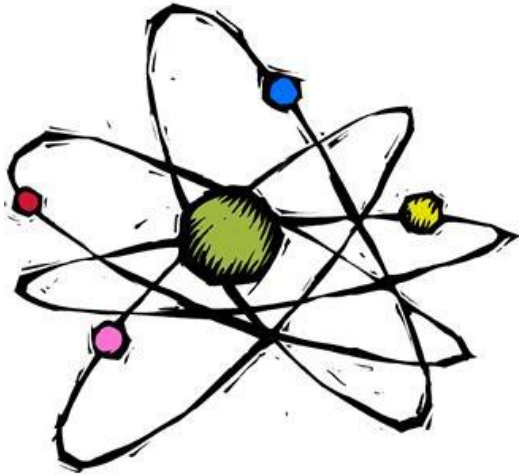
Parallel Programming

Parallel N-Body Solvers on the CPU

Adapted from Peter Pacheco’s textbook slides

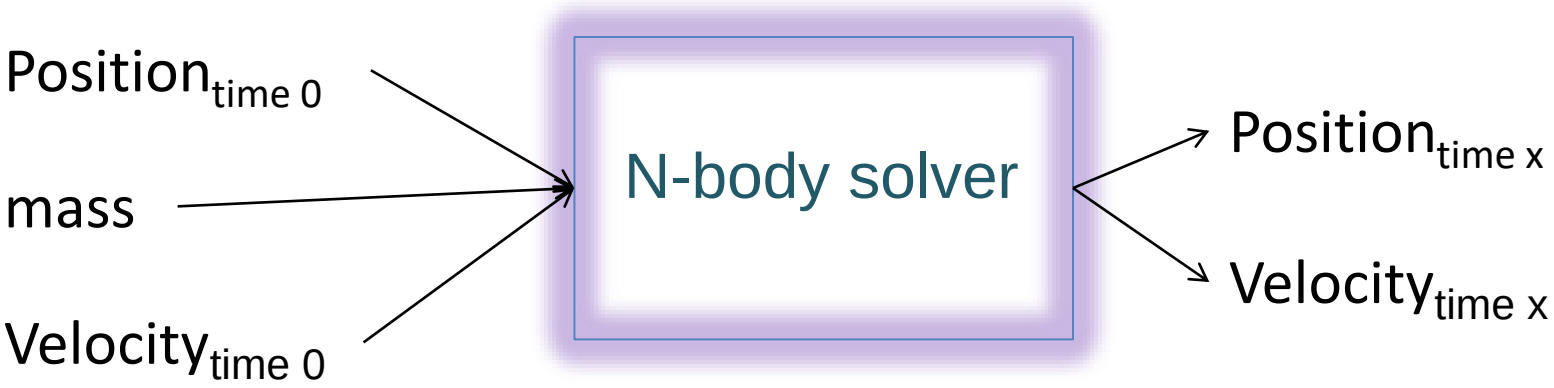


N-BODY SOLVERS

The n-body problem

- Find the positions and velocities of a collection of interacting particles over a period of time.

- An n-body solver is a program that finds the solution to an n-body problem by simulating the behavior of the particles.



Simulating motion of planets

- Determine the positions and velocities:
  - Newton's second law of motion.
  - Newton's law of universal gravitation.

Forces

$$\mathbf{f}_{qk}(t) = -\frac{Gm_q m_k}{\left|\mathbf{s}_q(t) - \mathbf{s}_k(t)\right|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_k(t)\right]$$

$$\mathbf{F}_q(t) = \sum_{\substack{k=0 \\ k\neq q}}^{n-1} \mathbf{f}_{qk} = -Gm_q \sum_{\substack{k=0 \\ k\neq q}}^{n-1} \frac{m_k}{\left|\mathbf{s}_q(t) - \mathbf{s}_k(t)\right|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_k(t)\right]$$

Acceleration

$$\mathbf{s}_q''(t) = -G \sum_{\substack{j=0 \\ j \neq q}}^{n-1} \frac{m_j}{\left|\mathbf{s}_q(t) - \mathbf{s}_j(t)\right|^3} \left[\mathbf{s}_q(t) - \mathbf{s}_j(t)\right]$$

$$t = 0, \Delta t, 2\Delta t, \ldots, T\Delta t$$

Serial pseudo-code

```
Get input data;
for each timestep {
   if (timestep output) Print positions and velocities of particles;
   for each particle q
      Compute total force on q;
   for each particle q
      Compute position and velocity of q;
}
Print positions and velocities of particles;
```

Computation of the forces

```
for each particle q {
   for each particle k != q {
      x_diff = pos[q][X] − pos[k][X];
      y_diff = pos[q][Y] − pos[k][Y];
      dist = sqrt(x_diff*x_diff + y_diff*y_diff);
      dist_cubed = dist*dist*dist;
      forces[q][X] −= G*masses[q]*masses[k]/dist_cubed * x_diff;
      forces[q][Y] −= G*masses[q]*masses[k]/dist_cubed * y_diff;
   }
}
```

