OpenMP
(with Labs)

Steve Lantz
Senior Research Associate
Cornell University Center for Advanced Computing (CAC)
slantz@cac.cornell.edu

Based on materials developed by Kent Milfeld at TACC

www.cac.cornell.edu
What is OpenMP?

- OpenMP is an acronym for Open Multi-Processing
- An Application Programming Interface (API) for developing parallel programs in shared-memory architectures
- Three primary components of the API are:
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- De facto standard -- specified for C, C++, and FORTRAN
- [http://www.openmp.org/](http://www.openmp.org/) has the specification, examples, tutorials and documentation
- OpenMP 4.0 specified July 2013
OpenMP = Multithreading

- All about executing concurrent work (tasks)
  - Tasks execute as independent *threads*
  - Threads access the same *shared memory* (no message passing!)
  - Threads synchronize only at *barriers*
- Simplest way to do multithreading – run tasks on multiple cores/units
  - Insert OpenMP *parallel directives* to create tasks for concurrent threads
  - So, shared-memory parallel programming is super-easy with OpenMP?
  - Nope! Updates to a shared variable, e.g., need special treatment…

```c
// repetitive work: OK
#pragma omp parallel for
for (i=0; i<N; i++)
a[i] = b[i] + c[i];

// repetitive updates: oops
#pragma omp parallel for
for (i=0; i<N; i++)
    sum = sum + b[i]*c[i];
```
Role of the Compiler

- OpenMP relies on the compiler to do the multithreading
  - Compiler recognizes OpenMP directives, builds in appropriate code
- A special flag is generally required to enable OpenMP
  - GNU: `gcc -fopenmp`
  - Intel: `icc -openmp`
- On Stampede, extra flags may be required for Intel Xeon Phi
  - May need to tell the Intel compiler to enable MIC instructions
  - Build OpenMP code native to MIC: `icc -openmp -mmic`
  - Offload marked sections to MIC: `icc -openmp`
  - Must do multithreading to make full use of the Phi!
Stampede OpenMP Use Cases

- **Two distinct pools of shared memory** exist on a Stampede node:
  - 32 GB for the Intel Xeon E5 host processors
  - 8 GB for the Intel Xeon Phi (MIC) coprocessor

- Thus, many possibilities exist for single-node OpenMP applications:
  - **Host only**: compile for E5, run threads on E5 cores
  - **MIC only**: compile natively for Phi, run threads on Phi cores
  - **Offload**: compile so code runs on E5 but offloaded sections run on Phi

- More possibilities exist for multi-node MPI/OpenMP **hybrid** codes:
  - **Symmetric**: compile separately for E5 and Phi, launch MPI tasks on both, spawn local OpenMP threads for all tasks
  - **Offload**: compile for E5, launch all MPI tasks on E5, offload some OpenMP sections to Phi
OpenMP Fork-Join Parallelism

- Programs begin as a single process: master thread
- Master thread executes until a parallel region is encountered
  - Master thread creates (forks) a team of parallel threads
  - Threads in team simultaneously execute tasks in the parallel region
  - Team threads synchronize and terminate (join); master continues
Parallel Region: C/C++ and Fortran

```
#pragma omp parallel
{
  code block
  a = work(...);
}
```

Line 1  Team of threads is formed at parallel region

Lines 2–3  Each thread executes code block and subroutine call, no branching into or out of a parallel region

Line 4  All threads synchronize at end of parallel region (implied barrier)

LAB: OMP Hello World

```
!$omp parallel
  code block
  call work(...)
!$omp end parallel
```
OpenMP on Shared Memory Systems

Hardware Model: Multiple Cores

Software Model: Threads in Parallel Region

M threads are usually mapped to M cores.
For HyperThreading, 2 SW threads are mapped to 2 HW threads on each core.
On the Intel Xeon Phi Coprocessors, there are 4 HW threads/core.
OpenMP Directives

- OpenMP directives are comments in source code that specify parallelism for shared-memory parallel (SMP) machines.
- FORTRAN compiler directives begin with one of the sentinels !$OMP, C$OMP, or *$OMP – use !$OMP for free-format F90.
- C/C++ compiler directives begin with the sentinel #pragma omp.

<table>
<thead>
<tr>
<th>Fortran 90</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>!$OMP parallel</td>
<td>#pragma omp parallel</td>
</tr>
<tr>
<td>...</td>
<td>{...}</td>
</tr>
<tr>
<td>!$OMP end parallel</td>
<td>}</td>
</tr>
<tr>
<td>!$OMP parallel do</td>
<td>#pragma omp parallel for</td>
</tr>
<tr>
<td>DO ...</td>
<td>for(...){{...}</td>
</tr>
<tr>
<td>!$OMP end parallel do</td>
<td>}</td>
</tr>
</tbody>
</table>
OpenMP Syntax

• OpenMP Directives: Sentinel, construct, and clauses

  #pragma omp construct [clause [[,]clause]…]  
  !$omp    construct [clause [[,]clause]…]  

  C  
  F90

  C  
  F90

  C  
  F90

• Most OpenMP constructs apply to a “structured block”, that is, a block of one or more statements with one point of entry at the top and one point of exit at the bottom.
OpenMP Constructs

OpenMP language “extensions”

- **parallel control**
  - governors flow of control in the program
  - **parallel directive**

- **work-sharing**
  - distributes work among threads
  - **do/for sections single directives**

- **control of one task**
  - assigns work to a thread
  - **task directive (OpenMP 3.0)**

- **data access**
  - specifies scoping of variables
  - **shared private reduction clauses**

- **synchronization**
  - coordinates execution of threads
  - **critical atomic barrier directives**

- **runtime environment**
  - sets/gets environment
    - **schedule omp_set_num_threads()**
    - **omp_get_thread_num()**
    - **OMP_NUM_THREADS**
    - **OMP_SCHEDULE clause, API, env. variables**
OpenMP Parallel Directives

- Replicated – executed by all threads
- Worksharing – divided among threads

```
PARALLEL
  {code}
END PARALLEL

PARALLEL DO
  do I = 1,N*4
    {code}
  end do
END PARALLEL DO

code
I=N+1,2N
I=2N+1,3N
I=3N+1,4N
I=1,N
```

Replicated

Worksharing

Combined
OpenMP Worksharing

Use OpenMP directives to specify worksharing in a parallel region, as well as synchronization.

```
#pragma omp parallel
{
    #pragma omp single
    #pragma omp master
    #pragma omp critical

    parallel do/for
    parallel sections
}
```

<table>
<thead>
<tr>
<th>Code block</th>
<th>Thread action</th>
</tr>
</thead>
<tbody>
<tr>
<td>for</td>
<td>Worksharing</td>
</tr>
<tr>
<td>sections</td>
<td>Worksharing</td>
</tr>
<tr>
<td>single</td>
<td>One thread</td>
</tr>
<tr>
<td>master</td>
<td>One thread</td>
</tr>
<tr>
<td>critical</td>
<td>One thread at a time</td>
</tr>
</tbody>
</table>

Directives can be combined, if a parallel region has just one worksharing construct.
Worksharing Loop: C/C++

```
#pragma omp parallel for
for (i=0; i<N; i++)
{
    a[i] = b[i] + c[i];
}
```

General form:
```
#pragma omp parallel
{
    #pragma omp for
    for (i=0; i<N; i++)
    {
        a[i] = b[i] + c[i];
    }
}
```

Line 1  Team of threads formed (parallel region).

Lines 2–6  Loop iterations are split among threads.
Implied barrier at end of block(s) {}.

