Department of Mathematical and Computational Sciences

Memory Parallelism using OpenMP

Course Teacher: Dr. Pushparaj Shetty D

Preliminary information:

Preferable Coding is “C”

The file is saved as regular ‘C’ file name: filename.c

Compilation using GCC: **gcc -fopenmpfilename.c**
./a.out

Note: Do not forget to include: **#include <omp.h>**

All the required OpenMP syntaxes are available in OpenMP-API-Specification-5.0.pdf. It is openly available for reference.

Introduction to shared memory Parallelism:

- **In a shared memory architecture, different processors are connected with a common address space.**
- **For symmetric multi-processors (SMP), this address space is physically the same memory location.**
- **Concurrent instructions run in different processors while using the same memory.**

Threads:

- **Symmetric multiprocessing (SMP) systems, shared memory parallel computers (and GPU-s too) provide system support for the execution of multiple independent instruction streams. These instruction streams which use data stream from a common address space are known as *threads***

.

Any program will have a certain sequential component in it. Threading can be done only on the parallel part of the program.

The threads are created and destroyed following a fork-join mode

OpenMP- Introduction:

OpenMP (Open Multi-Processing) is an application programming interface (API) which supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran on various shared memory platforms with different instruction-set architectures and operating systems, including Solaris, AIX, HP-UX, Linux, macOS, and Windows.

OpenMP consists of a set of compiler directives, library routines, and environment variables

- **OpenMP provides a portable and scalable platform for programmers to develop parallel programs Programmer can add openMP constructs over sequential codes to convert it into a multi-threaded parallel program**
- **OpenMP is managed by the nonprofit technology consortium OpenMP Architecture**
- **Review Board (or OpenMP ARB) jointly defined by a group of major computer hardware and software vendors**
- **OpenMP has been standardized over last 20 years in SMP programming**

Example

• ***#pragma omp parallel num_threads(4)***

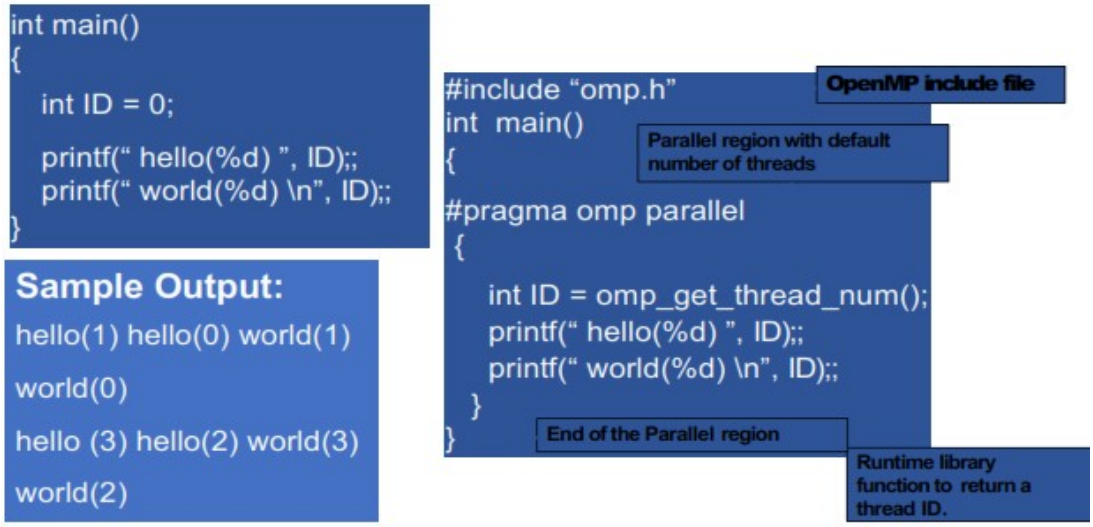
• Function prototypes and types in the file: #include <omp.h>
• Most OpenMP constructs apply to “structured block”

• Structured block: a block of one or more statements with one of entry at the top and one point of exit at the bottom.
• It’s OK to have an exit() within the structured block.
• Compiler Notes:
• Linux or OSx with gcc
• gcc –fopenmp foo.c
• ./a.out

