



# OpenMP API Version 5.0

(or: Pretty Cool & New OpenMP Stuff)

Michael Klemm  
Chief Executive Officer  
OpenMP Architecture Review Board  
[michael.klemm@openmp.org](mailto:michael.klemm@openmp.org)

# 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.



# Membership Structure

## ■ ARB Member

- Highest membership category
- Participation in technical discussions and organizational decisions
- Voting rights on organizational topics
- Voting rights on technical topics (tickets, TRs, specifications)

## ■ ARB Advisor & ARB Contributors

- Contribute to technical discussions
- Voting rights on technical topics (tickets, TRs, specifications)

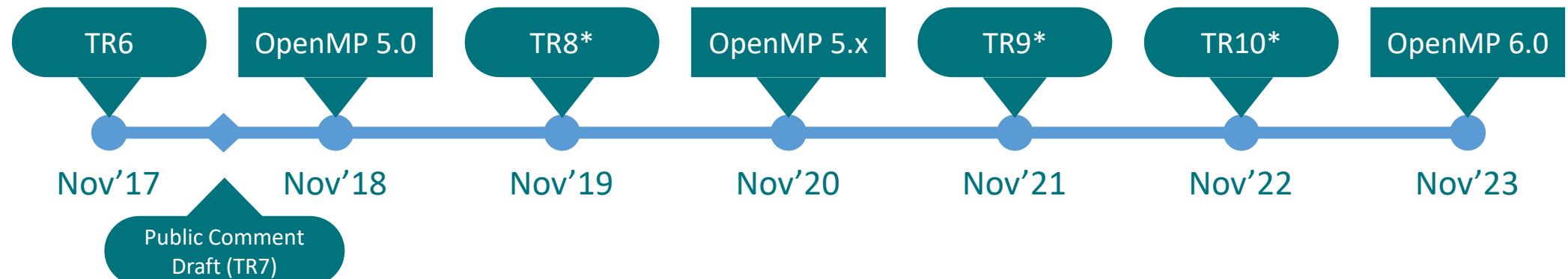
Your organization can join and influence the direction of OpenMP.

Talk to me or send email to [michael.klemm@openmp.org](mailto:michael.klemm@openmp.org).

# OpenMP Roadmap

## ■ OpenMP has a well-defined roadmap:

- 5-year cadence for major releases
- One minor release in between
- (At least) one Technical Report (TR) with feature previews in every year



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

# Levels of Parallelism in OpenMP 4.5

Cluster	Group of computers communicating through fast interconnect
Coprocessors/Accelerators	Special compute devices attached to the local node through special interconnect
Node	Group of cache coherent processors communicating through shared memory/cache
Core	Group of functional units within a die communicating through registers
Hyper-Threads	Group of thread contexts sharing functional units
Superscalar	Group of instructions sharing functional units
Pipeline	Sequence of instructions sharing functional units
Vector	Single instruction using multiple functional units

# OpenMP Version 5.0

- OpenMP 5.0 will introduce new powerful features to improve programmability

Task Reductions

C++14 and C++17 support

loop Construct

Multi-level Parallelism

Parallel Scan

Memory Allocators

Dependence Objects

Fortran 2008 support

Task-to-data Affinity

Data Serialization for Offload

Meta-directives

Function Variants

“Reverse Offloading”

