution of a single SIMD loop

 ● REC → online

Vectorized subroutines and functions (continued)

```
#pragma omp declare simd inbranch
float sqrdist(float x1, float y1, float x2, float y2) {
   return (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2);
}
```

- `inbranch`:
  - function **always** called from inside a conditional statement of a SIMD loop
  - ➢ generates **masked vector** version.
- If both `inbranch` and `notinbranch` versions are need, then two `#pragma omp declare simd` lines with both clauses are recommended
  - ➢ will generate both normal and masked vector version.

```
void example() {
  #pragma omp parallel for simd
    for (int i=0; i<N; i++) {
      if (x2 > x1) d[i] = sqrdist(x1[i],y1[i],x2[i],y2[i]);
      else         e[i] = sqrdist(x1[i],y1[i],x2[i],y2[i]);
}
```

Uses the **masked vector** version of this routine.

Caution:
- Automatic optimization (e.g., with function inlining) ←→ OpenMP SIMD
- No guarantees about what is better ☹
- Use OpenMP SIMD construct if the compiler auto-vectorization is not sufficient

 ● REC → online

Array sections

- Defined in OpenMP-4.0, Section 2.4, page 42
- With restriction:
  - “can appear only in clauses where it is explicitly allowed” (page 42, line 3)
- Allowed in:
  - `map` clause on `omp target` constructs
  - `depend` clause on `omp task` constructs

 ● REC → online

GPU programming

- See OpenMP-4.0, Section 2.9, pages 77-94
- Will be included into our GPU programming courses.

Major Extensions in OpenMP 4.5 (Released Nov. 2015)

OpenMP 4.5

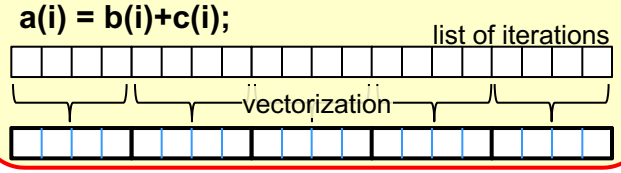
Version 4.0 to 4.5 Differences (page numbers in OpenMP 4.5) → p. 303

- New **`taskloop`** and **`taskloop simd`** ~~worksharing~~ constructs → p. 87-92
- Nearly complete support of Fortran 2003 → p. 22
- **`linear`** clause also for **`do`** and **`for`** loop worksharing → p. 56
- New **`simdlen`** clause for the **`simd`** construct → p. 72-75
- **`ordered(n)`** clause & **`ordered`** construct for nested loops, and dependencies can be explicitly specified → p. 56,166,169
- New **`priority`** clause for the **`task`** construct → p. 83,268,303
- Possibility of **`if`** clause for parts of a combined construct → p. 147
- New **`hint`** clause for the **`critical`** construct & new lock routines → p. 149, 273
- Additional **`ref`**, **`val`**, **`uval`** modifiers for the **`linear`** clause on **`declare simd`** construct → p. 207
- Use of some C++ reference types was allowed in some data sharing attribute clauses → p. 188
- Semantics for reductions on C/C++ array sections were added and restrictions on the use of arrays and pointers in reductions were removed → p. 207
- Enhanced support for accelerators

 ● REC → online

Vectorization = SIMD Constructs

```
#pragma omp simd
for (i=0; i<n, i++)
  a(i) = b(i)+c(i);
```



SIMD Construct (OpenMP 4.0, page 68)

C/C++

- `#pragma omp simd` *[clause [[,] clause] ...]*
  for-loops

Fortran

- `!$omp simd` *[clause [[,] clause]...]*
  do-loops
  [`!$omp end simd`*]*

Clauses:

- `safelen`(*lenA*)
- `simdlen`(*lenB*)
- `linear`(*list[:linear-step]*)
- `private`(*list*)
- `lastprivate`(*list*)
- `reduction`(*reduction-identifier*:*list*)
- `collapse`(*n*)
- `aligned`(*list[:alignment]*)

Since OpenMP-4.5 NEW also for omp for/do

Same with #pragma omp for/do

Discussion of safelen and simdlen is based on a communication with Michael Klemm, Intel.

OpenMP 5.0: Standard is **changed**, i.e., then `safelen(5)` will be best and correct.
Expectation: Compilers already implemented OpenMP-5.0 since … (nobody knows :-)

```
!$omp simd safelen(4)
DO i=100, 1000, 20
  → i=100, 120, 140, 160, 180, 200, 220, …  real iteration numbers
  → L= 0  1  2  3  4  5  6 … Logical iteration numbers
      Always maximal 5 elements can be together in a vector
  a(i) = a(i-100)*b(i)
  → Parallelization of iterations with ΔL = 5,10,… → race-conditions
END DO
```

4 = 5 – 1

NEW

The **simdlen** clause specifies the **preferred number of iterations** to be executed **concurrently**.

