Introduction to OpenMP

Ekpe Okorafor

School of Parallel Programming & Parallel Architecture for HPC

ICTP

October, 2014

A little about me!

- PhD Computer Engineering – Texas A&M University
- Computer Science Faculty
  - Texas A&M University
  - University of Texas at Dallas
  - Addis Ababa University, Ethiopia
  - African University of Science & Technology, Abuja, Nigeria
- Big Data Academy, Accenture Digital (Analytics) - USA

Topics

- Overview of OpenMP
- Basic Constructs
- Shared Data
- Parallel Flow Control
- Synchronization
- Reduction
- Synopsis of Commands

Intro (1)

- OpenMP is :
  - an API (Application Programming Interface)
  - NOT a programming language
  - A set of compiler directives that help the application developer to parallelize their workload.
  - A collection of the **directives**, **environment variables** and the **library routines**

Intro (2)

- OpenMP:
  - For shared memory systems
  - Add parallelism to functioning serial code
  - http://openmp.org
  - Compiler run-time does a lot of work for us
  - Divides up work
  - But we have to tell it how to use variables, where to run in parallel
  - Parallel regions need to be marked
    - Works by adding compiler directives to the code (This is invisible to non-openMP compilers)

Intro (3)

- OpenMP continues to evolve, with new constructs and features being added over time.
- Initially, the API specifications were released separately for C and Fortran. Since 2005, they have been released together.
- The table below chronicles the OpenMP API release history:

Month/Year	Version
Oct 1997	Fortran 1.0
Oct 1998	C/C++ 1.0
Nov 1999	Fortran 1.1
Nov 2000	Fortran 2.0
Mar 2002	C/C++ 2.0
May 2005	OpenMP 2.5
May 2008	OpenMP 3.0
Jul 2011	OpenMP 3.1
Jul 2013	OpenMP 4.0

A Programmer’s View of OpenMP

- OpenMP is a portable, threaded, shared-memory programming *specification* with “light” syntax
  - Exact behavior depends on OpenMP *implementation*!
  - Requires compiler support (C or Fortran)

- OpenMP will:
  - Allow a programmer to separate a program into *serial regions* and *parallel regions,* rather than T concurrently-executing threads*.*
  - Hide stack management
  - Provide synchronization constructs

- OpenMP will not:
  - Parallelize automatically
  - Guarantee speedup
  - Provide freedom from data races

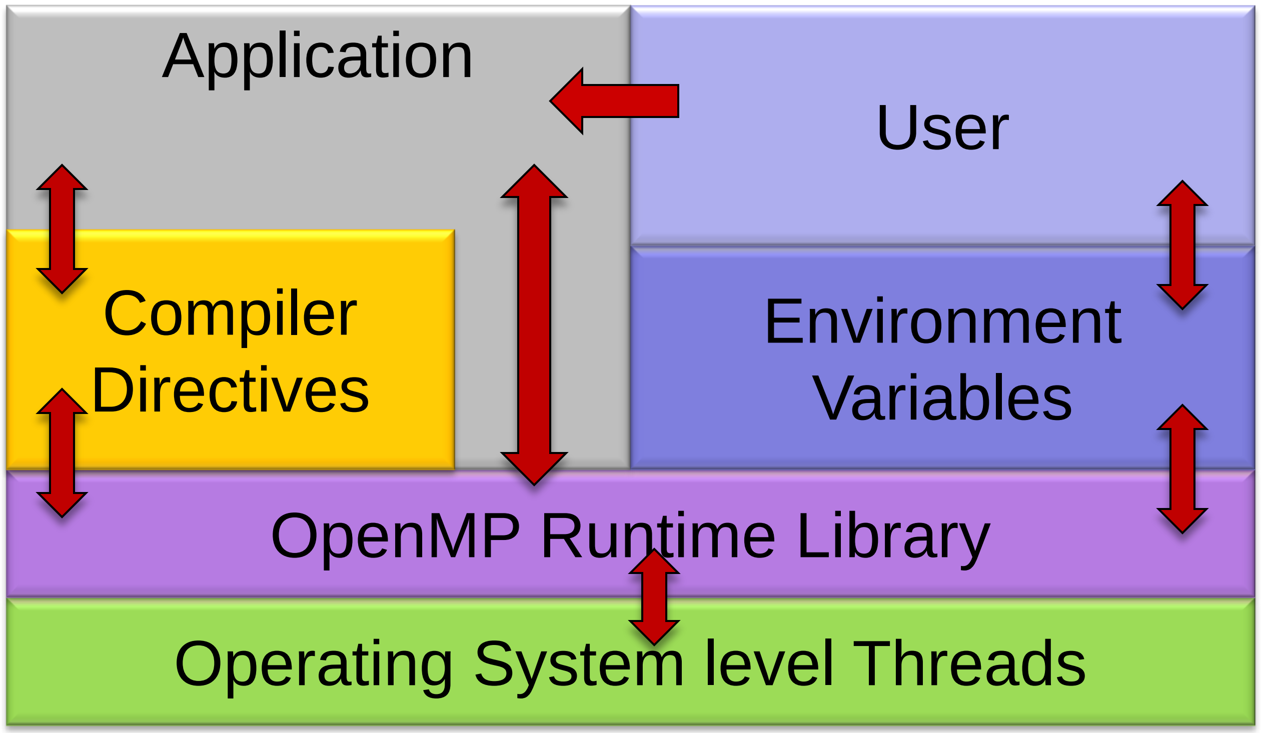
Components of OpenMP

Directives (44)
Parallel regions
Work sharing
Synchronization
Data scope attributes: • private • first private • last private • shared • reduction
Orphaning

