OpenMP

Introduction

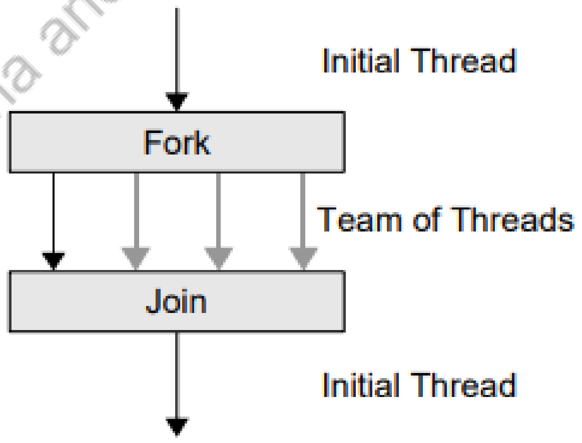
- The OpenMP **Application Programming Interface** (API) was developed to enable portable shared memory parallel programming
- The API is designed to permit an ***incremental*** approach to parallelizing an existing code, in which portions of a program are parallelized, possibly in successive steps
  - Contrast to the ***all-or-nothing*** conversion of an entire program in a single step that is typically required by other parallel programming paradigms

The Idea of OpenMP

- A thread is a **runtime entity** that is able to independently execute a stream of instructions
- The operating system creates a *process* to execute a program
  - It will allocate some resources to that process, including pages of memory and registers for holding values of objects
- If **multiple threads collaborate** to execute a program, they will share the resources, including the address space, of the corresponding process
- The individual threads need just a few resources of their own:
  - A **program counter** and an **area in memory** to save variables that are specific to it (including registers and a stack)

The Idea of OpenMP

- Multiple threads may be executed on a single processor or core via context switches;
  - They may be interleaved via simultaneous multithreading
- Threads running simultaneously on multiple processors or cores may work concurrently to execute a parallel program
- It supports the so-called **fork-join programming model**



The Idea of OpenMP

- The program starts as a single thread of execution, just like a sequential program
- The thread that executes this code is referred to as the *initial thread*
- Whenever an OpenMP parallel construct is encountered by a thread while it is executing the program, it *creates a team of threads* (this is the fork)
- It becomes the master of the team, and collaborates with the other members of the team to execute the code dynamically enclosed by the construct
- At the end of the construct, only the original thread, or master of the team, continues; all others terminate (this is the join)
- Each portion of code enclosed by a parallel construct is called a *parallel region*

The Feature Set

- The OpenMP API comprises a set of *compiler directives*, *runtime library routines*, and *environment variables* to specify shared-memory parallelism
  - **Directive:** A statement written in the source code of a program that lets the programmer instruct the compiler to perform a specific operation within the compilation phase.
  - **Environmental variables** define the characteristics of a specific environment
  - **Library routines** (API) that helps to do some task

- An OpenMP directive is a specially formatted comment or pragma that generally applies to the executable code immediately following it in the program
  - *A directive or OpenMP routine generally affects only those threads that encounter it*

- Many of the directives are applied to a *structured block of code*
  - A sequence of executable statements with a single entry at the top and a single exit at the bottom

The Feature Set

- OpenMP provides means for the user to:
  - Create teams of threads for parallel execution
  - Specify how to share work among the members of a team
  - Declare both shared and private variables
  - Synchronize threads and enable them to perform certain operations exclusively (i.e., without interference by other threads)

Creating Teams of Threads

- A team of threads is created to execute the code in a parallel region of an OpenMP program
  - To accomplish this, the programmer simply specifies the parallel region by inserting a parallel directive immediately before the code that is to be executed in parallel to mark its start
- At the end of a parallel region is an **implicit barrier synchronization**
  - This means that no thread can progress until all other threads in the team have reached that point in the program
- If a team of threads executing a parallel region encounters another parallel directive, each thread in the current team creates a new team of threads and becomes its master
- Nesting enables the realization of **multilevel parallel programs**

Sharing Work among Threads

- If the programmer does not specify how the work in a parallel region is to be shared among the executing threads, they will each **redundantly** execute all of the code
    - *This approach does not speed up the program*
- The OpenMP work-sharing directives are provided for the programmer to state how the computation in a structured block of code is to be distributed among the threads
- Unless explicitly overridden by the programmer, an implicit barrier synchronization also exists at the end of a work-sharing construct

Sharing Work among Threads

- Probably the most common work-sharing approach is to distribute the work **for (C/C++) loop** among the threads in a team
- To accomplish this, the programmer inserts the appropriate directive immediately before each loop within a parallel region that is to be shared among threads
- *Work-sharing directives cannot be applied to all kinds of loops that occur in C/C++ code*
- Many programs, especially scientific applications, spend a large fraction of their time in loops performing calculations on array elements and so this strategy is widely applicable and often very effective

Sharing Work among Threads

- All OpenMP strategies for sharing the work in loops assign one or more disjoint sets of iterations to each thread
- The programmer may specify the method used to **partition the iteration set**
- *The most straightforward strategy assigns one* ***contiguous chunk*** *of iterations to each thread*
- *More complicated strategies include* ***dynamically computing the next chunk*** *of iterations for a thread*
- If the programmer does not provide a strategy, then an implementation-defined default will be used.

Sharing Work among Threads

- ***It must be possible to determine the number of iterations in the loop upon entry, and this number may not change while the loop is executing***
  - ***While construct****, for example, may not satisfy this condition*
- Furthermore, a loop is suitable for sharing among threads only if its **iterations are independent**
- By this, we mean that the order in which the iterations are performed has no bearing on the outcome

Sharing Work among Threads

- If giving distinct pieces of work to the individual threads
- This approach is suitable when independent computations are to be performed and the order in which they are carried out is irrelevant
- It is straightforward to specify this by using the corresponding OpenMP directive
- The programmer must ensure that the computations can truly be executed in parallel
- *It is also possible to specify that just one thread should execute a block of code in a parallel region*

The OpenMP Memory Model

- OpenMP is based on the shared-memory model; hence, by default, data is shared among the threads and is visible to all of them
- Sometimes, however, one needs variables that have thread-specific values
- When each thread has its own copy of a variable, so that it may potentially have a different value for each of them, we say that the variable is private
  - For example, when a team of threads executes a parallel loop, each thread needs its own value of the iteration variable.
- The use of private variables can be beneficial in several ways:
  - They can reduce the frequency of updates to shared memory
  - Thus, they may help avoid hot spots, or competition for access to certain memory locations, which can be expensive if large numbers of threads are involved.

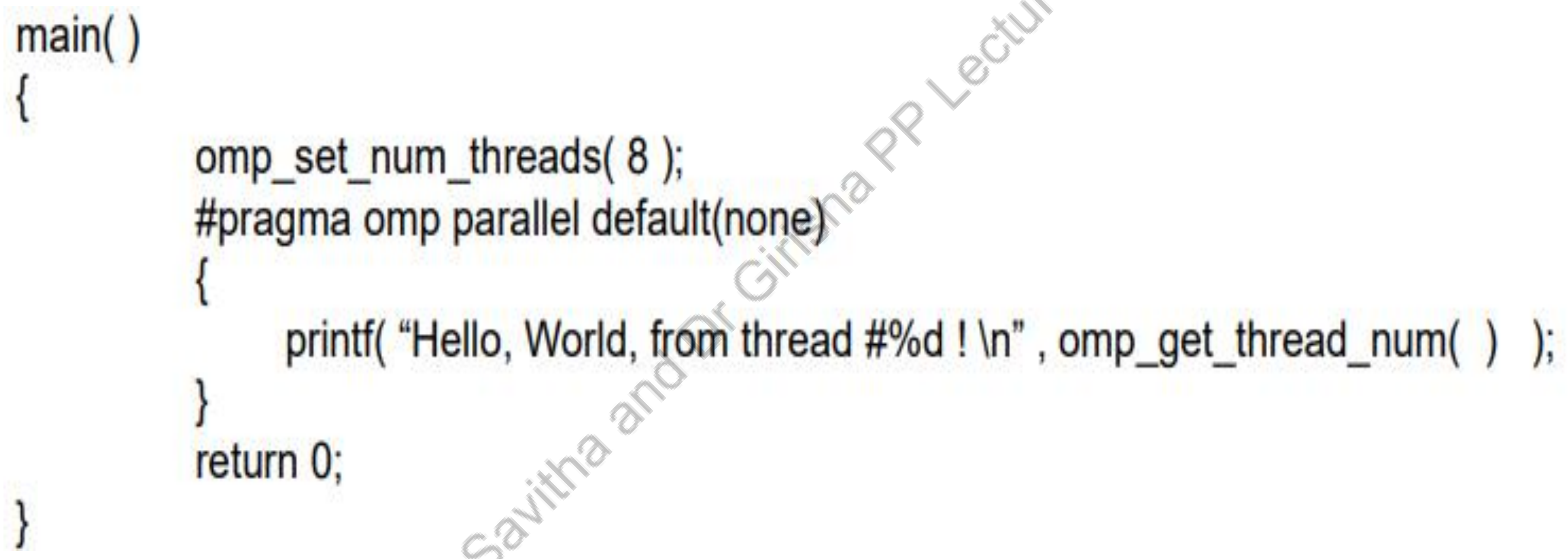
The OpenMP Memory Model

- Threads need a place to store their private data at run time
- For this, each thread has its own special region in memory known as ***the thread stack***
- A ***flush operation*** makes sure that the thread calling it has the same values for shared data objects as does main memory
- Hence, new values of any shared objects updated by that thread are written back to shared memory, and the thread gets any new values produced by other threads for the shared data it reads

OpenMP Code Structure

```
#include <stdlib.h>
#include<stdio.h>
 #include "omp.h“
 int main()
{
        #pragma omp parallel
        {
                int ID = omp_get_thread_num();
                printf("Hello (%d)\n", ID);
                 printf(" world (%d)\n", ID);
         }
 }
```

A sample OpenMP program



```
main( )
{
        omp_set_num_threads( 8 );
        #pragma omp parallel default(none)
        {
            printf( “Hello, World, from thread #%d ! \n” , omp_get_thread_num(  )   );
        }
        return 0;
}
```

A sample OpenMP program

First Run

```
Hello, World, from thread #6 !
Hello, World, from thread #1 !
Hello, World, from thread #7 !
Hello, World, from thread #5 !
Hello, World, from thread #4 !
Hello, World, from thread #3 !
Hello, World, from thread #2 !
Hello, World, from thread #0 !
```

Second Run

```
Hello, World, from thread #0 !
Hello, World, from thread #7 !
Hello, World, from thread #4 !
Hello, World, from thread #6 !
Hello, World, from thread #1 !
Hello, World, from thread #3 !
Hello, World, from thread #5 !
Hello, World, from thread #2 !
```

Third Run

```
Hello, World, from thread #2 !
Hello, World, from thread #5 !
Hello, World, from thread #0 !
Hello, World, from thread #7 !
Hello, World, from thread #1 !
Hello, World, from thread #3 !
Hello, World, from thread #4 !
Hello, World, from thread #6 !
```

Fourth Run

```
Hello, World, from thread #1 !
Hello, World, from thread #3 !
Hello, World, from thread #5 !
Hello, World, from thread #2 !
Hello, World, from thread #4 !
Hello, World, from thread #7 !
Hello, World, from thread #6 !
Hello, World, from thread #0 !
```

Thread Synchronization

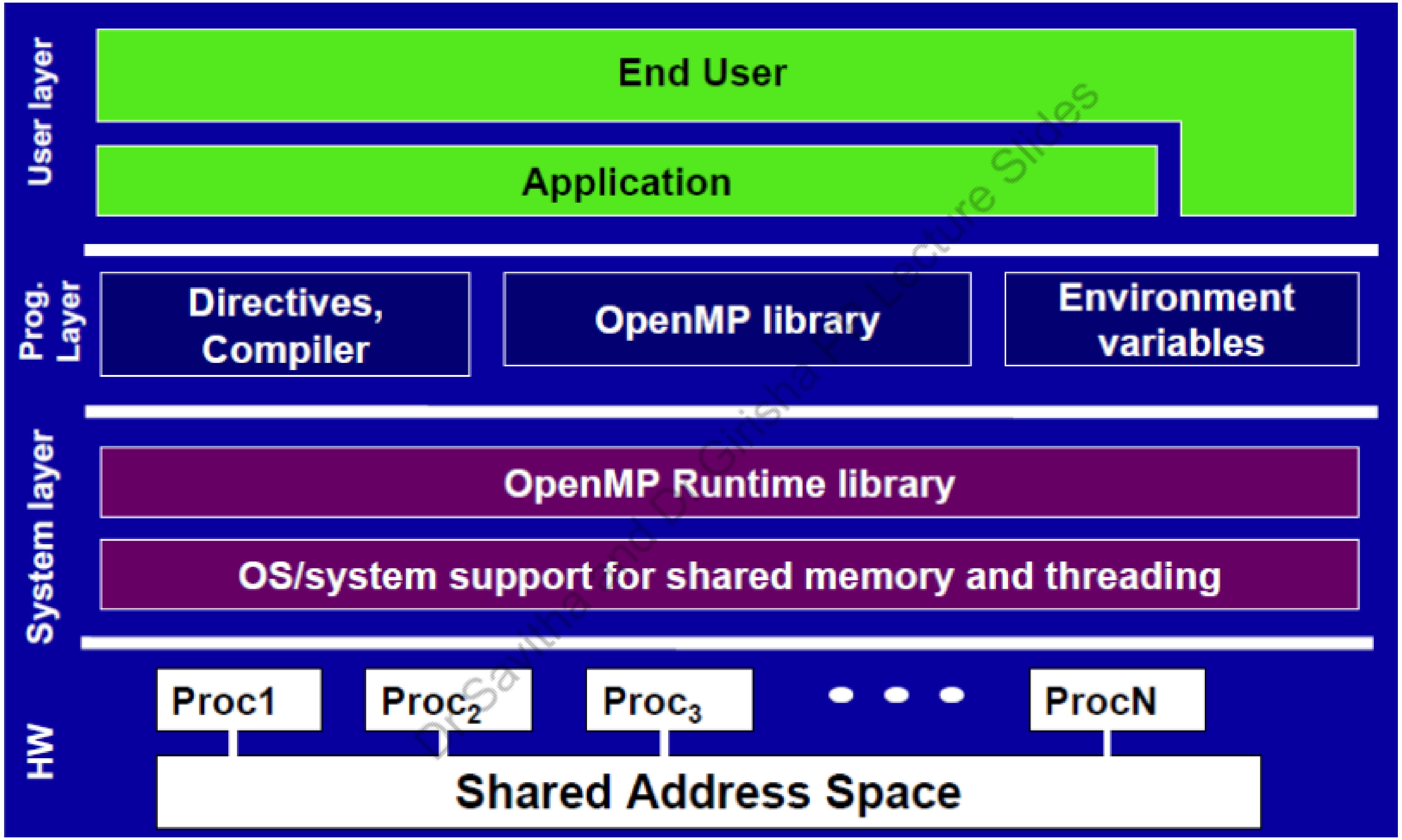
- *Why synchronization?:*
  - *Synchronizing, or coordinating the actions of threads is sometimes necessary in order to ensure the proper ordering of their accesses to shared data and to prevent data corruption*
- By default, OpenMP gets threads to wait at the end of a *worksharing construct or parallel region* until all threads in the team executing it have finished their portion of the work. Only then can they proceed. **This is known as a barrier**
- Sometimes a programmer may need to ensure that only one thread at a time works on a piece of code
- *Synchronization points are those places in the code where synchronization has been specified, either explicitly or implicitly*

OpenMP Programming Styles

- OpenMP encourages structured parallel programming and relies heavily on *distributing the work in loops among threads*. But sometimes the amount of loop-level parallelism in an application is limited
- *One can also write OpenMP programs that do not rely on work-sharing directives but rather assign work explicitly to different threads using their thread numbers.*
- This approach can lead to highly efficient code
  - However, the programmer must then insert synchronization manually to ensure that accesses to shared data are correctly controlled
  - Those approaches that require manual assignment of work to threads and that need explicit synchronization are often called ***"low-level programming"***

What is OpenMP?

- **Open specifications for Multi Processing**
- An **Application Program Interface** (API) that is used to explicitly direct multi-threaded, shared memory parallelism
- API components:
  - Compiler directives
  - Runtime library routines
  - Environment variables
- Portability
  - API is specified for C/C++ and Fortran
  - Implementations on almost all platforms including Unix/Linux and Windows
- Standardization
  - Jointly defined and endorsed by major computer hardware and software vendors
  - Possibility to become ANSI standard


User layer
End User
Application
Prog. Layer
Directives, Compiler
OpenMP library
Environment variables
System layer
OpenMP Runtime library
OS/system support for shared memory and threading
HW
Proc1
Proc$_2$
Proc$_3$
• • •
ProcN
Shared Address Space

Threads & Process

- A process is an instance of a computer program that is being executed. It contains the program code and its current activity.
- A thread of execution is the smallest unit of processing that can be scheduled by an operating system
- Differences between threads and processes:
  - **A thread is contained inside a process**. Multiple threads can exist within the same process and share resources such as memory. The threads of a process share the latter's instructions (code) and its context (values that its variables reference at any given moment)
  - Different processes do not share these resources.

Threads & Process

- A process contains all the information needed to execute the program
  - Process ID
  - Program code
  - Data on run time stack
  - Global data
  - Data on heap
- Each process has its own address space
- In multitasking, processes are given time slices in a round robin fashion
- If computer resources are assigned to another process, the status of the present process has to be saved, in order that the execution of the suspended process can be resumed at a later time.

Threads & Process

- Each process consists of multiple independent instruction streams (or threads) that are assigned computer resources by some scheduling procedure
- Threads of a process share the address space of this process
  - Global variables and all dynamically allocated data objects are accessible by all threads of a process
- Each thread has its own run-time stack, register, program counter
- Threads can communicate by reading/writing variables in the common address space

OpenMP Programming Model

- Shared memory, thread-based parallelism
  - OpenMP is based on the existence of multiple threads in the shared memory programming paradigm
  - A shared memory process consists of multiple threads
- Explicit Parallelism
  - Programmer has full control over parallelization. OpenMP is not an automatic parallel programming model
- Compiler directive based
  - Most OpenMP parallelism is specified through the use of compiler directives which are embedded in the source code

What OpenMP Isn’t

- OpenMP doesn’t check for **data dependencies**, **data conflicts**, **deadlocks**, or **race conditions**. You are responsible for avoiding those yourself
- OpenMP doesn’t check for **non-conforming** code sequences
- OpenMP doesn’t **guarantee identical behavior** across vendors or hardware, or even between multiple runs on the same vendor’s hardware
- OpenMP doesn’t guarantee the **order in which threads execute**, just that they do execute
- OpenMP is not **overhead-free**
- OpenMP does not prevent you from writing code that triggers **cache performance problems**

OpenMP: parallel regions

- A parallel region within a program is specified as

```
#pragma omp parallel [clause [[,] clause] …]
      Structured-block
```

- A team of threads is formed
- Thread that encountered the omp parallel directive becomes the **master thread** within this team
- **The structured-block is executed by every thread in the team**.
- At the end, there is an implicit **barrier**
- Only after **all threads have finished, the threads created by this directive are terminated and only the master resumes execution**
- A parallel region might be refined by a list of clause

- Each thread in the team is assigned a unique thread number (also referred to as the “thread id”) to identify it.
- They range from zero (for the master thread) up to one less than the number of threads within the team, and they can be accessed by the programmer.
- Although the parallel region is executed by all threads in the team, each thread is allowed to follow a different path of execution.

