GPU Programming with Directives

Introduction to OpenMP offload

archer2

epcc

Reusing this material

OpenMP Offload to GPU

epcc

- GPU offload introduced in OpenMP 4.0
  - fully integrated alongside OpenMP for CPU
  - significant revisions/extensions added in 4.5 and 5.0
  - https://www.openmp.org/wp-content/uploads/openmp-examples-6.0.pdf
  - Similar to OpenACC directives
  - OpenACC is an alternative standard for offloading to GPU
  - Developed before OpenMP 4.0
  - https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.2-final.pdf
- Several implementations of OpenMP GPU offload
  - NVIDIA, AMD, GCC, HPE (Cray), IBM, LLVM/clang, Intel
- Hardware agnostic
  - NVIDIA V100/A100/H100/GH200
  - AMD MI210/MI250(X)/MI300(A/X)
  - Intel Xe-HPC

OpenMP Device Model

epcc

- Host-centric model with one host device and multiple target devices of the same type.
- A device is a logical execution engine with local storage.
- A device data environment is a data environment associated with a target data or target region.
- `target` directives control how data and code are offloaded to devices.
- Data is mapped from a host data environment to a device data environment.

The Target Region

epcc

- The target region is the basic offloading mechanism in OpenMP.
  - It defines a section of code, e.g., a loop.
- During execution, when a target construct is encountered, the code it contains is executed on a device.
- By default, the code inside the target construct executes sequentially.
- Meanwhile, the host thread waits for the target region to finish, before executing the remainder of the code.

The Target Region

epcc

- The target region is the basic offloading mechanism in OpenMP.
  - It defines a section of code, e.g., a loop.
- During execution, when a target construct is encountered, the code it contains is executed on a device.
- By default, the code inside the target construct executes sequentially.
- Meanwhile, the host thread waits for the target region to finish, before executing the remainder of the code.

```
// Block A: executed on host
#pragma omp target
{
    // Block B: executed on device
}
// Block C: executed on host
```

C/C++

```
! Block A: executed on host
!$OMP TARGET
    ! Block B: executed on device
!$OMP END TARGET
! Block C: executed on host
```

Fortran

Host and Device Data

- The host and device have separate memory spaces
- Data declared on the host and accessed within a target region must first be mapped to the device.
- This (in general) makes a copy of the data on the device, but both host and device versions of a variable are referred to by the *same* name.
- Mapped data must not be accessed by the host until the target region has completed.
- Default behaviour (as from OpenMP 4.5):
  - scalars referenced in the target region are treated as **`firstprivate`**
    - i.e., scalar is initialised with value on host then copied to device
  - static arrays are copied to the device on entry and back to the host on exit

Map clause

How data is mapped to the device is specified via the **`map`** clause on the **`target`** construct.

```
#pragma omp target map(map-type: list)
```

C/C++

```
!$OMP TARGET MAP(map-type: list)
```

Fortran

`map-type` can be one of the following,

- **`to`** copy the data to the device on entry;
- **`from`** copy the data to the host on exit;
- **`tofrom`** copy the data to the device on entry and back on exit;
- **`alloc`** allocate an *uninitialised* copy on the device (don't copy values);

and **`list`** is simply a list of variables.

Map clause example

epcc

Two arrays and one scalar

```
#pragma omp target \
map(to: B, C), map(tofrom: sum)
{
  for (int i=0; i<N; i++) {
    sum += B[i] + C[i];
  }
}
```

C/C++

```
!$OMP TARGET
!$omp& map(to: B(1:N), C(1:N))
!$omp& map(tofrom: sum)
DO i = 1,N
  sum = sum + B(i) + C(i)
ENDDO
!$OMP END TARGET
```

Fortran

Map clause example

epcc

Two arrays and one scalar

```
#pragma omp target \
map(to: B, C), map(tofrom: sum)
{
  for (int i=0; i<N; i++) {
    sum += B[i] + C[i];
  }
}
```

C/C++

```
!$OMP TARGET
!$omp& map(to: B(1:N), C(1:N))
!$omp& map(tofrom: sum)
DO i = 1,N
  sum = sum + B(i) + C(i)
ENDDO
!$OMP END TARGET
```

Fortran

Execution on device is sequential!

Parallelism on a Device

In principle, we can use all the ”normal” OpenMP constructs inside a target region to create and use threads on the device.

- Parallel regions, worksharing, synchronisation, tasks, etc.

