OpenMP

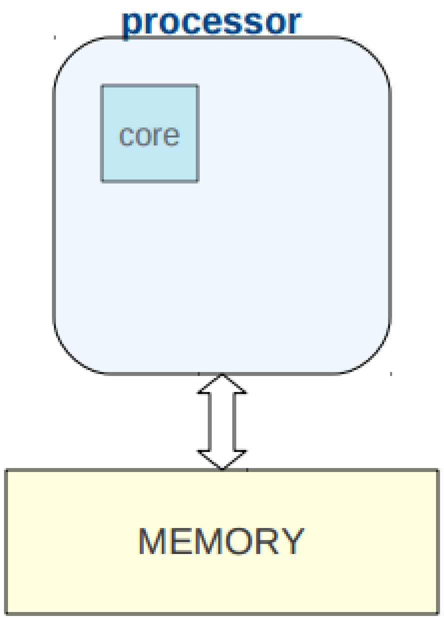
Open Specifications for Multi Processing

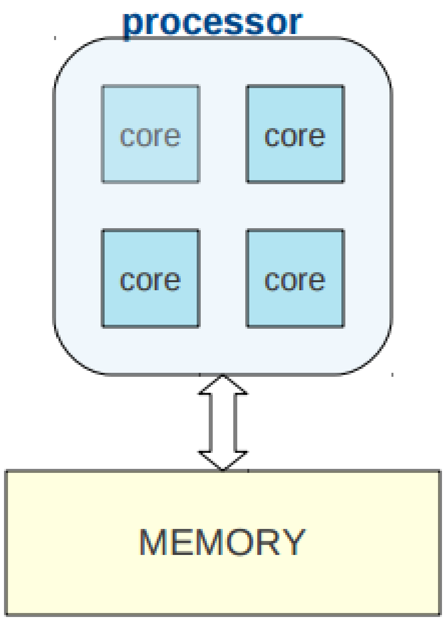
Raghu.H.V,
Scientist -D,
C-DAC, Bengaluru

OUTLINE

- Introduction
- OpenMP Programming Model
- OpenMP Directives
- Synchronization Constructs
- Runtime Libraries
- Environment Variables

Introduction





Older processors have only one CPU core to execute instructions

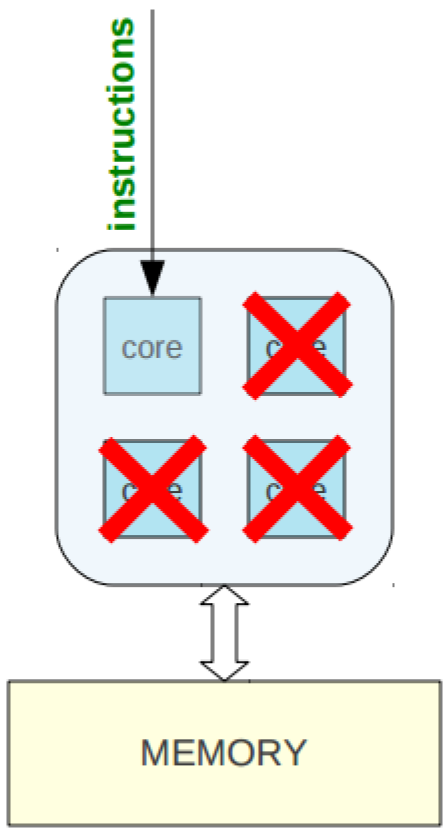
Modern processors have many CPU cores to execute instructions

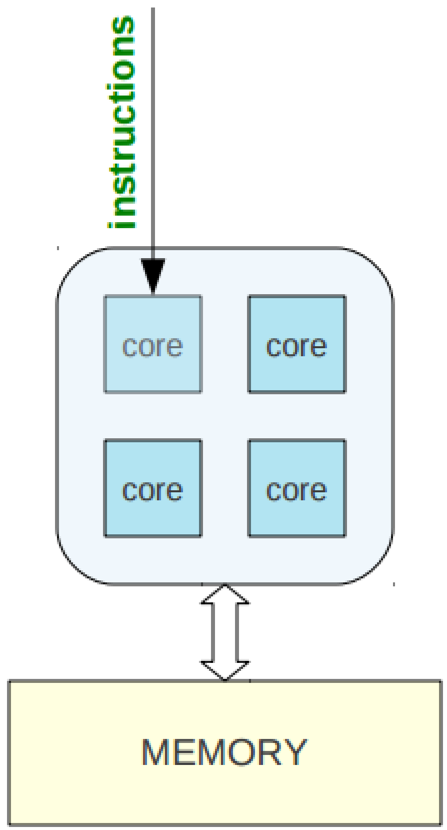
Why OpenMP?