A Reduced Algorithm for Computing N-Body Forces

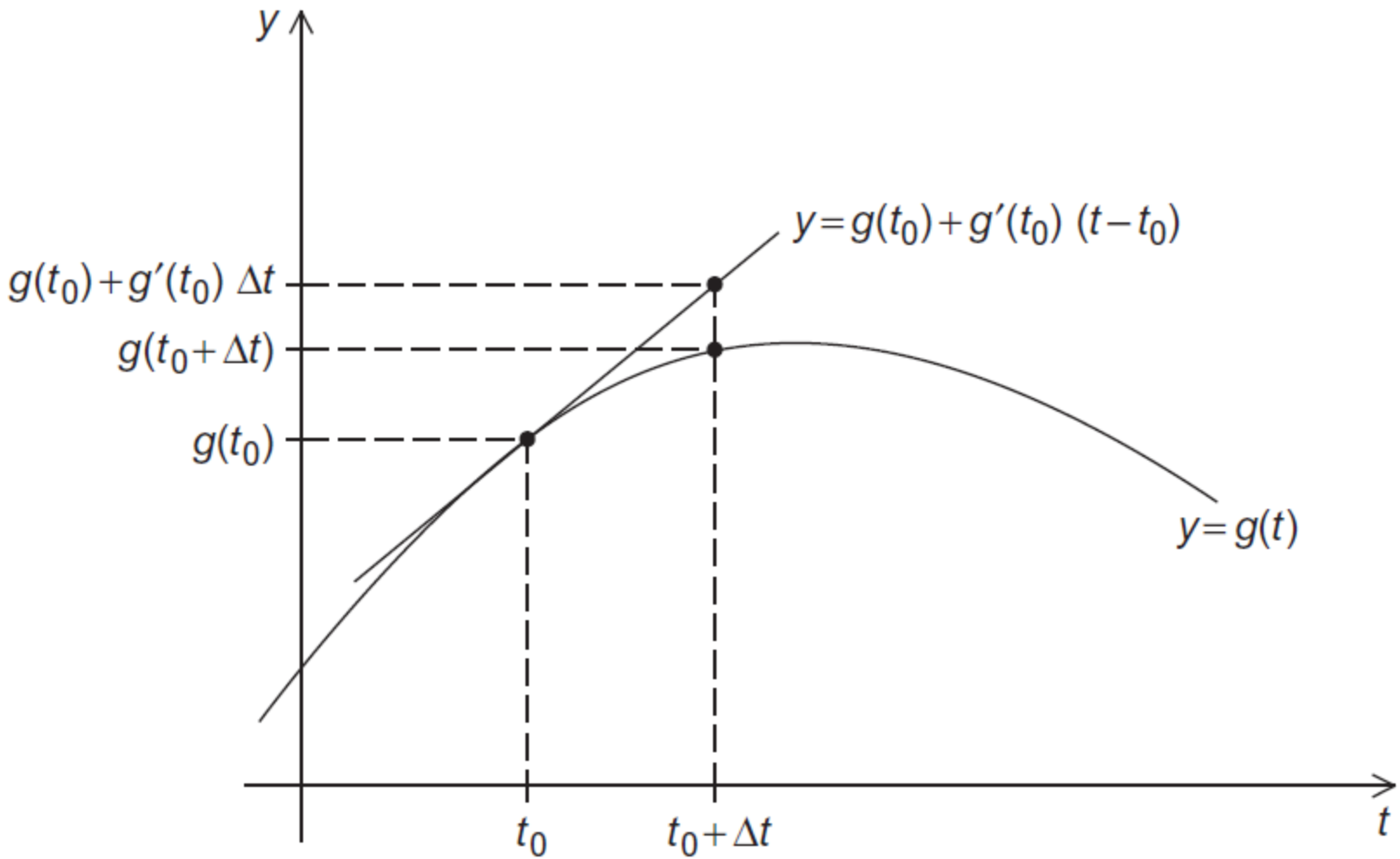
```
for each particle q
   forces[q] = 0;
for each particle q {
   for each particle k > q {
      x_diff = pos[q][X] − pos[k][X];
      y_diff = pos[q][Y] − pos[k][Y];
      dist = sqrt(x_diff∗x_diff + y_diff∗y_diff);
      dist_cubed = dist∗dist∗dist;
      force_qk[X] = G∗masses[q]∗masses[k]/dist_cubed ∗ x_diff;
      force_qk[Y] = G∗masses[q]∗masses[k]/dist_cubed ∗ y_diff

      forces[q][X] += force_qk[X];
      forces[q][Y] += force_qk[Y];
      forces[k][X] −= force_qk[X];
      forces[k][Y] −= force_qk[Y];
   }
}
```

