OpenMP 6.0 Part 1: New Host-side Features

Christian Terboven, RWTH Aachen University

Michael Klemm, OpenMP ARB

HORIZON-EUROHPC-JU-2023-COE

EuroHPC Joint Undertaking

1 January 2024– 31 December 2026

Grant Agreement No 101143931

Agenda

- Who are Michael and Christian? (C+M)
- OpenMP's development process (M)
- OpenMP 6.0 base language improvements (M)
- Tasking updates (C)
- Loop transformations (M)
- Some other features (C)
- Q&A (M+C)

Who are Michael and Christian?

Michael and Christian

- Michael …
  - → Principal Member of Technical Staff
  - → Works in HPC since 2003
  - → Works on the Fortran OpenMP offload compiler for AMD Instinct™ Accelerators
  - → Is a member of the OpenMP language committee since 2009
  - → Chief Executive Officer of the OpenMP ARB since April 2016
- Christian …
  - → … is a senior scientist at RWTH Aachen University and leads the HPC group
  - → … does research on Parallel Programming and Performance
  - → … is a member of the OpenMP language committee since 2008 and co-chair of the Affinity subcom.
  - → is co-author of the book "Using OpenMP - The Next Step", published by MIT Press

OpenMP's development process

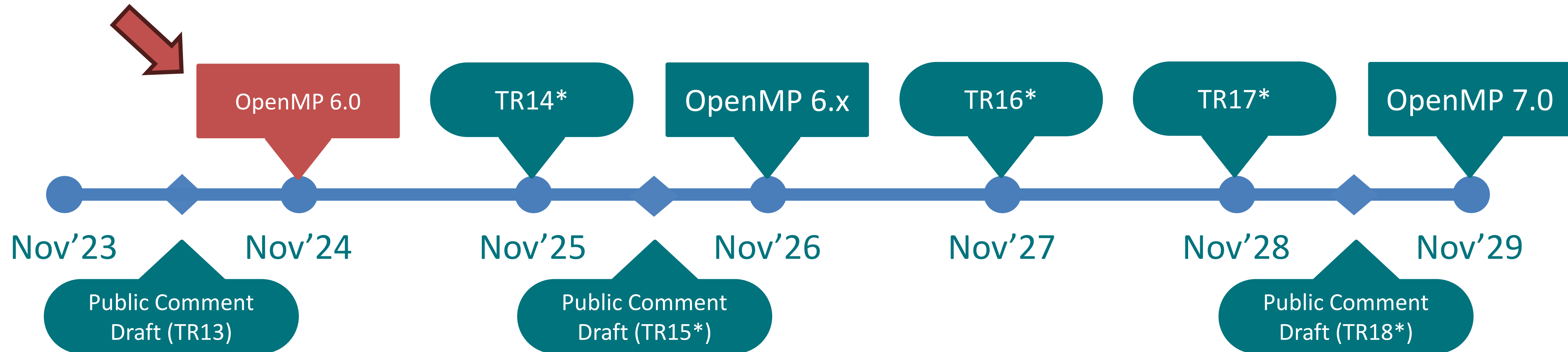
OpenMP Architecture Review Board

The mission of the OpenMP ARB (Architecture Review Board) is to standardize directive-based multi-language high-level parallelism that is performant, productive and portable.

The OpenMP API moves common approaches into an industry standard to simplify a developer's life.

