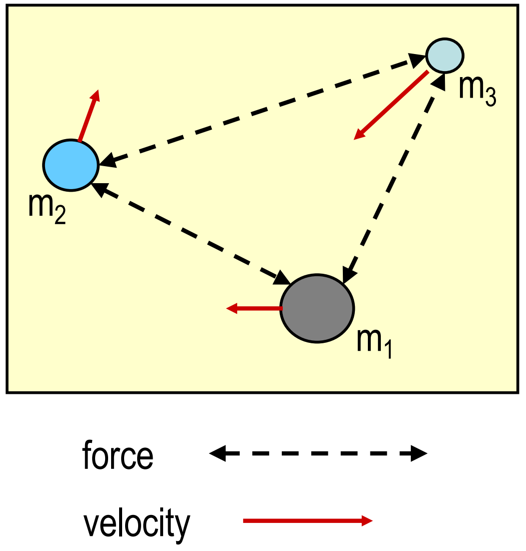
N-body problem

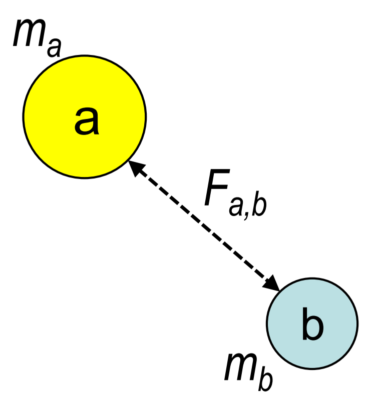
- N-body simulations are a class of computational problems where we calculate the effect of a force between *N* objects (or bodies)
- The problem is to calculate the positions and movements of a number of bodies in space as time advances
  - the bodies are affected by all other bodies via gravitation
  - long-range interactions: each object affects all other objects
- Here we will only consider the two-dimensional case
  - extension to three dimensions is straight forward
- We will only consider gravitational force
  - attraction between two bodies
  - can also use the same methods for other types of forces, for instance electrostatic attraction or repulsion

Problem description

- *We have N* bodies in 2-dimensional space
  - bodies are treated as point masses
  - shape and size does not affect behaviour
- Each body is described by its
  - mass *m*
  - position $X = (x_x, x_y)$
  - velocity $V = (v_x, v_y)$
    the rate of change in position over time, $V = dX/dt$
  - acceleration $A = (a_x, a_y)$
    the rate of change in velocity over time, $A = dV/dt$
- As a body is affected by a force, its velocity changes
  - Newton's laws describe how bodies in space affect each other with gravitation



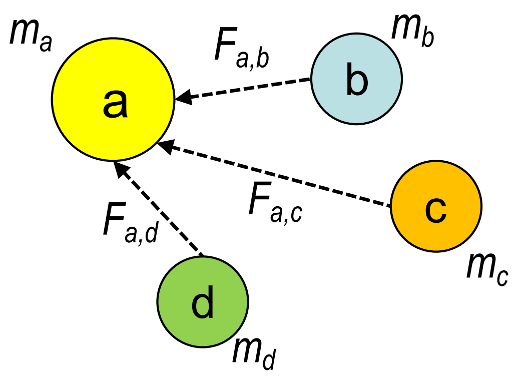
Laws of gravitation



- Newton's laws
  - two bodies *a* and *b* with masses m_a and m_b
  - positions are $X_a = (x_a, y_a)$ and $X_b = (x_b, y_b)$
  - gravitational force on *a* caused by *b*
    - $F_{a,b} = \frac{G m_a m_b}{r^2} \quad \frac{X_b - X_a}{r}$ where *G* is the gravitational constant, G = 6.67259e-11
    - and $r = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2}$ is the distance between *a* and *b*
  - forces are vectors with *x*- and *y*-components $F = (f_x, f_y)$
  - pairwise forces are symmetric but of opposite direction: $F_{a,b} = -F_{b,a}$

Laws of gravitation (cont.)