It is only a wish, no binding behavior! **safelen still needed!**

Restriction (OpenMP-4.5 page 75 lines 10-11):

OpenMP-4.5: ***lenB ≤ lenA*** **(this is stupid!)**
i.e., in the example: `simdlen(4), safelen(4)`

Correction in OpenMP 5.0: still *lenB ≤ lenA*
but now in the example: `simdlen(5), safelen(5)`

k=*expression*; **kstep=***expression*

```
!$omp simd linear(k:kstep)
DO i=1,n
  a(i) = b(i)+c(k)
  k=k+kstep
END DO
```

The integer variable's value is in linear relationship with the iteration index. k gets private. Default *linear-step* = 1.
See OpenMP-4.0, page 172.

taskloop *(a task generating construct)*

Idea

- Execute, e.g., 100,000 loop iterations as 100 tasks, each with *grain_size* 1,000 iterations

Advantages

- One or some threads can execute a less compute intensive application part as some tasks, while some other threads execute a loop as several tasks.
- No (inefficient) nested parallelism needed for this
- No load balancing problems between both numerical application parts

```
!$OMP PARALLEL num_treads(2)
 !$OMP SECTIONS
  !$OMP SECTION
   ! a less compute intensive application part
  !$OMP SECTION
   !$OMP PARALLEL num_treads(7)
    !$OMP DO
      do i=1,100000
        a(i) = b(i) +b(i-1)+b(i+1)+b(i-2)+…
      end do
    !$OMP END DO
   !$OMP PARALLEL
 !$OMP END SECTIONS
!$OMP END PARALLEL
```

Disadvantages

- `omp taskloop grain_size(1000)` has similar **disadvantages** as `omp do/for schedule(dynamic,1000)`

Before looking at the `taskloop` worksharing details

- Let's retake
  - task
  - single
  - sections
  - loop: do / for
- Several slides are skipped

 ● REC → online

OpenMP `task` Directive – Example:

Parallelized traversing of a tree

Retake from the OpenMP-3.0/3.1 course

OpenMP 3.0

C/C++

```
struct node {
   struct node *left;
   struct node *right;
};
extern void process(struct node *);
void traverse( struct node *p ) {
    if (p->left)
#pragma omp task // p is firstprivate by default
                traverse(p->left);
    if (p->right)
#pragma omp task // p is firstprivate by default
                traverse(p->right);
    process(p); // significant work with p
}
int main(int argc, char **argv)
{ struct node tree;
  ... // producing the tree
#pragma omp parallel
  {
#pragma omp single
     {
       traverse(&tree);//traversing the existing tree
     } // end of omp single
  } // end of omp parallel
}
```

- Starting the parallel team of threads
- Using only one thread for starting the traversal
- First execution with single thread (= 1st task)
- A new task is started (on a new thread)
- A recursive call to traverse() in this 2nd task
- 3rd task is started
- Work is done in 1st task
- Recursive calls starting 4th, 5th, … tasks

Trick: OpenMP can choose whether new tasks are <u>immediately</u> started <u>or deferred</u> until free thread is available.

Same example in Fortran: OpenMP 3.0, Exa. A.13.1f, page 178

 ● REC → online

Retake from the OpenMP-3.0/3.1 course

OpenMP `task` Directive – Syntax

OpenMP 3.0

- The **`task`** construct defines an explicit task.

Fortran

- Fortran:

```
!$OMP task [ clause [ [ , ] clause ] ... ]
    block
!$OMP end task
```

C/C++

- C/C++:

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

- Clauses:
  - `untied`
  - `default(shared | none | private | firstprivate)`
  - `private(`*list*`)`
  - `firstprivate(`*list*`)`
  - `shared(`*list*`)`
  - `if(`*scalar expression*`)`

 ● REC → online

Retake from the OpenMP-3.0/3.1 course

OpenMP `task` Directive – Principles

OpenMP 3.0

- When a thread encounters a **`task`** construct,
  a task is generated from the code for the associated structured block.
- The encountering thread
  - may immediately execute the task,
  - or may defer its execution.

  The number of tasks can be limited, e.g., to the number of threads.
- Completion of a task can be guaranteed using task synchronization constructs → **`taskwait`** construct.
- When `if(false)` clause exists, then execution is *"serial"*
- **Task scheduling points**:
  - In the generating task: Immediately following the generation of an explicit task.
  - In the generated task: After the last instruction of the task region.
  - If task is "**`untied`**": Everywhere inside of the task.
  - In implicit and explicit barriers.
  - In **`taskwait`**.

  At task scheduling points, tasks can be resumed or suspended.
  (Further constraints → OpenMP 3.0, Sect. 2.7.1, page 62)

 ● REC → online

Retake from the OpenMP-3.0/3.1 course

OpenMP `single` Directive – Syntax

- The block is executed by only one thread in the team (not necessarily the master thread)

Fortran

- Fortran:

```
!$OMP single [ clause [ [ , ] clause ] ... ]
    block
!$OMP end single [nowait]
```

C/C++

- C/C++:

```
#pragma omp single [ clause [ [ , ] clause ] ... ] new-line
    structured-block
```

- Implicit barrier at the end of **`single`** construct
  (unless a nowait clause is specified)
- To reduce the fork-join overhead, one can combine
  – several parallel parts (`for, do, workshare, sections`)
  – and sequential parts (`single`)
  in **one** parallel region (`parallel … end parallel`)

