Introduction to OpenMP Tasking and Offloading

Georg Hager, Markus Wittmann

based on work by
R. Bader (LRZ), G. Hager (RRZE), V. Weinberg (LRZ),
and R. v. d. Pas, E. Stotzer, C. Terboven: **Using OpenMP – The Next Step**. MIT Press, 2017, ISBN 978-0-262-53478-9

Shared-Memory Parallelization With Tasking

Tasks in OpenMP

- tasks in OpenMP refer to an instance of executable code and associated data environment
- we already used tasks unknowingly, e.g.:
  - internally `parallel` construct creates an **implicit** task of the associated structured block for each thread

- **explicit** tasks allow for greater flexibility
  - parallelize workloads which cannot be mapped to worksharing constructs
  - allow for dependencies between tasks

Creating Tasks

```
task [clauses…]
structured-block
```

```
#pragma omp parallel
{
  #pragma omp single
  {
    for (...) {
      #pragma omp task
      { /* work */ }
    }
  } /* implicit barrier */
}
```

- encountering thread creates a task from associated *structured block*
- task can be executed
  - **undeferred**: executed immediately
  - **deferred**: possibly executed later
- deferred tasks are enqueued to be processed by (waiting) threads
- tasks are executed in unspecified order
- barrier is only left iff
  - all threads have arrived
  - and **all tasks have been processed**

Data Sharing (Attributes) with Tasks

- specify explicitly with clauses:
  - **`default`**, **`private`**, **`shared`**, **`firstprivate`**
- rules (as already known):
  - static/global variables → shared
  - automatic (stack) variables inside region → private
- referenced variables become **`firstprivate`** iff:
  - no default clause present
  - variable not explicitly listed
  - variable not determined shared in enclosing constructs
  - ensures data is still alive when task is executed

```
#pragma omp parallel
#pragma omp single
{
  double d[100] = ...;
  #pragma omp task
  work(d, 100);
}
```

d **`firstprivate`** as determined **`private`** inside **`single`**

```
double d[100] = ...;
#pragma omp parallel
#pragma omp single
{

  #pragma omp task
  work(d, 100);
}
```

d **shared**

Task Clauses

```
if(expression)
```

- `if(true)`:
  - **deferred** task created, possibly executed later
  - the default
- `if(false)`:
  - **undeferred** task is created, executed immediately
  - only applies to task at hand
  - optimization:
    - stop generating tasks if enough have been generated, see `final`
    - reduce overhead
  - all other task semantics still apply

Task Synchronization

- waiting for completion of tasks:
  - explicit **`barrier`**
  - implicit barriers (does not apply for **`nowait`**)
- with explicit task synchronization constructs
  - **`taskwait`**
  - **`taskgroup`** (see later)

- **`taskwait`**: wait until all child tasks of current (implicit) task are completed
  - **NOTE**: child tasks include only direct children, not grandchildren

```
#pragma omp parallel
#pragma omp single
{
  #pragma omp task
  work1();

  #pragma omp taskwait

  #pragma omp task
  work2();
}
```

continue when **`work1`** has finished

wait in impl. barrier until **`work2`** has finished

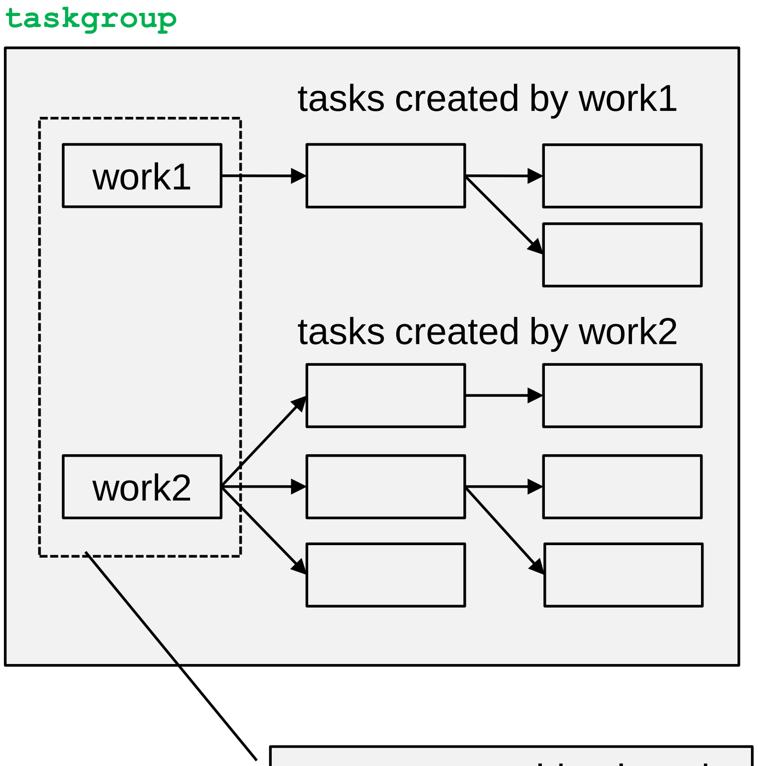
Task Synchronization with `taskgroup`

- wait for **all** tasks created within `taskgroup` region
  - not only the direct children as with `taskwait`

```
#pragma omp parallel
#pragma omp single
{
   #pragma omp taskgroup
   {
       #pragma omp task
       work1() /* spawns more tasks */

       #pragma omp task
       work2() /* spawns more tasks */
   }
}
```

