Outline:

• OpenMP:
  • loop dependencies
  • threadsafe and pure subroutines and functions
  • other directives, beyond "parallel do"

Reading:

• class notes: OpenMP section of Bibliography
• $UWHPSC/codes/openmp
Fernando Perez from Berkeley will talk about

IPython for parallel computing

eScience Seminar on IPython more generally,

IPython: tools for the lifecycle of computational ideas

At 4:00pm on May 13 in EE 303.
Outline:

- OpenMP:
  - loop dependencies
  - threadsafe and pure subroutines and functions
  - other directives, beyond "parallel do"

Reading:

- class notes: OpenMP section of Bibliography
- $UWHPSC/codes/openmp
Dependencies in loops

\[
\begin{align*}
\text{do } & i=1, n \\
& z(i) = x(i) + y(i) \\
& w(i) = \cos(z(i)) \\
\text{enddo}
\end{align*}
\]

There is a data dependence between the two statements in this loop.

The value \( w(i) \) cannot be computed before \( z(i) \).

However, this \textit{can} be paralellized with a parallel do since the same thread will always execute both statements in the right order for each \( i \).
Matrix-matrix multiplication

!$omp parallel do private(i,k)
do j=1,n
do i=1,n
c(i,j) = 0.d0
do k=1,n
c(i,j) = c(i,j) + a(i,k)*b(k,j)
enddo
doto
Loop-Carried Dependencies

\[ x = 1.0 \text{d}0 \quad ! \text{initialize all elements to 1} \]
\[ x(1) = 5.0 \text{d}0 \]

\[ \text{do } i=2,n \]
\[ \quad x(i) = x(i-1) \]
\[ \quad \text{enddo} \]

There is a loop-carried data dependence in this loop.

The assignment for \( i=3 \) must not be done before \( i=2 \) or it may get the wrong value.
Example: Solve ODE initial value problem

\[ y'(t) = 2y(t), \]
\[ y(0) = 1 \]

with Euler’s method

\[ y(t + \Delta t) \approx y(t) + \Delta t y'(t) = y(t) + \Delta t(2y(t)), \]

to approximate \( y(t) = e^{2t} \) for \( 0 \leq t \leq 5 \):

\[
\begin{align*}
y(1) &= 1.d0 \\
dt &= 0.001d0 \quad ! \text{time step} \\
n &= 5000 \quad ! \text{number of steps to reach } t=5 \\
do \ i=2,n \\
\quad y(i) &= y(i-1) + dt*2.d0*y(i-1) \\
enddo
\end{align*}
\]

Cannot easily parallelize.
y = 0.d0
do i=1,10
  if (i==3) y = 1.d0
  x(i) = y
enddo

There is a loop-carried data dependence in this loop.

In serial execution: only first two elements of x are 0.d0.

With !$omp parallel do:

  later index (e.g. i=6) might be executed before i=3.
Thread-safe functions

Consider this code:

```fortran
!$omp parallel do
  do i=1,n
    y(i) = myfcn(x(i))
  enddo
```

Does this give the same results as the serial version?
Consider this code:

```c
 !$omp parallel do
do i=1,n
    y(i) = myfcn(x(i))
enddo
```

Does this give the same results as the serial version?

Maybe not... it depends on what the function does!

If this gives the same results regards of the order threads call for different values of \(i\), then the function is thread safe.
Thread-safe functions

A thread-safe function:

```fortran
function myfcn(x)
  real(kind=8), intent(in) :: x
  real(kind=8), intent(out) :: myfcn
  real(kind=8) :: z ! local variable
  z = exp(x)
  myfcn = z*cos(x)
end function myfcn
```

Executing this function for one value of $x$ is completely independent of execution for other values of $x$.

Note that each call creates a new local value $z$ on the call stack, so $z$ is private to the thread executing the function.
Non-Thread-safe functions

Suppose $z$, $\text{count}$ are global variables defined in module 
globals.f90.

Then this function is **not thread-safe:**

```fortran
function myfcn(x)
    real(kind=8), intent(in) :: x
    real(kind=8), intent(out) :: myfcn
    use globals
    count = count+1 ! counts times called
    z = exp(x)
    myfcn = z*cos(x) + count
end function myfcn
```

The value of $\text{count}$ seen when calling $y(i) = \text{myfcn}(x(i))$
will depend on the order of execution of different values of $i$.

Moreover, $z$ might be modified by another thread between when
it is computed and when it is used.
Aside on global variables in Fortran

```fortran
module globals
  implicit none
  save
  integer :: count
  real(kind=8) :: z
end module globals
```

The `save` command says that values of these variables should be saved from one use to the next.

**Fortran 77 and before:** Instead used `common` blocks:

```fortran
common /globals/ z, count
```

can be included in any file where `z` and `count` should be available. *(Also not thread safe!)*
Non-Thread-safe functions

Beware of input or output...

Suppose unit 20 has been opened for reading in the main program, value on line \( i \) should be used in calculating \( y(i) \) ...

This function is not thread-safe:

```fortran
function myfcn(x)
    real(kind=8), intent(in) :: x
    real(kind=8), intent(out) :: myfcn
    real(kind=8) :: z

    read(20,*) z
    myfcn = z*cos(x)
end function myfcn
```

Will work in serial mode but if threads execute in different order, will give wrong results.
A subroutine can be declared pure if it:
- Does not alter global variables,
- Does not do I/O,
- Does not declare local variables with the save attribute, such as `real, save :: z`
- For functions, does not alter any input arguments.

