OpenMP

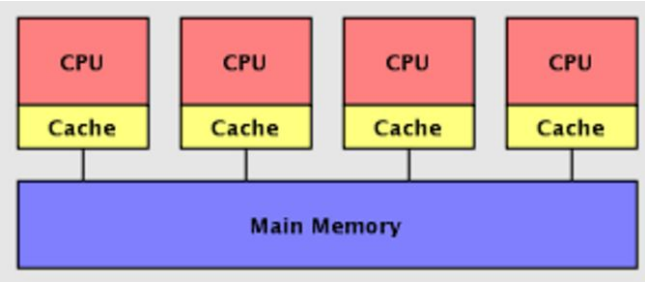
C-DAC Pune

Contents

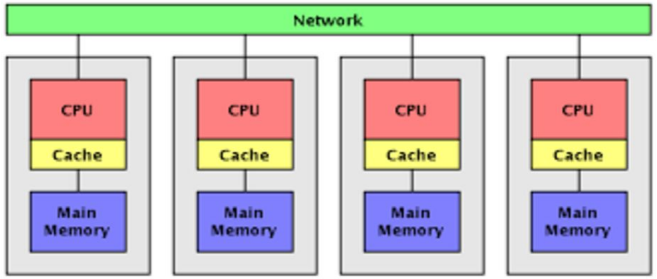
- Basic Architecture
- Sequential Program Execution
- Introduction to OpenMP
- OpenMP Programming Model
- OpenMP Syntax and Execution
- OpenMP Directives and Clauses
- Race Condition
- Environment variables
- OpenMP examples
- Good things and Limitation of OpenMP
- References

Parallel programing Architectures/Model ..

- Shared-memory Model



- Distributed-memory Model



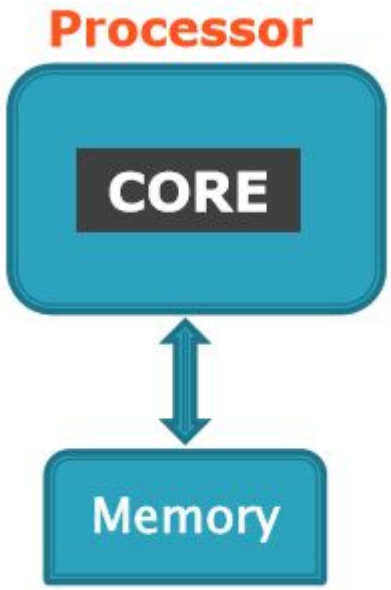
- UMA - Uniform Memory Access
- NUMA - Non-Uniform Memory Access

- **openMP**
- Pthreads ...