 ● REC → online

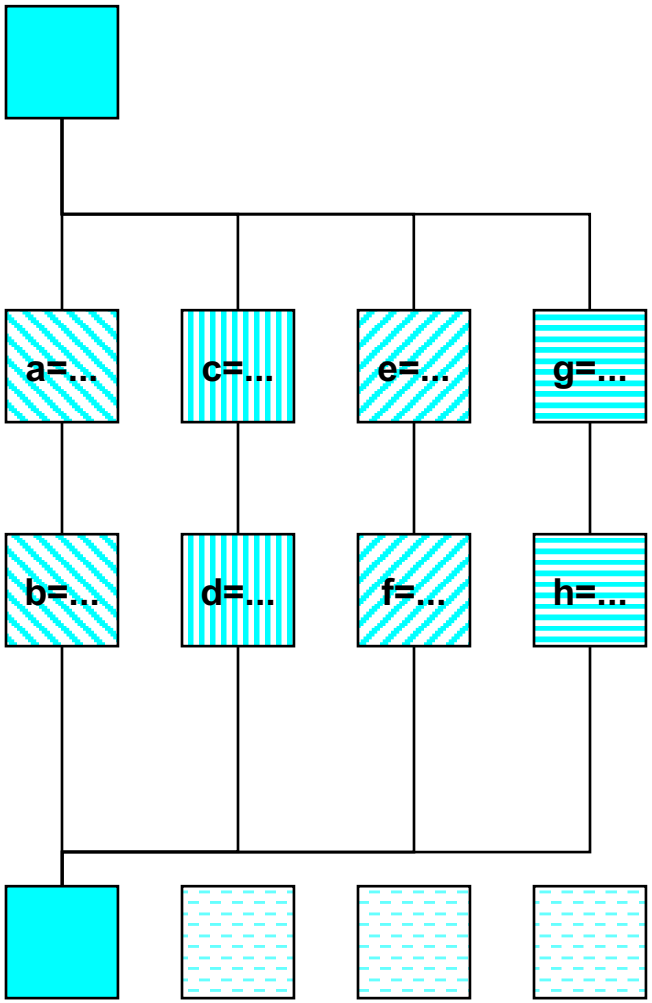
Retake from the OpenMP-3.0/3.1 course

C/C++

OpenMP `sections` Directives – C/C++

C / C++:

```
#pragma omp parallel
{
#pragma omp sections
  {{   a=...;
       b=...; }
#pragma omp section
    {  c=...;
       d=...; }
#pragma omp section
    {  e=...;
       f=...; }
#pragma omp section
    {  g=...;
       h=...; }
  } /*omp end sections*/
} /*omp end parallel*/
```



 ● REC → online

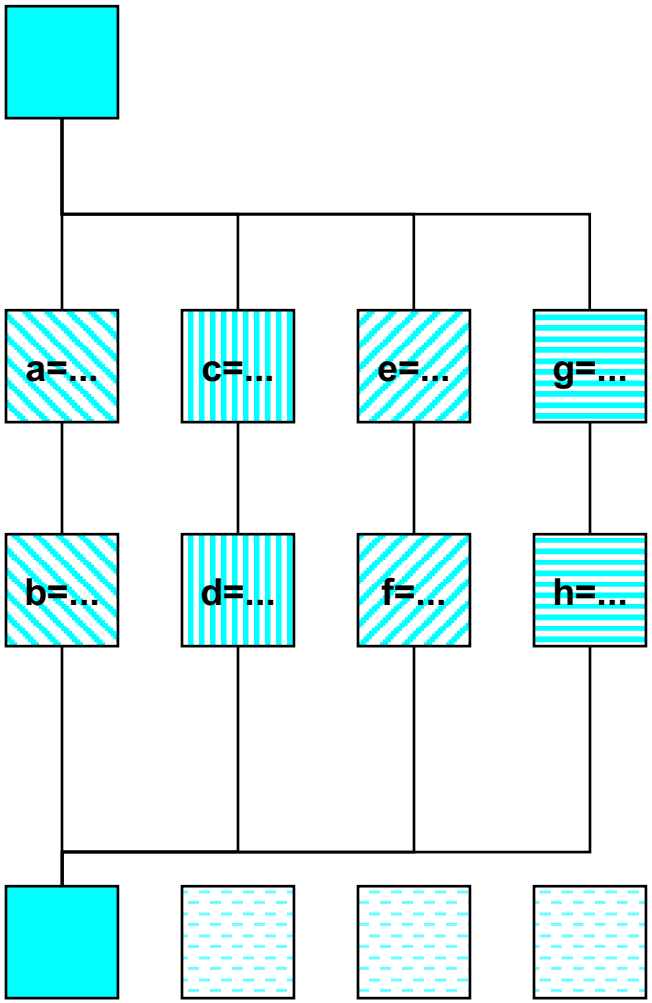
Retake from the OpenMP-3.0/3.1 course

Fortran

OpenMP `sections` Directives – Fortran

Fortran:

```
!$OMP PARALLEL
!$OMP SECTIONS
      a=...
      b=...
!$OMP SECTION
      c=...
      d=...
!$OMP SECTION
      e=...
      f=...
!$OMP SECTION
      g=...
      h=...
!$OMP END SECTIONS
!$OMP END PARALLEL
```