However, GPUs are not able to support a full threading model outside of a single streaming multiprocessor (NVIDIA) or compute unit (AMD).

- no synchronization or memory fences possible between SMs
- no coherency between L1 caches
- a parallel region inside a target region will only execute on one SM
- compare with CUDA – can only synchronise threads inside a thread block, not between thread blocks

GPU offload with reduction

```
#pragma omp target \
map(to: B, C), map(tofrom: sum)\
parallel for reduction(+:sum)
  for (int i=0; i<N; i++) {
    sum += B[i] + C[i];
  }
```

C/C++

```
!$OMP TARGET PARALLEL DO
!$omp& map(to: B(1:N), C(1:N))
!$omp& map(tofrom: sum)
!$omp& reduction(+:sum)
DO i = 1,N
  sum = sum + B(i) + C(i)
ENDDO
!$OMP END TARGET PARALLEL DO
```

Fortran

Only one GPU SM/CU utilised!

Teams Construct

- Creates multiple master threads inside a target region.
- Each master thread can spawn its own team of threads within separate parallel regions.
- Threads in different teams cannot synchronise with each other.
  - Barriers, critical regions, locks only apply to the threads within a team.
- Can control the number of teams and the number of threads per team.

Teams and Distribute Construct

```
#pragma omp target teams distribute parallel for \
map(to: B, C), map(tofrom: sum) reduction(+:sum)
for (int i=0; i<N; i++) {
  sum += B[i] + C[i];
}
```

C/C++

Teams and Distribute Construct

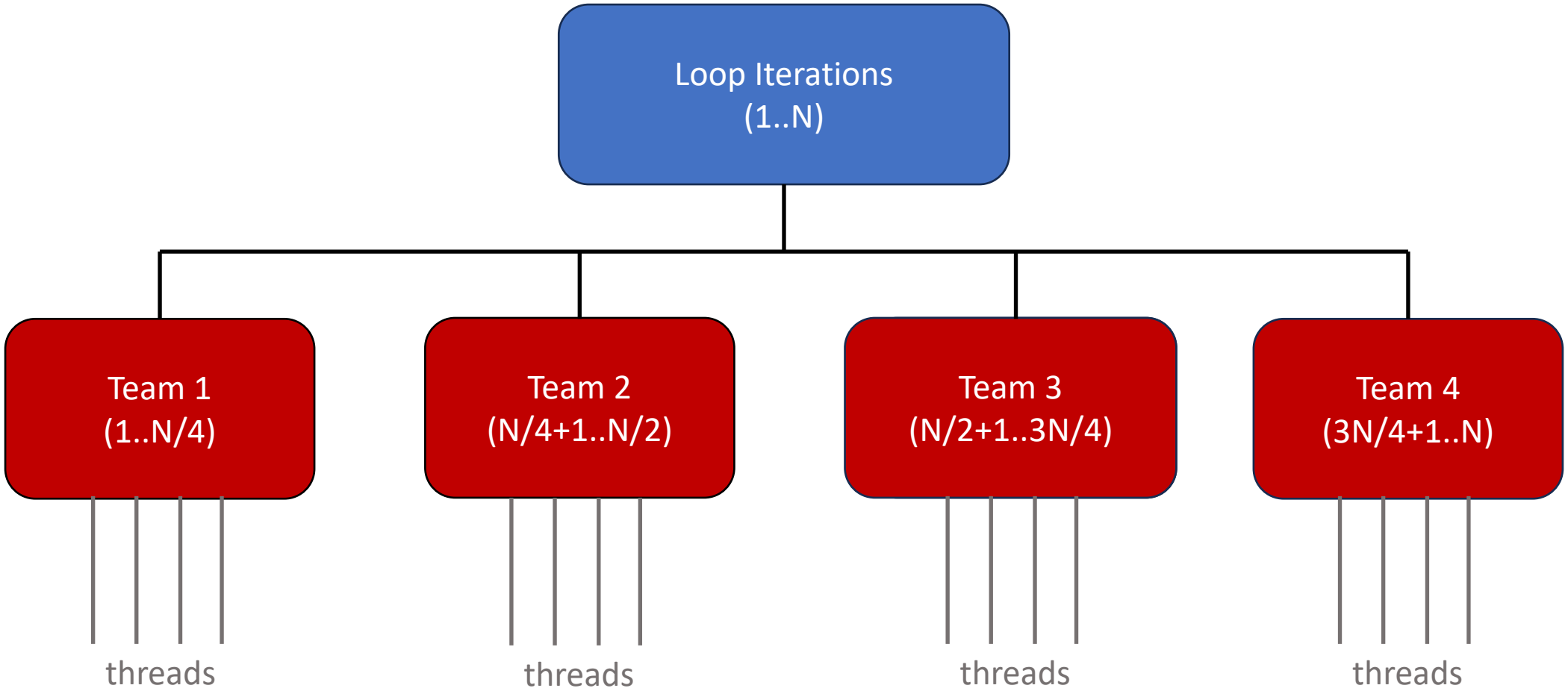
```
#pragma omp target teams distribute parallel for \
map(to: B, C), map(tofrom: sum) reduction(+:sum)
for (int i=0; i<N; i++) {
  sum += B[i] + C[i];
}
```

