High-Performance Computing
– Case Study: N-Body Simulations –

Christian Plessl

High-Performance IT Systems Group
Paderborn University, Germany

version 1.2.0 2018-01-08

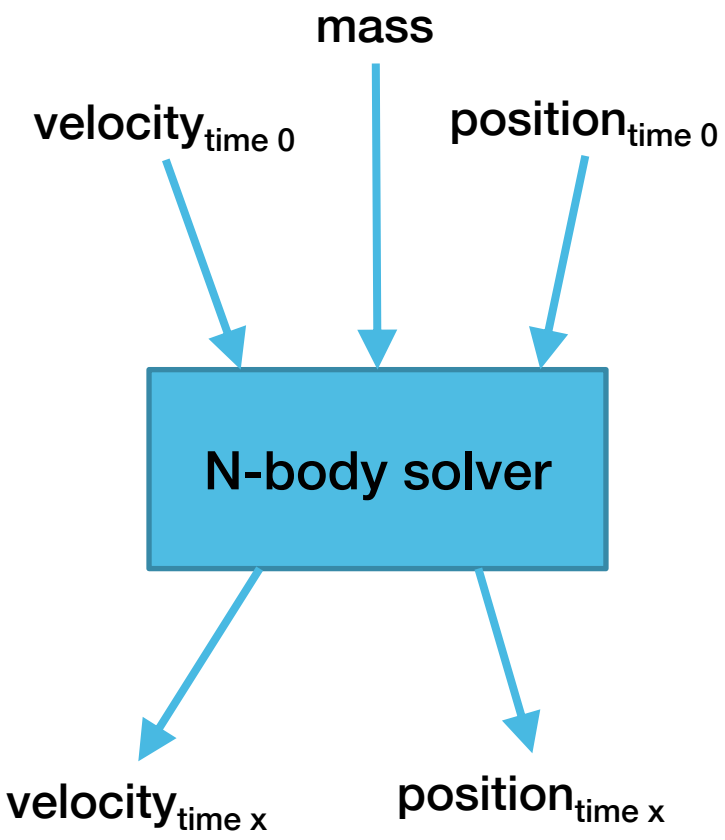
Paderborn
Center for
Parallel
Computing

Outline

- Introduction N-Body Problem
- Implementation of parallel N-Body Solvers
  - shared memory systems
  - distributed memory systems

N-Body Problems

- Compute positions and velocities of a collection of interacting particles over a period of time
  - many important use cases
- Astrophysics
  - particles: stars, planets, ...
  - forces: gravitation
  - applications: compute formation of galaxies
- Molecular dynamics
  - particles: atoms, molecules, ...
  - forces: van der Waals, electrostatic, ...
  - applications: material science, drug discovery
- N-body solvers compute solution to 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

$$F = \frac{G m_q m_k}{r^2}$$

gravitational force (scalar) between particles q and k with masses m_q and m_k

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

gravitational force (vector) between moving particles q and k
with positions $s_q(t)$ and $s_k(t)$

Gravitational Force between Masses

- Consider the interaction of all particles with a fixed particle q
  - summation forces exerted by all other particle k=0..n-1

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

- Newton's second law of motion

$$F_q(\mathrm{t}) = m_q a(t) = m_q s_q''(t)$$

- Applied to all particles

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

Basic Idea for N-Body Solver

- Goal: determine position and velocity at discrete time steps

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

- Pseudo code

```
1     Get input data;
2     for each timestep {
3        if (timestep output) Print positions and velocities of
             particles;
4        for each particle q
5           Compute total force on q;
6        for each particle q
7           Compute position and velocity of q;
8     }
9     Print positions and velocities of particles;
```

Computation of the Forces

- Data structures
  - `pos[]` array containing the positions of the particles
  - `forces[]` array for summing forces exerted on each particle in a time step

- Direct computation

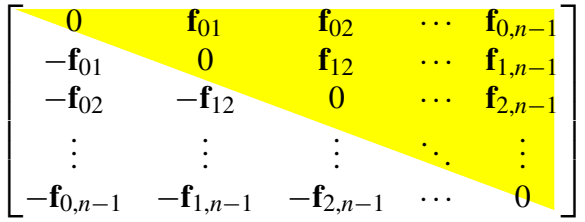
```
for each particle q {
   for each particle k != q {
      x_diff = pos[q][X] – pos[k][X];
      y_diff = pos[q][Y] – pos[k][Y];
      dist = sqrt(x_diff*x_diff + y_diff*y_diff);
      dist_cubed = dist*dist*dist;
      forces[q][X] –= G*masses[q]*masses[k]/dist_cubed * x_diff;
      forces[q][Y]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

Computation of the Forces (2)

- Direct (naïve) computation is wasteful
  - actio = reactio i.e. $f_{qk} = -f_{kq}$
- Individual forces shown as array

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

Reduced Algorithm for Computing 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];
   }
}
```

compute upper triangle force matrix only

Solving the Differential Equation

- We compute forces, but we are interested in positions and velocities of particles
- Use force to compute acceleration, velocity and position with Newton's law of motion

$$F_q(\mathrm{t}) = m_q a(t) = m_q s_q''(t)$$

- We don't don't work with an analytic representation here, thus we numerically solve this ordinary differential equation
- Euler method
  - there are many methods for numerically solving differential equations
  - we will use the Euler method, the most basic method

Leonhard Euler (1707-1783)

image: Wikipedia

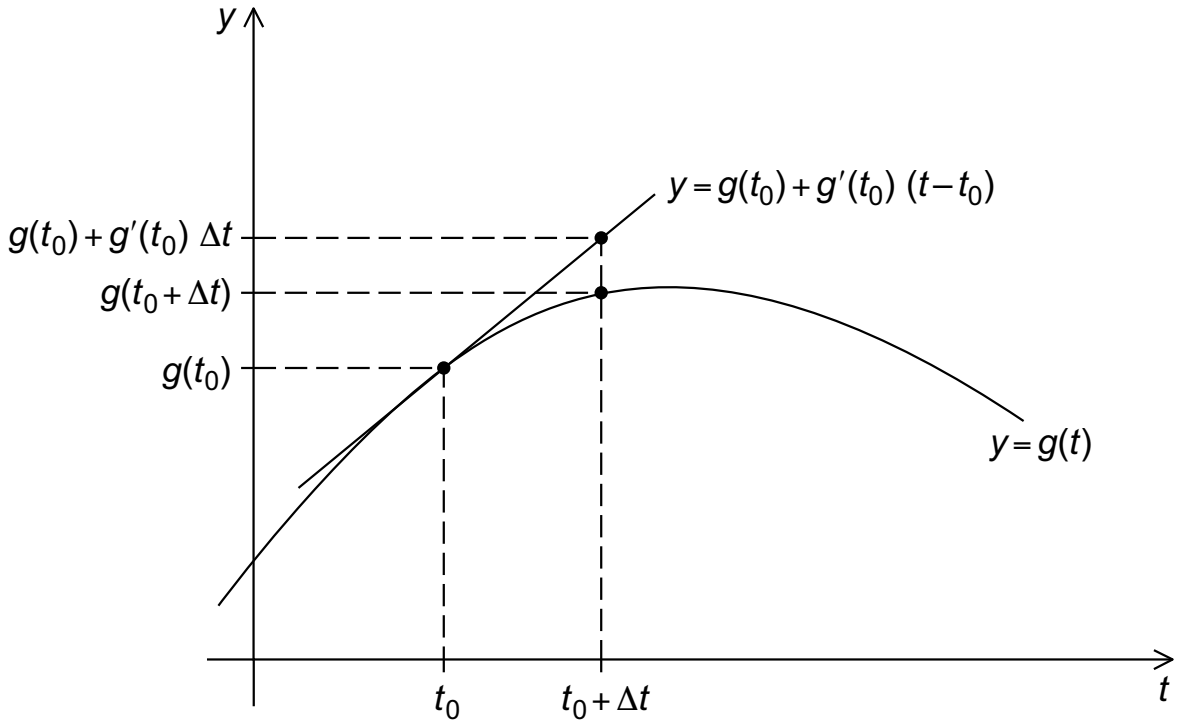
Euler Method (1)

