



中山大學
SUN YAT-SEN UNIVERSITY



国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

并行程序开发

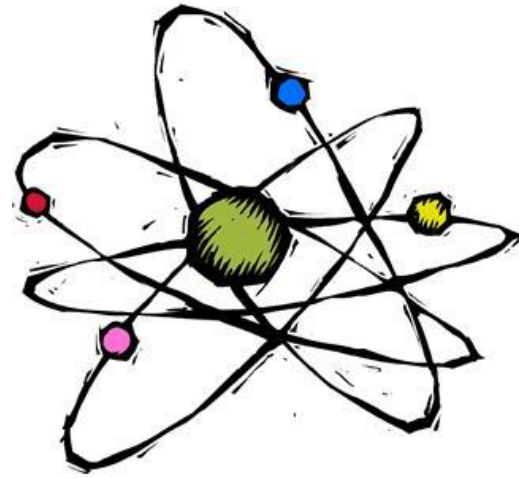
任课教师：吴迪

课程内容

- 参考资料：
 - 并行政程序设计导论, Peter S Pacheco, 机械工业出版社, 2016
 - Chapter 6 Parallel Program Development

Roadmap

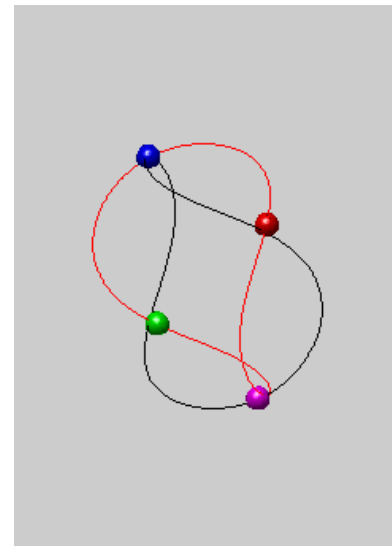
- Solving non-trivial problems.
- The n-body problem.
- The traveling salesman problem.
- Applying Foster's methodology.
- Starting from scratch on algorithms that have no serial analog.

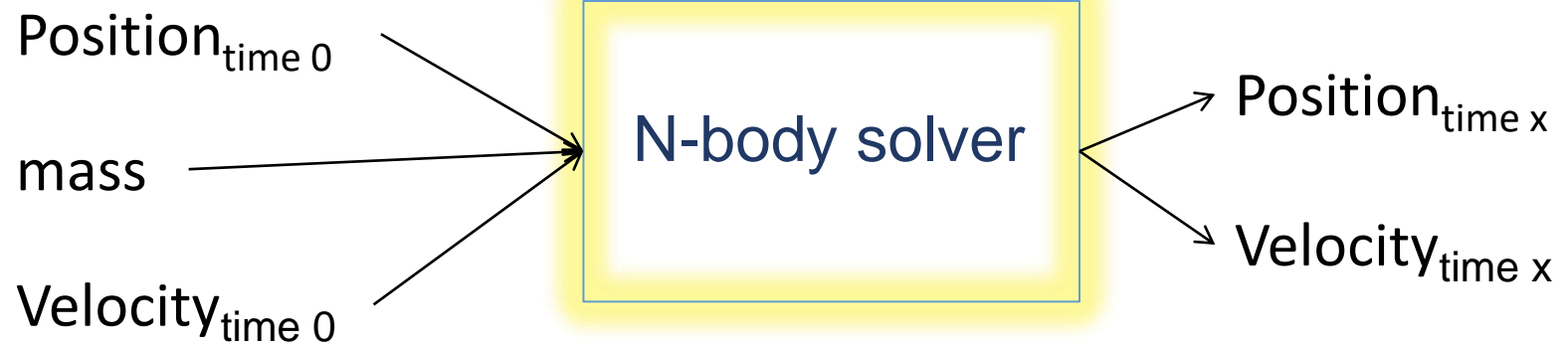


Two N-Body Solvers

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.

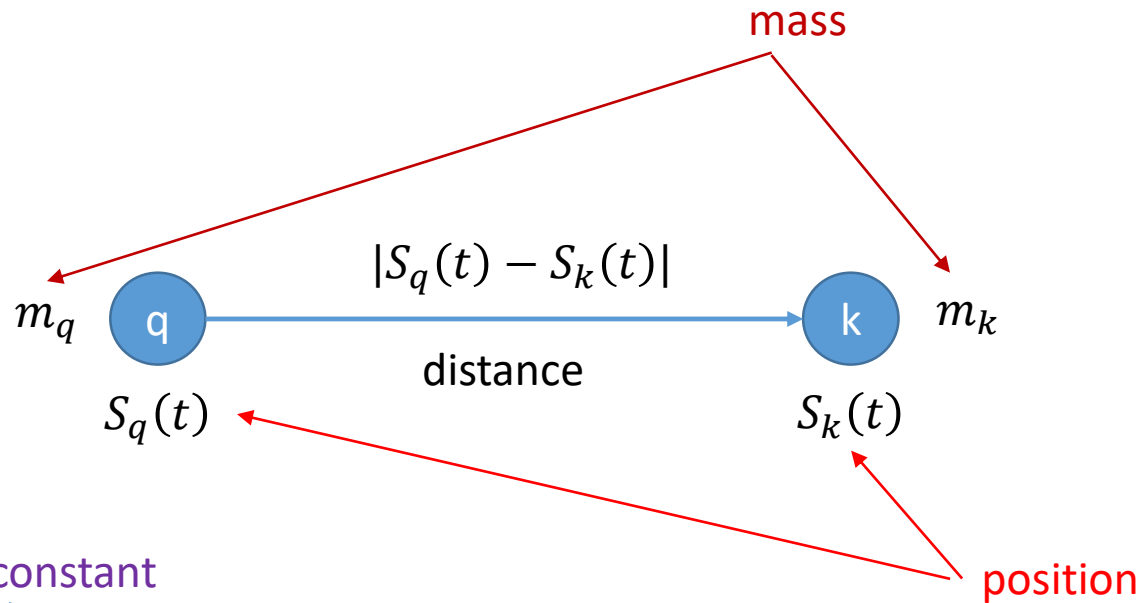




Simulating motion of planets

- Determine the positions and velocities:
 - Newton's [second law of motion](#).
 - Newton's [law of universal gravitation](#).

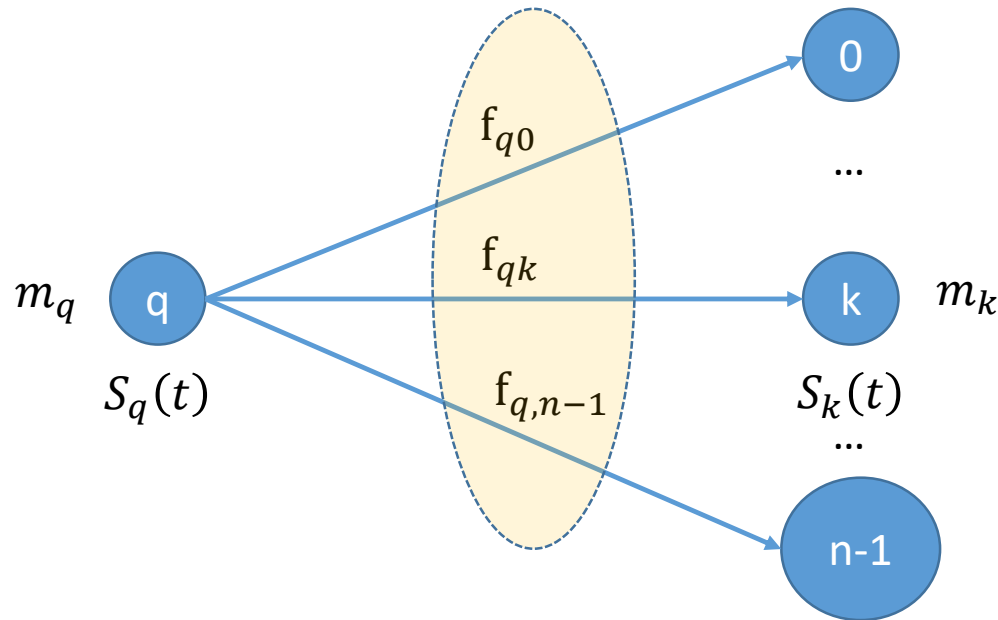
Force on particle q exerted by k



G: gravitational constant

$$\mathbf{f}_{qk}(t) = -\frac{\boxed{G}m_qm_k}{|S_q(t) - S_k(t)|^3} [S_q(t) - S_k(t)] \quad (6.1)$$

Total force on particle q



$$\mathbf{F}_q(t) = \sum_{\substack{k=0 \\ k \neq q}}^{n-1} \mathbf{f}_{qk} = -Gm_q \sum_{\substack{k=0 \\ k \neq q}}^{n-1} \frac{m_k}{|\mathbf{s}_q(t) - \mathbf{s}_k(t)|^3} [\mathbf{s}_q(t) - \mathbf{s}_k(t)] \quad (6.2)$$

Acceleration of particle q

- According to Newton's second law of motion:

$$F_q(t) = m_q a_q(t) = m_q S_q''(t)$$



$$a_q(t) = S_q''(t) = \frac{F_q(t)}{m_q}$$



$$s_q''(t) = -G \sum_{\substack{j=0 \\ j \neq q}}^{n-1} \frac{m_j}{|\mathbf{s}_q(t) - \mathbf{s}_j(t)|^3} [\mathbf{s}_q(t) - \mathbf{s}_j(t)] \quad (6.3)$$

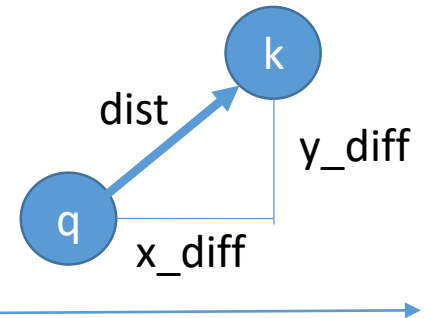
$t = 0, \Delta t, 2\Delta t, \dots, T\Delta t$

