Assignment : Introduction to OpenMP

1. What is OpenMP?

OpenMP is an Application Program Interface (API), jointly defined by a group of major computer hardware and software vendors. OpenMP provides a portable, scalable model for developers of shared memory parallel applications. The API supports C/C++ and Fortran on a wide variety of architectures.

2. OpenMP Components:

2.1 Compiler Directives and Clauses:

Compiler directives appear as comments in your source code and are ignored by compilers unless you tell them otherwise - usually by specifying the appropriate compiler flag, as discussed in the Compiling section later.

- OpenMP compiler directives are used for various purposes:

- Spawning a parallel region
- Dividing blocks of code among threads
- Distributing loop iterations between threads
- Serializing sections of code
- Synchronization of work among threads

- Compiler directives have the following syntax:

sentinel   directive-name   [clause, ...]

2.2 Runtime Libraries:

The OpenMP API includes an ever-growing number of run-time library routines.

- These routines are used for a variety of purposes:

- Setting and querying the number of threads
- Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the thread team size
- Setting and querying the dynamic threads feature
- Querying if in a parallel region, and at what level
- Setting and querying nested parallelism
- Setting, initializing and terminating locks and nested locks
- Querying wall clock time and resolution

2.3 Environment Variables (shared, private, first private, etc.):

OpenMP provides several environment variables for controlling the execution of parallel code at run-time.

- These environment variables can be used to control such things as:

- Setting the number of threads
- Specifying how loop interations are divided

- Binding threads to processors
- Enabling/disabling nested parallelism; setting the maximum levels of nested parallelism
- Enabling/disabling dynamic threads
- Setting thread stack size
- Setting thread wait policy

2.4 OpenMP Basic Syntax:

```
 #include <omp.h>

  main ()  {

  int var1, var2, var3;

  Serial code
       .
       .
       .

  Beginning of parallel region. Fork a team of threads.
  Specify variable scoping

  #pragma omp parallel private(var1, var2) shared(var3)
    {

    Parallel region executed by all threads
           .
    Other OpenMP directives
           .
    Run-time Library calls
           .
    All threads join master thread and disband

    }

  Resume serial code
       .
       .
       .

  }
```

2.5 Sample program:

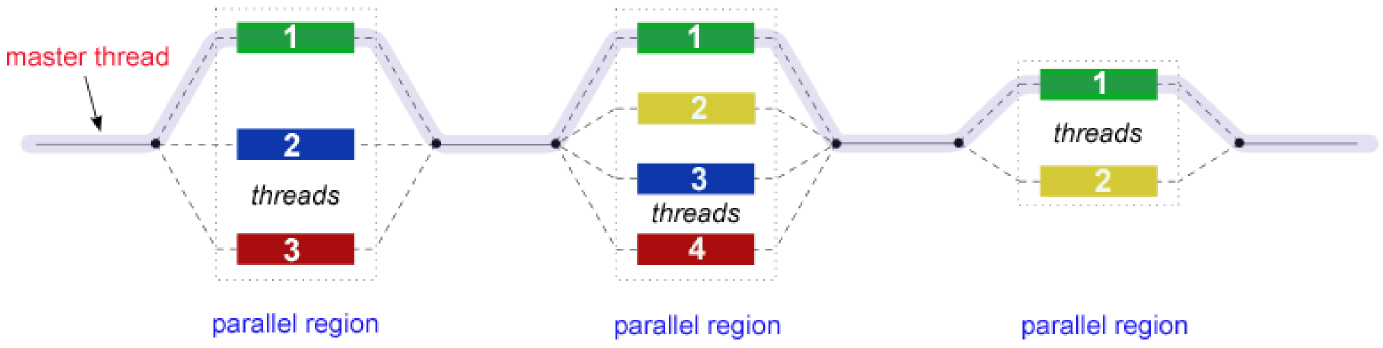
```
#include <omp.h>

 main(int argc, char *argv[]) {
```

```
int nthreads, tid;

/* Fork a team of threads with each thread having a private tid variable */
#pragma omp parallel private(tid)
 {

 /* Obtain and print thread id */
 tid = omp_get_thread_num();
 printf("Hello World from thread = %d\n", tid);

 /* Only master thread does this */
 if (tid == 0)
   {
   nthreads = omp_get_num_threads();
   printf("Number of threads = %d\n", nthreads);
   }

 }  /* All threads join master thread and terminate */

}
```

3. OpenMP Programming Model

Fork - Join Model:

- OpenMP uses the fork-join model of parallel execution



- All OpenMP programs begin as a single process: the **master thread**. The master thread executes sequentially until the first **parallel region** construct is encountered.
- **FORK:** the master thread then creates a team of parallel ***threads***.
- The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads.
- **JOIN:** When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.
- The number of parallel regions and the threads that comprise them are arbitrary.

4. Creation of Threads

A parallel region is a block of code that will be executed by multiple threads. This is the fundamental OpenMP parallel construct.
Format in C/C++:

```
#pragma omp parallel [clause ...]  newline
      if (scalar_expression)
      private (list)
      shared (list)
      default (shared | none)
      firstprivate (list)
      reduction (operator: list)
      copyin (list)
      num_threads (integer-expression)


structured_block
```

- When a thread reaches a PARALLEL directive, it creates a team of threads and becomes the master of the team. The master is a member of that team and has thread number 0 within that team.
- Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.
- There is an implied barrier at the end of a parallel region. Only the master thread continues execution past this point.
- If any thread terminates within a parallel region, all threads in the team will terminate, and the work done up until that point is undefined.
- The number of threads in a parallel region is determined by the following factors, in order of precedence:
    1. Evaluation of the **IF** clause
    2. Setting of the **NUM_THREADS** clause
    3. Use of the **omp_set_num_threads()** library function
    4. Setting of the **OMP_NUM_THREADS** environment variable
    5. Implementation default - usually the number of CPUs on a node, though it could be dynamic.
- Threads are numbered from 0 (master thread) to N-1

5. Synchrozation concept:

Synchronization is used to impose order constraints and to protect access to shared data. There are two levels of synchronization applicable to OpenMP. *High level synchronization* involves high-level compiler directive instructions, while *Low level synchronization* is similar to thread level locking mechanism.

High Level Synchronization

1. **Critical**
This is a high level construct for mutual exclusion making sure that only one thread is able to enter a critical region at a time.

```
#pragma omp parallel
 {
     int i, id, nthrds, l;
         #pragma omp single
     nthrds = omp_get_num_threads();
     for (i = omp_get_thread_num(); i < niters; i+nthrds) {
        l = local();
         #pragma omp critical my_crit
        global(l); // Threads wait their turn to call global()
     }
 }
```

OMP critical clause allows an optional name field – 'my_crit' above. This is important because OMP treats all critical sections with the same name as being part of the same critical section. For example, if we have more than one critical section with the same name, and a thread enters one of them, all other threads will be blocked from entering the other critical sections until the first thread exits the critical section. Similarly, all the unnamed critical sections will be treated as being part of the same shared region. Example, all 'cout' operations can be placed in unnamed critical section.

2. **Atomic**

Atomic provides similar mutual exclusion as the above critical pragma but it only applies to the update of a memory location. The OMP critical section is very general and there is an overhead every time a thread enters and exits the critical section. An atomic operation has much lower overhead. It relies on the hardware to do the atomic operation.

```
#pragma omp parallel
{
        double tmp, B;
        B = drool();
        #pragma omp atomic
        X += tmp;
}
```

3. **Barrier**

OMP barrier is a very important concept in synchronization. It provides a synchronization point at which threads in a parallel region will wait until all other threads in that section reach the same point. Statement execution past the omp barrier then continues in parallel. There are two kinds of barriers: *implicit* and *explicit*. Implicit barriers are at the end of structured block for parallel or any worksharing constructs (say, parallel for, etc.). If the user do not need implicit barriers, he can specify "nowait" clause at the end of worksharing pragma directive. Explicit barrier can be placed using directive clause "barrier".

```
#pragma omp parallel
{
    int id = omp_get_thread_num();
    A[id] = big_calc1(id);
#pragma omp barrier
#pragma omp for
    for (i=0; i < N; i++) {
        C[i] = big_calc3(i,A);
    } // implicit barrier
#pragma omp for nowait
    for (i=0; i < N; i++) {
        A[id] = big_calc4(id);
    } // no implicit barrier due to nowait
} // implicit barrier at the end of parallel region
```

This directive should not be used following an if, while, do, switch or label. The directive specifies a point where all or none threads wait.

4. **Ordered**

The ordered region executes in the sequential order. Parallel threads execute concurrently until they encounter the order block, which is then executed sequentially in the same order as it would get executed in a serial loop.

```
#pragma omp parallel for ordered schedule(dynamic, 4)
for (i=0; i <N; i++) {
   tmp = foo(i);
#pragma ordered
   res += consume(tmp);
}
```

Low Level Synchronization

1. **Flush**

The omp flush directive identifies a point at which the compiler ensures that all threads in a parallel region have the same view of specified objects, also called flush set, in memory. When no argument is specified to the flush clause, all thread visible variables are part of flush set. The action of flush is to guarantee that all read, write operations that overlap the flush set and occur prior to the flush complete before the flush executes. All read, write operations that overlap the flush set and occur after the flush don't execute until after the flush. Flushes with overlapping flush sets cannot be reordered.

Flush forces data to updated in memory so other threads see the most recent value. Implicit flushes are active at barriers, at entry to and exit from parallel, parallel worksharing (unless nowait), critical and ordered regions.

Why flush? Compiler routinely reorder instructions implementing a program, but they cannot generally move instructions past a barrier and past a flush on all variables. It can move them past a flush with a list of variables as long as those variables are not accessed.

Flush operation does not actually synchronize different threads. It just ensures that a thread's values are made consistent with main memory. For example, if a produce thread writes a shared variable a, and a consumer thread reads that shared variable. In order to synchronize the shared variable, the producer flush the variable as soon as it is written, and the consumer flush the variable before read.

```
#pragma omp sections
{
#pragma omp section // more about section later
{
        fill_rand(N, A); // producer
        #pragma omp flush
        flag = 1;
        #pragma omp flush (flag)
}
#pragma omp section
{
        #pragma omp flush (flag)
        while (flag != 1){
        #pragma omp flush (flag)
        }

        #pragma omp flush
        sum = Sum_array(N, A); // consumer
        }
}
```

2. **Locks**

Locks can be used to protect resources. A lock implies a memory fence (a "flush") of all thread variables.

```
omp_lock_t lck;
omp_init_lock(&lck); // init lock
#pragma omp parallel private(tmp, id)
{
        id = omp_get_thread_num();
        tmp = do_lots_of_work(id);
        omp_set_lock(&lck); // wait here for your turn
        printf("%d %d", id, tmp);
        omp_unset_lock(&lck); // release the lock
}
omp_destroy_lock(&lck); // free up storage
```

A nested lock is available if it is unset or if it is set but owned by the thread executing the nested lock function.

6. Advantages of OpenMP:

- Portable multithreading code (in C/C++ and other languages, one typically has to call platform-specific primitives in order to get multithreading).
- Simple: need not deal with message passing as MPI does.
- Data layout and decomposition is handled automatically by directives.
- Scalability comparable to MPI on shared-memory systems.
- Incremental parallelism: can work on one part of the program at one time, no dramatic change to code is needed.
- Unified code for both serial and parallel applications: OpenMP constructs are treated as comments when sequential compilers are used.
- Original (serial) code statements need not, in general, be modified when parallelized with OpenMP. This reduces the chance of inadvertently introducing bugs.
- Both coarse-grained and fine-grained parallelism are possible.
- In irregular multi-physics applications which do not adhere solely to the SPMD mode of computation, as encountered in tightly coupled fluid-particulate systems, the flexibility of OpenMP can have a big performance advantage over MPI.
- Can be used on various accelerators such as GPGPU.

FAQs:

1. What are the different environment variable?

OpenMP provides the following environment variables for controlling the execution of parallel code.

- All environment variable names are uppercase. The values assigned to them are not case sensitive.

OMP_SCHEDULE

Applies only to DO, PARALLEL DO (Fortran) and for, parallel for (C/C++) directives which have their schedule clause set to RUNTIME. The value of this variable determines how iterations of the loop are scheduled on processors. For example:

```
setenv OMP_SCHEDULE "guided, 4"
setenv OMP_SCHEDULE "dynamic"
```

OMP_NUM_THREADS

Sets the maximum number of threads to use during execution. For example:

```
setenv OMP_NUM_THREADS 8
```

OMP_DYNAMIC

Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. Valid values are TRUE or FALSE. For example:

```
setenv OMP_DYNAMIC TRUE
```

OMP_PROC_BIND

Enables or disables threads binding to processors. Valid values are TRUE or FALSE. For example:

```
setenv OMP_PROC_BIND TRUE
```

OMP_NESTED

Enables or disables nested parallelism. Valid values are TRUE or FALSE. For example:

```
setenv OMP_NESTED TRUE
```

OMP_STACKSIZE

Controls the size of the stack for created (non-Master) threads. Examples:

```
setenv OMP_STACKSIZE 2000500B
setenv OMP_STACKSIZE "3000 k "
setenv OMP_STACKSIZE 10M
setenv OMP_STACKSIZE " 10 M "
setenv OMP_STACKSIZE "20 m "
setenv OMP_STACKSIZE " 1G"
setenv OMP_STACKSIZE 20000
```

OMP_WAIT_POLICY

Provides a hint to an OpenMP implementation about the desired behavior of waiting threads. A compliant OpenMP implementation may or may not abide by the setting of the environment variable. Valid values are ACTIVE and PASSIVE. ACTIVE specifies that waiting threads should mostly be active, i.e., consume processor cycles, while waiting. PASSIVE specifies that waiting threads should mostly be passive, i.e., not consume processor cycles, while waiting. The details of the ACTIVE and PASSIVE behaviors are implementation defined. Examples:

```
setenv OMP_WAIT_POLICY ACTIVE
setenv OMP_WAIT_POLICY active
setenv OMP_WAIT_POLICY PASSIVE
setenv OMP_WAIT_POLICY passive
```

OMP_MAX_ACTIVE_LEVELS

Controls the maximum number of nested active parallel regions. The value of this environment variable must be a non-negative integer. The behavior of the program is implementation defined if the requested value of OMP_MAX_ACTIVE_LEVELS is greater than the maximum number of nested active parallel levels an implementation can support, or if the value is not a non-negative integer. Example:

```
setenv OMP_MAX_ACTIVE_LEVELS 2
```

OMP_THREAD_LIMIT

Sets the number of OpenMP threads to use for the whole OpenMP program. The value of this environment variable must be a positive integer. The behavior of the program is implementation defined if the requested value of OMP_THREAD_LIMIT is greater than the number of threads an implementation can support, or if the value is not a positive integer. Example:

```
setenv OMP_THREAD_LIMIT 8
```

2. Explain synchronization wrt openmp.

Synchronization is used to impose order constraints and to protect access to shared data. There are two levels of synchronization applicable to OpenMP. *High level synchronization* involves high-level compiler directive instructions, while *Low level synchronization* is similar to thread level locking mechanism.

High Level Synchronization

1. Critical
This is a high level construct for mutual exclusion making sure that only one thread is able to enter a critical region at a time.

2. Atomic

Atomic provides similar mutual exclusion as the above critical pragma but it only applies to the update of a memory location.

3. Barrier

OMP barrier is a very important concept in synchronization. It provides a synchronization point at which threads in a parallel region will wait until all other threads in that section reach the same point.

4. Ordered

The ordered region executes in the sequential order. Parallel threads execute concurrently until they encounter the order block, which is then executed sequentially in the same order as it would get executed in a serial loop.

Low Level Synchronization

1. Flush

The omp flush directive identifies a point at which the compiler ensures that all threads in a parallel region have the same view of specified objects, also called flush set, in memory.

2. Locks

Locks can be used to protect resources. A lock implies a memory fence (a “flush”) of all thread variables.