- Basic Idea: Approximate a function with a tangent
- Assume we have an unknown function g for which we know
  1. the value $g(t_0)$ at time t_0 and
  2. the derivative $g'(t_0)$ of the function at time t_0
- Then we can estimate the value of g at time g(t)

$$y = g(t_0) + g'(t_0)(t - t_0).$$

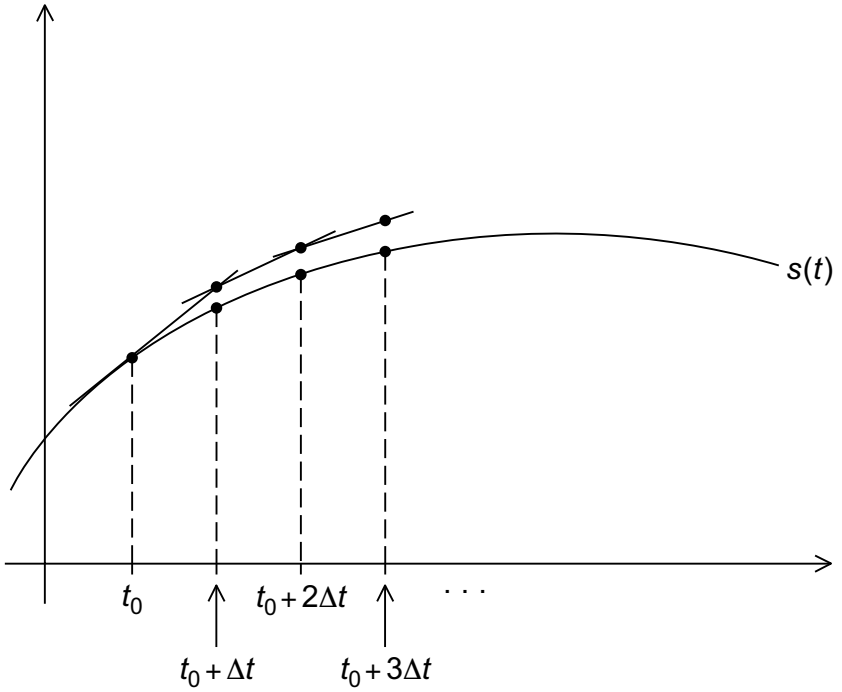
hence:

$$g(t + \Delta t) \approx g(t_0) + g'(t_0)(t + \Delta t - t) = g(t_0) + \Delta t g'(t_0).$$



Euler Method (2)

- The estimate will have an error, but if the error is small we can repeat this scheme to compute function s(t)



- Hence, we can complete our pseudo code with the computation of the positions and velocities

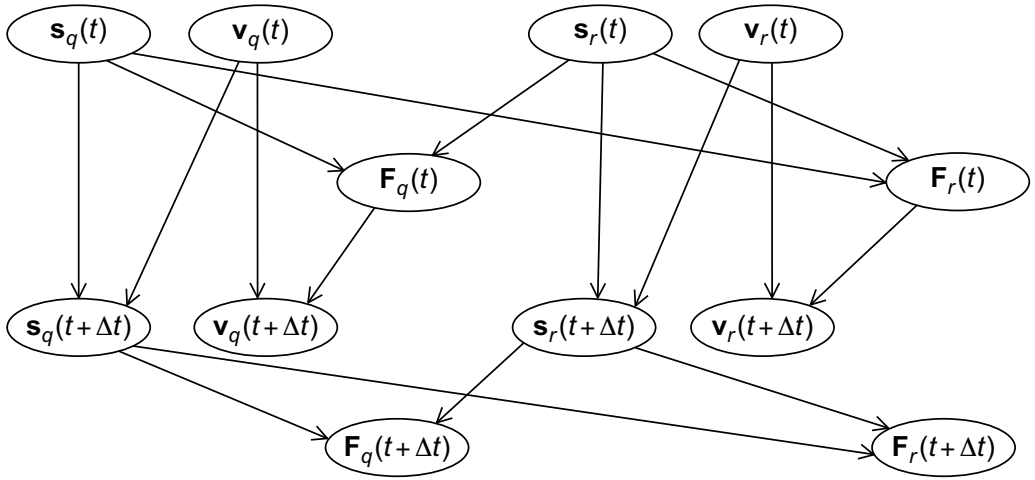
```
pos[q][X] += delta_t*vel[q][X];
pos[q][Y] += delta_t*vel[q][Y];
vel[q][X] += delta_t/masses[q]*forces[q][X];
vel[q][Y] += delta_t/masses[q]*forces[q][Y];
```

