Introduction to OpenMP

Sandeep Agrawal
C-DAC Pune

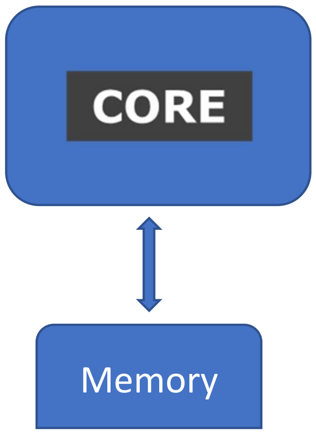
Parallelism

Source: https://en.wikipedia.org/wiki/Pit_stop

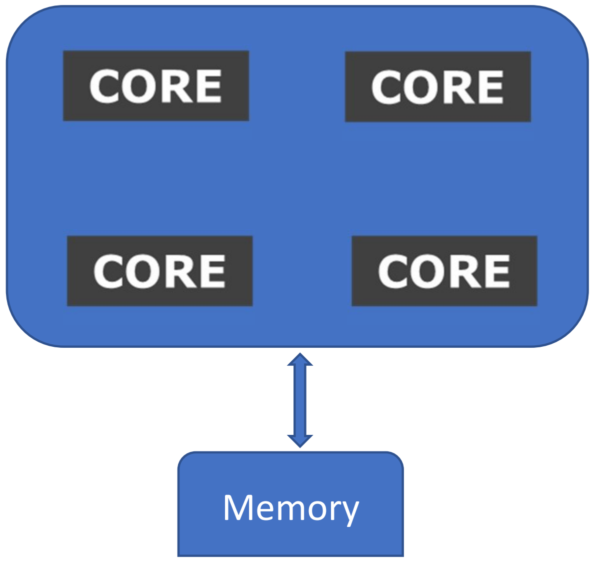
Contents

Basis System Architecture

Single Core Processor



Multi Core Processor



Sequential Program Execution

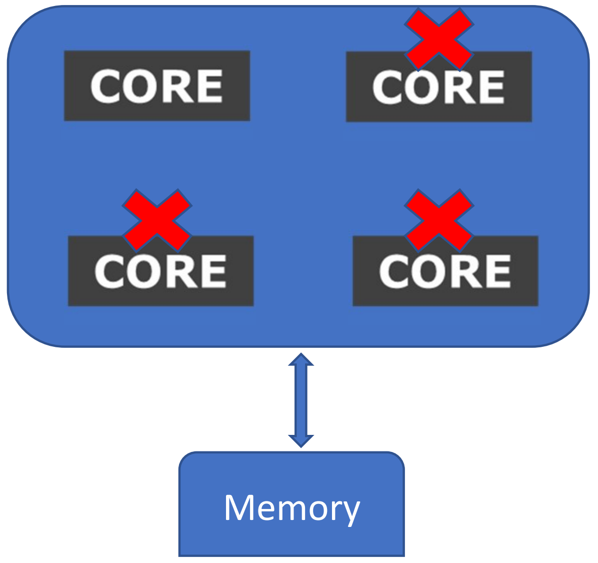
When you run sequential program

- Instructions executed in serial
- Other cores are idle

Waste of available resource… We want all cores to be used to execute program.

HOW ?