When you run a Sequential Program:

- Instructions executed on 1 core
- Other cores are idle
- Wastage of available resources – We want all cores to be used – How?

Use "OpenMP"





INTRODUCTION TO OPENMP

- Standard API for writing shared memory parallel applications in C, C++, and Fortran
- OpenMP API Consists of :
  - Compiler Directives
  - Runtime Library Routines
  - Environment variables
- Specification maintained by the OpenMP Architecture Review Board (http://www.openmp.org)

- Scenarios
  - Creating new program
  - Parallelizing existing one

- Use Explicit Parallelization – OpenMP
  - When compiler cannot find parallelism

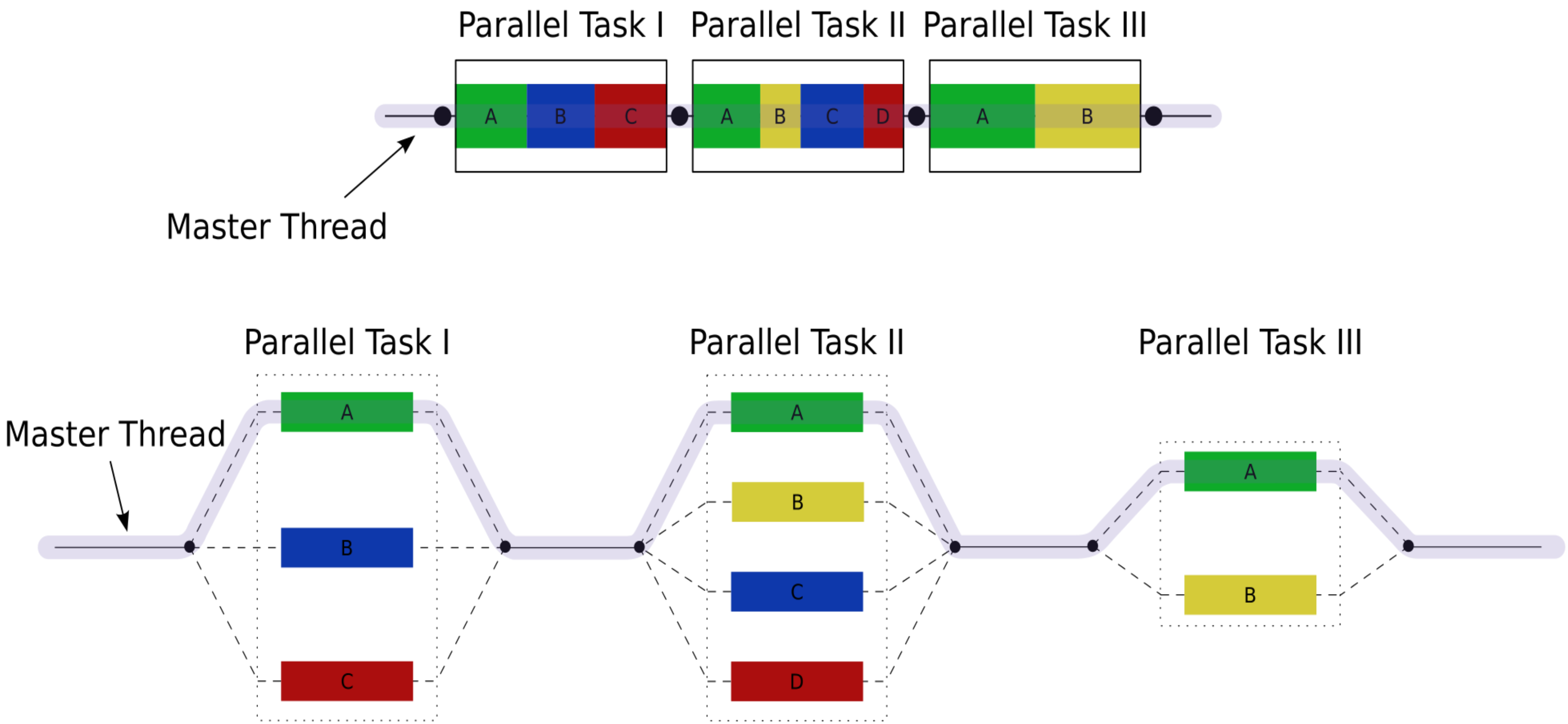
Characteristics of OpenMP Execution Model