Improved Task Dependences

Detachable Tasks

Tools APIs

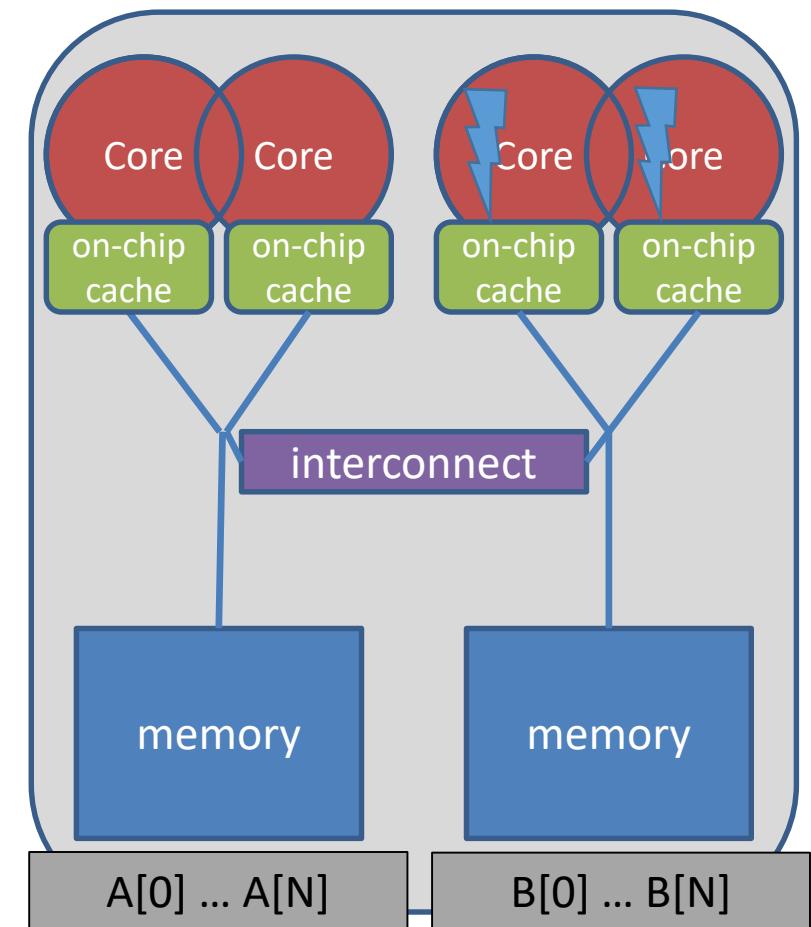
Unified Shared Memory

Collapse non-rect. Loops

# Task-to-data Affinity

- OpenMP API version 5.0 supports affinity hints for OpenMP tasks

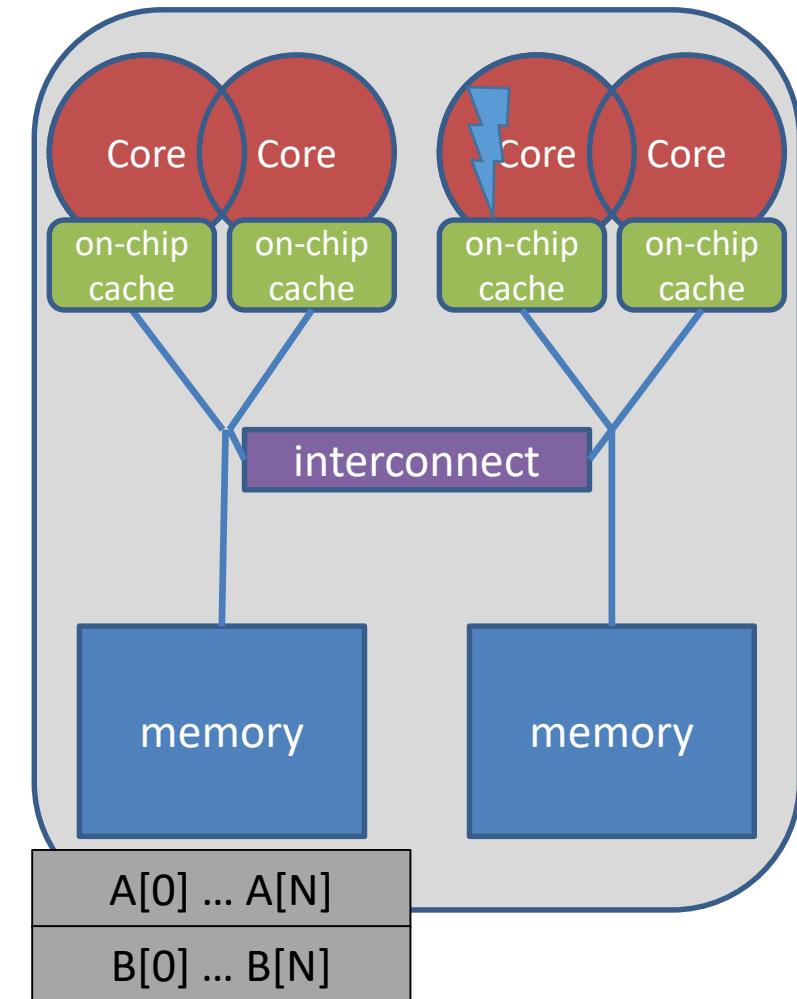
```
void task_affinity() {  
    double* B;  
#pragma omp task shared(B)  
    {  
        B = init_B_and_important_computation(A);  
    }  
#pragma omp task firstprivate(B)  
    {  
        important_computation_too(B);  
    }  
#pragma omp taskwait  
}
```



# Task-to-data Affinity

- OpenMP API version 5.0 supports affinity hints for OpenMP tasks

```
void task_affinity() {  
    double* B;  
#pragma omp task shared(B) affinity(A[0:N])  
    {  
        B = init_B_and_important_computation(A);  
    }  
#pragma omp task firstprivate(B) affinity(B[0:N])  
    {  
        important_computation_too(B);  
    }  
#pragma omp taskwait  
}
```



# Task Reductions

- Task reductions extend traditional reductions to arbitrary task graphs
- Extend the existing task and taskgroup constructs
- Also work with the taskloop construct

```
int res = 0;
node_t* node = NULL;
...
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp taskgroup task_reduction(+: res)
        {
            while (node) {
                #pragma omp task in_reduction(+: res) \
                    firstprivate(node)
                {
                    res += node->value;
                }
                node = node->next;
            }
        }
    }
}
```

