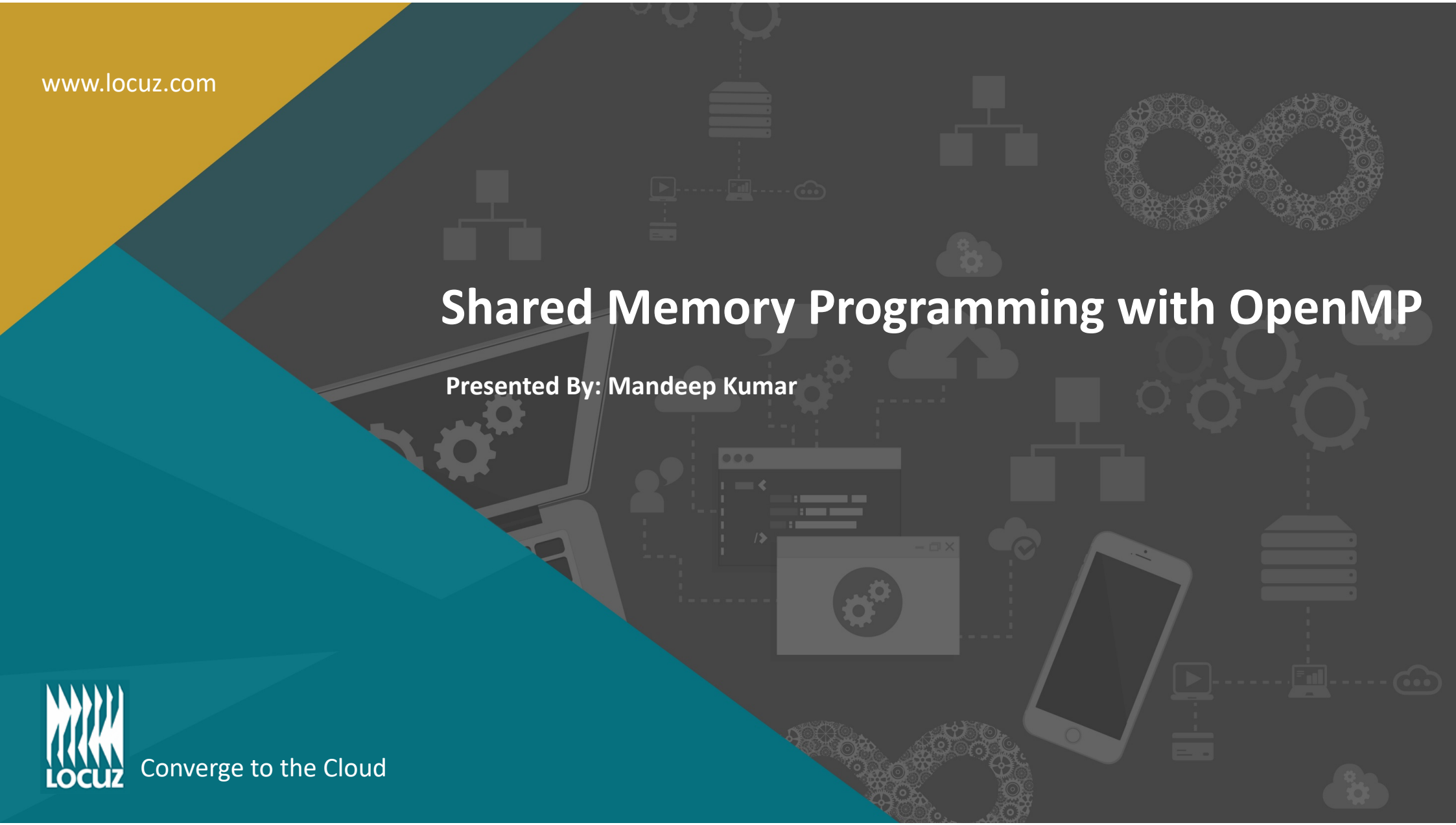
www.locuz.com

Shared Memory Programming with OpenMP

Presented By: Mandeep Kumar



LOCUZ

Converge to the Cloud

Parallelism

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

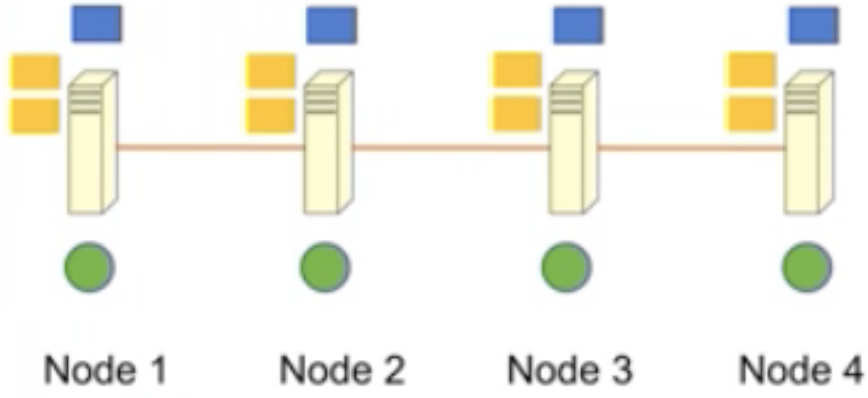
Why Parallelism Is Good

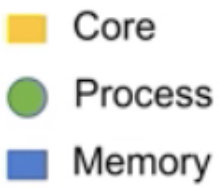
- The Trees : We like parallelism because, as the number of processing units working on a problem grows, we can solve the same problem in less time
- The Forest : We like parallelism because, as the number of processing units working on a problem grows, we can solve bigger problems