Each loop iteration must be independent of other iterations.
Worksharing Loop: Fortran

General form:

```fortran
!$omp parallel do
  do i=1,N
    a(i) = b(i) + c(i)
  enddo
!$omp end parallel do
```

1. Team of threads formed (parallel region).
2. Loop iterations are split among threads.
3. (Optional) end of parallel loop (implied barrier at enddo).
4. Each loop iteration must be independent of other iterations.
OpenMP Clauses

• *Directives* dictate what the OpenMP thread team will do
• Examples:
  – *Parallel regions* are marked by the `parallel` directive
  – *Worksharing loops* are marked by `do, for` directives (Fortran, C/C++)
• *Clauses* control the behavior of any particular OpenMP directive
• Examples:
  1. Scoping of variables: `private, shared, default`
  2. Initialization of variables: `copyin, firstprivate`
  3. Scheduling: `static, dynamic, guided`
  4. Conditional application: `if`
  5. Number of threads in team: `num_threads`
Private, Shared Clauses

- In the following loop, each thread needs a private copy of temp
  - The result would be unpredictable if temp were shared, because each processor would be writing and reading to/from the same location

```c
!$omp parallel do private(temp,i) shared(A,B,C)
  do i=1,N
    temp = A(i)/B(i)
    C(i) = temp + cos(temp)
  enddo
!$omp end parallel do
```

- A “lastprivate(temp)” clause will copy the last loop (stack) value of temp to the (global) temp storage when the parallel DO is complete
- A “firstprivate(temp)” initializes each thread’s temp to the global value
Worksharing Results

Speedup = \( \frac{\text{cputime}(1)}{\text{cputime}(N)} \)

If work is completely parallel, scaling is linear.

Scheduling, memory contention and overhead can impact speedup and Gflop/s rate.
Overhead to Fork a Thread Team

- Increases roughly linearly with number of threads
Merging Parallel Regions

The !$OMP PARALLEL directive declares an entire region as parallel; therefore, merging work-sharing constructs into a single parallel region eliminates the overhead of separate team formations.

```
!$OMP PARALLEL
 do i=1,n
   a(i)=b(i)+c(i)
 enddo
!$OMP END PARALLEL DO
!$OMP PARALLEL
 do i=1,n
   a(i)=b(i)+c(i)
 enddo
!$OMP END PARALLEL DO
```

```
!$OMP PARALLEL
 !$OMP DO
   do i=1,n
     a(i)=b(i)+c(i)
   enddo
!$OMP END DO
!$OMP PARALLEL
 !$OMP DO
   do i=1,m
     x(i)=y(i)+z(i)
   enddo
!$OMP END DO
!$OMP END PARALLEL
```

```
!$OMP PARALLEL DO
  do i=1,n
    a(i)=b(i)+c(i)
  enddo
!$OMP END PARALLEL DO
!$OMP PARALLEL DO
  do i=1,m
    x(i)=y(i)+z(i)
  enddo
!$OMP END PARALLEL DO
!$OMP END PARALLEL
```
Thread Memory Access: Race Conditions

• Every thread accesses “global” or shared memory
  – All threads share the same address space except for private variables
  – Thus, threads have no need to pass messages like MPI processes…
• But race conditions can occur with shared memory. Examples:
  – The last writer “wins”, if no order is imposed on multiple writers
  – The reader who “loses” to a writer will acquire the newly-updated value
• A race condition leads to unpredictable results!
  – Avoid introducing one; usually it’s a bug which is hard to debug
• What do you do to prevent a race condition? Synchronize!
  – Impose order with barriers (explicit/implicit synchronization)
  – Use mutual exclusion (mutex) directives to protect critical sections,
    where one thread must run at a time (at a performance penalty)
Illustration of a Race Condition

<table>
<thead>
<tr>
<th>Intended</th>
<th>Possible…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
<td>Thread 1</td>
</tr>
<tr>
<td>read ←</td>
<td></td>
</tr>
<tr>
<td>increment</td>
<td></td>
</tr>
<tr>
<td>write →</td>
<td></td>
</tr>
<tr>
<td>read ←</td>
<td></td>
</tr>
<tr>
<td>increment</td>
<td></td>
</tr>
<tr>
<td>write →</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>read ←</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>increment</td>
<td>read ←</td>
<td>0</td>
</tr>
<tr>
<td>write →</td>
<td>increment</td>
<td>1</td>
</tr>
<tr>
<td>write →</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

- In a critical section, need *mutual exclusion* to get intended result
- The following OpenMP directives prevent this race condition:
  