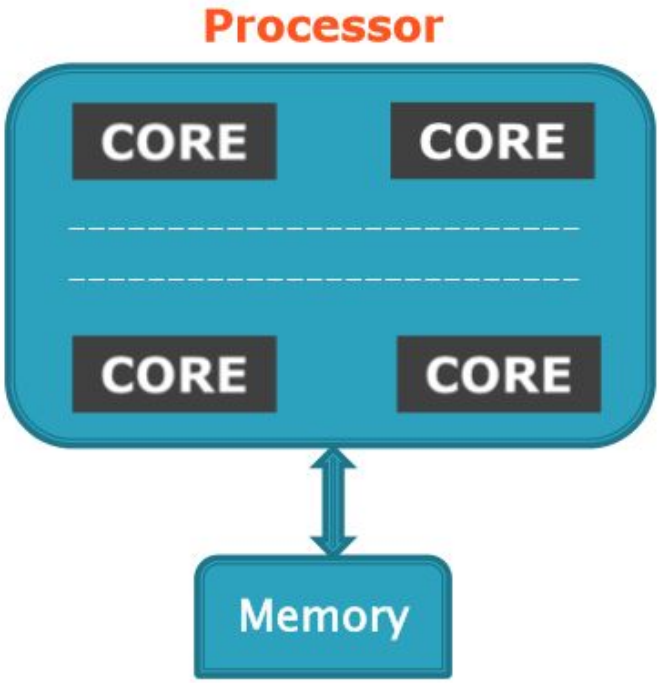
- MPI - Message Passing Interface

- Hybrid model

Basic Architecture



Older processor had only single CPU core to execute instructions



Processors have 4 or more independent CPU cores to execute instructions

Sequential Program Execution

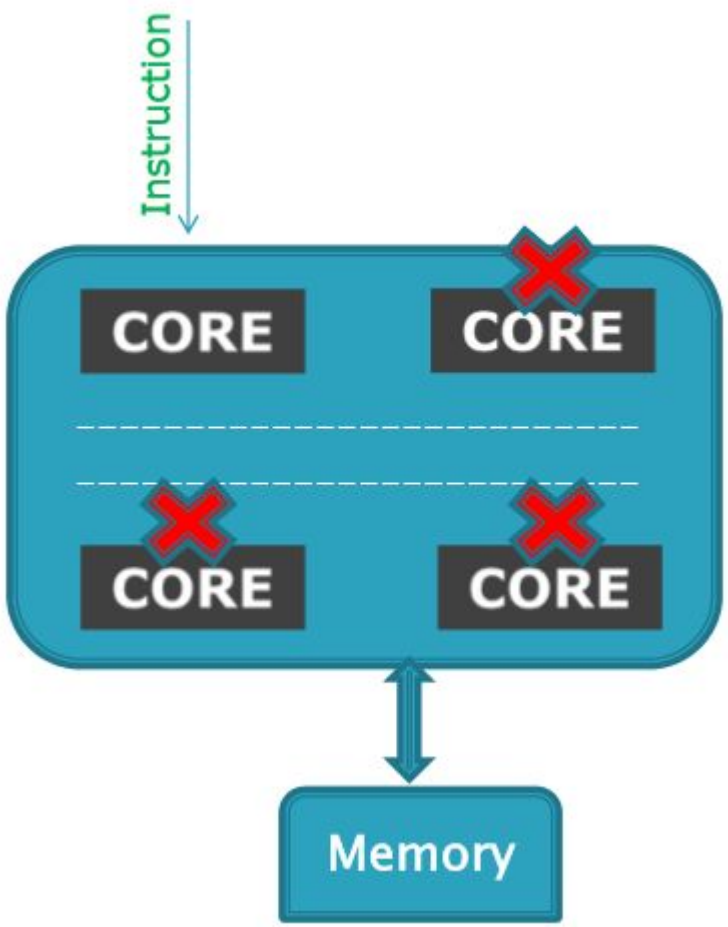
When you run sequential program

- Instructions executed in serial
- Other cores are idle

Waste of available resource...

We want all cores to be used to execute program.

HOW ?



OpenMP Introduction

- Open Specification for Multi Processing.
- It is an specification for

- OpenMP is an Application Program Interface (API) for writing multi-threaded, shared memory parallelism.
- Easy to create multi-threaded programs in C,C++ and Fortran.

OpenMP Versions

- Fortran 1.0 was released in October 1997
  - C/C++ 1.0 was approved in November 1998
  - version 2.5 joint specification in May 2005
  - OpenMP 3.0 API released May 2008
  - Version 4.0 released in July 2013
  - Version 4.5 released in Nov 2015

Why Choose OpenMP ?

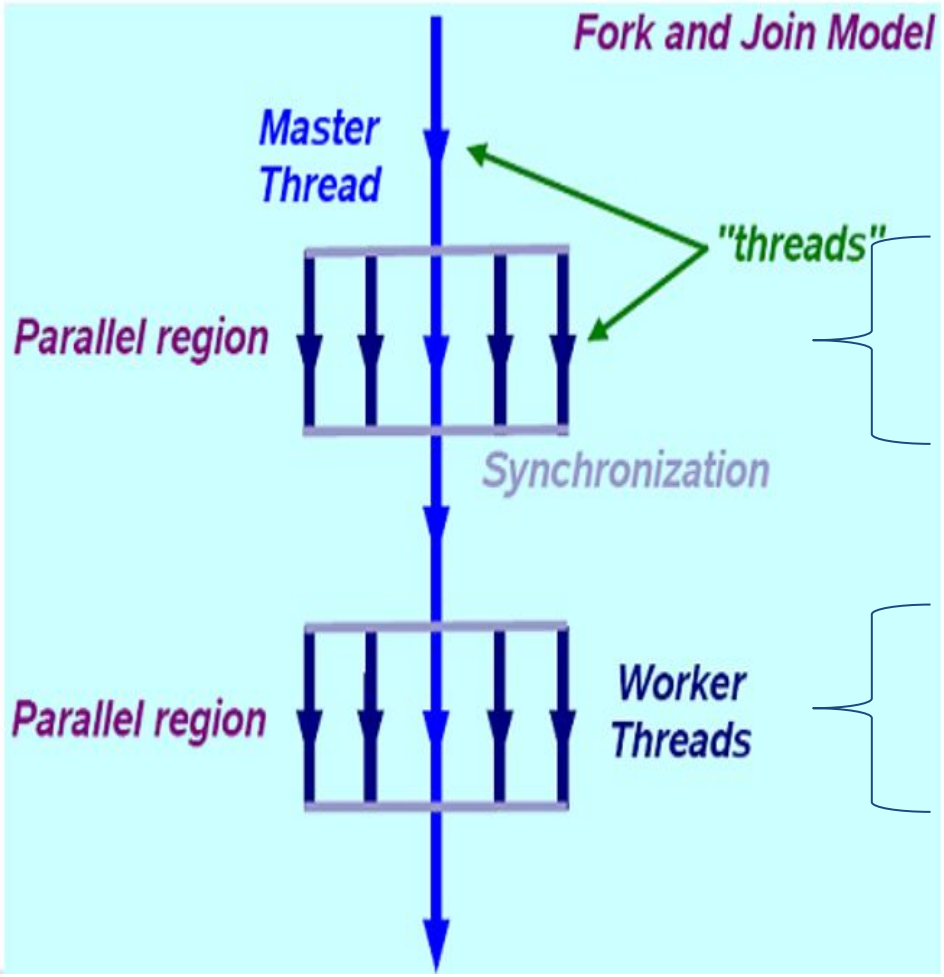
- **Portable**
  - standardized for shared memory architectures
- **Simple and Quick**
  - Relatively easy to do parallelization for small parts of an application at a time
  - Incremental parallelization
  - Supports both fine grained and coarse grained parallelism
- **Compact API**
  - Simple and limited set of directives
  - Not automatic parallelization

Execution Model

- OpenMP program starts single threaded
- To create additional threads, user starts a parallel region
  - additional threads are launched to create a team
  - original (master) thread is part of the team
  - threads “go away” at the end of the parallel region: usually sleep or spin
- Repeat parallel regions as necessary

Fork-join model

OpenMP Programming Model




```
main (..)
{

#pragma omp parallel
{
      ……
      …...
}

#pragma omp parallel
{
      ……
      …...
}

}
```

Communication Between Threads

- Shared Memory Model
  - threads read and write shared variable no need for explicit message passing
  - use synchronization to protect against race conditions
  - change storage attributes for minimizing synchronization and improving cache reuse

When to use openMP ?