Shared Memory Vs Distributed Memory Model

Shared Memory Model

Distributed Memory Model





OpenMP vs MPI

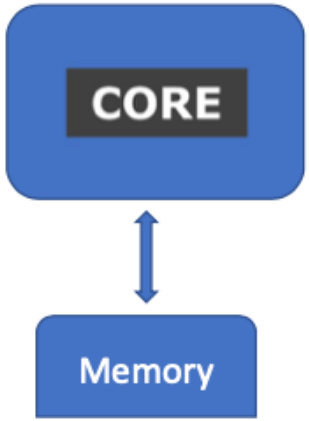
OpenMP	MPI
Shared memory model	Distributed memory model
On Multi-core processors	On Distributed network
Directive based	Message based
Easier to program and debug	Flexible and expressive

Agenda

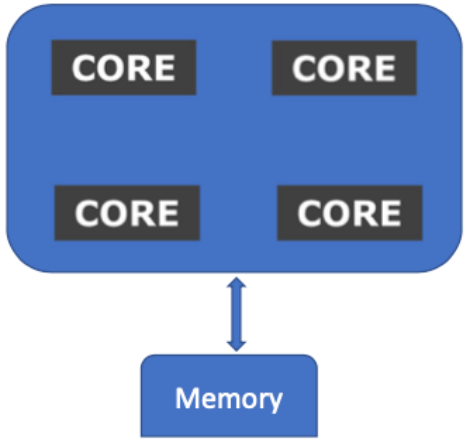
- **Core concepts**
- **What is OpenMP**
- **OpenMP Programming and Execution Model**
- **OpenMP Constructs**
- **Granularity of Parallelization**
- **Advantages and Disadvantages of OpenMP**

Basis System Architecture

Single Core Processor



Multi Core Processor

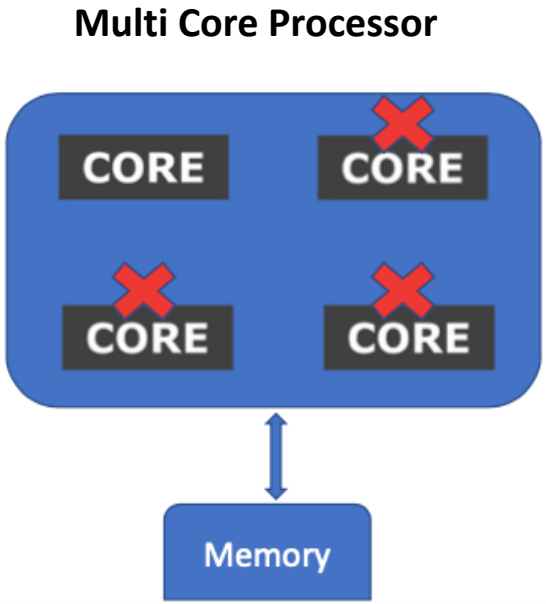


Sequential Program Execution

When you run sequential program
- Instructions executed in serial
- Other cores are idle

Waste of available resource… We want all core to be used to execution program.

How?

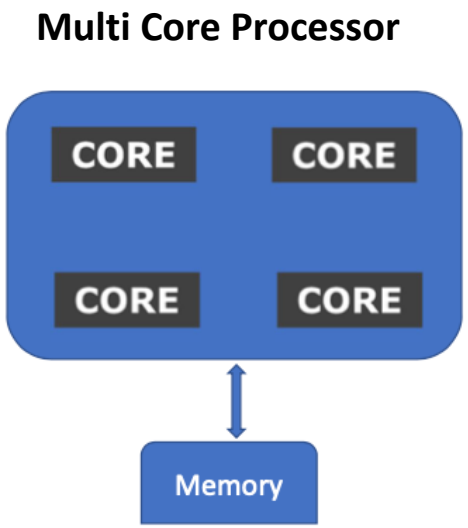


Process and Thread

- 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



What is OpenMP?

- Open Specification for Multi Processing
- Provides multi-threaded parallelism
- OpenMP is an Application Program Interface (API) for writing multi-threaded, shared memory parallelism.
- It is an specification for
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- Easy to create multi-threaded programs in C, C++ and Fortran.