 ● REC → online

Retake from the OpenMP-3.0/3.1 course

C/C++

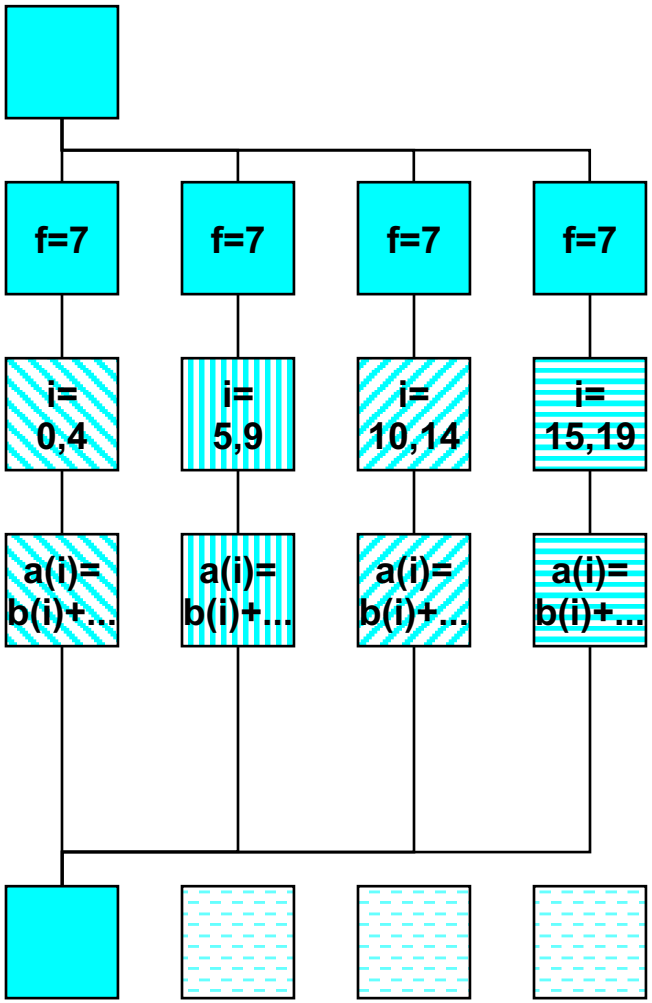
OpenMP `do/for` Directives – C/C++

C / C++:

```
#pragma omp parallel private(f)
{
      f=7;

#pragma omp for
      for (i=0; i<20; i++)
          a[i] = b[i] + f * (i+1);

} /* omp end parallel */
```



 ● REC → online

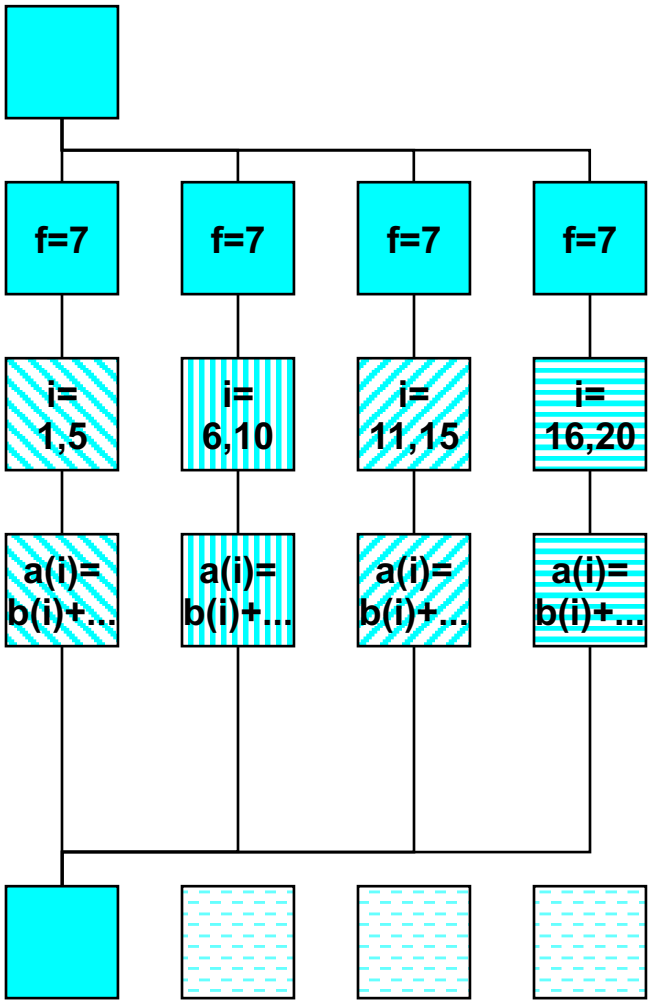
Retake from the OpenMP-3.0/3.1 course

Fortran

OpenMP do/for Directives – Fortran

Fortran:

```
!$OMP PARALLEL private(f)
      f=7
!$OMP DO
      do i=1,20
         a(i) = b(i) + f * i
      end do
!$OMP END DO
!$OMP END PARALLEL
```



 ● REC → online

OpenMP `taskloop` Directive – Syntax

- Immediately following loop executed in **several tasks.**
- **It is not a work-sharing among threads!**
- **→ Should be executed only by one thread!**

Fortran

- Fortran:
  `!$OMP taskloop` [*clause* [[,] *clause*] ...]
  do_loop
  [`!$OMP end taskloop`]

Loop iterations must be independent, i.e., they can be executed in parallel

- If used, then the `end taskloop` directive must appear immediately after the end of the loop

C/C++

- C/C++:
  `#pragma omp taskloop` [*clause* [[,] *clause*] ...] *new-line*
  for-loop
- The corresponding *for-loop* must have *canonical shape*
  → see slide on `#pragma omp for`

 ● REC → online

OpenMP `taskloop` Directive – Details

- taskloop clauses:
  - `if(`[`taskloop:`] *scalar-expr*`)` [a task clause]
  - `shared(`*list*`)` [a do/for clause] [a task clause]
  - `private(`*list*`), firstprivate(`*list*`)` [a do/for clause] [a task clause]
  - `lastprivate(`*list*`)` [a do/for clause]
  - `default(shared | none)` [a task clause]
  - `collapse(` *n* `)` [a do/for clause]
  - `grainsize(`*grain-size*`)`
  - `num_tasks(`*num-tasks*`)`

    Mutually exclusive
  - `untied, mergeable` [a task clause]
  - `final(` *scalar-expr* `), priority(` *priority-value* `)` [a task clause]

    final(true) → all children tasks from generated tasks will be sequentially included
  - `nogroup`
  - **`reduction(`*operator*`:`*list*`)`** [a do/for clause]