- Thread Based Parallelism
- Explicit Parallelism
- Fork - Join Model
- Compiler Directive Based

Threads v/s Process

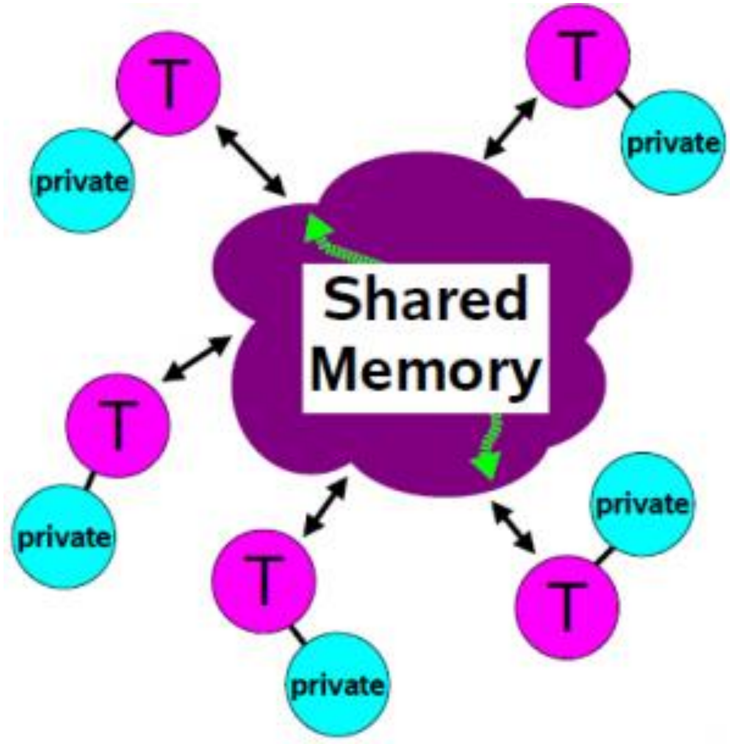
- A process is a program in execution. A thread is a light weight process.
- Threads share the address space of the process that created it, process have their own address space.
- Threads can directly communicate with other threads of the same process. Processes must use IPC to communicate with other process.
- Changes to main thread may affect behaviour of other threads of the process. Changes to the parent process do not affect child process.

Fork-Join Model



Memory Model

- Shared Memory.
- Data is private or shared.
- Private accessed only by owned threads.
- Synchronization takes place.



How do threads interact?

- OpenMP is an multi-threading, shared memory model.
  - Threads communicate by sharing variables.

- Unintended sharing of data causes race conditions:
  - Race condition: when the program's outcome changes as the threads are scheduled differently.

- To control race conditions:
  - Use synchronization to protect data conflicts.

- Synchronization is expensive so:
  - Change how data is accessed to minimize the need for synchronization.

General Syntax

- Header file
  - #include <omp.h>
- Parallel Region

```
#pragma  omp  parallel [clauses...]
{
   // … do some work here
} // end of parallel region/block
```

- **Functions and Environment variables**

An Example – Hello World

Sequential

```
void main( )
{
        int ID = 0;
        printf(" Thread: %d - Hello World ", ID);
}
```

OpenMP include file

OpenMP

```
#include <omp.h>
void main( )
{
        #pragma omp parallel
        {
                int ID = 0;
                printf(" Thread: %d - Hello World ", ID);
        }
}
```

Parallel region with default Number of threads

End of parallel region

Compilation

- GNU Compiler Example :
  - gcc -o helloc.x -fopenmp hello.c
- IBM AIX compiler :
  - xlc -o helloc.x -qsmp=omp hello.c
- Portland group compiler:
  - pgcc -o helloc.x -mp hello.c
- Intel Compiler Example:
  - icc -o helloc.x -openmp hello.c

OpenMP Components

Compiler Directives

A. Parallel construct
B. Work Sharing
C. Synchronization
D. Data Scope
   - Private
   - shared

Runtime Library Routines

- Number of threads
- Thread ID

Environment Variables

- Number of threads
- Scheduling Type

OpenMP Compiler Directives

- Used to divide blocks of code among threads.
- Distributing loop iterations among threads.
- Serializing sections of code.
- Synchronization of work among threads.

