OpenMP (Open Multi-Processing)

(https://www.openmp.org)

Take away

- What is OpenMP?
- How OpenMP programming model works ?
- How to write, compile and run OpenMP program?
- What are various OpenMP directives, runtime library API & Environment variables?
- How OpenMP program executes on accelerators (GPUs)?
- What are the best practices to write OpenMP program?

Introduction (1)

- Governed by OpenMP Architecture Review Board (ARB)
- Open source, simple and up to date with the latest hardware development
- Specification for shared memory parallel programming model for Fortran and C/C++ programming languages
  - compiler directives
  - library routines
  - environment variables
- Widely accepted by community – academician & industry

Introduction (2)

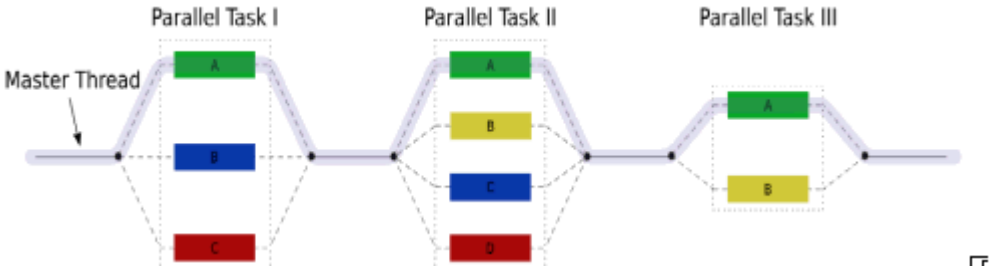
- Shorter learning curve
- Supported by many compiler vendors
  - GNU, LLVM, AMD, IBM, Cray, Intel, PGI, OpenUH, ARM
  - Compilers from research labs e.g. LLNL, Barcelona Supercomputing Centre
- Easy to maintain sequential, CPU parallel and GPU parallel versions of code.
- Used in application development from variety of fields and systems
  - Aeronautics, Automotive, Pharmaceutics, Finance
  - Supercomputers, Accelerators, Embedded multi-core systems

Specification History

- OpenMP 1.0 (1997 (Fortran) / 1998 (C/C++))
  - First release of Fortran & C/C++ specification (2 separate specifications)
- OpenMP 2.0 (2000 (Fortran) / 2002 (C/C++))
  - Addition of timing routines and various new clauses – firstprivate, lastprivate, copyprivate etc.
- OpenMP 2.5 (2005)
  - combined standard for C/C++ & Fortran
  - Introduce notion of internal control variables (ICVs) and new constructs – single, sections etc.
- OpenMP 3.0/3.1 (2008 – 2011)
  - Concept of *task* added in OpenMP execution model
- OpenMP 4.0 (2013)
  - Accelerator (GPU) support
  - SIMD construct for vectorization of loops
  - Support for FORTRAN 2003
- OpenMP 4.5 (2015)
  - Significant improvement for devices (GPU programming) & Fortran 2003
  - New taskloop construct
- **OpenMP 5.0 (2018)** – compilers are not yet support full set of features of this standard !
  - Extended memory model to distinguish different types of *flush* operations
  - Added *target-offload-var* ICV and OMP_TARGET_OFFLOAD environment variable
  - Teams construct extended to execute on host device

OpenMP Programming (1)

- The OpenMP API uses the fork-join model of parallel execution.
- The OpenMP API provides a relaxed-consistency, shared-memory model.
- An OpenMP program begins as a single thread of execution, called an initial thread. An initial thread executes sequentially.
- When any thread encounters a parallel construct, the thread creates a team of itself and zero or more additional threads and becomes the master of the new team.



OpenMP Programming (2)

- Compiler directives
  - Specified with **pragma** preprocessing
  - Starts with **#pragma omp** (For C/C++)
  - Case sensitive
  - Any directive is followed by one or more clauses
  - Syntax #pragma omp directive-name [clause[[,] clause] ...] new-line
- Runtime library routines
  - The library routines are external functions with “C” linkage
  - **omp.h** provides prototype definition of all library routines
- Environment variables
  - Set at the program start up
  - Specifies the settings of the Internal Control Variables (ICVs) that affect the execution of OpenMP programs
  - Always in upper case

OpenMP Hello World

helloWorld.c

```
#include <stdio.h>
#include <omp.h>

int main()
{
   int i = 0;
   int numThreads = 0;
   // call to OpenMP runtime library
   numThreads = omp_get_num_threads(numThreads);

   // OpenMP directive
   #pragma omp parallel
   {
      // call to OpenMP runtime library
      int threadNum = omp_get_thread_num();
      printf("Hello World from thread %d \n", threadNum);
   }
   return 0;
}
```

```
// Setting OpenMP environment variable
export OMP_NUM_THREADS=4

//compile helloWorld program
xlc -qsmp=omp helloWorld.c -o helloWorld

// Run helloWorld program
./helloWorld
```

Output

```
[aditya@hpcwsw7 openmp_tutorial]$ ./helloWorld
Hello World from thread 0
Hello World from thread 3
Hello World from thread 1
Hello World from thread 2

[aditya@hpcwsw7 openmp_tutorial]$ ./helloWorld
Hello World from thread 0
Hello World from thread 1
Hello World from thread 2
Hello World from thread 3
```

Compilation of OpenMP program

XL	GNU
• **CPU** -qsmp=omp	• **CPU** -fopenmp
• **GPU** -qsmp=omp -qoffload -qtgtarch=sm_70 (V100 GPU)	• **GPU** -fopenmp -foffload=nvptx-none