Multi Core Processor

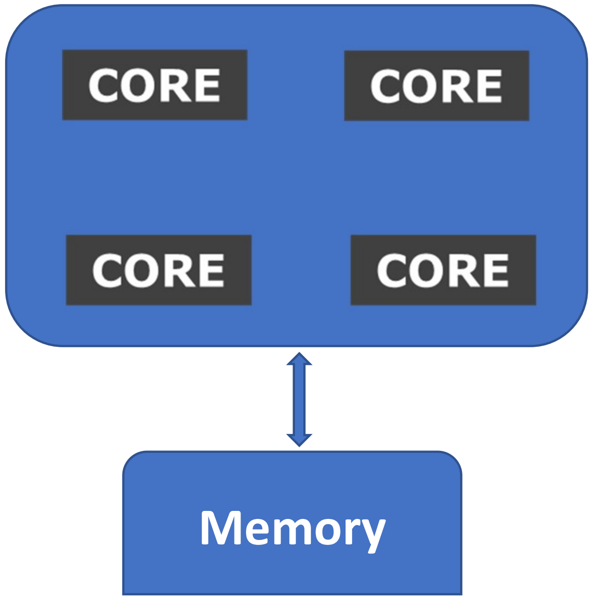


Process and Thread

- An executing instance of a program is called a process
- Process has its independent memory space
- A thread is a subset of the process – also called lightweight process allowing faster context switching
- Threads share memory space within process's memory
- Threads may have some (usually small) private data
- A thread is an independent instruction stream, thus allowing concurrent operation
- In OpenMP one usually wants no more than one thread per core

Shared Memory Model

- Multiple threads operate independently but share same memory resources
- Data is not explicitly allocated
- Changes in a memory location effected by one process is visible to all other processes
- Communication is implicit
- Synchronization is explicit



Open Multi-Processing (OpenMP)

OpenMP Introduction

- Open Specification for Multi Processing
- Provides multi-threaded parallelism
- It is an specification for
  - Directives
  - Runtime Library Routines
  - Environment Variables
- 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.

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

OpenMP Consortia and Release History

https://www.openmp.org/

OpenMP Architecture Review Board (ARB) members are from across academic, research, industrial organizations such as:

AMD, ARM, CRAY, IBM, Fujitsu, NEC, Intel, Red Hat …
ANL, LLNL, LBNL, ORNL, RWTH Aachen University, NASA ...

OpenMP Compilers for C/C++/Fortran:

GNU, Intel, PGI, LLVM/Clang, IBM, Absoft …

From GCC 4.9.1, OpenMP 4.0 is fully supported for C/C++/Fortran
From GCC 6.1, OpenMP 4.5 is fully supported for C and C++
From GCC 7.1, OpenMP 4.5 is partially supported for Fortran
From GCC 9.1, OpenMP 5.0 is partially supported for C and C++

Version	Year
Fortran 1.0	1997
C/C++ 1.0	1998
Fortran 1.1	1999
Fortran 2.0	2000
C/C++ 2.0	2002
OpenMP 2.5	2005
OpenMP 3.0	2008
OpenMP 3.1	2011
OpenMP 4.0	2013
OpenMP 4.5	2015
OpenMP 5.0	2018

