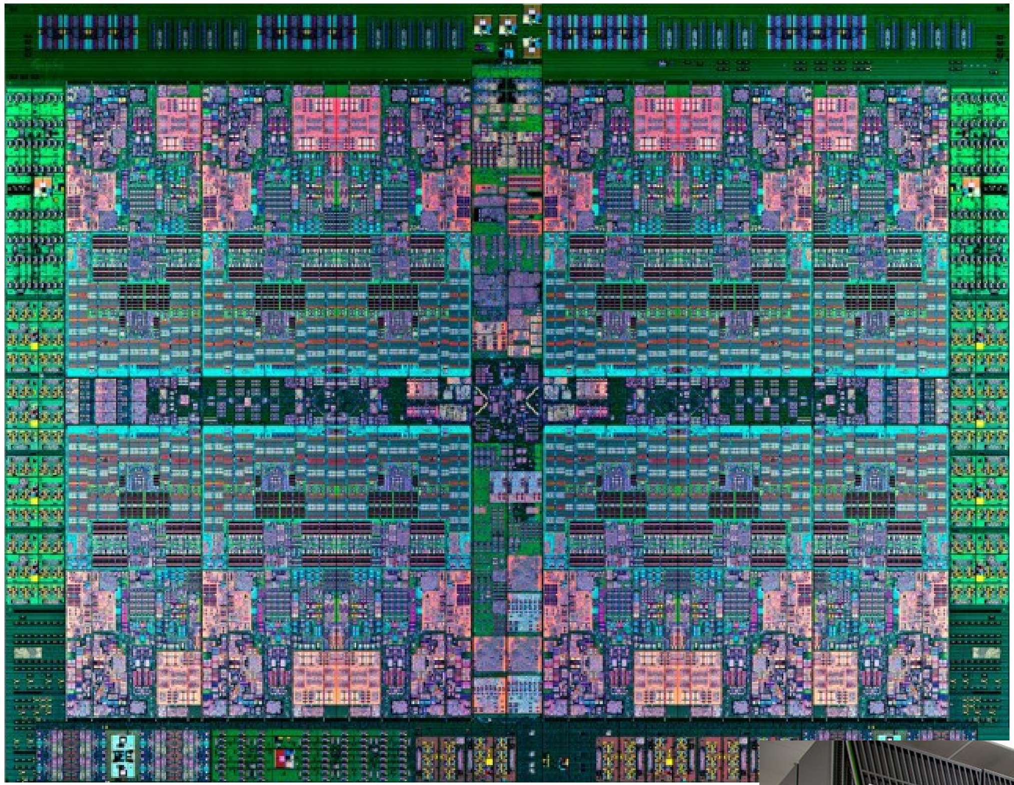
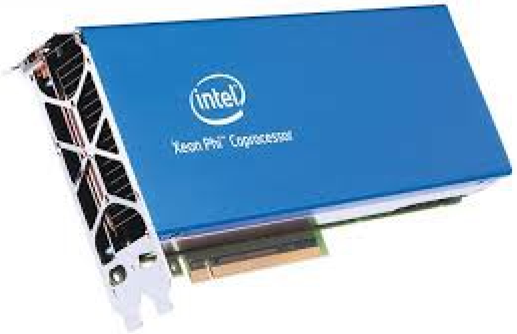
OpenMP 4.0 features

Barbara Chapman (University of Houston)
Deepak Eachempati (University of Houston)
Kelvin Li (IBM)

Aug 6, 2014



TESLA

intel
Xeon Phi Coprocessor

TITAN

IBM
MIRA
45

New features

- device constructs (a.k.a. accelerator support)
- array section syntax for C/C++
- SIMD constructs
- cancellation constructs (a.k.a. error model)
- thread affinity
- taskgroup construct
- task dependence
- user-defined reduction
- atomic construct extension
- OMP_DISPLAY_ENV environment variable
- partial Fortran 2003 support

New features

- device constructs (a.k.a. accelerator support)
- array section syntax for C/C++
- SIMD constructs
- cancellation constructs (a.k.a. error model)
- thread affinity
- taskgroup construct
- task dependence
- user-defined reduction
- atomic construct extension
- OMP_DISPLAY_ENV environment variable
- partial Fortran 2003 support

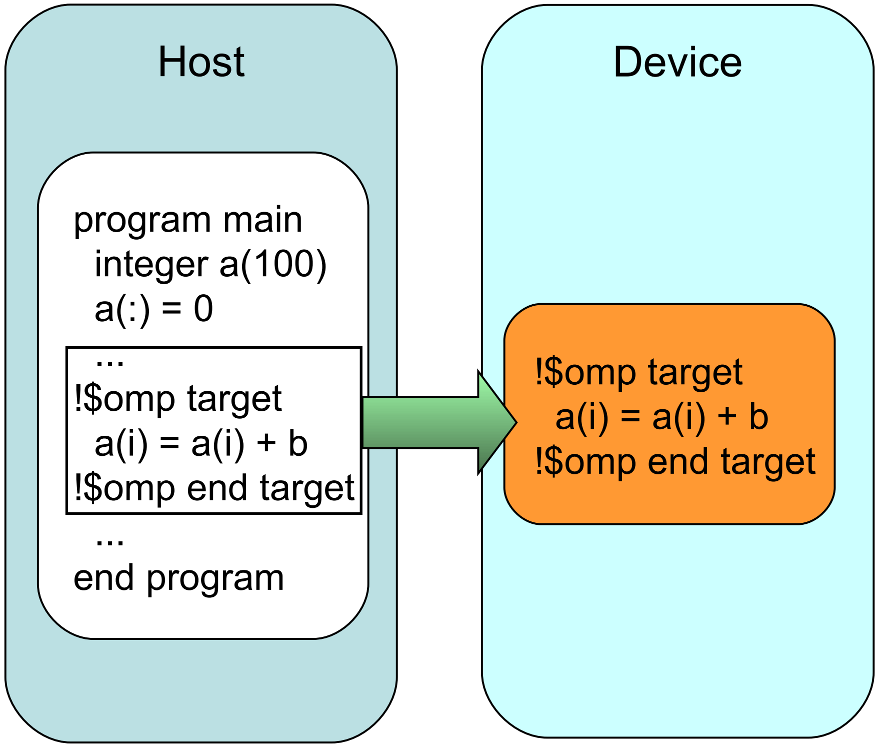
device constructs

- terminology
  - **device**: An implementation defined logical execution engine. (A device could have one or more processors.)
  - **host device**: The device on which the OpenMP program begins execution.
  - **target device**: A device onto which code and data may be offloaded from the host device. It is implementation defined (i.e. optional).

device constructs – execution model

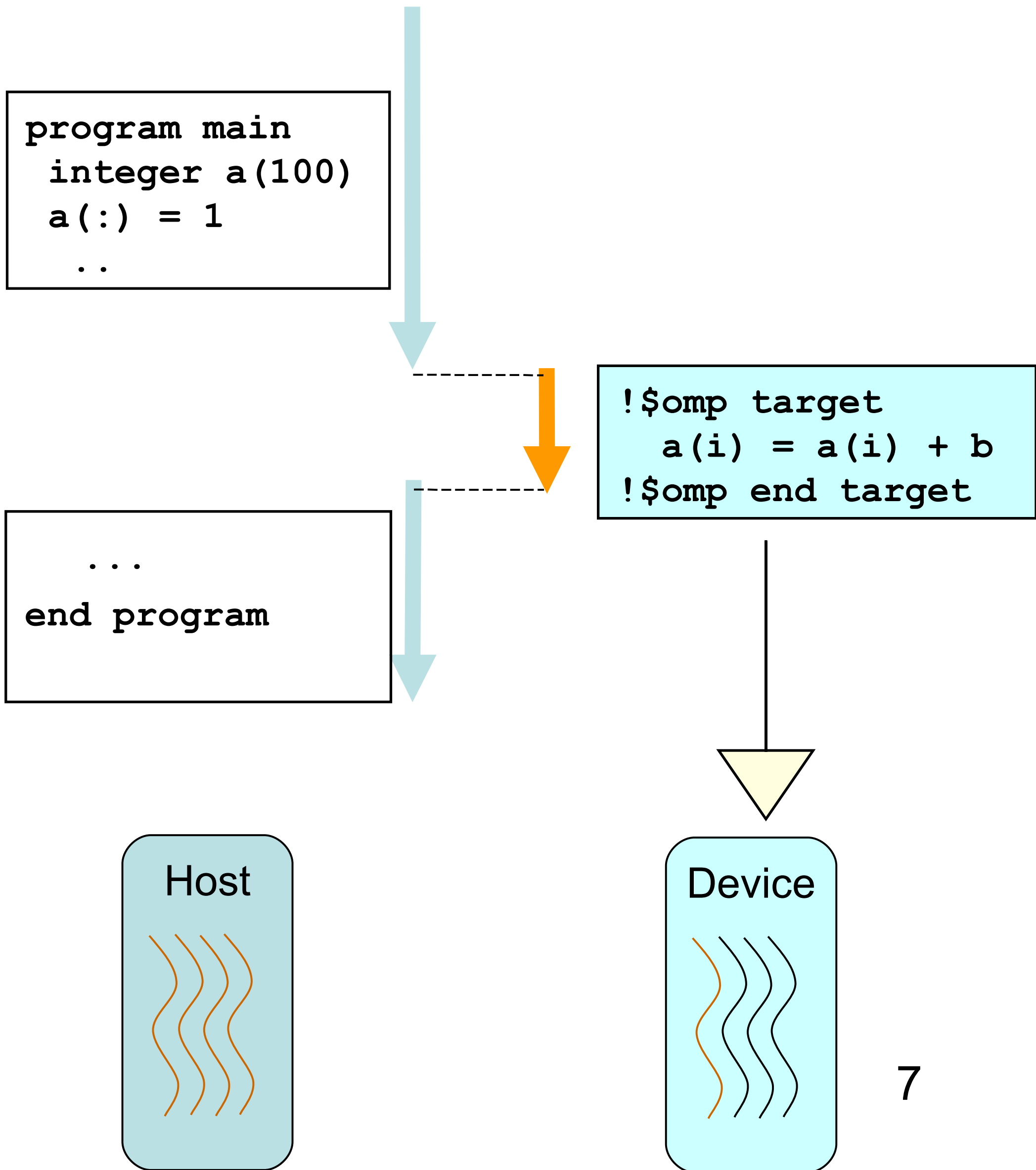
- host-centric; host device offloads target regions (code and data) to target devices
- the target region may be executed by the target device or host device
- the target region is executed by the host device if
  - a target device does not exist, or
  - the target device is not supported by the implementation, or
  - the target device cannot execute the target construct

```
!$omp target
  a(i) = a(i) + b
!$omp end target
```



device constructs – execution model

- the task that encounters the target construct waits at the end of the construct until execution of the region completes
  - it is synchronous
- a target region is executed on one target device
- each device has its own threads
  - no thread migration and no sharing among devices



```
program main
 integer a(100)
 a(:) = 1
   ..
```

