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.

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 $u_i(0) = \eta_i$.

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 embarrassingly 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

```fortran
!$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 \(n\text{steps}+1\) times.

Requires shared memory:
- don't know which thread will handle each \(i\).

Notes:

Coarse grain solution of ODEs

Split up \(i=1,2,...,n\) into \(n\text{threads}\) disjoint sets.
A set goes from \(i=\text{istart}\) to \(i=\text{iend}\)
These private values are different for each thread.

Each thread handles 1 set for the entire problem.

```fortran
!$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
!$omp end parallel
```

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

Setting \(\text{istart}\) and \(\text{iend}\)

Example: If \(n=100\) and \(n\text{threads} = 2\), we would want:

- **Thread 0**: \(\text{istart}=1\) and \(\text{iend}=50\),
- **Thread 1**: \(\text{istart}=51\) and \(\text{iend}=100\).

If \(n\text{threads}\) divides \(n\) evenly...

points_per_thread = \(n / n\text{threads}\)

```fortran
!$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*

```fortran
!$omp parallel private(thread_num, istart, iend, i)
  thread_num = 0 ! needed in serial mode
  !$ 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.

```fortran
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
!$omp end parallel
```

**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:

```fortran
! 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
```

### Notes:
R.J. LeVeque, University of Washington  AMath 483/583, Lecture 16, May 2, 2011
Example: Normalizing a vector

Compute the normalized vector \( \frac{x}{\|x\|_1} \), 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 \( \|x\|_1 \),
  \[ \text{norm_thread} = \sum_{i=\text{istart}}^{\text{iend}} |x_i|, \]

- Collaborate to compute total value \( \|x\|_1 \):
  \[ \|x\|_1 = \sum_{\text{threads}} \text{norm_thread} \]

- Loop over \( i = \text{istart}, \text{iend} \) to divide \( x_i \) by \( \|x\|_1 \).

Example: Normalizing a vector

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

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

! loop over i = istart, iend to divide x_i by \|x\|_1
!$omp end parallel
```

Normalizing a vector — possible bugs

1. Not declaring proper variables private
2. Setting \( \text{norm} = 0.d0 \) 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 \( \text{norm_thread} \).
3. Not using omp critical block to update global \( \text{norm} \).
   Data race.
4. Not having a barrier between updating norm and using it.
   First thread may use \( \text{norm} \) 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 $u_{\text{max}}$ = maximum value of $u_i$ over all $i$ and exit the time-stepping if $u_{\text{max}} > 100$.

Each thread has a private variable $u_{\text{max\_thread}}$ for the maximum value of $u_i$ for its values of $i$; updated for each $i$.

Each thread updates shared $u_{\text{max}}$ based on its $u_{\text{max\_thread}}$.
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

```c
!$omp parallel private(i,m,umax_thread, &
!$omp istart,iend,thread_num)
!$omp thread_num = omp_get_thread_num()
iend = min((thread_num+1) * points_per_thread, n)
!$omp single
!$omp end single
do m=1,nsteps
   umax_thread = 0.d0
   !$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
```

Notes:
OpenMP example with shared exit criterion

If there were no barriers, following could happen:
Thread 0 executes critical section first, setting \( \text{umax} \) to 90.
Thread 0 checks if \( \text{umax} > 100 \). False, starts next iteration.
Thread 1 executes critical section, updating \( \text{umax} \) to 110.
Thread 1 checks if \( \text{umax} > 100 \). True, so it exits.
Thread 0 might never reach \( \text{umax} > 100 \). Runs forever.

With only first barrier, following could happen:
\( \text{umax} < 100 \) in iteration \( m \).
Thread 0 checks if \( \text{umax} > 100 \). Go to iteration \( m + 1 \).
Thread 0 does iteration \( i \) and sets \( \text{umax} > 100 \).
\( \text{Stop} \) at first barrier.
Thread 1 (iteration \( m \)) checks if \( \text{umax} > 100 \). True, \text{Exits}.
Thread 1 never reaches first barrier again, code hangs.