wait here for **all** tasks in `taskgroup` region to finish



`taskwait` would only wait for work1 and work2

Task Synchronization with taskgroup

- allows for dedicated waiting on tasks

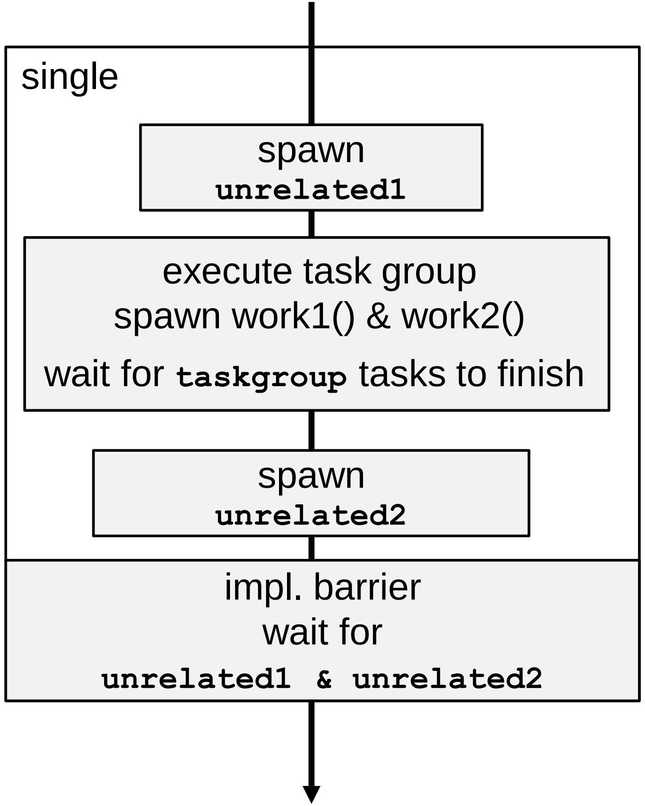
```
#pragma omp parallel
#pragma omp single
{
  #pragma omp task
  unrelated1();

  #pragma omp taskgroup
  {
      #pragma omp task
      work1() /* spawns more tasks */

      #pragma omp task
      work2() /* spawns more tasks */
  } /* wait for tasks */

  #pragma omp task
  unrelated2();
} /* implicit barrier */
```

no waiting for `unrelated1`



Task Scheduling Points

- threads can suspend execution of tasks and switch to another task (*task switch*), and also start new tasks
- **only** at predefined task scheduling points (TSPs):
  - `task` construct
  - end of task
  - at `taskyield` and `taskwait`
  - end of `taskgroup` construct
  - at implicit/explicit barrier
  - (`target` related constructs & API)
- `taskyield` introduces an explicit TSP

task scheduling points

```
#pragma omp parallel
#pragma omp single
{
  double d[100] = ...

  #pragm omp task
  {
    work(d, 100);

    #pragma omp taskyield

    more_work(d, 100)
  }

  #pragma omp taskwait
}
```

uninterrupted * (around `work(d, 100);` and around `more_work(d, 100)`)

- `#pragm omp task` — task construct
- `#pragma omp taskyield` — `taskyield`
- closing `}` of the task — end of task
- `#pragma omp taskwait` — `taskwait`
- closing `}` of the single region — impl. barrier

*assuming in `work()`/`more_work()` no TSPs occur

Task Scheduling Points

- best:
  - do not hold locks when crossing task scheduling points
  - avoid task scheduling points in critical regions