    Since OpenMP-5.0
- *do/ for clauses that* are **not** valid on a *taskloop directive*:
  - `schedule(` *type* [, *chunk*] `),`
  - `linear(`*list* [: *linaer-step*] `), ordered [(` *n* `)], nowait`

    taskloop schedule is similar to do/for schedule(dynamic, grain-size)

 ● REC → online

Retake from the OpenMP-3.0/3.1 course

First touch

First write of a byte in a memory page
→ memory page is located in the physical memory of the executing thread

Don't use **c**alloc() with OpenMP – no first touch
whole array is already mapped to physical memory

```
#define n 1000000
double *x; int i;
x = (double *) malloc(n*sizeof(double));
```

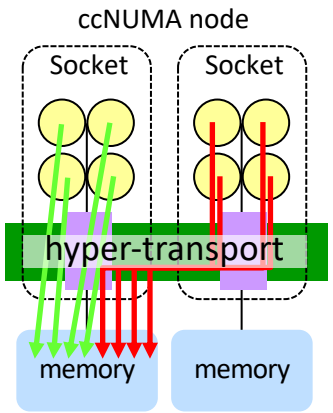
malloc() does not specify the physical location of the memory pages of the array!

```
// sequential initialization of the data
for (i=0 ; i<n; i++) x[i]=0;
```

First touch only by master thread
→ Whole array is in 1st CPU's memory

```
#pragma omp parallel
{  // parallelized numerical loop
   #pragma omp for schedule(static)
     for (i=0 ; i<n; i++) x[i]=huge_computation(i);
}
```

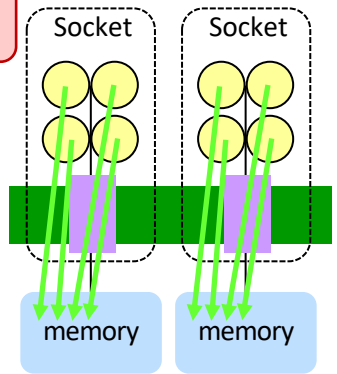
→ slower accesses by threads on 2nd CPU



impossible with any **dynamic schedule** or **taskloops** (i.e., only with static schedule)

```
#pragma omp parallel
{// parallel initialization of the data
   #pragma omp for schedule(static)
     for (i=0 ; i<n; i++) x[i]=0;
 // parallelized numerical loop
   #pragma omp for schedule(static)
     for (i=0 ; i<n; i++) x[i]=huge_computation(i);
}
```

Parallelized first touch **with same schedule** as in the numerical loop



→ Fast accesses by all threads because x[i] is in the own CPU's memory

 ● REC → online

C/C++ OpenMP `single` & `taskloop` Directives – C/C++

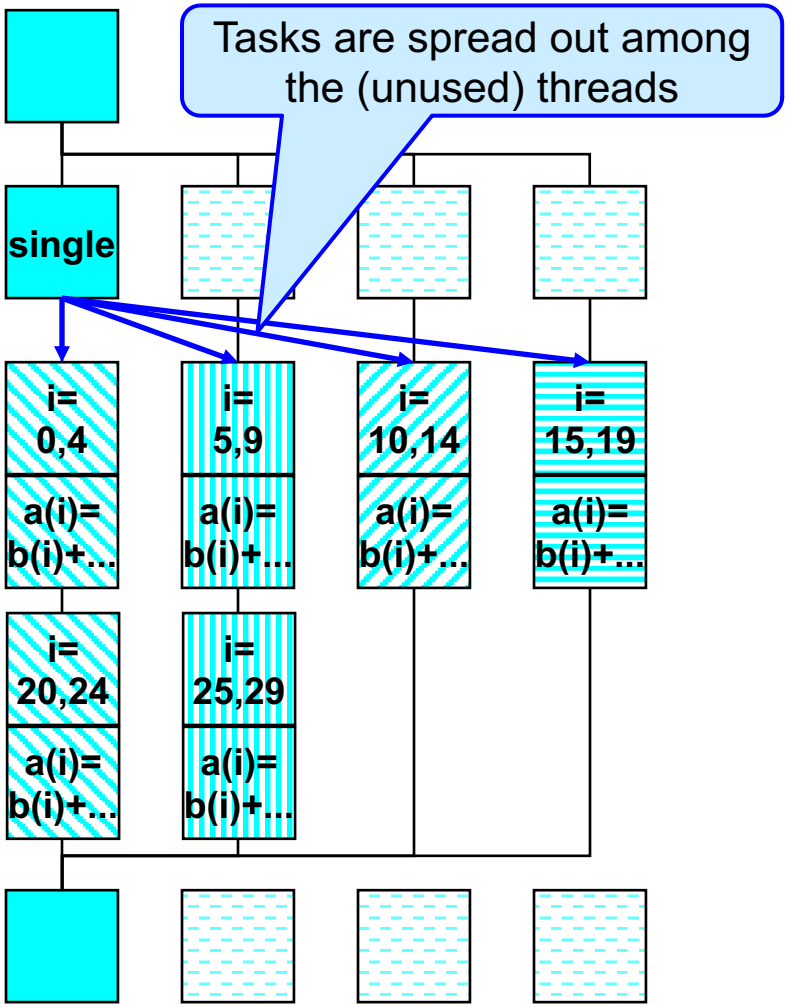
C / C++:

```
#pragma omp parallel
{
  #pragma omp single
  {

    #pragma omp taskloop
      for (i=0; i<30; i++)
          a[i] = b[i] + f * (i+1);

  }
  } /*omp end single*/
} /*omp end parallel*/
```