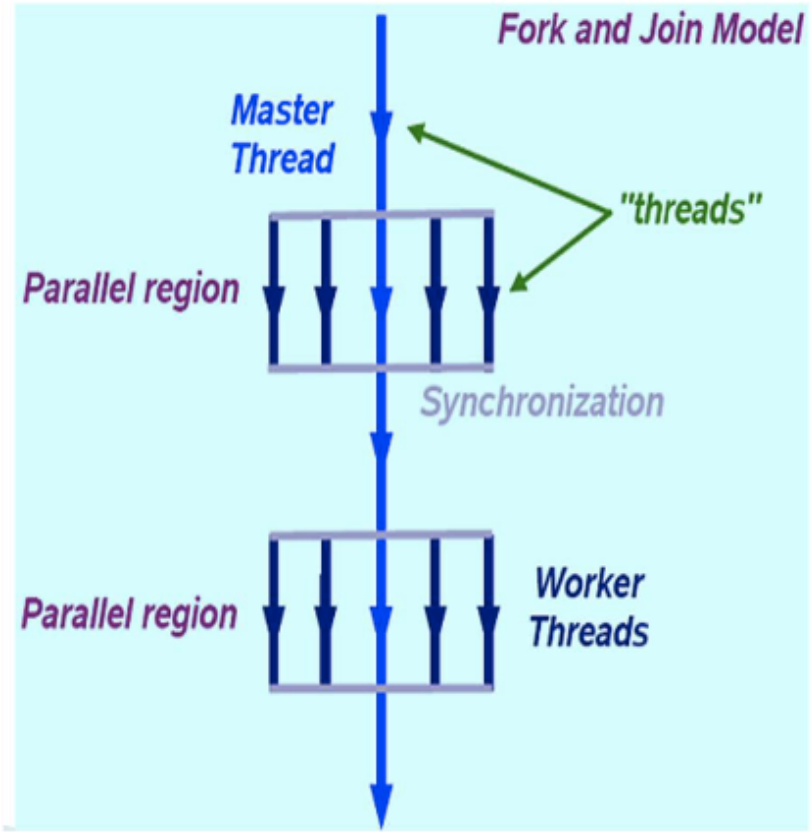
OpenMP History

- 1997: Version 1.0 for Fortran
- 1998: Version 1.0 for C/C++
- 2002-2005: Versions 2.0-2.5, Merger of Fortran and C/C++ specifications
- 2008: Version 3.0, Incorporates Task Parallelism
- 2013: Version 4.0, Support for Accelerators, SIMD support
- 2018: Version 5.0, C11/C++17/Fortran 2008 support

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



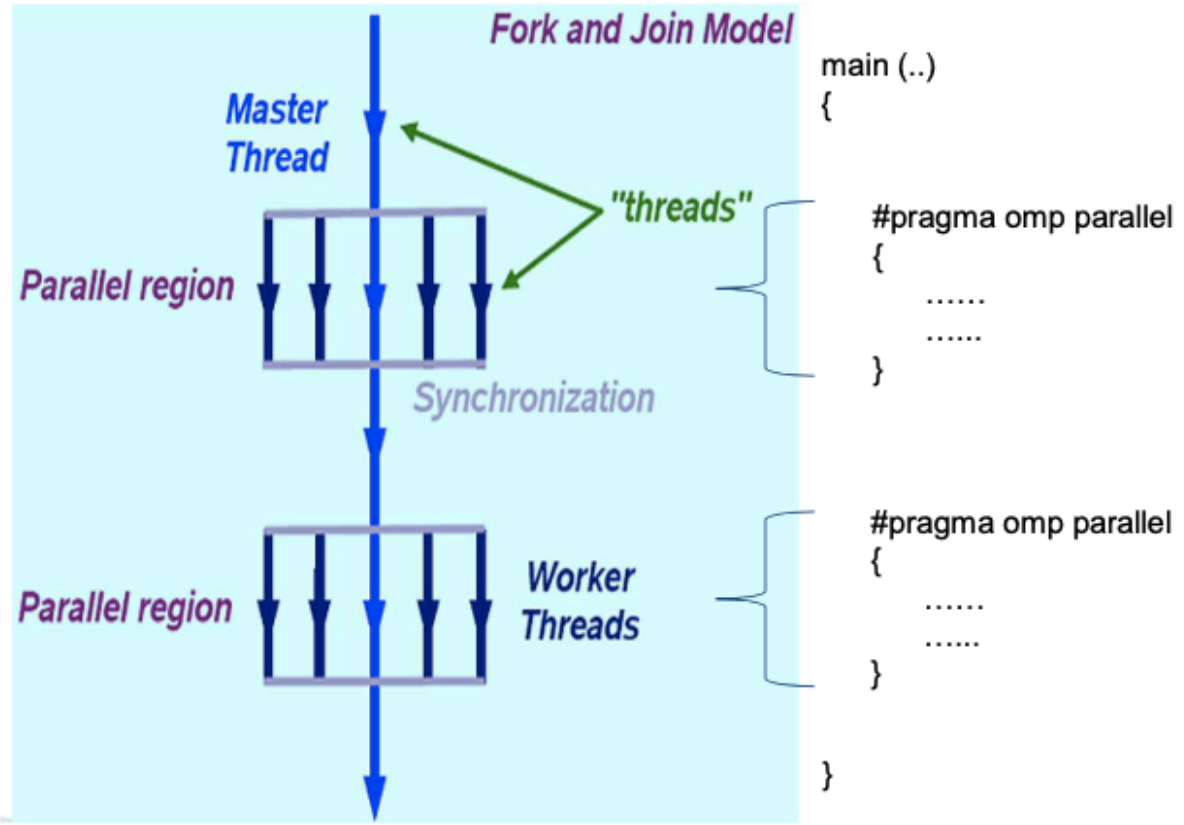
OpenMP Basic Syntax

Header file: `#include <omp.h>`

Parallel Region:

C:

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

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



Parallel Region

Fork a team of N threads {0…N-1}

Without it, all code 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

OpenMP Directives

- OpenMP directives are comments in source code that specify parallelism for shared memory parallel (SMP) machines
- 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

