## AMath 483/583 — Lecture 16

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`



## Notes:



## Guest lecture on Monday May 13

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.



## Notes:



## AMath 483/583 — Lecture 16

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`

R.J. LeVeque, University of Washington AMath 483/583, Lecture 16

## Notes:

R.J. LeVeque, University of Washington AMath 483/583, Lecture 16

## Dependencies in loops

```
do i=1,n
    z(i) = x(i) + y(i)
    w(i) = cos(z(i))
    enddo
```

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

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

However, this *can* be paralellized with a parallel do since the same thread will always execute both statements in the right order for each $i$.



## Notes:



## 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
     enddo
enddo
```

This works since `c(i,j)` is only modified by thread handling column `j`.



## Notes:



## Loop-Carried Dependencies

```
x = 1.d0   ! initialize all elements to 1
x(1) = 5.d0

do i=2,n
    x(i) = x(i-1)
    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.



## Notes:

## Loop-Carried Dependencies

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 $0 \le t \le 5$:

```
y(1) = 1.d0
dt = 0.001d0     ! time step
n = 5000         ! number of steps to reach t=5
do i=2,n
    y(i) = y(i-1) + dt*2.d0*y(i-1)
    enddo
```

Cannot easily parallelize.



## Notes:



## Loop-Carried Dependencies

```
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`.



## Notes:



## Thread-safe functions

Consider this code:

```
!$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.



## Notes:

## Thread-safe functions

A thread-safe function:

```
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.



## Notes:



## Non-Thread-safe functions

Suppose `z, count` are global variables defined in module `globals.f90`.

Then this function is not thread-safe:

```
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 `count` seen when calling `y(i) = 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.



## Notes:



## Aside on global variables in 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:

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

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



## Notes:

## 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:

```
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.



## Notes:



## Pure subroutines and functions

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:

```
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.



## Notes:



## Forall statement in Fortran 90

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.



## Notes:

## Forall statement in Fortran 90

Nested loops can be written with `forall`:

```
forall (i=1:n, j=1:n)
    a(i,j) = 2.d0*i*j
end forall
```

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

Can also include masks:

```
forall (i=1:n, j=1:n, b(i,j).ne.0.d0)
    a(i,j) = 1.d0 / b(i,j)
end forall
```



## Notes:



## 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

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.



## Notes:



## OpenMP parallel with do loops

Note: This code...

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

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

which will only print 10 lines!



## Notes:

## OpenMP parallel with do loops

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



## Notes:



## 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
```



## Notes:



## 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.



## Notes:

## 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!

Implied barriers after some blocks, e.g. `!$omp do` or `!$omp single`.



## Notes:



## Some other useful directives...

Sections:

```
!$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.



## Notes:



## From `$UWHPSC/codes/openmp/demo2.f90`

```
8     integer, parameter :: n = 100000
9     real(kind=8), dimension(n) :: x,y,z

14    !$omp parallel  ! spawn two threads
15    !$omp sections  ! split up work between them
16
17      !$omp section
18      x = 1.d0   ! one thread initializes x array
19
20      !$omp section
21      y = 1.d0   ! another thread initializes y array
22
23    !$omp end sections
24    !$omp barrier   ! not needed, implied at end of sections
25
26    !$omp single    ! only want to print once:
27    print *, "Done initializing x and y"
28    !$omp end single nowait  ! ok for other thread to continue
29
30    !$omp do   ! split work between threads:
31    do i=1,n
32        z(i) = x(i) + y(i)
33        enddo
34
35    !$omp end parallel
```



## Notes: