NHR FAU

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Introduction to OpenMP
Part 2

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

High Performance Computing

Outline

- thread affinity
- memory locality and programming for ccNUMA systems
- single instruction multiple data (SIMD) programming
- shared-memory parallelization with tasking
- accelerator programming via offloading

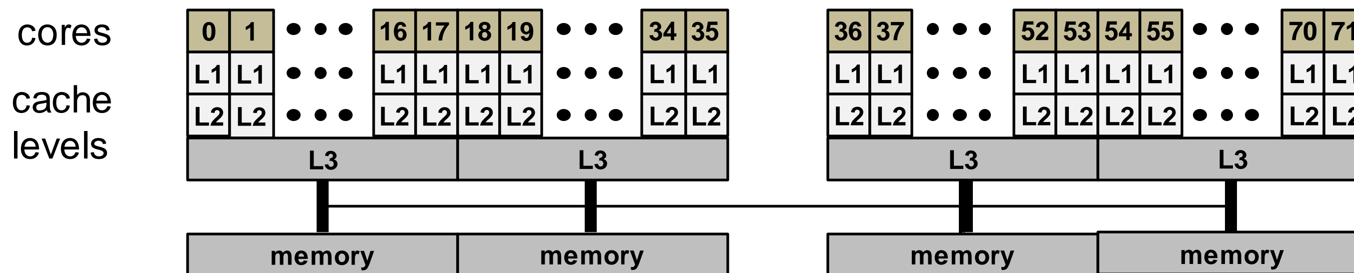
- 9:00 – 10:30 10:40 – 12:00
- 13:00 – 14:30 14:40 – 16:00

Thread Affinity

NHR FAU
FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
HPC High Performance Computing

Thread Affinity

- controls to which places threads are assigned
  - a.k.a. thread binding, thread pinning



- Why does it matter?
  - use shared/separate resources
  - avoid thread migration

Places – Where Threads Can Be Executed

- env. var. `OMP_PLACES`
  - values can be an abstract name

what is supported depends on the OpenMP version

abstract name	description
`threads`	HW threads, a.k.a. SMT-threads, virtual cores
`cores`	physical CPU cores
`ll_caches`	cores sharing a last level cache (v5.1)
`numa_domains`	cores belonging to the same NUMA domain (v5.1)
`sockets`	cores belonging to a socket

```
OMP_PLACES="cores"
OMP_PLACES="cores(4)"
OMP_PLACES="sockets"
OMP_PLACES="sockets(2)"
```

  - selected only a certain amount: **`abstract-name(count)`**
- hardware ids of cores, format examples
  - `<id>[,<id>[,…]]`
  - `{<ids>}`
  - `{<ids>},{<ids>},…`
  - `{<ids>}[:<len>[:<stride>]]`

```
OMP_PLACES="0,2,4,6,8"
OMP_PLACES="{0,1},{2,3}"
OMP_PLACES="{0}:5:2"
```

Control Affinity Policy

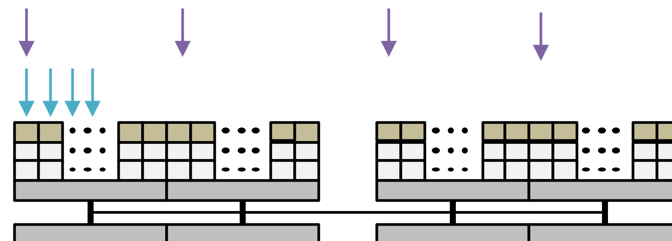
- env. var. `OMP_PROC_BIND` for setting affinity policy

value	description
`false`	disable affinity, `proc_bind` clause (`parallel` construct) is ignored
`true`	enable affinity, strategy is implementation defined
`close`	bind threads to adjacent places →typically used low latency
`spread`	distribute threads equally over available places → typically used for high bandwidth or separate resources
`primary`	bind all threads to the place of the initial thread

- `proc_bind` clause in `parallel` construct
  - values: close, spread, master (deprecated), primary

`OMP_PROC_BIND=` spread / close

```
OMP_NUM_THREADS=4
OMP_PLACES=cores
```



Show where threads are bound to

- env. var. `OMP_DISPLAY_AFFINITY=true`
  - print where threads are bound to

```
$ OMP_NUM_THREADS=6 OMP_PLACES=cores \
   OMP_PROC_BIND=true OMP_DISPLAY_AFFINITY=true ./a.out
level 1 thread 0x7f07a55e77c0 affinity 0-1
level 1 thread 0x7f07a51ff640 affinity 2-3
level 1 thread 0x7f07a49fe640 affinity 4-5
level 1 thread 0x7f07a41fd640 affinity 6-7
level 1 thread 0x7f07a39fc640 affinity 8-9
level 1 thread 0x7f07a31fb640 affinity 10-11
```

