Introduction to OpenMP

Yun (Helen) He
Cray XE6 Workshop
February 7-8, 2011
Outline

• About OpenMP
• Parallel Regions
• Using OpenMP on Hopper
• Worksharing Constructs
• Synchronization
• Data Scope
• Tasks
• Hands-on Exercises
What is OpenMP

• OpenMP is an industry standard API of C/C++ and Fortran for shared memory parallel programming.
  – OpenMP Architecture Review Board
    • Major compiler vendors: PGI, Cray, Intel, Oracle, HP, Fujitsu, Microsoft, AMD, IBM, NEC, Texas Instrument, …
    • Research institutions: cOMPunity, DOE/NASA Labs, Universities…
  – 3.1 draft just came out today for public comment.
OpenMP Components

- **Compiler Directives and Clauses**
  - Interpreted when OpenMP compiler option is turned on.
  - Each directive applies to the succeeding structured block.

- **Runtime Libraries**

- **Environment Variables**
OpenMP Programming Model

• **Fork and Join Model**
  – Master thread only for all serial regions.
  – Master thread forks new threads at the beginning of parallel regions.
  – Multiple threads share work in parallel.
  – Threads join at the end of the parallel regions.

• **Each thread works on global shared and its own private variables.**

• **Threads synchronize implicitly by reading and writing shared variables.**
Serial vs. OpenMP

**Serial:**
```c
void main ()
{
    double x(256);
    for (int i=0; i<256; i++)
        {some_work(x[i]);}
}
```

**OpenMP:**
```c
#include "omp.h"
Void main ()
{
    double x(256);
    #pragma omp parallel for
        for (int i=0; i<256; i++)
            {some_work(x[i]);}
}
```

OpenMP is not just parallelizing loops! It offers a lot more ....
Advantages of OpenMP

• **Simple programming model**
  – Data decomposition and communication handled by compiler directives

• **Single source code for serial and parallel codes**

• **No major overwrite of the serial code**

• **Portable implementation**

• **Progressive parallelization**
  – Start from most critical or time consuming part of the code
OpenMP vs. MPI

– Pure OpenMP Pro:
  • Easy to implement parallelism
  • Low latency, high bandwidth
  • Implicit Communication
  • Coarse and fine granularity
  • Dynamic load balancing

– Pure OpenMP Con:
  • Only on shared memory machines
  • Scale within one node
  • Possible data placement problem
  • No specific thread order

– Pure MPI Pro:
  • Portable to distributed and shared memory machines.
  • Scales beyond one node
  • No data placement problem

– Pure MPI Con:
  • Difficult to develop and debug
  • High latency, low bandwidth
  • Explicit communication
  • Large granularity
  • Difficult load balancing
OpenMP Basic Syntax

- **Fortran:** case insensitive
  - Add: `use omp_lib` or `include “omp_lib.h”`
  - Fixed format
    - **Sentinel** directive *clauses*
    - **Sentinel** could be: `!$OMP`, `*$OMP`, `c$OMP`
  - Free format
    - `!$OMP` directive *clauses*
- **C/C++:** case sensitive
  - Add: `#include “omp.h”`
  - `#pragma omp` directive *clauses* newline
Compiler Directives

• **Parallel Directive**
  – Fortran: PARALLEL … END PARALLEL
  – C/C++: parallel

• **Worksharing Constructs**
  – Fortran: DO … END DO, WORKSHARE
  – C/C++: for
  – Both: sections

• **Synchronization**
  – master, single, ordered, flush, atomic

• **Tasking**
  – task, taskwait
Clauses

• private (list), shared (list)
• firstprivate (list), lastprivate (list)
• reduction (operator: list)
• schedule (method [, chunk_size])
• nowait
• if (scalar_expression)
• num_thread (num)
• threadprivate(list), copyin (list)
• ordered
• collapse (n)
• tie, untie
• And more …
Runtime Libraries

- Number of threads: `omp_{set,get}_num_threads`
- Thread ID: `omp_get_thread_num`
- Scheduling: `omp_{set,get}_dynamic`
- Nested parallelism: `omp_in_parallel`
- Locking: `omp_{init,set,unset}_lock`
- Active levels: `omp_get_thread_limit`
- Wallclock Timer: `omp_get_wtime`
  - thread private
  - call function twice, use difference between end time and start time
- And more …
Environment Variables