Runtime library routines (35)
Number of threads
Thread ID
Dynamic thread adjustment
Nested Parallelism
Timers
API for locking

Environment variables (13)
Number of threads
Scheduling type
Dynamic thread adjustment
Nested Parallelism

OpenMP Architecture



Inspired by **OpenMp.org** introductory slides

Topics

- Overview of OpenMP
- Basic Constructs
- Shared Data
- Parallel Flow Control
- Synchronization
- Reduction
- Synopsis of Commands

OpenMP : Basic Constructs

To invoke library routines in C/C++ add

#include <omp.h>

near the top of your code

OpenMP Execution Model:

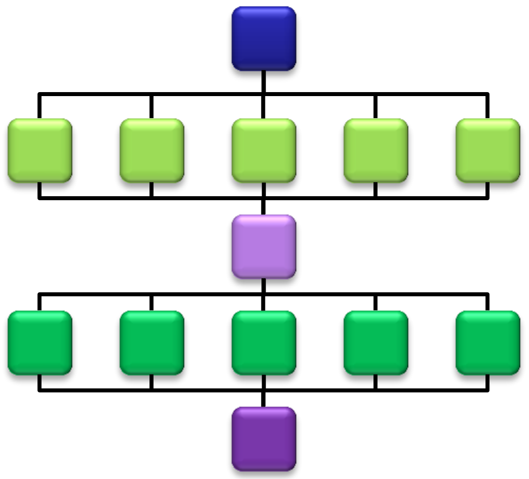
Sequential Part (master thread)

Parallel Region (group of threads)

Sequential Part (master thread)

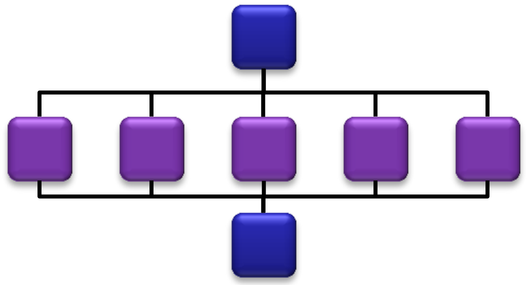
Parallel Region (group of threads)

Sequential Part (master thread)



C / C++ :

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



OpenMP : Basic Constructs

```
int main() {



  // Do this part in parallel

  {
    printf( "Hello, World!\n" );
  }

  return 0;
}
```

OpenMP : Basic Constructs

```
int main() {



   // Do this part in parallel
   #pragma omp parallel
   {
     printf( "Hello, World!\n" );
   }

   return 0;
 }
```

OpenMP : Basic Constructs

```
int main() {

   omp_set_num_threads(16);

   // Do this part in parallel
   #pragma omp parallel
   {
     printf( "Hello, World!\n" );
   }

   return 0;
 }
```

OpenMP : Runtime Library

Function:	omp_get_num_threads()
C/ C++	int omp_get_num_threads(void);
Fortran	integer function omp_get_num_threads()

Description:

Returns the total number of threads currently in the group executing the parallel block from where it is called.

Function:	omp_get_thread_num()
C/ C++	int omp_get_thread_num(void);
Fortran	integer function omp_get_thread_num()

Description:

For the master thread, this function returns zero. For the child nodes the call returns an integer between 1 and omp_get_num_threads()-1 inclusive.

HelloWorld in OpenMP

```
#include <omp.h>

main ()
{
   int nthreads, tid;
   #pragma omp parallel private(nthreads, tid)
   {
     tid = omp_get_thread_num();
     printf("Hello World from thread = %d\n", tid);
     if (tid == 0)
     {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
     }
   }
}
```

OpenMP directive to indicate START segment to be parallelized

Code segment that will be executed in parallel

OpenMP directive to indicate END segment to be parallelized

Compiling OpenMP Programs

C :

- Case sensitive directives
- Syntax :
  - #pragma omp directive [clause [clause]..]
- Compiling OpenMP source code :
  - **(IBM xlc compiler) : xlc_r -q64 -O2 -qsmp=omp file_name.c –o exe_file_name**
  - **(Linux C compiler) : gcc –o exe_file_name –openmp file_name.c**

Fortran :

- Case insensitive directives
- Syntax :
  - !$OMP directive [clause[[,] clause]…] (free format)
  - !OMP / COMP / *$OMP directive [clause[[,] clause]…] (free format)
- Compiling OpenMP source code :
  - **(IBM xlf compiler) : xlf_r -q64 -O2 -qsmp=omp file_name.f –o exe_file_name**
  - **(Linux Fortran compiler) : gfort -o exe_file_name –openmp file_name.f**

OpenMP: Environment Variables

Environment Variable:	OMP_NUM_THREADS
Usage :	OMP_NUM_THREADS ***n***
bash/sh/ksh:	*export* OMP_NUM_THREADS=8
csh/tcsh	*setenv* OMP_NUM_THREADS=8

Description:

Sets the number of threads to be used by the OpenMP program during execution.

Environment Variable:	OMP_DYNAMIC
Usage :	OMP_DYNAMIC {TRUE|FALSE}
bash/sh/ksh:	*export* OMP_DYNAMIC=TRUE
csh/tcsh	*setenv* OMP_DYNAMIC="TRUE"

Description:

When this environment variable is set to TRUE the maximum number of threads available for use by the OpenMP program is $OMP_NUM_THREADS.