SMT enabled, i.e. one physical core houses two virtual cores

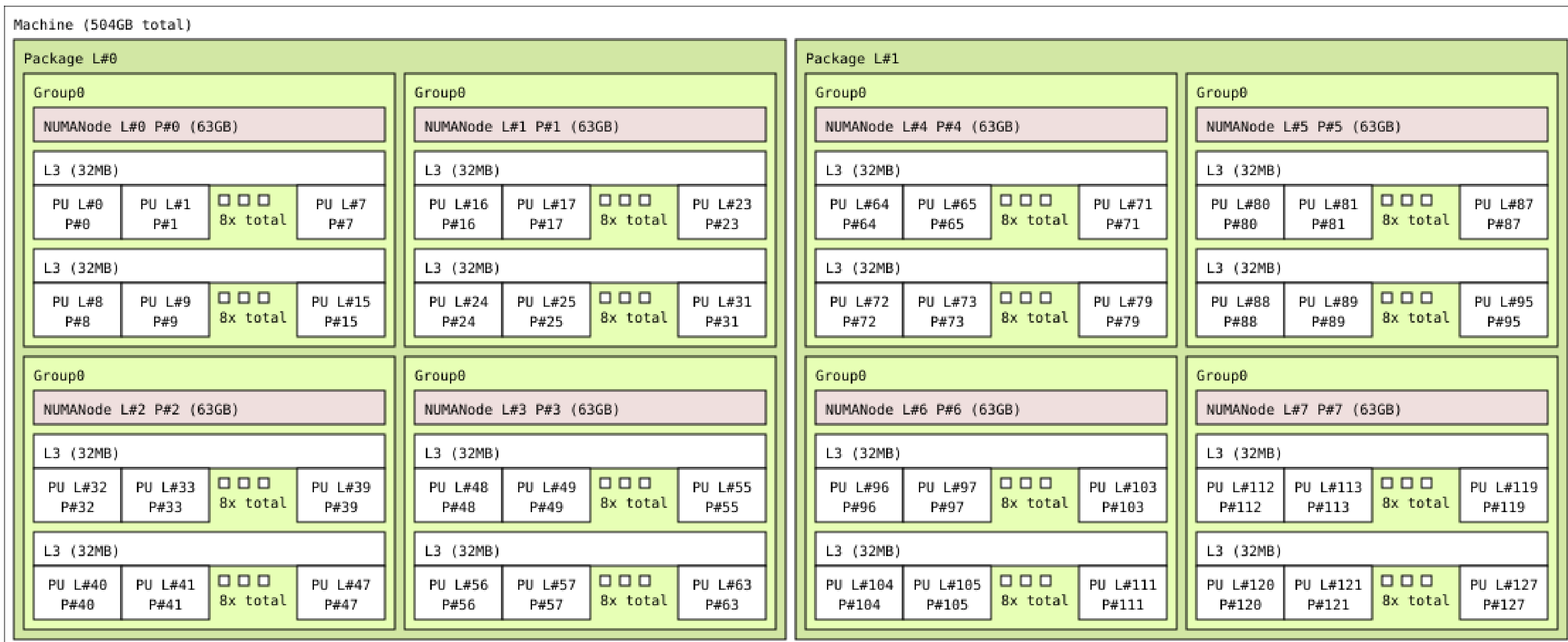
Examine Topology

- **`lscpu`**
  - CPU architecture/features, caches, NUMA LDs
- **`lstopo`** (hwloc)
  - CPUs, caches, NUMA LDs, GPUs, network interfaces, ….
- **`numactl`**
  - show NUMA LDs: **`numactl -H`**
  - also allows for controlling affinity, see later
- **`nvidia-smi`**
  - NUMA LDs, associated CPUs and GPUs
  - **`nvidia-smi topo -m`**
- **`likwid-topology`**
  - CPUs, caches, NUMA LDs

many more….

Topology of Alex A40 Node

used **`lstopo`**, not showing GPUs, IB devices, SSDs, ...



Memory Locality and Programming for ccNUMA Systems

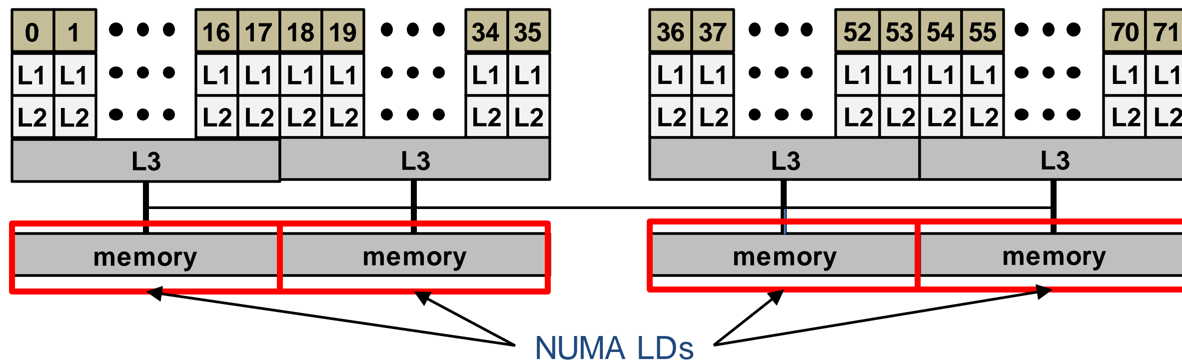
NHR FAU

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

High Performance Computing

ccNUMA

- ccNUMA – cache-coherent non-uniform memory access
  - memory is distributed over locality domains in granularity of pages
  - bandwidth & latency differ from core to locality domains
  - each core is assigned to a locality domain
    - typically the closest
    - highest bandwidth, lowest latency



First Touch Policy

- default policy: first touch
- typically memory is allocated in two stages
  1. memory is only reserved* but not yet associated with pages in RAM
  2. writing to not yet associated pages triggers allocation
- a memory page is placed into the locality domain the core touching it belongs to

```c
double * d = malloc(n * sizeof(double));

for (int i = 0; i < n; ++i)
  d[i] = i;
```

memory only "reserved"

real allocation and placement happens here

```fortran
allocate(d(n))

do i = 1, n
  d(i) = i
end do
```

* depending on the overcommit system settings more memory than available can be reserved

First Touch Policy

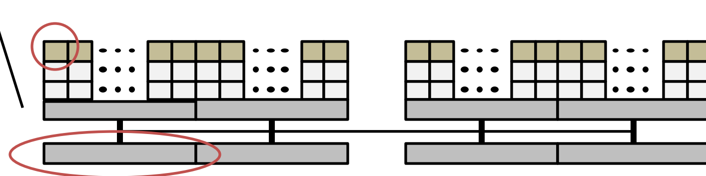
- core that "touches" memory first, places it into its locality domain

```
double * d = malloc(n * sizeof(double));

for (int i = 0; i < n; ++i)
  d[i] = i;

#pragma omp parallel for \
              schedule(static)
for (int i = 0: i < n; ++i)
  /* work on d[i] */
```

serial initialization causes only one core to place memory

```
allocate(d(n)

do i = 1, n
  d(i) = i
end do

!$omp parallel do &
!$omp   schedule(static)
do i = 1, n
  ! work on d(i)
end do
!$omp end parallel do
```

array d will not be distributed



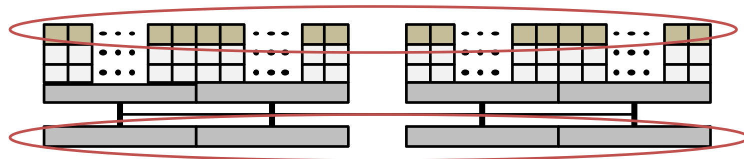
First Touch Policy

- place data how it is later accessed

```c
double * d = malloc(n * sizeof(double));

#pragma omp parallel for \
            schedule(static)
for (int i = 0; i < n; ++i)
  d[i] = i;

#pragma omp parallel for \
            schedule(static)
for (int i = 0: i < n; ++i)
  /* work on d[i] */
```

each thread places that part of `d` it will later access

```fortran
allocate(d(n))

!$omp parallel do &
!$omp   schedule(static)
do i = 1, n
  d(i) = i
end do
!$omp end parallel do

!$omp parallel do &
!$omp   schedule(static)
do i = 1, n
  ! work on d(i)
end do
!$omp end parallel do
```