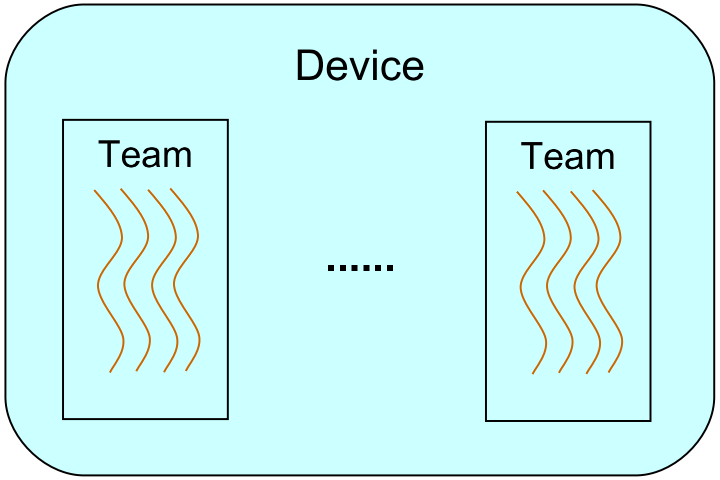
```
!$omp target
   a(i) = a(i) + b
!$omp end target
```

```
   ...
end program
```

device constructs – execution model

- the teams construct creates a league of thread teams
  - by default, one team is created (more about it later)
  - exploit extra level of parallelism on some hardware (e.g. Nvidia GPU)
  - work can be distributed among the teams (i.e. **`distribute`** construct ... more about it later)

```
#pragma omp target
#pragma omp teams
{
  for (i=1; i<N; i++)
    a[i] = a[i] + b;
}
```



device constructs – device data model

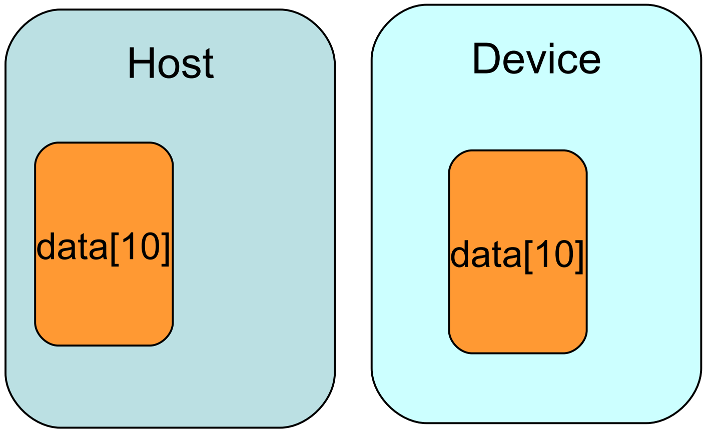
- very important to get the data right on host and target devices
- possible to be out of sync or unavailable
- data can implicitly mapped to the target region
- data can explicitly (via data-mapping attribute clause) mapped to the target region
  - provide more precise information to the compiler
  - reduce unnecessary data transfer – that is expensive!

`target data` construct

```
#pragma omp target data [ clause ... ] new-line
   structured-block
```

```
!$omp target data [ clause ... ]
   structured-block
!$omp end target data
```

- create a new device data environment
- executable statements inside a **`target data`** construct is executed on the host device
- a structured data mapping construct



`target` and **`target data`** constructs - clauses

```
clause:
•device(integer-expression)
•map([map-type:]list)
•if(scalar-expression)
```

- device clause
  - specify on which device the data environment is created as well as the code is executed if **`target`** is used
- if clause
  - if present and *false*, the device is the host
- map clause
  - *map-type*: **`alloc`**, **`to`**, **`from`**, **`tofrom`**
  - **`alloc`**: each new corresponding list item has an undefined initial value
  - **`to`**: each new corresponding list item is initialized with the original list item's value
  - **`from`**: on exit from the region the corresponding list item's value is assigned to each original list item
  - **`tofrom`**: each new corresponding list item is initialized with the original list item's value and that on exit from the region the corresponding list item's value is assigned to each original list item
  - default is **`tofrom`**

`target data` construct - example

- the **`target data`** region creates a new device environment
- variable **`N`** is mapped (**`tofrom`** map-type by default) to the target device and initialized
- variable **`i`** are mapped to the target device – to ensure that the variables are available
- the "**`i=10`**" is executed on the host device; i.e. variable **`i`** is initialized but not the corresponding variable on the target device

```
void foo()
{
  int N=0;
#pragma omp target data map(N)
{
  int i;

  i = 10;
}
}
```

target data construct - example

- if the system has multiple devices, different data can be mapped on different devices
- the **target data** region creates a new device environment on the device specified by the **device** clause
- variable **N** and **i** are mapped to device 1
- variable **M** and **k** are mapped to device 2

```
void foo()
{
  int N, M;
#pragma omp target data device(1) \
  map(N)
{
  int i
  ...
}

#pragma omp target data device(2) \
  map(M)
{
  int k;
  ...
}
}
```

there are rules for variables that are not listed on the **map** clause

`target data` construct - example

- use **`target data`** with the map clause to indicate how data is mapped particularly
- by default it is **`tofrom`**, data is transferred in to the target region upon entry and transfer out when exit
- use the default or **`tofrom`** type when you need the initial data in the region and also the resulting data after the region (i.e. the host can access the results from the target region)

```
subroutine compute_array()
{
  real arr0(100);
  arr0 = ...

!$omp target data map(arr0)
  ...
!$omp end target data

  ... = arr0
}
```

`target data` construct - example

- use **`target data`** with the map clause to indicate how data is mapped particularly
- the *map-type* is **`to`**
- data is transferred in to the target region but <u>not transferred out</u>
- after the **`target data`** region the value of **`arr0`** keeps the same as before entering the region

```
subroutine compute_array()
{
  real arr0(100);
  arr0 = ...

!$omp target data map(to: arr0)
  ...
!$omp end target data

  ... = arr0
}
```

`target data` construct - example

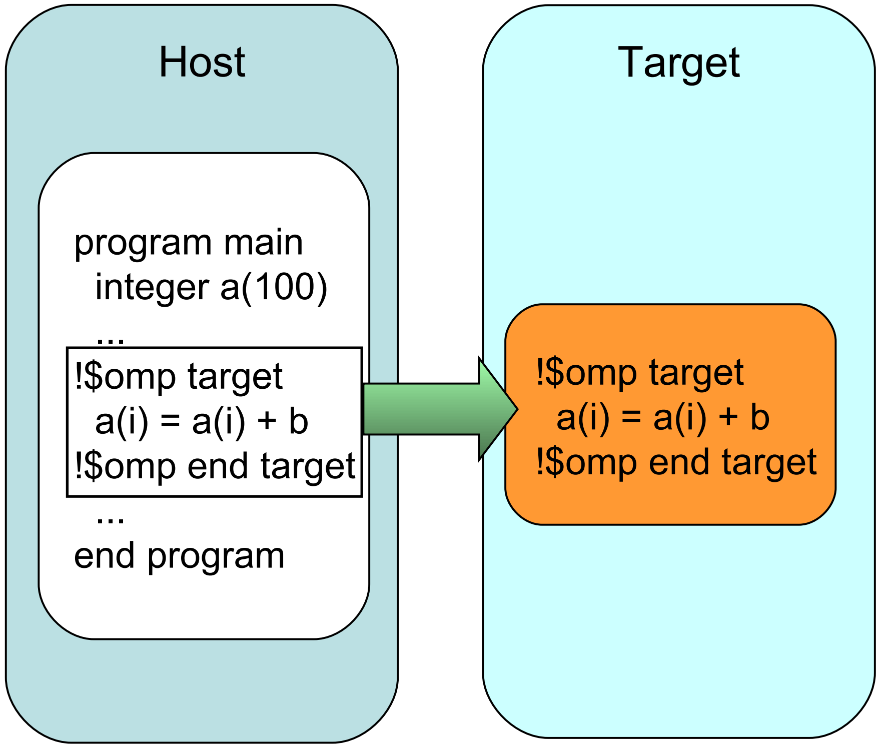
- use **`target data`** with the map clause to indicate how data is mapped particularly
- the *map-type* is **`from`**
- data is transferred out from the target region, but <u>not transferred in</u>

```
subroutine compute_array()
{
  real arr0(100);
  arr0 = ...

!$omp target data map(from: arr0)
  ...
!$omp end target data

  ... = arr0
}
```

`target` construct

- the code block enclosed by the target directives is offloaded and executed on the target device
- the compiler generates code of the target region for the target device
- the compiler will also generate the host version of the code – just in case, the target device is not available during the execution of the application
- the data required by the computation is expected to be available and up-to-date on the target device

```
!$omp target
  a(i) = a(i) + b
!$omp end target
```



`target` construct

```
#pragma omp target [ clause ... ] new-line
  structured-block
```

```
!$omp target [ clause ... ]
  structured-block
!$omp end target
```

- create a device data environment and <u>execute the construct on the same device</u> (superset of the functionality and restrictions provided by the target data directive)
- the encountering task waits for the device to complete the target region – synchronous operation

```
clause:
•device(integer-expression)
•map([map-type:]list)
•if(scalar-expression)
```

`target` construct

```
void update(int N, float *arr,
            float *brr, float C)
{
  int j;
#pragma omp target
{
  for (j=0; j<N; j++)
    arr[j] = C*arr[j] + brr[j];
}
}
```

arr, brr, C and j are implicitly mapped on the target device

it involves data transfer in and out

some data is read only in the target region, copyout is not necessary

- too much traffic between the host and the target devices
- need to reduce it!

`target` construct

```
void update(int N, float *arr,
            float *brr, float C)
{
  int j;
#pragma omp target map(tofrom:arr[0:N]) \
  map(in:C) map(in:brr[0:N])
{
  for (j=0; j<N; j++)
    arr[j] = C*arr[j] + brr[j];
}
}
```

arr is read and write in the target region

brr and C are read only in the target region

`target` construct - example

- the DO loop is offloaded to the target device
- the DO loop is executed sequentially on the target device
- does it work?
- *It works. But a lot of transfer is seen!!*
- why?
- data are implicitly mapped onto the target and needs to transfer in/transfer out

```
real function compute(N)
  integer, intent(in) :: N
  integer :: i, j, k
  real :: mat(N), c, psum

  psum = 0.0
!$omp target
  do i=1, N
    psum = psum + mat(k) + c
  enddo
!$omp end target

  compute = psum
end function
```

`target` construct - example

- use the **`target data`** directive to map the data on the target device
- map variables according to the activities in the region
- variable **`c`** and **`N`** are needed in the target region and it is reference only – we can just do **`map(to: c, N)`**
- variable **`mat`** needs to **`map(tofrom: mat)`** as the initial value is needed for the target region and the final value it needed after exit of the target region

```
real function compute(N)
  integer, intent(in) :: N
  integer :: i, j, k
  real :: mat(N), c, psum
!$omp target data &
!$omp& map(to: c, N) &
!$omp& map(from: psum) &
!$omp& map(tofrom: mat)
!$omp target
  psum = 0.0
  do i=1, N
    mat(i) = mat(i) + c
    psum = psum + mat(i)
  enddo
!$omp end target
!$omp end target data
  compute = psum
end function
```

`target` and `target data` construct - example

- so why is the **`target data`** directive needed?
- consider you have two target regions in your routine
- both regions need to access the same set of data
- what is the problem of having the two target regions and accessing array **`arr1`**?
- too much data transfer!