Runtime Library Routines

- Setting and querying the number of threads.
- Querying unique thread ID.
- To check whether inside parallel region.

Environment Variables

- Setting the number of threads.
- Specifying how loop iterations are divided.

1. OpenMP Compiler Directives

- Syntax:
  - #pragma omp directive-name [clause1 clause2..] new-line
- Example:
  - #pragma omp parallel private(pi)

- Example Directives
  - parallel
  - for/do
  - sections
  - single
  - critical
  - barrier

A. Parallel Directive

- The parallel construct forms a team of threads and starts parallel execution of a parallel region.
- A parallel region is a block of code that will be executed by multiple threads.

```
#pragma omp parallel   [clause1   clause2   …] newline
structured_block
```

Clauses:

if (scalar_expression)

private (list)

firstprivate (list)

num_threads (integer-expression)

Example: parallel construct

```
#include <omp.h>
main ()
{
int nthreads, tid;
/* Fork a team of threads with each thread having a private tid */
#pragma omp parallel private(tid)
{
        /* Obtain and print thread id */
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);

        /* Only master thread does this */
        if (tid == 0)
        {
                nthreads = omp_get_num_threads();
                printf("Number of threads = %d\n", nthreads);
        }

 }  /* Implicit Barrier - All threads join master thread and terminate */
}
```

- Main thread creates a team of threads and becomes the master of the team.
- The master is a member of that team and has thread id 0 within that team.
- There is an implied barrier at the end of a parallel section.
- If any thread terminates within a parallel region, all threads in the team terminate.

DIFFERENT WAYS OF THREAD CREATION

- The number of threads in a parallel region is determined by the following factors, in order of precedence (Low-High):

  1. Default number of Threads (number of cores)
  2. Setting of the OMP_NUM_THREADS environment variable

     export OMP_NUM_THREADS=2

  3. Use of the omp_set_num_threads() library function

     omp_set_num_threads(4);

  4. Setting of the NUM_THREADS clause

     #pragma omp parallel private(tid) num_threads(6)

- Implementation default - usually the number of cores.
- Threads are numbered from 0 (master thread) to N-1

B. Work Sharing Constructs

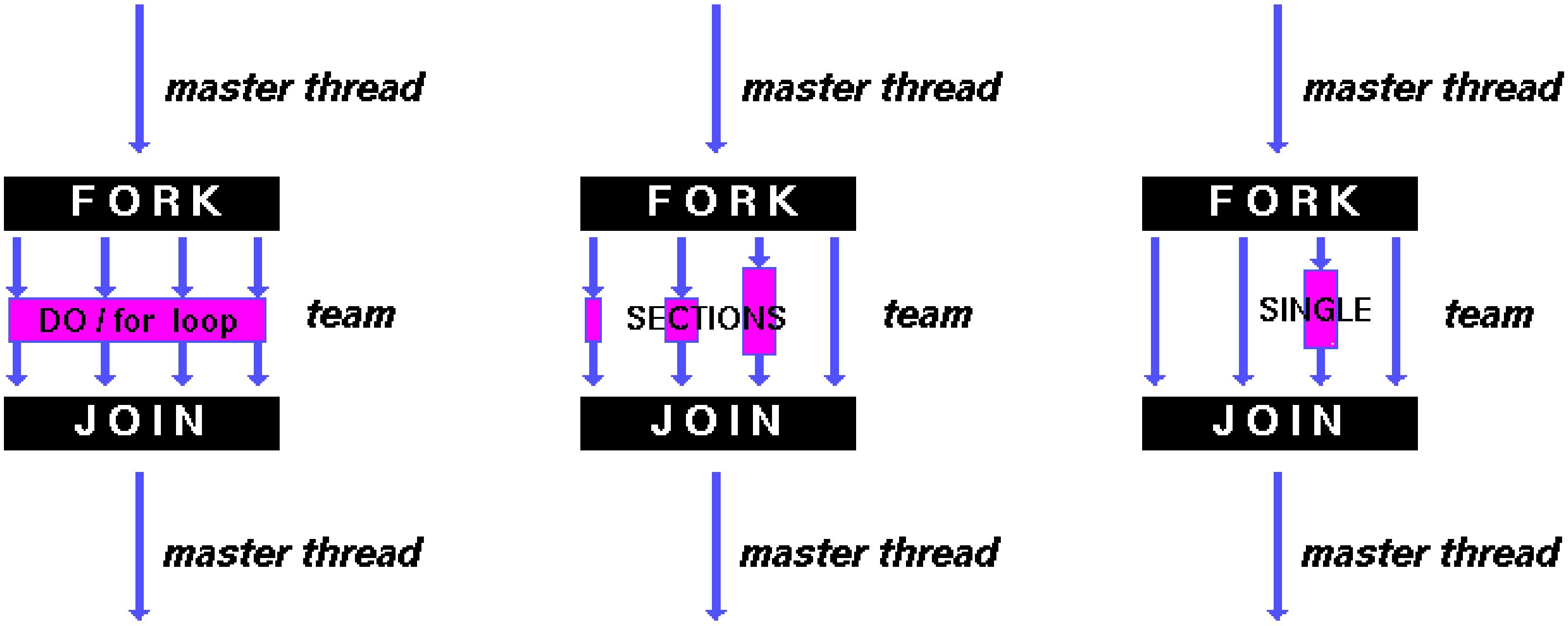
- Divides execution of code region among members of the team of threads.

