Parallel Machine Learning and Artificial Intelligence

Dr. Handan Liu

h.liu@northeastern.edu

Northeastern University

Content

- High Performance Parallel Computing
  - Overview (done)
  - Concepts and Terminology (done)
  - Parallel Computer Memory Architectures (done)
  - Parallel Programming Model (done)
    - ✔ Parallel Implementations – OpenMP Programming
    - ✔ Parallel Implementations – MPI Programming
  - Parallel Examples and Exercises

Parallel Implementations - OpenMP Programming

Pthreads/OpenMP

- Pthreads is a low-level API that allows you to implement pretty much any parallel computation exactly the way you want it. But it required high-level programming skills.
- However, in many cases, the user only wants to parallelize certain common situations:
  - For loop: partition the loop into chunks and have each thread process one chunk.
  - Hand-off a block of code (computation) to a separate thread.
- This is where OpenMP is useful. It simplifies the programming significantly.
  - In some cases, adding one line in a C code is sufficient to make it run in parallel.
- As a result, OpenMP is the standard approach in scientific computing for multicore processors.

What is OpenMP?

- An Application Program Interface (API) that may be used to explicitly direct **multi-threaded, shared memory** parallelism.
- OpenMP is based on directives.
- Powerful because of simple syntax.
- OpenMP provides a portable, scalable model for developers of shared memory parallel applications.
- Dangerous because you may not understand exactly what the compiler is doing.

Hence, it is more portable and general than **Pthreads**.

OpenMP website: https://www.openmp.org/
Wikipedia: https://en.wikipedia.org/wiki/OpenMP

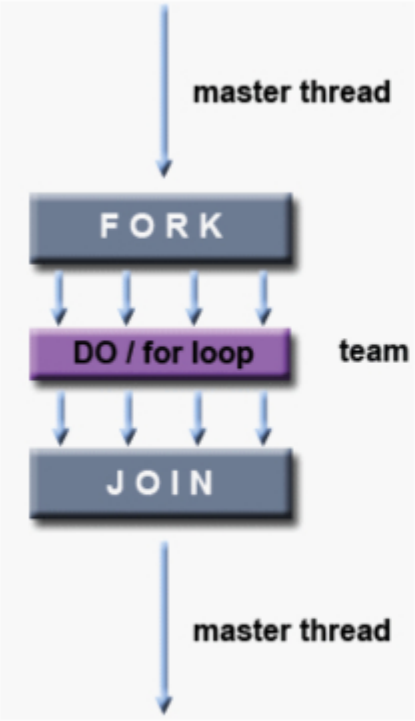
OpenMP Programming Model

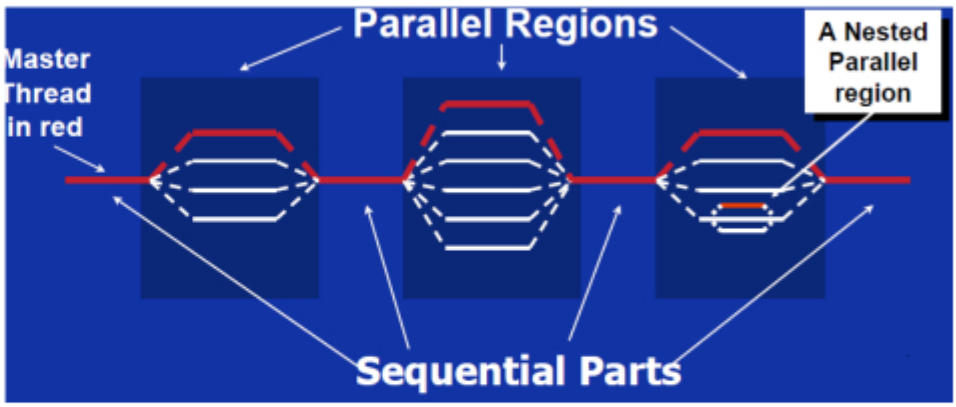
- Shared Memory Model:
  - OpenMP is designed for multi-processor/core, shared memory machines. The underlying architecture can be shared memory **UMA** or **NUMA**.
  - It is largely limited to **single node** parallelism.
- Motivation for Using OpenMP in HPC:
  - By itself, OpenMP parallelism is limited to a single node.
  - For HPC applications, OpenMP is combined with MPI for the distributed memory parallelism. This is often referred to as Hybrid Parallel Programming.
  - This allows parallelism to be implemented to the full scale of a cluster.

OpenMP Programming Model

- Thread Based Parallelism
  - OpenMP programs accomplish parallelism exclusively through the use of threads.
  - A thread of execution is the smallest unit of processing that can be scheduled by an operating system.
  - Threads exist within the resources of a single process. Without the process, they cease to exist.
  - Typically, the number of threads match the number of machine processors/cores. However, the actual use of threads is up to the application.
- Fork-Join Model:

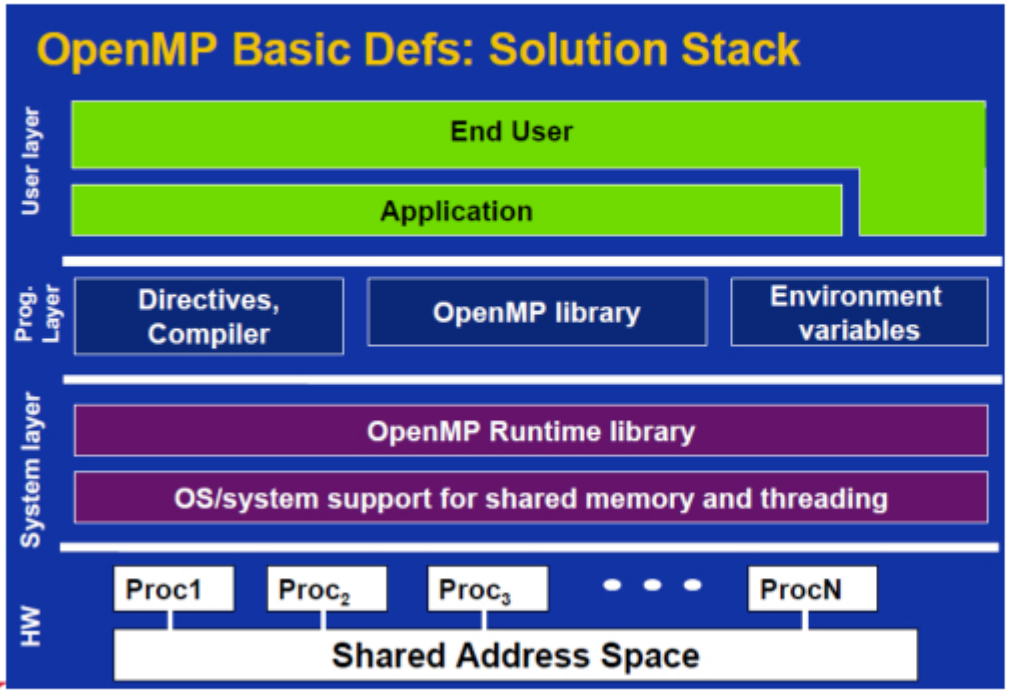
Fork-Join Model





OpenMP uses the fork-join model of parallel execution

OpenMP API Overview



Three Components:

- Compiler Directives
- Runtime Library Routines
- Environment Variables

OpenMP API Overview

Compiler Directives

Run-time Library Routines

Environment Variables

- Compiler directives have the following syntax:

sentinel directive-name [clause, ...]

Fortran	**!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(BETA,PI)**
C/C++	**#pragma omp parallel default(shared) private(beta,pi)**

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

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

OpenMP API Overview

Compiler Directives

Run-time Library Routines

Environment Variables

- For C/C++, all of the run-time library routines are actual subroutines.

Fortran	**INTEGER FUNCTION OMP_GET_NUM_THREADS()**
C/C++	**#include <omp.h>** **int omp_get_num_threads(void)**

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

Runtime Library Routines (32)

Routine	Purpose
OMP_SET_NUM_THREADS	Sets the number of threads that will be used in the next parallel region
OMP_GET_NUM_THREADS	Returns the number of threads that are currently in the team executing the parallel region from which it is called
OMP_GET_MAX_THREADS	Returns the maximum value that can be returned by a call to the OMP_GET_NUM_THREADS function
OMP_GET_THREAD_NUM	Returns the thread rank in a parallel region (0 ~ (N-1)).
OMP_GET_THREAD_LIMIT	Returns the maximum number of OpenMP threads available to a program
OMP_GET_NUM_PROCS	Returns the number of processors that are available to the program
OMP_IN_PARALLEL	Used to determine if the section of code which is executing is parallel or not
OMP_GET_NESTED	Used to determine if nested parallelism is enabled or not
OMP_GET_DYNAMIC	Used to determine if dynamic thread adjustment is enabled or not
OMP_GET_WTIME	Provides a portable wall clock timing routine

...plus a few less commonly used routines.

OpenMP API Overview

Compiler Directives

Run-time Library Routines

Environment Variables

For example:

csh/tcsh	**setenv OMP_NUM_THREADS 8**
sh/bash	**export OMP_NUM_THREADS=8 (Discovery)**

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

- Setting the number of threads
- Specifying how loop iterations 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

Example OpenMP Code Structure:

Fortran - General Code Structure

```
PROGRAM HELLO

INTEGER VAR1, VAR2, VAR3

Serial code
      .
      .
      .

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

!$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

!$OMP END PARALLEL

Resume serial code
      .
      .
      .

END
```

C / C++ - General Code Structure

```
#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
      .
      .
      .

}
```


OpenMP Directives

- C / C++ Directives Format

#pragma omp	**directive-name**	**[clause, ...]**	**newline**
Required for all OpenMP C/C++ directives.	A valid OpenMP directive. Must appear after the pragma and before any clauses.	Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.	Required. Precedes the structured block which is enclosed by this directive.

- Example:

```
#pragma omp parallel default(shared) private(beta,pi)
```

PARALLEL Region Construct

- Purpose:
  - A parallel region is a block of code that will be executed by multiple threads. This is the fundamental OpenMP parallel construct.
- Format:

```
#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 ..............}
```

PARALLEL Region Construct

- How Many Threads?
  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.

PARALLEL Region Construct

- Restrictions:
  - A parallel region must be a structured block that does not span multiple routines or code files
  - It is illegal to branch (goto) into or out of a parallel region
  - Only a single **IF** clause is permitted
  - Only a single **NUM_THREADS** clause is permitted
  - A program must not depend upon the ordering of the clauses

Example: Parallel Region

- Simple "Hello World" program
  - Every thread executes all code enclosed in the parallel region.
  - OpenMP library routines are used to obtain thread identifiers and total number of threads.

- Stay safe!
- See you next class!

Next Lecture will Continue:

Continue to OpenMP Programming