# OpenMP Current Status and Future Directions

Yun (Helen) He, Michael Klemm, Bronis R. De Supinski

# Architecture Review Board

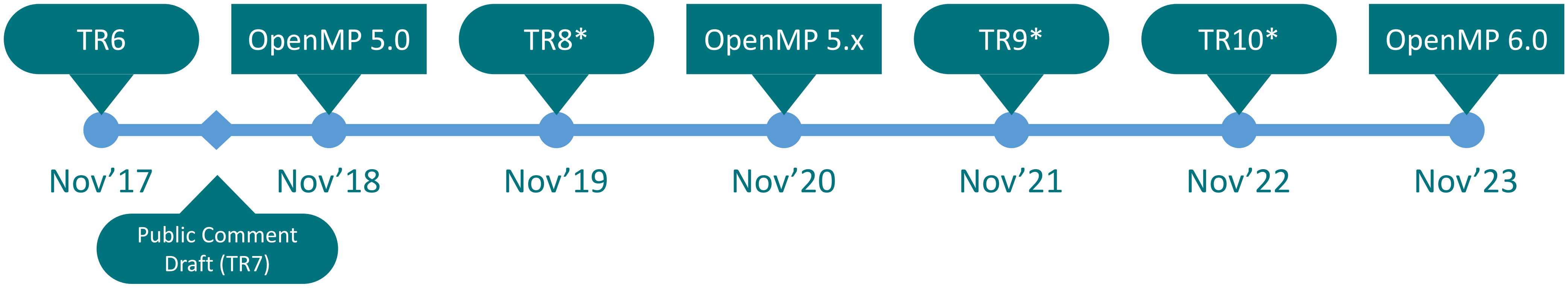
- The mission of the OpenMP ARB (Architecture Review Board) is to standardize directive-based multi-language high-level parallelism that is performant, productive and portable.
- 32 members currently.  More in the work to join.
- Please consider joining us too so you can also contribute!

# OpenMP Programming Model

- 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
- Using the hybrid MPI/OpenMP programming model is one of the main choices
  - for running scientific applications on many hardware architectures such as Intel Xeon, Xeon Phi, and Nvidia GPUs.

# OpenMP Roadmap

- OpenMP has a well-defined roadmap:
  - Last officially released versions: 4.0 (July 2013), 4.5 (Nov 2015)
  - 5-year cadence for major releases
  - One minor release in between
  - (At least) one Technical Report (TR) with feature previews in every year
  - Current release version is 4.5




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

# Current Status
## (OpenMP 4.5 and Earlier)

# Versions 4.0 and 4.5

- OpenMP has been significantly modernized since the OpenMP 4.0 (July 2013) and OpenMP 4.5 (Nov 2015) specification releases.
- Major additions include: SIMD, task dependencies, task groups, thread affinity, user defined reductions, taskloop, doacross.
- Target device support was first introduced in OpenMP 4.0 and was the focus for enhancement for OpenMP 4.5.

SIMD | Target Device Support | Task Groups | Task Dependencies

Thread Affinity | User Defined Reductions | Taskloop | Task Priority

doacross | Hint for locks and critical | Fortran 2003 Support

# OpenMP 4.0 Major Additions

- Device constructs
- SIMD constructs
- Cancellation
- Task dependences and task groups
- Thread affinity control
- User-defined reductions
- Initial support for Fortran 2003
- Support for array sections (including in C and C++)
- Sequentially consistent atomics
- Display of initial OpenMP internal control variables

# OpenMP 4.5 Focused on Device Support

- Unstructured data mapping
- Asynchronous execution
- Scalar variables are firstprivate by default
- Improvements for C/C++ array sections
- Device runtime routines: allocation, copy, etc.
- Clauses to support device pointers
- Ability to map structure elements
- New combined constructs
- New way to map global variables (`link`)

