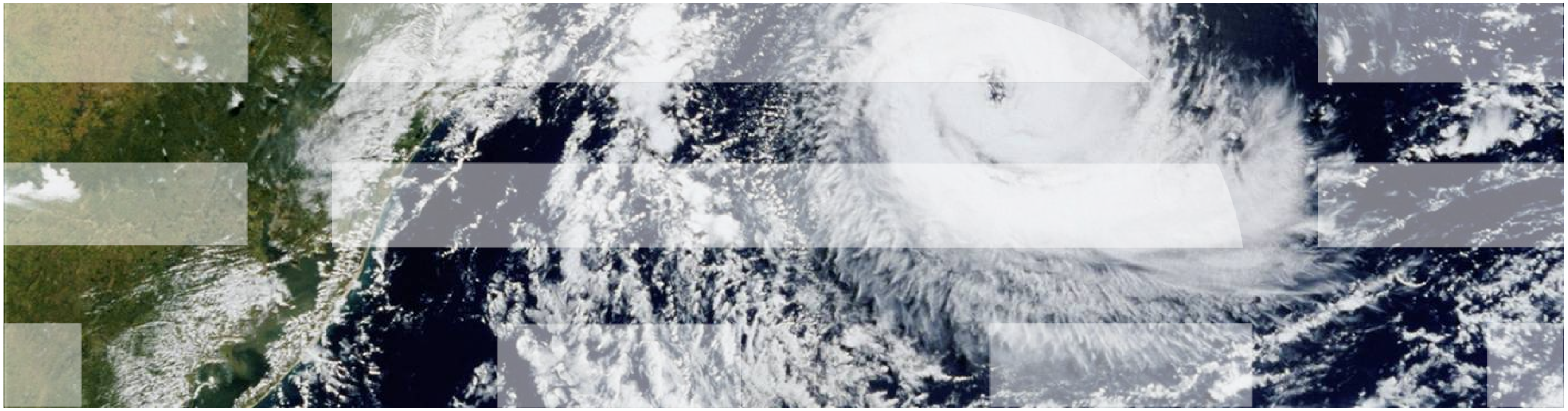
# Programming Techniques for Supercomputers

*(Heterogeneous and Distributed systems)*

**– Aditya Nitsure (IBM India)**



## 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?

## Content – day1 (2rd OCT)

- Introduction
- Compiling OpenMP program
- OpenMP CPU directives
  - Parallel construct
  - SIMD construct
  - Combined construct
  - Work sharing construct
  - synchronization construct
  - Tasking construct
- Hands-on – Experiment with OpenMP directives

## 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

## 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)

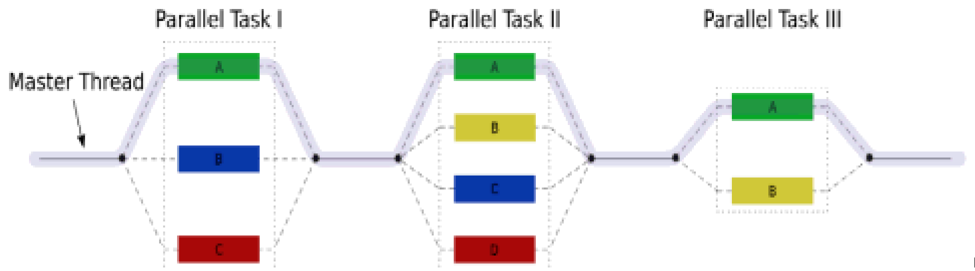
- Shorter learning curve
- Supported by many compiler vendors
  - GNU, LLVM, AMD, IBM, Cray, Intel, PGI, OpenUH, ARM
  - Compilers from research labs e.g. LLNL, Barcelona Supercomputing Centre
- Easy to maintain sequential, CPU parallel and GPU parallel versions of code.
- Used in application development 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();

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

**NVIDIA CUDA linking**

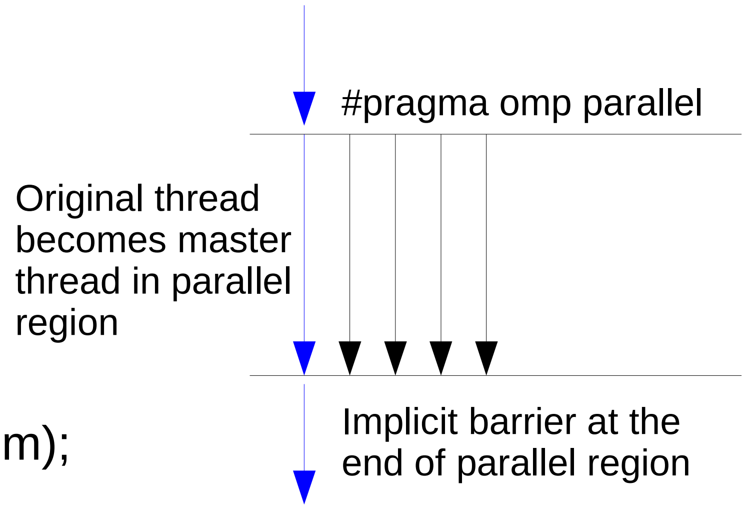
-lcudart -L/usr/local/cuda/lib64

## 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
Original thread becomes master thread in parallel region
Implicit barrier at the end of parallel region

