

CSE 160

Lecture 8

N-body methods
OpenMP

Announcements

- Quiz return today
- A1 due today
- A2 posted on Saturday
- Midterm on Thursday 2/14 at 5pm
Else we reschedule lecture on 3/1 to 5pm
- Thursday sections
 - We will only use Thursdays section for review sessions or at other times when we announce we need to use this time slot
 - The Friday meetings in APM generally serve as a section, but oriented toward programming labs

Corrections to last two slides of lecture 6

- Should swap pointers rather than copy values

Swap pointers: $I \leftrightarrow I^{\text{new}}$

Multithreaded Smoother()

Global Change, $I[:,:]$, $I^{\text{new}}[:,:]$

Local mymin = 1 + (\$TID * n / \$NT),
mymax = mymin + n / \$NT - 1;

Local done = FALSE;

while (!done) do

Local myChange = 0;

Change = 0;

update I^{new} and myChange

Change += myChange;

if (Change < Tolerance) done = TRUE;

Swap pointers: $I \leftrightarrow I^{\text{new}}$

end while

update I^{new} and myChange:
for i = mymin to mymax do
 for j = 1 to n do
 $I^{\text{new}}[i,j] = \dots$
 myChange += $(I^{\text{new}} - I^{\text{old}})^2$
 end for
end for

0
1
2
3

Is this code correct?

Correctness

Global Change, $I[:,:]$, $I^{\text{new}}[:,:]$

Local $\text{mymin} = 1 + (\$TID * n / \$nprocs)$,

$\text{mymax} = \text{mymin} + n / \$nprocs - 1$;

Local $\text{done} = \text{FALSE}$;

while (!done) do

Local $\text{myChange} = 0$;

BARRIER

Only on thread 0: $\text{Change} = 0$;

BARRIER

update I^{new} and myChange

CRITICAL SEC: $\text{Change} += \text{myChange}$

BARRIER

if ($\text{myChange} < \text{Tolerance}$) $\text{done} = \text{TRUE}$;

Only on thread 0: Swap pointers: $I \leftrightarrow I^{\text{new}}$

end while

0
1
2
3

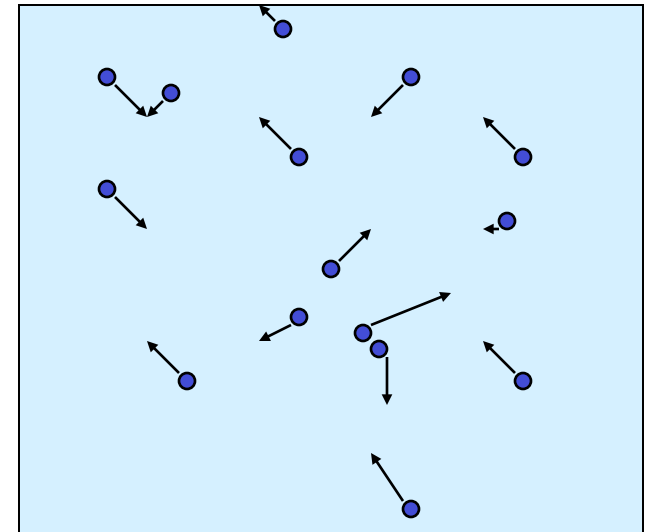
Does this code use minimal synchronization?

Today's lecture

- OpenMP
- N-body methods: A2

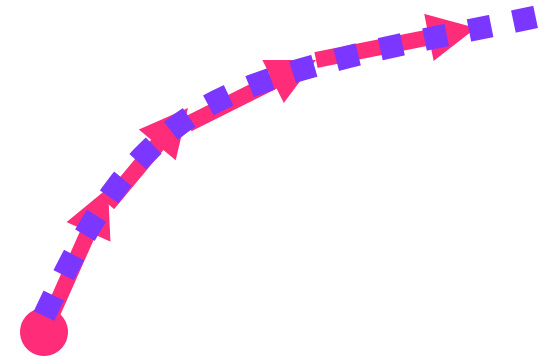
The N-body problem

- Compute trajectories of a system of N bodies often called *particles*, moving under mutual influence
 - The Greek word for particle: *somati'dion* = “little body”
 - No general analytic (exact) solution when $N > 2$
 - Numerical simulations required
 - N can range from thousands to millions
- A **force law** governs the way the particles interact
 - We may not need to perform all $O(N^2)$ force computations
 - Introduces non-uniformity due to uneven distributions



