Multithreading

- A programming model that allows multiple *threads* to exist in a single process, sharing the *same memory space*
- Thread – A single sequence of programming instructions

- Why multithreading?
  - Speedup – For an application that scales well, the speedup increases linearly with number of threads, theoretically
Goal

- In this talk, you will learn how to:
  - Parallelize portions of your application
  - Use the OpenMP API
  - Avoid common pitfalls in shared memory programming
What is OpenMP?

- OpenMP is an acronym for Open Multi-Processing
- An Application Programming Interface (API) for developing multithreading parallel program in shared memory architecture
- Three primary components of the API are:
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- OpenMP standard is maintained by the OpenMP Architecture Review Board
- [http://www.openmp.org/](http://www.openmp.org/) has the specification, examples, tutorials and documentation
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

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)
OpenMP: Multithreading

• All about executing concurrent work (tasks)
  – Tasks execute independently
  – Tasks access the same shared memory
  – Shared variable updates must be mutually exclusive
  – Synchronization through barriers

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

// repetitive updates
#pragma omp parallel for
for (i=0; i<N; i++)
  sum = sum + b[i]*c[i];
OpenMP: 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.
Thread Memory Access

- Every thread has access to “global” (shared) memory
  - All threads share the same address space
  - Threads don’t communicate like MPI processes
- But need to avoid race conditions with shared memory. Examples:
  1. If multiple writers are going in no particular order, last writer “wins”
  2. A reader may either precede or follow a writer – lack of synchronization
  3. Threads may overlap in a code block, causing conditions 1 and 2
- What do you with a race condition?
  - Don’t introduce one in the first place: it’s a bug, hard to debug
  - 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)
Example of a Critical Section

- 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
  ```

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>increment</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>write</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>read</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>increment</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>write</td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

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

- OpenMP directives:
  - specify parallelism for shared-memory parallel (SMP) machines
  - Appear like comments in source code
- FORTRAN compiler: !$OMP, C$OMP, or *$OMP – use !$OMP for free-format F90
- C/C++ compiler directive: #pragma omp

C/C++

```c
#pragma omp parallel
{
  ...
}

#pragma omp parallel for
for(...){...
}
```

Fortran 90

```fortran
$OMP parallel
...
$OMP end parallel

$OMP parallel do
for(...){...
}
```

6/11/2013 www.cac.cornell.edu
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`
OpenMP Compiler Directive Syntax

- OpenMP Directives: **Sentinel**, **construct**, and **clauses**
  
  ```
  #pragma omp construct [clause [[,]clause]...]  
  !$omp construct [clause [[,]clause]...]  
  ```

- Example
  
  ```
  #pragma omp parallel private(i) reduction(+:sum)  
  !$omp parallel private(i) reduction(+:sum)  
  ```

- 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
- governs 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 Construct

- 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
```

```
PARALLEL
  {code1}
END PARALLEL

DO
  do I = 1,N*4
    {code2}
  end do
END DO

{code3}
```

```
PARALLEL
  {code1}
END PARALLEL

DO
  do I = 1,N*4
    {code2}
  end do
END DO

{code3}
```

Replicated

Worksharing

Combined
OpenMP Worksharing, Mutual Exclusion

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

```c
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp single critical
    }
}
```

<table>
<thead>
<tr>
<th>Code block</th>
<th>Thread action</th>
</tr>
</thead>
<tbody>
<tr>
<td>parallel do/for</td>
<td>Worksharing</td>
</tr>
<tr>
<td>parallel sections</td>
<td>Worksharing</td>
</tr>
<tr>
<td>single</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++

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
```

Line 1  Team of threads formed (parallel region).
Lines 2–5  Loop iterations are split among threads.
Line 5  (Optional) end of parallel loop (implied barrier at enddo).

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

- Private: Variable is private to each thread
- Shared: Variable is shared among all threads
- 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
```
Worksharing Results

Speedup = cputime(1) / cputime(N)

If work is completely parallel, scaling is linear.

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

- Increases roughly linearly with number of threads
Merging Parallel Regions

Merging work-sharing constructs eliminates the overhead

```c
!$OMP PARALLEL
!$OMP DO
    do i=1,n
        a(i)=b(i)+c(i)
    enddo
!$OMP END DO
!$OMP PARALLEL DO
!$OMP DO
    do i=1,m
        x(i)=y(i)+z(i)
    enddo
!$OMP END DO
!$OMP END PARALLEL
```
## 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 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><strong>OMP_NUM_THREADS</strong></td>
<td>Set to permitted number of threads: this is the value returned by <code>omp_get_max_threads()</code></td>
</tr>
<tr>
<td><strong>OMP_DYNAMIC</strong></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><code>omp_set_dynamic()</code></td>
<td>Set state of dynamic threading: if equal to “true”, <code>omp_set_num_threads()</code> controls thread count</td>
</tr>
<tr>
<td><code>omp_get_dynamic()</code></td>
<td>True if dynamic threading is on</td>
</tr>
</tbody>
</table>
Additional OpenMP Features

- Schedule clause: specify how to divide work among threads
  
  \[
  \text{schedule}(\text{static}) \quad \text{schedule}(\text{dynamic}, M)
  \]

- Reduction clause: perform collective operations on shared variables
  
  \[
  \text{reduction}(+: \text{asum}) \quad \text{reduction}(\times: \text{aprod})
  \]

- Nowait clause: remove the barrier at the end of a parallel section
  
  \[
  \text{for} \ldots \ \text{nowait} \quad \text{end do nowait}
  \]

- Lock routines: make mutual exclusion more lightweight and flexible
  
  \[
  \text{omp_init_lock}(\text{var}) \quad \text{omp_set_lock}(\text{var})
  \]
Loop Nesting

- 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.
MIC Programming with OpenMP

• Intel compiler, *icc*, is needed
• 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
  – Also very helpful when debugging (print statements)
• Specific subroutines are automatically offloaded, including MKL subroutines
Example 1

2-D array (a) is filled with data on the coprocessor.

Data management done automatically by compiler:

- Memory is allocated on coprocessor for (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), & ! Par. region
private(x, i, j), schedule(dynamic)
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

Hand-coded routine, my_sgemm:
• specifies explicit in and out data movement

Use attributes to have routine compiled for the coprocessor, or link coprocessor-based MKL

LAB: Hand-coding vs. MKL

! snippet from the caller...
! offload MKL routine to accelerator
! offload hand-coded routine with data clauses
!dir$ offload target(mic) in(a,b) out(d)
call my_sgemm(d,a,b)

! 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)
    endo; endo; endo
endo subroutine
Heterogeneous Threading, Sequential

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

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

#pragma omp parallel
#pragma omp single
call offload();
#pragma omp end single

#pragma omp do
  do i=1,N; ...
  end do
#pragma omp end parallel
```

C/C++

F90
Heterogeneous Threading, Concurrent

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
```
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);
    }
}
```