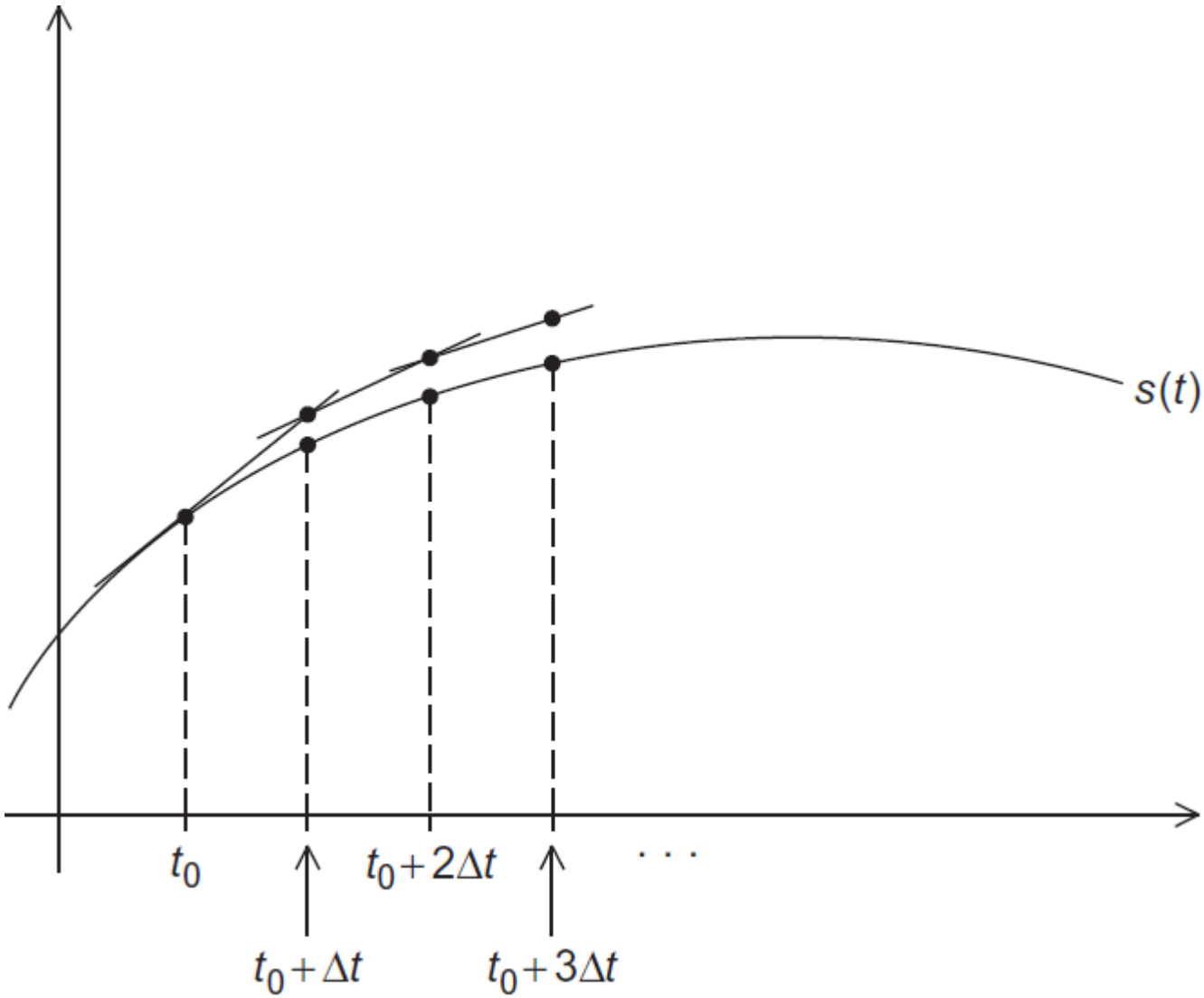
The individual forces

$$\begin{bmatrix} 0 & \mathbf{f}_{01} & \mathbf{f}_{02} & \cdots & \mathbf{f}_{0,n-1} \\ -\mathbf{f}_{01} & 0 & \mathbf{f}_{12} & \cdots & \mathbf{f}_{1,n-1} \\ -\mathbf{f}_{02} & -\mathbf{f}_{12} & 0 & \cdots & \mathbf{f}_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -\mathbf{f}_{0,n-1} & -\mathbf{f}_{1,n-1} & -\mathbf{f}_{2,n-1} & \cdots & 0 \end{bmatrix}$$