NVIDIA CUDA linking

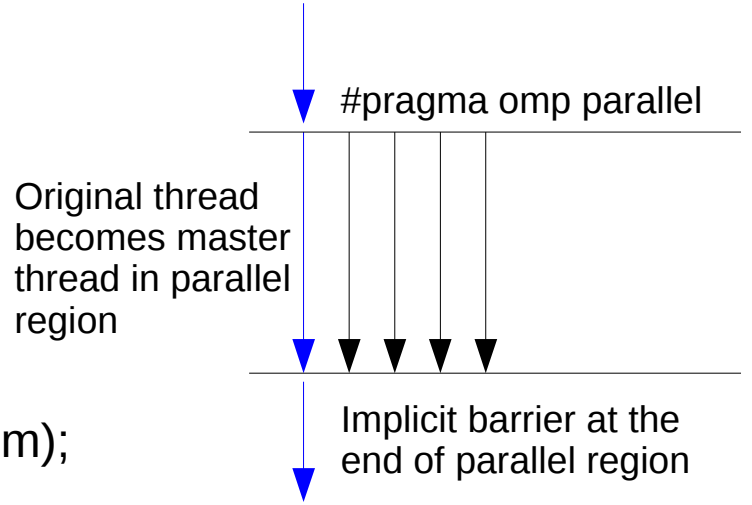
-lcudart -L/usr/local/cuda/lib64

OpenMP Directives

- Parallel construct
- SIMD construct
- Combined construct
- Work sharing construct
- Master and synchronization construct
- Tasking construct
- Device construct

Parallel construct

#pragma omp parallel [clause[[,] clause] ...] new-line
{
}



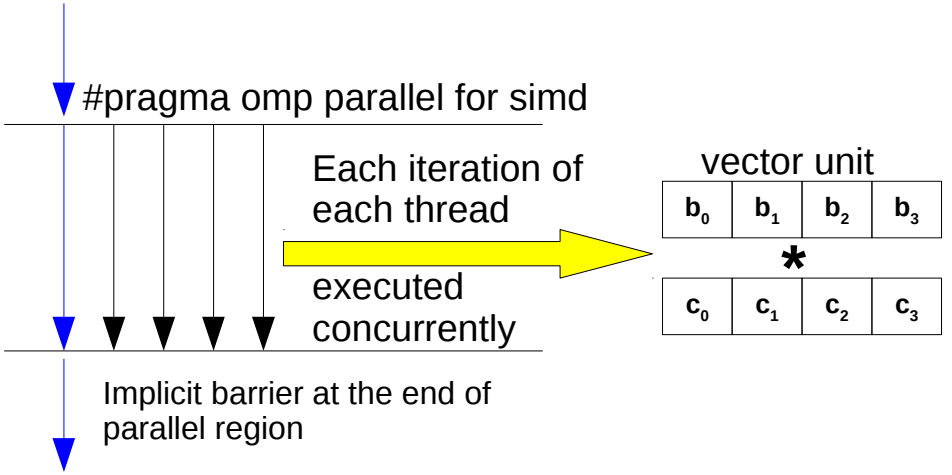

```
#pragma omp parallel
{
      int threadNum = omp_get_thread_num();
      printf("Hello World from thread %d \n", threadNum);
}
```

- The fundamental construct that starts parallel execution
- A team of threads is created to execute parallel region
- Original thread becomes *master* of the new team
- All threads in the team executes parallel region

SIMD construct

```
#pragma omp simd [clause[ [,] clause] ... ] new-line
{
 }
```

```
#pragma omp parallel for simd
for (i=0; i<N; i++)
{
      a[i] = b[i] * c[i]
}
```



- Applied on loop to transform loop iterations to execute concurrently using SIMD instructions
- When any thread encounters a simd construct, the iterations of the loop associated with the construct may be executed concurrently using the SIMD lanes that are available to the thread.

Combined construct

- Combination of more than one construct
  - Specifies one construct immediately nested inside another construct
- Clauses from both constructs are permitted
  - With some exceptions e.g. **nowait** clause cannot be specified in **parallel for** or **parallel sections**
- Examples
  - #pragma omp parallel for
  - #pragma omp parallel for simd
  - #pragma omp parallel sections
  - #pragma omp target parallel for simd

Worksharing construct

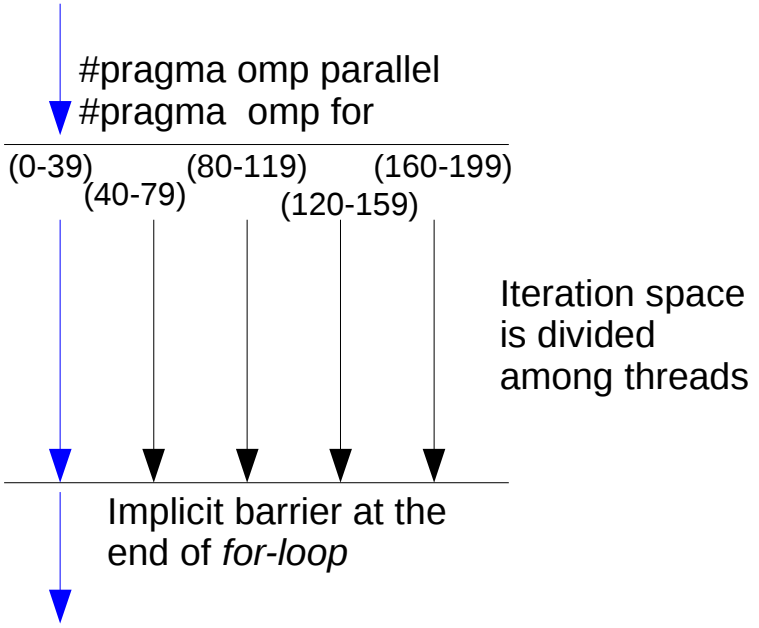
- Distributes the execution of the associated parallel region among the members of the team
- Implicit barrier at the end
- Types
  - Loop construct
  - Sections construct
  - Single construct
  - Workshare construct (only in Fortran)