velocities from previous time step

numerical integration

Parallelizing the N-Body Solvers

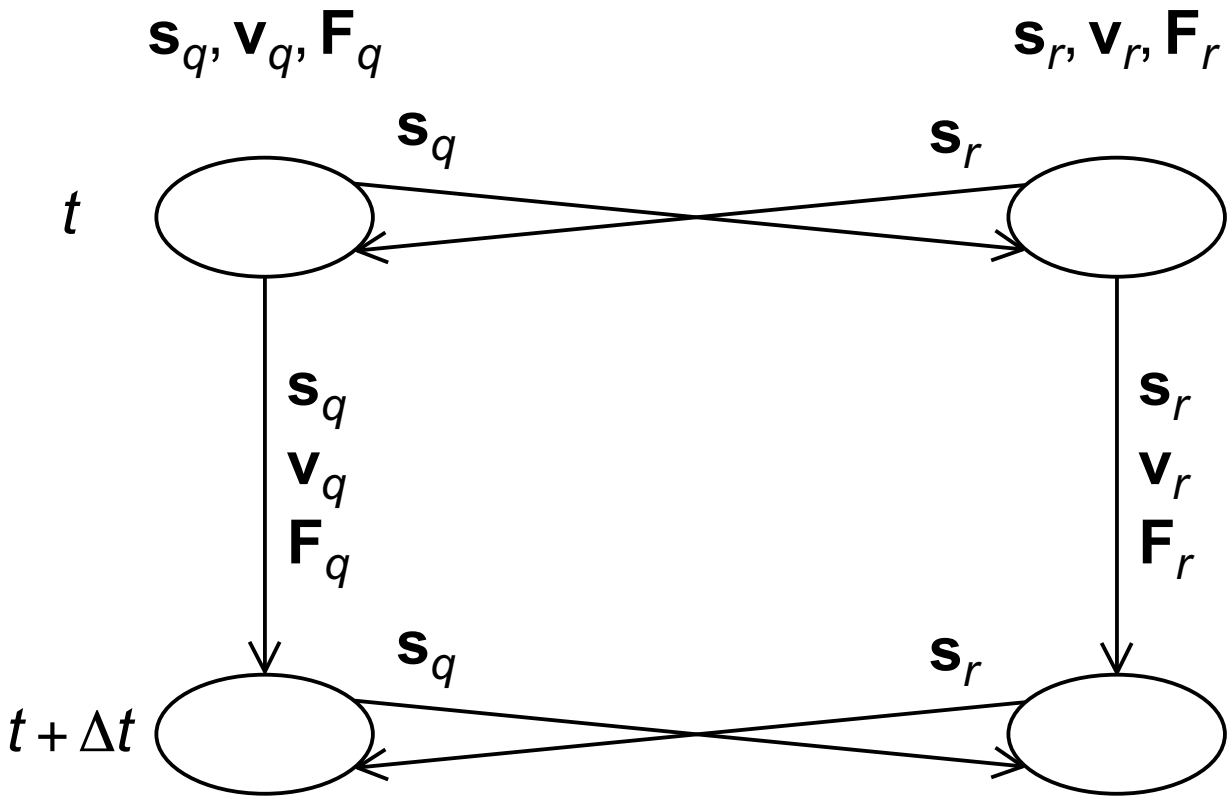
- Apply Foster's methodology
  - initially, we want a lot of tasks
  - tasks: computations of the positions, the velocities, and the total forces at each time step
- N-Body problems have abundant parallelism
  - $O(n^2)$ forces that can be computed independently



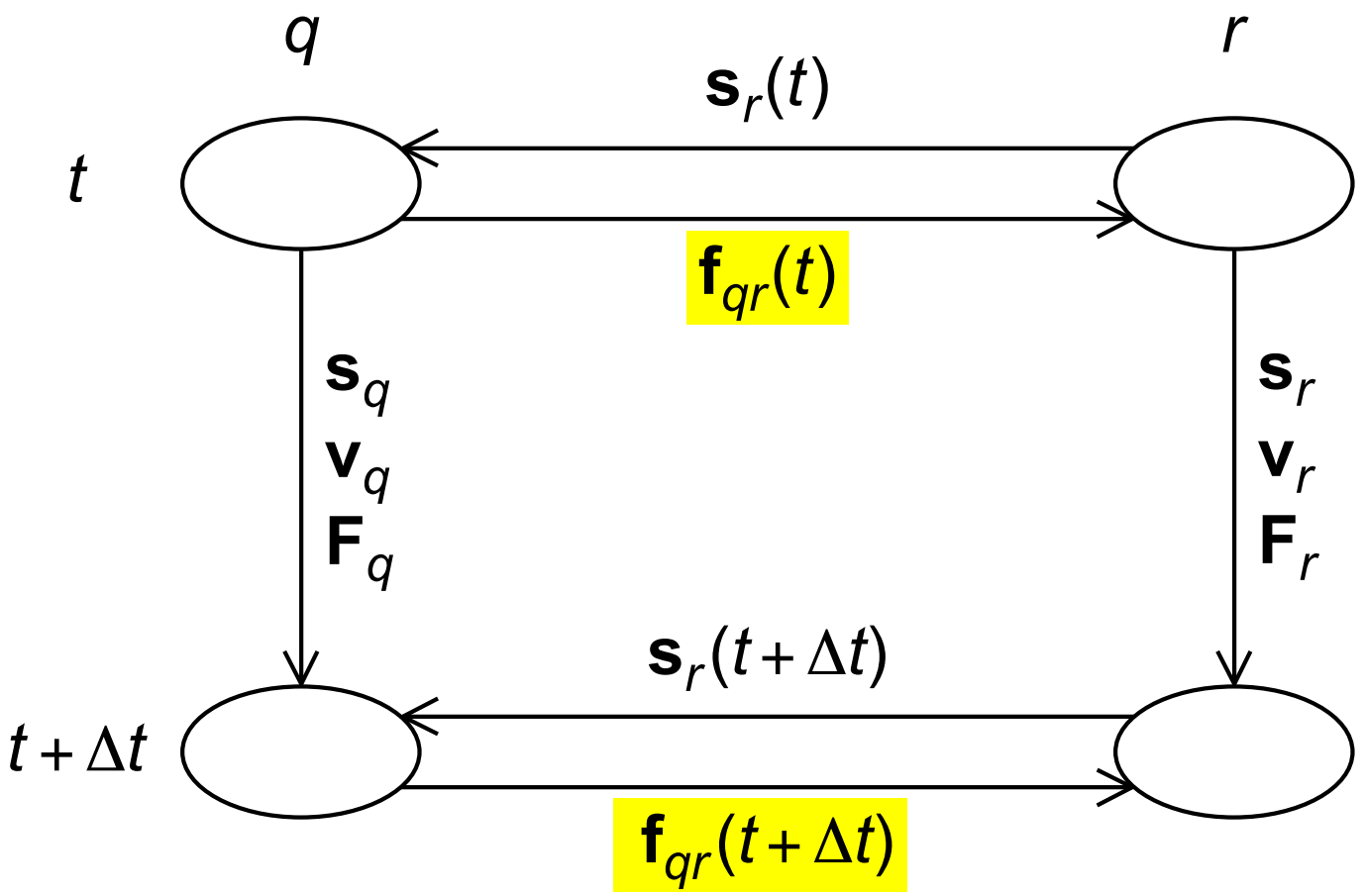
communication between tasks
(forces between particles q and r)

Task Agglomeration in Basic N-Body Solver

- Most communication occurs only between tasks concerning the same particle, simplify structure by agglomerating tasks for same time step and particle



Task Agglomeration in Reduced N-Body Solver



forces are computed only once: hence, task q sends f_{qr} to task r instead of its position