# OpenMP 4.5 Other New Features

- Many clarifications and minor enhancements
  - → SIMD extensions
  - → Addition of schedule modifiers: `simd`, `monotonic`, `nonmonotonic`
  - → Clarifications of thread affinity policies
  - → Grammar for `OMP_PLACES`
  - → Support for `if` clause on combined/composite constructs
  - → Reductions for C/C++ arrays
  - → Runtime routines to support affinity
- Support for doacross loops
- Divide loop into tasks with taskloop construct
- Hints for locks and `critical` sections
- Continues to increase Fortran 2003 support
- Task priorities
- Improved support for C++ reference types
- New terms to simplify discussion of new features

# Vectorization Before OpenMP 4.0

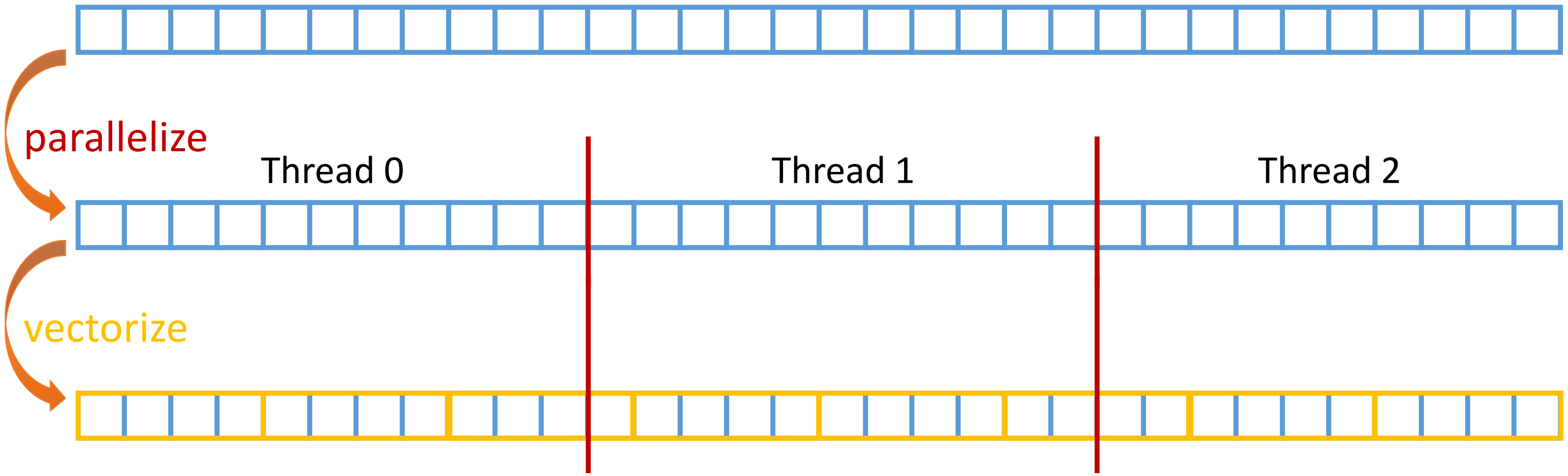
- Programmers had to rely on auto-vectorization…
- … or to use vendor-specific extensions
  - Programming models (e.g., Intel® Cilk™ Plus)
  - Compiler pragmas (e.g., `#pragma vector`)
  - Low-level constructs (e.g., `_mm_add_pd()`)

```
#pragma omp parallel for
#pragma vector always
#pragma ivdep
for (int i = 0; i < N; i++) {
    a[i] = b[i] + ...;
}
```

You need to trust your compiler to do the "right" thing.