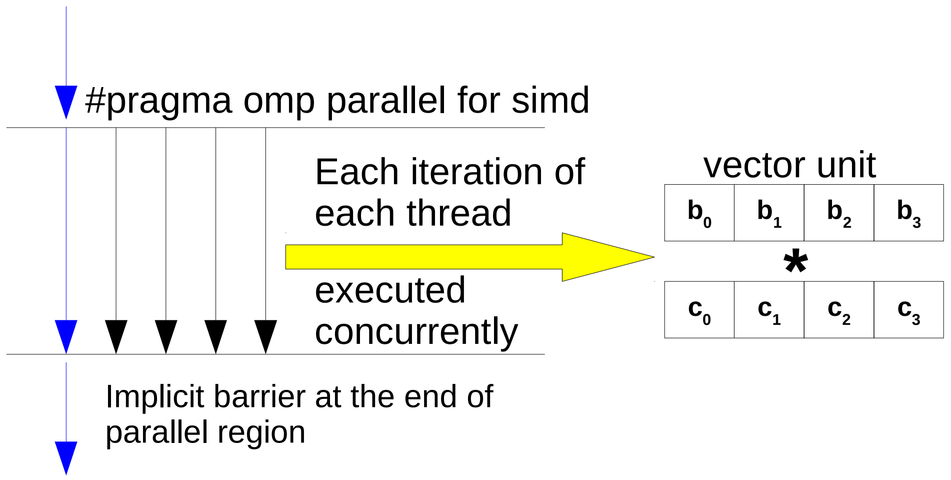
```
#pragma omp parallel
{
      int threadNum = omp_get_thread_num();
      printf("Hello World from thread %d \n", threadNum);
}
```

- 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]
}
```


#pragma omp parallel for simd
Each iteration of each thread executed concurrently
vector unit
$b_0$ $b_1$ $b_2$ $b_3$
*
$c_0$ $c_1$ $c_2$ $c_3$
Implicit barrier at the end of parallel region

- 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

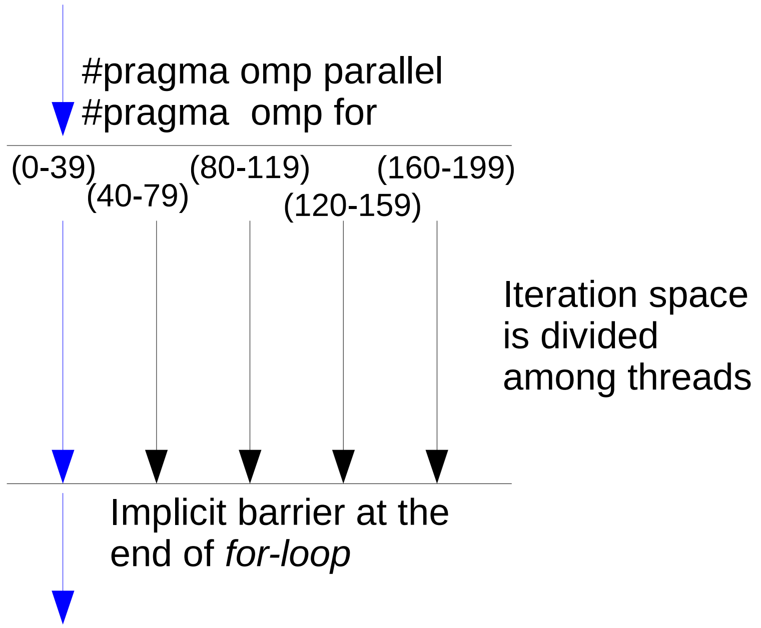
- Distributes the execution of the associated parallel region among the members of the team
- Implicit barrier at the end
- Types
  - Loop construct
  - Sections construct
  - Single construct
  - Workshare construct (only in Fortran)

# 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]
        }
}
```


#pragma omp parallel
#pragma  omp for
(0-39)
(40-79)
(80-119)
(120-159)
(160-199)
Iteration space is divided among threads
Implicit barrier at the end of *for-loop*