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

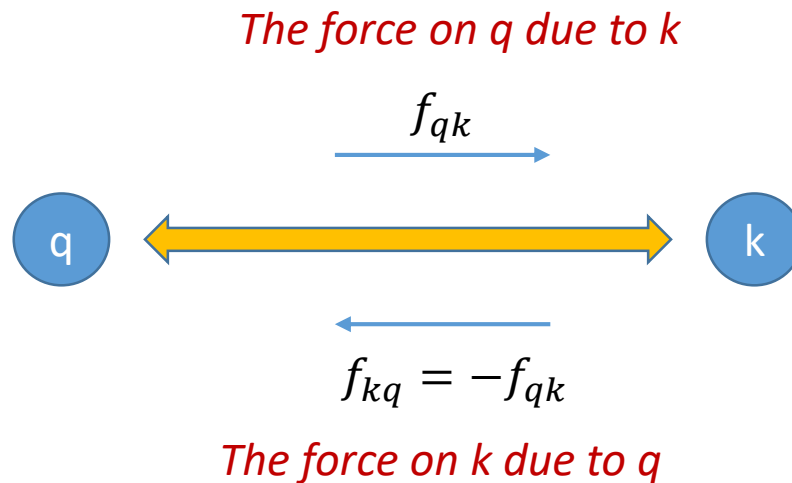
```
for each particle q {  
  for each particle k != q {  
    x_diff = pos[q][X] - pos[k][X];  
    y_diff = pos[q][Y] - pos[k][Y];  
    dist = sqrt(x_diff*x_diff + y_diff*y_diff);  
    dist_cubed = dist*dist*dist;  
    forces[q][X] -= G*masses[q]*masses[k]/dist_cubed * x_diff;  
    forces[q][Y]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



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

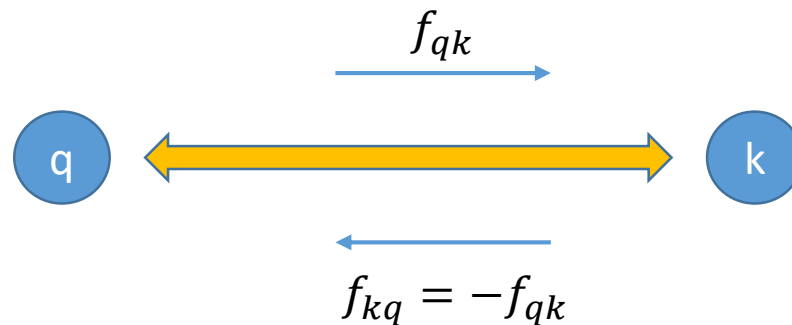
Newton's third law of motion

- For every action, there is an equal and opposite reaction.



The individual forces

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

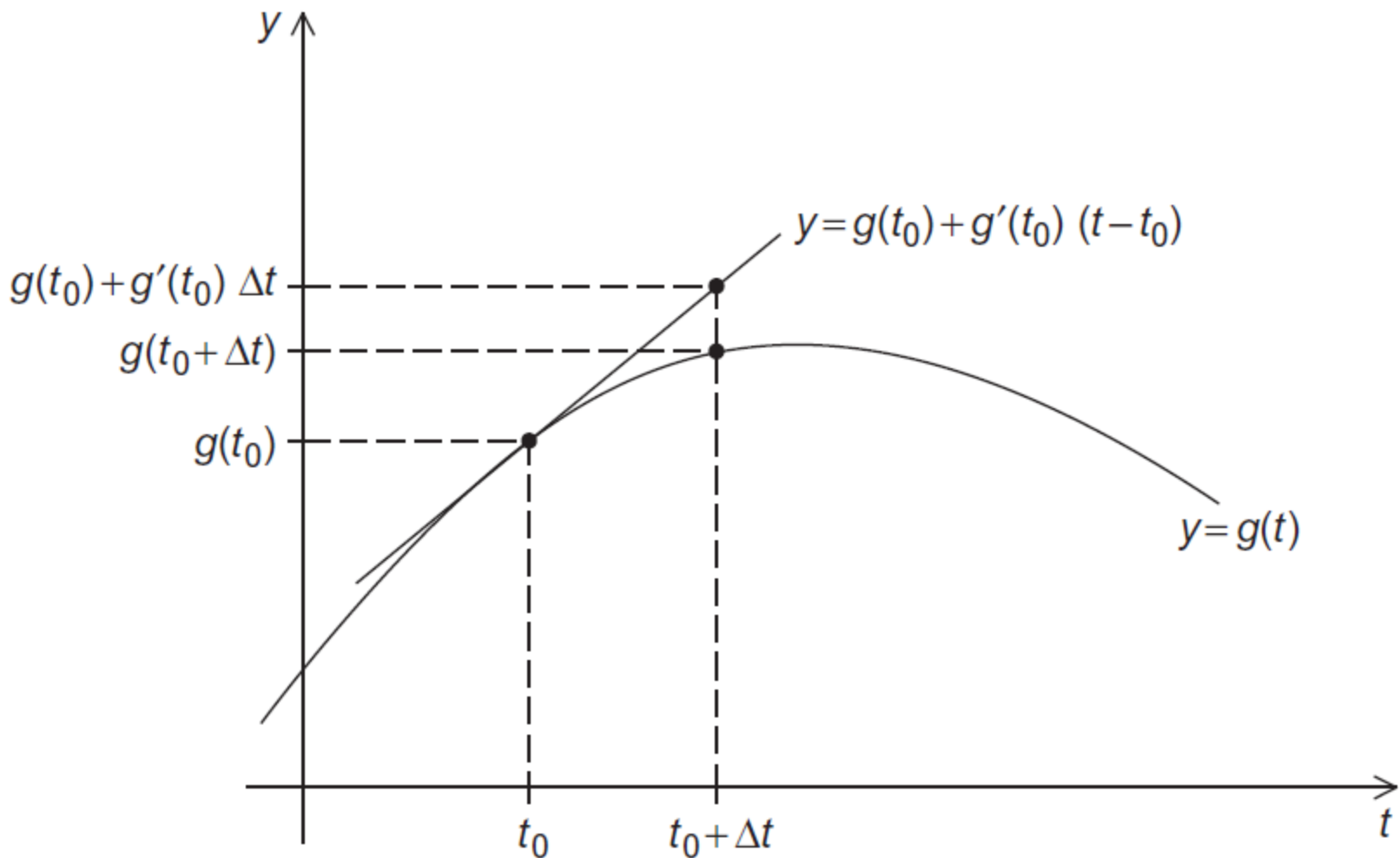


A Reduced Algorithm for Computing N-Body Forces

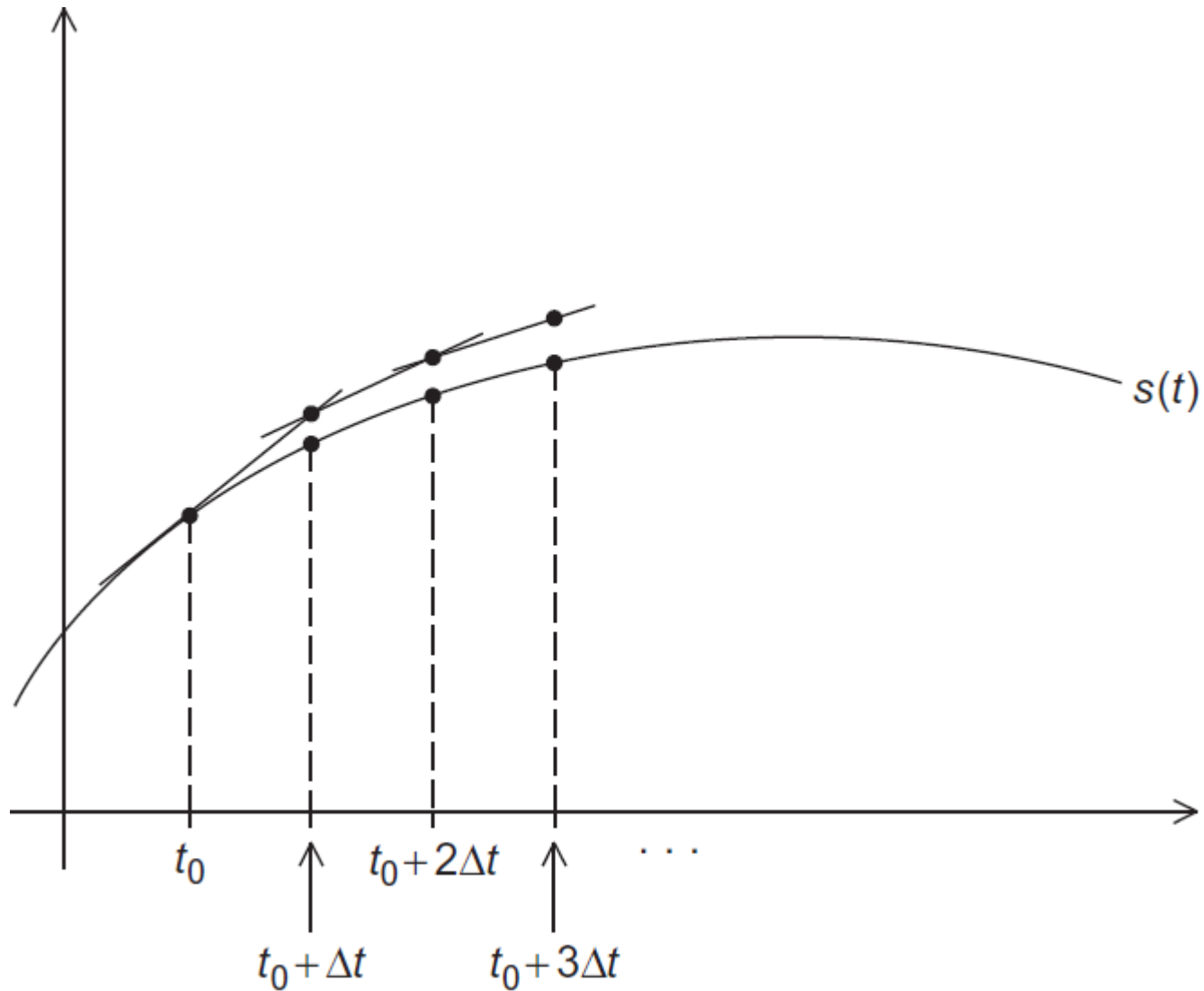
```
for each particle q
    forces[q] = 0;
for each particle q {
    for each particle k > q {
        x_diff = pos[q][X] - pos[k][X];
        y_diff = pos[q][Y] - pos[k][Y];
        dist = sqrt(x_diff*x_diff + y_diff*y_diff);
        dist_cubed = dist*dist*dist;
        force_qk[X] = G*masses[q]*masses[k]/dist_cubed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