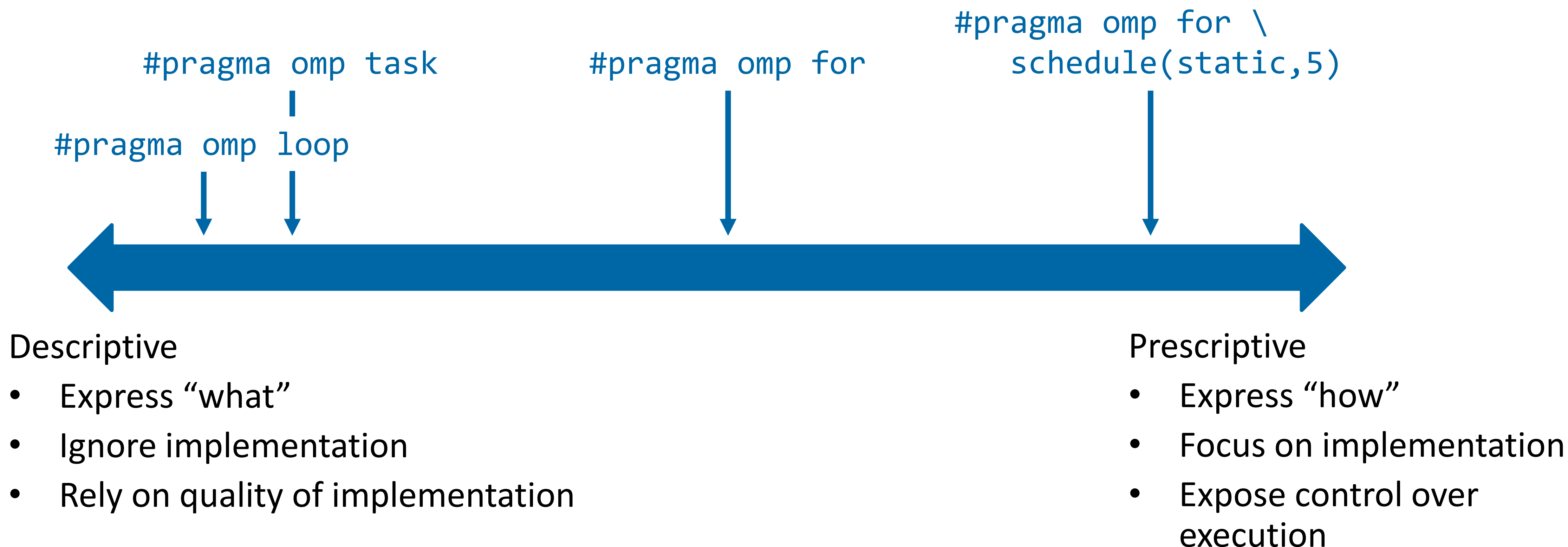
OpenMP Roadmap

- Roadmap for the releases of the OpenMP API
  - →5-year cadence for major releases, one minor release in between
  - →OpenMP 5.2 was an additional release before OpenMP version 6.0
  - →(At least) one Technical Report (TR) with feature previews in every year



* Numbers assigned to TRs may change if additional TRs are released.

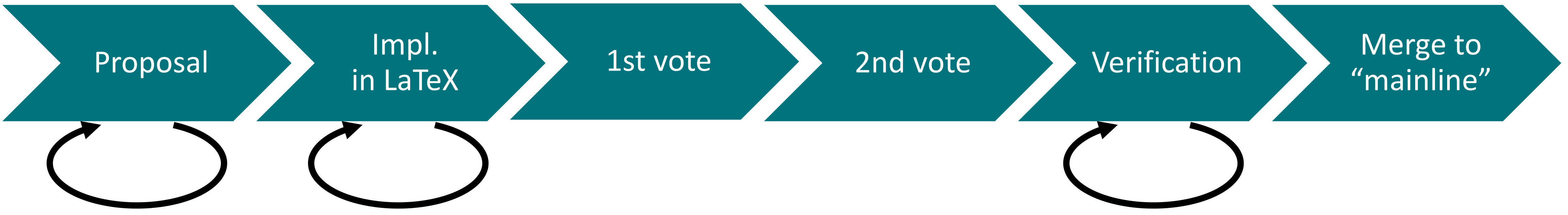
Continuum of Control



- OpenMP strives to
  - → Support a useful subset of this spectrum
  - → Provide a structured path from descriptive to prescriptive where needed

Development Process of the Specification

- Modifications of the OpenMP specification follow a (strict) process:



- Release process for specifications:

OpenMP 6.0 base language improvements

New Supported OpenMP Base Languages

- Complete support for C23
- Complete support for C++23
- Complete support for Fortran 2018
- Complete support for Fortran 2023

- There may be restrictions on using base language features, e.g.,
  - →Fortran: cannot use data-sharing clauses with Co-Arrays

