AMath 483/583 — Lecture 15 — April 29, 2011

Today:

- OpenMP
- Fine grain vs. coarse grain parallelism

Next week:

- Iterative methods for linear systems
- Adaptive quadrature
- Start MPI

Read: Class notes and references

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 could 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
     enddo
enddo
```

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

Loop-Carried Dependencies

```
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.

Loop-Carried Dependencies

Example: Solve ODE

$$y'(t) = 2y(t),$$
$$y(0) = 1$$

with Euler's method $y(t + \Delta t) \approx y(t) + \Delta t y'(t)$:

```
y(1) = 1.d0
dt = ...     ! time step
do i=2,n
    y(i) = y(i-1) + dt*2.d0*y(i-1)
    enddo
```

Cannot easily parallelize.

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 `parallel do`, later index (e.g. `i=5`) may be executed before `i=3`.

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?

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.

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.

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.

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!)

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.

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.

Forall statement

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.

Forall statement

Nested loops can be written with `forall`:

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

Can 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
```

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.

SPMD: Single program, multiple data

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.

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...

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

OpenMP parallel with do loops

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

is shorthand for:

```
!$omp parallel
!$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...

Sections:

```
!$omp parallel

!$omp sections

  !$omp section
      ! code executed by only one thread

  !$omp section
      ! code executed by a different thread

!$omp end sections

!$omp end parallel
```

Example: Read in two large data files simultaneously.

Fine vs. coarse grain parallelism

Fine grain: Parallelize at the level of individual loops, splitting work for each loop between threads.

Coarse grain: Split problem up into large pieces and have each thread deal with one piece.

May need to synchronize or share information at some points.

Fine vs. coarse grain parallelism

Fine grain: Parallelize at the level of individual loops, splitting work for each loop between threads.

Coarse grain: Split problem up into large pieces and have each thread deal with one piece.

May need to synchronize or share information at some points.

Domain Decomposition: Splitting up a problem on a large domain (e.g. three-dimensional grid) into pieces that are handled separated (with suitable coupling).

Solution of independent ODEs by Euler's method

Solve $u_i'(t) = c_i u_i(t)$ for $t \geq 0$
with initial condition $u_i(0) = \eta_i$.

Solution of independent ODEs by Euler's method

Solve $u_i'(t) = c_i u_i(t)$ for $t \geq 0$
with initial condition $u_i(0) = \eta_i$.

Exact solution: $u_i(t) = e^{c_i t}\eta_i$.

Solution of independent ODEs by Euler's method

Solve $u_i'(t) = c_i u_i(t)$ for $t \geq 0$
with initial condition $u_i(0) = \eta_i$.

Exact solution: $u_i(t) = e^{c_i t}\eta_i$.

Euler method: $u_i(t + \Delta t) \approx u_i(t) + \Delta t c_i u_i(t) = (1 + c_i \Delta t) u_i(t)$.

Implement this for large number of time steps for
$i = 1,\ 2,\ \ldots,\ n$ with n large too.

Solution of independent ODEs by Euler's method

Solve $u_i'(t) = c_i u_i(t)$ for $t \geq 0$
with initial condition $u_i(0) = \eta_i$.

Exact solution: $u_i(t) = e^{c_i t}\eta_i$.

Euler method: $u_i(t + \Delta t) \approx u_i(t) + \Delta t c_i u_i(t) = (1 + c_i\Delta t)u_i(t)$.

Implement this for large number of time steps for
$i = 1,\ 2,\ \ldots,\ n$ with n large too.

This problem is embarassingly parallel: Problem for each i is completely decoupled from problem for any other i. Could solve them all simultaneously with no communication needed.

Fine grain solution with parallel do loops

```
!$omp parallel do
do i=1,n
    u(i) = eta(i)
    enddo

do m=1,nsteps
    !$omp parallel do
    do i=1,n
        u(i) = (1.d0 + dt*c(i))*u(i)
        enddo
    enddo
```

Note that threads are forked `nsteps+1` times.

Requires shared memory:
  don't know which thread will handle each `i`.

Coarse grain solution of ODEs

Split up $i = 1,\ 2,\ \ldots,\ n$ into `nthreads` disjoint sets.

A set goes from `i=istart` to `i=iend`

These private values are different for each thread.

Each thread handles 1 set for the entire problem.

```
!$omp parallel private(istart,iend,i,m)

istart = ??
iend = ??

do i=istart,iend
    u(i) = eta(i)
    enddo

do m=1,nsteps
    do i=istart,iend
        u(i) = (1.d0 + dt*c(i))*u(i)
        enddo
    enddo
!$omp end parallel
```

Threads are forked only once,
Each thread only needs subset of data.

Setting `istart` and `iend`

Example: If `n=100` and `nthreads = 2`, we would want:

Thread 0: `istart= 1` and `iend= 50`,
Thread 1: `istart=51` and `iend=100`.

If `nthreads` divides `n` evenly...

```
points_per_thread = n / nthreads

!$omp parallel private(thread_num, istart, iend, i)

    thread_num = 0      ! needed in serial mode
    !$ thread_num = omp_get_thread_num()

    istart = thread_num * points_per_thread + 1
    iend = (thread_num+1) * points_per_thread

    do i=istart,iend
        ! work on thread's part of array
        enddo
    ...

!$omp end parallel
```

Setting `istart` and `iend` more generally

Example: If `n=101` and `nthreads = 2`, we would want:

Thread 0: `istart= 1` and `iend= 51`,
Thread 1: `istart=52` and `iend=101`.

If `nthreads` might not divide `n` evenly...

```
points_per_thread = (n + nthreads - 1) / nthreads

!$omp parallel private(thread_num, istart, iend, i)

    thread_num = 0      ! needed in serial mode
    !$ thread_num = omp_get_thread_num()

    istart = thread_num * points_per_thread + 1
    iend = min((thread_num+1) * points_per_thread, n)

    do i=istart,iend
        ! work on thread's part of array
        enddo
    ...

!$omp end parallel
```