Mapping Computations to Cores

- Last step of Foster's method
  - the algorithm offers plenty of parallelism
  - typically the number of particles is very high (orders of magnitude higher the #cores)

```
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

- Considerations
  - Euler methods must know $s_q(t)$, $v_q(t)$ and $a_q(t)$ to estimate $s_q(t+\Delta t)$ and $v_q(t+\Delta t)$, hence assigning particles to same core in each time step reduces need for communication
  - Assigning each core the same number of particles works for basic solver but leads to a load imbalance on reduced solver

First Attempt for OpenMP Parallelization

```
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;
}
```

Are there 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
  - iterations of outer loop (for each particle q) are partitioned among the threads, hence, only one thread ever writes to `forces[q]` array for a given particle q
  - shared arrays `pos` and `masses` are only read
  - the other variables hold only temporary values and can have private scope

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 either
  - arrays `pos`, `vel`, `forces` are accessed only by a single thread for any particle q
  - scalar `delta_t` is only read

Reduce Forking and Joining of Threads

```
#  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;
   }
```

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

ensure that only a single thread will print all the positions and velocities

Parallelizing the Reduced Solver w/ 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;
   }
```

- Consideration
  - does this code have any race conditions?
  - is the computational load balanced between threads?

Race Condition in Reduced Solver

- There is a race condition because writes to the forces array are not restricted to particle q

- Example: 4 particles, 2 threads, block partitioning
  - $F_3 = -f_{03} - f_{13} - f_{23}$
  - thread 0 computes f_{03} and f_{13}
  - thread 1 computes f_{23}
  - hence: updates to F_3 create a race condition

```
#  pragma omp for /* Can be faster than memset */
   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;

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

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];
   }
```

- Critical section with `#pragma omp critical` has severe drawbacks
  - access to forces arrays is effectively serialized
  - using a named critical section (one per thread) doesn't help either, because OpenMP supports only statically named critical sections

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]);
```

- Avoid global mutex on forces array, use fine-grained lock
  - OpenMP provides a library functions for locking
  - use one lock for each particle
- Performs much better than global lock but still very high overheads
  - system call for every lock
- Idea for improvement
  - use private forces array per thread, do summation later

First Phase for Reduced Alg. (Block Partitioning)

- Block partitioning leads to very poor load balancing

```
for each particle q {
   for each particle k > q {
       compute force f_qk
   }
}
```

thread	responsible for particles	forces computed
0	0 1	f01, f02, f03, f04, f05 f12, f13, f14, f15
1	2 3	f23, f24, f25 f34, f35
2	4 5	f45 –

First Phase for Reduced Alg. (Cyclic Partitioning)

- Cyclic partitioning improves load balancing

thread	responsible for particles	forces computed
0	0 3	f01, f02, f03, f04, f05 f34, f35
1	1 4	f12, f13, f14, f15 f45
2	2 5	f23, f24, f25

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];
      }
   }
```

- Store forces into thread-local array `loc_forces` (no race conditions)
- Aggregate forces in Phase II