- Work-sharing constructs do not launch new threads

- Restrictions
  - Must be enclosed within a parallel region.
  - Work is distributed among the threads

Examples:

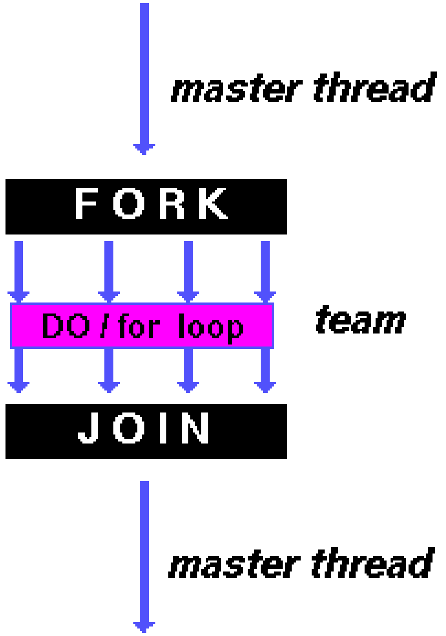
- for
- section
- single – serializes a portion of code



For Directive

- for directive divide the loop iterations among threads and execute in parallel.
- Syntax:

  #pragma omp for [clause1   clause2   ...] newline
   for_loop



Example: for construct

```
#include <omp.h>
#define N  1000

main ()
{
int i;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)  a[i] = b[i] = i * 1.0;

#pragma omp parallel shared(a,b,c) private(i)
{
        #pragma omp for
        for (i=0; i < N; i++)
                c[i] = a[i] + b[i];
 }  /* end of parallel section */
}
```

RESTRICTIONS

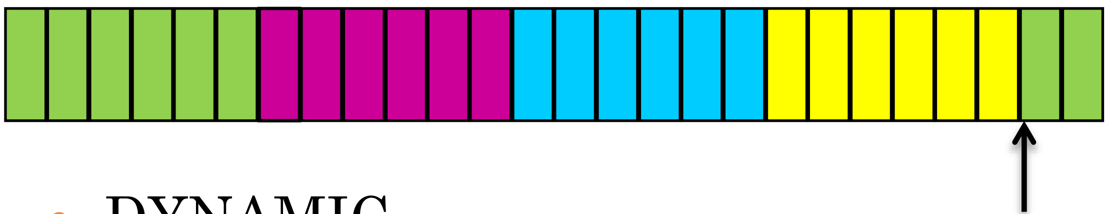
for (index = start ; index < end ; increment_expr)

It should be possible to determine the number of loop iterations before execution, to divide the iterations.

- increment must be same at each iteration
- all loop iterations must be done.

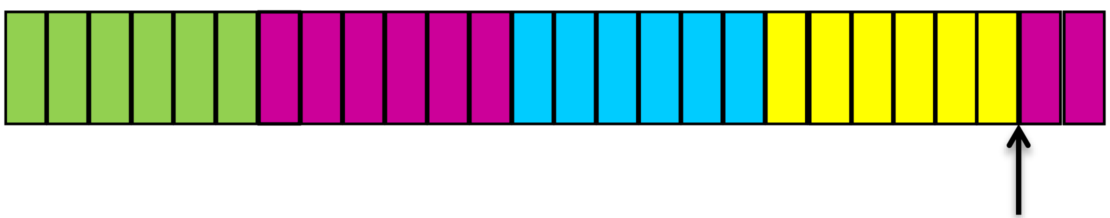
SCHEDULE: Describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent.

