Progamming the OpenMP API

Misc Topics & 6.0 Outlook

OpenMP Parallel Loops

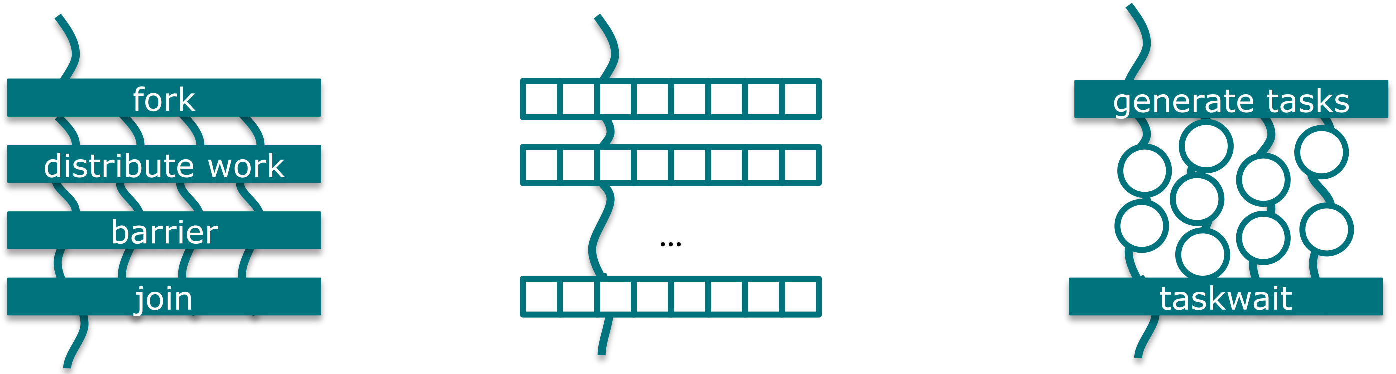
`loop` Construct

- Existing loop constructs are tightly bound to execution model:

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

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

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



- The **`loop`** construct is meant to tell OpenMP about truly parallel semantics of a loop.

OpenMP Fully Parallel Loops

```
int main(int argc, const char* argv[]) {
    float *x = (float*) malloc(n * sizeof(float));
    float *y = (float*) malloc(n * sizeof(float));
    // Define scalars n, a, b & initialize x, y

#pragma omp parallel
#pragma omp loop
    for (int i = 0; i < n; ++i){
      y[i] = a*x[i] + y[i];
    }
  }
}
```

`loop` Constructs, Syntax

- Syntax (C/C++)

```
#pragma omp loop [clause[[,] clause],…]
for-loops
```

- Syntax (Fortran)

```
!$omp loop [clause[[,] clause],…]
do-loops
[!$omp end loop]
```

`loop` Constructs, Clauses

- `bind(`*`binding`*`)`
  - → Binding region the loop construct should bind to
  - → One of: `teams`, `parallel`, `thread`
- `order(concurrent)`
  - → Tell the OpenMP compiler that the loop can be executed in any order.
  - → Default!
- `collapse(`*`n`*`)`
- `private(`*`list`*`)`
- `lastprivate(`*`list`*`)`
- `reduction(`*`reduction-id`*`:`*`list`*`)`

Extensions to Existing Constructs

- Existing loop constructs have been extended to also have truly parallel semantics.

- C/C++ Worksharing:

```
#pragma omp [for|simd] order(concurrent) \
                          [clause[[,] clause],…]
for-loops
```

- Fortran Worksharing:

```
!$omp [do|simd] order(concurrent) &
                 [clause[[,] clause],…]
do-loops
[!$omp end [do|simd}]
```

DOACROSS Loops

DOACROSS Loops

- "DOACROSS" loops are loops with special loop schedules
  - → Restricted form of loop-carried dependencies
  - → Require fine-grained synchronization protocol for parallelism

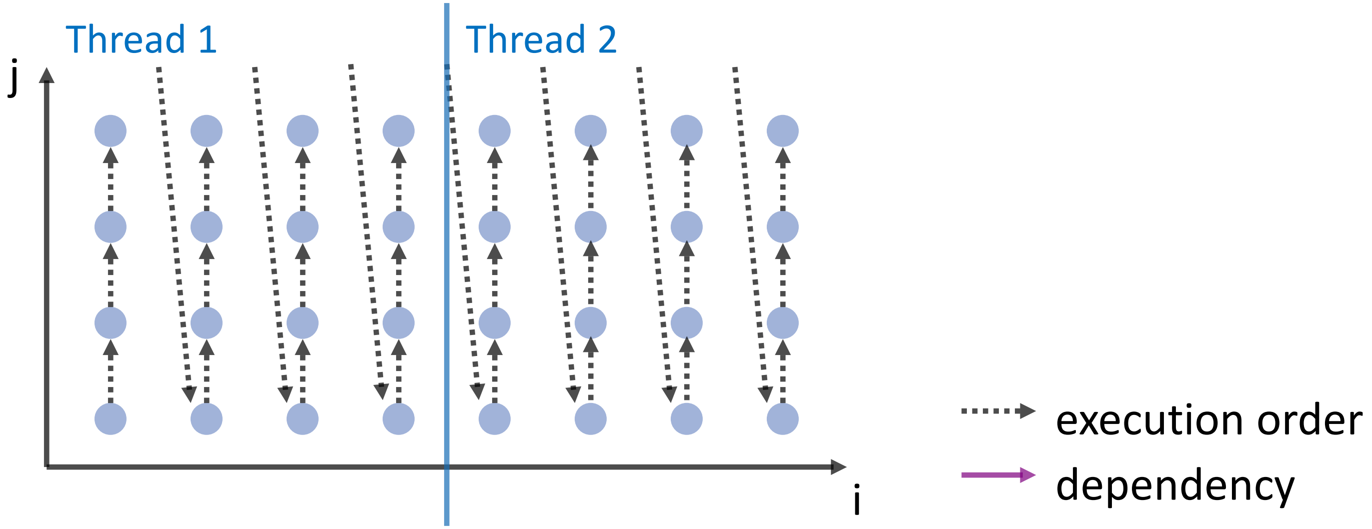
- Loop-carried dependency:
  - → Loop iterations depend on each other
  - → Source of dependency must scheduled before sink of the dependency

- DOACROSS loop:
  - → Data dependency is an invariant for the execution of the whole loop nest

Parallelizable Loops

- A parallel loop cannot not have any loop-carried dependencies (simplified just a little bit!)

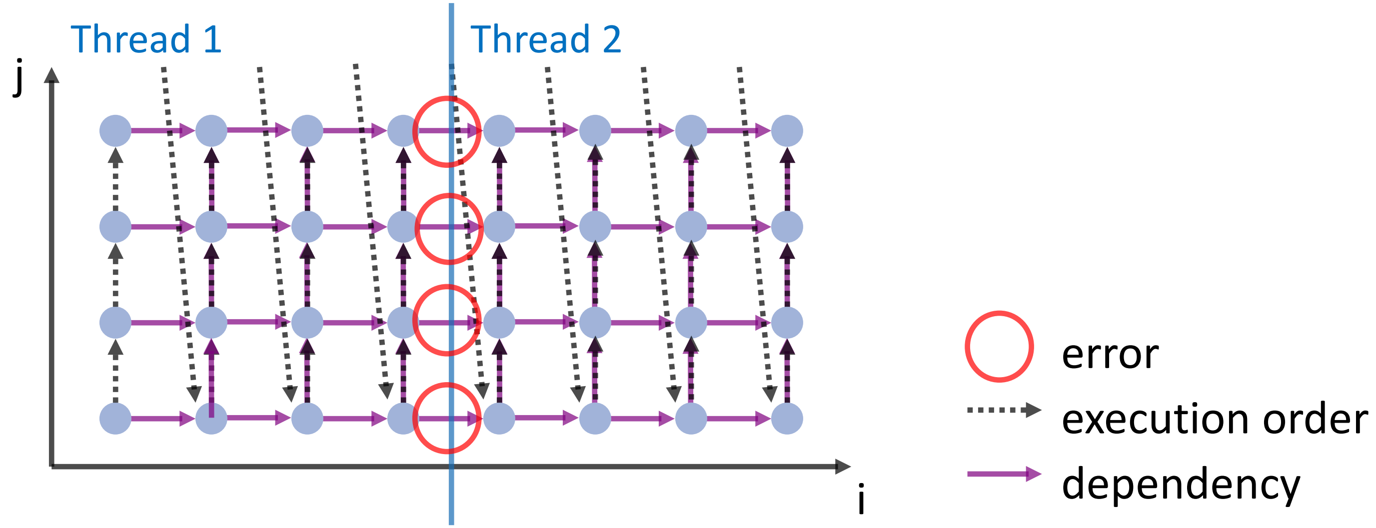
```
for (int i = 1; i < N; ++i) {
    for (int j = 1; j < M; ++j) {
        b[i][j] = f(b[i][j],
                    b[i][j], a[i][j]);
    }
}
```



Non-parallelizable Loops

- If there is a loop-carried dependency, a loop cannot be parallelized anymore (“easily” that is)

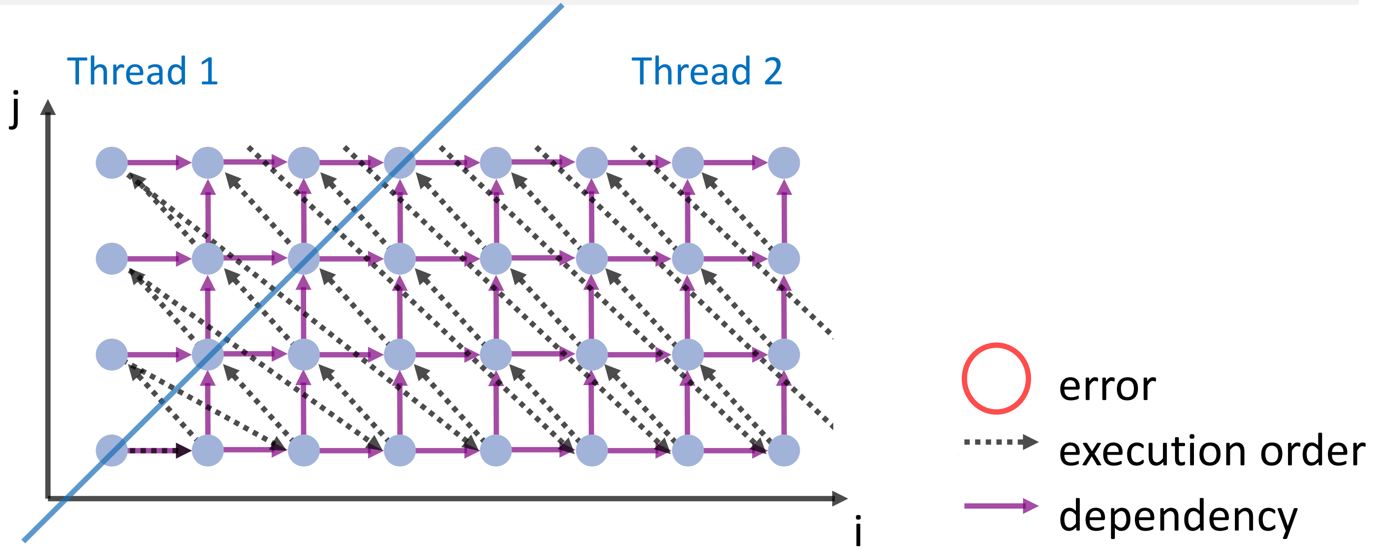
```
for (int i = 1; i < N; ++i) {
    for (int j = 1; j < M; ++j) {
        b[i][j] = f(b[i-1][j],
                    b[i][j-1], a[i][j]);
    }
}
```



Wavefront-Parallel Loops

- If the data dependency is invariant, then skewing the loop helps remove the data dependency

```
for (int i = 1; i < N; ++i) {
    for (int j = i+1; j < i+N; ++j) {
        b[i][j-i] = f(b[i-1][j-i],
                      b[i][j-i-1], a[i][j]);
    }
}
```



DOACROSS Loops with OpenMP

Deprecated in v5.2

- OpenMP 4.5 extends the notion of the ordered construct to describe loop-carried dependencies
- Syntax (C/C++):

```
#pragma omp for ordered(d) [clause[[,] clause],…]
for-loops
```

and

```
#pragma omp ordered [clause[[,] clause],…]
```

where clause is one of the following:

```
depend(source)
depend(sink:vector)
```

- Syntax (Fortran):

```
!$omp do ordered(d) [clause[[,] clause],…]
do-loops
!$omp ordered [clause[[,] clause],…]
```

Example

- The ordered clause tells the compiler about loop-carried dependencies and their distances

Deprecated in v5.2

```
#pragma omp parallel for ordered(2)
for (int i = 1; i < N; ++i) {
    for (int j = 1; j < M; ++j) {
#pragma omp ordered depend(sink:i-1,j) depend(sink:i,j-1)
        b[i][j] = f(b[i-1][j],
                    b[i][j-1], a[i][j]);
    }
#pragma omp ordered depend(source)
}
```

Example: 3D Gauss-Seidel

OpenMP

```
#pragma omp for ordered(2) private(j,k)
for (i = 1; i < N-1; ++i) {
  for (j = 1; j < N-1; ++j)    {
#pragma omp ordered depend(sink: i-1,j-1) depend(sink: i-1,j) \
                    depend(sink: i-1,j+1) depend(sink: i,j-1)
    for (k = 1; k < N-1; ++k) {
      double tmp1 = (p[i-1][j-1][k-1] + p[i-1][j-1][k] + p[i-1][j-1][k+1]
                   + p[i-1][j][k-1] + p[i-1][j][k] + p[i-1][j][k+1]
                   + p[i-1][j+1][k-1] + p[i-1][j+1][k] + p[i-1][j+1][k+1]);
      double tmp2 = (p[i][j-1][k-1] + p[i][j-1][k] + p[i][j-1][k+1]
                   + p[i][j][k-1] + p[i][j][k] + p[i][j][k+1]
                   + p[i][j+1][k-1] + p[i][j+1][k] + p[i][j+1][k+1]);
      double tmp3 = (p[i+1][j-1][k-1] + p[i+1][j-1][k] + p[i+1][j-1][k+1]
                   + p[i+1][j][k-1] + p[i+1][j][k] + p[i+1][j][k+1]
                   + p[i+1][j+1][k-1] + p[i+1][j+1][k] + p[i+1][j+1][k+1]);
      p[i][j][k] = (tmp1 + tmp2 + tmp3) / 27.0;
    }
#pragma omp ordered depend(source)
  }
}
```

DOACROSS Loops with OpenMP

- OpenMP 4.5 extends the notion of the ordered construct to describe loop-carried dependencies
- Syntax (C/C++):

```
#pragma omp for ordered [clause[[,] clause],…]
for-loops
```

and

```
#pragma omp ordered [clause[[,] clause],…]
```

where clause is one of the following:

`doacross(source:vector)`, vector can be `omp_cur_iteration`
`doacross(sink:vector)`

- Syntax (Fortran):

```
!$omp do ordered [clause[[,] clause],…]
do-loops
!$omp ordered [clause[[,] clause],…]
```

Example

- The ordered clause tells the compiler about loop-carried dependencies and their distances

```
#pragma omp parallel for ordered
for (int i = 1; i < N; ++i) {
    for (int j = 1; j < M; ++j) {
#pragma omp ordered doacross(sink:i-1,j) doacross(sink:i,j-1)
        b[i][j] = f(b[i-1][j],
                    b[i][j-1], a[i][j]);
    }
#pragma omp ordered doacross(source:omp_cur_iteration)
}
```

Example: 3D Gauss-Seidel

```
#pragma omp for ordered private(j,k)
for (i = 1; i < N-1; ++i) {
  for (j = 1; j < N-1; ++j)   {
#pragma omp ordered doacross(sink: i-1,j-1) doacross(sink: i-1,j) \
                    doacross(sink: i-1,j+1) doacross(sink: i,j-1)
    for (k = 1; k < N-1; ++k) {
      double tmp1 = (p[i-1][j-1][k-1] + p[i-1][j-1][k] + p[i-1][j-1][k+1]
                   + p[i-1][j][k-1] + p[i-1][j][k] + p[i-1][j][k+1]
                   + p[i-1][j+1][k-1] + p[i-1][j+1][k] + p[i-1][j+1][k+1]);
      double tmp2 = (p[i][j-1][k-1] + p[i][j-1][k] + p[i][j-1][k+1]
                   + p[i][j][k-1] + p[i][j][k] + p[i][j][k+1]
                   + p[i][j+1][k-1] + p[i][j+1][k] + p[i][j+1][k+1]);
      double tmp3 = (p[i+1][j-1][k-1] + p[i+1][j-1][k] + p[i+1][j-1][k+1]
                   + p[i+1][j][k-1] + p[i+1][j][k] + p[i+1][j][k+1]
                   + p[i+1][j+1][k-1] + p[i+1][j+1][k] + p[i+1][j+1][k+1]);
      p[i][j][k] = (tmp1 + tmp2 + tmp3) / 27.0;
    }
#pragma omp ordered doacross(source:omp_cur_iteration)
  }
}
```

OpenMP Meta-Programming

The `metadirective` Directive

- Construct OpenMP directives for different OpenMP contexts
- Limited form of meta-programming for OpenMP directives and clauses

```
#pragma omp target map(to:v1,v2) map(from:v3)
#pragma omp metadirective \
            when( device={arch(nvptx)}: teams loop ) \
            default( parallel loop )
for (i = lb; i < ub; i++)
   v3[i] = v1[i] * v2[i];
```

```
!$omp begin metadirective &
            when( implementation={unified_shared_memory}: target ) &
            default( target map(mapper(vec_map),tofrom: vec) )
!$omp teams distribute simd
do i=1, vec%size()
   call vec(i)%work()
end do
!$omp end teams distribute simd
!$omp end metadirective
```

Nothing Directive

The `nothing` Directive

- The `nothing` directive makes meta programming a bit clearer and more flexible.
- If a certain criterion matches, the nothing directive can stand to indicate that no (other) OpenMP directive should be used.
  - → The `nothing` directive is implicitly added if no condition matches

```
!$omp begin metadirective &
            when( implementation={unified_shared_memory}: &
                  target teams distribute parallel do simd) &
            default( nothing )
do i=1, vec%size()
   call vec(i)%work()
end do
!$omp end metadirective
```

Error Directive

`Error` Directive Syntax

- Syntax (C/C++)
  ```
  #pragma omp error [clause[[,] clause],…]
  for-loops
  ```

- Syntax (Fortran)
  ```
  !$omp error [clause[[,] clause],…]
  do-loops
  [!$omp end loop]
  ```

- Clauses
  one of: `at(compilation)`, `at(runtime)`
  one of: `severity(fatal)`, `severity(warning)`
  `message(`*`msg-string`*`)`

Error Directive

- Can be used to issue a warning or an error at compile time and runtime.
- Consider this a “directive version” of `assert()`, but with a bit more flexibility.

```
#pragma omp parallel
{
    if (omp_get_num_threads() % 2) {
#pragma omp error at(runtime) severity(warning) \
                  message(“Running on odd number of threads\n”);
    }
    do_stuff_that_works_best_with_even_thread_count();
}
```

Error Directive

- Can be used to issue a warning or an error at compile time and runtime.
- Consider this a “directive version” of `assert()`, but with a bit more flexibility.
- More useful in combination with OpenMP metadirective

```
!$omp begin metadirective &
            when( arch={fancy_processor}: parallel ) &
            default( error severity(fatal) at(compilation) &
                           message(“No implementation available” )
    call fancy_impl_for_fancy_processor()
!$omp end metadirective
```

Free-agent threads

(OpenMP 6.0 feature)

Recall the tasking execution model

- Supports unstructured parallelism
  - → unbounded loops

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

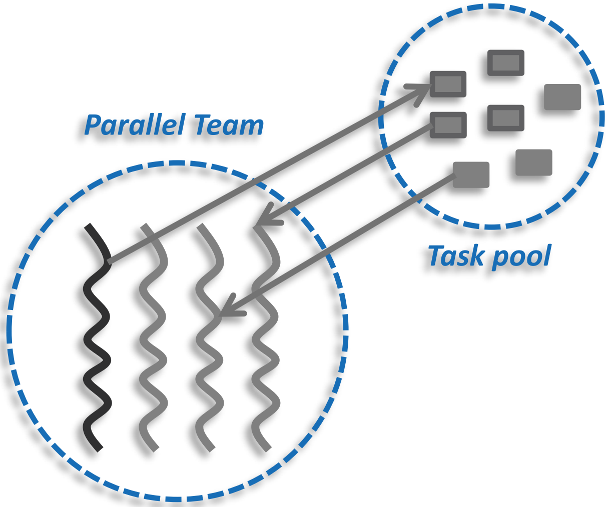
  - → recursive functions

```
void myfunc( <args> )
{
    ...; myfunc( <newargs> ); ...;
}
```

- Example (unstructured parallelism)

```
#pragma omp parallel
#pragma omp single
while (elem != NULL) {
    #pragma omp task
        compute(elem);
    elem = elem->next;
}
```

- Why are the **parallel** and **single** directives needed?
  - → Otherwise all threads in the team generate (duplicate) tasks
  - → Only threads in the team may execute tasks



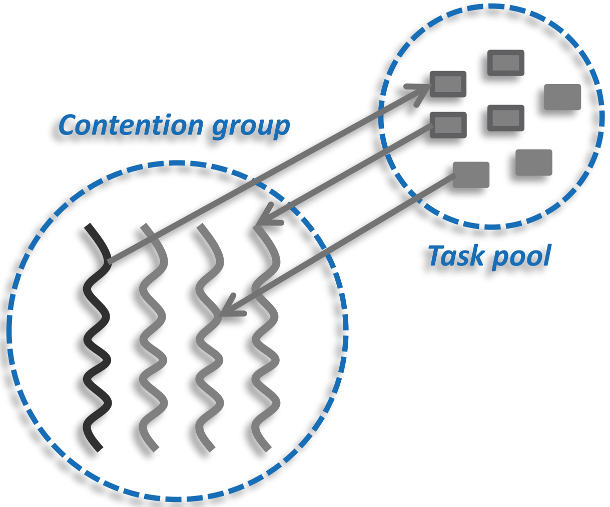
Is restricting tasks to a team good?

- Positive aspects
  - → Simplifies resource management
  - → Clear semantics with respect to other teams
- Negative aspects
  - → Ignores unutilized resources
  - → Complicates code structure for task-only programs

- Example (no **`parallel`** directive needed)

```
while (elem != NULL) {
   #pragma omp task threadset(omp_pool)
      compute(elem);
   elem = elem->next;
}
```

- Alternative starting in OpenMP 6.0: free-agent threads
  - → Unassigned threads in contention group may execute tasks
  - → Can provide parallelism in the implicit parallel region
  - → Exploits unused resources, common practice of parked threads



Some details for free-agent threads

- Existing behavior is preserved by default
  - → As if **`threadset`** clause is specified with value of **`omp_team`**

```
#pragma omp task threadset(omp_team)
{structured-block}
```

  - → Task synchronization (e.g., dependences, **`taskwait`** and **`taskgroup`**) unchanged
- Can use environment variables to control ICVs to reserve threads
  - → At least two threads available for structured parallelism, at least two available to act as free-agents
  - → Minimum for structured parallelism is one (the initial thread)
  - → Sum of reservations should not exceed *thread-limit-var* ICV

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

Future Directions

OpenMP 6.0 will be released in November 2024

- TR12 demonstrates appropriate progress for second TR of a major version
- Major new feature targets have been clearly identified and are on track for 2024
  - → Free-agent threads significantly change execution model, implementations
  - → User-defined induction and `induction` clause expand parallelism support
  - → Many significant device support improvements (e.g., `memscope(all)`) added or planned
  - → Several other additions and improvements planned, including:
    - → Rationalization of definition of combined constructs
    - → Task dependences between concurrently generated tasks
  - → Significant improvements to usability and correctness of specification
  - → TR13 (final comment draft) will be released in summer 2024

Major new features will characterize OpenMP 6.0

- Free-agent threads
  - → Support for top-level task parallelism (i.e., explicit `parallel` directive not needed)
  - → "Any" thread can execute explicit tasks for which `threadset` clause evaluates is `omp_pool`
  - → Adds associated runtime routines, environment variables and ICVs
- Major improvements for use of a single device
  - → Explicit progress guarantee adopted in TR11
  - → Default device and visible devices to simplify control of device use and availability
  - → Mechanisms to simplify use of device memory (by providing greater certainty or clarity)
    - → New `groupprivate` directive in TR11 is an initial mechanism in this direction
    - → Added `selfmap` modifier to ensure no copy is created when possible
    - → Unified host and device allocators and added significant cross-device improvements
  - → TR12 added `coexecute` directive (i.e., descriptive array language offload support)

OpenMP 6.0 will include other significant new features

- A more complete set of loop transforming directives
  - → TR12 includes `fuse`, `reverse` and `interchange` directives
  - → Considering other transformations that include `fission` and `nestify`
  - → Can now transform generated loops using the `apply` clause
- Clauses and directives to support generalized induction
  - → Capture computation that follows a well-defined sequence across loop iterations
  - → Generalizes behavior of `linear` clause and of loop iteration variables
  - → Related to reductions, including addition of `declare induction` directive