- Target platform is multi core or multiprocessor.
- Application is cross-platform
- Parallelizable loop
- Last-minute optimization

OpenMP Basic Syntax

- Header file

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

- Parallel region:

```
#pragma omp construct_name [clause, clause...]
{
// .. Do some work here
}
 // end of parallel region/block
```

Environment variables and functions

- In Fortran:

```
!$OMP construct [clause [clause] .. ]
C$OMP construct [clause [clause] ..]
*$OMP construct [clause [clause] ..]
```

Executing OpenMP Program

Compilation

gcc –fopenmp <program name> –o <execcutable>

gfortran –fopenmp <program name> –o <execcutable>

ifort <program name> -qopenmp –o <execcutable>

icc <program name> -qopenmp –o <execcutable>

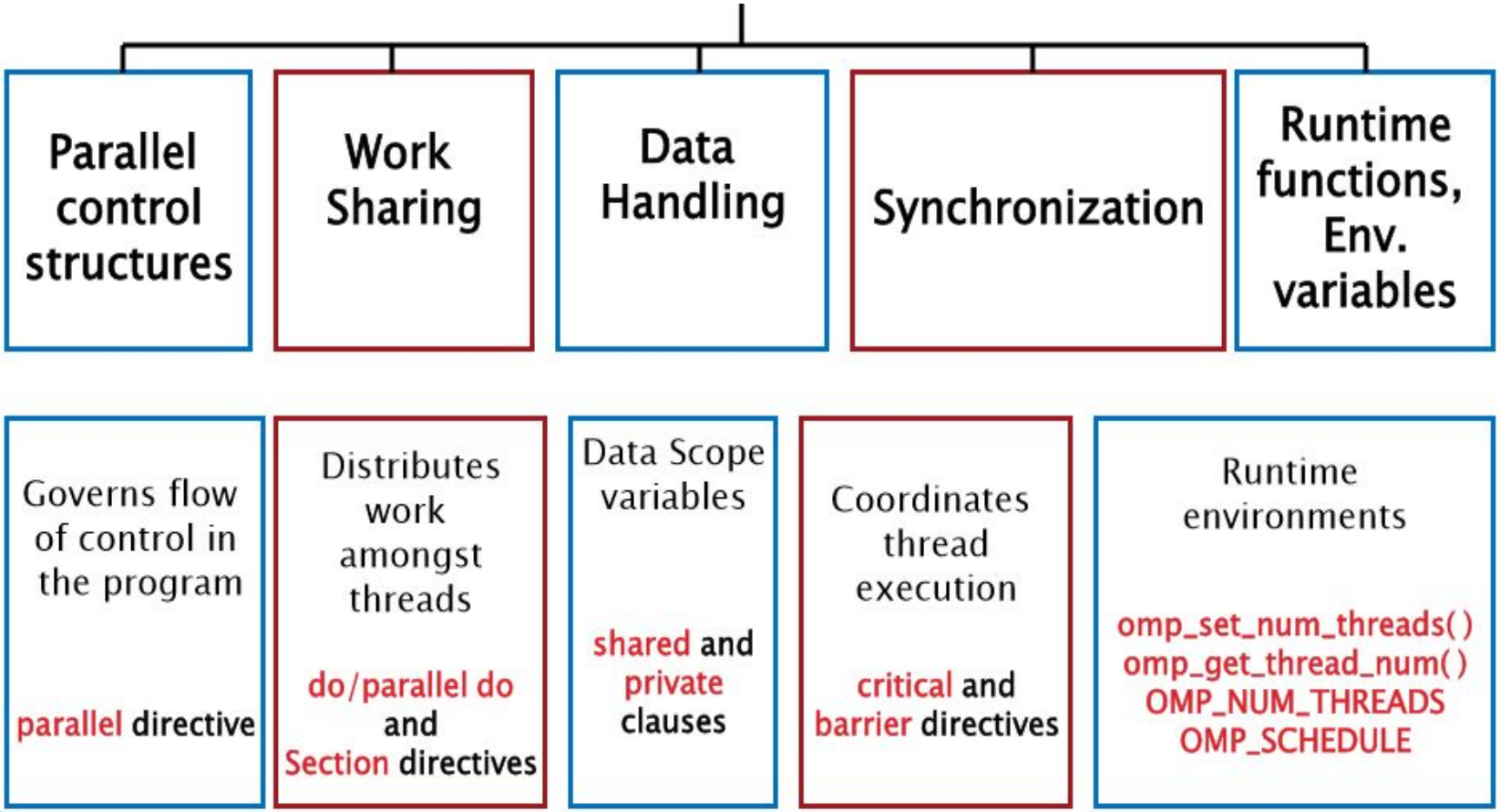
Execution:

./ <executable-name>

Compiler support for OpenMP

Compiler	Flag
GNU (gcc/g++)	-fopenmp
Intel (icc)	-qopenmp
Sun	-xopenmp
Many more….	