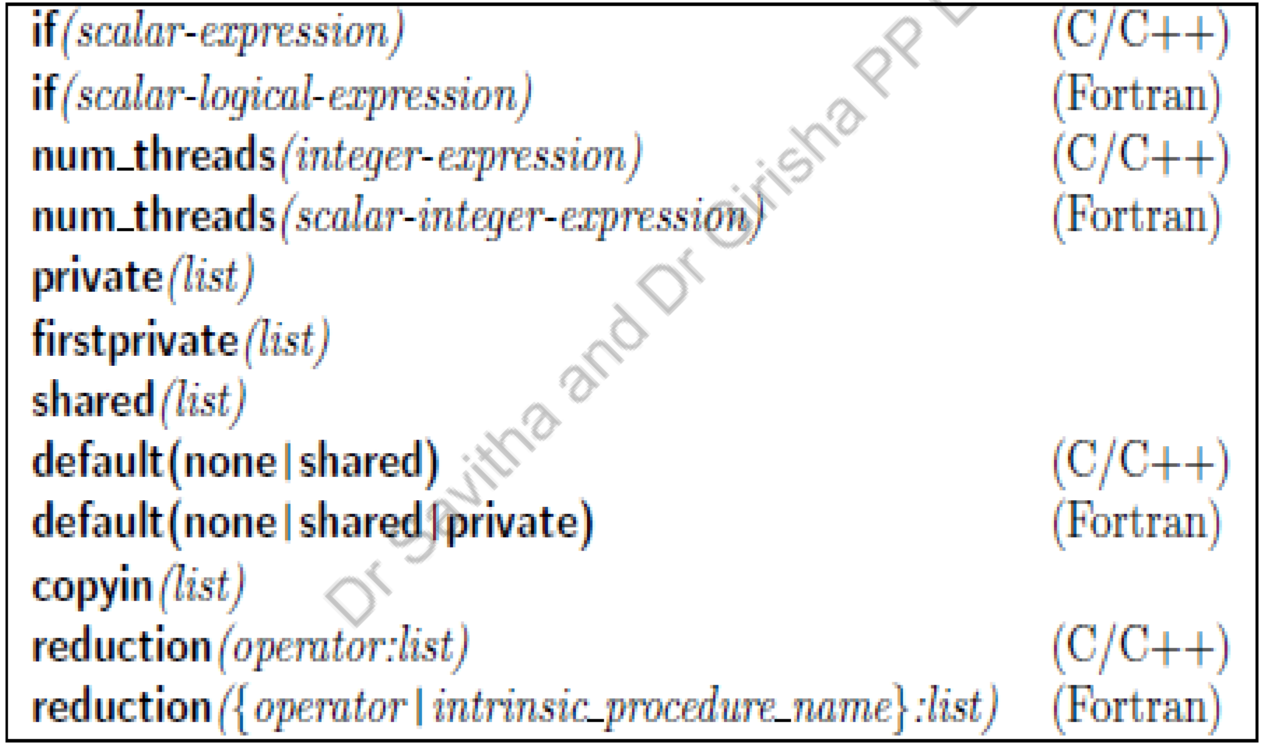
- Each thread will execute all code in the parallel region, so that each thread will perform the first print statement.
- However, only one thread will actually execute the second print statement (assuming there are at least three threads in the team), since we used the thread number to control its execution.

```
#pragma omp parallel
   {
      printf("The parallel region is executed by thread %d\n",
        omp_get_thread_num());

      if ( omp_get_thread_num() == 2 ) {
        printf("  Thread %d does things differently\n",
              omp_get_thread_num());
      }
   } /*-- End of parallel region --*/
```

Output:

```
The parallel region is executed by thread 0
The parallel region is executed by thread 3
The parallel region is executed by thread 2
  Thread 2 does things differently
The parallel region is executed by thread 1
```

Clauses supported by the parallel construct



if *(scalar-expression)*	(C/C++)
if *(scalar-logical-expression)*	(Fortran)
num_threads *(integer-expression)*	(C/C++)
num_threads *(scalar-integer-expression)*	(Fortran)
private *(list)*	
firstprivate *(list)*	
shared *(list)*	
default(none | shared)	(C/C++)
default(none | shared | private)	(Fortran)
copyin *(list)*	
reduction *(operator:list)*	(C/C++)
reduction *({operator | intrinsic_procedure_name}:list)*	(Fortran)

There are several restrictions on the parallel construct and its clauses:

- A program that branches into or out of a parallel region is nonconforming.
  - In other words, if a program does so, then it is *illegal*, and the behavior is undefined.
- A program must not depend on any ordering of the evaluations of the clauses of the parallel directive or on any side effects of the evaluations of the clauses.
- At most one ***if*** clause can appear on the directive.
- At most one ***num_threads*** clause can appear on the directive. The expression for the clause must evaluate to a positive integer value.
- In C++ there is an additional constraint. A throw inside a parallel region must cause execution to resume within the same parallel region, *and* it must be caught by the same thread that threw the exception.

Active **parallel region** and an *Inactive* **parallel region**

- A parallel region is active if it is executed by a team of threads consisting of more than one thread.
- If it is executed by one thread only, it has been serialized and is considered to be inactive.
- **Example:** - one can specify that a parallel region be conditionally executed, in order to be sure that it contains enough work for this to be worthwhile.
  - If the condition does not hold at run time, then the parallel region will be inactive.

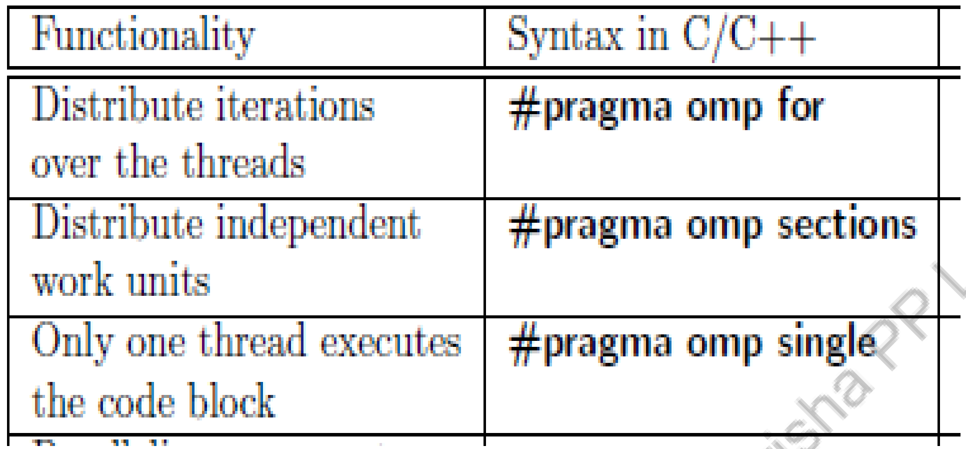
```
#pragma omp parallel if (n > 5) default(none)
```

Here the parallel region is executed only if n is greater than 5.

- A parallel region may also be inactive if it is nested within another parallel region and this feature is either disabled or not provided by the implementation

Sharing the Work among Threads in an OpenMP Program

- C/C++ has three work-sharing constructs.



Functionality	Syntax in C/C++
Distribute iterations over the threads	**#pragma omp for**
Distribute independent work units	**#pragma omp sections**
Only one thread executes the code block	**#pragma omp single**

- Many applications can be parallelized by using just a parallel region and one or more of these constructs, possibly with clauses
- Work-sharing construct, along with its terminating construct where appropriate, specifies a region of code whose work is to be distributed among the executing threads; it also specifies the manner in which the work in the region is to be parceled out.
- A work-sharing region must bind to an active parallel region in order to have an effect

- Since work-sharing directives may occur in procedures that are invoked both from within a parallel region as well as outside of any parallel regions, they may be exploited during some calls and ignored during others

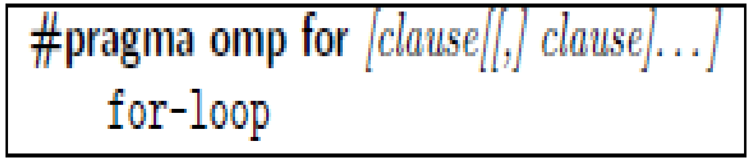
The two main rules regarding work-sharing constructs are as follows:

- Each work-sharing region must be encountered by all threads in a team or by none at all.
- The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in a team.

- A work-sharing construct does not launch new threads and does not have a barrier on entry. By default, threads wait at a barrier at the end of a work-sharing region until the last thread has completed its share of the work.

Loop Construct

- The *loop construct* causes the iterations of the loop immediately following it to be executed in parallel.
- At run time, the loop iterations are distributed across the threads. This is probably the most widely used of the work-sharing features.



```
#pragma omp for [clause[[,] clause]...]
    for-loop
```

Note the lack of curly braces.
These are implied with the construct.

- In C and C++ programs, the use of this construct is limited to those kinds of loops where the number of iterations can be counted;
  - that is, the loop must have an integer counter variable whose value is incremented (or decremented) by a fixed amount at each iteration until some specified upper (or lower) bound is reached

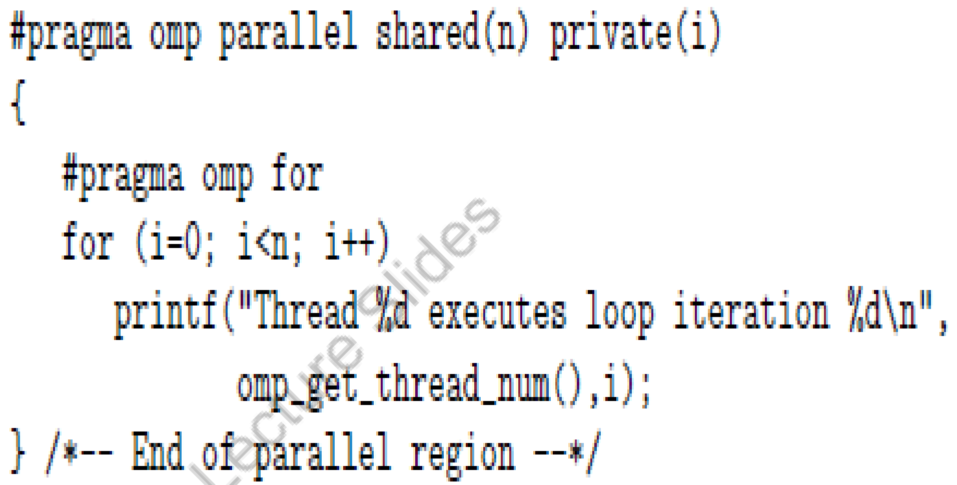
- The loop header must have the general form.

for (init-expr ; var relop b ; incr-expr)

where

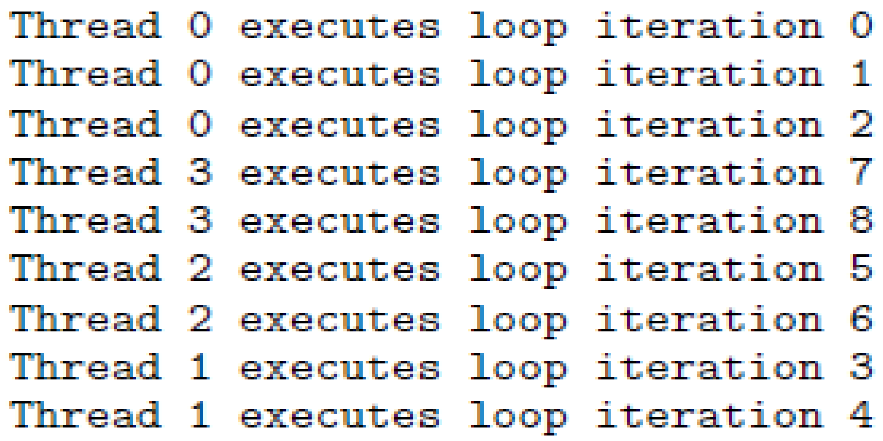
- *init-expr* stands for the initialization of the loop counter var via an integer expression
- b is also an integer expression
- *relop* is one of the following: <, <=, >, >=
- The *incr-expr* is a statement that increments or decrements var by an integer amount using a standard operator (++, –, +=, -=). Alternatively, it may take a form such as *var* = *var* + *incr*.

- A parallel directive is used to define a parallel region and then share its work among threads via the for work sharing directive:
  - the #pragma omp for directive states that iterations of the loop following it will be distributed
  - omp get thread num(), is used obtain and print the number of the executing thread in each iteration
  - Clauses are added to the parallel construct that state which data in the region is shared and which is private.
  - Loop variable i is explicitly declared to be a private variable by the compiler, which means that each thread will have its own copy of i.
  - Unless the programmer takes special action its value is also undefined after the loop has finished



```
#pragma omp parallel shared(n) private(i)
{
   #pragma omp for
   for (i=0; i<n; i++)
      printf("Thread %d executes loop iteration %d\n",
             omp_get_thread_num(),i);
} /*-- End of parallel region --*/
```

Output : The example is executed for $n = 9$ and uses four threads



```
Thread 0 executes loop iteration 0
Thread 0 executes loop iteration 1
Thread 0 executes loop iteration 2
Thread 3 executes loop iteration 7
Thread 3 executes loop iteration 8
Thread 2 executes loop iteration 5
Thread 2 executes loop iteration 6
Thread 1 executes loop iteration 3
Thread 1 executes loop iteration 4
```

Since the total number of iterations is 9 and four threads are used, one thread has to execute the additional iteration. In this case it turns out to be thread 0, the so-called master thread

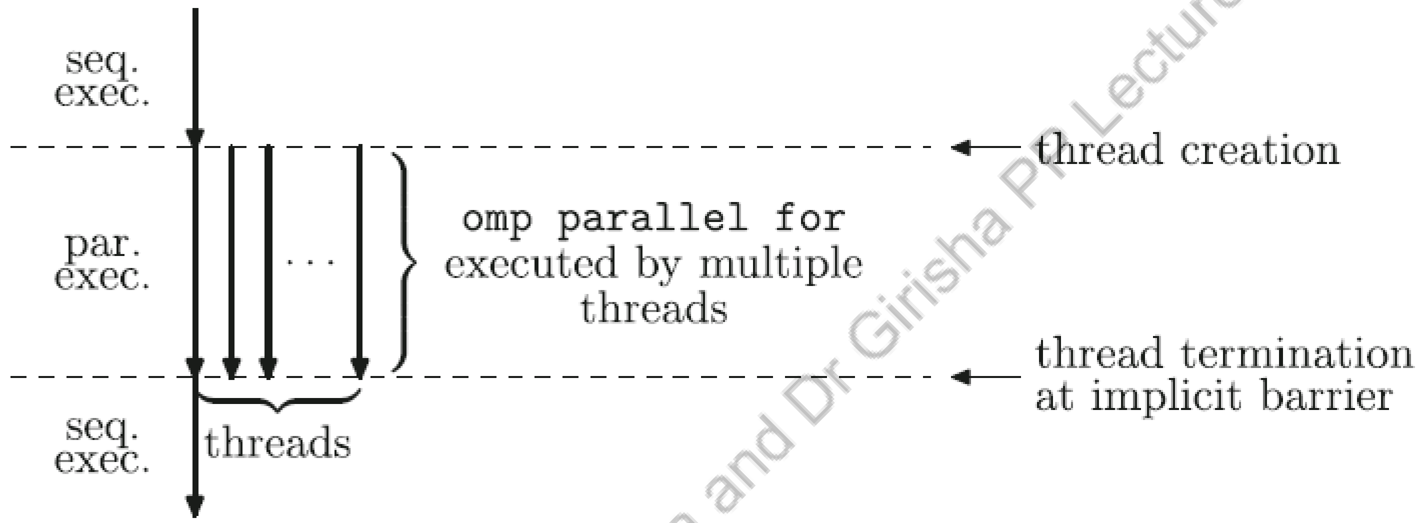

seq. exec.
par. exec.
seq. exec.
threads
. . .
omp parallel for executed by multiple threads
thread creation
thread termination at implicit barrier

Fig. 3.5 Execution of the program for printing out integers as implemented in Listing 3.3

Divide for-loop for parallel sections

```
for (int i=0; i<8; i++) x[i]=0; //run on 4 threads
```

```
#pragma omp parallel
        {
        int numt=omp_get_num_thread();
        int id = omp_get_thread_num(); //id=0, 1, 2, or 3
        for (int i=id; i<8; i +=numt)
                                  x[i]=0;


        }
```

// Assume number of threads=4

Thread 0

```
Id=0;
x[0]=0;
X[4]=0;
```

Thread 1

```
Id=1;
x[1]=0;
X[5]=0;
```

Thread 2

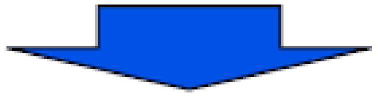
```
Id=2;
x[2]=0;
X[6]=0;
```

Thread 3

```
Id=3;
x[3]=0;
X[7]=0;
```

Use pragma parallel for

```
for (int i=0; i<8; i++) x[i]=0;
```



```
#pragma omp parallel for
{
            for (int i=0; i<8; i++)
                        x[i]=0;
}
```

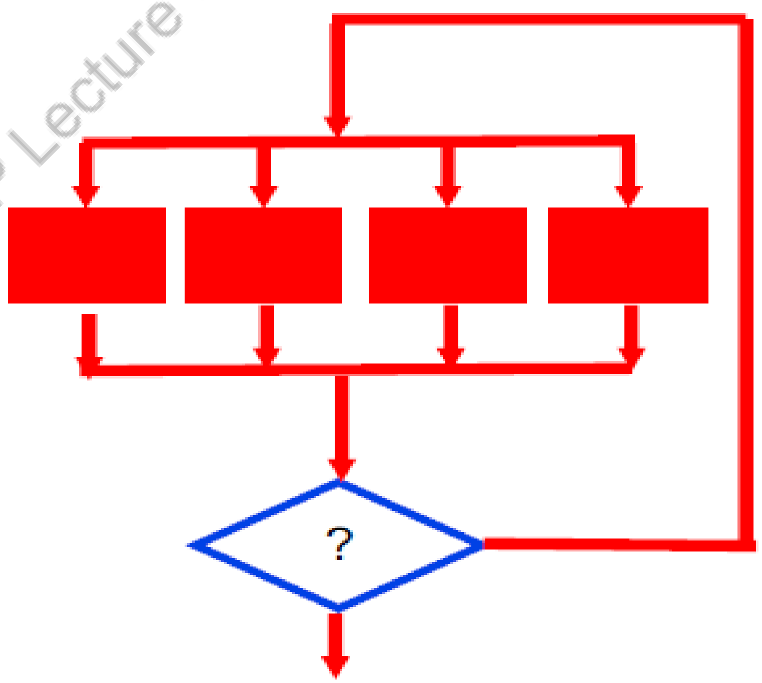
System divides loop iterations to threads

Id=0; **x[0]=0;** X[4]=0;	**Id=1;** **x[1]=0;** X[5]=0;	**Id=2;** **x[2]=0;** X[6]=0;	**Id=3;** **x[3]=0;** X[7]=0;

Programming Model – Parallel Loops

- Requirement for parallel loops
  - No data dependencies (reads/write or write/write pairs) between iterations!
- Preprocessor calculates loop bounds and divide iterations among parallel threads

```
#pragma omp parallel for

 for( i=0; i < 25; i++ )
 {

   printf("Foo");

 }
```



Example

```
for (i=0; i<max; i++) zero[i] = 0;
```

- Breaks *for loop* into chunks, and allocate each to a separate thread
  - e.g. if `max` = 100 with 2 threads:
    assign 0-49 to thread 0, and 50-99 to thread 1
- Must have relatively simple “shape” for an OpenMP-aware compiler to be able to parallelize it
  - Necessary for the run-time system to be able to determine how many of the loop iterations to assign to each thread
- No premature exits from the loop allowed ← **In general, don’t jump outside of any pragma block**
  - i.e. No `break`, `return`, `exit`, `goto` statements

OpenMP: parallel loops

```
x[ 0 ] = 0.;
y[ 0 ] *= 2.;
for( int i = 1; i < N; i++ )
{
            x[ i ] = x[ i-1 ] + 1.;
            y[ i ] *= 2.;
}
```

Because of the loop dependency, this whole thing is not parallelizable:

```
x[ 0 ] = 0.;
for( int i = 1; i < N; i++ )
{
            x[ i ] = x[ i-1 ] + 1.;
}

#pragma omp parallel for shared(y)
for( int i = 0; i < N; i++ )
{
            y[ i ] *= 2.;
}
```

But, it *can* be broken into one loop that is not parallelizable, plus one that is:

OpenMP: parallel loops

```
for( int i = 1; i < N; i++ )
{
          for( int j = 0; j < M; j++ )
          {
                    . . .
          }
}
```

Ah-ha – trick question. You put it on both!

