Introduction to Programming with OpenMP

Kent Milfeld
milfeld@tacc.utexas.edu
February 6th 2012, TACC
Outline

• What is OpenMP?

• How does OpenMP work?
  – Architecture
  – Fork-Join model of parallelism
  – Communication

• OpenMP Syntax
  – Compiler Directives
  – Runtime Library Routines
  – Environment variables

• What’s new? OpenMP 3.1
What is OpenMP?

• OpenMP stands for Open Multi-Processing

• An Application Programming Interface (API) for developing parallel programs for shared memory architectures

• Three primary components of the API are:
  – Compiler Directives
  – Runtime Library Routines
  – Environment Variables

• Standard specifies C, C++, and FORTRAN Directives & API

• http://www.openmp.org/ has the specification, examples, tutorials and documentation
Architecture

- Data: shared or private
- Shared data: all threads can access data in shared memory
- Private data: can only be accessed by threads that own it
- Data transfer is transparent to the programmer
OpenMP Fork-Join Parallelism

- Programs begin as a single process: master thread
- Master thread executes in serial mode until the parallel region construct is encountered
- Master thread creates a team of parallel threads (fork) that simultaneously execute statements in the parallel region
- After executing the statements in the parallel region, team threads synchronize and terminate (join) but master continues
Thread Memory Access

• Every thread has access to “global” memory (shared)

• All threads share the same address space

• Threads can synchronize through barriers, and communicate by reading/writing to the global memory

• Simultaneous updates to shared memory can create a race condition. Results change with different thread scheduling

• Use mutual exclusion to avoid data sharing - but don’t use too many because this will serialize performance
OpenMP Syntax

• Most of the constructs in OpenMP are compiler directives
  
  ```
  #pragma omp construct [clause [clause]...] C
  !$omp construct [clause [clause]...] F90
  ```

• Example
  
  ```
  #pragma omp parallel num_threads(4) C
  !$omp parallel num_threads(4) F90
  ```

• Function prototypes and types are in the file:
  
  ```
  #include <omp.h> C
  use omp_lib 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 structures
- parallel control work sharing
- data environment
- synchronization
- runtime environment

parallel directive

• governs flow of control in the program

do for directives

• distributes work among threads

shared private clauses

• specifies variables as shared or private

critical atomic barrier directive

• coordinates thread execution

omp_set_num_threads()
omp_get_thread_num()
OMP_NUM_THREADS
OMP_SCHEDULE

• scheduling static, dynamic, guided

runtime functions
• environment variables
OpenMP Directives

• OpenMP directives begin with special comments/pragmas that tell open-aware compilers that a parallel regions follows. Directive begin with sentinels:
  FORTRAN: !$OMP, C$OMP or *$OMP for fixed-format
  F90: !$OMP free-format
  C/C++: # pragma omp

• Parallel regions are marked by enclosing parallel directives
• Work-sharing loops are marked by parallel do/for

Fortran

!$OMP parallel
...
!$OMP end parallel

!$OMP parallel do
do ...
!$OMP end parallel do

C/C++

# pragma omp parallel
{
...
}

# pragma omp parallel for
for(){...}
Parallel Region & Work-Sharing

Use OpenMP directives to specify Parallel Region & Work-Sharing constructs

Parallel

End Parallel

Parallel DO/for
Parallel SECTIONS

Code block

Each Thread Executes

DO
Work Sharing
SECTIONS
Work Sharing
SINGLE
One Thread (Work sharing)
CRITICAL
One Thread at a time

Work-Sharing
Parallel Region
Parallel Regions