OpenMP Language extensions



OpenMP Constructs

- **Parallel Regions**

  #pragma omp parallel

- **Worksharing**

  #pragma omp for, #pragma omp sections

- **Data Environment**

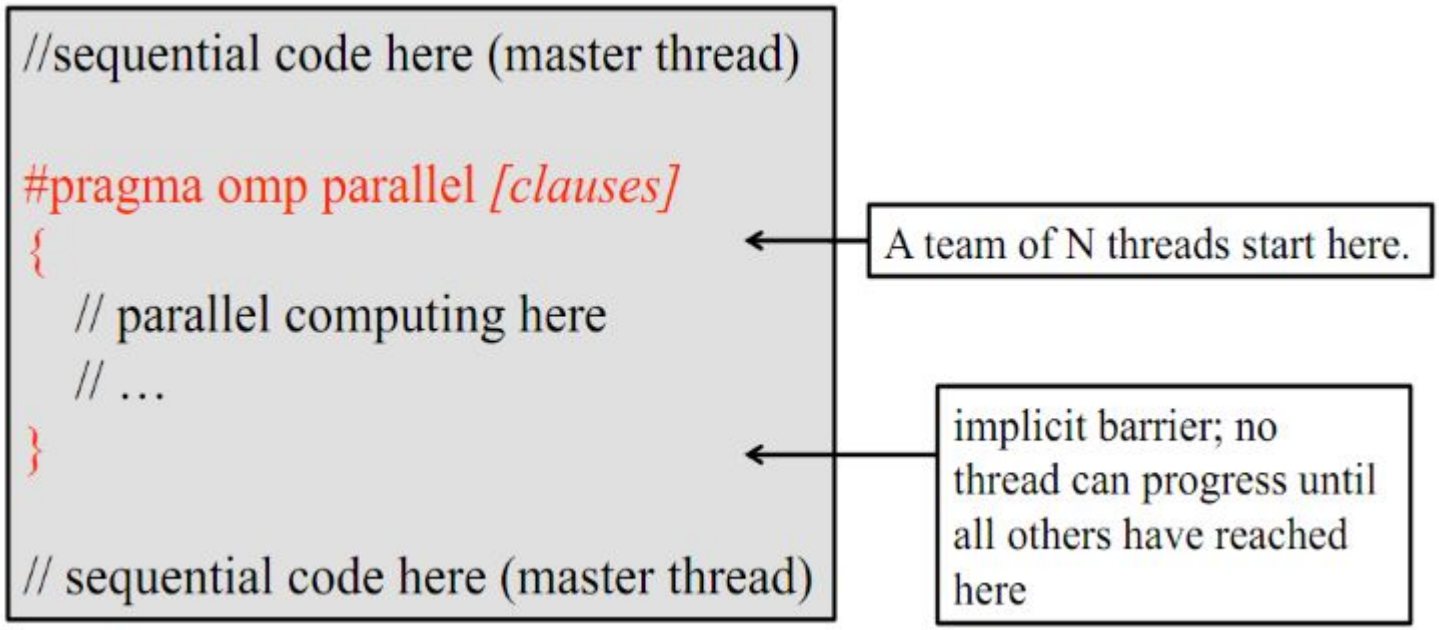
  #pragma omp parallel shared/private (...)

- **Synchronization**

  #pragma omp barrier, #pragma omp critical

Parallel Region

- **Fork a team of N threads {0.... N-1}**
- **Without it, all codes are sequential**



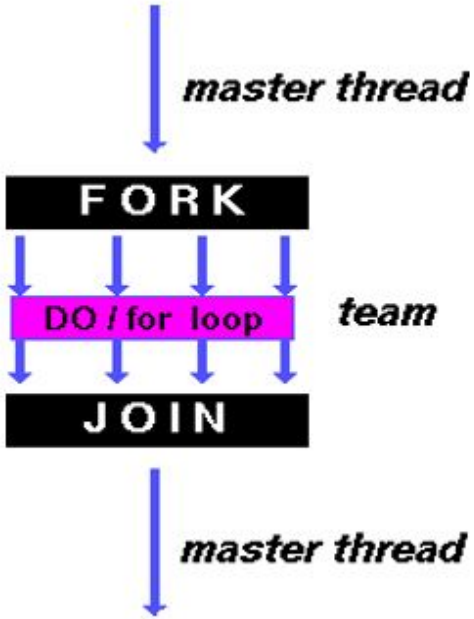
Loop Constructs: Parallel do/for

In C:

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

In Fortran:

```
!$omp parallel do
Do i=1, n
a(i) = b(i) +c(i)
enddo
```



Clauses for Parallel constructs

```
#pragma omp parallel [clause, clause, ...]
```

- **shared**
- **private**
- **firstprivate**
- **lastprivate**
- **nowait**
- **if**
- **reduction**
- **Schedule**
- **default**

private Clause

- The values of private data are undefined upon entry to and exit from the specific construct.

- Loop iteration variable is private by default.

Example:

```
#prgma omp parallel for private(a)
  for(i=0; i<n; i++)
  {
        a=i+1;
        printf("Thread %d has value of a=%d for i=%d\n",
        omp_get_thread_num(),a,i)
  }
```

firstprivate Clause

- The clause combines behavior of private clause with automatic initialization of the variables in its list.

Example:

```
b=50,n=1858;
printf("Before parallel loop: b=%d ,n=%d\n",b,n)
#pragma omp parallel for private(i), firstprivate(b), lastprivate(a)
for(i=0; i<n; i++)
      {
            a=i+b;
      }
c=a+b;
```

lastprivate Clause

- Performs finalization of private variables.
- Each thread has its own copy.