Using the Tangent Line to Approximate a Function



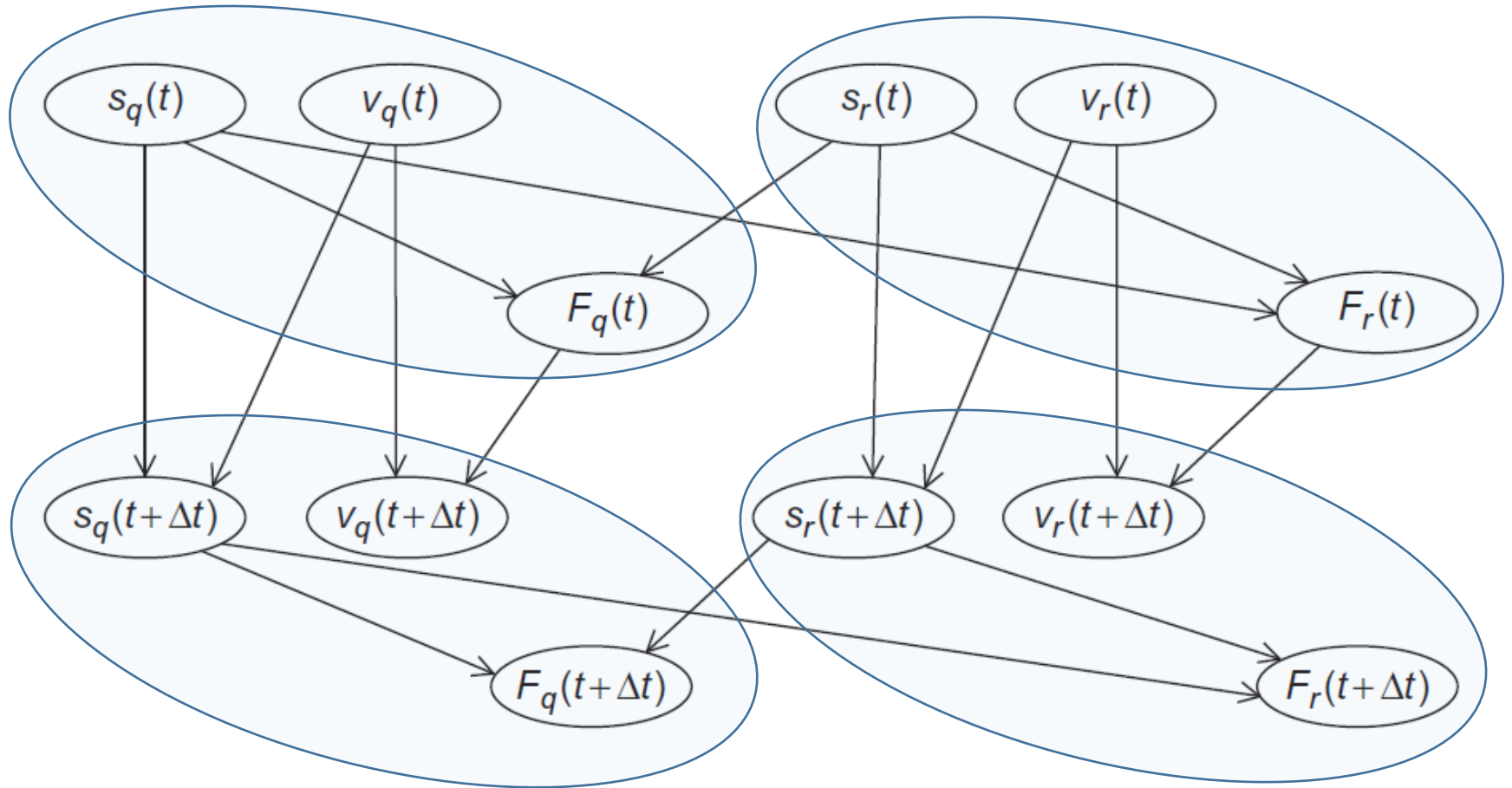
Euler's Method



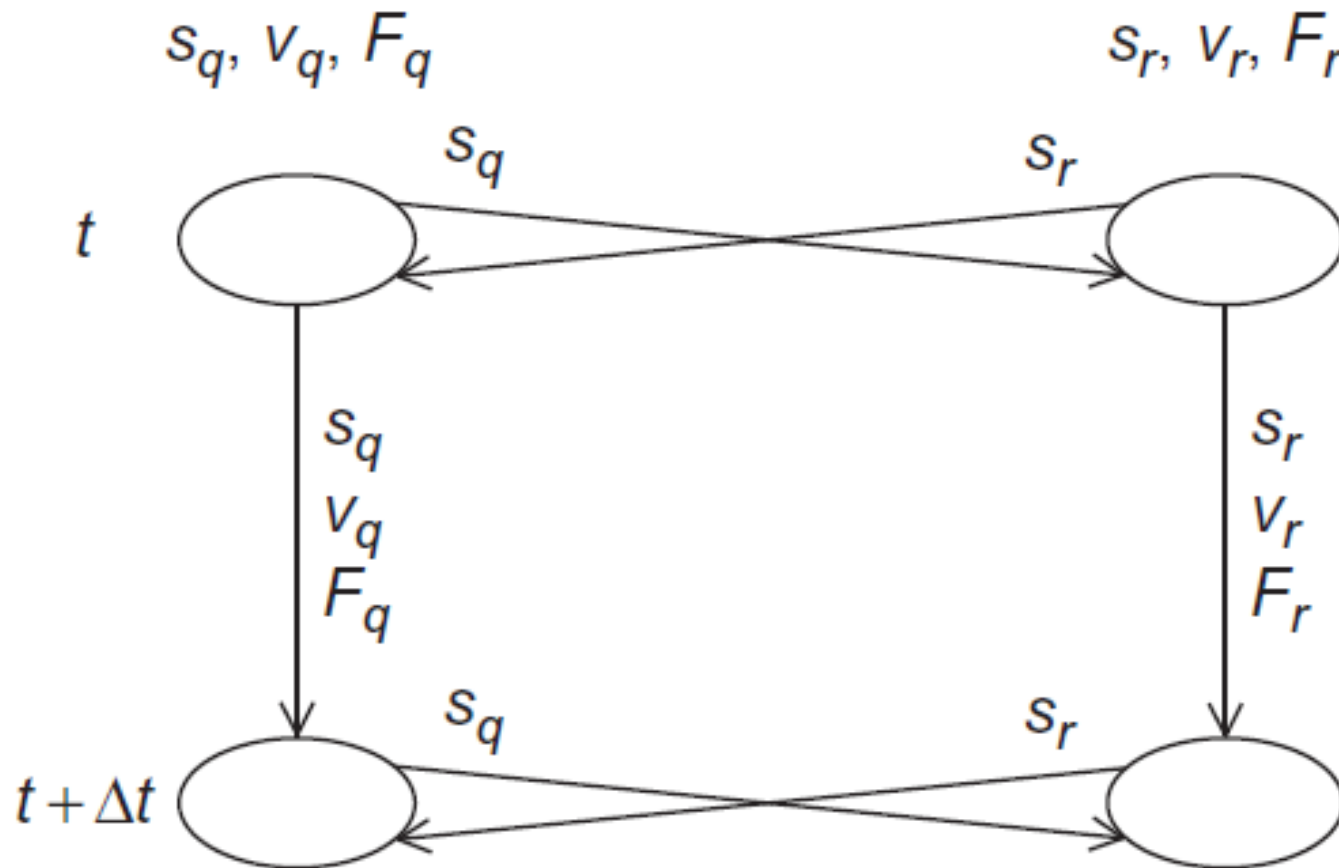
Parallelizing the N-Body Solvers

- 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.

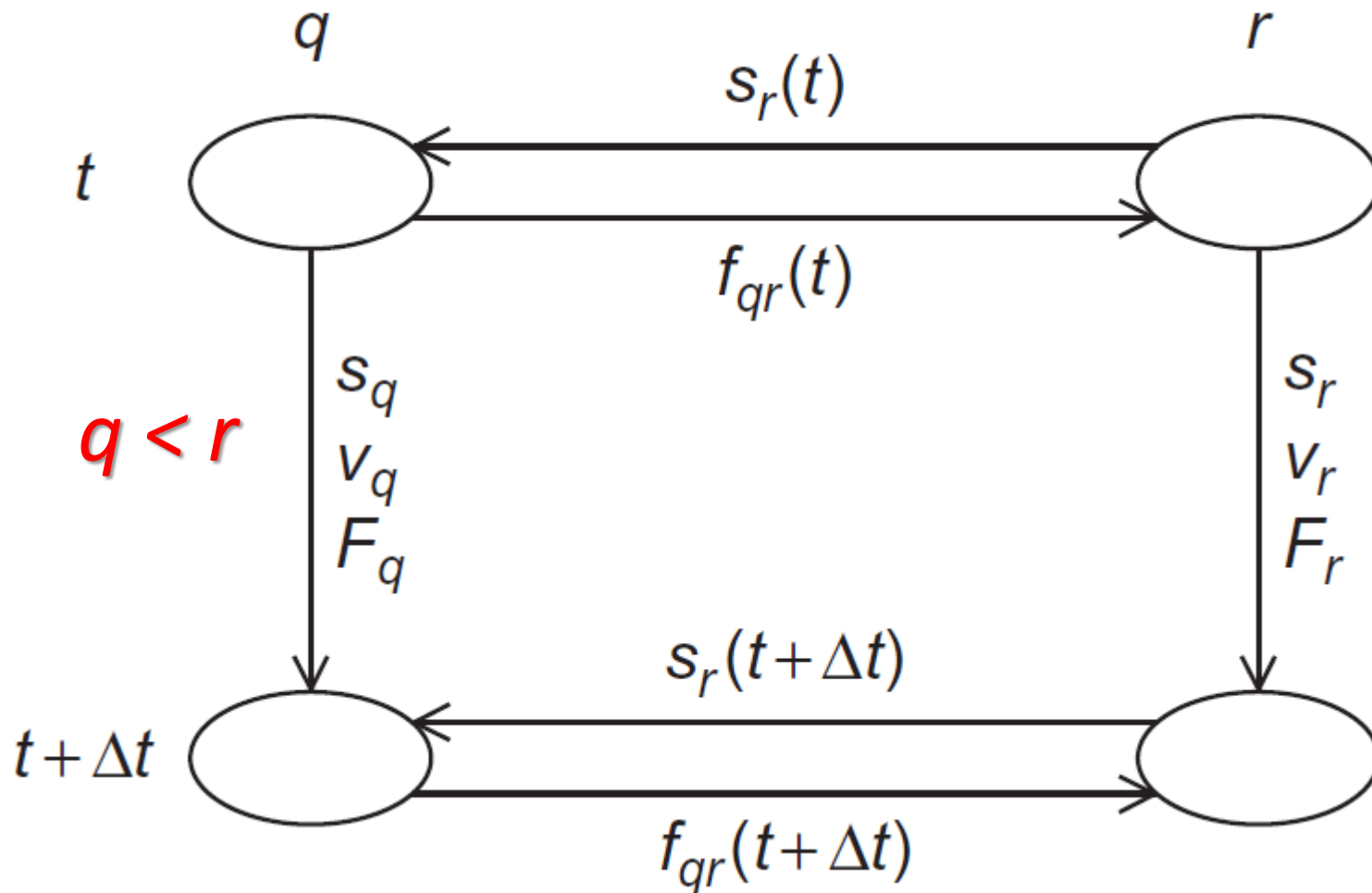
Communications Among Tasks in the Basic N-Body Solver



Communications Among Agglomerated Tasks in the Basic N-Body Solver



Communications Among Agglomerated Tasks in the Reduced N-Body Solver

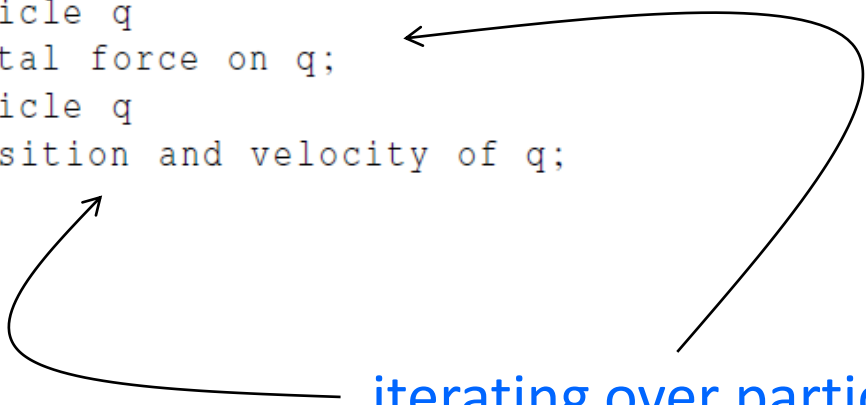


Computing the total force on particle q in the reduced algorithm

```
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

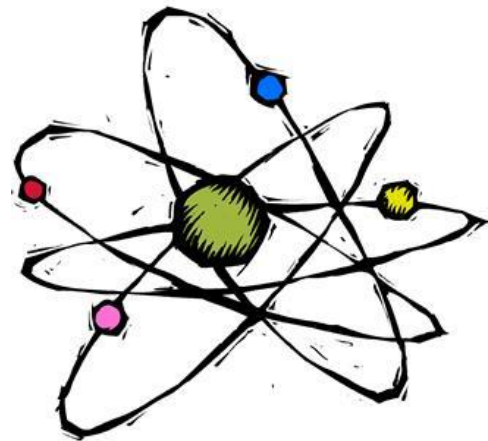
```
for each timestep {  
    if (timestep output) Print positions and velocities of particles;  
    for each particle q  
        Compute total force on q;  
    for each particle q  
        Compute position and velocity of q;  
}
```



iterating over particles

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

Parallelizing the Basic Solver using OpenMP



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;
    }
}
```

No race conditions in parallelization of the first loop

- Only one thread will access `forces[q]` for any `q`.
- Accesses to *pos* and *masses* array are read.

Second loop

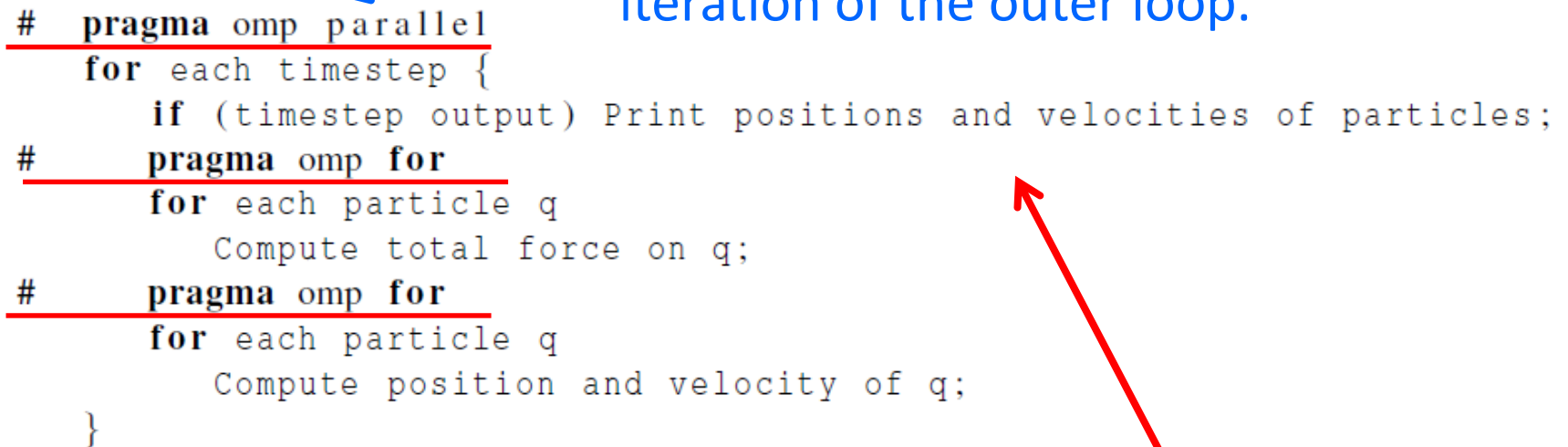
```
# pragma omp parallel for  
  for each particle q {  
    pos[q][X] += delta_t*vel[q][X];  
    pos[q][Y] += delta_t*vel[q][Y];  
    vel[q][X] += delta_t/masses[q]*forces[q][X];  
    vel[q][Y] += delta_t/masses[q]*forces[q][Y];  
  }
```

No race conditions in parallelization of the second loop

- A single thread accesses pos[q], vel[q], masses[q], forces[q] for any particle q.
- Scalar variables are only read.