How many for-loops to collapse into one loop

```
#pragma omp parallel for collapse(2)
for( int i = 1; i < N; i++ )
{
          for( int j = 0; j < M; j++ )
          {
                    . . .
          }
}
```

The Sections Construct

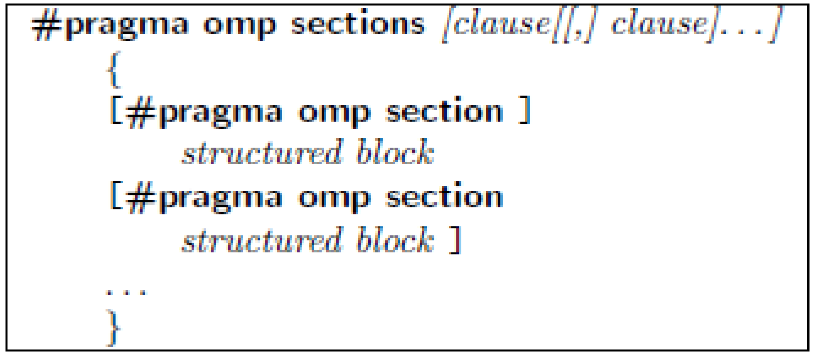
- The *sections construct* is the easiest way to get different threads to carry out different kinds of work, since it permits us to specify several different code regions, each of which will be executed by one of the threads.

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

   #pragma omp for
   for (i=0; i<n; i++)
      b[i] = 2 * a[i];
} /*-- End of parallel region --*/
```

Two work-sharing loops in one parallel region

- No guarantee that the distribution of iterations to threads is identical for both loops but the implied barrier ensures that results are available when needed.

- Each section must be a structured block of code that is independent of the other sections.
- At run time, the specified code blocks are executed by the threads in the team. Each thread executes one code block at a time, and each code block will be executed exactly once.
- If there are fewer threads than code blocks, some or all of the threads execute multiple code blocks.
- If there are fewer code blocks than threads, the remaining threads will be idle.
- The assignment of code blocks to threads is implementation-dependent.
- Commonly used to execute function or subroutine calls in parallel.



```
#pragma omp sections [clause[[,] clause]...]
     {
     [#pragma omp section ]
          structured block
     [#pragma omp section
          structured block ]
     ...
     }
```

Example of parallel sections

- If two or more threads are available, one thread invokes funcA() and another thread calls funcB(). Any other threads are idle.
- This code fragment contains one sections construct, comprising two sections.
- This limits the parallelism to two threads.
- If two or more threads are available, function calls funcA and funcB are executed in parallel.
- If only one thread is available, both calls to funcA and funcB are executed, but in sequential order.
- One cannot make any assumption on the specific order in which section blocks are executed.
- Even if these calls are executed sequentially, because the directive is not in an active parallel region, funcB may be called before funcA.

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
      (void) funcA();

    #pragma omp section
      (void) funcB();
  } /*-- End of sections block --*/

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

Load-balancing problem

- Depending on the type of work performed in the various code blocks and the number of threads used, this construct might lead to a *load-balancing* problem.
- This occurs when threads have different amounts of work to do and thus take different amounts of time to complete.
- A result of load imbalance is that some threads may wait a long time at the next barrier in the program, which means that the hardware resources are not being efficiently exploited.
- It may sometimes be possible to eliminate the barrier at the end of the construct but that does not overcome the fundamental problem of a load imbalance *within* the sections construct.

The Single Construct

- The *single construct* is associated with the structured block of code immediately following it and specifies that this block should be executed by one thread only.
- It does not state which thread should execute the code block; indeed, the thread chosen could vary from one run to another.
- This construct should be used when we do not care which thread executes this part of the application, as long as the work gets done by exactly one thread.
- The other threads wait at a barrier until the thread executing the single code block has completed.

Syntax:

#pragma omp single *[clause[[,] clause]. . .]* *structured block*

- Only one thread executes the structured block

Example of the single construct

- Only one thread initializes the shared variable a. This variable is then used to initialize vector b in the parallelized for-loop

```
#pragma omp parallel shared(a,b) private(i)
{
#pragma omp single
{
a = 10;
printf("Single construct executed by thread %d\n",
            omp_get_thread_num());
}

/* A barrier is automatically inserted here */
#pragma omp for
for (i=0; i<n; i++)
b[i] = a;
} /*-- End of parallel region --*/

printf("After the parallel region:\n");
for (i=0; i<n; i++)
printf("b[%d] = %d\n",i,b[i]);
```

- A barrier is essential before the ***#pragma omp for*** loop. Without such a barrier, some threads would begin to assign values to elements of b before a has been assigned a value
- Every thread would write the same value of 10 to the same variable a. However, this approach raises a hardware issue.
    - Depending on the data type, the processor details, and the compiler behavior, the write to memory might be translated into a sequence of store instructions, each store writing a subset of the variable.
    - Example: A variable 8 bytes long might be written to memory through 2 store instructions of 4 bytes each
    - Multiple threads could do this at the same time, resulting in an arbitrary combination of bytes in memory.
    - This issue is also related to the memory consistency model.
    - Moreover, multiple stores to the same memory address are bad for performance

Combined Parallel Work-Sharing Constructs

- They are shortcuts that can be used when a parallel region comprises precisely one work-sharing construct, that is, the work sharing region includes all the code in the parallel region.
- The combined parallel work-sharing constructs allow certain clauses that are supported by both the parallel construct and the workshare construct.

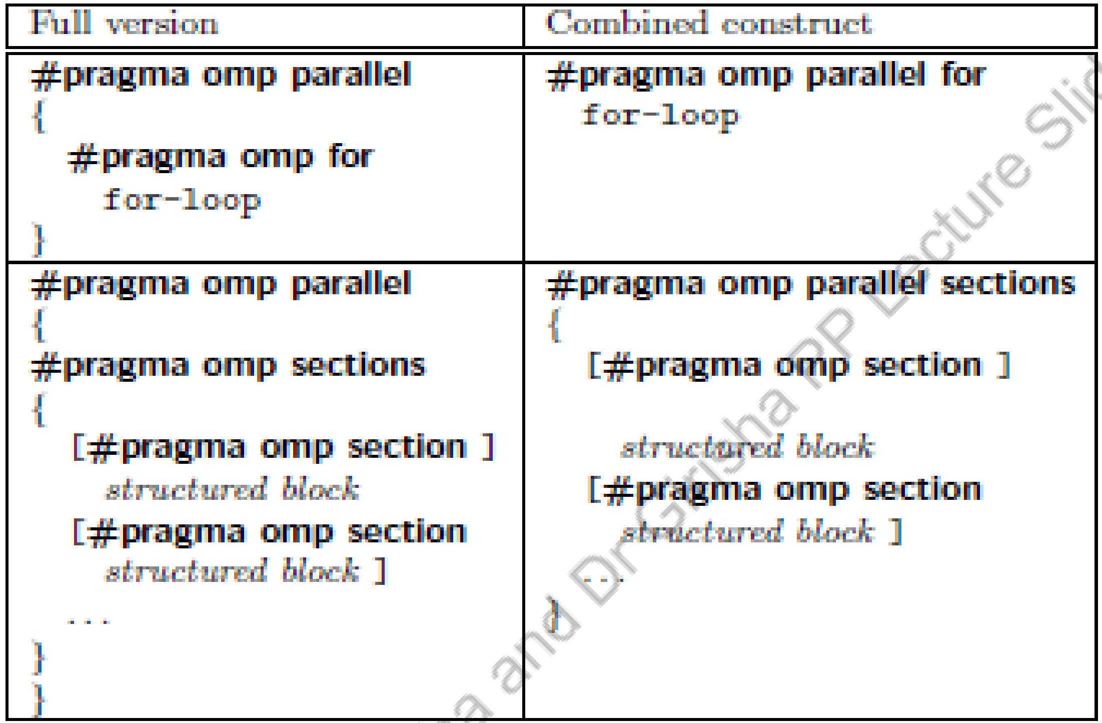
```
#pragma omp parallel
{
    #pragma omp for
     for (.....)
    }
```

A single work-sharing loop in a parallel region – For cases like this OpenMP provides a shortcut.

```
#pragma omp parallel for
for (.....)
```

Syntax of the combined constructs in C/C++

- The combined constructs may have a performance advantage over the more general parallel region with just one work-sharing construct embedded.



Full version	Combined construct
#pragma omp parallel { #pragma omp for for-loop }	#pragma omp parallel for for-loop
#pragma omp parallel { #pragma omp sections { [#pragma omp section] *structured block* [#pragma omp section *structured block*] ... } }	#pragma omp parallel sections { [#pragma omp section] *structured block* [#pragma omp section *structured block*] ... }

- The main advantage of using these combined constructs is readability.
- When the combined construct is used, a compiler knows what to expect and may be able to generate slightly more efficient code.

Clauses to Control Parallel and Work-Sharing Constructs

Shared Clause :

- The shared clause is used to specify which data will be shared among the threads executing the region it is associated with.
- Simply stated, there is one unique instance of these variables, and each thread can freely read or modify the values.
- The syntax for this clause is shared(*list*) . All items in the list are data objects that will be shared among the threads in the team.

```
#pragma omp parallel for
shared(a)
        for (i=0; i<n; i++)
        {
           a[i] += i;
         } /*-- End of parallel for --
*/
```

- Here, vector **a** is declared to be shared. This implies that all threads are able to read and write elements of **a**.
- Within the parallel loop, each thread will access the pre-existing values of those elements a[i] of **a** that it is responsible for updating and will compute their new values.
- After the parallel region is finished, all the new values for elements of **a** will be in main memory, where the master thread can access them.

- Multiple threads might attempt to simultaneously update the same memory location or that one thread might try to read from a location that another thread is updating.
- Special care has to be taken to ensure that neither of these situations occurs and that accesses to shared data are ordered as required by the algorithm.

Private Clause

- Since the loop iterations are distributed over the threads in the team, each thread must be given a unique and local copy of the loop variable i so that it can safely modify the value.
- Otherwise, a change made to **i** by one thread would affect the value of **i** in another thread's memory, thereby making it impossible for the thread to keep track of its own set of iterations
- private clause is used when data objects in a parallel region or work-sharing construct require which threads should be given their own copies

The syntax is private(*list*)

- Each variable in the list is replicated so that each thread in the team of threads has exclusive access to a local copy of this variable.
- Changes made to the data by one thread are not visible to other threads.

- If variable a had been specified in a shared clause, multiple threads would attempt to update the *same* variable with different values in an uncontrolled manner.
- The final value would thus depend on which thread happened to last update **a**. (This bug is a data race condition.) Therefore, the usage of a requires us to specify it to be a private variable, ensuring that each thread has its own copy.

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

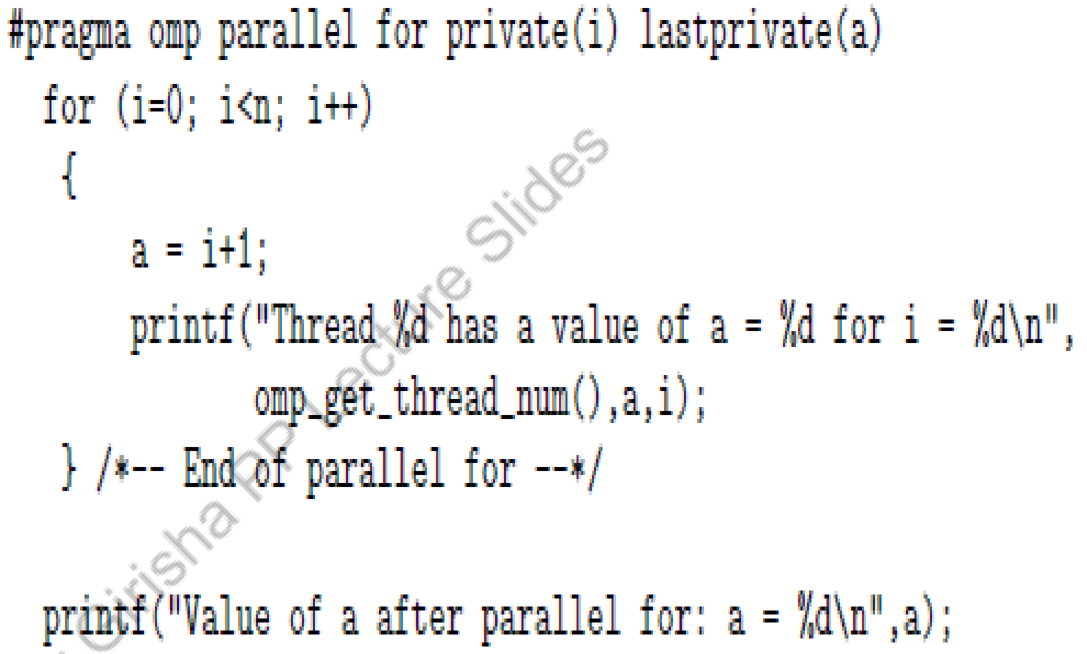
```
Thread 0 has a value of a = 1 for i = 0
Thread 0 has a value of a = 2 for i = 1
Thread 2 has a value of a = 5 for i = 4
Thread 1 has a value of a = 3 for i = 2
Thread 1 has a value of a = 4 for i = 3
```

Lastprivate Clause

- It ensures that the last value of a data object listed is accessible after the corresponding construct has completed execution.
- In a parallel program, we must explain what "last" means.
- In the case of its use with a work-shared loop, the object will have the value from the iteration of the loop that would be last in a sequential execution.
- If the lastprivate clause is used on a sections construct, the object gets assigned the value that it has at the end of the lexically last sections construct.

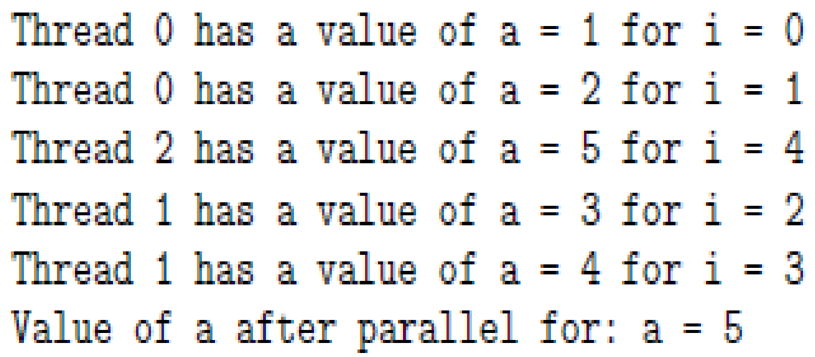
- The syntax is **lastprivate(*list*).**

- Variable **a** now has the lastprivate data-sharing attribute
- There is a print statement after the parallel region so that we can check on the value **a** has at that point.
- According to the definition of "last," the value of variable **a** after the parallel region, should correspond to that computed when i = n-1
- **Output**: Variable *n* is set to 5, and three threads are used. The last value of variable **a** corresponds to the value for *i* = 4



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

  printf("Value of a after parallel for: a = %d\n",a);
```

Output:

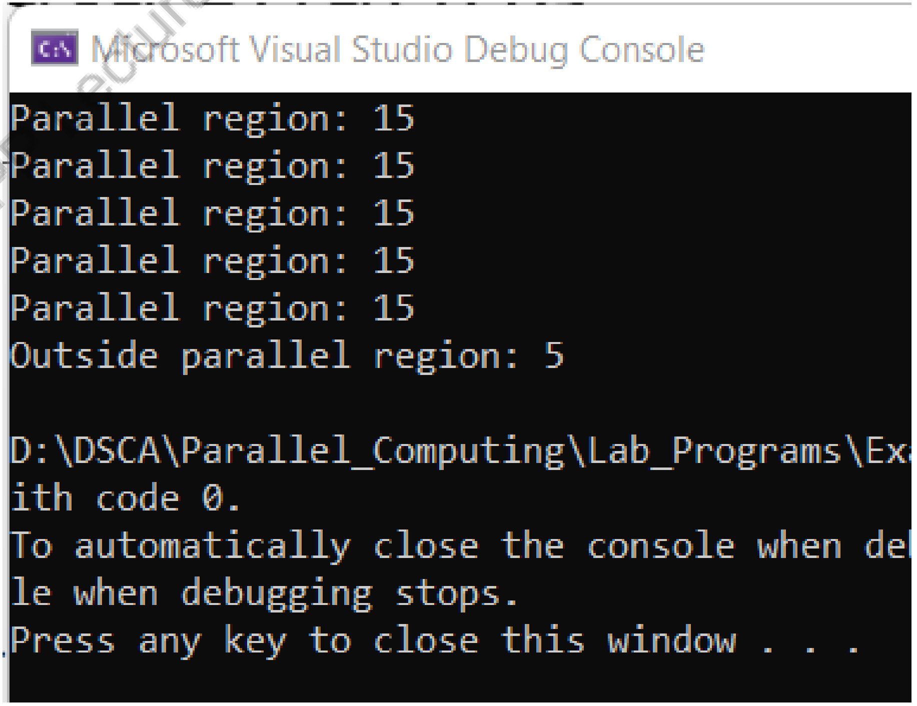


```
Thread 0 has a value of a = 1 for i = 0
Thread 0 has a value of a = 2 for i = 1
Thread 2 has a value of a = 5 for i = 4
Thread 1 has a value of a = 3 for i = 2
Thread 1 has a value of a = 4 for i = 3
Value of a after parallel for: a = 5
```

Firstprivate Clause

- Variables that are declared to be “firstprivate” are private variables, but they are pre-initialized with the value of the variable with the same name before the construct.
- The initialization is carried out by the initial thread prior to the execution of the construct.
- The firstprivate clause is supported on the parallel construct, plus the work-sharing loop, sections, and single constructs.
- The syntax is **firstprivate(*list*).**

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

int main()
{
    int a = 5;
    #pragma omp parallel num_threads(5) firstprivate(a)
    {
        int b = 10;
        a = a + b;
        printf("Parallel region: %d\n", a);
    }

    printf("Outside parallel region: %d\n", a);
    return 0;
}
```

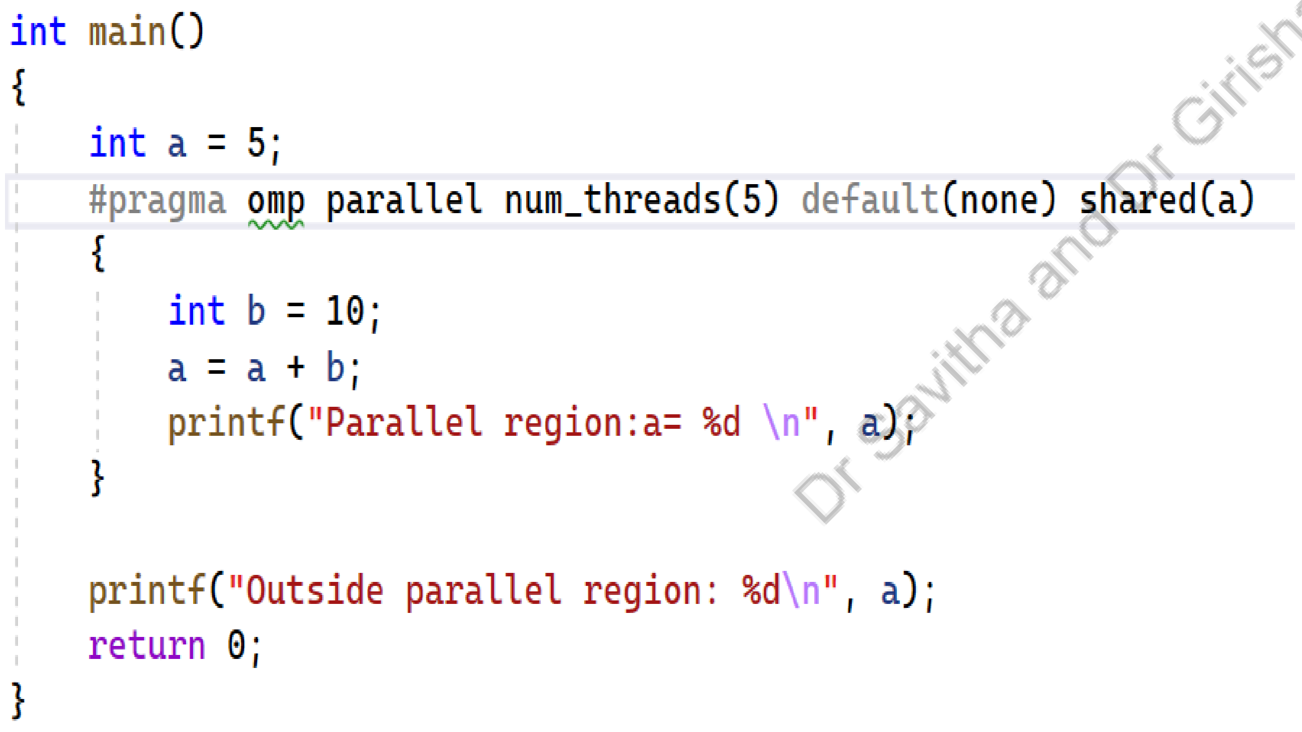

Microsoft Visual Studio Debug Console