Worksharing : Loop construct

#pragma omp for [clause[[,] clause] ...] new-line
// for-loop
{
}

```
int N = 200;
#pragma omp parallel
{
        #pragma omp for

        for (i=0; i<N; i++)
        {
                a[i] = b[i] * c[i]
        }
}
```



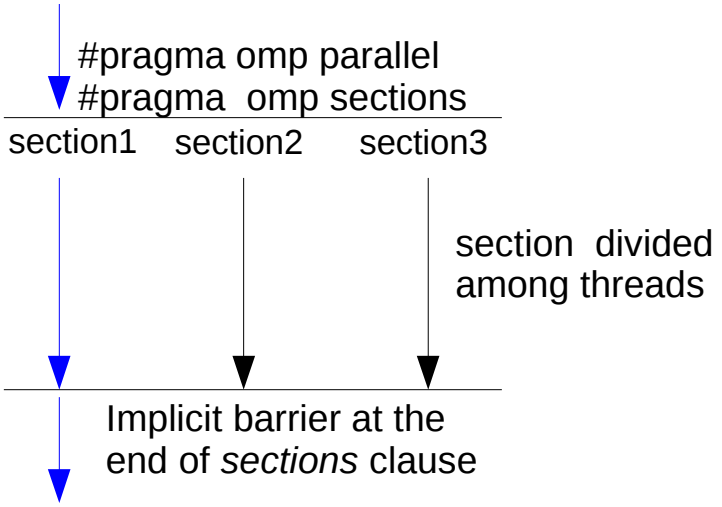
- Iterations of *for-loop* distributed among the threads
- One or more iterations executed in parallel
- Implicit barrier at the end unless *nowait*

Worksharing : Section construct

```
#pragma omp sections [clause[ [,] clause] ... ] new-line
{
   #pragma omp section new-line
   // structured block
   #pragma omp section new-line
   // structured block
}
```

- Non iterative worksharing construct
- Each section structured block is executed once
- The section structured blocks are executed in parallel (one thread per section)
- Implicit barrier at the end unless *nowait* specified

```
#pragma omp parallel
{
  #pragma omp sections
  {
     #pragma omp section
       {
       // do_work1
       }
       #pragma omp section
       {
       // do_work2
       }
  }
}
```



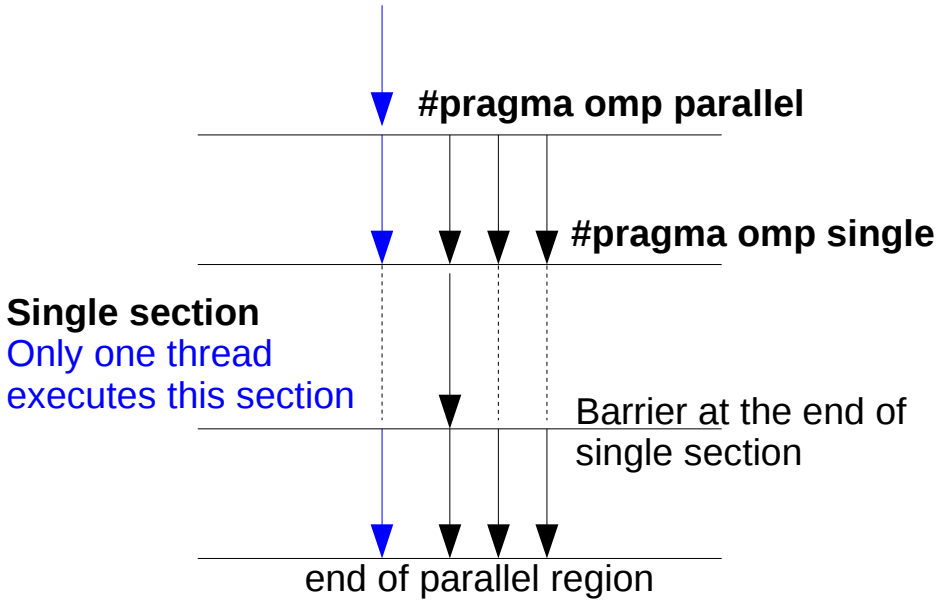
Worksharing : Single construct

#pragma omp single [clause[[,] clause] ...] new-line
{
}

```
#pragma omp parallel
{
   // do_some_work()

   #pragma omp single
   {
   //write result to file
   }

   // do_post_work()
}
```