A lot more tasks than threads may be produced to achieve a good load balancing



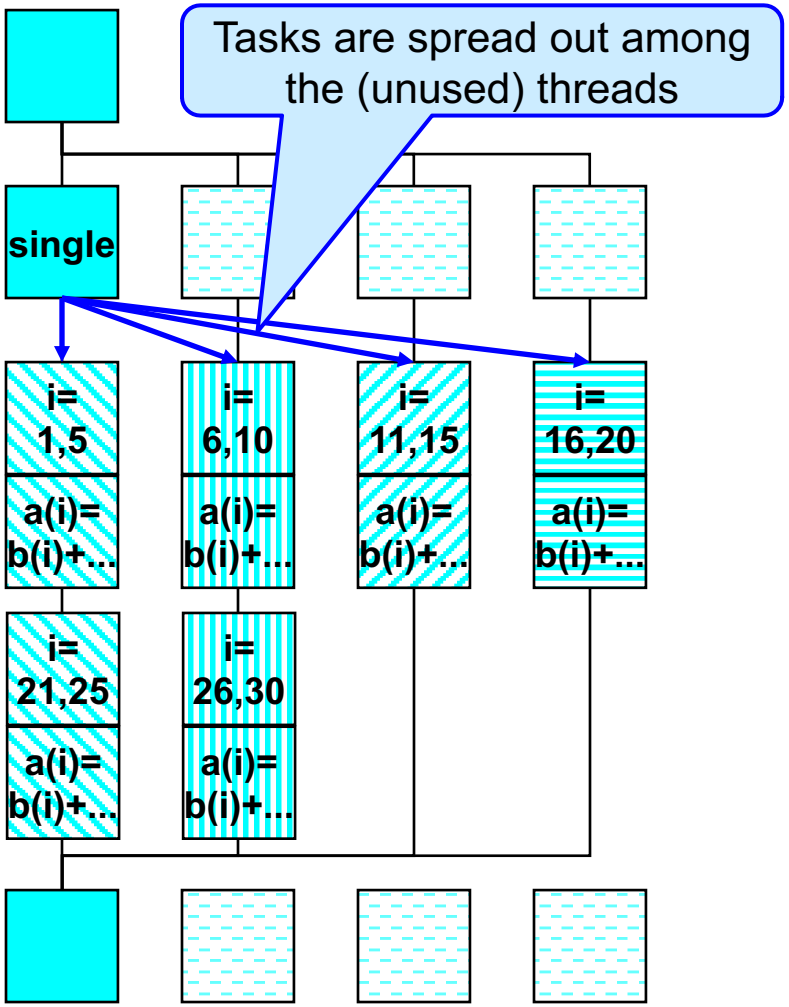
 ● REC → online

Fortran OpenMP `single` & `taskloop` Directives – Fortran

Fortran:

```
!$OMP PARALLEL
  !$OMP SINGLE
    !$OMP TASKLOOP
      do i=1,30
        a(i) = b(i) + f * i
      end do
    !$OMP END TASKLOOP
  !$OMP END SINGLE
!$OMP END PARALLEL
```

A lot more tasks than threads may be produced to achieve a good load balancing



 ● REC → online

OpenMP `sections` & `taskloop` Directives – C/C++
Use case: Hybrid MPI & OpenMP

C/C++

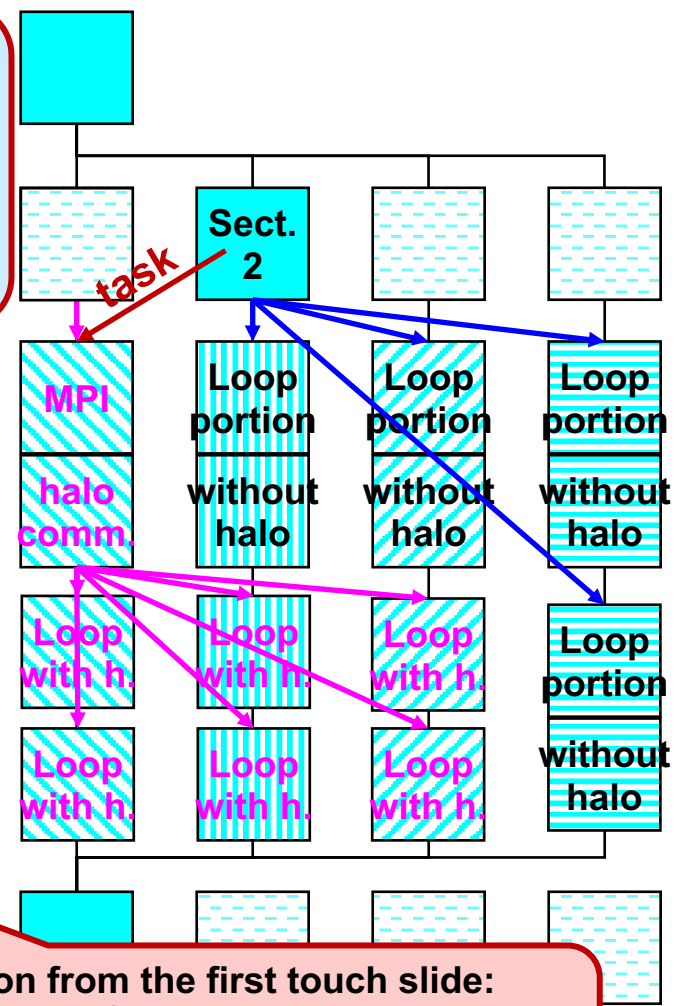
```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    {   // MPI halo communication:
        ....
        // numerical loop using halo data:
          #pragma omp taskloop
          for (i=0; i<100; i++)
             a[i] = b[i] +b[i-1]+b[i+1]+b[i-2]…;
    } /*omp end of "first" section*/
    #pragma omp section
    {   // numerical loop without using halo data:
          #pragma omp taskloop
          for (i=100; i<10000; i++)
             a[i] = b[i] +b[i-1]+b[i+1 ]+b[i-2]…;
        …
    } /*omp end of "second" section*/
  } /*omp end sections*/
} /*omp end parallel*/
```

single

task

Instead of sections, one can also use **single** and a **task** for the first section