```c
1 #pragma omp parallel
2 {
3     code block
4     work(...);
5 }
```

Line 1  Team of threads formed at parallel region
Lines 3-4 Each thread executes code block and subroutine calls. No branching (in or out) in a parallel region
Line 5  All threads synchronize at end of parallel region (implied barrier)

Use the thread number to divide work among threads
Parallel Regions

1  !$omp parallel
2  code block
3  call work(…)
4  !$omp end parallel

Line 1  Team of threads formed at parallel region.
Lines 2-3 Each thread executes code block and subroutine calls. No branching (in or out) in a parallel region.
Line 4  All threads synchronize at end of parallel region (implied barrier).

Use the thread number to divide work among.
Parallel Regions & Modes

There are two OpenMP “modes”

- **static** mode
  - Fixed number of threads -- set in the `OMP_NUM_THREADS` env.
  - Or the threads may be set by a function call (or clause) inside the code:
    - `omp_set_num_threads()` runtime function
    - `num_threads(#)` clause (on directive line)

- **dynamic** mode:
  - Number of threads can change under OS control from one parallel region to another using:

**Note:** the user can only define the maximum number of threads, compiler can use a smaller number
Work sharing: Loop

F90

1  !$omp parallel do
2  do i=1,N
3   a(i) = b(i) + c(i)
4  enddo
5  !$omp end parallel do

Line 1  Team of threads formed (parallel region).
Line 2-4 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.
Work-Sharing: Loop

C/C++

1       #pragma parallel for
2       for (i=0; i<N; i++)
3          {
4              a[i] = b[i] + c[i];
5          }  // Line 2-5 Loop iterations are split among threads.
              // implied barrier at end of block {}.

Line 1   Team of threads formed (parallel region).
Line 2-5 Loop iterations are split among threads.
          implied barrier at end of block {}.

Each loop iteration must be independent of other iterations.
Work-Sharing: Sections

C/C++

```c
#pragma omp sections
{
    #pragma omp section
    {work_1();}
    #pragma omp section
    { work_2(); } 
    ...
}
```

F90

```f90
!$omp sections
!$omp section
  work_1();
!$omp section
  work_2();
!$omp end sections
```

Line 1  Team of threads formed (parallel region).
Line 3-8 One thread is working on each section.
Line 9   End of parallel sections with an implied barrier.

Scales only to the number of sections.
Replicated: Work blocks are executed by all threads.
Worksharing: Work is divided among threads.

Assume thread count of 4

- **Replicated**: All threads execute the same code block.
- **Work-sharing**: Work is divided among threads.
- **Combined**: Work blocks are distributed among threads with overlapping execution.

```
PARALLEL
  {code}
END PARALLEL
```

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

```
PARALLEL
  {code1}
DO
  do I = 1, N*4
    {code2}
    {code}
  end do
  {code3}
END PARALLEL
```

OpenMP Parallel Constructs

---

TACC

Texas Advanced Computing Center
OpenMP Clauses

```c
#pragma omp directive-name [ clause [ [,] clause ] ... ]
!$omp directive-name [ clause [ [,] clause ] ... ]
```

Clauses control the behavior of an OpenMP directive:

1. **Schedule** (Guided, Static, Dynamic, runtime)
2. **Data scoping** (Private, Shared, Default)
3. **Initialization** (e.g. COPYIN, FIRSTPRIVATE, LASTPRIVATE)
4. Whether to parallelize a region or not (if-clause)
5. Number of threads used (NUM_THREADS)
Schedule Clause

**schedule(static)**
Each CPU receives one set of contiguous iterations

**schedule(static, C)**
Iterations are divided round-robin fashion in chunks of size C

**schedule(dynamic, C)**
Iterations handed out in chunks of size C as CPUs become available

**schedule(guided, C)**
Each of the iterations are handed out in pieces of exponentially decreasing size, with C minimum number of iterations to dispatch each time

**schedule (runtime)**
Schedule and chunk size taken from the OMP_SCHEDULE environment variable