- OMP_NUM_THREADS
- OMP_SCHEDULE
- OMP_STACKSIZE
- OMP_DYNAMIC
- OMP_NESTED
- OMP_WAIT_POLICY
- OMP_ACTIVE_LEVELS
- OMP_THREAD_LIMIT
- And more …
The parallel Directive

FORTRAN:
!$OMP PARALLEL PRIVATE(id)
   id = omp_get_thread_num()
   write (*,*) "I am thread", id
!$OMP END PARALLEL

C/C++:
#pragma omp parallel private(thid)
{
   thid = omp_get_thread_num();
   printf("I am thread %d\n", thid);
}

• The parallel directive forms a team of threads for parallel execution.
• Each thread executes within the OpenMP parallel region.
A Simple Hello_World OpenMP Program

C/C++:

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main () {
    int tid, nthreads;
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        printf("Hello World from thread %d\n", tid);
        #pragma omp barrier
        if ( tid == 0 ) {
            nthreads = omp_get_num_threads();
            printf("Total threads= %d\n",nthreads);
        }
    }
}
```

FORTRAN:

```fortran
Program main
use omp_lib (or: include "omp_lib.h")
integer :: id, nthreads
!$OMP PARALLEL PRIVATE(id)
   id = omp_get_thread_num()
   write (*,*) "Hello World from thread", id
!$OMP BARRIER
   if ( id == 0 ) then
      nthreads = omp_get_num_threads()
      write (*,*) "Total threads=",nthreads
   end if
!$OMP END PARALLEL
End program
```
Compile OpenMP on Hopper

• Use compiler wrappers:
  – ftn for Fortran codes
  – cc for C codes, CC for C++ codes

• Portland Group Compilers
  – Add compiler option “-mp=nonuma”
  – % ftn –mp=nonuma mycode.f90
  – Supports OpenMP 3.0 from pgi/8.0

• Pathscale Compilers
  – % module swap PrgEnv-pgi PrgEnv-pathscale
  – Add compiler option “-mp”
  – % ftn –mp=nonuma mycode.f90
Compile OpenMP on Hopper (2)

- **GNU Compilers**
  - `% module swap PrgEnv-pgi PrgEnv-gnu`
  - Add compiler option “-fopenmp”
  - `% ftn –fopenmp mycode.f90`
  - Supports OpenMP 3.0 from gcc/4.4

- **Cray Compilers**
  - `% module swap PrgEnv-pgi PrgEnv-cray`
  - No additional compiler option needed
  - `% ftn mycode.f90`
  - Supports OpenMP 3.0
Run OpenMP on Hopper

• Each Hopper node has 4 NUMA nodes, each with 6 UMA cores.

• Pure OpenMP code could use up to 24 threads per node.

• Interactive batch jobs:
  – Pure OpenMP example, using 6 OpenMP threads:
  – % qsub –I –V –q interactive –lmpwidth=24
  – wait for a new shell
  – % cd $PBS_O_WORKDIR
  – setenv OMP_NUM_THREADES 6
  – setenv PSC_OMP_AFFINITY FALSE (note: for Pathscale only)
  – % aprun –n 1 –N 1 –d 6 ./mycode.exe
Run OpenMP on Hopper (2)

Sample batch script:
(pure OpenMP example, Using 6 OpenMP threads)

```bash
#PBS -q debug
#PBS -l mppwidth=24
#PBS -l walltime=00:10:00
#PBS -j eo
#PBS –V
cd $PBS_O_WORKDIR
setenv OMP_NUM_THREADS 6