Fortran BLOCK Constructs

- Fortran BLOCK constructs provide additional scopes:
  - → Removes the need for OpenMP `end` directives
  - → Helps privatize variables as part of their scope

```
subroutine hello()
    use omp_lib
    implicit none
    integer :: tid
    !$omp parallel private(tid)
        tid = omp_get_thread_num()
        print '(A,I4)', &
              'Hello from ', &
              tid
    !$omp end parallel
end subroutine
```

```
subroutine hello()
    use omp_lib
    implicit none
    !$omp parallel
    block
        integer :: tid
        tid = omp_get_thread_num()
        print '(A,I4)', &
              'Hello from ', &
              tid
    end block
end subroutine
```

C/C++ Attribute Syntax

- C++ introduced attributes with C++11; C introduced it with C23:

```
template<typename T, typename F>
void process(std::vector<T> &input, std::vector<T> &output, F &&func) {
    [[omp::directive(parallel for)]]
    for (auto &&element : input) {
        output.push_back(func(element));
    }
}
```

```
template<typename T, typename F>
void process(std::vector<T> &input, std::vector<T> &output, F &&func) {
    [[omp::sequence(directive(parallel),directive(for))]]
    for (auto &&element : input) {
        output.push_back(func(element));
    }
}
```

Tasking updates

Parts of the examples on these slides have been created by Stephen Olivier (SNL), chair of the Tasking subcommittee.

Free-agent threads / 1

- OpenMP 6.0 defines OpenMP threads as members of logical thread pool

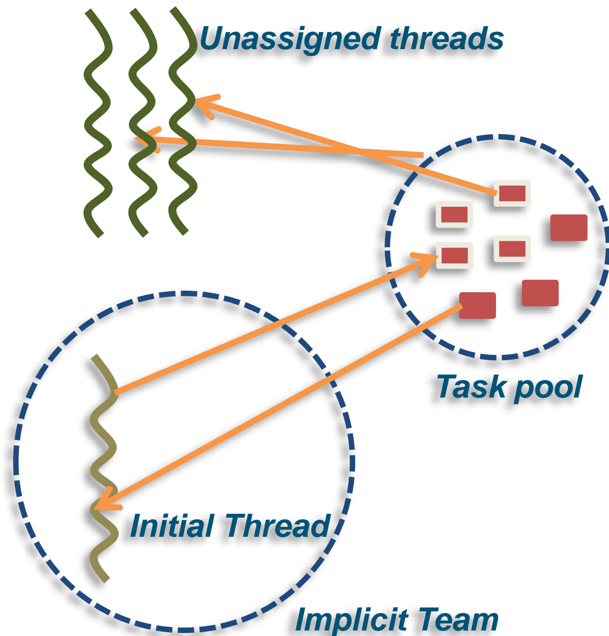
  → Pool size can be specified by `OMP_THREAD_LIMIT` environment variable

- OpenMP 6.0 also adds the concept of free-agent threads:
  "free-agent threads" **outside a team** can execute tasks
  - New `threadset` clause indicates which threads may execute the task:
    - `omp_team:` only threads in the team (default)
    - `omp_pool:` threads in the team AND unassigned threads in the contention group

Free-agent threads / 2

- Example:

```
// NO parallel masked NEEDED HERE!
while (elem != NULL) {
   #pragma omp task threadset(omp_pool)
      compute(elem);
   elem = elem->next;
}
```



Balance of structured parallelism and free-agent threads governed by ICVs that can be controlled through **OMP_THREADS_RESERVE**:

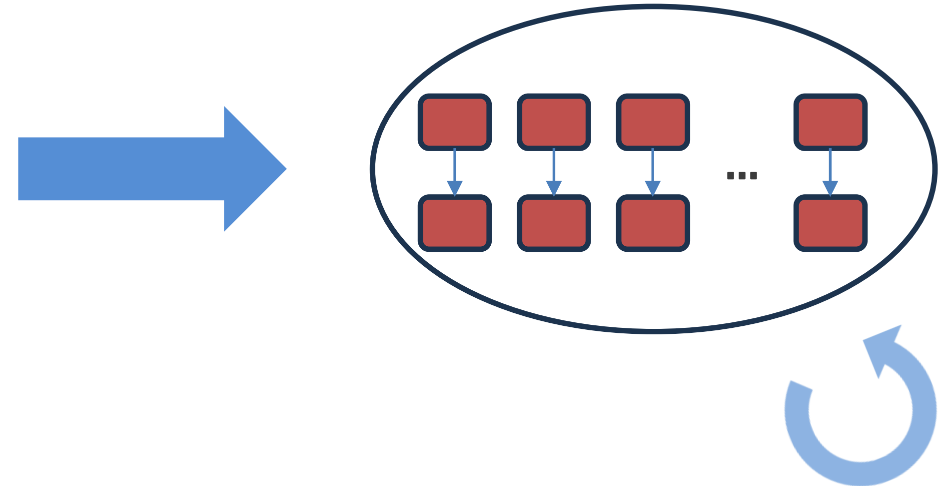
```
setenv OMP_THREADS_RESERVE "structured(4),free_agent(2)"
```

In example above, four threads reserved for structured parallelism (assignment to teams) and two threads to act as free-agents

Taskgraph

- `taskgraph` construct optimizes repeated tasks
  - → Implementation creates a record of the sequence of tasks and dependences

```
while (residual > TOLERANCE) {
   #pragma omp taskgraph graph_id(0)
   {
      for (j = 0; j < A_SIZE; ++j) {
         #pragma omp task depend (out: A[j])
            preprocess(A[j]);
         #pragma omp task depend (in: A[j])
            compute(A[j]);
      }
    }
    residual = calc_residual(A);
}
```



  - → **Tasks replayable by default, use `replayable(false)` to disallow replay**
    The `graph_reset` clause can be used to discard the existing record

Task iteration

- Support the `depend` and `affinity` clause in combination with the `taskloop` construct
  → Specify at the start of the loop body with the `task_iteration` directive

```
// Example: Dependencies between tasks within a taskloop as well as between taskloop and standalone task
#pragma omp taskloop nogroup
for (int i = 1; i < n; i++) {
    #pragma omp task_iteration depend(inout: A[i]) depend(in: A[i-1])
    A[i] += A[i] * A[i-1];
}

// TL2 taskloop + grainsize
#pragma omp taskloop grainsize(strict: 4) nogroup
for (int i = 1; i < n; i++) {
    #pragma omp task_iteration depend(inout: A[i]) depend(in: A[i-4]) if ((i % 4) == 0 || i == n-1)
    A[i] += A[i] * A[i-1];
}

// T3 other task
#pragma omp task depend(in: A[n-1])
printf("A[n-1] = %f\n",  A[n-1]);
```

Transparent tasks

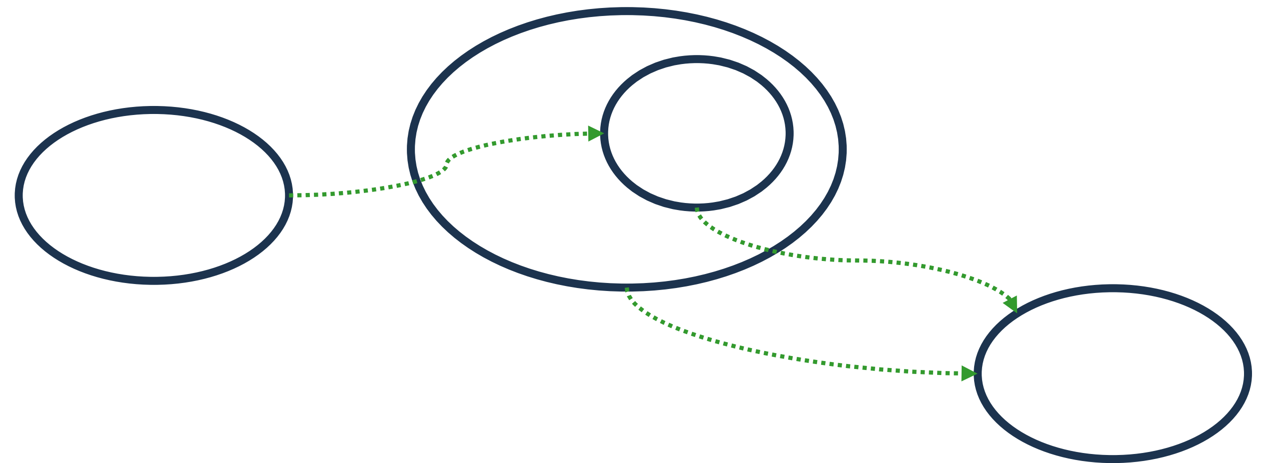
- Tasks can have dependencies on each other using the `depend` clause
  - → Traditionally, these were limited to tasks siblings (same `task` or `taskgroup`)
  - → Transparent tasks remove this limitation

```
#pragma omp task depend(out: x)
   { ... }        // T1

#pragma omp task depend(out: y) transparent
{                 // T2

   #pragma omp task depend(inout: x)
      { ... }     // T3 – must wait on T1
}

#pragma omp task depend(in: x, y)
   { ... }        // T4 – must wait on T2, T3
```



