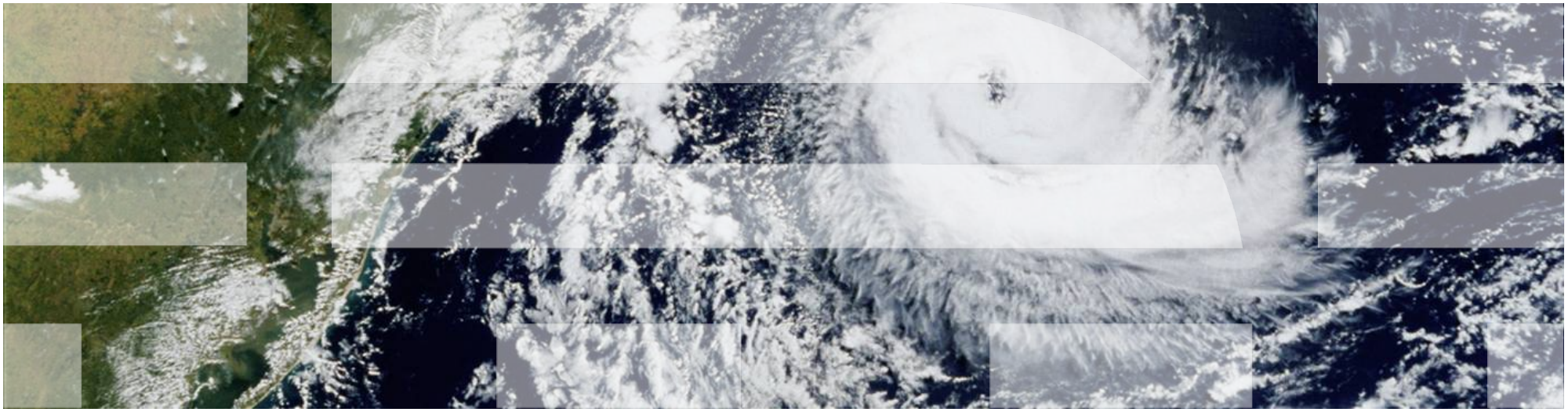
# Programming Techniques for Supercomputers

*(Heterogeneous and Distributed systems)*

**– Aditya Nitsure (IBM India)**



# OpenMP (Open Multi-Processing)

(https://www.openmp.org)

# OpenMP Directives

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

# Content – day2 (3rd OCT)

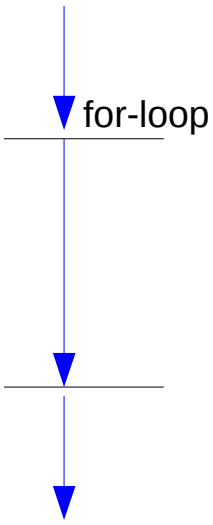
- Parallel constructs for GPU programming
- OpenMP Clauses
- Data sharing attributes
- Runtime library
- Environment variables
- Best practices
- Hands-on – Implement OpenMP directives in matrix multiplication program

# GPU offloading using OpenMP

# Device construct (1)

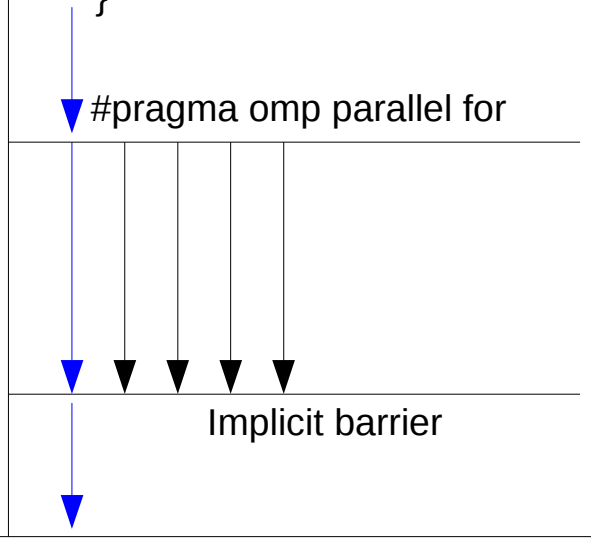
**CPU** (no threading)