# uncomment this line for pathscale
# setenv PSC_OMP_AFFINITY FALSE

aprun -n 1 -N 1 -d 6 ./mycode.exe
```

- **Run batch jobs:**
  - Prepare a batch script first
  - `% qsub myscript`

- **Notice to use for pathscale:**
  - `setenv PSC_OMP_AFFINITY FALSE`
First Hands-on Exercise

Get the Source Codes:
% cp –r /project/projedirs/training/XE6-feb-2011/openmp .

Compile and Run:
% cd openmp
% ftn –mp=nonuma –o hello_world hello_world.f90
(or % cc –mp=nonuma –o hello_world hello_world.c)
% qsub –V –I –q interactive –lmpwidth=24
...
% cd $PBS_O_WORKDIR
% setenv OMP_NUM_THREADS 6 (for csh/tcsh)
(or % export OMP_NUM_THREADS=6 for bash/ksh)
% aprun –n 1 –N 1 –d 6 ./hello_world

Sample Output: (no specific order)
Hello World from thread 0
Hello World from thread 2
Hello World from thread 4
Hello World from thread 3
Hello World from thread 5
Hello World from thread 1
Total threads= 6
Loop Parallelism

C/C++:
```c
#pragma omp parallel [clauses]
{
    #pragma omp for [clauses]
    {
        for (int i=0; i<1000; i++) {
            a[i] = b[i] + c[i];
        }
    }
}
```

Fortran:
```
!$OMP PARALLEL [Clauses]
...
!$OMP DO [Clauses]
    do i = 1, 1000
        a (i) = b(i) + c(i)
    enddo
!$OMP END DO [NOWAIT]
...
!$OMP PARALLEL
```

- Threads share the work in loop parallelism.
- For example, using 4 threads under the default “static” scheduling, in Fortran:
  - thread 1 has i=1-250
  - thread 2 has i=251-500, etc.
Combined Parallel Worksharing Constructs

**FORTRAN:**

```fortran
!$OMP PARALLEL DO
  do i = 1, 1000
    a (i) = b(i) + c(i)
  enddo
!$OMP PARALLEL END DO
```

**FORTRAN example:**

```fortran
!$OMP PARALLEL SECTIONS
!$OMP SECTION
  do i = 1, 1000
    c (i) = a(i) + b(i)
  enddo
!$OMP SECTION
  do i = 1, 1000
    d(i) = a(i) * b(i)
  enddo
!$OMP PARALLEL END SECTIONS
```

**FORTRAN only:**

```fortran
INTEGER N, M
PARAMETER (N=100)
REAL A(N,N), B(N,N), C(N,N), D(N,N)
!$OMP PARALLEL WORKSHARE
  C = A + B
  M = 1
  D= A * B
!$OMP PARALLEL END WORKSHARE
```

**C/C++:**

```c
#pragma omp parallel for
  for (int i=0; i<1000; i++) {
    a[i] = b[i] + c[i];
  }
```

**C/C++ only:**

```c
```
The schedule Clause

- **Static**: Loops are divided into \#thrds partitions.
- **Guided**: Loops are divided into progressively smaller chunks until the chunk size is 1.
- **Dynamic, \#chunk**: Loops are divided into chunks containing \#chunk iterations.
- **Auto**: The compiler (or runtime system) decides what to use.
- **Runtime**: Use OMP_SCHEDULE environment variable to determine at run time.
Second Hands-on Exercise

Sample codes: schedule.f90

-- Experiment with different number of threads.
-- Run this example multiple times.

% ftn –mp=nonuma –o schedule schedule.f90
% qsub –V –l –q debug –lmppwidth=24
...
% cd $PBS_O_WORKDIR
% setenv OMP_NUM_THREADS 3
% aprun –n 1 –N 1 –d 4 ./schedule
% setenv OMP_NUM_THREADS 6
...

-- Compare scheduling results with different scheduling algorithm.
-- Results change with dynamic schedule at different runs.
Third Hands-on Exercise

Sample code: sections.f90

-- Experiment with different number of threads.
-- Run this example multiple times.

% ftn –mp=nonuma –o sections.f90
% qsub –V –I –q debug –lmppwidth=24
…
% cd $PBS_O_WORKDIR
% setenv OMP_NUM_THREADS 3
% aprun –n 1 –N 1 –d 3 ./sections
% setenv OMP_NUM_THREADS 5
% aprun –n 1 –N 1 –d 5 ./sections

-- What happens when more sections than threads?
-- What happens when more threads than sections?
**Loop Parallelism: ordered and collapse**

**FORTRAN example:**
```fortran
!$OMP DO ORDERED
   do i = 1, 1000
      a (i) = b(i) + c(i)
   enddo
!$OMP END DO
```

**FORTRAN example:**
```fortran
!$OMP DO COLLAPSE (2)
   do i = 1, 1000
      do j = 1, 100
         a(i,j) = b(i,j) + c(i,j)
      enddo
   enddo
!$OMP END DO
```

- **ordered** specifies the parallel loop to be executed in the order of the loop iterations.
- **collapse (n)** collapse the $n$ nested loops into 1, then schedule work for each thread accordingly.
Loop-based vs. SPMD

Loop-based:

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

SPMD (Single Program Multiple Data):