C/C++

```
#pragma omp parallel
 {
  …
 }

#pragma omp parallel for
 for(…) { …
 }
```

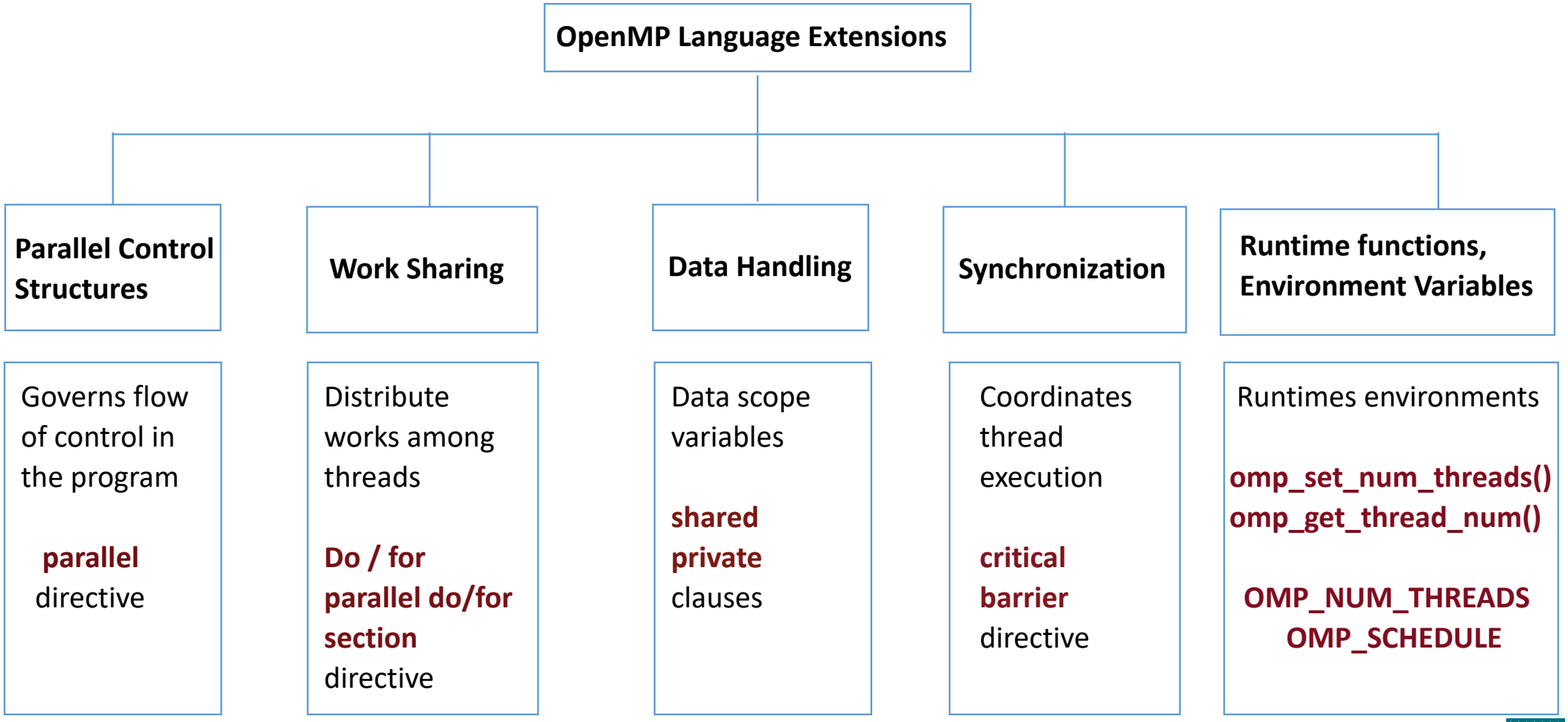
FORTRAN

```
!$OMP parallel
  …
!$OMP end parallel

!$OMP parallel do
  Do …
!$OMP end parallel do
```

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 synchronisation protect against race conditions
- Synchronization is expensive
  — change data storage attributes for minimizing synchronization and improving cache reuse

OpenMP Constructs



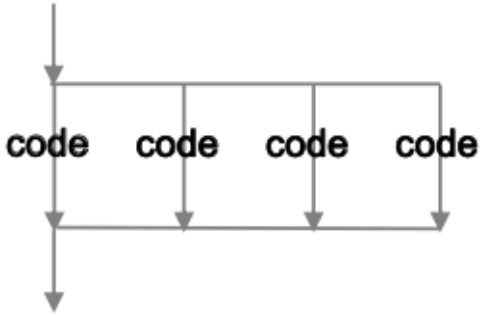
OpenMP Constructs

- 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`

Parallel Directives

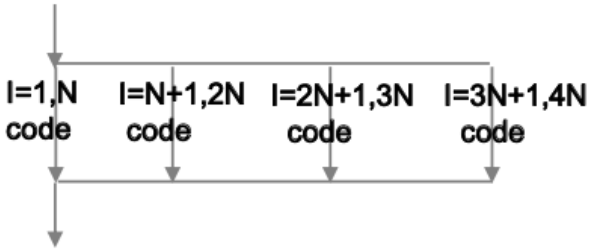
- **Replicated** — executed by all threads
- **Worksharing** — divided among threads

```
#pragma omp parallel
 {
    code();
 }
```