```c
#pragma parallel for schedule
for (i=0; i<N; i++)
{
    a[i] = b[i] + c[i];
}
```
## Comparison of Scheduling Options

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>chunk</th>
<th>chunk size</th>
<th>chunk #</th>
<th>static or dynamic</th>
<th>compute overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple static</td>
<td>static</td>
<td>no</td>
<td>N/P</td>
<td>P</td>
<td>static</td>
<td>lowest</td>
</tr>
<tr>
<td>Interleaved (round robin)</td>
<td>static</td>
<td>yes</td>
<td>C</td>
<td>N/C</td>
<td>static</td>
<td>low</td>
</tr>
<tr>
<td>simple dynamic</td>
<td>dynamic</td>
<td>optional</td>
<td>C</td>
<td>N/C</td>
<td>dynamic</td>
<td>medium</td>
</tr>
<tr>
<td>guided</td>
<td>Guided</td>
<td>optional</td>
<td>decreasing from N/P</td>
<td>fewer than N/C</td>
<td>dynamic</td>
<td>high</td>
</tr>
<tr>
<td>runtime</td>
<td>runtime</td>
<td>no</td>
<td>varies</td>
<td>varies</td>
<td>varies</td>
<td>varies</td>
</tr>
</tbody>
</table>
Example - schedule(static,16), threads = 4

```c
#pragma omp parallel do schedule(static,16)
do i=1,128
    A(i)=B(i)+C(i)
enddo
```

<table>
<thead>
<tr>
<th>Thread 0: i=1,16</th>
<th>Thread 2: i=33,48</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(i)=B(i)+C(i)</td>
<td>A(i)=B(i)+C(i)</td>
</tr>
<tr>
<td>do i=65,80</td>
<td>do i=97,112</td>
</tr>
<tr>
<td>A(i)=B(i)+C(i)</td>
<td>A(i)=B(i)+C(i)</td>
</tr>
<tr>
<td>enddo</td>
<td>enddo</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 1: i=17,32</th>
<th>Thread 3: i=49,64</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(i)=B(i)+C(i)</td>
<td>A(i)=B(i)+C(i)</td>
</tr>
<tr>
<td>do i=81,96</td>
<td>do i=113,128</td>
</tr>
<tr>
<td>A(i)=B(i)+C(i)</td>
<td>A(i)=B(i)+C(i)</td>
</tr>
<tr>
<td>enddo</td>
<td>enddo</td>
</tr>
</tbody>
</table>
OpenMP Data Environment

- Data scoping clauses control the sharing behavior of variables within a parallel construct.
- These include `shared`, `private`, `firstprivate`, `lastprivate`, `reduction` clauses

Default variable scope:

1. Variables are shared by default
2. Global variables are shared by default
3. Automatic variables within subroutines called from within a parallel region are private (reside on a stack private to each thread), unless scoped otherwise
4. Default scoping rule can be changed with `default` clause
Private & Shared Data

**shared** - Variable is shared (seen) by all processors

**private** - Each thread has a private instance (copy) of the variable

Defaults: A worksharing for-loop index is private, all other variables are shared

### C/C++

```c
#pragma omp parallel for \
shared(a,b,c,n) private(i)
for (i=0; i<n; i++){
    a[i] = b[i] + c[i];
}
```

### F90

```fortran
!$omp parallel do &
!$omp shared(a,b,c,n) private(i)
do i=0, n;
    a(i) = b(i) + c(i)
end do
```

All threads have access to the same storage areas for a, b, c, and n, but each loop has its own private copy of the loop index, i
Private Data Example

• In the following loop, each thread needs its own private copy of temp

• If temp were shared, the result would be unpredictable since each thread would be writing and reading to/from the same memory location

```
tmp=1.0
#pragma omp parallel for \\shared(a,b,c,n) \\
private(tmp,i)
for (i=0; i<n; i++){
    tmp = a[i] / b[i];
    c[i] = tmp + cos(tmp);
}
```

C/C++

```
!omp parallel for 
!omp shared(a,b,c,n) & 
!omp private(tmp,i)
do i=0,n
    tmp = a(i) / b(i);
    c(i) = tmp + cos(tmp);
end do
```

F90

• A `lastprivate(temp)` clause will copy the last loop(stack) value of tmp to the (global) temp storage when the parallel DO is complete.

• A `firstprivate(temp)` would copy the global temp value to each stack’s temp.
Parallel Region & Number of Threads

- Work is divided up by thread id in routine foo.