Hello Program

Compilation using GCC: **gcc -fopenmpfilename.c**

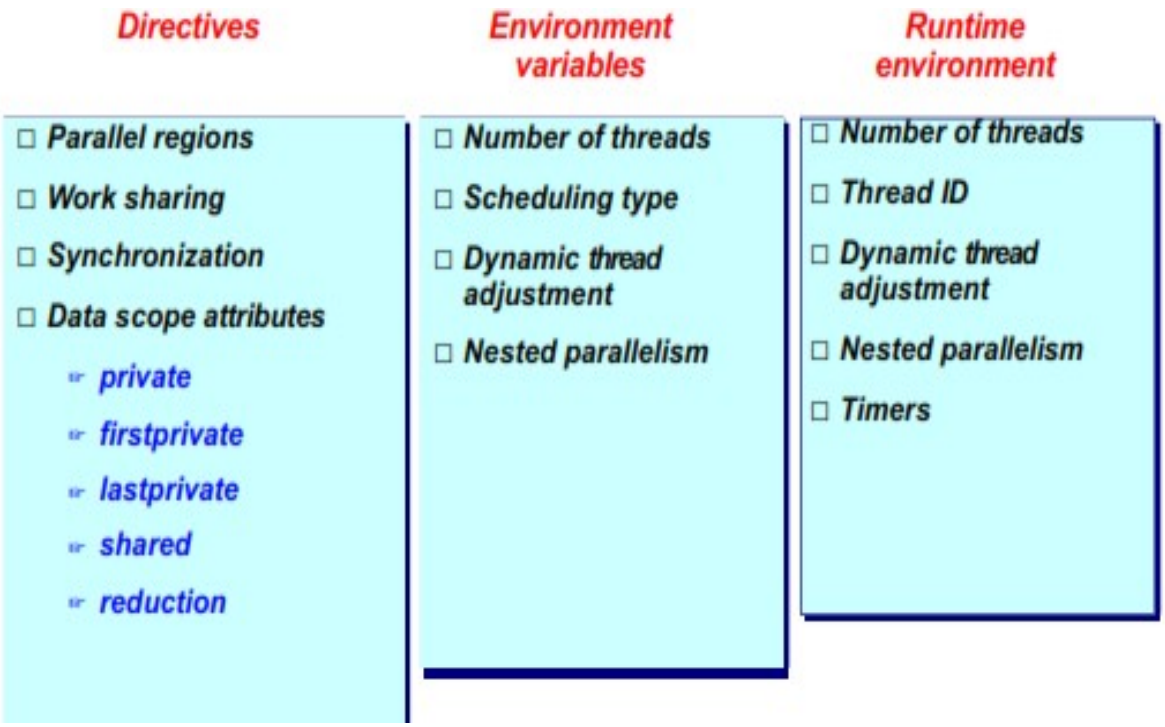
./a.out



Example :

```
#include <stdio.h>
#include <omp.h>
main(){
   int a[2]={-1,-1};
#pragma omp parallel
   {
      a[omp_get_thread_num()]  = omp_get_thread_num();
   }
   printf("a[0] = %d, a[1] = %d",a[0],a[1]);
}
```

Components of OpenMP:



The number of threads set can be checked at command prompt using

echo $OMP_NUM_THREADS

The number of threads can be set at command prompt using

export OMP_NUM_THREADS=4

1.Write a C/C++ simple parallel program to display the *thread_id* and total number of threads.

Aim: To understand and analyze the working of parallel directives and the private clause.

```
/*simpleomp.c*/
#include<stdio.h>
#include<omp.h>
int main()
{
int nthreads,tid;
#pragma omp parallel private(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);
                }
        }
}
```

Execute the program as follows:

```
$gcc –o simple –fopenmp simpleomp.c
$export OMP_NUM_THREADS=2
$./simple
```

Number of threads in a parallel region is determined by the *if* clause, *num_threads(), omp_set_num_threads(), OMP_NUM_THREADS.*

Use these various methods to set number of threads and mention the method of setting the same.