Number of tasks may be influenced with grainsize or num_tasks clauses



Caution / implication from the first touch slide:
if a and b represent data of the MPI subdomain, then **each MPI process should be within a NUMA domain** (and not the whole ccNUMA node)

 ● REC → online

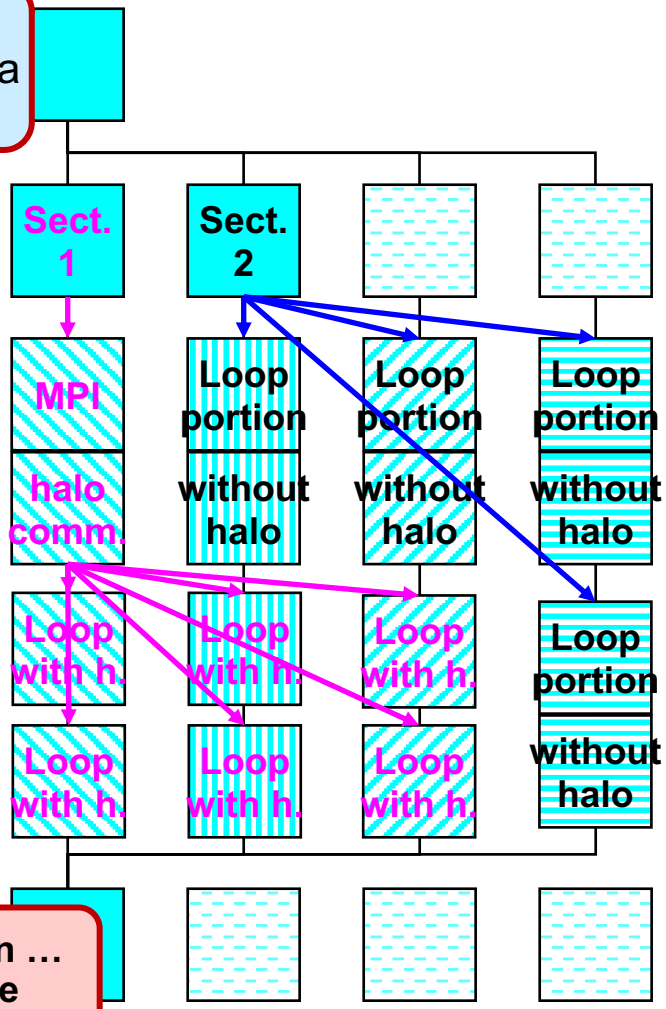
Fortran OpenMP `sections` & `taskloop` Directives – Fortran

```
!$OMP PARALLEL
 !$OMP SECTIONS
  !$OMP SECTION
   ! MPI halo communication:
     ....
   ! numerical loop using halo data:
   !$OMP TASKLOOP
     do i=1,100
      a(i) = b(i) +b(i-1)+b(i+1)+b(i-2)+…
    end do
   !$OMP END TASKLOOP
 !$OMP SECTION
   ! numerical loop without using halo data:
   !$OMP TASKLOOP
     do i=101,10000
      a(i) = b(i) +b(i-1)+b(i+1)+b(i-2)+…
    end do
   !$OMP END TASKLOOP
 !$OMP END SECTIONS
!$OMP END PARALLEL
```

Instead of sections, one can also use **single** and a **task** for the first section

Number of tasks may be influenced with grainsize or num_tasks clauses

Caution / implication …
→ see previous slide



 ● REC → online

Major Extensions in OpenMP 5.0 (Released Nov. 2018)

OpenMP 5.0

Version 4.5 to 5.0 Differences (page numbers in OpenMP 5.0) → p. 627-631

- `reduction` clause for `taskloop` and `taskloop simd` → p. 629:20-23
- `collapse(n)` also for **imperfectly** nested loops (e.g. triangles) → p. 628:27-28,34-37
- Default loop schedule **changed** from `monotonic` to `nonmonotonic`
  (if schedule is not static, and without ordered clause)
  → free sequence of the chunks within a thread (within a chunk, the sequence is still monotonic)
  → enables implementation of work stealing for dynamic/guided
  → **minimizes overhead** → see Open**MP**Con 2018, Cownie & Peyton → p. 628:31-33
- Scan option for reductions → p. 629:7-9
- Task reductions → Open**MP**Con 2018, Michael Klemm: OpenMP API 5.0 → p. 629:10-14
- A lot of other smaller enhancements → p. 627-631
- Enhanced support for accelerators

Further information:

- Open**MP**Con 2018 conference, slides: https://openmpcon.org/conf2018/program/
  - Michael Klemm: **OpenMP API 5.0 - Update on new Features**
  - Jim Cownie, Jonathan Peyton: **Small, Easy to Use, OpenMP* Features You May Have Missed**
  - Michael Klemm, Xavier Martorell and Xavier Teruel: **Advanced OpenMP Tutorial**