```
Parallel region: 15
Parallel region: 15
Parallel region: 15
Parallel region: 15
Parallel region: 15
Outside parallel region: 5

D:\DSCA\Parallel_Computing\Lab_Programs\Ex
ith code 0.
To automatically close the console when de
le when debugging stops.
Press any key to close this window . . .
```

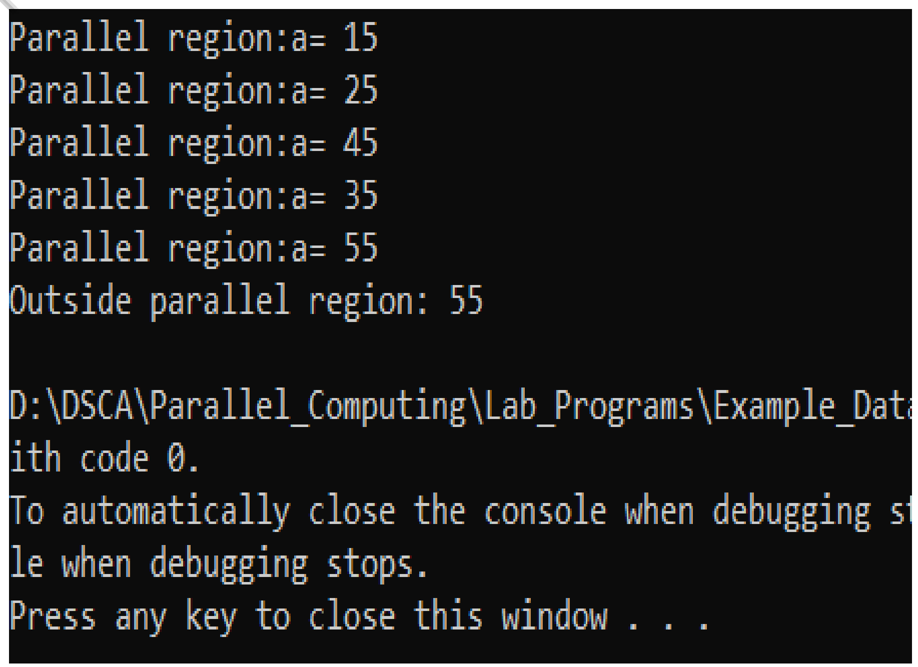
Default Clause

- The default clause is used to give variables a default data-sharing attribute
- Example:
  - default(shared) assigns the shared attribute to all variables referenced in the construct.
  - The default(private) clause, which is not supported in C/C++, makes all variables private by default. It is applicable to the parallel construct only.
- This clause is used to define the data-sharing attribute of the majority of the variables in a parallel region. Only the exceptions need to be explicitly listed:
  - **#pragma omp for default(shared) private(a,b,c),** declares all variables to be shared, with the exception of a, b, and c.

- If **default(none)** is specified, the programmer is forced to specify a data-sharing attribute for each variable in the construct.
- Although variables with a predetermined data-sharing attribute need not be listed in one of the clauses
- It is recommended that the attribute be explicitly specified for *all* variables in the construct.



```
int main()
{
    int a = 5;
    #pragma omp parallel num_threads(5) default(none) shared(a)
    {
        int b = 10;
        a = a + b;
        printf("Parallel region:a= %d \n", a);
    }

    printf("Outside parallel region: %d\n", a);
    return 0;
}
```



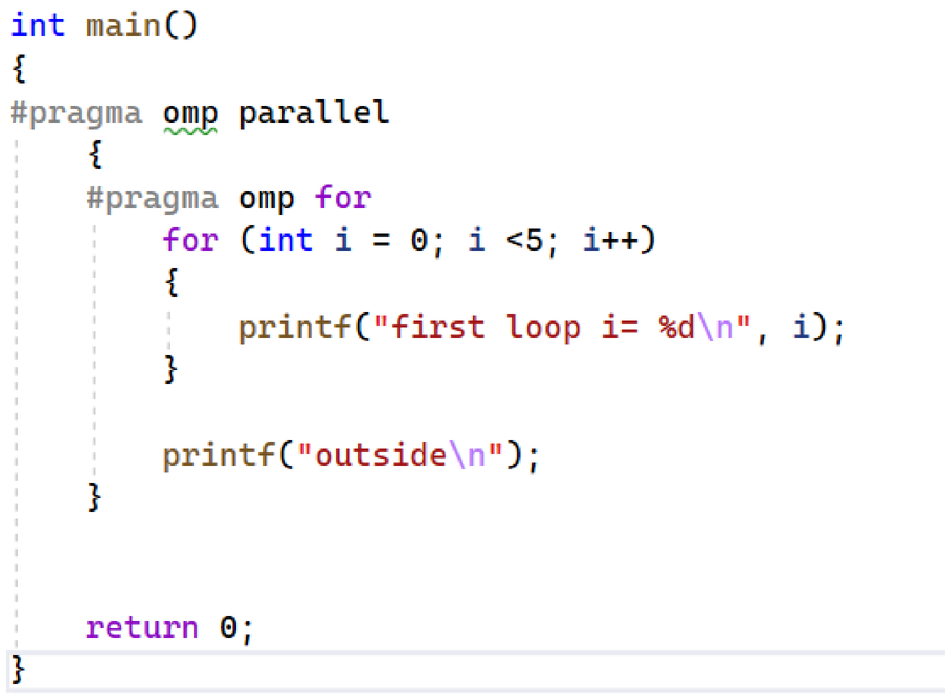
```
Parallel region:a= 15
Parallel region:a= 25
Parallel region:a= 45
Parallel region:a= 35
Parallel region:a= 55
Outside parallel region: 55

D:\DSCA\Parallel_Computing\Lab_Programs\Example_Dat
ith code 0.
To automatically close the console when debugging s
le when debugging stops.
Press any key to close this window . . .
```

Nowait Clause

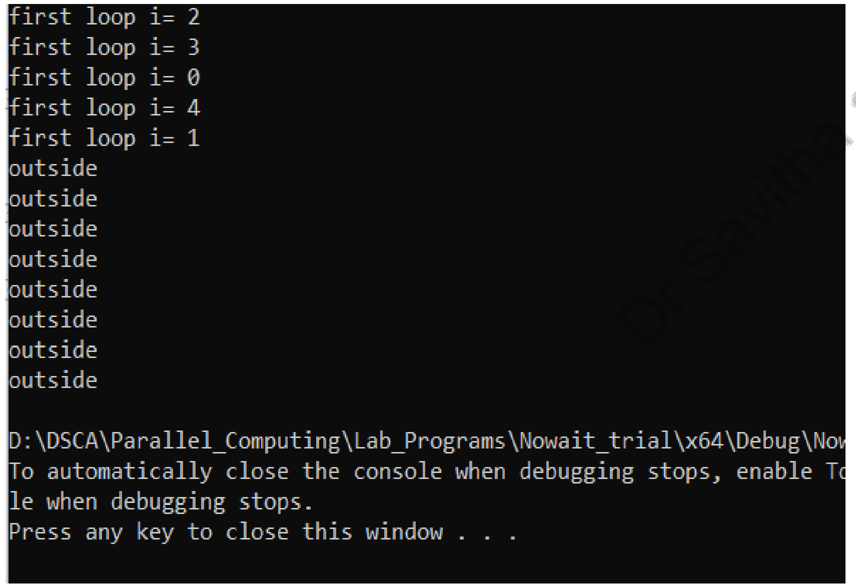
- The nowait clause allows the programmer to fine-tune a program's performance.
- In the work-sharing constructs, there is an implicit barrier at the end of them. This clause overrides that feature of OpenMP;
- That is, if it is added to a construct, the barrier at the end of the associated construct will be suppressed.
- When threads reach the end of the construct, they will immediately proceed to perform other work.
- However, the barrier at the end of a parallel region cannot be suppressed.

- When a thread is finished with the work associated with the parallelized for loop, it continues and no longer waits for the other threads to finish as well.
- The clause ensures that there is no barrier at the end of the loop.

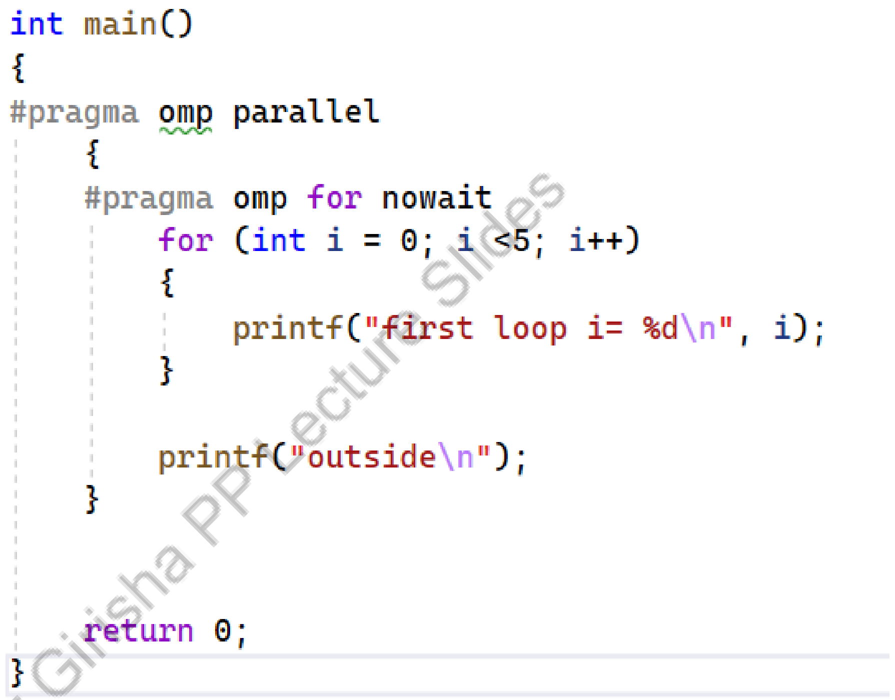
```
#pragma omp for nowait
  for (i=0; i<n; i++)
   {
       ............
   }
```



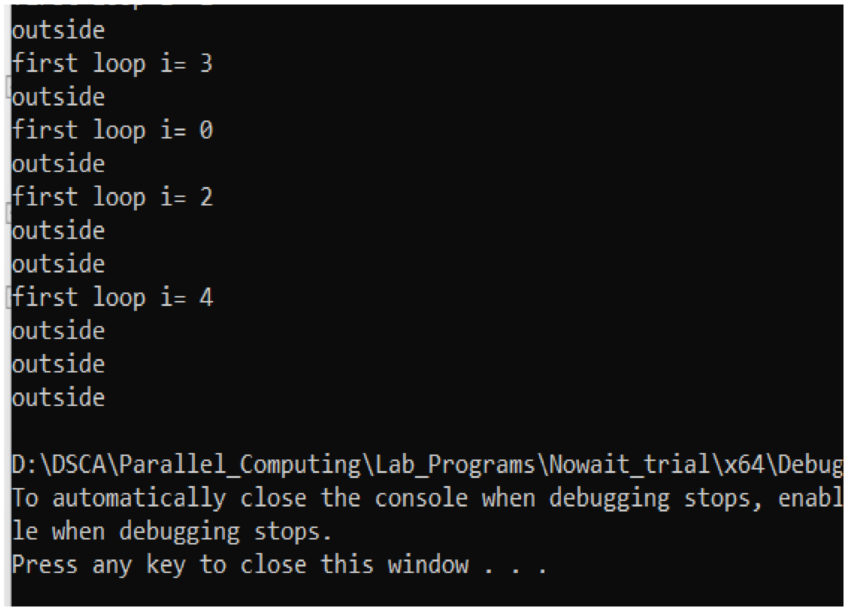
```
int main()
{
#pragma omp parallel
    {
    #pragma omp for
        for (int i = 0; i <5; i++)
        {
            printf("first loop i= %d\n", i);
        }

        printf("outside\n");
    }


    return 0;
}
```



```
first loop i= 2
first loop i= 3
first loop i= 0
first loop i= 4
first loop i= 1
outside
outside
outside
outside
outside
outside
outside
outside

D:\DSCA\Parallel_Computing\Lab_Programs\Nowait_trial\x64\Debug\Now
To automatically close the console when debugging stops, enable To
le when debugging stops.
Press any key to close this window . . .
```



```
int main()
{
#pragma omp parallel
    {
    #pragma omp for nowait
        for (int i = 0; i <5; i++)
        {
            printf("first loop i= %d\n", i);
        }

        printf("outside\n");
    }


    return 0;
}
```



```
outside
first loop i= 3
outside
first loop i= 0
outside
first loop i= 2
outside
outside
first loop i= 4
outside
outside
outside

D:\DSCA\Parallel_Computing\Lab_Programs\Nowait_trial\x64\Debug
To automatically close the console when debugging stops, enabl
le when debugging stops.
Press any key to close this window . . .
```

Schedule Clause

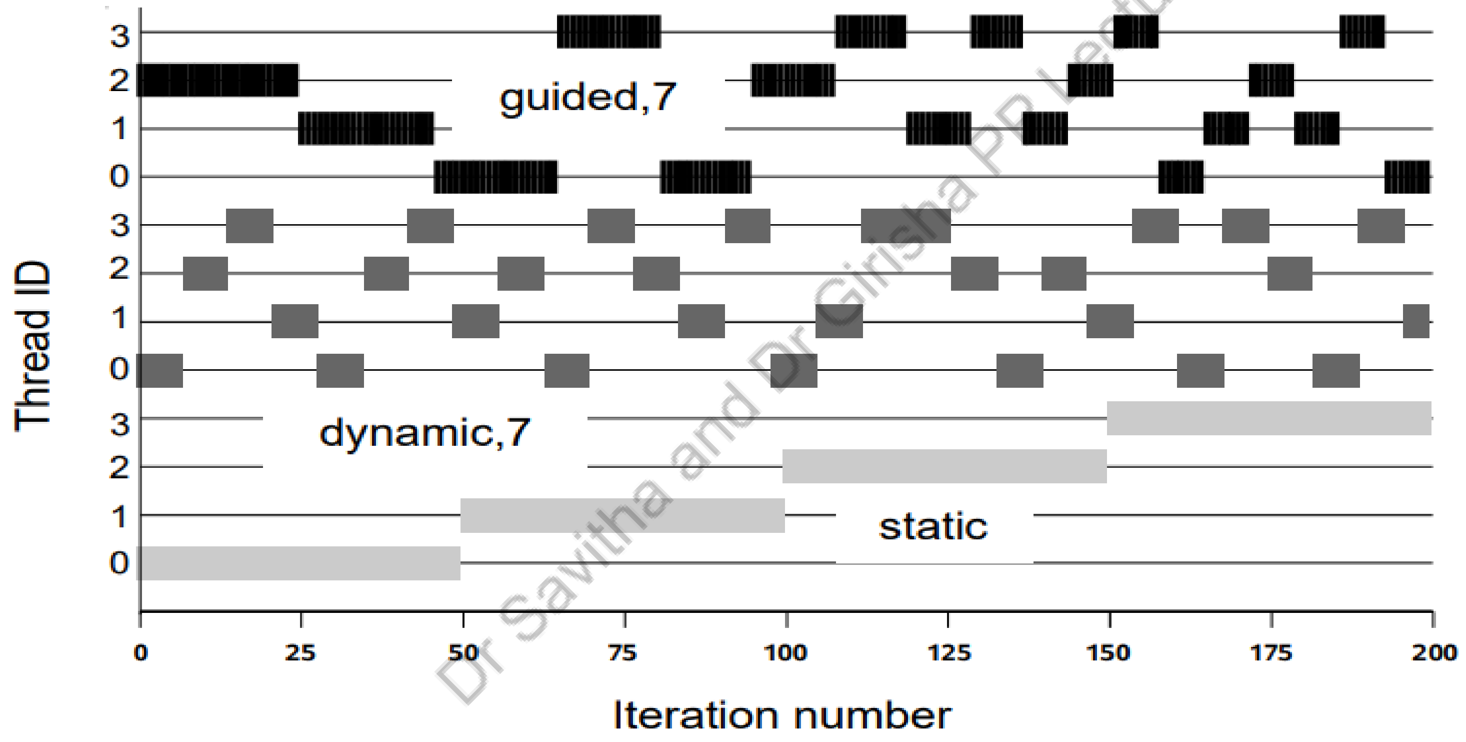
- The schedule clause is supported on the loop construct only.
- It is used to control the manner in which loop iterations are distributed over the threads, which can have a major impact on the performance of a program.
- The syntax is **schedule(*kind[,chunk_size]*)**
- The schedule clause specifies how the iterations of the loop are assigned to the threads in the team.
- The granularity of this workload distribution is a ***chunk***, a contiguous, nonempty subset of the iteration space.
- The chunk_size parameter need not be a constant; any loop invariant integer expression with a positive value is allowed

Kinds of Schedule

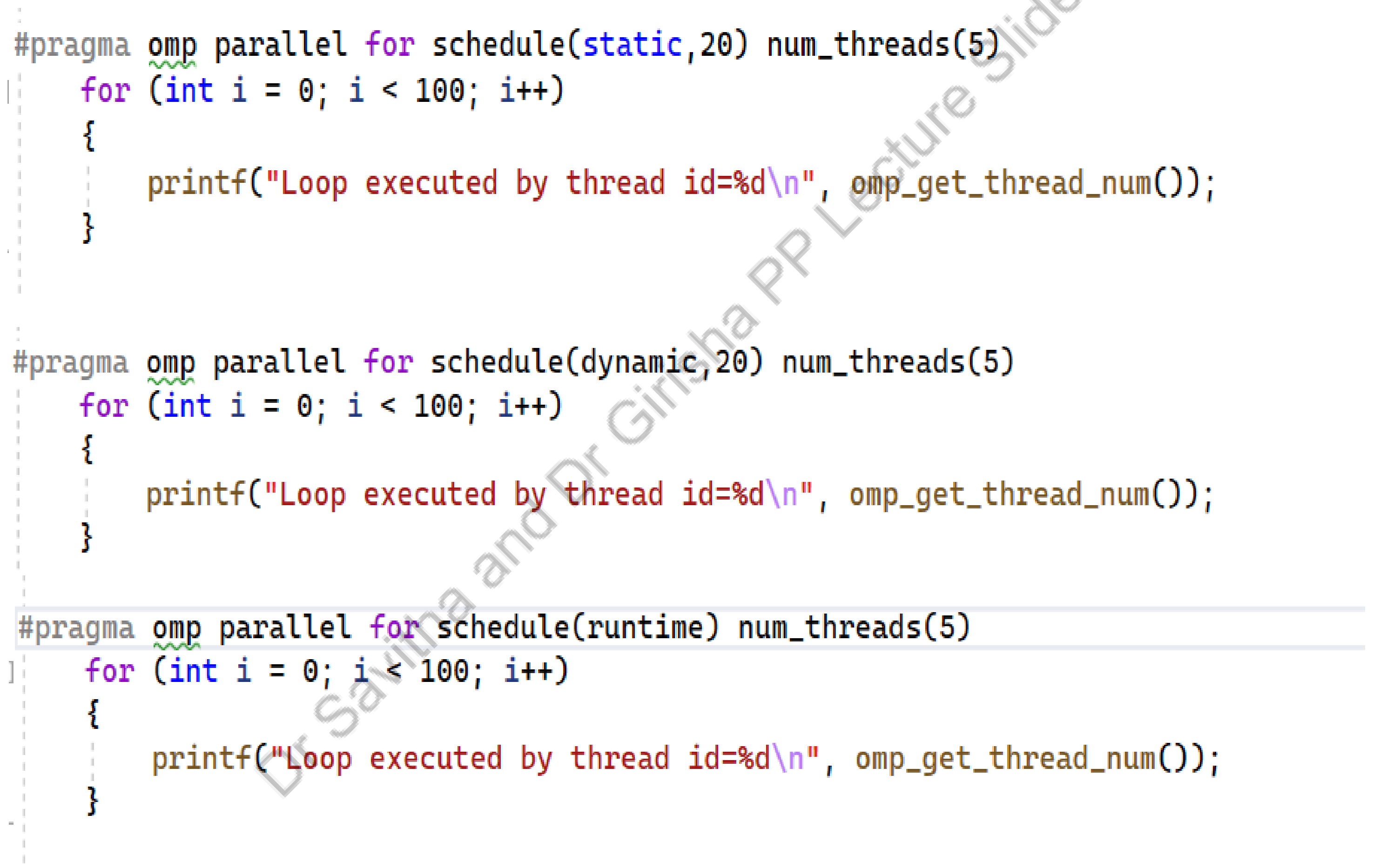
- **Static** :
  - Iterations are divided into chunks of size ***chunk size***.
  - The chunks are assigned to the threads statically in a **round-robin manner**, in the **order of the thread number**.
  - The last chunk to be assigned may have a smaller number of iterations.
  - When no ***chunk size*** is specified, the iteration space is divided into chunks that are approximately equal in size.
  - Each thread is assigned at most one chunk.
- **Dynamic**:
  - The iterations are assigned to threads as the threads request them.
  - The thread executes the chunk of iterations (controlled through the *chunk size* parameter), then requests another chunk until there are no more chunks to work on.
  - The last chunk may have fewer iterations than *chunk size*.
  - When no *chunk size* is specified, it defaults to 1.

- **Guided:**
  - The iterations are assigned to threads as the threads request them.
  - The thread executes the chunk of iterations (controlled through the *chunk size* parameter), then requests another chunk until there are no more chunks to work on.
  - For a *chunk size* of 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 size* of "*k*" (*k* > 1), the size of each chunk is determined in the same way, with the restriction that the chunks do not contain fewer than *k* iterations
  - When no *chunk size* is specified, it defaults to 1.
- **Runtime:**
  - If this schedule is selected, the decision regarding scheduling kind is made at run time.
  - The schedule and (optional) chunk size are set through the OMP_SCHEDULE environment variable.

OpenMP: Schedule Clause



OpenMP: Schedule Clause