- enclosing the target regions by a target data region can guarantee data still alive on the target device

```
void foo() {
#pragma omp target data
{
  float arr1[500];

#pragma omp target
{
  for (int k=0; k<500; k++) {
    arr1[k] = ...
  }
}
  ...
#pragma omp target
{
  for (int k=0; k<500; k++) {
    arr1[k] = arr1[k] + 1.0;
  }
}
}}
```

`target` and **`target data`** constructs - example

- how about C/C++? can we do the same thing?
- everything looks the same – a straight translation from Fortran to C
- but it does not work!
- the compiler does not know how large array **`mat`** is
- need Fortran-like array syntax

```
float compute(int N, float *mat) {
  int i, j;
  float c, psum;
#pragma omp target data map(to:
   c,N) map(from: psum)
   map(tofrom: mat)
{
#pragma omp target
{
  psum = 0.0;
  for (int k=0;k<N;k++) {
    mat[k] += c;
    psum += mat[k];
  }
}}}
```

New features

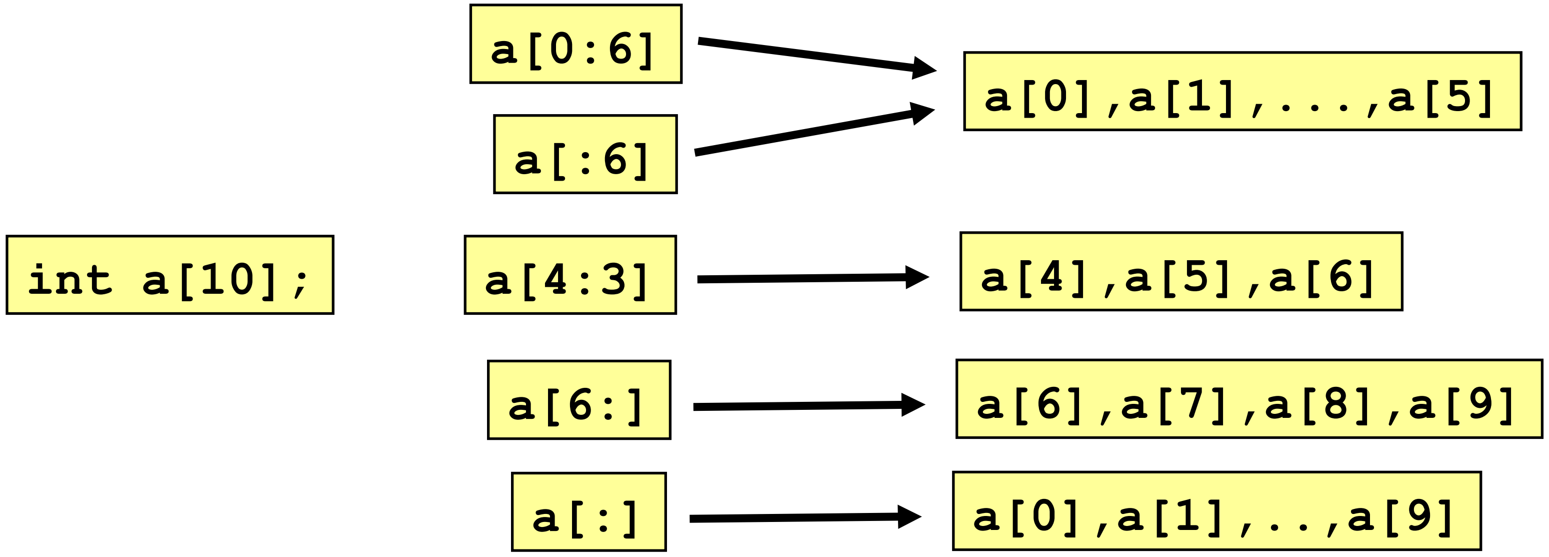
- device constructs (a.k.a. accelerator support)
- array section syntax for C/C++
- SIMD constructs
- cancellation constructs (a.k.a. error model)
- thread affinity
- taskgroup construct
- task dependence
- user-defined reduction
- atomic construct extension
- OMP_DISPLAY_ENV environment variable
- partial Fortran 2003 support

array section syntax for C/C++

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

- C and C++ array syntax extended to support array sections
- usage:
  - mark dependency of portion of an array in task dependence
  - can map a portion of array to target device (i.e. accelerator)
- array sections only applicable in the OMP pragma context
- array sections are only allowed in the following clauses:
  - **`to`** and **`from`** clause (**`target update`** directive)
    - must be contiguous
  - **`depend`** clause (**`task`** directive)
    - must not be zero-length
  - **`map`** clause (**`target`** and **`declare target`** directives)
    - must be contiguous

C/C++ array section syntax - examples



- `c[10][:][:0]` – a zero-length array
- `d[1:10][42][0:6]` – this array section is not contiguous
- `d[42][1:10][0:6]` – this array section is contiguous

`target` and `target data` constructs - example

- how to *fix* the compute routine in C?
- we need to specify the range of the array on the map clause
- **`mat[0:N]`** or **`mat[:N]`** is sufficient for the compiler to figure out

```
float compute(int N, float *mat)
{
  int i, j;
  float c, psum;

#pragma omp target map(to:c, N)\
  map(from: psum)\
  map(tofrom: mat[0:N])
{
  psum = 0.0;
  for (int k=0; k<N; k++) {
    mat[k] += c;
    psum += mat[k];
  }
}
  return psum;
}
```

`target` and `target data` constructs - example

- offloading to the target device may not always be beneficial, copying data is always costly
- if mat is large, the copy overhead may overwhelm performance benefit
- the **`if`** clause can be used to control **`if`** offloading to target device is beneficial

```
float compute(int N, float *mat) {
  int i, j;
  float c, psum;
#pragma omp target data map(to:
   c,N) map(from: psum)
   map(tofrom: mat[0:N]) if(N<500)
{
#pragma omp target if(N<500)
{
  psum = 0.0;
  for (int k=0;k<N;k++) {
    mat[k] += c;
    psum += mat[k];
  }
}}
  return psum;
}
```

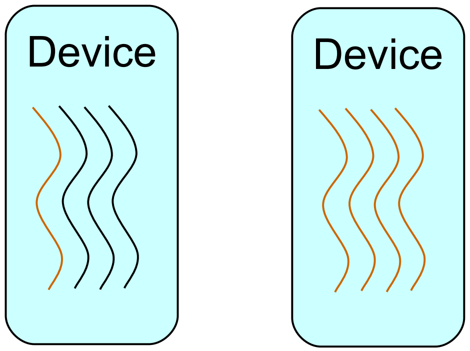
`target` and `target data` constructs - example

- if more than one target devices are supported and available, one can split the computation to the devices
- this example splits the computation onto two target devices – hence two target regions with device clause are created
- mapping has to be done accordingly

```
float compute(int N, float *mat) {
  int i, j, half;
  float c, psum[2];
  half = N/2;
  for (i=0; i<2; i++) {
    int start=i*half;
    int end=(i+1)*half-1;
#pragma omp target map(to: c,half)
   map(from: psum[i:1]) map(tofrom:
   mat[start:end]) if(half<250)
   device(i)
{
  psum[i] = 0.0;
  for (int k=start;k<=end;k++) {
    mat[k] += c;
    psum[i] += mat[k];
  }
}}
  return psum[0]+psum[1];
}
```

`target` constructs - example

- the for loop is executed sequentially on the target device
- may not fully exploit the parallelism available on the hardware
- parallel for loop can be added in the code to speed it up further



```
float compute(int N, float *mat) {
  int i, j;
  float c, psum;
#pragma omp target map(to: c,N)
   map(from: psum) map(tofrom:
   mat[0:N]) if(N<500)
{
  psum = 0.0;
#pragma omp parallel for
   reduction(+:psum)
  for (int k=0;k<N;k++) {
    mat[k] += c;
    psum += mat[k];
  }
}
  return psum;
}
```

`target` constructs - example

- we can further improve this compute routine to exploit more parallelism
- the execution of the `for` loop is independent – parallelize it
- achieve multiple levels of parallelism
  - on the host device
  - on the target device

```
float compute(int N, float *mat) {
  int i, j, half=N/2;
  float c, psum, sum[2];
#pragma omp parallel num_threads(2)
   private(psum)
{
  int nthd=omp_get_thread_num();
  int start=nthd*half;
  int end=(nthd+1)*half-1;
#pragma omp target
   map(to:start,end,c,half) map(from:
   psum) map(tofrom:mat[start:end])
   device(nthd)
{
  psum = 0.0;
#pragma omp parallel for
  for (int k=start;k<=end;k++) {
    mat[k] += c;
    psum += mat[k];
  }
}
  sum[nthd] = psum;
}
  return psum[0]+psum[1];}
```

`target update` construct

```
#pragma omp target update [ clause[[,] clause],... ] newline
```

```
!$omp target update [ clause[[,] clause],... ]
```

- makes the data consistent between the target device and the host device
- a standalone directive
- **`from`** clause – the value of the corresponding list item is assigned to the original list item
- **`to`** clause – the value of the original list item is assigned to the corresponding list item

motion-clause:
- **`to(`*`list`*`)`**
- **`from(`*`list`*`)`**

***clause*:**
- **`device(`*`integer-expression`*`)`**
- **`if(`*`scalar-expression`*`)`**
- ***motion-clause***

`target update` construct - example

- sometime we need to update the data either on the host device or on the target device
- the call statement is executed on the host – i.e. **`v1`**, **`v2`** may have different values from those on the target device
- the **`target update`** directive is to copy the up-to-date values in **`v1`** and **`v2`** to the target device

```
subroutine vec_mult(p, q, v1, v2, N)
  real :: p(N), q(N), v1(N), v2(N)
  integer :: i

!$omp target data map(to:v1,v2) map(from:p)
!$omp target
!$omp parallel do
  do i=1, N
    p(i) = v1(i)*v2(i)
  enddo
!$omp end target
  call init(v1,v2,N) ! executed on the host
!$omp target update to(v1,v2)
!$omp target
!$omp parallel do
  do i=1, N
    q(i) = v1(i)*v2(i)
  enddo
!$omp end target
!$omp end target data
end subroutine
```

`declare target` construct

```
#pragma omp declare target new-line
declarations-definition-seq
#pragma omp end declare target new-line
```

```
!$omp declare target ( list )
```

```
!$omp declare target
```

- specify variables (e.g. module variables and common blocks in Fortran; file scope or namespace scope variables in C/C++) and procedures being mapped to a device
- a declarative directive
- this directive instructs the compiler that the functions are called or the variables are referenced in the target region; a target device version of the code needs to be generated
- note that the Fortran and C/C++ syntax are quite different