2. Check the output of following program:

Aim : To understand the working of if clause in parallel directive..

```
/*ifparallel.c*/
#include<stdio.h>
#include<omp.h>
int main()
{
int val;
printf("Enter 0: for serial 1: for parallel\n");
scanf("%d",&val);
#pragma omp parallel if(val)
        {
        if(omp_in_parallel())
        printf("Parallel val=%d id= %d\n",val, omp_get_thread_num());
        else
        printf("Serial val=%d id= %d\n",val, omp_get_thread_num());
        }
}
```

3. Observe and record the output of following program

Aim: To understand and analyze shared clause in parallel directive.

```
/*shared.c*/
#include<omp.h>
int main()
{
int x=0;
#pragma omp parallel shared(x)
        {
        int tid=omp_get_thread_num();
        x=x+1;
```

```
        printf(“Thread [%d]\n value of x is %d”,tid,x);
        }
}
```

4. Learn the concept of private(), firstprivate()

```
/*learn.c*/
#include<stdio.h>
#include<omp.h>
int main()
{
int i=10;
printf("Value before pragma i=%d\n",i);
#pragma omp parallel num_threads(4) private(i)
        {
        printf("Value after entering pragma i=%d tid=%d\n",i, omp_get_thread_num());
        i=i+omp_get_thread_num();   //adds thread_id to i
        printf("Value after changing value i=%d tid=%d\n",i, omp_get_thread_num());
        }
printf("Value after having pragma i=%d tid=%d\n",i, omp_get_thread_num());
}
```

*** Note down the result by changing private() to firstprivate().**

5. Demonstration of reduction clause in parallel directive.

```
#include<stdio.h>
#include<omp.h>
void main()
{
int x=0;
#pragma omp parallel num_threads(6) reduction(+:x)
        {
        int id=omp_get_thread_num();
        int threads=omp_get_num_threads();
        x=x+1;
```

```
        printf("Hi from %d\n value of x : %d\n",id,x);
        }
printf("Final x:%d\n",x);
}
```

6. Execute following code and observe the working of threadprivate directive, copyin clause and synchronization directives:

```
#include<stdio.h>
#include<omp.h>
int tid,x;
#pragma omp threadprivate(x,tid)
void main()
{
x=10;
#pragma omp parallel num_threads(4) copyin(x)
        {
        tid=omp_get_thread_num();
                #pragma omp master
                {
                printf("Parallel Region 1 \n");
                x=x+1;
                }
                #pragma omp barrier
        if(tid==1)
        x=x+2;
        printf("Thread % d Value of x is %d\n",tid,x);
        }//#pragma omp barrier
#pragma omp parallel num_threads(4)
        {
                #pragma omp master
                {
```

```
                printf("Parallel Region 2 \n");
                }
        #pragma omp barrier
        printf("Thread %d Value of x is %d\n",tid,x);
}
printf("Value of x in Main Region is %d\n",x);
}
```

DO the following:

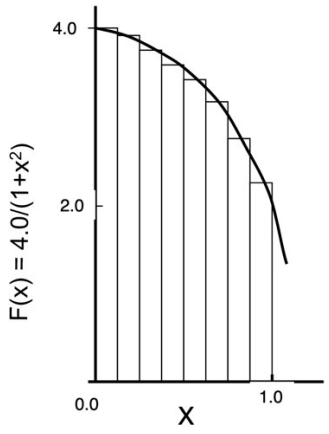
1. Remove copyin clause and check the output.

2. Remove copyin clause and initialize x globally.

Note the observation about threadprivate directive and copyin clause.

7. Compute the value of PI.

Numerical integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Serial PI-program:

```
static long num_steps = 100000;
double step;
int main ()
{           int i;      double x, pi, sum = 0.0;

            step = 1.0/(double) num_steps;

            for (i=0;i< num_steps; i++){
                        x = (i+0.5)*step;
                        sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

The code which numerically approximates pi:

```
 #include<stdio.h>
int main()
{
double pi,x;
int i,N;
pi=0.0;
N=1000;
#pragma omp parallel for
for(i=0;i<=N;i++)
{ x=(double)i/N;
pi+=4/(1+x*x);
}
pi=pi/N;
printf("Pi is %f\n",pi);
}
```

Compile this with gcc main.c -o test, ignoring the -fopenmp options, this means that the **#pragma omp parallel for** will be interpreted as a comment i.e. ignored. We run it and this is the result:

```
Pi is 3.142592
```

Now compile with the -fopenmp option and run:

```
$ gcc test.c -o test   -fopenmp
 prog@abc-system:~$ ./test
Pi is 2.785016
```

[Check for more than one run. You will get different values]

Let's examine what went wrong. Well, by default and as we have not specified it as private, the **variable x is shared**. This means all threads have the same memory address of the variavle X

Therefore, thread i will compute some value at x and store it at memory address &x, thread j will then compute its value of x and store it at &x **BEFORE** thread i has used its value to make its contribution to pi. The threads are all over writing each others values of x because they all have the same memory address for x. Our first correction is that x must be made private:

#pragma omp parallel for private(x)

Secondly, we have a **"Race Condition"** for pi. Let me illustrate this with a simple example. Here is what would ideally happen:

- Thread 1 reads the current value of pi : 0
- Thread 1 increments the value of pi : 1
- Thread 1 stores the new value of pi: 1
- Thread 2 reads the current value of pi: 1
- Thread 2 increments the value of pi: 2
- Thread 2 stores the value of pi: 2

What is actually happening is more like this:

- Thread 1 reads the current value of pi: 0
- Thread 2 reads the current value of pi: 0
- Thread 1 increments pi: 1
- Thread 2 increments pi: 1
- Thread 1 stores its value of pi: 1
- Thread 2 stores its value of pi: 1

The way to correct this is to tell the code to execute the read/write of pi only one thread at a time.

This can be achieved with critical or atomic.

Add **#pragma omp atomic**

Just before pi get's updated and you'll see that it works.

This scenario crops up time and time again where you are updating some value inside a parallel loop so in the end it had its own clause made for it. All the above can be achieved by simply making pi a **reduction variable**.

Reduction

To make pi a reduction variable the code is changed as follows:

```
 int main(void){
double pi,x;
int i,N;
pi=0.0;
N=1000;
#pragma omp parallel for private(x) reduction(+:pi)
for(i=0;i<=N;i++){
x=(double)i/N;
pi+=4/(1+x*x);
}
pi=pi/N;
printf("Pi is %f\n",pi);
```

8. The OpenMP omp critical pragma uses the following syntax:

Syntax:

The OpenMP omp critical pragma uses the following syntax:

#pragma omp critical [(*name*)]
< *C/C++ structured block* >

Within a parallel region, there may exist subregions of code that will not execute properly when executed by multiple threads simultaneously. This is often due to a shared variable that is written and then read again.

The omp critical pragma defines a subsection of code within a parallel region, referred to as a *critical section*, which will be executed one thread at a time. The first thread to arrive at a critical section will be the first to execute the code within the section. The second thread to arrive will not begin execution of statements in the critical section until the first thread has exited the critical section. Likewise, each of the remaining threads will wait its turn to execute the statements in the critical section.

Demonstration of critical pragma

```
#include <stdlib.h>
main(){
int a[100][100], mx=-1, lmx=-1, i, j;
    for (j=0; j<100; j++)
    for (i=0; i<100; i++)
        a[i][j]=1+(int)(10.0*rand()/(RAND_MAX+1.0));
#pragma omp parallel private(i) firstprivate(lmx)
    {
#pragma omp for
```

```
        for (j=0; j<100; j++)
        for (i=0; i<100; i++)
                lmx = (lmx > a[i][j]) ? lmx : a[i][j];
#pragma omp critical
        mx = (mx > lmx) ? mx : lmx;
    }
    printf ("max value of a is %d\n",mx);
}
```

Note: Check witout critical pragma

9. omp master

The OpenMP omp master pragma uses the following syntax:

```
#pragma omp master
< C/C++ structured block >
```

In a parallel region of code, there may be a sub-region of code that should execute only on the master thread. Instead of ending the parallel region before this subregion, and then starting it up again after this subregion, the omp master pragma allows the user to conveniently designate code that executes on the master thread and is skipped by the other threads. There is no implied barrier on entry to or exit from a master section. Nested master sections are ignored. Branching into or out of a master section is not supported.

Demonstration of master pragma

```
#include <stdio.h>
#include <omp.h>
main(){
int a[2]={-1,-1};
#pragma omp parallel
        {
        a[omp_get_thread_num()]  = omp_get_thread_num();
#pragma omp master
        printf("YOU SHOULD ONLY SEE THIS ONCE\n");
    }
    printf("a[0]=%d, a[1]=%d\n",a[0],a[1]);
}
```

10. omp single

The OpenMP omp single pragma uses the following syntax:

```
#pragma omp single [Clauses]
< C/C++ structured block >
```

Clauses:

```
private(list)
firstprivate(list)
nowait
```

In a parallel region of code, there may be a subregion of code that will only execute correctly on a single thread. Instead of ending the parallel region before this subregion, and then starting it up again after this subregion, the omp single pragma allows the user to conveniently designate

code that executes on a single thread and is skipped by the other threads. There is an implied
barrier on exit from a single process section unless the optional nowait clause is specified.

11. Illustration of omp for

```
#include <stdio.h>
#include <math.h>
main(){
float a[1000], b[1000];
int i;
    for (i=0; i<1000; i++)
        b[i] = i;
#pragma omp parallel
    {
#pragma omp for
        for (i=0; i<1000; i++)
            a[i] = sqrt(b[i]);
        ...
    }
    ...
  }
```

Schedule(type [,chunk]

SCHEDULE: Describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent.

STATIC

Loop iterations are divided into pieces of size *chunk* and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.

DYNAMIC

Loop iterations are divided into pieces of size *chunk*, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.

GUIDED

Iterations are dynamically assigned to threads in blocks as threads request them until no blocks remain to be assigned. Similar to DYNAMIC except that the block size decreases each time a parcel of work is given to a thread. The size of the initial block is proportional to:

number_of_iterations / number_of_threads

Subsequent blocks are proportional to

number_of_iterations_remaining / number_of_threads

The chunk parameter defines the minimum block size. The default chunk size is 1.

RUNTIME

The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause.

AUTO

The scheduling decision is delegated to the compiler and/or runtime system.

For simplicity, we assume that we have a loop of 16 iterations, which has been parallelized by OpenMP, and that we are about to execute that loop using 2 threads.

In **default scheduling**

- thread 1 is assigned to do iterations 1 to 8;
- thread 2 is assigned to do iterations 9 to 16.

In **static scheduling**, using a "chunksize" of 4:

- thread 1 is assigned to do iterations 1 to 4 and 9 to 12.
- thread 2 is assigned to do iterations 5 to 8 and 13 to 16.

In **dynamic scheduling**, using a "chunksize" of 3:

- thread 1 is assigned to do iterations 1 to 3.

- thread 2 is assigned to do iterations 4 to 6.

The next chunk is iterations 7 to 9, and will be assigned to whichever thread finishes its current work first, and so on until all work is completed.

Consider following example program:

This program explores the use of the for work-sharing construct. The program provided here adds two vectors together using a work-sharing approach to assign work to threads:

```
#include<omp.h>
#include<stdio.h>
#include<stdlib.h>
#define CHUNKSIZE 10
#define N 100
int main (int argc, char *argv[]) {
int nthreads, tid, i, chunk;
float a[N], b[N], c[N];
for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0; // initialize arrays
chunk = CHUNKSIZE;
#pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid) {
        tid = omp_get_thread_num();
        if (tid == 0)
        {
                nthreads = omp_get_num_threads();
                printf("Number of threads = %d\n", nthreads);
        }
                printf("Thread %d starting...\n",tid);
                #pragma omp for schedule(static,chunk)
                for (i=0; i<N;i++)
                {
                        c[i]=a[i]+b[i];
                        printf(“Thread %d: c[%d]=%f\n”,tid,i,c[i]);
                }
```

```
} /*end of parallel section*/
}
```

i) Observe following results from executing above program with **schedule(static,chunk)**.

a) Note down the range of data elements provided for each thread by setting number of threads =5 and chunk size=10

b) Note down the range of data elements provided for each thread by setting number of threads =5 and chunk size =25

ii) Observe following results from executing above program with **schedule(dynamic,chunk)**.

a) Note down the range of data elements provided for each thread by setting number of threads =5 and chunk size=10

b) Note down the range of data elements provided for each thread by setting number of threads =8 and chunk size =10

iii) Observe following results from executing above program with **schedule(guided,chunk)**.

a) Note down the range of data elements provided for each thread by setting number of threads =5 and chunk size=10

b) Note down the range of data elements provided for each thread by setting number of threads =5 and chunk size=5

c) Note down the range of data elements provided for each thread by setting number of threads =8 and chunk size =5

Briefly explain what you have understood by running the programs with various scheduling methods. Also mention, when each type of scheduling is useful in programming.

12. omp barrier

The OpenMP omp barrier pragma uses the following syntax:

```
#pragma omp barrier
```

There may be occasions in a parallel region when it is necessary that all threads complete work to that point before any thread is allowed to continue. The omp barrier pragma synchronizes all threads at such a point in a program. Multiple barrier points are allowed within a parallel region. The omp barrier pragma must either be executed by all threads executing the parallel region or by none of them.

13. omp parallel for

The omp parallel for pragma uses the following syntax.

```
#pragma omp parallel for [clauses]
< C/C++ for loop to be executed in parallel >
```

Clauses:

private(*list*) shared(*list*)
default(shared | none)
firstprivate(*list*)
lastprivate(*list*)
reduction(*operator*: *list*)
copyin (*list*)
if (*scalar_expression*)
ordered
schedule (*kind*[, *chunk*])

The semantics of the omp parallel for pragma are identical to those of a parallel region containing only a single parallel for loop and pragma.

14. omp sections

The omp sections pragma uses the following syntax:

#pragma omp sections [*Clauses*]
{
  [#pragma omp section]
   < *C/C++ structured block executed by processor i* >
  [#pragma omp section]
   < *C/C++ structured block executed by processor j* >
   …
}

Clauses:

private (*list*)
firstprivate (*list*)
lastprivate (*list*)
reduction(*operator*: *list*)
nowait

The omp sections pragma defines a non-iterative work-sharing construct within a parallel region. Each section is executed by a single thread. If there are more threads than sections, some threads will have no work and will jump to the implied barrier at the end of the construct. If there are more sections than threads, one or more threads will execute more than one section.

An omp section pragma may only appear within the lexical extent of the enclosing omp sections pragma. In addition, the code within the omp sections pragma must be a structured block, and the code in each omp section must be a structured block.

15. omp parallel sections

The omp parallel sections pragma uses the following syntax:

```
#pragma omp parallel sections [clauses]
{
   [#pragma omp section]
    < C/C++ structured block executed by processor i >
   [#pragma omp section]
    < C/C++ structured block executed by processor j >
   …
}
```

Clauses:

private(*list*)

shared(*list*)

default(shared | none)

firstprivate(*list*)

lastprivate (*list*)

reduction({*operator*: *list*)

copyin (*list*)

if (*scalar_expression*)

nowait

The omp parallel sections pragma defines a non-iterative work-sharing construct without the need to define an enclosing parallel region. Semantics are identical to a parallel region
containing only an
omp sections pragma and the associated structured block.

16. omp ordered

The OpenMP ordered pragma uses the following syntax:

```
#pragma omp ordered
< C/C++ structured block >
```

The ordered pragma can appear only in the dynamic extent of a for or parallel for pragma that includes the ordered clause. The structured code block appearing after the ordered pragma is executed by only one thread at a time, and in the order of the loop iterations. This sequentializes the ordered code block while allowing parallel execution of statements outside the code block. The following additional restrictions apply to the ordered pragma:

• The ordered code block must be a structured block. It is illegal to branch into or out of the block.

• A given iteration of a loop with a DO directive cannot execute the same ORDERED directive more than once, and cannot execute more than one
ORDERED directive.

17. omp atomic

The omp atomic pragma uses the following syntax:

#pragma omp atomic

< *C/C++ expression statement* >

The omp atomic pragma is semantically equivalent to subjecting the following single *C*/*C++*
expression statement to an
omp critical pragma. The expression statement must be of one of
the following forms:

• *x* <*binary_operator*>= *expr*

• *x*++

• ++*x*

• *x*--

• --*x*

where *x* is a scalar variable of intrinsic type, *expr* is a scalar expression that does not reference *x*,
<*binary_operator*> is not overloaded and is one of
+, *, -, /, &, ^, |, << or >>.

18. omp threadprivate

The omp threadprivate pragma uses the following syntax:

#pragma omp threadprivate (*list*)

Where *list* is a list of variables to be made private to each thread but global within the thread.

This pragma must appear in the declarations section of a program unit after the declaration of any
variables listed.

Run-time Library Routines

User-callable functions are available to the OpenMP *C*/*C*++ programmer to query and alter the parallel execution environment. Any program unit that invokes these functions should include the statement

#include <omp.h>.
The omp.h include file contains definitions for each of the
C/C++ library routines and two required type definitions.

#include <omp.h>
int omp_get_num_threads(void);
returns the number of threads in the team executing the parallel region from which it is called.
When called from a serial region, this function returns 1. A nested parallel region is the same as a single parallel region. By default, the value returned by this function is equal to the value of the
environment variable

OMP_NUM_THREADS or to the value set by the last previous call to the omp_set_num_threads() function defined below.

#include <omp.h>
void omp_set_num_threads(int num_threads);
sets the number of threads to use for the next parallel region. This function can only be called from a serial region of code. If it is called from within a parallel region, or within a function that is called from within a parallel region, the results are undefined. This function has precedence over the OMP_NUM_THREADS environment variable.

#include <omp.h>
int omp_get_thread_num(void);

returns the thread number within the team. The thread number lies between 0 and omp_get_num_threads()-1. When called from a serial region, this function returns 0. A nested parallel region is the same as a single parallel region.