- STATIC



SCHEDULE(STATIC,6)
26 iter on 4 threads

- DYNAMIC



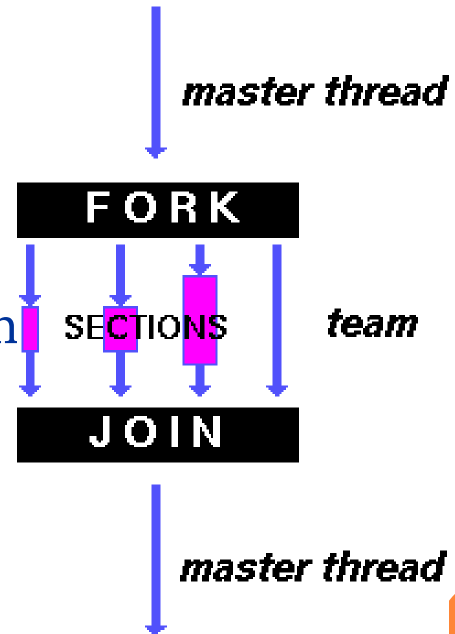
SCHEDULE(DYNAMIC,6)
26 iter on 4 threads

- To set at runtime – determined by the environment variable OMP_SCHEDULE

SECTION DIRECTIVE

- It specifies that the enclosed section(s) of code are to be divided among the threads in the team.
- Each SECTION is executed only once by a thread in the team.
- Syntax:

```
#pragma omp sections [clause1   clause2
{
        #pragma omp  section   <newlin
            structured_block
  #pragma omp  section   <newline>
            structured_block
 }
```



- Key points
  - Implicit barrier at the end of sections directive, if nowait clause is present.
  - If "too many" sections, some threads execute more than one section (round-robin).
  - If "too few" sections, some threads are idle.
  - We don't know in advance which thread will execute which section.

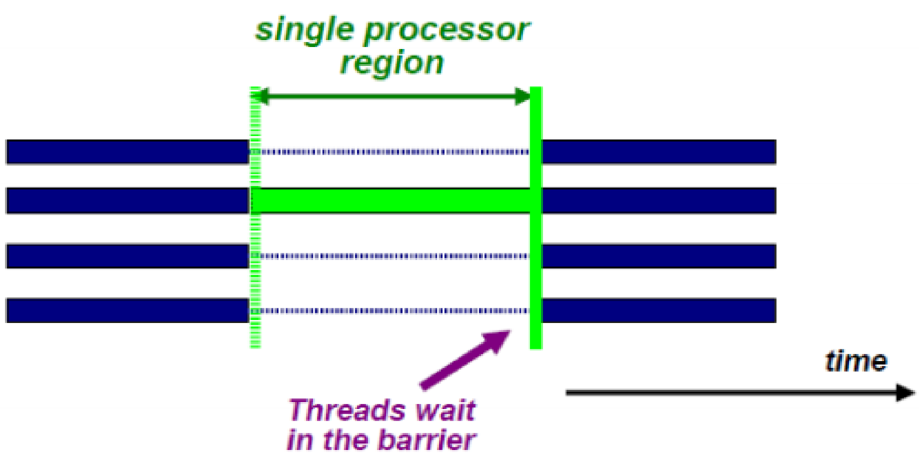
```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
       for (i=0; i<n-1; i++)
           b[i] = (a[i] + a[i+1])/2;

      #pragma omp section
       for (i=0; i<n; i++)
           d[i] = 1.0/c[i];

    } /*-- End of sections --*/

  } /*-- End of parallel region --*/
```

No need to wait to enter Or to exit from a section

SINGLE DIRECTIVE

- The enclosed code is to be executed by only one thread in the team.
- A barrier is implicitly set at the end of the single block (the barrier can be removed by the nowait clause)
- Syntax:

  #pragma omp single [clause1 clause2 …]
  structured_block



Synchronization

- Consider a case of 2 threads, both trying to increment a variable "x" at the same time (assume x = 0 initially)

- Problem: One possible execution sequence:
  - Thread 1 loads the value of x into register A.
  - Thread 2 loads the value of x into register A.
  - Thread 1 adds 1 to register A → x = x + 1 by thread1
  - Thread 2 adds 1 to register A → x = x + 1 by thread2
  - Thread 1 stores register A at location x
  - Thread 2 stores register A at location x
  - The resultant value of x will be 2, not 1 as it should be.