Controlling Placement with `numactl`

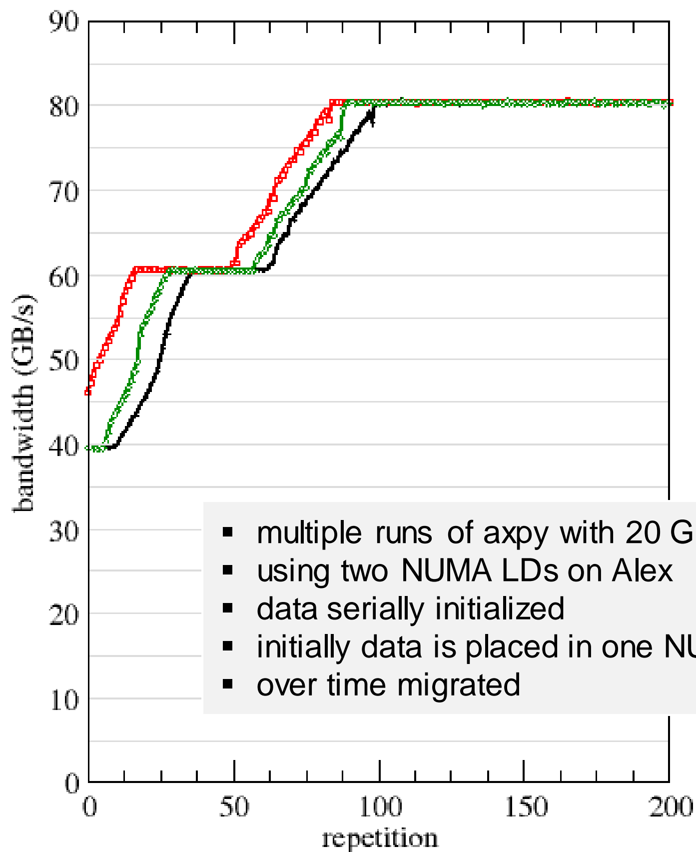
- with `numactl` other policies than first touch can be selected
- use only a subset of NUMA LDs:
  - `-m <nodes>, --membind=<nodes>`
  - `numactl -m <nodes> … <command> <args…>`
- round-robin placement of memory pages over NUMA LD subset:
  - `-i <nodes>, --interleave=<nodes>`
  - `numactl -i <nodes> … <command> <args…>`
- `<nodes>`:
  - comma separated list of single NUMA nodes or ranges thereof,
  - `all`, `!` (for negation)
  - devices, files → see man page
- check selected settings:
  - `numactl … numactl --show`

NUMA balancing

- automatically migrates pages between NUMA nodes
  - reduces remote NUMA traffic
  - incurs some overhead
- **`cat /proc/sys/kernel/numa_balancing`**
  - **`0`** # disabled
  - **`1`** # enabled
- tunable variables under **`/proc/sys/kernel/numa_balancing_*`**

```
# on alex:
$ grep .\* /proc/sys/kernel/numa_balancing*
/proc/sys/kernel/numa_balancing:1
/proc/sys/kernel/numa_balancing_scan_delay_ms:1000
/proc/sys/kernel/numa_balancing_scan_period_max_ms:60000
/proc/sys/kernel/numa_balancing_scan_period_min_ms:1000
/proc/sys/kernel/numa_balancing_scan_size_mb:256
```

NUMA balancing



- multiple runs of axpy with 20 GB workingset
- using two NUMA LDs on Alex
- data serially initialized
- initially data is placed in one NUMA domain
- over time migrated

do not do this, just use a BLAS library

```
void axpy(long n_el, double a,
          register double * x,
          register double * y)
{
    #pragma omp parallel for simd
    for (long i = 0; i < n_el; ++i)
        y[i] = a * x[i] + y[i];
}

...
for (int i = 0; i < n_repetitons; ++i) {
  double time = omp_get_wtime();
  axpy(n_el, a, x, y);
  double duration = omp_get_wtime() - time;
  /* report time and bandwidth */
}
```

NHR FAU

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

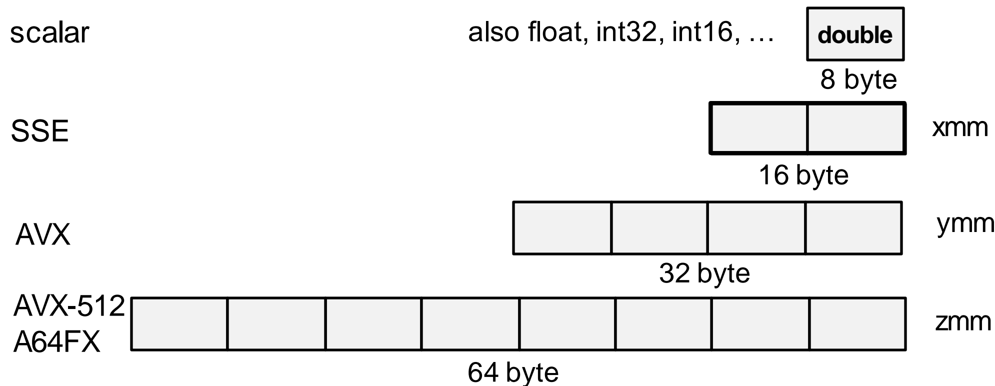
Single Instruction Multiple Data (SIMD) programming

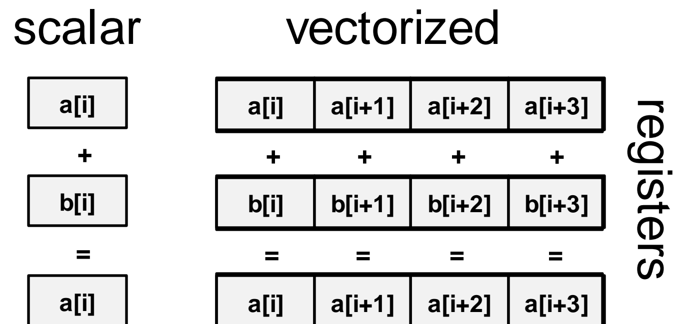
High Performance Computing

SIMD

- SIMD: single instruction multiple data
- registers hold multiple elements
- one instruction performs the operation on each element

- also special instructions for
  - fused-multiply-accumulate (FMA)
  - gather/scatter
  - masked operations
  - …





Vectorizing Loops

≥v4.0

- **concurrent** execution of **loop iterations** through SIMD instructions (**vectorization)**
  - loop is executed in SIMD chunks
  - each chunk consists of multiple SIMD lanes
- only local to the current task
  - single thread optimization
- requirements
  - no dependencies among loop iterations
    - see **`safelen`** if there are
  - no pointer aliasing

loop iterations

scalar

a[i]
+
b[i]
=
a[i]

vectorized

a[i]	a[i+1]	a[i+2]	a[i+3]
+	+	+	+
b[i]	b[i+1]	b[i+2]	b[i+3]
=	=	=	=
a[i]	a[i+1]	a[i+2]	a[i+3]

registers

```
#pragma omp simd [clauses]
for (int i = 0; i < n; ++i)
  a[i] += b[i];
```

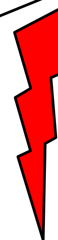
```
!$omp simd [clauses]
do = 1, n
  a(i) = a(i) + b(i)
end do
!$omp end simd
```