`declare target` constructs - example

- note that the C/C++ syntax is different from Fortran, there is a declare target/end declare target pair
- **`fastAdd`** routine is called inside the target region
- the **`declare target`** pragma instructs the compiler to generate the target device version of the routine
- however, the compiler should also generate the host device version of the **`fastAdd`** routine

```
#pragma declare target
float fastAdd(float, float)
{ ... }
#pragma end declare target
float compute(int N, float *mat) {
  int i, j;
  float c, psum;

#pragma omp target map(to: c,N)
   map(from: psum) map(tofrom:
   mat[0:N]) if(N>50)
{
  psum = 0.0;
  for (int k=0;k<N;k++) {
    mat[k] = fastAdd(mat[k],c);
    psum = fastAdd(psum, mat[k]);
  }
}
  return psum;
}
```

`declare target` constructs - example

- note that the Fortran syntax is different from the C/C++ syntax
- **`fastAdd`** routine is called inside the target region
- the **`declare target`** directive instructs the compiler to generate the target device version of the routine
- however, the compiler should also generate the host device version of the **`fastAdd`** routine

```
real function fastAdd(x1, x2)
  real, intent(in) :: x1, x2
!$omp declare target
  ...
end function
real function compute(N)
  integer, value :: N
  real :: mat(N), c, psum
  interface
    real function fastAdd(x1,x2)
    real, intent(in) :: x1, x2
!$omp declare target
    end function
  end interface
!$omp target map(to:c,N) &
!$omp& map(from:psum) map(tofrom:mat)
  psum = 0.0
  do i=1, N
    mat(k) = fastAdd(mat(k), c)
    psum = fastAdd(psum, mat(k))
  enddo
!$omp end target
end function
```

declare target constructs - example

- in this example, variable **c** is a module variable, it needs to be mapped to the target device via the **declare target** directive
- the **fastAdd** function is a module procedure, the **declare target** directive is required as the function is called inside the target region

```
module M
  real :: c
!$omp declare target(c)
contains
  real function fastAdd(x1, x2)
  real, intent(in) :: x1, x2
!$omp declare target
  ...
  end function
end module
real function compute(N)
  use M
  integer, value :: N
  real :: mat(N), psum
!$omp target map(to:c,N) map
   (from:psum) map(tofrom:mat)
  psum = 0.0
  do i=1, N
    mat(k) = fastAdd(mat(k), c)
    psum = fastAdd(psum, mat(k))
  enddo
!$omp end target
end function
```

target data and target constructs – Jacobi example

```
#pragma omp target data device(gpu0) map(to:n, m, omega, ax, ay, \
   b, f[0:n][0:m])  map(tofrom:u[0:n][0:m]) \
   map(alloc:uold[0:n][0:m])
while ((k<=mits) && (error>tol))
{
  // a loop copying u[][] to uold[][] is omitted here
  ...
#pragma omp target device(gpu0) map(to:n, m, omega, ax, ay, b,\
  f[0:n][0:m], uold[0:n][0:m]) map(tofrom:u[0:n][0:m])
#pragma omp parallel for private(resid,j,i) reduction(+:error)
  for (i=1;i<(n-1);i++)
    for (j=1;j<(m-1);j++)
    {
      resid = (ax*(uold[i-1][j] + uold[i+1][j]) \
              + ay*(uold[i][j-1] + uold[i][j+1]) \
              + b * uold[i][j] - f[i][j])/b;
      u[i][j] = uold[i][j] - omega * resid;
      error = error + resid*resid ;
    } // the rest code omitted  ...
}
```

teams construct

```
#pragma omp teams [clause[[,] clause],...] new-line
structured-block
```

```
!$omp teams [clause[[,] clause],...]
structured-block
!$omp end teams
```

- creates a league of thread teams
- the master thread of each team executes the region
- allow to exploit one more level of parallelism on some target devices (e.g. Nvidia)
- no implicit barrier at the end of a teams construct
- no statement/directive is allowed between **target** construct and **teams** construct

clause:

- `num_teams(`*`integer-expression`*`)`
- `thread_limit(`*`integer-expression`*`)`
- `[C/C++]`
  `default(shared|none)`
  `[Fortran]`
  `default(shared|none|private|firstprivate)`
- `private(`*`list`*`)`
- `firstprivate(`*`list`*`)`
- `shared(`*`list`*`)`
- `reduction(`*`reduction-identifer`*`:`*`list`*`)`

teams construct – runtime routines

```
int omp_get_num_teams(void);
```

```
integer function omp_get_num_teams()
```

- returns the number of teams in the current **teams** region

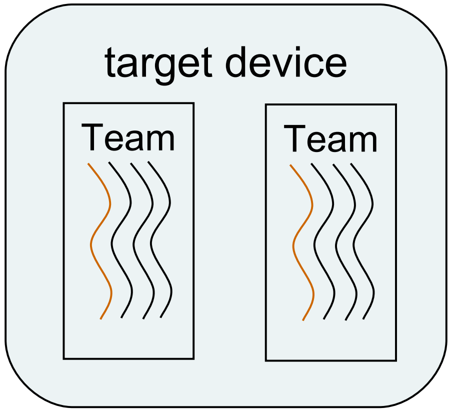
```
int omp_get_team_num(void);
```

```
integer function omp_get_team_num()
```

- returns the number of the team number of the calling thread

`teams` construct – example

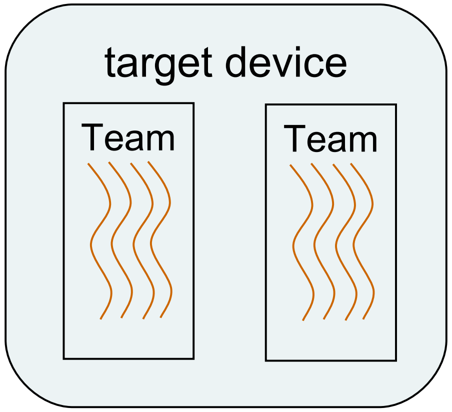
- two **`teams`** regions are created inside the **`target`** region
- each **`teams`** region compute a portion of the array operation
- note that since there is only one team region, the entire array **`mat`** is mapped to the target device



```
real function compute(N)
  integer :: i, N, ntm, start, end
  real :: mat(N), psum(2)
!$omp target map(to:c,N) &
!$omp& map(from:psum) map(tofrom:mat)
!$omp teams num_teams(2) &
!$omp& private(i,ntm,start,end)
  ntm = omp_get_team_num()+1
  psum(ntm) = 0.0
  start = (ntm-1)*(N/2)
  end = ntm*(N/2) - 1
  do i=start, end
    mat(i) = mat(i) + c;
    psum(ntm) = psum(ntm) + mat(i)
  enddo
!$omp end teams
!$omp end target
  compute = sum(psum)
end function
```

`teams` construct – example

- more parallelism can be exploited in the team regions since there are multiple threads in a team



```
real function compute(N)
  integer :: i, N, ntm, start, end
  real :: arr(N), psum
  psum = 0.0
!$omp target map(to:c,N) &
!$omp& map(tofrom:mat)
!$omp teams num_teams(2) &
!$omp& private(i,ntm,start,end) &
!$omp& reduction(+:psum)
  ntm = omp_get_team_num() + 1
  start = (ntm-1)*(N/2)
  end = ntm*(N/2) - 1
!$omp parallel do reduction(+:psum)
  do i=start, end
    mat(i) = mat(i) + c;
    psum = psum + mat(i)
  enddo
!$omp end teams
!$omp end target
  compute = psum
end function
```

`distribute` construct

```
#pragma omp distribute [clause[[,] clause],...] new-line
structured-block
```

```
!$omp distribute [clause[[,] clause],...]
structured-block
[ !$omp end distribute ]
```

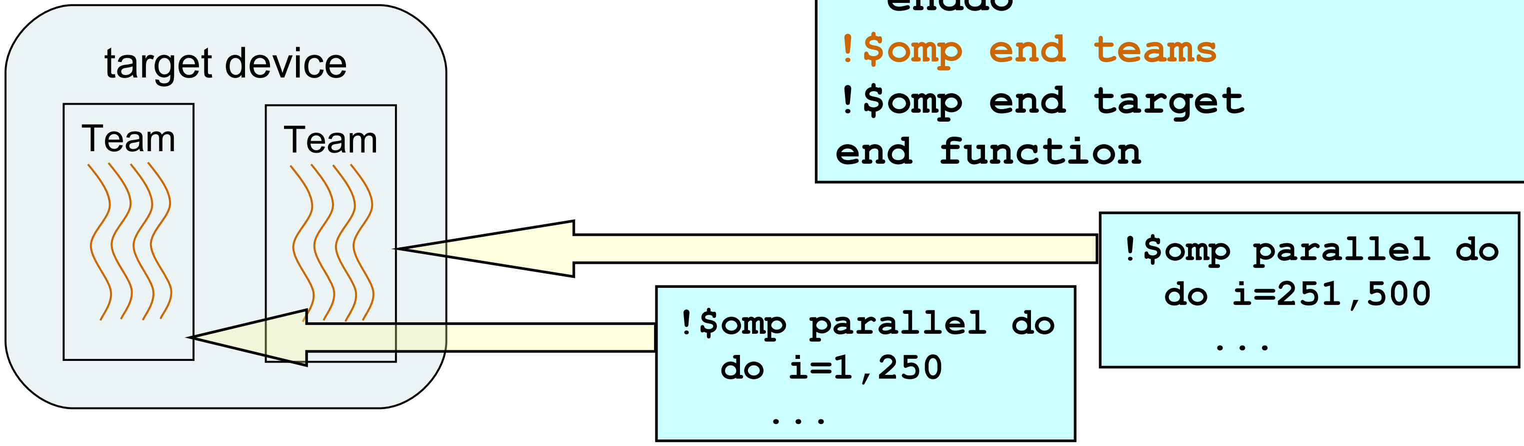
- iterations distributed among master threads of all teams
- specify to the loops only
- must be closely nested to the **`teams`** construct
- workshare among teams to exploit the parallelism on the target device

clause:
- **`private(`***`list`***`)`**
- **`firstprivate(`***`list`***`)`**
- **`collapse(`***`n`***`)`**
- **`dist_schedule(`***`kind[,chunk_size]`***`)`**

`distribute` construct – example

- the iterations are divided into blocks, each block is offloaded to the **`teams`** regions
- the **`distribute`** directive share the iterations among teams in the target region

```
function dotprod(B,C,N)
   real :: B(N), C(N), sum
   integer :: N, i, j
   sum = 0.0
!$omp target map(to:B,C)
!$omp teams num_teams(2) &
!$omp& reduction(+:sum)
!$omp distribute
   do j=1, N, N/2
!$omp parallel do reduction(+:sum)
      do i=j, min(j+N/2,N)
         sum = sum + B(i)*C(i)
      enddo
   enddo
!$omp end teams
!$omp end target
end function
```



```
!$omp parallel do
  do i=1,250
    ...
```

```
!$omp parallel do
  do i=251,500
    ...
```

New features

- device constructs (a.k.a. accelerator support)
- array section syntax for C/C++
- SIMD constructs
- cancellation constructs (a.k.a. error model)
- thread affinity
- taskgroup construct
- task dependence
- user-defined reduction
- atomic construct extension
- OMP_DISPLAY_ENV environment variable
- partial Fortran 2003 support