Example:

```
b=50,n=1858;
printf("Before parallel loop: b=%d ,n=%d\n",b,n)
#pragma omp parallel for private(i), firstprivate(b), lastprivate(a)
for(i=0; i<n; i++)
    {
        a=i+b;
    }
c=a+b;
```

shared Clause

- Shared among team of threads.
- Each thread can modify shared variables.
- Data corruption is possible when multiple threads attempt to update the same memory location.
- Data correctness is user's responsibility.

default Clause

- Defines the default data scope within parallel region.

- default(private | shared | none).

nowait Clause

```
#pragma omp parallel nowait
```

- Allow threads that finish earlier to proceed without waiting.
- If specified, then threads do not synchronize at the end of parallel loop.

if clause

```
#pragma omp parallel
{
// ...some stuff
}
```

- **If (integer expression)**
  - Determines if the region should be parallelized.
  - Useful option when data is too small (or too large)

reduction Clause

- Performs a reduction on variables subject to given operator.

```
#pragma omp parallel for reduction(+ : result)

for (unsigned i = 0; i < v.size(); i++)
{
result += v[i];
...
```

schedule Clause

```
#pragma omp parallel for schedule (type,[chunk size])
{
// ...some stuff
}
```

- Specifies how loop iteration are divided among team of threads.

- Supported Scheduling Classes:
  - Static
  - Dynamic
  - Guided
  - runtime

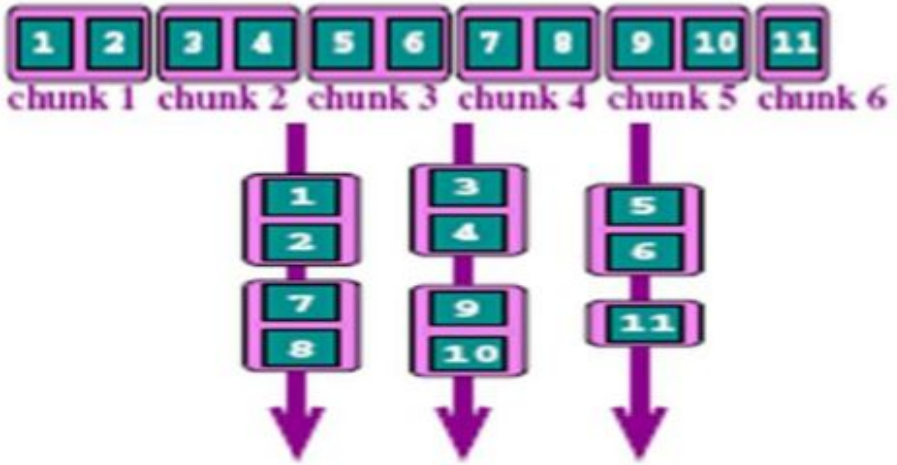
schedule Clause.. cont

schedule(static, [n])

- Each thread is assigned chunks in “round robin” fashion, known as block cyclic scheduling.

- If n has not been specified, it will contain CEILING(number_of_iterations / number_of_threads) iterations.

  - Example: loop of length 16, 3 threads, chunk size 2:



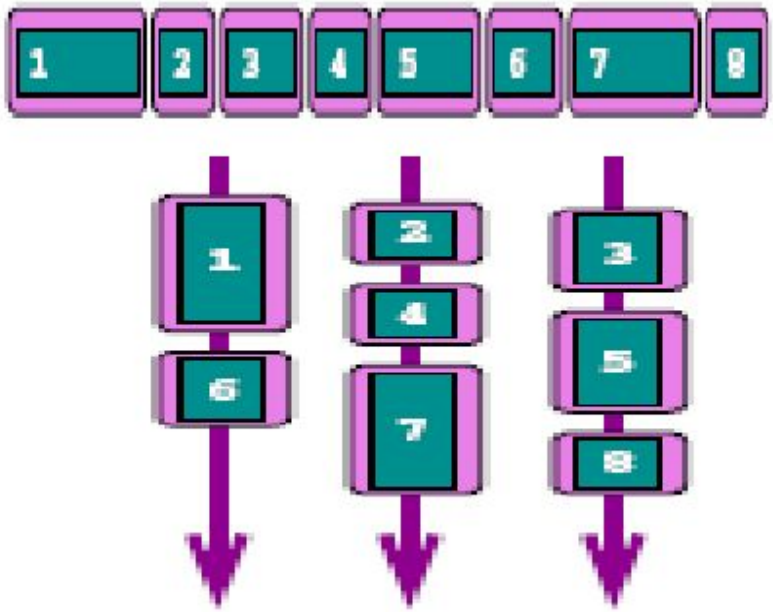
schedule Clause.. cont

schedule(dynamic, [n])

- Iteration of loop are divided into chunks containing n iterations each.
- Default chunk size is 1.

```
!$omp do schedule (dynamic)

 do i=1,8
    … (loop body)
end do
!$omp end do
```



schedule Clause.. cont

schedule(guided, [n])

- If you specify n, that is the minimum chunk size that each thread should posses.
- Size of each successive chunks is exponentially decreasing.
- initial chunk size

max((num_of_iterations/num_of_threads),n)

Subsequent chunks consist of

max(remaining_iterations/number_of_threads,n) iterations.