Pointer Aliasing

- with **simd** construct we guarantee loop iterations are independent

```
void daxpy(long n_el, double a,
           double * x, double * y)
{
    #pragma omp parallel for simd
    for (long i = 0; i < n_el; ++i)
        y[i] = a * x[i] + y[i];
}

double A[n];
/* init A */
daxpy(n - 1, a, A + 1, A);
```

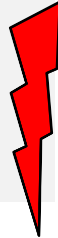
with **simd** we guarantee no pointers are aliased



```
A[0] = a * A[1] + A[0]
A[1] = a * A[2] + A[1]
A[2] = a * A[3] + A[2]
A[3] = a * A[4] + A[3]
…
```

Multiple loop iterations could write to the same memory location

```
/* elements of values array are inside
   the bounds of the hist array */
void compute(int * hist, int n_values,
             int * values) {
  #pragm omp simd
  for (int i = 0; i < n_values; ++i) {
    ++hist[values[i]];
} }
```



vectorization through gather and scatter instruction possible, however, multiple SIMD lanes could write to the same memory location

vectorization is only OK if:

- no two indices are the same
- when there are at least N different elements before the same element occurs again in **`values`** array, use **`safelen(N)`** clause

Data Environment

- clauses: **`private`**, **`lastprivate`**
- loop counter gets privatized as **`lastprivate`**

```
int n = 10;
/* define and init a and b */

int i = 0;
#pragma omp simd
for (i = 0; i < 10; ++i)
  a[i] += b[i];

printf("i: %d\n", n, i);
/* prints: i: 10 */
```

- privatization for **`simd`** loops means one private instance per SIMD lane

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

without **`private(tmp)`**, **`tmp`** would be shared and this would lead to races

`simd` construct clauses

- **`if(expr)`**
  - if false only one loop iteration is executed at a time
- **`simdlen(length)`**
  - hint of how many iterations should be executed concurrently
    - typically 2, 4, 8, 16, depending on variable types and hardware capabilities
    - compiler might unroll the loop beyond SIMD width
- **`safelen(length)`**
  - how many loop iterations can safely be executed concurrently
    - `simdlen` ≤ `safelen` required
- **`aligned(var[:alignment],...)`**
  - specify alignment in bytes for listed variables
    - must be correct, might help optimizer

`simd` construct clauses

- **`linear(`*`list`*`[:`*`step`*`])`**
  - listed variables have a linear relationship with the loop
  - why: help the compiler
  - if ***`step`*** is not specified its 1
  - ***`step`*** must be invariant inside the loop
  - listed variables are privatized

```
#pragma omp simd linear(j:2)
for (int i = 0; i < n / 2; ++i) {
  a[i] += b[j];
  j += 2;
}
```

`simd` construct clauses

- **`collapse(n)`**
  - associates n loops
  - might create complex non-optimal assembly
    - check this is what you expect

```
#pragma omp simd collapse(2)
for (int y = 1; y < ny - 1; ++y) {
  for (int x = 1; x < nx - 1; ++x) {
    a[y * nx + x] = (b[y * nx + x]
          + b[y * nx + x + 1]
          + b[y * nx + x - 1]
          + b[(y - 1) * nx + x]
          + b[(y + 1) * nx + x]) * 0.25;
  }
}
```

- **`reduction(rid:list)`**
  - works as already known
  - listed variables are privatized and aggregated at the end

```
double dotp = 0.0;
#pragma omp simd reduction(+:dotp)
for (int i = 0; i < n; ++i) {
  dotp += a[i] * b[i];
}
```

- clauses not discussed: `nontemporal`, `order`

enable `simd` construct support only

- enable only `simd` construct support without enabling other OpenMP constructs/features
- gcc/gfortran/clang: `-fopenmp-simd`
- icc/ifort/icx/ifx: : `-qopenmp-simd`
  - automatically active at ≥ `-O2`

combined `for/do simd` construct

- `simd` can be combined with `for/do` construct

C/C++

```
for simd [for and simd clauses …]
```

Fortran

```
do simd [for and simd clauses …]
```

- iterations of associated loop(s) get
  - vectorized and
  - distributed over threads

```
#pragma omp for simd \
         schedule(simd:static)
for (…)
```

all schedules are allowed

`simd` modifier in `schedule` causes chunk size to be a multiple of simd width

Vectorizing Functions For Usage Within `simd` Loops

```
declare simd [clauses]
```

- generate vector versions of functions to be called from a `simd` loop
  - vectorized math functions typically are already available

```
declare simd simdlen(n) [clauses]
```

- create a version for SIMD width `n`

- restrictions:
  - function cannot have side effects
  - C++: function must not throw

```
#pragma omp simd
for (int i = 0; i < n; ++i)
  a[i] = sin(i);
```

if available vectorized `sin` function called

also create versions of the function for SIMD width 2, 4, and 8

```
#pragma omp declare simd simdlen(2)
#pragma omp declare simd simdlen(4)
#pragma omp declare simd simdlen(8)
double sumit(double a, double b)
{
  return a + b;
}
...
#pragma omp simd
for (int i = 0; i < n; ++i)
  a[i] = sumit(b[i], c[i]);
```

declare simd construct

- **declare simd** + function definition
  - compiler generates SIMD version
- **declare simd** + function declaration
  - tells the compiler a SIMD versions exist

functions.c

```
#pragma omp declare simd simdlen(2)
double sumit(double a, double b)
{
  return a + b;
}
```

main.c

```
#pragma omp declare simd simdlen(2)
double sumit(double a, double b);
...
#pragma omp simd
for (int i = 0; i < n; ++i)
  a[i] = sumit(b[i], c[i]);
```

```
double precision function sumit(a, b)
  !$omp declare simd simdlen(2)
  double precision, intent(in) :: a, b
  sumit = a + b
end function
```

`declare simd` construct clauses

`uniform(list)`

- listed parameters of function will have the same value through concurrent calls from a SIMD loop

```
#pragma omp declare simd uniform(a)
double multit(double a, double b)
{
  return a + b;
}
...
#pragma omp simd
for (int i = 0; i < n; ++i)
  a[i] = multit(2.0, c[i]);
```

`linear(list[:step-size])`

- values of listed parameters have a linear relationship between their SIMD lanes in the form of *step-size*

```
#pragma omp declare simd uniform(by, a)
linear(index:1)
void incr(double by, double * a, int index)
{
  a[index] += by;
}
...
#pragma omp simd
for (int i = 0; i < n; ++i)
  incr(2.0, a, i);
```

`declare simd` construct clauses

`inbranch`

- function is called from inside a branch of a SIMD loop

`notinbranch`

- function is not called from inside a branch of a SIMD loop

- without `inbranch` and `notinbranch` generated code by the compiler must be able to handle both situation

```
#pragma omp declare simd inbranch
double incr(double a)
{ return a + 1.0; }

#pragma omp declare simd notinbranch
double square(double a)
{ return a * a; }

...
#pragma omp simd
for (int i = 0; i < n; ++i) {
  if (a[i] % 2 == 1)
    incr(a[i]);

  square(a[i]);
}
```