# SIMD Version of Scalar Product

```
void sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp for simd reduction(+:sum)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```



# SIMD Function Vectorization

```
#pragma omp declare simd
float min(float a, float b) {
    return a < b ? a : b;
}

#pragma omp declare simd
float distsq(float x, float y) {
    return (x - y) * (x - y);
}

void example() {
#pragma omp parallel for simd
    for (i=0; i<N; i++) {
        d[i] = min(distsq(a[i], b[i]
}   }
```

```
_ZGVZN16vv_min(%zmm0, %zmm1):
    vminps %zmm1, %zmm0, %zmm0
    ret
```

```
_ZGVZN16vv_distsq(%zmm0, %zmm1):
    vsubps %zmm0, %zmm1, %zmm2
    vmulps %zmm2, %zmm2, %zmm0
    ret
```

```
vmovups (%r14,%r12,4), %zmm0
vmovups (%r13,%r12,4), %zmm1
call _ZGVZN16vv_distsq
vmovups (%rbx,%r12,4), %zmm1
call _ZGVZN16vv_min
```

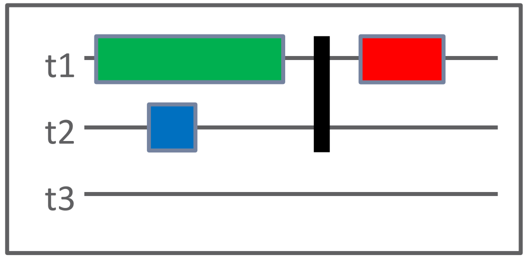
# Thread Affinity Control

- OpenMP 4.0 added **OMP_PLACES** environment variable to control thread allocation
  - Can be threads, cores, sockets, or a list with explicit CPU ids.
- **OMP_PROC_BIND** controls thread affinity within and between OpenMP places
  - OpenMP 3.1 only allows TRUE or FALSE.
  - OpenMP 4.0 still allows the above. Added options: close, spread, master.

# Task Synchronization w/ Dependencies

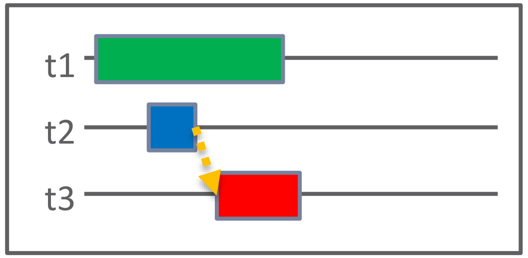
**OpenMP 3.1**

```
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task
  std::cout << x << std::endl;

  #pragma omp task
  long_running_task();

  #pragma omp taskwait

  #pragma omp task
  x++;
}
```




**OpenMP 4.0**

```
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(in: x)
  std::cout << x << std::endl;

  #pragma omp task
  long_running_task();



  #pragma omp task depend(inout: x)
  x++;
}
```



# taskloop Example: saxpy Operation

```
for (i = 0; i<SIZE; i+=1) {
   A[i]=A[i]*B[i]*S;
}
```

blocking

```
for (i = 0; i<SIZE; i+=TS) {
   UB = SIZE < (i+TS) ? SIZE : i+TS;
   for (ii=i; ii<UB; ii++) {
      A[ii]=A[ii]*B[ii]*S;
   }
}
```

```
for (i = 0; i<SIZE; i+=TS) {
   UB = SIZE < (i+TS) ? SIZE : i+TS;
   #pragma omp task private(ii) \
      firstprivate(i,UB) shared(S,A,B)
   for (ii=i; ii<UB; ii++) {
      A[ii]=A[ii]*B[ii]*S;
   }
}
```

taskloop