- The total force on a body *a* is the sum of the pairwise forces from all the other bodies
  - $$F_a = \sum_{j=0, j\neq a}^{N-1} F_{a,j} = G\, m_a \sum_{j=0, j\neq a}^{N-1} m_j \left[\frac{X_j - X_a}{r_{a,j}^3}\right]$$



- The force on a body affects its motion according to Newton's second law $F = m * a$
  - $a = F/m$ (acceleration is force divided by mass)
- Given the current position of a body (x_a, y_a) and the acceleration we can compute the velocity and position of the body in the next time step

Discrete solution

- We divide the time into short time intervals of length h (often denoted Δt)
  - starting from an initial state at time t_0, we calculate the position and velocity for each body at times t_1, t_2, t_3, ...
  - if the current timestep is t_i then the next timestep is $t_{i+1} = t_i + h$
- $O(N^2)$ algorithm

```
– for each time interval
    for each body b
        for all other bodies c
            calculate the force on b caused by c
        calculate new velocity for b
        calculate new position for b
```

- The time interval h has to be short enough to give an accurate solution
  - the amount of computation increases with a shorter interval

Discrete formulation

- For a body with mass *m* we compute the force *F* affecting it at time *t*
  - sum of the forces from all other bodies
- Then we compute the velocity and position of the body for the next timestep, i.e. at time *t+h*
  - new velocity is $V^{t+1} = V^t + A^t h$ where $A^t = \frac{F}{m}$
  - new position is $X^{t+1} = X^t + V^t h$
- When the bodies move to new positions, the forces change and the computation has to be repeated
- Called Euler's method

Sequential solution

- A straightforward sequential solution with Euler's method

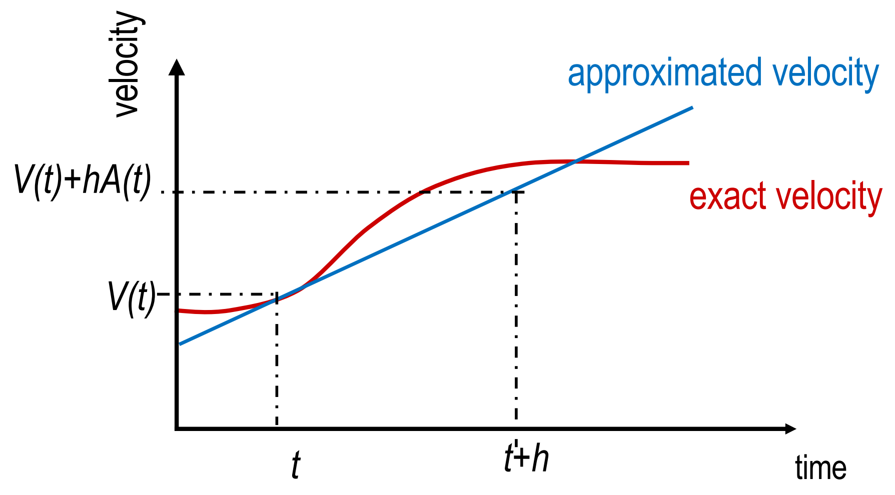
```
for (t=0; t<maxt; t++) {           /* For each timestep */
     for (i=0; i<nbodies; i++) {       /* For each body */
          F = Force(i);                    /* Force on i */
          V[i] = V_old[i]+F*deltat/m;      /* New velocity */
          X[i] = X_old[i]+V_old[i]*deltat; /* New position */
     }

  copy V to V_old;
  copy X to X_old;
}
```

- *F*, *V* and *X* are vectors in the 2-dimensional space
  - vector multiplication and addition
- We keep the values from previous time step in the arrays *V_old* and *X_old*