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];
      }
   }
```

- Each thread adds the forces computed by all the threads for its assigned particles
- Ensure we didn't introduce new race conditions
  - phase 1: all writes only to thread-private arrays → OK
  - phase 2: threads only write to global forces array for their assigned particles → OK
  - implied barrier guarantees that phase 2 starts only after completion of phase 1 → OK

Parallelizing the Solvers Using Pthreads

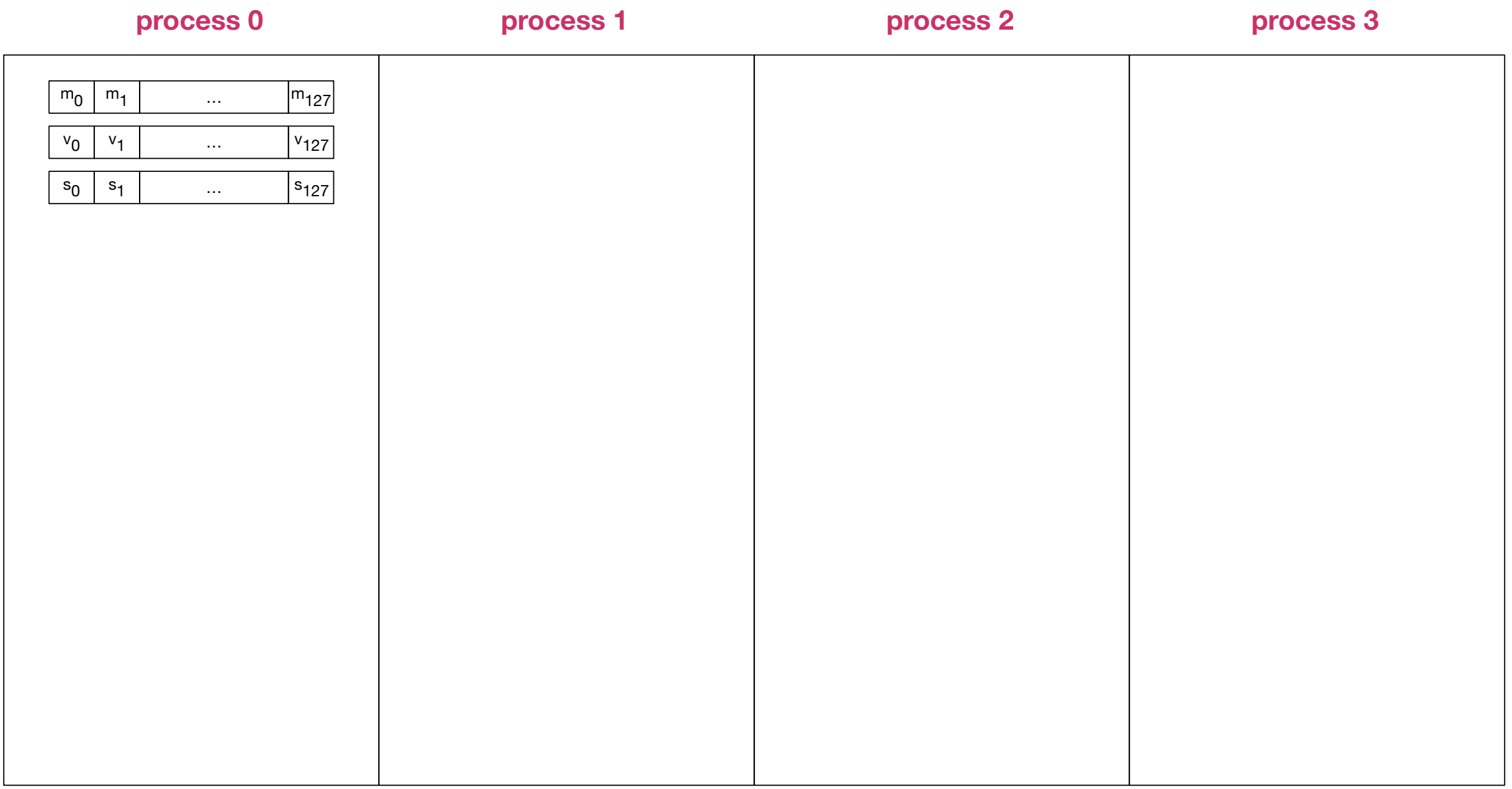
- The parallelization with Pthreads works very similar to OpenMP with two main differences
- 1. Barriers
  - not all Pthreads implementations provide barriers which is needed after the end of inner loops
  - Hence, if no barrier is available we need to either join and re-spawn the threads or use a condition variable
- 2. Loop parallelization
  - due to the lack of a “parallel for”-like operation in Pthreads the assignment of loop iterations to threads must be coded explicitly