OpenMP: Environment Variables

Environment Variable:	OMP_SCHEDULE
Usage :	OMP_SCHEDULE ***“schedule,[chunk]”***
bash/sh/ksh:	*export* OMP_SCHEDULE static,N/P
csh/tcsh	*setenv* OMP_SCHEDULE=“GUIDED,4”

Description:

Only applies to for and parallel for directives. This environment variable sets the schedule type and chunk size for all such loops. The chunk size can be provided as an integer number, the default being 1.

Environment Variable:	OMP_NESTED
Usage :	OMP_NESTED {TRUE|FALSE}
bash/sh/ksh:	*export* OMP_NESTED FALSE
csh/tcsh	*setenv* OMP_NESTED=“FALSE”

Description:

Setting this environment variable to TRUE enables multi-threaded execution of inner parallel regions in nested parallel regions.

OpenMP under the hood

Execution of an open MP based code :

- On encountering the C construct *#pragma omp parallel{* or the Fortran equivalent *!$omp parallel*, n extra threads are created
- *omp_get_num_threads()* returns the number of execution threads that can be utilized. The value returned by this call is between ***0*** to ***(OMP_NUM_THREADS – 1)*** for our convenience lets call this variable nthreads
- Code after the parallel directive is executed independently on each of the nthreads.
- On encountering the C construct *} (corresponding to #pragma omp parallel{)* or the corresponding Fortran equivalent *!$omp end parallel*, indicates the end of parallel execution of the code segment, the n extra threads are deactivated and normal sequential execution begins.

DEMO : Hello World

```
[griduser@localhost openMP]$ gcc -o helloc -openmp hello.c
hello.c(6) : (col. 4) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
[griduser@localhost openMP]$ export OMP_NUM_THREADS=8
[griduser@localhost openMP]$ ./helloc
Hello World from thread = 0
Number of threads = 8
Hello World from thread = 1
Hello World from thread = 2
Hello World from thread = 3
Hello World from thread = 4
Hello World from thread = 5
Hello World from thread = 6
Hello World from thread = 7
[griduser@localhost openMP]$
```

Topics

- Overview of OpenMP
- Basic Constructs
- Shared Data
- Parallel Flow Control
- Synchronization
- Reduction
- Synopsis of Commands

OpenMP: Data Environment … 1

- OpenMP program always begins with a single thread of control – *master thread*
- Context associated with the master thread is also known as Data Environment.
- Context is comprised of :
  - Global variables
  - Automatic variables
  - Dynamically allocated variables
- Context of the master thread remains valid throughout the execution of the program
- The OpenMP parallel construct may be used to either share a single copy of the context with all the threads or provide each of the threads with a private copy of the context.
- The sharing of Context can be performed at various levels of granularity
- Select variables from a context can be shared while keeping the context private etc.

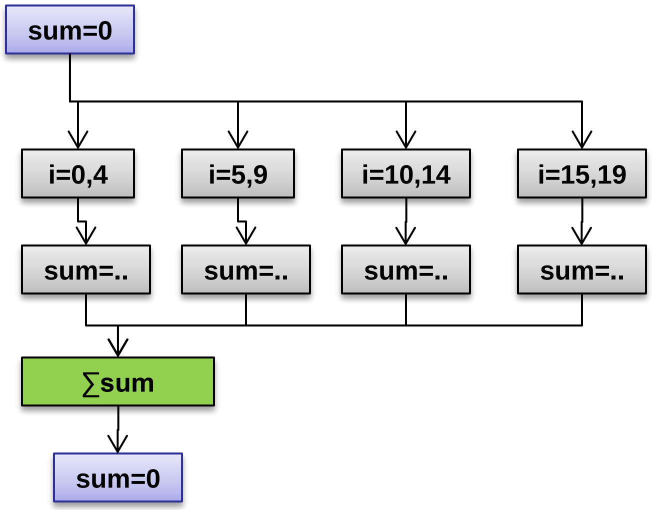
OpenMP: Data Environment … 2

- OpenMP data scoping clauses allows programmer to decide whether a variable’s execution context i.e. should the variable be shared or private.
- 3 main data scoping clauses in OpenMP (Shared, Private, Reduction) :
- *Shared :*
  - A variable will have a single storage location in memory for the duration of the parallel construct, i.e. references to a variable by different threads access the same memory location.
  - That part of the memory is shared among the threads involved, hence modifications to the variable can be made using simple read/write operations
  - Modifications to the variable by different threads is managed by underlying shared memory mechanisms
- *Private :*
  - A variable will have a separate storage location in memory for each of the threads involved for the duration of the parallel construct.
  - All read/write operations by the thread will affect the thread’s private copy of the variable .
- *Reduction :*
  - Exhibit both shared and private storage behavior. Usually used on objects that are the target of arithmetic reduction.
  - Example : summation of local variables at the end of a parallel construct

OpenMP: Reduction clause

- performs reduction on *shared variables* in list based on the operator provided.
- for C/C++ operator can be any one of :
- +, *, -, ^, |, ||, & or &&
- At the end of a reduction, the shared variable contains the result obtained upon combination of the list of variables processed using the operator specified.

```
sum = 0.0
#pragma omp parallel for reduction(+:sum)
  for (i=0; i < 20; i++)
    sum = sum + (a[i] * b[i]);
```



