Introduction to OpenMP
Definition of OpenMP

• Application Program Interface (API) for Shared Memory Parallel Programming

• Directive based approach with library support

• Targets existing applications and widely used languages:
  – Fortran API released October ‘97
  – C, C++ API released October ‘98

• Supports multiple platform
OpenMP Specification

• Application Program Interface (API) for Shared Memory Parallel Programming
  – non-profit organization: www.openmp.org

  • full reference manual http://www.openmp.org/specs
A Programmer’s View of OpenMP

• OpenMP is a portable, threaded, shared-memory programming specification with “light” syntax
  – Exact behavior depends on OpenMP implementation!
  – Requires compiler support (C/C++ or Fortran)

• OpenMP will:
  – Allow a programmer to separate a program into serial regions and parallel regions, rather than concurrently-executing threads.
  – Provide synchronization constructs

• OpenMP will not:
  – Parallelize automatically
  – Guarantee speedup
  – Provide freedom from data races
Multithreading in OpenMP

• OpenMP is an implementation of multithreading, a method of parallelization where master thread forks a specified number of slave threads and a task is divided among them.

• The threads then run concurrently, with the runtime environment allocating threads to different processors.

• The runtime environment allocates threads to processors depending on usage, machine load and other factors.

• The number of threads can be assigned by the runtime environment based on environment variables or in code using functions.

• The OpenMP functions are included in a header file labelled "omp.h" in C/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

- **All OpenMP programs begin as a single process:**
  - The master thread executes sequentially until the first parallel region construct is encountered
OpenMP programming model (Cont.)

- **FORK**: the master thread then creates a team of parallel threads
  - The statements in the program that are enclosed by 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
OpenMP programming model (Cont.)

- **Compiler Directive Based:**
  - OpenMP parallelism is specified through the use of compiler directives.

- **Nested Parallelism Support:**
  - API facilitates placement of parallel constructs inside 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.
OpenMP Interface Model

- Control structures
- Work sharing
- Synchronization
- Data scope attributes:
  - Private
  - Firstprivate
  - Lastprivate
  - Shared
  - Reduction

- Control and query routines:
  - Number of threads
  - Throughput mode
  - Nested parallelism
  - Lock API

- Runtime environment:
  - Schedule type
  - Max #threads
  - Nested parallelism
  - Throughput mode

Directives And pragmas
Runtime library routines
Environment Variables

Slides prepared by: Farzana Rahman
OpenMP Directives Format

• #pragma omp
  – Required for all OpenMP C/C++ directives

• directive-name

• [clause, …]
  – Optional clauses can be in any order, and repeated as necessary unless otherwise restricted.

• Newline

• Example:

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

• Parallel
  – Specifies a region that will executed in parallel.

• Do/for
  – Shares iterations of a loop across a team of threads.

• Sections
  – Breaks work into separate, discrete sections. Each section is executed by a thread.

• Single
  – Serialized a section of code.
OpenMP Synchronization

• Master
  – Specifies a region that is to be executed only by the master thread of the team.

• Critical
  – Specifies a region of code that must be executed only one thread at a time

• Barrier
  – Synchronizes all threads on the team
OpenMP Synchronization

• Atomic
  – Specifies that a memory location can only be written by one thread at a time.

• Flush
  – Identifies a point in which all thread visible variable are written back to memory.

• Ordered
  – Specifies that iterations of the enclosed loop will be executed in the same order as if they were executed on a serial processor.

Slides prepared by: Farzana Rahman
OpenMP basic syntax

• Most of the constructs in OpenMP are compiler directives.
  - `#pragma omp construct [clause [clause]...]`

• Function prototypes and types in the file:
  - `#include <omp.h>`

• Most OpenMP constructs apply to a structured block.

• Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom.

• It’s OK to have an exit() within the structured block.
Thread Creation: Parallel Regions

• You create threads in OpenMP with the parallel construct.

• For example, To create a 4 thread Parallel region:

```c
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID,A);
}
```
Synchronization

• High level synchronization:
  – Critical
  – Atomic
  – Barrier
  – ordered

• Low level synchronization
  – Flush
  – locks (both simple and nested)
Synchronization: critical

- Mutual exclusion: Only one thread at a time can enter a critical region.

```c
float res;
#pragma omp parallel
{
  float B; int i, id, nthrds;
  id = omp_get_thread_num();
  nthrds = omp_get_num_threads();
  for(i=id;i<niters;i+nthrds)
  {
    B = big_job(i);
    #pragma omp critical
    consume (B, res);
  }
}
```
Synchronization: Atomic

- Atomic provides mutual exclusion but only applies to the update of a memory location (the update of X in the following example)

```c
#pragma omp parallel
{
    double tmp, B;
    B = DOIT();
    tmp = big_ugly(B);
    #pragma omp atomic
    X += tmp;
}
```
The loop worksharing Constructs

• The loop worksharing construct splits up loop iterations among the threads in a team

```c
#pragma omp parallel
{
    #pragma omp for
    for (I=0;I<N;I++)
    {
        NEAT_STUFF(I);
    }
}
```
Combined parallel/worksharing construct

- OpenMP shortcut: Put the “parallel” and the worksharing directive on the same line

```c
double res[MAX]; int i;
#pragma omp parallel
{
    #pragma omp for
    for (i=0; i< MAX; i++)
    {
        res[i] = huge();
    }
}
```
Reduction

• OpenMP reduction clause:
  – reduction (op : list)

• Inside a parallel or a work-sharing construct:
  – A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”).
  – Compiler finds standard reduction expressions containing “op” and uses them to update the local copy.
  – Local copies are reduced into a single value and combined with the original global value.

