Week No 1: Basics of OPENMP

THEORY

OpenMP

OpenMP is a portable and standard Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism

OpenMP attempts to standardize existing practices from several different vendor-specific shared memory environments. OpenMP provides a portable shared memory API across different platforms including DEC, HP, IBM, Intel, Silicon Graphics/Cray, and Sun. The languages supported by OpenMP are FORTRAN, C and C++. Its main emphasis is on performance and scalability.

Goals of OpenMP

- **Standardization:** Provide a standard among a variety of shared memory architectures/platforms
- **Lean and Mean:** Establish a simple and limited set of directives for programming shared memory machines. Significant parallelism can be implemented by using just 3 or 4 directives.
- **Ease of Use:** Provide capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach and also provide the capability to implement both coarse-grain and fine-grain parallelism **Portability:** Supports Fortran (77, 90, and 95), C, and C++

OpenMP Programming Model

Shared Memory, Thread Based Parallelism:

- OpenMP is based upon the existence of multiple threads in the shared memory programming paradigm. A shared memory process consists of multiple threads.

Explicit Parallelism:

- OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.

Fork - Join Model:

OpenMP uses the fork-join model of parallel execution:

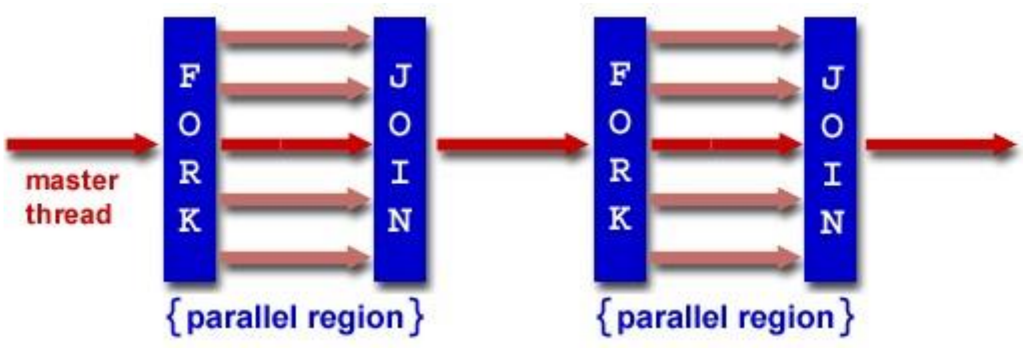

Figure 1 Fork and Join Model

- 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

Compiler Directive Based:

- Most OpenMP parallelism is specified through the use of compiler directives which are imbedded in C/C++ or Fortran source code.

Nested Parallelism Support:

- The API provides for the placement of parallel constructs inside of other parallel constructs.
- Implementations may or may not support this feature.

Dynamic Threads:

- The API provides for dynamically altering the number of threads which may used to execute different parallel regions.
- Implementations may or may not support this feature.

I/O:

- OpenMP specifies nothing about parallel I/O. This is particularly important if multiple threads attempt to write/read from the same file.
- If every thread conducts I/O to a different file, the issues are not as significant.
- It is entirely up to the programmer to insure that I/O is conducted correctly within the context of a multi-threaded program.

Components of OpenMP API

- Comprised of three primary API components:
    - Compiler Directives
    - Runtime Library
    - Routines
    - Environment Variables

C / C++ - General Code Structure

```
#include <omp.h>
main ()
{

      int var1, var2, var3;

      Serial code


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

            {
                    Parallel section executed by all threads
                         .
                         .
                         .
                    All threads join master thread and disband
              }

       Resume serial code
  }
```

Important terms for an OpenMP environment

Construct: A construct is a statement. It consists of a directive and the subsequent structured block. Note that some directives are not part of a construct.

directive: A C or C++ **#pragma** followed by the **omp** identifier, other text, and a new line. The directive specifies program behavior.

Region: A dynamic extent.

dynamic extent: All statements in the *lexical extent*, plus any statement inside a function that is executed as a result of the execution of statements within the lexical extent. A dynamic extent is also referred to as a *region*.

lexical extent: Statements lexically contained within a *structured block*.

structured block: A structured block is a statement (single or compound) that has a single entry and a single exit. No statement is a structured block if there is a jump into or out of that statement. A compound statement is a structured block if its execution always begins at the opening **{** and always ends at the closing **}**. An expression statement, selection statement, iteration statement is a structured block if the corresponding compound statement obtained by enclosing it in **{** and **}**would be a structured block. A jump statement, labeled statement, or declaration statement is not a structured block.

Thread: An execution entity having a serial flow of control, a set of private variables, and access to shared variables.

master thread: The thread that creates a team when a *parallel region* is entered.

serial region: Statements executed only by the *master thread* outside of the dynamic extent of any *parallel region*.

parallel region: Statements that bind to an OpenMP parallel construct and may be executed by multiple threads.

Variable: An identifier, optionally qualified by namespace names, that names an object.

Private: A private variable names a block of storage that is unique to the thread making the reference. Note that there are several ways to specify that a variable is private: a definition within a parallel region, a threadprivate directive, a private, firstprivate, lastprivate, or reduction clause, or use of the variable as a forloop control variable in a for loop immediately following a for or parallel for directive.

Shared: A shared variable names a single block of storage. All threads in a team that access this variable will access this single block of storage.

Team: One or more threads cooperating in the execution of a construct.

Serialize: To execute a parallel construct with a team of threads consisting of only a single thread (which is the master thread for that parallel construct), with serial order of execution for the statements within the structured block (the same order as if the block were not part of a parallel construct), and with no effect on the value returned by **omp_in_parallel()** (apart from the effects of any nested parallel constructs).

Barrier: A synchronization point that must be reached by all threads in a team. Each thread waits until all threads in the team arrive at this point. There are explicit barriers identified by directives and implicit barriers created by the implementation.

EXERCISE:

1. Discuss the different types of Parallel Programming Models.

____

____

____

2. List down the possible characteristics of an APIs (Application programming interface).