Problem with Euler's method

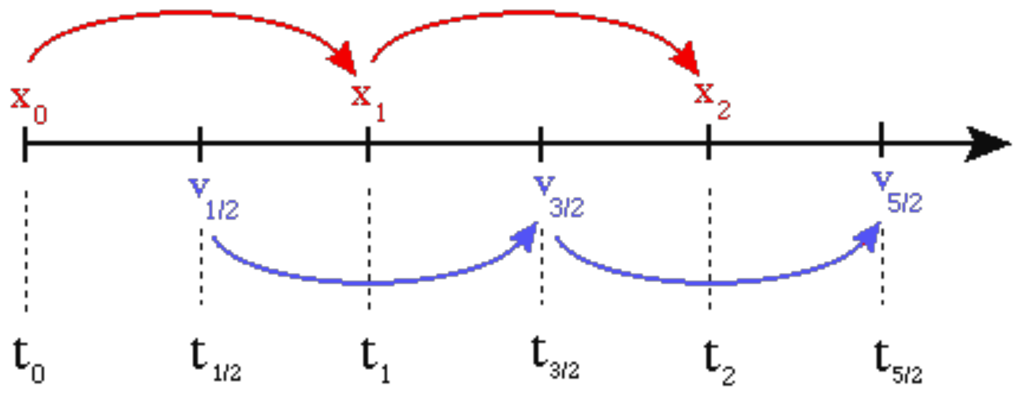
- In Euler's method, we assume that the acceleration is constant during the whole time interval
  - we use the acceleration at time *t* to calculate the new velocity at time *t*+*h*
- Acceleration is not constant during a time interval
  - when a body gets closer to another body, its acceleration increases
  - our approximation is based on the velocity at time *t*

- Have to use a small time step to get reliable results



The Leapfrog method

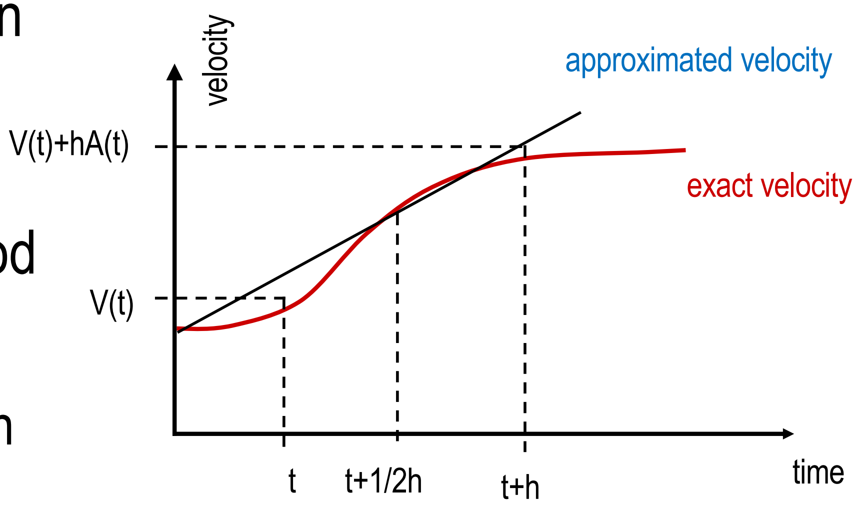
- In the Leapfrog method we use the *mid-point* of the interval to calculate the approximation of the velocity during the interval
  - this is used to calculate the positions at the next point in time
- Velocities and positions are not updated at the same point of time
  - update positions at the *beginning* of each time step and velocities at the *middle* of the time steps
- Positions are updated at times *h, 2h, 3h, …*
  - $X_{i+1} = X_i + h^*V_{i+1/2}$ for $i=0,1, 2, ….$
- Velocities are updated at times *1/2h*, *t+1/2h, 2t+1/2h, 3+1/2h, …*
  - $V_{i+1/2} = V(t+1/2h)$



Properties of the Leapfrog method

- Using the velocity at the midpoint of the interval gives a better approximation than the velocity at the beginning (or end) of the interval