- Use case: composable software components
  - → Task dependencies can now be maintained with libraries
  - → Deadlock freedom is still guaranteed

Loop transformations

Loop Unrolling

- Loop unrolling is a standard tuning practice to reduce loop overhead and increase potential for pipeline.

```
subroutine loop()
    do i = 1, 4
        call body(i)
    end do
end subroutine loop
```

```
subroutine loop()
    call body(i + 0)
    call body(i + 1)
    call body(i + 2)
    call body(i + 3)
end subroutine loop
```

```
subroutine loop()
    !$omp unroll full
    do i = 1, 4
        call body(i)
    end do
end subroutine loop
```

- "`full`" completely unrolls the loop
  - → Needs a compile-time constant upper bound.
  - → Loop is no longer present after unrolling took place.

Loop Unrolling

- Loop unrolling is a standard tuning practice to reduce loop overhead and increase potential for pipeline.

```
subroutine loop()
    do i = 1, n
        call body(i)
    end do
end subroutine loop
```

```
subroutine loop()
    do i = 1, n, 4
        call body(i + 0)
        call body(i + 1)
        call body(i + 2)
        call body(i + 3)
    end do
end subroutine loop
```

```
subroutine loop()
    !$omp unroll partial(4)
    do i = 1, n
        call body(i)
    end do
end subroutine loop
```

- "`partial(`*f*`)`" unrolls the loop with unroll factor *f*
  - → Upper bound can now be a runtime value
  - → Compiler will introduce remainder loops as necessary

Tiling

- Tiling is a useful to optimize a loop nest for the cache hierarchy and exploiting temporal/spatial locality

```
subroutine loop()
    !$omp tile sizes(2,2)
    do i = 1, n
        do j = 1, m
            call body(j, i)
        end do
    end do
end subroutine loop
```

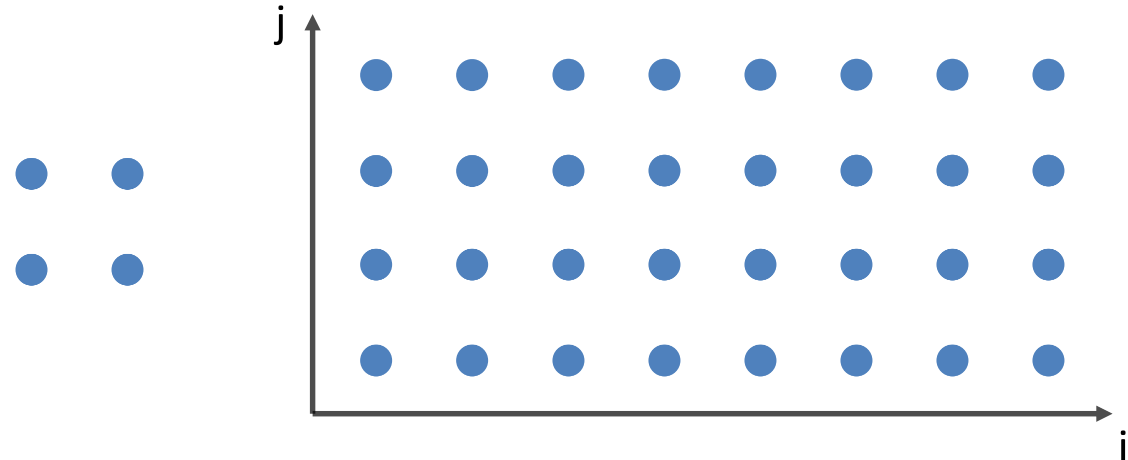
```
subroutine loop()
    do ii = 1, n, 2
        do jj = 1, m, 2
            do i = ii, ii + 2
                do j = jj, jj + 2
                    call body(j, i)
                end do
            end do
        end do
    end do
end subroutine loop
```

