Introduction to OpenMP
CS 540- High Performance Computing

Spring 2017

OpenMP

- OpenMP is an Application Program Interface (API)
- Defined by a group of major 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.
- OpenMP 3.1

OpenMP cont.

- An Application Program Interface (API) that may be used to develop **multi-threaded, shared memory** parallelism.
- Comprised of three primary API components:
- Compiler Directives
- Runtime Library Routines
- Environment Variables

OpenMP cont.

OpenMP Is Not:

- Designed for distributed memory parallel systems.
- Guaranteed to make the most efficient use of shared memory. Does not guarantee speed up
- Required to check for data dependencies, data conflicts, race conditions, deadlocks, etc.
- The programmer is responsible for synchronizing input and output.

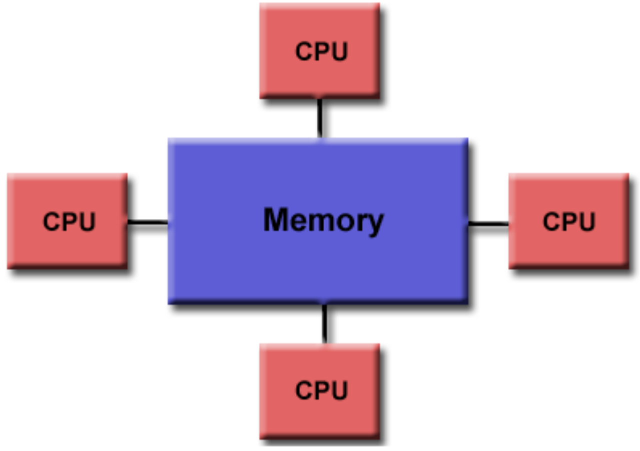
Goals of OpenMP

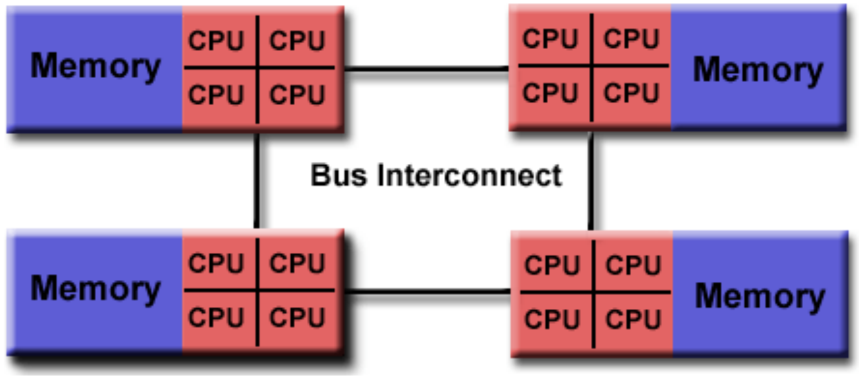
- Provide a standard among a variety of shared memory architectures/platforms
- Establish a simple and limited set of directives for programming shared memory machines.
- Parallelism can be accomplished by using roughly 3-4 directives.
- Provide capability to incrementally parallelize a serial program

OpenMP Model

- OpenMP is designed for multi-processor/core, shared memory machines.
- Supported architectures can be shared memory UMA or NUMA.

Open MP Model cont.





Source: Lawrence Livermore National Laboratory

OpenMP Parallelism

Thread Based Parallelism

- OpenMP programs accomplish parallelism exclusively through the use of threads.
- A thread of execution is the smallest unit of processing Threads exist within the resources of a single process.
- Typically, the number of threads match the number of machine processors/cores.
- The actual use of threads is the decision of the developer.

OpenMP Parallelism

Explicit Parallelism

- OpenMP is an explicit programming model,
- Gives the programmer full control over parallelization.
- Parallelization can be as simple as taking a serial program and inserting compiler directives.
- Complex as inserting subroutines to set multiple levels of parallelism, locks and even nested locks

OpenMP Parallelism

Fork Join

- All OpenMP programs begin as a single process
- The master thread executes sequentially until the first **parallel region** is encountered.
- When parallel region is encountered, the master thread then creates a team of parallel ***threads***.
- The statements in the program that are enclosed by the parallel region (directive) construct are then executed in parallel among the various team threads.

OpenMP Parallelism

Fork Join

- When the team threads complete the statements in the parallel region, they synchronize and terminate, leaving only the master thread.
- The number of parallel regions and the threads are arbitrary

OpenMP API

Three components:

- Compiler Directives
- Runtime Library Routines
- Environment Variables
- The developer decides how many of these components to employ.

Compiler Directives

- Compiler directives appear as comments in your source code.
- ignored by compilers unless you tell them otherwise - usually by specifying the appropriate compiler flag
- Invoking a parallel region
- Partitioning blocks of code among threads
- Distributing loop iterations between threads
- Serializing sections of code
- Synchronization of work among threads

Compiler Directives

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

```
#include <omp.h>
#include <stdio.h>
int main(){
    printf("Hello from main.\n");

    // Parallel region with default number of threads
    #pragma omp parallel
{
    int myID = 0;
    printf("hello(%d)", myID);
    printf("world(%d)\n", myID);
    }
}
```

Compiler Directives

- #pragma omp parallel
- A parallel region is a block of code that will be executed by multiple threads.
- This is the fundamental OpenMP parallel construct.
- When a thread reaches a PARALLEL directive, it creates a team of threads and becomes the master of the team.
- The master thread has thread number 0 within that team.

Compiler Directives

- The code is duplicated from the beginning of the parallel region 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

Compiler Directives

- How many threads can be invoked?
- The number of threads in a parallel region is determined by the following factors:

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.

Compiler Directives

- *#pragma omp parallel num_threads(24)* – sets the number of threads in a parallel region
- *#pragma shared(a,b,c)* – declares variables in its list to be shared among all threads in the team.
- *#pragma private(a,b,c)* – declares variables in its list to be private to each thread.
- *#pragma sections*– the code is partitioned into a section. A thread is then assigned to that section and will execute the code once.

Run-time Libraries

- The OpenMP API includes numerous run-time library functions.
- These functions are used for a variety of purposes
- Setting and querying the number of threads
- Retrieving a thread's unique identifier (thread ID), a thread's ancestor's id, the thread team size etc.
- Setting and querying the dynamic threads feature
- Querying if in a parallel region
- initializing and terminating locks and nested locks

Run-time Libraries

- *omp_get_thread_num*() – returns the id of a thread
- *omp_set_num_threads(24)* – modifies the default number of threads to the value passed in to the set function.
- *omp_get_num_procs()* – returns the number of cores on available.
- *omp_get_dynamic()* – returns true/false to determine if dynamic threads are supported

References

1. Blaise Barney, Lawrence Livermore National Laboratory, https://computing.llnl.gov/tutorials/parallel_comp/#Overview
2. https://en.wikipedia.org/wiki/Flynn's_taxonomy
3. https://www.citutor.org/index.php, Parallel Computing Explained