Using the Tangent Line to Approximate a Function



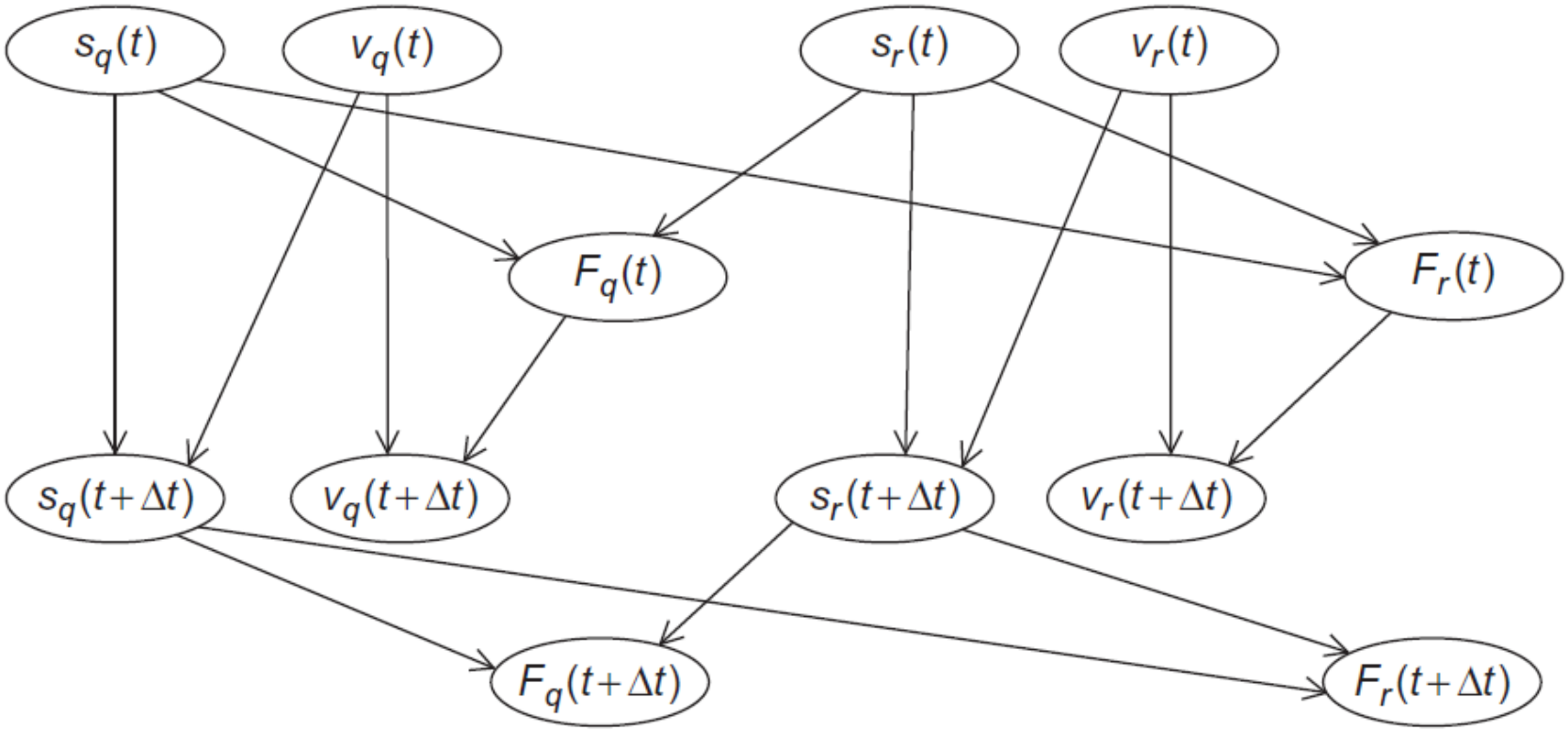
Euler's Method



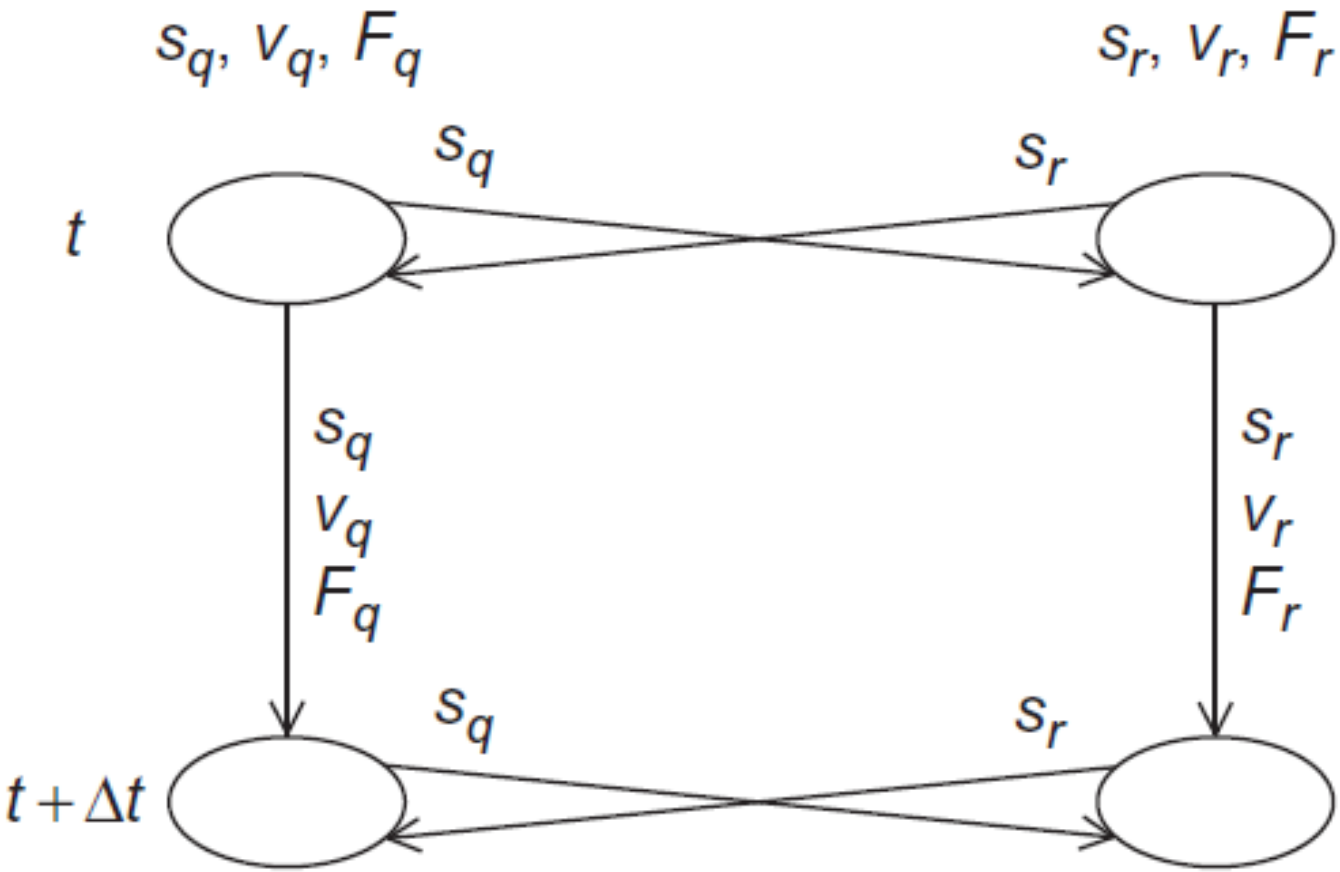
Parallelizing the N-Body Solvers

- Apply Foster's methodology.
- Initially, we want a lot of tasks.
- Start by making our tasks the computations of the positions, the velocities, and the total forces at each timestep.

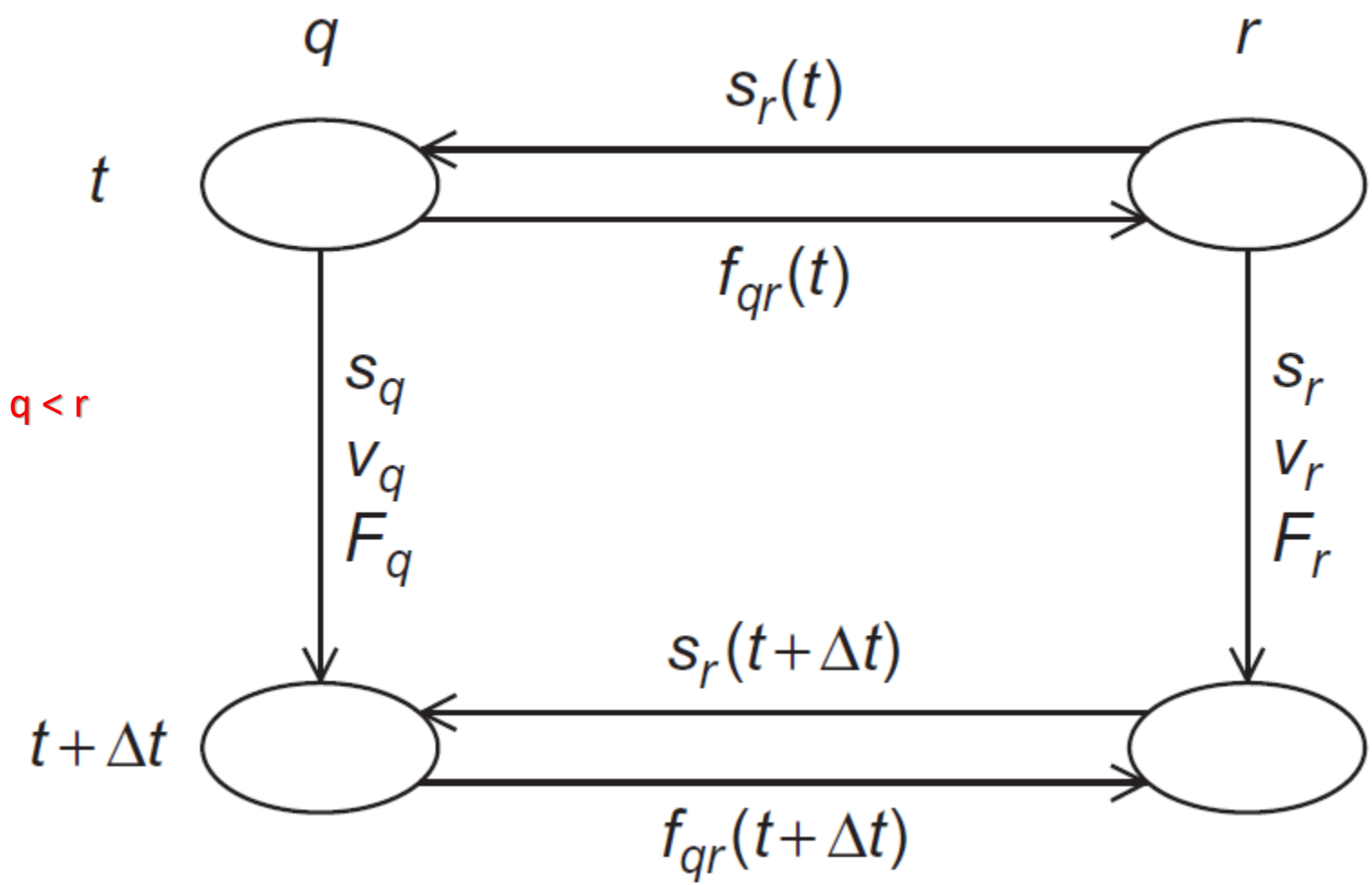
Communications Among Tasks in the Basic N-Body Solver