Topics

- Introduction
- Overview of OpenMP
- Basic Constructs
- Shared Data
- Parallel Flow Control
- Synchronization
- Reduction
- Synopsis of Commands

OpenMP: Work-Sharing Directives

- Work sharing construct divide the execution of the enclosed block of code among the group of threads.
- They do not launch new threads.
- No implied barrier on entry
- Implicit barrier at the end of work-sharing construct
- Commonly used Work Sharing constructs :
  - *for* directive *(C/C++ ; equivalent DO construct available in Fortran but will not be covered here) :* shares iterations of a loop across a group of threads
  - *sections* directive : breaks work into separate sections between the group of threads; such that each thread independently executes a section of the work.
  - *single* directive: serializes a section of code
- We will take a look at *for* and *sections* directives

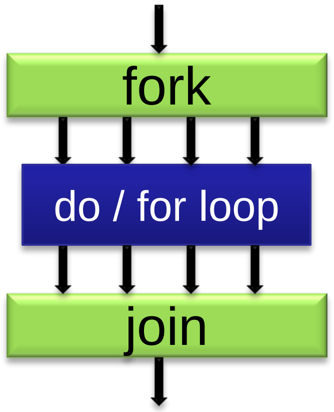
OpenMP: Schedule clause

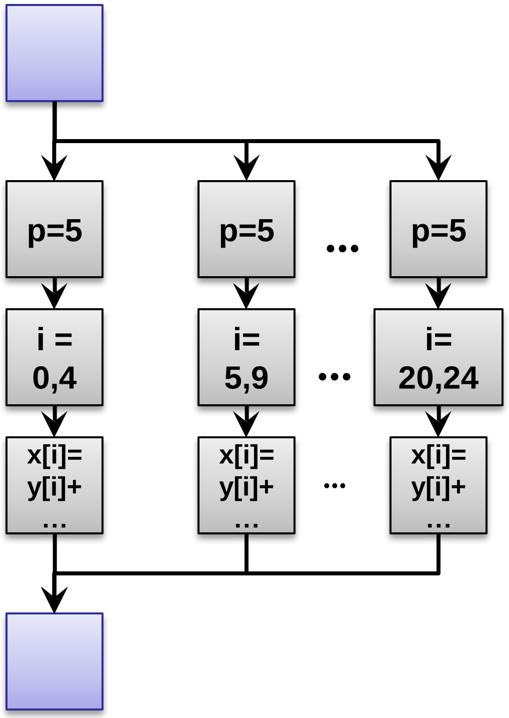
- The schedule clause defines how the iterations of a loop are divided among a group of threads
- *static* : iterations are divided into pieces of size chunk and are statically assigned to each of the threads in a round robin fashion
- *dynamic* : iterations divided into pieces of size chunk and dynamically assigned to a group of threads. After a thread finishes processing a chunk, it is dynamically assigned the next set of iterations.
- *guided* : For a chunk of size 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1. For a chunk with value k, the same algorithm is used in determining the chunk size with the constraint that no chunk should have less than k chunks except the last chunk.
- Default schedule is implementation specific while the default chunk size is usually 1

OpenMP *for* directive

- *for* directive helps share iterations of a loop between a group of threads
- If *nowait* is specified then the threads do not wait for synchronization at the end of a parallel loop
- The *schedule* clause describes how iterations of a loop are divided among the threads in the team (discussed in detail in the next few slides)

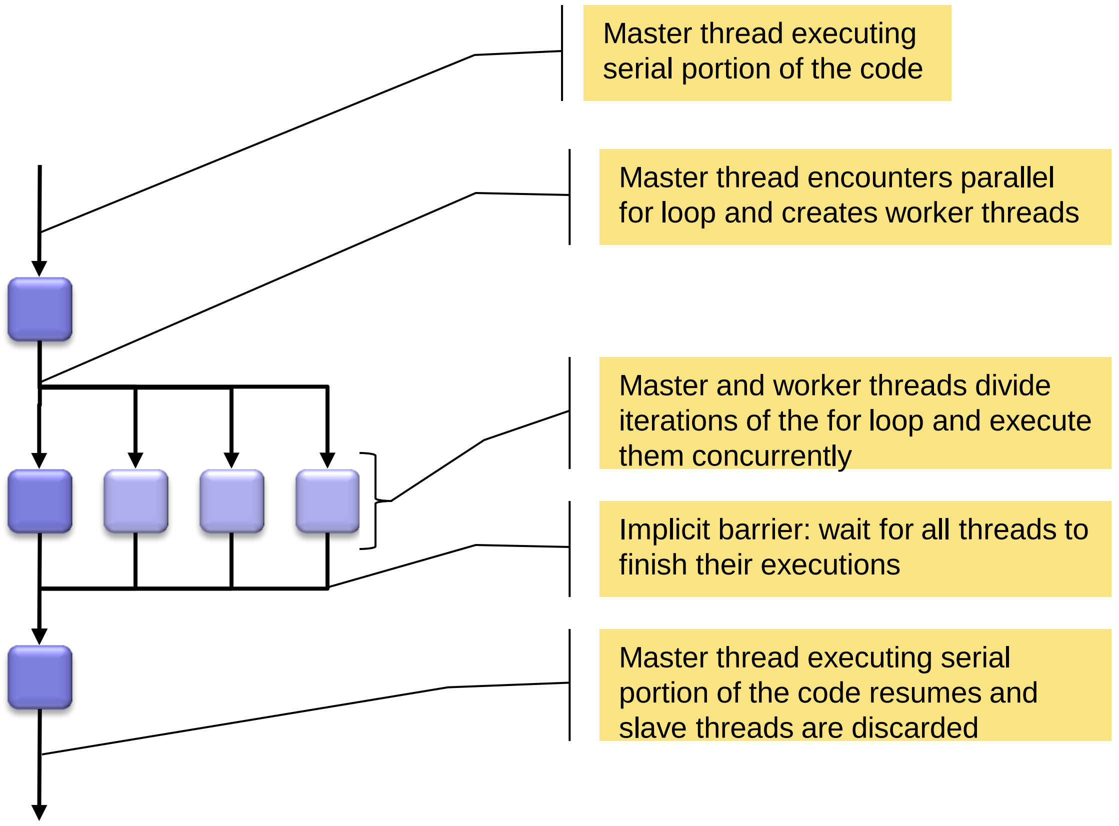
```
#pragma omp parallel private (p)
{
  p=5;
  #pragma omp for
     for (i=0; i<24; i++)
         x[i]=y[i]+p*(i+3)
      …
  …
} /* omp end parallel */
```