`simd` construct

```
#pragma omp simd [clause[[,] clause],...] new-line
for-loops
```

```
!$omp simd [clause[[,] clause],...]
do-loops
[ !$omp end simd ]
```

- a level of parallelism that exploits the hardware feature – instruction level parallelism (ILP)
- enable the execution of multiple iterations of the loops concurrently by means of SIMD instructions

clause:
- `safelen(`*length*`)`
- `linear(`*list[:linear-step]*`)`
- `aligned(`*list[:alignment]*`)`
- `private(`*list*`)`
- `lastprivate(`*list*`)`
- `reduction(`*reduction-identifier:list*`)`
- `collapse(`*n*`)`

`simd` construct

- terminology
  - SIMD instruction: A single machine instruction that can operate on multiple data elements.
  - SIMD lane: A software or hardware mechanism capable of processing one data element from a SIMD instruction.
  - SIMD chunk: A set of iterations executed concurrently, each by a SIMD lane, by a single thread by means of SIMD instructions.
  - SIMD loop: A loop that includes at least one SIMD chunk

`simd` construct – clause

`safelen(`*`length`*`)`

- no two iterations executed concurrently with SIMD instructions can have a greater distance in the logical iteration space than *n*, where *n* is an integer constant
- note: the number of iterations that are executed concurrently at any given time is implementation defined

```
#pragma omp simd safelen(4)
for (int i=0; i<N; i++)
   a[i] = a[i+4] + 0.5;
```

fe to vectorize
length
ss

a[0]=... and
a[4]=... should
not execute
concurrently

```
a[0] = a[4] + 0.5;
a[1] = a[5] + 0.5;
a[2] = a[6] + 0.5;
a[3] = a[7] + 0.5;
a[4] = a[8] + 0.5;
...
```

can use vector of length 4 or less

simd construct – clause

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

- has **private** clause semantics, and also firstprivate and lastprivate semantics

-
  ```
  int x=2;
  #pragma omp simd linear(x:
  4)
    for (int i=0; i<12; i++)
      ... = x;
  printf("%d\n", x);
  ```
- in each iteration, private x is initialized as x = x0 + i * 4, where x0 is the initial value of x before entering the SIMD construct
- the value of x in the sequentially last iteration is assigned to the original list item

```
#pragma omp simd linear(i:1)
for (i=0; i<N; i++)
  a[i] = b[i] * c[i];
```

i is incremented by 1 in every iteration

```
m = 1;
#pragma omp simd linear(m:2)
for (i=0; i<N; i++)
  a[i] = a[i]*m;
```

```
a[0] = a[0]*(1+0*2);
a[1] = a[1]*(1+1*2);
a[2] = a[2]*(1+1*3);
...
```

`simd` construct – clause

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

- **`aligned(x:8)`** – variable **`x`** is aligned to 8 byte
- if no alignment is specified in the clause, the alignment is implementation defined

`privat(`*`list`*`)`

- private to a SIMD lane

`lastprivate(`*`list`*`)`

- private to a SIMD lane and copy the value in the last iteration back to the original list item

`reduction(`*`reduction-identifier:list`*`)`

- each SIMD lane has a private copy and values of the private copies are combined at the end

`collapse(`*`n`*`)`

- associate the nested loops with the construct

declare simd construct

```
#pragma omp declare simd [clause[[,] clause],...] new-line
[#pragma omp declare simd [clause[[,]clause],...] new-line]
[...]
function definition declaration
```

```
!$omp declare simd(proc-name) [clause[[,] clause],...]
```

- specify for functions/subroutines
- create "one or more versions that can process multiple arguments using SIMD instructions from a single invocation from a SIMD loop"
- a function can be specified by multiple **declare simd** directives
  – allow different combinations of SIMD length and other attributes (uniform, linear, ... etc)

clause:
- **simdlen(*length*)**
- **linear(*list[:linear-step]*)**
- **aligned(*list[:alignment]*)**
- **uniform(*argument-list*)**
- **inbranch**
- **notinbranch**

`declare simd` construct – clause

```
#pragma omp declare simd \
 simdlen(4)
#pragma omp declare simd \
 simdlen(8)
void foo(float a);
```

```
void __simd_f4_foo(float a1,
float a2, float a3, float a4);

void __simd_f8_foo(float a1,
float a2, ..., float a8);
```

`simdlen(`***`length`***`)`

- ***`length`*** is the number of concurrent arguments of the function to be created
- if the **`simdlen`** clause is not specified, the number of concurrent arguments is implementation defined

`uniform(`***`argument-list`***`)`

- arguments that have an invariant value for all concurrent invocations of the function in the execution of a single SIMD loop

`inbranch / notinbranch`

- function always / never called from inside a conditional statement in a SIMD loop

loop SIMD construct

```
#pragma omp for simd [clause[[,]clause]...] new-line
for-loops
```

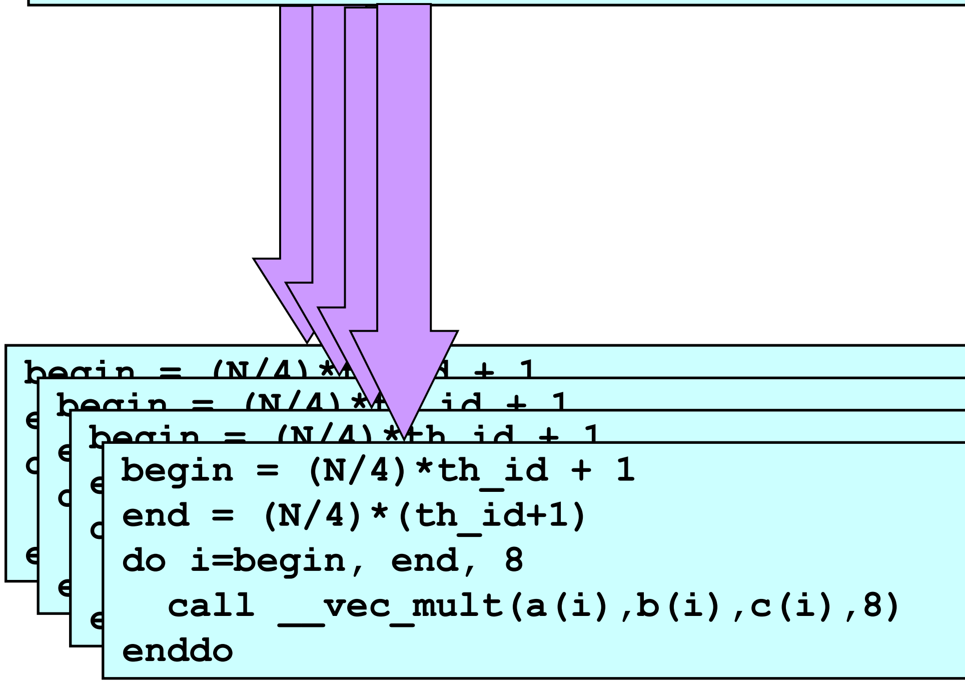
```
!$omp simd do [clause[[,] clause],...]
do-loops
[!$omp end do simd [nowait]]
```

- combine worksharing loop construct and the SIMD construct
  - executed concurrently using SIMD instructions
  - executed in parallel by threads in the team
- how it is done:
  - distribute the iterations across the implicit tasks (i.e. threads)
  - the chunks of iterations will then be converted to a SIMD loop
- ordered clause is not allowed

loop SIMD construct

- each implicit task (i.e. thread) takes on a chunk of 40 iterations
- the chunk is converted to the SIMD loop with safelen of 8 (i.e. no two iterations executed concurrently with SIMD instructions can have a greater distance in the logical iteration space than 8)
- it achieves multiple levels of parallelism

```
!$omp do simd num_threads(4) &
!$omp& safelen(8) private(i)
do i=1, N
  a(i) = b(i)*c(i)
enddo
```



```
begin = (N/4)*th_id + 1
end = (N/4)*(th_id+1)
do i=begin, end, 8
  call __vec_mult(a(i),b(i),c(i),8)
enddo
```

device and SIMD constructs – combination of constructs

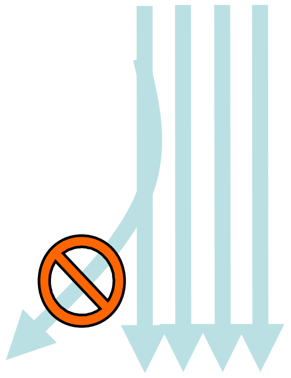
- **`distribute simd`**
- **`distribute parallel for/do`**
- **`distribute parallel for/do simd`**
- **`parallel for/do simd`**
- **`target teams`**
- **`teams distribute`**
- **`teams distribute simd`**
- **`target teams distribute`**
- **`target teams distribute simd`**
- **`teams distribute parallel for/do`**
- **`target teams distribute parallel for/do`**
- **`teams distribute parallel for/do simd`**
- **`target teams distribute parallel for/do simd`**

New features

- device constructs (a.k.a. accelerator support)
- array section syntax for C/C++
- SIMD constructs
- cancellation constructs (a.k.a. error model)
- thread affinity
- taskgroup construct
- task dependence
- user-defined reduction
- atomic construct extension
- OMP_DISPLAY_ENV environment variable
- partial Fortran 2003 support

`cancellation` constructs

- in pre-4.0, a parallel region or worksharing region cannot be stopped at the middle of the execution and continue the program
- it must execute to the end of the region
- what if a computation (a search) wants to exit the region after a certain goal is achieved



`cancel` construct

```
#pragma omp cancel construct-type-clause[[,] if-clause] new-line
```

```
!$omp cancel construct-type-clause[[,] if-clause]
```