C/C++

```
!$OMP TARGET TEAMS DISTRIBUTE PARALLEL DO
!$omp& map(to: B(1:N), C(1:N))
!$omp& map(tofrom: sum) reduction(+:sum)
DO i = 1,N
  sum = sum + B(i) + C(i)
ENDDO
!$OMP END TARGET TEAMS DISTRIBUTE PARALLEL DO
```

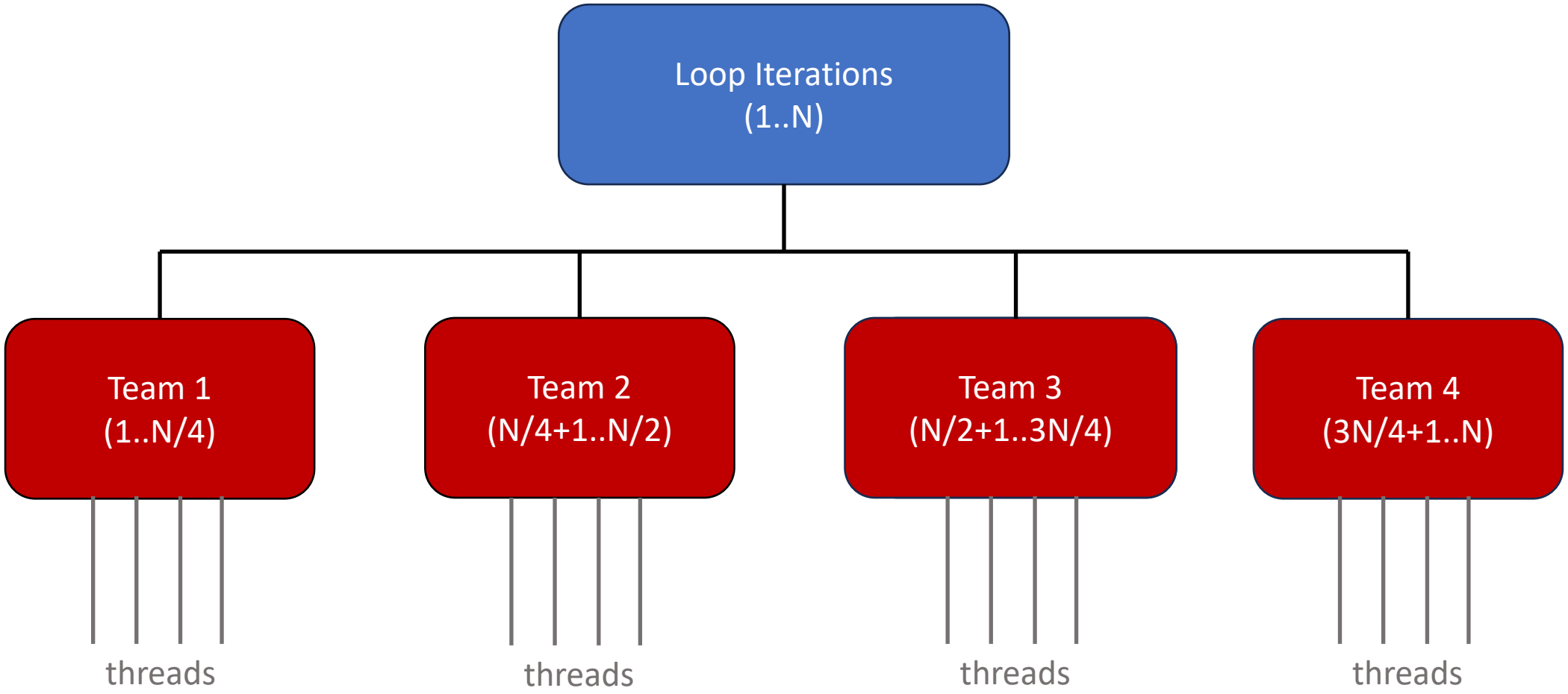