Synchronization

- Solution: To avoid a situation like this, the incrementing of x must be synchronized between the two threads to ensure that the correct result is produced.

- OpenMP Synchronization Constructs:
  - MASTER Directive
  - CRITICAL Directive
  - BARRIER Directive
  - ATOMIC Directive

MASTER

- To execute a region only by master thread of the team.
- All other threads on the team skip this section of code
- There is no implied barrier associated with this directive
- Syntax:

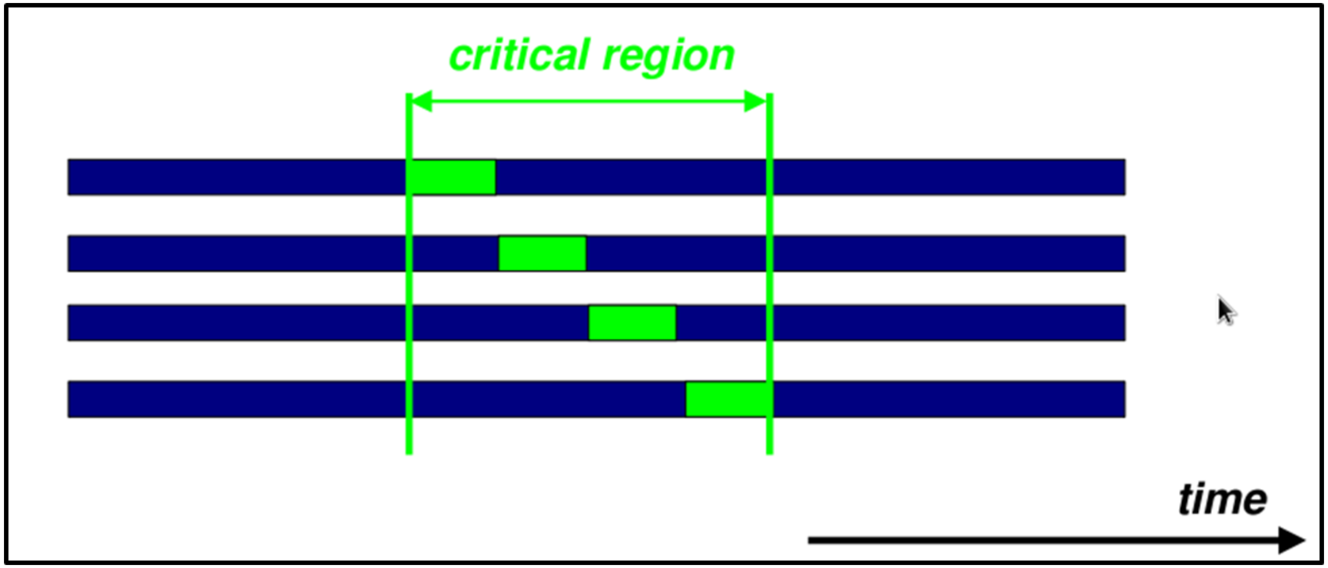
```
#pragma omp master newline
structured_block
```

CRITICAL

- The CRITICAL directive specifies a region of code that must be executed by only one thread at a time.
- Syntax:

  #pragma omp critical [name] newline
  structured_block

The optional name enables multiple different CRITICAL regions to exist.



www.cdac.in

CRITICAL

- If a thread is currently executing inside a CRITICAL region and another thread reaches that CRITICAL region and attempts to execute it, it will block until the first thread exits that CRITICAL region.
- The optional name enables multiple different CRITICAL regions to exist:
  - Names act as global identifiers. Different CRITICAL regions with the same name are treated as the same region.
  - All CRITICAL sections which are unnamed, are treated as the same section.

BARRIER

- On reaching BARRIER directive, a thread will wait at that point until all other threads have reached that barrier.
- All threads then resume executing in parallel the code that follows the barrier. Synchronizes all threads in a team.
- Syntax:

  #pragma omp barrier newline

```
#pragma omp parallel for
for (i=0; i < N; i++)

      a[i] = b[i] + c[i];
```

```
#pragma omp barrier
```

```
#pragma omp parallel for
for (i=0; i < M; i++)
          d[i] = a[i] + b[i];
```

ATOMIC

- Specifies that a specific memory location must be updated atomically, by a single thread
- Atomicity: We cannot split an operation – Example, A write operation.