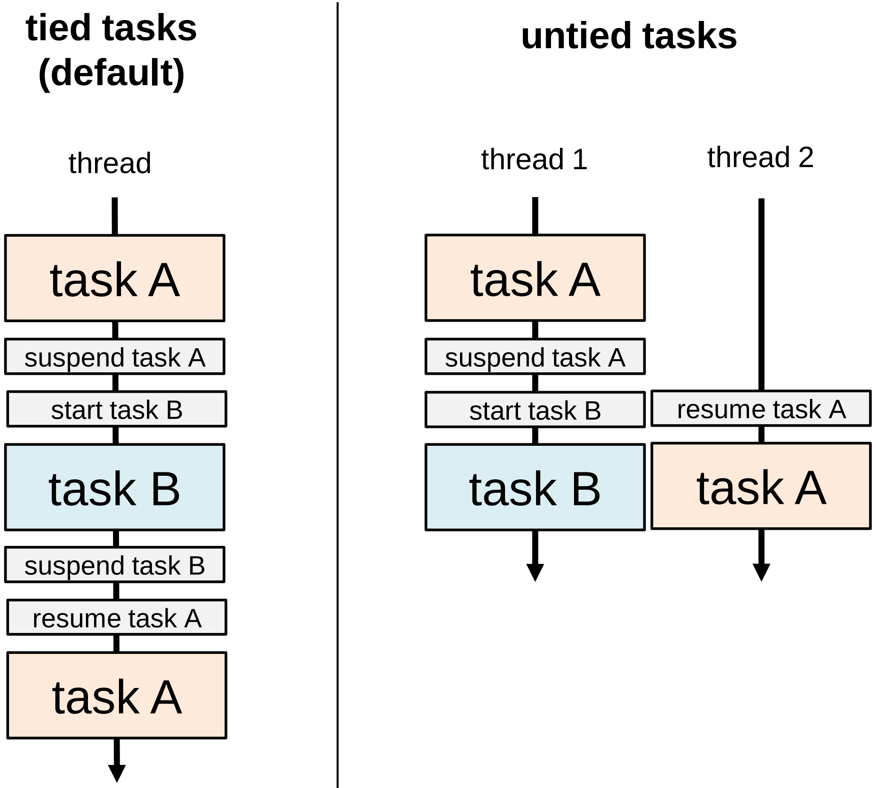
- deadlocks can occur
  - task A holds a lock/is inside a critical region
  - task A is suspended due to reaching a task scheduling point
  - task B is resumed by the same thread
  - task B tries to acquire the lock/enter the critical region
  - deadlock occurs

Tied and Untied Tasks

- tied tasks (default)
  - cannot leave thread that first started execution of task (≠ encountering thread)
- untied tasks
  - can be resumed by any thread in team

NOTE: tied might be desired if cache/NUMA locality is needed

```
#pragma omp task untied
task_a();
#pragma omp task untied
task_b();
```



Reductions with Tasks

≥ v5.0

- requires two components
- `taskgroup` with `task_reduction` clause
- `in_reduction` clause of `task`

```
#pragma omp parallel
#pragma omp single
{
  int sum = 0;
  #pragma omp taskgroup \
              task_reduction(+:sum)
  {
    #pragma omp task in_reduction(+:sum)
    { /* might spawn tasks that also have
           in_reduction(+:sum)  */
    }

    #pragma omp task { }
    /* does not take part */

  } /* implicit barrier */
  /* sum available */
}
```

Task Dependencies

- introduce dependencies between **sibling** tasks
- dependency types:
  - `in`: “read” from variables
  - `out/inout`: “read” from and “write” to variables
  - not covering: **`mutexinoutset`**, **`inoutset`**, **`depobj`**

- task graph is built by matching dependencies to dependencies of already submitted tasks

```
task depend(in:…) \
     depend(out:…) \
     depend(inout:…)
```

list of variables, array elements and sections

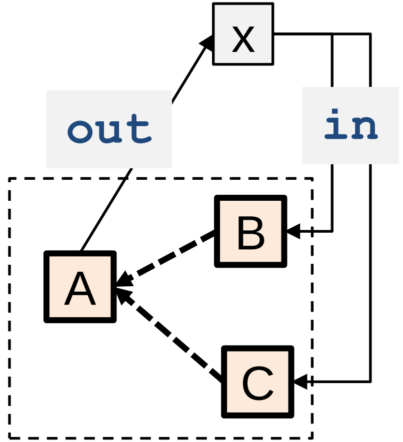
NOTE: tasks do not necessarily have to use the variables specified in dependencies

`in` dependency

- depends on last `out` dependency of the listed variables, if any
- can be scheduled parallel to other tasks with the same `in` dependency
- if no previous `out` dependency to listed variable exists, it is assumed as fulfilled

```
#pragma omp task depend(out:x) /*A*/
/*…*/
#pragma omp task depend(in:x)  /*B*/
/*…*/
#pragma omp task depend(in:x)  /*C*/
/*…*/
```

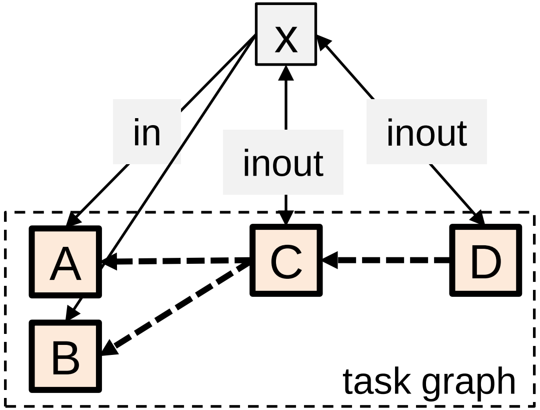
task graph



`out`/`inout` dependency

- depends on
  - last **`out`** dependency of the listed variables, if any
  - all **`in`** dependencies scheduled directly before
- if no previous **`in`/`inout`/`out`** dependency to listed variable exists, it is assumed as fulfilled
- **`out`** and **`inout`** are effectively the same

```
#pragma omp task depend(in:x) /*A*/
/*…*/
#pragma omp task depend(in:x) /*B*/
/*…*/
#pragma omp task depend(inout:x)  /*C*/
/*…*/
#pragma omp task depend(inout:x)  /*D*/
/*…*/
```