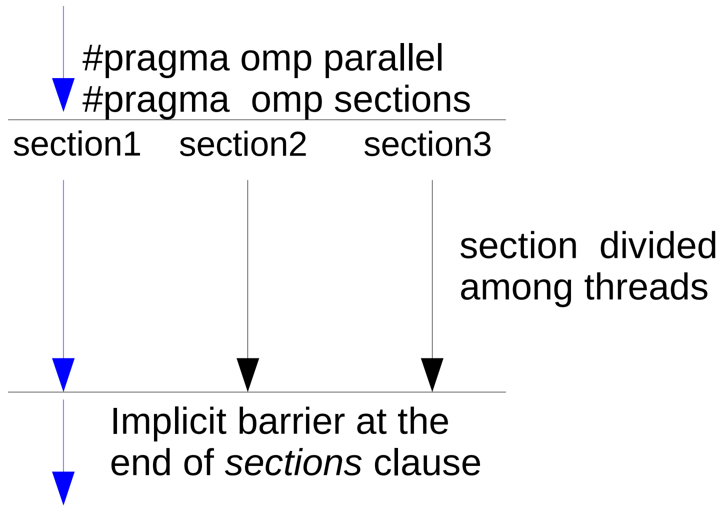
- Iterations of *for-loop* distributed among the threads
- One or more iterations executed in parallel
- Implicit barrier at the end unless *nowait*

# Worksharing : Section construct

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

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

- 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
section1
section2
section3
section  divided among threads
Implicit barrier at the end of *sections* clause

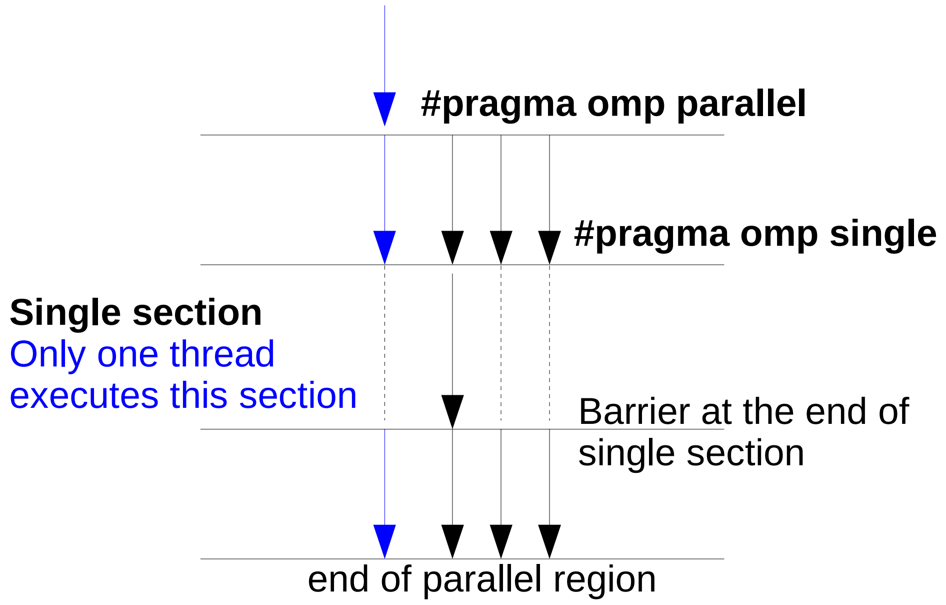
## Worksharing : Single construct

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

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

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

    // do_post_work()
}
```

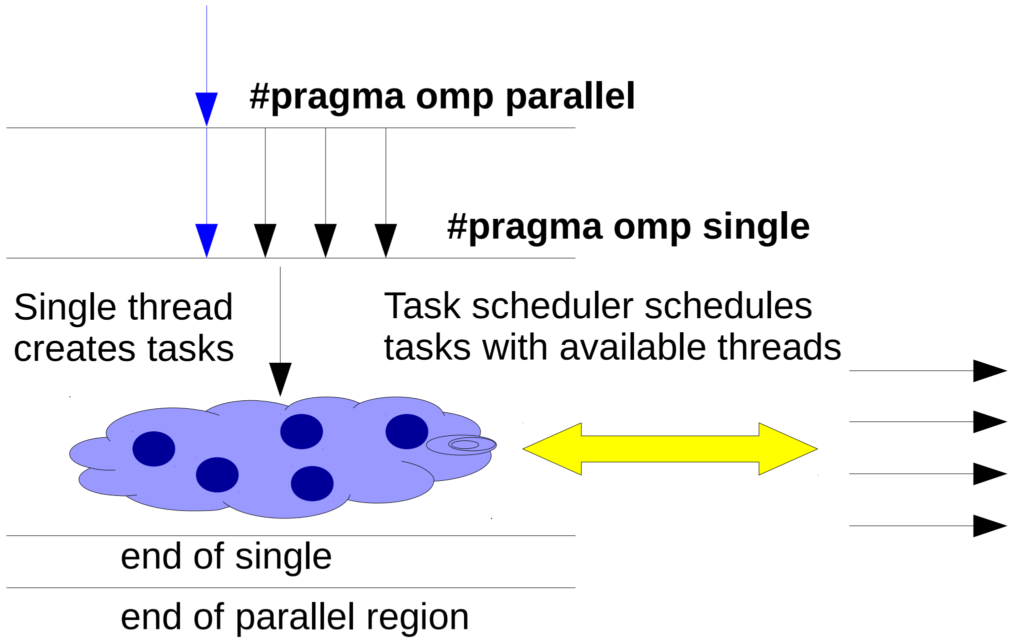


- Only one thread in the team executes the associated structured block
- Implicit barrier at the end
- All other threads wait until single block is finished
- *nowait* clause removes barrier at the end of single construct

# 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
```