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;
}
```

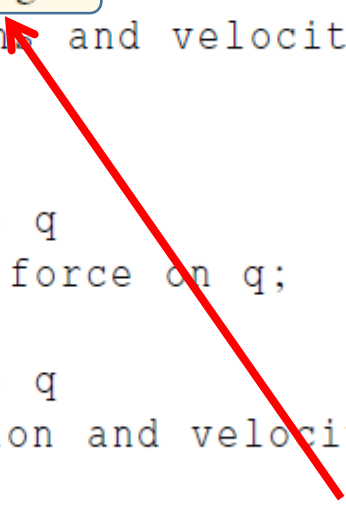
The diagram illustrates nested OpenMP loops. A blue arrow points from the text 'The same team of threads will be used in both loops and for every iteration of the outer loop.' to the `# pragma omp parallel` line. A red arrow points from the text '**Problem:** every thread will print all the positions and velocities.' to the `if (timestep output) Print positions and velocities of particles;` line. The code shows an outer loop over timesteps and two inner loops over particles. The first inner loop prints positions and velocities, and the second inner loop computes the total force and position/velocity. The `pragma omp parallel` and `pragma omp for` directives are underlined in red.

Problem: every thread will print all the positions and velocities.

Adding the single directive

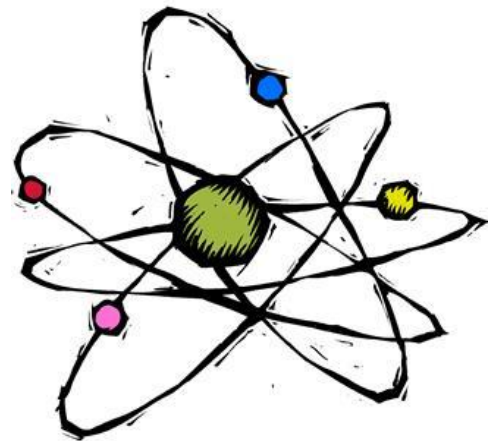
```
# pragma omp parallel
  for each timestep {
    if (timestep output) {
#      pragma omp single
        Print position and velocities of particles;
    }

#    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;
  }
```



Single: only one of the threads can execute the following block of code.


Parallelizing the Reduced Solver using OpenMP



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;
  }
```

Parallelization of the initialization of forces is fine, as there is no dependence among the iterations.



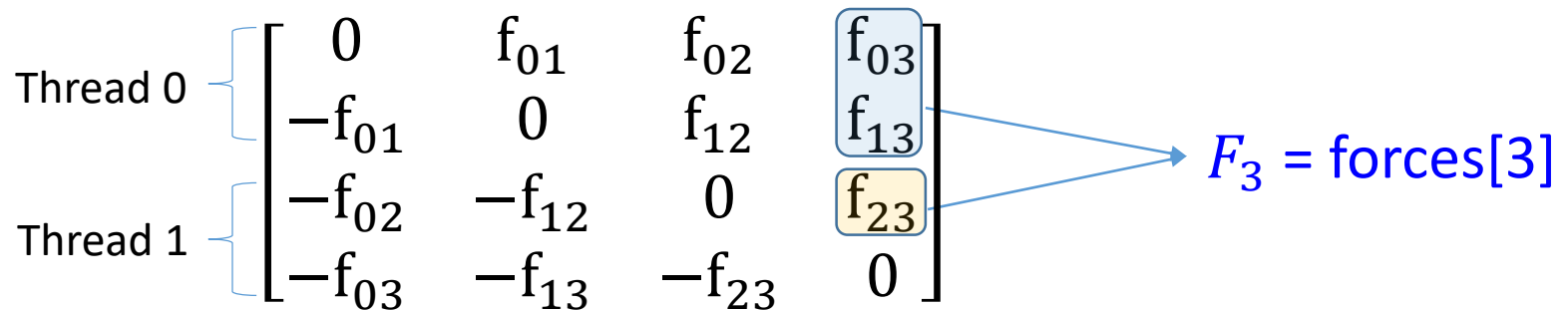
Problems



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


Updates to `forces[3]` create a **race condition**. **Why?**

Unlike basic version, in the reduced version, a thread **may** update elements in the forces array other than those corresponding to its assigned particles.



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];
}
```

Drawback: 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.

The performance is not as good as serial code.

Third solution attempt

Solution: to carry out the computation of forces in two phases:

(Phase I) each thread performs calculations and stores results in its **own** array of forces;

(Phase II) the thread that has been assigned particle q will **add** the **contributions** that have been computed by **different** threads.

$$\mathbf{F}_3 = -\underbrace{\mathbf{f}_{03}}_{\text{Thread 0}} - \underbrace{\mathbf{f}_{13}}_{\text{Thread 1}} - \mathbf{f}_{23}$$

Thread 0 Thread 1

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];
    }
}
```

(1) each thread performs calculations and stores results in its **own** array of forces;

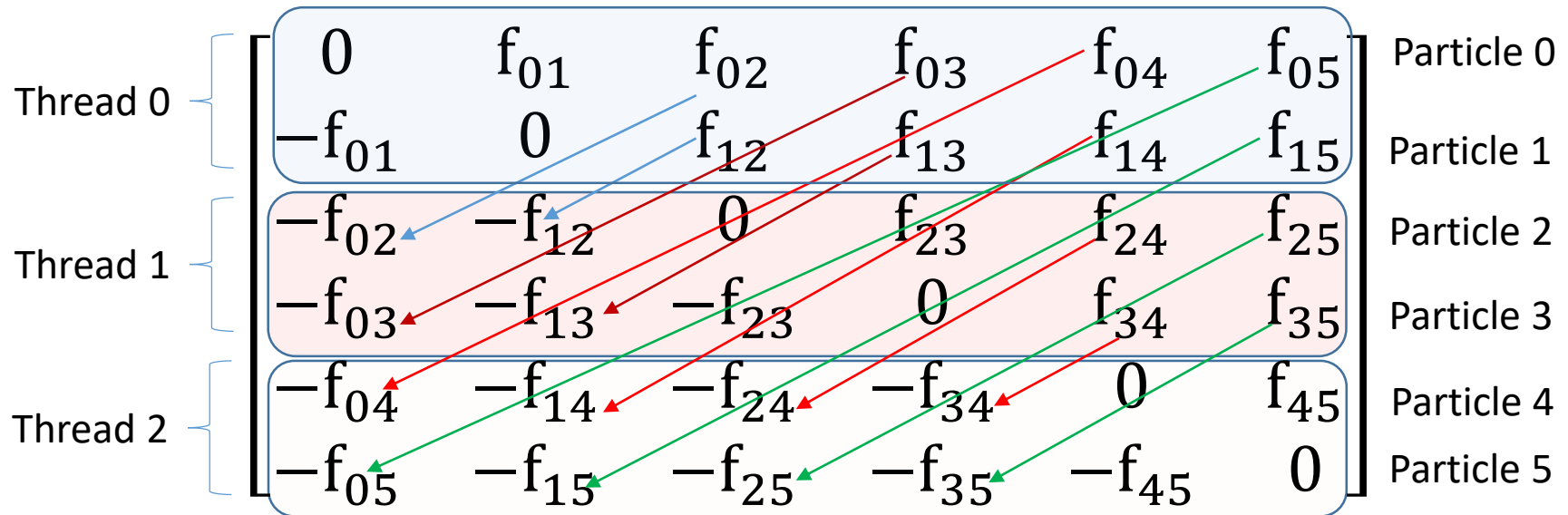
Revised algorithm – phase II

```
# 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];
    }
}
```

(2) the thread that has been assigned particle q will **add the contributions** that have been computed by **different** threads.

Another example

- 6 particles, 3 threads, **block partition**



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	$-\mathbf{f}_{05} - \mathbf{f}_{15}$	$-\mathbf{f}_{25} - \mathbf{f}_{35}$	$-\mathbf{f}_{45}$

Imbalanced workload among three threads

Another example - continued

- 6 particles, 3 threads, **cyclic partition**

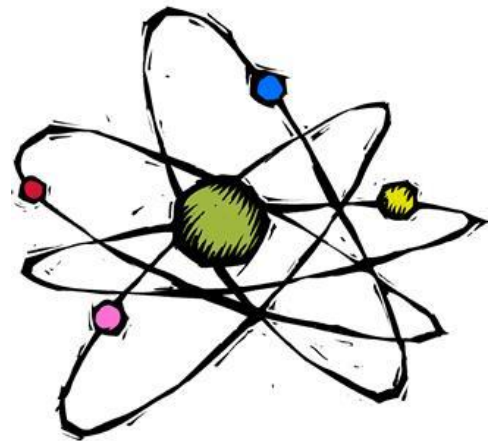
Thread 0	0	f_{01}	f_{02}	f_{03}	f_{04}	f_{05}	Particle 0
Thread 1	$-f_{01}$	0	f_{12}	f_{13}	f_{14}	f_{15}	Particle 1
Thread 2	$-f_{02}$	$-f_{12}$	0	f_{23}	f_{24}	f_{25}	Particle 2
Thread 0	$-f_{03}$	$-f_{13}$	$-f_{23}$	0	f_{34}	f_{35}	Particle 3
Thread 1	$-f_{04}$	$-f_{14}$	$-f_{24}$	$-f_{34}$	0	f_{45}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

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	$-\mathbf{f}_{05} - \mathbf{f}_{35}$	$-\mathbf{f}_{15} - \mathbf{f}_{45}$	$-\mathbf{f}_{25}$

More balanced workload among three threads with cyclic partition

Parallelizing the Solvers using Pthreads



Parallelizing the Solvers Using Pthreads

- By default, **local** variables in Pthreads are **private**.
- All **shared** variables are **global** in the Pthreads.
- The principle **data structures** in the Pthreads version are **identical** to those in the OpenMP version
- **Startup** for Pthreads is basically **the same** as the startup for OpenMP

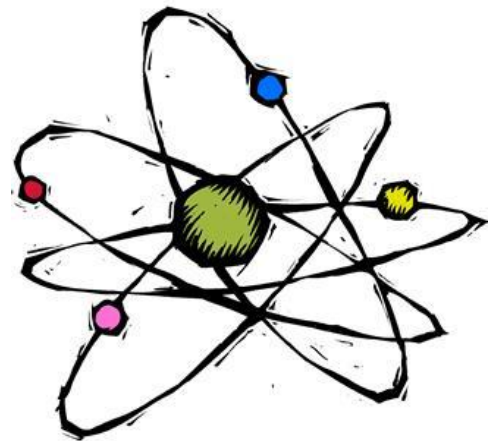
Parallelizing the Solvers Using Pthreads

- The main **difference** between Pthreads and OpenMP is in the **details of parallelizing the inner loops**.
- 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**.
 - For Pthreads, 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.

Parallelizing the Solvers using MPI



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**.
 - On process 0 $\text{local_pos} = \text{pos}$;
on process 1 $\text{local_pos} = \text{pos} + \text{loc_n}$;
etc.

