Outline:
- Fine grain vs. coarse grain parallelism
- Manually splitting loops between threads
- Examples with bugs

Reading:
- class notes: OpenMP section of Bibliography
- $UWHPSC/codes/openmp$
- https://computing.llnl.gov/tutorials/openMP/

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.

More similar to what must be done in MPI.

**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 \( u_i(0) = \eta_i \).
Decoupled system of ODEs for \( i = 1, 2, \ldots, n \)

**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 large \( n \).

For each \( i \) time stepping can’t be easily made parallel.

But for large \( n \), 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 \( n_{steps}+1 \) times.

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

Notes:

R.J. LeVeque, University of Washington
AMath 483/583, Lecture 17

Fine grain solution with parallel do loops

Might try to fork threads only once via:

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

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

Error: the loop on \( m \) will be done independently by each thread.
(Actually works in this case but not good coding.)

Notes:

R.J. LeVeque, University of Washington
AMath 483/583, Lecture 17

Fine grain solution with parallel do loops

Can rearrange loops:

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

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

Only works because ODEs are decoupled — can take all time steps on \( u_1(t) \) without interacting with \( u_2(t) \), for example.

Notes:

R.J. LeVeque, University of Washington
AMath 483/583, Lecture 17
**Coarse grain solution of ODEs**

Split up \( i = 1, 2, \ldots, n \) 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.

\[
\begin{align*}
!$omp & \text{ parallel private(istart,iend,i,m)} \\
\text{istart} &= ?? \\
\text{iend} &= ?? \\
do\ i = \text{istart}, \text{iend} & \\
\text{u}(i) &= \text{eta}(i) \\
\text{enddo} \\
do\ m = 1, \text{nsteps} & \\
do\ i = \text{istart}, \text{iend} & \\
\text{u}(i) &= (1.0 + \text{dt} \cdot \text{c}(i)) \cdot \text{u}(i) \\
\text{enddo} \\
!$omp & \text{ end parallel}
\end{align*}
\]

*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...*

\[
\begin{align*}
!$omp & \text{ parallel private(thread_num, istart, iend, i)} \\
\text{thread_num} &= 0 \quad \text{! needed in serial mode} \\
!$omp & \text{ end parallel}
\end{align*}
\]

---

**Setting \text{istart} and \text{iend} more generally**

*Example: If \( n = 101 \) and \( n_{\text{threads}} = 2 \), we would want:*

*Thread 0: \( \text{istart} = 1 \) and \( \text{iend} = 51 \),  
Thread 1: \( \text{istart} = 52 \) and \( \text{iend} = 101 \).*

*If \( n_{\text{threads}} \) might not divide \( n \) evenly...*

\[
\begin{align*}
!$omp & \text{ parallel private(thread_num, istart, iend, i)} \\
\text{thread_num} &= 0 \quad \text{! needed in serial mode} \\
!$omp & \text{ end parallel}
\end{align*}
\]
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))
endo

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

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 $UWHPSC/codes/openmp/normalize1.f90
norm = 0.d0
!$omp parallel private(i)
!$omp do reduction(+ : norm)
do i=1,n
    norm = norm + abs(x(i))
endo
!$omp barrier ! not needed (implicit)
!$omp do
do i=1,n
    x(i) = x(i) / norm
endo
!$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 $\|x\|_1$,
  
  $\text{norm\_thread} = \sum_{i=\text{istart}}^{i\text{end}} |x_i|$,  

- Collaborate to compute total value $\|x\|_1$:
  
  $\|x\|_1 = \sum_{\text{threads}}^{\text{norm\_thread}}$  

- Loop over $i = \text{istart}, i\text{end}$ to divide $x_i$ by $\|x\|_1$.
Example: Normalizing a vector

```fortran
! from $UWHPSC/codes/openmp/normalize2.f90
norm = 0.d0
%!omp parallel private(i,norm_thread, &
%!omp istart,iend,thread_num)
!$thread_num = omp_get_thread_num()
iistart = 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))
! update global norm with value from each thread:
!$omp critical
   norm = norm + norm_thread
!$omp end critical
!$omp barrier !! needed here
enddo
!
do i=istart,iend
   y(i) = x(i) / norm
!$omp end parallel
```

Notes:

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).

Example: Normalizing a vector — parallel block

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

Normalizing a vector — possible bugs

1. Not declaring proper variables private
2. Setting norm = 0.d0 inside parallel block.
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 $u'(t) = c_i u(t)$ 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 $j$.)

Notes:

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

Idea:

Each time step, compute $u_{\text{max}} = \text{maximum value of } u_i \text{ over all } i$ 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.

Study code in $UWHPSC/codes/openmp/umax1.f90$.

Notes:

!$omp parallel private(i,m,umax_thread, &
  istart,iend,thread_num)
!$omp thread_num = omp_get_thread_num()
!$omp istart = thread_num * points_per_thread + 1
!$omp iend = min((thread_num+1) * points_per_thread + 1
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
do loop in parallel block:

```fortran
   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
```

Notes:

OpenMP example with shared exit criterion

If there were no barriers, the following could happen:

- Thread 0 executes critical section first, setting umax to 0.5.
- 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 next sets umax to 0.4.

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

With only first barrier, the following could happen:

- umax < 100 in iteration m.
- Thread 1 checks if umax > 100. Go to iteration m + 1.
- Thread 1 does iteration on i and sets umax > 100, Stops at first barrier.
- Thread 0 (iteration m) checks if umax > 100. True, Exits.
- Thread 0 never reaches first barrier again, code hangs.