- cancels a region and causes the execution jump to the end of the canceled region
- applies to the innermost enclosing OpenMP construct of the type specified in the clause

construct-type-clause:
- `parallel`
- `sections`
- `for / do`
- `taskgroup`

if-clause:
- `if(`*scalar-expression*`)`

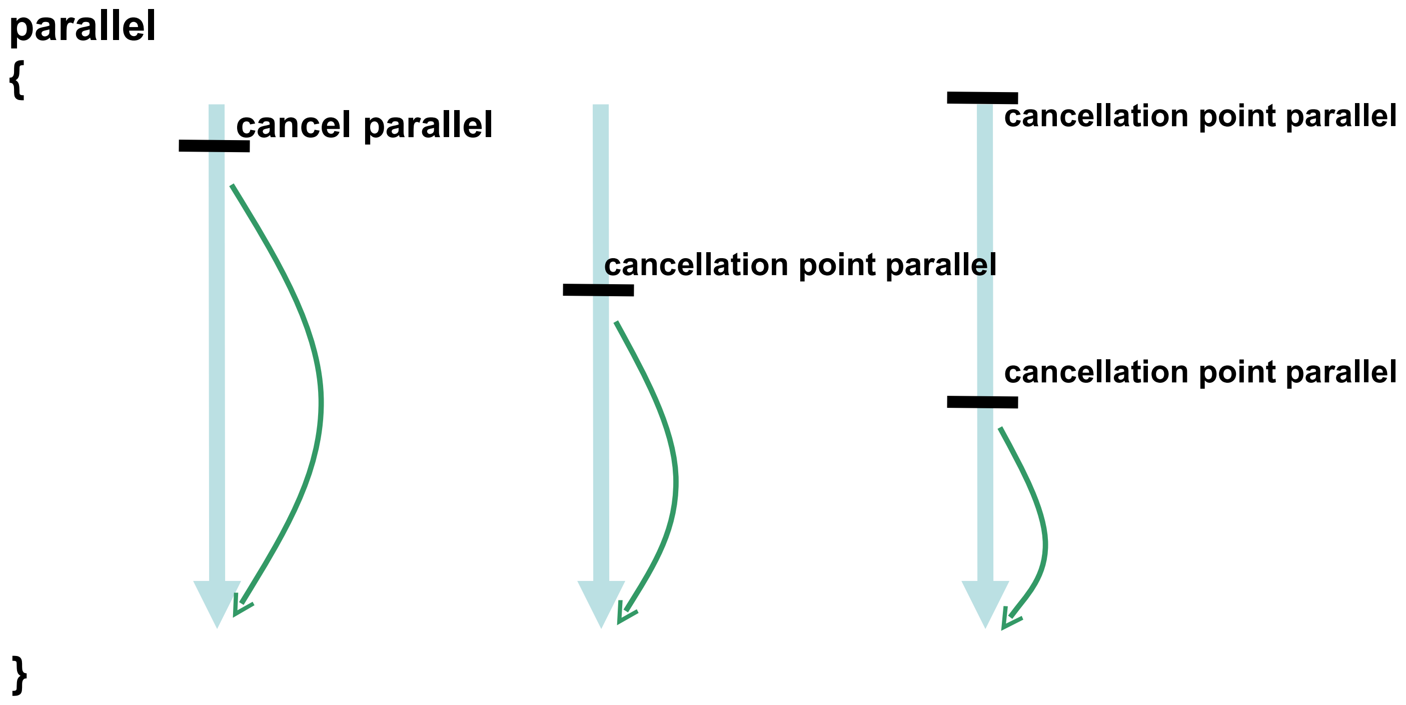
```
int omp_get_cancellation(void);
```

```
logical function omp_get_cancellation()
```

- returns if cancellation is activated for the whole program

`cancel` construct

- for parallel, sections or loop regions
  - "cause the encountering task to continue to execute at the end of the canceled construct"
  - other threads check at cancellation points if cancellation has been requested, jumps to the end of the cancelled construct


parallel
{
cancel parallel
cancellation point parallel
cancellation point parallel
cancellation point parallel
}

`cancel` construct

```
int omp_get_cancellation(void);
```

```
logical function omp_get_cancellation()
```

- returns if cancellation is activated for the whole program

```
OMP_CANCELLATION=true|false
```

- controls whether the effects of **`cancel`** construct and of cancellation points are enabled and cancellation is activated
- allow implementations to bypass the expensive runtime check to reduce the overhead of this feature

`cancel` construct

```
#pragma omp parallel
{
   ...
#pragma omp cancel parallel
   ...
#pragma omp sections
{
   ...
}
   ...
#pragma omp cancel sections
   ...
}
```

closely nested ... can cancel the enclosing parallel region

invalid ... not closely nested in a sections construct

closely nested inside an OpenMP construct that matches the clause(s) specified

`cancel` construct

```
#pragma omp parallel
{
   ...
   #pragma omp parallel
   {
   ...
   }
   ...
   #pragma omp cancel parallel
   ...
}
```

`may not be canceled`

`effective to the innermost enclosing parallel region`

applies to the innermost enclosing OpenMP construct of the type specified in the clause

`cancel` construct

```
void compute()
{
   ...
#pragma omp cancel parallel
}

int main()
{
#pragma omp parallel
{
   ...
   compute();
   ...
}}
```

invalid ... must be lexically nested in the type of construct specified in the clause

`cancellation point` construct

```
#pragma omp cancellation point construct-type-clause new-line
```

```
!$omp cancellation point construct-type-clause
```

- cancellation point – at which a check is done if cancellation has been requested; if cancellation has been requested, cancellation is performed.
  - implicit barriers (i.e. end of worksharing or parallel regions)
  - **`barrier`** regions
  - **`cancel`** regions
  - **`cancellation point`** regions
- allow users to define a cancellation point and specify the construct type

construct-type-clause:
- **`parallel`**
- **`sections`**
- **`for / do`**
- **`taskgroup`**

`cancel` construct - example

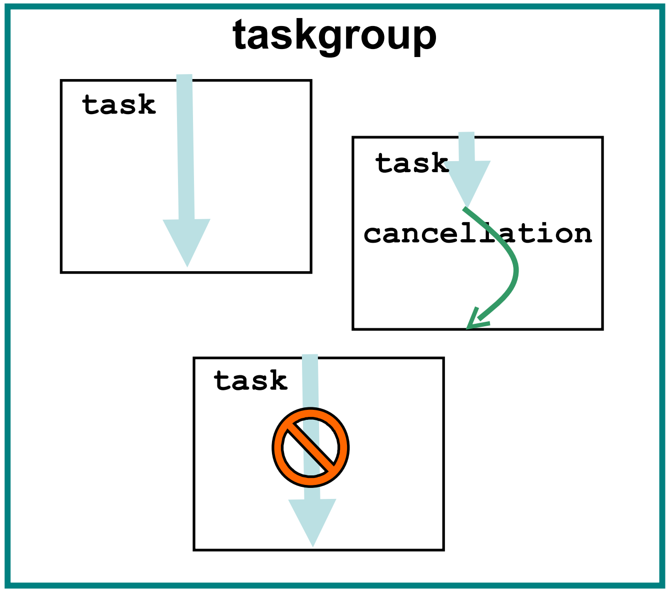
```
#pragma omp parallel private(found)
{
  int id = omp_get_thread_num();
  for (int db=0; db<N; db++)
  {
#pragma omp cancellation point
   parallel
    found = doSearch(id, db);

    if (found)
    {
#pragma omp cancel parallel
        ...
    }
  }
}
```

other threads: check if cancellation has been requested jumps to the end of the parallel regio

current thread: jump to the end of the parallel region

`cancel` construct

- for taskgroup
  - the task that encountered the **`cancel taskgroup`** construct continues execution at the end of its task region
  - any task has already begun execution must run to completion or until a cancellation point is reached
  - any task that has not begun execution may be discarded



`cancel` construct

```
!$omp parallel
!$omp single
!$omp taskgroup
  do while (.true.)
!$omp task
    call search(found)
    if (found) then
!$omp cancel taskgroup
    endif
!$omp end task
  enddo
!$omp end taskgroup
!$omp end single
!$omp end parallel
```

- one thread generating tasks
- when the search routine returns true and cancellation is requested, cancellation of tasks is activated
- the current task construct continues execution at the end of its task region
- the executing task will continue to execute until completion or encountering a cancellation point
- the queued task can be discarded

`cancel` construct

```
#pragma omp taskgroup
{
  #pragma omp task  // long running task
  {
     ...
  }
  #pragma omp task
  {
    if (done) {
      // clean up, print results etc.
      #pragma omp cancel taskgroup
    }
    ...
  }
  while (...) {
    #pragma omp task
    {
      #pragma omp cancellation point
      if (done) {
        #pragma omp cancel taskgroup
      }
    }
  }
}
```

continue to run to completion or a cancellation point encountered

skip if a cancellation is requested

`cancel` construct

- barrier
  "Each barrier region must be encountered by all threads or by none at all*, unless cancellation has been requested for the innermost enclosing parallel region*."
- worksharing region
  "Each worksharing construct must be encountered by all threads in a team or by none at all*, unless cancellation has been requested for the innermost enclosing parallel region*."

New features

- device constructs (a.k.a. accelerator support)
- array section syntax for C/C++
- SIMD constructs
- cancellation constructs (a.k.a. error model)
- thread affinity
- taskgroup construct
- task dependence
- user-defined reduction
- atomic construct extension
- OMP_DISPLAY_ENV environment variable
- partial Fortran 2003 support

Thread affinity

- support thread affinity policy
- allow users to have finer control how and where the OpenMP threads are bound
  - better locality between OpenMP threads
  - less false sharing
  - more memory bandwidth
- in pre-4.0, it only supports true or false for the environment variable **`OMP_PROC_BIND`** that is very limited
- in 4.0, this feature extends to a full range of specification

Acknowledgement: slides adapted from Alex Eichenberger (IBM)

Thread affinity

- OpenMP places
  - one or more processors / hardware threads per place
  - OpenMP threads are allowed to move within a place
  - OpenMP threads are not allowed to move between places
- Affinity policies
  - close: assign the threads to places close to the place of the parent thread
  - master: assign every thread in the team to the same place as the master thread
  - spread: create a sparse distribution of threads

Thread affinity – clause and env var

```
proc_bind(close | master | spread)
```

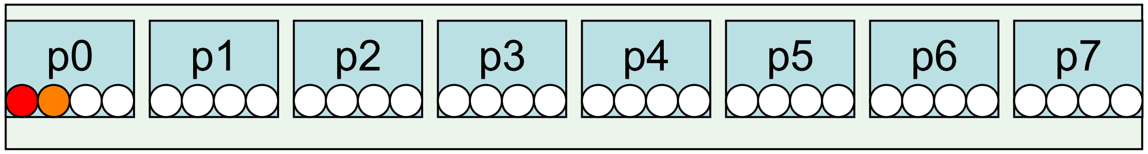
- add **`proc_bind`** clause to the parallel directive

```
OMP_PROC_BIND=policy[, policy ...]
```

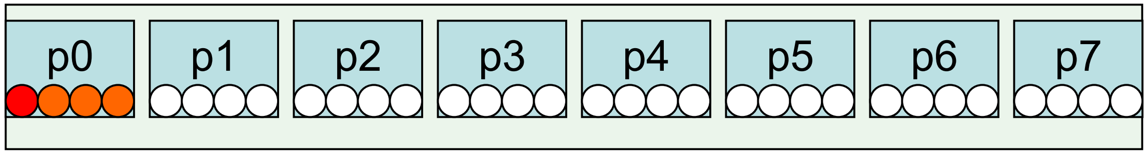
- extend the OMP_PROC_BIND environment variable to allow more policies
  i.e. **`true`**, **`false`**, **`close`**, **`master`**, or **`spread`**

Thread affinity

- Master affinity
  - for best data locality
  - assign OpenMP threads in the same place as the master
- **`parallel proc_bind(master)`**
  - master 2