```
#pragma omp taskloop grainsize(TS)
for (i = 0; i<SIZE; i+=1) {
   A[i]=A[i]*B[i]*S;
}
```

- Manual transformation is cumbersome and error prone
- Applying blocking techniques for large loops can be tricky
- `taskloop`: improved programmability

# Parallelizing doacross Loop

- Help with cross-iteration dependencies
- Use “ordered” clause to ensure structured blocks are executed on lexical order

2 loops contribute to the pattern of dependencies … so the dependency relations for each depend(sink) is of length 2

Threads wait here until x[r-1][c] and x[r][c-1] have been released

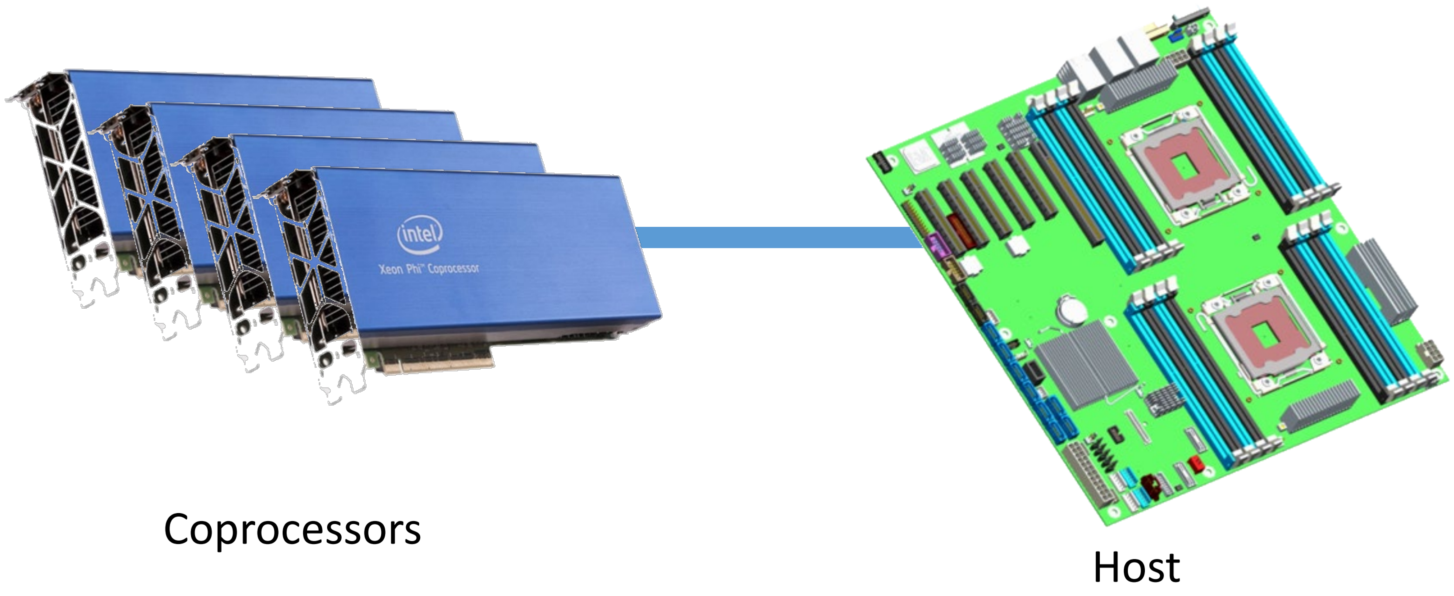
```
#pragma omp for ordered(2) collapse(2)
    for (r=1; r<N; r++) {
        for (c=1; c<N; c++) {
            // other parallel work ...
            #pragma omp ordered depend(sink:r-1,c) \
                                depend(sink:r,c-1)
                x[r][c] += fn(x[r-1][c], x[r][c-1]);
            #pragma omp ordered depend(source)
        }
    }
```

x[r][c] is complete and released for use by other threads

Example courtesy of Tim Mattson

# Device Model

- OpenMP 4.0 supports accelerators/coprocessors
- Device model:
  - One host
  - Multiple accelerators/coprocessors of the same kind



OpenMP

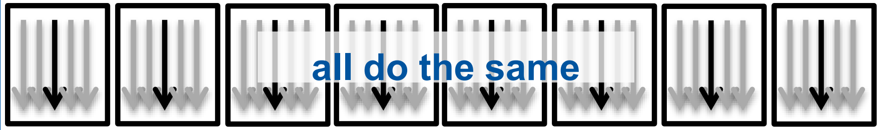
# Example

host | target | host | target | host

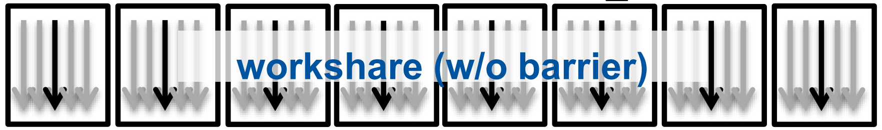
```
#pragma omp target data device(0) map(alloc:tmp[:N]) map(to:input[:N)) map(from:res)
  {
#pragma omp target device(0)
#pragma omp parallel for
    for (i=0; i<N; i++)
      tmp[i] = some_computation(input[i], i);

    update_input_array_on_the_host(input);