`declare simd` construct clauses

`aligned(`*`list:alignment`*`)`

- listed pointer(s) have specified alignment in bytes

```
#pragma omp declare simd aligned(a, b:64)
double process(double a, double b)
{ … }

...
double * a = aligned_alloc(64, n * sizeof(double));
double * b = aligned_alloc(64, n * sizeof(double));
double * c = aligned_alloc(64, n * sizeof(double));
/* init a, b */
#pragma omp simd
for (int i = 0; i < n; ++i)
   c[i] = process(a[i], b[i]);
```

- multiple declarations with different clauses are allowed

```
#pragma omp declare simd inbranch
#pragma omp declare simd notinbranch
#pragma omp declare simd notinbranch \
                         uniform(a)
double incr(double a)
{ return a + 1.0; }
```

NHR FAU

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

Shared-Memory Parallelization With Tasking

High Performance Computing

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 work-loads which cannot be mapped to worksharing constructs
  - allow for dependencies between tasks

Creating Tasks

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

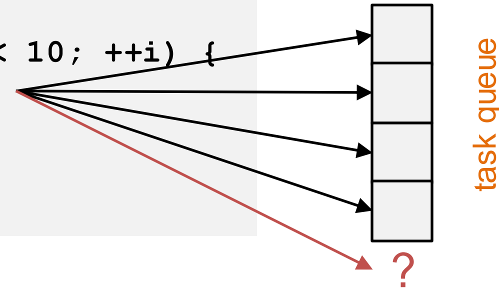
```
#pragma omp parallel
{
  #pragma omp single
  {
    for (...) {
      #pragma omp task
      { /* work */ }
    }
  } /* implicit barrier */
}
```

- 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**

Task Queue

- OpenMP runtimes typically have a task queue
- **deferred** tasks are enqueued there
- waiting threads pick tasks from this queue
- queue has limited capacity for enqueued tasks, i.e. a **threshold**
- if **threshold** is reached:
  - creation of new tasks can be suspended
  - tasks from the queue are processed

```
#pragma omp parallel
#pragma omp single
{
  for (int i = 0; i < 10; ++i) {
    #pragma omp task
    work(data[i]);
  }
}
```



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 Clauses

`priority(value)`

- **hint** to execute tasks with higher priority first
- *`value`*
  - by default `0`
  - range: `[0, max-priority]`
- must be enabled first
  - set environment variable: `OMP_MAX_TASK_PRIORITY=max-priority`
- application **must not** rely on tasks executed regarding their priority

- query maximum priority:
  - `int/integer omp_get_max_task_priority()`

```
#pragma omp parallel
#pragma omp single
{
  #pragma omp task
  low_prio_work();
  #pragma omp task priority(1)
  high_prio_work();
}
```

run with:

```
OMP_MAX_TASK_PRIORITY=1 ./omp-app
```

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)

```
#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

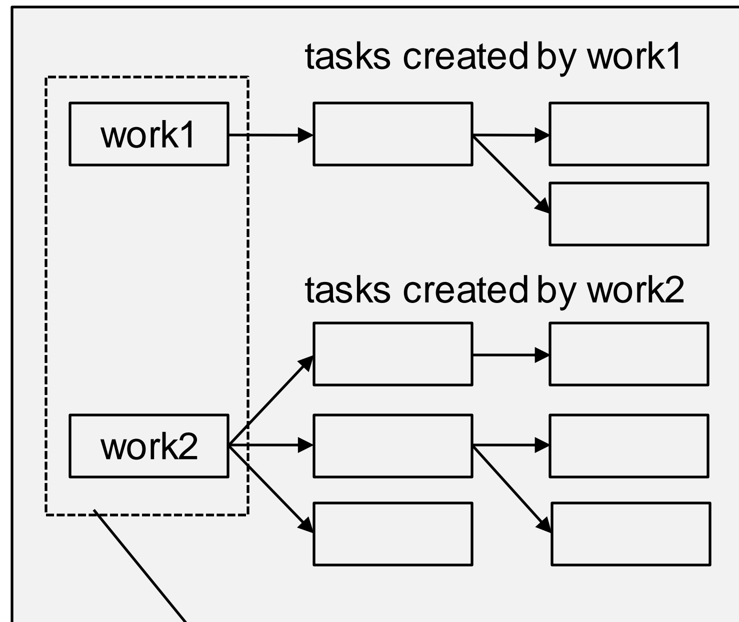
- **`taskwait`**: wait until all child tasks of current (implicit) task are completed
  - **NOTE**: child tasks include only direct children, not grandchildren

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



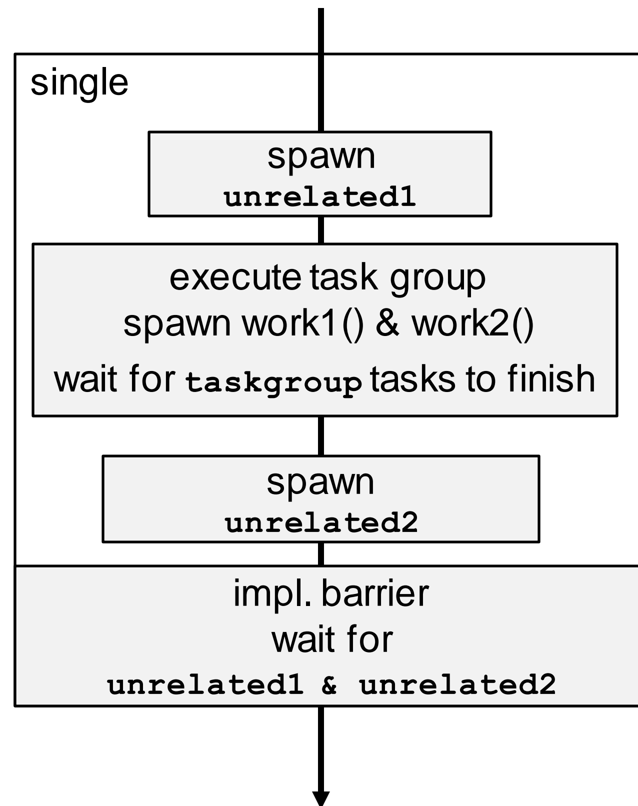
`taskwait` would only wait for work1 and work2

Task Synchronization with `taskgroup`

- allows for dedicated waiting on tasks

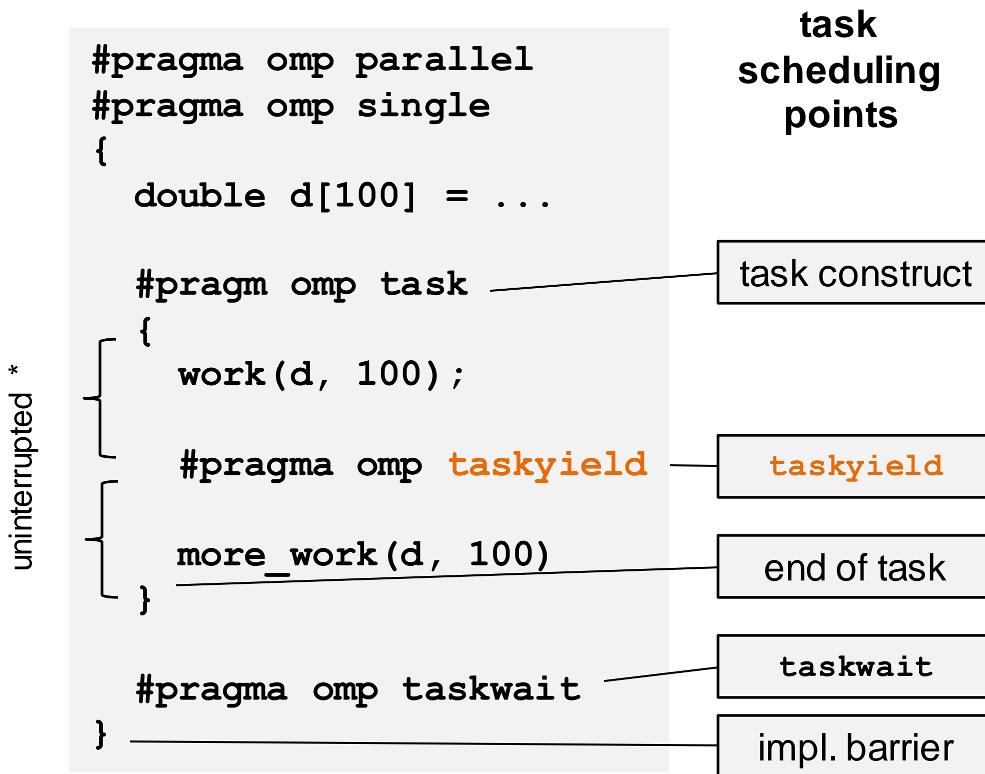
```
#pragma omp parallel
#pragma omp single
{
  #pragma omp task
  unrelated1();