- The directive applies only to a single, immediately following statement → Applies to only a single statement, allows only a limited number of expressions.

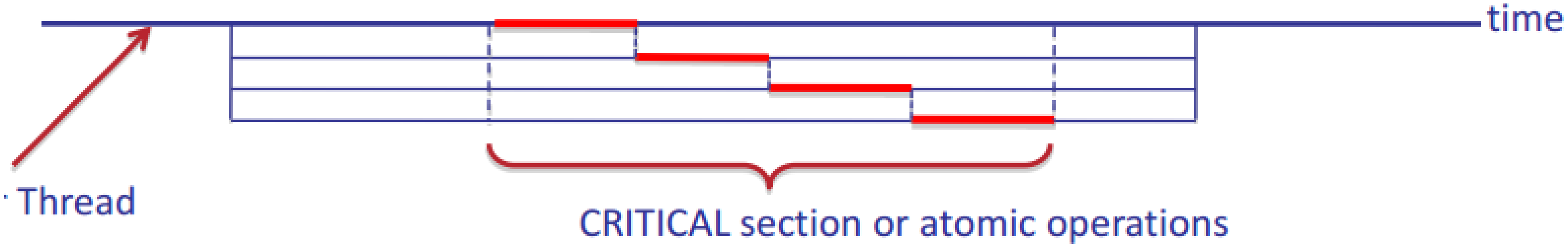
- Provides a mini critical section

- Syntax:

```
#pragma omp  atomic  newline
statement_expression
```

```
#pragma omp parallel shared(sum,x,y)
 ...
#pragma omp critical
    {
      update(x);
      update(y);
      sum=sum+1;
    }
...
!$OMP END PARALLEL
```

```
#pragma omp parallel shared(sum)
...
    {
#pragma omp atomic
       sum=sum+1;
       ...
    }
```



- This Directive is very similar to the CRITICAL directive.
- Difference is that ATOMIC is only used for the update of a memory location.
- Sometimes ATOMIC is also referred to as a mini critical section.

Data Scope Attributes

The OpenMP Data Scope Attribute Clauses are used to explicitly define how variables should be scoped. They include:

- IF
- PRIVATE
- FIRSTPRIVATE
- SHARED

Data Scope Attribute Clauses are used in conjunction with several directives (PARALLEL, DO/for, and SECTIONS) to control the scoping of enclosed variables

IF

- A clause to decide at run time if a parallel region should actually be executed in parallel (multiple threads) or just by the master thread:

- Syntax:
  - If(logical expr)

- Example:
  - #pragma omp parallel if (n>100000)

PRIVATE

- This declares variables in its list to be private to each thread
- Syntax:
  - private (list)

```
Eg:     int B = 10;
        #pragma omp parallel private(B)
        B=…;
```

- A un-initialised copy of B is created before the parallel region begins.

FirstPrivate

- Firstprivate: A private initialized copy of B is created on each thread's stack before the parallel region begins

  - Syntax:

    firstprivate (list)
  - Example:

```
int B = 10;
#pragma omp parallel firstprivate(B)
B = 10;
```

SHARED

- SHARED Clause
  - A shared variable exists in only one memory location and all threads can read or write to that address
  - Syntax:

    shared (list)

- It is the programmer's responsibility to ensure that multiple threads properly access SHARED variables (such as via CRITICAL sections)

2. Runtime Libraries

- Execution environment routines that can be used to control and to query the parallel execution environment

Example routines:

- OMP_SET_NUM_THREADS
  - omp_set_num_threads routine affects the number of threads to be used for subsequent parallel regions
    - C/C++ : void omp_set_num_threads(int num_threads);
- OMP_GET_NUM_THREADS
  - returns the number of threads in the current team.
    - C/C++ : int omp_get_num_threads(void);

- OMP_GET_THREAD_NUM
  - Returns the thread ID of the thread

```
#include <omp.h>
int omp_get_thread_num()
```

- OMP_GET_NUM_PROCS
  - To get the number of processors

```
#include <omp.h>
int omp_get_num_procs()
```

- OMP_IN_PARALLEL
  - Determine if the section of code which is executing is parallel or not.

```
#include <omp.h>
int omp_in_parallel()
```

3. Environment Variables

- OMP_SCHEDULE
  - export OMP_SCHEDULE “static"
  - export OMP_SCHEDULE "dynamic"

- OMP_NUM_THREADS
  - export OMP_NUM_THREADS=8


Thank You