Simple Loop Parallelization

```
#pragma omp parallel for
     for (i=0; i<n; i++)
        z( i) = a*x(i)+y
```


Master thread executing serial portion of the code
Master thread encounters parallel for loop and creates worker threads
Master and worker threads divide iterations of the for loop and execute them concurrently
Implicit barrier: wait for all threads to finish their executions
Master thread executing serial portion of the code resumes and slave threads are discarded

Example: OpenMP work sharing Constructs

```
#include <omp.h>
#define N     16
main ()
{
int i, chunk;
float a[N], b[N], c[N];
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = 4;
printf("a[i]  +  b[i]  =  c[i] \n");
#pragma omp parallel shared(a,b,c,chunk) private(i)
  {
  #pragma omp for schedule(dynamic,chunk) nowait
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];
  }  /* end of parallel section */
for (i=0; i < N; i++)
    printf(" %f   +   %f   =   %f  \n",a[i],b[i],c[i]);
}
```

Initializing the vectors a[i], b[i]

Instructing the runtime environment that a,b,c,chunk are shared variables and I is a private variable

The nowait ensures that the child threads donot synchronize once their work is completed

Load balancing the threads using a DYNAMIC policy where array is divided into chunks of 4 and assigned to the threads

DEMO : Work Sharing Constructs Shared / Private / Schedule

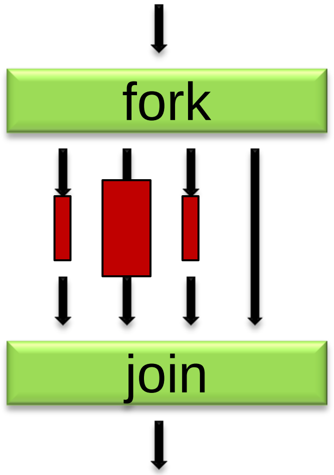
- Vector addition problem to be used
- Two vectors a[i] + b[i] = c[i]

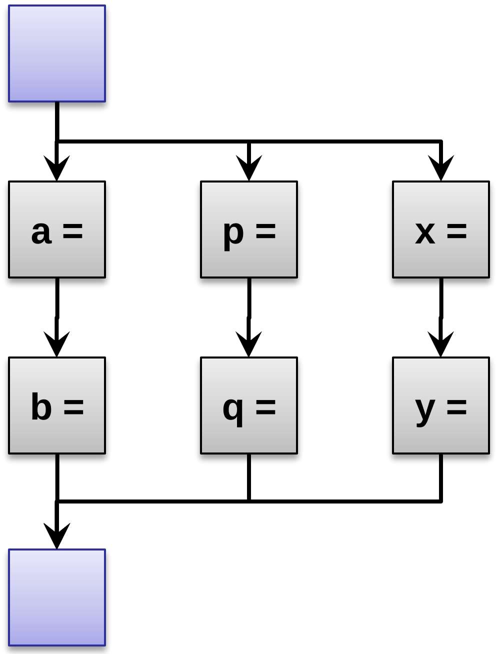
```
     a[i]      +      b[i]      =     c[i]
 0.000000    +  0.000000    =  0.000000
 1.000000    +  1.000000    =  2.000000
 2.000000    +  2.000000    =  4.000000
 3.000000    +  3.000000    =  6.000000
 4.000000    +  4.000000    =  8.000000
 5.000000    +  5.000000    =  10.000000
 6.000000    +  6.000000    =  12.000000
 7.000000    +  7.000000    =  14.000000
 8.000000    +  8.000000    =  16.000000
 9.000000    +  9.000000    =  18.000000
 10.000000   +  10.000000   =  20.000000
 11.000000   +  11.000000    =  22.000000
 12.000000   +  12.000000   =  24.000000
 13.000000   +  13.000000   =  26.000000
 14.000000   +  14.000000   =  28.000000
 15.000000   +  15.000000   =  30.000000
```

OpenMP *sections* directive

- *sections* directive is a non iterative work sharing construct.
- Independent *section* of code are nested within a *section**s*** directive
- It specifies enclosed *section* of codes between different threads
- Code enclosed within a *section* directive is executed once by a thread in the team