  #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*)
- **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



*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

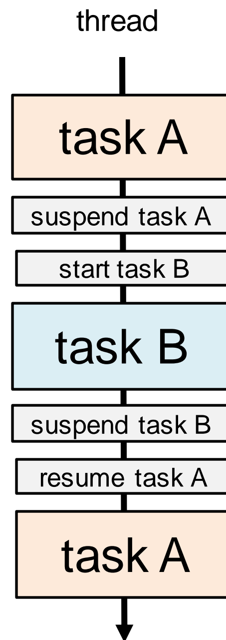
Tied and Untied Tasks

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

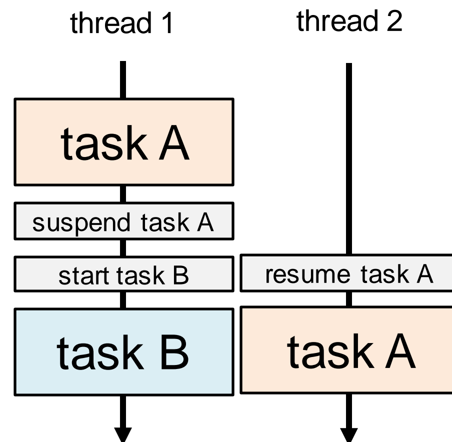
NOTE: tied might be desired if cache/NUMA locality is needed

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

tied tasks (default)



untied tasks



Example `untied`

```
#pragma omp parallel
#pragma omp single
{
  for (int i = 0; i < n; ++i) {
    #pragma omp task
    work(data[i]);
  }
}
```

if implicit task has been suspended, it can only be picked up by the thread that started it

```
#pragma omp parallel
#pragma omp single
{
  #pragma omp task untied
  for (int i = 0; i < n; ++i) {
    #pragma omp task   // (1)
    work(data[i]);
  }
}
```

if task has been suspended, it can be picked up by any thread

- thread executes untied task that generates new tasks
- if threshold of unassigned tasks is reached, the generating task might be **suspended** at TSP `(1)`
- thread now processes unassigned tasks
- if other threads complete their work earlier, they can pickup the suspended generating task

from: OpenMP Application Programming Interface Examples, 5.1 `task` and `taskwait` Constructs

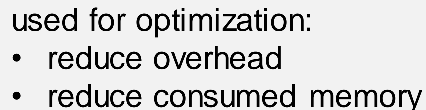
Final and Mergable Tasks

`final(finalize-expr)`

- if **`final-expr`** is true task is final
- final tasks are undeferred, i.e., executed immediately
- child tasks of final tasks are also final
  - in contrast to **`if`** clause

`mergable(mergable-expr)`

- no extra data environment for task is created if task is
  - final
  - or undeferred
- reduces memory overhead



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 build by matching dependencies to dependencies of already submitted tasks

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

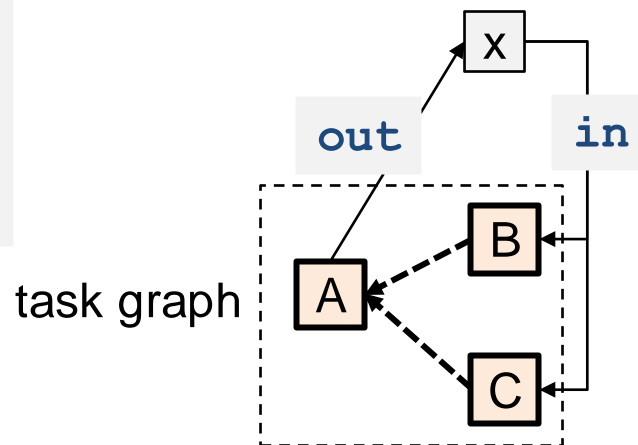
list of variables, array elements and sections

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

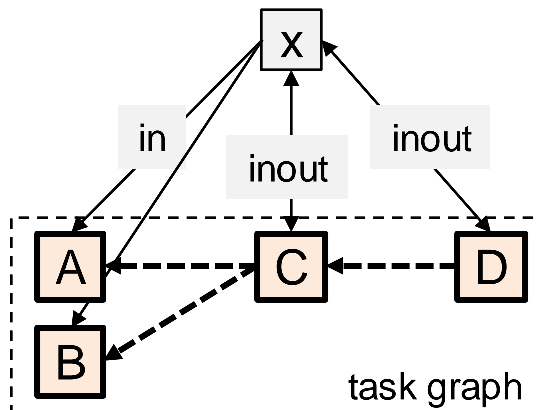
```
#pragma omp task depend(out:x) /*A*/
/*…*/
#pragma omp task depend(in:x)  /*B*/
/*…*/
#pragma omp task depend(in:x)  /*C*/
/*…*/
```