```c
!$OMP PARALLEL DO PRIVATE(start, end, i)
!$OMP&                   SHARED(a,b)
    num_thrds = omp_get_num_threads()
    thrd_id = omp_get_thread_num()
    start = n * thrd_id/num_thrds + 1
    end = n * (thrd_num+1)/num_thrds
    do i = start, end
        a(i) = a(i) + b(i)
    enddo
!$OMP END PARALLEL DO
```

SPMD code normally gives better performance than loop-based code, but is more difficult to implement:

- Less thread synchronization.
- Less cache misses.
- More compiler optimizations.
The reduction Clause

C/C++ example:
```c
int i;
#pragma omp parallel reduction(*:i)
{
    i=omp_get_num_threads();
}
printf(“result=%d\n”,i);
```

Fortran example:
```fortran
sum = 0.0
$OMP parallel reduction (+: sum)
do i =1, n
    sum = sum + x(i)
enddo
$OMP end do
$OMP end parallel
```

- **Syntax:** `Reduction (operator : list)`.  
- Reduces list of variables into one, using operator.  
- Reduced variables must be shared variables.  
- **Allowed Operators:**  
  - Arithmetic: `+ - * /`  
  - Fortran intrinsic: `max min`  
  - Bitwise: `& | ^`  
  - Logical: `&& ||`
Synchronization: the barrier Directive

FORTRAN:
```
!$OMP PARALLEL
    do i = 1, n
        a(i) = b(i) + c(i)
    enddo
!$OMP BARRIER
    do i = 1, n
        e(i) = a(i) * d(i)
    enddo
!$OMP END PARALLEL
```

C/C++:
```
#pragma omp parallel
{   ... some work;
    #pragma omp barrier
    ... some other work;
}
```

- Every thread waits until all threads arrive at the barrier.
- Barrier makes sure all the shared variables are (explicitly) synchronized.
Synchronization: the critical Directive

**FORTRAN:**
```
!$OMP PARALLEL SHARED (x)
   ... some work ...
!$OMP CRITICAL [name]
   x = x + 1.0
!$OMP END CRITICAL
   ... some other work ...
!$OMP END PARALLEL
```

**C/C++:**
```
#pragma omp parallel shared (x)
{
    #pragma omp critical
    {
        x = x + 1.0;
    }
}
```

- Each thread executes the critical region one at a time.
- Multiple critical regions with no name are considered as one critical region: single thread execution at a time.
### Synchronization: the master and single Directives

<table>
<thead>
<tr>
<th>FORTRAN:</th>
<th>C/C++:</th>
</tr>
</thead>
</table>
| !$OMP MASTER  
  ... some work ...  
 !$OMP END MASTER | #pragma omp master  
 {  
   ... some work ...  
 } |

<table>
<thead>
<tr>
<th>FORTRAN:</th>
<th>C/C++:</th>
</tr>
</thead>
</table>
| !$OMP SINGLE  
  ... some work ...  
 !$OMP END SINGLE | #pragma omp single  
 {  
   ... some work ...  
 } |

- **Master region:**
  - Only the master threads executes the MASTER region.
  - No implicit barrier at the end of the MASTER region.

- **Single region:**
  - First thread arrives the SINGLE region executes this region.
  - All threads wait: implicit barrier at end of the SINGLE region.
Synchronization: the **atomic** and **flush** Directives

**FORTRAN:**