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



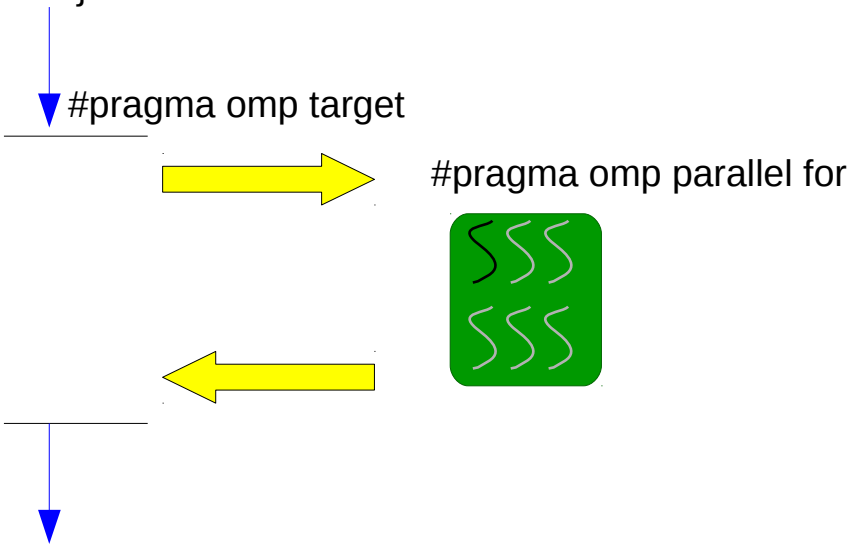

**CPU OpenMP**

```
#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**

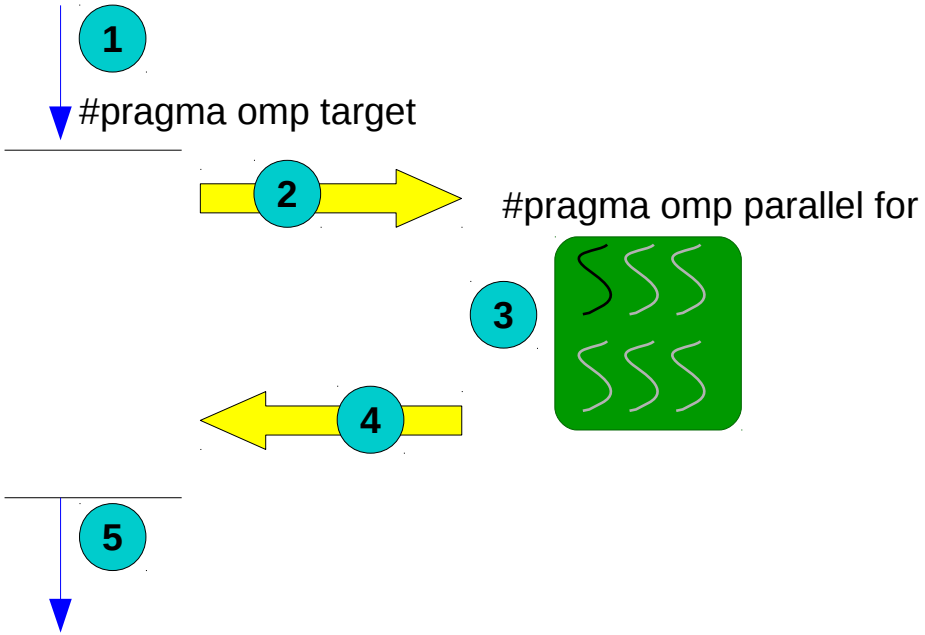
- No thread migration from one device to another.
- In absence of device or unsupported implementation for the target device, all target regions execute on the host.

## Target construct

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

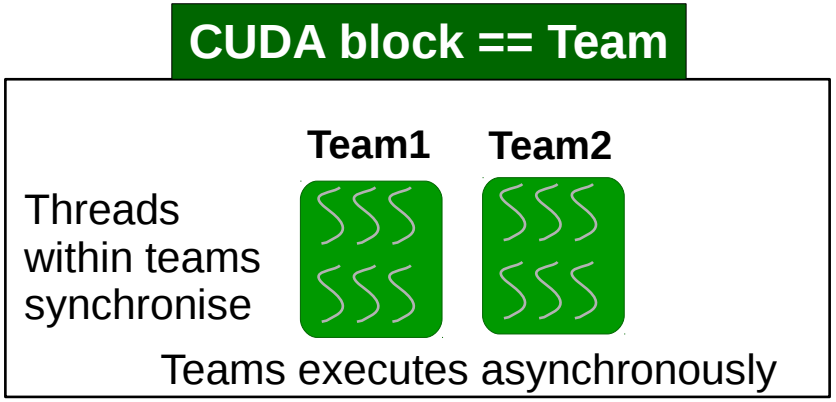
# Device Construct (2)