Oder of Creation Matters

```
int v = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(out:x)   /*A*/
  v = 1;

  #pragma omp task depend(inout:x) /*B*/
  v += 2;

  #pragma omp task depend(inout:x) /*C*/
  v *= 2;
}
```

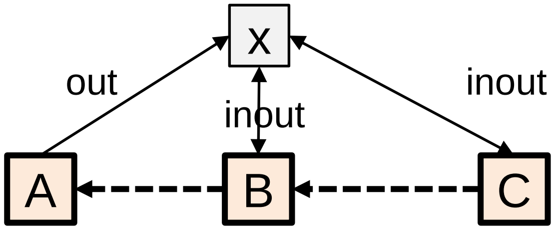
$x = ((1) + 2) * 2 = 6$

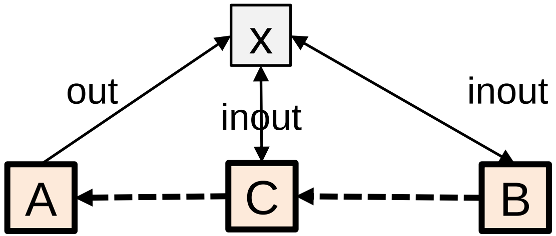
```
int v = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(out:x)   /*A*/
  v = 1;

  #pragma omp task depend(inout:x) /*C*/
  v *= 2;

  #pragma omp task depend(inout:x) /*B*/
  v += 2;
}
```

$x = ((1) * 2) + 2 = 4$





Dependencies between Siblings only

```
int x;

#pragma omp task depend(in: x)
{
  #pragma omp task depend(out: x)
  { … }
}

#pragma omp task depend(out: x)
{ … }
```

related, as tasks are siblings

unrelated as tasks are no siblings

`taskloop` construct

```
taskloop [clauses]
do-/for-loop
```

- wraps chunks of iterations of assoc. loops into tasks and executes them
  - **not** a worksharing construct
  - however: created tasks can be executed by all threads in current team

- advantages
  - can be arbitrarily nested
    - worksharing loops require nested parallelism
  - explicit tasks cannot encounter worksharing loops
  - automatic load balancing

```
#pragma omp parallel num_threads(2)
#pragma omp single
{
  int from = omp_get_thread_num();
  #pragma omp taskloop
  for (int i = 0; i < 5; ++i) {
    printf("%d %d %d\n",
      i, omp_get_thread_num(), from);
  }
}
```

one thread encounters it, all threads execute tasks, 5 lines of output

possible output:

```
3 0 0
4 0 0
0 1 0
1 1 0
2 1 0
```

`taskloop` is executed 2 times

```
#pragma omp parallel num_threads(2)
{
  #pragma omp taskloop
  for (int i = 0; i < 5; ++i) {…}
}
```

`taskloop` clauses

- loop related:
  - **`collapse`**, **`reduction`**
- task related clauses are applied to the created tasks:
  - **`final`**, **`if`**, **`in_reduction`**, **`mergeable`**, **`priority`**, **`untied`**
- chunk size related:
  - **`grainsize`**, **`num_tasks`**
- data sharing attributes:
  - **`firstprivate`**, **`private`**, **`shared`**, **`lastprivate`**
- **`taskloop`** is implicitly wrapped into a **`taskgroup`**:
  - **`nogroup`** removes impl. **`taskgroup`**

`taskloop` clauses

- **`grainsize([strict:]n)`**
  - task has between **`n`** and **`2n`** iterations
  - with **`strict`** each task has **`n`** iterations
  - last chunk can have less than **`n`** iterations

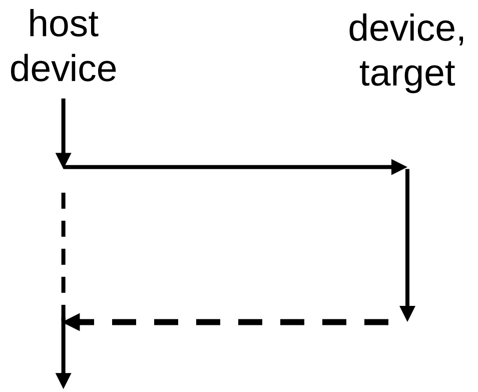
- **`num_tasks([strict:]n)`**
  - generated no. of tasks will be = **`min(n,`** ***`no. of iterations`*****`)`**