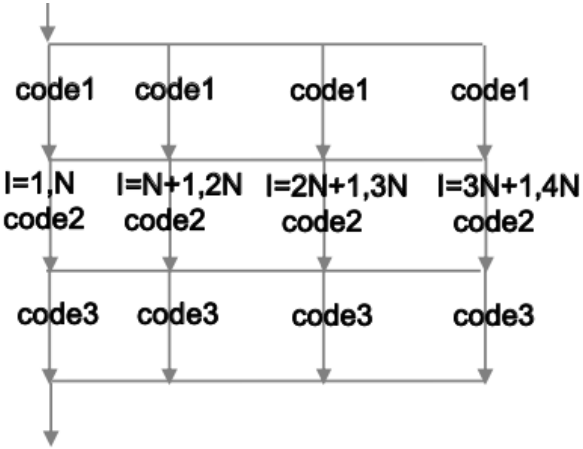
Replicated

```
#pragma omp for
  for(I=1; I<=4*N; I++)
   {
      code();
   }
```



Worksharing

```
#pragma omp parallel
 {
    code1();
    #pragma omp for
      for(I=1; I<=4*N; I++)
       {
         code2();
       }
     code3();
 }
```



Combined

Worksharing

Use OpenMP directives to specify worksharing in a parallel region, as well as synchronization

```
#pragma omp parallel
 {
 #pragma omp
 } //end parallel
```

Code block	Thread action
for	Worksharing
sections	Worksharing
single	One thread
master	One thread
critical	One thread at a time

parallel do/for
parallel sections

Directives can be combined, if a parallel region has just one worksharing construct.

First OpenMP Program

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

int main()
{
    #pragma omp parallel
   {
       printf("Parallel hello to you!\n");
   }
   return(0);
}
```

First OpenMP Program

- **Check your system support:** locate omp.h
  /usr/lib/gcc/x86_64-redhat-linux/4.8.2/include/omp.h

- **Compilation:** gcc -fopenmp first_OpenMP.c

- **Flags:**
  GNU: **-fopenmp** for Linux, Solaris, AIX, MacOSX, Windows.
  IBM: **-qsmp=omp** for Windows, AIX and Linux.
  Sun: **-xopenmp** for Solaris and Linux.
  Intel: **-qopenmp** on Linux or Mac, **/Qopenmp** on Windows.
  PGI: **-mp**

- **Conditional Compilation:**

```
#ifdef _OPENMP
        printf("Compiled with OpenMP support:%d", _OPENMP);
#else
        printf("Compiled for serial condition.");
#endif
```

- **Execution:** ./a.out

```
$ gcc –fopenmp first_OpenMP.c
$ ./a.out
Parallel hello to you!
Parallel hello to you!
Parallel hello to you!
Parallel hello to you!
Parallel hello to you!
Parallel hello to you!
Parallel hello to you!
Parallel hello to you!
Parallel hello to you!
Parallel hello to you!
$
```

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

LOCUZ

Setting Environment Variable and Executing

```
$ echo $SHELL                          //Know your shell
$ /bin/bash
$ export OMP_NUM_THREADS=4             // Setting environment variables
$ gcc -fopenmp first_OpenMP.c          //Compilation
$ ./a.out                              // Execution
Parallel hello to you!
Parallel hello to you!
Parallel hello to you!
Parallel hello to you!
```

Second Example

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

int main(int argc, char *argv[])
{
    int id;
    double wtime;

    printf("Number of processors available: %d\n",omp_get_num_procs());
    printf("Number of threads: %d\n",omp_get_max_threads());
    wtime=omp_get_wtime();
    printf("Outside the parallel region\n");
    id=omp_get_thread_num();
    printf("Hello from process %d\nGoing inside the parallel region\n”,id);

   #pragma omp parallel private(id)
    {
        id=omp_get_thread_num();
        printf("Hello from process %d\n",id);
    }

    wtime=omp_get_wtime() - wtime;
    printf("Back from the parallel region. Normal execution\n");
    printf("Elapsed wall clock time %f\n",wtime);
    return 0;
}
```

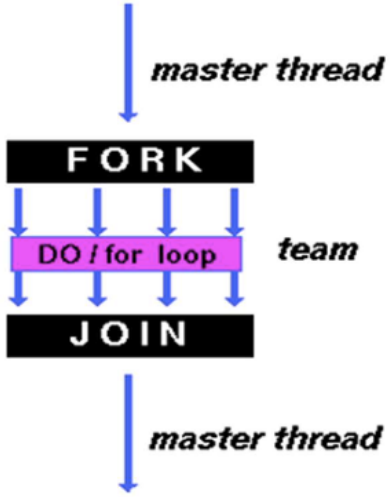
Second Example Output

```
Number of processors available: 10
Number of threads: 10
Outside the parallel region
Hello from process 0
Going inside the parallel region
Hello from process 8
Hello from process 5
Hello from process 4
Hello from process 1
Hello from process 2
Hello from process 7
Hello from process 6
Hello from process 9
Hello from process 0
Hello from process 3
Back from the parallel region. Normal execution
Elapsed wall clock time 0.006727
```

```
$ gcc -fopenmp second_OpenMP.c
$ ./a.out
```