Communications Among Agglomerated Tasks in the Basic N-Body Solver



Communications Among Agglomerated Tasks in the Reduced N-Body Solver



Computing the total force on particle q in the reduced algorithm

```
for each particle k > q {
   x_diff = pos[q][X] − pos[k][X];
   y_diff = pos[q][Y] − pos[k][Y];
   dist = sqrt(x_diff*x_diff + y_diff*y_diff);
   dist_cubed = dist*dist*dist;
   force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
   force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff;

   forces[q][X] += force_qk[X];
   forces[q][Y] += force_qk[Y];
   forces[k][X] −= force_qk[X];
   forces[k][Y] −= force_qk[Y];
}
```

Serial pseudo-code

```
for each timestep {
   if (timestep output) Print positions and velocities of particles;
   for each particle q
      Compute total force on q;
   for each particle q
      Compute position and velocity of q;
}
```

iterating over particles

In principle, parallelizing the two inner for loops will map tasks/particles to cores.

First attempt

```
   for each timestep {
      if (timestep output) Print positions and velocities of particles;
#     pragma omp parallel for
      for each particle q
         Compute total force on q;
#     pragma omp parallel for
      for each particle q
         Compute position and velocity of q;
   }
```

Let's check for race conditions caused by loop-carried dependences.

First loop

```
#  pragma omp parallel for
   for each particle q {
      forces[q][X] = forces[q][Y] = 0;
      for each particle k != q {
         x_diff = pos[q][X] − pos[k][X];
         y_diff = pos[q][Y] − pos[k][Y];
         dist = sqrt(x_diff*x_diff + y_diff*y_diff);
         dist_cubed = dist*dist*dist;
         forces[q][X] −= G*masses[q]*masses[k]/dist_cubed * x_diff;
         forces[q][Y] −= G*masses[q]*masses[k]/dist_cubed * y_diff;
      }
   }
```

Second loop

```
#  pragma omp parallel for
   for each particle q {
      pos[q][X] += delta_t*vel[q][X];
      pos[q][Y] += delta_t*vel[q][Y];
      vel[q][X] += delta_t/masses[q]*forces[q][X];
      vel[q][Y] += delta_t/masses[q]*forces[q][Y];
   }
```

Repeated forking and joining of threads

The same team of threads will be used in both loops and for every iteration of the outer loop.

```
#   pragma omp parallel
    for each timestep {
       if (timestep output) Print positions and velocities of particles;
#      pragma omp for
       for each particle q
          Compute total force on q;
#      pragma omp for
       for each particle q
          Compute position and velocity of q;
    }
```

But every thread will print all the positions and velocities.

Adding the *single* directive

```
#   pragma omp parallel
    for each timestep {
       if (timestep output) {
#         pragma omp single
          Print positions and velocities of particles;
       }
#      pragma omp for
       for each particle q
          Compute total force on q;
#      pragma omp for
       for each particle q
          Compute position and velocity of q;
    }
```

Parallelizing the Reduced Solver Using OpenMP

```
#   pragma omp parallel
    for each timestep {
        if (timestep output) {
#           pragma omp single
            Print positions and velocities of particles;
        }
#       pragma omp for
        for each particle q
            forces[q] = 0.0;
#       pragma omp for
        for each particle q
            Compute total force on q;
#       pragma omp for
        for each particle q
            Compute position and velocity of q;
    }
```

Problems

$$\mathbf{F}_3 = -\mathbf{f}_{03} - \mathbf{f}_{13} - \mathbf{f}_{23}$$

Updates to forces[3] create a race condition.

In fact, this is the case in general.

Updates to the elements of the forces array introduce race conditions into the code.

First solution attempt

before all the updates to forces

```
#  pragma omp critical
   {
      forces[q][X] += force_qk[X];
      forces[q][Y] += force_qk[Y];
      forces[k][X] -= force_qk[X];
      forces[k][Y] -= force_qk[Y];
   }
```

Access to the forces array will be effectively serialized!!!

Second solution attempt

```
omp_set_lock(locks[q]);
forces[q][X] += force_qk[X];
forces[q][Y] += force_qk[Y];
omp_unset_lock(locks[q]);

omp_set_lock(locks[k]);
forces[k][X] -= force_qk[X];
forces[k][Y] -= force_qk[Y];
omp_unset_lock(locks[k]);
```

Use one lock for each particle.

First Phase Computations for Reduced Algorithm with Block Partition

		Thread		
Thread	Particle	0	1	2
0	0	$\mathbf{f}_{01}+\mathbf{f}_{02}+\mathbf{f}_{03}+\mathbf{f}_{04}+\mathbf{f}_{05}$	0	0
	1	$-\mathbf{f}_{01}+\mathbf{f}_{12}+\mathbf{f}_{13}+\mathbf{f}_{14}+\mathbf{f}_{15}$	0	0
1	2	$-\mathbf{f}_{02}-\mathbf{f}_{12}$	$\mathbf{f}_{23}+\mathbf{f}_{24}+\mathbf{f}_{25}$	0
	3	$-\mathbf{f}_{03}-\mathbf{f}_{13}$	$-\mathbf{f}_{23}+\mathbf{f}_{34}+\mathbf{f}_{35}$	0
2	4	$-\mathbf{f}_{04}-\mathbf{f}_{14}$	$-\mathbf{f}_{24}-\mathbf{f}_{34}$	$\mathbf{f}_{45}$
	5	$-\mathbf{f}_{05}-\mathbf{f}_{15}$	$-\mathbf{f}_{25}-\mathbf{f}_{35}$	$-\mathbf{f}_{45}$