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

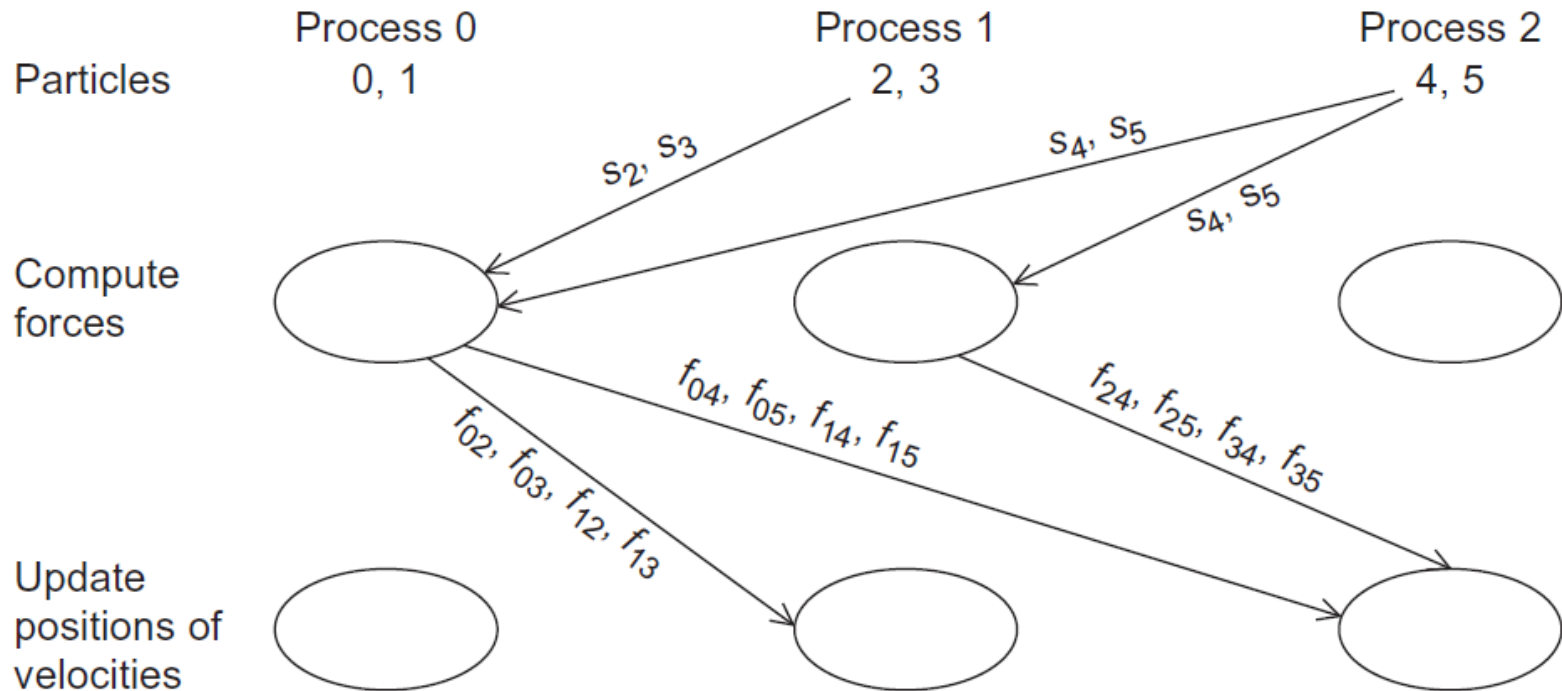
```
Get input data;
for each timestep {
    if (timestep output)
        Print positions and velocities of particles;
    for each local particle loc_q
        Compute total force on loc_q;
    for each local particle loc_q
        Compute position and velocity of loc_q;
    Allgather local positions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



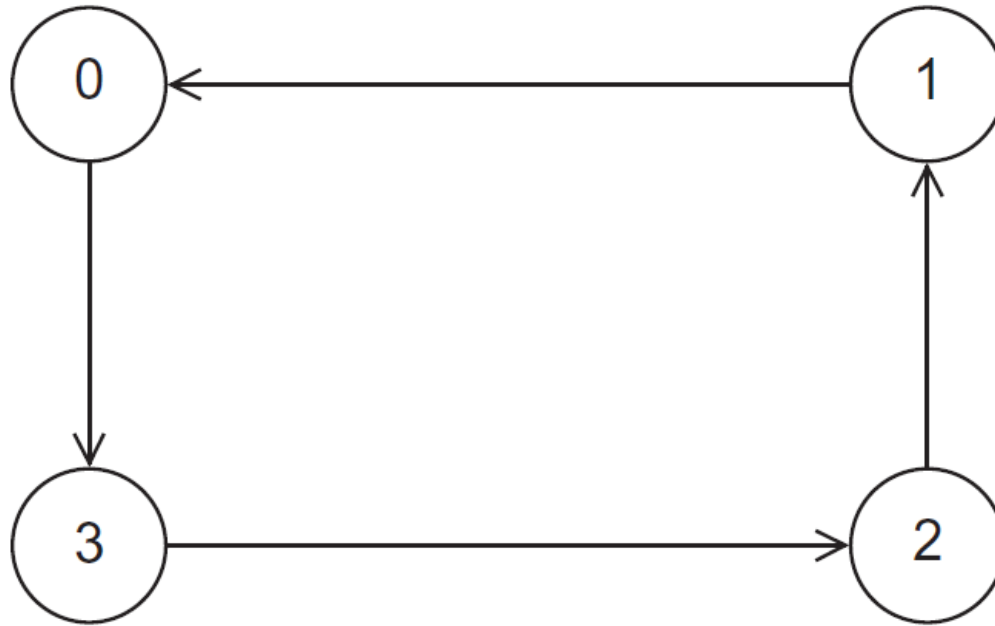
Communication In A Possible MPI Implementation of the N-Body Solver (for a reduced solver)



Complicated!

Unless the implementation were very carefully done,
it would probably be very slow.

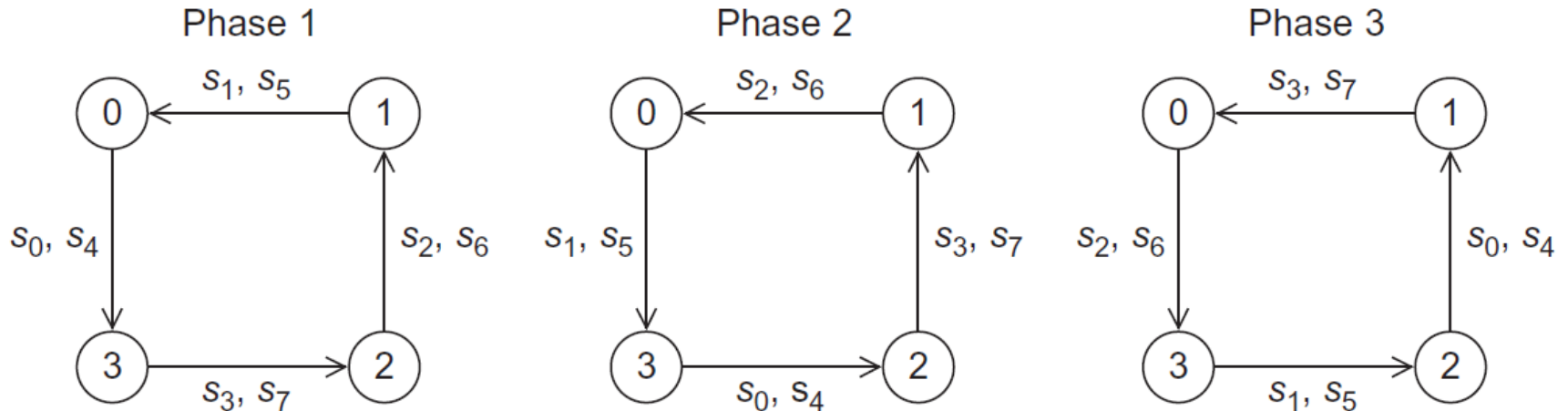
A Ring of Processes



By repeatedly sending and receiving data using the ring, each process has access to positions of all the particles.

Ring Pass of Positions

4 processes, 8 particles, cyclic partition



Phase 1: each process will **send** positions of its assigned particles to its “**lower-ranked**” neighbor and **receive** positions of particles assigned to its “**higher-ranked**” neighbor

Phase 2: each process will **forward** the positions **it received** in the first phase.

Computation of Forces in Ring Pass

In each phase, a process can

(1) **compute inter-particle forces** resulting from interaction between its **assigned particles** and the **particles** whose positions it has **received**;

(2) once an inter-particle force has been computed, the process can **add** the force into **a local array** of forces, and **subtract** the force from **the received array** of forces.

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;
```

Computation of Forces in Ring Pass

2 processes, 4 particles, cyclic partition

Time	Variable	Process 0	Process 1
Start	loc_pos loc_forces tmp_pos tmp_forces	s_0, s_2 $0, 0$ s_0, s_2 $0, 0$	s_1, s_3 $0, 0$ s_1, s_3 $0, 0$
After Comp of Forces	loc_pos loc_forces tmp_pos tmp_forces	s_0, s_2 $f_{02}, 0$ s_0, s_2 $0, -f_{02}$	s_1, s_3 $f_{13}, 0$ s_1, s_3 $0, -f_{13}$
After First Comm	loc_pos loc_forces tmp_pos tmp_forces	s_0, s_2 $f_{02}, 0$ s_1, s_3 $0, -f_{13}$	s_1, s_3 $f_{13}, 0$ s_0, s_2 $0, -f_{02}$
After Comp of Forces	loc_pos loc_forces tmp_pos tmp_forces	s_0, s_2 $f_{01} + f_{02} + f_{03}, f_{23}$ s_1, s_3 $-f_{01}, -f_{03} - f_{13} - f_{23}$	s_1, s_3 $f_{12} + f_{13}, 0$ s_0, s_2 $0, -f_{02} - f_{12}$
After Second Comm	loc_pos loc_forces tmp_pos tmp_forces	s_0, s_2 $f_{01} + f_{02} + f_{03}, f_{23}$ s_0, s_2 $0, -f_{02} - f_{12}$	s_1, s_3 $f_{12} + f_{13}, 0$ s_1, s_3 $-f_{01}, -f_{03} - f_{13} - f_{23}$
After Comp of Forces	loc_pos loc_forces tmp_pos tmp_forces	s_0, s_2 $f_{01} + f_{02} + f_{03}, -f_{02} - f_{12} + f_{23}$ s_0, s_2 $0, -f_{02} - f_{12}$	s_1, s_3 $-f_{01} + f_{12} + f_{13}, -f_{03} - f_{13} - f_{23}$ s_1, s_3 $-f_{01}, -f_{03} - f_{13} - f_{23}$

Final phase:
carry out a
vector sum

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)

Reduced solver is better than basic solver

Run-Times for OpenMP and MPI N-Body Solvers

(in seconds)

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

Basic OpenMP solver is faster than basic MPI solver.