Discretization

- Particles move continuously through space and time according to a force, a continuous function of position and time: $F(\mathbf{x}, t)$
- We approximate continuous values using a discrete representation
- Evaluate force field at discrete points in time, called *timesteps* Δt , $2\Delta t$, $3\Delta t$
 $\Delta t = \text{discrete time step}$ (a parameter)
- “Push” the bodies according to Newton’s third law: $\mathbf{F} = m\mathbf{a} = m \, d\mathbf{u}/dt$
- There is no self induced force



```
while (current time < end time)
  forall bodies  $i \in 1:N$ 
    compute force  $F_i$  induced by all
      bodies  $j \in 1:N$ 
  //  $F = \text{mass} * \text{Acceleration}$ 
  forall bodies  $i \in 1:N$ 
    update velocity  $v_i$  by  $a_i \Delta t$ 
    update position  $x_i$  by  $v_i \Delta t$ 
  current time +=  $\Delta t$ 
end
```

Computing the force

- The running time of the computation is dominated by the force computation, which runs in time $O(N^2)$

Force on particle $i = \sum_{j=0:N-1} F(x_i, x_j)$

$F(\)$ is the **force law**

- In our assignment we'll use a simple repulsive force

if $\text{dist}(x_i, x_j) > 0.01 \Rightarrow F(x,y) = 0$

else

$$F(x,y) = C * (dx, dy)$$

Where $C = (0.01/r^2 - 1/r^3)$

$$r^2 = \max(dx^2 + dy^2, 10^{-6})$$

$$(dx, dy) = ((x_j - x_i), (y_j - y_i))$$

The code for the force

- OpenMP parallelization of the outer loop
- Use these as a specification for what to parallelize

```
void apply_forces( particle_t* particles, int n){  
#pragma omp parallel for shared(particles,n) schedule(dynamic)  
    for( int i = 0; i < n; i++ ) {  
        particles[i].ax = particles[i].ay = 0;  
        if ((particles[i].vx != 0) || (particles[i].vy != 0))  
            for (int j = 0; j < n; j++ ){  
                if (i==j)  
                    continue;  
                particles[i].{ax,ay} += coef * {dx,dy};  
            }  
    }  
}
```

Overall simulation

- Whatever is not inside an OMP parallel for should run on just one thread
- Synchronization?

```
void SimulateParticles() {  
    for( int step = 0; step < nsteps; step++ ) {  
        apply_forces(particles,n);  
  
        if (imbal)  
            imbal_particles(particles,n);  
  
        move_particles(particles,n);  
  
        VelNorms(particles,n,uMax,vMax,uL2,vL2);  
    }  
}
```

OpenMP

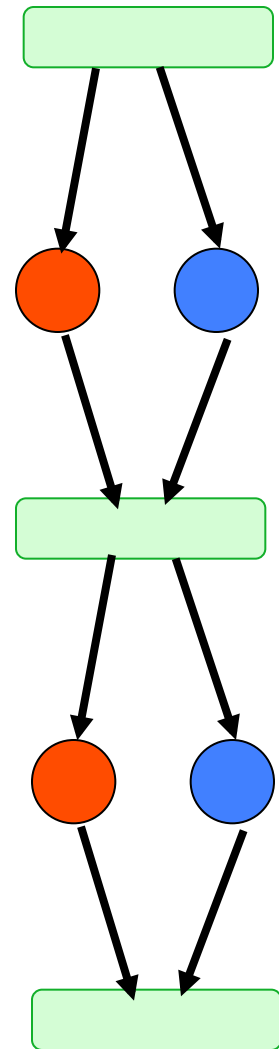
- A higher level interface for threads programming
- Parallelization handled via source code annotations
- See <http://www.openmp.org>
- Compare with explicit threads programming

```
#pragma omp parallel private(i)
                    shared(n)
{
    #pragma omp for
    for(i=0; i < n; i++)
        work(i);
}
```

```
i0 = $TID*n/$nthreads;
i1 = i0 + n/$nthreads;
for (i=i0; i< i1; i++)
    work(i);
```