Fortran

Teams and Parallel Regions



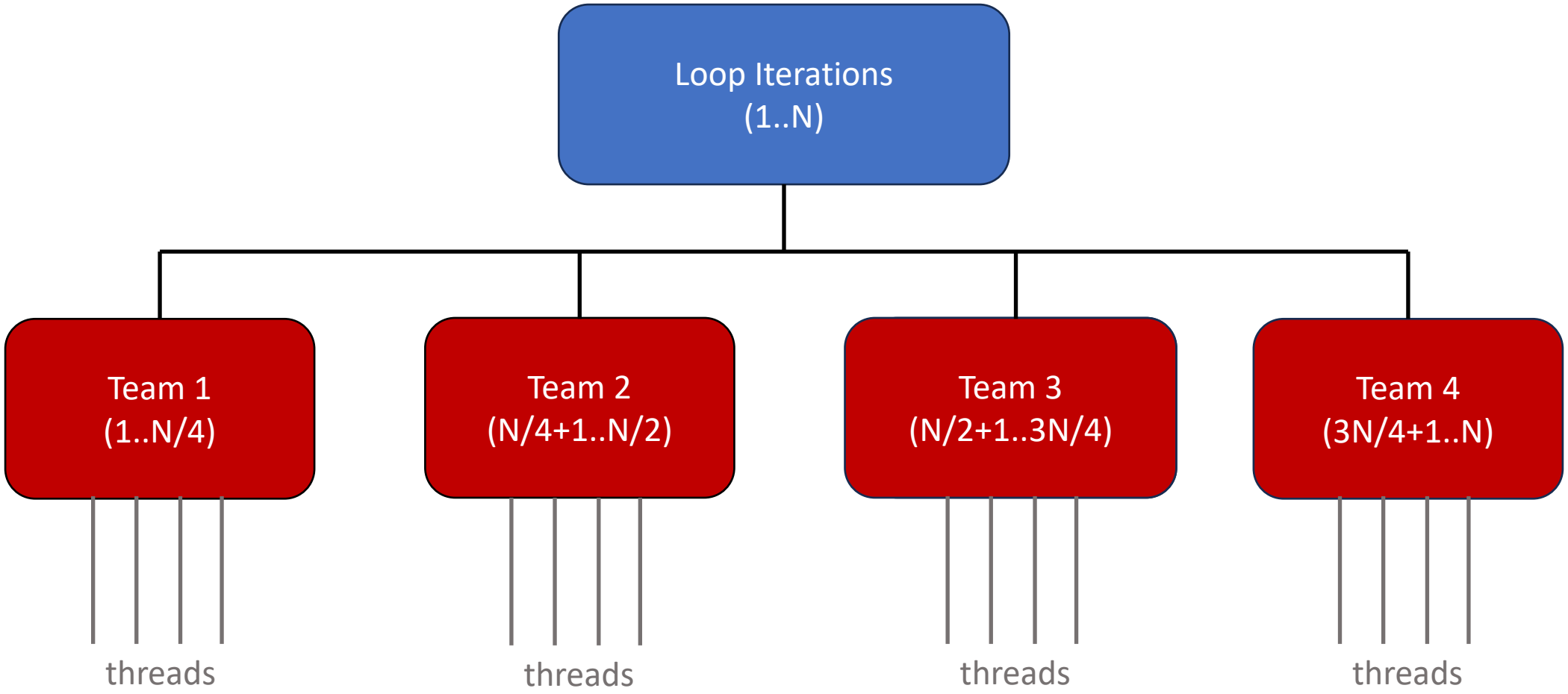
- **`distribute`** construct: distribute loop iterations across teams.
- **`for`** or **`do`** construct: further distribute iterations across team threads.