Loop Constructs: Parallel for

```
In C/C++:

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



OpenMP Clauses

- Directives dictate what the OpenMP thread team will do

- Examples:
  - Parallel regions are marked by the parallel directive
  - Work sharing loops are marked by do, for directives (Fortran, C/C++)

- Clauses control the behaviour of any particular OpenMP directive

- Examples:

1. Scoping of variables: private, shared, default
2. Initialization of variables: copyin, firstprivate
3. Scheduling: static, dynamic, guided
4. Conditional application: if
5. Number of threads in team: num_threads

Scheduling of loop iterations

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

- Supporting Scheduling types
  - **Static**
  - **Dynamic**
  - **Guided**
  - **Runtime**

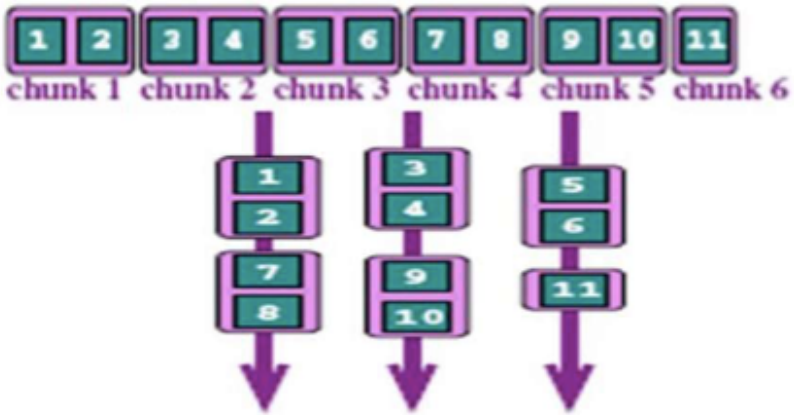
```
#pragma omp parallel for schedule (type, [chuck size])
  {
          // …some stuff
  }
```

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:



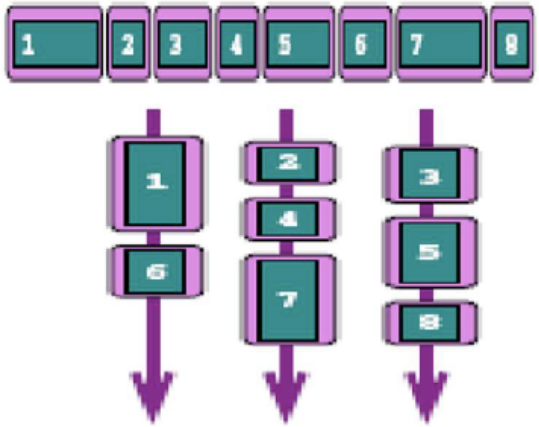
Example

```
#include <stdio.h>
#include <omp.h>
#include <unistd.h>
#define THREADS 8
int main()
{
    int i,N=10;
    printf("First part….\n");

    #pragma omp parallel for num_threads(THREADS)
    for (i = 0; i < N; i++) {
        printf("Thread %d is doing iteration %d.\n", omp_get_thread_num( ), i);
    }

    printf("\n\nSecond part….\n");

    #pragma omp parallel for schedule(static) num_threads(THREADS)
    for (i = 0; i < N; i++) {
        sleep(i); /* wait for i seconds */
        printf("Thread %d has completed iteration %d.\n", omp_get_thread_num( ), i);
    }

    return 0;
}
```

schedule Clause

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)
 }
```

```
#pragma omp parallel for schedule(dynamic) num_threads(THREADS)
    for (i = 0; i < N; i++) {
        sleep(i); /* wait for i seconds */
        printf("Thread %d has completed iteration %d.\n", omp_get_thread_num( ), i);
    }
```

schedule Clause

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

```
#pragma omp parallel for schedule(guided) num_threads(THREADS)
   for (i = 0; i < N; i++) {
      sleep(i); /* wait for i seconds */
      printf("Thread %d has completed iteration %d.\n", omp_get_thread_num( ), i);
   }
```

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:

```
#pragma 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 behaviour of private clause with automatic initialization of the variables in its list with values prior to parallel region

```
Example:


int a, i, 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 + b;

         }
```

lastprivate Clause (Data Scope)

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

```
Example:

int a, i, 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

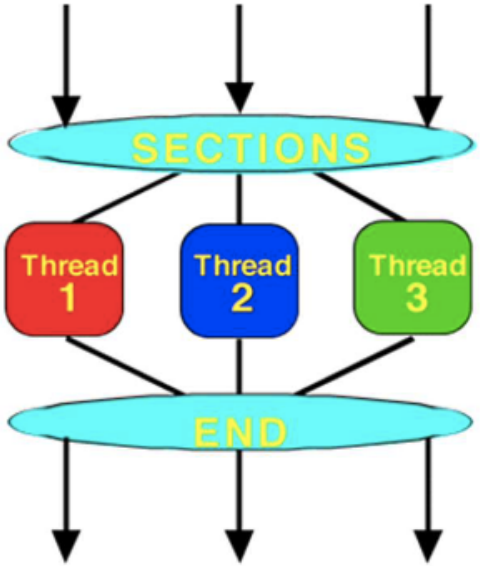
- Performs a collective operation on variables according to the given operators
  - — built-in reduction operations such as +, *, -, max, min, logical operators
  - — user can define his/her own operations
- Makes reduction variable as private
  - — The variable is initialized according to reduction 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 section
   {
     #pragma omp section
        x_calulation();
     #pragma omp section
        x_calulation();
     #pragma omp section
        x_calulation();
    }
```



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 and writing data etc.
}
```

```
#pragma omp master

- - block of code - -
```