```
#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

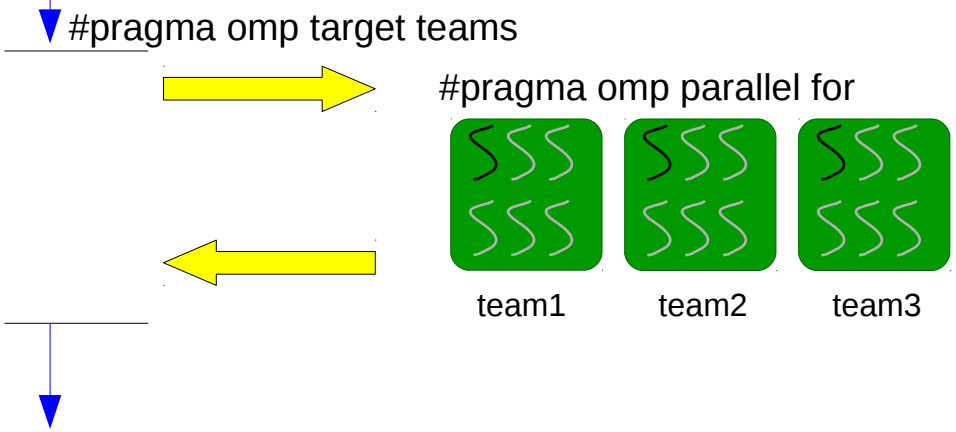
1. Thread starts on the host and host creates the data environments on the device(s).
2. The host then maps data to the device data environment. (data moves to device)
3. Host offloads target regions to target devices. (code executes on device)
4. After execution, host updates the data between the host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



# Device construct (3)

**GPU OpenMP**

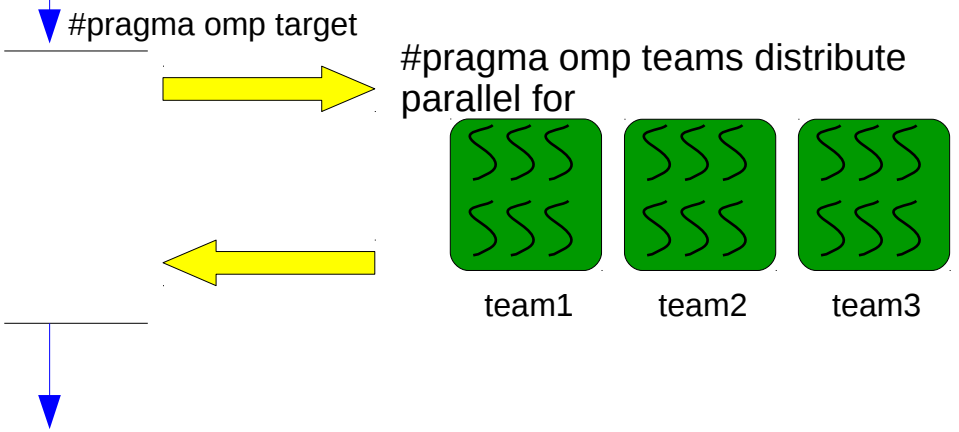
```
#pragma omp target teams
#pragma omp parallel for
for (i=0; i<N; i++)
{
        a[i] = b[i] + s * c[i];
}
```




## 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

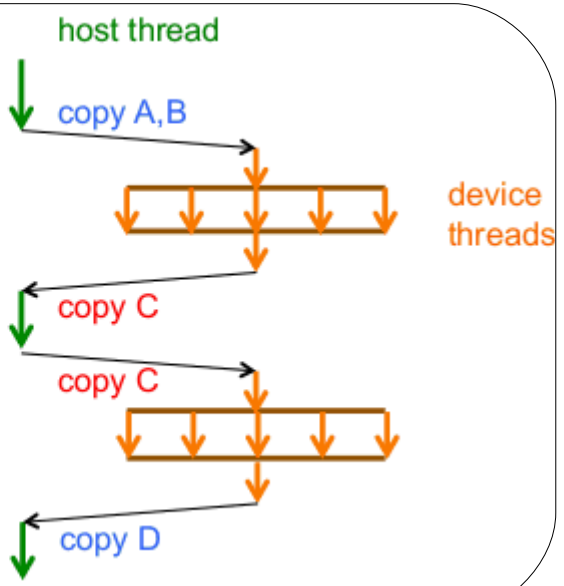
- Map clause
  - Maps variables from host data environment to corresponding variables in device data environment
  - map([ [map-type-modifier[,]] map-type : ] list)
  - map-type
    - to (initializes variables on device variables from host corresponding variables on host)
    - from (Writes back data 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

# Example submission

- Create your own github account (https://github.com/)
- Code submission
  - Create directories for respective codes
    - *mm_sequential* – for sequential matrix multiplication code
    - *mm_omp* – for OpenMP matrix multiplication code
    - *mm_mpi* – for MPI matrix multiplication code
- Upload your code at github and share github link.

# *git* useful commands

- Git clone <git repo name>
- Git add <list of files/directories>
- Git commit -m “<info about commit>”
- Git push origin master
- Git status

# Hands-on

# Revision

1. How will you set total number of thread in OpenMP?
2. How to compile OpenMP code?
3. Which is fundamental construct in OpenMP?
4. What is combined construct?
5. Which are worksharing construct in OpenMP?
6. What are all synchronization constructs in OpenMP?