- No more complicated to implement than the Euler scheme
- The initial velociy at $V_{1/2}$ can for instance be calculated by Eulers method
- The leapfrog method is time reversible
  - starting from any state at time t_i we can calculate backwards in time to t_0
- The leapfrog method is a second order approximation
  - the approximations of the positions have an accuracy of $O(\Delta t^2)$



Implementation of the Leapfrog method

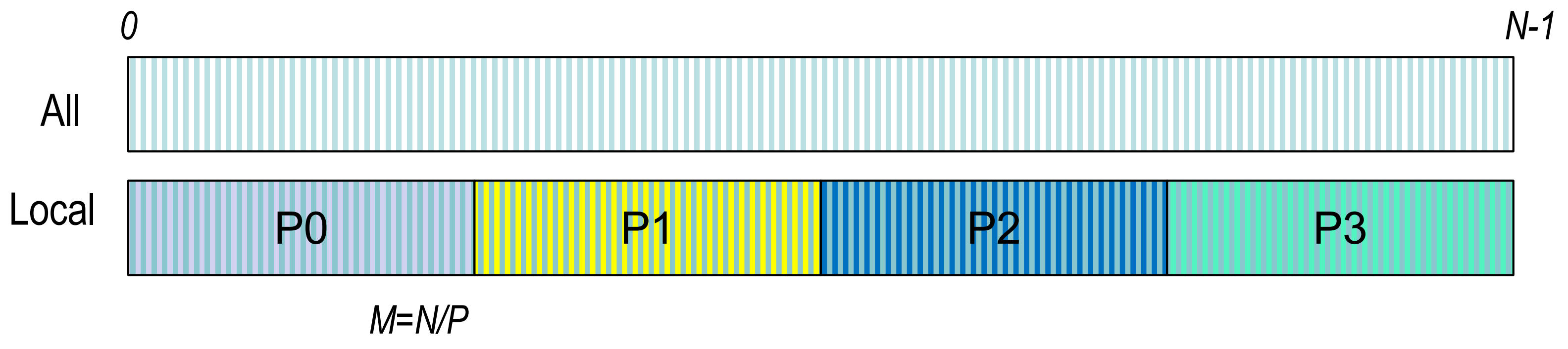
- A sequential implementation of the Leapfrog method is

```
set initial values of V_old to velocities at time 0.5*deltat

for (t=0; t<maxt; t++) {          /* For each timestep */
  for (i=0; i<nbodies; i++) {         /* For each body */
    X[i] = X_old[i]+V_old[i]*deltat; /* New position */
    F = Force(i);                        /* Force on i */
    V[i] = V_old[i]+F*deltat/m;      /* New velocity */
  }

  copy V to V_old;
  copy X to X_old;
}
```

- We now first update positions, then acceleration and velocities

Parallel *N*-body simulation: domain decomposition

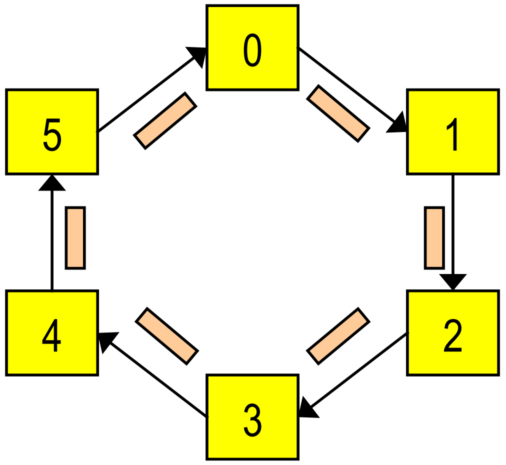
- *N* bodies, *P* processes
  - we divide the bodies evenly among the processes
  - *M* = *N/P* local bodies per process



- A process computes the force that affects it *local* bodies
  - for this, it needs information about all the other bodies
- When a process has updated the values for its own bodies, the results are communicated to all processes
  - the updated positions, velocities and acceleration are needed in the next time step