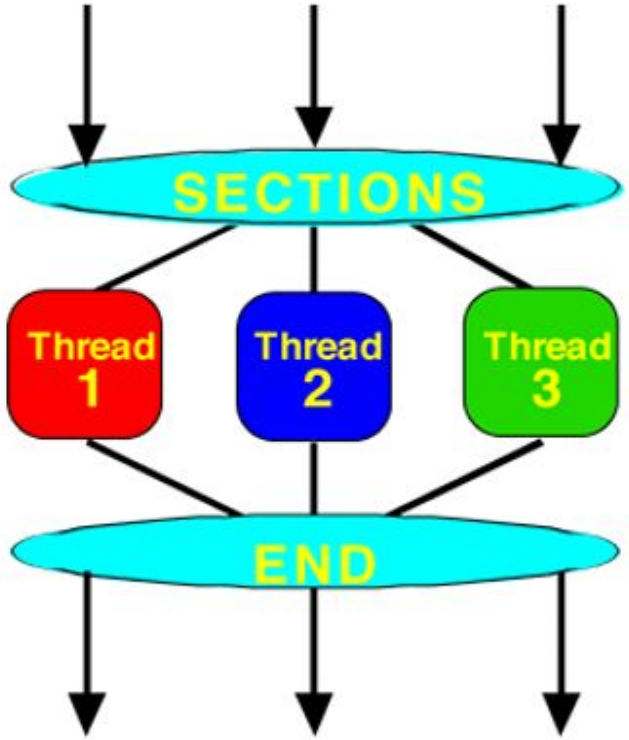
Schedule(runtime): export OMP_SCHEDULE “STATIC, 4”

- Determine the scheduling type at run time by the OMP_SCHEDULE environment variable.

Work sharing : Section Directive

- Each section is executed exactly once and one thread executes one section

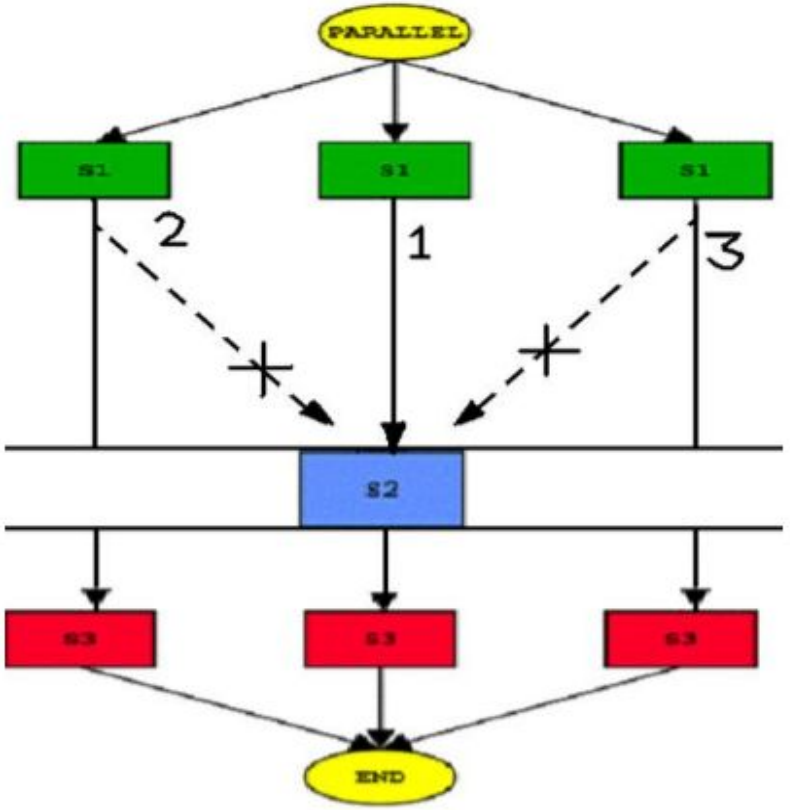
```
#pragma omp parallel
  #pragma omp sections
   {
    #pragma omp section
       x_calculation();
    #pragma omp section
      y_calculation();
    #pragma omp section
     z_calculation();
}
```



Work sharing : Single Directive

Designated section is executed by single thread only.

```
#pragma omp single
 {
      a =10;
 }
#pragma omp for
 {
  for (i=0;i<N;i++)
         b[i] = a;
}
```



Work sharing : Master

- Similar to single, but code block will be executed by the master thread only.

```
#pragma omp master
 {
      a =10;
 }
#pragma omp for
 {
      for (i=0;i<N;i++)
            b[i] = a;
 }
```

C/C++:

```
#pragma omp master

     ----- block of code--
```

Race condition

Finding the largest element in a list of numbers.

