OpenMP Overview: What is OpenMP?

- Model for parallel programming
- Shared-memory parallelism
- Portable across shared-memory architectures
- Scalable
- Incremental parallelization
- Compiler based
- Extensions to existing programming languages
  - mainly by directives
  - a few library routines
- Fortran and C/C++ binding
- Supports data parallelism

Acknowledgment: These slides are based on Allan’s Snavely’s overview lecture, SDSC CS260
Motivation: why should I use OpenMP?
Where should I use OpenMP?
Simple OpenMP Program

- Most OpenMP constructs are compiler directives or pragmas
- The focus of OpenMP is to parallelize loops
- OpenMP offers an incremental approach to parallelism

Serial Program:

```c
void main()
{
    double Res[1000];
    for(int i=0;i<1000;i++) {
        do_huge_comp(Res[i]);
    }
}
```

Parallel Program:

```c
void main()
{
    double Res[1000];
    #pragma omp parallel for
    for(int i=0;i<1000;i++) {
        do_huge_comp(Res[i]);
    }
}
```
Outline

- Standardization Body: Who owns OpenMP?
  - OpenMP Application Program Interface (API)
- Execution Model
  - Parallel regions: team of threads
  - Syntax
  - Data environment (part 1)
  - Environment variables
  - Runtime library routines
- Work-sharing directives
  - Which thread executes which statement or operation?
  - Synchronization constructs, e.g., critical sections
- Data environment and combined constructs
  - Private and shared variables
  - Combined parallel work-sharing directives
- Summary of OpenMP API
OpenMP Programming Model

OpenMP is a shared memory model.

- Workload is distributed between threads
  - Variables can be
    - shared among all threads
    - duplicated for each thread
  - Threads communicate by sharing variables
    - Unintended sharing of data can lead to race conditions:
  - Race condition: when the program’s outcome changes as the threads are scheduled differently.
    - Use synchronization to protect data conflicts.

- To control race conditions
  - Use synchronization to protect shared data access.

- Careless use of synchronization can lead to deadlocks
OpenMP Execution Model

Fork-join model of parallel execution

- Begin execution as a single process (master thread)

- Start of a parallel construct:
  - Master thread creates team of threads

- Completion of a parallel construct:
  - Threads in the team synchronize: implicit barrier

- Only master thread continues execution
OpenMP Execution Model
OpenMP parallel region construct

Block of code to be executed by multiple threads in parallel
Each thread executes the same code redundantly!
- C/C++:
  #pragma omp parallel [ clause [ clause ] ... ] new-line structured-block
- clause can be one of the following:
  private( list)
  shared( list)
OpenMP parallel region construct

C / C++:

```c
#pragma omp parallel
  structured block
/* omp end parallel */
```
OpenMP directive format C

#pragma directives

- Format:
  #pragma omp directive_name [ clause [ clause ] ... ] new-line

- Conditional compilation
  #ifdef _OPENMP
  block,
  e.g., printf("%d avail.processors\n",omp_get_num_procs());
  #endif

- Case sensitive
- Header file for library routines:
  #ifdef _OPENMP
  #include <omp.h>
  #endif

OpenMP Introduction
OpenMP data scope clauses

private ( list )
Declares the variables in list to be private to each thread in a team

shared ( list )
Makes variables that appear in list shared among all the threads in a team

- If not specified: default shared, but
  - stack (local) variables in called sub-programs are PRIVATE

- Loop control variable of parallel OMP
  - DO (Fortran)
  - for (C)
  is PRIVATE
OpenMP environment variables

OMP_NUM_THREADS
   – sets the number of threads to use during execution
   – when dynamic adjustment of the number of threads is enabled, the value of this environment variable is the maximum number of threads to use
   
   setenv OMP_NUM_THREADS 16 [csh, tcsh]
   export OMP_NUM_THREADS=16 [sh, ksh, bash]

OMP_SCHEDULE
   – applies only to do/for and parallel do/for directives that have the schedule type RUNTIME
   – sets schedule type and chunk size for all such loops
   
   setenv OMP_SCHEDULE GUIDED,4 [csh, tcsh]
   export OMP_SCHEDULE= GUIDED,4 [sh, ksh, bash]
OpenMP runtime library

Query functions

- Runtime functions
  - Run mode
  - Nested parallelism

- Lock functions

- C/C++: add #include <omp.h>
OpenMP runtime library

- **omp_get_num_threads** Function

  Returns the number of threads currently in the team executing the parallel region from which it is called

  ```c
  int omp_get_num_threads(void);
  ```

- **omp_get_thread_num** Function

  Returns the thread number, within the team, that lies between 0 and `omp_get_num_threads()-1`, inclusive. The master thread of the team is thread 0

  ```c
  int omp_get_thread_num(void);
  ```
Work sharing directives