# New Task Dependencies

```
int x = 0, y = 0, res = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(out: res) //T0
    res = 0;

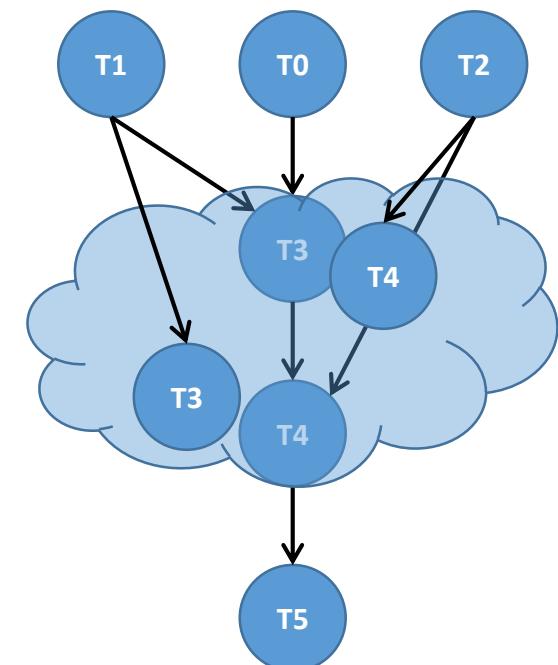
    #pragma omp task depend(out: x) //T1
    long_computation(x);

    #pragma omp task depend(out: y) //T2
    short_computation(y);

    #pragma omp task depend(in: x) depend(in:res) //T3
    res += x;

    #pragma omp task depend(in: y) depend(in:res) //T4
    res += y;

    #pragma omp task depend(in: res) //T5
    std::cout << res << std::endl;
}
```



# Asynchronous API Interaction

- Some APIs are based on asynchronous operations
  - MPI asynchronous send and receive
  - Asynchronous I/O
  - CUDA and OpenCL stream-based offloading
  - In general: any other API/model that executes asynchronously with OpenMP (tasks)
- Example: CUDA memory transfers

```
do_something();
cudaMemcpyAsync(dst, src, nbytes, cudaMemcpyDeviceToHost, stream);
do_something_else();
cudaStreamSynchronize(stream);
do_other_important_stuff(dst);
```

- Programmers need a mechanism to marry asynchronous APIs with the parallel task model of OpenMP
  - How to synchronize completion events with task execution?

# Use OpenMP Tasks

```
void cuda_example() {  
    #pragma omp task      // task A  
    {  
        do_something();  
        cudaMemcpyAsync(dst, src, nbytes, cudaMemcpyDeviceToHost, stream);  
    }  
    #pragma omp task // task B  
    {  
        do_something_else();  
    }  
    #pragma omp task // task C  
    {  
        cudaStreamSynchronize(stream);  
        do_other_important_stuff(dst);  
    }  
}
```

Race condition between the tasks A & C,  
task C may start execution before  
task A enqueues memory transfer.



# Detaching Tasks

```
omp_event_t *event;  
void detach_example() {  
#pragma omp task detach(event)  
{  
    important_code();  
}①  
#pragma omp taskwait ② ④  
}
```

Some other thread/task:

```
omp_fulfill_event(event); ③
```

1. Task detaches
2. **taskwait** construct cannot complete
3. Signal event for completion
4. Task completes and **taskwait** can continue

# Detachable Tasks and Asynchronous APIs

```
void CUDART_CB callback(cudaStream_t stream, cudaError_t status, void *cb_dat) {  
    ③omp_fulfill_event((omp_event_t *) cb_data);  
}  
  
void cuda_example() {  
    omp_event_t *cuda_event;  
#pragma omp task detach(cuda_event) // task A  
    {  
        do_something();  
        cudaMemcpyAsync(dst, src, nbytes, cudaMemcpyDeviceToHost, stream);  
        cudaStreamAddCallback(stream, callback, cuda_event, 0);  
    } ①  
#pragma omp task // task B  
    do_something_else();  
  
#pragma omp taskwait ② ④  
#pragma omp task // task C  
    {  
        do_other_important_stuff(dst);  
    } }
```



1. Task A detaches
2. taskwait does not continue
3. When memory transfer completes, callback is invoked to signal the event for task completion
4. taskwait continues, task C executes

# Detachable Tasks and Asynchronous APIs

```
void CUDART_CB callback(cudaStream_t stream, cudaError_t status, void *cb_dat) {  
    ②omp_fulfill_event((omp_event_t *) cb_data);  
}  
  
void cuda_example() {  
    omp_event_t *cuda_event;  
#pragma omp task depend(out:dst) detach(cuda_event) // task A  
    {  
        do_something();  
        cudaMemcpyAsync(dst, src, nbytes, cudaMemcpyDeviceToHost, stream);  
        ①cudaStreamAddCallback(stream, callback, cuda_event, 0);  
    }  
#pragma omp task // task B  
    do_something_else();  
  
#pragma omp task depend(in:dst) ③ // task C  
    {  
        do_other_important_stuff(dst);  
    } }
```