• The variables in “list” must be shared in same parallel region.
OpenMP: Reduction operands/initial-values

- Many different associative operands can be used with reduction:
- Initial values are the ones that make sense mathematically.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>&amp;</td>
<td>~0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>0</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Synchronization: Barrier

- **Barrier**: Each thread waits until all threads arrive.
- **implicit barrier at the end of a for worksharing construct**

```
#pragma omp parallel shared (A, B, C) private(id)
{
    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);}
    #pragma omp for nowait
    for(i=0;i<N;i++)
    { B[i]=big_calc2(C, i);}
    A[id] = big_calc4(id);
}
```
Master Construct

- The master construct denotes a structured block that is only executed by the master thread.
- The other threads just skip it

```c
#pragma omp parallel
{
    do_many_things();
    #pragma omp master
    { exchange_boundaries(); }
    #pragma omp barrier
    do_many_other_things();
}
```
Single worksharing Construct

• The single construct denotes a block of code that is executed by only one thread (not necessarily the master thread).

```c
#pragma omp parallel
{
    do_many_things();
    #pragma omp single
    {
        exchange_boundaries();
    }
    Do_other_things();
}
```
Synchronization: ordered

• Used in conjunction with a DO or SECTIONS construct to impose a serial order on the execution of a section of code.

```c
#pragma omp parallel private (tmp)
#pragma omp for ordered reduction(+:res)
for (I=0;I<N;I++) {
    tmp = NEAT_STUFF(I);
    #pragma ordered
    res += consum(tmp);
}
```
Data Sharing

- Data sharing is specified at the start of a parallel region or worksharing construct by using the SHARED and PRIVATE clauses.

- All variables in the SHARED clause are shared among the members of a team. The application must do the following:
  - Synchronize access to these variables.
  - Insure that all variables in the PRIVATE clause are private to each team member.
Data Sharing (Cont.)

- Initialize PRIVATE variables at the start of a parallel region, unless the FIRSTPRIVATE clause is specified.

- In this case, the PRIVATE copy is initialized from the global copy at the start of the construct at which the FIRSTPRIVATE clause is specified.

- Update the global copy of a PRIVATE variable at the end of a parallel region.

- However, the LASTPRIVATE clause of a DO directive enables updating the global copy from the team member that executed serially the last iteration of the loop.
Sections worksharing Construct

- Use the non iterative worksharing SECTIONS directive to divide the enclosed sections of code among the team.
- Each section is executed just one time by one thread.
- Each section should be preceded with a SECTION directive, except for the first section.
- The last section ends at the END SECTIONS directive.
- When a thread completes its section and there are no undispatched sections, it waits at the END SECTION directive unless you specify NOWAIT.
- The SECTIONS directive takes an optional comma-separated list of clauses that specifies which variables are PRIVATE, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION.
Schedule clause : loop worksharing constructs

• The schedule clause affects how loop iterations are mapped onto threads

• schedule(static [,chunk])
  – Deal-out blocks of iterations of size “chunk” to each thread.

• schedule(dynamic [,chunk])
  – Each thread grabs “chunk” iterations off a queue until all iterations have been handled.

• schedule(guided[chunk])
  – Threads dynamically grab blocks of iterations.

• schedule(runtime)
  – Schedule and chunk size taken from the OMP_SCHEDULE environment variable
Summary
Shared Memory Model

- Data can be shared or private
- Shared data is accessible by all threads
- Private data can be accessed only by the threads that owns it
- Data transfer is transparent to the programmer
Critical Construct

```c
sum = 0;
#pragma omp parallel private (lsum)
{
    lsum = 0;
    #pragma omp for
    for (i=0; i<N; i++) {
        lsum = lsum + A[i];
    }
    #pragma omp critical
    { sum += lsum; }
}
```

Threads wait their turn; only one thread at a time executes the critical section
Reduction Clause

sum = 0;
#pragma omp parallel for reduction (+:sum)
for (i=0; i<N; i++)
{
    sum = sum + A[i];
}

Shared variable
OpenMP example: PI

!$OMP PARALLEL PRIVATE(X, i)
   write (*,1004) omp_get_thread_num()
!$OMP DO REDUCTION (+:sum)
   DO i = 1, 1000000000, 1
       x = step*((-0.5)+i)
       sum = sum + 4.0/(1.0+x**2)
   ENDDO
!$OMP END DO NOWAIT
!$OMP END PARALLEL
OpenMP example

Running OpenMP applications on Steele

- qsub –I –l nodes=1:ppn=8
- module avail
- module load intel
- icc/ifort omp_pi.f –o omp_pi –openmp
- setenv OMP_NUM_THREADS 4
- time ./pi
OpenMP Scoping

• The OpenMP Data Scope Attribute Clauses are used to explicitly define how variable should be scoped. They include:
  – PRIVATE
  – FIRSTPRIVATE
  – LASTPRIVATE
  – SHARED
  – DEFAULT
  – REDUCTION
  – COPYIN
OpenMP Runtime Library Routines

- OMP_SET_NUM_THREADS
- OMP_GET_NUM_THREADS
- OMP_GET_MAX_THREADS
- OMP_GET_THREAD
- OMP_GET_NUM_PROCS
- OMP_IN_PARALLEL
- OMP_SET_DYNAMIC
- OMP_GET_DYNAMIC
- OMP_SET_NESTED
- OMP_GET_NESTED
OpenMP Lock Routines

- OMP_INIT_LOCK
- OMP_DESTROY_LOCK
- OMP_SET_LOCK
- OMP_UNSET_LOCK
- OMP_TEST_LOCK
For More Information…

• OpenMP
  – www.llnl.gov
  – www.openmp.org
The End