Objectives

At the completion of this module you will be able to

- Thread serial code with basic OpenMP pragmas
- Use OpenMP synchronization pragmas to coordinate thread execution and memory access
Agenda

What is OpenMP?
Parallel Regions
Worksharing Construct
Data Scoping to Protect Data
Explicit Synchronization
Scheduling Clauses
Other Helpful Constructs and Clauses

What Is OpenMP*?

Compiler directives for multithreaded programming
Creating teams of threads
sharing work among threads
synchronizing the threads

Library routines for setting and querying thread attributes

Environment variables for controlling run-time behavior of the parallel program

Easy to create threaded Fortran and C/C++ codes
Supports data parallelism model
Incremental parallelism
Combines serial and parallel code in single source
What Is OpenMP*?

http://www.openmp.org

Current spec is OpenMP 3.0
May 2008
326 Pages
(combined C/C++ and Fortran/Fortran 90)

Previous Version
OpenMP 2.5 - May 2005

OpenMP* Architecture

Fork-join model
Work-sharing constructs
Data environment constructs
Synchronization constructs
Extensive Application Program Interface (API) for finer control
Programming Model

Fork-join parallelism:
- **Master thread** spawns a team of threads as needed
- Parallelism is added incrementally: the sequential program evolves into a parallel program

OpenMP* Pragma Syntax

Most constructs in OpenMP* are compiler directives or pragmas.
- For C and C++, the pragmas take the form:

```
#pragma omp construct [clause [clause]...]
```
Parallel Regions

Defines **parallel region** over structured block of code

Threads are created as ‘**parallel’** pragma is crossed

Threads block at end of region

Data is shared among threads unless specified otherwise

**C/C++:**

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

---

How Many Threads?

Set environment variable for number of threads

```
set OMP_NUM_THREADS=4
```

There is no standard default for this variable

- Many systems:
  - # of threads = # of processors
  - Intel® compilers use this default
Work-sharing Construct

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

Splits loop iterations into threads
Must be in the parallel region
Must precede the loop

Work-sharing Construct

```c
#pragma omp parallel
#pragma omp for
g for(i = 0; i < 12; i++)
c[i] = a[i] + b[i]
```

Threads are assigned an independent set of iterations
Threads must wait at the end of work-sharing construct
Combining pragmas

These two code segments are equivalent

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

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

Restriction on Loop Threading in Ver 2.5

- The loop variable must be `signed` integer
- The comparison operation must be in the form
  
  `loop_variable op loop_invariant_integer`
  
  where `op` is `<`, `<=`, `>`, `>=`
- The third expression or increment portion must be either integer addition or subtraction and by `loop-invariant value`
- If the comparison operation is `<` or `<=`, the loop variable must increment on every iteration. If it is `>` or `>=`, the loop variable must decrement on every iteration.
- The loop must be `single entry and exit`. No jumps in or out of the loop are permitted. Goto or breaks must jump within loop. Exception must be handled in loop.
Restriction on Loop Threading in Ver 3.0

- The loop variable must be
  - **signed** or unsigned integer
  - C++ : random access iterator or pointer type
- The comparison operation must be in the form
  
  \[
  \text{loop_variable} \text{ op } \text{loop_invariant_expression} \text{ (compatible type)}
  \]
  
  where \( \text{op} \) is \(<, \leq, >, \geq\)
- The third expression or increment portion must be either addition or subtraction and by **loop-invariant integer expression**
- If the comparison operation is \(< \text{ or } \leq\), the loop variable must increment on every iteration. If it is \(> \text{ or } \geq\), the loop variable must decrement on every iteration.
- The loop must be **single entry and exit**. No jumps in or out of the loop are permitted. Goto or breaks must jump within loop. Execution must be handled in loop.

Non-Integer Loop Variables in Ver 3.0

The computation is performed for each loop in an integer type. This type is derived from the type of \(\text{var}\) as follows:

- If \(\text{var}\) is of an integer type, then the type is the type of \(\text{var}\).
- For C++, if \(\text{var}\) is of a random access iterator type, then the type is the type that would be used by `std::distance` applied to variables of the type of \(\text{var}\).
- For C, if \(\text{var}\) is of a pointer type, then the type is `ptrdiff_t`.