1. Task A detaches and task C will not execute because of its unfulfilled dependency on A
2. The Memory transfer completes and the callback is invoked to signal the event for task completion
3. Task A completes and C's dependency is fulfilled

# Memory Allocators

- New clause on all constructs with data sharing clauses:

- `allocate( [allocator:] list )`

- Allocation:

- `omp_alloc(size_t size, omp_allocator_t *allocator)`

- Deallocation:

- `omp_free(void *ptr, const omp_allocator_t *allocator)`
  - allocator argument is optional

- `allocate` directive

- Standalone directive for allocation, or declaration of allocation stmt.

# Example: Using Memory Allocators

```
void allocator_example(omp_allocator_t *my_allocator) {
    int a[M], b[N], c;
    #pragma omp allocate(a) allocator(omp_high_bw_mem_alloc)
    #pragma omp allocate(b) // controlled by OMP_ALLOCATOR and/or omp_set_default_allocator
    double *p = (double *) omp_alloc(N*M*sizeof(*p), my_allocator);

    #pragma omp parallel private(a) allocate(my_allocator:a)
    {
        some_parallel_code();
    }

    #pragma omp target firstprivate(c) allocate(omp_const_mem_alloc:c) // on target; must be compile-time expr
    {
        #pragma omp parallel private(a) allocate(omp_high_bw_mem_alloc:a)
        {
            some_other_parallel_code();
        }
    }

    omp_free(p);
}
```

# Partitioning Memory

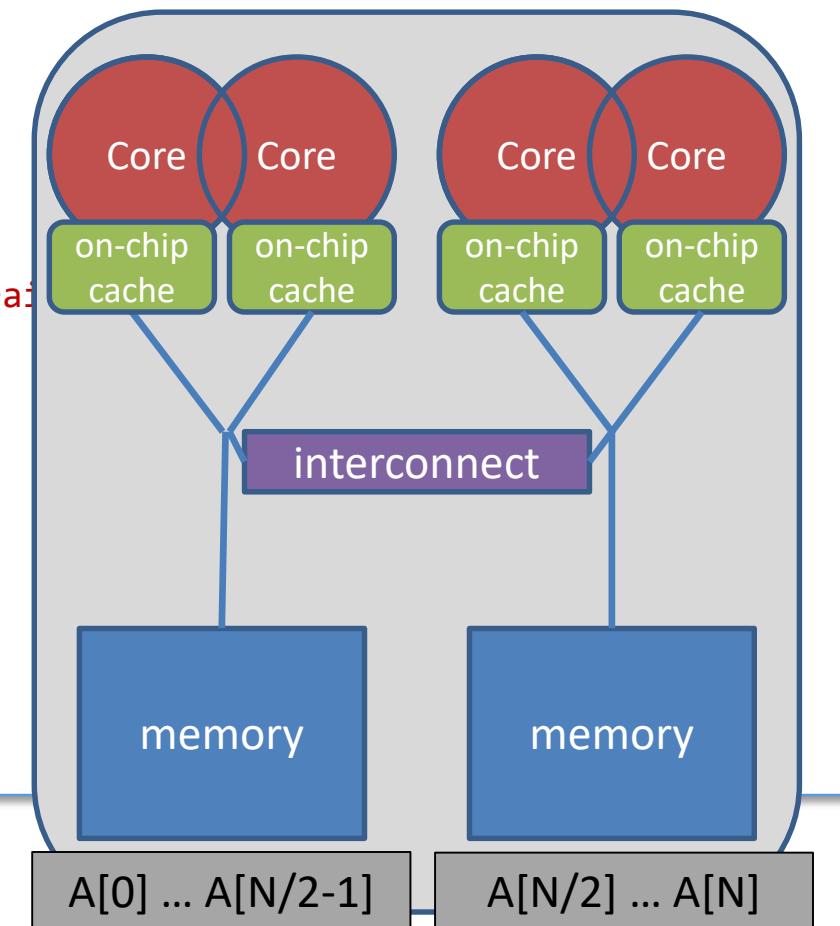
```
void allocator_example() {
    double *array;

    omp_allocator_t *allocator;
    omp_allocator_traits_t traits[] = {
        {OMP_ATK_PARTITION, OMP_ATV_BLOCKED}
    };
    int ntraits = sizeof(traits) / sizeof(*traits);
    allocator = omp_init_allocator(omp_default_mem_space, ntraits, traits);

    array = omp_alloc(sizeof(*array) * N, allocator);

#pragma omp parallel for proc_bind(spread)
    for (int i = 0; i < N; ++i) {
        important_computation(&array[i]);
    }

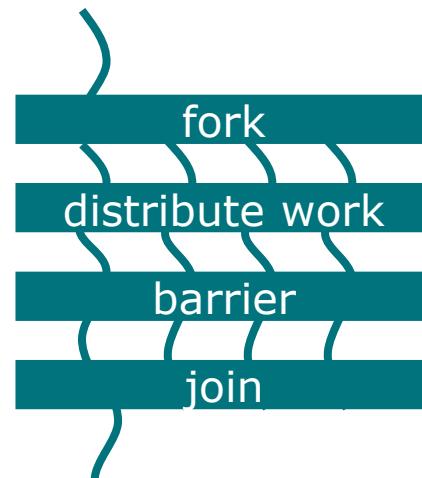
    omp_free(array);
}
```