#pragma omp target update device(0) to(input[:N])

#pragma omp target device(0)
#pragma omp parallel for reduction(+:res)
    for (i=0; i<N; i++)
      res += final_computation(input[i], tmp[i], i)
  }
```

# Multi-level Device Parallelism

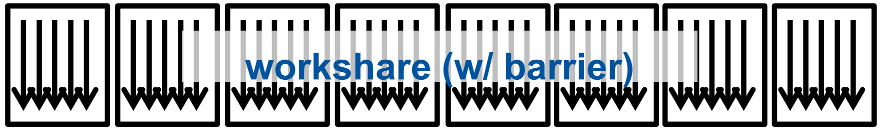
```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Define scalars n, a, b & initialize x, y

#pragma omp target data map(to:x[0:n])
{
#pragma omp target map(tofrom:y)
#pragma omp teams num_teams(num_blocks) num_threads(bsize)
```




```
#pragma omp distribute
  for (int i = 0; i < n; i += num_blocks){
```




```
#pragma omp parallel for
    for (int j = i; j < i + num_blocks; j++) {
```




```
        y[j] = a*x[j] + y[j];
} } } }
```

# Device Parallelism: Combined Constructs

```
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];
    }
  }
}
```

# Future Directions

(OpenMP 5.0 and Beyond)

# Version 5.0 is on its Way (Release @ SC18)

- OpenMP 5.0 will introduce new powerful features to improve programmability

Task Reductions

Memory Allocators

Detachable Tasks

C++14 and C++17 support

Dependence Objects

Tools APIs: OMPD,OMPT

`loop` Construct

Fortran 2008 support

Unified Shared Memory

Collapse non-rect. Loops

Multi-level Parallelism

Task-to-data Affinity

Data Serialization for Offload

Parallel Scan

Meta-directives

Display Affinity

“Reverse Offloading”

User Defined Function Variants

Improved Task Dependences

# TR4 was released in November 2016

- Included 24 passed tickets
- Major new feature was performance tool support (TR2+)
- Some significant extensions to existing functionality
  - →Support for task reductions, including on `taskloop` construct
  - →Implicit `declare target` directives and other verbosity reducing changes
- Many clarifications and minor enhancements, including:
  - →Use of any C/C++ *lvalue* in `depend` clauses
  - →Addition of `depend` clause to `taskwait` construct
  - →Addition of `conditional` modifier to `lastprivate` clause
  - →Permits `declare target` on C++ classes with virtual members
  - →Clarification of `declare target` C++ initializations

# TR6 was released in November 2017

- Includes 88 tickets beyond those in TR4 (112 tickets total)
- Many major additions and significant enhancements
  - → Adds memory allocators to support complex memory hierarchies
  - → User defined mappers provide deep copy support for map clauses
  - → Supports better control of device usage and specialization for devices
    - → Can require unified shared memory
    - → Can use functions specialized for a type of device
  - → Adds `concurrent` construct to support compiler optimization
  - → Adds support to display runtime thread affinity
  - → Support for third-party (debugging) tools
  - → Adds C11, C++11 and C++14 as normative base languages
  - → Expands task dependency mechanism for greater flexibility and control
  - → Release/acquire semantics added to memory model
  - → Supports collapse of imperfectly nested loops
  - → Support for != on C/C++ loops
- Many clarifications and other minor enhancements

# TR7 was released in July 2018

OpenMP

- Includes 131 tickets beyond those in TR6 (243 tickets total)
- Many major additions and significant enhancements
  - → Support for metadirectives and function variants
  - → Device refinements including reverse offload
  - → Revises `concurrent` construct to be `loop` construct
  - → Allows `teams` construct outside of `target` (i.e., on host)
  - → Supports task affinity, `task` modifier on reductions on other constructs, depend objects and detachable tasks
  - → Adds C++17 and Fortran 2008 as normative base languages, completes Fortran 2003
  - → Supports request to quiesce OpenMP threads
  - → Supports collapse of non-rectangular loops
  - → Adds scan operations (similar to reductions)
  - → Expands and refines memory allocator support
  - → Extensions and refinements of deep copy support
  - → Supports C/C++ array shaping
- Many clarifications and other minor enhancements

# Task Reductions

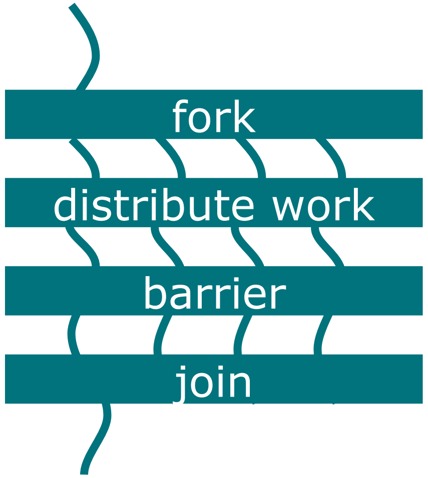
- Task reductions extend traditional reductions to arbitrary task graphs
- Extend the existing task and taskgroup constructs
- Also work with the taskloop construct