The behavior is unspecified if any intermediate result required to compute the iteration count cannot be represented in the type determined above.
Challenges in Threading Loops

- Loop threading is effectively a reordering transformation of loop
  - Valid if loop carries no dependence
- Conditions for Loop-carried dependence
  - Statement S2 is data dependent on statement S1 if
    1. S1 and S2 both reference memory location L for some execution path
    2. Execution of S1 occurs before S2
- Flow dependence
  - S1 writes L and L is later read by S2
- Output dependence
  - Both S1 and S2 write L
- Anti-dependence
  - S1 reads L before S2 writes L

Loop-carried and loop-independent dependence

- Loop-carried
  - S1 references L on one iteration; S2 references it on a subsequent iteration
- Loop-independent
  - S1 and S2 reference L on same loop iteration, but S1 executes it before S2
### Examples of loop-carried dependence

- **Loop-carried flow dependence**

<table>
<thead>
<tr>
<th></th>
<th>S1</th>
<th>S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write L</td>
<td>Read L</td>
<td></td>
</tr>
</tbody>
</table>

- **Loop-carried anti-dependence**

<table>
<thead>
<tr>
<th></th>
<th>S1</th>
<th>S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read L</td>
<td>Write L</td>
<td></td>
</tr>
</tbody>
</table>

**Loop-carried output dependence**

<table>
<thead>
<tr>
<th></th>
<th>S1</th>
<th>S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write L</td>
<td>Write L</td>
<td></td>
</tr>
</tbody>
</table>

### Loop-carried dependence example

```c
x[0] = 0;
y[0] = 1;
#pragma omp parallel for private(k)
   for (k=1; k<100; k++) {
      x[k] = y[k-1] + 1;  //s1
      y[k] = x[k] + 2;    //s2
   }
```
Loop-carried dependence example

```c
x[0] = 0;
y[0] = 1;
#pragma omp parallel for private(k)
for (k=1; k<100; k++)  {
    x[k] = y[k-1] + 1;  //s1  anti-dependence
    y[k] = x[k] + 2;    //s2  flow dependence
}
```

What happens?

- OpenMP will thread the loop
- Threaded code will fail
- What to do – remove the loop-carried dependence
- Two approaches
  - divide the loop in 2 nested loops
  - Use parallel sections
- Need to predetermine x[49] and y[49] !!!!
### Nested loops

```c
x[0] = 0; x[49] = 74;
y[0] = 1; y[49] = 74;
#pragma omp parallel for private(m, k)
for (m = 0; m < 2; m++) {
    for (k = m*49 + 1; k < m*50 + 50; k++) {
        x[k] = y[k-1] + 1;  // s1  anti-dependence
        y[k] = x[k-1] + 2;  // s2  flow dependence
    }
}
```

### Parallel Sections

Independent sections of code can execute concurrently

```c
#pragma omp parallel sections
{  
    #pragma omp section
       phase1();
    #pragma omp section
       phase2();
    #pragma omp section
       phase3();
}
```
Parallel Sections

```c
#pragma omp parallel sections private(k)
#pragma omp section
{ { 
    x[0] = 0; y[0] = 1;
    for (k=1; k<50; k++) {
        x[k] = y[k-1] + 1; // s1 anti-dependence
        y[k] = x[k-1] + 2; // s2 flow dependence
    }
} }
#pragma omp section
{ { 
    x[49] = 74; y[49] = 74;
    for (k=50; k<100; k++) {
        x[k] = y[k-1] + 1; // s1 anti-dependence
        y[k] = x[k-1] + 2; // s2 flow dependence
    }
} }
```

Data Environment

OpenMP uses a shared-memory programming model

- Most variables are shared by default.
- Global variables are shared among threads
  - C/C++: File scope variables, static
Data Environment

But, not everything is shared...

- Stack variables in functions called from parallel regions are PRIVATE
- Automatic variables within a statement block are PRIVATE
- Loop index variables are private (with exceptions)
  - C/C++: The first loop index variable in nested loops following a
    
    #pragma omp for

Data Scope Attributes

The default status can be modified with

```
default (shared | none)
```

Scoping attribute clauses

```
shared(varname,...)
```

```
private(varname,...)
```
The Private Clause

Reproduces the variable for each thread
- Variables are un-initialized; C++ object is default constructed
- Any value external to the parallel region is undefined

```c
void* work(float* c, int N) {
    float x, y; int i;
    #pragma omp parallel for private(x,y)
    for(i=0; i<N; i++) {
        x = a[i]; y = b[i];
        c[i] = x + y;
    }
}
```

Example: Dot Product

```c
float dot_prod(float* a, float* b, int N) {
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
        sum += a[i] * b[i];
    }
    return sum;
}
```

What is Wrong?
Protect Shared Data

Must protect access to shared, modifiable data

```c
float dot_prod(float* a, float* b, int N)
{
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
        #pragma omp critical
        sum += a[i] * b[i];
    }
    return sum;
}
```

OpenMP* Critical Construct

```c
#pragma omp critical [(lock_name)]
```

Defines a critical region on a structured block

Threads wait their turn – at a time, only one calls `consum()` thereby protecting R1 and R2 from race conditions.

Naming the critical constructs is optional, but may increase performance.
OpenMP* Reduction Clause

```
reduction (op : list)
```

The variables in “list” must be shared in the enclosing parallel region.