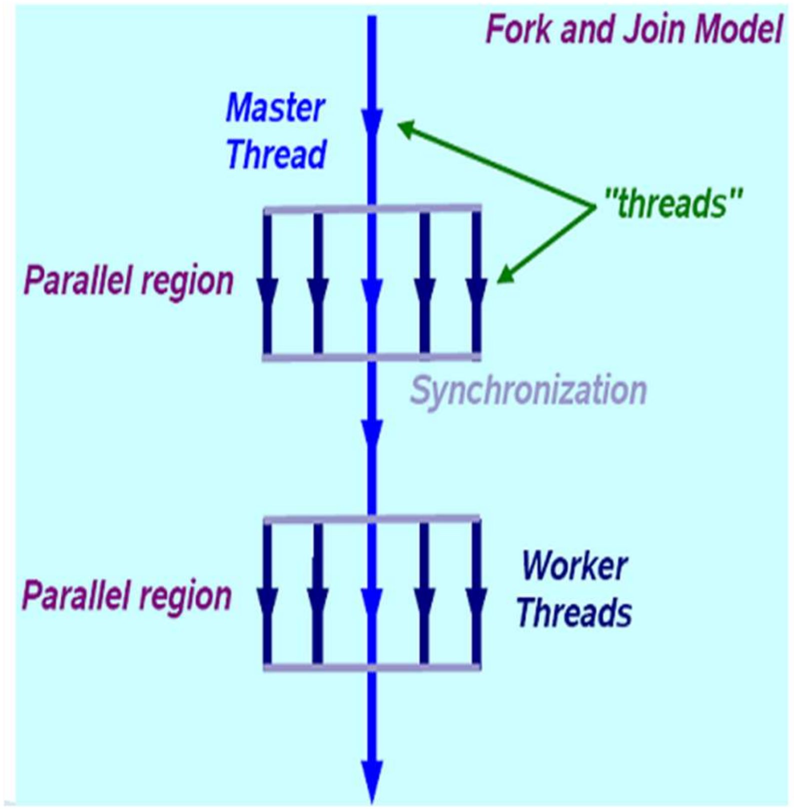
Execution Model

- 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

- Repeat parallel regions as necessary

  Fork-join model



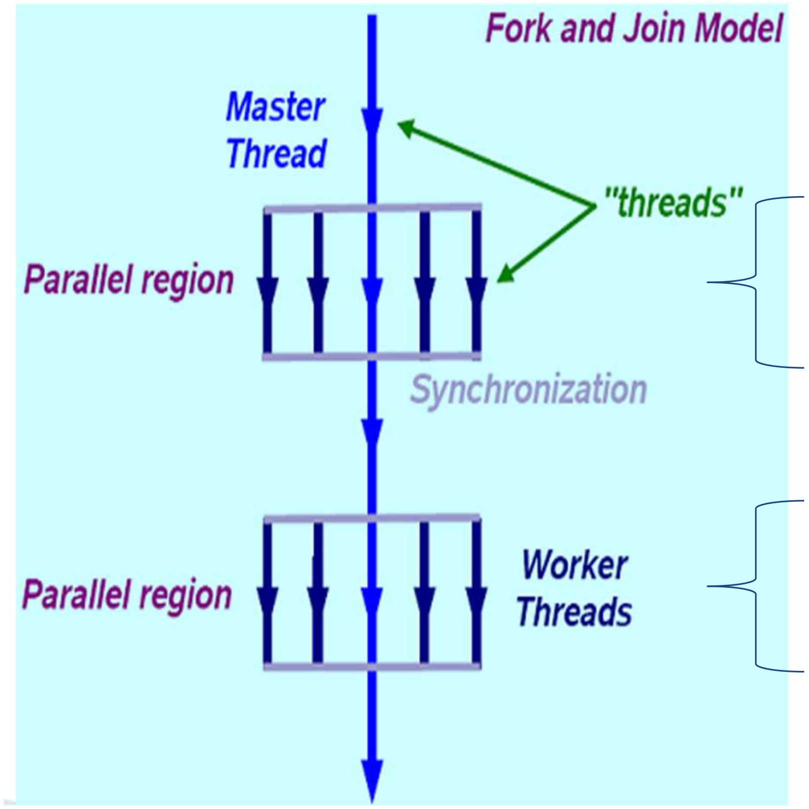
OpenMP Basic Syntax

Header file #include “omp.h”

Parallel region

C:

```
#pragma omp construct  [clauses...]
{
     // .. Do some work here

}   // end of parallel region/block
```