```c
double a[1000]; int id, nt;
...
NT = omp_get_num_threads();
#pragma omp parallel private(id)
{
    id = omp_get_thread_num();
    foo(id, NT, a, N);
}
```

- Each thread redundantly executes the code within the structured block
- Each thread calls foo(id,nt,a) for \( id = 0 \) to \#threads - 1
Parallel Region & Number of Threads

• Work is divided up by thread id in routine foo.

```fortran
real*8 a(1000); integer id,nt
...
NT=omp_get_num_threads()
!$omp parallel private(id)

    id = omp_get_thread_num();
call foo(id,NT, a,N);
!$omp end parallel
```

• Each thread redundantly executes the code within the structured block
• Each thread calls foo(id,nt,a) for id = 0 to #threads - 1
Parallel Region & Number of Threads

• Work is divided up by thread id in routine foo.

```c++
double a[1000]; int NT; ... NT = omp_get_num_threads(); #pragma omp parallel
{ int id = omp_get_thread_num(); foo(id,NT, a,N); }
```

id is private because it is generated within the parallel regions.

• Each thread redundantly executes the code within the structured block
• Each thread calls foo(id,nt,a) for id = 0 to #threads - 1
Reduction

• Operation that combines multiple elements to form a single result
• A variable that accumulates the result is called a reduction variable
• In parallel loops reduction operators and variables must be declared

```c/c++
float asum, aprod;
asum  = 0.;
aprod = 1.;
#pragma omp parallel for 
reduction(+:asum) reduction(*:aprod)
for (i=0; i<n; i++){
    asum  = asum  + a[i];
aprod = aprod * a[i];
}
```

Each thread has a private `asum` and `aprod`, initialized to the operator’s identity

• After the loop execution, the master thread collects the private values of each thread and finishes the (global) reduction
Synchronization

• Synchronization is used to impose order constraints and to protect access to shared data

• High-Level Synchronization
  – critical
  – atomic
  – barrier
  – ordered

• Low-Level Synchronization
  – locks
Synchronization: Critical/Atomic Directives

- When each thread must execute a section of code serially the region must be marked with `critical/end critical` directives.
- Use the `#pragma omp atomic` directive if executing only one operation serially.

```
#pragma omp parallel shared(sum,x,y)
...
#pragma omp critical
{
    update(x);
    update(y);
    sum=sum+1;
}
...
!$OMP END PARALLEL
```

```
#pragma omp parallel shared(sum)
...
{
    #pragma omp atomic
    sum=sum+1;
   ...
}
```

Diagram:
- Master Thread
- CRITICAL section or atomic operations

---

TACC
THE UNIVERSITY OF TEXAS AT AUSTIN
TEXAS ADVANCED COMPUTING CENTER
Synchronization: Barrier

- Barrier: Each thread waits until all threads arrive

```c
#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);
}  // Implicit barrier
#pragma omp for nowait
```

Implicit barrier

No implicit barrier due to nowait

Implicit barrier
Synchronization: Barrier

- Barrier: Each thread waits until all threads arrive

```c
!$omp parallel shared (A, B, C) private(id)

    id=omp_get_thread_num()
    A(id) = big_calc1(id)
    !$omp barrier
    !$omp do
    do i=0,N
        C(i)=big_calc3(i,A)
    end do  \---------- Implicit barrier
    !$omp do
    do i=1,N
        B(i)=big_calc2(C, i)
    end do  \---------- No implicit barrier due to nowait
    A(id} = big_calc4(id)

!$omp end parallel  \---------- Implicit barrier
```
Mutual Exclusion: Lock Routines

When each thread must execute a section of code serially locks provide a more flexible way of ensuring serial access than **CRITICAL** and **ATOMIC** directives.