Inside parallel or work-sharing construct:
- A PRIVATE copy of each list variable is created and initialized depending on the “op”.
- These copies are updated locally by threads.
- At end of construct, local copies are combined through “op” into a single value and combined with the value in the original SHARED variable.

Reduction Example

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

Local copy of `sum` for each thread.

All local copies of `sum` added together and stored in “global” variable.
C/C++ Reduction Operations

A range of associative and commutative operators can be used with reduction
Initial values are the ones that make sense

<table>
<thead>
<tr>
<th>Operator</th>
<th>Initial Value</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>^</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operator</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>~0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Numerical Integration Example

\[
\int_{0}^{4.0} \frac{4.0}{1 + x^2} \, dx
\]

```c
static long num_steps=100000;
double step, pi;

void main()
{
  int i;
  double x, sum = 0.0;

  step = 1.0/(double) num_steps;
  for (i=0; i< num_steps; i++){
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0 + x*x);
  }
  pi = step * sum;
  printf("Pi = \%f\n",pi);
}
```
Activity 2 - Computing Pi

```c
static long num_steps=100000;
double step, pi;

void main()
{
  int i;
  double x, sum = 0.0;
  step = 1.0/(double) num_steps;
  for (i=0; i< num_steps; i++){
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0 + x*x);
  }
  pi = step * sum;
  printf("Pi = %f\n",pi);
}
```

Parallelize the numerical integration code using OpenMP

What variables can be shared?
What variables need to be private?
What variables should be set up for reductions?

Assigning Iterations

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

- **schedule(static [,chunk])**
  - Blocks of iterations of size “chunk” to threads
  - Round robin distribution

- **schedule(dynamic[,chunk])**
  - Threads grab “chunk” iterations
  - When done with iterations, thread requests next set

- **schedule(guided[,chunk])**
  - Dynamic schedule starting with large block
  - Size of the blocks shrink; no smaller than “chunk”
Scheduling defaults

- If the schedule clause is missing, an implementation dependent schedule is selected. GOMP default is the static schedule where iterations are distributed approximately evenly among threads.
- Static scheduling has low overhead and provides better data locality since iterations generally touch memory sequentially.
- Dynamic and guided scheduling may provide better load balancing.
- Dynamic scheduling handles chunks on a first-come first-served basis with chunk size 1.

Guided Scheduling

- Allocates decreasingly large iterations to each thread until size reaches C. A variant of dynamic scheduling in which the size of the chunk decreases exponentially.
- Algorithm for chunk size:
  - N is number of threads
  - B₀ is number of loop iterations
  - Cₖ is size of kth chunk
  - Bₖ is number of loop iterations remaining when calculating the chunk size Cₖ
  - Cₖ = ceil( Bₖ / 2N )
  - When Cₖ chunk size gets too small it is set to C specified in the schedule clause (default 1)
  - Example: B₀ = 800, N=2, C=80
    - Partition is 200, 150, 113, 85, 80, 80, 12
  - Guided scheduling performs better than dynamic due to less overhead.
**Static Scheduling: Doing It By Hand**

Must know:
- Number of threads (Nthrds)
- Each thread ID number (id)

Compute start and end iterations:

```c
#pragma omp parallel
{
    int i, istart, iend;
    istart = id * N / Nthrds;
    iend = (id+1) * N / Nthrds;
    for(i=istart;i<iend;i++)
    {
        c[i] = a[i] + b[i];
    }
}
```

**Which Schedule to Use**

<table>
<thead>
<tr>
<th>Schedule Clause</th>
<th>When To Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATIC</td>
<td>Predictable and similar work per iteration</td>
</tr>
<tr>
<td>DYNAMIC</td>
<td>Unpredictable, highly variable work per iteration</td>
</tr>
<tr>
<td>GUIDED</td>
<td>Special case of dynamic to reduce scheduling overhead</td>
</tr>
</tbody>
</table>
Clause – schedule(type, size)

schedule(type, size)

- schedule(static)
  - Allocates ceiling(N/t) contiguous iterations to each thread, where N is the number of iterations and t is the number of threads
- schedule(static, C)
  - Allocates C contiguous iterations to each thread
- schedule(dynamic)
  - Allocates 1 iteration at a time, dynamically.
- schedule(dynamic, C)
  - Allocates C iterations at a time, dynamically. When a thread is ready to receive new work, it is assigned the next pending chunk of size C
- schedule(guided)
  - Allocates decreasingly large iterations to each thread until size reaches C. A variant of dynamic scheduling in which the size of the chunk decreases exponentially from chunk to C. Default value for chunk is ceiling(N/p)
- schedule(guided, C)
  - Same as (guided, C), with C = 1
- schedule(runtime)
  - Indicates that the schedule type and chunk are specified by environment variable OMP_SCHEDULE
  - Example of run-time specified scheduling
    setenv OMP_SCHEDULE “dynamic,2”