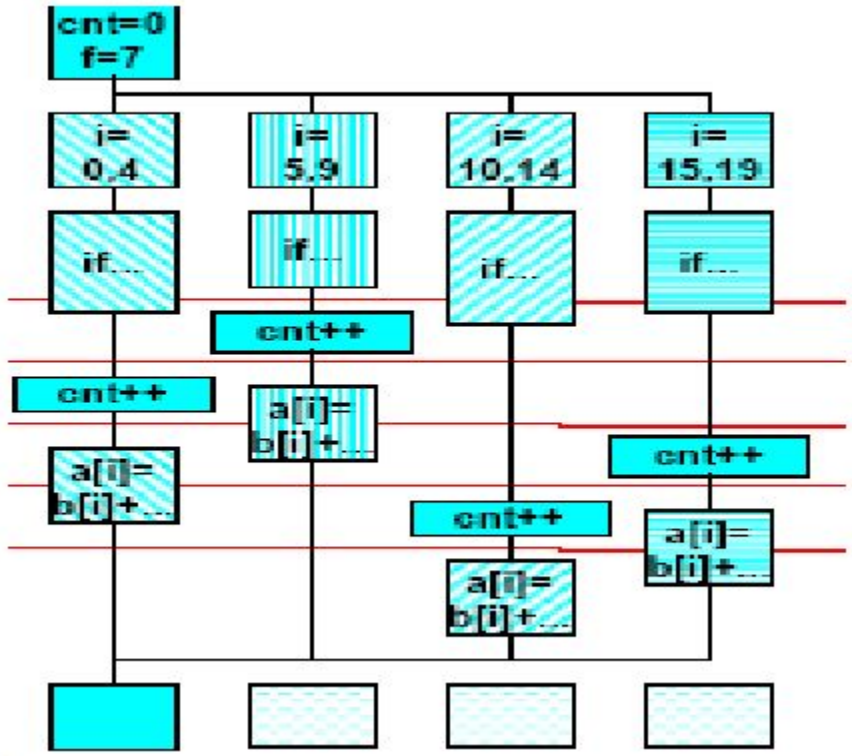
```
Max = 10
 !$omp parallel do
       do i=1,n
             if(a(i) .gt. Max) then
                   Max = a(i)
             endif
        enddo
```

Thread 0
Read a(i) value = 12
Read Max value = 10
If (a(i) > Max) (12 > 10)
Max = a(i) (i.e. 12)

Thread 1
Read a(i) value = 11
Read Max value = 10
If (a(i) > Max) (11 > 10)
Max = a(i) (i.e. 11

Synchronization: Critical Section

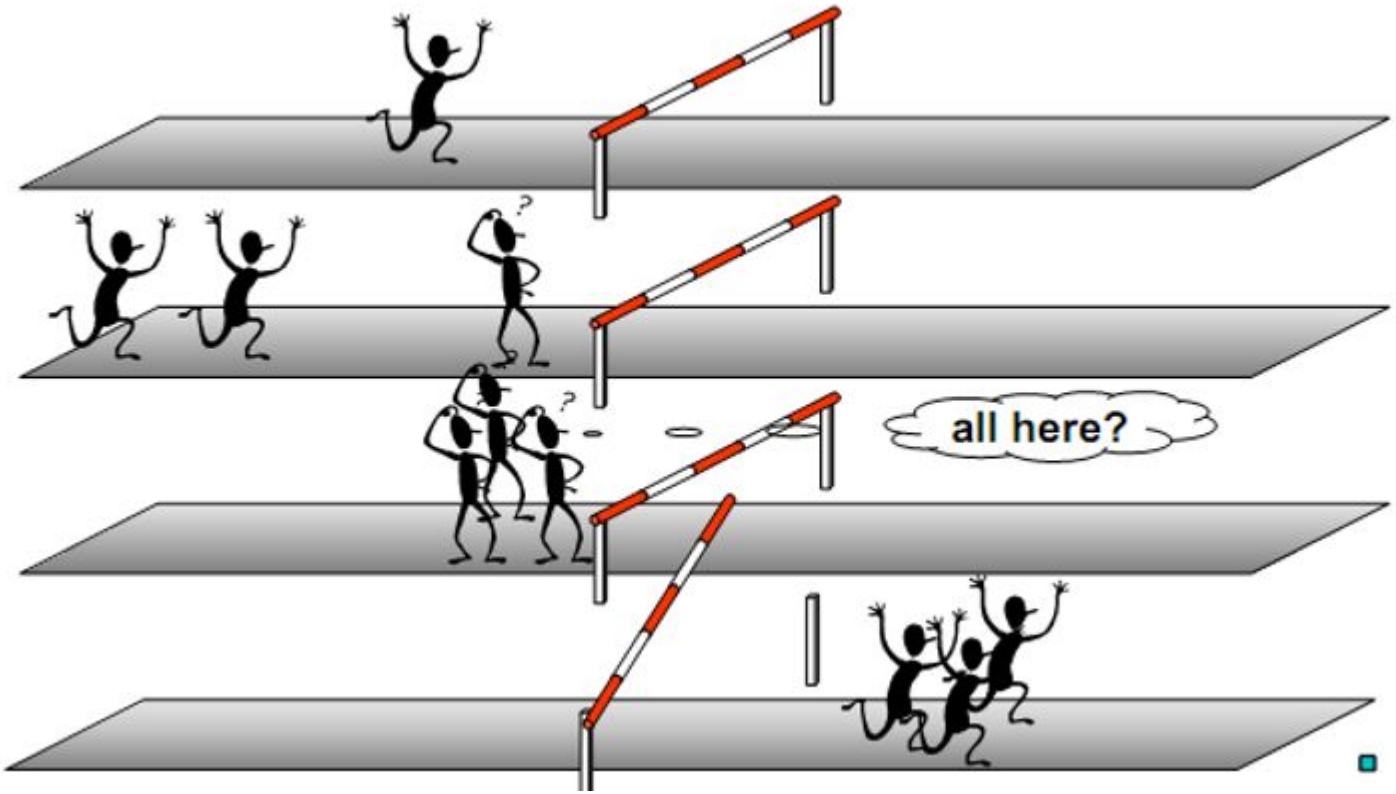
Directive restrict access to the enclosed code to only one thread at a time.




```
cnt = 0, f=7;
 #pragma omp parallel
       {
        #pragma omp for
              for(i=0;i<20;i++)
                    {
                          if(b[i] == 0)
                          {
                          #pragma omp
                    critical
                          cnt++;
                          }
                          a[i] = b[i] + f*(i+1);
                     }
       }
```

Synchronization:Barrier Directive

Synchronizes all the threads in a team.