code: cf. implementation provided by Pacheo

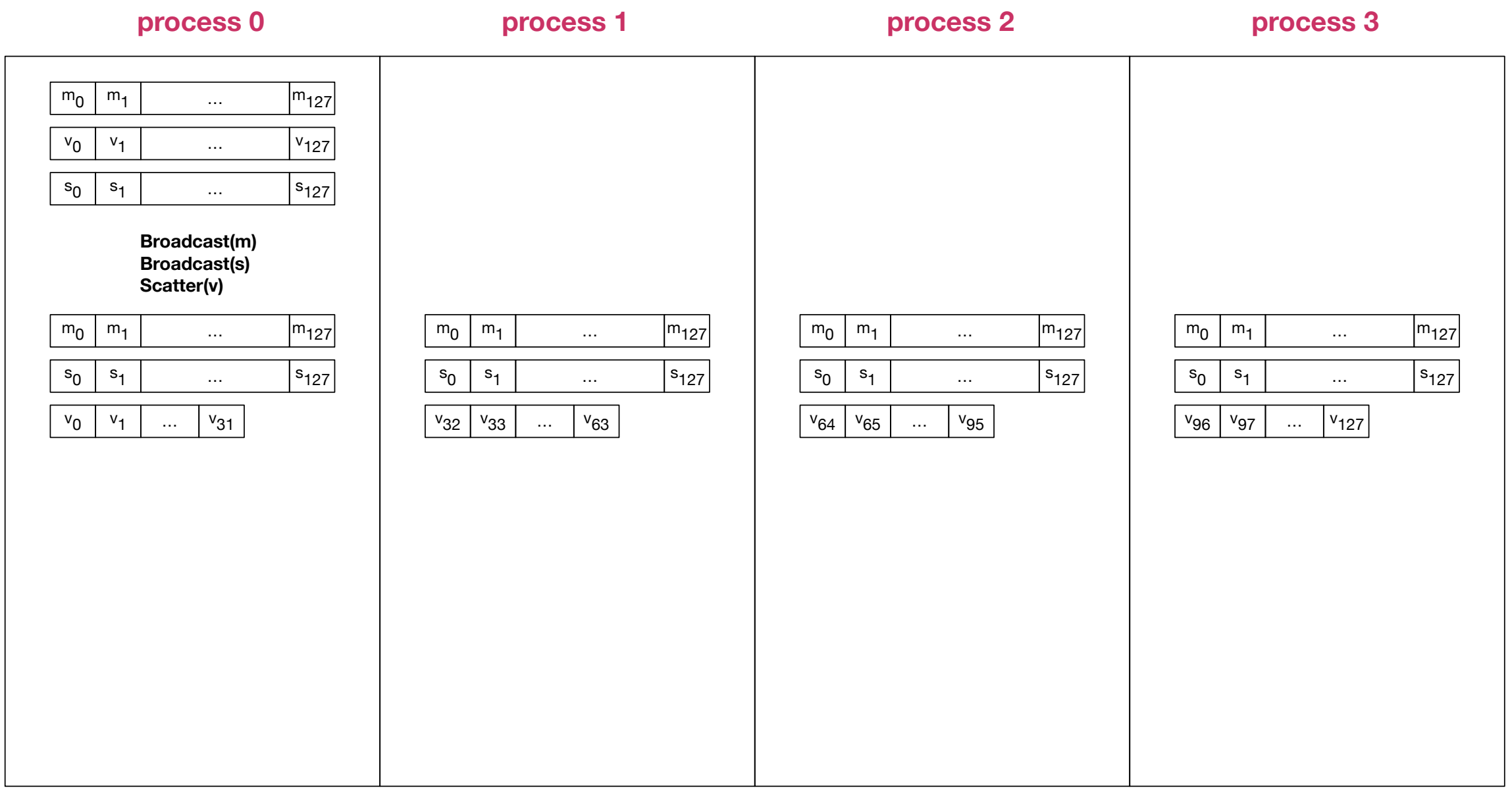
Parallelizing the Basic Solver Using MPI

- Basic parallelization of N-Body code with MPI is fairly straight-forward
- For computing new position of a particle the following data is needed
  - previous position and velocity of particle
  - positions and masses of all other particles
- Strategy
  - assign each process an equal share of particles
  - keep copy of all data required to compute forces for assigned particles in each process
  - compute forces, velocities and new positions
  - re-distribute positions at end of time step with MPI_Allgather

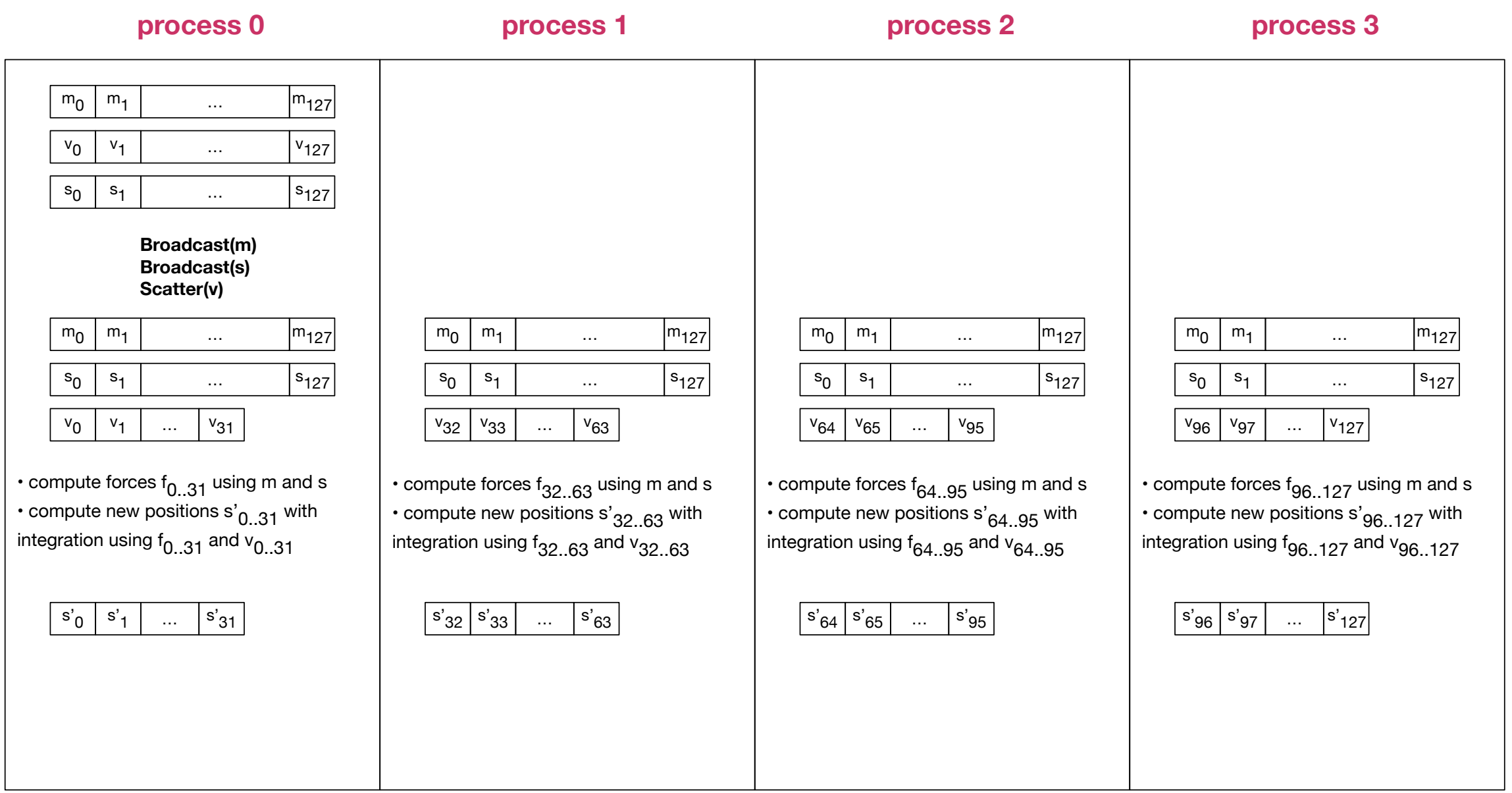
Parallelizing the Basic Solver Using MPI (2)



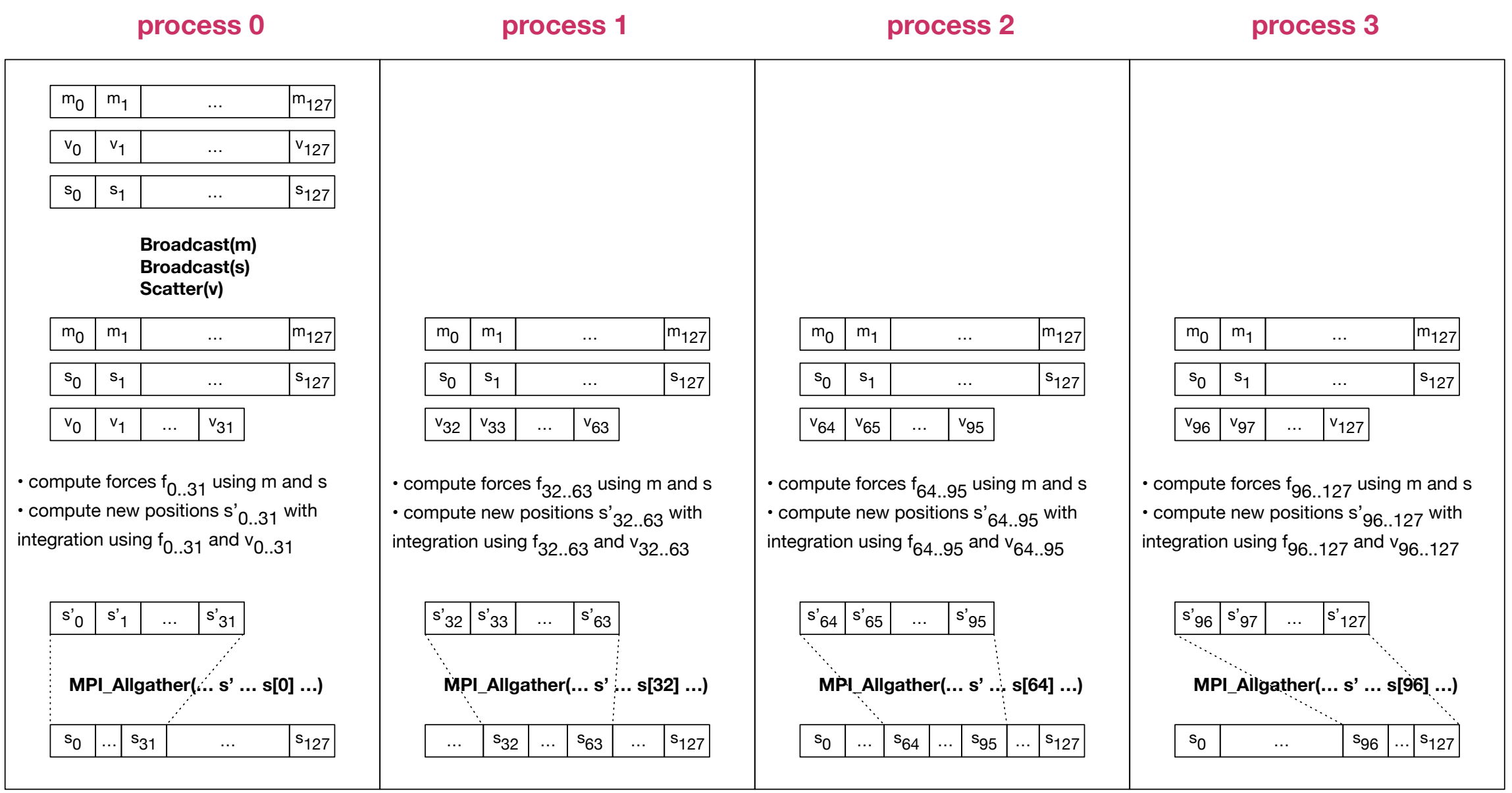
Parallelizing the Basic Solver Using MPI (2)



Parallelizing the Basic Solver Using MPI (2)



Parallelizing the Basic Solver Using MPI (2)



Data Structures for Basic Solver Using MPI (1)

- Array of Structs