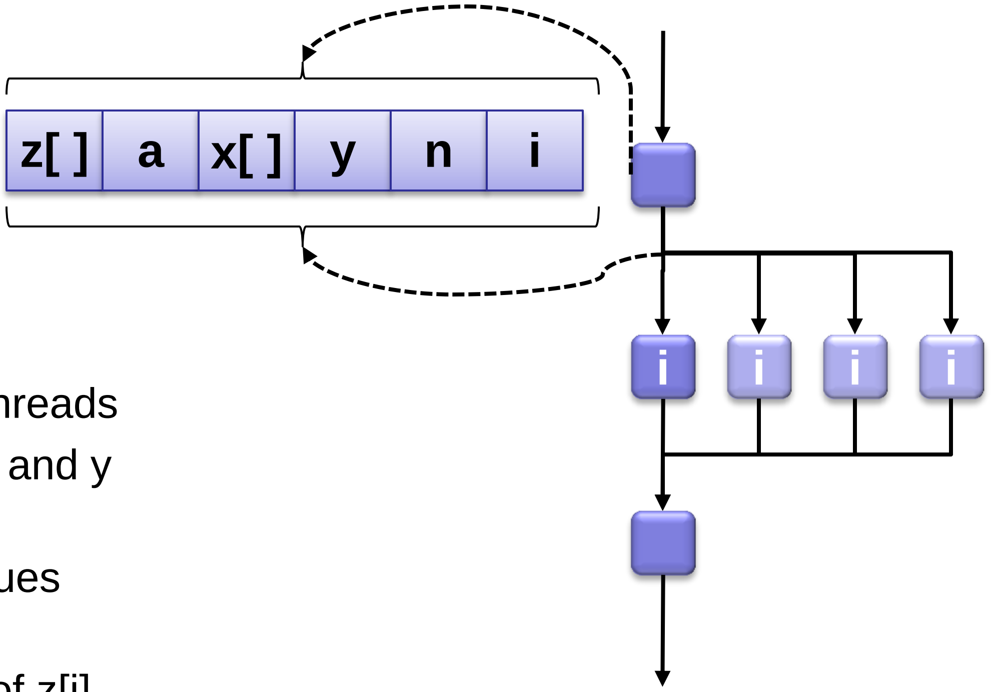
```
#pragma omp parallel private(p)
{
#pragma omp sections
{{  a=…;
   b=…;}
 #pragma omp section
 {  p=…;
   q=…;}
 #pragma omp section
 {  x=…;
   y=…;}
 } /* omp end sections */
} /* omp end parallel */
```





Understanding variables in OpenMP

```
#pragma omp parallel for
     for (i=0; i<n; i++)
        z[i] = a*x[i]+y
```



- Shared variable z is modified by multiple threads
- Each iteration reads the scalar variables a and y and the array element x[i]
- a,y,x can be read concurrently as their values remain unchanged.
- Each iteration writes to a distinct element of z[i] over the index range. Hence write operations can be carried out concurrently with each iteration writing to a distinct array index and memory location
- The parallel *for* directive in OpenMP ensure that the for loop index value (i in this case) is private to each thread.

Example : OpenMP Sections

```
#include <omp.h>
#define N  16
main (){
int i;
float a[N], b[N], c[N], d[N];
for (i=0; i < N; i++)
   a[i] = b[i] = i * 1.5;
#pragma omp parallel shared(a,b,c,d) private(i)
  {
  #pragma omp sections nowait
    {
    #pragma omp section
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];
    #pragma omp section
    for (i=0; i < N; i++)
      d[i] = a[i] * b[i];
    }  /* end of sections */
  }  /* end of parallel section */
…
```

Sections construct that encloses the section calls

Section : that computes the sum of the 2 vectors

Section : that computes the product of the 2 vectors

DEMO : OpenMP Sections

```
[user@addishpc]$ ./sections
 a[i]          b[i]          a[i]+b[i]       a[i]*b[i]
 0.000000      0.000000      0.000000        0.000000
 1.500000      1.500000      3.000000        2.250000
 3.000000      3.000000      6.000000        9.000000
 4.500000      4.500000      9.000000        20.250000
 6.000000      6.000000      12.000000       36.000000
 7.500000      7.500000      15.000000       56.250000
 9.000000      9.000000      18.000000       81.000000
 10.500000     10.500000     21.000000       110.250000
 12.000000     12.000000     24.000000       144.000000
 13.500000     13.500000     27.000000       182.250000
 15.000000     15.000000     30.000000       225.000000
 16.500000     16.500000     33.000000       272.250000
 18.000000     18.000000     36.000000       324.000000
 19.500000     19.500000     39.000000       380.250000
 21.000000     21.000000     42.000000       441.000000
 22.500000     22.500000     45.000000       506.250000
```

Topics

- Overview of OpenMP
- Basic Constructs
- Shared Data
- Parallel Flow Control
- Synchronization
- Reduction
- Synopsis of Commands

Synchronization

- “communication” mainly through read write operations on shared variables
- Synchronization defines the mechanisms that help in coordinating execution of multiple threads (that use a shared context) in a parallel program.
- Without synchronization, multiple threads accessing shared memory location may cause conflicts by :
  - Simultaneously attempting to modify the same location
  - One thread attempting to read a memory location while another thread is updating the same location.
- Synchronization helps by providing explicit coordination between multiple threads.
- Two main forms of synchronization :
  - Implicit event synchronization
  - Explicit synchronization – critical, master directives in OpenMP