OpenMP's Fork-Join Model

- A program begins life as a single thread
- Parallel regions spawn work groups of multiple threads
- The lexically enclosed program statements execute in parallel by all team members
- When we reach the end of the scope...
 - The team of threads synchronize at a barrier and are disbanded; they enter a wait state
 - Only the initial thread continues
- Thread teams can be created and disbanded many times during program execution, but this can be costly
- A clever compiler can avoid so many thread creations and joins



Fork join model with loops

Seung-Jai Min

```
cout << "Serial\n";  
N = 1000;
```

Serial

```
#pragma omp parallel for  
for (i=0; i<N; i++)  
    A[i] = B[i] + C[i];
```

Parallel

```
M = 500;
```

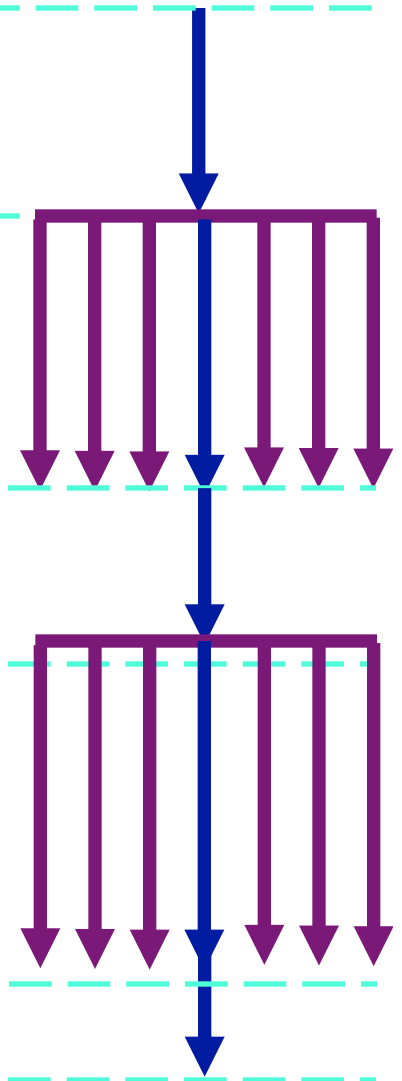
Serial

```
#pragma omp parallel for  
for (j=0; j<M; j++)  
    p[j] = q[j] - r[j];
```

Parallel

```
Cout << "Finish\n";
```

Serial



Workload decomposition

- Translator automatically generates appropriate local loop bounds
- Also inserts any needed barriers
- Decomposition can be static or dynamic
- Dynamic assignment for irregular problems (Monday)

```
#pragma omp parallel private(i) shared(n)
{
    #pragma omp for
    for(i=0; i < n; i++)
        work(i);
}
```

Parallelizing a nested loop with OpenMP

- We parallelize the outer loop index, indicated shared and private (local) variables
- Not all implementations can parallelize inner loops

```
#pragma omp parallel private(i) shared(n)
```

```
#pragma omp for
```

```
for(i=0; i < n; i++)
```

```
for(j=0; j < n; j++) {
```

```
     $u^{new}[i,j] = (u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i,j+1] - h^2$ 
```

```
    }
```

0
1
2
3

- Generated code

```
mymin = 1 + ($TID * n/nprocs),    mymax = mymin + n/nprocs - 1
```

```
for(i=mymin; i < mymax; i++)
```

```
for(j=0; j < n; j++)
```

```
     $u^{new}[i,j] = (u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i,j+1] - h^2 f[i,j])/4$ 
```

```
Barrier();
```

Your assignment

- Implement BLOCK partitioning
- Use the OpenMP annotations to guide parallelization
- You'll need to implement your own barrier; start with the mutex implementation as discussed in class; we'll look at more efficient barriers next week
- We'll discuss load balancing with dynamic scheduling on Monday
- Posted on Saturday morning by 11AM