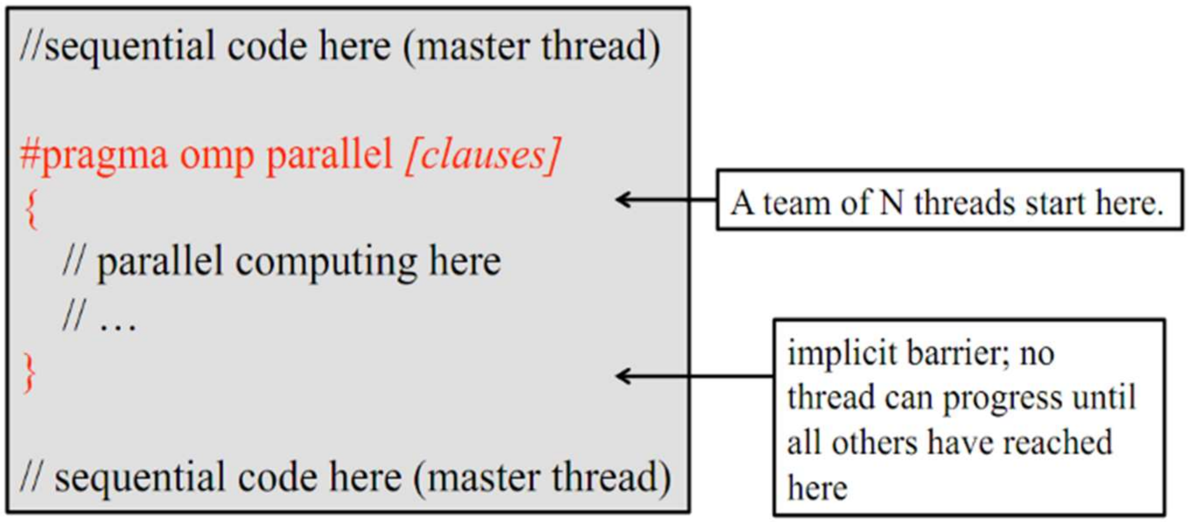


```
main (..)
{
    #pragma omp parallel
    {
        ……
        ……
    }

    #pragma omp parallel
    {
        ……
        ……
    }

}
```

Parallel Region

Fork a team of N threads {0.... N-1}

Without it, all codes are sequential



```
//sequential code here (master thread)

#pragma omp parallel [clauses]
{
    // parallel computing here
    // …
}

// sequential code here (master thread)
```

A team of N threads start here.
implicit barrier; no thread can progress until all others have reached here

Parallel Directive

- OpenMP directives are comments in source code that specify parallelism
- C/C++ compiler directives begin with the sentinel **#pragma omp**
- FORTRAN compiler directives begin with one of the sentinels **!$OMP**, **C$OMP**, or ***$OMP**
    - use **!$OMP**for free-format F90

Fortran

```
!$OMP parallel
       work ...
!$OMP end parallel

!$OMP parallel
        work ...
!$OMP end parallel
```

C/C++

```
# pragma omp parallel
 {
   work ...
  }

# pragma omp parallel
 {
       work...
  }
```

How do Threads Interact ?

- threads read and write shared variable
  - hence communication is implicit
- Unintended sharing of data causes race conditions
  - race condition can lead to different outputs across different runs
- use synchronization to protect against race conditions
- synchronization is expensive
  - change data storage attributes for minimizing synchronization and improving cache reuse

OpenMP Language Extensions

Parallel Control Structures	Work Sharing	Data Handling	Synchronization	Runtime functions, Environment Variables

Governs flow of control in the program
 	
`parallel`
`directive`
	Distribute works amongst threads
	
	`Do / for`
	`parallel do/for`
	`section`
	`directives`
		Data scope variables
		
		`shared`
		`private`
		`clauses`
			Coordinates thread execution
			
			`critical`
			`barrier`
			`directives`
				Runtime environments
				
				`omp_set_num_threads()`
				`omp_get_thread_num()`
				
				`OMP_NUM_THREADS`
				`OMP_SCHEDULE`

OpenMP Constructs

- Parallel region

  #pragma omp parallel

- Worksharing

  #pragma omp for

  #pragma omp sections

- Data Environment

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

- Synchronization

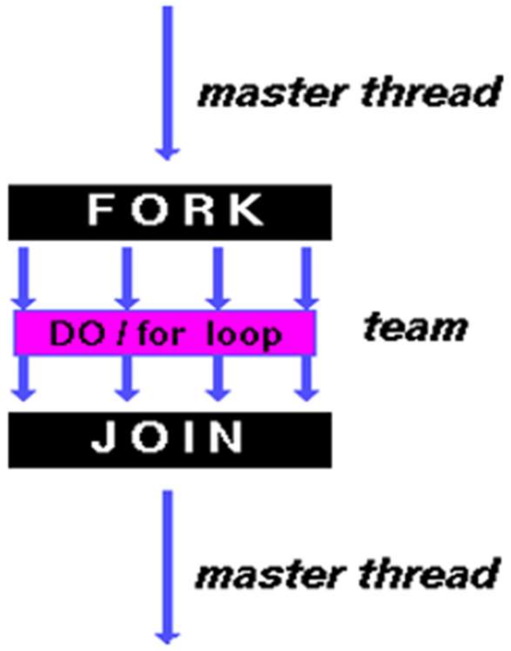
  #pragma omp barrier

  #pragma omp critical

Loop Constructs: Parallel for

In C/C++:

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



Scheduling of loop iterations

Schedule clause:
- specifies how loop iteration are divided among team of threads

Supported scheduling types

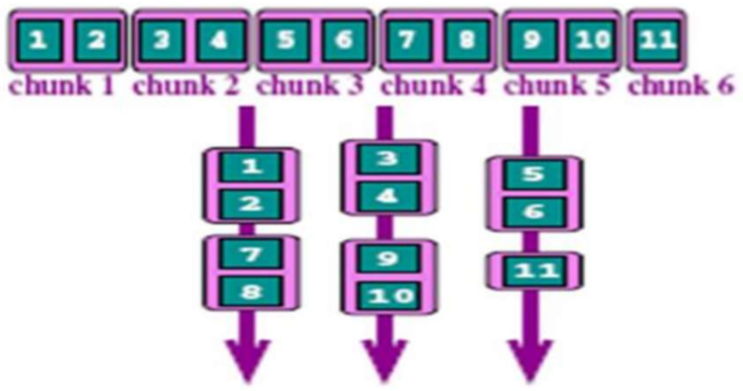
- Static
- Dynamic
- Guided
- Runtime

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

schedule Clause

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
- Deterministic



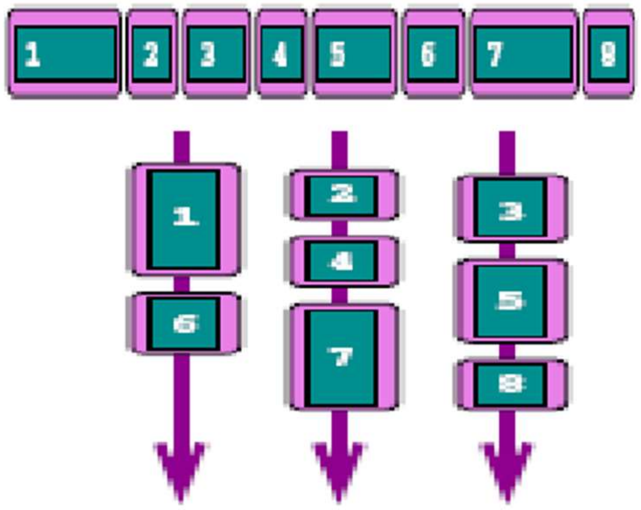
Example:
loop of length 16, with 3 threads, and chunk size of 2:

schedule Clause (cont…)

schedule(dynamic, [n])

- Iteration of loop are divided into chunks containing **n** iterations each
- Default chunk size is 1
- Iterations picked by threads depends upon the relative speeds of thread execution

```
#pragma omp parallel for schedule (dynamic)
for(i=0; i<8; i++)
{
    … (loop body)
}
```



schedule Clause (cont…)

schedule (guided, [n])

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

chunk size = max((num_of_iterations remaining / 2*num_of_threads), n)

- the formula may differ across compiler implementations

schedule (runtime)

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

```
export OMP_SCHEDULE=“static, 4”
```

Data Scoping in OpenMP

```
#pragma omp parallel [data scope clauses ...]
```

- shared
- private
- firstprivate
- lastprivate
- default

shared Clause (Data Scope)

- Shared data 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

private Clause (Data Scope)

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(tid)
  for(i=0; i<n; i++)
  {
        tid = omp_get_thread_num();
        printf(“ My rank is %d ”, tid)
  }
```

firstprivate Clause (Data Scope)

The clause combines behavior of private clause with automatic initialization of the variables in its list with values prior to parallel region

Example:

```
int b=51, n=100 ;
printf(“Before parallel loop: b=%d ,n=%d\n”,b,n)
#pragma omp parallel for private(i), firstprivate(b)
for(i=0; i<n; i++)
    {
        a[i] = i + b;
    }
```

lastprivate Clause (Data Scope)

Performs finalization of private variables

Each thread has its own copy

Example:

```
    b=51,n=100;
    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 ;
        }
//After parallel region:     a = 150
```

default Clause (Data Scope)

- Defines the default data scope within parallel region

- default (private | shared | none)

More clauses for parallel directive

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

- nowait
- if
- reduction

nowait Clause

```
#pragma omp parallel nowait
```

- By default there is implicit barrier at the end of parallel region
- Allows 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 if (flag != 0)
{
    // ...some stuff
}
```

if (integer expression)

- Determines if the region should be parallelized
- Useful option when data is too small

reduction Clause

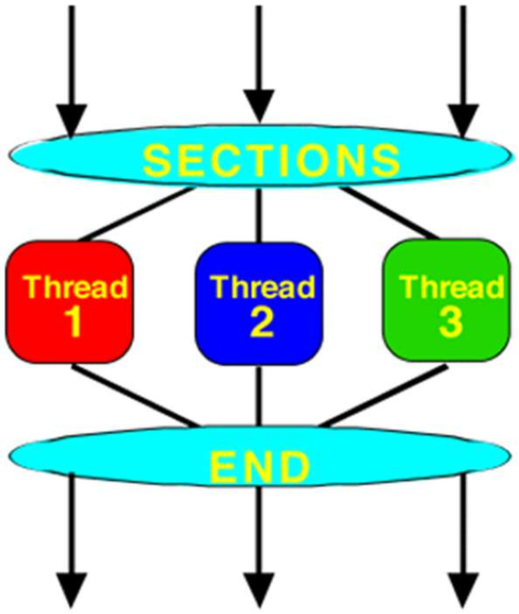
- Performs a collective operation on variables according to the given operators
  - built-in reduction operations such as +, *, -, max, min, logical operators
  - user can define his/her own operations
- Makes reduction variable as private
  - The variable is initialized according to reduction operator e.g. 0 for addition
- Each thread will perform the operation in its local variable
- Finally local results are combined into global result in shared variable

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

for (i = 1; i <= N; i++)
 {
      result += i ;
  }
```

Work sharing : Section Directive

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

```
#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
 {
     // read value of “a” from file
  }
#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
 {
   // reading or writing data etc.
 }
```

```
#pragma omp master
     ----- block of code--
```

Race condition

Problem:

Finding the largest element in a list of numbers