  ```
  #pragma omp critical  – for a code block (C/C++)
  #pragma omp atomic    – for single statements
  ```
OpenMP Reduction

• Recall previous example of parallel dot product
  – Simple parallel-for doesn’t work due to race condition on shared sum
  – Best solution is to apply OpenMP’s reduction clause
  – Doing private partial sums is fine too; add a critical section for sum of ps

```c
// repetitive updates: oops
#pragma omp parallel for
for (i=0; i<N; i++)
    sum = sum + b[i]*c[i];

// repetitive reduction: OK
#pragma omp parallel for \
    reduction(+:sum)
for (i=0; i<N; i++)
    sum = sum + b[i]*c[i];
```
## Runtime Library Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>omp_get_num_threads()</code></td>
<td>Number of threads in current team</td>
</tr>
<tr>
<td><code>omp_get_thread_num()</code></td>
<td>Thread ID, {0: N-1}</td>
</tr>
<tr>
<td><code>omp_get_max_threads()</code></td>
<td>Number of threads in environment, <code>OMP_NUM_THREADS</code></td>
</tr>
<tr>
<td><code>omp_get_num_procs()</code></td>
<td>Number of machine CPUs</td>
</tr>
<tr>
<td><code>omp_in_parallel()</code></td>
<td>True if in parallel region &amp; multiple threads are executing</td>
</tr>
<tr>
<td><code>omp_set_num_threads(#)</code></td>
<td>Changes number of threads for parallel region, if dynamic threading is enabled</td>
</tr>
</tbody>
</table>
# Environment Variables, More Functions

- To control the OpenMP runtime environment

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMP_NUM_THREADS</td>
<td>Set to permitted number of threads: this is the value returned by <code>omp_get_max_threads()</code></td>
</tr>
<tr>
<td>OMP_DYNAMIC</td>
<td>TRUE/FALSE for enable/disable dynamic threading (can also use the function below)</td>
</tr>
</tbody>
</table>

- To enable dynamic thread count (*not* dynamic scheduling!)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>omp_set_dynamic()</td>
<td>Set state of dynamic threading: if equal to “true”, <code>omp_set_num_threads()</code> controls thread count</td>
</tr>
<tr>
<td>omp_get_dynamic()</td>
<td>True if dynamic threading is on</td>
</tr>
</tbody>
</table>
• OpenMP 3.0 supports nested parallelism, older implementations may ignore the nesting and serialize inner parallel regions.

• A nested parallel region can specify any number of threads to be used for the thread team, new id’s are assigned.
Additional Topics to Explore...

- Schedule clause: specify how to divide work among threads
  - `schedule(static)`  `schedule(dynamic,M)`

- Reduction clause: perform collective operations on shared variables
  - `reduction(+:asum)`  `reduction(*:aprod)`

- Nowait clause: remove the barrier at the end of a parallel section
  - `for ... nowait`  `end do nowait`

- Lock routines: make mutual exclusion more lightweight and flexible
  - `omp_init_lock(var)`  `omp_set_lock(var)}`
Some Programming Models for Intel MIC

- **Intel Threading Building Blocks (TBB)**
  - For C++ programmers
- **Intel Cilk Plus**
  - Task-oriented add-ons for OpenMP
  - Currently for C++ programmers, may become available for Fortran
- **Intel Math Kernel Library (MKL)**
  - Automatic offloading by compiler for some MKL features
  - MKL is inherently parallelized with OpenMP
- **OpenMP**
  - On Stampede, TACC expects that this is the most interesting programming model for HPC users
MIC Programming with OpenMP

- Compile with the Intel compiler (icc)
- OpenMP pragma is preceded by MIC-specific `pragma`
  - Fortran: `!dir$ omp offload target(mic) <...>`
  - C: `#pragma offload target(mic) <...>`
- All data transfer is handled by the compiler
  - User control provided through `optional keywords`
- I/O can be done from within offloaded region
  - Data can “stream” through the MIC; no need to leave MIC to fetch new data from disk
  - Also very helpful when debugging (print statements)
- Specific subroutines can be offloaded, including MKL subroutines
Example 1