```
#include <omp.h>
int omp_get_max_threads(void);
```

returns the maximum value that can be returned by calls to omp_get_num_threads(). If omp_set_num_threads() is used to change the number of processors, subsequent calls to omp_get_max_threads() will return the new value. This function returns the maximum value whether executing from a parallel or serial region of code.

```
#include <omp.h>
int omp_get_num_procs(void);
```

returns the number of processors that are available to the program.

```
#include <omp.h>
int omp_in_parallel(void);
```

returns non-zero if called from within a parallel region and zero if called outside of a parallel region. When called from within a parallel region that is serialized, for example in the presence of an if clause evaluating to zero, the function will return zero.

```
#include <omp.h>
void omp_set_dynamic(int dynamic_threads);
```

is designed to allow automatic dynamic adjustment of the number of threads used for execution of parallel regions. This function is recognized, but currently has no effect.

```
#include <omp.h>
int omp_get_dynamic(void);
```

is designed to allow the user to query whether automatic dynamic adjustment of the number of threads used for execution of parallel regions is enabled. This function is recognized, but currently always returns zero.

```
#include <omp.h>
void omp_set_nested(int nested);
```

is designed to allow enabling/disabling of nested parallel regions. This function is recognized, but currently has no effect.

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

int omp_get_nested(void);
is designed to allow the user to query whether dynamic adjustment of the number of threads available for execution of parallel regions is enabled. This function is recognized, but currently always returns zero
.
#include <omp.h>
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
initializes a lock associated with the variable *lock* for use in subsequent calls to lock routines.
This initial state of *lock* is unlocked. It is illegal to make a call to this routine if *lock* is already associated with a lock.

#include <omp.h>
void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
disassociates a lock associated with the variable *lock*.

#include <omp.h>
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);

causes the calling thread to wait until the specified lock is available. The thread gains ownership of the lock when it is available. It is illegal to make a call to this routine if *lock* has not been
associated with a lock.

#include <omp.h>
void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);

causes the calling thread to release ownership of the lock associated with *lock*. It is illegal to make a call to this routine if *lock* has not been associated with a lock.

```
#include <omp.h>
int omp_test_lock(omp_lock_t *lock);
int omp_test_nest_lock(omp_nest_lock_t *lock);
```

causes the calling thread to try to gain ownership of the lock associated with *lock*. The function returns non-zero if the thread gains ownership of the lock, and zero otherwise. It is illegal to make a call to this routine if *lock* has not been associated with a lock.

Environment Variables

OMP_NUM_THREADS - specifies the number of threads to use during execution of parallel regions.
The default value for this variable is 1.

Note: *OMP_NUM_THREADS threads will be used to execute the program regardless of the number of physical processors available in the system. As a result, you can run programs using more threads than physical processors and they will execute correctly. However, performance of programs executed in this manner can be unpredictable, and oftentimes will be inefficient*

OMP_SCHEDULE - specifies the type of iteration scheduling to use for omp for and omp parallel for loops that include the schedule(runtime) clause. The default value for this variable is“static”. If the optional chunk size is not set, a chunk size of 1 is assumed .
Examples of the use of OMP_SCHEDULE are as follows:
$ setenv OMP_SCHEDULE “static, 5”
$ setenv OMP_SCHEDULE “guided, 8”
$ setenv OMP_SCHEDULE “dynamic”

References:

1. https://www.openmp.org/
2. https://computing.llnl.gov/tutorials/openMP/