First Phase Computations for Reduced Algorithm with Cyclic Partition

		Thread		
Thread	Particle	0	1	2
0	0	$\mathbf{f}_{01}+\mathbf{f}_{02}+\mathbf{f}_{03}+\mathbf{f}_{04}+\mathbf{f}_{05}$	0	0
1	1	$-\mathbf{f}_{01}$	$\mathbf{f}_{12}+\mathbf{f}_{13}+\mathbf{f}_{14}+\mathbf{f}_{15}$	0
2	2	$-\mathbf{f}_{02}$	$-\mathbf{f}_{12}$	$\mathbf{f}_{23}+\mathbf{f}_{24}+\mathbf{f}_{25}$
0	3	$-\mathbf{f}_{03}+\mathbf{f}_{34}+\mathbf{f}_{35}$	$-\mathbf{f}_{13}$	$-\mathbf{f}_{23}$
1	4	$-\mathbf{f}_{04}-\mathbf{f}_{34}$	$-\mathbf{f}_{14}+\mathbf{f}_{45}$	$-\mathbf{f}_{24}$
2	5	$-\mathbf{f}_{05}-\mathbf{f}_{35}$	$-\mathbf{f}_{15}-\mathbf{f}_{45}$	$-\mathbf{f}_{25}$

Revised algorithm – phase I

```
#  pragma omp for
   for each particle q {
      force_qk[X] = force_qk[Y] = 0;
      for each particle k > q {
         x_diff = pos[q][X] − pos[k][X];
         y_diff = pos[q][Y] − pos[k][Y];
         dist = sqrt(x_diff*x_diff + y_diff*y_diff);
         dist_cubed = dist*dist*dist;
         force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
         force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff;

         loc_forces[my_rank][q][X] += force_qk[X];
         loc_forces[my_rank][q][Y] += force_qk[Y];
         loc_forces[my_rank][k][X] −= force_qk[X];
         loc_forces[my_rank][k][Y] −= force_qk[Y];
      }
   }
```

Revised algorithm – phase II

```
#  pragma omp for
   for (q = 0; q < n; q++) {
      forces[q][X] = forces[q][Y] = 0;
      for (thread = 0; thread < thread_count; thread++) {
         forces[q][X] += loc_forces[thread][q][X];
         forces[q][Y] += loc_forces[thread][q][Y];
      }
   }
```

Parallelizing the Solvers Using Pthreads

- By default local variables in Pthreads are private. So all shared variables are global in the Pthreads version.
- The principle data structures in the Pthreads version are identical to those in the OpenMP version: vectors are two-dimensional arrays of doubles, and the mass, position, and velocity of a single particle are stored in a struct.
- The forces are stored in an array of vectors.

Parallelizing the Solvers Using Pthreads

- Startup for Pthreads is basically the same as the startup for OpenMP: the main thread gets the command line arguments, and allocates and initializes the principle data structures.
- The main difference between the Pthreads and the OpenMP implementations is in the details of parallelizing the inner loops.
- Since Pthreads has nothing analogous to a parallel for directive, we must explicitly determine which values of the loop variables correspond to each thread's calculations.

Parallelizing the Solvers Using Pthreads

- Another difference between the Pthreads and the OpenMP versions has to do with barriers.
- At the end of a parallel for OpenMP has an implied barrier.
- We need to add explicit barriers after the inner loops when a race condition can arise.
- The Pthreads standard includes a barrier.
- However, some systems don’t implement it.
- If a barrier isn't defined we must define a function that uses a Pthreads condition variable to implement a barrier.

Parallelizing the Basic Solver Using MPI

- Choices with respect to the data structures:
  - Each process stores the entire global array of particle masses.
  - Each process only uses a single n-element array for the positions.
  - Each process uses a pointer loc_pos that refers to the start of its block of pos.
  - So on process 0 local_pos = pos; on process 1 local_pos = pos + loc_n; and so on.

Pseudo-code for the MPI version of the basic n-body solver

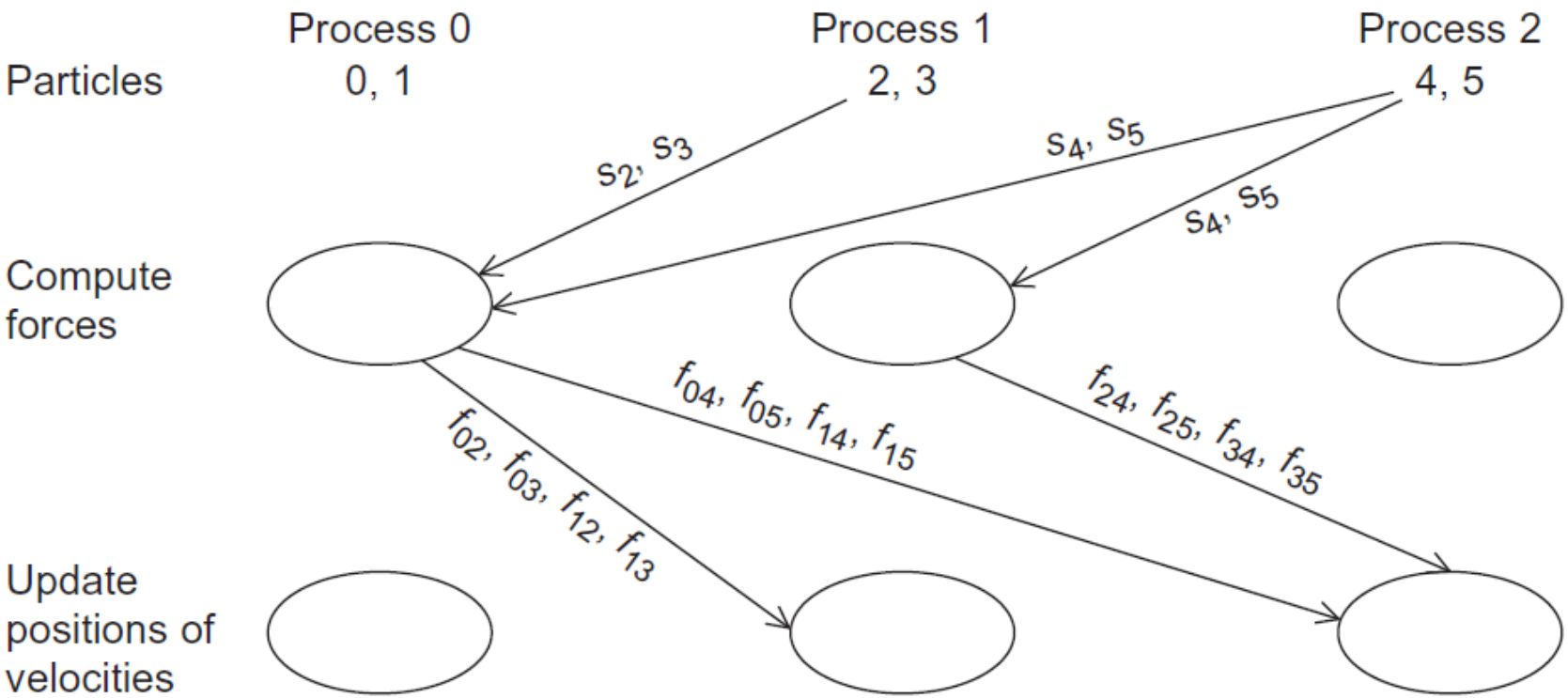
```
Get input data;
for each timestep {
   if (timestep output)
      Print positions and velocities of particles;
   for each local particle loc_q
      Compute total force on loc_q;
   for each local particle loc_q
      Compute position and velocity of loc_q;
   Allgather local positions into global pos array;
}
Print positions and velocities of particles;
```