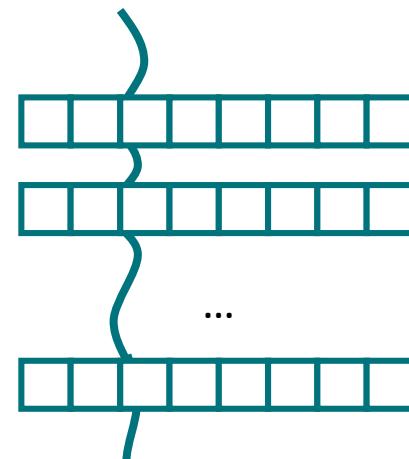
# loop Construct

- Existing loop constructs are tightly bound to execution model:

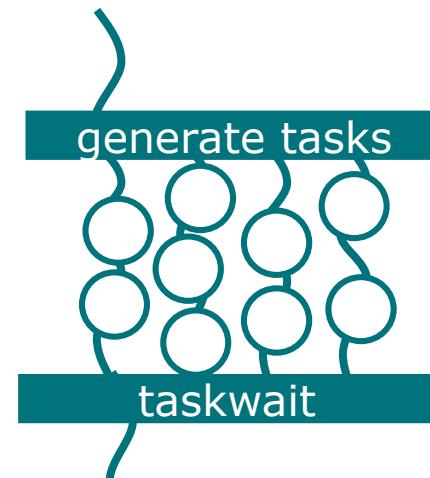
```
#pragma omp parallel for  
for (i=0; i<N;++i) {...}
```



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



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



- The loop construct is meant to let the OpenMP implementation pick the right parallelization scheme for a parallel loop.

# OpenMP 4.5 Multi-level Device Parallelism

```
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 target map(to:x[0:n]) map(tofrom:y)
{
#pragma omp teams distribute parallel for \
    num_teams(num_blocks) num_threads(bsize)
    for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
    }
}
}
```

# Simplifying Multi-level Device Parallelism

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

# Device Profiles – Unified Memory Architecture

- Heterogeneous programming requires map clauses to transfer (ownership of) data to target devices
- Not needed for unified memory devices (unless for optimization)

```
typedef struct mypoints {  
    struct myvec * x;  
    struct myvec scratch;  
    double useless_data[500000];  
} mypoints_t;  
  
#pragma omp requires unified_shared_memory  
  
mypoints_t p = new_mypoints_t();  
  
#pragma omp target // no map clauses needed  
{  
    do_something_with_p(&p);  
}
```

# 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
```

# Declare Variant

- OpenMP 4.5 can create multiple versions of functions

```
#pragma omp declare target
#pragma omp declare simd simdlen(4)
int important_stuff(int x)
{
    // code of the function 'important_stuff'
}
```

- `declare variant` injects user-defined versions of functions

```
#pragma omp declare variant( int important_stuff(int x) ) \
    match( context={target,simd} device={arch(nvptx)} )
int important_stuff_nvidia(int x)
{ /* Specialized code for NVIDIA target */ }