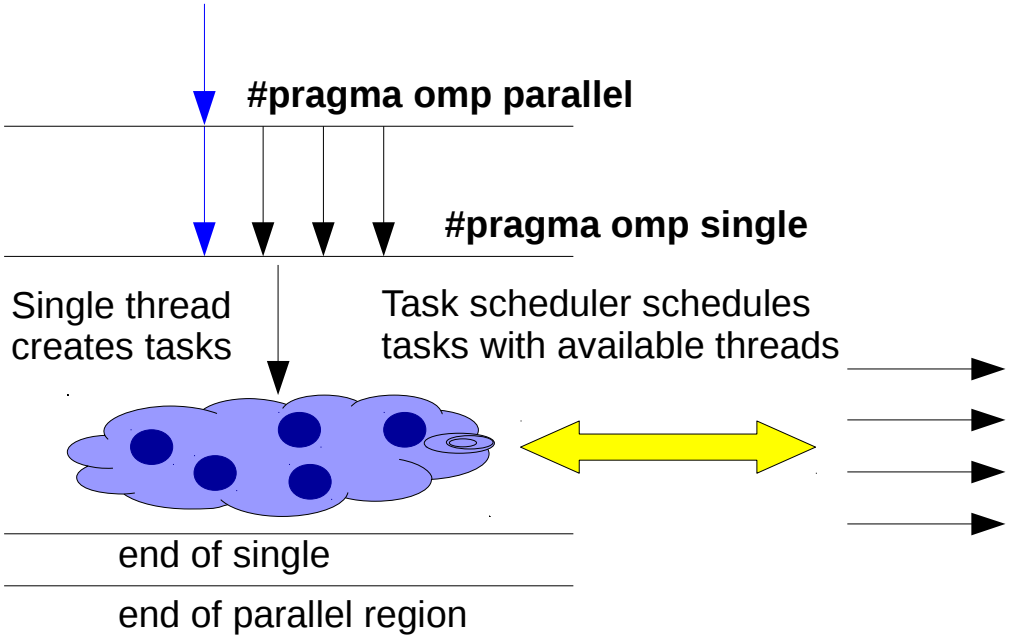


- Only one thread in the team executes the associated structured block
- Implicit barrier at the end
- All other threads wait until single block is finished
- *nowait* clause removes barrier at the end of single construct

Tasking construct

#pragma omp task [clause[[,] clause] ...] new-line
{
}

```
#pragma omp parallel
{
   #pragma omp single
   {
      While ( ptr != null)
      {
        #pragma omp task
        {
            //do_some_work();
        }
       } // end of while
   } // end of single
} // end of parallel
```

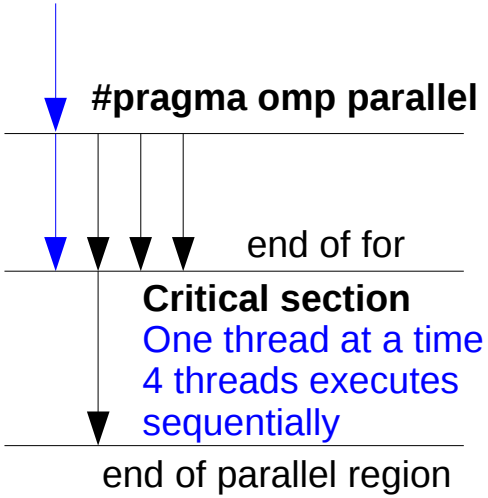


- Defines an explicit task
- A task is generated and corresponding data environment is created from the associated structured block
- Execution of task could be immediate or defer for later execution
- Examples : While-loop, recursion, sorting algorithms

Synchronization construct (1)

Critical construct

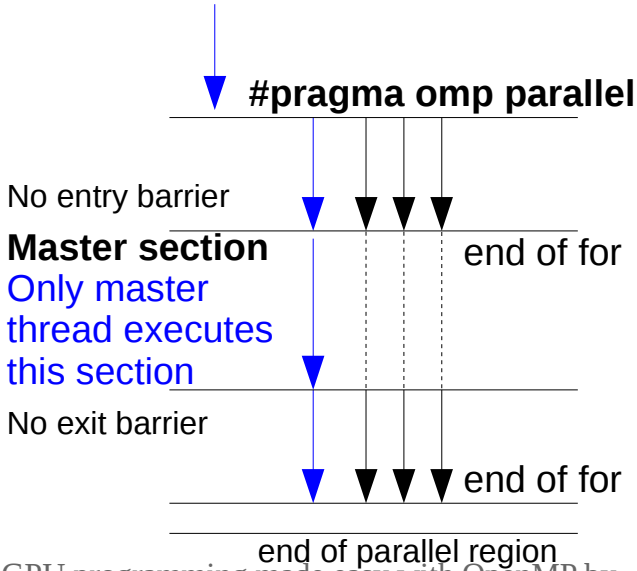
```
#pragma omp parallel
shared (sum) private (localSum)
{
   #pragma omp for
   for (i=0; i<N; i++)
   {
         localSum += a[i];
   }
   #pragma omp critical (global_sum)
   {
         sum += localSum;
   }
}
```



Master construct

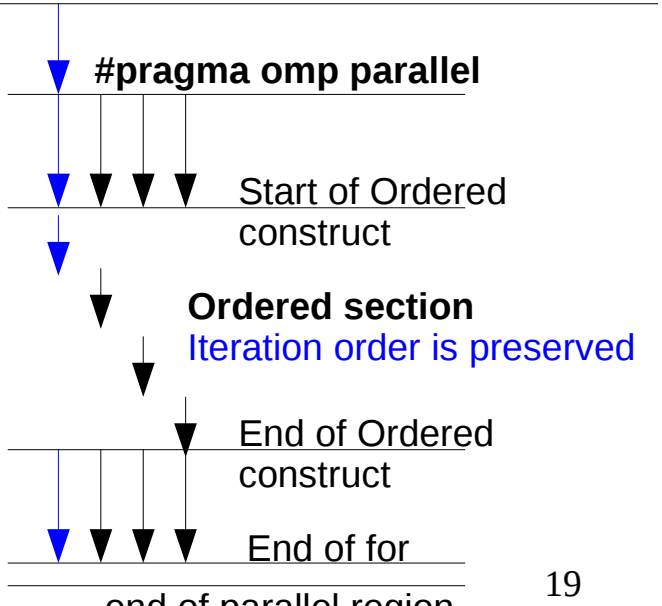
```
#pragma omp parallel
{
   #pragma omp for
         // do_pre_work()

   #pragma omp master
   {
         //write result to file
   }
   #pragma omp for
         // do_post_work()
}
```