Teams and Parallel Regions



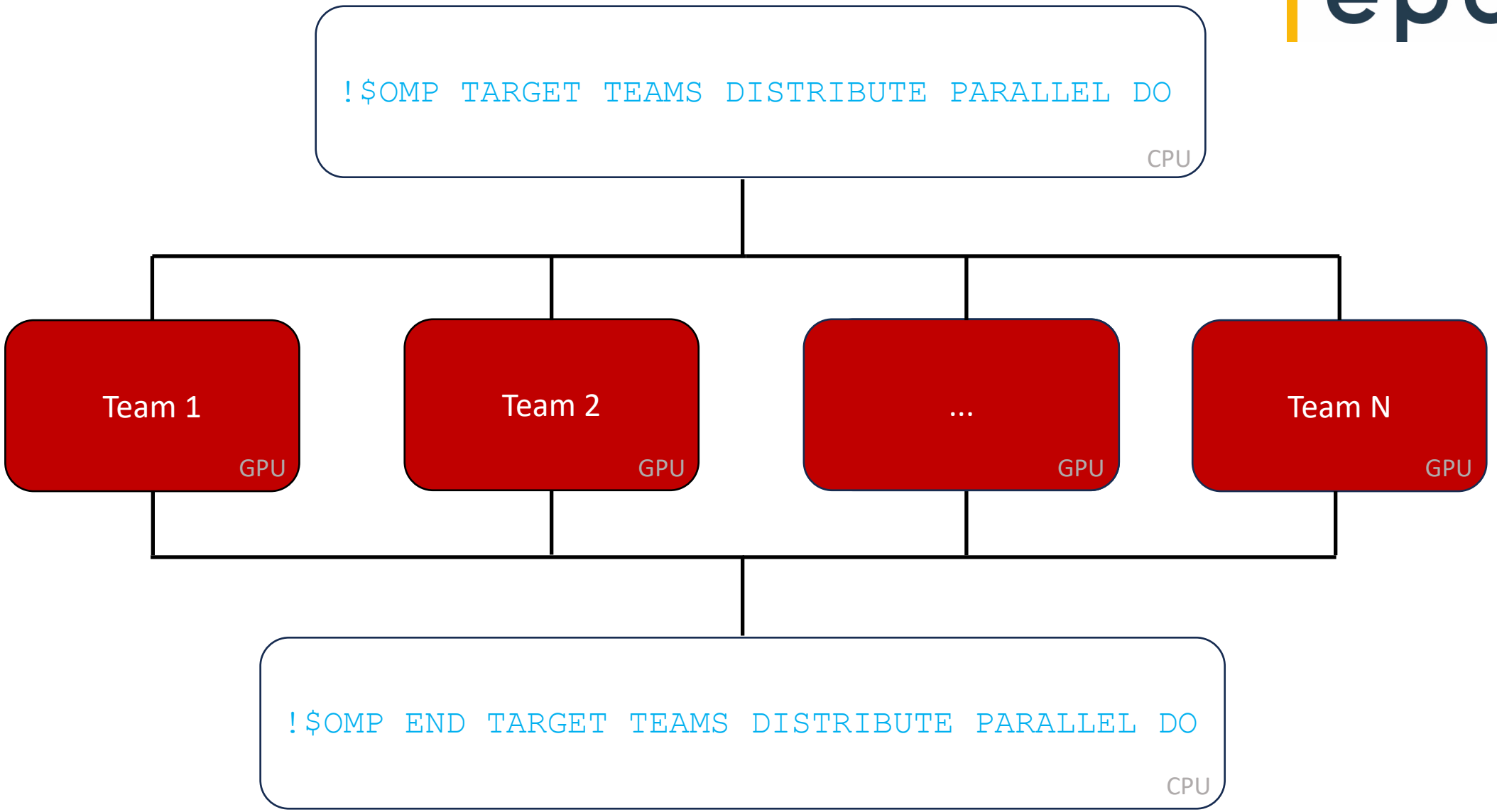
- Each team has its own master thread and runs on a single GPU SM.
- Each team creates a separate parallel region.
  - Typically, a thread runs on a single core within GPU SM.