 ● REC → online

Conclusions

- OpenMP-4.0 includes important new features
  - Based on experience with other products, e.g.,
    - Thread-affinity packages
    - MPI user-defined reductions
    - OpenACC
    - Long history of directives to support vectorizing
  - Now, integral part of the OpenMP-4.0 standard !
- OpenMP-4.5, major new features are
  - taskloop
  - extended GPU support

Acknowledgements

- Christian Terboven, Michael Klemm, Bronis R. de Supinski
  - “Advanced OpenMP Tutorial” at ISC 2013, Leipzig
- Christoph Niethammer, José Gracia (HLRS)

 ● REC → online

Exercises

OpenMP Exercise: pi_taskloop program (1)

- Goal: usage of
  - `taskloop` constructs
- Working directory: `~/OpenMP/#NR/pi_taskloop/`
  `#NR` = number of your PC, e.g., 07

  Always "01" in online courses
- Serial programs:
  - Fortran 90: `pi_taskloop.f90` and `pi_taskloop2.f90`
  - C: `pi_taskloop.c` and `pi_taskloop2.c`

Fortran

C/C++

- The taskloop program is a skeleton for using single + taskloop constructs
- The taskloop2 program is prepared for sections + 2 x taskloop constructs
  - Only the outer loop can be parallelized with taskloop
  - The inner loop contains a reduction → one can not use taskloop
- The skeleton splits the loop into a loop nest
  - The outer loop is without an reduction operation
  - Reason: OpenMP 4.5 did not provide the reduction clause for taskloops (resovled in OpenMP 5.0)

 ● REC → online

OpenMP Exercise: pi_taskloop program (2)

- compile serial program `pi.[f|f90|c]` and run
- add parallel region single and taskloop directives
- compile as OpenMP program
- set environment variable `OMP_NUM_THREADS` to 1, 2, 4 and run
  - value of pi? (should be correct)
  - examine the OMP_GET_WTIME time (should be 2x and 4x faster)

- After *successful execution*, you may compare your result with the provided solution:
  - `../../solution/pi_taskloop/pi_taskloop_solution.c` or `../../solution/pi_taskloop/pi_taskloop_solution.f90`

OpenMP Advanced Exercise: pi_taskloop2 program (3)

- compile serial program `pi.[f|f90|c]` and run
- add parallel region sections and two times the taskloop directives
- compile as OpenMP program
- set environment variable `OMP_NUM_THREADS` to 1, 2, 4 and run
  - value of pi? (should be correct)
  - examine the OMP_GET_WTIME time (should be 2x and 4x faster)

- After *successful execution*, you may compare your result with the provided solution:
  - `../../solution/pi_taskloop/pi_taskloop2_solution.c` or `../../solution/pi_taskloop/pi_taskloop2_solution.f90`

Appendix

pi_taskloop_solution.c – solution with taskloop

```
int main(int argc, char** argv)
{
  int i;
  double w,x,sum,pi;
  int i_outer;
  double sum_outer[100][64]; // A cache line should be not larger than the 64 ints.
                             // Each task j should use sum_outer[j][0], i.e., each of
                             // these summing-variables is in a different cache-line
…
/* calculate pi = integral [0..1] 4/(1+x**2) dx */
  w=1.0/n;
  sum=0.0;
  for (i_outer=0; i_outer<100; i_outer++) sum_outer[i_outer][0]=0.0;

#pragma omp parallel
{
# pragma omp single
#  pragma omp taskloop private(x,i)
  for (i_outer=0; i_outer<100; i_outer++)
     for (i=i_outer+1;i<=n;i+=100)
     {
       x=w*((double)i-0.5);
       sum_outer[i_outer][0] += f(x);
     }
} /*end omp parallel*/

  for (i_outer=0; i_outer<100; i_outer++) sum += sum_outer[i_outer][0];
  pi=w*sum;
  printf( "computed pi = %24.16g\n", pi );
  return 0;
}
```

Skeleton: Outer loop without reduction was added

To prevent cache-line false sharing

Solution: Using a taskloop within parallel / single

Skeleton: Summing up the outer loop outside of the numerical loop

Required for time measurements (hidden on this slide): `printf` is an external activity that prevents that the compiler removes all calculations as *"dead code"*

 ● REC → online

pi_taskloop_solution.f90 – solution with taskloop

```
program compute_pi
implicit none
integer i ; integer, parameter :: n=10000000
real(kind=8) w,x,sum,pi,f,a
integer i_outer
real(kind=8)sum_outer(0:63,0:99)!A cache line should be not larger than the 64 ints.
                                !Each task j should use sum_outer(0,j), i.e., each of
                                !these summing-variables is in a different cache-line
…
! calculate pi = integral [0..1] 4/(1+x**2) dx
w=1.0_8/n ; sum=0.0_8
do i_outer=0,99
  sum_outer(0,i_outer)=0.0
enddo
!$OMP PARALLEL
!$OMP  SINGLE
!$OMP   TASKLOOP PRIVATE(x,i)
  do i_outer=0,99
    do i=i_outer+1,n,100
      x=w*(i-0.5_8)
      sum_outer(0,i_outer) =  sum_outer(0,i_outer) + f(x);
    enddo
  enddo
!$OMP   END TASKLOOP
!$OMP  END SINGLE
!$OMP END PARALLEL
do i_outer=0,99
  sum = sum + sum_outer(0,i_outer)
enddo
pi=w*sum
write (*,'(/,a,1pg24.16)')   'computed pi = ', pi
```

Skeleton: Outer loop without reduction was added

To prevent cache-line false sharing

Solution: Using a taskloop within parallel / single

Skeleton: Summing up the outer loop outside of the numerical loop

 ● REC → online