Offloading

Introduction

- execute code on a device, typically an accelerator
  - not necessarily a GPU, can also be an FPGA, DSP, …
  - OpenMP tries to abstract from the targeted device's architecture
- **target**: device where code and data is offloaded to
- execution always starts on the **host device**

- here only a small fraction of the standard is covered



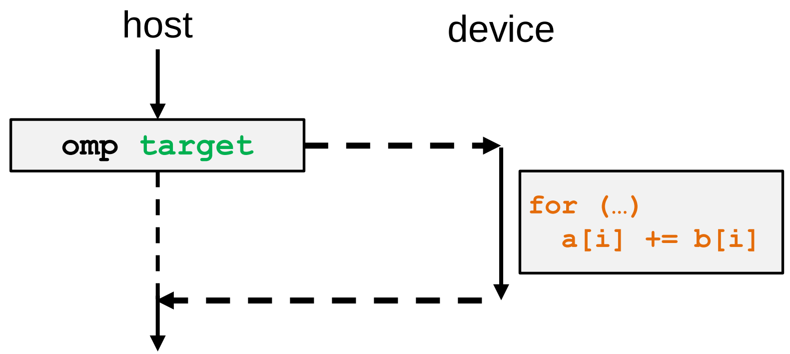
target construct

```
target [clauses…]
<structured block>
```

- execute associated structured block on the device
- on the target:
  - execution is initially single threaded
- on the host:
  - wait until offloaded code completes

- `target` construct cannot be nested inside another `target` construct

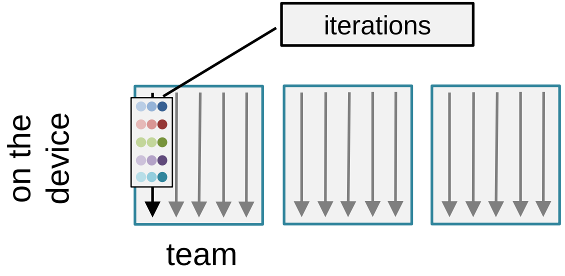
```
int a[1024], b[1024];
/* init a and b */
#pragma omp target
{
  for (int i = 0; i < 1024; ++i)
    a[i] += b[i];
} /* wait until complete */
```



Generating Parallelism

- **`target`** construct alone does not generate parallelism

```
#pragma omp target
for (int i = 0; i < 1024; ++i)
  a[i] += b[i];
```

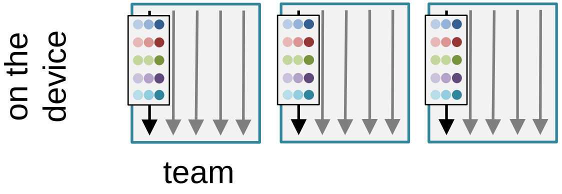


visualization idea based on: Using OpenMP 4.5 Target Offload for Programming Heterogeneous Systems, NASA Advanced Supercomputing Division, Mar 20, 2019

Generating Parallelism

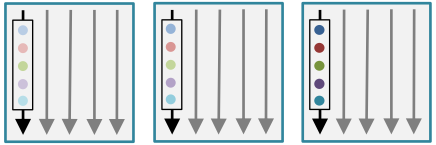
- `teams` construct
  - generate **league of teams**
  - a team has only one initial thread
  - each team executes the same code
  - how many teams: impl. defined
    - `num_teams(n)` clause

```
#pragma omp target teams
for (int i = 0; i < 1024; ++i)
  a[i] += b[i];
```



- `distribute` construct
  - distributes iteration space of associated loop(s) over teams

```
#pragma omp target teams distribute
for (int i = 0; i < 1024; ++i)
  a[i] += b[i];
```

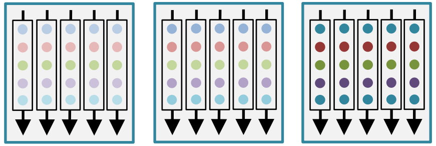


visualization idea based on: Using OpenMP 4.5 Target Offload for Programming Heterogeneous Systems, NASA Advanced Supercomputing Division, Mar 20, 2019

Generating Parallelism

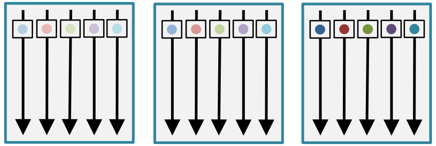
- `parallel` construct
  - gen. parallel region with multiple threads inside each team

```
#pragma omp target teams distribute \
                       parallel
for (int i = 0; i < 1024; ++i)
  a[i] += b[i];
```



- worksharing loop
  - distribute team's iteration space over all threads inside a team

```
#pragma omp target teams distribute \
                       parallel for
for (int i = 0; i < 1024; ++i)
  a[i] += b[i];
```



visualization idea based on: Using OpenMP 4.5 Target Offload for Programming Heterogeneous Systems, NASA Advanced Supercomputing Division, Mar 20, 2019