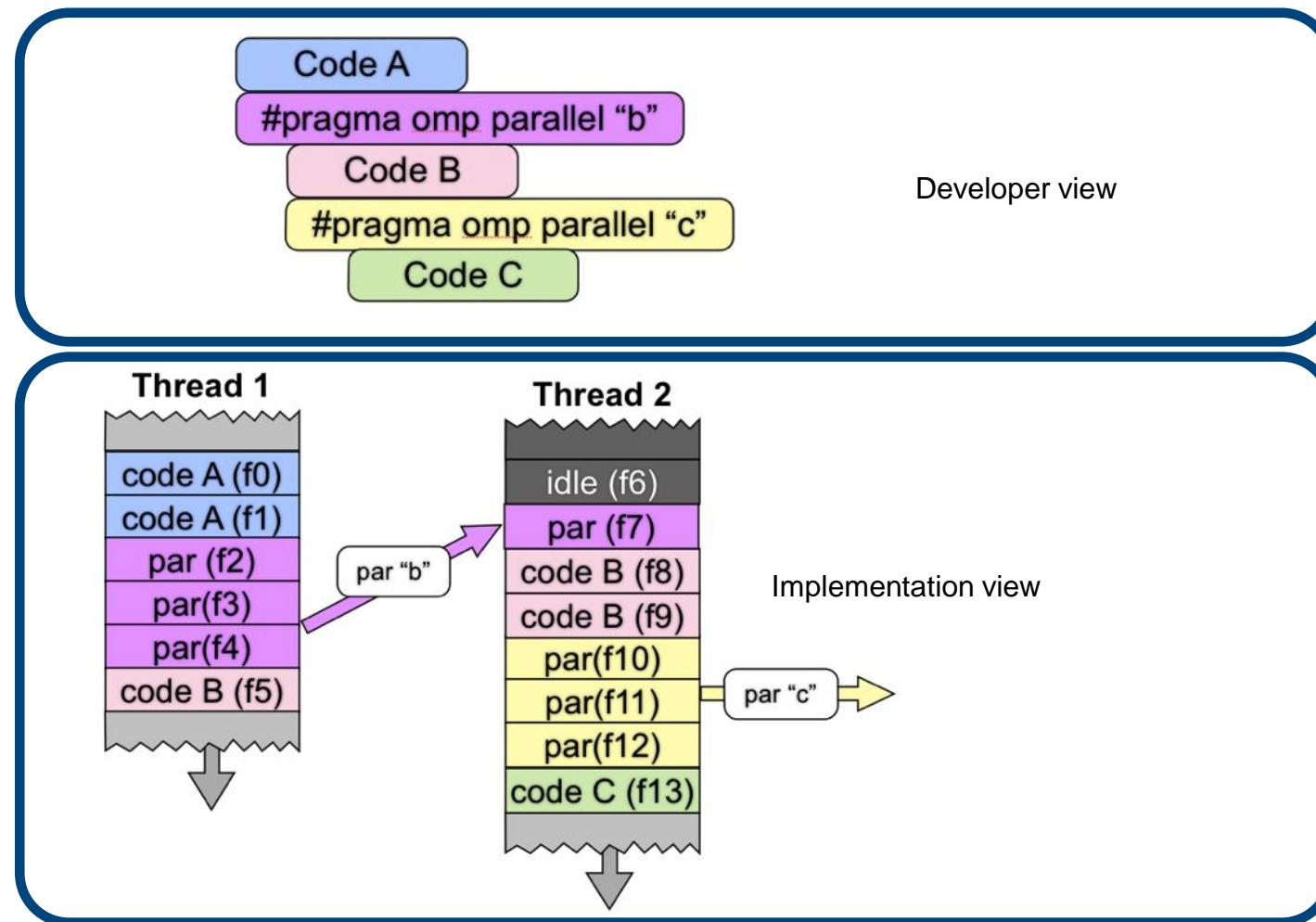
#pragma omp declare variant(int foo(int x)) \
    match( context={target, simd(simdlen(4))}, device={isa(avx2)} )
__m256i _mm256_epi32_important_stuff(__m256i x);
{ /* Specialized code for simd loop called on an AVX2 processor */ }
```

# Declare Variant – Execution Context

- Context describes lexical “scope” of an OpenMP construct and it’s lexical nesting in other OpenMP constructs:

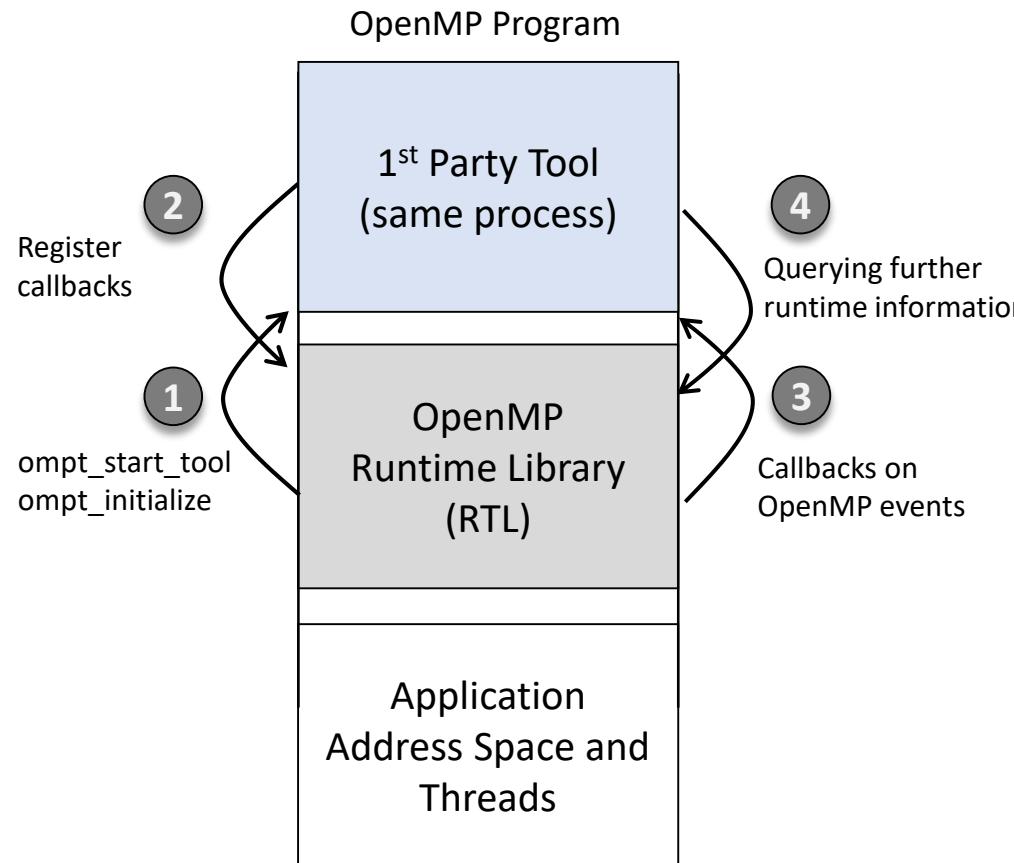
```
// context = {}  
#pragma omp target teams  
{  
    // context = {target, teams}  
    #pragma omp parallel  
    {  
        // context = {target, teams, parallel}  
        #pragma omp simd aligned(a:64)  
        for (...)  
        {  