Schedule Clause Example

```c
#pragma omp parallel for schedule (static, 8)
for( int i = start; i <= end; i += 2 )
{
    if ( TestForPrime(i) ) gPrimesFound++;
}
```

Iterations are divided into chunks of 8
- If start = 3, then first chunk is i={3,5,7,9,11,13,15,17}
Single Construct

Denotes block of code to be executed by only one thread

- Thread chosen is implementation dependent

Implicit barrier at end

```c
#pragma omp parallel
{
  DoManyThings();
#pragma omp single
  { 
    ExchangeBoundaries();
  } // threads wait here for single
  DoManyMoreThings();
}
```

Master Construct

Denotes block of code to be executed only by the master thread

No implicit barrier at end

```c
#pragma omp parallel
{
  DoManyThings();
#pragma omp master
  { // if not master skip to next stmt
    ExchangeBoundaries();
  }
  DoManyMoreThings();
}
```
Implicit Barriers

Several OpenMP* constructs have implicit barriers

- parallel
- for
- single

Unnecessary barriers hurt performance
- Waiting threads accomplish no work!

Suppress implicit barriers, when safe, with the nowait clause

Nowait Clause

Use when threads would wait between independent computations

```
#define omp for nowait
for(...) 
{ (...) }
```

```
#pragma single nowait
{ [...] }
```

```
#pragma omp for schedule(dynamic,1) nowait
for(int i=0; i<n; i++)
    a[i] = bigFunc1(i);

#pragma omp for schedule(dynamic,1)
for(int j=0; j<m; j++)
    b[j] = bigFunc2(j);
```
Barrier Construct

Explicit barrier synchronization
Each thread waits until all threads arrive

```c
#pragma omp parallel shared (A, B, C)
{
  DoSomeWork(A, B);
  printf("Processed A into B\n");
#pragma omp barrier
  DoSomeWork(B, C);
  printf("Processed B into C\n");
}
```

Atomic Construct

Special case of a critical section
Applies only to simple update of memory location

```c
#pragma omp parallel for shared(x, y, index, n)
for (i = 0; i < n; i++) {
#pragma omp atomic
  x[index[i]] += work1(i);
  y[i] += work2(i);
}
```
OpenMP* API

Get the thread number within a team

```
int omp_get_thread_num(void);
```

Get the number of threads in a team

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

Usually not needed for OpenMP codes

- Can lead to code not being serially consistent
- Does have specific uses (debugging)
- Must include a header file

```
#include <omp.h>
```

Points to note on reductions

1. Value of the reduction variable is undefined from time first thread reaches the region/loop with reduction clause and remains so until reduction is completed

2. If the loop has a `nowait` clause, the reduction variable remains undefined until a barrier synch is performed

3. The order in which the local values are combined is undefined, so the answer may be different to the serial one due to rounding effects
Monte Carlo Pi

\[ \frac{\text{# of darts hitting circle}}{\text{# of darts in square}} = \frac{1}{4 \pi r^2} \]

\[ \pi = 4 \frac{\text{# of darts hitting circle}}{\text{# of darts in square}} \]

Loop 1 to MAX

\[ x \text{.coor} = \text{(random#)} \]
\[ y \text{.coor} = \text{(random#)} \]
\[ \text{dist} = \sqrt{x^2 + y^2} \]
\[ \text{if } (\text{dist} \leq 1) \]
\[ \text{hits} = \text{hits} + 1 \]
\[ \text{pi} = 4 \times \text{hits}/\text{MAX} \]

Making Monte Carlo’s Parallel

**hits = 0**
call SEED48(1)
DO I = 1, max
\[ x = \text{DRAND48}() \]
\[ y = \text{DRAND48}() \]
\[ \text{IF } (\sqrt{x^2 + y^2} < 1) \text{ THEN} \]
\[ \text{hits} = \text{hits} + 1 \]
ENDIF
END DO
\[ \text{pi} = \text{REAL(hits)}/\text{REAL(max)} \times 4.0 \]

What is the challenge here?
Activity 3: Computing Pi

Use the Intel® Math Kernel Library (Intel® MKL) VSL:
- Intel MKL’s VSL (Vector Statistics Libraries)
- VSL creates an array, rather than a single random number
- VSL can have multiple seeds (one for each thread)

Objective:
- Use basic OpenMP* syntax to make Pi parallel
- Choose the best code to divide the task up
- Categorize properly all variables

Programming with OpenMP
What’s Been Covered

OpenMP* is:
- A simple approach to parallel programming for shared memory machines

We explored basic OpenMP coding on how to:
- Make code regions parallel (omp parallel)
- Split up work (omp for)
- Categorize variables (omp private...)
- Synchronize (omp critical...)

We reinforced fundamental OpenMP concepts through several labs