Synchronization Constructs

- **CRITICAL:** Mutual Exclusion
- **ATOMIC:** Atomic Update
- **BARRIER:** Barrier Synchronization

Example

```
#include <stdio.h>

int main()
{
    int x=0;

    x+=1;
    printf(“%d\n”,x);
    return 0;
}
```

```
$ gcc
example.c
$ ./a.out
1
$
```

Example

Race Condition

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

int main()
{
       int x=0, size=8;
       omp_set_num_threads(size);
       #pragma omp parallel shared(x)
       {
         x+=1;
       }
       printf("%d\n",x);
      return 0;
}
```

```
$ gcc -fopenmp example_OpenMP.c
$ ./a.out
5
$ ./a.out
8
$ ./a.out
8
$ ./a.out
4
$ ./a.out
3
$
```

Race Condition

Intended

Thread 0		Thread 1		Value
read	←			0
increment				0
write	→			1
		read	←	1
		increment		1
		write	→	2

Possible…

Thread 0		Thread 1		Value
				0
read	←			0
increment		read	←	0
write	→	increment		1
		write	→	1
				1

- In a critical section, need mutual exclusion to get intended result
- The following OpenMP directives prevent this race condition:

#pragma omp critical – for a code block (C/C++)
#pragma omp atomic – for single statement

Race Condition: Real Life Example

- So, how do we ensure that there is no contention and both person get to access the bathroom?

Use Lock and Key

Critical Construct

```
#pragma omp critical [name]
  structured-block
```

```
Example:
        #pragma omp parallel
         {
              #pragma omp critical(long_critical_name)
                    doSomeCriticalWork_1();
              #pragma omp critical
                    doSomeCriticalWork_2();
              #pragma omp critical
                    doSomeCriticalWork_3();
          }
```

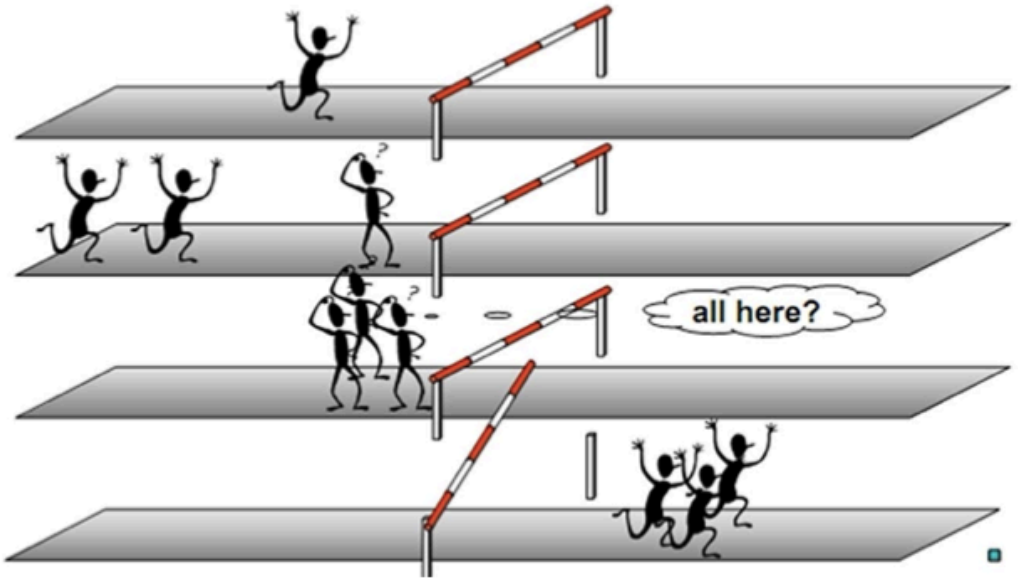
Example

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

int main()
{
        int x=0, size=8;
        omp_set_num_threads(size);
        #pragma omp parallel shared(x)
        {
           #pragma omp critical
           {
                x+=1;
           }
        }
        printf("%d\n",x);
       return 0;
}
```

```
$ gcc -fopenmp example_OpenMP.c
$ ./a.out
8
$ ./a.out
8
$ ./a.out
8
$ ./a.out
8
$ ./a.out
8
$
```

Barrier Directive

Synchronizes all the threads in a team



Barrier Directive

```
#pragma omp barrier   //Threads wait until all threads reach this point
```

```
Example:

#include <stdio.h>
#include <omp.h>
int main()
{
   int x=0;
   #pragma omp parallel
   {
     #pragma omp master
        x=1;
     #pragma omp barrier
        printf("%d\n",x);
   }
 return 0;
}
```

Be careful not to cause deadlock:

No barrier inside of critical, master, sections, single!