            // context = {target, teams, parallel, simd(aligned(a:64), simdlen(8), notinbranch) }  
            foo(a);  
        }  
    }  
}
```

# OpenMP Tools Interfaces



# OpenMP Tools Interface (OMPT)

- Provide interfaces for first-party tools to attach to the OpenMP implementation for performance analysis and correctness checking

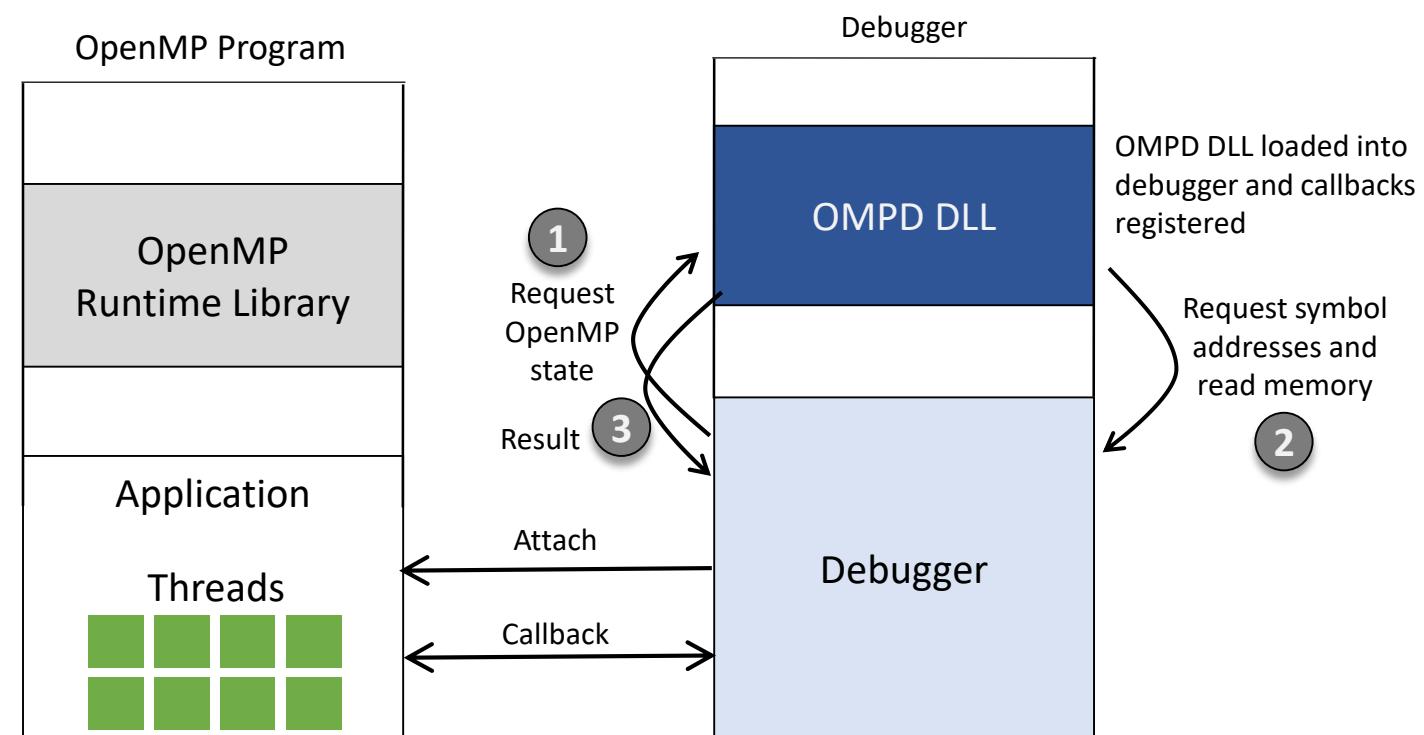


## Examples for OMPT callbacks for events

- `ompt_callback_parallel_begin`
- `ompt_callback_parallel_end`
- `ompt_callback_implicit_task`
- `ompt_callback_task_create`
- `ompt_callback_dependences`
- ...

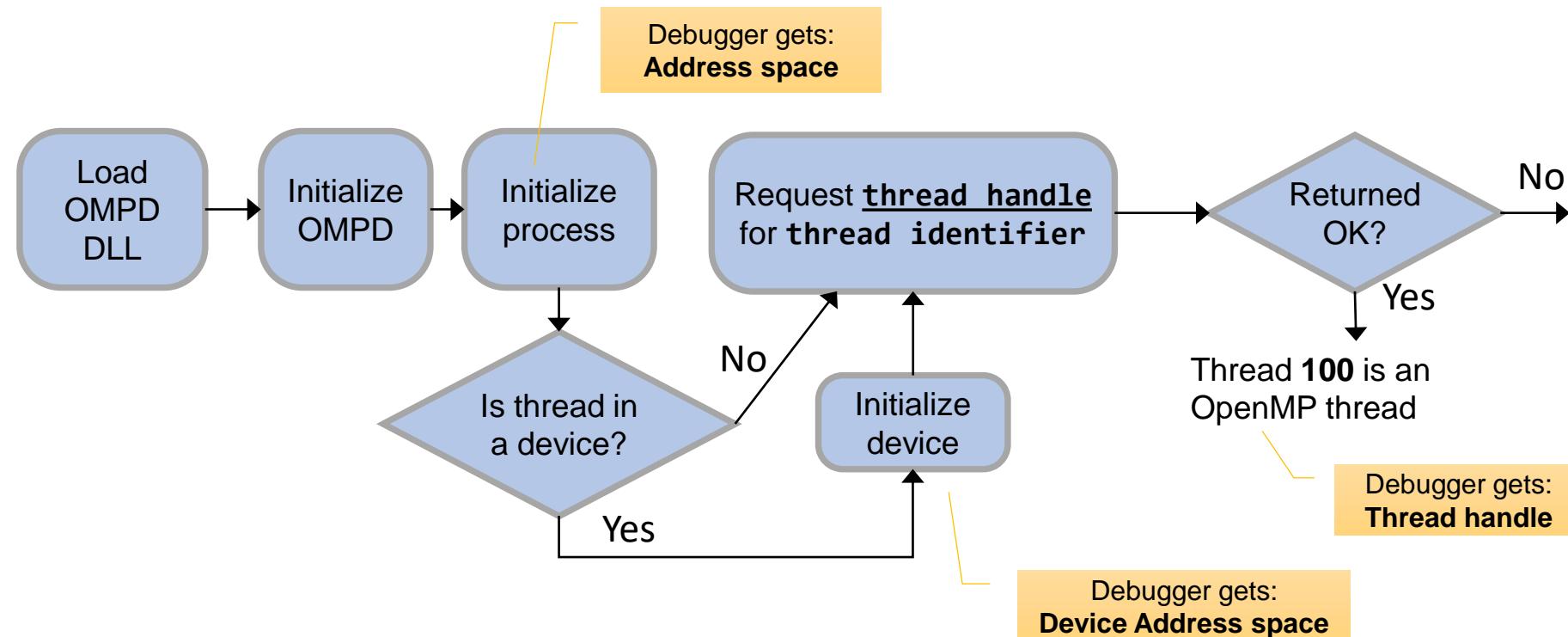
# OpenMP Debugging Interface (OMPDI)

- Provide interfaces for third-party tools to inspect the current state of OpenMP execution.



# OpenMP Debugging Interface (OMPDI)

- Segmentation fault occurs in thread 100; is this an OpenMP thread?



# The Last Slide

- OpenMP 5.0 will be a major leap forward

- To be released at SC18
  - Well-defined interfaces for tools
  - New ways to express parallelism

- OpenMP is a modern directive-based programming model

- Multi-level parallelism supported (coprocessors, threads, SIMD)
  - Task-based programming model is the modern approach to parallelism
  - Powerful language features for complex algorithms
  - High-level access to parallelism; path forward to highly efficient programming



Visit [www.openmp.org](http://www.openmp.org) for more information