Teams and Parallel Regions

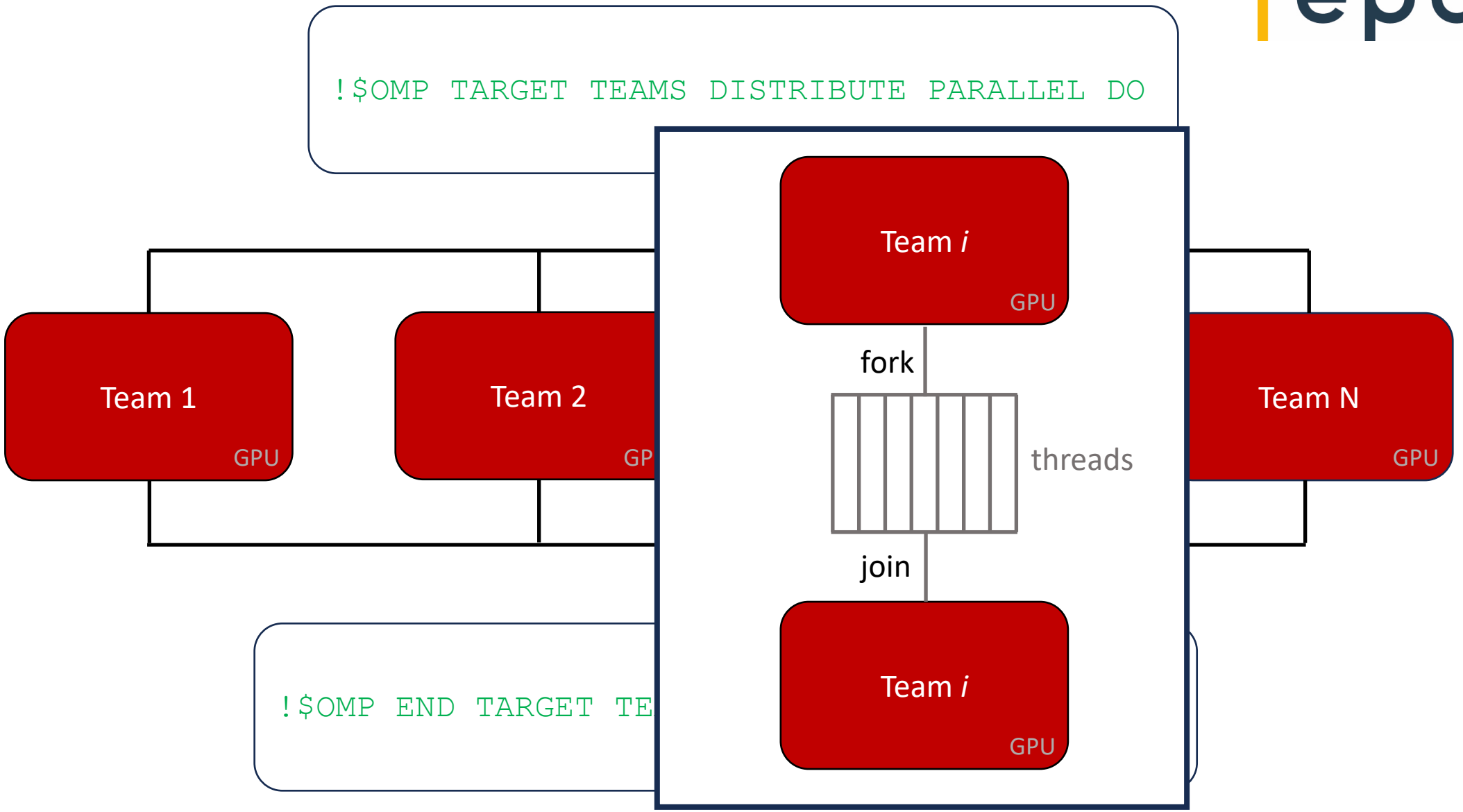


- The number of teams is limited by number of SM/CUs on GPU.
- The number of threads per team depends on number of cores per SM/CU.

Fork Join


!$OMP TARGET TEAMS DISTRIBUTE PARALLEL DO
CPU
Team 1
GPU
Team 2
GPU
...
GPU
Team N
GPU
!$OMP END TARGET TEAMS DISTRIBUTE PARALLEL DO
CPU

Fork Join


!$OMP TARGET TEAMS DISTRIBUTE PARALLEL DO
Team 1
GPU
Team 2
GP
Team i
GPU
fork
threads
join
Team i
GPU
Team N
GPU
!$OMP END TARGET TE

Teams Scheduling

epcc

```
#pragma omp target teams distribute parallel for \
map(to: B, C) map(tofrom: sum) reduction(+:sum)\
dist_schedule(static, 4)
for (int i=0; i<N; i++) {
  sum += B[i] + C[i];
}
```

C/C++

```
!$OMP TARGET TEAMS DISTRIBUTE PARALLEL DO
!$omp& map(to: B(1:N), C(1:N))
!$omp& map(tofrom: sum) reduction(+:sum)
!$omp& dist_schedule(static, 4)
DO i = 1,N
  sum = sum + B(i) + C(i)
ENDDO
!$OMP END TARGET TEAMS DISTRIBUTE PARALLEL DO
```