Reduced solver: OpenMP \approx MPI

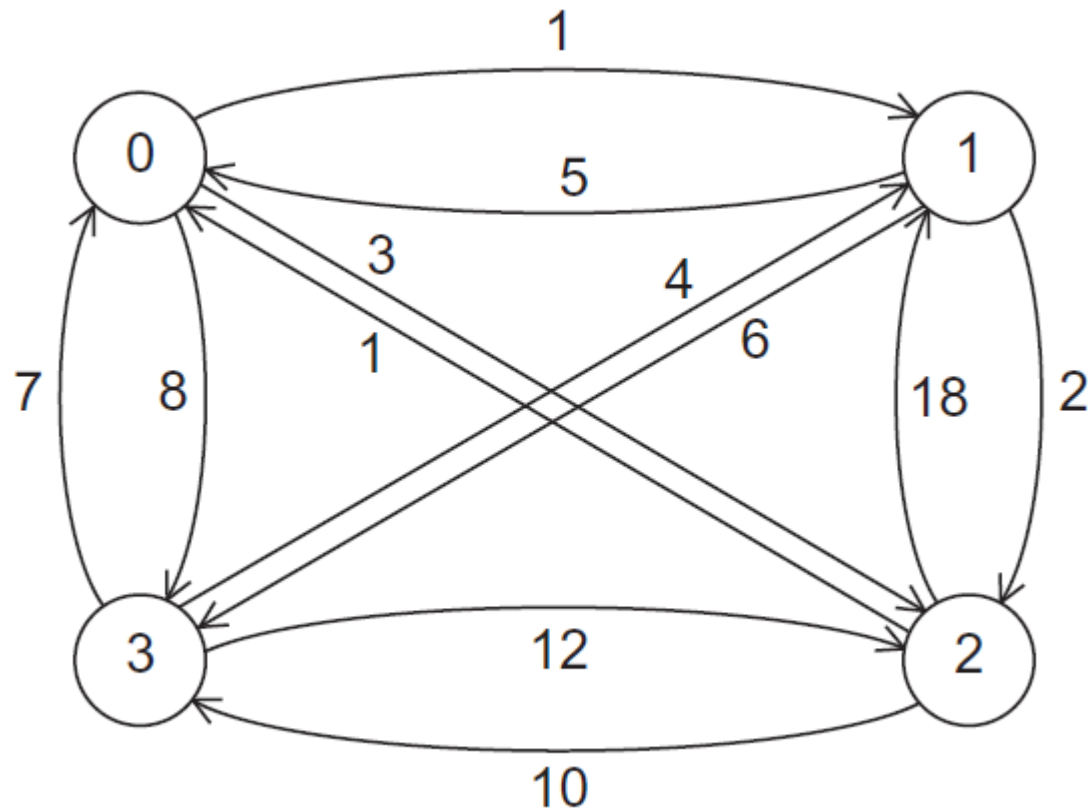
Tree search



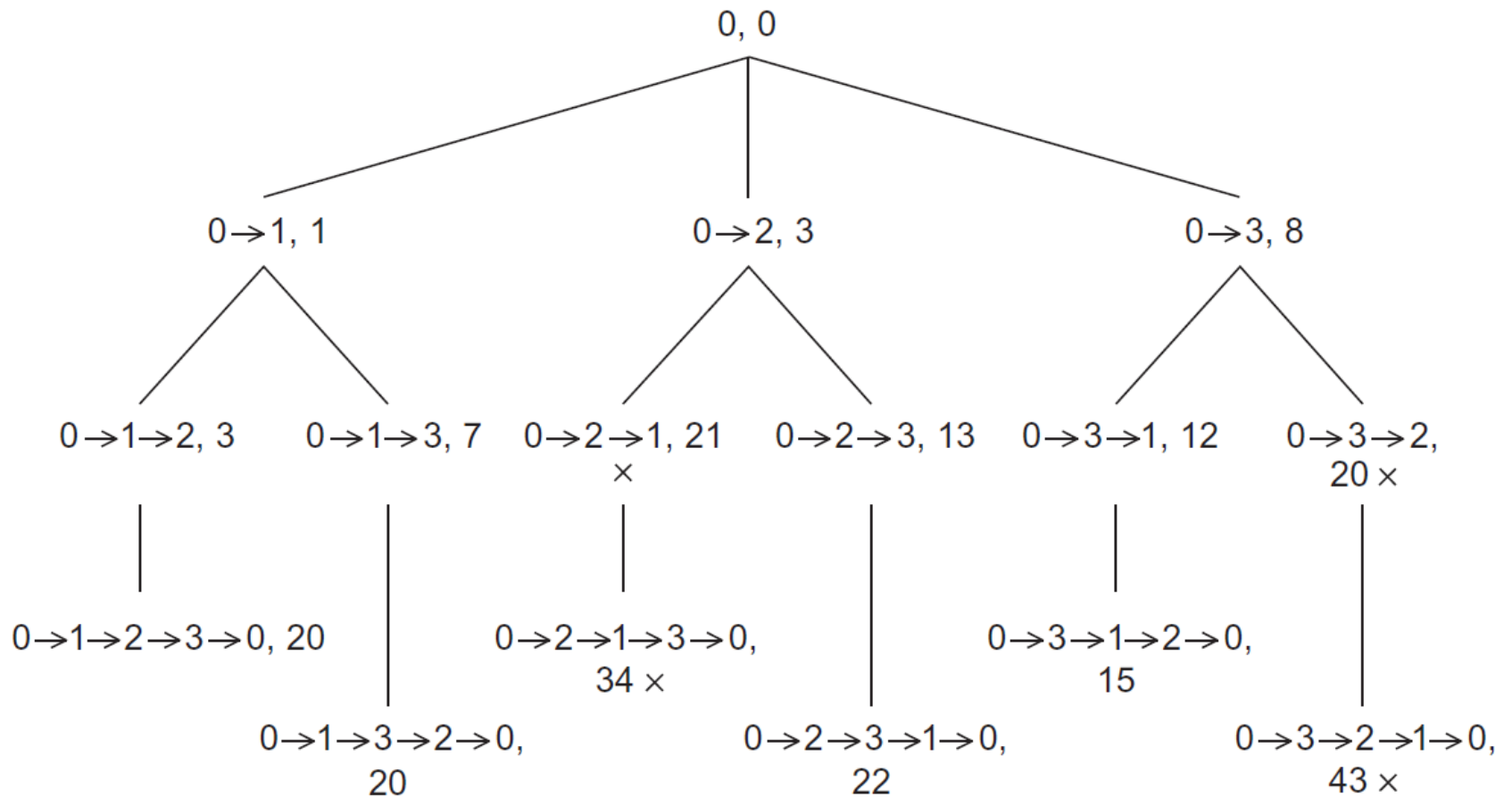
Tree search problem (TSP)

- An **NP-complete** problem.
- **No known** solution to TSP that is better in all cases than **exhaustive search**.
- Ex., the travelling salesperson problem, finding a minimum cost tour.

A Four-City TSP



Search Tree for Four-City TSP



Pseudo-code for a **recursive solution** to TSP using depth-first search

```
void Depth_first_search(tour_t tour) {  
    city_t city;  
  
    if (City_count(tour) == n) {  
        if (Best_tour(tour))  
            Update_best_tour(tour);  
    } else {  
        for each neighboring city  
            if (Feasible(tour, city)) {  
                Add_city(tour, city);  
                Depth_first_search(tour);  
                Remove_last_city(tour);  
            }  
    }  
}  
/* Depth_first_search */
```

To see if there are n cities in the partial tour

To see if the complete tour has a lower cost than the current "best tour"

Replace the current best tour with the tour

To see if city has been visited, and if not, whether the city can possibly lead to a least-cost tour.

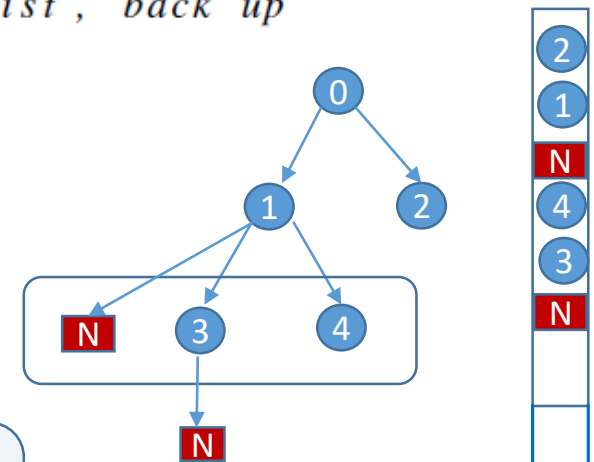
Remove city from the tour, as it shouldn't be included in the tour in subsequent recursive calls.

Pseudo-code for an implementation of a depth-first solution to TSP without recursion

```

for (city = n-1; city >= 1; city--)
    Push(stack, city);
while (!Empty(stack)) {
    city = Pop(stack);
    if (city == NO_CITY) // End of child list, back up
        Remove_last_city(curr_tour);
    else {
        Add_city(curr_tour, city);
        if (City_count(curr_tour) == n) {
            if (Best_tour(curr_tour))
                Update_best_tour(curr_tour);
            Remove_last_city(curr_tour);
        } else {
            Push(stack, NO_CITY);
            for (nbr = n-1; nbr >= 1; nbr--)
                if (Feasible(curr_tour, nbr))
                    Push(stack, nbr);
        }
    }
} // if Feasible */
} // while !Empty */
    
```

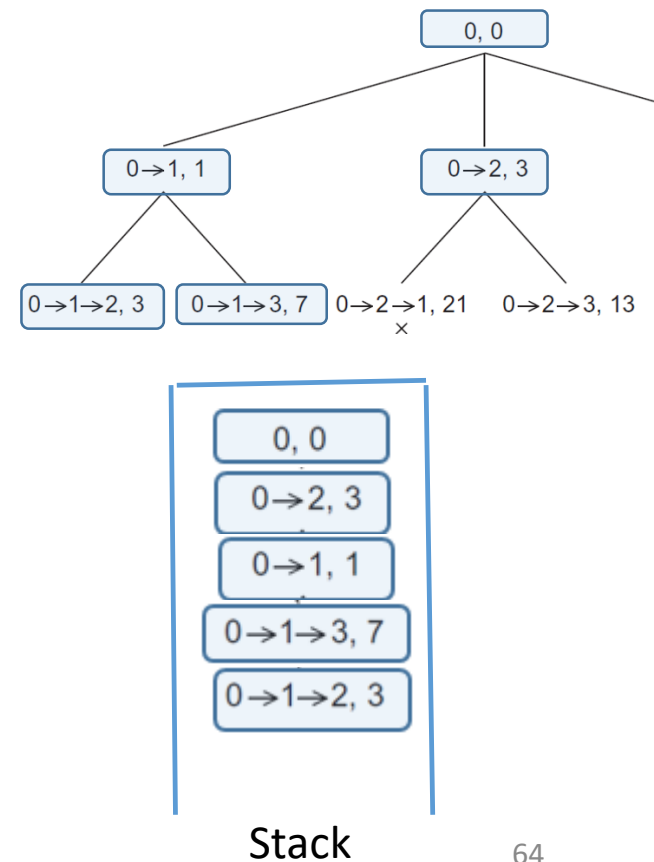
NO_CITY is a marker so that we can tell when we've visited all children of a tree node



Before pushing all children of a node, we push NO_CITY marker

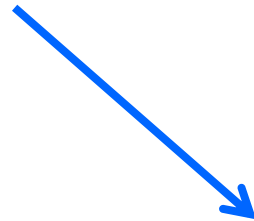
Pseudo-code for a **second** solution to TSP that **doesn't** use recursion

```
Push_copy(stack, tour); // Tour that visits only the hometown
while (!Empty(stack)) {
    curr_tour = Pop(stack);
    if (City_count(curr_tour) == n) {
        if (Best_tour(curr_tour))
            Update_best_tour(curr_tour);
    } else {
        for (nbr = n-1; nbr >= 1; nbr--)
            if (Feasible(curr_tour, nbr)) {
                Add_city(curr_tour, nbr);
                Push_copy(stack, curr_tour);
                Remove_last_city(curr_tour);
            }
    }
    Free_tour(curr_tour);
}
```



Using pre-processor macros

```
/* Find the ith city on the partial tour */  
int Tour_city(tour_t tour, int i) {  
    return tour->cities[i];  
} /* Tour_city */
```



```
/* Find the ith city on the partial tour */  
#define Tour_city(tour, i) (tour->cities[i])
```

Run-Times of the Three Serial Implementations of Tree Search

(in seconds)

Recursive	First Iterative	Second Iterative
30.5	29.2	32.9



- First iterative solution is faster, as it **eliminates** some **overhead** due to **repeated function calls**.
- Second iterative solution is slower because of **repeated copying** of tour data structure.
- The second iterative one is **easy to parallelize**.

Making sure we have the “best tour” (1)

- When a process finishes a tour, it needs to **check** if it has a **better** solution than **recorded so far**.
- The global **Best_tour** function only **reads** the global best cost, so we don't need to tie it up by locking it. There's **no contention** with other **readers**.
- If the process does not have a better solution, then it does **not** attempt an **update**.

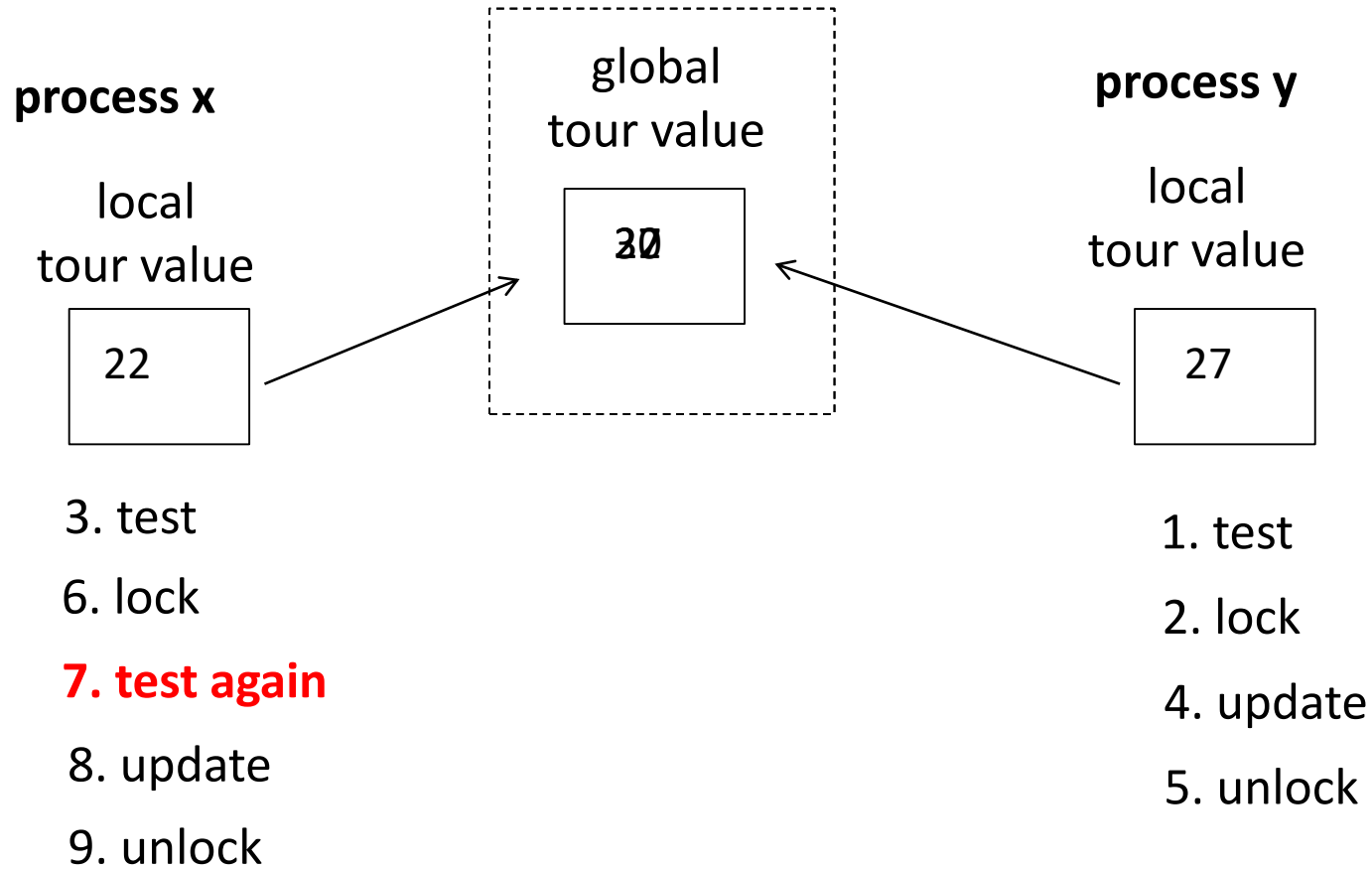
Making sure we have the “best tour” (2)

- If another thread is **updating** while we read, we may see the **old** value **or** the **new** value.
- The **new value is preferable**, but to ensure this would be more **costly** than it is worth.