Ordered construct

```
#pragma omp parallel
{
   #pragma omp for
   for (i=0; i<N; i++)
   {
      // do_some_work
      #pragma omp ordered
      {
         sum += a[i];
      }
      // do_post_work
   }
}
```



Synchronization construct (2)

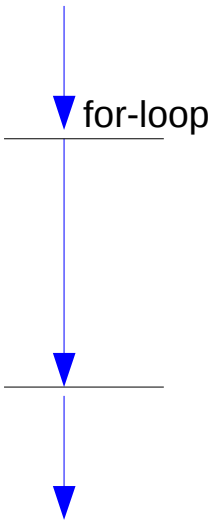
- **Barrier construct**
  - **#pragma omp barrier** new-line
  - Stand alone directive
  - “Must to execute” for all threads in parallel region
  - No one continues unless all threads reach barrier
- **Taskwait construct**
  - **#pragma omp taskwait** new-line
  - Stand alone directive
  - Waits until all child tasks completes execution before taskwait region
- **Atomic construct**
  - **#pragma omp atomic** [atomic-clause] new-line
  - ensures that a specific storage location is accessed atomically
  - Enforces atomicity for read, write, update or capture
- **Flush construct**
  - **#pragma omp flush** [(list)] new-line
  - Stand alone directive
  - Makes temporary view of thread’s memory consistent with main memory
  - When list is specified, the flush operation applies to the items in the list. Otherwise to all thread visible data items.

GPU offloading using OpenMP

Device construct (1)

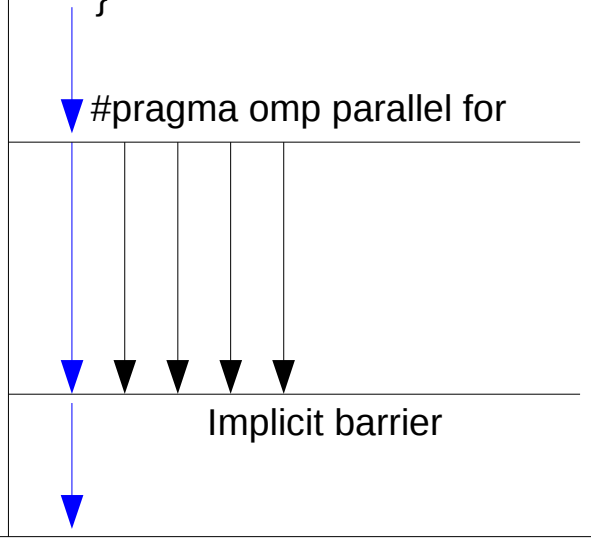
CPU (no threading)

```
for (i=0; i<N; i++)
{
        a[i] = b[i] + s * c[i];
}
```



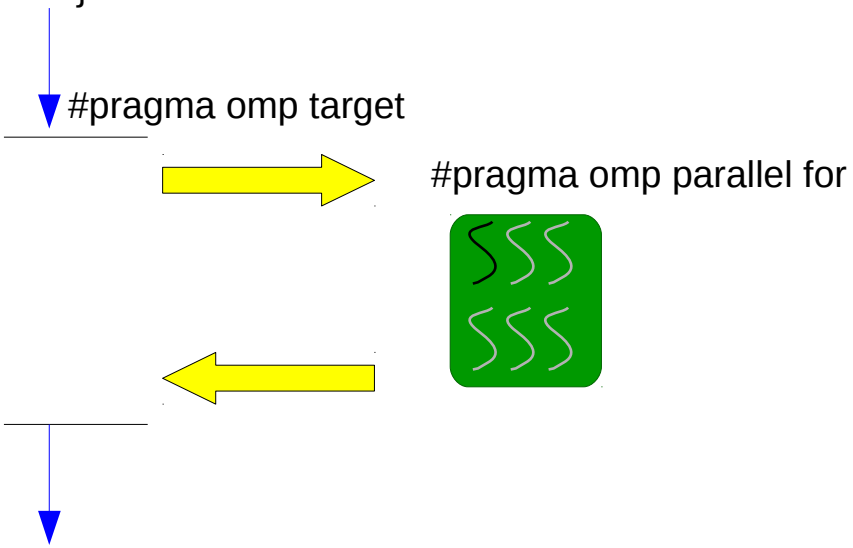
CPU OpenMP

```
#pragma omp parallel for
for (i=0; i<N; i++)
{
        a[i] = b[i] + s * c[i];
}
```



GPU OpenMP

```
#pragma omp target
#pragma omp parallel for
for (i=0; i<N; i++)
{
        a[i] = b[i] + s * c[i];
}
```



Note

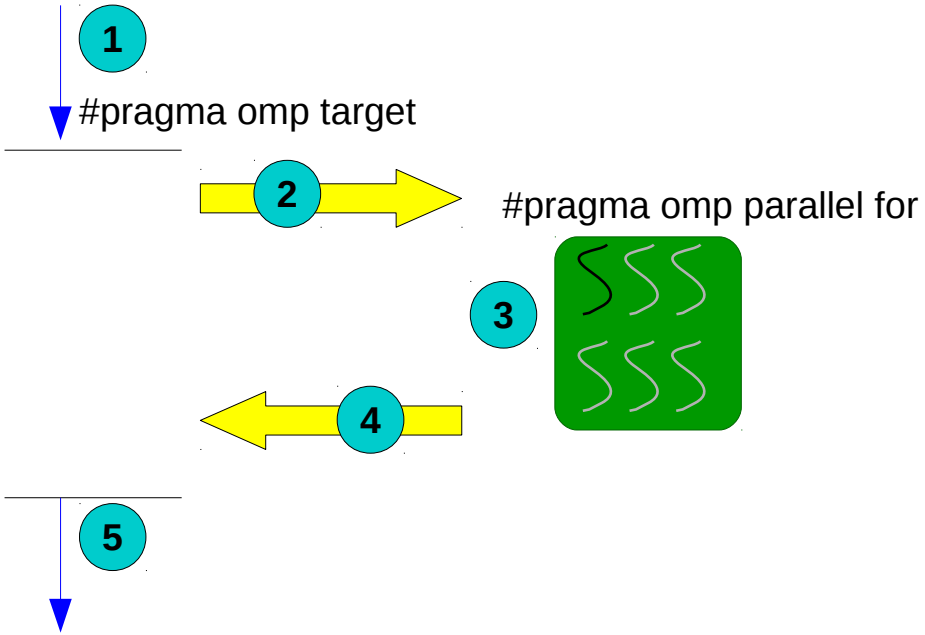
- No thread migration from one device to another.
- In absence of device or unsupported implementation for the target device, all target regions execute on the host.

Target construct

- Execute construct on device (GPU)
- Maps variables to device data environment

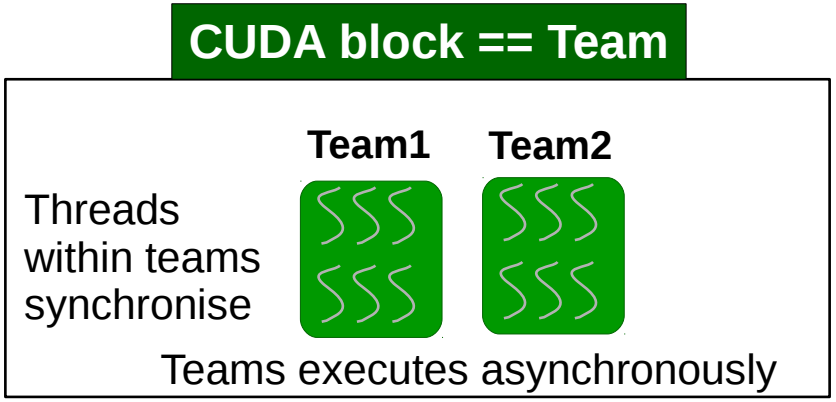
Device Construct (2)

```
#pragma omp target
#pragma omp parallel for
for (i=0; i<N; i++)
{
        a[i] = b[i] + s * c[i];
}
```



The host centric execution model

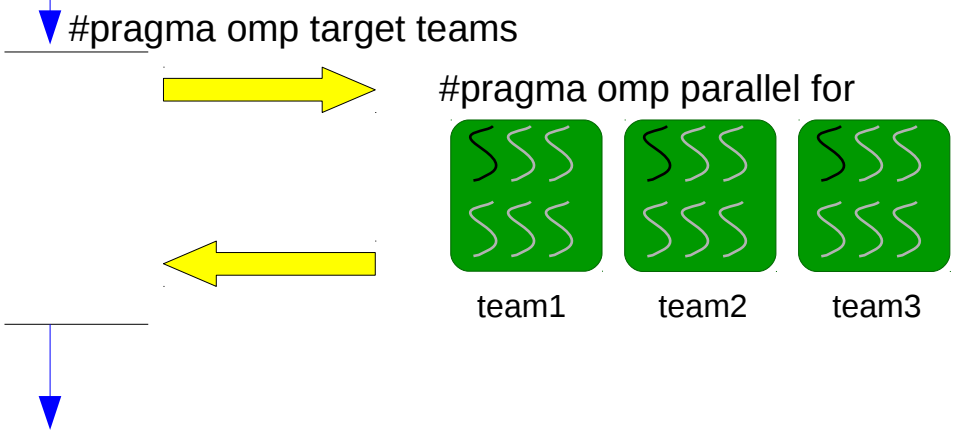
1. Thread starts on the host and host creates the data environments on the device(s).
2. The host then maps data to the device data environment. (data moves to device)
3. Host offloads target regions to target devices. (code executes on device)
4. After execution, host updates the data between the host and the device (transfers data from device to host)
5. Host destroys data environment on the device



Device construct (3)

GPU OpenMP

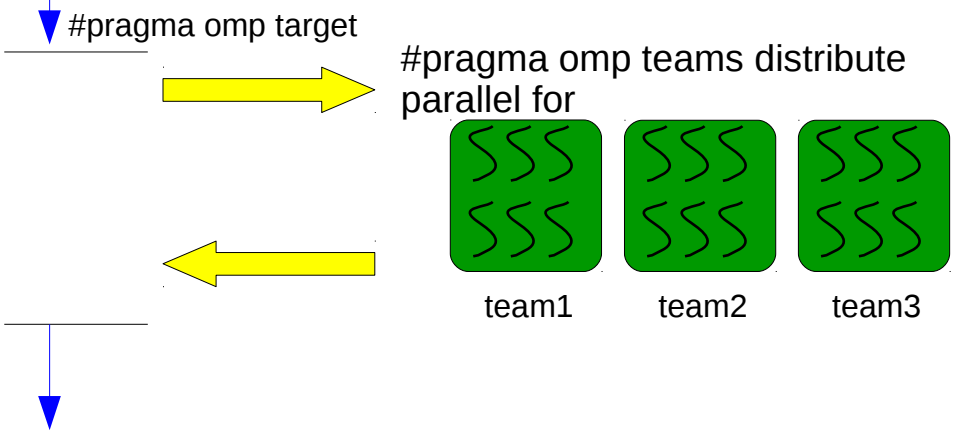
```
#pragma omp target teams
#pragma omp parallel for
for (i=0; i<N; i++)
{
        a[i] = b[i] + s * c[i];
}
```