```
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;
         }
      }
   }
}
```

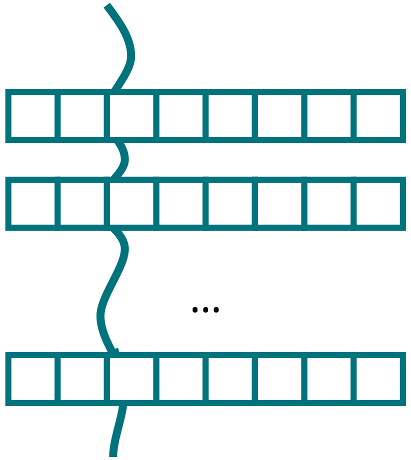
# Existing Parallel Loop Constructs

- Existing parallel loop constructs are tightly bound to execution model:

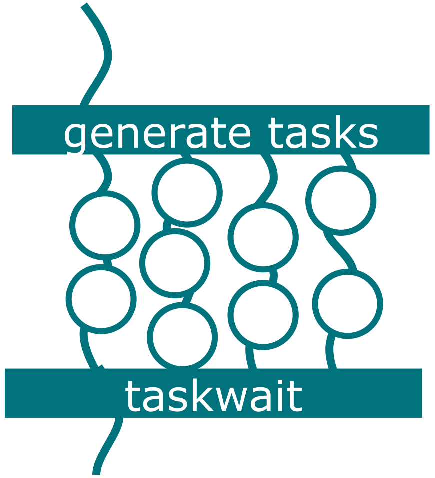
```
#pragma omp for
for (i=0; i<N;++i) {…}
```




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




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



# The new `loop` Construct

- The `loop` construct asserts to the compiler that the iterations of a loop are free of dependencies and may be run concurrently in any order.
  - Each iteration execute exactly once.
- It is meant to let the OpenMP implementation choose the right parallelization scheme.
  - Can be used on both host and device.