Fortran

Teams Scheduling

```
#pragma omp target teams distribute parallel for \
map(to: B, C) map(tofrom: sum) reduction(+:sum)\
dist_schedule(static, 4)
for (int i=0; i<N; i++) {
  sum += B[i] + C[i];
}
```

C/C++

```
!$OMP TARGET TEAMS DISTRIBUTE PARALLEL DO
!$omp& map(to: B(1:N), C(1:N))
!$omp& map(tofrom: sum) reduction(+:sum)
!$omp& dist_schedule(static, 4)
DO i = 1,N
  sum = sum + B(i) + C(i)
ENDDO
!$OMP END TARGET TEAMS DISTRIBUTE PARALLEL DO
```

Fortran

- Same as schedule clause on worksharing loops, but no dynamic or guided option.
- Iterations grouped into chunks and assigned to teams in a round-robin fashion.
- Number of iterations per chunk can be specified.
- If not specified, chunk size per team is approx. equal.

Synchronisation

- All of the “normal” OpenMP synchronisation constructs and routines can be used inside target regions.
- However, most of them, including `barrier`, `critical` and locks only synchronise the threads *within* a team, and not across different teams.
  - means that these are of limited use
- The `atomic` constructs, on the other hand, which protect concurrent accesses to memory locations, do synchronise across the whole device.
  - typically, these are efficiently supported by the hardware

Calling functions inside target regions (C/C++)

loop.c

```
...
#pragma omp target teams distribute parallel for \
map(to: B, C), map(tofrom: sum)\
dist_schedule(static)
for (int i=0; i<N; i++) {
  myfunc(i,N,B,C,&sum);
}
...
```

Calling functions inside target regions (C/C++)

loop.c

```
...
#pragma omp target teams distribute parallel for \
map(to: B, C), map(tofrom: sum)\
dist_schedule(static)
for (int i=0; i<N; i++) {
  myfunc(i,N,B,C,&sum);
}
...
```

myfunc.c

```
#pragma omp declare target
void myfunc(int i, int N, double B, double C, double *sum)
{
...
}
#pragma omp end declare target
```

Calling functions inside target regions (Fortran)

loop.f

```
...
!$OMP TARGET TEAMS DISTRIBUTE PARALLEL DO
!$omp& map(to: B(1:N), C(1:N))
!$omp& map(tofrom: sum)
!$omp& dist_schedule(static)
DO i = 1,N
  call mysub(i, N, B, C, sum)
ENDDO
!$OMP END TARGET TEAMS DISTRIBUTE PARALLEL DO
...
```

Calling functions inside target regions (Fortran)

loop.f

```
...
!$OMP TARGET TEAMS DISTRIBUTE PARALLEL DO
!$omp& map(to: B(1:N), C(1:N))
!$omp& map(tofrom: sum)
!$omp& dist_schedule(static)
DO i = 1,N
  call mysub(i, N, B, C, sum)
ENDDO
!$OMP END TARGET TEAMS DISTRIBU
...
```

mysub.f

```
subroutine mysub(i, B, C, N, sum)
implicit none
integer i, sum, N
real*8 B(N), C(N)
...
!$omp declare target
...
end subroutine mysub
```

Runtime Execution Environment Routines

epcc

Name	Description	Location
`omp_get_num_devices`	the number of non-host devices available for offloading code or data	host only
`omp_get_device_num`	the device number of the device on which the calling thread is executing	host and device
`omp_get_default_device` `omp_set_default_device`	the default target device	host only
`omp_get_max_teams`	an upper bound on the number of teams that could be created by a team's construct	host and device
`omp_get_num_teams` `omp_set_num_teams`	the number of initial teams in the current team's region	host and device
`omp_get_team_num`	the initial team number of the calling thread	host and device
`omp_get_teams_thread_limit` `omp_set_teams_thread_limit`	the maximum number of OpenMP threads per team	host and device

Runtime Execution Environment Routines

```C/C++
#include <mpif.h>
#include <omp.h>
void myfunc(...) {
  ...
  int rank, ierr, ndevices;
  ...
  ierr = MPI_comm_rank(MPI_COMM_WORLD,&rank);
  ...
  local_rank = rank % nranks_per_node;
  ...
  ndevices = omp_get_num_devices();
  omp_set_default_device(localrank % ndevices);
  ...
}
```

