AMath 483/583 — Lecture 16 — May 2, 2011

Today:

- Fine grain vs. coarse grain parallelism
- Manually splitting do loops among threads

Wednesday:

- Adaptive quadrature, recursive functions
- Start MPI?

Read: Class notes and references

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
```

Example: Normalizing a vector

Given a vector (1-dimensional array) x,
Compute the normalized vector $x/\|x\|_1$, with $\|x\|_1 = \sum_{i=1}^n |x_i|$

Fine-grain: Using `parallel do` loops.

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

!$omp parallel do
do i=1,n
    x(i) = x(i) / norm
    enddo
```

Note: Must finish computing `norm` before using for any `x(i)`,
so we are using the implicit barrier after the first loop.

Example: Normalizing a vector

Another fine-grain approach, forking threads only once:

```
! from $CLASSHG/codes/openmp/normalize1.f90
    norm = 0.d0
    !$omp parallel private(i)

    !$omp do reduction(+ : norm)
    do i=1,n
        norm = norm + abs(x(i))
        enddo
    !$omp barrier  ! not needed (implicit)

    !$omp do
    do i=1,n
        x(i) = x(i) / norm
        enddo
    !$omp end parallel
```

Example: Normalizing a vector

Compute the normalized vector $x/\|x\|_1$, with $\|x\|_1 = \sum_{i=1}^n |x_i|$

Coarse grain version:

Assign blocks of i values to each thread. Threads must:

- Compute thread's contribution to $\|x\|_1$,

$$\texttt{norm\_thread} = \sum_{\texttt{istart}}^{\texttt{iend}} |x_i|,$$

- Collaborate to compute total value $\|x\|_1$:

$$\|x\|_1 = \sum_{\text{threads}} \texttt{norm\_thread}$$

- Loop over `i = istart, iend` to divide x_i by $\|x\|_1$.

Example: Normalizing a vector

```
! from $CLASSHG/codes/openmp/normalize2.f90

    norm = 0.d0
    !$omp parallel private(i,norm_thread, &
    !$omp                  istart,iend,thread_num)
    !$ thread_num = omp_get_thread_num()
    istart = thread_num * points_per_thread + 1
    iend = min((thread_num+1) * points_per_thread, n)

    norm_thread = 0.d0
    do i=istart,iend
        norm_thread = norm_thread + abs(x(i))
        enddo

    ! update global norm with value from each thread:
    !$omp critical
      norm = norm + norm_thread
    !$omp end critical

    !$omp barrier  !! needed here

    do i=istart,iend
        y(i) = x(i) / norm
        enddo

    !$omp end parallel
```

Normalizing a vector — possible bugs

1. Not declaring proper variables `private`

Normalizing a vector — possible bugs

1. Not declaring proper variables `private`

2. Setting `norm = 0.d0` inside parallel block.

Ok if it's in a `omp single` block. Otherwise second thread might set to zero after first thread has updated by `norm_thread`.

Normalizing a vector — possible bugs

1. Not declaring proper variables `private`

2. Setting `norm = 0.d0` inside parallel block.

Ok if it's in a `omp single` block. Otherwise second thread might set to zero after first thread has updated by `norm_thread`.

3. Not using `omp critical` block to update global `norm`.

Data race.

Normalizing a vector — possible bugs

1. Not declaring proper variables `private`

2. Setting `norm = 0.d0` inside parallel block.

Ok if it's in a `omp single` block. Otherwise second thread might set to zero after first thread has updated by `norm_thread`.

3. Not using `omp critical` block to update global `norm`.

Data race.

4. Not having a `barrier` between updating norm and using it.

First thread may use `norm` before other threads have added their contributions.

Normalizing a vector — possible bugs

1. Not declaring proper variables `private`

2. Setting `norm = 0.d0` inside parallel block.

Ok if it's in a `omp single` block. Otherwise second thread might set to zero after first thread has updated by `norm_thread`.