```
!$OMP ATOMIC
... some memory update ...
```

**C/C++:**

```
#pragma omp atomic
... some memory update ...
```

**FORTRAN:**

```
!$OMP FLUSH [(var_list)]
```

**C/C++:**

```
#pragma omp flush [(var_list)]
```

- **Atomic:**
  - Only applies to the immediate following statement.
  - Atomic memory update: avoids simultaneous updates from multiple threads to the same memory location.

- **Flush:**
  - Makes sure a thread’s temporary view to be consistent with the memory.
  - Applies to all thread visible variables if no `var_list` is provided.
• In general, IO operations, general OS functionality, common library functions may not be thread safe. They should be performed by one thread only or serialized.

• Avoid race condition in OpenMP program.
  – Race condition: Multiple threads are updating the same shared variable simultaneously.
  – Use “critical” directive
  – Use “atomic” directive
  – Use “reduction” directive
Fourth Hands-on Exercise

Sample codes: pi.c, pi_omp_wrong.c, pi_omp1.c, pi_omp2.c, pi_omp3.c

-- Understand different versions of calculating pi.
-- Understand the race condition in pi_omp_wrong.c
-- Run multiple times with different number of threads

% qsub pi.pbs

Or:
% ftn –mp=nonuma –o pi_omp3.f90
% qsub –V –l –q debug –lmpwidth=24
...
% cd $PBS_O_WORKDIR
% setenv OMP_NUM_THREADS 16
% aprun –n 1 –N 1 –d 16 ./pi_omp3

-- Race condition generates different results.
-- Needs critical or atomic for memory updates.
-- Reduction is an efficient solution.
Data Scope

• **Most variables are shared by default:**
  - Fortran: common blocks, SAVE variables, module variables
  - C/C++: file scope variables, static
  - Both: dynamically allocated variables

• **Some variables are private by default:**
  - Certain loop indexes
  - Stack variables in subroutines or functions called from parallel regions
  - Automatic (local) variables within a statement block
Data Sharing: the firstprivate Clause

FORTRAN Example:

```fortran
PROGRAM MAIN
  USE OMP_LIB
  INTEGER I
  I = 1
  !$OMP PARALLEL FIRSTPRIVATE(I) &
  !$OMP PRIVATE(tid)
  I = I + 2  ! I=3
  tid = OMP_GET_THREAD_NUM()
  if (tid ==1) PRINT *, I  ! I=3
  !$OMP END PARALLEL
  PRINT *, I  ! I=1
END PROGRAM
```

- Declares the variables in the list private
- Initializes the variables in the list with the value when they first enter the construct.
Data Sharing: the lastprivate Clause

FORTRAN example:

Program main
Real A(100)
 !$OMP parallel shared (A) &
 !$OMP do lastprivate(i)
 DO I = 1, 100
  A(I) = I + 1
 ENDDO
 !$OMP end do
 !$OMP end parallel
 PRINT*, I ! I=101
 end program

• Declares the variables in the list private
• Updates the variables in the list with the value when they last exit the construct.
Data Sharing: the threadprivate and copyin Clauses

**FORTRAN Example:**

```fortran
PROGRAM main
  use OMP_LIB
  INTEGER tid, K
  COMMON /T/K
!$OMP THREADPRIVATE(/T/)
  K = 1

!$OMP PARALLEL PRIVATE(tid) COPYIN(/T/)
  PRINT *, "thread ", tid, ",K= ", K
  tid = omp_get_thread_num()
  K = tid + K
  PRINT *, "thread ", tid, ",K= ", K
!$OMP END PARALLEL

!$OMP PARALLEL PRIVATE(tid)
  tid = omp_get_thread_num()
  PRINT *, "thread ", tid, ",K= ", K
!$OMP END PARALLEL
END PROGRAM main
```

- A threadprivate variable has its own copies of the global variables and common blocks.
- A threadprivate variable has its scope across multiple parallel regions, unlike a private variable.
- The copyin clause: copies the threadprivate variables from master thread to each local thread.
Tasking: the task and taskwait Directives

Serial:
int fib (int n) {
    int x, y;
    if (n < 2) return n;
    x = fib (n – 1);
    y = fib (n – 2);
    return x+y;
}

OpenMP:
int fib (int n) {
    int x,y;
    if (n < 2) return n;
    #pragma omp task shared (x)
    x = fib (n – 1);
    #pragma omp task shared (y)
    y = fib (n – 2);
    #pragma omp taskwait
    return x+y;
}

• Major OpenMP 3.0 addition. Flexible and powerful.
• The task directive defines an explicit task.
• Threads share work from all tasks in the task pool.
• The taskwait directive makes sure all child tasks created for the current task finish.
Thread Affinity

- Thread affinity: forces each thread to run on a specific subset of processors, to take advantage of local process state.
- Current OpenMP standard has no specification for thread affinity.
- On Cray XE6, there is aprun command option “-cc”:
  - -cc cpu (default): Each PE’s thread is constrained to the CPU closest to the PE.
  - -cc numa_node: Each PE’s thread is constrained to the same NUMA node CPUs.
  - -cc none: Each thread is not binded to a specific CPU.
• OMP_STACKSIZE defines the private stack space each thread has.
• Default value is implementation dependent, and is usually quite small.
• Behavior is undefined if run out of space, mostly segmentation fault.
• To change, set OMP_STACKSIZE to \( n \) (B,K,M,G) bytes. For example:
  setenv OMP_STACKSIZE 16M
Fifth Hands-on Exercise

Sample codes: jacobi_serial.f90 and jacobi_omp.f90

-- Check the OpenMP features used in the real code.
-- Understand code speedup.

% qsub jacobi.pbs

Or:
% ftn –mp=nonuma –o jacobi_omp.f90
% qsub –V –l –q debug –lmpwidth=24
...
% cd $PBS_O_WORKDIR
% setenv OMP_NUM_THREADS 6
% aprun –n 1 –N 1 –d 6 ./jacobi_omp
% setenv OMP_NUM_THREADS 12
% aprun –n 1 –N 1 –d 12 ./jacobi_omp

-- Why not perfect speedup?
Performance Results

<table>
<thead>
<tr>
<th>Jacobi OpenMP</th>
<th>Execution Time (sec)</th>
<th>Speedup</th>
<th>Execution Time (sec) (larger input)</th>
<th>Speedup (larger input)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 thread</td>
<td>21.7</td>
<td>1</td>
<td>668</td>
<td>1</td>
</tr>
<tr>
<td>2 threads</td>
<td>11.1</td>
<td>1.96</td>
<td>337</td>
<td>1.98</td>
</tr>
<tr>
<td>4 threads</td>
<td>6.0</td>
<td>3.62</td>
<td>171</td>
<td>3.91</td>
</tr>
<tr>
<td>6 threads</td>
<td>4.3</td>
<td>5.05</td>
<td>116</td>
<td>5.76</td>
</tr>
<tr>
<td>12 threads</td>
<td>2.7</td>
<td>8.03</td>
<td>60</td>
<td>11.13</td>
</tr>
<tr>
<td>24 threads</td>
<td>1.8</td>
<td>12.05</td>
<td>36</td>
<td>18.56</td>
</tr>
</tbody>
</table>

- Why not perfect speedup?
  - Serial code sections not parallelized
  - Thread creation and synchronization overhead
  - Memory bandwidth
  - Memory access with cache coherence
  - Load balancing
  - Not enough work for each thread
General Programming Tips

• Start from an optimized serial version.
• Gradually add OpenMP, check progress, add barriers.
• Decide which loop to parallelize. Better to parallelize outer loop. Decide whether loop permutation, fusion, exchange or collapse is needed.
• Use different OpenMP task scheduling options.
• Adjust environment variables.
• Choose between loop-based or SPMD.
• Minimize shared, maximize private, minimize barriers.
• Minimize parallel constructs, if possible use combined constructs.
• Take advantage of debugging tools: totalview, DDT, etc.
More OpenMP Examples

- On NERSC machines: Franklin, Hopper2, and Carver:
  - % module load training
  - % cd $EXAMPLES/OpenMP/tutorial
- Try to understand, compile and run available examples.
  - Examples prepared by Richard Gerber, Mike Stewart, and Helen He
- Have fun!
Further References

• OpenMP 3.0 specification, and Fortran, C/C++ Summary cards. 
  http://openmp.org/wp/openmp-specifications/
• IWOMP2010 OpenMP Tutorial. Rudd van der Pas. 
  http://www.compunity.org/training/tutorials/3%20Overview_OpenMP.pdf
• Shared Memory Programming with OpenMP. Barbara Chapman, at UCB 
  2010 Par Lab Boot Camp. 
  http://parlab.eecs.berkeley.edu/sites/all/parlab/files/openmp-berkeley- 
  chapman-slides_0.pdf
• SC08 OpenMP Tutorial. Tim Mattson and Larry Meadows. 
  www.openmp.org/mp-documents/omp-hands-on-SC08.pdf
• Using OpenMP. Barbara Chapman, Gabrielle Jost, and Rudd van der Pas. 
• LLNL OpenMP Tutorial. Blaise Barney. 
  http://computing.llnl.gov/tutorials/openMP
• NERSC OpenMP Tutorial. Richard Gerber and Mike Stewart. 
  http://www.nersc.gov/nusers/help/tutorials/openmp