Runtime Execution Environment Routines

```fortran
subroutine mysub(...)
  use omp_lib
  implicit none
  include 'mpif.h'
  ...
  integer rank, ierr, ndevices
  ...
  call MPI_comm_rank(MPI_COMM_WORLD,rank,ierr)
  ...
  local_rank = MOD(rank,nranks_per_node)
  ...
  ndevices = omp_get_num_devices()
  call omp_set_default_device(MOD(localrank,ndevices))
  ...
end subroutine mysub
```

Fortran

Teams Construct Clauses: num_teams and thread_limit

epcc

```
#pragma omp target teams distribute parallel for \
map(to: B, C), map(tofrom: sum) reduction(+:sum)\
num_teams(4), thread_limit(32)
for (int i=0; i<N; i++) {
  sum += B[i] + C[i];
}
```

C/C++

Teams Construct Clause	Runtime Execution Environment Routine
num_threads	omp_set_num_teams
thread_limit	omp_set_teams_thread_limit

Teams Construct Clauses: num_teams and thread_limit

```
!$OMP TARGET TEAMS DISTRIBUTE PARALLEL DO
!$omp& map(to: B(1:N), C(1:N))
!$omp& map(tofrom: sum) reduction(+:sum)
!$omp& num_teams(4), thread_limit(32)
DO i = 1,N
   sum = sum + B(i) + C(i)
ENDDO
!$OMP END TARGET TEAMS DISTRIBUTE PARALLEL DO
```

Fortran

Teams Construct Clause	Runtime Execution Environment Routine
num_threads	omp_set_num_teams
thread_limit	omp_set_teams_thread_limit

Teams Loop Construct

epcc

For convenience, you can use the teams loop construct, which will request the compiler to generate the mapping/offload options.

```C/C++
#pragma omp target teams loop
for (int i=0; i<N; i++) {
  sum += B[i] + C[i];
}
```

```Fortran
!$OMP TARGET TEAMS LOOP
DO i = 1,N
  sum = sum + B(i) + C(i)
ENDDO
!$OMP END TARGET TEAMS LOOP
```

Teams Loop Construct

epcc

For convenience, you can use the teams loop construct, which will request the compiler to generate the mapping/offload options.

```C/C++
#pragma omp target teams loop
for (int i=0; i<N; i++) {
  sum += B[i] + C[i];
}
```

```Fortran
!$OMP TARGET TEAMS LOOP
DO i = 1,N
  sum = sum + B(i) + C(i)
ENDDO
!$OMP END TARGET TEAMS LOOP
```

Please note, the compiler might not generate any offload options,
or the options generated may not properly parallelise the code for the target device.

Teams Loop Construct

epcc

You can tell the compiler to show which offload options were chosen.

For example, for NVIDIA nvfortran/nvcc use the "`-Minfo=mp`" setting.

```
1   !$OMP TARGET TEAMS LOOP
2   DO i = 1,N
3      sum = sum + B(i) + C(i)
4   ENDDO
5   !$OMP END TARGET TEAMS LOOP
```

Fortran

```
...
 1, !$omp target teams loop
       1, Generating "nvkernel_cpt_dat_vals_p__F1L160_6" GPU kernel
          Generating NVIDIA GPU code
               2, Loop parallelized across teams, threads(128)
                  Generating implicit reduction(+:sum)
 1, Generating implicit map(tofrom:c(:),b(:))
...
```

Teams Loop Construct

You can tell the compiler to show which offload options were chosen.

For example, for NVIDIA nvfortran/nvcc use the "`-Minfo=mp`" setting.

```
1   !$OMP TARGET TEAMS LOOP
2   DO i = 1,N
3      sum = sum + B(i) + C(i)
4   ENDDO
5   !$OMP END TARGET TEAMS LOOP
```

Fortran

Such diagnostics can be obtained using other compilers...

```
gfortran -O3 -cpp -fopenmp -fdump-tree-all \
        -foffload=nvptx-none -foffload="-lm -lgfortran -latomic" ...

crayftn -O3 -cpp -h omp -h list=a ...
```

... but, some grepping may be required...

...search through generated "`<source file name>.f.*`" files.

Further Reading

OpenMP API Examples v6.0 Nov 2024

https://www.openmp.org/wp-content/uploads/openmp-examples-6.0.pdf

See the Devices chapter (No. 6)

Gives the OpenMP version when particular feature was introduced.