2-D array \( \mathbf{a} \) is filled with data on the coprocessor

Data handling is done automatically by compiler

- Memory is allocated on coprocessor for \( \mathbf{a} \)
- Private variables \( i, j, x \) are created
- Result is copied back

```fortran
use omp_lib ! OpenMP
integer :: n = 1024 ! Size
real, dimension(:,:), allocatable :: a ! Array
integer :: i, j ! Index
real :: x ! Scalar
allocate(a(n,n)) ! Allocation
!dir$ omp offload target(mic) ! Offloading
#$omp parallel do shared(a,n), & ! Parallel -
     private(x, i, j), schedule(dynamic) ! region
do j=1, n
   do i=j, n
      x = real(i + j); a(i,j) = x
   end do
end do
```

```c
#include <omp.h> /* C example */
const int n = 1024; /* Size of the array */
int i, j; /* Index variables */
float a[n][n], x
#pragma omp offload target(mic)
#pragma omp parallel for shared(a), \\
     private(x), schedule(dynamic)
for(i=0;i<n;i++) {
   for(j=i;j<n;j++) {
      x = (float)(i + j); a[i][j] = x; }
}
Example 2

I/O from offloaded region:

- File is opened and closed by one thread ([omp single]
- All threads take turns reading from the file ([omp critical])

Threads may also read in parallel (not shown)

- Parallel file system
- Threads read parts from different targets

```c
#pragma offload target(mic) //Offload region
#pragma omp parallel
{
    #pragma omp single /* Open File */
    {
        printf("Opening file in offload region\n");
        f1 = fopen("/var/tmp/mydata/list.dat","r");
    }

    #pragma omp for
    for(i=1;i<n;i++) {
        #pragma omp critical
        { fscanf(f1,"%f",&a[i]);
            a[i] = sqrt(a[i]);
        }

    #pragma omp single
    { printf("Closing file in offload region\n");
      fclose (f1);
    }
}
```
Example 3

Two routines, MKL’s \texttt{sgemm} and \texttt{my_sgemm}

- Both are called with \texttt{offload} directive
- \texttt{my_sgemm} specifies explicit \texttt{in} and \texttt{out} data movement

Use \texttt{attributes} to have routine compiled for the coprocessor, or link coprocessor-based MKL

---

LAB: Hand-Coding vs. MKL

```plaintext
! snippet from the caller...
! offload MKL routine to accelerator
!dir$ attributes offload:mic :: sgemm
!dir$ offload target(mic)
Call sgemm('N','N',n,n,n,alpha,a,n,b,n,beta,c,n)
! offload hand-coded routine with data clauses
!dir$ offload target(mic) in(a,b) out(d)
call my_sgemm(d,a,b)
```

```plaintext
! snippet from the hand-coded subprogram...
!dir$ attributes offload:mic :: my_sgemm
subroutine my_sgemm(d,a,b)
real, dimension(:,:) :: a, b, d
!omp parallel do
do j=1, n
  do i=1, n
    d(i,j) = 0.0
    do k=1, n
      d(i,j) = d(i,j)+a(i,k)*b(k,j)
    enddo; enddo; endo
end subroutine
```
Heterogeneous Threading, Sequential

C/C++

```c
#pragma omp parallel
{
    #pragma omp single
    { offload(); }

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

F90

```fortran
!$omp parallel
!$omp single
    call offload();
!$omp end single

!$omp do
    do i=1,N; ... 
    end do
!$omp end parallel
```
Heterogeneous Threading, Concurrent

MPI process, master thread

Generate parallel region

offload single nowait

workshare on cpu

assist when done in single

wait

C/C++

```c
#pragma omp parallel
{
#pragma omp single nowait
    { offload(); }

#pragma omp for schedule(dynamic)
    for(i=0; i<N; i++) {...}
}
```

F90

```fortran
!$omp parallel
!$omp single
    call offload();
!$omp end single nowait

!$omp do schedule(dynamic)
    do i=1,N; ... 
    end do
!$omp end parallel
```