```
Max = 10
#pragma omp parallel for
for (i=0;i<N;i++)
{
     if (a(i) > Max)
         Max = a(i) ;
}
```

Thread 0	**Thread 1**
Read a(i) value = 12	Read a(i) value = 11
Read Max value = 10	Read Max value = 10
If (a(i) > Max) (12 > 10)	If (a(i) > Max) (11 > 10)
Max = a(i) (i.e. 12)	Max = a(i) (i.e. 11)

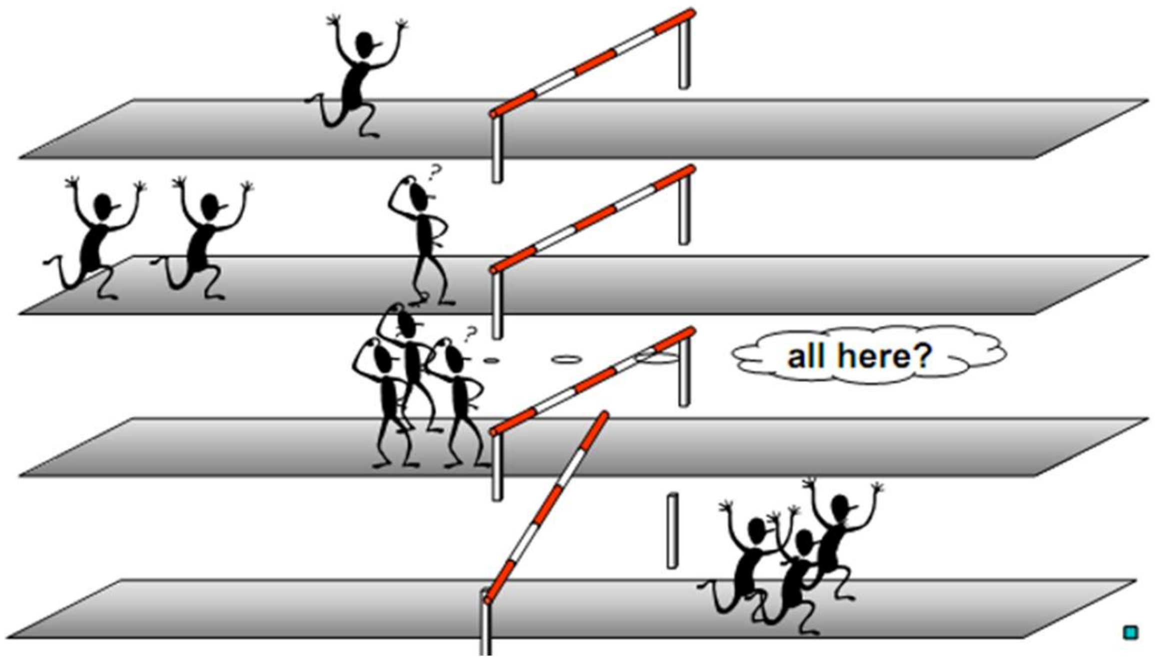
Synchronization: Critical Section

Critical section restricts access to the enclosed code to only one thread at a time

```
Max = 10
#pragma omp parallel for
for (i=0;i<N;i++)
{
    …. other work….
   #pragma omp critical
   {
     if (a(i) > Max)
         Max = a(i) ;
    }
    …. other work….
}
```

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(“thread %d: x=%d”,tid,x);
     #pragma omp barrier
         printf(“thread %d: 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

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

Some Runtime Library Routines

- Set number of threads for parallel region

  omp_set_num_threads(integer)

- Get number of threads for parallel region

  int omp_get_num_threads()

- Get thread ID / rank

  omp_get_thread_num()

Environment Variables

- 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 clause

  `#pragma omp parallel num_threads(integer)`

- Run-time function

  `omp_set_num_threads(integer)`

- Environment Variable

  `OMP_NUM_THREADS`

Priority

↓

Data Locality

Uniform Memory Access (UMA) – all cores have equal access times to shared memory

Non-uniform Memory Access (NUMA) – cores have higher access times to non-local shared memory

First touch policy

```
int a[N];
#pragma omp parallel for
For LOOP to initialize data
```

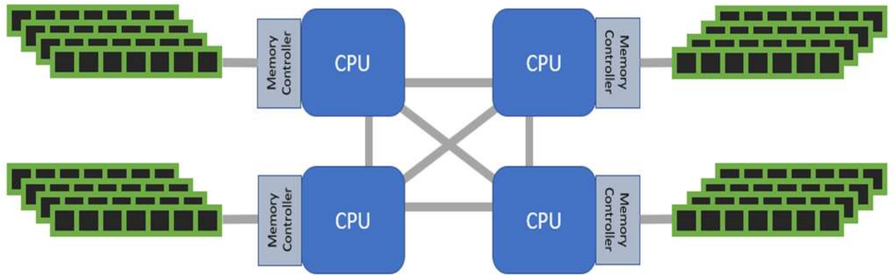


Fig: NUMA

CPU Pinning

Default thread placement policy depends upon the OpenMP implementation being used.

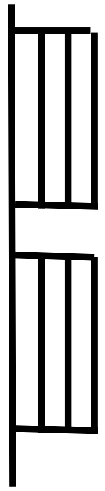
In absence of thread placement policy, during execution threads may migrate across different physical cores and therefore suffer data locality issues.

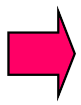
CPU pinning enables binding of threads to cores.

Granularity of Parallelization

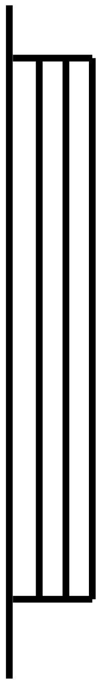
Coarse-grain parallelism vs. Fine grain parallelism

