Math 483/583 — Lecture 16 — May 2, 2011	Notes:
Today:	
Fine grain vs. coarse grain parallelism	
<ul> <li>Manually splitting do loops among threads</li> </ul>	
Wednesday:	
Adaptive quadrature, recursive functions	
Start MPI?	
Read: Class notes and references	
Read: Class holes and relefences	
R.J. LeVeque, University of Washington AMath 483/583, Lecture 16, May 2, 2011	R.J. LeVeque, University of Washington AMath 483/583, Lecture 16, May 2, 2011
ine vs. coarse grain parallelism	Notes:
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).	
Halluleu separateu (with suitable couping).	
R.J. LeVeque, University of Washington AMath 483/583, Lecture 16, May 2, 2011	R.J. LeVeque, University of Washington AMath 483/583, Lecture 16, May 2, 2011
olution of independent ODEs by Euler's method	Notes:
Solve $u_i'(t) = c_i u_i(t)$ for $t \ge 0$	
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 $i = 1, 2,, n$ with $n$ large too.	
This problem is embarassingly parallel: Problem for each $i$ is	
completely decoupled from problem for any other <i>i</i> . Could solve them all simultaneously with no communication needed.	
R.J. LeVeque, University of Washington AMath 483/583, Lecture 16, May 2, 2011	R.J. LeVeque, University of Washington AMath 483/583, Lecture 16, May 2, 2011

Fine grain solution with parallel do loops	Notes:
<pre>!\$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.</pre>	
R.J. LeVeque, University of Washington AMath 483/583, Lecture 16, May 2, 2011	R.J. LeVeque, University of Washington AMath 483/583, Lecture 16, May 2, 2011
Coarse grain solution of ODEs	Notes:
<pre>Split up i = 1, 2,, 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 !\$omp end parallel Threads are forked only once, Each thread only needs subset of data.</pre>	R.J. LeVeque, University of Washington AMath 483/583, Lecture 16, May 2, 2011
Setting istart and iend	Notes:
<pre>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</pre>	
R.J. LeVeque, University of Washington AMath 483/583, Lecture 16, May 2, 2011	R.J. LeVeque, University of Washington AMath 483/583, Lecture 16, May 2, 2011
The Leveque, University of Washington Amid[11460/060, Lecture 10, May 2, 2011	The Leveque, University of Washington Awatin 405/505, Lecture To, May 2, 2011



R.J. LeVeque, University of Washington AMath 483/583, Lecture 16, May 2, 2011

# 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
```

R.J. LeVeque, University of Washington AMath 483/583, Lecture 16, May 2, 2011

### 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  $||x||_1$ .

R.J. LeVeque, University of Washington AMath 483/583, Lecture 16, May 2, 2011

## 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
```

R.J. LeVeque, University of Washington AMath 483/583, Lecture 16, May 2, 2011

### 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.

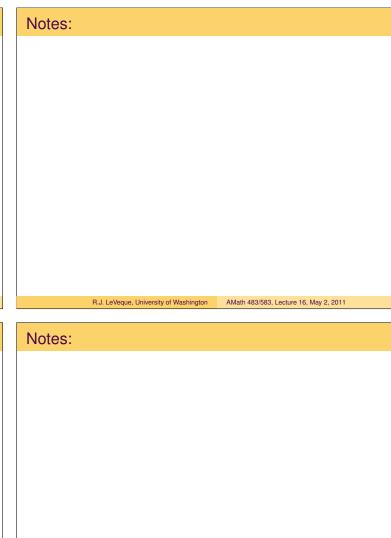
4. Not having a barrier between updating norm and using it.

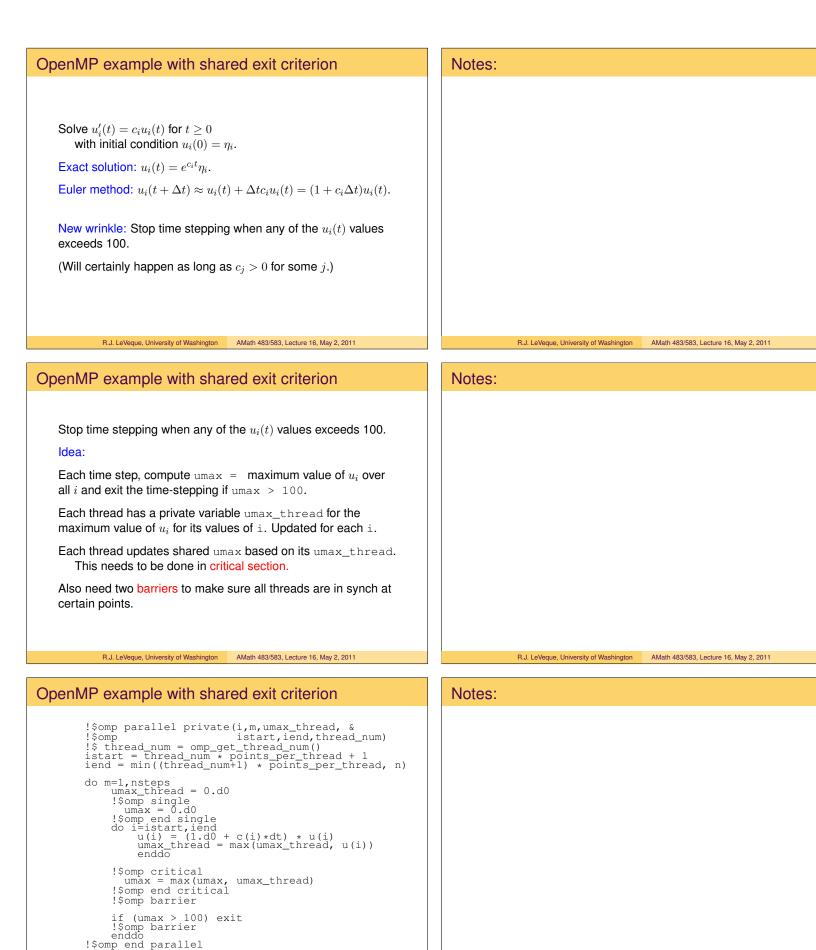
First thread may use norm before other threads have added their contributions.

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

# Notes:

### R.J. LeVeque, University of Washington AMath 483/583, Lecture 16, May 2, 2011





R.J. LeVeque, University of Washington AMath 483/583, Lecture 16, May 2, 2011

R.J. LeVeque, University of Washington AMath 483/583, Lecture 16, May 2, 2011

OpenMP example with shared exit criterion	Notes:
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.	
<pre>With only first barrier, following could happen: umax &lt; 100 in iteration m. Thread 0 checks if umax &gt; 100. Go to iteration m + 1. Thread 0 does iteration on i and sets umax &gt; 100, Stops at first barrier. Thread 1 (iteration m) checks if umax &gt; 100. True, Exits. Thread 1 never reaches first barrier again, code hangs.</pre>	
R.J. LeVeque, University of Washington AMath 483/583, Lecture 16, May 2, 2011	R.J. LeVeque, University of Washington AMath 483/583, Lecture 16, May 2, 2011