Handling of partial tiles needed!

Tiling

- Tiling is a useful to optimize a loop nest for the cache hierarchy and exploiting temporal/spatial locality

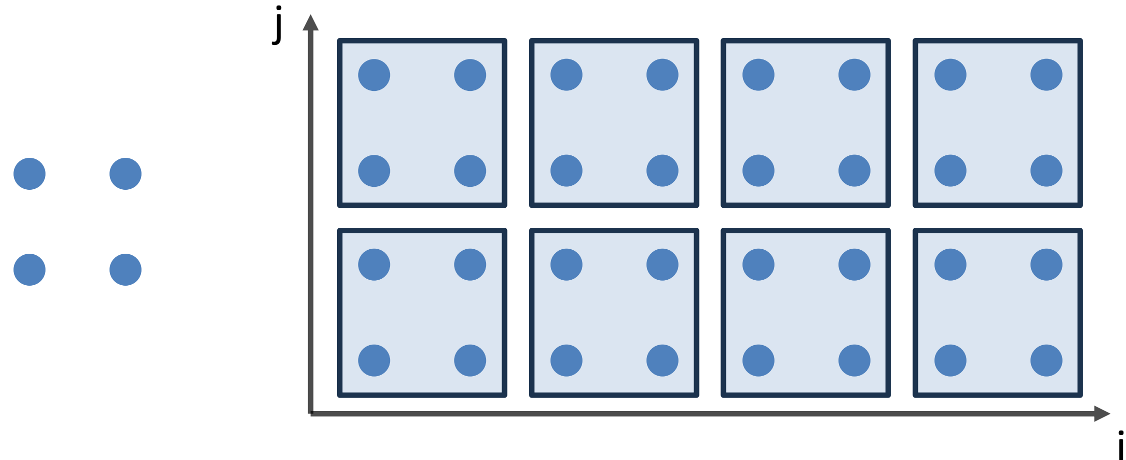
```
subroutine loop()
    !$omp tile sizes(2,2)
    do i = 1, n
        do j = 1, m
            call body(j, i)
        end do
    end do
end subroutine loop
```



Tiling

- Tiling is a useful to optimize a loop nest for the cache hierarchy and exploiting temporal/spatial locality

```
subroutine loop()
    !$omp tile sizes(2,2)
    do i = 1, n
        do j = 1, m
            call body(j, i)
        end do
    end do
end subroutine loop
```