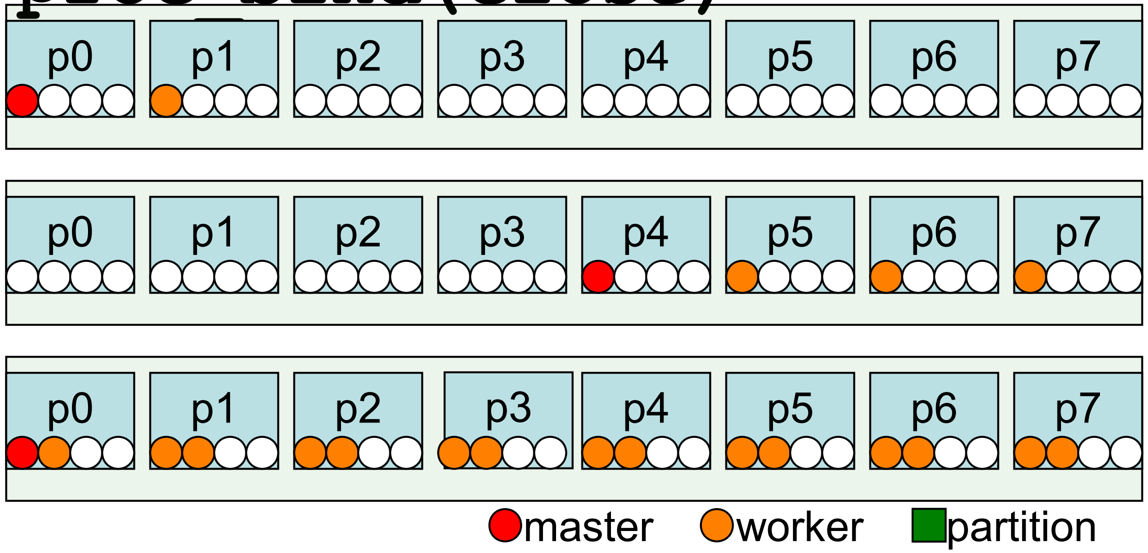
  - master 4



🔴master 🟠worker 🟩partition

Thread affinity

- Close affinity
  - for data locality, load-balancing, and more dedicated-resources
  - assign OpenMP threads near the place of the master
  - wrap around once each place gets one thread
- **`parallel proc_bind(close)`**
  - close 2
  - close 4
  - close 16



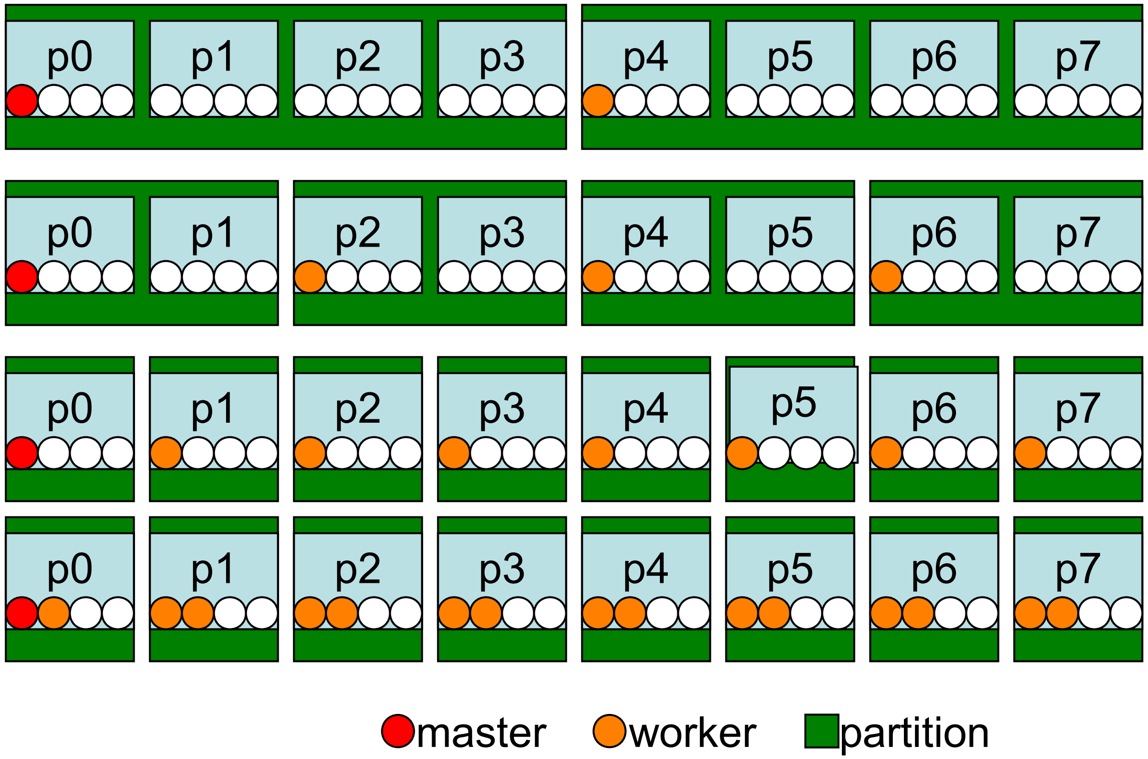
Just an exa
this is Neste
OpenMP

Thread affinity

- Spread affinity
  - for load balancing, most dedicated hardware resources
  - spread OpenMP threads as evenly as possible among places
  - create sub-partition of the place list
    - subsequent threads will only be allocated within sub-partition

Thread affinity

- Examples of “parallel proc_bind(spread)”
  - spread 2
  - spread 4
  - spread 8
  - spread16

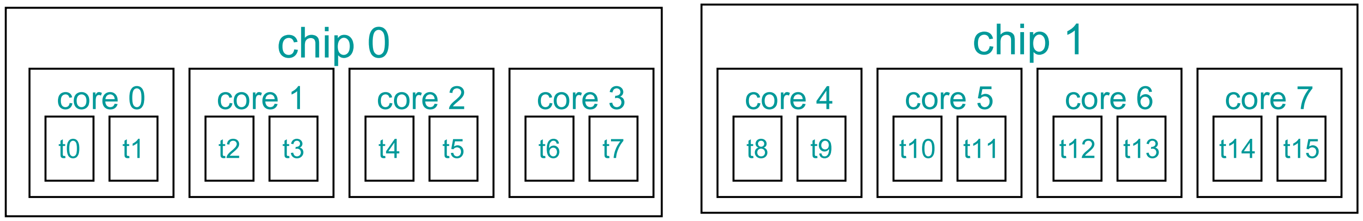


Thread affinity

- Add OMP_PLACES environment variable
- A few examples
  - OMP_PLACES="{0},{1},{2},{3},...,{15}"
  - OMP_PLACES="{0:4},{4:4},{8:4},{12:4}"
  - OMP_PLACES="threads(16)"

Thread affinity

- How to use Place list
- Consider a system with 2 chips, 4 cores and 8 hardware-threads



- One place per hardware-thread
  - OMP_PLACES="{0},{1},{2},...{15}"
  - OMP_PLACES="threads(16)"   # 16 threads
  - OMP_PLACES="threads"   # as many threads as available
- One place per core, including both hardware-threads
  - OMP_PLACES="{0,1},{2,3},{4,5},{6,7},{8,9},{10,11},{12,13},{14,15}"
  - OMP_PLACES="{0:2}:8:2"
  - OMP_PLACES="cores(4)"   # 4 cores
  - OMP_PLACES="cores"   # as many cores as available
- One place per chip, excluding one hardware-thread per chip
  - OMP_PLACES="{1:7},{9:7}"

New features

- device construct s(a.k.a. accelerator support)
- array section syntax for C/C++
- SIMD constructs
- cancellation constructs (a.k.a. error model)
- thread affinity
- taskgroup construct
- task dependence
- user-defined reduction
- atomic construct extension
- OMP_DISPLAY_ENV environment variable
- partial Fortran 2003 support

`taskgroup` construct

```
#pragma omp taskgroup new-line
```

```
!$omp taskgroup
structured-block
!$omp end taskgroup
```

- for deep synchronization against hierarchies of tasks
  - "specifies a wait on completion of child tasks of the current task <u>and their descendant tasks</u>"
- note: **`taskwait`** construct is shallow – just wait on the completion of the child tasks of the current task (i.e. siblings not their descendants)
- does not create a task region; only for group and synchronization purpose

```
#pragma omp task    // T1
{
  #pragma omp task // T2
  {
     ...
  }
}
#pragma omp task    // T3
{
    ...
}
#pragma omp taskwait
```

wait for T1 and T3 but not T2

`taskgroup` construct – example

```
subroutine sub1(N)
  do i=1, N
!$omp task
   ...
!$omp end task
  end do
end subroutine

subroutine sub2()
!$omp task
  ...
!$omp end task
end subroutine
```

```
!$omp taskgroup
!$omp task
!$omp task    ! T1
  call sub1(i)
!$omp end task
...
!$omp task    ! T2
  call sub2()
!$omp end task
!$omp taskwait
!$omp end task
!$omp end taskgroup
```

only wait for T1 and T2

wait for T1, T2 and all the tasks in sub1 and sub2

taskgroup construct - example

```
void compute_tree(tree_t tree)
{
  if (tree->left) {
   #pragma omp task
     compute(left);
  }

  if (tree->right) {
   #pragma omp task
     compute(right);
  }
}
```

```
int main() {
  int i;
  tree_t tree;
  init_tree(tree);

  #pragma omp parallel
  {
    #pragma omp task
      start_background_work();

    #pragma omp master
      for (i=0; i<max_steps; i++) {
        #pragma omp taskgroup
        {
          #pragma omp task
            compute(tree);
        } // wait on ALL tasks completed
      }
  } // only now is background work to
    // be complete
  print_results();
```

New features

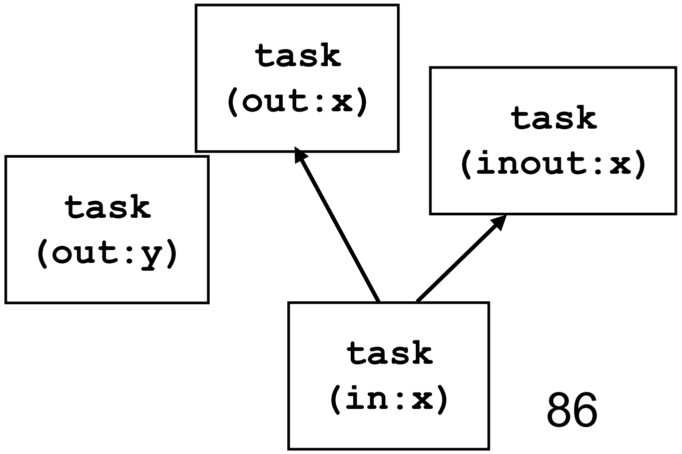
- device constructs (a.k.a. accelerator support)
- array section syntax for C/C++
- SIMD construct
- cancellation constructs (a.k.a. error model)
- thread affinity
- taskgroup construct
- task dependence
- user-defined reduction
- atomic construct extension
- OMP_DISPLAY_ENV environment variable
- partial Fortran 2003 support

Task dependence

```
depend(dependece-type: list)
```

```
dependence-type:
• in
• out
• inout
```

- a new clause for task construct
- list items – can be array sections and must be identical storage or disjoint storage
- dependent task – cannot be executed until its *predecessor tasks* have completed.
- predecessor task – a *task* that must complete before its *dependent tasks* can be executed.
- *dependence-type* :
  - **in**: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an **out** or **inout** clause
  - **out**, **inout**: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an **in**, **out**, or **inout** clause