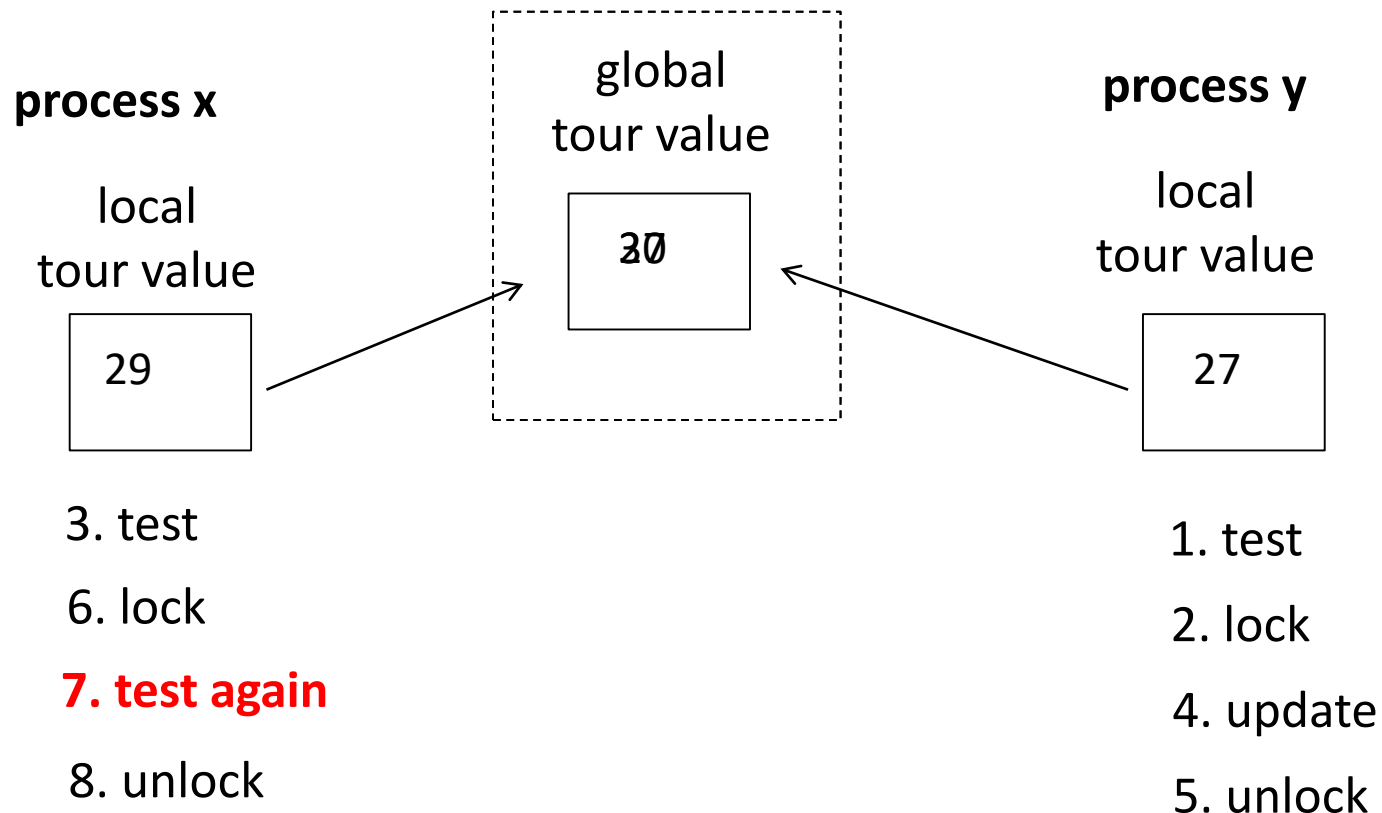
Making sure we have the “best tour” (3)

- In the case where a thread **tests** and **decides** it has a better global solution, we need to ensure two things:
 - 1) That the process **locks** the value with a **mutex**, preventing a race condition.
 - 2) In the possible event that the first check was against an old value **while another process was updating**, we do not put a worse value than the new one that was being written.
- We handle this by **locking**, then **testing again**.

First scenario



Second scenario



Parallelizing the Tree Search Programs Using Pthreads



Static Parallelization of Tree Search using Pthreads

- **Idea:** a single thread use BFS to generate at least `thread_count` partial tours to **distribute** among threads, then **each thread** takes its partial tours and runs iterative tree search on them.
- Four **differences** compared with iterative serial one:
 - The use of `my_stack` instead of `stack`;
 - **Initialization** of the stack;
 - Implementation of `Best_tour func`;
 - Implementation of `Update_best_tour func`.

Pseudo-code for a **Pthreads** implementation of a **statically** parallelized solution to TSP

```
Partition_tree(my_rank, my_stack);
```

```
while (!Empty(my_stack)) {  
    curr_tour = Pop(my_stack);  
    if (City_count(curr_tour) == n) {  
        if (Best_tour(curr_tour)) Update_best_tour(curr_tour);  
    } else {  
        for (city = n-1; city >= 1; city--)  
            if (Feasible(curr_tour, city)) {  
                Add_city(curr_tour, city);  
                Push_copy(my_stack, curr_tour);  
                Remove_last_city(curr_tour)  
            }  
    }  
    Free_tour(curr_tour);  
}
```

Dynamic Parallelization of Tree Search using Pthreads

Basic Idea:

- (1) when a thread **runs out of work**, it **waits to see** if another thread can provide more work.
- (2) if a thread still **has work** and finds that there is at least one thread without work, it can **split its stack** and provide work for one of the threads.

Note: using `pthread_cond_wait`, `pthread_cond_signal`, `pthread_cond_broadcast` for implementation.

Dynamic Parallelization of Tree Search Using Pthreads

- Code executed by a thread before it splits:
 - It checks that it has **at least two tours** in its stack.
 - It checks that there are **threads waiting**.
 - It checks whether the **new_stack** variable is **NULL**.
- Termination issues.
 - Only when **all the threads** have run out of work.
 - **Terminated function** is used instead of **Empty(stack)**

Pseudo-Code for Pthreads **Terminated Function** (1)

```
if (my_stack_size >= 2 && threads_in_cond_wait > 0 &&
    new_stack == NULL) {
    lock term_mutex;
    if (threads_in_cond_wait > 0 && new_stack == NULL) {
        Split my_stack creating new_stack;
        pthread_cond_signal(&term_cond_var);
    }
    unlock term_mutex;
    return 0; /* Terminated = False; don't quit */
} else if (!Empty(my_stack)) { /* Stack not empty, keep working */
    return 0; /* Terminated = false; don't quit */
} else { /* My stack is empty */
    lock term_mutex;
    if (threads_in_cond_wait == thread_count - 1) { /* Last thread */
                                                /* running */
        threads_in_cond_wait++;
        pthread_cond_broadcast(&term_cond_var);
        unlock term_mutex;
        return 1; /* Terminated = true; quit */
    }
```

Pseudo-Code for Pthreads Terminated Function (2)

```
    } else { /* Other threads still working, wait for work */  
        threads_in_cond_wait++;  
        while (pthread_cond_wait(&term_cond_var, &term_mutex) != 0);  
        /* We've been awakened */  
        if (threads_in_cond_wait < thread_count) { /* We got work */  
            my_stack = new_stack;  
            new_stack = NULL;  
            threads_in_cond_wait--;  
            unlock term_mutex;  
            return 0; /* Terminated = false */  
        } else { /* All threads done */  
            unlock term_mutex;  
            return 1; /* Terminated = true; quit */  
        }  
    } /* else wait for work */  
} /* else my_stack is empty */
```

Grouping the **termination variables**

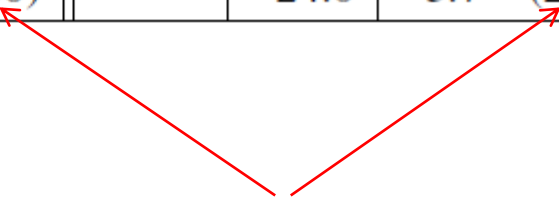
```
typedef struct {  
    my_stack_t new_stack;  
    int threads_in_cond_wait;  
    pthread_cond_t term_cond_var;  
    pthread_mutex_t term_mutex;  
} term_struct;  
typedef term_struct* term_t;  
  
term_t term; // global variable
```

Run-times of Pthreads tree search programs

15-city problems

Threads	First Problem				Second Problem			
	Serial	Static	Dynamic		Serial	Static	Dynamic	
1	32.9	32.7	34.7	(0)	26.0	25.8	27.5	(0)
2		27.9	28.9	(7)		25.8	19.2	(6)
4		25.7	25.9	(47)		25.8	9.3	(49)
8		23.8	22.4	(180)		24.0	5.7	(256)

(in seconds)



numbers of times
stacks were split

Parallelizing the Tree Search Programs Using OpenMP

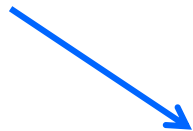


Parallelizing the Tree Search Programs Using OpenMP

- **Same basic issues** implementing the static and dynamic parallel tree search programs as Pthreads.
- A few small changes can be noted.

Pthreads

```
if (my_rank == whatever)
```



```
# pragma omp single
```

OpenMP

OpenMP emulated condition wait

```
/* Global vars */
```

```
int awakened_thread = -1;  
work_remains = 1; /* true */
```

```
. . .
```

```
omp_unset_lock(&term_lock);
```

```
while (awakened_thread != my_rank && work_remains);
```

```
omp_set_lock(&term_lock);
```

Busy-waiting !

Performance of OpenMP and Pthreads implementations of tree search

Th	First Problem				Second Problem			
	Static		Dynamic		Static		Dynamic	
	OMP	Pth	OMP	Pth	OMP	Pth	OMP	Pth
1	32.5	32.7	33.7 (0)	34.7 (0)	25.6	25.8	26.6 (0)	27.5 (0)
2	27.7	27.9	28.0 (6)	28.9 (7)	25.6	25.8	18.8 (9)	19.2 (6)
4	25.4	25.7	33.1 (75)	25.9 (47)	25.6	25.8	9.8 (52)	9.3 (49)
8	28.0	23.8	19.2 (134)	22.4 (180)	23.8	24.0	6.3 (163)	5.7 (256)

(in seconds)

For most parts, OpenMP implementation is comparable to Pthreads implementation.

Implementation of Tree Search Using MPI and Static Partitioning



Implementation using MPI and static partitioning

- Read the adjacency matrix on **process 0**, and **broadcast** it to all the processes;
- **Partitioning** the tree
- Checking and updating the **best tour**
- After the search has **terminated**, making sure that process 0 has a copy of the best tour for output

Partitioning the tree

- **MPI_Scatter**: **cannot** be used, as # of initial partial tours **may not** be evenly **divisible** by comm_sz.
- **MPI_Scatterv**: a variant of MPI_Scatter, which can send different # of objects to different processes.

```
int MPI_Scatterv(  
    void*      sendbuf      /* in */,  
    int*       sendcounts   /* in */,  
    int*       displacements /* in */,  
    MPI_Datatype sendtype   /* in */,  
    void*      recvbuf      /* out */,  
    int        recvcnt      /* in */,  
    MPI_Datatype recvttype  /* in */,  
    int        root         /* in */,  
    MPI_Comm    comm        /* in */)

```

Data sent to process q will begin in location (sendtype = MPI_INT)
sendbuf + displacement[q]

Gathering a different number of objects from each process in the communicator

- **MPI_Gatherv**: a variant of MPI_Gather, which can gather different # of objects from different processes.

```
int MPI_Gatherv(  
    void*      sendbuf          /* in */,  
    int        sendcount        /* in */,  
    MPI_Datatype sendtype       /* in */,  
    void*      recvbuf          /* out */,  
    int*        recvcounts       /* in */,  
    int*        displacements    /* in */,  
    MPI_Datatype recvtype       /* in */,  
    int         root             /* in */,  
    MPI_Comm    comm            /* in */)
```


Maintaining the best tour

- When a process finds a new best tour, it should send it to other processes.
- **MPI_Bcast** cannot be used, as it is **blocking**.
 - Every process in the communicator must call MPI_Bcast.
- New tour should be sent in a way that the sender won't block indefinitely.

