Outline:

- OpenMP:
  - Parallel blocks, critical sections, private and shared variables
  - Parallel do loops, reductions

Reading:

- class notes: OpenMP section of Bibliography
- $UWHPSC/codes/openmp
program test
    use omp_lib
    integer :: thread_num

    ! Specify number of threads to use:
    !$ call omp_set_num_threads(2)

    print *, "Testing openmp ..."

    !$omp parallel
    !$omp critical
    !$ thread_num = omp_get_thread_num()
    !$ print *, "This thread = ",thread_num
    !$omp end critical
    !$omp end parallel
end program test
Compiled with OpenMP:

$ gfortran -fopenmp test.f90
$ ./a.out

Testing openmp ...  
This thread = 0  
This thread = 1  
(or threads might print in the other order!)

Compiled without OpenMP:

$ gfortran test.f90
$ ./a.out

Testing openmp ...
OpenMP test code

```c
!$omp parallel
!$omp critical
!$ thread_num = omp_get_thread_num()
!$ print *, "This thread = ", thread_num
!$omp end critical
!$omp end parallel
```

The !$omp parallel block spawns two threads and each one works independently, doing all instructions in block.

Threads are destroyed at !$omp end parallel.

However, the statements are also in a !$omp critical block, which indicates that this section of the code can be executed by only one thread at a time, so in fact they are not done in parallel.

So why do this? The function omp_get_thread_num() returns a unique number for each thread and we want to print both of these.
Incorrect code without critical section:

```fortran
!$omp parallel
!$ thread_num = omp_get_thread_num()
!$ print *, "This thread = ", thread_num
!$omp end parallel
```

Why not do these in parallel?

1. If the prints are done simultaneously they may come out garbled (characters of one interspersed in the other).

2. `thread_num` is a shared variable. If this were not in a critical section, the following would be possible:

   Thread 0 executes function, sets `thread_num=0`
   Thread 1 executes function, sets `thread_num=1`
   Thread 0 executes print statement: "This thread = 1"
   Thread 1 executes print statement: "This thread = 1"

   There is a data race or race condition.
OpenMP test code

Could change to add a `private` clause:

```c
!$omp parallel private(thread_num)

!$ thread_num = omp_get_thread_num()

!$omp critical
!$ print *, "This thread = ", thread_num
!$omp end critical
!$omp end parallel
```

Then each thread has its own version of the `thread_num` variable.
OpenMP parallel do loops

```c
 !$omp parallel do
do i=1,n
  ! do stuff for each i
  enddo
$omp end parallel do ! OPTIONAL
```

indicates that the do loop can be done in parallel.

**Requires:**
- what’s done for each value of $i$ is independent of others
- Different values of $i$ can be done in any order.

The iteration variable $i$ is **private** to the thread: each thread has its own version.

By default, all other variables are **shared** between threads unless specified otherwise.
OpenMP parallel do loops

This code fills a vector \( y \) with function values that take a bit of time to compute:

\[
\text{! fragment of } \$UWHPSC/codes/openmp/yeval.f90
\]

\[
dx = 1.\text{d}0 / (n+1.\text{d}0)
\]

\[
\text{!}\$omp parallel do private(x)
do i=1,n 
  \hspace{1em} x = i*dx 
  \hspace{1em} y(i) = \exp(x) \times \cos(x) \times \sin(x) \times \sqrt{5x+6.\text{d}0}
enddo
\]

Elapsed time for \( n = 10^8 \), without OpenMP: about 9.3 sec.

Elapsed time using OpenMP on 2 processors: about 5.0 sec.
Memory stack

Note: Parallel threads use stack and you may need to increase the limit (e.g. on the VM):

```bash
$ gfortran -fopenmp yeval.f90
$ ./a.out
Segmentation fault

$ ulimit -s
8192

$ ulimit -s unlimited

$ ./a.out
Using OpenMP with 2 threads
Filled vector y of length 100000000
Memory stack

Note: Parallel threads use stack and you may need to increase the limit (e.g. on the VM):

$ gfortran -fopenmp yeval.f90
$ ./a.out
Segmentation fault

$ ulimit -s
8192

$ ulimit -s unlimited

$ ./a.out
Using OpenMP with 2 threads
Filled vector y of length 100000000

On Mac, there's a hard limit `ulimit -s hard`
Memory devoted to data for a program is generally split up:

**Heap**: Dynamically allocated memory — memory allocator looks for free block of memory, keeps track of free list, does garbage collection, etc.

**Stack**: Block of memory where space is allocated on “top” of the stack as needed and “popped” off the stack when no longer needed. Last in – first out (LIFO).

Fast relative to heap allocation.

Natural way to allocate storage for nested subroutine or function calls: If A calls B calls C, then when the variables used by C are popped off the stack, we’re back to the variables of B.
Memory devoted to data for a program is generally split up:

**Heap:** Dynamically allocated memory — memory allocator looks for free block of memory, keeps track of free list, does garbage collection, etc.

**Stack:** Block of memory where space is allocated on “top” of the stack as needed and “popped” off the stack when no longer needed. Last in – first out (LIFO).

Fast relative to heap allocation.

Natural way to allocate storage for nested subroutine or function calls: If A calls B calls C, then when the variables used by C are popped off the stack, we’re back to the variables of B.

**Private variables for threads** also put on stack, popped off when parallel block ends.
This code is not correct:

```c
!$omp parallel do
do i=1,n
    x = i*dx
    y(i) = exp(x)*cos(x)*sin(x)*sqrt(5*x+6.d0)
enddo
```

By default, \( x \) is a shared variable. Might happen that:

- Processor 0 sets \( x \) properly for one value of \( i \),
- Processor 1 sets \( x \) properly for another value of \( i \),
- Processor 0 uses \( x \) but is now incorrect.
This code is not correct:

```plaintext
!$omp parallel do
do i=1,n
   x = i*dx
   y(i) = exp(x)*cos(x)*sin(x)*sqrt(5*x+6.d0)
enddo
```

By default, \( x \) is a shared variable.

Might happen that:
- Processor 0 sets \( x \) properly for one value of \( i \),
- Processor 1 sets \( x \) properly for another value of \( i \),
- Processor 0 uses \( x \) but is now incorrect.
Correct version:

```c
!$omp parallel do private(x)
do i=1,n
  x = i*dx
  y(i) = exp(x)*cos(x)*sin(x)*sqrt(5*x+6.d0)
enddo
```

Now each thread has its own version of $x$.

Iteration counter $i$ is private by default.
Correct version:

```c
!$omp parallel do private(x)
do i=1,n
    x = i*dx
    y(i) = exp(x)*cos(x)*sin(x)*sqrt(5*x+6.d0)
enddo
```

Now each thread has its own version of \( x \).

Iteration counter \( i \) is private by default.

Note that \( dx, n, y \) are shared by default. **OK because:**

- \( dx, n \) are used but not changed,
- \( y \) is changed, but independently for each \( i \).
OpenMP parallel do loops

Incorrect code:

\[
\begin{align*}
  dx &= 1.d0 / (n+1.d0) \\
  !$omp parallel do private(x,dx) \\
  do i=1,n \\
    x &= i*dx \\
    y(i) &= \exp(x) \times \cos(x) \times \sin(x) \times \sqrt{5x+6.d0} \\
  enddo 
\end{align*}
\]

Specifying \(dx\) private won’t work here.

This will create a private variable \(dx\) for each thread but it will be uninitialized.

Will run but give garbage.
OpenMP parallel do loops

Could fix with:

```fortran
dx = 1.d0 / (n+1.d0)
!$omp parallel do firstprivate(dx)
do i=1,n
  x = i*dx
  y(i) = exp(x)*cos(x)*sin(x)*sqrt(5*x+6.d0)
enddo
```

The `firstprivate` clause creates private variables and initializes to the value from the master thread prior to the loop.
OpenMP parallel do loops

Could fix with:

```
  dx = 1.d0 / (n+1.d0)
  !$omp parallel do firstprivate(dx)
  do i=1,n
    x = i*dx
    y(i) = exp(x) * cos(x) * sin(x) * sqrt(5*x + 6.d0)
  enddo
```

The `firstprivate` clause creates private variables and initializes to the value from the master thread prior to the loop.

There is also a `lastprivate` clause to indicate that the last value computed by a thread (for \( i = n \)) should be copied to the master thread’s copy for continued execution.
OpenMP parallel do loops

! from $UWHPSC/codes/openmp/private1.f90

n = 7
y = 2.0
!$omp parallel do firstprivate(y) lastprivate(y)
do i=1,n
 y = y + 10.0
 x(i) = y
 !omp critical
      print *, "i = ",i," x(i) = ",x(i)
 !omp end critical
enddo
print *, "At end, y = ",y

Run with 2 threads: The 7 values of i will be split up, perhaps

i = 1, 2, 3, 4 executed by thread 0,
i = 5, 6, 7 executed by thread 1.

Thread 0’s private y will be updated 4 times, 2 → 12 → 22 → 32 → 42

Thread 1’s private y will be updated 3 times, 2 → 12 → 22 → 32
OpenMP parallel do loops

! from $UWHPSC/codes/openmp/private1.f90

n = 7
y = 2.d0
!$omp parallel do firstprivate(y) lastprivate(y)
do i=1,n
  y = y + 10.d0
  x(i) = y
!omp critical
  print *, "i = ",i," x(i) = ",x(i)
!omp end critical
enddo
print *, "At end, y = ",y

might produce:

i = 1       x(i) =  12.00000000000000000000
i = 5       x(i) =  12.00000000000000000000
i = 2       x(i) =  22.00000000000000000000
i = 6       x(i) =  22.00000000000000000000
i = 3       x(i) =  32.00000000000000000000
i = 7       x(i) =  32.00000000000000000000
i = 4       x(i) =  42.00000000000000000000
At end, y = 32.00000000000000000000

Order might be different but final y will be from \(i = 7\).
Default is that loop iterator is private, other variables shared. Can change this, e.g.