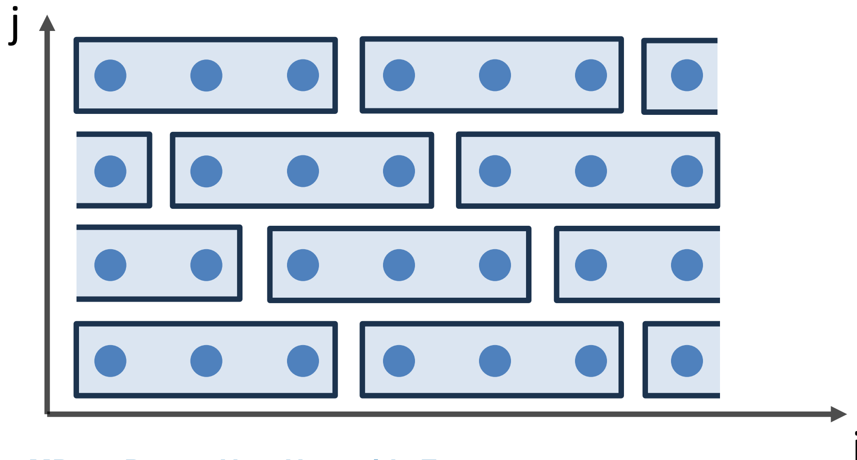


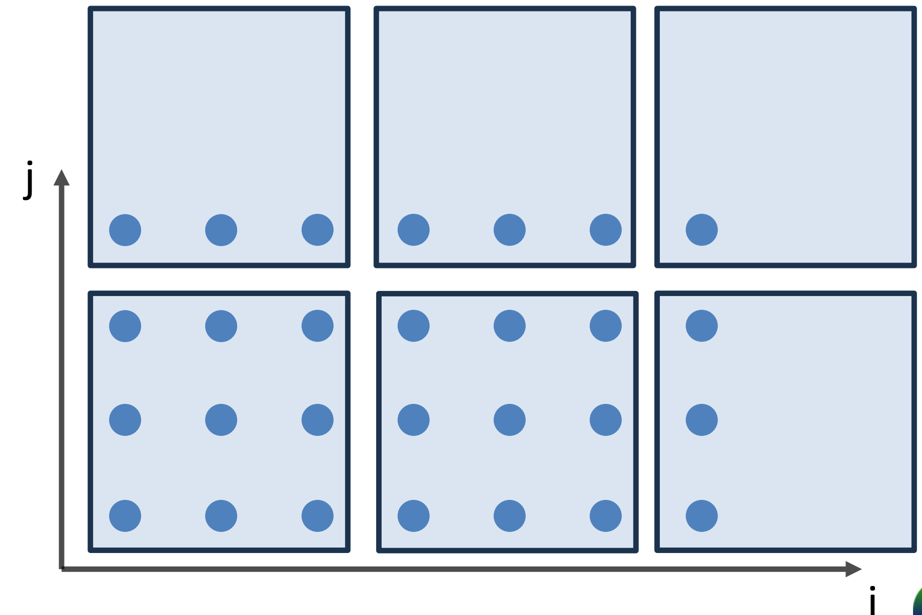
Tiling and Chunking

- One can think of tiling as “multi-dimensional” chunking:

```
!$omp for schedule(static, 3) &
             collapse(2)
do i = 1, n
     do j = 1, m
          call body(j, i)
     end do
end do
```

```
!$omp tile sizes(3,3)
do i = 1, n
     do j = 1, m
          call body(j, i)
     end do
end do
```





Other Loop Transformations /1

- Loop Interchange

```
!$omp interchange permutation(3,1,2)
do i = 1, n
    do j = 1, m
        do k = 1, l
            call body(j, i, k)
        end do
    end do
end do
```

```
do k = 1, l
    do i = 1, n
        do j = 1, m
            call body(j, i, k)
        end do
    end do
end do
```

- Loop Reversal

```
!$omp reverse
do i = 1, n
    call body(i)
end do
```

```
do i = 1, n
    call body(n - (i - 1))
end do
```

Other Loop Transformations /2

- Loop Fusion

```
!$omp fuse
do i = 1, n
    call body1(i)
end do
do i = 1, n
    call body2(i)
end do
!$omp end fuse
```

```
do i = 1, n
    call body1(i)
    call body2(i)
end do
```

- Loop Reversal

```
!$omp reverse
do i = 1, n
    call body(i)
end do
```

```
do i = 1, n
    call body(n - (i - 1))
end do
```

Other Loop Transformations /3

- Loop Index Splitting

```
!$omp split counts(k, omp_fill)
do i = 1, n
     call body(i)
end do
```

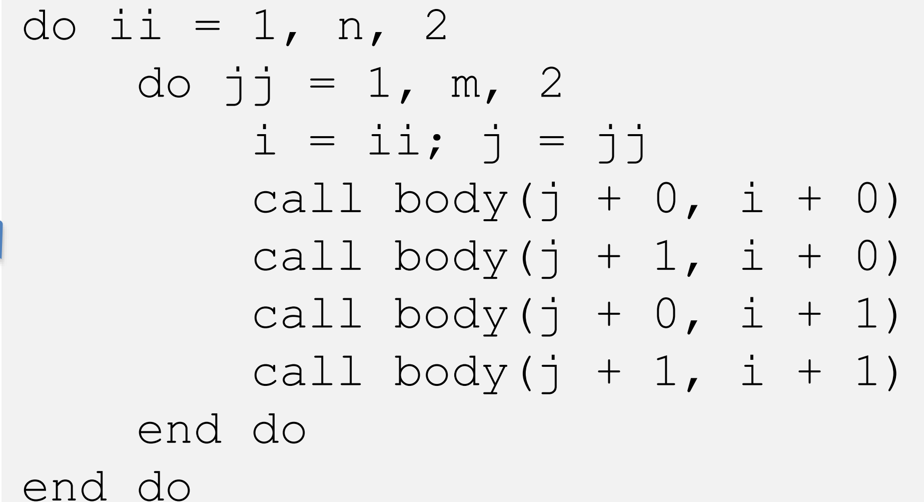
```
do i = 1, k
     call body(i)
end do
do i = k, n
     call body(i)
end do
```

- All these transformations can be useful:
  - →Fusion: reduce loop overhead and get more work per loop iteration
  - →Reversal: create forward memory references
  - →Index splitting: peel off loop iterations, e.g., for better SIMD/memory alignment

Composing Loop Transformations

- Loop transformations can be composed, e.g., tiling and unrolling:

```
!$omp tile sizes(2,2) &
      apply(intratile:unroll full, &
                      unroll full)
do i = 1, n
    do j = 1, m
        call body(j, i)
    end do
end do
```



```
do ii = 1, n, 2
    do jj = 1, m, 2
        i = ii; j = jj
        call body(j + 0, i + 0)
        call body(j + 1, i + 0)
        call body(j + 0, i + 1)
        call body(j + 1, i + 1)
    end do
end do
```

Some other features

Parts of the examples on these slides have been created by Bronis de Supinski (LLNL), chair of the Language committee.

User-defined inductions / 1

- Induction allows parallelization despite dependences
  - →**Iterative Computation:** A result is calculated through repeated application of an operation.
  - →**Parallel Accumulation:** These operations can be performed (at least partially) in parallel, with partial results being combined to form the final result.
  - →**Dependency or Ordering (sometimes):** Unlike reductions where the order of operations is often irrelevant (e.g., summation), "inductive" processes sometimes might have some inherent order or dependency between the steps.

```
xi = x0;
result = 0.0;
#pragma omp parallel for reduction(+: result) induction(step(x), *: xi)
for (I = 0; I < N; i++) {
  result += c[i] * xi;
  xi *= x;
}
```

User-defined inductions / 2

- User-defined inductions enable complex computations w/ dependences
  - → Can use `collector` clause to specify closed form function to enable starting at arbitrary iterations: `x_i = x_0 + (s * i)` for step `s`

```
typedef struct{ float a; int b; } component;
void add(A *x, int st) { x->a += st; x->b += st; }

#pragma omp declare induction( op : \                     /* name */
                               component : \              /* type */
                               int : \                    /* step type */
                               add(&omp_out, omp_step)) \ /* inductor */
                               collector( omp_step = omp_index * omp_step )

component c = /* ... */;
#pragma omp parallel for induction( op : c : 5 )
for(int i=0; i<N; i++) { /* do sth else */ add(&c, 5); /* do sth else */ }
```

  - → Here: amount added to struct's member per iteration is not constant, but multiplied by l. index
  - → User-defined inductions can be used with SIMD loops as well

Q&A

Performance Optimisation and Productivity 3

A Centre of Excellence in HPC

Contact:

https://www.pop-coe.eu

pop@bsc.es

@POP_HPC

youtube.com/POPHPC

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 101143931. The JU receives support from the European Union's Horizon Europe research and innovation programme and Spain, Germany, France, Portugal and the Czech Republic.