Maintaining the best tour

- The simplest way is to use **MPI_send** to send the **new best cost** to **all the other** processes.
 - Use a special tag **NEW_COST_TAG**.
- The destination processes can **periodically check** for the **arrival** of new best tour costs.
- **Problem:** we **cannot use MPI_Recv** as the process will **block** until a matching message arrives.

Maintaining the best tour

- **MPI_Iprobe:** checking to see if a message is **available**
 - It doesn't actually try to receive a message

```
int MPI_Iprobe(  
    int          source      /* in */,  
    int          tag         /* in */,  
    MPI_Comm     comm        /* in */,  
    int*         msg_avail_p /* out */,  
    MPI_Status*  status_p    /* out */);
```



If such a msg is available, *msg_avail_p is assigned TRUE

Status_p->MPI_SOURCE will be assigned the rank of message source

MPI code to check for new best tour costs

```
MPI_Iprobe(MPI_ANY_SOURCE, NEW_COST_TAG, comm,
           &msg_avail, &status);
while(msg_avail)
{
    MPI_Recv(&received_cost, 1, MPI_INT, status.MPI_SOURCE,
            NEW_COST_TAG, comm, MPI_STATUS_IGNORE );
    if (received_cost < best_tour_cost)
        best_tour_cost = received_cost;
    MPI_Iprobe(MPI_ANY_SOURCE, NEW_COST_TAG, comm,
              &msg_avail, &status);
}
```

Modes and Buffered Sends - 1

- MPI provides four modes for sends.
 - **Standard:** MPI_Send
 - **Synchronous:** MPI_Ssend
 - **Ready:** MPI_Rsend
 - **Buffered:** MPI_Bsend

Modes and Buffered Sends - 2

- **Standard: MPI_Send**

- MPI implementation decides whether to copy the message content into its own storage, or to block until a matching receive is posted.

- **Synchronous: MPI_Ssend**

- The send will **block** until a matching receive is posted

Modes and Buffered Sends - 3

- **Ready: MPI_Rsend**

- The send is erroneous unless a matching Receive is posted *before* the Send is started.

- **Buffered: MPI_Bsend**

- MPI implementation must copy the message into **local temporary storage** if a matching Receive hasn't been posted.
- The temporary storage must be **provided** by the **user program**, not MPI implementation.



Printing the best tour

- Have each process store its **local best tour**
- After completing searches, call **MPI_Allreduce**
- The process with the global best tour can then send it to **process 0 for output**

```
struct {  
    int cost;  
    int rank;  
} loc_data, global_data;
```

```
loc_data.cost = Tour_cost(loc_best_tour);  
loc_data.rank = my_rank;
```

```
MPI_Allreduce(&loc_data, &global_data, 1, MPI_2INT, MPI_MINLOC, comm);  
if (global_data.rank == 0) return; /* 0 already has the best tour */  
if (my_rank == 0)  
    Receive best tour from process global_data.rank;  
else if (my_rank == global_data.rank)  
    Send best tour to process 0;
```

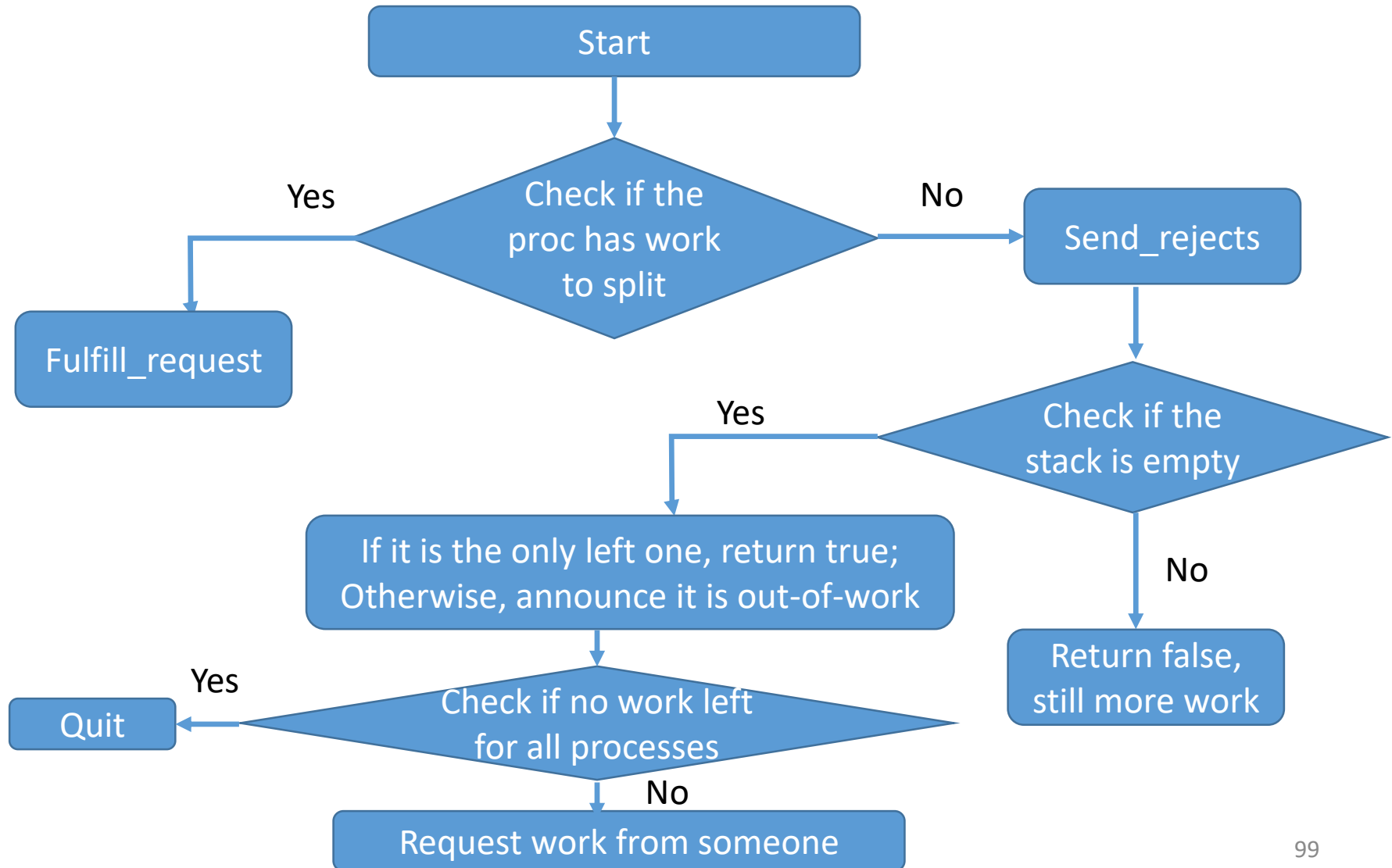

Implementation of Tree Search Using MPI and Dynamic Partitioning



Using MPI and Dynamic Partitioning

- **Key difference** with OpenMP/threads:
 - There is **no central repository** of information on which processes are **waiting for work**.
 - Rather than simply going into a busy-wait or termination, **a process** that has run out of work should **send a request for work** to another process.

Terminated function for a Dynamically Partitioned TSP solver that Uses MPI



Terminated Function for a Dynamically Partitioned TSP solver with MPI (1)

```
if (My_avail_tour_count(my_stack) >= 2) {  
    Fulfill_request(my_stack);  
    return false; /* Still more work */  
} else { /* At most 1 available tour */
```

the proc has work to split

the proc has no more
work to split

```
    Send_rejects(); /* Tell everyone who's requested */  
                    /* work that I have none */  
}
```

```
if (!Empty_stack(my_stack)) {  
    return false; /* Still more work */  
} else { /* Empty stack */
```

the proc has work to do,
cannot terminate

```
    if (comm_sz == 1) return true;
```

Only one proc is left, and
it has no work, quit

```
    Out_of_work();  
    work_request_sent = false;
```

```
    while (1) {
```

```
        Clear_msgs(); /* Messages unrelated to work, termination */
```

```
        if (No_work_left()) {  
            return true; /* No work left. Quit */  
        }
```

When stack is empty,
announce it is out-of-work

If all procs have no work left, quit

Terminated Function for a Dynamically Partitioned TSP solver with MPI (2)

Send work request to other proc

```
} else if (!work_request_sent) {  
    Send_work_request(); /* Request work from someone */  
    work_request_sent = true;  
}  
else {  
    Check_for_work(&work_request_sent, &work_avail);  
    if (work_avail) {  
        Receive_work(my_stack);  
        return false;  
    }  
}
```

```
    }  
} /* while */  
} /* Empty stack */  
} /* At most 1 available tour */
```

Check for work, if there is work available, receive work from the remote proc.

Splitting the stack and data packing

- **MPI_Pack**: Packing data into a buffer of contiguous memory
- **Split_stack** packs the contents of new stack into contiguous memory and send the block of contiguous memory to the receiver

```
int MPI_Pack(  
    void*          data_to_be_packed    /* in      */,  
    int            to_be_packed_count   /* in      */,  
    MPI_Datatype   datatype              /* in      */,  
    void*          contig_buf           /* out     */,  
    int            contig_buf_size       /* in      */,  
    int*           position_p            /* in/out  */,  
    MPI_Comm       comm                 /* in      */)
```

```
int MPI_Unpack(  
    void*          contig_buf           /* in      */,  
    int            contig_buf_size       /* in      */,  
    int*           position_p            /* in/out  */,  
    void*          unpacked_data         /* out     */,  
    int            unpack_count          /* in      */,  
    MPI_Datatype   datatype              /* in      */,  
    MPI_Comm       comm                 /* in      */) 102
```



Distributed Termination Detection

- *Out_of_work()* and *No_work_left()* implements the termination detection algorithm.
- The termination detection algorithm used in **shared-memory** programs will **have problems** in **MPI**.
- Suppose each proc stores a variable *oow* to indicate the number of processes that are out of work
 - **Initially, *oow* = 0**
 - Each time a process runs out of work (or receive work), it sends a message to other processes, so that **others will update** their copies of *oow*.

Table 6.10 Termination Events that Result in an Error

Time	Process 0	Process 1	Process 2
0	Out of Work Notify 1, 2 oow = 1	Out of Work Notify 0, 2 oow = 1	Working oow = 0
1	Send request to 1 oow = 1	Send Request to 2 oow = 1	Recv notify fr 1 oow = 1
2	oow = 1	Recv notify fr 0 oow = 2	Recv request fr 1 oow = 1
3	oow = 1	oow = 2	Send work to 1 oow = 0
4	oow = 1	Recv work fr 2 oow = 1	Recv notify fr 0 oow = 1
5	oow = 1	Notify 0 oow = 1	Working oow = 1
6	oow = 1	Recv request fr 0 oow = 1	Out of work Notify 0, 1 oow = 2
7	Recv notify fr 2 oow = 2	Send work to 0 oow = 0	Send request to 1 oow = 2
8	Recv 1st notify fr 1 oow = 3	Recv notify fr 2 oow = 1	oow = 2
9	Quit	Recv request fr 2 oow = 1	oow = 2

The **error** here is that the work sent from process 1 to process 0 is **lost**

The reason is that proc 0 receives notification that **proc 2 is out of work** before it receives notification that **proc 1 has received work**.

A simple distributed termination detection algorithm

- **Idea:** keeping track of a **quantity** that is **conserved** and can be measured precisely (e.g., energy)
- **Algorithm:**
 - Initially, each process has **1 unit** of energy
 - When a process runs out of work, it **sends its energy** to **process 0**.
 - When a process fulfills a request for work, it **divides** its energy in half, **keeping half** for itself, and sending half to another
 - The program should **terminate** when **process 0** receives a total energy of **comm_sz units**

Performance of MPI and Pthreads implementations of tree search

(in seconds)

Th/Pr	First Problem				Second Problem			
	Static		Dynamic		Static		Dynamic	
	Pth	MPI	Pth	MPI	Pth	MPI	Pth	MPI
1	35.8	40.9	41.9 (0)	56.5 (0)	27.4	31.5	32.3 (0)	43.8 (0)
2	29.9	34.9	34.3 (9)	55.6 (5)	27.4	31.5	22.0 (8)	37.4 (9)
4	27.2	31.7	30.2 (55)	52.6 (85)	27.4	31.5	10.7 (44)	21.8 (76)
8		35.7		45.5 (165)		35.7		16.5 (161)
16		20.1		10.5 (441)		17.8		0.1 (173)

- **Pthreads outperforms MPI** implementation for **small** shared-memory systems.
- For **large** problems, **MPI** program is much **more scalable** and can provide **better** performance.

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.