```
struct particle_t {
  double mass;
  double pos_x, pos_y;
  double v_x, v_y;
};
particle_t particles[N];
```

  - collect all information about particles in single data structure
  - can be expressed as MPI derived data type
  - can be communicated with single MPI transfer
  - communication of derived data types can be slower (marshalling MPI message)

- Flat Arrays

```
double mass[N];
double pos_x[N], pos_y[N];
double v_x[N], v_y[N];
```

  - problem data scattered over multiple arrays
  - use native MPI data types
  - communication requires several MPI transfers (one per array)
  - communicating basic MPI types is fast (simple marshalling)
  - more flexible, allows to communicate just required arrays instead of whole structure

Data Structures for Basic Solver Using MPI (2)

- Choices in Paceo's implementation
- Each rank stores
  - masses for all particles (immutable data, prevent retransmission)
  - positions of all particles (enables to compute all forces)
  - velocities and new positions for owned particles
- Data stored as simple arrays of tuples
  - position and velocity are vectors with 2 components (x, y)
  - definition of derived MPI data type `vect_mpi_t` for tuples (vector of two doubles)
- Tradeoffs
  - pro: simple implementation
  - con: duplication of data, (masses, positions)
  - acceptable solution for small problems, but for large problems an implementation with less redundant data storage is required (see Ring Buffer scheme, discussed later)

Pseudo-Code for the MPI Version of the Basic N-Body Solver

```
1   Get input data;
2   for each timestep {
3      if (timestep output)
4         Print positions and velocities of particles;
5      for each local particle loc_q
6         Compute total force on loc_q;
7      for each local particle loc_q
8         Compute position and velocity of loc_q;
9      Allgather local positions into global pos array;
10  }
11  Print positions and velocities of particles;
```

Pseudo-Code for Input and Output

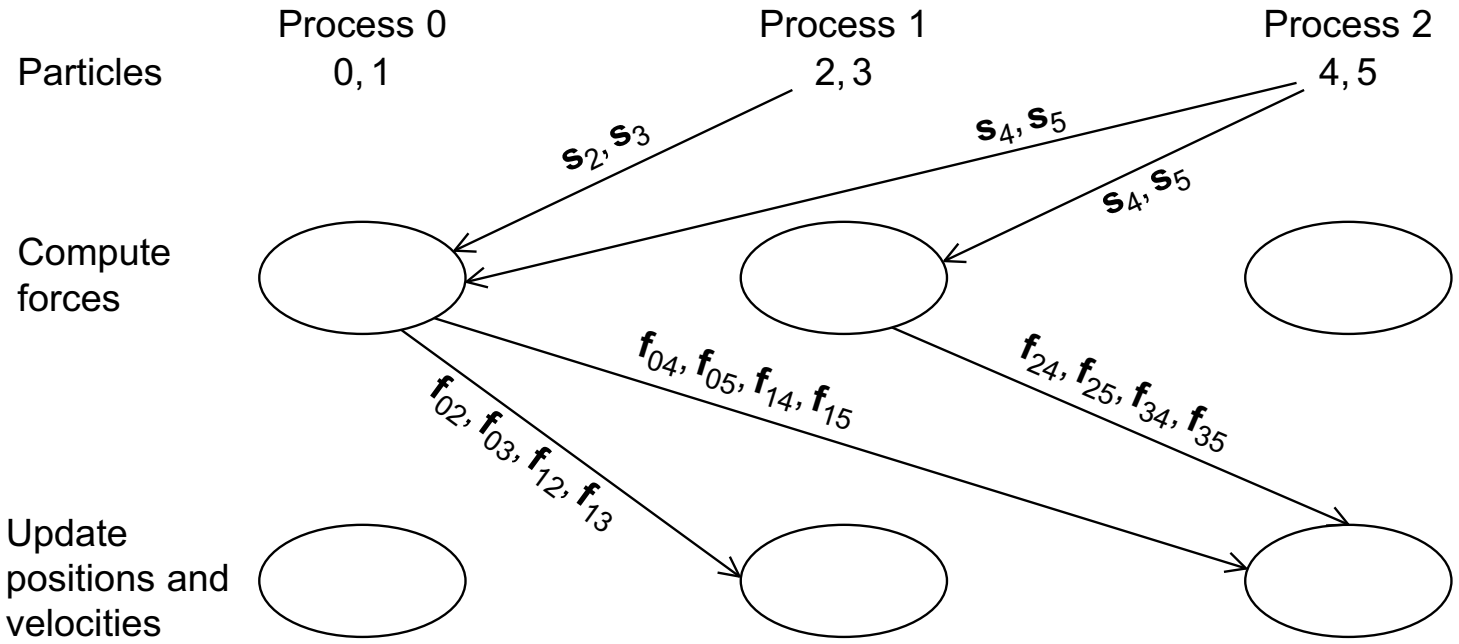
```
if (my_rank == 0) {
    for each particle
        Read masses[particle], pos[particle], vel[particle];
}
MPI_Bcast(masses, n, MPI_DOUBLE, 0, comm);
MPI_Bcast(pos, n, vect_mpi_t 0, comm);
MPI_Scatter(vel, loc_n, vect_mpi_t, loc_vel, loc_n, vect_mpi_t, 0, comm);
```

Input / Distribute data to processes

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

Output

MPI Implementation of a Reduced N-Body Solver

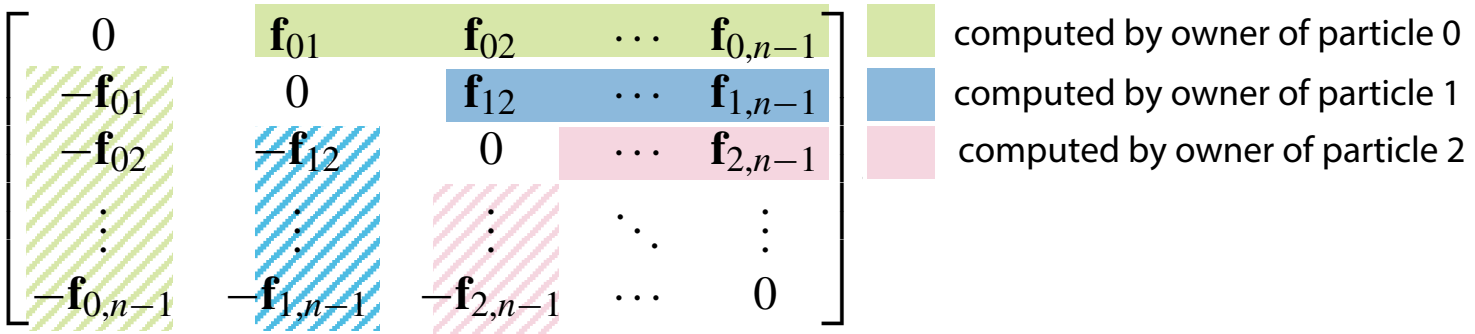


- Difficult and cumbersome to implement
- Irregular communication
  - each process must: 1) gather subset of positions; 2) compute forces; 3) scatter forces to processes that need them
- Load balancing further complicates implementation

MPI Implementation with Ring Pass

- Objective: Support simulations with very high particle count
  - avoid redundant data storage and computations
  - simplify communication scheme
  - find different tradeoff between storage, computation and communication
- Approach
  - each process owns a subset of particles and is responsible for computing and accumulating the corresponding forces in the upper triangle matrix (actio)
  - the counter-acting force (reactio) are also aggregated but not stored locally but communicated to the next process
  - i.e. each process participates in a ring communication scheme
    - receives positions s, masses m and partial forces f_i^- acting on these particles
    - uses additional particle information to compute additional (owned) forces f_i^+ and updates the partial forces f_i^- acting on the received particles
    - passes the information about particle position and partial forces to the next process in the ring
  - after the particle information has passed around the full ring once, process updates s, v, and a for owned particles

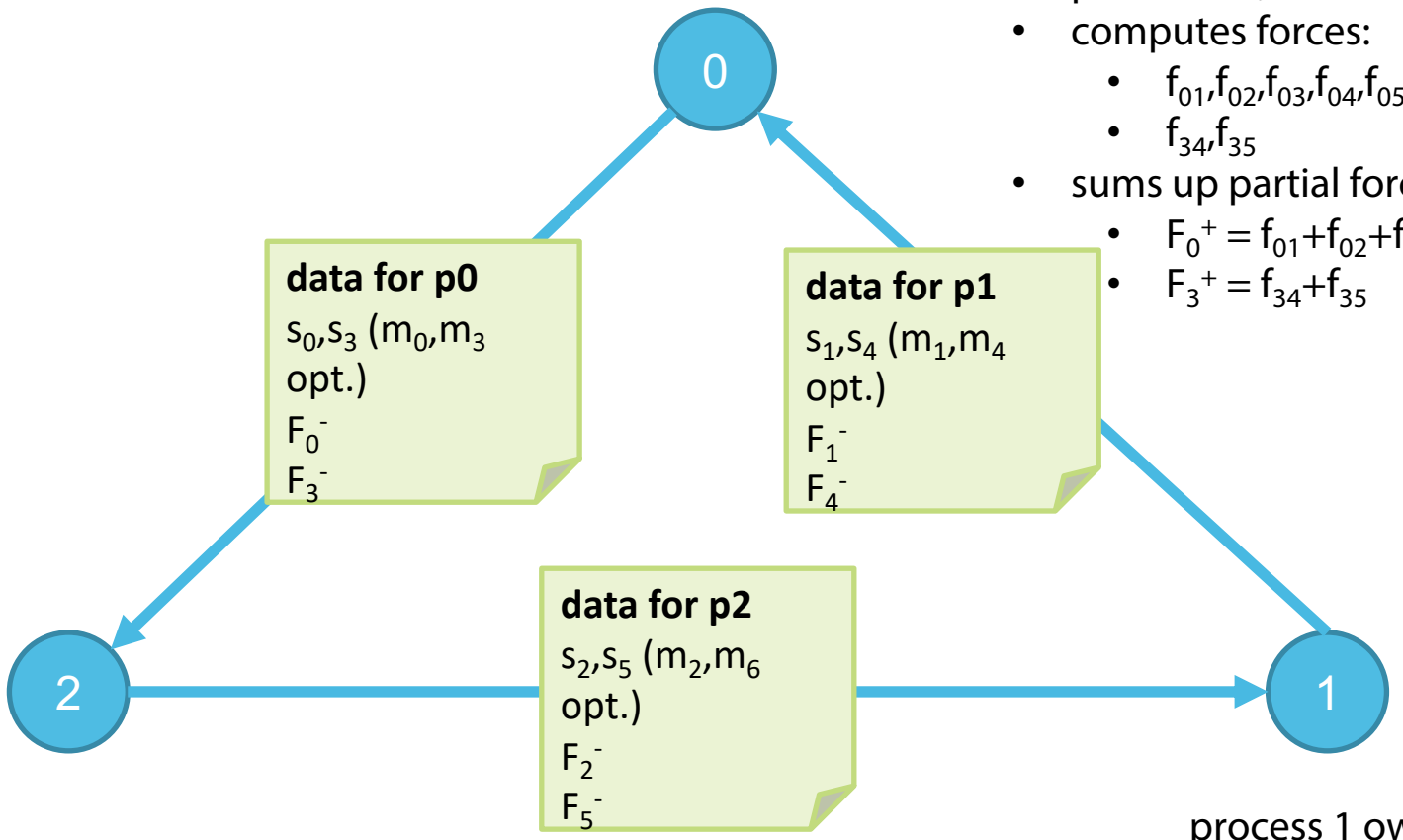
Ownership of Particles and Forces



$$F_2 = F_2^- + F_2^+ = (-f_{02} - f_{12}) + (f_{23} + f_{24} + f_{25})$$

- Example
  - 3 processes, 6 particles, cyclic partitioning
  - process 0
    - owns s0 and s3
    - computes forces $f_{01}, f_{02}, f_{03}, f_{04}, f_{05}, f_{34}, f_{35}$
    - sums up owned forces F_0^+, F_3^+
    - contributes to not-owned forces F_1^-, F_2^-, F_4^-, F_5^-

Ring Pass Scheme (1)



process 0 ownership
- particles: 0,3
- computes forces:
  - $f_{01},f_{02},f_{03},f_{04},f_{05}$
  - f_{34},f_{35}
- sums up partial forces:
  - $F_0^+ = f_{01}+f_{02}+f_{03}+f_{04}+f_{05}$
  - $F_3^+ = f_{34}+f_{35}$

process 1 ownership
- particles: 1,4
- …

process 2 ownership
- particles: 2,5
- …

Ring Pass Scheme (2)

- Algorithm for each time step