`out`/`inout` dependency

- depends on
  - last `out` dependency of the listed variables, if any
  - all `in` dependencies schedule 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*/
/*…*/
```



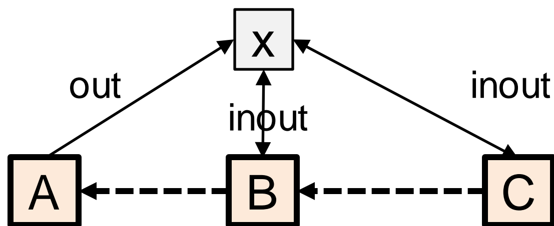
Oder of Creation Matters

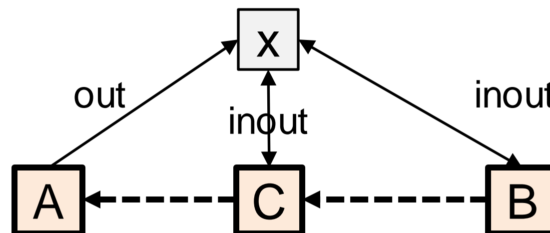
```
int v = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(out:x)   /*A*/
  v = 1;

  #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);
  }
}
```

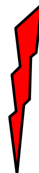
one thread encounters it, all threads execute tasks, 5 lines of output

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`*****`)`**

NHR FAU

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

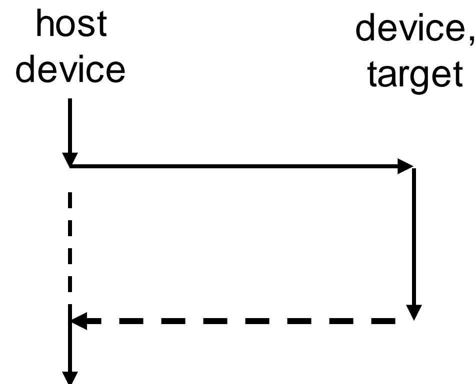
Offloading

High Performance Computing

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



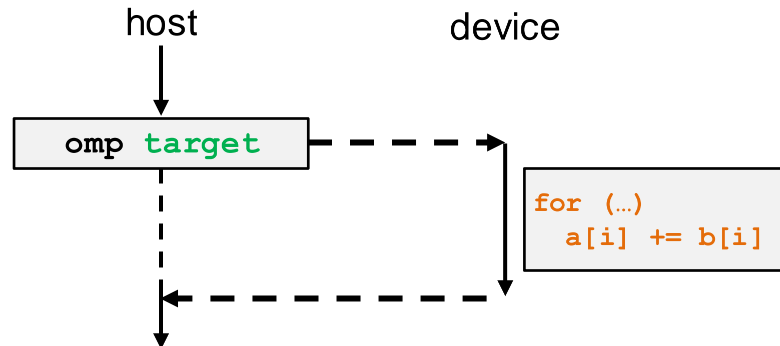
Offloading Code to the Target

```
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

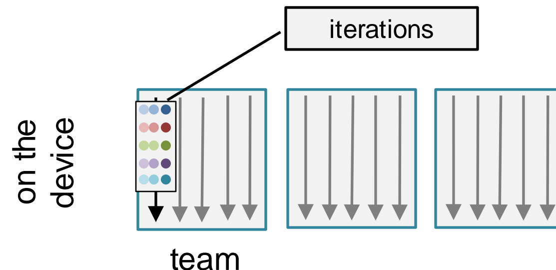
```
int a[1024], b[1024];
/* init a and b */
#pragma omp target
{
  for (int i = 0; i < 1024; ++i)
    a[i] += b[i];
} /* wait until complete */
```



Generating Parallelism on the Target

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

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

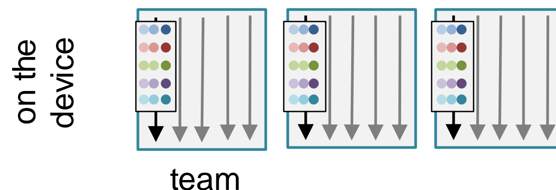


visualization idea based on: Using OpenMP 4.5 Target Offload for Programming Heterogeneous Systems, NASA Advanced Supercomputing Division, Mar 20, 2019

Generating Parallelism on the Target

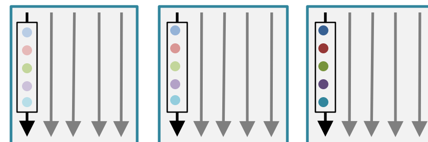
- `teams` construct
  - generate **league of teams**
  - a team has only one initial thread
  - each team executes the same code
  - how many teams: impl. defined
    - `num_teams(n)` clause

```
#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];
```

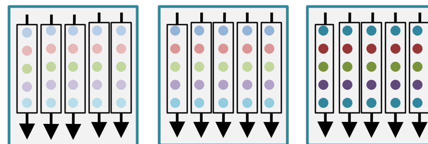


visualization idea based on: Using OpenMP 4.5 Target Offload for Programming Heterogeneous Systems, NASA Advanced Supercomputing Division, Mar 20, 2019

Generating Parallelism on the Target

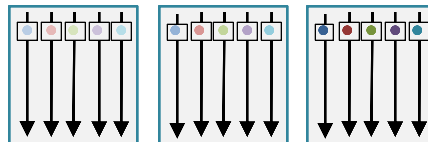
- `parallel` construct
  - gen. parallel region with multiple threads inside each team

```
#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 on the Target

- **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>
```

`distribute` must be a compound construct or strictly nested

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

- `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

`distribute` construct

- `private`, `firstprivate`, `lastprivate` clauses: usual meaning
- `order` clause: not handled here

- reproducible schedule:
  - `order(reproducible)`
  - `dist_schedule(static[,chunk_size]) order(...)` where order does not contain `unconstrained`

- avoid data races with `lastprivate`
  - `lastprivate` variables should not be accessed between end of `distribute` and `teams` construct

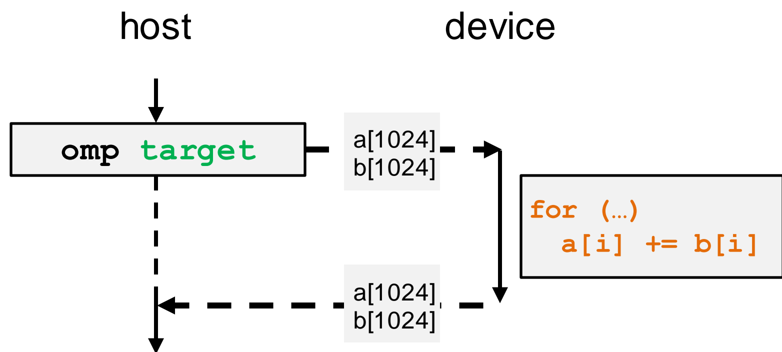
```
#pragma omp target teams
{
  #pragma omp distribute \
                 lastprivate(lp)
  { <loop> }
  /* other code        */
  /* do not access lp */
}
```