```
#pragma omp parallel for schedule(static,20) num_threads(5)
    for (int i = 0; i < 100; i++)
    {
        printf("Loop executed by thread id=%d\n", omp_get_thread_num());
    }


#pragma omp parallel for schedule(dynamic,20) num_threads(5)
    for (int i = 0; i < 100; i++)
    {
        printf("Loop executed by thread id=%d\n", omp_get_thread_num());
    }

#pragma omp parallel for schedule(runtime) num_threads(5)
    for (int i = 0; i < 100; i++)
    {
        printf("Loop executed by thread id=%d\n", omp_get_thread_num());
    }
```

All three workload distribution algorithms support an optional *chunk size* parameter.

- For example, a *chunk size* bigger than 1 on the static schedule may give rise to a round-robin allocation scheme in which each thread executes the iterations in a sequence of chunks whose size is given by *chunk size*.
- **It is not always easy to select the appropriate schedule and value for *chunk size* up front.**
- The choice may depend (among other things) not only on the code in the loop but also on the specific problem size and the number of threads used.
- **Therefore, the runtime clause is convenient.**
- Instead of making a compile time decision, the OpenMP OMP_SCHEDULE environment variable can be used to choose the schedule and (optional) *chunk size* at run time

Example for schedule clause

- The outer loop has been parallelized with the loop construct.
- The workload in the inner loop depends on the value of the outer loop iteration variable i.
- Therefore, the workload is not balanced, and the static schedule is probably not the best choice.

```
#pragma omp parallel for default(none) schedule(runtime) \
                    private(i,j) shared(n)
  for (i=0; i<n; i++)
   {
      printf("Iteration %d executed by thread %d\n",
          i, omp_get_thread_num());
      for (j=0; j<i; j++)
          system("sleep 1");
   } /*-- End of parallel for --*/
```

OpenMP Synchronization Constructs

- Help to organize accesses to shared data by multiple threads.
- An algorithm may require us to orchestrate the actions of multiple threads to ensure that updates to a shared variable occur in a certain order, or it may simply need to ensure that two threads do not simultaneously attempt to write a shared object.
- These features can be used when the implicit barrier provided with work-sharing constructs does not suffice to specify the required interactions or would be inefficient.
- Together with the work-sharing constructs, they constitute a powerful set of features that suffice to parallelize a large number of applications.

Types of Constructs:

- **Barrier Construct**
- **Ordered Construct**
- **Critical Construct**
- **Atomic Construct**
- **Locks**
- **Master Construct**

Barrier Construct

- A barrier is a point in the execution of a program where threads wait for each other:
  - no thread in the team of threads it applies to, may proceed beyond a barrier until all threads in the team have reached that point.
- Compiler automatically inserts a barrier at the end of the construct, so that all threads wait there until all of the work associated with the construct has been completed.
- Thus, it is often **not necessary** for the programmer to explicitly add a barrier to a code.

```
#pragma omp barrier
```

- Two important restrictions apply to the barrier construct:
  - Each barrier ***must*** be encountered by all threads in a team, or by none at all.
  - The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in the team.

- Without these restrictions, one could write programs where some threads wait forever (or until somebody kills the process) for other threads to reach a barrier.
- C/C++ imposes an additional restriction regarding the placement of a barrier construct within the application
  - The barrier construct may only be placed in the program at a position where ignoring or deleting it would result in a program with correct syntax.
- The most common use for a barrier is to avoid a **data race condition**
  - Inserting a barrier between the **writes to** and **reads from a shared variable** guarantees that the accesses are appropriately ordered

```
#pragma omp parallel private(TID)
  {
    TID = omp_get_thread_num();
    if (TID < omp_get_num_threads()/2 ) system("sleep 3");
    (void) print_time(TID,"before");

    #pragma omp barrier

    (void) print_time(TID,"after ");
  } /*-- End of parallel region --*/
```

- To ensure that some threads in the team executing the parallel region take longer than others to reach the barrier, we get half the threads to execute the sleep 3 command, causing them to idle for three seconds.
- We then get each thread to print out its the thread number (stored in variable TID), a comment string, and the time of day in the format hh:mm:ss.
- The barrier is then reached.
- After the barrier, each thread will resume execution and again print out this information.
- Four threads are used. Note that threads 2 and 3 wait for three seconds in the barrier.

```
Thread 2 before barrier at 01:12:05
Thread 3 before barrier at 01:12:05
Thread 1 before barrier at 01:12:08
Thread 0 before barrier at 01:12:08
Thread 1 after  barrier at 01:12:08
Thread 3 after  barrier at 01:12:08
Thread 2 after  barrier at 01:12:08
Thread 0 after  barrier at 01:12:08
```

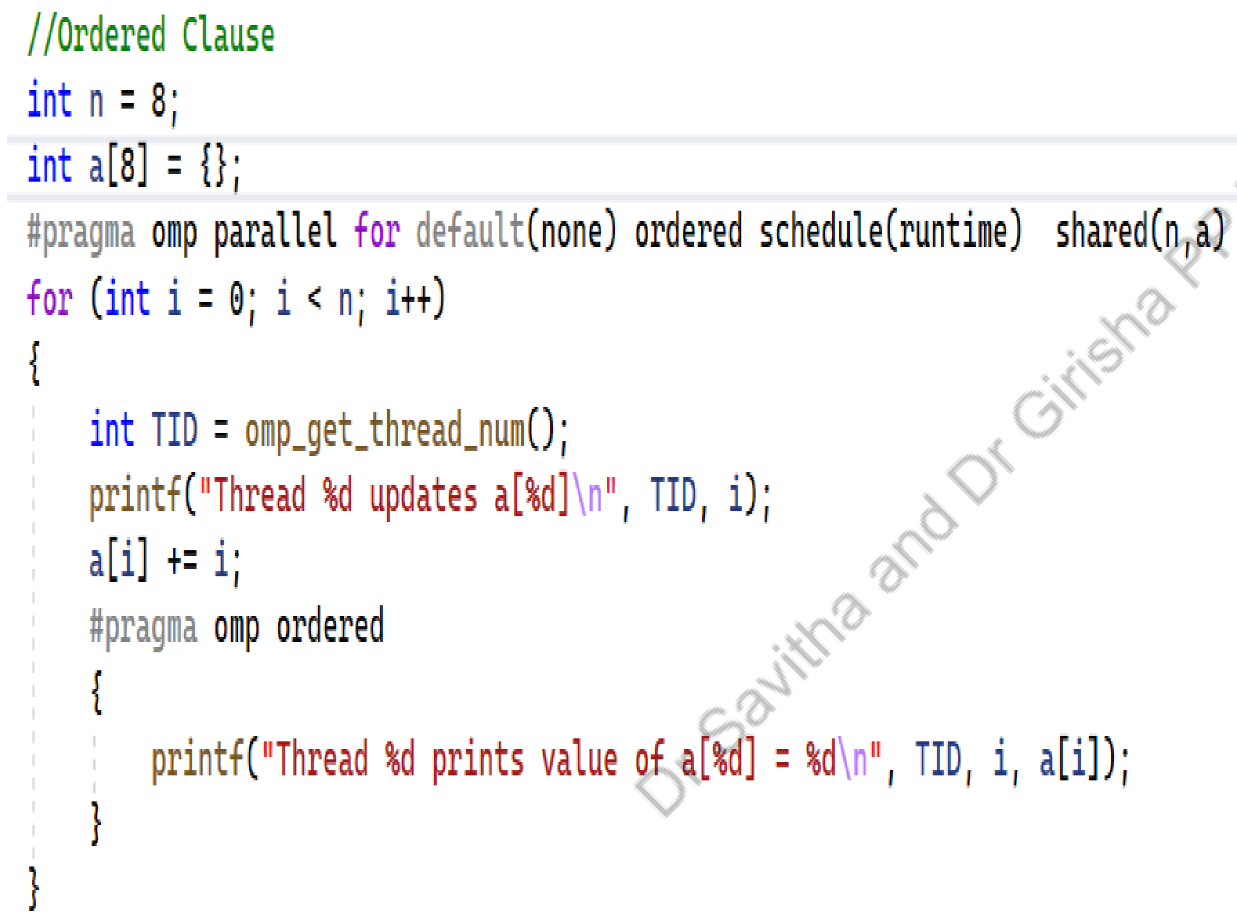
Ordered Construct

- Another synchronization construct, the ordered construct, allows one to execute a structured block within a parallel loop in sequential order.
- This is used to enforce an ordering on the printing of data computed by different threads.
- It may also be used to help determine whether there are any data races in the associated code.
- The syntax of the ordered construct

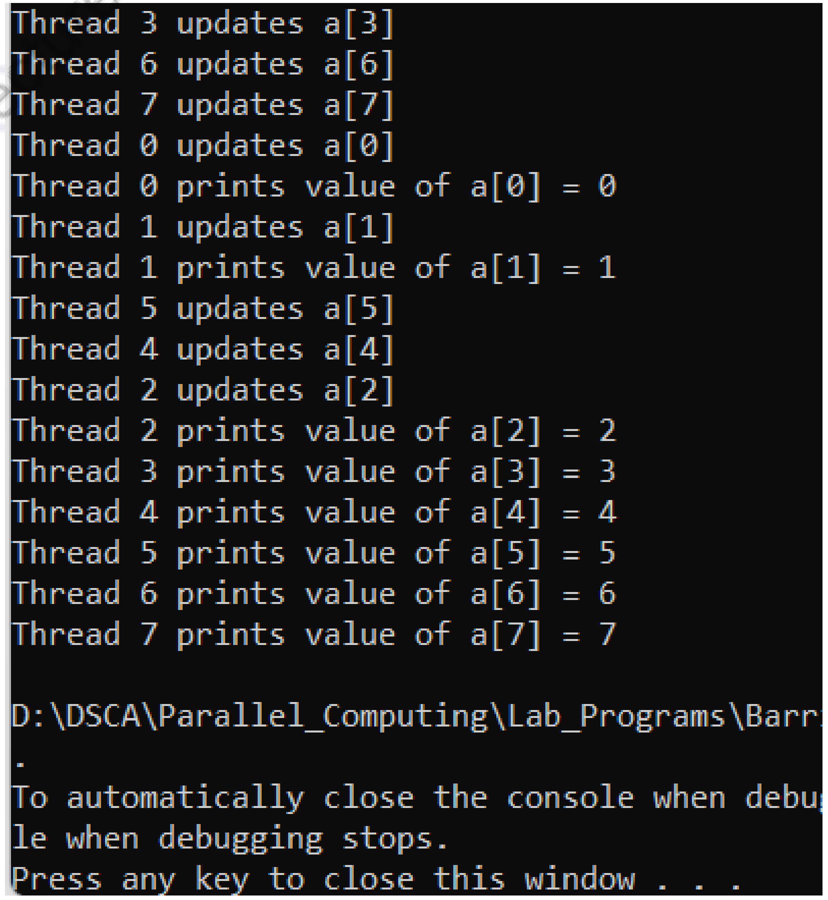
```
#pragma omp ordered
        structured block
```

- An ordered construct ensures that the code within the associated structured block is executed in sequential order.
- The code outside this block runs in parallel. When the thread executing the first iteration of the loop encounters the construct, it enters the region without waiting.
- When a thread executing any subsequent iteration encounters the construct, it waits until each of the previous iterations in the sequence has completed execution of the region

Synchronization Constructs: Ordered



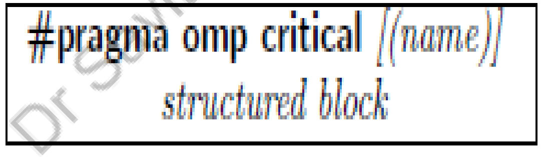
```
//Ordered Clause
int n = 8;
int a[8] = {};
#pragma omp parallel for default(none) ordered schedule(runtime)  shared(n,a)
for (int i = 0; i < n; i++)
{
    int TID = omp_get_thread_num();
    printf("Thread %d updates a[%d]\n", TID, i);
    a[i] += i;
    #pragma omp ordered
    {
        printf("Thread %d prints value of a[%d] = %d\n", TID, i, a[i]);
    }
}
```



```
Thread 3 updates a[3]
Thread 6 updates a[6]
Thread 7 updates a[7]
Thread 0 updates a[0]
Thread 0 prints value of a[0] = 0
Thread 1 updates a[1]
Thread 1 prints value of a[1] = 1
Thread 5 updates a[5]
Thread 4 updates a[4]
Thread 2 updates a[2]
Thread 2 prints value of a[2] = 2
Thread 3 prints value of a[3] = 3
Thread 4 prints value of a[4] = 4
Thread 5 prints value of a[5] = 5
Thread 6 prints value of a[6] = 6
Thread 7 prints value of a[7] = 7

D:\DSCA\Parallel_Computing\Lab_Programs\Barr
.
To automatically close the console when debu
le when debugging stops.
Press any key to close this window . . .
```

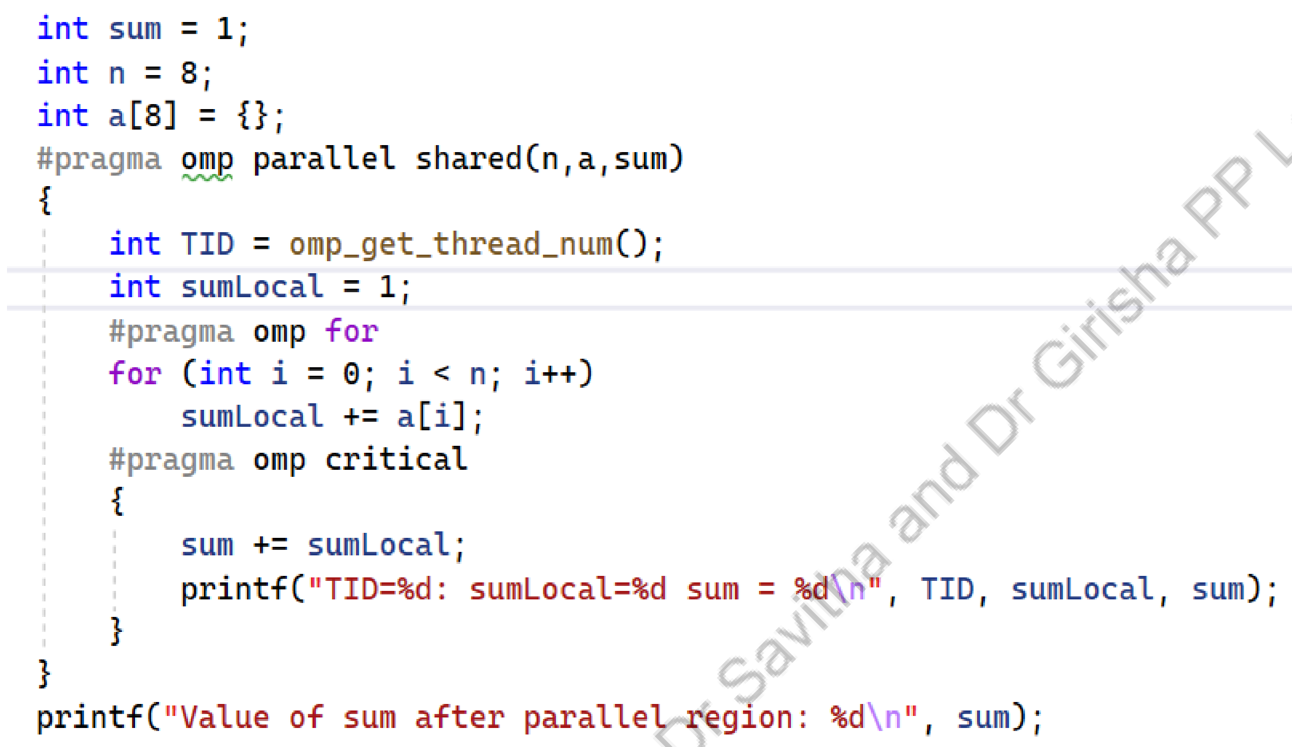
Critical Construct

- The critical construct provides a means to ensure that multiple threads do not attempt to update the same shared data simultaneously. The associated code is referred to as a critical region, or a *critical section*.
- An optional *name* can be given to a critical construct. In contrast to the rules governing other language features, this name is *global* and therefore should be unique
- When a thread encounters a critical construct, it waits until no other thread is executing a critical region with the same name.
- In other words, there is never a risk that multiple threads will execute the code contained in the same critical region at the same time.

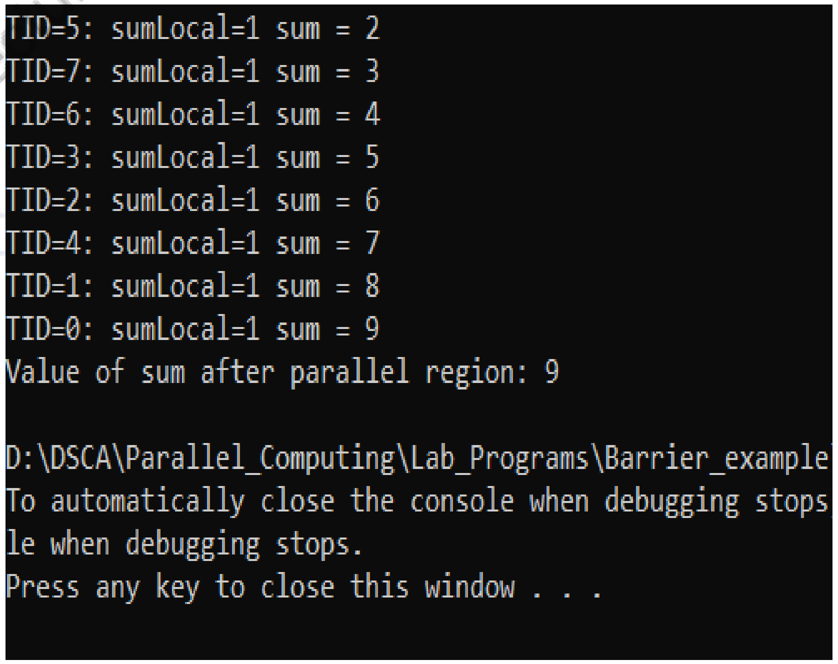


```
#pragma omp critical [(name)]
        structured block
```

Synchronization Constructs: Critical



```
int sum = 1;
int n = 8;
int a[8] = {};
#pragma omp parallel shared(n,a,sum)
{
    int TID = omp_get_thread_num();
    int sumLocal = 1;
    #pragma omp for
    for (int i = 0; i < n; i++)
        sumLocal += a[i];
    #pragma omp critical
    {
        sum += sumLocal;
        printf("TID=%d: sumLocal=%d sum = %d\n", TID, sumLocal, sum);
    }
}
printf("Value of sum after parallel region: %d\n", sum);
```



```
TID=5: sumLocal=1 sum = 2
TID=7: sumLocal=1 sum = 3
TID=6: sumLocal=1 sum = 4
TID=3: sumLocal=1 sum = 5
TID=2: sumLocal=1 sum = 6
TID=4: sumLocal=1 sum = 7
TID=1: sumLocal=1 sum = 8
TID=0: sumLocal=1 sum = 9
Value of sum after parallel region: 9

D:\DSCA\Parallel_Computing\Lab_Programs\Barrier_example
To automatically close the console when debugging stops
le when debugging stops.
Press any key to close this window . . .
```

Atomic Construct

- The atomic construct enables efficient updating of shared variables by multiple threads on hardware platforms which support *atomic* operations.
- The reason it is applied to just one assignment statement is that it protects updates to an individual memory location, the one on the left-hand side of the assignment.
- If a thread is atomically updating a value, then no other thread may do so simultaneously.
- This restriction applies to all threads that execute a program, not just the threads in the same team.
- Syntax is:

```
#pragma omp atomic
        statement
```

- The atomic construct may only be used together with an expression statement in C/C++. The supported operations are: +, *, -, /, &, ^, |, <<, >>

```
int ic, i, n;
ic = 0;
#pragma omp parallel shared(n,ic) private(i)
   for (i=0; i++, i<n)
    {
      #pragma omp atomic
         ic = ic + 1;
    }
printf("counter = %d\n", ic);
```

The atomic construct ensures that no updates are lost when multiple threads are updating a counter value.