Pseudo-code for output

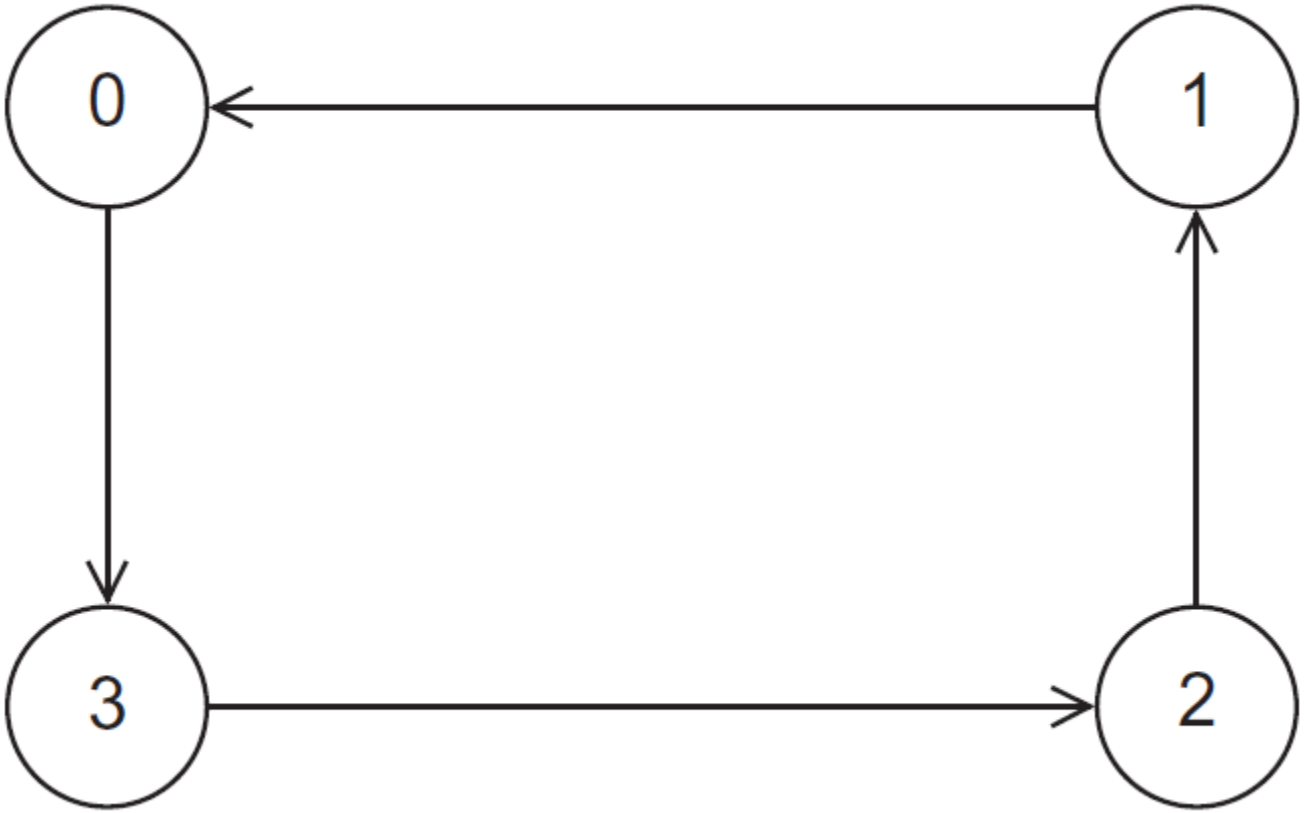
```
Gather velocities onto process 0;
if (my_rank == 0) {
   Print timestep;
   for each particle
      Print pos[particle] and vel[particle]
}
```

Communication In A Possible MPI Implementation of the N-Body Solver

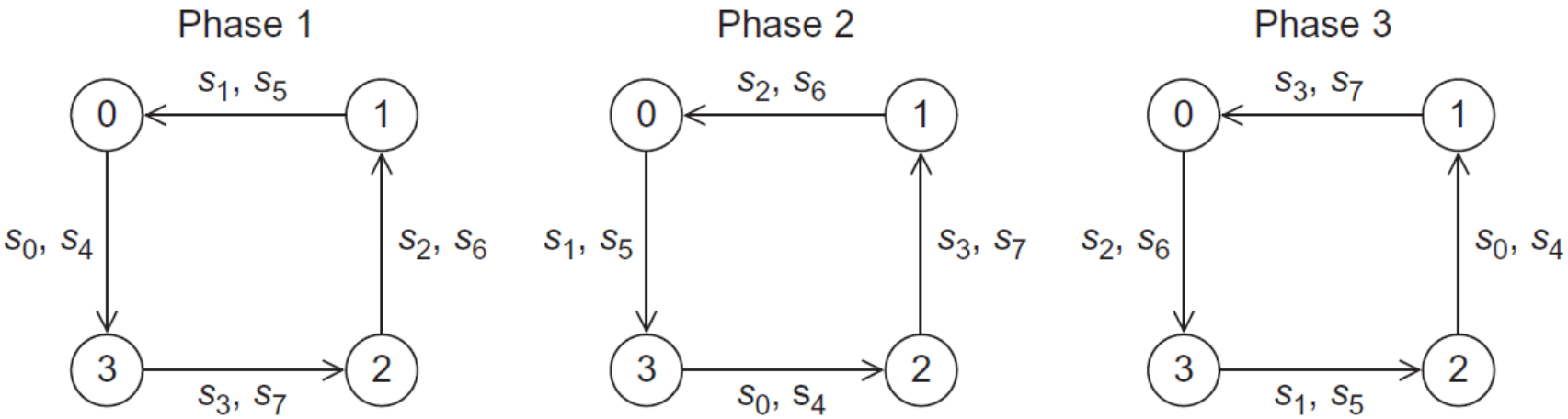
(for a reduced solver)