Team construct

- Creates league of thread teams
- Master thread of each team executes the region
- No implicit barrier at the end of team construct

```
#pragma omp target
#pragma omp teams distribute parallel for
for (i=0; i<N; i++)
{
        a[i] = b[i] + s * c[i];
}
```



Distribute construct

- Distributes iterations among master thread of all teams
- No implicit barrier at the end of distribute construct
- Always associated with loops and binds to team region

Device construct (4)

- Target data
  - #pragma omp target data [clause] new-line

    {structured block}
  - Maps variables to device data environment and task encountering the construct executes the region
- Target enter data / Target exit data
  - #pragma omp target **enter** data [clause] new-line
  - #pragma omp target **exit** data [clause] new-line
  - Stand alone directive
  - Variables are mapped/unmapped to/from device data environment
  - Generates a new task which enclosed target data region

Device Data Mapping

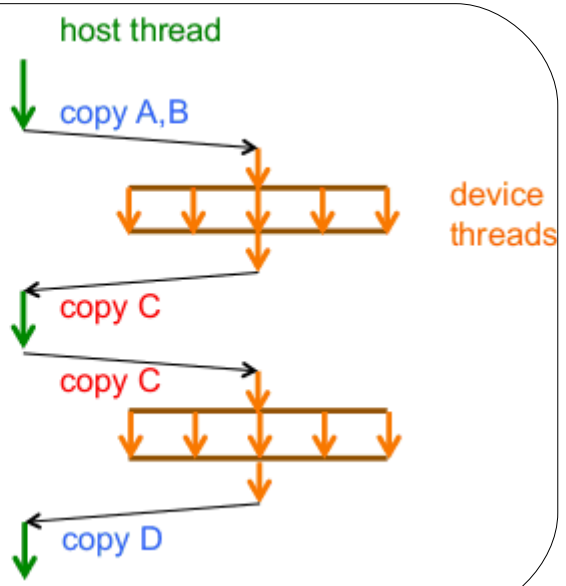
- Map clause
  - Maps variables from host data environment to corresponding variables in device data environment
  - map([[map-type-modifier[,]] map-type :] list)
  - map-type
    - to (initializes variables on device variables from host corresponding variables on host)
    - from (Writes back data from device to host)
    - tofrom – default map type (combination of to & from)
    - alloc (allocates device memory for variables)
    - release (reference count is decremented by one)
    - delete (deletes device memory of variables)

Device construct (5)

```
double A[n,n], B[n,n], C[n,n];

#pragma omp target \
   map(to: A, B) map(from: C)
{
  // define C in terms of A, B
}

#pragma omp target \
   map(to: C) map(from: D)
{
  // define D in terms of C
}
```




```
real(8), dimension(:), allocatable :: A, B, C
allocate(A(N), B(N), C(N))
call init (A, B, C)

$!omp target enter data map (alloc: C)
$!omp target enter data map (to: A, B)

call foo (A, B, C)

$!omp target exit data map (delete: C, B)
$!omp target exit data map (from: A)
```

```
double A[n,n], B[n,n], C[n,n];
#pragma omp target data map(alloc: C)
{
 #pragma omp target map(to: A, B) map(from: C)
 {
   // define C in terms of A, B
 }

 #pragma omp target map(from: D) map(to: C)
 {
   // define D in terms of C
 }
}
```

c is already in device scope

thus inner map clauses of c are ignored

Device construct (5)

- if clause
  - if (directive name modifier : scalar expression)
  - Used in target data, target enter/exit data, combined construct
  - In case of target clauses *if* clause expression evaluates to *false* then target region executed on host
- device (n)
  - Specifies the device (GPU) on which the target region to execute. ‘n’ is device id.
  - In absence of device clause, the target region executes on default device

OpenMP Clauses (1)

- **Schedule** [modifier : kind, chunk_size]
  - Kind
    - **Static** : Iterations are divided into chunks and assigned to threads in round-robin faction
    - **Dynamic** : Iterations are divided into chunks and assigned to threads on dynamically (first-come-first-serve basis)
    - **Guided** : Threads grab the chunk of iterations and gradually chunk reduces to *chunk_size* as execution proceeds
    - **Auto** : compiler or OpenMP runtime decides optimum schedule from static, dynamic or guided
    - **Runtime** : Deferred until runtime and schedule type and chunk size are taken from environment variable.
  - **Modifier** (new addition in OpenMP 4.5 specification)
    - **simd** : The chunk size is adjusted to simd width. Works only with SIMD construct e.g. schedule(simd:static, 5)
    - **Monotonic** : Chunks are assigned in increasing logical order of iterations to each threads.
    - **Nonmonotonic** : Assignment order of chunks to threads is unspecified.