```
int atomic_read(const int* x)
{
    int value;
    #pragma omp atomic read
    value = *x;
    return value;
}
void atomic_write(int* x, int value)
{
    #pragma omp atomic write
    *x = value;
}
```

Locks

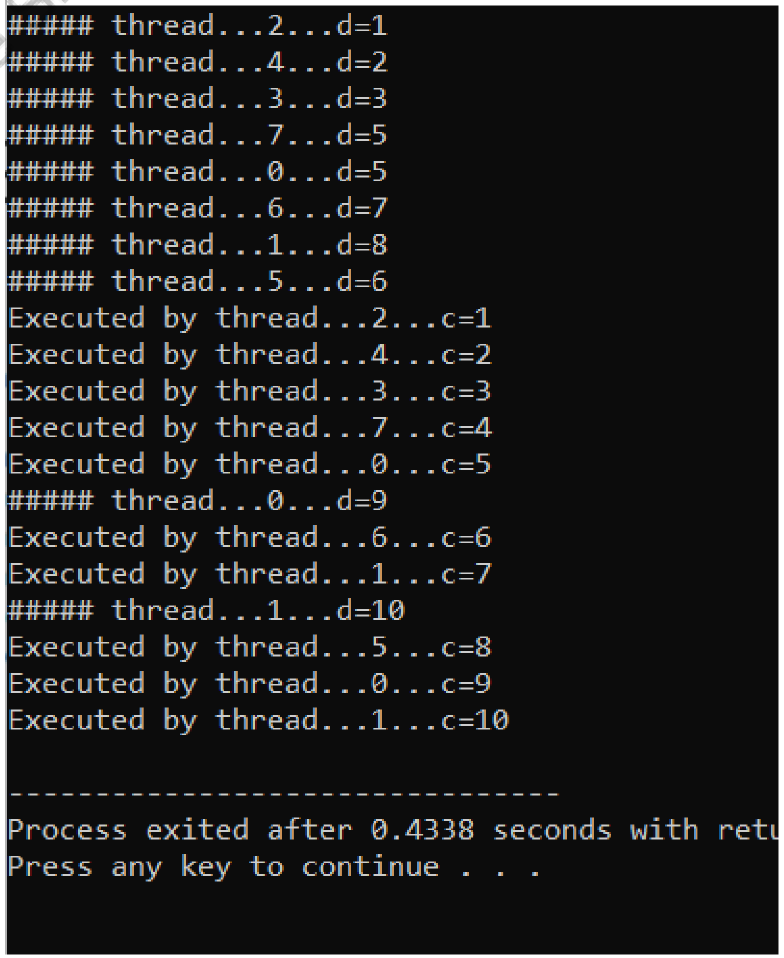
- A set of low-level, general-purpose runtime library routines, similar in function to the use of semaphores. These routines provide greater flexibility for synchronization than does the use of critical sections or atomic constructs.
- A thread lock is an object that can be held by at most one thread at a time
- An OpenMP lock can be in one of the following states:
- Uninitialized; Unlocked; or Locked
- If a lock is in the unlocked state, a thread can set the lock, which changes its state to locked
- The thread that sets the lock is then said to own the lock
- A thread that owns a lock can unset that lock, returning it to the unlocked state
- Syntax is : **void omp_*func*_lock (omp lock t *lck)**

- There are two types of locks:
- ***Simple locks***,
  - which may not be locked if already in a locked state
  - Simple lock variables are declared with the special type omp_lock_t in C/C++
- ***Nestable locks***,
  - Which may be locked multiple times by the same thread
  - Nestable lock variables are declared with the special type omp_nest_lock_t in C/C++

- The general procedure to use locks is as follows:
  - Define the lock variables using **omp_lock_t**
  - Initialize the lock via a call to **omp_init_lock()**
  - Set the lock using **omp_set_lock()** or **omp_test_lock()**. The latter checks whether the lock is actually available before attempting to set it. It is useful to achieve asynchronous thread execution
  - Unset a lock after the work is done via a call to **omp_unset_lock()**
  - Remove the lock association via a call to **omp_destroy_lock()**

Synchronization Constructs: Locks

```
omp_lock_t A;

omp_init_lock(&A);
int b=1;
int c=0;
int d=0;
#pragma omp parallel for
for (int i = 0; i < 10; i++)
{
    // some stuff
    d=d+b;
    printf("##### thread...%d...d=%d\n", omp_get_thread_num(),d);
    omp_set_lock(&A);
    c=c+b;
    printf("Executed by thread...%d...c=%d\n", omp_get_thread_num(),c);
    omp_unset_lock(&A);
    // some stuff
}

omp_destroy_lock(&A);
```



```
##### thread...2...d=1
##### thread...4...d=2
##### thread...3...d=3
##### thread...7...d=5
##### thread...0...d=5
##### thread...6...d=7
##### thread...1...d=8
##### thread...5...d=6
Executed by thread...2...c=1
Executed by thread...4...c=2
Executed by thread...3...c=3
Executed by thread...7...c=4
Executed by thread...0...c=5
##### thread...0...d=9
Executed by thread...6...c=6
Executed by thread...1...c=7
##### thread...1...d=10
Executed by thread...5...c=8
Executed by thread...0...c=9
Executed by thread...1...c=10

--------------------------------
Process exited after 0.4338 seconds with retu
Press any key to continue . . .
```

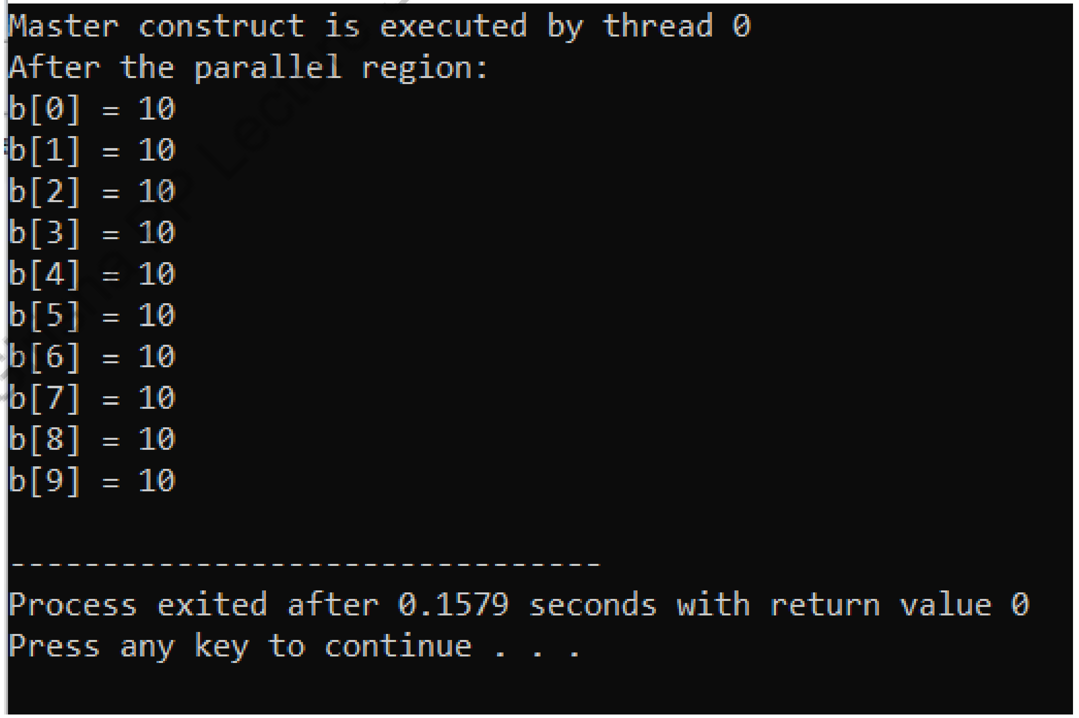
Master Construct

- The master construct defines a block of code that is guaranteed to be executed by the master thread only
- It is thus similar to the single construct
- The master construct is technically not a work-sharing construct, however, and it does not have an implied barrier on entry or exit
- If the master construct is used to initialize data, for example, care needs to be taken that this initialization is completed before the other threads in the team use the data
- Syntax is: **#pragma omp master**

  structured block

Synchronization Constructs: Master

```c
int a=0;
int b[10];
int i=0, n=10;
#pragma omp parallel shared(a,b) private(i)
{
    #pragma omp master
    {
    a = 10;
    printf("Master construct is executed by thread %d\n",
    omp_get_thread_num());
    }
    #pragma omp barrier
    #pragma omp for
    for (i=0; i<n; i++)
    b[i] = a;
} /*-- End of parallel region --*/

printf("After the parallel region:\n");
for (i=0; i<n; i++)
    printf("b[%d] = %d\n",i,b[i]);
```



```
Master construct is executed by thread 0
After the parallel region:
b[0] = 10
b[1] = 10
b[2] = 10
b[3] = 10
b[4] = 10
b[5] = 10
b[6] = 10
b[7] = 10
b[8] = 10
b[9] = 10


--------------------------------
Process exited after 0.1579 seconds with return value 0
Press any key to continue . . .
```

OpenMP: Other Clauses

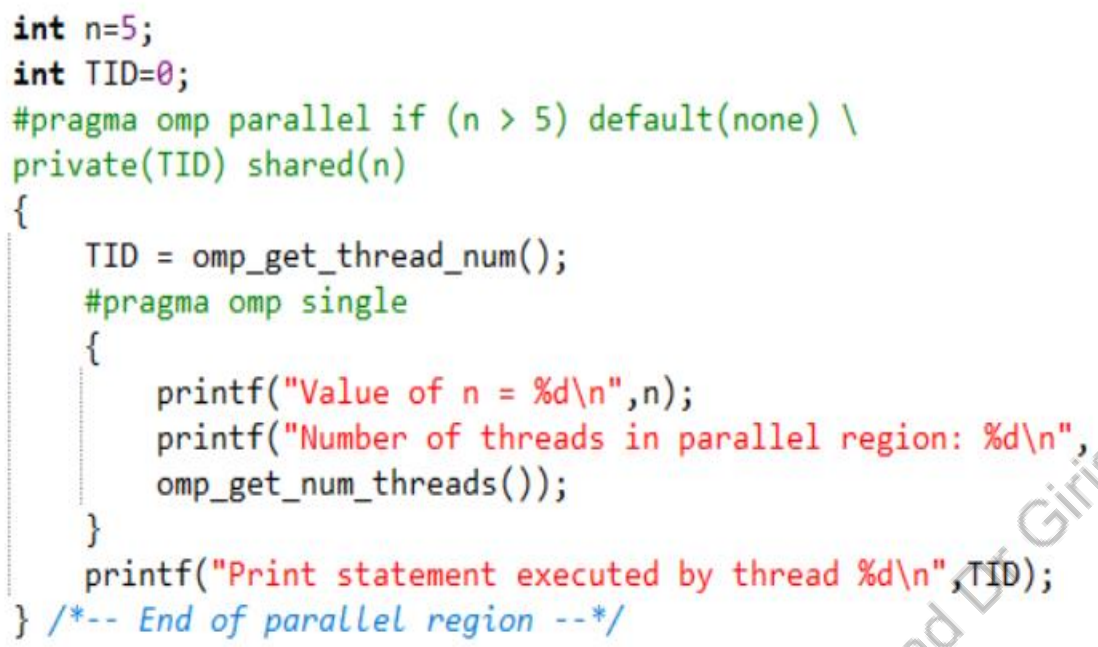
If Clause

- The **if clause** is supported on the parallel construct only, where it is used to specify conditional execution
- Since some overheads are inevitably incurred with the creation and termination of a parallel region, it is sometimes necessary to test whether there is enough work in the region to warrant its parallelization

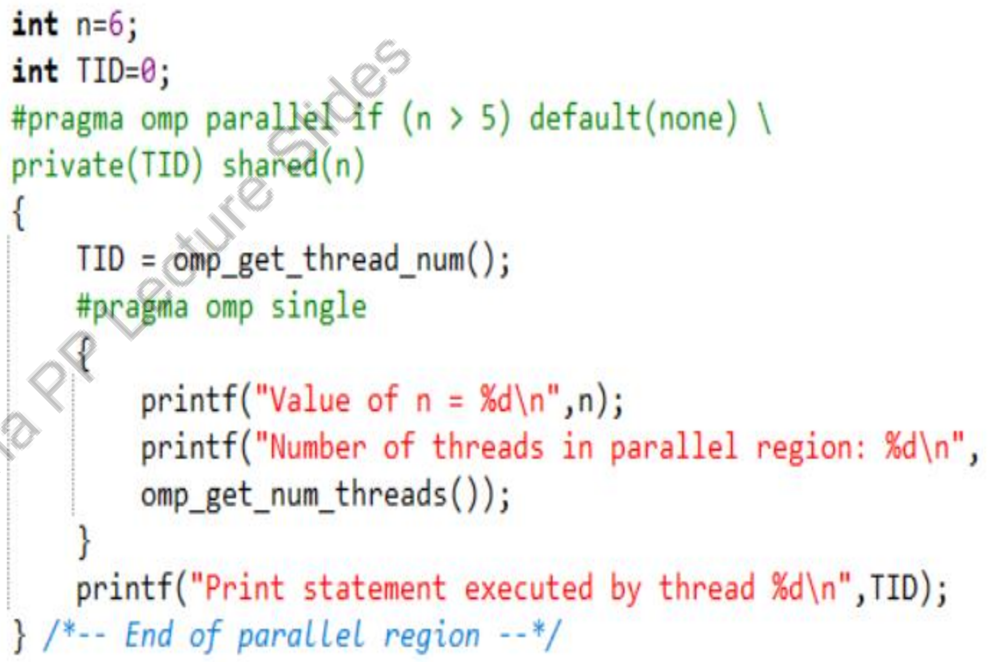
if(scalar-logical-expression)

- If the logical expression evaluates to true, which means it is of type integer and has a non-zero value in C/C++, the parallel region will be executed by a team of threads
- If it evaluates to false, the region is executed by a single thread only

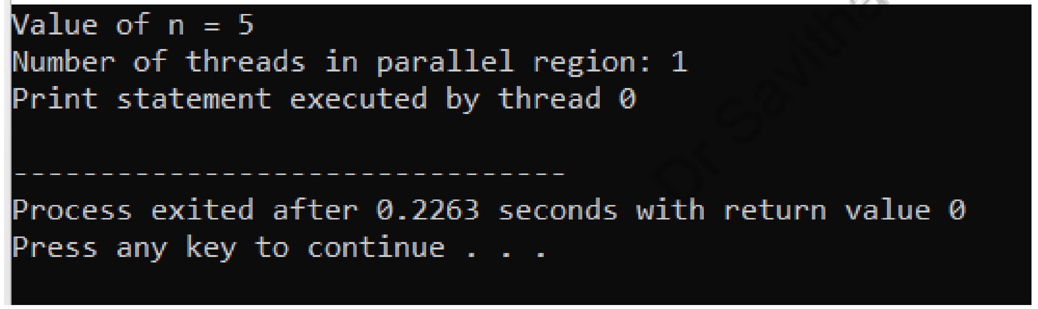
OpenMP: If Clause



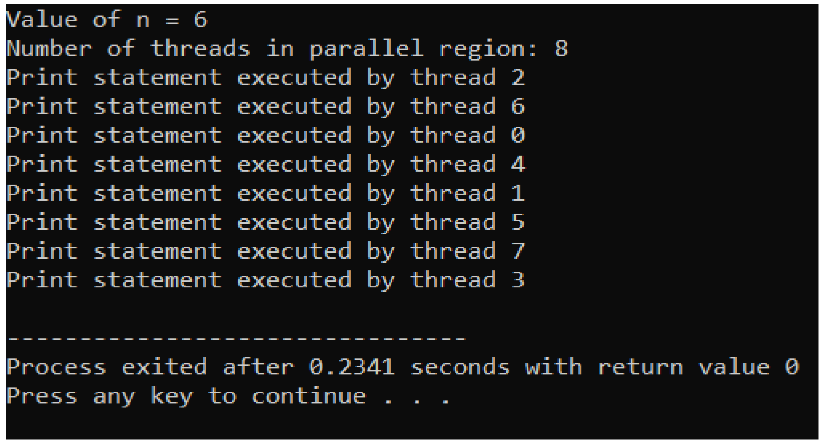
```c
int n=5;
int TID=0;
#pragma omp parallel if (n > 5) default(none) \
private(TID) shared(n)
{
    TID = omp_get_thread_num();
    #pragma omp single
    {
        printf("Value of n = %d\n",n);
        printf("Number of threads in parallel region: %d\n",
        omp_get_num_threads());
    }
    printf("Print statement executed by thread %d\n",TID);
} /*-- End of parallel region --*/
```



```c
int n=6;
int TID=0;
#pragma omp parallel if (n > 5) default(none) \
private(TID) shared(n)
{
    TID = omp_get_thread_num();
    #pragma omp single
    {
        printf("Value of n = %d\n",n);
        printf("Number of threads in parallel region: %d\n",
        omp_get_num_threads());
    }
    printf("Print statement executed by thread %d\n",TID);
} /*-- End of parallel region --*/
```



```
Value of n = 5
Number of threads in parallel region: 1
Print statement executed by thread 0

--------------------------------
Process exited after 0.2263 seconds with return value 0
Press any key to continue . . .
```



```
Value of n = 6
Number of threads in parallel region: 8
Print statement executed by thread 2
Print statement executed by thread 6
Print statement executed by thread 0
Print statement executed by thread 4
Print statement executed by thread 1
Print statement executed by thread 5
Print statement executed by thread 7
Print statement executed by thread 3

--------------------------------
Process exited after 0.2341 seconds with return value 0
Press any key to continue . . .
```

Num_threads Clause

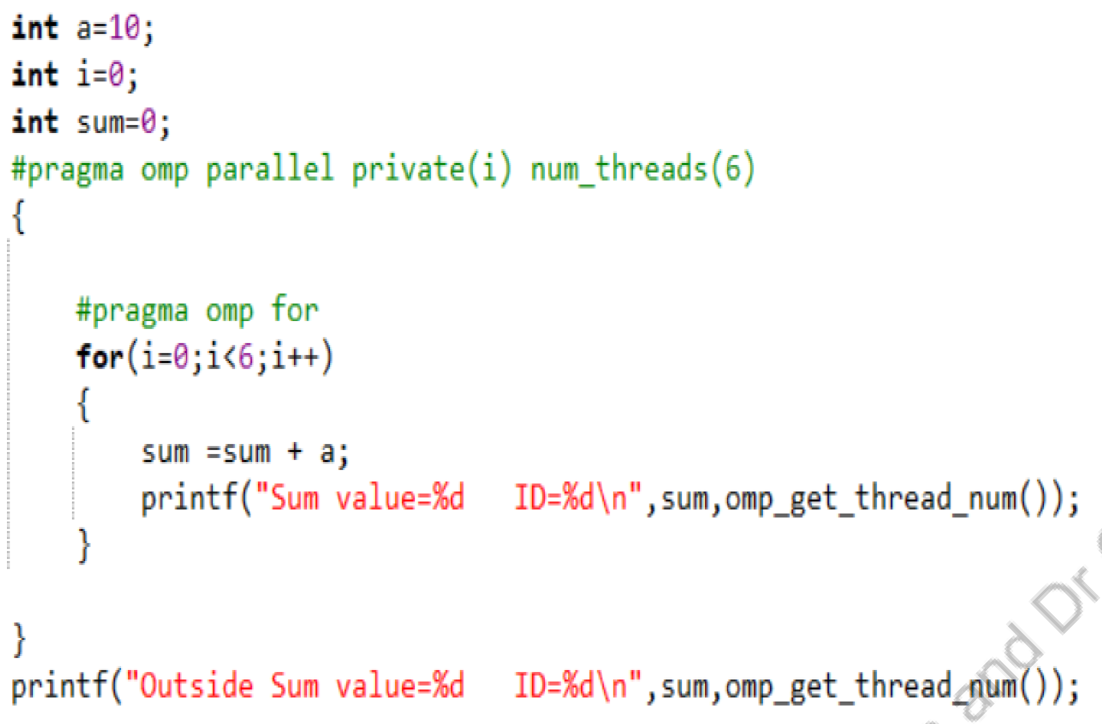
- The num threads clause is supported on the parallel construct only
- Can be used to specify how many threads should be in the team executing the parallel region
- Has higher priority over **omp_set_num_threads(4)**

Reduction Clause

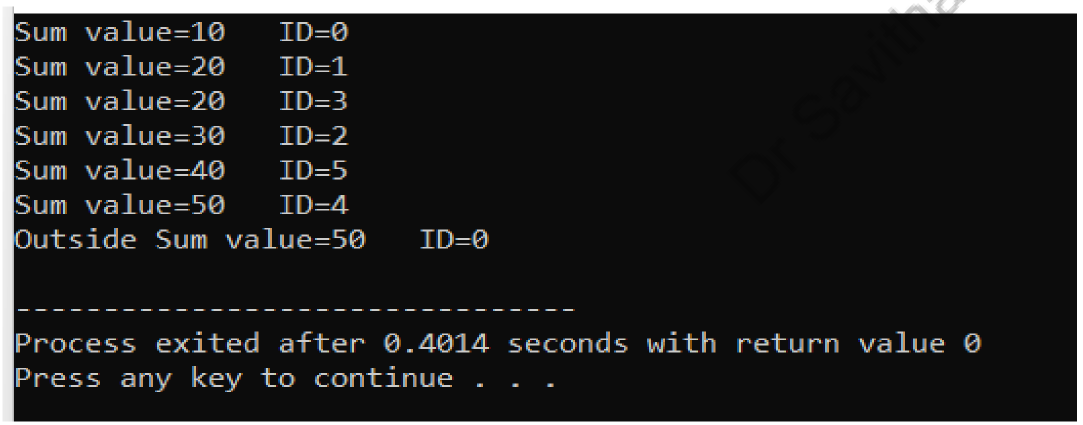
- OpenMP provides the reduction clause for specifying some forms of recurrence calculations
  - They can be performed in parallel without code modification
- The programmer must identify the operations and the variables that will hold the result values

reduction(operator :list)

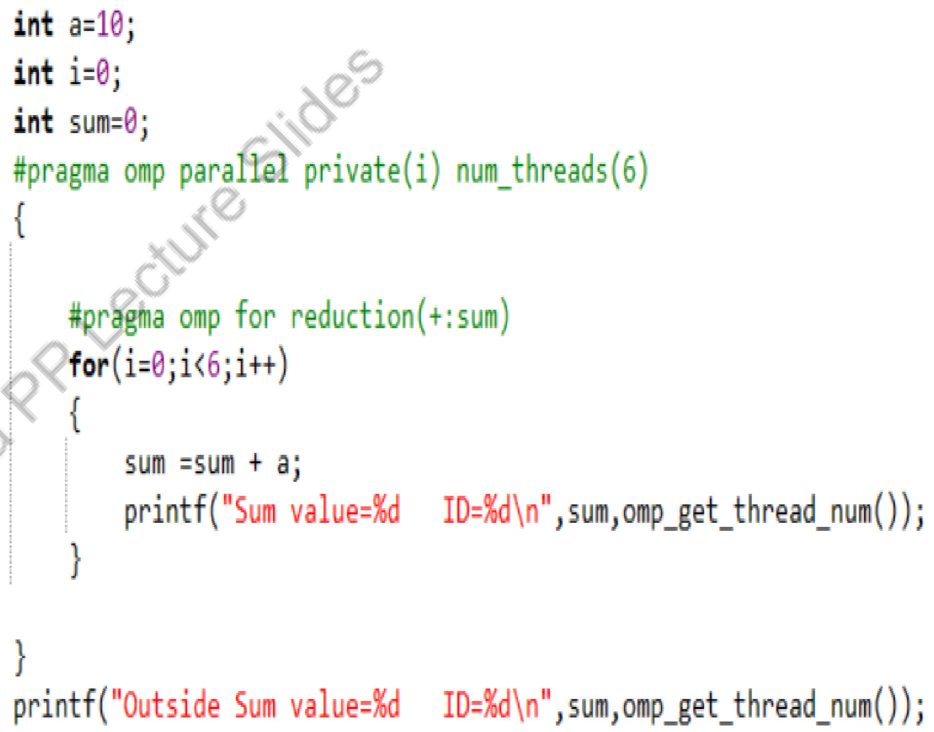
- The order in which thread-specific values are combined is unspecified
  - floating-point data are concerned, there may be numerical differences between the results of a sequential and parallel run



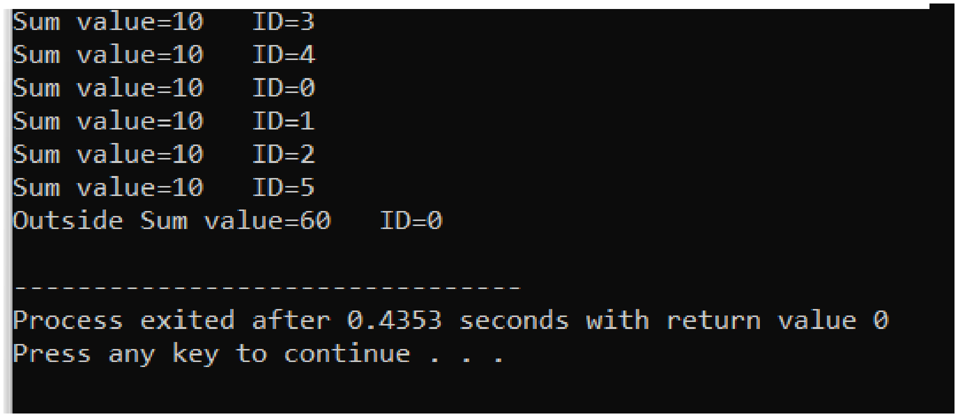
```c
int a=10;
int i=0;
int sum=0;
#pragma omp parallel private(i) num_threads(6)
{

    #pragma omp for
    for(i=0;i<6;i++)
    {
        sum =sum + a;
        printf("Sum value=%d   ID=%d\n",sum,omp_get_thread_num());
    }

}
printf("Outside Sum value=%d   ID=%d\n",sum,omp_get_thread_num());
```



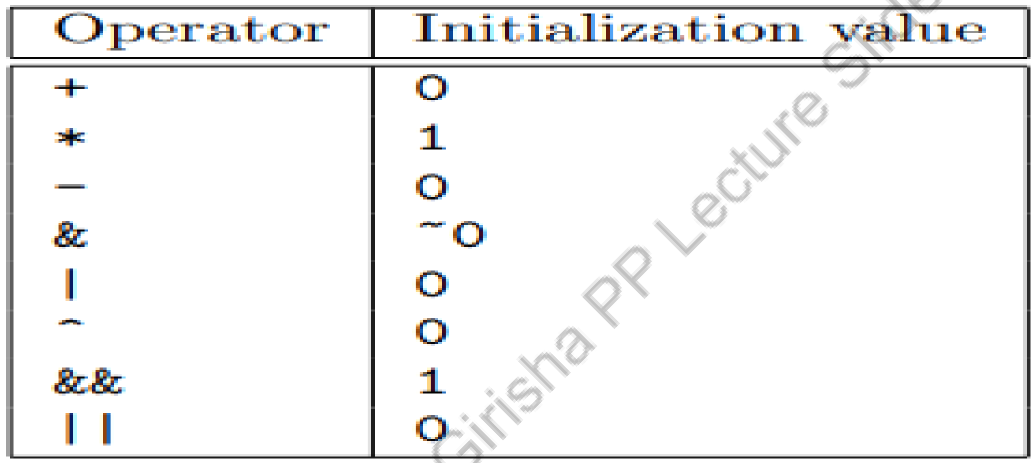
```
Sum value=10   ID=0
Sum value=20   ID=1
Sum value=20   ID=3
Sum value=30   ID=2
Sum value=40   ID=5
Sum value=50   ID=4
Outside Sum value=50   ID=0

--------------------------------
Process exited after 0.4014 seconds with return value 0
Press any key to continue . . .
```



```c
int a=10;
int i=0;
int sum=0;
#pragma omp parallel private(i) num_threads(6)
{

    #pragma omp for reduction(+:sum)
    for(i=0;i<6;i++)
    {
        sum =sum + a;
        printf("Sum value=%d   ID=%d\n",sum,omp_get_thread_num());
    }

}
printf("Outside Sum value=%d   ID=%d\n",sum,omp_get_thread_num());
```



```
Sum value=10   ID=3
Sum value=10   ID=4
Sum value=10   ID=0
Sum value=10   ID=1
Sum value=10   ID=2
Sum value=10   ID=5
Outside Sum value=60   ID=0

--------------------------------
Process exited after 0.4353 seconds with return value 0
Press any key to continue . . .
```



Operator	Initialization value
+	0
*	1
−	0
&	~0
|	0
^	0
&&	1
||	0

- Aggregate types (including arrays), pointer types, and reference types are not supported.
- A reduction variable must not be const-qualified.
- The operator specified on the clause can not be overloaded with respect to the variables that appear in the clause

Copyin Clause

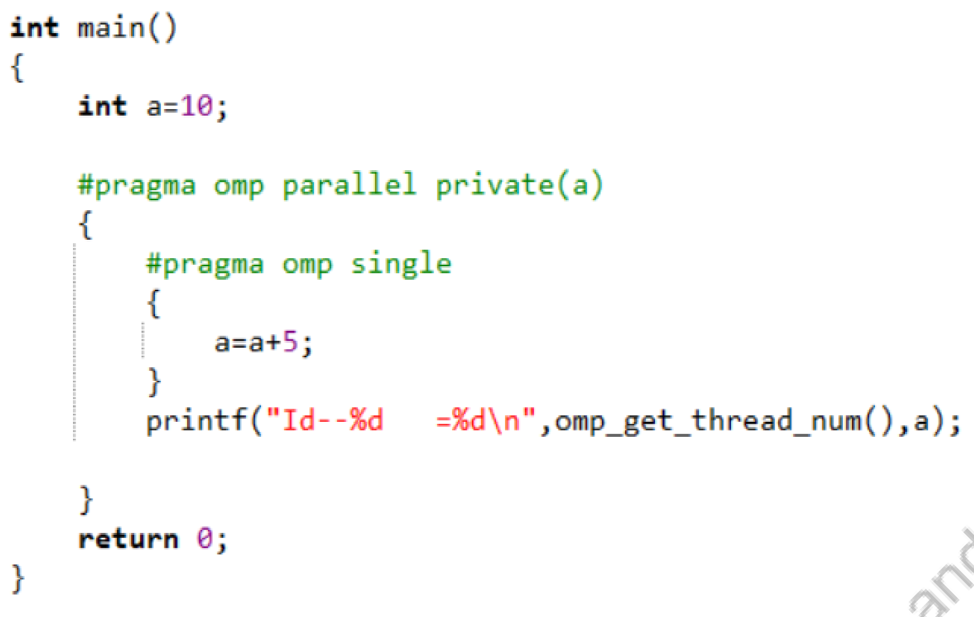
- Allows us to copy the value of the master thread's threadprivate variable(s) to the corresponding threadprivate variables of the other threads
- Threadprivate:
  - Static variables are generally shared by default
  - We can change this by using **threadprivate** clause (We will discuss this later)
- The initial values of private variables are undefined
- The copy is carried out after the team of threads is formed and prior to the start of execution of the parallel region, so that it enables a straightforward initialization of this kind of data object.

Copyprivate Clause

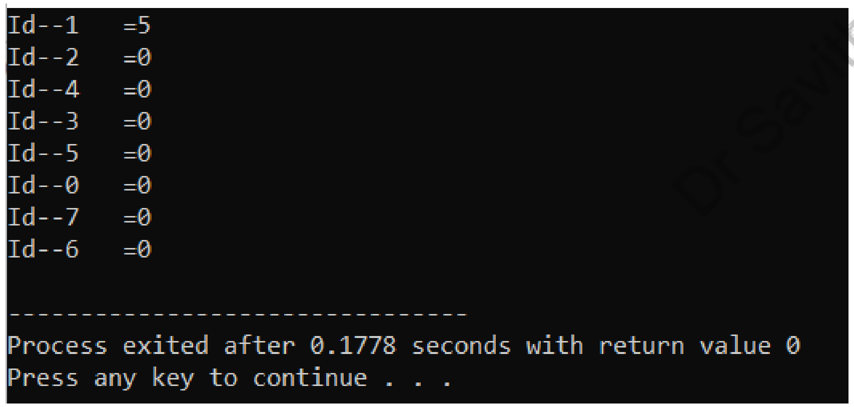
- The **copyprivate clause** is supported on the **single directive** only
- It provides a mechanism for broadcasting the value of a private variable from one thread to the other threads in the team
- **Uses:** one thread read or initialize private data that is subsequently used by the other threads as well
- After the single construct has ended, but before the threads have left the associated barrier, the values of variables specified in the associated list are copied to the other threads
- Since the barrier is essential in this case, the standard prohibits use of this clause in combination with the **nowait clause**

```c
int main()
{
    int a=10;

    #pragma omp parallel private(a)
    {
        #pragma omp single
        {
            a=a+5;
        }
        printf("Id--%d   =%d\n",omp_get_thread_num(),a);

    }
    return 0;
}
```

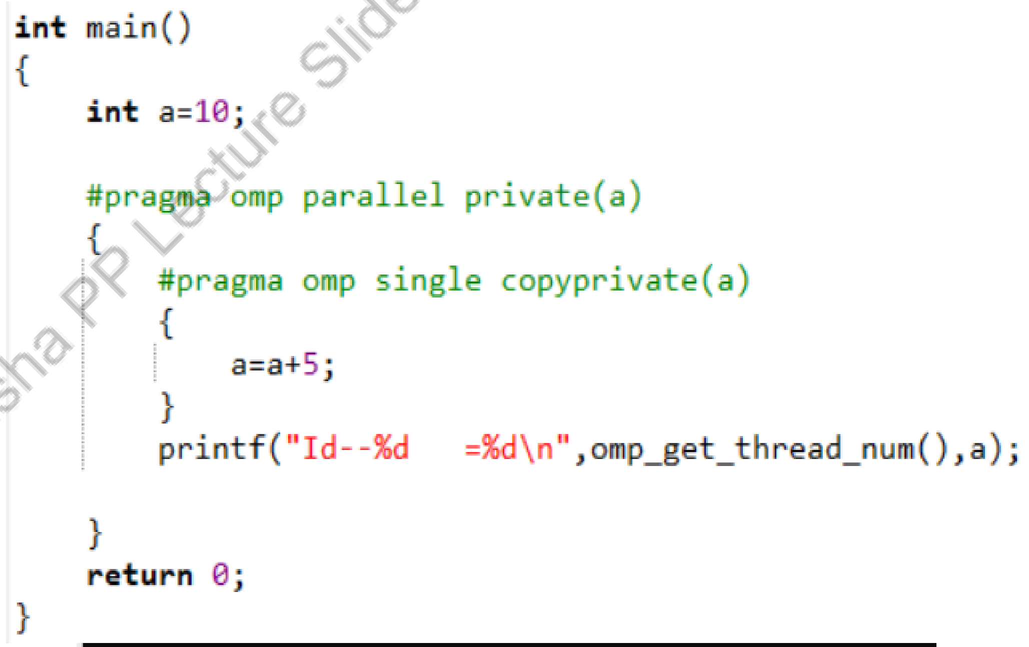
```c
int main()
{
    int a=10;

    #pragma omp parallel private(a)
    {
        #pragma omp single copyprivate(a)
        {
            a=a+5;
        }
        printf("Id--%d   =%d\n",omp_get_thread_num(),a);

    }
    return 0;
}
```



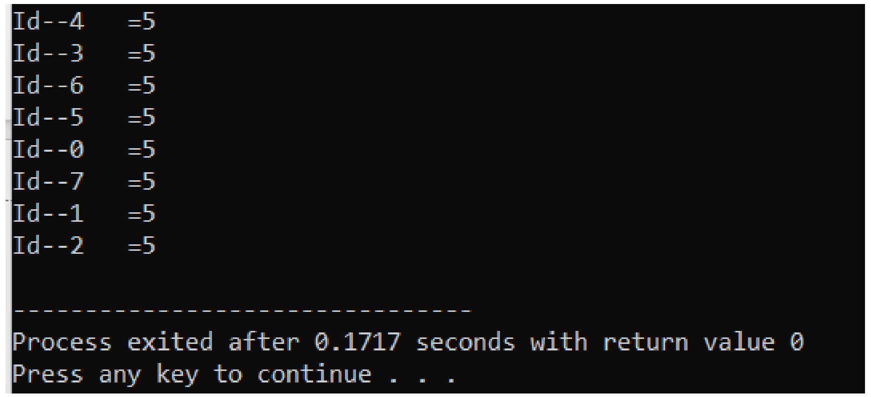
```
int main()
{
    int a=10;

    #pragma omp parallel private(a)
    {
        #pragma omp single
        {
            a=a+5;
        }
        printf("Id--%d   =%d\n",omp_get_thread_num(),a);

    }
    return 0;
}
```



```
Id--1   =5
Id--2   =0
Id--4   =0
Id--3   =0
Id--5   =0
Id--0   =0
Id--7   =0
Id--6   =0

--------------------------------
Process exited after 0.1778 seconds with return value 0
Press any key to continue . . .
```



```
int main()
{
    int a=10;

    #pragma omp parallel private(a)
    {
        #pragma omp single copyprivate(a)
        {
            a=a+5;
        }
        printf("Id--%d   =%d\n",omp_get_thread_num(),a);

    }
    return 0;
}
```



```
Id--4   =5
Id--3   =5
Id--6   =5
Id--5   =5
Id--0   =5
Id--7   =5
Id--1   =5
Id--2   =5