Basic Types of Synchronization

- Explicit Synchronization via mutual exclusion
  - Controls access to the shared variable by providing a thread exclusive access to the memory location for the duration of its construct.
  - Requiring multiple threads to acquiring access to a shared variable before modifying the memory location helps ensure integrity of the shared variable.
  - Critical directive of OpenMP provides mutual exclusion
- Event Synchronization
  - Signals occurrence of an event across multiple threads.
  - Barrier directives in OpenMP provide the simplest form of event synchronization
  - The barrier directive defines a point in a parallel program where each thread waits for all other threads to arrive. This helps to ensure that all threads have executed the same code in parallel upto the barrier.
  - Once all threads arrive at the point, the threads can continue execution past the barrier.
- Additional synchronization mechanisms available in OpenMP

OpenMP Synchronization : *master*

- The *master* directive in OpenMP marks a block of code that gets executed on a single thread.
- The rest of the treads in the group ignore the portion of code marked by the master directive
- Example

```
#pragma omp master
    structured block
```

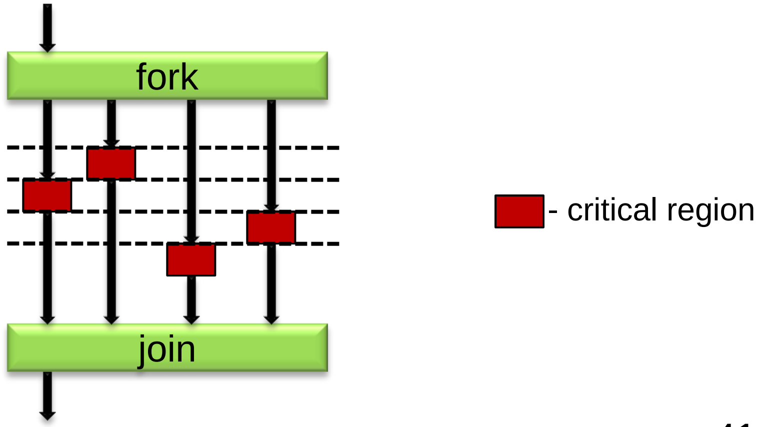
Race Condition :

Two asynchronous threads access the same shared variable and atleast one modifies the variable and the sequence of operations is undefined . Result of these asynchronous operations depends on detailed timing of the individual threads of the group.

OpenMP *critical* directive : Explicit Synchronization

- Race conditions can be avoided by controlling access to shared variables by allowing threads to have exclusive access to the variables
- Exclusive access to shared variables allows the thread to *atomically* perform read, modify and update operations on the variable.
- *Mutual exclusion* synchronization is provided by the *critical* directive of OpenMP
- Code block within the *critical region* defined by critical /end critical directives can be executed only by one thread at a time.
- Other threads in the group must wait until the current thread exits the critical region. Thus only one thread can manipulate values in the critical region.

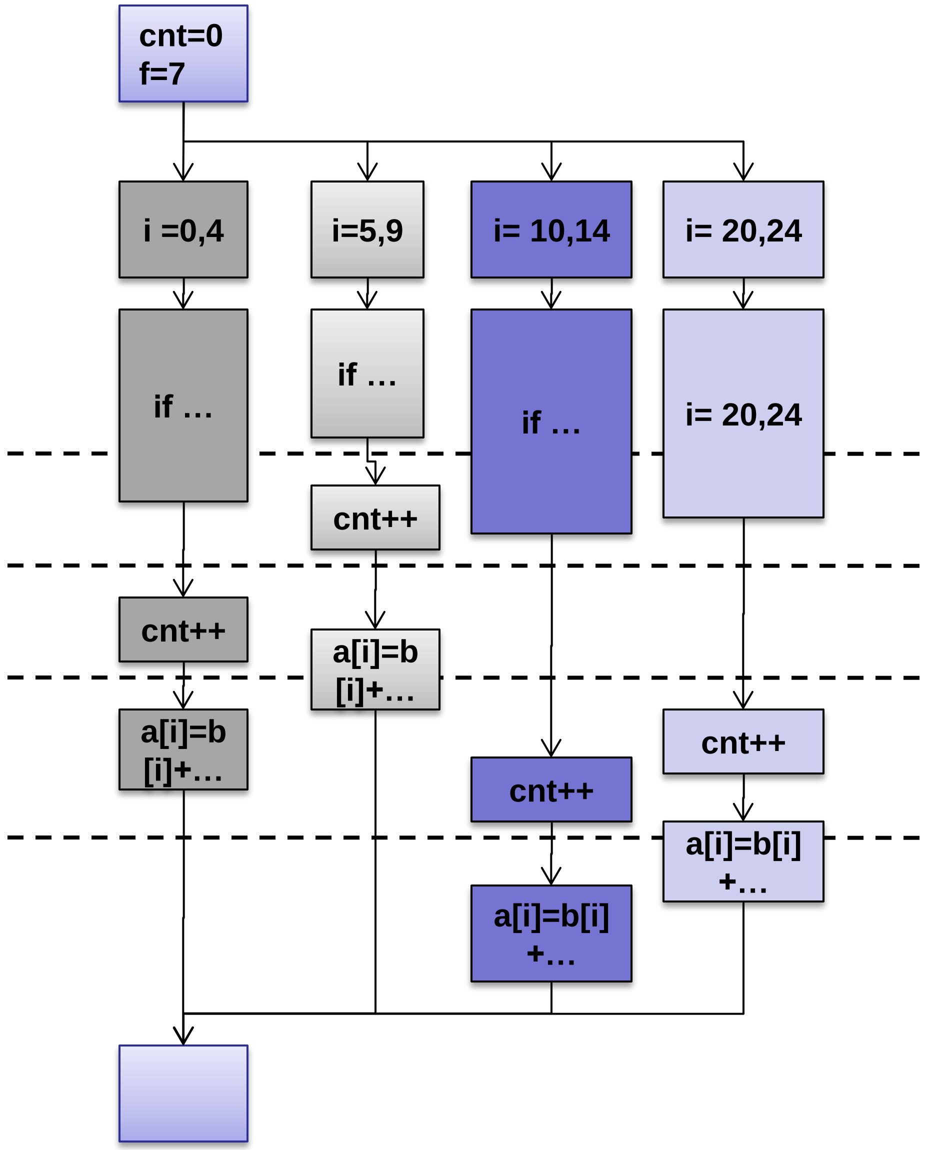
```
int x
x=0;
#pragma omp parallel shared(x)
{
  #pragma omp critical
      x = 2*x + 1;
} /* omp end parallel */
```