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 */
```

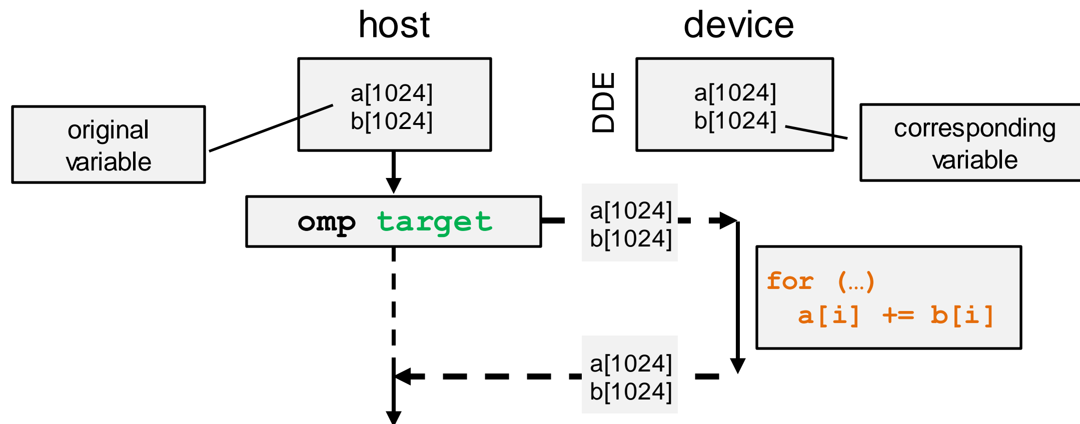
here, implicit mapping attributes cause variables to be mapped, note a[1024], b[1024]



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 */
```



Data Mapping Attributes

- explicit:
  - referenced in `private`, `firstprivate`, `is_device_ptr` clause: `private`
  - declared inside `target` construct: `private`
  - referenced in a `map` clause: selected *`map-type`*
- scalar variable: `firstprivate`
  - except if `target … defaultmap(tofrom:scalar)`
    - then map-type `tofrom`
- non-scalar variable: map-type `tofrom`
  - entry: copy to device, exit: copy back
- C/C++: pointer variable in pointer based array section: `private`

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

map clause

- map clause

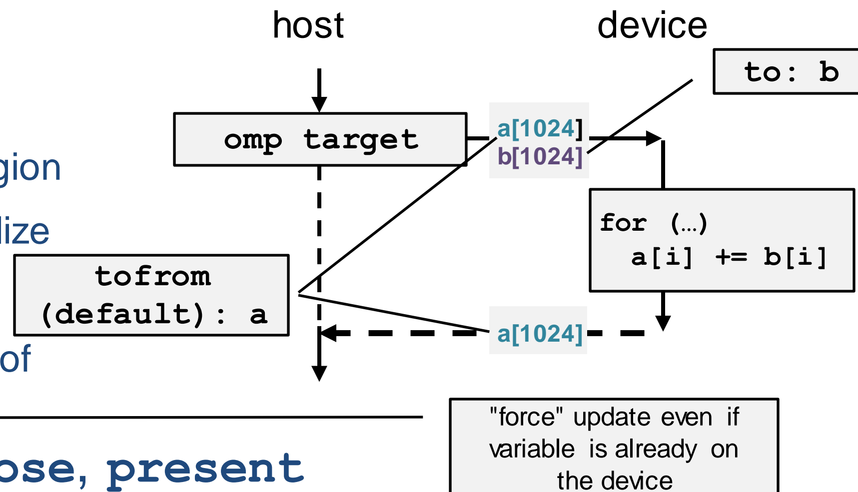
```
map([<mtm>,]<map-type>: <variables>)
```

- **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];
}
```

DDE and Reference Counts

- every variable is inside a device data environment (DDE)
  - exists only once
  - has a **reference count** (**RC**) associated
- an existing variable in a DDE has always RC ≥ 1

var. on map enter:

- if `RC=0`: var. newly allocated
- `++RC`
- if `map-type` in `to|tofrom` and (`RC=1 || ` *`mtm`*`=always`):
  - copy value of var. from host to device
- else:
  - no copy to the device takes place

var. on a map-exit:

- if `map-type` in `from|tofrom` and (`RC=1 || mtm=always`)
  - copy value of var. from device to host
- `--RC`
- if `map-type = delete` and `RC!=∞`
  - `RC=0`
- if `RC=0`: remove var. from DDE

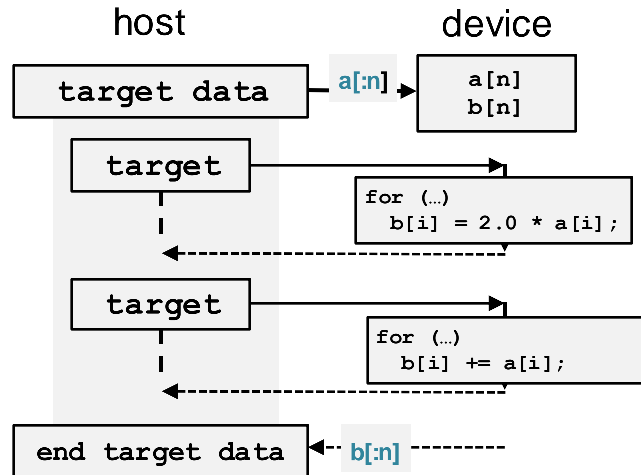
`mtm = map-type-modifier`

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

The Target Task

- **target task**:
  - device constructs and device memory API create a task
  - duty of this task: coordinate work between host and device
  - runs on the host
  - w/o nowait: included task → execution on host waits until task is completed
  - by default it is mergable and untied
  - spawned by: `target`, `target update`, `target enter data`, `target exit data`
- the target task completes when the work on the device is finished
- `nowait` clause makes this a deferrable task
  - host code does not wait for target task to complete
- `depend` clause(s) are "applied" to the target task
  - i.e. it can be used like any other task

> included task: execution is sequentially in the generation task region → it is undeferred

declare target clause

```
declare target
  globals, function definitions,
  function declarations
end declare target

declare target(list)
declare target clauses
```

C/C++ only

- map global/static variables to device
  - for the duration of the application
- map functions
  - generate a version for the target device
  - callable from the device

C/C++

```c
#pragma omp declare target
double sumit(double a, double b)
{ return a + b; }
#pragma omp end declare target

static double G = 1.23456;
#pragma omp declare target (G)

…
#pragma omp target
for (int i = 0; i < n; ++i) {
  c[i] = sumit(G, b[i]);
}
```

Fortran

```fortran
module xyz
  integer :: x
  !$omp declare target(x)
end module
```

```fortran
subroutine work(x)
  !$omp declare target
  integer, intent(in) :: x
  …
end subroutine
```

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`