--------------------------------
Process exited after 0.1717 seconds with return value 0
Press any key to continue . . .
```

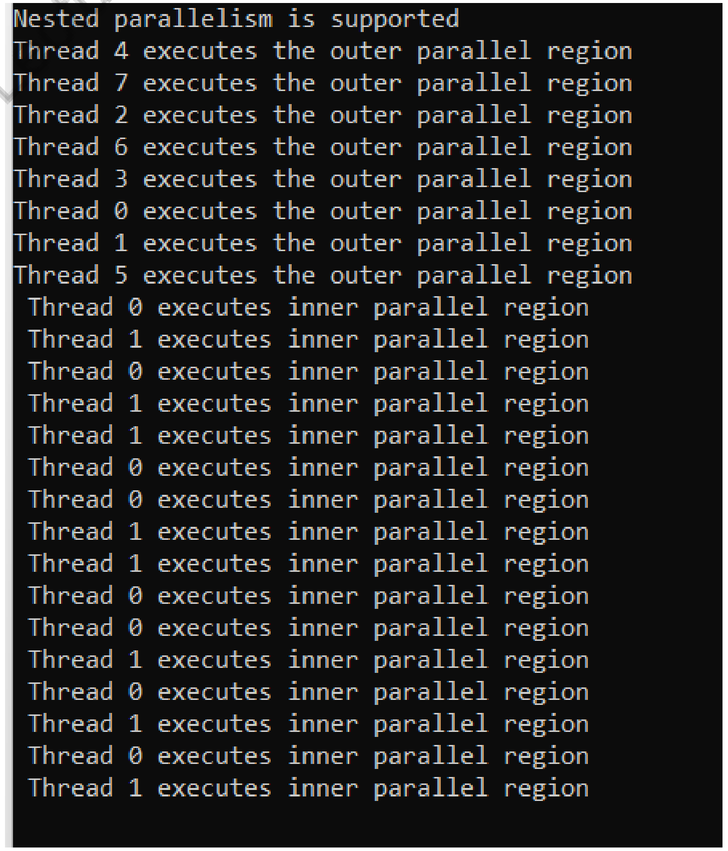
Nested parallelism

- If a thread in a team executing a parallel region encounters another parallel construct, it creates a new team and becomes the master of that new team
  - This is generally referred to in OpenMP as **"nested parallelism"**
- If nested parallelism is not supported
  - parallel constructs that are nested within other parallel constructs will be ignored
  - parallel region serialized (executed by a single thread only)
- **Frequent starting and stopping of parallel regions may introduce a non-trivial performance penalty**

- What will happen if we call **omp_get_thread_num()** function from the nested region?
  - It returns the thread id starting from 0 to one less than the number of threads in the current thread team
  - Thread numbers are no longer unique

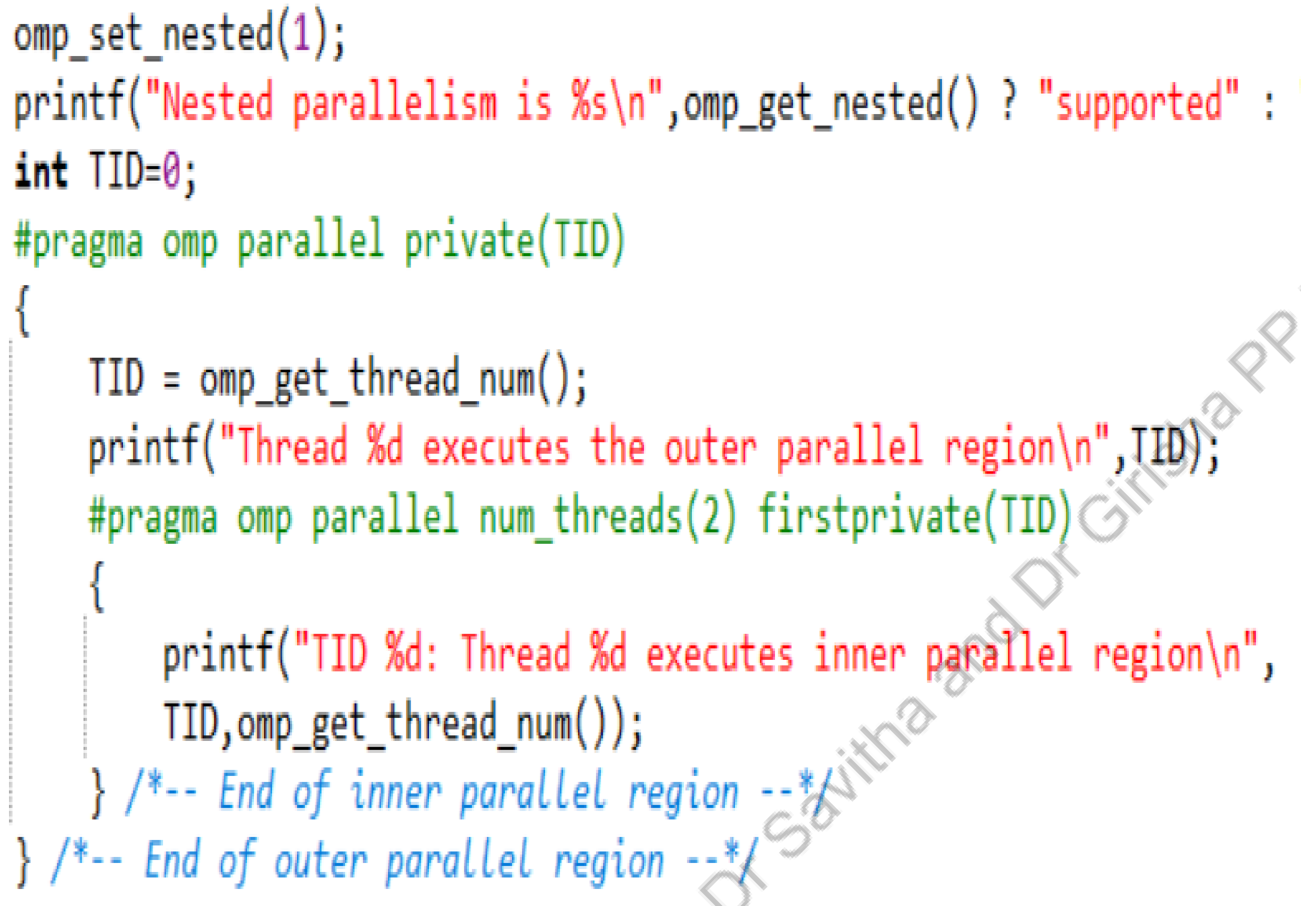
Nested parallelism

```
omp_set_nested(1);
printf("Nested parallelism is %s\n",omp_get_nested() ? "supported" : "not supported");
#pragma omp parallel
{
    printf("Thread %d executes the outer parallel region\n",
    omp_get_thread_num());
    #pragma omp parallel num_threads(2)
        {
        printf(" Thread %d executes inner parallel region\n",
        omp_get_thread_num());
        } /*-- End of inner parallel region --*/
} /*-- End of outer parallel region --*/
```

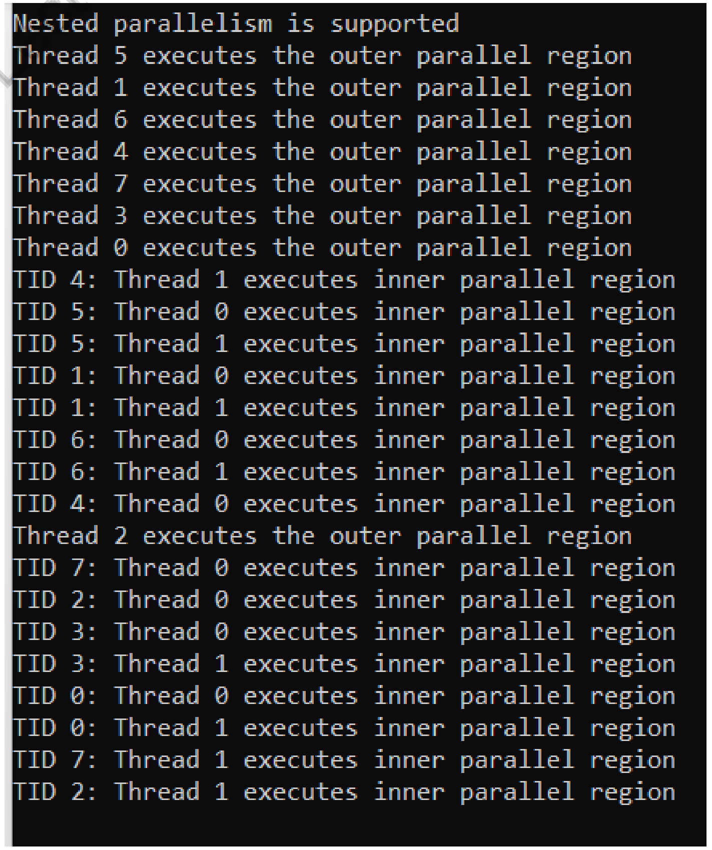


```
Nested parallelism is supported
Thread 4 executes the outer parallel region
Thread 7 executes the outer parallel region
Thread 2 executes the outer parallel region
Thread 6 executes the outer parallel region
Thread 3 executes the outer parallel region
Thread 0 executes the outer parallel region
Thread 1 executes the outer parallel region
Thread 5 executes the outer parallel region
 Thread 0 executes inner parallel region
 Thread 1 executes inner parallel region
 Thread 0 executes inner parallel region
 Thread 1 executes inner parallel region
 Thread 1 executes inner parallel region
 Thread 0 executes inner parallel region
 Thread 0 executes inner parallel region
 Thread 1 executes inner parallel region
 Thread 1 executes inner parallel region
 Thread 0 executes inner parallel region
 Thread 0 executes inner parallel region
 Thread 1 executes inner parallel region
 Thread 0 executes inner parallel region
 Thread 1 executes inner parallel region
 Thread 0 executes inner parallel region
 Thread 1 executes inner parallel region
```

Nested parallelism



```c
omp_set_nested(1);
printf("Nested parallelism is %s\n",omp_get_nested() ? "supported" : "not supported");
int TID=0;
#pragma omp parallel private(TID)
{
    TID = omp_get_thread_num();
    printf("Thread %d executes the outer parallel region\n",TID);
    #pragma omp parallel num_threads(2) firstprivate(TID)
    {
        printf("TID %d: Thread %d executes inner parallel region\n",
        TID,omp_get_thread_num());
    } /*-- End of inner parallel region --*/
} /*-- End of outer parallel region --*/
```



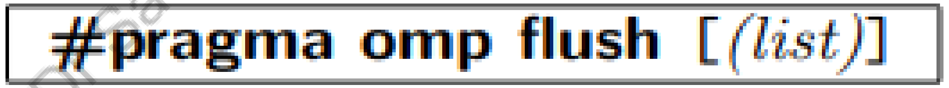
```
Nested parallelism is supported
Thread 5 executes the outer parallel region
Thread 1 executes the outer parallel region
Thread 6 executes the outer parallel region
Thread 4 executes the outer parallel region
Thread 7 executes the outer parallel region
Thread 3 executes the outer parallel region
Thread 0 executes the outer parallel region
TID 4: Thread 1 executes inner parallel region
TID 5: Thread 0 executes inner parallel region
TID 5: Thread 1 executes inner parallel region
TID 1: Thread 0 executes inner parallel region
TID 1: Thread 1 executes inner parallel region
TID 6: Thread 0 executes inner parallel region
TID 6: Thread 1 executes inner parallel region
TID 4: Thread 0 executes inner parallel region
Thread 2 executes the outer parallel region
TID 7: Thread 0 executes inner parallel region
TID 2: Thread 0 executes inner parallel region
TID 3: Thread 0 executes inner parallel region
TID 3: Thread 1 executes inner parallel region
TID 0: Thread 0 executes inner parallel region
TID 0: Thread 1 executes inner parallel region
TID 7: Thread 1 executes inner parallel region
TID 2: Thread 1 executes inner parallel region
```

OpenMP: Flush Directive

- OpenMP memory model distinguishes between shared data and private data:
  - Which is accessible and visible to all threads (**shared data)**
  - Which is local to an individual thread (**private data)**
- If a thread updates shared data, the new values will first be saved in a **register** and then stored back to the **local cache**
  - **Other threads doesn’t have access to these memories immediately**
- **Cache-Coherent machines:** Broadcasts these shared variables to other threads
- The OpenMP standard specifies that all modifications are written back to main memory
- Modifications are thus available to all threads, at synchronization points in the program

OpenMP: Flush Directive

- Between these synchronization points, threads are permitted to have new values for shared variables stored in their local memory rather than in the global shared memory
- This approach is called as **relaxed consistency model**
- Sometimes updated values of shared values must become visible to other threads in-between synchronization points
- The OpenMP API provides the **flush directive** to make this possible
- The purpose of the flush directive is to make a thread's temporary view of shared data consistent with the values in memory



```
#pragma omp flush [(list)]
```

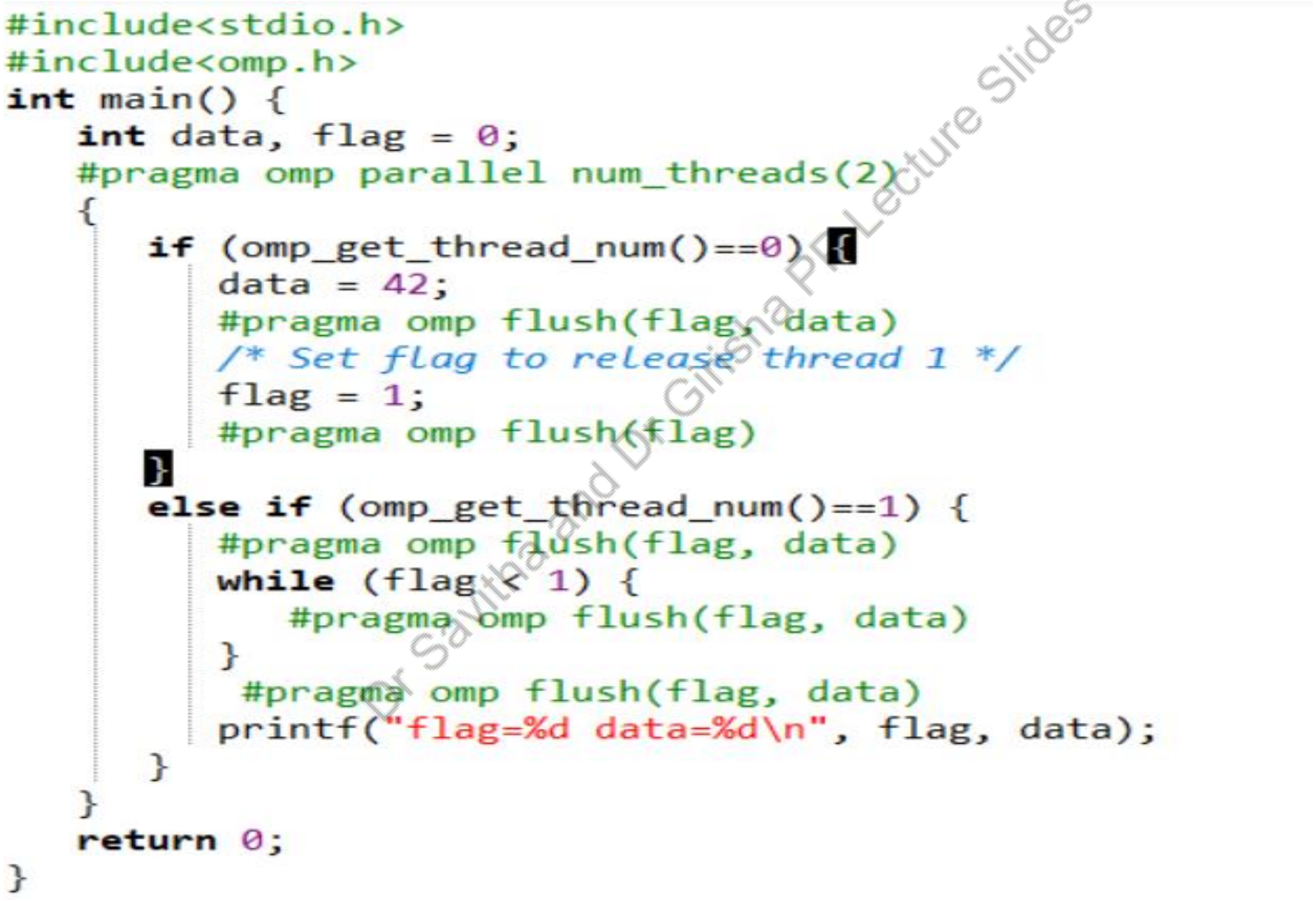
OpenMP: Flush Directive

- The flush operation applies to all variables specified in the list
- If no list is provided, it applies to all thread-visible shared data
- If the flush operation is invoked by a thread that has updated the variables, their new values will be flushed to memory and therefore be accessible to all other threads
- If the construct is invoked by a thread that has not updated a value, it will ensure that any local copies of the data are replaced by the latest value from main memory
- This does not synchronize the actions of different threads: rather, it forces the executing thread to make its shared data values consistent **with shared memory**
- Since the compiler reorders operations to enhance program performance, one cannot assume that the flush operation will remain exactly in the position, relative to other operations, in which it was placed by the programmer

OpenMP: Flush Directive

- Implicit flush operations with no list occur at the following locations
  - All explicit and implicit barriers (e.g., at the end of a parallel region or worksharing construct)
  - Entry to and exit from critical regions
  - Entry to and exit from lock routines

OpenMP: Flush Directive



```
#include<stdio.h>
#include<omp.h>
int main() {
    int data, flag = 0;
    #pragma omp parallel num_threads(2)
    {
        if (omp_get_thread_num()==0) {
            data = 42;
            #pragma omp flush(flag, data)
            /* Set flag to release thread 1 */
            flag = 1;
            #pragma omp flush(flag)
        }
        else if (omp_get_thread_num()==1) {
            #pragma omp flush(flag, data)
            while (flag < 1) {
                #pragma omp flush(flag, data)
            }
             #pragma omp flush(flag, data)
            printf("flag=%d data=%d\n", flag, data);
        }
    }
    return 0;
}
```

OpenMP: Threadprivate Directive

- By default, global data is shared
- Each thread gets a **private or "local"** copy of the specified global variables

```
#pragma omp threadprivate (list)
```

- By default, the threadprivate copies are not allocated or defined

OpenMP Code Structure

- **“Pragma”: stands for “pragmatic information**
- A pragma is a way to communicate the information to the compiler
- The information is non-essential in the sense that the compiler may ignore the information and still produce correct object program.

OpenMP Core Syntax

```
#include "omp.h"
int main ()
{
        int var1, var2, var3;
         // Serial code . . .
        // Beginning of parallel section.
        // Fork a team of threads. Specify variable scoping
         #pragma omp parallel private(var1, var2) shared(var3)
        {
                // Parallel section executed by all threads . . .
                // All threads join master thread and disband
         }
        // Resume serial code . . .
}
```

Thread Creation: Parallel Region Example

```
#include <stdio.h>
#include "omp.h"
int main()
{
    int nthreads, tid;
    #pragma omp parallel num_threads(4) private(tid)
    {
       tid = omp_get_thread_num();
       printf("Hello world from (%d)\n", tid);
       if(tid == 0)
      {
          nthreads = omp_get_num_threads();
          printf("number of threads = %d\n", nthreads);
     }
    } // all threads join master thread and terminates
}
```

Thread Creation: Parallel Region Example

```
#include <stdio.h>
#include "omp.h"
int main()
{
    int nthreads, A[100] , tid;
    // fork a group of threads with each thread having a private tid variable
    omp_set_num_threads(4);
    #pragma omp parallel private (tid)
   {
         tid = omp_get_thread_num();
        foo(tid, A);
   }
// all threads join master thread and terminates
}
```

OpenMP controlling number of threads

- Asking how many cores this program has access to:

```
num = omp_get_num_procs( );
```

- Setting the number of available threads to the exact number of cores available:

```
omp_set_num_threads( omp_get_num_procs( ) );
```

- Asking how many OpenMP threads this program is using right now:

```
num = omp_get_num_threads( );
```

- Asking which thread number this one is:

```
me = omp_get_thread_num( );
```

END