```
#pragma omp parallel for
 for(i=0; i<n; i++)
  {
        // work 1;
  }

#pragma omp parallel for
 for(i=0; i<n; i++)
  {
        // work 2 ;
  }
```



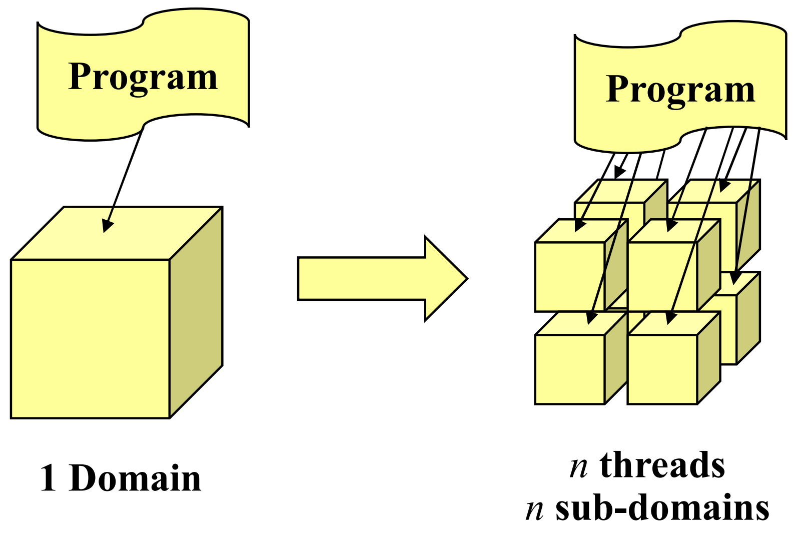


```
#pragma omp parallel
{
  #pragma omp for
   for(i=0; i<n; i++)
   {
        // work ;
   }
  #pragma omp for
   for(i=0; i<n; i++)
   {
        // work ;
   }
}
```



Subroutines having multiple independent DO/for Loops are good candidates

Domain Decomposition



```
#pragma omp parallel default(private) shared(N,nthreads)
{
  nthreads = omp_get_num_threads()
  iam = omp_get_thread_num()
  ichunk = N/nthreads
  istart = iam*ichunk
  iend = (iam+1)*ichunk -1

  my_sum(istart, iend, local)

   #pragma omp atomic
    global = global + local
}
```

Some Tips

- Identify Loop-level parallelism: Run the loop backwards and see if same results are produced
- Load imbalance due to branching statements, sparse matrices: schedule(dynamic)
- Parallelization of less compute intensive loops: Use small number of threads e.g.
  #pragma omp parallel num_threads(4)
- Parallelize initialization of input data – speedup and data locality

Advantages and Disadvantages

Advantages

- Shared address space provides user friendly programming
- Ease of programming
- Data sharing between threads is fast and uniform (low latency)
- Incremental parallelization of sequential code
- Leaves thread management to compiler
- Directly supported by compiler

Disadvantages

- Internal details are hidden
- Programmer is responsible for specifying synchronization, e.g. locks
- Cannot run across distributed memory
- Performance limited by memory architecture
- Lack of scalability between memory and CPUs
- Requires compiler which supports OpenMP
- Bigger machines are heavy on budget

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>
```

References

The contents of the presentation have been adapted from several sources. Some of the sources are as following:

www.openmp.org/

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

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

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

A "Hands-on" Introduction to OpenMP (Part 1/2) | Tim Mattson, Intel

Introduction to Parallel Computing on Ranger, Steve Lantz, Cornell University

Thank You