Task dependence – example

```
// x and y are shared
#pragma omp task depend(out: x,y)
{ ... }                           //
  A
#pragma omp task depend(inout: x)
{ ... }                           //
  B
#pragma omp task depend(inout: y)
{ ... }                           //
  C
#pragma omp task dep
{ ... }
  D

A -> B
A -> C
B,C -> D
```

```
// x and y are shared
#pragma omp task depend(out: x)
{ ... }                           // A
#pragma omp task depend(out: x)
{ ... }                           // B

A -> B
```

Task dependence – example

```
// x and y are shared
#pragma omp task depend(out: x)
{ ... }                                    // A
#pragma omp task depend(out: x)
{ ... }                                    // B
#pragma omp task depend(in: x,y)
{ ... }                                    // C

A -> B -> C
```

```
// x and y are shared
#pragma omp task depend(in: x)
{ ... }                                  //
  A
#pragma omp task depend(in: x)
{ ... }                                  //
  B
```

Task dependence – example

```
void matmul_block(int N, int BS, float *A, float *B, float *C);
// BS divides perfecty N
void matmul(int N, int BS, float A[N][N], float B[N][N], float
   C[N][N]) {
  int i, j, k;
  for (i = 0; i < N; i+=BS) {
    for (j = 0; j < N; j+=BS) {
      for (k = 0; k < N; k+=BS) {

#pragma omp task depend(in:A[i:BS][k:BS],B[k:BS][j:BS]) \
  depend inout:C[i:BS][j:BS])
        matmul_block(N, BS, &A[i][k], &B[k][j], &C[i][j]);
      }
    }
  }
}
```

New features

- device constructs (a.k.a. accelerator support)
- array section syntax for C/C++
- SIMD constructs
- cancellation constructs (a.k.a. error model)
- thread affinity
- taskgroup construct
- task dependence
- user-defined reduction
- atomic construct extension
- OMP_DISPLAY_ENV environment variable
- partial Fortran 2003 support

User-defined reduction

- in pre-4.0, reduction only applies to intrinsic/builtin types and specific operators
- types
  - (C/C++) char, int, float, double, ...
  - (Fortran) integer, real, complex, ...
- operators
  - min, max, +, -, etc
- a request from the user community – to have a more flexible reduction construct that allows users to define their own types etc.

User-defined reduction

- 4.0 extends the reduction construct to allow user-defined types and user-defined operators
- the feature didn’t make it to 3.1 due to various issues
- add a **`declare reduction`** directive that associates data type and its corresponding reduction operation
- also include the initialization of the private copies of the reduction variable
- enhance **`reduction`** clause for the user-defined reduction identifier/operator

User-defined reduction – **`declare reduction`** directive

```
#pragma omp declare reduction(reduction-identifier:
typename-list: combiner) [initializer-clause] new-line
```

```
!$omp declare reduction(reduction-identifier:
type-list: combiner) [initializer-clause]
```

reduction-identifier: to identify a reduction on specific type(s) and operator (e.g. **`myAdd`**, **`+`**)

type: user-defined types or intrinsic types (e.g. **`myType`**, **`float`**, **`int`**)

combiner: how the reduction variables combined to the final result (e.g. an expression, **`omp_out=omp_out+omp_in`**; a routine, **`myAddroutine(...)`**)

initializer: how to initialize the private copies for the reduction variable (e.g. an expression, **`omp_priv=0`**; or a routine, **`initMyType(omp_priv)`**)

special variable identifiers:

- not actual variables
- **`omp_in`**, **`omp_out`**: used by the *combiner* to combine partial results to the final value
- **`omp_priv`**: refer to the storage to be initialized

User-defined reduction – example

```
typedef struct {
  double real;
  double imag;
} complex_t;

complex_t complex_add(complex_t a, complex_t b) {
  complex_t c;
  c.real = a.real + b.real;
  c.imag = a.imag + b.imag;
  return c;
}
#pragma omp declare reduction(cmplxAdd: complex_t:
   omp_out=complex_add(omp_out, omp_in)) initializer(
   omp_priv={0.0, 0.0} )
```

User-defined reduction

- this example is equivalent to the reduction in 3.1

```
#pragma omp declare reduction(+: int, float: omp_out+=omp_in) \
  initializer(omp_priv=0)
#pragma omp declare reduction(*: int, float: omp_out=omp_out*omp_in) \
  initializer(omp_priv=1)
  ...
  int x;
  float z;
#pragma omp parallel reduction(+: x)
{
  x = x + 1;
}
#pragma omp parallel reduction(+: z)
{
  z = z*2.0;
}
```

User-defined reduction – example

```
type dt
  integer :: i
  real :: x
end type
interface
  subroutine addDT(x, y)
  type(dt), intent(in) :: x
  type(dt), intent(inout) :: y
  end subroutine
end interface

!$omp declare reduction(dtAdd: dt :
  addDT(omp_in, omp_out))
  initializer(initDT(omp_priv))
```

User-defined reduction – example

```
typedef struct {
  double real;
  double imag;
} complex_t;

complex_t complex_add(complex_t
   a, complex_t b) {
  complex_t c;
  c.real = a.real + b.real;
  c.imag = a.imag + b.imag;
  return c;
}

#pragma omp declare
   reduction(cmplxAdd:
   complex_t:
   omp_out=complex_add(omp_out,
   omp_in))
   initializer( omp_priv={0.0,
   0.0} )
```

```
complex_t x, y;

#pragma omp parallel for
   reduction(cmplxAdd: x)
  for (i=0; i<N; i++) {
    x = complex_add(x, y);
  }

x = (complex_t) {0.0, 0.0};
#pragma omp parallel
   num_threads(48)
   reduction(cmplxAdd: x)
{
  x = (complex_t) {1.0, -1.0};
}
// output: x={48., -48.}
```

User-defined reduction – example

```
type dt
  integer :: i
  real :: x
end type

interface
  subroutine addDT(x, y)
  type(dt), intent(in) :: x
  type(dt), intent(inout) :: y
  end subroutine
end interface

!$omp declare reduction(dtAdd :
   dt : addDT(omp_in, omp_out))
   initializer(initDT(omp_priv)
   )
```

```
type(dt) :: x, y

!$omp parallel for
   reduction(dtAdd: x)
  do i=0, N
    call addDT(x, y)
  enddo


x = dt(0, 0.0)
!$omp parallel num_threads(48)
   reduction(dtAdd: x)
{
  x = dtAdd(1, -1.0)
}
// output: x={48, -48.}
```

New features

- device constructs (a.k.a. accelerator support)
- array section syntax for C/C++
- SIMD constructs
- cancellation constructs (a.k.a. error model)
- thread affinity
- taskgroup construct
- task dependence
- user-defined reduction
- atomic construct extension
- OMP_DISPLAY_ENV environment variable
- partial Fortran 2003 support

Atomic construct extension

- Support atomic swap
  - using capture clause

  -
    ```
    #pragma omp atomic capture
    {
      v = x;
      x = expr;
    }
    ```
- Support more forms
  - ***x = expr binop x;***
    - `[3.1]` ***x = x binop expr;***
  - ***{ v = x; x = expr binop x; }***
  - ***{ x = expr binop x; v = x; }***
  - ***{ v = x; x = expr; }***
  - ***v = x = x binop expr;***
  - ***v = x = expr binop x;***

Atomic construct extension

- Support sequentially consistent atomic constructs
- Provide the same semantic for C11/C++11 **`memory_order_seq_cst`** and **`memory_order_relaxed`** atomic operations
- Add **`seq_cst`** clause
- "Any **`atomic`** construct with a **`seq_cst`** clause forces the atomically performed operation to include an implicit flush operation without a list."

New features

- device construct (a.k.a. accelerator support)
- array section syntax for C/C++
- SIMD construct
- cancel construct (a.k.a. error model)
- thread affinity
- taskgroup construct
- task dependence
- user-defined reduction
- atomic construct extension
- OMP_DISPLAY_ENV environment variable
- partial Fortran 2003 support

OMP_DISPLAY_ENV environment variable

- display the OpenMP version number
- display the value of the ICV’s associated with the environment variables
- useful for users to find out the default settings, runtime version etc.

OMP_DISPLAY_ENV environment variable

```
OPENMP DISPLAY ENVIRONMENT BEGIN
  OMP_DISPLAY_ENV='TRUE'

  _OPENMP='201107'
  OMP_DYNAMIC='FALSE'
  OMP_MAX_ACTIVE_LEVELS='5'
  OMP_NESTED='FALSE'
  OMP_NUM_THREADS='80'
 OMP_PLACES='{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15},{16,17,18,19},
   {20,21,22,23},{24,25,26,27},{28,29,30,31},{32,33,34,35},{36,37,38,39},
   {40,41,42,43},{44,45,46,47},{48,49,50,51},{52,53,54,55},{56,57,58,59},
   {60,61,62,63},{64,65,66,67},{68,69,70,71},{72,73,74,75},{76,77,78,79}'
   cores
  OMP_PROC_BIND='FALSE'
  OMP_SCHEDULE='STATIC,0'
  OMP_STACKSIZE='4194304'
  OMP_THREAD_LIMIT='80'
  OMP_WAIT_POLICY='PASSIVE'
OPENMP DISPLAY ENVIRONMENT END
```

OMP_DISPLAY_ENV environment variable

```
OPENMP DISPLAY RUNTIME BEGIN
  LOMP_VERSION='0.31'
  BUILD_LEVEL='OpenMP Runtime Version: 13.01(C/C++) and 15.01(Fortran) Level: 130612 ID:
    _Ld41MtKrEeKCbrnTkssZtQ'
  BUILT='xlC, level=12.1.0.0'
  BUILDTIME='Jun 12 2013, 16:14:57'
  TARGET='Linux, 32 bit'
OPENMP DISPLAY RUNTIME END

OPENMP DISPLAY ENVIRONMENT BEGIN
  OMP_DISPLAY_ENV='VERBOSE'

  ...
  XLSMPOPTS=' DELAYS=1000'
  XLSMPOPTS=' NOSTACKCHECK'
  XLSMPOPTS=' PARTHDS=80'
  XLSMPOPTS=' PARTHRESHOLD=   0.02'
  XLSMPOPTS=' PROFILEFREQ=0'
  XLSMPOPTS=' SCHEDULE=STATIC=0'
  XLSMPOPTS=' SEQTHRESHOLD=   0.02'
  XLSMPOPTS=' SPINS=64'
  XLSMPOPTS=' STACK=4194304'
  XLSMPOPTS=' USRTHDS=0'
  XLSMPOPTS=' YIELDS=64'
OPENMP DISPLAY ENVIRONMENT END
```

New features

- atomic construct extension
- taskgroup construct
- task dependence
- cancel construct (a.k.a. error model)
- OMP_DISPLAY_ENV environment variable
- thread affinity
- user-defined reduction
- SIMD construct
- device construct (a.k.a. accelerator support)
- array section syntax for C/C++
- partial Fortran 2003 support

Other changes

- move examples (appendix A) out of spec
- re-organize the ICV table (splitting it into three tables)