Synchronization:Barrier Directive

```
int x=2;
 #pragma omp parallel shared(x)
 {
    int tid = omp_get_thread_num();
   if(tid == 0)
      x=5;
   else
         printf("[1] thread %2d:
               x=%d\n",tid,x);
          #pragma omp barrier
     printf("[2] thread %2d:
            x=%d\n",tid,x);
}
```

Some threads may still have x=2 here

Cache flush + thread synchronization

All threads have x=5 here

Synchronization:Atomic Directive

- Mini Critical section.
- Specific memory location must be updated atomically.

C:

```
#pragma omp atomic
----- Single line code--
```

Task Construct

- Tasks in OpenMP are code blocks that the compiler wraps up and makes available to be executed in parallel.

```
int fib(int n)
{
    int i, j;
  if (n<2) return n;
  else
   {
     #pragma omp task shared(i) firstprivate(n)
        i=fib(n-1);
        #pragma omp task shared(j) firstprivate(n)
        j=fib(n-2);
        #pragma omp taskwait
        return i+j;
    }
}
```

Task Construct

1. **Taskgroup Construct**
- #pragma omp taskgroup

2.**Taskloop Construct**
- #pragma omp taskloop num_tasks (x)

3.**Taskyield Construct**
- #pragma omp taskyield

4.**Taskwait Construct**
- #pragma omp taskwait

5. **Task dependency**
- #pragma omp task depend(in: x)

6.**Task priority**
- #pragma omp task priority(1)

Taskloop Construct

OpenMP 4.0

```
#pragma omp taskgroup
{
  for (int tmp = 0; tmp < 32;
tmp++)
    #pragma omp task
      for (long l = tmp * 32; l
< tmp * 32 + 32; l++)
        do_something (l);
}
```

OpenMP 4.5

```
#pragma omp taskloop
num_tasks (32)
  for (long l = 0; l <
1024; l++)
    do_something (l);
```

OpenMP 4.5 Features

- Explicit map clause
- Target Construct
- Target data Construct
- Target Update Construct
- Declare target
- Teams construct
- Device run time routines
- Target Memory and device pointer routines
- Thread Affinity
- Linear clause in loop construct
- Asynchronous target execution with nowait clause
- Doacross loop construct
- SIMD construct

Environment Variable

- To set number of threads during execution

  export OMP_NUM_THREADS=4

- To allow run time system to determine the number of threads

  export OMP_DYNAMIC=TRUE

- To allow nesting of parallel region

  export OMP_NESTED=TRUE

- Get thread ID

  omp_get_thread_num()

Control the Number of Threads

- Parallel region

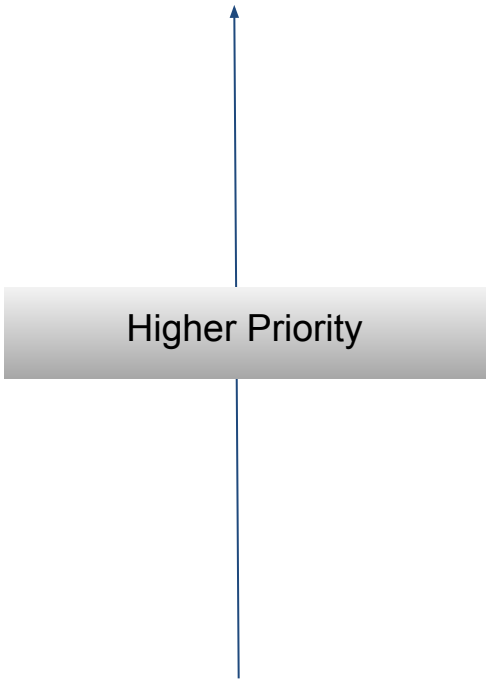
  #pragma omp parallel num_threads(integer)

- Run-time function/ICV

  omp_set_num_threads(integer)

- Environment Variable

  OMP_NUM_THREADS



OPENMP 3.1 TO 4+ FEATURES

Openmp 3.0	Openmp 4.0 to 4.0.2	Openmp 4.0.2 to 4.5
Parallel construct Worksharing construct Synronization construct Data sharing clauses Task Construct	Proc_bind clause Task group construct Task dependences Target construct Target update construct Declare target construct Teams construct Device runtime routines Distribute construct	Task Priority Task loop construct Device memory routines and device pointers Do across loop construct Linear clause in loop construct Cancellation Construct

Good Things about OpenMP

- Incrementally Parallelization of sequential code.
- Leave thread management to compiler.
- Directly supported by compiler.

Limitations

- Requires compiler which supports OpenMP
- Internal details are hidden
- Run efficiently in shared-memory multiprocessor platforms only but not on distributed memory
- Limited performance by memory architecture

References

https://computing.llnl.gov/tutorials/openMP/

http://wiki.scinethpc.ca/wiki/images/9/9b/D s-openmp.pdf

www.openmp.org/

http://openmp.org/sc13/OpenMP4.0_Intro_Y onghongYan_SC13.pdf

Thank You