Example:

```fortran
pure subroutine f(x,y)
  implicit none
  real(kind=8), intent(in) :: x
  real(kind=8), intent(inout) :: y
  y = x**2 + y
end subroutine f
```

**Good idea even for sequential codes:** Allows some compiler optimizations.
In place of

    do i=1,n
        x(i) = 2.d0*i
    end do

can write

    forall (i=1:n)
        x(i) = 2.d0*i
    end forall

Tells compiler that the statements can execute in any order.

Also may lead to compiler optimization even on serial computer.
Nested loops can be written with \texttt{forall}:

\begin{verbatim}
forall (i=1:n, j=1:n)
a(i,j) = 2.d0*i*j
end forall
\end{verbatim}

Tells compiler that it could reorder loops at will to optimize cache usage, for example.

Can also include \texttt{masks}:

\begin{verbatim}
forall (i=1:n, j=1:n, b(i,j).ne.0.d0)
a(i,j) = 1.d0 / b(i,j)
end forall
\end{verbatim}
OpenMP — beyond parallel loops

The directive !$omp parallel is used to create a number of threads that will each execute the same code...

    !$omp parallel
      ! some code
    !$omp end parallel

The code will be executed nthreads times, once by each thread.

**SPMD:** Single program, multiple data
The directive `!$omp parallel` is used to create a number of threads that will each execute the same code...

```
!$omp parallel
  ! some code
!$omp end parallel
```

The code will be executed \texttt{nthreads} times, once by each thread.

**SPMD:** Single program, multiple data

**Terminology note:**

**SIMD:** Single instruction, multiple data

refers to hardware (vector machines) that apply same arithmetic operation to a vector of values in lock-step. SPMD is a software term — need not be in lock step.
OpenMP parallel with do loops

Note: This code...

```c
!$omp parallel
  do i=1,10
    print *, "i = ",i
  enddo
!$omp end parallel
```

... is not the same as:

```c
!$omp parallel do
  do i=1,10
    print *, "i = ",i
  enddo
!$omp end parallel do
```
OpenMP parallel with do loops

Note: This code...

```fortran
!$omp parallel
do i=1,10
    print *, "i = ", i
enddo
!$omp end parallel
```

The entire do loop (i=1,2,...,10) will be executed by each thread!
With 2 threads, 20 lines will be printed.

... is not the same as:

```fortran
!$omp parallel do
do i=1,10
    do i=1,10
        print *, "i = ", i
    enddo
!$omp end parallel do
```

which will only print 10 lines!
OpenMP parallel with do loops

```c
!$omp parallel do
  do i=1,10
    print *, "i = ",i
  enddo
!$omp end parallel do

could also be written as:

!$omp parallel
  !$omp do
    do i=1,10
      print *, "i = ",i
    enddo
  !$omp end do
!$omp end parallel

More generally, if !$omp do is inside a parallel block, then the loop is split between threads rather than done in total by each
```
OpenMP parallel with do loops

The `!$omp do` directive is useful for...

```
!$omp parallel

! some code executed by every thread

!$omp do
do i=1,n
    ! loop to be split between threads
    enddo
!$omp end do

! more code executed by every thread

!$omp end parallel
```
Some other useful directives...

Execution of part of code by a single thread:

!$omp parallel
! some code executed by every thread

!$omp single
  ! code executed by only one thread
!$omp end single

!$omp end parallel

Can also use !$omp master to force execution by master thread.

**Example:** Initializing or printing out a shared variable.
Some other useful directives...

barriers:

```
$omp parallel
! some code executed by every thread

$omp barrier

! some code executed by every thread
$omp end parallel
```

Every thread will stop at barrier until all threads have reached this point.

Make sure all threads reach barrier or code will hang!
Some other useful directives...

barriers:

```c
$omp parallel
! some code executed by every thread
$omp barrier
! some code executed by every thread
$omp end parallel
```

Every thread will stop at barrier until all threads have reached this point.

Make sure all threads reach barrier or code will hang!

Implied barriers after some blocks, e.g.  

- `!$omp do`
- `!$omp single`
Some other useful directives...

Sections:

```c
!$omp parallel num_threads 2

!$omp sections

!$omp section
  ! code executed by only one thread

!$omp section
  ! code executed by a different thread

!$omp end sections  !! with implied barrier !!

!$omp end parallel
```

Example: Read in two large data files simultaneously.
integer, parameter :: n = 100000
real(kind=8), dimension(n) :: x, y, z

!$omp parallel     ! spawn two threads
!$omp sections     ! split up work between them

!$omp section
x = 1.d0           ! one thread initializes x array

!$omp section
y = 1.d0           ! another thread initializes y array

!$omp end sections
!$omp barrier      ! not needed, implied at end of sections

!$omp single       ! only want to print once:
print *, "Done initializing x and y"
!$omp end single nowait  ! ok for other thread to continue

!$omp do          ! split work between threads:
do i=1,n
   z(i) = x(i) + y(i)
enddo

!$omp end parallel