- Which thread executes which statement or operation?
- And when?
  - Work-sharing constructs
  - Master and synchronization constructs
- i.e., organization of the parallel work!!!
OpenMP work sharing constructs

Divide the execution of the enclosed code region among the members of the team

- Must be enclosed dynamically within a parallel region
- They do not launch new threads
- No implied barrier on entry

sections directive
do directive (Fortran)
for directive (C/C++)
OpenMP sections directive

Several blocks are executed in parallel

- C/C++:

```c
#pragma omp sections [ clause [ clause ] ... ] new-line
{
[#pragma omp section new-line]
structured-block1
[#pragma omp section new-line
structured-block2 ]
...
}
```
OpenMP sections directive

```c
#pragma omp parallel
{
#pragma omp sections
{{
    a=...;
    b=...;
}
#pragma omp section
{
    c=...;
    d=...;
}
#pragma omp section
{
    e=...;
    f=...;
}
#pragma omp section
{
    g=...;
    h=...;
}
} /*omp end sections*/
} /*omp end parallel*/
```
**OpenMP for directive**

Immediately following loop executed in parallel

- C/C++:

```c
#pragma omp for [ clause [ clause ] ... ] new-line
  for-loop
```
```c
#pragma omp parallel private(f)
{
    f=7;
#pragma omp for
    for (i=0; i<20; i++)
    a[i] = b[i] + f * (i+1);
} /* omp end parallel */
```
OpenMP for directive

- Clause can be one of the following:
  - `private(list)` [see later: Data Model]
  - `reduction(operator: list)` [see later: Data Model]
  - `schedule(type [, chunk])`
  - `nowait` (C/C++: on `#pragma omp for`)

  - Implicit barrier at the end of `do/for` unless `nowait` is specified
  - If `nowait` is specified, threads do not synchronize at the end of the parallel loop

- `schedule` clause specifies how iterations of the loop are divided among the threads of the team.
  - Default is implementation-dependent
OpenMP schedule clause

Within `schedule( type [, chunk ] )` type can be one of the following:

- **static**: Iterations are divided into pieces of a size specified by chunk.
  - The pieces are statically assigned to threads in the team in a roundrobin fashion in the order of the thread number.
  - Default chunk size: one contiguous piece for each thread.

- **dynamic**: Iterations are broken into pieces of a size specified by chunk.
  - As each thread finishes a piece of the iteration space, it dynamically obtains the next set of iterations. Default chunk size: 1.

- **guided**: The chunk size is reduced in an exponentially decreasing manner with each dispatched piece of the iteration space.
  - chunk specifies the smallest piece (except possibly the last).
  - Default chunk size: 1. Initial chunk size is implementation dependent.

- **runtime**: The decision regarding scheduling is deferred until run time.
  - The schedule type and chunk size can be chosen at run time by setting the `OMP_SCHEDULE` environment variable.
  - Default schedule: implementation dependent.
Loop scheduling

static
dynamic(3)
guided(1)
OpenMP synchronization

- Implicit Barrier
  - beginning and end of parallel constructs
  - end of all other control constructs
  - implicit synchronization can be removed with `nowait` clause

- Explicit
  critical
OpenMP critical directive

Enclosed code
– executed by all threads, but
– **restricted to only one thread at a time**
  • C/C++:
    #pragma omp critical [( name ) ] new-line
    structured-block

• A thread waits at the beginning of a critical region until no other thread in the team is executing a critical region with the same name.

All unnamed `critical` directives map to the same unspecified name.
OpenMP critical

C / C++:

```c
cnt = 0;
f = 7;
#pragma omp parallel
{
    #pragma omp critical
        cnt ++;
    } /* endif */
    a[i] = b[i] + f * (i+1);
} /* end for */
} /*omp end parallel */
```
OpenMP Atomic

- OpenMP atomic operations allow multiple threads to safely update a shared numeric variable.
- An atomic operation applies only to the single assignment statement that immediately follows it.

```c++
C / C++:
cnt = 0;
f=7;
#pragma omp parallel
{
    #pragma omp for
    for (i=0; i<20; i++) {
        if (b[i] == 0) {
            #pragma omp atomic
            cnt = cnt + 1;
        } /* endif */
        a[i] = b[i] + f * (i+1);
    } /* end for */
} /*omp end parallel */
```
OpenMP control structures - summary

- Parallel region construct
  parallel
- Work-sharing constructs
  sections
    do (Fortran)
    for (C/C++)
- Combined parallel work-sharing constructs [see later]
  parallel do (Fortran)
  parallel for (C/C++)
- Synchronization constructs
  critical