3. Not using `omp critical` block to update global `norm`.

Data race.

4. Not having a `barrier` between updating norm and using it.

First thread may use `norm` before other threads have added their contributions.

None of these bugs would give compile or run-time errors!
Just wrong results (sometimes).

OpenMP example with shared exit criterion

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

New wrinkle: Stop time stepping when any of the $u_i(t)$ values exceeds 100.

(Will certainly happen as long as $c_j > 0$ for some j.)

OpenMP example with shared exit criterion

Stop time stepping when any of the $u_i(t)$ values exceeds 100.

Idea:

Each time step, compute `umax =` maximum value of u_i over all i and exit the time-stepping if `umax > 100`.

OpenMP example with shared exit criterion

Stop time stepping when any of the $u_i(t)$ values exceeds 100.

Idea:

Each time step, compute `umax =` maximum value of u_i over all i and exit the time-stepping if `umax > 100`.

Each thread has a private variable `umax_thread` for the maximum value of u_i for its values of `i`. Updated for each `i`.

OpenMP example with shared exit criterion

Stop time stepping when any of the $u_i(t)$ values exceeds 100.

Idea:

Each time step, compute `umax =` maximum value of u_i over all i and exit the time-stepping if `umax > 100`.

Each thread has a private variable `umax_thread` for the maximum value of u_i for its values of `i`. Updated for each `i`.

Each thread updates shared `umax` based on its `umax_thread`.
This needs to be done in critical section.

OpenMP example with shared exit criterion

Stop time stepping when any of the $u_i(t)$ values exceeds 100.

Idea:

Each time step, compute `umax =` maximum value of u_i over all i and exit the time-stepping if `umax > 100`.

Each thread has a private variable `umax_thread` for the maximum value of u_i for its values of `i`. Updated for each `i`.

Each thread updates shared `umax` based on its `umax_thread`. This needs to be done in critical section.

Also need two barriers to make sure all threads are in synch at certain points.

OpenMP example with shared exit criterion

```
!$omp parallel private(i,m,umax_thread, &
!$omp                  istart,iend,thread_num)
!$ thread_num = omp_get_thread_num()
istart = thread_num * points_per_thread + 1
iend = min((thread_num+1) * points_per_thread, n)

do m=1,nsteps
    umax_thread = 0.d0
    !$omp single
      umax = 0.d0
    !$omp end single
    do i=istart,iend
        u(i) = (1.d0 + c(i)*dt) * u(i)
        umax_thread = max(umax_thread, u(i))
        enddo

    !$omp critical
      umax = max(umax, umax_thread)
    !$omp end critical
    !$omp barrier

    if (umax > 100) exit
    !$omp barrier
    enddo
!$omp end parallel
```

OpenMP example with shared exit criterion

If there were no barriers, following could happen:

Thread 0 executes critical section first, setting `umax` to 90.
Thread 0 checks if `umax > 100`. False, starts next iteration.
Thread 1 executes critical section, updating `umax` to 110.
Thread 1 checks if `umax > 100`. True, so it exits.

Thread 0 might never reach `umax > 100`. Runs forever.

OpenMP example with shared exit criterion

If there were no barriers, following could happen:

Thread 0 executes critical section first, setting `umax` to 90.
Thread 0 checks if `umax > 100`. False, starts next iteration.
Thread 1 executes critical section, updating `umax` to 110.
Thread 1 checks if `umax > 100`. True, so it exits.

Thread 0 might never reach `umax > 100`. Runs forever.

With only first barrier, following could happen:

`umax < 100` in iteration m.
Thread 0 checks if `umax > 100`. Go to iteration $m+1$.
Thread 0 does iteration on i and sets `umax > 100`,
Stops at first barrier.
Thread 1 (iteration m) checks if `umax > 100`. True, Exits.

Thread 1 never reaches first barrier again, code hangs.