Generating Parallelism

- **`simd`** construct
  - use SIMD lanes in each thread

```
#pragma omp target teams distribute \
                          parallel for simd
for (int i = 0; i < 1024; ++i)
  a[i] += b[i];
```

- how each directive maps to a GPU entity depends on the compiler

Generating Parallelism

- some possible combinations

```
omp target <sb>
omp target parallel <sb>
omp target parallel for/do <ln>
omp target parallel for/do simd <ln>
omp target simd <ln>
omp target teams <sb>
omp target teams distribute <ln>
omp target teams distribute parallel for/do <ln>
omp target teams distribute parallel for/do simd <ln>
omp target teams distribute simd <ln>
```

```
sb: structured block
ln: loop nest
```

not covered: section, loop construct

`target teams` construct

- each team has a new initial thread
- teams are loosely coupled
  - in contrast to the `parallel` construct
- no synchronization across teams

```
#pragma omp target teams
{ … }
```

```
#pragma omp target
#pragma omp teams
{ … }
```

`target teams` must be a compound construct or directly nested

clauses:

- `num_teams(expr)` clause
  - no. of teams to create
  - if unspecified gen. no. of teams is implementation defined
- `thread_limit(expr)` clause
  - max. no. of active threads in a team
- `if(expr)` clause
  - evaluate to true: create teams
  - evaluate to false: create only 1 team
- `shared, private, firstprivate, default:`
  - usual meaning
- `reduction` clause: see later

`distribute` construct

- distribute iterations of associated loop over teams
  - must be strictly nested inside a **`teams`** construct
  - iteration space must be the same for all teams
  - no implicit barrier at the end

```
#pragma omp target teams distribute
<loop>
```

```
#pragma omp target teams
#pragma omp distribute
<loop>
```

`distribute` must be a compound construct or strictly nested

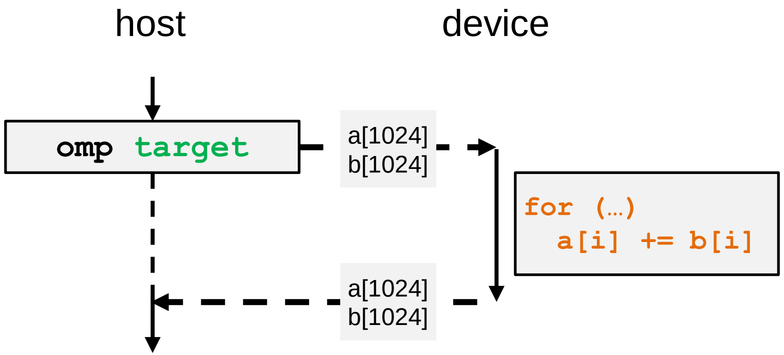
- **`dist_schedule(static[,chunk_size])`** clause
  - if unspecified: implementation defined
  - w/o **`chunk_size`**: each team gets one equally sized chunk
- **`collapse(n)`** clause
  - same as for **`for`**/**`do`** construct
  - associate and collapse iteration space of n nested loops

Data Mapping

- host and device memory can be separate
- mapping of variables ensures
  - a variable is accessible on the target, e.g. by copy or allocation
  - a consistent memory view
- what can be mapped:
  - variables, array sections, members of structures
- mapping causes a presence check
  - copy to device only if not already present
- mapping attributes can be
  - implicit or explicit

```
int a[1024], b[1024];
/* init a and b */
#pragma omp target
{
  for (int i = 0; i < 1024; ++i)
    a[i] += b[i];
} /* wait until complete */
```

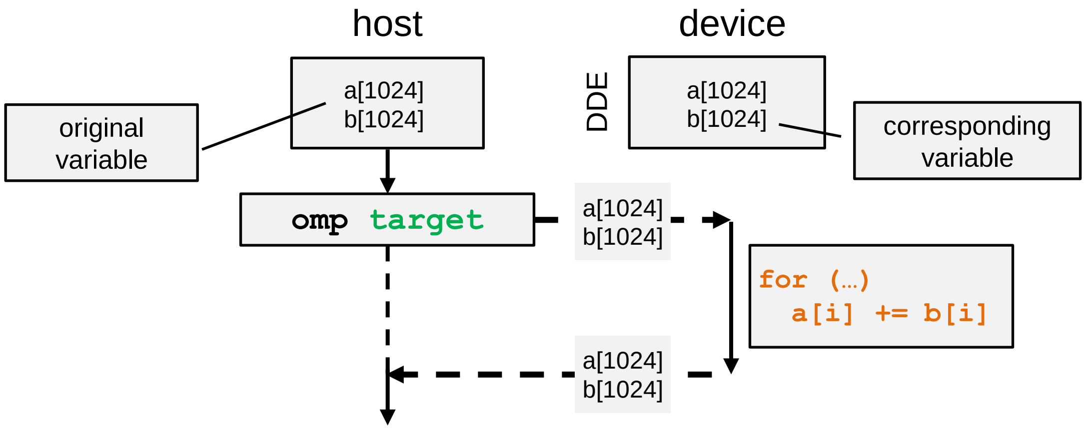
here, implicit mapping attributes cause variables to be mapped, note a[1024], b[1024]



Device Data Environment (DDE)

- exists for each device
  - exists beyond a single target region
- contains all variables accessible by threads running on the device
- mapping ensures a variable is in a device's DDE

```
int a[1024], b[1024];
/* init a and b */
#pragma omp target
{
  for (int i = 0; i < 1024; ++i)
    a[i] += b[i];
} /* wait until complete */
```



`map` clause

- map clause

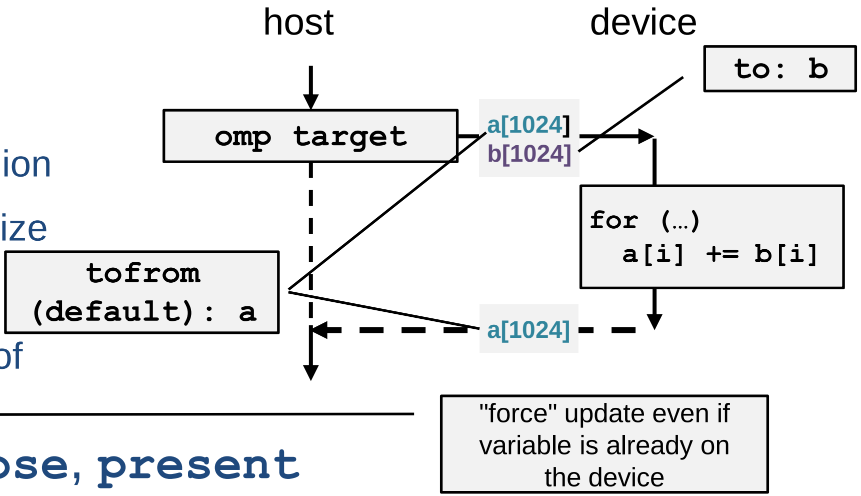
```
map([<mtm>,]<map-type>: <variables>)
```

- `map-type`: how a variable is mapped

`tofrom`	default, copy to device on entry of target region and back at the end
`to`	copy to device on entry of target region
`from`	allocate on entry of target region, copy from device to host on exit of target region
`alloc`	on entry, allocate on device, but do not initialize
`release`	counterpart to `alloc`
`delete`	removes variable from device (independent of RC)

- `mtm`: `map-type-modifier`: `always`, `close`, `present`

```
int a[1024], b[1024];
/* init a and b */
#pragma omp target map(a) map(to:b)
{
  for (int i = 0; i < 1024; ++i)
    a[i] += b[i];
} /* wait until complete */
```



Allocating on the Device

- map-type `alloc`
  - allocate variable/array on device
  - no initialization is performed
  - no copy back to host
- useful, e.g. when an array is only used on the device

`tmp` allocated on the device

```
int tmp[1024];

#pragma omp target map(alloc:tmp)
{
  for (int i = 0; i < 1024; ++i)
    tmp[i] = compute(i);

  for (int i = 0; i < 1024; ++i)
    work(tmp[i]);

  for (int i = 0; i < 1024; ++i)
    work2(tmp[i]);
}
```

`tmp` not copied back

How to map dynamically allocated arrays in C/C++

- map dynamically allocated arrays via array section syntax

```
array[[lower-bound]:length]
```

```
double * a = malloc(sizeof(double) * n_el);
double * b = malloc(sizeof(double) * n_el);
/* init a */

#pragma omp target map(to:a[:n_el]) \
                   map(alloc:b[:n_el])
for (int i = 0; i < n_el; ++i) {
  b[i] = a[i];
}
```

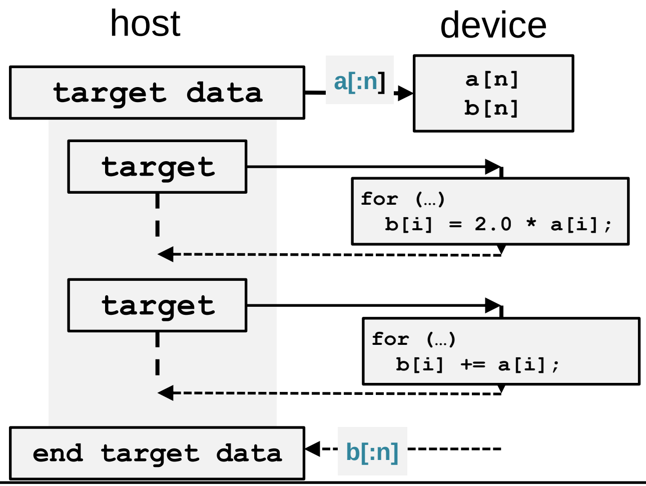
target data construct

```
target data [clauses]
<block>
```