Simple Example : *critical*

```
cnt = 0;
f = 7;
#pragma omp parallel
{
   #pragma omp for
      for (i=0;i<20;i++){
         if(b[i] == 0){

            #pragma omp critical
               cnt ++;
         } /* end if */
         a[i]=b[i]+f*(i+1);
      } /* end for */
} /* omp end parallel */
```



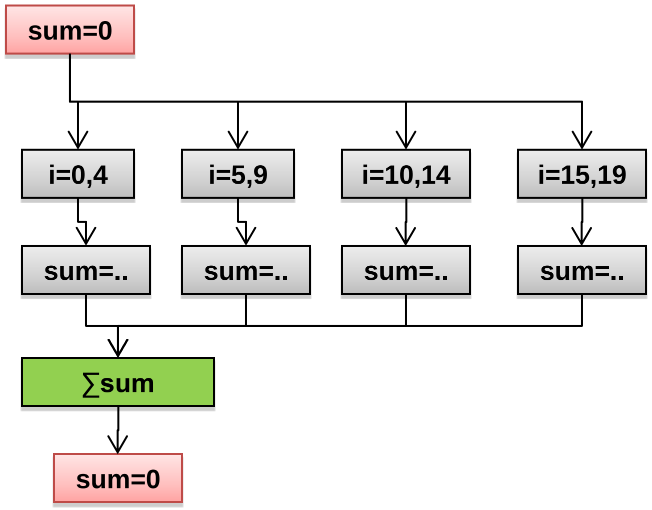
Topics

- Introduction
- Overview of OpenMP
- Basic Constructs
- Shared Data
- Parallel Flow Control
- Synchronization
- Reduction
- Synopsis of Commands

OpenMP : Reduction

- performs reduction on *shared variables* in list based on the operator provided.
- for C/C++ operator can be any one of :
  - +, *, -, ^, |, ||, & or &&
  - At the end of a reduction, the shared variable contains the result obtained upon combination of the list of variables processed using the operator specified.

```
sum = 0.0
#pragma omp parallel for reduction(+:sum)
  for (i=0; i < 20; i++)
    sum = sum + (a[i] * b[i]);
```



Example: Reduction

```
#include <omp.h>
main ()  {
int   i, n, chunk;
float a[16], b[16], result;
n = 16;
chunk = 4;
result = 0.0;
for (i=0; i < n; i++)
  {
     a[i] = i * 1.0;
     b[i] = i * 2.0;
  }
#pragma omp parallel for default(shared) private(i)  \
     schedule(static,chunk) reduction(+:result)
  for (i=0; i < n; i++)
        result = result + (a[i] * b[i]);
printf("Final result= %f\n",result);
}
```

Reduction example with summation where the result of the reduction operation stores the dotproduct of two vectors

$\sum a[i]*b[i]$

Demo: Dot Product using Reduction

```
[user@addishpc]$ ./reduction
 a[i]          b[i]          a[i]*b[i]
 0.000000   0.000000     0.000000
 1.000000   2.000000     2.000000
 2.000000   4.000000     8.000000
 3.000000   6.000000     18.000000
 4.000000   8.000000     32.000000
 5.000000   10.000000     50.000000
 6.000000   12.000000     72.000000
 7.000000   14.000000     98.000000
 8.000000   16.000000     128.000000
 9.000000   18.000000     162.000000
 10.000000   20.000000     200.000000
 11.000000   22.000000     242.000000
 12.000000   24.000000     288.000000
 13.000000   26.000000     338.000000
 14.000000   28.000000     392.000000
 15.000000   30.000000     450.000000
Final result= 2480.000000
```

Topics

- Overview of OpenMP
- Basic Constructs
- Shared Data
- Parallel Flow Control
- Synchronization
- Reduction
- Synopsis of Commands

Synopsis of Commands

- How to invoke OpenMP runtime systems *#pragma omp parallel*
- The interplay between OpenMP environment variables and runtime system (*omp_get_num_threads(), omp_get_thread_num()*)
- Shared data directives such as *shared, private* and *reduction*
- Basic flow control using *sections, for*
- Fundamentals of synchronization using *critical* directive and critical section.
- And directives used for the OpenMP programming part of the problem set.