```
receive message from neighbor
if message origin != my_rank
    for each owned force f_{m,n} that can be computed with local data and received message
        compute f_{m,n}
            update local partial force F_m^+ = F_m^+  + f_{m,n}
            update partial force in received message F_n^- = F_n^- - f_{m,n}
    pass updated message to neighbor
else
    for each owned particle m
        F_m = F_m^+ + F_m^-
        update a(t), v(t), s(t)
endif
```

Performance of the OpenMP and MPI N-Body Solvers

Table 6.5 Performance of the MPI *n*-Body Solvers (times in seconds)

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

Table 6.6 Run-Times for OpenMP and MPI *n*-Body Solvers (times 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

Concluding Remarks

- N-Body problems are used in many areas of science
- This lecture showed very simple, direct solvers
  - $O(n^2)$ in numbers of particles
  - simple Euler integration
- A lot of progress has bene made in N-Body problems
  - methods with lower complexity for computing force fields, e.g. Barnes-Hut, Fast Multipole
  - better numerical integration, e.g. Runge-Kutta

Acknowledgements

- Peter S. Pacheco / Elsevier
  - for providing the lecture slides on which this presentation is based

Change log

- 1.2.0 (2018-01-06)
  - adapt to new template
  - heavily revised description of MPI implementations
- 1.0.1 (2017-02-03)
  - minor corrections
- 1.0.0 (2017-02-03)
  - initial version of slides