```plaintext
call OMP_INIT_LOCK(lockvar)
!$OMP PARALLEL SHARED(X,Y)
...
call OMP_set_lock(lockvar)
call update(x)
call OMP_unset_lock(lockvar)
...
!$OMP END PARALLEL
call OMP.Destroy_LOCK(lockvar)
```
Synchronization: Ordered

• The ordered region executes in the sequential order

```c
#pragma omp parallel private (tmp)
#pragma omp for ordered reduction(+:count)
for (i=0;i<N;i++){
    tmp = foo(i);
    #pragma omp ordered
    count += consume(tmp);
}
```
Mutual Exclusion Overhead

<table>
<thead>
<tr>
<th>OMP exclusion directive</th>
<th>cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMP_SET_LOCK</td>
<td>330</td>
</tr>
<tr>
<td>OMP_UNSET_LOCK</td>
<td>330</td>
</tr>
<tr>
<td>OMP_ATOMIC</td>
<td>480</td>
</tr>
<tr>
<td>OMP_CRITICAL</td>
<td>510</td>
</tr>
</tbody>
</table>

All measurements made in dedicated mode
NOWAIT

• When a work-sharing region is exited, a barrier is implied - all threads must reach the barrier before any can proceed.

• By using the NOWAIT clause at the end of each loop inside the parallel region, an unnecessary synchronization of threads can be avoided.

```c
#pragma omp parallel
{
#pragma omp for nowait
{
    for (i=0; i<n; i++)
    {
        work(i);
    }
}
#pragma omp for schedule(dynamic,k)
{
    for (i=0; i<m; i++)
    {
        x[i]=y[i]+z[i];
    }
}
}
```
# Runtime Library Routines

<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 team, N</td>
</tr>
<tr>
<td><code>omp_get_thread_num()</code></td>
<td>Thread ID {0 -&gt; N-1}</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 thread executing</td>
</tr>
<tr>
<td><code>omp_set_num_threads(#)</code></td>
<td>Set the number of threads in the team</td>
</tr>
<tr>
<td><code>omp_get_dynamic()</code></td>
<td>True if dynamic threading is on</td>
</tr>
<tr>
<td><code>omp_set_dynamic()</code></td>
<td>Set state of dynamic threading (true/false)</td>
</tr>
</tbody>
</table>
## Environment Variables

<table>
<thead>
<tr>
<th>variable</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OMP_NUM_THREADS</strong></td>
<td>Set to default no. of threads to use</td>
</tr>
<tr>
<td><strong>OMP_SCHEDULE</strong></td>
<td>Control how “omp for schedule(runtime)” loop iterations are scheduled</td>
</tr>
<tr>
<td>Syntax: “schedule[, chunk_size]”</td>
<td></td>
</tr>
<tr>
<td><strong>OMP_DYNAMIC</strong></td>
<td>TRUE/FALSE for enable/disable dynamic threading</td>
</tr>
</tbody>
</table>
OpenMP Wallclock Timers

```plaintext
real*8 :: omp_get_wtime, omp_get_wtick()

(double omp_get_wtime(), omp_get_wtick());
```

```c
double t0, t1, dt, res;
...

t0 = omp_get_wtime();
<work>
t1 = omp_get_wtime();
dt = t1 - t0;
res = 1.0/omp_get_wtick();
printf("Elapsed time = %lf\n", dt);
printf("clock resolution = %lf\n", res);
```
OpenMP 3.0

• First update to the spec since 2005
• Tasking: move beyond loops with generalized tasks and support complex and dynamic control flows
• Loop collapse: combine nested loops automatically to expose more concurrency
• Enhanced loop schedules: Support aggressive compiler optimizations of loop schedules and give programmers better runtime control over the kind of schedule used
• Nested parallelism support: better definition of and control over nested parallel regions, and new API routines to determine nesting structure
Tasks Parallelism

- Allows to parallelize irregular problems
  - Recursive loops
  - Unbounded algorithms
  - Threads can jump between tasks
What is a Task?

- Tasks have been fully integrated into OpenMP
- Note: OpenMP has always had tasks but they were never called that way before the 3.0 release!
  - Thread encountering `parallel` construct packages up a set of implicit tasks, one per thread
  - Team of threads is created
  - Each thread in team is assigned to one of the tasks (and tied to it)
  - Barrier holds original master thread until all implicit tasks are finished
- Now we have a way to create a task explicitly for the team to execute
What is a Task?

- A specific instance of executable code and its data environment, generated when a thread encounters a task construct or a parallel construct.
- Tasks consist of
  - Code to execute
  - Data environment
  - Internal control variables (new from 2.5)
- Each encountering thread creates a new task which packages its own code and data.
- Execution of the new task could be immediate, or deferred until later.
- Can be nested into
  - Another task or a work sharing construct.
Tasks: Usage

Task Construct:

```c
#pragma omp task [clause[],clause] ...] structured-block
```

where clause can be

- Data scoping clauses
  - `shared` (list), `private` (list), `firstprivate` (list), `default` (shared | none)
- Scheduling clauses
  - `untied`
- Other clauses
  - `if` (expression)
Loop Nesting

While OpenMP 3.0 supports nested parallelism, many implementations may ignore the nesting by serializing the inner parallel regions.
References

- http://www.openmp.org/

- *Parallel Programming in OpenMP*, by Chandra, Dagum, Kohr, Maydan, McDonald, Menon

- *Using OpenMP*, by Chapman, Jost, Van der Pas (OpenMP2.5)


FOR FORTRAN USERS
Default variable scoping (Fortran example)

Program Main
Integer, Parameter :: nmax=100
Integer :: n, j
Real*8 :: x(n,n)
Common /vars/ y(nmax)
...

n=nmax; y=0.0
!$OMP Parallel do
  do j=1,n
    call Adder(x,n,j)
  end do
...
End Program Main

Subroutine Adder(a,m,col)
Common /vars/ y(nmax)
SAVE array_sum
Integer :: i, m
Real*8 :: a(m,m)

do i=1,m
  y(col)=y(col)+a(i,col)
end do
array_sum=array_sum+y(col)

End Subroutine Adder
## Default data scoping in Fortran (cont.)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Scope</th>
<th>Is use safe?</th>
<th>Reason for scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>shared</td>
<td>yes</td>
<td>declared outside parallel construct</td>
</tr>
<tr>
<td>j</td>
<td>private</td>
<td>yes</td>
<td>parallel loop index variable</td>
</tr>
<tr>
<td>x</td>
<td>shared</td>
<td>yes</td>
<td>declared outside parallel construct</td>
</tr>
<tr>
<td>y</td>
<td>shared</td>
<td>yes</td>
<td>common block</td>
</tr>
<tr>
<td>i</td>
<td>private</td>
<td>yes</td>
<td>parallel loop index variable</td>
</tr>
<tr>
<td>m</td>
<td>shared</td>
<td>yes</td>
<td>actual variable $n$ is shared</td>
</tr>
<tr>
<td>a</td>
<td>shared</td>
<td>yes</td>
<td>actual variable $x$ is shared</td>
</tr>
<tr>
<td>col</td>
<td>private</td>
<td>yes</td>
<td>actual variable $j$ is private</td>
</tr>
<tr>
<td>array_sum</td>
<td>shared</td>
<td>no</td>
<td>declared with SAVE attribute</td>
</tr>
</tbody>
</table>
Workshare directive

- **WORKSHARE** directive enables parallelization of Fortran 90 array expressions and FORALL constructs

```fortran
Integer, Parameter :: N=1000
Real*8 :: A(N,N), B(N,N), C(N,N)

!$OMP WORKSHARE
    A=B+C
!$OMP End WORKSHARE
```

- Enclosed code is separated into units of work
- All threads in a team share the work
- Each work unit is executed only once
- A work unit may be assigned to any thread
Reduction on array variables

- Supported in Fortran only!
- Array variables may now appear in the REDUCTION clause

```fortran
Real*8 :: A(N), B(M,N)
Integer :: i, j
A(1:m) = 3.
 !$OMP Parallel Do Reduction(+:A)
   do i=1,n
     A(1:m)=A(1:m)+B(1:m,i)
   end do
 !$OMP End Parallel Do
```

- Assumed size and allocatable arrays are not supported
- Variable must be shared in the enclosing context
Loop Collapse

• Allow collapsing of perfectly nested loops

• Will form a single loop and then parallelize it:

```
!$omp parallel do collapse(2)
do i=1,n
   do j=1,n
       ..... 
   end do
end do
```
Questions?