- map data for the duration of the associated block to the DDE
  - **<block>** still executed on host
  - **<block>** typically includes multiple **target** regions
- clauses:
  - **map()** with **to**, **from**, **tofrom**, **alloc**
  - not covered: **device**, **if**, **use_device_addr**, **use_device_ptr**

```
#pragma omp target data map(to:a[:n]) \
                        map(from:b[:n])
{
  #pragma omp target
  for (int i = 0; i < n; ++i)
  { b[i] = 2.0 * a[i]; }

  #pragma omp target
  for (int i = 0; i < n; ++i)
  { b[i] += a[i]; }
}
```



`target update` Construct

```
target update [clauses]
```

- copy data between host and device
  - runs on the host
  - cannot appear inside a `target` construct
  - copy is always performed
    - in contrast to `target map(…)`

```
#pragma omp target data map(to:a[:n]) \
                        map(from:b[:n])
{
  #pragma omp target
  for (int i = 0; i < n; ++i)
  { b[i] = 2.0 * a[i]; }

  #pragma omp target update from(b[:n])
  /* do something with b */

  #pragma omp target
  for (int i = 0; i < n; ++i)
  { b[i] += a[i]; }
}
```

- clauses
  - `to(var-list)` copy vars. to device
  - `from(var-list)` copy vars. to host
  - not covered: device, if, nowait, depend

`enter data`/`exit data` directives

```
target enter data map(…) [clauses]
target exit data map(…)  [clauses]
```

→ map data

→ unmap data

- **unstructured**
- can be called at any point on host
- at **`exit data`**: listed variables not present on the device are ignored
- clauses not covered: **`device`**, **`if`**, **`depend`**, **`nowait`**

allowed: to, alloc

```
double * vec_allocate(int n_el)
{
  double * a = malloc(…);
  #pragma omp target enter data \
              map(alloc:a[:n_el])
  return a;
}

void vec_free(double * a)
{
  #pragma omp target exit data \
              map(release:a[:n_el])
  free(a);
}
```

allowed: from, release, delete

Selecting a Device

- without specification the default device is used
- default device:
  - get: **`omp_get_default_device()`**
- logical device ids in the range from
  `0` to **`omp_get_num_devices() - 1`**
- use specific device with id:
  - env. var. **`OMP_DEFAULT_DEVICE`**
  - **`omp_set_default_device(id)`**
  - **`device(id)`** clause of **`target`** … clauses

Useful Runtime API Calls

- get default device
  - `int omp_get_default_device()`
  - `integer function omp_get_default_device()`

  H

- set default device
  - `void omp_set_default_device(int device)`
  - `subroutine omp_set_default_device(device) integer device`

  H

- return no. of non-host offload devices
  - `int omp_get_num_devices();`
  - `integer function omp_get_num_devices()`

  H

- return no. of initial/host device
  - `int omp_get_initial_device()`
  - `integer function omp_get_initial_device()`

  H

- return calling thread's device no.
  - `int omp_get_device_num()`
  - `integer function omp_get_device_num()`
  - on host returns the value of `omp_get_initial_device()`

  H/D

- return if calling thread runs on host
  - `int omp_is_initial_device()`
  - `integer function omp_is_initial_device()`

  H/D

callable from host H, device D

Env. Vars. related to Offloading

- **`OMP_DEFAULT_DEVICE=<n>`** **with n ≥ 0**
  - set the default device used for executing `target` constructs
- **`OMP_TARGET_OFFLOAD=mandatory | disabled | default`**
  - `mandatory`: usage of unsupported or unavailable device or invalid device number causes termination
  - `disabled`: if supported by the OpenMP RT, the only device available is the host
- **`OMP_TEAMS_THREAD_LIMIT=<n>`**
  - maximum no. of threads each team can have

Performance Aspects

- need to know what underlying architecture/RT will do
  - copy or not copy
  - avoid unnecessary copies
- mapped variables require a presence check on the device
  - hence: **`private`**/**`firstprivate`** variables are faster
- determine how your compiler maps directives to GPU entities
  - check how **`num_teams`**/**`thread_limit`** are interpreted

Inspecting Transfers

- GCC
  - **`GOMP_DEBUG=1 ./a.out`**
  - prints a lot of information
- LLVM/clang
  - env. var. **`LIBOMPTARGET_INFO`**
  - from https://openmp.llvm.org/design/Runtimes.html#llvm-openmp-target-host-runtime-libomptarget
    - **`0x01`**: show data args. when entering device kernel
    - **`0x02`**: show when a mapped address already exists on device
    - **`0x04`**: Dump the contents of the device pointer map at kernel exit
    - **`0x08`**: Indicate when an entry is changed in the device mapping table
    - **`0x10`**: Print OpenMP kernel information from device plugins
    - **`0x20`**: Indicate when data is copied to and from the device
  - **`LIBOMPTARGET_INFO=$((0x01 | 0x02)) ./a.out`**
- NVHPC
  - env. var. **`PGI_ACC_DEBUG=1`**
  - env. var. **`NVCOMPILER_ACC_NOTIFY=1`**