```
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];
    }
  }
}
```

# Display Thread Affinity at Runtime

- Getting the optimal process and thread affinity is critical to ensuring optimal performance and is an essential step before starting any code optimization attempts.
- Automatic display of affinity when **OMP_DISPLAY_AFFINITY** environment variable is set to **TRUE**.
- The format of the output can be customized by setting the **OMP_AFFINITY_FORMAT** environment variable to an appropriate string or use the runtime set/get routines
- Flexible runtime API calls omp_display_affinity() or omp_capture_affinity() to display or capture thread affinity info at selected locations within code.
- Sample **OMP_AFFINITY_FORMAT= "thrd_level= %L, parent_id= %A, thrd_id= %T, thrd_affinity= %A"**
- Sample output
  - →thrd_level= 1, parent_thrd= 0,thrd_id= 0, thrd_affinity= 0,2,4,6
  - →thrd_level= 1, parent_thrd= 0,thrd_id= 1, thrd_affinity= 1,3,5,7

# Memory Allocators

OpenMP

| Allocator name | Storage selection intent |
|---|---|
| omp_default_mem_alloc | use default storage |
| omp_large_cap_mem_alloc | use storage with large capacity |
| omp_const_mem_alloc | use storage optimized for read-only variables |
| omp_high_bw_mem_alloc | use storage with high bandwidth |
| omp_low_lat_mem_alloc | use storage with low latency |
| omp_cgroup_mem_alloc | use storage close to all threads in the contention group of the thread requesting the allocation |
| omp_pteam_mem_alloc | use storage that is close to all threads in the same parallel region of the thread requesting the allocation |
| omp_thread_local_mem_alloc | use storage that is close to the thread requesting the allocation |

# 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);
}
```

# Requires Unified Shared Memory

- Single address space over CPU and GPU memories
- Data migrated between CPU and GPU memories transparently to the application - no need to explicitly copy data

```
// No data directive needed.
#pragma omp requires unified_shared_memory
for (k=0; k < NTIMES; k++)
{
#pragma omp target teams distribute parallel for
  for (j=0; j<ARRAY_SIZE; j++) {
    a[j] = b[j] + scalar * c[j];
  }
}}
```

# Fortran 2003 Support in OpenMP

- OpenMP 4.0 added Fortran 2003 to list of base language versions
- OpenMP 4.5 has a list of unsupported Fortran 2003 features
  - →List initially included 24 items (some big, some small)
  - →List has been reduced to 10 items
  - →List in specification reflects approximate OpenMP 5.0 priority
  - →Priorities determined by importance and difficulty
- OpenMP 5.0 will fully support Fortran 2003

# Fortran 2008 Support in OpenMP

- OpenMP 5.0 will add Fortran 2008 (along with C11, C++11, C++14, and C++17) as normative references
- OpenMP 5.0 (see released TR7 specifications) has a list of unsupported Fortran 2008 features
- OpenMP 5.1 will work through the list to add more support. Some top priority features to consider are:
  - →DO CONCURRENT
  - →Coarrays
  - →Submodules

# Some Potential Topics for OpenMP 5.1 or 6.0

- Deeper support for descriptive and prescriptive control
- More support for memory affinity and complex hierarchies
- Support for pipelining, other computation/data associations
- Continued refinements and improvements to device support
- Unshackled threads
- Event-driven parallelism
- Completing support for new normative references
- Fortran: support assumed-type (type(*))

# Resources

**http://www.openmp.org**

- Lots of information available at ARB's website
  - Specifications, technical reports, **summary cards**
  - Compilers and Tools
  - Tutorials, presentations, and publications
- OpenMP Book
- OpenMP Events
  - Supercomputing Conference
  - OpenMPCon Workshop
  - IWOMP Workshop
  - UK OpenMP Users' Conference


# SC18 Tutorials and BoF

- Enjoy a promo video about OpenMP history and SC18 tutorials !
  - https://www.youtube.com/watch?v=sncF6s7xym4

- Tutorial: OpenMP Common Core: A “Hands-On” Exploration
  - Tim Mattson, Alice Koniges. Yun (Helen) He, David Eder
- Tutorial: Mastering Tasking with OpenMP
  - Michael Klemm, Sergi Mateo, Christian Terboven, Xavier Teruel, Bronis de Supinski
- Tutorial: Advanced OpenMP: Performance and 5.0 Features
  - James Beyer, Michael Klemm, Kelvin Li, Christian Terboven, Bronis de Supinski, Ruud van der Pas
- Tutorial: Programming Your GPU with OpenMP: A Hands-On Introduction
  - Simon McIntosh-Smith, Tim Mattson

- OpenMP BoF

# About OpenMP History and SC18 Tutorials