Communication structure

- There are different ways to implement the distributed computation and communication
- One possible solution is to connect the processes n a ring
  - circulate the local bodies of each process through the ring
  - after *P-1* steps, all bodies have visited all processes and all force interactions have been calculated



- In a system with point-to-point connected processors, communication can proceed simultaneously between all processes
  - all processes have the same amount of work, both computation and communication
- Another alternative solution is to do the exchange of data with collective communication

Algorithm structure

- Algorithm for a ring-based solution

```
– for each time interval
     update positions of local bodies
     calculate pairwise force between local bodies
     for all other processes in the ring
        send local bodies to next process
        receive bodies from previous process
        compute pairwise force between the received and
                the local bodies
     update velocities of local bodies
```

- The force-values are accumulate with contributions of particles from the other processes
  - all these are added together to get the total forces

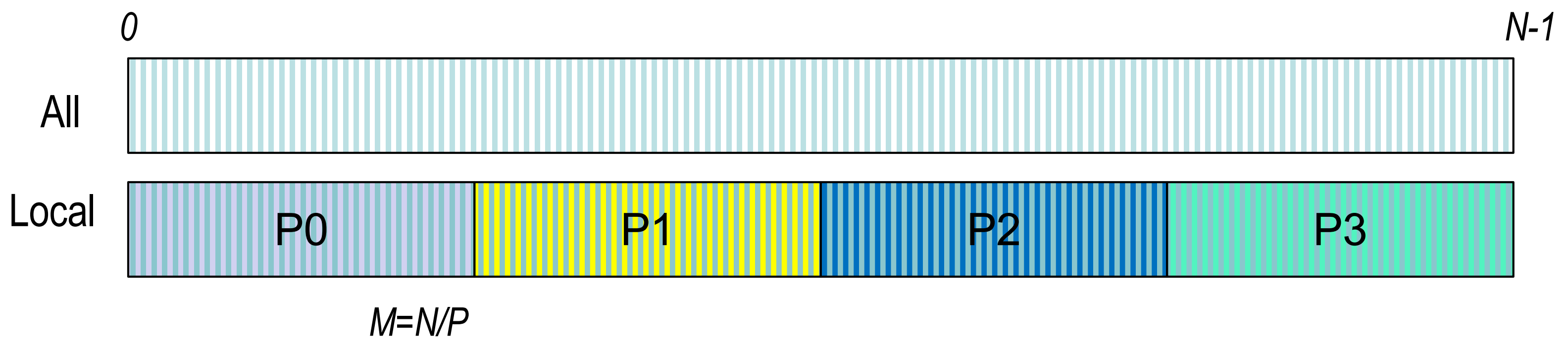
Ring algorithm for process *k*

- Each process has neighbours *next* and *previous*
- Local bodies are stored in an array *B*, non-local bodies are received in an array *inbuf*
- *ComputeForce* computes forces between two sets of bodies

```
next     = (k==P-1) ? 0 : k+1;   /* Next and previous   */
previous = (k==0) ? P-1 : k-1;   /* process in the ring */

for (t=0; t<maxt; t++) {         /* For each time step */
  update positions of local particles
  ComputeForce(B, B);  /* Forces between local bodies */
  for (step=1; step<P; step++) { /* Loop through ring */
    send B to next;
    receive inbuf from previous;    /* Forces between */
    ComputeForce(B, inbuf);        /* nonlocal bodies */
  }
  update velocities for particles
}
```

Implementation with collective communication

- We can also use collective communication for the communication
- Use MPI collective communication to exchange information about the bodies to all processes
  - the communication does not need to do the data exchange in *P-1* steps, but can directly send the needed data to all processes
  - the implementation is otherwise similar, but the communication structure becomes more simple


0
N-1
All
Local
P0
P1
P2
P3
M=N/P