OpenMP Clauses (2)

- **collapse (n)**
  - Collapses two or more loops to create larger iteration space
  - The parameter (n), specifies the number of loops associated with loop construct to be collapsed

```
#pragma omp parallel for collapse (2)
for (i=0; i<N; i++) {
    for (j=0; j<M; j++){
            for (k=0; k<M; k++){
          foo(i, j, k);
}}}
```

- **nowait**
  - Used for asynchronous thread movement in worksharing construct i.e. threads that finish early proceed to the next instruction without waiting for other thread.
  - Used in loop, sections, single, target and so on

Data Sharing Attribute Clauses (1)

- **Private (list)**
  - Declares variables as to be private to a task or thread
  - All references to the original variable are replaced by references to new variable when respective directive is encountered
  - Private variables are not initialized by default
- **shared (list)**
  - Declares variables to be shared by tasks or threads
  - All references to the variable points to the original variable when respective directive is encountered
- **firstprivate (list)**
  - Initializes private variable with value of original variable prior to execution of the construct
  - Used with parallel, task, target, teams etc.
- **lastprivate (list)**
  - Writes the value of the private variable back to original variable on existing the respective construct
  - Updated value is sequentially last iteration or section of worksharing construct.

Data Sharing Attribute Clauses (2)

- reduction (reduction identifier : list)
  - Updates original value with the final value of each of the private copies, using the combiner – operator or identifier
  - Reduction identifier is either identifier (max, min) or operator (+, -, * , &, |, ˆ, && and ||)
- threadprivate (list)
  - Makes global variables (file scope, static) private to each thread
  - Each thread keep its own copy of global variable

Data Copying Clauses

Copy value of private or threadprivate variable from one thread to another thread

- copyin(list)
  - Allowed on parallel and combined worksharing construct
  - Copy value of threadprivate variable of master thread to threadprivate variable of other threads before start of execution
- copyprivate(list)
  - Allowed on single construct
  - Broadcast the value of private variable from one thread to other threads in team

OpenMP Runtime library (1)

- Runtime library definitions
  - omp.h stores prototype of all OpenMP routines/functions and type.
- Execution environment routines (35+ routines)
  - Routines that affects and monitor threads, processors and parallel environment
    - void omp_set_num_threads(int num_threads);
    - int omp_get_thread_num(void);
    - int omp_get_num_procs(void);
    - void omp_set_schedule(omp_sched_t kind, int chunk_size);
- Timing routines
  - routines that support a portable wall clock timer
  - double omp_get_wtime(void);
  - double omp_get_wtick(void);

OpenMP Runtime library (2)

- Lock routines (6 routines)
  - Routines for thread synchronization
  - OpenMP locks are represented by OpenMP lock variables
  - 2 types of locks
    - Simple : can be set only once
    - Nestable : can be set multiple time by same task
  - Example APIs
    - void omp_init_lock(omp_lock_t * lock); void omp_destroy_lock(omp_lock_t * lock);
    - void omp_set_nest_lock(omp_nest_lock_t * lock); void omp_unset_nest_lock(omp_nest_lock_t * lock);
    - int omp_test_lock(omp_lock_t * lock);
- Device memory routines (7 routines)
  - routines that support allocation of memory and management of pointers in the data environments of target devices
    - void* omp_target_alloc(size_t size, int device_num);
    - void omp_target_free(void * device_ptr, int device_num);
    - int omp_target_memcpy(void * dst, void * src, size_t length, size_t dst_offset, size_t src_offset, int dst_device_num, int src_device_num);

OpenMP Environment Variables

1	OMP_SCHEDULE	sets the run-sched-var ICV that specifies the runtime schedule type and chunk size. It can be set to any of the valid OpenMP schedule types.
2	OMP_NUM_THREADS	sets the nthreads-var ICV that specifies the number of threads to use for parallel regions.
3	OMP_DYNAMIC	sets the dyn-var ICV that specifies the dynamic adjustment of threads to use for parallel regions
4	OMP_PROC_BIND	sets the bind-var ICV that controls the OpenMP thread affinity policy.
5	OMP_PLACES	sets the place-partition-var ICV that defines the OpenMP places that are available to the execution environment.
6	OMP_NESTED	sets the nest-var ICV that enables or disables nested parallelism.
7	OMP_STACKSIZE	sets the stacksize-var ICV that specifies the size of the stack for threads created by the OpenMP implementation.
8	OMP_WAIT_POLICY	sets the wait-policy-var ICV that controls the desired behavior of waiting threads.
9	OMP_MAX_ACTIVE_LEVELS	sets the max-active-levels-var ICV that controls the maximum number of nested active parallel regions.
10	OMP_THREAD_LIMIT	sets the thread-limit-var ICV that controls the maximum number of threads participating in a contention group.
11	OMP_CANCELLATION	sets the cancel-var ICV that enables or disables cancellation.
12	OMP_DISPLAY_ENV	instructs the runtime to display the OpenMP version number and the initial values of the ICVs, once, during initialization of the runtime.
13	OMP_DEFAULT_DEVICE	sets the default-device-var ICV that controls the default device number.
14	OMP_MAX_TASK_PRIORITY	sets the max-task-priority-var ICV that specifies the maximum value that can be specified in the priority clause of the task construct.

Best practices

- Always initialize data (arrays) with parallel code to benefit from first touch placement policy on NUMA nodes
- Bind OpenMP threads to cores (see OpenMP thread affinity policy)
- Watch for false sharing and add padding to arrays if necessary
- Collapse loops to increase available parallelism
- Always use teams and distribute to expose all available parallelism
- Avoid small parallel blocks on GPUs and unnecessary data transfer between host and device.
- Overlap data transfer with compute using asynchronous transfers