A Ring of Processes



Ring Pass of Positions



Computation of Forces in Ring Pass (1)

Time	Variable	Process 0	Process 1
Start	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$0, 0$	$0, 0$
	tmp_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	tmp_forces	$0, 0$	$0, 0$
After Comp of Forces	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$\mathbf{f}_{02}, 0$	$\mathbf{f}_{13}, 0$
	tmp_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	tmp_forces	$0, -\mathbf{f}_{02}$	$0, -\mathbf{f}_{13}$
After First Comm	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$\mathbf{f}_{02}, 0$	$\mathbf{f}_{13}, 0$
	tmp_pos	$\mathbf{s}_1, \mathbf{s}_3$	$\mathbf{s}_0, \mathbf{s}_2$
	tmp_forces	$0, -\mathbf{f}_{13}$	$0, -\mathbf{f}_{02}$
After Comp of Forces	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$\mathbf{f}_{01} + \mathbf{f}_{02} + \mathbf{f}_{03}, \mathbf{f}_{23}$	$\mathbf{f}_{12} + \mathbf{f}_{13}, 0$
	tmp_pos	$\mathbf{s}_1, \mathbf{s}_3$	$\mathbf{s}_0, \mathbf{s}_2$
	tmp_forces	$-\mathbf{f}_{01}, -\mathbf{f}_{03} - \mathbf{f}_{13} - \mathbf{f}_{23}$	$0, -\mathbf{f}_{02} - \mathbf{f}_{12}$

Computation of Forces in Ring Pass (2)

Time	Variable	Process 0	Process 1
After Second Comm	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$\mathbf{f}_{01} + \mathbf{f}_{02} + \mathbf{f}_{03}, \mathbf{f}_{23}$	$\mathbf{f}_{12} + \mathbf{f}_{13}, 0$
	tmp_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	tmp_forces	$0, -\mathbf{f}_{02} - \mathbf{f}_{12}$	$-\mathbf{f}_{01}, -\mathbf{f}_{03} - \mathbf{f}_{13} - \mathbf{f}_{23}$
After Comp of Forces	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$\mathbf{f}_{01} + \mathbf{f}_{02} + \mathbf{f}_{03}, -\mathbf{f}_{02} - \mathbf{f}_{12} + \mathbf{f}_{23}$	$-\mathbf{f}_{01} + \mathbf{f}_{12} + \mathbf{f}_{13}, -\mathbf{f}_{03} - \mathbf{f}_{13} - \mathbf{f}_{23}$
	tmp_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	tmp_forces	$0, -\mathbf{f}_{02} - \mathbf{f}_{12}$	$-\mathbf{f}_{01}, -\mathbf{f}_{03} - \mathbf{f}13 - \mathbf{f}_{23}$

Pseudo-code for the MPI implementation of the reduced n-body solver

```
source = (my_rank + 1) % comm_sz;
dest = (my_rank − 1 + comm_sz) % comm_sz;
Copy loc_pos into tmp_pos;
loc_forces = tmp_forces = 0;

Compute forces due to interactions among local particles;
for (phase = 1; phase < comm_sz; phase++) {
   Send current tmp_pos and tmp_forces to dest;
   Receive new tmp_pos and tmp_forces from source;
   /* Owner of the positions and forces we're receiving */
   owner = (my_rank + phase) % comm_sz;
   Compute forces due to interactions among my particles
      and owner's particles;
}
Send current tmp_pos and tmp_forces to dest;
Receive new tmp_pos and tmp_forces from source;
```

Loops iterating through global particle indexes

```
for (loc_part1 = 0, glb_part1 = my_rank;
     loc_part1 < loc_n-1;
     loc_part1++, glb_part1 += comm_sz)
   for (glb_part2 = First_index(glb_part1, my_rank, owner, comm_sz),
        loc_part2 = Global_to_local(glb_part2, owner, loc_n);
        loc_part2 < loc_n;
        loc_part2++, glb_part2 += comm_sz)
        Compute_force(loc_pos[loc_part1], masses[glb_part1],
              tmp_pos[loc_part2], masses[glb_part2],
              loc_forces[loc_part1], tmp_forces[loc_part2]);
```

Performance of the MPI n-body solvers

Processes	Basic	Reduced
1	17.30	8.68
2	8.65	4.45
4	4.35	2.30
8	2.20	1.26
16	1.13	0.78

(in seconds)

Run-Times for OpenMP and MPI N-Body Solvers

Processes/ Threads	OpenMP		MPI	
	Basic	Reduced	Basic	Reduced
1	15.13	8.77	17.30	8.68
2	7.62	4.42	8.65	4.45
4	3.85	2.26	4.35	2.30

(in seconds)

Concluding Remarks (1)

- In developing the reduced MPI solution to the n-body problem, the “ring pass” algorithm proved to be much easier to implement and is probably more scalable.

- In a distributed memory environment in which processes send each other work, determining when to terminate is a nontrivial problem.

Concluding Remarks (2)

- When deciding which API to use, we should consider whether to use shared- or distributed-memory.

- We should look at the memory requirements of the application and the amount of communication among the processes/threads.

Concluding Remarks (3)

- If the memory requirements are great or the distributed memory version can work mainly with cache, then a distributed memory program is likely to be much faster.

- On the other hand if there is considerable communication, a shared memory program will probably be faster.

Concluding Remarks (3)

- In choosing between OpenMP and Pthreads, if there's an existing serial program and it can be parallelized by the insertion of OpenMP directives, then OpenMP is probably the clear choice.

- However, if complex thread synchronization is needed then Pthreads will be easier to use.