```
!$omp parallel do default(private) shared(x,z) &
!$omp firstprivate(y) lastprivate(y)
do i=1,n
  etc.
```

With this change, only $x$ and $z$ are shared.

Note continuation character `&` and continuation line.
OpenMP synchronization

```c
!$omp parallel do
do i=1,n
    ! do stuff for each i
enddo
!$omp end parallel do  ! OPTIONAL

! master thread continues execution
```

There is an **implicit barrier** at the end of the loop.

The master thread will not continue until all threads have finished with their subset of \(1, 2, \ldots, n\).

Except if ended by:

```c
!$omp end parallel do nowait
```
Conditional clause

Loop overhead may not be worthwhile for short loops.
(Multi-thread version may run slower than sequential)

Can use conditional clause:

```plaintext
$omp parallel do if (n > 1000)
do i=1,n
  ! do stuff
enddo
```

If $n \leq 1000$ then no threads are created, master thread executes loop sequentially.
Nested loops

```fortran
!$omp parallel do private(i)
do  j=1,m
    do  i=1,n
        a(i,j) = 0.d0
    enddo
enddo
```

The loop on \( j \) is split up between threads.

The thread handling \( j=1 \) does the entire loop on \( i \), sets \( a(1,1), a(2,1), \ldots, a(n,1) \).
Nested loops

```
!$omp parallel do private(i)
  do j=1,m
    do i=1,n
      a(i,j) = 0.d0
    enddo
  enddo
enddo
```

The loop on \( j \) is split up between threads.

The thread handling \( j=1 \) does the entire loop on \( i \), sets \( a(1,1), a(2,1), \ldots, a(n,1) \).

**Note:** The loop iterator \( i \) must be declared **private**!

\( j \) is private by default, \( i \) is shared by default.
Nested loops

Which is better? (assume $m \approx n$)

```fortran
!$omp parallel do private(i)
do j=1,m
   do i=1,n
      a(i,j) = 0.d0
   enddo
enddo
```

or

```fortran
do j=1,m
   !$omp parallel do
      do i=1,n
         a(i,j) = 0.d0
      enddo
   enddo
enddo
```

The first has less overhead: Threads created only once.
The second has more overhead: Threads created $m$ times.

R.J. LeVeque, University of Washington
AMath 483/583, Lecture 14
Nested loops

Which is better?  (assume $m \approx n$)

```fortran
!$omp parallel do private(i)
do  j=1,m
   do  i=1,n
      a(i,j) = 0.d0
   enddo
enddo
```

or

```fortran
do  j=1,m
   !$omp parallel do
   do  i=1,n
      a(i,j) = 0.d0
   enddo
enddo
```

The first has less overhead: Threads created only once.
The second has more overhead: Threads created $m$ times.
Nested loops

But have to make sure loop can be parallelized!

Incorrect code for replicating first column:

```fortran
!$omp parallel do private(j)
do i=2,n
  do j=1,m
    a(i,j) = a(i-1,j)
  enddo
enddo
```

Corrected: (j’s can be done in any order, i’s cannot)

```fortran
!$omp parallel do private(i)
do j=1,m
  do i=2,n
    a(i,j) = a(i-1,j)
  enddo
enddo
```
Reductions

Incorrect code for computing $\|x\|_1 = \sum_i |x_i|$

```fortran
norm = 0.d0
!$omp parallel do
do i=1,n
    norm = norm + abs(x(i))
enddo
```

There is a race condition: each thread is updating same shared variable `norm`.

Correct code:

```fortran
!$omp parallel do reduction(+ : norm)
do i=1,n
    norm = norm + abs(x(i))
enddo
```

A reduction reduces an array of numbers to a single value.
A more complicated way to do this:

```fortran
norm = 0.d0
!$omp parallel private(mysum) shared(norm)
mysum = 0
!$omp do
do i=1,n
   mysum = mysum + abs(x(i))
  enddo

!$omp critical
norm = norm + mysum
!$omp end critical
!$omp end parallel
```
Some other reductions

Can do reductions using $+, −, \ast, \min, \max, \cdot \text{and}, \cdot \text{or}, \text{some others}$

General form:

```
$\texttt{!omp parallel do reduction(operator : list)}$
```

Example with max:

```
y = -1.d300 ! very negative value
$\texttt{!omp parallel do reduction(max: y)}$
do i=1,n
   y = max(y,x(i))
enddo
print *, 'max of x = ',y
```
Some other reductions

General form:

```$omp parallel do reduction(operator : list)```

Example with `.or.`:

```logical anyzero

! set x...
anyzero = .false.

!$omp parallel do reduction(.or.: anyzero)
do i=1,n
    anyzero = anyzero .or. (x(i) == 0.d0)
enddo
print *, 'anyzero = ', anyzero```

Prints `T` if any `x(i)` is zero, `F` otherwise.