Atomic Directive

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

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

Example

Race Condition

```
#include <stdio.h>

int main()
 {
    int i, n=5, sum;
    for (i = 1; i <= n ; i++)
    {
       sum += i ;
    }
    printf("%d\n",sum);
    return 0;
}
```

```
$ gcc example.c
$ ./a.out
15
$
```

Reduction Clause

Recall previous example,

— Simple parallel-for doesn't work due to race condition on shared sum
— Best solution is to apply OpenMP's reduction clause
— Doing private partial sums is fine too; add a critical section for sum of i

Example

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

int main()
{
       int i, n=5, sum;
       omp_set_num_threads(8);
       #pragma omp parallel for reduction(+:sum)
         for(i=1; i<=n;i++)
           {
               sum+=i;
           }
       printf("%d\n",sum);
      return 0;
}
```

```
$ gcc -fopenmp example_OpenMP.c
$ ./a.out
15
$ ./a.out
15
$ ./a.out
15
$
```

Table: OpenMP Directives

Name	Parallel	Worksharing	Synchronization	Data environment	Description
parallel	Yes	No	No	No	Defines a parallel region, which is code that will be executed by multiple threads in parallel
for	No	Yes	No	No	Causes the work done in a for loop inside a parallel region to be divided among threads
sections	No	Yes	No	No	Identifies code sections to be divided among all threads
single	No	Yes	No	No	Lets you specify that a section of code should be executed on a single thread, not necessarily the master thread
master	No	No	Yes	No	Specifies that only the master thread should execute a section of the program
critical	No	No	Yes	No	Specifies that code is only executed on one thread at a time
barrier	No	No	Yes	No	Synchronizes all threads in a team; all threads pause at the barrier, until all threads execute the barrier
atomic	No	No	Yes	No	Specifies that a memory location that will be updated atomically
flush	No	No	Yes	No	Specifies that all threads have the same view of memory for all shared objects
ordered	No	No	Yes	No	Specifies that code under a parallelized for loop should be executed like a sequential loop
threadprivate	No	No	No	Yes	Specifies that a variable is private to a thread

Table: OpenMP Functions

Name	Environment Execution	Lock	Description
omp_set_num_threads	Yes	No	Sets the number of threads in upcoming parallel regions, unless overridden by a num_threads clause.
omp_get_num_threads	Yes	No	Returns the number of threads in the parallel region.
omp_get_max_threads	Yes	No	Returns an integer that is equal to or greater than the number of threads that would be available if a parallel region without num_threads were defined at that point in the code.
omp_get_thread_num	Yes	No	Returns the thread number of the thread executing within its thread team.
omp_get_num_procs	Yes	No	Returns the number of processors that are available when the function is called.
omp_in_parallel	Yes	No	Returns nonzero if called from within a parallel region.
omp_set_dynamic	Yes	No	Indicates that the number of threads available in upcoming parallel regions can be adjusted by the run time.
omp_get_dynamic	Yes	No	Returns a value that indicates if the number of threads available in upcoming parallel regions can be adjusted by the run time.
omp_set_nested	Yes	No	Enables nested parallelism.
omp_get_nested	Yes	No	Returns a value that indicates if nested parallelism is enabled.
omp_init_lock	No	Yes	Initializes a simple lock.
omp_init_nest_lock	No	Yes	Initializes a lock.
omp_destroy_lock	No	Yes	Uninitializes a lock.
omp_destroy_nest_lock	No	Yes	Uninitializes a nestable lock.
omp_set_lock	No	Yes	Blocks thread execution until a lock is available.
omp_set_nest_lock	No	Yes	Blocks thread execution until a lock is available.
omp_unset_lock	No	Yes	Releases a lock.
omp_unset_nest_lock	No	Yes	Releases a nestable lock.
omp_test_lock	No	Yes	Attempts to set a lock but doesn't block thread execution.
omp_test_nest_lock	No	Yes	Attempts to set a nestable lock but doesn't block thread execution.

Table: OpenMP Datatypes

Name	Environment Execution	Lock	Description
omp_lock_t	No	Yes	A type that holds the status of a lock, whether the lock is available or if a thread owns a lock.
omp_nest_lock_t	No	Yes	A type that holds one of the following pieces of information about a lock: whether the lock is available, and the identity of the thread that owns the lock and a nesting count.

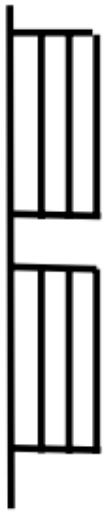
Table: OpenMP Environmental Variables

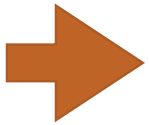
Environment Variable	Description
OMP_SCHEDULE	Modifies the behavior of the schedule clause when *schedule(runtime)* is specified in a *for* or *parallel for* directive.
OMP_NUM_THREADS	Sets the maximum number of threads in the parallel region, unless overridden by omp_set_num_threads or num_threads.
OMP_DYNAMIC	Specifies whether the OpenMP run time can adjust the number of threads in a parallel region.
OMP_NESTED	Specifies whether nested parallelism is enabled, unless nested parallelism is enabled or disabled with omp_set_nested.

Granularity of Parallelization

Coarse-grain parallelism vs. Fine grain parallelism

```
#pragma omp parallel for
  for(i=0, i<n; i++)
   {
      // work1;
   }

#pragma omp parallel for
  for(i=0, i<n; i++)
   {
      // work1;
   }
```





```
#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

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 parallization of sequential code
- Leaves thread management to compiler
- Directly support by compiler

Disadvantages

- Internal details are hidden
- Programmer is responsible for specifying synchronisation, 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

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/

Thanks!