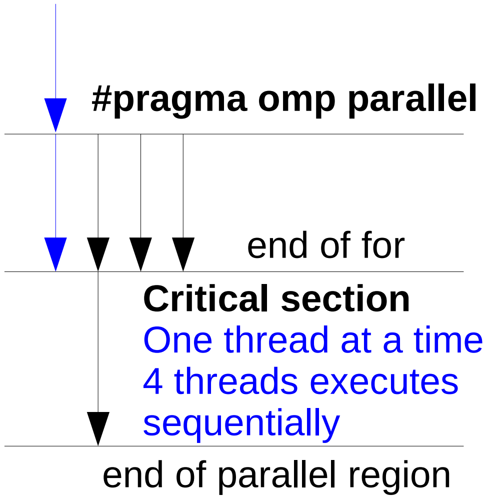
#pragma omp parallel
#pragma omp single
Single thread creates tasks
Task scheduler schedules tasks with available threads
end of single
end of parallel region

- 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**

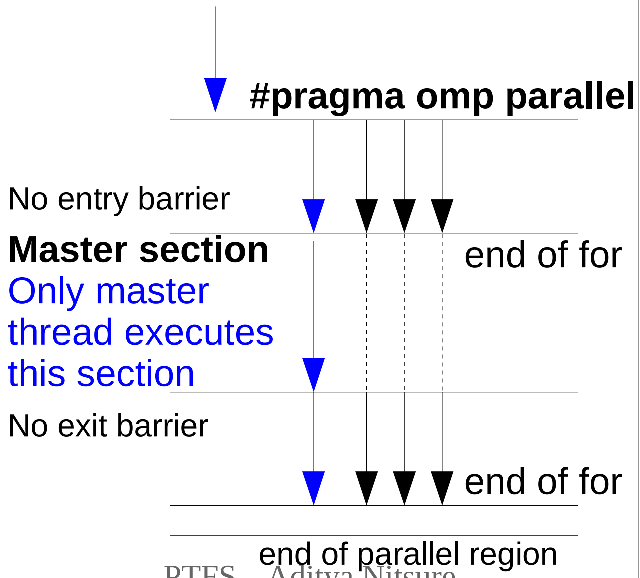
```
#pragma omp parallel
shared (sum) private (localSum)
{
    #pragma omp for
    for (i=0; i<N; i++)
    {
            localSum += a[i];
    }
    #pragma omp critical (global_sum)
    {
            sum += localSum;
    }
}
```



**Master construct**

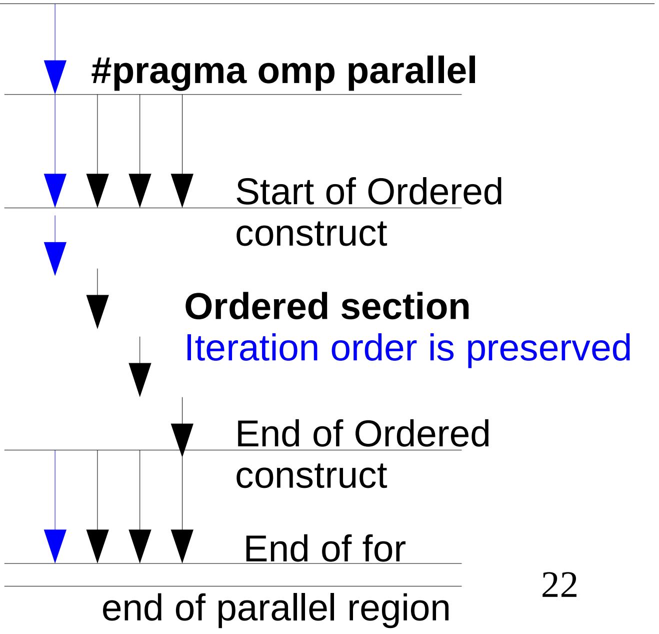
```
#pragma omp parallel
{
    #pragma omp for
            // do_pre_work()

    #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.

## Hands-on

### (OpenMP directive experimentation)